Design Methodology and Tools

8

8.1 Introduction

The manner in which you go about designing a particular system, chip, or circuit can have a profound impact on both the effort expended and the outcome of the design. IC designers have developed and adapted strategies from allied disciplines such as software engineering to form a cohesive set of principles to increase the likelihood of timely, successful designs. We will explore these principles in this chapter. While the broad principles of design have not changed in decades, the details of design styles and tools have evolved along with advances in technology and increasing levels of productivity. This chapter represents current CMOS design methods and provides an overview of a complex subject that could fill many books on its own. We encourage you to actively monitor the companies discussed and literature cited in the chapter to track the latest developments in this rapidly changing field.

As introduced in Section 1.6, an integrated circuit can be described in terms of three *domains*: (1) the *behavioral* domain, (2) the *structural* domain, and (3) the *physical* domain. The behavioral domain specifies what we wish to accomplish with a system. For instance, at the highest level, we might want to build an ultra-low-power radio for a distributed sensor network. The structural domain specifies the interconnection of components required to achieve the behavior we desire. Again, by way of example, our sensor radio might require a sensor, a radio transceiver, a processor and memory (with software), and a power source connected in a particular manner. Finally, the physical domain specifies how to arrange the components in order to connect them, which in turn allows the required behavior. Our example might start with the specification for an enclosure to hold the device, followed by a succession of physical drawings or specifications that may culminate in descriptions of geometry to be used to define a chip. Design flows from behavior to structure and ultimately to a physical implementation via a set of manual or automated transformations. At each transformation, the correctness of the transformation is tested by comparing the pre- and post-transformation design. For instance, if a power level is specified in the original behavioral description of the sensor radio, a test is run on the design in the structural domain with feedback from the physical domain to ensure this design goal is met.

In each of these domains there are a number of design options that can be selected to solve a particular problem. For instance, at the behavioral level, we can choose the wireless standard and the format in which data is transmitted by the sensor radio. In the structural

TSMC-1012
U.S. Patent 8,884,373

domain, we can select which particular circuit style, logic family, or clocking strategy to use. At the physical level, we have many options about how the circuit is implemented in terms of chips, boards, and enclosures. These domains can further be hierarchically divided into different levels of *design abstraction*. Classically, these have included the following for digital chips:

- Architectural or functional level
- Logic or Register Transfer level (RTL)
- Circuit level

For analog and RF circuits, the block diagram level replaces the logic level.

The relationship between description domains and levels of abstraction is elegantly shown by the *Gajski-Kuhn Y chart* in Figure 8.1. In this diagram, the three radial lines represent the behavioral, structural, and physical domains. Along each line are enumerated types of objects in that domain. In the behavioral domain, we have represented conventional software and hardware description language categories. As we move out along any of the radial axes, the increasing level of design abstraction is able to represent greater

FIG 8.1 Gajski-Kuhn Y chart

complexity. Thus, in the behavioral domain, the lowest level of abstraction is an instruction or a statement in software or HDL descriptions, respectively. Circles represent levels of similar design abstraction: the architectural, RTL, logic, and circuit levels. The particular abstraction levels and design objects may differ slightly depending on the design method.

In this chapter, we will examine how to transform a description from one domain into another while maintaining the integrity of the design. It is only in this way that we can start with a behavior and successfully build a product.

We begin by discussing some of the guiding principles that apply to most engineering projects. Then we survey the various design strategies available to the CMOS IC designer; these range from rapid prototyping or small-volume approaches to those suitable for high-volume digital, analog, or RF design. We then examine the economics of design, which can guide us to the right selection of an implementation strategy, and review documentation requirements. In conclusion, we look at the reasons for the gap in performance between custom and ASIC design methods.

8.2 Structured Design Strategies

The viability of an IC is in large part affected by the productivity that can be brought to bear on the design. This in turn depends on the efficiency with which the design can be converted from concept to architecture, to logic and memory, to circuit, and ultimately to physical layout. A good VLSI design system should provide for consistent descriptions in all three description domains (behavioral, structural, and physical) and at all relevant levels of abstraction (e.g., architecture, RTL/block, logic, circuit). The means by which this is accomplished can be measured in various terms that differ in importance based on the application. These parameters can be summarized in terms of:

* Performance—speed, power, function, flexibility
* Size of die (hence, cost of die)
* Time to design (hence, cost of engineering and schedule)
* Ease of verification, test generation, and testability (hence, cost of engineering and schedule)

Design is a continuous tradeoff to achieve adequate results for all of the above parameters. As such, the tools and methodologies used for a particular chip will be a function of these parameters. Certain end results have to be met (i.e., the chip must conform to certain performance specifications), but other constraints may depend on economics (i.e., size of die affecting yield) or even subjectivity (i.e., what one designer finds easy, another might find incomprehensible).

Given that the process of designing a system on silicon is complicated, the role of good VLSI-design aids is to reduce this complexity, increase productivity, and assure the

designer of a working product. A good method of simplifying the approach to a design is by the use of constraints and abstractions. By using constraints, the tool designer has some hope of automating procedures and taking a lot of the "legwork" (effort) out of a design. By using abstractions, the designer can collapse details and arrive at a simpler object to handle.

In this chapter, we will examine design methodologies that allow a variation in the freedom available in the design strategy. The choice, assuming all styles are equally available, should be entirely economic. According to function, suitable design methods are selected. Following these steps, the required chip cost is estimated and the quickest means of achieving that chip should be chosen. We will focus on structured approaches to design since they offer the most appropriate method of dealing with design complexity.

The successful implementation of almost any integrated circuit requires attention to the details of the engineering design process. Over the years, a number of structured design techniques have been developed to deal with complex hardware and software projects. Not surprisingly, the techniques have a great deal of commonality. Rigorous application of these techniques can drastically alter the amount of effort that has to be expended on a given project and also, in all likelihood, the chances of successful conclusion.

8.2.1 A Software Radio—A System Example

To guide you through the process of structured design, we will use as an example a hypothetical "software radio," as illustrated in Figure 8.2. This device is used to transmit and receive radio frequency (RF) signals. Information is modulated onto an RF *carrier* to transmit data, voice, or video. The RF carrier is demodulated to receive information. An ideal software radio could receive any frequency and decode or encode any type of information at any data rate. Some day, this might be possible, but given the limitations of current processes there are some bounds. To understand the impact of design methods on system solutions, we will examine the software radio in more detail. This system will then form the basis for discussion about structured approaches to design.

Figure 8.3 illustrates a typical transmit path for a generic radio transmitter, which is called an *IQ modulator*. An input data stream is encoded into *inphase* (*I*) and *quadrature* (*Q*) signals. The *I* and *Q* represent signal amplitudes of a (voltage) vector that vary instantaneously in time as shown in the bottom of Figure 8.3. For appropriate *I* and *Q* values, any form of modulated carrier can be synthesized. *I* is multiplied by an oscillator (sine)

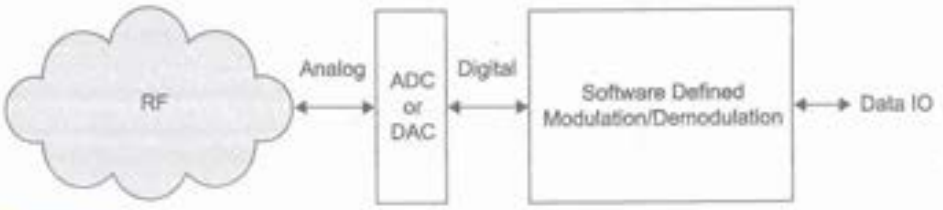


FIG 8.2 Software radio block diagram

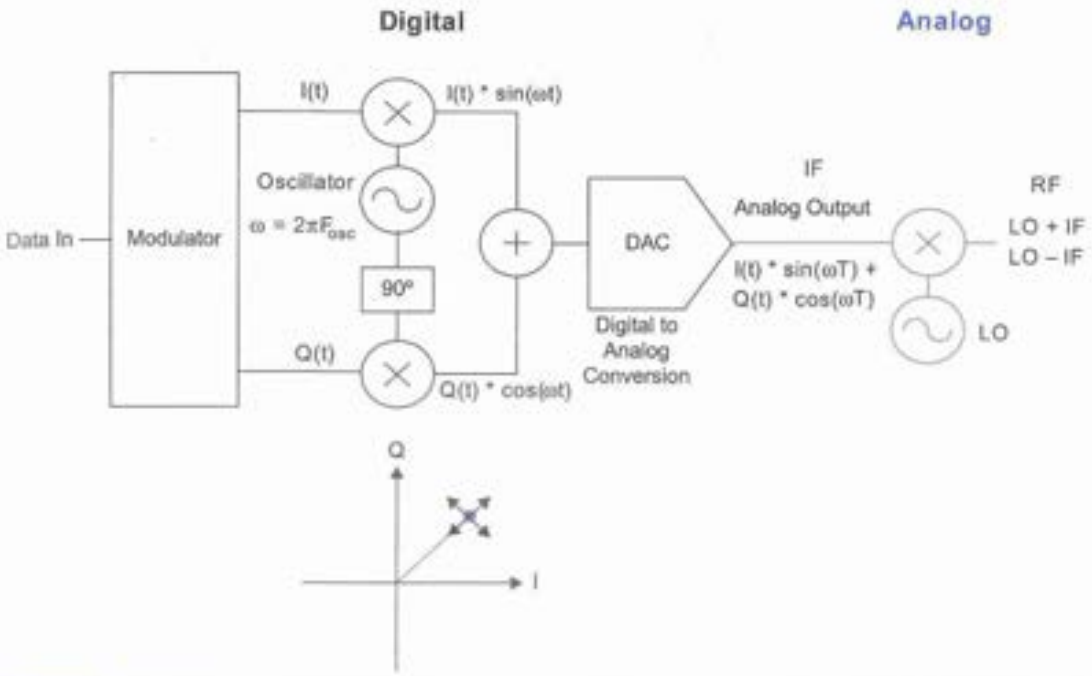


FIG 8.3 Software radio transmit path

operating at a frequency of F_{osc}. The quadrature (Q) signal is multiplied by the cosine of this frequency. The resultant signals are summed and passed to a digital-to-analog converter (DAC). In the design shown, this generates what we term an *Intermediate Frequency* or IF.

Typical IQ constellations are shown in Figure 8.4. *Amplitude Modulation* (AM), depicted in Figure 8.4(a), varies only in the magnitude of the carrier that varies in accordance with the amplitude of the modulation waveform. This is shown as a signal with an arbitrary phase angle (which we don't care about) and a vector that travels from the origin to a point on a circle that represents the maximum value of the carrier. In the case of an AM radio, the carrier frequency might be 800 KHz (in the AM band) and the modulation frequencies range from roughly 300 Hz to 6 KHz (voice and music frequencies). *Phase Modulation* is shown in Figure 8.4(b). Here, the vector travels around the maximum carrier amplitude circle varying the phase angle (δ) as the modulation changes. This is a constant amplitude modulation, which might be used with a carrier frequency of 100 MHz (in the FM broadcast band—we are loosely associating phase modulation with *frequency modulation* (FM) as they are closely related) and could have modulation frequencies of 200 Hz to 20 KHz (hi-fi audio). Finally, Figure 8.4(c) shows *Quadrature Phase Shift Keying* (QPSK) modulation, which is typical of data transmission systems. Two bits of data are encoded onto four phase points as shown in the diagram. A typical carrier frequency

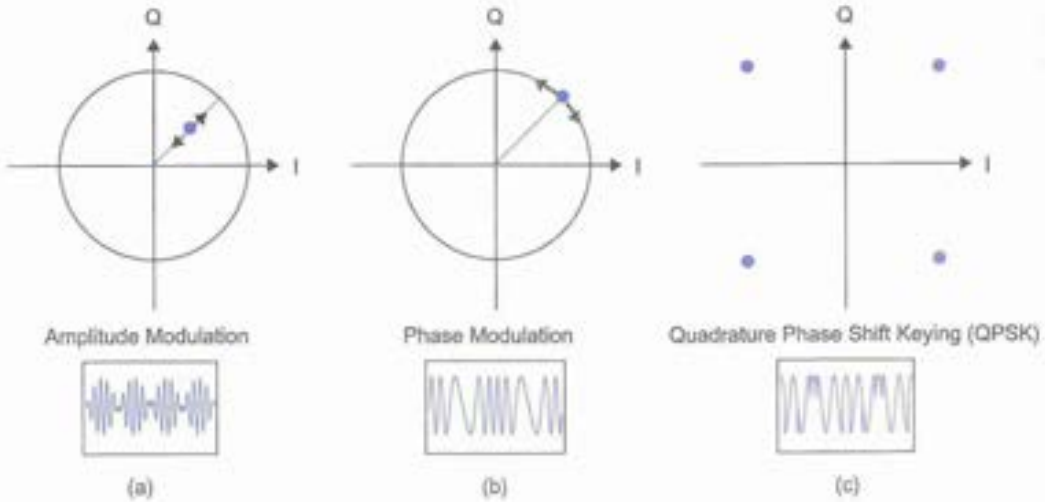


FIG 8.4 Examples of IQ modulation

might be 2.4 GHz in the Industrial Scientific and Medical (ISM) band and the modulation data rate might be 10 Megabits/second.

Clearly, the ranges of carrier and modulation frequencies vary considerably. Generally, for high carrier frequencies, the modulation can be performed at a moderate frequency and then "mixed" up to a higher frequency by analog multiplication. This is completed in the analog domain and is illustrated by the blue components on the right side of Figure 8.3. An analog multiplier (called a *mixer* in RF terminology) takes an analog Local Oscillator (LO) and the Intermediate Frequency (IF) signal that we have generated and produces sum and difference frequencies. (It is also possible to generate the desired RF frequency directly, but in this design we will use an intermediate frequency approach.) Analog bandpass filtering or a slightly more sophisticated mixer can be used to select the mixing component (LO+IF or LO-IF) that we desire. For instance, if we generate a data signal on a 20 MHz IF and mix it with a 2.4 GHz LO, we can generate a 2.402 or 2.398 GHz data signal. This is called *upconversion*.

To complete the software radio, the receive path is shown in Figure 8.5. It is roughly the reverse of the transmit path. As in the transmit case, higher frequencies can be *downconverted* to lower IF frequencies that are suitable for processing by practical ADCs. The RF signal is mixed with the LO and low pass filtered to produce the difference frequency. For example, if a 2.4 GHz LO is mixed with the 2.402 GHz RF signal, the 20 MHz IF signal is restored. An analog-to-digital converter (ADC) converts the modulated IF carrier into a digital stream of data. This data is mixed (multiplied) in the digital domain by an oscillator operating at the IF frequency. After digital low pass filtering (LPF), the original I and Q signals can be reconstructed and passed to a demodulator. For further details on digital radio, consult a communications theory text such as [Haykin00].

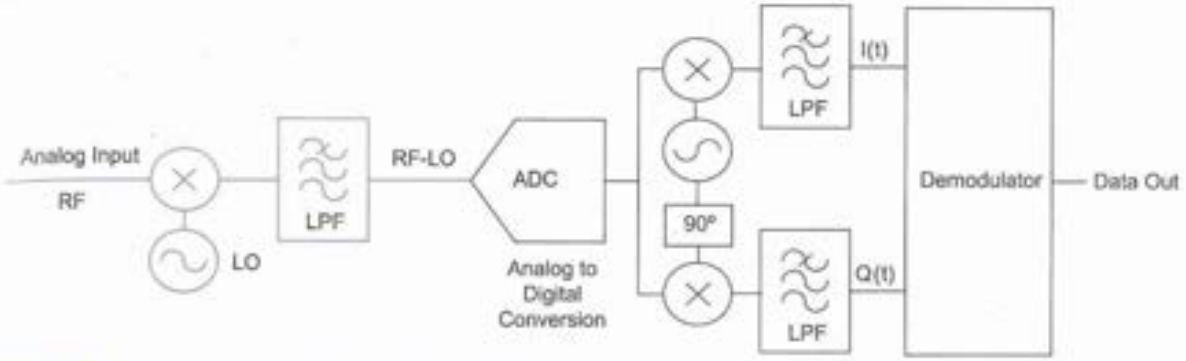


FIG 8.5 Software radio receive path

In summary, we see that multiplication, sine wave generation, and filtering are important for a software radio. While the modulation and demodulation have not been described in detail, operations can include equalization (multiplication), time to frequency conversion (fast Fourier transform), correlation, and other specialized coding operations. In the subsequent sections we will explore the design principles of hierarchy, regularity, modularity, and locality with concrete examples applied to the software radio.

8.2.2 Hierarchy

The use of hierarchy, or "divide and conquer," involves dividing a system into modules, then repeating this process on each module until the complexity of the submodules is at an appropriately comprehensible level of detail. This may entail stopping at a level where a prebuilt component is available for the particular function. The process parallels the software strategy in which large programs are split into smaller and smaller sections until simple subroutines with well-defined behavior and interfaces can be written. In the case of predefined modules, the design task involves using library code intended for the required function. The notion of "parallel hierarchy" can be used to aggregate descriptions in each of the behavioral, structural, and physical domains that represent a design (parallel hierarchy means a hierarchy—not necessarily identical—is used in each domain). Furthermore, equivalency tools can ensure the consistency of each domain. Because these tools can be applied hierarchically, you can progress in verification from the bottom to the top of a design, checking each level of hierarchy where domains are intended to correspond. For instance, a RISC processor core can have an HDL model that describes the behavior of the processor; a gate netlist that describes the type and interconnection of gates required to produce the processor; and a placement and routing description that describes how to physically build the processor in a given process. Later in the chapter, we will see how domain-to-domain comparisons are used to ensure consistency between domains.

Hierarchy allows the use of *virtual components*, soft versions of the more conventional packaged IC. Virtual components are placed into a chip design as pieces of code and come with support documentation such as verification scripts. They can be supplied by an independent *intellectual property* (IP) provider or can be reused from a previous product developed in your organization. Virtual components are discussed further in Section 8.5.7.

Example

The digital operations in the transmit path of the software radio (Figure 8.3) can be performed in software. Hence, a microprocessor can form the basis for the design. In this case, the design might have the hierarchy of a typical microprocessor, as shown in Figure 8.6. At the top level, the microprocessor contains an arithmetic logic unit (ALU), program counter (PC), register file, instruction decoder, and memory. The ALU can be further decomposed into an adder, a Boolean logic unit, and a shifter. The shifter and adder can together perform multiplication. The diagram illustrates how a relatively complex component can be rapidly decomposed into simple components within a few levels of hierarchy. Each level only has a few modules, which aids in the understanding of that level of the hierarchy.

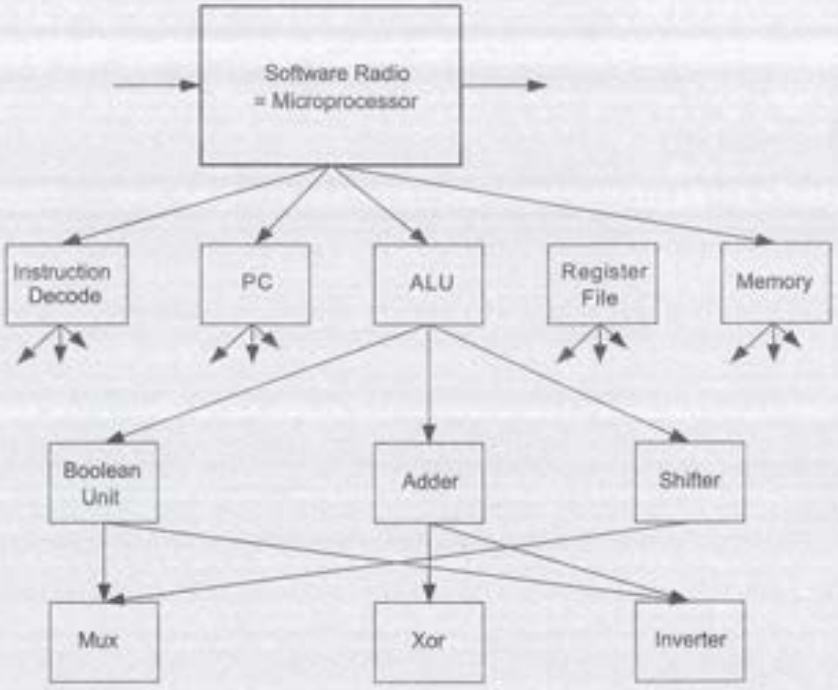


FIG 8.6 Possible hierarchy of software radio using a single microprocessor

Example

We can roughly estimate the performance required in the transmit path by noting that we require at least two multiplications, an addition, and two table lookups (sine and cosine). Another addition would be required to maintain a loop counter. An iterative multiply takes N cycles for an N-bit word, so for a 16-bit word width, the total number of cycles for the steps described would be approximately 16 + 16 + 1 + 2 + 2 + 1 (if table lookups take two clock cycles). This yields a total of roughly 40 clock cycles. For a 100 MHz processor, the fastest we could perform the IQ conversion would be approximately 400 ns, which, according to Nyquist's criteria ($F_{analog_max} = F_{sample}/2$), would be capable of generating a 1.25 MHz IF signal. This is, of course, without any extra processing for modulating the carrier. While we could add another processor, this may be wasteful of area and power, given the operation that has to be performed.

A better approach is to use dedicated hardware for the computationally intensive fixed-function blocks. The trick is to notice that the IQ modulator portion of the software radio transmit and receive path for a given DAC and ADC resolution has a relatively fixed architecture. For the transmit path, the hierarchy shown in Figure 8.7 can be used where the blue sections have been converted to fixed function blocks. This is a relatively safe bet because the IQ upconversion is a generic communications building block. In addition to the multipliers, a device called a *Numerically Controlled Oscillator* (NCO) has been introduced [Lu93, Lu93b, Hwang02]. The NCO, described in detail in the next section, generates sine or cosine waveforms at a speed determined by the delay through an N-bit adder where N is in the range of 16 to 32 for typical NCOs. The

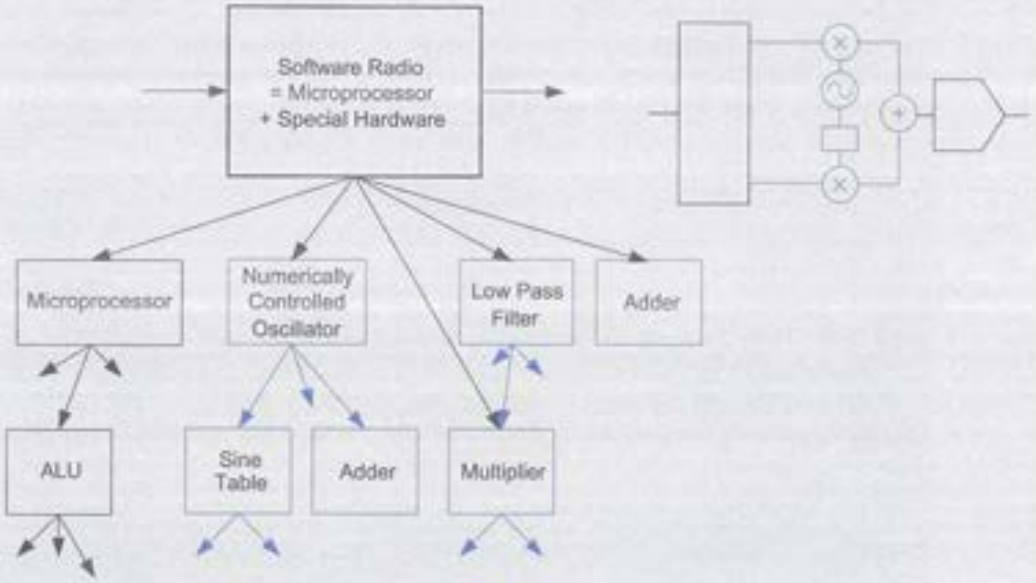


FIG 8.7 Transmit chain with dedicated IQ upconverter

move to dedicated hardware for the IQ upconversion allows the circuit to produce a new value once every clock cycle. If we conservatively say that the arithmetic blocks operate at the same speed that the microprocessor ALU does, then the circuit will now operate at 100 MHz. Taking into account sampling theory, this means that we can generate analog frequencies up to almost 50 MHz with a suitable DAC. The microprocessor now only has to respond at the modulation data rate, providing IQ values to the IQ upconverter.

8.2.3 Regularity

Hierarchy involves dividing a system into a set of submodules. However, hierarchy alone does not solve the complexity problem. For instance, we could repeatedly divide the hierarchy of a design into different submodules but still end up with a large number of different submodules. With regularity as a guide, the designer attempts to divide the hierarchy into a set of similar building blocks. Regularity can exist at all levels of the design hierarchy. At the circuit level, uniformly sized transistors can be used, while at the gate level, a finite library of fixed-height, variable-length logic gates can be used (see later in the chapter for a description of standard cells). At the logic level, parameterized RAMs and ROMs could be used in multiple places. At the architectural level, multiple identical processors can be used to boost performance.

Regularity aids in verification efforts by reducing the number of subcomponents to validate and by allowing formal verification programs (see Section 8.4.1.4) to operate more efficiently. Design reuse depends on the principle of regularity to use the same virtual component in multiple places or products.

Example

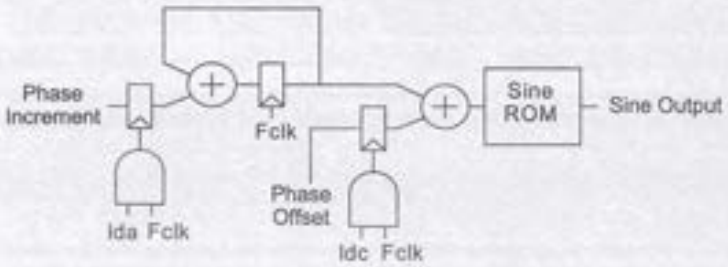
In an example of regularity applied to the software radio, we first look inside two of the blocks used in the designs shown in Figure 8.3 and Figure 8.5 to assess what kinds of functions are required.

The NCO is shown in Figure 8.8(a). It is composed of a registered adder that is incremented every clock cycle by a phase increment register. This implements a phase counter, which is used to step through a ROM lookup table that provides phase-to-amplitude conversion. A phase offset can be added to the phase incrementer to perform phase modulation. With this structure we are able to generate a digital sine wave.

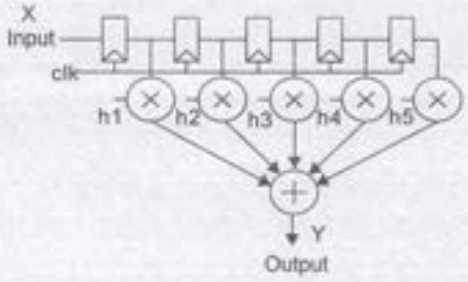
Turning to the low-pass filter shown in Figure 8.5, Figure 8.8(b) shows the structure for a commonly used low-pass filter implementation that is called a *Finite Impulse Response* (FIR) filter [Edwards93, Choi97]. The structure computes the function:

$$Y[n] = \sum X[n-k]b[k] \tag{8.1}$$

where $X[n]$ is the sampled input, $b[k]$ are the filter coefficients that characterize the particular filter, and $Y[n]$ is the output. As the structure indicates, the filter is composed of registers, multipliers, and an adder. Filters are characterized by the number of *taps* (coefficients). More taps yield better filters approaching an ideal "brick wall" filter with steeper cutoff and low ripple. This in turn requires more registers and more multipliers.



(a) Numerically Controlled Oscillator (NCO) Structure



(b) Finite Impulse Response (FIR) Filter Structure

FIG 8.8 Structure of numerically controlled oscillator and low-pass filter (implemented as a finite impulse response (FIR) filter)

Having examined the detail of these blocks, we notice that the common functions are registers, adders, and multipliers with precisions as yet undefined. Parallel N-bit adders can be composed of N single-bit full adders. Multipliers are also built from full adders. N-bit registers are built from 1-bit flip-flops. Thus, one form of

regularity might be to use the same full adder for all parallel adders and multipliers. Similarly, the same flip-flop would be used in all locations.

Typically, the phase counter adder in the NCO would be of the order of 16–32 bits wide. The phase increment adder might be 8–16 bits wide. The sizes of the multipliers and adders in the FIR filter vary widely, but depend on the input data width. This typically varies from 1–12 bits.

Example

As illustrated in the previous section, IQ upconversion and downconversion can be converted to fixed hardware, as highlighted in blue in Figure 8.9. Whether the hardware is shared (i.e., the NCO and the multipliers) is a determination that can be made at the time of design. Once this is decided, the IQ modulation and demodulation is still undefined. These blocks tend to be highly variable depending on the particular system. Software radios have been proposed in areas where the standards are likely to evolve as time progresses. Rather than have any product fixed to an old standard, a software radio allows the product to be updated in the field via a firmware update. Thus, in our quest for a software radio architecture, we still want programmability.

FIG 8.9 Common IQ blocks

A solution to maintaining programmability while increasing processing power might be to use a multiprocessor as shown in Figure 8.10. Here, the IQ up-and-down conversion has been retained and the IQ modulation/demodulation is performed by the four processors. The number of processors is arbitrary and would be ascertained by a detailed analysis of the required computational power.

Imagine that the computational power required slightly exceeds that provided by the four processors shown in Figure 8.10. Because multiplication is a frequently required operation in signal processing operations, it makes sense to build a multiplier into each microprocessor, as shown in Figure 8.11. Hence, we maintain regularity and improve processing power.

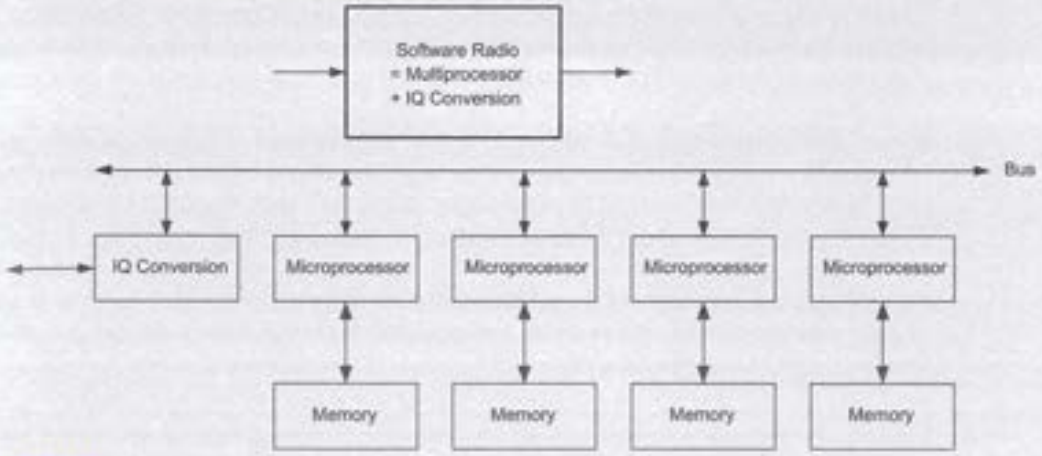


FIG 8.10 Software radio as a multiprocessor

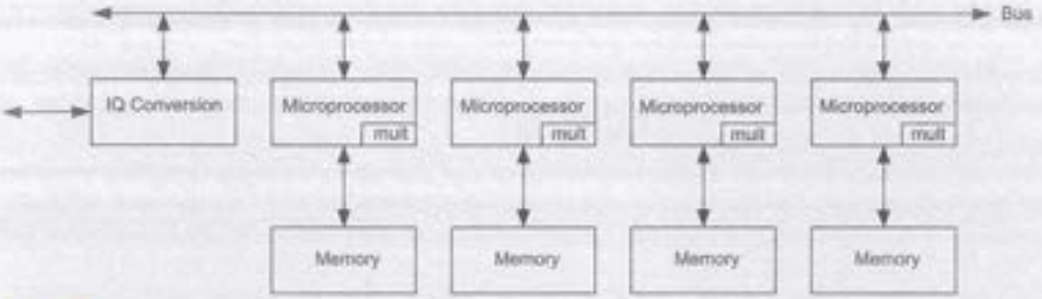


FIG 8.11 Enhanced multiprocessor for software radio

If the multiplication is a one-cycle operation, the throughput for multiplication-intensive operations can improve by a factor of up to M as compared to an M-bit processor with an iterative multiplication operation. This style of acceleration can be repeated for any operation that is computationally intensive. The application code is profiled, timing bottlenecks are identified, and custom hardware is added with appropriate instructions to access the hardware. In this manner the overall solution remains programmable while the speed of processing increases markedly. Tensilica sells extensible processors using such an approach. However, adding functional units increases die size and power dissipation, so tradeoffs are necessary.

8.2.4 Modularity

The tenet of modularity states that modules have well-defined functions and interfaces. If modules are "well-formed," the interaction with other modules can be well characterized. The notion of "well-formed" may differ from situation to situation, but a good starting point is the criteria placed on a "well-formed" software subroutine. First of all, a clearly defined interface is required. In the case of software, this is an argument list with typed variables. In the IC case, this corresponds to a clearly defined behavioral, structural, and physical interface that indicates the function as well as the name, signal type, and electrical and timing constraints of the ports on the design. Reasonable load capacitance and drive capability should be required for I/O ports. Too large a fan-in or too small a drive capability can lead to unexpected timing problems that take effort to solve, where we are trying to minimize effort. For noise immunity and predictable timing, inputs should only drive transistor gates, not diffusion terminals. The physical interface specification includes such attributes as position, connection layer, and wire width. In common with HDL descriptions, we usually classify ports as inputs, outputs, bidirectional, power, or ground. In addition, we would note whether a port is analog or digital. Modularity helps the designer clarify and document an approach to a problem, and also allows a design system to more easily check the attributes of a module as it is constructed (i.e., that outputs are not shorted to each other). The ability to divide the task into a set of well-defined modules also aids in System-On-Chip (SOC) designs where a number of IP sources have to be interfaced to complete a design.

Example

The low-pass filter in our software radio needs a large number of registers. Assuming that we wanted to implement a long filter (say 128 taps) with a 12-bit word, 3072 flip-flops are required for the I and Q filters. These can contribute significantly to the power of a portable communications IC, so it is tempting to prune transistors from the flip-flops. Figure 8.12 illustrates static and dynamic latches. The dynamic latch is much smaller but is sensitive to noise on the input and V_{DD}, suffers from the effects of leakage, and must be continuously clocked to preserve its state. The choice between the two options would depend to some extent on the design style that was being used. If the filter was being designed as a custom layout with known placement and hence wiring loads, the dynamic latch could be used. However, if the design was being automatically placed and routed, the static latch is the only prudent choice. For reasons discussed in Section 6.3, module inputs should only drive the gate terminals of MOS transistors used in a standard-cell place and route scenario. Usually, "discretion is the better part of valor" in these situations.

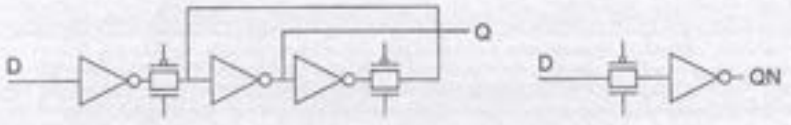


FIG 8.12 Cell level modularity example

Example

Another form of modularity, which aids in debugging and testability, is to ensure that all registers in a design are writable and readable. Figure 8.13 shows this principle applied to the NCO circuit. All registers have been made loadable via enables (lda, ldb, etc.) rather than the gated clocks shown in the original figure. Tristate buffers have been added so that the registers can also be read. Input and output registers have been added to the sine ROM to provide a temporally modular design. All modules should be registered on both their inputs and outputs. In this manner we know the input arrival time is a *D*-to-*clk* setup before the clock edge and the output timing is a *clk*-to-*Q* delay after the clock edge. The test task is identical for each register save for the number of bits that have to be written or read and the address of the register. In fact, by taking this approach, a table of register name, address, and bit width can be compiled and the HDL code to control the registers, documentation, and test programs automatically generated. With a more *ad hoc* approach, these productivity aids are not available.

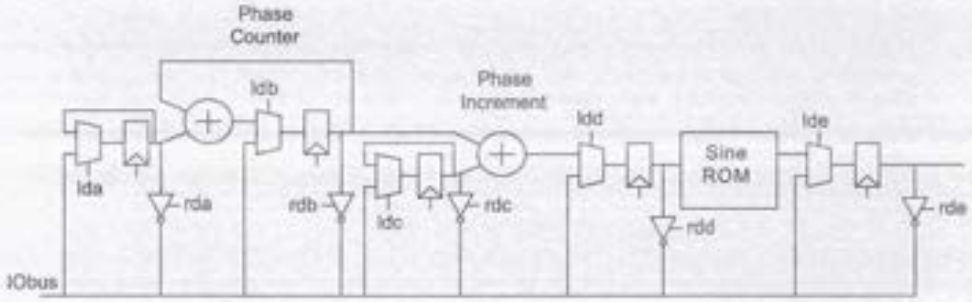


FIG 8.13 Modular test added to NCO

Example

In Figure 8.13, the phase-to-amplitude conversion is achieved using a sine ROM. The number of phase points determines the number of words in the ROM, and the desired precision determines the width of each word. A 7-bit ROM with an 8 bit-phase address will be 256 words long. Improving the phase precision involves increasing the length of the ROM. One trick to reduce the ROM length by a factor of four is to use the scheme shown in Figure 8.14. Only the first quadrant of the sine wave needs to be stored in the ROM. The sign bit (MSB) is used to conditionally invert the sine wave and the next bit in significance (*Phase*[*n*-1]) is used to count up or down through the ROM. Thus, a 256-word phase-to-amplitude table only needs 64 entries in the ROM.

While the sine lookup table is adequate for small ROMs, large ROMs can be slow and sometimes difficult to work into a standard cell design flow. An alternate structure that can

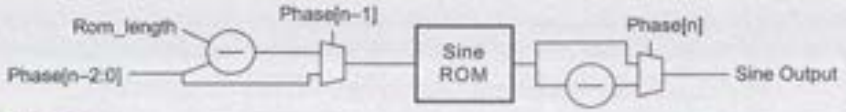


FIG 8.14 Sine ROM optimization

be used to generate a sine is called a *CORDIC processor*. The structure of the CORDIC processor is shown in Figure 8.15 [Volder59, Timmermann94, Gielis91, Grayver98, Sharma00]. It can be used to iteratively compute the functions:

$$x = x_0 \cos z_0 - y_0 \sin z_0$$
$$y = y_0 \cos z_0 + x_0 \sin z_0 \qquad (8.2)$$

Figure 8.15(a) shows a single CORDIC stage, which is composed of a parallel m-bit adder and two parallel m-bit adder/subtracters (an m-bit adder/subtracter is made up of an m-bit adder and m XOR gates, as described in Section 10.2.3.1). In addition, each stage contains two fixed shifters and a constant. The output of each stage is registered with three m-bit registers.

k stages are cascaded to perform the computation, as shown in Figure 8.15(b). If $x_0 = 1$, $y_0 = 0$, and $z_0 = \phi$, the structure will iteratively calculate $x_k \approx \cos \phi$, $y_k \approx \sin \phi$.[1] The right shift amount is equal to the iteration number (n), as shown in the figure. The constant is an approximation to the arctangent of an angle that decreases as the iteration index increases. The sign of the constant and the choice of addition or subtraction at stage n depends on the sign of z_{n-1}. See [Timmermann94] for more details. The amplitude precision is determined by the size of the adders and the phase precision is determined by the number of stages. The CORDIC processor can also compute square root, sinh, and cosh, and can perform IQ upconversion.

The objective of showing the CORDIC structure is to illustrate a possible improvement in regularity and modularity over the ROM implementation. The CORDIC only uses adders/subtracters and registers composed of regular logic gates and flip-flops, as opposed to a ROM structure. Moreover, the CORDIC is modular in three ways. First, it scales by simply extending the pipeline length or the bit width of the adders and registers. Second, it can provide a general class of polar functions, which might be of more utility in a situation where the maximum flexibility is required (i.e., as a coprocessor to a general-purpose microprocessor [Timmermann94]). If the cost is deemed satisfactory, the CORDIC provides a degree of future-proofing. Finally, it may be possible to eliminate the multipliers in the IQ

[1] Strictly speaking, CORDIC introduces a gain of approximately 1.647. To compensate, we can use $x_0 = 1/1.647$.

downconversion block by using an appropriately dimensioned CORDIC. Possible disadvantages include the pipeline delay through the CORDIC and perhaps power dissipation.

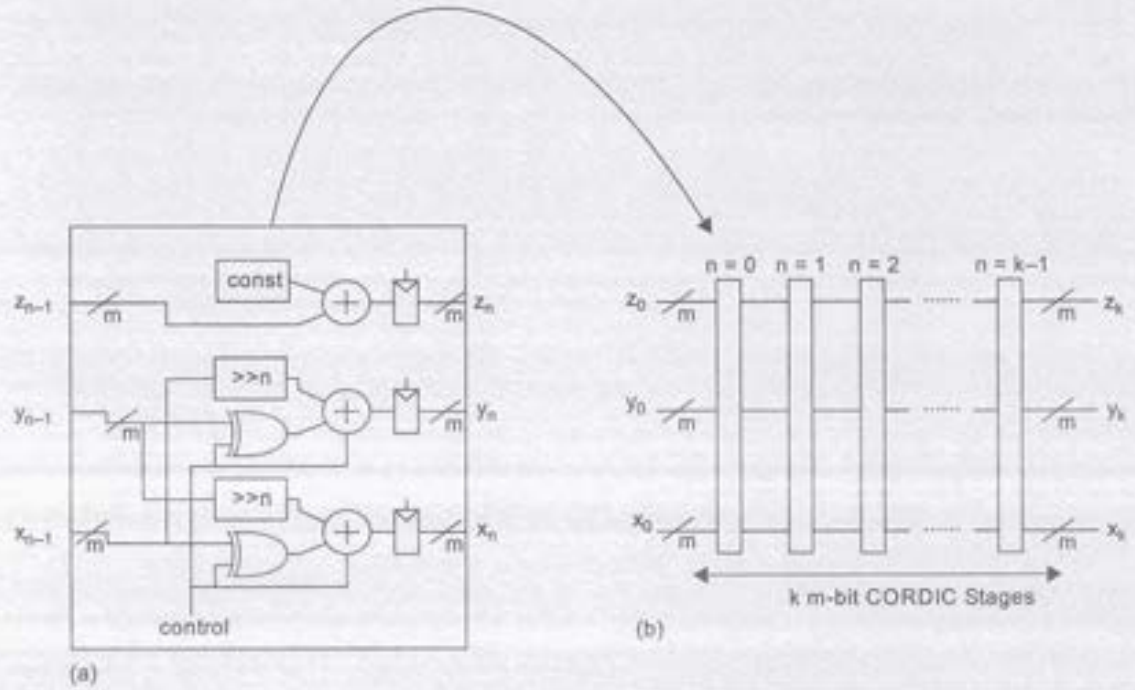


FIG 8.15 CORDIC processor as a sine generator

8.2.5 Locality

By defining well-characterized interfaces for a module, we are effectively stating that other than the specified external interfaces, the internals of the module are unimportant to other modules. In this way we are performing a form of "information hiding" that reduces the apparent complexity of the module. In the software and HDL world, this is paralleled by a reduction of global variables to a minimum (hopefully to zero). Increasingly, locality often means temporal locality or adherence to a clock or timing protocol. This is addressed in Chapter 7, where different clocking strategies are examined. One of the central themes of temporal locality is to reference all signals to a clock. Thus, input signals are specified with required setup and hold times relative to the clock, and outputs have delays related to the edges of the clock.

Example

In the example of the software radio, locality would probably be most evident in the floorplan of the chip. One example floorplan is shown in Figure 8.16. The analog blocks (ADC and DAC) are placed adjacent to the I/O pads. This is an example of physical locality because the analog blocks draw significant DC current and therefore the power busses have to be short and exhibit low resistance. Furthermore, the analog input and analog output signals can be routed to the pads without interference from digital signals. If necessary, the left edge of the chip can be guard-ringed and placed in a deep n-well if this process option is available. The digital IQ upconversion module is placed near the DAC and ADC, and the four programmable processor/memory composites are arrayed across the chip.

An alternative floorplan is shown in Figure 8.17. Here, the analog blocks and IQ conversion module are placed at the top of the chip. The four processor/memory blocks are then arrayed around a centrally located bus. The area for both array possibilities is roughly the same, but the second floorplan is better because the bus connecting the processors is shorter and hence faster and potentially dissipates less power. This is an example of physical locality used to obtain good temporal performance.

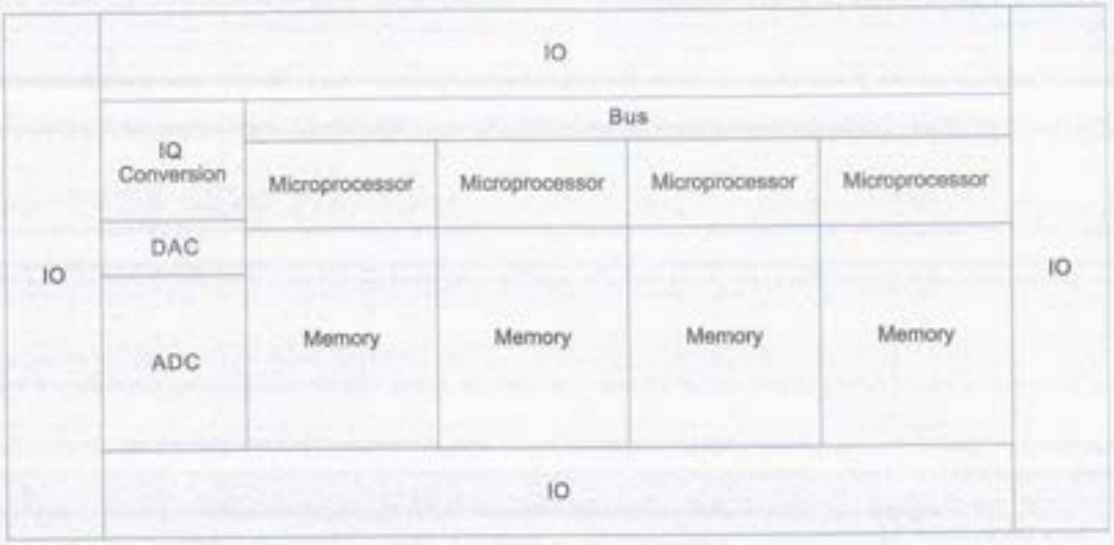


FIG 8.16 One possible floorplan for the software radio

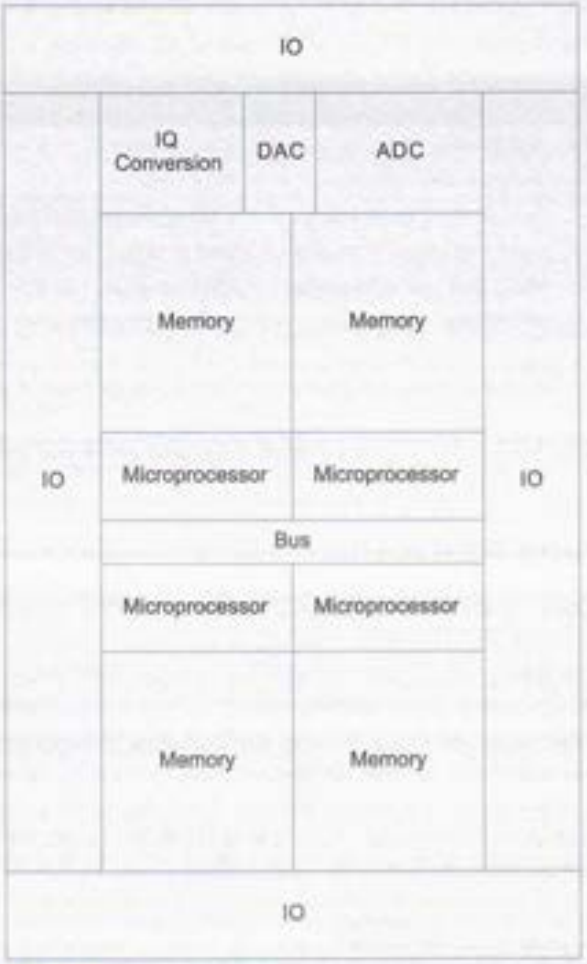


FIG 8.17 Alternate floorplan for software radio

8.2.6 Summary

There are strong parallels between the methods of design for software and hardware systems. Table 8.1 summarizes some of these parallels for the principles outlined above.

Table 8.1 Structured software and VLSI hardware design

Design Principle	Software	Hardware
Hierarchy	Subroutines, libraries	Modules
Regularity	Iteration, code sharing, object-oriented procedures	Datapaths, module reuse, regular arrays, gate arrays, standard cells
Modularity	Well-defined subroutine interfaces	Well-defined module interfaces, timing and loading data for modules, registered inputs and outputs
Locality	Local scoping, no global variables	Local connections through floorplanning

8.3 Design Methods

In this section, we will examine a range of design methods that can be used to implement a CMOS system. This section will concentrate on the target of the design method, in contrast to the design flow used to build a chip. Design flows, which deal with how a design progresses through a set of tools, will be dealt with in the subsequent section. The base design methods are arranged roughly in order of "increased investment," which loosely relates to the time and cost it takes to design and implement the system. It is important to understand the costs, capabilities, and limitations of a given implementation technology to select the right solution. For instance, it is futile to design a custom chip when an off-the-shelf solution that meets the system criteria is available for the same or lower cost.

8.3.1 Microprocessor/DSP

Many times, the most practical method to solve a system design problem is to use a standard microprocessor or digital signal processor (DSP). There are many single-chip microprocessors with built-in RAM and EEROM/EPROM available in the market. For example, the PIC family of processors from Microchip offers a wide range of clock speeds, memory sizes, and analog I/O capability (ADCs) in a small package. For more signal-intensive problems, classical DSPs from vendors such as Analog Devices and Texas Instruments can be employed. Microprocessors provide great flexibility because systems can be upgraded in the field through software patches. Do not underestimate the cost of software development for microprocessor-based systems.

Even when you decide to build a system with an off-the-shelf microprocessor, you should consider the possibility of eventual integration. For example, if your product becomes very successful and you want to reduce costs by integrating it into a single system-on-chip rather than building it as a board with a microprocessor and various support chips, you will need a microprocessor that is available in embedded form so that you can keep your software. Examples of embedded commercial processor cores include ARM, MIPS, and IBM's PowerPC. The (Sun Microsystems) SPARC V8 architecture is also available in VHDL for research or commercial use from Gaisler Research.

8.3.2 Programmable Logic

Often, the cost, speed, or power dissipation of a microprocessor may not meet system goals and an alternative solution is required. A variety of programmable chips are available that can be more efficient than general purpose microprocessors yet faster to develop than dedicated chips:

- Chips with programmable logic arrays
- Chips with programmable interconnect
- Chips with reprogrammable logic and interconnect

The system designer should be familiar with these options for two reasons:

- First, it allows the designer to competently assess a particular system requirement for an IC and recommend a solution, given the system complexity, the speed of operation, cost goals, time-to-market goals, and any other top-level concerns.
- Second, it familiarizes the IC designer with methods of making any chip reprogrammable at the hardware level and hence both more useful and of wider spread use.

8.3.2.1 Programmable Logic Devices The devices covered in this section are descended from chips that implement two-level sum-of-product programmable logic arrays (PLAs) discussed in Section 11.7. They differ from the field-programmable gate arrays described in the next section in that they have limited routing capability. Historically, process densities did not allow the transistor count and routing resources found in modern field-programmable gate arrays. Programmable logic devices based on PLAs allowed a useful product to be fielded and well-established techniques allowed logic optimization to target PLA structures, so the associated CAD tools were relatively simple. They are still occasionally used because the regular array and interconnect make timing very predictable.

A PLA consists of an AND plane and an OR plane to compute any function expressed as a sum of products. Each transistor in the AND and OR plane must be capable of being programmed to be present or not. This can be achieved by fully populating the AND and OR plane with a NOR structure at each PLA location. Each node is pro-

grammed with a floating-gate transistor (see Section 3.4.3), a fusible link, or a RAM-controlled transistor, as illustrated in Figure 8.18. The first two versions were the way these types of devices were programmed when device densities were low. These devices are generally used for low-complexity problems (a few hundred gates), which require fairly high speed (100's MHz).

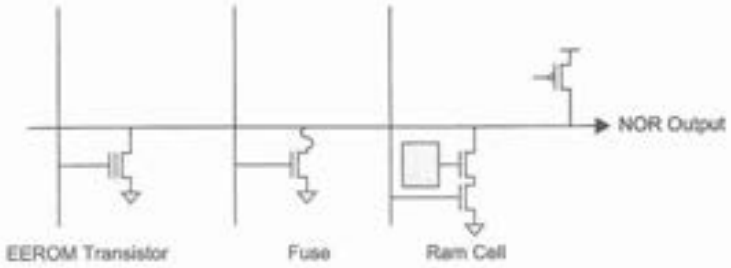


FIG 8.18 PLA NOR structure (one plane shown)

8.3.2.2 Field-programmable Gate Arrays (FPGAs) Field-Programmable Gate Arrays (FPGAs) use the high circuit densities in modern processes to construct ICs that, as their name suggests, are completely programmable even after a product is shipped or "in the field." Two basic versions exist. The first uses a special process option such as a fuse or antifuse to permanently program interconnect and personalize logic. These are one-time programmable. The second type uses small static RAM cells to customize routing and logic functions. In general, an FPGA chip consists of an array of logic cells surrounded by programmable routing resources.

As an example of the first type of FPGA, devices manufactured by Actel embed an array of logic modules within an interconnect matrix that is formed on the top metal layers. Successive routing channels run vertically or horizontally, as shown in Figure 8.19. At the intersection of routing traces is placed a special one-time programmable contact called an *antifuse*. These normally have high resistance (effectively an open circuit). Upon application of a special programming voltage across the contact, the resistance permanently drops to a few ohms. CMOS switches allow the programming voltage to be directed to any antifuse in the chip. The advantage of this type of routing is that the size of the programmable interconnect is tiny—the intersection area of two metal traces. Moreover, the on-resistance is low compared to a CMOS switch, so the circuit speed is not compromised. The disadvantage is that the interconnect is not reprogrammable, so once a chip is programmed, its function is fixed to the extent that the interconnect has been personalized.

The floorplan of a typical Actel chip is shown in Figure 8.20. An array of logic modules (LM) is placed on the chip with the routing passing adjacent to the cells. Routes extend into and out of the logic modules and intersect vertical or horizontal routing channels. The cell is personalized in terms of functionality and connectivity (to other cells) by

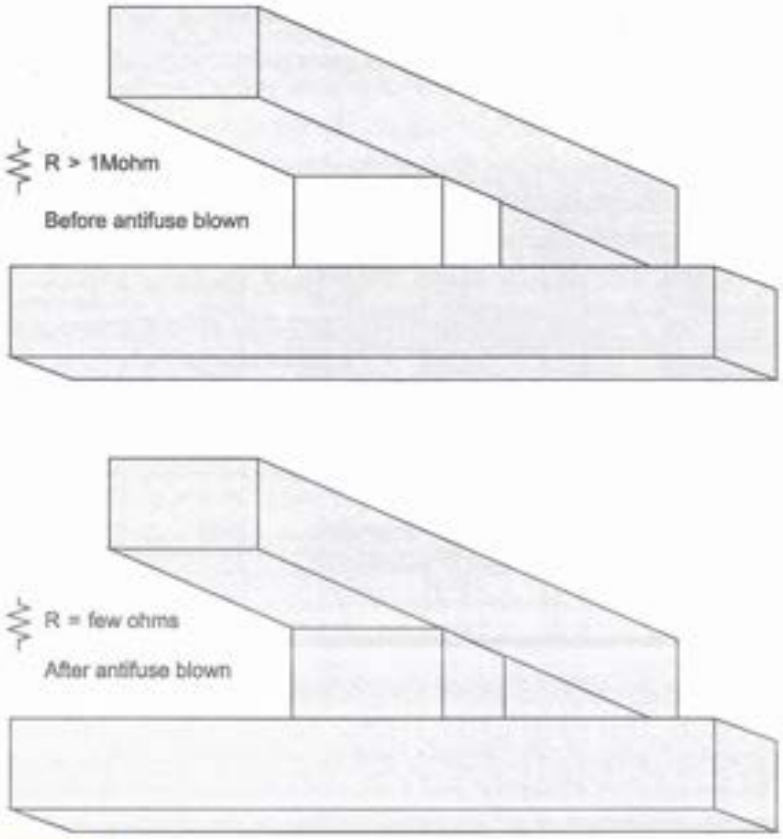


FIG 8.19 Actel programmable interconnect

programming these crossing interconnects. The logic module consists of a combinational block, which can compute more than 4000 combinational functions of five primary inputs. A flip-flop cell provides for asynchronous set and reset, active low enable, and programmable clock polarity. With typical utilization, effective gate count (equivalent two input gates) can range from 82K to over 1 million gates.

This type of array would be suitable for reasonably complex, high-speed (100's MHz) applications where cost and power constraints were moderate. Antifuses are only available in specialized CMOS processes.

Figure 8.21 shows the floorplan of a simplified RAM-based FPGA. The chip is composed of an array of *configurable logic blocks* (CLBs). Similar to the one-time programmable devices, metal routing tracks run vertically and horizontally between the array of CLBs. These terminate at the gray blocks, which are routing switches that can be implemented using CMOS transmission gates or tristate buffers. The routing resources can also be con-

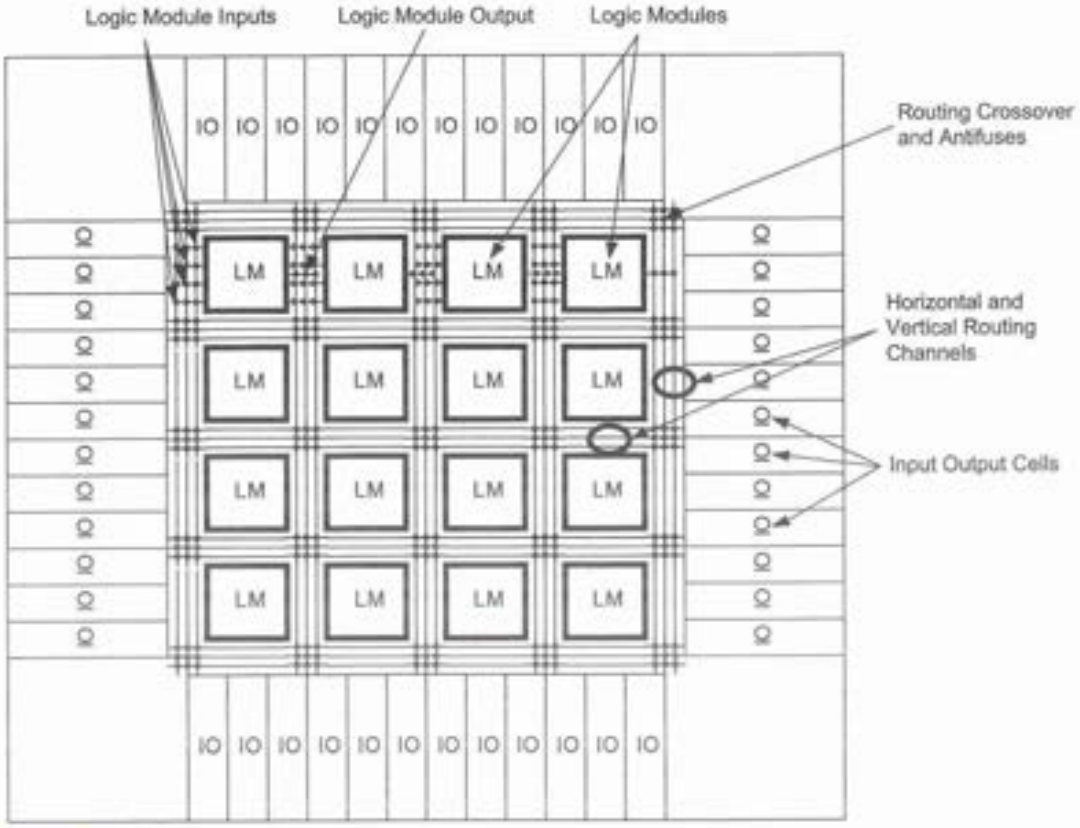


FIG 8.20 Representative Actel FPGA floorplan

nected to the inputs and outputs of the adjacent CLBs. CLBs use lookup tables to compute any function of several variables. The contents of the lookup table are stored in static RAM so the FPGA is programmed by reading the SRAM at startup from an external data source such as a serial EEPROM. Similarly, static RAM cells program the routing switches. Configurable I/O cells that can be used as input, output, or bidirectional pads surround the core array of CLBs.

A simple FPGA logic cell is shown in Figure 8.22. It is composed of a 16 × 1 static RAM as the logic element. This provides for any logic function of four variables merely by loading the RAM with the appropriate contents. Table 8.2 illustrates how the table should be loaded to perform various logic functions. A full adder can be implemented in two CLBs (one for carry and one for sum). The CLB shown also provides an optional output

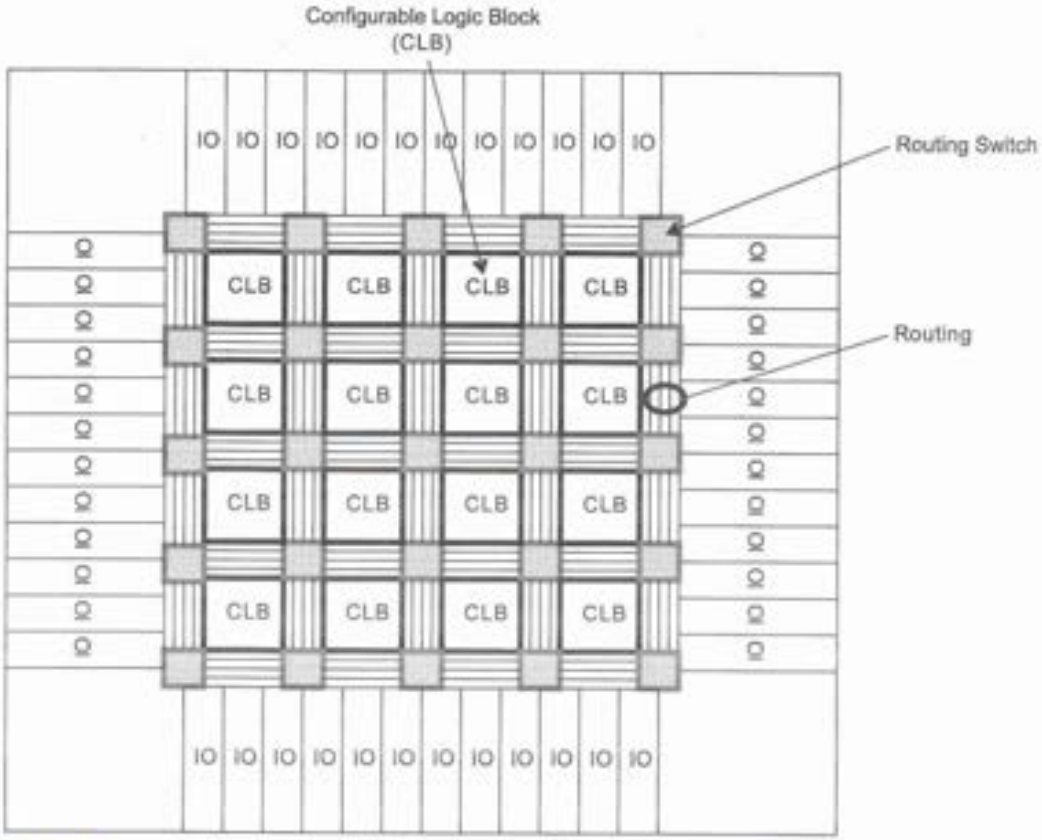


FIG 8.21 Simplified FPGA floorplan

register. While it may seem inefficient or slow to use a RAM to perform logic, specially designed single-data line RAMs are small and fast in current processes, and resources such as the routing tend to dominate modern designs from a density and speed viewpoint. The cell shown in Figure 8.22 is representative of first-generation FPGAs, and many optimizations have been made to improve performance. These include multiple dedicated clock lines, CLBs optimized for fast arithmetic, dedicated blocks of RAM, high-speed I/O interfaces, and even embedded RISC microprocessors. The Xilinx XCVP125 FPGA, state of the art in 2003, combines over 125,000 logic cells (approximately seven million two input gates), four RISC processors, multi-gigabit I/O and over 1000 user-available I/O pads (i.e., XILINX XC2VP125).

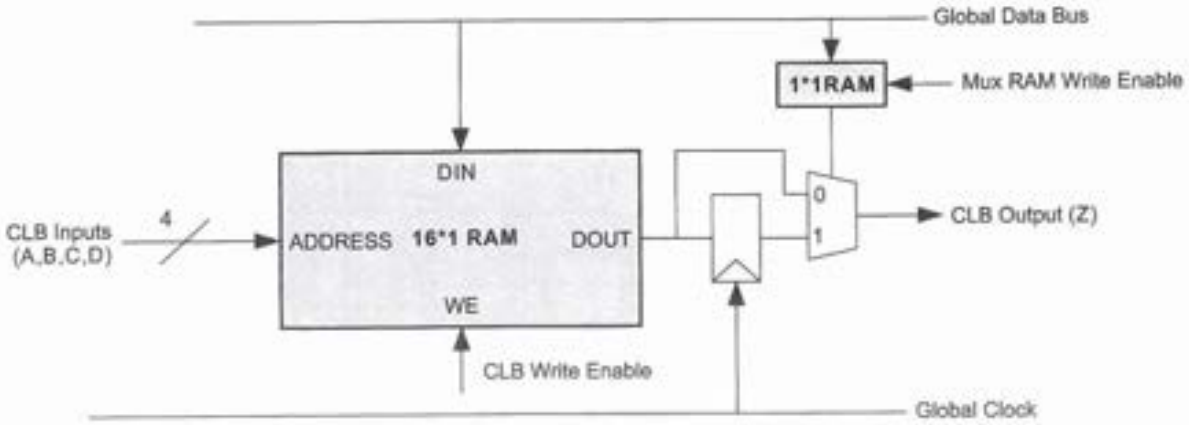
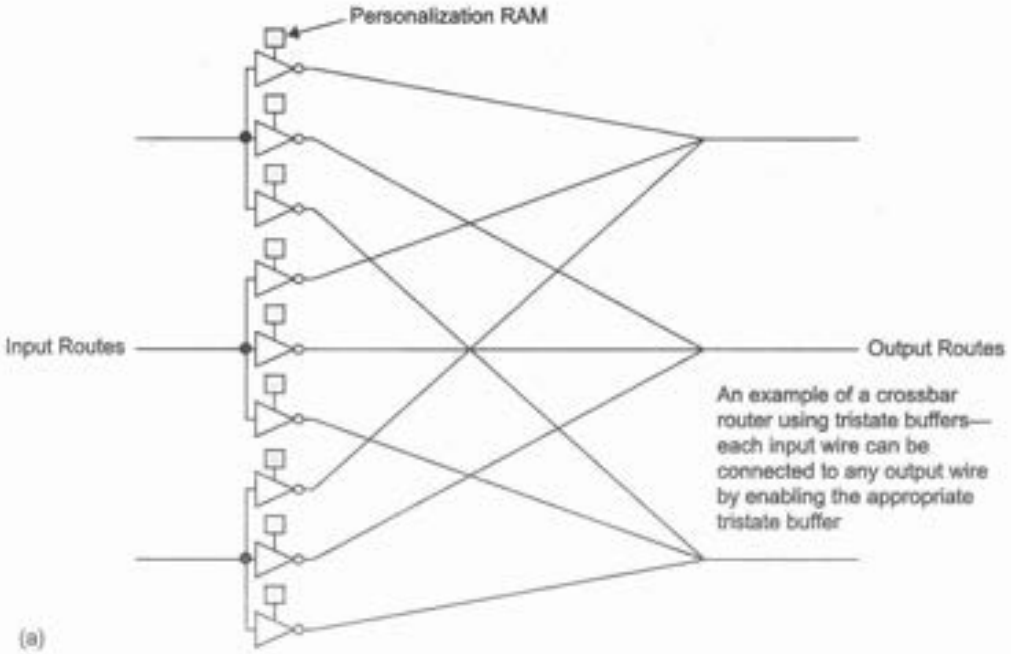


FIG 8.22 Simple FPGA logic cell

Table 8.2 RAM CLB functions

Address	ABCD	A • B • C • D	~A	SUM(A,B,C)
0	0000	0	1	0
1	1000	0	0	1
2	0100	0	1	1
3	1100	0	0	0
4	0010	0	1	1
5	1010	0	0	0
6	0110	0	1	0
7	1110	0	0	1
8	0001	0	1	0
9	1001	0	0	1
10	0101	0	1	1
11	1101	0	0	0
12	0011	0	1	1
13	1011	0	0	0
14	0111	0	1	0
15	1111	1	0	1



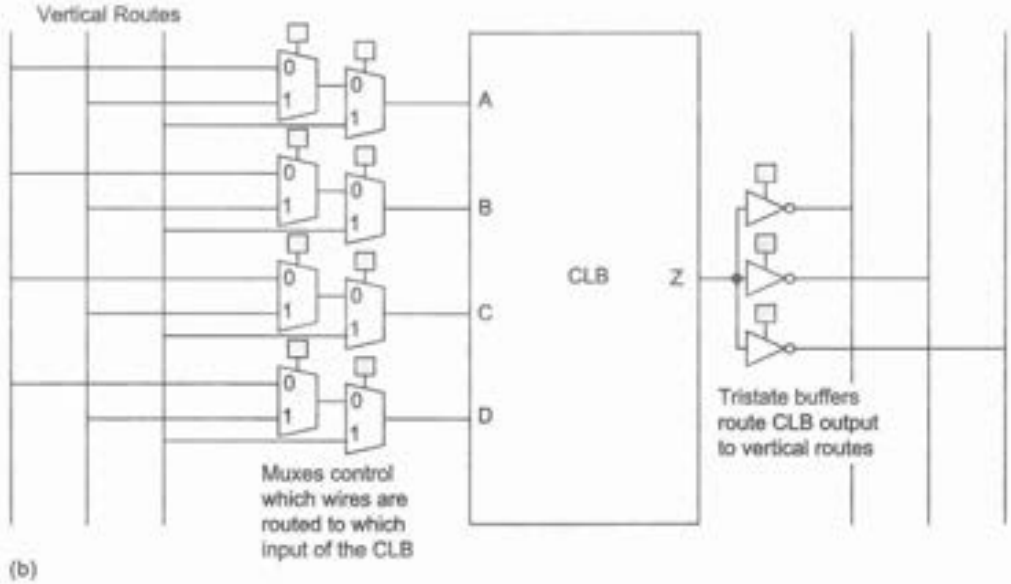


FIG 8.23 Simplified FPGA routing cell

One approach to routing cells is shown in Figure 8.23. The upper diagram shows a horizontal routing switch. Tristate buffers controlled by RAM bits determine which output routes are driven. The lower diagram shows the routing to the CLB of Figure 8.22. Each input can be connected to any of the three vertical routes to the left of the cell, while the tristate buffers can connect the output of the CLB to any route along the right of the cell.

Putting it all together, Figure 8.24 shows an example of a personalized array with the associated routing. Three inverting input pads enter on the left and are routed to two XOR gates, the output of which is routed out of the chip on the right.

FPGAs have matured to the point where they are the best choice for many low- to medium-volume custom logic applications. The economy of scale of modern processes allows extremely high densities of reprogrammable logic gates to be delivered at low cost to the designer, and hence to the consumer. Compared to conventional ICs, FPGAs are slower and are less efficient in area and power. However, the cost to manufacture is virtually zero (programming cost). The cost per part is moderate to high. As initial manufacturing costs for custom integrated circuits increase and FPGA costs decrease, FPGAs become attractive for more and more low- to mid-volume, moderate power applications.

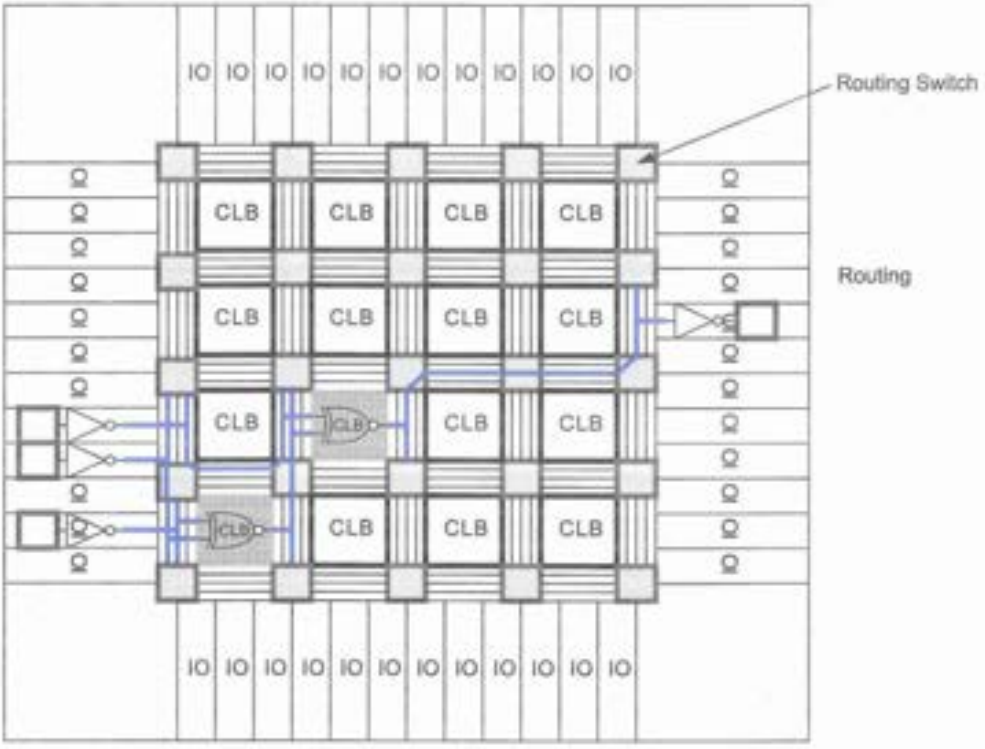


FIG 8.24 Personalized and routed FPGA

Note that (after sorting out the intellectual property rights with the appropriate patent holders) it is possible to implement FPGA blocks on any CMOS chip to provide some degree of programmability at the gate level.

8.3.3 Gate Array and Sea of Gates Design

The chips described in the previous section do not require a fabrication run. Designers typically strive to keep the nonrecurring engineering cost (NRE, see Section 8.5) as low as possible. One method of doing this is to construct a common base array of transistors and personalize the chip by altering the metallization (metal and via masks) that is placed on top of the transistors. This style of chip is called a Gate Array (GA). A particular subclass of a gate array is known as a Sea-of-Gates (SOG) chip. A vendor stocks *master* or *base* wafers that have been processed up to the polysilicon gate layer (i.e., the transistors have been formed). Contacts and metallization are then specified on a per-design basis to complete the chip. The cost can be kept low due to the following factors:

- The wafer cost can be reduced by producing many base wafers for a variety of different chips.
- Only the metallization masks are required to personalize the design; cost and processing time can be kept low for a small number of metal layers.
- Packaging cost is reduced by using standard packages and pinouts.
- Test and production costs are reduced by reusing common test fixtures.

Companies such as LSI Logic have pushed gate array technology to an advanced point by combining gate array structures with dedicated high-speed I/O circuitry. Their RapidChip technology boasts integrated RISC cores combined with nearly 20 million gates, 10 Mbits of RAM memory, and I/O speeds up to 4.25 Gbps. Speeds in the range of 300 MHz are possible with specially designed logic blocks.

Aside from their use as a System on Chip technology, it is worthwhile reviewing GA/SOG techniques because they can also be used on custom chips to provide an area of reprogrammable logic on an otherwise fixed function chip. The basic GA/SOG approach is shown in Figure 8.25. Rows of nMOS and pMOS transistors are arrayed across a chip area. Each logic row consists of an n row and p row. Gate arrays differ from SOG in that the array of transistors is not continuous and transistors can be grouped and perhaps individually sized for economic implementation of specific applications such as memories. Figure 8.26(a) shows an SOG structure, which features continuous rows of transistors. Figure 8.26(b) shows a gate array structure that uses groups of three transistor pairs. Because the SOG structure is a continuous array of transistors, it has to allow for the isolation of a group of transistors. Grounding the gate of the nMOS transistor or connecting the gate of the pMOS transistor to the V_{DD} rail provides this isolation.

Figure 8.27 shows a portion of an SOG structure programmed to be a 3-input NAND gate. Note that the nMOS and pMOS transistors at each end isolate the gate, as described previously. Personalization of this SOG structure commences at contact and

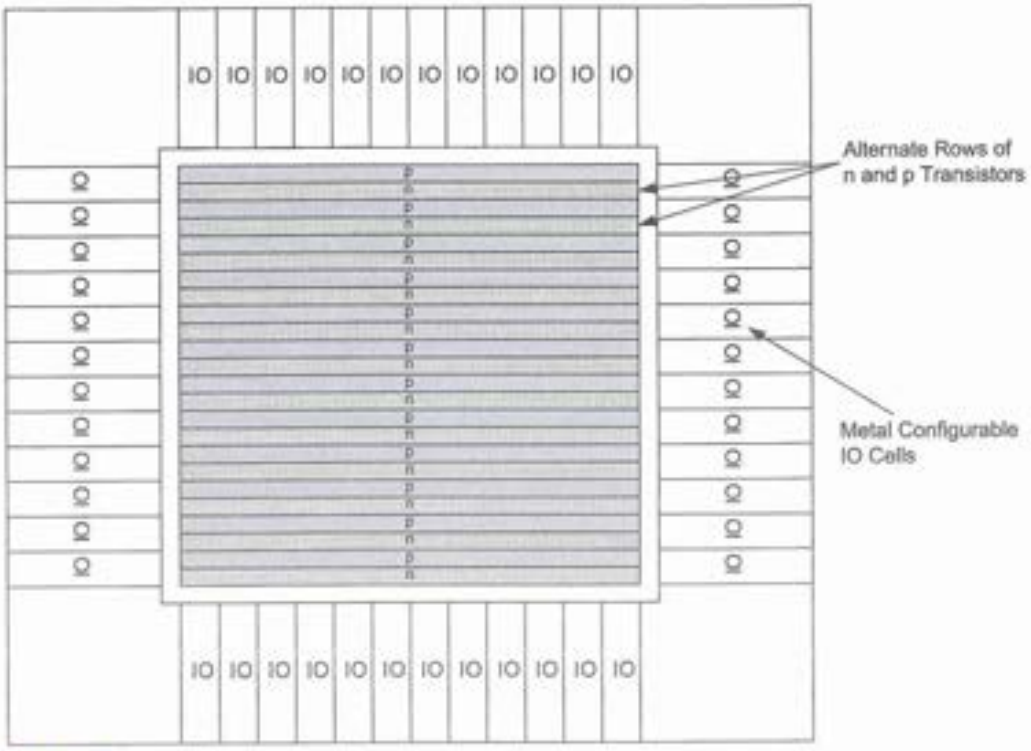


FIG 8.25 Sea of Gates floorplan and transistor layout

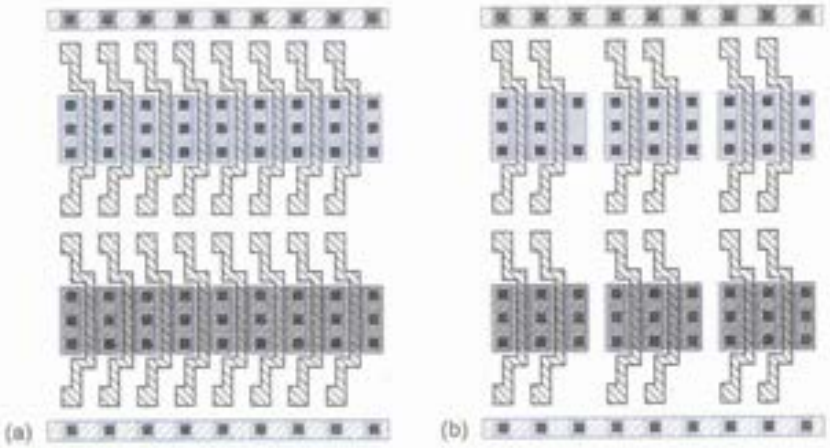


FIG 8.26 SOG and Gate Array cell layouts

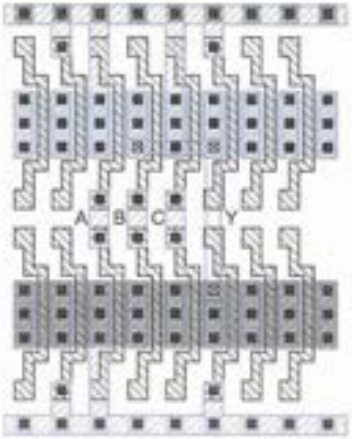


FIG 8.27 SOG programming example for 3-input NAND gate

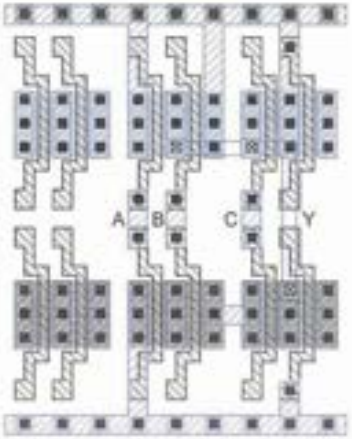


FIG 8.28 Gate Array programming example for 3-input NAND gate

metal1 masks, and can continue up for all metal layers available in the process. Figure 8.28 shows the 3-input NAND on a gate array.

Gate and SOG arrays have lower up-front costs because they require fewer masks. They are also faster to manufacture because only the metallization steps are required. Some vendors offer automatic conversions of field-programmable gate arrays into fixed gate arrays to reduce costs as a product moves from low- to mid-volume manufacturing.

Compared to an FPGA given the same process, a gate array dissipates less power and costs less in volume production. The NRE cost (i.e., cost to fabricate) can be a fraction of the cost of a custom mask set.

8.3.4 Cell-based Design

Cell-based design uses a standard cell library as the basic building blocks of a chip. The cells are placed in appropriate positions, then their interconnections are routed. Cell-based design can deliver smaller, faster, and lower-power chips than gate arrays or programmable logic but has high NRE costs to produce the custom mask set. Therefore, it is only economical for high volume parts or when the performance commands a lucrative sales price. As compared to full-custom design, cell-based design offers much higher productivity because it uses predesigned cells with layouts. Foundries and library vendors supply cells with a wide range of functionality. These include:

- Small-scale integration (SSI) logic (NAND, NOR, XOR, AOI, OAI, inverters, buffers, registers)

- Memories (RAM, ROM, CAM, register files)
- System level modules such as processors, protocol processors, serial interfaces, and bus interfaces
- Possibility of mixed-signal and RF modules

Whereas Medium Scale Integration (MSI) functions such as adders, multipliers, and parity blocks used to be supplied as cells, synthesis engines commonly construct these from base-level Small Scale Integration (SSI) gates in current design systems.

A typical standard cell library is shown in Table 8.3. A 1X (normal power) cell commonly is defined to use the widest transistors that fit within the vertical pitch of the standard cell. 2X and larger (high power) cells use wider transistors to deliver more current. They must fold the transistors (see Section 8.8.3) to fit within the cell; this comes at the expense of increased cell width. Gates are often available in low power versions as well. These cells use minimum-width transistors to reduce capacitance. Low-power cells tend to be slow because of the wire capacitance they must drive. Although they do not save area, they do reduce power consumption on noncritical paths.

Sophisticated libraries also generate memories of assorted sizes from a graphical user interface. The generators yield not only the physical layout but also a complete data sheet indicating access times, cycle times, and power dissipation.

In the event that a standard cell library may not be available for a process, it is worthwhile to review some of the approaches to standard cell design. Usually, standard cells are a fixed height with power and ground routed respectively at the top and bottom of the cells, as was shown in Figure 1.62. This allows the cells to be abutted end to end and to have the supply rails connect. A single row of nMOS transistors adjacent to GND (ground) and a single row of pMOS transistors adjacent to V_{DD} (power) are normally used. The polysilicon gate is connected from nMOS transistor to pMOS transistor and, in the case of multiplexers and registers, the polysilicon connection has to be crossed between vertically coincident nMOS and pMOS transistors. Decisions about the sizes of transistors have to be made. Following this decision, the cells are almost completely defined by the process design rules. Figure 8.29 illustrates this point. The height of the cell is defined by the sum of the nMOS and pMOS transistor widths, the separation on n and p regions, the spacing to V_{DD} and GND busses, and the width of these busses. The horizontal pitch is defined by the poly-to-metal2 contacted pitch, as shown in the figure. It is relatively easy to construct a software program to automatically generate cells like those shown in Figure 8.29. Cell delay is characterized through simulation to good agreement with silicon, as was discussed in Section 5.5.3. Fabrication of such cells to prove performance is rarely required. Options to standard cells include routing the clock with the power and ground busses and routing multiple supply voltages to each cell. The latter technique is sometimes used to reduce power by connecting gates that are not in the critical path to a lower than normal supply voltage. Recall that the power drops with the square of the supply voltage.

Table 8.3 Typical standard cell library

Gate Type	Variations	Options
Inverter / buffer / tristate buffers		Wide range of power options, 1X, 2X, 4X, 8X, 16X, 32X, 64X minimum size inverter
NAND / AND	2–8 inputs	High, normal, low power
NOR / OR	2–8 inputs	High, normal, low power
XOR / XNOR		High, normal, low power
AOI / OAI		High, normal, low power
Multiplexers	Inverting/noninverting	High, normal, low power
Schmitt trigger		High, normal, low power
Adder / half adder		High, normal, low power
Latches		High, normal, low power
Flip-flops	D, with and without synch/asych set and reset, scan	High, normal, low power
I/O pads	Input, output, tristate, bidirectional, boundary scan, slew rate limited, crystal oscillator	Various drive levels (1–16 mA) and logic levels

8.3.5 Full Custom Design

A number of techniques can be used to design standard cells or larger circuit blocks at the mask level. The oldest and most traditional technique is termed *custom mask layout*, in which a designer sits in front of a graphics display running an interactive editor and pieces designs together at the geometry level one rectangle at a time. This work is sometimes called polygon pushing. Historically, when there were no interactive graphics displays or editors (yes, there was a time when this was true!!), designers manually cut masks from a material called Rubylith. Sheets of the material (a red strippable layer on an otherwise clear Mylar plastic background) were used directly to produce photomasks for chips. Mask design under such conditions was laborious and prone to error because there were no tools to check the designs other than human eyes.

A variation of custom mask design is called *symbolic layout*. Rather than dealing with rectangles and polygons on various mask levels, the primitives are transistors, contacts, wires, and ports (points of connection). These primitives can also be manipulated by a graphics editor. Some systems allow for a "design rule free" placement of symbolic entities. The actual placement occurs after a spacing process that compacts each primitive as close to its neighbor as possible according to the design rules of the process in use. By using a symbolic layout system, layout topologies can be transported from process to process without a huge amount of effort.

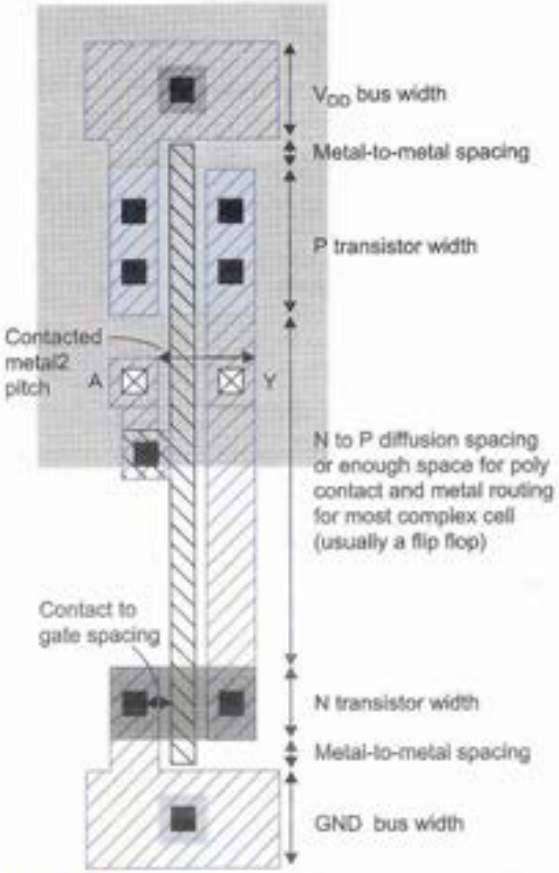


FIG 8.29 Typical standard cell layout with some of the constraints

In these times of cell-based design, digital CMOS ICs use custom mask design only for the highest of volume parts such as microprocessor datapaths. However, analog and RF designs, cell libraries, memories, and I/O cells still frequently use custom design. There are a variety of custom MOS layout hints in Section 8.8. Custom design is also worthwhile pedagogically because it completes the link from transistors to systems.

From time to time, we have mentioned software generators as a method of generating physical layout. This kind of idea has been around for a long time and was often referred to as *silicon compilation*. Complete microprocessors were typical of layouts that were generated. A "correct by construction" method was used to build the layouts hierarchically. In other words, only the mask description was generated, with perhaps a high-level instruction level simulator being the behavioral model. Generators are the most common method used today for library generation.

With modern design flows, many different "views" of a design are required to integrate with the regular path through the design system. For instance, in addition to the behavioral model, a timing view would be needed for timing verification, a logic view might be required for simulation, and a circuit view for layout versus schematic or netlist comparisons would be needed. Software generators can be used to provide all of these views automatically.

Modern versions of the venerable "silicon compiler" can be built in a structured hierarchical manner to generate memories, register files, and other special-purpose structures that can benefit from a customized layout. One of the most straightforward approaches is to write custom placement routines that in essence "hand place" certain standard cells within the row structure of a standard cell design. For instance, you may prefer a certain adder design and have a datapath layout for the adder. An algorithm can be written to place the cells on the standard cell grid. In addition, a linked algorithm can be written to generate a gate netlist in an HDL. In this way, both the physical and structural design are captured. The behavior can be represented by an HDL function or module call. Such custom placement can shorten wire lengths and thus improve speed and power.

To illustrate a memory generator, consider the sine ROM required for the NCO in the software radio. The ROM contents are available as a file with a list of numbers and we want to generate a layout and the Verilog. Generating the Verilog is straightforward and would proceed according to the following pseudo-code (loosely based on the C language):

```
function generate_ROM_verilog(ROM_length, ROM_width, ROM_filename);
   verilog_file_pointer = open("sinerom.v");
   generate_verilog_head(verilog_file_pointer, ROM_length, ROM_width);
   ROM_file_pointer = open(ROM_filename);
   index = 0;
   loop for index below 2<<ROM_length /* rom_length=6 -> 64 */
      ROM_entry = get_line_from_file(file_pointer);
      fprintf(verilog_file_pointer, " %d: q = %d'h%h",
            Index, ROM_width, ROM_entry);
      index++;
      file_pointer++;
   end_loop
   generate_verilog_tail(verilog_file_pointer);
end_function;

function generate_verilog_head(file_pointer, length, width);
   fprintf(file_pointer, "module quarter_wave(");
   fprintf(file_pointer, " input [%d:0] addr,", length-1);
   fprintf(file_pointer, " output reg [%d:0] q;)", width-1);
   fprintf(file_pointer, " always @(addr)");
   fprintf(file_pointer, " begin");
   fprintf(file_pointer, " case(addr)");
end_function;

function generate_verilog_tail(file_pointer);
   fprintf(file_pointer, " endcase");
   fprintf(file_pointer, " end");
   fprintf(file_pointer, "endmodule");
end_function;
```

The function is quite easy to follow. It first writes out the Verilog file header structure. This includes the module name, input/output ports, and the bit widths of the ports. These are specified to the function from the calling function. The program then sequences through the ROM file contents and formats each line of the case statement. Finally, the tail of the Verilog file is generated. The following code segment illustrates the output.

```
module quarter_wave(input      [5:0] addr,
                    output reg [6:0] q);
   always @(addr)
   begin
      case(addr)
         0: q = 7'h00;
         1: q = 7'h03;
         2: q = 7'h06;
         ...
         62: q = 7'h7f;
         63: q = 7'h7f;
      endcase
   end
endmodule
```

While this function is basic, the sophistication can be built up over time. There are a number of academic and commercial projects that have been aimed at HDL generation such as Verilog++ and Verilog2C++.

For the generation of the physical layout, we will concentrate on the generation of the ROM array. Details on ROM design may be found in Section 11.4. The main program is shown below. The program assumes that layout cells or tiles connect by abutment, i.e., the routing is implicit in the tile.

```
function generate_rom_layout(ROM_length, ROM_width, ROM_filename);
   lfp = open("sinerom.lay"); /*layout file pointer*/
   generate_rom(lfp, 0, 0, ROM_length, ROM_width, ROM_filename);
   x = length_of_rom_cell * ROM_length;
   generate_rom_right(lfp, x, ROM_length, ROM_width, ROM_filename);
endfunction
```

The notion of *tiling* is introduced in this code. The function `generate_rom` tiles two rectangular blocks horizontally. The first is the ROM array and row decoder. Then the `rom_right` is placed to the right of the array. This contains the ROM sense amplifier, which in this case is shown as a pMOS pull-up and buffer inverter for the pseudo-nMOS NOR structure used in the ROM. The bottom right of the `rom_right` structure can contain a column decoder, but it is not necessary in this design. The address buffers for the row decoder are placed in this corner. This physical structure is shown in Figure 8.30.

```
function generate_rom(lfp, xorigin, yorigin, length, width);
   generate_row_decoder(lfp, xorigin, yorigin, length, width);
   y = yorigin + row_decoder_height;
   generate_rom_array(lfp, xorigin, y, length, width);
end_function
```

The function `generate_rom` tiles two layout blocks vertically. The lower block is the ROM row decoder and the upper block is the ROM array proper. The row decoder gate is shown to the right of the floorplan. In this design, a pseudo-nMOS NOR gate is used for compactness and simplicity. This is shown in Figure 8.31.

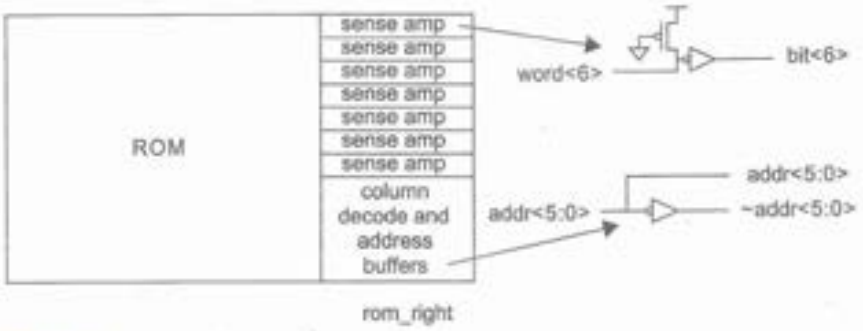


FIG 8.30 ROM layout—phase 1

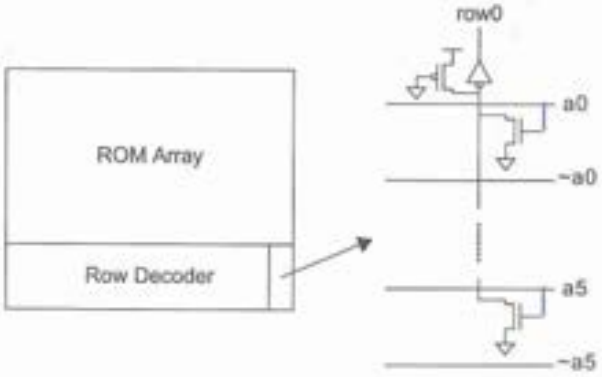


FIG 8.31 ROM layout—phase 2

Finally, the function `generate_rom_array` is invoked to build the ROM array itself.

```
function generate_rom_array(lfp, xorigin, yorigin, ROM_length, ROM_width, ROM_filename);
   ROM_file_pointer = open(ROM_filename);
   index = 0;
   x = xorigin;
   loop for index below 2<<ROM_length /* rom_length=6 -> 64 */
      ROM_entry = get_line_from_file(file_pointer);
      bit_index = 0;
      y = yorigin;
      loop for bit_index below ROM_width
         bit = get_bit_from_entry(bit_index, ROM_entry);
         if (bit) ROM_one(lfp, x, y);
         else ROM_zero(lfp, x, y);
         y = y + yorigin;
      end_loop
      x = x + rom_bit_width;
      index++;
      file_pointer++;
   end_loop
end_function;
```

This function tiles in both *x* and *y* directions and builds a ROM array as shown in Figure 8.32.

This builds a long, skinny ROM. For the sine ROM example, the array would be 64 cells long by 7 cells high. The loop variables can be changed in the function to generate, for example, four ROM words vertically, which would result in a structure that was 16 cells long by 28 cells high. A column decoder would be added to the ROM sense amplifier in the `rom_right` block to select one bit out of each group of four.

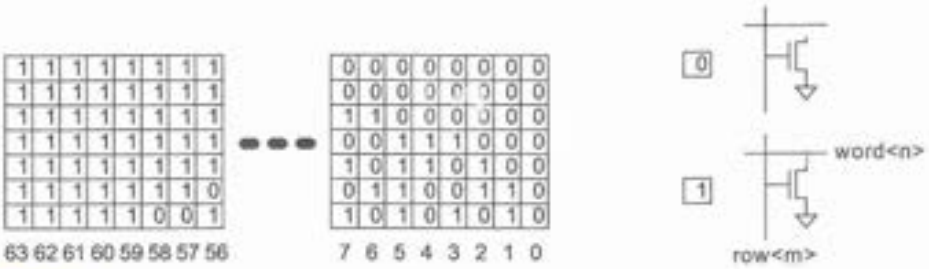


FIG 8.32 ROM array layout

Datapaths are another class of structure that can benefit from a software generator. The following code is representative of what might be used to place the cells in a CORDIC processor. Routing may be able to be incorporated into the tiled cells, or specific routing cells may have to be generated in addition to the pseudo-code shown below.

```
function generate_cordic_layout(bit_width, length);
   lfp = open("cordic.lay"); /*layout file pointer*/
   index = 0;
   x = 0;
   loop for index below length;
      generate_cordic_slice(lfp, x, width, index);
      index++;
      x = x + cordic_slice_width;
   end_loop
end_function
```

The function above arrays the CORDIC slices horizontally. This is shown in Figure 8.33.

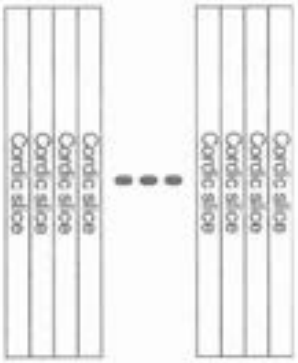


FIG 8.33 CORDIC processor

```
function generate_cordic_slice(lfp, x, width, index);
   y = 0;
   xleft = x;
   generate_cordic_shift(lfp, x, y, width, index);
   generate_cordic_constant(lfp, x, y + 2 * width * adder_height,
      width,index);
   x = x + shift_width;
   generate_cordic_adder(lfp, x, y, width,0); /* x slice */
   y = y + adder_height * width;
   generate_cordic_adder(lfp, x, y, width,0); /* y slice */
   y = y + adder_height * width;
   generate_cordic_adder(lfp, x, y, width,1); /* z slice */
   y = y + adder_height * width;
   generate_cordic_control(lfp, xleft, y); /* control on top */
end_function
```

The `generate_cordic_slice` function adds a shifter for the X and Y paths (`generate_cordic_shift`) and a constant for the Z path (`generate_cordic_constant`) and then builds the X and Y adder/subtracter and the Z adder vertically (`generate_cordic_adder`). Each is `width` bits wide. The control block resides on top of the three datapaths. This is shown in Figure 8.34. The shifter block would consist of wiring that would perform an n-bit shift and XY swizzle according to the horizontal index. Similarly, the constant block would generate a constant dependent on the horizontal index.

The next function generates the stage adders. This is shown in Figure 8.35.

```
function generate_cordic_adder(lfp, x, y, width, flag);
   index = 0;
   y = 0;
   loop for yindex below width
      generate_xyz_bit(lfp, x, y, flag);
      yindex++;
      y = y + adder_bit_height;
   end_loop
end_function
```

The X and Y adder bits have a multiplexer, XOR, ripple carry adder, and flip-flop, while the Z bits have a blank (null) cell, an adder, and a flip-flop. These bits are shown in Figure 8.36(a) and 8.36(b) respectively.

```
function generate_xyz_bit(lfp, x, y, flag);
   if (flag) place_null_cell(lfp, x, y);
   else place_xor(lfp, x, y);
   x = x + xor_width;
   place_adder(lfp, x, y);
   x = x + adder_width;
   place_flipflop(lfp, x, y);
end_function
```

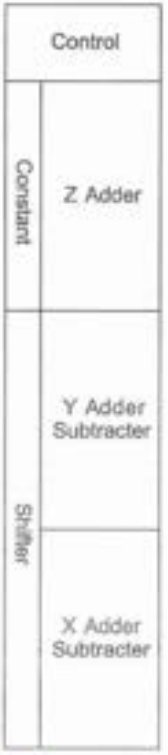


FIG 8.34 CORDIC processor slice

FIG 8.35 CORDIC XYZ adders

FIG 8.36 CORDIC XYZ adder bits

In the previous examples, the *x* and *y* locations for cell placement have been calculated using "magic" global dimensions (i.e., `adder_width` in the last code example). It is quite possible to build an automated tiling engine that just takes alternating ordered lists of cells and abuts them in the *x* or *y* direction, incrementing the origin of the cell to be placed by the dimensions of the previous cell. In this way, no global numbers need to be used in the code generation and the code is generic for a wide range of processes. The other detail omitted from the preceding examples is the generation of signal pins. These are usually needed eventually for layout routers to connect to the generated modules and for comparison with structural views.

8.3.6 Platform-based Design—System on a Chip

As systems have become more complex, the use of predefined intellectual property (IP) blocks has become commonplace. Designs frequently use a number of common blocks such as RISC processors, memory, and I/O functions attached to common busses. A *platform* can be used to implement a design by using common structures such as busses and common high-level languages (such as C) to program the processors. To a large extent, the RISC processor and memories can be interchanged and the number and type of peripherals can be changed while maintaining good design and verification times because the modules have been predesigned and the test and verification scripts come with the IP blocks. The design task is to put the blocks together, design any application-specific blocks, and place and route a correctly operational chip. Note that the last step, while automated, still takes considerable engineering effort.

As many current chips feature one or more embedded microprocessors, the task of writing software is added to the task of designing logic. Moreover, platform-based design poses the problem of partitioning the complete solution between hardware (HDL, gates) and software (programmed on the processor/s). This tends to remain a somewhat manual task, but is increasingly automated by CAD tools.

Platform-based systems typically consist of a basic RISC processor, which can be extended with multipliers, floating point units, or specialized DSP units. In addition (e.g., in Tensilica's Xtensa system), by profiling the executable code, special hardware can be added that corresponds to hardware-assisted instructions, which are introduced into the instruction set. In theory, additional hardware or extra processors can deal with a wide range of computational loads.

Manual techniques for hardware-software codesign mirror this approach. That is, the design begins with a software simulation (ideally on the embedded processor). Timing estimates are gathered, and manual decisions about what to commit to hardware are made. Special simulators to deal with embedded processors and logic have been developed.

With platform-based design, we have in essence come full circle from the first design method suggested: programming a microprocessor. This is the reason processor selection is important when starting out on a product design that may eventually be integrated. As the software effort will often exceed the hardware effort, you don't want to repeat that effort.

8.3.7 Summary

In this section, we have summarized a wide range of CMOS design options ranging from a software-based microprocessor to full custom design. The following table summarizes these options in terms of a variety of criteria. Each category is ranked in relation to each design method from low to high.

Table 8.4 Comparison of CMOS design methods

Design Method	Non-recurring Engineering	Unit Cost	Power Dissipation	Complexity of Implementation	Time to Market	Performance	Flexibility
Microprocessor/ DSP	low	medium	high	low	low	low	high
PLA	low	medium	medium	low	low	medium	low
FPGA	low	high	medium	medium	low	medium	high
Gate Array/SOG	medium	medium	low	medium	medium	medium	medium
Cell Based	high	low	low	high	high	high	low
Custom Design	high	low	low	high	high	very high	low
Platform Based	high	low	low	high	high	high	medium

In the next section, we review the design flows necessary to build a complete chip. While decisions can be difficult, the following diagram (Figure 8.37) should offer some assistance.

The diagram (Figure 8.37) indicates that the most cost-effective approach should be taken to hardware (or software) design given speed, power, and cost targets (occasionally, size will count as well). You should always use an off-the-shelf solution if system constraints are met, because the non-recurring engineering (NRE) costs are amortized over many units. The next most likely prospect is an FPGA design, especially for low-volume (100,000's) applications. Power and cost are the most likely attributes to be challenged in medium- to high-volume applications, and this is where standard cell designs will be used. Mixed-signal, RF, and high-speed digital designs require a cell-based or custom approach.

Increasingly, the NRE cost (predominantly mask cost) is approaching a level where even industry prototypes must be done using multi-project chips, amortizing the mask cost over multiple designs on the same reticle. Designs must be as re-programmable or adaptable as possible.

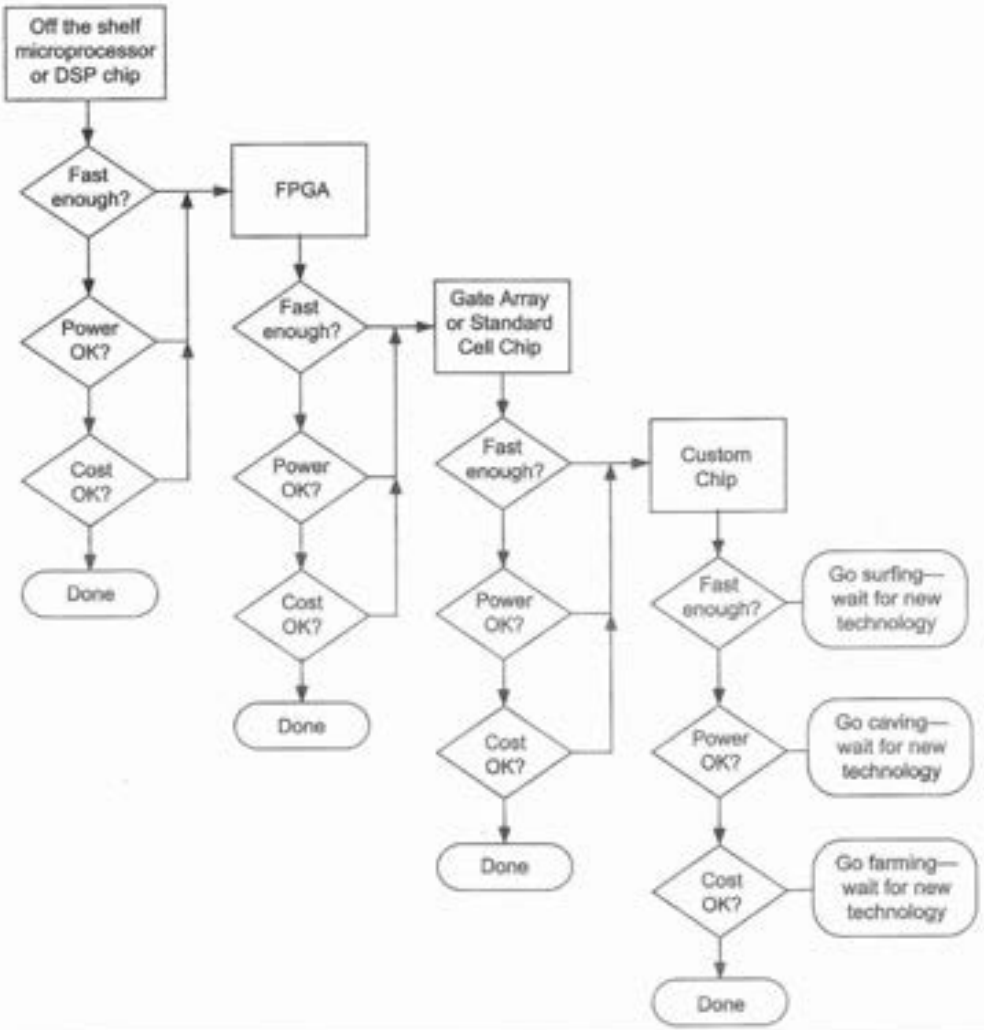


FIG 8.37 Design decision tradeoffs

8.4 Design Flows

A design flow is a set of procedures that allows designers to progress from a specification for a chip to the final chip implementation in an error-free way. In the previous section, we discussed the basic CMOS design methods without mentioning how we actually design an FPGA, gate array, or cell-based system. In this section, we will summarize the main design flows in use today.

A general design flow is shown in Figure 8.38. Design starts at the behavioral level and then proceeds to the structural level (gates and registers). This step is called behavioral or *Register Transfer Level* (RTL) synthesis because the designs are captured at the RTL (memory elements and logic) level in an HDL. The description is then transformed to a physical description suitable for chip fabrication. This step is called *physical synthesis* (or *layout generation*). Normally, the synthesis steps are automated, albeit guided by human judgment. The verification steps are also shown.

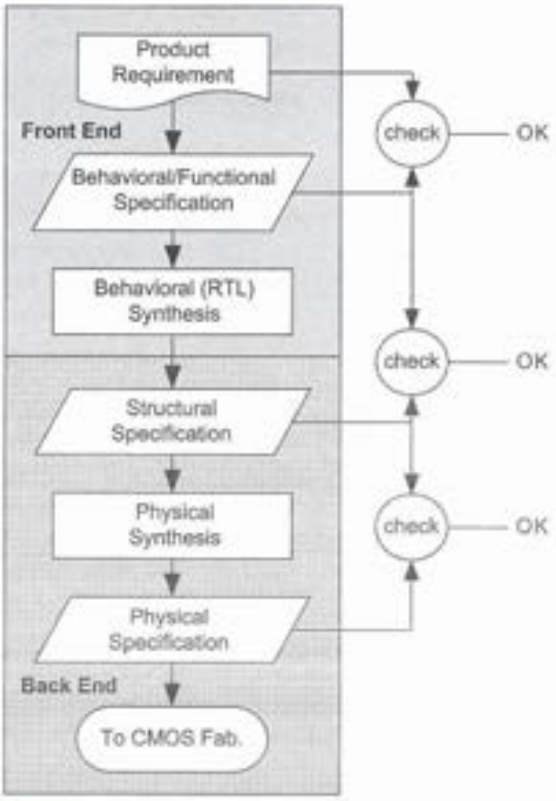


FIG 8.38 Generalized design flow

In Figure 8.38, the design has been partitioned into the *front end* stage at the behavioral level and the *back end* at the structural and physical levels. This is important because it illustrates a partitioning that is used to build *Application Specific Integrated Circuits* (ASICs). In an ASIC, the design can be developed at the HDL level and then passed to a company that completes the transition to an actual chip. In this way, the original design company does not have to invest the personnel or tools required to translate an HDL specification into a physical chip. Theoretically, in an ASIC flow, only a behavioral HDL needs to be designed and simulated (at the behavioral level). All subsequent operations can be completed by a third-party design service with only the final timing having to be verified by the back-end process. This is sometimes referred to as a "throw it over the wall" approach. While it works for moderately complex designs, the interaction between logic and layout is so important in more demanding circuits that such a flow becomes a schedule risk. Primarily, this occurs because the iteration time between logic design and physical placement takes too long when spread over two organizations. Multiple iterations are necessary because the prelayout timing estimates available to the HDL designer correlate poorly with the true postlayout timing because wire lengths are unpredictable before layout. Consider the case where the design cycle from logic to layout takes two hours when completed as an integrated task or one week if split into front-end and back-end tasks, as shown in the figure. If there are 100 iterations for the design, the integrated approach takes roughly 25 working days or five weeks, while the split approach takes two years (without vacations!). Having said this, companies are in business to make this approach work. If there are only ten iterations, the times are much more reasonable.

The next two sections summarize each of the tools required to perform the automatic transformation. We also will examine the verification tools required to guarantee the correctness of the transformation and look at specific design flows. Then, we will describe a

manual flow that is typical of a mixed-signal or RF design. Finally, we will outline a method of transforming directly from the behavioral to the physical level.

8.4.1 Behavioral Synthesis Design Flow (ASIC Design Flow)

At the behavioral level, the operation of the system is captured without having to specify the implementation. This level provides the most independence from implementation details and is the most dependent on the tool flow for a good design.

The most popular style of tools for behavioral synthesis are those that directly transform a behavioral RTL description to a structural gate-level netlist. A typical behavioral flow for an ASIC is shown in Figure 8.39. Tool suppliers include Synopsys, Cadence Design Systems, Mentor Graphics, and Synplicity.

8.4.1.1 Logic Design and Verification The design starts with a specification, which might be a text description or a description in a system specification language. The designer(s) convert this to an RTL behavioral description in an HDL such as Verilog, or VHDL. A set of test benches are then constructed and the HDL is simulated to verify the correct behavior as defined by the specification and product requirements. Typical interactive design environments and simulators include NC-Verilog/SystemC/VHDL or Desk-

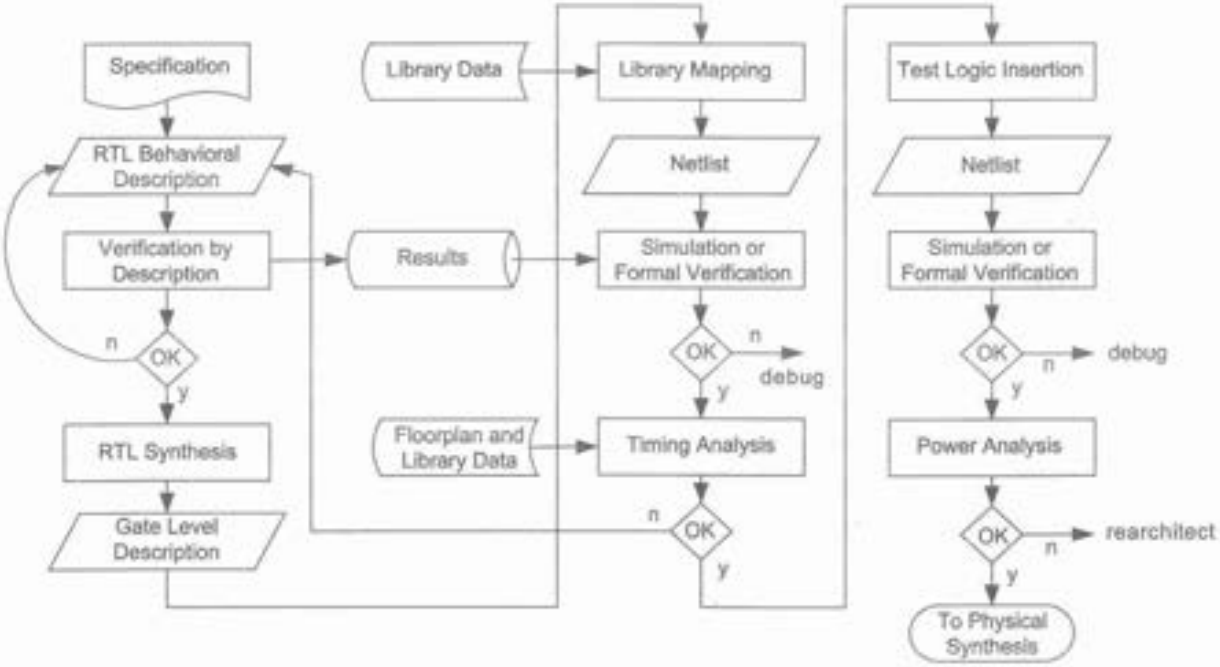


FIG 8.39 RTL synthesis flow

top Verilog/VHDL from Cadence Design Systems, VCS from Synopsys, ModelSim from Mentor Graphics and ActiveHDL from Aldec. Bear in mind that functional verification via simulation is usually carried out hierarchically. That is, after the overall architecture is defined, modules are successively built from the bottom up, verifying at each step. The design is iterated at this level until the correct behavior is evident. Test benches are covered further in Section 9.3.

Behavioral Verilog for an 8-bit implementation of the NCO previously introduced is presented below.

```
module nco #(parameter size = 8,
             counter_size = 16,
             table_size = 64)
            (input fclock, reset,
             input [counter_size-1:0] initial_phase, phase_increment,
             output [size-1:0] q);

   reg  [counter_size-1:0] phase;
   wire [size-3:0]         phase_part, inverted_adr, ROM_adr;
   wire [size-2:0]         ROM_data;
   wire [size-1:0]         wave_out;

   // numerically controlled oscillator
   // note that some constants are hardwired in the code below

   // phase counter
   always @(posedge fclock)
      if (reset) phase <= initial_phase;
      else phase <= phase + phase_increment;

   // add offset and determine ROM address
   assign phase_part = phase[counter_size-3:counter_size-8];
   assign inverted_adr = 7'3f - phase_part;
   assign ROM_adr = phase[counter_size-2] ? inverted_adr : phase_part;

   // look up data in ROM and negate if appropriate
   quarter_wave sine_table(ROM_adr, ROM_data);
   assign wave_out = phase[counter_size-1] ? -ROM_data : ROM_data;
   assign q = wave_out + 8'h80 + phase[counter_size-1];
endmodule
```

8.4.1.2 RTL Synthesis The next step is to *synthesize* the behavioral description. This involves converting the RTL to generic gates and registers, then optimizing the logic to improve speed and area. Other steps involved at this stage are state machine decomposition, datapath optimization, and power optimization. Typical products include Design Compiler from Synopsys, BuildGates from Cadence, and Synplify Pro from Synplicity.

The behavioral code is synthesized into the following structural code using a generic gate library.

```
module nco_struct(input          fclock, reset,
                  input  [15:0]  initial_phase, phase_increment,
                  output [7:0] q);

   wire [6:0] ROM_data;
   wire [15:0] phase_0;
   wire [5:0] nbus_5;
   wire [7:0] wave_out;
   wire [5:0] nbus_2;
   wire [15:0] nbus_1;
   wire [15:0] phase;

   xor i_7(wave_out[7], phase[15], 1'b0);
   xor i_6(wave_out[6], phase[15], ROM_data[6]);
   .
   .
   .
   xor i_2(wave_out[2], phase[15], ROM_data[2]);
   .
   .
   not i_10(nbus_5[2], phase[10]);
   not i_9(nbus_5[1], phase[9]);
   not i_8(nbus_5[0], phase[8]);
   generic_flip_flop phase_reg_15(.q(phase[15]), .d(phase_0[15]),
      .clk(fclock));
   .
   .
   .
   .
   quarter_wave sine_table(.addr(the_address), .q(ROM_Table));
   .
endmodule
```

The mapping to simple Verilog logic statements can be seen at the start of the program. Flip-flops are mapped to generic flip-flops that would implement a variety of flip-flop styles. The hierarchy is maintained, as illustrated by the call to the sine ROM.

8.4.1.3 Library Mapping *Library mapping* takes a generic HDL gate-level description and translates it to a netlist that specifies particular gates in the target library. This stage also maps predefined blocks such as memories to their appropriate descriptions. The following description is a portion of the mapped generic Verilog for the NCO shown above.

```
module nco_struct_mapped(input          fclock, reset,
                         input  [15:0] initial_phase, phase_increment,
                         output [7:0]  q);
   .
   .
   BUFX4 i_506(.A(n_355), .Y(q[7]));
```

```
    .
    MX2X1 i_00(.S0(reset), .B(initial_phase[15]), .A(nbus_1[15]),
       .Y(phase_0[15]));
    NAND2BX1 i_8(.AN(n_102), .B(n_101), .Y(n_104));
    XOR2X1 i_6(.A(phase[15]), .B(ROM_Table[6] ), .Y(n_103));
    .
    .
    DFFHQX1 phase_reg_0(.D(phase_0[15]), .CK(fclock), .Q(phase[15]));
    .
    .
    .
endmodule
```

Now the specific library elements are called out. `XOR2X1` is a 2-input normal-power XOR gate and `DFFHQX1` is a high-speed normal-power D flip-flop in the Artisan library used in the mapping.

8.4.1.4 Functional or Formal Verification We must now prove that the structural netlist performs the same function as the original behavioral HDL. Ideally, the netlist would be correct-by-construction, but ambiguities in HDLs sometimes cause the synthesizer to produce incorrect netlists from poorly written behavioral code. One verification strategy is to rerun the logic test benches and check that they produce exactly the same output for the behavioral and structural descriptions.

Another strategy is to use a *formal verification* program that compares the logical equivalence of the two descriptions. Formal verification tools are still maturing, but offer the advantage that they mathematically prove both descriptions have exactly the same Boolean functions [Anastasakis02]. In contrast, simulation only is as good as the choice of test vectors. Formality from Synopsys and Incisive Conformal from Cadence are examples of formal verifiers.

Other types of verification that can be run are semantic and structural checks on the HDL. An example of a semantic check would be ensuring that all bus assignments match in bit width, while an example of a structural check would include making sure all outputs are connected.

8.4.1.5 Static Timing Analysis At this point, the functional equivalence of the gate-level description and the original behavioral description has been established. Now the temporal requirements of the design have to be checked. For example, the adder may add, but does it add fast enough? At the behavioral level, clock cycle time is an abstract notion, but at the structural level, an actual cycle time has to be met by a particular set of gates. A *timing analyzer* is used to verify the timing.

The timing analyzer is a critical analytical tool in the arsenal of the modern CMOS digital designer. Timing can be verified in a cursory manner using a timing simulator, i.e., a simulator in which the actual gate timings are used rather than a cycle-based or unit delay simulator. While useful, this approach is usually neither complete nor rigorous and can take an extraordinary amount of time to run.

Static timing analysis, on the other hand, runs quickly and exhaustively evaluates *all* timing paths. The inputs to the timing analyzer at this point are derived from the basic timing of the library gates due to intrinsic gate delays and routing loads that can be either estimated statistically or derived from floorplanning data. (See Section 8.4.2.2 for a description of floorplanning.) Timing analyzers check for both *max-delay* (will all flip-flops meet their setup time at the required cycle time?) and *min-delay* (will any flip-flop violate its hold time?).

Static timing analysis can suffer from *false path* problems. Typical of this problem might be a reset line in a circuit that has many clock cycles to operate. The timing analyzer might report that it cannot complete in one cycle. The designer must manually flag such multicycle paths for the timing analyzer.

Typical timing analyzers include Pearl from Cadence and Pathmill and PrimeTime from Synopsys. The listing on the following page is a portion of a typical report for the NCO from a timing analyzer (this listing is from the Cadence BuildGates tool suite). The initial information specifies the required cycle time (11 ns) and then displays the calculated arrival time (5.89 ns). This results in what is termed a *slack time* of 5.11 ns. A table is then produced that lists the mapped Verilog instance, the signals that are changing and which direction the signals are toggling (by ^ and v characters for rising and falling), the cell type, the delay through that instance, and the arrival time. The required arrival time is also listed. Each line is examined for excessive timing and then normally the original Verilog is changed to improve delays through better pipelining or logic design. The synthesizer can also be directed to improve the timing. The sine table has been synthesized onto gates in this implementation.

8.4.1.6 Test Insertion Logic and registers are then inserted/modified to aid in manufacturing tests (see Section 9.6). Two basic techniques are used. One involves inserting scannable registers so that the state of a circuit can be set and monitored. Accompanying this option is a technique called *Automatic Test Pattern Generation* (ATPG), which is used to generate tests for a scannable design. The other technique, called *Built-In Self-Test* (BIST), modifies registers to allow *in situ* testing within the chip. Figure 8.40 shows the NCO after a test insertion program has run. The difference between the original design shown in Figure 8.13 and this design is the fact that blue *scan registers* have been inserted.

Typical commercially available test programs include DFT Compiler from Synopsys for scan insertion and Tetramax for ATPG. LogicVision markets Logic BIST and Memory BIST for built-in self-test.

8.4.1.7 Power Analysis The power consumption of the circuit is then estimated. Power consumption depends on the activity factors of the gates, which in turn depends on the inputs the chip receives. *Power analysis* can be performed for a particular set of test vectors by running a simulator and evaluating the total capacitance switched at each clock transition at each node. At this stage, if the power is too high, the design must return to the architectural level to rethink the solution. Commercial power analysis tools include PrimePower and Powermill from Synopsys.

```
Beginpoint: phase_reg_14/Q (^) triggered by leading edge of 'clock'
Required Time 11.00
- Arrival Time 5.89
= Slack Time 5.11
   Clock Rise Edge 0.00
   + Source Insertion Delay 0.20
   = Beginpoint Arrival Time 0.20
```

Instance	Arc	Cell	Delay	Arrival Time	Required Time
	fclock ^			0.20	5.31
phase_reg_14	CK ^ -> Q ^	DFFHQX1	0.58	0.78	5.89
i_34	A ^ -> Y ^	XOR2X1	0.78	1.57	6.68
sine_table	addr[3] ^	quarter_wave		1.57	6.68
sine_table/i_371	A ^ -> Y v	INVX1	0.45	2.02	7.13
sine_table/i_42	B v -> Y ^	NOR2X1	0.19	2.21	7.32
sine_table/i_44	AN ^ -> Y ^	NAND2BX1	0.16	2.37	7.48
sine_table/i_145	A0 ^ -> Y v	AOI211X1	0.10	2.47	7.58
sine_table/i_6	C0 v -> Y ^	OAI211X1	0.13	2.61	7.72
sine_table/i_484	A ^ -> Y ^	BUFX4	0.96	3.57	8.68
sine_table	q[0] ^	quarter_wave		3.57	8.68
i_120	B ^ -> Y v	NOR2BX1	0.22	3.79	8.90
i_7	AN v -> Y v	NOR2BX1	0.20	3.99	9.10
i_9	AN v -> Y v	NOR2BX1	0.21	4.20	9.31
i_20	B v -> Y ^	NAND2BX1	0.12	4.32	9.43
i_3	AN ^ -> Y ^	NAND2BX1	0.17	4.49	9.60
i_8	AN ^ -> Y ^	NAND2BX1	0.19	4.67	9.78
i_31	A ^ -> Y ^	XNOR2X1	0.28	4.96	10.07
i_504	A ^ -> Y ^	BUFX4	0.93	5.89	11.00
	q[6] ^		0.00	5.89	11.00

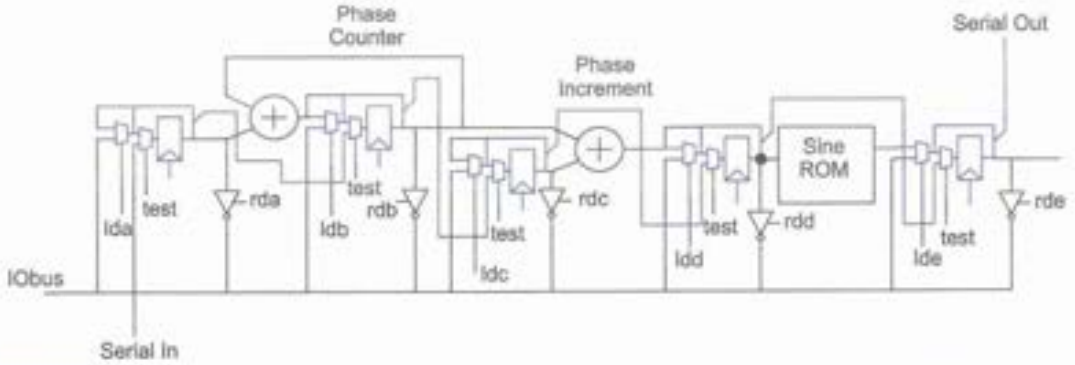


FIG 8.40 Scan register insertion for testing

Example

While automated tools are the solution to estimating power consumption in large circuits, sometimes you need a rapid, "back of the envelope" method of estimating power. How can we achieve this?

Solution: The following technique will give you a rough idea of the power dissipated in a circuit.

Step 1: Estimate the power in switched flip-flops and registers. Sum the clock input capacitance of all registers and, if possible, the internal switched capacitance. The latter is often hard to ascertain unless you have access to the internals of the cell library, but as a rough approximation, double the specified clock input capacitance. To this, add the capacitance of the clock routing. Again, conservatively, double the total clock load that you have previously calculated. The clock has an activity factor of 1. Now, given V_{DD} and the clock frequency, you can estimate the clock power using EQ (4.30). Thus, the power will be $2 \cdot 2 \cdot C_{clk} \cdot V_{DD}^2 \cdot f = 4C_{clk}V_{DD}^2f$.

Step 2: Estimate the power in the combinational circuits. Sum all input and output capacitances for all gates. This approximates the internal capacitance of the logic gates. To this we must add the routing capacitance. This is best done after some experience with the process and design tools you are using, but as a guide, use a number between 2 and 10. The activity factor has to be estimated. Again experience helps here, but 0.5 is very conservative. Thus, the total logic power will be roughly $5 \cdot 0.5 \cdot C_{logic} \cdot V_{DD}^2 \cdot f = 2.5C_{logic}V_{DD}^2f$.

These two steps are very approximate. After a bit of experience with a technology and design flow, you will be able to put your own numbers on the approximations.

8.4.1.8 Summary Apart from increasing design productivity, logic synthesis systems are useful for transforming between technologies. For instance, you might synthesize behavioral HDL onto multiple FPGAs and construct a prototype used to verify the operation of the circuit under real-world conditions. Then you can compile a single-chip version from the same HDL using a gate-array library.

8.4.2 Automated Layout Generation

Layout generation is the last step in the process of turning a design into a manufacturable database. It transforms a design from the structural to the physical domain. This step is sometimes called *physical synthesis* when the structural netlist is manipulated as the physical layout is generated. There are two main methods of automated layout generation, namely, standard cell place and route and the use of a software generator. Generators were discussed in Section 8.3.5, while this section emphasizes place and route tools used on most ASICs.

Figure 8.41 shows a standard place and route layout generation design flow. It begins with the structural netlist describing gates, flip-flops, and their interconnections. The

netlist might be provided in the *Design Exchange Format* (DEF) described in Section 8.9.4 or as a Verilog netlist like the one in Section 8.4.1.3. The placement tool also takes a standard cell library definition describing cell dimensions and port locations. Typically, this is in the *Library Exchange Format* (LEF), described in Section 8.11.3.

8.4.2.1 Placement The first step in Figure 8.41 is to place the standard cells. The key to automation of standard cell layouts is the use of constant-height, variable-width standard cells that are arrayed in rows across a chip, as shown in Figure 8.42. In contrast to SOG and gate array chips, standard cell chips can add application-specific custom blocks such as memories and analog blocks by allowing the standard cell rows to "flow" around the fixed-shape custom blocks. No separation has been shown between standard cell rows because routing takes place over the cells using multiple layers of metal. In older processes with two or three metal layers, a space between rows would be needed to allow routing. LEF summarizes the salient physical details of cells.

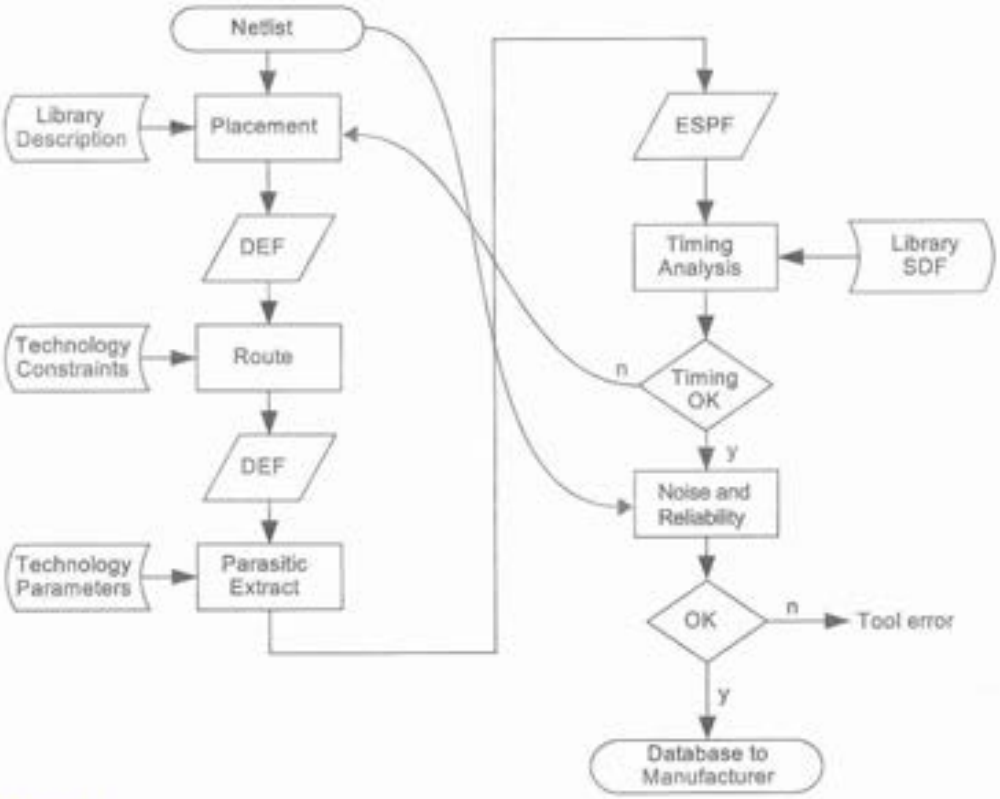


FIG 8.41 Standard cell place and route design flow

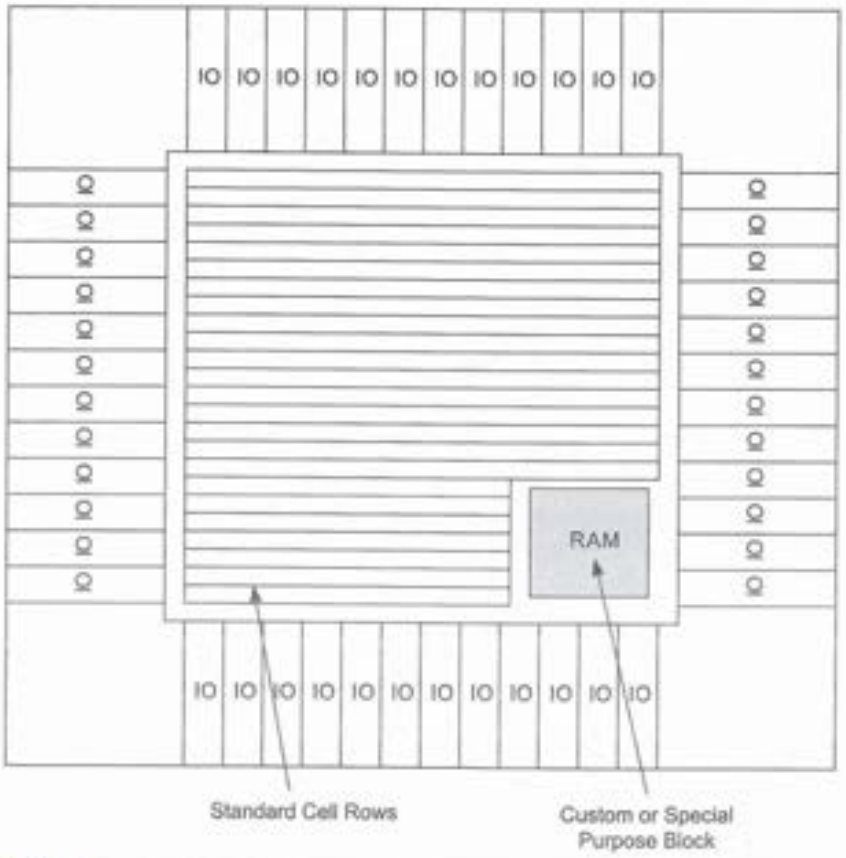


FIG 8.42 Standard cell chip layout

The objective of a simple placement algorithm is to minimize the length of wires. In *timing-driven placement*, the cost of wires is weighted to minimize delay on the critical paths. At the end of the placement phase, the cells have been fixed in position in the overall array. The placed design is saved in a standard format (e.g., DEF) for routing.

8.4.2.2 Floorplanning Increasingly a manual floorplanning step is required in the placement process. Rather than place a design "flat" (i.e., all cells at the same level of the hierarchy), modules are clustered in areas that are dictated by the need to communicate with other modules. Section 8.2.5 illustrated some floorplans for the software radio. This style of floorplanning might be completed prior to automatic placement.

8.4.2.3 Routing After placement of cells, the signal nets in the circuit need to be routed. Routing is normally divided into two steps: *global* routing and *detailed* routing.

A global router abstracts the routing problem to a notional set of abutting channels that cover the chip surface through which wires are routed. Routes are added to channels according to a cost function. Wires can be changed from channel to channel if the density of wires in a channel becomes too high. The detailed router places the actual geometry required to complete signal connections. Over time, a selection of detailed routers have been developed to automatically route signals. Older routers constrained signals to a grid of tracks, but newer *gridless* routers are more flexible for variable pitch wires. Moreover, they allow easy interface to foreign cells that may have I/O pin locations that are not on any specific routing grid. Routers also can route over the top of cells. LEF definitions are used to indicate obstructions on various layers in cell definitions. Advanced routers take into account manufacturability concerns such as redundant vias (more that one via inserted when space is available) and adjustable spacing (to separate wires and reduce coupling when there is room).

In the example of the flow shown in Figure 8.41, the router uses a technology file to specify routing layers and pitches for the process technology. It writes the results to another DEF file.

8.4.2.4 Parasitic Extraction The placed and routed design is then passed to the circuit *parasitic extractor*. In the example shown in Figure 8.42, the placed and routed design is provided to the extractor in DEF format and the output is an *Extended Standard Parasitic Format* (ESPF), *Reduced Standard Parasitic Format* (RSPF), or *Standard Parasitic Exchange Format* (SPEF) that describes the *R*'s and *C*'s associated with all nets in the layout. The extractor uses another technology file defining the interlayer capacitances and layer resistances. The formats are covered in more detail in Section 8.11.6.

The capacitance extractor can be a 2D, 2.5D, or 3D extractor. Two-dimensional (2D) extractors look at a cross-section assuming wires extend uniformly outside the section. A 2.5D extractor uses lookup tables to more accurately estimate capacitance near nonuniformities. A 3D extractor solves Maxwell's equations in three dimensions to precisely determine capacitance of complex geometries. 3D extraction used to be prohibitively time-consuming, but new statistical algorithms, such as those in QuickCap from Magma Design, deliver good accuracy with faster runtimes.

8.4.2.5 Timing Analysis Static timing analysis is now rerun with the actual routing loads placed on the gates. This is usually the bottleneck in the design process as the full reality of a physical realization is apparent. Multiple iterations of synthesis and placement & routing are usually necessary to converge on timing requirements.

Additionally, if possible (especially where dynamic circuits are used), a transistor-level timing simulation should be run. While this cannot usually be achieved using a SPICE-based simulator, a variety of transistor level simulators with "almost SPICE accuracy" have been in use since the late 1970s. These currently have the capacity to do whole-chip simulations at the transistor level, but at somewhat reduced transistor modeling accuracy. Pathmill and Nanosim from Synopsys and UltraSim from Cadence are examples of current simulators of this type.

8.4.2.6 Noise, V_{DD} Drop, and Electromigration Analysis Analyses are now run to check noise, IR drop in supply lines, and electromigration limits. Noise analysis is run to evaluate crosstalk due to interlayer routing capacitance. For static circuits, aggressor nodes can alter the timing of victim nodes but not their function, as described in Section 4.5.4. For poorly designed dynamic circuits, crosstalk can cause irrecoverable errors. SignalStorm, ElectronStorm, and VoltageStorm from Cadence are examples of such tools.

8.4.2.7 Timing-driven Placement The trouble with a place-then-route strategy is that after the layout is completed, the parasitic routing capacitance is extracted and the timing analysis is done to estimate timing. The timing is not known until the physical layout is complete. If timing problems are found, the cycle has to be repeated with some kind of constraint placed on the problematic paths. With complex designs this quickly gets out of control, to the point where changing something on one iteration could undo something fixed on a previous iteration. There are stories of designs that never were completed because of this problem.

The solution is to use a technique called *timing-driven placement*, which takes into account the timing (speed) of the circuit as cells are placed. Cells on critical paths are given priority to minimize wire delay. This approach, illustrated in Figure 8.43, has been successful and often results in a one-pass approach for many designs.

8.4.2.8 Clock-tree Routing Central to modern high-speed designs is the clock distribution strategy. In Section 12.5.4, a number of these approaches are explained. To minimize skew, it is often best to route the clock and its buffers before the main logic placement and routing is completed. This task is performed with a *clock tree router*. Any of the clocking approaches described in Chapter 7 can be implemented in this manner.

8.4.2.9 Power Analysis Power estimation can be repeated for the extracted design now that real wire capacitances are available. Similar techniques to those used during RTL synthesis are used.

8.4.3 Mixed-signal or Custom-design Flow

In the previous section, we described a flow that would be used for a purely digital chip in which the procedure for converting from HDL to layout is highly automated. This flow offers high productivity for most large digital chips with moderate performance requirements. But what of smaller analog, RF, and high-speed digital sections of a chip? For these sections we use a custom-design flow, which is shown in Figure 8.44.

The designer begins by drawing a schematic (or possibly writing a netlist). An electrical rule check (ERC) verifies port connectivity and checks for unconnected inputs or outputs—the kind of simple connectivity errors that can occur easily in a manually drawn schematic. When the schematic is deemed correct, circuit simulation is then carried out using a SPICE-type simulator to verify DC, AC, transient, noise, and/or RF performance.

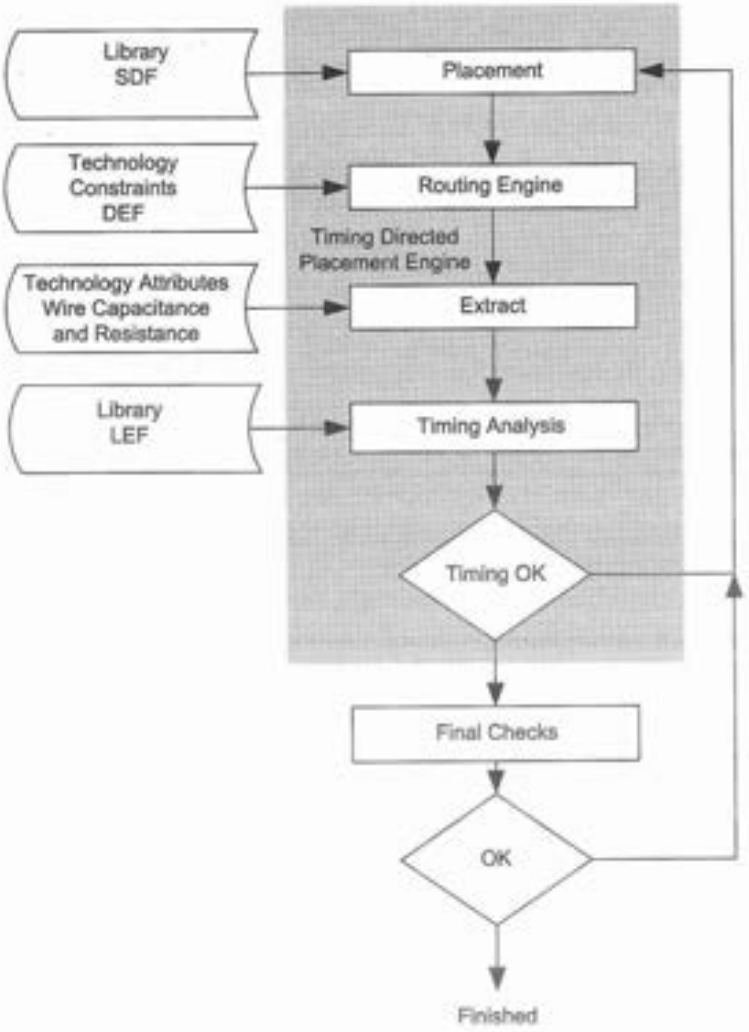


FIG 8.43 Timing directed placement design flow

Once the circuit behavior of the module has been verified, the layout can commence, starting with the floorplan. Floorplanning can be an iterative process that is refined as actual module sizes and critical paths become known. Custom layout is a very time-consuming task; for example, a large microprocessor can keep a hundred mask design technicians busy for two years. Automating noncritical parts of the layout is essential for productivity. When the layout for the module is complete, a layout circuit

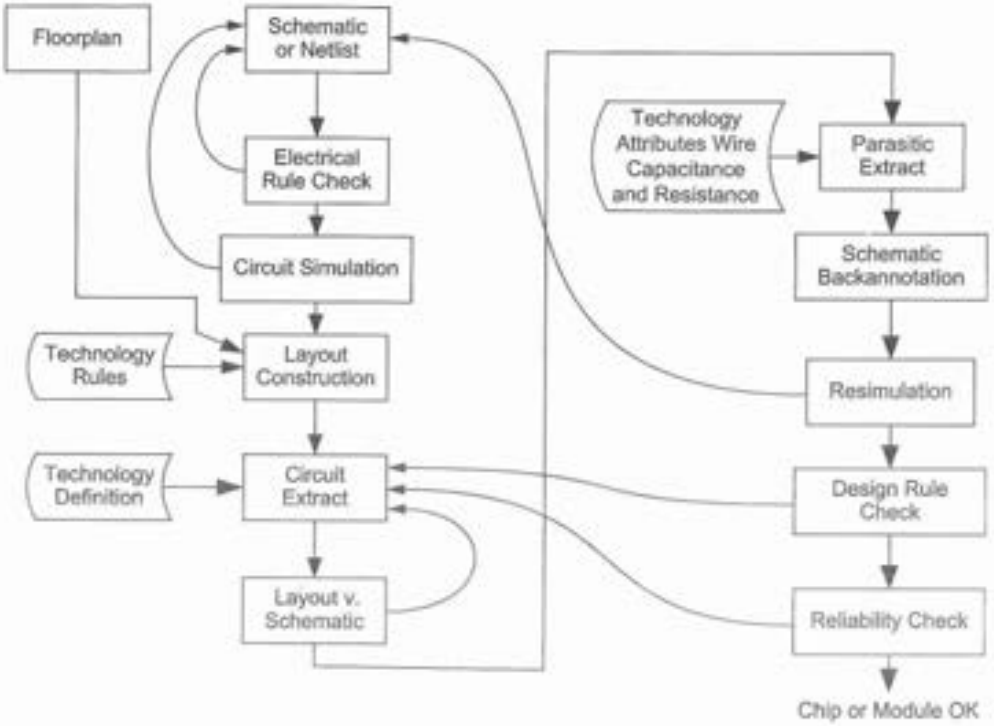


FIG 8.44 Mixed-signal or custom-design flow

extractor is invoked to determine the connectivity of primitives (MOS and bipolar transistors, diodes, resistors, capacitors, inductors) in the layout using rules like those illustrated in Section 3.5.2.

In the next step, the extracted netlist is compared to the schematic using a graph isomorphism program to determine whether the two netlists are identical in connectivity. This proceeds by assigning primitives to the nodes of a graph and the connections to the arcs in the graph. Graph coloring based on the connectivity and circuit parameters (i.e., transistor type, width, and length) determines the extent of the match. Once connectivity equivalence has been determined, each primitive attribute is checked for equivalence (i.e., capacitor or resistor value, transistor *W/L*). Discrepancies are reported to the user. Graphical feedback may be provided to help the designer find the source of any mismatch. This step is commonly called *layout versus schematic* (LVS).

Once the structural-to-physical equivalence has been established, the parasitic extract is completed. This adds the parasitic routing capacitance and resistance to the original primitive elements. In general, inductors are not extracted, but are dealt with by cookie cutting the inductor out of the layout and substituting a previously generated physical model. This is sometimes called *macro substitution*. The parasitic capacitance and resis-

tance can be *back annotated* onto the schematic and the complete circuit resimulated. It must be pointed out that this step is extremely important. Matching simulated behavior to real device behavior is of critical importance in being able to accurately predict performance. It is too late when the circuit has been built!

The module layout can then be design-rule checked (DRC). Alternatively, this step can be completed just as the layout is completed. Normally, the AC performance is more important than tweaking the last design-rule error because running DRC on a circuit that does not meet performance goals is a waste of time.

Following this, a set of manufacturability verification steps needs to be completed. These can be manual or automated. In common with the standard cell design flow, power bus widths should be checked to ensure that they comply with metal migration and IR drop constraints. Power consumption can be found directly from circuit simulation. Adequate substrate and well contacts should be present in a bulk CMOS design, and all external I/O must be guard-ringed. At this stage, a check can also be made for substrate noise injection from digital to analog circuits. SubstrateStorm from Cadence performs this task.

This process can be completed hierarchically to build up large modules. Usually, the ultimate limitation comes from trying to simulate vast numbers of transistors accurately in SPICE. A variety of fast transistor-level simulators have been developed to deal with this problem, although there is always some upper limit to what can be simulated at the desired accuracy.

8.4.4 Programmed Behavioral Synthesis

An alternate class of behavioral synthesizer directly synthesizes blocks in a fashion reminiscent of silicon compilers. This type of synthesizer takes a fairly targeted application space (such as digital signal processing) and constructs a logic, circuit, and physical description based on a high-level input description. An example of such a system is the design system used at the Berkeley Wireless Research Center, which takes high-level descriptions defined in Simulink, and automatically translates them through a number of steps to a mask layout. The basis of this system is a set of predefined modules available in the Simulink library that have targets in the physical domain. This type of flow is illustrated in Figure 8.45. The translation between behavioral and physical layers of abstraction occurs directly. RTL and structural descriptions can also be generated as a side product to aid in verification. For example, a multiplier for complex numbers could be specified as a Simulink block. An instance of this multiplier would trigger the automatic construction of the appropriate RTL, gate, and mask-level description following a preprogrammed algorithm.

8.5 Design Economics

It is important for the IC designer to be able to predict the cost and the time to design a particular IC or sets of ICs. This can guide the choice of an implementation strategy. This section will summarize a simplified approach to estimate these values.

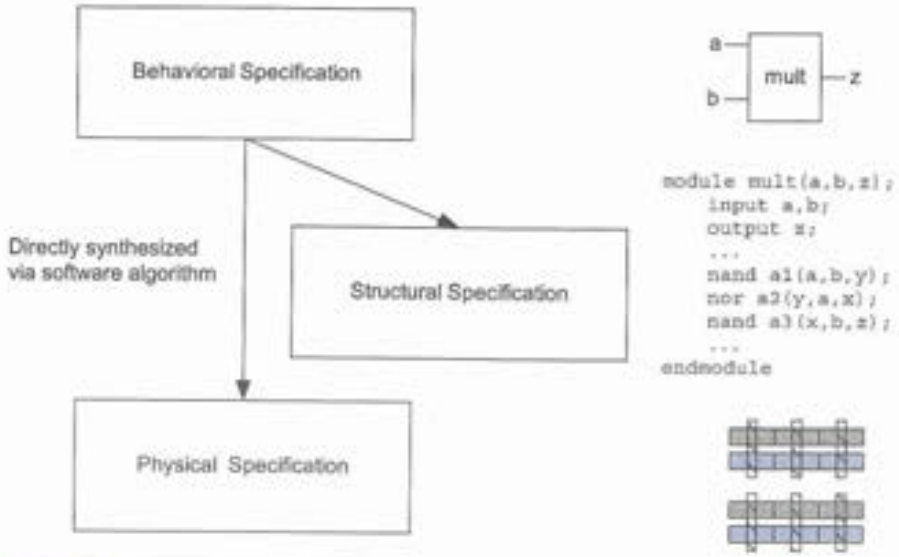


FIG 8.45 Direct behavioral-to-physical translation

In this section, we will concentrate on the cost of a single IC, although you should consider the overall system when making such decisions. System-level issues such as packaging and power dissipation can affect the cost of an IC.

The selling price S_{total} of an integrated circuit may be given by

$$S_{total} = C_{total} / (1-m) \tag{8.3}$$

where

C_{total} is the manufacturing cost of a single IC to the vendor
m is the desired profit margin

The margin has to be selected to ensure a profit after overhead (G&A) and the cost of sales (marketing and sales costs) have been considered.

The costs to produce an integrated circuit are generally divided into the following elements:

- Non-recurring engineering costs (NREs)
- Recurring costs
- Fixed costs

8.5.1 Non-recurring Engineering Costs (NREs)

Non-recurring engineering costs are those that are spent once during the design of an integrated circuit. They include

- engineering design cost E_{total}
- prototype manufacturing cost P_{total}

These costs are amortized over the total number of ICs sold. F_{total}, the total non-recurring cost, is given by

$$F_{total} = E_{total} + P_{total} \tag{8.4}$$

The NRE costs can be amortized over the lifetime volume of the chips. Alternatively, the non-recurring costs can be viewed as an investment for which there is a required rate of return. For instance, if \$1M is invested in NRE for a chip, then \$10M has to be generated for a rate of return of 10.

8.5.1.1 Engineering Costs The cost of designing the IC E_{total} hopefully will happen once during the chip design process. The costs include:

- personnel cost
- support costs

The personnel costs might include the labor for

- architectural design
- logic capture
- simulation for functionality
- layout of modules and chip
- timing verification
- DRC and tapeout procedures
- test generation

The support costs amortized over the life of the equipment and the length of the design project include

- computer costs
- CAD software costs
- education or re-education costs

Costs can be drastically reduced by reusing modules or acquiring fully completed modules from an intellectual property vendor. As a guide the per annum costs might break down as follows (these figures are in US dollars for engineers in the USA circa 2004):

Salary	\$50–\$100K
Overhead	\$10–\$30K
Computer	\$10K
CAD Tools (digital front end)	\$10K
CAD Tools (analog)	\$100K
CAD Tools (digital back end)	\$1M

The cost of the back-end tools clearly must be shared over the group designing the chips.

8.5.1.2 Prototype Manufacturing Costs These costs (P_{total}) are the fixed costs to get the first ICs from the vendor. They include

- the mask cost
- test fixture costs
- package tooling

The photo-mask cost is proportional to the number of steps used in the process. Mask costs increase as the process dimensions are reduced, so although newer, smaller processes generally have increased mask costs, masks on the metallization layers can be less expensive than on the lower layers. A mask currently costs between \$500 and \$30,000 or more. A complete mask set for 130 nm CMOS costs in the vicinity of \$500K to \$1M. These prices will increase as process linewidths are reduced.

A test fixture consists of a printed wiring board probe assembly to probe individual die at the wafer level and interface to a tester. Costs range from \$1000 to \$50,000, depending on the complexity of the interface electronics.

If a custom package is required, it may have to be designed and manufactured (tooled). The time and expense of tooling a package depends on the sophistication of the package. Where possible, standard packages should be used.

An economical way of prototyping chips is to use a multi-project reticle that combines a number of different chip designs onto one mask set. Thus, if there were 200 sites available on a mask set and 20 projects were implemented, each project would get 10 die per wafer and the mask cost per project would be $1/20^{th}$ of the cost of a complete mask set. This kind of service is provided by many of the silicon vendors and also MOSIS. For modest technology this can be quite cheap (~ \$1000 per mm^2 for 0.6 μm). Some commercial users worry about protection of intellectual property when they share a mask set.

Example

You are starting a company to commercialize your brilliant research idea. Estimate the cost to prototype a mixed-signal chip. Assume you have seven digital designers, three analog designers, and five support personnel and that the prototype takes two fabrication runs and two years.

Solution: The seven digital designers will cost 7 • (\$70K + \$30K + \$10K + \$10K) = \$840K. The three analog designers will cost 3 • (\$100K + \$30K + \$10K + \$100K) = \$720K. The five support personnel cost 5 • (\$40K + \$20K + \$10K) = \$350K. One fabrication run with the back-end tools will cost \$2M. Thus, the cost is \$3.91M per year with one fab run. The total predicted cost here is nearly \$8M.

You may see ways to improve this. Clearly, you can reduce the number of people and the labor cost. You might reduce the CAD tool cost and the fabrication cost by doing multiproject chips. However, the latter approach will not get you to a pre-production version, because issues such as yield and behavior across process variations will not be proved. Figure 8.46 shows the breakdown of the overall cost.

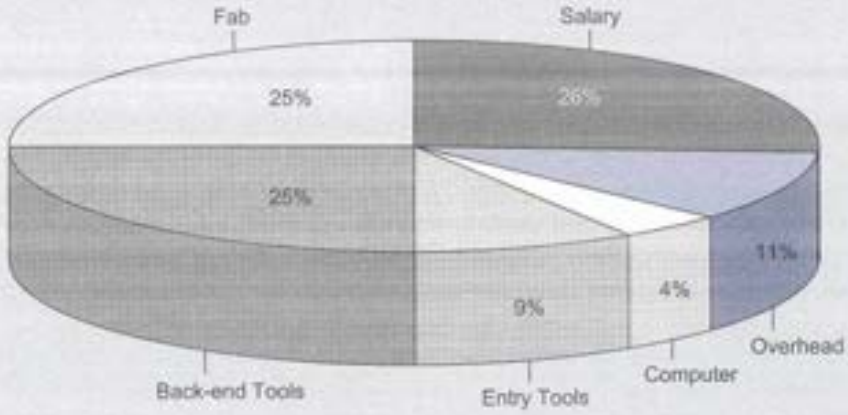


FIG 8.46 Pie chart showing prototyping costs for a mixed-signal IC

8.5.2 Recurring Costs

Once the development cost of an IC has been determined, the IC manufacturer will arrive at a price for the specific IC. A few large companies such as Intel, TI, STMicroelectronics, Toshiba, and IBM have in-house manufacturing divisions. Many fabless semiconductor companies outsource their manufacturing to a silicon foundry such as TSMC, Hitachi/

UMC, IBM, LSI Logic, or ST. This is a recurring cost; that is, it recurs every time an IC is sold. Another component of the recurring cost is the continuing cost to support the part from a technical viewpoint. Finally, there is what is called "the cost of sales," which is the marketing, sales force, and overhead costs associated with selling each IC. In a captive situation such as the IBM microelectronics division selling CPUs to the mainframe division, this might be zero.

The IC manufacturer will determine a part price for an IC based on the cost to produce that IC and a profit margin. The margin generally falls as the volume increases. An expression for the cost to fabricate an IC is as follows:

$$R_{total} = R_{process} + R_{package} + R_{test} \tag{8.5}$$

where

$R_{package}$ = package cost

R_{test} = test cost—the cost to test an IC is usually proportional to the number of vectors and the time to test.

$$R_{process} = W / (N \cdot Y_w \cdot Y_{pa}) \tag{8.6}$$

where

W = wafer cost (\$500–\$3000 depending on process and wafer size)

N = gross die per wafer (the number of complete die on a wafer)

Y_w = die yield per wafer (should be ~70%–90+% for moderate sized dice in a mature process)

Y_{pa} = packaging yield (should be ~95%–99%)

If a die has area A and is fabricated on a wafer with radius r, the gross number of dice per wafer is

$$N = \pi\left[\frac{r^2}{A} - \frac{2r}{\sqrt{2A}}\right] \tag{8.7}$$

where the second term accounts for wasted area around the edges of a circular wafer.

The packaging yield is the number of chips that pass testing after the wafer has been diced and the parts packaged. The die yield is affected by defects randomly distributed around the wafer. The probability of a random defect causing a particular die to fail depends on the size of the die A and average number of defects per unit area D. If defects obey a Poisson distribution, Y_w may be given by [Seeds67]

$$Y_w = e^{-AD} \tag{8.8}$$

For small dice ($AD << 1$), Y_w is nearly 1 and $R_{process}$ grows linearly with A. For large dice ($AD >> 1$), Y_w drops off rapidly because most chips will have defects and $R_{process}$ grows exponentially with A.

8.5.3 Fixed Costs

Once a chip has been designed and put into manufacture, the cost to support that chip from an engineering viewpoint may have a few sources. *Data sheets* describing the characteristics of the IC have to be written, even for application-specific ICs that are not sold outside the company that developed them. From time to time, *application notes* describing how to use the IC may be needed. In addition, specific application support may have to be provided to help particular users. This is especially true for ASICs, where the designer usually becomes the walking, talking, data sheet and application note. Another ongoing task may be failure or yield analysis if the part is in high volume and you want to increase the yield.

As a side comment, every chip or test chip designed should have accompanying documentation that explains what it is and how to use it. This even applies to chips designed in the academic environment because the time between design submission and fabricated chip can be quite large and can tax even the best memory.

Example

Suppose your startup seeks a return on investment of 5. The wafers cost \$2000 and hold 400 gross die with a yield of 70%. If packaging, test, and fixed costs are negligible, how much do you need to charge per chip to have a 60% profit margin? How many chips do you need to sell to obtain a 5-fold return on your \$8M investment?

Solution: $R_{total} = R_{process} = \$2000/(400 \cdot 0.7) = \$7.14$. For a 60% margin, the chips are sold at $\$7.14 / (1 - 0.6) = \$17.86$ with a profit of \$10.72 per unit. The desired ROI implies a profit of $\$8M \cdot 5 = \$40M$. Thus, $\$40M / \$10.72 = 3.73M$ chips must be sold. Clearly, a large market is necessary to justify the investment in custom chip design.

8.5.4 Schedule

At the outset of a system design project involving newly designed ICs, it is important to estimate the design cost and design time for that system. Estimating the cost can help you determine the method by which the ICs will be designed. Estimating the schedule is essential to be able to select a strategy by which the ICs will be available in the right time and at the right price. This second task is usually the least well specified and requires some experience.

If we assume that fixed costs are kept reasonable and that for a given IC size, $R_{process}$ is constant, the variables left in determining the cost of an IC are E_{total}, the engineering design cost, and P_{total}, the prototype manufacturing cost. P_{total} depends on the way in which the IC is implemented. We examined a variety of strategies for the design of CMOS systems earlier in the chapter. The fixed costs of prototyping P_{total} are relatively constant, given an implementation technology. The engineering costs depend on the complexity of the chip, the design strategy, and the amount of sustaining engineering needed. Usually, the design and verification engineering costs dominate. For this reason, it is important to be able to estimate a schedule for the design of an IC and then manage the available resources to bring the project to a successful conclusion.

Increased engineering effort can reduce the size of the die, which reduces $R_{process}$. Hence, it is important to be able to trade off the reduction in die cost with the increase in engineering effort. Opinions vary, but it is usually best to get a product first to market and then shrink the die when the product becomes successful. Optimizing without market feedback is usually a recipe for loss of market share or even failure to gain any market share at all.

[Paraskevopoulos87] suggests a number of fairly obvious methods for increasing productivity, thereby improving schedules:

- using a high-productivity design method
- improving the productivity of a given technique
- decreasing the complexity of the design task by partitioning

A final caution: Adding people to a project that is already late tends to make it later [Brooks95].

8.5.5 Personpower

To estimate the schedule, you must have some idea of the amount of effort required to complete the design. As we have seen, typical IC projects will involve the following tasks

1. architectural design
2. HDL capture
3. functional verification
4. place and route
5. timing verification, signal integrity, reliability verification
6. DRC and tapeout procedures (ERC, LVS, mask generation)
7. test generation

While some researchers have attempted to derive analytical formulae for productivity, the best predictor of design schedule for a team is previous performance. Design time for a

Example

While it is hard to predict the design and test time for a chip, we can at least identify the main tasks and corresponding fixed periods in a chip design project. A representative Gantt chart is shown in Figure 8.47 for a project running over one year. The logic design time is shown as 12 weeks, which would be appropriate for an extremely simple chip. Double this time would be representative of moderately complex digital chips. The fixed times tend to be the fabrication time and packaging time, which are shown to be 10 weeks in the example. The design, debug, and test times will expand or contract to fit the complexity of the chip. And, if you are meticulous and lucky, you will not have to respin the chip.

ID	Task Name	Start	Finish	Duration
1	Specification	1/1/2004	1/28/2004	4w
2	Digital Design	1/29/2004	4/21/2004	12w
3	Place and Route	4/22/2004	6/16/2004	8w
4	Fabrication	6/17/2004	8/11/2004	8w
5	Packaging	8/12/2004	8/25/2004	2w
6	Lab Test	8/26/2004	10/20/2004	8w
7	Respin	10/21/2004	11/17/2004	4w
8	Lab Test	11/18/2004	12/29/2004	6w

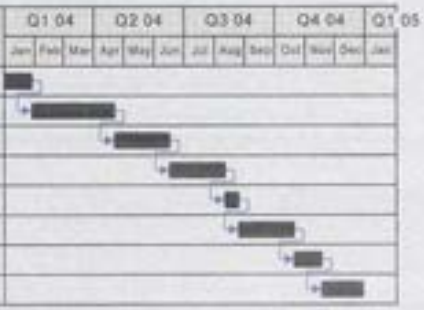


FIG 8.47 Gantt chart for simple chip

given team can be improved by design reuse or component-based design. It would seem that the time to design is proportional to the number of "modules" that are in the design raised to some power. That is, a four-module design is more than four times as complex as a single-module design. A module in this instance refers to a significant section of a chip such as a microprocessor, serial interface, or special functional unit.

Normally, projects are schedule-driven. In this case, it is important to make maximal use of design aids to meet the required schedule. Of importance is the cycle time of the so-called "edit-compile-debug" loop: i.e., the time it takes to make a change to the HDL; synthesize, place, and route it; and have a timing-verified final design. This can depend strongly on the efficiency of the design tools used, but if it is more than a day, design productivity can suffer. Ideally, the cycle is a few hours so that multiple bugs can be fixed each day.

Broadly speaking, schedules on the order of 18–24 months for a completely new chip seem to fit current average-complexity chips and state-of-the-art tools. For respins to slightly differentiate products, this can be reduced to six months or less, but there are certain fixed times such as IC fabrication and packaging that set hard limits on the complete design cycle time. Of course, for technologies such as FPGAs, design turnaround can be minutes (which is why FPGA verification is so important to ASIC or custom IC designs).

New microprocessors seem to take three to five years, and most experience one or more schedule slips.

8.5.6 Project Management

Project management is the overall supervision of the project. Tasks include making certain sufficient resources are available at the appropriate time, ensuring communication between different groups assigned to the project, and summarizing progress and risks to management. The development of processes for the conception, design, and ultimate manufacture of products is also the purview of the project manager.

There are two main ways to manage a chip design. The first is what might be called the rapid prototyping approach that is typical of startup companies, where a full-time project manager may be a luxury (and probably is more aptly named "seat-of-the-pants project management"). In this approach, a time goal is set and the workload is set to fit the time available. It is vital to rapidly get to the point where a prototype of the design is working—in essence, the skeleton—and the meat (detail) is gradually added. This can be risky.

The more conventional approach, which is appropriate for large companies and the military, is to preplan everything, estimating task times and putting these into a project planning tool. This approach, while necessary for large groups, tends to be feature-driven and rarely delivers products in shorter time scales than the rapid prototyping approaches. It is suitable when the tasks are well-defined and have been done before (then you know what the task times should be). The approach is stable and, depending on the team, often delivers products within budget and on time.

8.5.7 Design Reuse

Rarely is an IC designed as a single event. Rather, companies wish to amortize the development effort of a particular IC over several generations of products. This normally means that the design has to be transferred between several different processes. When design was mainly manual and at the mask level, a great deal of effort was expended on techniques to allow porting of designs between processes with the minimum of human intervention. Techniques used here include the use of symbolic layout methods and mask resizing software.

With the emergence of cell-based design, design migration falls into two steps:

- Acquiring or building a standard cell library in the new technology
- Retargeting the HDL description to the new cell library

The design and test generation does not have to be redone, although timing analysis and regression test bench simulation should definitely be completed.

In design flows where these steps cannot be followed, strict use of structured design techniques and software generator technologies can markedly improve porting times. Maintaining accurate and clear documentation will alleviate many problems downstream.

With the maturation of cell-based design, especially standard cell libraries and the use of hardware description languages, the notion of virtual components has become important as a method of transferring and reusing designs. Virtual components on an IC are notionally the same as discrete ICs used on a printed wiring board design. Each component has precisely defined behavior and a well-defined interface represented by a set of I/O pins and corresponding specifications for loading, setup and hold times, and delays. Components can be relatively simple or as complex as a RISC processor, MPEG decoder, or Wireless LAN modem. Virtual components can be classified as *hard*, *firm*, or *soft*. A hard module is normally defined at the mask level in a particular process. Thus, it will have a fixed floorplan, size, and a well-known set of timing parameters. A firm block will normally have a specific or generic netlist that describes each gate or register that must be used in the design (i.e., a 3-input NAND gate of normal power). This allows the design to be ported to multiple processes purely by netlist translation. The timing is dictated by the process and the final physical placement, however. A soft block is normally defined at the RTL level in the HDL. This captures the function of the block, but the detailed implementation is left to automated tools. Again, timing is dependent on the specific implementation. The Virtual Socket Interface Alliance monitors and encourages standards governing the implementation and use of virtual components.

8.6 Data Sheets and Documentation

A data sheet for an IC describes what it does and outlines the specifications for making the IC work in a system, such as power supply voltages, currents, input setup times, output delay times, and clock cycle times. The data sheet also includes package and pinout details.

A good habit to acquire is that of compiling a data sheet for any chip you might design. Not only is it the interface between the chip designer and the board-level designer, but also it is the interface to other members of the design team. In particular, it is good practice and is mandatory in industry to compile the data sheet for the chip and give it to the ultimate customer before the chip is fabricated. This prevents many undesirable scenarios that can arise when a perfectly designed chip meets a perfectly designed system. In this section, an outline of a typical data sheet will be reviewed by way of example.

8.6.1 The Summary

A summary of the chip includes the following details to orient the user:

- the designation and descriptive name of the chip
- a concise description of what the chip does
- a features list (optional for an internal product—but good for your ego!)
- a high-level block diagram of the chip function

8.6.2 Pinout

The pinout section should contain a description of the following pin attributes to document the external interface of the chip:

- name of the pin
- type of pin (i.e., whether input, output, tristate, digital, analog, etc.)
- a brief description of the pin function
- the package pin number

8.6.3 Description of Operation

This section should outline the operation of the chip as far as the user of the chip is interested. Programming options, data formats, and control options should be summarized.

8.6.4 DC Specifications

This section communicates the power dissipation and required voltages for the chip to correctly operate. The absolute maximum ratings should be stated for:

- supply voltage
- pin voltages
- junction temperature

The style of each I/O (i.e., TTL, CMOS, LVDS, ECL) should be summarized and the following DC specifications should be given over the operating range (temperature and voltage—i.e., mins and maxes):

- V_{IL} and V_{IH} for each input
- V_{OL} and V_{OH} for each output (at a given maximum drive current level)
- the input loading for each input
- quiescent current
- leakage current
- power-down current (if applicable)
- any other relevant voltages and currents

8.6.5 AC Specifications

The following timing specifications should be presented:

- setup and hold times on all inputs
- clock (and all other relevant inputs) to output delay times
- other critical timing such as minimum pulse widths

This data should be tabulated in table form and supported by a timing diagram where necessary. This is probably the most important section and an area where data provided ahead of the chip fabrication will aid the board designer. Designs are frequently snagged—for instance, when chip designers assume infinitely fast external memories and do not allow enough time between outputs changing and the next rising edge of the clock.

8.6.6 Package Diagram

A diagram of the package with the pin names attached should be supplied.

8.6.7 Principles of Operation Manual

Although the data sheet provides enough data to familiarize a user of a particular chip with the device, it is good practice to provide a Principles of Operation manual for internal users that have to test the chip or build support systems.

8.6.8 User Manual

A User Manual should also be provided. This is designed for use outside the group that designed the chip and can be a "cut down" version of the Principles of Operation manual.

8.7 Closing the Gap between ASIC and Custom

OPTIONAL

High-performance microprocessors and digital signal processing chips are typically designed in a custom methodology in which designers use a wide variety of circuit techniques and carefully tune all of the critical parts of the chip to achieve high performance at the expense of great design effort. Many other logic chips are built in a high-productivity ASIC methodology in which HDL code is synthesized onto a standard cell library and automatically placed and routed, as was described in Sections 8.4.1-8.4.2.

In a 180 nm process, custom microprocessors commonly operate at 1–2 GHz. In contrast, high-performance ASICs in comparable processes tend to operate at 200–350 MHz. Overall, custom design has consistently offered a 3–8x frequency advantage over ASIC designs. This is a remarkable difference, comparable to nearly 4–5 process generations of transistor improvement. [Chinnery02] offers a fascinating comparison of the two methodologies, explaining why custom flows offer so much higher performance and how ASIC designs can approach custom while maintaining high productivity. The remainder of this section summarizes the study's conclusions.

Measuring delays in terms of an FO4 inverter eliminates the process dependency and helps predict performance in the future. Overall, ASICs typically have cycle times of 50–100 FO4 inverter delays. High-performance ASICs such as the 520 MHz iCORE from STMicroelectronics can approach 25 FO4 delays per cycle. In comparison, the Pentium II and III operate at 20–24 FO4 delays, Alpha microprocessors at 13–16 FO4 delays, and the Pentium 4 at about 10 FO4 delays per cycle.

Chinnery identifies seven primary factors contributing to the custom performance advantage. Table 8.5 compares the effect of each of these factors. Custom design offers a large advantage over poor ASIC design in which performance is a secondary concern. Best practice ASIC methodologies can close much of the gap while still delivering far higher productivity than custom design. Poor is a relative term; often, economics demands fastest time to market rather than highest clock rate. Note that some factors affect only parts of the design so they cannot be simply multiplied to give the performance difference. The remainder of this section explores each of the factors.

Table 8.5 Relative performance of custom vs. ASIC design methodologies

Factor	vs. Poor ASIC	vs. Best Practice ASIC
Microarchitecture (e.g., pipelining)	1.8x	1.3x
Sequencing overhead: elements, skew, time borrowing	1.45x	1.1x
Circuit families (e.g., domino)	1.4x	1.2x
Logic design	1.3x	1.0x
Cell design, cell sizing, and wire sizing	1.45x	1.1x
Layout: floorplanning, placement, wire management	1.4x	1.0x
Exploiting process variation and accessibility	2x	1.2x

8.7.1 Microarchitecture

Many problems have inherent parallelism. Pipelining or performing multiple operations simultaneously can speed them up. The benefits depend on the degree of parallelism in the problem and in the sequencing overhead of the pipeline registers. Increasing pipelining or issuing more operations at once also increases the area, power consumption, and design effort. In the next section, we will see that ASICs have greater sequencing overhead and thus cannot benefit from pipelining as much as custom designs. However, many ASICs still have plenty of room to benefit from aggressive microarchitecture.

8.7.2 Sequencing Overhead

ASIC designers have little control of layout so they need sequencing elements that avoid races and are tolerant of noise. Most ASICs use very conservative flip-flops with a sequencing overhead (setup time + clock-to-Q delay) of 4–6 FO4 inverter delays.

In comparison, custom designs tend to use more aggressive sequential circuits. Some use pulsed latches or transparent latches. Some integrate logic into the latches to reduce sequencing overhead or use faster, but less robust elements when noise on the input or output can be better controlled. These improvements drive the sequencing overhead down to 2–3 FO4 delays.

Moreover, most systems are designed with a clock skew budget of about 10% of the cycle time. This corresponds to about 5 FO4 delays for a typical ASIC as compared to 1-2 for a custom design. The ASICs use clock distribution networks that are automatically generated by CAD tools. These networks are not as well balanced as in a good custom design and are much more susceptible to process variation. Moreover, they are not analyzed as carefully as in a custom design, so the clock skew budget must be conservative to account for uncertainties.

Finally, flip-flops impose hard edges, so clock skew directly impacts performance. They also are incapable of borrowing time to balance logic between pipeline stages or to opportunistically adjust for unpredicted delay variations. Custom latches have some skew tolerance. Custom designs also sometimes intentionally delay clocks to balance logic.

In summary, ASICs tend to have a sequencing overhead of about 10 FO4 delays, while custom designs have overhead as small as 3 FO4 delays. ASICs can reduce sequencing overhead by substituting pulsed latches or transparent latches for flip-flops on critical paths whenever tools determine that no races will occur. Better clock distribution networks and filtered power supplies on the PLL reduce skew.

8.7.3 Circuit Families

Most custom designs use skew-tolerant domino circuits on critical paths. Domino gates are 1.5–2x faster than complementary CMOS. Skew-tolerant domino pipelines use overlapping clocks to eliminate latches and their sequencing overhead. Custom designs occasionally use other circuit families such as CPL or pseudo-nMOS for specialized applications.

Domino is sensitive to noise and must be shielded from crosstalk. The monotonicity requirement tends to force the use of dual-rail gates. Domino has a high activity factor and thus high power consumption. It is not well understood by ASIC timing analyzers and test tools. While some groups are developing domino synthesis methodologies, many observers doubt that domino will gain widespread use in ASIC flows.

8.7.4 Logic Design

Custom designs tend to carefully optimize the logic design. Many functions have clever implementations that can be significantly faster. For example, lookahead adders are much faster than ripple carry adders, and Booth-encoded multipliers can be faster than those using simple arrays.

ASICs are synthesized from HDL code. Older synthesis tools took the code literally, so a Booth-encoded multiplier had to be specified manually. Modern tools are better at

choosing an appropriate implementation of common blocks like adders or multipliers (as long as you've purchased the vendor's intellectual property libraries!), but still need careful coding to produce good results for more exotic functions.

8.7.5 Cell and Wire Design

ASICs are synthesized onto a standard cell library. Some libraries have a small selection of gates. A library with only two drive strengths for NANDs, NORs, and inverters may be 25% slower than a library with a rich selection of drive strengths and buffer sizes and a full selection of compound and noninverting gates. The richer library can also reduce circuit area. Good libraries also offer asymmetric and skewed gates. While custom designs may use an unlimited set of cell sizes, this only offers a small percentage improvement over a good ASIC library with discrete sizes.

Wire capacitance and RC delay also are important. Custom designers choose the metal layer, wire width, and wire spacing and may even shield critical wires. ASIC designers have little control over the wires. Better ASIC tools are needed to automatically optimize wires on critical nets.

8.7.6 Layout

Global wires can account for a large fraction of total path delay and are getting worse with each process generation. Custom designers plan for these wires in advance and pipeline where necessary to tolerate the latency. They also add buffers and repeaters as needed. ASIC designers often do not think about the physical implementation and thus are surprised when long wires make a path much longer than simply counting gates would predict. Poor wire models in synthesis may mean these slow paths are not detected until after place and route has completed.

The key to managing wire delay is to carefully floorplan the chip. Units should be partitioned so that most communication happens inside the unit with short wires. Units that must communicate rapidly should be placed near each other. To reduce congestion, units communicating over wide busses should also be near each other. ASIC floorplanning tools are still maturing, but should eventually achieve parity with custom tools.

8.7.7 Process Variation

Custom designs are often fabricated on aggressive processes. For example, the Intel 130 nm process uses 60 nm effective channel lengths, while the TSMC process uses 80 nm effective lengths. Thus, an identical design will deliver a higher frequency on the Intel process. Intel also sometimes shrinks the effective channel length by another 5% as the process matures, improving performance even further. Foundries with higher performance processes tend to charge correspondingly higher prices.

Design corners also impact performance. Synchronous chips must be designed to operate at the worst temperature and voltage for which they are rated. Many ASICs are designed for the SS process corner so all of the chips fabricated without defects will meet timing. Thus, they can be tested only for functionality, not performance. Unfortunately,

the distribution of chips may have a long tail at the slow end, as shown in Figure 8.48.

Custom chips are often designed at the TT process corner to give a 17%–28% speedup at the expense of rejecting a few chips that fall in the tail. Moreover, the fastest 10% of the chips (the FF corner) may be up to 30% faster than TT. Microprocessors are commonly sorted by speed, and the fastest ones are sold at a premium. This is sometimes called *binning*. Over time, process improvements may yield even faster chips.

ASIC performance is being crippled by the lack of testing at speed. High-performance ASICs can be designed for the TT corner of a fast process rather than the SS corner of a slower process. In some markets, they may be binned to deliver some very fast chips.

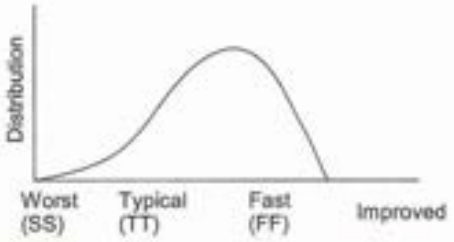


FIG 8.48 Distribution of chip behavior

8.7.8 Summary

Floorplanning and circuit families are commonly cited as the major advantages of custom over ASIC methodologies. Overall, [Chinnery02] concludes that these factors, while significant, are relatively overstated in their significance. Pipelining and process variation are of equal or greater significance. With attention to all of these details, ASICs may reduce their gap to within 2–3x the performance of custom designs.

8.8 CMOS Physical Design Styles

Basic gate layout was introduced in Section 1.5.4. In this section, we will examine the physical layout of CMOS gates in a general sense to understand the impact of the physical structure on the behavior and performance of circuits. For more extensive treatment by one of IBM's mask design instructors, see [Saint02].

8.8.1 Static CMOS Gate Layout

Complementary static CMOS gates can be designed using a single row of nMOS transistors below (or above) a single row of pMOS transistors, aligned at common gate connections. Most "simple" gates can be designed using an unbroken row of transistors in which abutting source/drain connections are made. This is sometimes called the "*line of diffusion*" rule, referring to the fact that the transistors form a line of diffusion intersected by polysilicon gate connections.

If we adopt this layout style, we can use automated techniques for designing such gates [Uehara81]. The CMOS circuit is converted to a graph where

1. the vertices in the graph are the source/drain connections
2. the edges in the graph are transistors gates that connect particular source/drain vertices

Two graphs, one for the pull-down network (n), and one for the pull-up network (p), result. Figure 8.49 shows an example of the graph transformation. The connection of edges in the graphs mirrors the series-parallel connection of the transistors in the circuits. Each edge is named with the gate signal name for that particular transistor. For example, the p-graph (light lines and circles) has four vertices: *Y*, *I1*, *I2*, and V_{DD}. It has four edges, representing the four transistors in the pull-up structure. Transistor *A* (*A* connected to gate) is an edge from the vertex *Y* to *I2*. The other transistors are similarly arranged in Figure 8.49(b). Note that the graphs are duals of each other because the pull-up and pull-down networks are the dual of each other. The n-graph (dark lines and crosses) overlays the p-graph in Figure 8.49(b) to illustrate this point. If two edges are adjacent in the p- or n-graph, then they can share a common source/drain connection and can be connected by abutment. Furthermore, if there exists a sequence of edges (containing all edges) in both graphs that have identical labeling, then the gate can be designed with no breaks in the line of diffusion. This path is known as a *Euler path*. The main points of the algorithm are:

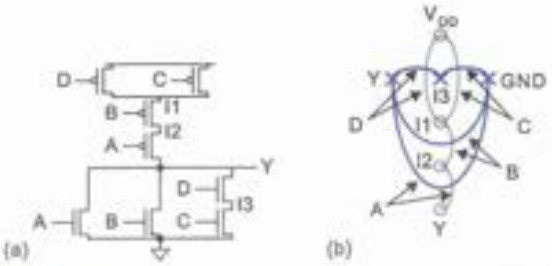


FIG 8.49 Circuit graphs

1. Find all Euler paths that cover the graph.
2. Find a p- and n-Euler path that have identical labeling (a labeling is an ordering of the gate labels on each vertex).
3. If the paths in step 2 are not found, then break the gate in the minimum number of places to achieve step 2 by separate Euler paths.

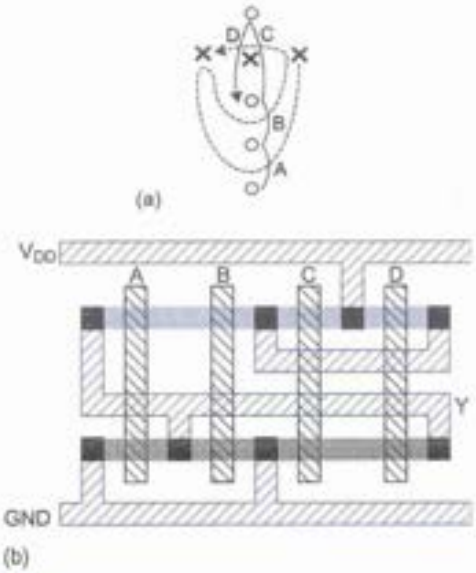


FIG 8.50 Stick diagram derived from Euler path

The original graph with a possible Euler path is shown in Figure 8.50(a). The sequence of gate signal labels in the Euler path is (*A*, *B*, *C*, *D*). To complete a layout, the transistors are arranged in the order of the labeling in parallel rows, as shown in stick diagram form in Figure 8.50(b). Vertical polysilicon lines form the gate connections. Metal routing wires complete the layout. This procedure can be followed when manually designing a gate, although good layouts usually become possible by inspection with a bit of practice.

A variation of the "line of diffusion" style occurs in circuits where a signal is applied to the gates of multiple transistors. In this case, transistors can be stacked on the appropriate gate signal using multiple rows of diffusion in a style called *gate matrix layout* [Wing82, Hu90]. This also

occurs in cascaded gates that cannot be constructed from a single row of transistors. A good example of this is the complementary XNOR gate. A schematic for this gate is shown in Figure 8.51(a). According to the style of layout that we have used to date, two possible layouts are shown in Figure 8.51(b) and Figure 8.51(c). The layout in Figure 8.51(b) uses the single row of n- and p-diffusion with a break, while that of Figure 8.51(c) uses a gate matrix layout. The selection of styles would depend on the overall layout—whether a short fat or long thin cell were needed. Note that the gate segments that are maximally connected to the power and ground rails should be placed adjacent to these signals.

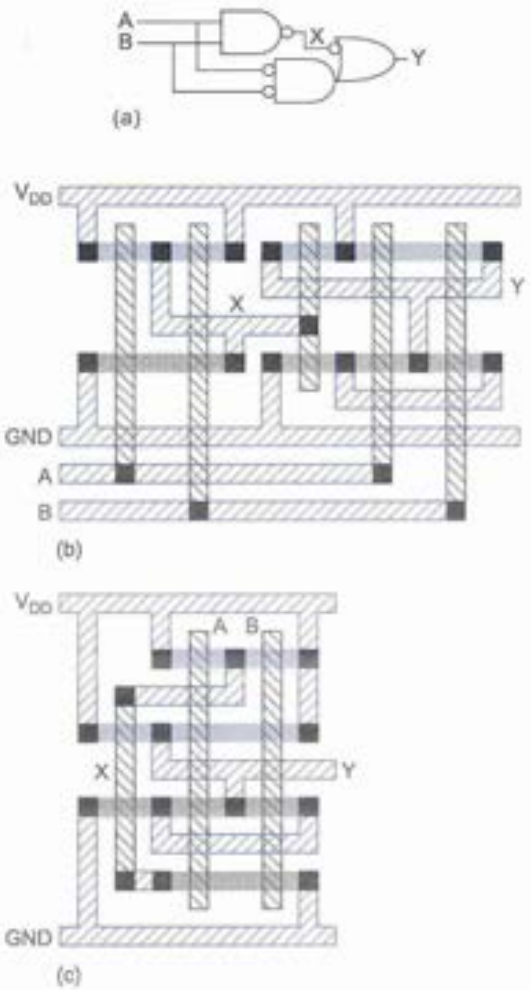


FIG 8.51 Broken line of diffusion and gate matrix cell layout styles

8.8.2 General CMOS Layout Guidelines

Layout can consume an unlimited amount of time because there are so many degrees of freedom and there is so much opportunity to squeeze a lambda here or there. In general, time to market is much more important than reducing chip area by a few percent, so it is important to settle on a simple and consistent layout design methodology. The following general layout guidelines can be stated:

1. Complete the electrical gate design and verification before layout. Circuit changes after layout is started become schedule busters.
2. Run V_{DD} and GND horizontally in metal at the top and bottom of the cell. Often these wires are wider than minimum to carry large DC currents without electromigration problems.
3. Run a vertical polysilicon line for each gate input.
4. Order the polysilicon gate signals to allow the maximal connection between transistors via abutting source/drain connections. These form gate segments.
5. Place n-diffusion segments close to GND and p-diffusion segments close to V_{DD}, as dictated by connectivity requirements.
6. Make connections to complete the logic gate in polysilicon (for short connections between gates) or metal. Squeeze transistors together to minimize diffusion between transistors.
7. Place well and substrate contacts under the supply lines in each cell.

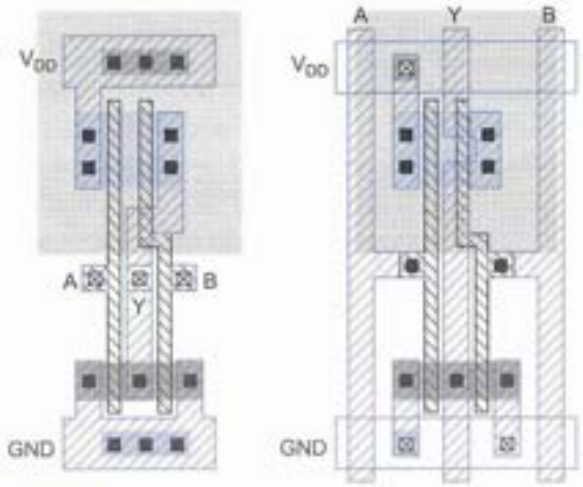


FIG 8.52 Standard cell metal usage

In general, metal layers should run perpendicular to each other to avoid "routing oneself into a corner." Exceptions are sometimes made to allow limited use of metal1 in the "wrong direction" to shorten connections or avoid the need for metal2 within a cell. Figure 8.52 shows two styles of standard cell layout for 2-input NOR gates. The first uses metal1 horizontally. The second uses metal1 vertically. Observe that the polysilicon gates are bent to minimize the diffusion between series transistors. The layouts assume that metal1–metal2 vias can be stacked on top of poly-metal1 contacts, as is common in modern planarized processes. If this is not allowed, the contacts must be placed adjacent to each other, sometimes increasing cell area.

For standard cells, inputs and outputs must usually be routed to contacts near the center in current processes or the top or bottom of the cell where they can connect to the routing channels for older processes with few metal layers. In the vertical metal1 style, this often increases the cell area because the metal1 cannot run over the top of other contacts within the cell. In datapath cells, however, inputs and outputs can contact bitlines running over the top of cells parallel with V_{DD} and GND. In this case, the vertical metal1 style may be preferred because metal2 bitlines are free to run horizontally over the cells.

Other layout guidelines include:

1. Diffusion has high resistance and capacitance. Never wire in diffusion. Minimize the area of diffusion regions. Fully contact large transistors to avoid series resistance through the diffusion between the contact and the edge of the transistor.
2. Polysilicon has high resistance, so use it only for short connections within cells. When long polysilicon lines are required (e.g., in the word line of a memory), strap the poly periodically with metal.
3. Lower levels of metal are thin and on a tight pitch. They are best for shorter connections (e.g., within a functional block) where density is important.
4. Upper levels of metal are thicker and on a wider pitch. They are faster and well-suited to global interconnections, the clock, and the global power/ground network. However, they are a scarce resource and must be carefully allocated.
5. Probe points should be placed on the top metal layer where they will be accessible during test (see Section 9.4).
6. Consider adding an assortment of unused gate array "happy gates" scattered through random logic. This facilitates making metal-only changes to fix logic bugs during silicon debug.

Note that the style of layout discussed involves optimizing the interconnection at the transistor level rather than the gate level. As a rule, smaller and perhaps faster layouts result by taking logic blocks with 10- to 100-transistor complexities rather than designing individual gates and trying to piece them together. For example, Figure 8.53(a) shows a transparent latch schematic. Figure 8.53(b) shows the latch layout built from simple standard cells, while Figure 8.53(c) shows an optimized layout with 2/3 the area. This improvement in density is due to a number of factors, including:

1. Better use of routing layers—routes can occur over cells
2. More "merged" source/drain connections
3. More use of "white space" (blank areas with no devices or connections) in sparse gates

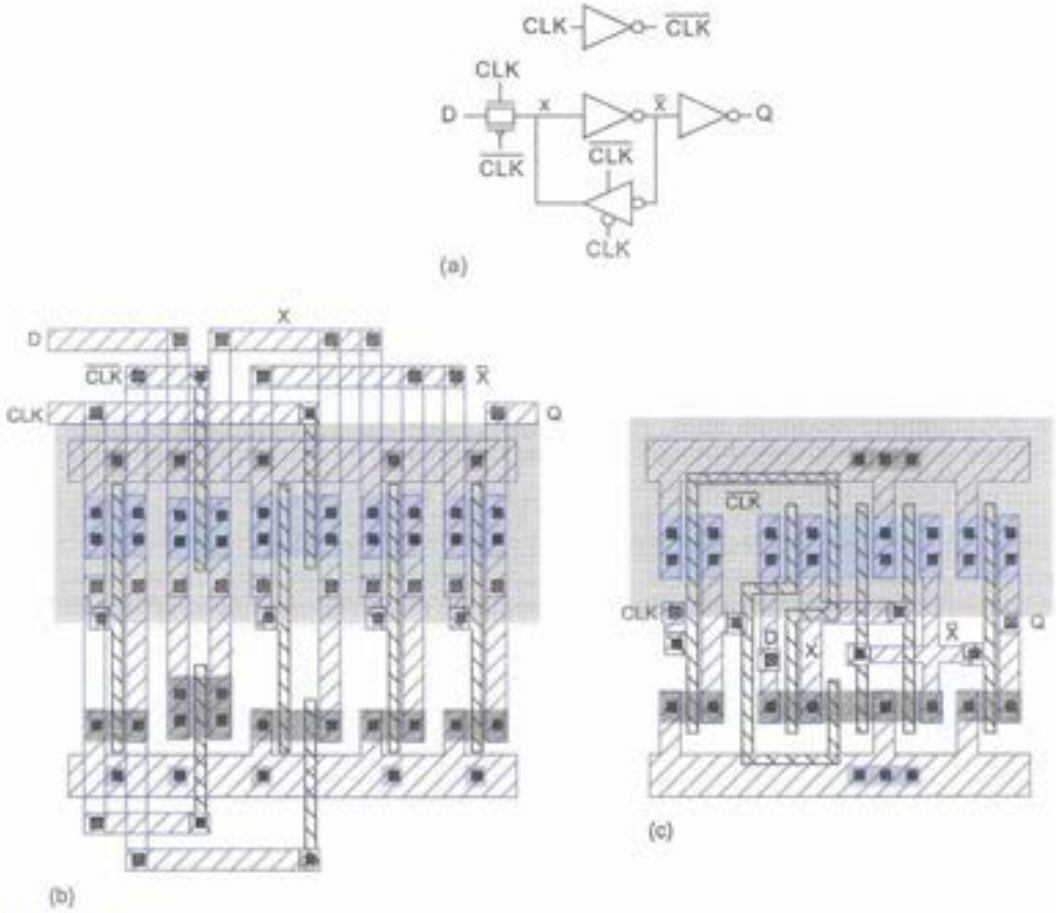


FIG 8.53 Transparent latch layouts

Improvements gained by optimizing at this level over a poorly implemented standard-cell approach can be up to 100% or more in area. However, such an approach is quite labor-intensive. These days, it is only worth investing manual effort in highly repetitive and reused structures like datapaths and widely used standard cells. Implementing random control logic manually in this manner is clearly a mistake because this type of logic often changes and the manual effort has to be continually spent to keep up with the changes. With modern multi-level metallization processes and optimized standard cell libraries, the density difference between custom-designed cells and hand or algorithmically placed standard cells is minimal if the same circuits are used because the transistor area fits under any routing. Density differences for custom circuits occur where the circuit is optimized to reduce the number of transistors (i.e., taking out buffer inverters in a latch). The point is that "in the old days" there was a much greater difference between custom and even a well-implemented standard cell design than there is today (given the same circuits).

8.8.3 Layout Optimization for Performance

Layout choices strongly impact the parasitic delay of logic gates. Good layout minimizes diffusion capacitance. Compact cells also lead to shorter wires between cells and lower wire parasitics. Figure 8.54 shows two implementations of a 24/12 λ inverter. Figure 8.54(a) is a conventional design. Figure 8.54(b) is a *folded* design in which each transistor is constructed from two parallel devices of half the width. Table 8.6 shows that the area and perimeter of the diffusion on the output *Y* has been reduced by nearly a factor of two. This does not change the logical effort, but reduces the parasitic delay by almost 50%. Wide transistors are folded multiple times to fit in layout more reasonably as well as to reduce diffusion capacitance.

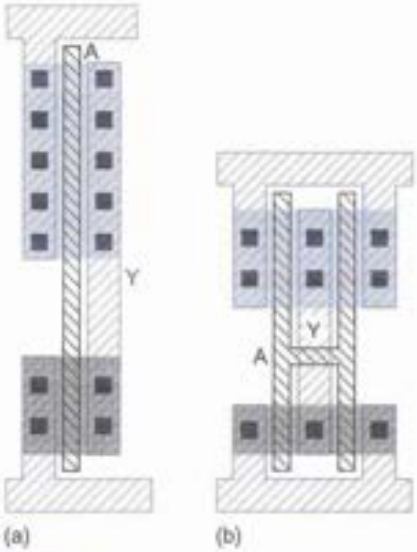


FIG 8.54 Folding transistors

A related case involves the design of parallel transistors. For example, Figure 8.55 shows three stick diagrams for a 2-input NOR gate. In Figure 8.55(a), the two nMOS drains are shared on the output as a single diffusion node, resulting in low parasitic capacitance on the output. The parallel connection of the two sources to the ground rail adds capacitance to the ground rail, but does not affect the output switching speed. In Figure 8.55(b), the two drains are separately contacted. This contributes more parasitic capacitance and results in a slower (and more power-hungry) gate, so such a layout should be avoided. When the transistors are wide, the entire series stack should be folded, as shown in Figure 8.55(c) for the pMOS stack.

The transistor sizes in datapaths can be optimized because the wire loads are known at the time of design. Standard cells can be over-designed to deal with larger routing loads that are unknown at the time of design. Modern practice in standard cells favors making the basic cell quite small and using scaled buffer inverters to provide appropriate drive for large routing loads.

Table 8.6 Diffusion dimensions for unfolded and folded transistors

Transistor	Unfolded			Folded		
	Width (λ)	Drain Area (λ^2)	Drain Perimeter (λ)	Width (λ)	Drain Area (λ^2)	Drain Perimeter (λ)
N1	12	60	22	6 + 6	36	12
P1	24	120	34	12 + 12	72	12

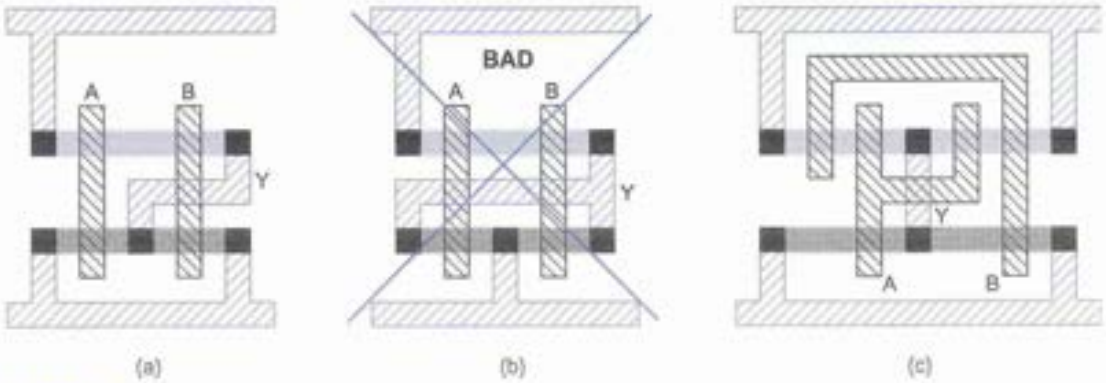


FIG 8.55 Parallel transistor layout

In long series stacks of transistors, the parasitic delay grows quadratically with the number of transistors, as shown in EQ (4.6). For gates driving small loads, the delay may be dominated by this parasitic component. For example, Figure 8.56(a) shows a 4-input footless dynamic NAND gate. It may be possible to reduce the parasitic delay by *tapering* transistors, as shown in Figure 8.56(b) [Shoji85, Wurtz93]. Tapering reduces the size of the inner transistors to reduce their diffusion capacitance at the expense of greater resistance, as shown in Table 8.7. Because the capacitance is discharged through all the series transistors while the resistance is only increased for the inner transistors, tapering reduces the parasitic delay. The greater resistance results in higher logical effort, so the technique is only appropriate for lightly loaded gates. Unfortunately, under most layout design rules, the spacing

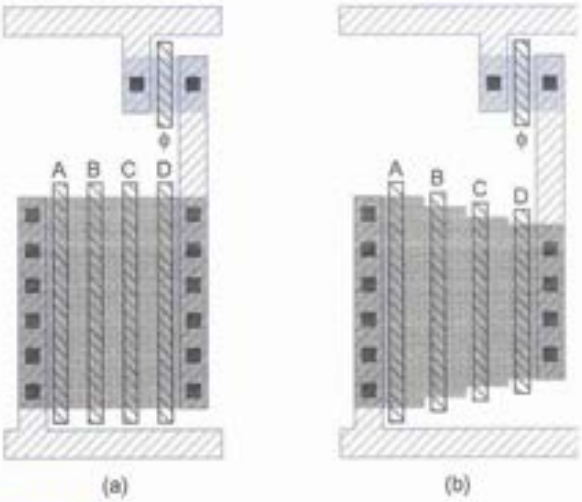


FIG 8.56 Tapering

Table 8.7 Diffusion dimensions in untapered and tapered stacks

Transistor	Untapered			Tapered		
	Width (λ)	Drain Area (λ^2)	Drain Perimeter (λ)	Width (λ)	Drain Area (λ^2)	Drain Perimeter (λ)
N1	30	90	6	30	117	11
N2	30	90	6	27	105	11
N3	30	90	6	24	94	10
N4	30	150	40	22	110	32
P1	8	40	18	8	40	18

between uncontacted series transistors must increase when the transistors are tapered. Notice that the internal diffusion areas were larger in the tapered stack, as shown in Figure 8.56. As a result, tapering series stacks in practice may result in comparable or even greater parasitic delay [Hoppe90]. Tapering compound gates at contacted diffusion nodes does not increase the diffusion area and may still be a reasonable practice.

8.9 Interchange Formats

Throughout this chapter, reference has been made to a variety of (mostly) text interchange formats. These are summarized in the following sections.

8.9.1 GDS2 Stream

GDS2 stream is an older binary format that is the *de facto* standard for describing mask geometry. It includes layering and geometric primitives such as rectangles and polygons.

8.9.2 Caltech Intermediate Format (CIF)

Caltech Intermediate Format (CIF) is an alternative text-based mask description language largely used by the academic community [Mead80].

8.9.3 Library Exchange Format (LEF)

Library Exchange Format (LEF) describes the physical attributes of library cells, including port locations, layers, and via definitions. LEF abstracts the lower-level geometric details of a cell. Enough information is specified to allow a router to connect to the cell without impinging on internal cell constraints. The following LEF code segment describes part of an analog to digital converter.

```
LAYER metal1
   TYPE ROUTING ;
   WIDTH 0.2 ;
   SPACING 0.4 ;
   PITCH 1.0 ;
   DIRECTION HORIZONTAL ;
   CAPACITANCE CPERSQDIST 0.00003 ;
END metal 1
...
MACRO adc
   ORIGIN 0 0 ;
   SIZE 100 BY 200 ;
   PIN in
   DIRECTION INPUT ;
   PORT
   LAYER METAL4 ;
   RECT 10 10 10.5 10.5
   END in
   ...
   OBS
   LAYER METAL1 ;
   RECT 0 100 20 150 ;
   END
END adc
```

The description is somewhat human-readable. First, some technology specifications are evident. These specify which layer does what on the process. For instance, metal1 is a routing layer with the width, spacing, and pitch shown. The direction for routing is specified as horizontal. Alternate layers usually route in orthogonal directions. Finally, the capacitance of the layer is specified, which allows tools to make rough estimates of delays if necessary.

The description then proceeds with a declaration of the cell name (adc). The origin and size are then specified. A typical pin (in) is specified with the direction, layer, size (and position) designated. The size is specified by the `RECT 10 10 10.5 10.5` command.

Finally, the OBS statement describes an obstruction—the areas in a given layer that a router must avoid within a cell.

8.9.4 Design Exchange Format (DEF)

Design Exchange Format (DEF) describes an actual design by listing the library elements and their placement and connectivity. DEF is used to pass designs between different design systems while maintaining the design intent. This is opposed to just passing geometric information such as GDS2 stream format. A segment of DEF is shown below.

```
VERSION 5.2 ;
NAMESCASESENSITIVE ON ;
DIVIDERCHAR "/" ;
BUSBITCHARS "[]" ;
DESIGN chip ;
UNITS DISTANCE MICRONS 1000 ;

COMPONENTS 1 ;
-adc_inst adc + FIXED ( -1000 -1000) N ;
END COMPONENTS

END DESIGN
```

Again, after some housekeeping, this is declared to be part of the design "chip." There is one placement of the adc block (`adc_inst` would be the Verilog role name). This block has been "fixed," i.e., the placement cannot be changed. Blocks can be unplaced, in which case, a placement program may move the blocks to improve timing or chip size.

8.9.5 Standard Delay Format (SDF)

SDF is an IEEE standard (P1497) to describe timing information in designs. It specifies pin-to-pin delays of modules, clock-to-data delays, and interconnect delays. A segment of SDF resulting from a timing analysis run is shown below.

```
(DELAYFILE
(SDFVERSION     "2.1")
(DESIGN         "chip")
(DATE           "March 24, 2003 11:57:3")
(VENDOR         "")
(PROGRAM        "PEARL")
(VERSION        "PEARL 5.1-s072 (64 bit)")
(DIVIDER        /)
(VOLTAGE        1.080:1.20:1.320)
(PROCESS        "slow=1.5:nom=1.0:fast=0.75")
(TEMPERATURE    80.000:25.000:-40.000)
(TIMESCALE      1ns)
(CELL
(CELLTYPE       "chip")
(INSTANCE)
(DELAY
   (ABSOLUTE
      (INTERCONNECT clk_mdi u_pad_clk_mdi/PAD (0.1:0.2:0.3) (0.4:0.5:0.6))
      (INTERCONNECT u_pad_clk_mdi/PAD clk_mdi (0.4:0.5:0.6) (0.7:0.8:0.9))
      )
      )
   (CELL
      (CELLTYPE "DFlipFlop")
      (INSTANCE foo/bar/reg)
      (DELAY
```

```
        (ABSOLUTE
           (IOPATH CK Q (.2:.25:.3) (.3:.35:.4))
        )
     )
  (TIMINGCHECK
     (WIDTH (posedge CK) (.06:.06:.06))
     (WIDTH (negedge CK) (.12:.12:.12))
     (SETUP (posedge D) (posedge CK) (0.1:0.1:0.1))
     )
  )
)
```

The segment begins with the design name and a log of what program ran to produce the SDF. Some interconnect delays are then given. The statement

```
(INTERCONNECT clk_mdi u_pad_clk_mdi/PAD (0.1:0.2:0.3) (0.4:0.5:0.6))
```

specifies an interconnect delay path between `clk_mdi` and `u_pad_clk_mdi/PAD` with rising and falling delays respectively of (`0.1:0.2:0.3`) and (`0.4:0.5:0.6`) ns, where the (`a:b:c`) format relates to (`slow:nominal:fast`) process corners. The specification of delay values can be extended from that shown here to include transitions between all states (0, 1, *Z*, *X*).

Next, the delay within a DFlipFlop cell is specified. This starts with an absolute delay from the clk-to-*Q* specified by

```
(IOPATH CK Q (.2:.25:.3) (.3:.35:.4))
```

Again the delay numbers noted here are for rise and fall times and slow, nominal, and fast process corners. The clock constraints on the DFlipFlop are then listed. The minimum clock widths are specified for each process corner. Finally, D-to-*clk* setup time is specified.

8.9.6 DSPF and SPEF

Detailed Standard Parasitic Format (DSPF) and Standard Parasitic Exchange Format (SPEF) are formats are used to pass parasitic RC values between extraction tools and timing verification tools. They are modeled after SPICE decks, as shown below.

```
*|DSPF 1.5
*|DESIGN "chip"
*|DATE "Mon Mar 24 07:26:34 2003"
*|VENDOR "Cadence Design Systems - HLD"
*|PROGRAM "hyperExtract 4.5.0"
*|VERSION "3.4E"
*|DIVIDER |
*|DELIMITER .
*|BUSBIT []

.SUBCKT chip in out
+ Vdd gnd
```

```
*Net Section
*
*|GROUND_NET gnd
*
*|NET name1 0.003PF
*I ???
*S ????
C1 node1 node2 1.2
C2 node3 node4 0.85
R1 node4 node5 0.5
*
.ENDS
```

8.9.7 Advanced Library Format (ALF)

Advanced Library Format (ALF) is an alternate format used to describe primitive library elements. It combines electrical performance characteristics such as capacitive pin loading and slew rates (e.g., SDF), physical modeling characteristics (such as pin layers and other geometric data, e.g., LEF) and behavioral specifications (such as the logic implemented by a cell, e.g. Verilog and VHDL).

8.9.8 WAVES Waveform and Vector Exchange Specification

WAVES is IEEE Std. 1029.1 and a subset of the VHDL standard. It provides for the graphical definition of stimulus and response patterns for digital systems. The following code is an example generated by the Aldec Active HDL suite.

```
% ************************************************************
% * This file is automatically generated WAVES vector file,   *
% * and can be used in Test Bench generator.                  *
% * ACTIVE-HDL Testbench Generator ver. 3.5.                  *
% * Copyright (C) ALDEC Inc.                                  *
% *                                                           *
% * This file was generated on: 10:29:49 AM 12/9/2003         *
% ************************************************************
%
% Begin Comment
%
% reset : sig reg
% clk : sig reg
%
% 4.00 us : END_SIMULATION_TIME
% End Comment
%
% Begin of Vectors
%
%# 0 0
```

```
x 0 : 15 ns ; % 0 fs
x 1 : 15 ns ; % 15 ns
1 0 : 15 ns ; % 30 ns
1 1 : 15 ns ; % 45 ns
.
1 0 : 15 ns ; % 330 ns
1 1 : 15 ns ; % 345 ns
1 0 : 15 ns ; % 360 ns
1 1 : 15 ns ; % 375 ns
0 0 : 15 ns ; % 390 ns
0 1 : 15 ns ; % 405 ns
0 0 : 15 ns ; % 420 ns
0 1 : 15 ns ; % 435 ns
0 0 : 15 ns ; % 450 ns
0 1 : 15 ns ; % 465 ns
```

The preceding code describes two signals: *reset* and *clk*, which are shown in Figure 8.57.

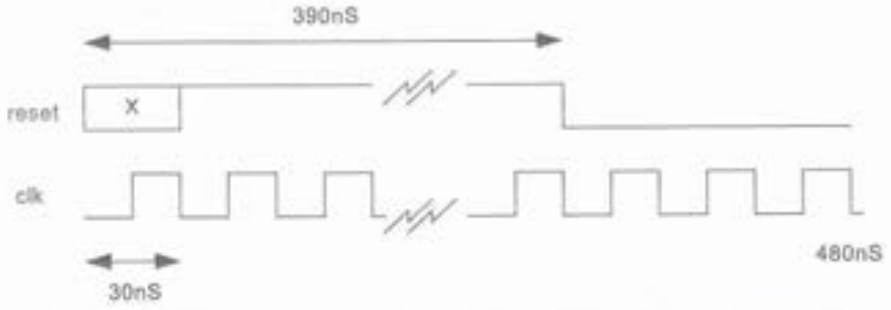


FIG 8.57 WAVES example waveforms

8.9.9 Physical Design Exchange Format (PDEF)

Physical Design Exchange Format (PDEF) is a Synopsys format to pass information between front-end and back-end tools. It describes hierarchical groupings of cells and the relevant interconnect information such as RC delays and cell loadings.

8.9.10 OpenAccess

Standard text file formats are used to exchange information between tools, designers and companies wishing to cooperate. A more effective manner of sharing data between tools running on the same machine is to provide an Application Programming Interface (API) or a set of software routines that communicate with a central database and extract data and insert data into that database. The OpenAccess API is one example of such an interface that has been applied to LEF and DEF descriptions.

8.10 Historical Perspective

When the second edition of this text was published, a few designers (including the author) were still using tools that were "home brewed," that is, cobbled together from publicly available or custom software. The designs were at a level of complexity where the ability to customize a tool to perform a function was quicker than waiting for a vendor to have a commercially available tool. The cost of entry into even custom IC design could be kept quite low. Now it is virtually impossible not to use a commercially available design flow without seriously impinging on schedules and design quality and integrity. Most large US, APAC, and EMEA universities now have commercially produced tools that are used for teaching. So, while the availability of advanced tools is an expensive startup cost for a small company, the basic tools and flows are available at a selection of tertiary institutes.

Small, well-defined designs can still be completed with a skeleton of tools; it just takes longer and is harder to prove correct. A layout editor, 2D extractor, LVS, and SPICE simulator can design a lot of circuits, especially in the analog mixed-signal area. But remember to engineer things properly and abide by the advice of this quote:

"A moment of convenience, a lifetime of regret." – David Moon on TECO madness.

8.11 Pitfalls and Fallacies

Inadequate design flow

In the previous section, we stated that it is possible to design small circuits without the benefit of a commercial design flow. On the other hand, using an inadequate design flow for a large chip is a recipe for failure. While commercial ASIC design flows are fairly well exercised, mixed-signal and advanced processes can tend to break tools (even commercial tools). Consider the case of moving from a 0.25 μm process to a 130 nm process without a signal integrity checker. In the latter technology case, this tool is almost mandatory to prevent timing anomalies due to unwanted wire-to-wire coupling.

Insufficient verification

Synopsys found that 82% of design spins for chips with functional flaws were due to lack of verification [Schutten03]. Another 47% of re-spins had incorrect specifications. And 14% had errors in imported IP. This outlines the need for good specifications and a well-thought-out verification plan. Verification is further covered in Chapter 9.

Inaccurate parasitic extraction

Parasitic extractions programs output reams of data relating to C and R values in a design. Unless these are guaranteed by your vendor, it is prudent to do a small design and compare the values with hand-calculated values. You can never be too careful when it comes to designing a chip. When the chip comes back, compare a known path with what was predicted by the tool set.

Exercises

8.1 The gate mix for a family of chips has found that 30% of the gates are scannable Dflip-flops, 20% are 3- or 4-input gates, and 40% are 2-input gates (NAND or NOR) and 10% are buffers of various sizes. Propose a gate array cell that best fits this gate mix for density. Is an SOG array a better choice from an area point of view?

8.2 An FIR filter for a GSM receiver with sigma-delta converter as shown in Figure 8.5(b) has a single-bit input. To what structure do the multipliers degenerate? If the coefficients are a single bit and a 288-tap filter has to operate at 13 MHz, what architecture would you use for the overall design?

8.3 What kind of RAM cell would you use to control a configurable logic block in an FPGA? Design the cell and outline the reasons for your choice.

8.4 Explain the tradeoffs between using a transmission gate or a tristate buffer to implement an FPGA routing block.

8.5 Using the MOSIS scalable rules, design a standard cell library where the nMOS transistor is 4 λ wide and the pMOS is 6 λ wide. What is the minimum vertical and horizontal pitch that you can achieve?

8.6 Write a software generator in a familiar high-level language to implement a programmable inverter (1x to 64x basic inverter size) for the standard cell designed in Exercise 8.5. The output should be terms of rectangles with the following syntax:

```
begin inv_Mx cell_name
.
.
   rect layer lower-x lower-y upper-x upper-y
.
.
end cell_name
```

layers are n-diff, p-diff, contact, poly, metal1, via12, metal2.

M is the number of paralleled unit inverters.

Research the syntax of LEF and write a LEF generator for the inverters.

8.7 Complete the 64-entry 7-bit sine ROM layout generator introduced in Section 8.3.5.

8.8 Write a software generator that generates a Verilog description for the sine ROM. Parameterize the ROM in amplitude and phase resolution.

8.9 If the ROM generator had to be implemented with static CMOS standard cells (as opposed to transistors), what cells would you use for each block in the example given in Section 8.3.5?

8.10 Show how power can be measured with SPICE for a digital circuit.

8.11 Research the detail of a CORDIC processor and design a CORDIC capable of generating an 8-bit sine wave with 8-bit phase accuracy. Code in an HDL. Compare this in terms of size, speed, and power dissipation with a ROM-based NCO.

8.12 Construct a data sheet for the CORDIC designed in the previous question.

8.13 Estimate the die cost of a 4 × 4 mm die, with Y_w = 80% and Y_{pw} = 98% for an 8" wafer costing \$2,200 each. The die may be shrunk to 3.3 × 3.3 mm in a more advanced process that costs \$3000 per wafer. Is it worth moving to the new process if the volume is large enough?

8.14 Sketch a stick diagram for a large inverter with an 80 λ pMOS transistor and 40 λ nMOS transistor. Fold the transistors so that no single transistor is wider than 20 λ.

8.15 Using the RC delay model, estimate the worst-case falling delay of the untapered and tapered 4-input dynamic NAND gates in Figure 8.57. Let the load be a 24/6 λ inverter. Assume the diffusion capacitance between two series unit transistors is $C/2$ if the transistors are the same width, $3C/4$ if the transistors are of different widths, and C if the node requires a contact. Does tapering produce a faster gate in this example?

8.16 Using the same assumptions as in Exercise 8.15, calculate the falling parasitic delay and logical effort from each input of the tapered and untapered 4-input dynamic NAND gates. For what electrical effort is the tapered gate faster?

Testing and Verification

9

9.1 Introduction

While in real estate the refrain is "Location! Location! Location!" the comparable advice in IC design should be "Testing! Testing! Testing!" For many chips, testing accounts for more effort than design.

Tests fall into three main categories. The first set of tests verifies that the chip performs its intended function. These tests are run before tapeout to verify the functionality of the circuit and are called *functionality tests* or *logic verification*. The second set of tests are run on the first batch of chips that return from fabrication. These tests confirm that the chip operates as it was intended and help debug any discrepancies. They can be much more extensive than the logic verification tests because the chip can be tested at full speed in a system. For example, a new microprocessor can be placed in a prototype motherboard to try to boot the operating system. This *silicon debug* requires creative detective work to locate the cause of failures because the designer has much less visibility into the fabricated chip compared to during design verification. The third set of tests verify that every transistor, gate, and storage element in the chip functions correctly. These tests are conducted on each manufactured chip before shipping to the customer to verify that the silicon is completely intact. These will be called *manufacturing tests*. In some cases, the same tests can be used for all three steps, but often it is easier to use one set of tests to chase down logic bugs and another, separate set optimized to catch manufacturing defects.

In Section 8.5.2, we noted that the yield of a particular IC was the number of good die divided by the total number of die per wafer. Because of the complexity of the manufacturing process, not all die on a wafer function correctly. Dust particles and small imperfections in starting material or photomasking can result in bridged connections or missing features. These imperfections result in what is termed a *fault*. Later in the chapter we will examine a number of fault mechanisms. The goal of a manufacturing test procedure is to determine which die are good and should be shipped to customers.

Testing a die (chip) can occur at the

- wafer level
- packaged chip level
- board level
- system level
- field level

By detecting a malfunctioning chip early, the manufacturing cost can be kept low. For instance, the approximate cost to a company of detecting a fault at the various levels [Williams86] is:

- wafer $0.01–$0.10
- packaged chip $0.10–$1
- board $1–$10
- system $10–$100
- field $100–$1000

Obviously, if faults can be detected at the wafer level, the cost of manufacturing is lower. In an extreme example, Intel failed to correct a logic bug in the Pentium floating-point divider until more than 4 million units had shipped in 1994. IBM halted sales of Pentium-based computers and Intel was forced to recall the flawed chips. The mistake and lack of prompt response cost the company an estimated $450 million.

It is interesting to note that most failures of first-time silicon result from problems with the functionality of the design; that is, the chip does exactly what the simulator said it would do, but for some reason (almost always human error) this functionality is not what the rest of the system expects.

The remainder of this section will provide an overview of the processes involved in logic verification, chip debug, and manufacturing test. Section 9.2 discusses the mechanics of testing and test programs. Sections 9.3–9.5 address the principles behind each phase of testing. If testing is not considered in advance, the manufacturing test can be extremely time consuming and hence expensive. Some chips have even proved impossible to debug because designers have so little visibility into the internal operation. Sections 9.6–9.9 focus on how to design chips to facilitate debug and manufacturing test at the chip and board level. Even if a chip works when it is shipped to the customer, it may wear out over time. Section 9.10 examines reliability and reliability testing.

9.1.1 Logic Verification

Verification tests are usually the first ones a designer might construct as part of the design process. Does this adder add? Does this counter count? Does this state-machine yield the right outputs each cycle? Does this modem decode data correctly?

In Chapter 8, we noted that verification tests were required to prove that a synthesized gate description was functionally equivalent to the source RTL. Figure 9.1 shows that we may want to prove that the RTL is equivalent to the design specification at a higher behavioral or specification level of abstraction. The behavioral specification might be a verbal description; a plain language textual specification; a description in some high-level computer language such as C, FORTRAN, Pascal, or LISP; a program in a system-modeling language such as SystemC; or a hardware description language such as VHDL or Verilog; or simply a table of inputs and required outputs. Often designers produce a *golden model* in one of the previously mentioned formats and it becomes the reference

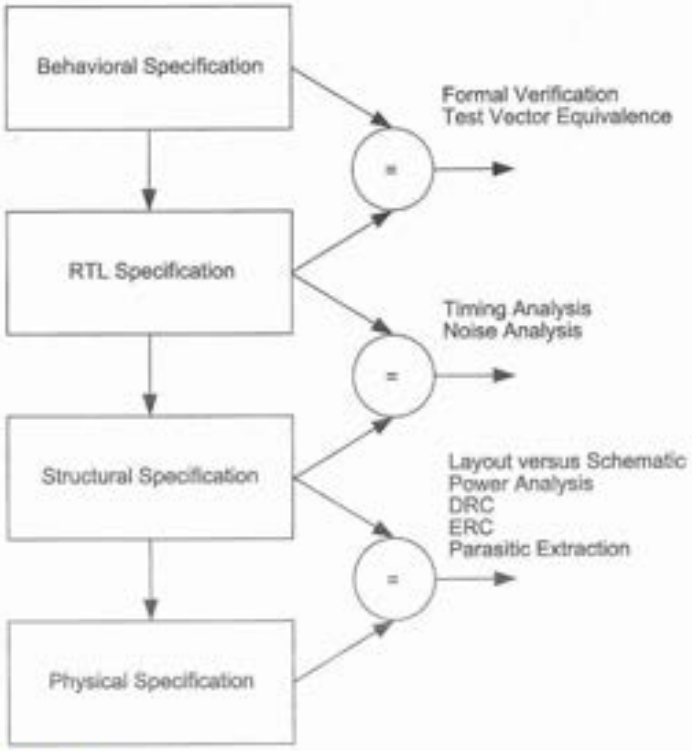


FIG 9.1 Functional equivalence at various levels of abstraction

against which all other representations are checked. Functional equivalence involves running a simulator at some level on the two descriptions of the chip (e.g., one at the gate level and one at a functional level) and ensuring that the outputs are equivalent at some convenient check points in time for all inputs applied. This is most conveniently done in an HDL by employing a *test bench*, i.e., a wrapper that surrounds a module and provides for stimulus and automated checking. The most detailed check might be on a cycle-by-cycle basis. Increasingly, verification involves real-time or near real-time emulation in an FPGA-based system to confirm system-level performance *in situ*, i.e., in the actual system that will use the end chip. This is recommended because of the increasing level of complexity of chips and the systems they implement. As an example, in the area of wireless local area network chips, without a real-time emulation system, it is virtually impossible to simulate the unseen effects of an unreliable channel with out-of-band interferers.

You can check functional equivalence through simulation at various levels of the design hierarchy. If the description is at the RTL level, the behavior at a system level may be able to be fully verified. For instance, in the case of a microprocessor, you can boot the operating system and run key programs for the behavioral description. However, this might be impractical (due to long simulation times) for a gate-level model and even harder

for a transistor-level model. The way out of this impasse is to use the hierarchy inherent within a system to verify chips and modules within chips. That, combined with well-defined modular interfaces, goes a long way in increasing the likelihood that a system composed of many VLSI chips will be first-time functional.

The best advice with respect to writing functional tests is to simulate as closely as possible the way in which the chip or system will be used in the real world. Often this is impractical due to slow simulation times and extremely long verification sequences. One approach is to move up the simulation hierarchy as modules become verified at lower levels. For instance, you could replace the gate-level adder and register modules in a video filter with functional models and then in turn replace the filter itself with a functional model. At each level, you can write small tests to verify the equivalence between the new higher-level functional model and the lower-level gate or functional level. At the top level, you can surround the filter functional model with a software environment that models the real-world use of the filter. For instance, you can feed a carefully selected subsample of a video frame to the filter and compare the output of the functional model with what the designer expected theoretically. You can also observe the video output on a video frame buffer to check that it looks correct (by no means an exhaustive test, but a confidence builder). Finally, if enough time is available, you can apply all or part of the functional test to the gate level and even the transistor level if transistor primitives have been used.

Verification at the top chip level using an FPGA emulator offers several advantages over simulation and, for that matter, the final chip implementation. Most noticeably, the emulation times can be near real time. This means that the actual analog signals (if used) can be interfaced with the chip. Additionally, to assess system performance, you can introduce fine levels of observation and monitoring that might not be included in the final chip. For instance, you could include a bit-error rate circuit in a communication modem to aid performance optimization.

In most projects the amount of verification effort greatly exceeds the design effort. Remember the following statement, culled from many years of IC design experience, whenever you are tempted to minimize verification effort to meet tight schedules:

"If you don't test it, it won't work! (guaranteed)"

9.1.2 Basic Digital Debugging Hints

Many times, when a chip returns from fabrication, the first set of tests are run in a lab environment, so you need to prepare for this event. You can begin by constructing a circuit board that provides the following attributes:

- Power for the IC with ability to vary V_{DD} and measure power dissipation
- Real-world signal connections (i.e., analog and digital inputs and outputs as required)
- Clock inputs as required (it is helpful to have a stable variable-frequency clock generator)
- A digital interface to a PC (either serial or parallel ports for slow data or PCI bus for fast data interchanges)

You can write software routines to interface with the chip through the serial or parallel port or the bus interface. The chip should have a serial UART port or some other interface that can be used independently of the normal operation of the chip. The lowest level of the software should provide for peeking (reading) and poking (writing) registers in the chip. An alternate or complementary approach is to provide interfaces for a logic analyzer. These are easily added to a PCB design in the form of multi-pinned headers. Figure 9.2 shows a typical test board, illustrating the *zero insertion force* (ZIF) socket for the chip (in the center of the board), an area for analog circuitry interface (on the left), a set of headers for logic analyzer connection (at the top and bottom) and a set of programmable power supplies (on the right). In addition, an interface is provided for control by a serial port of a PC (at the bottom left).

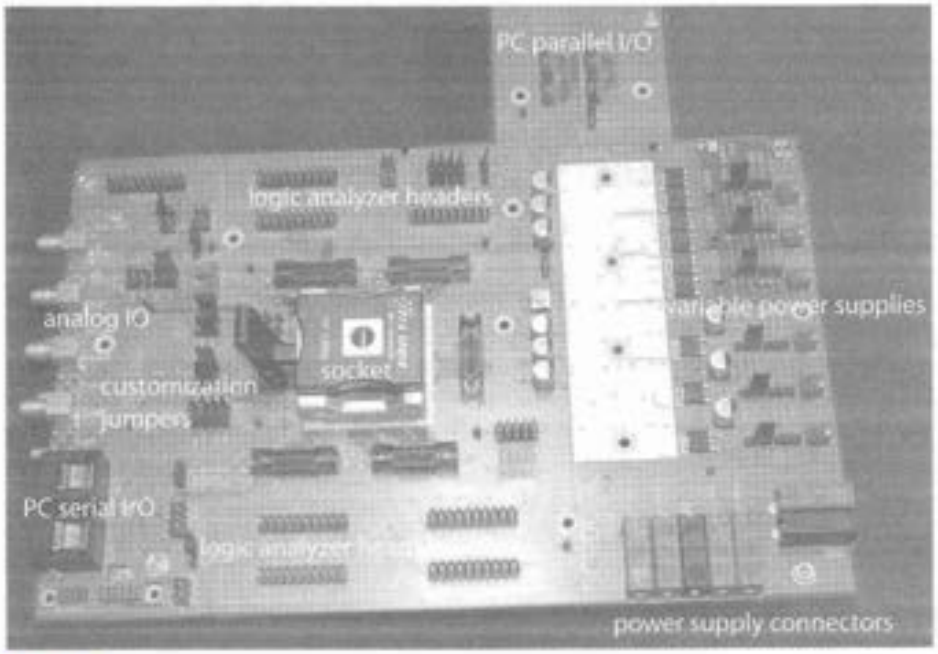


FIG 9.2 Typical test board

You should start with a "smoke test." This involves ramping the supply voltages from zero to V_{DD} while monitoring the current without any clocks running. For a fully static circuit, the current should remain at zero. Analog circuits will draw their quiescent current.

Following this, you can enable the clock(s); some dynamic current should be evident. Beware that many CMOS chips appear to operate when the clock is connected but the power supply is turned off because the clock may partially power the chip through the input protection diodes on the input pads. If possible, you should initially run the clock at reduced speed so that setup time failures are not the initial culprit in any debug operation.

In the case of a digital circuit, you should examine various registers for health using PC-based peek and poke software. This checks the integrity of the signal path from the PC to the chip. Often, designers place an ID in the register at address zero. Peeking at this register proves the read path from the chip. If the chip registers are reset to a known state, the registers can be read sequentially and compared with the design values. In the case of the logic analyzer, you can download the equivalent test pattern to exercise the chip. Frequently, these patterns can be automatically generated from the verification test bench. Up to this point, no functionality of the chip has been exercised apart from register reads and writes.

Where the chip has built-in self-test (see Sections 9.6–9.7), you can run the commercial software that provides for this functionality over a boundary scan interface. This type of system automatically runs a set of tests on the chip that completely verify the correct operation of all gates and registers as defined by the original RTL description. If this kind of a test interface was not used, you should pursue a manual effort in which the functionality of the chip is checked from the bottom-up. Of course, if you are a gambler, you can do a top-level test like running a piece of code or trying to boot the operating system right away. Experience shows that this often does not work, usually because of problems with the test fixture, and so you must revert to the bottom-up method to prove that one piece of the design works at a time.

If you detect anomalous behavior, you must go about debugging. The basic method is to postulate a method of failure, then test the hypothesis. Debug is an art in itself, but some pointers for sane debugging are as follows:

- Keep an annotated and dated logbook for all tests done.
- When postulating a cause for the bug and a test, do one change at a time and observe the result: Changing many things and then seeing if they work will not logically lead you to the bug and is commonly called the "shotgun approach."
- Check everything two or three times; never assume anything unless it is measured and logged in a notebook.
- Check signals and supply voltages at the pins of the IC; frequently, new test boards have errors.
- Double-check the specified chip I/O and perform a continuity check from the IC pins to expected places (i.e., test pins, supplies) on the board.
- Never count out a possible reason for a bug, however crazy, unless you can prove it isn't the cause.
- Use freeze spray or a heat gun to cool down or heat up a circuit to check for temperature problems.
- Check the state of any internal registers against that noted in the documentation.
- Evaluate the timing of any inputs and outputs with respect to the clock; often setup or hold times can be violated in a new test setup.

- When a bug is discovered and corrected, hunt for other portions of the design that might have a similar bug that hasn't been detected yet. Where there is one rat, there are many rats!
- Never assume anything—question everything—a slight touch of paranoia helps!!

When the chip is demonstrated to be operational, you can measure more subtle aspects of the design such as performance (power, speed, analog characteristics). This involves normal lab techniques of configure, measure, and record. Where possible, store all results as computer readable results (i.e., stored images from digital oscilloscope, screen dumps from logic analyzer) for communication with colleagues.

For the most part, if a digital chip simulates at the gate level and passes timing analysis checks during design, it will do exactly the same in silicon. Possible deviations from the simulated circuit occur in the following cases:

- Circuit is slower than predicted—fix—slow clock or raise V_{DD}
- Circuit has a race condition—fix—heat with heat gun if a logic gate caused race
- Circuit has dynamic logic problems—fix—don't do it again…
- Gnarly crosstalk problems—fix—get better tools
- Wrong functionality—fix—do a better job of verification

With analog circuitry, a wide range of issues can affect performance over and above what was simulated. These include power and ground noise, substrate noise, and temperature and process effects. However, you can employ the same basic debug approaches.

9.1.3 Manufacturing Tests

Whereas verification or functionality tests seek to confirm the function of a chip as a whole, manufacturing tests are used to verify that every gate operates as expected. The need to do this arises from a number of manufacturing defects that might occur during either chip fabrication or accelerated life testing (where the chip is stressed by over-voltage and over-temperature operation). Typical defects include:

- layer-to-layer shorts (e.g., metal-to-metal)
- discontinuous wires (e.g., metal thins when crossing vertical topology jumps)
- missing or damaged vias
- shorts through the thin gate oxide to the substrate or well

These in turn lead to particular circuit maladies, including:

- nodes shorted to power or ground
- nodes shorted to each other
- inputs floating/outputs disconnected

Tests are required to verify that each gate and register is operational and has not been compromised by a manufacturing defect. Tests can be carried out at the wafer level to cull out bad die, or can be left until the parts are packaged. This decision would normally be determined by the yield and package cost. If the yield is high and the package cost low (i.e., a plastic package), then the part can be tested only once after packaging. However, if the wafer yield was lower and the package cost high (i.e., an expensive ceramic package), it is more economical to first screen bad dice at the wafer level. The length of the tests at the wafer level can be shortened to reduce test time based on experience with the test sequence.

Apart from the verification of internal gates, I/O integrity is also tested, with the following tests being completed:

- I/O levels (i.e., checking noise margin for TTL, ECL, or CMOS I/O pads)
- speed test

With the use of on-chip test structures described in Section 9.6, full-speed wafer testing can be completed with a minimum of connected pins. This can be important in reducing the cost of the wafer test fixture.

In general, manufacturing test generation assumes the function of the circuit/chip is correct. It requires ways of exercising all gate inputs and monitoring all gate outputs.

Example

Consider testing the MIPS microprocessor from Chapter 1. Explain the difference between the tests you would use for logic verification or silicon debug and the tests you would use for manufacturing.

Solution: Logic verification should test that each operation can be performed. For example, a test program might exercise all of the instructions to demonstrate that each one behaves as intended. Logic verification will not necessarily prove that the instruction works for all possible addresses and data values. In contrast, manufacturing tests must prove that every gate operates correctly. They ideally stimulate each gate to produce both a 0 and a 1 to ensure the gate is not damaged. The manufacturing tests may be the only tests applied to a microprocessor prior to it being placed in a system and used. Clearly, it is a challenge to devise a set of tests that is both complete enough that customers receive very few defective chips and short enough to keep testing economical.

9.2 Testers, Test Fixtures, and Test Programs

To test a chip after it is fabricated, you need a tester, a test fixture, and a test program.

9.2.1 Testers and Test Fixtures

A tester is a device that can apply a sequence of stimuli to a chip or system under test and monitor and/or record the results of those operations. Testers come in various shapes and sizes.

To test a chip, one or more of four general types of test fixtures may be required. These are as follows:

- A *probe card* to test at the wafer level or unpackaged die level with a chip tester
- A *load board* to test a packaged part with a chip tester
- A printed circuit board (PCB) for bench-level testing (with or without a tester)
- A PCB with the chip *in situ*, demonstrating the application for which the chip is used

We will concentrate first on the cases where a general-purpose production tester is to be used. Production testers are usually expensive pieces of equipment with configurable I/O ports (drive current, output levels, input levels) and huge amounts of RAM behind each test pin. The tester drives input pins from this memory on a cycle-by-cycle basis and samples and stores the levels on output pins. Figure 9.3 shows a typical production tester. In the background, you can see the four-bay cabinet holding the drive electronics. To the right in the background is the controlling workstation. The test head is shown on the front center. This is where the chip is placed in the load board to be tested.

The probe card or load board for the *device under test* (DUT) is connected to the tester, as shown in Figure 9.4. The test program is compiled and downloaded into the tester and the tests are applied to the bare die or packaged chip. The tester samples the chip outputs and compares the values with those provided by the test program. If there are any differences, the chip is marked as faulty (with an ink dot) and the failing tests may be displayed for reference and stored for later analysis. In the case of a probe card, the card is raised, moved to the next die on the wafer, lowered, and the test procedure repeated. In the case of a load board with automatic part handling, the tested part is removed from the board and sorted into a good or bad bin. A new part is fed to the load board and the test is repeated. In most cases, these procedures take a few seconds for each part tested.

The ability to vary the voltage and timing on a per-pin basis with a tester allows a process known as "Shmooing" to be carried out. For instance, you could sweep V_{DD} from 3 V to 6 V on a 5 V part while varying the tester cycle time. This yields a graph called a *shmoo plot* that shows the speed sensitivity of the part with respect to voltage. Another

FIG 9.3 The Teradyne Catalyst: A typical production tester (Photo: John Haddy, Cisco Systems)

shmoo that is frequently performed is to skew the timing on inputs with respect to the chip clock to look for setup and hold variations. Examples of shmoo plots and their interpretations are given in Section 9.4.

Testers can be very expensive, especially for high-frequency and/or analog/RF chips. The cost of testing each chip depends on the amount of time it must be in the tester. Applying tests to check every node on the chip may be prohibitively costly, so some designs face a tradeoff between test cost and the fraction of defective chips that slip through testing. Tester usage is charged by time, so the shorter a test runs, the cheaper a part is to test.

Example

Suppose a $2 million tester has an expected useful life of two years before it becomes inadequate to test faster next-generation parts. How much does the tester cost per second?

Solution: Dividing the tester cost by the number of seconds in two years gives 3.2 cents/second.

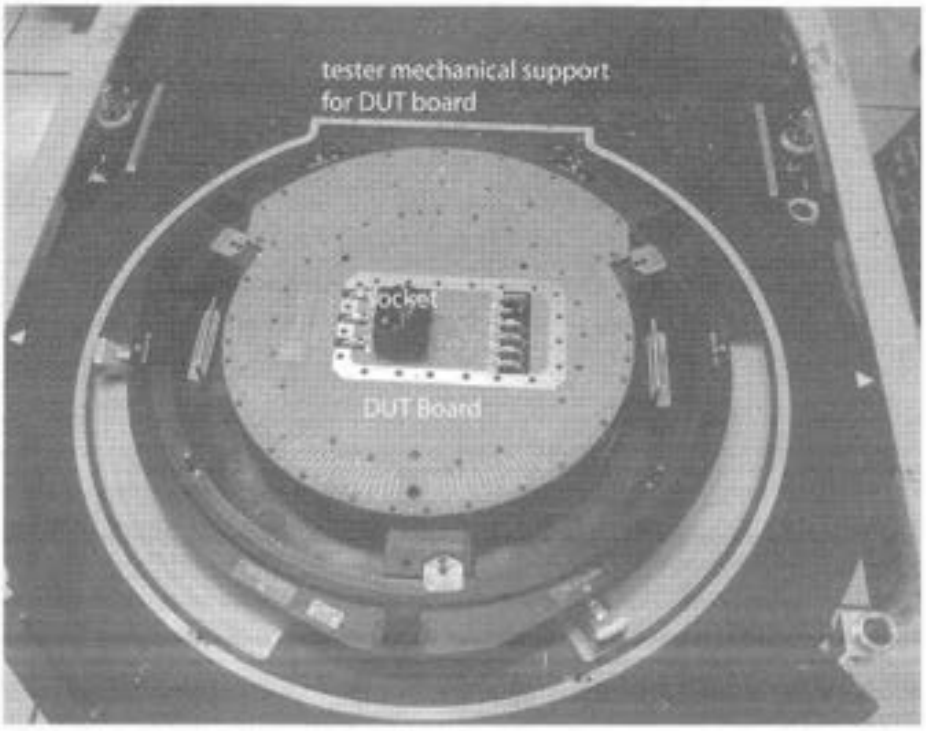


FIG 9.4 Tester load board in test head
(Photo: John Haddy, Cisco Systems)

Testers are available that can be used to test an IC in a laboratory environment. They mirror large production testers, but generally have less functionality (e.g., slower, less memory per pin, less expandability) and are markedly less expensive. A probe card that allows wafer probing or a socketed load board is required for each design. A good logic analyzer with a pattern generator and a socketed test board can also be used to test a chip. Some groups effectively design their own logic analyzers by surrounding a chip with FPGAs and using the logic and RAM within the FPGA to apply and observe test patterns.

9.2.2 Test Programs

The tester requires a *test program* (in verification and test, this is an overloaded term). This program is normally written in a high-level language (for instance, the IMAGE language used by Teradyne is based on C) that supports a library of primitives for a particular tester. The test program specifies a set of input patterns and a set of output *assertions*. If an output does not match the asserted value at the corresponding time, the tester will report an error. Before the patterns and assertions are applied, the test program has to set up the various attributes of a tester such as:

- Set the supply voltages
- Assign mapping between stimulus file signal names and physical tester pins
- Set the pins on the tester to be inputs or outputs and their V_{OH}/ V_{IH} levels
- Set the clock on the tester
- Set the input pattern and output assertion timing

And then on a per chip basis:

- Apply supply voltages
- Apply digital stimulus and record responses
- Check responses against assertions
- Report and log errors

A stimulus or pattern file can be derived from running a simulation on the design. Special *vector change descriptions* (VCDs) are used to compact simulation results. An example of a simple stimulus/pattern file for the case of a full adder is shown below:

```
          III        OO
                     SC
                     UA
                     MR
                      R
          ABC         Y
0         000        00
1         001        10
2         010        10
3         011        01
4         100        10
5         101        01
6         110        01
7         111        11
```

The first line designates the signal directions and shows three inputs (I) and two outputs (O). Reading downwards, the next five lines designate the signal names (A, B, C, SUM, CARRY). Thereafter, each line designates a new *test vector*. The first column is the test vector number. The next three columns are the binary value of the inputs and the following two columns are the expected output values. Each line represents a certain length clock cycle that is asserted by the tester. Signals change after a specified period in relation to an internal clock running at the required test period. Clock generation can be carried out in two different ways. First, the clock can be treated like any other signal, in which case, it takes two tester cycles to complete a single clock cycle: one for the clock low and one for the clock high. Alternatively, a timing generator can be used, which allows the clock rising edge (for instance) to be placed anywhere in the tester cycle. So for instance, if the inputs are changed at the start of the tester cycle, the clock might be programmed to rise at the middle of the cycle.

Each pin on the tester is connected to a function memory, which is used to either drive an input or check an output at a DUT pin. Multiple bits may be required per pin to control tristate input pins or mask outputs when they should be ignored. These memories have finite length, so sometimes with older testers, more than one vector load has to be used to test a part. This normally slows testing as the reload procedure may be slow. Modern testers seldom suffer from this problem.

The clock speed, T_c, is specified, as are supply voltage levels. The time at which pins are driven and sampled is also specified on a pin-by-pin basis (T_p). The format of the test data is usually chosen from Non Return to Zero (NRZ), Return To Zero (RTZ), or other formats such as Surround By Zero (SBZ).

9.2.3 Handlers

An IC handler is responsible for feeding ICs to a test fixture attached to a tester. Chutes or trays containing packaged chips can be used to gravity-feed the devices to the handler, which uses a variety of mechanical means to pick the chips up and place them in the test socket on the load board. The tester stimulus is then applied and chips are binned depending on whether or not they passed the test. It is possible to heat and cool a chuck to test the chip at temperature. However, package-level testing is not normally carried out at temperature because of the time it takes to temperature-cycle the chuck.

An example of a handler is shown in Figure 9.5. This is the NS-6040 from Seiko-Epson. The body of the machine holds the mechanical positioning equipment, while the upper central section supports the test fixture. The light on top indicates a functioning or stopped machine and is designed to be visible across a production floor where many machines might be operating. (The light of another is visible top center.) A screen at the top right provides status information to the operator. The unit has wheels for easy movement, but also has firm footings, which are lowered when the machine is in use.

Handlers add a constant time to the test process, typically around 1 second. Thus, load boards and handlers are often constructed to deal with two or four chips at once to reduce the cost of testing. Because a load board must be designed to fit to a given handler, select the handler before starting design of the load board.

9.3 Logic Verification Principles

9.3.1 Test Benches and Harnesses

A verification *test bench* or *harness* is a piece of HDL code that is placed as a wrapper around a core piece of HDL. In the simplest test bench, inputs are applied to the module under test and at each cycle, the outputs are examined to determine whether they comply with a predefined expected data set. The data set can be derived from another model and available as a file or the value can be computed on the fly.

FIG 9.5 Photograph of an Epson NS-6040 IC handler (Source: John Haddy—Cisco Systems)

Example

For the NCO introduced in Sections 8.2.3 and 8.4.1.1, the following is the core of a test bench that writes the sine values to a file. These may be subsequently used with a program such as MATLAB to calculate distortion or other attributes of interest.

```
// Test Bench for NCO test structure
`timescale 1ns/10ps

module test_nco;
   parameter delay = 50;
   parameter MAX_COUNT = 512;

   reg          reset, clk;
   reg  [15:0]  init_phase, phase_delta;
   wire [7:0]   sine;
   integer      plot_file;
   integer      i;

   initial begin
      init_phase = 0; phase_delta = 0; clk = 0;
   end
```

```
   nco dut(.reset(reset), .fclk(clk), .initial_phase(init_phase),
           .phase_increment(phase_delta), .q(sine));

   always #(delay/2) clk = -clk; // define a clock
   always @(posedge clk)
      $fwrite(plot_file, "%d %d \n", $time, sine);

   // main test loop just cycles for MAX_COUNT cycles
   initial begin
      plot_file = $fopen("plot.out");
      for (i = 0; i < MAX_COUNT; i = i + 1)
         @(posedge clk);
      $finish;
   end
endmodule
```

Alternatively, the desired sine and cosine values can be externally calculated and stored in a file and compared "on the fly." The following code segment (similar to that shown in Appendix A.8) demonstrates that method of testing.

```
...
reg [7:0] expected_sine[511:0];
...
// read in the expected sine values to a file
initial
   begin
      $readmemh("hexsine.dat", expected_sine);
   end
always #(delay/2) clk = -clk; // define a clock
   ...
// get the expected value and compare
for (i = 0; i < MAX_COUNT; i = i + 1)
always @(posedge clk)
   begin
       e_sine = expected_sine[i];
   if (sine !== expected_sine)
       begin
          $display("Error");
          $stop;
       end
   end
...
```

The sine file could be generated in MATLAB using the following code:

```
% MATLAB code generates one cycle of an 8 bit sine wave
t = [1:256];
t = t * 2 * pi / 256;
```

```
% approximates 8 bit resolution using fix
s = fix(sin(t) * 127) + 128;

f = fopen("sine.dat", "w");
f1 = fopen("hexsine.dat", "w");

for i1 = 1:256
  fprintf(f, "%d\n", s(i1));
  fprintf(f1, "%#.2x\n", s(i1));
end

fclose(f);
fclose(f1);
```

This produces the following `hexsine.dat` file:

```
0x80
0x83
0x86
0x89
...
0x71
0x74
0x77
0x7a
0x7d
```

The file comparison method is applicable to a wide range of simulation scenarios as files form a common basis for I/O between different design systems. The notion of a *golden model* is frequently used as the reference for establishing functional equivalence. A golden model might be a model for the system being designed in a high-level language such as C or in a design tool such as MATLAB. The golden model writes expected output files that are used as the basis for comparison.

Simulators usually provide settable break points and single or multiple stepping abilities to allow the designer to step through a test sequence while debugging discrepancies.

9.3.2 Regression Testing

High-level language scripts are frequently used when running large test benches, especially for *regression testing*. Regression testing involves performing a suite of simulations to automatically verify that no functionality has inadvertently changed in a module or set of modules. During a design, it is common practice to run a regression script every night after design activities have concluded to check that bug fixes or feature enhancements have not broken completed modules.

Example

In the software radio example, Figure 8.11 showed a possible architecture that used a combination of an IQ conversion block and a multiplier-based multiprocessor. The following regression testing might be done.

```
Test IQ Conversion
   Test Upconverter
      Test NCO
         Test Read and Write of All Registers
         Test Phase Incrementer
         Test Phase Adder
         Test Sine ROM (Read Contents)
         Test Overall NCO at a set of frequencies
      Test Multiplier
   Test Downconverter
      Test NCO
         ...
      Test Mulitplier
         ...
      Test Low Pass Filter
         ...
Test Microprocessor Memory Core
   Test Microprocessor
      Test ALU
      Test Instruction Decode
      Test Program Counter
      Test Register File Read/Write
      Exhaustive Instruction Test
   Test Memory Read/Write
Test Interprocessor Bus IO
Test IQ Conversion to Processor pathways
Test Overall Software Radio Functionality
```

Note the way in which the correctness of modules is slowly built up by verifying lower-level models first. The low-level tests are gradually built up in complexity until the complete functionality can be verified. At low levels, it is easier to exhaustively verify that logic is correct. For instance, we can verify that the sine ROM is in fact generating a sine wave for one frequency. We then use this knowledge to postulate that it generates correct sine waves for all input frequencies when we verify at the levels above the NCO. At the chip level, we assume that IQ conversion is correct for all combinations of signal frequency and local oscillator frequency even though we may only check a small subset. If we started at the top level and ran a simulation for a few frequencies, we could never have confidence that the lower levels were correct. In addition, if there is a problem, trying to locate the problem by debugging at the top level is futile. Running regression tests from the bottom up is designed to overcome this verification nightmare.

9.3.3 Version Control

Combined with regression testing is the use of versioning, that is, the orderly management of different design iterations. Unix/Linux tools such as CVS are useful for this.

Example

In the software radio example, the regression testing halts at the ALU test in the example given above. Working late, the design leader, Vanessa Eagleeye, examines the CVS history and discovers that Fred Codechanger has made an edit to the ALU design to try a new adder during the day. She is able to revert the code to what was previously working and then rerun the regression test and have a peaceful night's sleep. Fred corrects his mistake the next day and is advised to remember to run the regression verification step before making such hurried edits.

9.3.4 Bug Tracking

Another important tool to use during verification (and in fact the whole design cycle) is a bug-tracking system. Bug-tracking systems such as the Unix/Linux based GNATS allow the management of a wide variety of bugs. In these systems, each bug is entered and the location, nature, and severity of the bug noted. The bug discoverer is noted, along with the perceived person responsible for fixing the bug.

Example

From the example described in the previous section, Vanessa enters a bug report describing the bug. She cites Fred as the person responsible and the level as severe. The next day, Fred fixes the problem and changes the bug status to fixed. The bug report is kept in the system, but does not appear in any listing of outstanding bugs. It is kept to track the re-introduction of bugs, as this might give managers an idea of a problem area in the design management.

Tracking the number of bugs can give you an idea of the rate at which a design is converging toward a finished state. If the trend is downward, the design is converging. On the other hand, an upward trend tends to indicate a design early in its verification cycle.

9.4 Silicon Debug Principles

The area of basic digital debugging was introduced in Section 9.1.2. A major challenge in silicon debugging is when the chip operates incorrectly, but you cannot ascertain the cause by making measurements at the chip pins or scan chain outputs (see Section 9.6.2).

There are a number of techniques for directly accessing the silicon. First, specific signals can be brought to the top of the chip as *probe points*. These are small squares (5–10 μm on a side) of top-level metal that connect to key points in the circuit that the designer has had the foresight to include before debug. The overglass cut mask should specify a hole in the passivation over the probe pads so the metal can be reliably contacted. Typical of these kinds of test points might be internal bias points in linear circuits or perhaps key points in a high-speed signal chain (be careful not to excessively load the circuit to be probed). The exposed squares can be probed with a picoprobe (fine-tipped probe) in a fixture under a microscope. During design, the load of the picoprobe has to be taken into account by providing buffers if necessary. The Model 35 probe from GGB Industries has a capacitance of .05 pF, input resistance of 1.25 MΩ, and frequency response from DC to 26 GHz. It can probe down to a 10 μm by 10 μm window.

The die can also be probed electrically or optically if mechanical contact is not feasible. An *electron beam* (ebeam) probe uses a scanning electron microscope to produce a tightly focused beam of electrons to measure on-chip voltages. Similarly, *Laser Voltage Probing* (LVP) [Lasserre99] involves shining a laser at a circuit and observing the reflected light. The reflections are modulated by the electric fields so switching waveforms can be deduced. However, the probing can be invasive; the stream of photons may disturb sensitive dynamic nodes. *Picosecond Imaging Circuit Analysis* (PICA) [Knebel98] captures faint light emission naturally produced by switching transistors and hence is noninvasive. Silicon is partially transparent to infrared light, so both LVP and PICA can be performed through the substrate from the backside of a chip in a flip-chip package.

On a more coarse scale, infrared (IR) imaging can be used to examine "hot spots" in a chip, which may be the source of problems (for instance, a resistive short between power rails). There are also liquid crystal materials, which can be "painted on" to a die to indicate temperature problems at a coarse resolution.

If the location of the fault is known, a *Focused Ion Beam* (FIB) can be used to cut wires or lay new conductors down. Even with plastic-packaged parts, the plastic can be carefully ground off and these repairs completed. The reason for this kind of tool is that normally in any chip project, time is of the essence and FIB runs are quicker (and cheaper for a few parts) than frequent mask changes. Laser cutting is also possible. Commercial providers such as MEFAS offer these services.

Example

A short between V_{DD} and GND has rendered a chip just back from tapeout nonfunctional. The position of the fault is known and it can be corrected by a cut to the top level metal. Several packaged parts are sent to the FIB house with a location from a given fiducial mark and an accompanying plot of the position of the metal to be cut. The FIB house exposes the die (i.e., by grinding a plastic package). The operator then locates the cut position manually using a microscope and runs the FIB machine. The modified packages are then returned to the designers, where hopefully they celebrate an otherwise useless chip.

Debugging logic circuits will often involve extremely fast or novel circuits that are largely analog in nature. In this case, it is advisable to have a model of the circuit in question available in SPICE. Debugging analog circuits, as with purely digital circuits, involves making an assertion and then trying to prove the assertion is correct. This can begin with a SPICE simulation and then progress to silicon measurement.

Failures causes may be *manufacturing*, *functional*, or *electrical*. Manufacturing failures occur when a chip has a defect or is outside of the parametric specifications. Debug can reject chips with manufacturing problems, although circuits sensitive to weaknesses in the manufacturing process can be changed to improve yield, as will be discussed in Section 9.6.5. Functional failures are logic bugs or physical design errors that cause the chip to fail under all conditions. They arise from inadequate logic verification and are usually the easiest to fix. Electrical failures occur when the chip is logically correct, but malfunctions under certain conditions such as voltage, temperature, or frequency. Section 6.3 addressed many causes of electrical failures. Some electrical failures can be so severe that they appear as functional failures, while others occur rarely and are extremely difficult to reproduce and diagnose.

So-called Shmoo plots can help to debug electrical failures in silicon [Baker97]. A Shmoo plot is often made with voltage on the *X* axis and speed as the *Y* axis. The test vectors are applied at each combination of voltage and clock speed, and the success of the test is recorded. Often, only a set of vectors applicable to a particular module is applied to diagnose a problem in that module. The Shmoo plots on the facing page show a variety of conditions [Josephson02].

A healthy normal chip should operate at increasing frequency as the voltage increases. The brick wall pattern suggests that the chip may be randomly initialized in one of two states, only one of which is correct. For example, a register without a reset signal may randomly have an initial state of 0 or 1. The wall pattern in which the chip fails to operate at any frequency above or below a particular voltage can indicate charge sharing, coupling noise, or a race condition. The reverse speedpath behavior indicates a leakage problem in which a weakly held node leaks to an invalid level before the end of the cycle. At higher voltage, the leakage is exacerbated and appears at shorter clock periods. The floor is a variant on the leakage problem where the part fails at low frequency independent of the voltage. A finger indicates coupling problems dependent on the alignment of the aggressor and victim, where at certain frequencies the alignment always causes a failure.

A shmoo can also plot operating speed against temperature. At cold temperature, FETs are faster, have lower effective resistance, and have higher threshold voltages. A normal shmoo should show speed increasing as temperature decreases. Failures at low temperature could indicate coupling or charge sharing noise exacerbated by faster edge rates. Failures at high temperature could indicate excessive leakage or noise problems exacerbated by the lower threshold voltages. Walls at either temperature could indicate race conditions where the path that wins the race varies with temperature.

Figure 9.6 shows an actual shmoo from the 433 MHz Alpha 21164 [Gronowski96]. At 1.8 V, the chip works for clock periods of 2.3 ns and greater. At higher voltage, the chip can operate at shorter periods, as one would expect.

Clock period in ns on the left, frequency increases going up
Voltage on the bottom, increase left to right

** indicates a failure*

```
1.0   *   *   *   *   *   *
1.1   *   *   *   *   *
1.2   *   *   *   *
1.3   *   *   *
1.4   *   *
1.5   *
     1.0 1.1 1.2 1.3 1.4 1.5
```

Normal
Well-behaved shmoo
Typical Speedpath

```
1.0   *       *       *
1.1       *       *       *
1.2   *       *       *
1.3       *       *       *
1.4   *       *       *
1.5       *       *       *
     1.0 1.1 1.2 1.3 1.4 1.5
```

"Brick Wall"
Bistable
Initialization

```
1.0           *   *   *
1.1           *   *   *
1.2           *   *   *
1.3           *   *   *
1.4           *   *   *
1.5           *   *   *
     1.0 1.1 1.2 1.3 1.4 1.5
```

"Wall"
Fails at a certain voltage
Coupling, charge share, races

```
1.0       *   *   *   *   *
1.1           *   *   *   *
1.2               *   *   *
1.3                   *   *
1.4                       *
1.5                       *
     1.0 1.1 1.2 1.3 1.4 1.5
```

"Reverse speedpath"
Increase in voltage reduces frequency
Speedpath, leakage

```
1.0
1.1
1.2
1.3
1.4   *   *   *   *   *   *
1.5   *   *   *   *   *   *
     1.0 1.1 1.2 1.3 1.4 1.5
```

"Floor"
Works at high but not low frequency
Leakage

```
1.0
1.1
1.2   *   *   *   *
1.3           *   *
1.4
1.5
     1.0 1.1 1.2 1.3 1.4 1.5
```

"Finger"
Fails at a specific point in the shmoo
Coupling

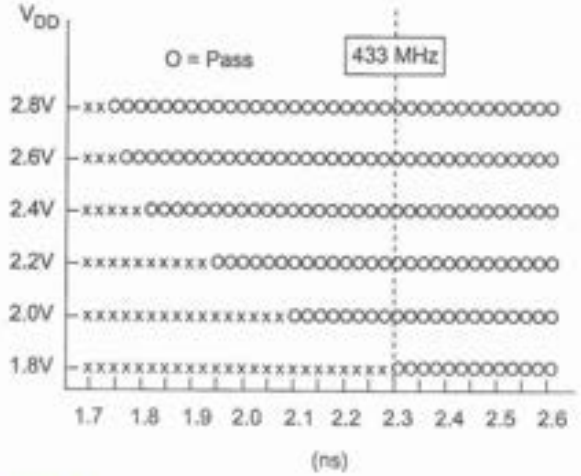


FIG 9.6 Alpha 21164 shmoo

9.5 Manufacturing Test Principles

A critical factor in all VLSI design is the need to incorporate methods of testing circuits. This task should proceed concurrently with any architectural considerations and not be left until fabricated parts are available (as is a recurring temptation to designers).

Figure 9.7(a) shows a combinational circuit with N inputs. To test this circuit exhaustively, a sequence of 2^N inputs (or test vectors) must be applied and observed to fully exercise the circuit. This combinational circuit is converted to a sequential circuit with addition of M registers, as shown in Figure 9.7(b). The state of the circuit is determined by the inputs and the previous state. A minimum of 2^{N+M} test vectors must be applied to exhaustively test the circuit. As observed by [Williams83] more than two decades ago,

> *With LSI, this may be a network with $N = 25$ and $M = 50$, or 2^{75} patterns, which is approximately 3.8×10^{22}. Assuming one had the patterns and applied them at an application rate of 1 μs per pattern, the best time would be over a billion years (10^9).*

Clearly, exhaustive testing is infeasible for most systems, even in the age of multi-GHz processors. Fortunately, the number of potentially nonfunctional nodes on a chip is much smaller than the number of states. Manufacturing test engineers must cleverly devise test vectors that detect any (or nearly any) defective node without requiring so many patterns.

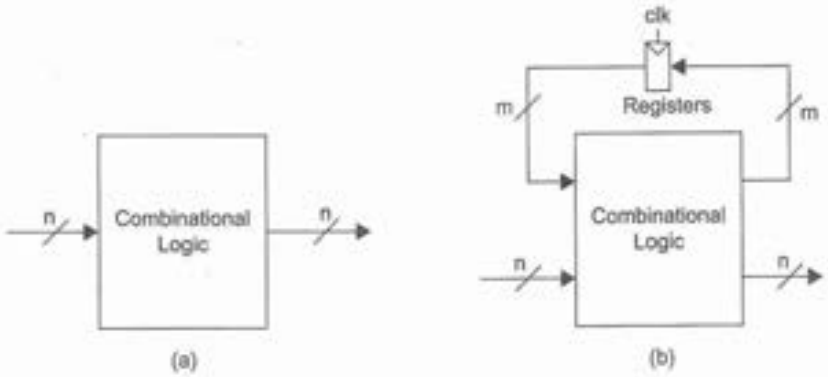


FIG 9.7 The combinational explosion in test vectors

9.5.1 Fault Models

To deal with the existence of good and bad parts, it is necessary to propose a *fault model*, i.e., a model for how faults occur and their impact on circuits. The most popular model is called the *Stuck-At* model. The *Short Circuit/Open Circuit* model can be a closer fit to reality, but is harder to incorporate into logic simulation tools.

9.5.1.1 Stuck-at Faults In the Stuck-At model, a faulty gate input is modeled as a *stuck at zero* (Stuck-At-0, S-A-0) or *stuck at one* (Stuck-At-1, S-A-1). This model dates from board-level designs, where it was determined to be adequate for modeling faults. Figure 9.8 illustrates how an S-A-0 or S-A-1 fault might occur. These faults most frequently occur due to gate oxide shorts (the nMOS gate to GND or the pMOS gate to V_{DD}) or metal-to-metal shorts.

9.5.1.2 Short-circuit and Open-circuit Faults Other models include *stuck-open* or *shorted* models [Jayasumana91]. Two bridging or shorted faults are shown in Figure 9.9. The short *S1* results in an S-A-0 fault at input A, while short *S2* modifies the function of the gate. It is evident that to ensure the most accurate modeling, faults should be modeled at the transistor level because it is only at this level that the complete circuit structure is known. For instance, in the case of a simple NAND gate, the intermediate node between the series nMOS transistors is hidden by the schematic. This implies that test generation should ideally take account of possible shorts and open circuits at the switch level [Galiay80]. Expediency dictates that most existing systems rely on Boolean logic representations of circuits and stuck-at fault modeling.

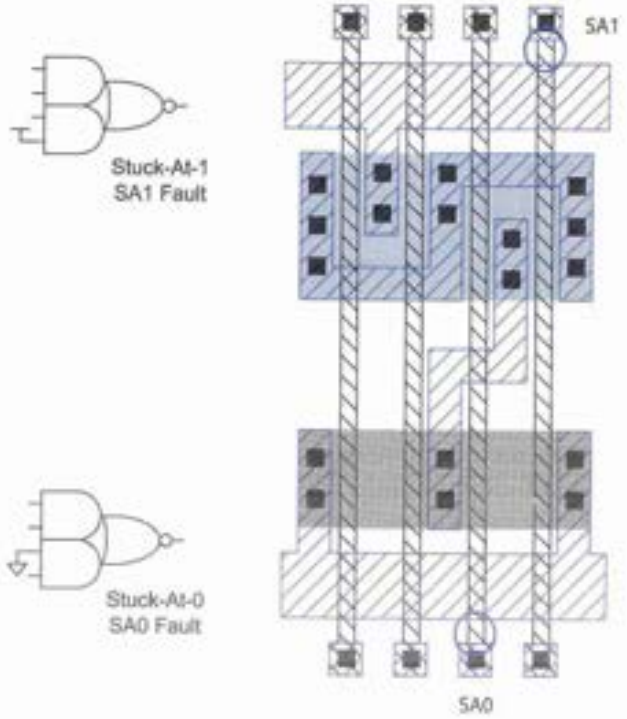


FIG 9.8 CMOS stuck-at faults

A particular problem that arises with CMOS is that it is possible for a fault to convert a combinational circuit into a sequential circuit. This is illustrated in Figure 9.10 for the case of a 2-input NOR gate in which one of the transistors is rendered ineffective. If nMOS transistor A is stuck open, then the function displayed by the gate will be

$$Z = \overline{A + B} + \overline{B}Z' \tag{9.1}$$

where Z' is the previous state of the gate. As another example, if either pMOS transistor is missing, the node would be arbitrarily charged (i.e., it might be high due to some weird charging sequence) until one of the nMOS transistors discharged the node. Thereafter, it would remain at zero, barring charge leakage effects.

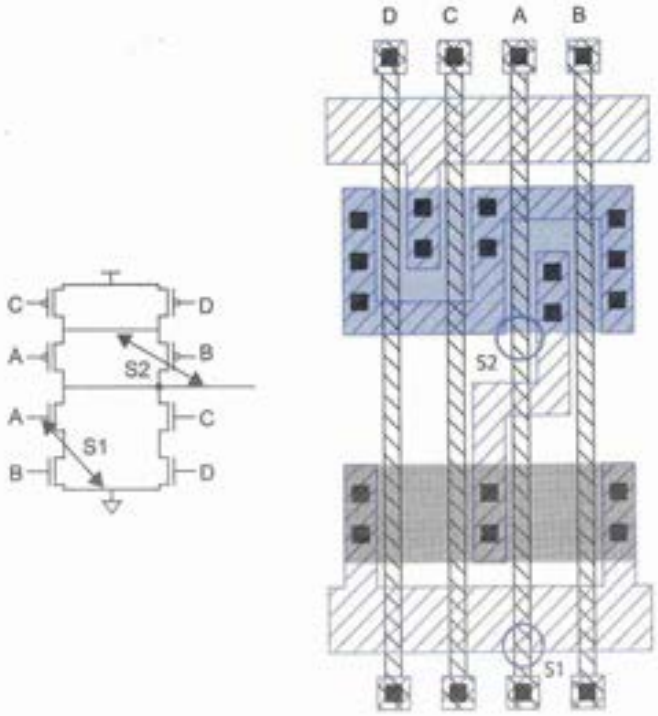


FIG 9.9 CMOS bridging faults

It is also possible for transistors to exhibit a stuck-open or stuck-closed state. Stuck-closed states can be detected by observing the static V_{DD} current (I_{DD}) while applying test vectors. Consider the fault shown in Figure 9.11, where the drain connection on a pMOS transistor in a 2-input NOR gate is shorted to V_{DD}. This could physically occur if stray metal (caused by a speck of dust at the photolithography stage) overlapped the V_{DD} line and drain connection as shown. If we apply the test vector 01 or 10 to the A and B inputs and measure the static I_{DD} current, we will notice that it rises to some value determined by size of the nMOS transistors.

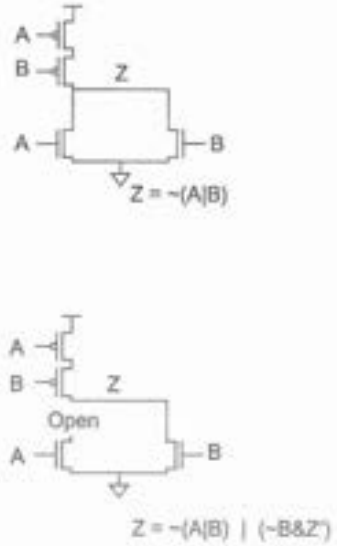


FIG 9.10 A CMOS open fault that causes sequential faults

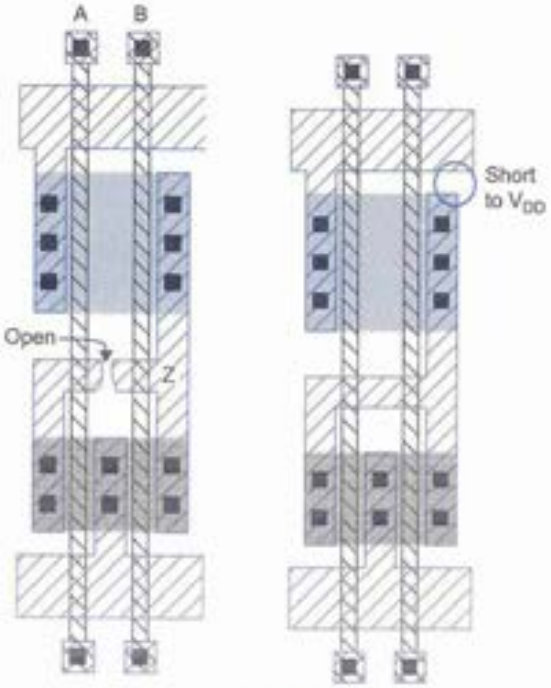


FIG 9.11 A defect that causes static I_{DD} current

9.5.2 Observability

The *observability* of a particular circuit node is the degree to which you can observe that node at the outputs of an integrated circuit (i.e., the pins). This metric is relevant when you want to measure the output of a gate within a larger circuit to check that it operates correctly. Given the limited number of nodes that can be directly observed, it is the aim of good chip designers to have easily observed gate outputs. Adoption of some basic design for test techniques can aid tremendously in this respect. Ideally, you should be able to observe directly or with moderate indirection (i.e., you may have to wait a few cycles) every gate output within an integrated circuit. While at one time this aim was hindered by the expense of extra test circuitry and a lack of design methodology, current processes and design practices allow you to approach this ideal. Section 9.6 examines a range of methods for increasing observability.

9.5.3 Controllability

The *controllability* of an internal circuit node within a chip is a measure of the ease of setting the node to a 1 or 0 state. This metric is of importance when assessing the degree of

difficulty of testing a particular signal within a circuit. An easily controllable node would be directly settable via an input pad. A node with little controllability might require many hundreds or thousands of cycles to get it to the right state. Often you will find it impossible to generate a test sequence to set a number of poorly controllable nodes into the right state. It should be the aim of good chip designers to make all nodes easily controllable. In common with observability, the adoption of some simple design for test techniques can aid in this respect tremendously. Making all flip-flops resettable via a global reset signal is one step toward good controllability.

9.5.4 Fault Coverage

A measure of goodness of a set of test vectors is the amount of *fault coverage* it achieves. That is, for the vectors applied, what percentage of the chip's internal nodes were checked? Conceptually, the way in which the fault coverage is calculated is as follows. Each circuit node is taken in sequence and held to 0 (S-A-0), and the circuit is simulated with the test vectors comparing the chip outputs with a *known good machine*—a circuit with no nodes artificially set to 0 (or 1). When a discrepancy is detected between the *faulty machine* and the good machine, the fault is marked as detected and the simulation is stopped. This is repeated for setting the node to 1 (S-A-1). In turn, every node is stuck (artificially) at 1 and 0 sequentially. The fault coverage of a set of test vectors is the percentage of the total nodes that can be detected as faulty when the vectors are applied. To achieve world-class quality levels, circuits are required to have in excess of 98.5% fault coverage.

9.5.5 Automatic Test Pattern Generation (ATPG)

Historically, in the IC industry, logic and circuit designers implemented the functions at the RTL or schematic level, mask designers completed the layout, and test engineers wrote the tests. In many ways, the test engineers were the Sherlock Holmes of the industry, reverse engineering circuits and devising tests that would test the circuits in an adequate manner. For the longest time, test engineers implored circuit designers to include extra circuitry to ease the burden of test generation. Happily, as processes have increased in density and chips have increased in complexity, the inclusion of test circuitry has become less of an overhead for both the designer and the manager worried about the cost of the die. In addition, as tools have improved, more of the burden for generating tests has fallen on the designer. To deal with this burden, *Automatic Test Pattern Generation* (ATPG) methods have been invented. The use of some form of ATPG is standard for most digital designs.

Commercial ATPG tools can achieve excellent fault coverage. However, they are computation-intensive and often must be run on servers or compute farms with many parallel processors. Some tools use statistical algorithms to predict the fault coverage of a set of vectors without performing as much simulation. Adding scan and built-in self-test, as described in Section 9.6, improves the observability of a system and can reduce the number of test vectors required to achieve a desired fault coverage.

9.5.6 Delay Fault Testing

The fault models dealt with until this point have neglected timing. Failures that occur in CMOS could leave the functionality of the circuit untouched, but affect the timing. For instance, consider the layout shown in Figure 9.12 for an inverter gate composed of paralleled nMOS and pMOS transistors. If an open circuit occurs in one of the nMOS transistor source connections to GND, then the gate would still function but with increased t_{pd}. In addition, the fault now becomes sequential as the detection of the fault depends on the previous state of the gate.

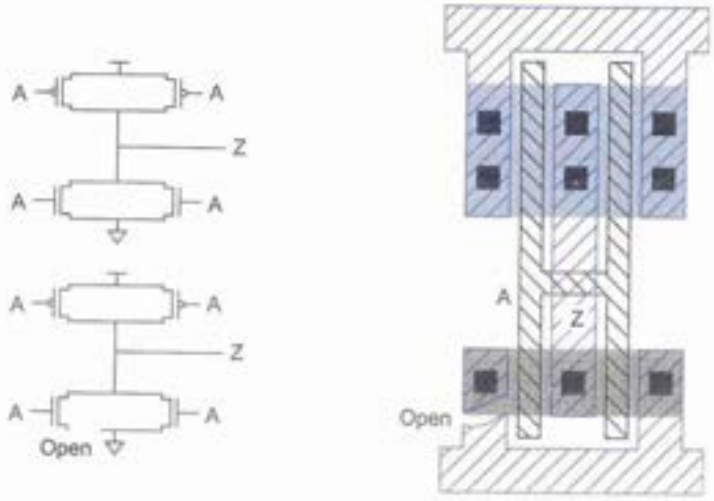


FIG 9.12 An example of a delay fault

Delay faults may be caused by crosstalk [Paul02]. Delay faults can also occur more often in SOI logic through the history effect. Software has been developed to model the effect of delay faults and is becoming more important as a failure mode as processes scale.

9.6 Design for Testability

The keys to designing circuits that are testable are controllability and observability. Restated, controllability is the ability to set (to 1) and reset (to 0) every node internal to the circuit. Observability is the ability to observe, either directly or indirectly, the state of any node in the circuit. Good observability and controllability reduce the cost of manufacturing test because they allow high fault coverage with relatively few test vectors. Moreover, they can be essential to silicon debug because physically probing most internal signals has become so difficult.

We will first cover three main approaches to what is commonly called *Design for Testability* (DFT). These may be categorized as:

- *Ad hoc* testing
- Scan-based approaches
- Built-in self-test (BIST)

Following this, we will look at the application of these techniques to particular types of circuits. In this treatment we will look at:

- Random logic (multilevel standard cell, two-level PLA)
- Regular logic arrays (datapaths)
- Memories (RAM, ROM, CAM)

9.6.1 *Ad hoc* testing

Ad hoc test techniques, as their name suggests, are collections of ideas aimed at reducing the combinational explosion of testing. They are summarized here for historical reasons. They are only useful for small designs where scan, ATPG, and BIST are not available. A complete scan-based testing methodology is recommended for all digital circuits. Having said that, common techniques for ad hoc testing involve:

- Partitioning large sequential circuits
- Adding test points
- Adding multiplexers
- Providing for easy state reset

A technique classified in this category is the use of the bus in a bus-oriented system for test purposes. Each register has been made loadable from the bus and capable of being driven onto the bus. Here, the internal logic values that exist on a data bus are enabled onto the bus for testing purposes.

Frequently, multiplexers can be used to provide alternative signal paths during testing. In CMOS, transmission gate multiplexers provide low area and delay overhead.

Any design should always have a method of resetting the internal state of the chip within a single cycle or at most a few cycles. Apart from making testing easier, this also makes simulation faster as a few cycles are required to initialize the chip.

In general, *ad hoc* testing techniques represent a bag of tricks developed over the years by designers to avoid the overhead of a systematic approach to testing, as will be described in the next section. While these general approaches are still quite valid, process densities and chip complexities necessitate a structured approach to testing.

9.6.2 Scan Design

The *scan-design* strategy for testing has evolved to provide observability and controllability at each register. In designs with scan, the registers operate in one of two modes. In *normal mode*, they behave as expected. In *scan mode*, they are connected to form a giant shift register called a *scan chain* spanning the whole chip. By applying *N* clock pulses in scan mode, all *N* bits of state in the system can be shifted out and new *N* bits of state can be shifted in. Therefore, scan mode gives easy observability and controllability of every register in the system.

Modern scan is based on the use of scan registers, as shown in Figure 9.13. The scan register is a *D* flip-flop preceded by a multiplexer. When the *SCAN* signal is deasserted, the register behaves as a conventional register, storing data on the *D* input. When *SCAN* is asserted, the data is loaded from the *SI* pin, which is connected in shift register fashion to the previous register *Q* output in the scan chain.

For the circuit shown, to load the scan chain, *SCAN* is asserted and *CLK* is pulsed eight times to load the first two ranks of 4-bit registers with data. *SCAN* is deasserted and *CLK* is asserted for one cycle to operate the circuit normally with predefined inputs. *SCAN* is then reasserted and *CLK* asserted eight times to read the stored data out. At the same time, the new register contents can be shifted in for the next test. Testing proceeds in this manner of serially clocking the data through the scan register to the right point in the circuit, running a single system clock cycle and serially clocking the data out for observation. In this scheme, every input to the combinational block can be controlled and every output can be observed. In addition, running a random pattern of 1's and 0's through the scan chain can test the chain itself.

Test generation for this type of test architecture can be highly automated. ATPG techniques can be used for the combinational blocks and, as mentioned, the scan chain is easily tested. The prime disadvantage is the area and delay impact of the extra multiplexer in the scan register. Designers (and managers alike) are in widespread agreement that this cost is more than offset by the savings in debug time and production test cost.

9.6.2.1 Parallel Scan You can imagine that serial scan chains can become quite long, and the loading and unloading can dominate testing time. A fairly simple idea is to split the chains into smaller segments. This can be done on a module-by-module basis or completed automatically to some specified scan length. Extending this to the limit yields an extension to serial scan called *random access scan* [Ando80]. To some extent, this is similar to that used inside FPGAs to load and read the control RAM.

The basic idea is shown in Figure 9.14. The figure shows a two-by-two register section. Each register receives a column (`column<m>`) and row (`row<n>`) access signal along with a row data line (`data<n>`). A global write signal (`write`) is connected to all registers. By asserting the row and column access signals in conjunction with the write signal, any register can be read or written in exactly the same method as a conventional RAM. The notional logic is shown to the right of the four registers. Implementing the logic required at the transistor level can reduce the overhead for each register.

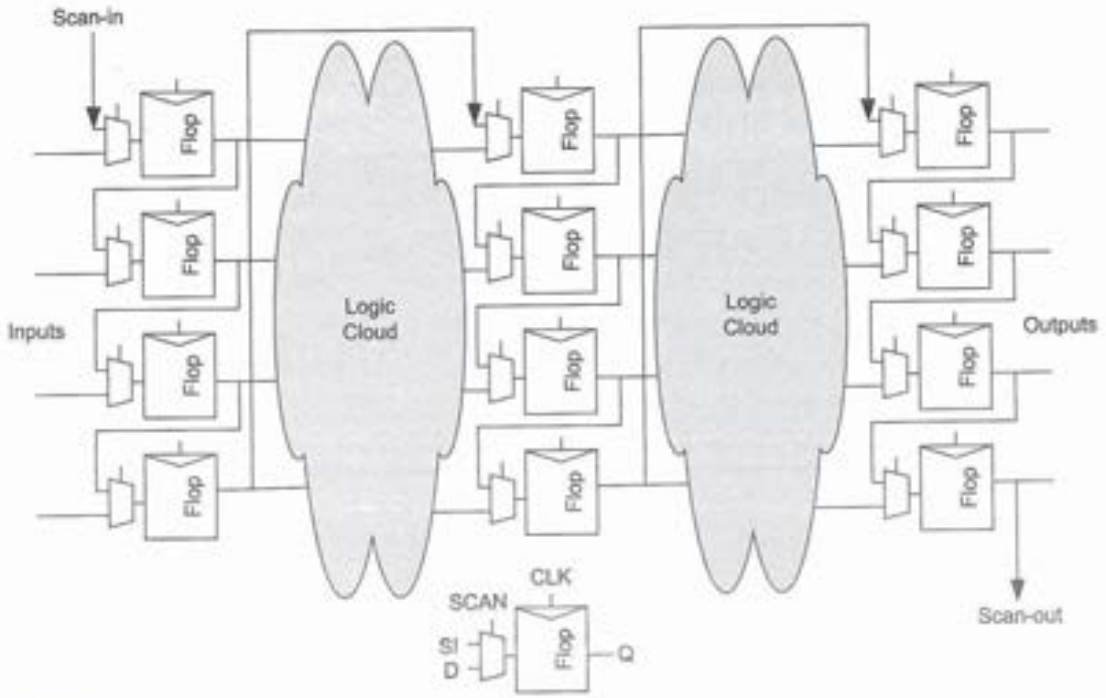


FIG 9.13 Scan-based testing

9.6.2.2 Partial Scan Sometimes, making every register scannable is too expensive, so only a partial set is scanned. In the CORDIC structure introduced in Section 8.2.4, each CORDIC slice has three m-bit registers. Converting all of these to scan registers may not be desirable. As the structure is a data pipeline, the test registers can be placed on the input and output of the pipeline as shown below in Figure 9.15. Partial scan is a throwback to *ad hoc* testing and should be avoided except in unusual circumstances.

9.6.2.3 Circuit Design of Scannable Elements As we have seen, an ordinary flip-flop can be made scannable by adding a multiplexer on the data input, as shown in Figure 9.16(a). Figure 9.16(b) shows a circuit design for such a scan register using a transmission-gate multiplexer. The setup time increases by the delay of the extra transmission gate in series with the D input as compared to the ordinary static flip-flop in Figure 7.19(b). Figure 9.16(c) shows a circuit using clock gating to obtain nearly the same setup time as the ordinary flip-flop. In either design, if a clock enable is used to stop the clock to unused portions of the chip, care must be taken that ϕ always toggles during scan mode.

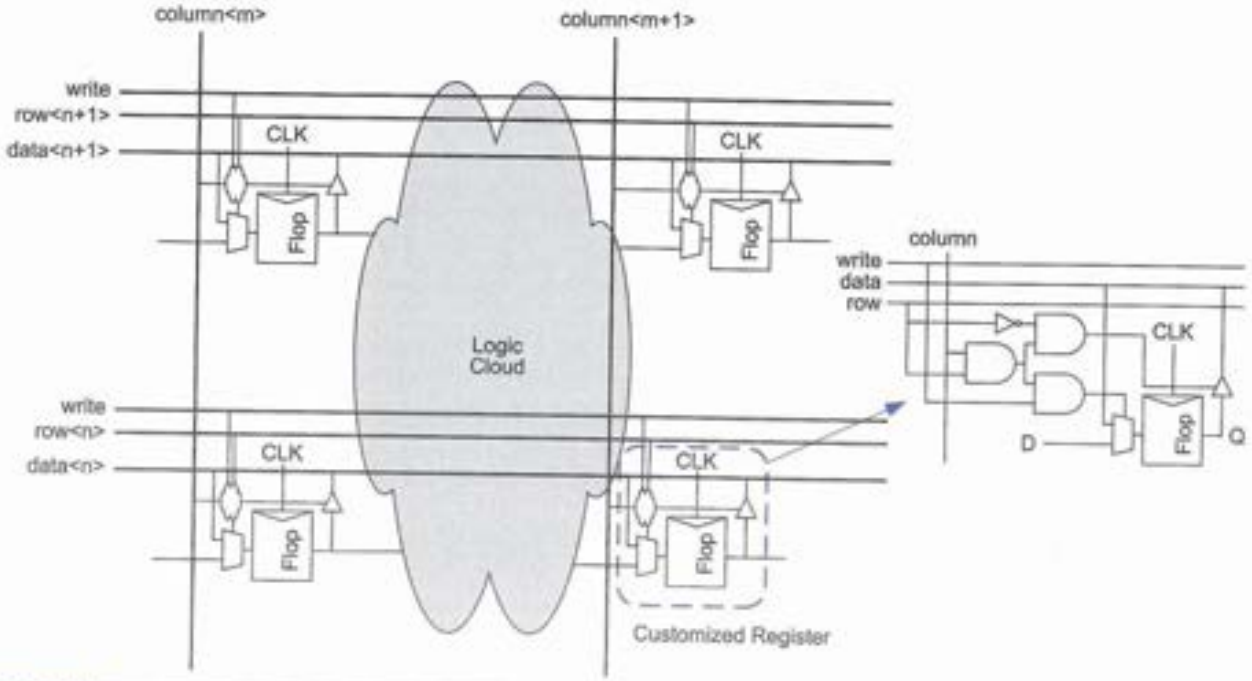


FIG 9.14 Parallel scan—basic structure

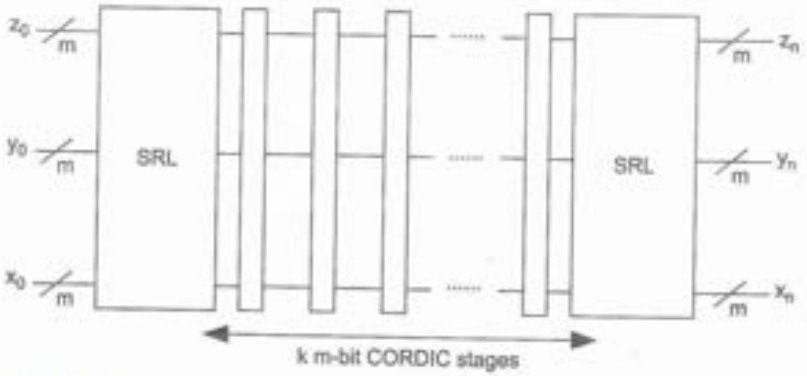


FIG 9.15 Example of partial scan as applied to the CORDIC processor

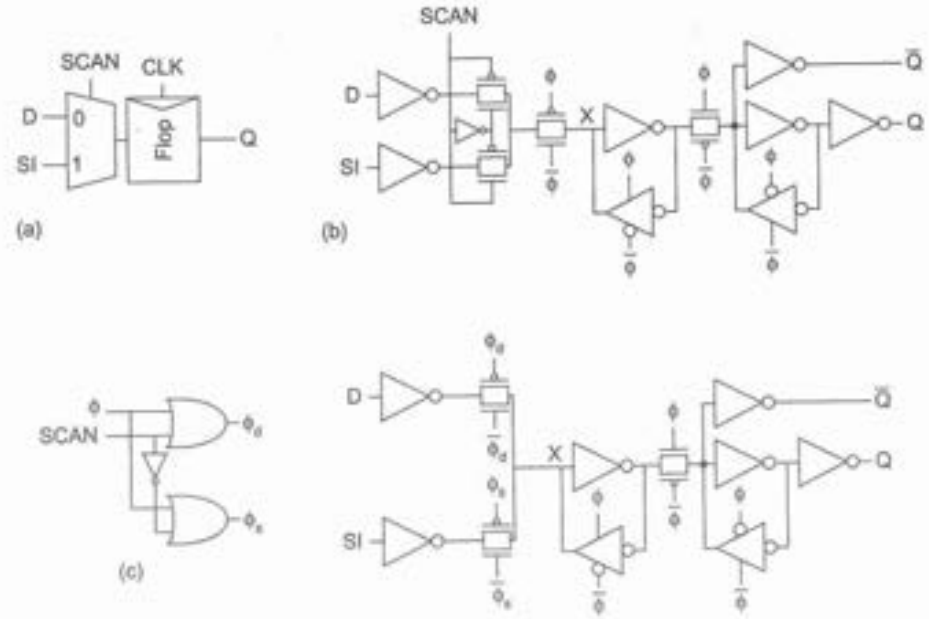


FIG 9.16 Scannable flip-flops

During scan mode, the flip-flops are connected back-to-back. Clock skew can lead to hold time problems in the scan chain. These problems can be overcome by adding delay buffers on the *SI* input to flip-flops that might see large clock skews. Another approach is to use nonoverlapping clocks to ensure hold times. For example, the *Level Sensitive Scan Design* (LSSD) methodology developed at IBM uses flip-flops with two-phase nonoverlapping clocks like those of Figure 7.21. During scan mode, a scan clock ϕ_s is toggled in place of ϕ_2, as shown in Figure 9.17. The nonoverlapping clocks also prevent hold time problems in normal operation, but increase the sequencing overhead of the flip-flop. Alternatively, ϕ_1 and ϕ_2 can be complementary clocks, but ϕ_s can be nonoverlapping to prevent races. Figure 9.17(c) shows a conventional design using a weak feedback inverter on the master latch that can be overpowered when either the ϕ_2 or ϕ_s transmission gates are on. Figure 9.17(d) shows a design from the PowerPC 603 microprocessor using a generalized tristate feedback [Gerosa94]. Figure 9.17(e) shows another gate-level LSSD flip-flop design [Eichelberger78]. Such a design is substantially larger and slower than a conventional pass-transistor circuit, so it is primarily of historical interest. In the IBM LSSD methodology, ϕ_s, ϕ_1, ϕ_2, and *SI* are often called *A*, *B*, *C*, and *I*, respectively.

Systems using latches can also be modified for scan. Typically, a scan input and an extra *slave scan latch* are added to convert the latch into a scannable flip-flop. Figure 9.18

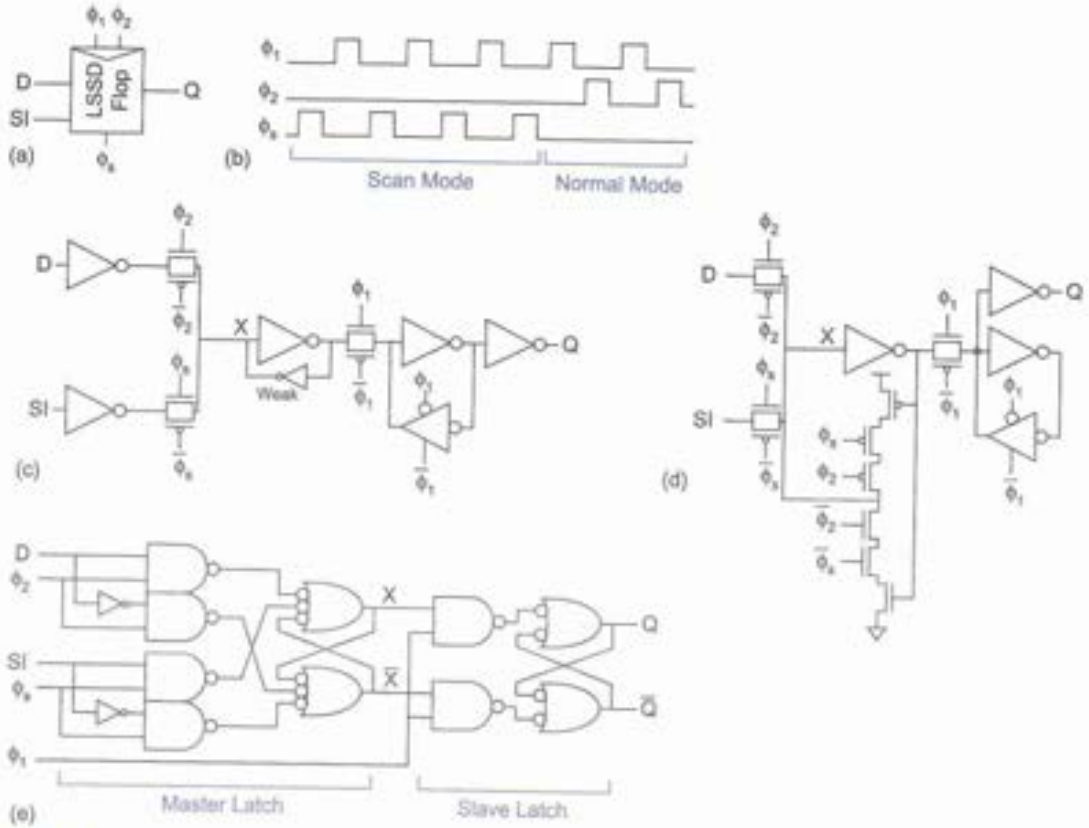


FIG 9.17 LSSD flip-flops

shows a scannable transparent latch. During scan, the global clock is stopped low, so ϕ_1 is low and the latch is opaque. Then a two-phase nonoverlapping scan clock ϕ_{1s} and ϕ_{2s} is toggled to march the data through the scan chain. The *SO* scan-out terminal of each latch connects to the *SI* scan-in terminal of the next latch. Figure 9.18(c) shows a faster and more compact, but less robust version of the scannable latch suitable for custom datapaths [Harris01a]. Scanning one latch in each cycle is adequate to provide good observability and controllability in a system; there is no need to scan the ϕ_2 latch.

The same principle applies to pulsed latches. Figure 9.19 shows the scannable Naffziger pulsed latch used on the Itanium 2 [Naffziger02] (see also Section 7.3.3). It uses a single-phase scan clock. The global clock is stopped during scan so the pulsed latches remain opaque. The scan input overpowers the feedback node *Y* to avoid loading the critical path from *D* to $\overline{Q}$. The transmission gate latch driving *SO* has a dynamic node *Z*, so ϕ_s has a limit on how long it can be high to properly retain data during scan. This is handled

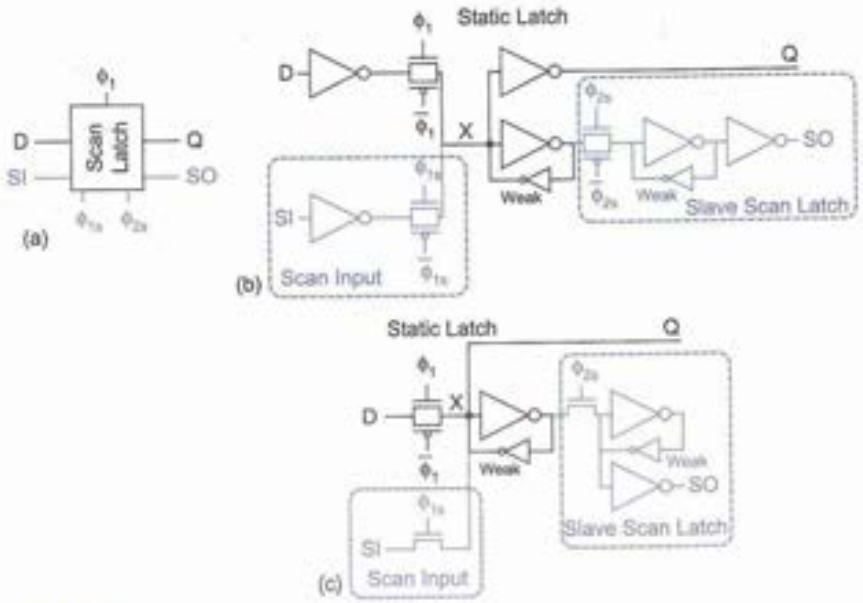


FIG 9.18 Scannable transparent latches

on-chip with a clock chopper that converts the external low-frequency scan clock into an on-chip ϕ_s with short pulses. The scan chain must also be checked for hold time races. Note that the *SO* transmission gate is ON during normal operation, loading the $\overline{Q}$ output and increasing power consumption through spurious transitions on *Z* and *SO*. Many designers would elect to use a second scan clock wire to avoid these problems.

Domino pipelines also can be scanned. Traditional domino pipelines incorporate scan into the two-phase transparent latches on the half-cycle boundaries. Skew-tolerant domino eliminates the latches and must include scan directly in the domino gates. One natural point to scan is the last gate of each cycle.

Figure 9.20(a) shows how to make the last ϕ_4 gate of each cycle in a skew-tolerant domino pipeline scannable [Harris01a]. The last dynamic gate has a full keeper and thus will retain its state when either high or low. The scan technique resembles that of a transparent latch from Figure 9.18(c). The key is to turn off both the precharge and the evaluation transistors so the output node floats and behaves like a master latch. Then a two-phase scan clock is toggled to shift data first onto the master node and then into a slave scan latch. These scan clocks are again called ϕ_{1s} and ϕ_{2s} and bear no relationship to the domino clocks ϕ_1 and ϕ_2. *gclk* is stopped low, so ϕ_4 is high and the precharge transistor is off. A special clock gater forces ϕ_{4s} low during scan to turn the evaluation transistor off. When scan is complete, *gclk* rises so the next ϕ_1 domino gate resumes normal operation. This scan approach adds a small amount of loading on the critical path through the

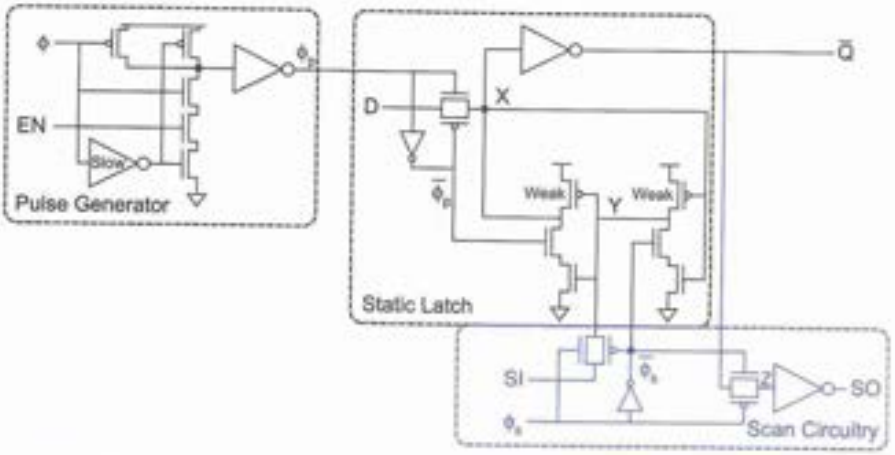


FIG 9.19 Scannable Naffziger pulsed latch

dynamic gate. Figure 9.20(b) shows a clock gater that produces the domino phases. It uses an SR latch to stop and release ϕ_{4_e} during scan, as illustrated in Figure 9.20(c). The gater also accepts an enable to stop the domino clocks when the pipeline is idle.

The Itanium 2 provides domino scan in a similar fashion, but with a single-phase scan clock that is compatible with scan of the Naffziger pulsed latches [Naffziger02]. The last domino gate in each half-cycle uses a dynamic latch converter, as was discussed in Section 7.5.5.2. Scan circuitry can be added to the DLC in much the same way as it is added to a latch.

Robust scan circuitry obeys a number of rules to avoid electrical failures. *SI* is locally buffered to prevent problems with directly driving diffusion inputs and overdriving feedback inside the latch. The output is also buffered so noise cannot back drive the state node. Two-phase nonoverlapping scan clocks prevent hold-time problems, and static feedback on the state node allows low-frequency operation. All internal nodes should swing rail-to-rail. These rules can be bent to save area at the expense of greater electrical verification on the scan chain, as was done for the Itanium 2.

9.6.3 Built-in Self-Test (BIST)

Self-test and built-in test techniques, as their names suggest, rely on augmenting circuits to allow them to perform operations upon themselves that prove correct operation. These techniques add area to the chip for the test logic, but reduce the test time required and thus can lower the overall system cost. [Stroud02] offers extensive coverage of the subject from the implementer's perspective.

One method of testing a module is to use *signature analysis* [Frowerk77, Nadig77] or *cyclic redundancy checking*. This involves using a *pseudo-random sequence generator* (PRSG)

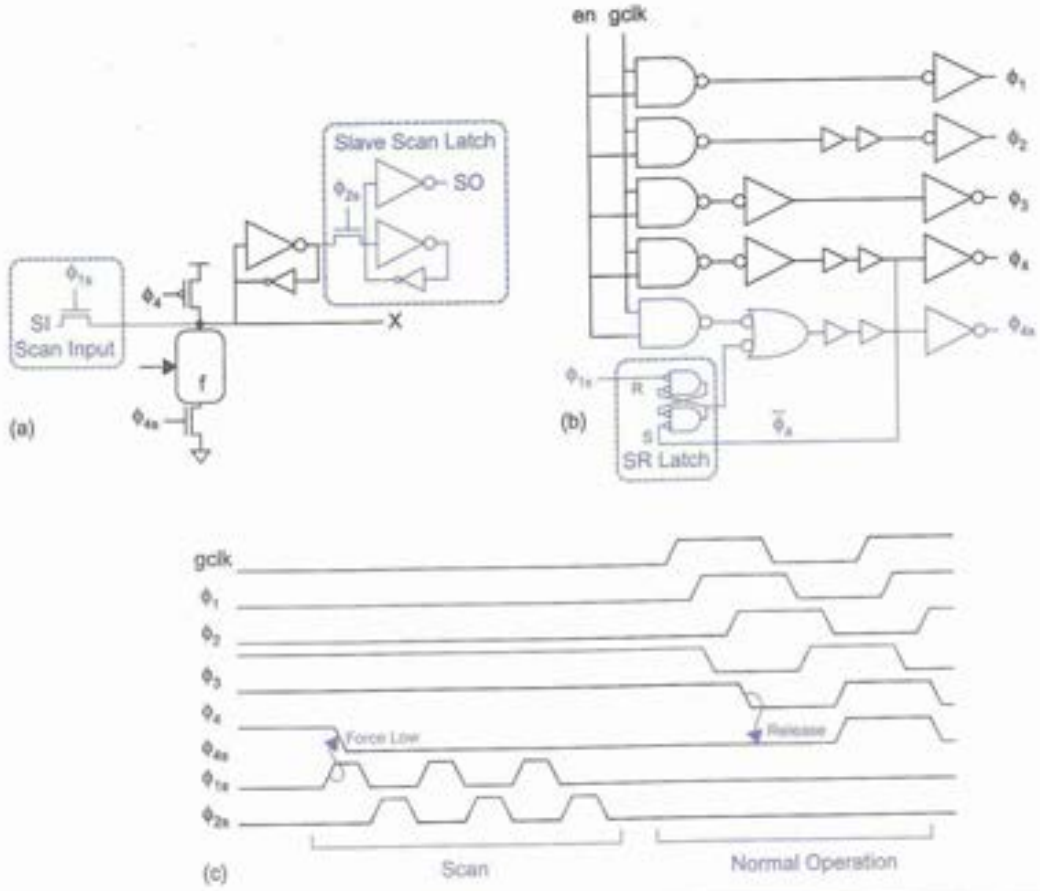


FIG 9.20 Scannable dynamic gate for four-phase skew-tolerant domino

to produce the input signals for a section of combinational circuitry and a *signature analyzer* to observe the output signals.

A PRSG is defined by a polynomial of some length n. It is constructed from a *linear feedback shift register* (LFSR), which in turn is made of n flip-flops connected in a serial fashion, as shown in Figure 9.22(a). The XOR of particular outputs are fed back to the input of the LFSR. An n-bit LFSR will cycle through 2^n-1 states before repeating the sequence. LFSRs are discussed further in Section 10.5.2. They are described by a *characteristic polynomial* indicating which bits are fed back. A *complete feedback shift register* (CFSR), shown in Figure 9.22(b), includes the zero state that may be required in some test situations [Wang86]. An n-bit LFSR is converted to an n-bit CFSR by adding an $n-1$ input NOR gate. When in state 0...01, the next state is 0...00. When in state 0...00,

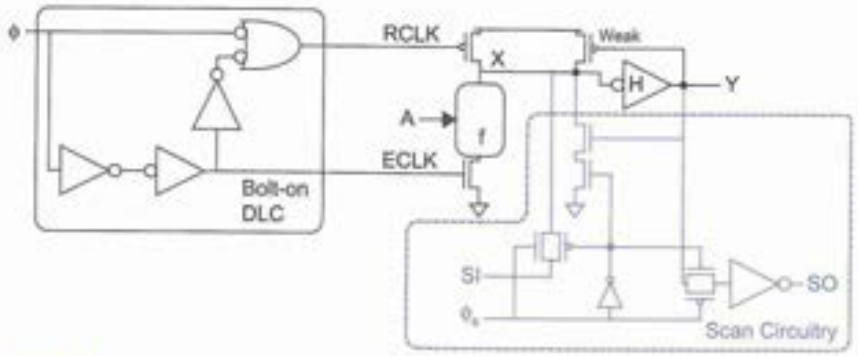


FIG 9.21 Itanium 2 scannable domino gate

the next state is 10…0. Otherwise, the sequence is the same. Alternatively, the bottom n bits of an $n + 1$-bit LFSR can be used to cycle through the all zeros state without the delay of the NOR gate.

A signature analyzer receives successive outputs of a combinational logic block and produces a *syndrome* that is a function of these outputs. The syndrome is reset to 0, and then XORed with the output on each cycle. The syndrome is swizzled each cycle so that a fault in one bit is unlikely to cancel itself out. At the end of a test sequence, the LFSR contains the syndrome that is a function of all previous outputs. This can be compared with the correct syndrome (derived by running a test program on the good logic) to determine whether the circuit is good or bad. If the syndrome contains enough bits, it is extremely improbable that a defective circuit will produce the correct syndrome.

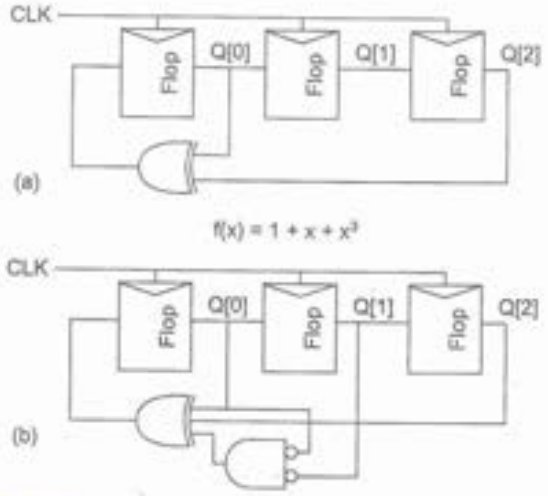
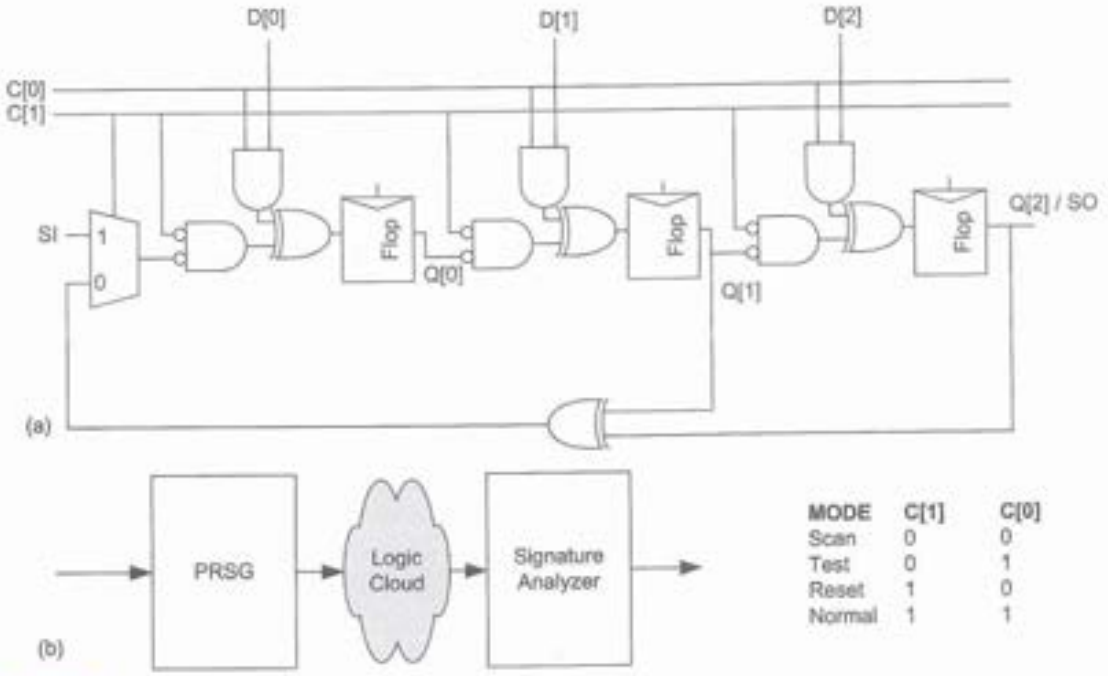


FIG 9.22 Pseudo-random sequence generator

9.6.3.1 BILBO The combination of signature analysis and the scan technique creates a structure known as *BILBO*—for *Built-In Logic Block Observation* [Koenemann79]—or *BIST*—for *Built-In Self-Test*. The 3-bit BILBO register shown in Figure 9.23 is a scannable, resettable register that also can serve as a pattern generator and signature analyzer. *C*[1:0] specifies the mode of operation. In the reset mode (10), all the flip-flops are synchronously initialized to 0. In normal mode (11), the flip-flops behave normally with their *D* input and *Q* output. In scan mode (00), the flip-flops are configured as a 3-bit shift register between *SI* and *SO*. Note that there is an inversion between each stage. In test mode (01), the register behaves as a pseudo-random sequence generator or signature analyzer. If all the *D* inputs are held low, the *Q* outputs loop through a pseudo-random bit sequence, which can serve as the input to the combinational logic. If the *D* inputs are taken from the combinational logic output, they are swizzled with the existing state to produce the syndrome. In summary, BIST is performed by first resetting the syndrome in the output register. Then both registers are placed in the test mode to produce the pseudo-random inputs and calculate the syndrome. Finally, the syndrome is shifted out through the scan chain.

Various companies have commercial design aid packages that support BIST. LogicVision has a package called Logic BIST, which takes a synthesized netlist and adds the scan registers and PRSG circuits automatically. It then checks the fault coverage and provides



MODE	C[1]	C[0]
Scan	0	0
Test	0	1
Reset	1	0
Normal	1	1

FIG 9.23 Built-In Logic Block Observation (BILBO/BIST): (a) individual register (b) use in a system

generated scripts to use boundary scan to run tests on the final chip. As an example, on a WLAN modem chip comprising roughly 1 million gates, a full at-speed test takes under a second with BIST. This comes with roughly a 7.3% overhead in the core area (but actually zero because the design was pad limited) and a 99.7% fault coverage level. The WLAN modem parts designed in this way were fully tested in less than ten minutes on receipt of first silicon. This kind of test method is incredibly valuable for productivity in manufacturing test generation.

Example

To test a 16-bit CORDIC using BIST, three identical 16-bit LFSRs are designed using the characteristic polynomial to be defined in Table 10.7. These LFSRs are placed at the start of the pipeline as shown in Figure 9.24. Similarly, three signature analyzers are placed at the end of the pipeline. When in BIST mode, the CORDIC can test at full operational speed. For instance, if we run 16384 cycles at 100 MHz, the total test takes 163.84 μs.

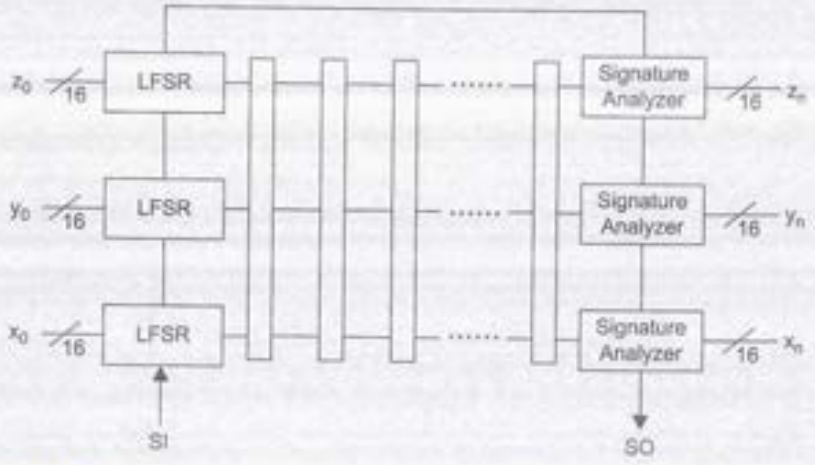


FIG 9.24 CORDIC with BIST

9.6.3.2 Memory Self-test Testing large memories on a production tester can be expensive because they contain so many bits and thus require so many test vectors. Embedding self-test circuits with the memories can reduce the number of external test vectors that have to be run. A typical read/write memory (RAM) test program for an *M*-bit address memory might be as follows [Nair78, Dekker90]:

```
FOR i=0 to M-1 write(~data)
FOR i=0 to M-1 read(~data) then write(data)
```

```
FOR i=0 to M-1 read(data) then write(~data)
FOR i=M-1 to 0 read(~data) then write(xFF)
FOR i=M-1 to 0 read(data) then write(~data)
```

where `data` is 1 and `~data` is 0 for a single-bit memory or a selected set of patterns for an n-bit word. For an 8-bit memory, `data` might be `x00`, `x55`, `xAA`, and `xFF`. These patterns test writing all zeroes, all ones, and alternating ones and zeroes. An address counter, some multiplexers, and a simple state machine result in a low-overhead self-test structure for read/write memories. [Oshawa87] describes a 4-Mbit RAM with self-test. The self-test consists of 256 K cycles that input a checkerboard pattern of alternating 1's and 0's to test for cell-to-cell interference. This is followed by 256 K cycles in which the data is read out. Then a complemented checkerboard is written and read. A total of 1 million cycles provide a test sufficient for system maintenance.

ROM memories can be tested by placing a signature analyzer at the output of the ROM and incorporating a test mode that cycles through the contents of the ROM. A significant advantage of all self-test methods is that testing can be performed when the part is in the field. With care, self-test can even be done during normal system operation.

Example

For the ROM used in the NCO, BIST circuitry can be added as shown in Figure 9.25. Here, a signature analyzer has been added to the output of the ROM as part of the output register. The phase counter is used to cycle through the ROM. The advantage of this style of test is that it is self-contained and can be embedded in the circuitry and so can form part of a self-test capability. The circuit is initialized and run for 256 cycles, and then the resulting signature can be read out and compared with the required value. Note that the signature analyzer is part of the "active" circuitry and so can be run at extremely high speeds—much higher than might be possible over a test bus that extends across the entirety of a chip. Again, this emphasizes the efficacy of a self-test strategy.

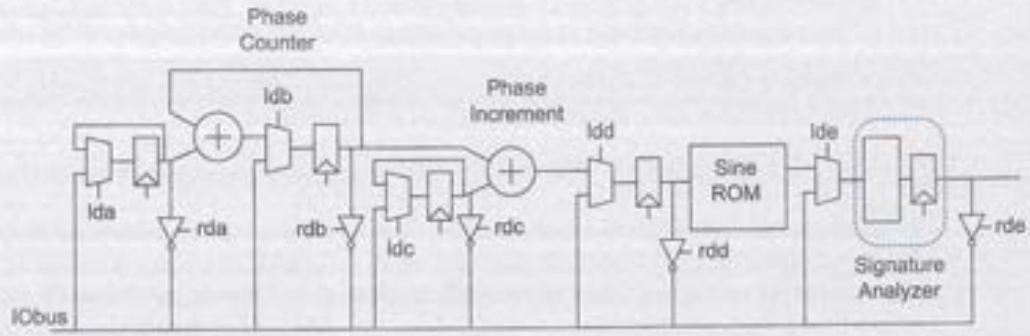


Fig 9.25 NCO ROM testing

9.6.4 IDDQ Testing

Bridging faults were introduced in Section 9.5.1.2. A method of testing for bridging faults is called IDDQ test (V_{DD} supply current Quiescent) or supply current monitoring [Acken83, Lee92]. This relies on the fact that when a complimentary CMOS logic gate is not switching, it draws no DC current (except for leakage). When a bridging fault occurs, then for some combination of input conditions, a measurable DC I_{DD} will flow. Testing consists of applying the normal vectors, allowing the signals to settle, and then measuring I_{DD}. As potentially only one gate is affected, the IDDQ test has to be very sensitive. In addition, to be effective, any circuits that draw DC power such as pseudo-nMOS gates or analog circuits have to be disabled. Dynamic gates can also cause problems. As current measuring is slow, the tests must be run slower (of the order of 1 ms per vector) than normal, which increases the test time.

IDDQ testing can be completed externally to the chip by measuring the current drawn on the V_{DD} line or internally using specially constructed test circuits. This technique gives a form of indirect massive observability at little circuit overhead. However, as subthreshold leakage current increases, IDDQ testing ceases to be effective because variations in subthreshold leakage exceed currents caused by the faults.

9.6.5 Design for Manufacturability

Circuits can be optimized for manufacturability to increase their yield. This can be done in a number of different ways.

9.6.5.1 Physical At the physical level (i.e., mask level), the yield and hence manufacturability can be improved by reducing the effect of process defects. The design rules for particular processes will frequently have guidelines for improving yield. The following list is representative.

- Increase the spacing between wires where possible—this reduces the chance of a defect causing a short circuit.
- Increase the overlap of layers around contacts and vias—this reduces the chance that a misalignment will cause an aberration in the contact structure.
- Increase the number of vias at wire intersections beyond one if possible—this reduces the chance of a defect causing an open circuit.

Increasingly, design tools are dealing with these kinds of optimizations automatically.

9.6.5.2 Redundancy Redundant structures can be used to compensate for defective components on a chip. For example, memory arrays are commonly built with extra rows. During manufacturing test, if one of the words is found to be defective, the memory can be reconfigured to access the spare row instead. Laser-cut wires or electrically programmable fuses can be used for configuration. Similarly, if the memory has many banks and one or more are found to be defective, they can be disabled, possibly even under software control.

9.6.5.3 Power Elevated power can cause failure due to excess current in wires, which in turn can cause metal migration failures. In addition, high-power devices raise the die temperature, degrading device performance and, over time, causing device parameter shifts. The method of dealing with this component of manufacturability is to minimize power through design techniques described elsewhere in this text. In addition, a suitable package and heat sink should be chosen to remove excess heat.

9.6.5.4 Process Spread We have seen that process simulations can be carried out at different process corners. Monte Carlo analysis, which was introduced in Section 5.5.6, can provide better modeling for process spread and can help with centering a design within the process variations.

9.6.5.5 Yield Analysis When a chip has poor yield or will be manufactured in high volume, dice that fail manufacturing test can be taken to a laboratory for yield analysis to locate the root cause of the failure. If particular structures are determined to have caused many of the failures, the layout of the structures can be redesigned. For example, during volume production ramp-up for a major microprocessor, the silicide over long thin polysilicon lines was found to often crack and raise the wire resistance. This in turn led to slower-than-expected operation for the cracked chips. The layout was modified to widen polysilicon wires or strap them with metal wherever possible, boosting the yield at higher frequencies.

9.7 Boundary Scan

Up to this point we have concentrated on the methods of testing individual chips. Many system defects occur at the board level, including open or shorted printed circuit board traces and incomplete solder joints. At the board level, "bed-of-nails" testers historically were used to test boards. In this type of a tester, the board-under-test is lowered onto a set of test points (nails) that probe points of interest on the board. These can be sensed (the observable points) and driven (the controllable points) to test the complete board. At the chassis level, software programs are frequently used to test a complete board set. For instance, when a computer boots, it might run a memory test on the installed memory to detect possible faults.

The increasing complexity of boards and the movement to technologies such as multichip modules (MCMs) and surface mount technologies (with an absence of through-board vias) resulted in system designers agreeing on a unified scan-based methodology called *boundary scan* for testing chips at the board (and system) level. Boundary scan was originally developed by the Joint Test Access Group and hence is commonly referred to as JTAG. Boundary scan has become a popular standard interface for controlling BIST features as well.

The IEEE 1149 boundary scan architecture [IEEE1149.1-01, Parker03] is shown in Figure 9.26. All of the I/O pins of each IC on the board are connected serially in a stan-

dardized scan chain accessed through the *Test Access Port* (TAP) so that every pin can be observed and controlled remotely through the scan chain. At the board level, ICs obeying the standard can be connected in series to form a scan chain spanning the entire board. Connections between ICs are tested by scanning values into the outputs of each chip and checking that those values are received at the inputs of the chips they drive. Moreover, chips with internal scan chains and BIST can access those features through boundary scan to provide a unified testing framework.

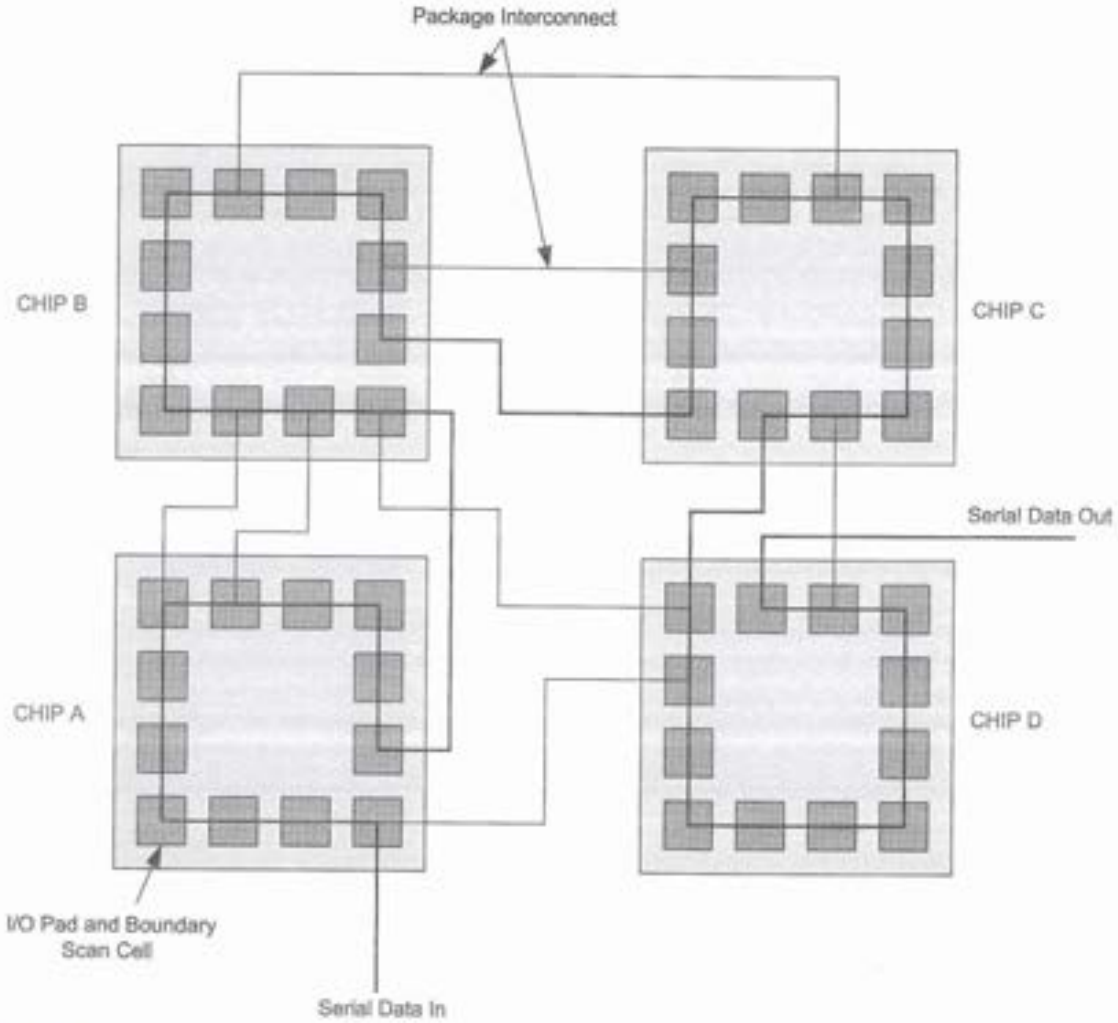


FIG 9.26 Boundary scan architecture

9.7.1 The Test Access Port (TAP)

The *Test Access Port* has four or five single-bit connections:

* *TCK*	Test Clock	Input	clocks tests into and out of the chip
* *TMS*	Test Mode Select	Input	controls test operations
* *TDI*	Test Data In	Input	test data into the chip
* *TDO*	Test Data Out	Output	test data out of the chip; driven only when TAP controller is shifting out test data
* *TRST**	Test Reset Signal	Input	optional active low signal to asynchronously reset the TAP controller if no power-up reset signal is automatically generated by the chip

When the chip is in normal mode, *TRST** and *TCK* are held low and *TMS* is held high to disable boundary scan. To prevent race conditions, inputs are sampled on the rising edge of *TCK* and outputs toggle on the falling edge.

9.7.2 The Test Logic Architecture and Test Access Port

The basic test architecture is shown in Figure 9.27. It consists of

* The TAP interface pins
* A set of two or more test-data registers (DR) to collect data from the chip
* An instruction register (IR) specifying the type of test to perform
* A TAP controller, which controls the scan of bits through the instruction and test-data registers

The TAP controller is a small finite-state machine that configures the system. In one mode, it scans an instruction into the instruction register specifying what boundary scan should do. In another mode, it scans data in and out of the test-data registers. The specification requires at least two test-data registers: the boundary scan register and the bypass register. The boundary scan register is associated with all the inputs and outputs on the chip so that boundary scan can observe and control the chip I/Os. The bypass register is a single flip-flop used to accelerate testing by avoiding shifting data into the boundary scan registers of idle chips when only a single chip on the board is being tested. Internal scan chain, BIST, or configuration registers can be treated as optional additional data registers controlled by boundary scan.

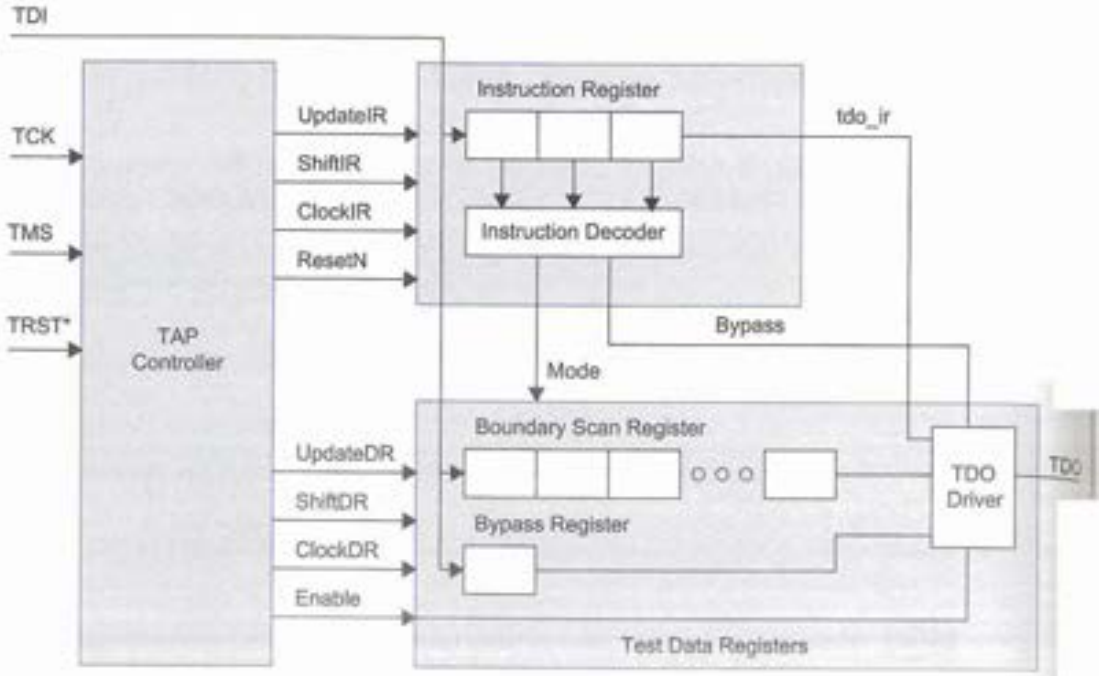


FIG 9.27 TAP architecture

9.7.3 The TAP Controller

The TAP controller is a 16-state FSM that proceeds from state to state based on the *TCK* and *TMS* signals. It provides signals that control the test-data registers and the instruction register. These include serial shift clocks and update clocks.

The state transition diagram is shown in Figure 9.28. The TAP controller is initialized to Test-Logic-Reset on power-up by *TRST** or an internal power-up detection circuit. It moves from one state to the next on the rising edge of *TCK* based on the value of *TMS*.

A typical test sequence will involve clocking *TCK* at some rate and setting *TRST** to 0 for a few cycles and then returning this signal to 1 to reset the TAP controller state machine. *TMS* is then toggled to traverse the state machine for whatever operation is required. These operations include serially loading an instruction register or serially loading or reading data registers that are used to test the chip. A variety of these operations will be described as this section unfolds.

The following Verilog code implements the TAP controller. The *TRST** is named `trstn`. Note that the controller produces gate clocks to control the data and instruction registers at the appropriate times.

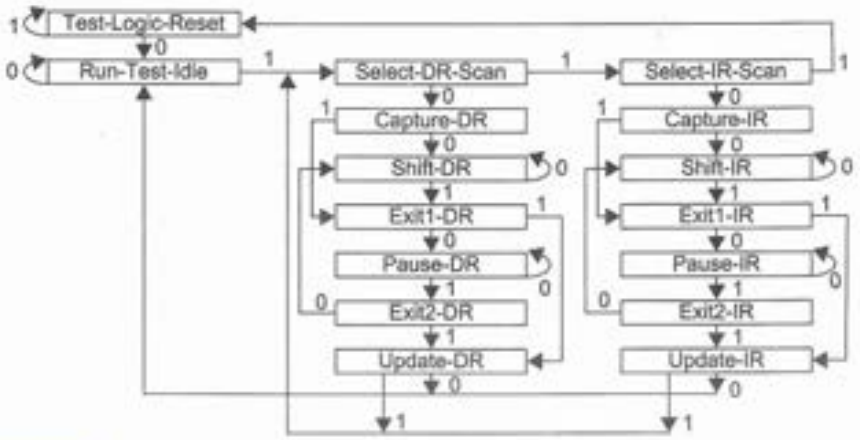


FIG 9.28 TAP controller state diagram

```
// TAP Controller States
`define TEST_LOGIC_RESET 4'b1111
`define RUN_TEST_IDLE    4'b1100
`define SELECT_DR_SCAN   4'b0111
`define CAPTURE_DR       4'b0110
`define SHIFT_DR         4'b0010
`define EXIT1_DR         4'b0001
`define PAUSE_DR         4'b0011
`define EXIT2_DR         4'b0000
`define UPDATE_DR        4'b0101
`define SELECT_IR_SCAN   4'b0100
`define CAPTURE_IR       4'b1110
`define SHIFT_IR         4'b1010
`define EXIT1_IR         4'b1001
`define PAUSE_IR         4'b1011
`define EXIT2_IR         4'b1000
`define UPDATE_IR        4'b1101

module tapcontroller(input      tms, tck, trstn,
                     output reg ShiftIR, ShiftDR,
                     output     ClockIR, ClockDR,
                     output     UpdateIR, UpdateDR,
                     output reg Resetn, Enable);

   reg [3:0] state;

   // next state logic
   always @(posedge tck, negedge trstn)
      if (~trstn) state = `TEST_LOGIC_RESET;
      else case (state)
```

```
        `TEST_LOGIC_RESET:state = (tms) ? state : `RUN_TEST_IDLE;
        `RUN_TEST_IDLE:   state = (tms) ? `SELECT_DR_SCAN : state;
        `SELECT_DR_SCAN:  state = (tms) ? `SELECT_IR_SCAN : `CAPTURE_DR;
        `CAPTURE_DR:      state = (tms) ? `EXIT1_DR : `SHIFT_DR;
        `SHIFT_DR:        state = (tms) ? `EXIT1_DR : state;
        `EXIT1_DR:        state = (tms) ? `UPDATE_DR : `PAUSE_DR;
        `PAUSE_DR:        state = (tms) ? `EXIT2_DR : state;
        `EXIT2_DR:        state = (tms) ? `UPDATE_DR : `SHIFT_DR;
        `UPDATE_DR:       state = (tms) ? `SELECT_DR_SCAN : `RUN_TEST_IDLE;
        `SELECT_IR_SCAN:  state = (tms) ? `TEST_LOGIC_RESET : `CAPTURE_IR;
        `CAPTURE_IR:      state = (tms) ? `EXIT1_IR : `SHIFT_IR;
        `SHIFT_IR:        state = (tms) ? `EXIT1_IR : state;
        `EXIT1_IR:        state = (tms) ? `UPDATE_IR : `PAUSE_IR;
        `PAUSE_IR:        state = (tms) ? `EXIT2_IR : state;
        `EXIT2_IR:        state = (tms) ? `UPDATE_IR : `SHIFT_IR;
        `UPDATE_IR:       state = (tms) ? `SELECT_DR_SCAN : `RUN_TEST_IDLE;
      endcase

    // Clock registers on rising edge of tck at end of state
    // otherwise idle clock high
    assign ClockIR  = tck | ~((state == `CAPTURE_IR) | (state == `SHIFT_IR));
    assign ClockDR  = tck | ~((state == `CAPTURE_DR) | (state == `SHIFT_DR));

    // Update registers on falling edge of tck
    assign UpdateIR = ~tck & (state == `UPDATE_IR);
    assign UpdateDR = ~tck & (state == `UPDATE_DR);

    // Change control signals on falling edge of tck
    always @(negedge tck, negedge trstn)
      if (~trstn) begin
         ShiftIR <= 0;
         ShiftDR <= 0;
         Resetn  <= 0;
         Enable  <= 0;
      end else begin
         ShiftIR <= (state == `SHIFT_IR);
         ShiftDR <= (state == `SHIFT_DR);
         Resetn  <= ~(state == `TEST_LOGIC_RESET);
         Enable  <= (state == `SHIFT_IR) | (state == `SHIFT_DR);
      end
endmodule
```

9.7.4 The Instruction Register

The instruction register has to be at least two bits long. Recall that boundary scan requires at least two data registers. The instruction register specifies which data register will be placed in the scan chain when the DR is selected. It also determines where the DR will

load its value from in the Capture-DR state and whether the values will be driven to output pads or core logic. Three instructions are required to be supported:

- BYPASS—This instruction places the bypass register in the DR chain so that the path from *TDI* to *TDO* involves only a single flip-flop. This allows specific chips to be tested in a serial scan chain without having to shift through the lengthy shift register stages in all the chips. This instruction is represented with all 1's in the IR.
- SAMPLE/PRELOAD—This instruction places the boundary scan registers (i.e., at the chip's I/O pins) in the DR chain. In the Capture-DR state, it copies the chip's I/O values into the DRs. They can then be scanned out in successive Shift-DR states. New values are shifted into the DRs, but not driven onto the I/O pins yet.
- EXTEST—This instruction allows for the testing of off-chip circuitry. It is similar to SAMPLE/PRELOAD, but also drives the values from the DRs onto the output pads. By driving a known pattern onto the outputs of some chips and checking for that pattern at the input of other chips, the integrity of connections between chips can be verified.

In addition to these instructions, the following are also recommended (others can be defined as needed):

- INTEST— This instruction allows for single-step testing of internal circuitry via the boundary scan registers. It is similar to EXTEST, but also drives the chip core with signals from the DRs rather than from the input pads.
- RUNBIST— This instruction is used to activate internal self-testing procedures within a chip.

Note that the instruction encodings are not part of the specification (except that BYPASS is all 1's). The component designer must document what encodings were selected for each instruction.

A typical IR bit is shown in Figure 9.29. Observe that it contains two flip-flops. The ClockIR flip-flops of each bit are connected to form a shift register. They are loaded with a constant value from the Data input in the Capture-IR state, and then are shifted out in the Shift-IR state while new values are shifted in. The constant value is user-defined, but must have a 01 pattern in the least significant two bits so that the integrity of the scan chain can be verified. In the Update-IR state, the contents of the shift register are copied in parallel to the IR output to load the entire instruction at once. This prevents the IR from momentarily having illegal values while new instructions are shifted in. On reset, the IR should be asynchronously loaded with an innocuous instruction such as BYPASS that does not interfere with the normal behavior of the core logic.

A minimal implementation of a 3-bit control register is shown below. Notice the instruction encoding definitions. This implements the six registers required for a 3-bit instruction. The instruction is decoded to produce `mode_in`, `mode_out`, and `bypass` signals to control the data registers, as will be discussed in the next sections.

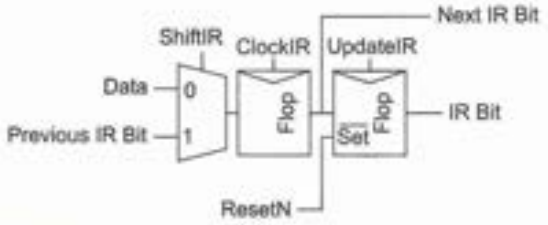


FIG 9.29 Instruction bit implementation

```
// Instructions
`define BYPASS          3'b111
`define SAMPLE_PRELOAD  3'b101
`define EXTEST          3'b110
`define NOP             3'b001
`define INTEST          3'b100

module inst_reg(input  tdi,
                input  Resetn, ClockIR, UpdateIR, ShiftIR,
                output tdo_ir, mode_in, mode_out, bypass);

   reg [2:0] shiftreg, instreg;

   always @(posedge ClockIR)
      shiftreg <= ShiftIR ? {tdi, shiftreg[2:1]} : `NOP;
   always @(posedge UpdateIR, negedge Resetn)
      if (~Resetn) instreg <= `BYPASS;
      else instreg <= shiftreg;

   assign tdo_ir = shiftreg[0];
   assign bypass = (instreg == `BYPASS);
   assign mode_in = (instreg == `INTEST);
   assign mode_out = (instreg == `INTEST) || (instreg == `EXTEST);
endmodule
```

9.7.5 Test Data Registers

The test data registers are used to set the inputs of modules to be tested and collect the results of running tests. The simplest data register configuration consists of a boundary scan register (passing through all I/O pads) and a bypass register (1 bit long). Figure 9.30 shows a generalized view of the data registers in which an internal data register has been added. This register might represent the scan chain within the chip or a BILBO signature register. Thus, boundary scan elegantly incorporates other built-in test structures. A multiplexer under the control of the TAP controller selects which data register is routed to the *TDO* pin. When internal data registers are added, the IR decoder must produce extra control signals to select which one is in the DR chain for a particular instruction.

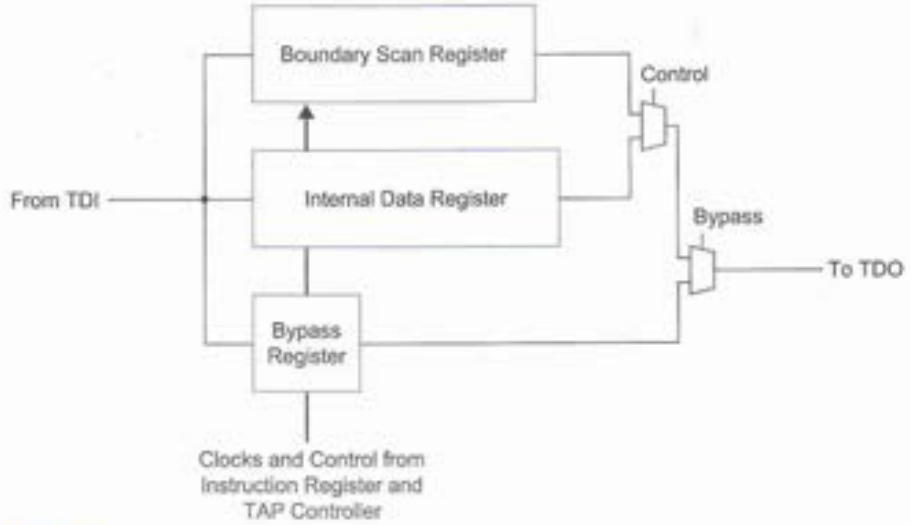


FIG 9.30 Test-data registers

9.7.5.1 Boundary Scan Register The boundary scan register connects to all of the I/O circuitry. Like the instruction register, it internally consists of a shift register for the scan chain and an additional bank of flip-flops to update the outputs in parallel. An extra multiplexer on the output allows the boundary scan register to override the normal path through the I/O pad so it can observe and control inputs and outputs. The schematic and symbol for a single bit of the boundary scan register are shown in Figure 9.31.

The boundary scan register can be configured as an input pad or output pad, as shown in Figure 9.32(a and b). As an input, the register receives `DataIn` from the pad and sends `Qout` to the core logic in the chip. As an output, the register receives `DataIn` from the core logic and drives `Qout` to a pad. Tristate and bidirectional pads use two or three boundary scan register cells, as shown in Figure 9.32(c and d).

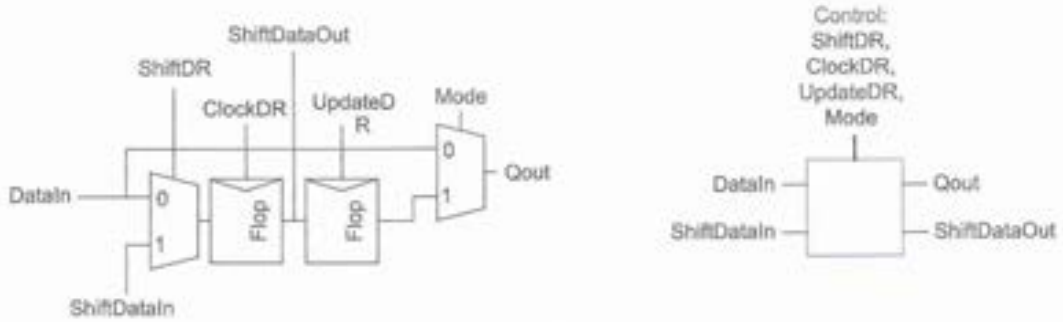


FIG 9.31 Boundary scan register bit

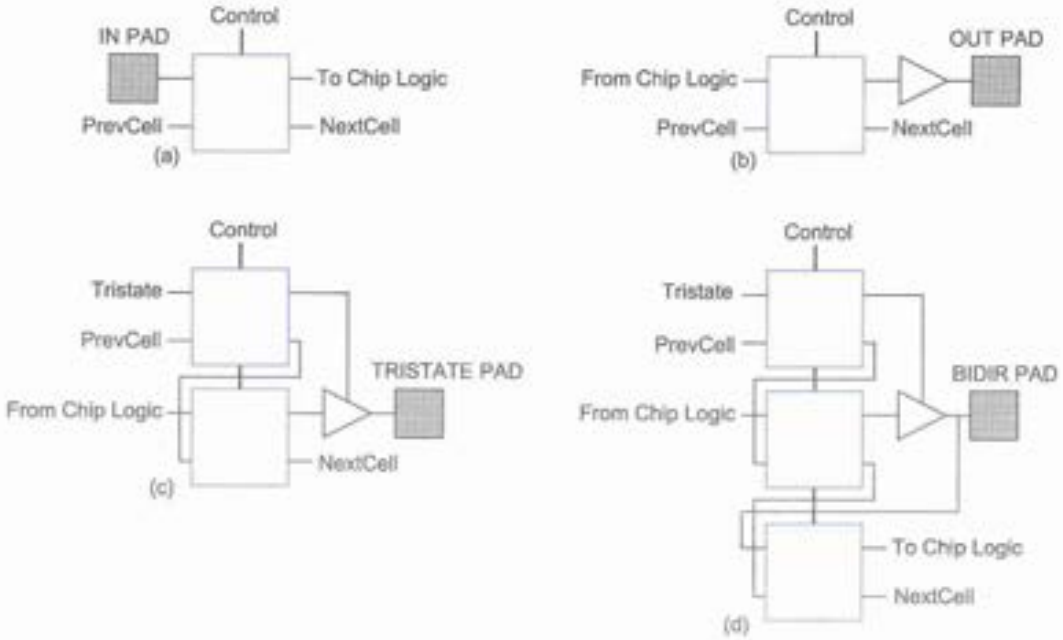


FIG 9.32 Boundary scan pad configuration

The `Mode` signal determines whether `Qout` should be taken from `DataIn` or the boundary scan register. Separate `mode_in` and `mode_out` signals are used for input and output pads so they can be controlled separately. In normal chip operation, both `mode` signals are 0, so the boundary scan registers are ignored. For the `EXTEST` instruction, `mode_out` = 1, so the outputs can be controlled by the boundary scan registers. For `INTEST` or `RUNBIST` instructions, `mode_in` and `mode_out` are both 1, so the core logic receives its inputs from the boundary scan registers and the outputs are also driven to known safe values by the boundary scan registers.

9.7.5.2 Bypass Register When executing the `BYPASS` instruction, the single-bit Bypass register is connected between *TDI* and *TDO*. It consists of a single flip-flop that is cleared during Capture-DR, and then scanned during Shift-DR, as shown in Figure 9.33.

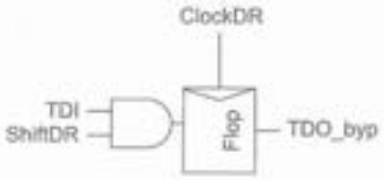


FIG 9.33 Bypass register

9.7.5.3 *TDO* Driver The *TDO* pin shifts out the least significant bit of the IR during Shift-IR or the least significant bit of one of the data registers during Shift-DR, depending on which instruction is active. The IEEE boundary scan specification requires that *TDO* change on the falling edge of *TCK* and be tristated except during the Shift states. This prevents race conditions when the value is clocked into the next chip in the rising edge of

TCK and allows multiple chips to be connected in parallel with their *TDO* pins tied together to reduce the length of the boundary scan chain.

Figure 9.34 shows a possible implementation of the *TDO* driver. The multiplexers choose among the possible shift registers including the instruction register, boundary scan register, and bypass register. Additional multiplexers would be used if more data registers were included. A flip-flop or latch delays the *TDO* signal until the falling edge of *TCK*. The tristate drives *TDO* during Shift-IR or Shift-DR.

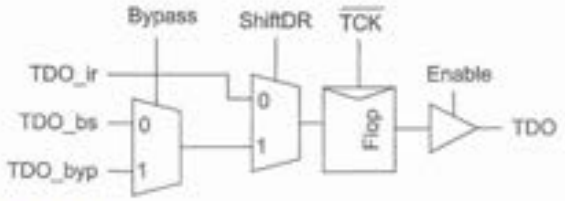


FIG 9.34 TDO driver

9.7.5.4 Complete Test Data Register Logic The Verilog code below describes the complete Test Data Register for a chip with four inputs a[3:0] and four outputs y[3:0]. The four input and four output boundary scan register bits are collected into a single 8-bit shift register. mode_in serves the four most significant bits connected to the inputs, while mode_out serves the four least significant bits connected to the outputs.

```
module data_reg(input  [3:0] a, fromlogic,
                input        tck, tdi, tdo_ir,
                input        ClockDR, UpdateDR, ShiftDR, Enable,
                input        mode_in, mode_out, bypass,
                output [3:0] y, tologic,
                output       tdo);

   reg  [7:0] shiftreg, datareg;
   wire       tdo_selected;
   reg        tdo_byp, tdo_delayed;

   // Boundary scan registers
   // four input registers and four output registers connected in 8-bit chain
   always @(posedge ClockDR)
      shiftreg <= ShiftDR ? {tdi, shiftreg[7:1]} : {a, fromlogic};
   always @(posedge UpdateDR)
      datareg <= shiftreg;
   assign tologic = mode_in ? datareg[7:4] : a;
   assign y = mode_out ? datareg[3:0] : fromlogic;

   // Bypass register
   always @(posedge ClockDR)
      tdo_byp <= tdi & ShiftDR;
```

```
   // tdo output driver
   // select appropriate register to shift out, delay to negative edge of tck
   assign tdo_selected = ShiftDR ? (bypass ? tdo_byp : shiftreg[0]) : tdo_ir;
   always @(negedge tck)
      tdo_delayed <= tdo_selected;
   assign tdo = Enable ? tdo_delayed : 1'bz;
endmodule
```

9.7.6 Summary

Figure 9.35 shows a complete implementation of boundary scan for a chip with four inputs and four outputs. It consists of the TAP controller state machine and state decoder, a 3-bit instruction register with instruction decode, the bypass register, four boundary scan input pads, and four boundary scan output pads. The other pads comprise the test access port. The boundary scan register control signals (UpdateDR, ClockDR, ShiftDR, `mode_in`, and `mode_out`) are shown as the Control bus.

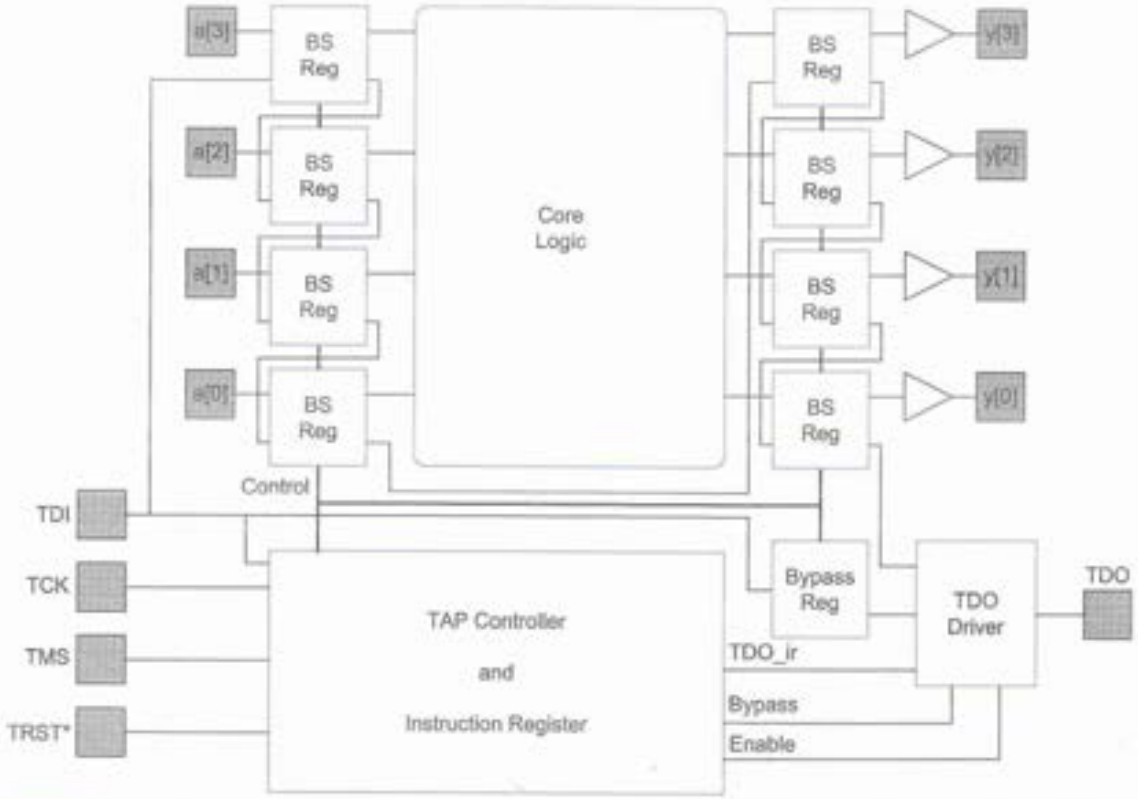


FIG 9.35 Complete boundary scan implementation

The Verilog for this design is shown below.

```
module core(input  [3:0] tologic,
            output [3:0] fromlogic);

   // a silly chip logic function
   assign fromlogic = {&tologic, |tologic, ^tologic, ~tologic[0]};
endmodule

module top(input        tck, tms, tdi, trstn,
           input  [3:0] a,
           output       tdo,
           output [3:0] y);

   wire [3:0] tologic, fromlogic;
   wire       UpdateIR, ShiftIR, ClockIR;
   wire       UpdateDR, ShiftDR, ClockDR;
   wire       Resetn, Enable;
   wire       mode_in, mode_out, bypass;
   wire       tdo_ir;

   // Core Logic
   core core(tologic, fromlogic);

   // TAP Controller
   tapcontroller tc(tms, tck, trstn, ShiftIR, ShiftDR, ClockIR, ClockDR,
                    UpdateIR, UpdateDR, Resetn, Enable);

   // Instruction register
   inst_reg ir(tdi, Resetn, ClockIR, UpdateIR, ShiftIR,
               tdo_ir, mode_in, mode_out, bypass);

   // Test data registers
   data_reg dr(a, fromlogic, tck, tdi, tdo_ir,
              ClockDR, UpdateDR, ShiftDR, Enable, mode_in, mode_out, bypass,
               y, tologic, tdo);
endmodule
```

Boundary scan testing typically begins with the SAMPLE/PRELOAD instruction. Then a data value is preloaded into the boundary scan registers. Next, the EXTEST or INTEST instruction is applied to activate the loaded value. Subsequent data values are shifted into the boundary scan registers and the results of the tests are shifted out.

Figure 9.36 shows waveforms for this operation. The TAP controller is initially reset. At this point, the core logic operates normally with an input pattern of 0000 and an output pattern of 0001. Then the IR is loaded with 101 (SAMPLE/PRELOAD). The data pattern 0111 is shifted in. The IR is loaded with 1000 (INTEST). This sends the 0111 pattern to the core logic, producing an output pattern of 0110. Finally, the data pattern 1111 is shifted in and the old output 0110 is shifted out. Because the INTEST is still active, the 1111 is applied to the core, producing a new output of 1100.

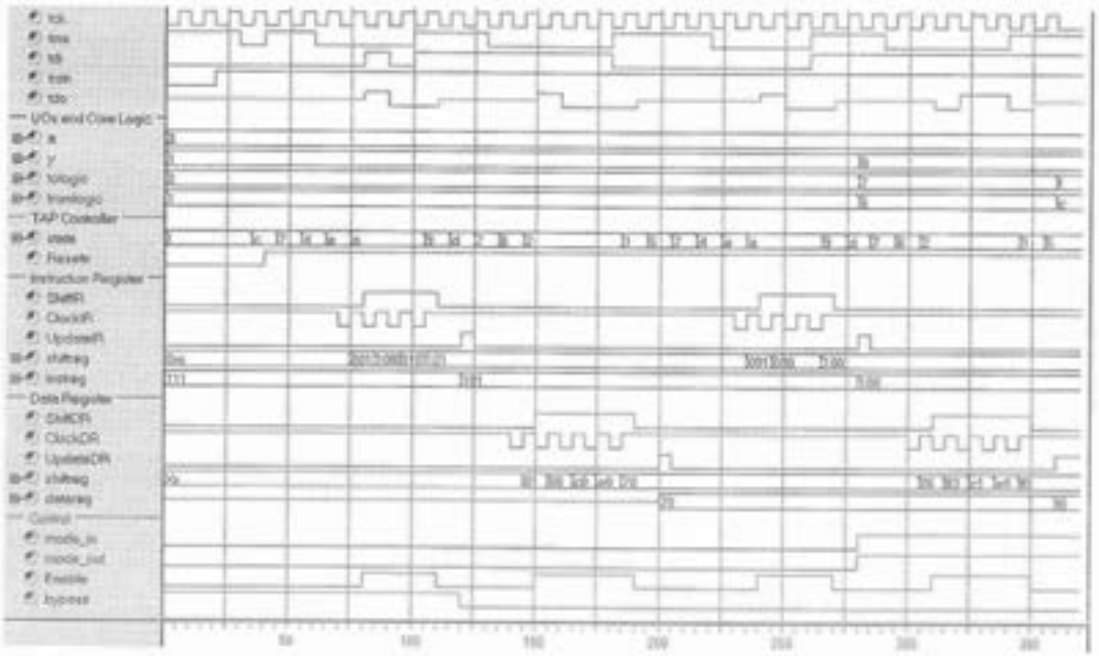


FIG 9.36 Boundary scan example waveforms

Boundary scan is in widespread use in chips today. It provides a uniform interface to single- and multiple-chip testing and circuit-board testing.

9.8 System-on-chip (SOC) Testing

System-on-Chip testing uses the techniques described so far to test the disparate system blocks that might exist on a typical complex chip. The IEEE P1500 Standard for Embedded Core Test (SECT) is being developed to standardize methods for testing SOCs.

Figure 9.37 shows the basic approach that P1500 takes. Each large chip module (i.e., a processor, memory, accelerator) is surrounded with a *wrapper* that resembles a boundary scan chain. The Wrapper Serial Port (*WSP*) has a serial input (*WSI*—Wrapper Serial Input), a serial output (*WSO*—Wrapper Serial Output), and a control port (*WSC*—Wrapper Serial Control). Because serial control and testing can be slow, an optional set of parallel ports is also specified. The Wrapper Parallel Port (*WPP*) consists of the Wrapper Parallel Input (*WPI*), Wrapper Parallel Output (*WPO*), and the Wrapper Parallel Control (*WPC*). The interface to the serial ports is standardized to enable "plug and play" operation. The parallel ports are user-defined to allow test flexibility.

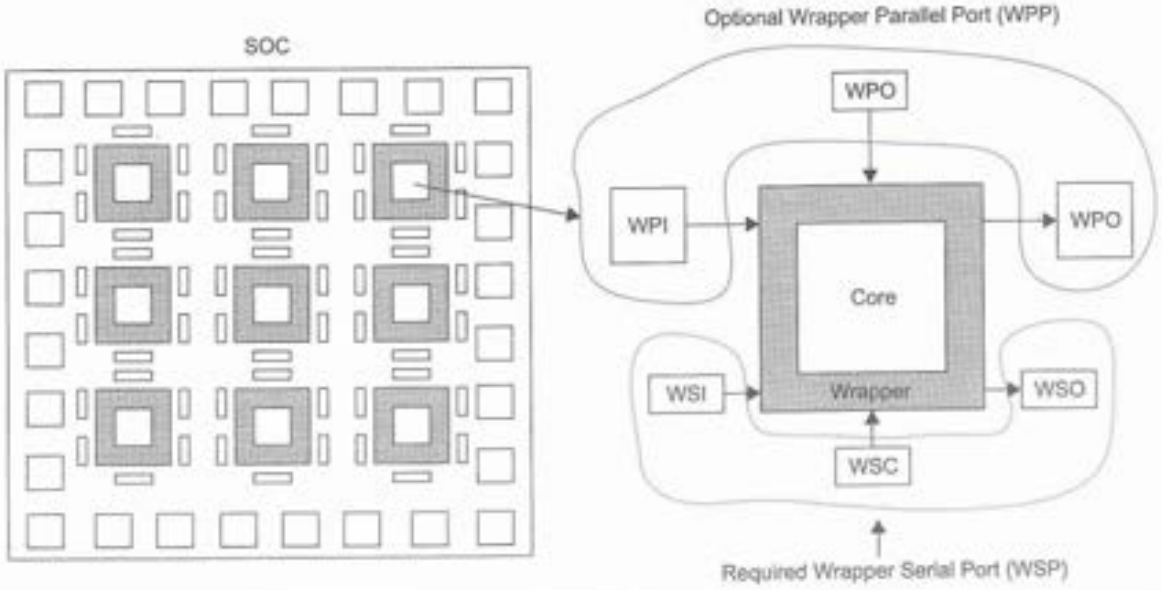


FIG 9.37 IEEE P1500 embedded core test strategy

Embedded cores are interconnected, as shown in Figure 9.38. The *WSP* blocks are daisy-chained in a serial fashion, as shown in the diagram. Global *WSI* and *WSC* signals are provided to control the embedded cores with test data. In addition, a global *WSO* signal is available for observation or daisy-chaining chips together. The serial wrapper ports can optionally control wrapper parallel ports via an enable (*ENA*) signal. The *test access method* (TAM) of the *WPP* can follow one of several architectures.

Some of these parallel TAM methods are shown in Figure 9.39. Figure 9.39(a) shows a daisy-chained method where blocks are interconnected in the same way as the required serial port wrappers. Figure 9.39(b) shows a bus-based TAM. This would allow faster and more direct access to modules. Figure 9.39(c) shows a parallel (direct access) method. This may have the disadvantage that the number of signals or control pins may increase beyond that desired. Finally, Figure 9.39(d) shows autonomous local TAM controllers that are designed to access and test each module. For example, these could be BIST modules that self-test each module and report success or failure over the serial port.

Together, serial scan, BIST, boundary scan, and P1500 form an ordered set of techniques to facilitate testing and verification of large systems on one or more chips.

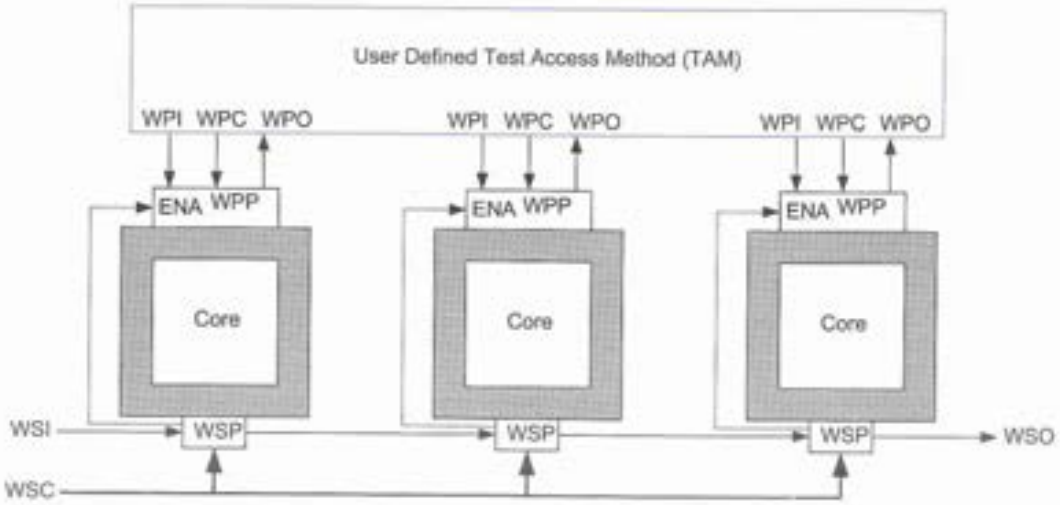


FIG 9.38 P1500 user-defined test access method (TAM)

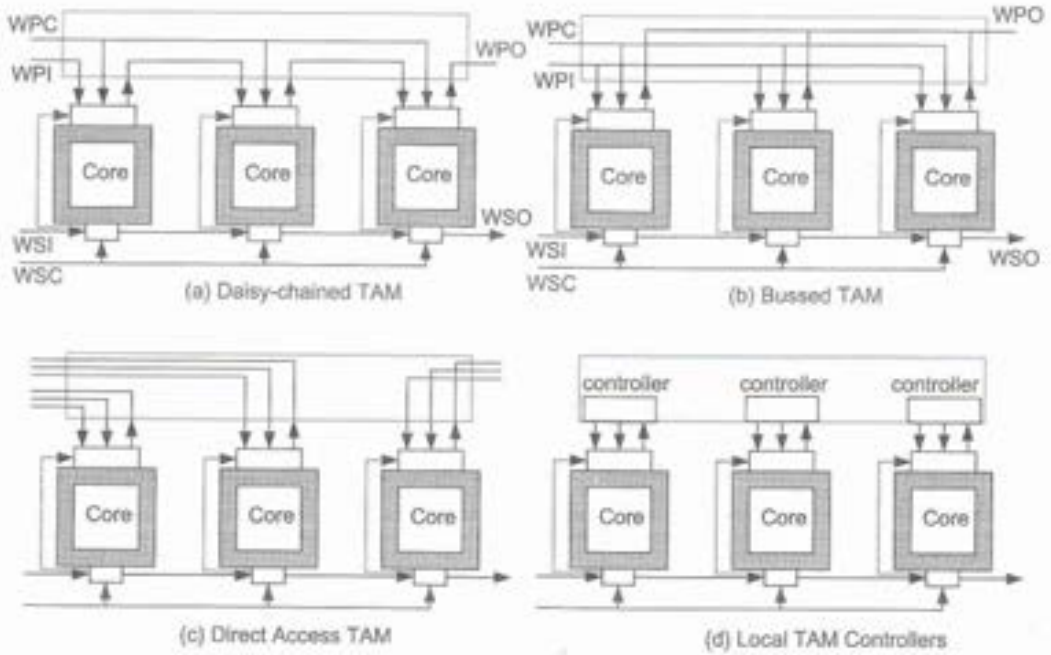


FIG 9.39 Various methods of TAM

9.9 Mixed-signal Testing

Mixed-signal chips contain both analog and digital circuitry. A full exposé on mixed-signal testing is beyond the scope of this book, but we will address a few points. When designing mixed-signal systems, thought should be put into choosing suitable methods of self-test or at least making the analog test engineer's job easier. Many times, analog and RF blocks can be used to help in the testing and verification of digital blocks and vice-versa. Often, the best way to view a dynamically changing situation is via an oscilloscope. On-chip sampling circuits can function as a high-speed equivalent-time oscilloscope to transfer waveforms out over a lower-bandwidth interface [Ho98]. If high-speed digital/analog converters (DACs) are available in the signal path of the chip, then they can be used to aid in system test and debug.

Example

Consider the IQ upconversion path in the software radio example. If multiplexers are placed in the signal path from the NCO to the DAC (Figure 9.40), a variety of signals can be routed to the DACs. The Phase Counter can be used as a ramp generator and fed to the DAC to aid in characterizing of the DAC (a linear ramp is required to measure DAC parameters—see Chapter 12). In addition, any other signal of interest can be routed to the DAC for observation in real time on an oscilloscope. For instance, the receive IQ signals at any point in the receive path can be routed to the DAC and observed on the DAC. This method of test and debug is extremely powerful. If possible, remembering the principle of regularity (and modularity)—making it possible to route any signal register to the DACs—can be invaluable. Circuits that don't even have a DAC can benefit from this technique with the inclusion of a DAC (they can be small and only take one extra pad).

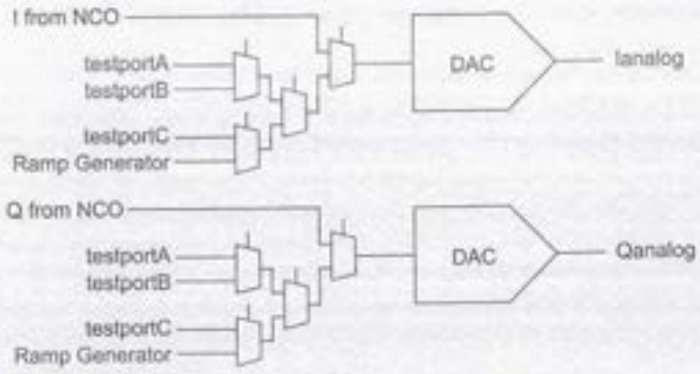


FIG 9.40 DAC testing and system debug

Analog/digital converter testing requires real-time access to the digital output of the ADC. Providing parallel digital test ports by reassigning pins on the chip I/O can facilitate this testing. If this is impossible, a "capture RAM" on chip can be used to capture results in real-time and then the contents can be transferred off-chip at a slower rate for analysis.

If both ADCs and DACs are present, a loopback strategy can be employed, as shown in Figure 9.41. Both analog and digital signals can loop back. Communication and graphics systems frequently have I/O systems that can be configured as shown. It is often worthwhile to add either a DAC or a small ADC to a system to allow a level of analog self-test.

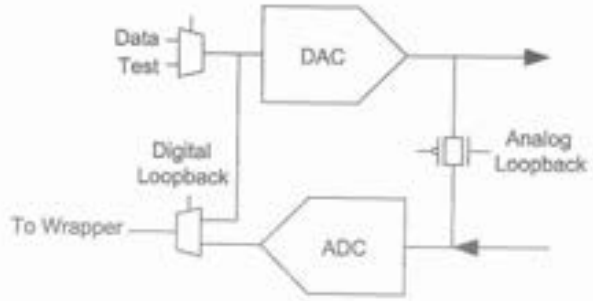


FIG 9.41 Analog and digital loopback

9.10 Reliability Testing

ICs do not have an infinite lifetime. Transistor parameters can drift and metal lines can eventually break after years of metal migration stress. This leads to the desire to assess the reliability (lifetime) of an IC.

As introduced in Chapter 4, reliability is measured in terms of failures in time or FITs. A FIT is a reliability term for one failure in 1 billion hours. Designers aim for failure rates from 10 to 100 FIT. 100 FIT is around 1000 years, so methods have been developed to accelerate reliability testing. Once a part has been proven to have adequate reliability, it is regarded as "qualified."

Tests to establish reliability are divided into electrical, environmental, and mechanical tests and adhere to standards such as the U.S. government's MIL-STD-883. This standard covers a variety of scenarios, including:

- Life testing
- Environmental testing, including vibration, temperature cycling, and shock and moisture testing
- Radiation testing

- Electrical tests, including noise margin, power consumption, and analog tests
- Package tests, including mechanical tests on wire bond strengths and ultrasonic die inspection

The reader is referred to the standard for a complete listing of all tests.

A part has to be *qualified* prior to shipping in production. Qualification can occur at various levels. A process has to be qualified before companies will use it for product. The qualification process guarantees that devices will not fail if they are exposed to normal voltages over a reasonable lifetime. Possible failure modes can include threshold voltage shifts and open circuits from electromigration. A particular standard cell logic and I/O library can be qualified. In particular, this means that the IP can stand up to prescribed conditions for the lifetime of the IC. Finally, the individual design has to be qualified. This involves assessing the reliability of the part.

One of the methods of achieving a reliability metric is to use accelerated life testing (ALT). This process runs a set of chips at elevated temperature and voltage for an extended period. At periodic intervals in this period the chips are re-tested (or tested *in situ*) and any failures logged. The time of any failures leads to a reliability number at that voltage and temperature. Lifetime typically decreases exponentially with voltage and temperature, so measurements made at a few elevated levels over modest periods of time are extrapolated back to predict failure rates at normal levels. ALT can be dynamic or static. Static ALT just supplies the chip with supply voltages and perhaps a clock. Dynamic ALT operates the chip as it would in a system. ALT uses what is termed a *burn-in board*. This has a number (usually around 40) of chips running at temperatures of 150°–200° C. The chips are periodically checked over a period of 4000 hours. Failures can be mapped to an MTBF number.

A burn-in test can be applied to each chip to ensure higher than normal reliability (i.e., for space-borne applications). This operates the chip at elevated temperatures for a particular period. The process is designed to cull out infant mortality failures (see Section 4.8.1). Normally, high-volume chips do not undergo burn-in because of time and cost pressures.

9.11 Testing in a University Environment

Industry environments are usually well-funded, and the appropriate testability tools are available to ensure a product-grade test effort. But what do you do in a university environment when the infrastructure might not be quite as affluent as in the industry setting? Not only may test tools be unavailable, but also the very act of building a test board can be a daunting extra amount of work on top of the chip design. The following are some tips that might help in this situation.

Taking the time to include circuitry to aid in testing on the chip is usually much easier than adding it at the board level. For a start, the integrated environment available for most IC design flows allows the designer to simulate the test circuitry. So, while it might seem

superfluous to the task at hand, including test circuitry can save a huge amount of effort after the chip returns. Moreover, on-chip circuitry can often test at speeds that are impossible off-chip without extremely expensive production test machines. The main point is to think ahead.

BIST is straightforward to integrate into a library of registers or I/O pads. In the case of data pipelines such as the NCO and CORDIC used as examples in this chapter and the previous one, we have shown how to add testing capability to the registers. The University of Tennessee has developed a "SmartFrame" pad frame compatible with the MOSIS AMI 0.6 mm process that incorporates a PRSG and signature analyzer into the pads [Bouldin03]. Alternatively, if a boundary scan interface that supports INTEST is incorporated onto the chip, the chip can be tested from a PC using a commercial boundary scan controller. For example, the Corelis NetUSB-1149.1/E can drive the scan chains at up to 80 MHz.

In the absence of BIST, there are several ways to test a chip. One is to breadboard or wirewrap a test board with switches for inputs and LEDs for outputs. This is tedious for all but the simplest chips. A custom-printed circuit board test fixture is even more labor-intensive, but often necessary for high-performance research chips. Another strategy is to use a logic analyzer with pattern generator. This approach requires a specialized test fixture to hold the chip and often has a steep learning curve for students, but it can perform tests at 10's to 100's of MHz. Yet another approach is to use a low-cost *functional chip tester* that reads the test vectors from pretapeout simulation, applies the digital patterns, and checks the results at low speed. Figure 9.42 shows the TestosterICs functional chip tester from One Hot Logic [Harris03] used by the author to reduce the time students spend testing their chips. The DUT boards contain zero insertion force (ZIF) sockets for 40-pin DIP and 16x16 PGA packages commonly used in class projects.

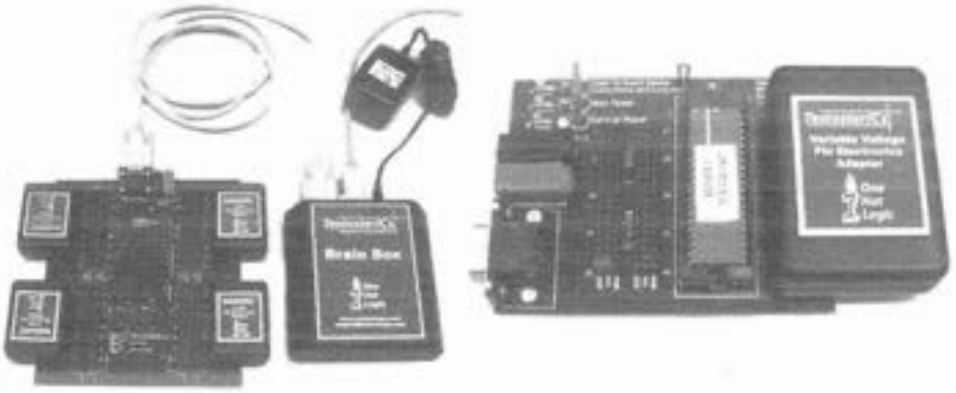


FIG 9.42 TestosterICs functional chip tester

9.12 Pitfalls and Fallacies

The following "war stories" are collected from real products at a wide variety of companies and published with permission, often under the condition of anonymity. They are presented to illustrate some of the pitfalls that can happen to smart people who are dealing with complex systems on a tight schedule. The skilled engineer learns from these mistakes; in most cases, the company extended their verification flow to ensure that similar problems would be caught before wreaking havoc on future products. Could one of these happen to you?

A Product in the Field Hangs Unpredictably

A microprocessor had been in the field for several years when reports began arriving from major customers that certain programs would cause the system to hang at unpredictable times with intervals of hours to days. The manufacturer appointed a tiger team to resolve the error. The hang rate proved to be insensitive to power supply voltage, operating temperature, and clock rate. It was observed on all versions of the chip regardless of foundry, manufacturing technology, or motherboard. The programs that failed all involved a mix of floating point and integer operations; none were purely integer codes.

After several months of work, the problem was isolated to a particular unit in the processor. By this point, 30 engineers were involved in chasing the problem. Picoprobing showed that when the hang occurred, an instruction was left stuck in the pipeline waiting to issue. A logic simulation of the RTL is orders of magnitude slower than running the actual code, but an engineer developed a simple test case that could trigger the hang on real hardware in a matter of seconds, and thus it could trigger the failure in simulation in a practical amount of time. Simulations showed that the RTL ran flawlessly, suggesting the error involved a circuit that did not match the RTL.

On this processor, the circuits had been verified against the RTL using a technique called "shadow-mode simulation." A "circuit understanding" tool parsed the transistor-level netlist into gates and identified the logic function of each gate. Circuits were verified to match the RTL by replacing a module of the RTL with the corresponding extracted circuit and simulating to check that the system produced identical results as the original RTL. The simulation is time-consuming, so each module is typically checked over tens of thousands of cycles, rather than the billions of cycles used in primary RTL verification.

A shadow-mode simulation using circuits from the failing unit still ran flawlessly. However, an engineer observed that a long wire crossing a large schematic was driven from both ends to reduce the RC delay. The signals X1 and X2 driving each end were intended to be identical (Figure 9.43). The engineer experimented with splitting the wire and checking that both drivers produced identical results, and on certain test cases they did not. This led to the wire experiencing contention and being driven to an indeterminate logic value. The invalid result propagated through other logic and hung the processor. Unfortunately, the circuit-understanding tool had incorrectly determined that the logic for the two ends was identical and had never detected the error. Even if the tool had been correct, the original test cases never would have exercised the patterns that caused the drivers to produce different results. A simple modification to the driver fixed the problem, but many units were already in the field. Fortunately, a software patch was developed to prevent the operations that caused the hang from ever being issued.

Hanging is a serious problem, but not as severe as unknowingly calculating the wrong answer. After the problem was corrected, engineers spent several more weeks proving to customers that the failure mode

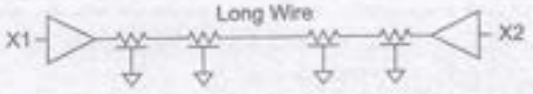


FIG 9.43 Long wire driven from both ends

would hang the machine but could never result in an incorrect calculation.

To avoid repeating this problem in the future, engineers have turned to formal verification tools that prove that RTL and schematics are equivalent in their Boolean function. Such tools are not susceptible to incomplete test patterns. However, the tools are often expensive, proprietary, and difficult to use.

A Product Fails after the Manufacturing Process Matures

A team designing a data communications product was comfortable with a particular microprocessor that was at the end of its production run. The team negotiated to order several thousand units of the discontinued microprocessor before production was shut down. The data communications product became successful and was shipped in large quantity. After it had been in the field for some time, major customers reported that the product would crash in large networks. These customers included large financial, government, and Internet service provider organizations who were adversely affected by the crashes. It took the data communications company weeks to isolate the problem to hanging of the microprocessor, and then a team of engineers at the microprocessor company began investigating the issue.

The microprocessor team investigated potential signal and power supply integrity issues. Although no signal integrity problems were apparent, a shmoo plot showed unusual sensitivity of minimum clock period to supply voltage. An engineer had recently read the application note for the power regulator on the system board and had learned that it had a propensity for oscillation if not properly bypassed. The system board lacked the bypass capacitors recommended in the application note, so the engineer wrote a memo to the product manager suggesting a change to the board. The memo was misinterpreted as a solution to the problem and customers were informed that a fix was on its way. Unfortunately, further testing showed that bypassing the regulator did not fix the crashes.

When the system crashed, it wrote its state to a core file. An engineer began reading a hexadecimal dump of the file and noticed a pattern that led to solving the crash. The pattern was associated with simultaneous access to many banks in an 8-way associative instruction cache. The cache had fuses associated with each bank, so banks containing bad blocks could be disabled during manufacturing test. During original product debug, the manufacturing process was relatively immature and most processors only had five operational cache banks. However, the processors manufactured at the end of the production run were built on a more mature process and often had all eight banks functional. Simultaneous access to all the banks tickled a signal integrity problem, resulting in power supply droop from excessive *IR* drops caused by poor contacts to the V_{DD} plane. The solution was a software change to disable three of the banks at system startup.

Better power supply analysis is performed to avoid repeating this problem.

A Wasted Spin

A microprocessor was taped out and came back nearly fully operational. Minor changes were made to the layout and documentation was developed; then a second revision (colloquially called a second spin of the chip) was taped out. The second revision came back completely nonfunctional, with a short between power and ground. Optical inspection while manufacturing the polysilicon layer showed that there was no field oxide on the chip.

Inspection of the masks showed that the active area mask specified active area (i.e., diffusion) for the entire chip rather than just where transistors belonged. The layout tool assigned each layer—such as active area or metal1—a unique number. However, although the layout for active area layer was correct, the mask did not appear to match the active layer.

Layout documentation had been annotated on an unused layer by drawing rectangles and text to indicate functional blocks. A larger rectangle defined the entire chip area. Careful tracing of the mask-generation soft-

ware found that the "unused" layer had been used for active area many years ago and that the documentation rectangles were merged with the true active area to form a blob of active covering the entire chip.

Another microprocessor from a different vendor also failed when it was first built. Visual inspection of the die showed that the entire cache was missing. The cache had been removed from the design database to speed up final verification because it had already been checked separately. An engineer neglected to put it back in before tapeout.

Both of these wasted fabrication runs could have been avoided by using more rigorous verification methods at both the design and mask fabrication facilities. Validation of dataset size by the designer would have caught the missing geometries. Use of the industry standard mask database inspection tools would have caught the error after mask build. Although in the past, fabrication of a modest number of parts for testing was a small part of the design cost, with the escalation of mask and wafer fabrication costs, these mistakes can be a multi-million-dollar error. The extra time to market has a large opportunity cost as well.

At High Voltage, a Chip Only Operates at Low Frequency

While booting the operating system during silicon debug, a microprocessor operated as expected at low voltage. At high voltage, the part only functioned at low frequency. The high-voltage roof is an indication of a potential coupling problem in which the coupling is exacerbated by the fast edge rates associated with high-voltage operation. Test cases revealed that the problem resulted from incorrect operation of the register file when certain instructions executed. When the designers inspected the scan latches, they found that the correct 0 value was sent to the register file to write, but that an incorrect 1 was read. This indicated that either read or write operation was failing at high voltage. Trying one operation at high voltage and the other at low voltage proved the problem was in the write path.

A schematic of the register file write circuitry is shown in Figure 9.44. The register file uses predischarged write bitlines that are conditionally pulled high, depending on the data. The appropriate cell is written by turning on the corresponding write access transistor. The register cell is intentionally unstable so that the value on the bitline can overpower the cell and write the appropriate value. A weak keeper holds the metal2 bitline low when writing a 0. However, the register file is large and the keeper is at the opposite end from the data transistor. The resistance of the long thin wire further reduces the effectiveness of the keeper against noise on the bitline.

When the neighboring bitlines switch high, they couple onto the victim line and tend to pull it high. The circuit fails if the aggressors introduce too much coupling noise. At high voltage, the aggressor drivers are stronger

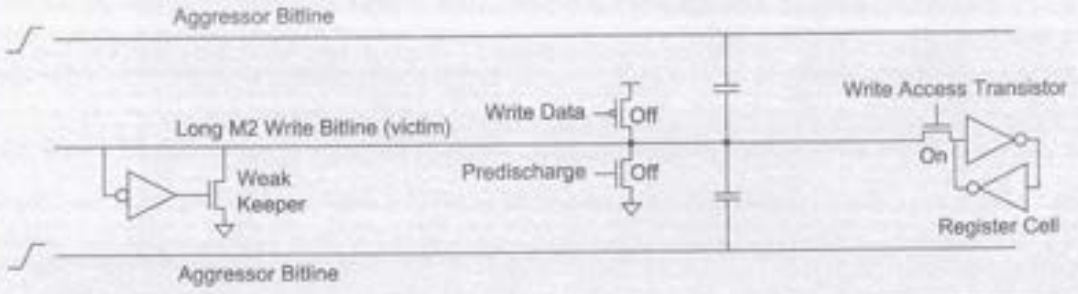


FIG 9.44 Register file write circuitry

and cause a momentary glitch on the victim. At low frequency, the keeper is sufficient to restore the victim to a low level.

The coupling problem had been flagged during design by an automated noise-checking tool. However, the tool is conservative and the area of the register file would have increased significantly if the bitlines were spaced far enough apart to satisfy the tool. Therefore, the designer checked for excessive coupling with a SPICE simulation. The simulation apparently did not properly model the combination of circumstances that caused the failure. A second engineer cross-checked all circuits that waived the noise-checker warning, but also did not discover the excessive coupling. The problem was solved by placing a second keeper near the write data transistor to fight against the coupling.

Another Funny Shmoo

During silicon debug, a microprocessor cache only functioned correctly over the peculiar range of voltages and frequencies shown in the shmoo in Figure 9.45[1]. Test code exercising the cache revealed that failures were

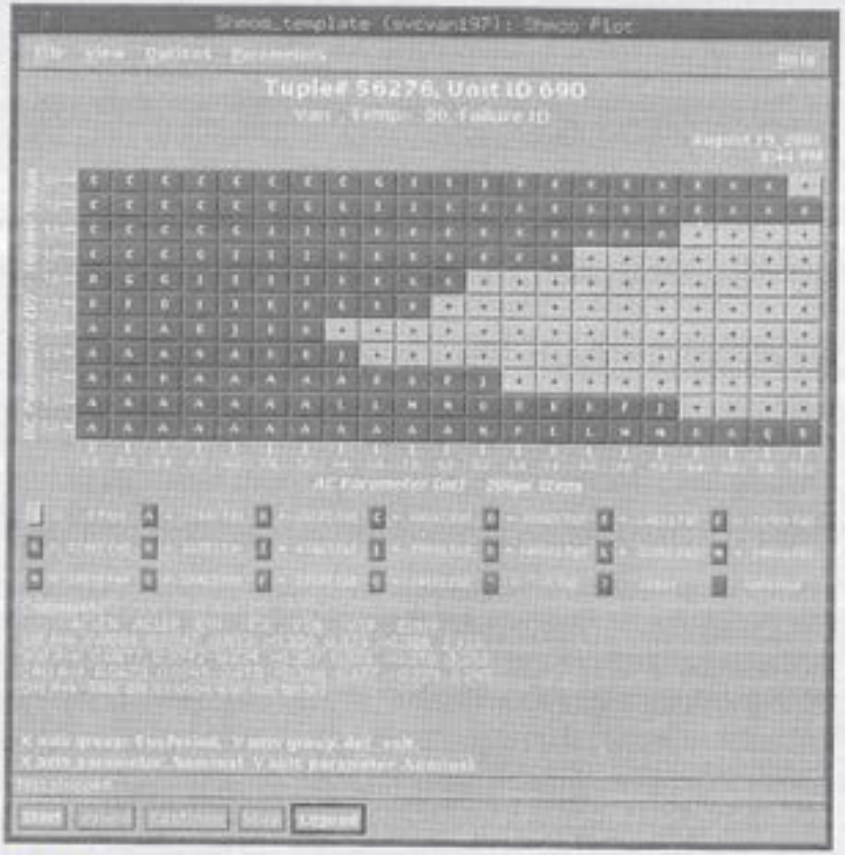


FIG 9.45 Flying saucer shmoo

[1]A shmoo of this type is sometimes called a *flying saucer*.

caused by bad data being read from the cache. Scan isolated the problem to a dynamic multiplexer choosing one of the global bitlines, as shown in Figure 9.46.

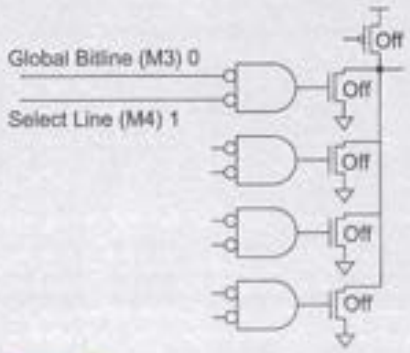


FIG 9.46 Dynamic bitline multiplexer

The multiplexer inputs were the NORs of dynamic metal3 global bitlines and corresponding select signals. The metal4 select lines were early and did not need to be dynamic, but were implemented as dynamic nodes anyway. All of the transistors in the dynamic multiplexer were supposed to remain OFF in this particular test case, leaving the multiplexer output high.

One input of the multiplexer had a low value on the global bitline, but was not selected, as shown. Therefore, the transistor should have been OFF. Nevertheless, the output of the multiplexer incorrectly discharged. One neighbor of the select line was ground; the other fell low. Coupling from a single neighbor is generally not enough to cause noise failure. However, many global bitlines ran over the top of the select line and also fell low. Laser voltage probing showed that the select line was incorrectly pulled low, apparently from coupling caused by these falling bitlines as well as the neighbor line. The odd shape of the shmoo happened because the failures only occurred when the neighbor and overhead lines both fell at about the same time; otherwise, the keeper on the select line was strong enough to recover from one noise event before the other arrived. Because the bitline and control paths were different, the noise events only happened simultaneously for certain voltages.

Noise analysis tools usually check only neighbors, and the single switching neighbor was not sufficient to trigger an error. In this circumstance, so many global bitlines ran over the top of the select wire that their coupling could not be neglected. The problem was fixed by converting the control line into a static signal more resistant to coupling noise. A better noise analyzer could have considered coupling from neighbors above and below, especially on dynamic nets. However, it is difficult to extract information about such orthogonal neighbors because they are often drawn at different levels of the layout hierarchy. Moreover, assuming all neighbors switch in the worst possible direction is usually pessimistic for long wires. Nevertheless, such a data-dependent failure mechanism is a source of nightmares for designers.

Incorrect Operation at Low Temperature

A floating-point coprocessor was tested by running the LINPACK benchmark. The benchmark performs a series of floating-point operations and generates a checksum to verify the result. The chip would occasionally produce the wrong checksum. One of the engineers heated the coprocessor by removing the heatsink and found that the coprocessor became reliable at higher temperature.

This suggested that the problem might be caused by coupling, which is generally more serious at lower temperature where the edge rates are faster. The error was tracked to a long on-chip bus with many wires laid out on a tight pitch. Although the wires were subject to coupling noise, they were not on a critical path and should have had plenty of time to settle to the correct value. Unfortunately, they drove the diffusion input of a latch. When crosstalk drove an input below $-V_t$, it would turn on the pass transistor and incorrectly discharge the latch (see Section 6.3.9).

The floating-point unit bug was holding up lucrative product shipments. While a corrected coprocessor was being fabricated, the old unit was shipped in products

with a bolt-on thermostat/heater unit used to guarantee a minimum operating temperature.

An obvious lesson of this experience is to avoid driving diffusion inputs with potentially noisy signals. More fundamentally, however, this bug demonstrated a marginal design of the cell library that should have been caught in the library review. Moreover, humans are inherently prone to errors. Electrical rules like no noisy diffusion inputs aren't worth the paper they are printed on unless computer code exists to enforce them.

Slower Than Expected Performance

An application-specific integrated circuit (ASIC) was fabricated on a gate array by a third-party gate array manufacturer. Although static timing analysis predicted that the chip would function fast enough, the manufacturer found that most of the chips would not operate at the desired frequency and instead had to be derated by about 20%.

The designer examined a die plot, looking for the source of the unexpectedly slow performance. The plot showed that the horizontal power and ground lines were only strapped along the edges of the chip, as shown in Figure 9.47(a). Some rows of gates consumed large amounts of power, causing large *IR* drops along their power lines. Measurements showed that the power supply sometimes drooped below 2 V, despite the nominal 3.3 V power supply. When the wide vertical power supply straps were added, as shown in Figure 9.47(b), most chips met target speed.

Modern chips require low-resistance on-chip power distribution networks and often use power and ground pads distributed across the die rather than just at the periphery to reduce the distance and resistance between the pads and the gates. Power integrity analysis should be performed to verify that the static or dynamic voltage droops remain within their budget everywhere on the chip.

Class Chip Failures

One of the authors has supervised a number of class project chips. Some of the reasons that chips have come back partially or completely nonfunctional include:

- *Insufficient simulation*
  A ring oscillator was placed on the chip as a test structure to verify that the hardware was at least partially functional even if the rest of the chip might not work. It didn't oscillate. It had not been simulated because it was "too simple." Inspection during debug found that the oscillator had an even number of inverters!

  Another chip was designed with a new CAD tool that had a buggy simulator. Most of the chip operated correctly, but the chip as a whole would not simulate. The problem was attrib-

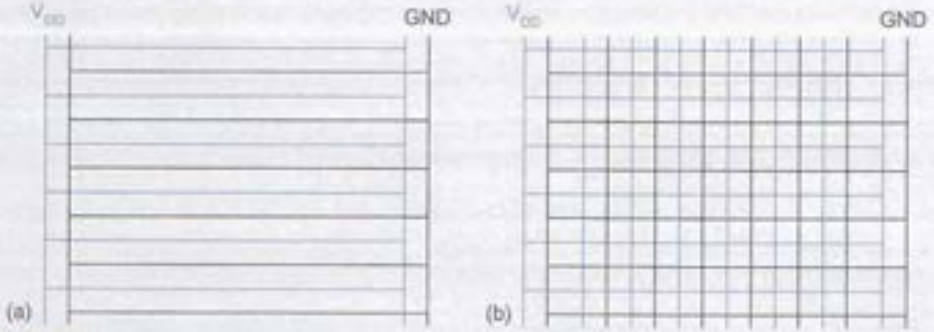


FIG 9.47 Power supply network

uted to a bug in the simulator and was taped out anyway. The chip came back nonfunctional.

- *Incomplete top-level verification*
  One year, a pad frame was used that was incompatible with the normal verification flow. The chip cores were verified, placed in the pad frame, and then routed to the pads. DRC and simulation were not performed on the connections to the pads, so students carefully scrutinized their routing by hand. Upon testing, three of the four different designs were found to have errors in the routing to the pads. No errors were found in the cores that had been verified. "If you don't test it, it won't work! (guaranteed)"
  - A neural network chip seemed to have a defective scan chain because the scan data out line never budged from 0 as configuration data was scanned into the chip. Testing found that the chip was correctly configured except in the last bit of the scan chain. Inspection of the layout revealed that the scan data out line (which came from the last bit of the scan chain) had been shorted to ground while being routed to the pads.
  - A carry-lookahead adder produced incorrect results on certain input patterns. The least significant bits were always correct. Inspection of the layout revealed that the A[4] input was routed from the pad most of the way to the core but part of the wire was missing, probably because the designer accidentally hit UNDO after finishing the route.
  - A GPS searcher chip had an inverter connected to a pair of pins to verify that the chip showed basic functionality. The output was stuck low. Inspection of the layout revealed that the input was attached to an output pad and the output to an input pad. The GPS searcher itself was fully operational.

While some of these may represent class situations, the same type of reasons for partial failure also plague industry chips. In particular, when time scales are stressed, the boundary conditions are often overlooked, which leads to problems when the chips are fabricated. Once a good verification methodology is put in place that includes a known-good pad frame, top-level DRC, and full-chip simulation, students have had a 100% success rate on class chips.

Summary

This chapter has summarized the important issues in CMOS chip testing and has provided some methods for incorporating test considerations into chips from the start of the design. Scan is now an indispensable technique to observe and control registers because probing signals directly has become extremely difficult. The importance of writing adequate tests for both the functional verification and manufacturing verification cannot be understated. It is probably the single most important activity in any CMOS chip design cycle and usually takes the longest time no matter what design methodology is used. If one message is left in your mind after this chapter, it should be that you should be absolutely rigorous about the testing activity surrounding a chip project and it should rank first in any design trade-offs.

Exercises

9.1 A circuit does not operate at the desired frequency. Cooling the circuit with freeze spray fixes the problem. A shmoo shows the circuit operates correctly at higher than nominal V_{DD}. What is the general nature of the likely problem and why?

9.2 You have to test a large die (1 cm × 1 cm) that is housed in a package that costs $5. Would you do wafer testing? Why?

9.3 A verification script detects a single discrepancy between the golden model and your design out of 400,000 vectors. Would you proceed to fabrication? Explain your decision.

9.4 Explain what is meant by a Stuck-at-1 fault and a Stuck-at-0 fault.

9.5 How are sequential faults caused in CMOS? Give an example.

9.6 Explain the different kinds of physical faults that can occur on a CMOS chip and relate them to typical circuit failures.

9.7 Explain the terms controllability, observability, and fault coverage.

9.8 Why is it important to have a high fault coverage for a set of test vectors?

9.9 Explain how serial-scan testing is implemented.

9.10 Explain the principles of Built-In Self-Test (BIST). What are the advantages and disadvantages of BIST?

9.11 You have to design an extremely fast divide by eight frequency divider that taxes the capabilities of the process you are using. What test strategy would you employ to test the divider? Explain the reasons for your choice.

9.12 Design a register that minimizes transistor count, but allows parallel scan to be implemented, as outlined in Figure 9.14.

9.13 Explain how a Pseudo-Random Sequence Generator (PRSG) can be used to test a 16-bit data path. How would the outputs be collected and checked?

9.14 Design a block diagram of a test generator for a 4K × 32 static RAM.

9.15 Using P1500, describe in block diagram form how you would perform SOC testing of the software radio shown in Figure 8.11. What architecture of TAM would you use and why?

Datapath Subsystems 10

10.1 Introduction

Most chip functions can be divided into the following categories:

- Datapath operators
- Memory elements
- Control structures
- Special-purpose cells
  - I/O
  - Power distribution
  - Clock generation and distribution
  - Analog

CMOS system design consists of partitioning the system into subsystems of the types listed above. Many options exist that make tradeoffs between speed, density, programmability, ease of design, and other variables. This chapter addresses design options for common datapath operators. The next chapter addresses arrays, especially those used for memory. Control structures are most commonly coded in a hardware description language and synthesized. Special-purpose subsystems are considered in Chapter 12.

As introduced in Chapter 1, datapath operators benefit from the structured design principles of hierarchy, regularity, modularity, and locality. They may use N identical circuits to process N-bit data. Related data operators are placed physically adjacent to each other to reduce wire length and delay. Generally, data is arranged to flow in one direction, while control signals are introduced in a direction orthogonal to the dataflow.

Common datapath operators considered in this chapter include adders, one/zero detectors, comparators, counters, Boolean logic units, error-correcting code blocks, shifters, and multipliers.

10.2 Addition/Subtraction

"Multitudes of contrivances were designed, and almost endless drawings made, for the purpose of economizing the time and simplifying the mechanism of carriage."

—Charles Babbage, on Difference Engine No. 1, 1864 [Morrison61]

Addition forms the basis for many processing operations, from counting to multiplication to filtering. As a result, adder circuits that add two binary numbers are of great interest to digital system designers. An extensive, almost endless, assortment of adder architectures serve different speed/area requirements. This section begins with half adders and full adders for single-bit addition. It then considers a plethora of carry-propagate adders (CPAs) for the addition of multi-bit words. Finally, related structures such as subtracters and multiple-input adders are discussed.

10.2.1 Single-bit Addition

The *half adder* of Figure 10.1(a) adds two inputs, A and B. The result is 0, 1, or 2, so two bits are required to represent the value; they are called the sum S and carry-out C_{out}. The carry-out is equivalent to a carry-in to the next more significant column of a multi-bit adder, so it can be described as having double the *weight* of the other bits. If multiple adders are to be cascaded, each must be able to receive the carry-in. Such a *full adder* shown in Figure 10.1(b) has a third input called C or C_{in}.

FIG 10.1 Half and full adders

The truth tables for the half adder and full adder are given in Table 10.1 and Table 10.2. For a full adder, it is sometimes useful to define *Generate* (G), *Propagate* (P) and perhaps *Kill* (K) signals. The adder generates a carry when C_{out} is true independent of C_{in}, so $G = A \cdot B$. The adder kills a carry when C_{out} is false independent of C_{in}, so $K = \overline{A} \cdot \overline{B} = \overline{A + B}$. The adder propagates a carry, i.e., produces a carry-out if and only if it receives a carry-in, when exactly one input is true: $P = A \oplus B$.

Table 10.1 Truth table for half adder

A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Table 10.2 Truth table for full adder

A	B	C	G	P	K	C_{out}	S
0	0	0	0	0	1	0	0
		1				0	1
0	1	0	0	1	0	0	1
		1				1	0
1	0	0	0	1	0	0	1
		1				1	0
1	1	0	1	0	0	1	0
		1				1	1

From the truth table, the half adder logic is:

$$\begin{aligned} S &= A \oplus B \\ C_{out} &= A \cdot B \end{aligned} \tag{10.1}$$

and the full adder logic is:

$$\begin{aligned} S &= A\overline{B}\,\overline{C} + \overline{A}B\overline{C} + \overline{A}\,\overline{B}C + ABC \\ &= (A \oplus B) \oplus C = P \oplus C \\ C_{out} &= AB + AC + BC \\ &= AB + C(A+B) \\ &= \overline{\overline{AB} + \overline{C}(\overline{A} + \overline{B})} \\ &= \mathrm{MAJ}(A,\ B,\ C) \end{aligned} \tag{10.2}$$

The most straightforward approach to designing an adder is with logic gates. Figure 10.2 shows a half adder. Figure 10.3 shows a full adder at the gate (a) and transistor (b) levels. The carry gate uses the factored definition and is also called a *majority* gate. Full adders are used most often, so they will receive the attention of the remainder of this section.

FIG 10.2 Half adder design

The full adder of Figure 10.3(b) employs 32 transistors (6 for the inverters, 10 for the majority gate, and 16 for the 3-input XOR). A more compact design is based on the observation that S can be factored to reuse the C_{out} term.

$$S = ABC + (A + B + C)\overline{C}_{out} \tag{10.3}$$

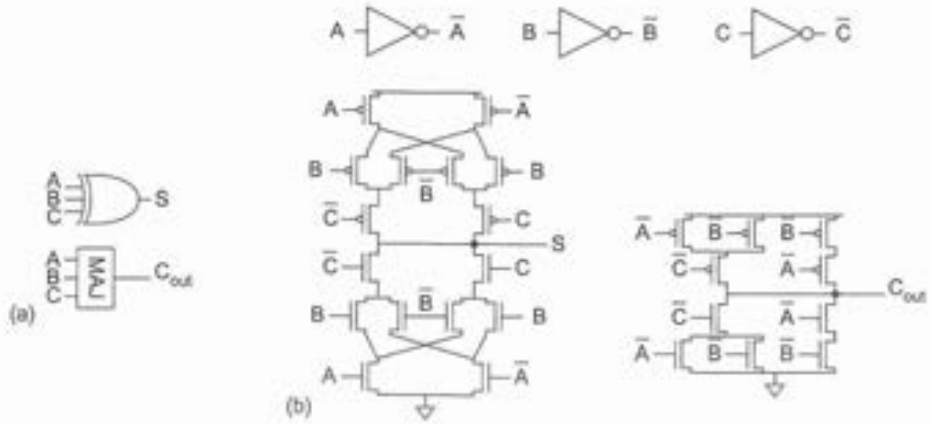


FIG 10.3 Full adder design

Such a design is shown at the gate (a) and transistor (b) levels in Figure 10.4 and uses only 28 transistors. Note that the pMOS network is identical to the nMOS network rather than being the conduction complement. This simplification reduces the number of series transistors and makes the layout more uniform. It is possible because the addition function is *symmetric*, i.e., the function of complemented inputs is the complement of the function.

This design has a greater delay to compute S than C_{out}. In carry-ripple adders (Section 10.2.2.1), the critical path goes from C (C_{in}) to C_{out} through many full adders, so the extra delay computing S is unimportant. Figure 10.4(c) shows the adder with transistor sizes optimized to favor the critical path using a number of techniques:

- Feed the carry-in signal (C) to the inner inputs so the internal capacitance is already discharged.
- Make all transistors in the sum logic whose gate signals are connected to the carry-in and carry logic minimum size (1 unit, e.g., 4 λ / 2 λ). This minimizes the branching effort on the critical path. Keep routing on this signal as short as possible to reduce interconnect capacitance.
- Determine widths of series transistors by logical effort and simulation. Build an asymmetric gate that reduces the logical effort from C to $\overline{C}_{out}$ at the expense of effort to S.
- Use relatively large transistors on the critical path so that stray wiring capacitance is a small fraction of the overall capacitance.
- Remove the output inverters and alternate positive and negative logic to reduce delay and transistor count to 24 (see Section 10.2.2.1).

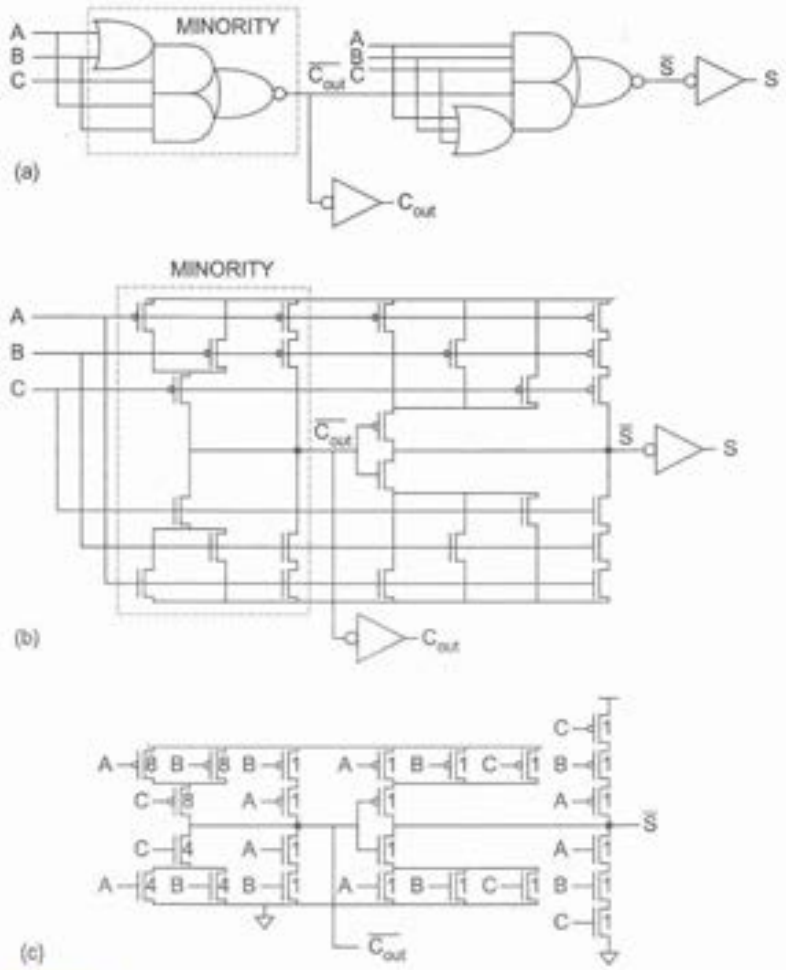


FIG 10.4 Full adder for carry-ripple operation

Figure 10.5 shows two layouts of the adder (see also the inside front cover). The choice of the aspect ratio depends on the application. In a standard-cell environment, the layout of Figure 10.5(a) might be appropriate when a single row of nMOS and pMOS transistors is used. The routing for the *A*, *B*, and *C* inputs is shown inside the cell, although it could be placed outside the cell because external routing tracks have to be assigned to these signals anyway. Figure 10.5(b) shows a layout of Figure 10.4(c) that might be appropriate for a datapath. Here, the transistors are rotated and all of the wiring is completed in polysilicon and metal1. This allows metal2 bus lines to pass over the cell horizontally. Moreover, the widths of the transistors can increase without impacting the bit-pitch (height) of the datapath. In this case, the widths are selected to reduce the C_{in} to C_{out} delay that is on the critical path of a carry-ripple adder.

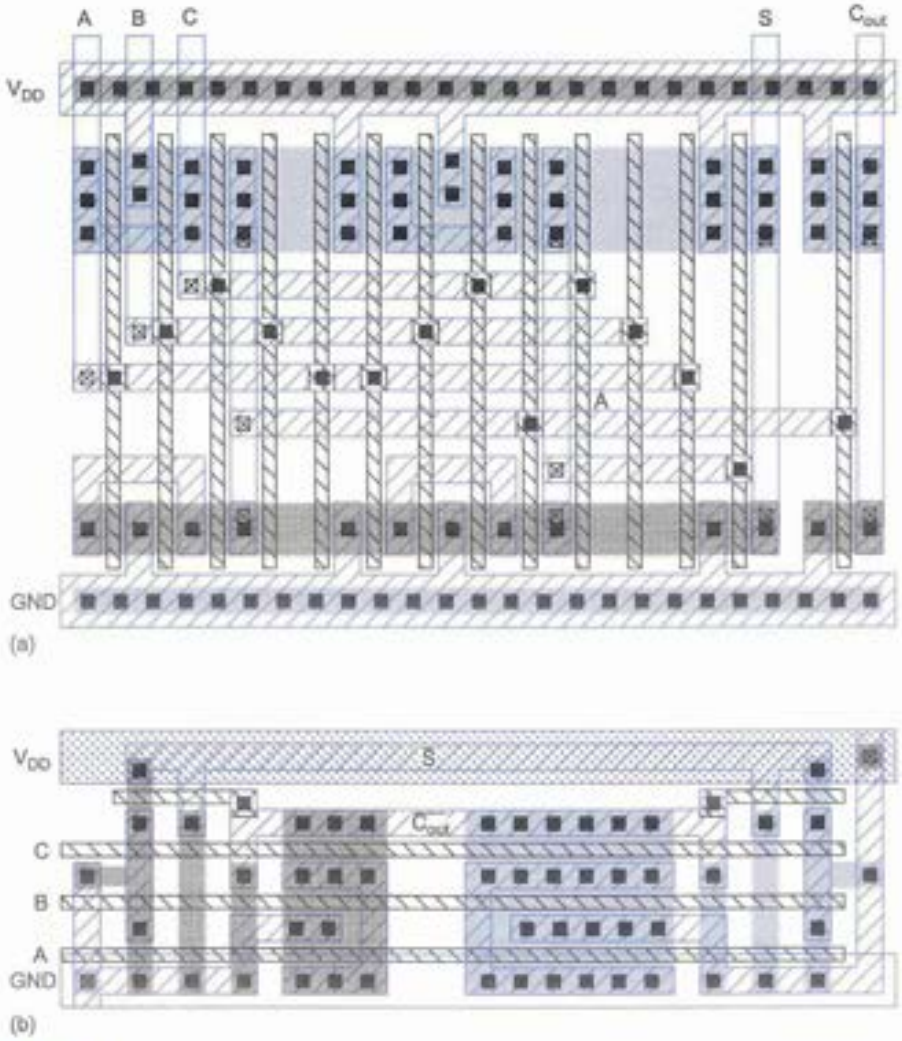


FIG 10.5 Full adder layouts. Color version on inside front cover.

A rather different full adder design uses transmission gates to form multiplexers and XORs. Figure 10.6(a) shows the transistor-level schematic using 24 transistors and providing buffered outputs of the proper polarity with equal delay. The design can be understood by parsing the transmission gate structures into multiplexers and an "invertible inverter" XOR structure (see Section 10.7.4), as drawn in Figure 10.6(b)[1]. Note that the multiplexer choosing S is configured to compute $P \oplus C$, as given in EQ (10.2).

[1]Some switch-level simulators, notably IRSIM, are confused by this XOR structure and may not simulate it correctly.

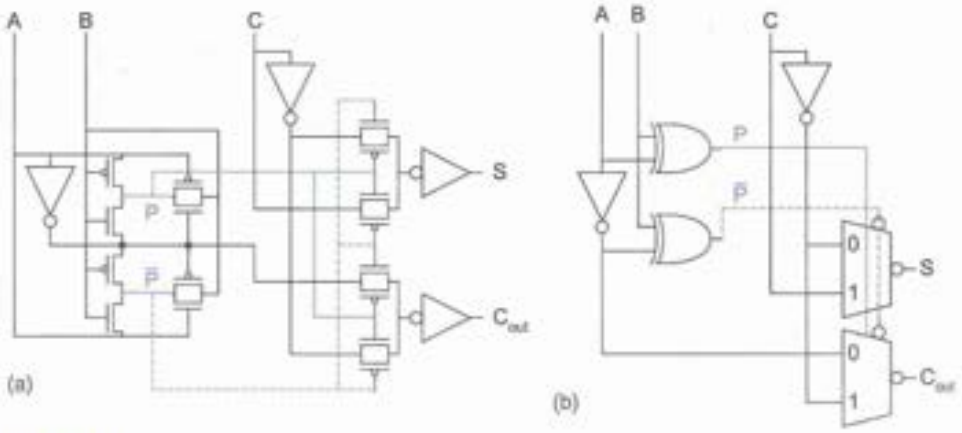


FIG 10.6 Transmission gate full adder

The transistor count can be reduced to 22 at the expense of an extra inverter delay by computing $\overline{P}$ from P rather than using a separate XOR. [Zhuang92] proposes such a design, shown in Figure 10.7. Note that in the Zhuang full adder, the multiplexer for S is replaced with an XOR, but the effect is the same.

Another transmission-gate approach uses complementary pass-transistor logic (CPL) [Yano90], shown in Figure 10.8. In comparison to a poorly optimized 40-transistor static CMOS full adder, the author finds CPL is twice as fast, 30% lower in power, and slightly smaller. On the other hand, in comparison to a careful implementation of the static CMOS full adder in Figure 10.4(b), [Zimmermann97a] finds the CPL delay slightly better, the power comparable, and the area much larger.

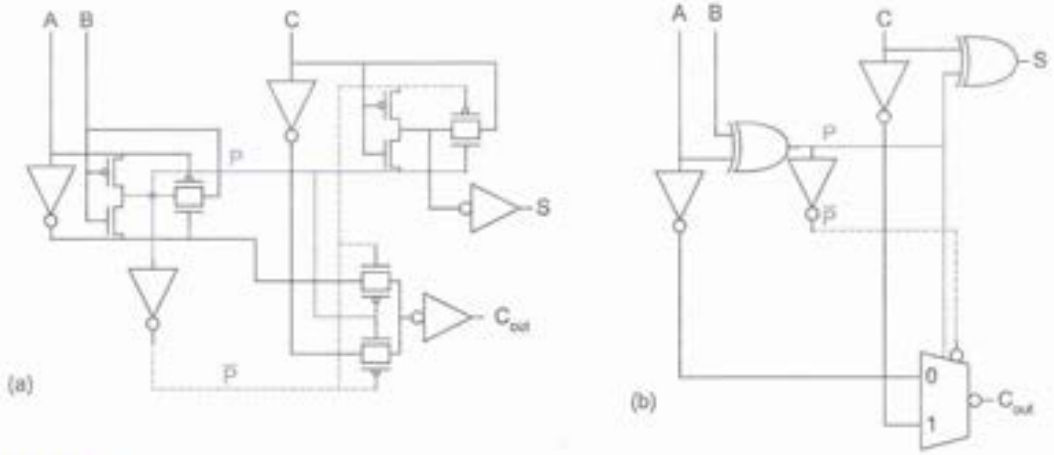


FIG 10.7 Zhuang full adder

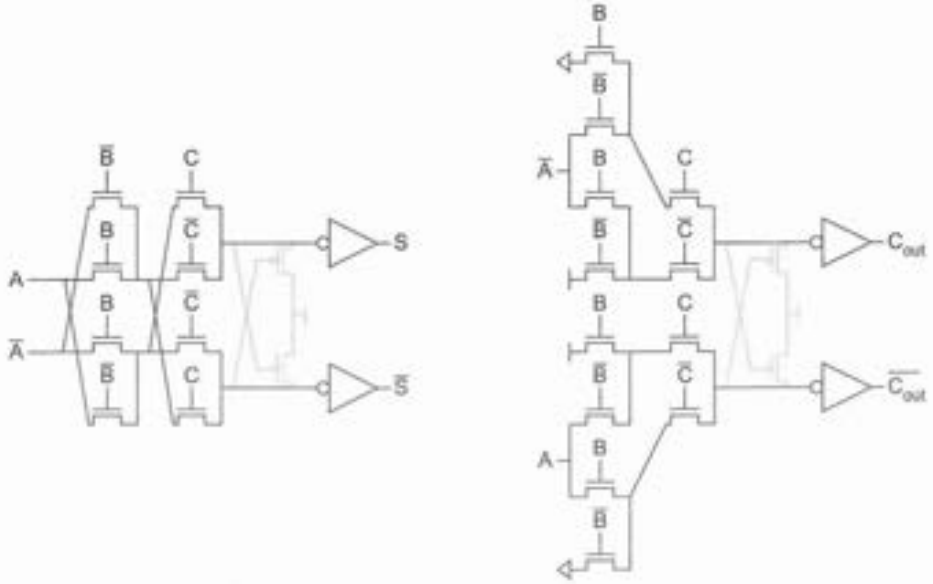


FIG 10.8 CPL full adder

Dynamic full adders are widely used in fast multipliers. As the sum logic inherently requires true and complementary versions of the inputs, dual-rail domino is necessary. Figure 10.9 shows such an adder using footless dual-rail domino XOR/XNOR and MAJORITY/MINORTY gates [Heikes94]. The delays to the two outputs are reasonably well balanced, which is important for multipliers where both paths are critical. It shares transistors in the sum gate to reduce transistor count and takes advantage of the symmetric property to provide identical layouts for the two carry gates.

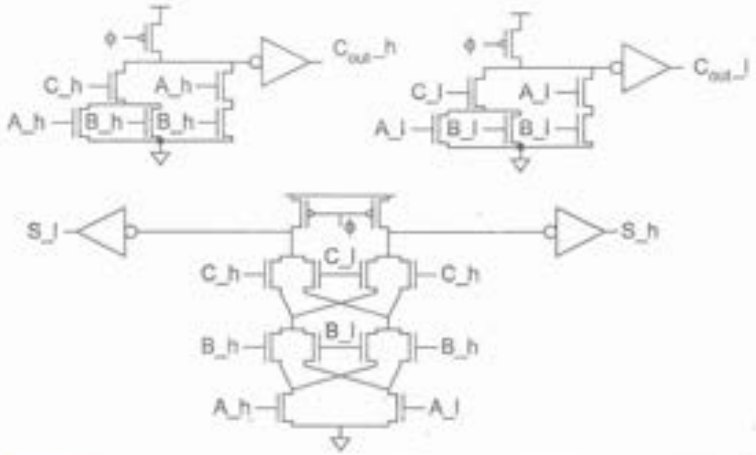


FIG 10.9 Dual-rail domino full adder

Static CMOS full adders typically have a delay of 2–3 FO4 inverters, while domino adders have a delay of about 1.5.

10.2.2 Carry-propagate Addition

N-bit adders take inputs $\{A_N, \ldots, A_1\}$, $\{B_N, \ldots, B_1\}$, and carry-in C_{in}, and compute the sum $\{S_N, \ldots, S_1\}$ and the carry-out of the most significant bit C_{out}, as shown in Figure 10.10. They are called *carry-propagate adders* (CPAs) because the carry into each bit can influence the carry into all subsequent bits. For example, Figure 10.11 shows the addition $1111_2 + 0000_2 + 0/1$, in which each of the sum and carry bits is influenced by C_{in}. The simplest design is the carry-ripple adder in which the carry-out of one bit is simply connected as the carry-in to the next. Faster adders look ahead to predict the carry-out of a multi-bit group. This is usually done by computing group PG signals to indicate whether the multi-bit group will propagate a carry-in or will generate a carry-out. Long adders use multiple levels of lookahead structures for even more speed.

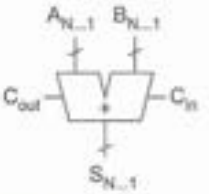


FIG 10.10 Carry-propagate adder

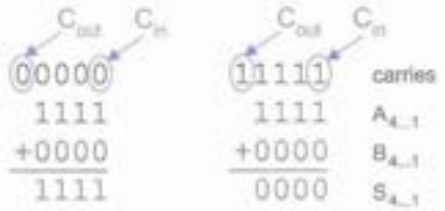


FIG 10.11 Example of carry propagation

10.2.2.1 Carry-ripple Adder An N-bit adder can be constructed by cascading N full adders, as shown in Figure 10.12(a) for $N = 4$. This is called a *carry-ripple adder* (or *ripple-carry adder*). The carry-out of bit i, C_i, is the carry-in to bit $i+1$. This carry is said to have twice the *weight* of the sum S_i. The delay of the adder is set by the time for the carries to ripple through the N stages, so the $t_{C\to C_{out}}$ delay should be minimized.

This delay can be reduced by omitting the inverters on the outputs, as was done in Figure 10.4(c). Because addition is a symmetric function, an inverting full adder receiving complementary inputs produces true outputs. Figure 10.12(b) shows a carry-ripple adder built from inverting full adders. Every other stage operates on complementary data. The delay in inverting the adder inputs or sum outputs is finessed out of the critical ripple-carry path.

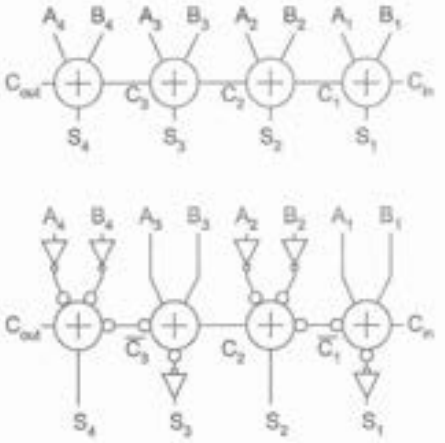


FIG 10.12 4-bit carry-ripple adder

10.2.2.2 Carry Generation and Propagation This section introduces notation commonly used in describing faster adders. Recall that the P and G *propagate*[2] and *generate* signals were defined in Section 10.2.1. We can generalize these signals to describe whether a group spanning bits $i \ldots j$, inclusive, generate a carry or propagate a carry. A group of bits generates a carry if its carry-out is true independent of the carry-in; it propagates a carry if its carry-out is true when there is a carry-in. These signals can be defined recursively for $i \geq k > j$ as

[2]Do not confuse the *propagate* term P with the *carry-propagate adder* (CPA).

$$G_{i:j} = G_{i:k} + P_{i:k} \cdot G_{k-1:j}$$
$$P_{i:j} = P_{i:k} \cdot P_{k-1:j} \tag{10.4}$$

with the base case

$$G_{i:i} \equiv G_i = A_i \cdot B_i$$
$$P_{i:i} \equiv P_i = A_i \oplus B_i \tag{10.5}$$

In other words, a group generates a carry if the upper (more significant) or the lower portion generates and the upper portion propagates that carry. The group propagates a carry if both the upper and lower portions propagate the carry.[3]

The carry-in must be treated specially. Let us define $C_0 = C_{in}$ and $C_N = C_{out}$. Then we can define generate and propagate signals for bit 0 as

$$G_{0:0} = C_{in}$$
$$P_{0:0} = 0 \tag{10.6}$$

Observe that the carry into bit i is the carry-out of bit i-1 and is $C_{i-1} = G_{i-1:0}$. This is an important relationship; *group generate* signals and *carries* will be used synonymously in the subsequent sections. We can thus compute the sum for bit i using EQ (10.2) as

$$S_i = P_i \oplus G_{i-1:0} \tag{10.7}$$

Hence, addition can be reduced to a three-step process of computing bitwise generate and propagate signals using EQs (10.5) and (10.6), combining these signals to determine group generates $G_{i:1:0}$ for all $N \geq i \geq 1$ using EQ (10.4), and then calculating the sums using EQ (10.7). These steps are illustrated in Figure 10.13. The first and third steps are routine, so most of the attention in the remainder of this section is devoted to alternatives for the group PG logic with different tradeoffs between speed, area, and complexity. Some of the hardware can be shared in the bitwise PG logic, as shown in Figure 10.14.

Many notations are used in the literature to describe the group PG logic. In general, PG logic is an example of a *prefix* computation [Leighton92]. It accepts inputs $[P_{N:N}, \ldots, P_{0:0}]$ and $[G_{N:N}, \ldots, G_{0:0}]$ and computes the *prefixes* $[G_{N:0}, \ldots, G_{0:0}]$ using the relationship given in EQ (10.4). This relationship is given many names in the literature including the *delta operator, fundamental carry operator,* and *prefix operator.* Many other problems such as priority encoding can be posed as prefix computations and all the techniques used to build fast group PG logic will apply, as we will explore in Section 10.10.

EQ (10.4) defines *valency-2* group PG logic because it combines pairs of smaller groups. It is also possible to define higher-valency group logic to use fewer stages of more complex gates [Beaumont-Smith99], as shown in EQ (10.8). For example, in valency-4

[3] Alternatively, some adders use $\overline{K}_i = A_i + B_i$ in place of P_i because OR is faster than XOR. The group logic becomes $G_{i:j} = G_{i:k} + \overline{K_{i:k}} \cdot G_{k-1:j}$ and $\overline{K_{i:j}} = \overline{K_{i:k}} \cdot \overline{K_{k-1:j}}$. However, P_i is still required in EQ (10.7) to compute the final sum.

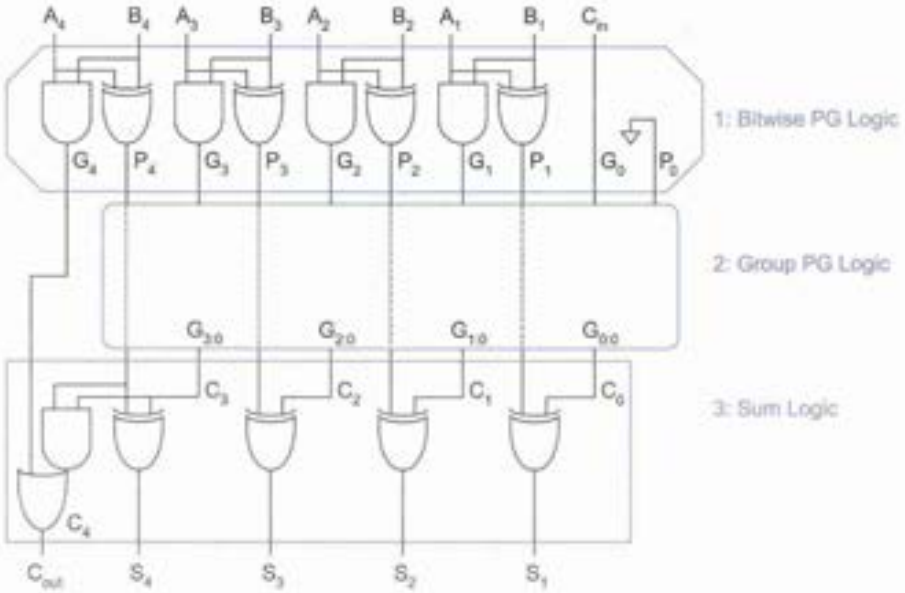


FIG 10.13 Addition with generate and propagate logic

group logic, a group propagates the carry if all four portions propagate. A group generates a carry if the upper portion generates, the second portion generates and the upper propagates, the third generates and the upper two propagate, or the lower generates and the upper three propagate. Logical effort teaches us that the best stage effort is about 4. Therefore, it is not necessarily better to build fewer stages of higher-valency gates; simulations or calculations should be done to compare the alternatives for a given technology.

FIG 10.14 Shared bitwise PG logic

$$\left.\begin{aligned} G_{i:j} &= G_{i:k} + P_{i:k} \cdot G_{k-1:l} + P_{i:k} \cdot P_{k-1:l} \cdot G_{l-1:m} + P_{i:k} \cdot P_{k-1:l} \cdot P_{l-1:m} \cdot G_{m-1:j} \\ &= G_{i:k} + P_{i:k}\left(G_{k-1:l} + P_{k-1:l}\left(G_{l-1:m} + P_{l-1:m} G_{m-1:j}\right)\right) \\ P_{i:j} &= P_{i:k} \cdot P_{k-1:l} \cdot P_{l-1:m} \cdot P_{m-1:j} \end{aligned}\right\} \quad (i \geq k > l > m > j) \qquad (10.8)$$

10.2.2.3 PG Carry-ripple Addition The critical path of the carry-ripple adder passes from carry-in to carry-out along the carry chain majority gates. As the P and G signals will have already stabilized by the time the carry arrives, we can use them to simplify the majority function into an AND-OR gate[4]:

[4]Whenever positive logic such as AND-OR is described, it is implicit that you can also use an AOI gate and alternate positive and negative polarity stages as was done in Figure 10.12(b) to save area and delay.

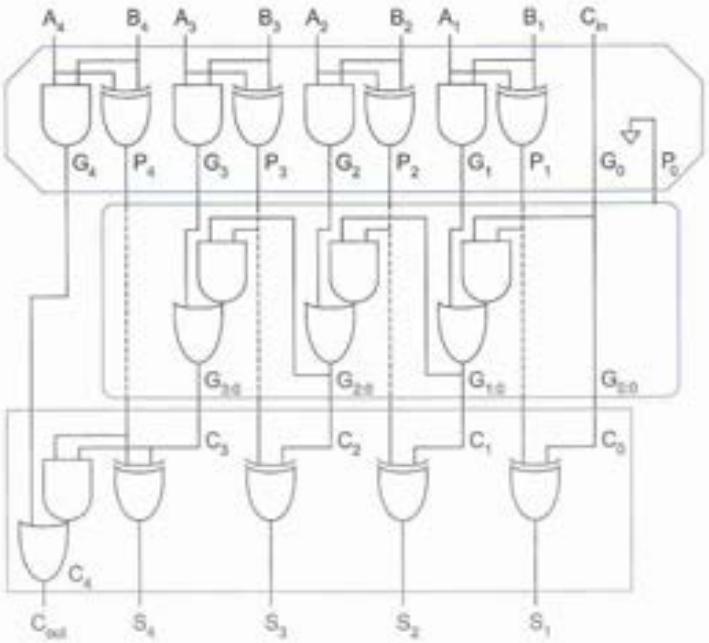


FIG 10.15 4-bit carry-ripple adder using PG logic

$$
\begin{aligned}
C_i &= A_iB_i + (A_i + B_i)C_{i-1} \\
&= A_iB_i + (A_i \oplus B_i)C_{i-1} \\
&= G_i + P_iC_{i-1}
\end{aligned}
\tag{10.9}
$$

Because $C_i = G_{i:0}$, carry-ripple addition can now be viewed as the extreme case of group PG logic in which a 1-bit group is combined with an i-bit group to form an $(i-1)$-bit group

$$G_{i:0} = G_i + P_i \cdot G_{i-1:0} \tag{10.10}$$

In this extreme, the group propagate signals are never used and need not be computed. Figure 10.15 shows a 4-bit carry-ripple adder. The critical carry path now proceeds through a chain of AND-OR gates rather than a chain of majority gates. Figure 10.16 illustrates the group PG logic for a 16-bit carry-ripple adder, where the AND-OR gates in the group PG network are represented with gray cells.

Diagrams like these will be used to compare a variety of adder architectures in subsequent sections. The diagrams use black cells, gray cells, and white buffers defined in

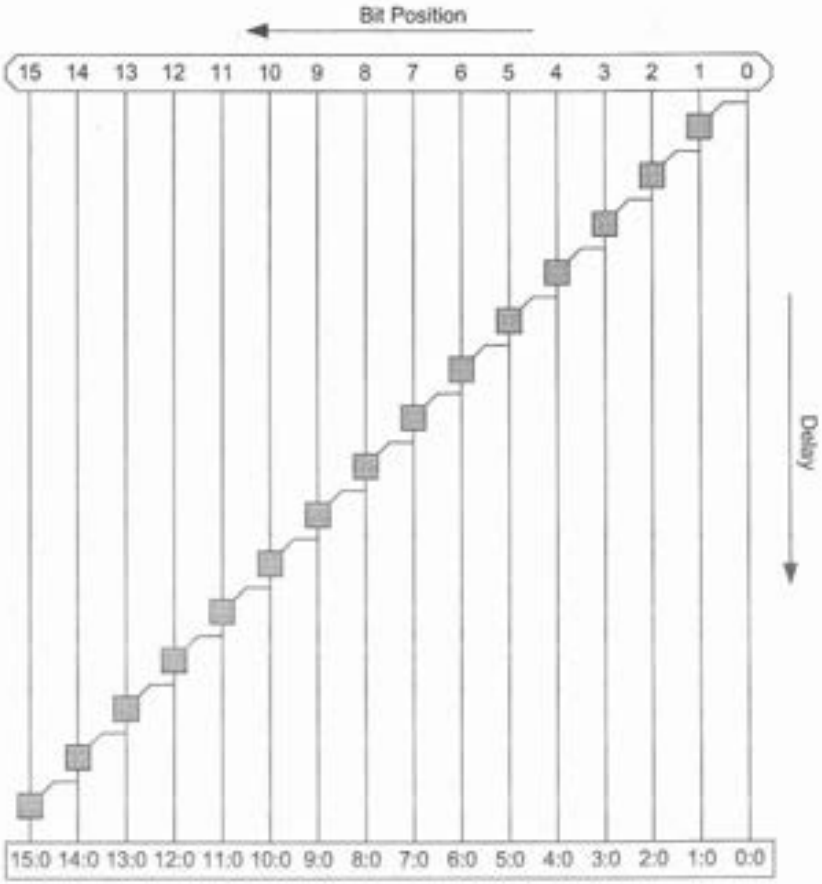


FIG 10.16 Carry-ripple adder group PG network

Figure 10.18(a) for valency-2 cells. Black cells contain the group generate and propagate logic (an AND-OR gate and an AND gate) defined in EQ (10.4). Gray cells containing only the group generate logic are used at the final cell position in each column because only the group generate signal is required to compute the sums. Buffers can be used to minimize the load on critical paths. Each line represents a *bundle* of the group generate and propagate signals (propagate signals are omitted after gray cells). The bitwise PG and sum XORs are abstracted away in the top and bottom boxes and it is assumed that an AND-OR gate operates in parallel with the sum XORs to compute the carry-out:

$$C_{out} = G_{N:0} = G_N + P_N G_{N-1:0} \tag{10.11}$$

The cells are arranged along the vertical axis according to the time at which they operate [Guyot97]. From Figure 10.16 it is apparent that the carry-ripple adder critical path delay is:

$$t_{ripple} = t_{pg} + (N-1)t_{AO} + t_{xor} \tag{10.12}$$

where t_{pg} is the delay of the 1-bit propagate/generate gates, t_{AO} is the delay of the AND-OR gate in the gray cell, and t_{xor} is the delay of the final sum XOR.

Often, using noninverting gates leads to more stages of logic than are necessary. Figure 10.17(b) shows how to alternate two types of inverting stages on alternate rows of the group PG network to remove extraneous inverters. For best performance, $G_{k-1:j}$ should drive the inner transistor in the series stack. You can also reduce the number of stages by using higher-valency cells, as shown in Figure 10.17(c) for a valency-4 black cell.

10.2.2.4 Manchester Carry Chain Adder The carry chain can also be built from switch logic using propagate, generate, and kill signals. The majority gate of Figure 10.18(a) can be replaced with a switch network. Figure 10.18(b) shows a static implementation operating on a complementary carry. The complementary carry can be propagated through the transmission gate, generated with the nMOS transistor, or killed with the pMOS transistor. Figure 10.18(c) shows a dynamic version that is faster and requires less hardware.

Multiple stages are directly connected to build a *Manchester carry chain*, as shown in Figure 10.19(a) [Kilburn59]. The resistance and capacitance of the carry chain grow with the length, so the delay grows with the square of length. This is clearly not viable for long adders. As with long wires, the delay can be made linear with length by periodically breaking the chain and inserting an inverter to buffer the signals. The best chain length depends on the parasitic capacitance and can be determined through simulation or calculations for a particular technology (see Exercise 10.4), but is typically 3 or 4; Figure 10.19(b) shows a valency-4 carry chain. The widths of the transistors along the chain can be tapered to reduce parasitic delay.

Observe that the Manchester carry chain computes the functions

$$\begin{aligned}
C_0 &= G_{0:0} = C_0 \\
C_1 &= G_{1:0} = G_1 + P_1 C_0 \\
C_2 &= G_{2:0} = G_2 + P_2(G_1 + P_1 C_0) \\
C_3 &= G_{3:0} = G_3 + P_3(G_2 + P_2(G_1 + P_1 C_0))
\end{aligned} \tag{10.13}$$

$G_{3:0}$ is like the valency-4 group generate circuit of EQ (10.8), while the other outputs are the generate signals for smaller groups (including a simple buffer of the input). In other words, the carry chain can be viewed as a buffer and three gray cells of increasing valency, as shown in Figure 10.19(c). If the carry chain of Figure 10.19(b) is redrawn in a more conventional form (Figure 10.20), it can also be seen to be another representation of a footless multiple-output domino gate, as was discussed in Section 6.2.4.6.

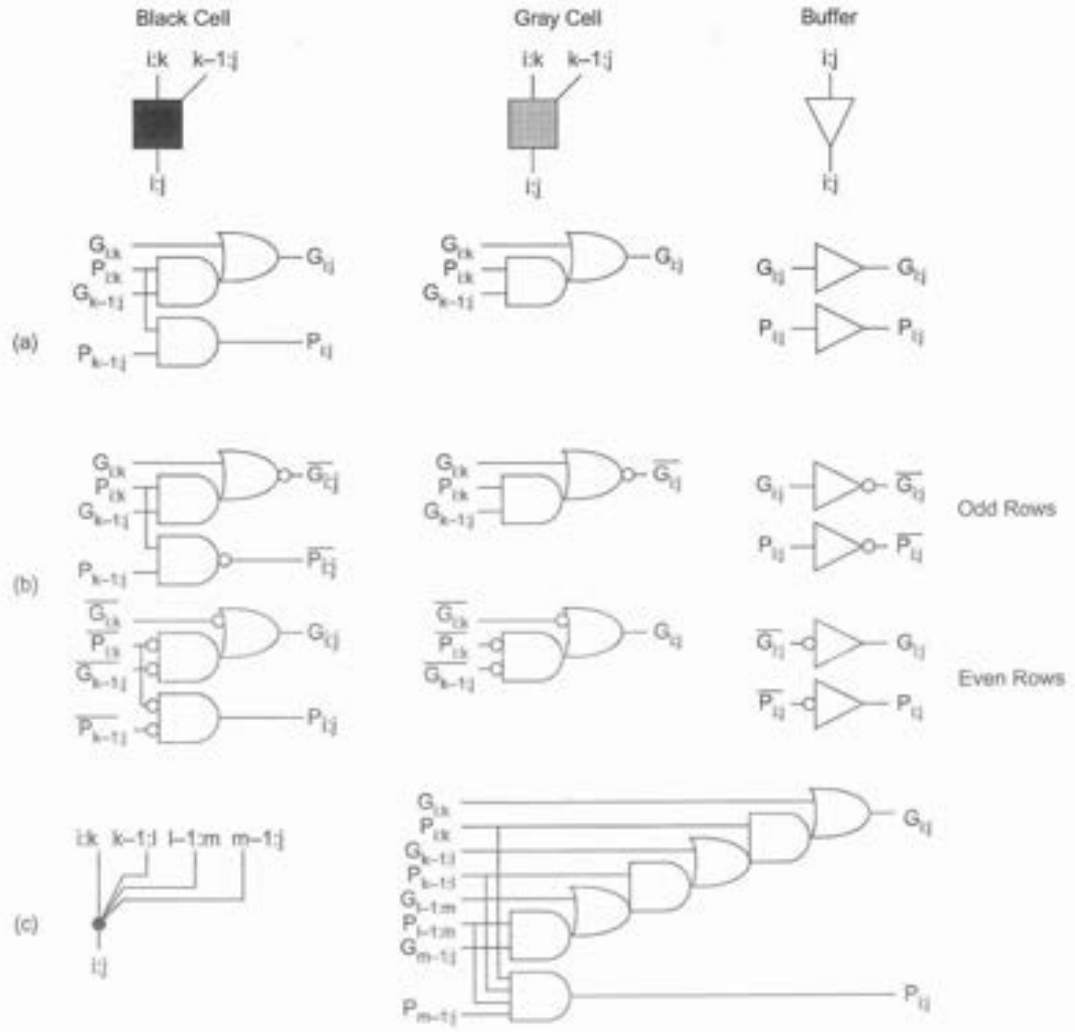


FIG 10.17 Group PG cells

Figure 10.21 shows a Manchester carry chain adder using valency-4 stages. It is similar to the carry-ripple adder, but uses $N/3$ stages.

10.2.2.5 Carry-skip Adder The critical path of a Manchester carry chain adder involves a series propagate transistor for each bit of the adder. This is a significant improvement over the carry-ripple adder, which uses a majority gate or AND-OR gate for each bit, but still may be too slow for large adders. The *carry-skip* (also called *carry-bypass*) adder, first

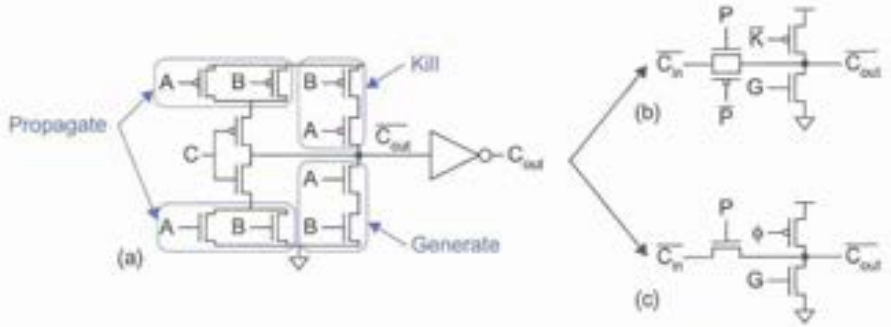


FIG 10.18 Carry chain designs

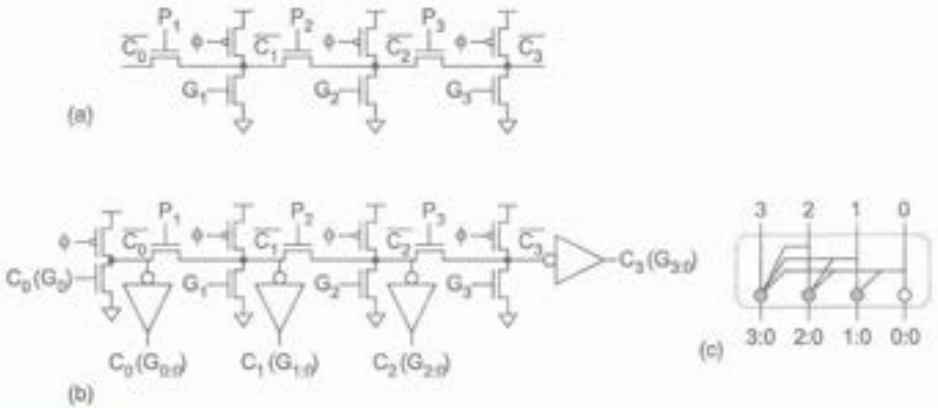


FIG 10.19 Manchester carry chains

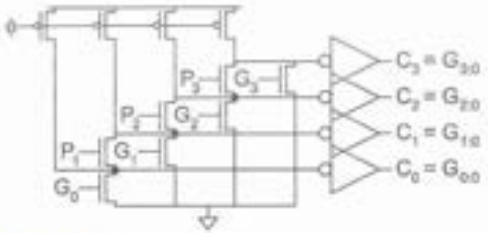


FIG 10.20 Equivalence of Manchester carry chain and multiple-output domino gate

proposed by Charles Babbage in the nineteenth century and used for many years in mechanical calculators, shortens the critical path by computing the group propagate signals for each carry chain and using this to skip over long carry ripples [Morgan59, Lehman61]. Figure 10.22 shows a carry skip adder built from 4-bit groups. The rectangles compute the bit-wise propagate and generate signals (as in Figure 10.15), and also contain a 4-input AND gate for the propagate signal of the 4-bit group. The skip multiplexer selects the group carry-in if the group propagate is true or the ripple adder carry-out otherwise.

The critical path through Figure 10.22 begins with generating a carry from bit 1, and then propagating it through the remainder of the adder. The carry must ripple through the

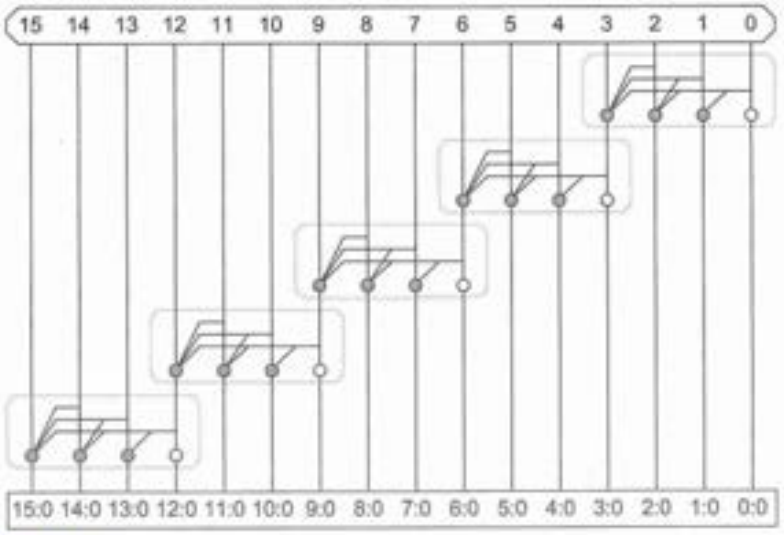


FIG 10.21 Manchester carry chain adder group PG network

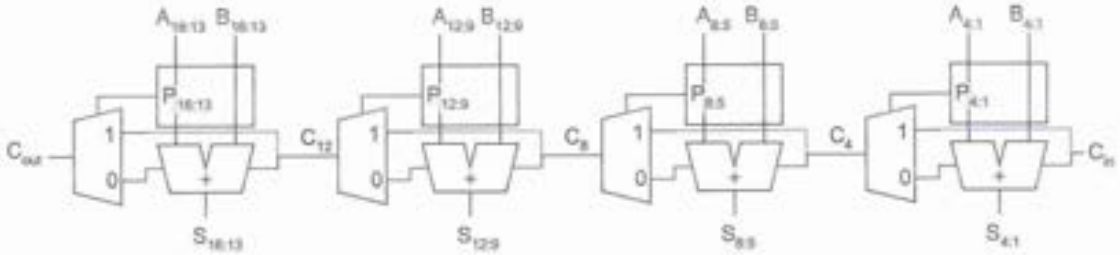


FIG 10.22 Carry-skip adder

next three bits, but then may skip across the next two 4-bit blocks. Finally, it must ripple through the final 4-bit block to produce the sums. This is illustrated in Figure 10.23. The 4-bit ripple chains at the top of the diagram determine if each group generates a carry. The carry skip chain in the middle of the diagram skips across 4-bit blocks. Finally, the 4-bit ripple chains with the blue lines represent *the same adders* that can produce a carry-out when a carry-in is bypassed to them. Note that the final AND-OR and column 16 are not strictly necessary because C_{out} can be computed in parallel with the sum XORs using EQ (10.11).

In general, an N-bit adder using k n-bit groups ($N = n \cdot k$) has a delay of

$$t_{skip} = t_{pg} + 2(n-1) + (k-1)t_{dO} + t_{xor} \tag{10.14}$$

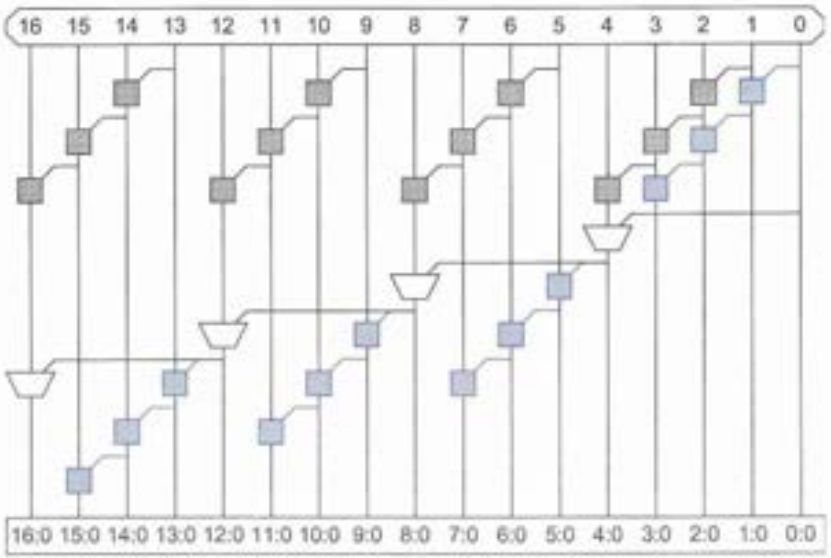


FIG 10.23 Carry-skip adder PG network

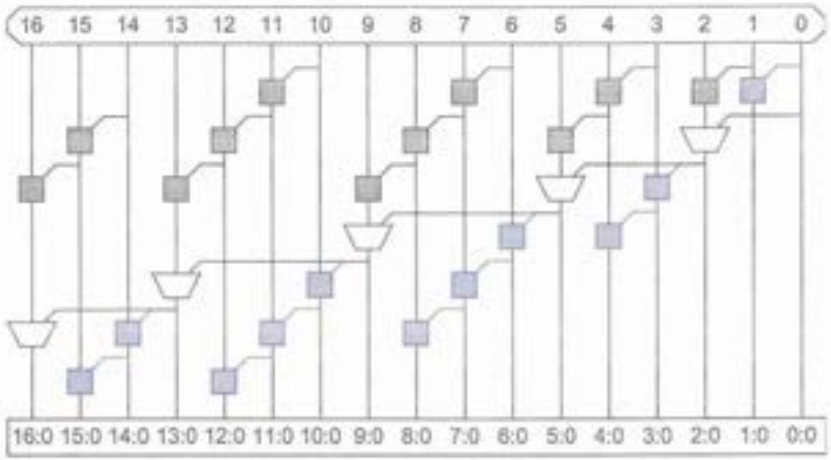


FIG 10.24 Variable group size carry-skip adder PG network

This critical path depends on the length of the first and last group and the number of groups. In the more significant bits of the network, the ripple results are available early. Thus, the critical path could be shortened by using shorter groups at the beginning and end and longer groups in the middle. Figure 10.24 shows such a PG network using groups

of length [2, 3, 4, 4, 3], as opposed to [4, 4, 4, 4], which saves two levels of logic in a 16-bit adder.

The hardware cost of a carry-skip adder is equal to that of a simple carry-ripple adder plus k multiplexers and k n-input AND gates. It is attractive when ripple-carry adders are too slow, but the hardware cost must still be kept low. For long adders, you could use a multi-level skip approach to skip across the skips. A great deal of research has gone into choosing the best group size and number of levels [Majerski67, Oklobdžija85, Guyot87, Chan90, Kantabutra91], although now, parallel prefix adders are generally used for long adders instead.

It might be tempting to replace each skip multiplexer in Figure 10.22 and Figure 10.23 with an AND-OR gate combining the carry-out of the n-bit adder or the group carry-in and group propagate. Indeed, this works for domino-carry skip adders in which the carry out is precharged each cycle; it also works for carry-lookahead adders and carry-select adders covered in the subsequent section. However, it introduces a sneaky long critical path into an ordinary carry-skip adder. Imagine summing 111…111 + 000…000 + C_{in}. All of the group propagate signals are true. If C_{in} = 1, every 4-bit block will produce a carry-out. When C_{in} falls, the falling carry signal must ripple through all N bits because of the path through the carry out of each n-bit adder. Domino-carry skip adders avoid this path because all of the carries are forced low during precharge, so they can use AND-OR gates.

Figure 10.25 shows how a Manchester carry chain can be modified to perform carry skip [Chan90]. A valency-5 chain is used to skip across groups of 4 bits at a time.

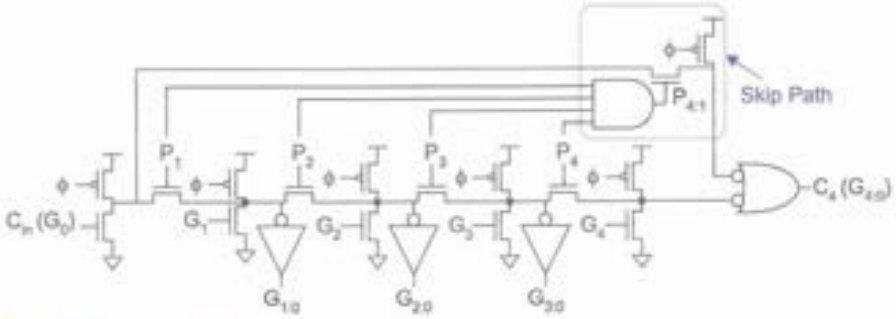


FIG 10.25 Carry-skip adder Manchester stage

10.2.2.6 Carry-lookahead Adder The *carry-lookahead adder* (CLA) [Weinberger58] is similar to the carry-skip adder, but computes group generate signals as well as group propagate signals to avoid waiting for a ripple to determine if the first group generates a carry. Such an adder is shown in Figure 10.26 and its PG network is shown in Figure 10.27 using valency-4 black cells to compute 4-bit group PG signals.

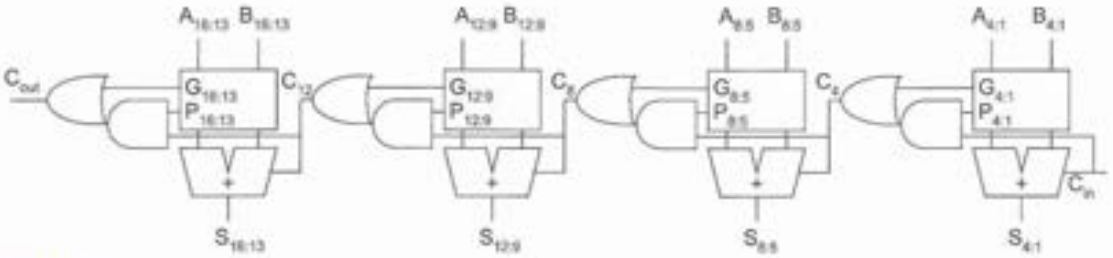


FIG 10.26 Carry-lookahead adder

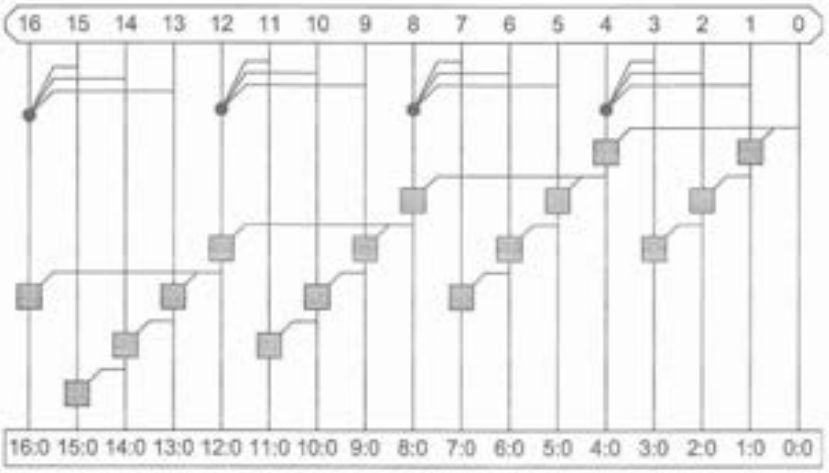


FIG 10.27 Carry-lookahead adder group PG network

In general, a CLA using k groups of n bits each has a delay of

$$t_{cla} = t_{pg} + t_{pg(n)} + \left[(n-1)+(k-1)\right]t_{AO} + t_{sum} \tag{10.15}$$

where $t_{pg(n)}$ is the delay of the AND-OR-AND-OR-...-AND-OR gate computing the valency-n generate signal. This is no better than the variable-length carry-skip adder in Figure 10.24 and requires the extra n-bit generate gate, so the simple CLA is seldom a good design choice. However, it forms the basis for understanding faster adders presented in the subsequent sections.

CLAs often use higher-valency cells to reduce the delay of the n-bit additions by computing the carries in parallel. Figure 10.28 shows such a CLA in which the 4-bit adders are built using Manchester carry chains or multiple static gates operating in parallel.

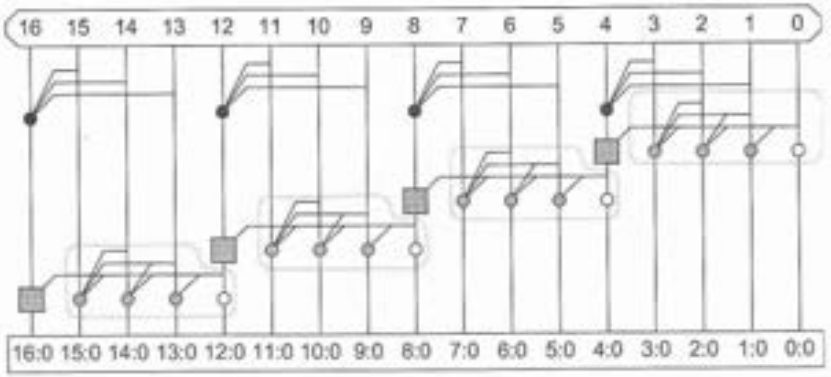


FIG 10.28 Improved CLA group PG network

10.2.2.7 Carry-select, Carry-increment, and Conditional-sum Adders The critical path of the carry-skip and carry-lookahead adders involves calculating the carry into each n-bit group, and then calculating the sums for each bit within the group based on the carry-in. A standard logic design technique to accelerate the critical path is to precompute the outputs for both possible inputs, and then use a multiplexer to select between the two output choices. The *carry-select adder* [Bedrij62] shown in Figure 10.29 does this with a pair of n-bit adders in each group. One adder calculates the sums assuming a carry-in of 0 while the other calculates the sums assuming a carry-in of 1. The actual carry triggers a multiplexer that chooses the appropriate sum. The critical path delay is

$$t_{select} = t_{pg} + [n + (k-2)]t_{dO} + t_{mux} \tag{10.16}$$

The two n-bit adders are redundant in that both contain the initial PG logic and final sum XOR. [Tyagi93] reduces the size by factoring out the common logic and simplifying

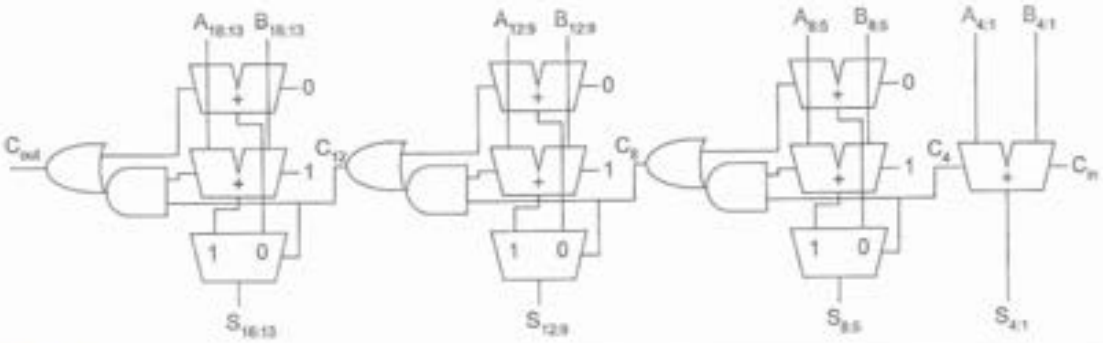


FIG 10.29 Carry-select adder

the multiplexer to a gray cell, as shown in Figure 10.30. This is sometimes called a *carry-increment* adder [Zimmermann96]. It uses a short ripple chain of black cells to compute the PG signals for bits within a group. The bits spanned by each group are annotated on the diagram. When the carry-out from the previous group becomes available, the final gray cells in each column determine the carry-out, which is true if the group generates a carry or if the group propagates a carry and the previous group generated a carry. The carry-increment adder has about twice as many cells in the PG network as a carry-ripple adder. The critical path delay is about the same as that of a carry-select adder because a mux and XOR are comparable, but the area is smaller.

$$t_{\text{increment}} = t_{pg} + \left[(n-1)+(k-1)\right]t_{AO} + t_{xor} \tag{10.17}$$

Of course, Manchester carry chains or higher-valency cells can be used to speed the ripple operation to produce the first group generate signal. In that case, the ripple delay is replaced by a group PG gate delay and the critical path becomes:

$$t_{\text{increment}} = t_{pg} + t_{pg(n)} + \left[(k-1)\right]t_{AO} + t_{xor} \tag{10.18}$$

As with the carry-skip adder, the carry chains for the more significant bits complete early. Again, we can use variable-length groups to take advantage of the extra time, as shown in Figure 10.31(a). With such a variable group size, the delay reduces to:

$$t_{\text{increment}} = t_{pg} + \sqrt{2N}\,t_{AO} + t_{xor} \tag{10.19}$$

The delay equations do not account for the fanout that each stage must drive. The fanouts in a variable-length group can become large enough to require buffering between stages. Figure 10.31(b) shows how buffers can be inserted to reduce the branching effort

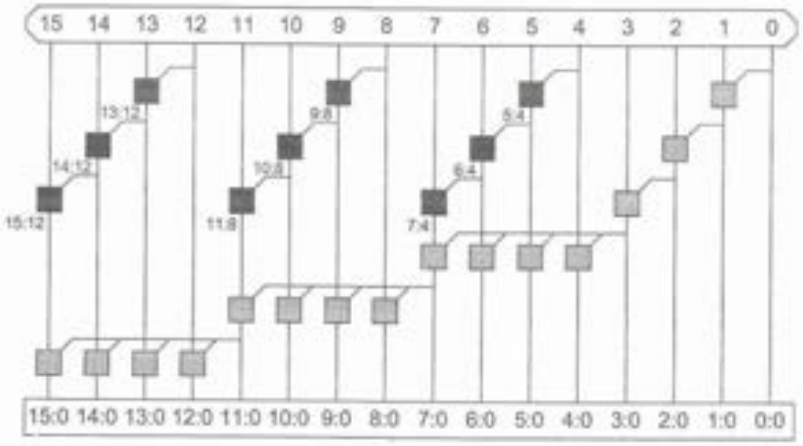


FIG 10.30 Carry-increment adder PG network

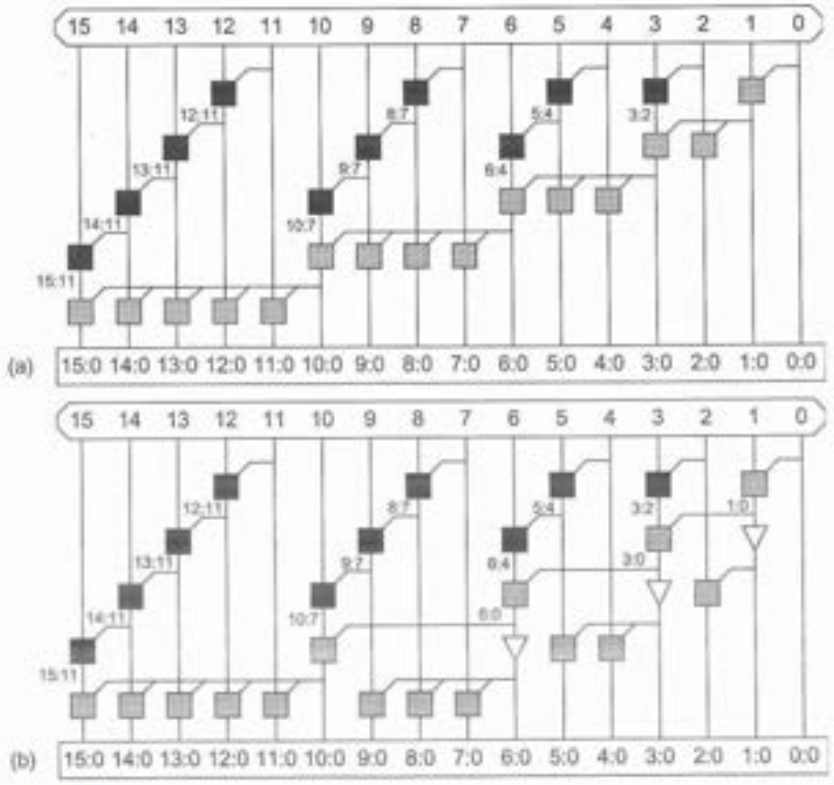


FIG 10.31 Variable-length carry-increment adder

while not impeding the critical lookahead path; this is a useful technique in many other applications.

In wide adders, we can recursively apply multiple levels of carry-select or carry-increment. For example, a 64-bit carry-select adder can be built from four 16-bit carry-select adders, each of which selects the carry-in to the next 16-bit group. Taking this to the limit, we obtain the *conditional-sum* adder [Sklansky60] that performs carry-select starting with groups of 1 bit and recursively doubling to $N/2$ bits. Figure 10.32 shows such an adder. In the first two rows, full adders compute the sum and carry-out for each bit assuming carries-in of 0 and 1, respectively. In the next two rows, multiplexer pairs select the sum and carry-out of the upper bit of each block of two, again assuming carries-in of 0 and 1. In the next two rows, multiplexers select the sum and carry-out of the upper two bits of each block of four, and so forth.

Figure 10.33 shows the operation of a conditional-sum adder in action for $N = 16$ with $C_{in} = 0$. In the block width 1 row, a pair of full adders compute the sum and carry-out for each column. One adder operates assuming the carry-in to that column is 0, while the other assumes it is 1. In the block width 2 row, the adder selects the sum for the upper half

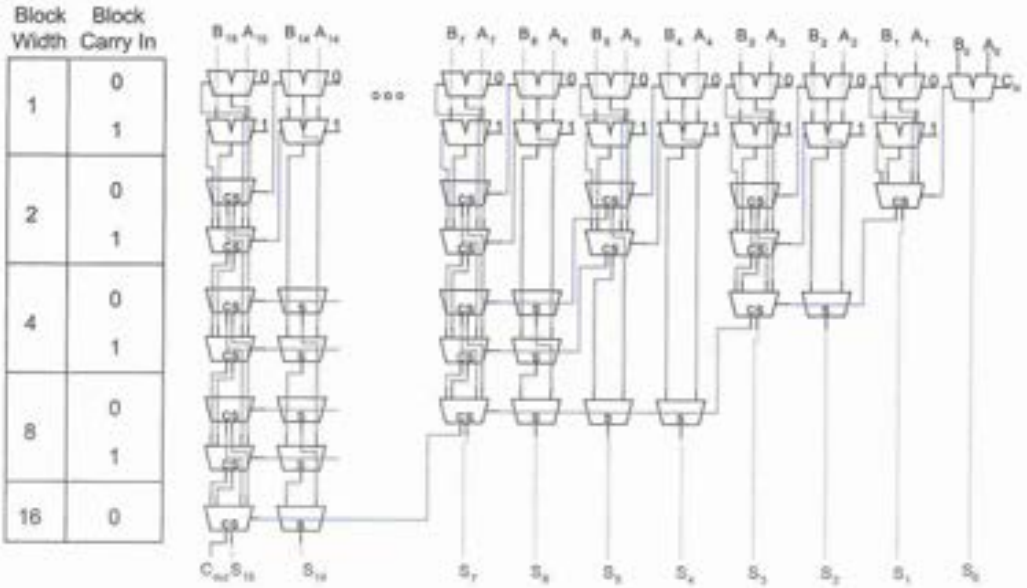


FIG 10.32 Conditional-sum adder

Block Width	Block Carry In		16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	C_{in}
		a	1	0	1	1	1	0	1	1	0	1	1	0	1	1	0	1	0
		b	0	0	0	1	1	0	0	1	1	0	1	1	0	1	1	0	
			Block Sum and Carry Out																
1	0	s	1	0	1	0	0	0	1	0	1	1	0	1	1	0	1	1	
		c	0	0	0	1	1	0	0	1	0	0	1	0	0	1	0	0	
	1	s	0	1	0	1	1	1	0	1	0	0	1	0	0	1	0		
		c	1	0	1	1	1	0	1	1	1	1	1	1	1	1	1		
2	0	s	1	0	0	0	0	0	0	0	1	1	0	1	0	0	1	1	
		c	0		1		1		1		0		1		1		0		
	1	s	1	1	0	1	0	1	0	1	0	0	1	0	0	1			
		c	0		1		1		1		1		1		1				
4	0	s	1	1	0	0	0	1	0	0	0	0	0	1	0	0	1	1	
		c	0				1				1				1				
	1	s	1	1	0	1	0	1	0	1	0	0	1	0					
		c	0				1				1								
8	0	s	1	1	0	1	0	1	0	0	0	0	1	0	0	0	1	1	
		c	0								1								
	1	s	1	1	0	1	0	1	0	1									
		c	0																
16	0	s	1	1	0	1	0	1	0	1	0	0	1	0	0	0	1	1	Sum
		c	0																C_{out}
	1	s																	
		c																	

FIG 10.33 Conditional-sum addition example

of each block (the even-numbered columns) based on the carry-out of the lower half. It also computes the carry-out of the pair of bits. Again, this is done twice, for both possibilities of carry-in to the block. In the block width 4 row, the adder again selects the sum for the upper half based on the carry-out of the lower half and finds the carry-out of the entire block. This process is repeated in subsequent rows until the 16-bit sum and the final carry-out are selected.

The conditional-sum adder involves nearly $2N$ full adders and $2N\log_2 N$ multiplexers. As with carry-select, the conditional-sum adder can be improved by factoring out the sum XORs and using AND-OR gates in place of multiplexers. This leads us to the Sklansky tree adder discussed in the next section.

10.2.2.8 Tree Adders For wide adders ($N > \sim 16$ bits), the delay of carry-lookahead (or carry-skip or carry-select) adders becomes dominated by the delay of passing the carry through the lookahead stages. This delay can be reduced by looking ahead across the lookahead blocks [Weinberger58]. In general, you can construct a multilevel tree of look-ahead structures to achieve delay that grows with log N. Such adders are variously referred to as *tree* adders, *logarithmic* adders, *multilevel-lookahead* adders, *parallel-prefix* adders, or simply *lookahead* adders. The last name appears occasionally in the literature, but is not recommended because it does not distinguish whether multiple levels of lookahead are used.

There are many ways to build the lookahead tree that offer tradeoffs among the number of stages of logic, the number of logic gates, the maximum fanout on each gate, and the amount of wiring between stages. Three fundamental trees are the Brent-Kung, Sklansky, and Kogge-Stone architectures. We begin by examining each in the valency-2 case that combines pairs of groups at each stage.

The *Brent-Kung* tree [Brent82] (Figure 10.34(a)) computes prefixes for 2-bit groups. These are used to find prefixes for 4-bit groups, which in turn are used to find prefixes for 8-bit groups, and so forth. The prefixes then fan back down to compute the carries-in to each bit. The tree requires $2(\log_2 N) - 1$ stages. The fanout is limited to 2 at each stage. The diagram shows buffers used to minimize the fanout and loading on the gates, but in practice, the buffers are generally omitted.

The *Sklansky* or *divide-and-conquer* tree [Sklansky60] (Figure 10.34(b)) reduces the delay to $(\log_2 N)$ stages by computing intermediate prefixes along with the large group prefixes. This comes at the expense of fanouts that double at each level: The gates fanout to [8, 4, 2, 1] other columns. These high fanouts cause poor performance on wide adders unless the gates are appropriately sized or the critical signals are buffered before being used for the intermediate prefixes. Transistor sizing can cut into the regularity of the layout because multiple sizes of each cell are required, although the larger gates can spread into adjacent columns. Note that the recursive doubling in the Sklansky tree is analogous to the conditional-sum adder of Figure 10.32. With appropriate buffering, the fanouts can be reduced to [8, 1, 1, 1], as explored in Exercise 10.7.

The *Kogge-Stone* tree [Kogge73] (Figure 10.34(c)) achieves both $(\log_2 N)$ stages and fanout of 2 at each stage. This comes at the cost of many long wires that must be routed between stages. The tree also contains more PG cells; while this may not impact the area if the adder layout is on a regular grid, it will increase power consumption. Despite these

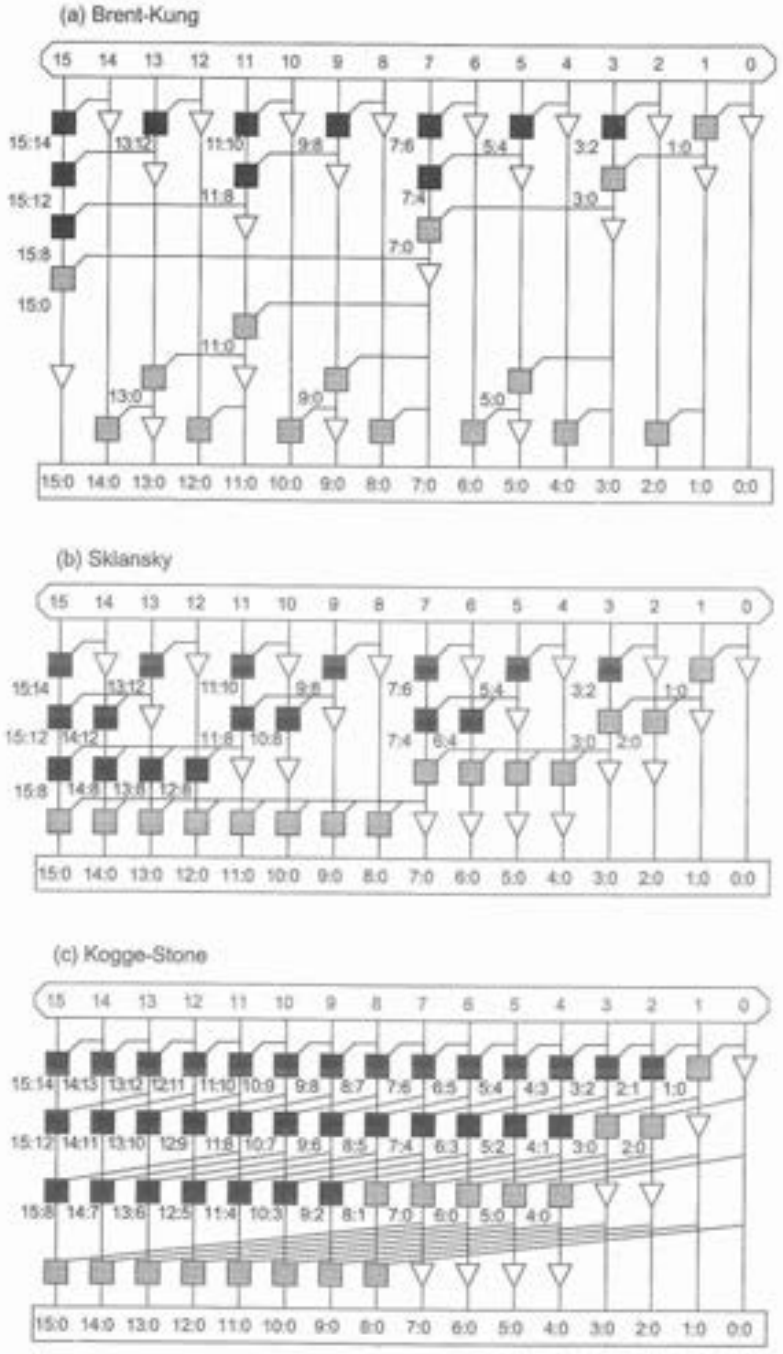
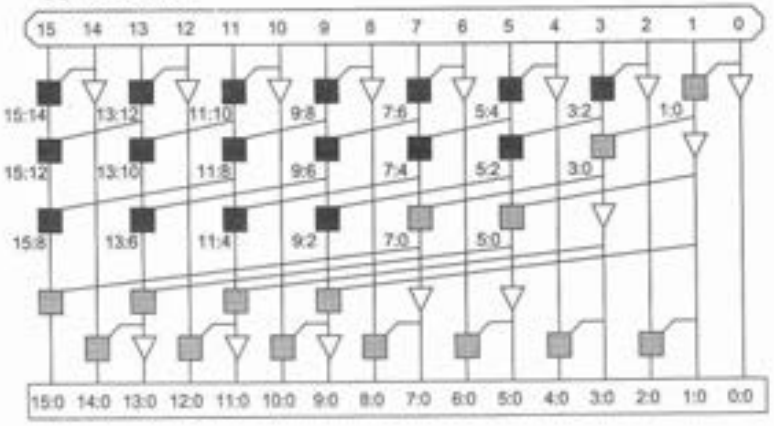
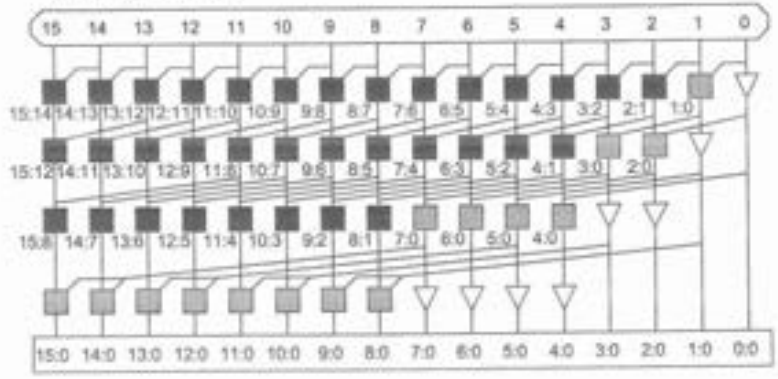


FIG 10.34 Tree adder PG networks

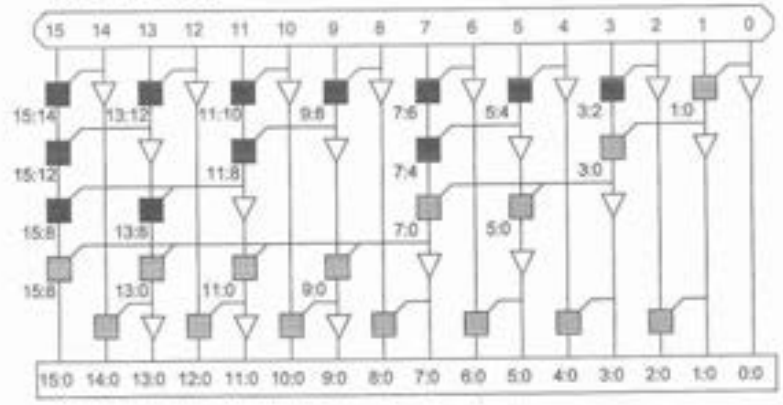
(d) Han-Carlson



(e) Knowles [2,1,1,1]



(f) Ladner-Fischer



costs, the Kogge-Stone tree is widely used in high-performance 32-bit and 64-bit adders.

In summary, a Sklansky or Kogge-Stone tree adder reduces the critical path to:

$$t_{tree} \approx t_{pg} + \lceil \log_2 N \rceil t_{dO} + t_{xor} \tag{10.20}$$

An ideal tree adder would have $\log_2 N$ levels of logic, fanout never exceeding 2, and no more than 1 wiring track ($G_{i,j}$ and $P_{i,j}$ bundle) between each row. The basic tree architectures represent cases that approach the ideal, but each differ in one respect. Brent-Kung has too many logic levels. Sklansky has too much fanout. And Kogge-Stone has too many wires. Between these three extremes, the Han-Carlson, Ladner-Fischer, and Knowles trees fill out the design space with different compromises between number of stages, fanout, and wire count.

The *Han-Carlson* trees [Han87] are a family of networks between Kogge-Stone and Brent-Kung. Figure 10.34(d) shows such a tree that performs Kogge-Stone on the odd-numbered bits, and then uses one more stage to ripple into the even positions.

The *Knowles* trees [Knowles01] are a family of networks between Kogge-Stone and Sklansky. All of these trees have $\log_2 N$ stages, but differ in the fanout and number of wires. If we say that 16-bit Kogge-Stone and Sklansky adders drive fanouts of [1, 1, 1, 1] and [8, 4, 2, 1] other columns, respectively, the Knowles networks lie between these extremes. For example, Figure 10.34(e) shows a [2, 1, 1, 1] Knowles tree that halves the number of wires in the final track at the expense of doubling the load on those wires.

The *Ladner-Fischer* trees [Ladner80] are a family of networks between Sklansky and Brent-Kung. Figure 10.34(f) is similar to Sklansky, but computes prefixes for the odd-numbered bits and again uses one more stage to ripple into the even positions. Cells at high-fanout nodes must still be sized or ganged appropriately to achieve good speed. Note that some authors use Ladner-Fischer synonymously with Sklansky.

An advantage of the Brent-Kung network and those related to it (Han-Carlson and the Ladner-Fischer network with the extra row) is that for any given row, there is never more than one cell in each pair of columns. These networks have low gate count. Moreover, their layout may be only half as wide, reducing the length of the horizontal wires spanning the adder. This reduces the wire capacitance, which may be a major component of delay in 64-bit and larger adders [Huang00].

Figure 10.35 shows a 3-dimensional taxonomy of the tree adders [Harris03]. If we let $L = \log_2 N$, we can describe each tree with three integers (l, f, t) in the range $[0, L-1]$. The integers specify:

- Logic Levels: $L + l$
- Fanout: $2^f + 1$
- Wiring Tracks: 2^t

The tree adders lie on the plane $l + f + t = L - 1$. 16-bit Brent-Kung, Sklansky, and Kogge-Stone represent vertices of the cube (3, 0, 0), (0, 3, 0) and (0, 0, 3), respectively. Han-Carlson, Ladner-Fischer, and Knowles lie along the diagonals. Another recently discovered family of adders lies on the plane inside the cube.

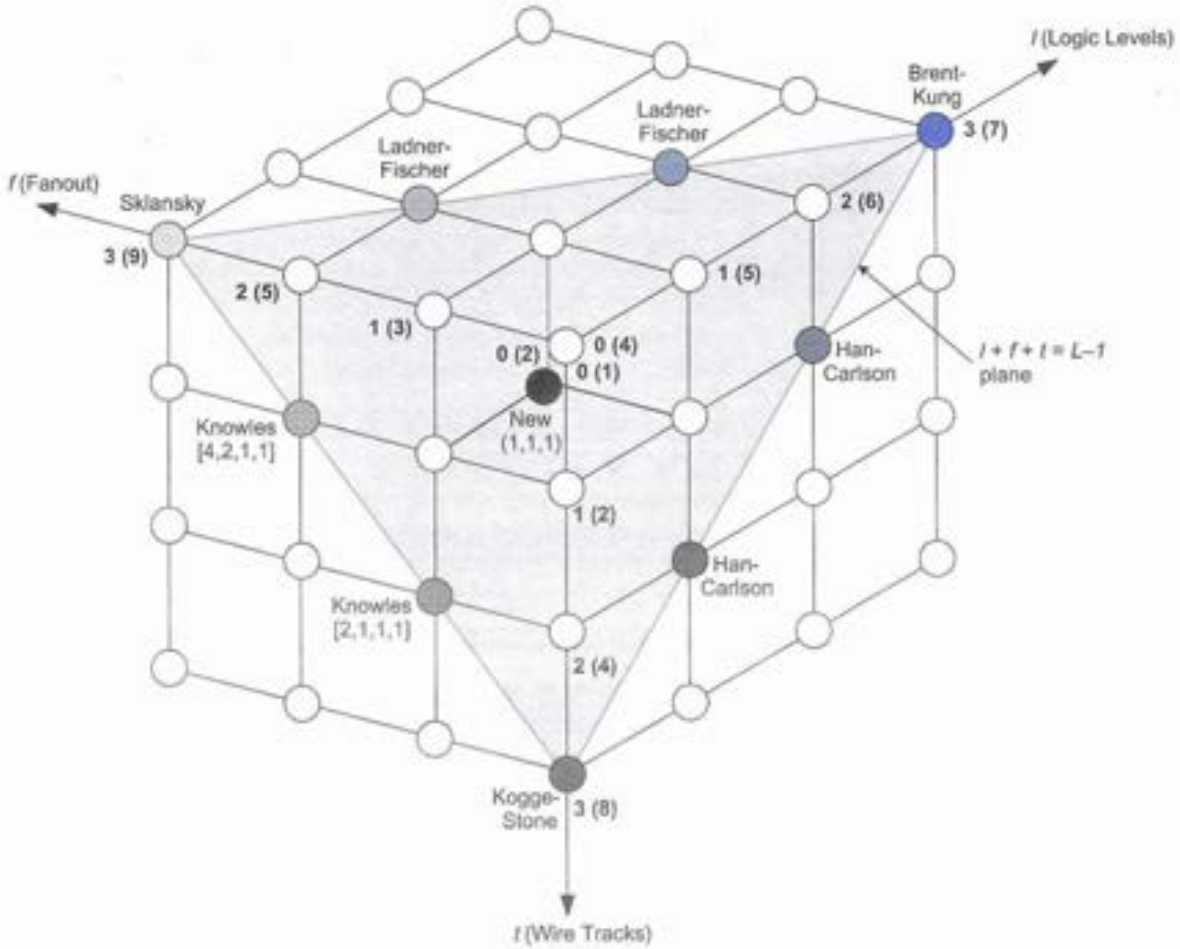


FIG 10.35 Taxonomy of prefix networks

10.2.2.9 Higher-valency Tree Adders Any of the trees described so far can combine more than two groups at each stage [Beaumont-Smith01]. The number of groups combined in each gate is called the *valency* or *radix* of the cell. For example, Figure 10.36 shows 27-bit valency-3 Brent-Kung, Sklansky, Kogge-Stone, and Han-Carlson trees. The rounded boxes mark valency-3 carry chains (that could be constructed using a Manchester carry chain, multiple-output domino gate, or several discrete gates). The trapezoids mark carry-increment operations. The higher-valency designs use fewer stages of logic, but each stage has greater delay. This tends to be a poor tradeoff in static CMOS circuits because the stage efforts become much larger than 4, but is good in domino because the logical efforts are much smaller so fewer stages are necessary.

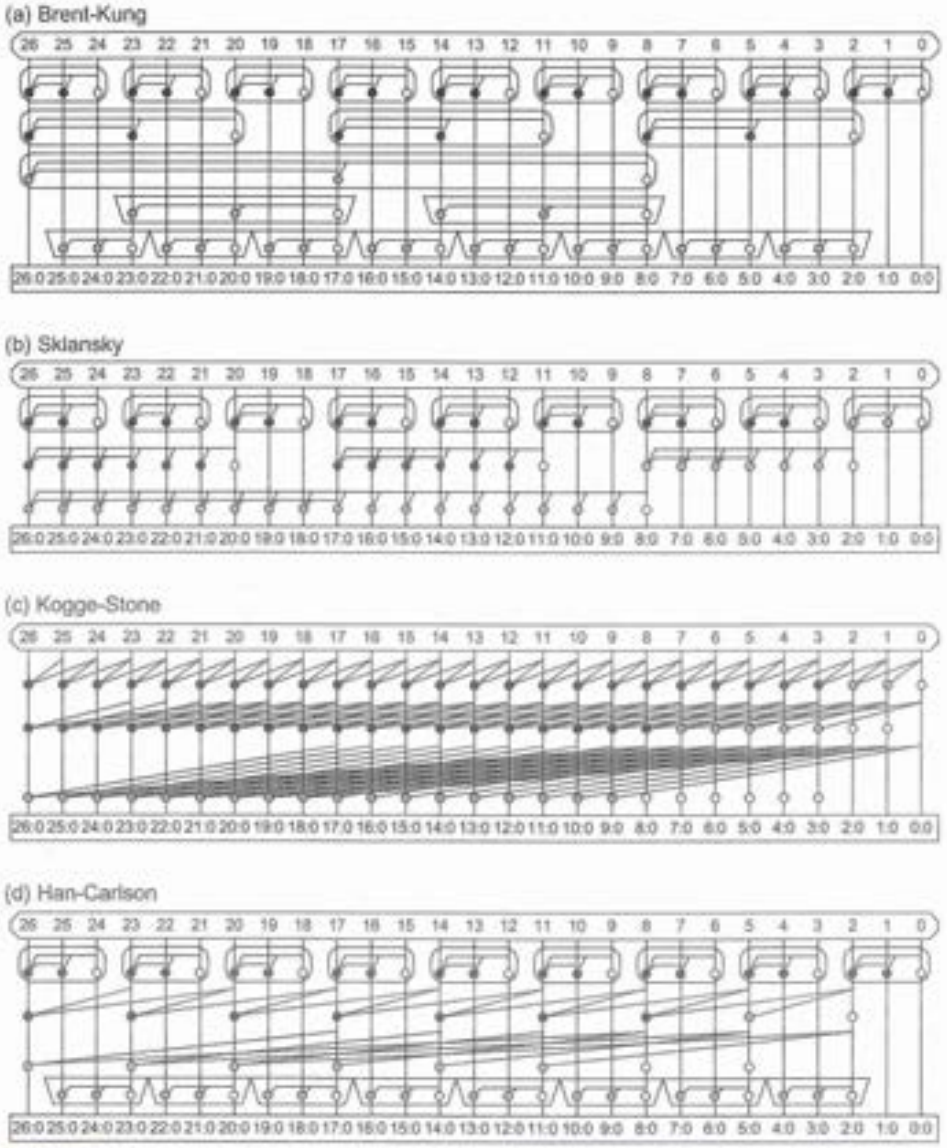


FIG 10.36 Higher-valency tree adders

Nodes with large fanouts or long wires can use buffers. The prefix trees can also be internally pipelined for extremely high-throughput operation. Some higher-valency designs combine the initial PG stage with the first level of PG merge. For example, the Ling adder described in Section 10.2.2.12 computes generate and propagate for 4-bit groups from the primary inputs in a single stage.

10.2.2.10 Hybrid Tree/Select Adders Tree adders can be merged with carry-select adders to form a hybrid with lower delay or fewer gates. The carry-select adder precomputes the sum for both carries-in of 0 and 1 for short groups (e.g., 4, 8, or 16 bits). Meanwhile, the tree adder computes the carry-in to each group. Finally, a multiplexer selects the correct sum for each group when the carry-in becomes available. The group length should be balanced such that the carry-in and precomputed sums become available at about the same time.

The *spanning-tree adder* [Lynch92] is one such hybrid based on a higher-valency Brent-Kung tree of Figure 10.36(a). Figure 10.37 shows a simple valency-3 version that precomputes sums for 3-bit groups and saves one logic level by selecting the output based on the carries into each group. The carry-out (C_{out}) is explicitly shown. Note that the least significant group requires a valency-4 gray cell to compute $G_{3:0}$, the carry-in to the second select block.

[Lynch92] describes a 56-bit spanning-tree design from the AMD AM29050 floating-point unit using valency-4 stages and 8-bit carry select groups. [Kantabutra93] and [Blackburn96] describe optimizing the spanning-tree adder by using variable-length carry-select stages and appropriately selecting transistor sizes.

A carry-select box spanning bits $i...j$ is shown in Figure 10.38(a). It uses short carry-ripple adders to precompute the sums assuming carry-in of 0 and 1 to the group, and then selects between them with a multiplexer, as shown in Figure 10.38(b). The adders can be simplified somewhat because the carry-ins are constant, as shown in Figure 10.38(c) for a 4-bit group.

The Brent-Kung adder involves more than the minimum number of logic levels. Kogge-Stone and Sklansky use fewer levels, but require many gates and either many wires or high fanout. [Mathew03] describes a 32-bit *sparse-tree adder* using a valency-2 tree similar to Sklansky to compute only the carries into each 4-bit group, as shown in Figure 10.39. This reduces the gate count and power consumption in the tree. The tree can be viewed as a (2, 2, 0) Ladner-Fischer tree with the final two tree levels and XOR replaced by the select multiplexer. The adder assumes the carry-in is 0 and does not produce a carry-out, saving one input to the least-significant gray box and eliminating the prefix logic in the four most significant columns.

These hybrid approaches are widely used in high-performance 32–64-bit higher-valency domino adders because they offer the small number of logic levels of higher-

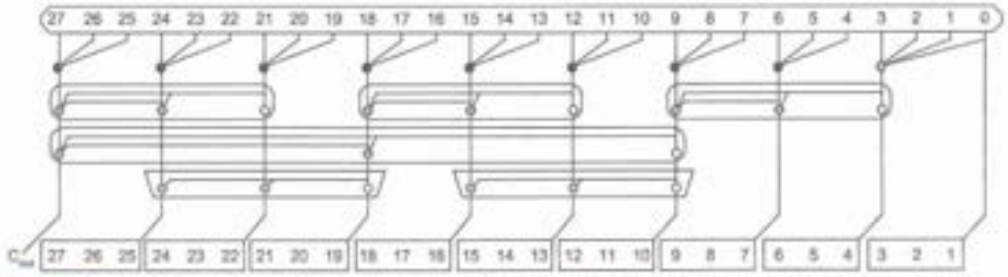


FIG 10.37 Spanning-tree adder

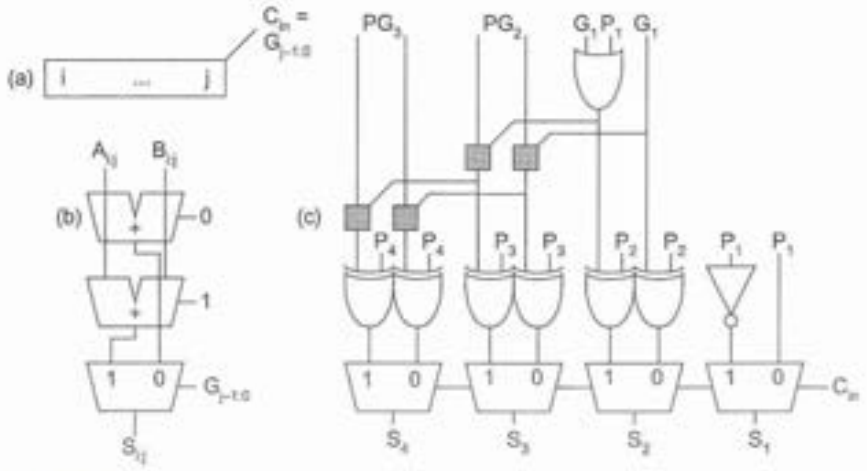


FIG 10.38 Carry-select implementation

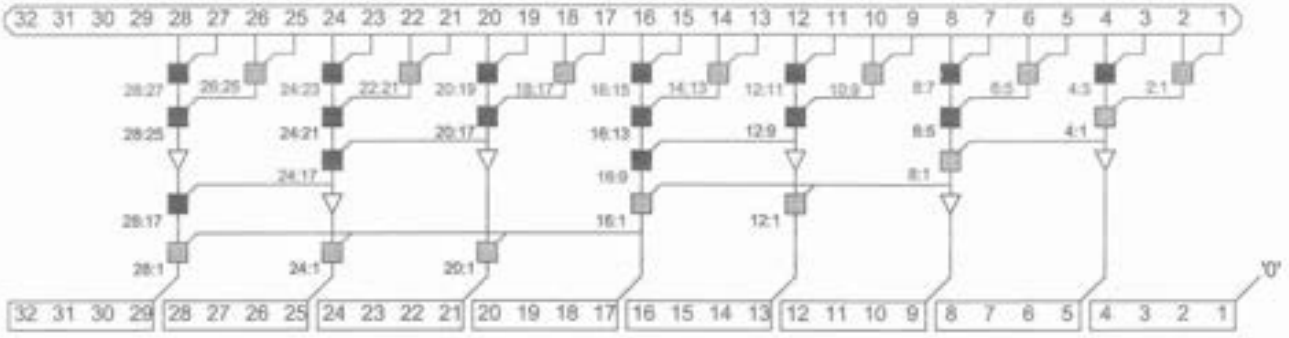


FIG 10.39 Sparse-tree adder

valency trees while reducing the gate count and power consumption in the tree. Figure 10.40 shows a 27-bit valency-3 Kogge-Stone design with carry-select on 3-bit groups. Observe how the number of gates in the tree is reduced threefold. Moreover, because the number of wires is also reduced, the extra area can be used for shielding to reduce path delay. This design can be viewed as the Han-Carlson adder of Figure 10.36(d) with the last logic level replaced by a carry-select multiplexer.

10.2.2.11 An Aside on Domino Implementation Issues Using $\overline{K} = A + B$ in place of P, all the group generate signals $G_{j:0}$ are monotonic functions of the noninverted inputs and can be computed with single-rail domino gates. However, the final sum XOR is inherently nonmonotonic and cannot be computed this way. The two common choices for designers of domino adders are to build the final sum XOR with static logic or to construct the entire adder out of dual-rail domino.

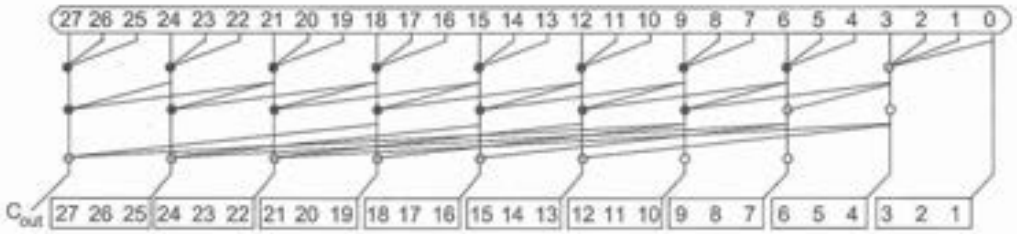


FIG 10.40 Hybrid Kogge-Stone/select adder

Domino adders with a static sum XOR produce nonmonotonic outputs that must be stabilized on a clock edge before driving subsequent domino gates. As adders are often used in self-bypass loops where the output of the adder serves as one of the inputs on the next cycle, this introduces a hard edge and the associated costs of setup time and clock skew into the critical path.

The alternative is to build a dual-rail domino sum XOR accepting monotonic true and complementary (_b and _l) versions of the carries. Producing these carries in turn requires extra hardware all the way back to the adder inputs, which also must be provided in dual-rail form. If the sum is also computed in dual-rail form, the outputs can be directly bypassed to the inputs in a skew-tolerant fashion. The drawback of such adders is the extra hardware involved in the group PG network. Again, there are two common cell designs. One is to build dual-rail group propagate and generate signals, i.e., four signals per bundle. Another is to use monotonic one-of-three hot propagate-generate-kill (PGK) signals.

The first approach uses the following logic:

$$\begin{array}{lll} G_i_b = A_i_b \cdot B_i_b & G_0_b = C_{in}_b & G_{i:j}_b = G_{i:k}_b + K_{i:k}_l \cdot G_{k-1:j}_b \\ K_i_b = A_i_l \cdot B_i_l & K_0_b = C_{in}_l & K_{i:j}_b = K_{i:k}_b + G_{i:k}_l \cdot K_{k-1:j}_b \\ G_i_l = A_i_l + B_i_l & G_0_l = 0 & G_{i:j}_l = G_{i:k}_l \cdot G_{k-1:j}_l \\ K_i_l = A_i_b + B_i_b & K_0_l = 0 & K_{i:j}_l = K_{i:k}_l \cdot K_{k-1:j}_l \end{array} \quad (10.21)$$

$$\begin{aligned} P_i_b &= A_i \oplus B_i = A_i_b \cdot B_i_l + A_i_l \cdot B_i_b = G_i_l \cdot K_i_l \\ P_i_l &= \overline{A_i \oplus B_i} = A_i_b \cdot B_i_b + A_i_l \cdot B_i_l = G_i_b + K_i_b \\ S_i_b &= G_{i-1:0} \oplus P_i = G_{i-1:0}_b \cdot P_i_l + K_{i-1:0}_b \cdot P_i_b \\ S_i_l &= \overline{G_{i-1:0} \oplus P_i} = G_{i-1:0}_b \cdot P_i_b + K_{i-1:0}_b \cdot P_i_l \end{aligned} \quad (10.22)$$

Observe that the group generate _b and _l and kill _b and _l signals are not truly complementary; they are sometimes called *pseudo-complements* [Wang93]. They take advantage of the symmetry of the addition function so that the same type of gate can be reused. It is left to the reader to recursively verify that $G_{i-1:0}_b$ and $K_{i-1:0}_b$ are the true and complementary versions of the carries into bit i. Schematics of each gate are shown in Figure 10.41(a).

The 1-of-3 hot version uses less logic:

$$
\begin{aligned}
G_i &= A_i_b \cdot B_i_b & G_0 &= C_{in}_b & G_{i:j} &= G_{i:k} + P_{i:k} \cdot G_{k-1:j} \\
P_i &= A_i_b \cdot B_i_l + A_i_l \cdot B_i_b = A \oplus B & P_0 &= 0 & P_{i:j} &= P_{i:k} \cdot P_{k-1:j} \\
K_i &= A_i_l \cdot B_i_l & K_0 &= C_{in}_l & K_{i:j} &= K_{i:k} + P_{i:k} \cdot K_{k-1:j}
\end{aligned}
\tag{10.23}
$$

$$
\begin{aligned}
P_i' &= G_i + K_i \\
S_i_b &= G_{i-1:0} \cdot P_i' + K_{i-1:0} \cdot P_i \\
S_i_l &= K_{i-1:0} \cdot P_i' + G_{i-1:0}_b \cdot P_i
\end{aligned}
\tag{10.24}
$$

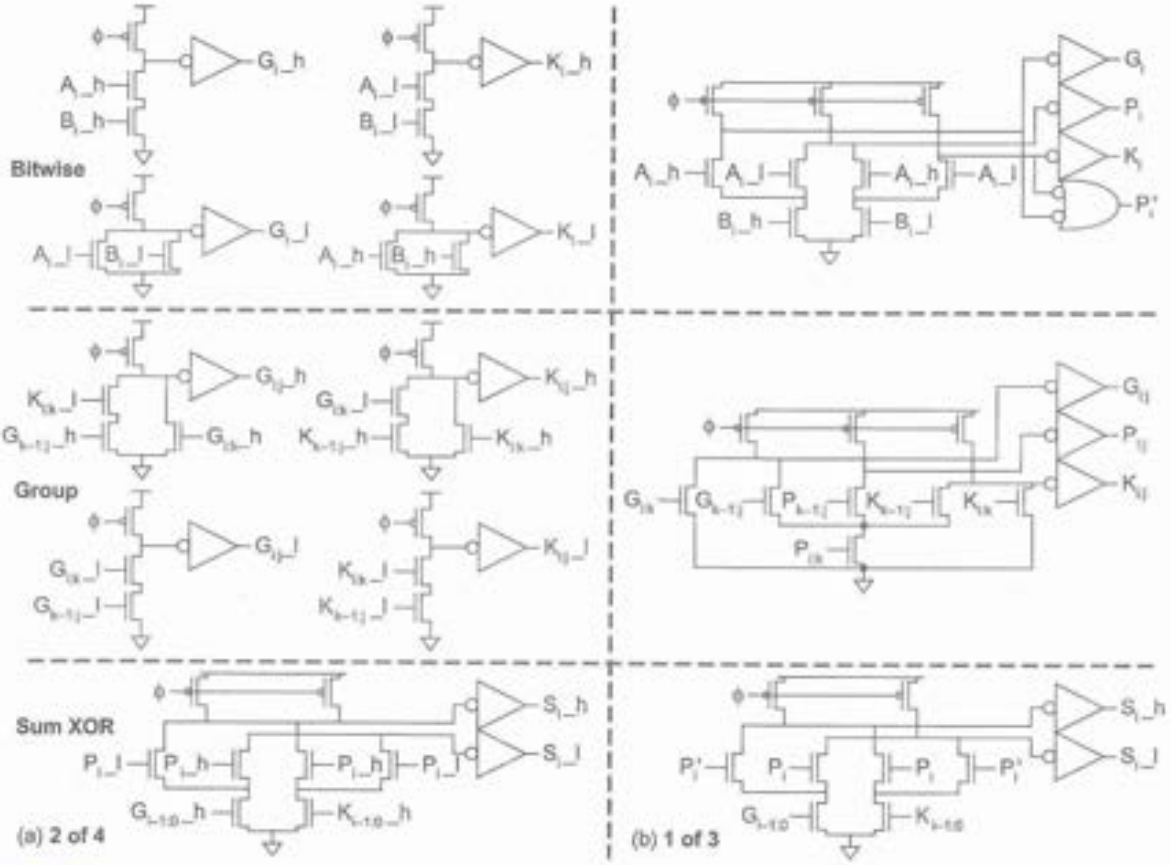


FIG 10.41 Domino adder circuit components

The approach gets its name because exactly one of the three signals P, G, or K is true for any group. The complementary propagate signal $P' = G + K$ is required for the final sum XOR, but nowhere earlier. Again notice that the G and K functions are identical, sim-

plifying design and layout. The group kill prefixes are the complements of the group generates ($G_{i-1:0} = \overline{K_{i-1:0}}$); this is used in EQ (10.24) to reduce the loading on each signal in the sum XOR. The 1-of-n hot technique can be useful for other domino applications, such as multiplexer select signals and shifter control signals; it also reduces switching activity and power consumption. Figure 10.41(b) shows how transistors can be shared between gates.

Manchester carry chains can also generate both polarities of carries. Figure 10.42 shows how the same type of carry chain can be used for both _h and _l carries as well as to find the group propagate signal using 1-of-3 hot encoding.

The domino gates are fast, but power-hungry. Figure 10.43 shows an interesting static version of the same circuit using triple-rail "push-pull" CPL to obtain nearly the same speed as domino at much lower power. The UltraSparc VI uses this cell (without *C1* and *C2* carry inverters) for a low-power ALU [Chillarige03].

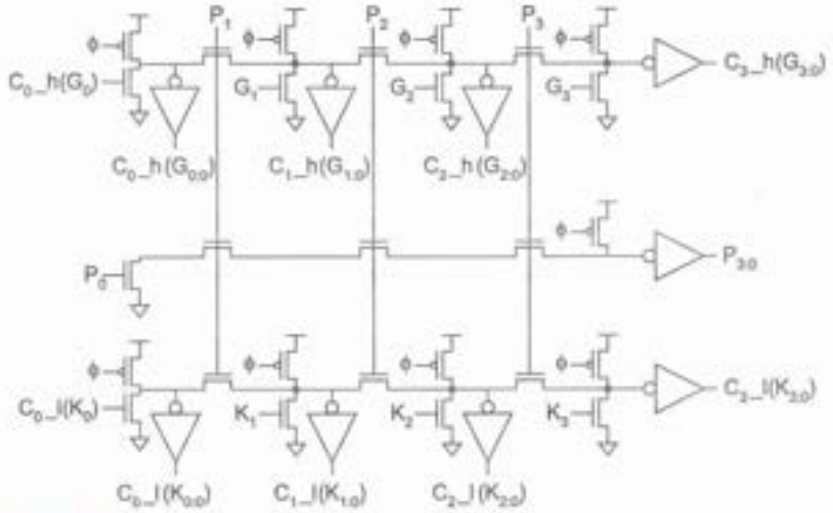


FIG. 10.42 1-of-3 hot domino Manchester carry chain

10.2.2.12 Ling and Naffziger Adders A 64-bit tree adder using valency-2 cells requires at least six levels of group PG logic plus the initial bitwise PG logic and final XOR for a total of eight levels of logic. Using valency-4 cells reduces this to three levels of group PG logic and a total of five levels of logic. The bitwise PG logic can be merged into the first stage of group PG logic, but this leads to an excessively complex gate with tall transistor stacks for valency-4. The Ling adder [Ling81] defines a *pseudo-generate* (sometimes called *pseudo-carry*) signal that is simple enough that the bitwise PG logic can be merged into the initial valency-4 stage, resulting in only four levels of logic on the critical path. The actual carries are produced later in the path with an OR operation. The Ling adder was originally developed for bipolar transistors where a wired-OR contributes negligible delay. Hewlett-Packard adapted it to domino logic in which the OR operation is just another transistor in parallel; this Naffziger adder [Naffziger96, Naffziger98] has been widely used

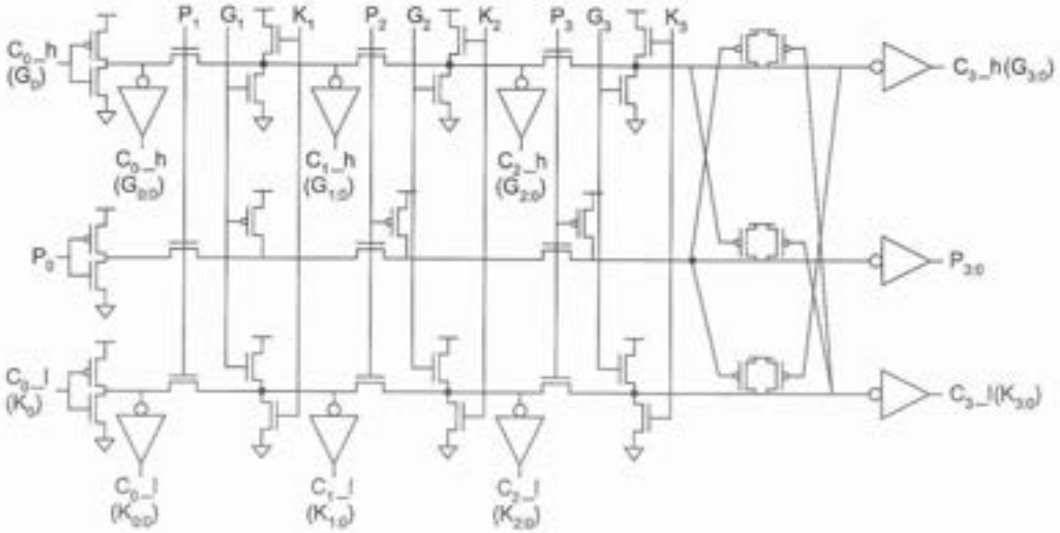


FIG 10.43 1-of-3 hot CPL Manchester carry chain

in Hewlett-Packard 64-bit microprocessors and on the Itanium 2 [Fetzer02]. The 64-bit Naffziger adder has a delay of about 7 FO4 inverters, making it one of the fastest published designs.

Define $P_i = A_i + B_i$ so that it can be computed with two parallel transistors in a domino circuit. (This differs from our convention of XOR for propagate, but is consistent with Naffziger's paper.) In an ordinary valency-4 tree adder, the first level of group PG cells computes 4-bit group generate using

$$G_{3:0} = G_3 + P_3\left(G_2 + P_2\left(G_1 + P_1G_0\right)\right) \tag{10.25}$$

If we were to compute $G_{3:0}$ directly from $A_3 \ldots A_0$ and $B_3 \ldots B_0$, the gate would be complicated and quite slow. Instead, let us define a group *pseudo-generate* signal $H_{3:0}$ in which the P_3 term is removed.

$$H_{3:0} = G_3 + \left(G_2 + P_2\left(G_1 + P_1G_0\right)\right) = G_3 + G_{2:0} = G_{3:0} + G_{2:0} \tag{10.26}$$

We will also use a group *pseudo-propagate* signal I that is a shifted copy of the actual group propagate signal. In general, we can define pseudo-generate and pseudo-propagate signals H and I as

$$\begin{aligned} H_{i:j} &= G_{i:j} + G_{i-1:j} \\ I_{i:j} &= P_{i-1:j-1} \end{aligned} \tag{10.27}$$

These signals are recursively combined into bigger groups (for $i \geq k > j$) in the same fashion as actual group generate and propagate signals (EQ (10.4)).

$$H_{i:j} = H_{i:k} + I_{i:k} H_{k-1:j}$$
$$I_{i:j} = I_{i:k} I_{k-1:j} \tag{10.28}$$

The actual group generate signal is formed from pseudo-generate using

$$\begin{aligned} P_{i:i} H_{i:j} &= P_{i:i}\left(G_{i:j} + G_{i-1:j}\right) \\ &= G_{i:j} + P_{i:i} G_{i-1:j} \\ &= G_{i:j} \end{aligned} \tag{10.29}$$

It is left to the reader (Exercise 10.11) to prove EQ (10.28) from EQ (10.27) and EQ (10.29). Now that we can compute the $G_{i-1:0}$ prefixes, all that remains is to compute the final sums with the usual XOR.

The 64-bit Naffziger adder combines these Ling equations with a hybrid tree/select adder (from Section 10.2.2.10), as shown in Figure 10.44. It uses the Ling equations to find the pseudo-generate signal into each 16-bit block in three levels of compound valency-4 dual-rail domino gates. Meanwhile, it computes the generate prefixes for each bit in a 16-bit block assuming pseudo-carry in signals to the block of both 0 and 1. The prefix logic consists of a short carry-ripple and Manchester carry chain. Finally, the adder selects the appropriate generate given the pseudo-carries and performs an XOR to find the sum. By doing the select based on the pseudo-carry instead of the actual carry, it saves one stage of logic.

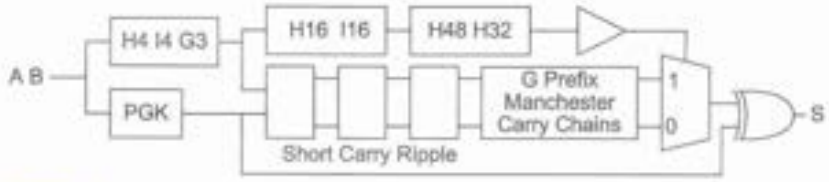


FIG 10.44 Naffziger adder

The Naffziger adder uses dual-rail domino throughout. The equations below describe only the true (_t) logic. The complementary (_f) logic uses identical gates on complementary inputs, as in EQ (10.21). As usual, let $A_0 = B_0 = C_{in}$.

The (H4 I4 G3) block computes the H and I signals for each 4-bit group using EQ (10.30) ($k = 0 \ldots 15$). As a byproduct, we find G for 3-bit groups, which we will use later when computing the generate prefixes. Figure 10.45 shows a domino gate that calculates G and H. It uses a clever combined static NOR/INV gate to produce both outputs. We use an nMOS secondary precharge transistor to prevent charge sharing on the heavily loaded internal node.

$$
\begin{aligned}
G_{4i+3:4i+1} &= G_{4i+3} + P_{4i+3}\left(G_{4i+2} + P_{4i+2}G_{4i+1}\right) \\
&= A_{4i+3}B_{4i+3} + \left(A_{4i+3} + B_{4i+3}\right)\left(A_{4i+2}B_{4i+2} + \left(A_{4i+2} + B_{4i+2}\right)A_{4i+1}B_{4i+1}\right) \\
H_{4i+4:4i+1} &= G_{4i+4} + \left(G_{4i+3} + P_{4i+3}\left(G_{4i+2} + P_{4i+2}G_{4i+1}\right)\right) \\
&= A_{4i+4}B_{4i+4} + G_{4i+3:4i+1} \\
I_{4i+4:4i+1} &= \left(A_{4i+3} + B_{4i+3}\right)\left(A_{4i+2} + B_{4i+2}\right)\left(A_{4i+1} + B_{4i+1}\right)\left(A_{4i} + B_{4i}\right)
\end{aligned}
\tag{10.30}
$$

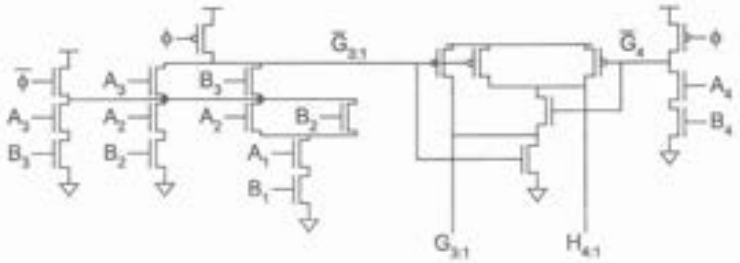


FIG 10.45 GH gate

At the same time, the 1-of-3 hot P_i, G_i, and K_i signals are computed in the PGK block for each bit (i = 1…64) using EQ (10.23). They will also be used later.

The next step is to compute the H and I signals for each 16-bit group (H16 I16 block). For the least significant group, the carry-in is incorporated into the pseudo-generate using $I_{4:1}$ so no pseudo-propagate is needed (EQ (10.31)). For the other three groups, both H and I are found using valency-4 black cells (EQ (10.32), k = 1…3).

$$
H_{16:1} = H_{16:13} + I_{16:13}\left(H_{12:9} + I_{12:9}\left(H_{8:5} + I_{8:5}\left(H_{4:1} + I_{4:1}\right)\right)\right) \tag{10.31}
$$

$$
\begin{aligned}
H_{16k+16:16k+1} &= H_{16k+16:16k+13} + \\
&\quad I_{16k+16:16k+13}\left(H_{16k+12:16k+9} + I_{16k+12:16k+9}\left(H_{16k+8:16k+5} + I_{16k+8:16k+5}H_{16k+4:16k+1}\right)\right) \\
I_{16k+16:16k+1} &= I_{16k+16:16k+13}I_{16k+12:16k+9}I_{16k+8:16k+5}I_{16k+4:16k+1}
\end{aligned}
\tag{10.32}
$$

Now we can find the pseudo-carry signals into each 16-bit block using EQ (10.33) (H48 H32 block).

$$
\begin{aligned}
H_{0:1} &= C_{in} \\
H_{16:1} &= H_{16:1} \\
H_{32:1} &= H_{32:17} + I_{32:17}H_{16:1} \\
H_{48:1} &= H_{48:33} + I_{48:33}\left(H_{32:17} + I_{32:17}H_{16:1}\right)
\end{aligned}
\tag{10.33}
$$

Meanwhile, we need to compute the generate signals for each bit within the 16-bit block assuming pseudo-carries of both 0 and 1 into the 16-bit block. This is done in two steps. First, we use a series of fast ripple gates to find the generate out of the third, seventh, eleventh, and fifteenth bits within the block using EQ (10.34) ($k = 0\ldots3$). The critical path goes through the 11th bit. The signal $G^0_{16k+:0}$ indicates whether the group spanning bits $16k + 1{:}0$ will generate a carry if the pseudo-carry $H_{16k:1}$ into the kth 16-bit block is 0. Observe how the 1-bit generate signals $G_{16k+4,8,12}$ can be folded in without adding to the number of series transistors in the domino gates.

$$
\begin{aligned}
G^0_{16k-1:0} &= 0\\
G^1_{16k-1:0} &= 1\\
G^0_{16k+3:0} &= G_{16k+3:16k+1}\\
G^1_{16k+3:0} &= G_{16k+3:16k+1} + I_{16k+4:16k+1}\\
G^0_{16k+7:0} &= G_{16k+7:16k+5} + I_{16k+8:16k+5}\left(G^0_{16k+3:0} + G_{16k+4}\right)\\
G^1_{16k+7:0} &= G_{16k+7:16k+5} + I_{16k+8:16k+5}\left(G^1_{16k+3:0} + G_{16k+4}\right)\\
G^0_{16k+11:0} &= G_{16k+11:16k+9} + I_{16k+12:16k+9}\left(G^0_{16k+7:0} + G_{16k+8}\right)\\
G^1_{16k+11:0} &= G_{16k+11:16k+9} + I_{16k+12:16k+9}\left(G^1_{16k+7:0} + G_{16k+8}\right)\\
G^0_{16k+15:0} &= G_{16k+15:16k+13} + I_{16k+16:16k+13}\left(G^0_{16k+11:0} + G_{16k+12}\right)\\
G^1_{16k+15:0} &= G_{16k+15:16k+13} + I_{16k+16:16k+13}\left(G^1_{16k+11:0} + G_{16k+12}\right)
\end{aligned}
\tag{10.34}
$$

Then pairs of Manchester carry chains compute the generate signal into each bit for both possibilities of pseudo-carries into the kth 16-bit block ($k = 0\ldots3$). Four pairs ($n = 0\ldots3$) are required for each block. The outputs of the carry chains are given in EQ (10.35). For the least significant chains ($n = 0$), the $G^{0/1}_{16k+1:0}$ signals are constants and can be optimized away.

$$
\begin{aligned}
G^0_{16k+4n:0} &= G_{16k+4n} + P_{16k+4n}G^0_{16k+4n-1:0}\\
G^1_{16k+4n:0} &= G_{16k+4n} + P_{16k+4n}G^1_{16k+4n-1:0}\\
G^0_{16k+4n+1:0} &= G_{16k+4n+1} + P_{16k+4n+1}\left(G_{16k+4n} + P_{16k+4n}G^0_{16k+4n-1:0}\right)\\
G^1_{16k+4n+1:0} &= G_{16k+4n+1} + P_{16k+4n+1}\left(G_{16k+4n} + P_{16k+4n}G^1_{16k+4n-1:0}\right)\\
G^0_{16k+4n+2:0} &= G_{16k+4n+2} + P_{16k+4n+2}\left(G_{16k+4n+1} + P_{16k+4n+1}\left(G_{16k+4n} + P_{16k+4n}G^0_{16k+4n-1:0}\right)\right)\\
G^1_{16k+4n+2:0} &= G_{16k+4n+2} + P_{16k+4n+2}\left(G_{16k+4n+1} + P_{16k+4n+1}\left(G_{16k+4n} + P_{16k+4n}G^1_{16k+4n-1:0}\right)\right)
\end{aligned}
\tag{10.35}
$$

Finally, we use a carry select multiplexer to choose the appropriate carry into each bit and an XOR to find the sum. These two functions can be merged into another compound domino gate using EQ (10.36) for blocks $k = 0 \ldots 3$ and bits $i = 1 \ldots 16$ within each block. Figure 10.46 shows a dual-rail domino implementation of the sum function taking advantage of the fact that $\overline{P}_i = G_i + K_i$. The H signals drive a large fanout and thus are first buffered with a pair of inverters.

$$S_{16k+i} = P_{16k+i} \oplus \left(\overline{H}_{16k:1} G^0_{16k+i-1:0} + H_{16k:1} G^1_{16k+i-1:0}\right) \tag{10.36}$$

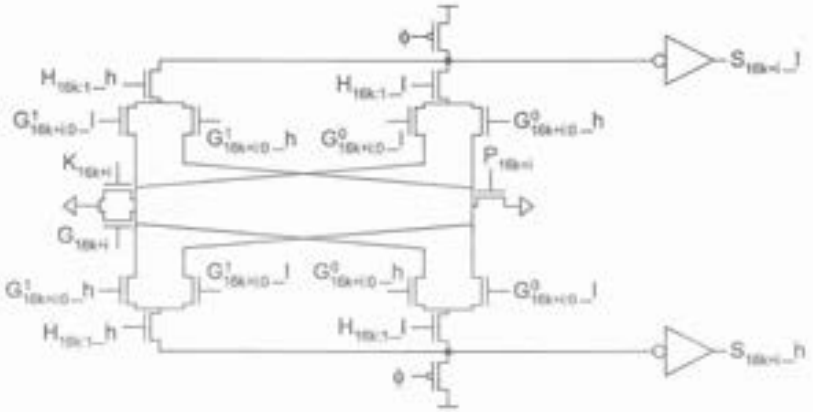


FIG 10.46 Dual-rail sum gate

10.2.2.13 Summary Having examined so many adders, you probably want to know which adder should be used in which application. Table 10.3 compares the various adder architectures that have been illustrated with valency-2 prefix networks. The category "logic levels" gives the number of AND-OR gates in the critical path, excluding the initial PG logic and final XOR. Of course, the delay depends on the fanout and wire loads as well as the number of logic levels. The category "cells" refers to the approximate number of gray and black cells in the network. Carry-lookahead is not shown because it uses higher-valency cells. Carry-select is also not shown because it is larger than carry-increment for the same performance.

In general, carry-ripple adders should be used when they meet timing constraints because they are compact and easy to build. When faster adders are required, carry-increment and carry-skip architectures work well, particularly for 8–16 bit lengths. Hybrids combining these techniques are also popular. At word lengths of 32 and especially 64 bits, tree adders are distinctly faster.

Table 10.3 Comparison of adder architectures

Architecture	Classification	Logic Levels	Max Fanout	Tracks	Cells
Carry-Ripple		$N-1$	1	1	N
Carry-Skip ($n = 4$)		$N/4 + 5$	2	1	$1.25N$
Carry-Increment ($n = 4$)		$N/4 + 2$	4	1	$2N$
Carry-Increment (variable group)		$\sqrt{2N}$	$\sqrt{2N}$	1	$2N$
Brent-Kung	$(L-1, 0, 0)$	$2\log_2 N - 1$	2	1	$2N$
Sklansky	$(0, L-1, 0)$	$\log_2 N$	$N/2 + 1$	1	$0.5\, N \log_2 N$
Kogge-Stone	$(0, 0, L-1)$	$\log_2 N$	2	$N/2$	$N \log_2 N$
Han-Carlson	$(1, 0, L-2)$	$\log_2 N + 1$	2	$N/4$	$0.5\, N \log_2 N$
Ladner Fischer ($l = 1$)	$(1, L-2, 0)$	$\log_2 N + 1$	$N/4 + 1$	1	$0.25\, N \log_2 N$
Knowles [2,1,...,1]	$(0, 1, L-2)$	$\log_2 N$	3	$N/4$	$N \log_2 N$

Good logic synthesis tools automatically map the "+" operator onto an appropriate adder to meet timing constraints while minimizing area. For example, the Synopsys DesignWare libraries contain carry-ripple adders, carry-select adders, carry-lookahead adders, and a variety of prefix adders. Figure 10.47 shows the results of synthesizing 32-bit and 64-bit adders under different timing constraints. As the latency decreases, synthesis selects more elaborate adders with greater area. The results are for a 0.18 μm commercial cell library with an FO4 inverter delay of (a rather slow) 89 ps in the TTTT corner and the area includes estimated interconnect as well as gates. The fastest designs use tree adders and achieve remarkably fast (prelayout) delays of 7.0 and 8.5 FO4 for 32-bit and 64-bit adders, respectively, by creating nonuniform designs with side loads carefully buffered off the critical path. The carry-select adders achieve an interesting area/delay tradeoff by using carry-ripple for the lower 3/4 of the bits and carry-select only on the upper 1/4. The results will be somewhat slower when true layout parasitics are included. In comparison, a good handcrafted 64-bit domino adder has a delay of about 7 FO4 and an area of under 3 $M\lambda^2$ [Naffziger96].

Other surveys of addition include [Ercegovac04, Parhami00, Koren02, and Zimmermann97b].

10.2.3 Adder Variants

We conclude the study of addition by examining subtraction and addition of multiple inputs. Incrementers are covered in Section 10.10.

10.2.3.1 Subtraction An N-bit subtracter uses the 2's complement relationship

$$A - B = A + \overline{B} + 1 \tag{10.37}$$

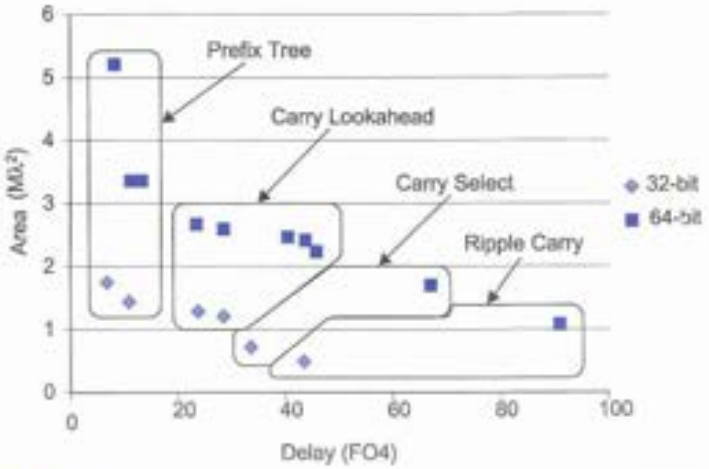


FIG 10.47 Area vs. delay of synthesized adders

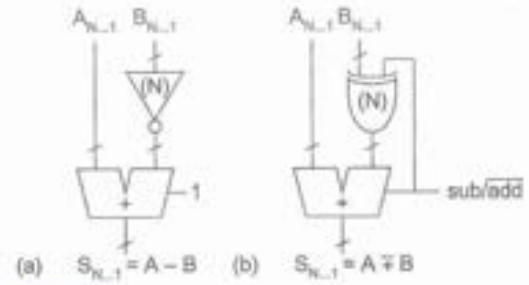


FIG 10.48 Subtracters

This involves inverting one operand to an N-bit CPA and adding 1 via the carry input, as shown in Figure 10.48(a). An adder/subtracter uses XOR gates to conditionally invert B, as shown in Figure 10.48(b). In prefix adders, the XOR gates on the B inputs are sometimes merged into the bitwise PG circuitry.

10.2.3.2 Multiple-input Addition The most obvious method of adding k N-bit words is with k–1 cascaded CPAs as illustrated in Figure 10.49(a) for 0001 + 0111 + 1101 + 0010, but this consumes a large amount of hardware and is slow. A better technique is to note that a full adder sums three inputs of unit weight and produces a sum output of unit weight and a carry output of double weight. If N full adders are used in parallel, they can accept three N-bit input words X_{N-1}, Y_{N-1}, and Z_{N-1}, and produce two N-bit output words S_{N-1} and C_{N-1}, as shown in Figure 10.49(b). The results correspond to the sums and carries-out of each adder. This is called *carry-save redundant format* because the carry outputs are preserved rather than propagated along the adder. The full adders in this application are sometimes called *(3,2) counters* or *carry-save adders* (CSA) because they accept three inputs and produce two outputs in carry-save form. When the carry word C is shifted left by one position (because it has double weight) and added to the sum word S with an ordinary CPA, the result is $X + Y + Z$. Alternatively, a fourth input word can be added to the carry-save redundant result with another row of CSAs, again resulting in a carry-save redundant result. Such carry-save addition of four numbers is illustrated in Figure 10.49(c), where the underscores in the carry outputs serve as reminders that the carries must be shifted left

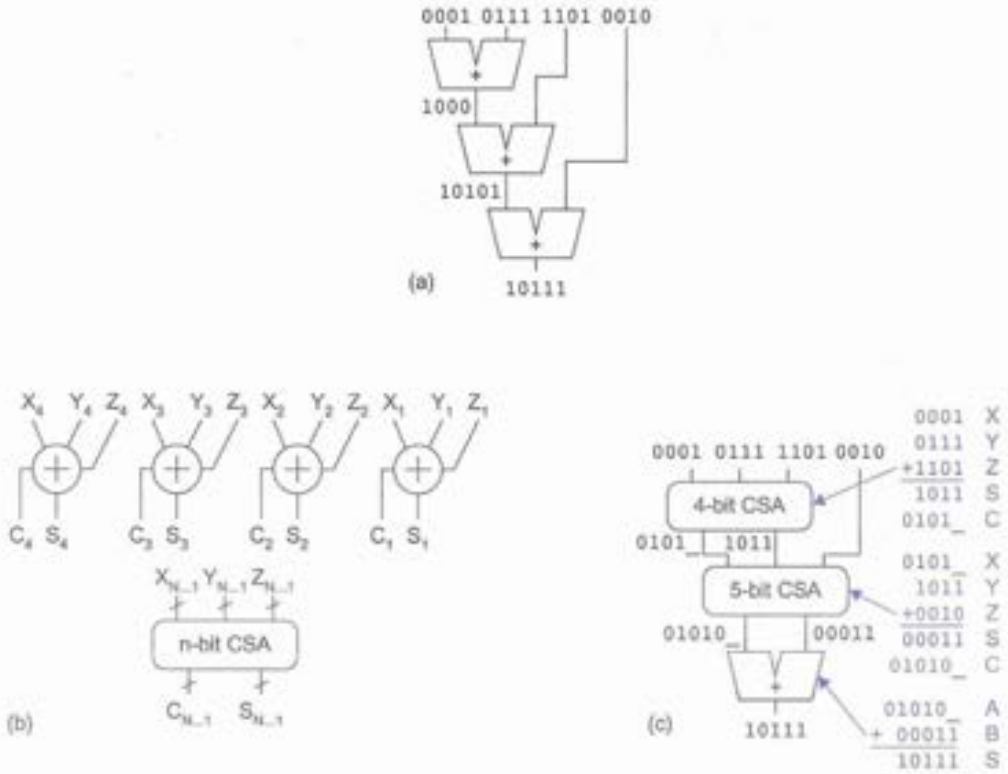


FIG 10.49 Multiple-input adders

one column on account of their greater weight. In general, k numbers can be summed with k–2 CSAs and only one CPA. This approach will be exploited in Section 10.9 to rapidly add many partial products in a multiplier. The technique dates back to von Neumann's early computer [Burks46].

10.3 One/Zero Detectors

Detecting all 1's or 0's on wide N-bit words requires large fan-in AND or NOR gates. Recall that by DeMorgan's law, AND, OR, NAND, and NOR are fundamentally the same operation except for possible inversions of the inputs and/or outputs. You can build a tree of AND gates, as shown in Figure 10.50(a). Here, alternate NAND and NOR gates have been used. The path has log N stages. In general, the minimum logical effort is achieved with a tree alternating NAND gates and inverters and the path logical effort is:

$$G_{and}(N) = \left(\frac{4}{3}\right)^{\log_2 N} = N^{\log_2 \frac{4}{3}} = N^{0.415} \tag{10.38}$$

A rough estimate of the path delay driving a path electrical effort of H using static CMOS gates is:

$$D \approx (\log_4 F)\, t_{FO4} = (\log_4 H + 0.415 \log_4 N)\, t_{FO4} \tag{10.39}$$

where t_{FO4} is the fanout-of-4 inverter delay.

If the word being checked has a natural skew in the arrival time of the outputs (such as at the output of a ripple adder), the designer might consider mimicking the adder delay in the detector, as shown in Figure 10.50(b). Here the delay from the last changing output A_7 is a single delay.

Another fast detector uses a pseudo-nMOS or dynamic NOR structure to perform the "wired-OR," as shown in Figure 10.50(c). This works well for words up to about 16 bits; for larger words, the gates can be split into 8–16-bit chunks to reduce the parasitic delay and avoid problems with subthreshold leakage.

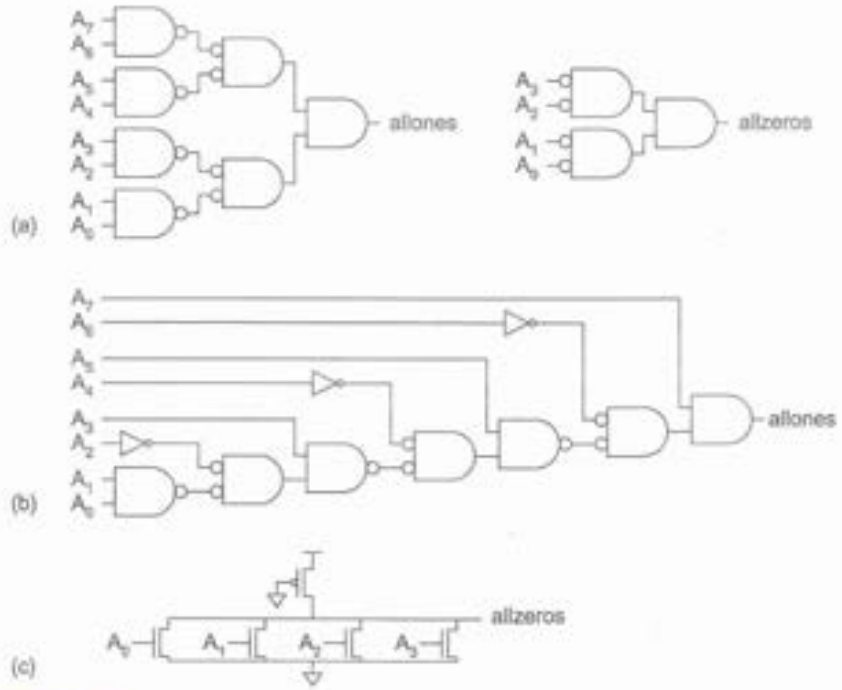


FIG 10.50 One/zero detectors

10.4 Comparators

10.4.1 Magnitude Comparator

A *magnitude comparator* determines the larger of two binary numbers. To compare two unsigned numbers A and B, compute $B - A = B + \overline{A} + 1$. If there is a carry-out, $A \leq B$. A zero detector indicates that the numbers are equal. Figure 10.51 shows a 4-bit unsigned comparator built from a carry-ripple adder and 2's complementer. The relative magnitude is determined from the carry-out (C) and zero (Z) signals according to Table 10.4. For wider inputs, any of the faster adder architectures can be used.

Comparing signed 2's complement numbers is slightly more complicated because of the possibility of overflow when subtracting two numbers with different signs. Instead of simply examining the carry-out, we must determine if the result is negative (N, indicated by the most significant bit of the result) and if it overflows the range of possible signed numbers. The overflow signal V is true if the inputs had different signs (most significant bits) and the output sign is different than the sign of B. The actual sign of the difference $B - A$ is $(N \oplus V)$ because overflow flips the sign. If this sign is negative, we know $A > B$. Again, the other relations can be derived from this corrected sign and the Z signal.

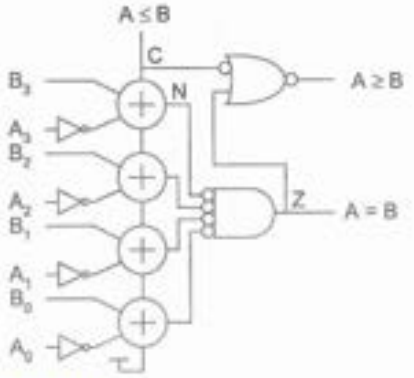


FIG 10.51 Unsigned magnitude comparator

Table 10.4 Magnitude comparison

Relation	Unsigned Comparison	Signed Comparison
$A = B$	Z	Z
$A \neq B$	$\overline{Z}$	$\overline{Z}$
$A < B$	$\overline{C} + Z$	$\overline{(N \oplus V) + Z}$
$A > B$	$\overline{C}$	$(N \oplus V)$
$A \leq B$	C	$(\overline{N \oplus V})$
$A \geq B$	$\overline{C} + Z$	$(N \oplus V) + Z$

10.4.2 Equality Comparator

An *equality comparator* determines if $(A = B)$. This can be done more simply and rapidly with XNOR gates and a 1's detector, shown in Figure 10.52. Again, this could be done with a pseudo-nMOS or dynamic gate in which the XOR functions were combined with a "wired-OR."

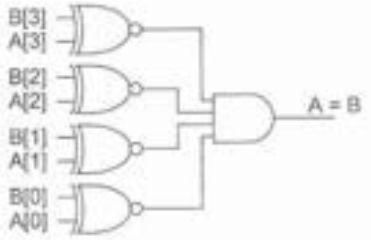


FIG 10.52 Equality comparator

10.4.3 $K = A + B$ Comparator

Sometimes it is necessary to determine if $(A + B = K)$. For example, the sum-addressed memory [Heald98] described in Section 11.2.2.3 contains a decoder that must match against the sum of two numbers, such as a register base address and an immediate offset. Remarkably, this comparison can be done faster than computing $A + B$ because no carry propagation is necessary. The key is that if you know A and B, you also know what the carry into each bit must be if $K = A + B$ [Cortadella92]. Therefore, you only need to check adjacent pairs of bits to verify that the previous bit produces the carry required by the current bit, and then use a 1's detector to check that the condition is true for all N pairs. Specifically, if $K = A + B$, Table 10.5 lists what the carry-in c_{i-1} must have been for this to be true and what the carry-out c_i will be for each bit position i.

Table 10.5 Required and generated carries if $K = A + B$

A_i	B_i	K_i	c_{i-1} (required)	c_i (produced)
0	0	0	0	0
0	0	1	1	0
0	1	0	1	1
0	1	1	0	0
1	0	0	1	1
1	0	1	0	0
1	1	0	0	1
1	1	1	1	1

From this table, you can see that the required c_{i-1} for bit i is:

$$c_{i-1} = A_i \oplus B_i \oplus K_i \tag{10.40}$$

and the c_{i-1} produced by bit i–1 is:

$$c_{i-1} = (A_{i-1} \oplus B_{i-1})\overline{K}_{i-1} + A_{i-1} \cdot B_{i-1} \tag{10.41}$$

Figure 10.53 shows one bitslice of a circuit to perform this operation. The XNOR gate is used to make sure that the required carry matches the produced carry at each bit position; then the AND gate checks that the condition is satisfied for all bits.

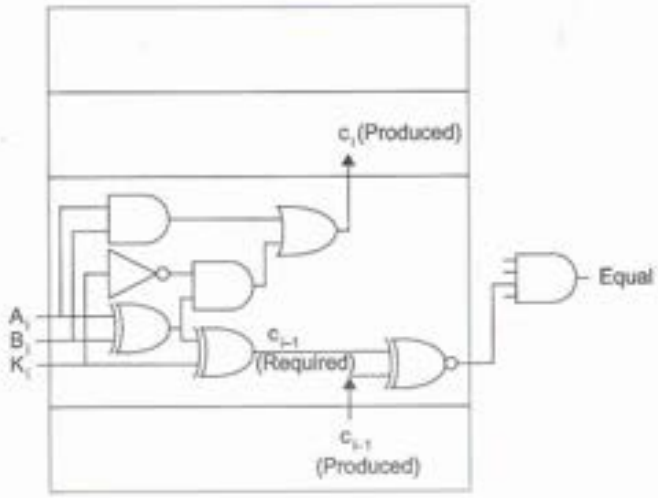


FIG 10.53 A + B = K comparator

10.5 Counters

Two commonly used types of counters are *binary counters* and *linear-feedback shift registers*. An N-bit binary counter sequences through 2^N outputs in binary order. It has a minimum cycle time that increases with N. An N-bit linear-feedback shift register sequences through up to 2^N–1 outputs in pseudo-random order. It has a short minimum cycle time independent of N, so it is useful for extremely fast counters as well as pseudo-random number generation.

In general, divide-by-M counters ($M < 2^N$) can be built using an ordinary N-bit counter and circuitry to reset the counter upon reaching M. M can be a programmable input if an equality comparator is used.

10.5.1 Binary Counters

The simplest binary counter is the *asynchronous ripple-carry counter*, shown in Figure 10.54. It is composed of N registers connected in toggle configuration, where the falling transition of each register clocks the subsequent register. Therefore, the delay can be quite long. It has no reset signal, making it extremely difficult to test. In general, asynchronous circuits introduce a whole assortment of problems, so the ripple-carry counter is shown mainly for historical interest and would not be recommended for commercial designs.

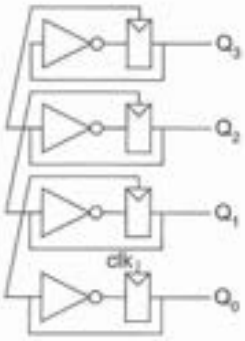


FIG 10.54 Asynchronous ripple-carry counter

A general *synchronous up/down counter* is shown in Figure 10.55(a). It uses a resettable register and full adder for each bit position. The cycle time is limited by the ripple-carry delay and can be improved using any of the faster adder techniques discussed in Section 10.2.2. If only an *up counter* (also called an *incrementer*) is required, the full adder degenerates into a half adder, shown in Figure 10.55(b). Including an input multiplexer allows the counter to load an initialization value. A clock enable is also often provided to each register for conditional counting.

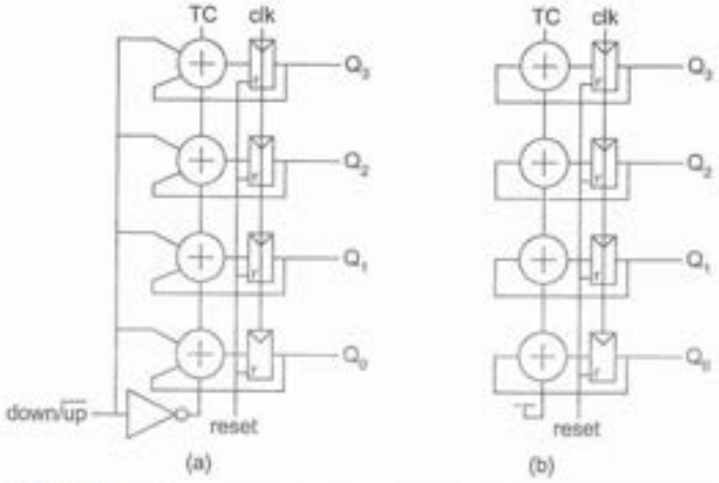


FIG 10.55 Synchronous counters

10.5.2 Linear-feedback Shift Registers

A linear-feedback shift register (LFSR) consists of N registers connected together as a shift register. The input to the shift register comes from the XOR of particular bits of the register, as shown in Figure 10.56 for a 3-bit LFSR. On reset, the registers must be initialized to a nonzero value (e.g., all 1's). The pattern of outputs for the LFSR is shown in Table 10.6.

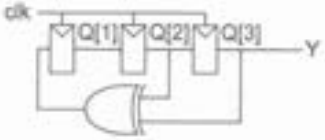


FIG 10.56 3-bit LFSR

This LFSR is an example of a *maximal-length* shift register because its output sequences through all 2^n-1 combinations (excluding all 0's). The inputs fed to the XOR are called the *tap sequence* and are often specified with a *characteristic polynomial*. For example, this 3-bit LFSR has the characteristic polynomial $1 + x^2 + x^3$ because the taps come after the second and third registers.

Table 10.6 LFSR sequence

Cycle	Q[1]	Q[2]	Q[3] / Y
0	1	1	1
1	0	1	1
2	0	0	1
3	1	0	0
4	0	1	0
5	1	0	1
6	1	1	0
7	1	1	1
repeats forever			

The output Y follows the 7-bit sequence [1110010]. This is an example of a *pseudo-random bit sequence* (PRBS) because it is spectrally random. LFSRs are used for high-speed counters and pseudo-random number generators. The pseudo-random sequences are handy for built-in self-test and bit-error-rate testing in communications links. They are also used in many spread-spectrum communications systems such as GPS and CDMA where their correlation properties make other users look like uncorrelated noise.

Table 10.7 lists characteristic polynomials for some commonly used maximal-length LFSRs. For certain lengths, N, more than two taps may be required. For many values of N, there are multiple polynomials resulting in different maximal-length LFSRs. Observe that the cycle time is set by the register and XOR delays, independent of N. [Golomb81] offers the definitive treatment on linear-feedback shift registers.

Table 10.7 Characteristic polynomials

N	Polynomial
3	$1 + x^2 + x^3$
4	$1 + x^3 + x^4$
5	$1 + x^3 + x^5$
6	$1 + x^5 + x^6$
7	$1 + x^6 + x^7$
8	$1 + x^3 + x^6 + x^7 + x^8$
9	$1 + x^5 + x^9$
15	$1 + x^{14} + x^{15}$
16	$1 + x^4 + x^{13} + x^{15} + x^{16}$
23	$1 + x^{18} + x^{23}$
24	$1 + x^{17} + x^{22} + x^{23} + x^{24}$
31	$1 + x^{28} + x^{31}$
32	$1 + x^{10} + x^{30} + x^{31} + x^{32}$

Example

Sketch an 8-bit linear-feedback shift register. How long is the pseudo-random bit sequence that it produces?

Solution: Figure 10.57 shows an 8-bit LFSR using the four taps after the first, sixth, seventh, and eighth bits, as given in Table 10.7. It produces a sequence of $2^8-1 = 255$ bits before repeating.

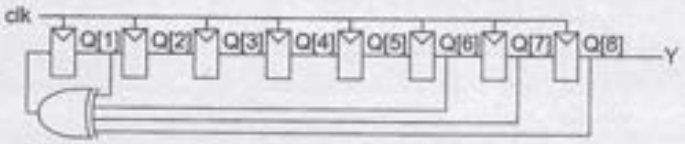


FIG 10.57 8-bit LFSR

10.6 Boolean Logical Operations

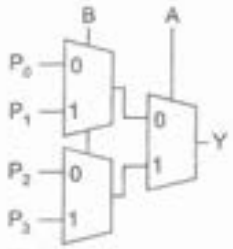


FIG 10.58 Boolean logical unit

Boolean logical operations are easily accomplished using a multiplexer-based circuit shown in Figure 10.58. Table 10.8 shows how the inputs are assigned to perform different logical functions. By providing different P values, the unit can perform other operations such as XNOR(A, B) or NOT(A). An *Arithmetic Logic Unit* (ALU) requires both arithmetic (add, subtract) and Boolean logical operations. You can either multiplex between an adder and Boolean unit or merge the Boolean unit into the adder as in the classic TTL 181 ALU [Weste93].

Table 10.8 Functions implemented by Boolean unit

Operation	P_0	P_1	P_2	P_3
AND(A, B)	0	0	0	1
OR(A, B)	0	1	1	1
XOR(A, B)	0	1	1	0
NAND(A, B)	1	1	1	0
NOR(A, B)	1	0	0	0

10.7 Coding

Error-detecting and error-correcting codes are commonly used to increase system reliability. Memory arrays are particularly susceptible to soft errors caused by alpha particles or

cosmic rays flipping a bit. Such errors can be detected or even corrected by adding a few extra *check bits* to each word in the array. Codes are also used to reduce the bit error rate in communication links.

The simplest form of error-detecting code is *parity*, which detects single-bit errors. More elaborate *error-correcting codes* (ECC) are capable of single-error correcting and double-error detecting (SECDED). Gray codes are another useful alternative to the standard binary codes. All of the codes are heavily based on the XOR function, so we will examine a variety of CMOS XOR designs.

10.7.1 Parity

A parity bit can be added to an N-bit word to indicate whether the number of 1's in the word is even or odd. In *even parity*, the extra bit is the XOR of the other N bits, which ensures the (N+1)-bit coded word has an even number of 1's:

$$A_n = \text{PARITY} = A_0 \oplus A_1 \oplus A_2 \oplus \ldots \oplus A_{n-1} \tag{10.42}$$

Figure 10.59 shows a conventional implementation. Multi-input XOR gates can also be used. In a datapath, the XORs can be placed in a linear column with a tree-routing channel for connections.

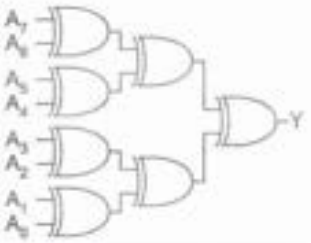


FIG 10.59 8-bit parity generator

10.7.2 Error-correcting Codes

The *Hamming distance* [Hamming50] between a pair of binary numbers is the number of bits that differ between the two numbers. A single-bit error transforms a data word into another word separated by a Hamming distance of 1. Error-correcting codes add check bits to the data word so that the minimum Hamming distance between valid words increases. Parity is an example of a code with a single check bit and a Hamming distance of 2 between valid words, so that single-bit errors lead to invalid words and hence are detectable. If more check bits are added so that the minimum distance between valid words is 3, a single-bit error can be corrected because there will be only one valid word within a distance of 1. If the minimum distance between valid words is 4, a single-bit error can be corrected and an error corrupting two bits can be detected (but not corrected). If the probability of bit errors is low and uncorrelated from one bit to another, such SECDED codes greatly reduce the overall error rate of the system. Larger Hamming distances improve the error rate further at the expense of more check bits.

In general, you can construct a distance-3 Hamming code of length 2^c-1 with c check bits and $N = 2^c-c-1$ data bits [Wakerly00] using a simple procedure. If the bits are numbered from 1 to 2^c-1, each bit in a position that is a power of 2 serves as a check bit. The value of the check bit is chosen to obtain even parity for all bits with a 1 in the same position as the check bit, as illustrated in Figure 10.60(a) for a 7-bit code with 4 data bits and 3 check bits. The bits are traditionally reorganized into contiguous data and check bits, as shown in Figure 10.60(b). The structure is called a *parity-check matrix* and each check bit can be computed as the XOR of the highlighted data bits:

$$
\begin{aligned}
C_0 &= D_3 \oplus D_1 \oplus D_0 \\
C_1 &= D_3 \oplus D_2 \oplus D_0 \\
C_2 &= D_3 \oplus D_2 \oplus D_1
\end{aligned}
\tag{10.43}
$$

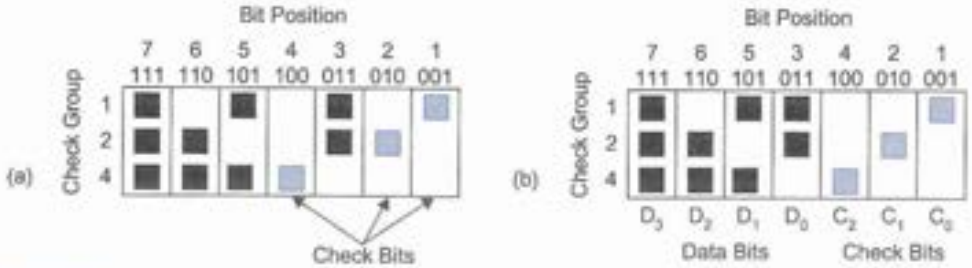


FIG 10.60 Parity-check matrix

The error-correcting decoder examines the check bits. If they all have even parity, the word is considered to be correct. If one or more groups have odd parity, an error has occurred. The pattern of check bits that have the wrong parity is called the *syndrome* and corresponds to the bit position that is incorrect. The decoder must flip this bit to recover the correct result.

Example

Suppose the data value 1001 were to be transmitted using a distance-3 Hamming code. What are the check bits? If the data bits were garbled into 1101 during transmission, explain what the syndrome would be and how the data would be corrected.

Solution: According to EQ (10.43), the check bits should be 100, corresponding to a transmitted word of 1001100. The received word is 1101100. The syndrome is 110, i.e., odd parity on check bits C_2 and C_1, which indicates an error in bit position 110 = 6. This position is flipped to produce a corrected word of 1001100 and the check bits are discarded, leaving the proper data value of 1001.

Many texts provide extensive information on a variety of error-correcting codes. For example, see [Lin83, Sweeney02].

10.7.3 Gray Codes

The *Gray codes*, named for Frank Gray, who patented their use on shaft encoders [Gray53], has a useful property that consecutive numbers differ in only one bit position. While there are many possible Gray codes, one of the simplest is the *binary-reflected Gray code* that is

generated by starting with all bits zero and successively flipping the right-most bit that produces a new string. Table 10.9 compares 3-bit binary and binary-reflected Gray codes. Finite state machines that typically move through consecutive states can save power by Gray-coding the states to reduce the number of transitions. When a counter value must be synchronized across clock domains, it can be Gray-coded so that the synchronizer is certain to receive either the current or previous value because only one bit changes each cycle.

Table 10.9 3-bit Gray code

Number	Binary	Gray Code
0	000	000
1	001	001
2	010	011
3	011	010
4	100	110
5	101	111
6	110	101
7	111	100

Converting between N-bit binary B and binary-reflected Gray code G is remarkably simple.

$$\begin{array}{lll} \text{Binary->Gray} & \text{Gray->Binary} & \\ G_{N-1} = B_{N-1} & B_{N-1} = G_{N-1} & \\ G_i = B_{i+1} \oplus B_i & B_i = B_{i+1} \oplus G_i & N-1 > i \geq 0 \end{array} \tag{10.44}$$

10.7.4 XOR/XNOR Circuit Forms

One of the chronic difficulties in CMOS circuit design is to construct a fast, compact, low-power XOR or XNOR gate. Figure 10.61 shows a number of common static single-rail 2-input XOR designs; XNOR designs are similar. Figure 10.61(a) and Figure 10.61(b) show gate-level implementations; the first is cute, but the second is slightly more efficient. Figure 10.61(c) shows a complementary CMOS gate. Figure 10.61(d) slightly improves the gate by optimizing out two contacts and is a commonly used standard cell design. Figure 10.61(e) shows a transmission gate design. Figure 10.61(f) is the 6-transistor "invertible inverter" design. When A is '0,' the transmission gate turns on and B is passed to the output. When A is '1,' the A input powers a pair of transistors that invert B. It is compact, but nonrestoring. Some switch-level simulators such as IRSIM cannot handle this unconventional design. Figure 10.61(g) [Wang94] is a compact and fast 4-transistor pass-gate design, but does not swing rail to rail.

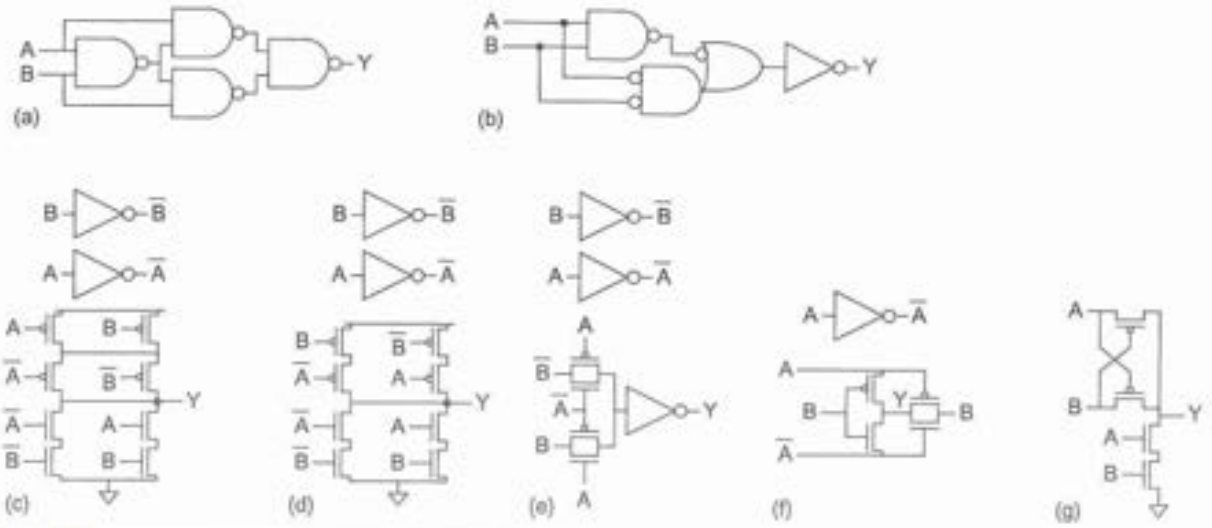


FIG 10.61 Static 2-input XOR designs

XOR gates with 3 or 4 inputs can be more compact, although not necessarily faster than a cascade of 2-input gates. Figure 10.62(a) is a 4-input static CMOS XOR [Griffin83] and Figure 10.62(b) is a 4-input CPL XOR/XNOR, while Figure 6.20(c) showed a 4-input CVSL XOR/XNOR. Observe that the true and complementary trees share most of the transistors. As mentioned in Chapter 6, CPL does not perform well at low voltage.

Dynamic XORs pose a problem because both true and complementary inputs are required, violating the monotonicity rule. The common solutions mentioned in Section 10.2.2.11 are to either push the XOR to the end of a chain of domino logic and build it with static CMOS or to construct a dual-rail domino structure. The dual-rail domino 2-input XOR was shown in Figure 6.30(c).

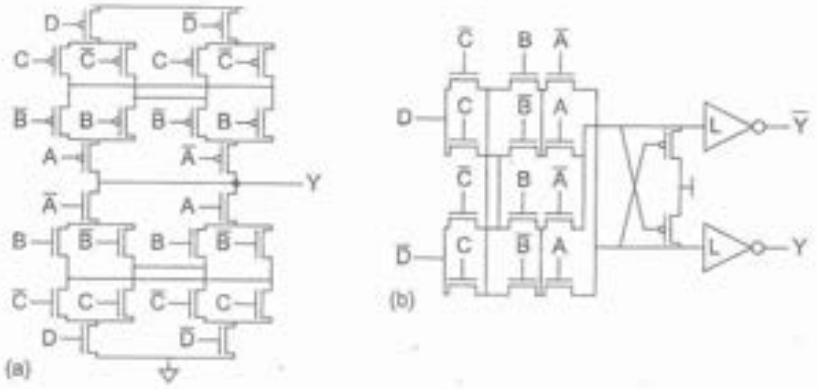


FIG 10.62 4-input XOR designs

10.8 Shifters

There are several commonly used shifters:

- *Logical shifter*: Shifts the number to the left or right and fills empty spots with 0's. Specified by << or >> in Verilog.
  - Example: 1011 LSR 1 = 0101; 1011 LSL 1 = 0110
- *Arithmetic shifter*: Same as logical shifter, but on right shifts fills the most significant bits with copies of the sign bit (to properly sign, extend 2's complement numbers when using right shift by k for division by 2^k). Specified by <<< or >>> in Verilog.
  - Example: 1011 ASR 1 = 1101; 1011 ASL 1 = 0110
- *Barrel shifter* (rotator): Rotates numbers in a circle such that empty spots are filled with bits shifted off the other end
  - Example: 1011 ROR 1 = 1101; 1011 ROL 1 = 0111

The *funnel shifter* can perform all three such operations. It concatenates two N-bit inputs and selects an N-bit subfield Y, as shown in Figure 10.63. Table 10.10 lists what inputs should be applied to perform a left or right shift of an N-bit word A by k bits. Generating the offset for left shifts requires a 2's complement subtraction. If N is a power of 2, the funnel shifter can be further simplified by accepting a $(2N-1)$-bit input Z and simply taking the 1's complement (i.e., bitwise inversion) of the shift amount for left shifts, as shown in Table 10.11. The funnel shifter can support all types of shifts and rotates by choosing suitable inputs Z, or it can be hardwired for only particular shifts.

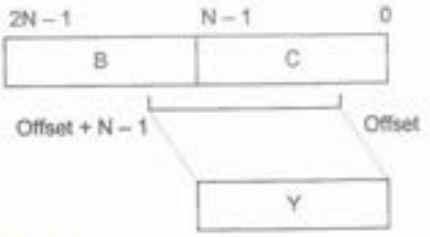


FIG 10.63 Funnel shifter function

Table 10.10 Funnel shifter operation

Shift Type	B	C	Offset
Logical Right	0…0	$A_{N-1}\ldots A_0$	k
Logical Left	$A_{N-1}\ldots A_0$	0…0	$N-k$
Arithmetic Right	$A_{N-1}\ldots A_{N-1}$ (sign extension)	$A_{N-1}\ldots A_0$	k
Arithmetic Left	$A_{N-1}\ldots A_0$	0	$N-k$
Rotate Right	$A_{N-1}\ldots A_0$	$A_{N-1}\ldots A_0$	k
Rotate Left	$A_{N-1}\ldots A_0$	$A_{N-1}\ldots A_0$	$N-k$

Table 10.11 Simplified funnel shifter

Shift Type	Z	Offset
Logical Right	$0..0, A_{N-1}...A_0$	k
Logical Left	$A_{N-1}...A_0, 0..0$	$\overline{k}$
Arithmetic Right	$A_{N-1}...A_{N-1}, A_{N-1}...A_0$	k
Arithmetic Left	$A_{N-1}...A_0, 0..0$	$\overline{k}$
Rotate Right	$A_{N-2}...A_0, A_{N-1}...A_0$	k
Rotate Left	$A_{N-1}...A_0, A_{N-1}..A_1$	$\overline{k}$

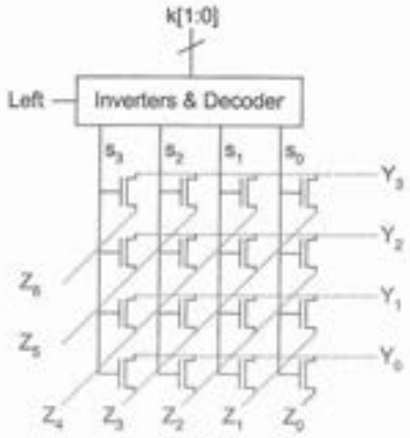


FIG 10.64 Array funnel shifter

The simplest funnel shifter design consists of an array of N N-input multiplexers accepting 1-of-N-hot select signals (one multiplexer for each output bit). Such an array shifter is shown in Figure 10.64 using nMOS pass transistors for a 4-bit shifter. The shift amount is conditionally inverted and decoded into select signals that are fed vertically across the array. The outputs are taken horizontally. Each row of transistors attached to an output forms one of the multiplexers. The $2N$-1 inputs run diagonally to the appropriate mux inputs. Figure 10.65 shows a stick diagram for one of the N^2 transistors in the array. nMOS pass transistors suffer a threshold drop, but the problem can be solved by precharging the outputs (done in the Alpha 21164 [Gronowski96]) or by using full CMOS transmission gates.

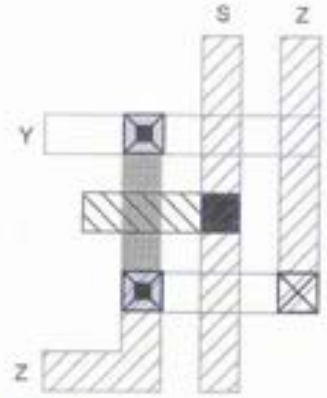


FIG 10.65 Array funnel shifter cell stick diagram

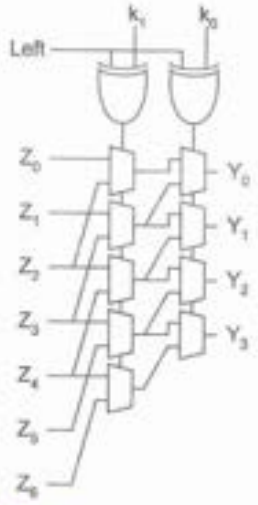


FIG 10.66 Multilevel funnel shifter

The array shifter works reasonably well for medium-sized shifters in transistor-level designs, but has high parasitic capacitance in larger shifters and is not amenable to standard cell designs. Figure 10.66 shows a 4-bit shifter based on multiple levels of smaller multiplexers (which, of course, can be transmission gates) [Lim72]. In general, the first level shifts by $N/2$, the second by $N/4$, and so forth until the final level shifts by 1. With such a structure, no decoder is necessary. The XOR gates on the control inputs conditionally invert the shift amount for left shifts. [Tharakan92] describes a domino implementation using 3:1 multiplexers to reduce the number of logic levels.

Other shift options, including shuffles, bit-reversals, and interchanges, are sometimes required. These are also built

from appropriate combinations of multiplexers. The speed of an N-bit shifter is proportional to log N, so shifting can be a fast operation. 4:1 multiplexers work well in larger shifters to reduce the number of logic levels.

10.9 Multiplication

Multiplication is a less common operation than addition, but is still essential for microprocessors, digital signal processors, and graphics engines. Multiplication algorithms will be used to illustrate methods of designing different cells so that they fit into a larger structure. The most basic form of multiplication consists of forming the product of two unsigned (positive) binary numbers. This can be accomplished through the traditional technique taught in primary school, simplified to base 2. For example, the multiplication of two positive 4-bit binary integers, 12_{10} and 5_{10}, proceeds as shown in Figure 10.67.

```
    1100 : 12₁₀        Multiplicand
    0101 : 5₁₀         Multiplier
    1100               ┐
   0000                │ Partial
  1100                 │ Products
 0000                  ┘
00111100 : 60₁₀        Product
```

FIG 10.67 Multiplication example

$M \times N$-bit multiplication can be viewed as forming N partial products of M bits each, and then summing the appropriately shifted partial products to produce an $M+N$-bit result P. Binary multiplication is equivalent to a logical AND operation. Therefore, generating partial products consists of the logical ANDing of the appropriate bits of the multiplier and multiplicand. Each column of partial products must then be added and, if necessary, any carry values passed to the next column. We denote the multiplicand as $Y = (y_{M-1}, y_{M-2}, \ldots, y_1, y_0)$ and the multiplier as $X = (x_{N-1}, x_{N-2}, \ldots, x_1, x_0)$. For unsigned multiplication, the product is given in EQ (10.45). Figure 10.68 illustrates the generation, shifting, and summing of partial products in a 6 × 6-bit multiplier.

$$P = \left(\sum_{j=0}^{M-1} y_j 2^j\right)\left(\sum_{i=0}^{N-1} x_i 2^i\right) = \sum_{i=0}^{N-1}\sum_{j=0}^{M-1} x_i y_j 2^{i+j} \qquad \textbf{(10.45)}$$

						y_5	y_4	y_3	y_2	y_1	y_0	Multiplicand
						x_5	x_4	x_3	x_2	x_1	x_0	Multiplier
						x_0y_5	x_0y_4	x_0y_3	x_0y_2	x_0y_1	x_0y_0	┐
					x_1y_5	x_1y_4	x_1y_3	x_1y_2	x_1y_1	x_1y_0		
				x_2y_5	x_2y_4	x_2y_3	x_2y_2	x_2y_1	x_2y_0			Partial
			x_3y_5	x_3y_4	x_3y_3	x_3y_2	x_3y_1	x_3y_0				Products
		x_4y_5	x_4y_4	x_4y_3	x_4y_2	x_4y_1	x_4y_0					
	x_5y_5	x_5y_4	x_5y_3	x_5y_2	x_5y_1	x_5y_0						┘
p_{11}	p_{10}	p_9	p_8	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0	Product

FIG 10.68 Partial products

Large multiplications can be more conveniently illustrated using *dot diagrams*. Figure 10.69 shows a dot diagram for a simple 16 × 16 multiplier. Each dot represents a placeholder for a single bit that can be a 0 or 1. The partial products are represented by a horizontal boxed row of dots, shifted according to their weight. The multiplier bits used to generate the partial products are shown on the right.

There are a number of techniques that can be used to perform multiplication. In general, the choice is based upon factors such as latency, throughput, area, and design complexity. An obvious approach is to use an M+1-bit carry-propagate adder (CPA) to add the first two partial products, then another CPA to add the third partial product to the running sum, and so forth. Such an approach requires N–1 CPAs and is slow, even if a fast CPA is employed. More efficient parallel approaches use some sort of array or tree of full adders to sum the partial products. We begin with a simple array for unsigned multipliers, and then modify the array to handle signed 2's complement numbers using the Baugh-Wooley algorithm. The number of partial products to sum can be reduced using Booth encoding and the number of logic levels required to perform the summation can be reduced with Wallace trees. Unfortunately, Wallace trees are complex to lay out and have long, irregular wires, so hybrid array/tree structures may be more attractive. For completeness, we consider a serial multiplier architecture. This was once popular when gates were relatively expensive, but is now less often necessary. Multipliers are discussed in more detail in many computer arithmetic texts [Ercegovac04, Parhami00, Koren02, Flynn01].

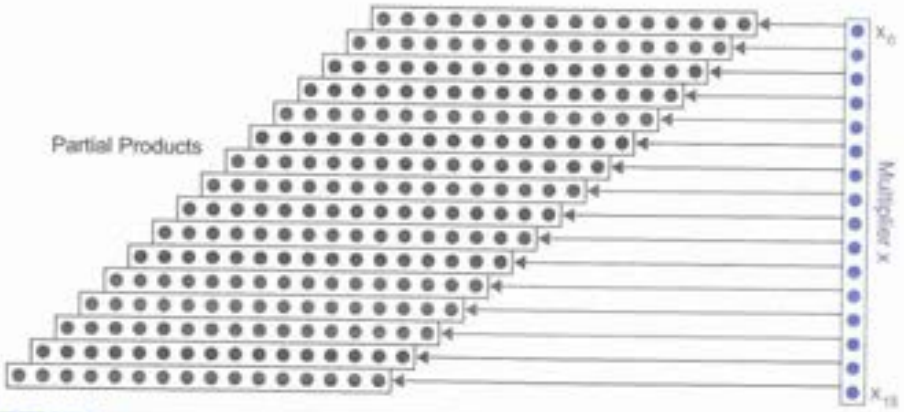


FIG 10.69 Dot diagram

10.9.1 Unsigned Array Multiplication

Fast multipliers use carry-save adders (CSAs, see Section 10.2.3.2) to sum the partial products. A CSA typically has a delay of 1.5–2 FO4 inverters independent of the width of the partial product, while a carry-propagate adder (CPA) tends to have a delay of 4–15* FO4 inverters depending on the width, architecture, and circuit family. Figure 10.70 shows a 4 × 4 array multiplier for unsigned numbers using an array of CSAs. Each cell

contains a 2-input AND gate that forms a partial product and a full adder (CSA) to add the partial product into the running sum. The first row converts the first partial product into carry-save redundant form. Each later row uses the CSA to add the corresponding partial product to the carry-save redundant result of the previous row and generate a carry-save redundant result. The least significant N output bits are available as sum outputs directly from CSAs. The most significant output bits arrive in carry-save redundant form and require an M-bit carry-propagate adder to convert into regular binary form. In the figure, the CPA is implemented as a carry-ripple adder. The array is regular in structure and uses a single type of cell, so it is easy to design and lay out. Assuming the carry output is faster than the sum output in a CSA, the critical path through the array is marked on the figure with a dashed line. The adder can easily be pipelined with the placement of registers between rows. In practice, circuits are assigned rectangular blocks in the floorplan so the parallelogram shape wastes space. Figure 10.71 shows the same adder squashed to fit a rectangular block.

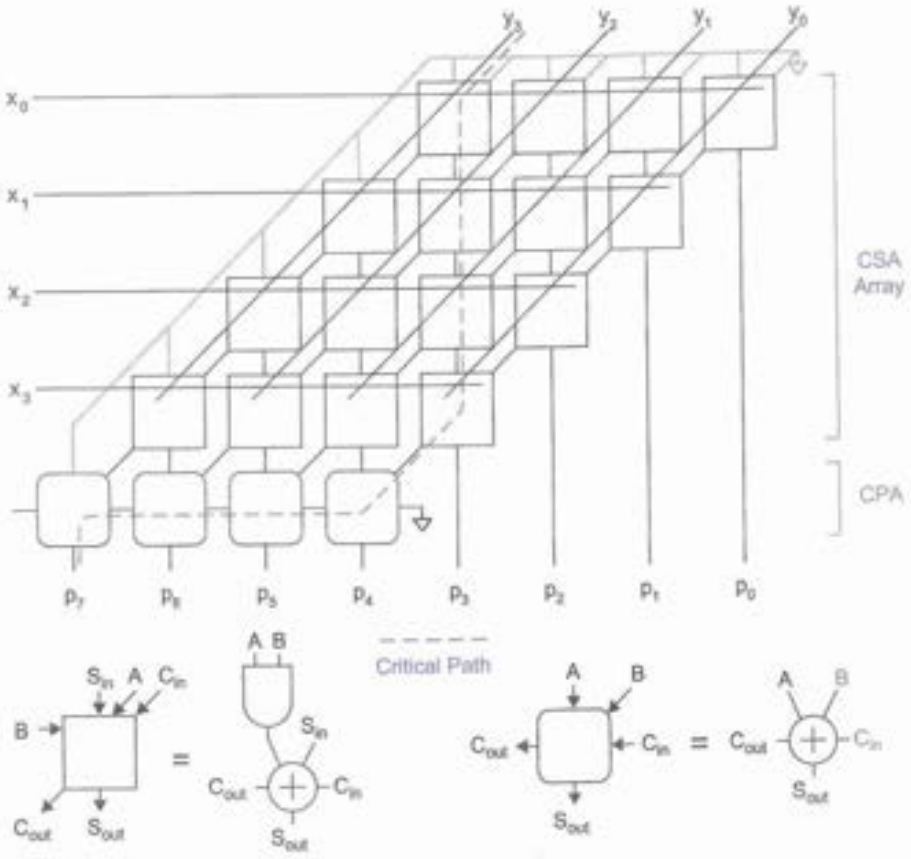


FIG 10.70 Array multiplier

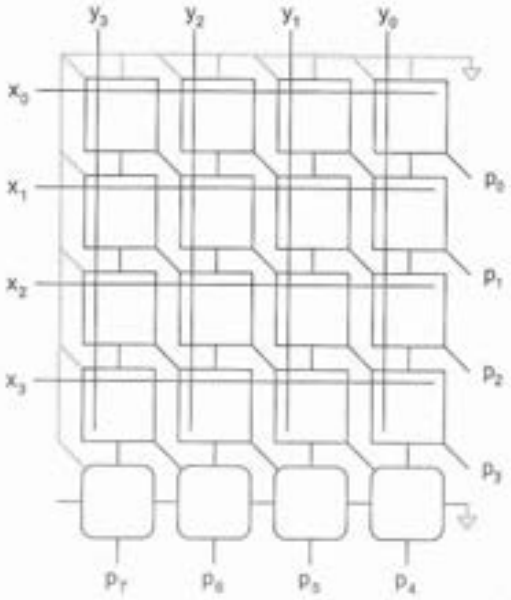


FIG 10.71 Rectangular array multiplier

A key element of the design is a compact CSA. This not only benefits area but also helps performance because it leads to short wires with low wire capacitance. An ideal CSA design has approximately equal sum and carry delays because the greater of these two delays limits performance. Note that the first row of CSAs adds the first partial product to a pair of 0's. This leads to a regular structure, but is inefficient. At a slight cost to regularity, the first row of CSAs can be used to add the first three partial products together. This reduces the number of rows by two and correspondingly reduces the adder propagation delay. It is also possible to use the first row of CSAs to add one or two other inputs with no extra delay. For example, digital signal processing chips frequently offer multiply-accumulate units (MACs) that compute $Y = A \cdot B + C$, a key operation for filters and transforms. Yet another way to improve the multiplier array performance is to replace the bottom row with a faster CPA such as a lookahead or tree adder. In summary, the critical path of an array multiplier involves N-2 CSAs and a CPA.

The CSA is an asymmetric circuit: certain inputs have more logical effort than others. The multiplier can be designed so that the sum output (with greater logical effort) drives the input to the next stage with lower capacitance [Sutherland99].

10.9.2 2's Complement Array Multiplication

Multiplication of 2's complement numbers at first might seem more difficult because some partial products are negative and must be subtracted. Recall that the most significant bit of a 2's complement number has a negative weight. Hence, the product is:

$$\begin{aligned} P &= \left(-y_{M-1}2^{M-1} + \sum_{j=0}^{M-2} y_j 2^j\right)\left(-x_{n-1}2^{N-1} + \sum_{i=0}^{N-2} x_i 2^i\right) \\ &= \sum_{i=0}^{N-2}\sum_{j=0}^{M-2} x_i y_j 2^{i+j} + x_{N-1}y_{M-1}2^{M+N-2} - \left(\sum_{i=0}^{N-2} x_i y_{M-1} 2^{i+M-1} + \sum_{j=0}^{M-2} x_{N-1} y_j 2^{j+N-1}\right) \end{aligned} \tag{10.46}$$

In EQ 10.46, two of the partial products have negative weight and thus should be subtracted rather than added. The *Baugh-Wooley* [Baugh73] multiplier algorithm handles subtraction by taking the 2's complement of the terms to be subtracted (i.e., inverting the bits and adding one). Figure 10.72 shows the partial products that must be summed. The upper parallelogram represents the unsigned multiplication of all but the most significant bits of the inputs. The next row is a single bit corresponding to the product of the most significant bits. The next two pairs of rows are the inversions of the terms to be subtracted. Each term has implicit leading and trailing 0's, which are inverted to leading and trailing 1's. Extra 1's must be added in the least significant column when taking the 2's complement.

							y_5	y_4	y_3	y_2	y_1	y_0
							x_5	x_4	x_3	x_2	x_1	x_0
$\sum_{i=0}^{N-2}\sum_{j=0}^{N-2} x_i y_j 2^{i+j}$								x_0y_4	x_0y_3	x_0y_2	x_0y_1	x_0y_0
							x_1y_4	x_1y_3	x_1y_2	x_1y_1	x_1y_0	
						x_2y_4	x_2y_3	x_2y_2	x_2y_1	x_2y_0		
					x_3y_4	x_3y_3	x_3y_2	x_3y_1	x_3y_0			
				x_4y_4	x_4y_3	x_4y_2	x_4y_1	x_4y_0				
$x_{N-1}\, y_{N-1}\, 2^{2N-N-2}$		x_5y_5										
$-\sum_{i=0}^{N-2} x_i\, y_{N-1}\, 2^{i+N-1}$	1	1	$\overline{x_4y_5}$	$\overline{x_3y_5}$	$\overline{x_2y_5}$	$\overline{x_1y_5}$	$\overline{x_0y_5}$	1	1	1	1	1
												1
$-\sum_{j=0}^{N-2} x_{N-1}\, y_j\, 2^{N+j-1}$	1	1	$\overline{x_5y_4}$	$\overline{x_5y_3}$	$\overline{x_5y_2}$	$\overline{x_5y_1}$	$\overline{x_5y_0}$	1	1	1	1	1
												1
	p_{11}	p_{10}	p_9	p_8	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0

FIG 10.72 Partial products for 2's complement multiplier

The multiplier delay depends on the number of partial product rows to be summed. The *modified Baugh–Wooley multiplier* [Hatamian86] reduces this number of partial products by precomputing the sums of the constant 1's and pushing some of the terms upward into extra columns. Figure 10.73 shows such an arrangement. The parallelogram-shaped array can again be squashed into a rectangle as shown in Figure 10.74, giving a design almost identical to the unsigned multiplier of Figure 10.71. The AND gates are replaced by NAND gates in the hatched cells and 1's are added in place of 0's at a few of the unused inputs. The signed and unsigned arrays are so similar that a single array can be used for both purposes if XOR gates are used to conditionally invert some of the terms depending on the mode.

						y_5	y_4	y_3	y_2	y_1	y_0
						x_5	x_4	x_3	x_2	x_1	x_0
					1	$\overline{x_0y_5}$	x_0y_4	x_0y_3	x_0y_2	x_0y_1	x_0y_0
					$\overline{x_5y_1}$	x_1y_4	x_1y_3	x_1y_2	x_1y_1	x_1y_0	
				$\overline{x_5y_2}$	x_2y_4	x_2y_3	x_2y_2	x_2y_1	x_2y_0		
			$\overline{x_5y_3}$	x_3y_4	x_3y_3	x_3y_2	x_3y_1	x_3y_0			
		$\overline{x_0y_4}$	x_4y_4	x_4y_3	x_4y_2	x_4y_1	x_4y_0				
1	x_5y_5	$\overline{x_4y_5}$	$\overline{x_3y_5}$	$\overline{x_2y_5}$	$\overline{x_1y_5}$	$\overline{x_0y_5}$					
p_{11}	p_{10}	p_9	p_8	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0

FIG 10.73 Simplified partial products for 2's complement multiplier

10.9.3 Booth Encoding

The array multipliers in the previous sections compute the partial products in a radix-2 manner, i.e., by observing one bit of the multiplier at a time. Radix 2^r multipliers produce N/r partial products, each of which depend on r bits of the multiplier. Fewer partial products leads to a smaller and faster CSA array. For example, a radix-4 multiplier produces $N/2$ partial products. Each partial product is 0, Y, $2Y$, or $3Y$, depending on a pair of bits of X. Computing $2Y$ is a simple shift, but $3Y$ is a *hard* multiple requiring a slow carry-propagate addition of $Y + 2Y$ before partial product generation begins.

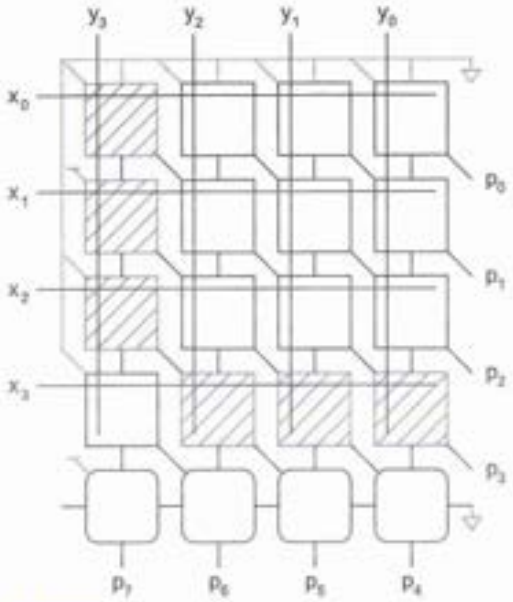


FIG 10.74 Modified Baugh-Wooley 2's complement multiplier

Booth encoding was originally proposed to accelerate serial multiplication [Booth51]. *Modified Booth encoding* [MacSorley61] allows higher radix parallel operation without generating the hard $3Y$ multiple by instead using negative partial products. Observe that $3Y = 4Y - Y$ and $2Y = 4Y - 2Y$. However, $4Y$ in a radix-4 multiplier array is equivalent to Y in the next row of the array that carries four times the weight. Hence, partial products are chosen by considering a pair of bits along with the most significant bit from the previous pair. If the most significant bit from the previous pair is true, Y must be added to the current partial product. If the most significant bit of the current pair is true, the current partial product is selected to be negative and the next partial product is incremented. Table 10.12 shows how the partial products are selected, based on bits of the multiplier. Negative partial products are generated by taking the 2's complement of the multiplicand (possibly left-shifted by one column).

Table 10.12 Radix-4 modified Booth encoding values

Inputs			Partial Product	Booth Selects		
x_{2i+1}	x_{2i}	x_{2i-1}	PP_i	X_i	$2X_i$	M_i
0	0	0	0	0	0	0
0	0	1	Y	1	0	0
0	1	0	Y	1	0	0
0	1	1	$2Y$	0	1	0
1	0	0	$-2Y$	0	1	1
1	0	1	$-Y$	1	0	1
1	1	0	$-Y$	1	0	1
1	1	1	-0 (= 0)	0	0	1

In a radix-4 Booth-encoded multiplier, each group of three bits (a pair, along with the most significant bit of the previous pair) is decoded into several select lines (X_i, $2X_i$, and M_i, given in the rightmost columns of Table 10.12) and driven across the partial product row as shown in Figure 10.75. The multiplier Y is distributed to all the rows. The select lines control Booth selectors that choose the appropriate multiple of Y for each partial product. The Booth selectors substitute for the AND gates of a simple array multiplier. Figure 10.75 shows a conventional Booth selector design that computes the jth partial product bit of the ith partial product [Chandrakasan01]. If the partial product has a magnitude of Y, y_j is selected. If it has a magnitude of $2Y$, y_{j-1} is selected. If it is negative, the multiple is inverted (and a 1 is added to the least significant column elsewhere in the array to form the 2's complement). Exercise 10.18 explores an alternative Booth encoding that simplifies the encoder logic.

Even in an unsigned multiplier, negative partial products must be sign-extended to be summed correctly. Figure 10.76 shows a 16-bit radix-4 Booth partial product array for an unsigned multiplier using the dot diagram notation. Each dot in the Booth-encoded multiplier is produced by a Booth selector rather than a simple AND gate. For each partial product i, the sign bit $s_i = M_i = x_{2i+1}$ is 1 for negative multiples (those in the bottom half of Table 10.12) or 0 for positive multiples. Observe how an extra 1 is added to the least significant bit in the next row to form the 2's complement of negative multiples. Inverting the implicit leading 0's generates leading 1's on negative multiples. The extra terms increase the size of the multiplier. PP_8 is required in case PP_7 is negative; this partial product is always 0 or Y because x_{16} and x_{17} are 0.

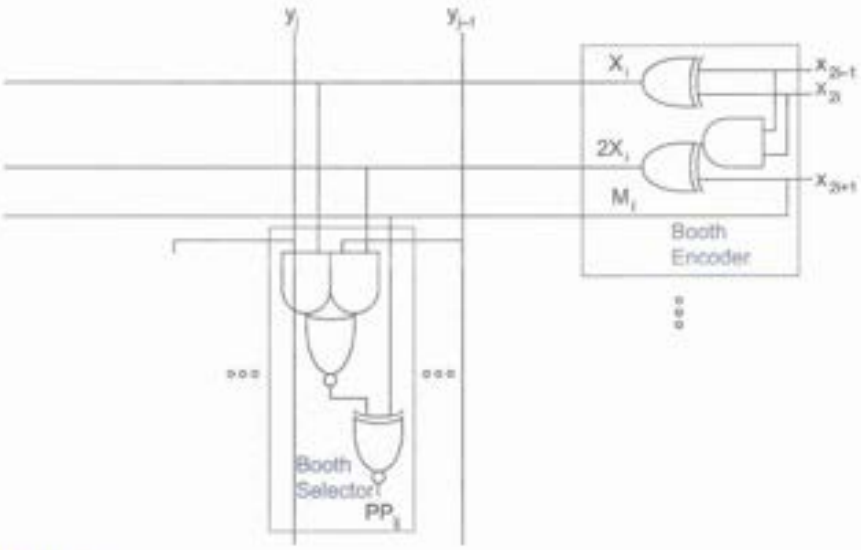


FIG 10.75 Radix-4 Booth encoder and selector

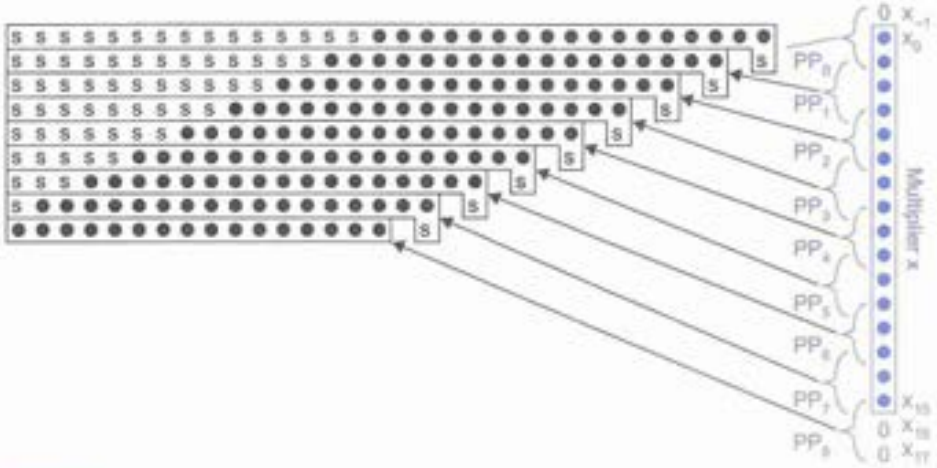


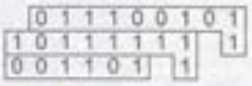
FIG 10.76 Radix-4 Booth-encoded partial products with sign extension

Observe that the sign extension bits are all either 1's or 0's. If a single 1 is added to the least significant position in a string of 1's, the result is a string of 0's plus a carry-out the top bit that may be discarded. Therefore, the large number of *s* bits in each partial product can be replaced by an equal number of constant 1's plus the inverse of *s* added to the least significant position, as shown in Figure 10.78. These constants mostly can be optimized out of the array by precomputing their sum. The simplified result is shown in Figure 10.78. As usual, it can be squashed to fit a rectangular floorplan.

Example

Sketch the partial products used by a radix-4 Booth-encoded multiplier to compute $01110_2 \times 01101_2$.

Solution: Figure 10.77 shows the three partial products. They sum to $14 \times 13 = 182$ as desired.



X_{2i+1}	X_{2i}	X_{2i-1}	P_i
1	0	n/a	–2Y = 100101 + 1
1	1	1	–0Y = 111111 + 1
n/a	0	1	Y = 001101 + 0

FIG 10.77 Example of Booth-encoded partial products

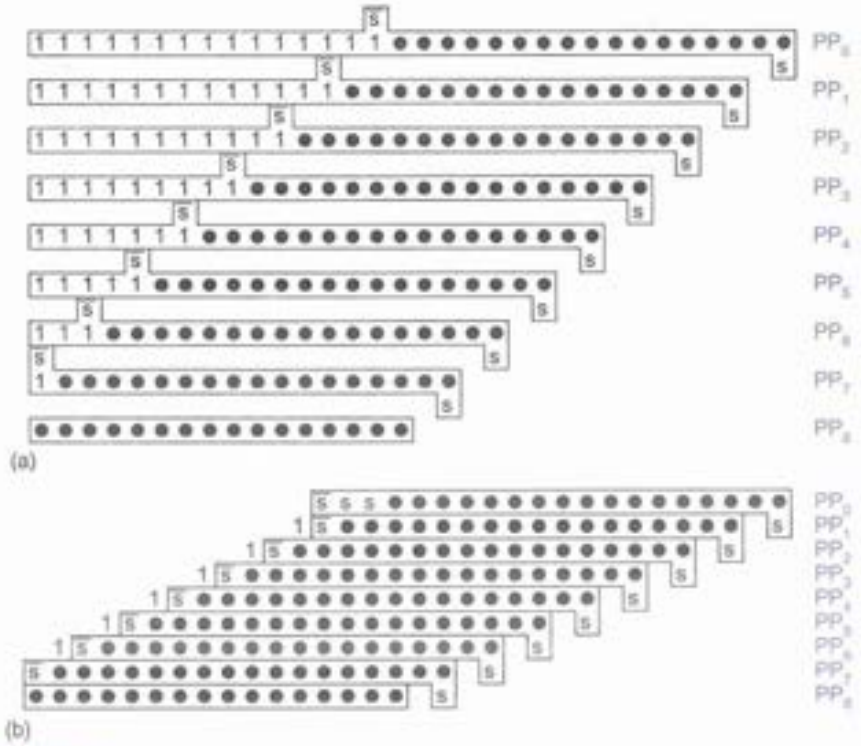


FIG. 10.78 Radix-4 Booth-encoded partial products with simplified sign extension

The critical path of the multiplier involves the Booth decoder, the select line drivers, the Booth selector, approximately $N/2$ CSAs, and a final CPA. Each partial product fills about $M + 5$ columns. 53×53-bit radix-4 Booth multipliers for IEEE double-precision floating-point units are typically 20%–50% smaller (and arguably up to 20% faster) than nonencoded counterparts, so the technique is widely used. The multiplier requires $M \cdot N/2$ Booth selectors. Because they account for a substantial portion of the area and only a small fraction of the critical path, they should be optimized for size over speed.

Signed 2's complement multiplication is similar, but the sign extension is based on the sign of the partial product (i.e., the most significant bit) rather than simply M_i because the multiplicand might have been negative [Bewick94]. Figure 10.79 shows such an array, where the sign extension bit is $e_i = M_i \oplus y_{15}$. Also notice that PP_8, which was either Y or 0 for unsigned multiplication, is always 0 and can be omitted for signed multiplication because the multiplier x is sign-extended such that $x_{17} = x_{16} = x_{15}$.

Large multipliers can use Booth encoding of higher radix. For example, ordinary radix-8 multiplication reduces the number of partial products by a factor of three, but requires hard multiples of $3Y$, $5Y$, and $7Y$. Radix-8 Booth-encoding only requires the hard

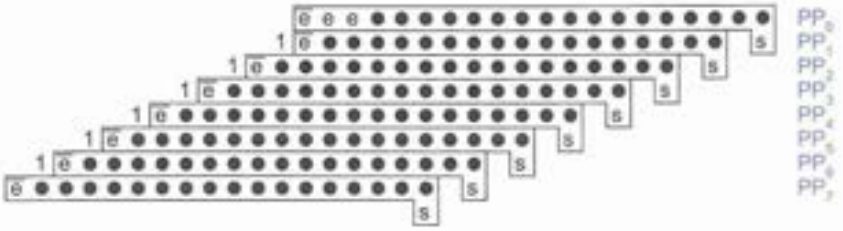


FIG 10.79 Radix-4 Booth-encoded partial products for signed multiplication

3*Y* multiple, as shown in Table 10.13. Although this requires a CPA before partial product generation, it can be justified by the reduction in array size and delay. Higher-radix Booth encoding is possible, but generating the other hard multiples appears not to be worthwhile for multipliers of fewer than 64 bits. Similar techniques apply to sign-extending higher-radix multipliers. Yet another approach to radix-8 multiplication is to generate *partially redundant* multiples [Bewick94], thus avoiding the hard multiple. This provides speed, area, and power advantages over radix-4 Booth.

Table 10.13 Radix-8 modified Booth encoding values

x_{j+2}	x_{j+1}	x_j	x_{j-1}	Partial Product
0	0	0	0	0
0	0	0	1	*Y*
0	0	1	0	*Y*
0	0	1	1	2*Y*
0	1	0	0	2*Y*
0	1	0	1	3*Y*
0	1	1	0	3*Y*
0	1	1	1	4*Y*
1	0	0	0	-4*Y*
1	0	0	1	-3*Y*
1	0	1	0	-3*Y*
1	0	1	1	-2*Y*
1	1	0	0	-2*Y*
1	1	0	1	-*Y*
1	1	1	0	-*Y*
1	1	1	1	-0

10.9.4 Wallace Tree Multiplication

Observe that a CSA is effectively a "1's counter" that adds the number of 1's on the *A*, *B*, and *C* inputs and encodes them on the sum and carry outputs, as summarized in Table 10.14. A CSA is therefore also known as a *(3,2) counter* [Dadda65] because it converts three inputs into a count encoded in two outputs. The carry-out is passed to the next more significant column, while a corresponding carry-in is received from the previous column. Therefore, for simplicity, a carry is represented as being passed directly down the column. Figure 10.80 shows a dot diagram of an array multiplier column that sums *N* partial products sequentially using *N*–2 CSAs. The output is produced in carry-save redundant form suitable for the final CPA.

Table 10.14 An adder as a 1's counter

A	B	C	Carry	Sum	Number of 1's
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	0	1	1
0	1	1	1	0	2
1	0	0	0	1	1
1	0	1	1	0	2
1	1	0	1	0	2
1	1	1	1	1	3

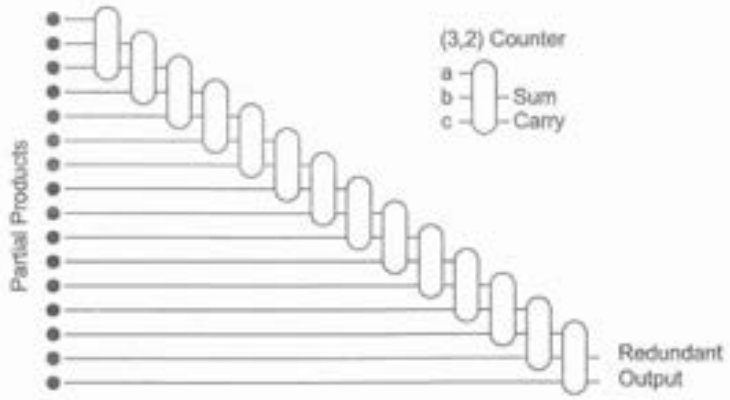


FIG 10.80 Dot diagram for array multiplier

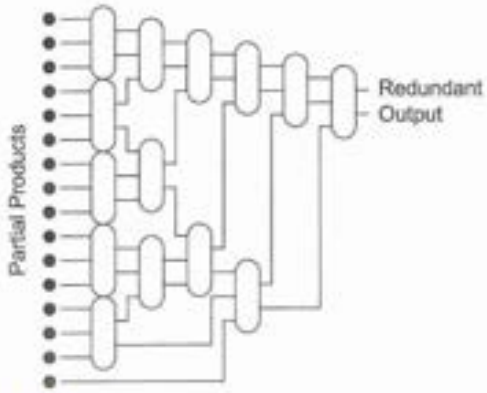


FIG 10.81 Dot diagram for Wallace tree multiplier

The column addition is slow because only one CSA is active at a time. Another way to speed the column addition is to sum partial products in parallel rather than sequentially. Figure 10.81 shows a *Wallace tree* using this approach [Wallace64]. The Wallace tree requires

$$\left\lceil \log_{3/2}\left(N/2\right)\right\rceil$$

levels of (3,2) counters to reduce N inputs down to 2 carry-save redundant form outputs. Unfortunately, the routing between levels becomes much more complicated. The longer wires have greater wire capacitance and the irregular tree is difficult to lay out.

[4:2] compressors can be used in a binary tree to produce a much more regular layout, shown in Figure 10.82 [Weinberger81, Santoro89]. A [4:2] compressor takes four inputs of equal weight and produces two outputs. It can be constructed from two (3,2) counters as shown in Figure 10.83(a). Along the way, it generates an intermediate carry into the next column and accepts a carry from the previous column, so it may more aptly be called a *(5,3) counter*. Only

$$\left\lceil \log_{2}\left(N/2\right)\right\rceil$$

levels of [4:2] compressors are required, although each has greater delay than a CSA. The regular layout and routing also make the binary tree attractive. Figure 10.83(b) shows a [4:2] compressor design using fewer levels of logic [Ohkubo95]. [Itoh01n] describes layout issues in a tree multiplier.

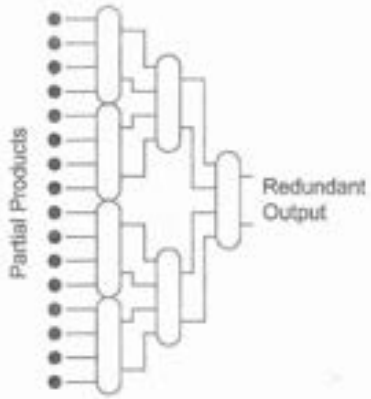


FIG 10.82 Dot diagram for binary-tree multiplier

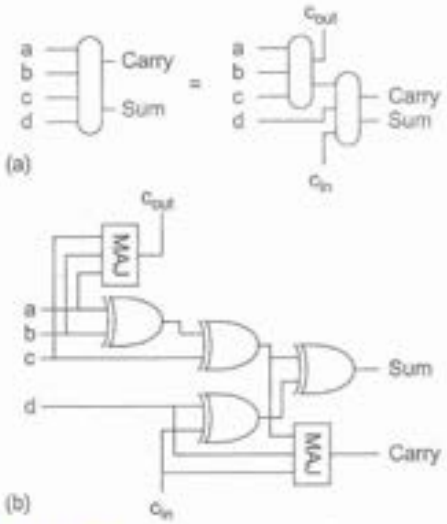


FIG 10.83 [4:2] compressor

10.9.5 Hybrid Multiplication

Arrays offer regular layout, but many levels of CSAs. Trees offer fewer levels of CSAs, but less regular layout and some long wires. A number of hybrids have been proposed that offer tradeoffs between these two extremes. These include *odd/even arrays* [Hennessy90], *arrays of arrays* [Dhanesha95], *balanced delay trees* [Zuras86], and *overturned-staircase trees* [Mou90]. They can achieve nearly as few levels of logic as the Wallace tree while offering more regular (and faster) wiring.

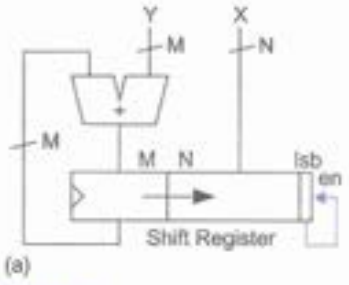
10.9.6 Fused Multiply-Add

Many algorithms, particularly in digital signal processing, require computing $P = X \cdot Y + Z$. While this can be done with a multiplier and adder, it is much faster to use a *fused multiply-add* unit, which is simply an ordinary multiplier modified to take another partial product Z. The extra partial product increases the delay of an array multiplier by just one extra CSA.

10.9.7 Serial Multiplication

Large parallel multipliers consume huge numbers of transistors. While transistor budgets have expanded to the point that this is often acceptable, designers of low-cost systems still may find serial multiplication attractive. Serial multiplication uses far less hardware, but requires multiple clock cycles to operate. Multiplication can be performed in a word-serial or bit-serial fashion.

Figure 10.84(a) shows a word-serial multiplication unit that only requires an M-bit adder and an $(M+N)$-bit shift register [Patterson04]. On each step, it conditionally adds the multiplicand Y to the running product if the appropriate bit of the multiplier X is 1. It is based on the observation that on the kth step, the running product has a length of $M+k$ bits and that bits $0 \ldots k-1$ of X have already been considered and are no longer necessary. The unit is initialized by loading all of X into the lower portion of the shift register and a running product of 0's into the upper portion. On step k, Y is added to the running product if $x_k = 1$. The shift register then shifts right, discarding x_k and doubling the weight at which the next partial product will be added to the running product. After N steps, the



(a)

Step	Shift Reg	Notes
	0000|0101	initialize
0a	1100|0101	add 1*Y
0b	01100|010	shift right
1a	01100|010	add 0*Y
1b	001100|01	shift right
2a	111100|01	add 1*Y
2b	0111100|0	shift right
3a	0111100|0	add 0*Y
3b	00111100|	shift right

(b)

FIG 10.84 Word-serial multiplier

shift register will contain the final product. Figure 10.84(b) demonstrates multiplying 1100 ×0101 = 00111100. The vertical bar separates the running product from the remaining bits of X.

The cycle time of word-serial multiplication is set by the M-bit carry-propagate addition on each step. This CPA delay can be shortened to a CSA delay by maintaining the partial product in carry-save redundant form. The cost is doubling the number of registers to hold the redundant partial product and a final CPA to convert the redundant result into a 2's complement number at the end of the multiplication.

10.10 Parallel-prefix Computations

Many datapath operations involve calculating a set of outputs from a set of inputs in which each output bit depends on all the previous input bits. Addition of two N-bit inputs $A_N...A_1$ and $B_N...B_1$ to produce a sum output $Y_N...Y_1$ is a classic example; each output Y_i depends on a carry-in c_{i-1} from the previous bit, which in turn depends on a carry-in c_{i-2} from the bit before that, and so forth. At first, this dependency chain might seem to suggest that the delay must involve about N stages of logic, as in a carry-ripple adder. However, we have seen that by looking ahead across progressively larger blocks, we can construct adders that involve only log N stages. Section 10.2.2.2 introduced the notion of addition as a prefix computation that involves a bitwise precomputation, a tree of group logic to form the prefixes, and a final output stage, shown in Figure 10.13. In this section, we will extend the same techniques to other prefix computations with associative group logic functions.

Let us begin with the *priority encoder* shown in Figure 10.85. A common application of a priority encoder circuit is to arbitrate among N units that are all requesting access to a shared resource. Each unit i sends a bit A_i indicating a request and receives a bit Y_i indicating that it was granted access; access should only be granted to a single unit with highest priority. If the least significant bit of the input corresponds to the highest priority, the logic can be expressed as:

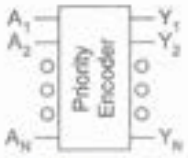


FIG 10.85 Priority encoder

$$
\begin{aligned}
Y_1 &= A_1 \\
Y_2 &= A_2 \cdot \overline{A_1} \\
Y_3 &= A_3 \cdot \overline{A_2} \cdot \overline{A_1} \\
&\ldots \\
Y_N &= A_N \cdot \overline{A_{N-1}} \cdot \ldots \cdot \overline{A_1}
\end{aligned}
\tag{10.47}
$$

We can express priority encoding as a prefix operation by defining a prefix X_{ij} indicating that none of the inputs $A_i...A_j$ are asserted. Then priority encoding can be defined with bitwise precomputation, group logic, and output logic with $i \geq k > j$:

$$
\begin{aligned}
X_{i:i} &= \overline{A_i} && \text{bitwise precomputation} \\
X_{i:j} &= X_{i:k} \cdot X_{k-1:j} && \text{group logic} \\
Y_i &= A_i \cdot X_{i-1:1} && \text{output logic}
\end{aligned}
\tag{10.48}
$$

Any of the group networks (e.g., ripple, skip, lookahead, select, increment, tree) discussed in the addition section can be used to build the group logic to calculate the $X_{i:0}$ prefixes. Short priority encoders use the ripple structure. Medium-length encoders may use a skip, lookahead, select, or increment structure. Long encoders use prefix trees to obtain log N delay. Figure 10.86 shows four 8-bit priority encoders illustrating the different group

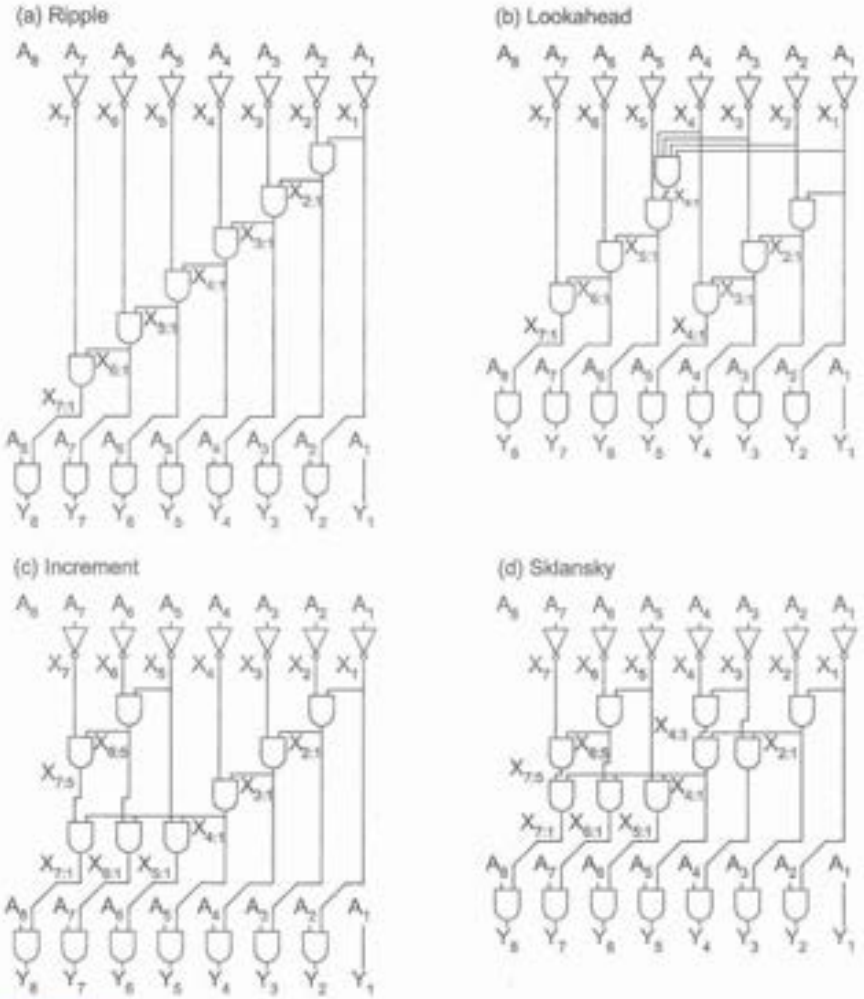


FIG 10.86 Priority encoder trees

logic. Each design uses an initial row of inverters for the $X_{i:i}$ precomputation and a final row of AND gates for the Y_i output logic. In between, ripple, lookahead, increment, and Sklansky networks form the prefixes with various tradeoffs between gate count and delay. Compare these trees to Figure 10.16, Figure 10.27, Figure 10.30, and Figure 10.34(b), respectively. [Wang00j, Delgado-Frias00, Huang02] describe a variety of priority encoder implementations.

An *incrementer* can be constructed in a similar way. Adding 1 to an input word consists of finding the least significant 0 in the word and inverting all the bits up to this point. The X prefix plays the role of the propagate signal in an adder. Again, any of the prefix networks can be used with varying area-speed tradeoffs.

$$\begin{aligned} X_{i:i} &= A_i && \text{bitwise precomputation} \\ X_{i:j} &= X_{i:k} \cdot X_{k-1:j} && \text{group logic} \\ Y_i &= A_i \oplus X_{i-1:1} && \text{output logic} \end{aligned} \tag{10.49}$$

Decrementers and *2's complement* circuits are also similar [Hashemian92]. The decrementer finds the least significant 1 and inverts all the bits up to this point. The 2's complement circuit negates a signed number by inverting all the bits above the least significant 1.

In a slightly more complicated example, consider a modified priority encoder that finds the first two 1's in a string of binary numbers. This might be useful in a cache with two write ports that needs to find the first two free words in the cache. We will use two prefixes: X and W. Again $X_{i:j}$ indicates that none of the inputs $A_i \ldots A_j$ are asserted. $W_{i:j}$ indicates exactly one of the inputs $A_i \ldots A_j$ are asserted. We will produce two 1-hot outputs, Y and Z, corresponding to th

$$\begin{aligned} X_{i:i} &= \overline{A_i} && \\ W_{i:i} &= A_i && \text{bitwise precomputation} \\ X_{i:j} &= X_{i:k} \cdot X_{k-1:j} && \\ W_{i:j} &= W_{i:k} \cdot X_{k-1:j} + X_{i:k} \cdot W_{k-1:j} && \text{group logic} \\ Y_i &= A_i \cdot X_{i-1:1} && \\ Z_i &= A_i \cdot W_{i-1:1} && \text{output logic} \end{aligned} \tag{10.50}$$

10.11 Pitfalls and Fallacies

Equating logic levels and delay

Comparing a novel design with the best existing design is difficult. Some engineers cut corners by merely comparing logic levels. Unfortunately, delay depends strongly on the logical effort of each stage, the fanout it must drive, and the wiring capacitance. For example, [Srinivas92] claims that a novel adder is 20%–28% faster than the fastest known binary lookahead adder, but does not present simulation results. Moreover, it reports some of the speed advantages to three or four significant figures. On closer examination [Dobson95], the adder proves to just be a hybrid tree/carry-select design with some unnecessary precomputation.

Designing circuits with threshold drops

In modern processes, single-pass transistors that pull an output to $V_{DD} - V_t$ are generally unacceptable because the threshold drop (amplified by the body effect) results in an output with too little noise margin. Moreover, when they drive the gate terminals of a subsequent stage, the stage turns partially ON and consumes static power. Many 10-transistor full-adder cells have been proposed that suffer from such a threshold drop problem.

Reinventing adders

There is an enormous body of literature on adders with various tradeoffs among speed, area, and power consumption. The design space has been explored fairly well and many designers (one of the authors included) have spent quite a bit of time developing a "new" adder, only to find that it is only a minor variation on an existing theme. Similarly, a number of recent publications on priority encoders reinvent prefix network techniques that have already been explored in the context of addition.

10.12 Historical Perspective

Given the wide variety of adders in the literature, which ones are used in practice? The carry-ripple adder is compact and fast enough for many noncritical paths. For high-performance microprocessors, however, the adder is often in the critical path in the ALU or the address generation units. In these applications, a tree adder of logarithmic depth is desirable. Ideally, an N-bit tree adder would have log N logic levels, fanout never exceeding 2, and a single wiring track between each logic level. Brent-Kung has extra logic levels. Sklansky has large fanout. Kogge-Stone has more wiring tracks. The best choice depends on the particular implementation technology. For example, in older processes when wires were cheaper than gates, Kogge-Stone was most attractive. In modern processes, the ratio of wire capacitance to gate capacitance has become larger, so designs with fewer tracks are preferred.

The Kogge-Stone and Han-Carlson adders have been widely used on 32-bit and 64-bit microprocessors including the UltraSparc III and several high-speed demonstration units [Silberman98, Heald00, Vangal02]. The Naffziger adder is used throughout the Itanium 2 and Hewlett Packard PA-RISC lines of 64-bit microprocessors [Fetzer02]. The AMD29050 microprocessor used the 64-bit "Redundant Cell" adder [Lynch91], which can be viewed as a hybrid valency-4 tree adder driving 8-bit carry-select multiplexers.

Summary

This chapter has presented a range of datapath subsystems. How one goes about designing and implementing a given CMOS chip is largely affected by the availability of tools, the schedule, the complexity of the system, and the final cost goals of the chip. In general, the simplest and least expensive (in terms of time and money) approach that meets the target goals should be chosen. For many systems, this means that synthesis and place-and-route

is good enough. Modern synthesis tools draw on a good library of adders and multipliers with various area/speed tradeoffs that are sufficient to cover a wide range of applications. For systems with the most stringent requirements on performance or density, custom design at the schematic level still provides an advantage. Domino parallel-prefix trees provide the fastest adders when the high power consumption can be tolerated. Domino CSAs are also used in fast multipliers. However, in multiplier design, the wiring capacitance is paramount and a multiplier with compact cells and short wires can be fast as well as small and low in power.

Exercises

10.1 Design a fast 8-bit adder. The inputs may drive no more than 30 λ of transistor width each and the output must drive a 20/10 inverter. Simulate the adder and determine its delay.

10.2 When adding two unsigned numbers, a carry-out of the final stage indicates an overflow. When adding two signed numbers in 2's complement format, overflow detection is slightly more complex. Develop a Boolean equation for overflow as a function of the carry-out and most significant bit of the output.

10.3 Repeat Exercise 10.2 for a signed add/subtract unit like that shown in Figure 10.48(b). Your overflow output should be a function of the carry-out, most significant output bit, and the sub-input.

10.4 Develop equations for the logical effort and parasitic delay with respect to the C_0 input of an n-stage Manchester carry chain computing $C_1 \ldots C_n$. Consider all of the internal diffusion capacitances when deriving the parasitic delay. Use the transistor widths shown in Figure 10.87 and assume the P_i and G_i transistors of each stage share a single diffusion contact.

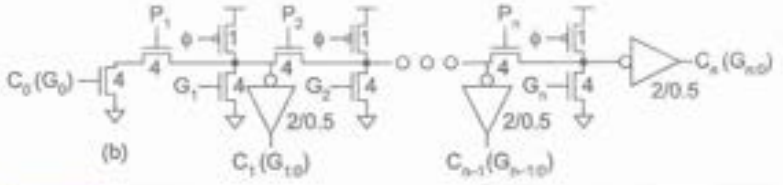


FIG 10.87 Manchester carry chain

10.5 Using the results of Exercise 10.4, what Manchester carry chain length gives the least delay for a long adder?

10.6 The carry lookahead adder in Figure 10.27(d) with variable block size requires five stages of valency-2 group PG cells for 16-bit addition. How many stages are required for 32-bit addition? For 64-bit addition?

10.7 Sketch the PG network for a modified 16-bit Sklansky adder with fanout of [8, 1, 1, 1] rather than [8, 4, 2, 1]. Use buffers to prevent the less-significant bits from loading the critical path.

10.8 Figure 10.34 shows PG networks for various 16-bit adders and Figure 10.35 illustrates how these networks can be classified as the intersection of the $l + f + t = 3$ plane with the face of a cube. The plane also intersects one point inside the cube at $(l, f, t) = (1, 1, 1)$ [Harris03]. Sketch the PG network for this 16-bit adder.

10.9 Sketch a diagram of the group PG tree for a 32-bit Ladner-Fischer adder.

10.10 Write a Boolean expression for C_{out} in the circuit shown in Figure 10.6(b). Simplify the equation to prove that the pass-transistor circuits do indeed compute the majority function.

10.11 Prove EQ (10.28) from EQ (10.27) and EQ (10.29).

10.12 Sketch a design for a comparator computing $A - B = k$.

10.13 Show how the layout of the parity generator of Figure 10.59 can be designed as a linear column of XOR gates with a tree-routing channel.

10.14 Design an ECC decoder for distance-3 Hamming codes with $c = 3$. Your circuit should accept a 7-bit received word and produce a 4-bit corrected data word. Sketch a gate-level implementation.

10.15 How many check bits are required for a distance-3 Hamming code for 8-bit data words? Sketch a parity-check matrix and write the equations to compute each of the check bits.

10.16 Find the 4-bit binary-reflected Gray code values for the numbers 0–15.

10.17 Design a Gray-coded counter in which only one bit changes on each cycle.

10.18 Table 10.12 and Figure 10.75 illustrated radix-4 Booth encoding using X_p, $2X_p$, and M_p. An alternative encoding is to use *POS*, *NEG*, and *DOUBLE*. *POS* is true for the multiples Y and $2Y$. *NEG* is true for the multiples $-Y$ and $-2Y$. *DOUBLE* is true for the multiples $2Y$ and $-2Y$. Design a Booth encoder and selector using this encoding.

10.19 Adapt the priority encoder logic of EQ (10.50) to produce three 1-hot outputs corresponding to the first three 1's in an input string.

10.20 Sketch a 16-bit priority encoder using a Kogge-Stone prefix network.

10.21 Use logical effort to estimate the delay of the priority encoder from Exercise 10.20. Assume the path electrical effort is 1.

10.22 Write equations for a prefix computation that determines the second location in which the pattern 10 appears in an N-bit input string. For example, 010010 should return 010000.

Array Subsystems 11

11.1 Introduction

Memory arrays often account for the majority of transistors in a CMOS system-on-chip. Arrays may be divided into categories as shown in Figure 11.1. *Programmable Logic Arrays* (PLAs) perform logic rather than storage functions, but are also discussed in this chapter.

Random access memory is accessed with an *address* and has a latency independent of the address. In contrast, *serial access memories* are accessed sequentially so no address is necessary. *Content addressable memories* determine which address(es) contain data that matches a specified *key*.

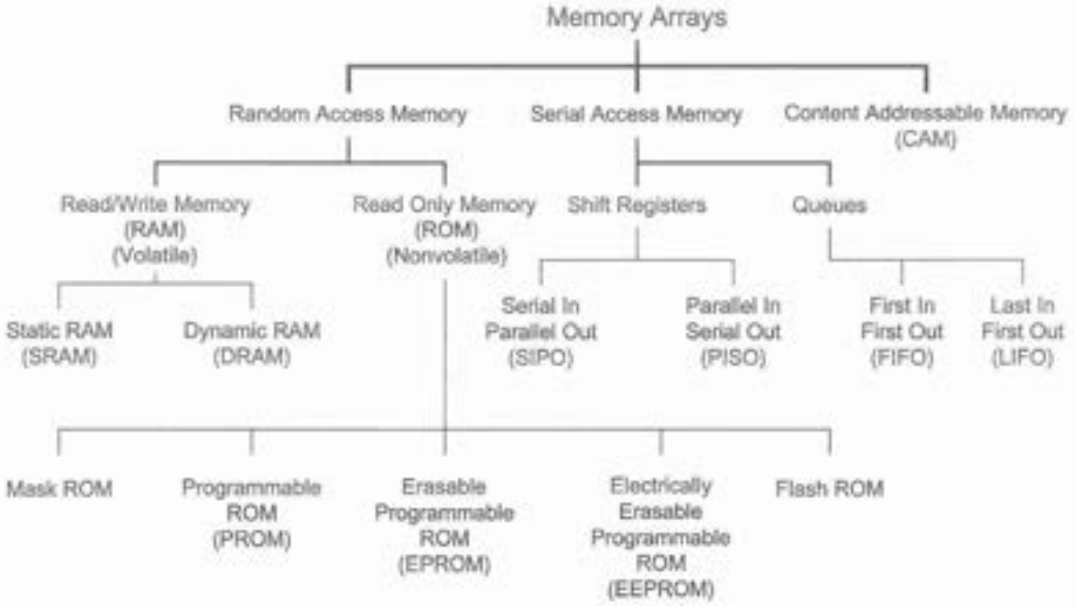


FIG 11.1 Categories of memory arrays

Random access memory is commonly classified as *read-only memory* (ROM) or *read/write memory* (confusingly called RAM). Even the term ROM is misleading because many ROMs can be written as well. A more useful classification is *volatile* vs. *nonvolatile* memory. Volatile memory retains its data as long as power is applied, while nonvolatile memory will hold data indefinitely. RAM is synonymous with volatile memory, while ROM is synonymous with nonvolatile memory.

Like sequencing elements, the memory cells used in volatile memories can further be divided into *static* structures and *dynamic* structures. Static cells use some form of feedback to maintain their state, while dynamic cells use charge stored on a floating capacitor through an access transistor. Charge will leak away through the access transistor even while the transistor is OFF, so dynamic cells must be periodically read and rewritten to refresh their state. Static RAMs (SRAMs) are faster and less troublesome, but require more area per bit than their dynamic counterparts (DRAMs).

Some nonvolatile memories are indeed read-only. The contents of a mask ROM are hardwired during fabrication and cannot be changed. But many nonvolatile memories can be written, albeit more slowly than their volatile counterparts. A *programmable* ROM (PROM) can be programmed once after fabrication by blowing on-chip fuses with a special high programming voltage. An *erasable programmable* ROM (EPROM) is programmed by storing charge on a floating gate. It can be erased by exposure to ultraviolet (UV) light for several minutes to knock the charge off the gate. Then the EPROM can be reprogrammed. *Electrically erasable programmable* ROMs (EEPROMs) are similar, but can be erased in microseconds with on-chip circuitry. *Flash* memories are a variant of EEPROM that erases entire blocks rather than individual bits. Sharing the erase circuitry across larger blocks reduces the area per bit. Because of their good density and easy in-system reprogrammability, Flash memories have replaced other nonvolatile memories in most modern CMOS systems.

Memory cells can have one or more *ports* for access. On a read/write memory, each port can be read-only, write-only, or capable of both read and write.

A typical small memory array architecture is shown in Figure 11.2. Central to the design is a memory array consisting of 2^n *words* of storage of 2^m bits each. In the simplest design, the array is organized with one row per word and one column per bit in each word. Often there are far more words in the memory than bits in each word, which would lead to a very tall, skinny memory that is hard to fit in the chip floorplan and slow because of the long vertical wires. Therefore, the array is often folded into fewer rows of more columns. After folding, each row of the memory contains 2^k words, so the array is physically organized as 2^{n-k} *rows* of 2^{m+k} *columns* or bits. The row decoder activates one of the rows by asserting one of the *wordlines*. During a read operation, the cells on this wordline drive the *bitlines*, which may have been conditioned to a known value in advance of the memory access. The column decoder controls a multiplexer in the column circuitry to select 2^m bits from the row as the data to access. The figure illustrates an 8-word by 4-bit 2-way multiplexed memory folded into a 4-row by 8-column array with $n = 3$, $m = 2$, $k = 1$. Larger

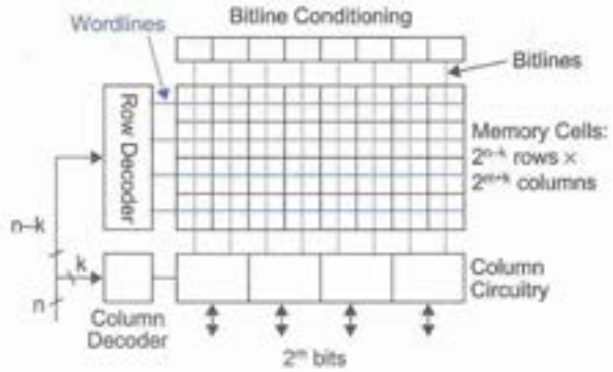


FIG 11.2 General memory array architecture

memories are generally built from multiple smaller subarrays so that the wordlines and bitlines remain reasonably short, fast, and low in power dissipation.

We begin in Section 11.2 with SRAM, the most widely used form of on-chip memory. SRAM also illustrates all the issues of cell design, decoding, and column circuitry design. Subsequent sections address DRAMs, ROMs, serial access memories, CAMs, and PLAs.

11.2 SRAM

The fundamental building block of a static RAM is the SRAM memory cell. The cell is activated by raising the wordline and is read or written through the bitline. Figure 11.3(a) shows a 12-transistor SRAM cell built from a simple static latch and tristate inverter. The cell has a single bitline. True and complementary read and write signals are used in place of a single wordline. A representative layout in Figure 11.3(b) has an area of 46 × 75 λ. The power and ground lines can be shared between mirrored adjacent cells, but the area is still limited by the wires and is undesirably large. However, the cell is easy to design because all nodes swing rail-to-rail and it is fast when used in small RAMs and register files.

Figure 11.4 shows a 6-transistor (6T) SRAM commonly used in practice. Such a cell uses a single wordline and both true and complementary bitlines. The complementary bitline is often called bit_b or $\overline{\text{bit}}$. The cell contains a pair of cross-coupled inverters and an *access transistor* for each bitline. True and complementary versions of the data are stored on the cross-coupled inverters. If the data is disturbed slightly, positive feedback around the loop will restore it to V_{DD} or GND. The wordline is asserted to read or write the cell.

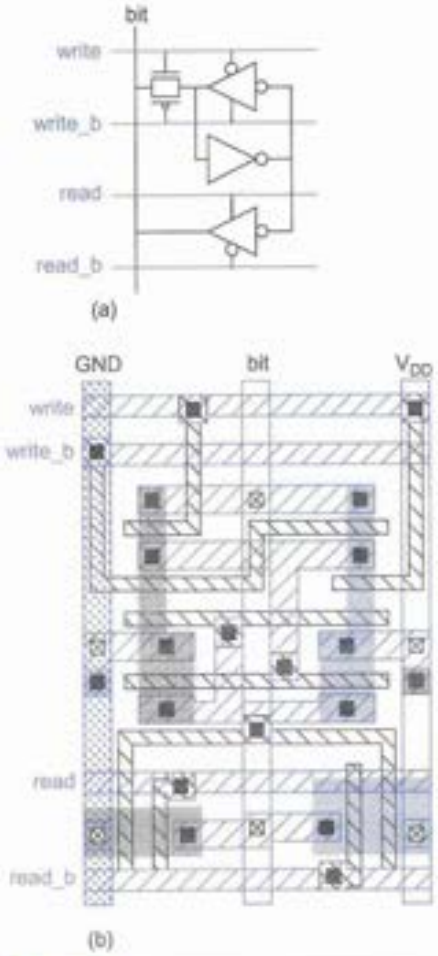


FIG 11.3 12-transistor SRAM cell

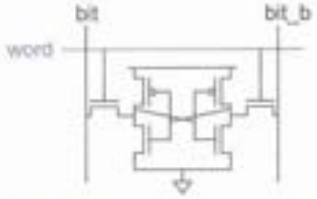


FIG 11.4 6-transistor SRAM cell

Throughout this chapter, wordlines will be highlighted in blue. The nMOS access transistors are best at passing '0's. For reads, the bitlines are initially precharged high and one is pulled down by the SRAM cell through the access transistor. For writes, the bitline or its complement is actively driven low and this low value overpowers the cell to write the new value. Careful choice of transistor sizes is necessary for correct operation, as will be examined in Section 11.2.1. The 6T cell achieves its compactness at the expense of more complex peripheral circuitry for reading and writing the cells. This is a good tradeoff in large RAM arrays where the cell size dominates the area. The small cell size also offers shorter wires and hence lower power consumption.

SRAM cells require clever layout to achieve good density. Figure 11.5(a) shows a stick diagram of a typical design. The cell is designed to be mirrored and overlapped to share V_{DD} and GND lines between adjacent cells along the cell boundary, as shown in Figure 11.5(b). Note how a single diffusion contact to the bitline is shared between a pair of cells. This halves the diffusion capacitance, and hence reduces the delay discharging the bitline during a read access. The wordline is run in both metal1 and polysilicon; the two layers must occasionally be strapped (e.g., every four or eight cells). Sample layouts derived from the stick diagram are shown in Figure 11.6. Figure 11.6(a) shows a conservative cell of 26 × 45 λ, obeying the MOSIS submicron design rules (see also inside front cover). In this layout, the metal1 and polysilicon wordlines are contacted in each cell. The substrate and well are also contacted in each cell. Figure 11.6(b) shows a scanning electron micrograph of the polysilicon and diffusion layers in a more aggressive (20 × 22 λ) cell in the LSI Logic 130 nm process [Kong01w]. SRAM is so important that design rules are very carefully studied and bent where possible to minimize cell area in commercial processes. Notice that diagonal lines are also used. Moreover, each substrate contact can be shared among multiple cells to save area at the expense of regularity.

SRAM operation is divided into two phases. As described in Section 7.4.3, the phases will be called ϕ_1 and ϕ_2, but may actually be generated from *clk* and its complement *clkb*. Assume that in phase 2, the SRAM is precharged. In phase 1, the SRAM is written or read by raising the appropriate wordline and either driving the bitlines to the value that should be written or leaving the bitlines floating and observing which one is pulled down. Reading a large SRAM can be slow because the capacitance of all the cells sharing the bitline is large. Sense amplifiers accelerate reads by detecting small differences between the bitline and its complement. The following sections discuss the role of each block from Figure 11.2 in the SRAM operation.

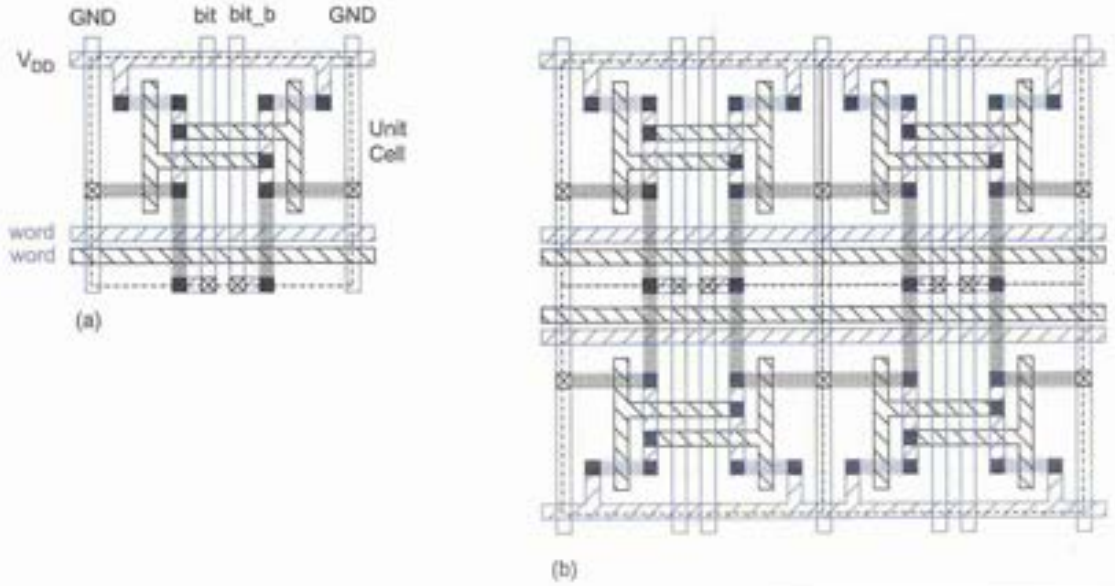


FIG 11.5 Stick diagram of 6T SRAM cell

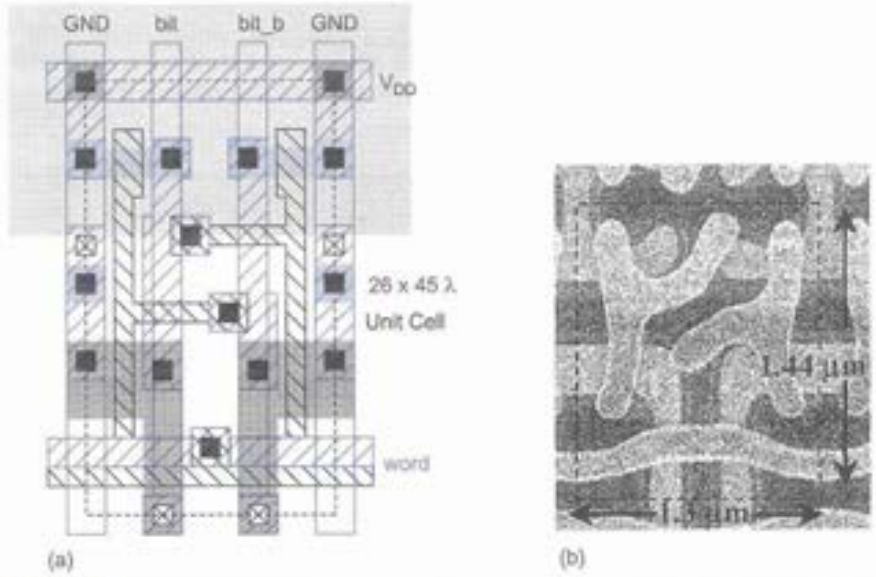


FIG 11.6 Layout of 6T SRAM cell. © IEEE 2001. Color version on inside front cover.

11.2.1 Memory Cell Read/Write Operation

Figure 11.7 shows a SRAM cell being read. The bitlines are both initially floating high. Without loss of generality, assume *A* is initially '0' and thus *A_b* is initially '1.' *A_b* and *bit_b* both should remain '1'. When the wordline is raised, *bit* should be pulled down through transistors *N1* and *N2*. At the same time *bit* is being pulled down, node *A* tends to rise. *A* is held low by *N1*, but raised by current flowing in from *N2*. Hence, *N1* must be stronger than *N2*. Specifically, the transistors must be ratioed such that node *A* remains below the switching threshold of the *P2/N3* inverter. This constraint is called *read stability*. Waveforms for the read operation are shown in Figure 11.7(b) as a 0 is read onto *bit*. Observe that *A* momentarily rises, but does not glitch badly enough to flip the cell.

Figure 11.8 shows the same cell in the context of a full column from the SRAM. During phase 2, the bitlines are precharged high. The wordline only rises during phase 1; hence, it can be viewed as a _q1 qualified clock. Many SRAM cells share the same bitline pair, which acts as a distributed dual-rail footless dynamic multiplexer. The capacitance of the entire bitline must be discharged through the access transistor. The output can be sensed by a pair of HI-skew inverters. By raising the switching threshold of the sense inverters, delay can be reduced at the expense of noise margin. The outputs are dual-rail monotonically rising signals, just as in a domino gate.

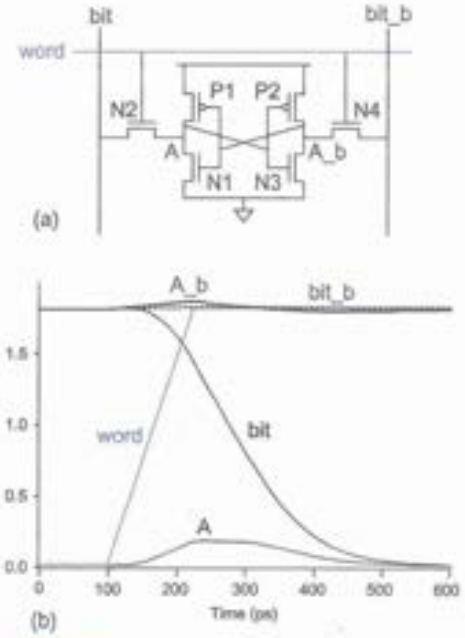


FIG 11.7 Read operation for 6T SRAM cell

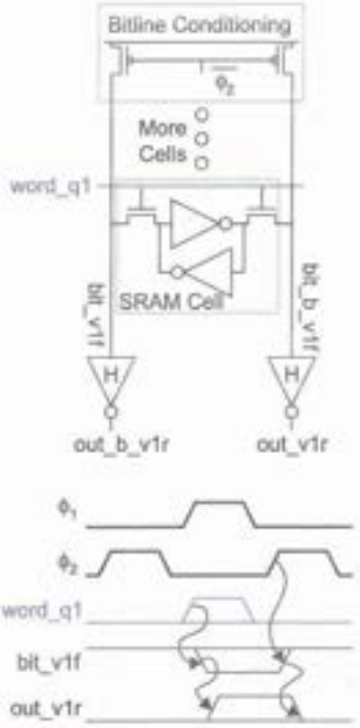


FIG 11.8 SRAM column read

The waveforms of Figure 11.9 show the SRAM cell being written. Again, assume *A* is initially '0' and that we wish to write a '1' into the cell. *bit* is precharged high and left floating. *bit_b* is pulled low by a write driver. We know on account of the read stability constraint that *bit* will be unable to force *A* high through *N2*. Hence, the cell must be written by forcing *A_b* low through *N4*. *P2* opposes this operation; thus, *P2* must be weaker than *N4* so that *A_b* can be pulled low enough. This constraint is called *writeability*. Once *A_b* falls low, *N1* turns OFF and *P1* turns ON, pulling *A* high as desired.

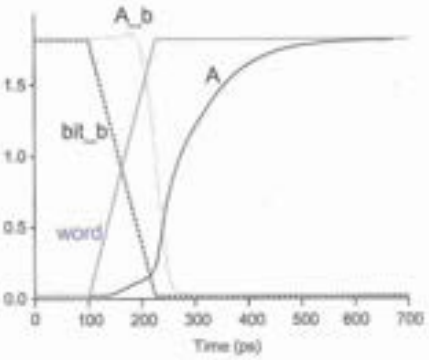


FIG 11.9 Write operation for 6T SRAM cell

Figure 11.10(a) again shows the cell in the context of a full column from the SRAM. During phase 2, the bitlines are precharged high. Write drivers pull the bitline or its complement low during phase 1 to write the cell. The write drivers can consist of a pair of transistors on each bitline for the data and the write enable, or a single transistor driven by the appropriate combination of signals (Figure 11.10(b)). In either case, the series resistance of the write driver, bitline wire, and access transistor must be low enough to overpower the pMOS transistor in the SRAM cell.

In summary, to ensure both read stability and writeability, the nMOS pull-down transistor in the cross-coupled inverters must be strongest. The access transistors are of intermediate strength, and the pMOS pull-up transistors must be weak. To achieve good layout density, all of the transistors must be relatively small. In Figure 11.6(a), the pull-downs are 8/2 λ, the access transistors 4/2, and the pull-ups 3/3. The SRAM cells must operate correctly in all process corners at all voltages and temperatures. This requires thorough simulation.

It is no longer common for designers to develop their own SRAM cells. Usually, the fabrication vendor will supply cells that are carefully tuned to the particular manufacturing process. In high-performance processes, two or more cells may be provided with different speed/density tradeoffs. [Glasser85] describes SRAM cells with four transistors and two large resistors, but the 6T cell is almost universally used in contemporary standard CMOS processes.

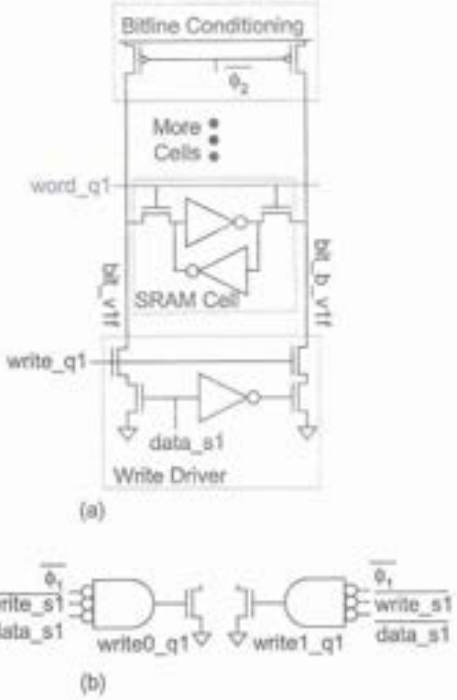


FIG 11.10 SRAM column write

11.2.2 Decoders

The simplest decoder is a collection of AND gates using true and complementary versions of the address bits. Figure 11.11 shows several straightforward implementations. The first implementation in Figure 11.11(a) is a static NAND gate followed by an inverter. This structure is useful for up to 5–6 inputs or more if speed is not critical. The NAND transistors are usually made minimum size to reduce the load on the buffered address lines because there are 2^{n-1} transistors on each true and complementary address line in the row

decoder. The second implementation shown in Figure 11.11(b) uses a pseudo-nMOS NOR gate buffered with two inverters. The NOR gate transistors can be made minimum size and the inverters can be scaled appropriately to drive the wordline.

The layout of the decoder must be pitch-matched to the memory array, i.e., the height of each decoder gate must match the height of the row it drives. This can be tricky for SRAM and even harder for ROMs and other arrays with small memory cells. Figure 11.12(a) shows a layout of a conventional standard-cell style approach. The minimum-sized transistors in the NAND gate drive a larger buffer inverter. The decoder height grows with the number of inputs. The AND gates are easily programmed by connecting the polysilicon inputs to the appropriate address inputs. Figure 11.12(b) shows a layout on a pitch that is tighter and independent of the number of inputs. The decoder is programmed by placement of transistors and metal straps; this is best done with scripting software that generates layout, as discussed in Section 8.3.5. The polysilicon address lines should be strapped with metal2 to reduce their resistance, but the metal2 is left out of the figure for readability. The decoder pitch is 5 tracks or 40 λ. If every other row is mirrored to share V_{DD} and GND, the pitch can be reduced to 4 tracks or 32 λ.

11.2.2.1 Predecoding Decoders with many inputs can be formed from a cascade of smaller gates. For example, Figure 11.13(a) shows a 16-word decoder in which the 4-input AND function is built from a pair of 2-input NANDs followed by a 2-input NOR.

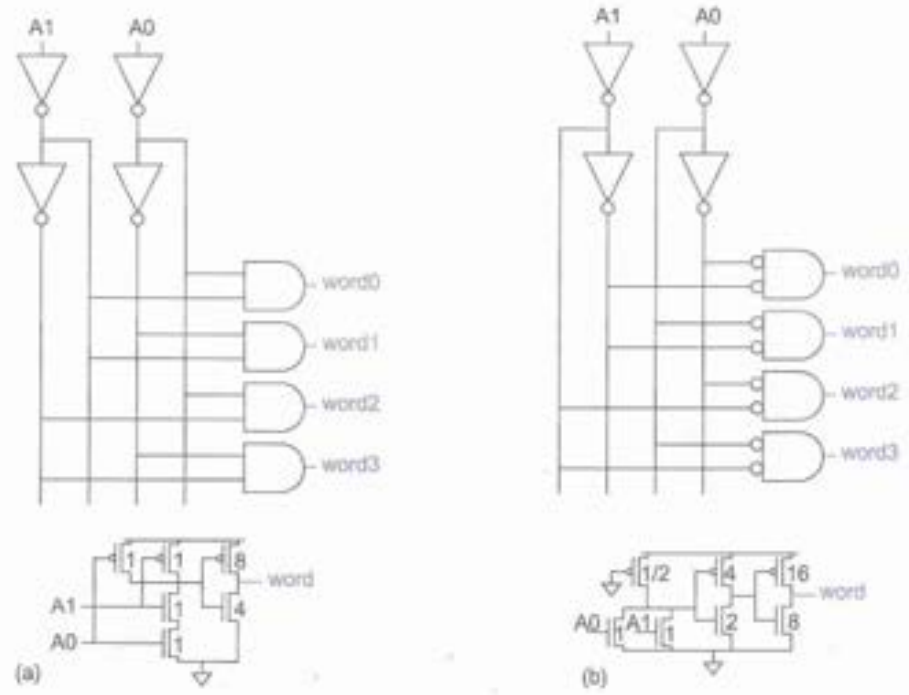


FIG 11.11 Decoders

Many NAND gates share exactly the same inputs and are thus redundant. The decoder area can be improved by factoring these common NANDs out, as shown in Figure 11.13(b). This technique is called *predecoding*. It does not change the path effort of the decoder, but does improve area. In general, blocks of p address bits can be predecoded into 1-of-2^p-hot predecoded lines that serve as inputs to the final stage decoder. For example, Figure 11.13(b) shows a $p = 2$-bit design that decodes each pair of address bits into a 1-of-4-hot code.

The wordline generally must be qualified with the clock for proper bitline timing. This is often performed with another AND gate after the decoder or with an extra clk input to the final stage of decoding.

11.2.2.2 Faster Decoders The logical effort of a decoder can be reduced by observing that only one of the outputs will be high so the pMOS transistors can be shared among many outputs [Lyon87]. A NOR gate pulls low efficiently through parallel transistors, but

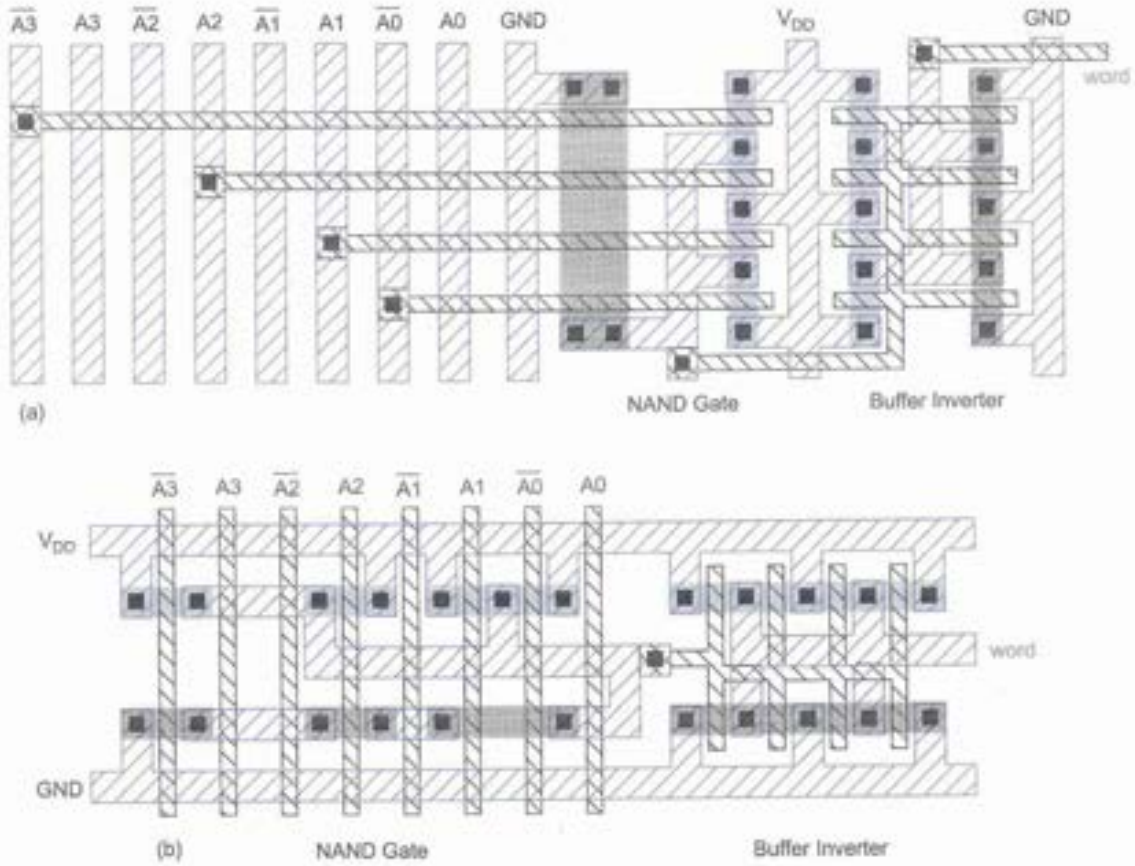


FIG 11.12 Stick diagrams of two decoder layouts

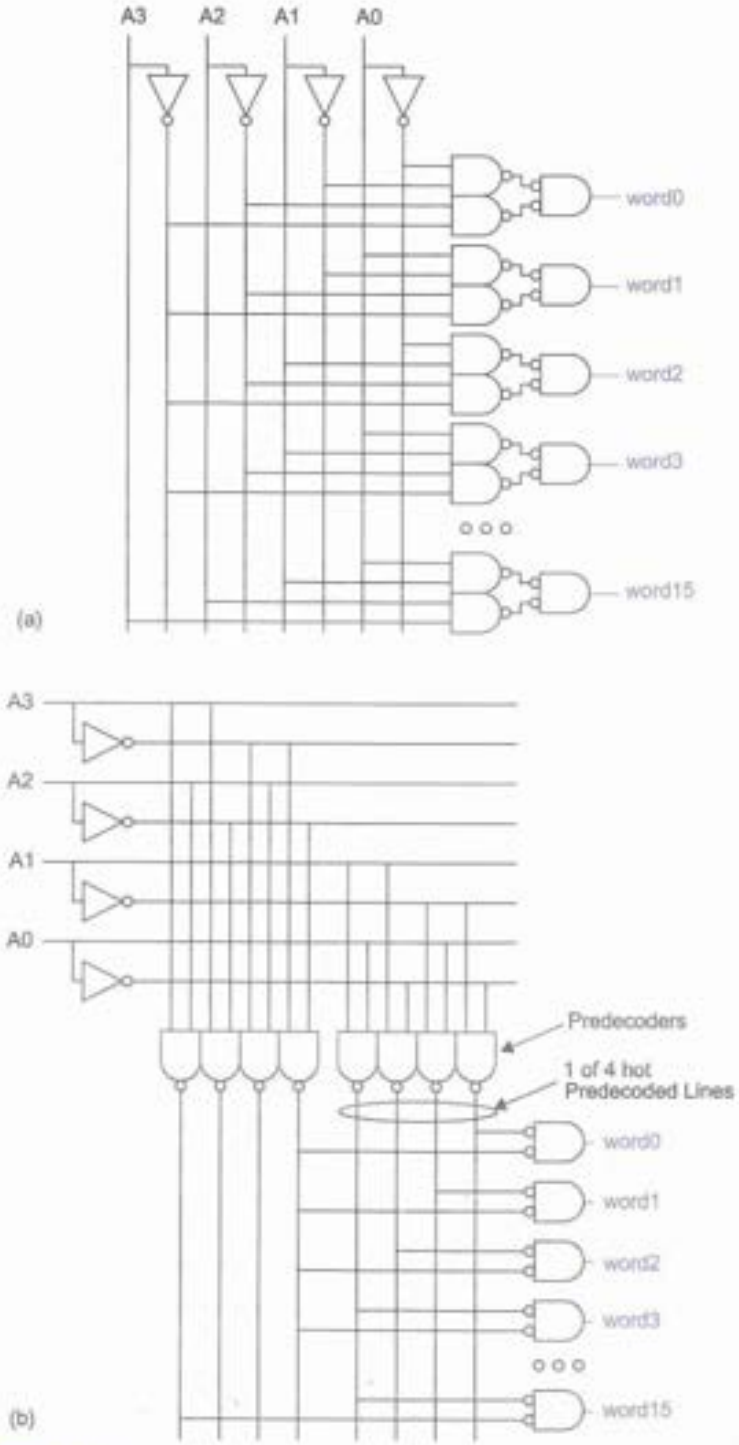


FIG 11.13 Ordinary and predecoding circuits

has a poor logical effort because the output is pulled high through wide series transistors. The Lyon-Schediwy decoder can be viewed as 2^n n-input NOR gates sharing pMOS pull-ups, as shown in Figure 11.14 for a 3:8 decoder. The cost of the wide transistors is amortized across many outputs. Relative transistor widths are chosen to present the same capacitance to each input while providing current drive equal to a unit inverter. The logical effort of each input is only $(1 + 3.5)/3 = 1.5$ per wordline output, as compared to 7/3 for an ordinary NOR3 (see Exercise 11.4).

Decoders typically have high electrical and branching effort. Therefore, they need many stages, so the fastest design is the one that minimizes the logical effort. A tree of 2- and 3-input NAND gates and inverters offers the lowest logical effort to build high fan-in gates in static CMOS [Sutherland99]. Dynamic gates are attractive for fast decoders because they have lower logical effort.

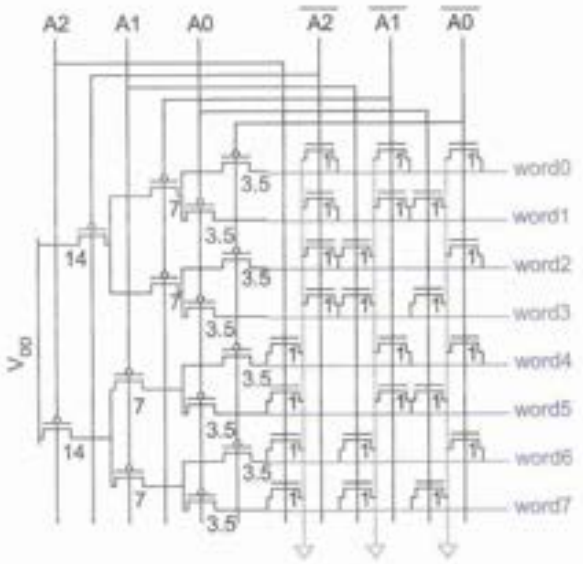


FIG 11.14 Lyon-Schediwy decoder

Example

Estimate the delays of 8:256 decoders using static CMOS and footed domino gates. Assume the decoder has an electrical effort of $H = 10$ and that both true and complementary inputs are available.

Solution: The decoder consists of 256 8-input AND gates. It has a branching effort of $B = 256/2 = 128$ because each of the true inputs and each of the complementary inputs are used by half the gates. Assuming the logical effort of the path G is close to 1, the path effort is $F = GBH = 1280$ and the best number of stages is

$\log_4 F = 5.16$. Let us consider a six-stage design using three levels of 2-input AND gates, each constructed from a 2-input NAND and an inverter.

The static CMOS design has a logical effort of $G = [(4/3) \cdot (1)]^3 = 64/27$. Therefore, the stage effort is $F = 3034$. The parasitic delay is $P = 3 \cdot (2 + 1) = 9$. The total delay is $D = NF^{1/N} + P = 31.8\ \tau$ or 6.4 FO4 inverter delays.

The footed domino design using HI-skew inverters has a logical effort of $[(1) \cdot (5/6)]^3 = 125/256$ and a stage effort of 625. The parasitic delay is $P = 3 \cdot (4/3 + 5/6) = 6.5$. The total delay is 4.8 FO4 inverter delays. In general, domino decoders are about 33% faster than static CMOS.

A major problem with traditional domino decoders is the high power consumption. For example, even though only one of the 256 wordlines in the previous example will rise on each cycle, all 256 AND gates must precharge so the clock load is extremely large. A much lower-power approach is to use self-resetting domino gates that only precharge the wordline that evaluated. Section 7.5.2.4 describes some of these self-resetting gates and [Amrutur01] shows some variations that work with long input pulses. Self-resetting domino has essentially the same performance as traditional domino because it uses the same basic gates.

Yet another approach for dynamic decoders is to use wide NOR structures in which N–1 of the N outputs discharge on each cycle. As most memories require monotonically rising outputs but the NORs are monotonically falling, such decoders require the race-based nonmonotonic techniques described in Section 7.5.4.3. For example, Figure 11.15 shows a 4-input AND gate with monotonically rising output using a race-based NOR structure [Nambu98]. This technique is slightly faster than a domino AND tree, but dissipates more power because the dynamic node X must be precharged on each cycle [Amrutur01].

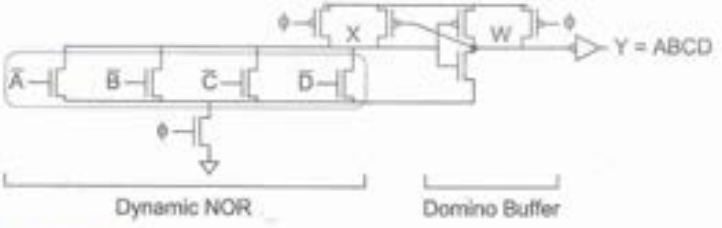


FIG 11.15 4-input AND using race-based NOR

11.2.2.3 Sum-addressed Decoders Many microprocessor instruction sets include addressing modes in which the effective address is the sum of two values, such as a base address and an offset. In conventional SRAMs used as caches, the two values must first be

added, and then the result decoded to determine the cache wordline. If access latency needs to be minimized, these two steps can be combined into one in a *sum-addressed memory* [Heald98].

Recall from Section 10.4.3 that checking if $A + B = K$ is faster than actually computing $A + B$ because no carry propagation need occur. A *sum-addressed decoder* for an N-word memory accepts two inputs, A and B. In a simple form, it contains N comparators driving the N wordlines. The first checks if $A + B = 0$. The second checks if $A + B = 1$, and so forth. The comparators contain redundant logic repeated across wordlines. [Heald98] shows how to reduce the area by factoring out common terms in a predecoder.

11.2.3 Bitline Conditioning and Column Circuitry

The bitline conditioning circuitry is used to precharge the bitlines high before operation. A simple conditioner consists of a pair of pMOS transistors, as shown in Figure 11.16(a). It is also possible to construct pseudo-nMOS SRAMs with weak pull-up transistors in place of the precharge transistors (Figure 11.16(b)) where no clock is available. Another technique is to precharge through nMOS transistors to $V_{DD} - V_t$ (Figure 11.16(c)). This results in faster single-ended bitline sensing because the bitlines do not swing as much, but reduces noise margins and may require more precharge time.

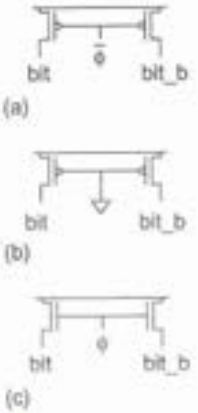


FIG 11.16 Bitline conditioning circuits

Each column must also contain write drivers and read sensing circuits. Figure 11.10 showed two examples of write drivers and Figure 11.8 showed the use of HI-skew inverters to sense reads. Many *sense amplifiers* have been invented to provide faster sensing by responding to a small voltage swing. The differential sense amplifier in Figure 11.17(a) is based on an analog differential pair and requires no clock. The differential gain of the amplifier is $g_{mN1}(r_{oN1} \parallel r_{oP1})$, as will be discussed in Section 12.6.5. However, the circuit consumes a significant amount of DC power. The clocked sense amplifier in Figure 11.17(b) only consumes power while activated, but requires a timing chain to activate at the proper time. When the sense clock is low, the amplifier is inactive. When the sense amplifier rises, it effectively turns on the cross-coupled inverter pair, which pulls one output low and the other high through *regenerative feedback*. The *isolation transistors* speed up the response by disconnecting the outputs from the highly capacitive bitlines during sensing. See Section 6.4.2 for more discussion of sense amplifier circuits. Power dissipation can be reduced for read operations by turning off the wordlines once sufficient differential voltage has been achieved on the bitlines. This reduces the bitline swing and hence the charge required to restore the bitlines to V_{DD} after sensing. *Current sense amplifiers* that detect a differential current rather than a small voltage swing are another promising technique [Wicht01].

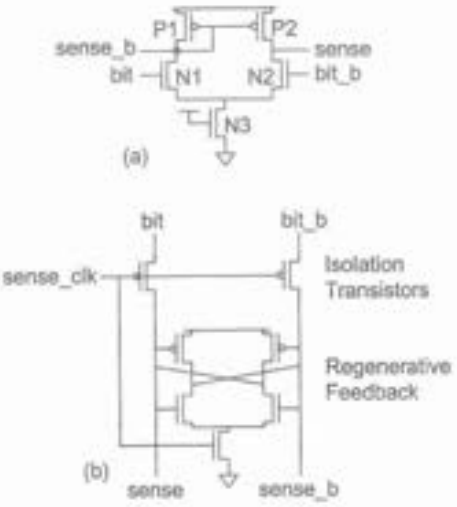


FIG 11.17 Sense amplifiers

Sense amplifiers are very susceptible to differential noise on the bitlines because they detect small voltage differences. If bitlines are not precharged long enough, residual voltages on the lines from the previ-

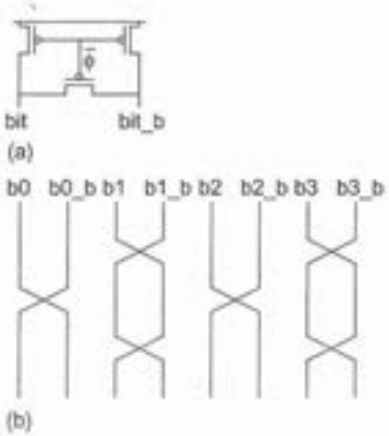


FIG 11.18 Bitline noise reduction through equalizers and twisting

ous read may cause pattern-dependent failure. An equalizer transistor (Figure 11.18(a)) can be added to the bitline conditioning circuits to reduce the required precharge time by ensuring that *bit* and *bit_b* are at nearly equal voltage levels even if they have not precharged quite all the way to V_{DD}. Coupling from transitioning bitlines in neighboring cells may also introduce noise. The bitlines can be *twisted* or *transposed* to cause equal coupling onto both the bitline and its complement, as shown in Figure 11.18(b). For example, careful inspection shows that *b1* couples to *b0_b* for the first quarter of its length, *b2* for the next quarter, *b2_b* for the third quarter, and *b0* for the final quarter. *b1_b* also couples to each of these four aggressors for a quarter of its length, so the coupling will be the same onto both lines.

The sense amplifier offset voltage is the differential input voltage (*bit* − *bit_b*) necessary to produce zero differential output voltage (*sense* − *sense_b*). If *N1* is identical to *N2* and *P1* to *P2*, the sense amplifier will ideally have zero offset voltage. In practice, the offset voltage is nonzero because of statistical dopant fluctuations that affect V_t. The differential input must substantially exceed the offset voltage to be sensed reliably. A typical budget for offset voltage is 50 mV [Amrutur00]. Unfortunately, the threshold variations and offset voltage are not changing very much with technology scaling, so the offset voltage is becoming a larger fraction of the supply voltage, making sense amplifiers less effective [Mizuno94].

Clocked sense amplifiers must be activated at just the right time. If they fire too early, the bitlines may not have developed enough voltage difference to operate reliably. If they fire too late, the SRAM is unnecessarily slow. The clock is generated by circuitry that must match the delay of the decoder, wordlines, and bitlines. This leads to all of the delay matching challenges discussed in Section 7.5.4.1. Many arrays use a chain of inverters, but inverters do not track the delay of the access path very well across process and environmental corners: A margin of more than 30% is often necessary in the typical corner for reliable operation in all corners. Alternatively, the array may use dummy or *replica* cells and bitlines to more closely track the access path. For example, [Amrutur98] describes a 16-kbit SRAM array that eliminates 3 FO4 inverter delays of margin by replacing the inverter chain with an extra replica row and column configured to operate no faster than the slowest real access.

In general, 2^k:1 column multiplexers may be required to extract 2^m bits from the 2^{m+k} bits of each row. The column multiplexers can either act as their own tree decoder or require a separate column decoder to generate select signals. Figure 11.19 shows an 8:1 column multiplexer using a tree decoder. The data is routed through pass transistors enabled by the column address lines. The address decoding is in essence distributed. Decoders for both polarities of the bitline are shown, although one of these can be omitted for single-ended read operations. The read (and usually of lesser importance, write) operations are somewhat delayed by the series pass transistors. Figure 11.20 shows a single-ended 4:1 column mux using a separate column decoder. The multiplexer is faster because data from the bitline must propagate through only one series transistor. The column decoding takes place in parallel with row decoding so it does not impact delay. In both

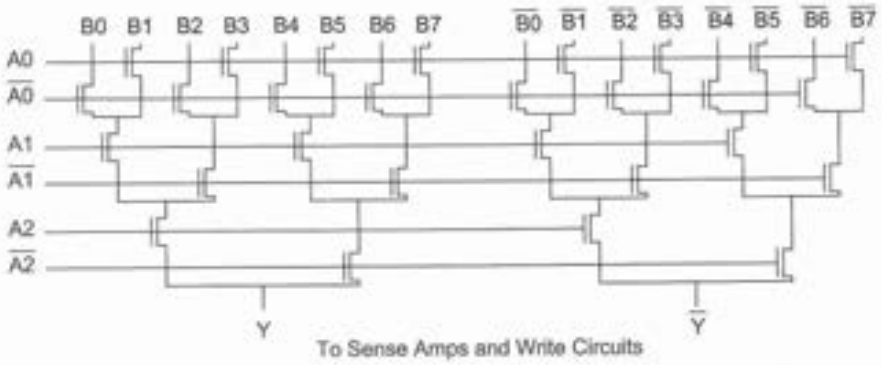


FIG 11.19 Tree decoder column multiplexer

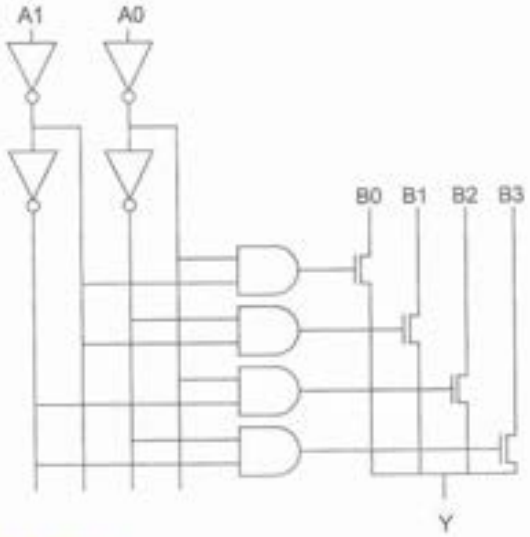


FIG 11.20 Column multiplexer with separate decoder

cases, the outputs may need to be precharged because only nMOS pass transistors are used. Column multiplexers can also use full transmission gates. pMOS pass transistors can be used if a sense amplifier responds to voltages near V_{DD}.

Figure 11.21 shows a complete pair of bits and associated column circuitry for a 2-way multiplexed SRAM. The output of the nMOS-only multiplexer is precharged high. Both the write drivers and the read sensing inverter are connected to the multiplexer outputs.

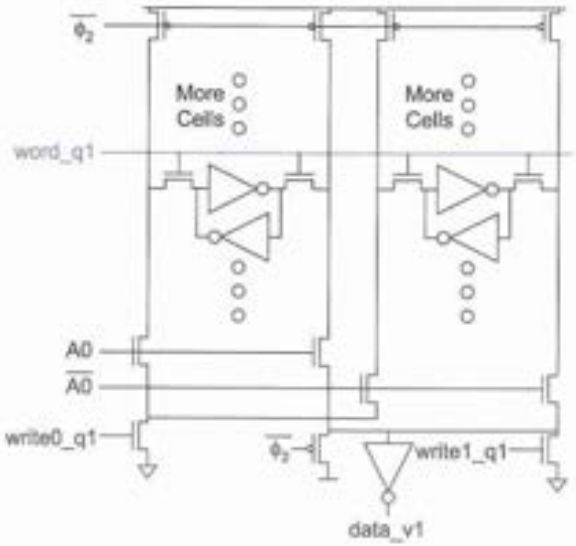


FIG 11.21 Complete pair of columns for 2-way multiplexed SRAM

Column multiplexing is also helpful because the bit pitch of each column is so narrow that it can be difficult to lay out a sense amplifier for each column. After multiplexing, multiple columns are available for the remainder of the column circuitry. Moreover, placing sense amplifiers after the column multiplexers reduces the number of power-hungry amplifiers required in the array.

11.2.4 Multi-ported SRAM and Register Files

Register files are generally fast SRAMs with multiple read and write ports. Data caches in superscalar microprocessors often require multiple ports to handle multiple simultaneous loads and stores.

A simple dual-ported SRAM adds a second wordline, as shown in Figure 11.22 [Horowitz87]. Such a cell can perform two reads or one write in each cycle. The reads are performed by independently selecting different words with the two wordlines. Read becomes a single-ended operation; one read appears on *bit*, while the other appears in complementary form on *bit_b*. For example, asserting *wordA*[7] and *wordB*[3] reads the third word onto *bit* and the complement of the seventh onto *bit_b*. Write still requires both *bit* and *bit_b*, so only a single write can occur. With careful timing, accesses can be performed each half-cycle, permitting two reads in the first phase and a write in the second phase, as commonly required for pipelined processors.

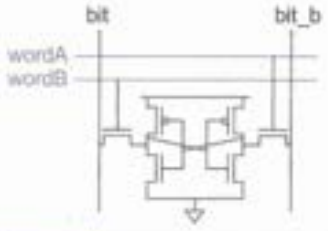


FIG 11.22 Simple dual-ported SRAM

A more general multi-ported SRAM with three write ports and four read ports is shown in Figure 11.23. Merely adding more access transistors causes problems for read stability when multiple ports attempt to read the same cell because all

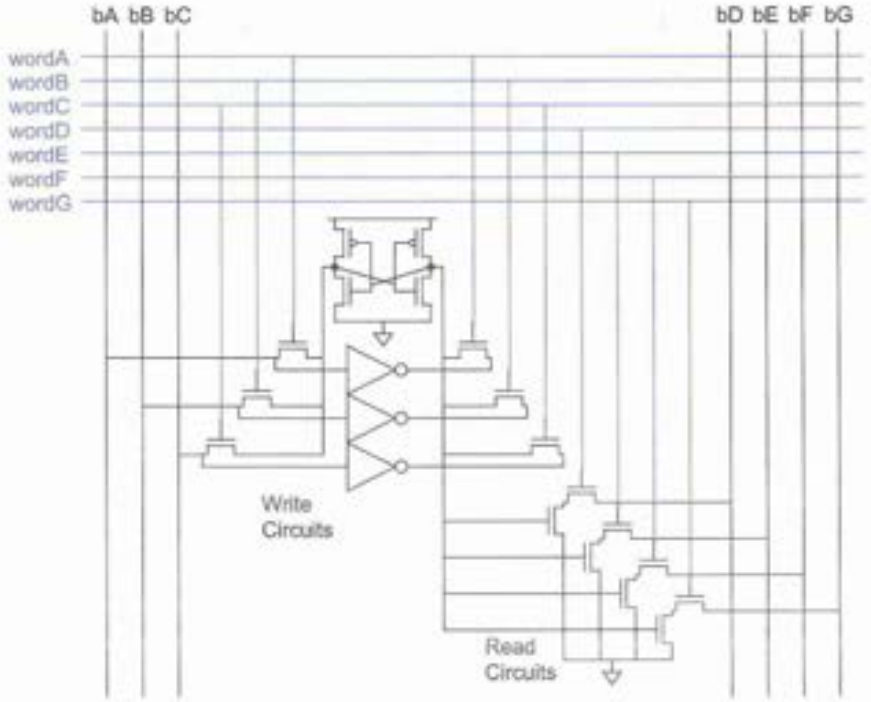


FIG 11.23 General multi-ported SRAM

the access transistors will be driving the cell high. Instead, the bitlines are isolated from the cell during reads. The multi-port SRAM cell still is built around the cross-coupled inverters in the center. Ports *A*, *B*, and *C* perform writes by driving a value onto one side of the inverter pair and its complement onto the other side. Ports *D*, *E*, *F*, and *G* perform reads by discharging the bitline if the cell stores a 0.

Register files for superscalar processors often require an enormous number of ports. For example, the Itanium 2 processor issues up to six integer instructions in a cycle, each of which requires two source registers and a destination. The register file requires two more write ports for late cache data returns, leading to a total of 12 read ports and 8 write ports [Fetzer02]. The area of the large register file is dominated by the mesh of wordlines and bitlines. A rough rule for estimating multiport SRAM cell area is to count the number of tracks for the wordlines and bitlines and then add three in each dimension for internal wiring. The number of bitlines can be reduced by performing single-ended read operations. Similarly, writes can use a local inverter placed in empty space under the wires so only one, rather than two bitlines are required for write ports. The Itanium 2 further reduces the area to only 12 wordlines by time-multiplexing, reading in one half-cycle, and then writing on the other using pulses on the wordlines. [Golden99] and [Heald00] show other designs for the large register files of the AMD Athlon and Sun UltraSparc III, respectively.

A single-ended register file with 16 read ports and 4 write ports has a cell size of 23 × 23 tracks, or about 184 × 184 λ = 33856 λ^2. The area can be improved by partitioning the register file into two parts, each with 8 read ports and 4 write ports. Write operations update both register files so the data remains consistent, but half the reads use one register file and half use the other. The cell size is now 15 × 15 tracks with an area of 14400 λ^2 per file, or 28800 λ^2 all together. The partitioned register file is not only smaller but also faster because of the shorter bitlines and wordlines.

11.2.5 Large SRAMs

The critical path in a static RAM read cycle includes the clock to address delay time, the row address driver time, row decode time, bitline sense time, and the setup time to any data register. The write operation is usually faster than the read cycle because the bitlines are actively driven by large transistors. However, the bitlines may have to be allowed to recover to their quiescent values before any more access cycles take place.

If the memory array becomes large, the wordlines and bitlines become rather long. The long lines have high capacitance, leading to long delay and high power consumption. Thus, large memories are partitioned into multiple smaller memory arrays, sometimes called *banks* or *subarrays*. Each subarray presents some area overhead for its periphery circuitry, so the size of the subarrays represents a tradeoff between area and speed.

Reading data onto a bitline is much like activating a wide footless dynamic multiplexer because the bitline is pulled down through two series transistors. The wordline transistor acts as the select and the nMOS in the cross-coupled inverter acts as the data signal. Recall that a dynamic multiplexer has a constant logical effort but a parasitic delay proportional to the number of inputs (i.e., words). Without sense amplifiers, the bitlines become quite slow for more than 32 words. Sense amplifiers permit smaller bitline swings, reducing the parasitic delay. With sense amplifiers, bitlines reasonably accommodate up to about 128 words.

The wordline presents an RC delay from the resistance of the wire and gate capacitance of the transistors it drives. This increases with the square of the number of bits on a wordline. Typical SRAMs use up to about 256 bits on each wordline.

Figure 11.24 shows a typical large SRAM partitioned into S 4-kbyte (128-word × 256-bit) subarrays [Amrutur00]. The *divided wordline* decoder operates in three stages: predecoding, global wordline decoding, and local wordline decoding [Yoshimoto83]. The 128 global wordlines are long, but lightly loaded and can use wide, upper-level metal wires with low resistance. The local wordline decoder gates the global wordline with a bank select line to activate the appropriate subarray. During a read, the subarray sense amplifiers detect the swing of the bitline and drive the results onto global datalines. Global sense amplifiers detect small swings on these datalines and drive the results out of the SRAM. During a write, the datalines and bitlines are driven in reverse. Subarrays with many words can also use *divided bitlines* to reduce the diffusion capacitance and improve the delay and power of the subarray.

Large memories with multiple subarrays can simulate more than one access port even if each subarray is single-ported. For example, in a system with two subarrays, even-num-

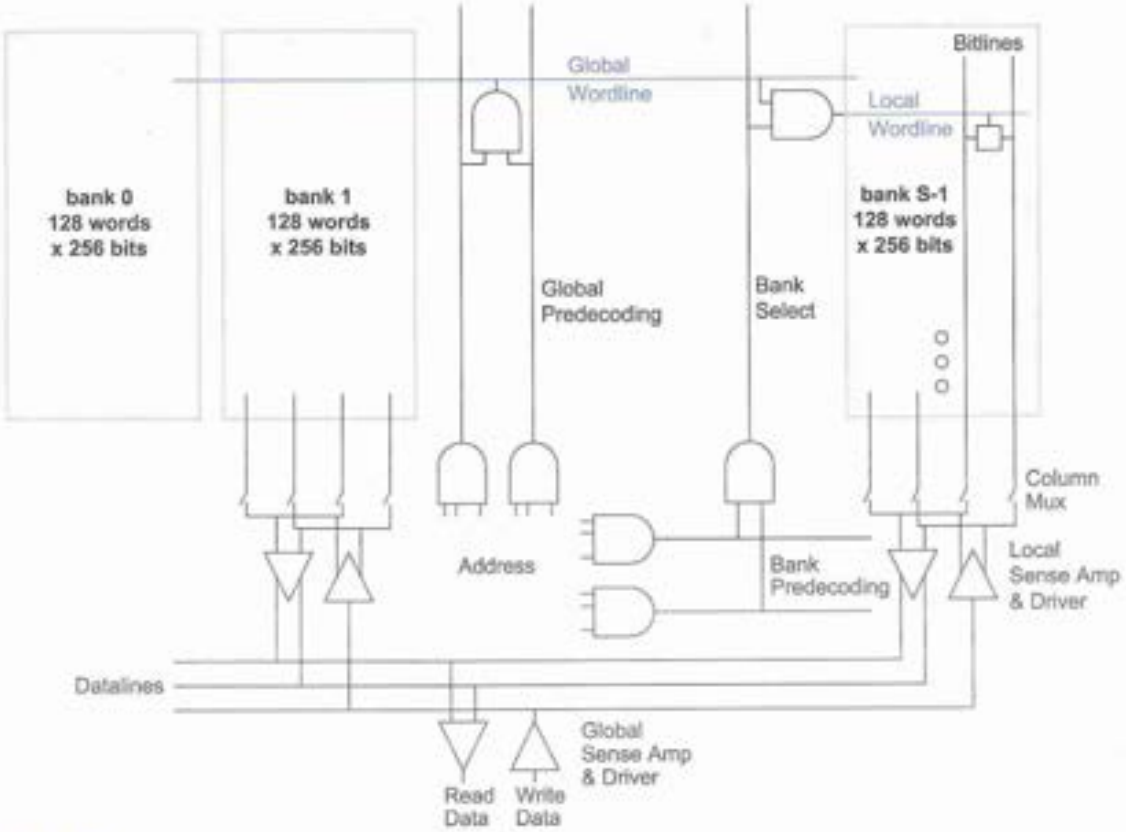


FIG 11.24 Large memory array architecture with subarrays

bered words could be stored in one subarray while odd-numbered words are stored in the other. Two accesses could occur simultaneously if one addresses an even word and another an odd word. If both address an even word, we encounter a *bank conflict* and one access must wait. Increasing the number of banks offers more parallelism and lower probability of bank conflicts.

11.2.6 Logical Effort of RAMs and Register Files

The method of logical effort is helpful to estimate the delay of a static RAM or register file. The critical read path for a small single-ported RAM with no column multiplexing involves the decoder to drive the wordline and the SRAM cell that pulls down the bitline. Figure 11.25 highlights this path for a 2^n word by 2^m-bit memory with total storage of 2^N bits ($N = m + n$).

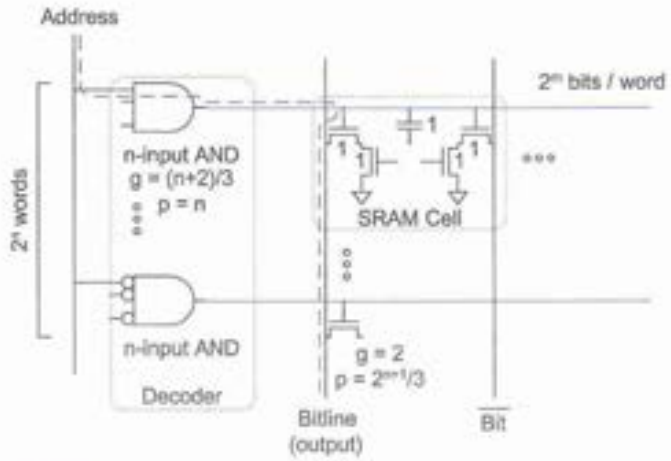


FIG 11.25 Critical path for read of small SRAM

The decoder is modeled as an n-input AND gate taking some combination of true and complemented address inputs. It has a logical effort of $(n + 2)/3$ and parasitic delay of n according to Tables 4.2 and 4.3. The bitline is discharged in the SRAM cell through two series transistors that behave like a dynamic multiplexer. Suppose each cell has two unit-sized access transistors and stray wire capacitance approximately equal to another unit-sized transistor, for a total capacitance of $3C$ presented by each cell to the wordline. Because there are two transistors in series, the cell delivers about half the current of a unit inverter with input capacitance $3C$. Hence, the logical effort is 2 because the cell delivers half the current of an inverter with the same input capacitance. Suppose each cell presents $1C$ of diffusion capacitance on the bitline, so the total bitline capacitance is 2^mC. The cell has an effective resistance of $2R$ discharging the bitline through two series unit transistors. Hence, the bitline has a parasitic delay of $2^{m-1}RC$. Normalized by $\tau = 3RC$, this gives $p = 2^{m-1}/3$.

Putting these two stages together, the path logical effort is $G = (n + 2)/3 \cdot 2$. If the true and complementary bitline outputs each drive capacitance equal to half that seen by the address inputs, the path electrical effort is $H = 1/2$. Within the path are a 2^n-way branch as each address bit is needed by each wordline decoder and another 2^m-way branch as each wordline drives all the bits on that word. Hence, the branching effort is $B = 2^N$. The path effort delay is $F = GBH = 2^N(n + 2)/3$. The parasitic delay is $P = n + 2^{m-1}/3$. The best number of stages is approximately $\log_4 F = N/2 + \log_4[(n + 2)/3]$. These stages would include buffers in the address driver, multiple levels of gates in the decoder, buffers to drive the wordline, and an inverter on the bitline output. The path delay is $D = 4\log_4 F + P = 2^N + 4\log_4[(n + 2)/3] + n + 2^{m-1}/3$. For a 32-word × 32-bit register file, $n = 5$, $N = 10$, and $D = 48.8\ \tau = 9.8$ FO4 inverter delays.

This model is clearly an oversimplification. The n-input AND gate is usually constructed out of a chain of low fan-in gates, but this only slightly improves its logical effort. We also neglect the effort of the clock gating to drive the wordlines on the clock edge. We assume the RAM is small enough that sense amplifiers are not used and that the wire resistance is negligible. The pull-down transistor inside the SRAM cell may be larger than the access transistor. Nevertheless, the model offers insights into the number of stages that the memory should use and its approximate delay. For example, it shows that, without sense amplifiers, putting too many words on a bitline causes excessive parasitic delay.

[Amrutur00] models the delay of large SRAMs using logical effort in substantially more detail than can be repeated here. The overall delay includes components contributed by both the gates and the wire RC. In a well-designed 2^N-bit SRAM ($N \geq 16$) using static CMOS decoders, the gate delay component is approximately $1.2 \cdot N - 4$ FO4 inverter delays. More aggressive decoders using domino or race-based NOR techniques from Section 11.2.2 can reduce this delay by about 15% [Amrutur01]. Wire delay becomes important for RAMs beyond the 1-Mbit ($N = 20$) capacity. A lower bound for wire delay is set by the speed of light at about 1.75 FO4 for 4-Mbit memories. This delay doubles for each quadrupling in memory size. In practice, the wire delay depends on the wire width and thickness and repeater strategy, but can be several times this lower bound. In processes beyond the 100 nm generation, sense amplifiers will need relatively larger bitline swings because their offset voltages are not scaling with the supply voltage. This will add several FO4 inverter delays to the bitline-sensing time.

11.2.7 Case Study: Itanium 2 Cache

Figure 7.92 showed a die photograph of the 1 GHz Itanium 2 processor [Weiss02] illustrating a large (24 Mbit) embedded SRAM used as the Level 3 cache in a 180 nm process. The cache is partitioned into 135 subarrays: 128 are used for data, while 5 are used for ECC and 2 are spares available to replace bad data. All of the subarrays are accessed in parallel through 8-bit paths, so the cache line size is 1024 bits.

Each subarray in turn is divided into eight 24-kbit banks organized as 96 words by 256 bits. The subarray decoder produces 96 global wordlines and 8 bank select lines. The local decoder activates the local wordline in the selected bank and the column multiplexer selects 8 bits from the 256-bit bank. One more level of 4:1 multiplexers returns the data in 2-bit chunks over four cycles to reduce the amount of global wiring. One load and one store can take place every four cycles at 1.2 GHz. Of this time, roughly half is used for the subarray access and half is used for distributing the address and data.

The individual 6T SRAM cells have an area of approximately 5.9 μm^2. The cache has a flexible floorplan to flow around the processor core and accommodate floorplan changes late in the design cycle. Overall, the cache has an 85% area efficiency, i.e., the 6T SRAM cells account for 85% of the area, while the decoders and other periphery circuitry account for only 15%.

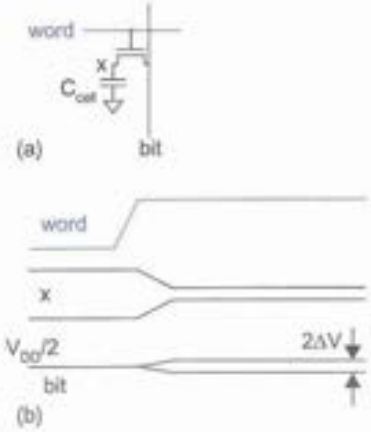
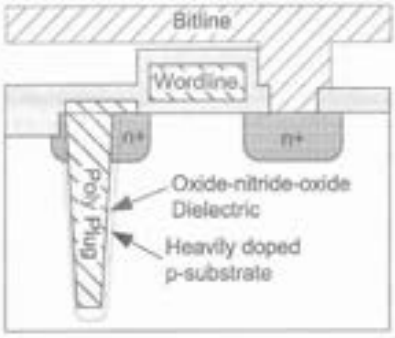


FIG 11.26 DRAM cell read operation



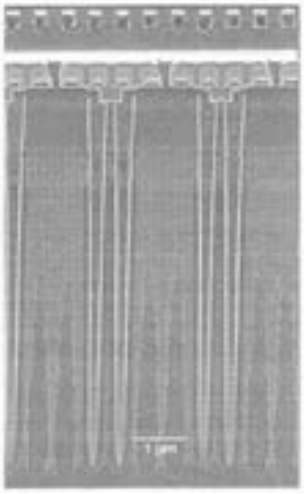


FIG 11.27 Trench capacitor

11.3 DRAM

Dynamic RAMs (DRAMs) store their contents as charge on a capacitor rather than in a feedback loop. Thus, the basic cell is substantially smaller than SRAM, but the cell must be periodically read and refreshed so that its contents do not leak away. Commercial DRAMs are built in specialized processes optimized for dense capacitor structures. They offer an order of magnitude greater density (bits/cm^2) than high-performance SRAM built in a standard logic process, but they also have much higher latency. DRAM circuit design is a very specialized art and is the topic of excellent books such as [Keeth01]. This section provides an overview of the general technique.

A 1-transistor (1T) dynamic RAM cell consists of a transistor and a capacitor, as shown in Figure 11.26(a). Like SRAM, the cell is accessed by asserting the wordline to connect the capacitor to the bitline. On a read, the bitline is first precharged to $V_{DD}/2$. When the wordline rises, the capacitor shares its charge with the bitline, causing a voltage change ΔV that can be sensed, as shown in Figure 11.26(b). The read disturbs the cell contents at x, so the cell must be rewritten after each read. On a write, the bitline is driven high or low and the voltage is forced onto the capacitor. Some DRAMs drive the wordline to $V_{DDP} = V_{DD} + V_t$ to avoid a degraded level when writing a '1.'

The DRAM capacitor C_{cell} must be as physically small as possible to achieve good density. However, the bitline is contacted to many DRAM cells and has a relatively large capacitance C_{bit}. Therefore, the cell capacitance is typically much smaller than the bitline capacitance. According to the charge-sharing equation, the voltage swing on the bitline during readout is

$$\Delta V = \frac{V_{DD}}{2} \frac{C_{cell}}{C_{cell} + C_{bit}} \tag{11.1}$$

We see that a large cell capacitance is important to provide a reasonable voltage swing. It also is necessary to retain the contents of the cell for an acceptably long time and to minimize soft errors, as was mentioned in Section 4.8.7. For example, 30 fF is a typical target. The most compact way to build such a high capacitance is to extend into the third dimension. For example, Figure 11.27 shows a cross-section and scanning electron microscope (SEM) image of *trench capacitors* etched under the source of the transistor. The walls of the trench are lined with an oxide-nitride-oxide dielectric. The trench is then filled with a polysilicon conductor that serves as one terminal of the capacitor attached to the transistor drain, while the heavily doped substrate serves as the other terminal. A variety of three-dimensional capacitor structures have been used in specialized DRAM processes that are not available in conventional CMOS processes.

11.3.1 Subarray Architectures

Like SRAMs described in Section 11.2.5, large DRAMs are divided into multiple subarrays. The subarray size represents a tradeoff between density and performance. Larger subarrays amortize the decoders and sense amplifiers across more cells and thus achieve better density. But they also are slow and have small bitline swings because of the high wordline and bitline capacitance. A typical subarray size is 256 words by 512 bits, as shown in Figure 11.28.

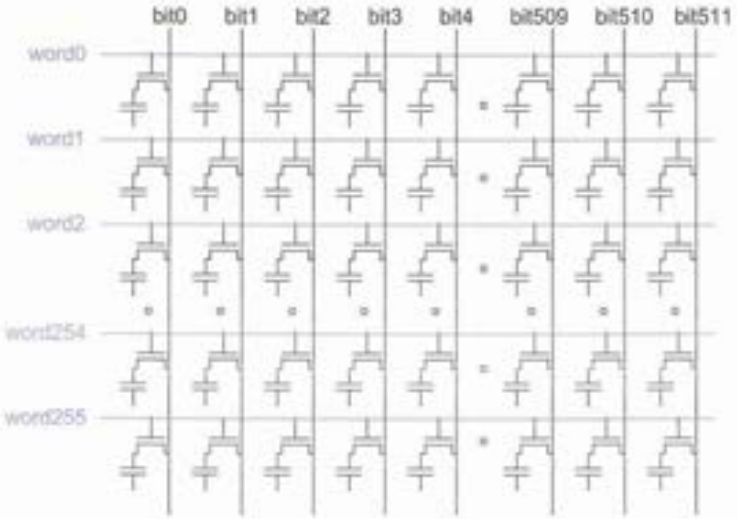


FIG 11.28 DRAM subarray

A subarray of this size has an order of magnitude higher capacitance on the bitline than in the cell, so the bitline voltage swing ΔV during a read is very small. The array uses a sense amplifier to compare the bitline voltage to that of an idle bitline (precharged to $V_{DD}/2$). The sense amplifier must also be very compact to fit the tight pitch of the array. The low-swing bitlines are very sensitive to noise. Three bitline architectures, *open*, *folded*, and *twisted*, offer different compromises between noise and area.

Until the 64 kbit generation, DRAMs used the open bitline architecture shown in Figure 11.29. In this architecture, the sense amplifier receives one bitline from each of two subarrays. The wordline is only asserted in one array, leaving the bitlines in the other array floating at the reference voltage. The arrays are very dense. However, any noise that affects one array more than the other will appear as differential noise at the sense amplifier. Thus, open bitlines have unacceptably low signal-to-noise ratios for high-density DRAM.

The folded bitline architecture is shown in Figure 11.30. In this architecture, each bitline connects to only half as many cells. Adjacent bitlines are organized in pairs as inputs to the sense amplifiers. When a wordline is asserted, one bitline will switch while

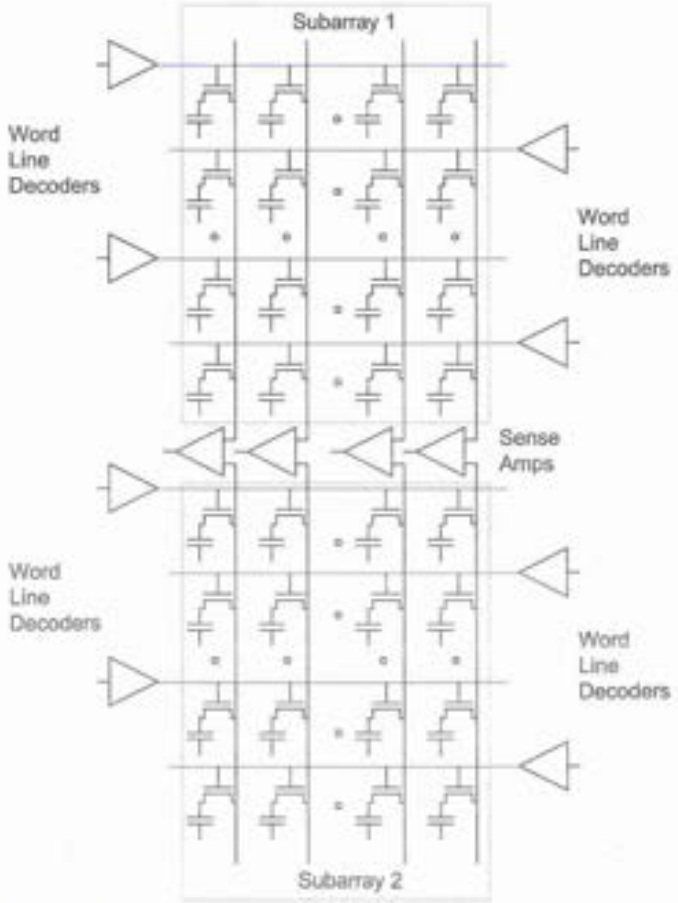


FIG 11.29 Open bitlines

its neighbor serves as the quiet reference. Many noise sources will couple equally onto the two adjacent bitlines so they tend to appear as common mode noise that is rejected by the sense amplifier. This noise advantage comes at the expense of greater layout area. Figure 11.31 shows a clever layout for a 6 × 8 folded bitline subarray that is only 33% larger than an open bitline layout. Observe how DRAM processes push the design rules and use diagonal polysilicon to reduce area. Notice how pairs of cells in the layout share a single bitline contact to minimize the bitline capacitance.

Unfortunately, the folded bitline architecture is still susceptible to noise from a neighboring switching bitline that capacitively couples more strongly onto one of the bitlines in the pair. Capacitive coupling is very significant in modern processes. The twisted bitline architecture [Hidaka89] solves this problem by swapping the positions of the folded bitlines part way along the array in much the same way as SRAM bitlines were twisted in Figure 11.18(b). The twists cost a small amount of extra area within the array.

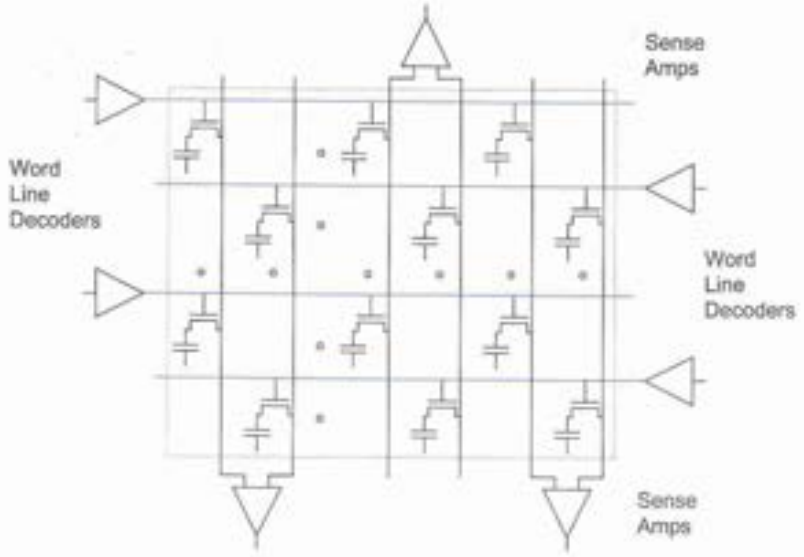


FIG 11.30 Folded bitlines

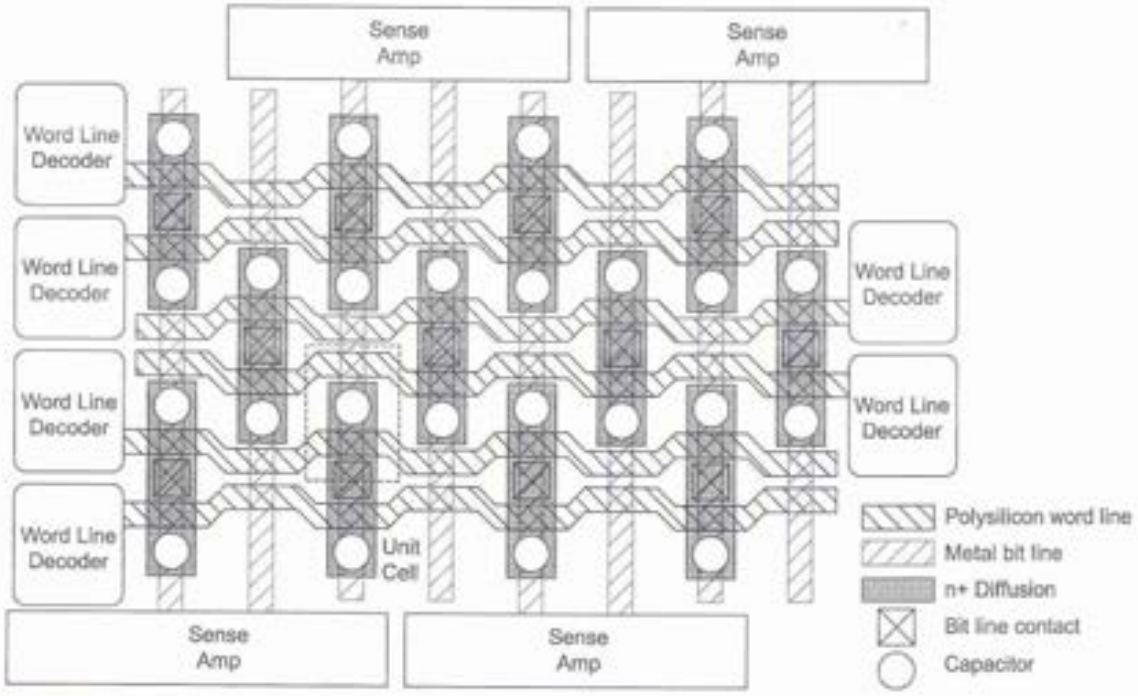


FIG 11.31 Layout of folded bitline subarray

11.3.2 Column Circuitry

The column circuitry in a DRAM includes the sense amplifiers, write drivers, column multiplexing, and bitline conditioning circuits. In a folded or twisted bitline architecture, the column circuitry is placed on both sides of the array so that it can be laid out on four times the pitch of a single column, as was shown in Figure 11.31. Part of the circuitry can be shared between two adjacent subarrays.

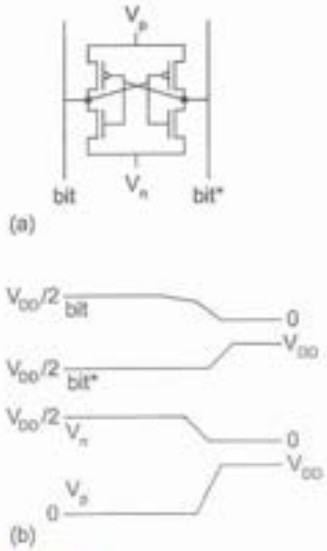


FIG 11.32 Sense amplifier

Figure 11.32(a) shows a basic sense amplifier built from cross-coupled inverters with supplies tied to control voltages. Initially, the two bitlines *bit* and *bit** are precharged to $V_{DD}/2$, the bottom voltage V_n is at $V_{DD}/2$, and the top voltage V_p is at 0 so all of the transistors in the amplifier are OFF. During a read, one of the bitlines will change by a small amount while the other floats at $V_{DD}/2$. V_n is then pulled low. As it falls to a threshold voltage below the higher of the two bitline voltages, the cross-coupled nMOS transistors will begin to pull the lower bitline voltage down to 0. After a small delay, V_p is pulled high. The cross-coupled pMOS transistors pull the higher bitline voltage up to V_{DD}. For example, Figure 11.32(b) shows the waveforms while reading a '0' on *bit* while using *bit** as a reference. Driving the active bitline to one of the rails has the side effect of rewriting the cell with the value that was just read.

Figure 11.33 shows a bitline conditioning circuit that precharges and equalizes a pair of bitlines to $V_{DD}/2$ when *EQ* is asserted. This consumes very little power because the voltage is reached by sharing charge between one bitline at V_{DD} and the other at GND.

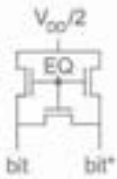


FIG 11.33 Bitline conditioning

Figure 11.34 puts together the complete column circuitry serving two folded subarrays. Each subarray column produces a pair of signals, *bit* and *bit**. The *CSEL* signal, produced by the column decoder, determines if this column will be connected to the I/O line for the array. Each subarray has its own equalization transistors and pMOS portion of the sense amplifier. However, the nMOS sense amplifier and I/O lines are shared between the subarrays. Either *ISO1* or *ISO2* is asserted to connect one subarray to the I/O lines while leaving the other isolated. During a read operation, the data is read onto the I/O lines. During a write, one I/O line is driven high and the other low to force a value onto the bitlines. The cross-coupled pMOS transistors pull the bitlines to a full logic level during a write to compensate for the threshold drop through the isolation transistor.

11.3.3 Applications to CMOS Systems-on-chip

The number of bits of RAM on a typical chip is increasing faster than the number of logic transistors. Recall that the Itanium II in Figure 7.92 devotes approximately 50% of the die area to SRAM cache. DRAM would be very attractive if it could more compactly substitute for the on-chip SRAM. Moreover, on-chip DRAM offers much higher bandwidth than external DRAM because the full bandwidth of the subarrays is directly available. Unfortunately, as of the time this book is written, DRAM is not widely used on standard CMOS processes. A good DRAM cell requires special capacitor structures and high-threshold, low-leakage transistors that are not available in a conventional CMOS process.

CMOS logic requires low-threshold transistors and many metal layers not available in a conventional DRAM process. Adding these process steps increases the manufacturing cost. If cost-effective solutions can be found, DRAM may become an important technology for the CMOS designer [Chandrakasan01].

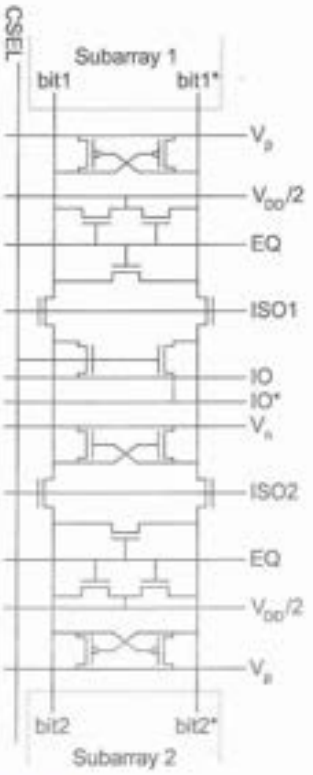


FIG 11.34 Column circuitry

11.4 Read-only Memory

Read-only Memory (ROM) cells can be built with only one transistor per bit of storage. A ROM is a nonvolatile memory structure in that the state is retained indefinitely — even without power. A ROM array is commonly implemented as a single-ended NOR array using any of the NOR gate structures studied so far, including the pseudo-nMOS and the footless dynamic NOR gate. As in SRAM cells and other footless dynamic gates, the wordline input must be low during precharge on dynamic NOR gates. In situations where DC power dissipation is acceptable and the speed is sufficient, the pseudo-nMOS ROM is the easiest to design, requiring no timing. The DC power dissipation can be significantly reduced in multiplexed ROMs by placing the pull-up transistors after the column multiplexer.

Figure 11.35 shows a 4-word by 6-bit ROM using pseudo-nMOS pull-ups with the following contents:

```
word0: 010101
word1: 011001
word2: 100101
word3: 101010
```

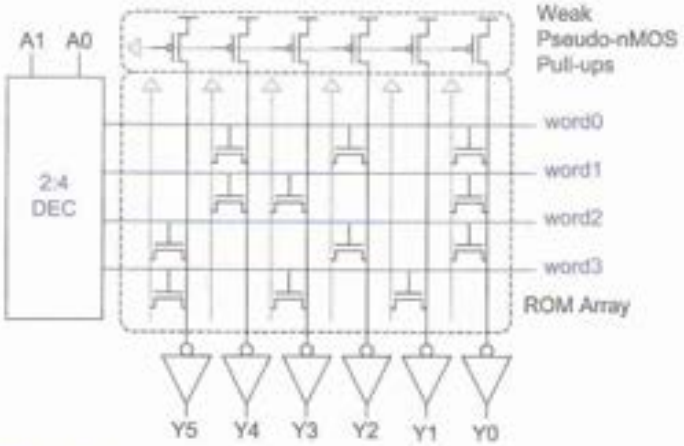


FIG 11.35 Pseudo-nMOS ROM

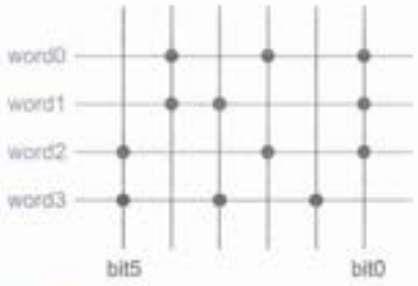


FIG 11.36 Dot diagram representation of ROM

The contents of the ROM can be symbolically represented with a dot diagram in which dots indicate the presence of 1's, as shown in Figure 11.36. The dots actually correspond to nMOS pull-down transistors connected to the bitlines, but the outputs are inverted.

Mask-programmed ROMs can be configured by the presence or absence of a transistor or contact, or by a threshold implant that turns a transistor permanently OFF where it is not needed. Omitting transistors has the advantage of reducing capacitance on the wordlines and power consumption. Programming with metal contacts was once popular because such ROMs could be completely manufactured except for the metal layer, and then programmed according to customer requirements through a metallization step. The advent of EEPROM and Flash memory chips has reduced demand for such mask-programmed ROMs. Figure 11.37 shows a layout for the 4-word by 6-bit ROM array. The wordlines run horizontally in polysilicon, while the bitlines and grounds run vertically in metal1. Notice how each ground is shared between a pair of cells. Each bit of the ROM occupies a $12 \times 8\ \lambda$ cell[1]. Polysilicon wordlines are only appropriate for small or slow ROMs. A larger ROM can run metal2 straps over the polysilicon and contact the two periodically (e.g., every eight columns). Occasional substrate contacts are also required.

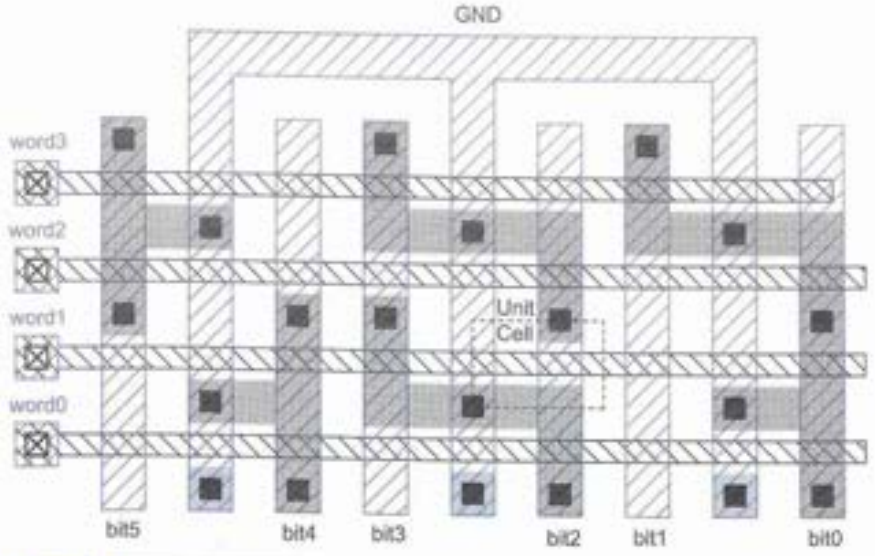


FIG 11.37 ROM array layout

[1]The cell can be reduced to $11 \times 7\ \lambda$ by running the ground line in diffusion and by reducing the width and spacing to $3\ \lambda$.

Row decoders for ROMS are similar to those for RAMs except that they are usually very constrained by the ROM wordline pitch. Figure 11.38 shows how each output of a 2:4 decoder can be shoehorned into a single horizontal track using vertical polysilicon true and complementary address lines and metal supply lines. Column decoders for ROMs are usually simpler than those for RAMs because single-ended sensing is commonly employed.

Figure 11.39 shows a complete pseudo-nMOS ROM including row decoder, cell array, pMOS pull-ups, and output inverters.

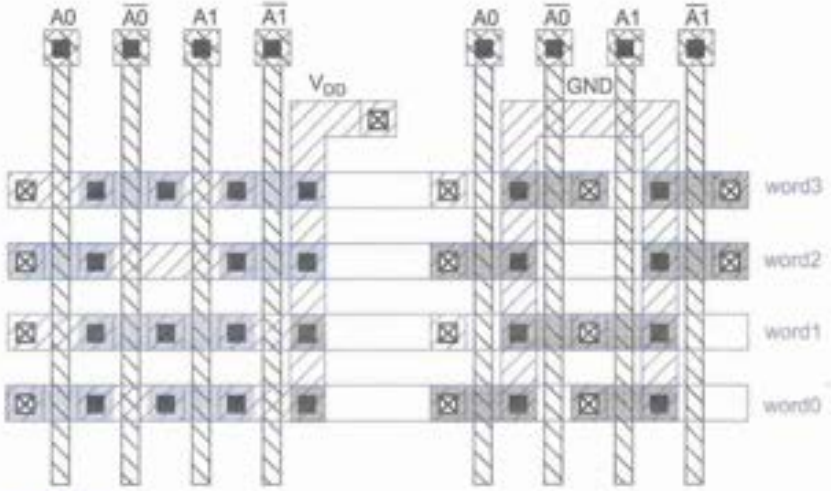


FIG 11.38 Row decoder layout on tight pitch

11.4.1 Programmable ROMs

It is often desirable for the user to be able to program or reprogram a ROM after it is manufactured. As discussed in Section 11.1, this contradicts the "Read Only" nature of the device; ROM has in practice become synonymous with *nonvolatile*, not read-only memory. Programming/writing speeds are generally slower than read speeds for ROMs. Four types of nonvolatile memories include *Programmable* ROMs (PROMs), *Erasable Programmable* ROMs (EPROMs), *Electrically Erasable Programmable* ROMs (EEPROMs), and *Flash* memories. All of these memories require some enhancements to a standard CMOS process: PROMs use fuses while EPROMs, EEPROMs, and Flash use charge stored on a floating gate.

Programmable ROMs can be fabricated as ordinary ROMs fully populated with pull-down transistors in every position. Each transistor is placed in series with a fuse made of polysilicon, nichrome, or some other conductor that can be burnt out by applying a high current. The user typically configures the ROM in a specialized PROM programmer before putting it in the system. As there is no way to repair a blown fuse, PROMs are also referred to as *one-time programmable* memories.

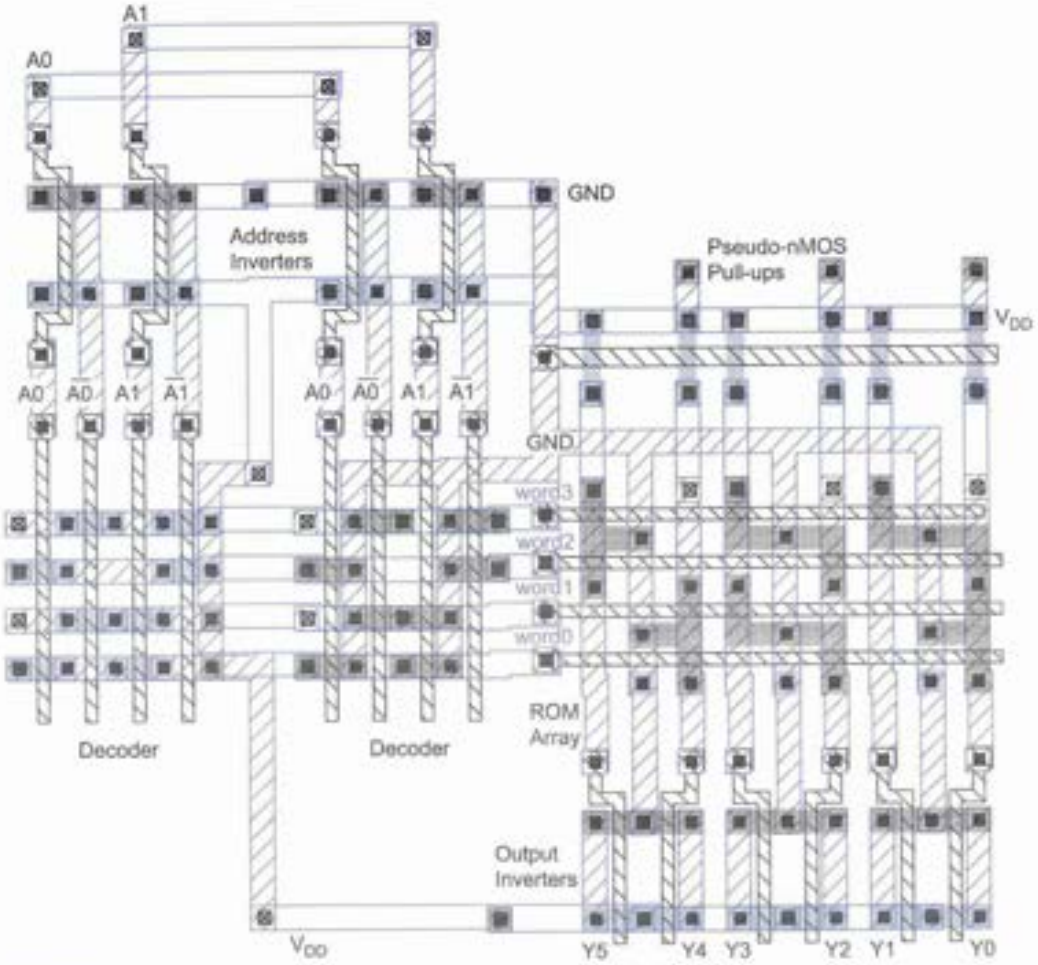


FIG 11.39 Complete ROM layout

As technology has improved, reprogrammable nonvolatile memory has largely displaced PROMs. These memories, including EPROM, EEPROM, and Flash, use a second layer of polysilicon to form a floating gate between the primary gate and the channel, as shown in Figure 11.40. The floating gate is a good conductor, but it is not attached to anything. Applying a high voltage to the upper gate causes electrons to jump through the thin oxide onto the floating gate through the processes called *avalanche injection* or *Fowler-Nordheim tunneling*. Injecting the electrons induces a negative voltage on the floating gate, effectively increasing the threshold voltage of the transistor to the point that it is always OFF.

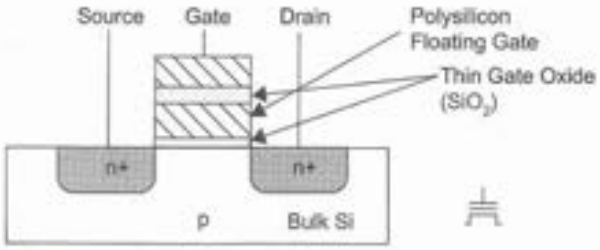


FIG 11.40 Cross-section of floating gate nMOS transistor

[Itoh01k] and [Rabaey03] offer a good overview of these memories. In brief, EPROM is programmed electrically, but it is erased through exposure to ultraviolet light that knocks the electrons off the floating gate. It offers a dense cell, but it is inconvenient to erase and reprogram. EEPROM and Flash can be erased electrically without being removed from the system. EEPROM offers fine-grained control over which bits are erased, while Flash is erased in bulk. EEPROM cells are larger to enable them to provide this versatility, so Flash has become the most economical form of convenient nonvolatile storage. For example, Flash memory cards are widely used in digital cameras to store pictures even after the camera is turned off. Flash is also useful for firmware or configuration data because it can be rewritten to upgrade a system in the field without opening the case or removing parts.

11.4.2 NAND ROMs

The ROM from Figure 11.35 is called a NOR ROM because each of the bitlines is just a pseudo-nMOS NOR gate. The bitline pulls down when a wordline attached to any of the transistors is asserted high. The size of the cell is limited by the ground line. Figure 11.41 shows a NAND ROM that uses active-low wordlines. Transistors are placed in series and the transistors on the nonselected rows are ON. If no transistor is associated with the selected word, the bitline will pull down. If a transistor is present, the bitline will remain high.

Figure 11.42(a) shows a layout of the NAND ROM. The cell size is only $7 \times 8\ \lambda$. The contents are specified by using either a transistor or a metal jumper in each bit position. The contacts limit the cell size. Figure 11.42(b) shows an even smaller layout in which transistors are located at every position. In this design, an extra implantation step can be used to create a negative threshold voltage, turning certain transistors permanently ON where they are not needed. In such a process, the cell size reduces to only $6 \times 5\ \lambda$, assuming that the decoder and bitline circuitry can be built on such a tight pitch.

A disadvantage of the NAND ROM is that the delay grows quadratically with the number of series transistors discharging the bitline. NAND structures with more than 8–16 series transistors become extremely slow, so NAND ROMs are often broken into multiple small banks with a limited number of series transistors. Nevertheless, these NAND

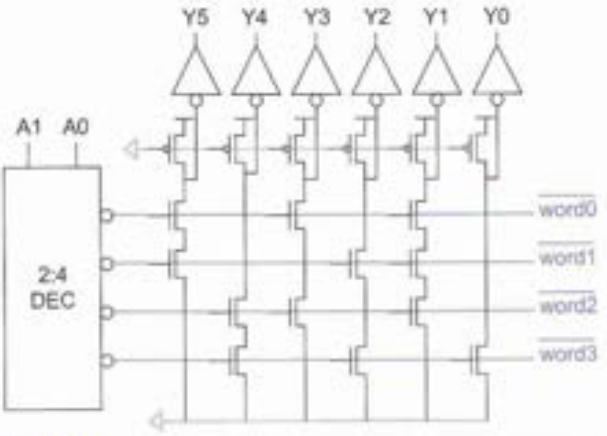


FIG 11.41 Pseudo-nMOS NAND ROM

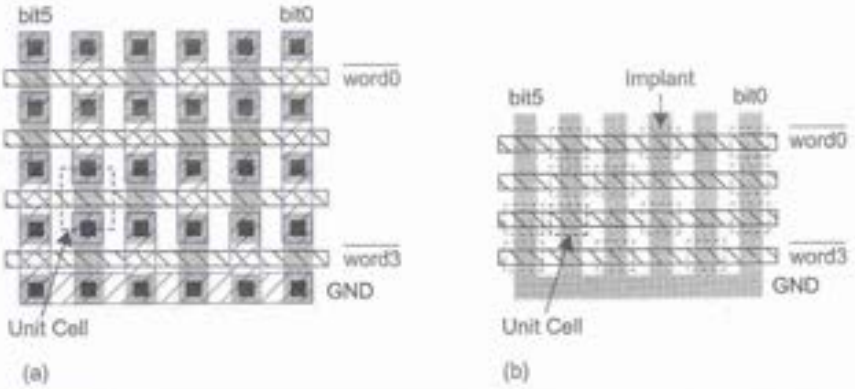


FIG 11.42 NAND ROM array layouts

structures are attractive for Flash memories in which density and cost are more important than access time.

11.5 Serial Access Memories

Using the basic SRAM cell and/or registers, we can construct a variety of serial access memories including shift registers and queues. These memories avoid the need for external logic to track addresses for reading or writing.

11.5.1 Shift Registers

A *shift register* is commonly used in signal-processing applications to store and delay data. Figure 11.43(a) shows a simple 4-stage 8-bit shift register constructed from 32 flip-flops. As there is no logic between the registers, particular care must be taken that hold times are satisfied. Flip-flops are rather big, so large, dense shift registers use dual-port RAMs instead. The RAM is configured as a circular buffer with a pair of counters specifying where the data is read and written. The read counter is initialized to the first entry and the write counter to the last entry on reset, as shown in Figure 11.43(b). Alternately, the counters in an N-stage shift register can use two 1-of-N hot registers to track which entries should be read and written. Again, one is initialized to point to the first entry and the other to the last entry. These registers can drive the wordlines directly without the need for a separate decoder, as shown in Figure 11.43(c).

One variant of a shift register is a *tapped delay line* that offers a variable number of stages of delay. Figure 11.44 shows a 64-stage tapped delay line that could be used in a video processing system. Delay blocks are built from 32-, 16-, 8-, 4-, 2-, and 1-stage shift registers. Multiplexers control pass-around of the delay blocks to provide the appropriate total delay.

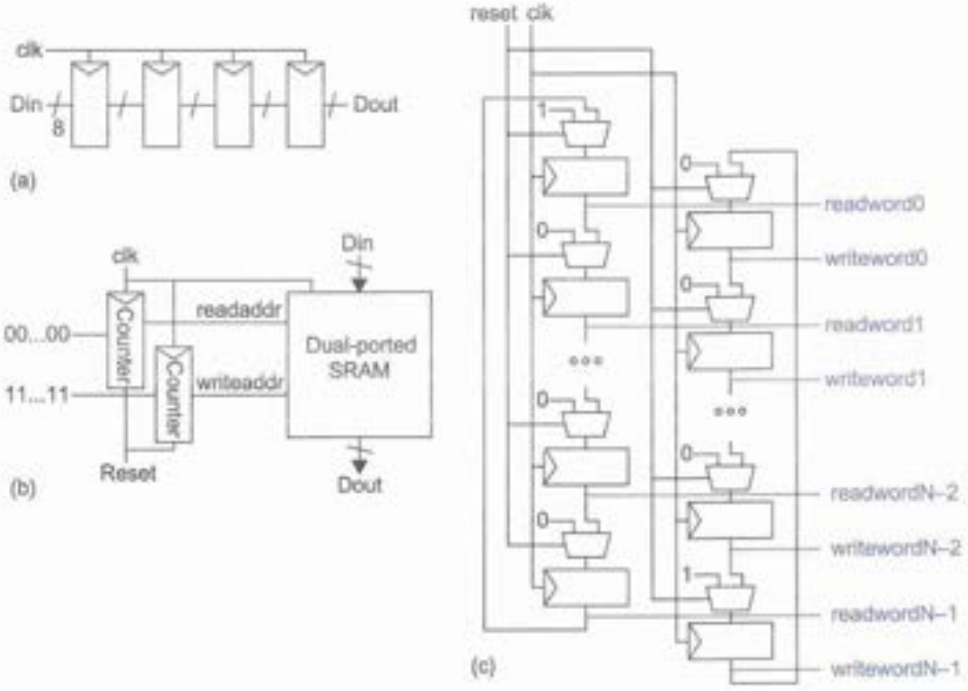


FIG 11.43 Shift registers

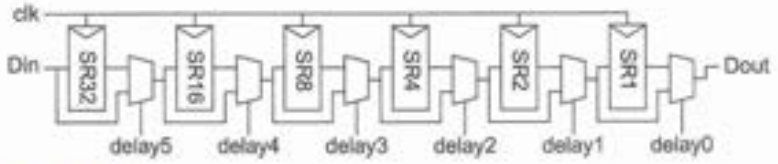


FIG 11.44 Tapped delay line

Another variant is a serial/parallel memory. Figure 11.45(a) shows a 4-stage Serial In Parallel Out (SIPO) memory and Figure 11.45(b) shows a 4-stage Parallel In Serial Out (PISO) memory. These are also often useful in signal processing and communications systems.

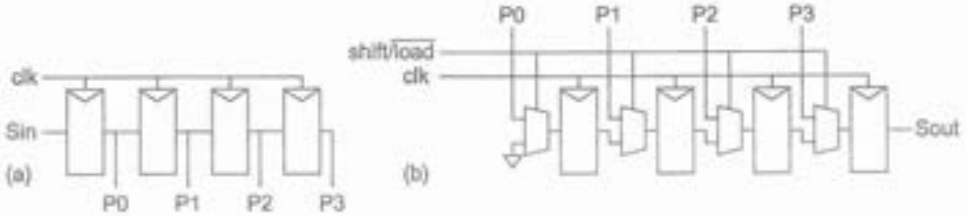


FIG 11.45 Serial/parallel memories

11.5.2 Queues (FIFO, LIFO)

Queues allow data to be read and written at different rates. Figure 11.46 shows an interface to a queue. The read and write operations each are controlled by their own clocks that may be asynchronous. The queue asserts the *FULL* flag when there is no room remaining to write data and the *EMPTY* flag when there is no data to read. Because of other system delays, some queues also provide ALMOST-FULL and ALMOST-EMPTY flags to communicate the impending state and halt write or read requests. The queue internally maintains read and write pointers indicating which data should be accessed next. As with a shift register, the pointers can be counters or 1-of-*N* hot registers.

FIG 11.46 Queue

First In First Out (*FIFO*) queues are commonly used to buffer data between two asynchronous streams. Like a shift register, the FIFO is organized as a circular buffer. On reset, the read and write pointers are both initialized to the first element and the FIFO is EMPTY. On a write, the write pointer advances to the next element. If it is about to catch the read pointer, the FIFO is FULL. On a read, the read pointer advances to the next element. If it catches the write pointer, the FIFO is EMPTY again.

Last In First Out (*LIFO*) queues, also known as *stacks*, are used in applications such as subroutine or interrupt stacks in microcontrollers. The LIFO uses a single pointer for both read and write. On reset, the pointer is initialized to the first element and the LIFO is EMPTY. On a write, the pointer is incremented. If it reaches the last element, the LIFO is FULL. On a read, the pointer is decremented. If it reaches the first element, the LIFO is EMPTY again.

11.6 Content-addressable Memory

Figure 11.47 shows the symbol for a content-addressable memory (CAM) [Grosspietsch92, Schultz95, Miyatake01]. The CAM acts as an ordinary SRAM that can be read or written given adr and data, but also performs *matching* operations. Matching asserts a *matchline* output for each word of the CAM that contains a specified *key*.

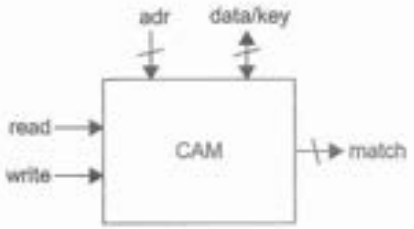


FIG 11.47 Content-addressable memory

A common application of CAMs is translation lookaside buffers (TLBs) in microprocessors supporting virtual memory. The virtual address is given as the key to the TLB CAM. If this address is in the CAM, the corresponding matchline is asserted. This matchline can serve as the wordline to access a RAM containing the associated physical address, as shown in Figure 11.48. A NOR gate processing all of the matchlines generates a *Miss* signal for the CAM. Note that the *read*, *write*, and *adr* lines for updating the TLB entries are not drawn.

10T and 9T implementations of the CAM cell are shown in Figure 11.49. The cells consist of a normal SRAM cell with additional transistors to perform the match. Multiple CAM cells in the same word are tied to the same matchline. The matchline is either precharged or pulled high as a distributed pseudo-nMOS gate. The key is placed on the bitlines. In Figure 11.49(a), if the key and the value stored in the cell differ, the matchline will be pulled down. Only if all of the key bits match all of the bits stored in the word of memory will the matchline for that word remain high. The key can contain a "don't care"

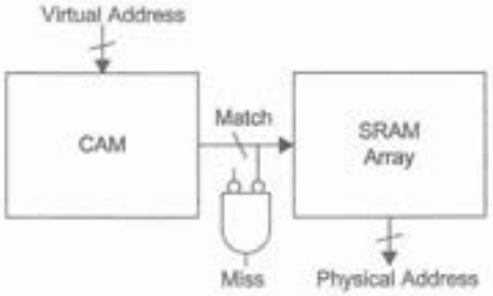


FIG 11.48 Translation Lookaside Buffer (TLB) using CAM

by setting both *bit* and *bit_b* low. Figure 11.50 shows a layout of this cell in a 56 × 43 λ area; CAMs generally have about twice the area of SRAM cells. Figure 11.49(b) shows another CAM cell design with one fewer transistor. *N1* and *N2* perform an XOR of the key and cell data. If the values disagree, *N3* is turned on to pull down the wordline. However, the gate of *N3* sees a degraded high logic level.

Figure 11.51 shows a complete 4 × 4 CAM array. Like an SRAM, it consists of an array of cells, a decoder, and column circuitry. However, each row also produces a dynamic matchline. The matchlines are precharged with the clocked pMOS transistors. The *miss* signal is produced with a distributed pseudo-nMOS NOR.

When the matchlines are used to access a RAM, the monotonicity problem must be considered. Initially, all the matchlines are high. During CAM operation, the lines pull down, leaving at most one line asserted to indicate which row contains the key. However, the RAM requires a monotonically rising wordline. Figure 11.52 refines Figure 11.48 with strobed AND gates driving the wordlines as early as possible after the matchlines have settled. The strobe can be timed with an inverter chain or replica delay line in much the same way that the sense amplifier clock for a SRAM was generated in Section 11.2.3. As usual, self-timing margin must be provided so the circuit operates correctly across all design corners.

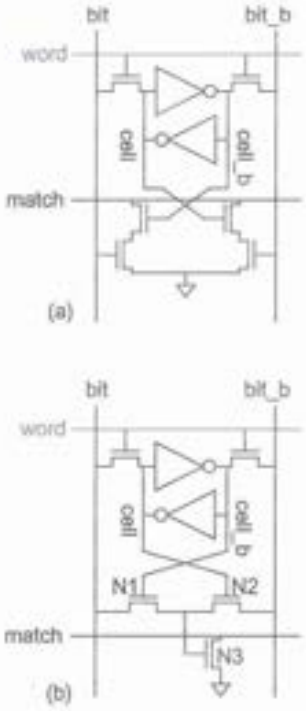


FIG 11.49 CAM cell implementations

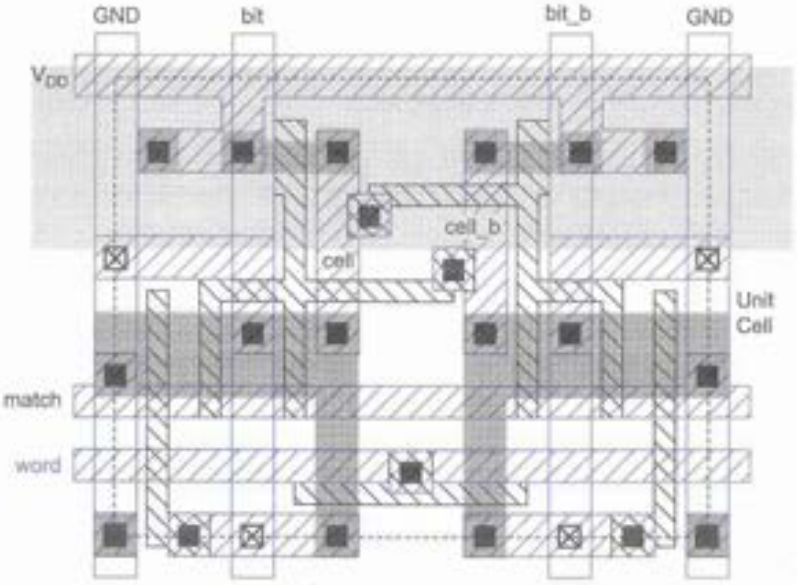


FIG 11.50 CAM cell layout. Color version on inside front cover.

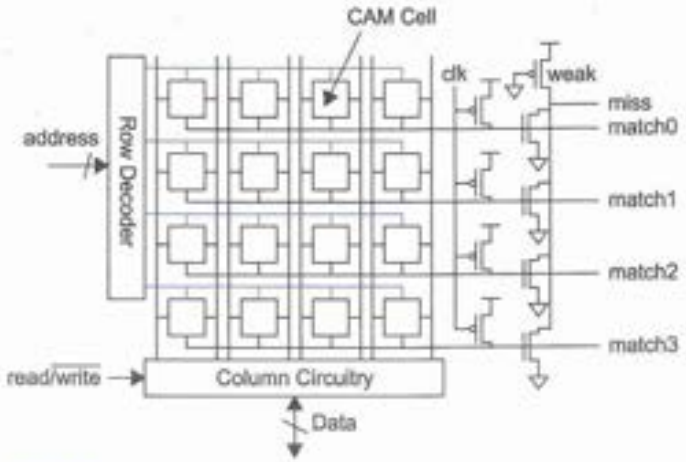


FIG 11.51 4 × 4 CAM array

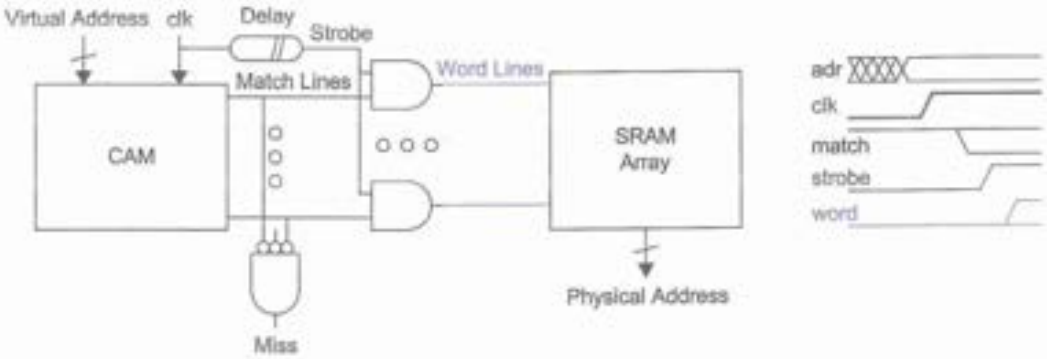


FIG 11.52 Refined TLB path with monotonic wordlines

Large CAMs can use many of the same techniques as large RAMs, including sense amplifiers and multiple subarrays. They tend to consume relatively large amounts of power because the bitlines and matchlines may all transition during a parallel search. [Miyatake01] describes a design of a mid-sized CAM using a pMOS match-line driver to reduce the swing of the matchlines and save power.

11.7 Programmable Logic Arrays

A *programmable logic array* (PLA) provides a regular structure for implementing combinational logic specified in *sum-of-products canonical form*. If outputs are fed back to inputs through registers, PLAs also can form finite state machines. PLAs were most popular in the early days of VLSI when two-level logic minimization was well understood, but multilevel logic optimizers were still immature. They are dense and fast ways to implement simple functions, and with suitable CAD support, are very easy to change when logic bugs are discovered. For more complex functions, modern logic synthesis often produces faster and more compact circuits, so PLAs are now used less frequently. Moreover, pseudo-nMOS PLAs dissipate static power and may be particularly slow pulling up, while dynamic PLAs require careful design of timing chains.

Any logic function can be expressed in sum-of-products form, i.e., where each output is the OR (sum) of the ANDs (products) of true and complementary inputs. The inputs and their complements are called *literals*. The AND of a set of literals is called a *product* or *minterm*. The outputs are ORs of minterms. The PLA consists of an *AND plane* to compute the minterms and an *OR plane* to compute the outputs.

NOR gates are particularly efficient in pseudo-nMOS and dynamic logic because they use only parallel, never series, transistors. Hence, we use DeMorgan's law to replace the AND and OR gates with NORs after inverting inputs and outputs, as shown in

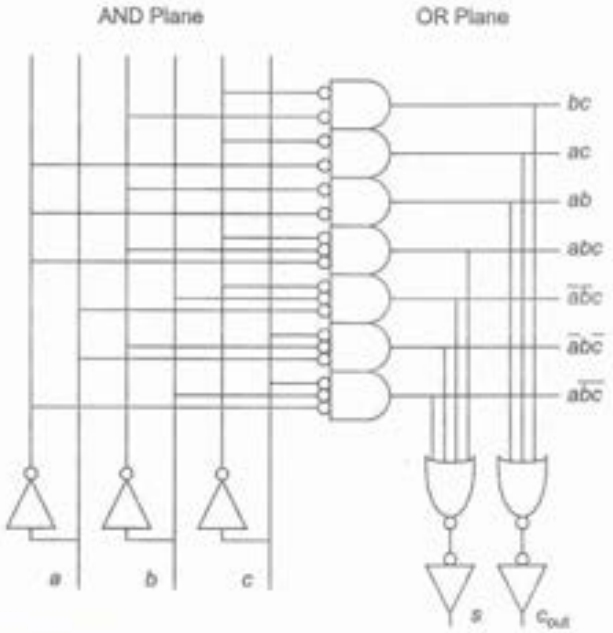


FIG 11.53 NOR/NOR representation of PLA

Example

Write the equations for a full adder in sum-of-products form. Sketch a 3-input, 2-output PLA implementing this logic.

Solution: Figure 11.54 shows the PLA. The logic equations are

$$s = ab\bar{c} + \bar{a}\bar{b}\bar{c} + \bar{a}\bar{b}c + abc$$
$$c_{out} = ab + bc + ac \tag{11.2}$$

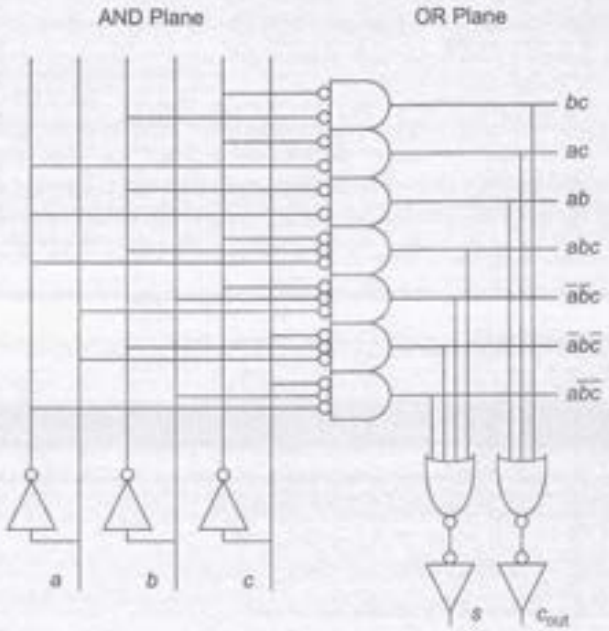


FIG 11.54 AND/OR representation of PLA

Figure 11.53. For brevity, we often represent the PLA with a dot diagram, shown in Figure 11.55. Observe that a ROM and a PLA are very similar in form. The ROM decoder is equivalent to an AND plane generating all 2^n minterms. The ROM array corresponds to an OR plane producing the outputs.

A generic floorplan for a simple PLA is shown in Figure 11.56. This has been designed as a set of tiles, designated by letters given in Table 11.1. Experienced designers often add a few unused rows and columns to their PLAs to accommodate last-minute design changes without changing the overall footprint of the PLA.

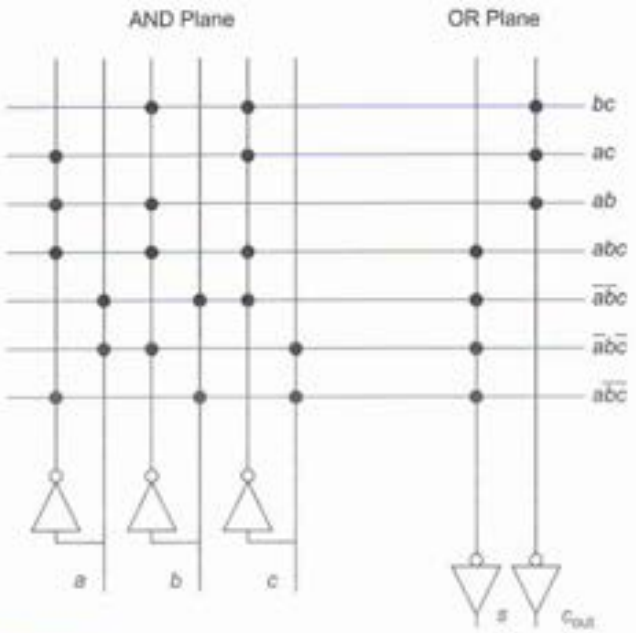


FIG 11.55 Dot diagram representation of PLA

TL	TA	TA	TM	TO	TO	TR
LA	AN	AN	AO	OR	OR	RO
LA	AN	AN	AO	OR	OR	RO
BL	BA	BA	BM	BO	BO	BR

FIG 11.56 Generic PLA floorplan

Table 11.1 PLA cell tiles

Tile	Function
AN	AND plane programming cell
OR	OR plane programming cell
AO	AND-OR communication cell
TA	Top AND plane input cell
BA	Bottom AND plane input cell
TO	Top OR plane output cell
BO	Bottom OR plane output cell
LA	Left AND plane cell
RO	Right OR plane cell
BL	Bottom left cell
BM	Bottom middle cell
BR	Bottom right cell
TL	Top left cell
TM	Top middle cell
TR	Top right cell

The most straightforward PLA design uses a pseudo-nMOS NOR gate. Figure 11.57 shows the circuit diagram for the full adder PLA. Design of the pseudo-nMOS NOR gates should follow the guidelines given in Section 6.2.2. Advantages of this PLA include simplicity and small size. Disadvantages include the static power dissipation of the NOR gates and slow pull-up response. A clocked register can be added if desired. Figure 11.58 shows a layout for the pseudo-nMOS PLA. The transistor gates are run in polysilicon and could be strapped with metal2. Observe how ground lines can be shared between pairs of minterms and outputs so that each minterm and output can be placed on a 1.5 track pitch. The inverters require careful layout to fit the tight pitch. You can see how the layout could be assembled by tiling the various cells listed in Table 11.1.

Dynamic PLAs eliminate the contention current and are slightly faster than their pseudo-nMOS counterparts. Figure 11.59(a) shows a PLA using footed dynamic NORs for both the AND and OR planes. Unfortunately, the AND plane must drive the OR plane directly, violating monotonicity. The OR plane must take a clock phase that is delayed until the minterms adequately discharge (to below V_t). This clock is often generated with a replica delay line that is guaranteed to be no faster than the slowest minterm in the AND plane. Moreover, the OR plane outputs must be captured before the AND plane precharges so that the results are not corrupted. To accomplish this, the PLA may be supplied by clocks similar to those shown in Figure 11.59(b).

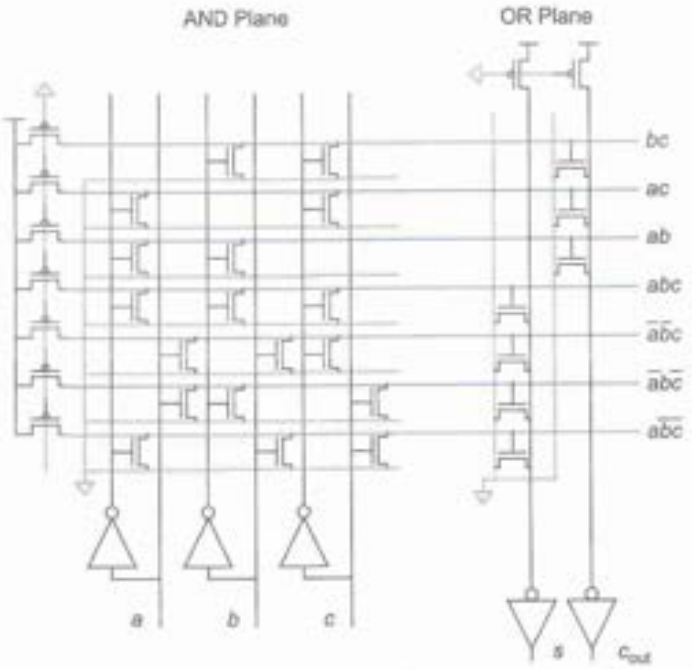


FIG 11.57 Pseudo-nMOS PLA schematic

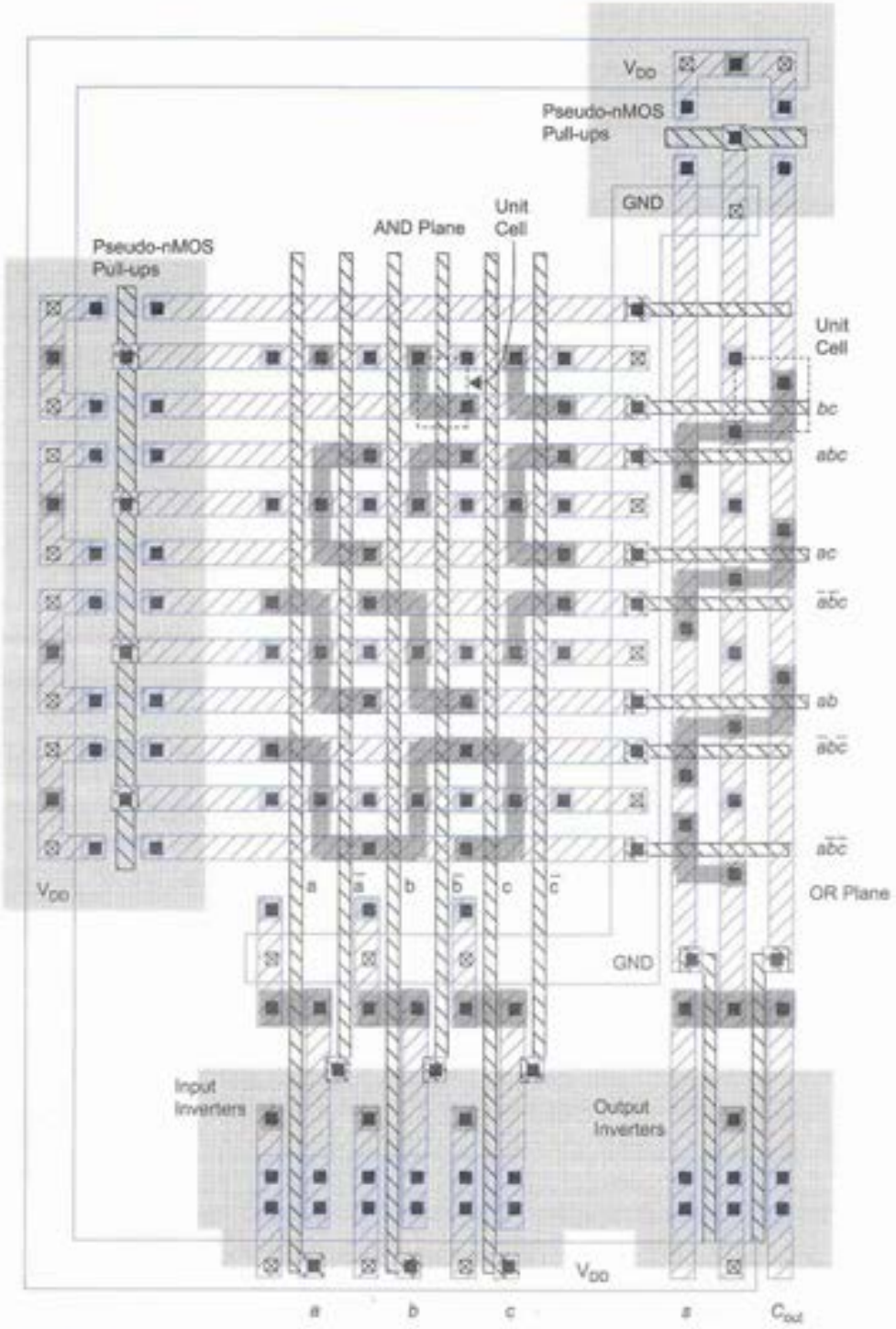


FIG 11.58 Pseudo-nMOS PLA layout

Figure 11.59(c) shows a self-timed dynamic PLA using two dummy rows as replica delay lines. Assume that the inputs arrive from flip-flops and settle shortly after the rising edge of the clock. The clocked circuitry acts as a pulse generator, producing a low-going precharge pulse on ϕ_{AND} shortly after the clock edge. The width of the pulse is equal to the delay of dummy AND row 1 plus two inverters and should be great enough to fully precharge all of the real AND rows. Thus, the loading on the dummy AND row is chosen to equal or exceed the worst loading of any real row. This worst loading consists of one nMOS drain for each input and one gate for each output. In this figure, the size of the inverter loading the AND line can be selected to contribute the desired gate load. Once the AND plane enters evaluation, the second dummy AND row starts to discharge through a single transistor. Again, this row is loaded to equal or exceed the delay of the worst real AND row. The three inverters provide some self-timing margin to ensure that ϕ_{OR} will not rise until the AND plane has fully evaluated. The output of the OR plane can be sampled into flip-flops on the next rising edge of the clock. [Wang01] surveys a variety of other PLA designs.

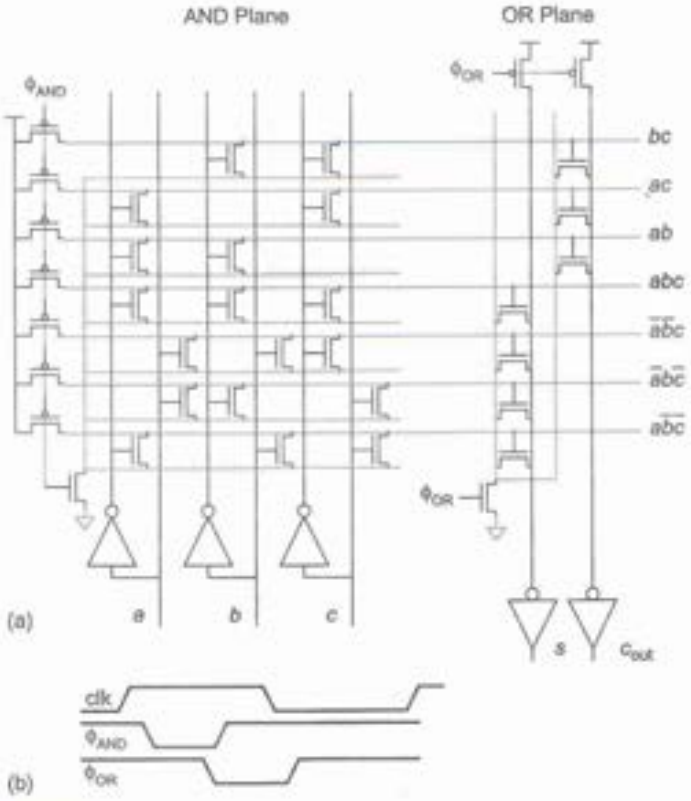


FIG 11.59 Dynamic PLA schematic (continues)

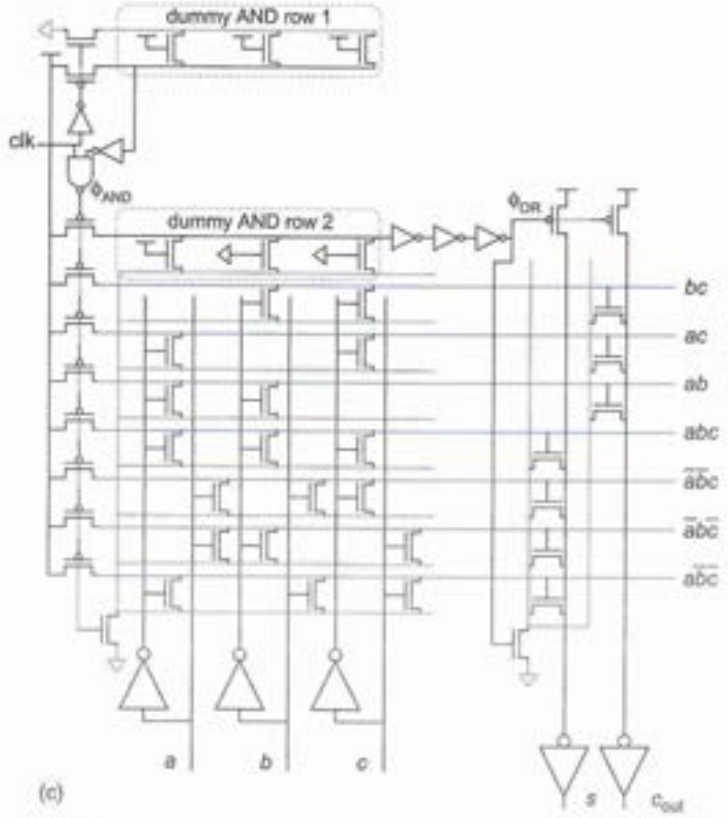


FIG 11.59 Dynamic PLA schematic (continued)

11.8 Array Yield, Reliability, and Self-test

Because arrays occupy a large fraction of the die area of many system-on-chip and microprocessor designs, they strongly influence the overall chip yield and reliability. Fortunately, their regular structure makes it easy to enhance the design for better yield and reliability.

A single defect in logic circuits will usually render the entire chip useless. The yield of memories can be improved by adding redundant and dummy wordlines and bitlines [Keeth01]. The dummy lines are placed at the edge because photolithography and etch problems occur most often near the edge of large repetitive structures. The dummy lines are sacrificed to provide better yield on the good lines. Redundant wordlines and bitlines serve as spares to replace defective lines and further improve yield. The row and column decoders can be reprogrammed during manufacturing test with laser-programmable fuses

to eliminate the bad lines and substitute the redundant lines, or can be configured at start-up using built-in self-test. Large memories add entire redundant subarrays to replace subarrays with defective decoders or excessive numbers of defects. For example, some microprocessors have many banks in a set-associative cache. A processor can be manufactured with 8 banks and sold with the guarantee that at least 6 are operational. The bad ones are located during manufacturing test and disabled with fuses.

Memories are also prone to soft errors that spontaneously flip a bit stored in one of the cells, as discussed in Section 4.8.7. Error-detecting and correcting codes (ECC) are commonly used to recover from such errors, as discussed in Section 10.7.2. For example, adding 8 check bits to a 64-bit word in a memory is sufficient to correct any error in the word and detect any pair of errors. ECC supplements redundant bitlines to dramatically improve yield as well.

Memories are generally tested by writing known patterns to each of the bits, reading the patterns, and then rewriting the opposite pattern and reading it back to ensure each bit can be set to 0 and to 1. Performing these tests on an external tester is time-consuming and expensive, particularly for embedded memories that are not directly accessible from the chip I/Os. Many memories now contain built-in self-test (BIST) that places multiplexers in the address and data paths to take over the memory during test mode, as discussed in Section 9.6.6.7. A small BIST controller containing an address counter and data pattern generator drives the memory at full speed during test. Self-test can be performed every time the chip is reset.

11.9 Historical Perspective

MOS memory made a splash in 1970 when Intel announced sales of the first 1103 1-kb DRAM chip and IBM replaced magnetic core memories with semiconductor memories in its 370-series mainframe computers. Since then, DRAM has become a commodity business characterized by ferocious price competition among a rather small number of manufacturers. Indeed, in 1986, Intel left what was then its core business when the market was flooded by cheap chips from Japan. DRAM capacity per chip has increased by 60% per year and cost per bit has decreased by 27% per year. Feature size improvement accounts for part but not all of the capacity gains. The area per bit has shrunk faster than feature size because of clever cell designs such as the 1T DRAM cell, innovative layout, and three-dimensional capacitor structures. Larger dice have become economical because of manufacturing yield improvements. Growing DRAM capacity has benefited system designers as much as the advances in processor performance.

DRAM density has quadrupled approximately every three years. Table 11.2 lists some of the innovations at each DRAM generation [Itoh01k]. The first generations of DRAMs used 3T or 4T memory cells shown in Figure 11.60. IBM patented the 1T cell in 1968 [Dennard68] and it soon became ubiquitous because the small cell size justified the complexity of the capacitor. Early DRAMs were built in nMOS processes requiring high supply voltages. V_{DD} standardized at 5 V through the 1980s and 1990s and CMOS peripheral

circuitry was eventually adopted to save power. Other improvements addressed the signal-to-noise ratio, bandwidth and latency, power consumption, and test time.

Table 11.2 DRAM generations

Capacity	Years of Volume Shipment	Power Supply (V)	Memory Cell	Circuit Innovations
1 kb	1970s	> 12	3T or 4T	MOS technology Differential sensing
4 kb	1970s	> 12	3T or 4T	Multiplexed addresses
16 kb	–1984	12	1T	Dynamic amplifier Dynamic driver
64 kb	1981–1987	5	1T	Folded bitline Word bootstrapping Substrate bias generator
256 kb	1984–1992	5	1T	Shared amplifier Metal-strapped wordline Redundancy
1 Mb	1987–1997	5	1T	CMOS peripheral circuits Half-V_{DD} precharge Multidivided data line BIST
4 Mb	1991–2000	5	1T	3-D capacitor structure
16 Mb	1994–2003	5	1T	On-chip voltage converter Twisted bitlines
64 Mb	1997–	3.3	1T	Synchronous small-signal I/O Multidivided wordlines
256 Mb	2001–	1.8–3.3	1T	Double data rate interface

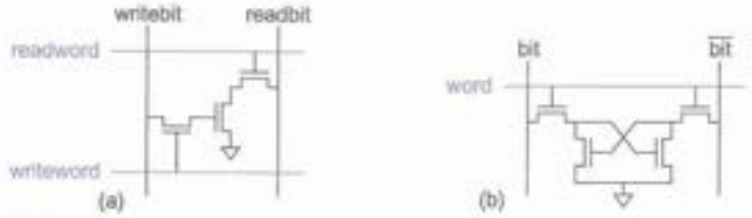


FIG 11.60 3T and 4T DRAM cells

Summary

Arrays repeat a basic cell in two dimensions. The cell is carefully optimized to provide very high density. For performance or density reasons, the nodes within the array do not always swing from rail to rail. Periphery circuitry restores the output swings to full digital logic levels.

The static RAM is very widely used in CMOS systems. The ubiquitous 6T cell consists of a cross-coupled inverter pair to hold the state and two access transistors for differential reads and writes. The bitlines are first preconditioned to a known value. A decoder asserts one of the wordlines. That word is read onto the bitlines and sensed. A column multiplexer may select only a subset of the bits as outputs. SRAMs are used in caches and other embedded memories. Multi-ported SRAMs are used in register files.

Content-addressable memories are similar to SRAMs. However, they also provide a lookup mode in which a key is placed on the bitlines and each word that contains that key asserts its matchline. CAMs are important for looking up addresses in translation look-aside buffers and network routers.

Dynamic RAMs store information on a capacitor using a single access transistor. With specialized process steps to build compact capacitors, they offer an order of magnitude higher density of data storage than SRAM. However, the data gradually leaks off the capacitors, so DRAMs must be periodically refreshed to maintain their state. DRAMs are usually built in specialized processes on dedicated chips, but potentially may be useful for high-capacity embedded memories on digital CMOS processes.

Read-only memories also use a single access transistor, but their contents are wired to a constant value. They are commonly used to store code and are convenient because they can be easily changed late in the design process to correct bugs or add features. EEPROM and Flash memories are even more convenient because they can be changed after fabrication or even in the field. Flash is also widely used for nonvolatile data storage.

A ROM can also be viewed as a lookup table. In general, a ROM of 2^x words by y bits can serve as a lookup table to perform any function of x inputs and y outputs. If a function is written in sum-of-products form, the ROM decoder performs the AND operation while the ROM array performs the OR. Many functions are relatively sparse. A programmable logic array optimizes out the unnecessary entries by replacing the decoder with an AND plane. In some cases, PLAs are smaller than ROMs, yet provide the same flexibility of easy changes late in the design cycle. PLAs were commonly used for microcoded finite state machines in the 1980s. They are still occasionally used, but good logic synthesis tools now deliver the same ease of change for random logic while avoiding the complicated circuit design needed for an efficient PLA.

A good design flow should provide automatic generators for simple SRAMs and ROMs. The designer should be comfortable with using these arrays where they are appropriate. High-performance designs need more elaborate multi-ported SRAM, large memory arrays, and CAMs. Most of these arrays demand skilled circuit design and careful simulation.

For more advanced reading on memory design, see [Keeth01, Prince99, or Itoh01k].

Exercises

11.1 An embedded SRAM contains 2048 8-bit words. If it is physically arranged in a square fashion, how many inputs does each column multiplexer require?

11.2 Estimate the dimensions of the SRAM array in Exercise 11.1 using the SRAM cell from Figure 11.6b, assuming periphery circuitry adds 10% to each dimension of the core.

11.3 Sketch designs for a 6:64 decoder with and without predecoding. Comment on the pros and cons of predecoding.

11.4 Sketch a 4:16 Lyon-Schediwy Decoder. Label transistor sizes so each input sees the same capacitance. Compute the logical effort and branching effort seen by the decoder.

11.5 Estimate the minimum delay of a 10:1024 decoder driving an electrical effort of $H = 20$ using

(a) static CMOS gates

(b) footless domino gates

11.6 Design the footless domino decoder from Exercise 11.5(b) using self-resetting domino gates. Assume the inputs are available in true and complementary form as pulses with a duration of 3 FO4 inverters and can each drive 48 λ of gate width. Indicate transistor sizes and estimate the delay of the decoder.

11.7 Develop a model of wordline decoder delay for a RAM with 2^n rows and 2^m columns. Assume true and complementary inputs are available and that the input capacitance equals the capacitance of one of the columns so $H = 2^m$. Use static CMOS gates and express your result in terms of n and m.

11.8 Explain the tradeoffs between open, closed, and twisted bitlines in a dynamic RAM array.

11.9 Sketch a dot diagram for a 2-input XOR using a ROM.

11.10 Sketch a dot diagram for a 2-input XOR using a PLA.

11.11 Sketch a schematic for an 8-word × 2-bit NAND ROM that serves as a lookup table to implement a full adder.

11.12 Explain the advantages and disadvantages of NAND ROMs as compared to NOR ROMs.

11.13 Develop a model for the read time of a ROM with 2^n rows and 2^m columns analogous to that of the SRAM from Section 11.2.6. Assume the wire capacitance in the ROM array is negligible compared to the gate and diffusion capacitance. Assume the ROM cells are laid out such that two cells share a single diffusion contact and hence each contributes only C/2 of diffusion capacitance.

Special-purpose Subsystems 12

12.1 Introduction

This chapter describes a variety of special-purpose subsystems that a digital designer may encounter. These subsystems are usually designed by a specialist or obtained from a third-party vendor, and each is the subject of entire books. However, the skilled digital designer should be conversant in each area in order to understand the impact of the other subsystems on a core digital design.

The chapter begins with packaging. The package strongly impacts the power distribution and I/O subsystems, which are discussed next. Then we look at clock generation and distribution, which are vital to achieve low clock skew and communicate synchronously with the external world. Finally, the chapter addresses rudimentary analog circuit design.

12.2 Packaging

The chip package provides a mechanical and electrical connection between the chip and a circuit board. It is no longer possible to separate the design of a high-performance integrated circuit from the design of its package. An ideal package has the following properties:

- Connects signals and power between the chip and board with little delay or distortion
- Removes heat produced by the chip
- Protects the chip from mechanical damage and thermal expansion stress
- Is inexpensive to manufacture and test

To provide good signal and power connections, the package must offer short wires with low resistance and inductance. The impacts of the package on the power supply and I/O are discussed further in Sections 12.3 and 12.4, respectively. The remainder of this

section describes some of the types of packages commonly available and how they remove heat from the chip.

12.2.1 Package Options

Table 12.1 lists a variety of common integrated circuit packages. Figure 12.1 shows photographs of these packages. The I/O count includes connections for both signals and power. I/O spacing is typically specified in the archaic unit of mils (1 mil = 0.001 inch = 25.4 μm). Packages come in both ceramic and plastic varieties; plastic is cheaper, but cannot remove as much heat. Older packages tended to use *through-hole* pins, which pass through holes in a printed circuit board and are soldered from below. The pins contribute inductance, and the size of the holes limits the density of the pins. *Surface mount* (SMT) packages are soldered to the surface of a printed circuit board to alleviate these problems. Dual Inline Packages (DIP), Pin Grid Array (PGA), and Plastic Leadless Chip Carrier (PLCC) packages are easy to insert into low-cost sockets, so they are convenient for components that might be removed for reprogramming or replacement. Ball Grid Array (BGA) packages have recently become the preferred approach for parts that require a large number of high-bandwidth signals in a compact form factor. Package design is a rapidly advancing field and new packages are being adopted each year.

Table 12.1 Package options

Package	# I/Os	Description
Dual Inline Package (DIP)	8, 14, 16, 20, 28, 40, 64	Two rows of through-hole pins on 100 mil centers. Low cost. Long wires between chip and corner pins.
Pin Grid Array (PGA)	65–391+	Array of through-hole pins on 100 mil centers. Low thermal resistance and high pin counts.
Small Outline IC (SOIC)	8,10, 14, 16, 20, 24, 28	Two rows of SMT pins on 50 mil centers. Low cost, good for low-power parts with small pin counts.
Thin Small Outline Package (TSOP)	28–86+	Two rows of SMT pins on 0.5 or 0.8 mm centers in a thin package. Commonly used for DRAMs.
Plastic Leadless Chip Carrier (PLCC)	20, 28, 44, 68, 84	J-shaped SMT pins on all four sides on 50 mil centers. Sturdy leads are convenient for socketing.
Quad Flat Pack (QFP)	44–240	SMT pins on all four sides on 15.7–50 mil centers. High density of I/Os. Available in thin (TQFP) and very thin (VQFP) forms as thin as 1.6 mm.
Ball Grid Array (BGA)	49–2000+	Array of SMT solder balls on underside of package on 15.7–50 mil centers. Extremely high density of I/Os with low parasitics. Requires specialized assembly and inspection equipment to blindly attach to array of pads on printed circuit board.
Flip-Chip	Many	Direct connection of chip to printed circuit board through solder balls on top metal layer of chip. Even higher I/O density and lower parasitics than BGA.

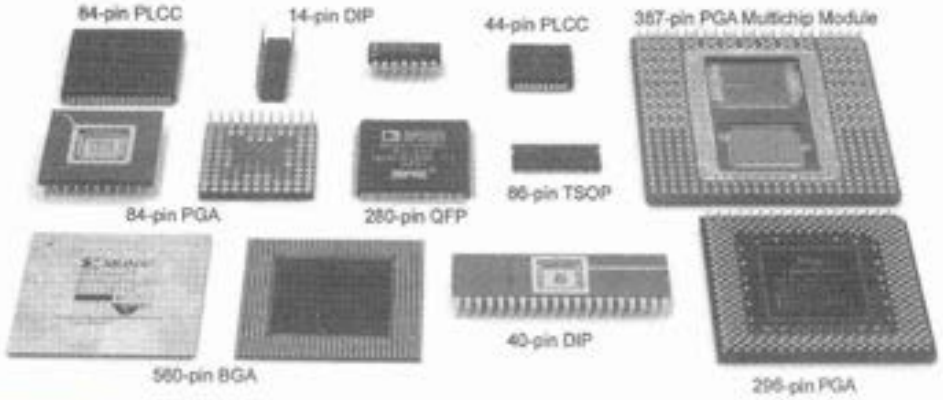


FIG 12.1 Integrated circuit packages. © 2003 Harvey Mudd College. Reprinted with permission.

Some high-performance systems are packaged in multi-chip modules (MCMs) that contain more than one die within the package. The MCM substrate connects the dice directly without the need for a printed circuit board, offering potentially higher speed interconnections. For example, the Pentium Pro shown in Figure 12.2 used an MCM to connect the microprocessor and one or two external cache dice. In another extreme example, the IBM z900 mainframe uses a multichip module containing 20 CPUs, 8 cache chips, and a kilometer of interconnect [Harrer02]. The module measures 127 mm on a side and dissipates 1.3 kW. MCMs are expensive to build and depend on the availability of tested but unpackaged dice, so they have not achieved the popularity anticipated in the mid-1990s.

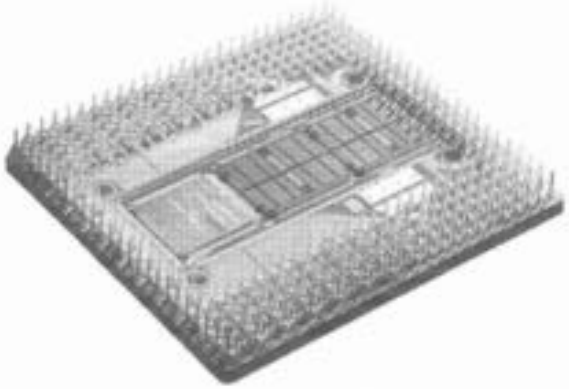


FIG 12.2 Pentium Pro MCM. Reprinted with permission of Intel Corporation.

12.2.2 Chip-to-package Connections

Conventionally, chips have been connected to their packages through thin (25 μm) gold wires bonded to metal pads. The pads are organized into a ring around the periphery of the chip called a *pad frame*. The minimum pitch of the pads is limited by the bonding machine to approximately 100–200 μm. Thus, a 1-cm² chip is limited to several hundred I/Os. Chips with large numbers of I/Os sometimes are *pad-limited*, meaning that the chip size is determined by the pad frame rather than by the logic within the chip. Figure 1.61 showed an example of a pad-limited chip in a 40-pin pad frame. Some chips have used a second ring of pads, but this approach results in longer bond wires and greater risk that the wires will accidentally touch.

The bond wires connect to a metal lead frame in the package. This lead frame distributes the I/Os to the periphery of the package and is bent to form the pins of the package. Many packages also include a heat spreader to help distribute the heat from the die across the package and ultimately out to the heat sink. Figure 12.3 shows a cutaway of a dual-in-line package showing a corner of the chip with bond wires connecting to the lead frame [Mahalingam85]. The metal leads contribute parasitic inductance and coupling capacitance to their neighbors. More advanced packages internally resemble printed circuit boards, using multiple layers of signals and power/ground planes to distribute the I/Os on controlled-impedance transmission lines.

Since the late 1990s, many manufacturers have adopted *flip-chip* connections. This technology, also called *Controlled Collapse Chip Connection* (C4), was developed by IBM in the 1960s and has been used on their mainframes for decades. In a flip-chip design, the surface of the chip is covered with an array of pads on the top level of metal. Lead solder balls are bonded to these pads in a final process step called *wafer bumping*. The chip is flipped upside down and connected to the package by heating the balls until they melt. The bonding requires careful alignment, but surface tension from the solder helps pull the

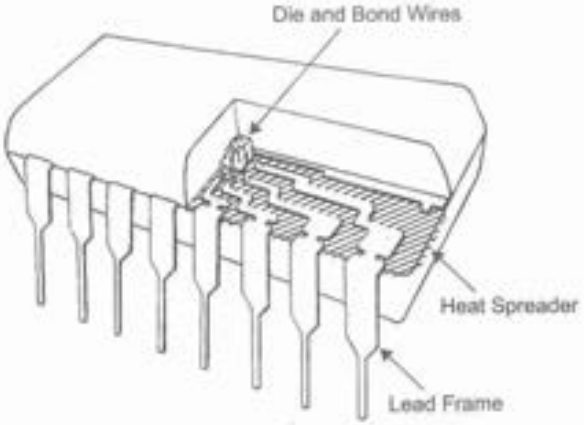


FIG 12.3 Cutaway view of dual-in-line package. © IEEE 1985. [Mahalingam85]

chip into place. The chip is in nearly direct contact with the package, eliminating the inductance associated with the bond wires. The bumps can be placed on a pitch of 200 μm or less, offering thousands of connections between the die and package. Flip-chip technology introduces new testing problems because the top-level metal wires are no longer accessible for probing during debug.

12.2.3 Package Parasitics

Figure 12.4 shows a model of an integrated circuit package. The bond wires and lead frame contribute parasitic inductance to the signal traces. They also have some mutual inductive and capacitive coupling to nearby signal traces, potentially causing crosstalk when multiple signals switch. The V_{DD} and GND wires also have inductance from both bond wires and the lead frame. Moreover, they have nonzero resistance, which becomes important for chips drawing large supply current. High-performance packages often include bypass capacitors between V_{DD} and GND. As we will see in Section 12.3.5, the bypass capacitors have their own parasitic resistance and inductance that limit their effectiveness at high frequencies.

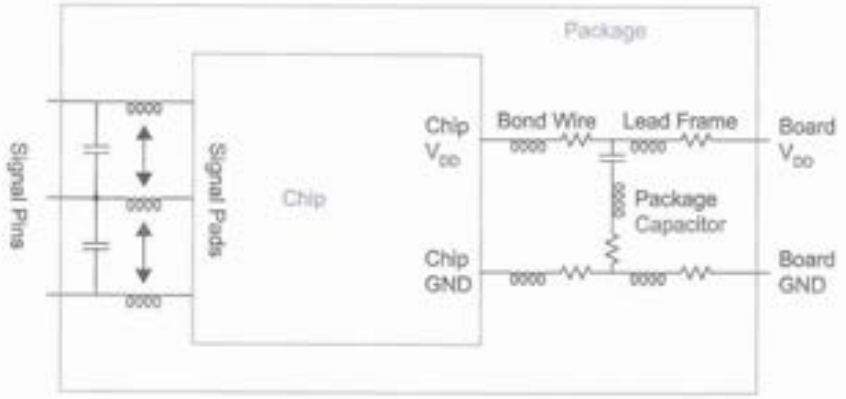


FIG 12.4 Package parasitics

12.2.4 Heat Dissipation

A 60-watt light bulb has a surface area of about 120 cm² and is too hot to touch. In comparison, the Itanium 2 dissipates 130 W on a 4 cm² die, resulting in a power density 60 times as great! Clearly, removing heat from chips is a major challenge for the package.

The heat generated by a chip flows from the transistor junctions where it is generated through the substrate and package. It can be spread across a heat sink, and then carried away through the air by means of convection. Just as current flow is determined by voltage difference and electrical resistance, the heat flow is determined by temperature difference

and thermal resistance. Thus, the temperature difference ΔT between the transistor junctions and the ambient air is:

$$\Delta T = \Theta_{ja} P \tag{12.1}$$

where Θ_{ja} is the thermal resistance (in °C/W) between the junction and ambient and P is the power consumption of the chip. The thermal resistance in turn can be modeled as the series resistance from the die to the package Θ_{jp} and from the package to the air Θ_{pa}.

$$\Theta_{ja} = \Theta_{jp} + \Theta_{pa} \tag{12.2}$$

For most low-cost packages, Θ_{pa} dominates the resistance. Still air can transfer about 0.001 W/(cm^2 °C) [Glasser85]. Thus, a package with a surface area of 10 cm^2 has a thermal resistance of about Θ_{pa} = 100° C/W. Such a package cannot handle chips dissipating more than about 1 watt. Forced air transfers 0.01–0.03 W/(cm^2 ° C). High-power chips add a large heat sink and a fan to the package to reduce the thermal resistance. For example, a 72-pin ceramic PGA package has a thermal resistance Θ_{pa} of 34° C/W in still air, 18° C/W in 400 ft/minute airflow, and 10° C/W in 400 ft/minute airflow with a good heat sink.

Example

You are planning to package an ASIC in a ball grid array package with a passive heat sink. The system box contains a large fan providing 250 linear feet/minute (LFM) of airflow. The package vendor specs the thermal resistance from the junction to package at 0.9° C/W. The heat sink vendor specs the thermal resistance from the package to ambient for this airflow at 4.0° C/W for the heat sink plus 0.1° C/W for the heat sink adhesive between the package and heat sink. The system box ambient temperature may reach 55° C. What is the maximum power dissipation of your ASIC if its junction temperature is not to exceed 100° C?

Solution: The thermal resistance is Θ_{ja} = 0.9 + 0.1 + 4.0 = 5° C/W. The temperature difference between the junction and ambient must not exceed ΔT = 100 − 55 = 45° C. Therefore, the maximum power dissipation is $P = \Delta T / \Theta_{ja}$ = 9 W.

Advances in heat sinks, fans, and packages have raised the practical limit for heat removal from about 8 W in 1985 to nearly 100 W today for affordable packaging. Forced-air cooling appears to be reaching its limits, setting a cap on the power consumption of chips. Liquid cooling is used in some mainframes and supercomputers to remove even greater amounts of heat. Perhaps mass production of liquid cooling systems will eventually drive the cost down to acceptable levels for mainstream systems.

12.3 Power Distribution

The power distribution subsystem of a chip consists of metal wires or planes on the chip, in the package, and on the printed circuit board. It also includes bypass capacitors to supply the instantaneous current requirements of the system. An ideal power distribution network has the following properties:

- Maintains a stable voltage with little noise
- Provides average and peak power demands
- Provides current return paths for signals
- Avoids wearout from electromigration and self-heating
- Consumes little chip area and wiring
- Is easy to lay out

Real networks must balance these competing demands, meeting targets of noise and reliability as inexpensively as possible. The noise goal is typically ±5% or ±10%; for example, a system with nominal V_{DD} = 1.8 V may guarantee the actual supply remains within 1.62–1.98 V. Reliability goals demand enough vias and metal cross-sectional area to carry the supply current, as was discussed in Section 4.8. The two fundamental sources of power supply noise are IR drops and L di/dt noise.

Figure 12.5 plots the power consumption versus time for a microprocessor [Gauthier02]. The power varies on a number of time scales. While the processor is active, the power depends on the operations and data. It also spikes near the clock edges when

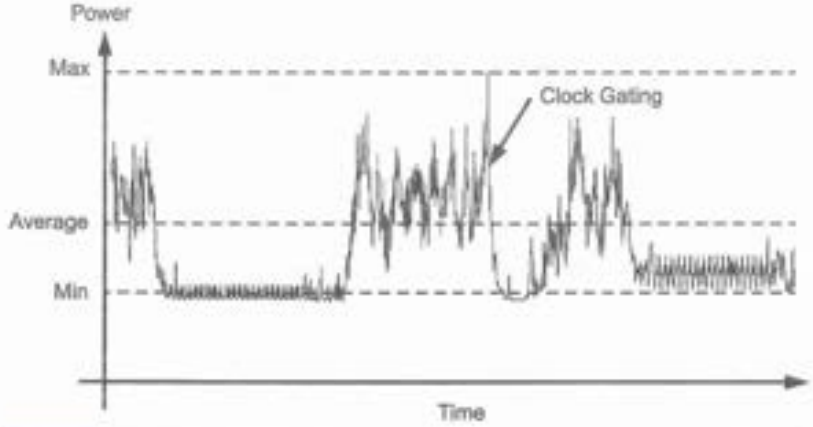


FIG 12.5 Time-dependent power consumption of microprocessor. Reprinted with permission of Sun Microsystems.

the large clock loads switch. When the processor becomes idle, clock gating turns off the clock to unused units, driving the power down significantly. As the supply voltage is nearly constant, the supply current I (also called I_{DD}) is proportional to the instantaneous power demand. As this current flows through the resistance R of the power distribution network, it causes a voltage droop proportional to IR. Moreover, as the changing current flows through the inductance of the printed circuit board and package, it also causes a voltage drop proportional to the rate of change: L di/dt.

This section begins by examining the physical design of a power distribution network. It then discusses IR drops and L di/dt noise. The key to controlling noise from current spikes is to provide adequate bypass capacitance on and off the chip to provide low supply impedance at all frequencies. The power network is complicated enough that manual analysis is inadequate; instead, it typically must be modeled in a finite element simulation. The power network also provides return paths for current flowing in signal wires. The geometry of the network affects the inductance of on-chip signals. Some critical circuits such as phase-locked loops and analog blocks require a quiet supply for good performance. RC filters can reduce much of the supply noise. In sensitive circuits, noise carried through the substrate is also important. For other introductions to power distribution subsystems, see [Bakoglu90, Dally98, Chandrakasan01, Mezhiba03].

12.3.1 On-chip Power Distribution Network

The on-chip power distribution network consists of power and ground wires within the cells and more wires connecting the cells together. Most cells contain internal power and ground busses routed on metal1 or metal2. These wires are typically wider than minimum to provide lower resistance and better electromigration immunity. For example, Figure 1.62 showed a cell library with 8 λ metal1 power/ground busses. These wires are normally connected between adjacent cells by abutment. Standard cell designs and datapaths both can use rows of cells sharing common power and ground lines.

In a small, low-power design, these rows can be strapped together with even wider vertical metal wires. Figure 12.6(a) shows an abstract diagram of this strapping. Figure 1.63 showed a standard cell design strapped with power on the left and ground on the right. In this example, the nMOS and pMOS transistors in adjacent rows are separated by a routing channel, so spacing between the wells is not a problem. In modern processes, the routing is typically done over the cell in upper-level metal. Therefore, the rows of cells can be packed more closely together and well spacing limits the packing density. Alternatively, every other row can be *mirrored* (flipped upside down) so that the wells of adjacent rows abut, as shown in Figure 12.6(b).

In a larger or high-power design, the resistance of the horizontal power and ground busses routed on thin lower-level metal will cause too much IR drop. Instead, the bulk of the current should be carried on a grid of wide and thick upper-level metal. Additional grids on the middle and lower layers carry the current down to the cells. Where layers connect, plenty of vias should be used to carry the high currents. Figure 12.7 shows such a power distribution grid in a six-level metal process with mirrored cells [Ayers03] and metal2 power busses within the cell. Notice that the metal3 lines share vertical tracks to

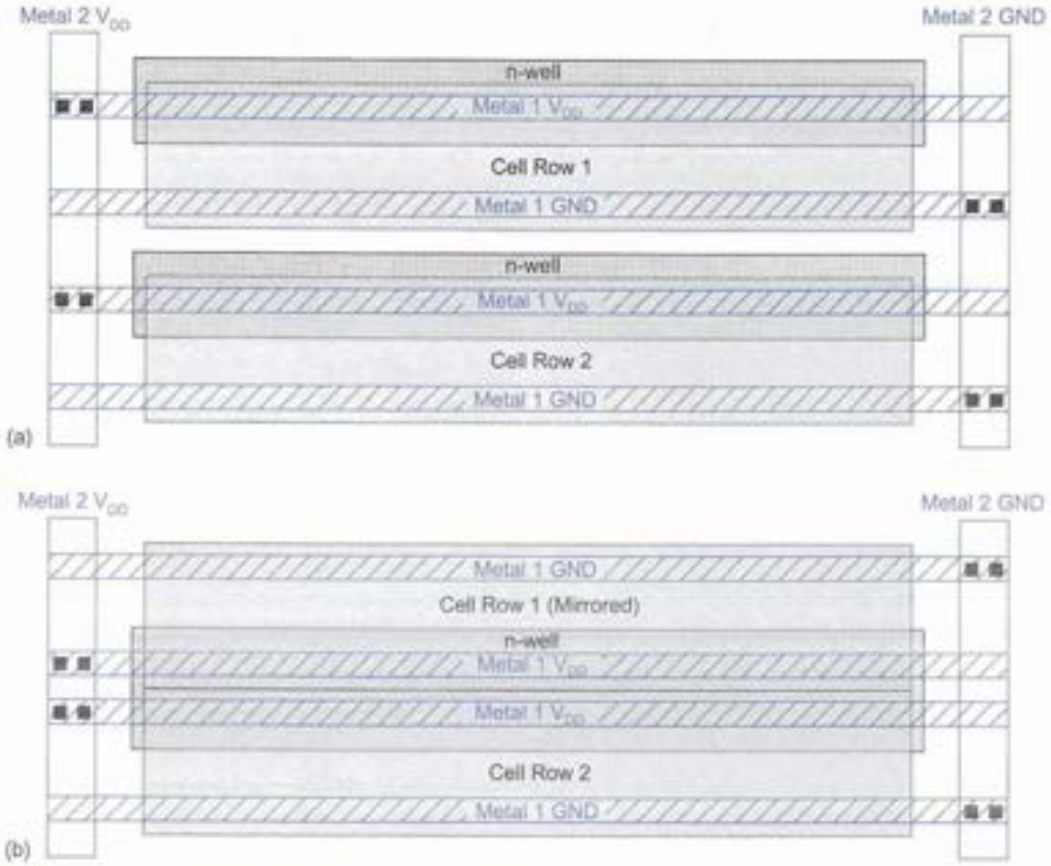


FIG 12.6 Power distribution for standard cell layout

conserve metal resources. Metal1 does not form a significant part of the power network and is left out of the figure.

The power grid extends across the entire chip. Ultimately, it must connect to the package through the I/O pads. When a pad ring is used, the connections are all near the periphery of the chip. Thus, the biggest IR drops occur near the center of the chip where the current flows through the longest wires and greatest resistance. Indeed, the Alpha 21264 devoted solid planes of metal to power and ground distribution to achieve acceptable IR drops at the center. C4 solder bumps distributed across the die are much better for power distribution because they can deliver the current from the low-resistance power plane in the package directly to the area of the chip where the current is needed. Thus, less on-chip metal resources are needed for power distribution.

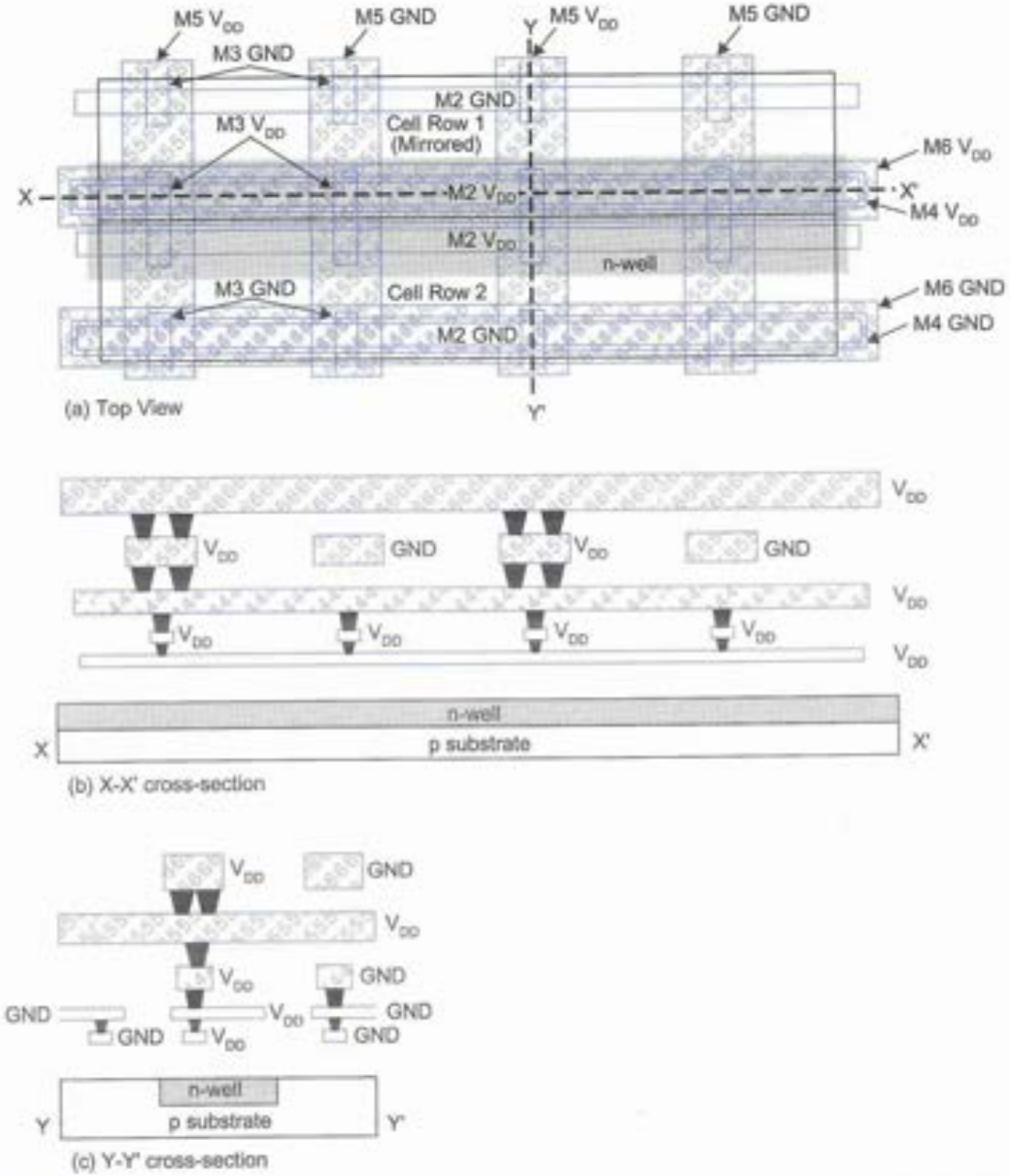


FIG 12.7 Power grid layout

12.3.2 IR Drops

The resistance of the power supply network includes the resistance of the on-chip wires and vias, the resistance of the bond wires or solder bumps to the package, the resistance of the package planes or traces, and the resistance of the printed circuit board places. Because the package and printed circuit board typically use copper that is much thicker and wider than on-chip wires, the on-chip network dominates the resistive drop.

IR drops arise from both average and instantaneous current requirements. The instantaneous current may be much larger than the average drop because current draw tends to locally spike near the clock edge when many registers and gates switch simultaneously. Bypass capacitance near the switching gates can supply much of this instantaneous current, so a well-bypassed power supply network only needs low enough resistance to deliver the average current demand, not necessarily the peak.

Example

Suppose a row of 64 repeaters share a common metal2 power bus like that shown in Figure 12.6(a). The bus is 320 μm long and 1 μm wide. The metal2 has a sheet resistance of 0.05 Ω/□. If the repeaters drive 0.4 pF wire loads with 200 ps transition times, estimate the power supply droop seen by the repeater for a 1.8 V nominal supply.

Solution: Each repeater draws a current of approximately

$$I = C\frac{\Delta V}{\Delta t} = (0.4\text{ pF})\frac{1.8\text{ V}}{200\text{ ps}} = 3.6\text{ mA}$$

The power and ground busses each have a length of 320 squares and thus a resistance of $R = 16\ \Omega$. The supply droop at the end of the wire caused by the 64 repeaters is 64 IR/2 = 1.85 V, or more than V_{DD}, which is obviously impossible. Instead, as the power supply begins to droop, the repeaters deliver less current, reducing the droop, but increasing the transition time and delay. One way to alleviate this problem is to use a power grid so that each repeater obtains its current from its own vertical wire rather than sharing the single horizontal wire with all of the simultaneously switching neighbors. Figure 12.8 shows a simulation of one of the repeaters. It compares the two power bus layouts. When all the repeaters share a single power wire, the power supply droops by nearly 30% and the propagation delay is more than doubled. When each repeater has its own power wire so the supply noise is negligible, the output is much crisper.

(continues)

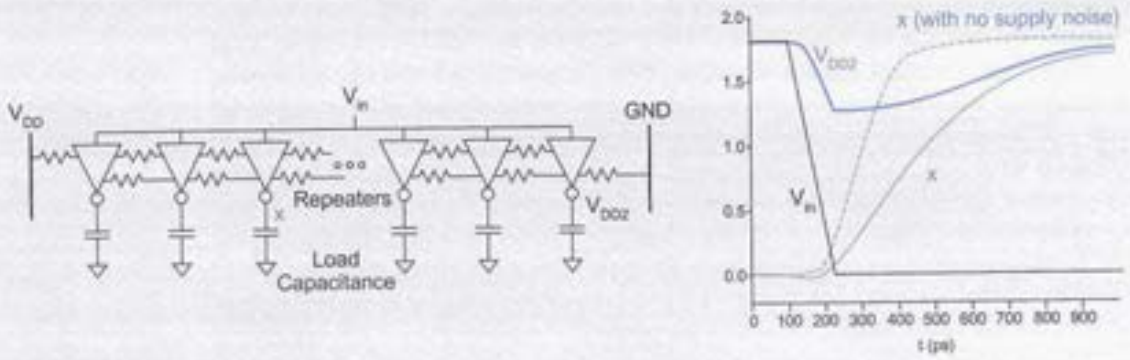


FIG 12.8 Power supply droop

12.3.3 L di/dt Noise

The inductance of the power supply is typically dominated by the inductance of the bond wires or C4 bumps connecting the die to the package. A typical bond wire has an inductance of about 1 nH/mm, while a C4 ball is on the order of 100 pH. Recall that the inductance of multiple inductors in parallel is reduced. Modern packages devote many (often 50% or more) of their pins to power and ground to minimize supply inductance. The two largest sources of current transients are switching I/O signals and changes between idle and active mode in the chip core.

Example

A 1 GHz chip transitions from idle to full power operation in a single cycle. The idle mode draws 20 A and the full power mode draws 60 A. If the power supply has 20 pH of series inductance, estimate the power supply noise caused by this transition if the chip has no internal bypass capacitance.

Solution: The current transient is:

$$\frac{\Delta I}{\Delta t} = \frac{(60\text{ A} - 20\text{ A})}{1\text{ ns}} = 40\text{ GA/s}$$

The inductive noise is L $\Delta I/\Delta t$ = 0.8 V. This is clearly unacceptable in a low-voltage process. Once again, the chip needs internal bypass capacitance to supply the instantaneous current, reducing the transient seen by the I/O pins.

L di/dt noise is becoming enough of a problem that some high-power systems must resort to microarchitectural solutions that prevent the chip from transitioning between minimum and maximum power in a single cycle. For example, a pipeline may enter or exit idle mode one stage at a time rather than all at once to spread the current change over many cycles.

12.3.4 On-chip Bypass Capacitance

As we have seen, chips need a substantial amount of capacitance between power and ground to provide the instantaneous current demands of the chip. This is called *bypass* or *decoupling* capacitance. The bypass capacitance is distributed across the chip so that a local spike in current can be supplied from nearby bypass capacitance rather than through the resistance of the overall power grid. It also greatly reduces the di/dt drawn from the package.

Example

How much bypass capacitance is needed to supply a sudden current spike of 40 A for 1 ns with no more than a 200 mV supply droop?

Solution: We solve

$$I = C\frac{\Delta V}{\Delta t} \Rightarrow C = \frac{(40\text{ A})(1\text{ ns})}{0.2\text{ V}} = 200\text{ nF}$$

Fortunately, the inherent gate capacitance of quiescent transistors provides a significant amount of *symbiotic* bypass capacitance [Dally98]. For example, Figure 12.9 shows one inverter driving another. The gate-to-source capacitances of the load inverter are shown explicitly. When $A = 1$ and $B = 0$, *M1* is ON, charging up C_{gs4}. Similarly, when $A = 0$ and $B = 1$, *M2* is ON, charging up C_{gs3}. The charged capacitor stores energy that can be released to supply sudden current demands. At any given time, approximately half of the gate capacitance of any quiescent circuit behaves as symbiotic bypass capacitance. Moreover, because only a small fraction of the gates are likely to be switching at any given time, nearly half of the entire gate capacitance on the chip will serve as bypass capacitance.

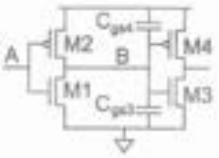


FIG 12.9 Symbiotic bypass capacitance

In most low- and medium-power chips, this symbiotic capacitance provides adequate bypassing to filter instantaneous IR drops and L di/dt noise. In high-power chips, additional explicit capacitance is sometimes necessary. The only dielectric available in a standard CMOS process to build compact high-capacitance structures is gate oxide, so the extra bypass capacitance is commonly built with an nMOS transistor with the gate tied to V_{DD} and the source and drain tied to GND, as shown in Figure 12.10(a).

Decoupling capacitor layout should maximize the capacitance per unit area. The transistor length is ideally as long as possible so that most of the area is devoted to gate oxide

Example

Estimate the symbiotic bypass capacitance per square millimeter for a chip with feature size f if gate capacitance is 2 fF/μm of transistor width and transistor gates occupy 9% of chip area.

Solution: The capacitance density of a 1 μm wide transistor of length f is $2/f$ fF/μm². At 9% utilization, this corresponds to $0.18/f$ nF/mm². Half of that, or $0.09/f$ nF/mm², serves as symbiotic bypass capacitance on average. In an f = 180 nm process, this means the symbiotic bypass capacitance is approximately 0.5 nF/mm².

capacitance rather than contacts. However, if the length is too great, the resistance of the ON transistor causes a slow response and ineffective bypass [Larsson97]. In practice, a length of 4–12 times minimum is acceptable. Figure 12.10(b) shows a "waffle" layout of a thin oxide decoupling capacitor. Most of the area consists of polysilicon over diffusion acting as capacitance. Occasional openings in the polysilicon are used to contact the diffusion to ground, while occasional field oxide bumps sit below polysilicon V_{DD} contacts (unless process design rules allow direct connection of metal to polysilicon over gate oxide). Small decoupling capacitors can be treated as unit cells. They are assembled into arrays by CAD tools. The high density of polysilicon and metal1 may increase process variation in nearby devices, so some tradeoffs in density or spacing to logic may be required. Decoupling capacitors must also be placed close (e.g., within about 200 μm in a 90 nm process) to the switching circuits they serve so they can deliver charge before much droop occurs. The direct V_{DD} connections to only gate oxide violate antenna rules in some processes. This can be avoided by coupling V_{DD} through a pMOS device, as shown in Figure 12.10(c). A typical cell using 0.35 μm I/O transistors achieves 320 fF/cell.

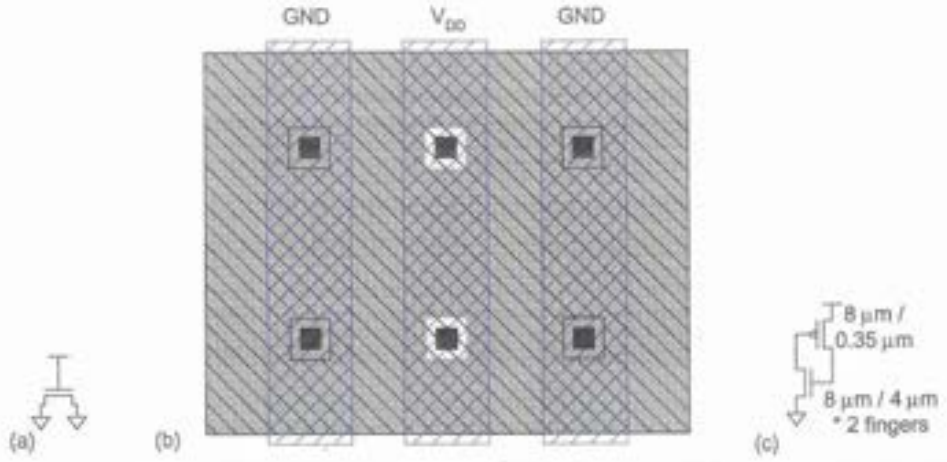


FIG 12.10 Waffle bypass capacitor layout

As gate oxide becomes thinner and tunneling increases, thin oxide capacitors may become undesirable because of their high leakage. This may force process engineers to offer an extra mask step to build specialized decoupling capacitors using either a medium-thickness oxide or a more exotic material with higher dielectric constant.

12.3.5 Power Network Modeling

Figure 12.11 shows a lumped model of the power distribution network for a system including the voltage regulator, the printed circuit board planes, the package, and the chip. The network also includes bypass capacitors near the voltage regulator, near the chip package, possibly inside the chip package, and definitely on chip. The external capacitors are modeled as an ideal capacitor with an effective series resistance (ESR) and effective series inductance (ESL) representing the parasitics of the capacitor package. Larger capacitors have bigger effective series inductances.

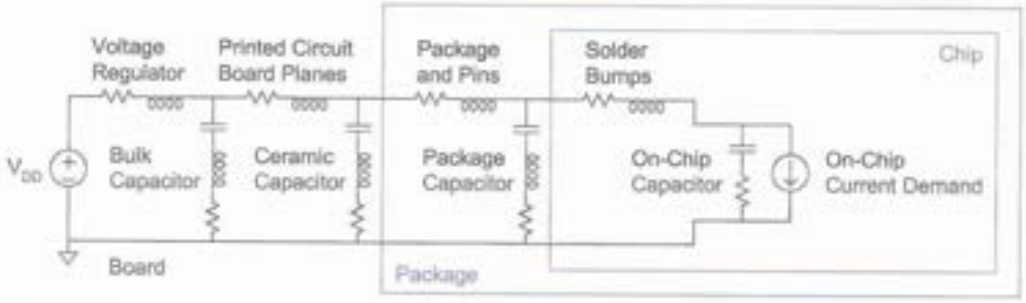


FIG 12.11 Power distribution system model

The voltage regulator seeks to produce a constant output voltage independent of the load current. It is modeled as an ideal voltage source in series with a small resistance and the inductance of its pins. Near the regulator is a large bulk capacitor (typically electrolytic or tantalum). Power and ground planes on the printed circuit board carry the supply current to the package, contributing some resistance and inductance. Typically, the board designer places several small ceramic capacitors near the package. The package and its pins again contribute resistance and inductance. High-frequency packages often contain small capacitors inside the package for further decoupling. Finally, the chip connects to the package through solder bumps or bond wires with additional resistance and inductance. The dynamic and static current demands of the chip are modeled as a variable current source with a waveform that might resemble Figure 12.5. The on-chip bypass capacitance consists of the symbiotic capacitance and possibly some explicit decoupling capacitance. It typically has negligible inductance because it is located so close to the switching loads.

12.3.5.1 Power Supply Impedance A good power distribution network should offer a low impedance at all frequencies of interest so that the supply voltage remains steady independent of the changing chip current demands. If the system had no bypass capacitance,

the distribution network would consist of only the resistance and inductance, so it would have an impedance of $Z = R + j\omega L$. This impedance increases with frequency ω and becomes unacceptably high for most systems by about 1 MHz.

The bypass capacitors in parallel with the supply provide an alternative low-impedance path at higher frequencies. An ideal capacitor has impedance that decreases with frequency as $Z = 1/j\omega C$. Unfortunately, the effective series inductance of the capacitors limits the useful frequency range of the real capacitor. The impedance of a capacitor C with effective series resistance R and inductance L is:

$$Z = \frac{1}{j\omega C} + R + j\omega L \tag{12.3}$$

This impedance has a minimum of $Z = R$ at the self-resonant frequency of

$$f_{\text{resonant}} = \frac{\omega_{\text{resonant}}}{2\pi} = \frac{1}{2\pi\sqrt{LC}} \tag{12.4}$$

Figure 12.12 plots the magnitude of the impedance of a 1 μF capacitor with 0.25 nH of series inductance and 0.03 Ω of series resistance. The capacitor has low impedance near its resonant frequency of 10 MHz, but higher impedance elsewhere.

Larger capacitors tend to have higher effective series inductances and therefore have lower self-resonant frequencies beyond which they are not useful. Thus, the system uses many capacitors of different sizes to provide low impedance over all the frequencies of interest. Also, capacitors closer to the chip are more useful at high frequencies because they have less inductance in the board and package between them and the chip. The bulk

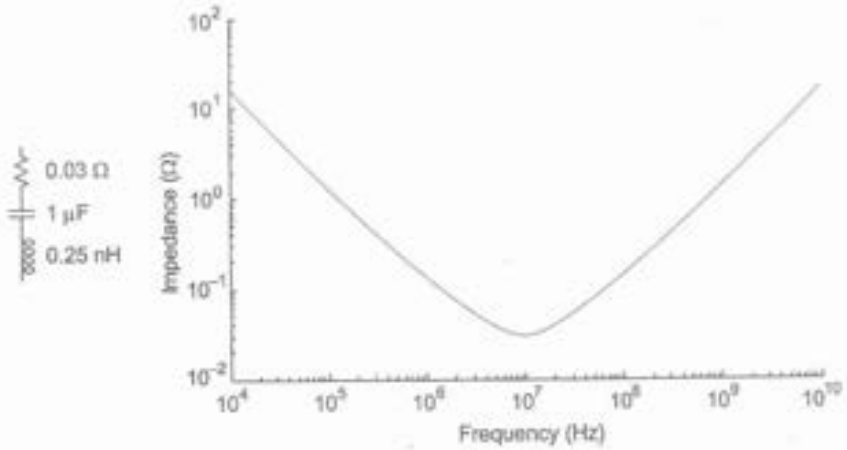


FIG 12.12 Impedance of bypass capacitor

and ceramic capacitors are most effective over the 1–10 MHz range. Capacitors in the package tend to be useful in the 10–200 MHz range. Above a few hundred MHz, the inductance of the solder bumps or bond wires renders all but the on-chip decoupling capacitors ineffective. Figure 12.13 plots an idealized supply impedance vs. frequency on a log-log scale for the complete power network model showing multiple resonances.

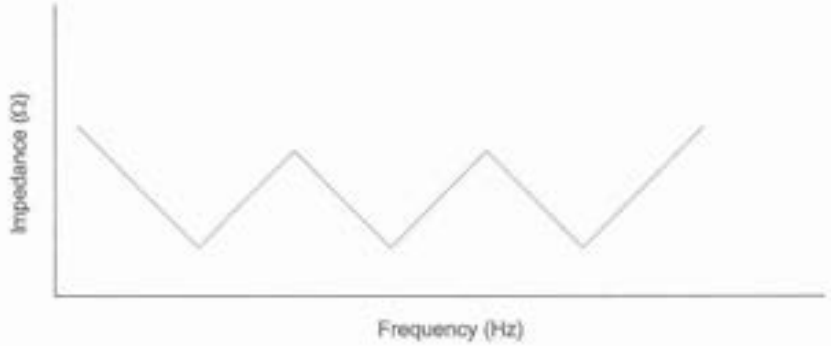


FIG 12.13 Supply impedance vs. frequency

The bypass capacitors also resonate with the series inductance of the package and solder bumps. This makes achieving low impedance across all frequencies difficult. Some packages have a high impedance somewhere in the 10–100 MHz range that is caused by such resonances. This is below the clock frequency of many chips. Power supply noise could be quite severe if the chip were to alternate between high- and low-power operations at the resonant frequency. This may be unlikely to occur by accident, but raises the threat of viruses that crash a computer by executing code that alternates power demand at just the wrong rate.

Another way to think about the need for nearby bypass capacitance is to imagine a sudden spike in current on the chip. Some round-trip propagation delay must occur before the spike reaches the power supply, the supply adjusts the current it is delivering, and that current returns to the chip. A lower bound on this delay is the speed of light. Therefore, when a gate switches, the voltage regulator will not know about the event until sometime after the transition has completed. The current to charge the load must come from the nearby bypass capacitors rather than the voltage regulator. This will cause a small droop on the power supply. Somewhat later, this droop will propagate to the regulator, which will boost its current and recharge the bypass capacitors.

12.3.5.2 Distributed Power Supply Models The model presented so far is a lumped approximation that is convenient for analysis and facilitates gaining intuition about chip behavior. Chip designers also are concerned about the variation in supply voltage across the chip. This requires a distributed model, which we can approximate with a mesh of small elements as shown in Figure 12.14. The mesh represents the resistance and induc-

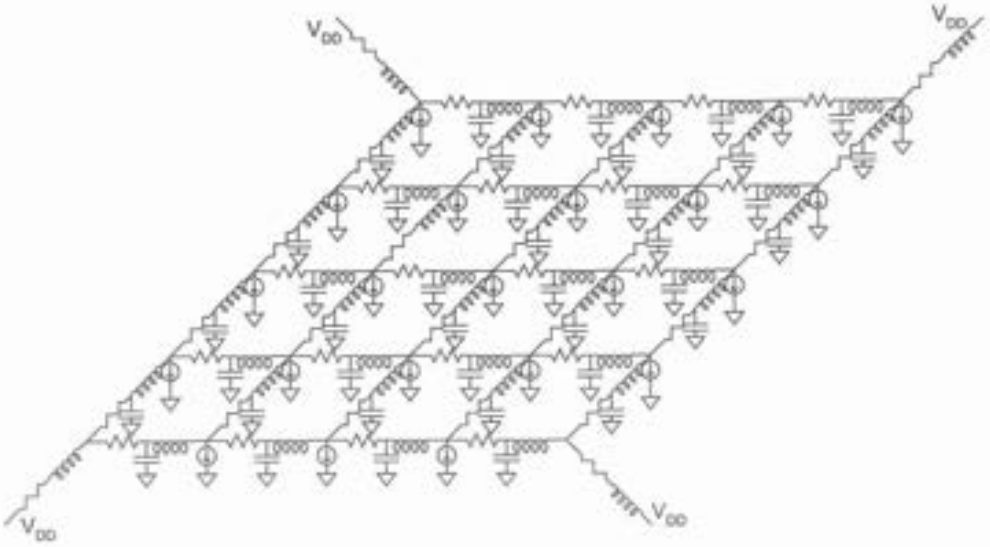


FIG 12.14 Distributed power supply model

tance of the on-chip power supply grid. Symbiotic or explicit decoupling capacitors are distributed across the chip. At each node, a current source represents the local current demand of the circuitry. The solder bumps or bond wires to the package are modeled with additional resistance and inductance. In this model, the package is treated as a perfect V_{DD} connected to the corners of the grid. In a more complex model, you also could add the distributed resistance, inductance, and bypass capacitance of the package itself.

For high-power chips, the designer can extract a mesh model as a SPICE netlist based on the power grid wiring and the amount of local decoupling capacitance. Different current waveforms can be applied at different nodes; for example, the current signatures of synthesized logic, SRAM, repeater banks, and domino logic are all quite different. The full-chip power grid simulation often takes many days to run and results in a map of voltage vs. time for the current pattern applied. Figure 6.56 shows a snapshot of the voltage droop on the Itanium 2 microprocessor. The droop was greatest in the integer execution unit, where several power-hungry domino adders all contribute to the *IR* drop.

12.3.6 Signal Return Paths

All current flows in loops. Whenever a signal is sent down a wire, the current eventually must return to the driver through the power or ground wires. Therefore, the power supply network also serves as a *signal return path*.

As we have seen, the impedance between V_{DD} and GND should be low at high frequency because of the decoupling capacitors. Hence, the V_{DD} wires are sometimes called

an *AC ground* because high-frequency alternating current can flow readily between power and ground through the capacitors. Therefore, both V_{DD} and GND grids serve as possible signal return paths regardless of whether the signal was 0 or 1.

As discussed in Section 4.5.5, the inductance of a signal wire is proportional to the area of its current loop. This inductance could increase delay or cause noise if it is too large, so a good power network provides return paths near all of the signal paths to keep the area of the current loops small. Signals sharing overlapping loop areas also couple onto each other through their mutual inductance.

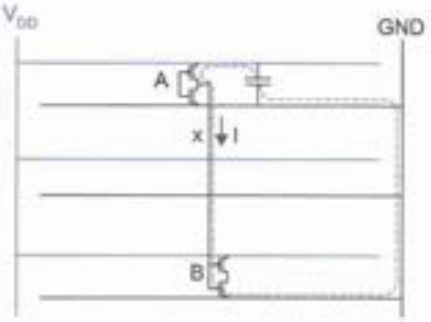


FIG 12.15 Fingered power network with long current return path

Figure 12.15 shows a circuit with a pathologically large current loop covering a large fraction of the chip area. The power and ground networks use a fingered layout [Johnson93]. Suppose the output of gate *A* rises. Some current *I* is drawn from the V_{DD} supply to charge up node *x* and gate *B*. The current flows through C_g of the nMOS transistor in gate *B* to the GND wire and back along the GND fingers. It eventually returns to V_{DD} through the on-chip decoupling capacitance. This large loop indicated by the dashed line results in high inductance. As the speed of light in a wire is set by its $\sqrt{LC}$ product, the signal becomes unusually slow. To avoid this problem, a power grid should occupy at least two orthogonal layers of metal rather than using fingers on one layer.

A common strategy is to place one V_{DD} or GND line for each *N* signal lines on each level of metal. A *signal-to-return* (SR) ratio such as $N = 10$ on at least two orthogonal upper levels of metal ensures that no path has outrageously long return paths that might cause "slow mode" wave propagation from the high inductance. Some engineers argue that an SR ratio of 2:1 is good practice at high frequencies [Morton02]. This ratio has the added benefit that every signal has either V_{DD} or GND as one of its neighbors, reducing capacitive crosstalk by a factor of two. However, it is quite expensive in terms of wiring resources.

12.3.7 Power Supply Filtering

Certain structures such as the phase-locked loop (PLL), clock buffers, and analog circuits are particularly sensitive to power supply noise. For example, supply noise on the clock buffers can directly increase clock jitter. Figure 12.16 shows an RC power supply filter circuit that eliminates the high-frequency noise on the local supply. The local filtered power supply is typically connected to the power grid through a single wire or solder bump. The resistance of this wire must be low enough to carry the current demand of the local circuitry without excessive *IR* drop, yet low enough to produce an RC time constant that will filter noise at frequencies of interest. Typically, this requires a huge filter capacitor as well, making power supply filtering expensive in terms of chip area.

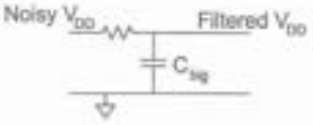


FIG 12.16 Power supply filter

For example, the Pentium 4 uses a power supply filter on the clock buffers to reduce clock jitter [Kurd01]. The filter attenuates typical supply noise from 10% to 2% of V_{DD} using a pMOS transistor as the resistor. It has an RC time constant of 2.5 ns with an *IR* drop of 70 mV.

12.3.8 Substrate Noise

The body terminal of a bulk CMOS transistor is connected to the substrate or well. The p-type substrate for an nMOS transistor is normally connected to GND and the n-well for a pMOS transistor is normally connected to V_{DD}. The connection is made through a relatively high-resistance substrate or well contact. Current flow in the substrate causes noise on the body terminal. This current may come from capacitive coupling through the reverse-biased source/drain to substrate diodes or from impact ionization as current flows through an ON transistor. The substrate noise modulates threshold voltages by means of the body effect.

Substrate noise is also a problem for *mixed-signal* designs where separate power supplies are used for noisy digital circuits and quiet analog circuits. The large number of rapidly switching digital circuits creates noise on the digital ground that propagates to the sensitive analog circuitry via the common substrate.

The substrate and well should use plenty of contacts to guarantee a low-resistance path to the power network. Guard rings, described in Section 4.8.5, provide some protection against noise caused by currents in the nearby substrate. Analog circuits should be physically separated from digital circuits and protected by guard rings connected to a quiet analog supply. Twin-tub or triple-well processes and SOI also experience much less substrate coupling because transistors are isolated in their own wells.

Modeling and analyzing substrate noise is beyond the scope of this book. See [Donnay03] for extensive coverage of the subject.

12.4 I/O

The input/output (I/O) subsystem is responsible for communicating data between the chip and the external world. A good I/O subsystem has the following properties:

- Drives large capacitances typical of off-chip signals
- Operates at voltage levels compatible with other chips
- Provides adequate bandwidth
- Limits slew rates to control high-frequency noise
- Protects chip against damage from electrostatic discharge (ESD)
- Protects against over-voltage damage
- Has a small number of pins (low cost)

I/O pad design requires specialized analog expertise and knowledge of process-specific ESD structures. [Dabral98] describes many of these issues. Thus, the system designer should obtain a well-characterized pad library from the processor or library vendor that is suited to the manufacturing process. This section summarizes some of the basic design options in I/O subsystems.

A pad consists of a square of top-level metal of approximately 100 μm on a side that is either soldered to a bond wire connecting to the package or coated with a lead solder ball. The term *pad* sometimes refers to just the metal square and other times to the complete cell containing the metal, ESD protection circuitry, and I/O transistors. Input and output pads usually contain built-in receiver and driver circuits to perform level conversion and amplification. Section 12.4.1 addresses some of these basic circuits, along with their ESD and overvoltage protection circuits. Section 12.4.2 examines the MOSIS 1.6 μm I/O pads to illustrate basic pad design.

12.4.1 Basic I/O Pad Circuits

Basic I/O pads include V_{DD} and GND, digital input, output, and bidirectional pads, and analog pads. This section is primarily concerned with simple digital pads that operate on the same CMOS voltage levels as the core.

12.4.1.1 V_{DD} and GND Pads Power and ground pads are simply squares of metal connected to the package and the on-chip power grid. Most high-performance chips devote about half of their pins to power and ground. This large number of pins is required to carry the high current and to provide low supply inductance.

One of the largest sources of noise in many chips is the ground bounce caused when output pads switch. The pads must rapidly charge the large external capacitive loads, causing a big current spike and high L di/dt noise. The problem is especially bad when many pins switch simultaneously, as could be the case in a 64-bit off-chip data bus. Such busses should be interdigitated with many power and ground pins to supply the output current through a low-inductance path. In many designs, the *dirty* power and ground lines serving the output pads are separated from the main power grid to reduce the coupling of I/O-related noise into the core.

12.4.1.2 Output Pads First and foremost, an output pad must have sufficient drive capability to deliver adequate rise and fall times into a given capacitive load. If the pad drives resistive loads, it must also deliver enough current to meet the required DC transfer characteristics. Given a load capacitance (typically 2–50 pF) and a rise/fall time specification, the output transistor widths can be calculated or determined through simulation. Typically, these transistors must be very wide and are folded into many legs.

Output pads generally contain additional buffering to reduce the load seen by the on-chip circuitry driving the pad. The method of logical effort tells us that the fastest buffers are built from strings of inverters with fanouts of about 4. In practice, a higher fanout (e.g., 6–8) gives nearly as good delay while reducing the area and power consumption of the buffer. The final stage may have an especially high fanout because the edge rates in the external world are normally an order of magnitude longer than those on chip. However, the final stage must be large enough to source or sink reasonable amounts of current with a small voltage drop.

Latchup is a particular problem near output pads, especially when the pads experience voltage transients above V_{DD} or below GND. These transients are likely to occur because of ringing from the bond wire inductance and/or from driving improperly terminated

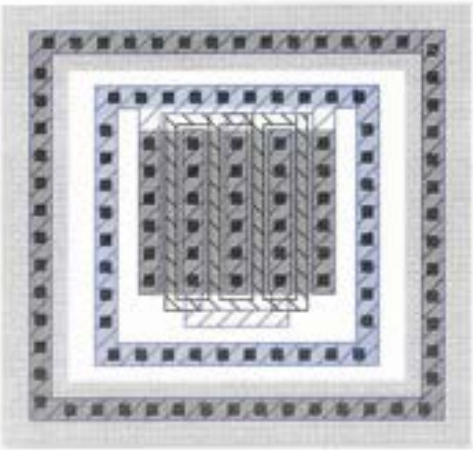


FIG 12.17 Double guard rings around folded nMOS output transistor

transmission lines. These transients cause the drain-to-body diodes to become forward-biased, forcing current to flow into the substrate or well and potentially causing latchup.

To avoid latchup, the nMOS and pMOS transistors should be separated by substantial distances and surrounded by guard rings. If possible, the output transistors (i.e., those whose drains connect directly to external circuitry) should be doubly guard-ringed, as shown in Figure 12.17. This means that an n-transistor should be encircled with p+ substrate contacts connected to GND, and then further encircled with n+ well contacts in an n-well connected to V_{DD}. The rings should be continuous in diffusion with frequent contacts to metal. Furthermore, dummy collectors consisting of p+ connections to GND and n+ in n-well connections to V_{DD} should be placed between the output transistors and any internal circuitry. These dummy collectors and guard rings serve to capture most of the stray carriers injected into the substrate when the diodes are forward-biased.

The output transistors also often have gates longer than normal to prevent avalanche breakdown damage when overvoltage is applied to the drains. Non-silicided gates are also preferable because the polysilicon gate resistance better distributes overvoltage across the legs of the output transistor, preventing damage.

12.4.1.3 Input Pads Input pads also contain an inverter or buffer as a level of protection between the pad and core circuitry. The buffer can perform some level conversion or noise filtering as well. For example, an input pad might receive a TTL signal with V_{OL} = 0.4 V and V_{OH} = 2.4 V. If the pad operates at 5 V, it should use a LO-skewed inverter with a trip point well below 2.4 V to properly receive the signal. Pads can include pull-up or pull-down resistors to place an unconnected pad in a certain state. Remember that even these structures must be guard-ringed if implemented in diffusion.

Some input pads also contain *Schmitt triggers*, as shown in Figure 12.18 [Schmitt38]. A Schmitt trigger has hysteresis that raises the switching point when the input is low and lowers the switching point when the input is high. This helps filter out glitches that might occur if the input rises slowly or is rather noisy.

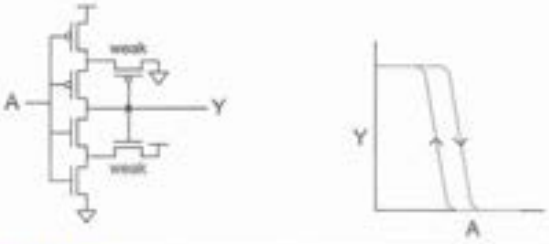


FIG 12.18 Schmitt trigger

On a dry day, you have probably experienced a shock when you touch a metal object because you have built up so much charge on your body while walking on carpets. Such shocks can destroy integrated circuits. Input pads have transistor gates connected directly to the external world. These gates are subject to damage from electrostatic discharge that can puncture and break down the oxide. The breakdown voltage was 40–100 V for older processes with thick (> 100 Å) oxides but now is 5 V or less for modern thin oxides.

Usually a combination of resistance and diode clamps is used to protect the gate oxides, as shown in Figure 12.19. The

diodes turn on if the pad voltage becomes greater than about V_{DD} + 0.7 V or less than −0.7 V. The resistor limits the peak current that flows into these diodes, preventing burnout. Resistor values anywhere from 100 Ω to 3 kΩ are used. This resistance, in conjunction with any input capacitance C, will lead to an RC time constant that can be important for high-speed circuits. The resistors are sometimes made from several squares of unsilicided p+ diffusion in an n-well. Clamping diodes are formed using n+ diffusion to the substrate and p+ diffusion to n-wells. As with output transistors, these diodes should be double guard-ringed so that they do not inject charge into the substrate and cause latchup. Some processes offer extra mask steps to build transistors with thicker oxides. These thick-oxide transistors are more robust against ESD and can operate at I/O voltages greater than the core voltage.

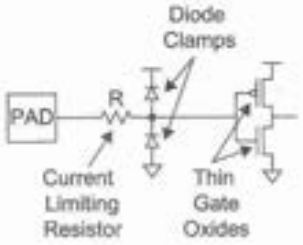


FIG 12.19 Input protection circuitry

ESD protection circuits are tested by zapping the pin with an external high voltage. Figure 12.20 shows a test circuit for the *Human Body Model* (HBM) of electrostatic discharge. This model simulates the discharge that takes place when a person who is improperly grounded touches a pin of a chip. The human is represented as a 100 pF capacitor in series with a 1500 Ω resistor. During test, the capacitor is charged to a high voltage, and then discharged through the resistor into the pad of the chip. The ESD robustness of the pad is measured as the maximum voltage that the pad can endure. For example, ±15 kV is good for parts such as serial port transceivers that might be exposed to ESD by an end user who handles a cable. Parts in an enclosed system are only subject to damage during assembly and can allow limits in the 2–4 kV range.

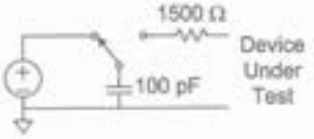


FIG 12.20 Human Body Model ESD test circuit

12.4.1.4 Bidirectional Pads Figure 12.21 shows a bidirectional pad with an output driver that can be tristated and an input receiver. The output driver consists of independently controlled nMOS and pMOS transistors. When the enable is '1,' one of the two transistors turns ON. When the enable is '0,' both transistors are OFF so the pad is tristated. This design is preferable to the four-transistor "totem pole" tristate from Section 1.4.7 when driving large capacitances because it has only two rather than four huge transistors in the final stage and the transistors need only be half as wide. Figure 12.22 shows a clever variation on this design in which the NAND and NOR are merged together into a single 6-transistor network with two outputs. Such a tristate buffer is smaller and presents less input capacitance on the D_{out} terminal.

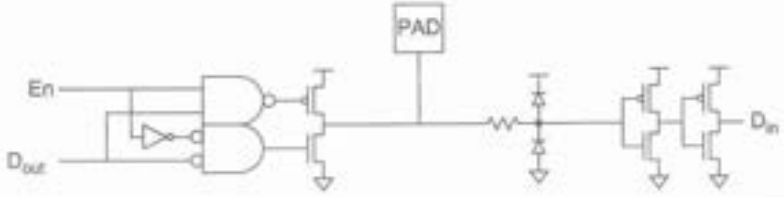


FIG 12.21 Bidirectional pad circuitry

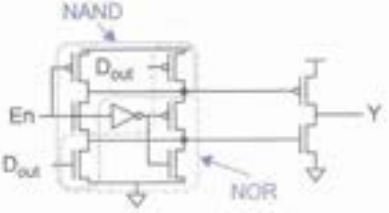


FIG 12.22 Improved tristate buffer

Many pad libraries provide only a bidirectional pad. By hardwiring the enable signal to '1' or '0', the pad can be used as an output or input.

12.4.1.5 Analog Pads Analog inputs and outputs connect to simple metal pads and then directly to the on-chip analog circuitry without any digital buffer or driver. Analog pads still require protection circuitry. The protection circuitry must be carefully designed so it does not degrade the bandwidth or signal integrity of the analog components. This is achieved by minimizing the protection diode area. RF pads are extremely demanding because any extra load can compromise performance.

12.4.2 Example: MOSIS I/O Pads

Figure 12.23 shows a layout of a bidirectional pad from the MOSIS service for a 1.6 μm two-metal layer process (see also the inside front cover). The overall cell is about 200 μm on a side. The pad is the large (100 x 75 μm) rectangle consisting of a sandwich of metal1 and metal2 connected with many vias. The SiO_2 overglass covering the metal2 is etched away over the pad so the bond wire can be connected directly to the pad. Two large metal2 rectangles cover most of the pad. The upper one with the legs sticking up is GND, while the lower is V_{DD}.

The bidirectional pad schematic is shown in Figure 12.24. The input protection circuitry consists of some resistance, a thick oxide transistor, and the drain diffusion diodes of the wide output transistors. The resistors are n+ and p+ diffusion wires, each 3.5 squares long. They have nominal sheet resistances of 53 and 75 Ω/□, so the parallel combination of resistance is 150 Ω. To the left and right of the metal pad are thick oxide nMOS transistors consisting of interdigitated fingers. They consist of a source and drain separated by 3 λ, but have no gate. They help protect the pad from ESD because high voltages will punch through the channel and dissipate. The effectiveness of thick oxide transistors is process-dependent. The pad uses many substrate/well contacts and is surrounded by double guard rings to prevent latchup during ESD events. The tristate driver and receiver use extensively folded transistors to fit in the space available.

12.4.3 Level Converters

Many chips require a low core voltage for the logic transistors, yet must interface with other chips operating at higher voltages. The I/O pads thus can include level converter circuits to translate between different voltage standards. As introduced in Section 3.2.7, these circuits often use transistors with longer channels and thicker oxides to endure the higher voltages.

Figure 12.25 shows some simple level converters for chips using a low V_{DDL} core voltage and higher V_{DDH} I/O voltage. Figure 12.25(a) is an output driver that takes a low-swing input voltage and produces a higher-swing output voltage. It uses a CVSL structure consisting of four high-voltage transistors indicated in bold. The inverter uses low-voltage transistors and the low-voltage power supply. The output Y can be followed by a high-voltage inverter or buffer to deliver more uniform rise/fall times. Figure 12.25(b) is an input receiver that takes a high-swing input voltage and produces a lower-swing voltage

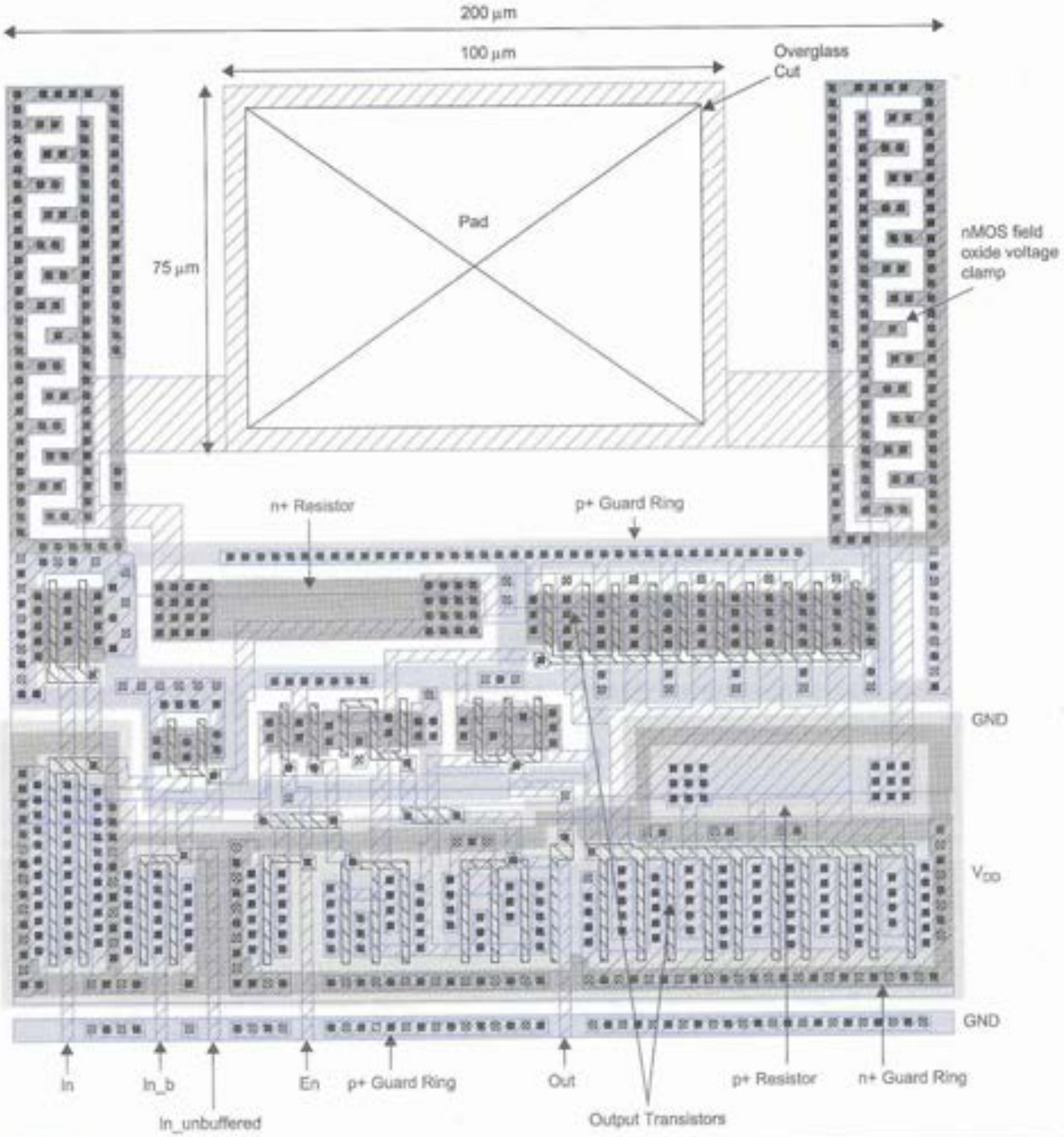


FIG 12.23 MOSIS 1.6 μm bidirectional pad. Color version on inside front cover.

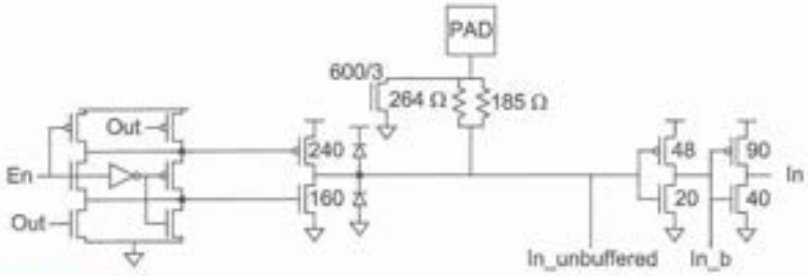


FIG 12.24 MOSIS bidirectional pad schematic

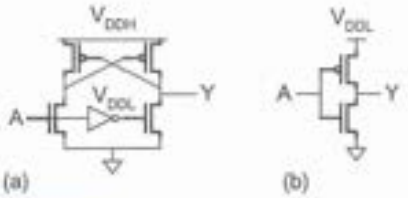


FIG 12.25 Level converters

for core circuits. It consists of a simple inverter using high-voltage transistors that can withstand the large gate voltages, but uses the lower voltage supply.

12.5 Clock

Synchronous systems use a clock to distinguish one step in a computation from the previous or next step. Ideally, this clock should arrive at all clocked elements in the system simultaneously so that the system shares a common time reference. These elements include latches and flip-flops, memories, and dynamic gates. In practice, the arrival time differs somewhat from one point to another; this difference is called *clock skew*. The central challenge in clock system design is to deliver the clock to all the clocked elements on the chip while finding an acceptable compromise among skew, power consumption, metal resource usage, and design effort.

12.5.1 Definitions

A system is designed to use one or more *logical clocks*. The logical clocks are idealized signals with no skew used by the logic designer when describing the system with a hardware description language. For example, a system with flip-flops requires a single logical clock, usually called *clk*. A system using two-phase transparent latches requires two logical clocks ϕ_1 and ϕ_2 (or `ph1` and `ph2` in a hardware description language). Unfortunately, mismatched clock network paths and processing and environmental variations make it impossible for all clocks to arrive at their ideal times, so the designer must settle for actually receiving a multitude of skewed *physical clocks*.

Distributing a single clock across the entire chip in a low-skew fashion is challenging. Distributing more than one is nearly impossible. Therefore, most systems distribute a single *global clock* even though they may need multiple logical clocks. *Local clock gaters* located near the clocked elements produce the physical clocks and drive them to the elements over short wires. Examples of clock gaters include buffers, AND gates to stop the clock to unused units, inverters to produce complementary clocks, and pulse generators for pulsed latches.

The term *clock skew* has been used informally in many ways. We will define skew precisely as the *difference between the nominal and actual interarrival time of a pair of physical clocks*. For example, Figure 12.26(a) shows a system with two flip-flops. Both should receive the logical clock *clk* with zero interarrival time, but they actually receive physical clocks clk_1 and clk_2. Because of differences in the delay of the clock distribution wires and the local clock buffers, clk_1 arrives 25 ps before clk_2. Therefore, we say the clock skew is 25 ps. Figure 12.26(b) shows a system with three transparent latches. The latches use complementary logical clocks ϕ_1 and ϕ_2 with a nominal interarrival time of $T_c/2$ between rising edges. They actually receive physical clocks ϕ_{1a}, ϕ_{1b} and ϕ_{2a}. We see the clock skews are:

$$t_{skew}^{\phi_{1a},\phi_{1b}} = 5 \text{ ps};\ t_{skew}^{\phi_{1a},\phi_{2a}} = 15 \text{ ps};\ t_{skew}^{\phi_{1b},\phi_{2a}} = 10 \text{ ps}$$

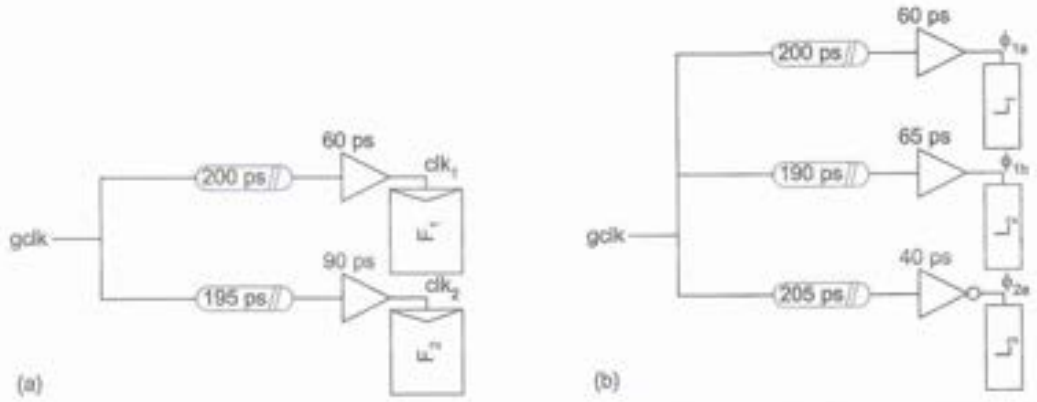


FIG 12.26 Clock skew example

Sometimes designers intentionally delay clocks to solve setup or hold time problems. For example, suppose that a critical path existed between F_1 and F_2 in Figure 12.26(a). The designer might intentionally delay the clock to F_2 by 30 ps to give the path more time by using the slower local clock buffer on clk_2. In this case, the nominal interarrival time of clk_1 and clk_2 is 30 ps. The actual interarrival time is 25 ps, so the clock skew is 5 ps. Some designers call this 30 ps delay *intentional skew*. We prefer to call it *intentional delay* and reserve the term *clock skew* to account for unintentional differences in clock arrival times.

Clock skew can also be measured between different edges of the clock or between different cycles. For example, Figure 12.27 shows two physical clock waveforms in which the edges differ from their nominal timing. The clock skews are defined based on the edge (rising/falling) and the number of intervening cycles as well as the physical clock:

$$t_{skew}^{clk_1,clk_1,(r,r,0)} = 0 \text{ ps};\ t_{skew}^{clk_1,clk_1,(r,f,0)} = 30 \text{ ps};\ t_{skew}^{clk_1,clk_1,(r,r,1)} = 70 \text{ ps}$$
$$t_{skew}^{clk_1,clk_2,(r,r,0)} = 0 \text{ ps};\ t_{skew}^{clk_1,clk_2,(r,f,0)} = 0 \text{ ps};\ t_{skew}^{clk_1,clk_2,(r,r,1)} = 40 \text{ ps}$$

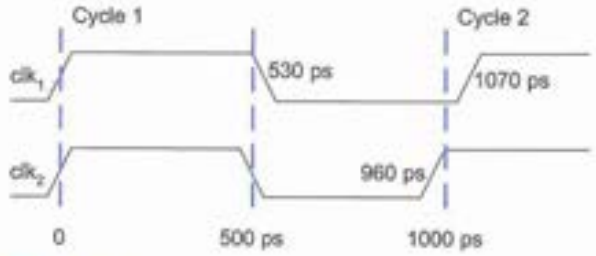


FIG 12.27 Skewed clock waveforms

For a path between two flip-flops, the hold time constraint depends on the skew between the same rising edges of both physical clocks. The setup time constraint depends on the skew between the rising edge of one physical clock and the subsequent rising edge of the other. We will see that clock distribution networks tend to introduce more skew from one cycle to the next so setup and hold time constraints can budget different amounts of skew.

The actual clock skew between two clocked elements varies with time and is different from one chip to another. Moreover, it is unknowable at design time. From the engineering perspective, a more useful parameter is the *clock skew budget*. The clock skew budget should be larger than the actual skew encountered on any long or short path on any working chip, yet no larger than necessary lest the chip be overdesigned.

While in principle designers could tabulate clock skew budgets between physical clocks at every pair of clocked elements on the chip, the table would be unreasonably large and unwieldy. Instead, they group physical clocks into *clock domains* and use a single skew budget to describe the entire domain. For example, you could define two latches to be in a local clock domain if their physical distance is no more than 500 μm. Then you could just define local and global skews, with the local skew being smaller than the global skew. If the clock period is long compared to the maximum skew, you can define only a single global skew budget and pessimistically assume all clocked elements might see this worst-case skew.

Clock skew sources can be classified as *systematic*, *random*, *drift*, and *jitter*. Figure 12.28(a) illustrates these sources in a simple clock distribution network. The global clock is distributed along wires to two gaters. One wire is 3 mm, while the other is 3.1 mm. The gaters are nominally identical, but one drives a lumped load of 1.3 pF while the other

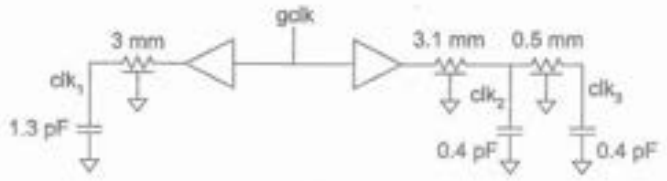


FIG 12.28 Simple clock distribution network

drives a load of 0.8 pF distributed along a 0.5 mm wire. The *systematic* clock skew is the portion that exists even under nominal conditions; this component can be predicted by simulation. By adjusting the size of one of the gaters, the systematic skew between clk_1 and clk_2 could be driven to zero. However, some systematic skew will always exist between clk_2 and clk_3 because of the flight time along the wire after the gater.

The *random* component of skew is caused by manufacturing variations that could affect the wire width, thickness, or spacing and the transistor channel length, threshold voltage, or oxide thickness. These cause unpredictable changes in resistance, capacitance, and transistor current, introducing additional skew. In principle, the actual random skew could be measured during chip test or on startup, and adjustable delay elements could be calibrated to compensate for the random skew.

Drift is caused by time-dependent environmental variations that occur relatively slowly. For example, after the chip turns on, it will heat up. The temperature affects gate and wire delay differently. Also, a temperature gradient across the chip leads to skew. Drift can also be nulled out with adjustable delay elements. Unlike random skew, compensating for drift must take place periodically rather than just once at startup. The frequency of calibration depends on the thermal time constant of the chip.

Jitter is caused by high-frequency environmental variation, particularly power supply noise. This noise leads to delay variation in the clock buffers and gaters in both time and space. Jitter is particularly insidious because it occurs too rapidly for compensation circuits to be able to counter it.

Some engineers do not report jitter as part of the skew. In such a case, they must include both jitter and skew in the setup and hold time budgets.

12.5.2 Clock System Architecture

Figure 12.29 shows an overview of a typical clock subsystem. The chip receives an external clock signal through the I/O pads. The clock generation unit may include a *phase-locked loop* (PLL) or *delay-locked loop* (DLL) to adjust the frequency or phase of the global clock, as shall be discussed in Section 12.5.3. This global clock is then distributed across the chip to points near all of the clocked elements. The clock distribution network must be carefully designed to minimize clock skew. Local clock gaters receive this global clock and drive the physical clock signals along short wires to small groups of clocked elements.

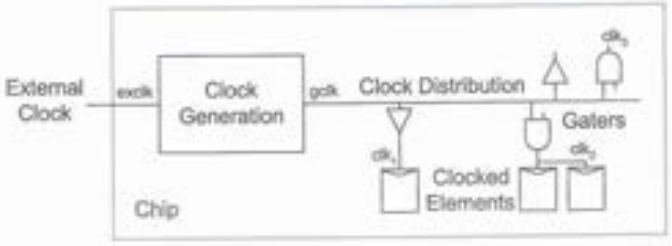


FIG 12.29 Clock subsystem

12.5.3 Global Clock Generation

The global clock generator receives an external clock signal and produces the global clock that will be distributed across the die. In this simplest case, the clock generator is simply a buffer to drive the large capacitance of the clock distribution network. However, the input pad, buffering, distribution network, and gaters have significant delay that leads to a large skew between the external clock and the physical clocks received at the clocked elements. Moreover, this delay varies with processing and environment. Because of this skew, clocked elements on the chip are no longer synchronized with the external I/O signals. Guaranteeing setup and hold times becomes problematic, particularly at high frequencies where the skew exceeds half of the clock period. More sophisticated clock generators use *phase-locked loops* (PLLs) to compensate for this delay. Moreover, phase-locked loops can perform frequency multiplication to provide an on-chip clock at a higher frequency than the external clock.

Figure 12.30 shows a typical synchronous chip interface using a phase-locked loop to compensate for on-chip clock delays [Chandrakasan01]. In this example, Chip 1 sends a clock and D_{out} to Chip 2 and receives D_{in} back from Chip 2. The data should be synchronized to the clock so each chip can sample it on the positive clock edge. However, the internal clock on Chip 2 is delayed through the clock distribution network. The PLL adjusts the internal clock in Chip 2 to correct for this delay and keep the clock synchronized with the data.

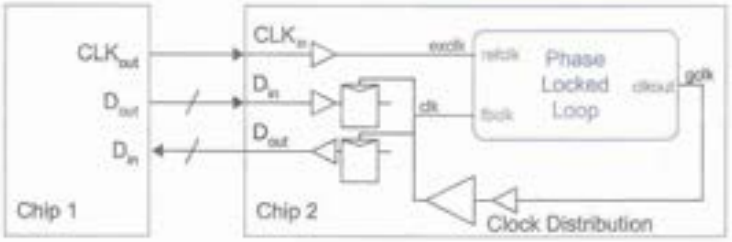


FIG 12.30 Synchronous chip interface with PLL

A phase-locked loop receives a *reference clock* and a *feedback clock* and produces an *output clock*. The phase and frequency of the output clock is automatically adjusted until the feedback clock is exactly aligned with the reference clock. In our application, the reference clock comes from Chip 1. The feedback clock is tapped off one of the physical clock wires that also drives the registers. The output clock is the *gclk* signal sent into the clock distribution network. Thus, the PLL adjusts *gclk* until the clock *clk* driving the data registers is aligned with the external clock *exclk*. This eliminates the systematic skew caused by the clock distribution delay.

Figure 12.31(a) shows a block diagram of a typical PLL. The heart of the PLL is a *voltage-controlled oscillator* (VCO). The control voltage is adjusted until the oscillator pro-

duces an output clock of the proper phase at the same frequency as the reference clock. This control is performed with a charge pump and RC filter. A phase detector determines whether the feedback clock leads or lags the reference clock. The charge pump consists of a pair of current sources enabled by the up and down signals to adjust the voltage on V_{ctrl} until the feedback clock becomes aligned with the reference.

Figure 12.31(b) shows a PLL that performs frequency multiplication. It uses a divide-by-N counter on the *fbclk* to generate a clock aligned in phase but at N times the frequency of the reference clock. For example, the Pentium 4 uses a 100 MHz external system clock that can be multiplied up to a 3 GHz core clock. With another divide-by-M counter on the *refclk* terminal, the PLL can produce any rational N/M multiple of the input frequency.

A PLL is a feedback system and requires careful analysis to ensure stability over all process and environmental corners. The RC loop filter behaves as a second-order system, although stray capacitance on V_{ctrl} often leads to third-order effects. Noise on V_{ctrl} causes the VCO to change frequency, which leads to a phase error that increases over time; this problem is called *phase-error accumulation* and appears as jitter in the clock skew budget. Therefore, the loop filter should have high enough bandwidth to rapidly correct for noise. Phase-error accumulation impacts the timing budget for one chip communicating with another. However, only cycle-cycle jitter is important for on-chip paths between flip-flops. RC loop filters are difficult to build because of uncertainties in the passive component values caused by process variation. Thus, many PLLs use an active filter.

Figure 12.32 shows a circuit-level implementation of a simple PLL. The phase detector is a *phase-frequency detector* consisting of a pair of flip-flops with asynchronous reset that pro-

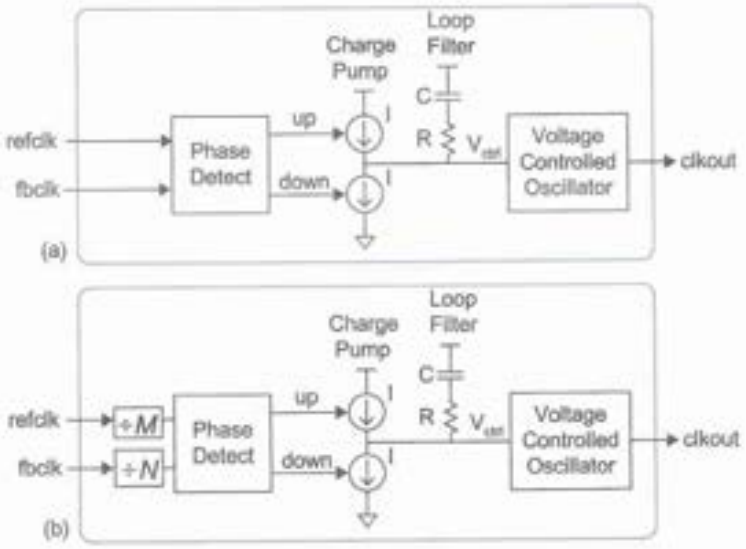


FIG 12.31 Phase-locked loop block diagram

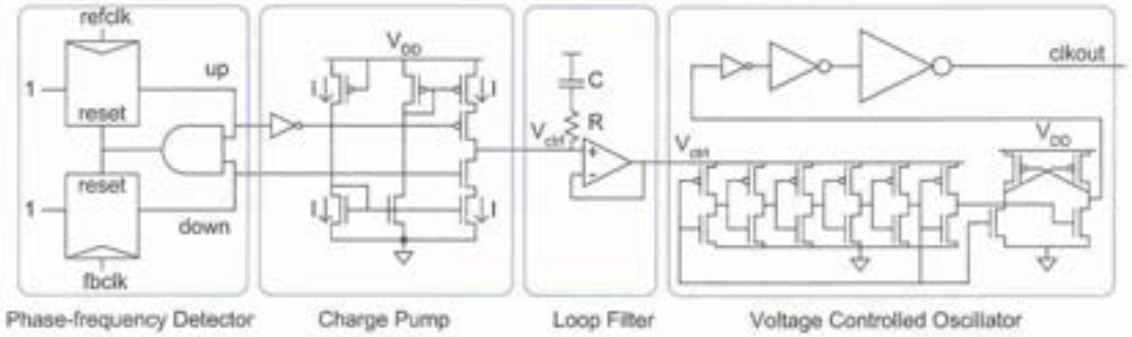


FIG 12.32 Simple PLL implementation

duces pulses to drive the frequency up or down depending on which clock arrived first. The charge pump uses a pair of current sources switched on by the *up* and *down* signals. Current sources are discussed further in Section 12.6.4. V_{ctl} is buffered with an amplifier such as that from Figure 12.63 hooked up as a unity-gain follower. The VCO consists of an ordinary ring oscillator running off V_{ctl} rather than V_{DD} so that its period increases as V_{ctl} decreases. The number of stages determines the range over which the frequency can be adjusted. A level converter and some buffers amplify the signal to drive *clkout* across the chip.

Power supply and substrate noise are the primary sources of PLL jitter, so the PLL should use a filtered power supply and generous guard rings to limit jitter. [Maneatis96, Maneatis03] describe self-biased delay elements that minimize sensitivity to supply noise and process variation. [Ingino01] describes a PLL with a filtered power supply operating up to 4 GHz with a ±25 ps peak-to-peak jitter.

A delay-locked loop (DLL) is a variant of a PLL that uses a voltage-controlled delay line rather than a voltage-controlled oscillator, as shown in Figure 12.33. It can be viewed as a control system that adjusts phase rather than frequency on *clkout*. DLLs are not capable of frequency multiplication. However, noise on V_{ctl} creates only a steady phase error, so DLLs avoid phase-error accumulation. DLLs use a first-order loop filter, so ensuring stability is also easier.

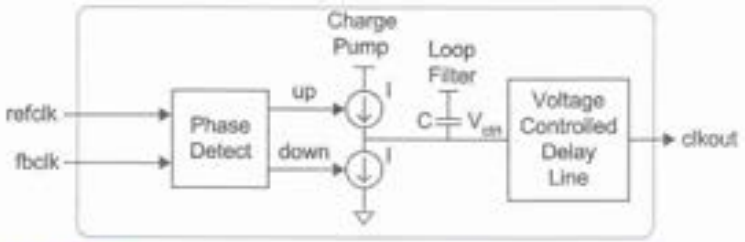


FIG 12.33 Delay-locked loop block diagram

PLLs and DLLs are notoriously difficult to build correctly. They require expertise in both feedback control systems and analog design. They must be carefully designed to acquire lock successfully and operate correctly with low jitter across process and environmental variations. A number of texts including [Chandrakasan01, Baker98, Dally98, Best03] offer introductions to loop design. For many applications, loops are best obtained as predesigned cells from a third party that specializes in loop design and offers a guarantee. True Circuits is a well-regarded PLL/DLL supplier.

12.5.4 Global Clock Distribution

The global clock must be distributed across the chip in a way that reaches all of the clocked elements at nearly the same time. In antiquated processes with slow transistors and fast wires, the clock wire had negligible delay and any convenient routing plan could be used to distribute the clock. In modern processes, the RC delay of the resistive clock wire driving its own capacitance and the clock load capacitance tends to be close to 1 ns for a well-designed distribution network covering a 15 mm square die. If the clock were routed randomly, this would lead to a clock skew of about 1 ns between physical clocks near and far from the clock generator. This could be several times the cycle time of the system. Thus, the clock distribution system must be carefully designed to equalize the *flight time* between the clock generator and the clocked receivers. Global clock distribution networks can be classified as *grids*, *H-trees*, *spines*, *ad-hoc*, or *hybrid* [Restle98].

Random skew, drift, and jitter from the clock distribution network are proportional to the delay through the network because they are caused by process or environmental variations in the distribution elements. Therefore, the designer should try to keep this distribution delay low. Unfortunately, as chips are getting larger, wires are getting slower, and clock loads are increasing, the distribution delay tends to go up even as cycle times are going down. In the past, systematic clock skew was the dominant component. Now, good clock distribution networks achieve low systematic skews, but the random, drift, and jitter components are becoming an increasing fraction of the cycle time.

12.5.4.1 Grids A clock grid is a mesh of horizontal and vertical wires driven from the middle or edges. The mesh is fine enough to deliver the clock to points nearby every clocked element. The resistance is low between any two nearby points in the mesh so the skew is also low between nearby clocked elements. This reduces the chance of hold-time problems because such problems tend to occur between nearby elements where the propagation delay between elements is also small. Grids also compensate for much of the random skew because shorting the clock together makes variations in delays irrelevant. They can be routed early in the design without detailed knowledge of latch placement. However, grids do have significant systematic skew between the points closest to the drivers and the points furthest away. They also consume a large amount of metal resources and hence have a high switching capacitance and power consumption. Section 12.8 traces the evolution of clock grids in the Alpha series of microprocessors.

12.5.4.2 H-Trees An H-tree is a fractal structure built by drawing an H shape, then recursively drawing H shapes on each of the vertices, as shown in Figure 12.34. With enough recursions, the H-tree can distribute a clock from the center to within an arbitrarily short distance of every point on the chip while maintaining exactly equal wire lengths. Buffers are added as necessary to serve as repeaters. If the clock loads were uniformly distributed around the chip, the H-tree would have zero systematic skew. Moreover, the trees tend to use less wire and thus have lower capacitance than grids [Restle98].

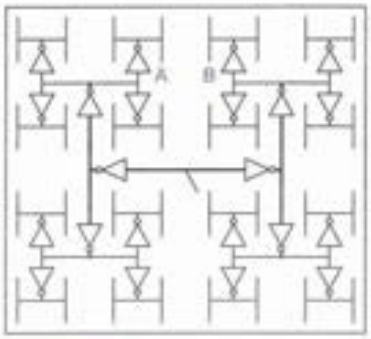


FIG 12.34 H-tree

In practice, the H-tree still shows some skew because the clock loads are not uniform, loading some leaves of the tree more than others. Moreover, the tree often must be routed around obstructions such as memory arrays. The leaves of the H do not reach every point on the chip, so some short physical clock wires are required after the local clock gater. Nevertheless, with careful tapering of the wires and sizing of the clock gaters, H-trees can deliver nearly zero systematic skew. A drawback of H-trees is that they may have high random skew, drift, and jitter between two nearby points that are leaves of different legs of the tree. For example, the points A and B in Figure 12.34 might experience large skews. As the points are close, this is a particular problem for hold times.

Figure 12.35 shows a modified H-tree used on the Itanium 2. The primary clock driver in the center of the chip sends a differential output to four differential repeaters on the leaves of the H. These repeaters drive a somewhat irregular pattern of wiring to second-level clock buffers (SLCBs) serving units all across the chip. The wiring and SLCB placement is determined by the nonuniform clock loads and obstructions on the chip. A custom clock router automatically generated the tree based on the actual clock loads so that the tree could be easily rerouted when loads change late in the design process. The SLCBs drive local clock gaters, producing the multitude of clock waveforms used on the microprocessor. Some of these waveforms were shown in Section 7.9.2.

Figure 12.36 shows the differential driver used as a primary clock buffer and repeater on the Itanium 2 [Anderson02]. The input stage is a differential amplifier sensitive to the point where the differential inputs cross over. The repeater pulses either p_1 or n_1 and p_2 or n_2 to switch the internal nodes y and $\bar{y}$. The small tristate keeper prevents these nodes from floating after the pulse terminates. The SLCB uses the same structure, but produces only a single-ended output. It also provides a current-starved adjustable delay line to compensate for systematic skew and to help locate critical paths during debug. The repeater provides a high drive capability with a low input capacitance. Thus, few stages of clock buffering are needed in the network. With so few repeaters, the area overhead of providing a filtered power supply is modest. Although the repeaters are relatively slow, their jitter is controlled with supply filtering.

12.5.4.3 Spines Figure 12.37 shows a clock distribution scheme using a pair of spines. As with the grid, the clock buffers are located in a few rows across the chip. However, instead of driving a single clock grid across the entire die, the spines drive length-matched serpentine wires to each small group of clocked elements. If the loads are uniform, the

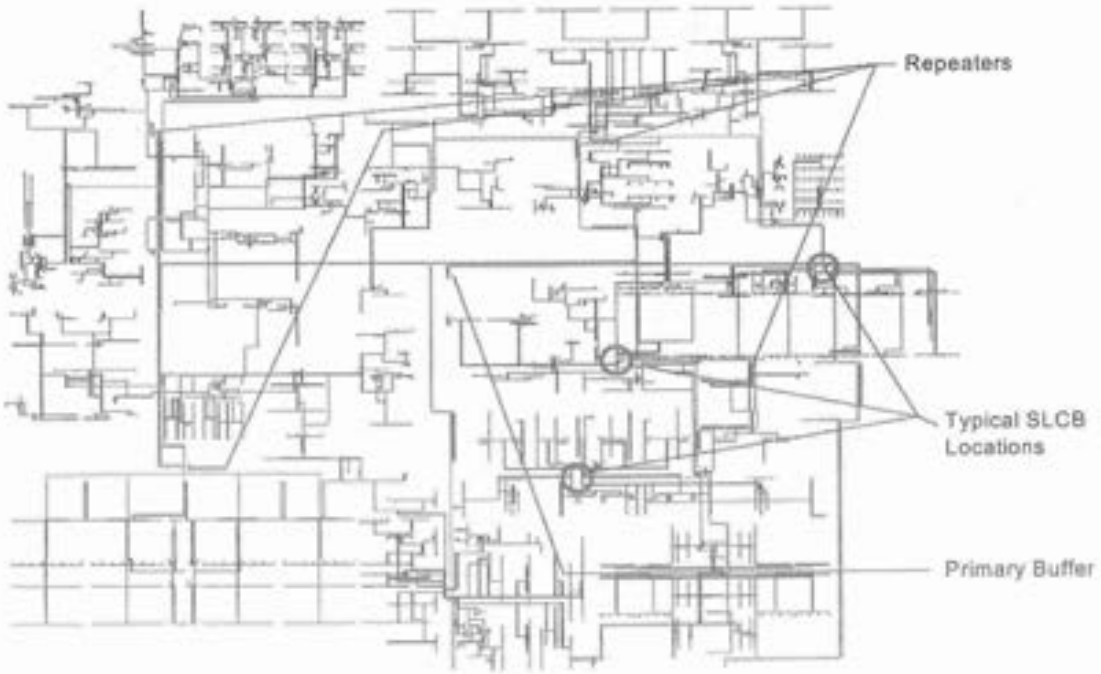


FIG 12.35 Itanium 2 modified H-tree

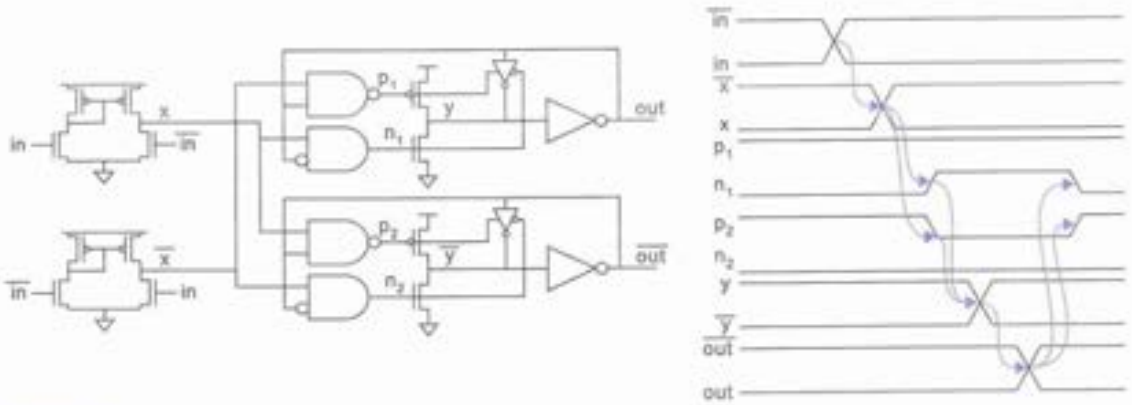


FIG 12.36 Itaniuim 2 repeater

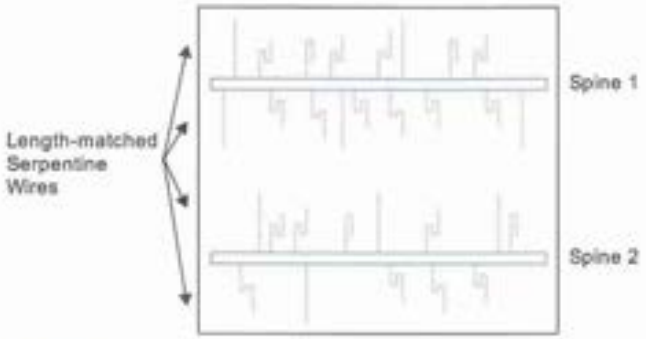


FIG 12.37 Clock spines with serpentine routing

spine avoids the systematic skew of the grid by matching the length of the clock wires. Each serpentine is driven individually so gaters can be used to save power by not switching certain wires. The serpentine is also easy to design and each load can be tuned individually. However, a system with many clocked elements may require a large number of serpentine routes, leading to high area and capacitance for the clock network. Like trees, spines also may have large local skews between nearby elements driven by different serpentines.

The Pentium II and III use a pair of clock spines [Geannopoulos98]. The Pentium 4 adds a third clock spine to reduce the length of the final clock wires [Kurd01]. Figure 12.38(a) shows the global clock buffers distributing the clock to the three spines on the Pentium 4 with zero systematic skew while Figure 12.38(b) shows a photograph of the chip annotated with the clock spine locations. The spines drive 47 independent clock domains, each of which can be gated individually. The clock domain gaters also contain adjustable delay buffers used to null out systematic and random skew and even to force deliberate clock delay to improve performance.

12.5.4.4 Ad-hoc Many ASICs running at relatively low frequencies (100's of MHz) still get away with an ad-hoc clock distribution network in which the clock is routed haphazardly with some attempt to equalize wire lengths or add buffers to equalize delay. Such ad-hoc networks can have reasonably low systematic skews because the buffer sizes can be adjusted until the nominal delays are nearly equal. However, they are subject to severe random skew when process variations affect wire and gate delays differently. This is the level that most commonly available tools support. Most design teams using ad-hoc clock networks also lack the resources to do a careful analysis of random skew, jitter, and drift. Therefore, they should be conservative in defining a skew budget and must be careful about hold time violations.

12.5.4.5 Hybrid A hybrid combination of the H-tree and grid offers lower skew than either an H-tree or grid alone. In the hybrid approach, an H-tree is used to distribute the clock to a large number of points across the die. A grid shorts these points together. Com-

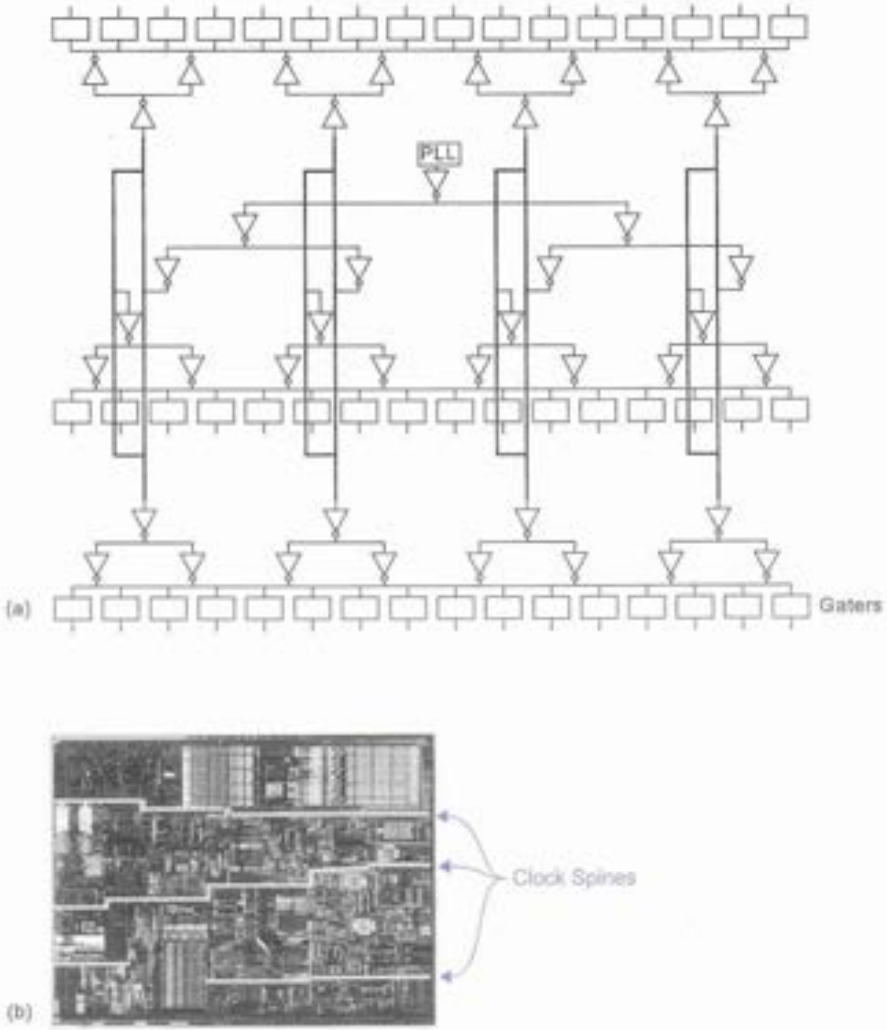


FIG 12.38 Pentium 4 clock spines. (b) © 2001 IEEE.

pared to a simple grid, the hybrid approach has lower systematic skew because the grid is driven from many points instead of just the middle or edge. Compared to an H-tree, the hybrid approach is less susceptible to skew from nonuniform load distributions. The grid also reduces local skew and brings the clock near every location where it is needed. Finally, the hybrid approach is regular, making layout of well-controlled transmission line structures easier.

IBM has used such a hybrid distribution network on a variety of microprocessors including the Power4, PowerPC, and S/390 [Restle01]. A primary buffered H-tree drives

16–64 sector buffers arranged across the chip. Each sector buffer drives a smaller tree network. Each tree can be tuned to accommodate nonuniform load capacitance by adjusting the wire widths. Together, the tunable trees drive the global clock grid at up to 1024 points. IBM uses a specialized tool to perform the tuning.

12.5.4.6 Layout Issues High-speed clock distribution networks require careful layout to minimize skew. The two guiding principles are that the network should be as uniform and as fast as possible. In a uniform network, chip-wide process or environmental variations should affect all clock paths identically. In a fast network, localized variations that cause a fractional difference between two clock path delays lead only to modest amounts of skew. For example, voltage noise that causes a 10% delay variation between two paths through an H-tree will lead to 80 ps of jitter if the tree delay is 800 ps, but 160 ps of jitter if the tree delay is 1600 ps.

Building a fast clock network requires low-resistance global clock wires with proper repeater insertion. The thick, top-level metal layer is well-suited to clock distribution. The wide wires should be shielded on both sides with V_{DD} or GND lines to prevent capacitive coupling between the clock and signal lines. The clock can even be shielded on a lower metal layer to form a microstrip waveguide [Anderson02].

Wide, low-resistance wires also have significant inductive effects, including faster than expected edge rates and overshoot. The fast edges are desirable, but overshoot should be minimized to prevent overvoltage damage. High-performance clock networks must be extracted with a field solver and modeled as transmission lines [Huang03]. Uniformity is again important: Even if the RC delays appear to be matched in a nonuniform layout, the *RLC* delays can be significantly different. As discussed in Section 4.5.5, wide wires should be split into multiple narrower traces interdigitated with V_{DD}/GND wires that provide a low-inductance current return path and minimize skin effect.

12.5.5 Local Clock Gaters

Local clock gaters receive the global clock and produce the physical clocks required by the clocked elements. The output of the gaters typically run a short distance (< 1 mm) to the clocked elements. Clock gaters are often used to stop or *gate* the clock to unused blocks of logic to save power. As discussed in Chapter 7, they can produce a variety of modified clock waveforms including pulsed clocks, delayed clocks, stretched clocks, nonoverlapping clocks, and double-frequency pulsed clocks. When used to modify the clock edges, they are sometimes called *clock choppers* or *clock stretchers*. Figure 12.39 shows a variety of clock gaters.

Most systems require a large number of clock gaters, so it is impractical to filter the power supply at every one. Variations in clock gater delay caused by voltage noise, cross-die process variation, and nonuniform temperature distribution cause skew between clocks produced by different gaters. The best way to limit this skew is to make the gater delay as short as possible. Variations in the input threshold of the clocked elements also causes skew. The best way to limit this skew is to produce crisp rise/fall times at the clock gaters. The final stage should have a fanout of no more than about 4.

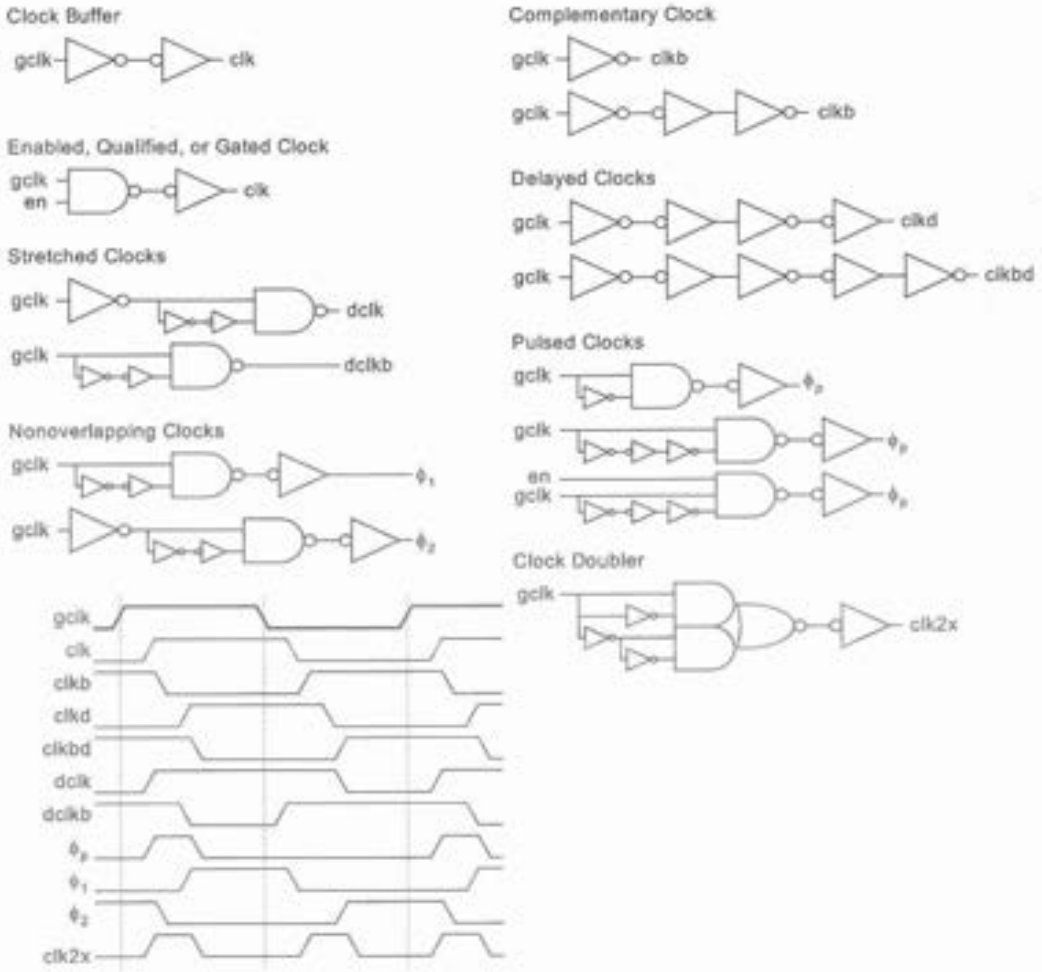


FIG 12.39 Clock gaters

Clock gaters may introduce some systematic delay between phases. For example, if *clkb* is produced with three inverters while *clk* is produced with only two, *clkb* may be delayed slightly from *clk*. The designer can either choose to carefully size the inverters such that the net delay is equal or accept that the delays are unequal and simply roll the systematic difference into timing analysis.

Figure 12.40 shows a circuit in which the delay of two inverters is matched against the delay of three when driving a fanout of *F*. The

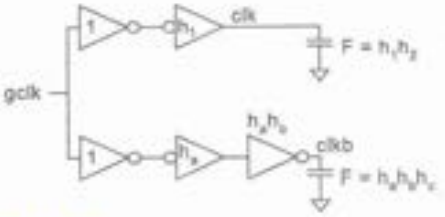


FIG 12.40 2- and 3-inverter paths

inverters are annotated with their size. The two inverters have electrical efforts of h_1 and h_2, respectively, while the three inverters have electrical efforts of h_a, h_b, and h_c. The electrical efforts should be chosen so that the delays of the chains are equal:

$$D = h_1 + h_2 + 2p_{inv} = h_a + h_b + h_c + 3p_{inv} \tag{12.5}$$

Even if the inverters have equal rise and fall delays in the TT corner, they will have unequal delays in the FS or SF corner. This can lead to skew between *clk* and *clkb* in these corners. However, if the delay of the second inverter in each chain is equal ($h_2 = h_b$), the two gaters will have equal delay in all process corners [Shoji86].

We can solve for the best electrical efforts that satisfy this constraint while giving least delay through the path. Recall that a path has least delay when its stage efforts are equal. Thus, choose $h_a = h_c = h^*$. This implies $h_1 = h^{*2}$. The delay of the first inverter in the *clk* path must equal the sum of the delays of the first and third inverter in the *clkb* path:

$$h^{*2} + p_{inv} = 2h^* + 2p_{inv} \tag{12.6}$$

This gives a quadratic equation that can be solved for h*:

$$h^* = 1 + \sqrt{1 + p_{inv}} \tag{12.7}$$

For $p_{inv} = 1$, this implies the best stage efforts are:

$$\begin{aligned} h_1 &= 5.8 \quad h_2 = \frac{F}{5.8} \\ h_a &= 2.4 \quad h_b = \frac{F}{5.8} \quad h_c = 2.4 \end{aligned} \tag{12.8}$$

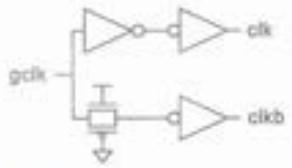


FIG 12.41 2- and 1-inverter paths

In this case, the rise/fall times of the different stages may be rather different, so the logical effort delay model is not especially accurate. These efforts make a good starting point, but further tuning should be done with a circuit simulator. The same approach can be used when the gater uses a NAND gate in place of one of the inverters.

Another approach is to try to match the delay of two inverters against one inverter and a transmission gate, as shown in Figure 12.41. This matching will not be perfect across all process corners. However, the gater may have less overall delay and hence produce less jitter from power supply noise.

12.5.6 Clock Skew Budgets

Developing an appropriate clock skew budget for design is a tricky process. The designer has a number of choices, including ignoring clock skew, budgeting worst-case clock skew everywhere, or budgeting different amounts of clock skew between different clock domains. Ultimately, the designer's objective is to build a system that achieves perfor-

mance targets and has no hold time failures while consuming as little area, power, and design effort as possible. The performance target can be a fixed number set by a standard or can simply be "as fast as possible."

It is possible to ignore clock skew if you are conservative about hold times and simply want the system to run as fast as possible. You must take reasonable care in the clock distribution network so that the skew between back-to-back flip-flops is unlikely to be too large. Many ASIC and FPGA flip-flops are designed with long contamination delays so they can tolerate significant skew before violating hold times. Build the system to run as fast as possible. When it is manufactured, clock skew will cause it to run slower than expected. The advantage of this methodology is that designers can be more productive because they do not need to think about clock skew. A disadvantage is that it uses slow flip-flops. Another drawback is that some paths really will have more skew than others. If all paths are designed to have equal delay, the paths with more skew will limit performance, while the other paths will be overdesigned and will consume more area and power than necessary. Moreover, if skew-tolerant circuit techniques are used in some places but not others, the non-tolerant circuits will tend to form the critical paths.

A related approach is to estimate the worst-case clock skew and budget it everywhere. In systems using only flip-flops, this can be done by designing to a shorter clock period. For example, if an ASIC must meet a 4 ns clock period and is predicted to have 500 ps of skew, it can be designed to meet a 3.5 ns clock period with no skew. This method requires work on the part of the clock designer to predict the clock skew, but still protects most of the designers from worrying about skew.

As cycle times get shorter than about 25 FO4 inverter delays, budgeting worst-case skew everywhere makes design impossible. Instead, multiple skew budgets must be developed that reflect smaller amounts of skew between elements in a local clock domain. This method entails more thought on the part of designers to take advantage of locality and requires a static timing analyzer that applies the appropriate skew. A good timing analyzer also properly handles skew-tolerant techniques such as transparent latches and domino gates with overlapping clocks [Harris99]. Be careful—some commercial timing analyzers such as PrimeTime do not handle these circuits well.

12.5.6.1 Clock Skew Sources As discussed earlier, clock skew comes from many sources. The output of the phase-locked loop has some jitter because of noise in the PLL and jitter in the external clock source. The clock distribution network introduces more skew from variations in the buffers and wire. The buffers may have different delays because of differences in V_{DD} and temperature, as well as random variations in their channel length and threshold voltages. The wire length and loading between buffers may not be perfectly matched. Each gater drives a physical clock along a wire, so clocked elements at different ends of the wire will see different RC delays. As mentioned in Section 2.3.2, the effective gate capacitance of the clocked loads depends on the switching activity of the source and drain. For some clocked elements, this causes significant data dependence in the clocked capacitance and the local wire delay.

For hold time checks, we are concerned with the skew between two consecutive clocked elements at a particular moment in time. For setup time checks, we are concerned

with the skew between elements from one cycle to the next. Jitter in the clock distribution network can affect the instantaneous clock period, so setup time skew budgets must include the cycle-to-cycle jitter of the entire clock distribution system even for elements in the same local clock domain. Hence, we can define separate clock skew budgets for setup time and hold time analyses.

The sources can be categorized as systematic, random, drift, and jitter. Recall that systematic skews can be modeled as extra delay and taken out of the skew budget if you are willing to do the modeling. Good clock distribution networks have close to zero systematic skew. Systematic and random skews can also be eliminated by calibrating delay lines, as will be discussed in Section 12.5.7. Drift occurs slowly enough that it can be eliminated by periodic recalibration of the delay lines. Ultimately, jitter is the most serious source of skew because it changes too rapidly to predict and counteract.

12.5.6.2 Statistical Clock Skew Budgeting The most conservative approach to estimating clock skew is to find the worst-case value of each skew source and sum these values. A real chip is unlikely to simultaneously see all of these worst cases, so such a sum is pessimistic and makes design of high-speed chips nearly impossible.

Most skew sources do not have Gaussian distributions, so taking the root sum square of the sources is inappropriate. A better approach is to perform a Monte Carlo simulation of the different skew sources to find the likely distribution of skews. The skew budget is selected at some point in this distribution. For hold times, the skew must be budgeted conservatively because the chip will not work if a hold time is violated. For example, the hold time skew budget can be selected so that 95–99% of chips will have no hold time violations.

If the goal is to build a chip that operates as fast as possible, any fixed amount of skew that affects all paths equally is irrelevant to the designer because there is nothing to do about it from the point of view of meeting setup times. However, if different paths experience different amounts of skew, a path that sees less skew can contain more logic than a path that sees a larger skew. Moreover, a path using skew-tolerant sequencing elements can contain more logic than a path between flip-flops. Hence, it is useful to predict the median skew seen in various clock domains for the purposes of setup time analysis.

As the systematic clock skew tends to be low, most clock skew sources occur from random process variations and noise. However, critical paths also experience random process variation and noise, so some will be slower than simulation predicts while others will be faster. If the chip is tuned until many critical paths have nearly the same cycle time in simulation, it is likely that a few paths will be slower than expected in the fabricated part and will limit the chip speed. It is improbable that the paths with worst-case variations in data delay are also those affected by the worst clock skew. Hence, a Monte Carlo simulation considering both variations in delay of the data paths and clock network will predict a smaller and more realistic clock skew budget [Harris01b].

Overall, choosing the appropriate clock skew budget is an ongoing source of research and debate among designers. In practice, many design teams seem to perform some calculations, and then fudge the numbers until the clock skew budget is about 10% of the cycle time. This strategy has historically led to functional chips most of the time, but becomes

more risky as cycle times decrease. Measured clock skew numbers reported in publications are notoriously optimistic; for example, [Mule02] finds an average reported skew of 3.2% of the cycle time in recent microprocessors. Part of the reason is that measuring the worst case skew is difficult. Measurements tend to be made at only a few clocked elements for a small number of clock cycles, while the chip must be designed to operate correctly for the largest skew seen anywhere on the chip anytime during its ~10^{17} cycle life span.

12.5.6.3 Case Study: Itanium 2 Skew Budget As illustrated in Section 12.5.4, the 1.0 GHz Itaniuim 2 uses an H-tree for clock distribution. The distribution network uses four levels of buffering between the PLL and the clocked elements. These buffers are called the primary driver (PD), repeater, second-level clock buffer (SLCB), and gater. The clocks are distributed on wide, shielded upper-level metal.

Table 12.2 lists the sources of variation impacting clock skew. Two adjacent pulsed latches sharing the same gater will see much less skew than two latches on opposite corners of the chip that have only the primary driver in common. Hence, we can define a hierarchy of four clock domains characterized by whether two elements share the same gater, SLCB, repeater, or only the PD. Setup times are concerned with cycle-to-cycle jitter, while hold times are not. In summary, we will define eight distinct skew budgets for these different scenarios [Harris01b]. The sources affecting each of these budgets are indicated in the eight columns.

The dominant components in the skew budget are the PLL jitter, the voltage, temperature, channel length, and threshold variations at each buffer, and the systematic mismatches in wire length and loading. Wire capacitance and resistance variations were considered negligible in the manufacturing process. Note that certain variations such as channel length of the primary driver have no effect on clock skew because they affect all physical clocks equally.

The buffer delays are 150 ps for the primary driver and repeater, 280 ps for the SLCB, and 180 ps for a simple gater, excluding wire RC flight times. The impact of the skew sources on each of these buffer delays includes:

- **Voltage**: The power network is designed to have less than ±100 mV noise on a 1.2 V supply. This full noise can be seen between any two points on the chip or from one cycle to the next at any given point; it exhibits little temporal or spatial locality. The voltage variation leads to a 13% delay change/100 mV.
- **Temperature**: The full-chip power simulation gives a temperature map shown in Figure 6.57 that predicts a variation of 20° C across the core (excluding the caches). This causes a 1.5% delay variation.
- **Channel length**: Transistors can experience systematic variations of up to ±12.5 nm from their nominal drawn L_e of 180 nm. This leads to a ±10% delay variation.
- **Threshold voltage:** The standard deviation in threshold voltages is 16.8 mV for small nMOS transistors (< 12.5 μm wide), 14.6 mV for small pMOS, 7.9 mV for wide nMOS, and 6.5 mV for wide pMOS. Monte Carlo simulations of the clock buffers show that this leads to a distribution of delays with a standard deviation of 2%.

Table 12.2 Clock skew components

	Component	Type	Same Clock Edge (hold)				Cycle-to-cycle (setup)			
			PD	Repeater	SLCB	Gater	PD	Repeater	SLCB	Gater
PLL	PLL Jitter	Jitter					x	x	x	x
PD	Voltage	Jitter					x	x	x	x
	Temperature	Drift								
	L_v	Rand								
	V_t	Rand								
	Wire	Syst	x				x			
Repeater	Voltage	Jitter	x				x	x	x	x
	Temperature	Drift	x				x			
	L_v	Rand	x				x			
	V_t	Rand	x				x			
	Loading	Syst	x				x			
	Wire	Syst	x	x			x	x		
SLCB	Voltage	Jitter	x	x			x	x	x	x
	Temperature	Drift	x	x			x	x		
	L_v	Rand	x	x			x	x		
	V_t	Rand	x	x			x	x		
	Loading	Syst	x	x			x	x		
	Wire	Syst	x	x	x		x	x	x	
Gater	Voltage	Jitter	x	x	x		x	x	x	x
	Temperature	Drift	x	x	x		x	x	x	
	L_v	Rand	x	x	x		x	x	x	
	V_t	Rand	x	x	x		x	x	x	
	Loading	Syst	x	x	x		x	x	x	
	Wire RC	Syst	x	x	x	x	x	x	x	x

- **Wire and loading**: The worst case differences in wire length and load capacitance were found through simulation of the routed H-tree.

Table 12.3 summarizes the magnitude of each of these sources of delay variation. Some sources are described by the half-range of a uniform distribution. Others are described by a standard deviation of a Gaussian distribution.

The skew budget was selected by performing a Monte Carlo simulation. $N = 400$ chips were simulated. For each chip, the systematic, random, and drift sources of skews were assigned a random value from their distribution. Jitter sources were assigned their worst case because they vary rapidly in time and the chip must work on all cycles. The

Table 12.3 Magnitude of skew sources (ps)

	Component	Half-range	Standard Deviation
PLL	PLL Jitter	7.5	
PD	Voltage	19.5	
	Temperature	n/a	
	L_e	n/a	
	V_t	n/a	
	Wire	1	
Repeater	Voltage	19.5	
	Temperature	1.1	
	L_e	15	
	V_t		5
	Loading	5	
	Wire	1.5	
SLCB	Voltage	36.4	
	Temperature	2.1	
	L_e	28	
	V_t		5.6
	Loading	10	
	Wire	4	
Gater	Voltage	23.4	
	Temperature	1.4	
	L_e	18	
	V_t		3.6
	Loading	7.5	
	Wire RC	10	

assignments were done for every clock buffer on the chip and the skew budget for that chip for each domain was defined as the maximum skew between any two elements in that domain. This gives another distribution of skews in each domain across the N chips. The setup time skew was taken from the median of the distribution to represent a "typical" chip. The hold time skew was selected at the 95th percentile in the distribution so that nearly all chips would have no hold time problems. Table 12.4 lists the skew budgets. Jitter represents more than 200 ps of the setup time skew budgets, indicating that power supply noise on the PLL and clock buffers is a major problem for the H-tree. A more careful investigation into the actual voltage noise seen at the buffers and a better effort to filter the supply noise at the buffers could offer a substantial improvement in this jitter. Hold time skew budgets are smaller because they suffer less from jitter.

Table 12.4 Clock skew budgets (ps)

Same Clock Edge (hold)				Cycle-to-cycle (setup)			
PD	Repeater	SLCB	Gater	PD	Repeater	SLCB	Gater
280	229	106	20	312	302	267	232

A more realistic budget considers that variation takes place in the logic paths as well as the clock distribution network. Thus, some critical paths will be longer than predicted while others will be shorter. This variation is called *data skew*. The total skew impacting a path is the sum of its clock and data skew. The cycle-limiting path is the one that has the worst total skew, not necessarily the one with just the worst clock skew. Another Monte Carlo simulation incorporating data skew gives total skew budgets shown in Table 12.5. The total skew is slightly higher than the clock skew alone in most cases. However, the difference between the skew in the PD and gater delays is substantially smaller. When the design goal is to maximize frequency, the difference between global and local setup time skew is most important because it indicates how much less logic can go in a global path than in a local path if all paths should be equally critical. Also note that the hold time skew in the PD domain decreased slightly because there were relatively few hold time paths crossing this domain and it was uncommon that one with bad data skew also experienced bad clock skew.

Table 12.5 Total skew budgets (ps)

Same Clock Edge (hold)				Cycle-to-cycle (setup)			
PD	Repeater	SLCB	Gater	PD	Repeater	SLCB	Gater
275	226	117	44	322	313	309	285

12.5.7 Adaptive Deskewing

Just as a PLL or a DLL can compensate for the overall clock distribution delay, additional adjustable delay buffers can compensate for mismatches in clock distribution delay along various paths. For example, the Pentium II and 4 use such buffers at the leaves of the clock spine to eliminate systematic and random variations in the clock distribution network. Figure 12.42 shows an example of a digitally adjustable delay line with eight levels of adjustment. The select signals use a *thermometer code*[1] to produce a monotonically decreasing propagation delay as more pass transistors are turned on.

In the Pentium II, a phase comparator checks the arrival times of the physical clocks and adjusts the digitally controlled delay lines to make all clocks arrive simultaneously. The loop bandwidth is low enough to ignore jitter, but high enough to compensate for tem-

[1]In an N-bit thermometer code, a number $n \in [0,N]$ is represented with n 1's in the least significant positions. For example, the number 3 is represented in an 8-bit thermometer code as 00000111.

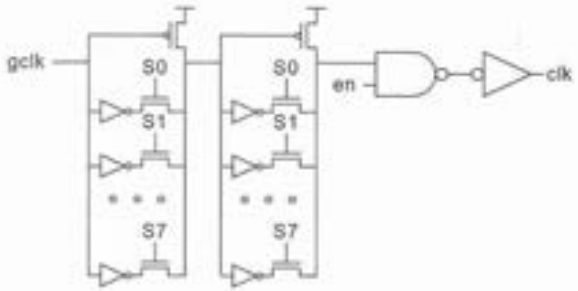


FIG 12.42 Digitally adjustable delay line

perature drift. This technique is sometimes called *adaptive deskewing* [Geannopoulos98]. In the Pentium 4, the delay line is adjusted using a scan chain through the boundary scan test access port. 46 phase comparators measure the phase of the clock gaters. Their results can be shifted out through the TAP. The delay lines can be adjusted to reduce systematic and random skew to ±8 ps, as compared to approximately 64 ps before adjustment. The delay lines can also deliberately delay certain clocks to improve performance or assist with debug [Kurd01]. The Itanium series of microprocessors uses similar deskew techniques [Tam00, Anderson02, Stinson03, Tam04]. In the 1.5 GHz Itanium 2, deskew takes place during manufacturing test; on-chip fuses are blown to eliminate the systematic and random skew without needing calibration upon reset or during normal operation.

A drawback of adaptive deskewing is that the buffers introduce extra delay. Voltage noise on the buffers appears as jitter. Unless all of the deskew buffers use well-filtered power supplies, the extra jitter from the deskew buffers can overwhelm the improvement in systematic and random skew.

12.5.8 Clocking Alternatives

A number of radical alternatives to standard clocking have been proposed. [Mule02] offers a survey of many of these methods.

Clock distribution consumes significant amounts of power and introduces much of the clock skew. On-chip metal wires have high resistance, contributing to the problems. Alternatively, the clock could be distributed on the printed circuit board or in the package using low-resistance transmission lines, and then brought onto the chip in many local regions. The clock could also be distributed optically to photodetectors around the chip or broadcast wirelessly using microwaves. Each of these methods suffers from the difficulty of testing the chip before it is packaged. If the package is expensive and the yield is not extremely high, testing chips after packaging adds significantly to the average part cost. Optical or wireless clocking also requires the development of on-chip photodetectors or antennae.

Another strategy is to distribute the clock on-chip, but use a novel architecture. Many oscillators can be distributed across the chip and operated in phase so clock distribution becomes localized [Gutnik00]. Rotary traveling-wave oscillator arrays exploit the inherent natural frequency, set by the clock network inductance and capacitance [Wood01]. If pumped at this frequency, it can oscillate by transferring the energy back and forth between magnetic and electric forms, reducing the power consumption.

Asynchronous systems eliminate the clock entirely. Proponents argue that no clocks means no clock skew and no clock power consumption. Advocates of synchronous systems point out that asynchronous systems still must distribute control signals to all the sequential elements and that variation in this delay appears as sequencing overhead in just the same way as clock skew. Moreover, the control signals dissipate power. [Sparsø01] provides a good tutorial introduction to some of the advantages of, challenges in, and techniques for asynchronous design. Synchronous designers have borrowed many techniques such as self-timed memories, self-resetting domino, and source-synchronous clocking that might once have been considered asynchronous. The debate has raged for decades, but nearly all commercial systems are still synchronous and will likely remain so indefinitely.

12.6 Analog Circuits

Although the emphasis of this book has been on digital circuits, system-on-chip designers commonly need to use some analog or radio-frequency (RF) circuitry to interface with the real world. This section offers a brief introduction to the subject of analog circuit design for digital designers. [Baker02, Gray01, Johns96] offer excellent coverage of CMOS analog circuit design while [Lee04, Razavi98] pioneered RF circuits in CMOS. The combination of analog and digital circuitry is naturally called *mixed-signal* design.

12.6.1 MOS Small-signal Model

Although the MOS transistor is a nonlinear device, it can be approximated as linear for small changes around a bias point. This is useful for understanding the behavior of amplifiers and other analog circuits. The currents and voltages can be written as

$$\begin{aligned} V_{gs} &= V_{GS} + v_{gs} \\ V_{ds} &= V_{DS} + v_{ds} \\ I_{ds} &= I_{DS} + i_{ds} \end{aligned} \tag{12.9}$$

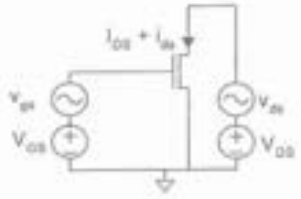


FIG 12.43 Small-signal variations around bias point

where the gate source voltage V_{gs} is expressed as a *bias-point* voltage V_{GS} plus a *small-signal* offset v_{gs}, as shown in Figure 12.43. The drain current also is expressed as a bias-point current I_{DS} plus a small-signal offset i_{ds} proportional to v_{gs} and v_{ds}.

MOS transistors are typically used in their saturation region in analog circuits. By expanding the saturation current model of EQ (2.8) around the operat-

ing point, we find the dependence of output current on small changes in input voltage.

$$
\begin{aligned}
I_{ds} &= \frac{\beta}{2}\left(V_{gs} - V_t\right)^2 \\
&= \frac{\beta}{2}\left(V_{GS} + v_{gs} - V_t\right)^2 \\
&= \frac{\beta}{2}\left[\left(V_{GS} - V_t\right)^2 + 2\left(V_{GS} - V_t\right)v_{gs} + v_{gs}^{\ 2}\right] \\
&= \underbrace{\frac{\beta}{2}\left(V_{GS} - V_t\right)^2}_{I_{DS}} + \underbrace{\beta\left(V_{GS} - V_t\right)v_{gs} + O\left(v_{gs}^{\ 2}\right)}_{i_{ds}}
\end{aligned}
\tag{12.10}
$$

If v_{gs} is small enough, the $O(v_{gs}^2)$ term is negligible.

In general, we can find the sensitivity of current to small changes in voltage by taking the first-order Taylor series around the operating point:

$$
I_{ds} = I_{DS} + \underbrace{\left.\frac{\partial I_{ds}}{\partial V_{gs}}\right|_{V_{gs}=V_{GS}} v_{gs} + \left.\frac{\partial I_{ds}}{\partial V_{ds}}\right|_{V_{ds}=V_{DS}} v_{ds}}_{i_{ds}}
\tag{12.11}
$$

We commonly write the sensitivities as

$$
i_{ds} = g_m v_{gs} + g_{ds} v_{ds}
\tag{12.12}
$$

where

$$
g_m = \left.\frac{\partial I_{ds}}{\partial V_{gs}}\right|_{V_{gs}=V_{GS}} = \beta\left(V_{GS} - V_t\right)
\tag{12.13}
$$

and because the saturation current is ideally independent of V_{ds},

$$
g_{ds} = \left.\frac{\partial I_{ds}}{\partial V_{ds}}\right|_{V_{ds}=V_{DS}} = 0
\tag{12.14}
$$

This Taylor series approach gives the same results as the direct expansion in EQ.(12.10).

In a real MOSFET, the output current does increase with V_{ds} because of channel length modulation. Considering EQ.(2.29),

$$
g_{ds} = \left.\frac{\partial I_{ds}}{\partial V_{ds}}\right|_{V_{ds}=V_{DS}} = \lambda\beta\frac{\left(V_{GS} - V_t\right)^2}{2} = \lambda I_{DS}
\tag{12.15}
$$

g_m is called the *transconductance* because it reflects the dependence of the drain current on the gate voltage. g_{ds} is the *output conductance*, reflecting the dependence of the drain current on the drain voltage. Here λ is the channel length modulation coefficient, not the unit of distance. Recall that λ is inversely dependent on channel length. Current sources and high-gain analog amplifiers require low output conductance and thus often use longer than minimum transistors. Often it is convenient to think about the reciprocal, *output resistance*.

$$r_0 = r_{ds} = \frac{1}{g_{ds}} = \frac{1}{\lambda I_{DS}} \tag{12.16}$$

Figure 12.44 shows the I-V characteristics of an nMOS transistor annotated with the bias-point and small-signal parameters. The transistor is biased in saturation at $V_{GS} = 0.9$, $V_{DS} = 1.0$. The transconductance and output conductance are the slope of the I_{ds} curve with respect to small changes in V_{gs} and V_{ds} around the operating point.

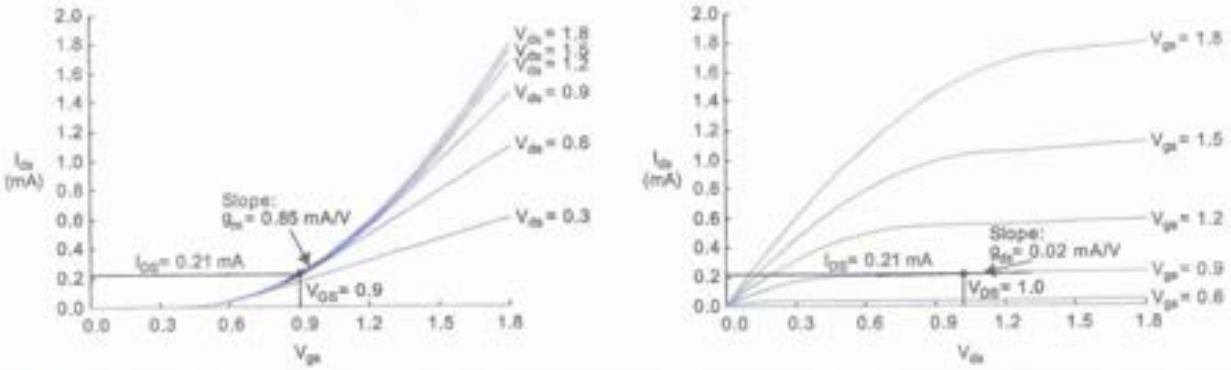


FIG 12.44 Bias-point and small-signal behavior

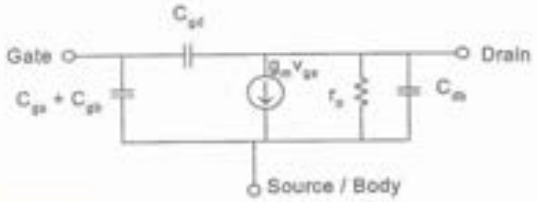


FIG 12.45 Small-signal model for an MOS transistor

Assuming the body is at the same potential as the source, we can model the transistor with the small-signal equivalent circuit of Figure 12.45 to relate small-signal voltages and currents. The current source reflects the dependence of i_{ds} on v_{gs} and the resistor reflects the dependence of i_{ds} on v_{ds}. The capacitors can be considered for high-frequency operation or ignored for low-frequency operation. The results of the small-signal model are added to the results of the bias point to find the overall behavior of the circuit.

12.6.2 Common Source Amplifier

Figure 12.46 shows a common source amplifier with a resistive load. The amplifier is biased at $V_{GS} = 0.9$ V, setting an output bias point V_{OUT}. If we increase V_g, by some small v_{in}, the transistor will turn on harder, pulling the output down by some small v_{out}. The gain of the amplifier is thus $A = v_{out} / v_{in}$.

Figure 12.47 shows the simulated DC transfer characteristics of the common source amplifier. The bias point is indicated on the graph. The slope of the transfer characteristic around the operating point is the gain. The transistor is a nonlinear device so its gain varies with the operating point. However, the slope and gain are relatively constant for modest values of v_{in}.

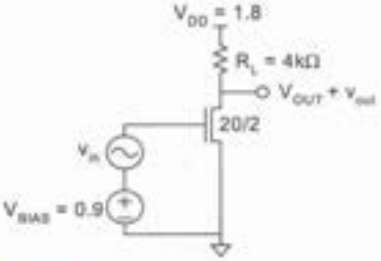


FIG 12.46 Common source amplifier

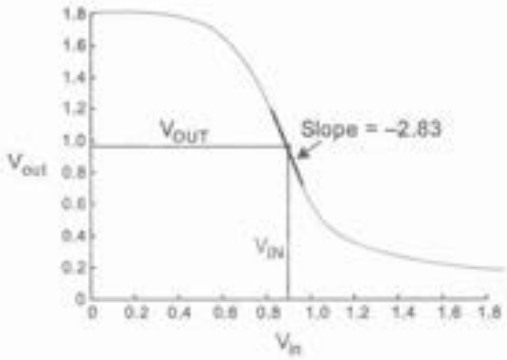


FIG 12.47 DC transfer characteristics of common source amplifier

Example

Calculate the bias point V_{OUT} and small-signal low-frequency gain if $V_t = 0.4$, $\beta = 1550\ \mu A/V^2$ and the nMOS output impedance is infinite.

Solution: At the bias point of $V_{GS} = 0.9$, $I_{DS} = 1550 \cdot 10^{-6} \cdot (0.9 - 0.4)^2/2 = 193\ \mu A^2$ and thus the output voltage is $V_{OUT} = 1.8 - I_{DS}R_L = 1.03$ V. Figure 12.48 shows a small-signal equivalent circuit around this bias point. The model omits the capacitors because they act as open circuits at low frequency and also leaves out r_o because it is infinite. The load resistor ties to a small-signal ground because V_{DD} is constant. According to EQ.(12.13), $g_m = 1550 \cdot 10^{-6} \cdot (0.9 - 0.4) = 0.77$ mA/V. Using Kirchhoff's current law at the output node shows $v_{out} = -g_m R_L v_{in}$, or the gain $A = v_{out}/v_{in} = -g_m R_L = -3.1$.

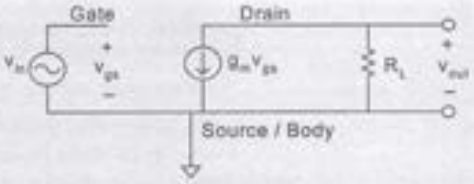


FIG 12.48 Small-signal model of common source amplifier

[2]Note that this differs slightly from Figure 12.44 because channel length modulation is neglected.

Example

Repeat the previous example if the transistor has a channel length modulation coefficient of $\lambda = 0.1\ V^{-1}$.

Solution: The bias point changes only slightly on account of channel length modulation. Assuming from the previous example that $V_{DS} \sim 1.0$, we find $I_{DS} = 1550 \cdot 10^{-6} \cdot (0.9 - 0.4)^2 \cdot (1 + 1.0 \cdot 0.1)/2 = 212\ \mu A$. The output voltage is now $V_{OUT} = 1.8 - I_{DS}R_L = 0.95$, justifying our assumption. The output resistance is $r_o = 1/(0.1 \cdot 212\ \mu A) = 47\ k\Omega$. Figure 12.49 shows a small-signal equivalent circuit around this bias point including r_o. Now the gain depends on the parallel combination of the output and load resistances, $A = -g_m(R_L \,||\, r_o) = -2.83$, where the parallel combination of two resistors is

$$R_1 \,||\, R_2 = \frac{R_1 R_2}{R_1 + R_2} \tag{12.17}$$

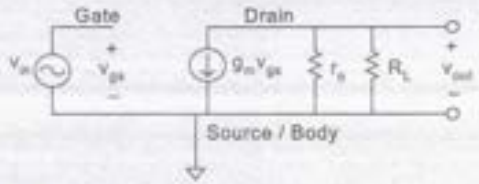


FIG 12.49 Small-signal model of common source amplifier including output resistance

The pull-up resistor can be built as a pMOS transistor of width P operating in its linear region, as shown in Figure 12.50. The gain now depends on the effective resistance of the pMOS transistor. This is simply a pseudo-nMOS inverter. Its DC transfer characteristics were discussed in Section 2.5.4.

12.6.3 The CMOS Inverter as an Amplifier

We can increase the gain of the common source amplifier by using an active load that turns ON when the nMOS transistor turns OFF and OFF when the nMOS turns ON. This can be done by building the load from a pMOS transistor connected to the input, as shown in Figure 12.51. Our high-gain amplifier is simply a CMOS inverter. Hence, the CMOS inverter is an analog amplifier operated under saturating conditions. It can also be viewed as an nMOS common-source amplifier driving a pMOS common-source amplifier. Near the input threshold voltage, the CMOS inverter acts as an inverting linear amplifier with a characteristic of

$$v_{out} = -Av_{in} \tag{12.18}$$

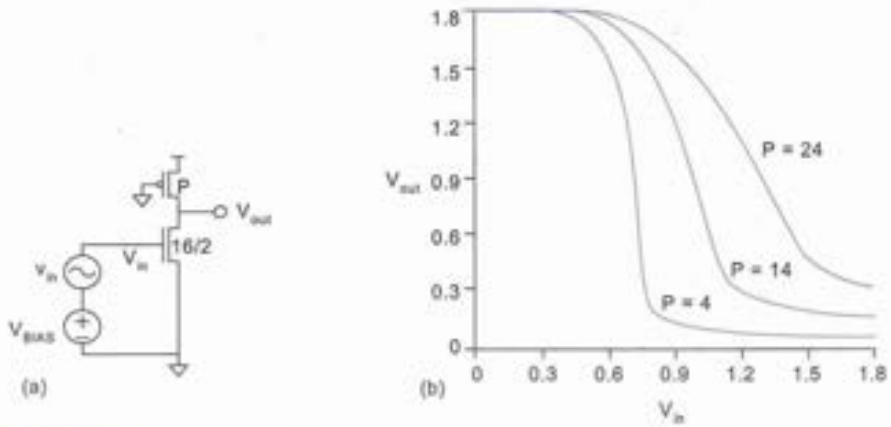


FIG 12.50 Pseudo-nMOS inverter viewed as common source amplifier with pMOS load

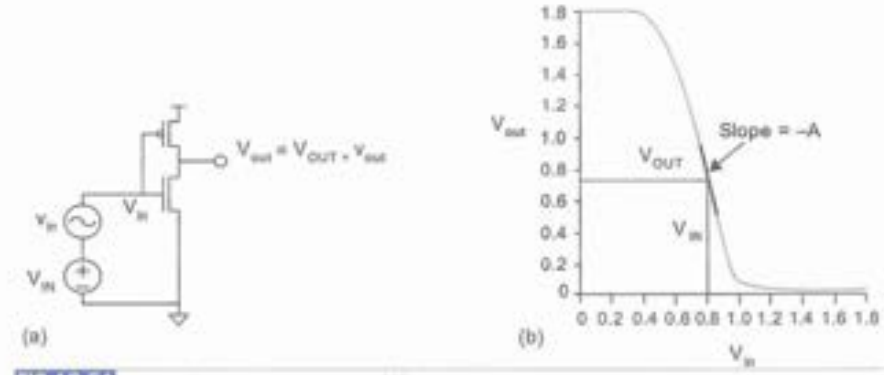


FIG 12.51 CMOS inverter as an amplifier

where A is the amplifier gain.

We can further examine this region with a circuit simulator by using the circuit shown in Figure 12.52 with a high-value resistor (10 MΩ) between input and output to *DC bias* the inverter at V_{inv}. The input is *AC coupled* using a capacitor. The gain of this amplifier is estimated using the small-signal transistor model from Figure 12.45 to construct an equivalent circuit (Figure 12.53) valid for small-signal swings around the linear operating point of the amplifier. The gain is approximately given by

FIG 12.52 AC-coupled CMOS inverter

$$
\begin{aligned}
A &= g_{m-\text{total}} r_{s-\text{effective}} \\
&= \left(g_{mn} + g_{mp}\right)\left(r_{on} \mid r_{op} \mid 10\text{M}\Omega\right) \qquad \textbf{(12.19)} \\
&\approx g_m r_s \ (\text{if } g_{mn} = g_{mp} \text{ and } r_{on} = r_{op} << 10\text{M}\Omega)
\end{aligned}
$$

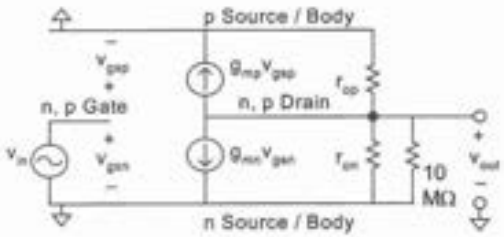


FIG 12.53 Small-signal model of amplifier

The gain depends on the channel length modulation. We can estimate it as

$$g_m = \beta\left(V_{gs} - V_t\right)$$
$$r_o = \frac{1}{\lambda I_{ds}} = \frac{1}{\lambda\left(\frac{\beta}{2}\right)\left(V_{gs} - V_t\right)^2} \tag{12.20}$$
$$A = g_m r_o\Big|_{V_{gs} = \frac{V_{DD}}{2}} = \frac{4}{\lambda\left(V_{DD} - 2V_t\right)}$$

FIG 12.54 Current mirrors

12.6.4 Current Mirrors

The *current mirrors* in Figure 12.54 replicate the input current at the output. The voltage V adjusts itself to the correct level to sink or source I_{in} through $N1$. This voltage also controls the gate of $N2$. If both transistors operate in saturation where I_{ds} depends only on the gate voltage, not the drain voltage, then $I_{out} = I_{in}$. Such an ideal current source has infinite output impedance because the current is independent of the output voltage.

Real devices suffer from channel length modulation and have a finite output resistance. This output resistance makes I_{out} vary somewhat with the drain voltage on $N2$. Good current mirrors use long-channel transistors to achieve high output resistance and nearly constant I_{out}. The output impedance at the operating point is:

$$R_{out} = \frac{v_{out}}{i_{out}} \tag{12.21}$$

This can be found by applying a test voltage v_{out} to the small signal model and computing the current i_{out}. Figure 12.55 shows a small-signal model of the current mirror with the test source. Observe that the small-signal gate voltage v is pulled to ground through the parallel combination of g_{m1} and r_{o1}, indicating that the gate voltage will not stray from the bias point. Applying Kirchhoff's current law (KCL) on the drain of $N2$ shows

$$i_{out} - \frac{v_{out}}{r_{o2}} = 0 \tag{12.22}$$

so the output impedance is $R_{out} = r_{o2}$.

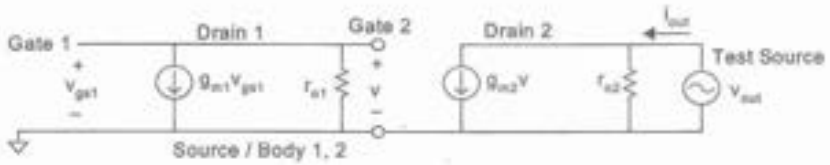


FIG 12.55 Small-signal model for current mirror output impedance

Example

Figure 12.56(a) shows a simple current source constructed from a resistor and a current mirror. It can be modeled as an ideal current source in parallel with a finite output resistance, as shown in Figure 12.56(b). Find I_{out}, the current source output impedance R_{out}, and the range of V_{out} over which the current source operates correctly. Again assume $V_t = 0.4$, $\beta = 1550\ \mu A/V^2$, and $\lambda = 0.1\ V^{-1}$.

Solution: The effect of channel length modulation on bias point is small, so we will neglect it in our analytical solution. We find the bias point by solving the nonlinear equation for input current $I_1 = 103\ \mu A$ at $V_1 = 0.765$.

$$I_1 = \frac{\beta}{2}(V_1 - V_t)^2 = \frac{\beta}{2}(1.8 - I_1 R_1 - V_t)^2 \tag{12.23}$$

At the bias point, $I_{out} = I_1$ and $R_{out} = r_o = 1/(0.1 \cdot I_1) = 97\ k\Omega$. The current source operates correctly as long as the output transistor remains in saturation. This is true as long as $V_{out} > V_1 - V_t = 0.365$ V. A good current source should have high—ideally infinite—output impedance.

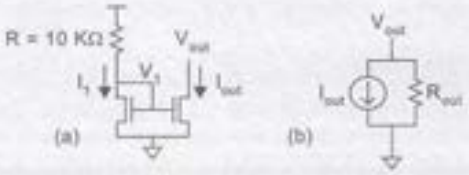


FIG 12.56 Simple current source

The output impedance can be raised by using a *cascoded* current mirror shown in Figure 12.57. In this design, V_1 and V_2 adjust to whatever they must be to sink I_{in} through $N1$ and $N3$. As $I_{out} \approx I_{in}$, the gate-to-source voltages for $N3$ and $N4$ must be nearly equal. Hence $V_3 \approx V_1$. Thus, $N1$ and $N2$ have nearly the same drain voltage as well as gate voltage, making their currents nearly equal even if the transistors have significant output conductance. By KCL, $N4$ must have the same current as $N2$, so I_{out} is very weakly sensitive to V_{out}. In other words, the cascoded current mirror has high output impedance.

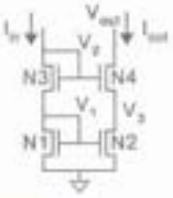


FIG 12.57 Cascoded current mirror

Figure 12.58 shows a small-signal model for the cascoded current mirror[3]. The output impedance increases to (see Exercise 12.5)

$$R_{out} = r_{o2} r_{o4}\left[\frac{1}{r_{o2}} + \frac{1}{r_{o4}} + g_{m4}\right] = (g_{m4} r_{o2}) r_{o4} \tag{12.24}$$

[3]Note that we ignore the body effect in this and subsequent examples. The body effect leads to another transconductance element g_{mb} between drain and source dependent on V_{bs}.

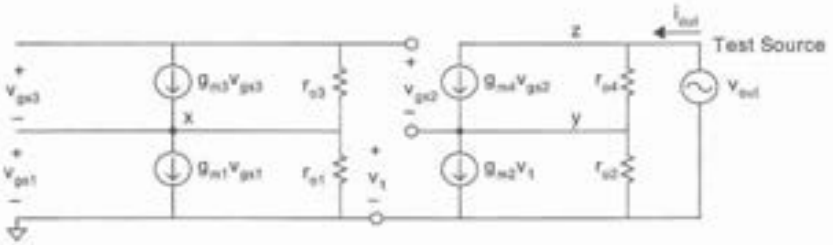


FIG 12.58 Small-signal model for cascode output impedance

Example

Find the output impedance of a cascoded current mirror using the same parameters and bias current as in the previous example.

Solution: Each transistor operates at the same current $I_{out} = 103\ \mu A$ and has the same output resistance $r_o = 97\ k\Omega$. At this current, $g_m = 1550 \cdot 10^{-6} \cdot (0.765 - 0.4) = 0.57$ mA/V. Thus, the cascoded current mirror has an output impedance of $R_{out} = (0.57 \cdot 10^{-3})(97 \cdot 10^{3})^2 = 5.3\ M\Omega$, more than 50 times that of an ordinary current mirror.

Current mirrors can use multiple output transistors to create multiple copies of an input current. The mirror can also multiply a current by N by using an output transistor N times as wide as the input, as shown in Figure 12.59. Better yet, the output can be driven by N identical transistors in parallel because identical transistors match more closely.

FIG 12.59 Current mirrors with multiple outputs and with current gain

12.6.5 Differential Pairs

A differential pair steers current to two outputs. In Figure 12.60, the current I_{ref} is divided between the two outputs depending on the difference between the two input voltages. If the input voltages are equal, the output currents are equal. If one input is substantially higher than the other, it draws all of the current. *Common mode* noise that affects both inputs equally causes no change in the output current. Hence, differential pairs are widely

used because they are insensitive to many noise sources. For this reason, differential pairs are used in sense amplifiers on RAM bitline circuitry.

Differential pairs are easiest to analyze by finding the input voltage difference from the output current difference rather than vice versa. In analog circuits, the transistors normally operate in saturation. We define $V_{gs} = V_g - V_t$ as the *gate overdrive* of a transistor. An ideal transistor would deliver saturation current proportional to the square of the gate overdrive. According to the α-power law model, we can estimate the current of a real transistor with velocity saturation as

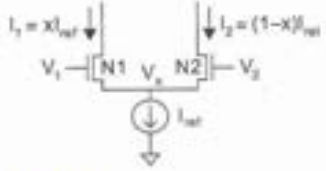


FIG 12.60 Differential pair

$$I_{ds} = kV_{gs}^{\alpha} \tag{12.25}$$

for some $1 < \alpha < 2$.

When $V_1 = V_2$, the amplifier is in its balanced condition and $I_1 = I_2 = I_{ref}/2$. Both transistors have some gate overdrive V_{gs}. We use this to express k in terms of I_{ref} and V_{gs}, so we will be able to eliminate it from subsequent equations.

$$\frac{I_{ref}}{2} = kV_{gs}^{\alpha} \Rightarrow k = \frac{I_{ref}}{2V_{gs}^{\alpha}} \tag{12.26}$$

We would like to know how much differential voltage ΔV is required between V_1 and V_2 so that $N1$ draws xI_{ref} and $N2$ draws $(1-x)I_{ref}$. If we increase the gate overdrive of $N1$ by some fraction δ_1, its current becomes

$$xI_{ref} = k\left[V_{gs}(1+\delta_1)\right]^{\alpha} \tag{12.27}$$

Substituting EQ.(12.26), we can solve for the δ_1 required to obtain x

$$\begin{aligned} xI_{ref} &= \frac{I_{ref}}{2V_{gs}^{\alpha}}\left[V_{gs}(1+\delta_1)\right]^{\alpha} \\ \delta_1 &= (2x)^{\frac{1}{\alpha}} - 1 \end{aligned} \tag{12.28}$$

Similarly, the gate overdrive of $N2$ must go down by some fraction δ_2.

$$\delta_2 = 1 - \left(2(1-x)\right)^{\frac{1}{\alpha}} \tag{12.29}$$

Combining EQ.(12.28) and (12.29) shows that the fractional currents will be x and $1-x$ when a differential voltage ΔV is applied.

$$\Delta V = (\delta_1 + \delta_2)V_{gs} = \left[(2x)^{\frac{1}{\alpha}} - \left(2(1-x)\right)^{\frac{1}{\alpha}}\right]V_{gs} \tag{12.30}$$

This relationship is plotted in Figure 12.61 for several values of α. Values of α closer to two give higher gain. In any case, if one voltage is up and the other down by an overdrive, the second transistor will be completely OFF. Hence, a voltage differential of $2V_{gs}$ is always enough for one transistor to hog all the current.

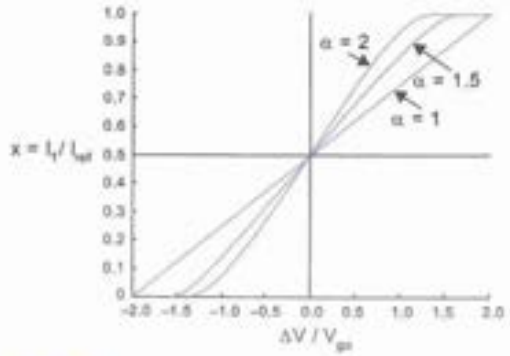


FIG 12.61 Differential pair transfer characteristics

Taking the derivative of EQ (12.30) around $x = 1/2$ and manipulating gives the small-signal transconductance of the differential pair (see Exercise 12.11).

$$\left.\frac{\partial I_1}{\partial \Delta V}\right|_{\Delta V=0} = \frac{\alpha}{4}\frac{I_{ref}}{V_{gs}} = \frac{g_m}{2} \tag{12.31}$$

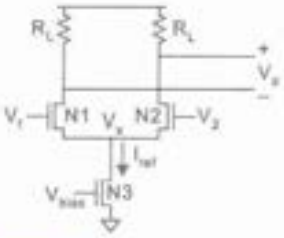


FIG 12.62 Differential amplifier

Figure 12.62 shows a differential amplifier. The amplifier consists of a differential pair ($N1$ and $N2$) driving a resistive load. The differential pair uses a transistor $N3$ with an adjustable bias voltage as the current source; this voltage could be set by a current mirror elsewhere. The high impedance load is often built from a pMOS current mirror because it is much more compact than large passive resistors.

The output impedance is $R_{out} = (R_L \,||\, r_o)$. Because the output is taken differentially, the voltage gain is twice that of each half.

$$A = \frac{V_o}{\Delta V} = 2\frac{\partial I_1}{\partial \Delta V}R_{out} = g_m\left(R_L \,||\, r_o\right) \tag{12.32}$$

An ideal differential amplifier is sensitive only to the voltage difference between the two inputs, not the average (*common mode*) input voltage. A real amplifier works well only while all the transistors remain in saturation. The gain drops off when one of the transistors enters the linear region. This means the headroom available is diminishing as V_{DD} shrinks.

12.6.6 Simple CMOS Operational Amplifier

Figure 12.63 shows a simple CMOS operational amplifier. It consists of a differential amplifier followed by a common source amplifier to achieve high gain. The differential amplifier uses a pMOS current mirror as a load to get high impedance r_{op2} in a compact area. The output is taken single-ended to a pMOS common source amplifier *P3* loaded by nMOS current mirror *N5*. *N3* and *R* set the bias voltage and current for the op-amp.

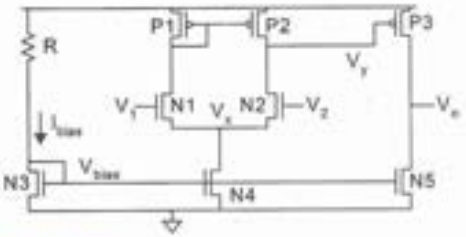


FIG 12.63 Two-stage CMOS operational amplifier

The small-signal gain of the differential stage is computed from EQ (12.32).

$$A_{\text{diff}} = \frac{v_y}{v_1 - v_2} = g_{mn2}\left(r_{on2} \,||\, r_{op2}\right) \tag{12.33}$$

The small-signal gain of the common source amplifier is:

$$A_{\text{commonsource}} = \frac{v_o}{v_y} = -g_{mp3}\left(r_{op3} \,||\, r_{on5}\right) \tag{12.34}$$

Hence, the overall gain of the amplifier is:

$$A = \frac{v_o}{v_2 - v_1} = g_{mn2} g_{mp3}\left(r_{on2} \,||\, r_{op2}\right)\left(r_{op3} \,||\, r_{on5}\right) \tag{12.35}$$

This operational amplifier works well as a comparator. It senses a small difference between the two inputs and drives V_o high or low depending on which input is higher. It is only suited to driving capacitive loads; it does not deliver enough current to drive resistive loads well. It also has a limited common-mode input range to keep all of the transistors in saturation. If the circuit is used in a feedback application, a compensation capacitor may need to be connected between V_y and V_o to ensure stability. [Baker98] and [Gray01] describe op-amp design and frequency response in much more detail.

12.6.7 Digital-to-analog and Analog-to-digital Converter Basics

Digital-to-analog converters (DACs) are relatively easy to design if the target resolution and speed are moderate. Speed, linearity, power dissipation, size, and ease of design are of importance when selecting a DAC architecture. Analog-to-digital converters (ADCs) are more of a challenge, but converters with low speed and precision can be implemented quite easily. The descriptions given in the following two sections can form the start of investigations if implementation is envisaged.

DACs and ADCs are sampled data systems. As such, some more circuits are normally required apart from the basic converter. Figure 12.64 illustrates a DAC in use in a system.

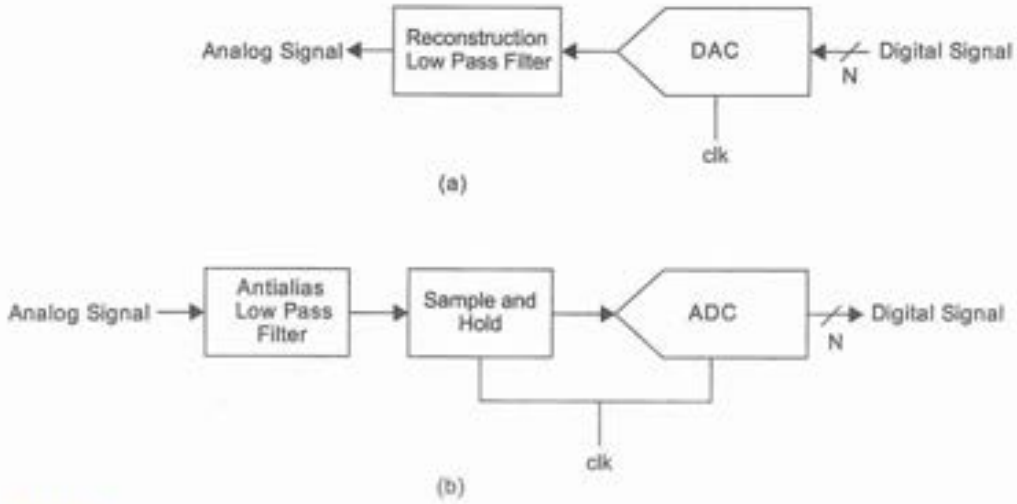


FIG 12.64 DAC and ADC in system

The DAC is followed by a reconstruction filter that converts the quantized DAC output to a smoothed analog value. The complete ADC circuit is similar, but converts from analog to digital. It also has a low-pass filter on the input and usually a sample and hold circuit to hold the input while the ADC does a conversion. The purpose of these blocks will become more apparent as the next two sections unfold.

Before discussing DACs and ADCs in detail, a few DAC metrics will be explained [Hoeschele94]. The parameters are also applicable to ADCs, except rather than a specification on a digital-to-analog transformation, the ADC parameters refer to the deviation from an ideal quantized analog-to-digital transformation.

12.6.7.1 Resolution and Full-scale Range The first parameter of interest in a DAC (or ADC) is the *resolution*. This specifies how many individual quantized steps the DAC possesses. For instance, an N-bit DAC has 2^N individual steps. Thus, an 8-bit DAC/ADC has 256 steps. The *full-scale range* (FSR) is the maximum output voltage (or current) of the DAC/ADC. The resolution (R) of the DAC/ADC is given by $R = FSR/2^N$. So for a 1 V FSR 8-bit DAC/ADC, the resolution is roughly 4 mV while a 10-bit DAC/ADC would have a resolution of approximately 1 mV. The FSR is related to the voltage supply. In the case of a DAC or ADC, the FSR can not be greater than V_{DD} and common implementations have FSRs of the order of V_{DD} minus one or two V_t drops. As processes scale and V_{DD} decreases, analog performance becomes more difficult to achieve because the resolution for a given size converter decreases in relation to any noise that might be present.

12.6.7.2 Linearity A primary concern with a DAC/ADC is the linearity or accuracy of transformation from a digital code to a quantized analog value. Linearity of a DAC/ADC

is determined by component linearity, where the components in CMOS are transistors, resistors, and capacitors. It is also affected by the introduction of unwanted signals that are classified as noise. Static (DC) linearity is normally calculated using measures called the *Integral Nonlinearity* (INL) and *Differential Nonlinearity* (DNL).

- DNL: The departure from one LSB in transitioning from one digital code to the next. For instance, in Figure 12.65(a) the DNL highlighted is approximately 0.5 least significant bits (LSB) for the 3-bit DAC shown.
- INL: The maximum deviation from a straight line drawn between the endpoints of the DAC (ADC) output (input) characteristic. Again, the INL illustrated is around 0.5 LSB.
- Offset: The difference between (nominally) zero and the actual value when the digital code for zero is applied. This is a fraction of an LSB in the figure.

A good DAC is monotonic; i.e., for each increase (or decrease) in digital code input the DAC, the analog output increases (or decreases) in value. Correspondingly, for each increase in analog input, an ADC would expect to see an increase in the digital code. A non-monotonic DAC step is shown in Figure 12.65(b), where the analog output decreases to an increase in the applied digital code. Depending on the application, this may be undesirable.

Typical INL and DNL plots are shown in Figure 12.66. These are usually measured at DC. In the plots, the INL is +0.2/−0.25 LSBs and the DNL is ±0.11 LSBs.

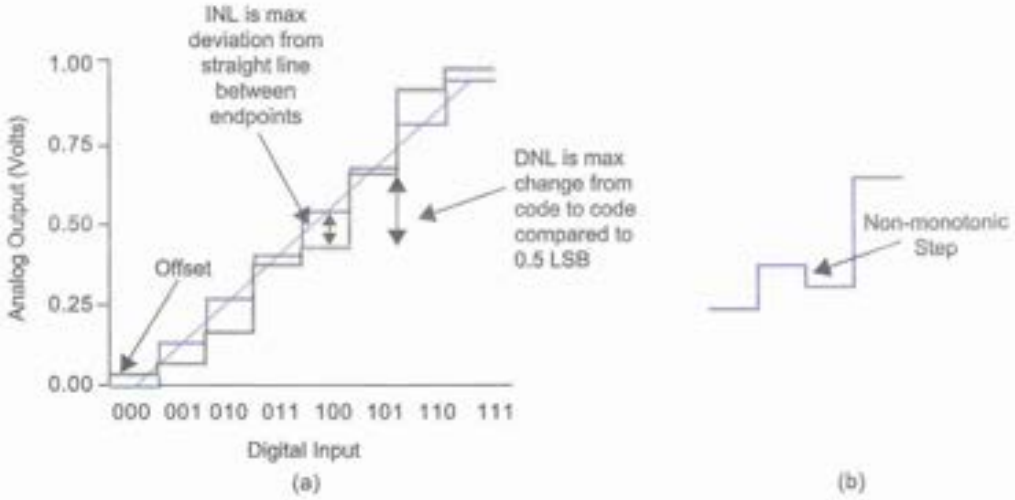


FIG 12.65 DAC linearity measures

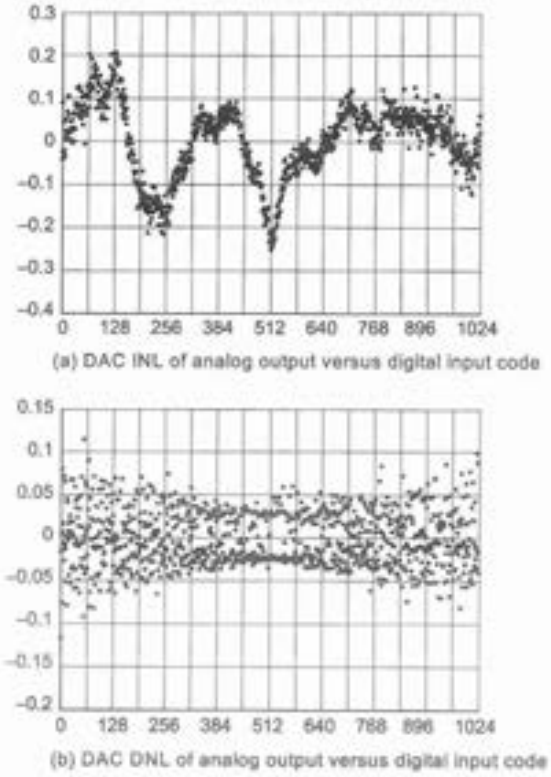


FIG 12.66 Typical DAC INL and DNL plots

12.6.7.3 Noise and Distortion Measures With a (digital) sine wave applied to the DAC, the ratio of the desired signal energy to harmonics and noise is called the *Signal-to-Noise Ratio* (SNR). In practice, to measure this in a DAC, the DAC is fed a digital sine wave from a numerically controlled oscillator. The DAC is low-pass filtered, as shown in Figure 12.64, and then fed to a spectrum analyzer. The undesired signals are then apparent.

In comparison, an ADC measures this by applying a high-quality sine wave to the ADC and doing an FFT on the stored digital samples for a number of cycles of the input.

In addition, dynamic measures such as the *Total Harmonic Distortion* (THD) and *Spurious Free Dynamic Range* (SFDR) are also of interest. The THD measures all unwanted harmonic content and expresses this as a ratio of desired-to-undesired outputs. For instance, in a DAC, for a desired frequency of f_{signal}, there may be significant second harmonic ($2f_{signal}$) and third harmonic ($3f_{signal}$) noise caused by device non-linearity. These unwanted harmonics distort the required signal and degrade the THD.

SNR and THD are classical measures of analog linearity, which originally were applied to totally analog systems. A more popular measure with DACs and ADCs that takes into consideration digital artifacts is the *spurious free dynamic range* (SFDR). The SFDR is a measure over a specified frequency range of the desired output versus unwanted signals. These include noise from sources such as switches and noise injected from other sources on chip. SNR, THD, and SFDR are measured in dB (decibels).

The *Intermodulation Distortion* (IMD) is measured by outputting two simultaneous sine waves and measuring the amplitude of spurious products. This is a measure of the extent of unwanted multiplication products that may not show up with a simple single frequency (tone) test. ADCs measure this by applying two analog sine waves to the ADC and analyzing the digital output.

Figure 12.67 shows a typical DAC spectrum. The harmonics and SFDR are shown. The "fuzz" is typical of sampled data systems.

The noise and distortion values can be transformed into a single value called the *Effective Number of Bits* (ENOB), which is defined by:

$$\text{ENOB} = (\text{SNR} - 1.76) / 6.02$$

This rolls all performance numbers into a single value, which can be used to compare DAC implementations. The effective SNR can also be evaluated from an ENOB number.

$$\text{SNR} = \text{ENOB} \cdot 6.02 + 1.76 \text{ dB}$$

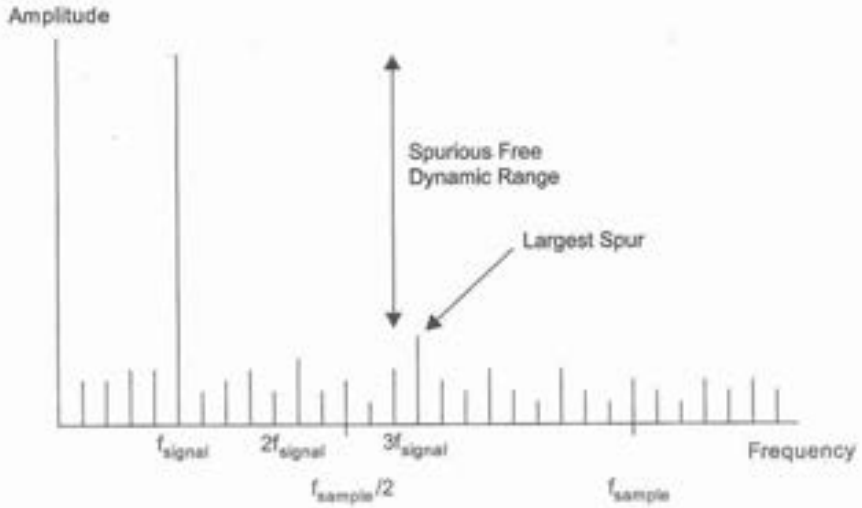


FIG 12.67 Typical DAC frequency plot showing harmonics and noise

Example

An 8-bit converter has a maximum SNR of 49.92 dB. If an implementation has an ENOB value of 7.1 bits, what is the SNR (in dB)?

Solution: SNR = 7.1 • 6.02 + 1.76 = 44.5 dB

Example

If the required SFDR is 47 dB, how many ENOBs are required?

Solution: ENOB = (47 − 1.76)/6.02 = 7.51 bits

As mentioned previously, DACs and ADCs are sampled data systems. Two major artifacts are present in DACs and ADCs. The first effect is due to sampling theory. This results in the replication of copies of the baseband signals at multiples of the clock frequency, as is shown in Figure 12.68(a). This requires that a low-pass filter follow a DAC to eliminate the unwanted signals. This is commonly called a *reconstruction filter*. In the figure, the Nyquist frequency ($f_{sample}/2$) is noted. This is the maximum signal frequency that can be generated by a DAC clocked at f_{sample}.

The second artifact results from the frequency response of a pulse, which is a $\sin(x)/x$ function as shown in Figure 12.68(b). This results in a roll-off with frequency of the

desired (and undesired) signals. This is also illustrated in Figure 12.68(a). This may have to be compensated for by the digital circuits driving the DAC if a flat response is required. Similarly, an ADC responds to signals at multiples of the sampling frequency. Thus, an *antialiasing filter* has to be provided so that the ADC does not see these signals. In addition, compensation may be needed for the $\sin(x)/x$ response.

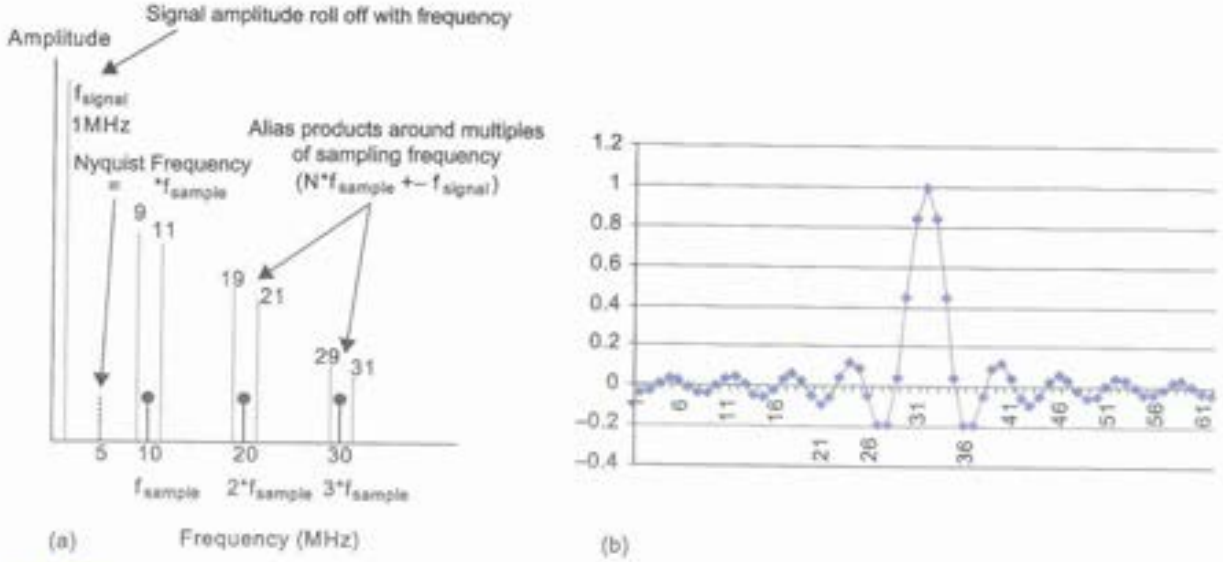


FIG 12.68 Frequency response of a DAC

Example

For a sample frequency of 10 Mhz, what are the alias products for a 1 MHz signal?

Solution: Products appear at the signal frequency offset from multiples of the sample frequency. So the first is at $f_{sample} - f_{signal} = 9$ MHz and the next at $f_{sample} + f_{signal} = 11$ MHz. The next two are offset from the second harmonic, i.e., 19 MHz and 21 Mhz, and so on. These are illustrated in Figure 12.68(a).

12.6.8 Digital-to-analog Converters

This section will present a selection of DACs roughly in order of increasing implementation complexity. All DACs presented here are within the capability of a careful CMOS circuit designer. DACs are simpler than ADCs.

A DAC can be implemented in software if a processor or DSP is available on the chip already, as shown in Figure 12.69. The processor provides a stream of 1's and 0's, which, when integrated (low-pass filtered), provides the required analog output. The RC time

constant of the low-pass filter is designed to cut out the harmonics present in the output at multiples of the sampling frequency. Typically, a simple DAC such as this is highly *oversampled*; i.e., the clock frequency of the serial bit stream is many times the eventual analog output frequency. For the simple first-order filter shown, this oversampling ratio might be in the range of 256–1024.

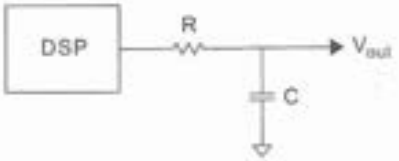


FIG 12.69 A DSP or processor-controlled DAC

If no processor is available, a converter called the *Pulse Width Modulated* (PWM) DAC, shown in Figure 12.70, can be implemented. This DAC employs a digital counter and a comparator. The counter cycles through a set of 2^N values and the comparator is set high when the count is less than the digital input B. This results in a waveform with a varying duty cycle. If B is small, the waveform will spend the majority of the time low. Conversely, if B is large, the output is high for most of the counter cycle. The linearity of this and the previous converter can be quite high, as it is limited by the linearity of the on-chip R and C in the filter, which can be reasonably good.

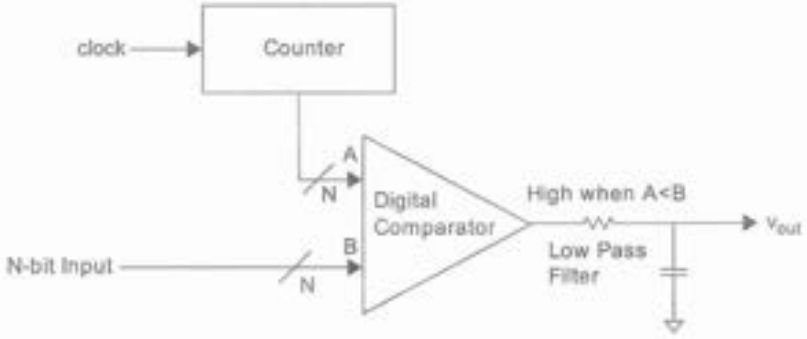


FIG 12.70 A pulse width modulated DAC

The *resistor string DAC* is perhaps the most straightforward Nyquist-rate DAC to design. A Nyquist-rate DAC is one in which the analog output values change at the DAC update rate and the usable analog frequencies extend to half the Nyquist (clock) rate. Two versions of this DAC are shown in Figure 12.71. A reference *voltage ladder*, consisting of 2^N resistors for an N-bit DAC, is connected from the supply to ground. A 2-bit converter using four resistors is shown in Figure 12.71(a). CMOS switches are used to switch the appropriate reference voltage to the output. A decoder switch can be used as shown, in which case the switch depth is N. Alternatively, each switch can be individually decoded as shown in Figure 12.71(b), in which case, the switch depth is one. While simple, this DAC is slower than other designs due to the RC time constant through the ladder and switches. In addition, the load resistance has to be high compared to the resistor string. Typically, it is useful as a reference DAC driving a CMOS buffer, op-amp, or comparator. It is suitable for moderate speed applications. Resistors can be constructed using polysilicon. The DAC is inherently monotonic due to the resistor string.

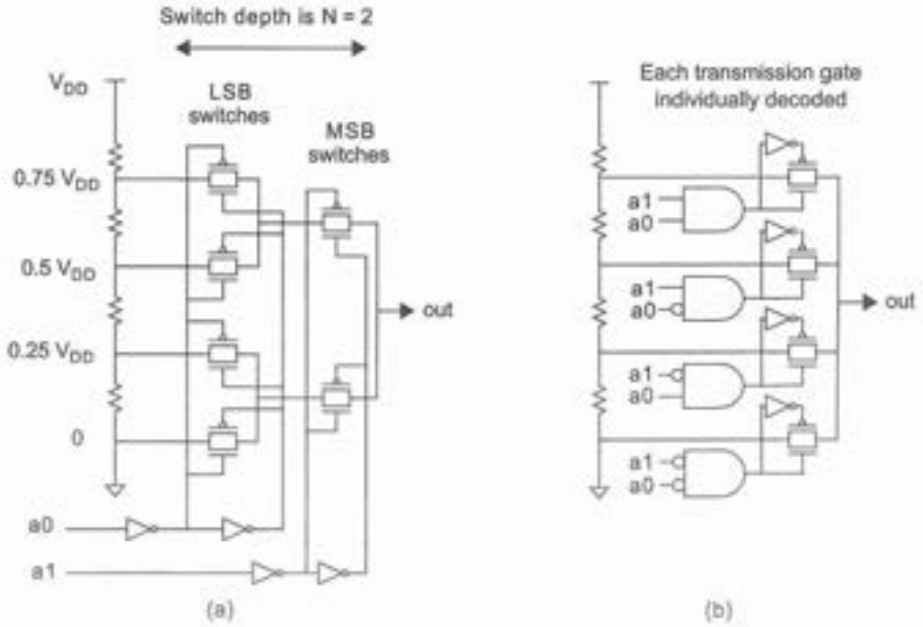


FIG 12.71 Resistor string DACs

The *R-2R DAC* uses an array of resistors of value R and $2R$, as the name suggests. A 4-bit version of this converter is shown in Figure 12.72. The DAC employs resistors, CMOS switches, and a buffer amplifier. The number of resistors required is $2N + 1$. The buffer has to be designed carefully to allow the input to be linearly amplified.

The fastest DACs in CMOS are those built from current sources. The basic principle of a 1-bit current DAC is shown in Figure 12.73(a). A digitally controlled current is switched into a resistor to create a voltage swing proportional to the product of the resistance and current. A simple current source ($P1$) is shown in Figure 12.73(b) along with a differential switch ($P2$, $P3$). The switch toggles the current from one leg to the other of a differential resistor network ($R1$, $R2$) under the control of complementary signals S and SN. The current source $P1$ is set with V_{bias}. The differential voltage ($V_{out} - V_{outn}$) is the DAC output. Figure 12.73(c) shows an improved cascode current source ($P1$, $P2$) with better linearity. Although pMOS devices have been shown to allow a GND-referenced system, nMOS devices can also be used for a V_{DD}-referenced system.

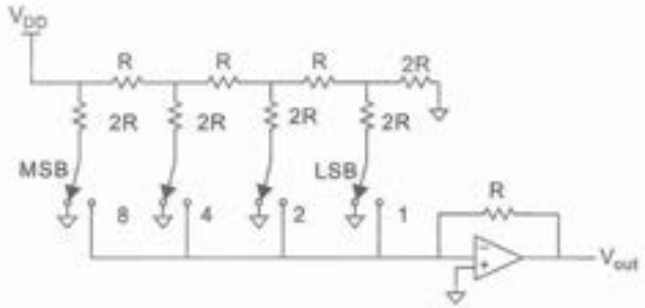


FIG 12.72 R-2R DAC

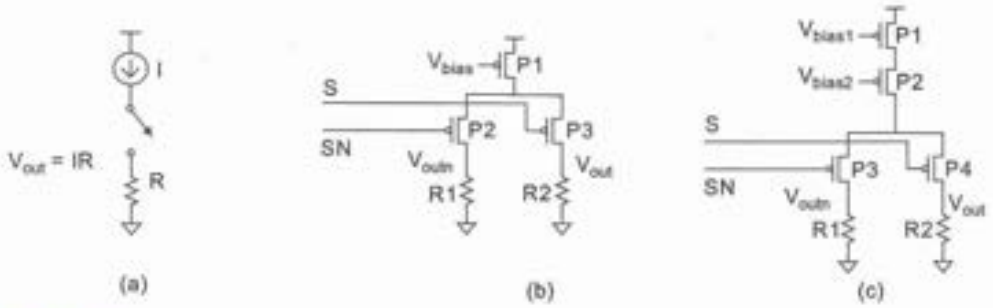


FIG 12.73 A current DAC

There are two basic methods of building a *current DAC*. The first is shown in Figure 12.74(a). Here, *N* weighted current sources are used to build an *N*-bit DAC. Each current source is built by sizing the current source in line with the bit weighting. For a 4-bit converter, a 1X, 2X, 4X, and 8X current source would be required. These could be built from

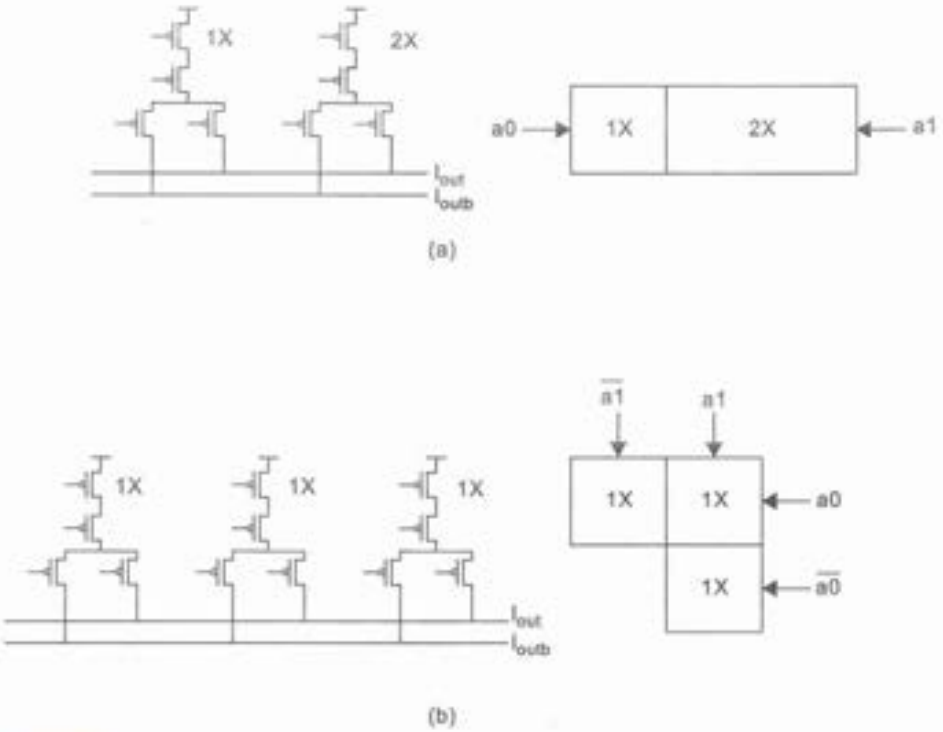


FIG 12.74 Current DAC architectures

1X, 2X, 4X, and 8X the basic current source using current mirroring and appropriately sizing the transistors. This style of current DAC is suitable for small DACs or DACs where speed is paramount. Problems can surface when designers try to match the scaled current sources to each other.

Figure 12.74(b) shows an alternative architecture which uses $2^N - 1$ identical current sources. This is called a *fully segmented DAC*. The linearity can be quite good if the matching of the current sources can be maintained. A 2-bit converter is shown in schematic form and a possible floorplan is shown.

Figure 12.75 shows a full implementation of a 4-bit current DAC. To the basic cascode current source and switch, a decode OAI gate is added in each current cell. An array of fifteen current cells is used. To the left of the current array, a row decoder, and at the bottom a column decoder, have been added. These decode the two LSBs and two MSBs respectively, and drive row and column lines into the array. Depending on which lines are activated, a number of current sources are turned on in proportion to the presented digital value. Each current source is biased by the common V_{bias1} and V_{bias2} voltages. These can be generated using a *replica bias* generator (using a replica of the unit current source) as shown (at the top left) and these in turn can be buffered by operational amplifiers. Some design hints have been included in the figure. Examples of this style of DAC can be found in [Bugeja00, Tiilikainen01].

12.6.9 Analog-to-digital Converters

Like DACs, Analog-to-digital converters (ADCs) are rated by precision, speed of conversion, power dissipation, chip size, and ease of design. ADCs also feature Nyquist converters and oversampled converters. Achieving state of the art ADC performance is an exacting science and the descriptions here about designing an ADC are not presented in great detail. However, moderate ADC design is quite within the capability of the CMOS designer who has an interest in analog circuits. A good strategy is to gain experience by "having a go" in the academic environment by designing and following an ADC design through fabrication and measurement. With the accuracy of simulation and layout extraction tools available for CMOS processes, many ADC (and DAC) architectures can be simulated in extreme detail, with the simulation results being close to fabricated chip results. Thus, experimentation can be completed "virtually."

An ADC is primarily categorized by the speed of conversion and number of bits. In common with DACs, the effective number of bits (ENOBs) tell the real story of a particular ADC. In common with DACs, other measures of interest are the offset, INL, DNL, SNR, and SFDR.

Figure 12.76 shows a *dual slope ADC*, which uses CMOS switches, an integrator and comparator, and digital logic to implement a relatively high-precision converter. In operation, the integrator is first reset by closing SWc. Next, the (negative) input voltage is integrated for a known time by closing switch SWa and opening SWc. SWa is opened and SWb closed with a (positive) reference voltage connected. The time at which the output crosses zero is measured and the input voltage can be calculated from:

$$V_{in}/|V_{ref}| = t_{measure}/t_{integrate}$$

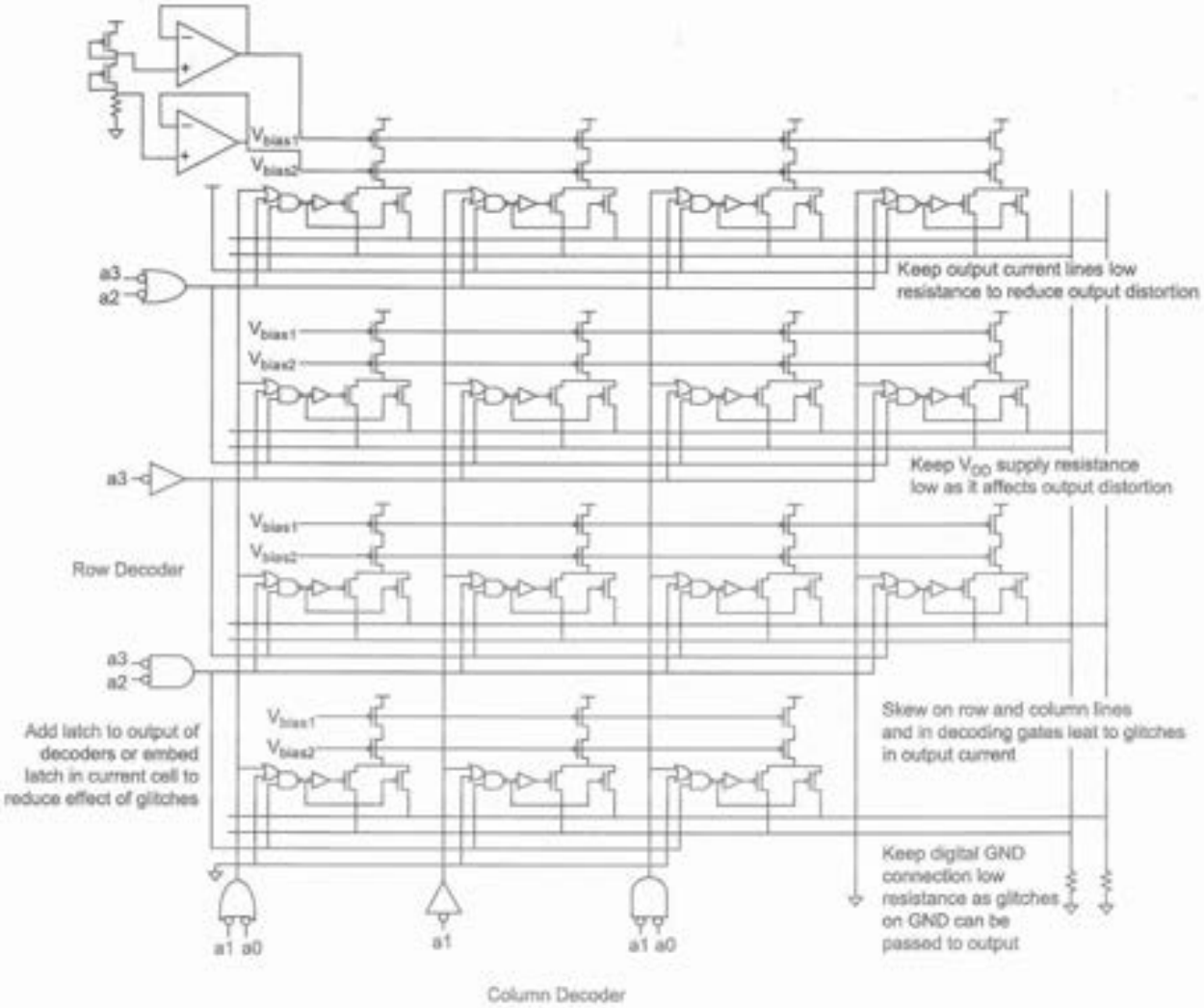


FIG 12.75 A 4-bit current DAC

This converter is relatively slow due to the integration times, but can achieve good precision with simple analog components. CMOS inverters can be used for the amplifiers, as shown later in this section. An example of a dual slope ADC can be found in [Rodgers89].

Figure 12.77 shows an ADC that compares the input to a value set by a DAC. The easiest algorithm for conversion is to ramp the DAC from zero to full scale and observe when the comparator switches. This would take 2^N clock cycles for an N-bit converter. A quicker method is to use a successive approximation algorithm. The following pseudo-

code demonstrates the successive approximation algorithm. This algorithm completes in $N + 1$ cycles and is preferred for this reason (i.e., 9 cycles versus 256 for an 8-bit ADC). The normal input range is zero to $V_{fullscale} \cdot (2^{N-1})/(2^N)$.

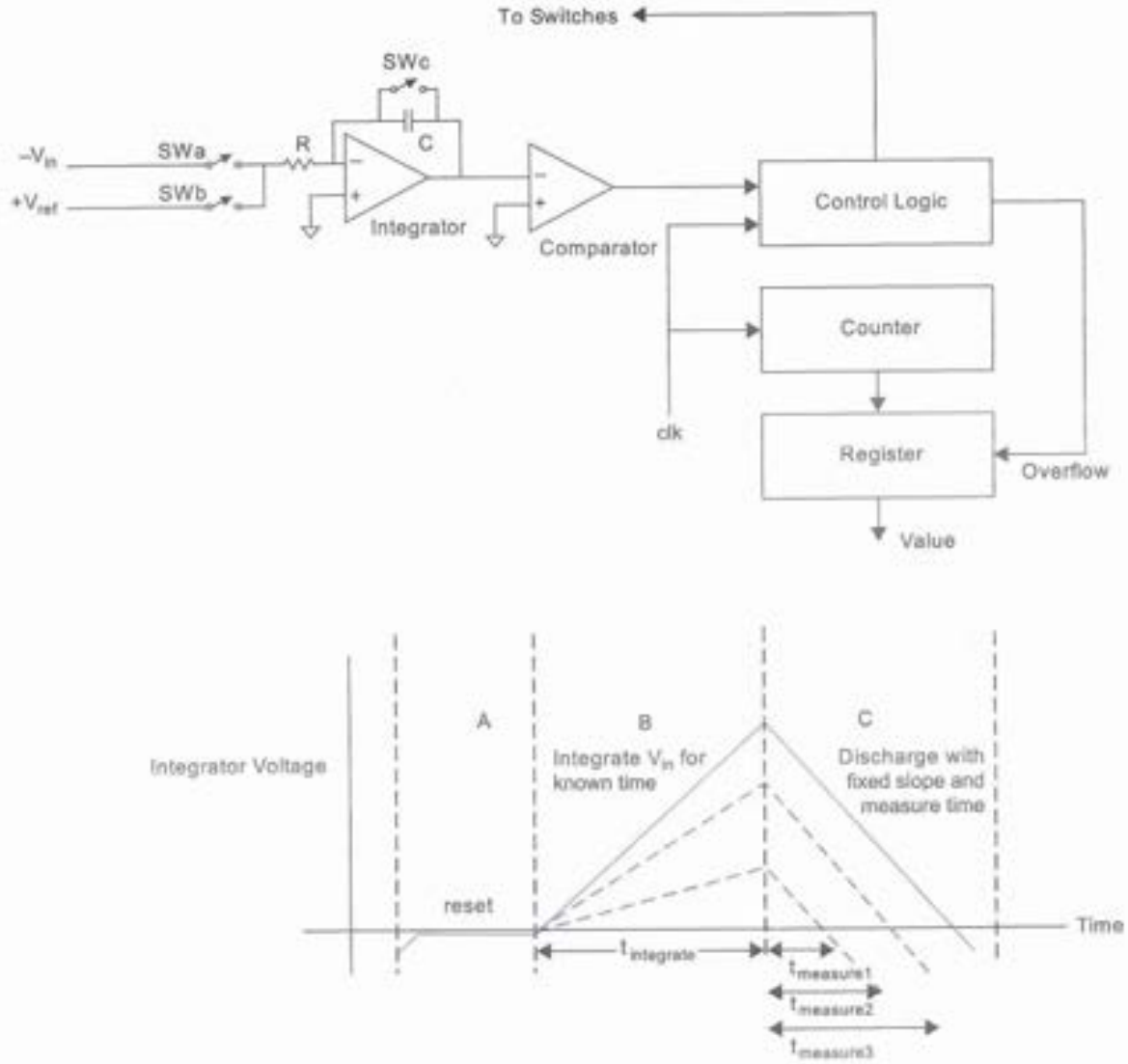


FIG 12.76 Dual slope ADC

```
Initialize:  Vdac = Vfullscale/2
             Vstep = Vfullscale/4
             Vout = 0

   loop repeat N
   write(Vdac) ;;output DAC value
   flag = read(comparator) ;; flag = Vin>Vdac
   if (flag)
      {
         Vdac = Vdac + Vstep ;; DAC value is high so reduce
         Vout = Vout + 1 ;;note contribution to output
      }
      else
      {
         Vdac = Vdac - Vstep ;; DAC is low so increase
      }
      Vout = Vout << 1 ;; multiply output by two
      Vstep = Vstep >> 1 ;; divide step by two
   repeat
```

Typical operation of the *successive approximation ADC* is shown in Figure 12.78 for a 3-bit converter with a 1 V full scale and 0.28 V applied to the input. In time slot 0, V_{dac} is set to half scale and the comparator sampled. As V_{dac} is greater than V_{in}, V_{step} of 0.25 V is subtracted from V_{dac}. In cycle 2, the process is repeated and 0.125 V is added to V_{dac}. Finally, 0.0625 V is subtracted for a final value 0.3125 V. The next cycle successively approximates 0.6875 V for a 0.72 V input. V_{out} lags the input by a conversion cycle of four clock cycles.

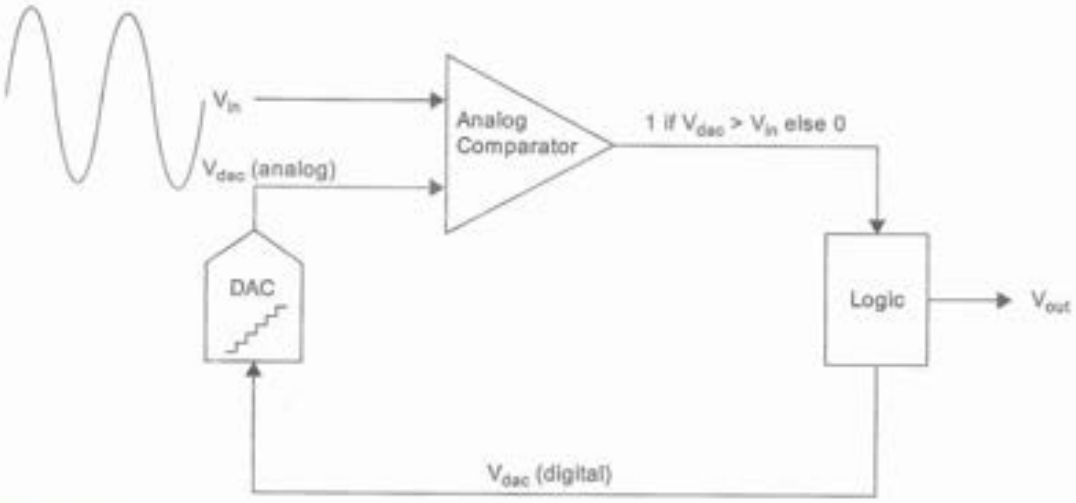


FIG 12.77 Successive approximation ADC

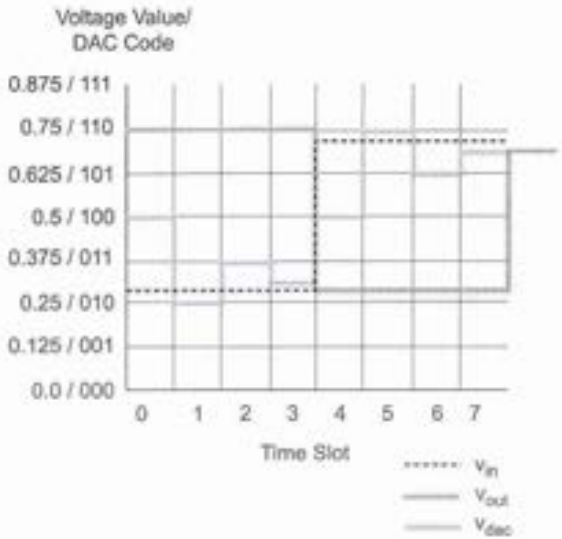


FIG. 12.78 Successive approximation ADC operation

Representative successive approximation ADCs can be found in [Sauerbrey03, Promizter01, Mortezapour00]. One of the original commercial implementations can be found in [Timko80].

The *flash* converter is shown in Figure 12.79. This is the fastest converter architecture. The design consists of 2^N parallel comparators, each of which is fed with the input and a monotonically increasing reference voltage. This is most easily generated by using a string of resistors that is grounded at one end and fed at the other end with a reference voltage. When the analog input rises above the reference value fed to a particular comparator, the comparator turns on. Thus, a thermometer-like sequence of 1's will appear at the output of the comparators, rising and falling in sympathy with the analog input. The whole system is clocked at some frequency determined by the fastest time that the signal can be sampled and the comparator switched. The *thermometer code* is then fed to digital logic to extract a binary number corresponding to the analog input.

While much design effort can go into the comparator and associated sampling circuitry, a simple but still useful cir-

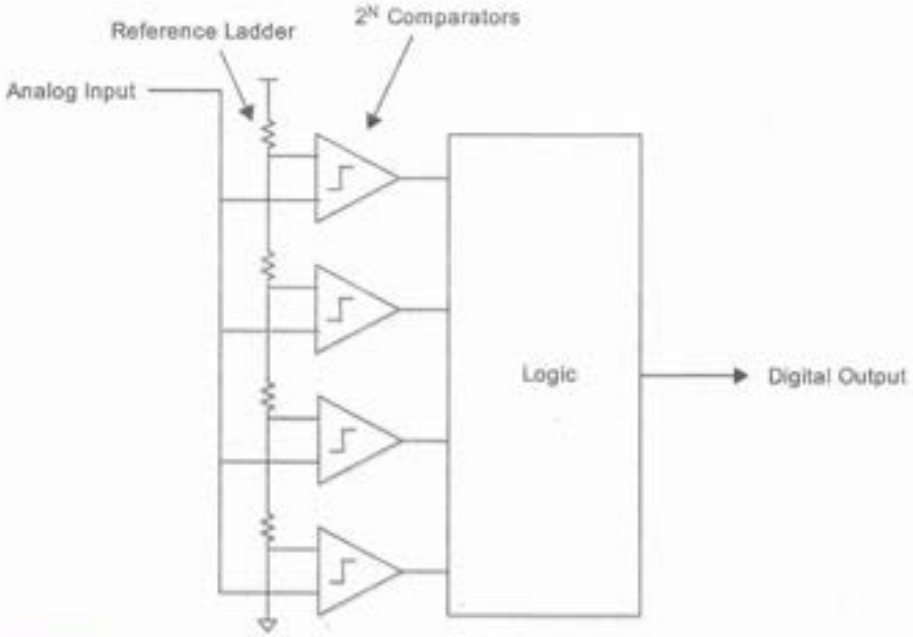


FIG. 12.79 Flash ADC architecture

cuit uses one or two inverters [Dingwall79, Dingwall85]. The basic circuit is shown in Figure 12.80(a). The circuit consists of three CMOS switches, a capacitor, and a CMOS inverter, connected as shown. The operation is explained in Figure 12.80(b). In the reset cycle, the inverter is biased by connecting its input to its output. This settles the output at the inverter threshold voltage, as defined in Section 2.5.1. One end of the sampling capacitor is connected to the input of the inverter. In addition, the analog input is connected to the other end of the capacitor. In this configuration, the capacitor assumes a charge $Q = C \cdot (V_a - V_{cm})$, where V_{cm} is the inverter threshold voltage and V_a is the analog input voltage. In the second phase (Figure 12.80(c)), the reference voltage V_{ref} is connected to the capacitor and the charge is maintained. As the inverter is in high gain mode, the difference in analog input and reference voltage is amplified by the inverter. If the gain is high enough, the inverter drives the output voltage to the rails.

If a single stage does not have enough gain, another capacitively coupled gain stage can be added, as shown in Figure 12.81 [Dingwall85]. This figure shows the complete circuit as well as a latch to store the comparator output.

The design decisions are fairly simple with this design. The capacitor size has to be selected. The size of the capacitor in relation to the input size of the inverter determines the overall gain of a single stage. A value of roughly 10x the input inverter gate capacitance is a start. The inverter also has to be sized. Normally, there is no advantage to making the inverter much larger than minimum-sized because its size determines the sampling capacitor and the overall input capacitance of the converter. However, with sub-micron transistors, it is advantageous to lengthen the gate from minimum to flatten the I-V characteristic and reduce g_{ds}. The nMOS and pMOS transistors should be sized to place V_{cm} in the center of V_{swh} which is commonly V_{DD}. The disadvantages of such a simple circuit compared to differential circuits are that the comparator is sensitive to common mode noise on the input, ground, and V_{DD}. Using V_{DD} as a reference also implies that this supply should be filtered. However, as supply voltages are reduced, the opportunity to build more sophisticated circuits is lessened and careful layout and design can yield good results for this comparator. This is especially true with an insulated SOI substrate or triple-well processes.

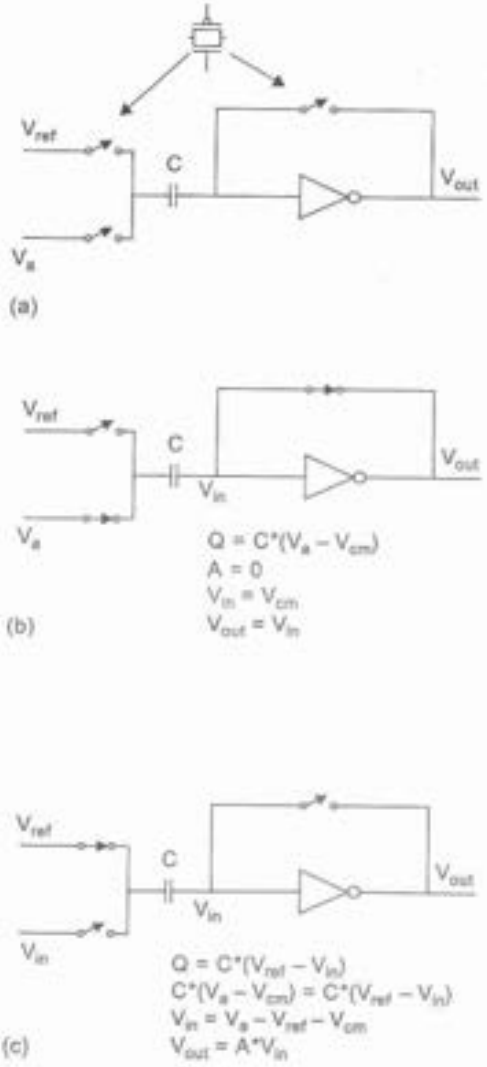


FIG 12.80 Simple CMOS inverter-based comparator

The output of the comparators is a thermometer code and the next stage has to decode this to a binary value. A "bubble gate," as shown in Figure 12.82, is used to determine the highest one in the thermometer code. While a simple 2-input gate can be used, the gate shown detects a sequence of 110 in the output. This prevents a spurious one (and zero bubble) from falsely triggering the decoder.

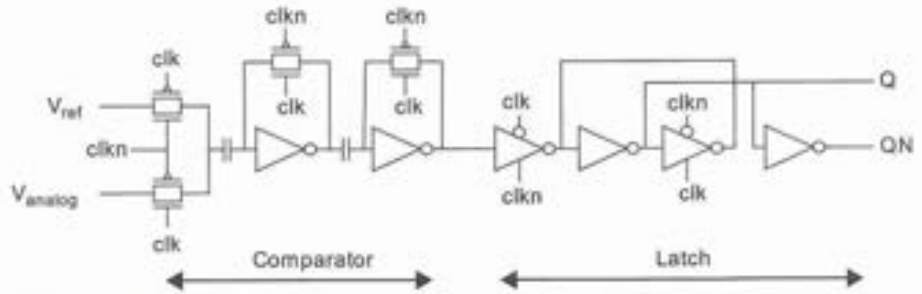


FIG 12.81 Improved gain comparator circuit with output latch

FIG 12.82 Thermometer bubble gate

The output of the bubble gate is then fed to a 1-hot-to-binary encoder. The first example shown in Figure 12.83(a) uses pseudo-nMOS NOR gates to convert to a binary number. The second encoder, shown in Figure 12.83(b), uses multiplexers. Another method is to count the number of 1's in the thermometer code with a cascade of adders.

This completes the design of the flash converter. Although it is the fastest converter, the large number of comparators places a significant load on any circuit driving the ADC. This usually limits the flash converter to fewer than 8 bits. Representative flash converters can be found in [Jiang03, Uyttenhove03, Donovan02, Scholtens02]. Some reduction in the number of comparators can be achieved using interpolation techniques and a technique called *folding*. Representative converters can be found in [Nauta95, Li03].

The *pipeline ADC* essentially trades the high speed and low latency of the flash converter for longer latency and a slightly more complicated (and hence lower speed) design. However, the power dissipation can be much lower than for a flash converter of the same speed. The design is outlined in Figure 12.84(a). The ADC is composed of (ideally) N identical stages for an N-bit converter. Each stage contributes a bit to the overall result. A single stage is shown is Figure 12.84(b). The input V_{in} is presented to an ADC that subtracts or adds a reference voltage (V_{ref}) from the input. The difference is then amplified by

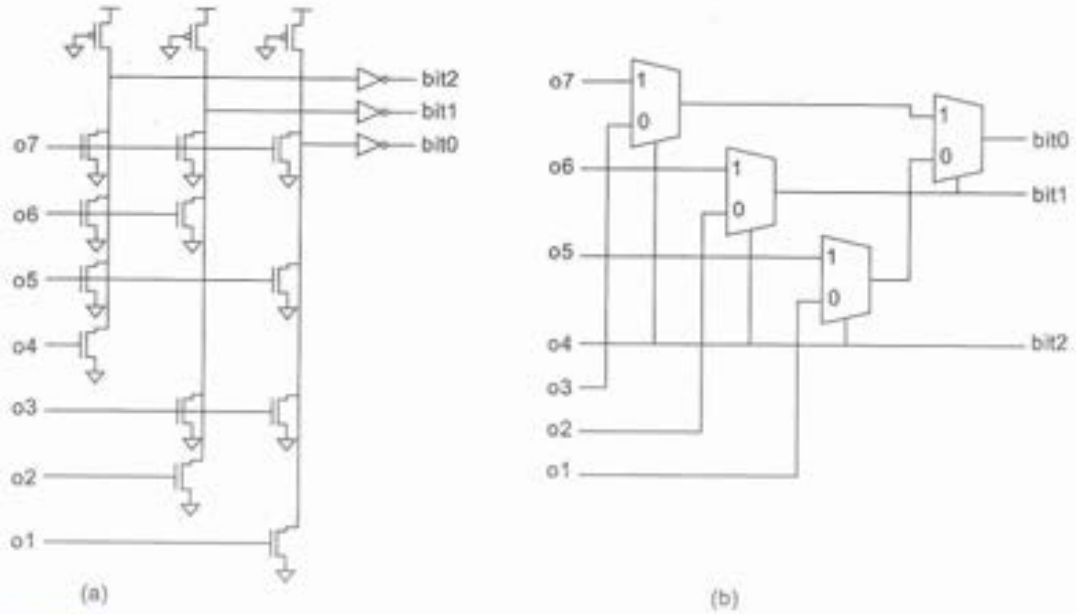


FIG 12.83 Flash ADC encoder

a factor of two and the process repeated. (This amounts to a distributed successive approximation converter.)

While all of these operations may seem difficult to achieve, a typical stage is shown in Figure 12.85 using CMOS switches, capacitors, and a gain stage.

The gain stage in the pipeline converter has to have enough gain to accurately subtract and perform the multiplication (assuming matching capacitors). The *folded cascode* stage shown in Figure 12.86 is a popular gain stage because it combines good gain into a single stage. This illustrates that the circuit complexity can be moderate. However, the design of the amplifier is beyond the scope of this text and the reader is referred to the literature for further details [Gray01].

Representative pipeline converters and descriptions of their operation can be found in [Lin91, Cho95, Jamal02, Chang03, Poulton03].

One of the most ubiquitous ADC architectures in use today is the *sigma-delta* architecture shown in Figure 12.87. This converter, which was developed in the late 1970s, is ideal for processes where digital circuits are easier to implement than analog circuits [Candy76]. The converter works by subtracting a delayed digitized sample of the analog input from itself. This is passed through a filter, $F(s)$, and then sent to a comparator. In the first-order example shown, this is then applied to a 1-bit DAC, which completes the loop. The digital output of the comparator is passed through a digital filter and the digital code for the analog input retained. Sigma-delta converters are oversampled converters. That is, they operate at a multiple of the required signal frequency. Oversampling ratios of

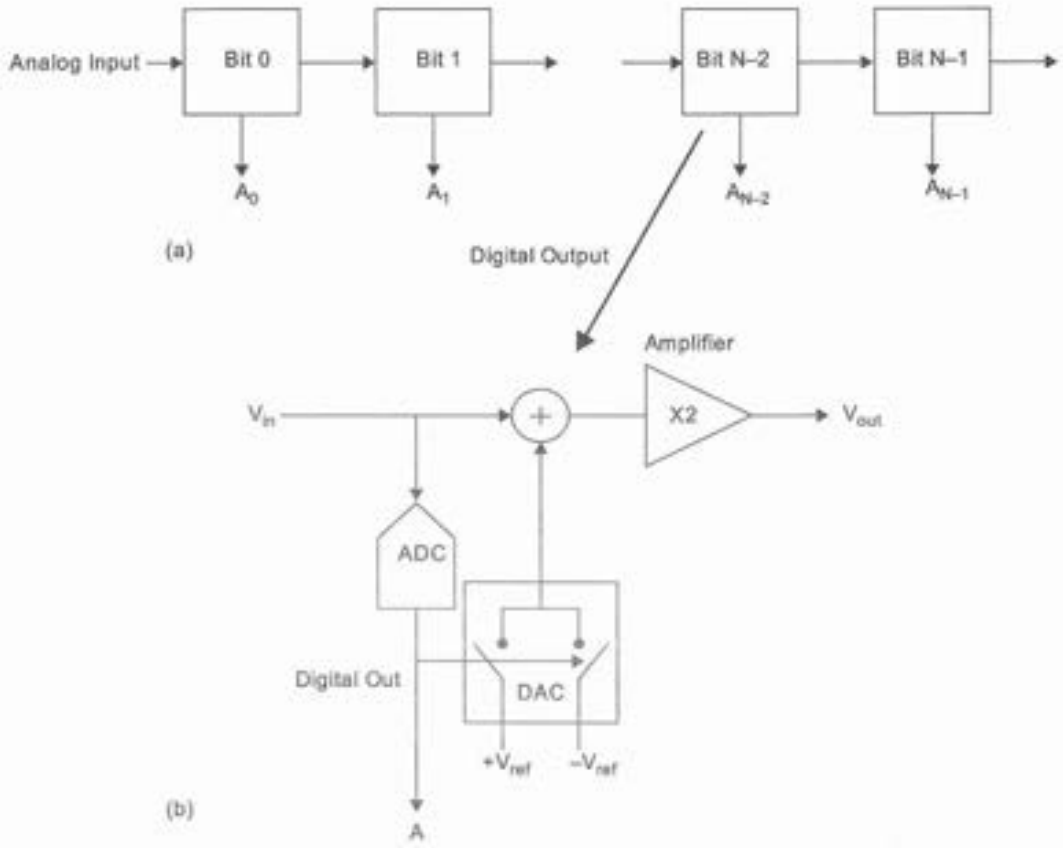


FIG 12.84 Pipeline ADC block diagram

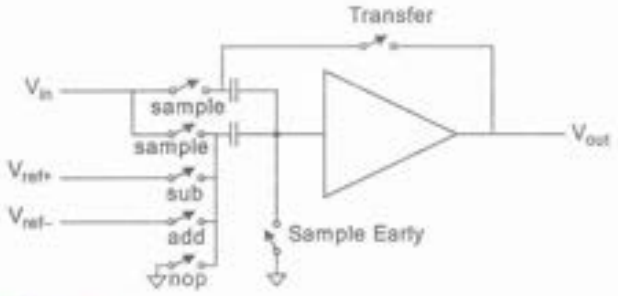


FIG 12.85 Pipeline ADC single stage

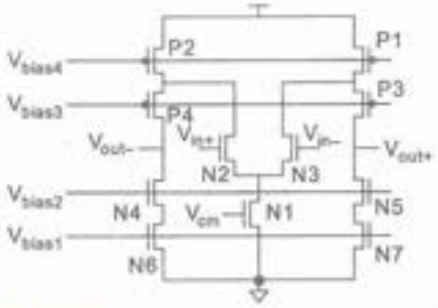


FIG 12.86 CMOS folded cascode op-amp

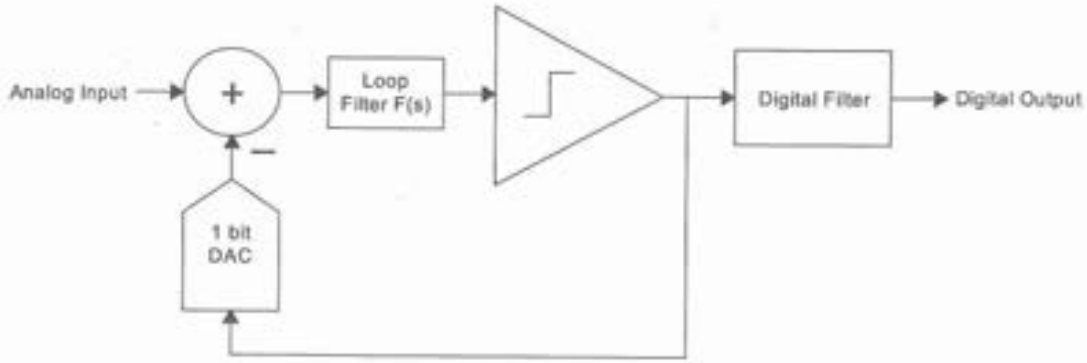


FIG 12.87 Sigma-delta ADC

between 40 to over 1000 are commonly used. As the oversampling ratio increases, so does the ADC precision (of course, at the expense of speed). Sigma-delta ADCs are commonly used in CD players at audio frequencies with resolutions up to 20 bits, but they have also been used at radio frequencies (10 MHz) for moderate precision (8–10 bits). They are popular CMOS converters because the analog components are limited to a few blocks, which can be designed carefully to obtain high performance. Representative examples can be found in [Kappes03, Sauerbrey02, Gupta03].

Apart from the basic ADCs presented, many more architectures have been invented. A single flash converter can be modified to be used twice at half the resolution and half the speed. This is the basis for a two-step flash converter. Finally, any converter can be used in parallel with delayed clocks to form an arbitrarily high-speed converter by interleaving the parallel converters. A CMOS converter implemented from 80 8-bit 250 MHz current-mode pipeline converters demonstrated operation at 20 GHz [Poulton03].

12.6.10 Radio Frequency (RF) Circuits

RF circuits are generally low in device count but high in design effort. While we will illustrate some RF circuits in this section, it must be understood that device sizing and component selection should be done in conjunction with a specialist RF design text [Lee98, Razavi98, Leung02]. To repeat, the point of showing these circuits is to encourage digital designers to explore these allied CMOS design areas.

As an introduction, a typical radio transceiver is shown in Figure 12.88. The receiver is what is called a *direct conversion* or *homodyne* architecture. It takes a 5 GHz carrier and downconverts this to a baseband signal by multiplying with a 5 GHz on-chip oscillator. The resulting signal is filtered by a 10 MHz low-pass filter and then amplified by a variable-gain amplifier. This signal is then fed to a 40 MHz ADC. While this is a simplified receiver, it includes most of the key modules required for RF applications at any frequency, namely, amplifiers (fixed-gain and variable-gain), oscillators, mixers, and filters.

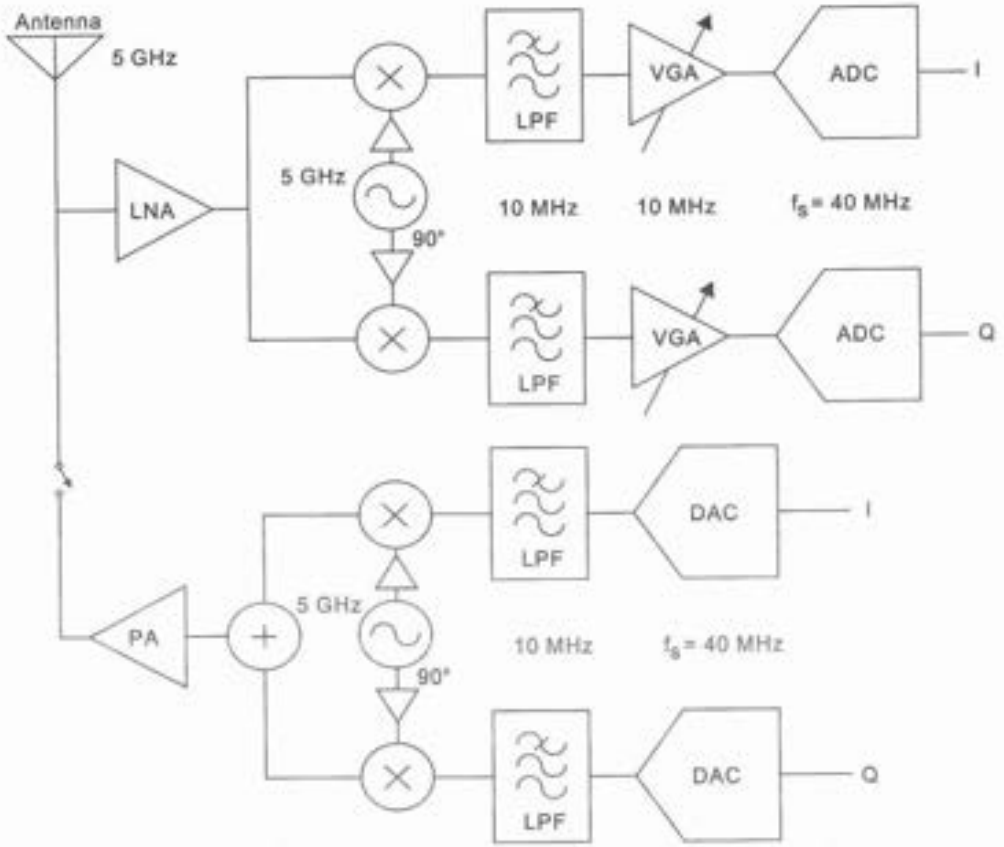


FIG 12.88 Typical CMOS radio transceiver

The transmitter path is roughly the reverse of the receiver. It starts with digital IQ values, which are fed to a DAC. The output is filtered by a low-pass reconstruction filter and then upconverted to 5 GHz. The output of the upmixer is amplified and fed to an antenna via an external transmit/receive switch. The transmitter is simpler than the receiver because of the larger signal amplitudes [vanZeijl02].

The rest of this section will introduce representative examples of each type of circuit introduced in this architecture.

A variety of CMOS RF amplifiers are shown in Figure 12.89 [Lee98]. Figure 12.89(a) shows a simple resistively loaded common source gain stage. It is biased by resistor $R1$ connected to V_{bias}. These circuits can give bandwidths in the GHz regions for submicron processes and should not be overlooked despite their simplicity. The stage is inherently wide-band. Figure 12.89(b) shows a tuned amplifier. Inductor $L1$ resonates with device and load capacitance to provide gain at a particular frequency. The stage is biased similarly to the resistively loaded example. Figure 12.89(c) shows the corresponding dif-

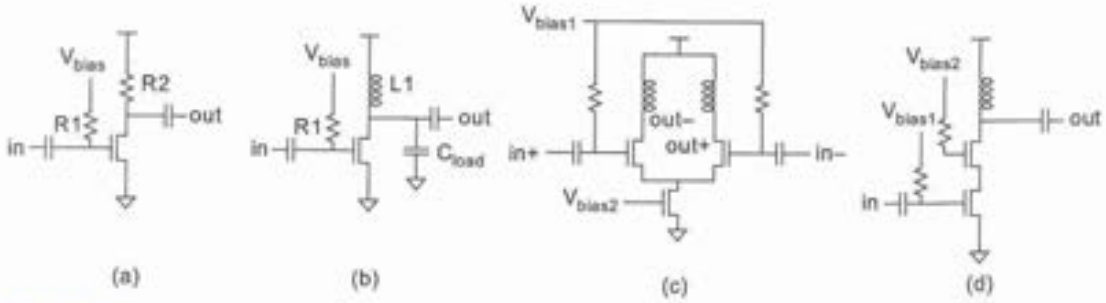


FIG 12.89 CMOS RF amplifiers

ferential stage, while Figure 12.89(d) shows a cascode gain stage. This is normally the starting point for a low-noise amplifier (LNA) that might be used at the front-end of a receiver.

Figure 12.90 shows the cascode RF amplifier with some added inductors. The source and gate inductances are used to tune out the input capacitance and present a resistive (50 Ω usually) load at the input. This stage can also be implemented differentially, which illustrates that while the initial circuit might be simple, extracting top performance from a simple gain stage requires sophisticated design.

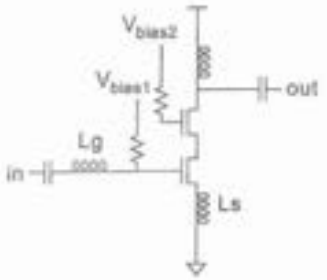


FIG 12.90 Cascode RF amplifier

Figure 12.91 shows an LC *voltage-controlled oscillator* (VCO) capable of operation well into the microwave regions (> 10 GHz) with modern processes in conjunction with the circuitry required to implement a phase locked loop. The inductors (*L1*, *L2*) resonate with the stray capacitance at the drain of *M3* and *M4* to oscillate at the required frequency. The cross-coupled nMOS transistors (*M3*,

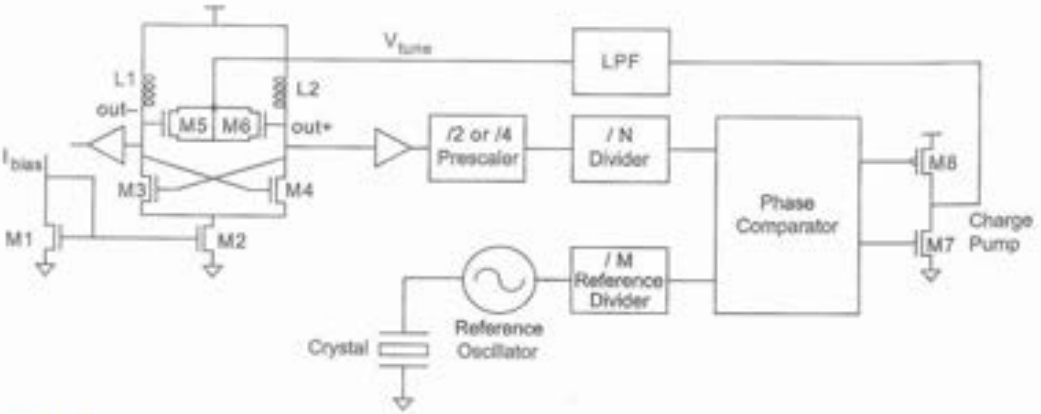


FIG 12.91 LC oscillator in a phased locked loop

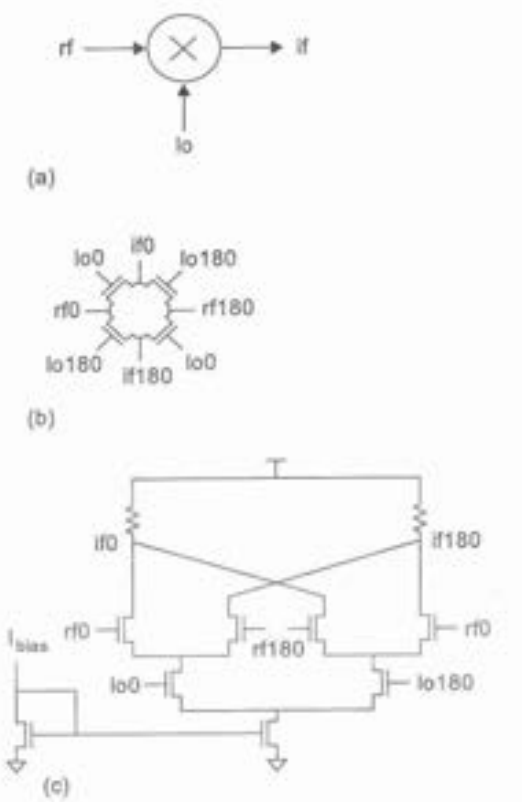


FIG 12.92 CMOS mixers

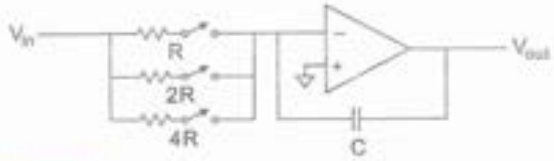


FIG 12.93 Low-pass filter

*M*4) provide the gain for the oscillator. Transistors *M*5 and *M*6 are used as *varactors* or voltage-variable capacitors to tune the oscillator frequency. *M*1/*M*2 forms a current mirror to bias the oscillator. The output of the oscillator is buffered and fed to a special *prescaler* that divides by two or four and presents a lower-frequency digital waveform to a conventional CMOS logic divider. A crystal oscillator is used as a reference by a phase comparator. The phase comparator feeds a charge pump, which, in conjunction with a low-pass filter, produces an increasing or decreasing analog voltage V_{tune} that is fed back to the *LC* oscillator. When configured properly with the correct ratios in the VCO divider and the reference divider, the feedback loop stabilizes the *LC* oscillator and prevents it from drifting with voltage or temperature.

Parameters of interest in an oscillator include the frequency of operation, power dissipation, and phase noise. The latter often determines the usefulness of a given oscillator in a particular system. Phase noise in turn is determined mainly by the gain of the transistors at the frequency of interest and the circuit *Q* of the inductors. A lot of effort goes into creating high *Q* inductors to achieve low-phase noise oscillators.

A *mixer* is an analog multiplier that converts one frequency to another. Figure 12.92(a) shows a symbol for a mixer and a typical application where a high radio frequency (RF) signal (say 5 GHz) is converted to a lower intermediate frequency (IF) signal (say 1 GHz) by mixing with a local oscillator (say 4 GHz). The mixer produces the sum and difference of the RF and LO (i.e., 1 GHz and 9 GHz). This is simply the result of multiplying two sine waves together. The unwanted product (9 GHz) is eliminated by filtering at the output of the mixer. The simplest CMOS mixer is the quad FET switch shown in Figure 12.92(b). Signals with their corresponding phases are shown. Correct bias has to be applied to each port. Figure 12.92(c) shows an active mixer based on a *Gilbert cell* [Zhou03, Melly01]. This has higher gain and lower noise than the ring mixer, but also has lower dynamic range.

Low-pass filters are circuits of use in an RF environment. Figure 12.93 shows a simple low-pass active filter that can be tuned by altering resistor combinations using CMOS switches. This form of continuous time filter can be increased in order, but the same type

of tuning can be employed. Alternatively, capacitors can be switched and fixed resistors used.

The CMOS switch idea can also be used to build variable-gain amplifiers as shown in Figure 12.94. This design employs binary weighted capacitors.

As mentioned previously, decreasing V_{DD} presents challenges for linear circuits, especially amplifiers. Figure 12.95 shows a high-speed op-amp with feed-forward compensation that only uses pMOS loaded differential stages, which work well at low supply voltages [Harrison03]. This is suitable for the filter and active gain control (AGC) applications mentioned previously. As with other circuits included in this section, the circuit is included here to illustrate the point that high-frequency amplifiers are not necessarily complex in circuit terms.

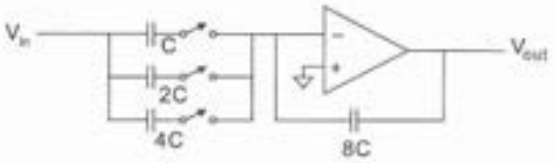


FIG 12.94 Gain controlled amplifier

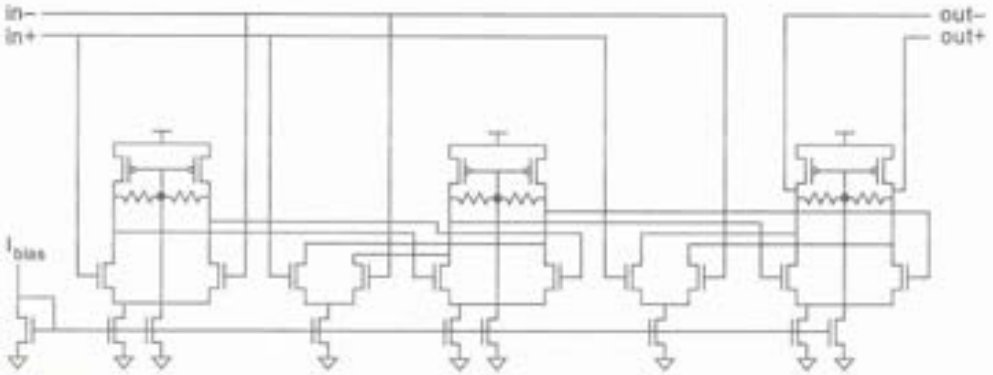


FIG 12.95 A high-speed CMOS op-amp

12.6.11 Analog Summary

This section was included to provide a brief introduction to the type of circuits that a designer might encounter in a mixed-signal analog or RF system-on-chip design. In such a short space, we can hardly do justice to this expansive area. However, we hope that the principles and circuits provided here will provide a springboard to understanding these circuits in more depth by consulting specialist texts and experimenting with circuits through simulation.

12.7 Pitfalls and Fallacies

Neglecting package parasitics

The resistance, capacitance, and inductance of the package have enormous impact on the power and I/O signal integrity of high-speed digital chips. They must be incorporated into modeling.

Using an inadequate power grid

A power grid should use generous amounts of the top two metal layers running in orthogonal directions. A mesh that mostly runs in only one direction is subject to excessive IR drops when many gates on a single wire switch simultaneously. It can also lead to serious inductive problems because of the huge current loops. The power grid should use many narrow wires interdigitated with the signals to provide a low S:R ratio rather than a few wide wires forming large current return loops. The grid should also avoid slots and other discontinuities that might lead to large current loops and high inductance.

Goofing your PLL/DLL

Phase-locked loops are notoriously difficult to design correctly. If poorly designed, they can oscillate at the wrong frequency, fail to acquire lock, or have excessive jitter. Careful circuit design is necessary to ensure they work across process variation and reject power supply noise. If the PLL does not work, testing the rest of the chip can be difficult or impossible. Most successful companies either have an in-house team that specializes in PLLs or they license their loops from a reputable third party.

Top Six Ways to Fool the Masses about Clock Skew

1) **Calculate clock skew without using process variation data**
   Random skew depends entirely on the mismatch of transistors (especially L_g) and wires on a chip. This mismatch varies with distance and layout technique. The process corners model worst-case variation from chip to chip, which can be far greater than between two nearby transistors; this results in unacceptably conservative skew budgets. But reliable data for on-chip variation can be hard to obtain, especially for small ASIC design teams and universities. Unless this data is used, clock skew budgeting is largely a matter of guesswork.

2) **Claim "zero skew"**
   Many papers state that a system has *zero skew* when the writers really mean that it has zero systematic skew. These systems may have significant random skew as well as drift and jitter. The term *zero skew* is deceptive and is best avoided.

3) **Report only systematic skew**
   Many papers also report only the systematic skew. In a well-balanced clock distribution network, systematic skew is often smaller than random skew and jitter.

4) **Ignore jitter**
   Jitter depends on time and space and is difficult to model or estimate. Unsophisticated clocking strategies sometimes ignore jitter. This results in unrealistic skew budgets. In particular, active deskew buffers increase clock distribution delay. Voltage noise on the buffers appears as jitter. Unless the supplies are unusually quiet, the buffers can increase jitter more than they decrease systematic or random skew.

5) **Report measured skew at only two elements over a brief period of time in a quiet environment**
   Measuring skew on a chip is also difficult. Some papers measure clock interarrival times at only two or a few points on the chip for a brief period of time and report those as the skew. As a chip has many clocked elements, you are unlikely to find the worst-case skew by measuring just a few points. Moreover, measurements over a brief time interval are unlikely to capture worst-case jitter. The chip

should be exercised through a variety of modes that cause large fluctuations in supply current to cause maximum power supply noise and clock jitter.

6) **Don't report the skew budget used during design** Designers often choose rather conservative clock skew budgets during design because they must ensure the design will operate correctly. Reporting a "measured" skew rather than a skew budget will give a smaller number.

12.8 Historical Perspective

Clock distribution is a persistent challenge in VLSI design. The clock often accounts for 30–50% of the chip's dynamic power. As clock periods have decreased, the clock distribution network requires more elaborate design to deliver clocks with skews below 10% of the cycle time. The DEC Alpha series of microprocessors led the clock frequency race through much of the 1990s. In this section, we will trace the evolution of their clock grids. Figure 12.96(a) shows the grids for the Alpha 21064, 21164, and 21264 microprocessors, while Figure 12.96(b) plots the systematic clocks skew across the die [Gronowski98].

In the 200 MHz Alpha 21064 [Dobberpuhl92], the clock grid drove TSPC latches directly without any local clock gaters. The final clock load is 3.5 nF, driven by a 35 cm (!) wide inverter arranged along the center of the chip. Such a wide inverter is built from many smaller inverters ganged in parallel. A binary tree layout is used to distribute the external clock to all of the inverters simultaneously. The clock skew is zero near the center and increases toward the edges. The clock driver generates so much heat that it raises the chip temperature by 30° C nearby. Checking for hold time violations is simply a matter of counting enough gates between latches.

In the 300–433 MHz Alpha 21164 [Bowhill95, Gronowski96], the clock grid directly drove conventional two-phase dynamic transmission gate latches. Two banks of clock buffers located midway between the center and the edges of the die drive the 3.75 nF clock load. The banked drivers also lead to a more even temperature distribution across the die.

The 600 MHz Alpha 21264 [Gieseke97, Gronowski98, Bailey98] used a clock grid that in turn drove an assortment of local clocks through gaters. The grid is divided into four "window panes," each of which is driven from its four edges. Each driver along the edge of a pane uses many inverters in parallel. The clock is distributed to the inverters in a fashion that equalizes the RC delay by tapering the wires appropriately. The systematic skew is close to zero along the edges of the panes and remains relatively small near the center because the wires are short. Some gaters perform clock gating to reduce power consumption. Others generate delayed clocks to fix critical paths. Because the system now has multiple logical clocks with different arrival times, checking for hold time violations requires more careful timing analysis.

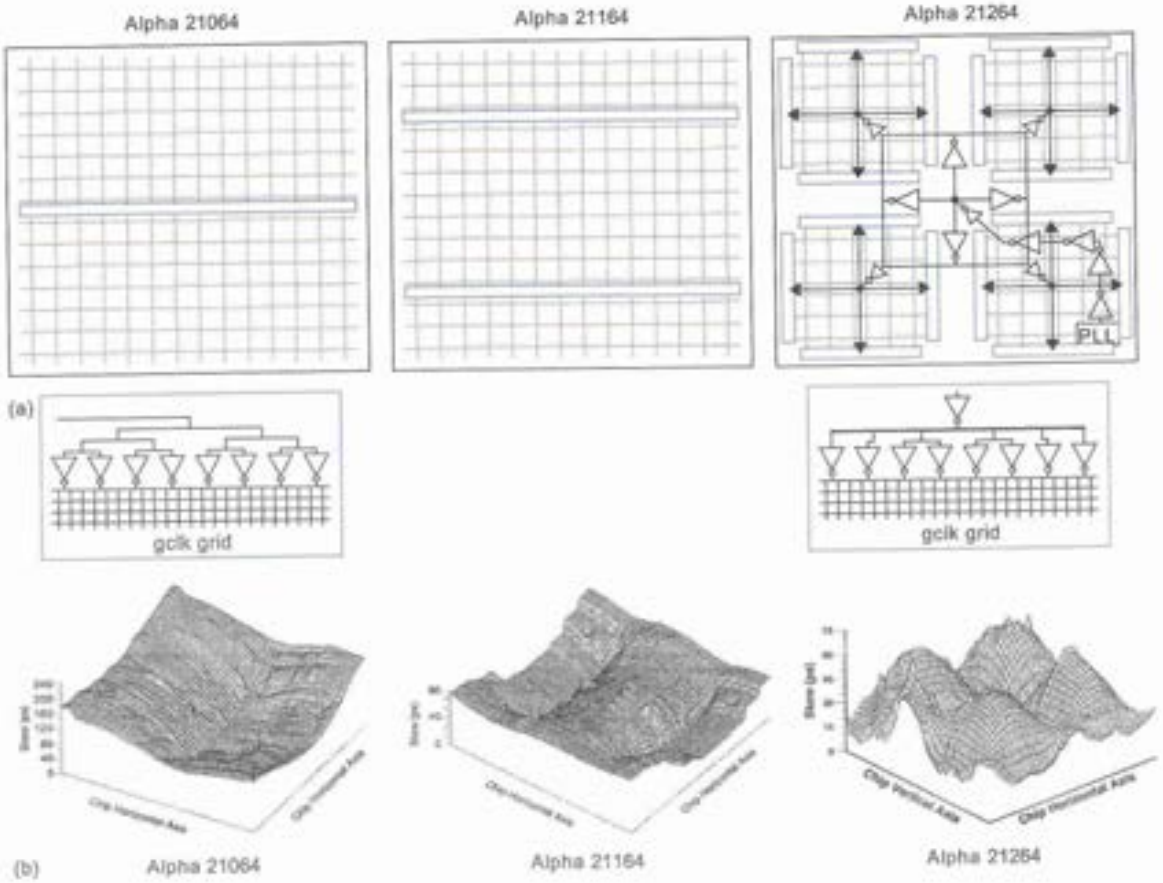


FIG 12.96 Alpha clock grids and systematic skew. (b) © 1998 IEEE.

In the Alpha 21064 and 21164, the global clock distribution network must have low resistance to directly drive such a large load capacitance. This in turn requires an enormous amount of metal wiring dedicated to the clock grid. The wire also has a high capacitance, so the clock grid consumes a large amount of power. For example, the 21164 clock distribution system consumes 20 W, or 40% of total chip power. In the 21264, the local clock gaters have electrical effort that reduces the capacitance seen directly on the global clock. This is beneficial because it reduces the metal usage and capacitance of the global clock grid while simultaneously reducing global clock skew. However, the local gaters introduce additional skew from random variation, drift, and jitter.

Summary

This chapter has surveyed package, power distribution, I/O, clock, and analog subsystem design. While each topic is a book in itself and a specialty design area, the short fat VLSI designer must understand enough about each area to optimize the system as a whole.

Packages connect the chip to the board or module, protect the chip, and are the first link in removing heat. They should offer plenty of connections, low thermal resistance, and low parasitics, while still being inexpensive to manufacture and test. Flip-chip packaging using solder bumps distributed across the die has become popular because of the large number of connections and low inductance.

The power distribution network consists of elements on the chip, package, and board. It must deliver a stable voltage across the chip under fluctuating current demands. Noise is caused by both average and peak current requirements. Multiple bypass capacitors offer low impedance to help filter high-frequency IR and L di/dt noise, but the DC supply resistance must be low enough to deliver the average current. V_{DD} and GND lines should be interdigitated in both directions with signal wires to provide small current return loops and low inductance. The supply wires must also have enough cross-sectional area to avoid electromigration problems. These requirements imply large amounts of metal and bypass capacitance, yet cost constraints dictate no more chip area than necessary.

I/O signals include inputs, outputs, bidirectional signals, and analog signals. The I/O pads must deliver adequate bandwidth to large off-chip capacitances at voltage levels compatible with other chips. They must also protect the core circuitry against overvoltage and electrostatic discharge.

A clocking subsystem includes clock generation, distribution, and gater elements. The clock generator can use a PLL to align the on-chip clock to an external reference for synchronous communication and to perform frequency multiplication. The clock distribution network should send the global clock to all clocked elements with low skew, yet not consume excessive power or area. The gaters perform local clock stopping or can produce multiple phases from the single global clock.

Basic analog building blocks include common source amplifiers, current mirrors, and differential amplifiers. From these, we can construct operational amplifiers, D/A converters, and A/D converters.

Exercises

12.1 A ceramic PGA package with a good heat sink and fan has a thermal resistance to the ambient of 10° C/W. The thermal resistance from the die to the package is 2° C/W. If the package is in a chassis that will never exceed 50° C and the maximum acceptable die temperature is 110° C, how much power can the chip dissipate?

12.2 Explain how an electrostatic discharge event could cause latchup on a CMOS chip.

12.3 Comment on the advantages and disadvantages of H-trees and clock grids. How does the hybrid tree/grid improve on a standard grid?

12.4 Calculate the bias-point and small-signal low-frequency gain of the common source amplifier from Figure 12.46 if $V_t = 0.7$, $\beta = 240\ \mu A/V^2$, and the nMOS output impedance is infinite. Let $V_{BIAS} = 3\ V$, $V_{DD} = 15\ V$, and $R_L = 10\ k\Omega$.

12.5 Prove EQ (12.24).

12.6 Calculate the output impedance of the Wilson current mirror in Figure 12.97.

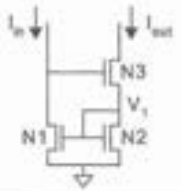


FIG 12.97 Wilson current mirror

12.7 Design a current source that sinks 200 μA, using the process parameters from the example in Section 12.6.4. What is the minimum drain voltage over which your current source operates?

12.8 Find the output impedance of your current source from Exercise 12.7. By what fraction does the current change as the output node changes by 1 V?

12.9 Simulate the operational amplifier of Figure 12.63. Using minimum-size transistors and a 10 kΩ resistor, what is the gain?

12.10 What changes would you make to the amplifier from Exercise 12.9 to increase the gain? What are the tradeoffs involved? What gain can you achieve using reasonable changes?

12.11 Prove EQ (12.31).

12.12 Use SPICE to find the transconductance and output resistance of a minimum-size transistor in your process biased at $V_{gs} = V_{ds} = V_{DD}/2$. What is the $g_m r_o$ product?

12.13 Repeat Exercise 12.12 for a transistor with 2x minimum channel length. How does the product change?

12.14 In Section 12.6.8 on resistor string DACs, it was mentioned that a similar DAC can be implemented with capacitors. Design the architecture of a 4-bit capacitor DAC.

12.15 A bias generator is required to generate 16 steps from 0 to 100 μA to bias an amplifier. Design a CMOS DAC to do this, assuming the presence of a 50 μA reference current.

12.16 Differential circuits provide good noise immunity and have the advantage of dual rail inputs and outputs. Pseudo-differential circuits based on CMOS inverter amplifiers can be implemented by using two signal paths that process each signal, but do not provide for the same level of noise immunity. Design a single stage of a pipeline ADC that uses this style of differential circuit.

12.17 To reduce clock and decoder skew in a current-mode DAC, a latch is often included in the current cell. Design the circuit for such a cell, demonstrating where the latch would be placed. If this is a slave latch, where would the master latch be located?

Appendix A Verilog

A.1 Introduction

This appendix gives a quick introduction to the Verilog Hardware Description Language (HDL). There are many texts on Verilog ([Smith00, Thomas02, Ciletti99] and others) that provide a more in-depth treatment. The IEEE standard itself is quite readable as well as authoritative [IEEE1364-01]. Many books treat Verilog as a programming language, which is not the best way of viewing it. Verilog is better understood as a shorthand for describing digital hardware. It is best to begin your design process by planning, on paper or in your mind, the hardware you want. (For example, the MIPS processor consists of an FSM controller and a datapath built from registers, adders, multiplexers, etc.) Then, write Verilog that implies that hardware to a synthesis tool. A common error among beginners is to write a program without thinking about the hardware that is implied. If you don't know what hardware you are implying, you are almost certain to get something that you didn't want. Sometimes this means extra latches appearing in your circuit in places you didn't expect. Other times, it means that the circuit is much slower than required or it takes far more gates than it would have if it were more carefully described.

The Verilog language was developed by Gateway Design Automation as a proprietary language for logic simulation in 1984. Gateway was acquired by Cadence in 1989 and Verilog was made an open standard in 1990 under the control of Open Verilog International. The language, with some revisions, became an IEEE standard in 1995 and was updated in 2001. This appendix is consistent with the 2001 standard.

As mentioned in Section 1.8.4, there are two general styles of description: *behavioral* and *structural*. Structural Verilog describes how a module is composed of simpler modules or basic primitives such as gates or transistors. Behavioral Verilog describes how the outputs are computed as functions of the inputs. There are two general types of statements used in behavioral Verilog. *Continuous assignment* statements necessarily imply combinational logic[1] because the output on the left side is a function of the inputs on the right side. *Always* blocks can imply combinational logic or sequential logic, depending on how they are used. It is good practice to partition your design into combinational and sequential components and then write Verilog in such a way that you get what you want. If you

[1] Recall that the outputs of *combinational logic* depend only on the present inputs, while outputs of *sequential logic* depend on both past and present inputs. In other words, combinational logic is memoryless, while sequential logic has memory or *state*.

don't know whether a block of logic is combinational or sequential, you are likely to get the wrong thing. A particularly common mistake is to use `always` blocks to model combinational logic, but to accidentally imply latches or flip-flops.

This appendix focuses on a subset of Verilog sufficient to synthesize any hardware function. The language contains many other commands that are beyond the scope of this tutorial.

A.2 Behavioral Modeling with Continuous Assignments

A 32-bit adder is a complex design at the schematic level of representation. It can be constructed from 32 full adder cells, each of which in turn requires about six 2-input gates. Verilog provides a much more compact description. In each of the examples in this appendix, Synplify Pro was used to synthesize the Verilog into hardware. The Verilog code is shown adjacent to the schematic it implies.

```
module adder(input  [31:0] a,
             input  [31:0] b,
             output [31:0] y);

   assign y = a + b;
endmodule
```

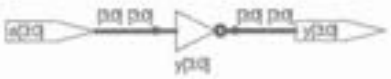
A Verilog module is like a cell in a schematic. It begins with a description of the inputs and outputs, which in this case are 32-bit busses.

During simulation, an `assign` statement causes the left side (`y`) to be updated any time the right side (`a/b`) changes. This necessarily implies combinational logic; the output on the left side is a function of the current inputs given on the right side. A 32-bit adder is a good example of combinational logic.

A.2.1 Bitwise Operators

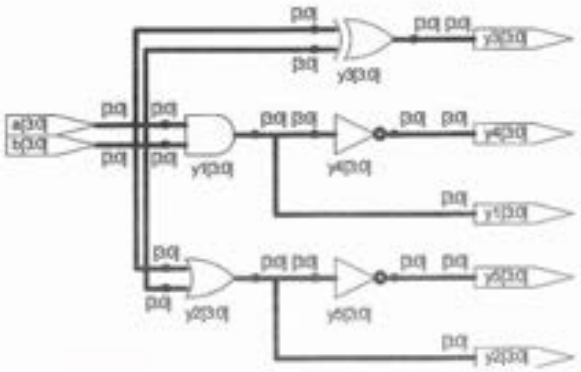
Verilog has a number of *bitwise* operators that act on busses. For example, the following module describes four inverters.

```
module inv(input  [3:0] a,
           output [3:0] y);

   assign y = ~a;
endmodule
```



Similar bitwise operations are available for the other basic logic functions:

```
module gates(input [3:0]  a, b,
             output [3:0] y1, y2, y3, y4, y5);

   /* Five different two-input logic
      gates acting on 4 bit busses */
   assign y1 = a & b;     // AND
   assign y2 = a | b;     // OR
   assign y3 = a ^ b;     // XOR
   assign y4 = ~(a & b); // NAND
   assign y5 = ~(a | b); // NOR
endmodule
```



A.2.2 Comments and White Space

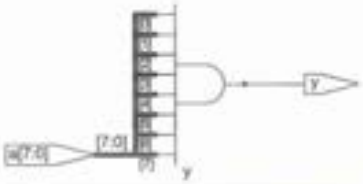
The previous examples showed two styles of comments, just like those used in C or Java. Comments beginning with `/*` continue, possibly across multiple lines, to the next `*/`. Comments beginning with `//` continue to the end of the line. It is important to properly comment complex logic so that six months from now, you can understand what you did or so that some poor slob assigned to fix your buggy code will be able to figure it out. ☺

Verilog is not picky about the use of white space. Nevertheless, proper indenting and spacing is helpful to make nontrivial designs readable. Verilog is case-sensitive. Be consistent in your use of capitalization and underscores in signal and module names. Be sparing with the use of underscores because they can increase the risk of carpal tunnel syndrome.

A.2.3 Reduction Operators

Reduction operators imply a multiple-input gate acting on a single bus. For example, the following module describes an 8-input AND gate with inputs `a[0]`, `a[1]`, `a[2]`, ..., `a[7]`.

```
module and8(input  [7:0] a,
            output       y);

   assign y = &a;
endmodule
```

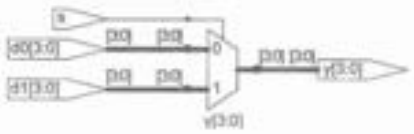


As one would expect, `|`, `^`, `~&`, and `~|` reduction operators are available for OR, XOR, NAND, and NOR as well. Recall that a multi-bit XOR performs parity, returning true if an odd number of inputs are true.

A.2.4 Other Operators

The conditional operator `?:` works like the same operator in C or Java and is useful for describing multiplexers. It is called a *ternary operator* because it takes three inputs. If the first input is nonzero, the result is the expression in the second input. Otherwise, the result is the expression in the third input.

```
module mux2(input  [3:0] d0, d1,
            input        s,
            output [3:0] y);

   assign y = s ? d1 : d0;
endmodule
```



A number of arithmetic functions are supported, including `+`, `-`, `*`, `<`, `>`, `<=`, `>=`, `==`, `!=`, `<<`, `>>`, `<<<`, `>>>`, `/`, and `%`. Recall from other languages that % is the modulo operator: `a%b` equals the remainder of `a` when divided by `b`. These operations imply a vast amount of hardware. `==` and `!=` (equality/inequality) on N-bit inputs require N 2-input XNORs to determine equality of each bit and an N-input AND or NAND to combine all the bits, as shown in Figure 10.52. Addition, subtraction, and comparison all require an adder, which is expensive in hardware. Variable logical left and right shifts `<<` and `>>` and arithmetic left and right shifts `<<<` and `>>>` imply a barrel shifter. Multipliers are even more costly. Do not use these statements without contemplating the number of gates you are generating. Moreover, the implementations are not necessarily efficient for your problem.

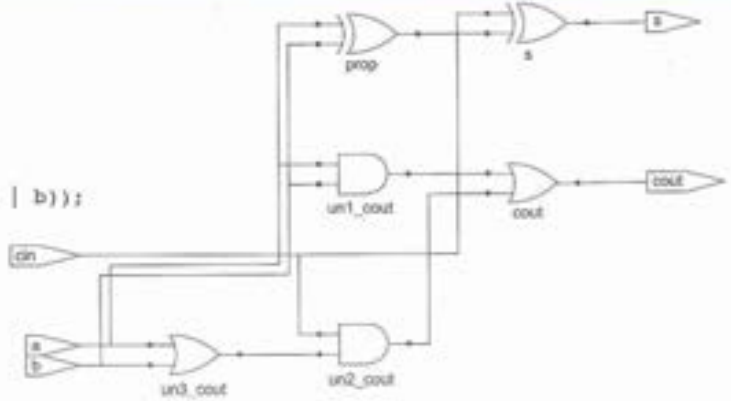
Some synthesis tools ship with optimized libraries for special functions like adders and multipliers. For example, the Synopsys DesignWare libraries produce reasonably good multipliers. If you do not have a license for the libraries, you'll probably be disappointed with the speed and gate count of a multiplier your synthesis tool produces from when it sees `*`. Many synthesis tools choke on `/` and `%` because these are nontrivial functions to implement in combinational logic.

A.3 Basic Constructs

A.3.1 Internal Signals

Often it is convenient to break a complex calculation into intermediate variables. For example, in a full adder, we sometimes define the propagate signal as the XOR of the two inputs A and B. The sum from the adder is the XOR of the propagate signal and the carry-in. We can declare the propagate signal using a `wire` statement, in much the same way we use local variables in a programming language.

```
module fulladder(input  a, b, cin,
                 output s, cout);

   wire prop;

   assign prop = a ^ b;
   assign s = prop ^ cin;
   assign cout = (a & b) | (cin & (a | b));
endmodule
```



Technically, it is not necessary to declare single-bit wires. However, it is necessary to declare multi-bit busses and is good practice to declare all signals. Some Verilog simulation and synthesis tools give errors that are difficult to decipher when a wire is not declared.

A.3.2 Precedence

Notice that we fully parenthesized the `cout` computation. We could take advantage of operator precedence to use fewer parentheses:

```
assign cout = a&b | cin&(a|b);
```

The operator precedence from highest to lowest is much as you would expect in other languages, as shown in Table A.1. AND has precedence over OR.

Table A.1 Operator Precedence

Symbol	Meaning	Precedence
~	NOT	Highest
*, /, %	MUL, DIV, MODULO	
+, -	PLUS, MINUS	
<<, >>,	Logical Left/Right Shift	
<<<, >>>	Arithmetic Left/Right Shift	
<, <=, >, >=	Relative Comparison	
==, !=	Equality Comparison	
&, ~&	AND, NAND	
^, ~^	XOR, XNOR	
|, ~|	OR, NOR	
?:	Conditional	Lowest

A.3.3 Constants

Constants can be specified in binary, octal, decimal, or hexadecimal. Table A.2 gives examples.

Table A.2 Constants

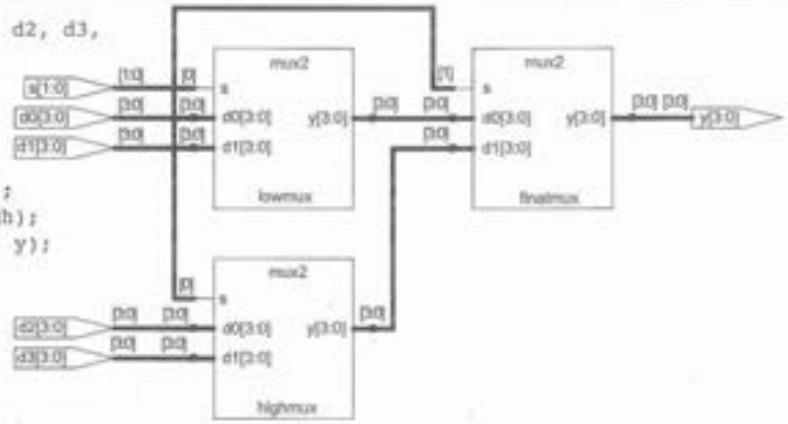
Number	# Bits	Base	Decimal Equivalent	Stored
`3'b101`	3	Binary	5	101
`'b11`	unsized	Binary	3	000000..00011
`8'b11`	8	Binary	3	00000011
`8'b1010_1011`	8	Binary	171	10101011
`3'd6`	3	Decimal	6	110
`6'o42`	6	Octal	34	100010
`8'hAB`	8	Hexadecimal	171	10101011
`42`	unsized	Decimal	42	0000...00101010

It is good practice to specify the length of the number in bits, even though the second row shows that this is not strictly necessary. If you don't specify the length, one day you may be surprised when Verilog assumes the constant has additional leading 0's that you didn't intend. Underscores in numbers are ignored and can be helpful in breaking long numbers into more readable chunks. If the base is omitted, the number is assumed to be decimal.

A.3.4 Hierarchy

Nontrivial designs are developed in a hierarchical form, in which complex modules are composed of submodules. For example, a 4-input multiplexer can be constructed from three 2-input multiplexers:

```
module mux4(input  [3:0] d0, d1,  d2, d3,
            input  [1:0] s,
            output [3:0] y);

   wire [3:0] low, high;

   mux2 lowmux(d0, d1, s[0], low);
   mux2 highmux(d2, d3, s[0], high);
   mux2 finalmux(low, high, s[1], y);
endmodule
```



This is an example of the structural coding style because the mux is built from simpler modules. It is good practice to avoid (or at least minimize) mixing structural and behavioral descriptions within a single module. Generally, simple modules are described behaviorally and larger modules are composed structurally from these building blocks.

A.3.5 Tristates

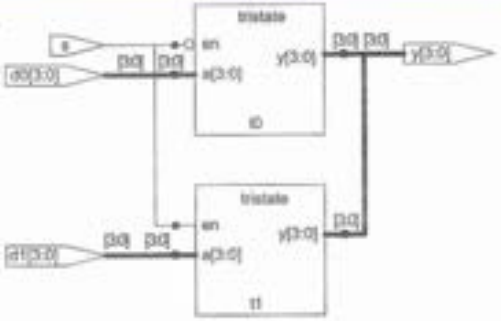
It is possible to leave a bus floating rather than drive it to 0 or 1. This floating value is called `'z` in Verilog. For example, a tristate buffer produces a floating output when the enable is false.

```
module tristate(input  [3:0] a,
                input        en,
                output [3:0] y);

   assign y = en ? a : 4'bz;
endmodule
```

Floating inputs to gates causes undefined outputs, displayed as `'x` in Verilog. At start-up, state nodes such as the internal node of flip-flops are also usually initialized to `'x`, as we will see later.

We could define a multiplexer using two tristates so that the output is continuously driven by exactly one tristate. This guarantees that there are no floating nodes.

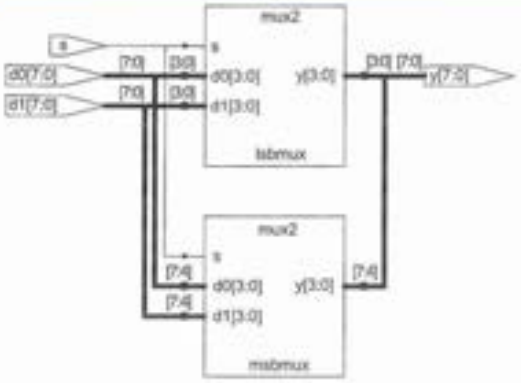
```
module mux2(input  [3:0] d0, d1,
            input        s,
            output [3:0] y);

   tristate t0(d0, ~s, y);
   tristate t1(d1, s, y);
endmodule
```



A.3.6 Bit Swizzling

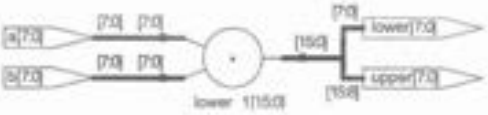
Often it is necessary to operate on parts of a bus or to concatenate (join together) signals to construct busses. In the `mux4` example, the least significant bit `s[0]` of a 2-bit select signal was used for the low and high muxes and the most significant bit `s[1]` was used for the final mux. Use ranges to select subsets of a bus. For example, an 8-bit wide 2-input mux can be constructed from two 4-bit wide 2-input muxes.

```
module mux2_8(input  [7:0] d0, d1,
              input        s,
              output [7:0] y);

   mux2 lsbmux(d0[3:0], d1[3:0], s, y[3:0]);
   mux2 msbmux(d0[7:4], d1[7:4], s, y[7:4]);
endmodule
```



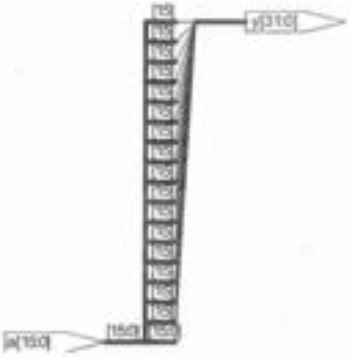
The {} notation is used to concatenate busses. For example, the following 8 × 8 multiplier produces a 16-bit result, which is placed on the upper and lower 8-bit result busses.

```
module mul(input [7:0] a, b,
           output [7:0] upper, lower);

   assign {upper, lower} = a*b;
endmodule
```



A 16-bit 2's complement number is sign-extended to 32 bits by copying the most significant bit to each of the upper 16 positions. The Verilog syntax concatenates 16 copies of `a[15]` to the 16-bit `a[15:0]` bus. Some synthesis tools produce a warning that `a` is a "feedthrough net." This means that the input "feeds through" to the output. `y[15:0]` should have the same value as `a[15:0]`, so we did intend a feedthrough. If you get a feedthrough net warning where you did not intend one, check for a mistake in your Verilog.

```
module signextend(input  [15:0] a,
                  output [31:0] y);

   assign y = {{16{a[15]}}, a[15:0]};
endmodule
```



The next statement generates a silly combination of two busses. Don't confuse the 3-bit binary constant `3'b101` with bus `b`. Note that it was important to specify the length of 3 bits in the constant; otherwise many additional 0's might have appeared in the middle of `y`.

```
assign y = {a[2:1], {3{b[0]}}, a[0], 3'b101, b[1:3]};
```

This produces

```
y = a[2] a[1] b[0] b[0] b[0] a[0] 1 0 1 b[1] b[2] b[3]
```

A.3.7 Delays

The delay of a statement can be specified in arbitrary units. For example, the following code defines an inverter with a 42-unit propagation delay. Delays have no impact on synthesis, but can be helpful while debugging simulation waveforms because they make cause and effect more apparent.

```
assign #42 y = ~a;
```

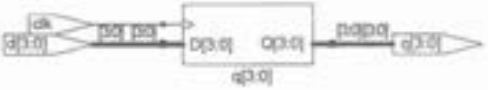
A.4 Behavioral Modeling with `Always` Blocks

`Assign` statements are reevaluated every time any term on the right side changes. Therefore, they must describe combinational logic. `Always` blocks are reevaluated only when signals in the header (called a *sensitivity list*) change. Depending on the form, `always` blocks can imply either sequential or combinational circuits.

A.4.1 Registers

Verilog refers to edge-triggered flip-flops as *registers*. Registers are described with an `always @(posedge clk)` statement:

```
module flop(input            clk,
            input      [3:0] d,
            output reg [3:0] q);

   always @(posedge clk)
      q <= d;
endmodule
```



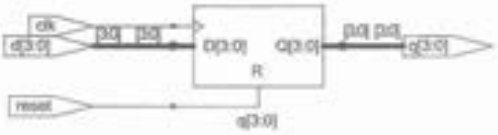
The body of the `always` statement is only evaluated on the rising (positive) edge of the clock. At this time, the output `q` is copied from the input `d`. The `<=` is called a *nonblocking assignment* and is pronounced "*gets*," as in "*q gets d*." Think of it as a regular equals sign

for now; we'll return to the more subtle points in Section A.4.6. Notice that `<=` is used instead of `assign` inside the `always` block.

All the signals on the left side of assignments in `always` blocks must be declared as `reg`. This is a confusing point for new Verilog users. In this circuit, `q` is also the output. Declaring a signal as `reg` does not mean the signal is actually a register! All it means is that it appears on the left side in an `always` block. We will see examples of combinational signals later that are declared `reg`, but that have no flip-flops.

At startup, the `q` output is initialized to `'x`. Generally, it is good practice to use resettable registers so that on power-up you can put your system in a known state. The reset can be either asynchronous or synchronous, as discussed in Section 7.3.4. Asynchronous resets occur immediately. Synchronous resets only change the output on the rising edge of the clock.

```
module flopr(input            clk,
             input            reset,
             input      [3:0] d,
             output reg [3:0] q);

   // asynchronous reset
   always @(posedge clk, posedge reset)
      if (reset) q <= 4'b0;
      else       q <= d;
endmodule
```

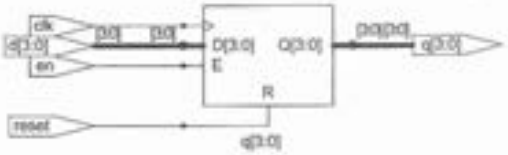



```
module flopr(input            clk,
             input            reset,
             input      [3:0] d,
             output reg [3:0] q);

   // synchronous reset
   always @(posedge clk)
      if (reset) q <= 4'b0;
      else       q <= d;
endmodule
```

Note that the asynchronously resettable flop evaluates the always block when either `clk` or `reset` rise so that it immediately responds to `reset`. The synchronously reset flop is not sensitized to `reset` in the `@` list, so it waits for the next clock edge before clearing the output.

You can also consider registers with enables that only respond to the clock when the enable is true. The following register with enable and asynchronous reset retains its old value if both `reset` and `en` are false.

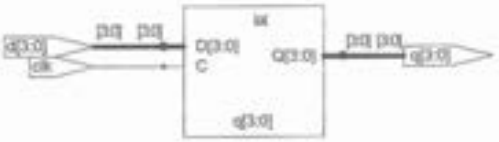
```
module flopren(input          clk,
               input          reset,
               input          en,
               input      [3:0] d,
               output reg [3:0] q);

   // asynchronous reset
   always @(posedge clk, posedge reset)
      if (reset) q <= 4'b0;
      else if (en) q <= d;
endmodule
```



A.4.2 Latches

`Always` blocks can be used to model transparent latches, also known as D latches When the clock is high, the latch is *transparent* and the data input flows to the output. When the clock is low, the latch goes *opaque* and the output remains constant.

```
module latch(input          clk,
             input      [3:0] d,
             output reg [3:0] q);

   always @(clk, d)
      if (clk) q <= d;
endmodule
```



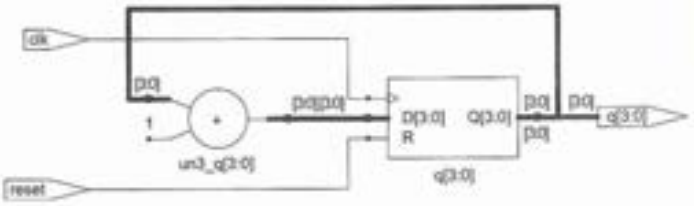
The latch evaluates the `always` block any time either `clk` or `d` change. If `clk` is high, the output gets the input. Notice that even though `q` is a latch node, not a register node, it is still declared as `reg` because it is on the left side of a `<=` in an `always` block. Some synthesis tools are primarily intended to target edge-triggered flip-flops and will produce warnings when generating latches.

A.4.3 Counters

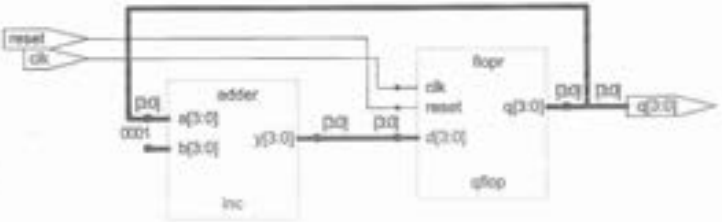
Consider two ways of describing a 4-bit counter with asynchronous reset. The first scheme (behavioral) implies a sequential circuit containing both the 4-bit register and an adder. The second scheme (structural) explicitly declares modules for the register and adder.

Either scheme is good for a simple circuit such as a counter. As you develop more complex finite state machines, it is a good idea to separate the next state logic from the registers in your Verilog code. Verilog does not protect you from yourself here and there are many simple errors that lead to circuits unlike those you intended, as will be explored in Section A.9.

```
module counter(input          clk,
               input          reset,
               output reg [3:0] q);

   // counter using always block

   always @(posedge clk)
      if (reset) q <= 4'b0;
      else       q <= q+1;
endmodule
```



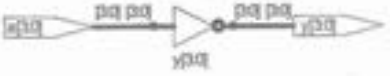

```
module counter(input        clk,
               input        reset,
               output [3:0] q);

   wire [3:0] nextq;

   // counter using module calls

   flopr qflop(clk, reset, nextq, q);
   adder inc(q, 4'b0001, nextq);
   // assumes a 4-bit adder
endmodule
```



A.4.4 Combinational Logic

`Always` blocks imply sequential logic when some of the inputs do not appear in the `@` stimulus list or might not cause the output to change. For example, in the flop module, `d` is not in the `@` list, so the flop does not immediately respond to changes of `d`. In the latch, `d` is in the `@` list, but changes in `d` are ignored unless `clk` is high. `Always` blocks can also be used to imply combinational logic if they are written in such a way that the output is reevaluated every time there are changes in any of the inputs. The following code shows how to define a bank of inverters with an `always` block. Note that `y` must be declared as `reg` because it appears on the left side of a `<=` or `=` sign in an `always` block. Nevertheless, `y` is the output of combinational logic, not a register.

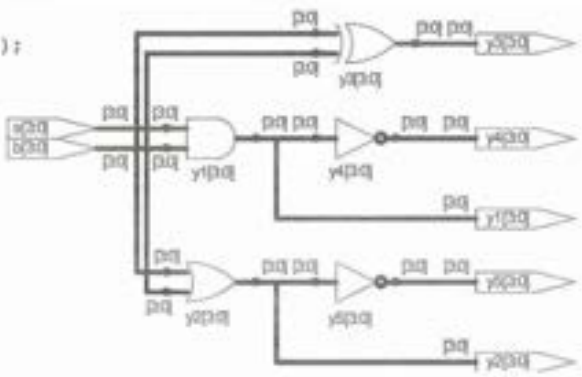
```
module inv(input      [3:0] a,
           output reg [3:0] y);

   always @(*)
      y <= ~a;
endmodule
```



`always @(*)` evaluates the statements inside the `always` block whenever any of the signals on the right side of `<=` or `=` change inside the `always` block. Thus `@(*)` is a safe way to model combinational logic. In this particular example, `@(a)` would also have sufficed.

Similarly, the next example defines five banks of different kinds of gates. In this case, an `@(a, b)` would have been equivalent to `@(*)`. However, `@(*)` is better because it avoids common mistakes of missing signals in the stimulus list. Also notice that the `begin` / `end` construct is necessary because multiple commands appear in the `always` block. This is analogous to `{ }` block structure in C or Java. The `begin` / `end` was not needed in the `flopr` example because an `if` / `else` command counts as a single statement.

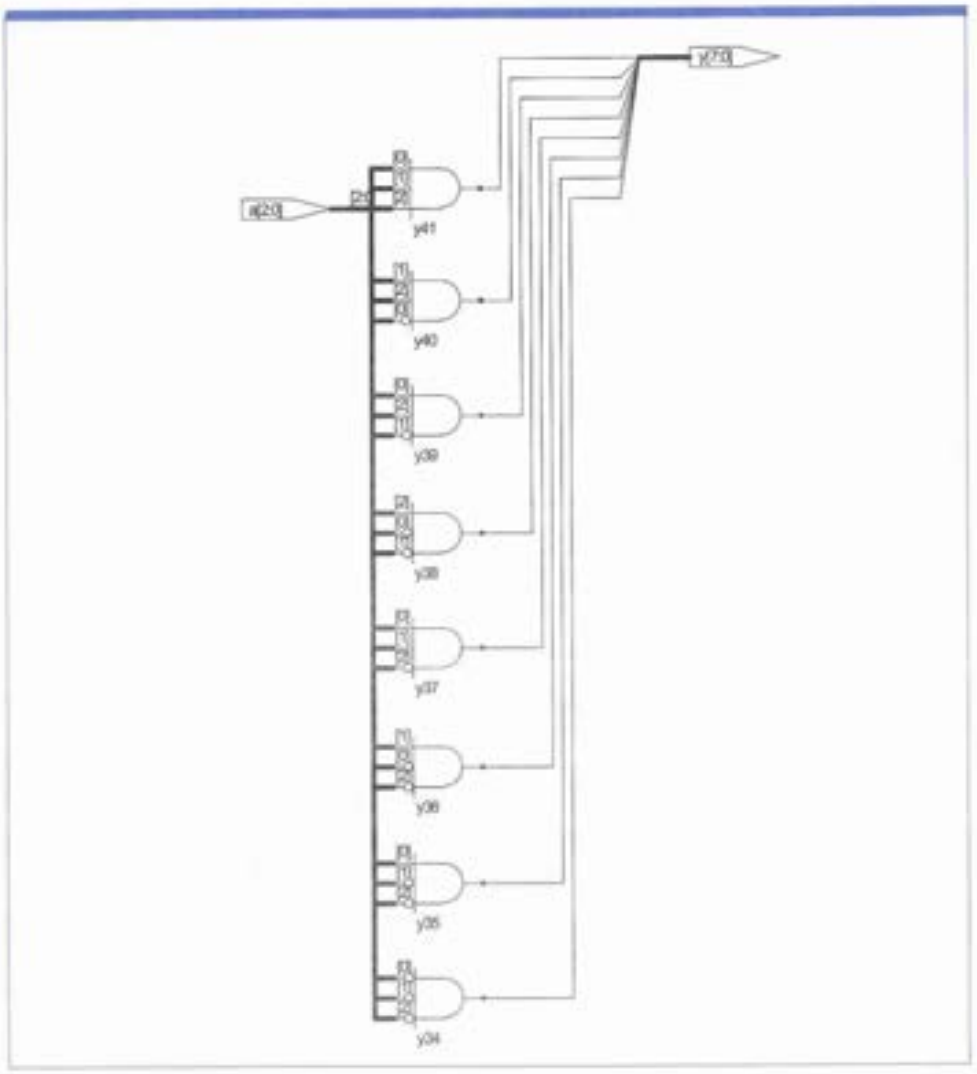
```
module gates(input       [3:0] a, b,
             output reg [3:0] y1, y2, y3, y4, y5);

   always @(*)
      begin
         y1 <= a & b; // AND
         y2 <= a | b; // OR
         y3 <= a ^ b; // XOR
         y4 <= ~(a & b); // NAND
         y5 <= ~(a | b); // NOR
      end
endmodule
```



These two examples are poor applications of `always` blocks for modeling combinational logic because they require more lines than the equivalent approach with `assign` statements. Moreover, they pose the risk of inadvertently implying sequential logic (see Section A.9.2). A better application of the `always` block is a decoder, which takes advantage of the `case` statement that can only appear inside an `always` block.

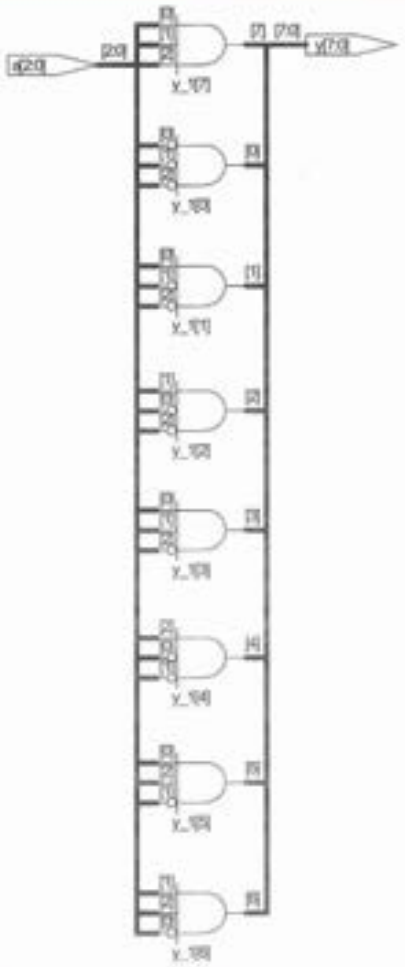
```
module decoder_always(input       [2:0] a,
                      output reg [7:0] y);

   // a 3:8 decoder
   always @(*)
      case (a)
         3'b000:   y <= 8'b00000001;
         3'b001:   y <= 8'b00000010;
         3'b010:   y <= 8'b00000100;
         3'b011:   y <= 8'b00001000;
         3'b100:   y <= 8'b00010000;
         3'b101:   y <= 8'b00100000;
         3'b110:   y <= 8'b01000000;
         3'b111:   y <= 8'b10000000;
      endcase
endmodule
```

continued



Using the `case` statement is probably clearer than using Boolean equations in an `assign` statement:

```
module decoder_assign(input  [2:0] a,
                      output [7:0] y);

  assign y[0] = ~a[0] & ~a[1] & ~a[2];
  assign y[1] =  a[0] & ~a[1] & ~a[2];
  assign y[2] = ~a[0] &  a[1] & ~a[2];
  assign y[3] =  a[0] &  a[1] & ~a[2];
  assign y[4] = ~a[0] & ~a[1] &  a[2];
  assign y[5] =  a[0] & ~a[1] &  a[2];
  assign y[6] = ~a[0] &  a[1] &  a[2];
  assign y[7] =  a[0] &  a[1] &  a[2];
endmodule
```



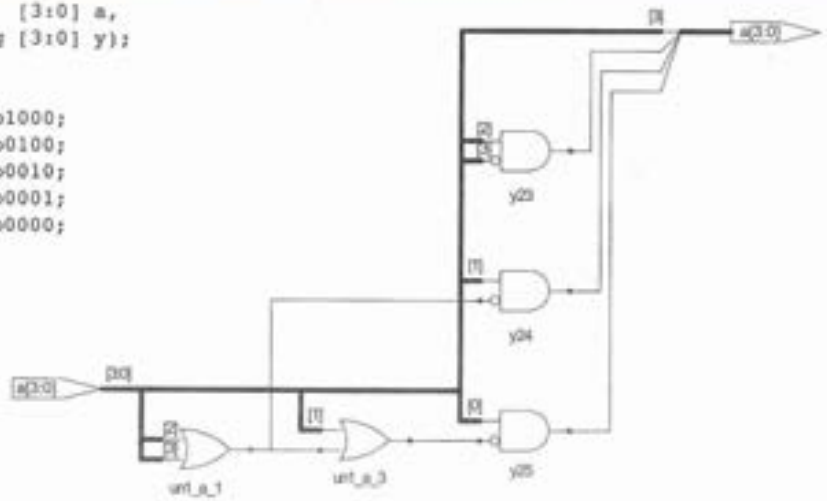
An even better example is the logic for a 7-segment display decoder from [Ciletti99]. The 7-segment display is shown in Figure A.1. The decoder takes a 4-bit number and displays its decimal value on the segments. For example, the number 0111 = 7 should turn on segments `a`, `b`, and `c`. The equivalent logic with `assign` statements describing the detailed logic for each bit would be tedious. This more abstract approach is faster to write, clearer to read, and can be automatically synthesized down to an efficient logic implementation. This example also illustrates the use of parameters to define constants to make the code more readable. The `case` statement has a `default` to display a blank output when the input is outside the range of decimal digits.

FIG A.1 7-segment display mapping

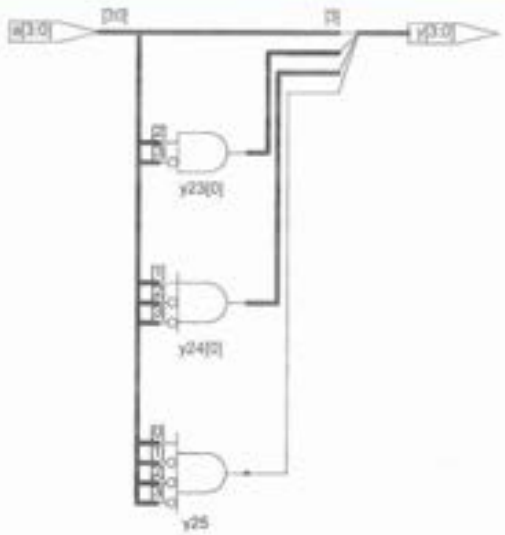
```
module sevenseg(input       [3:0] data,
                output reg [6:0] segments);

   // Segment  #          abc_defg
   parameter   BLANK = 7'b000_0000;
   parameter   ZERO  = 7'b111_1110;
   parameter   ONE   = 7'b011_0000;
   parameter   TWO   = 7'b110_1101;
   parameter   THREE = 7'b111_1001;
   parameter   FOUR  = 7'b011_0011;
   parameter   FIVE  = 7'b101_1011;
   parameter   SIX   = 7'b101_1111;
   parameter   SEVEN = 7'b111_0000;
   parameter   EIGHT = 7'b111_1111;
   parameter   NINE  = 7'b111_1011;

   always @(*)
      case (data)
         0: segments <= ZERO;
         1: segments <= ONE;
         2: segments <= TWO;
         3: segments <= THREE;
         4: segments <= FOUR;
         5: segments <= FIVE;
         6: segments <= SIX;
         7: segments <= SEVEN;
         8: segments <= EIGHT;
         9: segments <= NINE;
         default: segments <= BLANK;
      endcase
endmodule
```

Finally, compare three descriptions of a priority encoder that sets one output true corresponding to the most significant input that is true. The `if` statement can appear in `always` blocks and makes the logic quite readable. The `casez` statement also appears in `always` blocks and allows don't care's in the case logic, indicated with the `?` symbol. The `assign` statements synthesize to the same results, but are arguably less clear to read. Note that `a[3]` is another example of a feedthrough net because `y[3] = a[3]`. Of these three styles, the `if/else` approach is recommended for describing priority encoders because it is the easiest for most engineers to recognize. `case` statements are best reserved for functions specified by truth tables and `casez` statements should be used for functions specified by truth tables with don't cares. In the first two descriptions, `y` must be declared as a `reg` because it is assigned inside an `always` block.

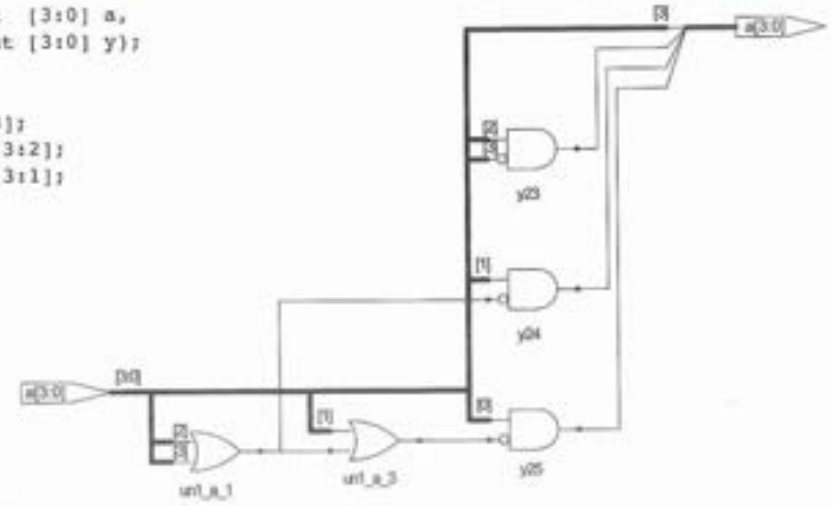
```
module priority_if(input       [3:0] a,
                   output reg [3:0] y);

   always @(*)
      if      (a[3]) y <= 4'b1000;
      else if (a[2]) y <= 4'b0100;
      else if (a[1]) y <= 4'b0010;
      else if (a[0]) y <= 4'b0001;
      else           y <= 4'b0000;
endmodule
```




```
module priority_casez(input       [3:0] a,
                      output reg [3:0] y);

   always @(*)
      casez(a)
         4'b1???: y <= 4'b1000;
         4'b01??: y <= 4'b0100;
         4'b001?: y <= 4'b0010;
         4'b0001: y <= 4'b0001;
         default: y <= 4'b0000;
      endcase
endmodule
```



```
module priority_assign(input  [3:0] a,
                       output [3:0] y);

   assign y[3] = a[3];
   assign y[2] = a[2] & ~a[3];
   assign y[1] = a[1] & ~|a[3:2];
   assign y[0] = a[0] & ~|a[3:1];
endmodule
```

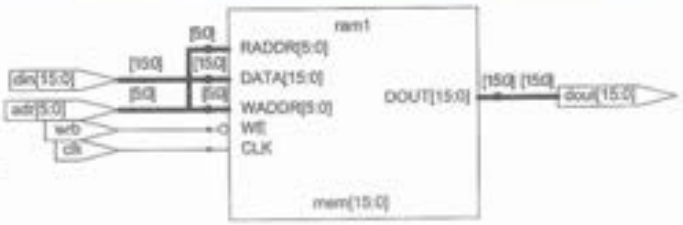


It is easy to accidentally imply sequential logic with `always` blocks when combinational logic is intended. The resulting bugs can be difficult to track down. Therefore, to imply combinational logic, it is safer to use `assign` statements than `always` blocks. Nevertheless, the convenience of constructs such as `if` or `case` that must appear in `always` blocks justifies the modeling style as long as you thoroughly understand what you are doing.

A.4.5 Memories

Verilog has an array construct used to describe memories. The following module describes a 64-word × 16-bit synchronous RAM that is written on the positive edge of the clock when `wrb` is low. The internal signal `mem` is declared as `reg` rather than `wire` because it is assigned inside an `always` block.

```
module ram(input        clk,
           input  [5:0] addr,
           input        wrb,
           input  [15:0] din,
           output [15:0] dout);

   reg  [15:0] mem[63:0]; // the memory

   always @(posedge clk)
      if (~wrb) mem[addr] <= din;

   assign dout = mem[addr];
endmodule
```



Synthesis tools are often restricted to generating gates from a library and produce poor memory arrays. A specialized memory generator is commonly used instead.

A.4.6 Blocking and Nonblocking Assignment

Verilog supports two types of assignments inside an `always` block. *Blocking assignments* use the = statement. *Nonblocking assignments* use the <= statement. Do not confuse either type with the `assign` statement, which cannot appear inside `always` blocks at all.

A group of blocking assignments inside a `begin/end` block are evaluated sequentially, just as you would expect in a standard programming language. A group of nonblocking assignments are evaluated in parallel; all of the statements are evaluated before any of the left sides are updated. This is what you would expect in hardware because real logic gates all operate independently rather than waiting for the completion of other gates.

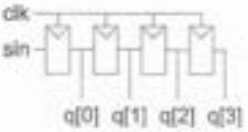
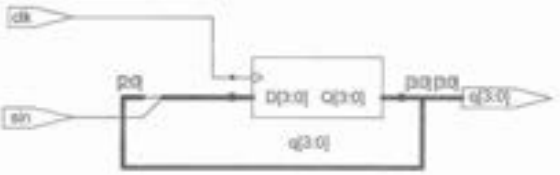


FIG A.2 Intended shift register

For example, consider two attempts to describe a shift register. On each clock edge, the data at `sin` should be shifted into the first flop, as shown in Figure A.2. The first flop shifts to the second flop. The data in the second flop shifts to the third flop, and so on until the last element drops off the end.

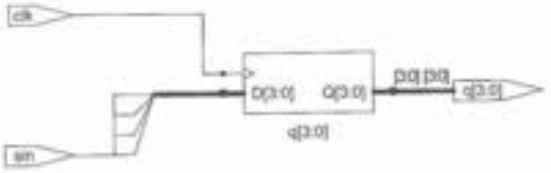
```
module shiftreg(input          clk,
                input          sin,
                output reg [3:0] q);

   // This is a correct implementation
   // using nonblocking assignment

   always @(posedge clk)
      begin
         q[0] <= sin; // nonblocking <=
         q[1] <= q[0];
         q[2] <= q[1];
         q[3] <= q[2];
         // even better to write
         // q <= {q[2:0], sin};
      end
endmodule
```



The nonblocking assignments mean that all of the values on the right sides are assigned simultaneously. Therefore, `q[1]` will get the original value of `q[0]`, not the value of `sin` that gets loaded into `q[0]`. This is what we would expect from real hardware. Of course, all of this could be written on one line for brevity.

Blocking assignments are more familiar from traditional programming languages, but they inaccurately model hardware. Consider the same module using blocking assignments. When `clk` rises, the Verilog says that `q[0]` should be copied from `sin`. Then `q[1]` should be copied from the new value of `q[0]` and so forth. All four registers immediately get the `sin` value.

```
module shiftreg(input          clk,
                input          sin,
                output reg [3:0] q);

   // This is a bad implementation
   // using blocking assignment

   always @(posedge clk)
      begin
         q[0] = sin; // blocking =
         q[1] = q[0];
         q[2] = q[1];
         q[3] = q[2];
      end
endmodule
```



The moral of this illustration is to use nonblocking assignments in `always` blocks when modeling sequential logic. Using sufficient ingenuity, such as reversing the orders of the four commands, you could make blocking assignments work correctly, but they offer no advantages and harbor great risks.

Finally, note that each `always` block implies a separate block of logic. Therefore, a given `reg` can be assigned in only one `always` block. Otherwise, two pieces of hardware with shorted outputs will be implied.

A.5 Finite State Machines

There are two styles of finite state machines. In *Mealy machines* (Figure A.3(a)), the output is a function of the current state and inputs. In *Moore machines* (Figure A.3(b)), the output is a function of only the current state.

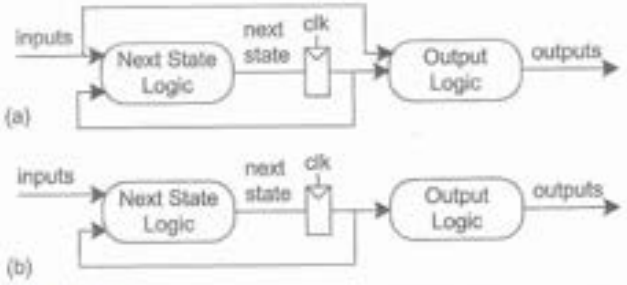


FIG A.3 Moore and Mealy machines

FSMs are modeled in Verilog with an `always` block defining the state registers and combinational logic defining the next state and output logic.

Let us first consider a simple finite state machine with one output and no inputs, a divide-by-3 counter. The output should be asserted every three clock cycles. The state transition diagram for a Moore machine is shown in Figure A.4. The output value is labeled in each state because the output is only a function of the state.

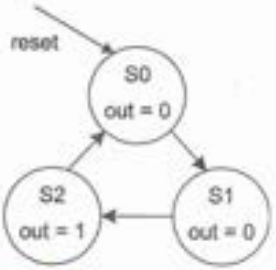
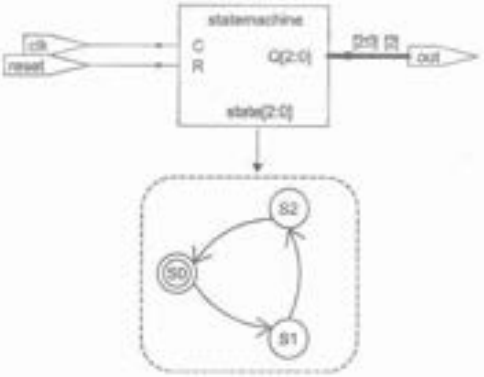


FIG A.4 Divide-by-3 counter state transition diagram

```
module divideby3FSM(input  clk,
                    input  reset,
                    output out);

   reg  [1:0] state, nextstate;

   parameter S0 = 2'b00;
   parameter S1 = 2'b01;
   parameter S2 = 2'b10;

   // State Register
   always @(posedge clk, posedge reset)
      if (reset) state <= S0;
      else       state <= nextstate;

   // Next State Logic
   always @(*)
      case (state)
         S0: nextstate <= S1;
         S1: nextstate <= S2;
         S2: nextstate <= S0;
         default: nextstate <= S0;
      endcase

   // Output Logic
   assign out = (state == S2);
endmodule
```



The FSM model is divided into three portions: the *state register*, *next state logic*, and *output logic*. The state register logic describes an asynchronously resettable register that resets to an initial state and otherwise advances to the computed next state. Defining states with parameters allows the easy modification of state encodings and makes the code easier to read. The next state logic computes the next state as a function of the current state and

inputs; in this example, there are no inputs. A `case` statement in an `always @(state or inputs)` block is a convenient way to define the next state. It is important to have a `default` if not all cases are enumerated; otherwise `nextstate` would not be assigned in the undefined cases. This implies that `nextstate` should keep its old value, which would require the existence of latches. Finally, the output logic may be a function of the current `state` alone in a Moore machine or of the current `state` and inputs in a Mealy machine. Depending on the complexity of the design, `assign` statements, `if` statements, or `case` statements may be most readable and efficient.

The next example shows a finite state machine with an input `a` and two outputs. Output `x` is true when the input is the same now as it was last cycle. Output `y` is true when the input is the same now as it was for the past two cycles. This is a Mealy machine because the output depends on the current inputs as well as the state. The outputs are labeled on each transition after the input. The state transition diagram is shown in Figure A.5.

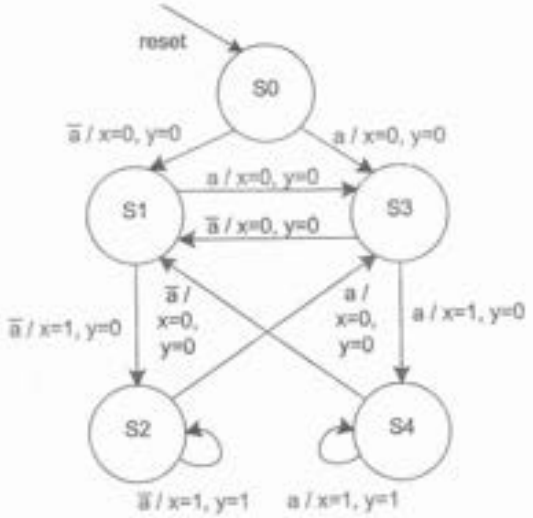
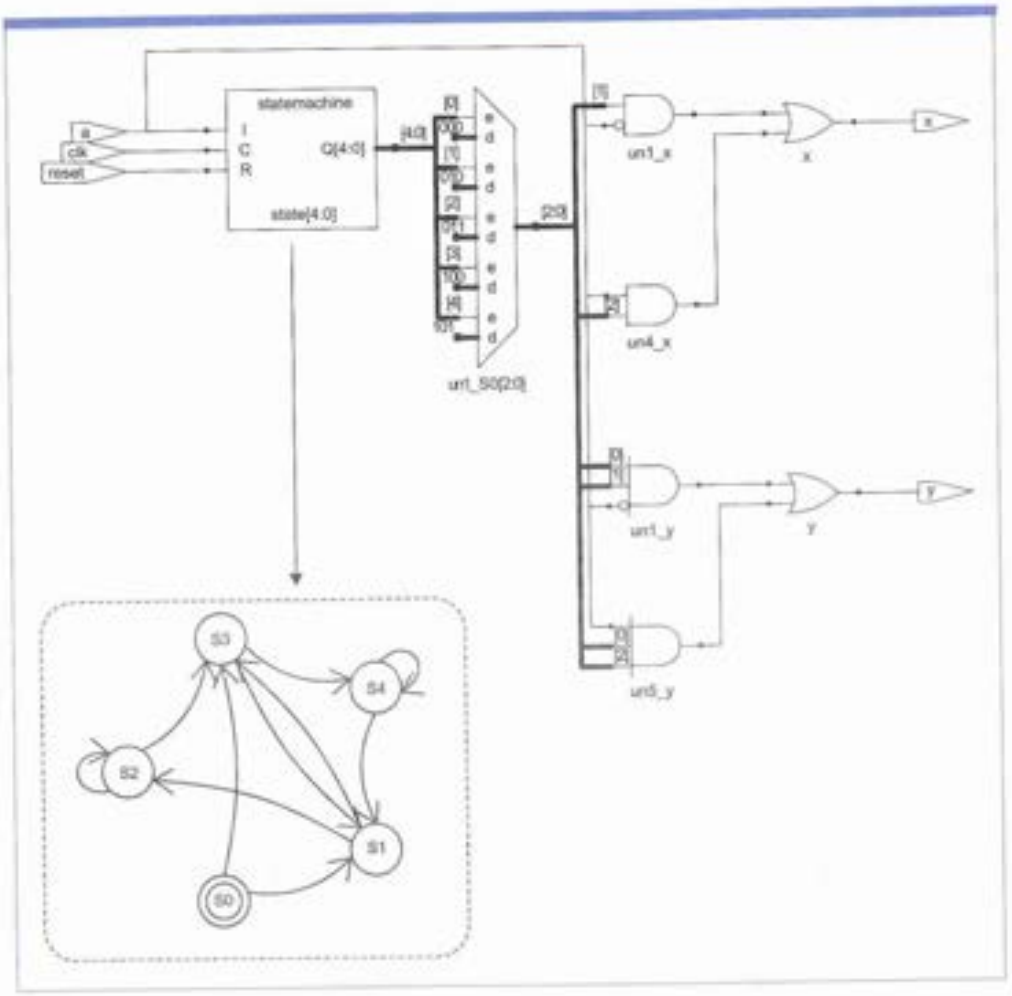


FIG A.5 History FSM state transition diagram

```
module historyFSM(input  clk,
                  input  reset,
                  input  a,
                  output x, y);

   reg  [2:0] state, nextstate;

   parameter S0 = 3'b000;
   parameter S1 = 3'b010;
   parameter S2 = 3'b011;
   parameter S3 = 3'b100;
   parameter S4 = 3'b101;

   // State Register
   always @(posedge clk, posedge reset)
      if (reset) state <= S0;
      else       state <= nextstate;

   // Next State Logic
   always @(*)
      case (state)
         S0: if (a) nextstate <= S3;
             else   nextstate <= S1;
         S1: if (a) nextstate <= S3;
             else   nextstate <= S2;
         S2: if (a) nextstate <= S3;
             else   nextstate <= S2;
         S3: if (a) nextstate <= S4;
             else   nextstate <= S1;
         S4: if (a) nextstate <= S4;
             else   nextstate <= S1;
         default: nextstate <= S0;
      endcase

   // Output Logic
   assign x = (state[1] & ~a) |
              (state[2] & a);
   assign y = (state[1] & state[0] & ~a) |
              (state[2] & state[0] & a);
endmodule
```

continued



The output logic equations depend on the specific state encoding and were worked out by hand. A more general approach is independent of the encodings and requires less thinking, but more code:

```
// Output Logic
always @(state or a)
   case (state)
      S0:  begin
              x <= 0; y <= 0;
           end
      S1:  if (a) begin
              x <= 0; y <= 0;
           end else begin
              x <= 1; y <= 0;
           end
      S2:  if (a) begin
              x <= 0; y <= 0;
           end else begin
              x <= 1; y <= 1;
           end
      S3:  if (a) begin
              x <= 1; y <= 0;
           end else begin
              x <= 0; y <= 0;
           end
      S4:  if (a) begin
              x <= 1; y <= 1;
           end else begin
              x <= 0; y <= 0;
           end
   endcase
```

You may be tempted so simplify the `case` statement. For example, case `S4` could be reduced to:

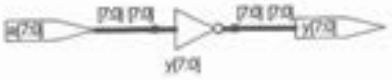
```
// bad simplification of S4
S4:   if (a) begin
         y <= 1;
      end else begin
         x <= 0; y <= 0;
      end
```

The designer reasons that to get to state `S4`, we must have passed through state `S3` with a high, setting x high. Therefore, the assignment of x is optimized out of `S4` when a is high. This is incorrect reasoning. The modified approach implies sequential logic. Specifically, a latch is implied that holds the old value of x when x is not assigned. The latch holds its output under a peculiar set of circumstances; a and the state must be used to compute the latch clock signal. This is undoubtedly not what you wanted, but was easy to inadvertently imply. The moral of this example is that if any signal gets assigned in any branch of an `if` or `case` statement, it must be assigned in all branches lest a latch be implied.

A.6 Parameterized Modules

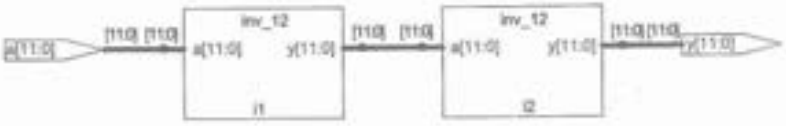
So far, all of our modules have had fixed-width inputs and outputs. Thus, we have needed separate modules for 4- and 8-bit wide 2-input multiplexers. Verilog permits variable bit widths using *parameterized modules*. For example, we can declare a parameterized bank of inverters with a default of 8 gates as:

```
module inv
   #(parameter width = 8)
    (input  [width-1:0] a,
     output [width-1:0] y);

   assign y = ~a;
endmodule
```



We can adjust the parameter when we instantiate the block. For example, we can build a bank of 12 buffers using a pair of banks of 12 inverters.

```
module buffer
   #(parameter numbits = 12)
    (input  [numbits-1:0] a,
     output [numbits-1:0] y);

   wire [numbits-1:0] x;

   inv #(numbits) i1(a, x);
   inv #(numbits) i2(x, y);
endmodule
```



A.7 Structural Primitives

When coding at the structural level, primitives exist for basic logic gates and transistors. Examples for a full adder carry circuit (majority gate) were given in Section 1.8.4.

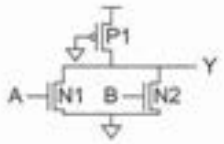
Gate primitives include `not`, `and`, `or`, `xor`, `nand`, `nor`, and `xnor`. The output is declared first; multiple inputs may follow. For example, a 4-input AND gate may be given as

```
and g1(y, a, b, c, d);
```

Transistor primitives include `tranif1`, `tranif0`, `rtranif1`, and `rtranif0`. `tranif1` is an nMOS transistor that turns ON when the gate is '1' while `tranif0` is a pMOS transistor. The `rtranif` primitives are resistive transistors, i.e., weak transistors that can be overcome by a stronger driver. For example, a pseudo-nMOS NOR gate with a weak pull-up

is modeled with three transistors. Most synthesis tools map only onto gates, not transistors, so these transistor primitives are only for simulation.

```
module nor2(input  a, b,
            output y);

   tranif1  n1(y, gnd, a);
   tranif1  n2(y, gnd, b);
   rtranif0 p1(y, vdd, gnd);
endmodule
```



The `tranif` devices are bi-directional; that is, the source and drain are symmetric. Verilog also supports unidirectional `nmos` and `pmos` primitives that only allow a signal to flow from the input terminal to the output terminal. Real transistors are inherently bi-directional, so unidirectional models can result in simulation not catching bugs that would exist in real hardware. Therefore, `tranif` primitives are preferred for simulation.

A.8 Test Benches

Verilog models are tested through simulation. For small designs, it may be practical to manually apply inputs to a simulator and visually check for the correct outputs. For larger designs, this procedure is usually automated with a *test bench*.

The following code shows an adder and its associated test bench. The test bench uses nonsynthesizable system calls to read a file, apply the *test vectors* to the *device under test* (DUT), check the results, and report any discrepancies. The `initial` statement defines a block that is executed only on startup of simulation. In it, the `$readmemh` reads a file (in hexadecimal form) into an array in memory. The `testvector.tv` example file shows four test vectors, each consisting of two 32-bit inputs and an expected 32-bit output. As the array is much larger than the number of test vectors, the remaining entries are filled with x's. The next `always` block defines a clock that repeats forever, being low for 50 units of time, then high for 50. On each positive edge of the clock, the next test vector is applied to the inputs and the expected output is saved. The actual output is sampled on the negative edge of the clock to permit some time for it to settle. It is not uncommon for bad logic to generate `z` and `x` values. The `!=` and `==` comparison operations return `x` if either argument has `z` or `x`, so they are not reliable. The `!==` and `===` commands check for an exact match, including `z` or `x` values. If there is a mismatch, the `$display` command is used to print the discrepancy. Each time `vectornum` is incremented, the test bench checks to see if the test is complete. It terminates testing with the `$finish` command after 100 vectors have been applied or the test vector consists of x's; for our `testvector.tv` file, it will end after the four vectors are applied.

```
module adder(a, b, y);
    input      [31:0]     a, b;
    output     [31:0]     y;

    assign y = a + b;
endmodule

module testbench();
    reg     [31:0]  testvectors[1000:0];
    reg             clk;
    reg     [10:0]  vectornum, errors;
    reg     [31:0]  a, b, expectedy;
    wire    [31:0]  y;

    // instantiate device under test
    adder   dut(a, b, y);

    // read the test vector file and initialize test
    initial
        begin
            $readmemh("testvectors.tv", testvectors);
            vectornum = 0; errors = 0;
        end

    // generate a clock to sequence tests
    always
        begin
            clk = 0; #50; clk = 1; #50;
        end

    // on each clock step, apply next test
    always @(posedge clk)
        begin
            a = testvectors[vectornum*3];
            b = testvectors[vectornum*3 + 1];
            expectedy = testvectors[vectornum*3 + 2];
        end

    // then check for correct results
    always @(negedge clk)
        begin
            vectornum = vectornum + 1;
            if (y !== expectedy) begin
                $display("Inputs were %h, %h", a, b);
                $display("Expected %h but actual %h", expectedy, y);
                errors = errors + 1;
            end
        end
```

```
    // halt at the end of file
    always @(vectornum)
        begin
            if (vectornum == 100 || testvectors[vectornum*3] === 32'bx)
                begin
                    $display("Completed %d tests with %d errors.",
                                vectornum, errors);
                    $finish;
                end
        end
endmodule
```

testvectors.tv file:

```
00000000
00000000
00000000

00000001
00000000
00000001

ffffffff
00000003
00000002

12345678
12345678
2468acf0
```

A.9 Pitfalls

This section includes a set of style guidelines and examples of a number of bad circuits produced by common Verilog coding errors. The examples include the warnings given by Synopsys Design Compiler and Synplify Pro synthesis tools.

A.9.1 Verilog Style Guidelines

1. Use only nonblocking assignments inside `always` blocks.
2. Define your combinational logic using `assign` statements when practical. Only use `always` blocks to define combinational logic if constructs like `if` or `case` make your logic much clearer or more compact.
3. When modeling combinational logic with an `always` block, if a signal is assigned in any branch of an `if` or `case` statement, it must be assigned in all branches.

4. Include `default` cases in your `case` statements.
5. Partition your design into leaf cells and non-leaf cells. Leaf cells contain behavioral code (`assign` statements or `always` blocks), but do not instantiate other cells. Non-leaf cells contain structural code (i.e., they instantiate other cells, but contain no logic). Minor exceptions to this guideline can be made to keep the code readable.
6. Use parameters to define state names and constants.
7. Properly indent your code, as shown in the examples in this guide.
8. Use comments liberally.
9. Use meaningful signal names. Use `a, b, c, ...` for generic logic gate inputs. Use `x, y, z` for generic combinational outputs and `q` for a generic state element output. Use descriptive names for nongeneric cells. Do not use `foo`, `bar`, or `baz`!
10. Be consistent in your use of capitalization and underscores.
11. Do not ignore synthesis warnings unless you understand what they mean.

The clock, registers, and latches are common places to introduce bugs. Many FPGA and ASIC design flows use a conservative methodology of supplying a single clock to edge-triggered, asynchronously resettable registers with clock enables. While this will sacrifice performance and cost extra area, it reduces both the number of errors a designer can make and the expense of debugging these errors. The following guidelines are common to such a style.

1. Use only positive edge-triggered registers. Avoid `@(negedge clk)`, SR latches, and transparent latches.
2. Be certain not to inadvertently imply latches. Check synthesis reports for warnings such as

   (Synopsys) *Warning: Latch inferred in design '...' read with 'hdlin_check_no_latch'.*
   (Synplicity) *@W: ...Latch generated from always block for signal ...*
3. Provide an asynchronous reset to all of your registers with a common signal name.
4. Provide a common clock to all of your registers whenever possible. Avoid gated clocks, which may lead to extra clock skew and hold time failures. Use a clock enable instead.
5. If you get any "Bus Conflict" messages or `x`'s in your simulation, be sure to find their cause and fix the problem.

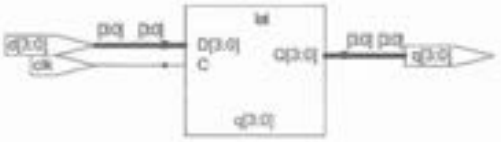
A.9.2 Incorrect Stimulus List

The following circuit was intended to be a transparent latch, but the `d` input was omitted from the stimulus list. When synthesized, it still produces a transparent latch, but with a warning. If the Verilog is simulated, `q` will change on the rising edge of `clk` but not on a change in `d` while `clk` is stable high. Thus, the circuit will simulate as if it were a flip-flop.

Inconsistencies between simulation and synthesis are a common reason for chips to fail, so correcting these warnings is vital.

(Synopsys) *Warning: Variable 'd' is being read*
in routine notquitealatch line 5 in file 'notquitealatch.v',
but does not occur in the timing control of the block which begins there. (HDL-180)
(Synplicity) *@W: notquitealatch.v(5): Incomplete sensitivity list - assuming completeness*
@W:" notquitealatch.v":5:12:5:15
@W: notquitealatch.v(6): Referenced variable d is not in sensitivity list
@W:"c: notquitealatch.v":6:20:6:21

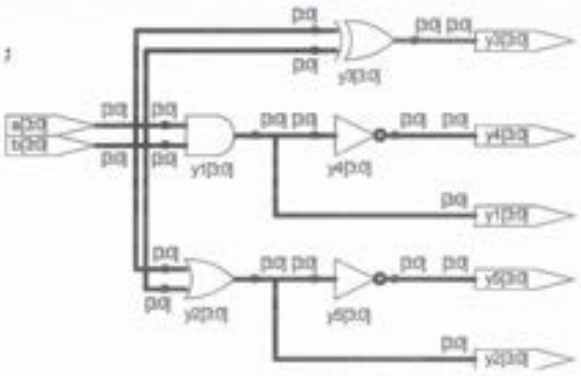
```
module notquitealatch(input            clk,
                      input      [3:0] d,
                      output reg [3:0] q);

   always @(clk) // left out 'or d'
      if (clk) q <= d;
endmodule
```



Similarly, the `b` input in the following combinational logic was omitted from the stimulus list of the `always` block. Synthesis tools do generate the intended gates, but give another warning.

(Synopsys) *Warning: Variable 'b' is being read*
in routine gates line 4 in file 'gates.v',
but does not occur in the timing control of the block which begins
there. (HDL-180)
(Synplicity) *@W: gates_bad.v(4): Incomplete sensitivity list - assuming completeness*
@W:" gates_bad.v":4:12:4:13
@W: gates_bad.v(6): Referenced variable b is not in sensitivity list
@W:"c: gates_bad.v":6:19:6:20

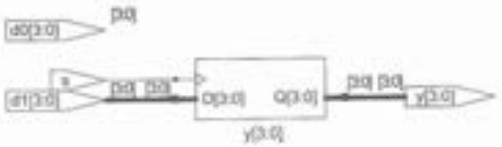
```
module gates(input      [3:0] a, b,
             output reg [3:0] y1, y2, y3, y4, y5);

   always @(a) // missing 'or b'
      begin
         y1 <= a & b; // AND
         y2 <= a | b; // OR
         y3 <= a ^ b; // XOR
         y4 <= ~(a & b); // NAND
         y5 <= ~(a | b); // NOR
      end
endmodule
```



The next example is supposed to model a multiplexer, but the author incorrectly wrote `@(posedge s)` rather than `@(s)`. Because `s` is by definition high immediately after its positive edge, this circuit actually behaves as a flip-flop that captures `d1` on the rising edge of `clk` and ignores `d0`. Some tools will synthesize the flip-flop, while others will just produce an error message.

(Synopysis) *Error: clock variable s is being used as data. (HDL-175)*
(Synplicity) *@W: badmux.v(1): Input d0 is unused @W:"badmux.v":1:31:1:33*

```
module badmux(input       [3:0] d0, d1,
              input             s,
              output reg [3:0] y);

   always @(posedge s)
      if (s) y <= d1;
      else   y <= d0;

endmodule
```



A.9.3 Missing begin/end Block

In the following example, two variables are supposed to be assigned in the `always` block. The `begin/end` block is missing. This is a syntax error.

(Synopsys) *Error: syntax error at or near token '{' (File: notquiteatwobitflop.v Line: 7) (VE-0)*
Error: Can't read 'verilog' file 'J:/Classes/E155/Fall2000/synopsys/flop2.v'. (UID-59)
(Synplicity) *@E: notquiteatwobitflop.v(7): Expecting ; @E:"notquiteatwobitflop.v":7:9:7:9*

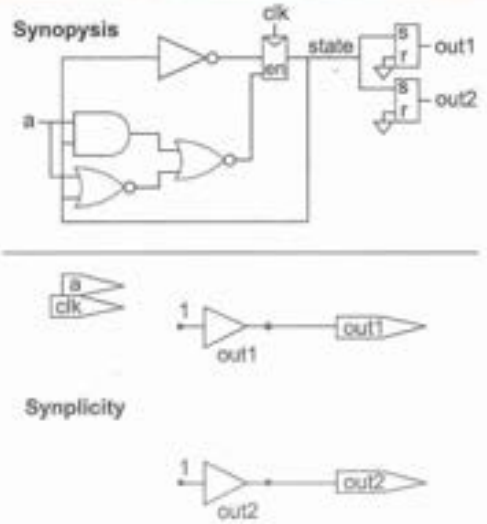
```
module notquiteatwobitflop(input          clk,
                           input      [1:0] d,
                           output reg [1:0] q);

   always @(posedge clk)
      q[1] = d[1];
      q[0] = d[0];
endmodule
```

A.9.4 Undefined Outputs

In the next example of a finite state machine, the user intended `out1` to be high when the state is 0 and `out2` to be high when the state is 1. However, the code never sets the outputs low. Synopsys produces a circuit with an SR latch and a transparent latch that can set the

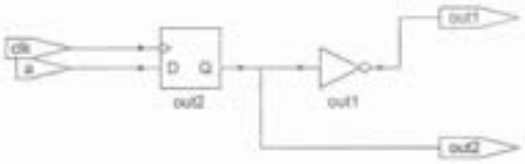
output high, but never resets the output low. Synplicity produces a circuit in which both outputs are hardwired to 1.

```
module FSMbad(input      clk,
              input      a,
              output reg out1, out2);

   reg  state;

   always @(posedge clk)
      if (state == 0) begin
         if (a) state <= 1;
      end else begin
         if (~a) state <= 0;
      end

   always @(*)
      if (state == 0) out1 <= 1;
      else            out2 <= 1;
      // neglect to set out1/out2 to 0
endmodule
```



A corrected version of the code produces the desired state machine.

```
module FSMgood(input      clk,
               input      a,
               output reg out1, out2);

   reg  state;

   always @(posedge clk)
      if (state == 0) begin
         if (a) state <= 1;
      end else begin
         if (~a) state <= 0;
      end

   always @(*)
      if (state == 0) begin
         out1 <= 1;
         out2 <= 0;
      end else begin
         out2 <= 1;
         out1 <= 0;
      end
endmodule
```



A.9.5 Incomplete Specification of Cases

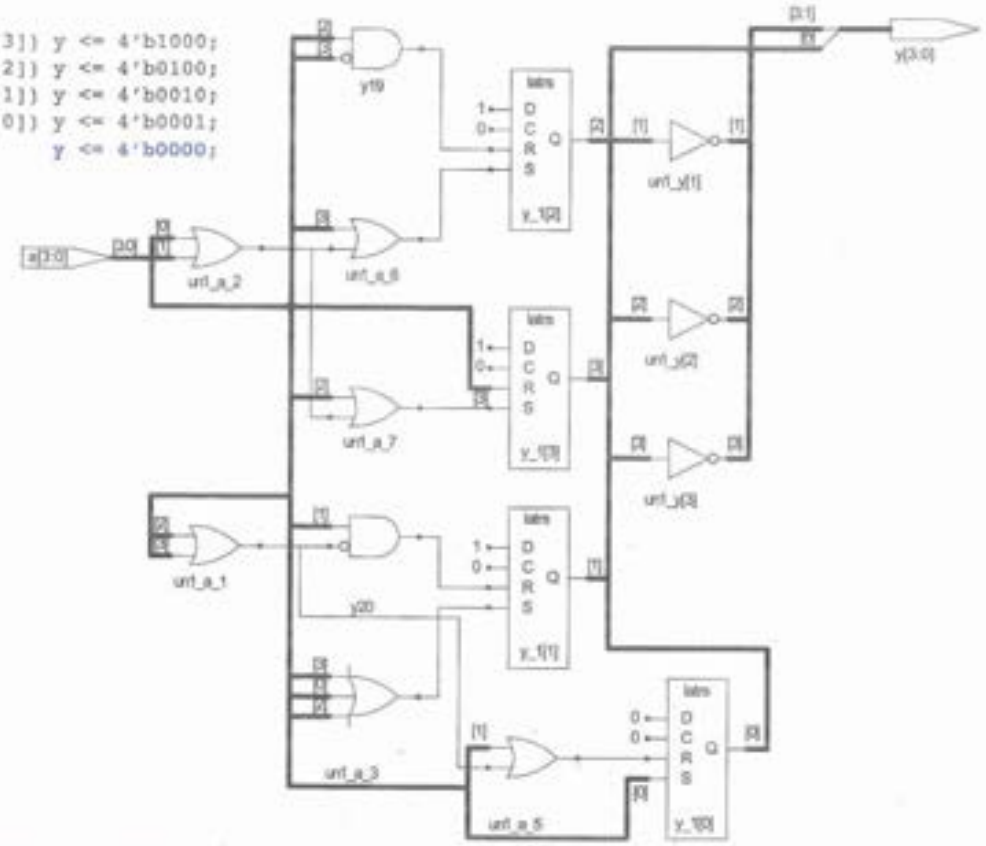
The next examples show an incomplete specification of input possibilities. The priority encoder fails to check for the possibility of no true inputs. It therefore incorrectly implies latches to hold the previous output when all four inputs are false. Synplicity produces a bizarre circuit with SR latches.

The synthesis tool will warn that a latch or memory device is implied. The astute designer will detect the problem by knowing that a priority encoder should be a combinational circuit and therefore have no memory devices.

(Synopsys) *Inferred memory devices in process in routine priority_always line 4 in file 'priority_always_bad.v'.*

(Synplicity) *@W: priority_always_bad.v(5): Latch generated from always block for signal y[3:0], probably caused by a missing assignment in an if or case stmt @W:"priority_always_bad.v":5:6:5:8*

```
module priority_always(input       [3:0] a,
                       output reg [3:0] y);

   always @(*)
      if      (a[3]) y <= 4'b1000;
      else if (a[2]) y <= 4'b0100;
      else if (a[1]) y <= 4'b0010;
      else if (a[0]) y <= 4'b0001;
      // else        y <= 4'b0000;
endmodule
```



The next example of a 7-segment display decoder shows the same type of problem in a case statement.

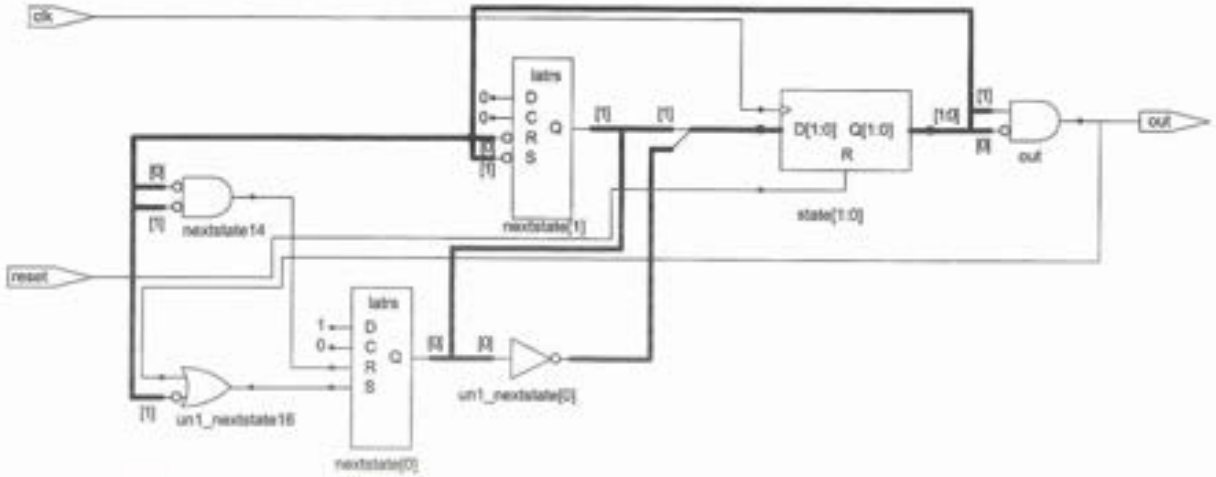
```
module seven_seg_display_decoder(input      [3:0] data,
                                 output reg [6:0] segments);

   always @(*)
      case (data)
         0: segments <= 7'b000_0000; // ZERO
         1: segments <= 7'b111_1110; // ONE
         2: segments <= 7'b011_0000; // TWO
         3: segments <= 7'b110_1101; // THREE
         4: segments <= 7'b011_0011; // FOUR
         5: segments <= 7'b101_1011; // FIVE
         6: segments <= 7'b101_1111; // SIX
         7: segments <= 7'b111_0000; // SEVEN
         8: segments <= 7'b111_1111; // EIGHT
         9: segments <= 7'b111_1011; // NINE
         // default: segments <= 7'b000_0000;
      endcase
endmodule
```

Similarly, it is a common mistake to forget the `default` in the next state or output logic of an FSM.

```
module divideby3FSM(input  clk,
                    input  reset,
                    output out);

   reg  [1:0] state, nextstate;

   parameter S0 = 2'b00;
   parameter S1 = 2'b01;
   parameter S2 = 2'b10;

   // State Register
   always @(posedge clk, posedge reset)
      if (reset) state <= S0;
      else       state <= nextstate;

   // Next State Logic
   always @(state)
      case (state)
         S0: nextstate <= S1;
         S1: nextstate <= S2;
         S2: nextstate <= S0;
         //default: nextstate <= S0;
      endcase
```

continued

```
    // Output Logic
    assign out = (state == S2);
endmodule
```



A.9.6 Shorted Outputs

Bad code can sometimes lead to shorted outputs of gates. For example, the tristate drivers in the following multiplexer should have mutually exclusive enable signals, but instead are both active simultaneously and produce a conflict when `d0` and `d1` are not equal.

Synthesis may not report any errors. However, during simulation, you will observe x's rather than 0's or 1's when the bus is simultaneously being driven high and low. You may also get a "Bus Conflict" warning message.

```
module mux2(input  [3:0] d0, d1,
            input        s,
            output [3:0] y);

   tristate t0(d0, s, y); // wanted ~s
   tristate t1(d1, s, y);
endmodule
```

Another cause of shorted outputs is when a `reg` is assigned in two different `always` blocks. For example, the following code tries to model a register with asynchronous reset and asynchronous set. The first `always` block models the reset and ordinary operation. The second `always` block attempts to incorporate the asynchronous set. Synthesis infers a separate piece of hardware for each `always` block, with a shorted output and may report an error.

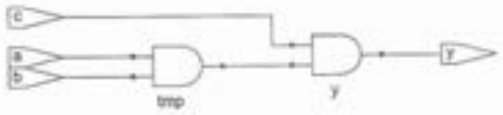
(Synopsys) *Error: the net '/over1/q' has more than one driver*
(Synplicity) *@E: floprs_bad.v(5): Only one always block may assign a given variable q[3:0]*
@E:"floprs_bad.v":5:33:5:34

```
module floprsen(input          clk,
                input          reset,
                input          set,
                input      [3:0] d,
                output reg [3:0] q);

   // bad asynchronous set and reset

   always @(posedge clk, posedge reset)
      if (reset) q <= 0;
      else q <= d;

   always @(set)
      if (set) q <= 1;
endmodule
```

A.9.7 Incorrect Use of Nonblocking Assignments

Section A.4.6 recommended using nonblocking assignments in all `always` blocks. This can occasionally get you in trouble, as shown in the following 3-input AND gate.

```
module and3bad(a, b, c, y);
   input          a, b, c;
   output          y;

   reg             y;
   reg             tmp;

   always @(a, b, c)
   begin
      tmp <= a & b;
      y <= tmp & c;
   end
endmodule
```



In this example, suppose that initially, `a` = 0 and `b` = `c` = 1. Therefore, `tmp` initially is 0. When `a` rises, the always block is triggered. `tmp` is given the value 1, but `y` is in parallel given the value 0 because the new value of `tmp` has not yet been written. The code can synthesize correctly, but simulate incorrectly without warning. The problem can be avoided by using always `@(*)` to describe combinational logic.

A.10 Example: MIPS Processor

To illustrate a nontrivial Verilog design, this section lists the Verilog code and test bench for the MIPS processor subset discussed in Chapter 1. The example handles only the `LB`, `SB`, `ADD`, `SUB`, `AND`, `OR`, `SLT`, `BEQ`, and `J` instructions. It uses an 8-bit datapath and only eight registers. Because the instruction is 32 bits wide, it is loaded in four successive fetch cycles across an 8-bit path to external memory.

The test bench initializes a 512-byte memory with instructions and data from a text file. The code exercises each of the instructions. The `mipstest.asm` assembly language file and `memfile.dat` text file are shown below. The test bench runs until it observes a memory write. If the value 7 is written to address 5, the code probably executed correctly. If all goes well, the test bench should take 100 cycles (1000 ns) to run.

```
# mipstest.asm
# 9/16/03 David Harris David_Harris@hmc.edu
#
# Test MIPS instructions.  Assumes memory was
# initialized as:
# word 16: 3 — be careful of endianness
# word 17: 5
# word 18: 12

main:     #Assembly Code          effectMachine Code
          lb $2, 68($0)           # initialize $2 = 580020044
          lb $7, 64($0)           # initialize $7 = 380070040
          lb $3, 69($7)           # initialize $3 = 1280e30045
          or $4, $7, $2           # $4 <= 3 or 5 = 700e22025
          and $5, $3, $4          # $5 <= 12 and 7 = 400642824
          add $5, $5, $4          # $5 <= 4 + 7 = 1100a42820
          beq $5, $7, end         # shouldn't be taken10a70008
          slt $6, $3, $4          # $6 <= 12 < 7 = 00064302a
          beq $6, $0, around      # should be taken10c00001
          lb $5, 0($0)            # shouldn't happen80050000
around:   slt $6, $7, $2          # $6 <= 3 < 5 = 100e2302a
          add $7, $6, $5          # $7 <= 1 + 11 = 1200c53820
          sub $7, $7, $2          # $7 <= 12 - 5 = 700e23822
          j end                   # should be taken0800000f
          lb $7, 0($0)            # shouldn't happen80070000
end:      sb $7, 0($2)            # write adr 5 <= 7a0470000
```

```
memfile.dat
80020044
80070040
80e30045
00e22025
00642824
00a42820
10a70008
0064302a
10c00001
80050000
00e2302a
00c53820
00e23822
0800000f
80070000
a0470000
03000000
05000000
0c000000
```

```
//-----------------------------
// mips.v
// Max Yi (byyi@hmc.edu) and David_Harris@hmc.edu 12/9/03
// Model of subset of MIPS processor described in Ch 1
//-----------------------------

// top level design for testing
module top #(parameter WIDTH = 8, REGBITS = 3)();

   reg                 clk;
   reg                 reset;
   wire                memread, memwrite;
   wire    [WIDTH-1:0] adr, writedata;
   wire    [WIDTH-1:0] memdata;

   // instantiate devices to be tested
   mips #(WIDTH,REGBITS) dut(clk, reset, memdata, memread, memwrite, adr, writedata);

   // external memory for code and data
   exmemory #(WIDTH) exmem(clk, memwrite, adr, writedata, memdata);

   // initialize test
   initial
      begin
         reset <= 1; # 22; reset <= 0;
      end

   // generate clock to sequence tests
   always
      begin
         clk <= 1; # 5; clk <= 0; # 5;
      end

   always@(negedge clk)
      begin
         if(memwrite)
            if(adr == 5 & writedata == 7)
               $display("Simulation completely successful");
            else $display("Simulation failed");
      end
endmodule
```

```
// external memory accessed by MIPS
module exmemory #(parameter WIDTH = 8)
                 (input                  clk,
                  input                  memwrite,
                  input      [WIDTH-1:0] adr, writedata,
                  output reg [WIDTH-1:0] memdata);

   reg  [31:0] RAM [(1<<WIDTH-2)-1:0];
   wire [31:0] word;

   initial
      begin
         $readmemh("memfile.dat",RAM);
      end

   // read and write bytes from 32-bit word
   always @(posedge clk)
      if(memwrite)
         case (adr[1:0])
            2'b00: RAM[adr>>2][7:0] <= writedata;
            2'b01: RAM[adr>>2][15:8] <= writedata;
            2'b10: RAM[adr>>2][23:16] <= writedata;
            2'b11: RAM[adr>>2][31:24] <= writedata;
         endcase

   assign word = RAM[adr>>2];
   always @(*)
      case (adr[1:0])
         2'b00: memdata <= word[31:24];
         2'b01: memdata <= word[23:16];
         2'b10: memdata <= word[15:8];
         2'b11: memdata <= word[7:0];
      endcase
endmodule

// simplified MIPS processor
module mips #(parameter WIDTH = 8, REGBITS = 3)
             (input              clk, reset,
              input  [WIDTH-1:0] memdata,
              output             memread, memwrite,
              output [WIDTH-1:0] adr, writedata);

   wire [31:0] instr;
   wire        zero, alusrca, memtoreg, iord, pcen, regwrite, regdst;
   wire [1:0]  aluop,pcsource,alusrcb;
   wire [3:0]  irwrite;
   wire [2:0]  alucont;

   controller  cont(clk, reset, instr[31:26], zero, memread, memwrite,
                    alusrca, memtoreg, iord, pcen, regwrite, regdst,
                    pcsource, alusrcb, aluop, irwrite);
   alucontrol  ac(aluop, instr[5:0], alucont);
   datapath    #(WIDTH, REGBITS)
               dp(clk, reset, memdata, alusrca, memtoreg, iord, pcen,
                  regwrite, regdst, pcsource, alusrcb, irwrite, alucont,
                  zero, instr, adr, writedata);
endmodule

module controller(input clk, reset,
                  input      [5:0] op,
                  input            zero,
                  output reg       memread, memwrite, alusrca, memtoreg, iord,
                  output           pcen,
                  output reg       regwrite, regdst,
                  output reg [1:0] pcsource, alusrcb, aluop,
                  output reg [3:0] irwrite);
```

```
parameter   FETCH1  =  4'b0001;
parameter   FETCH2  =  4'b0010;
parameter   FETCH3  =  4'b0011;
parameter   FETCH4  =  4'b0100;
parameter   DECODE  =  4'b0101;
parameter   MEMADR  =  4'b0110;
parameter   LBRD    =  4'b0111;
parameter   LBWR    =  4'b1000;
parameter   SBWR    =  4'b1001;
parameter   RTYPEEX =  4'b1010;
parameter   RTYPEWR =  4'b1011;
parameter   BEQEX   =  4'b1100;
parameter   JEX     =  4'b1101;

parameter   LB      =  6'b100000;
parameter   SB      =  6'b101000;
parameter   RTYPE   =  6'b0;
parameter   BEQ     =  6'b000100;
parameter   J       =  6'b000010;

reg [3:0] state, nextstate;
reg       pcwrite, pcwritecond;

// state register
always @(posedge clk)
   if(reset) state <= FETCH1;
   else state <= nextstate;

// next state logic
always @(*)
   begin
      case(state)
         FETCH1:  nextstate <= FETCH2;
         FETCH2:  nextstate <= FETCH3;
         FETCH3:  nextstate <= FETCH4;
         FETCH4:  nextstate <= DECODE;
         DECODE:  case(op)
                     LB:      nextstate <= MEMADR;
                     SB:      nextstate <= MEMADR;
                     RTYPE:   nextstate <= RTYPEEX;
                     BEQ:     nextstate <= BEQEX;
                     J:       nextstate <= JEX;
                     default: nextstate <= FETCH1; // should never happen
                  endcase
         MEMADR:  case(op)
                     LB:      nextstate <= LBRD;
                     SB:      nextstate <= SBWR;
                     default: nextstate <= FETCH1; // should never happen
                  endcase
         LBRD:    nextstate <= LBWR;
         LBWR:    nextstate <= FETCH1;
         SBWR:    nextstate <= FETCH1;
         RTYPEEX: nextstate <= RTYPEWR;
         RTYPEWR: nextstate <= FETCH1;
         BEQEX:   nextstate <= FETCH1;
         JEX:     nextstate <= FETCH1;
         default: nextstate <= FETCH1; // should never happen
      endcase
   end

always @(*)
   begin
   // set all outputs to zero, then conditionally assert just the appropriate ones
         irwrite <= 4'b0000;
         pcwrite <= 0; pcwritecond <= 0;
         regwrite <= 0; regdst <= 0;
         memread <= 0; memwrite <= 0;
         alusrca <= 0; alusrcb <= 2'b00; aluop <= 2'b00;
         pcsource <= 2'b00;
         iord <= 0; memtoreg <= 0;
```

```
case(state)
   FETCH1:
      begin
         memread <= 1;
         irwrite <= 4'b1000;
         alusrcb <= 2'b01;
         pcwrite <= 1;
      end
   FETCH2:
      begin
         memread <= 1;
         irwrite <= 4'b0100;
         alusrcb <= 2'b01;
         pcwrite <= 1;
      end
   FETCH3:
      begin
         memread <= 1;
         irwrite <= 4'b0010;
         alusrcb <= 2'b01;
         pcwrite <= 1;
      end
   FETCH4:
      begin
         memread <= 1;
         irwrite <= 4'b0001;
         alusrcb <= 2'b01;
         pcwrite <= 1;
      end
   DECODE: alusrcb <= 2'b11;
   MEMADR:
      begin
         alusrca <= 1;
         alusrcb <= 2'b10;
      end
   LBRD:
      begin
         memread <= 1;
         iord    <= 1;
      end
   LBWR:
      begin
         regwrite <= 1;
         memtoreg <= 1;
      end
   SBWR:
      begin
         memwrite <= 1;
         iord     <= 1;
      end
   RTYPEEX:
      begin
         alusrca <= 1;
         aluop   <= 2'b10;
      end
   RTYPEWR:
      begin
         regdst   <= 1;
         regwrite <= 1;
      end
   BEQEX:
      begin
         alusrca     <= 1;
         aluop       <= 2'b01;
         pcwritecond <= 1;
         pcsource    <= 2'b01;
      end
   JEX:
      begin
         pcwrite  <= 1;
         pcsource <= 2'b10;
      end
```

```
        endcase
      end
   assign pcen = pcwrite | (pcwritecond & zero); // program counter enable
endmodule

module alucontrol(input      [1:0] aluop,
                  input      [5:0] funct,
                  output reg [2:0] alucont);

   always @(*)
      case(aluop)
         2'b00: alucont <= 3'b010;  // add for lb/sb/addi
         2'b01: alucont <= 3'b110;  // sub (for beq)
         default: case(funct)       // R-Type instructions
                     6'b100000: alucont <= 3'b010; // add (for add)
                     6'b100010: alucont <= 3'b110; // subtract (for sub)
                     6'b100100: alucont <= 3'b000; // logical and (for and)
                     6'b100101: alucont <= 3'b001; // logical or (for or)
                     6'b101010: alucont <= 3'b111; // set on less (for slt)
                     default:   alucont <= 3'b101; // should never happen
                  endcase
      endcase
endmodule

module datapath #(parameter WIDTH = 8, REGBITS = 3)
                 (input                clk, reset,
                  input  [WIDTH-1:0] memdata,
                  input                alusrca, memtoreg, iord, pcen, regwrite, regdst,
                  input  [1:0]         pcsource, alusrcb,
                  input  [3:0]         irwrite,
                  input  [2:0]         alucont,
                  output               zero,
                  output [31:0]        instr,
                  output [WIDTH-1:0] adr, writedata);

   // the size of the parameters must be changed to match the WIDTH parameter
   parameter CONST_ZERO = 8'b0;
   parameter CONST_ONE =  8'b1;

   wire [REGBITS-1:0] ra1, ra2, wa;
   wire [WIDTH-1:0]   pc, nextpc, md, rd1, rd2, wd, a, src1, src2, aluresult,
                      aluout, constx4;

   // shift left constant field by 2
   assign constx4 = {instr[WIDTH-3:0],2'b00};

   // register file address fields
   assign ra1 = instr[REGBITS+20:21];
   assign ra2 = instr[REGBITS+15:16];
   mux2       #(REGBITS) regmux(instr[REGBITS+15:16], instr[REGBITS+10:11], regdst, wa);

   // independent of bit width, load instruction into four 8-bit registers over four cycles
   flopen     #(8)       ir0(clk, irwrite[0], memdata[7:0], instr[7:0]);
   flopen     #(8)       ir1(clk, irwrite[1], memdata[7:0], instr[15:8]);
   flopen     #(8)       ir2(clk, irwrite[2], memdata[7:0], instr[23:16]);
   flopen     #(8)       ir3(clk, irwrite[3], memdata[7:0], instr[31:24]);

   // datapath
   flopenr    #(WIDTH)  pcreg(clk, reset, pcen, nextpc, pc);
   flop       #(WIDTH)  mdr(clk, memdata, md);
   flop       #(WIDTH)  areg(clk, rd1, a);
   flop       #(WIDTH)  wrd(clk, rd2, writedata);
   flop       #(WIDTH)  res(clk, aluresult, aluout);
   mux2       #(WIDTH)  adrmux(pc, aluout, iord, adr);
   mux2       #(WIDTH)  src1mux(pc, a, alusrca, src1);
   mux4       #(WIDTH)  src2mux(writedata, CONST_ONE, instr[WIDTH-1:0],
                                constx4, alusrcb, src2);
   mux4       #(WIDTH)  pcmux(aluresult, aluout, constx4, CONST_ZERO, pcsource, nextpc);
   mux2       #(WIDTH)  wdmux(aluout, md, memtoreg, wd);
   regfile    #(WIDTH,REGBITS) rf(clk, regwrite, ra1, ra2, wa, wd, rd1, rd2);
```

```
   alu        #(WIDTH) alunit(src1, src2, alucont, aluresult);
   zerodetect #(WIDTH) zd(aluresult, zero);
endmodule

module alu #(parameter WIDTH = 8)
            (input      [WIDTH-1:0] a, b,
             input      [2:0]       alucont,
             output reg [WIDTH-1:0] result);

   wire     [WIDTH-1:0] b2, sum, slt;

   assign b2 = alucont[2] ? ~b:b;
   assign sum = a + b2 + alucont[2];
   // slt should be 1 if most significant bit of sum is 1
   assign slt = sum[WIDTH-1];

   always@(*)
      case(alucont[1:0])
         2'b00: result <= a & b;
         2'b01: result <= a | b;
         2'b10: result <= sum;
         2'b11: result <= slt;
      endcase
endmodule

module regfile #(parameter WIDTH = 8, REGBITS = 3)
                (input                clk,
                 input                regwrite,
                 input  [REGBITS-1:0] ra1, ra2, wa,
                 input  [WIDTH-1:0]   wd,
                 output [WIDTH-1:0]   rd1, rd2);

   reg  [WIDTH-1:0] RAM [(1<<REGBITS)-1:0];

   // three ported register file
   // read two ports combinationally
   // write third port on rising edge of clock
   // register 0 hardwired to 0
   always @(posedge clk)
      if (regwrite) RAM[wa] <= wd;

   assign rd1 = ra1 ? RAM[ra1] : 0;
   assign rd2 = ra2 ? RAM[ra2] : 0;
endmodule

module zerodetect #(parameter WIDTH = 8)
                   (input [WIDTH-1:0] a,
                    output            y);

   assign y = (a==0);
endmodule

module flop #(parameter WIDTH = 8)
             (input                  clk,
              input      [WIDTH-1:0] d,
              output reg [WIDTH-1:0] q);

   always @(posedge clk)
      q <= d;
endmodule

module flopen #(parameter WIDTH = 8)
               (input                  clk, en,
                input      [WIDTH-1:0] d,
                output reg [WIDTH-1:0] q);

   always @(posedge clk)
      if (en) q <= d;
endmodule
```

```
module flopenr #(parameter WIDTH = 8)
                (input                  clk, reset, en,
                 input      [WIDTH-1:0] d,
                 output reg [WIDTH-1:0] q);

   always @(posedge clk)
      if      (reset) q <= 0;
      else if (en)    q <= d;
endmodule

module mux2 #(parameter WIDTH = 8)
             (input  [WIDTH-1:0] d0, d1,
              input              s,
              output [WIDTH-1:0] y);

   assign y = s ? d1 : d0;
endmodule

module mux4 #(parameter WIDTH = 8)
             (input      [WIDTH-1:0] d0, d1, d2, d3,
              input      [1:0]       s,
              output reg [WIDTH-1:0] y);

   always @(*)
      case(s)
         2'b00: y <= d0;
         2'b01: y <= d1;
         2'b10: y <= d2;
         2'b11: y <= d3;
      endcase
endmodule
```

Appendix B VHDL

B.1 Introduction

This appendix gives a quick introduction to VHDL. VHDL is an acronym for the VHSIC Hardware Description Language. VHSIC is in turn an acronym for the US Department of Defense Very High Speed Integrated Circuits program. [Ashenden01] offers a definitive and readable treatment of the language.

VHDL was originally developed in 1981 by the Department of Defense as a language to describe the structure and function of hardware. The IEEE standardized it in 1987, and a revised version was adopted in 1993, updated in 2000, and updated again in 2002 [IEEE1076-02]. The language was first envisioned for documentation, but quickly was adopted for simulation and synthesis. Compared to Verilog, VHDL is more verbose and cumbersome, as you might expect of a language developed by committee. However, VHDL has some features that are convenient for large team design projects, and government contractors and telecommunications companies use it extensively. Religious wars have raged over which HDL is superior, but both are used too widely for CAD vendors not to support them.

As mentioned in Section 1.8.4, there are two general styles of description: *behavioral* and *structural*. Structural VHDL describes how a module is composed of simpler modules or basic primitives such as gates or transistors. Behavioral VHDL describes how the outputs are computed as functions of the inputs. There are two general types of statements used in behavioral VHDL. *Concurrent signal assignments* imply combinational logic. *Processes* can imply combinational logic or sequential logic, depending how they are used. It is good practice to partition your design into combinational and sequential components and then write VHDL in such a way that you get what you want. If you don't know whether a block of logic is combinational or sequential, you are likely to get the wrong thing.

This appendix focuses on a synthesizable subset of the VHDL language. The language also contains extensive capabilities for system modeling that are beyond the scope of this tutorial.

B.2 Behavioral Modeling with Concurrent Signal Assignments

A 32-bit adder is a complex design at the schematic level of representation. It can be constructed from 32 full adder cells, each of which in turn requires about six 2-input gates. VHDL provides a more compact description:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity adder is
    port(a, b: in  STD_LOGIC_VECTOR(31 downto 0);
         y:    out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture synth of adder is
begin
    y <= a + b;
end;
```

This example has three parts: the *library use clauses*, the *entity declaration*, and the *architecture body*. The library part will be discussed in Section B.3.7. The entity defines the inputs and outputs of the adder *block*, which in this case are 32-bit busses. The architecture describes what the block does. In this case, the output `y` is computed as the sum of `a` and `b`. `y <= a + b` is called a *concurrent signal assignment* statement because multiple statements could happen in parallel just as multiple logic gates can operate concurrently.

B.2.1 Bitwise Operators

VHDL has a number of *bitwise* operators that act on busses. For example, the following module describes four inverters.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity inv is
    port(a: in  STD_LOGIC_VECTOR(3 downto 0);
         y: out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of inv is
begin
    y <= not a;
end;
```

Similar bitwise operations are available for the other basic logic functions:

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity gates is
    port(a, b:                  in  STD_LOGIC_VECTOR(3 downto 0);
         y1, y2, y3, y4, y5: out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of gates is
begin
    -- Five different two-input logic gates acting on 4 bit busses
    y1 <= a and b;
    y2 <= a or b;
    y3 <= a xor b;
    y4 <= a nand b;
    y5 <= a nor b;
end;
```

B.2.2 Comments and White Space

The previous examples showed a comment. Comments begin with -- and continue to the end of the line. Comments spanning multiple lines use -- at the beginning of each line.

VHDL is not picky about the use of white space. Nevertheless, proper indenting and spacing is helpful to make nontrivial designs readable. VHDL is not case-sensitive, but a consistent use of upper and lower case also makes the code readable and more portable to other tools that are case-sensitive.

B.2.3 Other Operators

VHDL defines a number of operators available in concurrent assignment statements including:

Multiplying Operators:	`*, /, mod, rem`
Adding Operators:	`+, -`
Relational Operators:	`=, /=, <, <=, >, >=`
Logical Operators:	`not, and, or, nand, nor, xor, xnor`

`=` and `/=` (equality / inequality) on N-bit inputs require N 2-input XNORs to determine equality of each bit and an N-input AND or NAND to combine all the bits. Addition, subtraction, and comparison all require an adder, which is expensive in hardware. Multipliers are even more costly. Do not use these statements without contemplating the number of gates you are generating. Moreover, the implementations are not always particularly efficient for your problem. Division, `mod` (modulo), and `rem` (remainder) are only supported when the right operand is a power of two.

Some synthesis tools ship with optimized libraries for special functions like adders and multipliers. For example, the Synopsys DesignWare libraries produce reasonably good multipliers. If you do not have a license for the libraries, you'll probably be disappointed with the speed and gate count of a multiplier your synthesis tool produces from when it sees `*`.

B.2.4 Conditional Signal Assignment Statements

Conditional signal assignments perform different operations depending on certain conditions. For example, a 2:1 multiplexer can use conditional signal assignment to select one of two 4-bit inputs.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux2 is
    port(d0, d1: in  STD_LOGIC_VECTOR(3 downto 0);
         s:      in  STD_LOGIC;
         y:      out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of mux2 is
begin
    y <= d0 when s = '0' else d1;
end;
```

A 4:1 multiplexer can select one of four inputs using multiple `else` clauses in the conditional signal assignment.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux4 is
    port(d0, d1, d2, d3: in  STD_LOGIC_VECTOR(3 downto 0);
         s:              in  STD_LOGIC_VECTOR(1 downto 0);
         y:              out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of mux4 is
begin
    y <= d0 when s = "00" else
         d1 when s = "01" else
         d2 when s = "10" else
         d3;
end;
```

B.2.5 Selected Signal Assignment Statements

Selected signal assignment statements provide a shorthand when selecting from one of several possibilities. They are analogous to using a `case` statement in place of multiple `if/`

else statements, as will be described in Section B.4.4. The 4:1 multiplexer could have been written with a selected signal assignment as:

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux4 is
    port(d0, d1, d2, d3: in  STD_LOGIC_VECTOR(3 downto 0);
         s:              in  STD_LOGIC_VECTOR(1 downto 0);
         y:              out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth2 of mux4 is
begin
    with s select y <=
        d0 when "00",
        d1 when "01",
        d2 when "10",
        d3 when others;
end;
```

B.3 Basic Constructs

B.3.1 Blocks, Entities, and Architectures

A cell or module in VHDL is called a *block* and is the basic unit of design. It has inputs and outputs specified in an *entity declaration* using the *port* statement. An *architecture body* defines what the block does. VHDL separates the architecture body from the entity declaration to allow multiple bodies for a single block. The syntax of the architecture body is

```
architecture <name> of <block> is
begin
   -- body goes here
end;
```

For example, we have seen body for the adder block with an architecture named synth. It uses a concurrent signal assignment in a behavioral description of the block. The name of the architecture carries no special meaning to VHDL, but helps the human reader understand that the code is intended to be synthesizable (rather than just for simulation). As the design is refined, you can imagine creating a structural description such as 32 full adders connected in a ripple-carry fashion. This architecture might be named struct. While VHDL permits selecting among multiple architectures for a single entity, the syntax is complicated and the simplest approach is to only provide one architecture for each entity.

A side effect of separating architectures from entities is that simple blocks involve substantially more typing in VHDL than in Verilog.

B.3.2 Internal Signals

Often it is convenient to break a complex calculation into intermediate variables. For example, in a full adder, we sometimes define the propagate signal as the XOR of the two inputs a and b. The sum from the adder is the XOR of the propagate signal and the carry-in. We can name the propagate signal using a `signal` statement, in much the same way we use local variables in a programming language.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity fulladder is
    port(a, b, cin:  in STD_LOGIC;
         s, cout:    out STD_LOGIC);
end;

architecture synth of fulladder is
    signal prop: STD_LOGIC;
begin
    prop <= a xor b;
    s <= prop xor cin;
    cout <= (a and b) or (cin and (a or b));
end;
```

B.3.3 Precedence

Notice that we fully parenthesized the `cout` computation. This is necessary in VHDL because the logical operators all have equal precedence. If we had written

```
cout = a and b or cin and (a or b);
```

it could have been interpreted as either

```
cout = ((a and b) or cin) and (a or b));
out = (a and b) or (cin and (a or b)));
```

These are definitely not the same function, so VHDL demands parentheses.

Table B.1 lists operator precedence. All the operators on a particular row have equal precedence and have higher precedence than the row below. Thus, multiplication takes place before addition, as you would expect in most programming languages or in mathematics. On the other hand, AND does not take place before OR, unlike what you would expect in a conventional Boolean equation. Parentheses are used to group expressions when necessary.

Table B.1 Operator Precedence

not	Highest
*, /, mod, rem	
+, -, &	
=, /=, <, <=, >, >=	
=, = =, !=	
and, or, nand, nor, xor	Lowest

B.3.4 Hierarchy

Nontrivial designs are developed in a hierarchical form, in which complex blocks are composed of simpler blocks. For example, a 4-input multiplexer can be constructed from three 2-input multiplexers that were defined earlier.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux4 is
    port(d0, d1, d2, d3: in  STD_LOGIC_VECTOR(3 downto 0);
         s:              in  STD_LOGIC_VECTOR(1 downto 0);
         y:              out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture struct of mux4 is
    component mux2 port(d0, d1: in  STD_LOGIC_VECTOR(3 downto 0);
                        s:      in  STD_LOGIC;
                        y:      out STD_LOGIC_VECTOR(3 downto 0));
    end component;
    signal low, high: STD_LOGIC_VECTOR(3 downto 0);
begin
    lowmux:   mux2 port map(d0, d1, s(0), low);
    highmux:  mux2 port map(d2, d3, s(0), high);
    finalmux: mux2 port map(low, high, s(1), y);
end;
```

This is an example of the structural coding style because the mux4 is built from simpler blocks. It is good practice to avoid (or at least minimize) mixing structural and behavioral descriptions within a single module. Generally, simple modules are described behaviorally and larger modules are composed structurally from these building blocks.

The architecture must first declare the mux2 ports using the *component declaration* statement. This allows VHDL tools to make sure that the component you wish to use has the same ports as the component that was declared somewhere else in an entity statement, preventing errors caused by changing the entity but not the use. However, it makes VHDL code rather cumbersome.

B.3.5 Bit Swizzling

Often it is necessary to work on parts of a bus or to concatenate (join together) signals to construct busses. The mux4 example showed using the least significant bit `s(0)` of a 2-bit select signal for the low and high muxes and the most significant bit `s(1)` for the final mux. Use ranges to select subsets of a bus. For example, an 8-bit wide 2-input mux can be constructed from two 4-bit wide 2-input muxes:

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux2_8 is
    port(d0, d1: in  STD_LOGIC_VECTOR(7 downto 0);
         s:      in  STD_LOGIC;
         y:      out STD_LOGIC_VECTOR(7 downto 0));
end;
```

```
architecture struct of mux2_8 is
    component mux2 port(d0, d1: in  STD_LOGIC_VECTOR(3 downto 0);
                        s:      in  STD_LOGIC;
                        y:      out STD_LOGIC_VECTOR(3 downto 0));
    end component;
begin

    lsbmux:  mux2 port map(d0(3 downto 0), d1(3 downto 0),
                        s, y(3 downto 0));
    msbhmux: mux2 port map(d0(7 downto 4), d1(7 downto 4),
                        s, y(7 downto 4));
end;
```

The & operator is used to concatenate busses. For example, the following code multiplies a number by four by shifting it left two positions.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity shift2 is
    port(a: in  STD_LOGIC_VECTOR(7 downto 0);
         y: out STD_LOGIC_VECTOR(7 downto 0));
end;

architecture synth of shift2 is
begin
    y <= a(5 downto 0) & "00";
end;
```

Variable left and right shifts can also be done with the SHL and SHR functions:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity shifter is
    port(a:   in  STD_LOGIC_VECTOR(7 downto 0);
         amt: in  STD_LOGIC_VECTOR(2 downto 0);
         y:   out STD_LOGIC_VECTOR(7 downto 0));
end;

architecture synth of shifter is
begin
    y <= SHL(a, amt);
end;
```

B.3.6 Types

VHDL uses a strict data typing system that can be clumsy at times. In logic design, our signals are typically binary digits or binary words. In addition to '0' and '1,' it is convenient to have values such as 'z' (high impedance, representing a tristate buffer with a floating output as described in Section B.3.8), 'u' (uninitialized, representing a flip-flop at startup

before reset where the value might be high or low), and 'x' (unknown, representing a node with contention that is driven high by one gate and low by another or the output of a gate whose inputs are 'z')[1]. The STD_LOGIC and STD_LOGIC_VECTOR types represent single- and multiple-bit binary values. Table B.2 shows a truth table for an AND gate using STD_LOGIC types. Notice that the operators are optimistic in that they can sometimes determine the result despite some inputs being unknown. For example '0' and 'z' returns '0' because the output of an AND gate is always '0' if either input is '0.'

Table B.2 AND gate truth table

		A				
		0	1	z	x	u
B	0	0	0	0	0	0
	1	0	1	x	x	u
	z	0	x	x	x	u
	x	0	x	x	x	u
	u	0	u	u	u	u

Despite its fundamental importance, the STD_LOGIC type is not built into VHDL. Instead, it is part of the IEEE.STD_LOGIC_1164 library defined by the IEEE 1164 standard [IEEE1164-93]. Thus, every file must include the library statements we have seen in previous examples. VHDL does have built-in BIT and BIT_VECTOR types that define only '0' and '1,' but we will avoid these types in this tutorial because STD_LOGIC is better suited for logic simulation with tristates, uninitialized nodes, and contention.

In simulation, it may be necessary to define what to do when an input is 'x' or 'u.' For example, the select signal to a multiplexer might be an illegal or undefined value. The mux4 examples in Sections B.2.4 and B.2.5 default to selecting d3 in such an event.

VHDL also has a *Boolean* type with two values: true and false. Boolean values are created by comparisons (like s = '0') and used in conditional statements such as if or when. Boolean and STD_LOGIC values are not interchangeable despite the temptation to think true means '1' and false means '0.' Thus, the following statements are illegal:

```
y <= d1 when s else d0;
q <= (state = S2);
```

Instead we must write

```
y <= d1 when s = '1' else d0;
q <= '1' when state = S2 else '0';
```

[1]Technically, STD_LOGIC also has values of W, L, H, and—for weak unknown, low, and high values and for don't cares, but we will not need them in this tutorial.

While we will not declare any signals to be Boolean, they are automatically implied by comparisons and used by conditional statements.

Similarly, VHDL has an INTEGER type representing a 2's complement integer of at least 32 bits (spanning values $-2^{31} \ldots 2^{31}-1$). Integer values are used as array indices. For example, in the statement

```
lowmux:   mux2 port map(d0, d1, s(0), low);
```

0 is an integer serving as an index to choose one bit of the s signal. We cannot directly index an array with a STD_LOGIC or STD_LOGIC_VECTOR signal. Instead, we must convert the signal to an integer. This is demonstrated in the following 8:1 multiplexer that selects one bit from a vector using a 3-bit index. The conv_integer function is defined in STD_LOGIC_UNSIGNED and performs the conversion from STD_LOGIC_VECTOR to integer for positive (unsigned) values.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity mux8 is
   port(d: in  STD_LOGIC_VECTOR(7 downto 0);
        s: in  STD_LOGIC_VECTOR(2 downto 0);
        y: out STD_LOGIC);
end;

architecture synth of mux8 is
begin
    y <= d(conv_integer(s));
end;
```

VHDL also permits enumeration types. For example, the divide-by-3 finite state machine described in Section B.5 uses three states. We can give the states names using the enumeration type rather than referring to them by binary values.

```
type statetype is (S0, S1, S2);
signal state, nextstate: statetype;
```

B.3.7 Library and Use Clauses

VHDL uses a number of standard libraries that extend the built-in capabilities of the language. As we have seen, the IEEE.STD_LOGIC_1164 library defines the essential STD_LOGIC and STD_LOGIC_VECTOR types. Unfortunately, it does not define basic operations such as addition, comparison, shifts, or conversion to integer for STD_LOGIC_VECTOR data.

Synopsys has developed another freely available library to perform these functions for unsigned numbers. Many other CAD vendors have adopted it for compatibility. The library is typically called STD_LOGIC_UNSIGNED. An equivalent library called STD_LOGIC_SIGNED handles STD_LOGIC_VECTORs representing signed 2's complement

numbers. Right-shifts fill the most significant bit with 0's for unsigned numbers and sign-extend for signed numbers.

The syntax to load a library and use all of its functions is:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
```

B.3.8 Tristates

It is possible to leave a bus floating rather than drive it to '0' or '1.' This floating value is called 'z' in VHDL. For example, a tristate buffer produces a floating output when the enable is '0.'

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity tristate is
    port(a:  in  STD_LOGIC_VECTOR(3 downto 0);
         en: in  STD_LOGIC;
         y:  out STD_LOGIC_VECTOR(3 downto 0));
end;


architecture synth of tristate is
begin
    y <= "ZZZZ" when en = '0' else a;
end;
```

Multiple tristates driving different nonfloating values onto a bus causes contention, displayed as 'x' for STD_LOGIC signals. Floating inputs to gates also cause undefined outputs, displayed as 'x.' Thus, a bus driven by multiple tristates should have one and only one driver active at any given time.

We could define a multiplexer using two tristates so that the output is always driven by exactly one tristate. This guarantees that there are no floating nodes.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux2 is
    port(d0, d1: in  STD_LOGIC_VECTOR(3 downto 0);
         s:      in  STD_LOGIC;
         y:      out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture struct of mux2 is
    component tristate port(a:  in  STD_LOGIC_VECTOR(3 downto 0);
                            en: in  STD_LOGIC;
                            y:  out STD_LOGIC_VECTOR(3 downto 0));
    end component;
    signal sbar: STD_LOGIC;
begin
    sbar <= not s;
    t0: tristate port map(d0, sbar, y);
    t1: tristate port map(d1, s, y);
end;
```

B.3.9 Delays

The delay of a statement can be specified in arbitrary units. For example, the following code defines an inverter with a 100 ps propagation delay. Delays have no impact on synthesis, but can be helpful while debugging simulation waveforms because they make cause and effect more apparent.

```
y <= not a after 100 ps;
```

B.4 Behavioral Modeling with Process Statements

Concurrent signal assignments are reevaluated every time any term on the right side changes. Therefore, they must describe combinational logic. `Process` statements are reevaluated only when signals in the header (called a *sensitivity list*) change. Depending on the form, `process` statements can imply either sequential or combinational circuits.

B.4.1 Flip-flops

Positive edge-triggered flip-flops are described with a process controlled by `clk`:

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity flop is
    port(clk: in  STD_LOGIC;
         d:   in  STD_LOGIC_VECTOR(3 downto 0);
         q:   out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of flop is
begin
    process(clk) begin
        if clk'event and clk = '1' then -- or use "if RISING_EDGE(clk) then"
            q <= d;
        end if;
    end process;
end;
```

VHDL has several idioms for specifying positive edge-triggered flip-flops. In this example, the flip-flop copies `d` to `q` when an event takes place on `clk` and `clk` has a value of '1.' An event is a change in a signal value, i.e., a rising or falling transition. The `RISING_EDGE(clk)` syntax is also acceptable.

At startup, the `q` output is initialized to 'u' by the simulator before the first clock edge arrives. Generally, it is good practice to use resettable registers so that on power-up you

can put your system in a known state. The reset can be either asynchronous or synchronous. Asynchronous resets occur immediately. Synchronous resets only change the output on the rising edge of the clock.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity flopr is
    port(clk, reset: in  STD_LOGIC;
         d:          in  STD_LOGIC_VECTOR(3 downto 0);
         q:          out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synchronous of flopr is
begin
    process(clk) begin
        if clk'event and clk = '1' then
            if reset = '1' then
                q <= "0000";
            else q <= d;
            end if;
        end if;
    end process;
end;

architecture asynchronous of flopr is
begin
    process(clk, reset) begin
        if reset = '1' then
            q <= "0000";
        elsif clk'event and clk = '1' then
            q <= d;
        end if;
    end process;
end;
```

Note that the asynchronously resettable flop evaluates the `process` statement when either `clk` or `reset` change so that it immediately responds to `reset`. The synchronously reset flop is not sensitized to reset, so it waits for the next clock edge before clearing the output.

You can also consider registers with enables that only respond to the clock when the enable is true. The following register retains its old value if both `reset` and `en` are false.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity flopenr is
    port(clk, reset, en: in  STD_LOGIC;
         d:              in  STD_LOGIC_VECTOR(3 downto 0);
         q:              out STD_LOGIC_VECTOR(3 downto 0));
end;
```

```
architecture asynchronous of flopenr is -- asynchronous reset
begin
    process(clk, reset) begin
        if reset = '1' then
            q <= "0000";
        elsif clk'event and clk = '1' then
            if en = '1' then
                q <= d;
            end if;
        end if;
    end process;
end;
```

B.4.2 Latches

Transparent latches are also modeled with process statements. When the clock is high, the latch is *transparent* and the data input flows to the output. When the clock is low, the latch goes *opaque* and the output remains constant.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity latch is
    port(clk: in  STD_LOGIC;
         d:   in  STD_LOGIC_VECTOR(3 downto 0);
         q:   out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of latch is
begin
    process(clk, d) begin
        if clk = '1' then q <= d;
        end if;
    end process;
end;
```

The latch evaluates the `process` statement any time either `clk` or `d` change. If `clk` is high, the output gets the input.

B.4.3 Counters

Consider two ways of describing a 4-bit counter with asynchronous reset. The first scheme (behavioral) implies a sequential circuit containing both the 4-bit register and an adder. The second scheme (structural) explicitly declares modules for the register and adder. Either scheme is good for a simple circuit such as a counter. As you develop more complex finite state machines, it is a good idea to separate the next state logic from the registers in your code.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity counter is
    port(clk, reset: in     STD_LOGIC;
         q:          buffer STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of counter is
begin
    process(clk) begin
        if clk'event and clk = '1' then
            if reset = '1' then q <= "0000";
            else q <= q + "0001";
            end if;
        end if;
    end process;
end;

architecture struct of counter is
    component flopr port(clk, reset: in  STD_LOGIC;
                         d:          in  STD_LOGIC_VECTOR(3 downto 0);
                         q:          out STD_LOGIC_VECTOR(3 downto 0));
    end component;
    component adder port(a, b:  in  STD_LOGIC_VECTOR(3 downto 0);
                         y:     out STD_LOGIC_VECTOR(3 downto 0));
    end component;
    signal nextq: STD_LOGIC_VECTOR(3 downto 0);
begin
    qflop: flopr port map(clk, reset, nextq, q);
    inc:   adder_4 port map(q, "0001", nextq);
end;
```

Note that `q` is defined as `buffer` instead of `out`. `Buffer` ports are required when a signal is used as both input and output within a block, as in the case of `q <= q + "0001"`. Also, `adder_4` is identical to the adder block defined earlier except that it is 4 bits wide.

B.4.4 Combinational Logic

`Process` statements imply sequential logic when some of the inputs do not appear in the `@` stimulus list or might not cause the output to change. For example, in the flop module, `d` is not in the `@` list, so the flop does not immediately respond to changes of `d`. In the latch, `d` is in the `@` list, but changes in `d` are ignored unless `clk` is high so the latch is also sequential. Processes can also be used to imply combinational logic if they are written in such a way that the process is reevaluated every time there are changes in any of the inputs. The following architecture shows how to define a bank of inverters with a `process`. Only the architecture is listed; the rest of the code is the same as the other inverter example. The `begin` and `end process` statements are required even though the process only contains one assignment.

```
architecture proc of inv is
begin
    process(a) begin
        y <= not a;
    end process;
end;
```

Similarly, the next example defines five banks of different kinds of gates.

```
architecture proc of gates is
begin
    process(a, b) begin
        y1 <= a and b;
        y2 <= a or b;
        y3 <= a xor b;
        y4 <= a nand b;
        y5 <= a nor b;
end process;
end;
```

Processes can also contain `if` statements. The following example describes a priority encoder that determines the most significant input bit that is asserted.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity priority is
    port(a: in  STD_LOGIC_VECTOR(3 downto 0);
         y: out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of priority is
begin
    process(a) begin
        if    a(3) = '1' then y <= "1000";
        elsif a(2) = '1' then y <= "0100";
        elsif a(1) = '1' then y <= "0010";
        elsif a(0) = '1' then y <= "0001";
        else                  y <= "0000";
        end if;
    end process;
end;
```

Processes can also contain `case` statements. For example, a 3:8 decoder could be written using a `case` statement. This is easier to read than a description of the same decoder using Boolean equations.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity decoder is
    port(a: in  STD_LOGIC_VECTOR(2 downto 0);
         y: out STD_LOGIC_VECTOR(7 downto 0));
end;
```

```
architecture proc_case of decoder is
begin
    process(a) begin
        case a is
           when "000" =>  y <= "00000001";
           when "001" =>  y <= "00000010";
           when "010" =>  y <= "00000100";
           when "011" =>  y <= "00001000";
           when "100" =>  y <= "00010000";
           when "101" =>  y <= "00100000";
           when "110" =>  y <= "01000000";
           when others => y <= "10000000";
        end case;
    end process;
end;

architecture boolean of decoder is
begin
    y(0) <= not a(2) and not a(1) and not a(0);
    y(1) <= not a(2) and not a(1) and a(0);
    y(2) <= not a(2) and a(1) and not a(0);
    y(3) <= not a(2) and a(1) and a(0);
    y(4) <= a(2) and not a(1) and not a(0);
    y(5) <= a(2) and not a(1) and a(0);
    y(6) <= a(2) and a(1) and not a(0);
    y(7) <= a(2) and a(1) and a(0);
end;
```

Another even better application of case is the logic for a 7-segment display decoder. The 7-segment display is shown in Figure B.1. The decoder takes a 4-bit number and displays its decimal value on the segments. For example, the number 0111 = 7 should turn on segments a, b, and c. The equivalent logic with concurrent signal assignments describing the detailed logic for each bit would be tedious. This more abstract approach is faster to write, clearer to read, and can be automatically synthesized down to an efficient logic implementation.

FIG B.1 7-segment display mapping

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity seven_seg_decoder is
    port(data:     in  STD_LOGIC_VECTOR(3 downto 0);
         segments: out STD_LOGIC_VECTOR(6 downto 0));
end;

architecture synth of seven_seg_decoder is
--                                        Segment #'s: abcdefg
    constant BLANK: STD_LOGIC_VECTOR(6 downto 0) := "0000000";
    constant ZERO:  STD_LOGIC_VECTOR(6 downto 0) := "1111110";
    constant ONE:   STD_LOGIC_VECTOR(6 downto 0) := "0110000";
    constant TWO:   STD_LOGIC_VECTOR(6 downto 0) := "1101101";
    constant THREE: STD_LOGIC_VECTOR(6 downto 0) := "1111001";
```

```
    constant FOUR:  STD_LOGIC_VECTOR(6 downto 0) := "0110011";
    constant FIVE:  STD_LOGIC_VECTOR(6 downto 0) := "1011011";
    constant SIX:   STD_LOGIC_VECTOR(6 downto 0) := "1011111";
    constant SEVEN: STD_LOGIC_VECTOR(6 downto 0) := "1110000";
    constant EIGHT: STD_LOGIC_VECTOR(6 downto 0) := "1111111";
    constant NINE:  STD_LOGIC_VECTOR(6 downto 0) := "1111011";
begin
    process(data) begin
        case data is
            when "0000" => segments <= ZERO;
            when "0001" => segments <= ONE;
            when "0010" => segments <= TWO;
            when "0011" => segments <= THREE;
            when "0100" => segments <= FOUR;
            when "0101" => segments <= FIVE;
            when "0110" => segments <= SIX;
            when "0111" => segments <= SEVEN;
            when "1000" => segments <= EIGHT;
            when "1001" => segments <= NINE;
            when others => segments <= BLANK;
        end case;
    end process;
end;
```

This example shows the use of *constant declarations* to make the code more readable. The `case` statement has an `others` clause to display a blank output when the input is outside the range of decimal digits.

Overall, concurrent, conditional, and selected signal assignment statements can mimic the effects of processes with sequential assignments, `if` statements, and `case` statements. Moreover, processes can inadvertently imply sequential logic if the output is not assigned a value for all inputs. One reasonable style of VHDL coding is to use processes only for flip-flops and latches.

B.4.5 Memories

VHDL has an array construct used to describe memories. The following module describes a 64-word × 16-bit RAM that is written when `wrb` is low and otherwise read. Synthesis tools are often restricted to generating gates from a library and produce poor memory arrays. A specialized memory generator is commonly used instead.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity memory is
    port(addr: in  STD_LOGIC_VECTOR(5 downto 0);
         wrb:  in  STD_LOGIC;
         din:  in  STD_LOGIC_VECTOR(15 downto 0);
         dout: out STD_LOGIC_VECTOR(15 downto 0));
end;
```

```
architecture synth of memory is
    type ramtype is array (63 downto 0) of STD_LOGIC_VECTOR(15 downto 0);
    signal mem: ramtype;
begin
    process(addr, wrb, din) begin
        if wrb = '0' then mem(conv_integer(addr)) <= din;
        else dout <= mem(conv_integer(addr));
        end if;
    end process;
end;
```

Note that the memory contents are stored in the `signal mem`. It is declared to be of `type ramtype`, an array of 64 16-bit words. Arrays are indexed with an integer, but `addr` is a `STD_LOGIC_VECTOR`. Hence, the `conv_integer` function is used to convert between data types. This does not produce any hardware, but keeps the VHDL type system happy.

B.5 Finite State Machines

There are two styles of finite state machines. In *Mealy machines* (Figure B.2(a)), the output is a function of the current state and inputs. In *Moore machines* (Figure B.2(b)), the output is a function of only the current state. FSMs are modeled in VHDL with `process` defining the *state registers* and combinational logic defining the *next state* and *output logic*.

Let us first consider a simple finite state machine with one output and no inputs, a divide-by-3 counter. The output should be asserted every three clock cycles. The state transition diagram for a Moore machine is shown in Figure B.3. The output value is labeled in each state because the output is only a function of the state.

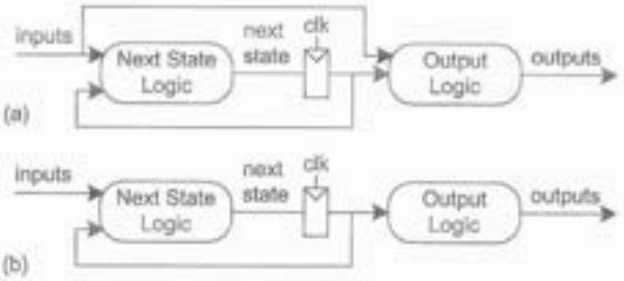


FIG B.2 Moore and Mealy machines

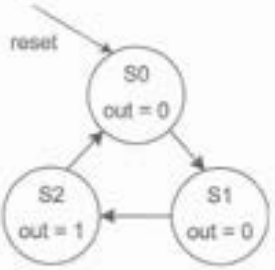


FIG B.3 Divide-by-3 counter state transition diagram

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity divideby3FSM is
    port(clk, reset: in  STD_LOGIC;
         q:          out STD_LOGIC);
end;

architecture synth of divideby3FSM is
    type statetype is (S0, S1, S2);
    signal state, nextstate: statetype;
begin
    -- state register
    process(clk, reset) begin
```

```
        if reset = '1' then state <= S0;
        elsif clk'event and clk = '1' then state <= nextstate;
        end if;
    end process;

    -- next state logic
    nextstate <= S1 when state = S0 else
                 S2 when state = S1 else
                 S0;

    -- output logic
    q <= '1' when state = S2 else '0';
end;
```

Defining states with a type rather than a binary encoding makes the code easier to read. It also allows the synthesis tool to search for an encoding that is fastest or uses the least area. For example, the synthesis tool might choose `S0` = 00, `S1` = 01, and `S2` = 10 or might choose `S0` = 000, `S1` = 010, `S2` = 100.

The FSM model is divided into three portions: the state register, next state logic, and output logic. The state register logic describes an asynchronously resettable register that resets to an initial state and otherwise advances to the computed next state. The next state logic computes `nextstate` as a function of `state` and the inputs; in this example, there are no inputs. The final `else` is essential to make sure `nextstate` gets a value even when `state` has an illegal value; otherwise, latches might be implied. Finally, the output logic may be a function of the current state alone in a Moore machine or of the current state and inputs in a Mealy machine.

It would be tempting to write the output logic as

```
q <= (state = S2);
```

Recall that `(state = S2)` returns a `boolean` result, either `true` or `false`. `q` is of type `STD_LOGIC`. VHDL is picky about types, so the `when/else` clause is needed to convert Boolean to '1' or '0' logic values.

The next example shows a finite state machine with an input `a` and two outputs. Output `x` is true when the input is the same now as it was last cycle. Output `y` is true when the input is the same now as it was for the past two cycles. This is a Mealy machine because the output depends on the current inputs as well as the state. The outputs are labeled on each transition after the input. The state transition diagram is shown in Figure B.4.

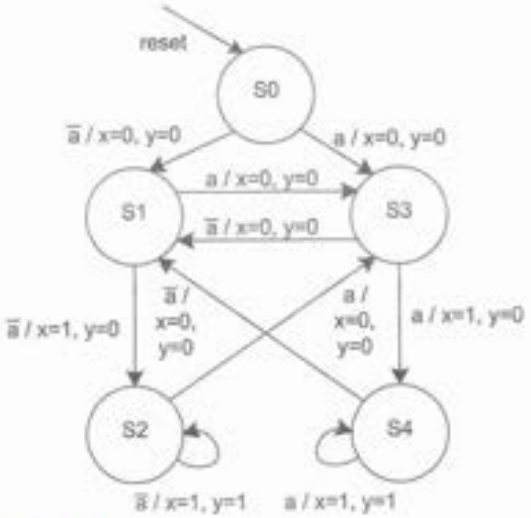


FIG B.4 History FSM state transition diagram

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity historyFSM is
    port(clk, reset: in  STD_LOGIC;
```

```
        a:          in  STD_LOGIC;
        x, y:       out STD_LOGIC);
end;

architecture synth of historyFSM is
    type statetype is (S0, S1, S2, S3, S4);
    signal state, nextstate: statetype;
begin
    -- state register
    process(clk, reset) begin
        if reset = '1' then state <= S0;
        elsif clk'event and clk = '1' then state <= nextstate;
        end if;
    end process;

    -- next state logic
    process(state, a) begin
        case state is
            when S0 =>  if a = '1' then nextstate <= S3;
                        else            nextstate <= S1;
                        end if;
            when S1 =>  if a = '1' then nextstate <= S3;
                        else            nextstate <= S2;
                        end if;
            when S2 =>  if a = '1' then nextstate <= S3;
                        else            nextstate <= S2;
                        end if;
            when S3 =>  if a = '1' then nextstate <= S4;
                        else            nextstate <= S1;
                        end if;
            when S4 =>  if a = '1' then nextstate <= S4;
                        else            nextstate <= S1;
                        end if;
            when others =>              nextstate <= S0;
        end case;
    end process;

    -- output logic
    x <= '1' when ((state = S1 or state = s2) and a = '1') or
                  ((state = S3 or state = S4) and a = '1')
             else '1';
    y <= '1' when (state = S2 and a = '1') or (state = S4 and a = '1')
             else '1';
end;
```

B.6 Parameterized Blocks

So far, all of our blocks have had fixed-width inputs and outputs. Thus, we have needed separate entities for 4- and 8-bit wide 2-input multiplexers. VHDL can automatically

produce hardware with parameterized bit width. For example, we can declare a parameterized bank of inverters with a default of 8 gates as:

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity inv is
    generic(width: integer := 8);
    port(a: in  STD_LOGIC_VECTOR(width-1 downto 0);
         y: out STD_LOGIC_VECTOR(width-1 downto 0));
end;

architecture synth of inv is
begin
    y <= not a;
end;
```

We can adjust the parameter when we instantiate the block. For example, we can build a bank of 12 buffers using a pair of banks of 12 inverters.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity buf is
    generic(numbits: integer := 12);
    port(a: in  STD_LOGIC_VECTOR(numbits-1 downto 0);
         y: out STD_LOGIC_VECTOR(numbits-1 downto 0));
end;

architecture synth of buf is
    component inv generic(width: integer);
                  port(a: in  STD_LOGIC_VECTOR(width-1 downto 0);
                       y: out STD_LOGIC_VECTOR(width-1 downto 0));
    end component;
    signal x: STD_LOGIC_VECTOR(numbits-1 downto 0);
begin
    i1: inv generic map(numbits) port map(a, x);
    i2: inv generic map(numbits) port map(x, y);
end;
```

VHDL also has the ability to generate an adjustable number of gates connected in arbitrary ways. The following code produces an N-input AND gate as a cascade of $N-1$ 2-input AND gates (N = `width` = 8). It uses the `generate` command in conjunction with `for` and `if` statements to conditionally produce an array of gates. `Generate` can easily produce large numbers of gates, so use it with care to be sure you intend to imply all the hardware.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity andN is
    generic(width: integer := 8);
    port(a: in  STD_LOGIC_VECTOR(width-1 downto 0);
         y: out STD_LOGIC);
end;
```

```
architecture synth of andN is
    signal i: integer;
    signal x: STD_LOGIC_VECTOR(width-1 downto 1);
begin
    AllBits: for i in width-1 downto 1 generate
        LowBit: if i = 1 generate
            A1: x(1) <= a(0) and a(1);
          end generate;
          OtherBits: if i /= 1 generate
            Ai: x(i) <= a(i) and x(i-1);
          end generate;
    end generate;
    y <= x(width-1);
end;
```

In a similar example, `generate` can instantiate an array of other blocks. For example, an *N*-bit ripple-carry adder can be built from *N* full adders with the carry signals chained together. Of course, using the + symbol to imply an adder is less effort and gives the synthesis tool more freedom to trade off area and delay as the constraints demand.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity rippleadder is
    generic(width: integer := 8);
    port(a, b: in  STD_LOGIC_VECTOR(width-1 downto 0);
         s:    out STD_LOGIC_VECTOR(width-1 downto 0));
end;

architecture synth of rippleadder is
    component fulladder port(a, b, cin:  in  STD_LOGIC;
                             s, cout:    out STD_LOGIC);
    end component;
    signal i: integer;
    signal c: STD_LOGIC_VECTOR(width-1 downto 0);
begin
    AllBits: for i in width-1 downto 0 generate
        LowBit: if i = 0 generate
            FA0: fulladder port map(a(0), b(0), '0', s(0), c(0));
        end generate;
        OtherBits: if i /= 0 generate
            FAi: fulladder port map(a(i), b(i), c(i-1), s(i), c(i));
        end generate;
    end generate;
end;
```

B.7 Example: MIPS Processor

To illustrate a nontrivial VHDL design, this section lists the VHDL code and test bench for the MIPS processor subset discussed in Chapter 1. The example handles only the `LB`, `SB`, `ADD`, `SUB`, `AND`, `OR`, `SLT`, `BEQ`, and `J` instructions. It uses an 8-bit datapath and only eight

registers. As the instruction is 32 bits wide, it is loaded in four successive fetch cycles across an 8-bit path to external memory.

The test bench initializes a 512-byte memory with instructions and data from a text file. The code exercises each of the instructions. The `mipstest.asm` assembly language file and `memfile.dat` text file are shown in Section A.10. The test bench runs until it observes a memory write. If the value 7 is written to address 5, the code executed correctly. If all goes well, the test bench should take 100 cycles (1000 ns) to run.

The code uses the `conv_std_logic_vector` function that converts an integer into a `STD_LOGIC_VECTOR` of the specified width. This is helpful to create constants of parameterized widths. The function is in the `IEEE.STD_LOGIC_ARITH` library. It also uses the `**` exponentiation operator to determine the number of register file entries given the number of bits of register address.

```
-------------------------------------------------------------
-- mips.vhd
-- David_Harris@hmc.edu 9/9/03
-- Model of subset of MIPS processor described in Ch 1
-------------------------------------------------------------

-------------------------------------------------------------
-- Entity Declarations
-------------------------------------------------------------

library IEEE; use IEEE.STD_LOGIC_1164.all; use IEEE.STD_LOGIC_UNSIGNED.all;
entity top is -- top-level design for testing
    generic(width:   integer := 8;      -- default 8-bit datapath
            regbits: integer := 3);     -- and 3 bit register addresses (8 regs)
end;

library IEEE; use IEEE.STD_LOGIC_1164.all; use STD.TEXTIO.all;
use IEEE.STD_LOGIC_UNSIGNED.all;  use IEEE.STD_LOGIC_ARITH.all;
entity memory is -- external memory accessed by MIPS
    generic(width: integer);
    port(clk, memwrite:  in STD_LOGIC;
         adr, writedata: in STD_LOGIC_VECTOR(width-1 downto 0);
         memdata:        out STD_LOGIC_VECTOR(width-1 downto 0));
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity mips is -- simplified MIPS processor
    generic(width:   integer := 8;      -- default 8-bit datapath
            regbits: integer := 3);     -- and 3 bit register addresses (8 regs)
    port(clk, reset:          in  STD_LOGIC;
         memdata:             in  STD_LOGIC_VECTOR(width-1 downto 0);
         memread, memwrite:   out STD_LOGIC;
         adr, writedata:      out STD_LOGIC_VECTOR(width-1 downto 0));
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity controller is -- control FSM
    port(clk, reset:                          in  STD_LOGIC;
         op:                                  in  STD_LOGIC_VECTOR(5 downto 0);
         zero:                                in  STD_LOGIC;
         memread, memwrite, alusrca, memtoreg,
         iord, pcen, regwrite, regdst:        out STD_LOGIC;
         pcsource, alusrcb, aluop:            out STD_LOGIC_VECTOR(1 downto 0);
         irwrite:                             out STD_LOGIC_VECTOR(3 downto 0));
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity alucontrol is -- ALU control decoder
    port(aluop:   in  STD_LOGIC_VECTOR(1 downto 0);
         funct:   in  STD_LOGIC_VECTOR(5 downto 0);
```

```
        alucont: out STD_LOGIC_VECTOR(2 downto 0));
end;

library IEEE; use IEEE.STD_LOGIC_1164.all; use IEEE.STD_LOGIC_ARITH.all;
entity datapath is  -- MIPS datapath
    generic(width, regbits: integer);
    port(clk, reset:          in  STD_LOGIC;
         memdata:             in  STD_LOGIC_VECTOR(width-1 downto 0);
         alusrca, memtoreg, iord, pcen,
         regwrite, regdst:    in  STD_LOGIC;
         pcsource, alusrcb:   in  STD_LOGIC_VECTOR(1 downto 0);
         irwrite:             in  STD_LOGIC_VECTOR(3 downto 0);
         alucont:             in  STD_LOGIC_VECTOR(2 downto 0);
         zero:                out STD_LOGIC;
         instr:               out STD_LOGIC_VECTOR(31 downto 0);
         adr, writedata:      out STD_LOGIC_VECTOR(width-1 downto 0));
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all; use IEEE.STD_LOGIC_UNSIGNED.all;
entity alu is -- Arithmetic/Logic unit with add/sub, AND, OR, set less than
    generic(width: integer);
    port(a, b:    in  STD_LOGIC_VECTOR(width-1 downto 0);
         alucont: in  STD_LOGIC_VECTOR(2 downto 0);
         result:  out STD_LOGIC_VECTOR(width-1 downto 0));
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all; use IEEE.STD_LOGIC_ARITH.all;
entity regfile is -- three-port register file of 2**regbits words x width bits
    generic(width, regbits: integer);
    port(clk:          in  STD_LOGIC;
         write:        in  STD_LOGIC;
         ra1, ra2, wa: in  STD_LOGIC_VECTOR(regbits-1 downto 0);
         wd:           in  STD_LOGIC_VECTOR(width-1 downto 0);
         rd1, rd2:     out STD_LOGIC_VECTOR(width-1 downto 0));
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity zerodetect is -- true if all input bits are zero
    generic(width: integer);
    port(a: in  STD_LOGIC_VECTOR(width-1 downto 0);
         y: out STD_LOGIC);
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity flop is -- flip-flop
    generic(width: integer);
    port(clk: in  STD_LOGIC;
         d:   in  STD_LOGIC_VECTOR(width-1 downto 0);
         q:   out STD_LOGIC_VECTOR(width-1 downto 0));
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity flopen is -- flip-flop with enable
    generic(width: integer);
    port(clk, en: in  STD_LOGIC;
         d:       in  STD_LOGIC_VECTOR(width-1 downto 0);
         q:       out STD_LOGIC_VECTOR(width-1 downto 0));
end;

library IEEE; use IEEE.STD_LOGIC_1164.all; use IEEE.STD_LOGIC_ARITH.all;
entity flopenr is -- flip-flop with enable and synchronous reset
    generic(width: integer);
    port(clk, reset, en: in  STD_LOGIC;
         d:              in  STD_LOGIC_VECTOR(width-1 downto 0);
         q:              out STD_LOGIC_VECTOR(width-1 downto 0));
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity mux2 is -- two-input multiplexer
    generic(width: integer);
    port(d0, d1: in  STD_LOGIC_VECTOR(width-1 downto 0);
         s:      in  STD_LOGIC;
         y:      out STD_LOGIC_VECTOR(width-1 downto 0));
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity mux4 is  -- four-input multiplexer
    generic(width: integer);
```

```
    port(d0, d1, d2, d3: in  STD_LOGIC_VECTOR(width-1 downto 0);
         s:              in  STD_LOGIC_VECTOR(1 downto 0);
         y:              out STD_LOGIC_VECTOR(width-1 downto 0));
end;

-----------------------------------------------------------
-- Architecture Definitions
-----------------------------------------------------------
architecture test of top is
    component mips generic(width:   integer := 8;   -- default 8-bit datapath
                           regbits: integer := 3);  -- and 3 bit register addresses (8 regs)
        port(clk, reset:        in  STD_LOGIC;
             memdata:           in  STD_LOGIC_VECTOR(width-1 downto 0);
             memread, memwrite: out STD_LOGIC;
             adr, writedata:    out STD_LOGIC_VECTOR(width-1 downto 0));
    end component;
    component memory generic(width: integer);
        port(clk, memwrite:  in  STD_LOGIC;
             adr, writedata: in  STD_LOGIC_VECTOR(width-1 downto 0);
             memdata:        out STD_LOGIC_VECTOR(width-1 downto 0));
    end component;
    signal clk, reset, memread, memwrite: STD_LOGIC;
    signal memdata, adr, writedata: STD_LOGIC_VECTOR(width-1 downto 0);
begin
    -- mips being tested
    dut: mips generic map(width, regbits)
             port map(clk, reset, memdata, memread, memwrite, adr, writedata);
    -- external memory for code and data
    exmem: memory generic map(width)
                 port map(clk, memwrite, adr, writedata, memdata);

    -- Generate clock with 10 ns period
    process begin
        clk <= '1';
        wait for 5 ns;
        clk <= '0';
        wait for 5 ns;
    end process;

    -- Generate reset for first two clock cycles
    process begin
        reset <= '1';
        wait for 22 ns;
        reset <= '0';
        wait;
    end process;

    -- check that 7 gets written to address 5 at end of program
    process (clk) begin
        if (clk'event and clk = '0' and memwrite = '1') then
            if (conv_integer(adr) = 5 and conv_integer(writedata) = 7) then
                report "Simulation completed successfully";
            else report "Simulation failed.";
            end if;
        end if;
    end process;
end;

architecture synth of memory is
begin
    process is
        file mem_file: text open read_mode is "memfile.dat";
        variable L: line;
        variable ch: character;
        variable index, result: integer;
        type ramtype is array (511 downto 0) of STD_LOGIC_VECTOR(7 downto 0);
        variable mem: ramtype;
    begin
        -- initialize memory from file
        for i in 0 to 511 loop -- set all contents low
            mem(conv_integer(i)) := "00000000";
        end loop;
        index := 0;
        while not endfile(mem_file) loop
            readline(mem_file, L);
            for j in 0 to 3 loop
                result := 0;
                    for i in 1 to 2 loop
                        read(L, ch);
                        if '0' <= ch and ch <= '9' then
                            result := result*16 + character'pos(ch)-character'pos('0');
                        elsif 'a' <= ch and ch <= 'f' then
```

```
                        result := result*16 + character'pos(ch)-character'pos('a')+10;
                    else report "Format error on line " & integer'image(index)
                        severity error;
                    end if;
                end loop;
            mem(index*4+j) := conv_std_logic_vector(result, width);
        end loop;
        index := index + 1;
    end loop;
    -- read or write memory
    loop
        if clk'event and clk = '1' then
            if (memwrite = '1') then mem(conv_integer(adr)) := writedata;
            end if;
        end if;
        memdata <= mem(conv_integer(adr));
        wait on clk, adr;
    end loop;
  end process;
end;

architecture struct of mips is
    component controller
        port(clk, reset:                    in  STD_LOGIC;
             op:                            in  STD_LOGIC_VECTOR(5 downto 0);
             zero:                          in  STD_LOGIC;
             memread, memwrite, alusrca, memtoreg,
             iord, pcen, regwrite, regdst: out STD_LOGIC;
             pcsource, alusrcb, aluop:     out STD_LOGIC_VECTOR(1 downto 0);
             irwrite:                      out STD_LOGIC_VECTOR(3 downto 0));
    end component;
    component alucontrol
        port(aluop:      in  STD_LOGIC_VECTOR(1 downto 0);
             funct:      in  STD_LOGIC_VECTOR(5 downto 0);
             alucont:    out STD_LOGIC_VECTOR(2 downto 0));
    end component;
    component datapath generic(width, regbits: integer);
        port(clk, reset:        in  STD_LOGIC;
             memdata:           in  STD_LOGIC_VECTOR(width-1 downto 0);
             alusrca, memtoreg, iord, pcen,
             regwrite, regdst:  in  STD_LOGIC;
             pcsource, alusrcb: in  STD_LOGIC_VECTOR(1 downto 0);
             irwrite:           in  STD_LOGIC_VECTOR(3 downto 0);
             alucont:           in  STD_LOGIC_VECTOR(2 downto 0);
             zero:              out STD_LOGIC;
             instr:             out STD_LOGIC_VECTOR(31 downto 0);
             adr, writedata:    out STD_LOGIC_VECTOR(width-1 downto 0));
    end component;
    signal instr: STD_LOGIC_VECTOR(31 downto 0);
    signal zero, alusrca, memtoreg, iord, pcen, regwrite, regdst: STD_LOGIC;
    signal aluop, pcsource, alusrcb: STD_LOGIC_VECTOR(1 downto 0);
    signal irwrite: STD_LOGIC_VECTOR(3 downto 0);
    signal alucont: STD_LOGIC_VECTOR(2 downto 0);
begin
    cont: controller port map(clk, reset, instr(31 downto 26), zero,
                              memread, memwrite, alusrca, memtoreg,
                              iord, pcen, regwrite, regdst,
                              pcsource, alusrcb, aluop, irwrite);
    ac: alucontrol port map(aluop, instr(5 downto 0), alucont);
    dp: datapath generic map(width, regbits)
                port map(clk, reset, memdata, alusrca, memtoreg,
                         iord, pcen, regwrite, regdst,
                         pcsource, alusrcb, irwrite,
                         alucont, zero, instr, adr, writedata);
end;

architecture synth of controller is
    type statetype is (FETCH1, FETCH2, FETCH3, FETCH4, DECODE, MEMADR,
                       LBRD, LBWR, SBWR, RTYPEEX, RTYPEWR, BEQEX, JEX);
        constant LB:    STD_LOGIC_VECTOR(5 downto 0) := "100000";
        constant SB:    STD_LOGIC_VECTOR(5 downto 0) := "101000";
        constant RTYPE: STD_LOGIC_VECTOR(5 downto 0) := "000000";
        constant BEQ:   STD_LOGIC_VECTOR(5 downto 0) := "000100";
        constant J:     STD_LOGIC_VECTOR(5 downto 0) := "000010";
        signal state, nextstate:     statetype;
        signal pcwrite, pcwritecond: STD_LOGIC;
begin
    process (clk) begin -- state register
        if clk'event and clk = '1' then
            if reset = '1' then state <= FETCH1;
```

```
        else state <= nextstate;
        end if;
    end if;
end process;

process (state, op) begin -- next state logic
    case state is
        when FETCH1 => nextstate <= FETCH2;
        when FETCH2 => nextstate <= FETCH3;
        when FETCH3 => nextstate <= FETCH4;
        when FETCH4 => nextstate <= DECODE;
        when DECODE => case op is
                           when LB | SB => nextstate <= MEMADR;
                           when RTYPE => nextstate <= RTYPEEX;
                           when BEQ => nextstate <= BEQEX;
                           when J => nextstate <= JEX;
                           when others => nextstate <= FETCH1; -- should never happen
                       end case;
        when MEMADR => case op is
                           when LB => nextstate <= LBRD;
                           when SB => nextstate <= SBWR;
                           when others => nextstate <= FETCH1; -- should never happen
                       end case;
        when LBRD => nextstate <= LBWR;
        when LBWR => nextstate <= FETCH1;
        when SBWR => nextstate <= FETCH1;
        when RTYPEEX => nextstate <= RTYPEWR;
        when RTYPEWR => nextstate <= FETCH1;
        when BEQEX => nextstate <= FETCH1;
        when JEX => nextstate <= FETCH1;
        when others => nextstate <= FETCH1; -- should never happen
    end case;
end process;

process (state) begin
    -- set all outputs to zero, then conditionally assert just the appropriate ones
    irwrite <= "0000";
    pcwrite <= '0'; pcwritecond <= '0';
    regwrite <= '0';  regdst <= '0';
    memread <= '0'; memwrite <= '0';
    alusrca <= '0'; alusrcb <= "00"; aluop <= "00";
    pcsource <= "00";
    iord <= '0'; memtoreg <= '0';

    case state is
        when FETCH1 => memread <= '1';
                       irwrite <= "1000";
                       alusrcb <= "01";
                       pcwrite <= '1';
        when FETCH2 => memread <= '1';
                       irwrite <= "0100";
                       alusrcb <= "01";
                       pcwrite <= '1';
        when FETCH3 => memread <= '1';
                       irwrite <= "0010";
                       alusrcb <= "01";
                       pcwrite <= '1';
        when FETCH4 => memread <= '1';
                       irwrite <= "0001";
                       alusrcb <= "01";
                       pcwrite <= '1';
        when DECODE => alusrcb <= "11";
        when MEMADR => alusrca <= '1';
                       alusrcb <= "10";
        when LBRD =>   memread <= '1';
                       iord <= '1';
        when LBWR =>   regwrite <= '1';
                       memtoreg <= '1';
        when SBWR =>   memwrite <= '1';
                       iord <= '1';
        when RTYPEEX => alusrca <= '1';
                        aluop <= "10";
        when RTYPEWR => regdst <= '1';
                        regwrite <= '1';
        when BEQEX =>  alusrca <= '1';
                       aluop <= "01";
                       pcwritecond <= '1';
                       pcsource <= "01";
        when JEX =>    pcwrite <= '1';
                       pcsource <= "10";
```

```
        end case;
    end process;

    pcen <= pcwrite or (pcwritecond and zero); -- program counter enable
end;

architecture synth of alucontrol is
begin
    process(aluop, funct) begin
        case aluop is
            when "00" => alucont <= "010"; -- add (for lb/sb/addi)
            when "01" => alucont <= "110"; -- sub (for beq)
            when others => case funct is          -- R-type instructions
                               when "100000" => alucont <= "010"; -- add (for add)
                               when "100010" => alucont <= "110"; -- subtract (for sub)
                               when "100100" => alucont <= "000"; -- logical and (for and)
                               when "100101" => alucont <= "001"; -- logical or (for or)
                               when "101010" => alucont <= "111"; -- set on less (for slt)
                               when others   => alucont <= "---"; -- should never happen
                           end case;
        end case;
    end process;
end;

architecture struct of datapath is
    component alu generic(width: integer);
        port(a, b:    in  STD_LOGIC_VECTOR(width-1 downto 0);
             alucont: in  STD_LOGIC_VECTOR(2 downto 0);
             result:  out STD_LOGIC_VECTOR(width-1 downto 0));
    end component;
    component regfile generic(width, regbits: integer);
        port(clk:          in  STD_LOGIC;
             write:        in  STD_LOGIC;
             ra1, ra2, wa: in  STD_LOGIC_VECTOR(regbits-1 downto 0);
             wd:           in  STD_LOGIC_VECTOR(width-1 downto 0);
             rd1, rd2:     out STD_LOGIC_VECTOR(width-1 downto 0));
    end component;
    component zerodetect generic(width: integer);
        port(a: in  STD_LOGIC_VECTOR(width-1 downto 0);
             y: out STD_LOGIC);
    end component;
    component flop generic(width: integer);
        port(clk: in  STD_LOGIC;
             d:   in  STD_LOGIC_VECTOR(width-1 downto 0);
             q:   out STD_LOGIC_VECTOR(width-1 downto 0));
    end component;
    component flopen generic(width: integer);
        port(clk, en: in  STD_LOGIC;
             d:       in  STD_LOGIC_VECTOR(width-1 downto 0);
             q:       out STD_LOGIC_VECTOR(width-1 downto 0));
    end component;
    component flopenr generic(width: integer);
        port(clk, reset, en: in  STD_LOGIC;
             d:              in  STD_LOGIC_VECTOR(width-1 downto 0);
             q:              out STD_LOGIC_VECTOR(width-1 downto 0));
    end component;
    component mux2 generic(width: integer);
        port(d0, d1: in  STD_LOGIC_VECTOR(width-1 downto 0);
             s:      in  STD_LOGIC;
             y:      out STD_LOGIC_VECTOR(width-1 downto 0));
    end component;
    component mux4 generic(width: integer);
        port(d0, d1, d2, d3: in  STD_LOGIC_VECTOR(width-1 downto 0);
             s:              in  STD_LOGIC_VECTOR(1 downto 0);
             y:              out STD_LOGIC_VECTOR(width-1 downto 0));
    end component;
    constant CONST_ONE:  STD_LOGIC_VECTOR(width-1 downto 0) := conv_std_logic_vector(1, width);
    constant CONST_ZERO: STD_LOGIC_VECTOR(width-1 downto 0) := conv_std_logic_vector(0, width);
    signal ra1, ra2, wa: STD_LOGIC_VECTOR(regbits-1 downto 0);
    signal pc, nextpc, md, rd1, rd2, wd, a,
           srcl, src2, aluresult, aluout, dp_writedata, constx4: STD_LOGIC_VECTOR(width-1 downto 0);
    signal dp_instr: STD_LOGIC_VECTOR(31 downto 0);

begin
    -- shift left constant field by 2
    constx4 <= dp_instr(width-3 downto 0) & "00";

    -- register file addres fields
    ra1 <= dp_instr(regbits+20 downto 21);
    ra2 <= dp_instr(regbits+15 downto 16);
```

```
    regmux:  mux2 generic map(regbits) port map(dp_instr(regbits+15 downto 16),
                                                dp_instr(regbits+10 downto 11), regdst, wa);

    -- independent of bit width, load dp_instruction into four 8-bit registers over four cycles
    ir0:     flopen generic map(8) port map(clk, irwrite(0), memdata(7 downto 0), dp_instr(7 downto 0));
    ir1:     flopen generic map(8) port map(clk, irwrite(1), memdata(7 downto 0), dp_instr(15 downto 8));
    ir2:     flopen generic map(8) port map(clk, irwrite(2), memdata(7 downto 0), dp_instr(23 downto 16));
    ir3:     flopen generic map(8) port map(clk, irwrite(3), memdata(7 downto 0), dp_instr(31 downto 24));
    -- datapath
    pcreg:   flopenr generic map(width) port map(clk, reset, pcen, nextpc, pc);
    mdr:     flop generic map(width) port map(clk, memdata, md);
    areg:    flop generic map(width) port map(clk, rd1, a);
    wrd:     flop generic map(width) port map(clk, rd2, dp_writedata);
    res:     flop generic map(width) port map(clk, aluresult, aluout);
    adrmux:  mux2 generic map(width) port map(pc, aluout, iord, adr);
    srcimux: mux2 generic map(width) port map(pc, a, alusrca, src1);
    src2mux: mux4 generic map(width) port map(dp_writedata, CONST_ONE,
                                              dp_instr(width-1 downto 0), constx4, alusrcb, src2);
    pcmux:   mux4 generic map(width) port map(aluresult, aluout, constx4, CONST_ZERO, pcsource, nextpc);
    wdmux:   mux2 generic map(width) port map(aluout, md, memtoreg, wd);
    rf:      regfile generic map(width, regbits) port map(clk, regwrite, ra1, ra2, wa, wd, rd1, rd2);
    alunit:  alu generic map(width) port map(srcl, src2, alucont, aluresult);
    zd:      zerodetect generic map(width) port map(aluresult, zero);

    -- drive outputs
    instr <= dp_instr; writedata <= dp_writedata;
end;

architecture synth of alu is
    signal b2, sum, slt: STD_LOGIC_VECTOR(width-1 downto 0);
begin
    b2 <= not b when alucont(2) = '1' else b;
    sum <= a + b2 + alucont(2);
    -- slt should be 1 if most significant bit of sum is 1
    slt <= conv_std_logic_vector(1, width) when sum(width-1) = '1'
           else conv_std_logic_vector(0, width);
    with alucont(1 downto 0) select result <=
        a and b when "00",
        a or b  when "01",
        sum     when "10",
        slt     when others;
end;

architecture synth of regfile is
    type ramtype is array (2**regbits-1 downto 0) of STD_LOGIC_VECTOR(width-1 downto 0);
        signal mem: ramtype;
begin
    -- three-ported register file
    -- read two ports combinationally
    -- write third port on rising edge of clock
    process(clk) begin
        if clk'event and clk = '1' then
            if write = '1' then mem(conv_integer(wa)) <= wd;
            end if;
        end if;
    end process;
    process(ra1, ra2) begin
        if (conv_integer(ra1) = 0) then rd1 <= conv_std_logic_vector(0, width); -- register 0 holds 0
        else rd1 <= mem(conv_integer(ra1));
        end if;
        if (conv_integer(ra2) = 0) then rd2 <= conv_std_logic_vector(0, width);
        else rd2 <= mem(conv_integer(ra2));
        end if;
    end process;
end;

architecture synth of zerodetect is
    signal i: integer;
    signal x: STD_LOGIC_VECTOR(width-1 downto 1);
begin -- N-bit AND of inverted inputs
    AllBits: for i in width-1 downto 1 generate
        LowBit: if i = 1 generate
                    A1: x(1) <= not a(0) and not a(1);
                end generate;
                OtherBits: if i /= 1 generate
                    Ai: x(i) <= not a(i) and x(i-1);
                end generate;
        end generate;
    y <= x(width-1);
end;
```

```
architecture synth of flop is
begin
    process(clk) begin
        if clk'event and clk = '1' then -- or use "if RISING_EDGE(clk) then"
            q <= d;
        end if;
    end process;
end;

architecture synth of flopen is
begin
    process(clk) begin
        if clk'event and clk = '1' then
            if en = '1' then q <= d;
            end if;
        end if;
    end process;
end;

architecture synchronous of flopenr is
begin
    process(clk) begin
        if clk'event and clk = '1' then
            if reset = '1' then
                q <= CONV_STD_LOGIC_VECTOR(0, width); -- produce a vector of all zeros
            elsif en = '1' then q <= d;
            end if;
        end if;
    end process;
end;

architecture synth of mux2 is
begin
    y <= d0 when s = '0' else d1;
end;

architecture synth of mux4 is
begin
    y <= d0 when s = "00" else
         d1 when s = "01" else
         d2 when s = "10" else
         d3;
end;
```

References

The IEEE Journal of Solid-State Circuits is abbreviated as *JSSC* because it is cited heavily. Most of the references in IEEE publications since 1988 can be obtained from `ieeexplore.ieee.org`.

[Acken83] J. Acken, "Testing for bridging faults (shorts) in CMOS circuits," *Proc. Design Automation Conf.*, 1983, pp. 717-718.

[Acosta95] A. Acosta, M. Valencia, A. Barriga, M. Bellido, and J. Huertas, "SODS: A new CMOS differential-type structure," *JSSC*, vol. 30, no. 7, July 1995, pp. 835-838.

[Afghahi90] M. Afghahi and C. Svensson, "A unified single-phase clocking scheme for VLSI systems," *JSSC*, vol. 25, no. 1, Feb. 1990, pp. 225-233.

[Allam00] M. Allam, M. Anis, and M. Elmasry, "High-speed dynamic logic styles for scaled-down CMOS and MTCMOS technologies," *Proc. Intl. Symp. Low Power Electronics and Design*, 2000, pp. 155-160.

[Allam01] M. Allam and M. Elmasry, "Dynamic current mode logic (DyCML): a new low-power high-performance logic style," *JSSC*, vol. 36, no. 3, March 2001, pp. 550-558.

[Alvandpour02] A. Alvandpour, R. Krishnamurthy, K. Soumyanath, and S. Borkar, "A sub-130-nm conditional keeper technique," *JSSC*, vol. 37, no. 5, May 2002, pp. 633-638.

[Amrutur98] B. Amrutur and M. Horowitz, "A replica technique for wordline and sense control in low-power SRAM's," *JSSC*, vol. 33, no. 8, Aug. 1998, pp. 1208-1219.

[Amrutur00] B. Amrutur and M. Horowitz, "Speed and power scaling of SRAM's," *JSSC*, vol. 35, no. 2, Feb. 2000, pp. 175-185.

[Amrutur01] B. Amrutur and M. Horowitz, "Fast low-power decoders for RAMs," *JSSC*, vol. 36, no. 10, Oct. 2001, pp. 1506-1515.

[Anastasakis02] D. Anastaskasis, R. Damiano, H. Ma, and T. Stanion, "A practical and efficient method for compare-point matching," *Proc. Design Automation Conf.*, June 2002, pp. 305-310.

[Anderson02] F. Anderson, J. Wells, and E. Berta, "The core clock system on the next generation Itanium microprocessor," *Proc. IEEE Intl. Solid-State Circuits Conf.*, Feb. 2002, pp. 146-147, 453.

[Ando80] H. Ando, "Testing VLSI with random access scan," *Digest of Papers COMPCON 80*, Feb. 1980, pp. 50-52.

[Artisan02] Artisan Components, *TSMC 0.18μm Process 1.8-Volt SAGE-X Standard Cell Library Databook*, Release 4.0, Feb. 2002.

[Ashenden01] P. Ashenden, *The Designer's Guide to VHDL*, 2nd ed., San Francisco, CA: Morgan Kaufmann, 2001.

[Ayers03] D. Ayers, "VLSI Power Delivery," EE371 Lecture Notes, Stanford University, April 29, 2003.

[Baghini02] M. Baghini and M. Desai, "Impact of technology scaling on metastability performance of CMOS synchronizing latches," *Proc. Intl. Conf. VLSI Design*, 2002, pp. 317-322.

[Bailey98] D. Bailey and B. Benschneider, "Clocking design and analysis for a 600-MHz Alpha microprocessor," *JSSC*, vol. 33, no. 11, Nov. 1998, pp. 1627-1633.

[Baker97] K. Baker and J. van Beers, "Shmoo plotting: the black art of IC testing," *IEEE Design and Test of Computers*, vol. 14, no. 3, July-Sept. 1997, pp. 90-97.

[Baker98] R. Jacob Baker, H. Li, and D. Boyce, *CMOS Circuit Design, Layout, and Simulation*, New York: Wiley-Interscience, 1998.

[Baker02] J. Baker, *CMOS Mixed-Signal Circuit Design*, Piscataway, NJ: IEEE Press, 2002.

[Bakoglu90] H. Bakoglu, *Circuits, Interconnections, and Packaging for VLSI*, Reading, MA: Addison-Wesley, 1990.

[Balamurugan01] G. Balamurugan and N. Shanbhag, "The twin-transistor noise-tolerant dynamic circuit technique," *JSSC*, vol. 36, no. 2, Feb. 2001, pp. 273-280.

[Barke88] E. Barke, "Line-to-ground capacitance calculation for VLSI: a comparison," *IEEE Trans. Computer-Aided Design*, vol. 7, no. 2, Feb. 1988, pp. 295-298.

[Baugh73] C. Baugh and B. Wooley, "A two's complement parallel array multiplication algorithm," *IEEE Trans. Computers*, vol. C-22, no. 12, Dec. 1973, pp. 1045-1047.

[Beaumont-Smith99] A. Beaumont-Smith, N. Burgess, S. Lefrere, and C. Lim, "Reduced latency IEEE floating-point adder architectures," *Proc. IEEE Symp. Computer Arithmetic*, April 1999, pp. 35-42.

[Beaumont-Smith01] A. Beaumont-Smith and C. Lim, "Parallel prefix adder design," *Proc. IEEE Symp. Computer Arithmetic*, 2001, pp. 218-225.

[Bedrij62] O. Bedrij, "Carry-select adder," *IRE Trans. Electronic Computers*, vol. 11, June 1962, pp. 340-346.

[Bernstein99] K. Bernstein, K. Carrig, C. Durham, P. Hansen, D. Hogenmiller, E. Nowak, and N. Rohrer, *High Speed CMOS Design Styles*, Boston: Kluwer Academic Publishers, 1999.

[Bernstein00] K. Bernstein and N. Rohrer, *SOI Circuit Design Concepts*, Boston: Kluwer Academic Publishers, 2000.

[Best03] R. Best, *Phase-Locked Loops: Design, Simulation, and Applications*, 5th ed., McGraw-Hill, 2003.

[Bewick94] G. Bewick, Fast Multiplication: Algorithms and Implementation, Ph.D. Thesis, Stanford University, CSL-TR-94-617, 1994.

[Black69] J. Black, "Electromigration—A brief survey and some recent results," *IEEE Trans. Electron Devices*, vol. ED-16, no. 4, April 1969, pp. 338-347.

[Blackburn96] J. Blackburn, L. Arndt, and E. Swartzlander, "Optimization of spanning tree carry lookahead adders," *Proc. 30th Asilomar Conf. Signals, Systems, and Computers*, vol. 1, 1996, pp. 177-181.

[Booth51] A. Booth, "A signed binary multiplication technique," *Quarterly J. Mechanics and Applied Mathematics*, vol. IV, part 2, June 1951, pp. 236-240.

[Borkar03] S. Borkar, T. Kamik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De, "Parameter variations and impact on circuits and microarchitecture," *Proc. Design Automation Conf.*, 2003, pp. 338-342.

[Bouldin03] D. Bouldin, A. Miller, and C. Tan, "Teaching custom integrated circuit design and verification," *Proc. Microelectronics Systems Education Conf.*, 2003, pp. 48-49.

[Bowhill95] W. Bowhill et al., "Circuit implementation of a 300-MHz 64-bit second-generation CMOS Alpha CPU," *Digital Technical Journal*, vol. 7, no. 1, 1995, pp. 100-115.

[Bowman99] K. Bowman, B. Austin, J. Eble, X. Tang, and J. Meindl, "A physical alpha-power law MOSFET model," *JSSC*, vol. 34, no. 10, Oct. 1999, pp. 1410-1414.

[Brent82] R. Brent and H. Kung, "A regular layout for parallel adders," *IEEE Trans. Computers*, vol. C-31, no. 3, March 1982, pp. 260-264.

[Brooks95] F. Brooks, *The Mythical Man-Month*, Boston: Addison-Wesley, 1995.

[Brown03] A. Brown, "Fast films," *IEEE Spectrum*, vol. 40, no. 2, Feb. 2003, pp. 36-40.

[Bugeja00] A. Bugeja and B. Song, "A self-trimming 14-b 100MSample/s CMOS DAC," *JSSC*, vol. 35, no. 12, Dec. 2000, pp. 1841-1852.

[Burks46] A. Burks, H. Goldstine, and J. von Neumann, *Preliminary discussion of the logical design of an electronic computing instrument, part 1*, vol. 1, Inst. Advanced Study, Princeton, NJ, 1946.

[Burleson98] W. Burleson, M. Ciesielski, F. Klass, and W. Liu, "Wave-pipelining: a tutorial and research survey," *IEEE Trans. VLSI*, vol. 6, no. 3, Sept. 1998, pp. 464-474.

[Calma84] Calma Corporation, GDS II Stream Format, July 1984.

[Candy76] J. Candy, W. Ninke, and B. Wooley, "A per-channel A/D converter having 15-segment μ-255 companding," *IEEE Transactions on Communications*, vol. 24, no. 1, Jan. 1976, pp. 33-42.

[Carr72] W. Carr and J. Mize, *MOS/LSI Design and Application*, New York: McGraw-Hill, 1972.

[Celik02] M. Celik, L. Pileggi, and A. Odabasioglu, *IC Interconnect Analysis*, Boston: Kluwer Academic Publishers, 2002.

[Chan90] P. Chan and M. Schlag, "Analysis and design of CMOS Manchester adders with variable carry-skip," *IEEE Trans. Computers*, vol. 39, no. 8, Aug. 1990, pp. 983-992.

[Chandrakasan01] A. Chandrakasan, W. Bowhill, and F. Fox, ed., *Design of High-Performance Microprocessor Circuits*, Piscataway, NJ: IEEE Press, 2001.

[Chaney73] T. Chaney and C. Molnar, "Anomalous behavior of synchronizer and arbiter circuits," *IEEE Trans. Computers*, vol. C-22, April 1973, pp. 421-422.

[Chaney83] T. Chaney, "Measured flip-flop responses to marginal triggering," *IEEE Trans. Computers*, vol. C-32, no. 12, Dec. 1983, pp. 1207-1209.

[Chang03] D. Chang and U. Moon, "A 1.4-V 25-MS/s pipelined ADC using opamp-reset switching technique," *JSSC*, vol. 38, no. 8, Aug. 2003, pp. 1401-1404.

[Chao89] H. Chao and C. Johnston, "Behavior analysis of CMOS D flip-flops," *JSSC*, vol. 24, no. 5, Oct. 1989, pp. 1454-1458.

[Chappell91] T. Chappell, B. Chappell, S. Schuster, J. Allan, S. Klepner, R. Joshi, and R. Franch, "A 2-ns cycle, 3.8-ns access 512-kb CMOS ECL SRAM with a fully pipelined architecture," *JSSC*, vol. 26, no. 11, Nov. 1991, pp. 1577-1585.

[Cheng99] Y. Cheng and C. Hu, *MOSFET Modeling & BSIM3 User's Guide*, Boston: Kluwer Academic Publishers, 1999.

[Cheng00y] Y. Cheng, C. Tsai, C. Teng, and S. Kang, *Electrothermal Analysis of VLSI Systems*, Boston: Kluwer Academic Publishers, 2000.

[Chern92] J. Chern, J. Huang, L. Arledge, P. Li, and P. Yang," Multilevel metal capacitance models for CAD design synthesis systems," *IEEE Electron Device Letters*, vol. 13, no. 1, Jan. 1992, pp. 32-34.

[Childs84] R. Childs, J. Crawford, D. House, and R. Noyce, "A processor family for personal computers," *Proc. IEEE*, vol. 72, no. 3, March 1984, pp. 363-376.

[Chillarige03] Y. Chillarige, S. Dubey, S. Sompur, and B. Wong, "A 399ps arithmetic logic unit (ALU) implemented using Propagate (P), Generate (G), and Kill (K) signals in push-pull style for a next generation UltraSparc microprocessor," *Proc. IEEE Custom Integrated Circuits Conf.*, 2003.

[Chinnery02] D. Chinnery and K. Keutzer, *Closing the Gap Between ASIC and Custom: Tools and techniques for high-performance ASIC design*, Boston: Kluwer Academic Publishers, 2002.

[Cho95] T. Cho and P. Gray, "A 10 b, 20 Msample/s, 35mW pipeline A/D converter," *JSSC*, vol. 30, no. 3, March 1995, pp. 166-172.

[Choi97] J. Choi, L. Jang, S. Jung and J. Choi, "Structured design of a 288-tap FIR filter by optimized partial product tree compression," *JSSC*, vol. 32, no. 3, March 1997, pp. 468-476.

[Choudhury97] M. Choudhury and J. Miller, "A 300 MHz CMOS microprocessor with multi-media technology," *Proc. IEEE Intl. Solid-State Circuits Conf.*, 1997, pp. 170-171.

[Chu86] K. Chu and D. Pulfrey, "Design procedures for diffential cascode voltage switch circuits," *JSSC*, vol. SC-21, no. 6, Dec. 1986, pp. 1082-1087.

[Chu87] K. Chu and D. Pulfrey, "A comparison of CMOS circuit techniques: differential cascode voltage switch logic versus conventional logic," *JSSC*, vol. SC-22, no. 4, Aug. 1987, pp. 528-532.

[Ciletti99] M. Ciletti, *Modeling, Synthesis, and Rapid Prototyping with the VERILOG HDL*, Upper Saddle River, NJ: Prentice Hall, 1999.

[Clark02] L. Clark, S. Demmons, N. Deutscher, and F. Ricci, "Standby power management for a 0.18μm microprocessor," *Proc. Intl. Symp. Low Power Electronics and Design*, Aug. 2002, pp. 7-12.

[Cobbold66] R. Cobbold, "Temperature effects on M.O.S. transistors," *Electronics Letters*, vol. 2, no. 6, June 1966, pp. 190-192.

[Cobbold70] R. Cobbold, *Theory and Application of Field Transistors*, New York: Wiley Interscience, 1970.

[Collins01] P. Collins, M. Arnold, and P. Avouris, "Engineering carbon nanotubes and nanotube circuits using electrical breakdown," *Science*, vol. 292, 27 April 2001, pp. 706-709.

[Colwell95] R. Colwell and R. Steck, "A 0.6 μm BiCMOS processor with dynamic execution," *Proc. IEEE Solid-State Circuits Conf.*, 1995, pp. 176-177.

[Cortadella92] J. Cortadella and J. Llaberia, "Evaluation of A+B=K conditions without carry propagation," *IEEE Trans. Computers*, vol. 41, no. 11, Nov. 1992, pp. 1484-1487.

[Covino97] J. Covino, "Dynamic CMOS circuit with noise immunity," US Patent 5,650,733, 1997.

[Crews03] M. Crews and Y. Yuenyongsgool, "Practical design for transferring signals between clock domains," *EDN Magazine*, Feb. 20, 2003, pp. 65-71.

[Curran02] B. Curran et al., "IBM eServer z900 high-frequency microprocessor technology, circuits, and design methodology," *IBM J. Research and Development*, vol. 46, no. 4/5, July/Sept. 2002, pp. 631-644.

[Dabral98] S. Dabral and T. Maloney, *Basic ESD and I/O Design*, New York: John Wiley & Sons, 1998.

[Dadda65] L. Dadda, "Some schemes for parallel multipliers," *Alta Frequenza*, vol. 34, no. 5, May 1965, pp. 349-356.

[Dally98] W. Dally and J. Poulton, *Digital Systems Engineering*, Cambridge, UK: Cambridge University Press, 1998.

[Davari99] B. Davari, "CMOS technology: present and future," *Symp. VLSI Circuits Digest Tech. Papers*, 1999, pp. 5-10.

[Dekker90] R. Dekker, F. Beenker, and L. Thijssen, "A realistic fault model and test algorithms for static random access memories," *IEEE Trans. Computer-Aided Design*, vol. 9, no. 6, June 1990, pp. 567-572.

[Deleganes02] D. Deleganes, J. Douglas, B. Kommandur, and M. Patyra, "Designing a 3GHz, 130nm, Intel Pentium 4 processor," *Symp. VLSI Circuits Digest Tech. Papers*, 2002, pp. 130-133.

[Delgado-Frias00] J. Delgado-Frias and J. Nyathi, "A high-performance encoder with priority lookahead," *IEEE Trans. Circuits and Systems I*, vol. 47, no. 9, Sept. 2000, pp. 1390-1393.

[Dennard68] R. Dennard, "Field-effect transistor memory," US Patent 3,387,286, 1968.

[Dennard74] R. Dennard et al., "Design of ion-implanted MOSFET's with very small physical dimensions," *JSSC*, vol. SC-9, no. 5, Oct. 1974, pp. 256-268.

[Dhanesha95] H. Dhanesha, K. Falakshahi, and M. Horowitz, "Array-of-arrays architecture for parallel floating point multiplication," *Proc. Conf. Advanced Research in VLSI*, 1995, pp. 150-157.

[Dike99] C. Dike and E. Burton, "Miller and noise effects in a synchronizing flip-flop," *JSSC*, vol. 34, no. 6, June 1999, pp. 849-855.

[Dingwall79] A. Dingwall, "Monolithic expandable 6 bit 20 MHz CMOS/SOS A/D converter," *JSSC*, vol. SC-14, no. 6, Dec. 1979, pp. 926-932.

[Dingwall85] A. Dingwall and V. Zazzu, "An 8-MHz CMOS subranging 8-bit A/D converter," *JSSC*, vol. SC-20, no. 6, Dec. 1985, pp. 1138-1143.

[Dobbalaere95] I. Dobbalaere, M. Horowitz, and A. El Gamal, "Regenerative feedback repeaters for programmable interconnect," *JSSC*, vol. 30, no. 11, Nov. 1995, pp. 1246-1253.

[Dobberpuhl92] D. Dobberpuhl et al., "A 200-MHz 64-b dual-issue CMOS microprocessor," *JSSC*, vol. 27, no. 11, Nov. 1992, pp. 1555-1867.

[Dobson95] J. Dobson and G. Blair, "Fast two's complement VLSI adder design," *Electronics Letters*, vol. 31, no. 20, Sept. 1995, pp. 1721-1722.

[Donnay03] S. Donnay and G. Gielen, eds., *Substrate Noise Coupling in Mixed-Signal ASICs*, Boston: Kluwer Academic Publishers, 2003.

[Donovan02] C. Donovan and M. Flynn, "A "digital" 6-bit ADC in 0.25μm CMOS," *JSSC*, vol. 37, no. 3, March 2002, pp. 432-437.

[Doyle91] B. Doyle, B. Fishbein, and K. Mistry, "NBTI-enhanced hot carrier damage in p-channel MOSFETs," *Proc. Intl. Electron Devices Meeting*, 1991, pp. 529-532A.

[Draper97] D. Draper et al., "Circuit techniques in a 266-MHz MMX-enabled processor," *JSSC*, vol. 32, no. 11, Nov. 1997, pp. 1650-1664.

[D'Souza96] G. D'Souza, "Dyanmic logic circuit with reduced charge leakage," US Patent 5,483,181, 1996.

[Edwards93] B. Edwards, A. Corry, N. Weste and C. Greenberg, "A single-chip video ghost canceller," *JSSC*, vol. 28, no. 3, March 1993, pp. 379-383..

[Eichelberger78] E. Eichelberger and T. Williams, "A logic design structure for LSI testability," *J. Design Automation and Fault Tolerant Computing*, vol. 2, no. 2, May 1978, pp. 165-178.

[Elmore48] W. Elmore, "The transient response of damped linear networks with particular regard to wideband amplifiers," *J. Applied Physics*, vol. 19, no. 1, Jan. 1948, pp. 55-63.

[Ercegovac04] M. Ercegovac and T. Lang, *Digital Arithmetic*, San Francisco: Morgan Kaufmann, 2004.

[Estreich82] D. Estreich and R. Dutton, "Modeling latch-up in CMOS integrated circuits," *IEEE Trans. Computer-Aided Design*, vol. CAD-1, no. 4, Oct. 1982, pp. 157-162.

[Faggin96] F. Faggin, M. Hoff, S. Mazor, and M. Shima, "The history of the 4004," *IEEE Micro*, vol. 16, no. 6, Dec. 1996, pp. 10-20.

[Fahim02] A. Fahim and M. Elmasry, "Low-power high-performance arithmetic circuits and architectures," *JSSC*, vol. 37, no. 1, Jan. 2002, pp. 90-94.

[Fetzer02] E. Fetzer, M. Gibson, A. Klein, N. Calick, C. Zhu, E. Busta, and B. Mohammad, "A fully bypassed six-issue integer datapath and register file on the Itanium-2 microprocessor," *JSSC*, vol. 37, no. 11, Nov. 2002, pp. 1433-1440.

[Flannagan85] S. Flannagan, "Synchronization reliability in CMOS technology," *JSSC*, vol. SC-20, no. 4, Aug. 1985, pp. 880-882.

[Flynn01] M. Flynn and S. Oberman, *Advanced Computer Arithmetic Design*, New York: John Wiley & Sons, 2001.

[Foty96] D. Foty, *MOSFET Modeling with SPICE: Principles and Practices*, Upper Saddle River, NJ: Prentice Hall, 1996.

[Friedman84] V. Friedman and S. Liu, "Dynamic logic CMOS circuits," *JSSC*, vol. SC-19, no. 2, April 1984, pp. 263-266.

[Frohman69] D. Frohman-Bentchkowsky and A. Grove, "Conductance of MOS transistors in saturation," *IEEE Trans. Electron Devices*, vol. ED-16, no. 1, Jan. 1969, pp. 108-113.

[Frowerk77] R. Frowerk, "Signature Analysis: A New Digital Field Service Method," *Hewlett Packard Journal*, May 1977, pp. 2-8.

[Gajski83] D. Gajski and R. Kuhn, "New VLSI tools," *Computer*, vol. 16, no. 12, Dec. 1983, pp. 11-14.

[Galiay80] J. Galiay, Y. Crouzet, and M. Vergniault, "Physical versus logical fault models MOS LSI circuits: impact on their testability," *IEEE Trans. Computers*, vol. C-29, no. 6, June 1980, pp. 527-531.

[Gauthier02] C. Gauthier and B. Amick, "Inductance: Implications and solutions for high-speed digital circuits: the chip electrical interface," *Proc. IEEE Intl. Solid-State Circuits Conf.*, vol.2, 2002, pp. 565-565.

[Geannopoulos98] G. Geannopoulos and X. Dai, "An adaptive digital deskewing circuit for clock distribution networks," *Proc. IEEE Intl. Solid-State Circuits Conf.*, 1998, pp. 400-401.

[Gelsinger01] P. Gelsinger, "Microprocessors for the new millennium: challenges, opportunities, and new frontiers," *Proc. IEEE Intl. Solid-State Circuits Conf.*, 2001, pp. 22-25.

[George96] S. George, A. Ott, and J. Klaus, "Surface chemistry for atomic layer growth," *J. Phys. Chem.*, vol. 100, 1996, pp. 13121-13131.

[Gerosa94] G. Gerosa et al., "A 2.2 W, 80 MHz superscalar RISC microprocessor," *JSSC*, vol. 29, no. 12, Dec. 1994, pp. 1440-1452.

[Gielis91] G. Gielis, R. van de Plassche, and J. van Valburg, "A 540-MHz 10-b polar-to-cartesian converter," *JSSC*, vol. 26, no. 11, Nov. 1991, pp. 1645-1650.

[Gieseke97] B. Gieseke et al., "A 600-MHz superscalar RISC microprocessor with out-of-order execution," *Proc. IEEE Intl. Solid-State Circuits Conf.*, 1997, pp. 176-177, 451.

[Glasser85] L. Glasser and D. Dobberpuhl, *The Design and Analysis of VLSI Circuits*, Reading, MA: Addison Wesley, 1985.

[Golden99] M. Golden et al., "A seventh-generation x86 microprocessor," *JSSC*, vol. 34, no. 11, Nov. 1999, pp. 1466-1477.

[Golomb81] S. Golomb, *Shift Register Sequences*, Revised Edition, Laguna Hills, CA: Aegean Park Press, 1981.

[Gonclaves83] N. Gonclaves and H. DeMan, "NORA: a racefree dynamic CMOS technique for pipelined logic structures," *JSSC*, vol. SC-18, no. 3, June 1983, pp. 261-266.

[Gonzalez96] R. Gonzalez and M. Horowitz, "Energy dissipation in general purpose microprocessors," *JSSC*, vol. 31, no. 9, Sept. 1996, pp. 1277-1284.

[Gray53] F. Gray, "Pulse code communications," US Patent 2,632,058, 1953.

[Gray01] P. Gray, P. Hurst, S. Lewis, and R. Meyer, *Analysis and Design of Analog Integrated Circuits*, 4th ed., New York: John Wiley & Sons, 2001.

[Grayver98] E. Grayver and B. Daneshrad, "Direct digital frequency synthesis using a modified CORDIC," *Proc. IEEE Intl. Symp. Circuits and Systems*, May 1998, pp. 241-244.

[Greenhill02] D. Greenhill, Design for Reliability Tutorial, *Proc. IEEE Intl. Solid-State Circuits Conf.*, 2002.

[Griffin83] W. Griffin and J. Hiltebeitel, "CMOS 4-way XOR circuit," *IBM Technical Disclosure Bulletin*, vol. 25, no. 11B, April 1983, pp. 6066-6067.

[Gronowski96] P. Gronowski et al., "A 433-MHz 64-b quad-issue RISC microprocessor," *JSSC*, vol. 31, no. 11, Nov. 1996, pp. 1687-1696.

[Gronowski98] P. Gronowski, W. Bowhill, R. Preston, M. Gowan, and R. Allmon, "High-performance microprocessor design," *JSSC*, vol. 33, no. 5, May 1998, pp. 676-686.

[Grosspietsch92] K. Grosspietsch, "Associative processors and memories: a survey," *IEEE Micro*, vol. 12, no. 3, June 1992, pp. 12-19.

[Grotjohn86] T. Grotjohn and B. Hoefflinger, "Sample-set differntial logic (SSDL) for complex high-speed VLSI," *JSSC*, vol. SC-21, no. 2, April 1986, pp. 367-369.

[Gupta03] S. Gupta and V. Fong, "A 64-MHz clock-rate sigma-delta ADC with 88-dB SNDR and –105-dB IM3 distortion at a 1.5-MHz signal frequency," *JSSC*, vol. 37, no. 12, Dec. 2002, pp. 1653-1661.

[Gutierrez01] E. Gutierrez, J. Deen, and C. Claeys (eds.), *Low Temperature Electronics: Physics, Devices, Circuits, and Applications*, New York: Academic Press, 2001.

[Gutnik00] V. Gutnik and A. Chandrakasan, "Active GHz clock network using distributed PLLs," *JSSC*, vol. 35, no. 11, Nov. 2000, pp. 1553-1560.

[Guyot87] A. Guyot, B. Hochet, and J. Muller, "A way to build efficient carry-skip adders," *IEEE Trans. Computers*, vol. 36, no. 10, Oct. 1987, pp. 1144-1152.

[Guyot97] A. Guyot and S. Abou-Samra, "Modeling power consumption in arithmetic operators," *Microelectronic Engineering*, vol. 39, 1997, pp. 245-253.

[Hamming50] R. Hamming, "Error detecting and error correcting codes," *Bell Systems Technical Journal*, vol. 29, pp. 147-160.

[Hamzaoglu02] F. Hamzaoglu and M. Stan, "Circuit-level techniques to control gate leakage for sub-100nm CMOS," *Proc. Intl. Symp. Low Power Electronics and Design*, 2002, pp. 60-63.

[Han87] T. Han and D. Carlson, "Fast area-efficient VLSI adders," *Proc. IEEE Symp. Computer Arithmetic*, 1987, pp. 49-56.

[Harame01a] D. Harame and B. Meyerson, "The early history of IBM's SiGe mixed signal technology," *IEEE Transactions on Electron Devices*, vol. 48, no. 11, Nov. 2001, pp. 2555-2567.

[Harame01b] D. Harame et al., "Current status and future trends of SiGe BiCMOS technology," *IEEE Transactions on Electron Devices*, vol. 48, no. 11, Nov. 2001, pp. 2575-2594.

[Haring96] R. Haring et al., "Self-resetting logic register and incrementer," *Symp. VLSI Circuits Digest Tech. Papers*, 1996, pp. 18-19.

[Harrer02] H. Harrer et al., "First and second-level packaging for the IBM eServer z900," *IBM J. Research and Development*, vol. 46, no. 4/5, July/Sept. 2002, pp. 397-420.

[Harris97] D. Harris and M. Horowitz, "Skew-tolerant domino circuits," *JSSC*, vol. 32, no. 11, Nov. 1997, pp. 1702-1711.

[Harris99] D. Harris, M. Horowitz, and D. Liu, "Timing analysis including clock skew," *IEEE Trans. Computer-Aided Design*, vol. 18, no. 11, Nov. 1999, pp. 1608-1618.

[Harris01a] D. Harris, *Skew-Tolerant Circuit Design*, San Francisco, CA: Morgan Kaufmann, 2001.

[Harris01b] D. Harris and S. Naffziger, "Statistical clock skew modeling with data delay variations," *IEEE Trans. VLSI*, vol. 9, no. 6, Dec. 2001, pp. 888-898.

[Harris03] D. Harris, "A taxonomy of prefix networks," *Proc. 37th Asilomar Conf. Signals, Systems, and Computers*, 2003, pp. 2213-2217.

[Harrison03] J. Harrison and N. Weste, "A 500 MHz CMOS anti-alias filter using feed-forward op-amps with local common-mode feedback," *Proc. IEEE Intl. Solid-State Circuits Conf.*, Feb. 2003, pp. 132-133.

[Hashemian92] R. Hashemian and C. Chen, "A new parallel technique for design of decrement/increment and two's complement circuits," *Proc. IEEE Midwest Symp. Circuits and Systems*, vol. 2, 1992, pp. 887-890.

[Hashimoto02] T. Hashimoto et al., "Integration of a 0.13-μm CMOS and a high performance self-aligned SiGe HBT featuring low base resistance," *Proc. Intl. Electron Devices Meeting*, Dec. 2002, pp. 779-782.

[Hatamian86] M. Hatamian and G. Cash, "A 70-MHz 8-bit x 8-bit parallel pipelined multiplier in 2.5-μm CMOS," *JSSC*, vol. 21, no. 4, Aug. 1986, pp. 505-513.

[Haykin00] S. Haykin, *Digital Communications*, New York: John Wiley & Sons, 2000.

[Hazucha00] P. Hazucha, C. Svensson, and S. Wender, "Cosmic-ray soft error rate characterization of a standard 0.6-μm CMOS process," *JSSC*, vol. 35, no. 10, Oct. 2000, pp. 1422-1429.

[Heald93] R. Heald and J. Holst, "A 6-ns cycle 256 kb cache memory and memory management unit," *JSSC*, vol. 28, no. 11, Nov. 1993, pp. 1078-1083.

[Heald98] R. Heald et al., "64-Kbyte sum-addressed-memory cache with 1.6-ns cycle and 2.6-ns latency," *JSSC*, vol. 33, no. 11, Nov. 1998, pp. 1682-1689.

[Heald00] R. Heald et al., "A third-generation SPARC v9 64-b microprocessor," *JSSC*, vol. 35, no. 11, Nov. 2000, pp. 1526-1538.

[Hedenstierna87] N. Hedenstierna and K. Jeppson, "CMOS circuit speed and buffer optimization," *IEEE Trans. Computer-Aided Design*, vol. CAD-6, no. 2, March 1987, pp. 270-281.

[Heikes94] C. Heikes, "A 4.5mm^2 multiplier array for a 200MFLOP pipelined coprocessor," *Proc. IEEE Intl. Solid-State Circuits Conf.*, 1994, pp. 290-291.

[Heller84] L. Heller, W. Griffin, J. Davis and N. Thoma, "Cascode voltage switch logic: a differential CMOS logic family," *Proc. IEEE Intl. Solid-State Circuits Conf.*, 1984, pp. 16-17.

[Hennessy90] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, San Mateo, CA: Morgan Kaufmann Publishers, Inc. 1990.

[Hess94] C. Hess and L. Weiland, "Drop in process control checkerboard test structure for efficient online process characterization and defect problem debugging," *Proc. IEEE Int. Conf. Microelectronic Test Structures*, vol. 7, March, 1994, pp. 152-159.

[Hidaka89] H. Hidaka, K. Fujishima, Y. Matsuda, M. Asakura, and T. Yoshihara, "Twisted bit-line architectures for multi-megabit DRAM's," *JSSC*, vol. 24, no. 1, Feb. 1989, pp. 21-27.

[Hill68] C. Hill, "Noise margin and noise immunity in logic circuits," *Microelectronics*, vol. 1, April 1968, pp. 16-21.

[Hinton01] G. Hinton et al., "A 0.18-μm CMOS IA-32 processor with a 4-GHz integer execution unit," *JSSC*, vol. 36, no. 11, Nov. 2001, pp. 1617-1627.

[Hisamoto98] D. Hisamoto et al., "A folded-channel MOSFET for deep-sub-tenth micron era," *Tech. Digest Intl. Electron Devices Meeting*, San Francisco, Dec. 1998, pp. 1032-1034..

[Ho98] R. Ho, B. Amrutur, K. Mai, B. Wilburn, T. Mori, and M. Horowitz, "Application of on-chip samplers for test and measurement of integrated circuits," *Symp. VLSI Circuits Digest Tech. Papers*, 1998, pp. 138-139.

[Ho01] R. Ho, K. Mai, and M. Horowitz, "The future of wires," *Proc. IEEE*, vol. 89, no. 4, April 2001, pp. 490-504.

[Ho03a] R. Ho, K. Mai, and M. Horowitz, "Efficient on-chip global interconnects," *Symp. VLSI Circuits Digest Tech. Papers*, 2003, pp. 271-274.

[Ho03b] R. Ho, K. Mai, and M. Horowitz, "Managing wire scaling: a circuit perspective," *Proc. IEEE Interconnect Technology Conf.*, 2003, pp. 177-179.

[Hoeneisen72] B. Hoeneisen and C. Mead, "Fundamental limitations in Microelectronics-I. MOS technology," *Solid-State Electronics*, vol. 15, 1972, pp. 819-829.

[Hoeschele94] D. Hoeschele, *Analog-to-Digital and Digital-to-Analog Conversion Techniques*, New York: Wiley-Interscience, 1994.

[Hoppe90] B. Hoppe, G. Neuendorf, D. Schmitt-Landsiedel, and W. Specks, "Optimization of high-speed CMOS logic circuits with analytical models for signal delay, chip area, and dynamic power dissipation," *IEEE Trans. Computer-Aided Design*, vol. 9, no. 3, March 1990, pp. 236-247.

[Horowitz83] M. Horowitz and R. Dutton, "Resistance extraction from mask layout data," *IEEE Trans. Computer-Aided Design*, vol. CAD-2, no. 3, July 1983, pp. 145-150.

[Horowitz87] M. Horowitz et al., "MIPS-X: a 20-MIPS peak, 32-bit microprocessor with on-chip cache," *JSSC*, vol. SC-22, no. 5, Oct. 1987, pp. 790-799.

[Horowitz02] M. Horowitz, EE371 Course Notes, Stanford University, Spring 2002, www.stanford.edu/class/ee371

[Horstmann89] J. Horstmann, H. Eichel, and R. Coates, "Metastability behavior of CMOS ASIC flip-flops in theory and test," *JSSC*, vol. 24, no. 1, Feb. 1989, pp. 146-157.

[Hrishikesh02] M. Hrishikesh et al., "The optimal logic depth per pipeline stage is 6 to 8 FO4 inverter delays," *Proc. Intl. Symp. Computer Architecture*, 2002, pp. 14-24.

[Hsu91] W. Hsu, B. Sheu, and S. Gowda, "Design of reliable VLSI circuits using simulation techniques," *JSSC*, vol. 26, no. 3, March 1991, pp. 452-457.

[Hsu92] W. Hsu, B. Sheu, S. Gowda and C. Hwang, "Advanced integrated-circuit reliability simulation including dynamic stress effects," *JSSC*, vol. 27, no. 3, March 1992, pp. 247-257.

[Hu90] Y. Hu and S. Chen, "GM_Plan: A Gate Matrix Layout Algorithm Based on Artificial Intelligence Planning Techniques," *IEEE Trans. Computer-Aided Design*, vol. 9, no. 8, Aug. 1990, pp. 836-845.

[Hu92] C. Hu, "IC reliability simulation," *JSSC*, vol. 27, no. 3, March 1992, pp. 241-246.

[Hu95] C. Hu, K. Rodbell, T. Sullivan, K. Lee, and D. Bouldin, "Electromigration and stress-induced voiding in fine Al and Al-alloy thin-film lines," *IBM J. Research and Development*, vol. 39, no. 4, July 1995, pp. 465-497.

[Huang00] Z. Huang and M. Ercegovac, "Effect of wire delay on the design of prefix adders in deep-submicron technology," *Proc. 34th Asilomar Conf. Signals, Systems, and Computers*, vol. 2, 2000, pp. 1713-1717.

[Huang02] C. Huang, J. Wang and Y. Huang, "Design of high-performance CMOS priority encoders and incrementer/decrementers using multilevel lookahead and multilevel folding techniques," *JSSC*, vol. 37, no. 1, Jan. 2002, pp. 63-76.

[Huang03] X. Huang et al., "Loop-based interconnect modeling and optimization approach for multigigahertz clock network design," *JSSC*, vol. 38, no. 3, March 2003, pp. 457-463.

[Huh98] Y. Huh, Y. Sung, and S. Kang, "A study of hot-carrier-induced mismatch drift: a reliability issue for VLSI circuits," *JSSC*, vol. 33, no. 6, June 1998, pp. 921-927.

[Huitema03] E. Huitema et al., "Plastic transistors in active-matrix displays," *Proc. IEEE Intl. Solid-State Circuits Conf.*, Feb. 2003, pp. 380-381.

[Hwang89] I. Hwang and A. Fisher, "Ultrafast compact 32-bit CMOS adders in multiple-output domino logic," *JSSC*, vol. 24, no. 2, April 1989, pp. 358-369.

[Hwang99a] W. Hwang, R. Joshi, and W. Henkels, "A 500-MHz, 32-Word x 64-bit, eight-port self-resetting CMOS register file," *JSSC*, vol. 34, no. 1, Jan. 1999, pp. 56-67.

[Hwang99b] W. Hwang, G. Gristede, P. Sanda, S. Wang, and D. Heidel, "Implementation of a self-resetting CMOS 64-bit parallel adder with enhanced testability," *JSSC*, vol. 34, no. 8, Aug. 1999, pp. 1108-1117.

[Hwang02] D. Hwang, F. Dengwei, A. Willson, Jr, "A 400-MHz processor for the efficient conversion of rectangular to polar coordinates for digital communications applications," *Symp. VLSI Circuits Digest Tech. Papers*, June 2002, pp. 248-51.

[IEEE1076-02] *IEEE Standard 1076-2002, VHDL Language Reference Manual.*

[IEEE1149.1-01] *IEEE Standard 1149.1-2001, Test Access Port and Boundary-Scan Architecture.*

[IEEE1164-93] *IEEE Standard 1164-1993, Multivalue Logic System for VHDL Model Interoperability (Std_logic_1164).*

[IEEE1364-01] *IEEE Standard 1364-2001, Verilog Hardware Description Language.*

[Ingino01] J. Ingino and V. von Kaenel, "A 4-GHz clock system for a high-performance system-on-a-chip design," *JSSC*, vol. 36, no. 11, Nov. 2001, pp. 1693-1698.

[Intel03] Intel Corporation, *Microprocessor Quick Reference Guide*, http://www.intel.com/pressroom/kits/quickreffam.htm, 2003.

[Ismail99] Y. Ismail, E. Friedman, and J. Neves, "Figures of merit to characterize the importance of on-chip interconnect," *IEEE Trans. VLSI*, vol. 7, no. 4, Dec. 1999, pp. 442-449.

[Itoh01k] K. Itoh, *VLSI Memory Chip Design*, Berlin: Springer-Verlag, 2001.

[Itoh01n] N. Itoh et al., "A 600-MHz 54 x 54-bit multiplier with rectangular-styled Wallace tree," *JSSC*, vol. 36, no. 2, Feb. 2001, pp. 249-257.

[Jamal02] S. Jamal et al., "A 10-b 120-Msample/s time-interleaved analog-to-digital converter with digital background calibration," *JSSC*, vol. 37, no. 12, Dec. 2002, pp. 1618-1627.

[Jayasumana91] A. Jayasumana, Y. Malaiya, and R. Rajsuman," Design of CMOS circuits for stuck-open fault testability," *JSSC*, vol. 26, no. 1, Jan. 1991, pp. 58-61.

[Jiang03] X. Jiang, Z. Wang, and F. Chang, "A 2 GS/s 6 b ADC in 0.18 μm CMOS," *Proc. IEEE Intl. Solid-State Circuits Conf.*, Feb. 2003, pp.322-323.

[Ji-ren87] Y. Ji-ren, I. Karlsson, and C. Svensson, "A true single-phase-clock dynamic CMOS circuit technique," *JSSC*, vol. SC-22, no. 5, Oct. 1987, pp. 899-901.

[Johns96] D. Johns and K. Martin, *Analog Integrated Circuit Design*, New York: John Wiley & Sons, 1996.

[Johnson88] M. Johnson, "A symmetric CMOS NOR gate for high-speed applications," *JSSC*, vol. SC-23, no. 5, Oct. 1988, pp. 1233-1236.

[Johnson91] B. Johnson, T. Quarles, A. Newton, D. Pederson, A. Sangiovanni-Vincentelli, *SPICE3 Version 3e User's Manual*, UC Berkeley, April 1991.

[Johnson93] H. Johnson and M. Graham, *High-Speed Digital Design: A Handbook of Black Magic*, Upper Saddle River, NJ: Prentice Hall, 1993.

[Josephson02] D. Josephson, "The manic depression of microprocessor debug," *Proc. Intl. Test Conf.*, 2002, pp. 657-663.

[Juhnke95] T. Juhnke and H. Klar, "Calculation of the soft error rate of submicron CMOS logic circuits," *JSSC*, vol. 30, no. 7, July 1995, pp. 830-834.

[Jung01] S. Jung, S. Yoo, K. Kim, and S. Kang, "Skew-tolerant high-speed (STHS) domino logic," *Proc. IEEE Intl. Symp. Circuits and Systems*, 2001, pp. 154-157.

[Kamon94] M. Kamon, J. Tsuk, and J. White, "FASTHENRY: a multipole-accelerated 3-D inductance extraction program," *IEEE Trans. Microwave Theory and Techniques*, vol. 42, no. 9, Sept. 1994, pp. 1750-1758.

[Kang03] S. Kang and Y. Leblebici, *CMOS Digital Integrated Circuits*, 3rd ed., Boston: McGraw Hill, 2003.

[Kantabutra91] V. Kantabutra, "Designing optimum carry-skip adders," *Proc. IEEE Symp. Computer Arithmetic*, 1991, pp. 146-153.

[Kantabutra93] V. Kantabutra, "A recursive carry-lookahead / carry-select hybrid adder," *IEEE Trans. Computers*, vol. 42, no. 12, Dec. 1993, pp. 1495-1499.

[Kao01] J. Kao and A. Chandrasakan, "MTCMOS sequential circuits," *Proc. 27th European Solid-State Circuits Conf.*, 2001, pp. 332-335.

[Kappes03] M. Kappes, "A 2.2-mW CMOS bandpass continuous-time delta-sigma ADC with 68 dB of dynamic range and 1-MHz bandwidth for wireless applications," *JSSC*, vol. 38, no. 7, July 2003, pp. 1098-1104.

[Karnik01] T. Karnik, B. Bloechel, K. Soumyanath, V. De, and S. Borkar, "Scaling trends of cosmic rays induced soft errors in static latches beyond 0.18m," *Symp. VLSI Circuits Digest Tech. Papers*, 2001, pp. 61-62.

[Keeth01] B. Keeth and J. Baker, *DRAM Circuit Design: A Tutorial*, Piscataway, NJ: IEEE Press, 2001.

[Keshavarzi01] A. Keshavarzi et al., "Effectivness of reverse body bias for leakage control in scaled dual Vt CMOS ICs," *Proc. Intl. Symp. Low Power Electronics and Design*, 2001, pp. 207-212.

[Keyes70] R. Keyes, E. Harris, and K. Konnerth, "The role of low temperatures in the operation of logic circuitry," *Proc. IEEE*, vol. 58, no. 12, Dec. 1970, pp. 1914-1932.

[Kielkowski95] R. Kielkowski, SPICE: *Practical Device Modeling*, Boston: McGraw-Hill, 1995.

[Kilburn59] T. Kilburn, D. Edwards, and D. Aspinall, "Parallel addition in a digital computer - a new fast carry," *Proc. IEE*, vol. 106B, 1959, pp. 460-464.

[Kim02] Y. Kim et al., "50 nm gate length logic technology with 9-layer Cu interconnects for 90 nm node SoC applications," *Proc. Intl. Electron Devices Meeting*, 2002, p. 69.

[Kio01] S. Kio, L. McMurchie, and C. Sechen, "Application of output prediction logic to differential CMOS," *Proc. IEEE Computer Society Workshop on VLSI*, 2001, pp. 57-65.

[Klass99] F. Klass et al., "A new family of semidynamic and dynamic flip-flops with embedded logic for high-performance processors," *JSSC*, vol. 34, no. 5, May 1999, pp. 712-716.

[Klaus98] J. Klaus, A. Ott, A. Dillon, and S. George, "Atomic layer controlled growth of Si_3N_4 films using sequential surface reactions," *Surf. Sci.*, vol. 418, 1998, pp. L14-L19.

[Knebel98] D. Knebel et al., "Diagnosis and characterization of timing-related defects by time-dependent light emission," *IEEE Intl. Test Conf.*, 1998, pp. 733-739.

[Knowles01] S. Knowles, "A family of adders," *Proc. IEEE Symp. Computer Arithmetic*, 2001, pp. 277-284.

[Koenemann79] B. Koenemann, J. Mucha, and G. Zwiehoff, "Built-in logic block observation techniques," *Proc. Intl. Test Conf.*, Oct. 1979, pp. 37-41.

[Kogge73] P. Kogge and H. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations," *IEEE Trans. Computers*, vol. C-22, no. 8, Aug. 1973, pp. 786-793.

[Kong01w] W. Kong, R. Venkatraman, R. Castagnetti, F. Duan, and S. Ramesh, "High-density and high-performance 6T-SRAM for system-on-chip in 130 nm CMOS technology," *Tech. Digest Symp. VLSI Technology*, 2001, pp. 105-106.

[Koren02] I. Koren, *Computer Arithmetic Algorithms*, 2nd ed., Natick, Mass.: A.K. Peters, 2002.

[Kovacs98] G. Kovacs, *Micromachined Transducers Sourcebook*, Boston: McGraw-Hill, 1998.

[Kozu96] S. Kozu et al., "A 100 MHz 0.4W RISC processor with 200 MHz multiply-adder, using pulse-register technique," *Proc. IEEE Intl. Solid-State Circuits Conf.*, 1996, pp. 140-141.

[Krambeck82] R. Krambeck, C. Lee, and H. Law, "High speed compact circuits with CMOS," *JSSC*, vol. SC-17, no. 3, June 1982, pp. 614-619.

[Kumar94] R. Kumar, "ACMOS: an adaptive CMOS high performance logic," *Electronics Letters*, vol. 30, no. 6, March 1994, pp. 483-484.

[Kumar01] R. Kumar, "Interconnect and noise immunity design for the Pentium 4 processor," *Intel Technology Journal*, vol. 5, no. 1, Q1 2001, pp. 1-12.

[Kuo01] J. Kuo and S. Lin, *Low-Voltage SOI CMOS VLSI Devices and Circuits*, New York: Wiley Interscience, 2001.

[Kurd01] N. Kurd, J. Barkatullah, R. Dizon, T. Fletcher, and P. Madland, "A multigigahertz clocking scheme for the Pentium 4 microprocessor," *JSSC*, vol. 36, no. 11, Nov. 2001, pp. 1647-1653.

[Kuroda96] T. Kuroda et al., "A 0.9-V, 150-MHz, 10-mW, 4 mm^2, 2-D discrete cosine transform core processor with variable threshold-voltage (VT) scheme," *JSSC*, vol. 31, no. 11, Nov. 1996, pp. 1770-1779.

[Ladner80] R. Ladner and M. Fischer, "Parallel prefix computation," *J. ACM*, vol. 27, no. 4, Oct. 1980, pp. 831-838.

[Lai97] F. Lai and W. Hwang, "Design and implementation of differential cascode voltage switch with pass-gate (DCVSPG) logic for high-performance digital systems," *JSSC*, vol. 32, no. 4, April 1997, pp. 563-573.

[Lakshmanan01] A. Lakshmanan and R. Sridhar, "Input controlled refresh for noise tolerant dynamic circuits," *Proc. 14th IEEE Intl. ASIC/SOC Conf.*, 2001, pp. 129-133.

[Larsson94] P. Larsson and C. Svensson, "Impact of clock slope on true single phase clocked (TSPC) CMOS circuits," *JSSC*, vol. 29, no. 6, June 1994, pp. 723-726.

[Larsson97] P. Larsson, "Parasitic resistance in an MOS transistor used as on-chip decoupling capacitance," *JSSC*, vol. 32, no. 4, April 1997, pp. 574-576.

[Lasserre99] F. Lasserre et al., "Laser beam backside probing of CMOS integrated circuits," *Microelectronics and Reliability*, June 1999, vol. 39, no. 6, pp. 957-961.

[Leblebici96] Y. Leblebici, "Design considerations for CMOS digital circuits with improved hot-carrier reliability," *JSSC*, vol. 31, no. 7, July 1996, pp. 1014-1024.

[Lee02] S. Lee and H. Yoo, "Race logic architecture (RALA): a novel logic concept using the race scheme of input variables," *JSSC*, vol. 37, no. 2, Feb. 2002, pp. 191-201.

[Lee04] T. Lee, *The Design of CMOS Radio-Frequency Integrated Circuits*, 2nd ed., Cambridge: Cambridge University Press, 2004.

[Lee86] C. Lee and E. Szeto, "Zipper CMOS," *IEEE Circuits and Systems Magazine*, May 1986, pp. 10-16.

[Lee92] K. Lee and M. Breuer, "Design and test rules for CMOS circuits to facilitate IDDQ testing of bridging faults," *IEEE Trans. On CAD of Integrated circuits*, vol. 11, no. 5, May 1992, pp. 659-670.

[Lee98] M. Lee, "A multilevel parasitic interconnect capacitance modeling and extraction for reliable VLSI on-chip clock delay evaluation," *JSSC*, vol. 33, no. 4, April 1998, pp. 657-661.

[Lehman61] M. Lehman and N. Burla, "Skip technique for high-speed carry-propagation in binary arithmetic units," *IRE Trans. Electronic Computers*, vol. 10, Dec. 1961, pp. 691-698.

[Leighton92] F. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays; Trees; Hypercubes*, San Francisco: Morgan Kaufmann, 1992.

[Leung02] B. Leung, *VLSI for Wireless Communication*, Upper Saddle River, NJ: Prentice Hall, 2002.

[Lev95] L. Lev et al., "A 64-b microprocessor with multimedia support," *JSSC*, vol. 30, no. 11, Nov. 1995, pp. 1227-1238.

[Li03] Y. Li and E. Sanchez-Sinencio, "A wide input bandwidth 7-bit 300-Msample/s folding and current-mode interpolating ADC," *JSSC*, vol. 38, no. 8, Aug. 2003, pp.1405-1410.

[Liew90] B. Liew, N. Cheung, and C. Hu, "Projecting interconnect electromigration lifetime for arbitrary current waveforms," *IEEE Trans. Electron Devices*, vol. 37, no. 5, May 1990, pp. 1343-1351.

[Lim72] R. Lim, "A barrel switch design," *Computer Design*, Aug. 1972, pp. 76-78.

[Lin83] S. Lin and D. Costello, *Error Control Coding: Fundamentals and Applications*, Upper Saddle River, NJ: Prentice Hall, 1983.

[Lin91] Y. Lin, B. Kim and P. Gray, "A 13-b 2.5-MHz self-calibrated pipelined A/D converter in 3-μm CMOS," *JSSC*, vol. 26, no. 4, April 1991, pp. 628-636.

[Ling81] H. Ling, "High-speed binary adder," *IBM J. Research and Development*, vol. 25, no. 3, May 1981, pp. 156-166.

[Liu01] X. Liu, C. Lee, C. Zhou, and J. Han, "Carbon nanotube field-effect inverters," *Appl. Phys. Letters*, vol. 79, no. 20, Nov. 2001, pp. 3329-3331

[Lohstroh79] J. Lohstroh, "Static and dynamic noise margins of logic circuits," *JSSC*, vol. SC-14, no. 3, June 1979, pp. 591-598.

[Lohstroh83] J. Lohstroh, E. Seevinck, and J. de Groot, "Worst-case static noise margin criteria for logic circuits and their mathematical equivalence," *JSSC*, vol. SC-18, no. 6, Dec. 1983, pp. 803-807.

[Lovett98] S. Lovett, M. Welten, A. Mathewson, and B. Mason, "Optimizing MOS transistor mismatch," *JSSC*, vol. 33, no. 1, Jan. 1998, pp. 147-150.

[Lu88b] S. Lu, "Implementation of iterative networks with CMOS differential logic," *JSSC*, vol. 23, no. 4, Aug. 1988, pp. 1013-1017.

[Lu91] S. Lu and M. Ercegovac, "Evaluation of two-summand adders implemented in ECDL CMOS differential logic," *JSSC*, vol. 26, no. 8, Aug. 1991, pp. 1152-1160.

[Lu93] F. Lu, H. Samueli, J.Yuan, C. Svensson, "A 700 MHz 24-b pipelined accumulator in 1.2-μm CMOS for application as a numerically controlled oscillator," *JSSC*, vol. 28, no. 8, Aug 1993, pp. 878-886.

[Lu93b] F. Lu and H. Samueli, "A 200-MHz CMOS pipelined multiplier-accumulator using a quasi-domino dynamic full-adder cell design," *JSSC*, vol. 28, no. 2, Feb. 1993, pp. 123-132.

[Lynch91] T. Lynch and E. Swartzlander, "The redundant cell adder," *Proc. IEEE Symp. Computer Arithmetic*, 1991, pp. 165-170.

[Lynch92] T. Lynch and E. Swartzlander, "A spanning tree carry lookahead adder," *IEEE Trans. Computers*, vol. 41, no. 8, Aug. 1992, pp. 931-939.

[Lyon87] R. Lyon and R. Schediwy, "CMOS static memory with a new four-transistor memory cell," *Proc. Advanced Research in VLSI*, March 1987, pp. 111-132

[Ma98] T. Ma, "Making silicon nitride film a viable gate dielectric," *IEEE Trans. Electron Devices*, vol. 45, no. 3, March 1998, pp. 680-690.

[MacSorley61] O. MacSorley, "High-speed arithmetic in binary computers," *Proc. IRE*, vol. 49, part 1, Jan. 1961, pp. 67-91.

[Mahalingam85] M. Mahalingam, "Thermal management in semiconductor device packages," *Proc. IEEE Custom Integrated Circuits Conf.*, 1985, pp. 46-49.

[Maier97] C. Maier et al., "A 533-MHz BiCMOS superscalar RISC microprocessor," *JSSC*, vol. 32, no. 11, Nov. 1997, pp. 1625-1634.

[Majerski67] S. Majerski, "On determination of optimal distributions of carry skips in adders," *IEEE Trans. Electronic Computers*, vol. EC-16, no. 1, 1967, pp. 45-58.

[Maluf00] N. Maluf, *An Introduction to Microelectromechanical Systems Engineering*, Norwood, MA: ArtechHouse, 2000.

[Maneatis96] J. Maneatis, "Low-jitter process-independent DLL and PLL based on self-biased techniques," *JSSC*, vol. 31, no. 11, Nov. 1996, pp. 1723-1732.

[Maneatis03] J. Maneatis, I. McClatchie, J. Maxey, and M. Shankaradas, "Self-biased high-bandwidth low-jitter 1-to-4096 multiplier clock generator PLL," *JSSC*, vol. 38, no. 11, Nov. 2003, pp. 1795-1803.

[Mathew03] S. Mathew, M. Anders, R. Krishnamurthy, and S. Borkar, "A 4-GHz 130-nm address generation unit with 32-bit sparse-tree adder core," *JSSC*, vol. 38, no. 5, May 2003, pp. 689-695.

[Matsui94] M. Matsui et al., "A 200 MHz 13 mm^2 2-D DCT macrocell using sense-amplifier pipeline flip-flop scheme," *JSSC*, vol. 29, no. 12, Dec. 1994, pp. 1482-1490.

[May79] T. May and M. Woods, "Alpha-particle-induced soft errors in dynamic memories," *IEEE Trans. Electron Devices*, vol. ED-26, no. 1, Jan. 1979, pp. 2-9.

[McMurchie00] L. McMurchie, S. Kio, G. Yee, T. Thorp, and C. Sechen, "Output prediction logic: a high-performance CMOS design technique," *Proc. Intl. Conf. Computer Design*, 2000, pp. 247-254.

[Mead80] C. Mead and L. Conway, *Introduction to VLSI Systems*, Reading, MA: Addison-Wesley, 1980.

[Mehta99] G. Mehta, D. Harris, and D. Singh, "Pulsed Domino Latches," US Patent 5,880,608, 1999.

[Meier99] N. Meier, T. Marieb, P. Flinn, R. Gleixner, and J. Bravman, "In-situ studies of electromigration voiding in passivated copper interconnects," *AIP Conf. Proc. 491*, Fifth Intl. Workshop on Stress-Induced Phenomena in Metallization, June 1999, p. 180.

[Meijs84] N. van der Meijs, J. Fokkema, "VLSI circuit reconstruction from mask topology," *Integration, The VLSI Journal, vol. 2*, no. 2, June 1984, pp. 85-119.

[Melly01] T. Melly, A. Porret, C. Enz, and E. Vittoz, "An analysis of flicker noise rejection in low-power and low-voltage CMOS mixers," *JSSC*, vol. 36, no. 1, Jan. 2001, pp. 102-109.

[Merchant01] S. Merchant, S. Kang, M. Sanganeria, B. van Schravendijk, and T. Mountsier, "Copper interconnects for semiconductor devices," *JOM: Journal of the Minerals, Metals, and Materials Society*, vol. 53, no. 6, June 2001, pp. 43-48.

[Messerschmitt90] D. Messerschmitt, "Synchronization in digital system design," *IEEE J. Selected Areas Communications*, vol. 8, no. 8, Oct. 1990, pp. 1404-1419.

[Mezhiba03] A. Mezhiba and E. Friedman, *Power Distribution Networks in High Speed Integrated Circuits*, Boston: Kluwer Academic Publishers, 2003.

[Miller00] R. Miller et al., "The development of 157nm small field and mid-field microsteppers," *Proc. SPIE*, vol. 4000, Optical Microlithography XIII, Christopher J. Progler ed., 2000, pp. 567-578.

[Miyatake01] H. Miyatake, M. Tanaka, and Y. Mori, "A design for high-speed low-power CMOS fully parallel content-addressable memory macros," *JSSC*, vol. 36, no. 6, June 2001, pp. 956-968.

[Mizuno94] T. Mizuno, J. Okumtura, and A. Toriumi, "Experimental study of threshold voltage fluctuation due to statistical variation of chanel dopant number in MOSFET's," *IEEE Trans. Electron Devices*, vol. 41, no. 11, Nov. 1994, pp. 2216-2221.

[Moazzami90] R. Moazzami and C. Hu, "Projecting gate oxide reliability and optimizing reliability screens," *IEEE Trans. Electron Devices*, vol. 37, no. 7, July 1990, pp. 1643-1650.

[Montanaro96] J. Montanaro et al., "A 160-MHz, 32-b, 0.5-W CMOS RISC microprocessor," *JSSC*, vol. 31, no. 11, Nov. 1996, pp. 1703-1714.

[Moore65] G. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, no. 8, April 1965.

[Moore03] G. Moore, "No exponential is forever: but 'forever' can be delayed!" *Proc. IEEE Intl. Solid-State Circuits Conf.*, 2003, pp. 1-19.

[Morgan59] C. Morgan and D. Jarvis, "Transistor logic using current switching routing techniques and its application to a fast carry-propagation adder, *Proc. IEE*, vol. 106B, 1959, pp. 467-468.

[Morrison61] P. Morrison and E. Morrison, eds., *Charles Babbage: On the Principles and Development of the Calculator*, New York: Dover, 1961.

[Mortezapour00] S. Mortezapour and E. Lee, "A 1-V, 8-bit successive approximation ADC in standard CMOS process," *JSSC*, vol. 35, no. 4, April 2000, pp. 642-646.

[Morton99] S. Morton, "On-chip inductance issues in multiconductor systems," *Proc. Design Automation Conf.*, 1999, pp. 921-926.

[Morton02] S. Morton, "Inductance: Implications and sobutions for high-speed digital circuits," *Proc. IEEE Intl. Solid-State Circuits Conf.*, vol. 2 , Feb 2002, pp. 554-557.

[Mou90] Z. Mou and F. Jutand, "A class of close-to-optimum adder trees allowing regular and compact layout," *Proc. IEEE Intl. Conf. Computer Design*, 1990, pp. 251-254.

[Mule02] A. Mule, E. Glytsis, T. Gaylord, and J. Meindl, "Electrical and optical clock distribution networks for gigascale microprocessors," *IEEE Trans. VLSI*, vol. 10, no. 5, Oct. 2002, pp. 582-594.

[Muller03] R. Muller, T. Kamins, and M. Chan, *Device Electronics for Integrated Circuits*, 3rd ed., New York: John Wiley & Sons, 2003.

[Murabayashi96] F. Murabayashi et al., "2.5 V CMOS circuit techniques for a 200 MHz superscalar RISC processor," *JSSC*, vol. 31, no. 7, July 1996, pp. 972-980.

[Murphy80] B. Murphy, "Unified field-effect transistor theory including velocity saturation," *JSSC*, vol. SC-15, no. 3, June 1980, pp. 325-327.

[Mutoh95] S. Mutoh et al., "1-V power supply high-speed digital circuit technology with multithreshold-voltage CMOS," *JSSC*, vol. 30, no. 8, Aug. 1995, pp. 847-854.

[Nabors92] K. Nabors, S. Kim, and J. White, "Fast capacitance extraction of general three-dimensional structures," *IEEE Trans. Microwave Theory and Techniques*, vol. 40, no. 7, July 1992, pp. 1496-1506.

[Nadig77] H. Nadig, "Signature analysis—concepts, examples and guidelines," *Hewlett Packard Journal*, vol. 28, no. 9, May 1977, pp. 15-21.

[Naffziger96] S. Naffziger, "A subnanosecond 0.5μm 64b adder design," *Proc. IEEE Intl. Solid-State Circuits Conf.*, 1996, pp. 362-363.

[Naffziger98] S. Naffziger, "High speed addition using Ling's equations and dynamic CMOS logic," US Patent 5,719,803, 1998.

[Naffziger02] S. Naffziger, G. Colon-Bonet, T. Fischer, R. Riedlinger, T. Sullivan, and T. Grutkowski, "The implemetnation of the Itanium 2 microprocessor," *JSSC*, vol. 37, no. 11, Nov. 2002, pp. 1448-1460.

[Nagel75] L. Nagel, *SPICE2: a computer program to simulate semiconductor circuits*, Memo ERL-M520, Dept. of Electrical Engineering and Computer Science, University of California at Berkeley, May 9, 1975.

[Nair78] R. Nair, S. Thatte, and J. Abraham, "Efficient algorithms for testing semiconductor random-access memories," *IEEE Trans. Computers*, vol. C-27, no. 6, June 1978, pp. 572-576.

[Nalamalpu02] A. Nalamalpu, S. Srinivasan, and W. Burleson, "Boosters for driving long onchip interconnects—design issues, interconnect synthesis, and comparison with repeaters," *IEEE Trans. Computer-Aided Design*, vol. 21, no. 1, Jan. 2002, pp. 50-62.

[Nambu98] H. Nambu et al., "A 1.8-ns access, 550-MHz, 4.5-Mb CMOS SRAM," *JSSC*, vol. 33, no. 11, Nov. 1998, pp. 1650-1658.

[Narayanan96] V. Narayanan, B. Chappell, and B. Fleischer, "Static timing analysis for self-resetting circuits," *Proc. Intl. Conf. Computer-Aided Design*, 1996, pp. 119-126.

[Narendra99] S. Narendra, D. Antoniadis, and V. De, "Impact of using adaptive body bias to compensate die-to-die Vt variation on within-die Vt variation," *Proc. Intl. Symp. Low Power Electronics and Design*, 1999, pp. 229-232.

[Narendra01] S. Narendra, S. Borkar, V. De, D. Antoniadis, and A. Chandrakasan, "Scaling of stack effect and its application for leakage reduction," *Proc. Intl. Symp. Low Power Electronics and Design*, 2001, pp. 195-200.

[Narendra03] S. Narendra, A. Keshavarzi, B. Bloechel, S. Borkar, and V. De, "Forward body bias for microprocessors in 130-nm technology generation and beyond," *JSSC*, vol. 38, no. 5, May 2003, pp. 696-701.

[Nauta95] B. Nauta and A. Venes, "A 70-MS/s 110-mW 8-b CMOS folding and interpolating A/D converter," *JSSC*, vol. 30, no. 12, Dec. 1995, pp. 1302-1308.

[Ng96] P. Ng, P. Balsara, and D. Steiss, "Performance of CMOS differential circuits," *JSSC*, vol. 31, no. 6, June 1996, pp. 841-846.

[Nikolić00] B. Nikolić, V. Oklobdija, V. Stojanović, W. Jia, J. Chiu, and M. Leung, "Improved sense-amplifier-based flip-flop: design and measurements," *JSSC*, vol. 35, no. 6, June 2000, pp. 876-884.

[Noice83] D. Noice, *A clocking discipline for two-phase digital integrated circuits*, Stanford University Technical Report, Jan. 1983.

[Nowak02] E. Nowak, "Maintaining the benefits of CMOS scaling when scaling bogs down," *IBM J. Research and Development*, vol. 46, no. 2/3, March/May 2002, pp. 169-180

[Nowka98] K. Nowka and T. Galambos, "Circuit design techniques for a gigahertz integer microprocessor," *Proc. Intl. Conf. Computer Design*, 1998, pp. 11-16.

[Ohkubo95] N. Ohkubo et al., "A 4.4 ns CMOS 54 x 54-b multiplier using pass-transistor multiplexer," *JSSC*, vol. 30, no. 3, March 1995, pp. 251-257.

[Oklobdija85] V. Oklobdija and E. Barnes, "Some optimal schemes for ALU implementation in VLSI technology," Proc. IEEE Symp. Comp. Arithmetic, 1985.

[Oklobdija86] V. Oklobdija and R. Montoye, "Design-performance trade-offs in CMOS-domino logic," *JSSC*, vol. SC-21, no. 2, April 1986, pp. 304-309.

[Ortiz-Conde02] A. Ortiz-Conde, F. Sánchez, J. Liou, A. Cerdeira, M. Estrada, and Y. Yue, "A review of recent MOSFET threshold voltage extraction methods," *Microelectronics Reliability*, vol. 42, 2002, pp. 583-596.

[Oshawa87] T. Oshawa et al., "A 60-ns 4-Mbit CMOS DRAM with built-in self-test function," *JSSC*, vol. SC-22, no. 5, Oct. 1987, pp. 663-668.

[Paik96] W. Paik, H. Ki, and S. Kim, "Push-pull pass-transistor logic family for low voltage and low power," *Proc. 22nd European Solid-State Circuits Conf.*, 1996, pp. 116-119.

[Parameswar96] A. Parameswar, H. Hara, and T. Sakurai, "A swing restored pass-transistor logic-based multiply and accumulate circuit for multimedia applications," *JSSC*, vol. 31, no. 6, June 1996, pp. 804-809.

[Paraskevopoulos87] D. Paraskevopoulos and C. Fey, "Studies in LSI technology economics III: design schedules for application-specific integrated circuits," *JSSC*, vol. SC-22, no. 2, April 1987, pp. 223-229.

[Parhami00] B. Parhami, *Computer Arithmetic Algorithms and Hardware Designs*, New York: Oxford University Press, 2000.

[Parihar01] S. Parihar et al., "A high density 0.10 μm CMOS technology using low K dielectric and copper interconnect," *Proc. Intl. Electron Devices Meeting*, 2001, pp. 11.4.1-11.4.4.

[Park99] J. Park, J. Lee, and W. Kim, "Current sensing differential logic: a CMOS logic for high reliability and flexibility," *JSSC*, vol. 34, no. 6, June 1999, pp. 904-908.

[Parker03] K. Parker, *The Boundary-Scan Handbook*, Boston: Kluwer Academmic Publishers, 2003.

[Partovi94] H. Partovi and D. Draper, "A regenerative push-pull differential logic family," *Proc. IEEE Intl. Solid-State Circuits Conf.*, 1994, pp. 294-295.

[Partovi96] H. Partovi et al., "Flow-through latch and edge-triggered flip-flop hybrid elements," *Proc. IEEE Intl. Solid-State Circuits Conf.*, 1996, pp. 138-139.

[Pasternak87] J. Pasternak, A. Shubat, and C. Salama, "CMOS differential pass-transistor logic design," *JSSC*, vol. SC-22, no. 2, April 1987, pp. 216-222.

[Pasternak91] J. Pasternak and C. Salama, "Design of submicrometer CMOS differential pass-transistor logic circuits," *JSSC*, vol. 26, no. 9, Sept. 1991, pp. 1249-1258.

[Patterson04] D. Patterson and J. Hennessy, *Computer Organization and Design*, 3rd ed., San Francisco, CA: Morgan Kaufmann, 2004.

[Paul02] B. Paul and K. Roy. Testing cross-talk induced delay faults in static CMOS circuit through dynamic timing analysis. *Proc. Intl. Test Conf.*, Oct. 2002, pp. 384-390.

[Pelgrom89] M. Pelgrom, A. Duinmaijer, and A. Welbers, "Matching properties of MOS transistors," *JSSC*, vol. 24, no. 5, Oct. 1989, pp. 1433-1440.

[Peng02] C. Peng et al., "A 90 nm generation copper dual damascene technology with ALD TaN barrier," *Tech. Digest Intl. Electron Devices Meeting*, Dec. 2002, pp. 603-606.

[Penney72] W. Penney and L. Lau, *MOS Integrated Circuits*, New York: Van Nostrand Reinhold, 1972.

[Petegem94] W. van Petegem, B. Geeraerts, W. Sansen, and B. Graindourze, "Electrothermal simulation and design of integrated circuits," *JSSC*, vol. 29, no. 2, Feb. 1994, pp. 143-146.

[Pfennings85] L. Pfennings, W. Mol, J. Bastiens, and J. van Dijk, "Differential split-level CMOS logic for subnanosecond speeds," *JSSC*, vol. SC-20, no. 5, Oct. 1985, pp. 1050-1055.

[Pihl98] J. Pihl, "Single-ended swing restoring pass transistor cells for logic synthesis and optimization," *Proc. IEEE Intl. Symp. Circuits and Systems*, vol. 2, 1998, pp. 41-44.

[Piña02] C. Piña, "Evolution of the MOSIS VLSI educational program," *Proc. Electronic Design, Test, and Applications Workshop*, 2002, pp. 187-191.

[Poulton03] K. Poulton et al., "A 20Gs/s 8 b ADC with a 1 MB Memory in 0.18 μm CMOS," *Proc. IEEE Solid-State Circuits Conf.*, Feb. 2003, pp. 318-319.

[Pretorius86] J. Pretorius, A. Shubat, and A. Salama, "Latched domino CMOS logic," *JSSC*, vol. SC-21, no. 4, Aug. 1986, pp. 514-522.

[Price95] D. Price, "Pentium FDIV flaw—lessons learned," *IEEE Micro*, vol. 15, no. 2, April 1995, pp. 86-88.

[Prince99] B. Prince, *High Performance Memories: New Architecture DRAMs and SRAMs—Evolution and Function*, New York: John Wiley & Sons, 1999.

[Proebsting91] R. Proebsting, "Speed enhancement technique for CMOS circuits," US Patent 4,985,643, 1991.

[Promitzer01] G. Promitzer, "12-bit low-power fully differential switched capacitor noncalibrating successive approximation ADC with 1MS/s," *JSSC*, vol. 36, no. 7, July 2001, pp. 1138-1143.

[Quader94] K. Quader, E. Minami, W. Huang, P. Ko, and C. Hu, "Hot-carrier-reliability design guidelines for CMOS logic circuits," *JSSC*, vol. 29, no. 3, March 1994, pp. 253-262.

[Rabaey03] J. Rabaey, A. Chandrakasan, and B. Nikolic, *Digital Integrated Circuits*, 2nd Ed., Upper Saddle River, NJ: Prentice Hall, 2003.

[Razavi98] B. Razavi, *RF Microelectronics*, Upper Saddle River, NJ: Prentice Hall, 1998.

[Reddy02] V. Reddy et al., "Impact of negative bias temperature instability on digital circuit reliability," *Proc. 40th IEEE Intl. Reliability Physics Symp.*, 2002, pp. 248-254.

[Restle98] P. Restle and A. Deutsch, "Designing the best clock distribution network," *Symp. VLSI Circuits Digest Tech. Papers*, 1998, pp. 2-5.

[Restle01] P. Restle et al., "A clock distribution network for microprocessors," *JSSC*, vol. 36, no. 5, May 2001, pp. 792-799.

[Riordan97] M. Riordan and L. Hoddeson, *Crystal Fire: The Invention of the Transistor and the Birth of the Information Age*, New York: W. W. Norton & Co, 1998.

[Rodgers89] B. Rodgers and C. Thurber, "A Monolithic ±5½-digit BiMOS A/D converter," *JSSC*, vol. 24, no. 3, June 1989, pp. 617-626.

[Rotella02] F. Rotella, V. Blaschke, and D. Howard, "A broad-band scalable lumped-element inductor model using analytic expressions to incorporate skin effect, substrate loss, and proximity effect," *Tech. Digest Intl. Electron Devices Meeting*, Dec. 2002, pp. 471-474.

[Ruehli73] A. Ruehli and P. Brennan, "Efficient capacitance calculations for three-dimensional multiconductor systems," *IEEE Trans. Microwave Theory and Techniques*, vol. MTT-21, No. 2, Feb. 1973, pp. 76-82.

[Rusu00] S. Rusu and G. Singer, "The first IA-64 microprocessor," *JSSC*, vol. 35, no. 11, Nov. 2000, pp. 1539-1544.

[Rusu03] S. Rusu, J. Stinson, S. Tam, J. Leung, H. Muljono, and B. Cherkauer, "A 1.5-GHz 130-nm Itanium 2 processor with 6-MB on-die L3 cache," *JSSC*, vol. 38, no. 11, Nov. 2003, pp. 1887-1895.

[Ryan01] P. Ryan et al., "A single chip PHY COFDM modem for IEEE 802.11a with integrated ADCs and DACs," *Proc. IEEE Intl. Solid-State Circuits Conf.*, 2001, pp. 338-339, 463.

[Rzepka98] S. Rzepka, K. Banerjee, E. Meusel, and C. Hu, "Characterization of self-heating in advanced VLSI interconnect lines based on thermal finite element simulation," *IEEE Trans. Components, Packaging, and Manufacturing Technology* - Part A, vol. 21, no. 3, Sept. 1998, pp. 406-411.

[Sah64] C. T. Sah, "Characteristics of the Metal-Oxide-Semiconductor Transistors," *IEEE Trans. Electron Devices*, ED-11, July 1964, pp. 324-345.

[Saint02] C. Saint and J. Saint, *IC Mask Design: Essential Layout Techniques*, New York: McGraw-Hill, 2002.

[Sakurai83] T. Sakurai, "Approximation of wiring delay in MOSFET LSI," *JSSC*, vol. SC-18, no.4, Aug. 1983, pp. 418-426.

[Sakurai86] T. Sakurai, K. Nogami, M. Kakumu, and T. Iizuka, "Hot-carrier generation in submicrometer VLSI environment," *JSSC*, vol. SC-21, no. 1, Feb. 1986, pp. 187-192.

[Sakurai90] T. Sakurai and R. Newton, "Alpha-Power Law MOSFET Model and its Applications to CMOS Inverter Delay and Other Formulas," *JSSC*, vol. 25, no. 2, April 1990, pp. 584-594.

[Samavati98] H. Samavati, A. Hajimiri, A. Shahani, G. Nazzerbakht, and T. Lee, "Fractal capacitors," *JSSC*, vol. 33, no. 12, Dec. 1998, pp. 2035-2041.

[Santoro89] M. Santoro, Design and Clocking of VLSI Multipliers, Ph.D. Thesis, Stanford University, CSL-TR-89-397, 1989.

[Sauerbrey02] J. Sauerbrey, T. Tille, D. Schmitt-Landsiedel and R. Thewes, "A 0.7-V MOSFET-only switched-opamp sigma delta modulator in standard digital CMOS technology," *JSSC*, vol. 37, no. 12, Dec. 2002, pp. 1662-1669.

[Sauerbrey03] J. Sauerbrey, D. Schmitt-Landsiedel and R. Thewes, "A 0.5V 1-μW successive approximation ADC," *JSSC*, vol. 38, no. 7, July 2003, pp. 1261-1265.

[Schellenberg03] F. Schellenberg, "A little light magic," *IEEE Spectrum*, vol. 40, no. 9, Sept. 2003, pp. 34-39.

[Schmitt38] O.H. Schmitt, "A thermionic trigger," *J. Scientific Instruments*, vol. 15, Jan. 1938, pp. 24-26.

[Scholtens02] P. Scholtens and M. Vertregt, "A 6-b 1.6-Gsample/s flash ADC in 0.18 μm CMOS using averaging termination," *JSSC*, vol. 37, no. 12, Dec. 2002, pp. 1599-1609.

[Schultz90] K. Schultz, R. Francis and K. Smith, "Ganged CMOS: trading standby power for speed," *JSSC*, vol. SC-25, no. 3, June 1990, pp. 870-873.

[Schultz95] K. Schultz and P. Gulak, "Architectures for large-capacity CAMs," *Integration*, vol. 18, no. 2-3, 1995, pp. 151-171.

[Schutten03] R. Schutten, T. Fitzbatrick, "Design for verification—blueprint for productivity and product quality," Synopsys white paper, 2003.

[Seeds67] R. Seeds, "Yield and cost analysis of bipolar LSI," *Intl. Electron Device Meeting*, Oct. 1967.

[Shahidi02] G. Shahidi, "SOI technology for the GHz era," *IBM J. Research and Development*, vol. 46, no. 2/3, March/May 2002, pp. 121-131.

[Sharma00] `http://www.free-ip.com/cordic/`

[She02] M. She et al., "JVD silicon nitride as tunnel dielectric in p-channel flash memory," *IEEE Electron Device Letters*, vol. 23, no. 2, Feb. 2002, pp. 91-93.

[Shepard99] K. Shepard, V. Narayanan, and R. Rose, "Harmony: static noise analysis of deep submicron digital integrated circuits," *IEEE Trans. Computer-Aided Design*, vol. 18, no. 8, Aug. 1999, pp. 1132-1150.

[Sheu87a] B. Sheu and P. Ko, "Measurement and modeling of short-channel MOS transistor gate capacitances," *JSSC*, vol. SC-22, no. 3, June 1987, pp. 464-472.

[Sheu87b] B. Sheu, D. Scharfetter, P. Ko, and M. Jeng, "BSIM: Berkeley short-channel IGFET model for MOS transistors," *JSSC*, vol. SC-22, no. 4, Aug. 1987, pp. 558-566.

[Shichman68] H. Shichman and D. Hodges, "Modeling and simulation of insulated-gate field-effect transistor switching circuits," *JSSC*, vol. SC-3, no. 3, Sept. 1968, pp. 285-289.

[Shockley52] W. Shockley, "A unipolar 'field-effect' transistor," *Proc. IRE*, vol. 40, 1952, pp. 1365-1376.

[Shoji82] M. Shoji, "Electrical design of BELLMAC-32a microprocessor," *Proc. IEEE Intl. Conf. Circuits and Computers, Sept. 1982*, pp. 112-115.

[Shoji85] M. Shoji, "FET scaling in domino CMOS gates," *JSSC*, vol. SC-20, no. 5, Oct. 1985, pp. 1067-1071.

[Shoji86] M. Shoji, "Elimination of process-dependent clock skew in CMOS VLSI," *JSSC*, vol. SC-21, no. 5, Oct. 1986, pp. 875-880.

[SIA97] Semiconductor Industry Association, *International Technology Roadmap for Semiconductors*, 1997.

[SIA02] Semiconductor Industry Association, *International Technology Roadmap for Semiconductors*, 2002 Update, public.itrs.net.

[Silberman98] J. Silberman et al., "A 1.0-GHz single-issue 64-bit PowerPC integer microprocessor," *JSSC*, vol. 33, no. 11, Nov. 1998, pp. 1600-1608.

[Simon92] T. Simon, "A fast static CMOS NOR gate," *Proc. Advanced Research in VLSI and Parallel Systems*, 1992, pp. 180-192.

[Sklansky60] J. Sklansky "Conditional-sum addition logic," *IRE Trans. Electronic Computers*, vol. EC-9, June 1960, pp. 226-231.

[Smith00] D. Smith and P. Franzon, *Verilog Styles for Synthesis of Digital Circuits*, Upper Saddle River, NJ: Prentice Hall, 2000.

[Soeleman01] H. Soeleman, K. Roy, and B. Paul, "Robust subthreshold logic for ultra-low power operation," *IEEE Trans. VLSI*, vol. 9, no. 1, Feb. 2001, pp. 90-99.

[Solomatnikov00] A. Solomatnikov, D. Somasekhar, K. Roy, and C. Koh, "Skewed CMOS: noise-immune high-performance low-power static circuit family," *Proc. IEEE Intl. Conf. Computer Design*, 2000, pp. 241-246.

[Somasekhar96] D. Somasekhar and K. Roy, "Differential current switch logic: a low power DCVS logic family," *JSSC*, vol. 31, no. 7, July 1996, pp. 981-991.

[Somasekhar98] D. Somasekhar and K. Roy, "LVDCSL: a high fan-in, high-performance, low-voltage differential current switch logic family," *IEEE Trans. VLSI*, vol. 6, no. 4, Dec. 1998, pp. 573-577.

[Somasekhar00] D. Somasekhar, S. Choi, K. Roy, Y. Ye, and V. De, "Dynamic noise analysis in precharge-evaluate circuits," *Proc. Design Automation Conf.*, 2000, pp. 243-246.

[Song96] M. Song, G. Kang, S. Kim, and B. Kang, "Design methodology for high speed and low power digital circuits with energy economized pass-transistor logic (EEPL)," *Proc. 22nd European Solid-State Circuits Conf.*, 1996, pp. 120-123.

[Song01] S. Song et al., "On the gate oxide scaling of high performance CMOS transistors," *Proc. Intl. Electron Devices Meeting*, 2001, pp. 3.2.1-3.2.4.

[Sowiati01] T. Sowlati, V. Vathulya, and D. Leenaerts, "High density capacitance structures in submicron CMOS for low power RF applications," *Proc. Intl. Symp. Low Power Electronics and Design*, Aug. 2001, pp. 243-246.

[Sparsø01] J. Sparsø and S. Furber, eds., *Principles of Asynchronous Circuit Design: A Systems Perspective*, Boston: Kluwer Academic Publishers, 2001.

[Srinivas92] H. Srinivas and K. Parhi, "A fast VLSI adder architecture," *JSSC*, vol. 27, no. 5, May 1992, pp. 761-767.

[Stinson03] J. Stinson and S. Rusu, "A 1.5 GHz third generation Itanium processor," *Proc. Design Automation Conf.*, 2003, pp. 706-709.

[Stojanovic99] V. Stojanovic and V. Oklobdija, "Comparative analysis of master-slave latches and flip-flops for high-performance and low-power systems," *JSSC*, vol. 34, no. 4, April 1999, pp. 536-548.

[Stroud02] C. Stroud, *A Designer's Guide to Built-in Self-Test*, Boston: Kluwer Academic Publishers, 2002.

[Sun87] J. Sun, Y. Taur, R. Dennard, and S. Klepner, "Submicrometer-channel CMOS for low-temperature operation," *IEEE Trans. Electron Devices*, vol. ED-34, no. 1, Jan. 1987, pp. 19-26.

[Sutherland99] I. Sutherland, B. Sproull, and D. Harris, *Logical Effort: Designing Fast CMOS Circuits*, San Francisco, CA: Morgan Kaufmann, 1999.

[Suzuki73] Y. Suzuki, K. Odagawa and T. Abe, "Clocked CMOS calculator circuitry," *JSSC*, vol. SC-8, no. 6, Dec. 1973, pp. 462-469.

[Suzuki93] M. Suzuki, N. Ohkubo, T. Shinbo, T. Yamanaka, A. Shimizu, K. Sasaki, and Y. Nakagome, "A 1.5-ns 32-b CMOS ALU in double pass-transistor logic," *JSSC*, vol. 28, no. 11, Nov. 1993, pp. 1145-1151.

[Sweeney02] P. Sweeney, *Error Control Coding: From Theory to Practice*, New York: John Wiley & Sons, 2002.

[Sylvester98] D. Sylvester and K. Keutzer, "Getting to the bottom of deep submicron," *Proc. IEEE/ACM Intl. Conf. Computer-Aided Design*, 1998, pp. 203-211.

[Tam00] S. Tam, S. Rusu, U. Desai, R. Kim, J. Zhang, and I. Young, "Clock generation and distribution for the first IA-64 microprocessor," *JSSC*, vol. 35, no. 11, Nov. 2000, pp. 1545-1552.

[Tam04] S. Tam, R. Limaye, and U. Desai, "Clock generation and distribution for the 130-nm Itanium 2 processor with 6-MB on-die L3 cache," *JSSC*, vol. 39, no. 4, Apr. 2004.

[Tharakan92] G. Tharakan and S. Kang, "A new design of a fast barrel switch network," *JSSC*, vol. 27, no. 2, Feb. 1992, pp. 217-221.

[Thomas02] D. Thomas and P. Moorby, *The Verilog Hardware Description Language*, 5th Ed. Boston: Kluwer Academic Publishers, 2002.

[Thorp99] T. Thorp, G. Yee, and C. Sechen, "Design and synthesis of monotonic circuits," *Proc. IEEE Intl. Conf. Computer Design*, 1999, pp. 569-572.

[Tiilikainen01] M. Tiilikainen, "A 14-bit 1.8-V 20-mW 1-mm^2 CMOS DAC," *JSSC*, vol. 36, no. 7, July 2001, pp. 1144-1147.

[Timko80] M. Timko and P. Holloway, "Circuit techniques for achieving high speed-high resolution A/D conversion," *JSSC*, vol. SC-15, no. 6, Dec. 1980, pp. 1040-1051.

[Timmermann94] D. Timmermann, B. Rix, H. Hahn and B. Hosticka, "A CMOS floating-point vector-arithmetic unit," *JSSC*, vol. 29, no. 5, Sept. 1994, pp. 634-639.

[Tobias95] P. Tobias and D. Trindade, *Applied Reliability*, 2nd ed., New York: Van Nostrand Reinhold, 1995.

[Troutman86] R. Troutman, *Latchup in CMOS Technology: The Problem and its Cure*, Boston: Kluwer Academic Publishers, 1986.

[Tschanz02] J. Tschanz et al., "Adaptive body bias for reducing impacts of die-to-die and within-die parameter variations on microprocessor frequency and leakage," *JSSC*, vol. 37, no. 11, Nov. 2002, pp. 1396-1402.

[Tsividis99] Y. Tsividis, *Operation and Modeling of the MOS Transistor*, 2nd ed., Boston: McGraw-Hill, 1999.

[Tyagi93] A. Tyagi, "A reduced-area scheme for carry-select adders," *JSSC*, vol. 42, no. 10, Oct. 1993, pp. 1163-1170.

[Uehara81] T. Uehara and W. vanCleemput, "Optimal layout of CMOS functional arrays," *IEEE Trans. Computers*, vol. C-30, no. 5, May 1981, pp. 305-312.

[Unger86] S. Unger and C. Tan, "Clocking schemes for high-speed digital systems," *IEEE Trans. Computers*, vol. 35, no. 10, Oct. 1986, pp. 880-895.

[Usami94] K. Usami and M. Horowitz, "Clustered voltage scaling technique for low-power design," *Proc. Intl. Symp. Low Power Electronics*, 1994, pp. 3-8.

[Uyttenhove03] K. Uyttenhove and M. Steyaert, "A 1.8-V 6-bit 1.3 GHz flash ADC in 0.25μm CMOS," *JSSC*, vol. 38, no. 7, July 2003, pp. 1115-1122.

[Vadasz66] L. Vadasz and A. Grove, "Temperature dependence of MOS transistor characteristics below saturation," *IEEE. Trans. Electron Devices*, vol. ED-13, no. 13, 1966, pp. 863-866.

[Vadasz69] L. Vadasz, A. Grove, T. Rowe, and G. Moore, "Silicon-gate technology," *IEEE Spectrum*, vol. 6, no. 10, Oct. 1969, pp. 28-35.

[van Berkel99] C. van Berkel and C. Molnar, "Beware the three-way arbiter," *JSSC*, vol. 34, no. 6, June 1999, pp. 840-848.

[Vangal02] S. Vangal et al., "5-GHz 32-bit integer execution core in 130-nm dual-V_T CMOS," *JSSC*, vol. 37, no. 11, Nov. 2002, pp. 1421-1432.

[vanZeijl02] P. van Zeijl et al., "A Bluetooth radio in 0.18μm CMOS," *JSSC*, vol. 37, no. 12, Dec. 2002, pp. 1679-1687.

[Veendrick80] H. Veendrick, "The behavior of flip-flops used as synchronizers and prediction of their failure rate," *JSSC*, vol. SC-15, no. 2, April 1980, pp. 169-176.

[Veendrick84] H. Veendrick, "Short-circuit dissipation of static CMOS circuitry and its impact on the design of buffer circuits," *JSSC*, vol. SC-19, no. 4, Aug. 1984, pp. 468-473.

[Vittal99] A. Vittal et al., "Crosstalk in VLSI interconnections," *IEEE Trans. Computer-Aided Design*, vol. 18, no. 12, Dec. 1999, pp. 1817-1824.

[Volder59] J. Volder, "The CORDIC trigonometric computing technique," *IRE Trans. Electronic Computers*, vol. EC-8, no. 3, Sept. 1959, pp-330-334.

[Vollertsen99] R. Vollertsen, "Burn-in," *IEEE Integrated Reliability Workshop Final Report*, 1999, pp. 167-173.

[Wadell01] B. Wadell, *Transmission Line Design Handbook*, Norwood, MA: Artech House, 1991.

[Wakerly00] J. Wakerly, *Digital Design Principles and Practices*, 3rd ed., Upper Sadde River, NJ: Prentice Hall, 2000.

[Wallace64] C. Wallace, "A suggestion for a fast multiplier," *IEEE Trans. Electronic Computers*, Feb. 1964, pp. 14-17.

[Wang86] L. Wang and E. McCluskey, "Complete feedback shift register design for built-in self test," *Proc. Design Automation Conf.*, Nov. 1986, pp. 56-59.

[Wang89] J. Wang, C. Wu, and M. Tsai, "CMOS nonthreshold logic (NTL) and cascode nonthreshold logic (CNTL) for high-speed applications," *JSSC*, vol. 24, no. 3, June 1989, pp. 779-786.

[Wang93] Z. Wang, G. Jullien, W. Miller, and J. Wang, "New concepts for the design of carry-lookahead adders," *Proc. IEEE Intl. Symp. Circuits and Systems*, 1993, vol. 3, pp. 1837-1840.

[Wang94] J. Wang, S. Fang, and W. Feng, "New efficient designs for XOR and XNOR functions on the transistor level," *JSSC*, vol. 29, no. 7, July 1994, pp. 780-786.

[Wang97] Z. Wang, G. Jullien, W. Miller, J. Wang, and S. Bizzan, "Fast adders using enhanced multiple-output domino logic," *JSSC*, vol. 32, no. 2, Feb. 1997, pp. 206-214.

[Wang00j] J. Wang and C. Huang, "High-speed and low-power CMOS priority encoders," *JSSC*, vol. 35, no. 10, Oct. 2000, pp. 1511-1514.

[Wang00l] L. Wang and N. Shanbhag, "An energy-efficient noise-tolerant dynamic circuit technique," *IEEE Trans. Circuits and Systems II*, vol. 47, no. 11, Nov. 2000, pp. 1300-1306.

[Wang01] J. Wang, C. Chang, and C. Yeh, "Analysis and design of high-speed and low-power CMOS PLAs," *JSSC*, vol. 36, no. 8, Aug. 2001, pp. 1250-1262.

[Wanlass63] F. Wanlass and C. Sah, "Nanowatt logic using field effect metal-oxide semiconductor triodes," *Proc. IEEE Intl. Solid-State Circuits Conf.*, 1963, pp. 32-33.

[Webb97] C. Webb et al., "A 400-MHz S/390 microprocessor," *JSSC*, vol. 32, no. 11, Nov. 1997, pp. 1665-1675.

[Wei98] L. Wei, Z. Chen, M. Johnson, K. Roy, and V. De, "Design and optimization of low voltage high performance dual threshold CMOS circuits," *Proc. Design Automation Conf.*, 1998, pp. 489-494.

[Weinberger58] A. Weinberger and J. Smith, "A logic for high-speed addition," System Design of Digital Computer at the National Bureau of Standards: Methods for High-Speed Addition and Multiplication, National Bureau of Standards, Circular 591, Section 1, Feb. 1958, pp. 3-12.

[Weinberger81] A. Weinberger, "4-2 carry-save adder module," *IBM Technical Disclosure Bulletin*, vol. 23, no. 8, Jan. 1981, pp. 3811-3814.

[Weiss02] D. Weiss, J. Wuu, and V. Chin, "The on-chip 3-MB subarray-based third-level cache on an Itanium microprocessor," *JSSC*, vol. 37, no. 11, Nov. 2002, pp. 1523-1529.

[Weste93] N. Weste and K. Eshraghian, *Principles of CMOS VLSI Design*, 2nd ed., Reading, MA: Addison-Wesley, 1993.

[Wicht01] B. Wicht, S. Paul, and D. Schmitt-Landsiedel, "Analysis and compensation of the bitline multiplexer in SRAM current sense amplifiers," *JSSC*, vol. 36, no. 11, Nov. 2001, pp. 1745-1755.

[Williams83] T. Williams and K. Parker, "Design for Testability—A Survey," *Proc. IEEE*, vol. 71, no. 1, Jan. 1983, pp. 98-112.

[Williams86] T. Williams, "Design for testability," *Proc. NATO Advanced Study Inst. Computer Design Aids for VLSI Circuits*, (P. Antognetti et al. ed.), NATO AIS Series, 1986, Martinus Nijhoff Publishers, pp. 359-416.

[Williams91] T. Williams and M. Horowitz, "A zero-overhead self-timed 160-ns 54-b CMOS divider," *JSSC*, vol. 26, no. 11, Nov. 1991, pp. 1651-1661.

[Wing82] O. Wing, "Automated gate matrix layout," *Proc. IEEE Intl. Symp. Circuits and Systems*, vol. 2, 1982, pp. 681-685.

[Wolf00] S. Wolf and R. Tauber, *Silicon Processing for the VLSI Era*, 2nd ed., Sunset Beach, CA: Lattice Press, 2000.

[Wong02] H. Wong, "Beyond the conventional transistor," *IBM Journal of Research and Development*, vol. 46, no. 2/3, March/May 2002, pp. 133-168.

[Wood01] J. Wood, T. Edwards, and S. Lipa, "Rotary traveling-wave oscillator arrays: a new clock technology," *JSSC*, vol. 36, no. 11, Nov. 2001, pp. 1654-1665.

[Wu87] C. Wu, J. Wang and M. Tsai, "The analysis and design of CMOS multidrain logic and stacked multidrain logic," *JSSC*, vol. SC-22, no. 1, Feb. 1987, pp. 47-56.

[Wu91] C. Wu and K. Cheng, "Latched CMOS differential logic (LCDL) for complex high-speed VLSI," *JSSC*, vol. 26, no. 9, Sept. 1991, pp. 1324-1328.

[Wurtz93] L. Wurtz, "An efficient scaling procedure for domino CMOS logic," *JSSC*, vol. 28, no. 9, Sept. 1993, pp. 979-982.

[Yamada95] H. Yamada, T. Hotta, T. Nishiyama, F. Murabayashi, T. Yamauchi, and H. Sawamoto, "A 13.3 ns double-precision floating-point ALU and multiplier," *Proc. Intl. Conf. Computer Design*, 1995, pp. 466-470.

[Yang98] S. Yang et al., "A high performance 180 nm generation logic technology," *Tech. Digest Intl. Electron Device Meeting*, Dec. 1998, pp. 197-200.

[Yano90] K. Yano, T. Yamanaka, T. Nishida, M. Saito, K. Shimohigashi, and A. Shimizu, "A 3.8-ns 16 x 16-b multiplier using complementary pass-transistor logic," *JSSC*, vol. 25, no. 2, April 1990, pp. 388-395.

[Yano96] K. Yano, Y. Sasaki, K. Rikino, and K. Seki, "Top-down pass-transistor logic design," *JSSC*, vol. 31, no. 6, June 1996, pp. 792-803.

[Ye98] Y. Ye, S. Borkar, and V. De, "A new technique for standby leakage reduction in high-performance circuits," *Symp. VLSI Circuits Digest Tech. Papers*, 1998, pp. 40-41.

[Ye00] Y. Ye, J. Tschanz, S. Narendra, S. Borkar, M. Stan, and V. De, "Comparative delay, noise and energy of high-performance domino adders with stack node preconditioning (SNP)," *Symp. VLSI Circuits Digest Tech. Papers*, 2000, pp. 188-191.

[Yee00] G. Yee and C. Sechen, "Clock-delayed dominio for dynamic circuit design," *IEEE Trans. VLSI*, vol. 8, no. 4, Aug. 2000, pp. 425-430.

[Yoon02] J. Yoon et al., "CMOS-compatible surface-micromachined suspended-spiral inductors for multi-GHz silicon RF Ics" *IEEE Electron Device Letters*, vol. 23, no. 10, Oct. 2002, pp. 591-593.

[Yoshimoto83] M. Yoshimoto et al., "A divided word-line structure in the static RAM and its application to a 64K full CMOS RAM," *JSSC*, vol. SC-18, no. 5, Oct. 1983, pp. 479-485.

[Young00] K. Young et al., "A 0.13 μm CMOS technology with 193 nm lithography and Cu/Low-k for high performance applications," *Proc. Intl. Electron Devices Meeting*, 2000, pp. 563-566.

[Yuan82] C. Yuan and T. Trick, "A simple formula for the estimation of the capacitance of two-dimensional interconnects in VLSI circuits," *IEEE Electron Device Letters*, vol. EDL-3, Dec. 1982, pp. 391-393.

[Yuan89] J. Yuan and C. Svensson, "High-speed CMOS circuit technique," *JSSC*, vol. 24, no. 1, Feb. 1989, pp. 62-70.

[Zhou99] X. Zhou, K. Lim, and D. Lim, "A simple and unambiguous definition of threshold voltage and its implications in deep-submicron MOS device modeling," *IEEE Trans. Electron Devices*, vol. 46, no. 4, April 1999, pp. 807-809.

[Zhou03] Y. Zhou and J. Yuan, "A 10-bit wide-band CMOS direct digital RF amplitude modulator," *JSSC*, vol. 38, no. 7, July 2003, pp. 1182-1188.

[Zhuang92] N. Zhuang and H. Wu, "A new design of the CMOS full adder," *JSSC*, vol. 27, no. 5, May 1992, pp. 840-844.

[Ziegler02] J. Ziegler, *Ion-Implantation—Science and Technology, 2002 Edition*, IIT Press, 2002.

[Ziegler96] J. Ziegler, "Terrestrial cosmic rays," *IBM J. Research and Development*, vol. 40, no. 1, Jan. 1996, pp. 19-39.

[Zimmermann96] R. Zimmermann, "Non-heuristic optimization and synthesis of parallel-prefix adders," *Proc. Intl. Workshop on Logic and Architecture Synthesis*, Dec. 1996, pp. 123-132.

[Zimmermann97a] R. Zimmermann and W. Fichtner, "Low-power logic styles: CMOS versus pass-transistor logic," *JSSC*, vol. 32, no. 7, July 1997, pp. 1079-1090.

[Zimmermann97b] R. Zimmermann, *Binary Adder Architectures for Cell-Based VLSI and their Synthesis*, ETH Dissertation 12480, Swiss Federal Institute of Technology, 1997.

[Zuras86] D. Zuras and W. McAllister, "Balanced delay trees and combinatorial division in VLSI," *JSSC*, vol. SC-21, no. 5, Oct. 1986, pp. 814-819.

Index

α 84, 191
α-power law model 84, 89
β 73, 176
ε 180
ε_0 72
ε_{ox} 72
γ 87
η 89
λ 87, 130, 196, 280
μ 72
Θ_{ja} 766
ρ 180, 198
τ 164, 307
ψ_0 81

-- 897
!== 875
$readmemh 875
% 852
& 851, 902
(3,2) counter 678, 703
(5,3) counter 704
* 275
+ 281
.alter 291
.dc 280, 305
.end 275
.global 282
.include 280
.lib 291
.measure 284
.op 286
.option accurate 286
.option autostop 286
.option post 275
.param 280
.plot 276
.print 276
.scale 280
.temp 286, 291
.tran 276
=== 875
>> 852
>>> 852
?: 852
@(*) 861
^ 851
_h 336
_l 336
_q 419
_s 419
_v 419
{} 856
| 851
~ 850
'event 906
1101 SRAM 3
12T SRAM 715
1T DRAM 734
2's complement multiplication 696
2's complementer 708
2-phase latches 385
4004 microprocessor 3
6T SRAM 715
7-segment display 863

abstraction 37, 480
AC:
  coupling 813
  ground 779
accelerated life testing 239
accelerometer 149
acceptor 114
access transistor 715
accumulation 67
ACM 289
active 151
active area 126
activity factor 191
AD 281, 299
adaptive body bias 194
adaptive deskew 806
ADC 819, 828
adder
  carry propagate 645
  carry select 657
  carry-increment 658
  carry-lookahead 655
  carry-ripple 645, 647
  carry-skip 651
  comparison 677
  conditional sum 659
  domino 668
  hybrid 667
  Ling 671
  Manchester 650
  multiple input 678
  Naffziger 671
  spanning-tree 667
  sparse-tree 667
  taxonomy 664
  tree 661
  Verilog 850
addition 638
address 713
advanced library format 562
aggressor 208
ALD 118
ALF 562
alignment mark 130
alpha particle 245
alpha-power law model 84, 89
ALU 389, 686
always block 857
amplifier
  common source 811
  differential 818

amplifier (continued)
operational 819
analog 55, 808
analog-to-digital converter 819, 828
AND 17
and 874, 897
AND plane 750
AND-OR-INVERT 13, 321
annealing 117
annihilation gate 444
anode 7
antenna rule 153
antialiasing filter 824
antifuse 149, 500
AOI 321
aperture 423, 453
application note 541
application specific integrated circuit 521
arbiter 463
architecture 36, 39
architecture body 896, 899
area 52
estimation 59
calculation method 289
arithmetic logic unit 686
arithmetic shifter 691
array 55, 713
array multiplier 694
array shifter 692
arrays of arrays 705
arrival time 170
arsenic 7
AS 80, 281
ASIC 521, 547
aspect ratio 196
assembler 40
assembly language 40
assertions 577
assign 850
assignment 867
asymmetric gate 324
asymptotic waveform evaluation 219
asynchronous 460, 465, 808
ripple-carry counter 683
reset 858
atomic layer deposition 118
ATPG 526, 593
automatic test pattern generation 526, 593
avalanche injection 742
average power 188
AVG 310
AWAVES 277
AWE 219

B 176
Babbage, Charles 638, 652
back annotation 535
back end 521
back-gate coupling 356, 358
balanced delay tree 705
ball grid array 762
band-to-band tunneling 194
bank 730
Bardeen, John 1
barrel shifter 691
base 3, 148
bathtub curve 239
Baugh-Wooley multiplier 696
behavioral
see Bitdiddle, Ben
description 849
domain 37
HDL 48
synthesis 522
Ben Bitdiddle
see Bitdiddle, Ben
Berkeley short-channel IGFET model 288
beta ratio 97
BGA 762
bias
forward 7
point 808
reverse 7
BiCMOS 148, 365
bidirectional pad 783
BILBO 605
bin 234, 289
binary counter 683
binary-reflected Gray code 688
bipolar transistor 3, 148
bird's beak 126
BIST 602, 605
memory 757
Bitdiddle, Ben
see Ben Bitdiddle
bitline 360, 714, 715, 725, 730, 735
bitwise operator 850, 896
black cell 648
Black's equation 240
block 896, 899
blocking assignment 867
body 8, 67, 370
bias 194
effect 87
Boltzmann's constant 81
Boolean unit 686
booster 223
Booth
encoding 698
selector 699
bootstrapping 172
boundary scan 609
branching effort 176
braniac 468
Brattain, Walter 1
breakdown 244
breakdown voltage 119, 153, 782
Brent-Kung 661
BSIM 288
bubble 9
pushing 321
buffer 909
bug tracking 584
built-in logic block observation 605
built-in potential 81
built-in self-test 602, 605
bulk 8
bumping 124, 764
bundle 649
buried contact 128
buried oxide 139
burn-in 235, 239
conditional keeper 340
burn-in board 627
bus 18
bypass capacitance 773
bypass capacitor 777

C 160
C^2MOS 403
C4 764, 769
cache 733
Caltech interchange format 61
Caltech intermediate format 558
CAM 747
capacitance 200
coupling 358

diffusion 76, 80, 281, 289
extraction 531
fringe 200
gate 75, 296
gate-drain 172
gate-source 171
input 165
intrinsic 77
load 165
overlap 78
parallel plate 200
parasitic 70, 80, 299
sidewall 81
symbiotic 773
transformation 178
wire 200
capacitor 143
bypass 773
DRAM 734
carbon nanotube 150
card 274
carry 638, 646
carry-bypass adder 651
carry-increment adder 658
carry-lookahead adder 655, 661
carry-propagate adder 645
carry-ripple adder 645, 647
carry-save adder 678, 694
carry-save redundant format 678
carry-select adder 657
carry-skip adder 651
cascaded reset domino 433, 438
cascode 331
cascode nonthreshold logic 360
cascode voltage switch logic 331
cascoded current mirror 815
case 861, 910
casez 864
cathode 7
CD domino 443, 447
C_{db} 76
C_{diff} 160
C-element 225
cell 32, 36
cell library 550
cell-based design 509
CFSR 603
C_g 71, 160
C_{gdel} 78
C_{gs} 75
C_{gsol} 78
channel 9, 67, 68
length 803
length modulation 86, 92, 809
stop diffusion 119
characteristic polynomial 603, 684
charge compensation 227
charge pump 791
charge sharing 340, 353, 358
chemical mechanical polishing 119
chemical vapor deposition 28, 117
chip tester 628
CIF 61, 558
C_{in} 165
circuit design 36, 49
circuit families 319, 549
comparison 367
CJ 81, 290
C_{jb} 299
C_{jbs} 80
C_{jbsnew} 80
C_{jbsw} 299
CJSW 290
CLA 655
CLB 501
clk 384
clock 20, 786
activity factor 191
chopper 407, 430, 798
distribution 415, 793
domain 788
feedback 790
gater 786, 798
gating 410, 417
global 786
logical 786
physical 786
reference 790
skew budget 800
stretcher 430, 798
tree router 532
clock skew 399, 786
budget 788
intentional 425
clock-blocking 441
clock-delayed domino 443
clocked CMOS 403
clocked deracer 416
clock-to-Q contamination delay 387
clock-to-Q propagation delay 387
clustered voltage scaling 366
CMOS 3
CMOS multidrain logic 330
CMOSTG 347
CMP 119
CNTL 360
collector 3, 148
column multiplexer 726
combinational logic 383, 849, 860, 909
comment 275, 851, 897
commercial temperature range 233
common mode input 818
common mode noise 816
common source amplifier 811
comparator 681
compiler 40
complementary CMOS 11, 320
complementary input 336
complementary metal oxide semiconductor 3
complementary pass transistor logic 348
complementary signal generator 446
complete feedback shift register 603
component declaration 901
compound gate 13, 321
compressor 704
concatenate 856, 902
concurrent signal assignment 896
conditional keeper 453
conditional signal assignment 898
conditional-sum adder 659
conduction complements 14
configurable logic block 501
constant 854, 911
constant current method 293
constant field scaling 246
constant voltage scaling 248
contact 128, 151
resistance 199
contact printing 115
contamination delay 159, 422
content-addressable memory 713, 747
continuous assignment 850
control card 275
controllability 592
controlled collapse chip connection 764
conv_integer 904
conv_std_logic_vector 918

coplanar waveguide 147
copper 142, 240
core-limited 54
corner 233
cosmic ray 245
COSMOSCOPE 277
cost
  fixed 541
  NRE 537
  prototype 538
  recurring 539
counter 683, 859, 908
coupling 207, 337
C_{out} 165
C_{ox} 72
CPA 645
$C_{permicron}$ 76
CPL 348
critical charge 245
critical layer 116
critical path 158
cross-section 23, 124
crosstalk 196, 207, 211
  control 227
crowbarred 11
crystal 7
CSA 678, 694, 703
C_{sb} 76
CSG 446
cubic crystal 7
current DAC 827
current density 240
current mirror 814
current source 815
custom design 511
cutoff region 68, 71, 88
CVD 117
CVSL 331, 348, 784
cyclic redundancy checking 602
Czochralski method 114

D 176
d 164
D flip-flop 21
  see also flip-flop
D latch 20
  see also latch
D1 333
D2 333
DAC 819, 824
damascene process 142
data sheet 541, 545
data skew 806
datapath 55
DC
  analysis 274
  bias 813
  source 275
  transfer characteristic 94, 305, 811
DCSL 364
DCVS 331, 336
DCVSL 331
DCVSPG 350
debugging 570
deck 274
decoder 181, 719, 741
decoupling capacitance. 773
decrementers 708
DEEP 130
DEF 559
delay 93, 159
  crosstalk 207
  effort 164
  Elmore 161
  intentional 787
  inverter 164
  fault 594
  matching 237, 442
  minimum 178
  multistage networks 174
  parasitic 164, 281
  path 176
  repeated 222
  slope dependence 169
  tracking 237
  Verilog 857
  VHDL 906
delayed clocking domino 433
delayed keeper 453
delayed precharge 440
delayed reset domino 433
delay-locked loop 789, 792
delta operator 646
DeMorgan's Law 10, 321
depletion 67
depletion mode transistor 375
deposition 117
design
  abstraction 480
  corner 233, 290, 303
  exchange format 559
  flows 520
  for manufacturability 608
  for testability 594
  margin 231
  methods 498
  reuse 257, 544
  rules 28, 125, 130, 133, 137
  rule checking 61, 151
  structured 481
  verification 60
device 8
  under test 575
DF 176
DFT 595
di/dt noise 353, 767, 772
DIBL 88, 194, 353
dielectric 139, 196
diff 126
differential amplifier 818
differential cascode voltage switch logic 331
differential cascode voltage switch with pass gate logic 350
differential circuit switch logic 364
differential circuits 359
differential flip-flop 412
differential keeper 339
differential nonlinearity 821
differential pair 235, 816
differential pass transistor logic 350
differential signaling 227
differential split-level 360
diffusion 25, 126
  area 80, 281
  capacitance 76, 80, 281, 289
  input 348, 352, 357
  isolated 76
  merged 76
  perimeter 281
  shared 76
  sidewall perimeter 80
digital signal processor 498
digital-to-analog converter 819, 824
diode 7, 89, 154
DIP 762
dirty power 781
distributed RC 205
divide-by-M counter 683
divided bitline 730

divided wordline 730
DLC 452
DLL 789, 792
domain 479, 788
domino
  adder issues 668
  clocking 427
  four-phase 431
  gate 334
  global STP 438
  N-phase 433
  scannable 601
  self-resetting 433
  -to-static interface 450
  two-phase 430
  unfooted 438
donor 114
dopants 7
doping level 81
dot diagram 694
double pass transistor logic 350
double rail logic 16
DPL 350
DPTL 350
DRACULA 151
drain 8, 68, 70
  lightly doped 121
drain saturation voltage 73
drain-induced barrier lowering 88
DRAM 734, 757
DRC 61, 151
drift 236, 788
drive 166
driver 159
dry etch 124
DSL 360
DSP 498
DSPF 561
D-to-Q delay 387
dual damascene 142
dual inline package 762
dual slope ADC 828
dual-inline package 62
dual-ported SRAM 728
dual-rail domino 336
dummy
  gate 443
  line 756
  resistor 144
DUT 575
dynamic
  circuits 332
  latch converter 452
  logic 376
  memory 714
  node 338
  noise 357
  noise margin 99
  power 190
  RAM 734
  sequencing 426
  storage 383

ebeam 585
ECC 687
ECDL 363
economics 535
ector change description 578
edge rate 159, 301
edge-triggered 384
  flip-flop 20
  latch 407
EEPL 350
EEPROM 714, 740, 741
effective channel length 92, 235
effective number of bits 822
effective oxide thickness 120
effective resistance 103, 159
effective series resistance 775
effort 164
  delay 164, 165
ELAT 450
Electric 39
electrical effort 164, 165
  path 175
electrical rule checker 61
electrically erasable programmable ROM 147, 714, 741
electromigration 240, 532
electron 67
electron beam 585
electrostatic discharge 244, 783
electrothermal simulation 354
Elmore delay 161, 168, 216
else 898
emitter 3, 148
enable 17, 410, 907
enable/disable CMOS differential logic 363
enabled register 858
energy 186, 309
  delay product 193
energy economized pass transistor logic 350
enhancement mode transistor 376
ENOB 822
entity declaration 896, 899
entry latch 450
epitaxy 117
EPROM 714, 741
equality comparator 681
erasable programmable ROM 714, 741
ERC 61
error 239
error-correcting code 687
E_{sat} 84
ESD 244, 783, 784
ESL 775
ESR 775
ETL 407
evaluation 332
evaporation 124
even parity 687
event 906
exposed state node 402
extraction 152

F 176
f 165
fab 61, 114
fabrication 23
failure 586
failures in time 239
fall time 159
false path 526
fan 766
fanout 164, 165
fanout-of-4 inverter 174
fast 233
fault 567
fault coverage 593
fault model 589
FD SOI 370
feature size 30, 126
feedback clock 790
FET 8
FF 290
FIB 585
Fibonacci number 40
field device 119

field oxide 24, 119
field solver 312
field-programmable gate array 500
FIFO 746
fill 204
filter 779
  antialias 824
  reconstruction 823
finfet 139
finger 144
finite state machine 868, 913
first in first out queue 746
first-order model 71
FIT 239
flash ADC 832
flash memory 147, 714, 740, 741
flatten 48
flight time 205
flip-chip 762, 764
flip-flop 384, 385, 415, 473, 857, 906
  circuit design 405
  differential 412
  edge-triggered 20
  enabled 410
  K6 413
  resettable 408
  scannable 597
  synchronizer 458
  TSPC 414
floating 11, 208
  body 370
  gate 147, 742
floorplan 52, 530
fluorosilicate glass 142
FO4 174
  inverter delay 235, 249, 281, 303
focused ion beam 585
folded bitline 735
folding 556
foot 333
forbidden zone 98
fork 185
formal verification 61, 525
forward bias 7
forward body bias 194
foundry 509, 539
four-phase domino 431
Fowler-Nordheim tunneling 742
FPGA 500
fractal capacitor 144
freeze spray 92
fringe capacitance 200
fringe capacitor 144
front end 521
FSM 913
full adder 638
  CMOS 639
  CPL 643
  delay 645
  dynamic 644
  transmission gate 642
  Zhuang 643
full keeper 428
full-scale range 820
fully depleted SOI 370
fully restored 16
functional block 36
fundamental carry operator 646
funnel shifter 691
fuse 147, 149
fused multiply-add 705

G 175
g 165, 173
gain 811
Gajski-Kuhn Y chart 480
GAMMA 287
ganged CMOS 330
garbage 273
gate 8, 67
  AND 17
  array 507
  asymmetric 324
  capacitance 75, 296
  compound 13, 321
  delay 178
  dielectric 139
  extension 127
  leakage 90
  matrix layout 552
  NAND 10
  NOR 12
  NOT 10
  overdrive 817
  oxide 23, 67, 119, 138, 775
  pseudo-nMOS 327
  shrink 247
  size 178
  skewed 325
  stack 120
gater 796, 798
gate-source capacitance 171
Gaussian variation 231
GDS 61, 558
generalized Muller C-element 434
generate 638, 645, 916
  group 646
  pseudo 671
generic 916
GIGO 273
Gilbert cell 840
glitch 387
global clock 786
global STP domino 438
global wire 249
globally-reset domino 438
g_{ds} 809
g_m 809
GNATS 584
GND 9
golden model 568, 582
gray cell 648
Gray code 462, 688
grid
  clock distribution 793
  power 212, 768
ground plane 220
ground rule 125
Grove-Frohman model 288
guard ring 244, 356, 780, 792

H 175
h 165, 173, 196
half adder 638
half-cycle 386
Hamming distance 687
Han-Carlson 664
handler 579
handshake 460
hard edge 396
hard error 239
hardware description language 46, 48, 849
harness 579
HDL 46, 849
heat 761, 765
  gun 92
  sink 764, 766
  spreader 764
height 196

HF 25
hierarchy 36, 47, 485, 498, 854, 901
high-impedance 11
high-k 139
high-level language 40
high-speed domino 453
HI-skew 97, 325, 334
history effect 372
hold time 387, 393, 423, 805
hole 7, 67
hot carriers 241
hot electrons 241, 327
hot spot 354
HSPICE 274
  see also SPICE
H-tree clock distribution 793, 794
human body model 783
hybrid adder 667, 673
hybrid clock distribution 793, 796
hydrofluoric acid 25
hysteresis 782

I/O 780
I/O pad 53, 781
IDDQ test 608
I_{di} 69
I_{dsat} 73
IEEE 1149 609
IEEE.STD_LOGIC_1164 904
IEEE.STD_LOGIC_UNSIGNED 904
if 910
if/else 861
IMD 822
immediate 40
implantation 117
in 896
incrementer 684, 708
independent source 277
indeterminate region 98
inductance 210, 772
inductive crosstalk 211
inductor 146
industrial temperature range 233
infant mortality 239
initial 875
inner input 324
input 850
  arrival time 170
  capacitance 165
  order 324
  pad 782
  slope 169, 301
  threshold 97
input/output 55, 780
instantaneous power 186
INTEGRAL 309
integral nonlinearity 821
integrated circuit 2
intellectual property 257, 486
intentional clock skew 425
intentional delay 787
intentional skew 787
intentional time borrowing 398
interconnect 196, 311
  scaling 249
inter-die variation 233
interdigitated bus 221, 337
intermodulation distortion 822
International Technology Roadmap for Semiconductors 251
intrinsic capacitance 77
intrinsic carrier concentration 81
intrinsic delay 166
intrinsic silicon 114
inversion 68
inverter
  as amplifier 812
  cross-section 23
  DC characteristics 94
  FO4 174
  FO4 delay 235, 281, 303
  layout 32
  mask set 26
  pseudo-nMOS 100, 327
  schematic 10
  skewed 325
  symbol 10
ion implantation 25
IP block 257, 486
IR drop 767, 771
IR drops 353
isolated diffusion 76
isolation 119
ITRS 251, 473

jamb latch 403
jitter 236, 788, 803
JTAG 609
jumper 219
junction 114
junction leakage 89

k 81, 209
K6 flip-flop 413
keeper 235, 338, 353
  delayed 453
  full 428
Kilby, Jack 1
kill 638
Klass semidynamic flip-flop 411
Knowles 664
Kogge-Stone 661
KP 287

L 72, 92, 210
L di/dt noise 767, 772
ladder 161
Ladner-Fischer 664
LAMBDA 287
lambda 28
large-scale integration 4
laser voltage probing 585
last in first out queue 747
latch 20, 384, 859, 908
  circuit design 402
  enabled 410
  incorporating logic 410
  jamb 403
  resettable 408
  scannable 600
  TSPC 414
latched CMOS differential logic 364
latched domino 445
latchup 242, 372, 781
lateral diffusion 25
lateral scaling 247
lattice 7
layout 23, 550, 551
  design rules 28, 125
  gate matrix 552
  generation 521
  symbolic 511
  versus schematic 61, 534
LCDL 364
LDD 121
L_{drawn} 92
lead frame 764
leaf 37
leakage 88, 93, 188, 235, 338, 352, 357

leaker 340
lean integration with pass transistors 349
LEAP 349
LEF 558
L_{eff} 235
length 31, 196, 233
Level 1 model 287
Level 2/3 models 288
level converter 784
level sensitive scan design 599
levelize 443
level-sensitive latch 20
LFSR 603, 684
library 48, 509, 904
  exchange format 558
  mapping 524
  use clause 896
LIFO 747
light, speed of 211
lightly doped drain 121
line of diffusion 32, 551
linear extrapolation method 294
linear feedback shift register 603
linear region 70, 71, 73
linear-feedback shift register 684
linearity 820
Ling adder 671
liquid cooling 766
liquid nitrogen 216
literal 750
lithography 25
LMAX 289
LMIN 289
L-model 205
load 100, 159
load board 575
load capacitance 165
local interconnect 123
local wire 249
locality 495, 498
LOCOS 119
logarithmic adder 661
logic design 36, 46, 549
logic simulator 48
logic synthesis 48
logic verification 568, 579
logical clock 786
logical effort 164, 165, 319
  catalog 167
  definition 166
  dynamic gates 341
  estimation 166
  extraction from datasheets 185
  input order 324
  measurement 166
  memories 731
  method 173
  path 175
  pseudo-nMOS 327
  simulation 306
  summary 183
  wire 227
logical shifter 691
lookahead adder 661
loop 789
LO-skew 97, 325
lossy multiconductor transmission line 312
low-k dielectric 142, 196
low-power design 191
  logic 366
  sequential circuits 417
low-swing signaling 229
LSI 4
LSI Logic 507
LSSD 599
LVDCSL 364
LVP 585
LVS 61, 534
Lyon-Schediwy decoder 723

machine language 40
magnetic field 210
magnitude comparator 681
majority gate 639
majority-carrier 67
Manchester carry chain 343, 650, 655
manpower
  see personpower
manufacturing test 573, 588
margin 231, 726
  delay match 443
mask 25
mask-programmed ROM 740
master 20
master-slave flip-flop 21
matched delay 237
matching 93, 235, 747
max-delay 388, 526
maximal-length shift register 684
maximum-g_m method 294
max-time 159
MCF 207
MCM 763
Mealy machine 868
mean time between failures 239
mean time to failure 240
meander resistor 144
medium-scale integration 4
memory 713, 866, 912
  content-addressable 747
  DRAM 734
  ROM 739
  self-test 606
  SRAM 715
memory element 383
MEMS 149
merged
  contact 128
  diffusion 76
mesochronous 465
metal 128, 151
  gate 121
  -insulator-metal capacitor 144
  layer selection 219
  layer stack 196
  oxide semiconductor 3
metallization 124
metastability 454
metrology 125
microarchitecture 36, 42, 548
microelectromechanical systems 149
micron design rules 135
microstrip 147
military temperature range 233
miller coupling factor 207
Miller effect 173, 207, 282
MIM capacitor 144
min-delay 393, 526
minority carrier injection 355, 358
minterm 750
min-time 159
MIPS 39
mirrored cell rows 768
mismatch 235, 310
mixed-signal
  design 808
  test 625
mixer 840

MJ 81, 290
MJSW 290
mobility 72, 90, 104, 140
mobility degradation 84
mod 897
model
  α-power law 84
  ideal 107
  nonideal 83
  RC 103
MODL 343
modularity 492, 498
module 850
  parameterized 874
modulo 852
moment matching 219
monotonic static CMOS 339
monotonicity 334, 669
MONTE 311
Monte Carlo simulation 310
Moore machine 868
Moore, Gordon 4
Moore's Law 4
MOS 8
MOS capacitor 143
MOSFET 3, 8
  see also transistor
  switch model 9
MOSIS 30, 130
  design rules 132
  I/O pad 784
Motoroil 68W86 181
MSI 4
MTBF 239, 459
MTCMOS 195
MTTF 240
Muller C-element 225
  generalized 434
multi-chip module 763
multidrain logic 330
multilayer capacitance 201
multilevel-lookahead adder 661
multiple threshold CMOS 195
multiple threshold voltages 138, 327
multiple-output domino logic 343
multiplexer 18, 345
  column 726
multiplication 693
multiplier
  array, signed 697
  array, unsigned 694
  Booth 699
  fused 705
  higher radix 701
  hybrid 705
  serial 705
  tree 704
multiply-add 705
multi-ported SRAM 728
multistage delay 174
mutual inductance 779
mux-latch 411

n 88
n+ 8
N_A 81
Naffziger adder 671
Naffziger pulsed latch 407
NAND 10, 169, 321
nand 874, 897
NAND ROM 743
nanotubes 150
NBTI 242
NCO 487
N_D 81
n-diffusion 28
negative bias temperature instability 242
negative edge triggered flip-flop 21
negative-level-sensitive latch 20
netlist 50
n_i 81
nichrome 144
N_{MH} 98
N_{ML} 98
nMOS 3, 8
nMOS logic 375
no race flip-flop 406
noise 767
  analysis 532
  budget 358
  common mode 816
  crosstalk 208
  diffusion input 357
  feedthrough 99, 358
  immunity 98
  margin 98, 305, 357
  power supply 353, 779
  substrate 780
noise-tolerant precharge 339, 447
nominal 233
nonblocking assignment 857, 867
noninverting functions 17
nonmonotonic 724
nonmonotonic dynamic logic 442
nonoverlapping clock 386, 418
non-recurring engineering 519, 537
nonrestoring 17, 19
nonsaturated region 70
nonthreshold logic 360
nonvolatile 714
non-volatile memory 147
NOR 12, 169, 321
  pseudo-nMOS 328
  ROM 739
nor 874, 897
NORA 343, 406
normal skew 97
normal variation 231
NOT 10
not 874, 896
NP domino 343
N-phase domino 433
npn transistor 148, 365
NRE 519, 537
NRZ 579
n-select 126, 151
NTL 360
NTP 339
ntra-die variation 233
n-type 7, 114
numerically controlled oscillator 487
NVM 147
n-well 25
nwell 151

object code 40
observability 592
odd/even array 705
OFF 9
off-axis illumination 116
offset 821
ohmic contact 24
ON 9
one detector 679
one-shot 407
one-time programmable memory 147, 741
op-amp 819
opaque 20, 384, 859

OPC 116
open bitline 735
open circuit 589
OpenAccess 563
operating life, 239
operating temperature 231
operational amplifier 819
OPL 446
opportunistic time borrowing 399
  domino 431
optical clock distribution 807
optical proximity correction 116
or 874, 897
OR plane 750
OR-AND-INVERT 15
order of operations 853, 900
OTB Domino 431
others 911
out 896
outer input 324
output 850
  conductance 810
  pad 781
  prediction logic 446
  resistance 810
overdrive 817
overglass 124, 129
overlap 126
overlap capacitance 78
overturned-staircase tree 705
overvoltage 244
oxidation 25, 118
oxide 23
  thickness 244
oxynitride 120

P 176
p 165, 173
P(t) 186
p+ 8
P/N ratio 284, 326
package 761, 765
pad 53, 781
  analog 784
  bidirectional 783
  frame 53, 764
  input 782
  MOSIS 784
  output 781
  oxide 119
  supply 781
pad-limited 54
parallel 11
parallel plate capacitance 200
parallel-prefix adder 661
parameter 864, 874
parameterized block 915
parameterized module 874
parasitic
  capacitance 70, 80, 299
  estimator 311
  extract 531
  package 765
parasitic delay 164, 165, 281
  catalog 168
  definition 167
  dynamic gates 341
  input order 324
  simulation 306
parity 687
  check matrix 687
partial product 693
partially depleted SOI 370
partially redundant multiples 702
Partovi pulsed latch 407
pass gate 16
  leakage 372
pass transistor 15, 101
passivation 124, 129
pass-transistor 345
path
  branching effort 176
  delay 176
  effort 176
  effort delay 176
  electrical effort 175
  logical effort 175
  parasitic delay 176
PB 290
PBSW 290
PD 281, 299
PD SOI 370
PDEF 563
p-diffusion 28
peek 572
periodic 465
permeability 147, 211
permittivity 72
perpetrator 208
personpower 542
PGA 762
phase detector 791
phase shift masks 116
phase-error accumulation 791
phase-frequency detector 791
phase-locked loop 779, 789
PHI 287
photolithography 25, 115
photomask 115
photoresist 25, 115
PHP 290
physical α-power law model 89
physical clock 786
physical design 36, 52
physical design exchange format 563
physical domain 37
physical synthesis 521, 528
PICA 585
picoprobe 585
picosecond imaging circuit analysis 585
piecewise linear source 275
pin grid array 762
pinch-off 70
P_{inv} 168
pipeline ADC 834
piranha etch 25
pirhana solution 124
PISO 746
pitch 33, 131, 196
  matching 55, 720
pitfalls 350
PLA 499, 750
place and route 49, 528
placement 529
planarize 119
plane 221
plastic leadless chip carrier 762
plastic transistor 141
platform 518
PLCC 762
plesiochronous 465
PLL 779, 789
plug 124
π-model 205, 312
pMOS 3, 8
pnp transistor 148
poke 572
poly 121, 151

polycide 123
polycrystalline silicon 8
poly-insulator-poly capacitor 144
polysilicon 8, 28, 121, 126
  bias 130
  jumper 219
port 714, 899
posedge 857
positive-edge triggered flip-flop 21
positive-level-sensitive latch 20
postcharge domino 433
power 186
  analysis 526, 532
  distribution 767
  dynamic 190, 297
  estimation 528
  grid 212
  metrics 193
  scaling 254
  simulation 309
  static 188
  supply filter 779
  supply noise 353, 358
power-delay product 193
PPL 350
PRBS 685
precedence 853, 900
precharge 332, 716
  race 442
predecode 720
predicated self-reset 434
predischarge 340
prefix computation 646, 706
prefix operator 646
prescaler 840
price, selling 536
printing 115
priority encoder 706, 865, 910
probe card 575
probe point 585
process
  comparison 301
  corner 233, 290, 303, 357, 358
  statement 906
  tilt 233, 236
  variation 231, 233, 550
product 693, 750
productivity 256
programmable logic 499
programmable logic array 499, 750
programmable ROM 714, 741
projection printing 115
PROM 714, 741
propagate 638, 645
  pseudo 672
propagated noise 99
propagation delay 159, 422, 456
proximity printing 115
PRSG 602
PS 80, 281
p-select 126, 151
pseudo-complement 669
pseudo-generate 671
pseudo-nMOS 100, 190, 327
pseudo-propagate 672
pseudo-random bit sequence 685
pseudo-random sequence generator 602
PSM 116
p-type 7, 114
pull-down 328
pull-down network 11
pull-up 328
pull-up network 11
pulse source 275
pulse width modulated DAC 825
pulsed domino flip-flop 450
pulsed latch 385, 416, 473
  circuit design 407
  Naffziger 407
  Partovi 407
punchthrough 244
push-pull CPL 671
push-pull pass transistor logic 350
PWL 275

Q 146
q 81
$Q_{channel}$ 71
Q_{crit} 245
QFP 762
qualification 627
qualified clock 419
queue 746

R 159
R-2R DAC 826
race condition 393, 445
radio 482, 837
radio-frequency circuit 808
radix 665, 698
rails 14, 32
RAM 714
random 685, 788
random access memory 713
random access scan 596
random logic 55
random mismatch 236
rapid prototyping 544
ratio failure 352
ratioed circuit 100, 327
RC delay 205
  ladder 161
  model 103, 159
  tree 218
reachable radius 253
read stability 718
read/write memory 714
read-only memory 714, 739
reconstruction filter 823
recurring cost 539
reduction operator 851
reference clock 790
refractory metal 122
register 21, 857
register file 181, 729
regression test 582
regularity 488, 498
reliability 239, 626, 767
rem 897
repeater 221, 771
  farm 253
  staggered 227
replica bias generator 828
replica biasing 330
replica cell 726
reset 907
  asynchronous 408
  synchronous 408
resistance 159, 198
  contact 199
  effective 299
  sheet 199
resistive region 70
resistivity 198
resistor 144
resistor string DAC 825
resolution 820
  enhancement 116, 155
restoring 16

reticle 115
retrograde well 117
return path 778
return to zero 579
reuse 257
reverse bias 7, 89
  body 194
RF circuit 808, 837
ring oscillator 175
ripple-carry counter 683
rise time 159
RISING_EDGE 906
r_o 810
ROM 714, 739
root 37
rotary traveling-wave oscillator 808
rotator 691
routing 530
routing channel 55
routing track 33
RTL 521
rtranif 874
RTZ 579
rubylith 511
run set 150
runner 83, 207

S 246
s 196
SA-F/F 412
sales 1
salicide 123
sample set differential logic 362
sapphire 139
saturation
  field 84
  region 70, 71, 73, 86
  velocity 84
scalable CMOS design rules 130
scaled wire 249
scaling 5, 245
scan 415, 596
scan chain 596
scanning electron microscopy 125
schedule 541
Schmitt trigger 782
Schottky diode 24
SCMOS 130
SCR 242
scribe line 130
SDF 560
SDFF 411
sea-of-gates 507
SECDED 687
secondary precharge transistor 340
segmented DAC 828
select 899
selected signal assignment 898
self-aligned gate 121
self-aligned process 28
self-biased loop 792
self-bypass 389, 396
self-heating 241, 354, 373
self-resetting domino 433, 441, 724
self-test 757
Semiconductor Industry Association 251
semidynamic flip-flop 411
sense amplifier 235, 360, 725
sense-amplifier flip-flop 412
sensitivity list 857, 906
separation 126
sequencing 385
  element 383
  overhead 383, 548
sequential logic 383, 849, 860, 909
SER 245
serial access memory 713
serial multiplication 705
serial/parallel memory 746
series 11
set 409
setup time 387, 388, 422, 805
SFDR 822
SFPL 331
shallow trench isolation 119
shared diffusion 76
sheet resistance 198
Shichman-Hodges Model 287
shielding 221
shift register 745, 867
shifter 691
SHL 902
shmoo 575, 586
Shockley model 71, 287
Shockley, William 2
short circuit 191, 589
SHR 902
Si 7
SIA 251
SiGe 140, 148
sigma-delta ADC 835
signal 900
signal return path 778
signal return ratio 215
signaling, low-swing 229
signal-to-noise ratio 822
signal-to-return ratio 779
signature analysis 602
signed 681
signed multiplier 696
sign-extend 856
SiH_4 28
silane 28
silicide 122, 200
silicide block 144
silicon 7
silicon debug 584
silicon dioxide 23
silicon foundry 539
silicon on insulator 138
silicon-controlled rectifier 242
silicon-on-insulator 369
SiLK 143
simulator 273
SiO2 23, 118, 196
SIPO 746
size 178
skew 97, 325, 399, 786
  budget 800
  data 806
  definition 787
  intentional 787
  pitfalls 842
  tolerance 400
skew-tolerant domino 429
skew-tolerant high-speed domino 453
skin effect 211
Sklansky 661
slave 20
slave scan latch 599
sleep mode 192
slice plan 59
slope 159, 169, 185, 301
slow 233
small-scale integration 4
small-signal model 808
smoke test 571
SMT 762
snap-together cell 55

SNR 822
SOC 35, 257
SOC test 622
soft error 245, 358, 415
software radio 482
SOI 138, 196, 244, 369
SOIC 762
solder ball 764
solder bump 769
source 8, 68, 70
source code 40
source follower pull-up logic 331
source-synchronous clocking 808
space 219
spacer 122
spacing 196
spanning-tree adder 667
sparse-tree adder 667
speed demon 468
speed of light 211
SPEF 561
SPICE 274
  BSIM 288
  continuation card 281
  control card 275
  DC analysis 280
  DC source 275
  deck 274
  device characterization 292
  elements 277
  functions 284
  levels 287
  measurement 284
  models 287
  multiplier M 284
  optimization 284
  piecewise linear source 275
  printing and plotting 276
  pulse source 275
  subcircuit 282
  transient analysis 276
  units 277
spine clock distribution 793, 794
spiral inductor 146
split contact 128
spurious free dynamic range 822
sputtering 124
square 198
SR ratio 215, 779
SRAM 715
  dual-ported 728
  layout 716
  read 718
  write 719
SRCMOS 436
SRPL 350
SS 290
SSDL 362
SSI 4
stable 419
stack 747
stack effect 196
stacked via 129
stage effort 165, 181, 341
  best 180
stages, best number 178
staggered repeaters 227
standard cell 48, 55
  library 509
standard delay format 560
standard parasitic format 531, 561
state 849
static CMOS 11, 320
static evaluation transistor 436
static load 100
static memory 714
static noise margin 99
static power 188
static RAM 715
static storage 383
static timing analysis 526
staticize 338
static-to-domino interface 449
STD_LOGIC 903
STD_LOGIC_UNSIGNED 904
STD_LOGIC_VECTOR 903
stepper 115
STI 119
stick diagram 33
strained silicon 141
stream 558
strength 14
structural description 849
structural domain 37
structural HDL 48
structural primitive 874
structured design 36, 481
stuck-at 589
subarray 730, 735
subcircuit 282
SUBM 130
substrate 8, 23
  contact 24
  current 241
  noise 780
subthreshold 71
  conduction 88, 188
  leakage 138, 139, 303, 352
  slope 89, 371
subtracter 677
successive approximation ADC 831
sum-addressed decoder 724
sum-of-products 750
SUP 291
SUPPLY 280, 291
supply current 768
supply impedance 776
supply rails 32
supply voltage 231, 232
surface mount 762
surface state charge 117
SWEEP 279, 306
swing-restored pass transistor logic 350
swizzle 855, 901
symbiotic capacitance 773
symbolic layout 511
symmetric function 640
symmetric gate 325
symmetric NOR 330
synchronizer 454
synchronous 465
synchronous reset 408, 858
synchronous up/down counter 684
synchrony 464
syndrome 604, 688
synthesis 523
system 431
systematic 788
systematic mismatch 236
systematic variability 236
system-on-chip 35, 257

T 81, 91
T_c 385
t 196
t_s 423
TAP 610
tap 242
tap sequence 684

tapeout 61
tapering 557
tapped delay line 745
TARG 284
t_{coq} 386, 387
t_{cd} 159, 386
t_{cdq} 386, 387
TCK 611
TDDB 244
TDI 611
TDO 611
technology node 251
temperature 90, 232, 803
  coefficient 91, 146
  interconnect 216
  plot 354
ternary operator 852
test 567
  access method 623
  access port 610
  bench 60, 569, 579, 875
  fixture 575
  logic 579
  manufacturing 573, 588
  program 577
  reliability 626
  vector 60, 578, 875
  SOC 622
tester 575, 628
TestosterICs 628
t_f 159
THD 822
thermal expansion 761
thermal resistance 766
thermal voltage 81
thermometer code 806, 832
thickness 196, 233
thinox 126
t_{hold} 386, 387
threshold drop 93, 102, 351
threshold implant 117
threshold voltage 87, 91, 138, 233, 236, 293, 803
  multiple 327
through-hole 762
tilt 233
time borrowing 396
time-dependent dielectric breakdown 244
timing analysis 525, 531
timing diagram 387
timing types 419
timing-driven placement 532
TLB 747
T-model 205
TMS 611
token 383
top view 23
total harmonic distortion 822
TOX 287
t_{ox} 72
t_{poq} 386, 387
t_{pd} 159, 386
t_{pdf} 286
t_{pdq} 386, 387
t_{pdr} 286
t_r 159
track 33
tranif 874
transconductance 810
transfer characteristic 305
transient analysis 274, 276
transistor 1
  bipolar 3, 148, 365
  first 2
  first-order model 71
  folding 556
  high voltage 141
  MOS 3, 8
  pass 15
  plastic 141
  scaling 247
  switch model 9
  symbol 67
translation lookaside buffer 747
transmission gate 16, 102, 105, 345
transmission line 147, 312
transparent 20, 384, 859
transparent latch 416
tree 218
tree adder 661
trench 119
trench capacitor 734
t_{rf} 159
TRIG 284
triple-well process 117
tristate 17, 102, 855, 905
  buffer 783, 17
  inverter 18
TRST* 611
true input 336
true single-phase-clock 414
truth table 10
t_{setup} 386, 387
TSOP 762
TSPC 414
TT 290
tunneling 90, 188, 194, 742
twisted bitline 735
twisted differential signaling 227
two-phase
  clocks 418
  domino 430
  latches 385
  transparent latch 473
type 904
typical 233

u 902
uncertainty 236
unfooted 333
unfooted domino 438
uniform variation 231
unit 36
  transistor 32
unsaturated region 70
unsigned 681
unskewed 97
unskewed gate 325
up/down counter 684
use 905
useful operating life 239
user manual 547

valency 646, 665
valid 419
varactor 840
variability 236
VCD 578
VCO 790, 839
V_{DD} 9, 767
V_{di} 69
V_{dsat} 73
velocity saturation 84, 93, 300
velocity saturation index 84
verification 568, 579
Verilog 48, 849
vernier 130
version control 584
very large-scale integration 4

V_{gs} 68
VHDL 48, 895
VHSIC 895
via 124, 129
  resistance 199
victim 208
V_{IH} 98
V_{IL} 98
virtual component 486, 545
VLSI 4
V_{OH} 98, 782
V_{OL} 98, 782
volatile 714
voltage 803
voltage ladder 825
voltage regulator 775
voltage source 275
voltage-controlled delay line 792
voltage-controlled oscillator 790, 839
v_{sat} 84
V_{SS} 9
V_t 67, 293
  drop 351
V_{t0} 87
v_T 81
VTO 287

W 72, 92
W_{drawn} 92
w 196
W element 312
W/L ratio 31
wafer 8, 61, 114
  bumping 764
  cost 540
waffle capacitor 774
Wallace tree multiplier 704
war stories 629
wave pipelining 464
WAVES 562
well 117, 126
well contact 24
wet etch 124
when 898
white buffer 648
width 31, 196, 219
wire 196, 852
  bond 62
  engineering 219
  model 312
  scaling 249
wired-OR 330
wireless clock distribution 807
wiring channel 59
WMAX 289
WMIN 289
womanpower
  see personpower
word 714
word line 181
wordline 714, 715, 730
wrapper 622
writeability 719

X 11
x 855, 903
XL 92, 310
XNOR 689
xnor 874
XOR 689
xor 874, 897
XW 92, 310

Y chart 37, 38, 480
yield 540, 756
  analysis 609

Z 11, 17
z 855, 902, 903, 905
zero detector 679
zero insertion force socket 571
ZIF socket 571
zipper 56
zipper domino 344

Abbreviated MOSIS design rules					
Layer	Rule	Description	SCMOS	SUBM	DEEP
Well	1.1	Width	10	12	12
	1.2	Spacing to well at different potential	9	18	18
	1.3	Spacing to well at same potential	6	6	6
Active (diffusion)	2.1	Width	3	3	3
	2.2	Spacing to active	3	3	3
	2.3	Source/drain surround by well	5	6	6
	2.4	Substrate/well contact surround by well	3	3	3
	2.5	Spacing to active of opposite type	4	4	4
Poly	3.1	Width	2	2	2
	3.2	Spacing to poly over field oxide	2	3	3
	3.2a	Spacing to poly over active	2	3	4
	3.3	Gate extension beyond active	2	2	2.5
	3.4	Active extension beyond poly	3	3	4
	3.5	Spacing of poly to active	1	1	1
Select	4.1	Spacing from substrate/well contact to gate	3	3	3
	4.2	Overlap of active	2	2	2
	4.3	Overlap of substrate/well contact	1	1	1.5
	4.4	Spacing to select	2	2	4
Contact (to poly or active)	5.1, 6.1	Width (exact)	2x2	2x2	2x2
	5.2b, 6.2b	Overlap by poly or active	1	1	1
	5.3, 6.3	Spacing to contact	2	3	4
	5.4, 6.4	Spacing to gate	2	2	2
	5.5b	Spacing of poly contact to other poly	4	5	5
	5.7b, 6.7b	Spacing to active/poly for multiple poly/active contacts	3	3	3
	6.8b	Spacing of active contact to poly contact	4	4	4
Metal1	7.1	Width	3	3	3
	7.2	Spacing to metal1	2	3	3
	7.3, 8.3	Overlap of contact or via	1	1	1
	7.4	Spacing to metal for lines wider than 10 λ	4	6	6
Via1–Via(N-1)	8.1, 14.1, …	Width (exact)	2x2	2x2	3x3
	8.2, 14.2, …	Spacing to via on same layer	3	3	3
	8.4	Spacing to contacts (if no stacked vias)	2	2	n/a
	8.5	Spacing of via1 to poly or active edge	2	2	n/a
	14.4	Spacing of via2 to via1 (if no stacked vias)	2	2	n/a
Metal2–Metal(N-1)	9.1, …	Width	3	3	3
	9.2, …	Spacing to same layer metal	3	3	4
	9.3, …	Overlap of via	1	1	1
	9.4, …	Spacing to metal for lines wider than 10 λ	6	6	8
Metal3 (3-layer process)	15.1	Width	6	5	n/a
	15.2	Spacing to metal3	4	3	n/a
	15.3	Overlap of via2	2	2	n/a
	15.4	Spacing to metal for lines wider than 10 λ	8	6	n/a

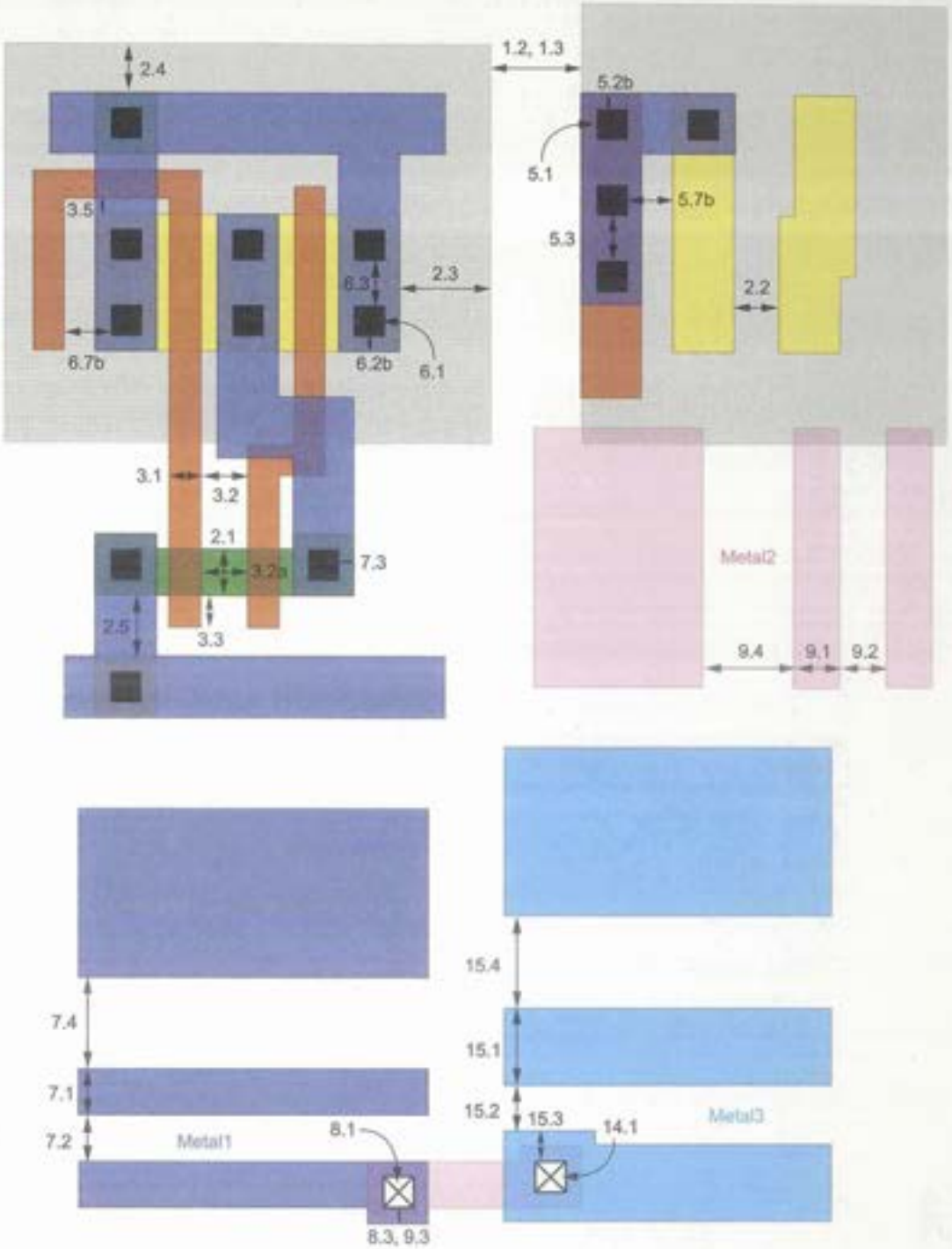


Fig 3.11 MOSIS Design Rules

CMOS VLSI

CMOS VLSI DESIGN

Third Edition

This best-selling text on CMOS VLSI design has been extensively revised to include details on modern techniques for the design of complex and high-performance CMOS systems on a chip. Covering CMOS design from the digital systems level to the circuit level, *CMOS VLSI Design*, Third Edition includes both an explanation of fundamental principles and a guide to good design practices.

Neil H.E. Weste and new coauthor David Harris draw upon their extensive industry and classroom experience to explain modern practices of chip design. Having worked at such companies as Intel, Cisco, and Bell Labs, the authors bring a real-world perspective to the topic by sharing lessons learned in their professional practices.

New to this Edition

- Comprehensive coverage of key CMOS design issues: circuits, interconnect, and clocking
- Detailed treatment of low-power techniques
- Integration of logical effort for efficient design of fast circuits
- Tutorial chapters on Verilog, VHDL, and SPICE (circuit) simulation
- Extensive exercises and worked-out examples for learning reinforcement

About the Authors

Neil H.E. Weste is Director of NHEW R&D Pty Ltd. and an Adjunct Professor at both Macquarie University and the University of Adelaide. He received his B.Sc., B.E.(Elec) and Ph.D. degrees from the University of Adelaide. His interests are wireless technologies, systems on a chip, analog, RF and digital IC design, and technology incubation.

David Harris is an Associate Professor of Engineering at Harvey Mudd College in Claremont, CA. David received his Ph.D. from Stanford University and his S.B. and M. Eng. degrees from MIT. His research interests include high-speed CMOS VLSI design, microprocessors, and computer arithmetic. He holds seven patents, is the author of two other VLSI books, and has designed chips at Sun Microsystems, Intel, Hewlett-Packard, and Evans & Sutherland.

PEARSON
Addison Wesley

Addison-Wesley Computing • Leading Authors • Quality Products
Log on to **aw-bc.com/computing** for a full list of titles.

90000
9 780321 149015
ISBN 0-321-14901-7