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.

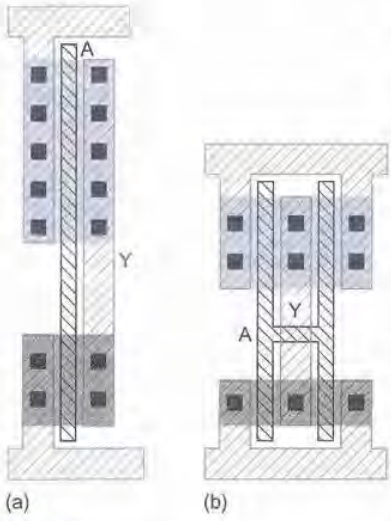


FIG 8.54 Folding transistors

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.

TSMC-1012
U.S. Patent 8,796,779

Table 8.6 Diffusion dimensions for unfolded and folded transistors

Transistor	Unfolded			Folded		
	Width (λ)	Drain Area (λ²)	Drain Perimeter (λ)	Width (λ)	Drain Area (λ²)	Drain Perimeter (λ)
N1	12	60	22	6 + 6	36	12
P1	24	120	34	12 + 12	72	12

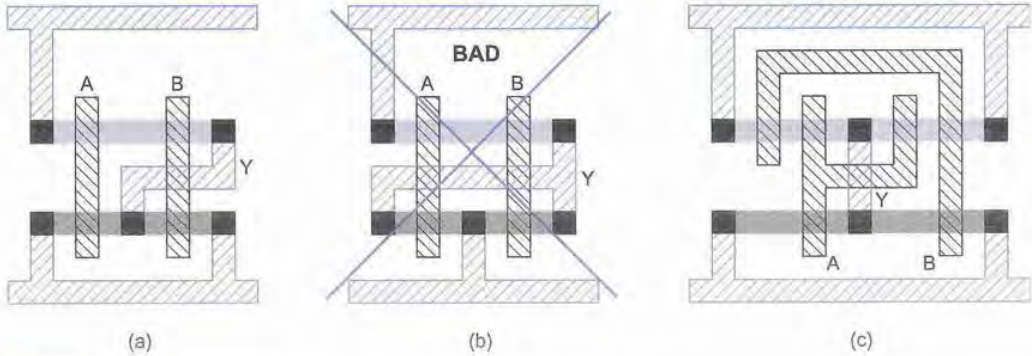


FIG 8.55 Parallel transistor layout

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

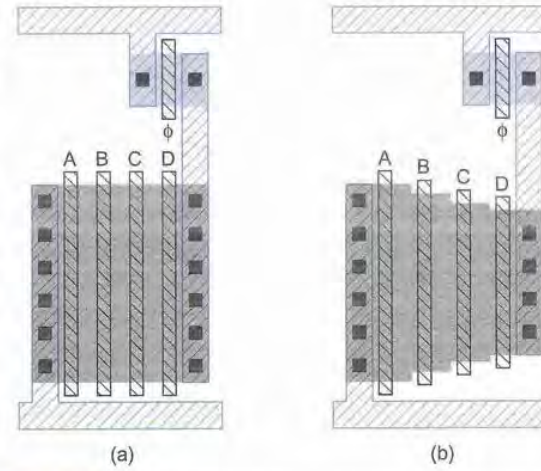


FIG 8.56 Tapering

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.

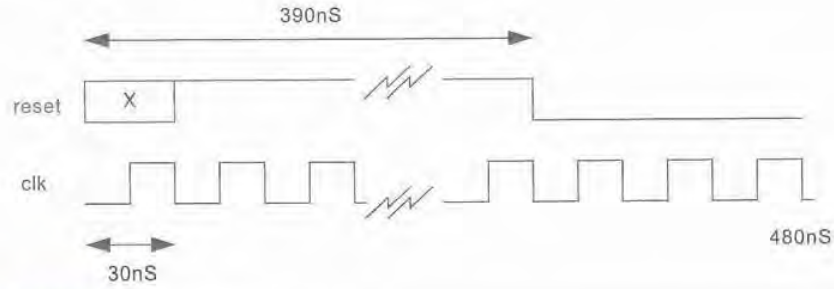


FIG 8.57 WAVES example waveforms

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 $Y_{ws} = 80\%$ and $Y_{pa} = 98\%$ 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

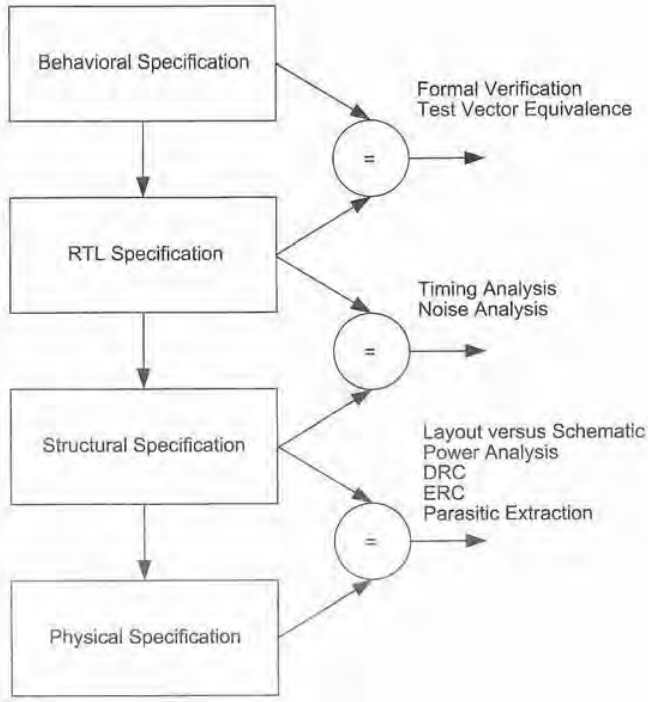


FIG 9.1 Functional equivalence at various levels of abstraction

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).

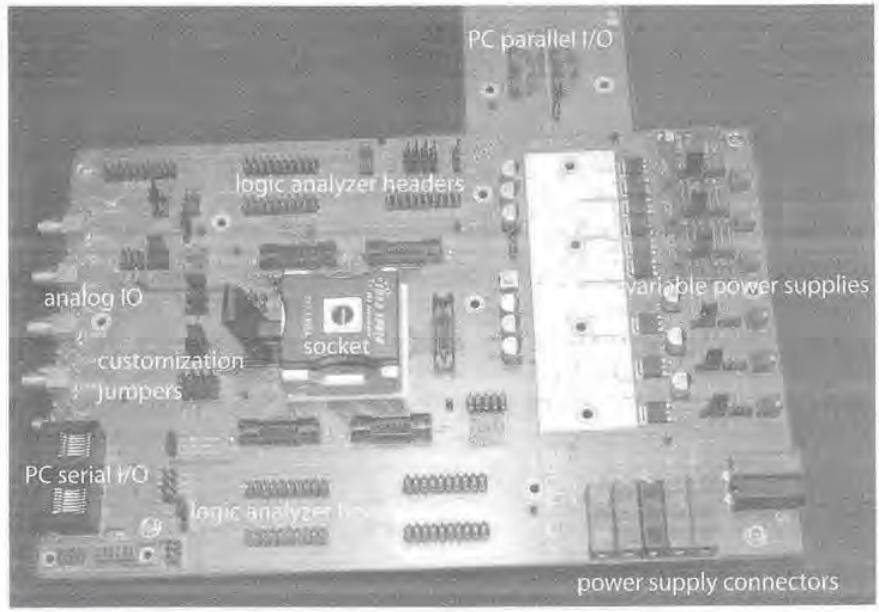


FIG 9.2 Typical test board

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

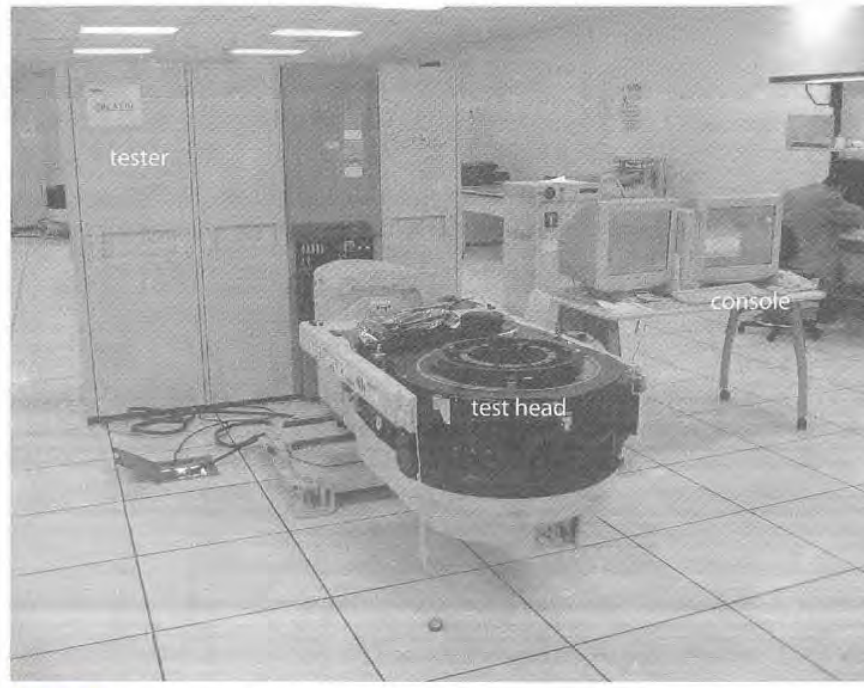


FIG 9.3 The Teradyne Catalyst: A typical production tester (Photo: John Haddy, Cisco Systems)

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.

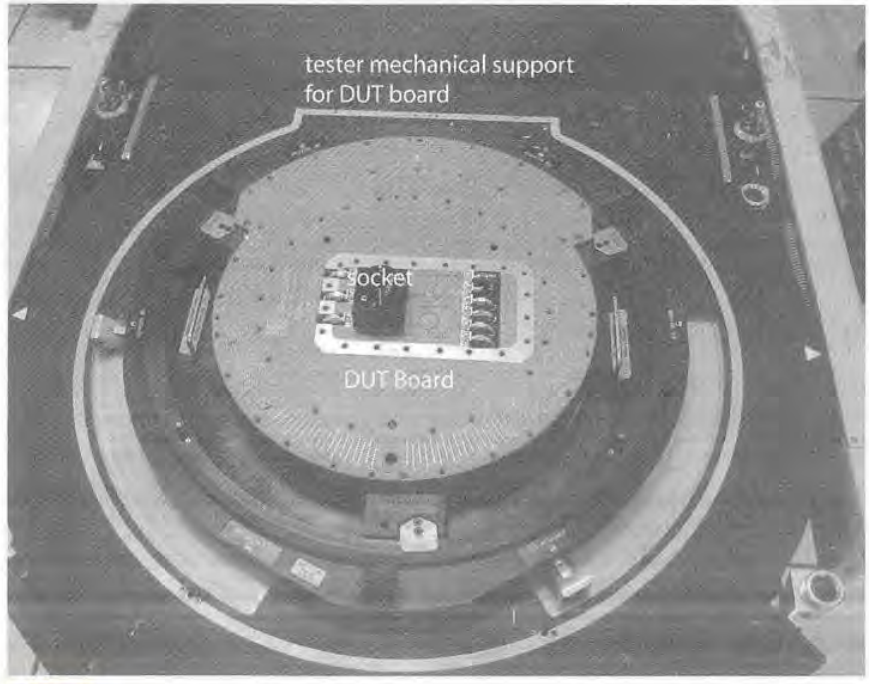


FIG 9.4 Tester load board in test head
(Photo: John Haddy, Cisco Systems)

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_s). 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.

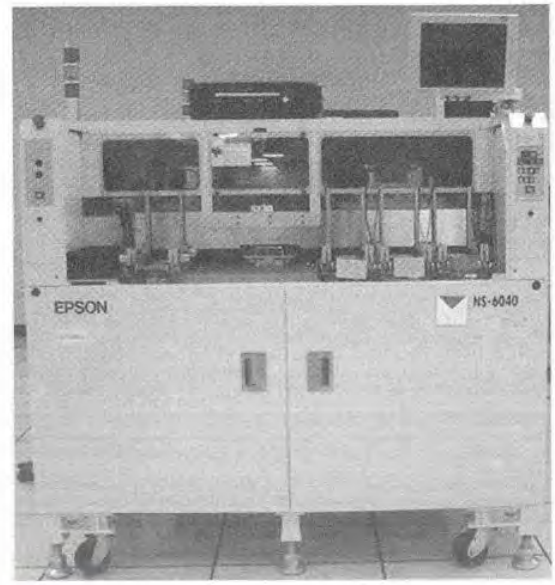


FIG 9.5 Photograph of an Epson NS-6040 IC handler (Source: John Haddy—Cisco Systems)

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

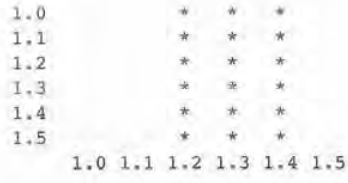
** indicates a failure*



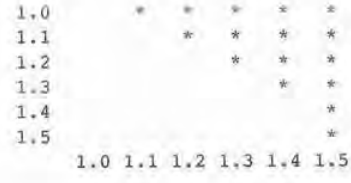
Normal
Well-behaved shmoo
Typical Speedpath



"Brick Wall"
Bistable
Initialization



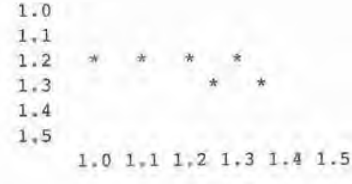
"Wall"
Fails at a certain voltage
Coupling, charge share, races



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



"Floor"
Works at high but not low frequency
Leakage



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

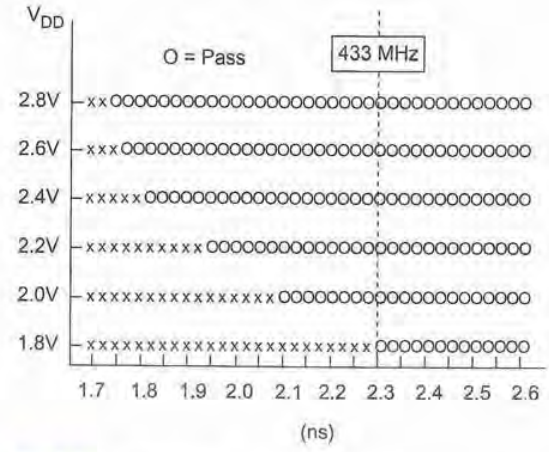


FIG 9.6 Alpha 21164 shmoo

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.

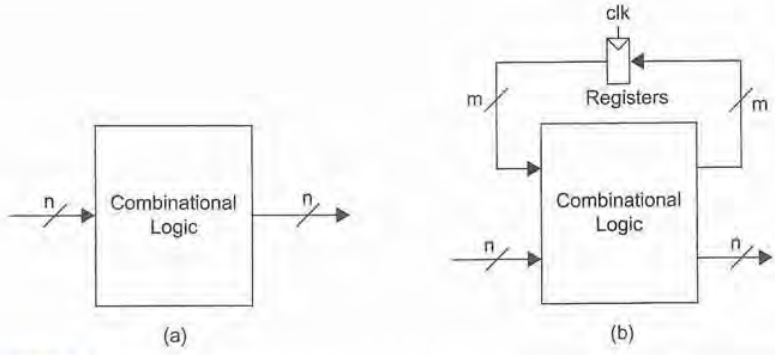


FIG 9.7 The combinational explosion in test vectors

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-l, S-A-l). 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.

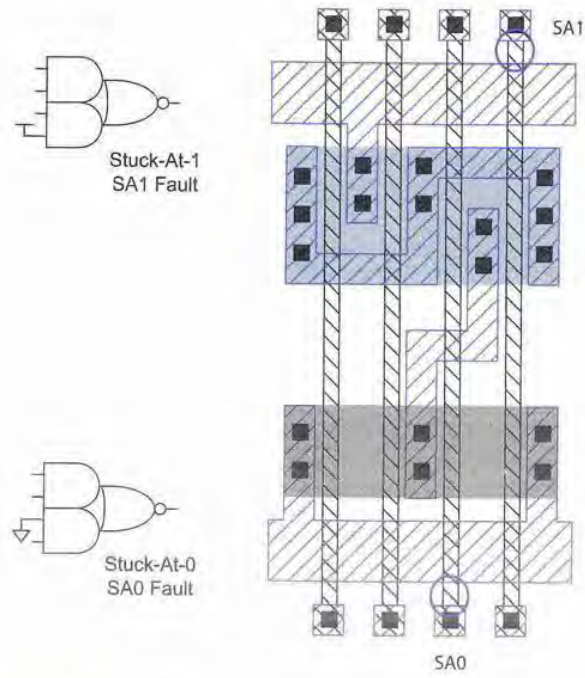


FIG 9.8 CMOS stuck-at faults

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

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

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.

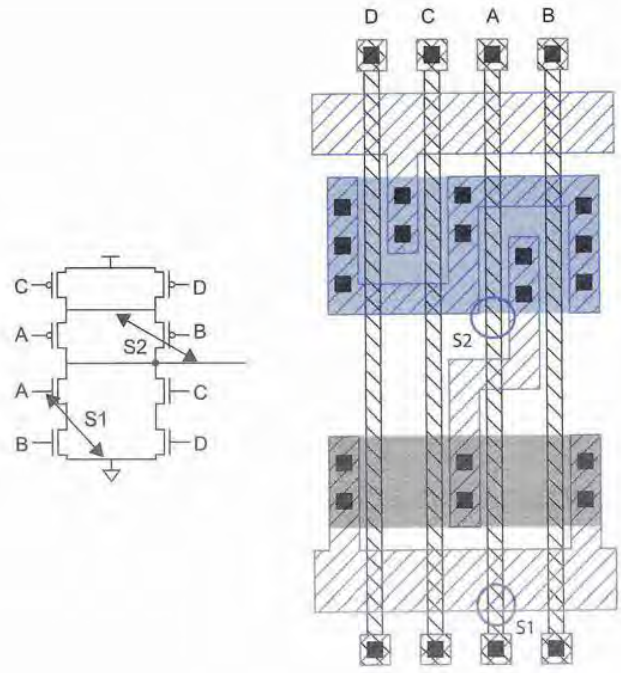


FIG 9.9 CMOS bridging faults

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.

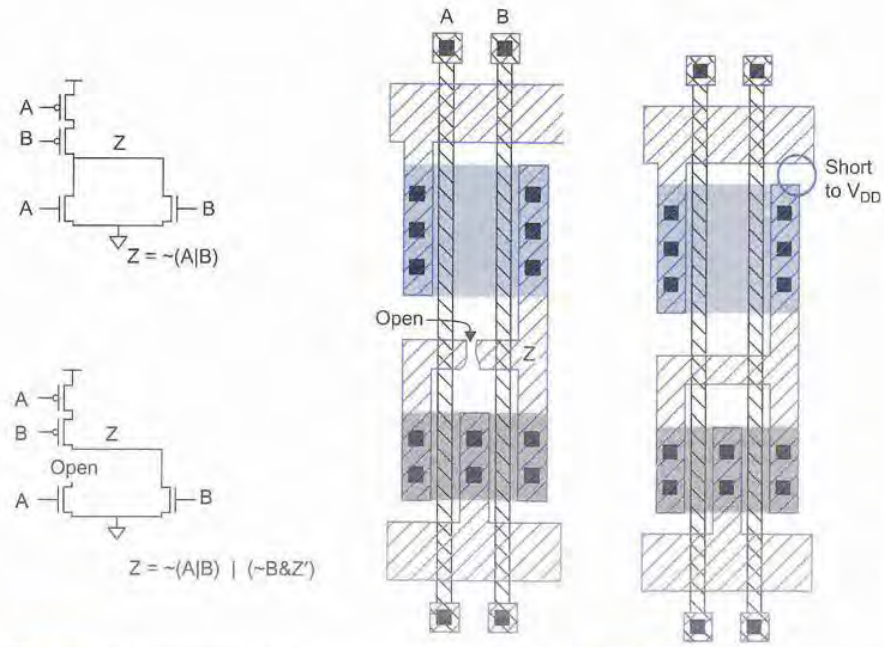


FIG 9.10 A CMOS open fault that causes sequential faults

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_{pdf}. In addition, the fault now becomes sequential as the detection of the fault depends on the previous state of the gate.

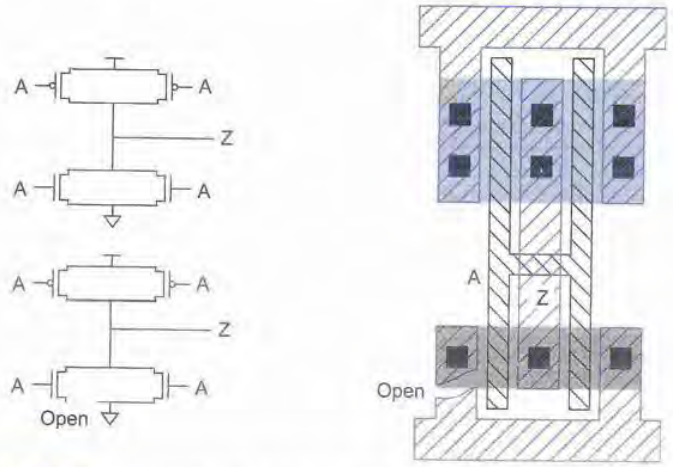


FIG 9.12 An example of a delay fault

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.

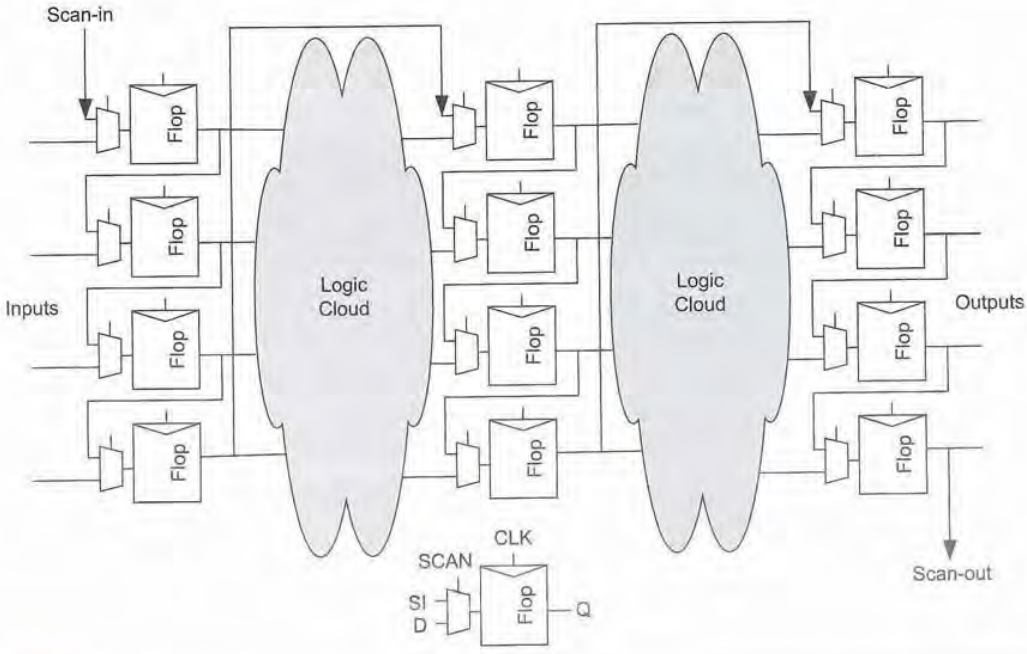


FIG 9.13 Scan-based testing

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.

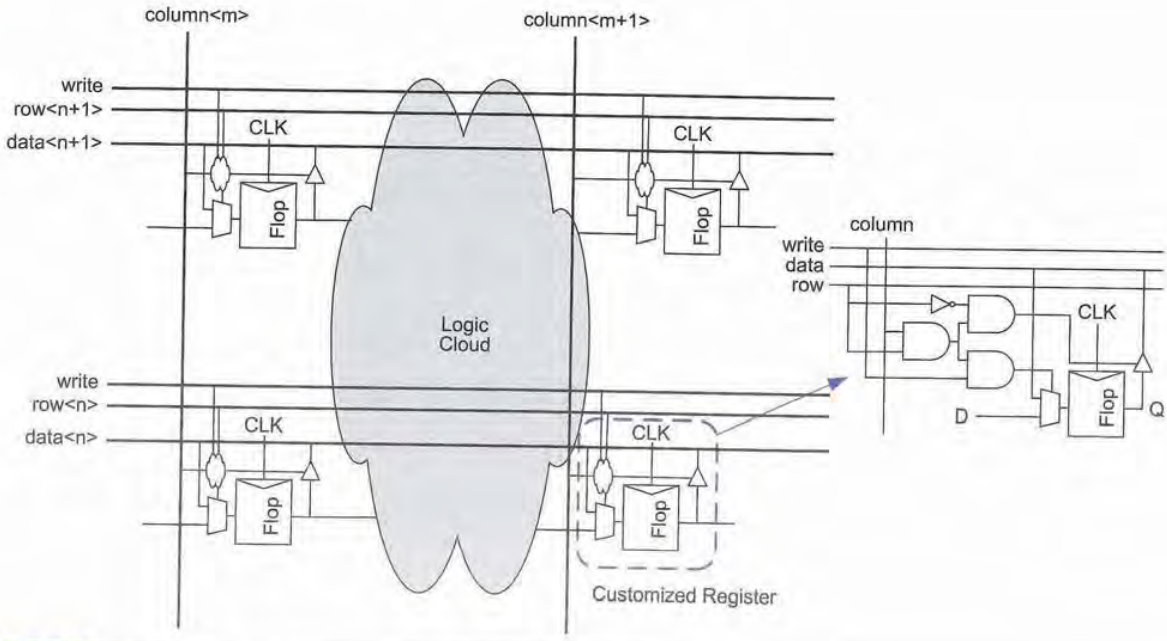


FIG 9.14 Parallel scan—basic structure

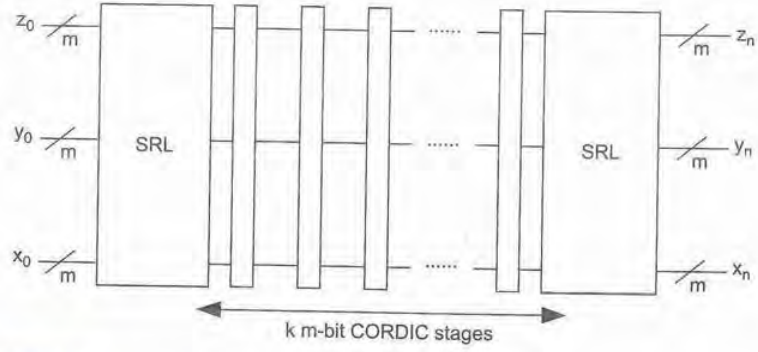


FIG 9.15 Example of partial scan as applied to the CORDIC processor

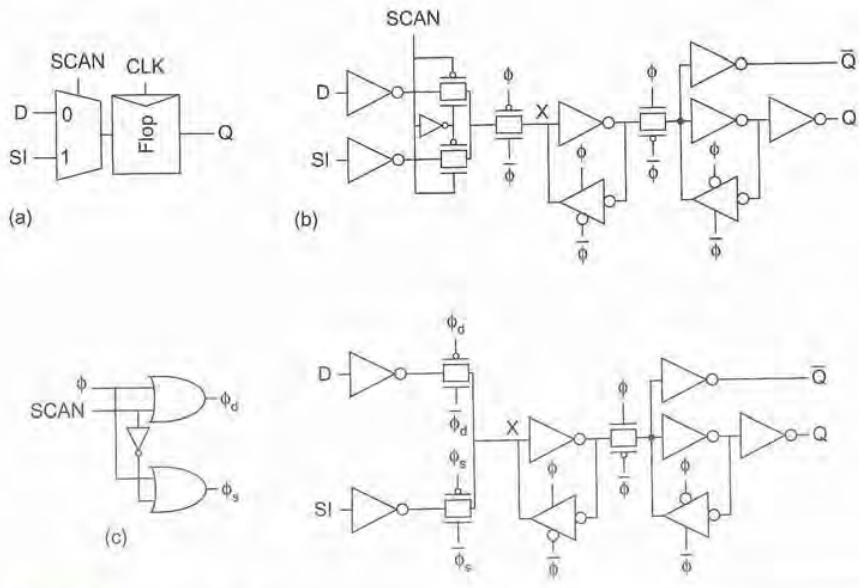


FIG 9.16 Scannable flip-flops

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

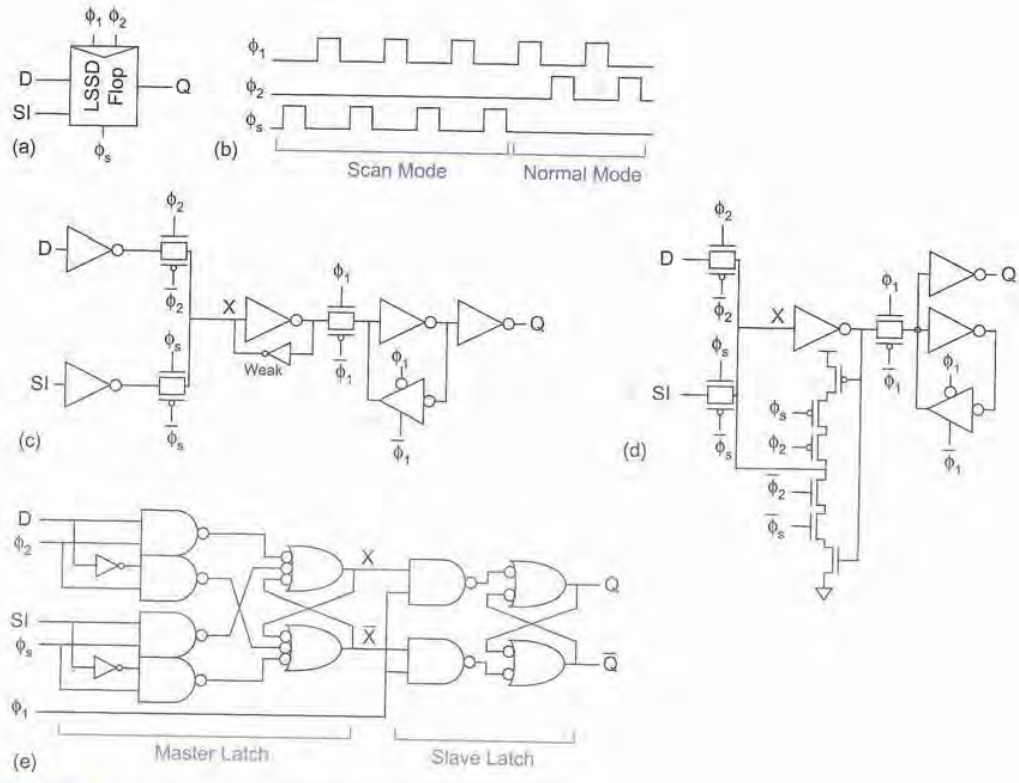


FIG 9.17 LSSD flip-flops

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

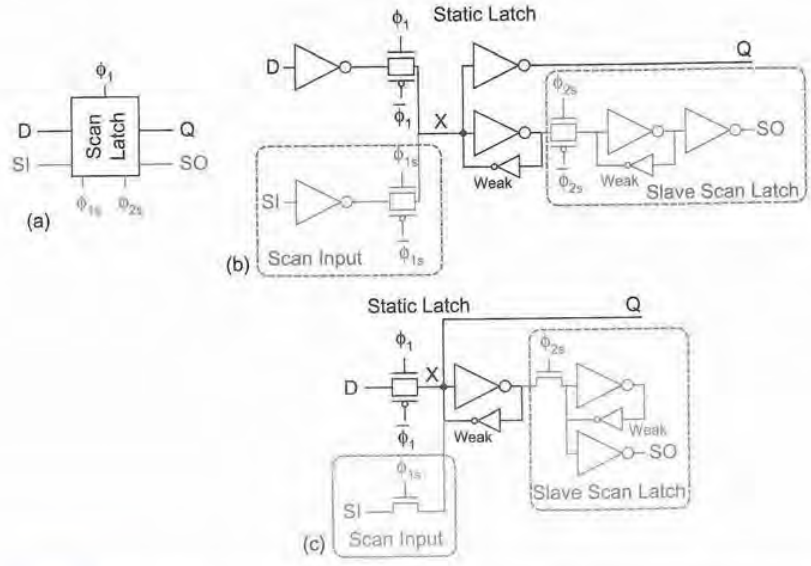


FIG 9.18 Scannable transparent latches

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

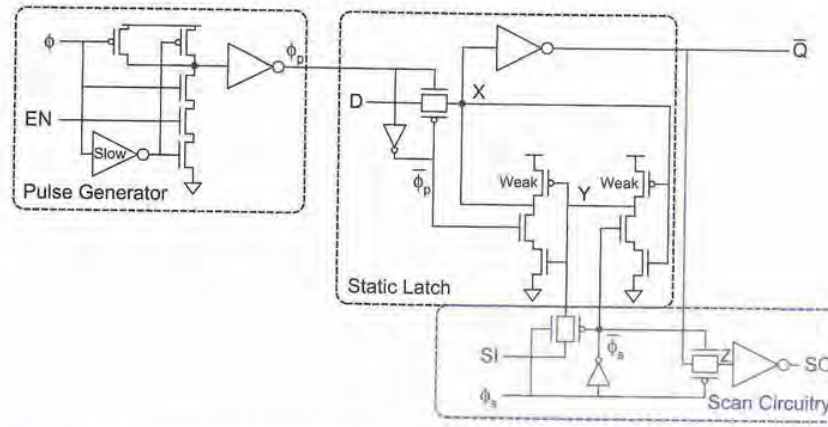


FIG 9.19 Scannable Naffziger pulsed latch

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_{4s} 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)

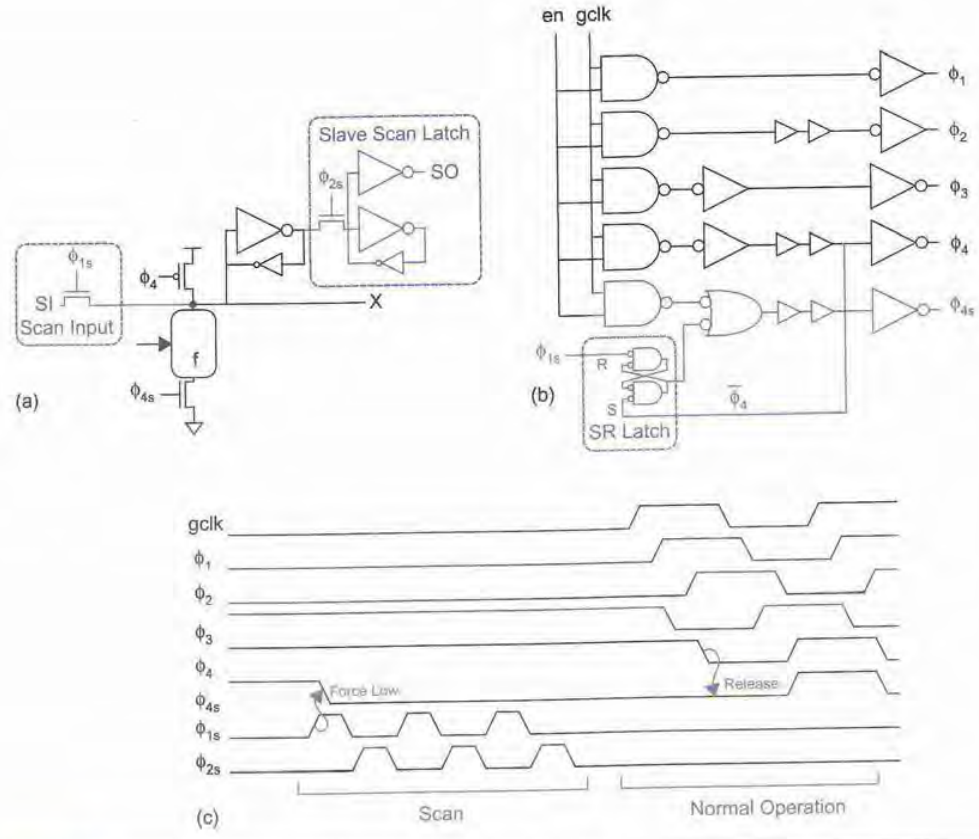


FIG 9.20 Scannable dynamic gate for four-phase skew-tolerant domino

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,

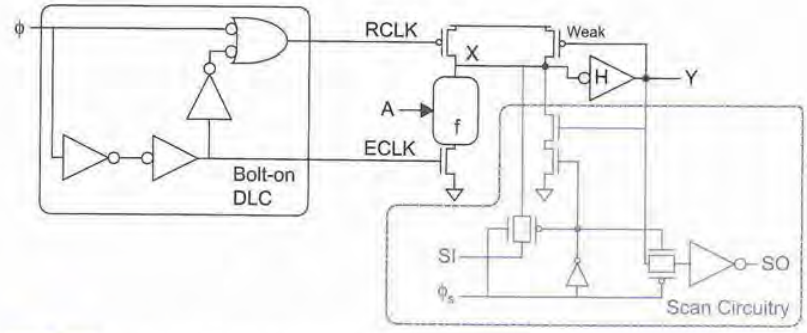


FIG 9.21 Itanium 2 scannable domino gate

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.

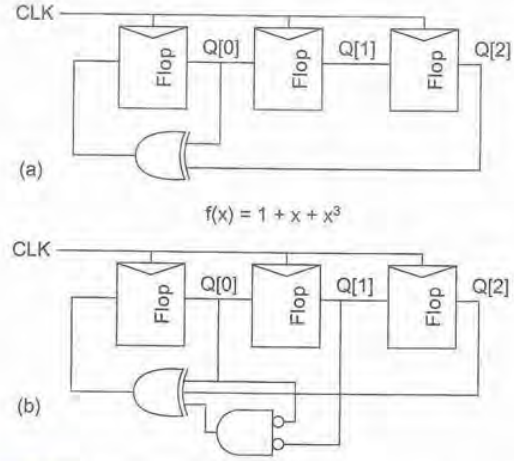


FIG 9.22 Pseudo-random sequence generator

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

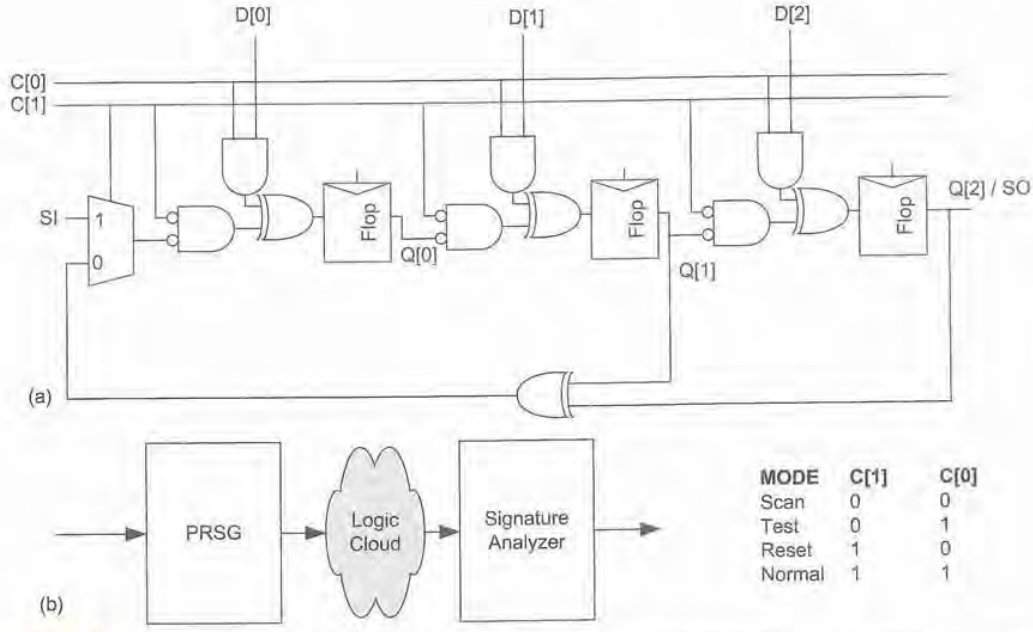


FIG 9.23 Built-In Logic Block Observation (BILBO/BIST): (a) individual register (b) use in a system

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.

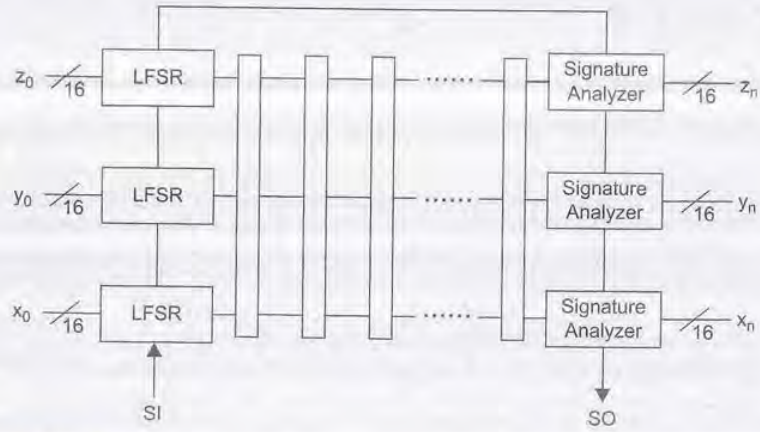


FIG 9.24 CORDIC with BIST

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.

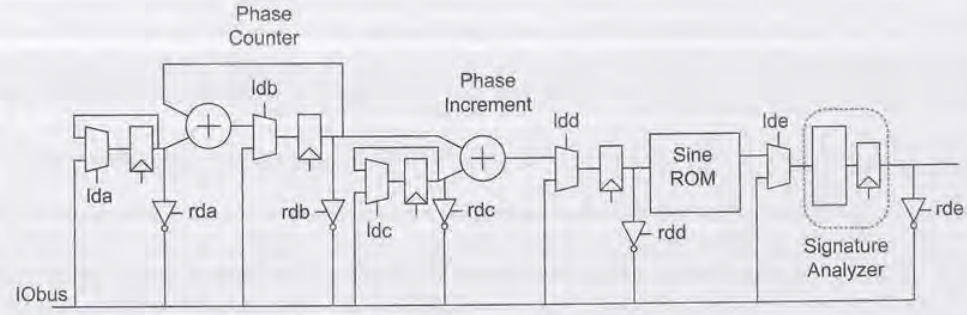


FIG 9.25 NCO ROM testing

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.

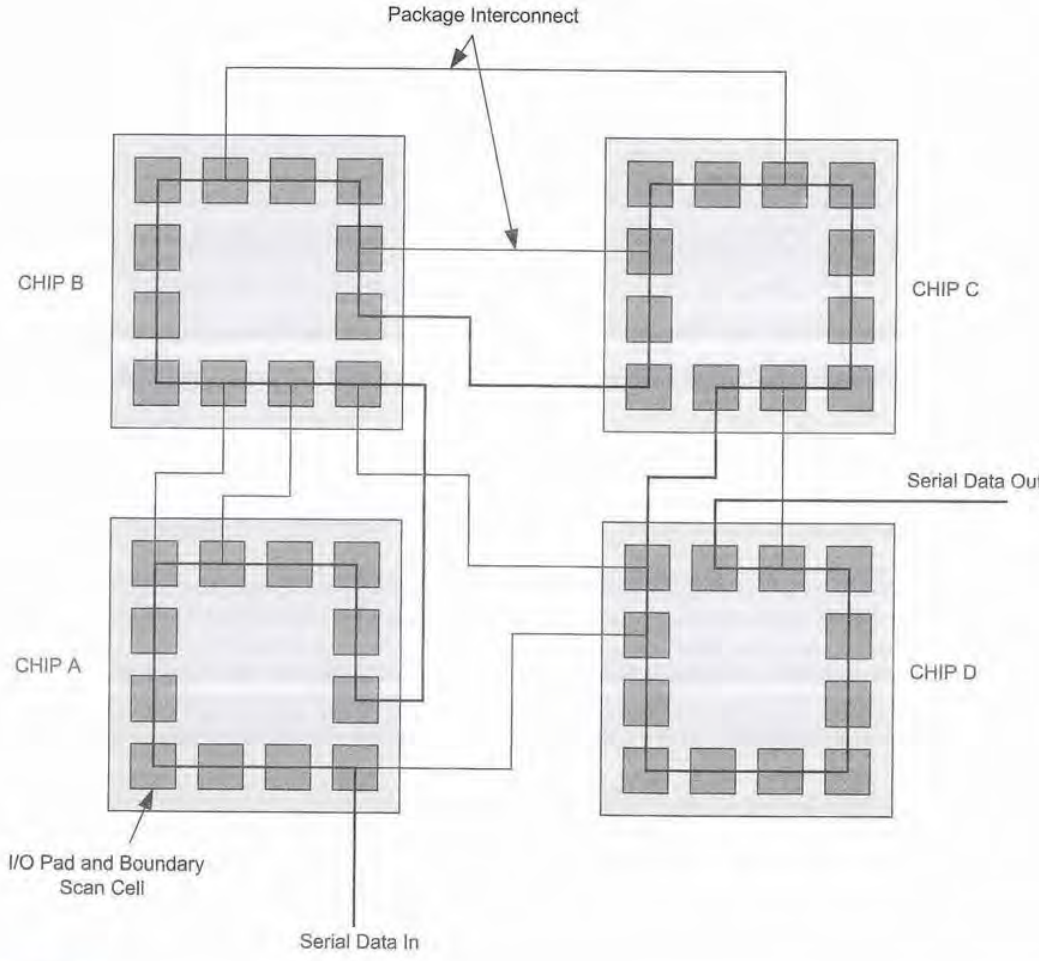


FIG 9.26 Boundary scan architecture

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.

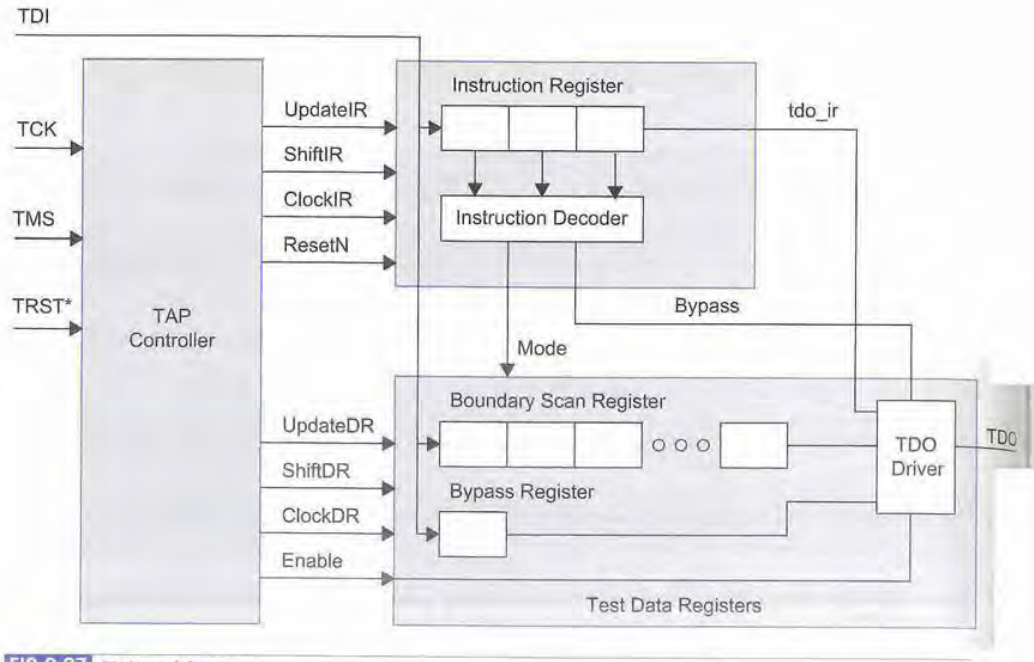


FIG 9.27 TAP architecture

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.

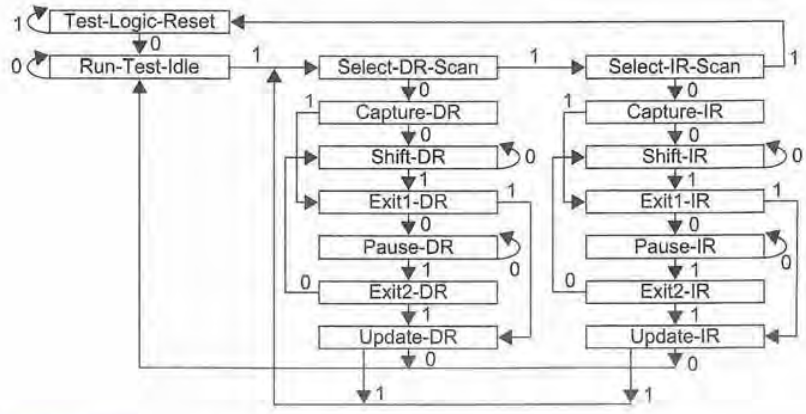


FIG 9.28 TAP controller state diagram

```
// 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.

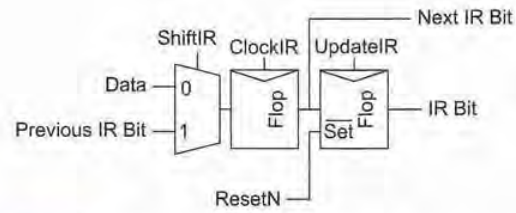


FIG 9.29 Instruction bit implementation

```
// 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.

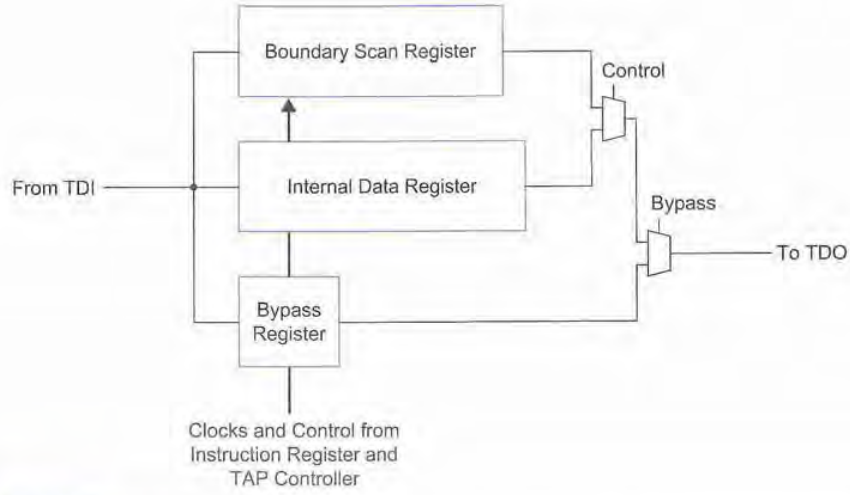


FIG 9.30 Test-data registers

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).

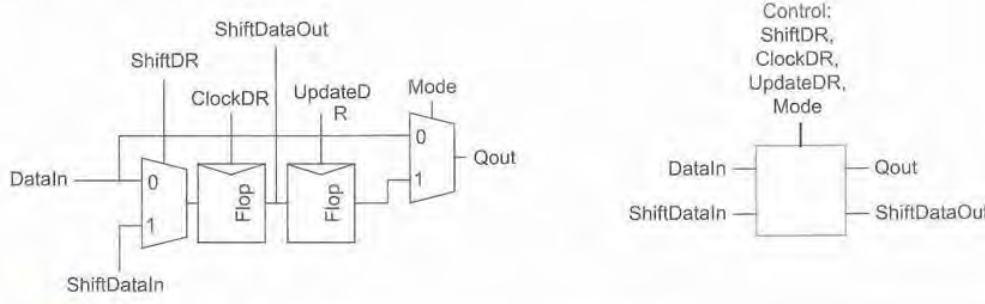


FIG 9.31 Boundary scan register bit

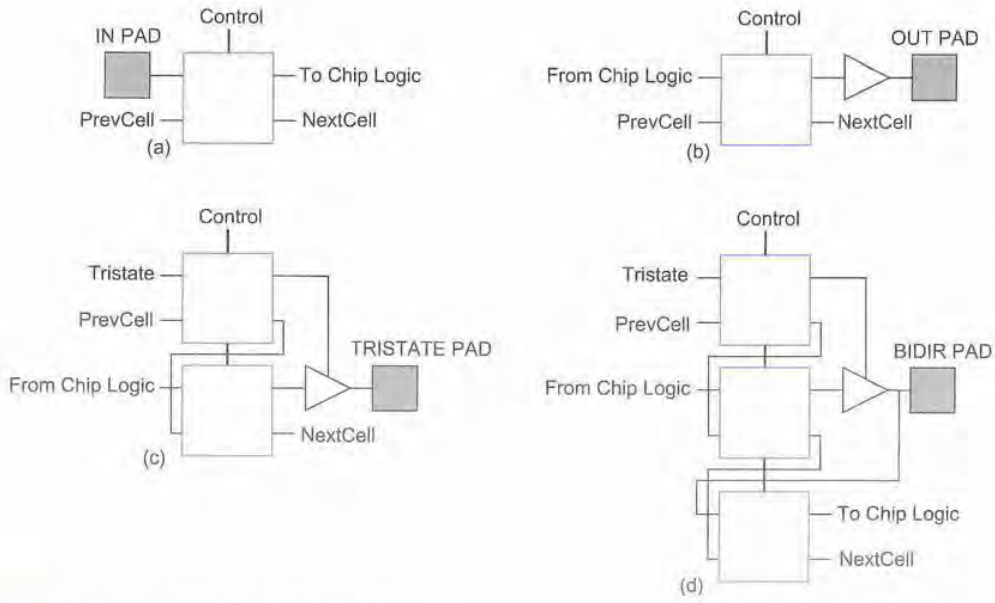


FIG 9.32 Boundary scan pad configuration

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.

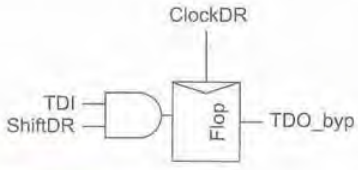


FIG 9.33 Bypass register

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.

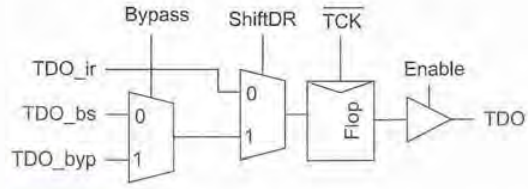


FIG 9.34 TDO driver

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.

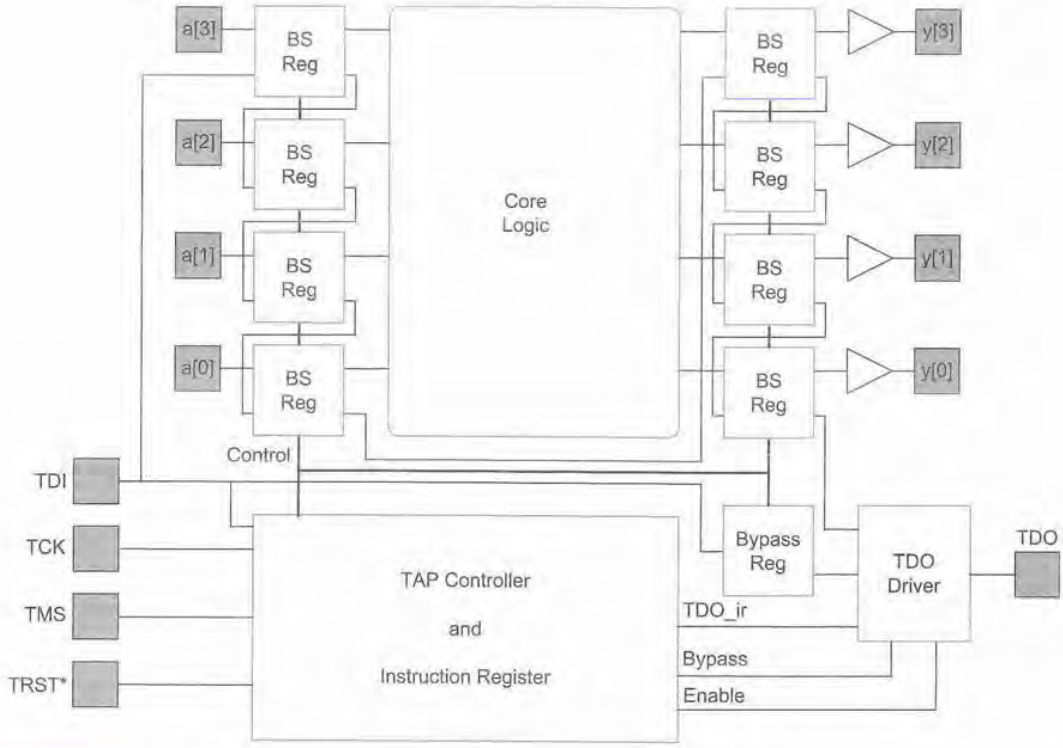


FIG 9.35 Complete boundary scan implementation

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.

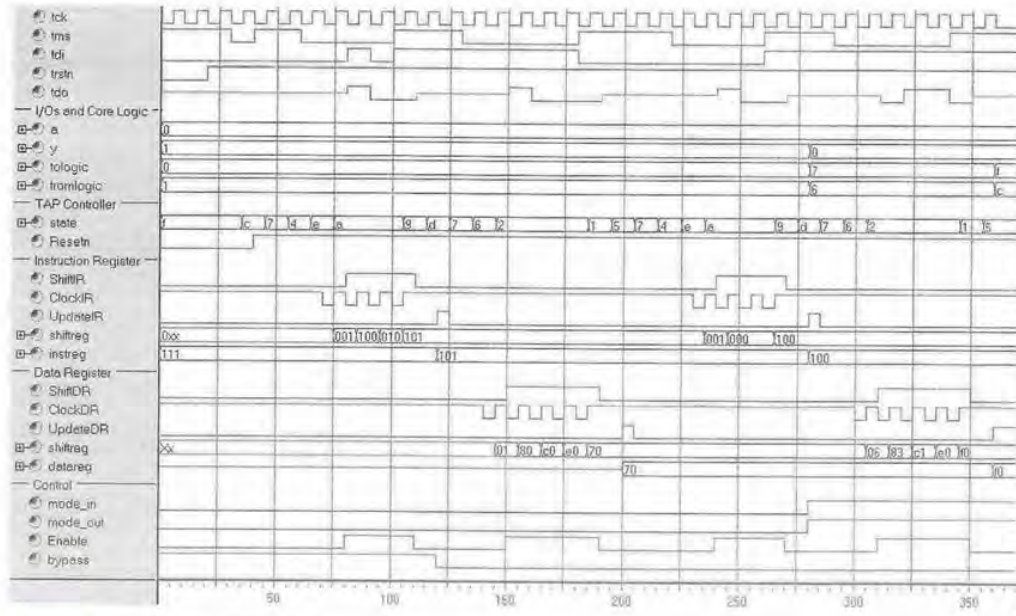


FIG 9.36 Boundary scan example waveforms

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.

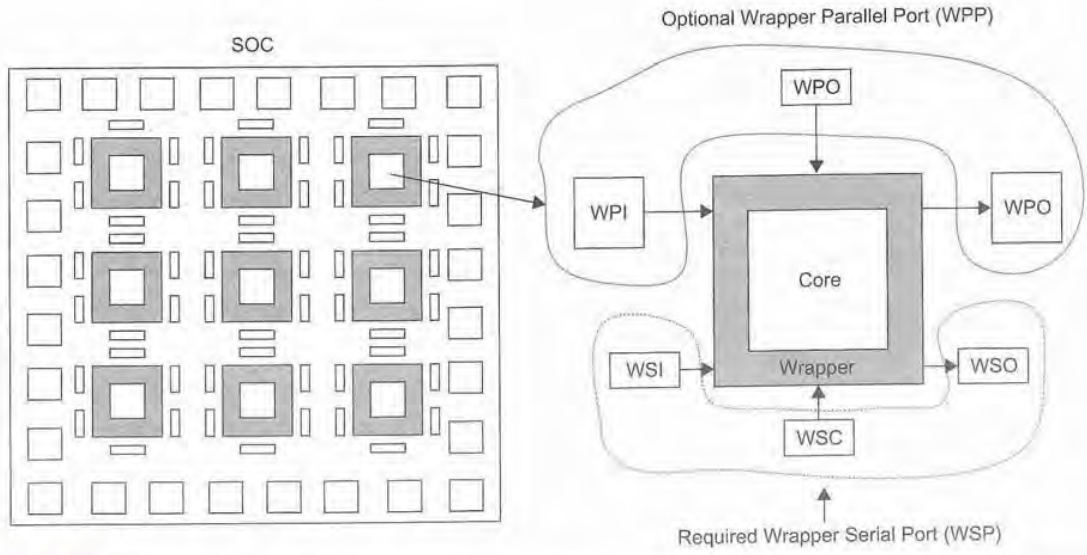


FIG 9.37 IEEE P1500 embedded core test strategy

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.

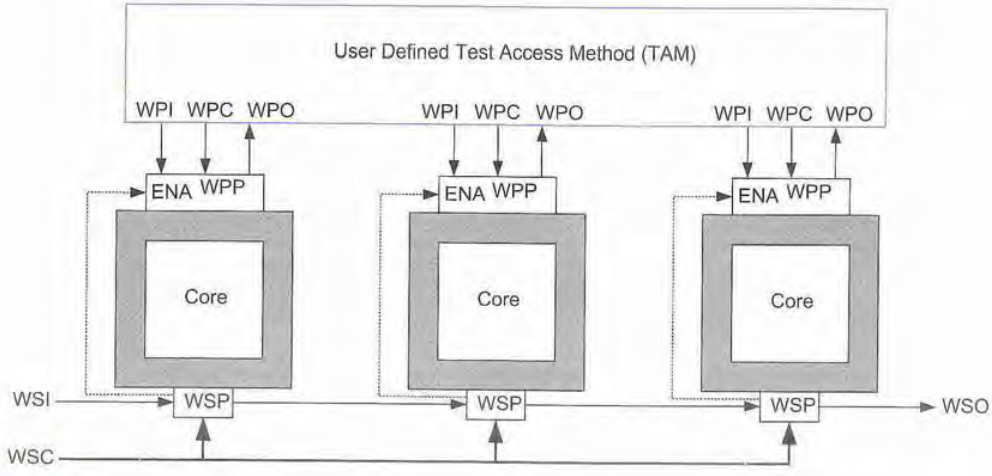


FIG 9.38 P1500 user-defined test access method (TAM)

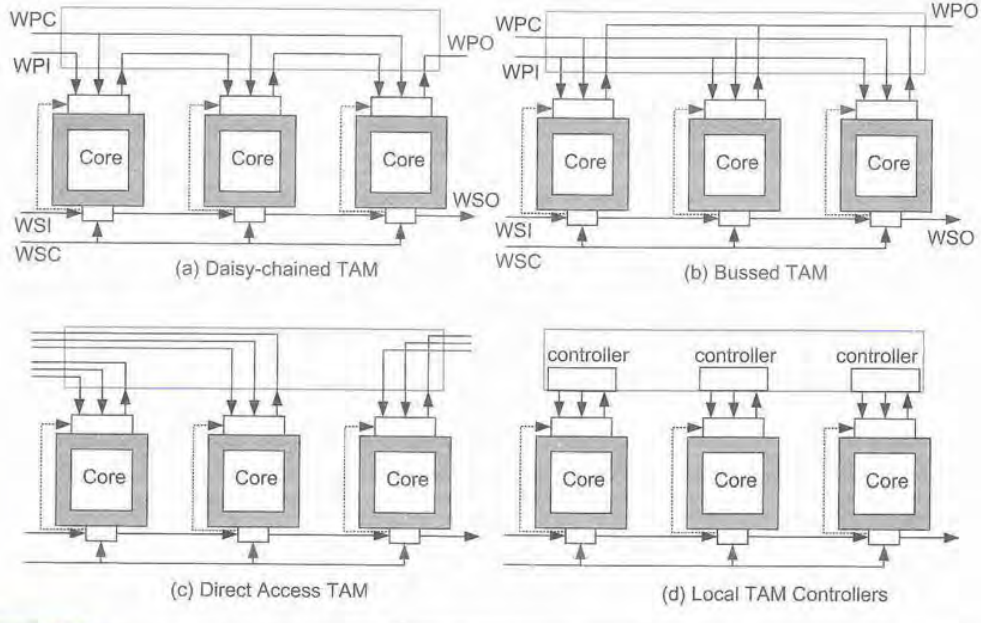


FIG 9.39 Various methods of TAM

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).

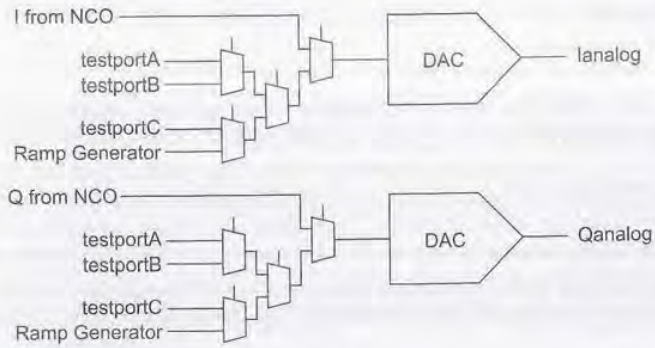


FIG 9.40 DAC testing and system debug

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.

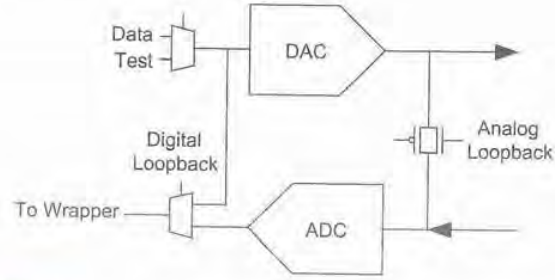


FIG 9.41 Analog and digital loopback



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.

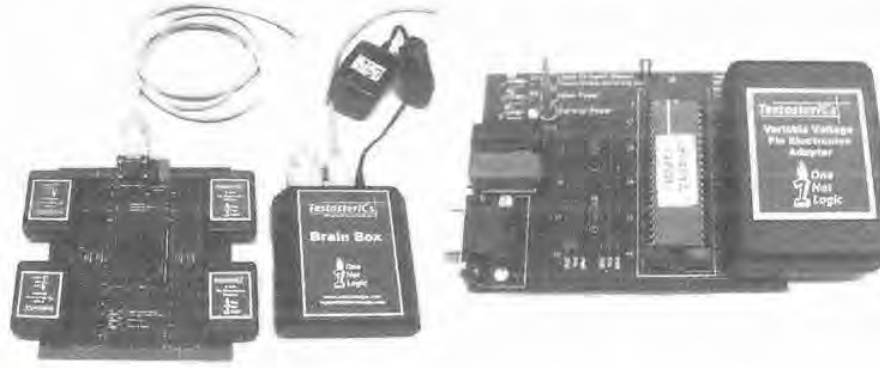


FIG 9.42 TestosterICs functional chip tester

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

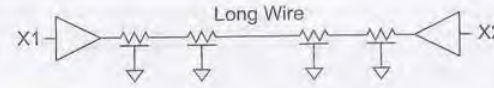


FIG 9.43 Long wire driven from both ends

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 precharged 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

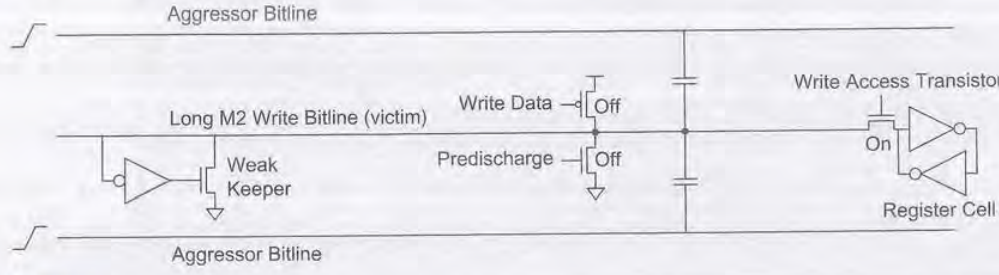


FIG 9.44 Register file write circuitry

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

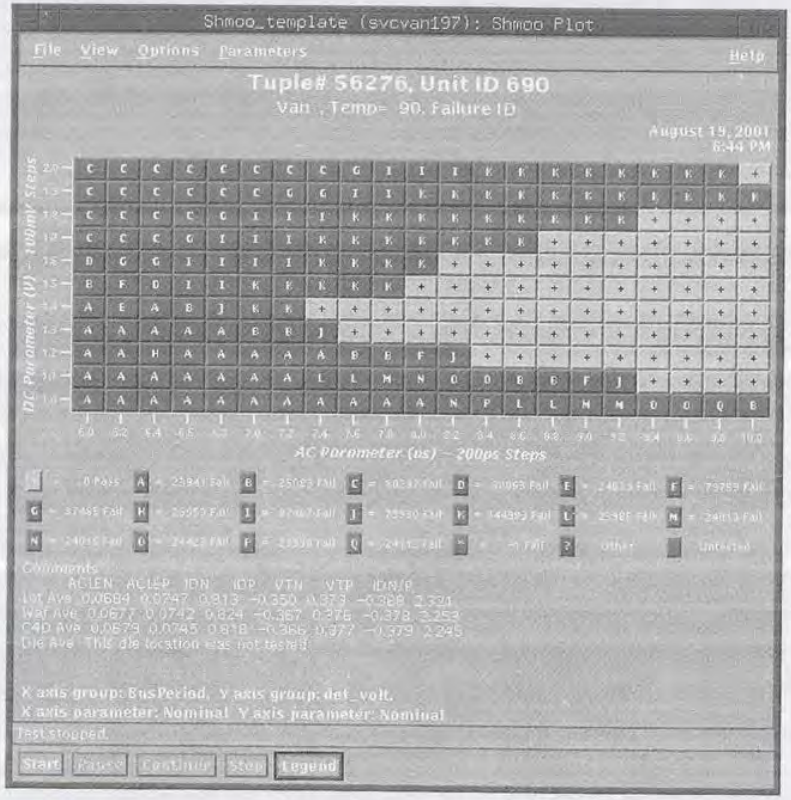


FIG 9.45 Flying saucer shmoo

[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.

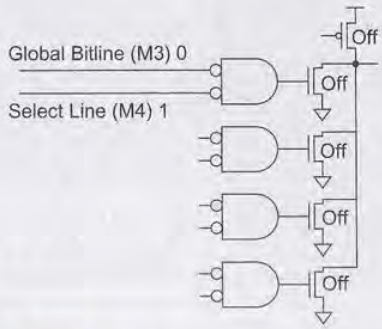


FIG 9.46 Dynamic bitline multiplexer

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-
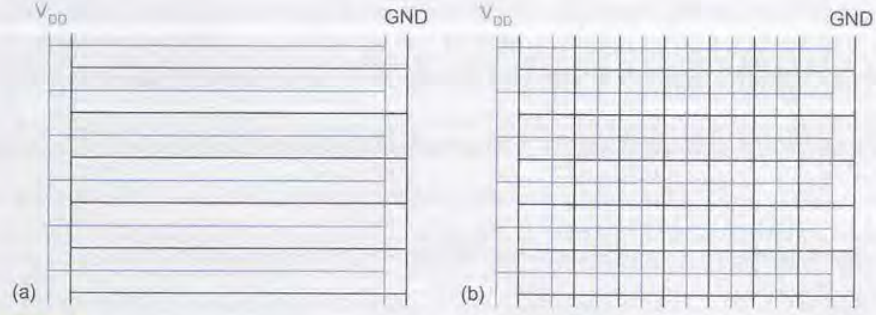


FIG 9.47 Power supply network

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}.

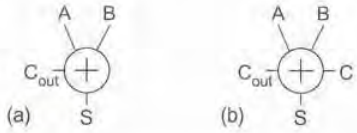


FIG 10.1 Half and full adders

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:

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

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.

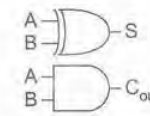


FIG 10.2 Half adder design

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.

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

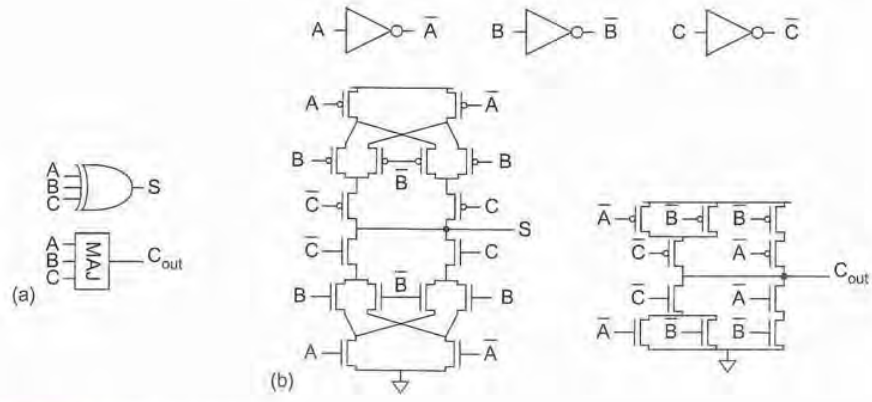


FIG 10.3 Full adder design

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).

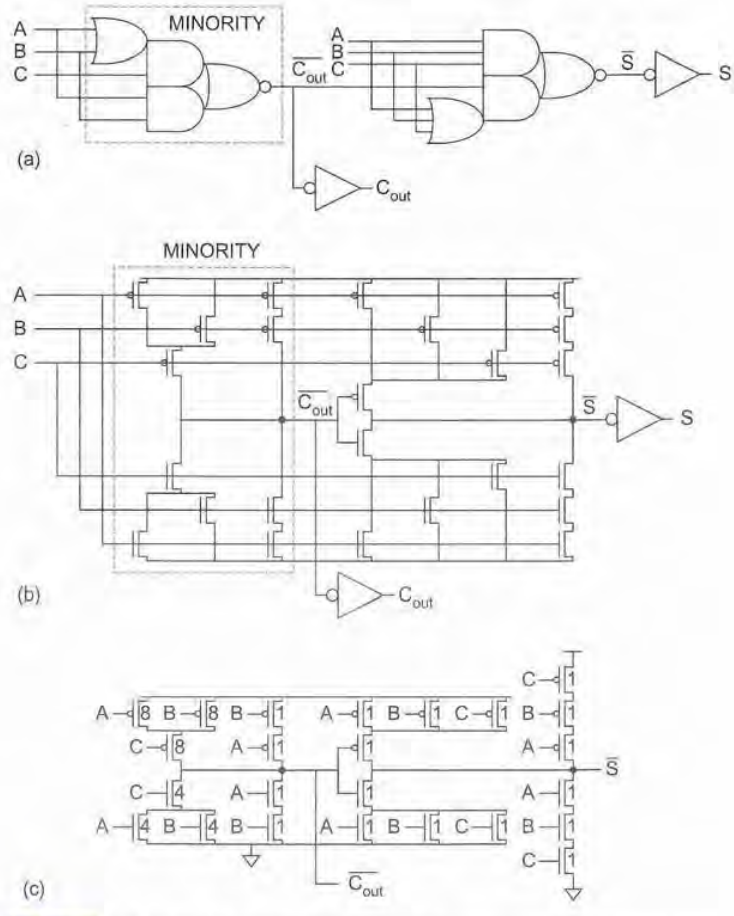


FIG 10.4 Full adder for carry-ripple operation

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.

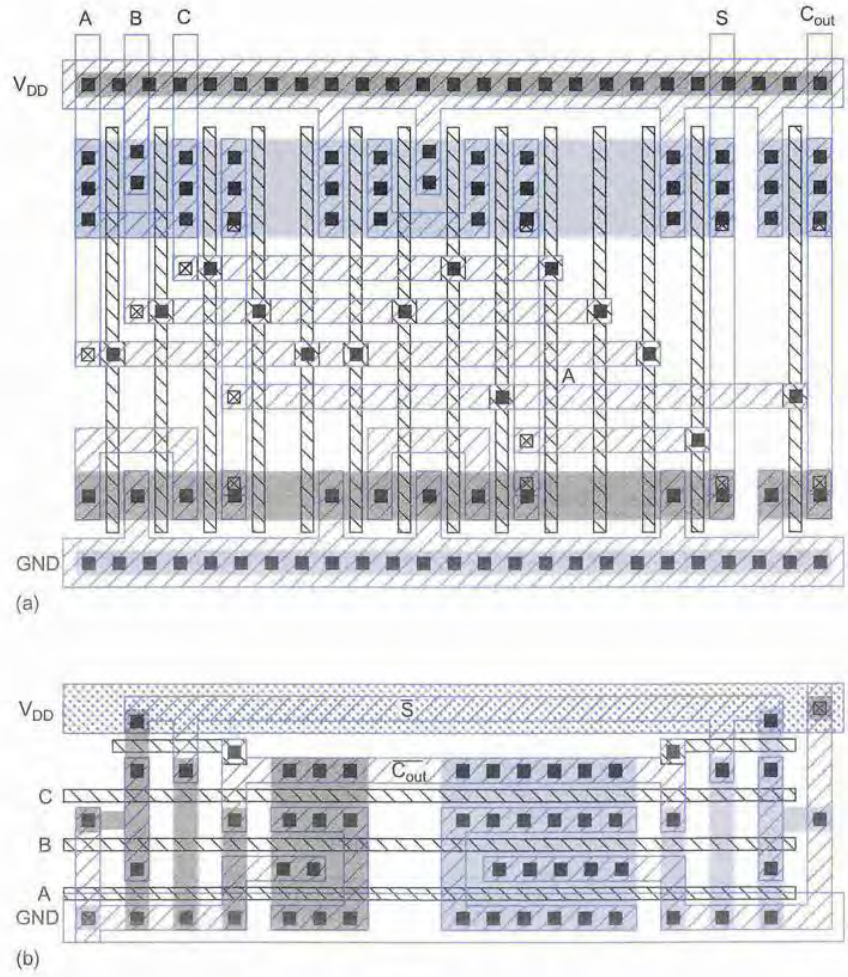


FIG 10.5 Full adder layouts. Color version on inside front cover.

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.

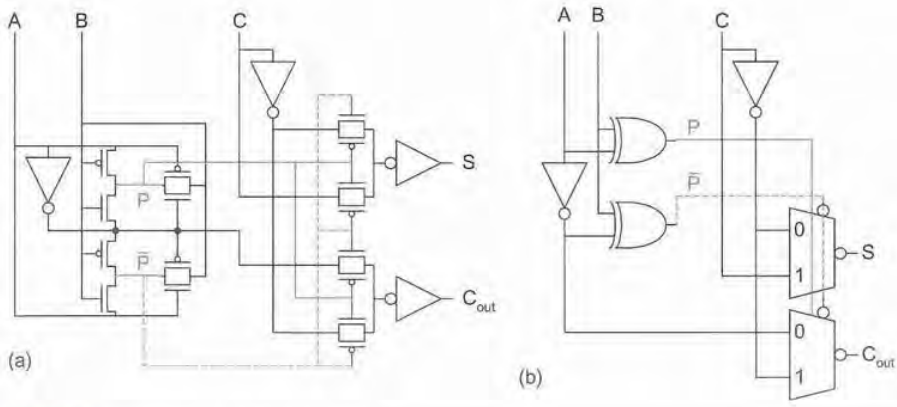


FIG 10.6 Transmission gate full adder

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.

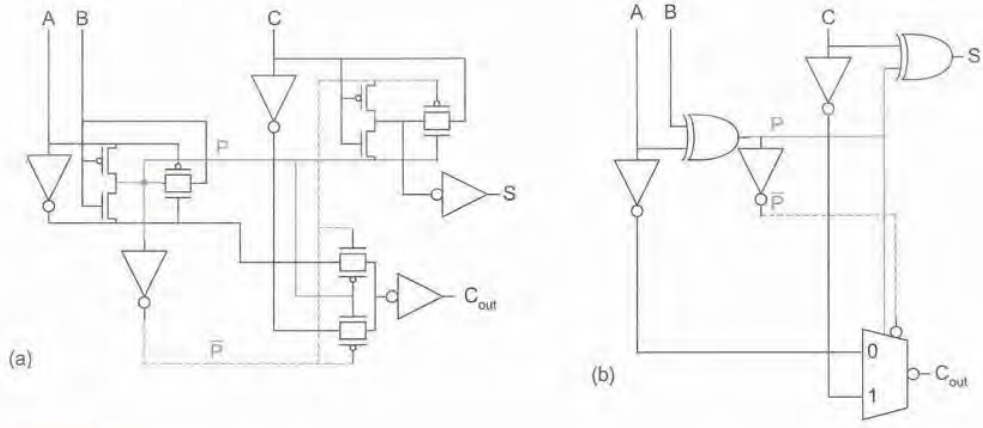


FIG 10.7 Zhuang full adder

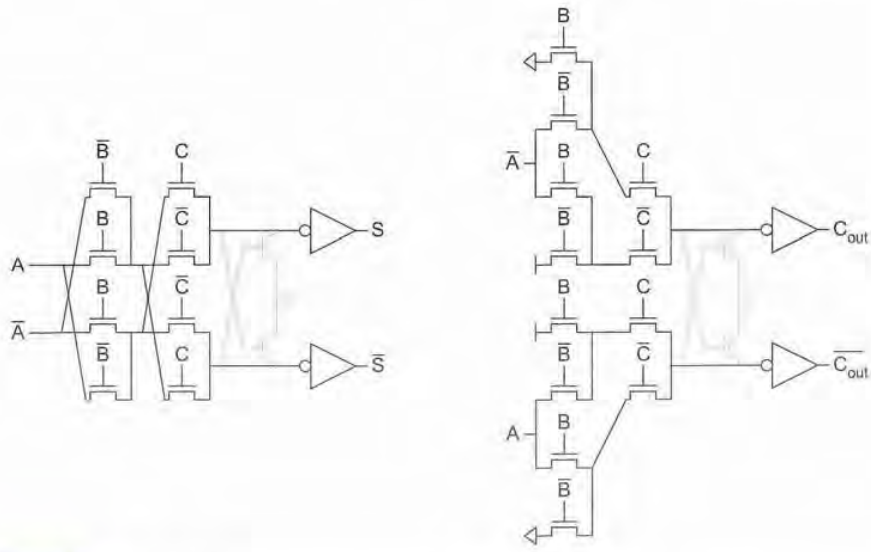


FIG 10.8 CPL full adder

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.

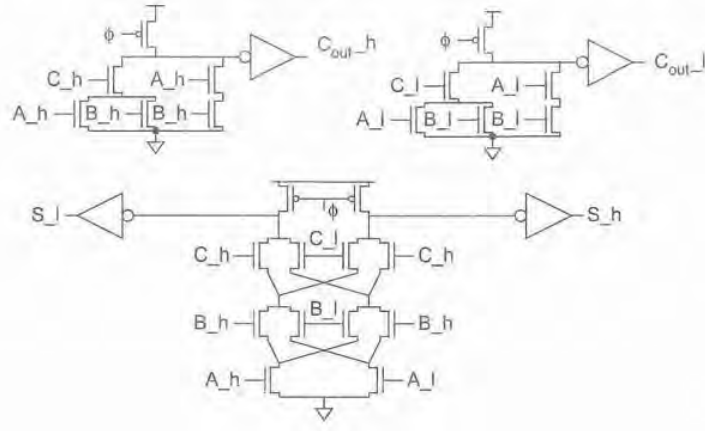


FIG 10.9 Dual-rail domino full adder

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.

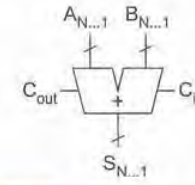


FIG 10.10 Carry-propagate adder

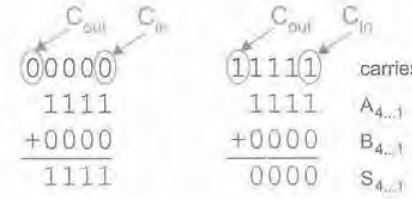


FIG 10.11 Example of carry propagation

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 $t_{C \to Cout}$ 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.

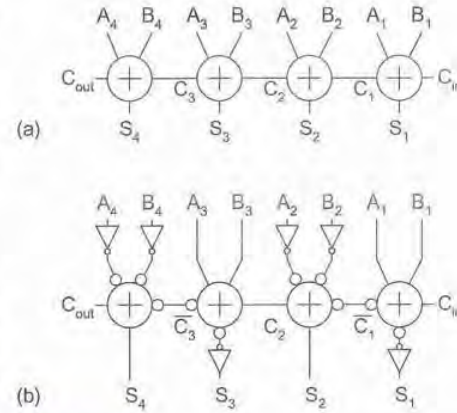


FIG 10.12 4-bit carry-ripple adder

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).

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

with the base case

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

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 $G_{i:j} = G_{i:k} + \overline{K_{i:k}} \bullet G_{k-1:j}$ and $\overline{K_{i:j}} = \overline{K_{i:k}} \bullet \overline{K_{k-1:j}}$. However, P_i is still required in EQ (10.7) to compute the final sum.

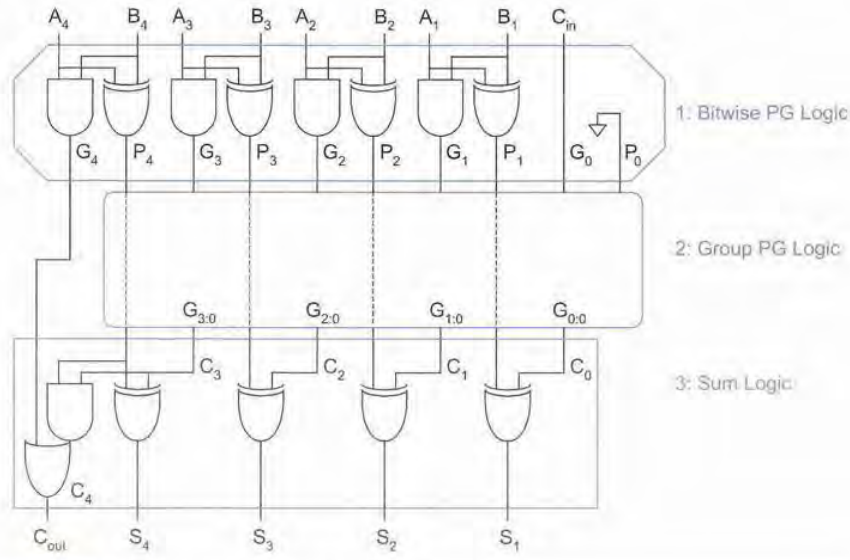


FIG 10.13 Addition with generate and propagate logic

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.

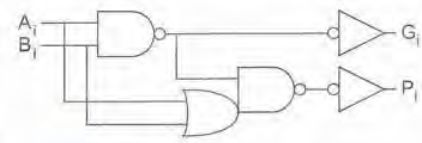


FIG 10.14 Shared bitwise PG logic

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

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.

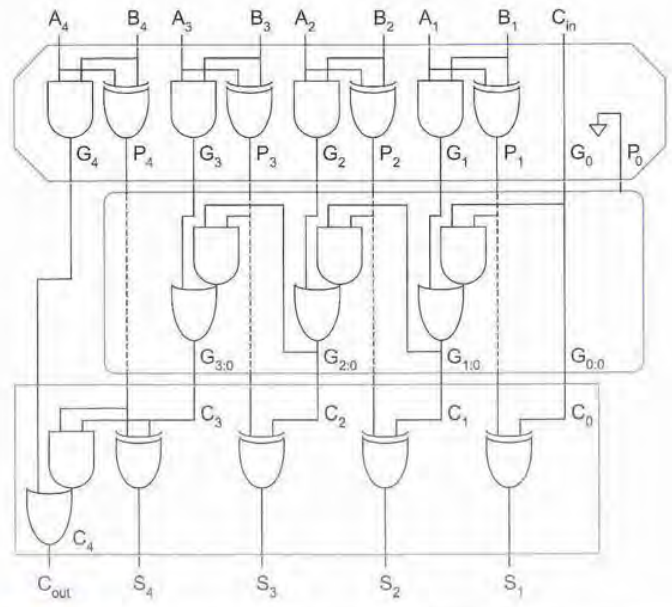


FIG 10.15 4-bit carry-ripple adder using PG logic

$$\begin{aligned} C_i &= A_iB_i + (A_i + B_i)C_{i-1} \\ &= A_iB_i + (A_i \oplus B_i)C_{i-1} \\ &= G_i + P_iC_{i-1} \end{aligned} \tag{10.9}$$

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

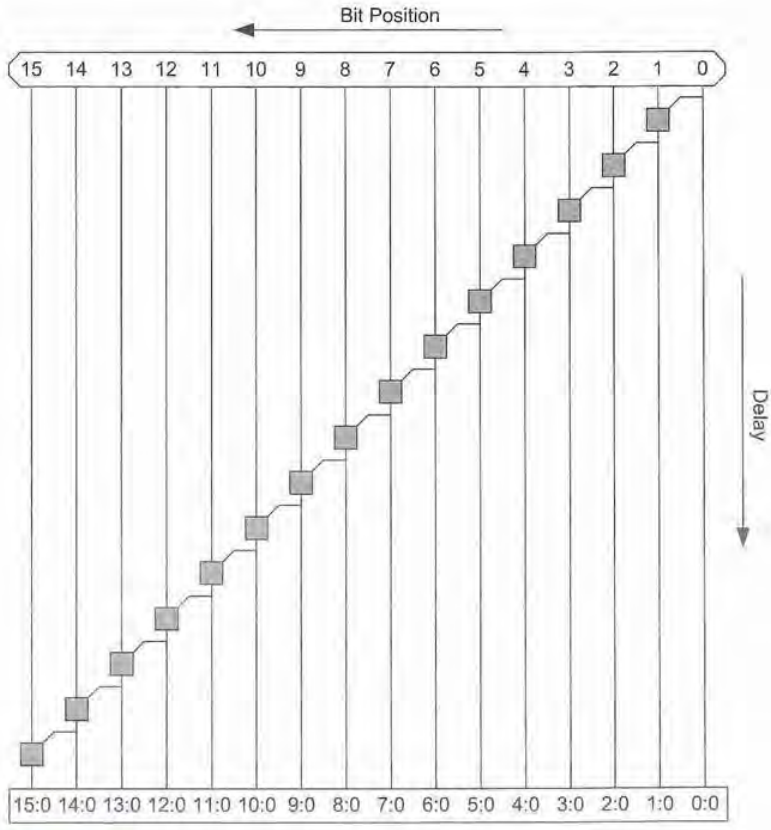


FIG 10.16 Carry-ripple adder group PG network

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:

$$t_{\text{ripple}} = t_{pg} + (N-1)t_{AO} + t_{\text{xor}} \tag{10.12}$$

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.

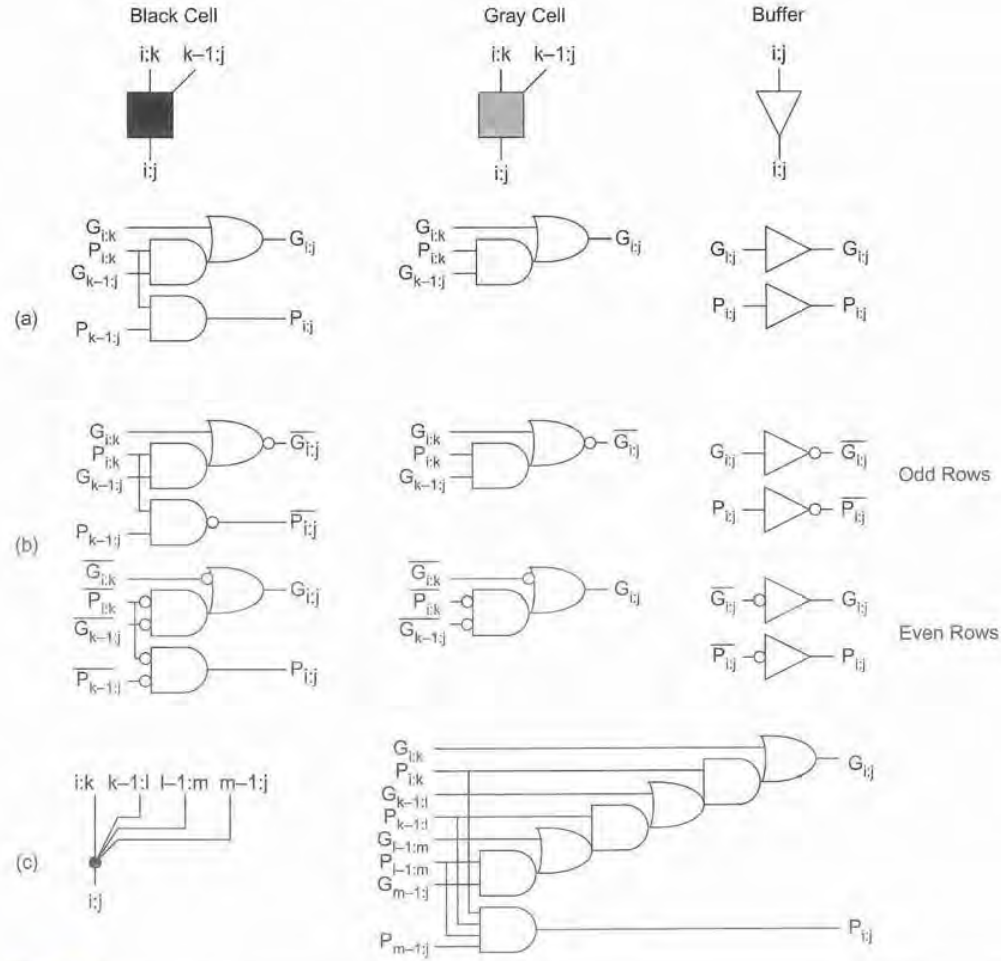


FIG 10.17 Group PG cells

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

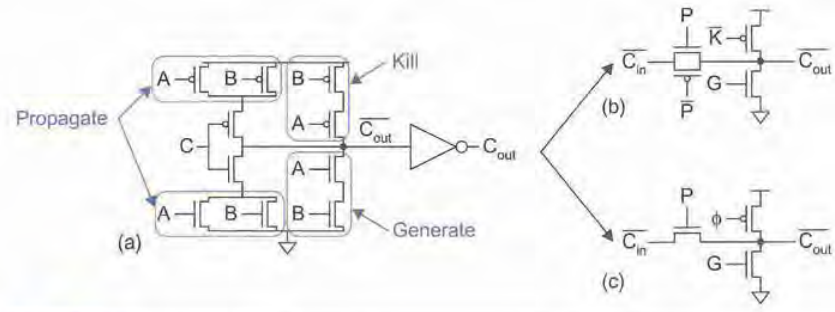


FIG 10.18 Carry chain designs

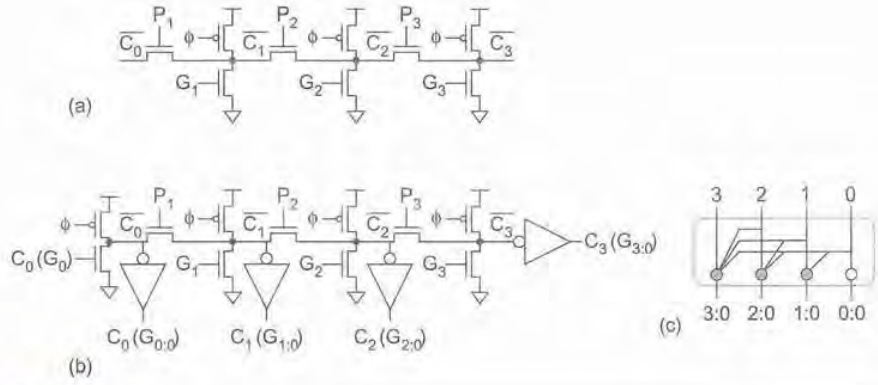


FIG 10.19 Manchester carry chains

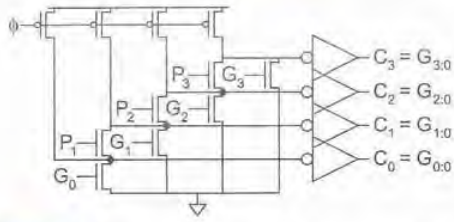


FIG 10.20 Equivalence of Manchester carry chain and multiple-output domino gate

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

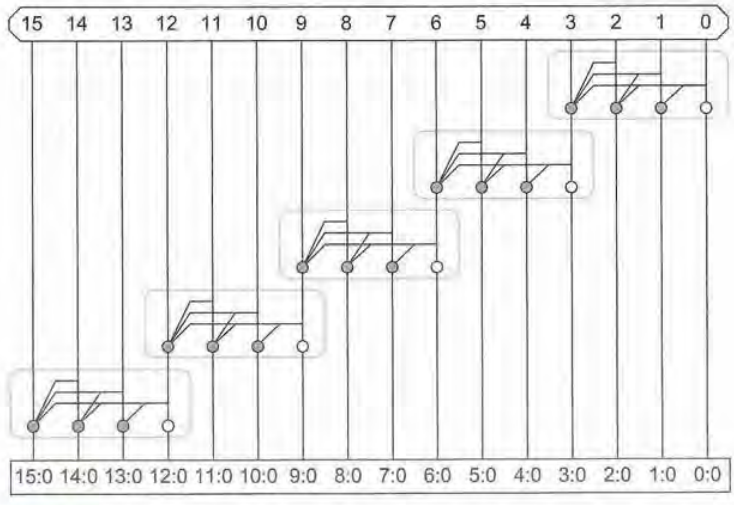


FIG 10.21 Manchester carry chain adder group PG network

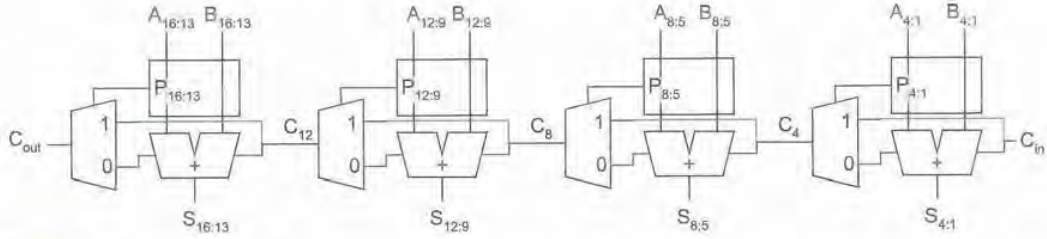


FIG 10.22 Carry-skip adder

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 ($N = n * k$) has a delay of

$$t_{skip} = t_{pg} + 2(n-1) + (k-1)t_{AO} + t_{xor} \qquad \textbf{(10.14)}$$

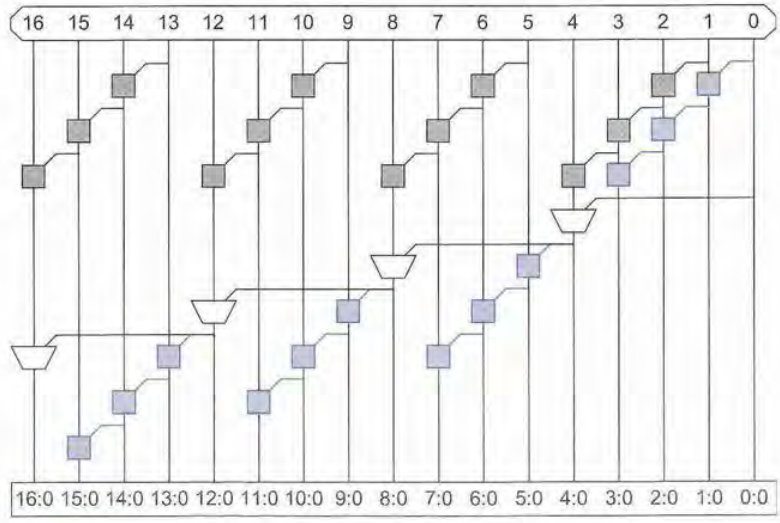


FIG 10.23 Carry-skip adder PG network

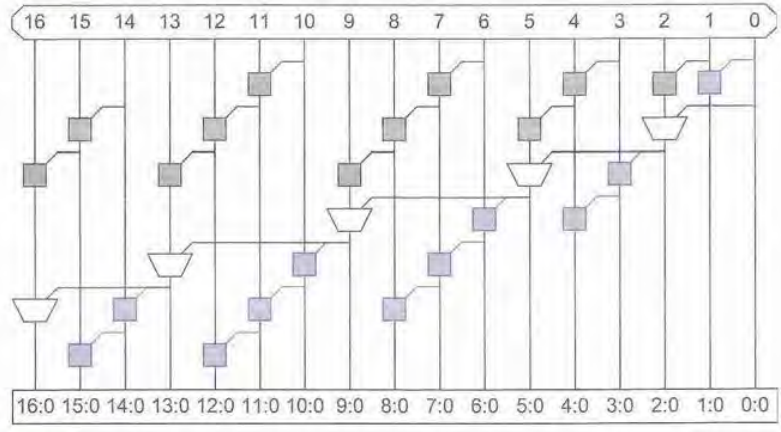


FIG 10.24 Variable group size carry-skip adder PG network

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.

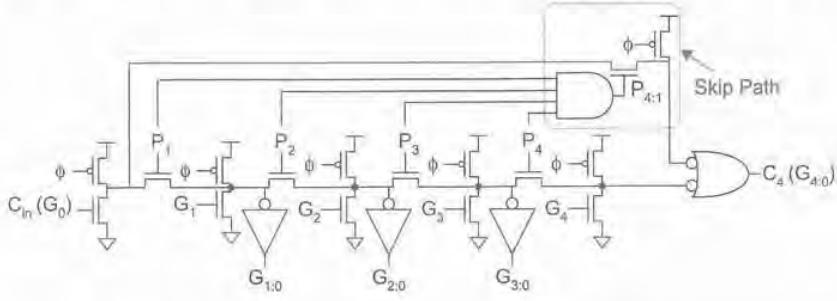


FIG 10.25 Carry-skip adder Manchester stage

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.

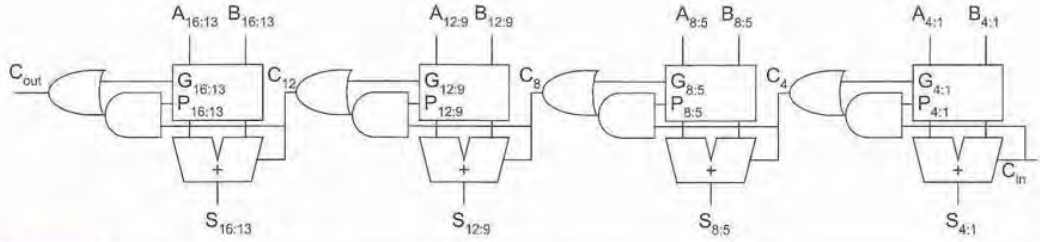


FIG 10.26 Carry-lookahead adder

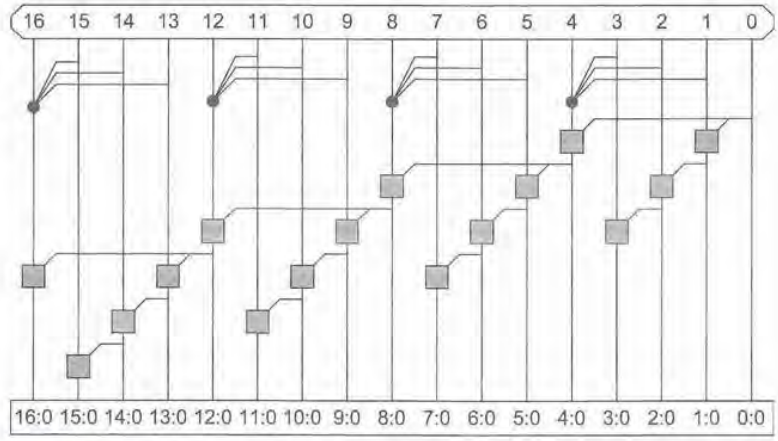


FIG 10.27 Carry-lookahead adder group PG network

In general, a CLA using k groups of n bits each has a delay of

$$t_{\text{cla}} = t_{pg} + t_{pg(n)} + \left[(n-1)+(k-1)\right]t_{AO} + t_{\text{xor}} \qquad \textbf{(10.15)}$$

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.

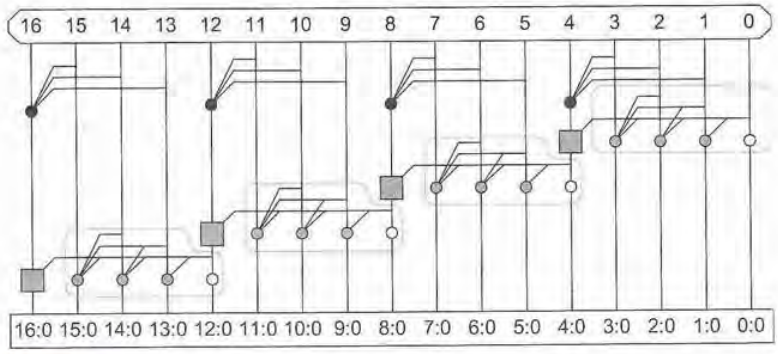


FIG 10.28 Improved CLA group PG network

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

$$t_{\text{select}} = t_{pg} + \left[n + (k-2)\right]t_{AO} + t_{\text{mux}} \qquad \textbf{(10.16)}$$

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

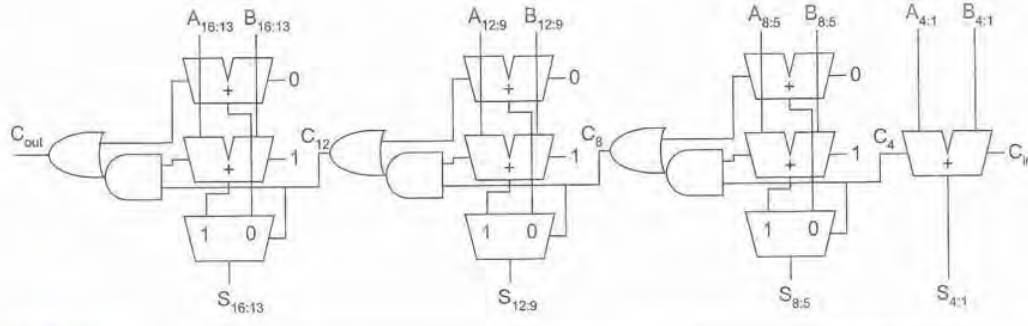


FIG 10.29 Carry-select adder

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.

$$t_{\text{increment}} = t_{pg} + \left[(n-1)+(k-1)\right]t_{AO} + t_{\text{xor}} \tag{10.17}$$

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:

$$t_{\text{increment}} = t_{pg} + t_{pg(n)} + \left[(k-1)\right]t_{AO} + t_{\text{xor}} \tag{10.18}$$

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:

$$t_{\text{increment}} \approx t_{pg} + \sqrt{2N}\,t_{AO} + t_{\text{xor}} \tag{10.19}$$

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

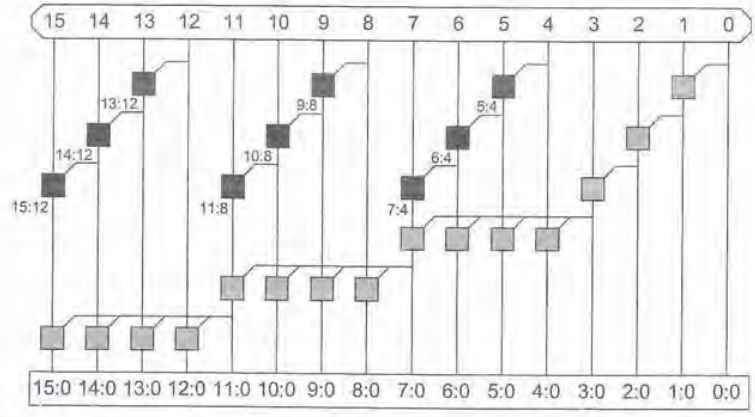


FIG 10.30 Carry-increment adder PG network

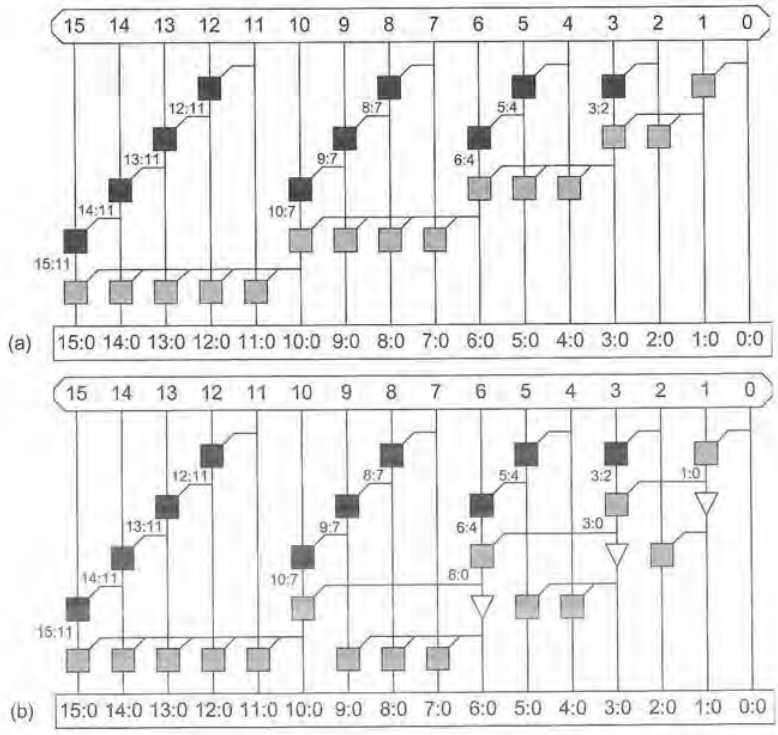


FIG 10.31 Variable-length carry-increment adder

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

Block Width	Block Carry In
1	0
	1
2	0
	1
4	0
	1
8	0
	1
16	0

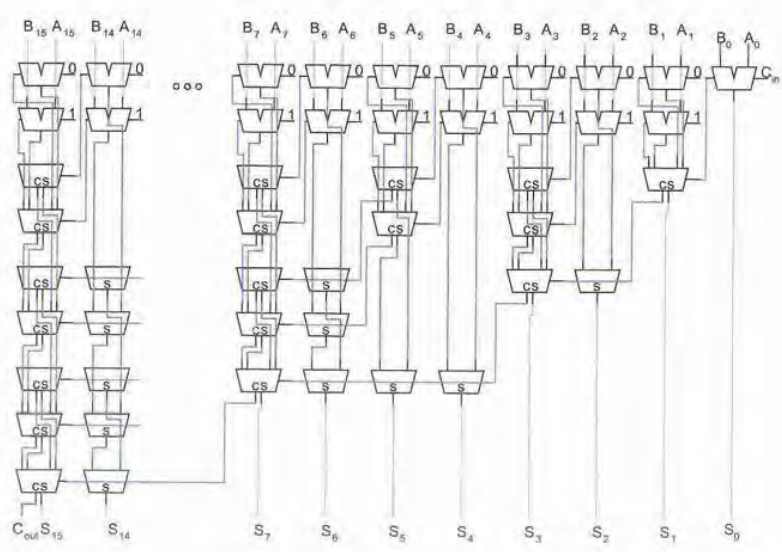


FIG 10.32 Conditional-sum adder

Block Width	Block Carry In		a	1	0	0	1	1	1	0	1	1	0	1	1	0	1	1	0	1	0
			b	0	0	0	1	1	1	0	0	1	1	0	1	1	0	1	0	1	1	0	0
			Block Sum and Carry Out																			C_{in}
			16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1				
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

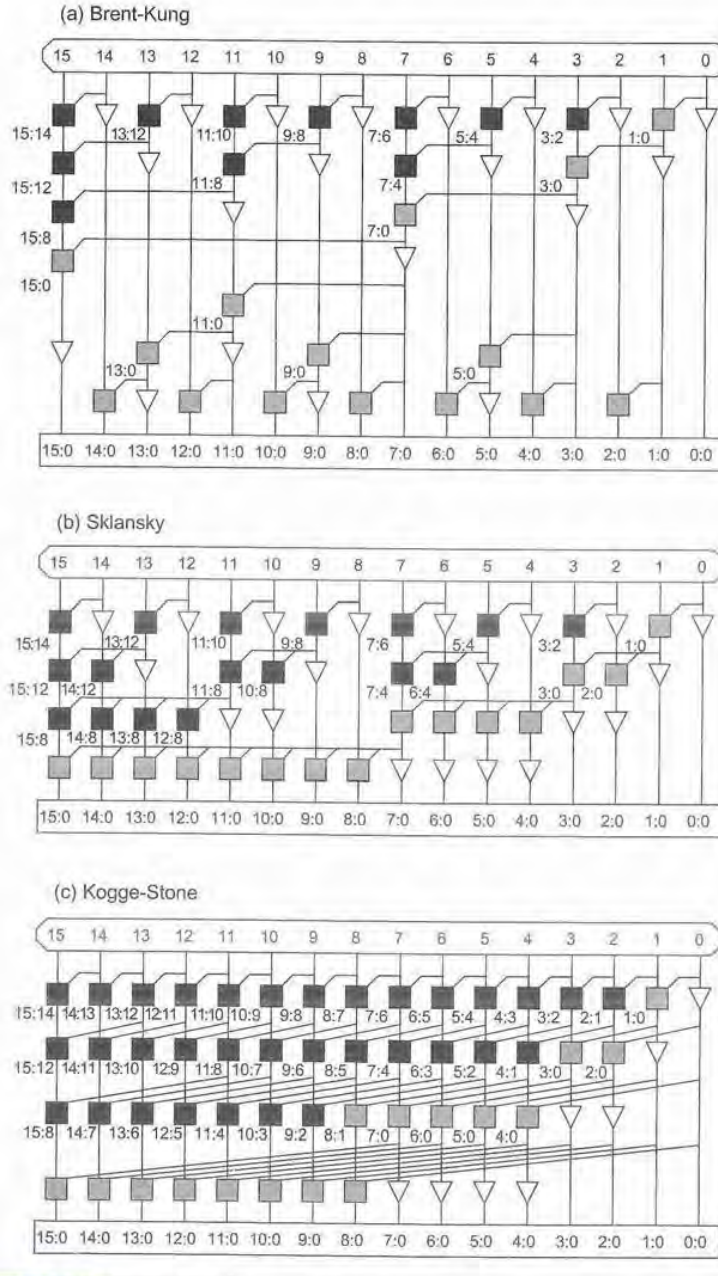
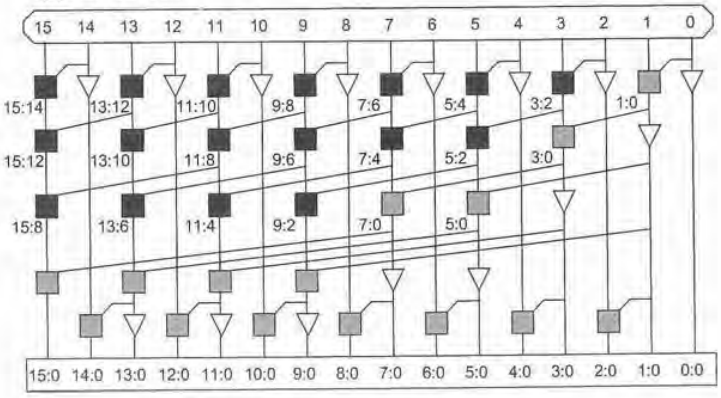
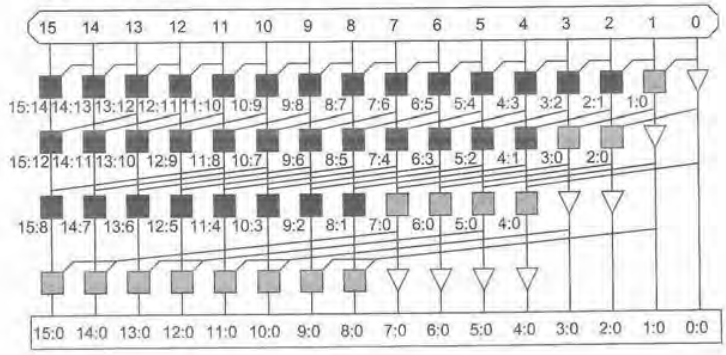


FIG 10.34 Tree adder PG networks

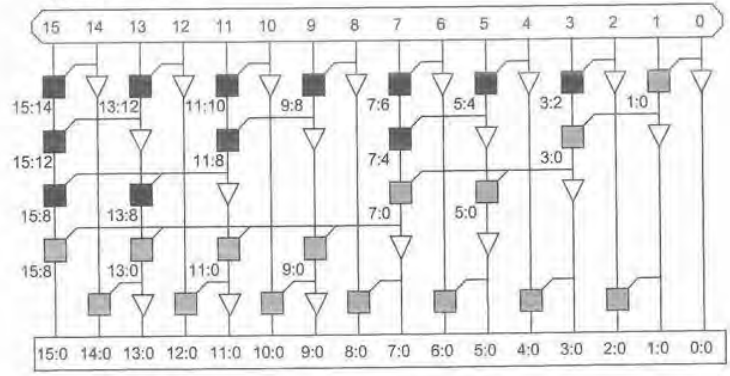
(d) Han-Carlson



(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:

$$t_{tree} \approx t_{pg} + \lceil \log_2 N \rceil t_{AO} + t_{xor} \quad \textbf{(10.20)}$$

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.

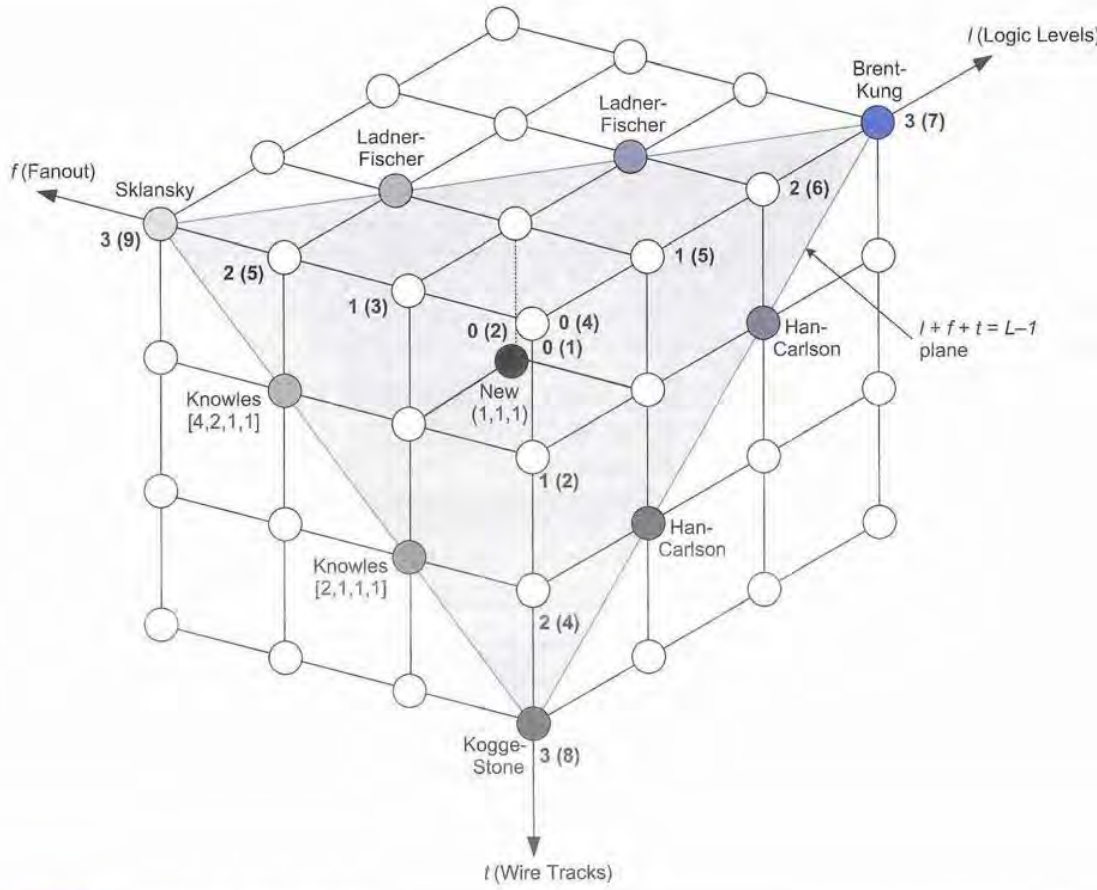


FIG 10.35 Taxonomy of prefix networks

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.

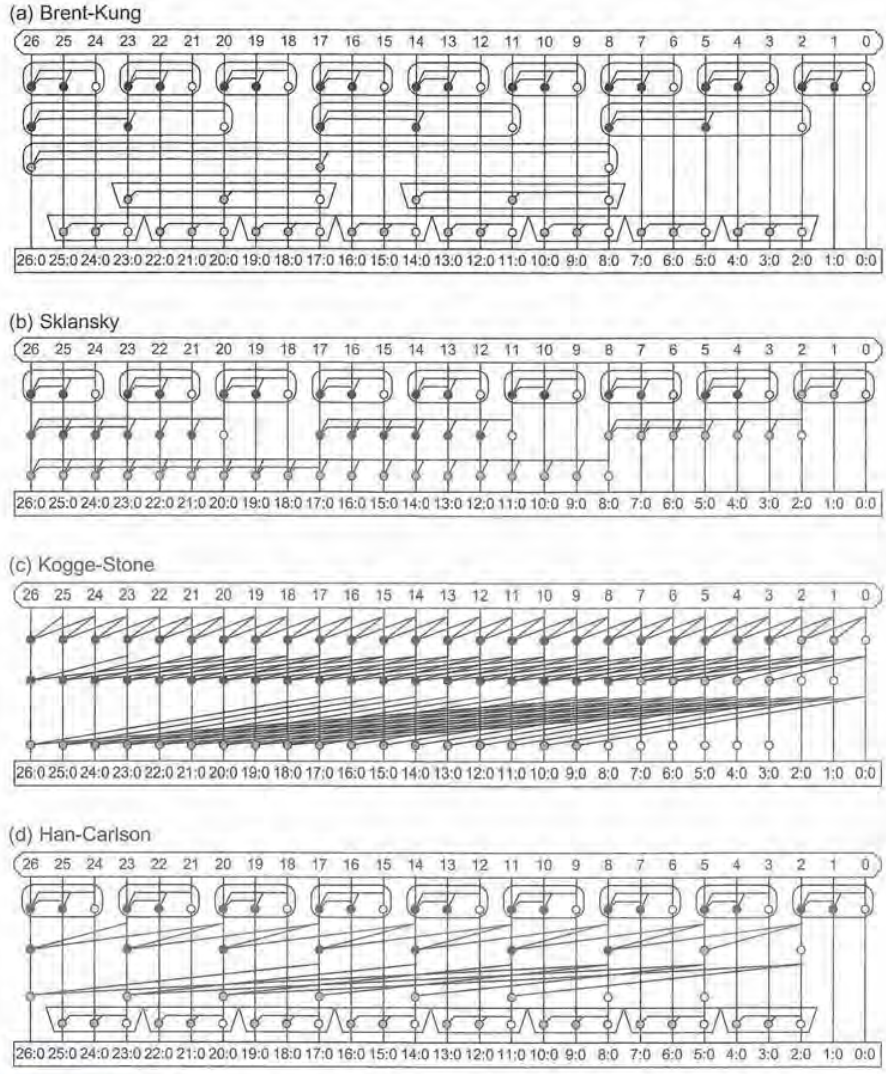


FIG 10.36 Higher-valency tree adders

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-

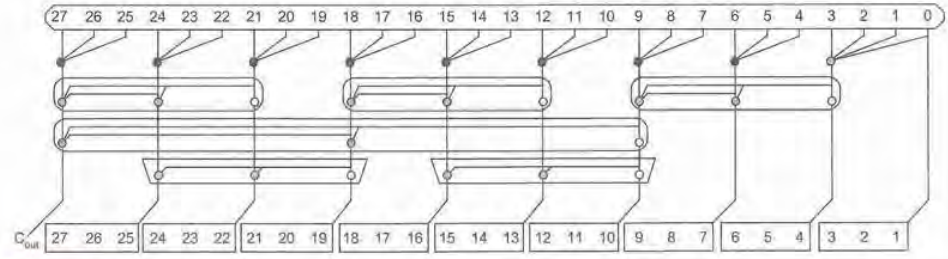


FIG 10.37 Spanning-tree adder

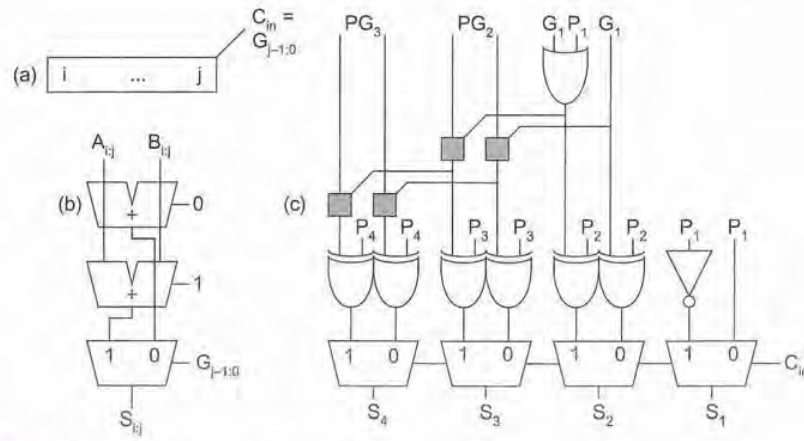


FIG 10.38 Carry-select implementation

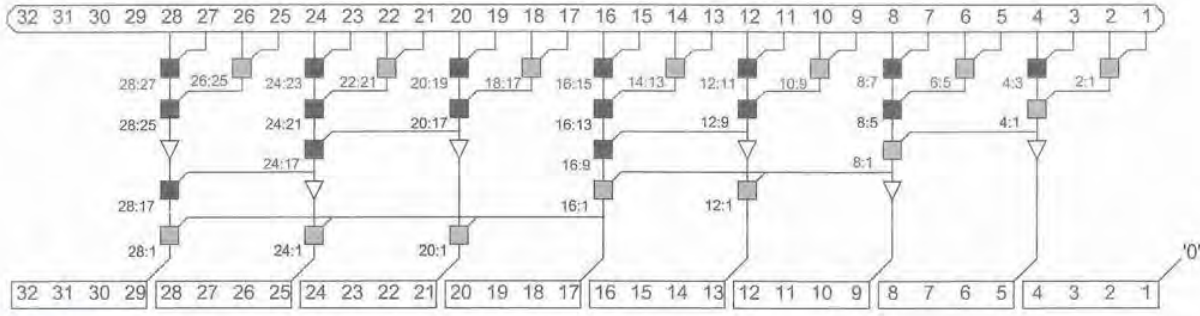


FIG 10.39 Sparse-tree adder

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 $G_{i:0}$ 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.

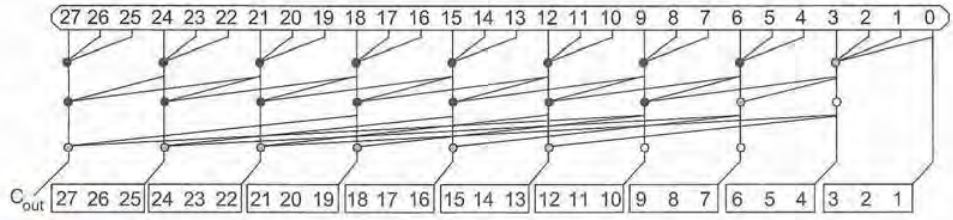


FIG 10.40 Hybrid Kogge-Stone/select adder

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 (_*h* 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:

$$\begin{array}{lll} G_i_h = A_i_h \cdot B_i_h & G_0_h = C_{in}_h & G_{i:j}_h = G_{i:k}_h + K_{i:k}_l \cdot G_{k-1:j}_h \\ K_i_h = A_i_l \cdot B_i_l & K_0_h = C_{in}_l & K_{i:j}_h = K_{i:k}_h + G_{i:k}_l \cdot K_{k-1:j}_h \\ G_i_l = A_i_l + B_i_l & G_0_l = 0 & G_{i:j}_l = G_{i:k}_l \cdot G_{k-1:j}_l \\ K_i_l = A_i_h + B_i_h & K_0_l = 0 & K_{i:j}_l = K_{i:k}_l \cdot K_{k-1:j}_l \end{array} \quad (10.21)$$

$$\begin{aligned} P_i_h &= A_i \oplus B_i = A_i_h \cdot B_i_l + A_i_l \cdot B_i_h = G_i_l \cdot K_i_l \\ P_i_l &= \overline{A_i \oplus B_i} = A_i_h \cdot B_i_h + A_i_l \cdot B_i_l = G_i_h + K_i_h \\ S_i_h &= G_{i-1:0} \oplus P_i = G_{i-1:0}_h \cdot P_i_l + K_{i-1:0}_h \cdot P_i_h \\ S_i_l &= \overline{G_{i-1:0} \oplus P_i} = G_{i-1:0}_h \cdot P_i_h + K_{i-1:0}_h \cdot P_i_l \end{aligned} \quad (10.22)$$

Observe that the group generate _*h* and _*l* and kill _*h* 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 $G_{i-1:0}_h$ and $K_{i-1:0}_h$ 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:

$$
\begin{array}{llll}
G_i = A_i_h \cdot B_i_h & G_0 = C_{\text{in}}_h & G_{i:j} = G_{i:k} + P_{i:k} \cdot G_{k-1:j} \\
P_i = A_i_h \cdot B_i_l + A_i_l \cdot B_i_h = A \oplus B & P_0 = 0 & P_{i:j} = P_{i:k} \cdot P_{k-1:j} \\
K_i = A_i_l \cdot B_i_l & K_0 = C_{\text{in}}_l & K_{i:j} = K_{i:k} + P_{i:k} \cdot K_{k-1:j}
\end{array}
\tag{10.23}
$$

$$
\begin{aligned}
P_i' &= G_i + K_i \\
S_i_h &= G_{i-1:0} \cdot P_i' + K_{i-1:0} \cdot P_i \\
S_i_l &= K_{i-1:0} \cdot P_i' + G_{i-1:0}_h \cdot P_i
\end{aligned}
\tag{10.24}
$$

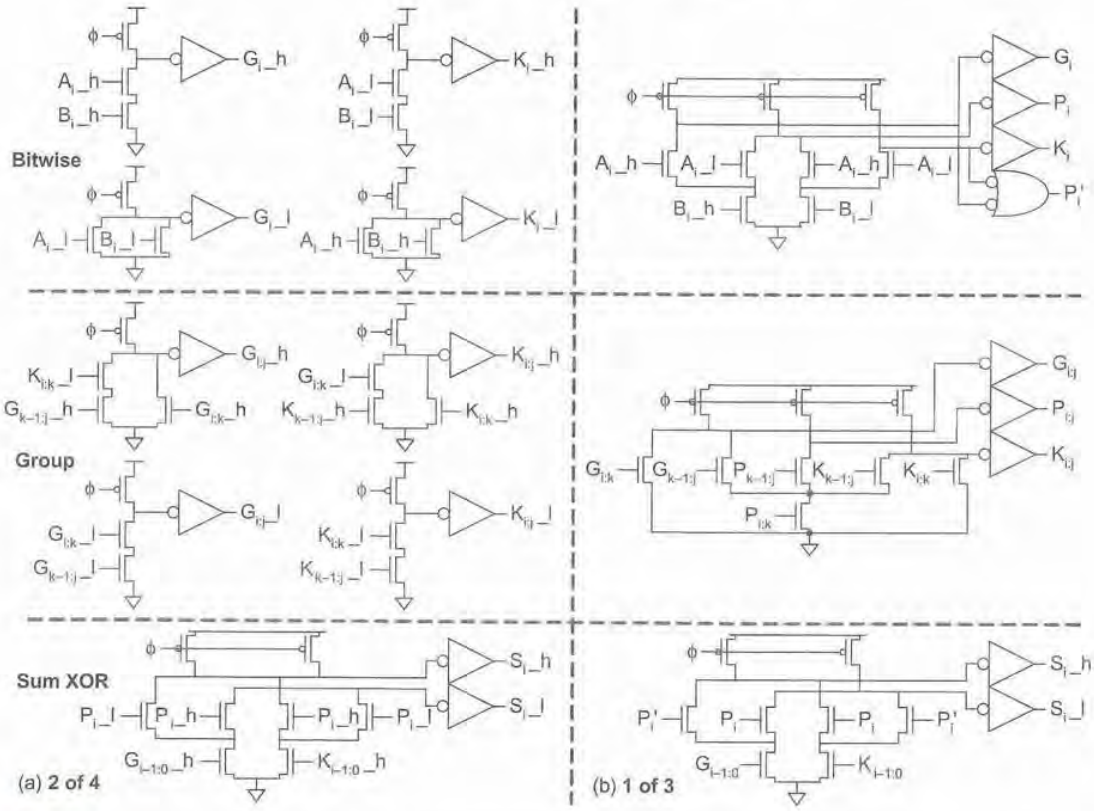


FIG 10.41 Domino adder circuit components

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].

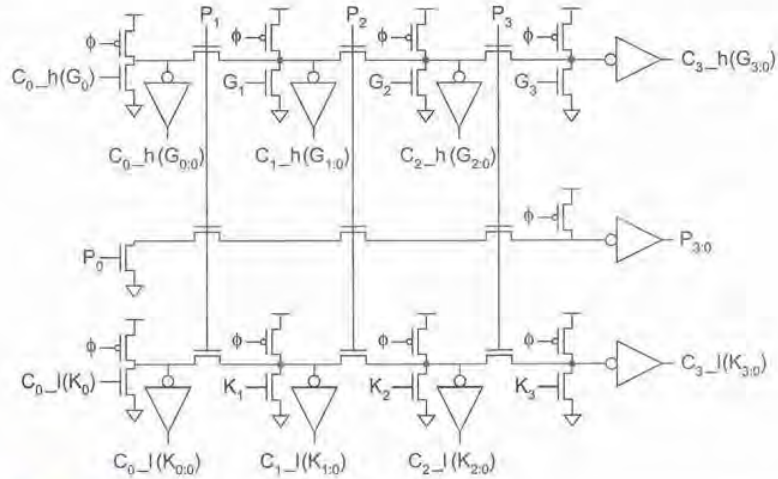


FIG 10.42 1-of-3 hot domino Manchester carry chain

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



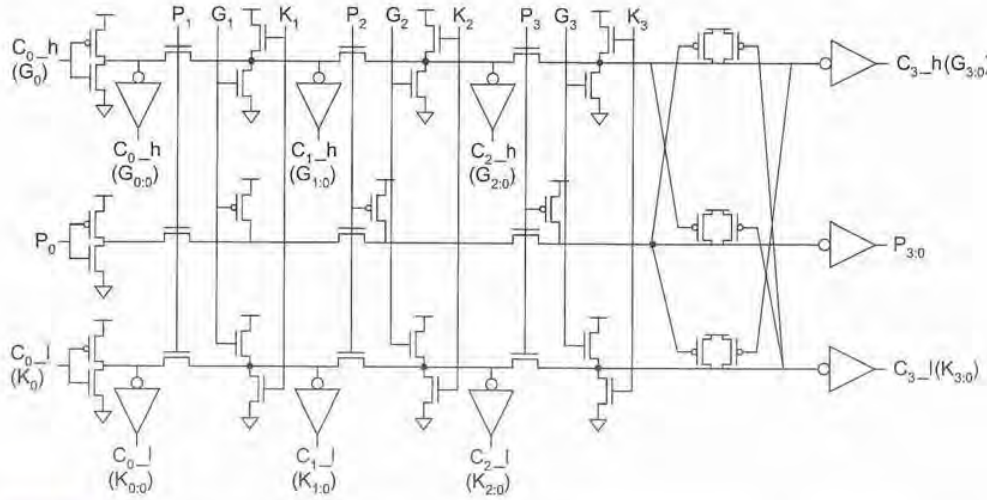


FIG 10.43 1-of-3 hot CPL Manchester carry chain

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

$$G_{3:0} = G_3 + P_3\left(G_2 + P_2\left(G_1 + P_1 G_0\right)\right) \tag{10.25}$$

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.

$$H_{3:0} = G_3 + \left(G_2 + P_2\left(G_1 + P_1 G_0\right)\right) = G_3 + G_{2:0} = G_{3:0} + G_{2:0} \tag{10.26}$$

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.

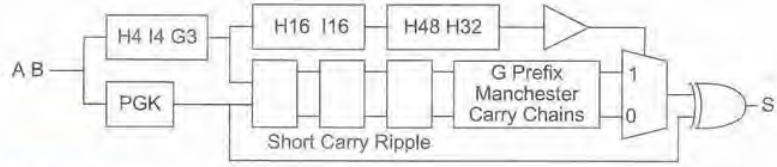


FIG 10.44 Naffziger adder

The Naffziger adder uses dual-rail domino throughout. The equations below describe only the true (_*h*) logic. The complementary (_*l*) 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.

$$
\begin{aligned}
G_{4k+3:4k+1} &= G_{4k+3} + P_{4k+3}\left(G_{4k+2} + P_{4k+2}G_{4k+1}\right) \\
&= A_{4k+3}B_{4k+3} + \left(A_{4k+3} + B_{4k+3}\right)\left(A_{4k+2}B_{4k+2} + \left(A_{4k+2} + B_{4k+2}\right)A_{4k+1}B_{4k+1}\right) \\
H_{4k+4:4k+1} &= G_{4k+4} + \left(G_{4k+3} + P_{4k+3}\left(G_{4k+2} + P_{4k+2}G_{4k+1}\right)\right) \\
&= A_{4k+4}B_{4k+4} + G_{4k+3:4k+1} \\
I_{4k+4:4k+1} &= \left(A_{4k+3} + B_{4k+3}\right)\left(A_{4k+2} + B_{4k+2}\right)\left(A_{4k+1} + B_{4k+1}\right)\left(A_{4k} + B_{4k}\right)
\end{aligned}
\tag{10.30}
$$

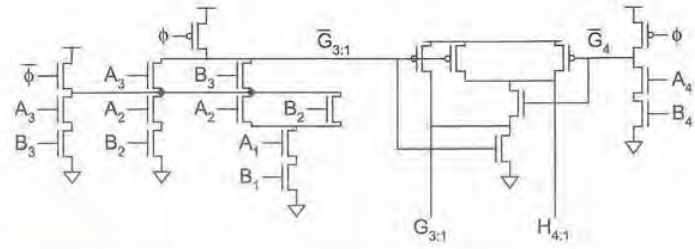


FIG 10.45 GH gate

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 ($i = 1 \ldots 64$) 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), $k = 1 \ldots 3$).

$$
H_{16:1} = H_{16:13} + I_{16:13}\left(H_{12:9} + I_{12:9}\left(H_{8:5} + I_{8:5}\left(H_{4:1} + I_{4:1}\right)\right)\right) \tag{10.31}
$$

$$
\begin{aligned}
H_{16k+16:16k+1} &= H_{16k+16:16k+13} + \\
&\quad I_{16k+16:16k+13}\left(H_{16k+12:16k+9} + I_{16k+12:16k+9}\left(H_{16k+8:16k+5} + I_{16k+8:16k+5}H_{16k+4:16k+1}\right)\right) \\
I_{16k+16:16k+1} &= I_{16k+16:16k+13}I_{16k+12:16k+9}I_{16k+8:16k+5}I_{16k+4:16k+1}
\end{aligned}
\tag{10.32}
$$

Now we can find the pseudo-carry signals into each 16-bit block using EQ (10.33) (H48 H32 block).

$$
\begin{aligned}
H_{0:1} &= C_{\text{in}} \\
H_{16:1} &= H_{16:1} \\
H_{32:1} &= H_{32:17} + I_{32:17}H_{16:1} \\
H_{48:1} &= H_{48:33} + I_{48:33}\left(H_{32:17} + I_{32:17}H_{16:1}\right)
\end{aligned}
\tag{10.33}
$$

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 $G^{0}_{16k+1:0}$ 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 $k = 0\ldots3$ and bits $i = 1\ldots16$ 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.

$$S_{16k+i} = P_{16k+i} \oplus \left(\overline{H}_{16k:1} G^0_{16k+i-1:0} + H_{16k:1} G^1_{16k+i-1:0}\right) \tag{10.36}$$

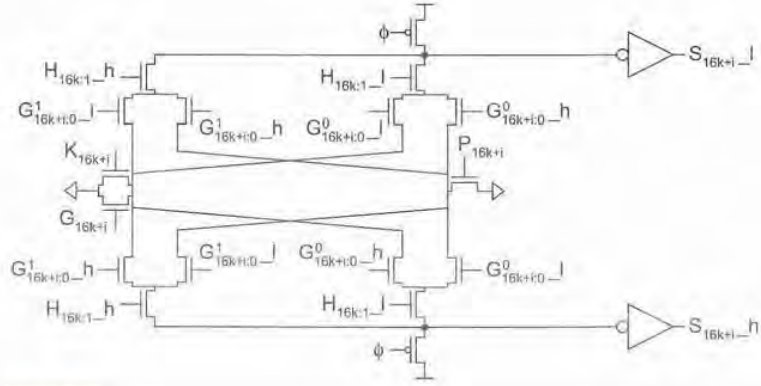


FIG 10.46 Dual-rail sum gate

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

$$A - B = A + \overline{B} + 1 \tag{10.37}$$

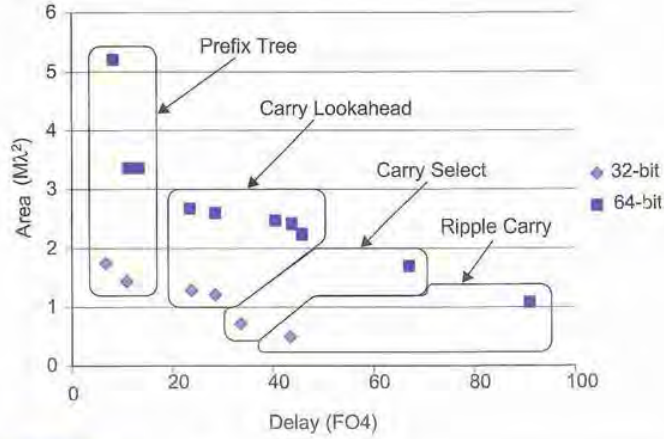


FIG 10.47 Area vs. delay of synthesized adders

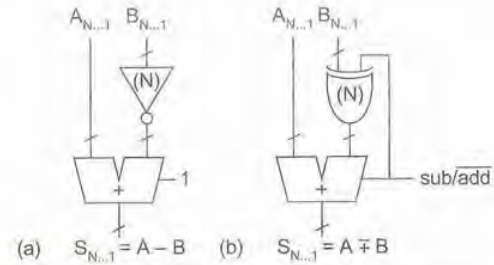


FIG 10.48 Subtracters

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

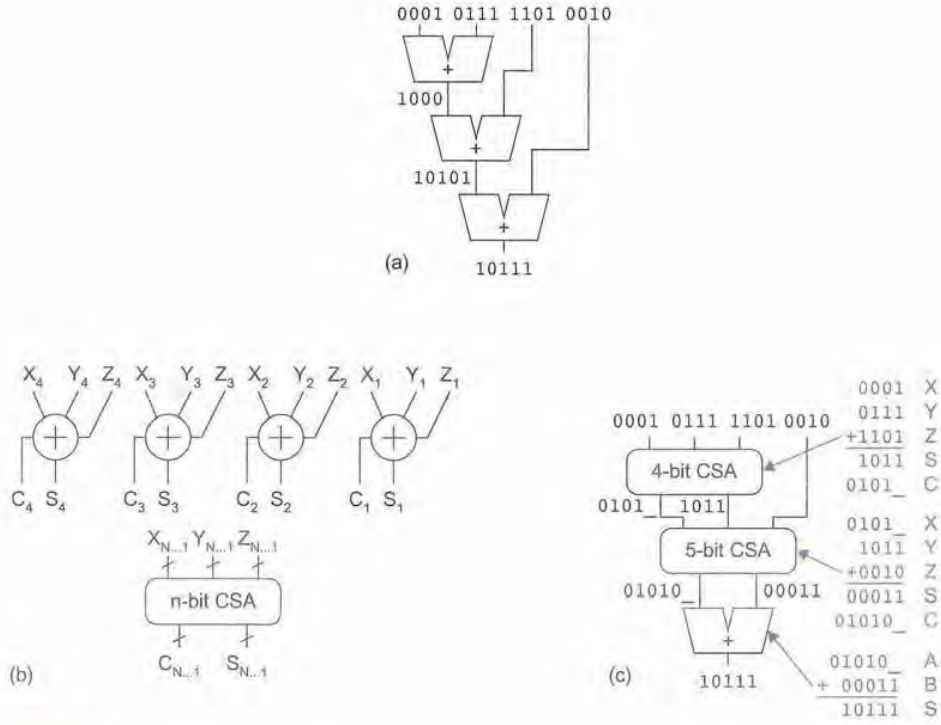


FIG 10.49 Multiple-input adders

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:

$$G_{\text{and}}(N) = \left(\frac{4}{3}\right)^{\log_2 N} = N^{\log_2 \frac{4}{3}} = N^{0.415} \tag{10.38}$$

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.

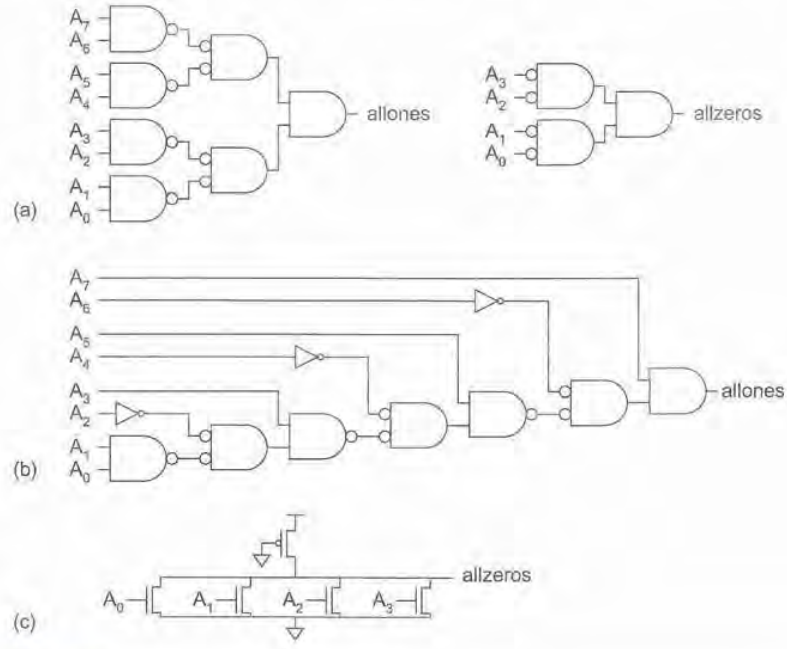


FIG 10.50 One/zero detectors

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.

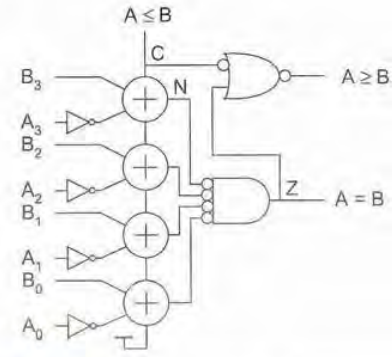


FIG 10.51 Unsigned magnitude comparator

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."

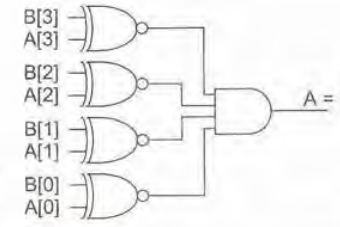


FIG 10.52 Equality comparator



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.

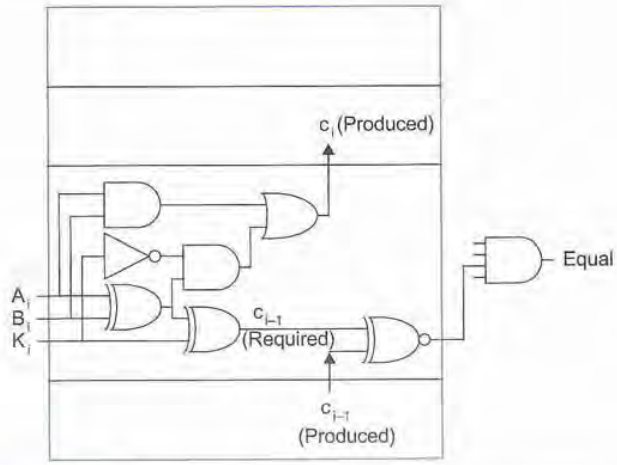


FIG 10.53 A + B = K comparator

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.

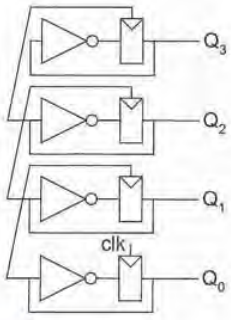


FIG 10.54 Asynchronous ripple-carry counter

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.

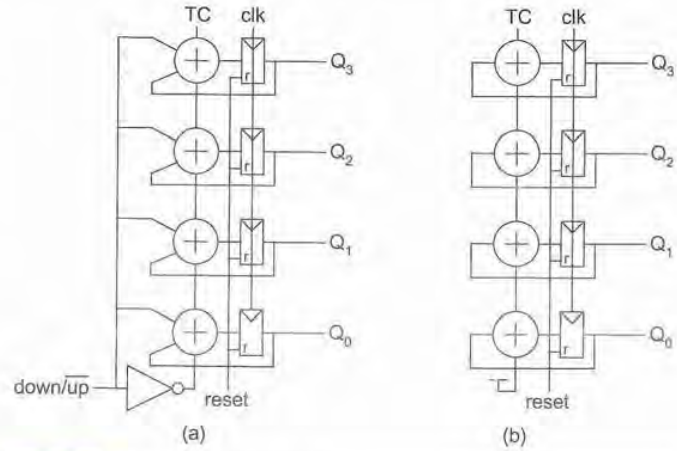


FIG 10.55 Synchronous counters

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.

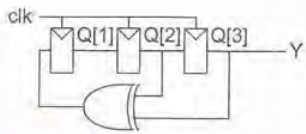


FIG 10.56 3-bit LFSR

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	$1 + x^1 + x^6 + x^7 + x^8$
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.

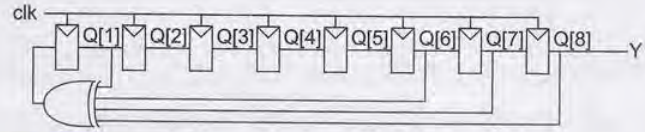


FIG 10.57 8-bit LFSR

10.6 Boolean Logical Operations

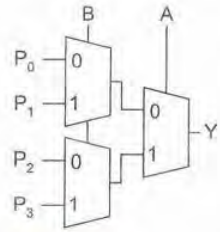


FIG 10.58 Boolean logical unit

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.

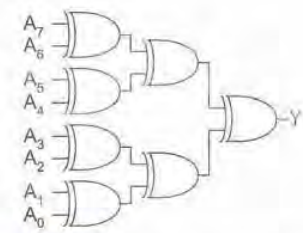


FIG 10.59 8-bit parity generator

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:

$$C_0 = D_3 \oplus D_1 \oplus D_0$$
$$C_1 = D_3 \oplus D_2 \oplus D_0 \qquad (10.43)$$
$$C_2 = D_3 \oplus D_2 \oplus D_1$$

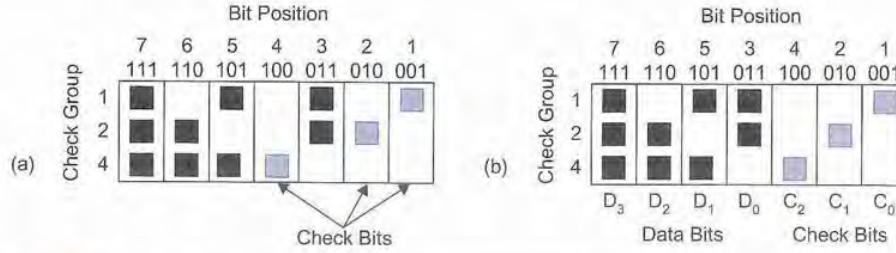


FIG 10.60 Parity-check matrix

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.

$$\begin{array}{ll} \text{Binary-}>\text{Gray} & \text{Gray-}>\text{Binary} \\ G_{N-1} = B_{N-1} & B_{N-1} = G_{N-1} \\ G_i = B_{i+1} \oplus B_i & B_i = B_{i+1} \oplus G_i \quad N-1 > i \geq 0 \end{array} \tag{10.44}$$

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.

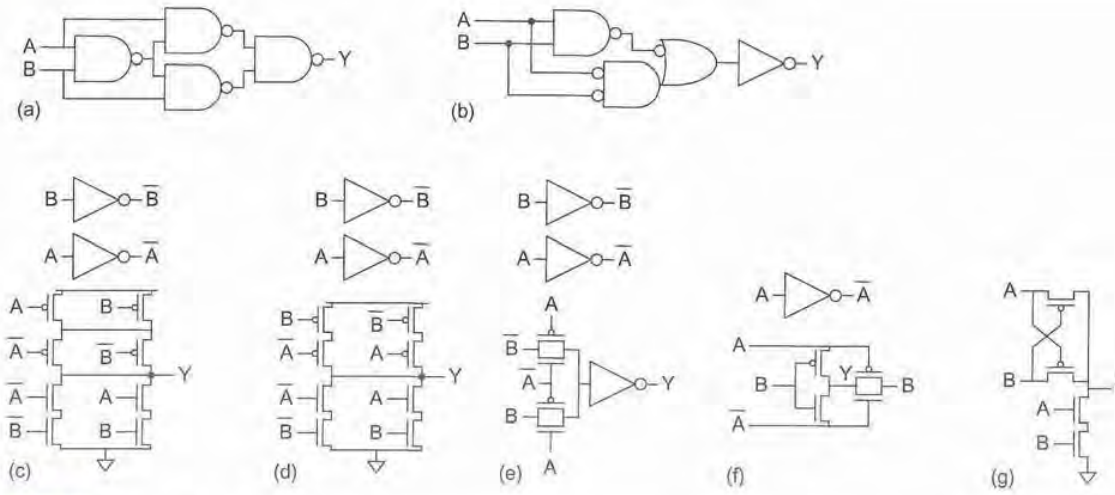


FIG 10.61 Static 2-input XOR designs

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).

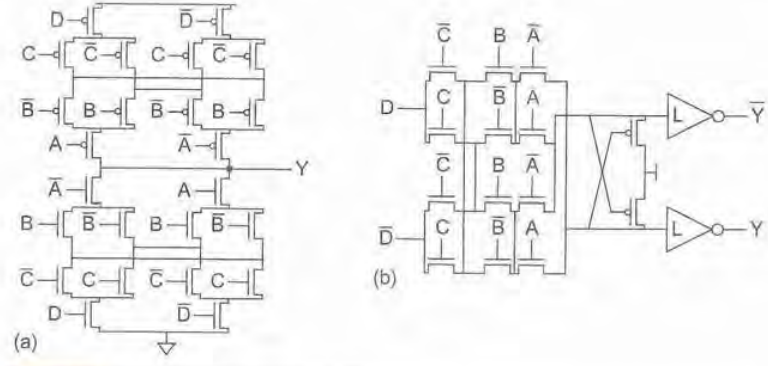


FIG 10.62 4-input XOR designs

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.

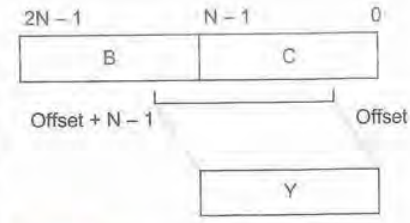


FIG 10.63 Funnel shifter function

Table 10.10 Funnel shifter operation

Shift Type	B	C	Offset
Logical Right	0…0	$A_{N-1}…A_0$	k
Logical Left	$A_{N-1}…A_0$	0…0	$N–k$
Arithmetic Right	$A_{N-1}…A_{N-1}$ (sign extension)	$A_{N-1}…A_0$	k
Arithmetic Left	$A_{N-1}…A_0$	0	$N–k$
Rotate Right	$A_{N-1}…A_0$	$A_{N-1}…A_0$	k
Rotate Left	$A_{N-1}…A_0$	$A_{N-1}…A_0$	$N–k$

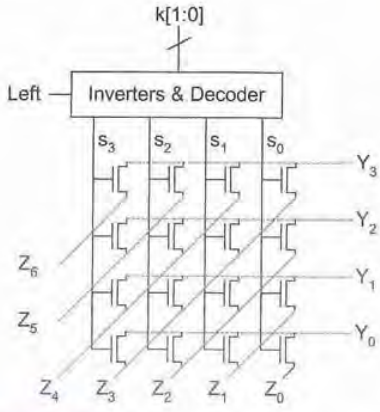


FIG 10.64 Array funnel shifter

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	$A_{N-1}...A_0, A_{N-1}..A_1$	$\overline{k}$

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.

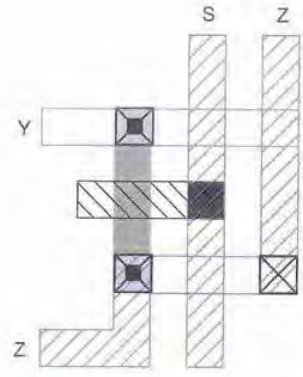


FIG 10.65 Array funnel shifter cell stick diagram

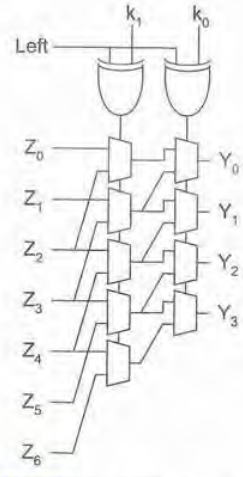


FIG 10.66 Multilevel funnel shifter

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.

$$P = \left(\sum_{j=0}^{M-1} y_j 2^j \right) \left(\sum_{i=0}^{N-1} x_i 2^i \right) = \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} x_i y_j 2^{i+j} \qquad (10.45)$$

						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 Products
			x_3y_5	x_3y_4	x_3y_3	x_3y_2	x_3y_1	x_3y_0				
		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].

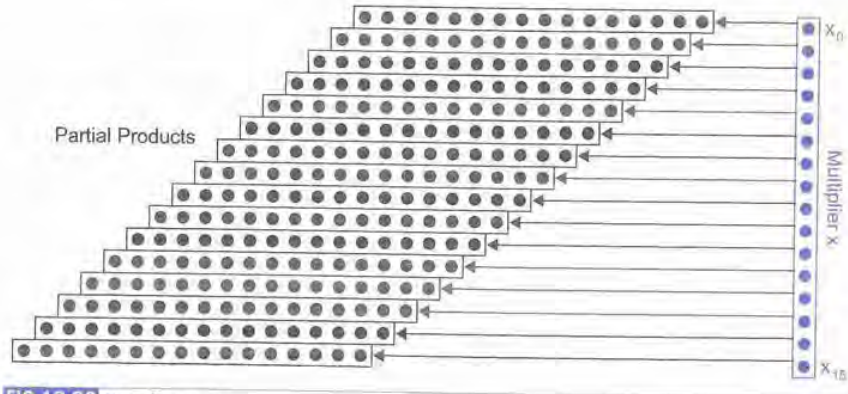


FIG 10.69 Dot diagram

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.

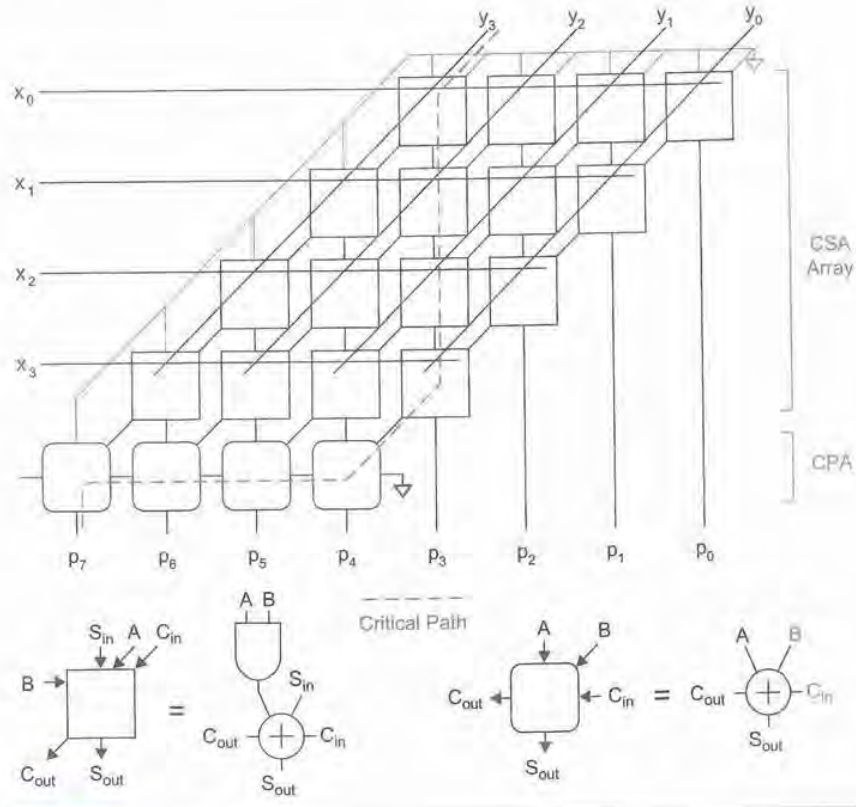


FIG 10.70 Array multiplier

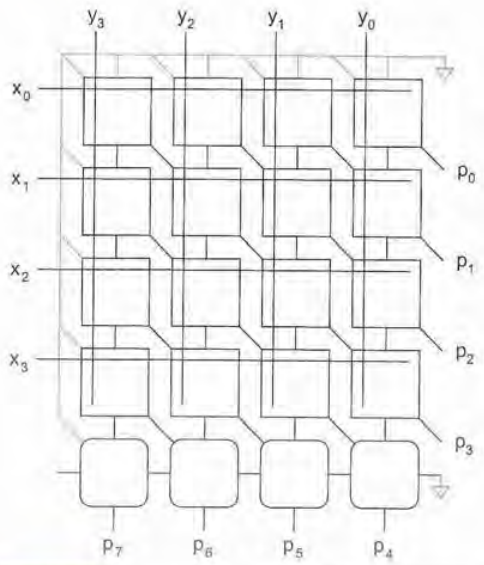


FIG 10.71 Rectangular array multiplier

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:

$$\begin{aligned} P &= \left(-y_{M-1}2^{M-1} + \sum_{j=0}^{M-2} y_j 2^j\right)\left(-x_{n-1}2^{N-1} + \sum_{i=0}^{N-2} x_i 2^i\right) \\ &= \sum_{i=0}^{N-2}\sum_{j=0}^{M-2} x_i y_j 2^{i+j} + x_{N-1}y_{M-1}2^{M+N-2} - \left(\sum_{i=0}^{N-2} x_i y_{M-1} 2^{i+M-1} + \sum_{j=0}^{M-2} x_{N-1} y_j 2^{j+N-1}\right) \end{aligned} \quad (10.46)$$

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.

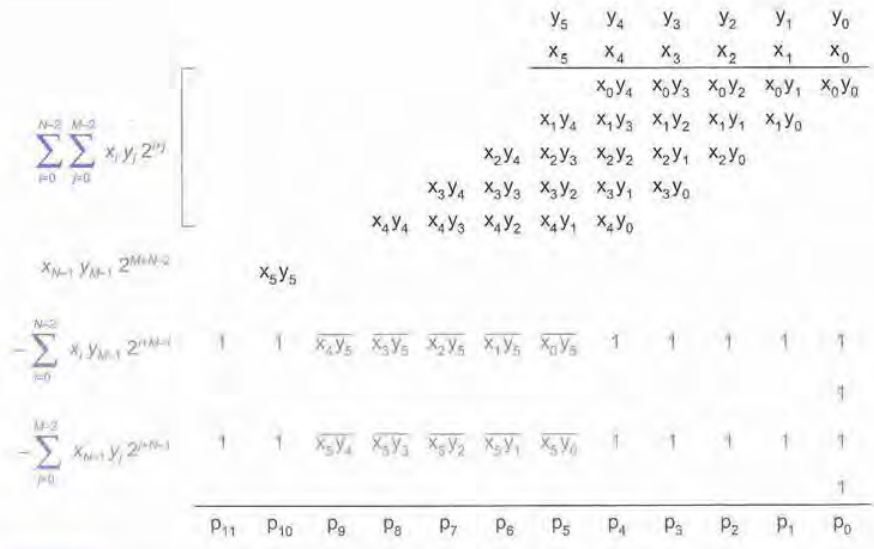


FIG 10.72 Partial products for 2's complement multiplier

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.

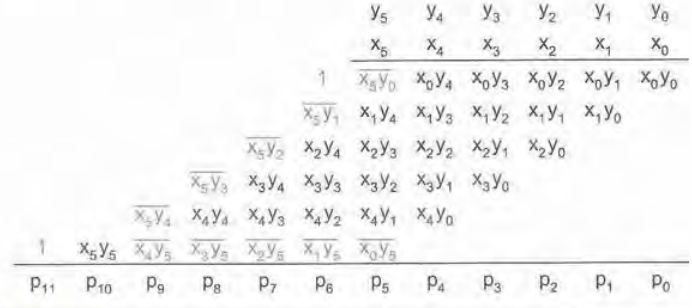


FIG 10.73 Simplified partial products for 2's complement multiplier

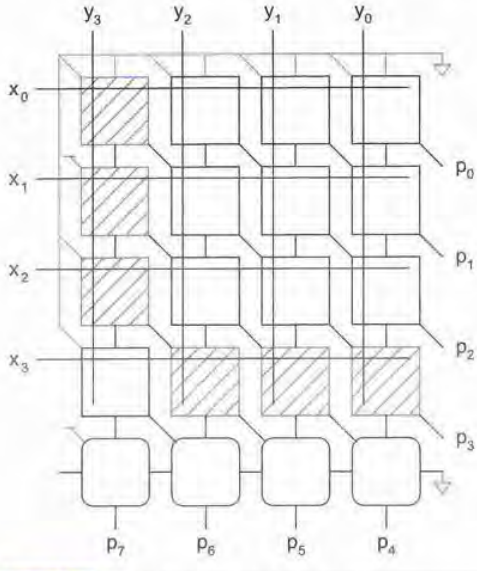


FIG 10.74 Modified Baugh-Wooley 2's complement multiplier

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.

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.

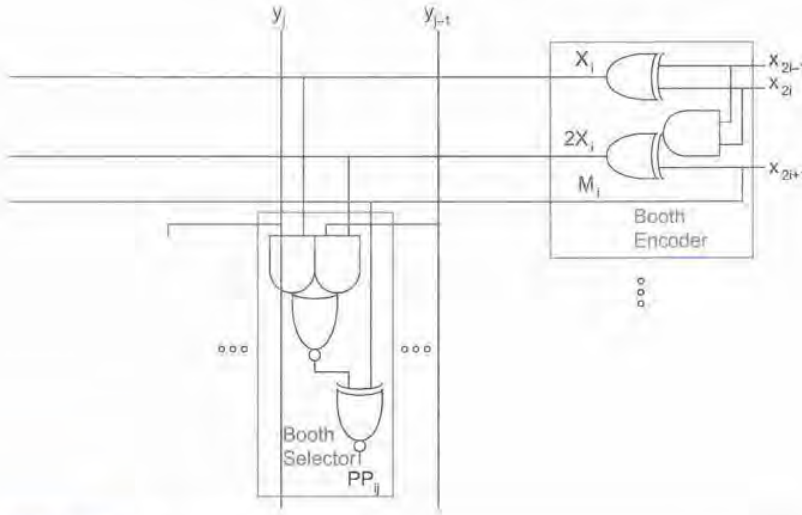


FIG 10.75 Radix-4 Booth encoder and selector

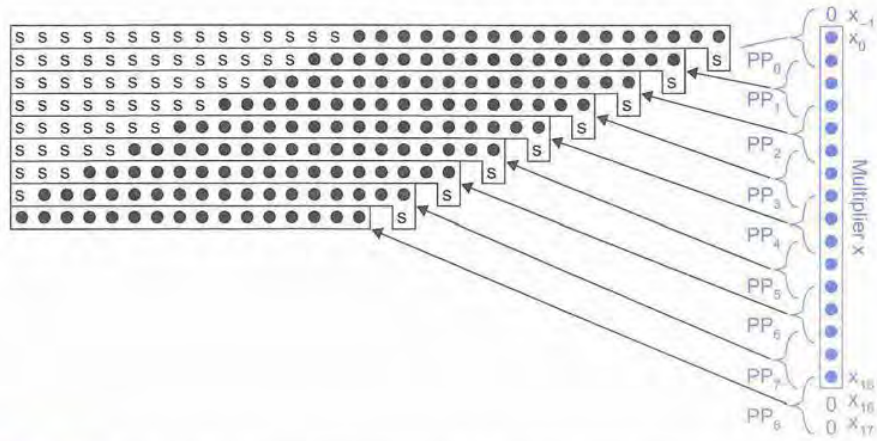


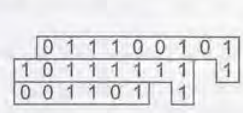
FIG 10.76 Radix-4 Booth-encoded partial products with sign extension

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

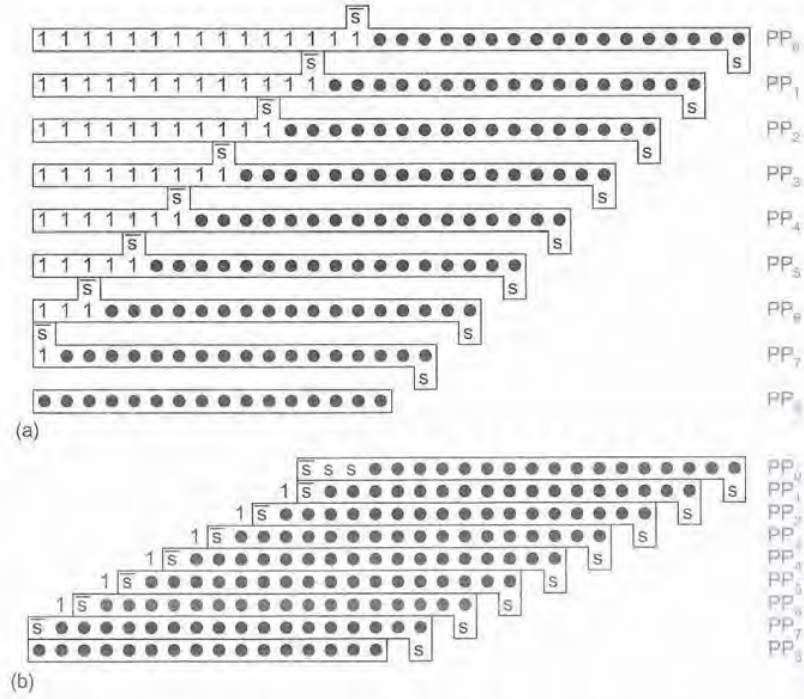


FIG 10.78 Radix-4 Booth-encoded partial products with simplified sign extension

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

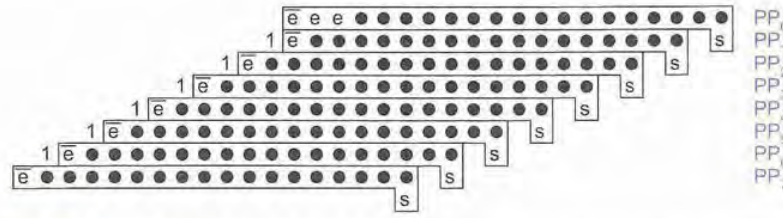


FIG 10.79 Radix-4 Booth-encoded partial products for signed multiplication

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_{i+2}	x_{i+1}	x_i	x_{i-1}	Partial Product
0	0	0	0	0
0	0	0	1	Y
0	0	1	0	Y
0	0	1	1	$2Y$
0	1	0	0	$2Y$
0	1	0	1	$3Y$
0	1	1	0	$3Y$
0	1	1	1	$4Y$
1	0	0	0	$-4Y$
1	0	0	1	$-3Y$
1	0	1	0	$-3Y$
1	0	1	1	$-2Y$
1	1	0	0	$-2Y$
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

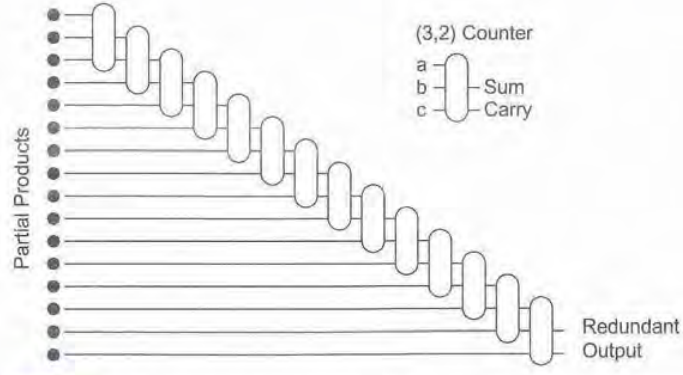


FIG 10.80 Dot diagram for array multiplier

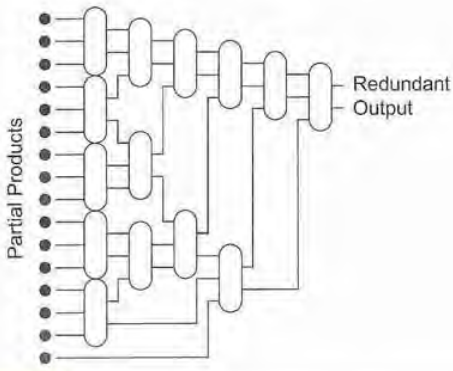


FIG 10.81 Dot diagram for Wallace tree multiplier

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

$$\lceil \log_{3/2}(N/2) \rceil$$

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

$$\lceil \log_2(N/2) \rceil$$

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.

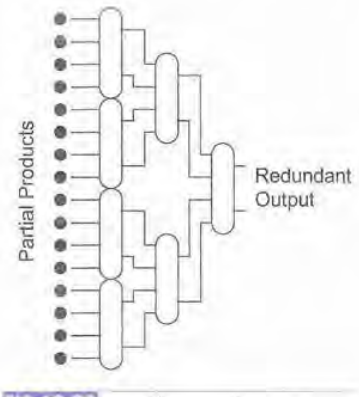


FIG 10.82 Dot diagram for binary-tree multiplier

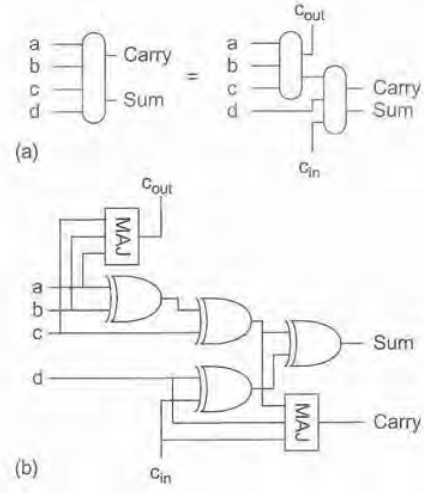


FIG 10.83 [4:2] compressor

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



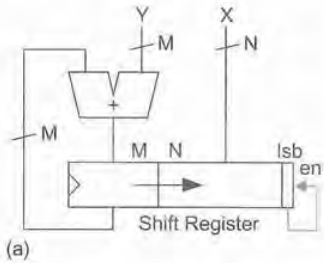
Many algorithms, particularly in digital signal processing, require computing $P = X \cdot Y + Z$. While this can be done with a multiplier and adder, it is much faster to use a *fused multiply-add* unit, which is simply an ordinary multiplier modified to take another partial product Z. The extra partial 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:

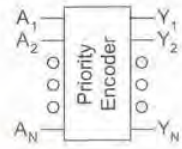


FIG 10.85 Priority encoder

$$
\begin{aligned}
Y_1 &= A_1 \\
Y_2 &= A_2 \cdot \overline{A_1} \\
Y_3 &= A_3 \cdot \overline{A_2} \cdot \overline{A_1} \\
&\ldots \\
Y_N &= A_N \cdot \overline{A_{N-1}} \cdot \ldots \cdot \overline{A_1}
\end{aligned}
\tag{10.47}
$$

We can express priority encoding as a prefix operation by defining a prefix $X_{i,j}$ 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 $i \geq k > j$:

$$
\begin{aligned}
X_{i:i} &= \overline{A_i} && \text{bitwise precomputation} \\
X_{i:j} &= X_{i:k} \bullet X_{k-1:j} && \text{group logic} \\
Y_i &= A_i \bullet X_{i-1:1} && \text{output logic}
\end{aligned}
\qquad \textbf{(10.48)}
$$

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

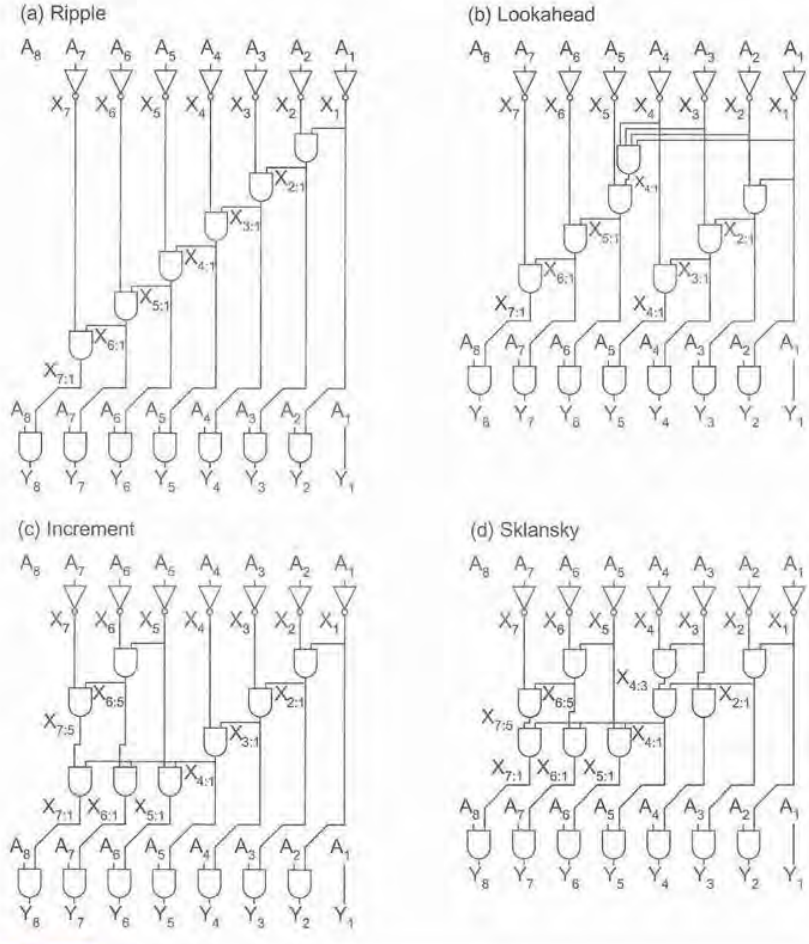


FIG 10.86 Priority encoder trees

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

$$\begin{aligned} X_{i:i} &= \overline{A_i} && \text{bitwise precomputation} \\ W_{i:i} &= A_i \\ X_{i:j} &= X_{i:k} \cdot X_{k-1:j} \\ W_{i:j} &= W_{i:k} \cdot X_{k-1:j} + X_{i:k} \cdot W_{k-1:j} && \text{group logic} \\ Y_i &= A_i \cdot X_{i-1:1} \\ Z_i &= A_i \cdot W_{i-1:1} && \text{output logic} \end{aligned} \tag{10.50}$$

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.

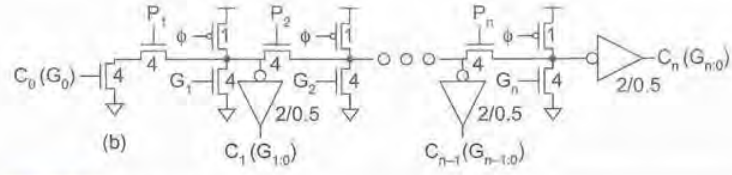


FIG 10.87 Manchester carry chain

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_i, $2X_i$, and M_i. 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*.

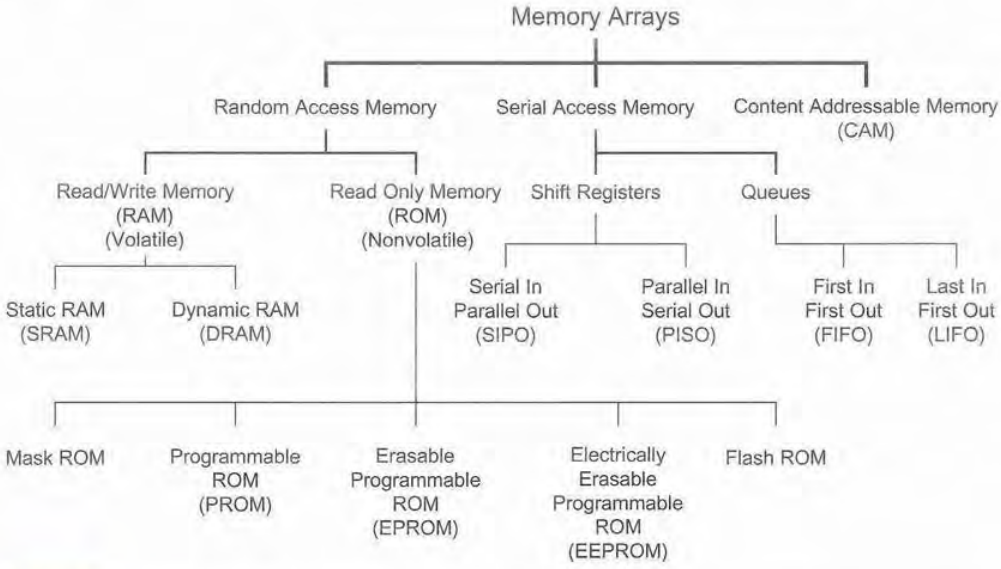


FIG 11.1 Categories of memory arrays

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

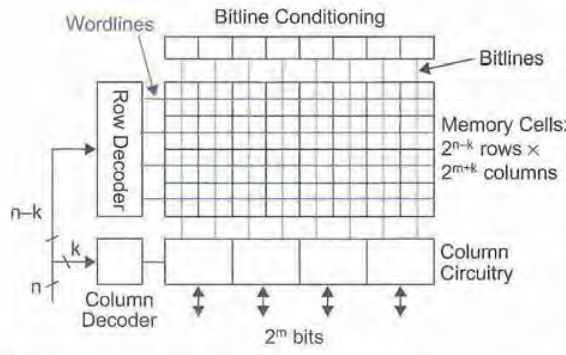


FIG 11.2 General memory array architecture

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.

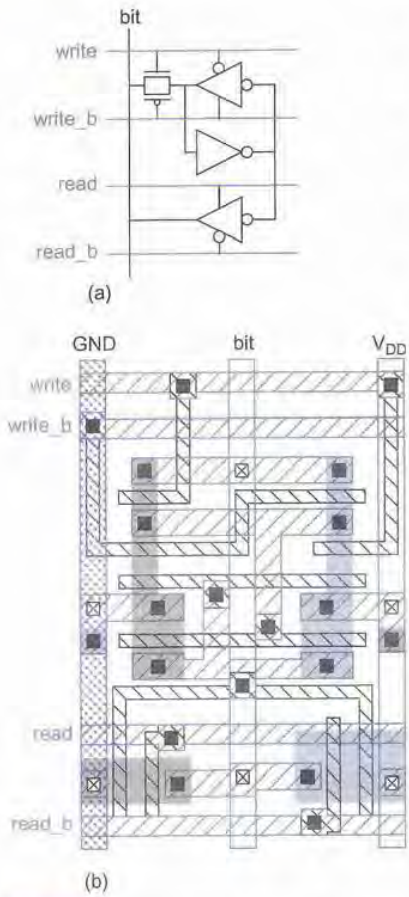


FIG 11.3 12-transistor SRAM cell

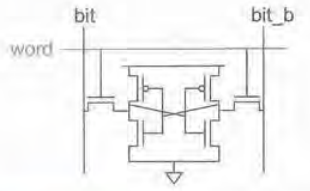


FIG 11.4 6-transistor SRAM cell

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.

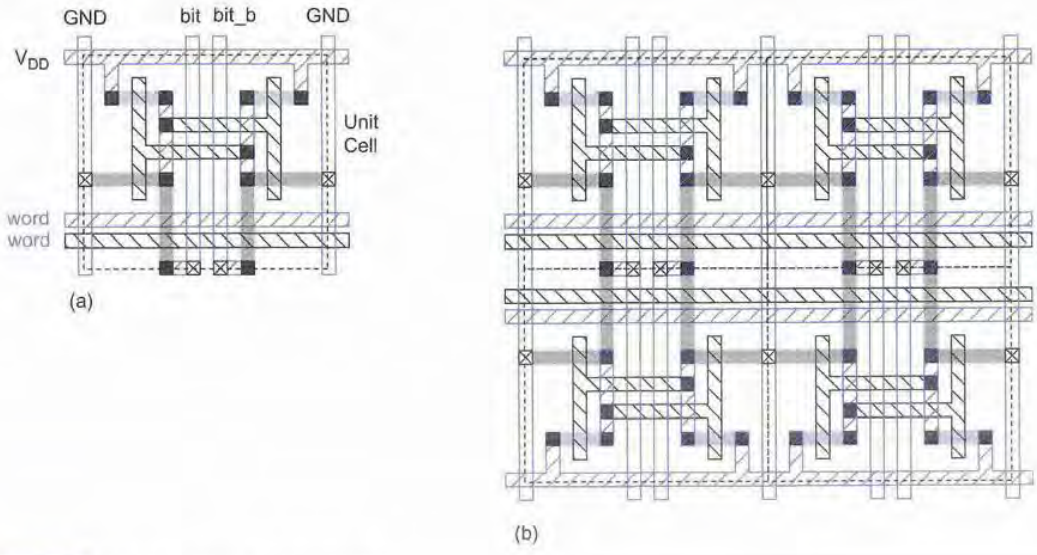


FIG 11.5 Stick diagram of 6T SRAM cell

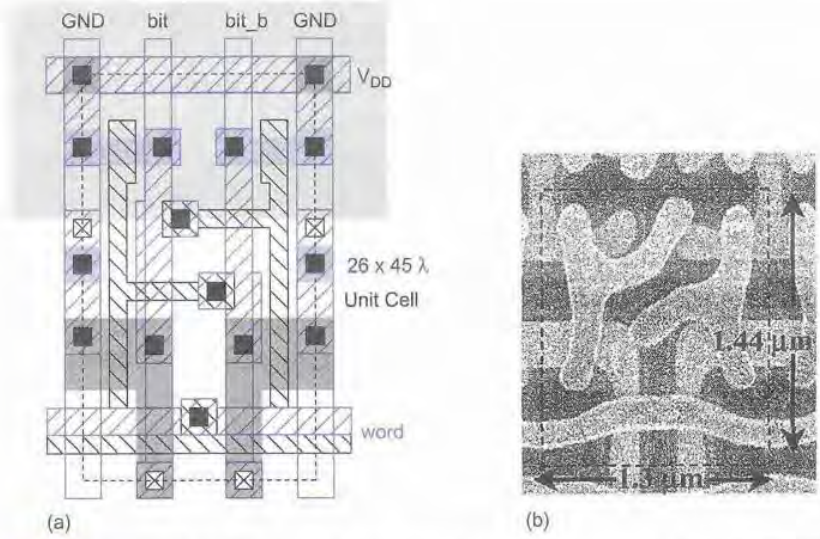


FIG 11.6 Layout of 6T SRAM cell. © IEEE 2001. Color version on inside front cover.

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.

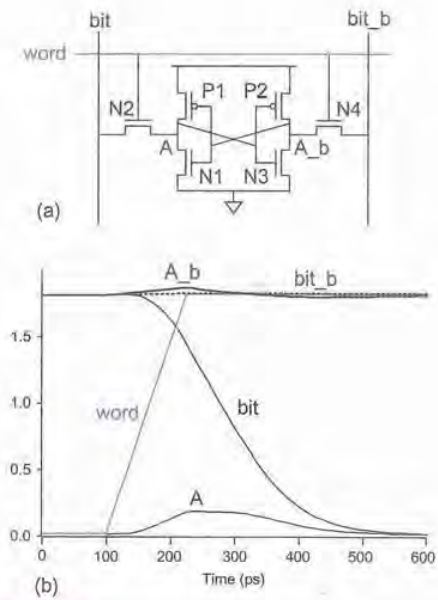


FIG 11.7 Read operation for 6T SRAM cell

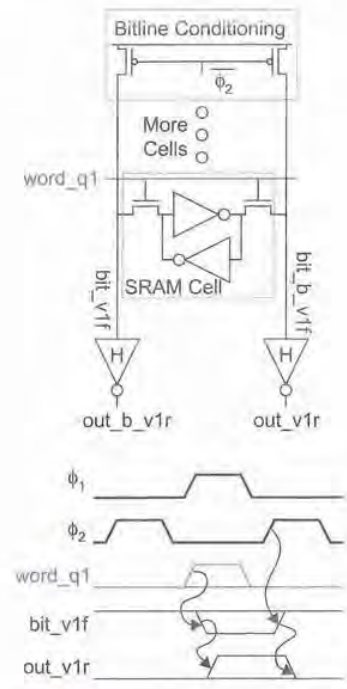


FIG 11.8 SRAM column read

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.

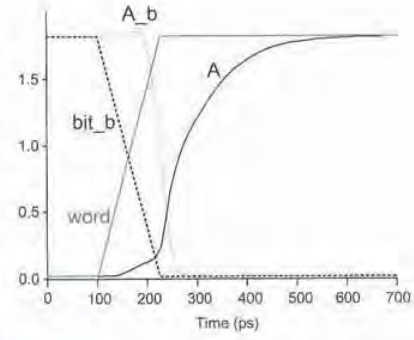


FIG 11.9 Write operation for 6T SRAM cell

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.

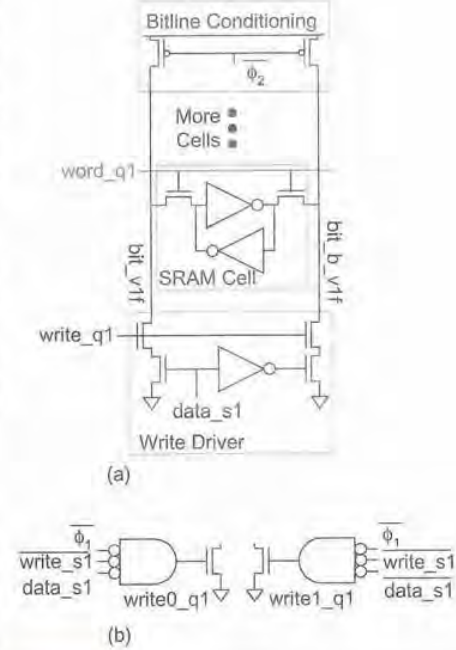


FIG 11.10 SRAM column write

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-k} 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.

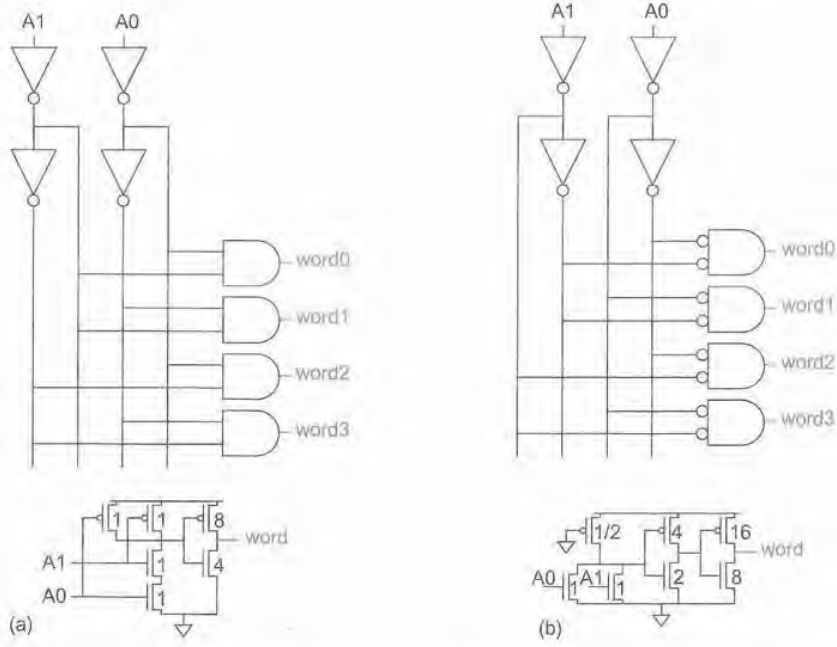


FIG 11.11 Decoders

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

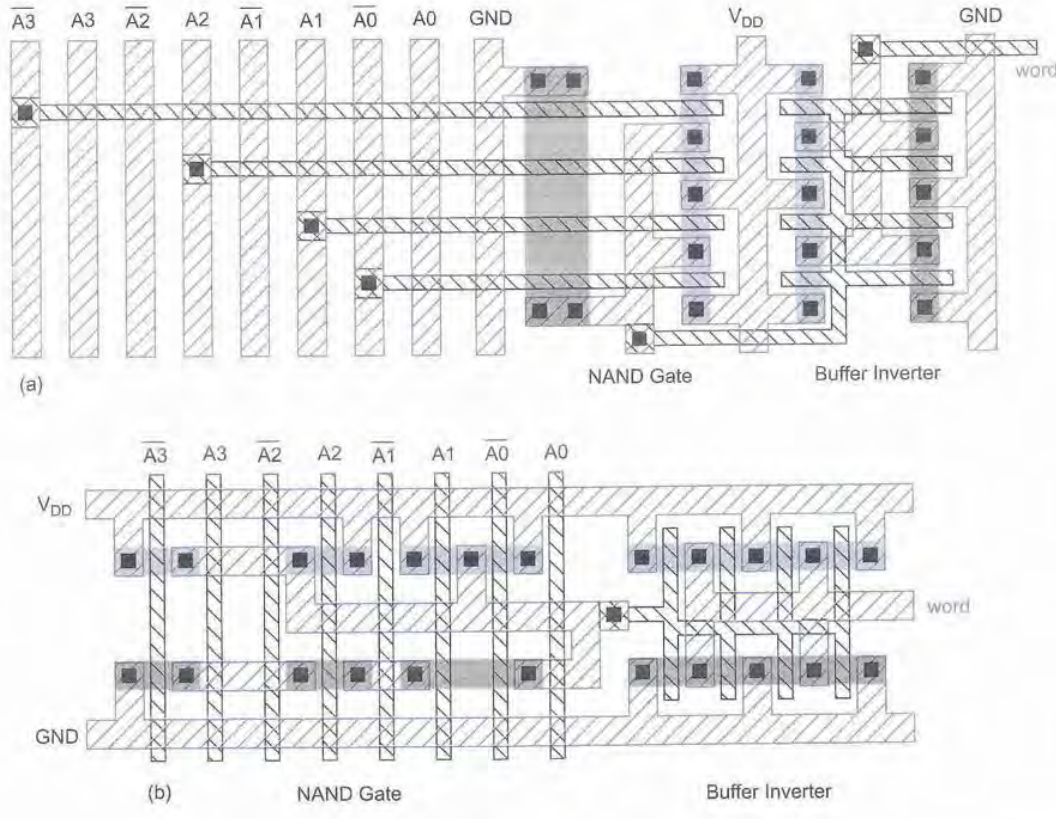


FIG 11.12 Stick diagrams of two decoder layouts

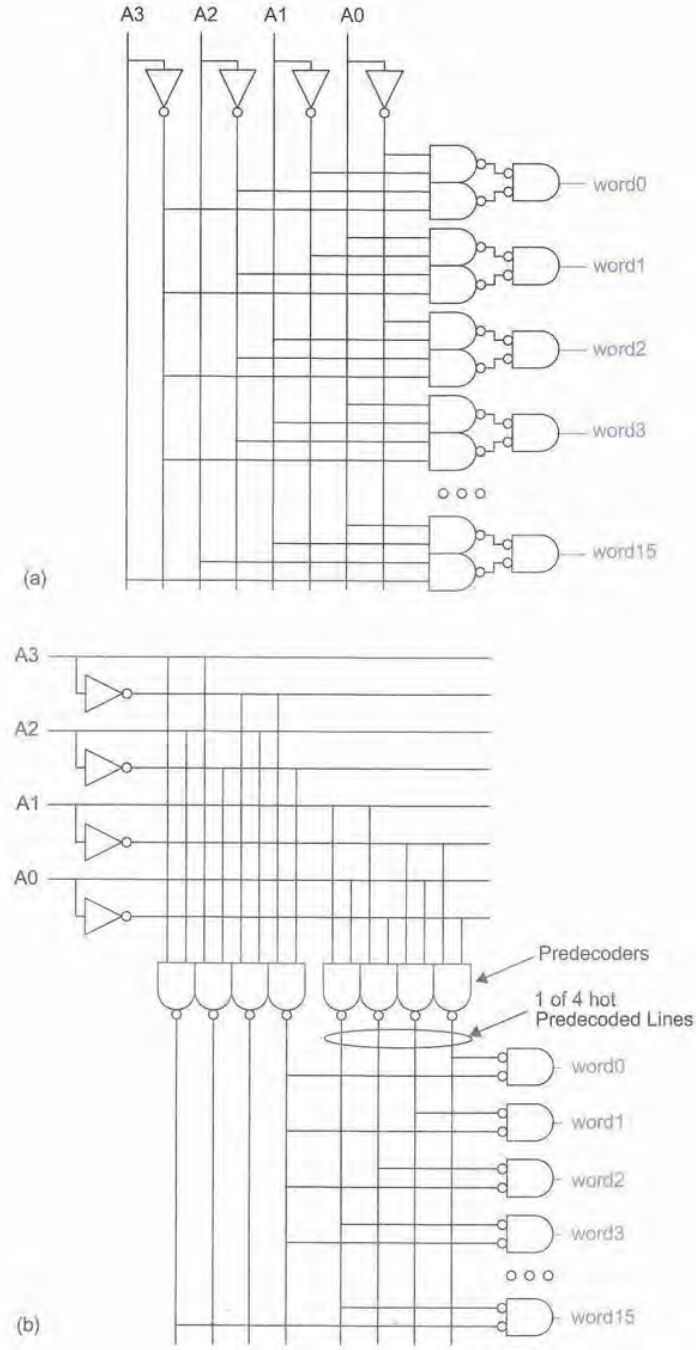


FIG 11.13 Ordinary and predecoding circuits

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.

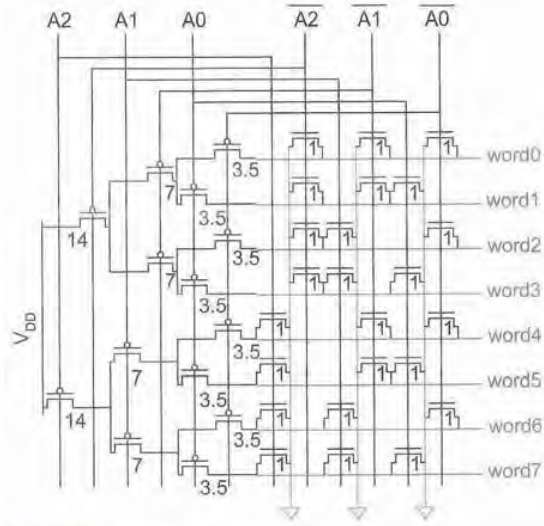


FIG 11.14 Lyon-Schediwy decoder

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].

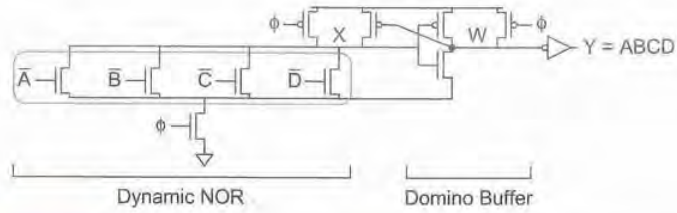


FIG 11.15 4-input AND using race-based NOR



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.

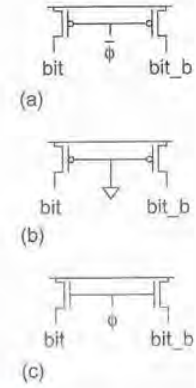


FIG 11.16 Bitline conditioning circuits

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].

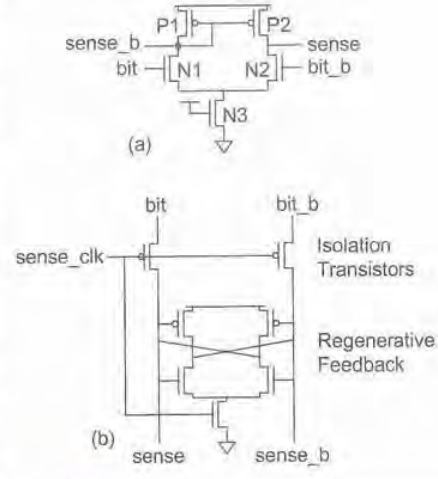


FIG 11.17 Sense amplifiers

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-

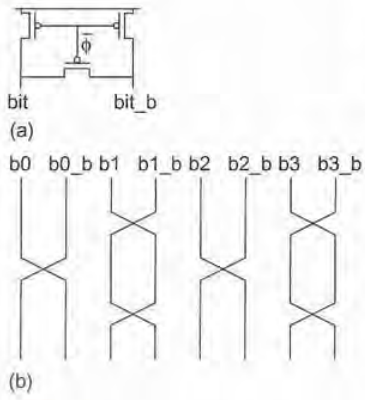


FIG 11.18 Bitline noise reduction through equalizers and twisting

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

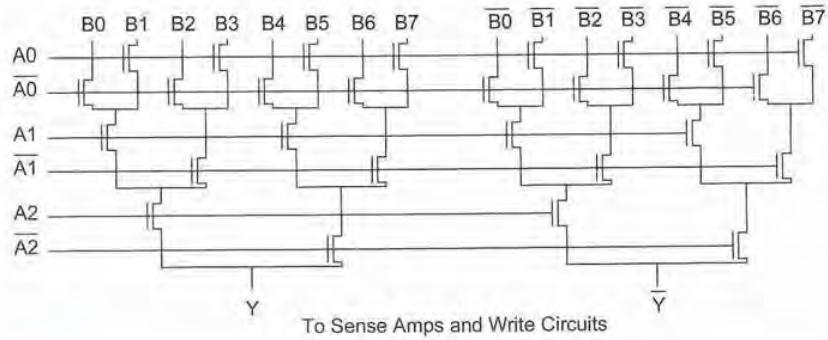


FIG 11.19 Tree decoder column multiplexer

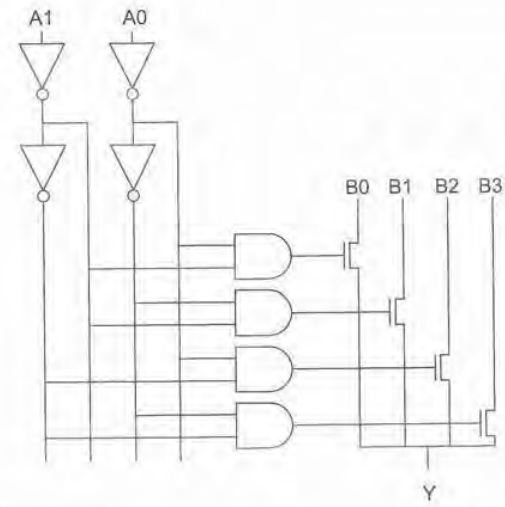


FIG 11.20 Column multiplexer with separate decoder

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.

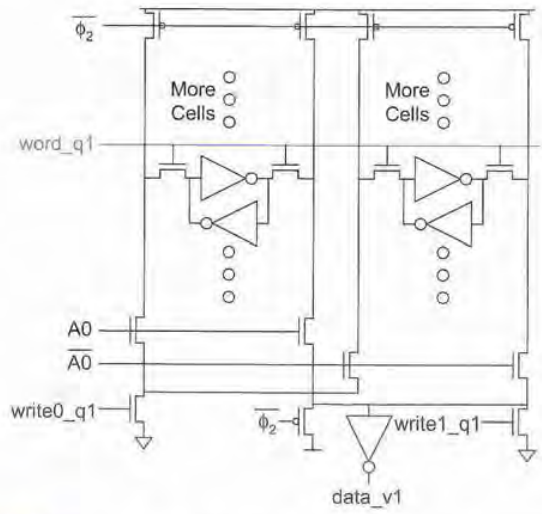


FIG 11.21 Complete pair of columns for 2-way multiplexed SRAM

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.

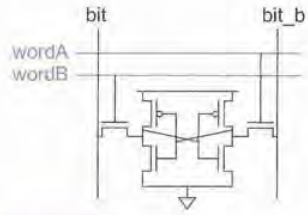


FIG 11.22 Simple dual-ported SRAM

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

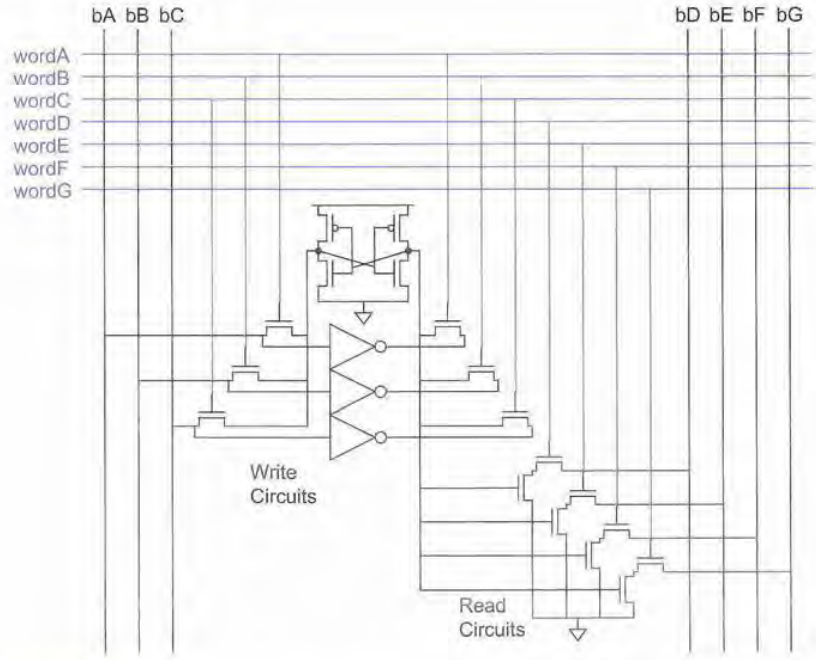


FIG 11.23 General multi-ported SRAM

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-

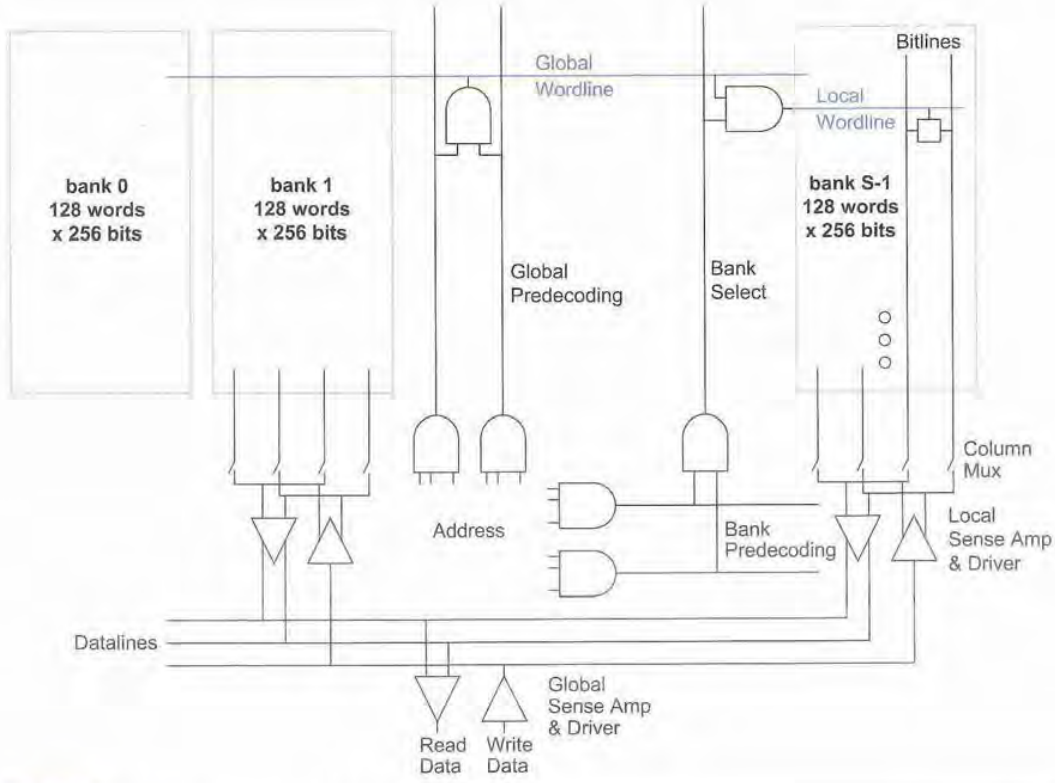


FIG 11.24 Large memory array architecture with subarrays

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 ($N = m + n$).

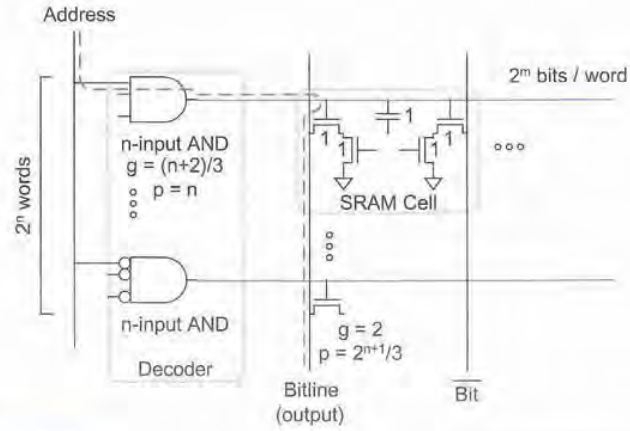


FIG 11.25 Critical path for read of small SRAM

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^nC. The cell has an effective resistance of $2R$ discharging the bitline through two series unit transistors. Hence, the bitline has a parasitic delay of $2^{n+1}RC$. Normalized by $\tau = 3RC$, this gives $p = 2^{n+1}/3$.

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 $P = n + 2^{n+1}/3$. 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 $D = 4\log_4 F + P = 2^N + 4\log_4[(n + 2)/3] + n + 2^{n+1}/3$. 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%.

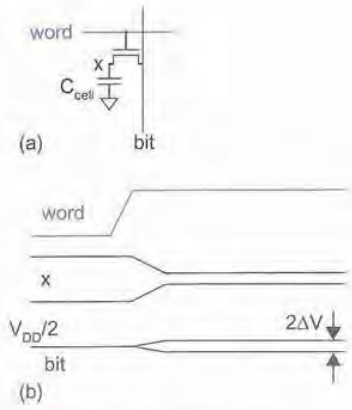
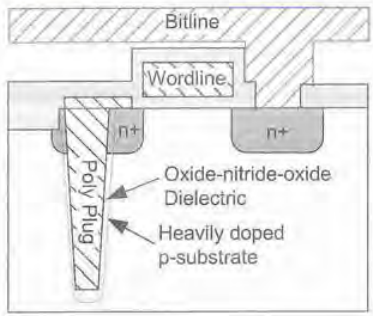


FIG 11.26 DRAM cell read operation



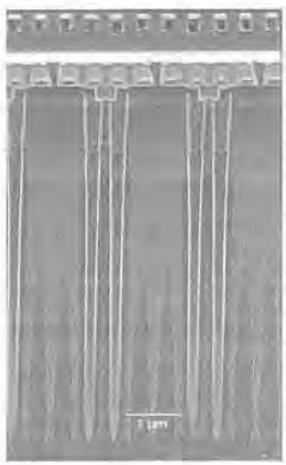


FIG 11.27 Trench capacitor

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.

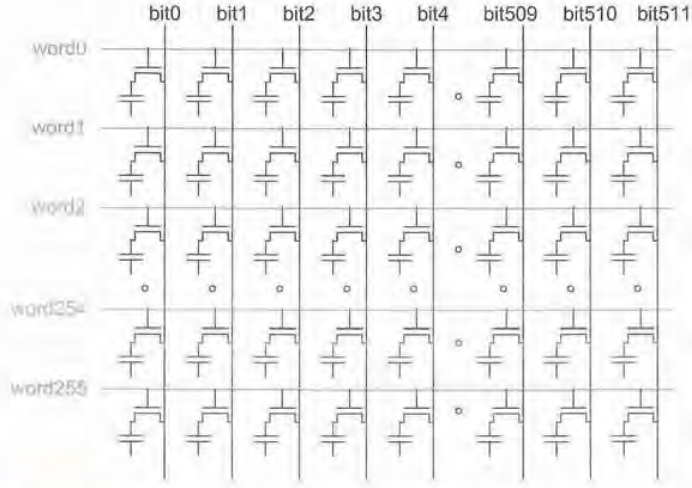


FIG 11.28 DRAM subarray

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

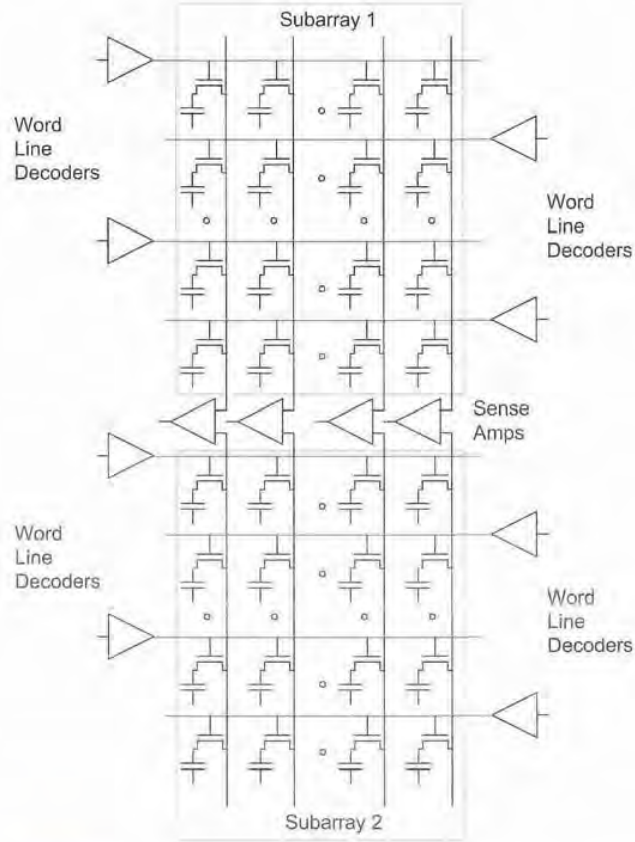


FIG 11.29 Open bitlines

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.

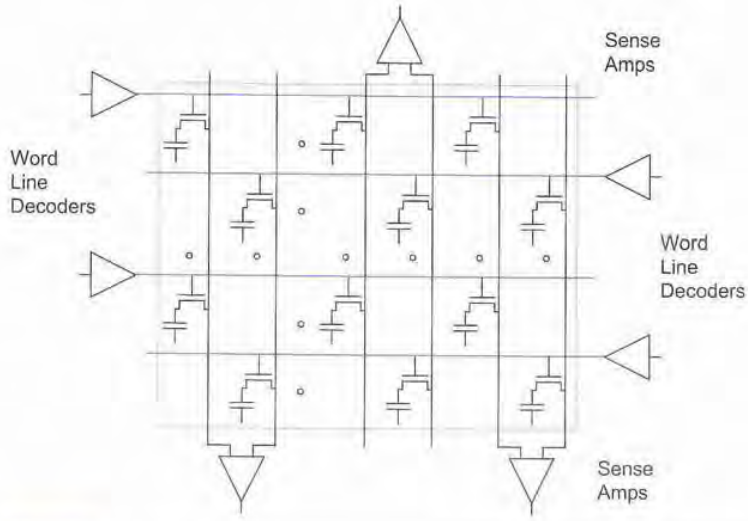


FIG 11.30 Folded bitlines

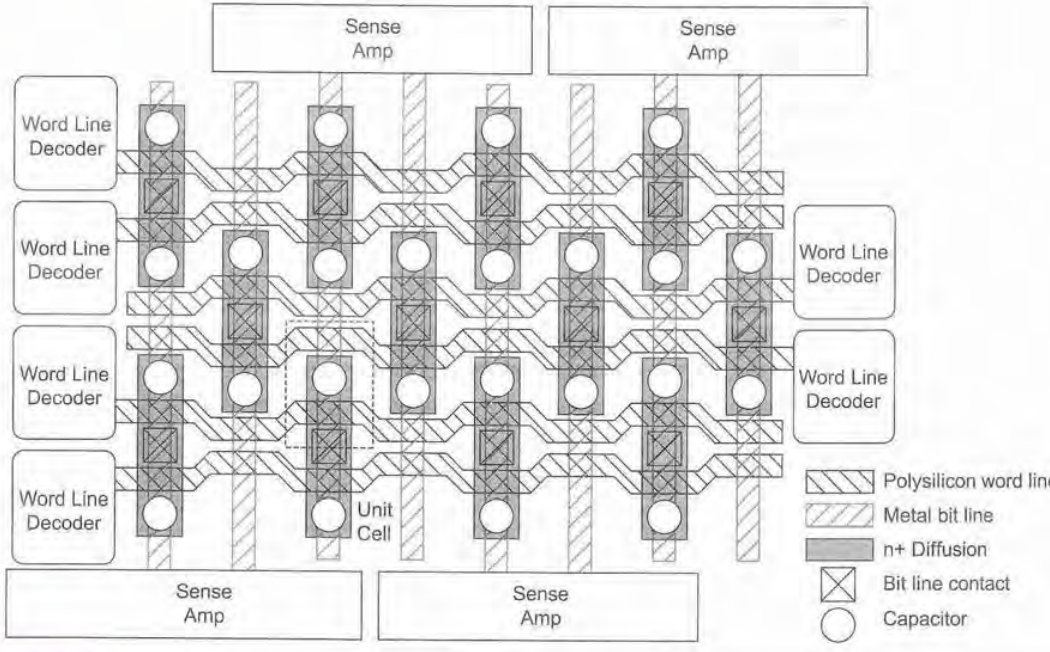


FIG 11.31 Layout of folded bitline subarray

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.

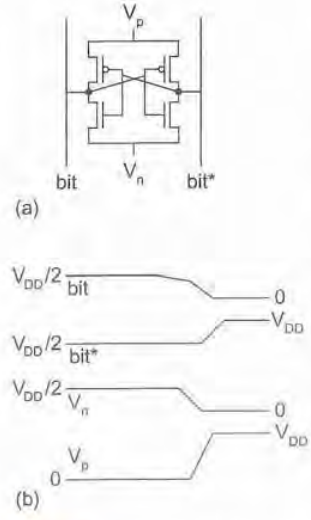


FIG 11.32 Sense amplifier

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.

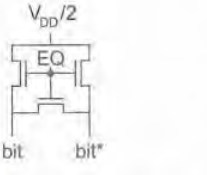


FIG 11.33 Bitline conditioning

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.

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].

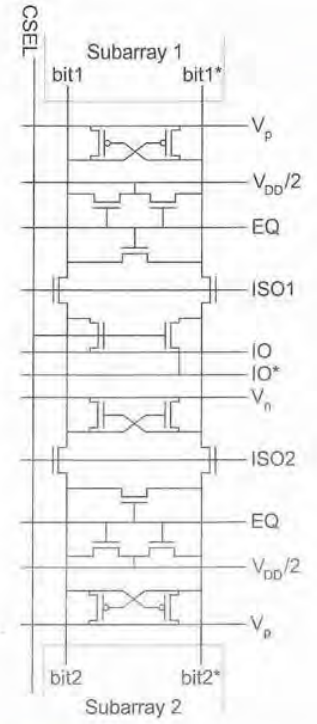


FIG 11.34 Column circuitry

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
```

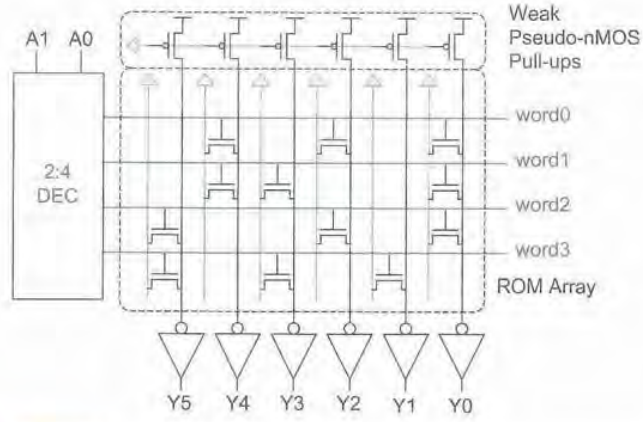


FIG 11.35 Pseudo-nMOS ROM

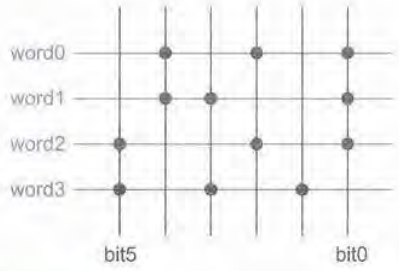


FIG 11.36 Dot diagram representation of ROM

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.

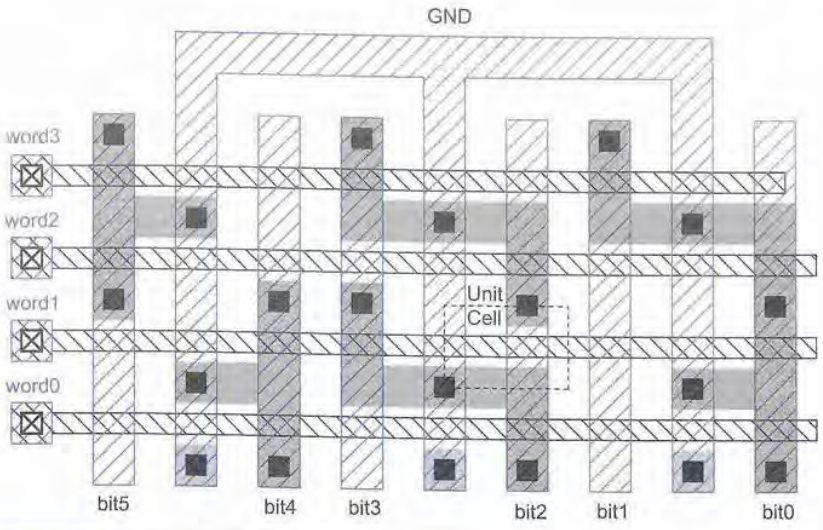


FIG 11.37 ROM array layout

[1]The cell can be reduced to 11 x 7 λ by running the ground line in diffusion and by reducing the width and spacing to 3 λ.

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.

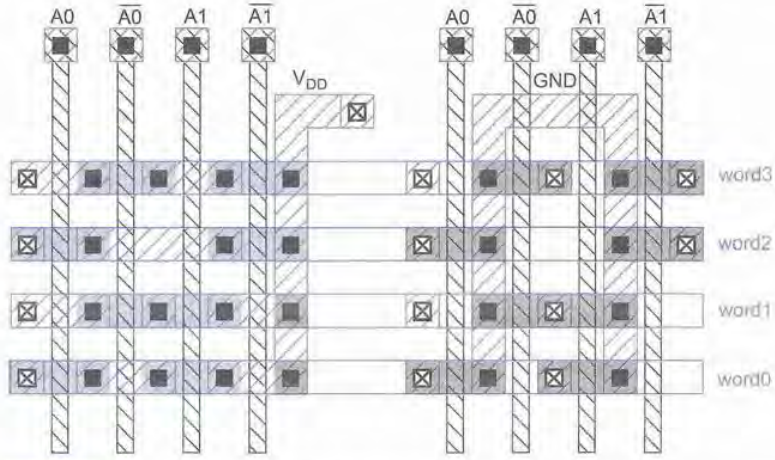


FIG 11.38 Row decoder layout on tight pitch

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.

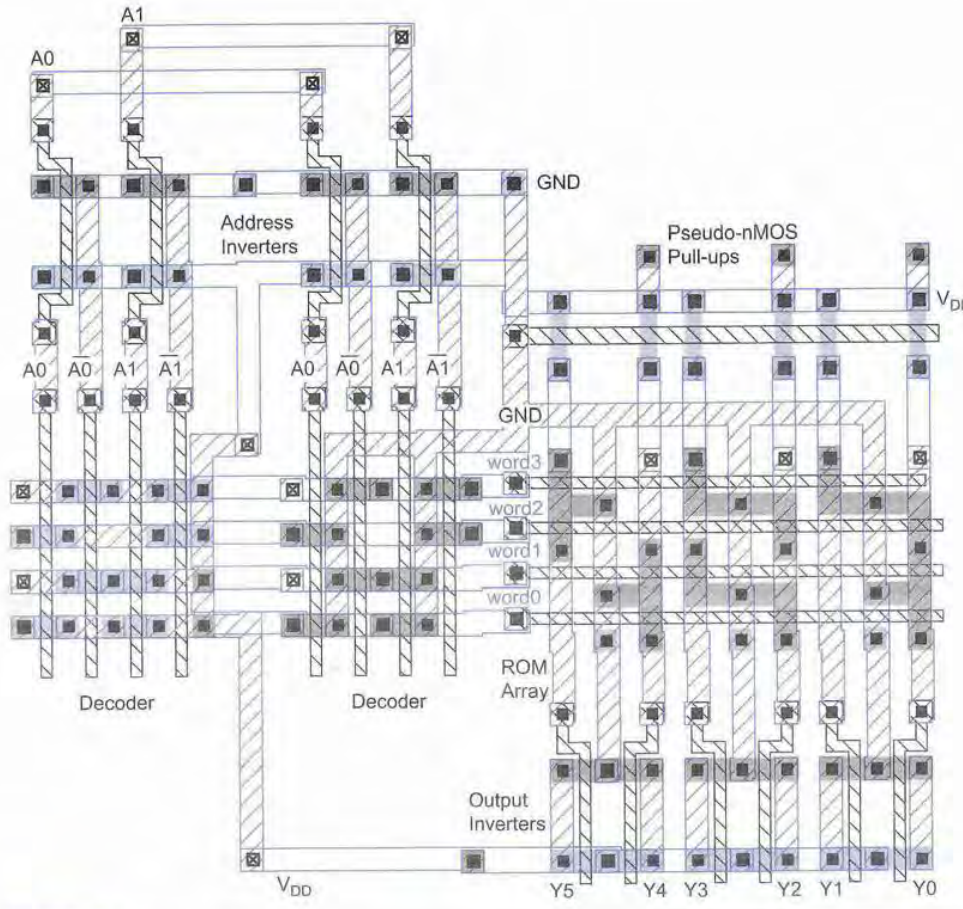


FIG 11.39 Complete ROM layout

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.

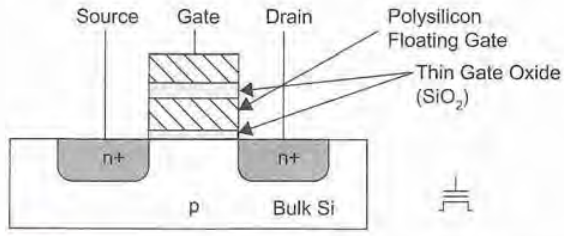


FIG 11.40 Cross-section of floating gate nMOS transistor

[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

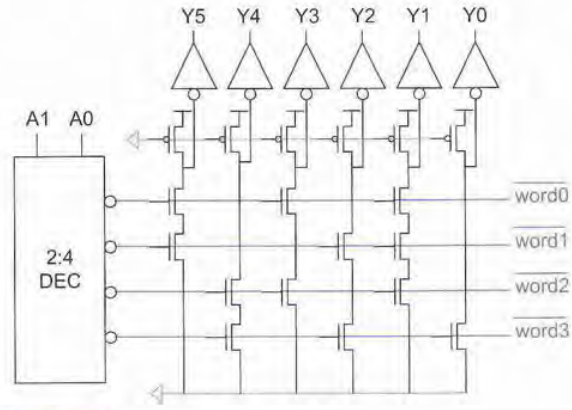


FIG 11.41 Pseudo-nMOS NAND ROM

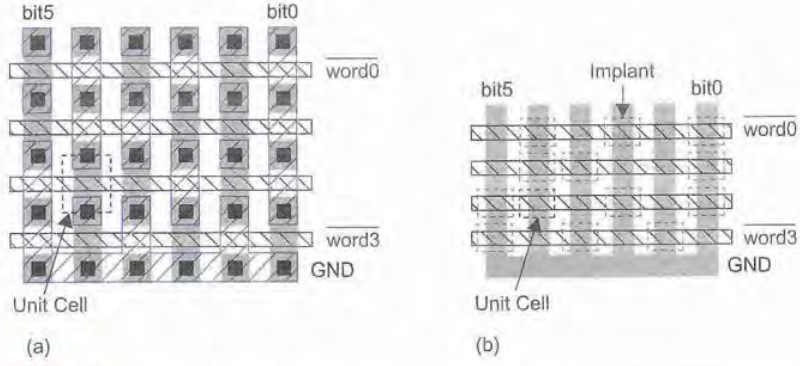


FIG 11.42 NAND ROM array layouts

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.

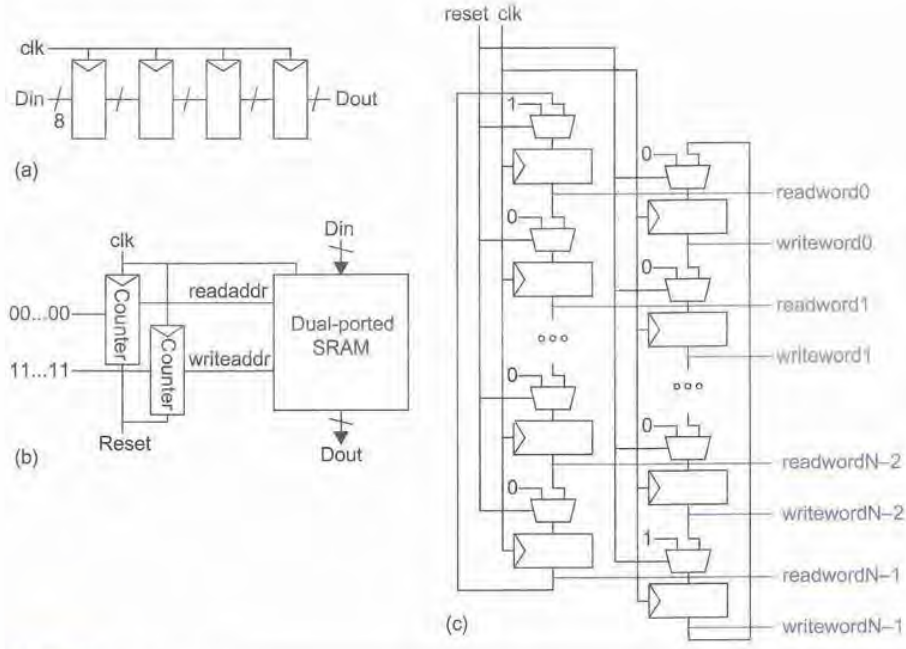


FIG 11.43 Shift registers

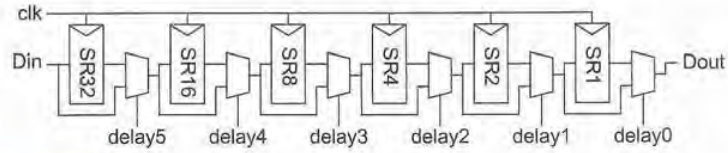


FIG 11.44 Tapped delay line

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.

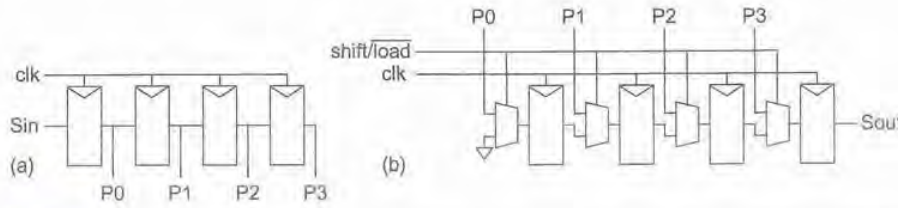


FIG 11.45 Serial/parallel memories

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.

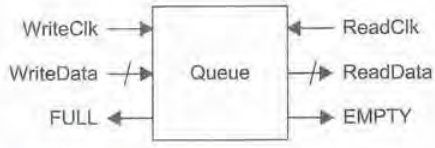


FIG 11.46 Queue

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*.

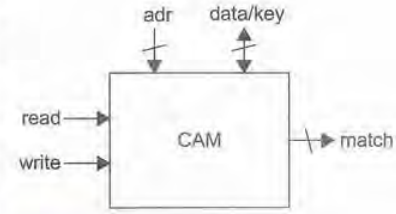


FIG 11.47 Content-addressable memory

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"

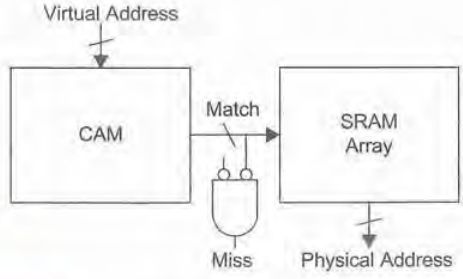


FIG 11.48 Translation Lookaside Buffer (TLB) using CAM

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.

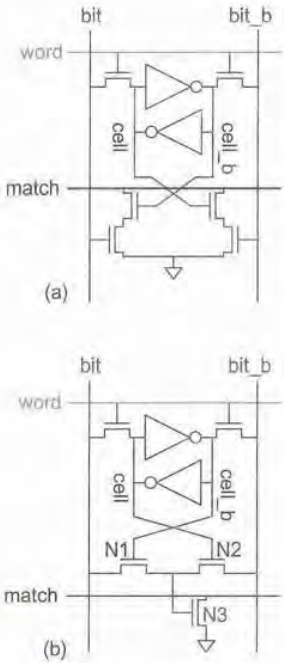


FIG 11.49 CAM cell implementations

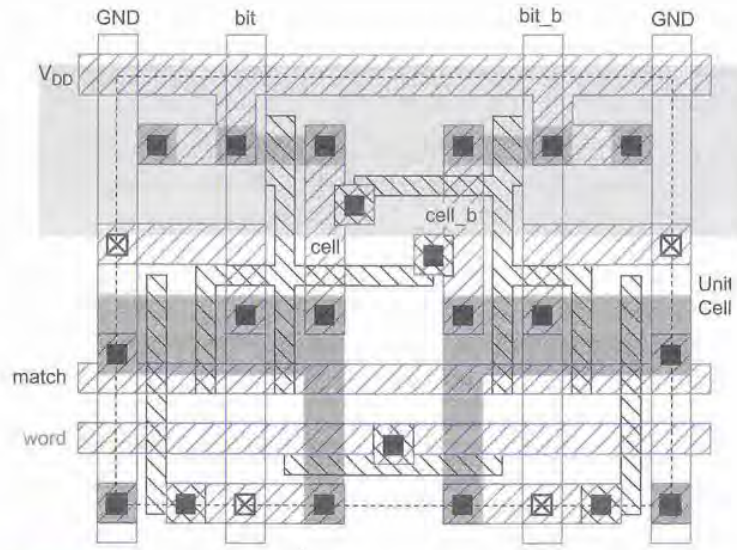


FIG 11.50 CAM cell layout. Color version on inside front cover.

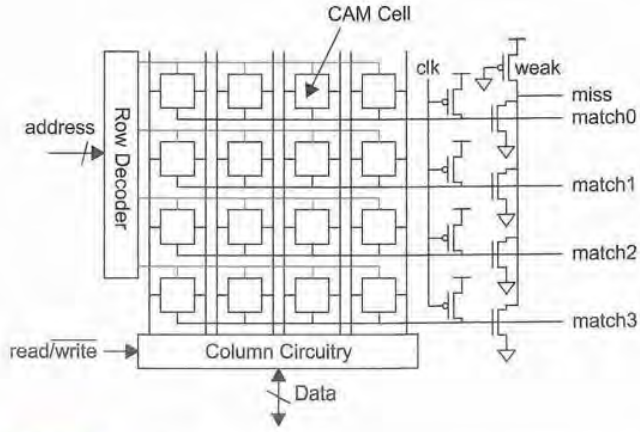


FIG 11.51 4 × 4 CAM array

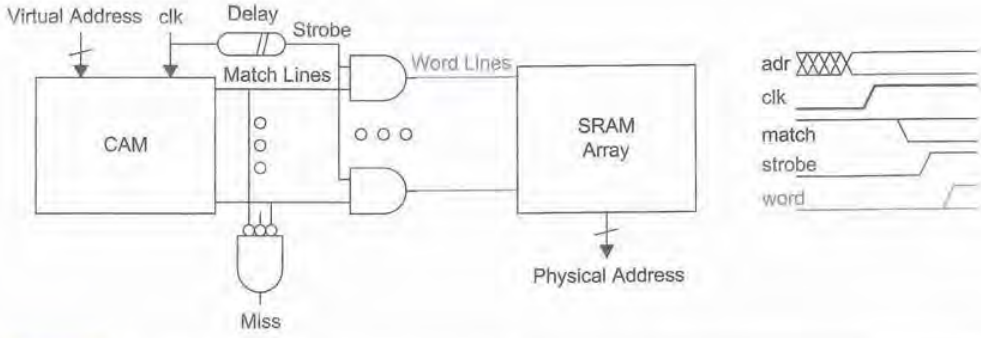


FIG 11.52 Refined TLB path with monotonic wordlines

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

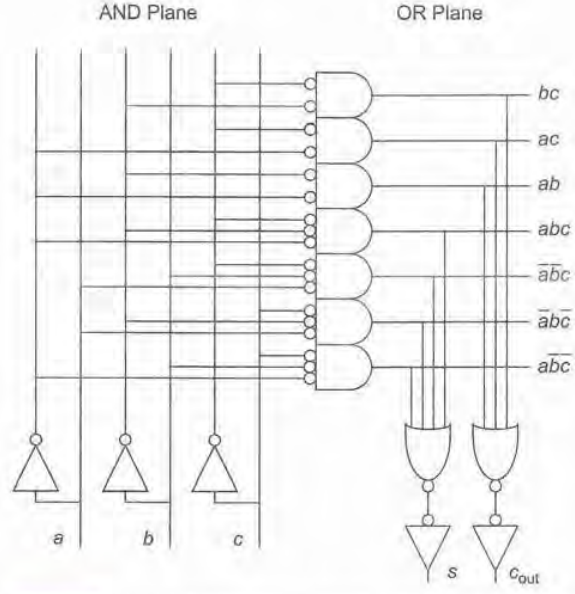


FIG 11.53 NOR/NOR representation of PLA

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

$$s = a\bar{b}\bar{c} + \bar{a}b\bar{c} + \bar{a}\bar{b}c + abc$$
$$c_{out} = ab + bc + ac \tag{11.2}$$

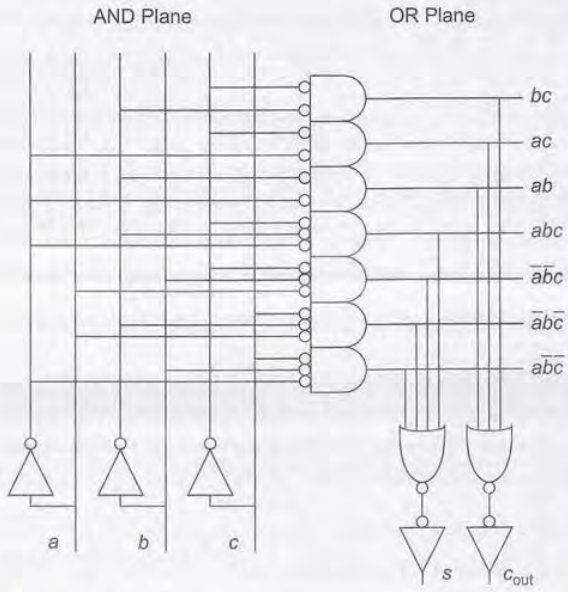


FIG 11.54 AND/OR representation of PLA

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.

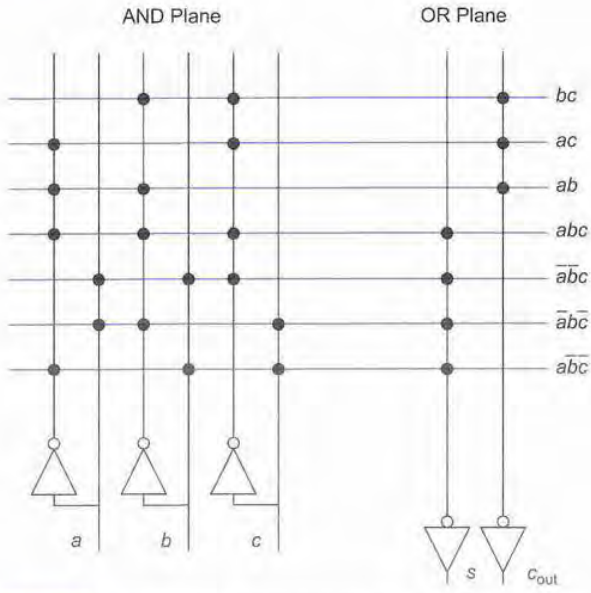


FIG 11.55 Dot diagram representation of PLA

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).

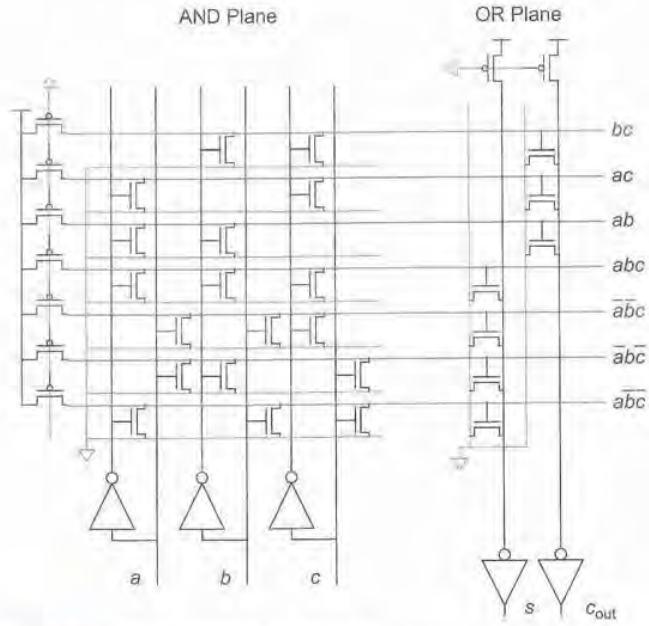


FIG 11.57 Pseudo-nMOS PLA schematic

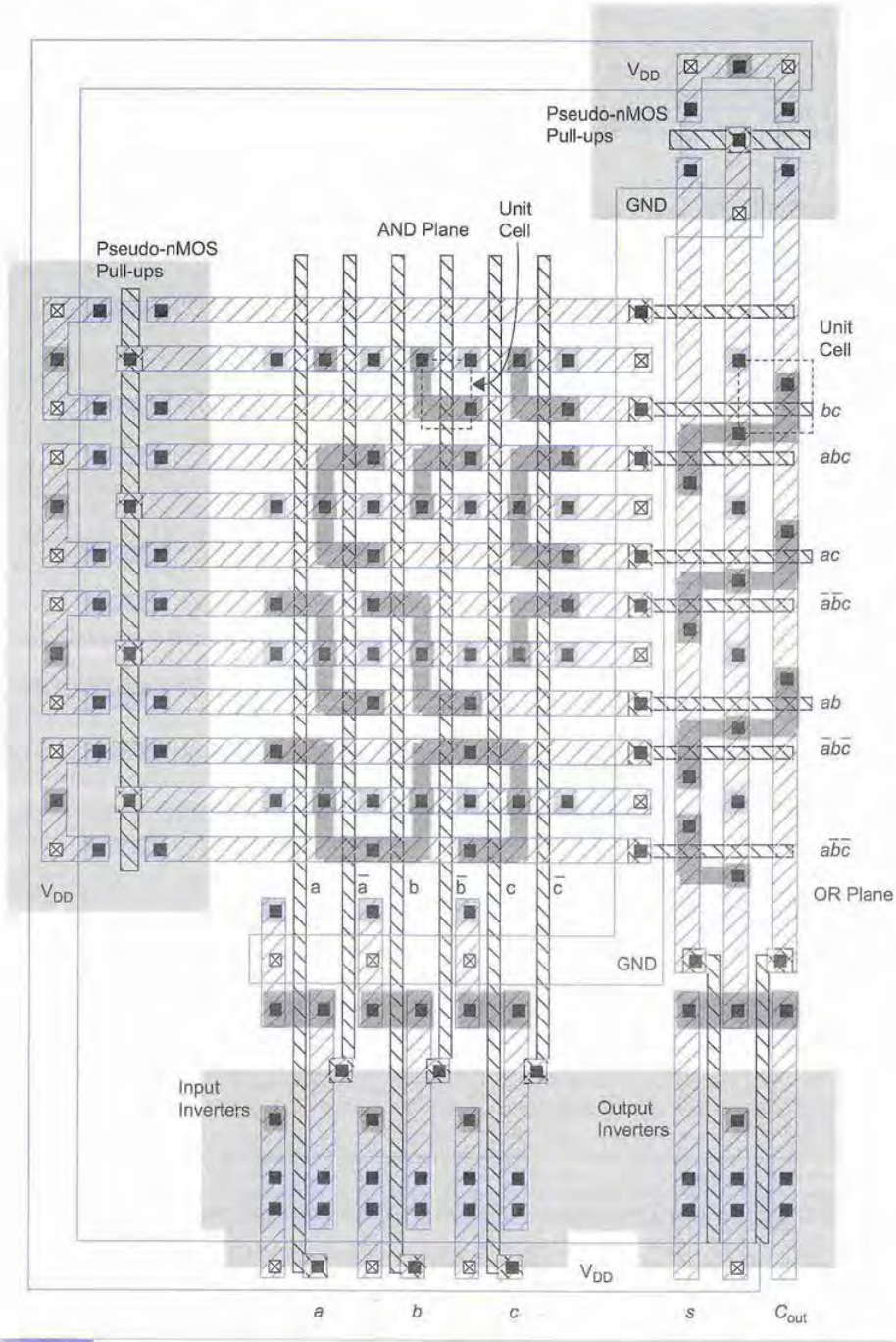


FIG 11.58 Pseudo-nMOS PLA layout

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.

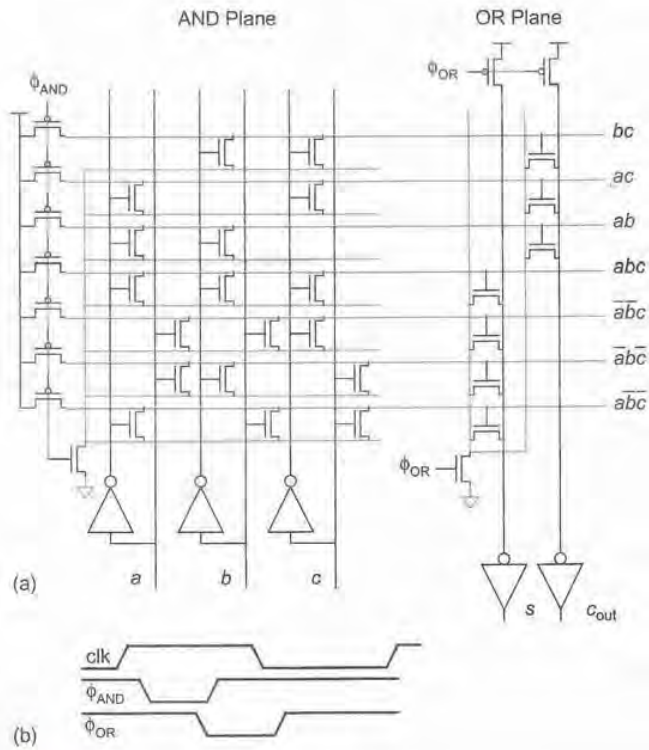


FIG 11.59 Dynamic PLA schematic (continues)

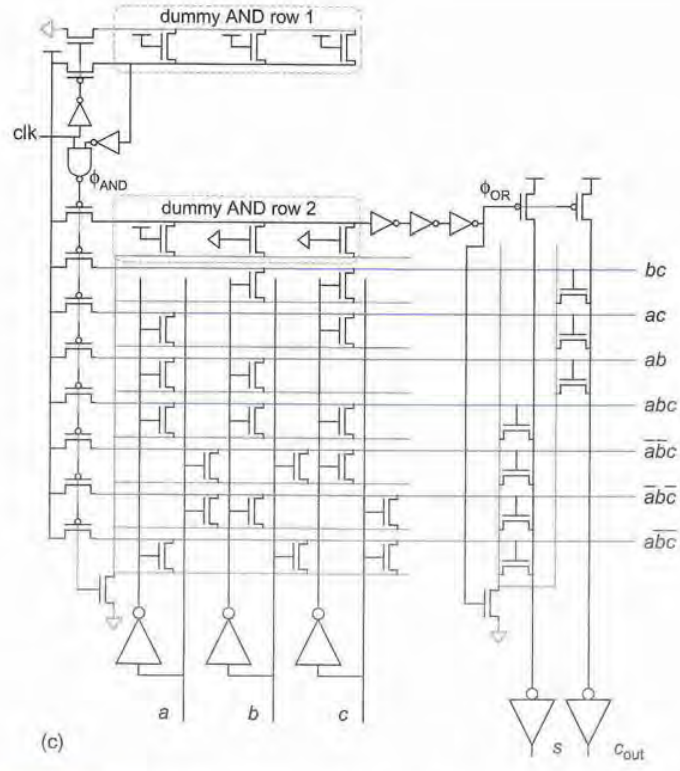


FIG 11.59 Dynamic PLA schematic (continued)

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

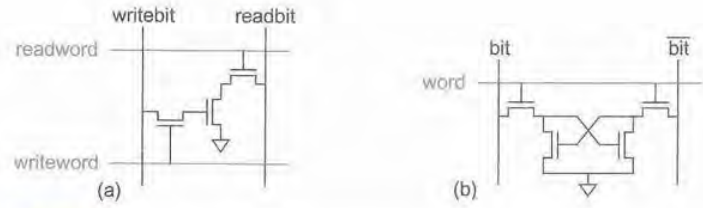


FIG 11.60 3T and 4T DRAM cells

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 lookaside 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.

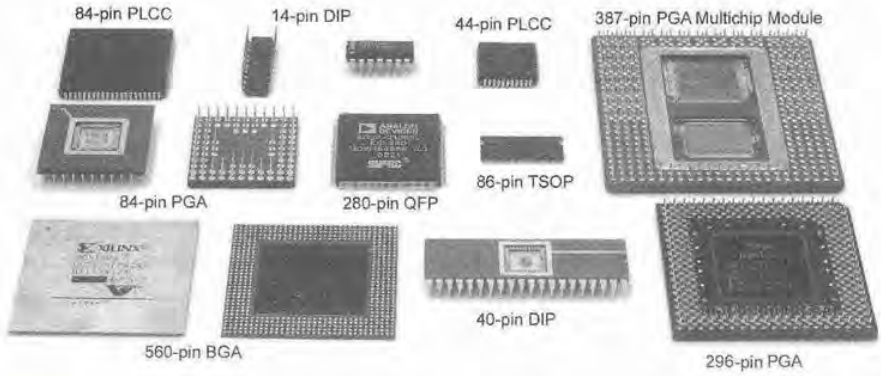


FIG 12.1 Integrated circuit packages. © 2003 Harvey Mudd College. Reprinted with permission.

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.

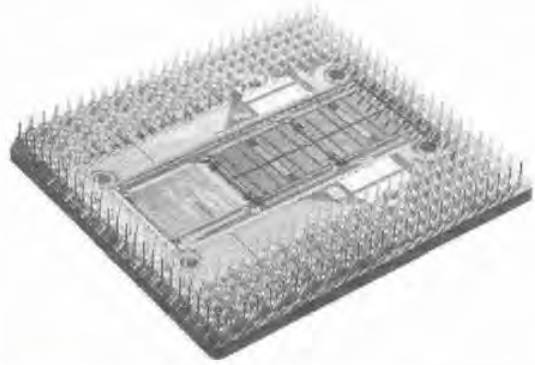


FIG 12.2 Pentium Pro MCM. Reprinted with permission of Intel Corporation.

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

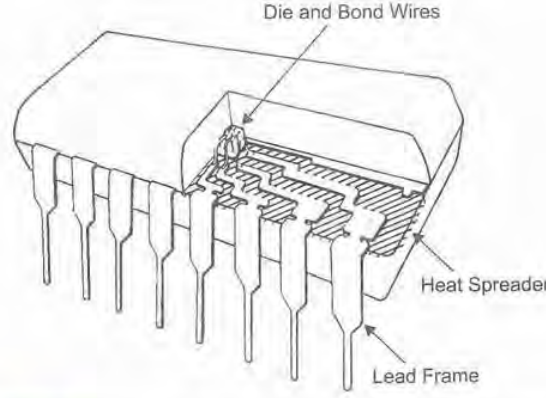


FIG 12.3 Cutaway view of dual-in-line package. © IEEE 1985 [Mahalingam85]

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.

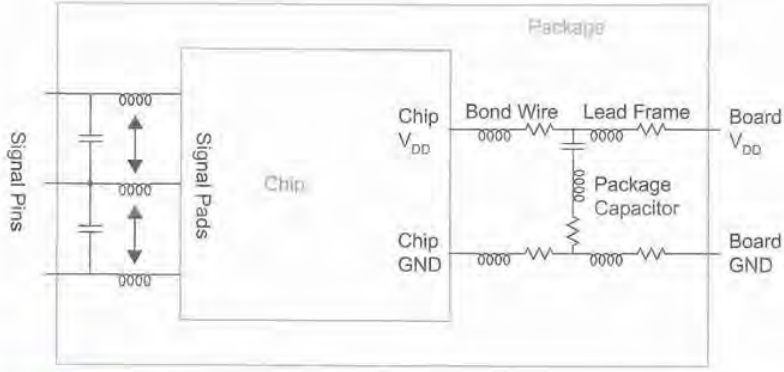


FIG 12.4 Package parasitics

12.2.4 Heat Dissipation

A 60-watt light bulb has a surface area of about 120 cm^2 and is too hot to touch. In comparison, the Itanium 2 dissipates 130 W on a 4 cm^2 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/(cm² °C) [Glasser85]. Thus, a package with a surface area of 10 cm² 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/(cm² ° 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

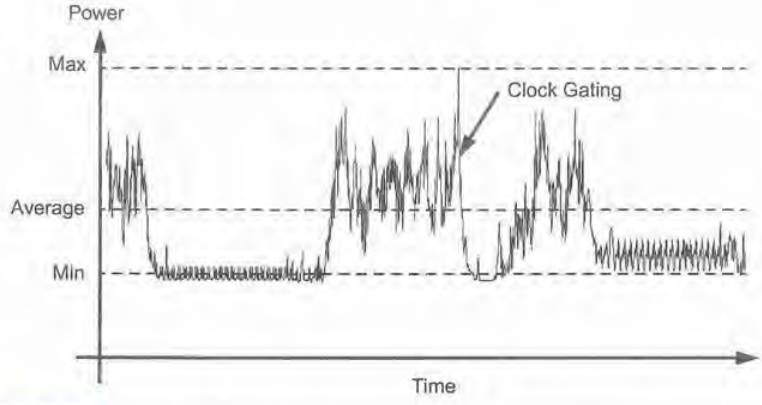


FIG 12.5 Time-dependent power consumption of microprocessor. Reprinted with permission of Sun Microsystems.

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

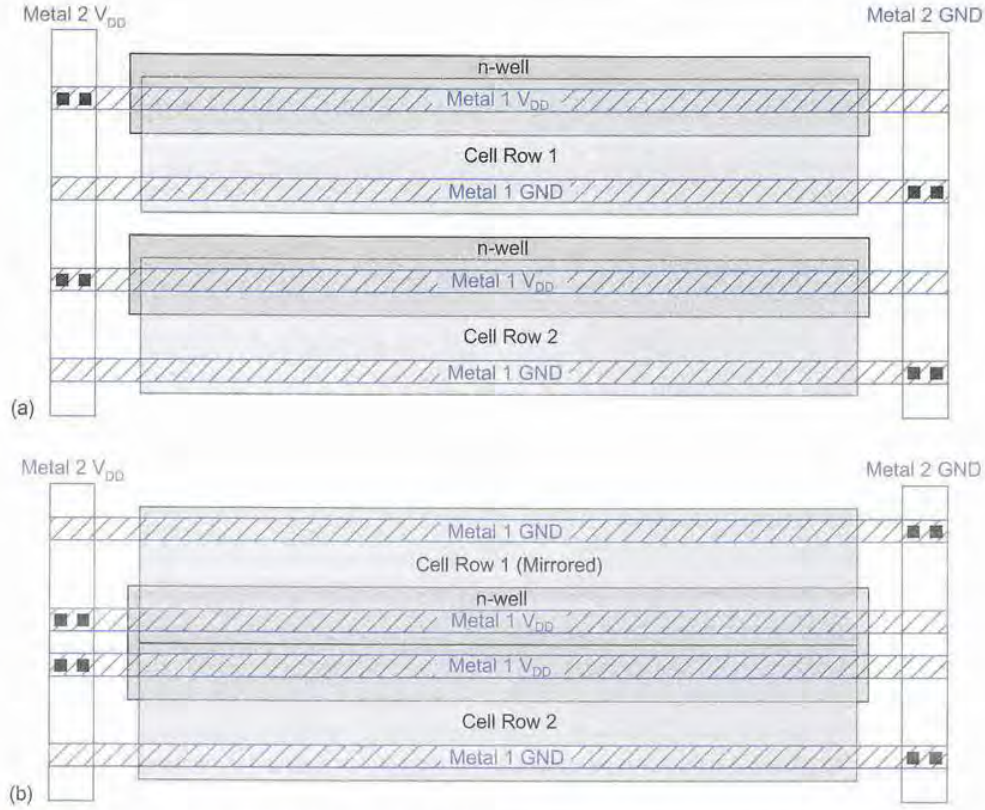


FIG 12.6 Power distribution for standard cell layout

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.

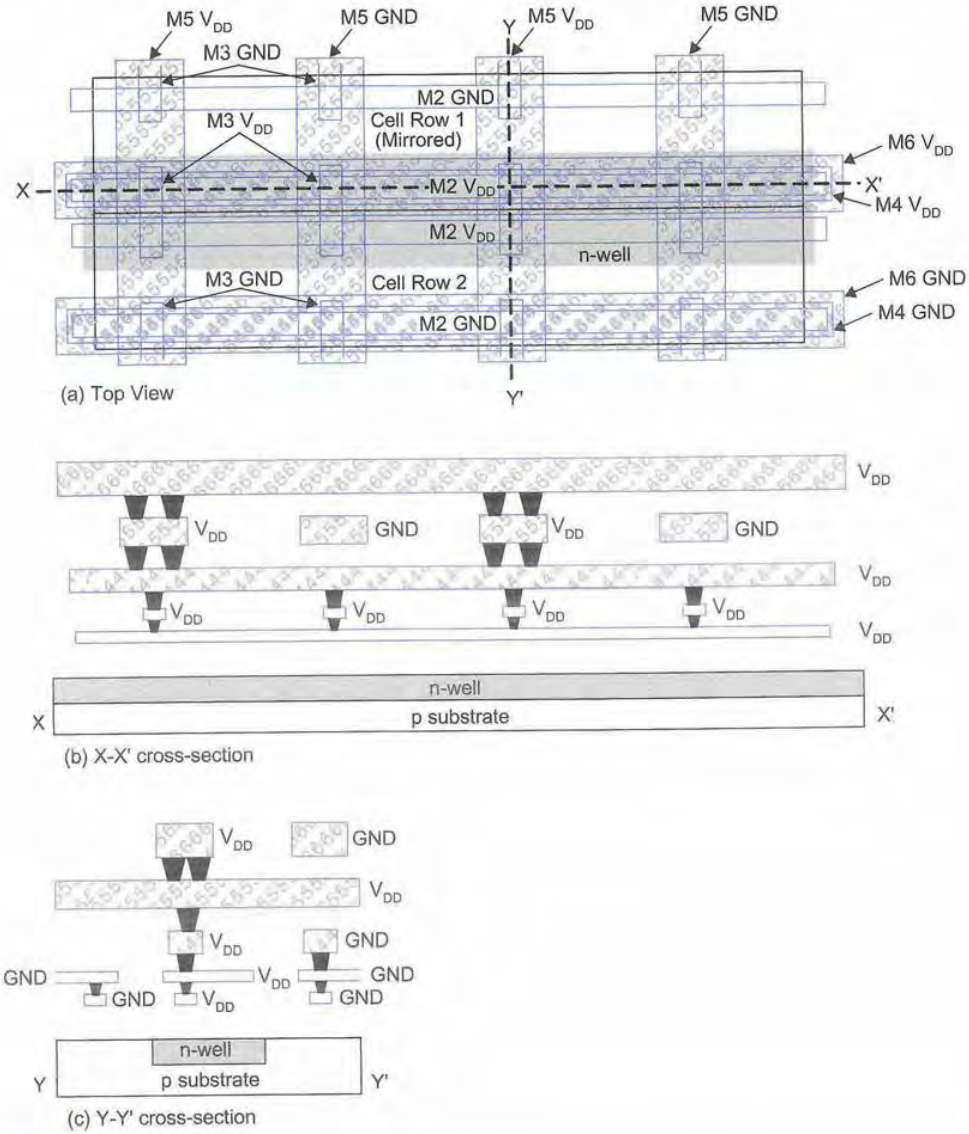


FIG 12.7 Power grid layout

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)

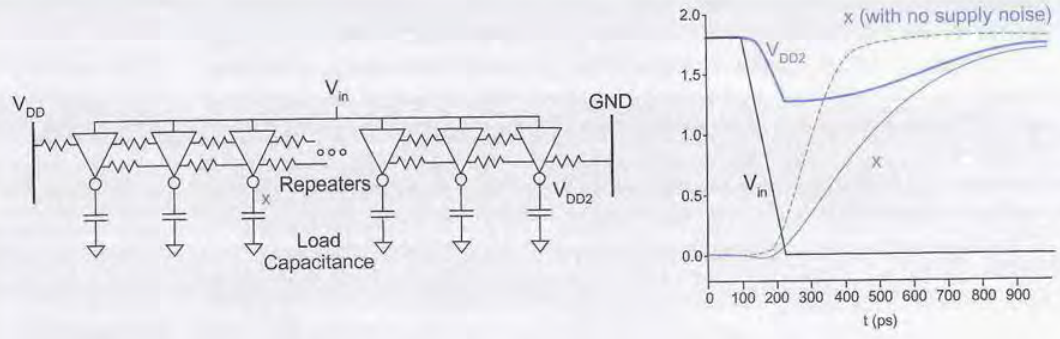


FIG 12.8 Power supply droop

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.

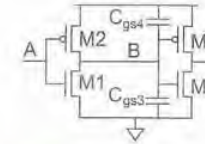


FIG 12.9 Symbiotic bypass capacitance

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/μm^2. At 9% utilization, this corresponds to $0.18/f$ nF/mm^2. Half of that, or $0.09/f$ nF/mm^2, serves as symbiotic bypass capacitance on average. In an f = 180 nm process, this means the symbiotic bypass capacitance is approximately 0.5 nF/mm^2.

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.

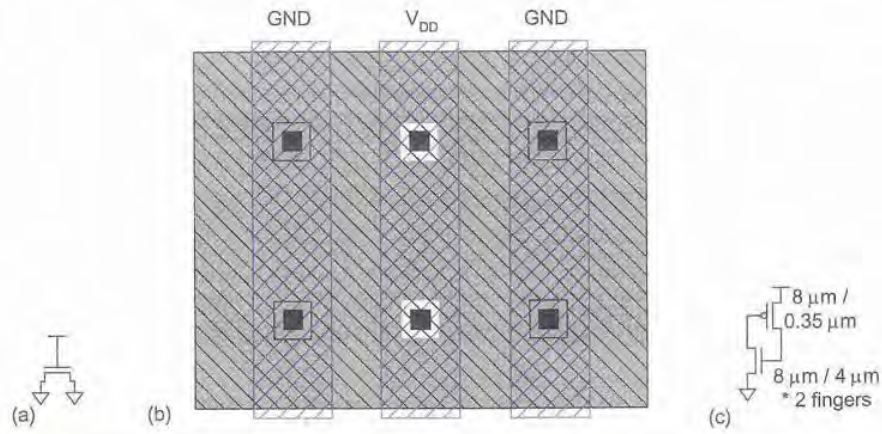


FIG 12.10 Waffle bypass capacitor layout

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.

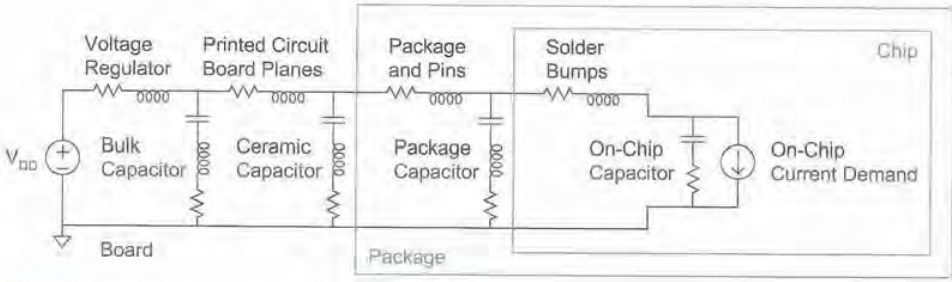


FIG 12.11 Power distribution system model

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

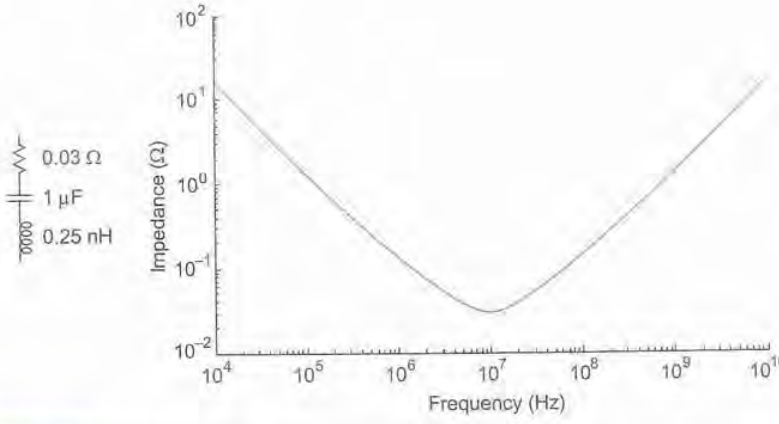


FIG 12.12 Impedance of bypass capacitor

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.

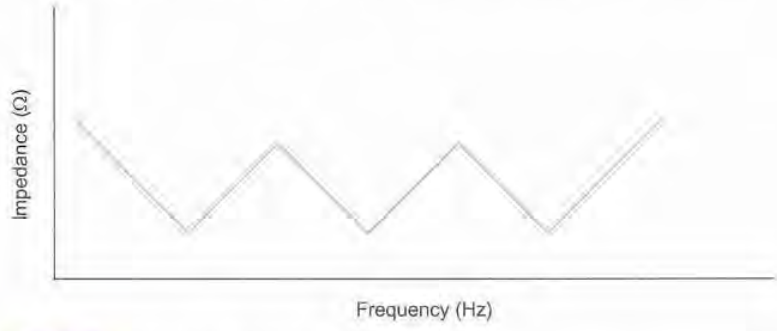


FIG 12.13 Supply impedance vs. frequency

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-

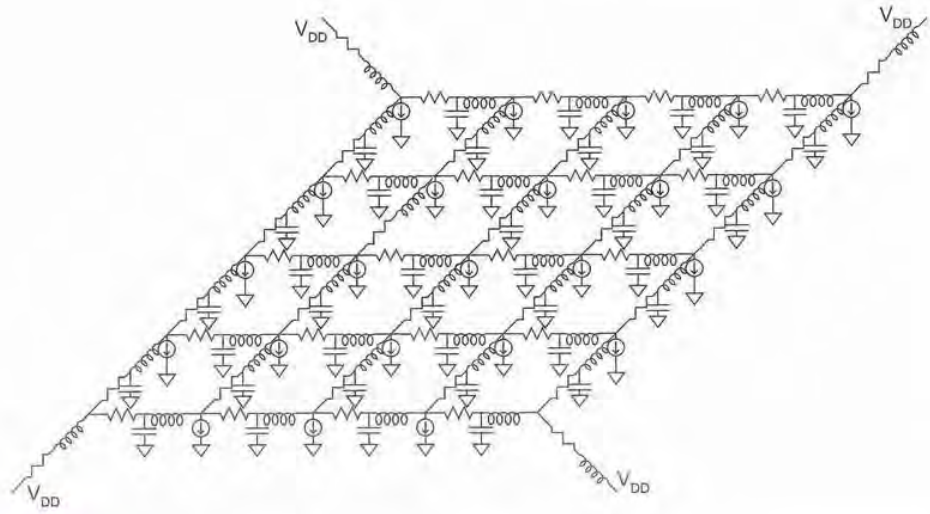


FIG 12.14 Distributed power supply model

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.

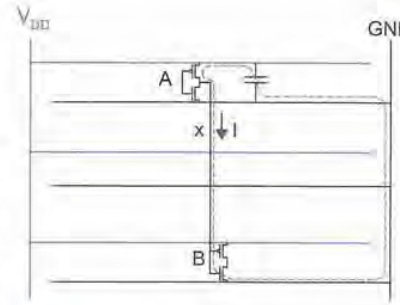


FIG 12.15 Fingered power network with long current return path

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.

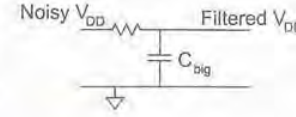


FIG 12.16 Power supply filter

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

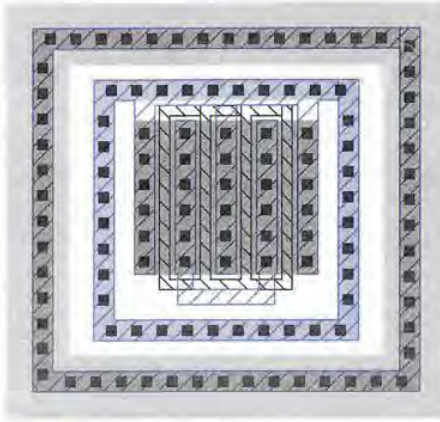


FIG 12.17 Double guard rings around folded nMOS output transistor

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.

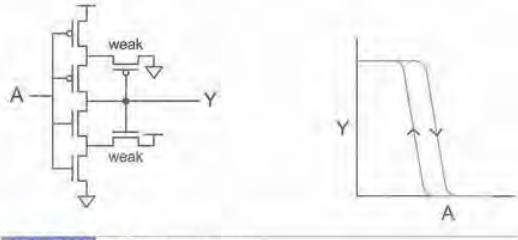


FIG 12.18 Schmitt trigger

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.

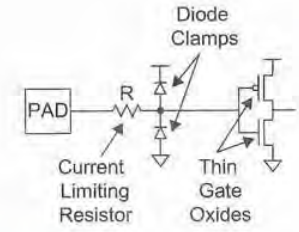


FIG 12.19 Input protection circuitry

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.

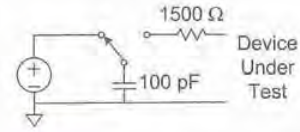


FIG 12.20 Human Body Model ESD test circuit

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.

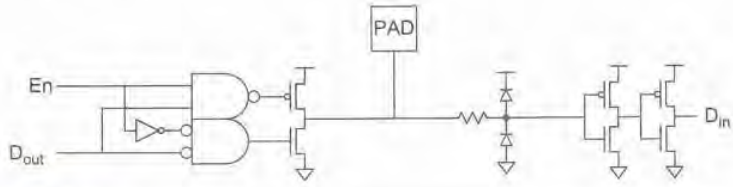


FIG 12.21 Bidirectional pad circuitry

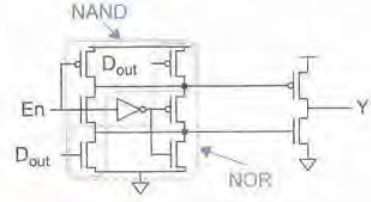


FIG 12.22 Improved tristate buffer

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

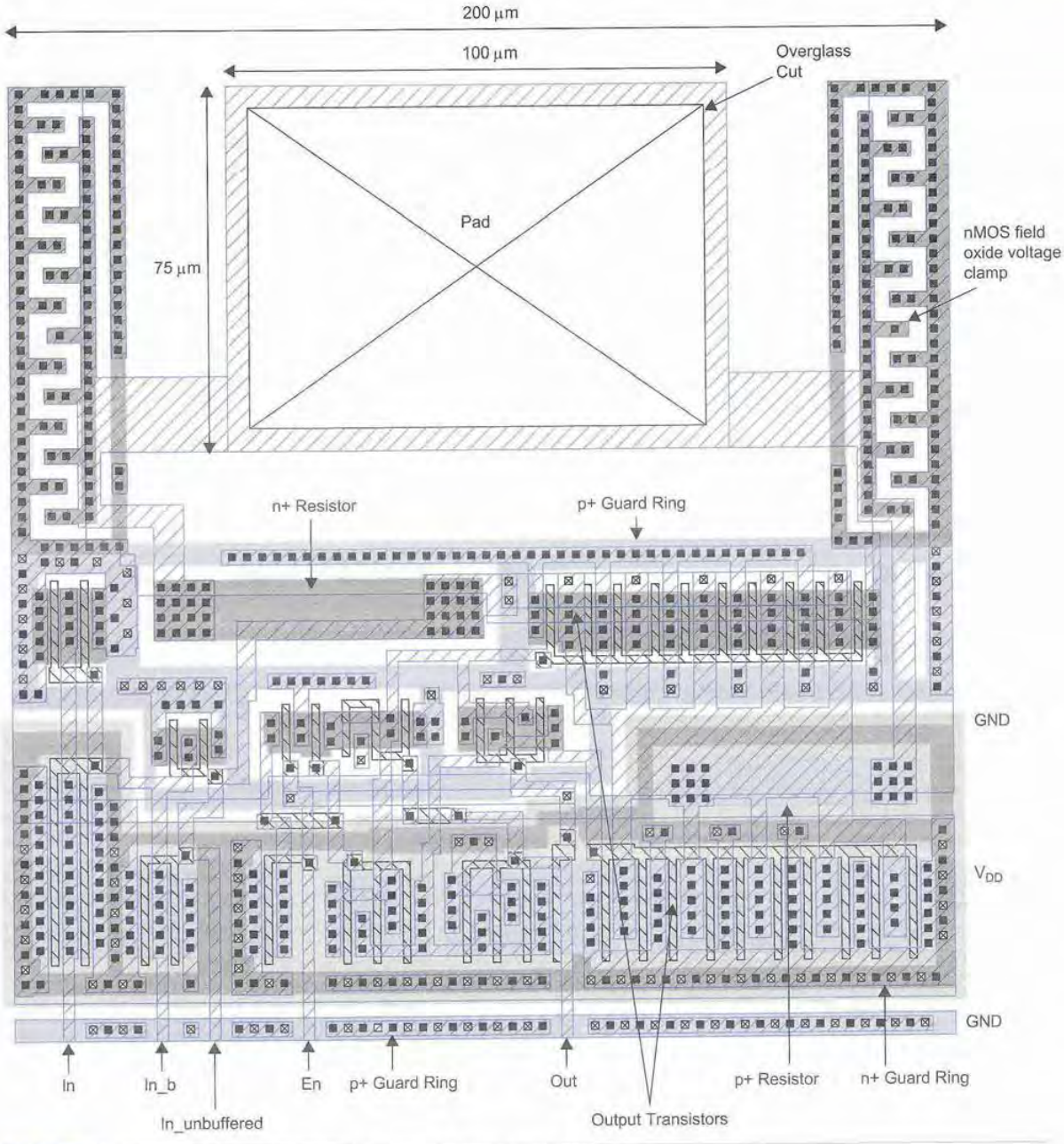


FIG 12.23 MOSIS 1.6 μm bidirectional pad. Color version on inside front cover.

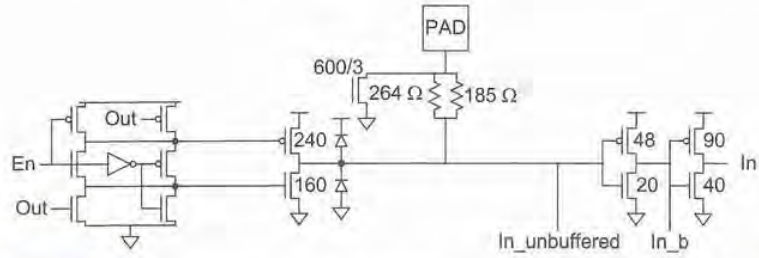


FIG 12.24 MOSIS bidirectional pad schematic

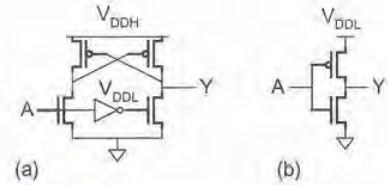


FIG 12.25 Level converters

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:

$$t_{\text{skew}}^{\phi_{1a},\phi_{1b}} = 5 \text{ ps}; \; t_{\text{skew}}^{\phi_{1a},\phi_{2a}} = 15 \text{ ps}; \; t_{\text{skew}}^{\phi_{1b},\phi_{2a}} = 10 \text{ ps}$$

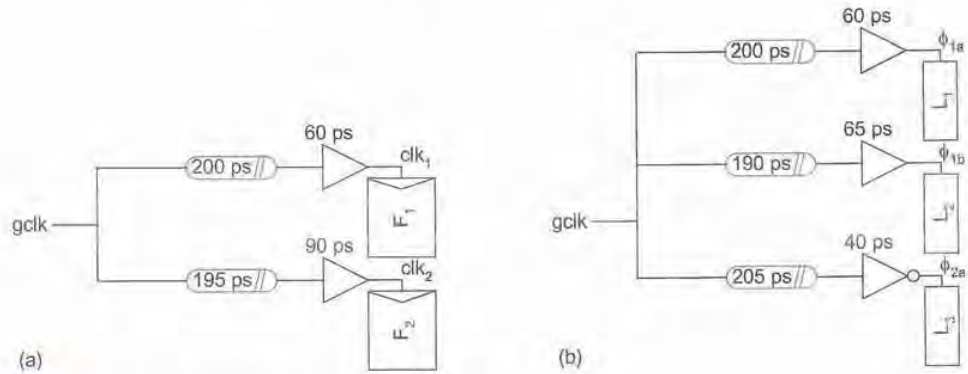


FIG 12.26 Clock skew example

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 (*r*ising/*f*alling) and the number of intervening cycles as well as the physical clock:

$$t_{\text{skew}}^{clk_1,clk_1,(r,r,0)} = 0 \text{ ps}; \; t_{\text{skew}}^{clk_1,clk_1,(r,f,0)} = 30 \text{ ps}; \; t_{\text{skew}}^{clk_1,clk_1,(r,r,1)} = 70 \text{ ps}$$
$$t_{\text{skew}}^{clk_1,clk_2,(r,r,0)} = 0 \text{ ps}; \; t_{\text{skew}}^{clk_1,clk_2,(r,f,0)} = 0 \text{ ps}; \; t_{\text{skew}}^{clk_1,clk_2,(r,r,1)} = 40 \text{ ps}$$