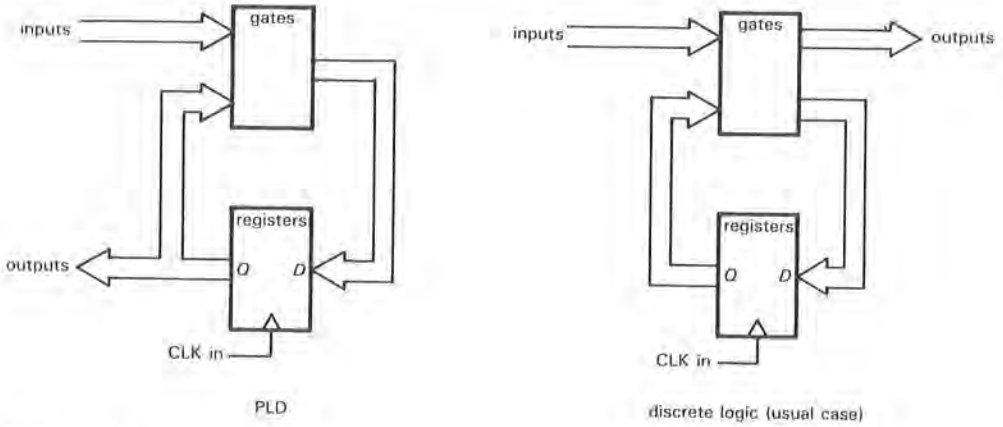


Figure 8.83

standard-function MSI logic types. In fact, just two of the newer GALs (generic array logic) mimic a whole set of PALs, by making their internal architecture (as well as the logic connections) programmable. In particular, the 20-pin GAL16V8 and 24-pin GAL20V8 can each mimic 21 standard PALs. What's more, they can be programmed as mongrel in-between PALs (e.g., an odd number of registers).

PAL loose ends

I/O pins. Three-state output pins that are internally wired as inputs to the AND array can be used as *inputs*. For example, the 16L8 shown in Figure 8.45 has 16 inputs (each true and complemented) to the AND array; 10 of these are dedicated as input pins, and 6 are fed back from three-state outputs. The latter can be converted to "permanent" inputs by disabling the corresponding output (connect a true/complement pair to its AND control); alternatively, those outputs can operate bidirectionally, by enabling the three-state driver according to some logic variables.

"Super-PALs." As we remarked earlier, more flexible programmable logic has followed the original PALs. Notable among these are erasable CMOS variants from Lattice, VTI, Altera, and others.

For example, the "generic array logic" (GAL) from Lattice uses electrically erasable programmable logic, so you can reprogram the chip. Even better, the output structure (called a "macrocell") is itself programmable – each output can be either registered or combinational, true or complemented; there's similar programmable flexibility for the three-state enable line and the feedback line (the latter can come before or after the three-state buffer, or from the adjacent output); see Figure 8.84. The result is that you can mimic any of the common 20-pin PALs with the single GAL16V8 (and any of the common 24-pin PALs with the GAL20V8). This kind of flexibility helps keep your inventory within manageable bounds.

Altera has a line of programmable CMOS logic that can be erased with UV light, just like EPROMs (the IC has a quartz window over the chip). They call their devices EPLDs, for "erasable programmable logic devices." Their smallest chip (the EP320) has output macrocells, and it mimics all 20-pin PALs, just like the GAL16V8. Furthermore, it runs at very low power, unlike the original power-hungry PALs (see below). Finally, Altera makes a number of larger EPLDs, as well

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

as programmable microsequencers, etc. Cypress and VTI also make erasable CMOS programmable logic, complete with macrocells.

Another approach to programmable logic is typified by the Xilinx RAM-configurable gate arrays. These impressive chips contain huge blocks of configurable logic, holding the connection configuration in on-chip RAM (volatile memory). This memory gets loaded from external memory after power has been applied, either by downloading from a microprocessor or by loading itself from nonvolatile ROM-type memory.

Speed and power. The original bipolar transistor PALs introduced by Monolithic Memories (and quickly copied by National and AMD) consumed considerable current – about 200mA for the 16L8/16R8 – and had propagation delays of 40ns. The subsequent "half-power" bipolar PALs were more reasonable, running 90mA with speeds of 35ns. However, the fastest PALs still burn plenty of power; for example, AMD's 16R8D and 16R8-7 have propagation times of 10 and 7.5ns, respectively, but require 180mA (max). The CMOS devices are significantly better: Lattice's "quarter-power" GALs (GAL20V8-15Q) draw 45mA, with a delay time of 15ns, and Altera's EP320-1 delivers 25ns delays with 5mA of current. More important for low-power design, the Altera chips (and AMD's Z-series PALs) can be put into a "zero-power" (10μA typical) standby mode. The designers of future programmable logic will surely continue this healthy trend toward high speed and low power; the days of watt-guzzler programmable logic are behind us!

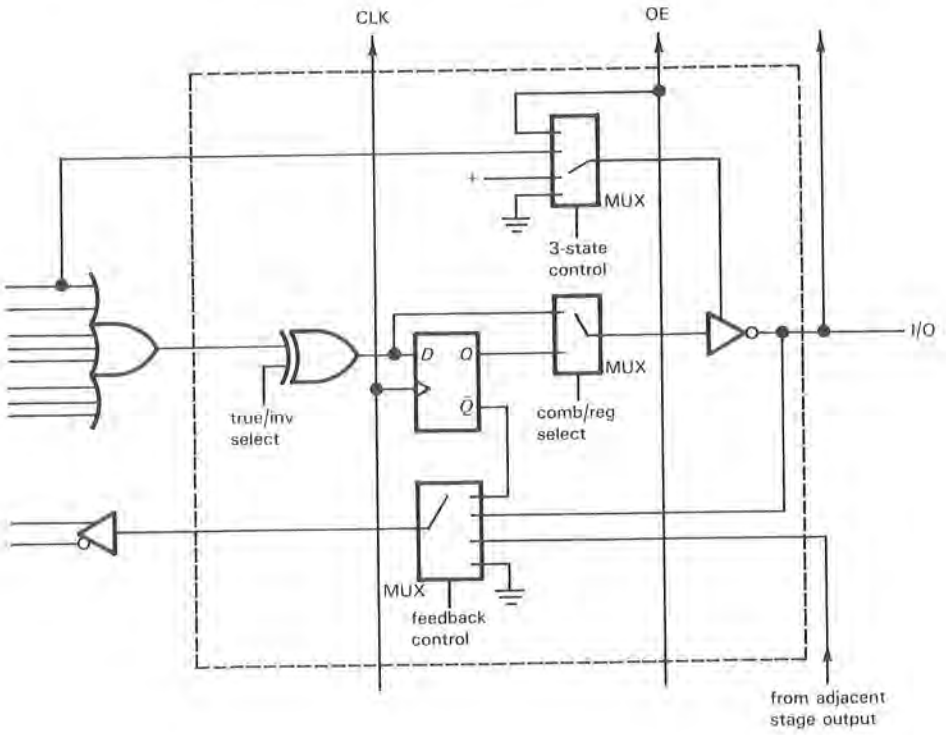


Figure 8.84. GAL programmable-output macrocell.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

Glitches. Programmable logic devices are wonderful things. But you can occasionally get into trouble if you forget about the possibility of logic races. Figure 8.85 shows a 2-input multiplexer implemented in the obvious way; both the PAL diagram and the equivalent circuit are shown. It looks fine, the logic is fine, but the circuit has a flaw: If both data inputs (A and B) are HIGH, and the SELECT line changes state, the output may produce a glitch, as shown. That is because the internal gate delays in S and S' can be unequal, causing a transient state in which both AND gates have one LOW input. The solution in this case (Fig. 8.85B) is to add a redundant term, $A * B$, which you can easily prove to yourself will guarantee no output glitches.

EXERCISE 8.33
Prove that the additional term eliminates all possibility of glitches.

EXERCISE 8.34
What logic terms need to be added to the 4-input multiplexer example (Fig. 8.75) to eliminate glitches?

You can visualize this so-called logic hazard in terms of Karnaugh maps: Diagram the 2-input multiplexer of Figure 8.85A as a Karnaugh map (Fig. 8.85C). Each group in the map is one product term that forms an input to the common OR gate. The OR output is true if any of the product terms is true; but a transition between product groups can produce a glitch if the variables of the starting group are disasserted before the variables of the final group are asserted. The cure (which we used earlier) consists of adding redundant terms to ensure that any possible transition between 1's is included in a single product term; in other words, any 1's that lie in adjacent rows or columns must be enclosed by a product group. This prescription can be cast into a generalized form that applies

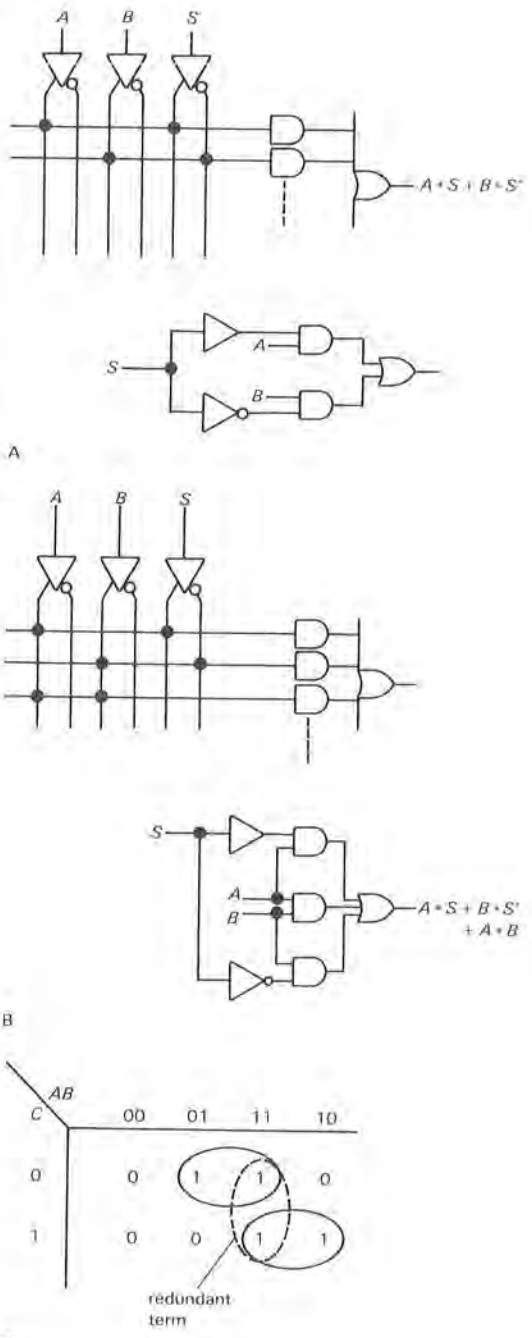


Figure 8.85. PAL glitch elimination.
A. 2-input MUX.
B. Added product term to eliminate glitch.
C. Karnaugh map showing necessary redundant term.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

to the Boolean logic expression directly, rather than the Karnaugh map. That's useful for logic with more than four variables, for which Karnaugh maps become awkward.

The foregoing example is called a *static* glitch, because the output should have remained static. There are also *dynamic* glitches, in which an output that should make a single transition makes multiple transitions instead. It is important to be aware of these internal race conditions when you use programmable logic. In general, design aids like PALASM, ABEL, and CUPL do not attempt to identify such problems. If anything, they usually make things worse, because their logic optimizers work zealously to eliminate just such redundant terms needed to prevent glitches!

8.28 Miscellaneous sequential functions

With the widespread availability of large-scale integration (LSI, equivalent to 1000 gates or more on a chip) and very large scale integration (VLSI), you can get weird and wonderful gadgets all on one chip. This brief section will present just a sampling.

□ *First-in/first-out memory*

A first-in/first-out (FIFO) memory is somewhat akin to a shift register in that data entered at the input appear at the output in the same order. The important difference is that with a shift register the data get "pushed along" as additional data are entered and clocked, but with a FIFO the data "fall through" to the output queue with only a small delay. Input and output are controlled by separate clocks, and the FIFO keeps track of what data have been entered and what data have been removed. A helpful analogy might be a bowling alley, in which black and white bowling balls (bits) are returned to the bowling station: The bits are input by the pin-setting machine, and the time it takes for a ball to roll the length of the alley is the "ripple-through delay time" of the FIFO (typically $1\mu s$–$25\mu s$), whereupon the bits are available at the output to be removed at the whim (asynchronously) of the user.

FIFOs are useful for buffering asynchronous data. The classic application is buffering a keyboard (or other input device, such as magnetic tape) to a computer or sluggish instrument. By this method, no data are lost if the computer isn't ready for each word as it is generated, provided the FIFO isn't allowed to fill up completely. Some typical FIFOs are the 74F433 (TTL, 64 words of 4 bits each, 10MHz, $4\mu s$ fall-through) and the IDT7202 (CMOS, 4096×9, 15MHz, zero fall-through).

A FIFO is unnecessary if the device to which you are sending data can always get it before the next data arrive. In computer language, you must ensure that the maximum *latency* is less than the minimum time between data words. Note that a FIFO will not help if the data recipient is not able, *on the average*, to keep up with the incoming data.

□ *Rate multiplier*

Rate multipliers are used to generate output pulses at a frequency that is related to the clock frequency by a rational fraction. For instance, a 3-decade BCD rate multiplier allows you to generate output frequencies of $nnn/1000$ of the input frequency, where nnn is a 3-digit number specified as three BCD input characters. This isn't the same as a modulo-n counter, since, for instance, you cannot generate an output frequency of 3/10 of the input frequency with a modulo-n divider. One important note: The output pulses generated by a rate multiplier are not, in general, equally spaced. They coincide with input pulses, and therefore they come in funny

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

patterns whose *average* rate is as above. Examples are the '97 (6-bit binary) and the '167 (BCD) rate multipliers.

Frequency counters

Intersil has a good selection of integrated frequency counters. These include provision to gate the input signal for accurately known intervals, up to 8 digits of BCD counter, display drivers, auto-decimal-point and leading-zero blanking, etc. These chips generally require very little external circuitry.

Digital voltmeters

You can get complete digital voltmeters (DVMs) on a single chip. They include analog/digital conversion circuitry and the necessary timing, counting, and display circuitry. Examples are the low-power $3\frac{1}{2}$-digit ICL7136 and the $4\frac{1}{2}$-digit ICL7129; both use LCD 7-segment displays and run from a single 9-volt battery.

□ *Special-purpose circuits*

There are nice collections of LSI chips for arcane jobs like radio communications (e.g., frequency synthesizers), digital signal processing (multiplier/accumulators, digital filters, correlators, arithmetic units), data communications (UARTs, modems, network interfaces, data encryption/decryption ICs, serial format converters), and the like. Often these chips are used in conjunction with microprocessor-based devices, and many of them cannot stand alone.

□ *Consumer chips*

The semiconductor industry loves to develop ICs for use in large-market consumer products. You can get single chips to make digital (or "analog") watches, clocks, locks, calculators, smoke detectors, telephone dialers, music synthesizers, rhythm and accompaniment generators, etc. The guts of radios, TVs, and compact discs are nearly empty these days, thanks to large-scale integration. Speech synthesis (and ultimately speech recognition) has seen a lot of work recently; that's why elevators, Coke machines, automobiles, and even kitchen appliances now speak to us in those sci-fi voices we've all come to love. The development of effective automobile circuits (for engine functions, collision-avoidance systems, etc.) seems to be the next big frontier.

Microprocessors

The most stunning example of the wonders of LSI is the microprocessor, a computer on a chip. At one extreme there are powerful number crunchers like the 68020/30 and 80386/486 (32-bit fast processors with prefetch and cache, large address space, virtual memory, and powerful numeric coprocessors) and chips like the MicroVAX that emulate existing mainframe computers. At the other extreme are single-chip processors with various input, output, and memory functions included on the same chip, for stand-alone use. An example of the latter is the Toshiba TLCS-90 (Fig. 8.86), a CMOS low-power microcontroller with 6 channels of 8-bit A/D converter, internal timers, RAM and ROM, 20 bidirectional digital I/O lines, a serial port, and two stepping-motor ports. This latter type is intended as a dedicated controller in an instrument, rather than as a versatile computation device.

The microprocessor revolution hasn't begun to slow, and we have seen a doubling of computer power and memory size (now 1Mbit per chip, compared with 16kbit/chip at the time the previous edition of this book was written) each year; at the same time, prices have dropped dramatically (Fig. 8.87). Along with bigger and better

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

Figure 8.86. Single-chip microprocessor with lots of on-chip I/O. (Courtesy of Toshiba America, Inc.)

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

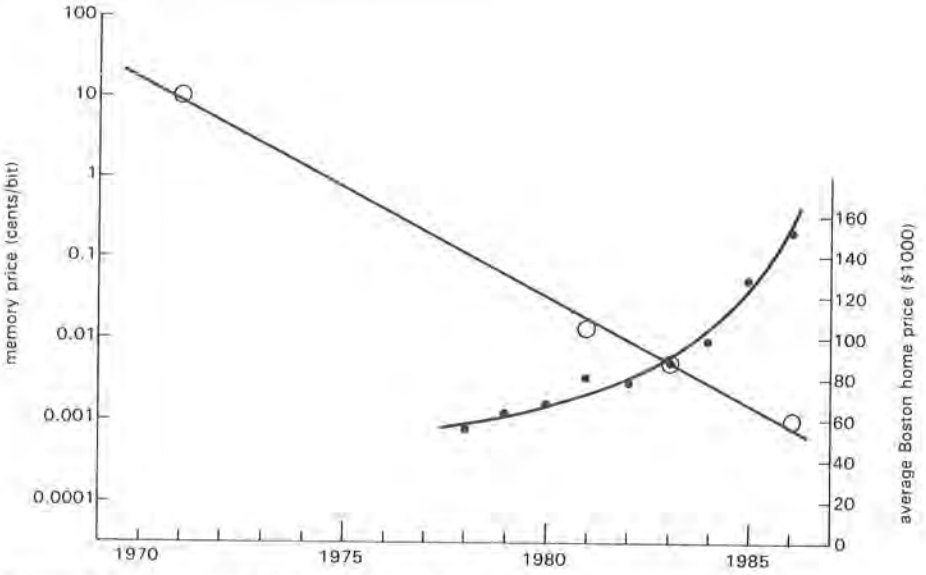


Figure 8.87. The law of Silicon Valley: learning curve.

processors and memory, recent activity in very high speed devices and large parallel architectures promises more excitement in coming years.

SOME TYPICAL DIGITAL CIRCUITS

Thanks to the efforts of the semiconductor industry, digital design is wonderfully easy and pleasant. It's almost never necessary to "breadboard" a digital circuit, as so often is the case with linear design. Generally speaking, the only serious pitfalls involve timing and noise. We'll have more the say about the latter in the next chapter. This is a good place to illustrate timing with some sequential design examples. Some of these functions can be performed with LSI circuits, but the implementations shown are reasonably efficient and illustrate the kind of circuit design being done with what's now available.

8.29 Modulo-n counter: a timing example

The circuit in Figure 8.88 produces one output pulse for every $n+1$ input clock pulses, where n is the 8-bit number you've set on the pair of hexadecimal thumbwheel switches. The '163s are 4-bit synchronous up-counters, with synchronous load (when LD′ is LOW) via the D inputs. The idea is to load the counters with the *complement* of the desired count, then count up to FF_H, reloading at the next clock pulse. Since we've generated the preload levels with pullups to +5 (with the switch common grounded), those levels are negative-true for the displayed switch settings; that makes the preload values, interpreted as positive-true, equal to the 1's complement of the switch settings.

EXERCISE 8.35
Show that the last statement is true, by figuring out the positive-true value that will be loaded for the switch settings in Figure 8.88.

Circuit operation is entirely straightforward. To cascade synchronous counter, you tie all clocks together, then tie a "maximum-count" output of each counter to an enable of the successive counter. For an enabled '163, the RCO (ripple-clock output) goes HIGH at maximum count,

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

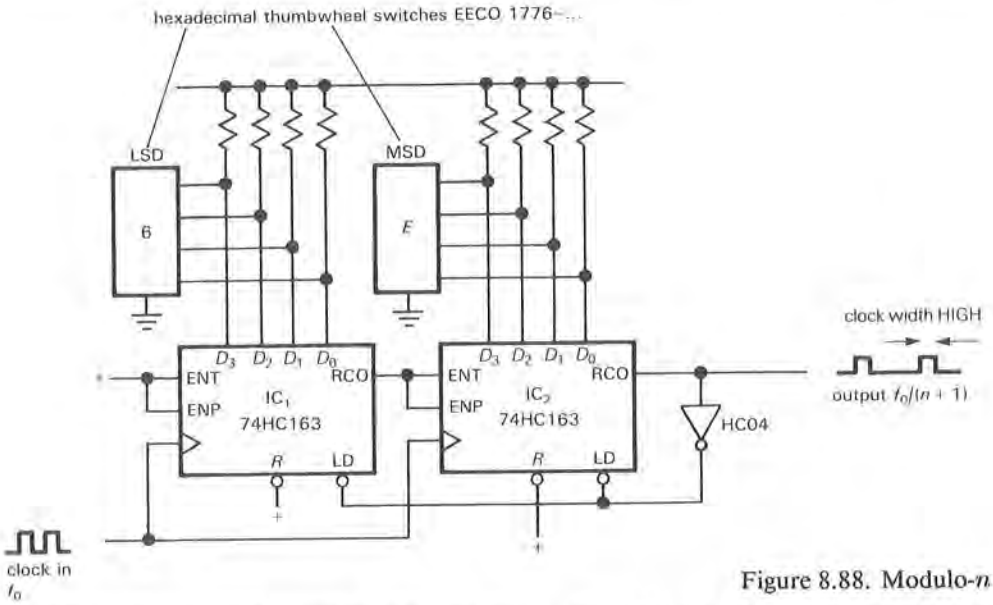


Figure 8.88. Modulo-n counter.

enabling the second counter via the enabling inputs ENT and ENP. Thus IC_1 advances at each clock, and IC_2 advances at the clock after IC_1 reaches F_H. The pair thus counts in binary until the state FF_H, at which point the LD′ input is asserted. This causes synchronous preload at the next clock. In this example we've chosen counters with *synchronous* load in order to avoid the logic race (and runt pulse RCO) that you would get with a *jam*-loaded counter. Unfortunately, this makes the counter divide by $n+1$, rather than n.

EXERCISE 8.36

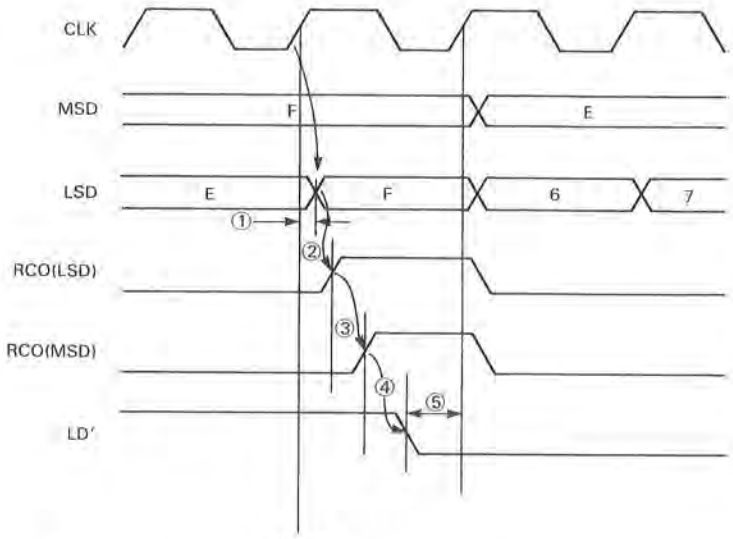
Explain what would happen if jam-load counters (e.g. '191s) were substituted for the synchronous-load '163s. In particular, show how a runt pulse would be created. Demonstrate also that the foregoing circuit divides by $n+1$, whereas the asynchronous-load would divide by n (if it worked at all!).

Timing

How fast can our modulo-n counter count? The 74HC163 specifies a guaranteed f_{max} of 27MHz. However, in our circuit there are additional time delays associated with the cascading connection (IC_2 has to know that IC_1 has reached maximum count in time for the next clock pulse), and also the load-on-overflow connection. To figure the maximum frequency at which the circuit is guaranteed to work, we have to add up the worst-case delays and make sure there is enough setup time remaining. Look at Figure 8.89, where we've drawn a timing diagram showing the load sequence that occurs at maximum count.

A LOW-to-HIGH change on any Q output follows the positive edge of CLK by 34ns max. That's interesting, but not relevant, because the load sequence uses the RCO output; IC_1's RCO follows the rising edge of the CLK pulse that brings it to maximum count by 35ns max, and IC_2's RCO follows its input enable (assuming, of course, that it is at maximum count) by 32ns max. The 74HC04 adds a delay of 19ns max to generate LD′, which must precede CLK (t_{setup}) by 30ns min. That brings us to the next CLK; therefore, $1/f_{max} = (35 + 32 + 19 + 30)$ns, or $f_{max} = 8.6$MHz. This is considerably less than the maximum guaranteed counting frequency of a single 74HC163.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783



		ns, max
①	CLK to *Q*	34
②	CLK to RCO	35
③	ENT to RCO	32
④	*A* to *Y* ('04)	19
⑤	LD' setup to CLK	30(min)

Figure 8.89. Timing diagram and calculation for modulo-n counter.

EXERCISE 8.37

Show by a similar calculation that a pair of synchronously cascaded 74HC163s (without load on overflow) have a maximum guaranteed counting rate of 15.4MHz.

Of course, if greater speed is needed, you can always use faster logic. Doing the same calculation for 74F logic (for which the maximum count rate of a single 74F163 is 100MHz), we find $f_{max} = 29$MHz.

Modulo-n counter devotees should take note of the 'HC40103, an 8-bit synchronous *down*-counter with parallel load (synchronous *or* jam load!), decoded zero-state output, and reset-to-maximum input. It has a close cousin, the 'HC40102, identical except organized as 2-digit BCD.

8.30 Multiplexed LED digital display

This example illustrates the technique of display multiplexing: displaying an n-digit number by displaying successive digits rapidly on successive 7-segment LED displays (of course, the characters need not be numbers, and the displays can have a different organization than the popular 7-segment arrangement). Display multiplexing is done for reasons of economy and simplicity: Displaying each digit continuously requires separate decoders, drivers, and current-limiting resistors for each digit, as well as separate connections from each register to its corresponding decoder (4 lines) and from each driver to its corresponding display (7 wires); it's a mess!

With multiplexing, there's only one decoder/driver and one set of current-limiting resistors. Furthermore, since LED displays come in n-character "sticks" with the corresponding segments of all characters tied together, the number of interconnections is enormously reduced. An 8-digit display requires 15 connections when multiplexed (7 segment inputs, common to all digits, plus one cathode or anode return for each digit), rather than the 57 required for continuous display. An interesting bonus of multiplexing is that the subjective brightness perceived by the eye is greater than if all digits were illuminated

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

continuously with the same average brightness.

Figure 8.90 shows the schematic diagram. The digits to be displayed are resident in register IC_1–IC_4; they could be counters, if the device happened to be a frequency counter, or perhaps a set of latches receiving data from a computer, or possibly the output of an A/D converter, etc. In any case, the technique is to assert each digit successively onto an internal 4-bit "bus" (in this case with 4503 CMOS three-state buffers) and decode and display it while on the bus (4511 BCD-to-7-segment decoder/driver).

In this circuit a pair of inverters is used to form a classic CMOS oscillator operating at about 1kHz, driving a 4022 octal counter/decoder. As each successive output of the counter goes HIGH, it enables one digit onto the bus and simultaneously pulls the corresponding digit's cathode LOW via the high-current open-drain 40107 buffers. The 4022 is rigged up to

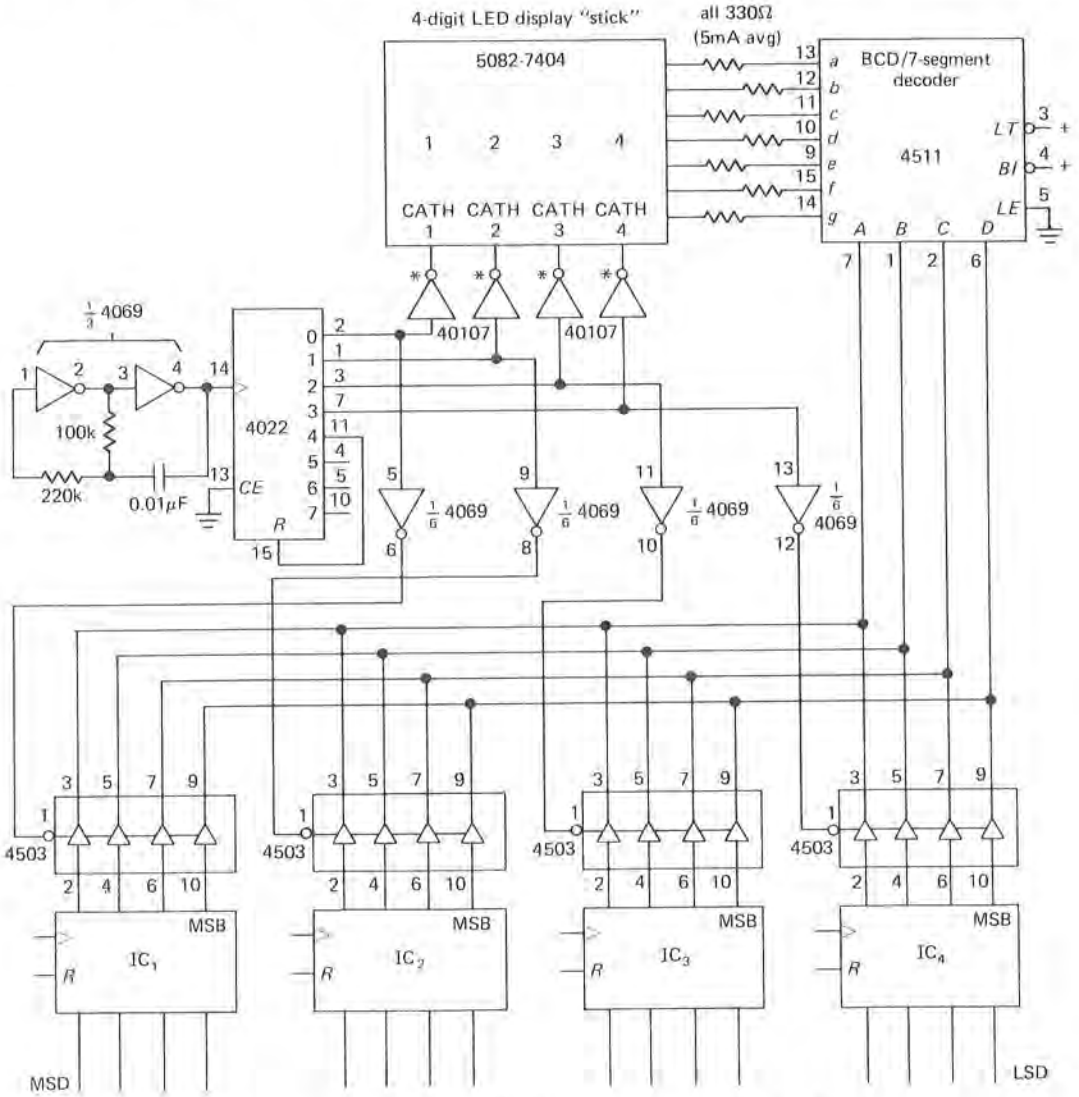


Figure 8.90. Four-digit multiplexed display. Numbers outside symbols are IC pin numbers.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

cycle through the states 0–3 by resetting when the count reaches 4. Display multiplexing will work with greater numbers of digits, and it is universally used in instruments with multidigit LED displays. Try waving a calculator around in front of your eyes – you get numeric alphabet soup!

Many LSI display-oriented chips, such as counters, timers, and watches, include on-chip display multiplexing (and even driver) circuitry. In addition, you can get LSI display controller chips (e.g., the 74C911 and 74C912) for handling the kind of job done earlier with MSI circuits.

□ 8.31 Sidereal telescope drive

The circuit in Figure 8.91 was designed to drive Harvard's 61-inch optical telescope. We needed a 60Hz power source for the equatorial drive motor (1 revolution/day), accurately settable to any frequency near 60Hz (55Hz to 65Hz, say). You wouldn't want exactly 60Hz, for several reasons: (a) stars move at the sidereal rate, not the solar rate, so you would want 60.1643Hz, approximately; (b) starlight gets bent, traveling obliquely through the atmosphere; this "refraction" depends on zenith angle, so the apparent motion is at a slightly different rate; (c) sometimes you want to look at the moon, planets, or comets, which have different rates. The solution here was to use a 5-digit rate multiplier to generate output pulses at a rate $f_{\text{in}}n/10^5$, where n is a 5-digit number set by front-panel BCD thumbwheel switches.

The output is then near 600kHz, since f_{in} is an accurate 1MHz generated by a stable crystal oscillator. The output of the rate multiplier is divided by 10^4 by four decade counters, with the last counter arranged as a divide-by-5 followed by a divide-by-2 for symmetrical square waves at 60Hz. The output is clamped by a zener for stable square-wave amplitude and filtered by a 6-pole Butterworth low-pass filter ($f_0 = 90$Hz) to generate a good sine wave (you can think of the filter as stripping away the higher Fourier components, or "overtones," of the square wave). Then 115 volts ac is generated by the "overcompensated" amplifier illustrated in Section 4.35. The output of the Butterworth looks "perfect" on a scope, as it should, since in this case a 6-pole Butterworth reduces the largest overtone to 1.5% of its unfiltered amplitude; this means that the distortion is more than 35dB down. Note that this technique of sine-wave generation is convenient only if the input frequency is confined to a narrow range.

The ±10% guiding inputs alter the synthesized output frequency 10% by changing the third divider to divide-by-9 or divide-by-11. That stage is a modulo-n divider constructed along the lines of Figure 8.88.

□ 8.32 An n-pulse generator

The n-pulse generator is a useful little test instrument. It generates a burst of n output pulses following an input trigger signal (or you can push a button), with a set of selectable pulse repetition rates. Figure 8.92 shows the circuit. The 'HC40102s are high-speed CMOS 2-decade down-counters, clocked continuously by a selected power-of-10 subdivision of the fixed 10MHz crystal oscillator, but disabled by having both APE (asynchronous preset enable) asserted and CI (carry in) disasserted. When a trigger pulse comes along (note the use of 'HCT logic at this input, for compatibility with bipolar TTL), flip-flop 1 enables the counter, and flip-flop 2 synchronizes counting following the next rising edge of the clock. Pulses are passed by NAND gate 3 until the counter reaches zero, at which time both flip-flops are reset; this parallel-loads the counter to n from the BCD switches, disables counting, and readies the circuit for another trigger. Note that the use of pulldown

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

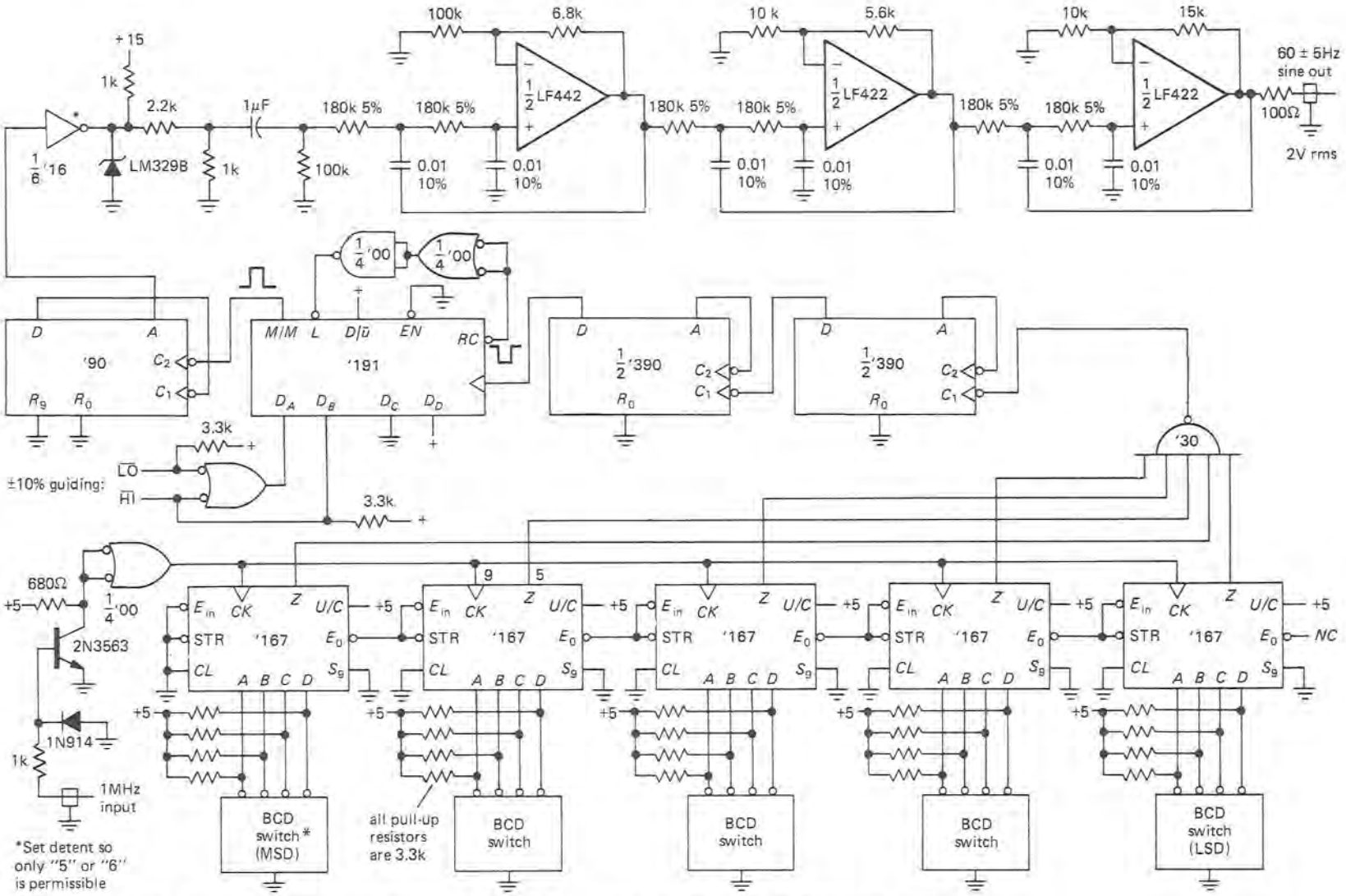


Figure 8.91. Precision 60Hz ac signal source. *Output* frequency = $xx.xxx$; e.g., to generate sidereal rate, set switches to 60165.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

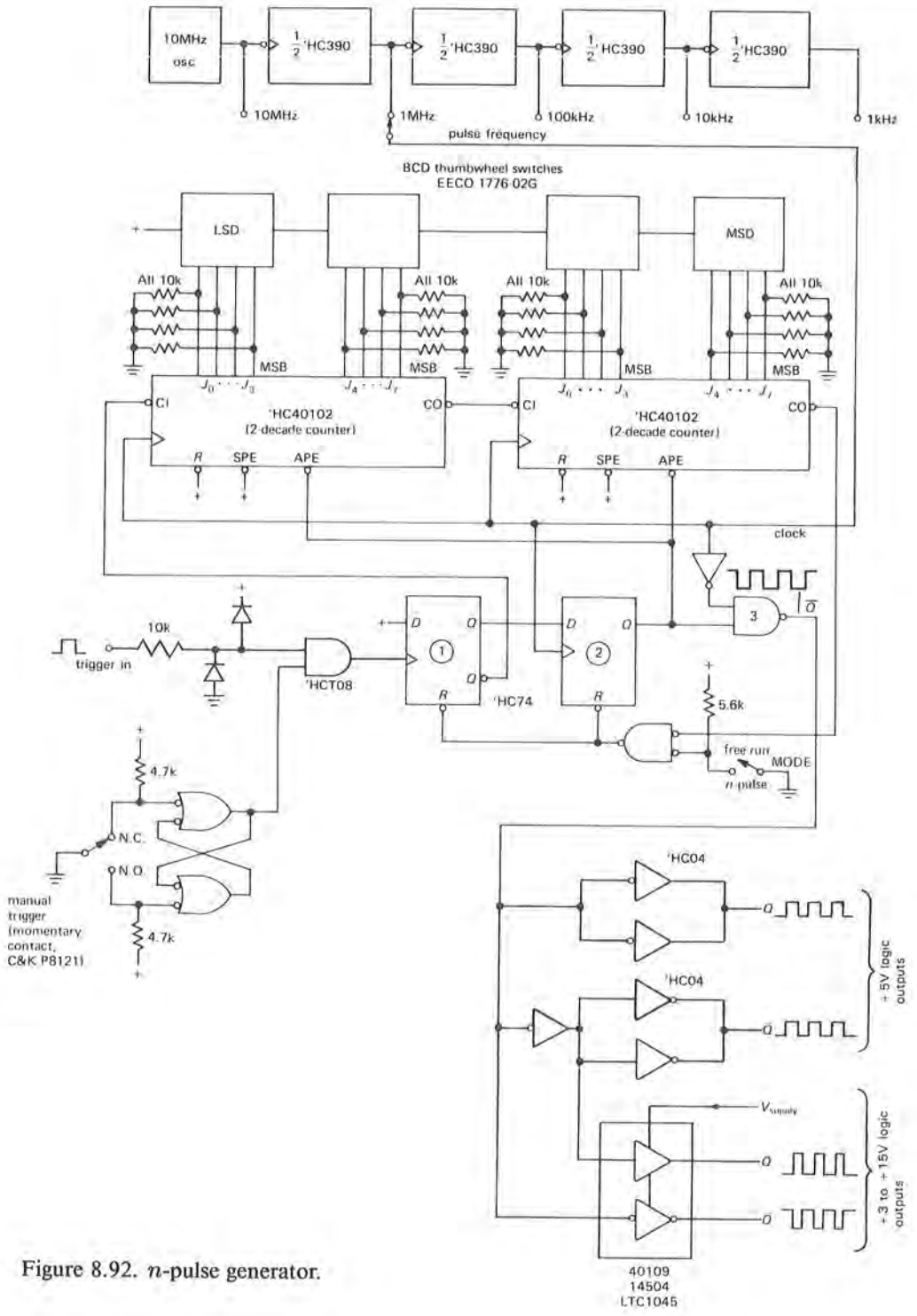


Figure 8.92. n-pulse generator.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

resistors in this circuit means that true (rather than complemented) BCD switches must be used. Note also that the manual trigger input must be debounced, since it clocks a flip-flop. That is not necessary for the free-run/n-pulse switch, which simply enables a continuous stream of output pulses.

The output stage delivers two pairs of true/complement signals. The paralleled 'HC04 inverters give normal +5 volt logic swings, saturating cleanly at the rails because we're using CMOS. We paralleled them to increase drive capability; the circuit as drawn can drive at least ±10mA with logic levels within 0.3 volt of the rails. If you need more output current, you can replace them with 'AC04s, a paralleled pair of which will give you ±50mA with logic levels within 0.3 volt of the rails. We added the boxed driver pair so that you can drive logic that runs on voltages other than +5 volts. For example, low-power design often uses 4000B or 74C series CMOS running directly from 9 volt batteries (it's rated for 3V to 15V operation); 'HC logic accepts 2 to 6 volt supply voltage; and there have been suggestions recently that 'AC-style CMOS be operated at +3.3 volts (JEDEC Standard No. 8). The 40109, 14504, and LTC1045 are all level shifters, chips with active pullups to a second supply terminal connected (in this case) to the driven circuit's V_{DD} line, which can be higher or lower than the pulse generator's +5 volt supply. In that way you get clean CMOS output levels at the right voltage for the circuit under test.

LOGIC PATHOLOGY

There are interesting, and sometimes amusing, pitfalls awaiting the unsuspecting digital logician. Some of these, such as logic races and lockup conditions, can occur regardless of the logic family in use. Others (e.g., "SCR latchup" in CMOS chips) are "genetic abnormalities" of one logic family or another. In the following sections we have collected our bad experiences in the hope that such anecdotes can help others avoid such problems.

8.33 dc problems

Lockup

It is easy to fall into the trap of designing a circuit with a lockup state. Suppose you have some gadget with a number of flip-flops, all going through their proper states. Everything seems to work fine. Then one day it just stops dead. The only way you can get it to work is to turn the power off and back on again. The problem is that there is a lockup state (an excluded state of the system that you can't escape from), and you got into it because of some power-line transient that sent the system into the forbidden state. It is very important to look for such states when you design the circuit and rig up logic so that the circuit recovers automatically. At a minimum, things should be arranged so that a RESET signal (generated manually, at start-up, etc.) brings the system to a good state. This may not require any additional components (e.g., Exercise 8.24).

Start-up clear

A related issue is the state of the system at start-up. It is always a good idea to provide some sort of RESET signal at start-up. Otherwise the system may do weird things when first turned on. Figure 8.93 shows a suitable circuit. The series resistor is necessary with CMOS to prevent damage when power is removed from the circuit, since otherwise the electrolytic capacitor will try to power the system via the CMOS input-gate protection diode. A Schmitt trigger (4093, '14) may be a good idea, to make the RESET signal switch off cleanly. The hysteresis symbol shown in the figure indicates an inverter with Schmitt trigger input, e.g., the TTL 74LS14 (hex inverter) or CMOS 40106 or 74HC14.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

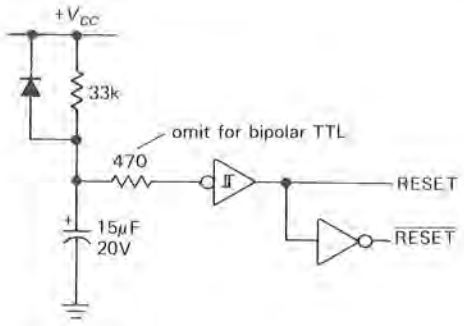


Figure 8.93. Power-on clear circuit.

8.34 Switching problems

Logic races

Lots of subtle traps lurk here. The classic race was illustrated with the pulse synchronizer in Section 8.19. Basically, in any situation where gates are enabled by signals coming from flip-flops (or any clocked device), you must be sure that a gate doesn't get enabled and then disabled a flip-flop delay time later. Likewise, make sure that signals appearing at flip-flop inputs aren't delayed with respect to the clock (another plus for synchronous systems!). In general, delay the clock rather than the data. It is surprisingly easy to overlook a race condition.

□ *Metastable states*

As we remarked earlier, a flip-flop (or any clocked device) can get confused if the data input changes during the setup-time interval preceding a clock pulse. As long as the flip-flop makes *some* decision in this ambiguous case, all is well. However, there is a chance that the input may have changed at just the wrong time, at exactly the "moment of truth," such that the flip-flop can't make up its mind; its output can hover at the logic threshold literally for microseconds (by comparison, normal propagation delays are around 20ns for HC or LS TTL), or (even worse) go into one state, then change its mind and switch back to the other state! (Carver Mead compares the metastable condition with Jean Buridan's paradox – a dog could starve if placed midway between two equal amounts of food.)

This problem does not arise in properly designed synchronous systems, where setup times are always satisfied (by using logic fast enough so that inputs to flip-flops are stable by t_{setup} before the next clocking pulse). However, it can create problems in situations where asynchronous signals (e.g., going from device A, with its own clock, to device B, with a separate clock) must be synchronized. In these cases you cannot guarantee that input transitions do not occur during the setup interval; in fact, you can calculate how often they do! The metastability problem has been blamed for mysterious computer crashes, although we are skeptical. The cure generally involves a set of concatenated synchronizers, or a "metastable state detector" that resets the flip-flop. Awareness of this problem is growing, and there are even "metastable-hardened" logic families, e.g., the AMD 29800 bus-interface series, which claims a maximum metastable delay of 6ns before finally making up its mind.

Clock skew

Clock skew primarily affects slow CMOS logic. The problem arises when you have a clocking signal of slow rise time driving several interconnected devices (Fig. 8.94). In this case two shift registers are clocked by a slowly rising edge, caused by capacitive loading of a relatively high impedance CMOS output (around 500Ω, when operating from +5V). The problem is that the first register may have its threshold at a

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

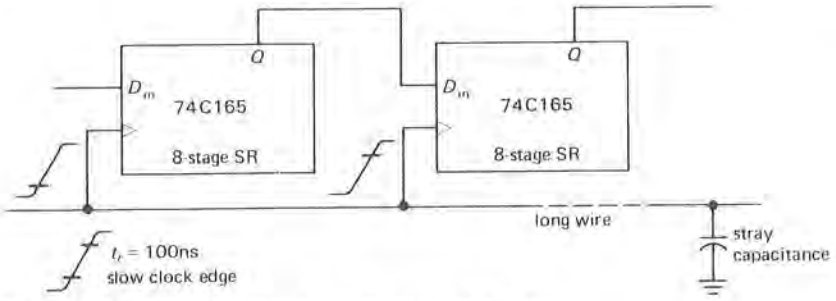


Figure 8.94. Slow rise times cause clocking skew.

lower voltage than that of the second register, and this causes it to shift earlier than the second register. The last bit of the first register is then lost. CMOS devices can display quite a spread of input threshold voltages, which compounds the problem (the threshold is specified only to be between one-third and two-thirds of V_{DD}, and they mean it!). The best cure is to use a nearby chip without much capacitive loading to drive clock inputs in this sort of situation.

Speaking generally, edge-triggered clock inputs on any digital IC should always be treated with respect. For example, clock lines with noise or ringing should always be cleaned up with a gate (perhaps one with input hysteresis) before driving the clocked chip. You're especially likely to have problems with clock lines that come from another board, or from a different family of logic. For example, slow 4000B or 74C logic driving the faster HC or AC families is likely to exhibit problems of clock skew or multiple transitions.

Runt pulses

In Section 8.29 (modulo-n counter) we remarked that some delay should be added if a counter's output clears itself, in order to prevent a pulse of substandard width. The same comment goes for LOAD pulses when using counters or shift registers. Runt pulses will make your life miserable, since you may have marginal operation or intermittent failures. Use the worst-case propagation delay specifications when designing.

Unspecified rules

As the semiconductor industry was finding its way, beginning with the simplest RTL integrated circuits of the 1960s (see Section 9.01 for a brief chronology), then the improved TTL and Schottky families, to the modern high-performance CMOS families, there was an understandable lack of standardization of pinouts, specifications, and functionality. As examples, the 7400 (NAND) had its gates pointing "down," but the 7401 (open-collector NOR) was built with the gates going the other way. This created so much confusion that it had to be mutated into the 7403, which is a 7401 with 7400-style pinouts; a similar disaster happened with the 7490 (BCD ripple counter), with power-supply pins in the middle instead of the corners. (Ironically, mid-chip power-supply pins are making a comeback in fast CMOS, because of reduced inductance and better isolation.)

An important legacy of this early anarchy is the hodge-podge of "unspecified rules" that we're stuck with. For example, the ever-popular '74 D flip-flop exists in every logic family; asserting both SET and CLEAR makes both outputs HIGH in every family except 74C, where it makes both outputs LOW! That's not exactly an unspecified rule, since if you look carefully in the fine print you'll find the inconsistency; the technical term for it is

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

a *gotcha*. Another of our favorite gotchas is the '96, a 5-bit shift register with tricky jam-load inputs: They can SET, but not CLEAR!

A genuine unspecified rule, and in fact a very important one, is "removal time": That's the amount of time you must wait after disasserting a jam-type input before a clocked device is guaranteed to clock properly. Chip designers didn't bother specifying this (though circuit designers always wanted to know it) until the logic families of the early 1980s, specifically the advanced Schottky and fast CMOS families. If you're designing with earlier logic (e.g., 74C), our advice is to be conservative; for example, assume that the removal time is the same as the data setup time [it's usually less; for example, the 74HC74 *D*-type flip-flop specifies a minimum removal time (preset or clear to clock) of 5ns, while the minimum data setup time is 20ns].

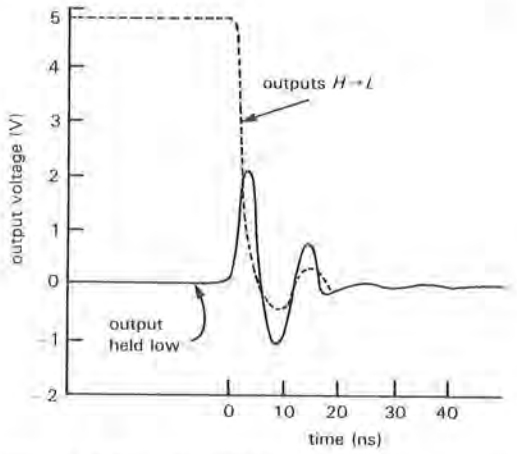


Figure 8.95. 74AC244 octal buffer, driving seven 50pF loads H→ L, holding eighth output LOW. "Ground" is a copper plate (1 oz/ft^2). (After Fig. 1.1-4, TI Advanced CMOS Logic Designer's Handbook.)

8.35 Congenital weaknesses of TTL and CMOS

We will divide this section into nuisance problems and really bizarre behavior.

Nuisance problems

Bipolar TTL. You have to remember that TTL inputs *source* current when held in the LOW state (e.g., 0.25mA for LS, 0.5mA for F). That makes it difficult to use *RC* delays, etc., because of the low impedances necessary, and in general you have to give some thought when interfacing linear levels to TTL inputs.

The TTL threshold (and that of its imitators, HCT and ACT) is too close to ground, making the whole logic family somewhat noise-prone (more on this in Chapter 9). The high speed of these logic families makes them recognize short spikes on the ground line; those spikes, in turn, are generated by the fast output transition speeds, making the problem worse.

Bipolar TTL makes demands on the power supply (+5V, ±5%, with relatively high quiescent power dissipation). Power-supply current spikes generated by the active pullup output circuitry generally require liberal use of power-supply bypassing, ideally one 0.1μF capacitor per chip (Fig. 8.96).

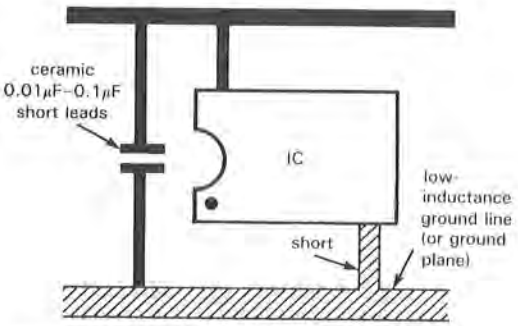


Figure 8.96. It's always a good idea to use robust low-inductance ground wiring, with liberal use of bypass capacitors.

CMOS. CMOS inputs are prone to damage from static electricity. The mortality rate really climbs in winter! Recent families with polysilicon gates [HC(T), AC(T)] are much more rugged than their metal-gate ancestors. CMOS inputs show a large

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

spread in logic threshold, which can lead to problems of clock skew (Section 8.34), particularly with the slow CMOS families (4000B, 74C) owing to their high output impedance (200–500Ω). These slow families can even exhibit double output transitions when driven with slowly rising inputs. CMOS requires *all* unused inputs, even those of unused gate sections, to be connected to HIGH or LOW.

An interesting congenital problem with the newer fast CMOS families, particularly AC and ACT, is the presence of "ground bounce": A fast CMOS chip driving its capacitive load generates enormous transient ground currents, causing the chip's ground line to jump momentarily, and thereby carrying with it LOW outputs that happen to be innocent bystanders on the same chip. Figure 8.95 shows the sort of thing you see. Notice particularly the magnitude of the effect – 1 to 2 volts is not uncommon! When you consider that a 3ns, 5 volt transition into 50pF amounts to a transient current $I = C\,dV/dt = 83$mA, and that an octal buffer might drive eight such loads simultaneously (total current of $\frac{2}{3}$A!), this behavior isn't surprising. This problem is turning out to be harder to solve than anyone imagined, leading to a controversial new set of AC/ACT circuits with "center-pin" power and grounds (for lower inductance). At the time of writing, the major players have taken sides, with TI championing the new pinouts, and RCA and Fairchild defending the conventional corner-pin layout. We have received handsomely bound polemics from both sides, pointing out the inferiority of both the opposing arguments and the opposing chips. At the very least, users should be aware of this serious problem and take measures to keep ground inductance as low as possible when using AC/ACT. It's best to use circuit boards with a dedicated ground plane and plenty of low-inductance bypass capacitors. Better still, if you don't need the speed, stay away from AC/ACT entirely, and use HC/HCT instead.

Bizarre behavior

Bipolar TTL. TTL doesn't do many really weird things. However, some TTL monostables will trigger on a glitch on the supply (or ground) line, and they generally behave somewhat fidgety. A circuit that works well with LS TTL may malfunction when replaced with AS TTL, because of faster edge times and consequently larger ground-line currents and ringing (74F TTL seems better in this regard). Most weird TTL operation can be traced to noise problems.

CMOS. CMOS can drive you crazy! For example, the chip can go into "SCR latchup" if the input (or output) is driven beyond the supply momentarily. The resultant current (50mA or so) through the input-protection diodes turns on a pair of parasitic cross-connected transistors that are a side effect of the "junction-isolated" CMOS process (see Fig. 3.50 and Section 14.16). This effectively shorts V_{DD} to ground; the chip gets hot, and you have to turn off the power supply before it will behave itself again. If you let this happen for more than a few seconds, you'll have to replace the chip. Some of the newer CMOS designs (the polysilicon-gate HC processes from GE/RCA and National) claim to be immune to latchup even with input swings *5 volts* beyond the rails and to operate properly for input swings 1.5 volts beyond the rails.

CMOS has some strange and subtle failure modes. One of the output FETs can open up, giving pattern-sensitive failures that are difficult to detect. An input may begin to sink or source current. Or the whole chip may start drawing substantial supply current. Putting a 10 ohm resistor in series with each chip's V_{DD} lead makes it easy to locate faulty CMOS chips that are drawing quiescent supply current (for chips driving many outputs, or power drivers such as the AC series, use 1Ω sense resistors).

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

Besides the input threshold variation between chips, a single chip can exhibit different thresholds for several functions driven from a single input. For example, the RESET input of a 4013 can bring Q' HIGH before it brings Q LOW. This means that you should not terminate a reset pulse based on the output at Q', since the runt pulse that will be generated may actually fail to clear the flip-flop.

Open inputs on CMOS chips are bad news! You might have a circuit that intermittently misbehaves. You put you scope probe onto a point in the circuit, and it shows zero volts, as it should. Then the circuit works fine for a few minutes – before malfunctioning again! What happened was that the scope discharged the open input, and it took a long time to charge back up to the logic threshold.

Here's the craziest of them all: You forgot to wire up the V_{DD} pin on a CMOS chip, but the circuit works just fine! That's because it is being powered by one of its logic inputs (via the input-protection diodes from the input to the V_{DD} line in the chip). You might get away with this for a long time, but suddenly the circuit reaches a state where all the logic inputs to the chip are simultaneously LOW; the chip loses its power and forgets its state. Of course, this is a bad situation anyway, since the output stage isn't adequately powered and can't source much current. The trouble is that this situation may produce symptoms only occasionally, and it can have you running around in circles until you figure out what's going on.

SELF-EXPLANATORY CIRCUITS

8.36 Circuit ideas

Figure 8.97 shows some digital circuit ideas.

8.37 Bad circuits

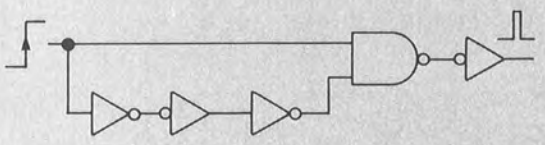
Figure 8.98 shows some classic digital circuit blunders.

ADDITIONAL EXERCISES

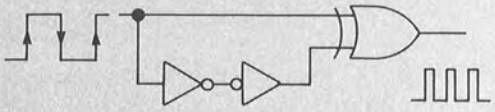
(1) Show how to make a JK flip-flop using a type D flip-flop and a 4-input multiplexer. Hint: Use the address inputs for J and K.
(2) Design a circuit that reads out, on 7-segment digits, how many milliseconds you've held a button down. The device should be smart enough to reset itself each time. Use a 1.0MHz oscillator.
(3) Design a reaction timer. "A" pushes his button; an LED goes on, and a counter begins counting. When "B" pushes her button, the light goes out and an LED display reads the time, in milliseconds. Be sure to design the circuit so that it will function properly even if A's button is still held down when B's button is pushed.
(4) Design a period counter: a device to measure the number of microseconds in one period of an input waveform. Use a Schmitt comparator to generate TTL; use a 1.0MHz clock frequency. Make it work so that pushing a button initiates the next measurement.
(5) Add latches to the period counter, if you haven't already.
(6) Now make it measure the time for 10 periods. Also, have it light an LED while it's counting.
(7) Design a true electronic stopwatch. Button A starts and stops the count. Button B resets the count. The output should be of the form $xx.x$ (seconds and tenths); assume that you have a 1.0MHz square wave.
(8) Some stopwatches use a single button (start, stop, reset, start, etc., each time it is pushed). Design an electronic equivalent.
(9) Design a nice frequency counter to measure the number of cycles per second of an input waveform. Include lots of

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

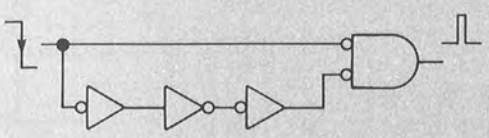
Circuit ideas



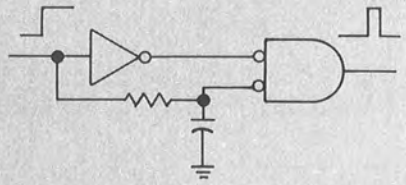
A. rising edge pulse generator



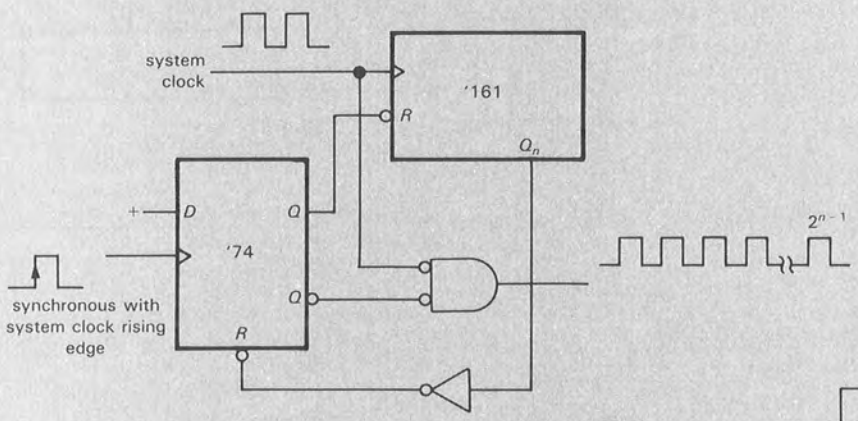
E. both-edge pulse generator



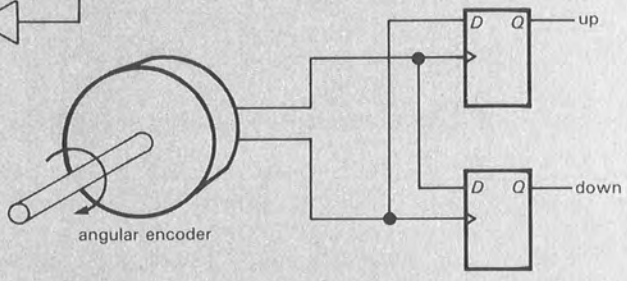
B. falling-edge pulse generator



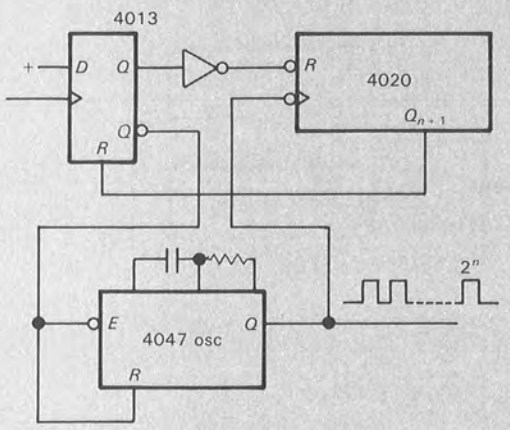
F. adjustable rising-edge pulse generator



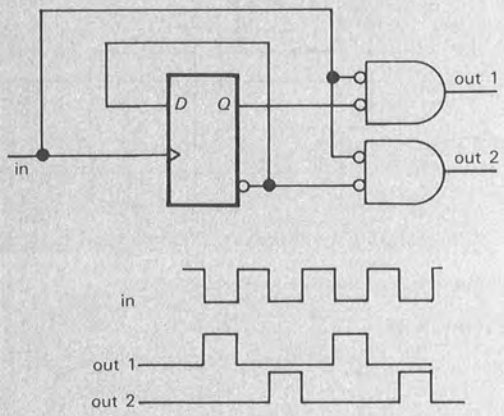
C. synchronous 2^n pulse generator



G. quadrature to up-down-counter conversion; use to obtain rotary position from angular encoder



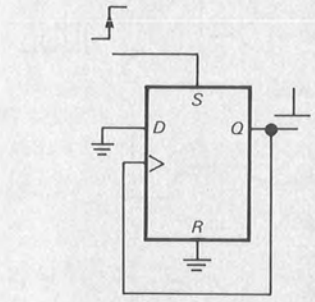
D. asynchronous 2^n pulse generator



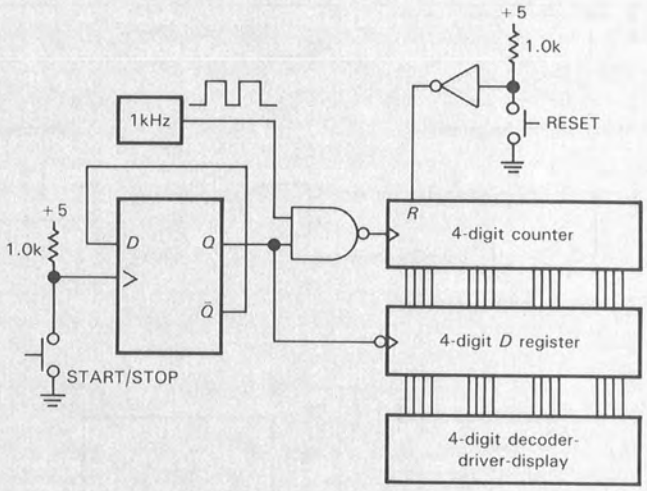
H. quadrature clock pulse generator

Figure 8.97

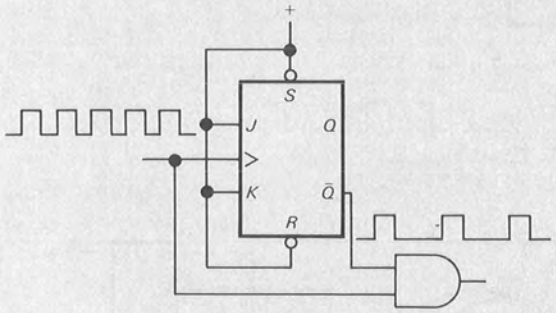
Petitioner Intel Corp., Ex. 1037
IPR2023-00783



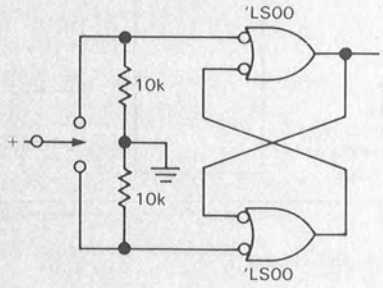
A. short pulse generator



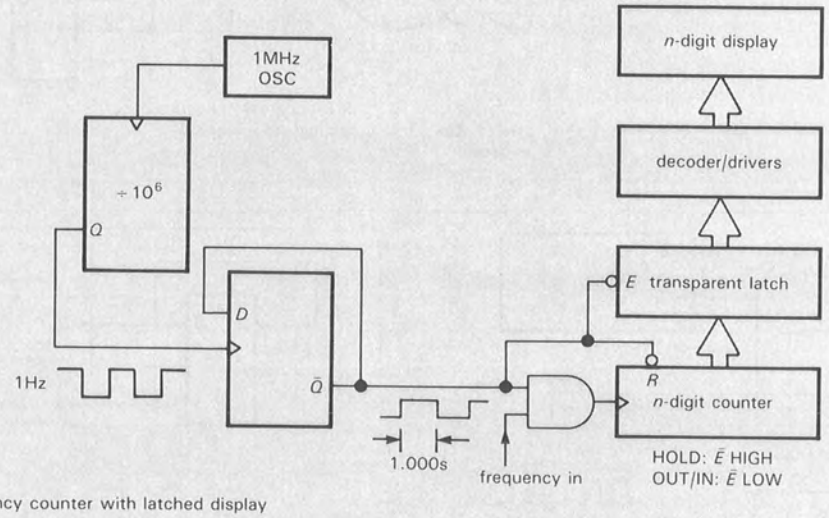
B. stopwatch (with single START/STOP button)



C. circuit to remove every other pulse from an input train of 1μs pulses (subtle error)



D. switch debouncer



E. frequency counter with latched display

Figure 8.98

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

digits, latched count while counting the next interval, and choice of 1 second, 0.1 second, or 0.01 second counting interval. You might add a good input circuit with several sensitivities, a Schmitt trigger with adjustable hysteresis and trigger point (use a fast comparator), and a logic signal input for TTL signals. How about a BCD output? Multiplex the digits on output, as well as parallel output? Spend some time on this one.

(10) Design a circuit, using HC logic, to time a speeding bullet. The projectile breaks a thin wire stretched across its path; then, some measured distance farther along its path, it breaks a second wire. Beware of problems like "contact bounce." Assume that you have a 10MHz logic square wave, and design your circuit to read out, in microseconds (4 digits), the time interval between breaking the two wires. A push button should reset the circuit for the next shot.

(11) Make a 1-of-16 decoder from two 74HC42s (1-of-10). The input is a 4-bit binary number. Output will be negative-true (as with the 74HC42). Hint: MSB input to the 74HC42 can be used as an "ENABLE."

(12) Imagine that you have four 256-bit ROMs, TTL style, each of which has an 8-bit parallel address input, a three-state positive-true output, and an output enable (negative-true); i.e., the ROM asserts the selected data bit at its output if the enable is LOW. Show how to "expand" these into a 1024-bit ROM, using whatever else you need. (A 74HC138 might be handy, or you can do it with gates. Try both.)

(13) Invent a circuit to keep a running sum of successive 4-bit numbers that are input to it. Keep your result to only 4 bits (i.e., perform a sum modulo-16). (Such a sum is useful as a "check sum" to be written on inherently error-prone data-recording media, e.g., punched paper tape.) Assume that a positive 1μs TTL pulse occurs once during the time that each input number is valid. Provide a reset input. Thus, your circuit is as shown in Figure 8.99.

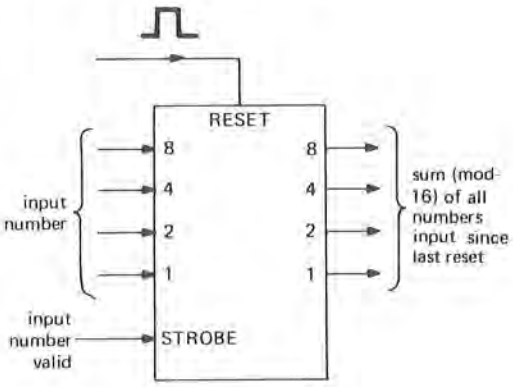
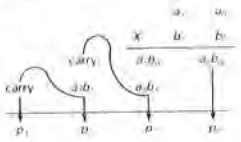


Figure 8.99

Now add another feature to the circuit, namely an output bit that is 1 if the *total number* of 1 bits in all the input numbers (since the last reset input) is odd, 0 if even. Hint: An XOR "parity tree" will tell you if the sum of 1's in each number is odd; figure it out from there.

(14) In Exercise 8.14 you designed a 2×2 multiplier by using a Karnaugh map for each output bit. In this problem you are to accomplish the same task by the process of "shift and add." Begin by writing out the product the way you would in elementary school:



This process has a simple repeating pattern, requiring 2-input gates (what kind?) to generate the intermediate terms $a_0 b_0$, etc., and 1-bit "half adders" (adders that have a carry out but no carry in) to sum the intermediate terms.

(15) Now design a 4×4 multiplier along the same lines, this time using three 4-bit full adders (74HC83) and sixteen 2-input gates.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

TABLE 8.4. BUFFERS

Type		Bits	Pins	Output drive[a]		Family	Enable inputs[b]	Output[c]	Broadside[d]	Comments
true	inv			sink (mA)	source (mA)					
'125	–	4	14	24	2.6	LS	4 L	3S	–	each enable 1 bit
'126	–	4	14	24	2.6	LS	4 H	3S	–	each enable 1 bit
'365	'366	6	16	24	2.6	LS	2 L	3S	–	
'367	'368	6	16	24	2.6	LS	2 L	3S	–	enable 2,4 bits
'1034	'1004	6	14	24	15	LS	–	2S	–	74AS is 48/48 mA
'230	–	8	20	64	15	AS	2 L	3S	–	4 true, 4 inv; each enable 4 bits
–	'231	8	20	64	15	AS	L,H	3S	–	each enable 4 bits
'241	–	8	20	24	15	LS	L,H	3S	–	each enable 4 bits
'244	'240	8	20	24	15	LS	2 L	3S	–	each enable 4 bits
'465	'466	8	20	24	2.6	LS	2 L	3S	–	
'467	'468	8	20	24	2.6	LS	2 L	3S	–	each enable 4 bits
'541	'540	8	20	24	15	LS	2 L	3S	•	
'656	'655	8	24	64	15	F	3 L	3S	•	+ parity gen
'2966	'2965	8	20	12	1		2 L	3S	–	25Ω R_{out}; each enable 4 bits
'827	'828	10	24	64	15	F	2 L	3S	•	Am29827/8

[a] output drive capability for members of the indicated family. [b] ENABLE inputs, HIGH or LOW true.
[c] 2S = 2-state; 3S = 3-state. [d] broadside (flow-through) pinout.

TABLE 8.5. TRANSCEIVERS

Type		Bits	Pins	Registers	Output drive[a]		Enables		Broadside[c]	Comments[d]
true	inv				sink (mA)	source (mA)	pol	mode[b]		
'243	'242	4	14	–	24	15	L,H	LR	•	
'245	–	8	20	–	24	15	L	DE	•	
'543	'544	8	24	2			L	LR	•	
'545	–	8	20	–			L	DE	•	
'550	'551	8	28	2			L	LR	–	Am2950/1
'552	–	8	28	2			L	LR	–	+ parity gen
'588	–	8	20	–			L	DE	•	IEEE-488 term resistors
'623	'620	8	20	–	24	15	L,H	LR	•	'621/2 are OC
'639	'638	8	20	–	24	15	L	DE	•	3-state in one dir, OC in other
'643	–	8	20	–	24	15	L	DE	•	Q and Q' outputs; '644 is OC
'645	'640	8	20	–	24	15	L	DE	•	'641/2 are OC
'646	'648	8	24	2	24	15	L	DE	•	'647/9 are OC
'652	'651	8	24	2	24	15	L,H	LR	•	'654/3 are 3S one dir, OC other
'657	–	8	24	–			L,H	DE	•	+ parity gen
'2623	'2620	8	20	–	12	2	L,H	LR	•	25Ω output resistors
'2645	'2640	8	20	–	12	2	L	DE	•	25Ω output resistors
2952	'2953	8	24	2	24	6.5	L	LR	•	'2950/1 have handshake

[a] for LS-family devices. [b] DE - DIRection and ENable inputs; LR - separate Left and Right enables.
[c] broadside (flow-through) pinout. [d] all are 3-state, unless stated otherwise.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

TABLE 8.6. DECODERS

Type	Bits[a]	Pins	E/E'[b]	Outputs qty[a]	Outputs pol[c]	Output enables[a]	Comments
'42	4	16	0/0	10	L	–	'156 is OC
'131	3	16	1/1	8	L	–	
'137	3	16	1/1	8	L	–	input D flip-flop
'138	3	16	1/2	8	L	–	
'139	2+2	16	0/1+1	4+4	L	–	
'155	2	16	0+1/2+1	4+4	L	–	input latch
'538	3	20	2/2	8	H	2	'537 is 1 of 10
'539	2+2	20	0/1+1	4+4	H	1+1	
'547	3	20	2/1	8	L	–	input latch; ACK output when enabled
'548	3	20	2/2	8	L	–	ACK output when enabled

(a) multiple sections are indicated by "*X+Y*," where *X* and *Y* are the number of bits in each section
(b) "*H/L*" indicates the number of active-H/L enable inputs; for multiple sections see note (a).
(c) H = active HIGH; L = active LOW.

TABLE 8.7. MAGNITUDE COMPARATORS

Type	Bits	Pins	Pullup	Enable	Latch	Outputs Q	Q'	<	>	Comments
'85	4	16	–	•	–	•	–	•	•	
'518	8	20	Q	•	–	•	–	–	–	OC; '519 lacks pullups
'520	8	20	Q	•	–	–	•	–	–	'521 lacks pullups; '522 is OC
'524	8	20	–	•	•	•	–	•	•	latch is ser/par load SR with 3S outputs
'682	8	20	Q	–	–	–	•	–	•	'683 has OC
'684	8	20	–	–	–	–	•	–	•	'685 has OC
'686	8	24	–	•	–	–	•	–	•	'687 has OC
'688	8	20	–	•	–	–	•	–	–	'689 has OC
'866	8	28	–	•	•	•	–	•	•	P,Q output latches; Q reset; logical or arith comp
'885	8	24	–	–	P	–	–	•	•	input latch for P; logical or arith comp

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

TABLE 8.8. MONOSTABLE MULTIVIBRATORS

Type	# sect	Pins	Trig logic	Reset	Retrig	Int R[a]	PRD[b]	Comments
'121	1	14	(L+L)•H	–	–	•	–	precision
'221	2	16	L•H	L	–	–	Y	precision
'122	1	14	(L+L)•H•H	L	•	•	Y	
'422	1	14	(L+L)•H•H	L	•	•	N	'122 w/o trigger on clear
'123	2	16	L•H	L	•	–	Y	
'423	2	16	L•H	L	•	–	N	'123 w/o trigger on clear
'4098	2	16	L+H	L	•	–		
'4538	2	16	L+H	L	•	–	N	'4528
'9601	1	14	(L+L)•H•H	–	•	–	–	
'9602	2	16	L+H	L	•	–	N	

[a] internal timing resistor. [b] generates pulse on RESET disassert, if triggering logic is asserted.

TABLE 8.9. D-TYPE REGISTERS AND LATCHES[a]

Type											
true		inverted									
D-type	latch	D-type	latch	Bits	Pins	Enable	Reset[b]	Broadside[c]	Q/Q'[d]	Output[e]	Comments
'173	–	–	–	4	16	-	A	•	-	3S	
'175	'375	'175	'375	4	16	-	A	-	•	2S	
'298	–	–	–	4	16	-	-	•	-	2S	MUXed input
'379	–	'379	–	4	16	•	-	-	•	2S	sim to '175, but CLR→EN
'398	–	'398	–	4	20	-	-	-	•	2S	MUXed input
'399	–	–	–	4	16	-	-	-	-	2S	MUXed input
'174	–	–	–	6	16	-	A	-	-	2S	
'378	–	–	–	6	16	•	-	-	-	2S	sim to '174, but CLR→EN
'273	–	–	–	8	20	-	A	-	-	2S	
'374	'373	'534	'533	8	20	-	-	-	-	3S	
'377	–	–	–	8	20	•	-	-	-	2S	sim to '374, but 3S→EN
–	'412	–	'432	8	24	•	A	-	-	3S	Intel 8212; status bit
'574	'573	'564	'563	8	20	-	-	•	-	3S	broadside '374/3; also '576/'580
'575	–	'577	–	8	24	-	S	•	-	3S	'574 with synch CLR
'825	'845	'826	'846	8	24	•	A	•	-	3S	Am29825
'823	'843	'824	'844	9	24	•	A	•	-	3S	Am29823
'821	'841	'822	'842	10	24	-	-	•	-	3S	Am29821; 10-bit '374
'396	–	–	–	4+4	16	-	-	-	-	2S	cascaded 4-bit reg; 8-bit out
'874	'873	'876	'880	4+4	24	-	A	•	-	3S	
'878	–	'879	–	4+4	24	-	S	•	-	3S	'874 with synch CLR
–	'604	–	–	8+8	28	-	-	-	-	2S	MUXed input; '605 is OC
–	'606	–	–	8+8	28	-	-	-	-	2S	glitch-free '604; '607 is OC

[a] see also "TRANSCEIVERS," some of which have latches. [b] A – asynchronous; S – synchronous. [c] broadside (flow-through) pinout. [d] both true and complemented outputs. [e] 2S – push-pull (2-state) output; 3S – 3-state output.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

TABLE 8.10. COUNTERS

Type		Bits	Pins	Clock[a]	Load[a]	Reset[a]	U/D[b]	Direct/ latch[c]	Output[d]	Comments
binary	BCD									
'93	'90	4	14	A	-	A	-	D	2S	non-st'd Vcc, gnd; '92 is modulo-12
'161	'160	4	16	S	S	A	-	D	2S	
'163	'162	4	16	S	S	S	-	D	2S	
'169	'168	4	16	S	S	-	•	D	2S	
'191	'190	4	16	S	A	-	•	D	2S	
'193	'192	4	16	S	A	-	•	D	2S	separate U/D clock inputs
'197	'196	4	14	A	A	A	-	D	2S	
'293	'290	4	14	A	-	A	-	D	2S	'93 with st'd Vcc, gnd
'561	'560	4	20	S	B	B	-	D	3S	
'569	'568	4	20	S	S	B	•	D	3S	25LS2569/8
'669	'668	4	16	S	S	-	•	D	2S	improved '169
'691	'690	4	20	S	S	A	-	B	3S	
'693	'692	4	20	S	S	S	-	B	3S	
'697	'696	4	20	S	S	A	•	B	3S	
'699	'698	4	20	S	S	S	•	B	3S	
'4516	'4510	4	16	S	A	A	•	D	2S	
–	'4017	5	16	S	-	A	-	D	2S	decoded 1-of-10 outputs
'4024	–	7	14	A	-	A	-	D	2S	
'69	'68	8	16	A	-	A	-	D	2S	
'269	–	8	24	S	S	-	•	D	2S	skinny-DIP
'393	'390	8	14/16	A	-	A	-	D	2S	dual '93/'90
'461	–	8	24	S	S	S	-	D	3S	PAL
'469	–	8	24	S	S	-	•	D	3S	PAL
'579	–	8	20	S	S	B	•	D	3S	8 bidirectional in/out lines
'590	–	8	16	S	-	A	-	L	3S	
'591	–	8	16	S	-	A	-	L	OC	
'592	–	8	16	S	A	A	-	L	2S	8 inputs, 1 output (MAX CNT)
'593	–	8	16	S	A	A	-	L	3S	8 bidirectional in/out lines
'779	–	8	16	S	S	-	•	D	3S	8 bidirectional in/out lines
'867	–	8	24	S	S	A	•	D	2S	skinny-DIP
'869	–	8	24	S	S	S	•	D	2S	skinny-DIP
'4520	'4518	8	16	S	-	A	-	D	2S	pos or neg edge clk
'40103	'40102	8	16	S	B	A	D	D	2S	
'4040	–	12	16	A	-	A	-	D	2S	
'4020	–	14	16	A	-	A	-	D	2S	
'4060	–	14	16	A	-	A	-	D	2S	

[a] A - asynchronous; all A clock inputs count on neg edge. S - synchronous; all S clock inputs count on positive edge. B - both. [b] D - count down only. [c] B - both. [d] 2S - 2-state (totem-pole); 3S - 3-state.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

TABLE 8.11. SHIFT REGISTERS

Type	Bits	Pins	Serial/parallel input	Serial/parallel output	Direction	Latches[a]	Reset[b]	Output[c]	Comments[d]
'95	4	14	P/S	P	R	-	-	2S	
'194	4	16	P/S	P	R/L	-	A	2S	
'195	4	16	P/JK	P	R	-	A	2S	
'295	4	14	P/S	P	R	-	-	3S	
'395	4	16	P/S	P/S	R	-	A	3S	
'671	4	20	P/S	P	R/L	O	A	3S	output MUXed: SR or latch; reset SR only
'672	4	20	P/S	P	R/L	O	S	3S	'671 with synch CLR
'96	5	16	P/S	P	R	-	A	2S	high load only
'91	8	14	2S	2S	R	-	-	2S	
'164	8	14	2S	P	R	-	A	2S	
'165	8	16	P/S	2S	R	-	-	2S	
'198	8	24	P/S	P	R/L	-	A	2S	
'299	8	20	P/S	P/S	R/L	-	A	3S	common I/O pins
'322	8	20	P/S	P/S	R	-	A	3S	common parallel I/O
'323	8	20	P/S	P/S	R/L	-	S	3S	'299 with synch reset
'589	8	16	P/S	S	R	I	-	3S	power-up clear (SR only)
'594	8	16	S	P/S	R	O	2A	2S	'599 is O/C; separate resets
'595	8	16	S	P/S	R	O	A	3S	'596 is O/C; reset SR only
'597	8	16	P/S	S	R	I	A	2S	reset SR only
'598	8	20	P/2S	P/S	R	I	A	3S	common parallel I/O; reset SR only
'673	16	24	S	P/S	R	O	A	2S	common serial 3S I/O; reset latch only; CS, R/W
'674	16	24	P/S	S	R	-	-	3S	common serial 3S I/O; CS, R/W
'675	16	24	S	P/S	R	O	-	2S	CS, R/W
'676	16	24	P	S	R	-	-	2S	CS

[a] O - at output; I - at input. [b] A - asynchronous; S - synchronous. [c] 2S - 2-state (totem pole); 3S - 3-state.
[d] CS - chip select input; R/W - read/write input.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

DIGITAL MEETS ANALOG

CHAPTER 9

Although sheer "number crunching" is an important application of digital electronics, the real power of digital techniques is seen when digital methods are applied to analog (or "linear") signals and processes. In this chapter we will begin with a brief chronology of the rise and fall of digital logic families and a review of the input and output properties of the surviving TTL and CMOS families that you are likely to use in circuit design. This is essential to understand how to interface logic families to each other and to digital input devices (switches, keyboards, comparators, etc.) and output devices (indicator lamps, relays, etc.). We will also look at n-channel MOS logic, since it is widely used in LSI functions. We will continue with the important subject of bringing digital signals on and off circuit boards, in and out of instruments, and through cables. Then we will discuss the major subject of conversion between analog and digital signals. Finally, with an understanding of these techniques, we will look at a number of applications in which combined analog and digital techniques provide powerful solutions to interesting problems.

CMOS AND TTL LOGIC INTERFACING

□ 9.01 Logic family chronology

In the prehistoric early 1960s, adventurous people who didn't want to build their logic from discrete transistors struggled with RTL (resistor-transistor logic), a simple logic family introduced by Fairchild and characterized by poor fanout and poor noise immunity. Figure 9.1 shows the problem, namely a logic threshold at one V_{BE} above ground, and miserable fanout (in some cases one output could drive only one input!) caused by passive pullup and a low-impedance current-sinking load. Those were the days of small integration, and the most complicated function you could get was a dual flip-flop, which would operate to 4MHz. We bravely built circuits with RTL; they sometimes malfunctioned when you switched on a soldering iron in the same room.

The death knell for RTL came a few years later with the introduction by Signetics of DTL (diode-transistor logic) and, soon thereafter, Sylvania's SUHL – "Sylvania Universal High Speed Logic" –

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

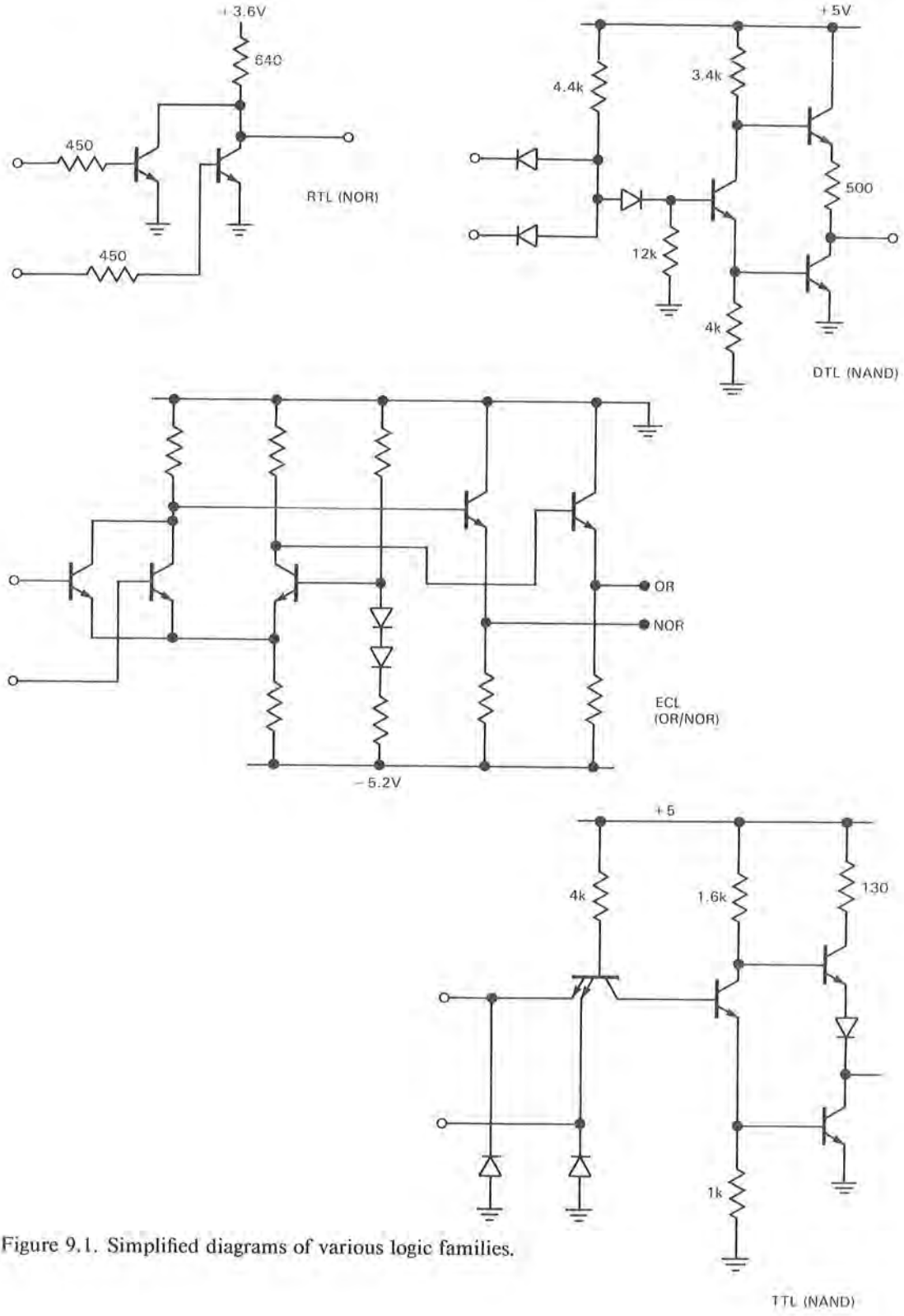
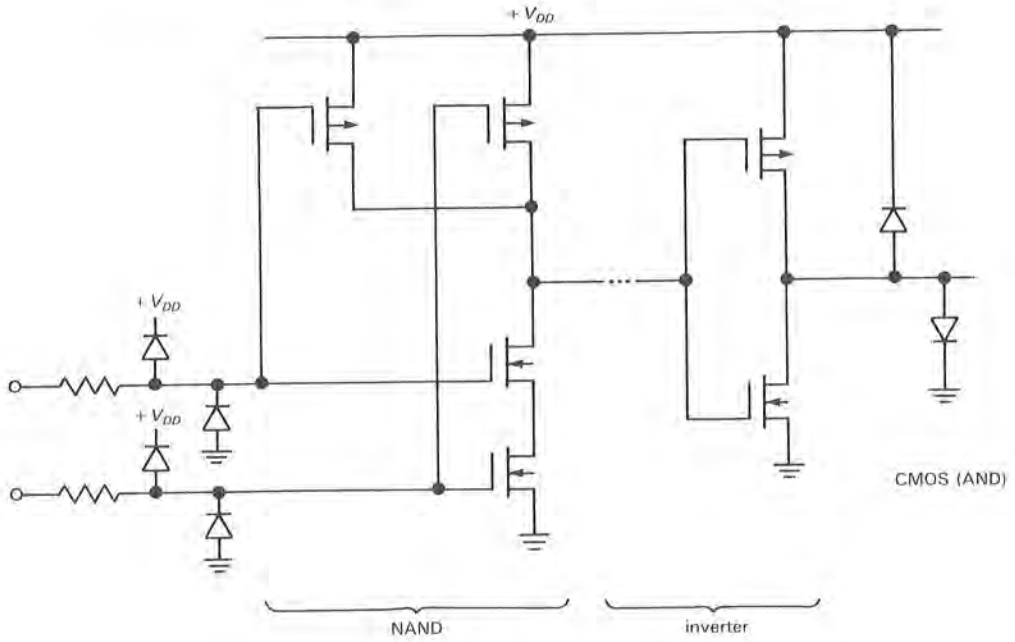


Figure 9.1. Simplified diagrams of various logic families.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783



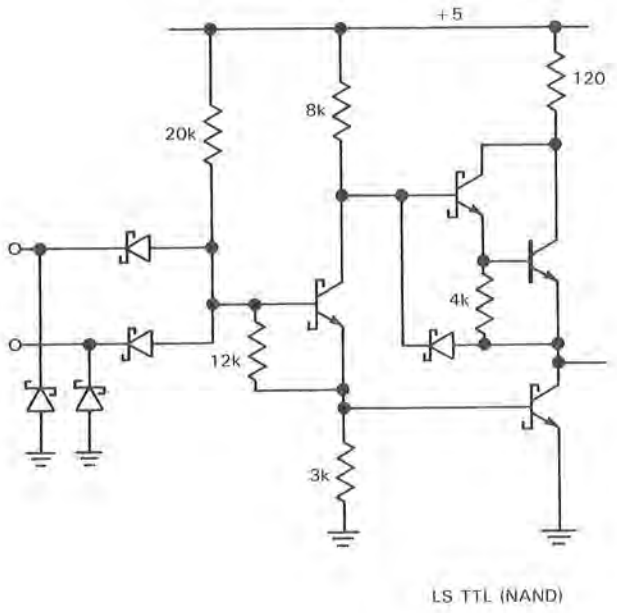


Figure 9.1 (*cont*).

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

which is now called TTL (transistor-transistor logic). Signetics had a popular mixture of the two, called 8000-series DCL Utilogic ("Designer's Choice Logic"). TTL caught on quickly, particularly in the "74xx" numbering system originated by Texas Instruments. These families used current-*sourcing* inputs with logic threshold at two V_{BE}s and (usually) push-pull "totem-pole" outputs (Fig. 9.1). DTL and TTL began the era of +5 volt logic (RTL used +3.6V), and offered speeds of 25MHz and fanouts of 10 (i.e., an output could drive 10 inputs). Designers rejoiced at the speed, reliability, and complex functions (divide-by-10 counters, for example) of these families. It seemed to us that you couldn't ask for more; TTL would live forever.

People are greedy, though. They wanted more speed. They wanted less power consumption. They soon got both, sort of. In the high-speed arena, a souped-up TTL (74H series; "High-speed" TTL) delivered roughly twice the speed, at twice the power! (It accomplished this underwhelming feat by cutting all resistor values in half.) Another family, ECL (emitter-coupled logic), delivered real speed (30MHz in its original version), using a negative power supply and rather closely spaced logic levels (−0.9V and −1.75V); it consumed lots of power (30mW/gate) and only came in small integration. For low power there was a souped-down TTL (74L series; "Low-power" TTL) with 1/4 the speed at 1/10 the power of the corresponding 7400 "standard" TTL.

Back at RCA the first of the MOSFET logic families was developed, 4000-series CMOS. It had zero quiescent power consumption and wide supply range (+3V to +12V). The outputs swung rail-to-rail, and the inputs drew no current. That was the good news. The bad news was the speed (1MHz at 10V supply) and the price (about $20 for a package of 4 gates). In spite of the price, a whole generation of battery-powered instrument designers grew up on the micropower CMOS, simply because there was no alternative. They learned the true meaning of static electricity as they worked with the easily damaged inputs.

This, then, was the situation at the beginning of the 1970s – two main lines of bipolar logic (TTL and ECL) and the extraordinary CMOS. The TTL variants were essentially compatible, except that 74L TTL had feeble output drive (3.6mA sink) and could drive only two standard (74-series) TTL loads (whose inputs sourced 1.6mA when held LOW). There was almost no compatibility between the major families (though a pulled-up TTL could drive CMOS, and 5V CMOS could just barely drive a single 74L TTL load).

During the 1970s there were steady improvements on all fronts. TTL sprouted the non-saturating "Schottky-clamped" (Section 13.23) families: first the 74S (Schottky) series, which made 74H obsolete by giving three times the speed at twice the power; then the 74LS ("Low-power Schottky"), which made the "standard" 74-series TTL obsolete by delivering slightly improved speed at 1/5 the power. Life with 74LS and 74S was nice, but then Fairchild came up with 74F (F for FAST: "Fairchild Advanced Schottky TTL"), which is up to 50% faster than 74S at 1/3 the power; it also has other improved properties that make it extremely nice to design with. Texas Instruments, the originators of many 74xx lines, came up with a pair of advanced Schottky families, 74AS ("Advanced Schottky") and 74ALS ("Advanced Low-power Schottky"); the former is intended to replace 74S, and the latter is intended to replace 74LS. All these TTL families have the same logic levels and plenty of output drive, and so they can be mixed within a circuit. Table 9.1 and Figure 9.2 compare the speed and power of these families.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

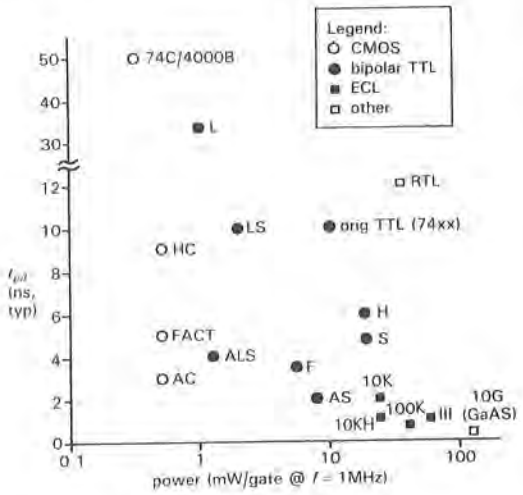


Figure 9.2. Speed versus power for various logic families.

Meanwhile, the 4000-series CMOS evolved into the improved 4000B series, with wider supply range (3V to 18V), better input protection, and higher speed (3.5MHz at 5V). The 74C series is essentially the same, with 74-family functions and pinouts to take advantage of the tremendous success of the 74-family bipolar logic. ECL sprouted the ECL II, ECL III, ECL 10,000, and ECL 100,000 series, with speeds to 500MHz.

The situation in 1980, then, was this: Most design was done with 74LS, with 74F (or 74AS) mixed in where higher speed was needed. This same TTL was used as "glue" to hold together NMOS microprocessor circuits, whose inputs and outputs are TTL-compatible. Micropower design was always done with 4000B or 74C CMOS, equivalent and compatible with each other. And for the highest speeds (100MHz–500MHz), ECL was used. There wasn't much mixing between families, except occasional combination of CMOS and TTL, or perhaps TTL interfaces to an ECL high-speed circuit.

During the 1980s came the remarkable development of CMOS logic with the speed and output drive of TTL: first 74HC ("High-speed CMOS"), with the same speed as 74LS, and, of course, zero quiescent current; then 74AC ("Advanced CMOS"), with the same speed as 74F or 74AS. With rail-to-rail output swings, and input threshold at half the supply voltage, this logic combines the best features of previous TTL and CMOS and should gradually replace bipolar TTL. There is an incompatibility, though, since the logic HIGH output of TTL or NMOS (2.4V min) is insufficient to drive the HC and AC input. Since there is likely to be a period in which you want to use some of the older bipolar TTL or NMOS, each CMOS family is offered in a variant with the lower input threshold. These are named 74HCT and 74ACT ("High-speed CMOS with TTL threshold"). Don't be tempted to use them universally – the CMOS-threshold devices (HC, AC) have better noise immunity and are the families of choice. During the 1980s, also, LSI and VLSI (microprocessors, memory, etc.) have been gradually switching over from NMOS to CMOS (with consequent low power and CMOS compatibility), at the same time increasing speed and complexity. And at the extreme high-speed end, GaAs (gallium arsenide) devices are delivering speeds of several gigahertz.

Note that all CMOS families (which includes 4000B, 74C, HC, HCT, AC, and ACT) have the pleasant characteristic of zero "static" (i.e., nothing happening) power dissipation, with typical quiescent currents less than a microamp. But CMOS does draw "dynamic" current when logic levels are switching, because of the combined effects of (a) transient rail-to-rail conduction of internal push-pull pairs during the middle of the logic swing and (b) dynamic current needed to charge and discharge internal and load capacitances. Dynamic supply currents are proportional to switching frequency and can rival bipolar logic as you reach maximum operating frequencies. See Section 8.10 (Fig. 8.18)

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

TABLE 9.1. LOGIC FAMILIES

Family	t_{pd}('00) (C_L=50pF) typ (ns)	t_{pd}('00) (C_L=50pF) max (ns)	f_{clk}('74) max (MHz)	P_{diss} (C_L=0) @1MHz (mW/gate)	I_{OL} @0.5V max (mA)	I_{IL} max (mA)	V_{th} typ (V)	V_{supply} min (V)	V_{supply} nom (V)	V_{supply} max (V)	Date of intro
CMOS											
AC	3	5.1	125	0.5	24	0	$V_+/2$	2	5 or 3.3	6	1985
ACT							1.4	4.5	5	5.5	
HC	9	18	30	0.5	8	0	$V_+/2$	2	5	6	1982
HCT							1.4	4.5	5	5.5	
4000B/74C @10V	30	60	5	1.2	1.3	0	$V_+/2$	3	5-15	18	1970
4000B/74C @5V	50	90	2	0.3	0.5	0					
TTL											
AS	2	4.5	105	8	20	0.5	1.5	4.5	5	5.5	1980
F	3.5	5	100	5.4	20	0.6	1.6	4.75	5	5.25	1979
ALS	4	11	34	1.3	8	0.1	1.4	4.5	5	5.5	1980
LS	10	15	25	2	8	0.4	1.1	4.75	5	5.25	1976
ECL											
ECL III	1.0	1.5	500	60	–	–	-1.3	-5.19	-5.2	-5.21	1968
ECL 100K	0.75	1.0	350	40	–	–	-1.32	-4.2	-4.5	-5.2	1981
ECL 100KH	1.0	1.5	250	25	–	–	-1.29	-4.9	-5.2	-5.5	1981
ECL 10K	2.0	2.9	125	25	–	–	-1.29	-5.19	-5.2	-5.21	1971
GaAs											
10G	0.3	0.32	2700	125	–	–	-1.3	-3.3	-3.4	-3.5	1986
								-5.1	-5.2	-5.5	

and Section 14.16 (Fig. 14.38) for more detail.

We wrap up our brief history with a recommendation: Use 74HC logic for nearly all your new designs, mixing in (a) 74HCT for compatibility with existing NMOS or TTL and (b) 74AC(T) for speed. Bipolar TTL (74LS/ALS and 74F/AS) are OK to use, but probably are outclassed by CMOS. Choose the older 4000B/74C where the extended supply voltage range is needed and speed is unimportant (e.g., a portable device powered by an unregulated 9V battery).

9.02 Input and output characteristics

Digital logic families are designed so that the output from a chip can properly drive many inputs within the same logic family. A typical fanout capability is 10 loads, meaning that an output from a gate or flip-flop, for example, can be connected to 10 inputs and still perform within specifications. In other words, in normal digital design practice you can get by without knowing anything about the electrical properties of the chips you're using, as long as your circuit consists only of digital

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

logic driving more digital logic of the same type. In practice, this means that you don't often have to worry about what's actually going on at logic inputs and outputs.

However, as soon as you attempt to drive digital circuitry with externally generated signals, whether digital or analog, or whenever you use digital logic outputs to drive other devices, you must face the realities of what it takes to drive a logic input and what a logic output can drive. Furthermore, when mixing logic families it is essential to understand the circuit properties of logic inputs and outputs. Interfacing between logic families is not an academic question. In order to take advantage of advanced LSI chips, or special functions that are available in only one logic family, you must know how to mix logic types. In the next few sections we will consider the circuit properties of logic inputs and outputs in detail, with examples of interfacing between logic families and between logic devices and the outside world.

Input characteristics

The graphs in Figure 9.3 show the important properties of CMOS and TTL inputs: input current and output voltage (for an inverter) as a function of input voltage. We have extended the graphs to input voltages beyond the range normally encountered in digital circuits, since in interface situations the input signals might easily exceed the power-supply voltages. As the graphs imply, both CMOS and TTL are normally operated with the negative supply pin connected to ground.

A TTL input sources a sizable current when held LOW, and it draws only a small current when HIGH (typically a few microamperes, never more than 20μA). To drive a TTL input, you must be able to sink a milliamp or so (see Table 9.1 for exact values) while holding the input below 0.4 volt. Failure to understand this may lead to widespread circuit malfunction in interfacing situations! For input voltages below ground, a TTL input looks like a clamping diode to ground; for inputs above +5 volts, the current is set by breakdown of a diode (LS, F) or base-emitter junction (ALS, AS), with onset of current somewhere above 10 volts.

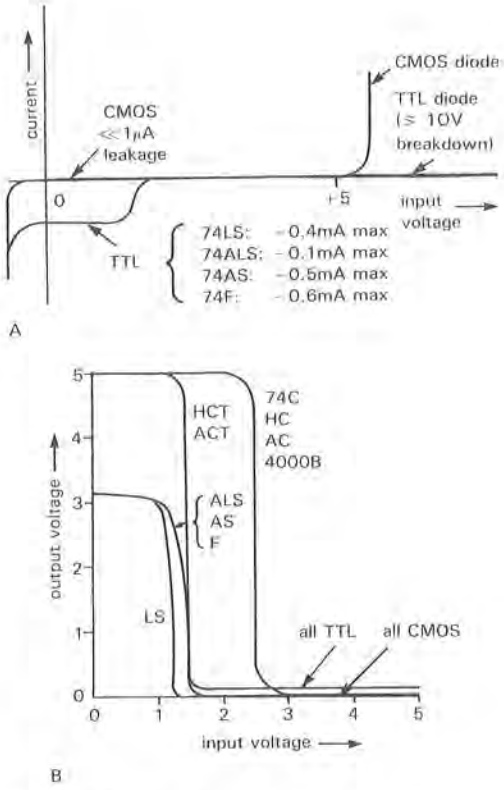


Figure 9.3. Logic gate characteristics.
A. Input current.
B. Transfer function.

The TTL input threshold is typically around +1.3 volts, although the specifications only guarantee that it is between +0.8 and +2.0 volts. TTL gates with Schmitt-trigger inputs are available ('13, '14, '132), with ±0.4 volt hysteresis; they are indicated by a hysteresis symbol within the gate outline (e.g., Fig. 9.9). V_{supply} (usually called V_{CC}) is +5.0 volts ±5%.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

A CMOS input draws no current (except for leakage current, typically $10^{-5}\mu$A) for input voltages between ground and the supply voltage. For voltages beyond the supply range, the input looks like a pair of clamping diodes to the positive supply and ground (Fig. 9.1). Momentary currents greater than about 10mA through these diodes is all it takes to put many CMOS devices into SCR latchup (see Section 8.35; newer designs withstand higher currents and tend to be resistant, or immune, to this disease; for example, the inputs of HC and HCT families can be driven 1.5V beyond the supply rails without malfunction or damage). These are the famous input-protection diodes, without which CMOS would be extremely susceptible to damage from static electricity during handling (as it is, CMOS is still relatively fragile). For 4000B, 74C, 74HC, and 74AC, the input logic threshold is typically at half the supply voltage, but it can range from about one-third to two-thirds of V_+ (V_+ is usually called V_{DD}); for 74HCT and 74ACT types the input threshold is around +1.5 volts, for compatibility with TTL. As with TTL, CMOS gates with Schmitt-trigger inputs are available. CMOS supply voltages are +2 volts to +6 volts (HC, AC), +5 volts ±10% (HCT, ACT), and +3 volts to +18 volts (4000B, 74C).

Output characteristics

The TTL output circuit is an *npn* transistor to ground and an *npn* follower (or Darlington) to V_+ with a current-limiting resistor in its collector. One transistor is saturated and the other is off. As a result, a TTL device can sink a large current (8mA for 74LS, 24mA for 74F) to ground with a small (saturation) voltage drop and can source at least a few milliamps with its output HIGH (about +3.5V). The output circuit is designed to drive TTL inputs, with a fanout of 10.

The CMOS output circuit is a push-pull pair of complementary MOSFETs, one ON and the other OFF (Fig. 9.1). The output looks like a MOSFET r_{ON} to ground or to V_+ when it is within a volt or so of the respective rail, becoming something like a current source when you draw so much current that the output is forced more than a volt or two away from the supply rails. Typical values of r_{ON} are 200 ohms to 1k (4000B/74C), 50 ohms [74HC(T)], and 10 ohms [74AC(T)]. Figure 9.4 shows a summary of the output characteristics of CMOS and TTL.

We have plotted the typical output voltage, for both HIGH and LOW output states, against output current. To simplify the graph, output current is always drawn positive. Note that CMOS pulls its outputs all the way to V_+ or ground, generating a full swing unless heavily loaded; when driving only CMOS loads (zero dc current), the swing is fully rail-to-rail. TTL levels, by comparison, are typically 50 – 200mV (LOW) or +3.5 volts (HIGH) when driving other TTL devices as loads. With a pullup resistor (discussed later), HIGH TTL outputs go all the way to +5 volts.

9.03 Interfacing between logic families

It is important to know how to make different logic families talk to each other, because there are situations when you must mix logic types. For example, many desirable LSI chips are built with NMOS, which has TTL-like output levels (i.e., a HIGH output around +3V) that cannot directly drive 74HC. Another example is the nice 74C9*xx* series of counters, which you might want to use within an existing circuit built with 74LS. Or you might want to put some 5 volt logic around the edges of a 12 volt CMOS system, for easy connection to external TTL-compatible signals and for driving cables.

The three things that can keep you from connecting any pair of logic chips together

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

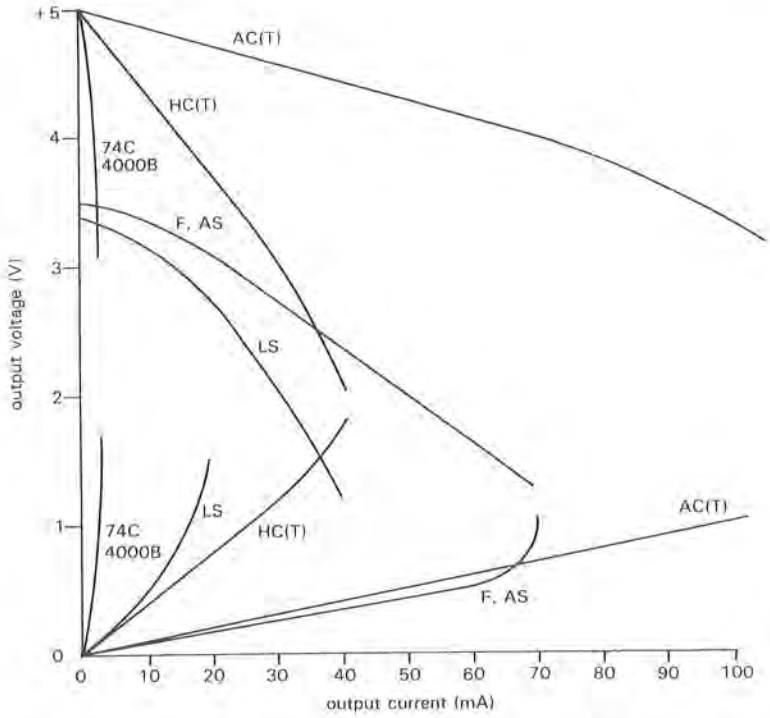


Figure 9.4. Logic gate output characteristics.

are (a) input logic-level incompatibility, (b) output drive capability, and (c) supply voltages. Rather than bore you with pages of explanations of what works and what doesn't, we've boiled down the interface problem to a simple table (Table 9.2). Let's take a quick tour.

TTL requires a +5 volt supply and typically pulls HIGH only to about +3.5 volts; it has good sinking, almost to ground. So it can drive logic with low threshold, namely TTL, HCT, ACT, and NMOS (which is routinely designed to be compatible). You need full swing to +5 volts to drive HC, AC, and 4000B/74C running at 5 volts, which you can do with a resistor pullup to +5 volts, or by interposing an HCT buffer (recall that HCT and ACT have full-swing outputs). If you use a pullup, note that the resistor value is a compromise – smaller is faster, but uses more power. A value of 4.7k is typical. A pullup resistor brings the TTL HIGH output all the way to +5 volts, though the last part of the rising waveform (during which the resistor is doing all the work) is slow. To drive high-voltage CMOS, use a level translator like the 40109, 14504, or LTC1045; they are very slow, but who cares, since you're driving slow CMOS anyway. NMOS outputs are similar to TTL, but generally with less drive capability. You therefore use the same techniques.

All CMOS families swing their outputs rail-to-rail, which means you can drive TTL, NMOS, and 5 volts CMOS directly from 5V-powered CMOS. Note, however, that old-fashioned CMOS (4000B/74C) has feeble drive when running from 5 volts (0.5mA sinking), crippling its ability to drive TTL. For all these families, use a level up-translator to drive high-voltage CMOS.

A nice solution to the CMOS-to-TTL/NMOS interface problem is to run the CMOS at reduced supply voltage; JEDEC Standard No. 8 specifies +3.3 volts, which puts the input threshold near the usual TTL value of 1.4 volts. Thus, TTL can drive 3.3V-powered HC/AC

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

TABLE 9.2. LOGIC FAMILY CONNECTIONS

TO → / *FROM* ↓	TTL	HCT ACT	HC AC	HC, AC @3.3V	NMOS LSI	4000B, 74C @5V	4000B, 74C @10V
TTL	OK	OK	A	OK	OK	A	B
HCT ACT	OK	OK	OK	NO	OK	OK	B
HC AC	OK	OK	OK	NO	OK	OK	B
HC, AC @3.3V	OK	OK	NO	OK	OK	B	B
NMOS LSI	OK	OK	A	OK	OK	A	B
4000B, 74C @5V	OK[a]	OK	OK	NO	OK	OK	B
4000B, 74C @10V	C	C	C	C	C	C	OK

[a] with limited fanout. A – pullup to +5V, or use HCT as interface.
B – use *i)* OC pullup to +10V, or *ii)* 40109, 14504, or LTC1045 level translator.
C – use 74C901/2, 4049/50, 14504, or LTC1045 level translator.

directly, and vice versa. As an added bonus, 3.3 volt operation decreases dynamic power consumption (see Sections 8.10 and 14.16 and Figs. 8.18 and 14.38) by 55% from its 5 volt value, while increasing propagation delays by about 40%. Note, however, that you cannot interconnect (in either direction) 3.3 volt CMOS from other CMOS running at 5 volts.

EXERCISE 9.1.
Explain why the last statement is true.

Finally, high-voltage CMOS can drive 5 volt logic if you interpose a level translator (74C901/2, 14504, LTC1045, or 4049/4050) to generate 5 volt output swings. It is probably safe to drive LS TTL directly from high-voltage CMOS, since there are no input-protection diodes, and input breakdown usually exceeds 10 volts; however, a strict interpretation of LS specifications (7V absolute maximum input voltage) requires the translator.

Warning: Although the static logic levels are OK, there is an interesting *dynamic* incompatibility that sometimes crops up when you try to drive edge-sensitive inputs (e.g., clocking inputs of counters) of

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

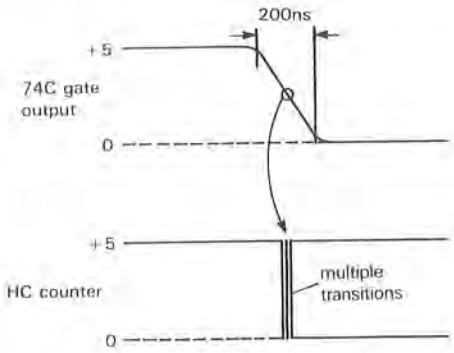


Figure 9.5. Edge-sensitive fast logic should not be driven with slow signals (e.g., slow logic parts).

HC or AC logic from the outputs of slower 4000B- or 74C-type logic. Figure 9.5 shows the multiple logic transitions you often see; sometimes the HC chip refuses to count at all, unless an oscilloscope probe (or small capacitor) is attached! Apparently the combination of slow transition time and relatively high output impedance from the sluggish CMOS is the culprit.

Figure 9.6 illustrates most of the cross-family hookups that you're likely to encounter.

9.04 Driving CMOS and TTL inputs

Switches as input devices

It is easy to drive digital inputs from switches, keyboards, comparators, etc., if you keep in mind the input characteristics of the logic you're driving. The simplest way is with a pullup resistor (Fig. 9.7). When driving TTL, a pullup resistor with switch closure to ground is by far the best method, because of the input properties

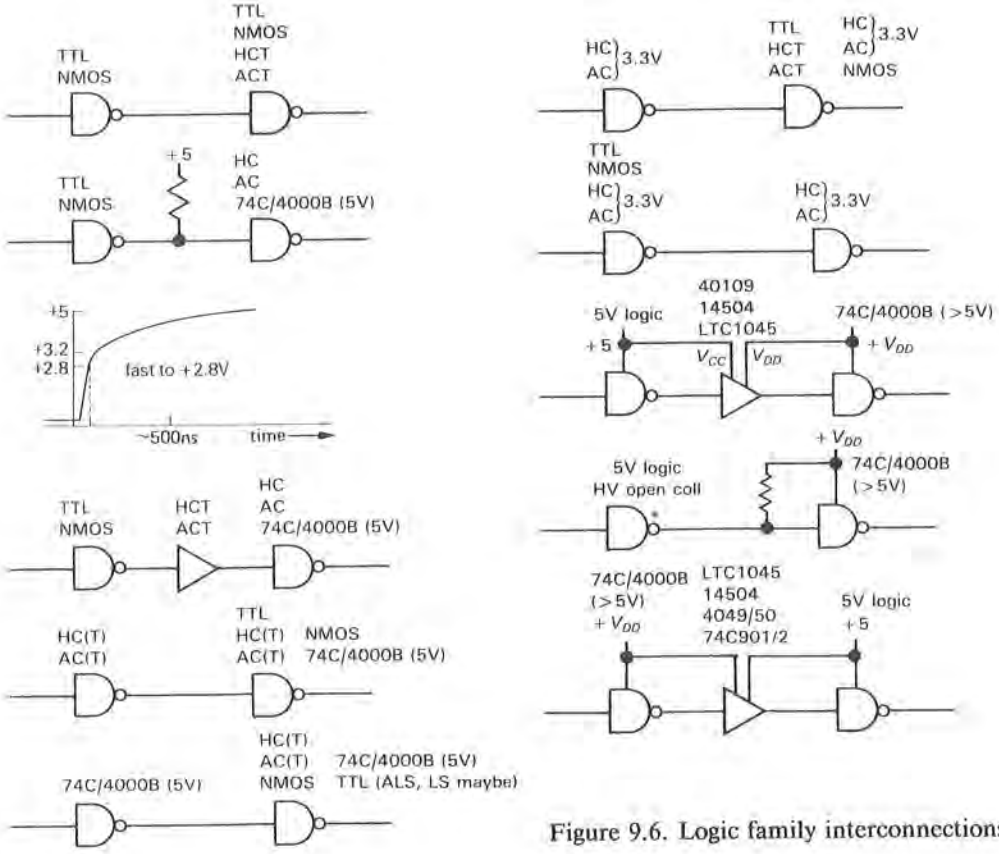


Figure 9.6. Logic family interconnections.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

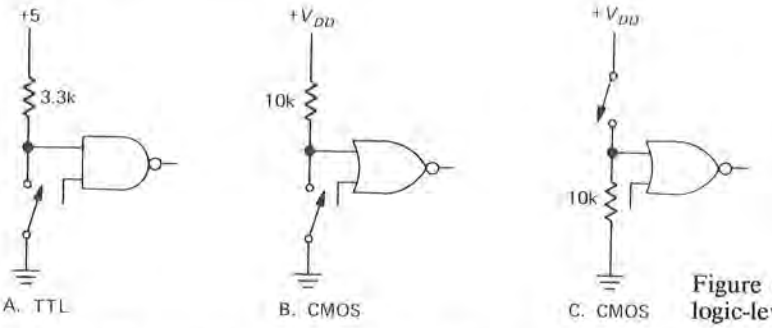


Figure 9.7. Mechanical switch to logic-level circuits (not debounced).

of TTL. The switch easily sinks the LOW-state input current, and the pullup brings the HIGH state to +5 volts, giving good noise immunity; in addition, it is convenient to have the switch return to ground.

The alternative possibility, a pulldown resistor to ground with switch closure to +5 volts, is undesirable because it requires a small pulldown resistor (such as 220 Ω) to guarantee a TTL LOW of a few tenths of a volt, which means you have rather large currents with the switch closed. In the pullup circuit, the noise immunity with the switch open (the worst case, from the standpoint of noise pickup) is at least 3 volts, whereas in the undesirable pulldown circuit it could be as little as 0.6 volt (FAST TTL, −0.6mA input current, LOW threshold at +0.8V).

Either pullup or pulldown with CMOS is fine, since the inputs draw no current and the threshold is typically halfway from V_{DD} to ground. It is usually more convenient to ground one side of the switch, but if the circuit is simplified by having a HIGH input when the switch is closed, the method with pulldown resistor will be perfectly OK. Figure 9.7 shows these three methods.

Switch bounce

As we remarked in Chapter 8, mechanical switch contacts usually bounce for about a millisecond after closure. With large switches the bounce can last for as much as 50ms. This can wreak havoc with circuits that are sensitive to changes of state, or "edges" (a flip-flop or counter would toggle many times if clocked directly from a switch input, for instance). In such cases it is essential to debounce the switch electronically. Here are a few methods:

1. Use a pair of gates to make a jam-type SR flip-flop. Use pullups at the inputs to the debouncer, of course (Fig. 9.8). You could use a flip-flop with SET and CLEAR inputs instead (e.g., a '74); in that case, ground the clocking input.

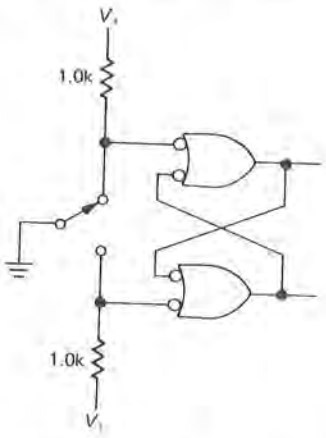


Figure 9.8. Switch debouncer: SR flip-flop.

2. Use an integrated version of the preceding arrangement. The '279, 4043, and 4044 are quad SR latches.

3. Use an RC slowdown network to drive a CMOS Schmitt trigger (Fig. 9.9). The low-pass filter R_2C_1 smooths the bouncy waveform so that the Schmitt-trigger gate

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

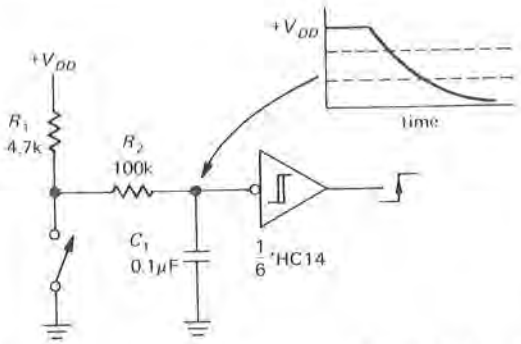


Figure 9.9. Switch debouncer: *RC* plus Schmitt trigger.

makes only one transition. A 10ms to 25ms *RC* time constant is generally long enough. This method isn't well suited to TTL because of the low driving impedance that TTL inputs require.

4. Use a chip like the 4490 "hex contact bounce eliminator," a nifty device that uses a digital delay (a 5-bit shift register for each switch) as a sort of digital low-pass filter. It includes internal pullup resistors and clock circuitry. The user supplies a timing capacitor that sets the oscillator frequency, thus determining the delay time.

5. Use the circuit shown in Figure 9.10, with either a noninverting gate or buffer.

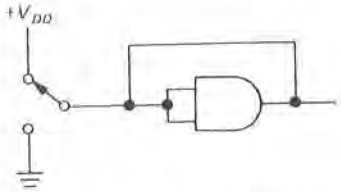


Figure 9.10. Switch debouncer: noninverting gate with feedback.

It's always OK to override a logic output by shorting it to V_+ or ground, providing the duration is kept short. In this circuit there is no problem, because the output is forced only for a gate propagation delay, after which it holds itself in the new state.

6. Use a device with built-in debouncer. Keyboard encoders, for instance, are designed with mechanical switches in mind as input devices, and they usually include debouncing circuitry.

7. Use a Hall-effect switch. These are magnetically operated solid-state switches, available as panel switches or keyboard switches. In either case the magnet and switch come as a complete assembly. They require +5 volts and produce a debounced logic output suitable for driving TTL or CMOS (operated from +5V). Since they have no mechanical contact assembly to wear out, Hall-effect switches last just about forever (although we once had an epidemic of progressive magnet anemia in a Hall-effect keyboard; we trust this disease has now been eradicated).

A few general comments on switches as input devices: Note that SPST switches (sometimes called "form A") can be used with methods 3 and 4 (and usually 6), whereas SPDT switches (form B) are necessary with the other methods. Keep in mind, also, that often it isn't necessary to debounce switch inputs, since they aren't always used to drive edge-sensitive circuitry. Another point: Well-designed switches are usually "self-wiping" to maintain a clean contact surface (take one apart to see what that means), but it's a good idea to choose circuit values so that a current of at least a few milliamps flows through the switch contacts to clean them. With suitable choice of contact material (e.g., gold) and mechanical design, switches can be designed to avoid this "dry-switching" problem and will work properly even when switching zero current.

9.05 Driving digital logic from comparators and op-amps

Comparators and op-amps, together with analog-to-digital converters, are the common input devices by which analog signals can drive digital circuits. Figure 9.11

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

shows some examples. In the first circuit, a comparator drives TTL directly. Since most comparators have an *npn* output transistor with open collector and grounded emitter, all that's needed is a pullup to +5 volts. The same scheme works for CMOS as well with pullup to $+V_{DD}$. The comparator may not have to run from a split supply, since many are designed for single-supply operation (V_- grounded), and some will even operate with a single 5 volt supply (e.g., the 311, 339, 393, or 372/4).

The second circuit shows an op-amp driving CMOS with only a series current-limiting resistor. The input-protection diodes of CMOS device form effective clamps to V_{DD} and ground, provided that input current is kept below 10mA. In the third circuit, an op-amp switches an *npn* transistor into saturation to drive a TTL load; the diode prevents base-emitter reverse breakdown (~6V). In this circuit you can eliminate R_1 and D_1 by replacing the *npn* transistor with an *n*-channel MOSFET. The final circuit is not highly recommended, but it does work. The input clamping diode of the TTL device limits its negative swing to a diode drop below ground, and the external diode limits positive swing. The series resistor prevents damage if base-emitter reverse breakdown occurs in the TTL input transistor. The series resistor is chosen small enough to sink the TTL LOW-state input current when the op-amp output is a few volts negative.

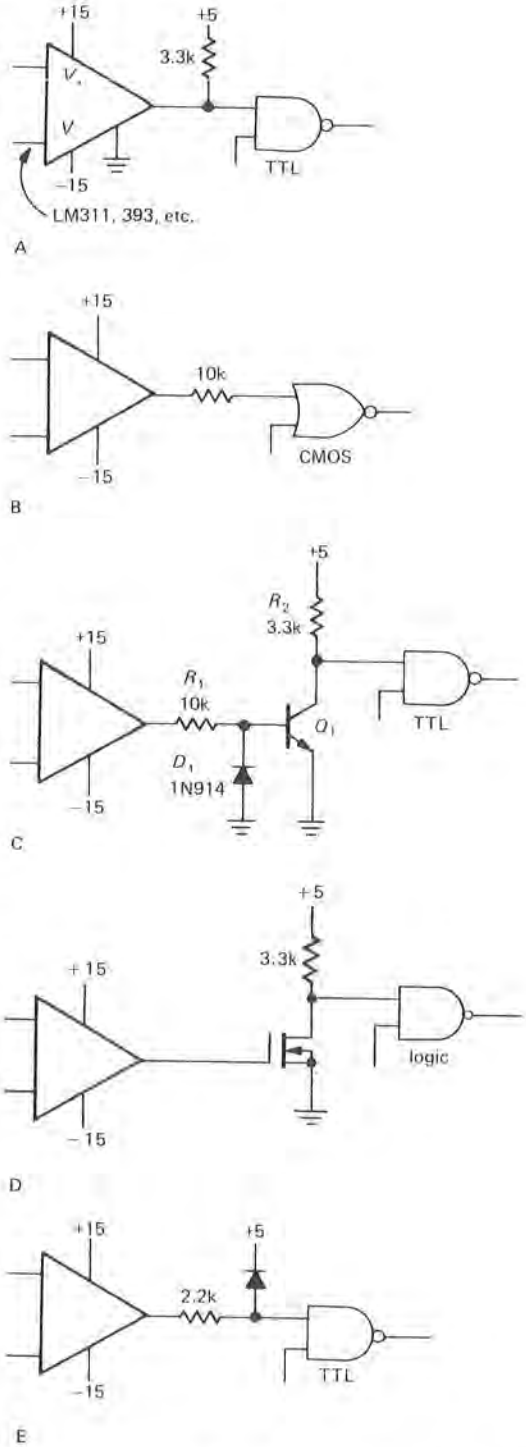


Figure 9.11. Driving logic from comparators and op-amps.

Clock inputs: hysteresis

A general comment about driving digital logic from op-amps: Don't try to drive clock inputs from these op-amp interfaces; the transition times are too long, and you may get glitches as the input signal passes through the logic threshold voltage. If you intend to drive clocking inputs (of flip-flops, shift registers, counters,

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

monostables, etc.), it is best to use a comparator with hysteresis or to buffer the input with a gate (or other logic device) with Schmitt-trigger input. The same comment goes for signals derived from transistor analog circuitry. Figure 9.12 shows the idea. R_2 is chosen for 50mV hysteresis. A small capacitor (C_2) across the feedback resistor is a good idea to ensure fast transitions and to prevent multiple pulses as the input passes through the threshold (the 311, in particular, has a tendency to do this). Bypass capacitor C_1 is also important in preventing glitches on the reference supply during transitions. In many cases the reference voltage will be ground, in which case C_1 is omitted.

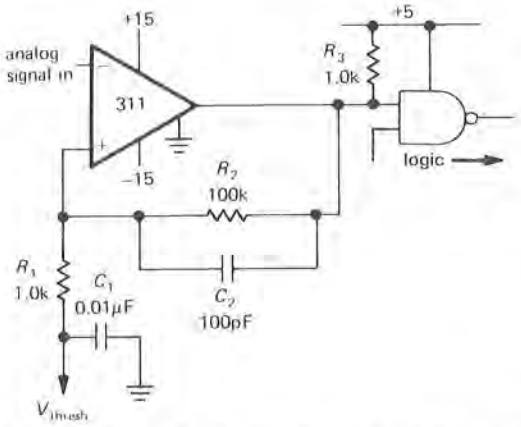


Figure 9.12. Threshold detector with hysteresis.

9.06 Some comments about logic inputs

TTL chips (and their HC and AC counterparts) tend to be designed with active LOW control inputs; for instance, flip-flops generally have SET and RESET inputs that are enabled when LOW. External digital signals used as inputs will therefore nearly always have a pullup resistor and will pull LOW (sinking current) when active, a convenient arrangement, because switches, etc., can use a common ground return. It also leads to greater noise immunity, since a line held near +5 volts has 3 volts of noise immunity, as compared with the 0.8 volt of noise immunity of a line held near ground. This intrinsic weakness of TTL logic (poor noise immunity in the LOW state) is particularly apparent when you realize that a negative 0.5 volt spike on the chip's ground line can be interpreted as a HIGH input by the chip. Such spikes are not uncommon, since they can be generated by short current spikes through the ground-line inductance. See Section 9.11 for further discussion of this troublesome problem.

With CMOS, the noise immunity is the same for either state, so you can use pullup or pulldown resistors as input terminations when driving from devices that have an open state. Pulldown resistors tend to be used more often, although you will see pullups when the driving device is something like a switch, with a ground return.

An open TTL input is "barely HIGH." It sits at the logic threshold (~1.3V), but since no current is being sunk, it does not turn on the input transistor. You may occasionally see "designs" in which inputs that should be tied HIGH on TTL devices are left open. Don't *ever* do this! It is foolish and dangerous: There is zero noise immunity on an open input, so capacitive coupling from nearby signals can generate short LOW spikes at the input. This generates glitches at the outputs of combinational devices (gates), which is bad enough, but in the case of a flip-flop or a register it can be devastating, since an open RESET′ input can clear the device at unpredictable times. The offending glitches may be impossible to see on a scope, since they may be "single-event" pulses of about 20ns duration. Although you may be able to get away with this unsavory practice most of the time, especially if there is low capacitance between the open pin and neighboring pins, it is still a very poor idea; if you try to

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

troubleshoot the circuit by clipping a logic analyzer or test clip over the IC, you've got a new circuit, since the additional capacitance of the test device is almost certain to cause transient LOWs at the open pin(s). Besides, why build unreliable circuits when you know how to make them reliable with a few simple connections? (End of tirade.)

Unused inputs

Unused inputs that affect the logic state of a chip (e.g., a RESET input of a flip-flop) must be tied HIGH or LOW, as appropriate. Inputs that have no effect (e.g., inputs of unused gate sections in the same package) may be left unconnected in TTL, but not in CMOS. An open input of an unused CMOS gate, for instance, can float up to the logic threshold, causing the output to go to half the supply voltage, with both MOS output transistors conducting, thus drawing considerable class A current. This results in excessive supply current and can even lead to failure in devices with hefty output stages. It is best to ground all inputs of unused functions on every CMOS chip.

With TTL you can ignore unused sections of a chip, as well as irrelevant inputs of circuits you are using. For instance, you can leave the parallel-load data lines of a counter unconnected if you never enable the LOAD line.

9.07 Comparators

Comparators were introduced briefly in Section 4.23 to illustrate the use of positive feedback (Schmitt trigger) and to show that special-purpose comparator ICs deliver considerably better performance than general-purpose op-amps used as comparators. These improvements (short delay times, high output slew rate, and relative immunity to large overdrive) come at the expense of the properties that make op-amps useful (in particular, careful control of phase shift versus frequency). Comparators are not frequency-compensated (Section 4.33) and cannot be used as linear amplifiers.

Comparators provide an important interface between analog (linear) input signals and the digital world. In this section we would like to look at comparators in some detail, with emphasis on their output properties, their flexibility regarding power-supply voltages, and the care and feeding of input stages.

Supply voltages and outputs

Most comparators have open-collector outputs, suitable for driving logic inputs (with a pullup resistor, of course) or higher current/voltage loads. The 311, for instance, can drive loads connected to supplies up to 40 volts with currents up to 50mA, and the 306 can handle even more current. These comparators have a ground pin in addition to the negative and positive supply pins, so the load is pulled to ground regardless of supply voltages. Comparators intended for very high speed (521, 527, 529, 360, 361, Am686, CMP-05, LT1016, and VC7695/7) often come with active pullup output circuits. These are meant to drive 5 volt digital logic, and they usually have 4 power-supply pins: V_+, V_-, V_{CC} (+5), and ground.

One thing to watch for is the fact that many comparators require positive and negative supplies, even if the inputs never go negative. Examples are the 306, 710, and 711, in addition to the comparators with active pullup listed previously. It is inconvenient to have to generate a negative supply just to run a comparator in an instrument that otherwise uses only positive supplies, so it is important to know about comparators that will run on a single positive supply (e.g., the 311, 319, 339, 393, 365, CA3290, HA4905,

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

CMP-01, CMP-02, LT1016, AD790, and TLC372/4). In fact, these will all operate with a single 5 volt supply, a great advantage in a digital system. When operated from a single +5 volt supply, the 339, 393, 365, CA3290, HA4905, LT1017/18, AD790, and TLC372/4 have an input common-mode range all the way down to ground. These last comparators are intended specifically for single-supply operation and, with the exception of the 4905, 365, and 790, have only two supply pins (V_+ and ground); if operated from split supplies, the output will be pulled to V_-. Several of these also have the unusual property of being able to operate from a single supply of as little as +2 volts.

While on the subject of power supplies, it should be mentioned that some comparators are designed to run on low supply current, generally less than 0.5mA; examples are the LP311, LP339, TLC373/4, TLC339/393, TLC3702/4, CMP-04, LT1017/8, MC14574, and LP365. The last two are quad comparators with programmable operating current. The price you pay for low power is low speed, with response times measured in microseconds. See Chapter 14 for the full story on low-power electronics; Table 14.8 lists low-power comparators.

Inputs

There are some general cautions concerning input circuits of comparators. Hysteresis (Section 4.24) should be used whenever possible, because erratic switching is otherwise likely to result. To see why, imagine a comparator without hysteresis in which the differential input voltage has just passed through zero volts, slewing relatively slowly, since it is an analog waveform. A mere 2mV input differential causes the output to change state, with switching times of 50ns or less. Suddenly you have 3000mV fast digital logic transitions in your system, with current pulses impressed on the power supplies, etc. It would be a miracle if some of these fast waveforms didn't couple into the input signal, at least to the extent of a few millivolts, overcoming the 2mV input differential and thus causing multiple transitions and oscillations. This is why generous amounts of hysteresis (including a small capacitor across the feedback resistor), combined with careful layout and bypassing, are generally required to get sensitive comparator circuits to function well. It is generally a good idea to avoid driving comparator inputs directly from high-impedance signals; use an op-amp output instead. It is also a good idea to avoid high-speed comparators, which only aggravate these problems, if speed is not needed. Then, too, some comparators are more troublesome in this regard than others; we have had plenty of headaches using the otherwise admirable 311.

Another caution on inputs: Some comparators have a very limited differential input voltage range, as little as 5 volts for some types (e.g., the CMP-05, 685-7, and VT9695/7). In such cases it may be necessary to use diode clamps to protect the inputs, since excessive differential input voltage will degrade h_{FE} and cause permanent input offset errors and may even destroy the base-emitter junctions of the input stage. General-purpose comparators are generally better in this respect, with typical differential input voltage ranges of ±30 volts (e.g., the 311, 393, LT1011, etc.).

An important feature of comparator inputs is the bias current at the input terminals and the way it changes with differential input voltage. Most comparators use bipolar transistors for their input stages, with input bias currents ranging from tens of nanoamps to tens of microamps. Since the input stage is just a high-gain differential amplifier, the bias current changes as the input signal takes the comparator through its threshold. In addition, internal protection circuitry may cause a larger change in bias current a few volts away

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

from threshold. The curve shown in Figure 9.13 (for the CMP-02) is typical. The small current "step" at zero volts (differential) is actually a smooth transition taking place over 100mV or so, and represents the voltage change necessary to switch the input differential amplifier stage fully from one state to the other.

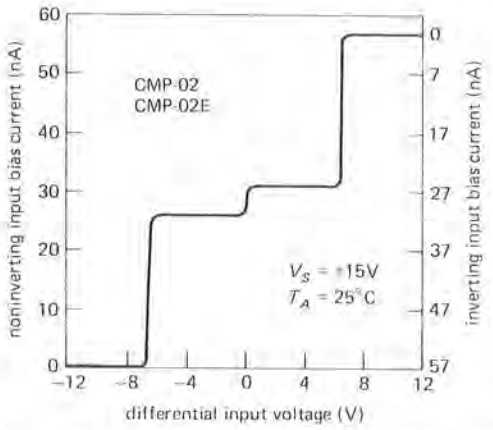


Figure 9.13. Input bias current versus differential input voltage for the CMP-02 comparator. (Courtesy of Precision Monolithics, Inc.)

For comparator applications where extremely low input current is necessary, there are FET-input comparators available. Examples are the CA3290 and TLC372, TLC3702, and TLC393 MOSFET-input dual comparators and the JFET LF311, a version of the popular 311. The latter offers maximum input currents of 50pA, compared with 250nA for the 311, with essentially no sacrifice in offset voltage or speed. In situations where the properties of a particular comparator are needed, but with lower input current, one solution is to add a matched-pair FET follower at the input.

One last comment on input properties: Thermal gradients set up on the chip from dissipation in the output stage can degrade input offset voltage specifications. In particular, it is possible to have "motorboating" (a slow oscillation of the output state) take place for input signals near zero volts (differential), since the state-dependent heat generated at the output can cause the input to switch.

Overall speed

It is convenient to think of a comparator as an ideal switching circuit for which any reversal in the differential input voltage, however small, results in a sudden change at the output. In reality, a comparator behaves like an amplifier, for small input signals, and the switching performance depends on the gain properties at high frequencies. As a result, a smaller input "overdrive" (i.e., more than enough signal to cause saturation at dc) causes a greater propagation delay and (often) a slower rise or fall time at the output. Comparator specifications usually include a graph of "response times for various input overdrives." Figure 9.14 shows some for the 311. Note particularly the reduced performance in the configuration where the output transistor is used as a follower, i.e., with less gain. Increased input drive speeds things up because the amplifier's reduced gain at high frequencies is overcome by a larger signal. In addition, larger internal amplifier currents cause internal capacitances to charge faster.

Table 9.3 compares the properties of most of the comparators currently available.

9.08 Driving external digital loads from CMOS and TTL

It is easy to drive on/off devices like lamps (LEDs), relays, displays, and even ac loads from the outputs of CMOS or TTL logic. Figure 9.15 shows some methods. Circuit A shows the standard method of driving LED indicator lamps from 5 volt logic. For TTL, which is better at sinking than sourcing, the LED is returned to +5 volts; for CMOS you can return the LED either

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

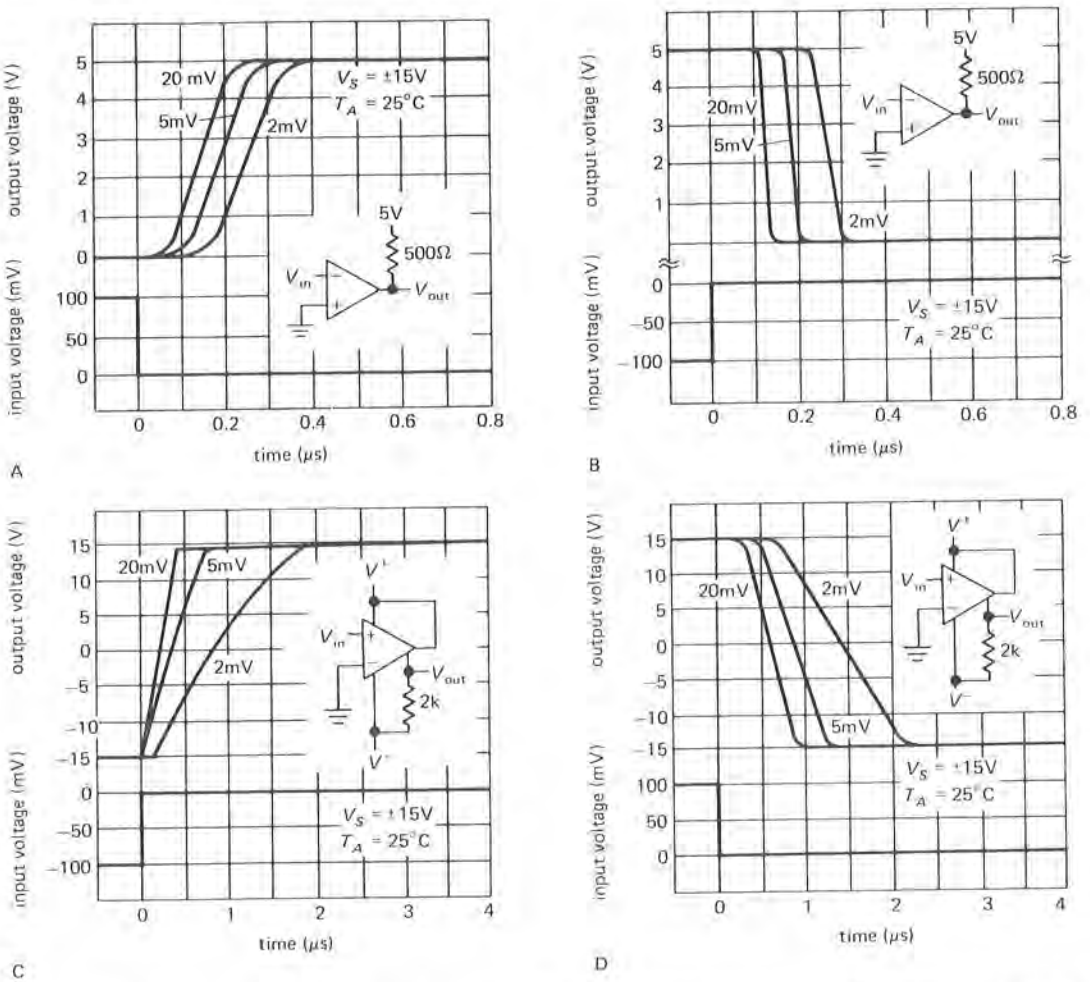


Figure 9.14. LM311 comparator response times for various input overdrives. (Courtesy of National Semiconductor Corp.)

to V_+ or to ground. The LED behaves like a diode with a forward drop of 1.5 to 2.5 volts at typical operating currents of 5mA to 20mA; with some of the newer high-efficiency LEDs (you can get some real dazzlers from Stanley) you get plenty of light with just a few milliamps. Instead of using a discrete LED plus resistor, you can get LEDs with integral current-limiting resistor (or current regulator) from most manufacturers; check the catalogs of Dialight, General Instrument, Siemens, and Hewlett-Packard.

Circuit B shows how to drive a 5 volt low-current relay directly from logic, sinking current as in circuit A. The diode is essential to clamp the inductive spike. The relay shown comes in a standard DIP IC package and has 500 ohms coil resistance (10mA, within the drive capability of most 5 volt logic). For driving higher-voltage loads, the schemes shown in circuits C through E are effective. In circuit C, a 74LS26 open-collector gate with 15 volt rating drives a 12 volt relay, and in circuit D a 75451 "dual peripheral driver" chip rated at up to 300mA and 30 volts drives an unspecified load. Open-collector

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

TABLE 9.3. COMPARATORS

Type	Mfg[a]	Qty per pkg	t_r typ[b] (ns)	V_{os} max (mV)	I_b max (μA)	CM to V_-[c]	CM input range abs max[d] min (V)	CM input range abs max[d] max (V)	Abs max diff'l input[e] (V)	Power supplies: Positive min (V)	Positive max (V)	Negative min (V)	Negative max (V)
CMP01C	PM	1	110	2.8	0.9	–	V_-	V_+	11	5	30	0	-30
CMP02	PM	1	190	0.8	0.003	–	V_-	V_+	11	5	30	0	-30
CMP04	PM	4	1300	1	0.1	•	-0.3	30	36	3	36	0	-30
CMP05	PM	1	40	0.6	1.2	–	V_-	V_+	5	5[i]	6	-5.2[i]	-18
LM306	NS	1	28	6.5	5	–	-7	7	5	12[i]	-	-3	-12
LM311	NS	1/2	200	3	0.1	–	V_+-30	V_-+30	30	5	30	0	-30
LF311	NS	1	200	4	0.00005	–	V_+-30	V_-+30	30	5	30	0	-30
LP311	NS	1	2000	7.5	0.1	–	V_-	V_-+30	30	3	30	0	-30
LM319	NS	2	80	4	0.5	–	V_-	V_+	5	5	30	0	-30
LM339	NS	4/2	1300	5	0.25	•	-0.3	36	36	2	36	-	-
LP339	NS	4	10000	5	0.025	•	-0.3	36	36	2	36	-	-
TLC339	TI	4/2	2500	5	0.005[f]	•	-0.3	V_+	18	3	16	-	-
TLC3702	TI	2/4	2500	5	0.005[f]	•	-0.3	V_+	18	3	16	-	-
LM360	NS	1	14	5	20	–	V_-	V_+	5	4.5	6.5	-4.5	-6.5
LM361	NS	1	14	5	30	–	-6	6	5	5	15[i]	-6	-15
LP365	NS	4	2000	6	0.075	•	-0.3	36	36	2	36	0	-36
TLC372	TI	2/4	650	10	1pA[f]	•	-0.3	18	18	2	18	-	-
CMP404	PM	4	3500	1	0.05	•	-0.3	V_+	V_+	5	30	-	-
TL510C	TI	1/2	30	3.5	20	–	-7	7	5	10	14	-5	-7
NE521	SN	2	11	7.5	20	–	-5	5	6	5[i]	-	-5[i]	-
NE522	SN	2	14	7.5	20	–	-5	5	6	5[i]	-	-5[i]	-
NE527	SN	1	33	6	2	–	-6	6	5	5	10	-6	-10
NE529	SN	1	20	6	20	–	-6	6	5	5	10	-6	-10
Am685	AM	1	6	2	10	–	-4	4	6	6[i]	-	-5.2[i]	-
Am686	AM	1	9	2	10	–	-4	4	6	5[i]	-	-6[i]	-
Am687	AM	2	7	2	10	–	-4	4	6	5[i]	-	-5.2[i]	-
Bt687	BT	2	1.8			–	-3.3	3.3		5[i]	-	-5.2[i]	-
AD790K	AD	1	35	0.3	3	•	-18	18	15	3.5	18	0	-18
TL810C	TI	1/2	30	3.5	20	–	-7	7	5	10	14	-5	-7
LT1011	LT	1	150	0.5	0.025	–	V_--0.6	V_++0.6	36	3	36	0	-36
LT1016	LT	1	10	3	10	–	V_-	V_+	5	4.5	7	0	-7
LT1017	LT	2	20000	1	0.015	•	-0.3	40	40	1.1	40	-	-
LT1018	LT	2	6000	1	0.075	•	-0.3	40	40	1.1	40	-	-
LT1040	LT	1	80000	0.5	0.0003[f]	•	V_-+0.3	V_+-0.3		2.8	16	-	-
SP1650B	PL	2	3.5[k]	20	10	–	-3	2.5	5	5[i]	-	-5.2[i]	-
EL2018C	EL	1	20	3	0.3	–	V_-	V_+	36	4.5	18	-4.5	-18
EL2019C	EL	1	6[l]	5	0.3	–	V_-	V_+	36	4.5	18	-4.5	-18
CA3290A	RC	2	1000	10	0.00004	•	V_--5	V_++5	36	4	36	-	-
RC4805A	RA	1	22	0.3	1.2	–	-4	4	3	4.5	5.5	-4.5	-16.5
HA4905	HA	4	150	7.5	0.15	•	V_-	V_+	15	5	30	0	-30
VC7695	VT	1	1.5	5	5[n]	–	-5	5	3.5	-	6	-	-6
VC7697	VT	2	2	5	5[n]	–	-5	5	3.5	-	6	-	-6
SP9685	PL	1	2.3	5	20	–	-5	3	5	5[i]	-	-5.2[i]	-
SP9687	PL	2	2.8	5	20	–	-5	5	5	5[i]	-	-5.2[i]	-
MC14574	MO	4	20000	30	0.00005	•	-0.5	V_++0.5	V_+	3	15	-	-
SP93808	PL	8	1.0	3.5	9	–	V_-	V_+	3.8	1.5	7.3	-4.9	-5.5

[a] see footnote to Table 4.1. [b] 100mV step with 5mV overdrive. [c] operating input common-mode range includes negative supply. [d] maximum range without input breakdown; will not operate properly over entire range. [e] maximum allowable voltage between input terminals. [f] ability to accept signals of both polarities and drive unipolar logic. [g] E - output (open NPN emitter) is designed to drive ECL logic; G - output pulled

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

Type	Total supply min (V)	Total supply max (V)	Gain typ	Output parameters: Gnd pin?	Single +5 OK?	Drive TTL?	±Sig to logic?[f]	OC output?	Active pullup?	Q and Q' out?	Strobe?	Latch?	V_{OL}[g]	Pullup V_{max}[h]	Comments
CMP01C	5	36	500k	–	•	•	•	–	•	–	–	–	R	32	
CMP02	5	36	500k	–	•	•	•	–	•	–	–	–	R	32	precision
CMP04	3	36	200k	•	•	•	–	•	–	–	–	–	G	30	precision 339
CMP05	9.5	24	16k	•	–	•	•	–	•	–	–	•	G	-	fast precision
LM306	-	30	40k	•	–	•	•	–	•	–	•	–	G	24	high output current
LM311	4.5	36	200k	•	•	•	•	•	–	–	–	–	R	40	may osc; popular; dual=2311
LF311	4.5	36	200k	•	•	•	•	•	–	–	–	–	R	40	JFET 311
LP311	3	36	200k	•	•	•	•	•	–	–	–	–	R	40	low power 311
LM319	4.5	36	40k	–	•	•	•	•	–	–	–	–	R	36	
LM339	2	36	200k	–	•	•	–	•	–	–	–	–	G	30	favorite; low pwr; dual=393
LP339	2	36	500k	–	•	•	–	•	–	–	–	–	G	30	low power 339
TLC339	3	16	-	–	•	•	–	•	–	–	–	–	G	18	MOSFET; dual=393
TLC3702	3	16	-	–	•	•	–	–	•	–	–	–	G	V_+	MOSFET; quad=3704
LM360	9	13	3k	•	–	•	•	–	•	•	–	–	G	-	similar to 760
LM361	11	30	3k	•	–	•	•	–	•	•	•	–	G	7	similar to 529
LP365	4	36	300k	•	•	•	•	•	–	–	–	–	R	36	prog; specs @ I_{set} = 10µA
TLC372	2	18	200k	•	•	•	–	•	–	–	–	–	G	18	MOSFET; quad=374
CMP404	5	30	400k	•	•	•	–	•	–	–	–	–	G	-	low power
TL510C	15	21	33k	•	–	•	•	–	•	–	•	–	G	-	TL514C=dual
NE521	9.5	10.5	-	•	–	•	•	–	•	–	•	–	G	-	
NE522	9.5	10.5	-	•	–	•	•	–	•	–	•	–	G	-	
NE527	10	20	-	•	–	•	•	–	•	•	•	–	G	15	529 with Darlington
NE529	10	20	-	•	–	•	•	–	•	•	•	–	G	15	
Am685	9.7	14	1600	•	–	–	•	–	–	•	•	•	E	-	ECL; also CMP-07
Am686	9.7	14	-	•	–	•	•	–	•	–	•	–	G	-	fastest TTL comp
Am687	9.7	14	-	•	–	–	•	–	–	•	•	•	E	-	ECL
Bt687	-	12	100	•	–	–	•	–	–	•	•	•	E	-	ECL; fastest 687-type
AD790K	3.5	36	10k	•	•	•	•	–	•	–	–	•	G	-	fast single +5V
TL810C	15	21	33k	•	–	•	•	–	•	–	–	–	G	-	510 w/o STB; 820C=dual
LT1011	3	36	500k	–	•	•	•	•	–	–	•	–	R	50	improved 311
LT1016	5	14	3k	•	•	•	•	–	•	•	–	•	G	-	fastest single +5V
LT1017	1.1	40	500k	•	•	•	–	–	•	–	–	–	G	-	low power
LT1018	1.1	40	2M	•	•	•	–	–	•	–	–	–	G	-	low power
LT1040	2.8	16	-	•	•	•	•	–	•	•	•	–	G	-	micropower, sampling
SP1650B	-	-	-	•	–	–	•	–	–	–	•	•	E	-	ECL; 1651 is faster
EL2018C	9	36	40k	•	–	•	•	–	•	–	•	•	G	-	fast, accurate, HV
EL2019C	9	36	-	•	–	•	•	–	•	–	•	•	G	-	fast, HV, clocked
CA3290A	4	36	150k	•	•	•	–	•	–	–	–	–	G	36	MOSFET
RC4805A	-	22	20k	•	–	•	•	–	•	–	–	•	G	-	fastest precision
HA4905	5	33	400k	–	•	•	•	–	•	–	–	–	R	-	flexible output stage
VC7695	-	12	-	•	–	–	•	–	–	•	–	•	E	-	ultra-fast
VC7697	-	12	-	•	–	–	•	–	–	•	–	•	E	-	fastest dual
SP9685	-	12	300	•	–	–	•	–	–	•	•	•	E	-	ECL; fast Am685
SP9687	-	12	300	•	–	–	•	–	–	•	•	•	E	-	ECL; fast Am687
MC14574	3	18	100k	•	•	•	–	–	•	–	–	–	G	V_+	CMOS, prog; specs@100µA
SP93808	6.5	13	20	•	–	–	•	–	•	•	–	•	E[o]	-	ultra-fast octal

to GND; R - low output is saturated NPN transistor to terminal that may be operated at voltages other than GND. [h] maximum voltage to which output can be externally pulled up. [i] nominal. [j] and additional +5V logic supply. [k] 100mV overdrive. [l] t_{setup}. [n] offset current. [o] use -5.2 and -10V supplies for ECL output. [t] typical.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

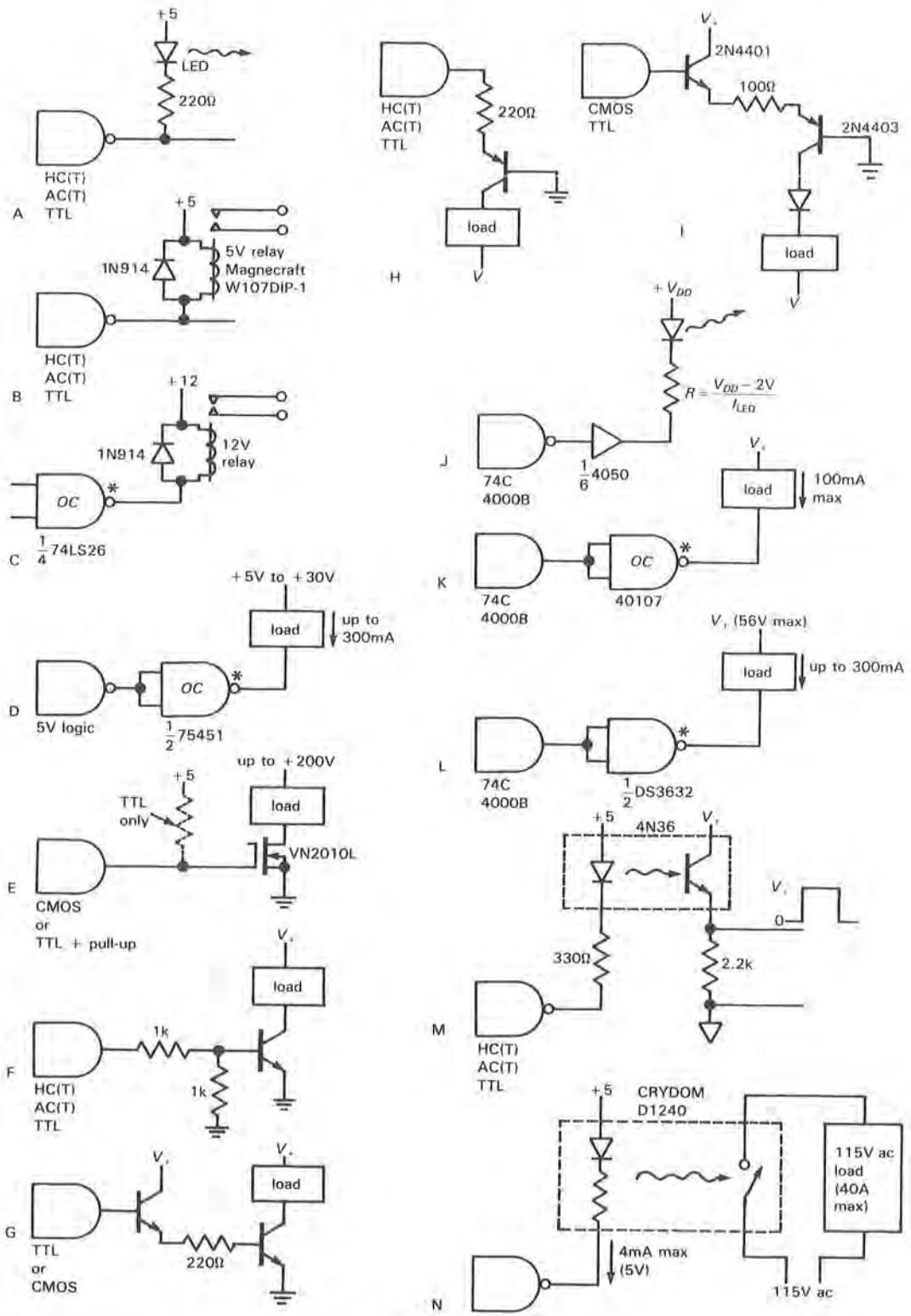


Figure 9.15. Driving loads from logic levels.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

devices like these are available with 80 volt ratings and above, and with even higher current capability; look at the DS3600 series from National and the series of power drivers from Sprague (UCN/UDN/ULN), which include some very nice octal drivers in DIPs. In circuit E, we've used a low-threshold *n*-channel power MOSFET, especially convenient because of its high input impedance; if driven with TTL output levels, it's best to use a pullup to guarantee good conduction, since TTL's minimum guaranteed HIGH output (2.4V) is otherwise too low.

The schemes mentioned previously may not be practical with HC, LS, or 74C logic, because of their limited output drive (5, 8, and 3.5mA sinking, respectively). It may be a good idea to set aside a chip like the 74AS1004, a robust hex inverter (48mA sink or source), for driving large LEDs. When driving high-current loads directly from logic chips, take particular care to use a hefty ground wire to the chip, since the current from the load returns through the chip into the logic supply ground. In some cases a separate ground return path may be desirable.

Circuit F shows how to use an *npn* transistor to switch a high-current load from 5 volt logic. To switch higher currents, use a second transistor, as in G. The circuits H and I show one way to drive loads returned to a negative supply. A HIGH output state turns on the *pnp* transistor, pulling the collector into saturation at one diode drop above ground. In circuit H, the resistor (or positive current limit of the gate) sets the emitter current, and therefore the maximum collector (load) current, whereas in the improved circuit I, an *npn* follower is used as a buffer, while a diode in series with the output keeps the load from swinging above ground. In both cases the maximum load current is equal to the drive current to the *pnp* transistor's emitter. Similar IC circuits are available, with CMOS/TTL-compatible inputs and high-voltage outputs with up to a few hundred milliamps drive capability; check out National's DS3687 (300mA, −56V) and the ever-popular UDN series from Sprague.

If you're using the low-current 4000B/74C logic, with barely a milliamp of output drive, it is generally necessary to use some sort of power driver, even to light LEDs. Circuit J shows the trusty 4050 hex buffer driving an LED. It can typically sink 5mA to 50mA with 5 to 15 volt supplies, respectively (output drive capability increases with increasing supply voltage). The circuits K and L show the application of even heftier drivers: The 40107 has a large "open-drain" *n*-channel MOS output transistor good for sinking 16mA to 50mA (5-15V supply voltages, respectively), and the DS3632 uses an *npn* Darlington driver, good for 300mA. Of course, discrete external transistors can always be used, as in circuits G and I, but they will be limited to less than a milliamp of base drive current. The discrete *n*-channel MOSFET solution used in circuit E works particularly well with gutless CMOS.

For driving remote loads, or loads with an independent ground system, the best method is an opto-coupler. This is a device with an LED (on the driver side) illuminating a photodetector (on the load side). Opto-couplers are available in various speeds, and with different input/output configurations [logic-compatible input or bare LED; logic output, saturated transistor (or Darlington) output, MOSFET output, or SCR or triac output; see Fig. 9.26]. A typical example is the popular 4N36 shown in circuit M, with bare LED input, *npn* transistor output, isolation of 2500 volt, and 4μs switching speed. It has a minimum current transfer ratio of 1.0, so you only need to drive the LED with a current equal to the maximum output current, as shown. There are some opto-couplers that conveniently use logic levels at both input and output. An example is the 74OL6000 from General Instrument;

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

it has LS inputs and outputs, 60ns propagation times (15MHz), and 2500 volt isolation. It costs $3 in quantity.

For driving ac loads, the easiest method is to use a "solid-state relay," as in circuit N. It is an optically coupled triac with logic-compatible input and 1 to 40 amp load-current capability when switching a 115 volt ac load. The low-current variety are available in DIP packaging (e.g., the "chipswitch" series from International Rectifier), while the heftier ones come as a chassis-mounting rectangular block roughly 2 inches square. Alternatively, you can switch ac loads with an ordinary relay, energized from logic. However, be sure to check the specifications, because most small logic-driven relays cannot drive heavy ac loads, and you may have to use the logic relay to drive a second larger relay. Most solid-state relays use "zero-crossing" (or "zero-voltage") switching, which is actually a combination of zero-voltage turn-on and zero-current turn-off; it is a desirable feature that prevents spikes and noise from being impressed onto the power line. Much of the "garbage" on the ac power line comes from triac controllers that don't switch at zero crossings, e.g., phase-controlled dimmers used on lamps, thermostatic baths, motors, etc. As an alternative to the optical coupling used internally in circuit N, you sometimes see a pulse transformer used to couple triggering pulses to a triac or SCR.

When driving 7-segment numeric displays, it is usually simplest to take advantage of one of the decoder/driver combinations available. They come in a bewildering variety, including LED and LCD drivers, sinking or sourcing polarities, etc. Typical examples are the 74HC4511 (common cathode LED) and 74HC4543 (LCD) latch/decoder/drivers. More on this in the opto-electronics section (Section 9.10).

9.09 NMOS LSI interfacing

Most large scale and very large scale integrated circuits (LSI, VLSI) are now being designed as CMOS, with pretty much the same interfacing properties as the 5 volt CMOS logic gates and other MSI (medium-scale integration) functions we just discussed. However, for a long time, LSI and VLSI chips were made from *n*-channel enhancement-mode MOSFETs alone, because of simpler processing and higher density. Because of the widespread use of this NMOS logic, it is important to know how to interface between NMOS and CMOS/TTL and how to connect between NMOS inputs/outputs and external discrete circuitry. Most NMOS LSI chips are designed for TTL compatibility; however, there are some subtle points to be considered.

NMOS inputs

The input circuit of an *n*-channel MOS IC intended for use with TTL driving circuitry is shown in Figure 9.16. Q_1 is

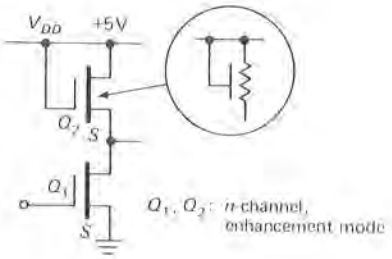


Figure 9.16. NMOS logic input circuit.

an inverter, and Q_2 is a small-geometry source follower providing pullup current (resistors take up too much space, so MOSFETs are universally used as drain loads); the alternative symbol shown for Q_2 is widely used. The threshold voltage of the input transistor is in the range of 1 to 1.5 volts in modern silicon-gate designs, so it can be driven directly from either

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

TTL or CMOS. In some older designs the threshold can be in the range of 2 to 3 volts, for which it is best to use a 1k–10k pullup when driving from TTL; as usual, no pullup is needed with a CMOS driver.

NMOS outputs

The output stage of 5 volt NMOS logic is shown in Figure 9.17. Q_1 is a switch, and Q_2 is a source follower. To put the output in the LOW state, Q_1's gate is brought to +5 volts, forcing the output below 0.5 volt even while sinking a few milliamps. The situation in the HIGH output state is dismal by comparison: At a minimum TTL HIGH output of +2.4 volts, Q_2 finds only 2.6 volts from gate to source, resulting in relatively high R_{ON}, with the situation rapidly deteriorating for higher output voltages.

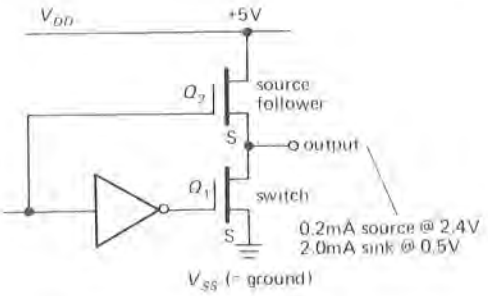


Figure 9.17. NMOS logic output circuit.

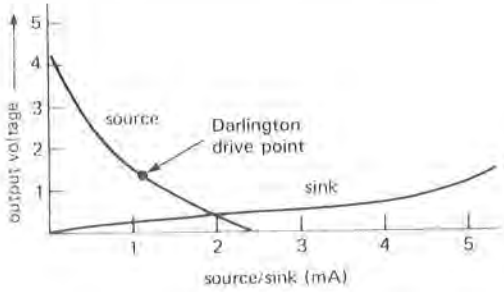


Figure 9.18. Typical NMOS output source and sink current capability.

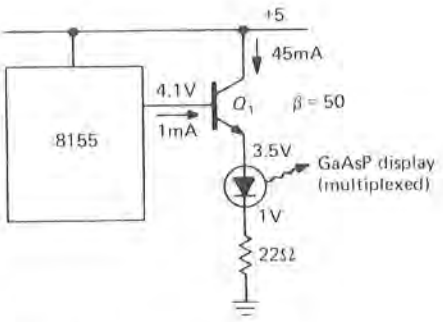


Figure 9.19

The graph in Figure 9.18 illustrates the situation. As a result, the NMOS output drive capability might be only 0.2mA (sourcing) at +2.4 volts output. This is fine for driving TTL inputs, but it is marginal for 5 volt CMOS (use a pullup, or interpose an HCT or ACT gate), and it is a disaster in a circuit like that shown in Figure 9.19.

To operate the LED with multiplexed-display current levels (25-50mA while ON), the NMOS output would have to put out about 1mA at +4.1 volts. That's impossible, since V_{GS} would then be only 0.9 volt, possibly even below the threshold voltage for the FET. Remember, too, that all 5 volt logic circuits should work with supply voltages as low as +4.5 volts to allow for ±10% tolerance of supply voltage. To drive LEDs (or other high-current devices) from NMOS, the circuits in Figure 9.20 are good.

In the first circuit, the NMOS output LOW sinks 2mA, driving the *pnp* transistor into hard conduction. In the second circuit, a Darlington *npn* transistor is switched ON by the small output current of the NMOS device while in the HIGH state. This circuit clamps the HIGH output two diode drops above ground, which might seem a bit unfriendly, but it turns out that NMOS outputs are designed so that they can be shorted to ground in this manner, with output currents controlled at small enough values so that they can drive

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

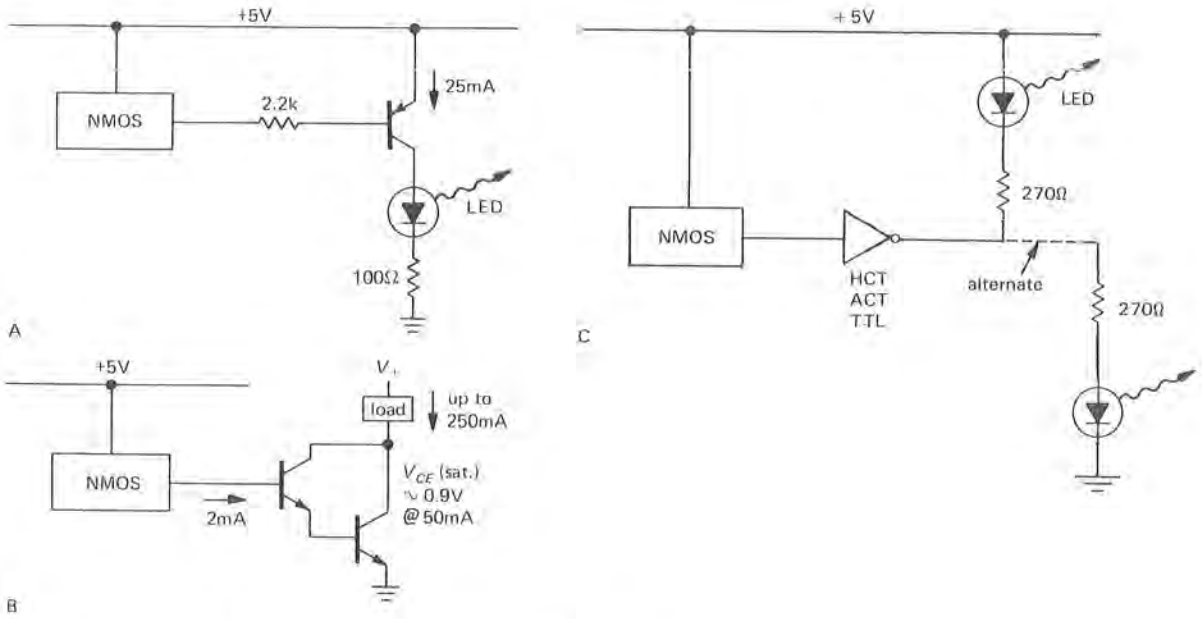


Figure 9.20. NMOS logic outputs driving loads.

the base of a grounded emitter Darlington without damage. A typical NMOS output would source 2mA into the Darlington's base at +1.5 volts, giving an output sink capability of 250mA at 1 volt with an IC like the 75492 hex Darlington array. The ULN series from Sprague has several hex and octal Darlington arrays in DIP packages.

Finally, you can always buffer NMOS outputs with an HCT or ACT (or even TTL) buffer, since NMOS outputs are fully compatible with those families' inputs. The buffered outputs can then sink current from a load; with HCT or ACT you can equally well *source* current, since the CMOS families have the same output drive capability whether sinking or sourcing.

9.10 Opto-electronics

In the last two chapters we have been using LED indicators and LED numeric display devices in various circuit contexts, as we needed them. LEDs belong to the general area of *opto-electronics*, which includes displays based on other technologies as well, notably liquid crystals, fluorescence, and gas discharge. It also includes optical electronics used for purposes other than indicators and displays: light-coupled isolators ("opto-isolators"), solid-state relays, position sensors ("interrupters" and "reflective sensors"), diode lasers, array detectors ("charge-coupled devices," CCDs), image intensifiers, and a variety of components used in fiber optics.

Although we will continue to conjure up miscellaneous magic devices as we need them, this seems like a good place to pull together the area of opto-electronics, since it is related to the logic interface problems we have just been discussing.

Indicators

Electronic instruments look nicer, and are more fun to use, if they have pretty little colored lights on them. LEDs have replaced all earlier technologies for this purpose. You can get red, yellow, and green

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

indicators, and you can get them in many packages, the most useful of which are (a) panel-mounting lights, and (b) printed-circuit board (PCB) mounting types. The catalogs present you with a bewildering variety of them, differing mostly in size, color, efficiency, and illumination angle. The latter deserves some explanation: A "flooded" LED has some diffusing stuff mixed in, so the lamp looks uniformly bright over a range of viewing angles; that's usually best, but you pay a price in brightness.

An LED looks electrically like a diode, with a forward drop of about 2 volts (they're built with gallium arsenide phosphide, which has a larger band gap, and hence a larger forward drop, than silicon). Typical flooded panel-type LED indicators look good at 10mA forward current; on a board inside an instrument you can usually get away with 2–5mA, particularly if you use a narrow-angle LED.

Figure 9.21 shows how to drive LED indicators. Most of the circuits are obvious, but note that bipolar TTL is poor at *sourcing* current, so you always arrange the circuit so that logic LOW turns on the LED; by comparison, CMOS families have symmetrical drive capability. NMOS circuits not only share the feeble sourcing of bipolar TTL but also tend to have rather limited current-sinking capability; it's best to interpose a buffer (an HCT gate is good), or perhaps a discrete MOSFET. Note also that some LED indicators come with internal current-limiting resistors (or even internal constant-current circuits) – with these you omit the external resistor.

You can get little arrays of indicators – sticks of 2, 4, or 10 LEDs in a row – designed for PCB mounting. The latter are actually intended for linear "bar-graph" readouts. They come in upright or right-angle mounting. You can also get panel-mounting indicators that combine a red LED and green LED in one uncolored package. These make an impressive panel, with lights changing color to indicate good/bad conditions.

We've used LED indicators from manufacturers such as Dialight, General Instrument, HP, Panasonic, Siemens, and Stanley. The latter specializes in lamps of unusually high efficiency; you can usually locate their exhibit at electronics shows by the dazed look (and incipient sunburn) of recent visitors.

Displays

A *display* means an opto-electronic device that can show a number ("numeric" display); a hexadecimal digit, namely 0–9 and A–F ("hexadecimal display"), or any letter or number ("alphanumeric display"). The dominant display technologies today are LEDs and LCDs (liquid-crystal displays). LCDs are the newer technology, with significant advantages for (a) battery-operated equipment, owing to its very low power dissipation, (b) equipment for use outdoors or in high ambient light levels, (c) displays that require custom shapes and symbols, and (d) displays with many digits or characters. LEDs, by comparison, are somewhat simpler to use, particularly if you only need a few digits or characters. They also come in three colors, and they look good in subdued light, where their good contrast makes them easier to read than LCD displays.

For displays of many characters – say a line or two of text – gas discharge ("plasma") display panels compete with LCDs, particularly if you care about clarity and contrast. They do require significant power, however, so LCD displays are usually preferred for battery-powered applications.

LED displays. Figure 9.22 shows the choices you have in LED displays. The original 7-segment display is the simplest and can display the digits 0–9 and the hex extension (A–F), albeit somewhat crudely (the hex letters are displayed as

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

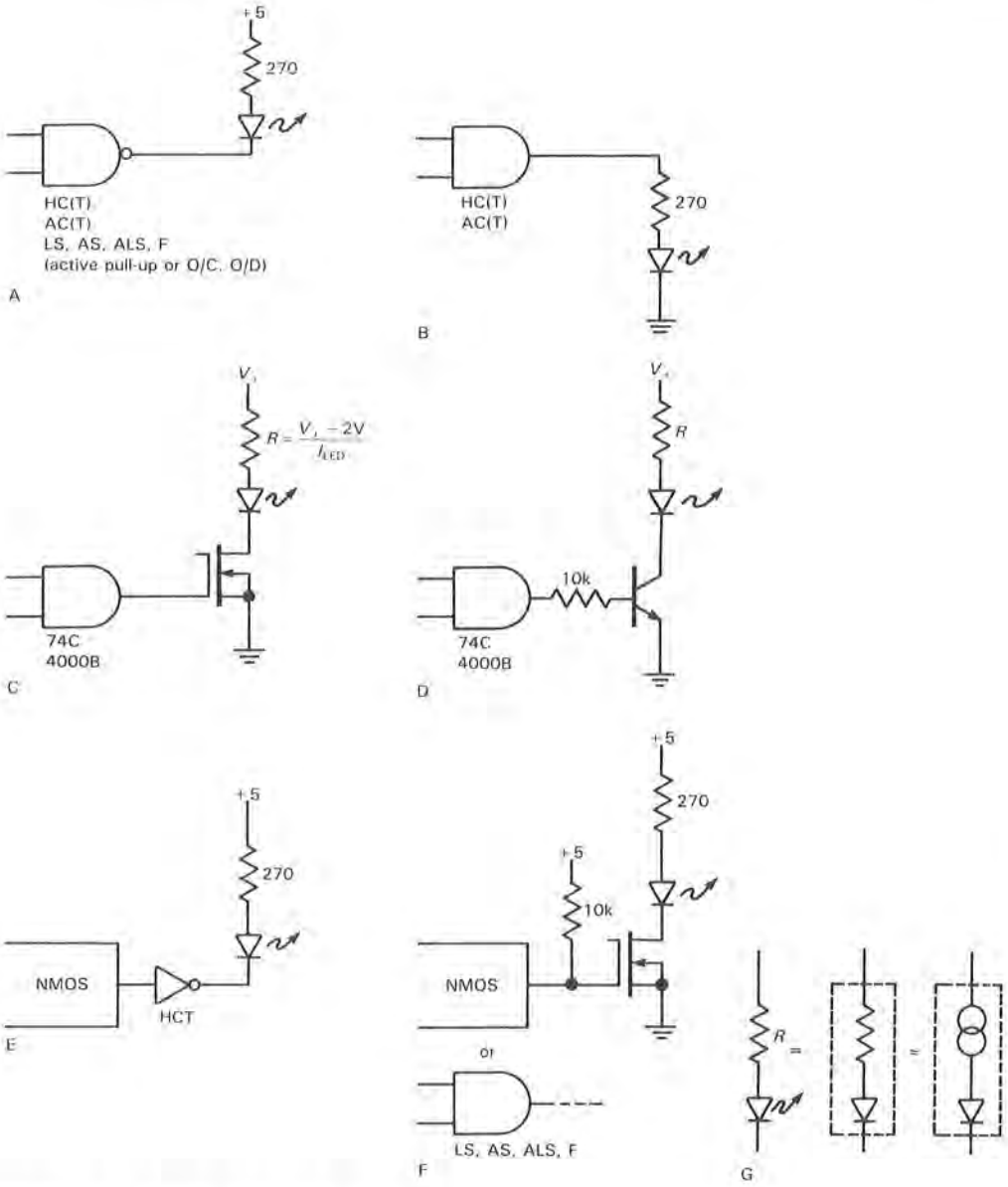


Figure 9.21. Driving LED indicators.

"AbcdEF"). You can get single-character 7-segment displays in many sizes, and in "sticks" containing 2, 3, 4, or 8 characters (generally intended to be "multiplexed" – the characters displayed one at a time in rapid sequence). Single-character displays bring out leads for the 7 segments and the common electrode; the two flavors are thus "common cathode" and "common anode." Multiple-character sticks bring each character's common electrode out, but tie the corresponding segments together, which is what you want for multiplexing.

Sixteen-segment and 5×7 dot-matrix displays are available in two varieties: "dumb" displays that bring out the

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

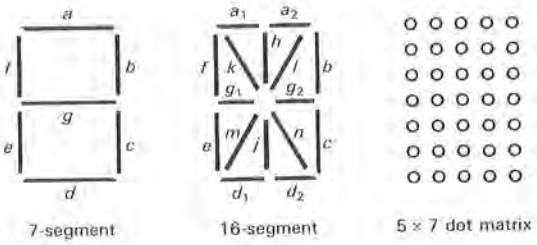


Figure 9.22

segments and common leads (the same as with 7-segment displays) and "smart" displays that do the hard work of decoding and driving for you.

Rather than speak further in generalities, let's look at some examples (Fig. 9.23). The first circuit shows how to drive a single-digit 7-segment common-cathode LED display. The 'HC4511 is a "BCD-to-7-segment latch-decoder-driver," able to source about 15mA while holding its active outputs at +4.5 volts. The series resistors make sure the segment current is limited to that value, with a forward diode drop of 2 volts. You can get arrays of equal-value resistors in convenient SIPs (single in-line package).

You only need a single decoder-driver chip, even if you are displaying multiple digits, as long as you multiplex the display, i.e., illuminate only one displayed digit at a time. Figure 9.23B shows the idea, in this case using an LSI 4-digit counter chip with built-in 7-segment multiplexed drivers. The 74C925 asserts its segment drivers (active HIGH, with plenty of drive capability) for each digit in turn, simultaneously asserting an active HIGH on the corresponding digit output A–D. The rest of the circuitry is self-evident, except perhaps for the unsavory manner in which the digit outputs are pinned a diode drop above ground; luckily, the 74C925 specifies proper operation with this circuit, since the digit outputs are buffered and current-limited.

Figure 9.24A shows how to drive a single hexadecimal display, implemented with a 5×7 dot matrix. The HP 5082-7340 is an example of a "smart" display, with built-in latch, decoder, and driver. All you have to do is assert the 4-bit data, wait at least 50ns, then bring the latch enable HIGH. If you don't want to use its latch, just keep the enable LOW. In Figure 9.24B we've shown one of Siemens' "intelligent" (smarter than "smart"?) displays, in this case a 4-character stick that uses 16-segment displays. This display is intended to look like memory to a microprocessor, something we'll learn about in the next two chapters. To make a long story short, you just assert any 7-bit character and its position ("address," 2 bits), then assert WR′ (write) while making sure the chip is enabled. The data then get stored

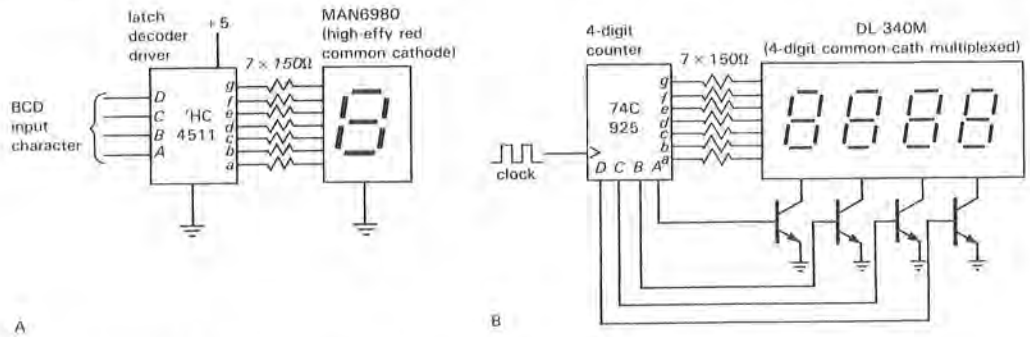


Figure 9.23. Driving 7-segment LED displays.
A. Single-digit.
B. Multiplexed.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

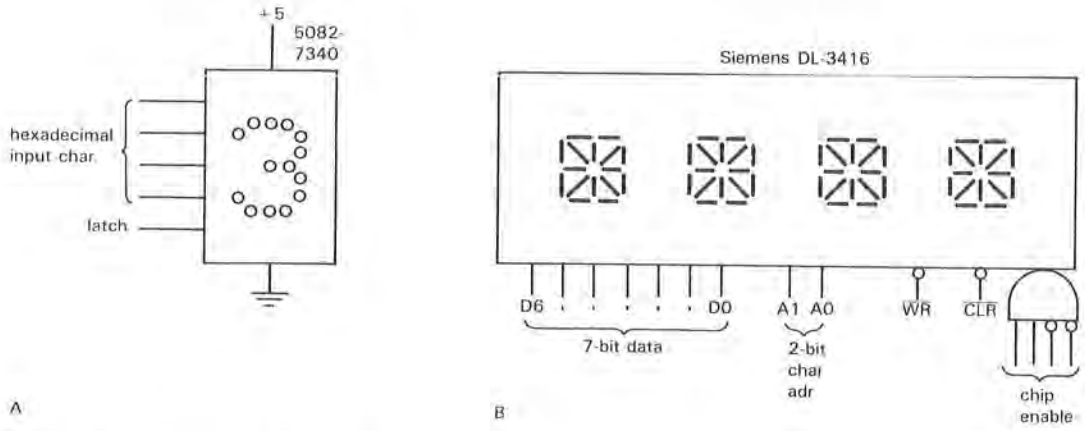


Figure 9.24. Integrated displays.
A. Single-character, dot matrix.
B. 4-character, 16-segment, addressable.

internally, and the corresponding display position changes to the new character. Figure 9.25 shows the character set that can be displayed.

If you want to use a dumb display (perhaps it isn't available with intelligence), but you're spoiled by the simplicity of these intelligent displays, you can simply interpose a chip like the 8-digit Intersil ICM7218/28, which looks like memory to the microprocessor, and which drives a dumb LED display stick with appropriate segment and digit drives. Another alternative is to let the microprocessor do all the work of figuring out which segments and digits to drive, using bits of its "parallel ports" to drive the appropriate lines. This will make more sense to you after you've digested the two microprocessor chapters (Chapters 10 and 11).

LCD and gas-discharge displays. Much of what we have said about LED displays applies also to LCD displays. However, there are some important differences. For one thing, LCDs must be driven by an *ac* waveform; otherwise their liquid guts are ruined. So LCD driver chips usually have

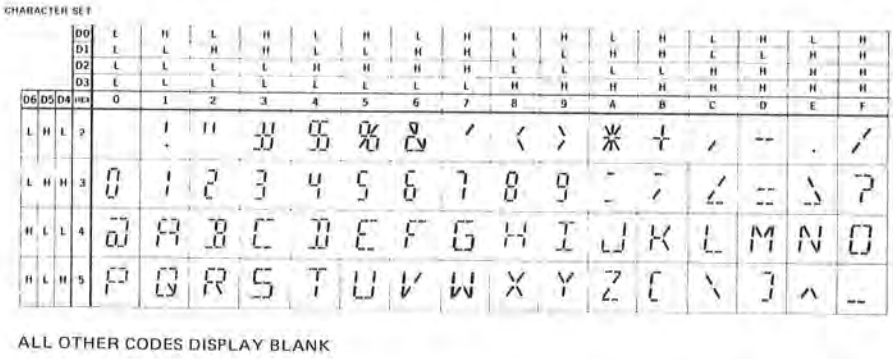


Figure 9.25. Display codes for Siemens DL-3416 16-segment display. (Courtesy of Siemens Components, Inc.)

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

some way to generate a square-wave segment drive, synchronized to the LCD backplane waveform. An example is the 'HC4543, the LCD cousin of the 'HC4511 LED latch/decoder/driver.

Another difference with LCD displays is that you don't often see single-character displays. Instead, they come in rather large panels that can display a line or two of text. Luckily, the manufacturers realize how complicated things can get, and therefore they provide complete displays that are more than intelligent, they're positively at the genius level. In general, you're talking to these things with a microprocessor, so (as with the display in Figure 9.24) the display is configured to look like a block of memory. Whatever you write gets displayed. Some of the fancier displays go even further, with the ability to store several messages and communicate via serial ports. Look in the EEM (see bibliography) for manufacturers.

Gas-discharge displays feature those handsome red-orange characters you see on some higher-priced portable computers. They require high-voltage drivers, and the manufacturers generally provide the driver circuitry. You can get single- and multiple-digit displays, and you can get large multi-character panels complete with memory and convenient interface. Examples of the latter are the multi-line displays by Cherry, with battery-backed memory that can hold 512 messages, interleave real-time data, and let you edit its memory. You probably shouldn't think of these as *displays*, but as computers that happen to have a display attached!

Opto-couplers and relays

An LED emitter, combined with a photodetector in close proximity, forms a very useful object known as an *opto-coupler* or *opto-isolator*. In a nutshell, opto-couplers let you send digital (and sometimes analog) signals between circuits with separate grounds. This "galvanic isolation" is a good way to prevent ground loops in equipment that drives a remote load. It is essential in circuits that interact with the ac power mains. For example, you might want to turn a heater on and off from a digital signal provided by a microprocessor; in this case you would probably use a "solid-state" relay, which consists of an LED coupled to a high-current triac. Some ac-operated switching supplies (e.g., the supply used in the IBM PC-AT) use opto-couplers for the isolated feedback path (see Section 6.19). Similarly, designers of high-voltage power supplies sometimes use opto-couplers to get a signal up to a circuit floating at high voltage.

Even in less exotic situations you can take advantage of opto-isolators. For example, an opto-FET lets you switch analog signals with no charge injection whatsoever; the same goes for sample-and-hold circuits and integrators. Opto-isolators can keep you out of trouble when driving industrial current loops, hammer drivers, etc. Finally, the galvanic isolation of opto-isolators comes in handy in high-precision or low-level circuits: For example, it is difficult to take full advantage of a 16-bit analog-to-digital converter, because the digital output signals (and noise on the digital ground to which you connect the converter's output) get back into the analog front end. You can extricate yourself from "noise city" with optical isolation of the digital half.

Opto-couplers typically provide 2500 volts (rms) isolation, 10^{12} ohms insulation resistance, and less than a picofarad coupling between input and output.

Before looking at actual opto-couplers, let's take a quick look at photodiodes and phototransistors. Visible light causes ionization in silicon, producing charge pairs in the exposed base region; this mimics exactly the effect of an externally applied base current. There are two ways to use a phototransistor: (a) You can use it as

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

a *photodiode*, connecting only to the base and collector leads; in that case the detected photocurrent will typically be a few percent of the LED drive current. A photodiode generates its photocurrent whether or not you apply a bias voltage; thus you can hook it directly into an op-amp summing junction (a virtual short circuit), or you can back-bias it (Fig. 9.26A,B). (b) If you let the photodiode current act as a base current, you get normal transistor current multiplication, with a resulting I_{CE} that is typically 100 times larger; in this case you must bias the transistor, as in Figure 9.26C. You pay for the increased current with slower response, because of the open base circuit. You can add a resistor from base to emitter to improve the speed; however, this produces a threshold effect, since the phototransistor doesn't begin to conduct until the photodiode current is large enough to produce a V_{BE} across the external base resistor. In digital applications the threshold can be useful, but in analog applications it is an undesirable nonlinearity.

Figure 9.26D–S shows typical examples of nearly every kind of opto-coupler you are likely to encounter. The earliest (and simplest) is typified by the 4N35, an LED-phototransistor pair with 40% (min) current transfer ratio (CTR) as a phototransistor, and sluggish 5μs turn-off time (t_{OFF}) into a 100 ohm load. The figure shows how to use it: A gate and resistor generate current-limited 8mA drive, and a relatively large collector resistor guarantees saturated switching of the output between logic levels. Note the use of a Schmitt-trigger inverter, a good idea here because of the long switching times. You can get LED-phototransistor pairs with CTRs of 100% or more (e.g., the MCT2201, with CTR=100% min), and you can get LED-photo-Darlingtons, as shown; they're even slower than phototransistors! To get improved speed, the manufacturers sometimes use separate photodiode and transistor, as shown in the 6N136 and 6N139 opto-transistor and opto-Darlingtons.

These opto-couplers are nice, but somewhat annoying to use because you have to supply discrete components at both input and output. Furthermore, the input loads ordinary logic gates to their maximum capability, and the passive pullup output suffers from slow switching and mediocre noise immunity. To remedy these deficiencies the silicon wizards bring us "logic" opto-couplers. The 6N137 in Figure 9.26I goes halfway, with diode input and logic output; you still need plenty of input current (specified as 6.3mA, min, to guarantee output switching), but you get clean logic swings at the output (albeit open-collector), and speeds to 10Mbit/sec. Note that you must supply +5 volts to the internal output circuitry. The newer 74OL6000 series from General Instrument (Fig. 9.26J) does what you really want: It accepts logic-level inputs, and produces logic-level outputs, with both totem-pole and open-collector types available; these opto-couplers operate to 15Mbit/sec. Because of the internal logic circuitry at both input and output, both sides of the chip require logic supply voltages.

Figure 9.26 continues with some variations on the LED-phototransistor theme. The IL252 hooks a pair of LEDs back-to-back, so you can drive it with ac. The IL11 uses a long isolation gap (and package) to obtain 10kV rms voltage standoff, compared with the usual 2.5kV rms value for all the other couplers shown. The H11C4 is an opto-*SCR*, useful for switching high voltages and currents. The MCP3023 replaces the unidirectional SCR with a *triac*, which is a bidirectional SCR; it can drive an ac load directly, as in Figure 9.15N. When driving ac loads, it's best to switch on the load during a zero crossing of the ac waveform, in order to avoid putting spikes onto the power line. This is easily done, with an opto-triac containing

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

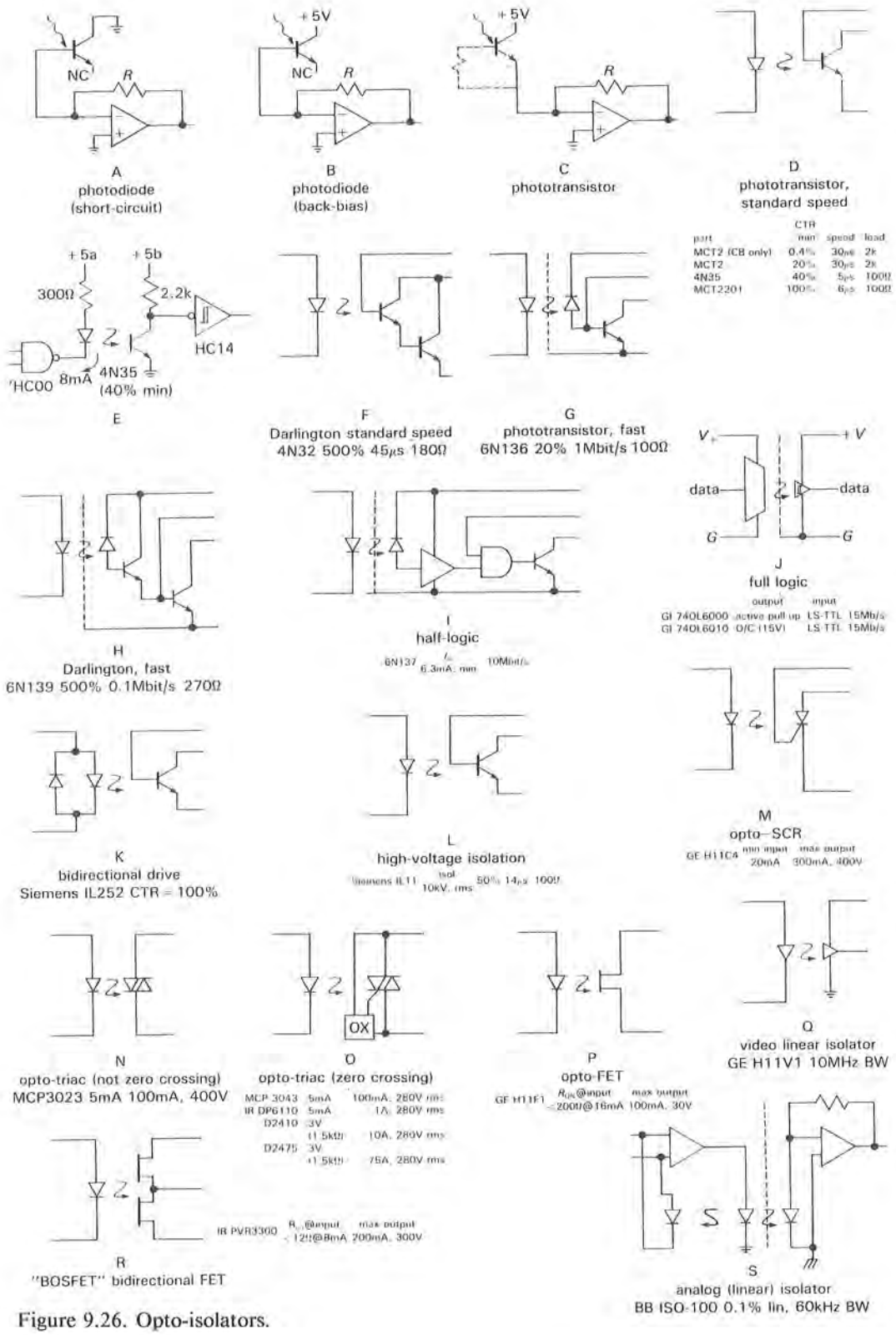


Figure 9.26. Opto-isolators.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

internal "zero-voltage-switching" circuitry (which blocks triac drive until the next zero crossing); the small MCP3043 shown uses such circuitry, as do the higher-current "solid-state relays" shown. The DP6110 "ChipSwitch" from IR comes in a 16-pin DIP (with all but 4 pins missing), while the mighty D2410 and D2475 are packaged in 1.75″×2.25″×1" power modules intended for mounting to a heat sink.

The remaining opto-couplers in Figure 9.26 can be used for linear signals. The H11F series of opto-FETs can be used as an isolated variable resistor or isolated analog switch. There are no problems of compatible voltage levels, SCR latchup, or charge injection. You might use one of these in a sample/hold or integrator. The PVR series of "BOSFETs" are similar, but with a pair of power MOSFETs hooked in series as the output element. They are intended primarily to switch ac loads directly, in the manner of an opto-triac. The H11V1 is a linear video isolator, with 10MHz bandwidth. And the ISO-100 from Burr-Brown is a clever analog isolation element, in which the LED couples to *two* matched photodiodes; one is used in a feedback loop to linearize the coupled response to the second photodiode.

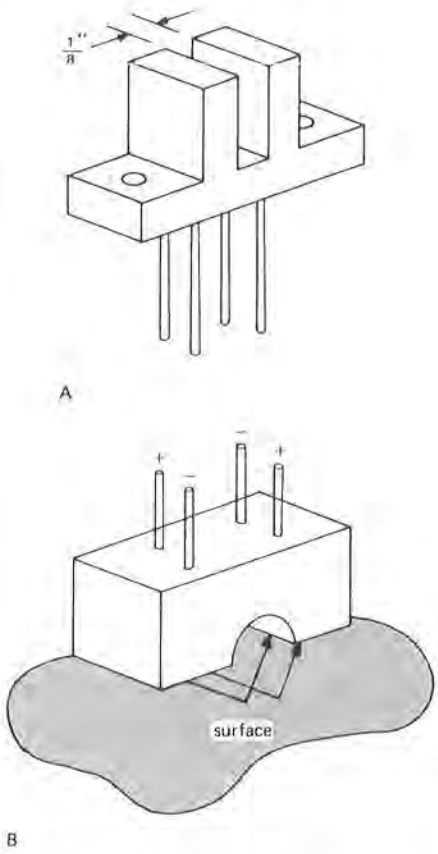


Figure 9.27. A. Optical interrupter. B. Reflective object sensor.

□ *Interrupters*

You can use LED-phototransistor technology to sense proximity or motion. An "optical interrupter" consists of an LED coupled to a phototransistor across a 1/8 inch slot. It can sense the presence of an opaque strip, for example, or the rotation of a slotted disk. An alternative form has the LED and photodetector looking in the same direction, and it senses the presence of a reflective object nearby (most of the time, anyway!). Take a look at Figure 9.27. Optical interrupters are used in disk drives and printers, to sense the end of travel of the moving assembly. You can get optical "rotary encoders" that generate a quadrature pulse train (two outputs, 90° out of phase) as the shaft is rotated. These provide a nice alternative to resistive panel controls (potentiometers). See Section 11.09. In any application where you're considering an optical interrupter or reflective sensor, take a look at Hall-effect sensors as an alternative; they use solid-state magnetic-field sensors to indicate proximity. They're commonly used in automobile ignition systems as an alternative to mechanical breaker points.

□ *Emitters and detectors*

We've already mentioned LEDs, both for displays and opto-couplers. A recent development in opto-electronics is the

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

availability of the inexpensive solid-state diode *laser*, a true coherent source of light, unlike the diffuse LED. You can see one of these when you open the top door of a portable compact-disk player. Diode lasers cost about $20, sold by the companies that manufacture consumer electronics (Matsushita, Mitsubishi, Sharp, and Sony). A typical diode laser generates 10mW of light output at 800nm (invisible, in the "near infrared") when powered by 80mA through its forward diode drop of 2 volts. The output beam emerges directly from a tiny spot on the GaAlAs chip, diverging with an angle of about 10°–20°; it can be collimated with a lens to form a parallel beam or a very small focal spot. LED lasers are used extensively in optical fiber transmission.

Another recent emitter technology is embodied in high-density linear LED arrays, 300 emitters/inch or more, intended for LED printers. If the semiconductor technology is successful, these are likely to replace laser printers, because they are simpler and more reliable and ultimately of higher resolution.

In the detector arena there are several alternatives to the simple photodiode and phototransistor that we discussed earlier, particularly when speed or sensitivity is important. We will discuss PIN diodes, CCDs, and intensifiers in Section 15.02.

DIGITAL SIGNALS AND LONG WIRES

Special problems arise when you try to send digital signals through cables or between instruments. Effects such as capacitive loading of the fast signals, common-mode interference pickup, and "transmission-line" effects (reflections from impedance mismatching, see Section 13.09) become important, and special techniques and interface ICs are often necessary to ensure reliable transmission of digital signals. Some of these problems arise even on a single circuit board, so a knowledge of digital transmission techniques is generally handy. We will begin by considering on-card problems. Then we will go on to consider the problems that arise when signals are sent between cards, on data buses, and finally between instruments via twisted-pair or coaxial cables.

9.11 On-board interconnections

Output-stage current transient

The push-pull output circuit for TTL and CMOS ICs consists of a pair of transistors going from V_+ to ground. When the output changes state, there is a brief interval during which both transistors are ON; during that time, a pulse of current flows from V_+ to ground, putting a short negative-going spike on the V_+ line and a short positive spike on the ground line. The situation is shown in Figure 9.28. Suppose that IC_1 makes a transition, with a momentary large current from +5 to ground along the paths as indicated [with 74F*xx* or 74AC(T)*xx* circuits the current might reach 100mA]. This current, in combination with the inductance of the ground and V_+ leads, causes short voltage spikes relative to the reference point, as shown. These spikes may be only 5ns to 20ns long, but they can cause plenty of trouble: Suppose that IC_2, an innocent bystander located near the offending chip, has a steady LOW output that drives IC_3 located some distance away. The positive spike at IC_2's ground line appears also at its output, and if it is large enough, it gets interpreted by IC_3 as a short HIGH spike. Thus, at IC_3, some distance away from the troublemaker IC_1, a full-size bona fide logic output pulse appears, ready to mess up an otherwise well-behaved circuit. It doesn't take very much to toggle or reset a flip-flop, and this sort of ground-current spike can do the job nicely.

The best therapy for this situation consists of (a) using hefty ground lines throughout the circuit, even to the extent of

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

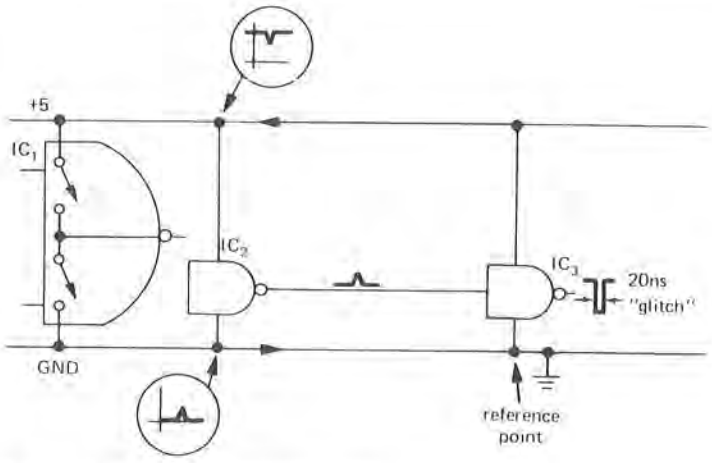


Figure 9.28. Ground current noise.

using a large "ground plane" (one whole side of a double-sided printed-circuit board devoted to ground), and (b) using bypass capacitors liberally throughout the circuit. Large ground lines mean smaller current-induced spikes (lower inductance and resistance), and bypass capacitors from V_+ to ground sprinkled throughout the circuit mean that current spikes travel only over short paths, with the reduced inductance resulting in much smaller spikes (the capacitor acts as a local voltage source, since its voltage does not change appreciably during the brief current spikes). It is best to use a 0.05μF to 0.1μF capacitor near each IC, although one capacitor for every two or three ICs may suffice. In addition, a few larger tantalum capacitors (10μF, 20V is a good value) scattered throughout the circuit for energy storage is a good idea. Incidentally, bypass capacitors from power-supply lines to ground are recommended in any circuit, digital or linear. They help make the supply lines low-impedance voltage sources at high frequencies, and they prevent signal coupling between circuits via the power supply. Unbypassed power-supply lines can cause peculiar circuit behavior, oscillations, and headaches.

Spikes caused by driving capacitive loads

Even with the supplies bypassed, your problems aren't over. Figure 9.29 shows why. A digital output sees the stray wiring capacitance and the input capacitance of the chip it drives (5–10pF, typically) as part of its overall load. To make a fast transition between states it must sink or source a large current into such a load, according to $I = C(dV/dt)$. For instance, consider a 74ACxx chip (5V output swing in 3ns) driving a total load capacitance of 25pF (equivalent to three or four logic loads connected with short leads). The current during the logic transition is 40mA, nearly the maximum output capability of the driving chip! This current returns through ground (HIGH-to-LOW transition) or the +5 volt line (LOW-to-HIGH transition), producing those cute little spikes at the receiving end, as before. [Just to get an idea of the magnitude of this effect, consider the fact that wiring inductance is roughly 5nH/cm. An inch of ground wire carrying this logic transition current would have a spike of $V = L(dI/dt) = 0.2$V.] If the chip happened to be an octal buffer, with simultaneous

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

transitions on a half dozen outputs, the ground spike would be over a volt; see Figure 8.95. A similar (though generally smaller) ground spike is generated near the driven chip, where the drive-current spikes return to ground through the input capacitance of the driven device.

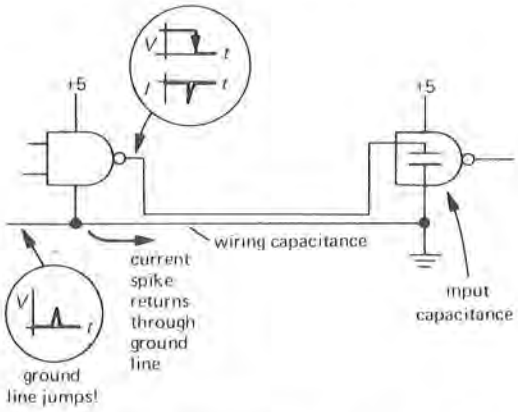


Figure 9.29. Capacitive-load ground current noise.

In a synchronous system, with a number of devices making output transitions simultaneously, the noise-spike situation can become so serious that the circuit will not function reliably. This is especially true in a large printed-circuit card, with long interconnections and long ground runs. The circuit may fail only occasionally, when a whole group of data lines unluckily happen to make a simultaneous HIGH-to-LOW transition, generating a momentary very large ground current. This kind of pattern sensitivity is characteristic of noise-induced error and is a good reason for running extensive memory tests on microprocessor systems (where you typically have 16 data lines and 24 address lines bouncing around in crazy patterns).

The best design approach is to use massive ground runs (for low inductance), preferably in the form of an internal ground-plane layer on a multilayer circuit board (see Chapter 12), or at least a perpendicular "gridded" arrangement of grounds on both sides of a simpler two-sided board. Copious use of bypass capacitors is mandatory. These problems are much less severe with high-voltage CMOS (because of slow edge times); conversely, they are most severe with F, AS, and AC(T) logic families. In fact, the AC(T) families are so prone to dynamic current spikes that some manufacturers (led by TI) have abandoned the popular "corner-pin" power/ground arrangement in favor of a "center-pin" arrangement with lower internal lead inductance; to improve things still further, they use up to four adjacent pins for a lower-inductance ground. Because of these noise problems, it is best not to use a faster logic family than you need; that is why we recommend HC logic, rather than AC, for general-purpose use.

9.12 Intercard connections

With logic signals going between circuit cards, the opportunities for trouble multiply rapidly. There is greater wiring capacitance, as well as longer ground paths through cables, connectors, card extenders, etc., so the ground spikes induced by drive currents during logic transitions are generally larger and more troublesome. It is best to avoid sending clocking signals with large fanout between boards, if possible, and the ground connections to the individual cards should be robust. If clocking signals are sent between boards, it is important to use a gate as input buffer on each board. In extreme cases it may be necessary to use line driver and receiver chips, as we will discuss later. In any case, it is best to try to keep critical circuits together on one card, where you can control the inductance of the ground paths and keep wiring capacitance at a minimum. The problems you will encounter in sending fast signals around through several cards should not

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

be underestimated; they can turn out to be the major headache of an entire project!

□ 9.13 Data buses

Where many subcircuits are connected together by a data bus (more on this in Chapters 10 and 11), the sort of problems already mentioned become more severe. In addition, a new factor comes into the picture: transmission-line effects due to the length and inductance of the signal lines themselves. With the fastest ECL chips ("ECL III," "ECL100K," with rise times of less than 1ns), these effects are so severe that signal runs of more than 1 inch must be treated as transmission lines and properly terminated!

The best approach with data buses of any substantial length (a foot or more) is probably the use of a "motherboard" with ground plane. As we will explain in Chapter 12, a motherboard is simply a printed-circuit board containing a row of printed-circuit edge connectors to accept the individual circuit cards that make up the logic circuit. Motherboards are economical solutions to the problem of interconnecting cards, and if they are properly laid out, they are electrically superior as well. Wires that run close to a ground plane have lower inductance and less tendency to couple capacitively to adjacent signal lines, so a good way to arrange a simple motherboard is with all the signal lines on one side, with the other side being a solid ground plane (though double-sided printed-circuit boards are standard, multiple-layer boards are being increasingly used for complicated circuits).

One last note on this problem: In desperation you may be tempted to use the common trick of putting a capacitor directly across the input of a gate driven from a long line, when transmission-line effects like "ringing" or ground spikes are driving you up the wall. In spite of the fact that we have done it ourselves, we don't recommend this inelegant fix, since it only compounds the problem of large ground currents during logic transitions (Section 9.11).

□ *Bus termination*

With buses of substantial length, it is common to "terminate" the signal lines at the far end with a resistive pullup or pulldown. As we will discuss in Chapter 13, long pairs of wires or coaxial cables have a "characteristic impedance" Z_0. For a cable terminated with that impedance (which is always a resistance), a signal traveling down the cable will be entirely absorbed, with no reflection whatsoever. Any other value of termination, including an open circuit, will produce reflected waves, with amplitude and phase depending on the impedance mismatch. The sorts of conductor width and spacing used on printed-circuit boards give a characteristic impedance in the neighborhood of 100 ohms, which is also close to the characteristic impedance of twisted pairs made from ordinary insulated wire of 24 gauge or so.

A popular method of terminating a TTL bus is to use a voltage divider from +5 volts to ground. In that way the logic HIGH is kept around +3 volts, which means that less swing (and therefore less current into the load capacitance) is necessary during logic transitions. The combination of a 180 ohm resistor to +5 and a 390 ohm resistor to ground is typical (Fig. 9.32). Another method that works well with either TTL or CMOS is to use an ac termination, consisting of a series resistor/capacitor from the data line to ground (Fig. 9.30). The resistor value should be close to the characteristic impedance of the bus (typically 100Ω); the capacitor value should be chosen for a low capacitive reactance at a frequency equal to the inverse of the edge time (100pF generally works well).

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

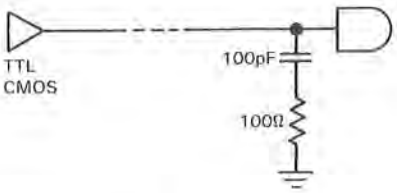


Figure 9.30. ac termination.

Bus drivers

When driving bus lines of substantial length or fanout, it is necessary to use special logic functions with high current capability. Here are the most popular types:

Type	Description
'365–8	hex 3S
'1004/34	hex 2S
'11004/34	hex 2S broadside
'240–4	octal 3S
'540–1	octal 3S broadside
'827–8	10-bit 3S broadside

Family	Sink/source (mA)
LS	24/15
AS, F	64/15
AS1*xxx*	48/48
HC(T)	6/6
AC(T)	24/24
Am298*xx*	48/24

"Broadside" means that the chip's pins are arranged with corresponding inputs and outputs on opposite sides. There are also *transceiver* chips of comparable drive capability; those can buffer data in either direction by using a parallel pair of three-state buffers back-to-back for each data line, with a "direction" input determining which way data are going. See Tables 8.4 and 8.5 for additional device types.

9.14 Driving cables

You can't run digital signals from one instrument to another by simply stringing a single conductor between them, because such an arrangement is prone to pickup of interference (as well as generating interference of its own). Digital signals are generally piped through coaxial cables, twisted pairs, flat ribbon cables (sometimes with ground plane or shield), multiwire bundled cables, and, increasingly, fiber-optic cables. We will see coaxial cables (affectionately called "coax") again in Chapter 13 in connection with radiofrequency techniques; here we would like to describe some of the methods used to send digital signals between boxes of electronics, since these methods constitute an important part of digital interfacing. In most cases there are special-purpose driver/receiver chips available to make your job easier.

RS-232

For relatively slow transmission (thousands of bits per second) through simple multiwire cables, the popular RS-232C (or newer RS-232D) signal standard is convenient. It specifies bipolarity levels of ± 5 volts to ± 15 volts (positive and negative supply voltages are needed for the driver, but not usually for the receiver), and the receivers usually allow control of hysteresis and response time for particularly noisy environments. With RS-232 you can use a multiwire cable without any special shielding, because the drivers are deliberately limited to a maximum slewing rate of $30V/\mu s$ to minimize cross-talk. In addition to the original TTL-compatible quad driver/receiver pair (the 1488/1489), there is now a selection of improved chips, including low-power versions (LT1032, 1039, and MC145406, see Section 14.17) and versions that operate from a single $+5$ volt supply (MAX-232 and LT1130 series, LT1080); the latter have on-board flying-capacitor voltage converters to generate the necessary negative voltage. Figure 9.31 shows a typical setup.

RS-232 is used extensively for communication between computers and terminals,

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

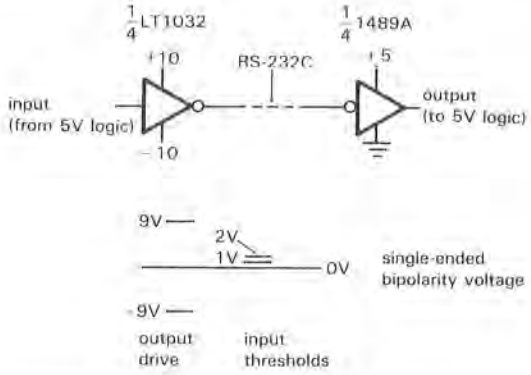


Figure 9.31. RS-232 high-noise-immunity cable driver and receiver.

at standardized data rates ranging from 110 to 38,400 bits per second. The full standard even specifies the pin connections to be used with a 25-pin type D subminiature connector and is usually used for transferring ASCII-encoded data, as we will discuss in Section 10.19.

Direct 5 volt logic drive

Just as with data buses, it is possible to drive lines of moderate length directly with logic levels; in general, gates of high current capability should be used (see list under "bus drivers," above). Figure 9.32 shows some methods. In the first circuit, a buffer (which could be open-collector) drives a terminated line, with a TTL Schmitt trigger as a receiver for improved noise immunity. If noise is a serious problem, the *RC* slowdown network shown in the second circuit can be used, with the *RC* time constant (and the transmitted bit rate!) adjusted to give good noise immunity; in this circuit the Schmitt trigger is essential. In the last circuit a robust CMOS buffer drives a line with ac termination and a CMOS Schmitt-trigger receiver.

Direct logic drive will work over modest lengths (say 10 feet) of twisted-pair, flat

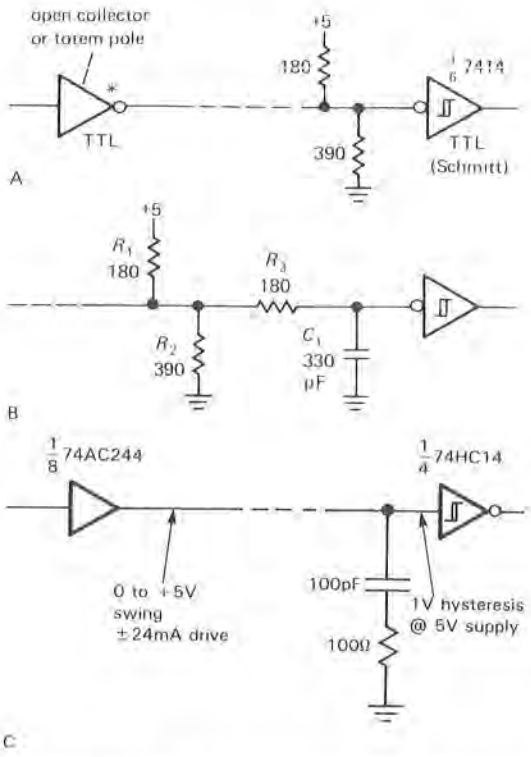


Figure 9.32. Use of termination with logic-level drive.

ribbon cable, or coax. Because of the fast edge times, capacitive pickup on adjacent lines is a problem; the usual cure is to alternate ground lines (ribbon cable), or to pair signal lines with ground (twisted-pair). This signal pickup problem makes direct logic drive with bundled multiwire cables almost hopeless. In the next section we've got some interesting scope photos that illustrate the problem, and another effective cure, namely *differential* logic drive.

Important note: *Never* attempt to drive long lines from the output of unbuffered clocked devices (flip-flops, monostables, counters, some shift registers); the capacitive loading and transmission-line effects can cause erratic or erroneous behavior. A "buffered" device interposes output drivers between the internal registers and

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

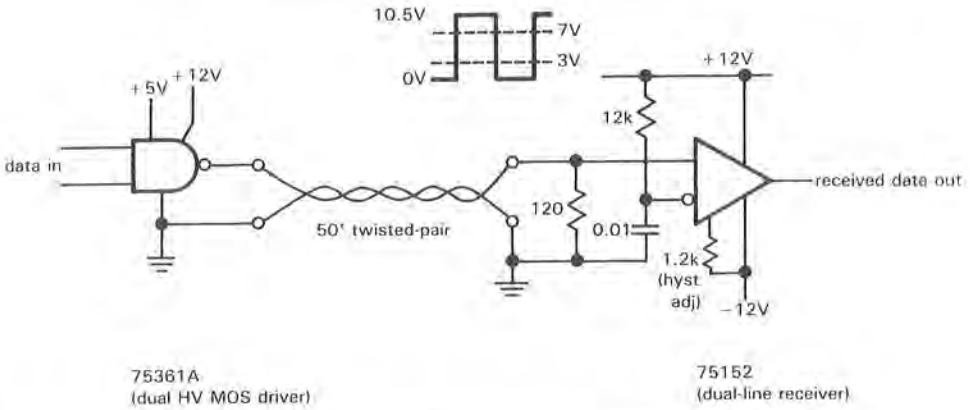


Figure 9.33. High-voltage cable driver improves noise immunity.

the output pins and thus, because the chip doesn't see the (degraded) signals actually present on the output lines, does not have this problem.

□ *High-voltage logic drive*

When you use direct logic drive to send signals through cables, you can improve the noise immunity by increasing the signal swing. Figure 9.33 shows an example, using the 75361A "TTL-to-MOS driver" as a generator of 12 volt logic swings for a twisted-pair line. The receiver is a 75152, which lets you set the input threshold (the input resistance is about 9k, hence the 12k biasing resistor to set the threshold at +5 volt) and the hysteresis (here set to ±2 volt). The 120 ohm line termination matches the characteristic impedance of the twisted-pair.

□ *Trapezoidal drive*

National makes a line driver/receiver (DS3662; DS3890 series) with controlled driver transition time combined with controlled receiver response time, to reduce the problem of capacitive coupling from adjacent lines. It is meant to drive lines terminated with the network of Figure 9.32.

Differential drive; RS-422

Much better noise immunity can be obtained by using differential signals, i.e., Q and Q' on a twisted-pair, with a differential receiver (Fig. 9.34). In this example, paired TTL inverters drive a terminated twisted-pair with true and complemented signals, and a 75115 differential line receiver regenerates clean TTL levels from the mess. We selected bipolar TTL drivers, rather than CMOS, because of their greater resistance to damage from static electricity, and to SCR latchup from transmission-line reflections. This scheme gives very good common-mode rejection and generates clean logic levels from line signals that can look just awful. The waveform shown gives an idea of what you might see on the individual signal lines in a relatively clean system; the individual signals tend to acquire bumps and wiggles, although they generally remain monotonic through the transition (they don't reverse direction).

The 75115 is an example of a line receiver with adjustable response time; the 75152, another differential receiver, allows control of hysteresis instead. For peace of mind, it is a nice idea to use a receiver with hysteresis (and adjustable time constant), given the bizarre waveforms these receivers are called on to recognize.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

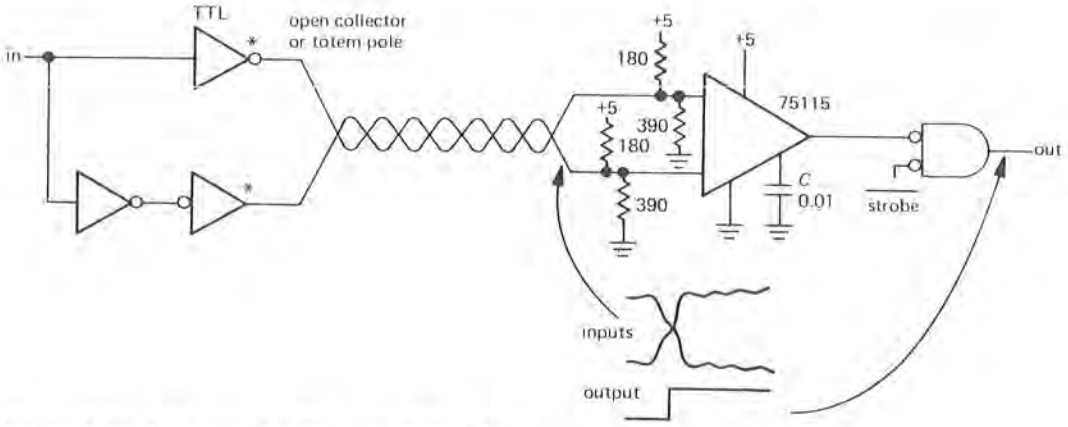


Figure 9.34. High-speed differential TTL cable driver and receiver.

□ *Current-sinking drivers.* Chips like the 75S110 and MC3453 have switched current-sinking outputs, which can be used as single-ended outputs or in a differential mode, as in Figure 9.35. The 75107 is a companion differential receiver chip, normally used with a termination as shown. Several drivers can share a common differential line in a "party line" configuration, since their outputs can be disabled via a "three-state" enable; in that case the terminations at each driver are omitted, and moved to the end of the line farthest from the receiver.

In our experience, differential current-sinking drivers allow really impressive data rates, probably because the high-impedance current-source drive ensures that the cable can be terminated in its characteristic impedance, for both states of the driver. The data sheets claim rates of more than 1 megabit per second (Mb/s) over line lengths of 2000 feet, and rates up to 10Mb/s over lengths of several hundred feet or less.

The actual scope photos in Figure 9.36 show how effective differential current-sinking drive can be in eliminating problems of common-mode noise. In the example shown, a 50mV pp signal is contaminated by 4 volts pp of common-mode noise.

RS-422/423. This is a data-transmission standard intended to supplant the popular RS-232 and is intended primarily for use with twisted-pair or flat cable. It provides for both unbalanced ("RS-423,"100kb/s max) and balanced

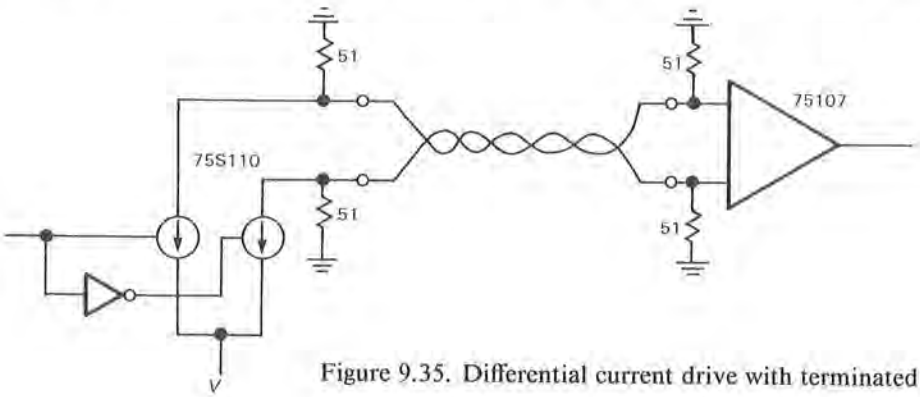


Figure 9.35. Differential current drive with terminated receiver.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

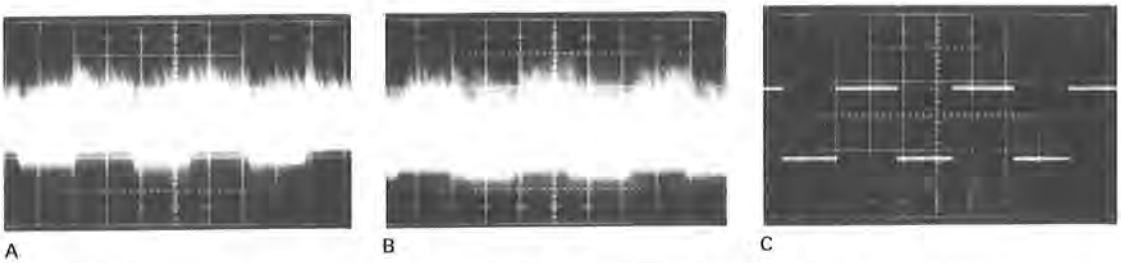


Figure 9.36. Scope photos showing excellent noise immunity of differential data transmission (75108 differential receiver). (Courtesy of Texas Instruments, Inc., Dallas, Texas.)
A. (+) receiver input.
B. (−) receiver input.
C. Receiver output.

("RS-422," 10Mb/s max) schemes. In the unbalanced mode you use bipolarity signal levels (±5V supplies) with controlled slew rates, analogous to RS-232. In the balanced mode you use unipolarity TTL-like levels (and thus single +5V supplies at both ends), with no slew-rate limiting. Figure 9.37 shows the trade-off of practical data rate versus line length.

A popular series of RS-422/3 driver/receivers is the 26LS30–34 from AMD, widely second-sourced by other manufacturers; the recent 75ALS192/4 and DS34F30/80 series claim improved performance and less power. We used RS-422 with twisted-pair flat cable in an application where we wanted to tie together the parallel ports and control signals of a set

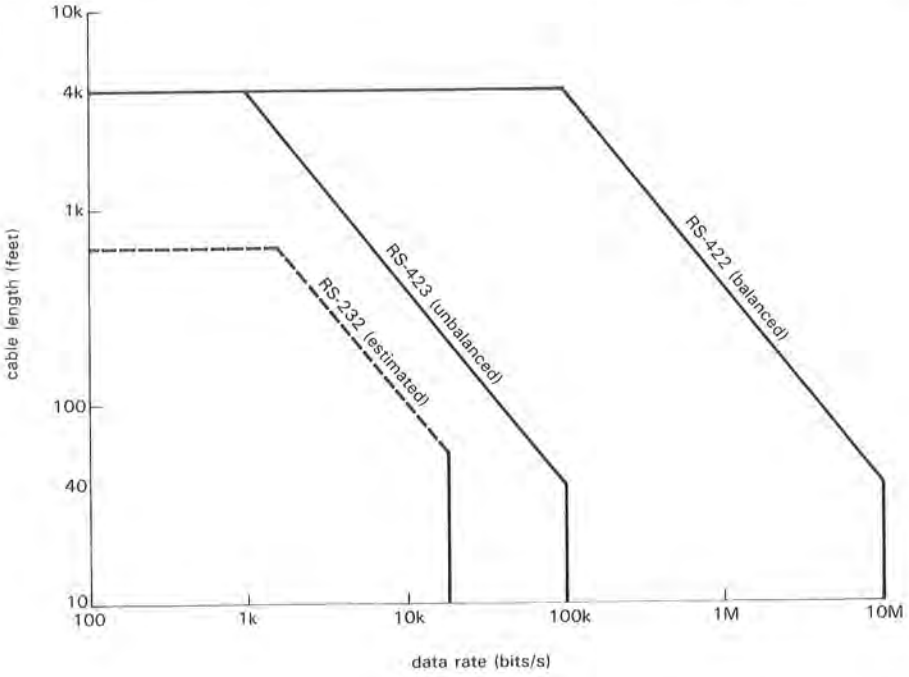


Figure 9.37. Serial communication data rate versus length.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

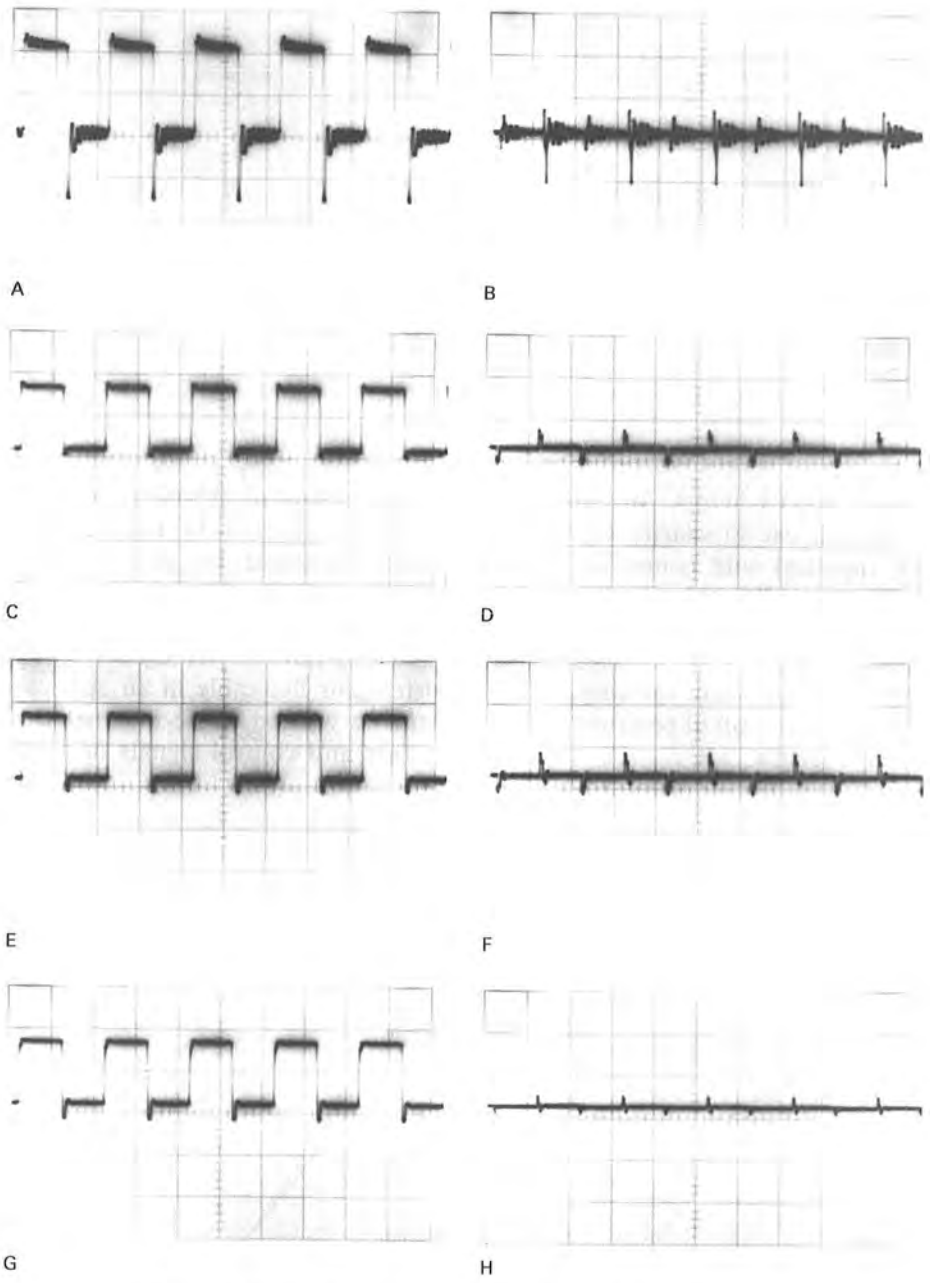


Figure 9.38. Degradation and cross-talk of digital signals.
A. 1MHz TTL square wave, through 10 feet of flat ribbon cable, alternating grounds, unterminated; 1V/div.
B. Pair next to A, but driven by TTL LOW.
C,D. Same as A,B, but terminated with 220Ω/330Ω network to +5 volts.
E,F. Same as C,D, but twisted-pair instead of flat ribbon cable.
G,H. Same as C,D, but flat ribbon with ground plane.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

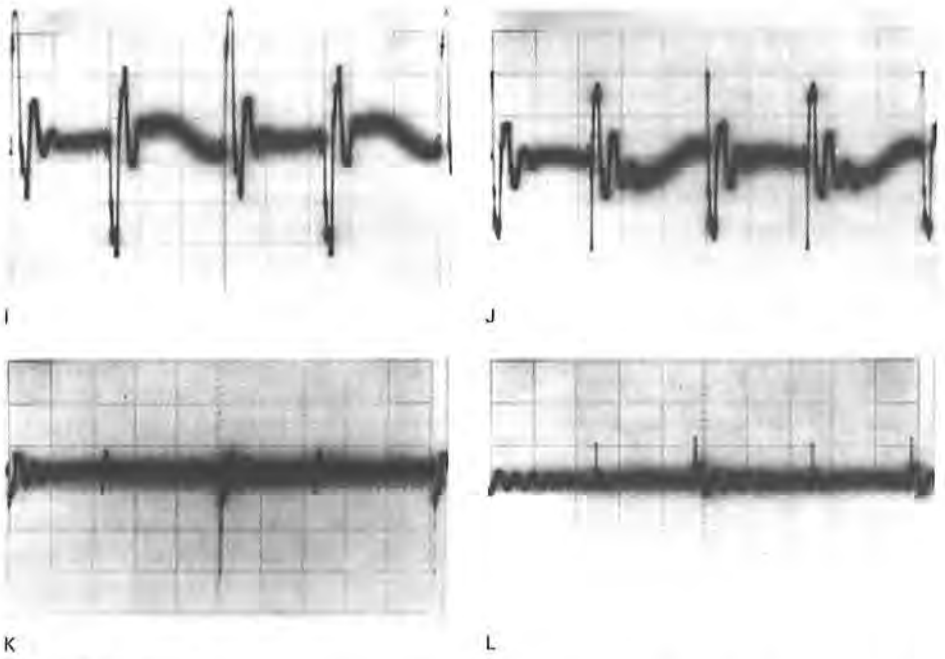


Figure 9.38. (*cont.*)
I. LOW RS-422 level pair, through 100 feet of flat ribbon cable adjacent to pair driven by 100kHz RS-422 differential square wave; 0.1V/div (note scale change).
J. Same as I, but signals separated by a grounded pair.
K,L. Same as I,J, but "twist-and-flat" pair instead of flat ribbon cable.

of 144 microprocessor boards in a "star" network. We made 9 groups of 16 processor boards, each group also including one interface board, using TTL signals between the processors and interface; the 9 interface boards were then strung together with an external computer using (differential) RS-422. The total cable length was about 25 feet, terminated by bridging 100 ohm resistors across each pair at both ends. The whole system is extremely clean and reliable at our data rates of about 1Mb/s.

We favor differential signal transmission where reliability and good noise immunity are important. The differential signal results in very small signal coupling ("crosstalk"), because of cancellation effects. The use of twisted-pair, rather than flat cable, makes things even better. Figure 9.38 shows some measured waveforms, for both RS-422 and direct logic drive, using both flat cable and flat twisted-pair (the latter was actually a variant known as "twist-and-flat," consisting of twisted-pair cable interrupted for 2 inches every 20 inches to make a flat termination area). For RS-422 we used 100 feet of cable, driving the 6 volt pp 100kHz signal on one pair and looking at cross-talk on the adjacent pair; both were terminated. For the direct logic drive we used 74LS244 drivers at 1MHz through 10 feet of cable, with or without termination as indicated. From the waveforms it should be clear that RS-422 is very safe, even with long cable runs, whereas direct logic drive is marginal even with modest cable runs, though improved by termination and the use of flat cable with integral ground plane. Unexpectedly, twisted-pair did not fare any better than flat cable for direct logic drive.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

Differential line receivers work well, as long as the received signal stays within the allowable common-mode voltage range, usually a few volts (±3V for the 75108). With long lines, however, you can get into a situation where either (a) high-frequency common-mode noise or (b) low-frequency (power-line) ground-voltage differences exceed the receiver's common-mode range. If the problem isn't too severe, you can use a pair of resistive dividers at the receiver's input, or use a receiver with built-in attenuator, for example the 26LS33 RS-422 receiver, with ±15 volt common-mode range.

For really long cable runs, or very noisy environments, the usual solution is to use transformer coupling. With transformers, of course, you can't send dc logic levels: You are forced to encode the data in some fashion, for example with a "carrier" signal. Local-area networks (Section 10.21) usually use transformer coupling.

□ *AMD "TAXI chip®."* AMD has introduced an interesting differential transmitter/receiver pair, the Am7968/9, with all sorts of on-chip registers to make it easy to use (Fig. 9.39). For example, you can treat the transmitter chip like an 8-bit latch with strobe and handshake; in that mode the chip converts the bytes to serial data, appends appropriate synchronization bits, transmits the data over a serial link, and reconstructs the byte at the other end. To the user the link looks like a simple parallel register. These chips include the cable drivers and receivers for 50 ohm cable, they run on a single +5 volt supply, and they go *fast*: Data throughput rates are 32–100 megabits/s (4 to 12.5 megabytes/s). The TAXI chips are intended as a high-speed general-purpose data link, with dc or ac coupling. The actual transmission medium can be direct connection via wire, twisted-pair, coax, transformer-coupled cable, or even a fiber-optic link.

Coaxial-cable drivers

Coaxial cable gives very good protection against interference because of its completely shielded geometry. In addition, its uniformity of diameter and spacing (compared with a certain randomness in

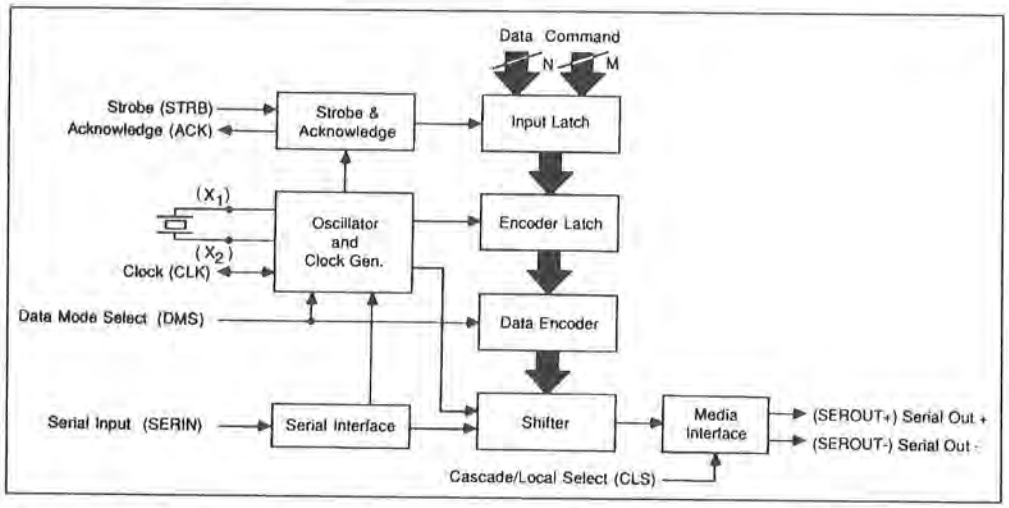


Figure 9.39. AMD "TAXI chip®" set: transparent high-speed serial link. (Diagram courtesy of Advanced Micro Devices of Sunnyvale, California.)
A. Am7968 transmitter.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

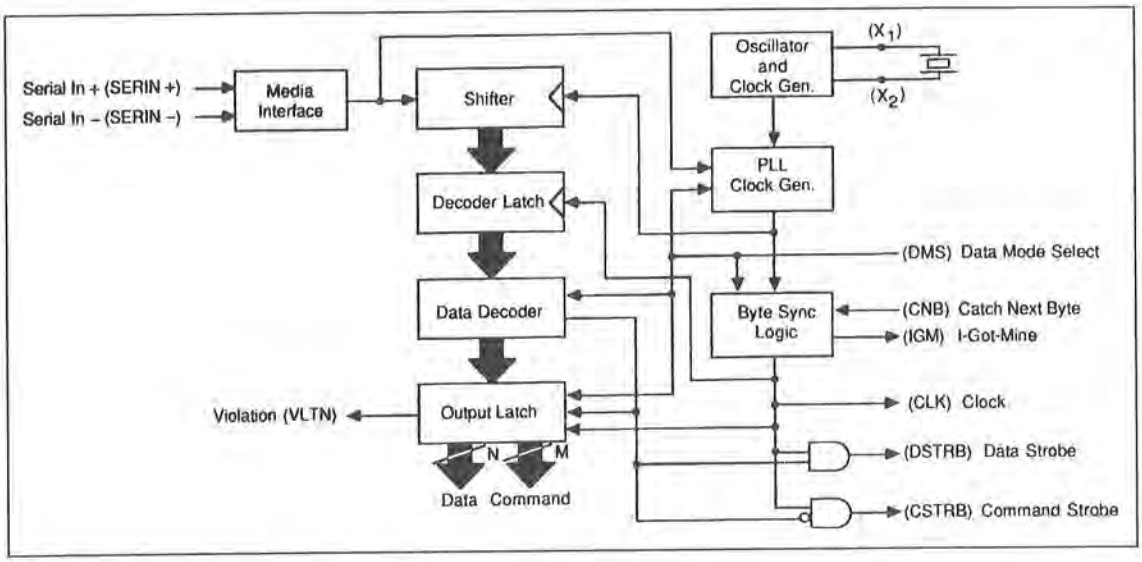


Figure 9.39. B. Am7969 receiver.

the case of bundled cables or twisted-pair) leads to predictable characteristic impedance and therefore excellent transmission properties; that is why it is used exclusively for analog radiofrequency signals.

There are several driver/receiver pairs that are suited for digital transmission with coax; Figure 9.40 shows an example. The cable is terminated in its characteristic impedance, in this case 51 ohms. The 8T23 can drive a 50 ohm load directly, and the 8T24 has a fixed amount of hysteresis for noise immunity and fast output transition time. This circuit allows bit rates of 100kb/s over a mile of cable and up to 20Mb/s over shorter lines. Other driver/receiver pairs are available in the 8Txx and 75xxx interface families. The 74F3037 (quad) and 74F30244 (octal) drivers are meant to drive cables with impedances down to 30 ohms (e.g., a cable terminated at both ends). Be sure to use the specified receivers when driving 50 ohm coaxial lines, since the voltage levels on a terminated cable may be less than ordinary logic levels.

The various ECL logic families provide

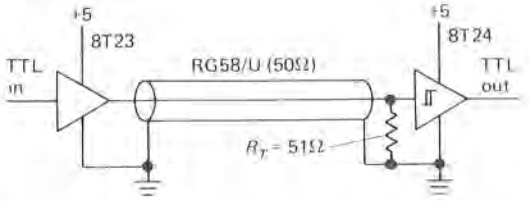


Figure 9.40. 50 ohm cable driver and receiver.

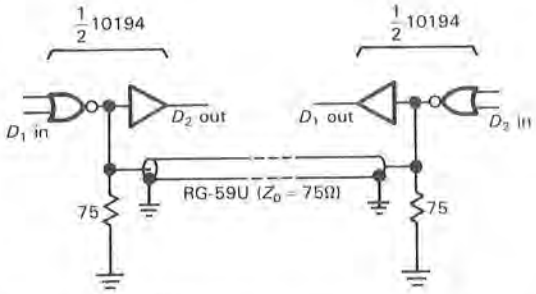


Figure 9.41. ECL current-sourcing transceiver (full duplex).

a number of 50 ohm coax driver/receiver pairs, for example the 10128/10129 pair. A very nice coax driver chip is the 10194: It's actually a bus transceiver, designed to

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

transmit and receive *simultaneously* ("full duplex") on the same line (Fig. 9.41). When used this way, each chip can be transmitting to the other transceiver while receiving asynchronously from it at the same time, without cross-talk, at speeds of 100MHz or more.

You can drive a length of coax directly from +5 volt logic if you use a single *npn* emitter follower (Fig. 9.42). The 2N4401 is a husky little transistor with plenty of beta at high currents ($h_{FE} > 100$ at $I_C =$ 150mA). The 10 ohm resistor is for short-circuit protection. In comparison with the carefully designed and expensive 50 ohm cable driver chips that are available, this circuit is embarrassingly simple. Note that the "open-emitter" output must be loaded with a low-resistance return to ground in order to work, which is also true of some IC cable driver chips.

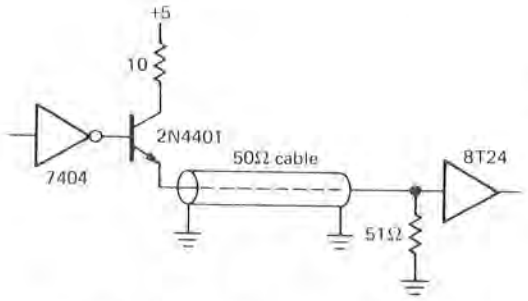


Figure 9.42. Simple 50 ohm cable driver.

Fiber-optics

An exciting new method of signal transmission is provided by fiber-optic cables. These come as nice plastic-coated cables, with matching connectors, emitters, and detectors. High-performance fiber-optic cables can transmit bandwidths of several gigahertz over distances of tens or hundreds of kilometers, with losses of fractional decibels per kilometer. Compared with coaxial cables, which are dispersive (different frequencies travel at different speeds, and with different amounts of loss, thereby corrupting waveforms), fiber-optic cables can be designed so that dispersion is negligible. And fiber-optic cables are insulators, so they can be used to send signals between devices with separate grounds, or at different voltages. Unlike ordinary cables, they do not act as antennas for radio interference or impulsive noise. They are lighter, safer, and more rugged than conventional cables, and they are potentially cheaper, too.

There are several types of fiber-optic cables, the choice involving a trade-off between cost and performance (length times bandwidth). The cheapest is the so-called step-index, multimode fiber, typically a plastic fiber 1mm in diameter; you would normally drive it with an infrared LED (as opposed to a laser diode) and use a phototransistor or pin-diode detector. Motorola makes an inexpensive set of driver/receivers (less than $1 each) that mate directly with jacketed fiber-optic cable (the MFOE71/MFOD71-73 series); with them you can send data at rates up to 1Mb/s over 30 feet of the plastic cable just described (see Fig. 9.43). Higher-performance cables use glass fibers – step-index multimode, graded-index multimode (better), or single-mode (best). With 200μm glass step-index fiber you can run 5Mb/s over a 1km path with standard off-the-shelf fiber-optic components, which include connectors, couplers, splitter/combiners, and detectors with built-in amplifiers. The current record for high-bandwidth long-haul fiber-optic communication is 4GHz over a 120km path, using no repeaters.

ANALOG/DIGITAL CONVERSION

9.15 Introduction to A/D conversion

In addition to applying the purely "digital" interfacing (switches, lights, etc.) discussed in the last few sections, it is

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

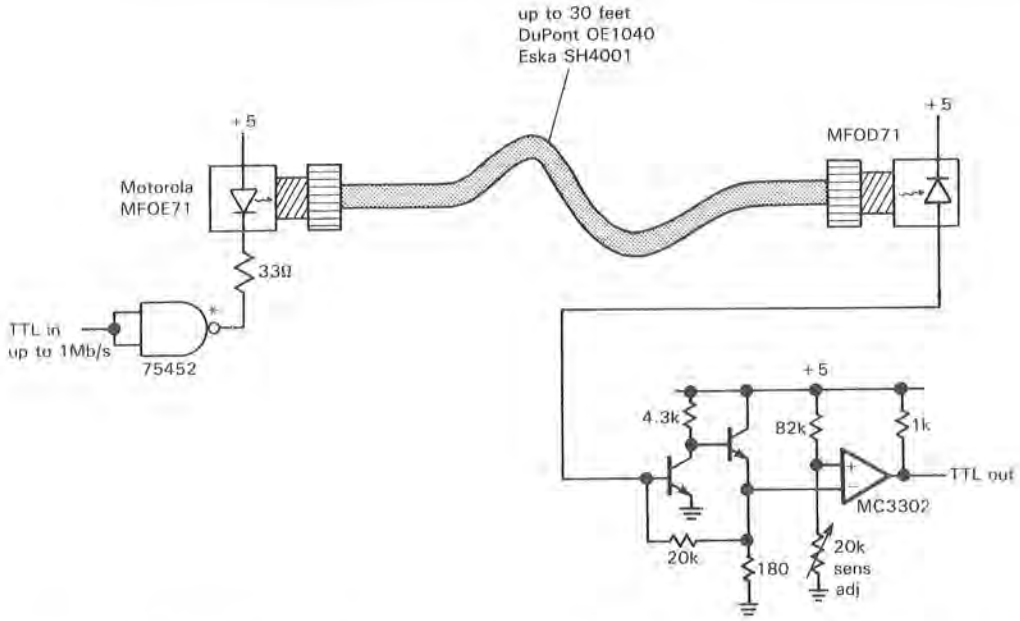


Figure 9.43. Inexpensive fiber-optic link. (Adapted from Fig. 7, MFOD71 Data Sheet, Motorola, Inc.)

often necessary to convert an analog signal to an accurate digital number proportional to its amplitude, and vice versa. This is essential in any application in which a computer or processor is logging or controlling an experiment or process, or whenever digital techniques are used to do a "normally" analog job. Applications in which analog information is converted to an intermediate digital form for error- and noise-free transmission [e.g., "digital audio" or pulse-code modulation (PCM)] make heavy use of analog/digital conversion. It is necessary in a wide variety of measurement instruments (including ordinary bench instruments such as digital multimeters and more exotic instruments such as transient averagers, "glitch catchers," and digital memory oscilloscopes), as well as signal-generation and processing instruments such as digital waveform synthesizers and data encryption devices.

Finally, conversion techniques are essential ingredients for the generation of analog displays by a digital instrument, e.g., a meter indication or xy display (or plot) created by a computer. Even in a relatively unsophisticated electronic apparatus there are plenty of nice applications that call for A/D and D/A conversion, and it is worth developing familiarity with the various techniques and available modules used in analog/digital conversion, especially since the day of the $5 A/D and D/A converter is here.

Our treatment of the various conversion techniques will not be aimed at developing skill in converter design itself. Rather, we will try to point out the advantages and disadvantages of each method, since in most cases the sensible thing is to buy commercially available chips or modules, rather than build the converter from scratch. An understanding of conversion techniques and idiosyncrasies will guide

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

you in choosing among the hundreds of available units.

Codes

At this point you should review Section 8.03 on the various number codes used to represent signed numbers. Offset binary and 2's complement are commonly used in A/D conversion schemes, with sign-magnitude and Gray codes also popping up from time to time. Here is a reminder:

	Offset binary	*2's Complement*
+Full scale	11111111	01111111
+Full scale−1	11111110	01111110
↓	↓	↓
0+1 LSB	10000001	00000001
0	10000000	00000000
0−1LSB	01111111	11111111
↓	↓	↓
−Full scale+1	00000001	10000001
−Full scale	00000000	10000000

Converter errors

The subject of A/D and D/A errors is a complicated one, about which whole volumes could be written. According to Bernie Gordon at Analogic, if you think a high-accuracy converter system lives up to its claimed specifications, you probably haven't looked closely enough. We won't go into the application scenarios necessary to support Bernie's claim, but we will show the four most common types of converter errors. Rather than bore you with a lot of complicated talk, we'll just present self-explanatory graphs of the four most common errors: offset error, scale error, nonlinearity, and nonmonotonicity (Fig. 9.44).

9.16 Digital-to-analog converters (DACs)

The goal is to convert a quantity specified as a binary (or multidigit BCD) number to a voltage or current proportional to the value of the digital input. There are several popular methods:

Scaled resistors into summing junction

As you saw in Section 4.09, by connecting a set of resistors to an op-amp summing junction you get an output proportional to a weighted sum of the input voltages. (Fig. 9.45). This circuit generates an output from zero to −10 volts, with full output corresponding to an input count of 64. Actually, the maximum input count is always 2^n-1, i.e., all bits set to 1. In this case the maximum input count is 63, with output voltage of $-10 \times 63/64$. By changing the feedback resistor, you can generate an output of zero to −6.3 volts (i.e., output in volts numerically equal to −1/10 of input count), or you can add an inverting amplifier, or dc offset at the summing junction, to get positive outputs. By changing the input resistor values you can properly convert a multidigit BCD input code, or any other weighted code. The input voltages must be clamped to an accurate reference, and the smaller input resistors must be of correspondingly higher precision. Of course, the switch resistance must be smaller than $1/2^n$ of the smallest resistor, an important consideration since in actual circuits the switching is done with transistor or FET switches. This conversion technique is used only in fast, low-precision converters.

EXERCISE 9.2

Design a 2-digit BCD D/A converter. Assume that the inputs are zero or +1 volt; the output should go from zero to 9.9 volts.

R-2R ladder

The scaled resistor technique becomes awkward when you have more than a few bits. For example, in a 12-bit converter you would need a range of resistor values of 2000:1, with corresponding precision in the small resistor values. An elegant solution is provided by the *R-2R* ladder,

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

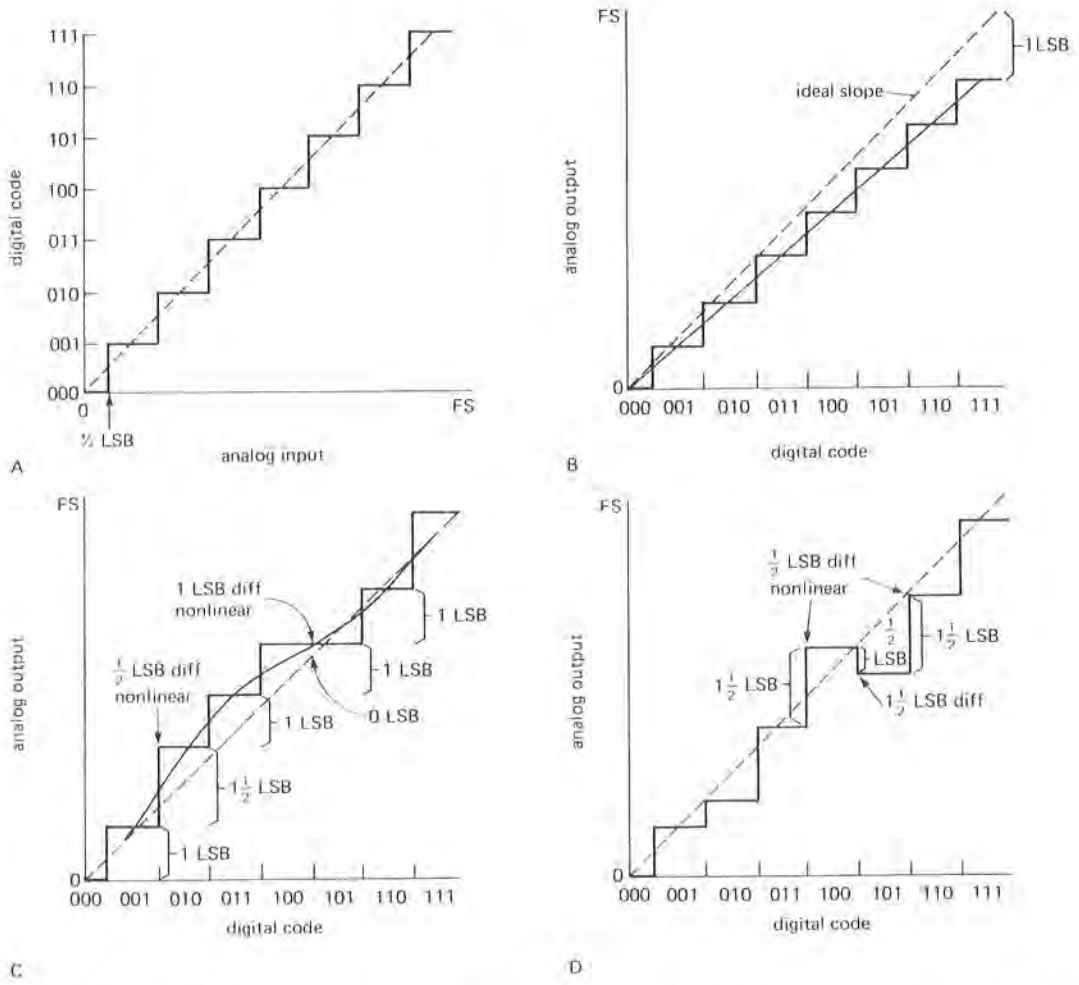


Figure 9.44. Graphs illustrating the definitions of four common digital conversion errors. (Courtesy of National Semiconductor Corp.).
A. ADC transfer curve, $\frac{1}{2}$ LBS offset at zero.
B. Linear, 1LSB scale error.
C. $\pm\frac{1}{2}$ LSB nonlinearity (implies 1 LSB possible error); 1 LSB differential nonlinearity (implies monotonicity).
D. Nonmonotonic (must be $> \pm\frac{1}{2}$ LSB nonlinear).

shown in Figure 9.46. Only two resistor values are needed, from which the *R-2R* network generates binary scaled currents. Of course, the resistors must be precisely matched, though the actual resistor values are not critical. The circuit shown generates an output of zero to -10 volts, with "full output" corresponding to an input count of 16 (again, the maximum input count is 15, with output voltage of $10 \times 15/16$). With a few modifications, an *R-2R* scheme can be used for BCD conversion.

EXERCISE 9.3
Prove that the foregoing *R-2R* ladder above works as advertised.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

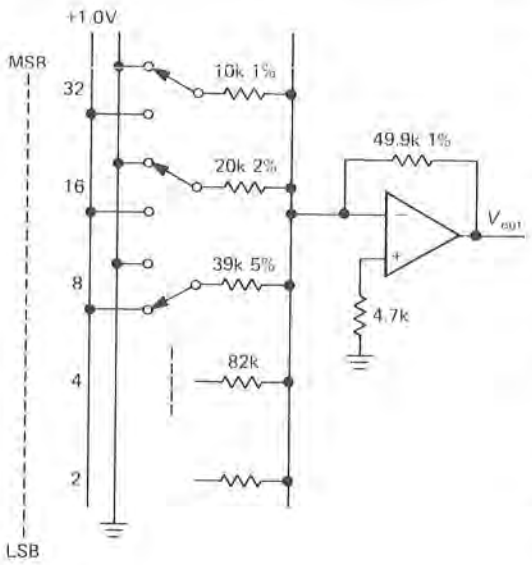


Figure 9.45

Scaled current sources

In the *R-2R* converter in the preceding paragraph, the op-amp converts binary scaled *currents* to an output *voltage*. Although an output voltage is often most convenient, the op-amp tends to be the slowest part of the converter circuit. In situations where you can use a converter with current output, you will get better performance, usually at lower price. Figure 9.47 shows the general idea. The currents can be generated by an array of transistor current sources with scaled emitter resistors, although IC designers usually use instead an *R-2R* ladder of emitter resistors. In most converters of this type, the current sources are ON all the time, and their output current is switched to the output terminal or to ground, under control of the digital input code. Watch out for limited output compliance in current-output DACs; it can be as little as 0.5 volt, though values of a few volts are typical.

Generating a voltage output

There are a few ways to generate an output voltage from a current DAC. Figure 9.48 shows some ideas. If the load capacitance is low, and large voltage swings aren't needed, a simple resistor to ground will do nicely. With the usual 1mA full-scale output current, a 100 ohm load resistor will give 100mV full-scale output with 100 ohms output impedance. If the capacitance of the DAC's output combined with the load capacitance doesn't exceed 100pF, you will get 100ns settling time in the preceding example, assuming the DAC is that fast. When worrying about the effect of *RC* time constants on DAC output response, don't forget that it takes quite a few *RC* time constants for the output to settle to within

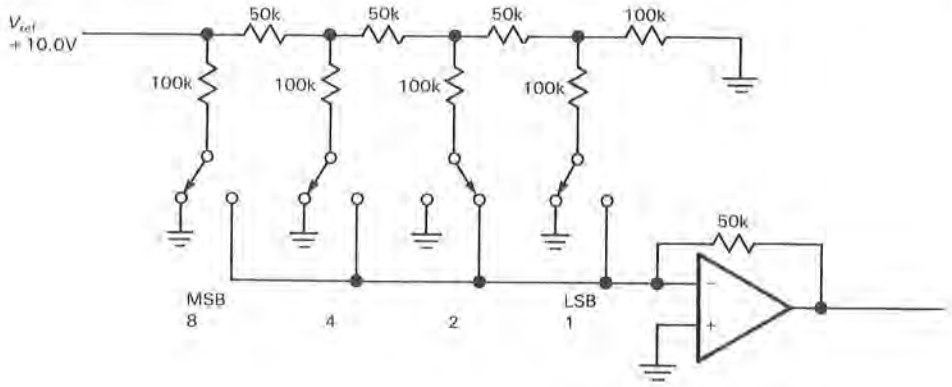


Figure 9.46. *R-2R* ladder.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

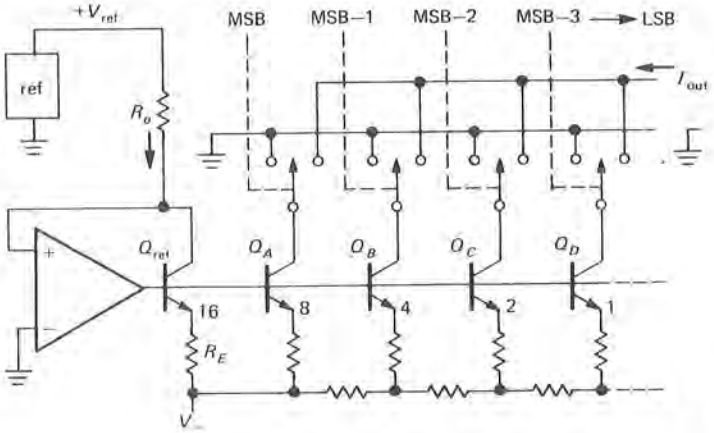


Figure 9.47. Classic current-switched DAC.

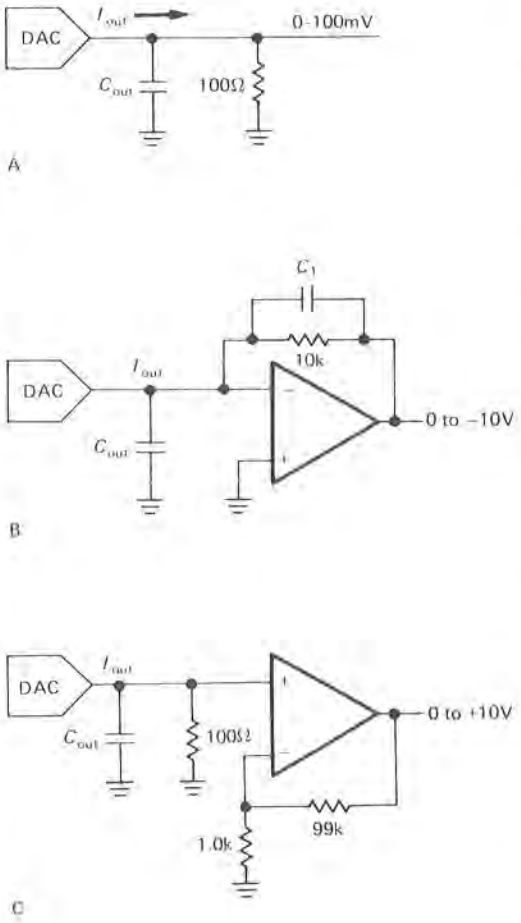


Figure 9.48. Generating voltages from current-output DACs.

½ LSB of the final voltage. It takes 7.6 RC time constants, for instance, to settle to within 1 part in 2048, which is what you would want for a 10-bit converter output.

To generate large swings, or to buffer into small load resistances or large load capacitances, an op-amp can be used in the transresistance configuration (current-to-voltage amplifier), as shown. The capacitor across the feedback resistor may be necessary for stability, because the DAC's output capacitance in combination with the feedback resistance introduces a lagging phase shift; unfortunately, that compromises the speed of the amplifier. It is an irony of this circuit that a relatively expensive high-speed (fast-settling) op-amp may be necessary to maintain the high speed of even an inexpensive DAC. In practice, the last circuit may give better performance, since no compensation capacitor is needed. Watch out for offset voltage error, since the op-amp's input offset voltage is amplified 100 times.

Commercial D/A converter modules are available with precision ranging from 6 bits up to 18 bits, with settling times of 25ns to 100μs (for converters of the highest precision). Prices range from a few dollars to several hundred dollars. A typical popular unit is the AD7248, a

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

12-bit converter with latch and internal reference, settling time of 5μs for voltage output, and a price of about $10.

□ 9.17 Time-domain (averaging) DACs

□ *Frequency-to-voltage converters*

In conversion applications the "digital" input may be a train of pulses or other waveform of some frequency; in that case, direct conversion to a voltage is sometimes more convenient than the alternative of counting for predetermined time, then converting the binary count via the preceding methods. In direct F/V conversion, a standard pulse is generated for each input cycle; it may be a voltage pulse or a pulse of current (i.e., a fixed amount of charge).

An RC low-pass filter or integrator then averages the pulse train, giving an output voltage proportional to the average input frequency. Of course, some output ripple results, and the low-pass filter necessary to keep this ripple less than the D/A precision (e.g., $\frac{1}{2}$ LSB) will, in general, cause a slow output response. To ensure less than $\frac{1}{2}$ LSB output ripple, the time constant T of a simple RC low-pass filter must be at least $T = 0.69(n+1)T_o$, where T_o is the output period of the n-bit F/V converter corresponding to maximum input frequency. The output of this RC network will settle to $\frac{1}{2}$ LSB, following a full scale change at the input, in $0.69(n+1)$ filter time constants. In other words, the output settling time to $\frac{1}{2}$ LSB will be approximately $t = 0.5(n+1)^2T_o$. A 10-bit F/V converter with 100kHz maximum input frequency, smoothed with an RC filter, will have an output voltage settling time of 0.6ms. With more complicated low-pass filters (sharp cutoff) you can get improved performance. Before you get carried away with fancy filter design, however, you should remember that F/V techniques are most often used when a voltage output is not what's needed. For some perspective, we will next comment about intrinsically slow loads in connection with pulse-width modulation.

□ *Pulse-width modulation*

In this technique the digital input code is used to generate a train of pulses of fixed frequency, with width proportional to the input count; this can be easily done with a counter, magnitude comparator, and high-frequency clock (see Exercise 9.4). Once again, a simple low-pass filter can be used to generate an output voltage proportional to the average time spent in the HIGH state, i.e., proportional to the digital input code. More often, this sort of D/A conversion is used when the load is itself a slowly responding system; the pulse-width modulator then generates precise parcels of energy, averaged by the system connected as a load. For example, the load might be capacitive (as in a switching regulator, see Chapter 6), thermal (a thermostated bath with heater), mechanical (a tape speed servo), or electromagnetic (a large electromagnet controller).

EXERCISE 9.4

Design a circuit to generate a 10kHz train of pulses of width proportional to an 8-bit binary input code. Use counters and magnitude comparators (suitably expanded).

□ *Averaged rate multiplier*

The rate-multiplier circuit described in Section 8.28 can be used to make a simple D/A converter. A parallel binary or BCD input code is converted to a train of output pulses of average frequency proportional to the digital input; simple averaging, as in the preceding F/V converter, can be used to generate a dc output proportional to the digital input code, although in this case the resulting output time constant may be intolerably long, since the rate-multiplier

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

output will have to be averaged for a time equal to the longest output period it can generate. Rate multipliers are most advantageously used as D/A converters when the output is intrinsically averaged by the slowly responding characteristics of the load itself, as described earlier.

Perhaps the nicest application is in digital temperature control, where complete cycles of ac power are switched across the heater for each rate-multiplier output pulse. In this application the rate multiplier is arranged so that its lowest output frequency is an integral submultiple of 120Hz, and a solid-state relay (or triac) is used to switch the ac power (at zero crossings of its waveform) from logic signals.

Note that the last three conversion techniques involve some time averaging, whereas the resistor-ladder and current-source methods are "instantaneous," a distinction that also exists in the various methods of analog-to-digital conversion. Whether a converter averages the input signal or converts an instantaneous sample of it can make an important difference, as you will see shortly in some examples.

9.18 Multiplying DACs

Most of the preceding techniques can be used to construct a "multiplying DAC," in which the output equals the product of an input voltage (or current) and an input digital code. For instance, in a scaled current-source DAC, you can scale all the internal current sources by an input programming current. Multiplying DACs can be made from DACs that have no internal reference by using the reference input for the analog input signal. However, not all DACs are optimized for use in this way, so it is best to check the data sheets of the converters you're considering for details. A DAC with good multiplying properties (wide analog input range, high speed, etc.) will usually be called a "multiplying DAC" right at the top of the data sheet. The AD7541, 7548, 7845, and DAC1230 are all examples of 12-bit multiplying DACs, priced around $10 to $20.

Multiplying DACs (and the A/D equivalent) open the possibility of *ratiometric* measurements and conversions. If a sensor of some sort (e.g., a variable-resistance transducer like a thermistor) is powered by a reference voltage that also supplies the reference for the A/D or D/A converter, then variations in the reference voltage will not affect the measurement. This concept is very powerful, since it permits measurement and control with accuracy greater than the stability of voltage references or power supplies; conversely, it relaxes the requirements on supply stability and accuracy. The ratiometric principle is used in its simplest form in the classic *bridge* circuit, in which two ratios are adjusted to equality by nulling the differential signal taken between the two voltage-divider outputs (see Section 15.02). Devices like the 555 (see Section 5.14) achieve good stability of output frequency with large variations of supply voltage by using essentially a ratiometric scheme: The capacitor voltage, generated by an RC network from the supply, is compared with a fixed fraction of the supply voltage ($\frac{1}{3}V_{CC}$ and $\frac{2}{3}V_{CC}$), giving an output frequency that depends only on the RC time constant. We will have more to say on this important subject in connection with A/D converters later in this chapter, as well as in Chapter 15, when we discuss scientific measurement techniques.

9.19 Choosing a DAC

To guide you in choosing a DAC for a particular application, we've assembled in Table 9.4 a representative selection of DACs of various precisions and speeds. This listing is by no means exhaustive, but it does include many of the more popular converters and some more recent

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

TABLE 9.4. DIGITAL-TO-ANALOG CONVERTERS

Type	Mfg[a]	# per pkg	Bits	Latch	V or I ?	Speed to MSB/2 (ns)	Pol[b]	Input org	Pkg[c]	V_{supply} (V)	I_{supply} (mA)	Reference	Trim[d]	Multiplying?	Ext'd compl?[e]	Low glitch?	Price	Comments
AD9702	AD	3	4	1	I	5	–	12	24	±6	1.8W	ext	–	–	4.2	•	$45	color video; ECL or TTL
AD7225	AD	4	8	2	V	5000	+	4x8	24S	+15	10	ext	NT	–	–	–	$18	double buffered; 7226 is single buff.
AD558	AD	1	8	1	V	1000	+	8	16	5 to 15	15	int	NT	–	–	–	$6	complete, easy to use
DAC0830	NS	1	8	2	I	1000	M	8	20	5 to 15	2	0 to ±25	NT	•	N	–		same pinout as 12-bit DAC1230
AD7528	AD	2	8	1	I	350	M	8	20	5 to 15	0.1	0 to ±25	NT	•	N	–	$6	dual, easy to use
DAC8408	PM	4	8	1	I	190	M	4x8	28	+5	0.05	ext	NT	•	N	–		can read back buffers
Bt-110	BT	8	8	1	I	100	+	8	40	+5	30	int	•	–	N	–		octal
AD7524	AD	1	8	1	I	100	M	8	16	5 to 15	0.1	0 to ±10	NT	•	N	–	$5	indus std multiplying DAC
DAC-08	AD	1	8	-	I	85	–	8	16	±15	+2,-6	ext	–	–	28	–	$1	obsolete; indus std
Bt-453	BT	3	8	1	I[f]	15	+	8	40	+5	160	ext	–	–	N	–	$58	256x24 palette, for 256 of 16M colors
HDG0807	AD	1	8	-	V[f]	14	+	8	24	+5	185	int	NT	–	–	–	$43	video, 75Ω output
TDC1018	TR	1	8	1	I	10	–	8	24	-5.2	100	ext	–	–	4	•		video
AD9768	AD	1	8	-	I	5	–	8	18?	±5	+15,-70	int	NT	–	N	–	$40	ECL inputs
TDC1318	TR	3	8	1	I	5	–	3x8	40	-5.2	200	int	NT	–	N	–		high res'n color video; many 2nd sources
TQ6112	TQ	1	8	1	V	1	–	8	44	-3.5,-9	3.5W	ext	–	–	N	•		GaAs: ultra-fast
IDT75C29	ID	1	9	1	I	8	–	9	24	+5	80	ext	–	–	N	–		
DAC1000	NS	1	10	2	I	500	M	8+2	20	5 to 12	0.5	0 to ±25	NT	•	N	–		double buffered; cheap
AD7248	AD	1	12	2	V	5000	±	8+4	20	±15	5	int	NT	–	–	–	$10	single supply possible; +5V ref out
AD7537	AD	2	12	2	I	1500	M	8+4	24S	+15	5	ext	NT	–	N	–	$15	
AD7548	AD	1	12	2	I	1000	M	8+4	20	5 to 15	1	0 to ±25	NT	•	N	–	$10	
DAC1230	NS	1	12	2	I	1000	M	8+4	20	+15	1.2	0 to ±25	NT	•	N	–		same pinout as DAC0830
AD568	AD	1	12	-	I	35	±	12	24S	±15	+30,-8	int	•	–	N	•	$42	
AD7534	AD	1	14	2	I	1500	M	8+4	20	+15,-0.3	0.5	0 to ±25	•	•	N	–	$17	7535,6,8 have 14-bit bus
AD569	AD	1	16	2	V	6000	M	8+8	28	±12	±6	±5	NT	•	–	–	$28	
DAC71/72	all	1	16	-	I	1000	±	16	24	+5,±15	10,+10,-30	int	•	–	11	–	$45	indus std; V_{out} also: 10µs
PCM54	BB	1	16	-	I	350	±	16	28	±5 to ±15	±13	int	NT	–	N	•	$11	digital audio, cheap; V_{out} also: 3µs
DAC729	BB	1	18	-	I	300	–	18	40	+5, +15	18,+30,-40	int	•	–	6	–	$141	V_{out} also: 4µs; 18 bits = 4ppm!

[a] see footnote to Table 4.1. [b] M - multiplying. [c] all are DIP; S - skinny-DIP. [d] NT - no trim required. [e] extended compliance, in volts. [f] to drive 75Ω.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

entries that were intended as improved replacements.

When looking for a DAC for some application, some issues to keep in mind are (a) precision, (b) speed, (c) accuracy (external trimming required?), (d) input structure (latched? CMOS/TTL/ECL-compatible?), (e) reference (internal, or externally supplied?), (f) output structure (current output? compliance? voltage output? range?), (g) required supply voltages and power dissipation, (h) package (low pin count and 0.3″ wide "skinny-DIP" packaging are desirable), and (i) price.

9.20 Analog-to-digital converters

There are half a dozen basic techniques of A/D conversion, each with its peculiar advantages and limitations. Since you usually use a commercial A/D module or chip rather than build your own, we will describe the various conversion techniques somewhat briefly, mainly to serve as a guide for intelligent selection in a given application. The next section of the chapter will illustrate some typical A/D applications. In Chapter 11 we will discuss some A/D converters that use exactly the same conversion methods, but that have outputs designed for simplified interfacing to microprocessors.

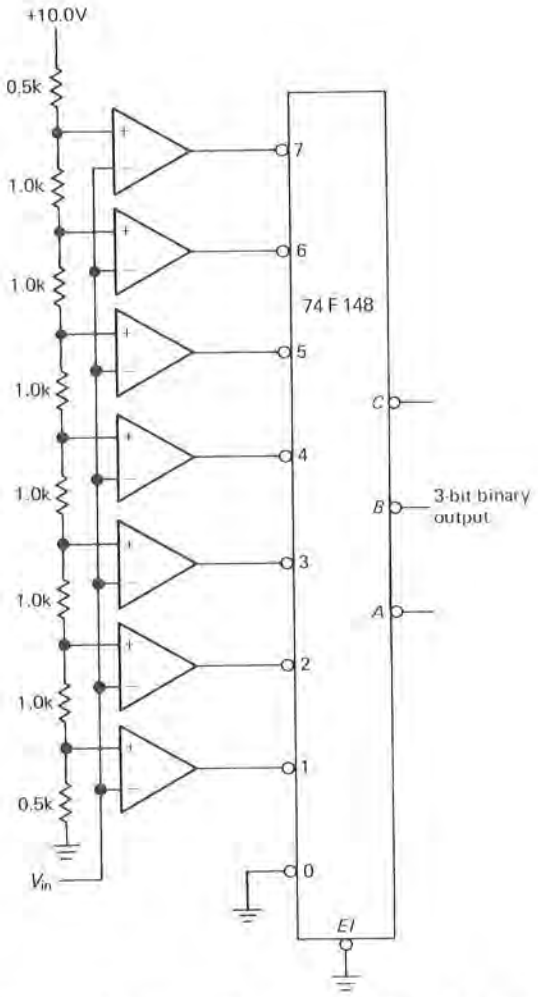


Figure 9.49. Parallel-encoded ("flash") A/D converter (ADC).

Parallel encoder

In this method the input signal voltage is fed simultaneously to one input of each of n comparators, the other inputs of which are connected to n equally spaced reference voltages. A priority encoder generates a digital output corresponding to the highest comparator activated by the input voltage (Fig. 9.49).

Parallel encoding (also called "flash" encoding) is the fastest method of A/D conversion. The delay time from input to output equals the sum of comparator plus encoder delays. Commercial parallel encoders are available with 16 to 1024 levels (4-bit to 10-bit outputs). Beyond that they become prohibitively expensive and bulky. Speeds range from 15 to 300 megasamples per second (MSPS). A typical flash ADC is the TRW TDC1048, a bipolar 8-bit 20MSPS converter in a 28-pin package, which sells for about $100; IDT makes the 75C48, a CMOS equivalent with improved specs.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

A variant on the simple parallel encoder is the so-called half-flash technique, a two-step process in which the input is flash-converted to half the final precision; an internal DAC converts this approximation back to analog, where the difference "error" between it and the input is flash-converted to obtain the least significant bits (Fig. 9.50). This technique yields low-cost converters that are faster than anything else except full flash converters. It is used in inexpensive converters like the 8-bit ADC0820 (National) and AD7820/4/8 (Analog Devices).

Flash encoders are worth considering in waveform digitizing applications even when the conversion rate is relatively slow, because their high speed (or, more precisely, their short *aperture* interval, during which the comparator outputs are latched) ensures that the input signal is effectively not changing during the conversion. The alternative – the slower converters we'll describe next – usually requires an analog sample-and-hold circuit to freeze the input waveform while conversion is going on.

Successive approximation

In this popular technique you try various output codes by feeding them into a D/A converter and comparing the result with the analog input via a comparator (Fig. 9.51). The way it's usually done is to set all bits initially to 0. Then, beginning with the most significant bit, each bit in turn is set provisionally to 1. If the D/A output does not exceed the input signal voltage, the bit is left as a 1; otherwise it is set back to 0. For an n-bit A/D, n such steps are required. What you're doing could be described as a binary search, beginning at the middle. A successive-approximation A/D module has a BEGIN CONVERSION input and a CONVERSION DONE output. The digital output is always provided in parallel format (all bits at once, on n separate output lines) and usually in serial format as well (n successive output bits, MSB first, on a single output line).

In our electronics course the students construct a successive-approximation ADC, complete with DAC, comparator, and control logic. Figure 9.52A shows the

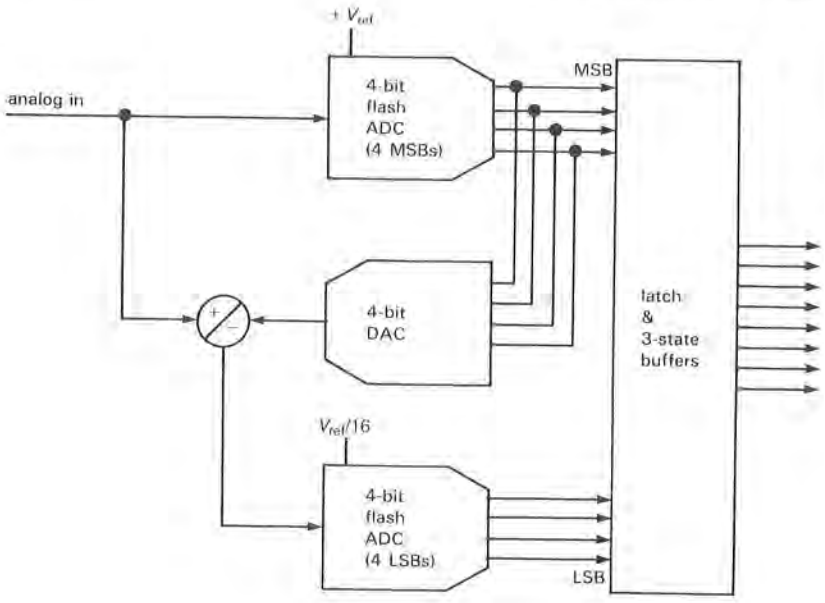


Figure 9.50. Half-flash ADC.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

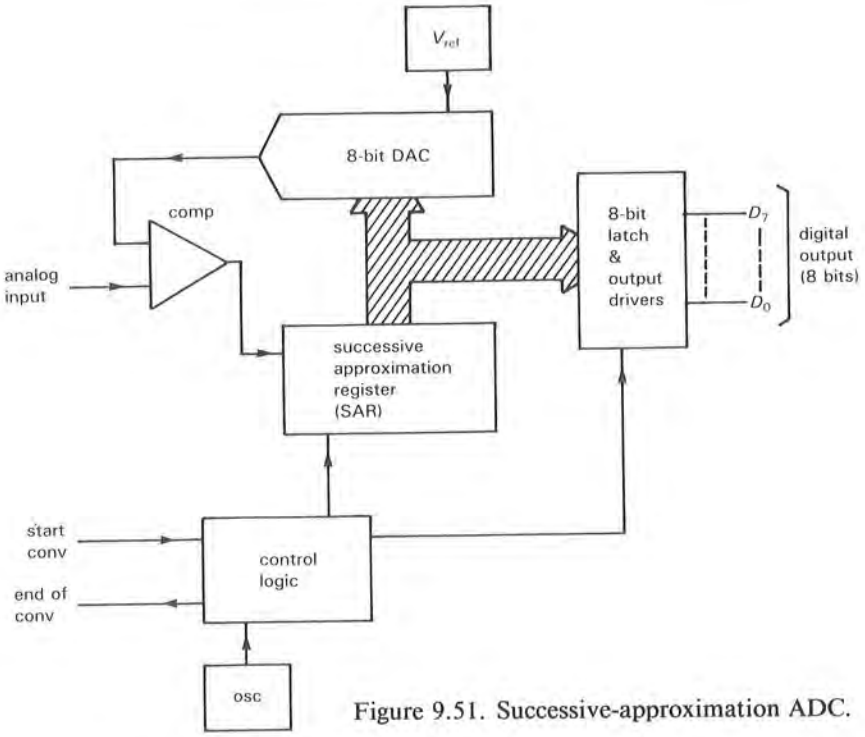
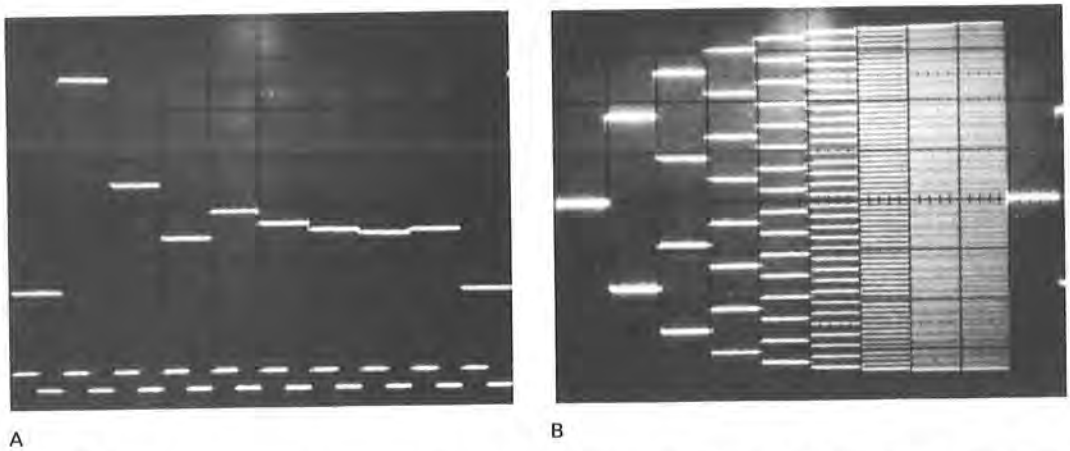


Figure 9.51. Successive-approximation ADC.

successive outputs from the DAC, along with the eight clock pulses, as the trial analog output converges to the input voltage. Figure 9.52B shows the full 8-bit "tree," a pretty picture you can generate by watching the DAC output while driving the input with a slow ramp that runs over the full analog input range.

Successive-approximation A/D converters are relatively accurate and fast,



A B

Figure 9.52. 8-bit successive-approximation waveforms. (Courtesy of P. Emery, R. Lovett, and K. Rudin.).
A. Analog output converging to final value; note clock waveform.
B. Full "tree".

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

requiring only n settling times of the DAC for n-bit precision. Typical conversion times range from $1\mu s$ to $50\mu s$, with accuracies of 8 to 12 bits commonly available; prices range from about \$10 to upwards of \$400. This type of converter operates on a brief sample of the input voltage, and if the input is changing during the conversion, the error is no greater than the change during that time; however, spikes on the input are disastrous. Although generally quite accurate, these converters can have strange nonlinearities and "missing codes."

National Semiconductor uses a nice trick in its ADC0800-series converters to eliminate missing codes: Instead of using a conventional *R-2R* ladder DAC, they use a chain of 2^n resistors and analog switches to generate the trial analog voltages (Fig. 9.53), in the manner of a flash encoder.

A variation known as a "tracking A/D converter" uses an up/down counter to generate successive trial codes; it is slow in responding to jumps in the input signal, but it follows smooth changes somewhat more rapidly than a successive-approximation converter. For large changes its slew rate is proportional to its internal clocking rate.

□ *Voltage-to-frequency conversion*

In this method an analog input voltage is converted to an output pulse train whose

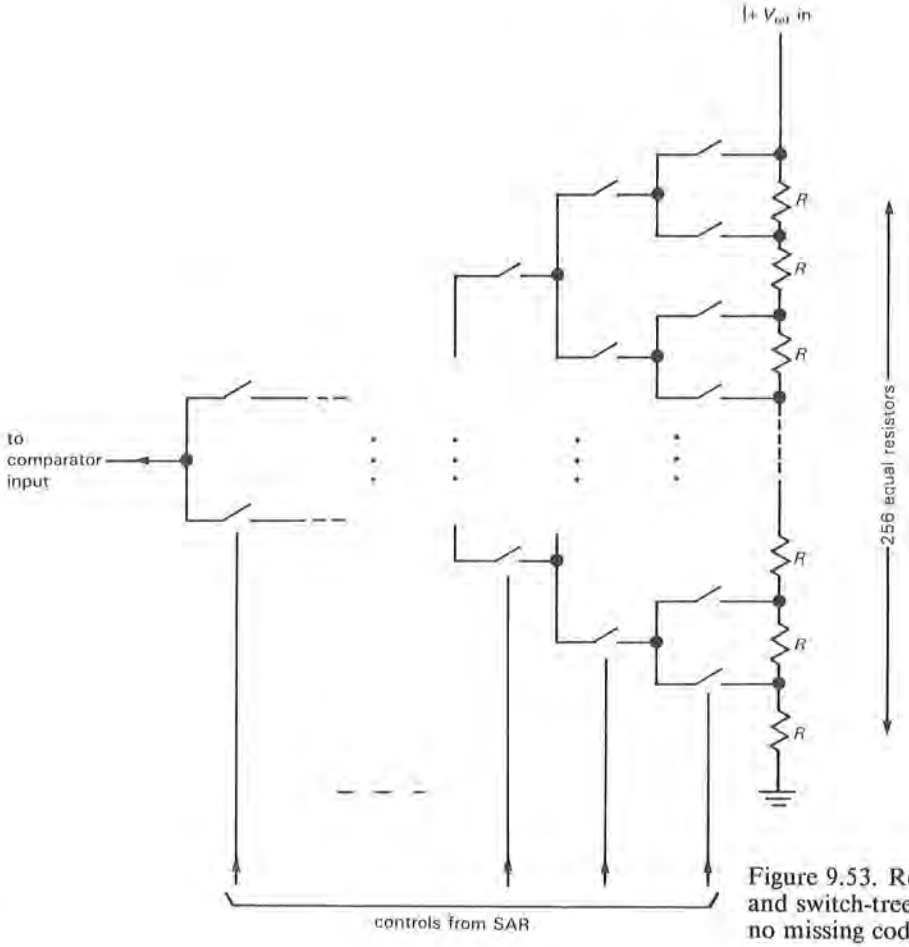


Figure 9.53. Resistor-ladder and switch-tree DAC: no missing codes.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

frequency is proportional to the input level. This can be done simply by charging a capacitor with a current proportional to the input level and discharging it when the ramp reaches a preset threshold. For greater accuracy, a feedback method is generally used. In one technique you compare the output of an F/V circuit with the analog input level and generate pulses at a rate sufficient to bring the comparator inputs to the same level. In the more popular methods, a "charge-balancing" technique is used, as will be described in greater detail later (in particular, the "capacitor-stored charge-dispensing" method).

Typical V/F output frequencies are in the range 10kHz to 1MHz for maximum input voltage. Commercial V/F converters are available with the equivalent of 12-bit resolution (0.01% accuracy). For example, the excellent AD650 from Analog Devices (Section 5.15) has a typical nonlinearity of 0.002% when operating from 0 to 10kHz. They are inexpensive, and they are handy when the output is to be transmitted digitally over cables or when an output frequency (rather than digital code) is desired. If speed isn't important, you can get a digital count proportional to the average input level by counting the output frequency for a fixed time interval. This technique is popular in moderate-accuracy (3-digit) digital panel meters.

□ *Single-slope integration*

In this technique an internal ramp generator (current source + capacitor) is started to begin conversion, and at the same time a counter is enabled to count pulses from a stable clock. When the ramp voltage equals the input level, a comparator stops the counter; the count is proportional to the input level, i.e., it's the digital output. Figure 9.54 shows the idea.

At the end of the conversion the circuit discharges the capacitor and resets the counter, and the converter is ready for another cycle. Single-slope integration is simple, but it is not used where high accuracy is required because it puts severe requirements on the stability and accuracy of the capacitor and comparator. The method of "dual-slope integration" eliminates that problem (and several others as well) and is now generally used where precision is required.

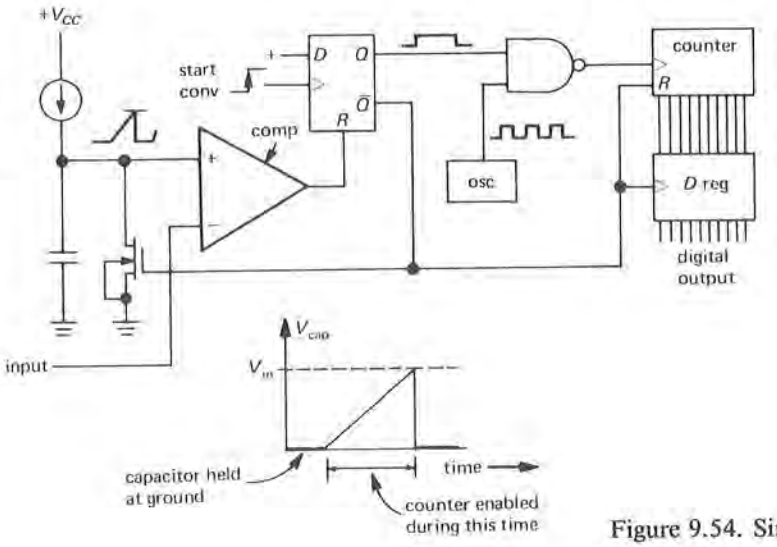


Figure 9.54. Single-slope ADC.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

Single-slope integration is still alive and well, particularly in applications that don't require absolute accuracy, but rather need conversion with good resolution and uniform spacing of adjacent levels. A good example is pulse-height analysis (see Section 15.16), where the amplitude of a pulse is held (peak detector) and converted to an address. Channel width equality is essential for this application, for which a successive-approximation converter would be totally unsuitable. The technique of single-slope integration is also used in time-to-amplitude conversion (TAC).

9.21 Charge-balancing techniques

There are several techniques that have in common the use of a capacitor to keep track of the ratio of an input signal level to a reference. These methods all average (integrate) the input signal for a fixed time interval for a single measurement. There are two important advantages:

1. Because these methods use the same capacitor for the signal and reference, they are relatively forgiving of capacitor stability and accuracy. These methods also make fewer demands on the comparator. The result is better accuracy for equivalent-quality components, or equivalent accuracy at reduced cost.

2. The output is proportional to the *average* input voltage over the (fixed) integration time. By choosing that time interval to be a multiple of the power-line period, the converter becomes insensitive to 60Hz "hum" (and its harmonics) on the input signal. As a result, the sensitivity to interfering signals as a function of frequency is as shown in Figure 9.55 (0.1s integration).

This nulling of 60Hz interference requires accurate control of the integration time, since an error of a fraction of a percent in the clock timing will result in incomplete cancellation of hum. One possibility is to use a crystal oscillator. You will see in Section 9.29 an elegant method of synchronizing the workings of an integrating converter to a multiple of the line frequency in order to make this rejection perfect.

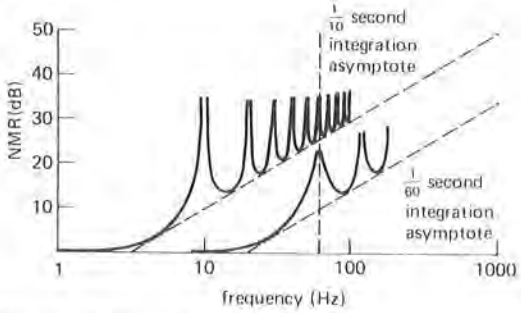


Figure 9.55. Normal-mode rejection with integrating A/D converters.

Integrating techniques have the disadvantage of slow speed, as compared with successive approximation.

Dual-slope integration

This elegant and very popular technique eliminates most of the capacitor and comparator problems inherent in single-slope integration. Figure 9.56 shows the idea. First, a current accurately proportional to the input level charges a capacitor for a fixed time interval; then the capacitor is discharged by a constant current until the voltage reaches zero again. The time to discharge the capacitor is proportional to the input level and is used to enable a counter driven from a clock running at a fixed frequency. The final count is proportional to the input level, i.e., it's the digital output.

Dual-slope integration achieves very good accuracy without putting extreme requirements on component stability. In particular, the capacitor value doesn't have to be particularly stable, since the charge cycle and the discharge cycle both go at a rate inversely proportional to C. Likewise, drifts or scale errors in the comparator are canceled out by beginning and ending each conversion cycle at the same voltage

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

and, in some cases, at the same slope. In the most accurate converters, the conversion cycle is preceded by an "auto-zeroing" cycle in which the input is held at zero. Since the same integrator and comparator are used during this phase, subtracting the resulting "zero-error" output from the subsequent measurement results in effective cancellation of errors associated with measurements near zero; however, it does not correct for errors in overall scale.

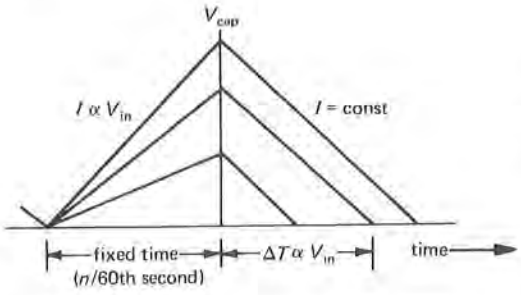


Figure 9.56. Dual-slope conversion cycle.

Note that even the clock frequency does not have to have high stability in dual-slope conversion, because the fixed integration time during the first phase of the measurement is generated by subdivision from the same clock used to increment the counter. If the clock slows down by 10%, the initial ramp will go 10% higher than normal, requiring 10% longer ramp-down time. Since that's measured in clock ticks that are 10% slower than normal, the final count will be the same! Only the discharge current has to be of high stability in a dual-slope converter with internal auto-zeroing. Precision voltage and current references are relatively easy to produce, and the (adjustable) reference current sets the scale factor in this type of converter.

When choosing components for dual-slope conversion, be sure to use a high-quality capacitor with minimum dielectric absorption ("memory" effect; see the model in Fig. 4.42) – polypropylene, polystyrene, or Teflon capacitors work best. Although these capacitors are not polarized, you should connect the outside foil (indicated with a band) to the low-impedance point (the output of the integrator op-amp). To minimize errors, choose integrator R and C values to use nearly the full analog range of the integrator. A high clock frequency improves resolution, although you gain little once the clock period becomes shorter than the comparator response time.

When using precision dual-slope converters (and, indeed, any kind of precision converter) it is essential to keep digital noise out of the analog signal path. Converters usually provide separate "analog ground" and "digital ground" pins for this purpose. It is often wise to buffer the digital outputs (say with a '244 three-state octal driver, asserted only when reading the output) in order to decouple the converter from the digital roar of a microprocessor bus (see next chapter). In extreme cases you might use opto-couplers to quarantine the noise of a particularly dirty bus. Be sure to use liberal power-supply bypassing right at the converter chip. And be careful not to introduce noise during the critical endpoint of the integration, as the ramp reaches the comparator trip point: For example, some converters let you check for end-of-conversion by reading the output word; don't do it! Instead, use the separate BUSY line, suitably isolated.

Dual-slope integration is used extensively in precision digital multimeters, as well as in conversion modules of 10-bit to 18-bit resolution. It offers good accuracy and high stability at lowest cost, combined with excellent rejection of power-line (and other) interference, for applications where speed is not important. For a fixed amount of money, you will get greatest precision with a module that uses this technique. The digital output codes are strictly monotonic with increasing input.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

□ *Delta-sigma converters*

There are several methods of A/D conversion that involve cancellation of the (average) signal input current with a switched internal source of current or charge. Figure 9.57 shows a functional diagram of a "delta-sigma" converter.

The input voltage drives an integrator, whose output is compared with any fixed voltage, such as ground. Depending on the comparator output, pulses of current of fixed length (i.e., fixed increments of charge) are switched into the summing junction or to ground at each clock transition, with the effect of maintaining zero average current into the summing junction. This is the balancing concept. A counter keeps track of the number of charge pulses switched into the summing junction for a given number of clock pulses, say 4096. That count is proportional to the average input level during the 4096 clock pulses, i.e., it's the output.

Delta-sigma converters can also be constructed with the current pulses generated with a resistor from a stable reference voltage, since the summing junction is a virtual ground. In that case you have to make sure that the switch ON resistance is small compared with the series resistor, so that variations of R_{ON} don't cause drifts.

□ *Switched-capacitor A/D*

A closely related charge-balancing method uses the "capacitor-stored charge-dispensing A/D," or "switched-capacitor" A/D. In this technique, fixed amounts of charge are created by repeatedly charging a capacitor from a stable reference voltage, then discharging it into the summing junction. The comparator looks at the integrator output, as previously, and controls the rate at which the capacitor is switched. That rate is counted for a fixed time interval to generate the digital output. This method has advantages for circuits that are meant to operate from a single supply voltage, since the effective polarity of charge transferred from the capacitor to the summing junction can be reversed by suitably connected FET switches (i.e., by switching both sides of the capacitor).

An example of this technique is the LM331 voltage/frequency converter, which

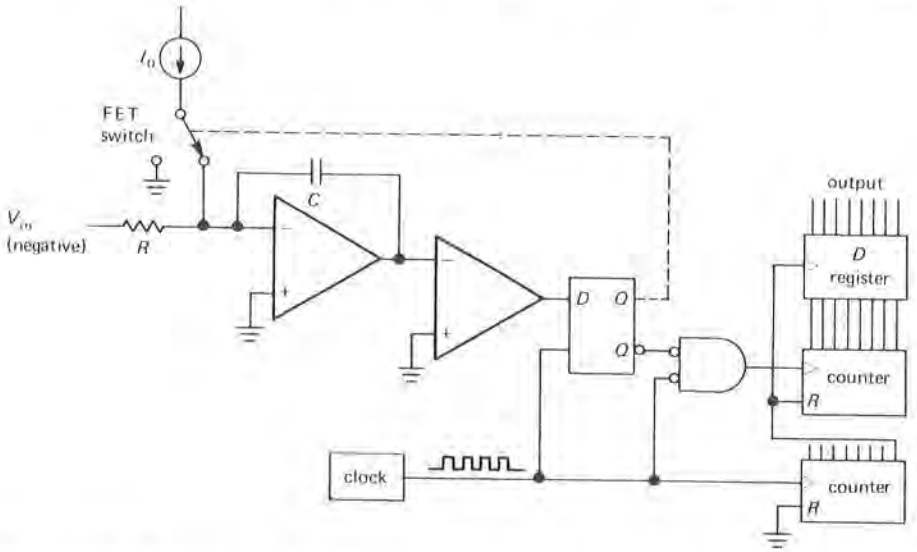


Figure 9.57. Delta-sigma charge-balancing ADC.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

has the unique advantage of operation from a single +5 volt supply. We showed its application as a VCO in Section 5.14.

Comments on integrating A/D converters

As with the dual-slope integration A/D, these charge-balancing methods all average the input over fixed time intervals, so they can be made insensitive to 60Hz pickup and its harmonics. In general, charge-balancing methods are inexpensive (they don't require a particularly good comparator, for instance) and accurate, and they produce a strictly monotonic output. However, they are slow, as compared with successive approximation. The AD1170 gives 18-bit resolution with 66ms conversion time, with a price of about $100. By comparison, the AD76 is a 16-bit successive-approximation converter with 15μs conversion time and $120 price tag. Compared with dual-slope converters, delta-sigma and switched-capacitor methods can be characterized by low-accuracy comparators following the integrators, but with precise charge-switching circuits, whereas dual-slope methods use highly repeatable endpoint comparators, with somewhat simpler switch requirements, at least from the viewpoint of speed and charge injection. Continuing our comparison of actual devices, the AD1175K is a multiple-slope 22-bit converter with 50ms conversion time and $800 price (Section 9.22).

One interesting point to keep in mind with any of the integrating techniques (single- and dual-slope integration and charge balancing) is that the input to the integrator can be either a current, or a voltage in series with a resistor. In fact, some converters have two input terminals, one tied directly to the summing junction for use with devices that source a current directly. When used with a current input, the offset voltage of the integrator becomes unimportant, whereas with a voltage input (with internal series resistor) the integrator op-amp produces an error equal to its input offset voltage. Current input is useful, therefore, to get wide dynamic range, particularly if the A/D is used with a device that has a current output anyway; examples are photomultipliers and photodiodes. Beware of this specmanship "gotcha," though: A/D accuracy may be specified assuming input current, for converters that accept either current or voltage input; don't expect as good performance for small inputs when using such a converter with voltage input.

Note that these charge-balancing methods all include a highly accurate V/F converter and that they can be used as such if an output *frequency* is desired, as indicated in Figure 9.58.

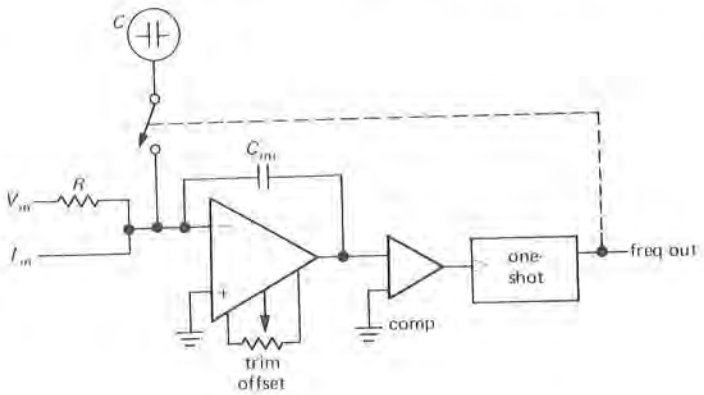


Figure 9.58. Charge-balancing V/F converter.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

□ 9.22 Some unusual A/D and D/A converters

Four recent introductions by Analog Devices, a traditional leader in conversion ICs and modules, are too interesting to pass by without comment.

□ *AD7569 combined DAC/ADC*

The monolithic AD7569 combines 8-bit A/D and D/A on a single chip, complete with sample/hold, internal clock, and reference (Fig. 9.59). The successive-approximation ADC converts in 2μs, and the DAC provides a voltage output with 1μs typical settling time. This chip is great for microprocessor systems: The converters share a single 8-bit digital port, with convenient control signals and fast timing (unlike many sluggish converter chips that require extra "wait" states and awkward setup times; see Chapters 10 and 11), and the chip can run from a single +5 volt supply. Furthermore, it needs no external components or trimming, the chip is packaged in a convenient 24-pin skinny-DIP, it runs on low power (60mW), and the price is right ($6 in 100 piece quantities).

□ *AD1175 22-bit integrating ADC*

This impressive module (Fig. 9.60) uses auto-zeroing multiple-slope integration to obtain 22-bit ($6\frac{1}{2}$-digit) accuracy, with unusual conversion speed (20 conversions per second). To get an idea what this means, consider the alternative – a benchtop (or rack-sized) meter, typically costing $4000, and delivering perhaps 2 conversions per second. The AD1175, by comparison, occupies 10 cubic inches, uses 3 watts, and costs $800 in single quantities. It contains an internal microprocessor and lets you set coarse gain and offset via its digital bus (which is used both for command inputs and converted data output).

□ *HDG0807 and AD9502 video converters*

These converters (Figs. 9.61 and 9.62) are just what you need for digital video. The HDG0807 is an 8-bit DAC with standard video signal levels and corresponding 75 ohm output impedance. Furthermore, it even generates the proper "composite sync" pulses, overlaid with the analog video signal, to create a complete video output signal. This converter is ready to use, fast (to 50MHz), and affordable. The AD9700 is the monolithic core of this hybrid converter; it operates to 100MHz.

The AD9502 video ADC does the opposite job, namely digitizing a video input signal. It has internal circuitry that strips the sync pulses from the analog composite

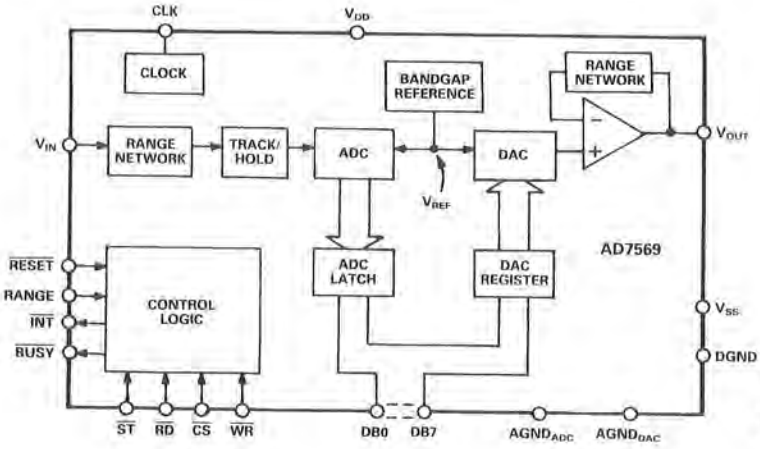


Figure 9.59. AD7569 combined 8-bit DAC/ADC. (Courtesy of Analog Devices.)

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

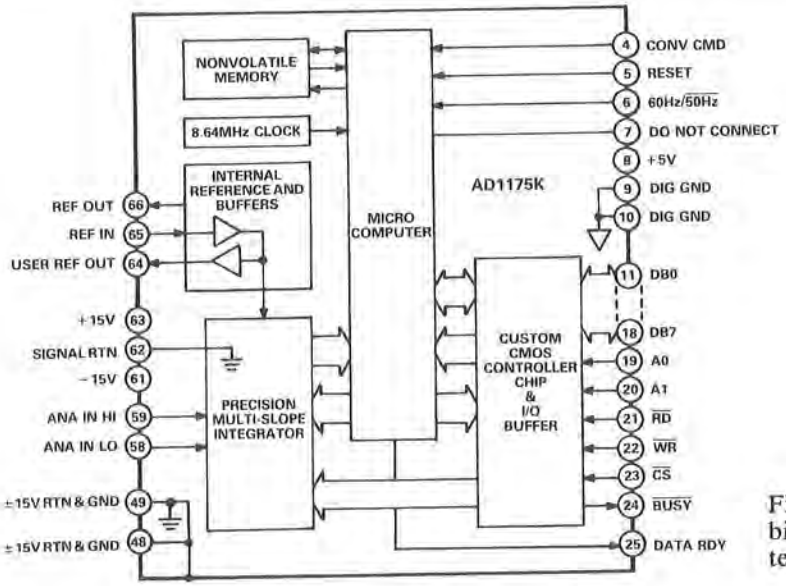


Figure 9.60. AD1175K 22-bit integrating ADC. (Courtesy of Analog Devices.)

video input, uses an internal phase-locked loop to generate a "pixel clock" synchronized to the raster sweep, then converts the analog voltage to 8-bit digital quantities. It can convert at a 13MHz rate, generating digitized outputs with 512×512 screen resolution.

9.23 Choosing an ADC

As with DACs, we've pulled together (Tables 9.5 and 9.6) a selection of ADCs that span a useful range of performance and price. We've tried to include both the most popular units and some of the recent

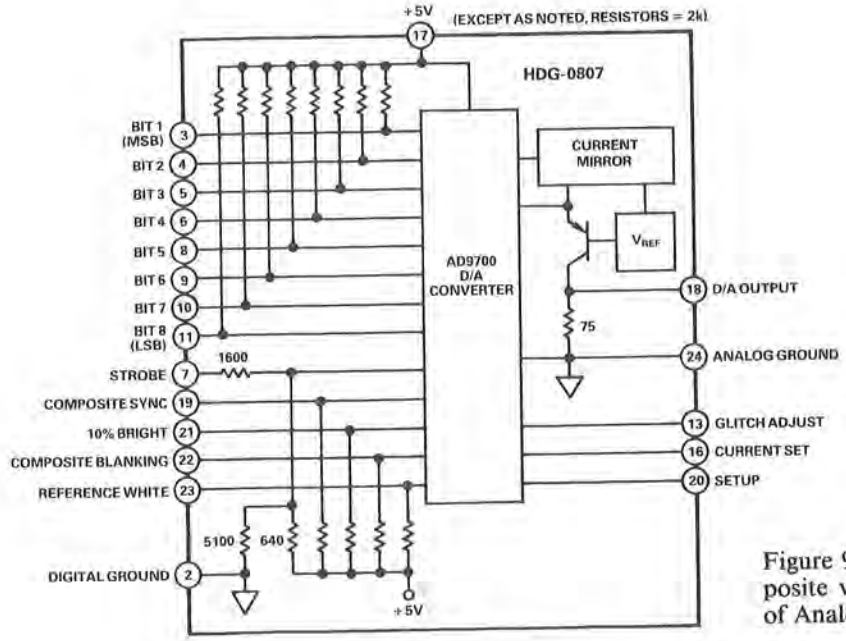


Figure 9.61. HDG0807 composite video DAC. (Courtesy of Analog Devices.)

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

TABLE 9.5. ANALOG-TO-DIGITAL CONVERTERS

Type	Mfg[a]	Bits	Method[b]	Speed max (µs)	Pkg[c]	Supply voltage (V)	Supply curr typ (mA)	V_{ref}	+5 ref OK?	Input range (V)
HS9582	HS	6	F	0.07	18	+5	30	ext 1-5	•	0-V_{ref}
TDC1047	TR	7	F	0.05	24	+5,-5.2	+20,-140	ext –1	–	0-V_{ref}
ADC0844	NS	8	SA	40	20	+5	1	ext 1-5	•	0-V_{ref}; 2V_{ref}
ADC0831	NS	8	SA	32	8	+5	1	ext 1-5	•	0-V_{ref}; 2V_{ref}
TLC548	TI	8	SA	22	8	+5	1.9	ext	•	0-5
AD670	AD	8	SA	10	20	+5	30	int	–	0.25; 2.5
AD7575	AD	8	SA	5	18	+5	3	ext 1.2	–	0-2V_{ref}
ADC0820	NS	8	HF	2.5	20	+5	8	ext 1-5	•	0-V_{ref}; 2V_{ref}
AD7820	AD	8	HF	1.6	20	+5	8	ext 1-5	•	0-V_{ref}
HS9583	HS	8	F	0.2	24	+5	20	ext +5V	•	0-5
AD9002	AD	8	F	0.007	28	-5.2	150	ext	–	-2 to 0
AD770	AD	8	F	0.005	40	+5,-5.2	270,125	ext	–	-1 to 1
CXA1176K	SO	8	F	0.003	68LCC	-5.2	300	ext	–	-2 to 0
TDC1049	TR	9	F	0.03	64	-5.2	950	ext –2	–	-2 to 0
HADC77600	HO	10	F	0.02	72PGA	+5,-5.2	+440,-380	ext ±2	–	±0.5; ±2
ADC1001	NS	10	SA	200	20	+5	1.5	ext 1-5	•	0-V_{ref}; 2V_{ref}
AD573	AD	10	SA	20	20	+5,-12	15,-9	int	–	0-10; -5 to 5
AD7578	AD	12	SA	100	24S	+5,±12	0.1,±3	ext +5V	•	0-5
AD574A	AD	12	SA	25	28	+5,±12	30,+2,-18	int 10.0	–	0-10; ±5; ±10
ADC80	BB	12	SA	25	32	+5,±12	11,+5,-21	int 6.3	–	0-5;0-10;±5;±10
AD7572	AD	12	SA	5	24S	+5,-15	5,-10	int –5.2	–	0-5
AD7672	AD	12	SA	3	24S	+5,-12	7,-12	ext –5	–	0-5; 0-10; ±5
AD578	AD+	12	SA	3	32	+5,±12	100,+3,-22	int 10.0	–	0-10;0-20;±5;±10
ADC511	DA	12	HF	1	24	+5,±15	65, ±25	int	–	0-10; ±5
AD9003	AD	12	HF	1	40	+5,±15	2.5W	int	–	0-5
THC1201	TR	12	HF?	0.1	46	±15	160	int	–	-1 to 1
CAV1220	AD	12	HF	0.05	PCB	±5,±15	20W	int	–	±1
TLC1205B	TI	13	SA	10	24	±5	3	ext	•	±5
ICL7115	IL	14	SA	40	40	±5	±2	ext +5V	•	0-5
ADC71	AD	16	SA	50	32HY	+5,±15	70,±20	int 6.3V	–	0-5;0-10;±5;±10
ADC76	BB+	16	SA	15	32	+5,±12	10,+14,-17	int	–	0-5;0-10;±5;±10
CX20018	SO	16	DS	9	28	±5	10,100	ext		±10
ADAM-826-3	AN	16	SA	1.5	PCB	+5,-6,±15	3W	int	–	0-10; ±10
MN5420	MN	20[h]	SA	3	40HY	+5,±15	+400,±150	int	–	±5

[a] see footnote to Table 4.1. [b] DS - dual slope; F - flash; HF - half-flash; SA - successive approx.
[c] all are DIP, unless otherwise noted; HY - hybrid; S - skinny-DIP. [d] AZ - auto-zero; NT - no trim req'd.
[e] 20k with internal amp. [f] 1000-pc qty. [g] needs external op-amps. [h] floating point.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

Type	Input				Output				Price (plastic 100 pcs) ($)	Comments
	Z_{in}	Diff'l?	S/H?	Trim?[d]	Bits	3-state?	Serial?	Clock		
HS9582	∞	–	–	NT	6	•	–	ext	15	fast
TDC1047	100k	–	–	NT	7	•	–	–	40	latched output
ADC0844	∞	–	–	AZ	8	•	–	int	3.85	4-input MUX; easy to use; +5V
ADC0831	∞	•	–	AZ	–	–	•	ext	2.70	small, no glue
TLC548	1µA	–	•	AZ	–	–	•	int		easy-to-use miniDIP
AD670	∞	•	–	NT	–	•	–	int	6	instrumentation amp input
AD7575	∞	–	•	NT	–	•	–	extRC	5.50	fast
ADC0820	∞	–	•	NT	–	•	–	int		latched output; overflow
AD7820	∞	–	•	NT	–	•	–	int	10	fast; 4 & 8-ch versions; overflow
HS9583	100k	–	–	NT	–	•	–	ext	44	fast
AD9002	20k	–	–	NT	–	–	–	–	90	fast, lo-pwr flash, w latch & ovflw
AD770	3.3k	–	–	NT	8	–	–	–	175	fast, w latch & ovflw
CXA1176K		–	–	NT	–		–	–		world speed champion
TDC1049	16k	–	–	•	10	–	–	–		9-bit flash
HADC77600	1k[e]	–	–	•	10	–	–	–		highest resolution flash
ADC1001	∞	•	–	AZ	8+2	•	–	extRC		5V, minimum glue
AD573	5k	–	–	NT	8+2	•	–	int	14	fast, no glue
AD7578	∞	–	–	AZ	8	•	–	extRC	20	low power, cheap
AD574A	5k	–	–	•	8+4	•	–	int	28	V_{ref} out; classic
ADC80	5k	–	–	•	12	–	•	int	33	V_{ref} out, classic, clk out
AD7572	2.5k	–	–	NT	8+4	•	–	int	46	fast, no glue, clk out
AD7672	5k	–	–	–	8+4	•	–	extRC	75	fast
AD578	5k	–	–	NT	12	–	•	int	100	fast, no glue, V_{ref} out, clk out
ADC511	2.5k	–	–	NT	12	•	–	int	99[f]	no missing codes over temp
AD9003	1k	–	•	NT	12	–	–	int	250	
THC1201		–	•	NT	12	•	–	–		very fast, easy to use
CAV1220	1k	–	•	NT	12	–	–	int	2500	fastest
TLC1205B	1µA	•	•	–	8+5	•	–	ext	30	
ICL7115	5k	–	–	AZ	8+6	•	–	ext	50	radix 1.85 + internal ROM
ADC71	5k	–	–	•	16	–	•	int	63	indus std, clk out, ref out
ADC76	2.5k	–	–	•	16	–	•	int	100	indus std; AD376
CX20018	10k[g]		•	•	–	–	•	ext	18	digital audio, 2 ch
ADAM-826-3	1.4k	–	–	NT	8+8	•	–	int		module; versions w S/H, buffer
MN5420	5k	–	–	–	16	•	–	ext		float pt: 4-bit exp, 12-bit mantissa

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

TABLE 9.6. INTEGRATING A/D CONVERTERS

Type	Mfg[a]	Chips	Bits	Method[b]	Conv/sec	Pkg[c]	V_{supply} (V)	V_{ref} (V)	5V ref OK?	Input range (V)	Input Z_{in} (Ω)	Input Ext trim?	Input Differential	Output Bits parallel	Output 3-state?	Output UART compat?	Output Clock	Comments
AD7552	AD	1	12[d]	QS	6	40	+12,±5	1 to 5	•	0 to ±0.5V_{ref}	1M	–	–	8+6	•	–	ext	
TSC804	TS	1	12[d]	QS	30	60	±5	int	–	±4	∞	–	•	8+6	•	•	xtal	8-channel MUX
ICL7109	IL	1	12[d]	DS	30	40	±5	0.2 to 2	–	0 to ±2V_{ref}	∞	–	•	8+6	•	•	xtal	industry standard
AD7550	AD	1	13	QS	25	40	+12,±5	1 to 5	•	0 to 0.5V_{ref}	1M	–	–	8+6	•	–	ext	
TSC800	TS	1	15[d]	DS	2.5	40	±5	0.2 to 2	–	0 to ±2V_{ref}	∞	–	•	8+8	•	•	xtal	
TSC850	TS	1	15[d]	TS	40	40	±5	1.6,0.025	–	0 to 2V_{ref}	∞	V_{ref}	•	8+8	•	–	ext	fast, low pwr (2mA)
ICL7104-16[e]	IL	2	16[d]	DS	3	40	+5,±15	int	–	0 to ±4	∞	V_{ref}	–	8+8+2	•	–	xtal	use 8052 for low I_{in}
CSZ5316	CR	1	16	CB	20k	18	±5	int	–	±2.75	30k	–	–	serial	•	–	ext	int S/H
AD1170	AD	1	18	CB	1000[f]	40	+5,±15	int	•	±5	100M	–	–	8+8+6	•	–	xtal	software trims
AD1175K	AD	1	22	QS	20	66M	+5,±15	int	–	±5	1000M	–	•	8+8+6	•	–	int	accurate, stable, software trims

[a] see footnote to Table 4.1. [b] CB - charge balancing; DS - dual slope; TS - triple slope; QS - quad slope. [c] M - module. [d] plus additional sign bit.
[e] plus 8068A companion chip. [f] prog conversion rate; value shown is max, at which resolution is degraded. Full resolution can be obtained at 50 conv/sec.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

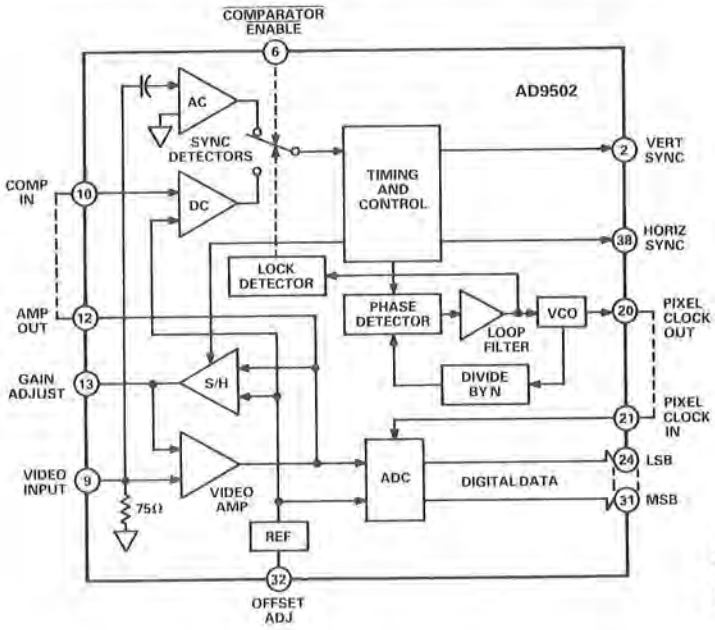


Figure 9.62. AD9502 composite video ADC. (Courtesy of Analog Devices.)

entries that should be next year's winners.

When looking for an ADC, some of the factors to consider are (a) precision, (b) speed, (c) accuracy (external trimming required? guaranteed monotonicity?), (d) required supply voltages (some work with +5V alone) and power dissipation, (e) small package, (f) reference and clock (internal or externally supplied? if external reference, is +5V OK? if internal, is it accessible externally, e.g. for ratiometric measurements? how good is it? can you override it?), (g) input impedance and analog voltage range (unipolar, bipolar, or both?), (h) input structure (differential? internal multiplexer or sample/hold? inverted polarity, i.e., more negative for larger output?), (i) output structure (parallel, serial, or both? is the parallel output "microprocessor-compatible," as a pair of individually enabled byte-wide groups?), and, of course, (j) price.

Complete A/D subsystems

If you need a high-accuracy A/D converter, particularly with an input multiplexer and sample/hold, you should look carefully at the "A/D subsystems" offered by several manufacturers. These are modules, rather than ICs, usually supplied as a metal box typically 0.4" thick and 2"×4" (or 3"×5") in size, with pin connections that mate with a special socket (or solder directly into a printed circuit board). Although these converters aren't cheap, they are delightfully easy to use. Better still, the manufacturers have solved the really thorny problems that plague high-resolution conversion, such as noise pickup, isolation of digital and analog sections, stable references and amplifier offsets, etc.

The DT-5716A from Data Translation (Fig. 9.63) is typical of these modules. It accepts 16 single-ended (or 8 differential) inputs, with an analog multiplexer at the input, followed by a sample/hold, programmable gain amplifier, and 16-bit A/D converter. It can convert at a 20kHz throughput rate, with the output organized as a pair of bytes for easy connection to a microprocessor bus (see Chapters 10 and 11).

A/D subsystem modules are available with resolution of 12 to 16 bits, both with

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

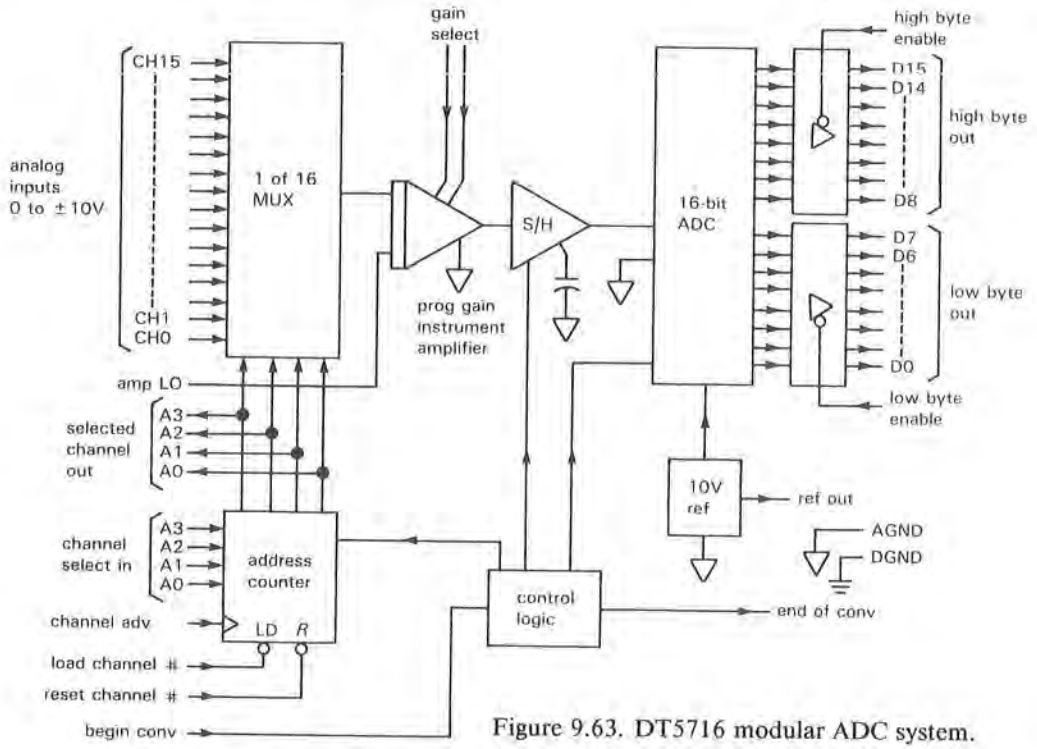


Figure 9.63. DT5716 modular ADC system.

and without input multiplexers. You pay a real premium for high accuracy and high speed, and most of the available modules are considerably cheaper than the units cited earlier. For example, the DAS1157-9 series from Analog Devices are single-channel converters with 14 to 16 bits of resolution and 18kHz throughput; they cost less than $300 in 100-pc quantities. You can get conversion modules from several companies, including Analog Devices, Analogic, Data Translation, and Intech.

SOME A/D CONVERSION EXAMPLES

9.24 16-Channel A/D data-acquisition system

Figure 9.64 shows a circuit to digitize any of 16 analog inputs on command, with a 12-bit digital output. This could form the "front end" of a microprocessor-controlled data-taking experiment.

The HI-506 is a 16-channel CMOS analog multiplexer with CMOS-compatible digital inputs. This particular multiplexer has some very nice properties. In particular, its switches are of the "break-before-make" variety, which means that the various input channels don't find themselves shorted together during address changes in the MUX. In addition, the inputs may swing beyond the supply rails, and when they do there is no "SCR latchup" or cross-talk between inputs. Watch out for considerations of this type when shopping for linear switches. They sometimes involve a compromise. For example, "break-before-make" results in a slower switching-time specification, since the "make" must be delayed to allow the switch to open.

The multiplexer's single analog output drives an LF398, a $2 monolithic sample/hold amplifier (Fig. 4.41) in convenient 8-pin DIP. The latter is used as a "track-and-hold," freezing the analog waveform only when conversion begins. With a

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

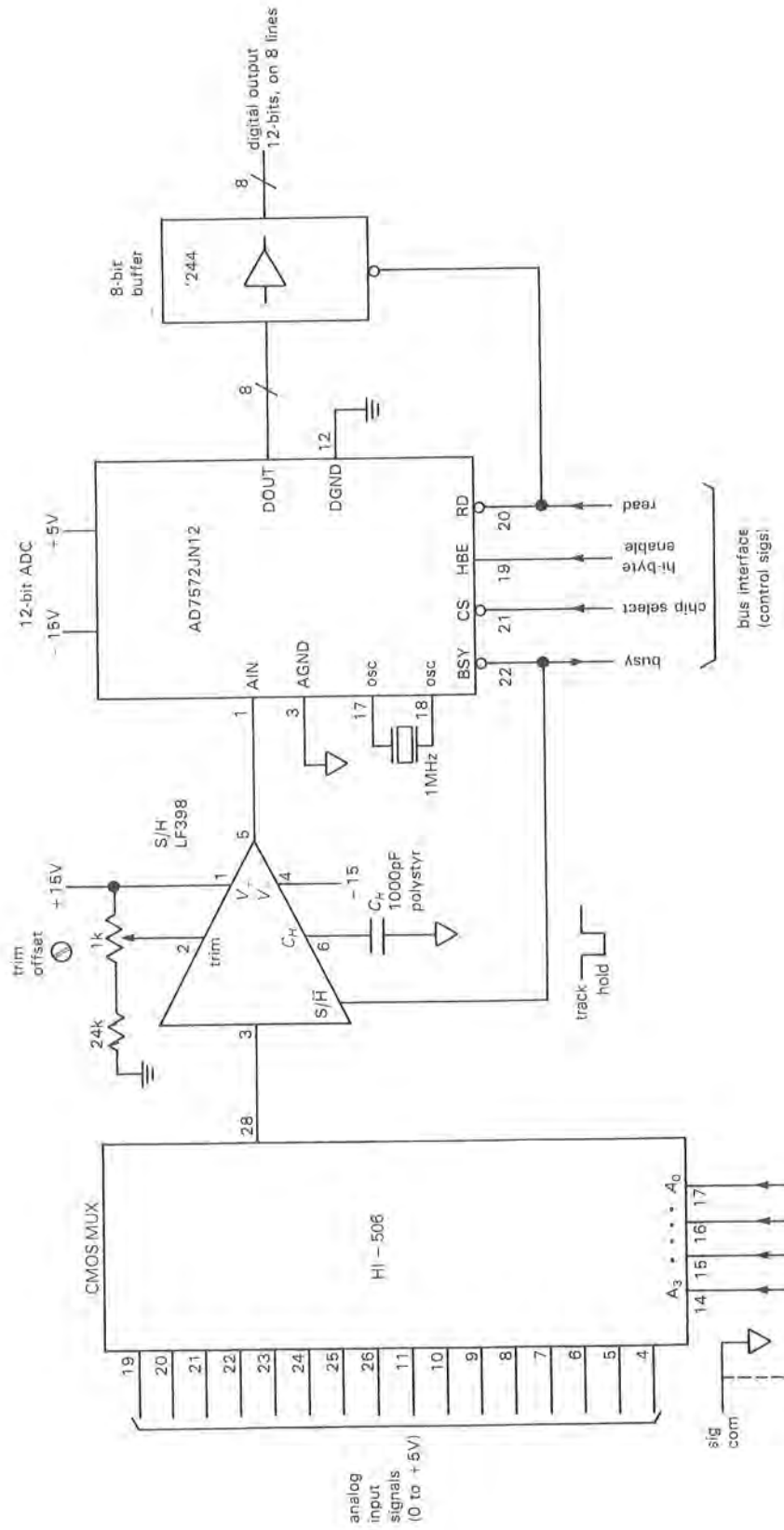


Figure 9.64. A 12-bit 16-channel successive-approximation A/D converter system ($12\mu s$/conversion).

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

1000pF capacitor the LF398 settles to 1 LSB in 0.5μs following "HOLD" and droops less than 3μV during the subsequent 12μs conversion. The AD7572 is a very nice low-power 12-bit ADC with on-chip voltage reference and clock; it has convenient controls for microprocessor interfacing, including the option of multiplexing the 12-bit result onto 8 lines (a "byte-wide" data bus) in two successive cycles.

The device controlling this circuit normally asserts an address to the multiplexer, then initiates a conversion by asserting CS′ and RD′. The ADC responds by asserting BUSY′, which freezes the analog input. The conversion is complete 12μs later, at which time BUSY′ goes HIGH. The full 12-bit result is then available, if you want to use all 12 D_{out} lines; however, if you have an 8-bit bus, you can read the least significant 8 bits first, then assert HBEN ("high-byte enable") to put the most significant 4 bits on $D_0 - D_3$.

After initiating conversion, the device controlling the converter can check BUSY′ to see when conversion is complete. A simpler alternative is just to wait 12μs (a software "timing loop" will waste the requisite time; see Chapter 11). The controlling device is obliged to wait 4μs after conversion is complete before initiating another conversion; this is the LF398's "acquisition time" – the time for the output to track the input again to within 0.1%. During that time the controller could be reading the digital output, of course. The overall conversion cycle is thus 16μs minimum, or 60,000 conversions per second.

Several points are worth making about this circuit: (a) For full 12-bit accuracy you have to provide an offset trim, to compensate for three errors: *(i)* the S/H has a V_{os} of 7mV (max), *(ii)* the S/H FET introduces a small voltage step at HOLD, owing to FET gate charge injection (Section 3.12), in this case a relatively stable 10mV negative step, and *(iii)* the ADC itself has a V_{os} specified as 4 LSB (equivalent to 5mV for a 0–5V input range). We put the trimming circuit on the LF398, using the manufacturer's recommended network. (b) The hold capacitor value is a compromise. Small values yield a short acquisition time, but introduce a larger droop and charge-injection step. The value we chose produces negligible droop and produces a HOLD step equivalent to 8 LSB; the step is relatively stable and can be canceled by proper adjustment of the offset trim control. (c) The circuit is wired for unipolar (0–5V) input; if you wanted bipolarity input signal range, you would have to add an op-amp offsetting circuit, taking care to keep errors less than 1 LSB (1 part in 4000). The AD7572 provides a precision reference voltage out, to make your job easier, but it still takes a bunch of components. (d) A nice addition to this sort of circuit is a programmable-gain amplifier, so the controlling microprocessor can command a change of gain to accommodate a range of input signal amplitudes. The AD526 from Analog Devices is a single-chip solution, providing programmable gains of 1, 2, 4, 8, and 16, with gain accuracy of 0.02% (12-bit accuracy); the LF13006/7 from National is an alternative, containing the resistors and FET switches (but not the amplifier itself) to set gains of 1 to 128 (by factors of 2) or 1 to 100 (1-2-5 sequence); these parts have a gain accuracy of 0.5% (8-bit accuracy).

A successive-approximation A/D is the natural choice here, since speed is important when jumping from one input to another. We picked components to minimize cost; the circuit shown will cost about $50 at today's prices, dominated by the $35 converter.

9.25 $3\frac{1}{2}$-Digit voltmeter

Figure 9.65 shows a circuit that exploits the advantages of dual-slope integration. Almost the entire circuit is included in the

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

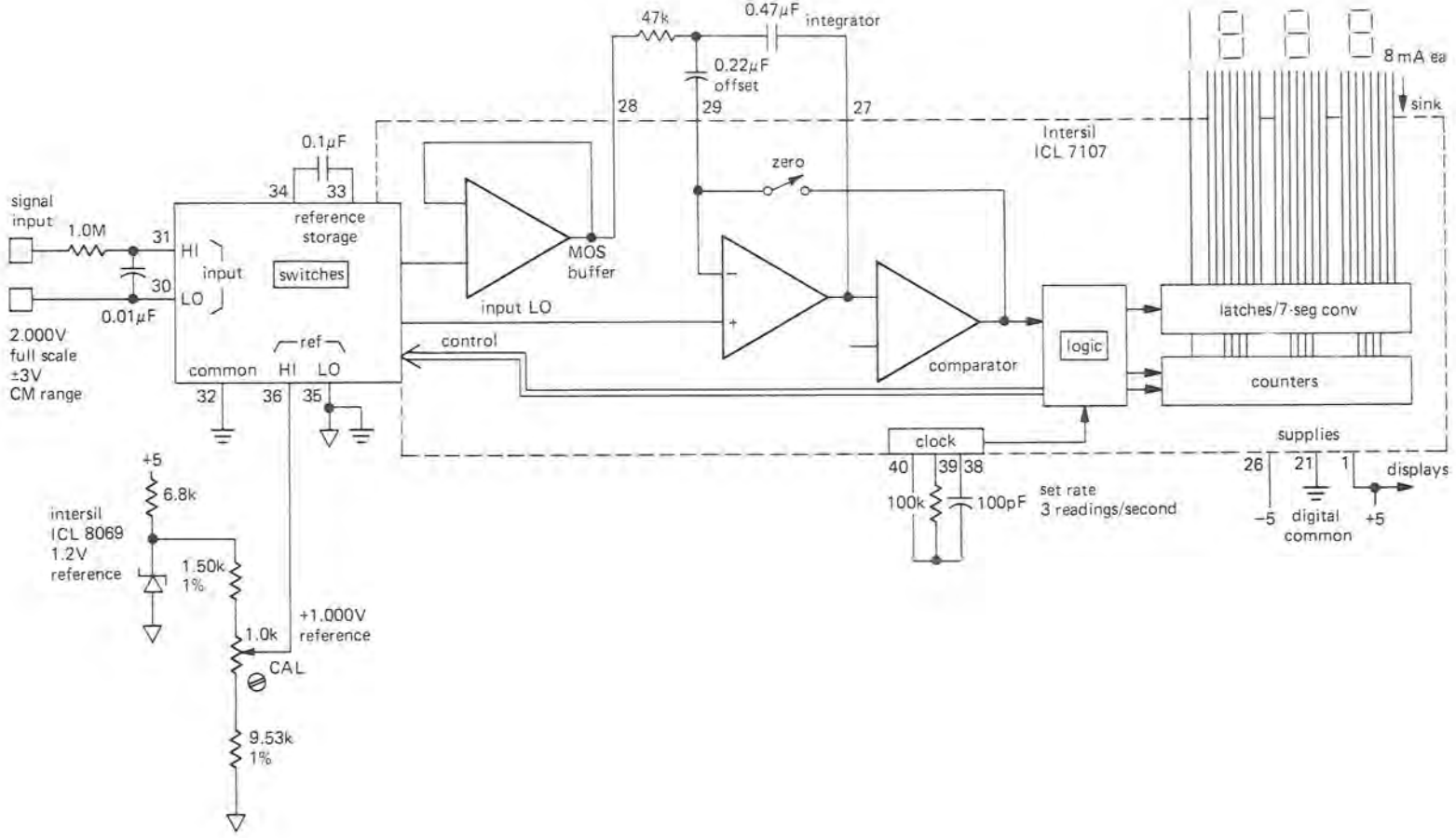


Figure 9.65. Single-chip $3\frac{1}{2}$-digit dual-slope DVM.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

digital-voltmeter CMOS LSI chip, the only external components being the integrator and clock RCs, an accurate voltage reference, and the display itself. The ICL 7107 includes an automatic zeroing cycle in its operation, and it even generates all the 7-segment multiplexed outputs to drive a 4-digit LED display directly. By using an external attenuator at the input (or a reference of a different voltage), you can generate other full-scale voltage ranges. Dual-slope conversion is well suited to DVM operation because it provides good accuracy (including auto-zeroing) and 60Hz rejection in an averaging instrument at low cost; the converter chip used here costs less than $20.

□ 9.26 Coulomb meter

The circuit in Figure 9.66 is a charge-balancing current integrator, or "coulomb meter." This instrument can be used to measure the integrated current (total charge) over some time period; it might find application in electrophoresis or electrochemistry. The action begins in the lower left-hand corner, where the current to be integrated flows through a precision 4-wire power resistor, generating a proportional voltage. IC_2 is a relatively low cost (under $5) precision single-supply op-amp with low initial voltage offset (80μV max) and low drift of offset with time and temperature (less than 2μV per degree and 0.5μV per month). It generates an output current, programmed by the current being measured, to drive the charge-balancing integrator, IC_3. Five decades of input sensitivity are selectable via the rotary switch at the input, with 200μA collector current in Q_1 corresponding to full-scale input in any range. Q_1 is a MOSFET (rather than a BJT), to eliminate control current error.

The charge-balancing circuitry is a standard delta-sigma scheme, with p-channel enhancement-mode MOSFET Q_2 doling out parcels of charge as directed by the state of flip-flop IC_{5a} after each clock cycle. IC_{5b} acts as a monostable, incrementing binary scaler IC_7 for each clock cycle during which Q_2 conducts. This circuit doesn't count for a fixed number of clock cycles, but simply integrates until it is stopped. The 4-digit counters IC_9 and IC_{10} deep track of the total charge, driving an 8-digit LED display.

If the current being measured ever exceeds the full-scale current of the range selected, Q_2 will be unable to balance Q_1's current even if it is ON continuously, and the measured charge registered in the counters will be in error. IC_{4a} checks for this overrange condition, lighting the LED if the integrator output rises past a fixed reference voltage (chosen comfortably larger than the output excursions of the integrator under normal conditions).

□ *Design calculations*

There are several interesting decisions that have to be made in designing a circuit like this. For instance, most of the CMOS logic is operated from +15 volts in order to simplify switching of Q_2. Since the 4-digit counters require +5 volts, a 4049 is used to interface the high-level CMOS logic signals to the counters. IC_4 is operated with single supply so that its output goes between ground and +15, for simple connection to IC_{5a}. The reference voltage for the integrator and comparator is put at about +4.7 volts by zener D_2, in order to allow headroom for Q_1; a simple zener is fine, since no accuracy is needed here. Note that a precision reference rides on the +4.7 volt level used to scale the current switched into the integrator. The REF-02's operating current is conveniently used to bias the zener.

The choice of switch (Q_2) can critically affect the overall precision of the instrument. If it has too much capacitance, the additional charge residing on its drain will

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

cause error. The scheme used in the previous circuit example (switching to ground during current-OFF cycles) is not used here because offset voltage errors in IC_3 will cause a fixed error at very low currents. You get increased dynamic range at the expense of some accuracy (owing to residual charge on Q_2's drain that gets integrated each cycle) by using an SPST switch configuration as shown. Note that the integrator op-amp chosen is a low-bias-current MOSFET type for negligible current error (10pA typ). Since FET op-amps tend to have larger voltage offsets than bipolar transistor types, this choice of op-amp will aggravate the dynamic-range problem just discussed if SPDT switching is used.

□ ***Dynamic range***

It is important to understand that this instrument is designed to have a large dynamic range, accurately integrating a current that may vary over several orders of magnitude during the measurement. That is why great care has been exercised in the design of the "front end," using a precision op-amp with an offset trimming scheme capable of precise adjustment (the normal trimming scheme usually has a total range of a few millivolts, making it difficult to adjust the offset precisely to zero). With IC_2 trimmed to $10\mu V$ offset or better, the instrument's dynamic range exceeds 10,000:1.

PHASE-LOCKED LOOPS

9.27 Introduction to phase-locked loops

The phase-locked loop (PLL) is a very interesting and useful building block, available from several manufacturers as a single integrated circuit. A PLL contains a phase detector, amplifier, and voltage-controlled oscillator (VCO) and represents a blend of digital and analog techniques all in one package. A few of its applications, which we will discuss shortly, are tone decoding, demodulation of AM and FM signals, frequency multiplication, frequency synthesis, pulse synchronization of signals from noisy sources (e.g., magnetic tape), and regeneration of "clean" signals.

There has traditionally been some reluctance to use PLLs, partly because of the complexity of discrete PLL circuits and partly because of a feeling that they cannot be counted on to work reliably. With inexpensive and easy-to-use PLLs now widely available, the first barrier to their acceptance has vanished. And with proper design and conservative application, the PLL is as reliable a circuit element as an op-amp or flip-flop.

Figure 9.67 shows the classic PLL configuration. The phase detector is a device that compares two input frequencies, generating an output that is a measure of their phase difference (if, for example, they differ in frequency, it gives a periodic output at the difference frequency). If f_{IN} doesn't equal f_{VCO}, the phase-error signal, after being filtered and amplified, causes the VCO frequency to deviate in the direction of f_{IN}. If conditions are right (lots more on that soon), the VCO will quickly "lock" to f_{IN}, maintaining a fixed phase relationship with the input signal.

At that point the filtered output of the phase detector is a dc signal, and the control input to the VCO is a measure of the input frequency, with obvious applications to tone decoding (used in digital transmission over telephone lines) and FM detection. The VCO output is a locally generated frequency equal to f_{IN}, thus providing a clean replica of f_{IN}, which may itself be noisy. Since the VCO output can be a triangle wave, sine wave, or whatever, this provides a nice method of generating a sine wave, say, locked to a train of input pulses.

In one of the most common applications of PLLs, a modulo-n counter is hooked

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

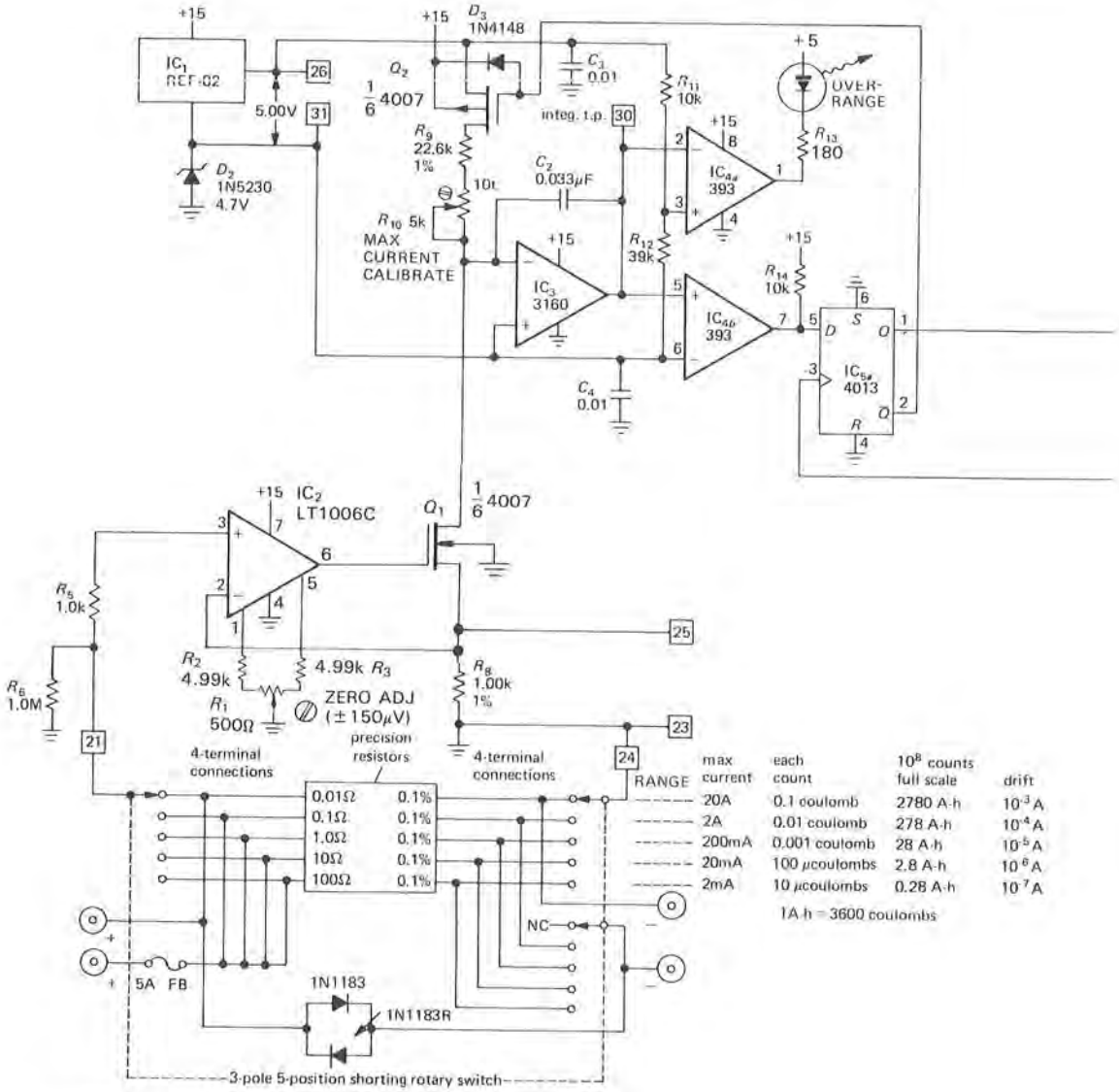


Figure 9.66. Coulomb meter: accumulated charge counter.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

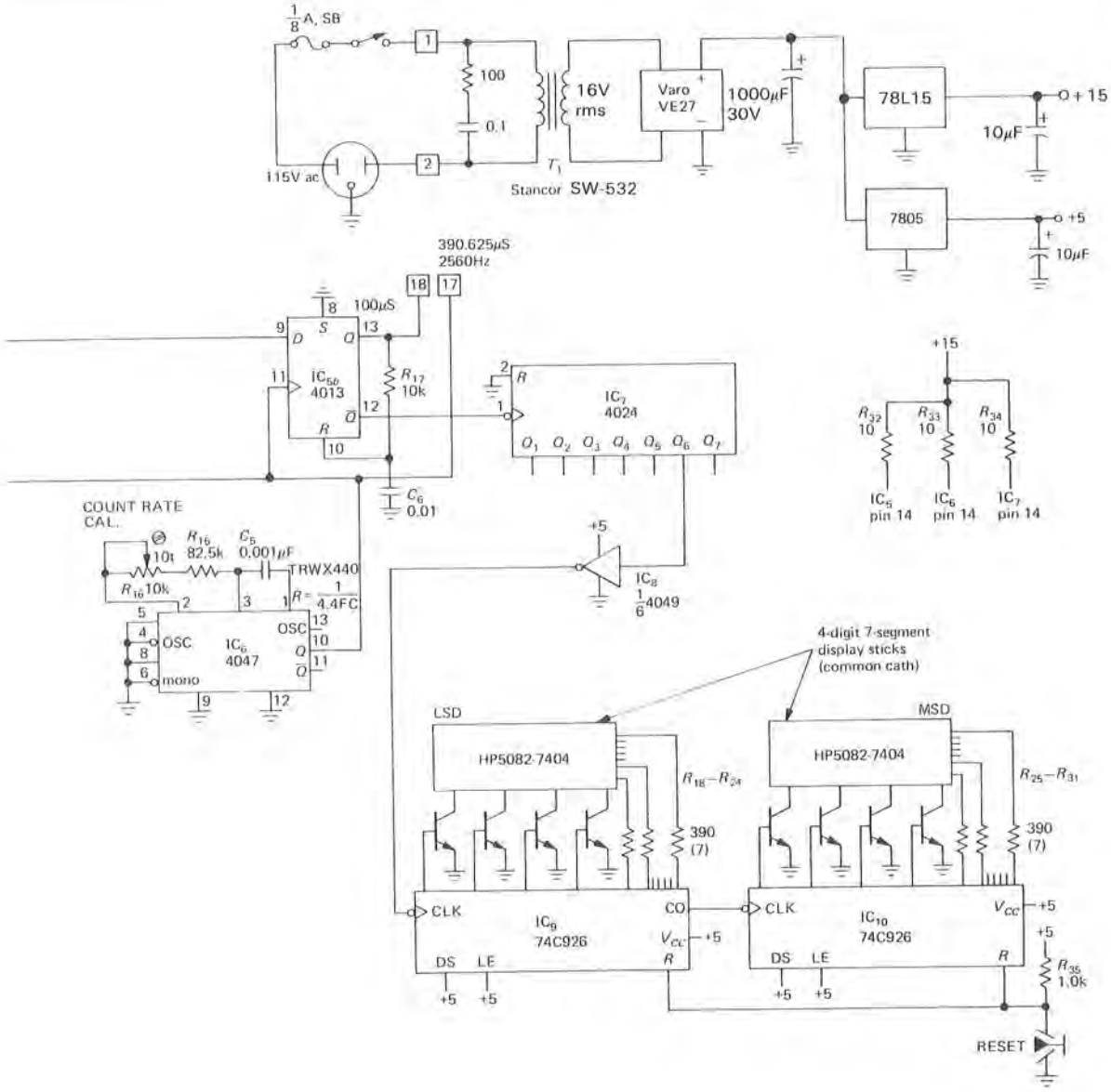


Figure 9.66. (*cont.*)

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

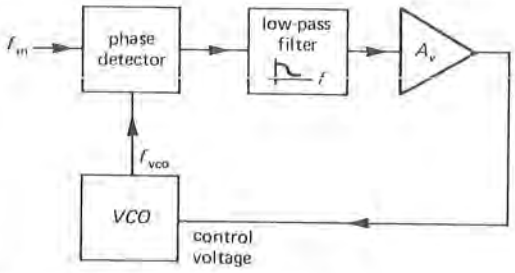


Figure 9.67. Phase-locked loop.

between the VCO output and the phase detector, thus generating a multiple of the input reference frequency f_{IN}. This is an ideal method for generating clocking pulses at a multiple of the power-line frequency for integrating A/D converters (dual-slope, charge-balancing), in order to have infinite rejection of interference at the power-line frequency and its harmonics. It also provides the basic technique of frequency synthesizers.

☐ PLL components

☐ *Phase detector.* Let's begin with a look at the phase detector. There are actually two basic types, sometimes referred to as type I and type II. The type I phase detector is designed to be driven by analog signals or digital square-wave signals, whereas the type II phase detector is driven by digital transitions (edges). They are typified by the 565 (linear, type I) and the CMOS 4046, which contains both.

The simplest phase detector is the type I (digital), which is simply an exclusive-OR gate (Fig. 9.68). With low-pass filtering, the graph of the output voltage versus phase difference is as shown, for input square waves of 50% duty cycle. The type I (linear) phase detector has similar output-voltage-versus-phase characteristics, although its internal circuitry is actually a "four-quadrant multiplier," also known as a "balanced mixer." Highly linear phase detectors of this type are essential for *lock-in detection*, a lovely technique we will discuss in Section 15.15.

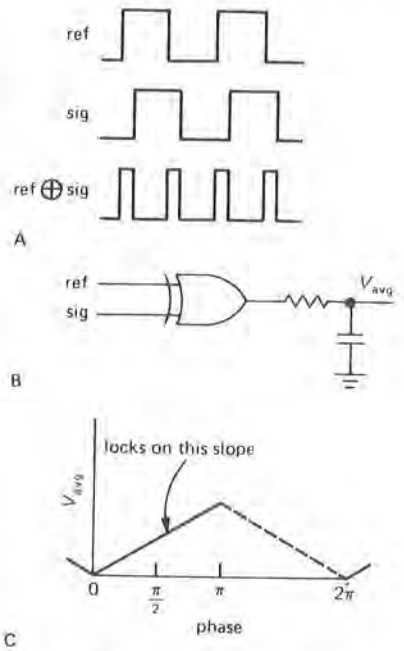


Figure 9.68. Exclusive-OR-gate phase detector (type I).

The type II phase detector is sensitive only to the relative timing of *edges* between the signal and VCO input, as shown in Figure 9.69. The phase comparator circuitry generates either *lead* or *lag* output pulses, depending on whether the transitions of the VCO output occur before or after the transitions of the reference signal, respectively. The width of these pulses is equal to the time between the respective edges, as shown. The output circuitry then either sinks or sources current (respectively) during those pulses and is otherwise open-circuited, generating an average output-voltage-versus-phase difference like that in Figure 9.70. This is completely independent of the duty cycle of the input signals, unlike the situation with the type I phase comparator discussed earlier. Another nice feature of this phase detector

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

is the fact that the output pulses disappear entirely when the two signals are in lock. This means that there is no "ripple" present at the output to generate periodic phase modulation in the loop, as there is with the type I phase detector.

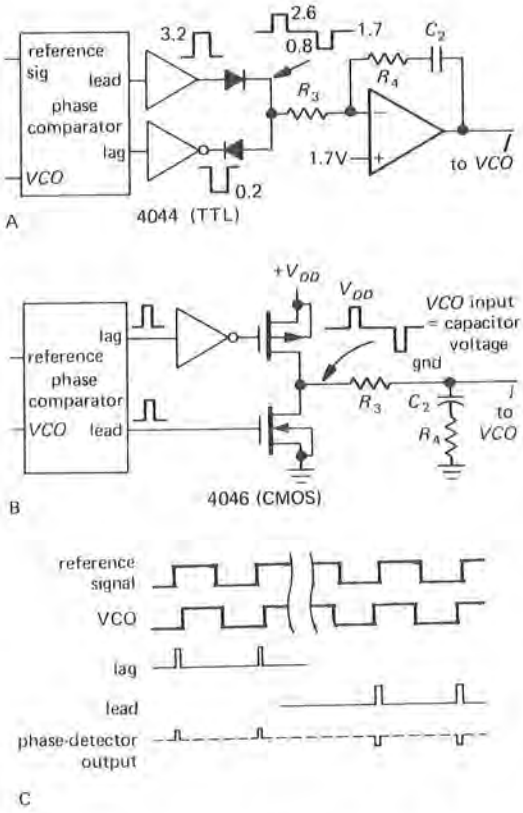


Figure 9.69. Edge-sensitive lead-lag phase detector (type II).

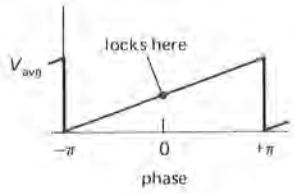


Figure 9.70

Here is a comparison of the properties of the two basic types of phase detector:

	Type I exclusive-OR	*Type II edge-triggered ("charge pump")*
Input duty cycle	50% optimum	irrelevant
Lock on harmonic?	yes	no
Rejection of noise	good	poor
Residual ripple at $2f_{IN}$	high	low
Lock range (L)	full VCO range	full VCO range
Capture range	$fL(f < 1)$	L
Output frequency when out of lock	f_{center}	f_{min}

There is one additional point of difference between the two kinds of phase detectors. The type I detector is always generating an output wave, which must then be filtered by the loop filter (much more on this later). Thus in a PLL with type I phase detector, the loop filter acts as a low-pass filter, smoothing this full-swing logic-output signal. There will always be residual ripple, and consequent periodic phase variations, in such a loop. In circuits where phase-locked loops are used for frequency multiplication or synthesis, this adds "phase-modulation sidebands" to the output signal (see Section 13.18).

By contrast, the type II phase detector generates output pulses only when there is a phase error between the reference and VCO signal. Since the phase detector output otherwise looks like an open circuit, the loop filter capacitor then acts as a voltage-storage device, holding the voltage that gives the right VCO frequency. If the reference signal moves away in frequency, the phase detector generates a train of short pulses, charging (or discharging) the capacitor to the new voltage needed to put the VCO back into lock.

□ *VCOs.* An essential component of a phase-locked loop is an oscillator whose frequency can be controlled by the phase

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

detector output. Some PLL ICs include a VCO (e.g., the linear 565 and the CMOS 4046). Then there are separate VCO chips, listed in Table 5.4. An interesting class of VCOs is composed of the sine-wave-output types (8038, 2206, etc.), since they let you generate a clean sine wave locked to some horrendous input waveform. Don't overlook yet another class of VCOs, namely "voltage-to-frequency converters" (V/F), which are generally designed to optimize linearity; they tend to have a modest maximum frequency (1MHz or less), and logic-level pulses as output (see Section 5.15).

One thing to keep in mind is the fact that the VCO doesn't have to be restricted to logic speeds. You could, for instance, use a radiofrequency oscillator tuned with a varactor (variable capacitor) diode (Fig. 9.71).

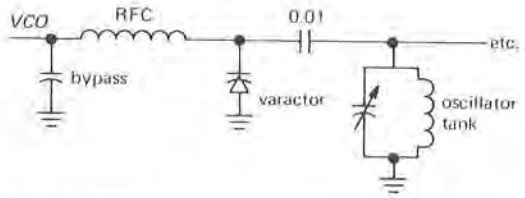


Figure 9.71

Carrying this idea one step further, you could even use something like a reflex klystron, a microwave (gigahertz) oscillator that is electrically tuned by varying the voltage on the *repeller*. Of course, a phase-locked loop built with such oscillators would require a radiofrequency phase detector.

A VCO for use in a phase-locked loop doesn't have to be particularly linear in its frequency-versus-control-voltage characteristic, but if it is highly nonlinear, the loop gain will vary according to the signal frequency, requiring better loop stability.

□ 9.28 PLL design

□ *Closing the loop*

The phase detector gives us an error signal related to the phase difference between the signal and reference inputs. The VCO allows us to control its frequency with a voltage input. It would seem straightforward to treat this like any other feedback amplifier, closing the loop with some gain, just as we did with op-amp circuits.

However, there is one essential difference. Previously, the quantity adjusted by feedback was the same quantity measured to generate the error signal, or at least a proportional quantity. For example, in a voltage amplifier we measured output voltage and adjusted input voltage accordingly. In a PLL there's an integration; we measure *phase*, but adjust *frequency*, and phase is the integral of frequency. This introduces a 90° phase shift in the loop.

This integrator included within the feedback loop has important consequences, since an additional 90° of lagging phase shift at a frequency where the loop gain is unity can produce oscillations. A simple solution is to avoid any further lagging components within the loop, at least at frequencies where the loop gain is close to unity. After all, op-amps have a 90° lagging phase shift over most of their frequency range, and they work quite nicely. This is one approach, and it produces what is known as a "first-order loop." It looks just like the PLL block diagram shown earlier, with the low-pass filter omitted.

Although they are useful in many circumstances, first-order loops don't have the desirable property of acting as a "flywheel," allowing the VCO to smooth out noise or fluctuations in the input signal. Furthermore, a first-order loop will not maintain a fixed phase relationship between the reference and VCO signals, since the phase detector output drives the VCO

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

directly. A "second-order loop" has additional low-pass filtering within the feedback loop (as drawn earlier), carefully designed to prevent instabilities. This provides flywheel action and also reduces the "capture range" and increases the capture time. Furthermore, with type II phase detectors, a second-order loop guarantees phase lock with zero phase difference between reference and VCO, as will be explained later. Second-order loops are used almost universally, since the applications of phase-locked loops usually demand an output frequency with low phase noise and some "memory," or flywheel action. Second-order loops permit high loop gain at low frequencies, resulting in high stability (in analogy with the virtues of high loop gain in feedback amplifiers). Let's get right down to business, illustrating the use of phase-locked loops with a design example.

□ 9.29 Design example: frequency multiplier

Generating a fixed multiple of an input frequency is one of the most common applications of PLLs. This is done in frequency synthesizers, where an integer multiple n of a stable low-frequency reference signal (1Hz, say) is generated as an output; n is settable digitally, giving you a flexible signal source that can even be controlled by a computer. In more mundane applications, you might use a PLL to generate a clock frequency locked to some other reference frequency already available in the instrument. For example, suppose we want to generate a 61,440Hz clock signal for a dual-slope A/D converter. That particular choice of frequency permits 7.5 measurement cycles per second, allowing 4096 clock periods for the ramp-up (remember that dual-slope conversion uses a constant time interval) and 4096 counts full scale for the constant-current ramp-down. The unique virtue of a PLL scheme is that the 61.440kHz clock can be locked to the 60Hz power line ($61{,}440 = 60 \times 1024$), giving infinite rejection of 60Hz pickup present on any signal input to the converter, as we discussed in Section 9.21.

We begin with the standard PLL scheme, with a divide-by-n counter added between the VCO output and the phase detector (Fig. 9.72). In this diagram we have indicated the units of gain for each function in the loop. That will be important in our stability calculations. Note particularly that the phase detector converts phase to voltage and that the VCO converts voltage to the time derivative of phase (i.e., frequency). This has the important consequence that the VCO is actually an integrator, thinking of phase as the variable in the lower part of the diagram; a fixed input voltage error produces a linearly rising phase error at the VCO output. The low-pass filter and the divide-by-n counter both have unitless gain.

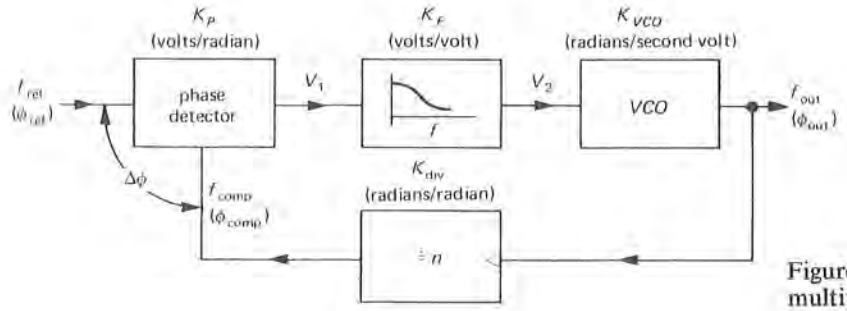


Figure 9.72. Frequency multiplier block diagram.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

☐ *Stability and phase shifts*

The trick to a stable second-order phase-locked loop is shown in the Bode plots of loop gain in Figure 9.73. The VCO acts as an integrator, with $1/f$ response and 90° lagging phase shift (i.e., its response is proportional to $1/j\omega$, a current source driving a capacitor). In order to have a respectable phase margin (the difference between 180° and the phase shift around the loop at the frequency of unity loop gain), the low-pass filter has an additional resistor in series with the capacitor to stop the rolloff at some frequency (fancy name: a "zero"). The combination of these two responses produces the loop gain shown. As long as the loop gain rolls off at 6dB/octave in the neighborhood of unity loop gain, the loop will be stable. The "lead-lag" low-pass filter does the trick, if you choose its properties correctly (this is exactly the same as lead-lag compensation in op-amps). Next you will see how it is done.

☐ *Loop gain calculations*

Figure 9.74 shows the schematic of the 61.440Hz PLL synthesizer. Both the phase detector and VCO are parts of a 4046 CMOS PLL. We have used the edge-triggered type of phase detector in this circuit (the 4046 actually contains both kinds). Its output comes from a pair of CMOS transistors generating saturated pulses to V_{DD} or ground. It is really a three-state output as explained earlier, since it is in the high-impedance state except during actual phase-error pulses.

The VCO allows you to set the minimum and maximum frequencies corresponding to control voltages of zero and V_{DD}, respectively, by choosing R_1, R_2, and C_1 according to some design graphs. We have made the choices shown. Note: The 4046 has chronic severe supply sensitivity disease; check the graphs on the data sheet. The rest of the loop is standard PLL procedure.

Having rigged up the VCO range, the remaining task is the low-pass-filter design. This part is crucial. We begin by writing down the loop gain, as in the "PLL gain calculation" box, considering each component (refer to Fig. 9.72). Take special pains here to keep your units consistent; don't switch from f to ω or (worse) from hertz to kilohertz. The only gain term still to be decided is K_F. We do this by writing down the overall loop gain, remembering that the VCO is an integrator:

$$\phi_{\text{out}} = \int V_2 K_{\text{VCO}} dt$$

PLL GAIN CALCULATION

Component	*Function*	*Gain*	*Gain calculation* ($V_{DD} = +10$V)
Phase detector	$V_i = K_P \Delta\phi$	K_P	0 to $V_{DD} \leftrightarrow 0°$ to 360°
Low-pass filter	$V_2 = K_F V_1$	K_F	$K_F = \dfrac{1 + j\omega R_4 C_2}{1 + j\omega(R_3 C_2 + R_4 C_2)}$ volts/volt
VCO	$\dfrac{d\phi_{\text{out}}}{dt} = K_{\text{VCO}} V_2$	K_{VCO}	20kHz ($V_2 = 0$) to 200kHz ($V_2 = 10$V) $\rightarrow K_{\text{VCO}} = 18$kHz/volt $= 1.13 \times 10^5$ radians/second-volt
Divide-by-n	$\phi_{\text{comp}} = \frac{1}{n}\phi_{\text{out}}$	K_{div}	$K_{\text{div}} = \frac{1}{n} = \frac{1}{1024}$

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

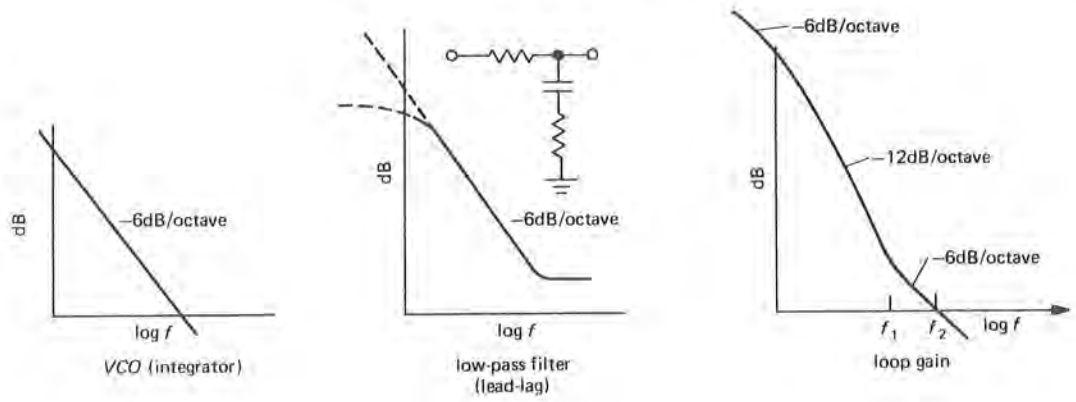


Figure 9.73. PLL Bode plots.

The loop gain is therefore given by

$$\text{Loop gain} = K_P K_F \frac{K_{\text{VCO}}}{j\omega} K_{\text{div}}$$

$$= 1.59 \times \frac{1 + j\omega R_4 C_2}{1 + j\omega(R_3 C_2 + R_4 C_2)} \times \frac{1.13 \times 10^5}{j\omega} \times \frac{1}{1024}$$

Now comes the choice of frequency at which the loop gain should pass through unity. The idea is to pick a unity-gain frequency high enough so that the loop can follow input frequency variations you want to follow, but low enough to provide flywheel action to smooth over noise and jumps in the input frequency. For example, a PLL designed to demodulate an FM input signal, or decode a rapid sequence of input tones, needs to have rapid response (for the FM input signal, the loop should have as much bandwidth as the input signal, i.e., response up to the maximum modulating frequency, while to decode input tones, its response time must be short compared with the time duration of the tones). On the other hand, a loop such as this one, designed to generate a fixed multiple of a stable and slowly varying input frequency, should have a low

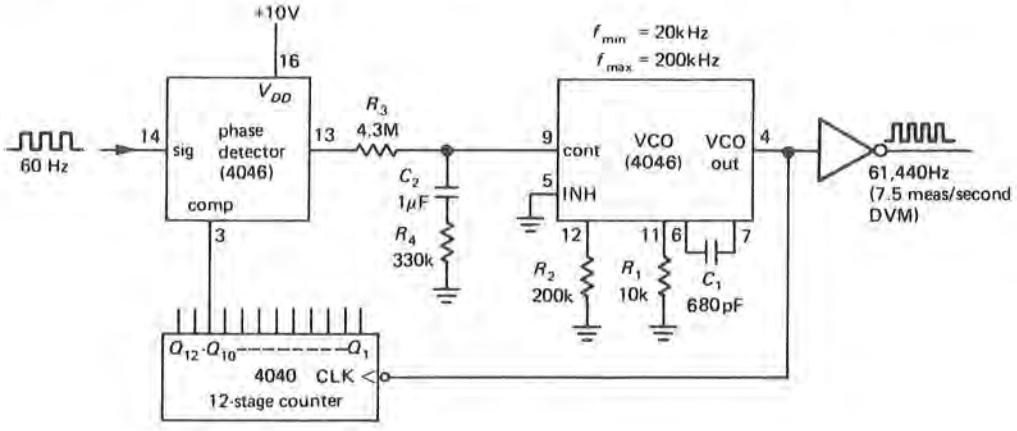


Figure 9.74. Using a PLL multiplier to generate a clock locked to the 60Hz ac line.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

unity-gain frequency. That will reduce phase noise at the output and make the PLL insensitive to noise and glitches on the input. It will hardly even notice a short dropout of input signal, because the voltage held on the filter capacitor will instruct the VCO to continue producing the same output frequency.

In this case, we choose the unity-gain frequency f_2 to be 2Hz, or 12.6 radians per second. This is well below the reference frequency, and you wouldn't expect genuine power-line frequency variations on a scale shorter than this (remember that the 60Hz power is generated by enormous generators with lots of mechanical inertia). As a rule of thumb, the breakpoint of the low-pass filter (its "zero") should be lower by a factor of at least 3 to 5, for comfortable phase margin. Remember that the phase shift of a simple RC goes from 0° to 90° over a frequency range of roughly 0.1 to 10 times the −3dB frequency (its "pole"), with a 45° phase shift at the −3dB frequency. In this case we put the frequency of the zero, f_1, at 0.5Hz, or 3.1 radians per second (Fig. 9.75). The breakpoint f_1 determines the time constant R_4C_2 : $R_4C_2 = 1/2\pi f_1$. Tentatively, take $C_2 = 1\mu$F and $R_4 = 330$k. Now all we do is choose R_3 so that the magnitude of the loop gain equals 1 at f_2. In this case, that works out to $R_3 = 4.3\text{M}\Omega$.

EXERCISE 9.5

Show that these choices of filter components actually give a loop gain of magnitude 1.0 at $f_2 = 2.0$Hz

Sometimes the filter values are inconvenient, so you have to readjust them, or move the unity-gain frequency somewhat. With a CMOS phase-locked loop these values are acceptable (the VCO input terminal has a typical input impedance of $10^{12}\Omega$). With bipolar transistor PLLs (the 4044, for example), you might want to use an external op-amp to buffer the impedances.

We used an edge-triggered (type II) phase detector in this circuit example because of its simplified loop filter; in practice that might not be the best choice for a PLL locked to the 60Hz power line because of the relatively high noise level present on the 60Hz signal. With careful design of the analog input circuit (e.g., a low-pass filter followed by a Schmitt trigger) it would probably perform well; otherwise an exclusive-OR (type I) phase detector should be used.

□ *"Cut and try"*

For some people, the art of electronics consists in fiddling with filter component values until the loop "works." If you are one of those, we will oblige you by looking the other way. We have presented these loop calculations in detail because we suspect that much of the PLL's bad reputation is the result of too many people "looking the other way." Nevertheless, we can't resist supplying a hot tip for cut-and-try addicts: R_3C_2 sets the smoothing (response) time of the loop, and R_4/R_3 determines the damping, i.e., absence of overshoot for step changes in frequency. You might begin with $R_4 = 0.2R_3$.

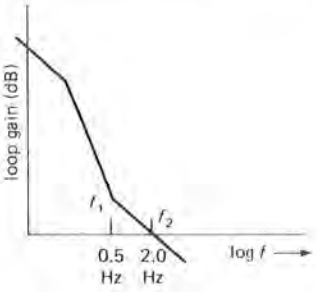


Figure 9.75

□ *Video clock generation*

Another nice application of a high-frequency oscillator locked to the 60Hz power line is in video signal generation, as

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

in alphanumeric computer terminals. The standard video display rate is 30 pictures per second. Since a small amount of 60Hz pickup is almost inevitable, the picture will "weave" slowly sideways unless the vertical video sync rate is locked exactly to the power-line frequency. PLLs provide a nice way. You just lock a high-frequency VCO (around 15MHz) to a predetermined multiple of 60Hz, so that subdivisions of that high-frequency clock generate (successively) the dots for each displayed character, the number of characters on each line, and the number of horizontal lines in each picture.

□ 9.30 PLL capture and lock

Once locked, it is clear that a PLL will stay locked as long as the input frequency doesn't wander outside the range of the feedback signal. An interesting question to ask is how PLLs get locked in the first place. After all, an initial frequency error results in a periodic output from the phase detector at the difference frequency. After filtering by the low-pass filter, it is reduced to small-amplitude wiggles, rather than a nice clean dc error signal.

□ *Capture transient*

The answer is a little complicated. First-order loops will always lock, because there is no low-pass attenuation of the error signal. Second-order loops may or may not lock, depending on the type of phase detector and the bandpass of the low-pass filter. In addition, the exclusive-OR (type I) phase detector has a limited *capture* range that depends on the filter time constant (this fact can be used to advantage, if you want a PLL that will lock to signals only within a certain frequency range).

The capture transient goes like this: As the (phase) error signal brings the VCO frequency closer to the reference frequency, the error-signal waveform varies more slowly, and vice versa. So the error signal is asymmetric, varying more slowly over that part of the cycle during which f_{VCO} is closer to f_{ref}. The net result is a nonzero average, i.e., a dc component that brings the PLL into lock. If you look carefully at the VCO control voltage during this *capture transient*, you'll see something like what is shown in Figure 9.76. That final

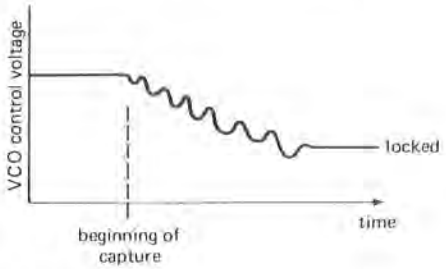


Figure 9.76

overshoot has an interesting cause. Even when the VCO *frequency* reaches its correct value (as indicated by correct VCO control voltage), the loop isn't necessarily in lock, since the *phase* may be wrong. So it may overshoot. Each capture transient is an individual – it looks a bit different each time!

□ *Capture and lock range*

For the exclusive-OR (type I) phase detector, the capture range is limited by the low-pass-filter time constant. This makes sense, because if you begin sufficiently far away in frequency, the error signal will be attenuated so much by the filter that the loop will never lock. It should be evident that a longer filter time constant results in narrower capture range, as does reduced loop gain. It turns out that the edge-triggered phase detector does not have this limitation. Both types have a lock range extending to the limits of the VCO, given the available control input voltage.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

□ 9.31 Some PLL applications

We have spoken already of the common use of phase-locked loops in frequency multiplication. The latter application, as in the preceding example, is so straightforward that there should be no hesitation about using these mysterious PLLs. In simple frequency-multiplication applications (e.g., the generation of higher clock frequencies in a digital system) there isn't even any problem of noise on the reference signal, and a first-order loop may suffice.

We would like to point out some other interesting applications, just to give an idea of the diversity of PLL uses.

□ *FM detection*

In frequency modulation, information is encoded onto a "carrier" signal by varying its frequency proportional to the information waveform. We will talk about FM and other modulation techniques in some detail in Chapter 13. There are two methods of recovering the modulating information using phase detectors or PLLs. The word *detection* is used to mean a technique of demodulation.

In the simplest method, a PLL is locked to the incoming signal. The voltage setting the VCO frequency is proportional to the input frequency and is therefore the desired modulating signal (Fig. 9.77). In such a system you would choose the filter bandwidth wide enough to pass the modulating signal, i.e., the response time of the PLL must be short compared with the time scale of variations in the signal being recovered. As you will see in Chapter 13, the signal applied to the PLL does not have to be the actual transmitted waveform; it can be an "intermediate frequency" (IF) generated in the receiving system by the process of *mixing*. A high degree of linearity in the VCO is desirable in this method of FM detection, for low audio distortion.

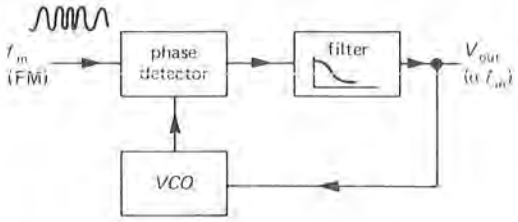


Figure 9.77. PLL FM discriminator.

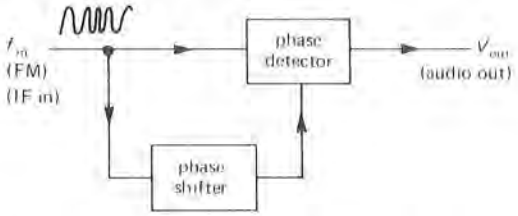


Figure 9.78. Quadrature FM detection.

The second method of FM detection involves a phase detector, although not in a phase-locked loop. Figure 9.78 shows the idea. Both the input signal and a phase-shifted version of the signal are applied to a phase detector, generating some output voltage. The phase-shifting network is diabolically arranged to have a phase shift varying linearly with frequency in the region of the input frequency (this is usually done with resonant LC networks), thus generating an output voltage with linear dependence on input frequency. That is the demodulated output. This method is called doubly balanced quadrature FM detection, and it is used in many IF amplifier/detector integrated circuits (e.g., the CA3189).

□ *AM detection*

Wanted: a technique to give an output signal proportional to the instantaneous *amplitude* of a high-frequency signal. The usual method involves rectification (Fig. 9.79). Figure 9.80 shows a fancy method ("homodyne detection") using PLLs. The PLL generates a square wave at the same frequency as the modulated carrier.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

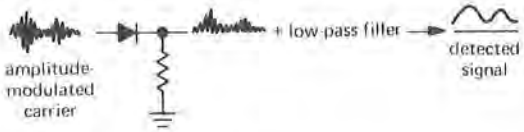


Figure 9.79. AM detection.

Multiplying the input signal by this square wave generates a full-wave-rectified signal that only needs some low-pass filtering to remove the remnants of the carrier frequency, leaving the modulation *envelope*. If you use the exclusive-OR type of phase detector in the PLL, the output is shifted 90° relative to the reference signal, so a 90° phase shift would have to be inserted in the signal path to the multiplier.

□ *Pulse synchronization and clean-signal regeneration*

In digital signal transmission, a string of bits containing the information is sent over a communications channel. The information may be intrinsically digital, or it may be digitized analog signals, as in "pulse-code modulation" (PCM, see Section 13.20). A closely related situation is the decoding of digital information from magnetic tape or disk. In both cases there may be noise or variations in pulse rate (e.g., from tape stretch), and it is desirable to have a clean clock signal at the same rate as the bits you are trying to read. PLLs work very nicely here. The low-pass filter would be chosen to eliminate the jitter and noise in the input synchronizing signal while following slow variations in tape speed, for example.

Another example of signal synchronization might be the circuit in Section 8.31, in which an accurate digitally generated "60Hz" signal (actually, anything from 50Hz to 70Hz) is used to generate a nice output sine wave. In that circuit we used a 6-pole Butterworth low-pass filter to convert the square wave to a sine. An attractive alternative would be to use a sine-wave VCO chip (e.g., the 8038) phase-locked to the precision 60Hz square wave. That would guarantee a constant sine-wave amplitude, permit a wide range of frequency variation, and allow you to eliminate jitter in the rate-multiplier output.

LC oscillator

Figure 9.81 shows an example of a PLL that uses an *LC* oscillator, with digital phase comparison at a lower frequency. We needed a stable and precise source of 14.4MHz, locked to the master observatory oscillator at 10MHz. A varactor (tuning diode, see Section 5.18) fine tunes the JFET *LC* oscillator, according to the type II phase detector output of the 'HC4046. Note that the varactor's tuning range of 18pF–30pF (5V to 1V, respectively) translates to a change in *LC* shunt capacitance of about 2pF (8.2pF to 10pF), for a tuning range of ±0.5% in oscillator frequency. We intentionally made the tuning range narrow to achieve good oscillator stability.

Both reference and output are digitally divided to 400kHz, where the phase detector works well. Note the use of an 'HC-style gate, biased at its logic threshold with a large feedback resistor, to convert the input sine wave to logic levels. Note also the simple emitter follower output stage (with

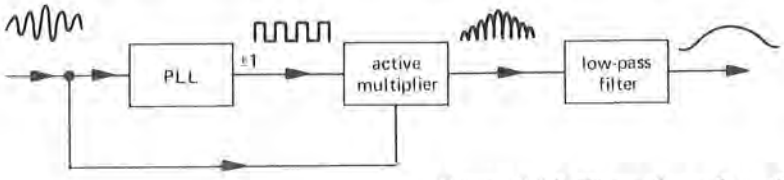


Figure 9.80. Homodyne detection.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

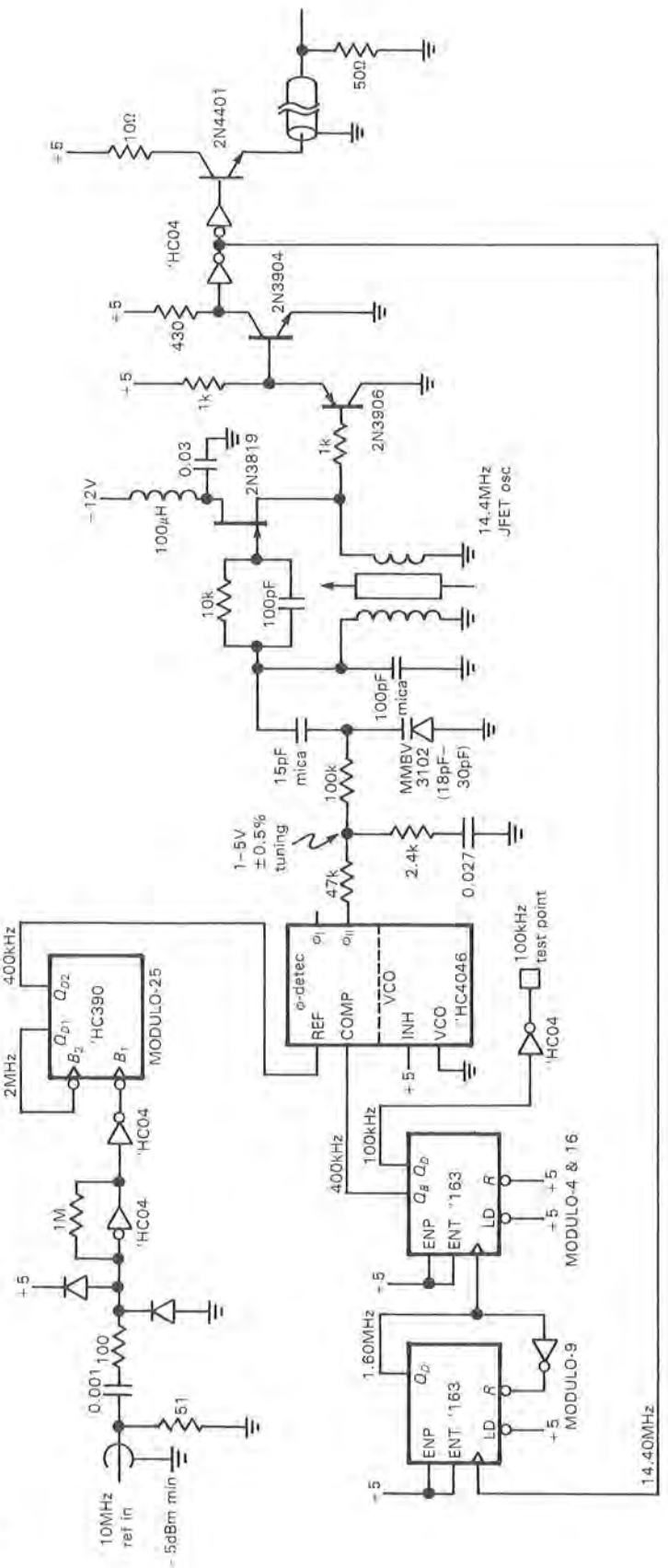


Figure 9.81. Varactor-tuned PLL.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

current limiting), used to drive 50 ohm terminated cable, as in Figure 9.42. To align this circuit, you tune the ferrite core in the oscillator until the phase detector filter output swings through its range; adjust the core for mid-range output.

Motorola makes a nice series of inexpensive "PLL Frequency Synthesizer" chips, the MC145145–59, that contain type II phase detectors and on-chip modulo-n dividers for both the signal and reference signals; both dividers are programmable, with 14-bit (or more) precision. Keep these in mind next time you need to synthesize weird frequencies.

PSEUDO-RANDOM BIT SEQUENCES AND NOISE GENERATION

□ 9.32 Digital noise generation

An interesting blend of digital and analog techniques is embodied in the subject of pseudo-random bit sequences (PRBS). It turns out to be remarkably easy to generate sequences of bits (or words) that have good randomness properties, i.e., a sequence that has the same sort of probability and correlation properties as an ideal coin-flipping machine. Since these sequences are generated by standard deterministic logic elements (shift registers, to be exact), the bit sequences generated are in fact predictable and repeatable, although any portion of such a sequence looks for all the world just like a random string of 0's and 1's. With just a few chips you can generate sequences that literally go on for centuries without repeating, making this a very accessible and attractive technique for the generation of digital bit sequences or analog noise waveforms. In fact, there is even an inexpensive "digital noise source" chip available in a mini-DIP package (National MM5437), and shift register noise generators are included in the innards of many sound effects chips.

□ *Analog noise*

Simple low-pass filtering of the output bit pattern of a PRBS generates band-limited white Gaussian noise, i.e., a noise voltage with a flat power spectrum up to some cutoff frequency (see Chapter 7 for more on noise). Alternatively, a weighted sum of the shift register contents (via a set of resistors) performs *digital filtering*, with the same result. Flat noise spectra out to several megahertz can easily be made this way. As you will see later, such digitally synthesized analog noise sources have many advantages over purely analog techniques such as noise diodes or resistors.

□ *Other applications*

Besides their obvious applications as analog or digital noise sources, pseudo-random bit sequences are useful in a number of applications that have nothing to do with noise. They can be used for encipherment of messages or data, since an identical PRBS generator at the receiving end provides the key. They are used extensively in error-detecting and error-correcting codes, since they allow the transcription of blocks of data in such a way that valid messages are separated by the greatest "Hamming distance" (measured by the number of bit errors). Their good autocorrelation properties make them ideal for radar ranging codes, in which the returned echo is compared (cross-correlated, to be exact) with the transmitted bit string. They can even be used as compact modulo-n dividers.

□ 9.33 Feedback shift register sequences

The most popular (and the simplest) PRBS generator is the feedback shift register (Fig. 9.82). A shift register of length m bits is clocked at some fixed rate, f_0. An exclusive-OR gate generates the

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

serial input signal from the exclusive-OR combination of the nth bit and the last (mth) bit of the shift register. Such a circuit goes through a set of states (defined by the set of bits in the register after each clock pulse), eventually repeating itself after K clock pulses; i.e., it is cyclic with period K.

The maximum number of conceivable states of an m-bit register is $K = 2^m$, i.e., the number of binary combinations of m bits. However, the state of all 0's would get "stuck" in this circuit, since the exclusive-OR would regenerate a 0 at the input. Thus, the maximum-length sequence you can possibly generate with this scheme is $2^m - 1$. It turns out that you can make such "maximal-length shift register sequences" if m and n are chosen correctly, and the resultant bit sequence is pseudo-random. (The criterion for maximal length is that the polynomial $1 + x^n + x^m$ be irreducible and prime over the Galois field.) As an example, consider the 4-bit feedback shift register in Figure 9.83. Beginning with the state 1111 (we could start anywhere except 0000), we can write down the states it goes through:

1111
0111
0011
0001
1000
0100
0010
1001
1100
0110
1011
0101
1010
1101
1110

We have written down the states as 4-bit numbers $Q_A Q_B Q_C Q_D$. There are 15 distinct states ($2^4 - 1$), after which it begins again; therefore it is a maximal-length register.

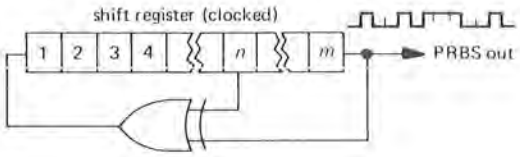


Figure 9.82. Pseudo-random bit sequence generator.

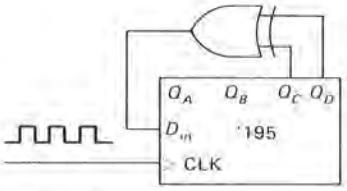


Figure 9.83

EXERCISE 9.6

Demonstrate that a 4-bit register with feedback taps at the second and fourth bits is not maximal length. How many distinct sequences are there? How many states within each sequence?

☐ *Feedback taps*

Maximal-length shift registers can be made with exclusive-OR feedback from more than two taps (in these cases you use several exclusive-OR gates in the standard parity-tree configuration, i.e., modulo-2 addition of several bits). In fact, for some values of m, a maximal-length register can only be made with more than two taps. Here is a listing of all values of m up to 40 for which maximal-length registers can be made with just two taps, i.e., feedback from the nth bit and the mth (last) bit, as previously. A value is given for n and for the cycle length K, in clock cycles. In some cases there is more than one possibility for n, and in every case the value $m - n$ can be used instead of n; thus, the earlier 4-bit example could have used taps at $n = 1$ and $m = 4$.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

m	*n*	*Length*
3	2	7
4	3	15
5	3	31
6	5	63
7	6	127
9	5	511
10	7	1023
11	9	2047
15	14	32767
17	14	131071
18	11	262143
20	17	1048575
21	19	2097151
22	21	4194303
23	18	8388607
25	22	33554431
28	25	268435455
29	27	536870911
31	28	2147483647
33	20	8589934591
35	33	34359738367
36	25	68719476735
39	35	549755813887

Since shift register lengths of multiples of 8 are common, you may want to use one of those lengths. In that case, more than two taps are necessary. Here are the magic numbers:

m	*Taps*	*Length*
8	4, 5, 6	255
16	4, 13, 15	65535
24	17, 22, 23	16777215

The MM5437 IC noise-generator chip uses a 23-bit register with a tap at stage 18. Its internal clock runs at about 160kHz, generating white noise output up to about 70kHz (3dB down) with a cycle time of about 1 minute; Figure 7.61 showed it in a "pink noise" generator circuit. With a 33-bit register clocked at 1MHz, the cycle time would be over 2 hours. A 100-bit register clocked at 10MHz would have a cycle time a million times longer than the age of the universe!

□ *Properties of maximal-length shift register sequences*

We generate a string of pseudo-random bits from one of these registers by clocking it and looking at successive output bits. The output can be taken from any position of the register; it is conventional to use the last (mth) bit as the output. Maximal-length shift register sequences have the following properties:

1. In one full cycle (K clock cycles), the number of 1's is one greater than the number of 0's. The extra 1 comes about because of the excluded state of all 0's. This says that heads and tails are equally likely (the extra 1 is totally insignificant for any reasonable-length register; a 17-bit register will produce 65,536 1's and 65,535 0's in one of its cycles).

2. In one full cycle (K clock cycles), half the runs of consecutive 1's have length 1, one-fourth the runs have length 2, one-eighth have length 3, etc. There are the same numbers of runs of 0's as of 1's, again with the exception of a missing 0. This says that the probability of heads and tails does not depend on the outcome of past flips, and therefore the chance of terminating a run of successive 1's or 0's on the next flip is $\frac{1}{2}$ (contrary to the man-in-the-street's understanding of the "law of averages").

3. If one full cycle (K clock cycles) of 1's and 0's is compared with the same sequence shifted cyclically by any number of bits n (where n is not 0 or a multiple of K), the number of disagreements will be one greater than the number of agreements. In fancy language, the autocorrelation function is a Kronecker delta at zero delay, and $-1/K$ everywhere else. This absence of "side lobes" in the autocorrelation function is what makes PRBSs so useful for radar ranging.

EXERCISE 9.7

Show that the 4-bit shift register sequence listed earlier (taps at $n = 3$, $m = 4$) satisfies these properties, considering the Q_A bit as the "output": 100010011010111.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

□ 9.34 Analog noise generation from maximal-length sequences

Advantages of digitally generated noise

As we remarked earlier, the digital output of a maximal-length feedback shift register can be converted to band-limited white noise with a low-pass filter whose cutoff frequency is well below the clock frequency of the register. Before getting into the details, we will point out some of the advantages of digitally generated analog noise. Among other things, it allows you to generate noise of known spectrum and amplitude, with adjustable bandwidth (via clock frequency adjustment), using reliable and easily maintained digital circuitry. There is none of the variability of diode noise generators, nor are there interference and pickup problems that plague the sensitive low-level analog circuitry used with diode or resistor noise generators. Finally, it generates repeatable "noise" and, when filtered with a weighted digital filter (more about this later), repeatable noise waveforms independent of clocking rate (output noise bandwidth).

□ 9.35 Power spectrum of shift register sequences

The output spectrum generated by maximal-length shift registers consists of noise extending from the repeat frequency of the entire sequence, f_{clock}/K, up to the clock frequency and beyond. It is flat within ±0.1dB up to 12% of the clock frequency (f_{clock}), dropping rather rapidly beyond its −3dB point of 44% f_{clock}. Thus a low-pass filter with a high-frequency cutoff of 5%–10% of the clock frequency will convert the unfiltered shift register output to a band-limited analog noise voltage. Even a simple RC filter will suffice, although it may be desirable to use active filters with sharp cutoff characteristics (see Chapter 5) if a precise frequency band of noise is needed.

To make these statements more precise, let's look at the shift register output and its power spectrum. It is usually desirable to eliminate the dc offset characteristic of digital logic levels, generating a bipolarity output with 1 corresponding to $+a$ volts and 0 corresponding to $-a$ volts (Fig. 9.84). This can be easily done with the

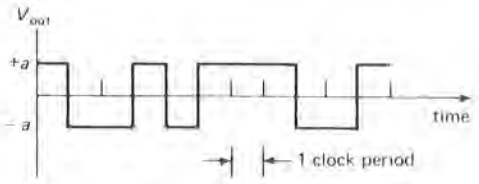


Figure 9.84

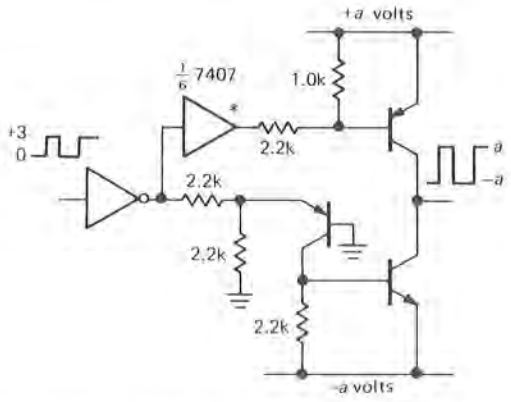


Figure 9.85. Precise bipolarity output stage with low Z_{out}.

sort of transistor push-pull stage shown in Figure 9.85. Alternatively, you can use MOS transistors, a circuit with clamping diodes to stable reference voltages, a fast op-amp with adjustable dc offset current into a summing junction, or a '4053 CMOS switch, running from ±a volts, with a pair of inputs tied to the supplies.

As we remarked earlier, the string of output bits has a single peak in its autocorrelation. If the output states represent +1 and −1, the digital autocorrelation (the sum of the product of corresponding bits,

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

when the bit string is compared with a shifted version of itself) is as shown in Figure 9.86.

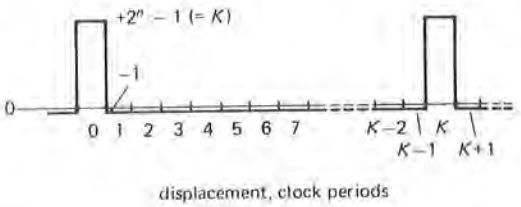


Figure 9.86. Full-cycle discrete autocorrelation for a maximal-length shift register sequence.

Don't confuse this with a *continuous* autocorrelation function, which we will consider later. This graph is defined only for shifts corresponding to a whole number of clock cycles. For all shifts that aren't zero or a multiple of the overall period K, the autocorrelation function has a constant -1 value (because there is an extra 1 in the sequence), negligible when compared with the zero-offset value of K. Likewise, if we consider the unfiltered shift register output as an *analog* signal (whose waveform happens to take on values of $+a$ and $-a$ volts only), the normalized autocorrelation becomes a continuous function, as shown in Figure 9.87. In other words, the waveform is totally uncorrelated with itself when shifted more than one clock period forward or backward.

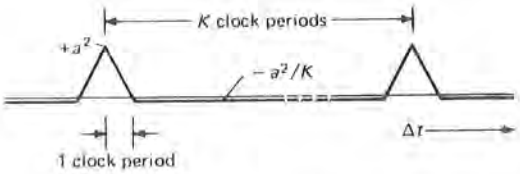


Figure 9.87. Full-cycle continuous autocorrelation for a maximal-length shift register sequence.

The power spectrum of the unfiltered digital output can be obtained form the autocorrelation by standard mathematical techniques. The result is a set of equally spaced series of spikes (delta functions), beginning at the frequency at which the whole sequence repeats, f_{clock}/K, and going up in frequency by equal intervals f_{clock}/K. The fact that the spectrum consists of a set of discrete spectral lines reflects the fact that the shift register sequence eventually (and periodically) repeats itself. Don't be alarmed by this funny spectrum; it will look continuous for any measurement or application that takes less time than the cycle time of the register. The envelope of the spectrum of the unfiltered output is shown in Figure 9.88. The envelope is proportional to the square of $(\sin x)/x$. Note the peculiar property that there is *no* noise power at the clock frequency or its harmonics.

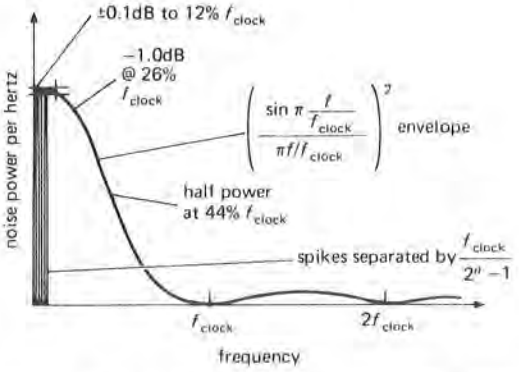


Figure 9.88. Power spectrum of unfiltered digital shift register output signal.

□ *Noise voltage*

Of course, for analog noise generation you use only a portion of the low-frequency end of the spectrum. It turns out to be easy to calculate the noise power per hertz in terms of the half-amplitude *(a)* and the clock frequency (f_{clock}). Expressed as an rms noise *voltage*, the answer is

$$V_{\text{rms}} = a\left(\frac{2}{f_{\text{clock}}}\right)^{1/2} \qquad \text{V/Hz}^{1/2}$$
$$(f \leq 0.2 f_{\text{clock}})$$

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

This is for the bottom end of the spectrum, the part you usually use (you can use the envelope function to find the power density elsewhere).

For example, suppose we run a maximal-length shift register at 1.0MHz and arrange it so that the output voltage swings between +10.0 and −10.0 volts. The output is passed through a simple RC low-pass filter with 3dB point at 1kHz (Fig. 9.89). We can calculate the rms noise voltage at the output exactly. We know from the preceding equation that the output from the level shifter has an rms noise voltage of 14.14mV per root hertz. From Section 7.21 we know that the noise bandwidth of the low-pass filter is $(\pi/2)(1.0\text{kHz})$, or 1.57kHz. So the output noise voltage is

$$V_{\text{rms}} = 0.01414(1570)^{1/2} = 560\,\text{mV}$$

with the spectrum of a single-section RC low-pass filter.

□ 9.36 Low-pass filtering

□ *Analog filtering*

The spectrum of useful noise from a pseudo-random-sequence generator extends from a low-frequency limit of the reciprocal repeat period (f_{clock}/K) up to a high-frequency limit of perhaps 20% of the clock frequency (at that frequency the noise power per hertz is down by 0.6dB). Simple low-pass filtering with RC sections, as illustrated in the earlier example, is adequate provided that its 3dB point is set far below the clock frequency (e.g., less than 1% of f_{clock}). In order to use the spectrum closer to the clock frequency, it is advisable to use a filter with sharper cutoff, e.g., a Butterworth or Chebyshev. In that case the flatness of the resultant spectrum depends on the filter characteristics, which should be measured, since component variations can produce ripples in the passband gain. Likewise, the filter's actual voltage gain should be measured if the precise value of noise voltage per root hertz is important.

□ *Digital filtering*

A disadvantage of analog filtering is the need to readjust the filter cutoff if the clock frequency is changed by large factors. In situations where that is desirable, an elegant solution is provided by digital filtering, in this case performed by taking an analog weighted sum of successive output bits (nonrecursive digital filtering). In this way the effective filter cutoff frequency changes to match changes in the clock frequency. In addition, digital filtering lets you go to extremely low cutoff frequencies (fractions of a hertz) where analog filtering becomes awkward.

In order to perform a weighted sum of successive output bits simultaneously, you can simply look at the various parallel outputs of successive shift register bits, using resistors of various values into an op-amp summing junction. For a low-pass filter the weights should be proportional to $(\sin x)/x$; note that some levels will have to be inverted, since the weights are of both signs. Since no capacitors are used in

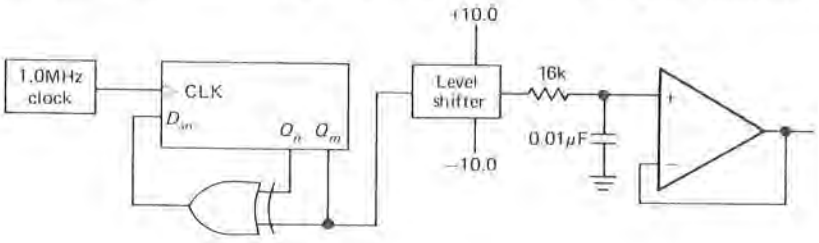


Figure 9.89. Simple pseudo-random noise source.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

this scheme, the output waveform consists of a set of discrete output voltages.

The approximation to Gaussian noise is improved by using a weighting function over many bits of the sequence. In addition, the analog output then becomes essentially a continuous waveform. For this reason it is desirable to use as many shift register stages as possible, adding additional shift register stages outside the exclusive-OR feedback if necessary. As before, pull-ups or MOS switches should be used to set stable digital voltage levels (CMOS logic is ideal for this application, since the outputs saturate cleanly at V_{DD} and ground).

The circuit in Figure 9.90 generates pseudo-random analog noise, with bandwidth selectable over an enormous range, using this technique. A 2.0MHz crystal oscillator drives a 14536 24-stage programmable divider, generating clock frequencies going from 1.0MHz down to 0.12Hz by factors going from 1.0MHz down to 0.12Hz by factors of 2. A 32-bit shift register is connected with feedback from stages 31 and 18, generating a maximal-length sequence with 2 billion states (at the maximum clock frequency the register completes one cycle in a half hour). In this case we have used a $(\sin x)/x$ weighted sum over 32 successive values of the sequence. U_1 and U_2 amplify the inverted and noninverted terms, respectively, driving differential amplifier U_3. The gains are chosen to generate a 1.0 volt rms output with no dc offset into a 50 ohm load impedance (2.0V rms open circuit). Note that this noise amplitude is independent of the clock rate, i.e., the total bandwidth. This digital filter has a cutoff at about $0.05 f_{\text{clock}}$, giving a white noise output spectrum extending from dc to 50kHz (maximum clock frequency) down to dc to 0.006Hz (minimum clock frequency), in 24 steps of bandwidth. The circuit also provides an unfiltered output waveform, going between +1.0 volt and −1.0 volt.

There are a few interesting points about this circuit. Note that an exclusive-NOR gate is used for feedback so that the register can be simply initialized by bringing it to the state of all zeros. This trick of inverting the serial input signal makes the excluded state the state of all 1's (rather than all 0's as with the usual exclusive-OR feedback), but it leaves all other properties unaffected.

A weighted sum of a finite number of bits cannot ever produce truly Gaussian noise, since the peak amplitude is limited. In this case it can be calculated that the peak output amplitude (into 50Ω) is ±4.34 volts, giving a "crest factor" of 4.34. That calculation is important, by the way, because you must keep the gain of U_1 through U_3 low enough to prevent clipping. Look carefully at the methods used to generate an output of zero dc offset from the CMOS levels of +6.0 volts average value (LOW=0V, HIGH=12.0V).

This method of digital low-pass filtering of maximal-length shift register sequences is used in many commercial noise generators.

□ 9.37 Wrap-up

A few comments about shift register sequences as analog noise sources: You might be tempted to conclude from the three properties of maximal-length shift registers listed earlier that the output is "too random," in the sense of having exactly the right number of runs of a given length, etc. A genuine random coin-flipping machine would not generate exactly one more head than tail, nor would the autocorrelation be absolutely flat for a finite sequence. To put it another way, if you used the 1's and 0's that emerge from the shift register to control a "random walk," moving forward one step for a 1 and back one step for a 0, you would wind up exactly one step away from your beginning point after the register had gone

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

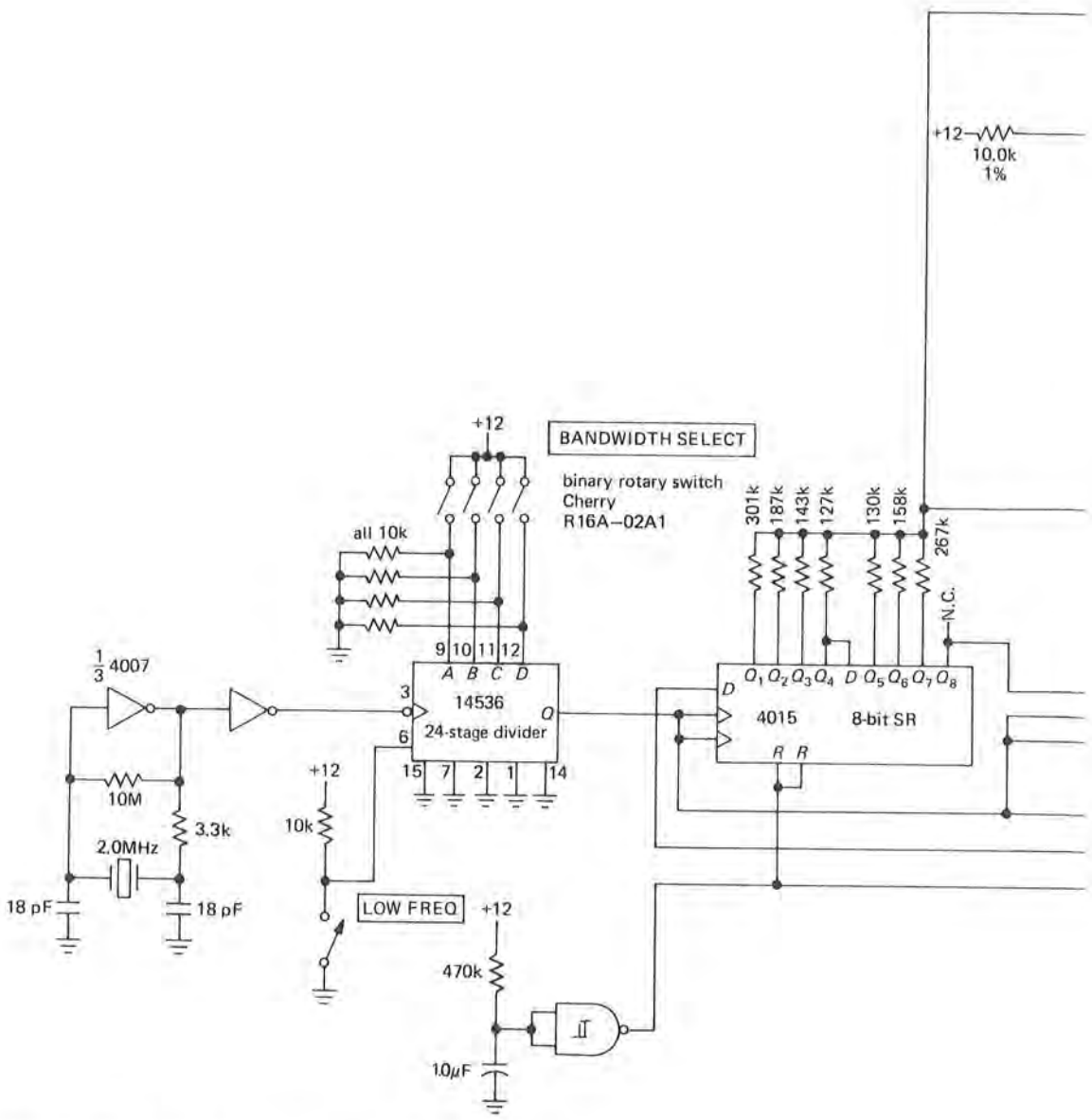


Figure 9.90. Wide-frequency-range laboratory noise source.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

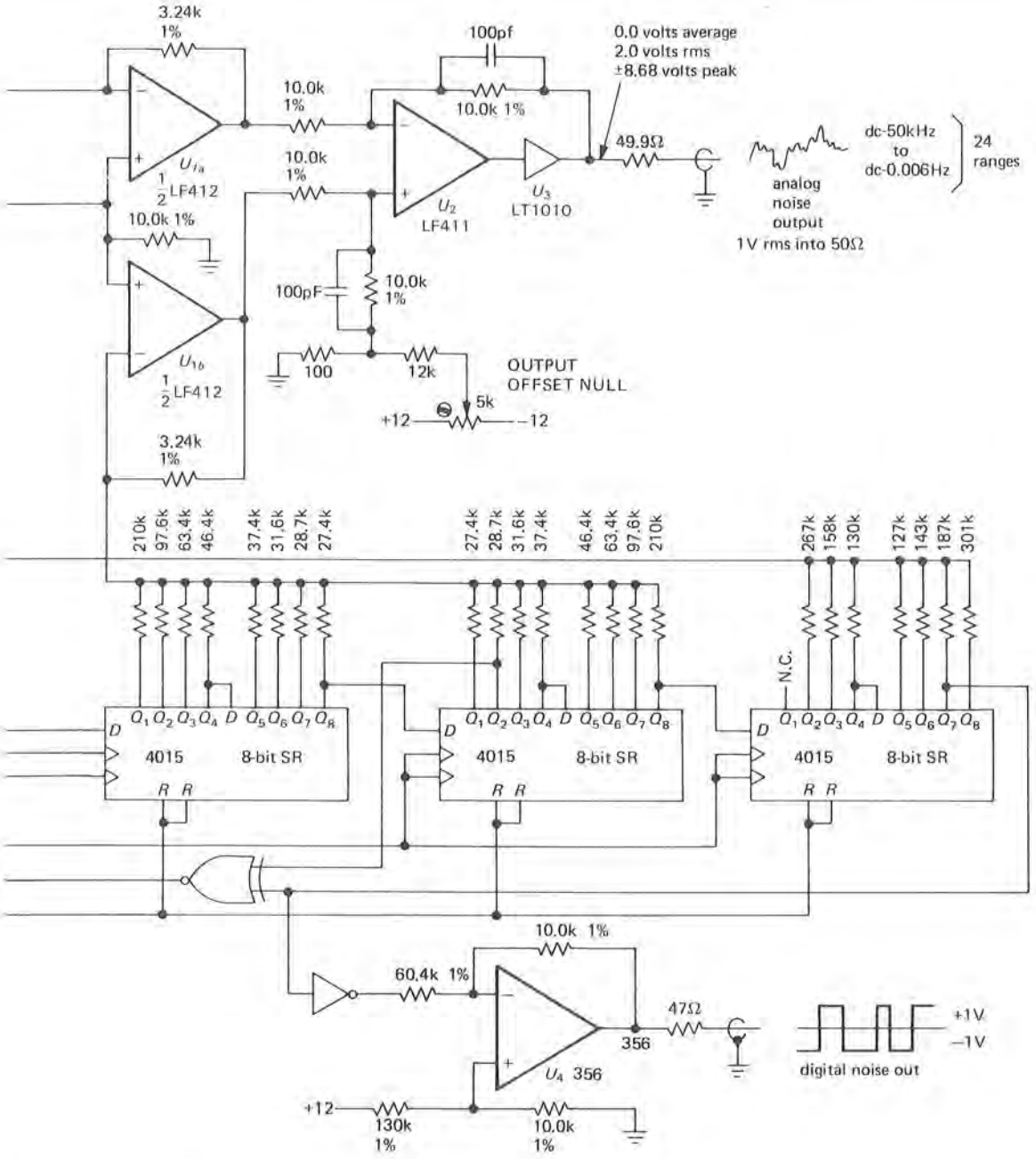


Figure 9.90 *cont.*

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

through its entire cycle, a result that is anything but "random"!

However, the shift register properties mentioned earlier are only true of the entire sequence of $2^n - 1$ bits, *taken as a whole*. If you use only a section of the entire bit sequence, the randomness properties closely approximate a random coin-flipper. To make an analogy, it is as if you were drawing red balls and blue balls at random from an urn initially containing K balls in all, half red and half blue. If you do this *without replacing them*, you would expect to find approximately random statistics at first. As the urn becomes depleted, the statistics are modified by the requirement that the total numbers of red and blue balls must come out the same.

You can get an idea how this goes by thinking again about the random walk. If we assume that the only nonrandom property of the shift sequence is the exact equality of 1's and 0's (ignoring the single excess 1), it can be shown that the random walk as described should reach an average distance from the starting point of

$$X = [r(K - r)/(K - 1)]^{1/2}$$

after r draws from a total population of $K/2$ 1's and $K/2$ 0's. (We are indebted to E. M. Purcell for this calculation.) Since in a completely random walk X equals the square root of r, the factor $(K-r)/(K-1)$ expresses the effect of finite urn contents. As long as $r \ll K$, the randomness of the walk is only slightly reduced from the completely random case (infinite urn contents), and the pseudo-random-sequence generator is indistinguishable from the real thing. We tested this with a few thousand PRBS-mediated random walks, each a few thousand steps in length, and found that the randomness was essentially perfect, as measured by this simple criterion.

Of course, the fact that PRBS generators pass this simple test does not guarantee that they would satisfy some of the more sophisticated tests of randomness, e.g., as measured by higher-order correlations. Such correlations also affect the properties of analog noise generated from such a sequence by filtering. Although the noise amplitude distribution is Gaussian, there may be higher-order amplitude correlations uncharacteristic of true random noise. Current thinking on this subject is that the use of many (preferably about $m/2$) feedback taps (using an exclusive-OR parity-tree operation to generate the serial input) generates "better" noise in this respect.

Noise generator builders should be aware of the 4557 CMOS variable-length shift register (1 to 64 stages); of course, you have to use it in combination with a parallel-output register (such as the '4015 or '164) in order to get at the n tap.

In Section 7.20 there is a discussion of noise, with an example of a "pink noise" generator using the MM5437 maximal-length shift register IC.

□ 9.38 Digital filters

The last example brought up the interesting topic of digital filtering, in that case the generation of a low-pass-filtered analog output signal by taking a weighted sum of 32 samples of pseudo-random bits, each corresponding to a voltage level of 0 volts or +12 volts. The "filter" accepted as its input a waveform that happened to have only two voltage levels. In general, the same thing can be done with an analog waveform as input, forming a weighted sum over its values (x_i) at equally spaced times

$$Y_i \sum_{k=-\infty}^{\infty} h_k x_{i-k}$$

The x_is are the discrete input signal samples, the h_ks are the weights, and the y_is are the output of the filter. In real life a digital filter will sum only over a finite set of input values, as for example in the noise generator circuit where we used 32 terms.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

Figure 9.91 shows schematically what is happening.

Note that such a filter can have the interesting property of being symmetrical in time, i.e., averaging past and future "history" to arrive at its present output. Of course, real analog filters can only look backward in time, corresponding to a digital filter with nonzero weighting factors h_k only for $k \geq 0$.

□ Symmetrical filter frequency response

For a symmetrical filter ($h_k = h_{-k}$) it can be shown that the frequency response is given by

$$H(f) = h_0 + 2\sum_{k=1}^{\infty} h_k \cos 2\pi k f t_s$$

where t_s is the time between samples. Thus the individual h_ks are recognizable (to those who know about such things!) as the Fourier series components of the desired frequency response curve, which explains why the weights in the noise-generator circuit shown earlier were chosen proportional to $(\sin x)/x$, the Fourier components of a "brick-wall" low-pass filter. For such a symmetrical filter the phase shift at any frequency is either zero or 180°.

□ Recursive filters

An interesting class of digital filters can be made if you allow the filter to use as its inputs the value of its own output, in addition to the value of the signal being filtered. You can think of the filter as having "feedback." The fancy name for such a filter is a *recursive* (or *infinite impulse response*, IIR) digital filter, as opposed to the nonrecursive (or *finite impulse response*, FIR) filters just discussed. For example, you could form the outputs y_i according to

$$y_i = Ay_{i-1} + (1 - A)x_i$$

This happens to give you a low-pass response, equivalent to that of a simple RC low-pass filter, according to

$$A = e^{-t_s/RC}$$

where t_s is the time between successive samples x_i of the input waveform. Of course, the situation is not *identical* with an analog low-pass filter operating on an analog waveform because of the discrete nature of the sampled waveform.

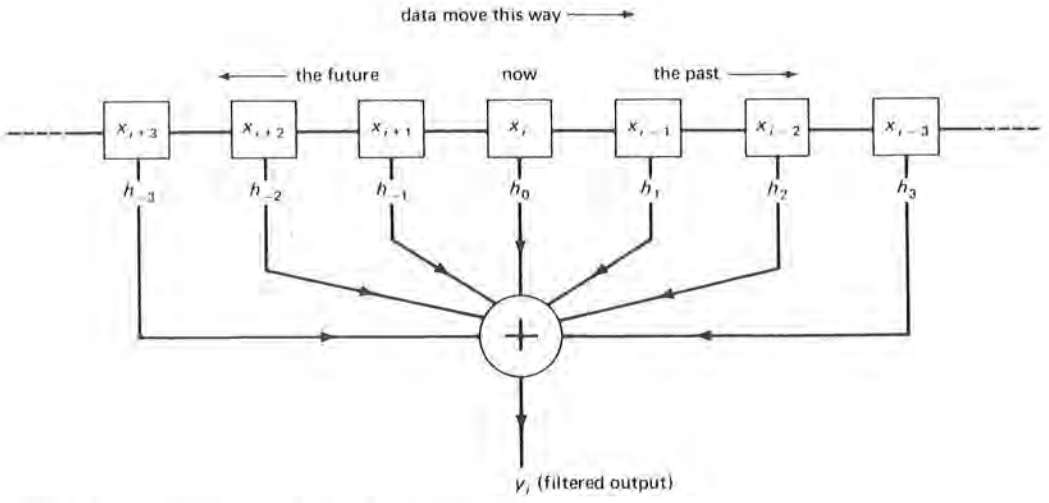


Figure 9.91. Nonrecursive digital filter.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

☐ Low-pass filter example

As a numerical example, suppose you want to filter a set of numbers representing a signal, with a low-pass 3dB point at $f_{3dB} = 1/20t_s$. Thus the time constant equals the time for 20 successive samples. Then $A = 0.95123$, and so the output is given by

$$y_i = 0.95123y_{i-1} + 0.04877x_i$$

The approximation to a real low-pass filter becomes better as the time constant becomes long compared with the time between samples, t_s.

You would probably use a filter like this to process data that are already in the form of discrete samples, e.g., an array of data in a computer. In that case the recursive filter becomes a trivial arithmetic pass once through the data. Here is what the low-pass filter would look like in a FORTRAN program:

```
      A = EXP(-TS/TC)
      B = 1. - A
      DO 10 I = 2,N
10    X(I) = A * X(I - 1) + B * X(I)
```

where X is the array of data, TS is the time between samples of the data (i.e., TS $= 1/f_s$), and TC is the desired filter time constant. Ideally, TC $\gg$ TS. This little program does the filtering *in place*, i.e., it replaces the original data by the filtered version. You could, of course, put the filtered data into a separate array.

☐ Low-pass commutating filter

This same filter can be built with hardware, using the circuit shown in Figure 9.92. The FET switches S_1 and S_2 are toggled at some clock rate f_s, repeatedly charging C_1 to the input voltage, then transferring its charge to C_2. If C_2 has voltage V_2, and C_1 charges up to the input level V_1, then when C_1 is connected to C_2, the new voltage will be

$$V = \frac{C_1V_1 + C_2V_2}{C_1 + C_2}$$

i.e., it is identical with the preceding low-pass recursive filter, with

$$y_i = \frac{C_2}{C_1 + C_2}y_{i-1} + \frac{C_1}{C_1 + C_2}x_i$$

Equating these coefficients to the value of A, given earlier, gives us

$$f_{3dB} = \frac{1}{2\pi}f_s \log_e\left(\frac{C_1 + C_2}{C_2}\right)$$

EXERCISE 9.8
Prove that this result is correct.

This filter is perfectly practicable, and it offers the nice feature of electronic tuning via the clock rate f_s. In practice, you would use CMOS switches, and C_1 would probably be much larger than C_2. Consequently, the switch-driving waveform should be unsymmetrical, spending most of its time closing S_1.

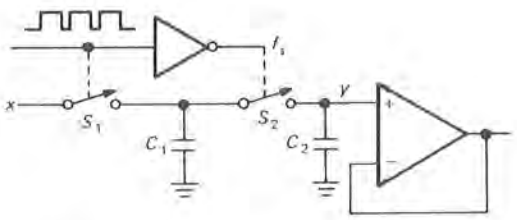


Figure 9.92. Recursive switched-capacitor filter.

The preceding circuit is a simple example of a *commutating filter*, which includes filters made from arrays of switched capacitors. They have periodic frequency response properties that make them particularly suitable for "comb" and notch filters.

It is possible to synthesize discrete approximations to all the classic filters (Butterworth, Chebyshev, etc.) in high-pass, low-pass, bandpass, and band-reject forms, either symmetrical in time or with the genuine "lagging" time response. Such filters are extremely useful when processing digitally quantized data, which is clearly the way of the future.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

Inexpensive switched-capacitor filter ICs are now widely available. For example, the MF4 from National is a 4-pole Butterworth low-pass filter in a mini-DIP; it requires no external components and runs from a single +5 volt to +14 volt supply. The filter's cutoff frequency (0.1Hz min, 20kHz max) is set by the externally supplied clock, $f_{clk} = 100 f_{3dB}$. The MF5 and MF10 "universal switched capacitor filters" work somewhat differently. A few external resistors set the type of filter (high-pass, low-pass, bandpass, notch) and filter characteristic (Butterworth, Chebyshev, etc.), with the clock rate setting the cutoff frequency as before. Some other manufacturers of switched-capacitor filters are American Microsystems (AMI), Linear Technology (LTC), and Reticon. LTC, as usual, has found some clever tricks to improve performance. Their LTC1062 (or MAX280) is similar to the MF4, but with 5 poles and with *zero* dc error! They accomplish the latter by putting the filter outside the dc path (Fig. 9.94). The flexible MAX260 series allows microprocessor control of important filter parameters.

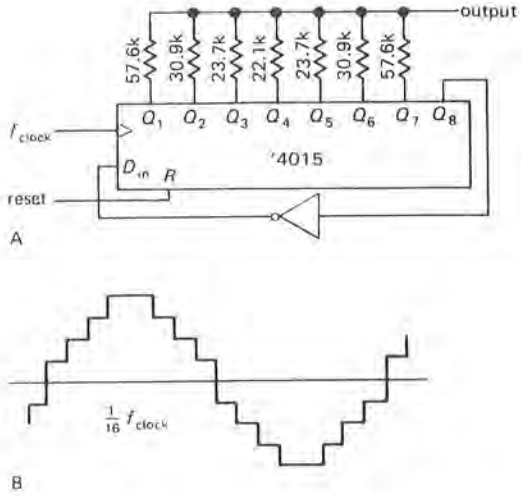


Figure 9.93. Digital sine-wave generation.

In general, these switched-capacitor filters operate only to the top of the audio range. They also tend to suffer from significant clock feedthrough to the output, typically in the range of 10–25mV. This limits their dynamic range in applications where the clock frequency is within the bandpass (e.g., high-pass filters). They can have significant amounts of noise, sometimes limiting their dynamic range to 80dB or less (compared with 140dB or more for a good op-amp). On the positive side, switched-capacitor filters are easy to use and let you tune your filter response effortlessly (via f_{clk}). They are finding widespread use in modems (data communication via audio telephone circuits) and other communications applications. See Section 5.11.

□ *Digital sine-wave generation*

An interesting technique related to nonrecursive digital filtering is the synthesis of sine waves by taking weighted sums from the outputs of a Johnson counter (or "twisted-ring" or "walking-ring" counter). The circuit in Figure 9.93 shows the way. The 4015 is an 8-stage parallel-output shift register. By driving the input with the complement of the last stage, you get a Johnson counter, which goes through 16 states ($2n$ states, in general, for an n-stage shift register). Beginning with the state of all 0's, 1's begin marching in from the left until the register contains all 1's, at which point 0's march in again, and so on. The weighting shown generates an 8-level approximation to a sine wave, as shown, with a frequency 1/16th that of the clock, and with the first nonzero distortion term (assuming perfect resistor values) being the 15th harmonic, which is down by 24dB.

SELF-EXPLANATORY CIRCUITS

9.39 Circuit ideas

Figure 9.95 shows a few examples of interfacing between logic and linear signals.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

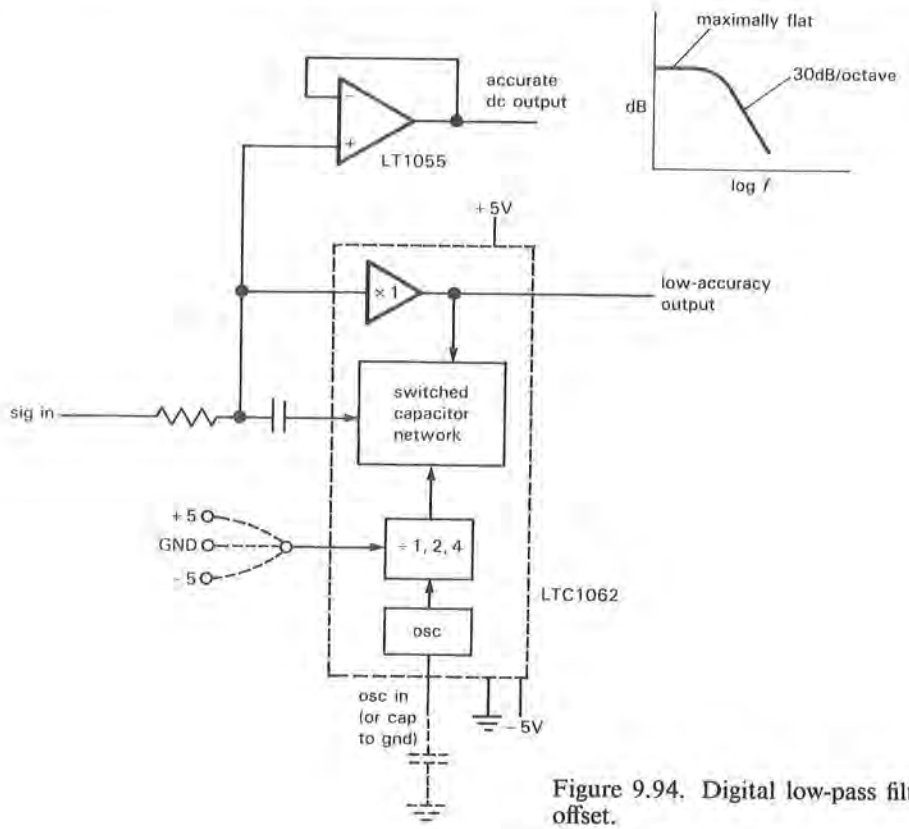


Figure 9.94. Digital low-pass filter with low dc offset.

9.40 Bad circuits

The circuits in Figure 9.96 illustrate some basic interfacing blunders; in each case try to figure out what is wrong and how to fix it.

ADDITIONAL EXERCISES

1. Design a circuit to indicate if the logic power (+5V) has failed momentarily. It should have a push button to RESET it and a LED lamp that indicates CONTINUOUS POWER; make it operate from the +5 volt logic supply.

2. Why can't two n-bit DACs be used to make a $2n$-bit DAC by just summing their outputs proportionally ($OUT_1 + OUT_2/2^n$)?

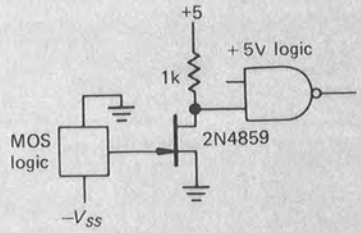
3. Verify that the peak output of the pseudo-random-noise generator (Fig. 9.90) is ±8.68 volts.

4. An experiment is being controlled by a programmable calculator interfaced to various stimulus and measurement devices. The calculator increments a variable under its control (e.g., the wavelength of light coming from a monochromator) and processes the corresponding measurement (e.g., the amount of transmitted light, corrected for the known sensitivity curve of the detector). The result is a set of x, y pairs. Your job is to design a circuit so that they can be plotted on analog x, y plotter.

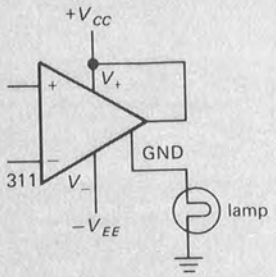
The calculator outputs each x, y pair as two 3-digit BCD characters. To reduce the number of connections necessary, the numbers are presented one digit at a time ("bit parallel, character serial"),

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

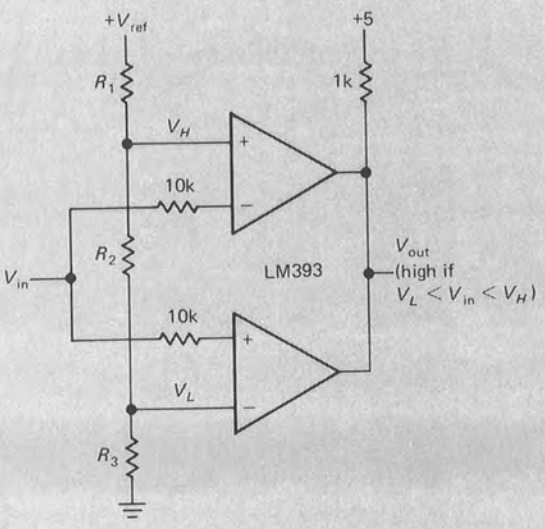
Circuit ideas



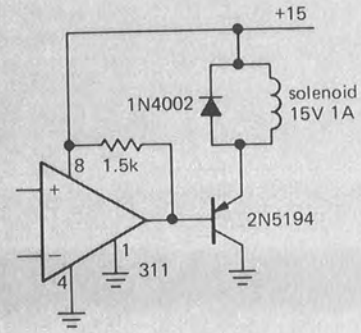
A. negative logic to TTL shifter



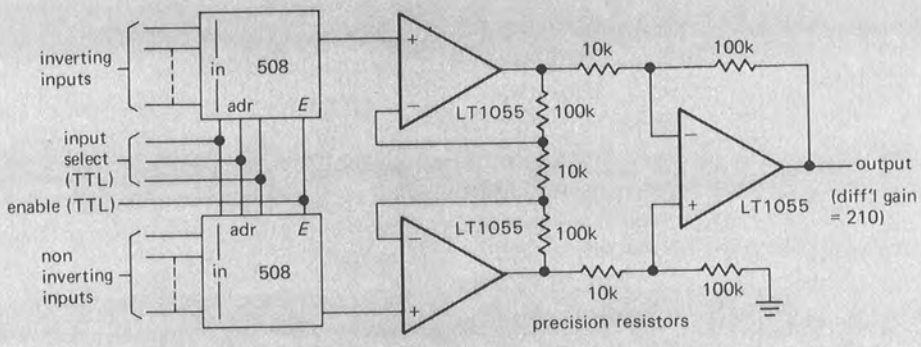
B. driving ground-returned load



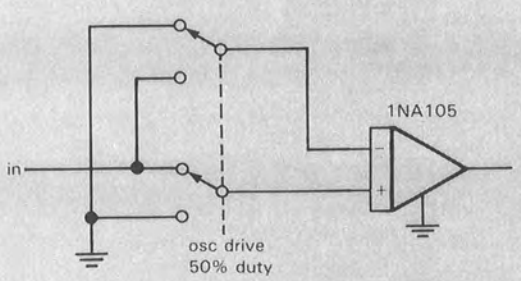
C. window discriminator



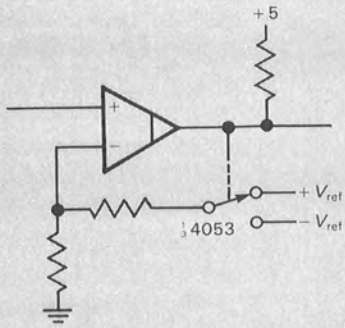
D. solenoid driver



E. 8-channel multiplexer with differential input



F. phase detector with 60dB osc rejection

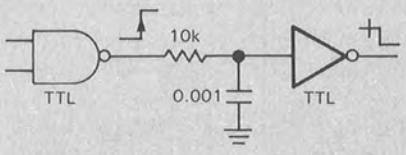


G. precision Schmitt trigger

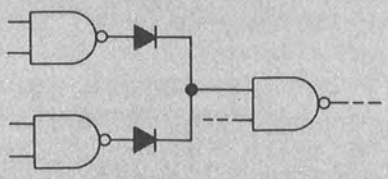
Figure 9.95

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

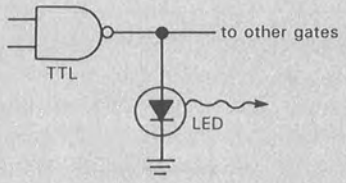
Bad circuits



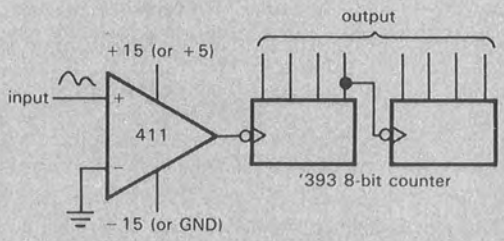
A. delayed edge generator



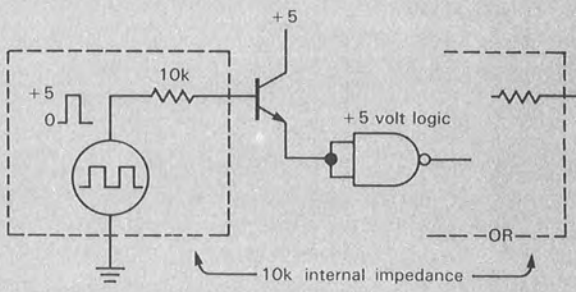
E. wired-OR from active pull-up gates



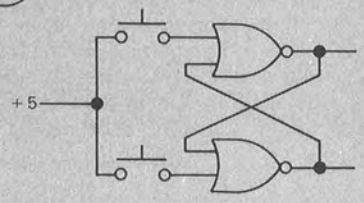
B. logic-state indicator



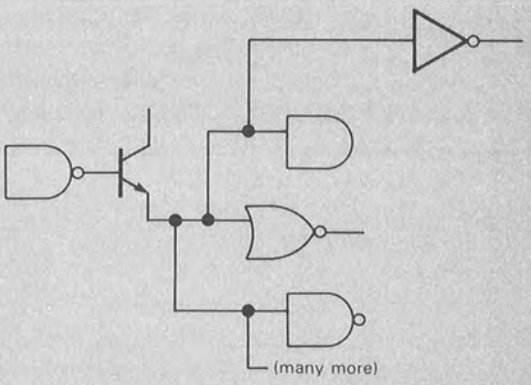
F. zero-crossing counter



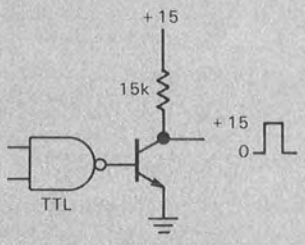
C. high-Z-to-logic interface (2 bad circuits)



G. SR flip-flop



D. increasing logic fanout with a follower



H. TTL-to-high-level interface

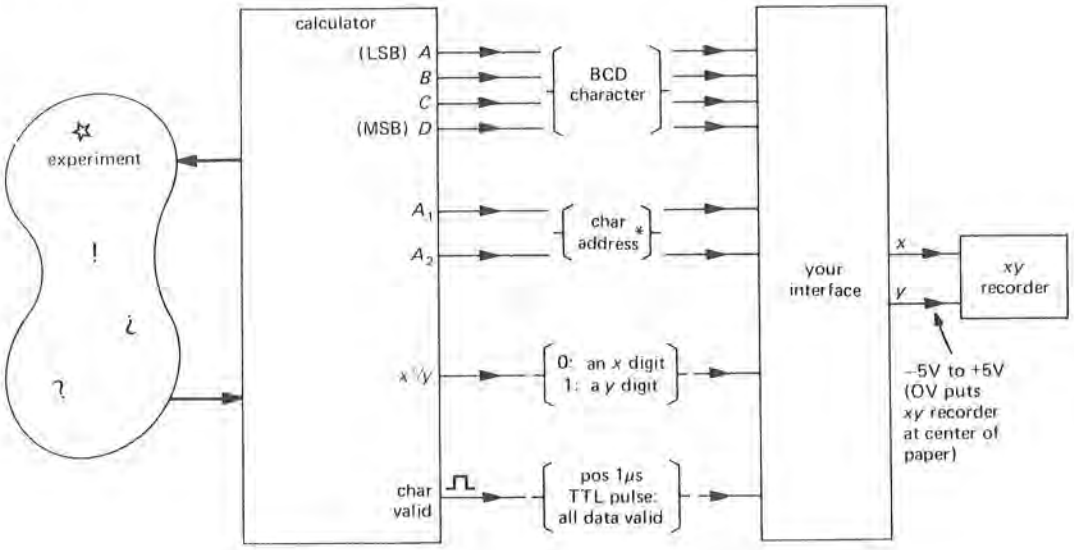
Figure 9.96

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

along with an address (2 bits). A CHARACTER VALID pulse signifies that the data and address are valid and can be latched, for example. An x'/y level tells whether the character being presented belongs to the x or y number. Figure 9.97 presents a summary.

The data are sent in the order x_n (LSD) ... x_n (MSD), y_n (LSD) ... y_n (MSD), so you know you've got one complete x, y pair after receiving the MSD of a y value ($A_1 = 0$, $A_2 = 1$, $x'/y = 1$). At *that* point you should update the digits seen by your D/A converters (don't update them one at a time).

In your circuit you needn't use particular device numbers; just label them generically, e.g., a type D flip-flop, or a 1-of-10 decoder. Be sure to indicate where inputs or outputs are inverted (by showing small circles). Assume that you have some D/A converters that accept 3-digit logic-level BCD inputs, and whose outputs are currents, zero to 1mA corresponding to inputs of 000 to 999. Since the x, y plotter has 10 volt full-scale sensitivity, you'll have to convert the current to a voltage. As an additional obstacle, so that you can exercise your ingenuity, assume that the D/A converters have an output compliance of only 1 volt.



* character address decoding:

A_1	A_2	character
0	0	LSD
1	0	↓
0	1	MSD

Figure 9.97

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

MICROCOMPUTERS

CHAPTER 10

MINICOMPUTERS, MICROCOMPUTERS, AND MICROPROCESSORS

The availability of inexpensive ($1k) small computers has made it attractive to control experiments and processes, collect data, and perform computation directly under the control of a computer. Small computers are commonly used in laboratory and industrial settings, and knowledge of their capabilities, program languages, and interfacing requirements is an essential part of electronics know-how.

The microcomputer evolved from the earlier *minicomputer*, a small machine whose central processing unit (CPU) was constructed from SSI and MSI ICs, usually occupying one or more large printed-circuit boards. As large-scale integration improved, it became possible to put minicomputer CPU performance into a single LSI chip; thus a *microcomputer* is a computer whose CPU is constructed from just a few (often only one) LSI microcircuits; the CPU chip (or chip set) constitutes a *microprocessor*. For example, DEC's popular PDP-11 minicomputers (CPU on several interconnected boards) were succeeded by a family of similarly named computers whose CPUs were built from a few LSI chips in place of many SSI/MSI chips; at about the same time, Motorola introduced a high-performance microprocessor (the 68000) that has many similarities to the PDP-11 and was obviously influenced by it.

Most modern small computers are in fact microcomputers, relying on the impressive performance of the present generation of microprocessors. The phrase "superminicomputer" has recently surfaced and seems to signify a class of machines that achieve higher performance, in some cases rivaling the large and expensive "mainframe" computers. In some cases the distinction refers more to physical size or number of peripherals than to the scale of integration used in the construction of the CPU.

A more important distinction separates microcomputers from *microcontrollers*, a term used to describe the use of a microprocessor, along with a small amount of memory and other support chips, for dedicated control of a process or instrument. In this role a microprocessor plus a few

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

assorted chips and some ROM (read-only memory) can flexibly replace a complicated logic circuit of gates, flip-flops, and analog/digital conversion functions and should be considered whenever embarking on a large design project. There are microprocessors optimized for this kind of application, generally characterized by on-chip timers, ports, and other functions that usually require extra ICs, at the expense of the computational power and large address space that characterizes microprocessors intended for microcomputer-based computational tasks.

In this chapter we will describe microcomputer architecture, programming, and interfacing, with some examples of useful and simple interfacing of peripherals to the IBM PC/XT (here we refer to the original PC bus and its derivatives such as the PC/AT and compatibles, and the low end of the PS/2 line). Most of the ideas introduced in this chapter will carry over to the next chapter, where we will get into a detailed discussion of the selection and construction of microprocessor-based circuits and systems; for those examples we will use the 68008 microprocessor, a member of the Motorola 68000 family that, together with the Intel 8086 family, dominates small computers. Generally speaking, with microcomputers the design of the computer itself, including the integration of memory, disks, and I/O control, as well as system programming and utility program development, is taken care of by the manufacturer (and suppliers of compatible hardware and software). The user need only worry about special-purpose interfaces and the job of user programming. By contrast, in a dedicated microprocessor system, the choices of memory types, system interconnection, and programming generally have to be made by the designer. Microcomputer manufacturers are generally committed to providing system and utility software as part of a complete computing system (often including peripherals), whereas the microprocessor manufacturers (semiconductor companies) generally see the design and marketing of microprocessor and support chips as their central tasks. In this chapter, then, we will describe computer architecture and programming and will concentrate on the details of internal communication and interfacing.

10.01 Computer architecture

Figure 10.1 summarizes the organization typical of most computers. Let's take it from left to right:

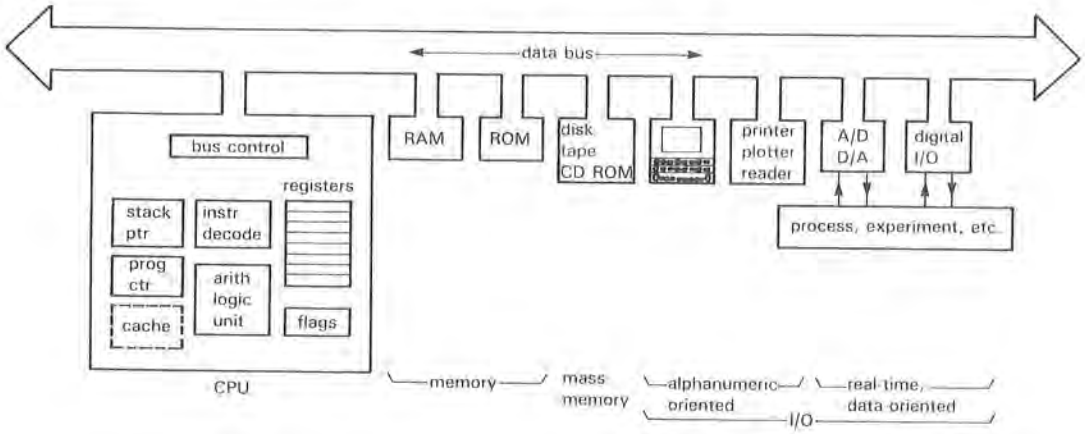


Figure 10.1. Block diagram of a computer.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

CPU

The central processing unit, or CPU, is the heart of the machine. Computers do their computation in the CPU on chunks of data organized as computer *words*. Word size can range from 4 bits to 32 bits or more, with a 16-bit word size being the most popular in current microcomputers. A *byte* is 8 bits (half a byte, or 4 bits, is sometimes called a "nybble"). A portion of the CPU called the *instruction decoder* interprets the successive instructions (fetched from memory), figuring out what should be done in each case. The CPU has an *arithmetic unit*, which can perform the instructed operations, such as add, complement, compare, shift, move, etc., on quantities contained in *registers* (and sometimes in *memory*). The *program counter* keeps track of the current location in the executing program. It normally increments after each instruction, but it can take on a new value after a "jump" or "branch" instruction. The *bus control circuitry* handles communication with memory and I/O. Most computers also have a *stack pointer register* (more on that later) and a few *flags* (carry, zero, sign) that get tested for conditional branching. Many high-performance processors also include *cache* memory, which holds values recently fetched from memory for quicker access.

There has been a lot of activity in the experimental field of "parallel processing," in which you interconnect many CPUs to get tremendous computational power. With time this trend may become dominant in high-performance processing. For the time being, however, our single-CPU machine, executing instructions serially, represents the standard microcomputer architecture.

Memory

All computers have some fast random-access memory, called RAM (it used to be called "core," because tiny magnetic cores held the data, one bit per core). In a large microcomputer this may include 10 megabytes or more, although a megabyte is more typical, and as little as 16K may be used in a microcontroller. (When used to describe memory sizes, K doesn't mean 1000, but rather 1024, or 2^{10}; thus, 16K bytes is actually 16,384 bytes. We employ the lower-case symbol k to mean 1000.) This memory can typically be read and written in about 100ns. RAM is almost always *volatile*, which means that its information evaporates when power is removed (maybe it should be called "forgettory"!). All computers therefore include some nonvolatile memory, usually ROM (read-only memory), to "bootstrap" the computer, i.e., get it started from a state of total amnesia when power is first turned on. Additional ROM is often programmed with system routines, graphics routines, and other programs that you want to be there all the time.

To get or store information in memory, the CPU "addresses" the desired word. Most computers address memory by bytes, beginning at byte 0 and going sequentially through to the last byte in memory. Since most computer words are several bytes long, you are usually storing or fetching a group of bytes at a time; this is usually expedited by having a data bus that is several bytes wide. For example, microcomputers that use the 80386 or 68020 use a bus 32 bits (4 bytes) wide, so that a 32-bit word can be moved to/from memory in one memory fetch. (There are control signals to specify how many contiguous bytes are being moved, since even with a large bus you may want only 1 or 2 bytes.)

In a computer with lots of memory, it takes three or four bytes to specify an arbitrary memory address anywhere in the machine. Since most memory references in an actual program are usually "nearby," all computers provide for simplified addressing modes: "Relative" addressing specifies an address by its distance from the present instruction; "indirect" addressing uses the

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

contents of a CPU register to point to a location in memory; "paged" addressing uses a shortened address to refer to a memory location within a small area (a page); "direct" or "absolute" addressing uses the next few bytes in memory to specify an address. A modern CPU embellishes this short list with additional "indexed," "autoincrementing," and other useful addressing modes, which we'll learn about in the next chapter.

Both programs and data are kept in memory during program execution. The CPU fetches instructions from memory, figures out what they mean, and does the appropriate things, often involving data stored somewhere else in memory. General-purpose computers usually store programs and data in the same memory, and in fact the computer doesn't even know one from the other. Amusing things start to happen if a program goes awry and you "execute" data!

Since computer programs spend most of their time looping through a relatively short sequence of instructions, you can enhance performance by providing a small, but fast, *cache* memory, in which you routinely store copies of the most recently used memory locations. A cached CPU checks its local cache first, before fetching from (slower) main memory; when looping through familiar territory, you often achieve a cache "hit" rate of 95% or better, dramatically improving execution speed.

Mass memory

Computers intended for program development or computation, as opposed to dedicated control processors, usually have one or more mass-storage devices. "Hard" disks (also called "Winchester") and "floppy" disks ("diskettes") are the usual ones, with storage capacities going from a few hundred kilobytes to a few megabytes (floppy disks), and from a few tens of megabytes to a few hundred megabytes (hard disks). Most well-endowed computers also have a tape drive or two, ranging from a simple cartridge-tape "streaming" drive to a full-fledged 9-track, half-inch, large-reel tape (the kind that are always spinning in the background in science fiction movies). A newer technology uses 8mm videotape (the kind that lives in those little hand-held video cameras) to store a gigabyte on a small tape cartridge. And the latest in mass storage is the "CD ROM," which uses the same optical disk technology as audio CDs (compact discs); they store 600 megabytes on one side of a 5 inch plastic disk, with much faster access than any tape medium. Unlike audio CDs, there are CD ROM drives that let you write as well as read, by laser-burning pits in a blank CD; they're called "WORM," for "write once, read many." Furthermore, fully erasable read/write magneto-optic disk memories are also available.

Compared with RAM, mass-storage media are generally slow, magnetic tape being the slowest, with access times of many seconds, and hard disks being the fastest (and most expensive), with average access times of tens of milliseconds. With all mass-storage devices, data transfer is rapid (10K to 100K bytes per second or more) once the data has been located. You generally keep programs, data files, plot files, etc., on some sort of mass-storage device and bring these into RAM only when doing computation. Many users can simultaneously fit their programs on one disk; a moderate-size optical disk can hold the contents of the *Encyclopaedia Britannica* several times over.

If your computer has lots of RAM, a nice way to speed up computer operations that make heavy use of disk is to form a "RAM disk" by loading all the relevant disk files into RAM when you start. Thus you might put a text editor, compiler, and linker/loader into RAM; then you can switch back and forth without waiting for the disk. Be careful, though; because none

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

of your work is being saved on nonvolatile disk, you lose all your work if the computer crashes.

Alphanumeric and graphic I/O

It is nice to have a powerful computer, capable of millions of smart computations per second, but it doesn't do you any good if it keeps all its results to itself. Peripherals such as a keyboard and screen (the combination is a "terminal"), "mouse," printer, etc., let man and machine communicate, and these are essential in any "friendly" computer system. These peripherals are mostly oriented toward programming, word processing, spreadsheets, and graphics; you use them when writing programs, debugging, listing, writing and printing documents, manipulating quantities and objects, and playing flight simulator. These sorts of peripherals, together with suitable interfaces, are available from many sources, including the microcomputer manufacturer.

Real-time I/O

For experiment or process control and data logging, or for exotic applications such as speech or music synthesis, you need A/D and D/A devices that can communicate with the computer in "real time," i.e., while things are happening. The possibilities are almost endless here, although a general-purpose set of multiplexed A/D converters, a few fast D/As, and some digital "ports" (serial or parallel) for exchange of digital data will permit many interesting applications. Such general-purpose peripherals are commercially available for most popular computer buses. If you want something fancier, such as improved performance (higher speed, more channels) or special-purpose functions (tone generation, frequency synthesis, time-interval generation, etc.), you may have to build it yourself. This is where a knowledge of bus interfacing and programming techniques is essential, though it's helpful in any case.

Network interface

Powerful desktop computers become even more powerful when they can exchange files with other computers. One way to do this is to "log on" to a remote computer via telephone lines, then use the features of the remote computer that you need. That might include access to a large data base or special programs, a powerful supercomputer, computer "mail," or a colleague's text or data file. For these purposes you need a "modem" (modulator/demodulator), which either plugs directly into your computer's bus or hooks onto a serial data port. We'll have more to say about this later.

Another way to extend the scope of your machine is to use a local area network (LAN) to link a group of computers together. An example is Ethernet, which provides communication at rates up to 10Mb/s among linked machines, via a single coaxial cable. A LAN lets you access files on anyone's machine; in fact, with a good LAN you would probably pool your resources, sharing a fast large disk, high-priced plotters and printers, etc. Each "workstation" would then have only limited mass storage, but enough computational and display capability for the work you want to do with it. Such a setup is ideal for a publishing house or newspaper, for example, where different people work on manuscripts as they are readied for publication. You can get Ethernet (and other LAN) interfaces for most microcomputers.

Data bus

For communication within the computer between the CPU and memory or peripherals, all computers use a *bus*, a set of shared lines for exchange of digital words. (Many buses also allow communication

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

between peripherals, though this capability is used less often.) The use of a shared bus vastly simplifies interconnections, since otherwise you would need multiwire cables connecting every pair of communicating devices. With a little care in bus design and implementation, everything works fine.

The bus contains a set of DATA lines (generally the same number as bits in a word – 8 for microcontrollers and low-performance PCs, 16 or 32 for more sophisticated microcomputers), some ADDRESS lines for determining who should "talk" or "listen" on the line, and a bunch of CONTROL lines that specify what action is going on [data going to or from the CPU, interrupt handling, DMA (direct memory access) transfers, etc.]. All the DATA lines, as well as a number of others, are *bidirectional* – they're driven by three-state devices, or in some cases by open-collector gates with resistor pullups somewhere (usually at the end of the bus, where they also serve as terminators to minimize reflections, see Section 13.09); pullups may be necessary with three-state drivers also, if the bus is physically long.

Three-state or open-collector devices are used so that devices connected to the bus can disable their bus drivers, since in normal operation only one device is asserting data onto the bus at any time. Each computer has a well-defined protocol for determining who asserts data, and when. If it didn't, total chaos would result, with everyone shouting at once (so to speak). (Computer people can't resist personalizing their machines, peripherals, etc. Engineers are even worse, with flip-flops and even gates coming to life. Naturally, we follow the trend.)

There is one interesting distinction in computer buses. They can be either *synchronous* or *asynchronous*, with examples of each in currently popular microcomputers. You will see what this means when we get into the details of communication via the bus.

We'll return to the bus in detail, with interface examples, using the example of the popular IBM PC/XT family. First, though, we need to look at the CPU's instruction set.

A COMPUTER INSTRUCTION SET

10.02 Assembly language and machine language

In order to understand bus signals and computer interfacing, you've got to understand what the CPU does when it executes various instructions. At this point, therefore, we would like to introduce the instruction set that goes with the IBM PC/XT family. Unfortunately, the instruction sets of most real-world microprocessors tend to be rich with complexities and extra features, and the Intel 8086 series is no exception. However, since our purpose is only to illustrate bus signals and interfacing (not fancy programming), we'll take a shortcut by laying out a subset of 8086 instructions. By leaving out the "extra" instructions we'll wind up with a compact set of instructions that is both understandable and complete enough to do any programming task. We'll then use it to show some examples of interfacing and programming. These examples will help convey the idea of programming at the "machine-language" level, something quite different from programming in a high-level language like FORTRAN or C.

First, a word on "machine language" and "assembly language." As we mentioned earlier, the computer's CPU is designed to interpret certain words as instructions and carry out the appointed tasks. This "machine language" consists of a set of binary instructions, each of which may occupy one or more bytes. Incrementing (increasing by one) the contents of a CPU register would be a single-byte instruction, for example, whereas loading

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

a register with the contents of a memory location would usually require at least two bytes, perhaps as many as five (the first would specify the operation and register destination, and four more would be necessary to specify an arbitrary memory location in a large machine). It is a sad fact of life that different computers have different machine languages, and there is no standard whatsoever.

Programming directly in machine language is extremely tedious, since you wind up dealing with columns of binary numbers, each bit of which has to be bit-perfect, so to speak. For this reason you invariably use a program called an *assembler*; it allows you to write programs using easily remembered mnemonics for the instructions, and symbolic names of your own choosing for memory locations and variables. This *assembly-language* program, really nothing more than a number of cryptic-looking lines of letters and numbers, is massaged by a program called an *assembler* to produce as its output a finished program in machine-language *object code* that the computer can execute. Each line of assembly code gets turned into a few machine-language bytes (1 to 6 bytes, for the 8086). The computer cannot execute assembly-language instructions directly. To make these ideas concrete, let's look at our subset of the 8086/8 assembly language and do a few examples.

10.03 Simplified 8086/8 instruction set

The 8086 is a 16-bit processor with a rich, and somewhat idiosyncratic, instruction set; part of its complexity stems from the designers' objective to maintain compatibility with the earlier 8080 8-bit processor. Newer CPUs, such as the 80286 and 80386, can still execute the full 8086 instruction set. We've gone through the instructions with a machete, keeping 10 arithmetic operations and 11 others. Here they are:

Instruction	What you call it	What it does
arithmetic		
`MOV` *b,a*	move	*a*→*b*; *a* unchanged
`ADD` *b,a*	add	*a*+*b*→*b*; *a* unchanged
`SUB` *b,a*	subtract	*b*−*a*→*b*; *a* unchanged
`AND` *b,a*	and	*a* AND *b*→*b* bitwise; *a* unchanged
`OR` *b,a*	or	*a* OR *b*→*b* bitwise; *a* unchanged
`CMP` *b,a*	compare	set flags as if *b*−*a*; *a,b* unchanged
`INC` *rm*	increment	*rm*+1→*rm*
`DEC` *rm*	decrement	*rm*−1→*rm*
`NOT` *rm*	not	1's complement of *rm*→*rm*
`NEG` *rm*	negate	negative (2's comp) of *rm*→*rm*
stack		
`PUSH` *rm*	push	push *rm* onto stack (2 bytes)
`POP` *rm*	pop	pop 2 bytes from stack to *rm*
control		
`JMP` *label*	jump	jump to instr *label*
`Jcc` *label*	jump conditional	jump to instr *label* if *cc* true
`CALL` *label*	call	push next adr, jump to instr *label*
`RET`	return	pop stack, jump to that adr
`IRET`	return from int	pop stack, restore flags, return
`STI`	set interrupt	enable interrupts
`CLI`	clear interrupt	disable interrupts
input/output		
`IN` `AX(AL)`,*port*	input	*port*→AX (or AL)
`OUT` *port*,`AX(AL)`	output	AX (or AL)→*port*

notes

b,a: any of *m,r r,m r,r m,imm r,imm*
rm: *r* or *m*, via various addressing modes
cc: any of `Z NZ G GE LE L C NC`
label: via various addressing modes
port: byte (via *imm*) or word (via `DX`)

A quick tour

Some explanations: The first six arithmetic instructions operate on pairs of numbers ("2-operand" instructions), which we've abbreviated as *b,a*, and which can be any of the 5 pairs listed in the notes; *m* means the contents of a memory location, *r* means the contents of a CPU register (there are 8), and *imm* means an *immediate* argument, which is a number stored in the next 1 to 4 bytes of memory following the

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

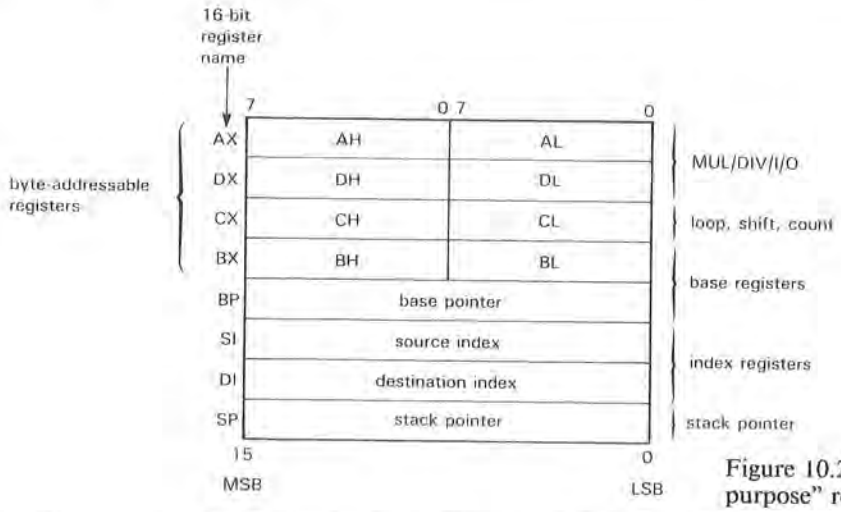


Figure 10.2. 8086 "general-purpose" registers.

instruction. Thus, for example, the instructions

```
MOV   count,CX
ADD   small,02H
AND   AX,007FH
```

have arguments of the form *m,r*, *m,imm*, and *r,imm*, respectively. The first copies the contents of register CX to a memory location that we've named "count"; the second adds 2 to the contents of another memory location called "small"; the third clears the top 9 bits of 16-bit register AX, while preserving the bottom 7 bits unchanged (a so-called masking operation). Note Intel's argument convention: The first argument is replaced or modified by the second argument. (In the next chapter we'll learn that Motorola decided to do it the other way around!)

The last four arithmetic operations take only a single operand, which can be either the contents of a register or memory. Here are two examples:

```
INC   count
NEG   AL
```

The first adds 1 to the contents of memory location "count," while the second changes the sign of register AL.

A detour: addressing

Before continuing, a word on registers and memory addressing. The 8086 claims to have 8 "general-purpose" registers, but after reading the fine print you'll realize that most of them have special uses (Fig. 10.2). Four of them (A–D) can be used either as single 16-bit registers (e.g., AX; think of "X" as "extended") or as a pair of byte registers (AH, AL; "high" and "low" halves). The BX and BP registers can hold addresses, as can the SI and DI registers, and tend to be used for addressing (see below). Special looping instructions (which we omitted from our short list) use register C, while multiply/divide and I/O instructions make analogous use of registers A and D.

Data used in instructions can be an immediate constant, a value held in a register, or a value in memory. You specify immediates by value, and registers by name, as in the examples above. To address memory, the 8086 provides six addressing modes, three of which are described by the diagrams in Figure 10.3. You can just name the variable *directly*, in which case its address gets assembled as a pair of bytes immediately following the instruction; you can put the variable's address in an addressing register (BX, BP, SI, or DI), then use an instruction that specifies addressing *indirectly* through the register; or you can combine the above, adding an immediate *displacement* to the value in a designated addressing register to get the variable's address. The indirect

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

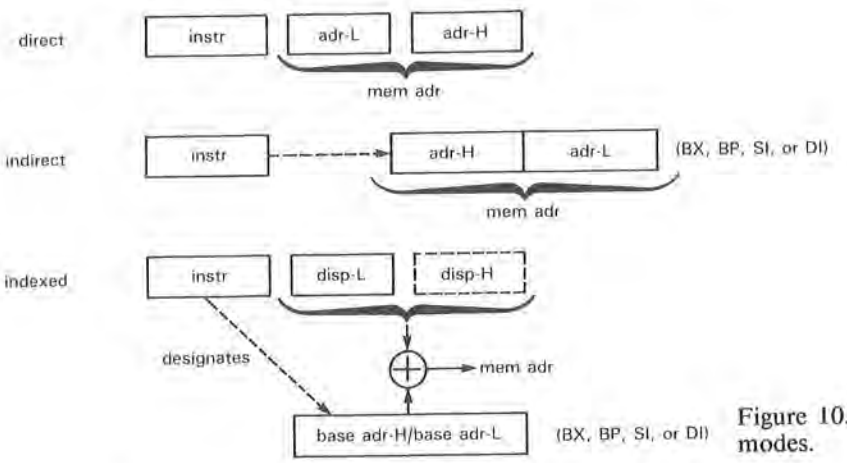


Figure 10.3. Some addressing modes.

mode is faster (assuming the address has already been loaded into an addressing register) and much better if you want to do something to a whole set of numbers (a *string* or *array*). Here are a few addressing examples:

```
MOV   count,100H       (direct,immediate)
MOV   [BX],100H        (indirect,immediate)
MOV   [BX+1000H],AX    (indexed,register)
```

The last two assume you've already put an address into BX. The last instruction copies the contents of AX to a memory location 4K (1000 hex) higher than BX points in memory; we'll give an example shortly showing how you could use this to copy an array.

There's one other complexity of 8086 memory addressing that we've swept under the rug: The "address" generated by any of the above addressing modes is not actually the final address, as should be obvious from the fact that it has only 16 bits (which can address only 64K bytes of memory). In fact, it's called an *offset*; to get an actual address, you add to the offset a 20-bit *base* formed by shifting left 4 bits the contents of a 16-bit *segment register* (there are four such registers). In other words, the 8086 lets you access groups of 64K bytes of memory at a time, with the location of those "segments" within a total memory size of 1Mbyte set by the contents of the segment registers. The use of 16-bit addressing in the 8086 was basically a big mistake, inherited from earlier generations of microprocessors. Newer processors (80386 onward, and the 68000 series) are done right, with 32-bit addressing throughout. Rather than complicate our examples, we'll simply ignore segments entirely; in real life you would, of course, have to worry about them.

Instruction set tour (continued)

The *stack* instructions PUSH and POP come next. A stack is a portion of memory, organized in a special way: When you put data onto the stack (a *push*), it goes into the next available spot ("top" of the stack); and when you retrieve data (a *pop*), it is taken from the top, i.e., it is the item last pushed onto the stack. Thus a stack is a consecutive list of data, stored last-in, first-out (LIFO). It may help to think of a bus driver's coin dispenser (or a lunchroom tray dispenser).

Figure 10.4 shows how it works. The stack lives in ordinary RAM, with the CPU's *stack pointer* (SP) keeping track of the location of the current "top" of the stack. The 8086 stack holds 16-bit words and grows *down* in memory as you push data onto it. The SP is automatically decremented by 2 before each PUSH, and incremented by 2 after each POP. Thus, in

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

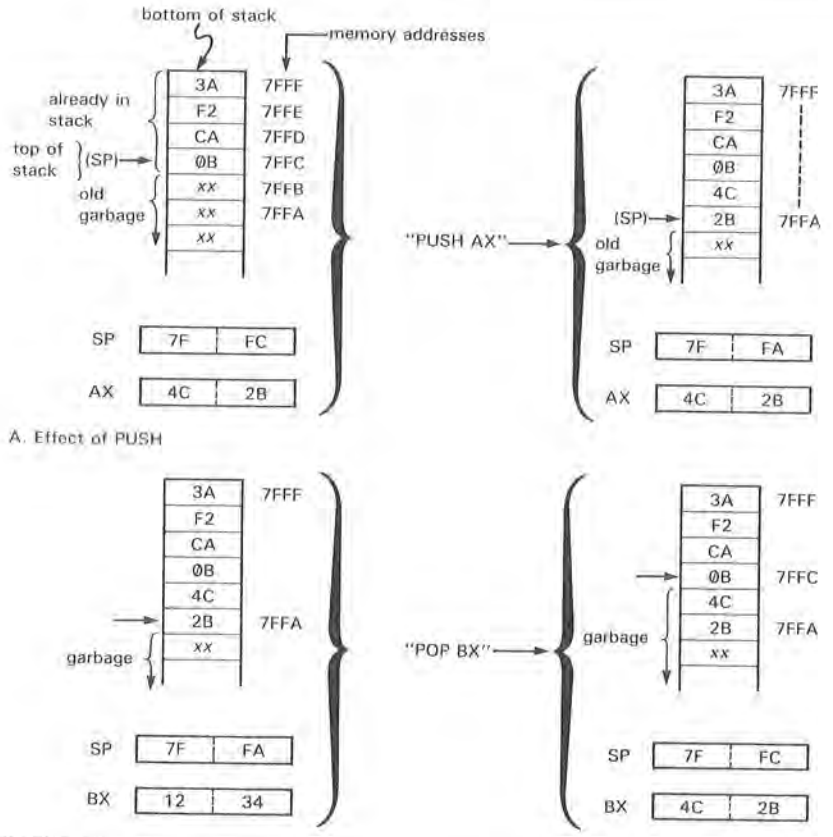


Figure 10.4. Stack operation.

the example, the 16-bit data in register AX is copied onto the top of the stack by the instruction PUSH AX; the SP is left pointing at the last byte pushed. POP reverses the process, as shown. As we will see, the stack plays a central role in subroutine calls and interrupts.

JMP causes the CPU to depart from its usual habit of executing instructions in sequential order, detouring instead to the instruction that you jump to. Conditional jumps (there are eight possibilities, indicated generically as *Jcc*) test the flag register (which lives in the CPU, and whose bits are set according to the result of the most recent arithmetic operation), then either jump (if the condition is true) or execute the next instruction in sequence (if the condition is not true). Program 10.1 shows an example. It copies 100 words from the array beginning at 1000 hex to a new array beginning 1K bytes (400H) higher.

Program 10.1

```
        MOV   BX,1000H          ;put array address in BX
        MOV   CL,100            ;initialize loop counter
LOOP:   MOV   AX,[BX]           ;copy array element to AX
        MOV [BX+400H],AX        ;then to new array
        ADD   BX,2              ;increment array pointer
        DEC   CL                ;decrement counter
        JNZ   LOOP              ;loop if count not zero
NEXT:   (next statement)        ;exit here when done
```

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

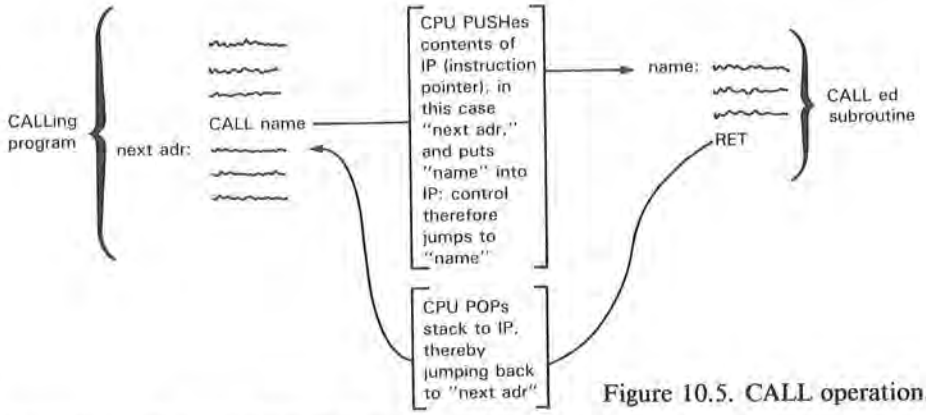


Figure 10.5. CALL operation.

Note the explicit loading of the pointer (to address register BX) and the loop count (to CL). The actual array of words had to move through a register (we chose AX) because the 8086 does not permit memory-to-memory operations (see the instruction set notes). At the end of the 100th pass through the loop, CZ = 0, and the jump nonzero (JNZ) instruction no longer jumps. This example will work, but in practice you would probably use one of the 8086's faster *string move* instructions. Also, it's good programming practice to use symbolic names for sizes and arrays, rather than constants like 400H and 1000H.

The CALL statement is a subroutine call; it's like a jump, except that the return address (the address of the instruction that would have come next, except for the intervening CALL) is pushed onto the stack. At the end of the subroutine you execute a RET statement, which pops the stack so the program can find its way home (Fig. 10.5). The three statements STI, CLI, and IRET have to do with interrupts, which we'll illustrate with a circuit example later in the chapter. Finally, the I/O instructions IN and OUT move a word or byte between the A register and the addressed port; more on this shortly.

10.04 A programming example

As the example above suggests, assembly language tends to verbosity, with a lot of little steps needed to do a basically simple thing. Here's another example: Suppose you want to increment a number, N, if it equals another number, M. This will typically be a tiny step in a larger program, and in higher level languages it will be a single instruction:

```
if (n==m) ++n;          (C)
IF (N.EQ.M) N=N+1       (Fortran)
if n=m then n:=n+1;     (Pascal), etc.
```

In 8086 assembler, it looks like Program 10.2. The assembler program will

Program 10.2

```
n   DW 0                        ;n (a "word") lives here, and
m   DW 0                        ;m lives here, both initialized to 0

        MOV   AX,n              ;get n
        CMP   AX,m              ;compare
        JNZ   NEXT              ;unequal, do nothing
        INC   n                 ;equal, increment n
NEXT:   (next statement)
          o
          o
          o
```

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

convert this set of mnemonics to machine language, generally translating each line of assembler *source code* to several machine-language bytes, and the resultant machine-language code will get loaded into successive locations in memory before being executed. Note that it is necessary to tell the assembler to assign some storage space for variables. This you do with the assembler *pseudo-op* "DW" (define word) (*pseudo-op* because it doesn't produce executable code). Unique symbolic labels (e.g., NEXT) can be used to tag instructions; this is usually done only if there is a jump to that location (JNZ NEXT). Giving some locations understandable (to you!) names and adding comments (separated by a semicolon) make the job of programming easier; it also means that you have a chance of understanding what you've written a few weeks later. Programming in assembly language can still be a nuisance, but it is often necessary to write short routines in it, callable from a higher-level language, to handle I/O. Assembly-language programs run faster than programs compiled from a higher language, so it is often used where speed is crucial (e.g., the innermost loop of a long numerical calculation). To some extent the development of the powerful C programming language has minimized the occasions when you must use assembly code. In any case, you can't really understand computer interfacing without understanding the nature of assembly-language I/O. The correspondence between mnemonic assembly language and executable machine language is explored further in Section 11.03, in that case illustrated by 68000 microprocessor programming.

BUS SIGNALS AND INTERFACING

A typical microcomputer data bus has about 50–100 signal lines, devoted to the transfer of data, addresses, and control signals. The IBM PC/XT is typical of a small machine, with 53 signal lines and 8 power/ground lines. Rather than throw them all at you at once, we will approach the subject by building up the bus, beginning with the signal lines necessary for the simplest kind of data interchange (programmed I/O) and adding additional signal lines as they become necessary. We will give some useful interface examples as we go along, to keep things comprehensible and interesting.

10.05 Fundamental bus signals: data, address, strobe

To move data on a shared (party-line) bus, you have to be able to specify the data, the recipient, and the moment when data is valid. Thus, a minimum bus must have DATA lines (for the data to be transferred), ADDRESS lines (to identify the I/O device or memory address), and some STROBE lines (which tell when data is being transferred). There are usually as many DATA lines as bits in the computer word, so a whole word can be transferred at once. In the PC, however, there are only 8 DATA lines (D0–D7); you can move a byte in one transfer, but to move a 16-bit word you have to do two transfers. The number of ADDRESS lines determines the number of addressable devices: If the bus is used for both I/O and memory (the usual situation) there will be 16 to 32 ADDRESS lines (corresponding to a 64Kbyte to 4Gbyte address space); a bus used for I/O only might have 8 to 16 ADDRESS bits (256 to 64K I/O devices). [The IBM PC talks to both memory and I/O on its bus, and has 20 ADDRESS lines (A0–A19), corresponding to a 1Mbyte address space.] Finally, data transfer itself is synchronized by pulses on additional "strobing" bus lines. There are two ways in which this can be done: by having separate READ and WRITE lines, with a pulse on one or the other synchronizing data transfer; or by having one STROBE line and one READ/WRITE′ line, with a pulse on STROBE synchronizing

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

data transfer in a direction specified by the level on the READ/WRITE′ line. The IBM PC uses the first scheme, with (active-LOW) read/write lines called IOR′, IOW′, MEMR′, and MEMW′; there are four because the PC distinguishes between memory and I/O, with individual pairs of read/write strobes for each.

These bus signals – DATA, ADDRESS, and the four strobes – would normally be all you need to do the simplest kind of data transfers. However, on the PC bus you need one more, called ADDRESS ENABLE (AEN), to distinguish normal I/O transfers from what's called "direct memory access" (DMA). We'll get to DMA in Section 10.12; for now, all you need to know is that AEN is LOW for normal I/O, and HIGH for DMA. We now have 33 bus signals: D0–D7, A0–A19, IOR′, IOW′, MEMR′, MEMW′, and AEN. Let's see how they work.

10.06 Programmed I/O: data out

The simplest method of data exchange on a computer bus is known as "programmed I/O," meaning that data is transferred via an `IN` or `OUT` statement in the program (the directions for `IN` and `OUT` are among the few things on which all computer manufacturers agree: `IN` always means *toward* the CPU, and `OUT` always means *from* the CPU). The whole process of data `OUT` (and memory write) is extremely simple and logical (Fig. 10.6). The ADDRESS of the recipient and the DATA to be sent are put onto the respective bus lines by the CPU. A write strobe (IOW′ or MEMW′) is asserted (LOW) by the CPU to signal the recipient that data is good. On the PC's bus the address is guaranteed valid beginning about 100ns before IOW′, and the data are guaranteed valid at least 500ns before the end of IOW′ (and for another 185ns thereafter). To play the game, the peripheral (in this case, an XY "vector" scope display) looks at the ADDRESS and DATA lines. When it sees its own address, it latches the information on the DATA lines, using the trailing edge of the IOW′ pulse as a clocking signal. That's all there is to it.

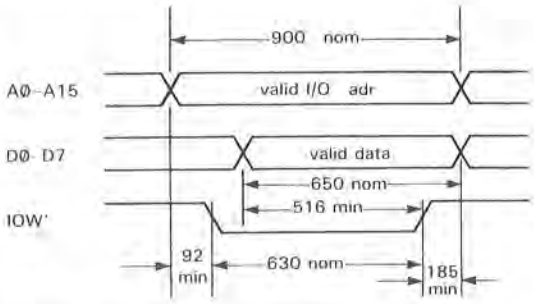


Figure 10.6. I/O WRITE cycle.

Let's look at the example shown in Figure 10.7. Here we have designed an XY scope display; you send it successive X, Y pairs of numbers, and it plots each point in turn on an XY display oscilloscope. First we have to pick an I/O address. Figure 10.8 shows the reserved and available I/O addresses on the IBM PC; we've chosen $3C0_H$ for the X register, and $3C1_H$ for the Y register. The '688 is an octal comparator with enable and LOW-true output on equality, giving a LOW output when the eight high-order bits A2–A9 match the fixed comparison bits, in this case when the address bus contains addresses 3C0–3C3 (you could use a bunch of gates, but an address comparator is more compact). We've also required AEN to be LOW, as explained earlier. The 3-input NANDs complete the address decoding, using A0 and A1, to give LOW outputs on individual addresses 3C0 and 3C1 (another method will be described shortly). Finally, these outputs are ANDed with IOW′ to get the clocks for the X and Y registers, which are '574 octal *D* flip-flops. These latch bytes from the data bus when (a) the correct address is present, (b) AEN is LOW, and (c) an IOW′ is sent. The 8-bit DACs convert the latched bytes to analog voltages, to drive the X and Y inputs

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

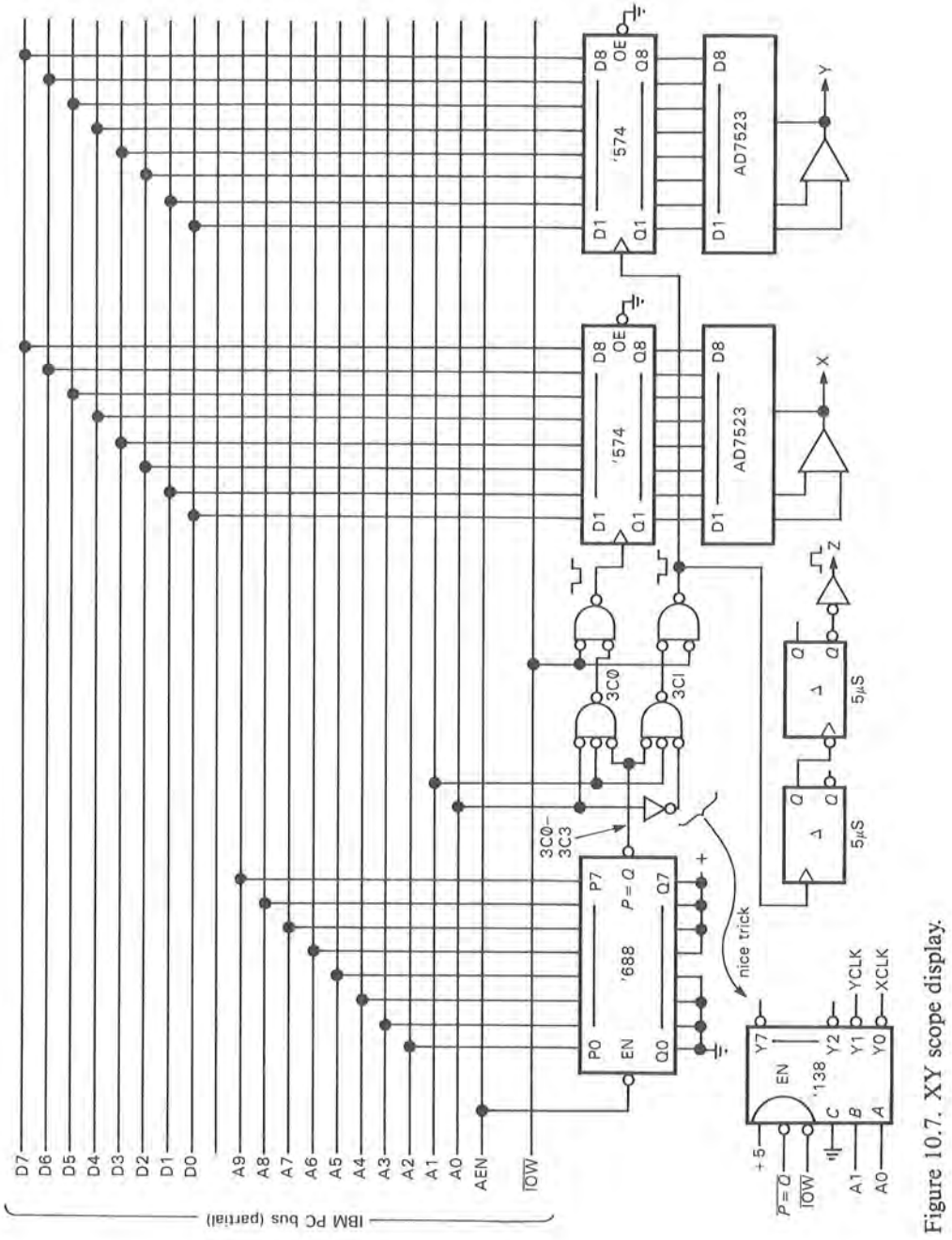


Figure 10.7. XY scope display.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

of a display scope. A pair of monostables generates a 5μs "unblanking" pulse a few microseconds after the Y coordinate has been latched, to intensify the selected spot on the scope (all scopes have a "Z input" for that purpose). To draw a graph or set of characters on the screen, all you do is output successive XY coordinates repetitively (send X, then Y), fast enough so the eye doesn't see the flicker. Microcomputers are fast enough to display a few thousand XY pairs repetitively without annoying flicker. Given the fact that video (raster-scanned) displays for viewing are commonplace on microcomputers, this example might be more useful as an ultra-high-resolution plotter for photographic "hard copy," using 14-bit DACs and a micro-spot-size hard-copy scope display (see the next exercise).

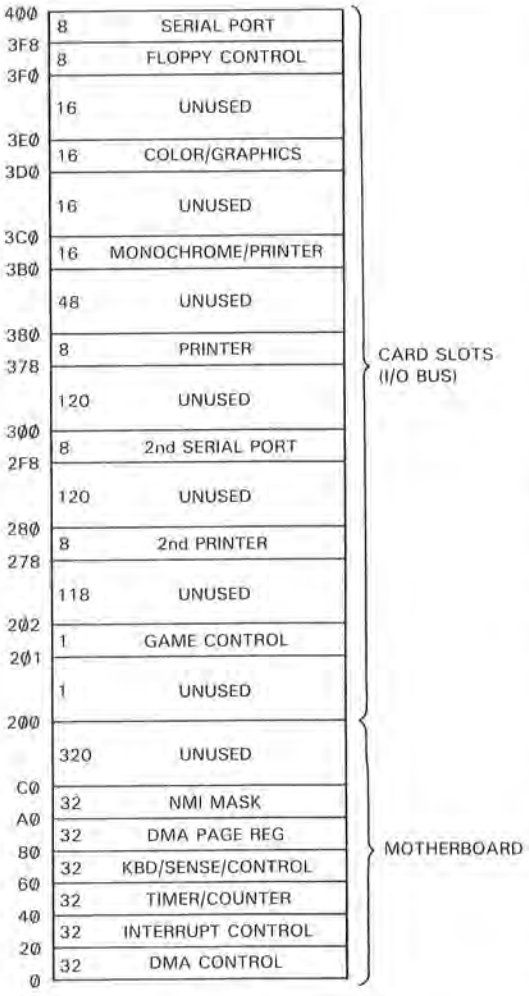


Figure 10.8. I/O addresses for IBM PC.

Some useful comments: (a) Note that we've arranged polarities so that the *trailing* edge of IOW′ clocks the *D* flip-flops; that is essential, since the data isn't yet valid on the leading edge of IOW′. If we were being very careful, we would check to see that required setup and hold times are satisfied for the '574s; in fact, for a slow bus like the PC's, you can't go wrong, since there are more than 500ns from valid data to trailing edge of IOW′. (b) You can save a few parts by using a strobed decoder in the address decoding circuitry, as indicated. Decoders like the '138 (3-line to 8-line) and '139 (dual 2-line to 4-line) include one or more enable inputs, and they are handy in this sort of application. (c) Note also that we could have combined the 3-input and 2-input NANDs into 4-input NANDs; we kept them separate only for clarity, decoding the addresses first, then ANDing with the IOW′ strobe. (d) In fact, we could have ignored A1 entirely, and the circuit would work just the same! However, it would then respond also to addresses 3C2 and 3C3 (as X and Y, respectively), in effect "wasting" two I/O locations. In practice you often cheat in this way, incompletely decoding the address, because it saves parts (and there is plenty of I/O space, even if you waste some). In this example we could then have connected IOW′ where A1 is now connected, and omitted the 2-input NANDs entirely. (e) An interface like this is more flexible if its address can be set using a DIP switch (or DIP jumper block); then you can always make sure its address doesn't conflict with that of another interface you've got somewhere else. In this case the change is simple – replace the "hardwired" address lines to the comparator with eight lines that

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

have switches to ground and pullups to +5 volts. (f) We used separate octal registers and DACs in this example for clarity. In real life you would probably choose a DAC with built-in latch (e.g., the "microprocessor-compatible" AD7528, a dual DAC with input latches); these even come in quad versions, (e.g., the AD7226) and in "double-buffered" versions with two cascaded latches for each DAC (e.g., the AD7225 quad).

EXERCISE 10.1

Redraw the address comparator logic with selectable I/O address.

EXERCISE 10.2

Redraw the XY display interface, using 16-bit DACs for both X and Y. You'll need four consecutive addresses: Assign the first two to the X register, and the last two to the Y register; use DIP-selectable I/O base address, of course. In each case the even address is the low-order byte, and the odd address is the high-order byte; that's the good choice, because that's how the 8086 stores 16-bit words, so you can use *word* I/O instructions to send data to your interface.

Programming the scope display

The programming to run this interface is straightforward. Program 10.3 shows what you do. The addresses of the first X and Y, and the number of points to be plotted, have to be available to the program. The display program will probably be a subroutine, with those parameters passed as arguments in the subroutine call. The program puts the addresses of the X and Y arrays (i.e., the address of the first X and Y) into address pointer registers SI and DI, and the byte count into CX. It then enters a loop in which successive XY pairs are sent to I/O ports 3C0 and 3C1. The X and Y pointers are advanced each time around, and the counter is decremented and tested for zero, which means the last point has been displayed; the pointers and counter are then reinitialized, and the process begins again.

A couple of important points: Once started, this program displays the XY array forever. In real life the program would probably check the keyboard to see if the operator wants the plot terminated. Alternatively, the display could be terminated after a specified time had elapsed, or by an "interrupt," which we will discuss shortly. With this sort of "refreshed" display, there usually isn't time to do much computing while displaying. A display device refreshed from its own memory takes that burden off the computer, and this is generally a better method. Nevertheless, if the objective is to make a precision plot for photographic hard copy, this program and interface (souped up as in Exercise 10.2) will do the job nicely.

Program 10.3

```
                                ;routine to drive XY display
INIT:   MOV   SI,xpoint         ;initialize x pointer
        MOV   DI,ypoint         ;initialize y pointer
        MOV   CX,npoint         ;initialize counter

PLOT:   MOV   AL,[SI]           ;get x byte
        OUT   3C0H,AL           ;send it out
        MOV   AL,[DI]           ;get y byte
        OUT   3C1H,AL           ;send it out
        INC   SI                ;advance x pointer
        INC   DI                ;advance y pointer
        DEC   CX                ;decrement counter
        JNZ   PLOT              ;not done, plot more stuff
        JMP   INIT              ;done, start over
```

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

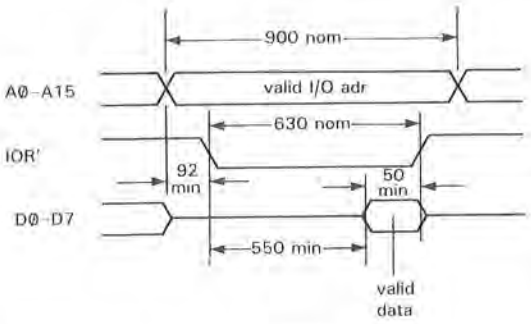


Figure 10.9. I/O READ cycle.

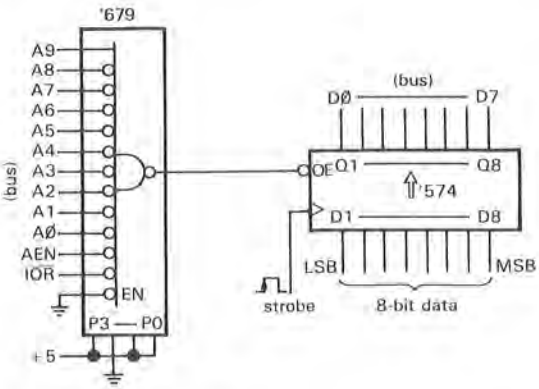


Figure 10.10. Parallel input port.

10.07 Programmed I/O: data in

The other direction of programmed I/O is equally simple. The interface looks at the ADDRESS lines as before. If it sees its own address (and AEN is LOW), it puts data onto the DATA lines coincident with the IOR′ pulse (Fig. 10.9). Figure 10.10 shows an example. This interface lets the PC read a byte latched in the '574 *D*-type register. Since the clock input and data inputs of the register are accessible to an external device, the register could hold just about any sort of digital information (the output of a digital instrument, A/D converter, etc.). For variety, we've eliminated all gates by using a '679 "12-bit address decoder" IC. It's a clever chip with 12 address inputs, an enable, and 4 "programming" inputs. If you want to decode a fixed address, it does the trick: It's functionally a 12-input NAND gate, for which a programmable number of the inputs can be inverted; the inverted inputs are always the lowest-numbered ones, and the number of them is the number you have asserted at the (4-bit) programming inputs.

In this case we've decided to plunder that lonely unused port at I/O address 200_H (Fig. 10.8). We need to recognize the state A9 = HIGH, A0–A8 = LOW. We might as well use the '679 to qualify the decoded address with AEN = LOW and IOR′ = LOW. So altogether we need a NAND with 11 inverted inputs and 1 noninverted input, which we get by hard-wiring the programming inputs with 11 in binary (1011). Then we connect the address lines and strobe as shown. When an

```
IN  AL,200H
```

instruction is executed, the CPU asserts 200_H on A0–A9, waits a while, then asserts IOR′ for 630ns. The CPU latches what it sees on the DATA bus (D0–D7) at the trailing edge of IOR′, then disasserts A0–A9. The peripheral's responsibility is to get the data onto D0–D7 at least 50ns before the end of IOR′; that's pretty relaxed timing, since it has known that data is being requested from it for at least 600ns. With typical HC or LS gate propagation times of 10ns, 600ns looks like forever.

Beginning with this example, we will omit the tangle of bus lines and simply call them out by name.

Bus signals: bidirectional versus one-way

From the two examples we've done so far you can see that some bus lines are *bidirectional*, for example the DATA lines: They are asserted by the CPU during write, but asserted by the peripheral during read. Both CPU and peripheral use three-state drivers for these lines. Others, like IOW′ and IOR′, are always driven by the CPU,

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

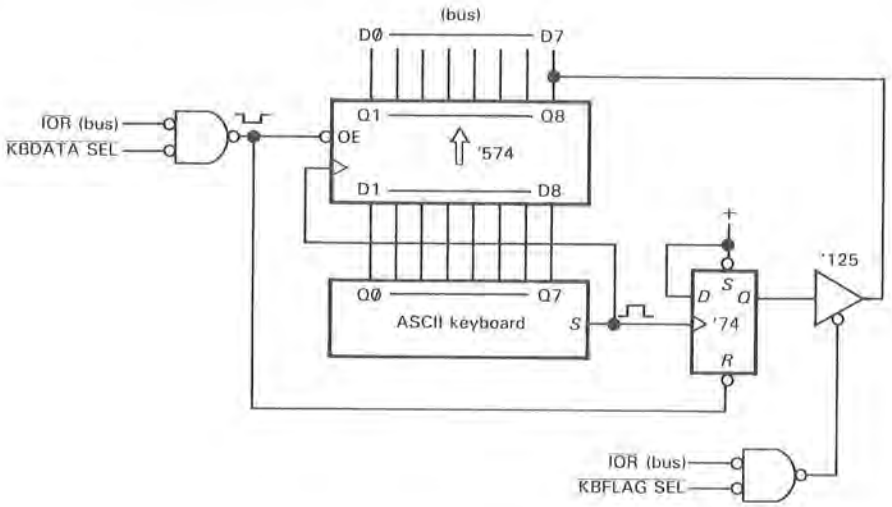


Figure 10.11. Keyboard interface with status bit.

with standard totem-pole driver chips. It is typical of computer buses to have both kinds of lines, using bidirectional lines for data that goes both ways, and one-way lines for signals that are always generated by the CPU (or, more accurately, generated by the associated bus control logic). There is always some clear protocol, like our rules for asserting/reading according to IOW′, IOR′, and ADDRESS, to prevent "bus contention" on these shared lines.

Of the signals so far, only the DATA lines are bidirectional; the ADDRESS lines, AEN, and strobes are one-way from the CPU. (Lest we give the wrong impression, we should point out that more complex computer systems permit other riders on the bus to become bus "masters"; obviously in such a system nearly all bus signals must be shared and bidirectional. The PC is unusually simple.)

10.08 Programmed I/O: status registers

In our last example, the computer can read a byte from the interface any time it wants to. That's nice, but how does it know when there's something worth reading? In some situations you may want the computer to read data at equally spaced intervals, as determined by its "real-time clock." Perhaps the computer instructs an A/D converter to begin conversions at regular intervals (via an `OUT` command), then reads the result a few microseconds later (via an `IN` command). That might suffice in a data-logging application. However, it is often the case that the external device has a mind of its own, and it would be nice if it could communicate what's happening to the computer without having to wait around.

A classic example is an alphanumeric input terminal, with someone banging away at a keyboard. You don't want characters to get lost; the computer has to get every character, and without much delay. With a fast storage device like disk or tape the situation is even more serious; data must be moved at rates up to 100,000 bytes per second without delay. There are actually three ways to handle this general problem: status registers, interrupts, and direct memory access. Let's begin with the simplest method – status registers – illustrated by the keyboard interface in Figure 10.11.

In this example, an ASCII keyboard drives a '574 octal *D*-type register, clocking in a character via the keyboard's STB

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

(strobe) output pulse when a key is struck. We rig up the standard programmed data-incircuit, as shown, using the three-state outputs of the '574 to drive the DATA bus directly. The input labeled KBDATA SEL′ comes from an address decoding circuit of the sort shown explicitly in the previous examples, and it goes LOW when the particular address chosen for this interface appears on the ADDRESS lines of the bus (in combination with AEN asserted LOW).

What's new in this example is the flip-flop, which gets set when a character is struck, and cleared when a character is read by the computer. It's a 1-bit *status* register, HIGH if there's a new character available, LOW otherwise. The computer can query the status bit by doing a data IN from the other address of this device, decoded (with gates, decoders, or whatever) as KBFLAG SEL′. You need only one bit to convey the status information, so the interface drives only the most significant bit, in this case with a '125 three-state buffer. (*Never* drive a bidirectional line with a totem-pole output!) The line coming into the side of the buffer symbol is the three-state output enable, asserted when LOW, as indicated by the negation bubble.

Program example: keyboard terminal

The computer now has a way to find out when new data is ready. Program 10.4 shows how. This is a routine to get characters from keyboard terminal, whose data port address is KBDATA (it's good programming style to define the actual numerical port addresses – which correspond to what the hardware decodes as KBDATA SEL, etc. – in some statements near the beginning of the program, as shown); each character is "echoed" on the computer's display device (port address = OUTBYTE). When it has gotten a whole line, it transfers control to a line-handling routine, which might do just about anything, based on what the line says. When it's ready for another line, it types an asterisk. This sort of function should make sense to you if you've had some experience with computers.

The program begins by initializing the character buffer pointer, by moving the *address* of the buffer that we just allocated to the address register BP. Note we can't just say

```
MOV  BP,charbuf
```

because that would load the *contents* of charbuf, not its address; in 8086 assembly language you use the word "offset" in front of a memory label to signify its address. The program then reads the keyboard status bit via an IN instruction, ANDs it with 80_H to keep only the status bit (this is called "masking"), and tests for zero. Zero means the bit isn't set, so the program loops. When a nonzero status bit is detected, it reads the keyboard data port (which clears the status flag flip-flop), stores it consecutively in the line buffer, increments the pointer (BP), and calls the routine that echoes the character to the screen. Finally, it checks to see if the line was terminated by a carriage return: If it wasn't, it goes back and loops on the keyboard status flag again; if it was a CR, it transfers control to the line handler, after which it types an asterisk and begins the entire process anew.

A subroutine has been used to display a character, since even that simple operation requires some flag checking and masking. The routine first saves the byte into AH, then reads and masks the screen's busy flag. Nonzero means the screen is busy, so it keeps checking; otherwise it restores the character to AL, sends it to the screen's data port, and returns.

Some notes on the program: (a) We could have omitted the keyboard flag masking step, since the MSB (where we put the flag bit in our hardware) is the sign bit; thus we could have used the instruction JPL KFCHK. However, this trick works only for testing the MSB and thus is

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

Program 10.4

```
                                ;keyboard handler -- uses flags
KBDATA  equ ***H                ;put kbd data port adr here
KBFLAG  equ ***H                ;ditto for kbd flag
KBMASK  equ 80H                 ;kbd flag mask
OUTBYTE equ ***H                ;put disp port adr here
OUTFLAG equ ***H                ;ditto for disp port flag
OUTMASK equ ***H                ;disp port busy mask

charbuf  DB 100 dup(0)          ;allocates buffer of 100 bytes

INIT:  MOV  BP,offset charbuf ;initialize char buffer pointer
KFCHK: IN   AL,KBFLAG         ;read kbd flag
       AND  AL,KBMASK         ;mask unused bits
       JZ   KFCHK             ;flag not set -- no data
       IN   AL,KBDATA         ;get new kbd byte
       MOV [BP],AL            ;store it in line buffer
       INC  BP                ;advance pointer
       CALL TYPE              ;echo last char to display
       CMP  AL,0DH            ;was it carriage return?
       JNZ  KFCHK             ;if not, get next char
LINE:    o                    ;if so, do something with line
         o                    ;keep at it
         o                    ;don't quit now
         o                    ;done at last!
       MOV  AL,'*'
       CALL TYPE              ;type a "prompt" -- asterisk
       JMP  INIT              ;get another line

                              ;routine to type character
                              ;types and preserves AL
TYPE:  MOV  AH,AL             ;save the char in AH
PCHK:  IN   AL,OUTFLAG        ;check printer busy?
       AND  AL,OUTMASK        ;printer flag mask
       JNZ  PCHK              ;if busy check again
       MOV  AL,AH             ;restore char to AL
       OUT  OUTBYTE,AL        ;type it
       RET                    ;return
```

somewhat specialized. (b) In keeping with good programming practice, the carriage return symbol (0DH) and asterisk probably should be defined constants, similar to KBMASK. (c) The line handler probably should be a subroutine, also. (d) Characters will be lost if the line handler takes too long; this leads us to the more elegant approach of *interrupts*, which we'll take up shortly. (e) Keyboard and terminal handlers are used so often that the PC provides built-in handlers, accessed through "software interrupts" (we'll see them later); thus, our program isn't even needed!

Status bits generalized

This keyboard example illustrates status bit protocol; but it's so simple that you may come away with the wrong idea. In an actual peripheral interface of some complexity, there will usually be several flags to signal various conditions. For example, in a magnetic-tape interface you will normally have status bits for beginning of tape, end of reel, parity error, tape in motion, etc. The usual procedure is to put all the status bits into one byte or word, so that a data IN command from the status

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

register gets all bits at once. Typically you would have a bit indicating any of a set of error conditions as the MSB of the status word, so a simple check of sign tells if there are *any* errors; if there are, you test specific bits of the word (by ANDing with masks) to find out what's wrong. Furthermore, in a complex interface you probably wouldn't have the status bits reset "automatically," as we did with our single bit; instead, a data OUT statement might be used, each bit of which clears a specific flag.

EXERCISE 10.3

With our keyboard interface there is no way for the computer to know if it missed a character. Modify the circuit so there are two status bits: CHAR READY (that's what we have already) and LOST DATA. The LOST DATA flag should be readable as D6 on the same status port as CHAR READY; it is 1 if a key was struck before the previous character was fetched by the computer, zero otherwise.

EXERCISE 10.4

Add a program segment to Program 10.4 that checks for lost data. It should call a subroutine called LOST if it detects lost data; otherwise it should work as before.

10.09 Interrupts

The use of status flags just illustrated is one of three ways for a peripheral device to "tell" the computer when some action needs to be taken. Although it will suffice in many simple situations, it has the serious drawback that the peripheral cannot "announce" that some action needs to be taken – it has to wait to be "asked" by the CPU, via a data IN command from its status register. Devices that need quick action (such as disks or latency-sensitive real-time I/O) would have to have their status flags queried often, and with a few such devices in a computer system the CPU would soon find itself spending most of its time checking status flags, as in the last example.

Furthermore, even with continual status flag checking you can still get in trouble: In the last example, for instance, the CPU will have no trouble keeping up with someone typing at the keyboard when it is in the main (flag checking) loop. But what if it spends 1/10 second in the line-handling portion? Or what if the display device is a slow one, making the program wait for its busy flag to clear?

What is needed is a mechanism for a peripheral to *interrupt* the normal action of the CPU when something needs to be done. The CPU can then check the status register to find out what the trouble is, take care of what needs to be done, and go back to its normal business.

To add interrupt capability to a computer, it is necessary to add a few new bus signals: At least one shared line for peripherals to signal an interrupt, and (usually) a pair of lines by which the CPU can determine who interrupted. As luck would have it, the IBM PC is not a very instructive example, because it does not implement a full interrupt capability. What it lacks in power, though, it more than makes up for in simplicity; implementing hardware interrupts in a PC peripheral interface is like falling off a log.

Here's how it works: The PC bus has a set of 6 *interrupt request* lines, called IRQ2–IRQ7. They are positive-true inputs to the CPU's support circuitry (specifically, to the 8259 interrupt controller). To make an interrupt, you simply bring one of the lines HIGH. If interrupts are enabled in general (along with the particular IRQ you assert), the CPU will break off after its next instruction, then (after saving its flags and current location onto the stack) jump to an "interrupt-handler" program somewhere in memory. You write the handler to do what you want (e.g., get keyboard data), and you can put the handler anywhere you wish, because the CPU figures out where to jump by

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

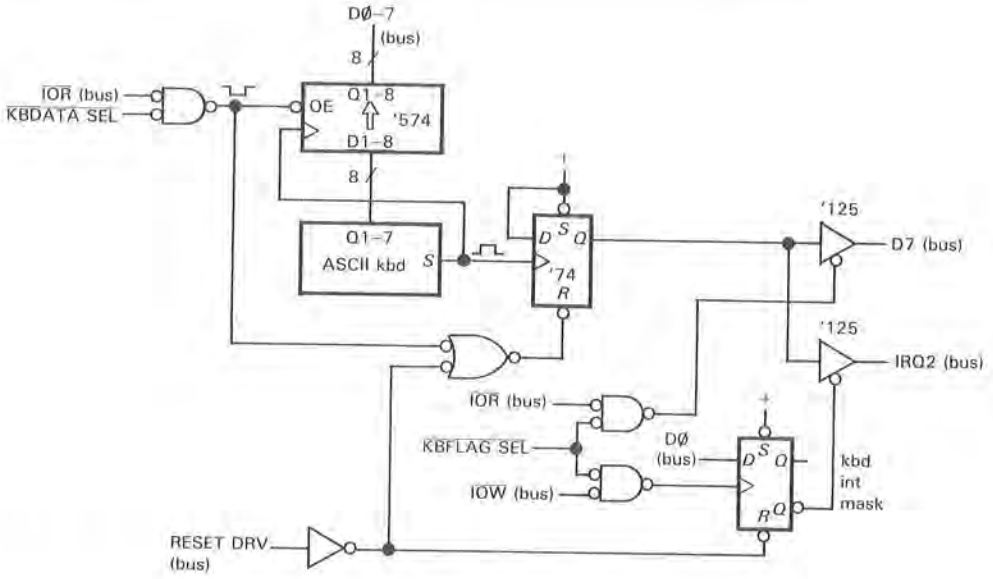


Figure 10.12. Keyboard interface with interrupts.

looking for the handler's 4-byte address in a special location in low memory. That location depends on which IRQ you've asserted; for the 8086 it is given in hex by $20 + 4n$, where n is the interrupt level. For example, the CPU would respond to an interrupt on IRQ2 by jumping to the (4-byte) address stored in locations 28_H through $2B_H$ (it's just like indirect addressing, except that the address is found in memory rather than in a register); of course, you would have cleverly arranged for the starting address of your handler to be there. At the end of your handler you execute an IRET instruction, which causes the CPU to restore the preexisting flag register and jump back to wherever it was when the interrupt happened.

Let's illustrate by adding interrupts to the keyboard interface (Fig. 10.12). We've left the flag bit ("character ready") and programmed I/O circuitry essentially as before, except that we've ORed the flag clear with a new bus line, RESET DRV, a CPU output that is momentarily asserted HIGH when the computer is turned on. This signal is generally used to force your flip-flops and other sequential logic into a known state at power-up. Obviously it should reset a flag that indicates a valid byte is ready to be claimed (and that, in our new interface, will even cause an interrupt). The only other change we've made is to use a compact notation for the byte-wide data paths, to make the diagram easy to read.

The new interrupt circuitry consists of a driver to assert IRQ2 when a character is ready. That's all the new hardware you need. Although not strictly necessary, we've added the capability to disable the interrupt driver (it's a three-state buffer) by sending a byte with D0 LOW to the KBFLAG port address. This would be used if you wanted to plug in another peripheral with interrupts at the same IRQ level, allowing only one peripheral to use its interrupts at any given time (later we'll have further explanation on this awkward point).

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

10.10 Interrupt handling

The IBM PC/XT family makes interrupt handling easy (though limited in flexibility) by using an 8259 interrupt-controller IC on the motherboard. This chip does most of the hard work, which consists of prioritizing, masking, and asserting vectors (we'll describe these after finishing the example). The CPU, for its part, recognizes the interrupt and responds by saving the instruction pointer and flag register, disabling further interrupts, then making a jump via the corresponding address stored in the low-memory vector area. Your handler program does the rest, namely: (a) save (push) any registers you'll be using (remember that the interrupted program can't prepare for the interrupt, since it can happen anywhere in the running program; it's a bolt out of the blue), (b) figure out what needs to be done, by reading status register(s) if necessary, (c) do it, (d) restore the saved registers from the stack, (e) tell the 8259 you're done (by sending an "end-of-interrupt" byte 20_H to its register at I/O address 20_H), and finally (f) execute a return from interrupt instruction IRET; this causes the CPU to restore the old flag register that it saved on the stack, and jump (via the old instruction pointer it saved on the stack) back to the program that was interrupted. Somewhere in the program, you must have (g) loaded the handler's address into the vector location corresponding to the IRQ level used by the hardware, and told the 8259 to enable interrupts at that level.

Program 10.5 shows the code for the keyboard with interrupt. Here's the overall scheme: The main program sets things up, then loops on a flag (in memory, not hardware) that the interrupt handler sets when it recognizes a carriage return; when the main program sees that flag set, it goes off and does something with the line, then returns to the flag-checking loop. The handler, entered at each interrupt, puts a character into the line buffer, sets the flag if it was a carriage return, then returns.

Let's look at the program in some detail. After defining port addresses and the all-critical vector location for IRQ2, it allocates 100 bytes (initially filled with zeros) for the character buffer. The actual program execution begins by putting the buffer address in address register SI, zero in the end-of-line flag, and the address of the handler (which begins with KBINT) in location 28_H. To enable level-2 interrupts in the 8259, we clear bit 2 of its existing mask (IN, AND, OUT); then we enable CPU interrupts and send a 1 to KBFLAG, which enables the three-state driver. Now we're running. The program then loops, with interrupts secretly happening right under the main program's nose, until it mysteriously finds "buflg" set. It resets the pointer and flag immediately (in case another interrupt occurs soon), then gobbles up the line. It would be well advised either to move quickly or to copy the line to another buffer, since another interrupt (with a new byte to go in the buffer) could come along in a few milliseconds; in that time you can execute a few thousand instructions, however, more than enough to copy the line.

The interrupt handler is a separate little piece of code, with no entry from the main program. It gets entered upon a level-2 interrupt, via its address that we initially loaded into 28_H. It knows exactly what it has to do, and it does it without complaining: It saves AX (since it plans to clobber it), reads the character from the keyboard data port, puts it in the buffer, increments the pointer, echoes the character to the screen, sets the flag if it was a carriage return, sends end-of-interrupt to the 8259, restores AX, and returns.

If you look back at our list of handler tasks above, you'll see that we omitted just

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

Program 10.5

```
                                ;keyboard handler -- uses interrupts
KBVECT equ word pntr 0028H      ;INT2 vector
KBDATA equ ***H                 ;put kbd data port adr here
KBFLAG equ ***H                 ;put kbd flag port adr here

buflg    DB   0                 ;allocates "buffer-full" flag
charbuf  DB 100 dup(0)          ;allocates 100-byte character buffer

       CLI                      ;disable interrupts
       MOV  SI,offset charbuf   ;initialize buffer pointer
       MOV  buflg,0             ;and end-of-line flag
       MOV  KBVECT,offset KBINT ;handler adr to vector area
       IN   AL,21H              ;existing 8259 int mask
       AND  AL,0FBH             ;clear bit 2 to enable INT2
       OUT  21H,AL              ;and send to 8259 OCW1
       STI                      ;enable interrupts
       MOV  AL,1
       OUT  KBFLAG,AL           ;enable hardware 3-state driver

LNCHK: MOV  AL,buflg
       JZ   LNCHK               ;loop until end-of-line flag set

LINE:  MOV  SI,offset charbuf   ;reset pointer
       MOV  buflg,0             ;and line flag
       MOV  AL,'*'
       CALL TYPE                ;type prompt "*"
         o                      ;do something with line
         o
         o
       JMP  LNCHK               ;and wait for another line

                                ;keyboard interrupt handler
                                ;an INT2 lands you here, via vector we loaded
KBINT: PUSH AX                  ;save AX register, used here
       IN   AL,KBDATA           ;get data byte from keyboard
       MOV  [SI],AL             ;put it in line buffer
       INC  SI                  ;and advance pointer
       CALL TYPE                ;echo to screen
       CMP  AL,0DH              ;check for carriage return
       JNZ  HOME                ;not a CR -- return
       MOV  buflg,0FFH          ;CR -- set end-of-line flag
HOME:  MOV  AL,20H
       OUT  20H,AL              ;end-of-interrupt signal to 8259
       POP  AX                  ;restore old AX
       IRET                     ;and return
```

one step, namely reading status flags to figure out which of several actions needed to be done. That's unnecessary here, though, because there's only one reason to interrupt, namely a new keyboard character needs to be read. (The programmer obviously has to understand under what conditions the hardware makes an interrupt, and what is required to service the interrupt.)

A few notes on this program: First, even though we're using interrupts, the program seems as dumb as before – it loops continually on the end-of-line flag.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

However, it could be doing other things, if there were things to do. In fact, it does just that beginning at statement LINE, where it processes the finished line; during that time, interrupts make sure that new characters are put into the buffer, whereas they would have been lost in our previous example without interrupts.

This brings up a second point, namely, even with interrupts we're still in trouble if the program is doing things with the previous line when the next line has been completely entered. Of course, *on the average* the program simply has to keep up with keyboard entry; but you could have a situation in which the line user occasionally spends a lot of time, and you need to buffer more than one line temporarily. One solution to this is to make a copy to a second buffer, or to alternate between two buffers. An elegant alternative is to organize input as a queue, implemented as a "ring buffer" (or "circular buffer"), in which a pair of pointers keep track of where the next input character goes, and where the next character is removed. The interrupt handler advances the input pointer, and the line user advances the output pointer. Such a ring buffer might typically be 256 bytes long, permitting the line user to get behind by a few lines.

A third point concerns the interrupt handler itself. It's usually best to keep it short and simple, perhaps setting flags to signal the need for complicated operations in the main program. If the handler does become long-winded, you risk losing data from other interrupting devices, because interrupts are disabled when the CPU jumps into the handler. The solution in this case is to re-enable interrupts *within* your handler with an STI instruction, after doing the critical things that have to be done first. Then if an interrupt occurs, your interrupt handler will itself be interrupted. Since flags and return addresses are stored on the stack, the program will find its way back, first to your handler, finally to the main program.

10.11 Interrupts in general

Our keyboard example illustrates the essence of interrupts – a spontaneous hardware request for attention by a peripheral, producing a program jump to a dedicated handler routine (usually resulting in some programmed I/O), followed by a return to the code that was interrupted. Other examples of interrupting devices are real-time clocks, in which a periodic interrupt (often 10 per second, but 18.2 per second in the PC) signals a timekeeping routine to advance the current time; another example is a parallel printer port, which interrupts each time it is ready for a new character. By using interrupts, these peripherals let the computer interleave other tasks simultaneously; that's why you can be doing word processing while your PC is printing a file (and, of course, keeping proper time throughout).

The IBM PC does not, however, illustrate the full generality of interrupts. As we saw, it has a set of six IRQ lines on the bus, each one of which can be used only for a single interrupting device. The IRQ lines are numbered according to priority; in the event of multiple interrupts, the lowest-numbered interrupt is serviced first. Four of the IRQ lines are preassigned to essential peripherals, namely the serial port (IRQ4), hard disk (IRQ5), floppy disk (IRQ6), and printer port (IRQ7), leaving only IRQ2 and IRQ3 available [lines for two additional IRQ levels recognized in the IBM PC are not even brought out on the bus, being used on the motherboard for the 18.2Hz clock (IRQ0) and the keyboard (IRQ1)]. If you were to add a streaming tape backup or local area network, you would have to use IRQ2 and IRQ3. Furthermore, the interrupt is *edge-triggered*, which frustrates any reasonable possibility of using wired-OR to combine several peripherals on a single IRQ line.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

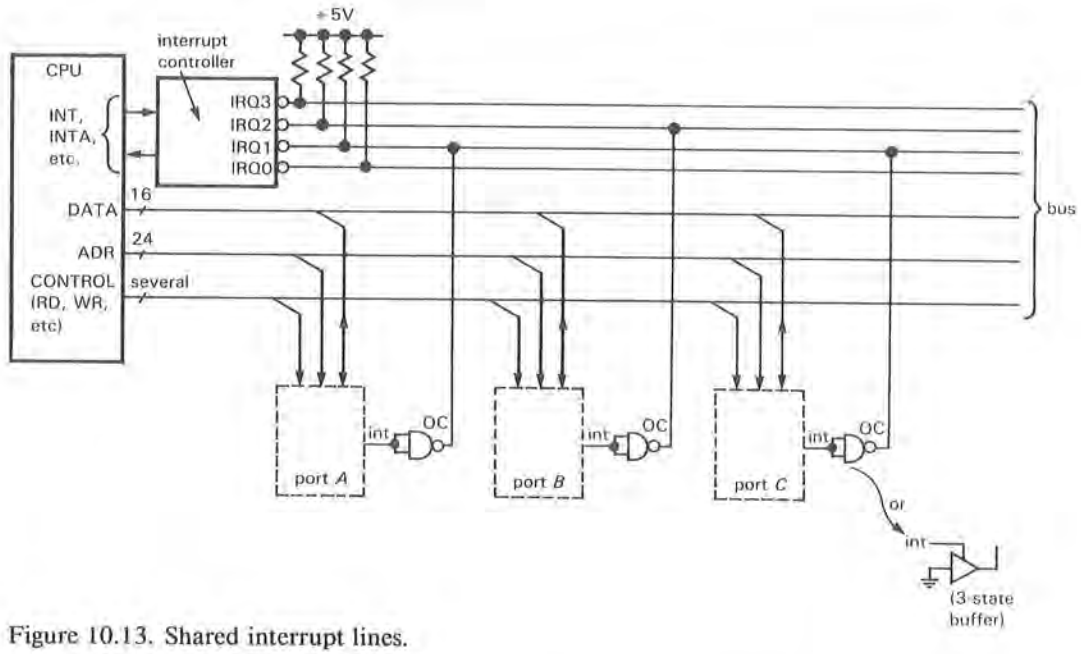


Figure 10.13. Shared interrupt lines.

□ *Shared interrupt lines*

The usual interrupt protocol, as implemented on many microcomputers, circumvents these limitations. Look at Figure 10.13. There are several (prioritized) IRQ-type lines; these are negative-true inputs to the CPU (or its immediate support circuitry). To request an interrupt, you pull one of the IRQ′ lines LOW, using an open-collector (or three-state) gate, as shown (note the trick for using a three-state gate to mimic an open-collector gate). The IRQ′ lines are shared, with a single resistive pullup, so you can put as many devices on each IRQ′ line as you want; in our example two ports share IRQ1. You would generally connect a latency-sensitive (impatient) device to a higher-priority IRQ′ line.

Since the IRQ′ lines are shared, there could always be another device interrupting on the same line at the same time. The CPU needs to know who interrupted, so it can jump to the appropriate handler. There is a simple way, and a complicated way, to do this. The simple way is called *autovectored polling* and is used nearly universally (though not on the IBM PC). Here's how it works.

□ *Autovectored polling.* Some circuitry on the CPU board (we'll have an example in Chapter 11) instructs the microprocessor that it is to use autovectoring, which works just like the IBM PC – each level of interrupt forces a jump through a corresponding vectoring location in low memory. For example, the 68000 microprocessor family we'll meet in Chapter 11 has seven levels of prioritized interrupt, which autovector through 4-byte pointers stored in the 28 (7×4) locations 64_H through $7F_H$. You put the addresses of the handlers in those locations, just as in our example above. For example, you would put the (4-byte) address of the handler for a level-3 interrupt in hex locations 6C through 6F.

Once in the handler, you know which level of interrupt you're servicing; you just don't know which particular device caused the interrupt. To find out, you

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

simply check the status registers of each of the devices connected to that level of interrupt (a device *never* requests an interrupt without also indicating its need by setting one or more readable status bits). If a bit is set indicating that something needs to be done, you do it, including whatever it takes to cause the device to disassert its IRQ′: Some devices (like our keyboard) clear their interrupt when read, whereas others may need a particular byte sent to some I/O port address.

If the device you serviced was the only one interrupting at that level, that IRQ′ will now be HIGH upon returning to the interrupted program, and execution will continue. However, if there had been a second interrupting device at the same level, that IRQ′ line will still be held LOW (by the wired-OR action of the shared IRQ′ line) upon return from the service routine, so the CPU will immediately autovector back to the same handler. This time the polling will find the other interrupting device, do its thing, and return. Note that the order in which you poll status registers effectively sets up a "software priority," in addition to the hardware priority of the multiple IRQ′ levels.

□ *Interrupt acknowledgment.* We shouldn't leave the subject of interrupts without mentioning a more sophisticated procedure for identifying who interrupted – *interrupt acknowledgment.* In this method the CPU doesn't need to poll the status registers of possible interrupters, because the interrupting device *tells* the CPU its name, when asked. The interrupter does this by putting an "interrupt vector" (usually a unique 8-bit quantity) onto the DATA lines in response to an "interrupt acknowledge" signal that the CPU generates during the interrupt processing.

Nearly every microprocessor generates the needed signals. The sequence of events goes like this: (a) The CPU notices a pending interrupt. (b) The CPU finishes the current instruction, then asserts (i) bus signals that announce an interrupt, (ii) the interrupt level being serviced (on the low-order ADDRESS lines), and (iii) READ-like strobes that invite the interrupting device to identify itself. (c) The interrupting device responds to this bus activity by asserting its identity (interrupt vector) onto the DATA lines. (d) The CPU reads the vector and jumps into the corresponding unique handler for the interrupting device. (e) The handler software, as in our last example, reads flags, gets and sends data, etc., as needed; among its other duties, it must make sure the interrupting device disasserts its interrupt. (f) Finally, the interrupt handler software returns control to the program that was interrupted.

Sharp-eyed readers may have noticed a flaw in the procedure just outlined. In particular, there has to be a protocol to ensure that only one device asserts its vector, since there may be several simultaneous interrupting devices at the same IRQ level. The usual way to handle this is to have a bus signal (call it INTP, "interrupt priority") that is unusual in not being shared by devices on the bus, but rather is passed along *through* each device's interface circuit, beginning as a HIGH level at the device closest to the CPU and threading along through each interface. That's called a "daisy chain" in the colorful language of electronics. The rule for INTP hardware logic is as follows: If you have not requested an interrupt at the level being acknowledged, pass INTP through to the next device unchanged; if you *have* interrupted at that level, hold your INTP output LOW. Now the rule for asserting your vector goes like this: Put your vector number onto the data bus when requested by the CPU only if (a) you have an interrupt pending at the level being acknowledged and (b) your input INTP is HIGH. This guarantees that

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

only one device asserts its vector; it also establishes a "serial priority" chain within each IRQ level, with devices electrically closest to the CPU getting serviced first. Computers that implement this scheme have little jumper plugs to pass INTP over unused motherboard slots. Don't forget to remove these jumpers when you plug in a new interface card (and put them back when you take one out!).

There is a nice alternative to the serial daisy-chain method of interrupt acknowledgment: Instead of threading a line through each possible interrupter, you bring individual lines back from each one to a priority encoder (Section 8.14), which in turn acknowledges the interrupt by asserting the identity of the highest-priority interrupting device. This scheme avoids the nuisance of daisy-chain jumpers. We will describe it in detail in Section 11.4 (Fig. 11.8).

In most microcomputer systems it isn't worth implementing the full-blown interrupt acknowledgment just described. After all, with 8-level autovectoring you can handle up to 8 interrupting devices without polling, and several times that number with polling. Only in large computer systems, in which you demand fast response with dozens of interrupting devices present, might you succumb to the complexity of the interrupt acknowledgment protocol, whether with serial daisy-chained hardware priority or with parallel priority encoding.

However, it is important to realize that even simple computers may be using vectored interrupt acknowledgment *internally*. For example, the simple 6-level autovector interrupt scheme of the IBM PC seen by the bus user is actually generated by an 8259 "programmable interrupt controller" chip that lives close to the CPU and generates the proper interrupt acknowledgment sequence just described (see below). This is necessary because the 8086 (and successors) can't implement autovectoring by themselves. On the other hand, the popular 68000 series of CPU chips can implement autovectoring internally, with just a single external gate (see Chapter 11).

□ *Interrupt masks*

We put a flip-flop in our simple keyboard example so that its interrupts could be disabled, even though the 8259 controller lets you turn off ("mask") each level of interrupt individually. We did that so that some other device could then use IRQ2. For a bus with shared (*level*-sensitive) IRQ' lines, it is especially important to make each interrupt source maskable, again with an I/O output port bit. For example, a printer port normally interrupts each time its output buffer is empty ("give me more data"); when you've finished printing, though, you don't care. The obvious solution is to turn off printer interrupts. Since there might be other devices hooked to the same interrupt level, you must not mask that whole level; instead, you just send a bit to the printer port to disable its interrupts.

□ *How the IBM PC got the way it is*

The 8086/8 microprocessor used in the IBM PC actually implements the full vectored interrupt acknowledgment protocol. To keep things simple, however, the PC designers used an 8259 interrupt controller IC on the motherboard. The way it is used in the PC, it has a set of IRQ inputs from the I/O bus card slots (that's where you make your interrupt requests), and it connects to the microprocessor's data bus and signal lines. When it gets a request on an IRQ line from a peripheral, it figures out priority and goes through the whole business of asserting the corresponding vector onto the data bus. It has a mask register (accessible as I/O port 21_H) so that you can disable any specified group of interrupts.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

The 8259 lets you select (through software) either *level-* or *edge*-triggered interrupts on its IRQ input lines, according to a byte sent to a control register (I/O port 20_H). Unfortunately, the PC designers decided to use edge triggering, probably because that makes it a little easier to implement interrupts (for example, you can just connect the real-time clock's square-wave output directly to IRQ0). If they had selected level-sensitive interrupts instead, you would be able to hang multiple interrupting devices on each IRQ′ line, with software polling as illustrated above. Unfortunately, the PC's ROM BIOS (Basic Input/Output System) and operating system (not to mention the hardware) assume edge triggering, so the choice is irrevocable. (Nearly all other computers, including even the successors to the PC and PC/AT, use level-sensitive interrupts.)

There is a partial solution to this problem. As long as there is an IRQ line available, you *can* combine several interrupting devices on a single PC board, with logic to generate edge-triggered interrupts on that single IRQ line; in fact, you could use your own 8259 (with its I/O ports accessible to the CPU) to do the job. But, since the interrupting devices have to know about each other, you can't use this scheme for independent plug-in peripherals. Furthermore, you still use up an IRQ line per card, and in a complicated system, given that there are only two IRQ levels free in the IBM PC, there will not be enough to go around.

Software interrupts

The Intel 8086-series of CPUs have an instruction ("`INT n`," where n is 0–255) that allows you to produce the same kind of vectored jump as an actual hardware interrupt. In fact, among its 256 possible jump vectors are duplicates of the 8 levels of IRQ-requested hardware interrupts (`INT 8` through `INT 15`, to be exact). Thus, you can make a "software interrupt" from a program statement. The IBM PC uses these software interrupts to let you communicate with the operating system and its various ROM-based utilities. For example, `INT 5` sends a replica of the screen to the printer. `INT 21H` turns out to be particularly important, because it is a function call to the operating system; you tell the system which DOS function you want by putting the corresponding number in register `AH` before executing the `INT 21H`.

Don't confuse these software interrupts with the externally triggered hardware interrupts we've been talking about. Software interrupts turn out to be a handy way of implementing vectored jumps from user code into system software. But they are not real interrupts, in the sense of a hardware call for attention from an external autonomous device. On the contrary, you build these into your software, you know when they are coming (that's why you can pass arguments through registers), and they are merely the response (albeit identical with what follows a true interrupt) of the CPU to its own code. You might think of software interrupts as a clever way to extend the instruction set.

10.12 Direct memory access

There are situations in which data must be moved very rapidly to or from a device. The classic examples are fast mass-storage devices like disk or tape, and on-line data-acquisition applications such as multichannel pulse-height analysis. Interrupt-initiated programmed processing of each data transfer in these examples would be awkward, and probably too slow. For example, data come from a "high-density" floppy disk at about 500kbit/s, or one byte every 16μs. With all the bookkeeping involved in handling an interrupt, data would almost certainly be lost, even

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

if the floppy were the only interrupting device in the system; with a few such devices the situation becomes hopeless. Even worse, a typical hard disk transfers a byte every 2μs, completely beyond the capability of programmed I/O. Devices like disks and tapes (not to mention real-time signals and data) can't stop in mid-stream, so a method must be provided for reliably fast response and high overall byte transfer rates. Even with peripherals with low average data transfer rates, there are sometimes requirements for short *latency* time, the time from initial request to actual movement of data.

The solution to these problems is direct memory access, or DMA, a method for direct communication from peripheral to memory. In some microcomputers (e.g., the IBM PC) the communication is actually handled by the CPU hardware, but that doesn't really matter. The important point is that no programming is involved in the actual transfer of data; bytes are moved between memory and peripheral via the bus, without program intervention. The only effect on the executing program is some slowing down of execution time, because DMA activity "steals" bus cycles that would otherwise be used to access memory for program execution. DMA usually involves more hardware complexity in the interface itself, and it should not be used unless necessary. However, it is useful to know what can be done, so we will describe briefly what you need to make a DMA interface. As with interrupts, the IBM PC designers streamlined their DMA protocol; a "DMA controller" chip on the motherboard does the hard work for you, making a DMA interface relatively straightforward. In general, though, DMA interfaces tend to be machine-dependent and complicated. We'll first explain the more usual "bus mastership" method of DMA, then the PC's simplified DMA protocol.

A typical DMA protocol

In DMA transfers, the peripheral requests access to the bus via special "bus request" lines (prioritized like the IRQ lines) that are part of the bus. The CPU gives permission and releases control of address, data, and strobing lines. The peripheral then asserts memory addresses onto the bus and either sends or receives data, one byte at a time, according to the strobes it asserts; in other words, it takes over the bus (it becomes "bus master") and acts like a CPU, directing data transfers. The DMA bus master is responsible for generating addresses (usually a block of successive addresses, generated with a binary counter) and keeping track of the number of bytes moved. The usual way to do this is to have a byte counter and an address counter in the interface. These are initially loaded from the CPU, via programmed I/O, to set up the DMA transfer desired. On command from the CPU (via a command bit, written with programmed I/O), the interface makes its DMA request and begins to move its data. It may release the bus between each byte (allowing the CPU to sneak a few instructions in), or it may take the more antisocial approach of keeping the bus for a block of transfers. When all transfers are complete, it releases the bus for the last time and notifies the program that it is finished by setting a status bit and requesting an interrupt, whereupon the CPU can decide what to do next.

Getting data or programs from disk is a common example of DMA transfer: The executing program asks for some "file" by name; the "operating system" (more about this soon) translates this into a set of programmed data `OUT` commands to the disk interface's control (or "command") register, byte count register, and address register (specifying where to go on the disk, how many bytes to read, and where to put them in memory). Then the disk interface finds the right place on the disk, makes a

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

DMA request, and begins moving blocks of data to the specified place in memory. When it's done, it sets bits in its status register to signify completion and then makes an interrupt. The CPU, which has meanwhile been executing other instructions (or possibly just waiting for data from the disk), responds to the interrupt, finds out from the status register of the disk interface that the data are now in memory, and then goes on to the next task. Thus, programmed I/O to the interface (the simplest kind of I/O) was used to set up the DMA transfer, DMA itself (stealing bus cycles from the CPU) was used for rapid transfer of data, and an interrupt was used to let the computer know the task was done. This sort of I/O hierarchy is extremely common, especially with mass-storage devices; you can expect maximum DMA transfer rates of one to ten million words per second on a typical microcomputer bus.

□ *DMA on the IBM PC*

The IBM PC, which is basically a simple microcomputer, has a simpler DMA protocol. The motherboard has a DMA controller (the Intel 8237) with built-in address and byte counters, along with the logic to disable the CPU and take over the bus. A peripheral that wants to do DMA, therefore, doesn't have to generate addresses and drive the bus. Instead, it signals the controller (via one of the three DRQ1–DRQ3 "DMA request" lines), which in turn responds by returning the corresponding DACK0–3′ ("DMA acknowledge"). The controller then controls the transfers, asserting address and strobing lines, with the peripheral asserting (or receiving) data to (or from) memory. In this whole process the memory sees nothing unusual going on, since addresses and memory strobes (MEMW′ or MEMR′), normally supplied by the CPU, are supplied by the 8237 controller, and if it's DMA *to* memory, data are supplied by the peripheral. The peripheral, on the other hand, knows something special is happening since it requested DMA access (and received confirmation via DACK′); so when the DMA controller asserts IOR′ (or IOW′), the peripheral supplies (or accepts) successive bytes. You might wonder why some innocent bystander peripheral doesn't get hurt in the DMA process, since both I/O strobes and addresses are being asserted, whereas the addresses are in fact the *memory* addresses that go with the memory strobes MEMW′ or MEMR′ asserted by the controller; they have nothing to do with I/O port addresses. The secret is our old friend AEN, specifically added to the bus just to solve this problem. AEN is asserted HIGH during DMA transfers, and all I/O port addressing must be qualified by ANDing with AEN LOW to prevent spurious responses to DMA memory addresses.

Even with the use of a separate controller chip, you still have to set up the starting address, byte count, and direction for the impending DMA transfer. These data go to the 8237, which is obliging, having a set of registers that you write (via programmed I/O) from the CPU. It's pretty straightforward (see Eggebrecht's book for clear guidance), except that, as with most peripheral LSI chips, there is a confusing variety of choices of "modes" (single transfer, block transfer, etc.). Luckily, the PC is sufficiently primitive that you're allowed to use only "single transfer," which transfers only one byte per DRQ request. If you insist on transferring a whole block of data by holding DRQ high, the 8237 releases the bus for one CPU cycle between each DMA cycle; that keeps the computer alive, even if you have a greedy peripheral that tries to hog the bus. The standard PC has a rather modest DMA capacity, about 2μs per byte transferred. As with interrupts, the PC is sparse on DMA channels: Three

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

TABLE 10.1. IBM PC BUS SIGNALS

Signal name	Number	Active	Type[a]	Direction CPU↔I/O	Pin #	Function
A0–A19	20	H	2S	→	A31-A12	address (A0-A15 for I/O)
D0–D7	8	H	3S	↔	A9-A2	data
IOR'	1	L	2S	→	B14	I/O read strobe
IOW'	1	L	2S	→	B13	I/O write strobe
MEMR'	1	L	2S	→	B12	memory read strobe
MEMW'	1	L	2S	→	B11	memory write strobe
AEN	1	H	2S	→	A11	DMA address signal
IRQ2–IRQ7	6	↑	2S	←	B4,B25-B21	interrupt request
RESET DRV	1	H	2S	→	B2	power-on reset
DRQ1–DRQ3	3	H	2S	←	B18,B6,B16	DMA request
DACK0'–DACK3'	4	L	2S	→	B19,B17,B26,B15	DMA acknowledge
ALE	1	H	2S	→	B28	"address latch enable"
CLK	1	–	2S	→	B20	CPU clock (4.77MHz)
I/O CH CK'	1	L	OC	←	A1	I/O error - makes NMI
I/O CH RDY	1	H	OC	←	A10	pull LOW for wait states
OSC	1	–	2S	→	B30	14.31818MHz (3×CPU clk)
T/C	1	H	2S	→	B27	DMA terminal count
GND	3	–	PS	→	B1,B10,B31	signal & power gnd
+5V DC	2	–	PS	→	B3,B29	+5V supply
+12V DC	1	–	PS	→	B9	+12V supply
–5V DC	1	–	PS	→	B5	-5V supply
–12V DC	1	–	PS	→	B7	-12V supply

[a] OC - open-collector; PS - power supply; 2S - 2-state (totem-pole); 3S - 3-state.

channels (DRQ1–DRQ3) are accessible on the I/O bus (DRQ0 is already used internally to refresh dynamic memory): DRQ1 is used for hard disk, and DRQ2 is used for floppy disk. That leaves DRQ3 for everything else.

10.13 Summary of the IBM PC's bus signals

Through our examples – programmed I/O, interrupts, and DMA – we've seen most of the bus signals that go to the card slots in the IBM PC. Table 10.1 (and Fig. 10.14) lists the full bus, with pin connections. For completeness we'll summarize them all here, beginning with the ones we've already met.

A0–A19

Address bus. Two-state, output only, active-HIGH. All 20 lines are used to address memory (with MEMR′ and MEMW′ as strobes, analogous to IOR′ and IOW′), but only the 16 least significant lines are used during I/O access

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

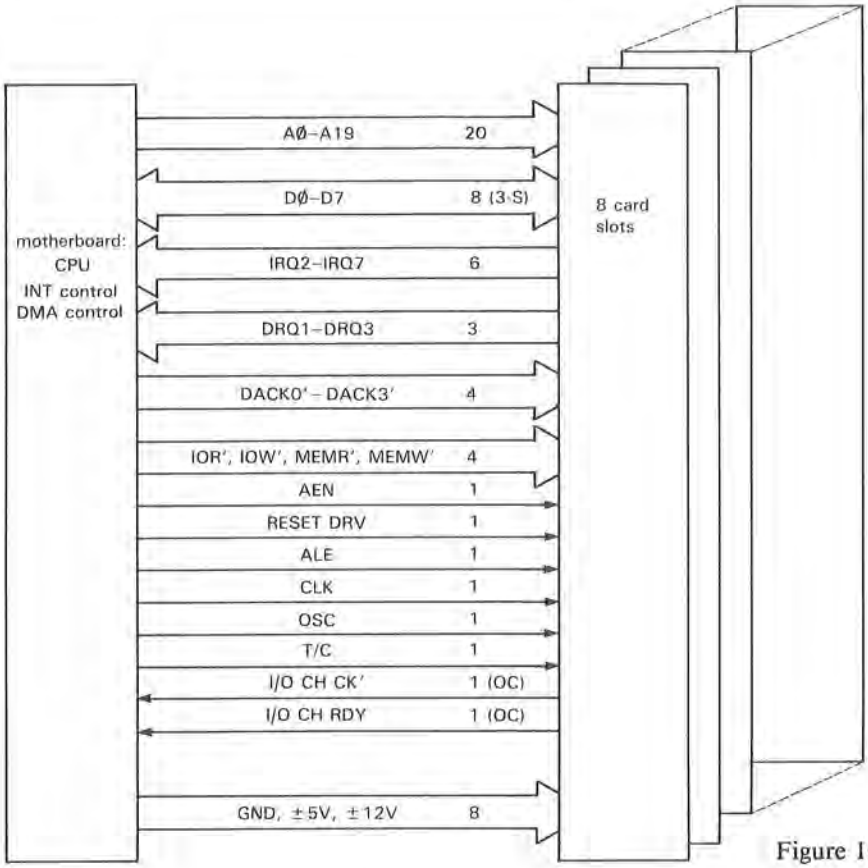


Figure 10.14. IBM PC bus.

(64K port addresses); I/O devices should qualify address with AEN LOW. Important note: I/O on the motherboard only looks at A0–A9, and uses 000_H–$1FF_H$; so external I/O must have its low ten bits in the range 200_H–$3FF_H$. You can be clever, though, by roosting in an unused 10-bit I/O address, then using the top 6 bits to create 64 I/O port addresses.

D0–D7

Data bus. Three-state, bidirectional, active-HIGH. Asserted by CPU during memory or I/O write; asserted by memory during memory read or DMA from memory; asserted by I/O port during I/O read or DMA to memory.

IOR′, IOW′, MEMR′, MEMW′

Data strobes. Two-state, output only, active-LOW. Asserted by CPU during read or write. On writes, data should be latched on trailing (rising) edge, qualified by address; on reads, data should be asserted coincident with strobe, qualified by address.

AEN

Address enable. Two-state, output only, active-HIGH. Asserted by CPU during DMA cycles. I/O ports must not respond with normal address decoding to IOR′ and IOW′; instead, I/O port that received DACK uses IOR′ or IOW′ to strobe DMA data bytes.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

IRQ2–IRQ7

Interrupt request. Two-state, input only, rising-edge-triggered. Asserted by interrupting device. Prioritized, with IRQ2 highest, IRQ7 lowest. Maskable in 8259 interrupt controller, via CPU write to port 21_H. Each IRQ level can be used by only one device at a time.

RESET DRV

Reset driver. Two-state, output only, active-HIGH. Asserted by CPU during power-on. Used to initialize I/O devices to known start-up state.

DRQ1–DRQ3

DMA request. Two-state, input only, active-HIGH. Asserted by I/O device requesting DMA channel. Prioritized, with DRQ1 highest, DRQ3 lowest. Acknowledged by DACK1′-DACK3′.

DACK0′–DACK3′

DMA acknowledge. Two-state, output only, active-LOW. Asserted by CPU (DMA controller) to indicate grant of corresponding DMA request.

ALE

Address latch enable. Two-state, output only, active-HIGH. The 8088 used a multiplexed data/address bus, and this signal corresponds to the 8088's strobing signal, used by latches on the motherboard to latch the address. Can be used to signal beginning of a CPU cycle; usually ignored in I/O design.

CLK

Clock. Two-state, output only. This is the CPU's clocking signal; it's asymmetrical, 1/3 HIGH and 2/3 LOW. The original PC used a 4.77MHz clock, but higher speeds are common. CLK is used to synchronize wait-state requests (via I/O CH RDY), in order to stretch an I/O cycle for slow devices.

OSC

Oscillator. Two-state, output only. This is a 14.31818MHz square wave, which can be used (when divided by 4) as a color-burst oscillator for color display.

T/C

Terminal count. Two-state, output only, active-HIGH. Tells I/O port that a DMA block data transfer is complete. A DMA device must qualify it with DACK′ for the channel in use, since T/C is asserted when any of the DMA channels finishes a block transfer.

I/O CH CK′

I/O channel check. Open-collector, input only, active-LOW. Generates highest-priority interrupt (NMI, "nonmaskable interrupt"); used to signal error condition from some peripheral. CPU figures out who's in trouble by device polling (Section 10.11); each peripheral that can assert I/O CH CK′ must therefore have a status bit that can be read by the CPU.

I/O CH RDY

I/O channel ready. Open-collector, input only, active-HIGH. Generates "wait states" if disasserted (i.e., pulled LOW) before the second CLK rising edge of a processor cycle (normally 4 CLKs). Used to extend bus cycle for slow I/O or memory.

GND, +5VDC, −5VDC, +12VDC, −12VDC

Ground and dc supplies. Regulated dc voltages that are bused for use by peripheral interface cards. Check the specifications of your computer for power limitations, which are machine-dependent. Generally speaking, there should be enough power to run anything you can fit into the I/O slots.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

□ 10.14 Synchronous versus asynchronous bus communication

The data IN/OUT protocol we described earlier constitutes a *synchronous* exchange of data; data are asserted onto or retrieved from the bus synchronously with strobing signals generated in the CPU (or DMA controller). Such a scheme has the virtue of simplicity, but it does open the possibility of problems with long buses, since the long propagation delays you get mean that data may not be asserted soon enough during a data `IN` operation for reliable transmission. In fact, with a synchronous bus the device sending the data never even knows if it was received! This sounds like a serious disadvantage, but in reality computer systems with synchronous buses seem to work just fine.

The alternative is an *asynchronous* bus, in which a data `IN` transfer, for example, goes something like this: The CPU asserts the port ADDRESS and a *level* (not a pulse) on a strobing line (call it IOR′, as before) that signifies data `IN` from the addressed device. The addressed device then asserts the DATA and a level signifying that the DATA is valid (call it DTACK′, "data transfer acknowledged"). When the CPU sees DTACK, it latches the DATA and then releases its IOR′ level. When the interface sees the IOR′ line go HIGH, it releases the DTACK′ and DATA lines. In other words, the CPU says "Give me data." The peripheral then says "OK, here it is." The CPU then says "OK, got it." And the peripheral finally says "Great! I'll go back to sleep again." This is sometimes referred to as "interlocked communication," or "handshaking."

Asynchronous bus protocol allows long buses and gives the communicating devices assurance that data are being moved. If a remote device is switched off, the CPU will know about it! Actually, that information is available via status registers with any kind of bus, and the chief advantage of asynchronous protocol is the flexibility of using any length of bus, bought at the expense of slightly greater hardware complexity.

There are situations where you want to attach relatively slow interface ICs to a bus; an example is slow-access ROM, or even RAM. All buses provide some way to stretch a bus cycle: With an asynchronous bus it's automatic, since the bus cycle goes on until the DTACK′ handshake is returned. With synchronous buses there is always some sort of HOLD′ line (it's called I/O CH RDY on the PC) to create wait states, effectively stretching the strobes and thereby delaying the end of the cycle. The overall bus cycle is always lengthened by an integral number of CPU clock cycles; that is the number of "wait states" inserted. For example, a standard IBM PC has a clock frequency of 4.77MHz (period of 210ns), and a standard memory access bus cycle is 4 clock periods (840ns). If I/O CH RDY is brought LOW before the second rising edge of CLK during a memory access, and brought HIGH again before the third, one wait state is generated, stretching the bus cycle (and MEMW′ or MEMR′) to 5 clocks (1050ns). If you hold I/O CH RDY LOW for additional clocks, you get additional wait states, up to a maximum of 10 clock periods.

Now we can reveal a well-kept secret about synchronous versus asynchronous buses: In reality, all single-processor (or, more precisely, single-bus-master) microcomputer buses are really synchronous, because all timing is slaved to a single CPU oscillator (like the 4.77MHz clock signal for the original IBM PC). Thus, if a peripheral device delays its handshake on an "asynchronous" bus, the cycle is always stretched by an integral number of CPU clocks. The distinction that is usually called

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

synchronous versus asynchronous is really this: On an "asynchronous" bus, wait states are inserted by default unless a wired-OR line (DTACK′) is asserted LOW, whereas on a "synchronous" bus the default bus cycle has no wait states, which are generated only if a wired-OR line (HOLD′) is asserted LOW. The difference is more than semantic – you can't have a physically long bus with "synchronous" protocol, because the HOLD′ signal gets back too late to stretch the cycle, whereas with an "asynchronous" bus the CPU won't terminate the bus cycle without your permission (DTACK′). In our usually humble manner, we offer the following suggestive terminology to clear up this confusion: If the bus makes wait states by default ("asynchronous"), call it *default-wait*; if the bus makes wait states only when you ask ("synchronous"), call it *request-wait*. The IBM PC is request-wait, whereas the VME bus (see below) is default-wait.

This whole bus situation becomes more complicated with multiprocessor systems, in which bus mastership changes hands. A synchronous bus with multiple masters requires all masters to use a single clock, whereas an asynchronous bus permits different clock rates. Luckily for you, multiprocessor systems are beyond the scope of this book!

A possible point of confusion: You don't add wait states because you have a slow *peripheral* (a printer, for example); you do it only because you have a slow IC (ROM, say, with 250ns access time, or a slow LSI peripheral chip), which cannot latch (or produce) the data within the normal bus access time. A slow peripheral is usually hopelessly slow (*milli*seconds, not nanoseconds); the solution is to send (or receive) a byte at full bus speed, latching it in a byte-wide register chip, then wait for an interrupt (or possibly a status flag) before doing another full-speed transfer.

10.15 Other microcomputer buses

We chose the IBM PC to illustrate microcomputer bus architecture – bus signals, memory and programmed I/O, interrupts, and DMA. Since the PC is widely copied and widely used in engineering and data acquisition/control, this is a good illustrative choice for an electronics book. Furthermore, the PC bus is extraordinarily simple and easy to use.

However, simplicity has its costs. The original PC bus is seriously limited in important ways, some of which we have already mentioned (e.g., scarcity of interrupt and DMA channels). Even more seriously, by today's standards the PC bus has too little address space (20 bits, only 640K usable), too narrow a data path (8 bits), insufficient data transfer rate (1.2Mbyte/s, max), and no provision for multiple bus masters. IBM has evolved improved buses in subsequent PC generations, first the PC/AT (a compatible enhancement of the original PC), then the all-new (= incompatible!) "microchannel" bus of the PS/2 series. Outside the IBM world there are competing buses peculiar to a given manufacturer (e.g., DEC's Q-bus and VAXBI bus), and generic buses (Multibus, NuBus, VME bus). Let's take a quick tour of the computer buses listed in Table 10.2.

PC/AT and Micro Channel

IBM's PC/AT (for "Advanced Technology") was introduced in 1984, and discontinued in 1987, at the peak of its popularity, to make way for IBM's PS/2 series of "clone-killer" computers, which use the improved "Micro Channel" bus. [The PC/AT continued to thrive, however, since the clone manufacturers (and many buyers) initially ignored IBM's newer machines, whose advanced features required nonexistent software.] The PC/AT uses the 80286 CPU and a compatible

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

TABLE 10.2. COMPUTER BUSES

Bus	Raw bandwidth (Mbyte/s)	Data width	Address width	Block xfer?	MUXed data/adr?	Multimaster?	Sync/Async	IRQ lines[a]	Drivers	Connector[b]	Comments
STD bus		8	16	–	–	–	S	1	TTL	CE	controller-type applications
PC/XT	1.2	8	20	–	–	–	S	5E	TTL	CE	original IBM PC & compatibles
PC/AT	5.3	8,16	20,24	–	–	(c)	S	10E	TTL	CE	accepts PC/XT cards
EISA	33	8,16,32	20,24,32	•	–	•	S	11P	TTL	CE	enhanced PC/AT; auto-configure
MicroChannel	20	8,16,(32)	24,(32)	•	–	•	A	11	TTL	CE	IBM PS/2; auto-configure
Q-bus	2	16	22	•	•	•	A	4	(d)	CE	LSI-11, µVAX-I,II; daisy-chained IACK
Multibus I	10	8,16	20,24	–	–	•	A	8	TTL	CE	Intel; SUN-I and others
CAMAC	3	24	9	•	–	–	S	L	TTL/OC	CE	data acquisition & control bus
VAX BI	13.3	8,16,24,32	32	•	•	•	S	4	TTL	ZIF	VAX 780, 8600 series; parity
Multibus II	40	8,16,24,32	16,32	•	•	•	S	M	TTL	DIN	parity; 40MB/s for blk xfer, 20M otherwise
NuBus	40	32	32	•	•	•	S	M	TTL	DIN	Macintosh II adds 1 dedicated INT per slot;""
VME	40	8,16,32	16,24,32	•	–	•	A	7	TTL	DIN	daisy-chained IACK; SUN-3
Futurebus	120					•	A	–	(d)		
Fastbus	160	32	32	•	•	•	A	M	ECL	H	communication across many crates

[a] E - edge-sensitive; L - LAM ("look at me"); M - "int" via bus mastership; P - programmable edge- or level-sensitive interrupts.
[b] CE - card-edge; DIN - 2-part "Eurocard" 96-pin connector; H - high density 2-part conn. [c] almost. [d] National Semi special.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

enhancement of the original PC bus: An additional (and optional) connector carries an extra 8 bits of data, 4 bits of address, and 5 additional IRQ lines (edge-triggered, as before). The resulting 16-bit data path and higher CPU clock speed raise the maximum bandwidth to 5.3Mbyte/s, which, with the additional address space and interrupts, makes the PC/AT a serious microcomputer. The PC/AT bus (sometimes called Industry Standard Architecture, or ISA) even supports multiple bus masters, though its abilities here are limited. Cards that work on the original PC bus will work in the PC/AT (if they are fast enough), because you can ignore the bus enhancement carried on the extra connector; in that case, of course, you revert to an 8-bit data path and 20-bit address space. AT-compatible computers generally run their I/O bus at higher speeds, which can create additional timing problems with older plug-in boards.

The Micro Channel bus was first used in IBM's PS/2 series of second-generation personal computers, introduced in 1987. It allows for data and address paths up to 32 bits wide (in the high-end 80386-based machines), 11 levels of shared (level-sensitive) interrupts, multiple bus masters, and asynchronous protocol. Cards that plug into the Micro Channel don't have hardwired I/O port addresses; instead, the CPU assigns an address (and other configuration choices) at start-up, based on information it reads from ROM on the card. This pleasant feature means that you don't have to set little switches on each card, and you don't have to worry about cards using overlapping address space. Micro Channel cards have tight dimensional tolerances, owing to the daring use of 0.050 inch spacing between pads on the edge connectors.

EISA

The Extended Industry Standard Architecture (EISA) is the clone-makers' answer to the Micro Channel. It was introduced in 1988 by nine manufacturers of AT-compatible computers. By adding an extra connector to the AT bus, the EISA's designers implemented many of the desirable features of the Micro Channel, *while maintaining compatibility with existing AT plug-in cards.* Thus, you can plug standard AT boards into EISA and get normal AT functionality. Moreover, when used with boards designed specifically for it, the EISA supports 32-bit data transfers (with peak transfer rates of 33Mbyte/s), 32-bit memory addressing, multiple bus masters, programmable level or edge-triggered interrupts, and automatic board configuration.

Multibus I and II

Originally introduced by Intel, the Multibus formats have found their way into many computers. The original Multibus I is a capable bus with 16-bit data path and 24-bit address space, and it allows multiple bus masters. Multibus II is intended for high-performance multiprocessor systems, with 32-bit data and address paths, parity checking, distributed arbitration, and message-passing protocols. It uses a synchronous 10MHz clock and can transfer up to 40Mbyte/s to sequential addresses in "block transfer" mode. In common with some other large buses (NuBus, Fastbus), Multibus II saves pins by multiplexing data and address on a common set of 32 lines. It also uses a 96-pin card-mounted DIN connector, rather than the simple gold-plated "card-edge" connector: By using a well-designed card-mounted ("2-part") connector, you get better reliability and a connection system that is insensitive to card warp and rough handling.

Although Multibus II seems to have all the advantages, its flexibility can make your work hard. For example, it doesn't have conventional interrupts; instead, you

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

"interrupt" by requesting bus mastership, then sending a message to the processor you want to interrupt! For simple systems, the simpler Multibus I (or some other simple bus) may be better.

NuBus

This is another high-performance synchronous multiprocessor bus with multiplexed 32-bit data and address paths, DIN connectors, and high data-transfer rates (to 40Mbyte/s in "block transfer" mode). In common with Multibus II, it forces you to go through a bus mastership protocol to interrupt. It is used in the high-end Macintosh computers where, thankfully, Apple added a dedicated interrupt line to each slot. Thus, each card slot has a unique vector assigned; the corresponding software handler knows which card interrupted without polling and has to poll only if that card has more than one possible interrupting device.

VME bus

The VME bus, like NuBus and Multibus II, is intended for multiprocessor 32-bit systems. Unlike those buses, however, it does not use multiplexed data/address lines. Nor does it use a synchronous master clock, preferring asynchronous protocol; this lets you mix processors of varying speeds without pain. The VME bus also implements conventional multilevel IRQ-type interrupts, with full interrupt acknowledgment (complete with daisy-chained INTP line). The VME bus is often viewed as an alternative to Multibus; for example, the original Sun computer from Sun Microsystems used Multibus, whereas their more recent Sun 2 and Sun 3 use VME. VME bus and Multibus II are currently slugging it out in the trade press, cheered on by Motorola and Intel, complete with diatribes and name-calling.

Fastbus and Futurebus

These are *very* high performance buses, with blazing speed. The Fastbus uses large cards (14×16 inches), ECL drivers, and arbitration protocols to support multiple bus masters. In fact, bus communication is one of its strong points, with capability for sophisticated "geographic" communication beyond the immediate crate of cards.

Q-bus and VAXBI

These are proprietary buses used in DEC computers. The Q-bus, used in the LSI-11 and early MicroVAX computers, evolved from DEC's original PDP-11 "Unibus." It supports 16-bit data and 22-bit addressing, asynchronous protocol with multiple masters, and multilevel IRQ-type interrupts. The VAXBI is a high-performance multiplexed 32-bit data/address bus used in the larger VAX 8600-series machines.

10.16 Connecting peripherals to the computer

Interfaces are usually built on printed-circuit cards or Wire-Wrap cards (see Chapter 12) designed to plug into the microcomputer's card slots. Microcomputers generally contain a number of unused slots for just this purpose (or they can be "expanded" to accommodate extra cards), with power-supply voltages and bus signals distributed to the card slots. Some machines use a "proprietary" bus (e.g., the IBM PC), others use a standardized microcomputer bus (e.g., the Sun 3 workstation, which rides on the VME bus), and some have no bus slots at all (e.g., the original Macintosh). Each bus has a standard card size (or sizes), ranging from the small 3.2×11.5 inch IBM PS/2 cards to the giant 14.4×15.9 inch Fastbus cards. Depending on the particular bus, each card has 50 to 300 connections along one

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

edge, either in the form of a set of gold-plated printed-circuit edge connections or as a set of multipin connectors that are soldered to the board; the latter are known as "two-part" connectors and are generally more reliable than PCB edge connectors.

Commercially available interfaces for common tasks (disk, graphics, communications, analog I/O) are usually built on cards that plug into unused bus slots. Cables then go from connectors on the interface card to the peripheral (if any); if the interface involves many inputs or outputs (e.g., a digital logic analyzer), it may connect by cable to an external panel or box where there is more room for connectors (and additional circuitry). In either case it is common to use flat *ribbon cable*, with some care being taken to prevent cross-coupling of strobing signals with data. One method is to ground every other wire in the ribbon; another technique uses ribbon cable bonded to a flexible metal groundplane to reduce inductance and coupling, at the same time maintaining a nearly constant cable impedance. In both cases you can get nice multipin "mass-termination" connectors that attach to the cable with one simple crimping operation; check the catalogs of AMP, Berg, T& B Ansley, 3M, etc. An alternative to ribbon cable is a cable made of multiple twisted-pairs, each pair consisting of one signal line and one ground wire. Twisted-pair cable is available in many configurations, including a nifty ribbonlike flat cable (Allied/Spectra "Twist-'n-flat") in which there is a flat untwisted region every 20 inches for easy connection to crimp-on connectors of the type used for ordinary ribbon cable. Because of the strobed data-transfer protocol used between an interface card and the device it controls, it generally isn't necessary to use signal/ground pairs for *all* signal lines, just for the synchronizing pulses and other strobing or enabling lines. Suitable terminations and driver/receiver combinations should be used for long lines, as described in Section 9.14.

Custom interfaces are best handled in the same way, either laying out printed-circuit boards or using one of the general-purpose interfacing cards available commercially from companies such as Douglas, Electronic Solutions, and Vector. These blank cards have places for ICs and other components (including mass-termination connectors for external cables), and they come in solder and Wire-Wrap styles (more in Chapter 12). Some of them include built-in circuitry to handle bus communication, including interrupts and even DMA.

In some cases the best plan may be to build an interface that resides partly in the computer and partly outside, as suggested in Figure 10.15. In such cases the interface circuitry that goes in the computer will probably be a simple parallel input/output port, either a commercially available parallel port card or a custom card you design. The cable connecting the two parts of the interface is simple and could use one of the high-performance driver/receiver combinations we discussed in Section 9.14 if high-speed communication over long cable runs is needed (for example, RS-422, or the differential current-sinking 75S110 IC, or even fiber optics). This sort of scheme may be particularly useful for interfaces that handle low-level analog signals, since the noise-susceptible linear circuitry can be kept away from the general roar of digital interference present in the computer (and close to its analog signal source); this also allows you to pay careful attention to maintaining clean analog signal ground lines.

SCSI, IEEE-488, and other interfaces

There are literally hundreds of plug-in boards, performing an incredible variety of functions, available for common buses

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

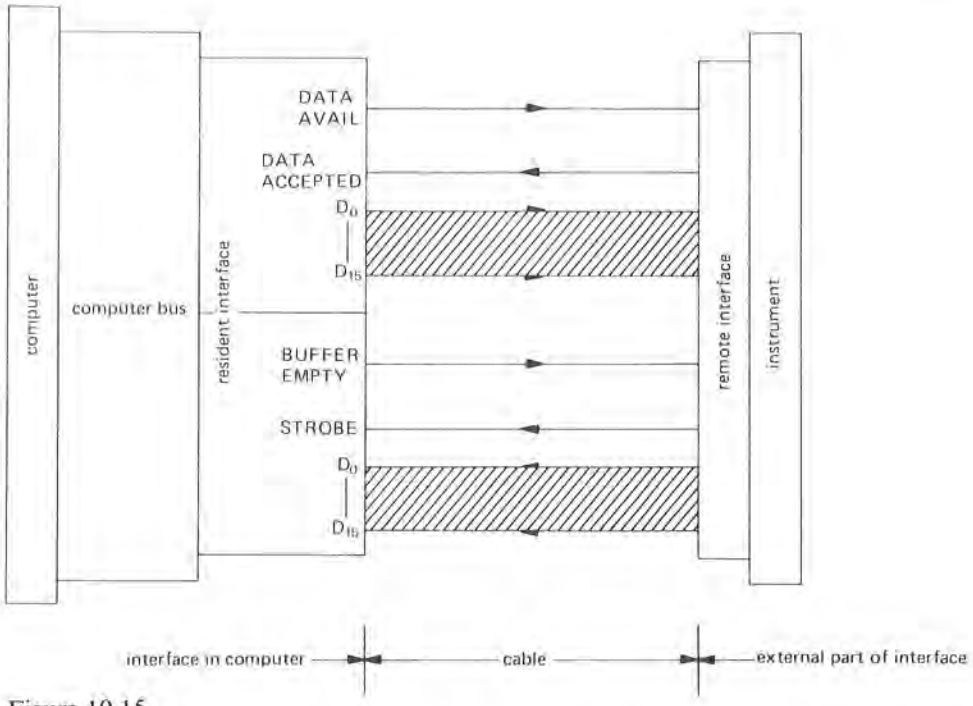


Figure 10.15

such as the IBM PC, Multibus, VME, and Q-bus. These are so inexpensive and easy to use that you should always check out the possibility that either (a) the board you are designing already exists or (b) you can use a simple parallel-port card as a computer-resident part of your interface, as described in the last section. Another possibility is to use a standard built-in "Centronics" parallel port, or an RS-232 serial port (see Sections 10.19 and 10.20), to couple a custom gadget to a microcomputer. This scheme has the virtue of making your gadget portable, even to a microcomputer with a different bus (or no bus at all!), since these ports look the same on all computers. Such a gadget for connection to a serial port will probably have its own microprocessor, so you might tend to think of it as a computer rather than a peripheral. But, as we'll explain in the next chapter, it's fun and easy (and cheap) to build little microprocessor-based instruments; there's really no good reason to treat a microprocessor differently from any other LSI chip, which you wouldn't hesitate to include in a custom circuit.

Following this idea a step further, there are a few "cable interface" standards that have become popular recently. They have names like SCSI ("small computer system interface"), IPI ("intelligent peripherals interface"), ESDI ("enhanced small-disk interface"), and IEEE-488 (also known as HPIB and GPIB, "general-purpose interface bus"). SCSI (pronounced "skuzzy") in particular is now standard equipment on many microcomputers, thanks to the availability of disks and other peripherals that connect directly to a SCSI port. You can get add-in SCSI interface cards for computers without built-in SCSI ports. SCSI is actually a descendant of SASI (Shugart Associates System Interface, a simple parallel bus that Shugart cooked up for their hard-disk drives) and in its simplest

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

incarnation is a byte-wide bidirectional parallel protocol with handshaking. It allows several modes, including synchronous or asynchronous transfer, with single-ended or differential drivers; although it originally was used to connect a single CPU to a single disk, it can be used to couple multiple CPUs to multiple disks. Typical transfer rates are 1.5Mbyte/s (asynch) and 4Mbyte/s (synch); asynchronous protocol is slower because the handshakes are bouncing back and forth during each transfer. SCSI can go 20 feet with single-ended drivers, and 80 feet with differential drivers.

The IEEE-488 bus, originated by Hewlett-Packard as the HPIB, was designed for connecting laboratory instruments to computers. There is a full protocol for connecting multiple instruments on a bus, with phrases like "talkers" and "listeners." IEEE-488 is firmly entrenched in the instrumentation field, with manufacturers like Hewlett-Packard, Keithley, Philips/Fluke, Tektronix, and Wavetek offering it on most of their instrumentation. You can get 488 interfaces for nearly all microcomputers. We'll have more to say on SCSI and IEEE-488 in Section 10.20.

SOFTWARE SYSTEM CONCEPTS

In this section we will discuss some general aspects of small-computer programming, since a knowledge of computer interfacing is of limited value without an understanding of the hierarchy of programs that actually make the computer come to life. In particular, we would like to discuss the important areas of programming, operating systems, files, and use of memory. It is easy to get carried away admiring the beauty of computer hardware and underestimate the importance of good software. Software is what makes the computer fly, and a good operating system and package of "utilities" can make all the difference.

Following our discussion of software and systems, we will end the chapter with a section on communications concepts, in particular the standardized RS-232 serial ASCII protocol, the "Centronics" parallel port, other parallel communications schemes (SCSI, IPI, GPIB), and local area networks.

10.17 Programming

Assembly language

As we mentioned earlier in the chapter, the computer's CPU recognizes certain groupings of bits as valid instructions, which it then acts upon. It is extremely rare to program directly in this binary machine language. Instead, you write programs in a mnemonic assembly language (like our interfacing examples earlier), which a program called an *assembler* converts into relocatable machine code. Assembly language is very close to machine language; each instruction is converted directly into one line or a few lines of machine code (the first line is usually the operation code, with the extra lines generally completing the addressing of the variables, or furnishing constants). Assembly-language programming produces the most efficient code and allows you to get at flags and registers that are inaccessible from higher-level languages. But it is tedious programming, as the examples illustrate, and for most computing jobs (especially those involving plenty of numerical computation) it pays to use a compiled or interpreted high-level language, such as C or FORTRAN, with calls to assembly-language routines only where necessary.

Compilers and interpreters

C, FORTRAN, PASCAL, and BASIC are popular examples of high-level languages. You write a program with algebraic types of commands, for instance

$$x = (-b + sqrt(b * b - 4 * a * c))/(2 * a)$$

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

and with control structures like *if ... elseif ... else*, *for ...* , *while ...* , and *do ...* . You don't have to shuttle your little bytes hither and thither, or worry about addressing, saving registers, etc.; you just declare variables and arrays by type and size and use them in arithmetic or logical expressions. Everything is chocolate-coated.

This is called *source* code, from which there are two routes to a running program. Languages like C and FORTRAN are *compiled*, a process in which a language compiler converts the source-code statements to assembly code; from there it's business as usual, with the assembler converting that intermediate assembly language into machine language. Languages like BASIC and APL have traditionally been *interpreted*; instead of compiling an assembly-language program from the source program, an *interpreter* program "looks at" the statements and executes appropriate computer instructions.

In general, interpreted languages run much more slowly than compiled languages. However, since there's no compilation, assembly, or linking (discussed next), there's no delay after entering a program before it can run. Interpreters often include a simple editor, convenient for quick modification and retrial of a program you're debugging. Interpreted BASIC gained popularity in the early days of microcomputers, when hard disks were a rarity, since it ran entirely in memory; this contrasted with the tedious multipass compilation process. With today's fast disks and efficient compilers, there's not much to complain about. In fact, recent compilers have followed the lead of Borland's interpreted "Turbo Pascal" by providing a "total environment" in which you can hop around effortlessly between editor and running program: If there's a bug, the system puts you back into the editor, pointing to the bad statement; these compilers include debuggers, provision for making "libraries," and other pleasant features.

The current all-around favorite among heavy-duty programmers seems to be C, which combines the power of high-level languages with the beauty of structured languages and the bit-pushing flexibility of assembly code. However, FORTRAN still claims the lion's share of scientific computing.

Linkers and libraries

The assembler produces machine code (well, almost; it's actually called "relocatable machine code") from the assembly code produced by the compiler and from separate subroutines written in (or compiled to) assembly code. In addition, there are usually routines needed by particular commands in the high-level program. For example, a C program might need a math function like *sqrt*, or a host of I/O functions like *printf* or *fopen*. A program called a "linker" handles the bureaucratic nightmare of getting the appropriate subroutines (in relocatable form) from a "library," then rigging up all the linking jumps and addressing so the whole mess fits together in memory. It is the linker's job to put final numerical values into the memory references and variable addresses of the assembled code, and it can do this only when it knows which program calls which, and how long each program is. That's why the code produced by the assembler must be in relocatable form, as must the assembled subroutines that sit in the various libraries [there are usually several – a library of compiler functions, an I/O library, a math library, a library of system calls, and perhaps a home-grown (or store-bought) library of useful subroutines].

Editors and formatters

In prehistoric days (before 1970) you could find card-carrying computer programmers,

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

literally: You wrote your programs by hand on coding forms, then punched them (or paid someone else to punch them) onto those handsome "IBM cards" that had rows of numbers printed on pastel cardboard. Nowadays even toddlers know how to use computer editors, the universal program entry method. Old-timers (those over 30) can still remember the first awkward "interactive" computer editors, with which you could create and modify a text file that, for some reason, the editor never let you see much of. Don Lancaster teased us with his "TV Typewriter," a build-it-yourself project that let you display a line of text on a television. That's all it did. No editing, no storage, no nothing. Our joy was truly unbounded, therefore, when we first used "full-screen" editors.

A good editor (and they're all good, now) lets you type and correct as you go, search for words, change text, move blocks of text around, open multiple windows on multiple files, and write "macro" definitions that do complex manipulations. The screen should redraw quickly, even if you add text near the beginning of a large file. Very large files shouldn't slow things down.

A general-purpose editor doesn't know, or care, what you are writing; it could be a program, a sonnet, or a book. It just creates the text file according to your keyboard instructions. If the file consists of statements in a programming language, the compiler, interpreter, or assembler reads it directly. If, on the other hand, the file is text that you want to print, you have two choices: You can send it directly to a printer, or you can mark it up with formatting information and send it to a formatter program that tells the printer how it should be printed. A good text formatter takes care of margins and line justification, proportional spacing, changes of font, italics, boldface, underline, and so on. The editor and formatter are often combined, sometimes with a screen display showing what the printed page will look like (that's called WYSIWYG, pronounced "wizzywig": what you see is what you get), but more often with the screen display only partially faithful to the final page. The most advanced formatters are capable of typesetting mathematical and scientific formulas. For "camera-ready" quality, you do your printing on a typesetting machine, which exposes photographic paper or film directly; laser or LED printers offer quite good quality at moderate cost and high speed; "dot-matrix" impact printers are the cheapest, as the result shows.

Editor/formatters go by names like MacWrite, Manuscript, Microsoft Word, Sprint, and WordPerfect. Popular technical formatters (which do both text and equations) are TEX and Troff. One caution: When creating text (as opposed to programs), most editor/formatters insert unusual characters in the edited text stream, for example to indicate italics, or temporary end-of-line. These characters are unacceptable to compilers and assemblers. Thus, you've got to force the editor to run in a "vanilla" mode, in order to create unadorned source code that the compiler, etc., won't choke on.

Here's some free advice: (a) Find a good editor and stick with it, and (b) Don't try to persuade others that your editor is better than theirs.

10.18 Operating systems, files, and use of memory

Operating systems

As you might guess from the preceding discussion, you frequently want to run different programs at different times, trading data back and forth between them. For instance, in writing and running a program you begin by running the editor program, creating a text file from

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

the keyboard (good programmers never set pencil to paper, as far as we can tell). After temporarily storing the text file, you bring in the compiler program and compile the stored text file to form an assembly-language file. You store that, bring in the assembler, and produce a relocatable machine-language file from the stored assembly-language file. Finally, the linker combines the relocatable machine code with other assembled subroutines and library routines to produce the executable machine-language program, which (at last!) you run. For all these operations you need some sort of super program to juggle things around, getting programs from disk, putting them into memory, and transferring control to the relevant programs. In addition, it would be nice if each program didn't have to contain all the commands necessary to do disk reads and writes (including interrupt handling, loading of status and command registers, etc.), or, indeed, any of the other detailed data communications tasks.

These are some of the tasks of the *operating system*, a vast program that oversees the loading and running of user programs (the ones you write) and utility programs (editor, compiler, assembler, linker, debugger, etc.), as well as the handling of I/O and interrupts, and file creation and manipulation. The operating system includes a *monitor* for user interface (you tell it to run the editor, compile a program, or run a program) and many "system calls" that permit a running program to read or write a line of text from some device, find out the time of day, swap control to another program, let several multitasking "processes" share CPU time and communicate among themselves, bring in a program "overlay," etc. Good operating systems handle all the busywork of I/O handling, including "spooling" (the buffering of input or output data so that the program can run at the same time that data are being read or written to some device). When running under an operating system, a user program doesn't have to worry about interrupts; an interrupt is taken care of by the system, and it affects the running program only if it wants to take part in the handling of a particular device's interrupts. The whole business of successful "time sharing" (using one computer to handle many users at once), with the disk providing "virtual memory" for unlimited program size, is system programming at its finest.

Some popular microcomputer operating systems are MS-DOS (used on the IBM PC and its imitators), OS/2 (used on the PS/2, IBM's successor to the PC), UNIX (created at Bell Labs, widely used on VAX and 68000-based machines), MacOS, and VMS (company-supplied VAX operating system).

Files

The mass-storage medium in widest use currently is magnetic disk, either flexible ("floppy"), with contacting read/write heads, or rigid ("hard" disk, or "Winchester"), with flying heads. Typical storage capacities are in the range of 1Mbyte for floppies and 20–500Mbyte for small Winchesters. The data are organized into *files*. Text, user programs, utility programs (e.g., editor, assembler, compiler), libraries, etc., are all stored in similar ways, and all constitute files. Although the mass-storage medium is divided into physical blocks or sectors of well-defined size (512 bytes/sector is common), the files themselves may have any length. The operating system mercifully takes care of track/sector addressing, etc.; it gets the data you want, if you know the file name. There are all sorts of interesting details having to do with file organization that we don't have space to describe here. What is important is to understand

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

that all those programs (editor, compiler, etc., as well as user source text, compiled programs, and even data) reside on some mass-storage device as named files, and the system can get them for you (read the next subsection, however, on "ramdisk"). In the normal course of its duties, the system does enormous amounts of file handling.

Recent additions to the mass-storage stable are based on consumer electronics media and provide very high density storage in small packages: (a) Optical disks of the kind used in audio CD players store nearly a gigabyte, as prerecorded "read-only" memory, as *WORM* memory ("Write Once, Read Many"), or (as with magnetic media) as fully erasable read/write memory. (b) Videotape, in both VHS and 8mm formats, lets you store a gigabyte of read/write memory on inexpensive tape; the major drawback is the long access time. Both storage systems use sophisticated error-correction schemes to overcome errors due to media blemishes, etc., which are a minor nuisance in the original audio/video applications of these media, but would be devastating for data or program storage if uncorrected.

Use of memory

Files are stored in some mass-storage device, but a program must reside in memory while being executed. A simple stand-alone program of the sort we'll talk about in the next chapter can be loaded almost anywhere in memory. But in a microcomputer with an operating system there are special areas reserved for special functions. For example, the MS-DOS operating system itself, along with its command interpreter, disk buffers, stack, etc., is usually loaded at the bottom of memory, taking care to put its interrupt vectors in the specific locations in low memory that the CPU requires, while the portion of MS-DOS that is in ROM is located high in memory, above the portion of memory reserved for video display buffers. When operating under an operating system, the allocation of memory for user programs will be handled by the system. This is particularly important to understand if you intend to use DMA; in that case you have to let the system figure out where your data buffer wound up, and use that as the starting address for the DMA block transfer.

The situation is even more complicated if programs are being swapped in and out of memory, or moved around in memory. There may be several programs in memory simultaneously, sharing "time slices" of the CPU in a multitasking mode. To add to the complexity, most microcomputers use "memory mapping," in which *physical* memory addresses (what's actually on the bus lines) are mapped to different *logical* addresses (where your program thinks it is). If that isn't enough to confuse you, consider "virtual memory," a feature of advanced microcomputers in which your program is diced up into little "pages," any of which may or may not be in memory at any instant; the program "pages" them in and out in a crazy quilt of frenzied activity.

No discussion of memory use is complete without mentioning *ramdisk*, which can be invoked even on relatively simple machines, if they have enough memory. The basic idea is to make memory look like disk, from the operating-system point of view; you then load into this ramdisk memory the programs that you need frequently. This can be handy during program development, when you need to keep using the editor, compiler, assembler, and linker. With ramdisk, things move along quickly, since no actual disk access is required. It does have the hazard that you can lose all your work if the computer crashes, since files are not automatically saved on disk. A related concept is a disk *cache*, in which an area of RAM holds the results of recent disk accesses.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

Drivers

The computer world is rich with diversity – each month we see products using novel technologies in data storage (magnetic, optical), printers (laser, LED), networks, etc. Different hardware requires different controlling signals, with different timing requirements, etc. This would appear to create real programming problems, since publishing software designed for a dot-matrix printer, for example, would appear to be totally inappropriate for a laser typesetter.

The solution consists of software *drivers*, which are special programs designed to create a uniform programming interface to each particular piece of hardware. Thus, for example, the typesetting language TEX creates output in the form of *dvi* (device-independent) files; a printer driver (specific to the particular printer you are using) eats the *dvi* file and spits out the corresponding idiosyncratic printer codes to instruct the printer. TEX works with any printer, once you have the *dvi*-translating driver. The same sort of device independence goes for mass-storage devices such as disk drives, so that you can attach any of a variety of disks to UNIX, PC-type, or Macintosh computers.

Drivers are really part of the overall system software, and the average computer user is unaware of their operation. If you are designing new computer hardware, however, you will probably find yourself quickly becoming an expert on these essential software modules, since you will have to write your own drivers to make your hardware play with the rest of the team.

DATA COMMUNICATIONS CONCEPTS

A small computer system will usually be configured with some mass-storage devices, such as disks and tape, and some "hard-copy" or interactive devices, such as alphanumeric terminals, printers, plotters, etc. In addition, it may have a modem (modulator-demodulator) so that it can dial up other computers through ordinary telephone lines. Finally, local area networks (LANs) are becoming increasingly popular. With a LAN you can have access to files stored in other computers on the network, as well as the ability to share expensive resources (for example, large disks, tape drives, printers, and typesetters). In each case your CPU has to communicate data. Let's see how it works.

Incompatibility

In the dark "middle ages" of computers (say, up to 1975) the situation was pretty bleak. Each brand of computer had its own bus structure and interfacing protocol (not to mention programming language). You bought (or sometimes built) interfacing cards to fit the particular computer, with custom cables going from the interface to the peripheral itself. This general lack of compatibility extended to the peripherals themselves: You couldn't hook a tape drive to a disk interface, or a terminal to a plotter interface, etc. To make matters worse, the peripherals offered by different manufacturers generally used different signals and data-transfer conventions and were not "plug-compatible."

Compatibility

Some of this incompatibility was unavoidable, since to maximize performance different peripherals transfer their data to and from the interface differently. For example, a magnetic disk moves words in parallel byte-wide format for high speed, and the corresponding interface must use DMA transfer, as we explained earlier; by contrast, a keyboard terminal uses a standardized alphanumeric bit-serial format, with the interface using simpler interrupt-driven programmed I/O. Although some of this

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

incompatibility is still with us, the situation is vastly improved, with most of the industry standardizing on a few agreed-upon data communications standards. The introduction of the IBM PC defined a much-needed small-machine format and data bus, while nonproprietary high-performance buses like VME and Multibus became the backplane for a number of other computers. You can get interface cards for these buses (and others, like DEC's Q-bus) from many manufacturers, which simplifies things enormously. Even more important, the manufacturers of peripherals have agreed on a few standardized "cable interfaces." The most important of these are (a) RS-232 serial format, usually used with alphanumeric ASCII data, (b) Centronics' parallel printer format, (c) SCSI parallel bus, (d) IPI bus, and (e) IEEE-488 (GPIB) instrument bus. Let's take a look at these, and then finish the chapter with a brief description of two popular kinds of local area networks, Ethernet and token-ring networks.

10.19 Serial communication and ASCII

As mentioned earlier, alphanumeric communication between a computer and devices of moderate speed is most frequently done using the 7-bit ASCII code (American Standard Code for Information Interchange), with bit-serial transmission over a single line. Table 10.3 presents a listing of the 7-bit codes. Devices communicating via serial ASCII almost always send an 8th bit, but it is not part of the ASCII code; it is most often a hardware parity bit (sometimes odd parity, sometimes even, but most often set to 0 and ignored), but it is occasionally used as a "meta" shift key to generate an additional 128 characters, which may be Greek symbols, alternate fonts, etc. There are no standards for these extra symbols. (The 8th bit also gets used when you ship *binary* data via a serial connection; this doesn't always work, though, because serial data links are so used to getting rid of the 8th bit during ASCII transfer that they may not permit you to retain it as data.)

A few notes on the ASCII table. The upper-case alphabet begins at 41_H; setting bit 5 to a 1 generates the corresponding lower-case character. The ASCII value for a digit is just the digit plus 30_H. The first 32 ASCII characters are nonprinting, or "control" characters. Some of them are important enough to have earned their own keys on keyboards, for example CR (which may be labeled "return," since keyboards don't have carriages), BS ("backspace"), HT ("tab"), and ESC ("escape"). You can generate any control character (including the above) by holding down CTRL and typing the corresponding letter from the upper-case alphabet; for example, CR is CTRL-M (try it on your computer). The control characters are used to control printing or program execution, or they can be used by programs that otherwise expect to receive alphanumeric characters, e.g., text editors. Some other important control characters, besides the ones listed above, are NUL (null), a character of all zeros often used to delimit character strings; FF (form feed), used to begin a new page; ETX (end of text, affectionately called "control C"), which many operating systems interpret as a command to abort a running program; DC3 (control S), used as a "soft handshake" to stop serial transmission; and DC1 (control Q), the complementary character to resume transmission.

Unfortunately, ASCII doesn't provide for subscripts, exponents, or any Greek or scientific characters. As a minimum, it would be nice to have π, μ, Ω, and the degree symbol (°), which crop up frequently in technical writing. Of course, it is possible to use a control character (or sequence of characters) to indicate a change of font or alphabet. This is

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

TABLE 10.3. ASCII CODES

non-printing					printing			printing			printing		
Name	Control char	Char	Hex	Dec	Char	Hex	Dec	Char	Hex	Dec	Char	Hex	Dec
null	ctrl-@	NUL	00	00	SP	20	32	@	40	64	`	60	96
start of heading	ctrl-A	SOH	01	01	!	21	33	A	41	65	a	61	97
start of text	ctrl-B	STX	02	02	"	22	34	B	42	66	b	62	98
end of text	ctrl-C	ETX	03	03	#	23	35	C	43	67	c	63	99
end of xmit	ctrl-D	EOT	04	04	$	24	36	D	44	68	d	64	100
enquiry	ctrl-E	ENQ	05	05	%	25	37	E	45	69	e	65	101
acknowledge	ctrl-F	ACK	06	06	&	26	38	F	46	70	f	66	102
bell	ctrl-G	BEL	07	07	'	27	39	G	47	71	g	67	103
backspace	ctrl-H	BS	08	08	(	28	40	H	48	72	h	68	104
horizontal tab	ctrl-I	HT	09	09	)	29	41	I	49	73	i	69	105
line feed	ctrl-J	LF	0A	10	*	2A	42	J	4A	74	j	6A	106
vertical tab	ctrl-K	VT	0B	11	+	2B	43	K	4B	75	k	6B	107
form feed	ctrl-L	FF	0C	12	,	2C	44	L	4C	76	l	6C	108
carriage return	ctrl-M	CR	0D	13	-	2D	45	M	4D	77	m	6D	109
shift out	ctrl-N	SO	0E	14	.	2E	46	N	4E	78	n	6E	110
shift in	ctrl-O	SI	0F	15	/	2F	47	O	4F	79	o	6F	111
data line escape	ctrl-P	DLE	10	16	0	30	48	P	50	80	p	70	112
device control 1	ctrl-Q	DC1	11	17	1	31	49	Q	51	81	q	71	113
device control 2	ctrl-R	DC2	12	18	2	32	50	R	52	82	r	72	114
device control 3	ctrl-S	DC3	13	19	3	33	51	S	53	83	s	73	115
device control 4	ctrl-T	DC4	14	20	4	34	52	T	54	84	t	74	116
neg acknowledge	ctrl-U	NAK	15	21	5	35	53	U	55	85	u	75	117
synchronous idle	ctrl-V	SYN	16	22	6	36	54	V	56	86	v	76	118
end of xmit block	ctrl-W	ETB	17	23	7	37	55	W	57	87	w	77	119
cancel	ctrl-X	CAN	18	24	8	38	56	X	58	88	x	78	120
end of medium	ctrl-Y	EM	19	25	9	39	57	Y	59	89	y	79	121
substitute	ctrl-Z	SUB	1A	26	:	3A	58	Z	5A	90	z	7A	122
escape	ctrl-[	ESC	1B	27	;	3B	59	[	5B	91	{	7B	123
file separator	ctrl-\	FS	1C	28	<	3C	60	\	5C	92	|	7C	124
group separator	ctrl-]	GS	1D	29	=	3D	61	]	5D	93	}	7D	125
record separator	ctrl-^	RS	1E	30	>	3E	62	^	5E	94	~	7E	126
unit separator	ctrl-_	US	1F	31	?	3F	63	_	5F	95	DEL	7F	127

the usual method used in technical word processing, where the formatter interprets subsequent ASCII characters differently. This is probably the best solution anyway, since, given the variety of symbols needed for any serious technical writing, you wouldn't be happy for long even with a very large fixed ASCII alphabet.

Note that computer keyboards are often implemented not simply as ASCII code generators, one code per keystroke; instead, recent practice is to generate unique "key down" and "key up" codes for each key. Special system software (a "keyboard driver," see Section 10.18) may then translate the keystrokes into vanilla

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

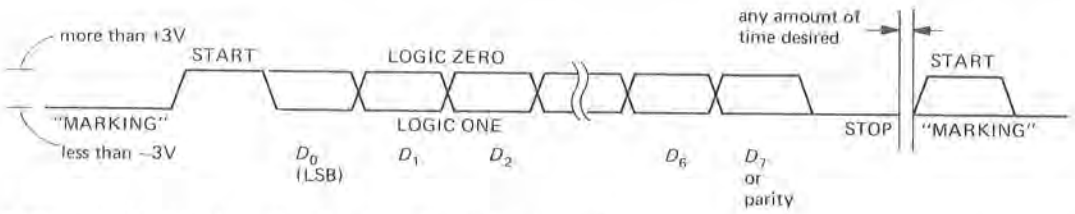


Figure 10.16. RS-232 serial data-byte timing waveform.

ASCII. However, this implementation allows much greater flexibility, since you can configure the keyboard driver to have auto-repeat keys, multiple shifts, keyboard remapping (e.g., a Dvorak keyboard), "hot keys," etc.

Bit-serial transmission

ASCII (or any other alphanumeric code) can be transmitted either as a parallel 8-bit group (8 separate wires) or as a serial string of 8 bits, one after the other. For transmission at low to moderate speeds it is most convenient to use serial transmission, to simplify wiring. A modem (discussed later in this section) converts a serial bit stream to an audio signal, and vice versa (e.g., by using one audio tone for "1," another for "0"), which can then be sent via telephone lines; serial transmission is a natural here, too. Serial transmission has a standard bit-transmission protocol and fixed bit rates: With *asynchronous* transmission, a *start* bit and a *stop* bit (sometimes two) are attached to the ends of each 8-bit character, forming a 10-bit group. The sender and receiver use a fixed bit rate, the most popular of which are 300, 1200, 2400, 4800, 9600, and 19,200 baud (= clock periods per second). Figure 10.16 shows the idea.

When no information is being sent, the transmitter sits in the "marking" state (the language comes from the teletypewriter days, with "mark" and "space"). Every character begins with a START bit, followed by the 8 ASCII bits, least significant bit first (usually organized as 7 data bits, plus 1 optional parity bit), and a final STOP bit; the latter must be at least one clock period, but may extend any amount longer. At the receiving end, a UART ("universal synchronous/asynchronous receiver/transmitter," see Section 11.11) operating at the same baud rate synchronizes to each 10-bit group, generating successive 8-bit parallel data groups from the input serial string. By resynchronizing on the START and STOP bits of each character, the receiver doesn't require a highly accurate clock; it only has to be accurate and stable enough for the transmitter and receiver to stay synchronized to a fraction of a bit period over the time of one character, i.e., an accuracy of a few percent. The receive UART is triggered by the transition at the beginning of the START bit, waits for half a bit cell to be sure the START bit is still present, and then examines the data value at the middle of each data cell. The STOP bit terminates the character and is the resting state if no new characters are sent immediately. The receive UART looks for the STOP bit level 10.5 bit-cell intervals after the START transition, to help verify a correctly sent character. "Break" is a continuous space, which cannot occur during normal character transmission. Programmable baud-rate generators (i.e., programmable dividers) are available that generate any of the standard baud rates from a single oscillator input frequency, with the output baud rate selected by a binary input code. Most modern UARTs (for example the dual-channel synchronous/asynchronous 8530 from Zilog) include internal software-programmable baud-rate generators.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

RS-232

The actual serial ASCII signals can be sent in one of several ways. The original method, which dates back many decades, consists of switching a 20mA (or sometimes 60mA) current at the selected baud rate. This is known as "current-loop" signaling. It is sometimes available as an option, but has been superseded for moderate baud rates by the EIA RS-232C standard of 1969 (and subsequent RS-232D standard of 1986), which uses bipolarity *voltage* signaling. The RS-232 standard specifies the properties of both drivers and receivers: A driver must generate voltage levels of +5 to +15 volts (logic LOW input), and −5 to −15 volts (logic HIGH input), into a load of 3k to 7k, with a slew rate of less than 30V/μs, and the ability to withstand a short to any other output (which can be as inhospitable as ±5V@500mA); a receiver must present a 3k to 7k load resistance, converting an input of +3 to +25 volts to logic LOW, and an input of −3 to −25 volts to logic HIGH. Note that logic 1 gets inverted by the RS-232 driver to a negative level, called "mark"; logic 0 is a positive level ("space"). In current-loop transmission, current flows during logic 1 (mark), and ceases during logic 0 (space).

RS-232 receivers usually have voltage hysteresis at the input, and some types let you limit the response speed with a capacitor, to reduce susceptibility to noise pulses. Look at Sections 9.14 and 14.17 for a discussion of official RS-232 driver and receiver ICs. RS-232 works well up to 38,400 baud over distances of tens of feet, even with unshielded bundled multiwire cable; for short links it is sometimes used at 115,200 baud.

RS-232 also specifies the connector type and pin assignments. Unfortunately, it doesn't specify enough! This is an eternal source of confusion because, in general, two RS-232 devices, when connected together, won't work. The problem is so annoying that readers of the previous edition of this book have even complained to *us*, because we didn't tell them what to do about it. Luckily for you, you're reading the second edition. Here's the story:

There are two basic problems in this business: (a) There are two flavors of device defined, with input pins of one type corresponding to output pins of the other; you may want to connect two similar devices together, or you may want to connect two complementary types together. (b) There are five "handshaking" signals; some devices send them out, and expect to receive them back, while others ignore their inputs (and don't drive their outputs). To make things work, you've got to understand these in detail. Let's plunge in.

RS-232 was designed for connecting DTEs ("data terminal equipment") to DCEs ("data communication equipment"). A terminal always looks like a DTE, and a modem always looks like a DCE; but other devices, including microcomputers, can be either. The IBM PC looks like a DTE with a male connector, although most large computers are DCE-like. When you connect a DTE to a DCE, you just connect corresponding pins of their DB-25 connectors (which can be either male or female, at either end!), and, with some luck, it may work. We say *may*, because it still depends on which handshaking lines each device expects from the other, and bothers to drive itself. (Of course, even when the cable is right, you still have to agree on baud rate, parity, and a few other software parameters!) When you want to connect two *similar* devices, on the other hand, you can't connect corresponding pins, because that would connect the two outputs together: A DTE transmits on pin 2 and receives on pin 3, while a DCE does the reverse. So you have to connect them with a cable (called a "null modem") that criss-crosses

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

TABLE 10.4. RS-232 SIGNALS

Name	Pin number 25-pin	9-pin	Direction (DTE↔DCE)	Function (as seen by DTE)	
TD	2	3	→	transmitted data	} data pair
RD	3	2	←	received data	
RTS	4	7	→	request to send (= DTE ready)	} handshake pair
CTS	5	8	←	clear to send (= DCE ready)	
DTR	20	4	→	data terminal ready	} handshake pair
DSR	6	6	←	data set ready	
DCD	8	1	←	data carrier detect	} enable DTE input
RI	22	9	←	ring indicator	
FG	1	–		frame ground (= chassis)	
SG	7	5		signal ground	

pins 2 and 3. Unfortunately, that's not all there is to it.

Table 10.4 shows all the important lines. TD and RD are the data transmit and receive lines; RTS and CTS are "ready to send" and "clear to send"; DTR, DSR, and DCD are "data terminal ready," "data set ready," and "data carrier detect." There are, in addition, two grounds: a "frame ground" (or chassis, pin 1) and a "signal ground" (pin 7); most machines just tie them together. The five signals that aren't data are *handshaking*-type control signals: A DTE asserts RTS and DTR when it's ready to receive, and a DCE asserts CTS and DSR when it is ready to receive. Some DTEs also expect their DCD input to be asserted before they will do anything. All signal lines are RS-232 bipolarity levels, with data (TD, RD) asserted *negative*, but control lines (RTS, CTS, DSR, DTR, DCD) asserted *positive*.

Note that the signal names make sense only as viewed by the DTE: For instance, pin 2 is called TD ("transmitted data") by *both* sides, even though the DTE asserts it and the DCE receives it. Thus, the name of a pin isn't enough to tell you if it's an input or output – you also need to know whether the device thinks it's a DTE or a DCE (or you can cheat and use a voltmeter!).

If all RS-232 devices asserted everything they are supposed to and listened to everything they are supposed to, then you could just connect corresponding pins (for DTE ↔ DCE), or cross corresponding pairs (for DCE ↔ DCE, or DTE ↔ DTE). However, when you connect a device that ignores all handshaking lines to one that expects them, nothing happens. So you have to tailor your strategy to the reality; this sometimes involves trickery. Figure 10.17 shows how to make cables that actually work, for all (well, *nearly* all) situations. In part A we show the connection for DTE ↔ DCE when both devices use full handshaking. RTS/CTS is one pair of handshakes, and DTR/DSR is the other. In C we show the same thing, but with a "null modem" cable to cross inputs and outputs for a DTE ↔ DTE pair. The same cable works for a DCE ↔ DCE pair, but you should reverse the arrows in the picture, and omit the connections to pin 8. These cables won't work, though, if one device is looking for handshaking and the other isn't providing it. In that case the easiest thing

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

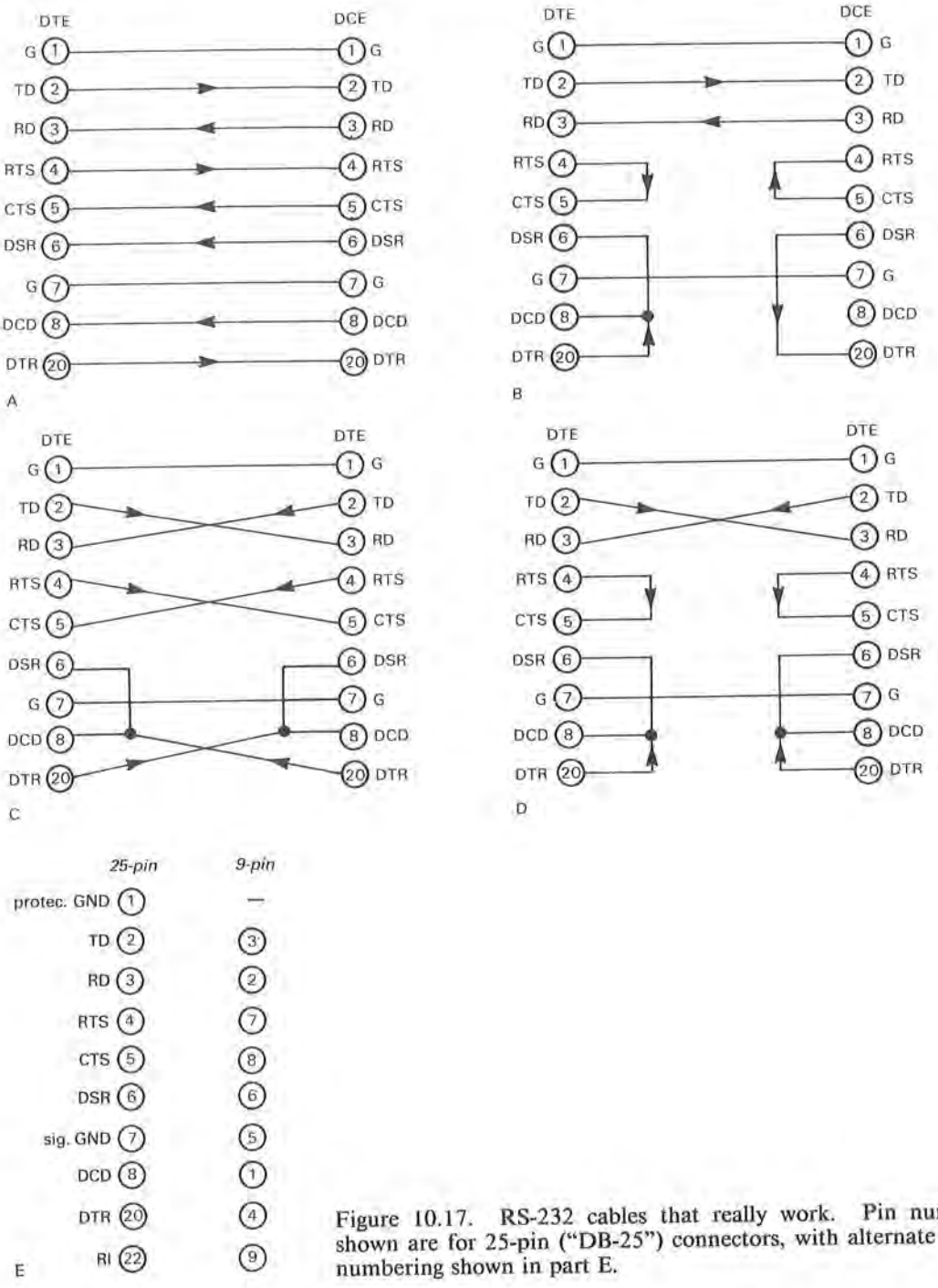


Figure 10.17. RS-232 cables that really work. Pin numbers shown are for 25-pin ("DB-25") connectors, with alternate 9-pin numbering shown in part E.

is to wire the cable so that each device provides its own handshakes, i.e., tells itself to go ahead. That's shown in B for DTE ↔ DCE and in D for DTE ↔ DTE (works also for DCE ↔ DCE, but you should omit the connections to pin 8).

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

How to become an RS-232 genius. If you make up these four cables, with a male *and* female connector at each end, you can make anything work with anything else (almost). Your colleagues will swear you're a genius. They will, that is, unless they've discovered the *real* professional's gimmick, an "RS-232 breakout box." It has LEDs for each line, so that you can see who is asserting what, and it has little jumpers so that you can connect any given pin to any other pin. Instructions: Look at the lights to get TD and RD connected right, then look again to see who asserts the handshakes. If a device asserts RTS, it probably looks at CTS. If both do, connect them together; otherwise, loop its RTS back to its CTS. Play the same game with DTR and DSR. If only one pair of handshakes is implemented, it is usually DTR/DSR. In general, the DTR/DSR pair is used to make sure the other side is connected and turned on, while the RTS/CTS pair is used to start and stop transmission as one side gets ahead of the other.

If you're too cheap to buy a breakout box, use a voltmeter to check for implemented signals: Any line with a large (>4V) negative or positive level is asserted; any line floating near ground is not.

Software handshaking. Some devices use the RTS/CTS hardware handshakes to start and stop data transmission while the slower device (e.g., a printer) catches up. Others transmit a "software handshake": CTRL-S (to stop) and CTRL-Q (to resume). If you're lucky, you'll have a choice. The software method means you can use a simpler cable, and if the devices ignore the the hardware lines altogether, your cable is extremely simple, with only pins 1, 2, 3, and 7 connected (all you have to figure out is whether or not to cross pins 2 and 3). The devices may still expect the hardware handshakes to be connected to enable the link, even if they use CTRL-S and CTRL-Q for detailed handshaking. In that case you can get away with the scheme of Figure 10.17B,D. Just make sure you remember to turn on the power at both ends, because neither side has any way to know that the other is alive, or even connected!

Other serial standards: RS-422, RS-423, and RS-485

The RS-232C standard was frozen in 1969, when serial data communication was a relatively leisurely occupation. It works well up to 50 feet, at speeds up to 19,200 baud. But computer and peripheral speeds have been doubling every year or two, and a better standard for serial communication was needed.

As we discussed in Section 9.14, RS-423 is an improved bipolarity single-ended protocol, good to 100kbaud and to 4000 feet (not at the same time); it is essentially compatible with RS-232. RS-422 is a unipolarity differential protocol good to 10Mbaud and to 4000 feet (see Fig. 9.37 for the speed/length tradeoff). RS-485 is similar to RS-422, but with additional specifications so that many drivers and receivers can share a single line. Table 10.5 summarizes the characteristics of these four standards.

Modems

As we remarked earlier, a *modem* ("*mo*dulator/*dem*odulator") is used to convert bit-serial digital quantities into analog signals that can be sent over telephone lines or other transmission paths (Fig. 10.18). An *internal* modem plugs into a slot in your computer (or comes built-in), whereas an *external* modem is a stand-alone box, powered from the ac power line, with RS-232 connection to

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

TABLE 10.5. SERIAL DATA STANDARDS

	RS-232C/D	RS-423A	RS-422A	RS-485
Mode	single-ended	single-ended	differential	differential
Maximum number				
drivers	1	1	1	32
receivers	1	10	10	32
Maximum cable length	15m	1200m	1200m	1200m
Maximum data rate (bits/s)	20k	100k	10M	10M
Transmit levels	±5V min ±15V max	±3.6V min ±6.0V max	±2V min (diff'l)	±1.5V min
Receive sensitivity	±3V	±0.2V	±0.2V	±0.2V
Load impedance	3k to 7k	450Ω min	100Ω min	60Ω min
Output current limit	500mA to V_{cc} or gnd	150mA to gnd	150mA to gnd	150mA to gnd 250mA to -8V or +12V
Driver Z_{out}, min (pwr off)	300Ω	60k	60k	120k

your computer's serial port. In either case the modem communicates with the telephone line, in one of two ways: (a) direct connection, via a telephone-type "modular jack," or (b) "acoustically coupled," by seating the telephone handset into a rubbery cradle containing microphone and speaker. Acoustically coupled modems are pretty much out of style these days, although they can be handy in hotel rooms where you may not want to crawl around under the beds looking for a modular jack (which may not even exist!).

In most situations you want to be able to send data on a single telephone channel in both directions simultaneously ("full duplex"), sharing the telephone audio bandwidth, which is roughly 300Hz–3kHz. There are three full-duplex formats in common use: 300 baud FSK (Bell 103), 1200 baud dibit PSK (Bell 212A), and 2400 baud dibit PSK (FSK stands for "frequency-shift keying," and PSK stands for "phase-shift keying"). A modem designed for 1200 baud, say, generally also supports 300 baud communication, etc. Although you don't need to understand how the modem encodes its data in order

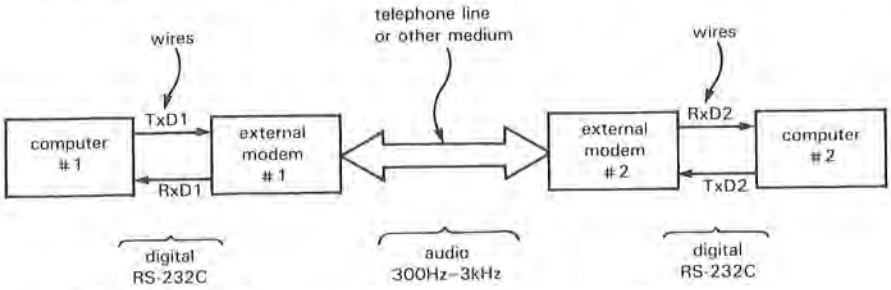


Figure 10.18. Modem communication.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

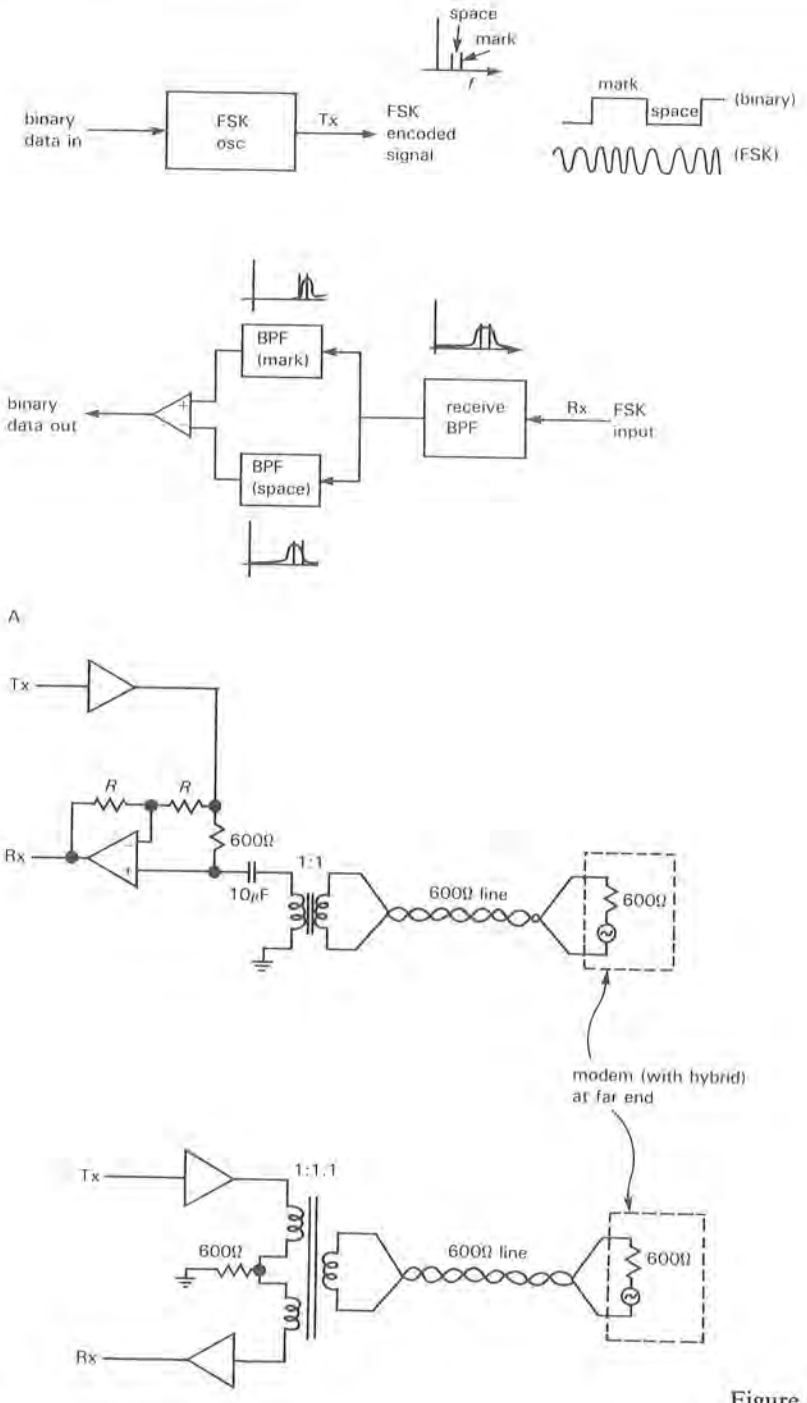


Figure 10.19. A. FSK modem. B. Hybrid couplers

to use it, the methods are interesting in their own right, and we can't resist describing them briefly.

The 300 baud standard (Bell 103) uses frequency-shift keying (FSK), in which a designated pair of audio tones represents mark and space: 1270Hz (mark) and 1070Hz (space) in one direction, 2225Hz and 2025Hz in the other. A Bell 103 modem is very simple, with a switchable oscillator for transmitting, and a pair of audio filters for receiving (Fig. 10.19A). Note also the use of a *hybrid* circuit (Fig. 10.19B) to isolate the outgoing signal from the received signal: Assuming the telephone line is close to its nominal 600 ohms impedance, none of the modem's own transmitted signal (Tx) appears back at its received-signal (Rx) output. In practice, hybrids don't work that well, because the telephone line impedance can deviate substantially from the nominal 600 ohms (see Section 14.5). Thus, it is important to have a very sharp receive filter, which adds some complexity to the modem circuit.

EXERCISE 10.5

Figure out how the hybrid circuits in Figure 10.19 work. Then impress your friends with your new knowledge.

The 1200 baud standard (Bell 212A) works differently. The digital data stream is grouped into bit *pairs* ("dibits"); each of the four possible dibits is transmitted as a designated phase shift of a fixed-frequency carrier (00: +90°, 01: 0°, 10: 180°, and 11: −90°), with smooth transitions of phase from each transmitted dibit to the next. Thus, dibits are transmitted at a 600Hz rate. The (phase-modulated) carrier frequency is 1200Hz in one direction, 2400Hz in the other. The receiving modem decodes by looking at the *difference* in phase of adjacent dibits. This clever idea has one pitfall, namely that the receiver loses track of relative phase if there is a long run of similar dibits. Therefore, in order to prevent long runs of constant phase, the transmitted data stream is randomized by exclusive-ORing it with a pseudo-random sequence (generated by a 17-bit shift register with XOR feedback from the 14th bit, see Section 9.32), with an identical descrambling process at the receiving end.

The 2400 baud full-duplex modems also use phase-encoded dibits, though with a different set of phases. These sophisticated modems tend to use real-time adaptive equalizers to correct the frequency and time-delay errors of the telephone line, and highly optimized filters for both transmitted and received signals. The end result is that the error rate is not significantly degraded when compared with the earlier 300 baud FSK modems.

You don't have to construct a modem from scratch, because complete modem chips and modules are made by AMI/Gould, Exar, National, Rockwell, Silicon Systems, and TI. Your life is made even easier, however, if you buy a complete modem, whether in the form of an internal plug-in card or an external box with RS-232 connection to your computer. Modems cost $100–$300, depending on features. Look for "Hayes-compatible" modems, which accept standardized commands for dialing, etc., that are now the de facto standard used by all communications software.

Some good advice: When using a modem to transfer data files between computers, be sure to use a block-checking modem protocol such as Kermit or XMODEM. These send the data in fixed-length blocks, each with error-checking checksums. The receiving modem compares the checksums, automatically insisting on retransmission of bad blocks. Files received this way are guaranteed error-free; files sent with plain unformatted ASCII transmission, by contrast, can almost be guaranteed to have errors!

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

10.20 Parallel communication: Centronics, SCSI, IPI, GPIB (488)

For cable communications with high-speed peripherals, parallel transmission is generally better than serial. Here are the popular favorites.

Centronics

This is a simple byte-wide unidirectional parallel port with handshaking, originated by Centronics and now widely used for printers. Unlike RS-232, *it always works!* Table 10.6 lists the signals, which are supposed to be sent with twisted-pairs and terminated in a 36-pin connector. Figure 10.20 shows the corresponding timing.

The basic signals are listed in the first group: D0–D7, STROBE′, ACKNLG′, and BUSY. BUSY is a flag: When LOW, the printer is not "busy," i.e., it's ready to accept data; the data source (computer) therefore asserts DATA, then a STROBE′ (with data guaranteed valid on both sides).

TABLE 10.6. CENTRONICS (PRINTER) SIGNALS

Name	Pin number sig	Pin number com	Direction	Description
STROBE'	1	19	OUT	data strobe
D0	2	20	OUT	data LSB
D1	3	21	OUT	•
D2	4	22	OUT	•
D3	5	23	OUT	•
D4	6	24	OUT	•
D5	7	25	OUT	•
D6	8	26	OUT	•
D7	9	27	OUT	data MSB
ACKNLG'	10	28	IN	finished with last char; pulse
BUSY	11	29	IN	not ready (note 1)
PE'	12	30	IN	HIGH = no paper
SLCT	13	–	IN	pulled HIGH
AUTO FEED XT'	14	–	OUT	auto LF
INIT'	31	16	OUT	initialize printer
ERROR'	32	–	IN	can't print (note 2)
SLCT IN'	36	–	OUT	deselect protocol (note 3)
GND	–	33	–	additional ground
CHASSIS GND	17	–	–	chassis ground

note 1: BUSY = HIGH
i) during each char transfer
ii) if buffer full
iii) if off-line
iv) if error state

note 2: ERROR' = LOW
i) if out-of-paper
ii) if off-line
iii) if error state

note 3: normally LOW
i) sending DC3 when SLCT IN' = HIGH deselects printer
ii) can only re-select by sending DC1 when SLCT IN' = HIGH

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

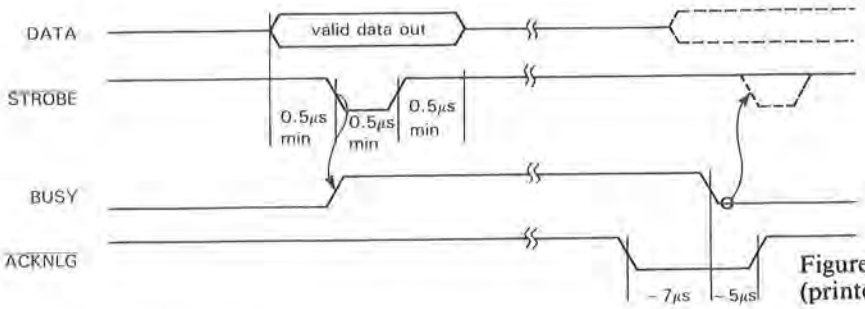


Figure 10.20. Centronics (printer) interface timing.

BUSY then goes HIGH, and it comes LOW again only when the printer is ready for another byte. The computer should look at BUSY, as shown, in order to know when it can send another byte. ACKNLG′ (which is a pulse, not a level) can be used to trigger an interrupt; don't try to use it instead of BUSY, though, because it may be gone by the time you look, and you'll wait forever.

There are several other signals, to indicate that the printer is out of paper (PE′), or off-line (ERROR′ or BUSY); the computer can initialize the printer (INIT′), ask for automatic line feed (AUTO FEED XT′), or send a byte to deselect the printer (set SLCT IN′ HIGH, then send an ASCII DC3). Note the relaxed timings, obviously intended for a slow (mechanical) device that can't accept data at a high rate. Most printers have some buffer memory, so they can accept data at a high rate initially; on the average, though, you can send bytes only at the printing rate. For a dot-matrix printer you're talking 100–300 bytes per second.

If you need to design a Centronics interface to go on some computer's bus, the easiest thing is to drive all the output lines from latched data via programmed I/O: Make D0–D7 one port, and the remaining lines (including STROBE′) a second port. For the input signals (BUSY, etc.), don't latch anything, just enable them onto the bus for programmed IN. A nice touch is to use ACKNLG′ to make an interrupt. Figure 10.21 shows the idea, for the IBM PC bus. Note that interrupts are easy here, because the PC uses edge triggering; just use the trailing edge of ACKNLG′, as shown. We've used one of the latched output bits to disable the interrupt line, as discussed in Sections 10.09 and 10.11. Note also the use of the bus signal RESET DRV to disassert all outputs (and also interrupts) at power-on; that's why we chose the '273 octal D register (which has a RESET′ input).

To use this interface, you assert and disassert output control lines selectively by sending OUT bytes to port B, with appropriate bits set to 1 or 0. With a latched output arrangement like this you can always safely change the state of one output bit without introducing glitches on the unchanged outputs. For this purpose, keep a copy in memory of the current byte latched in port B, so you can send out a new byte to port B with only one bit changed (by using AND and OR, see example below). To generate a STROBE′ pulse you must use software, since the interface has no ugly monostables. Program 10.6 shows how you make a "software pulse" on the STROBE′ line. Note the use of AND and OR, to clear and set a single bit, respectively. In this example we didn't bother updating the byte stored in "current," because at the end it was unchanged. If instead we had changed (and left changed) one of the other control bits, we would have saved the new byte with a "`MOV current,AL`" instruction at the end.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

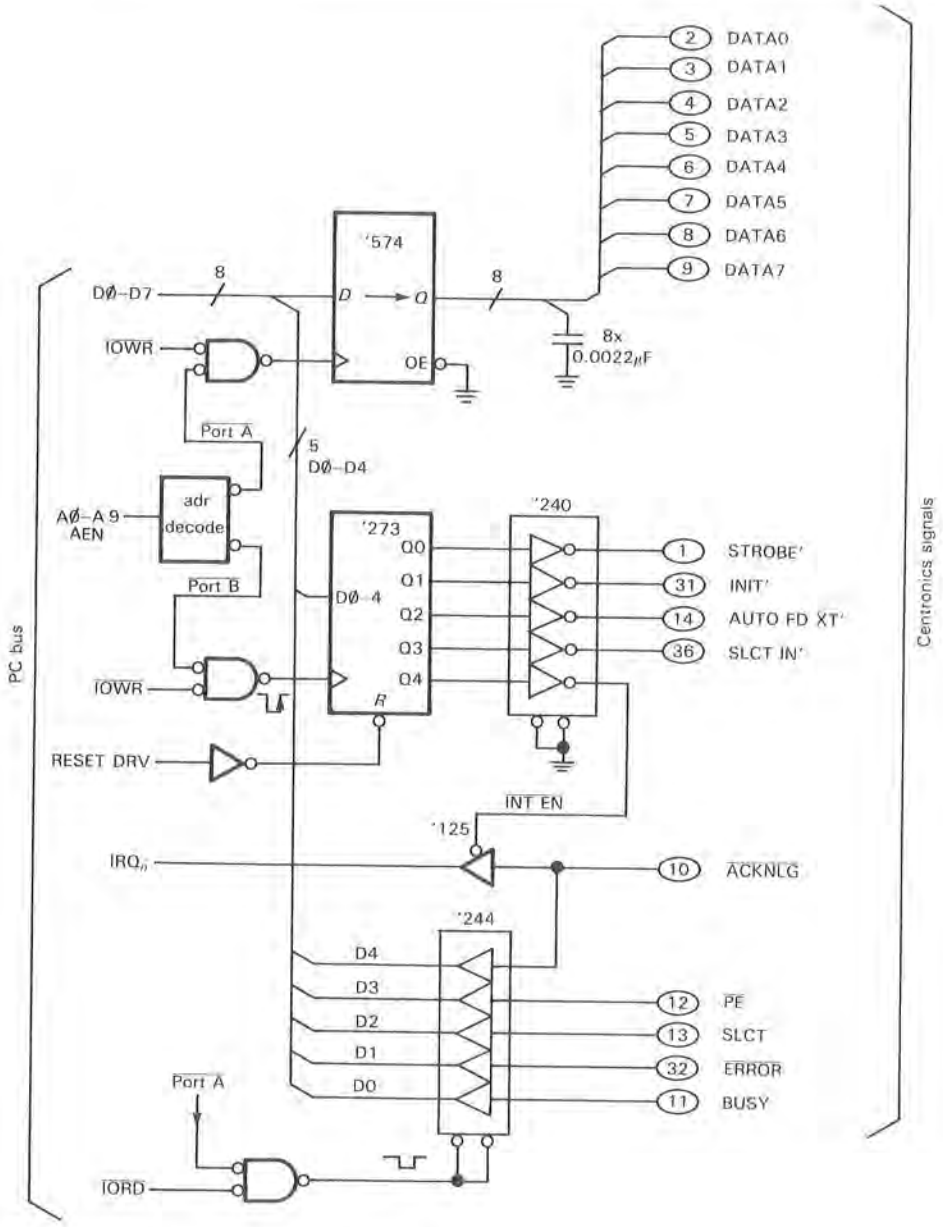


Figure 10.21. Centronics port for PC.

A hardware alternative to keeping a copy of the port byte in memory is to add a "readback" port to the interface, so a programmed IN lets you look at what's actually latched. The next example lets you discover how.

EXERCISE 10.6

Imagine that you are feeling energetic and want to add a readback port to the Centronics interface circuit. Make an IN from port B do the job. You should be pleasantly surprised at how little hardware is required.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

Program 10.6

```
                                ;make a software pulse
                                ;assume Cenronics "port B" address is in DX
                                ;assume strobe bit (bit D0) is initially "1"
current DB 0                    ;copy of port B kept here
        o
        o
      MOV  AL,current           ;current value of control byte
      AND  AL,0FEH              ;clear D0
      OUT  DX,AL                ;send to port B
      OR   AL,1                 ;set D0
      OUT  DX,AL                ;and send it out again
        o
        o
```

EXERCISE 10.7

Now rewrite Program 10.6, using your new port and omitting the use of "current."

Centronics ports are standard on nearly all microcomputers; don't hesitate to take advantage of it, if you need a quick and simple parallel output port. In many cases (but not on the IBM PC) the microcomputer will even let you use the port bidirectionally; the usual way that's done is by sending a control bit to the port to reverse the direction of the single 8-bit data path.

SCSI and IPI

These are universal parallel interface standards for connecting disks and other high-performance peripherals to microcomputers, as mentioned briefly in Section 10.16. SCSI ("Small Computer System Interface") is an 8-bit parallel cable interface with handshakes and protocols for handling multiple hosts and multiple peripherals. It has both asynchronous and synchronous modes, and defined software protocols. You can get SCSI interface cards to plug into most popular microcomputer buses, including VME and Multibus I and II; you then connect this SCSI "host adapter" to the peripheral's controller card via a flat-cable SCSI bus (Fig. 10.22). The controller card is often part of the peripheral itself (e.g., it may be attached to a hard-disk drive) and communicates with the drive by a "device-level interface," which will have a name like "ST-506/412," ESDI, or SMD.

SCSI has the advantage of effectively making all microcomputers compatible with all peripherals. Everyone's rushing to adopt SCSI, and new microcomputer designs incorporate it right on the CPU motherboard. At the peripheral end, manufacturers are eliminating the controller by going to an "embedded-SCSI" architecture, in which the SCSI bus becomes also the device-level interface. In other

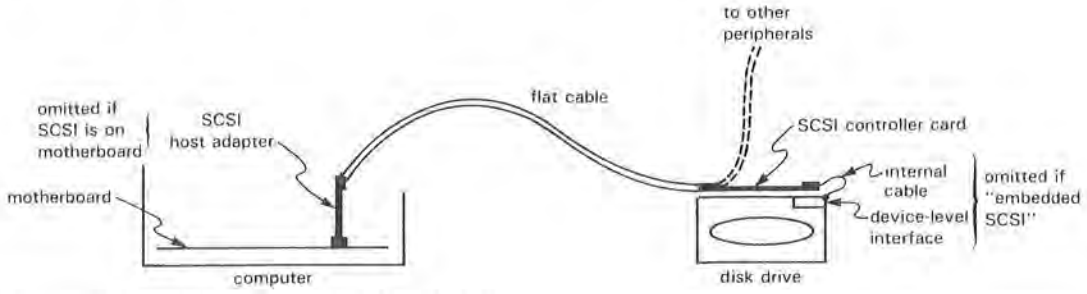


Figure 10.22. SCSI bus with single peripheral.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

words, you just hook a cable from the microcomputer's motherboard to the disk drive. SCSI supports data rates to 1.5Mbyte/s (asynchronous) or 4Mbyte/s (synchronous), with cable lengths to 20 feet (single-ended) or 80 feet (differential).

SCSI is complicated enough that we don't have room here to define all its signals, modes, command protocols, and interfacing possibilities. However, because of its popularity, there are single-IC interface chips (e.g., the NCR 5380 series, Western Digital 33C90 series, and others from Fujitsu, Ferranti, etc.) to make your life easy.

SCSI works well with current-generation disks. However, in order to increase data-transfer rates, the industry is considering going to a 16-bit wide interface bus. For this the IPI ("Intelligent Peripheral Interface") may be the next interface bus of choice. IPI specifies a 16-bit parallel bus operating to 10Mbyte/s (5MHz transfer rate); like SCSI, it also works with multiple hosts and peripherals. Hard-disk drives have been getting denser and faster at an amazing pace lately; given the increasing transfer rates, the world is heading rapidly toward universal embedded-bus interfaces (SCSI or IPI). In a few years you probably won't see any other formats.

□ *IEEE-488 (GPIB, HPIB)*

When laboratory instruments first became available with actual data outputs on the back, it was a case of "each company for itself." There were nearly as many interface protocols as there were instruments, with parallel and serial modes, positive and negative polarities, and all sorts of crazy handshakes. It was total pandemonium. We remember vividly designing a huge-digit (6 inches high) display for use in Harvard's lecture halls: It had separate input circuitry for each instrument we owned!

Hewlett-Packard decided in the mid-1960s to end this craziness by defining a universal instrument interface. They modestly called it the Hewlett-Packard Interface Bus (HPIB) and implemented it as the only option on all new designs. It permits up to 15 instruments on a single bus cable up to 20 meters long, with a cleverly designed connector that you can stack at each node. The HPIB bus protocol is byte-wide with handshakes, and it allows data-transfer rates to 1Mbyte/s; it includes software commands to enable any connected device to become a "talker" (source of data), and any combination of the remaining devices to be "listeners" (recipients of data). A "controller" (dictator) tells everybody what to do.

HPIB worked so well that a standards committee was set up by the IEEE to make it official. The resulting standard is known as IEEE-488-1975/ANSI MC1.1, which everyone except HP refers to as "GPIB" ("general-purpose interface bus") or "488-bus." It has become the universal digital interface for laboratory instrumentation. The instruments of all companies can be strung together on the same GPIB, with a microcomputer (or fancy desk calculator) giving the orders. For example, you can set the waveform, frequency, and amplitude of a frequency synthesizer, then take voltage measurements from the same experiment or process.

10.21 Local area networks

In prehistoric times, computing was done in "batch mode" on large centralized computers. They were powerful (slower than the least powerful of today's personal computers, with a tiny fraction of the memory) and expensive (comparable to today's supercomputers). You punched your programs on decks of cards, then submitted the job. With luck, your aborted output was available by the end of the day, so you could resubmit the job the next morning to find the next bug.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

Nowadays we're all spoiled by incredible desktop horsepower, fast disks, beautiful graphics. We want more. We want to be able to exchange files with the guy down the hall without getting out of our chairs. We want instant access to everyone's data bases, printers, and fancy peripherals. The way we get it is with networking – both the worldwide networks like BITNET and DECNET and "local area networks" (LANs) like Ethernet and LocalTalk.

The field of networking is still in its infancy, and we expect dramatic changes in the next decade. A few trends have emerged, however, and it's worth describing the kinds of LANs in use today.

CSMA/CD (Ethernet)

Ethernet typifies "carrier-sense multiple-access/collision-detection" (CSMA/CD) networks. It uses coaxial line to transmit 10Mbit/s signals to the addressed recipient. An Ethernet message is sent in "packets," with a preamble and error-checking. The sending protocol goes like this: (a) wait until you see no activity on the network; (b) begin sending your message packet (see below); (c) while sending, check simultaneously for interference (a "collision"); (d) (i) as long as all is clear, continue sending your message, but (ii) if you detect interference, jam the network intentionally (to ensure that everyone else sees the collision!), then abort your transmission, wait a random length of time, and try again; wait a longer "random" time after each successive failure.

Ethernet messages are organized into relatively short packets (≈1kbyte maximum), each of which includes a *header* (identifying recipient and sender), a few bytes telling the packet's length, type, and sequence number, the actual group of data bytes, and finally a "cyclical redundancy checksum" (CRC), from which the recipient can verify error-free transmission. Note that a collision can occur only during the beginning of transmission of a packet, since [by rule (a) above] a transmission in progress for twice the network travel time will not be interfered with.

Ethernet was invented by Xerox and is widely used. It has ample bandwidth for most local area networks, and its performance degrades somewhat gracefully under heavy use, owing to the random retry protocol. You can get Ethernet controllers for most serious microcomputers (VAX, IBM PC, etc.) and buses (Multibus, VME), and it's the official network for the popular Sun and NeXT workstations. An Ethernet network can go up to 1km per segment, with up to 2 repeaters; you can also have fiber-optic "bridges" of greater length. A number of desktop computers can share a multiple-port RS-232 "server," tied into one node on the Ethernet coax. Servers can also tie into shared resources such as printers and large disks.

Token-ring networks

A token-ring network visits a closed set of nodes, in a ring configuration. Collisions are not allowed here, and the rules of the game go like this: Imagine some token object; whoever has it is permitted to send messages, while all others can only listen. In a token ring, the token is a short message that can be passed around when the owner is finished. At any time, one node owns the token and is free to send messages. As with Ethernet (and any other sensible network), the messages are packetized, often using the SDLC format ("Synchronous Data Link Control": one packet = flag + address + header + message + checksum + flag). The message packets circulate around the ring until the addressed recipient receives them. When the sender is finished sending the full message (normally many packets),

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

he sends the token. It circulates around until some other node in the ring, desiring to send a message, swallows it, becoming the new token owner.

LocalTalk

LocalTalk (formerly Appletalk) is a simplified collision network, designed by (guess who) Apple Computer. It is a linear network, not a ring. One node can transmit, while all listen. The cable is a single differential pair, with RS-422 signals transformer-coupled at each node. The packet format is SDLC. Maximum network length is 1000 feet, with up to 32 nodes attached. The network bandwidth is 230.4kbit/s. A compatible variant known as PhoneNET (Farallon Computing Inc.) uses standard telephone cable and connectors and claims to work up to 4000 feet.

The protocol is similar to Ethernet, but simpler: If you hear no activity, you may send a packet. The network hardware doesn't attempt to detect collisions; it just forwards received packets with valid checksums up to the next higher level of software. A collision generally clobbers the colliding packets, rendering both their checksums invalid; thus, the software never gets the message at all! It is the software's job to notice this: For example, the sender of a message expects a reply; if he doesn't get one after a while, he initiates an identical message and tries again. LocalTalk is a "CSMA/CA" network; the "CA" stands for collision *avoidance*, rather than Ethernet's collision *detection*.

LocalTalk has defined protocols for sharing of files and resources (printers, modems, etc.), and it has a method for naming devices connected to the network. You can even get LocalTalk interfaces for non-Apple computers, letting you ship files between Macintoshes, IBM-compatibles, and UNIX computers, and to shared resources such as laser printers.

□ 10.22 Interface example: hardware data packing

If all your instruments connect to a standardized interface bus (such as the GPIB), you're in great shape: Just buy the interface card for your computer, buy some cables, string things together, and hire a programmer. It doesn't take much talent, only money. However, this chapter is about bus interfacing, so we would like to conclude with a complete design example.

If you're like us, you probably don't throw out all your functioning instruments when something new comes along. Some extremely capable measurement instruments were made before the era of GPIB; you can bring new life to them by cooking up an interface to your lab computer. As an example, an 8-digit frequency counter with multiplexed display is likely to have a rear-panel output that gives you one digit after another ("digit-serial, bit-parallel"), encoded as 4-bit BCD, and probably presented at the display's internal refresh rate. You have no control over the timing; each valid digit, along with its 3-bit digit-position address, is signaled with a strobe. Such an instrument most likely uses TTL output levels.

Figure 10.23 shows how to interface such an instrument to an IBM PC. This is a complete interface, including a status flag, interrupt, and selectable I/O port address. The action begins at the lower left, where the counter is busy putting out successive digits, their addresses (0–7), and a STROBE′ pulse when the data is valid. The counter goes from the least significant digit (LSD) to the most significant digit (MSD), so a complete output cycle ends with the receipt of the MSD (digit 7). The eight '173 registers (4-bit *D* registers with three-state outputs) latch the successive digits, being driven in parallel and separately clocked via the decoded digit addresses. Note the use of a '138 strobed 1-of-8 decoder to generate

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

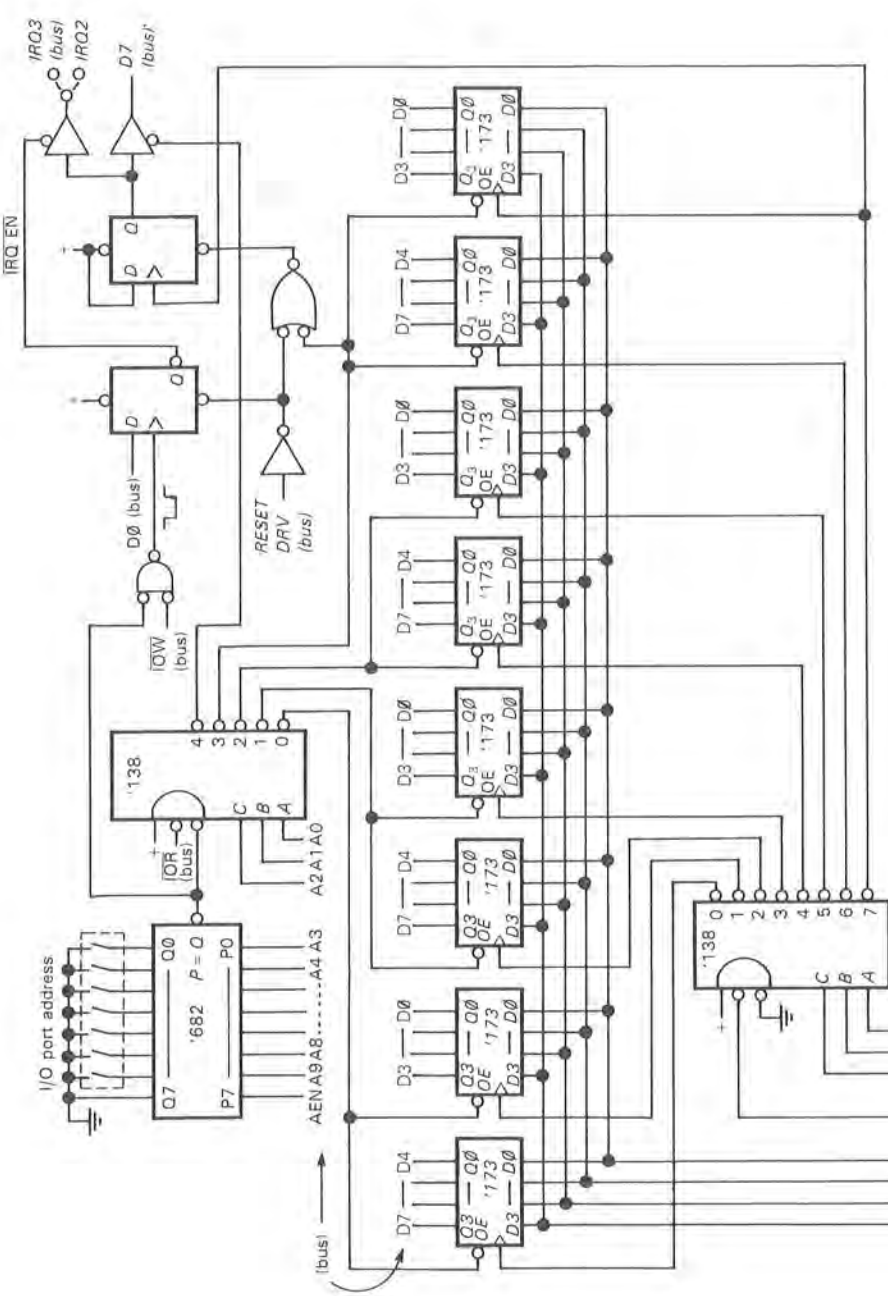


Figure 10.23. Character-serial interface.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

the digit clocking signals from the address and strobe.

The counter output is thus latched in the eight 4-bit registers, with the outputs connected as four 2-digit groups (8 bits each). The PC can thus bring in all eight digits with four byte-wide data `IN` commands, from four successive I/O port addresses (beginning with the one set on the DIP switch). In fact, it can do even better by reading from a 16-bit register (i.e., doing an "`IN AX,DX`," rather than an "`IN AL,DX`"), which causes two successive byte reads from consecutive I/O port addresses.

Note the simple address decoding scheme: A '682 octal comparator generates a LOW output when the 7 high-order address bits match the switch settings (and also the nuisance AEN is LOW); this "base address" enables a '138 1-of-8 decoder, strobed by IOR′, which decodes the low-order three address bits to generate the separate data `IN` enabling pulses corresponding to successive port addresses. This is a common method of handling address decoding, since you usually assign a few contiguous port addresses to the various registers of a single interface.

The status flag is set when the last digit of each group is received from the counter; it can be read with a data `IN` from PORT+4, where PORT is the address set with the DIP switch. The flag is cleared when the CPU reads the last (most significant) data byte (from PORT+3). This interface also has provision to make interrupts, jumper-selectable on either IRQ2 or IRQ3, and enabled by sending a 1 to PORT (and disabled by sending a 0); note the lazy address decoding we've used for `OUT`, to save a gate. In a spirit of good citizenship, both the status flag and interrupt enable flip-flops are cleared at power-on.

This interface is an example of "packing" data, the process by which several numbers are stuffed into one computer word. If the "numbers" happen to consist of single bits, you can pack 16 of them into each 16-bit word. This isn't as crazy as it sounds: In digital signal processing you sometimes deal with periodically sampled "hard-clipped" waveforms (which you can think of as 1-bit A/D conversion); for highest I/O throughput rate you pack in hardware (as we did in this example) and read in bus-wide words. Of course, if speed is not important, the simplest thing is to bring in the data with the least hardware and then do the packing and conversion in software. In the preceding example, for instance, you might latch and transfer to the CPU one digit at a time if you can be sure that the latency time of the computer is short enough that no digits will be lost.

EXERCISE 10.8

Modify the interface circuit so that the IRQ line used by the interface is *programmable*: Sending 01_H to PORT enables interrupts on IRQ2, and sending 02_H to PORT enables interrupts on IRQ3; both are disabled by sending 0 to PORT, and also at power-on.

A practical note about this interface circuit: In general it is best to avoid loading bus lines excessively. Our circuit ties each Dn line to the outputs of four '173 three-state registers, which is an undesirably large capacitive load. Although our circuit would undoubtedly work properly, it might limit the number of additional cards you could plug into the bus (particularly if the others sinned in the same way!). In this example, a single '244 three-state octal buffer, interposed between the D0–D7 outputs and the PC data bus, would be a good solution. It should be enabled with the AND of the decoded port address and IOR.

10.23 Number formats

In the preceding example, the bytes (or words) brought in are not in the computer's internal binary-number format; they're really BCD, packed two digits per byte (or

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

four per word). To do meaningful computation, it is best to convert them into an integer or a floating-point number (although there are "decimal-adjust" operations that let you do arithmetic directly on packed BCD numbers). Let's take a look at the usual number formats used in computers (Fig. 10.24), a subject we touched on briefly at the beginning of Chapter 8.

Integers

Signed integers are always represented in 2's complement, using either 1, 2, or 4 bytes, as shown. Thus, the most significant bit (MSB) tells the sign, even though 2's complement is not the same as sign/magnitude representation (e.g., −1 is 11111111, not 10000001; see Section 8.03). You can think of 2's complement as offset binary with inverted MSB; alternatively, you can think of it as an integer with the bit values as shown in the figure. Many computers let you declare variables as *unsigned* integers, in addition to 2's complement signed integers. A 2-byte unsigned integer can have values from 0 to 65535.

Floating-point numbers

Floating-point numbers, also called *real* numbers, are usually 32-bit ("single precision") or 64-bit ("double precision"), with an additional 80-bit format sometimes used for temporary values during calculations. Unfortunately there are several common representations in use. The most popular is the recently completed IEEE standard (officially known as ANSI/IEEE Std 754-1985), which has been implemented by nearly all floating-point chip sets (including Intel's 8087/287/387, Motorola's 68881, and chip sets from AMD, Weitek, et al.) and is therefore universal in microcomputers that accept those chips (this includes the IBM PC).

Figure 10.24 shows the IEEE 32-bit and 64-bit formats. The 32-bit single-precision format has 1 sign bit, 8 exponent bits, and 23 bits of fraction. The exponent tells the power of 2 that the fraction (see below) should be multiplied by. The exponent is "biased" by adding 127, so that the exponent field 01111111 corresponds to an exponent of 0; exponents thus go from −127 to +128. The fraction itself uses an interesting trick, originated by DEC in their floating-point format. A floating-point number in binary can always be written in the form $f.fff \times 2^e$, where f.fff is the (base-2) mantissa ("significand"), and e is the (power-of-2) exponent. In order to maximize the precision you get with a given number of mantissa bits, you "normalize" it by shifting the mantissa left (and decrementing the exponent) until the leading bit is non-zero, thus casting it in the form $1.fff \times 2^e$. Now, here's the "hidden-bit" trick: Since the resulting normalized significand always has a nonzero MSB, it would be redundant to display it; i.e., you don't put 1fff in the number, just the fff, with the leading 1 assumed. The resulting number gains one bit of precision, and has a range of $\pm 1.2 \times 10^{-38}$ to $\pm 3.4 \times 10^{38}$.

EXERCISE 10.9
Show that the range of normalized floating-point numbers is as claimed, by constructing the smallest and largest numbers.

The IEEE double-precision format is similar, but with the significand precision more than doubled (by attaching 29 more bits) and with the exponent fortified by an additional 3 bits. The range of numbers is as shown in the figure. There is also a whopping "extended-precision" (80-bit) format, as shown. The IEEE format allows non-normalized numbers also, to give some additional range at the small end (at the expense of precision); these "denormalized" numbers go down to $\pm 1.4 \times 10^{-45}$. The standard also defines zero (e = fff = 0; thus there are two

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

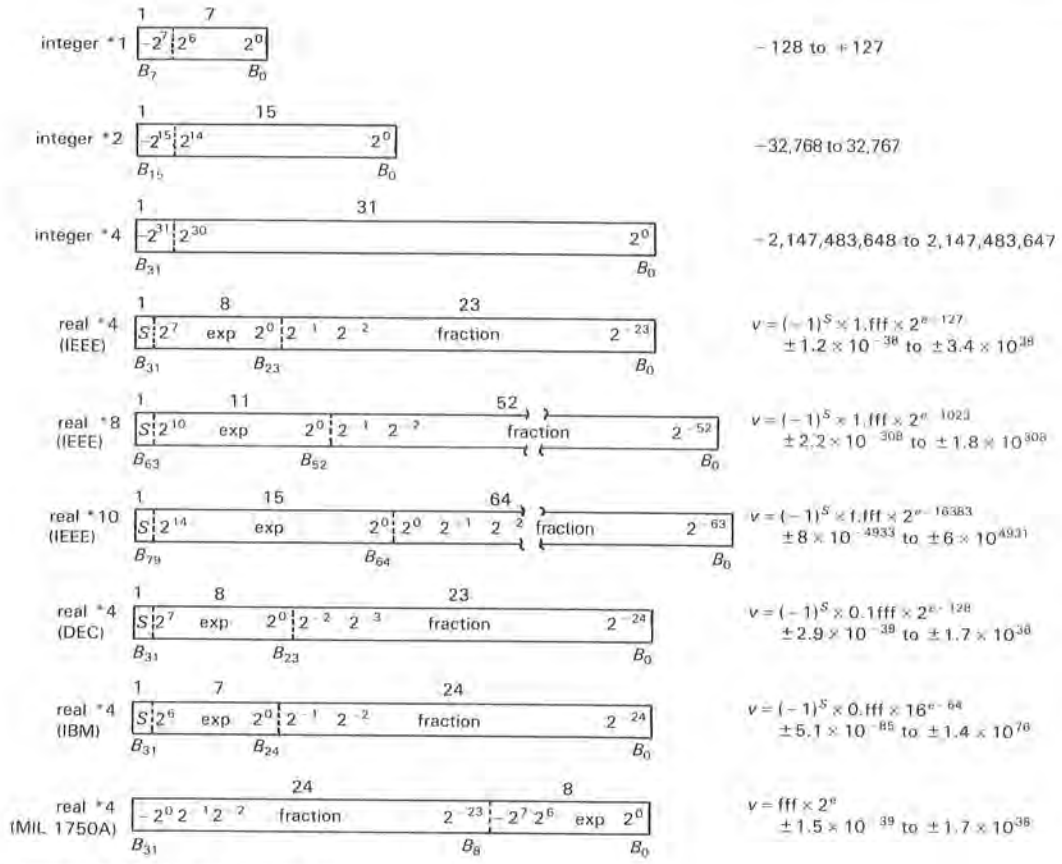


Figure 10.24. Number formats.

zeros, +0 and −0), infinity (e = all 1s, fff = 0; therefore both signs), and a curious class of reserved quantities known officially as NANs (NAN = "not a number")

The other important microcomputer floating-point format is DEC's, used in the MicroVAX and LSI-11 computers (and their ancestors, the VAX and PDP-11 minicomputers). It is very close to the IEEE standard, with the same number of exponent and mantissa bits (including the use of a hidden bit) used for single-precision numbers. In fact, the only differences are the exponent bias (128 instead of 127) and the fact that the mantissa has no leading bits, being instead of the form .1fff (with the "1" hidden). DEC defines only one zero (all bits zero), and does not permit non-normalized numbers or infinity; there are, however, analogs of the IEEE NANs. DEC also has a 64-bit double-precision format.

The last two formats in Figure 10.24 are used in large or special-purpose computers, but not in microcomputers. The "IBM" format has been used in mainframe IBM computers for some time and even in minicomputers like the Nova line from Data General. The 7-bit biased exponent tells the power of 16, rather than 2, giving greater exponent range. The mantissa therefore may have up to three leading zeros; i.e., a normalized fraction has a nonzero most significant hex digit.

EXERCISE 10.10

In order to understand the meaning of this last statement, write out the IBM representation of the number 1.0. Now write the next smaller number that can be represented in this format.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

By its choice of exponent radix the IBM format sacrifices some precision for dynamic range. Furthermore, the precision varies somewhat from one number to another, owing to the variable number of leading binary zeros; this is known as "wobble." IBM format has no infinities or NANs, and only one zero (all bits zero); it does permit non-normalized numbers. IBM also has a 64-bit double-precision format.

The last format in the figure is MIL-STD-1750A, used in military systems. It is unusual in departing from the "sign/magnitude" convention of the previous formats, using instead a 2's complement mantissa with a 2's complement exponent. (Actually, the previous formats are more accurately described as sign/magnitude mantissa with offset-binary exponents.) It has no infinities, NANs, or non-normalized numbers; it, too, has a double-precision version.

Number storage in memory

Microprocessor designers like to express their individuality by storing numbers in memory in peculiar orders. The 8086/8 (therefore the IBM PC and compatibles) stores numbers beginning with the least significant byte in the lowest-numbered memory byte; the 68000 family does it the other way around. Lots of luck!

I/O data conversion

We detoured earlier to discuss number formats in the context of our hardware interface with its packed-BCD format. What is the best way to handle the kind of 8-digit data you would get from such an interface? Depending on the type of input data, the number of significant digits, its range of variation, etc., it may be best to convert the incoming data to floating-point (for greatest dynamic range) or to integers (for best resolution) or to do some other sort of numerical massaging (e.g., taking differences from the average value, or between successive data). This might be done in the particular device's software "driver," the section of program that handles the actual input of data. In this sense the software cannot be optimized without an understanding of the hardware and what its data means. Just another reason why it is important to know your way around the wonderful world of electronic hardware!

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

MICROPROCESSORS

CHAPTER 11

The microcomputers we discussed in the last chapter are stand-alone computational systems built around a microprocessor, usually including mass storage (disks), graphics, printers, and perhaps some networking. They come complete with terminal, memory, and I/O ports, and with plug-in boards you can augment their native capabilities. They're wonderful machines for computation, word processing, computer-aided design (CAD), and even computer-aided manufacture (CAM). With commercially available hardware enhancements you can use them to program logic devices, perform as logic analyzers, or in fact serve as the front end for a variety of engineering tools.

You can have the power of a microcomputer in the instruments and systems you design, by incorporating a microprocessor and some associated circuitry. In this sort of "dedicated" application the processor executes a fixed program you've burned into ROM ("firmware"), and there generally is no mass storage (disks, tape), terminals, etc. From the outside the instrument may look quite ordinary, although a keypad often reveals its superior IQ. Microprocessor-based instruments generally deliver better performance at lower cost and with simpler construction than do equivalent instruments implemented with discrete logic chips. Furthermore, making changes and improvements is often as simple as writing new firmware. As a consequence, no competent designer can afford to ignore these versatile devices. If any further incentive is needed, we might point out that microprocessors are fun; designing an instrument with them gives you an exhilarating sense of power.

When microprocessors are designed into an instrument as dedicated devices, the designer must play a greater role in the design and programming than he would with a microcomputer system. In particular, design with microprocessors includes tasks such as choosing the type of memory (static or dynamic RAM, EPROM, EEPROM) and deciding where in "memory space" to put it, deciding the form that input/output takes (including the choice of I/O hardware, whether constructed from the conventional MSI functions we dealt with in Chapters 8 and 9, or constructed from custom LSI "peripheral

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

support" chips), and writing and debugging the dedicated software (firmware) in the context of the instrument it controls. In general, designers of microprocessor-based instruments must have a thorough knowledge of both hardware and assembly-language software techniques in order to be successful.

Most of the bus interfacing and programming concepts we introduced in the previous chapter in connection with microcomputers are directly applicable to circuits with dedicated microprocessors, and this chapter assumes a familiarity with the contents of Chapter 10. In this chapter we will begin by looking in detail at a small microprocessor with an elegant instruction set, the Motorola 68008, which is really a 32-bit processor (the 68000) with an 8-bit external data bus. After looking at its architecture and instruction set, we will show a complete design example – an analog "signal averager," complete with graphic XY display, digital serial/parallel ports, and some other niceties. No microprocessor system is complete without software, and we will show the programming necessary for this example. Along the way we will have used LSI peripheral and memory chips, and we will have some additional advice about them. The chapter will continue with a discussion of timing, data buses, and other popular processors, including the highly integrated "microcontroller" chips. Finally, we will step back and look at the overall process of electronic design with microprocessors – development systems, evaluation boards, and emulators.

A DETAILED LOOK AT THE 68008

The abundance of different microprocessor types can present a real problem to the circuit designer. Incompatibility between different microprocessor chips, both in their hardware implementation (signal lines, interfacing protocol, etc.) and in their instruction sets, is the rule in the microprocessor world. Rather than attempt to choose precisely the best microprocessor for each job, it is perhaps best to settle on a sufficiently good microprocessor, then build a good development system and gain expertise with it. This is especially true in view of the fact that software development costs and effort often exceed those of the hardware design in microprocessor-based designs.

In this chapter we will concentrate on the Motorola 68008, a junior member of the elegant and popular 68000 family of processors that are used in microcomputers like the Macintosh, NeXT, Sun, and Apollo. It is essentially identical with the 68000 (16-bit data bus, 24-bit address bus), but packaged in a 48-pin DIP with an 8-bit data bus and a 20-bit address bus. It executes exactly the same code as the 68000; the smaller data bus is entirely transparent to the user.

11.01 Registers, memory, and I/O

Registers

Figure 11.1 shows the internal registers of the 68000 (we will use "68000" to refer to features that are common to the 68000 and 68008). There are 8 *data registers* and 7 *address registers*, all of which are completely general-purpose; this contrasts with the 8086/8, in which `AX` (`AL`) must be used for I/O, `DX` for port addressing, etc. The *data registers* can hold bytes (8 bits), "words" (16 bits), or "longs" (32 bits), with the data type in any given operation specified in the assembly-language instruction itself (see below). You do your computation and byte-pushing in the data registers.

The *address registers* are used as pointers into memory or I/O in 5 of the 68000's 12 possible addressing modes; only a few arithmetic operations are allowed on address registers (add, subtract, compare,

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

move). There are no segments or segment registers; with the 68000 family you always have access to the full address space (1 or 4Mbyte for the 68008 in DIP or quad package, 16Mbyte for the 68000, 4Gbyte for the 68020/30).

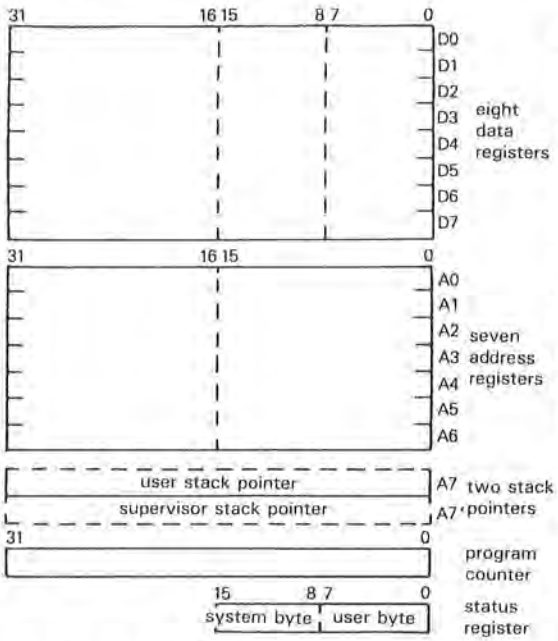


Figure 11.1. 68000/8 registers.

There is a dedicated stack pointer (actually there are two, with only one active at any given time), program counter (or "PC," called the "instruction pointer" in the 8086/8), and status register (SR). The latter holds the flags (zero, carry, overflow, etc.) and also the interrupt mask and mode bits.

Memory and I/O

Unlike the 8086/8, the 68000 processors don't have separate I/O bus signals, and they don't have I/O instructions like IN and OUT. Instead, they treat I/O just like memory, asserting a full-length address and a strobing signal. To attach an I/O port, you decode the address lines and make your port registers look like nonexistent memory. This is called "memory-mapped I/O" and is used in many microprocessors. (Of course, even with the 8086's separate I/O protocol, you could always have put I/O into memory space, using the MEMR′ and MEMW′ strobes.)

The advantage of memory-mapped I/O is that you can operate on port registers with all the instructions that normally operate on memory: MOVE replaces the solitary IN/OUT of the 8086/8; but you can also do arithmetic operations (add, subtract, rotate, compare, test), logical operations (and, or), and bit-manipulation operations (bit set, bit test) directly on ports. The only real disadvantage of memory-mapped I/O is that you have to decode a large number of address lines; in practice this is not a problem, because the large address space and small number of I/O ports mean that you can incompletely qualify the address decoding (examples later).

11.02 Instruction set and addressing

Table 11.1 lists the complete set of instruction op-codes for the 68000. To form a valid assembly-language instruction you must specify the operands (via one of 12 addressing modes) and the data type (byte, word, or long). In Motorola's assembly-language syntax, an instruction looks like

```
OPCODE.size source,destination
```

where OPCODE is from Table 11.1, *size* is either B, W, or L (byte, word, or long, respectively), and the *source* and *destination* can be registers, immediate constants, or memory. Here are some examples:

```
MOVE.W #$FFFF,D0    (immediate, register)
MOVE.B (A0),(A1)    (indirect, indirect)
ADD.L  D5,(A2)+     (direct, indirect with
                      postincrement)
BTST.B #2,$C0000    (immediate, absolute
                      long)
```

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

TABLE 11.1. 68000/8 INSTRUCTION SET

Op-code	Description
Arithmetic	
ABCD	add packed BCD
ADD	add
ASL	arith shift left
ASR	arith shift right
CLR	clear operand
DIVS	divide (signed)
DIVU	divide (unsigned)
EXT	sign extend
LSL	logical shift left
LSR	logical shift right
MOVE	move
MULS	multiply (signed)
MULU	multiply (unsigned)
NBCD	negate packed BCD
NEG	negate
SBCD	subtract packed BCD
SUB	subtract
Logical	
AND	logical AND
BCHG	bit test and change
BCLR	bit test and clear
BSET	bit test and set
BTST	bit test
CHK	check reg against bounds
CMP	compare
EOR	exclusive OR
EXG	exchange registers
NOT	bitwise complement
OR	logical OR
RESET	pulse RESET line
ROL	rotate left w/o extend
ROR	rotate right w/o extend
ROXL	rotate left with extend
ROXR	rotate right with extend
S*cc*[a]	set a byte conditional
SWAP	swap register halves
TAS	test and set operand
TST	test operand, set flags

Op-code	Description
Control	
B*cc*[a]	branch conditionally
BRA	branch always (relative)
BSR	branch to subr (relative)
DB*cc*[a]	test, decrement, and branch
JMP	branch always (7 modes)
JSR	branch to subr (7 modes)
LEA	load effective address
LINK	link stack
NOP	no operation
PEA	push effective address
RTE	return from exception
RTR	return, restore cc's
RTS	return from subroutine
STOP	stop
TRAP	trap (vectored exception)
TRAPV	trap on overflow
UNLK	unlink stack

Condition codes ("cc")

CC	carry clear
CS	carry set
EQ	equal to zero
F	never true[b]
GE	≥ zero
GT	greater than zero
HI	high
LE	≤ zero
LS	low or same
LT	less than zero
MI	minus
NE	not equal to zero
PL	plus
T	always true[b]
VC	overflow clear
VS	overflow set

(a) see "Condition codes." (b) not available for the B*cc* instruction; use BRA for "BT."

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

The first example sets 16 bits of register D0 to all 1's (the symbol "$" means hexadecimal, and the "#" signifies an "immediate" constant); the second instruction copies a byte from the memory location whose address is in A0 to the memory location whose address is in A1; the third adds the 32-bit signed integer in D5 to the 4-byte ("long") integer that begins in the memory byte addressed by A2, then increments A2 by 4; the last instruction tests bit #2 in memory location $C0000_H$, setting the Z (zero) flag (for a subsequent branch instruction) accordingly. Note that the operands are in the order *src,dest*, which is the reverse of the 8086.

In general, the 68000 permits nearly all addressing modes and operand sizes with any instruction (the 68000 Programmer's Reference Manual tells precisely what you can do; we've condensed the most useful information into Table 11.2). As a result, it is relatively painless to write good and efficient assembly code. For example, with the 8086 you have to clear out the AL register in order to test an I/O port flag, requiring 5 instructions (PUSH, IN, TEST, POP, Jcc). By contrast, with the 68000 the BTST instruction, above, followed by a Bcc test, does the same job; *no* registers are needed, because the 68000 lets you test a memory bit (and therefore a port register) directly. Furthermore, "auto-incrementing" addressing modes like "(A2)+" facilitate array operations. Even though we haven't explained all the addressing modes and instructions yet, you should be able to do this one yourself:

EXERCISE 11.1

Copy an array of $100 bytes from a table beginning at location $A0000 to a table beginning at location $A8000. You might find the instruction `BGT label` (branch if greater than zero) useful.

Addressing modes

In the examples above, instructions operate on *constants*, values held in *registers*, and values held in *memory* (or ports). The 68000 provides a nice selection of "addressing modes" by which you specify these operands. Table 11.3 lists the 12 addressing modes, which Motorola counts as 14. Here's what they mean:

Register direct

Syntax: Dn (or An)
Example: `MOVE.W D0,D1`
The operand is the contents of the named register

Immediate

Syntax: # xxxx
Example: `MOVE.B #$FF,D0`
The operand is the specified constant

Memory absolute

Syntax: xxx.W or xxx.L
Example: `ADD.W D0,$B000.W`
The operand's address is given as an immediate constant

Indirect

Syntax: (An)
Example: `SUB.W D0,(A0)`
The specified register contains the operand's address

Indirect postincrement

Syntax: (An)+
Example: `MOVE.B (A0)+,(A1)+`
Same as *indirect*, then An is incremented by *size*

Indirect predecrement

Syntax: −(An)
Example: `MOVE.W D0,-(A7)`
An is first decremented by *size*, then same as indirect

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

TABLE 11.2. ALLOWABLE ADDRESSING MODES[a]

Operation	Size			Source, count, or bit							Destination					
	B	W	L	Dn	An[b]	( )[c]	abs	PC rel	imm	SR	Dn	An[b]	( )	abs	PC rel	SR
ADD	•	•	•	•	-	-	-	-	•	-	•	•	•	•	-	-
"	•	•	•	•	•	•	•	•	•	-	•	•	-	-	-	-
ADDQ	•	•	•	-	-	-	-	-	3	-	•	•	•	•	-	-
AND	•	•	•	•	-	-	-	-	•	-	•	-	•	•	-	-
"	•	•	•	•	-	•	•	•	•	-	•	-	-	-	-	-
"	-	•	-	-	-	-	-	-	•	-	-	-	-	-	-	•
ASL, ASR	•	•	•	•	-	-	-	-	3	-	•	-	-	-	-	-
"	•	•	•	-	-	-	-	-	(d)	-	-	-	•	•	-	-
B*cc*, BSR	•	•	-	-	-	-	-	-	-	-	-	-	-	-	-	-
BCHG, BCLR,	•	-	-	•	-	-	-	-	8	-	-	-	•	•	-	-
BSET	-	-	•	•	-	-	-	-	8	-	•	-	-	-	-	-
BTST	•	-	-	•	-	-	-	-	8	-	-	-	•	•	•	-
"	-	-	•	•	-	-	-	-	8	-	•	-	-	-	-	-
CLR	•	•	•	-	-	-	-	-	-	-	•	-	•	•	-	-
CMP	•	•	•	•	•	•	•	•	•	-	•	•	-	-	-	-
"	•	•	•	-	-	-	-	-	•	-	•	-	•	•	-	-
DB*cc*	-	•	-	-	-	-	-	-	-	-	•	-	-	-	-	-
DIVS, DIVU	-	•	-	•	-	•	•	•	•	-	•	-	-	-	-	-
EOR	•	•	•	•	-	-	-	-	•	-	•	-	•	•	-	-
"	-	•	-	-	-	-	-	-	•	-	-	-	-	-	-	•
EXT	-	•	•	-	-	-	-	-	-	-	•	-	-	-	-	-
LEA	-	-	•	-	-	(e)	•	•	-	-	-	•	-	-	-	-
LSL, LSR	*(see ASL, ASR)*															
MOVE	•	•	•	•	•	•	•	•	•	-	•	•	•	•	-	-
"	-	•	-	•	-	•	•	•	•	-	-	-	-	-	-	•
"	-	•	-	-	-	-	-	-	-	•	•	-	•	•	-	-
MOVEM	-	•	•	•	•	-	-	-	-	-	-	-	(f)	•	-	-
"	-	•	•	-	-	(g)	•	•	-	-	•	•	-	-	-	-
MOVEQ	-	-	•	-	-	-	-	-	-	8	•	-	-	-	-	-
MULS, MULU	-	•	-	•	-	•	•	•	•	-	•	-	-	-	-	-
NEG, NOT	•	•	•	-	-	-	-	-	-	-	•	-	•	•	-	-
OR	*(see AND)*															
PEA	-	-	•	-	-	(e)	•	•	-	-	-	-	-	-	-	-
ROXL, ROXR	*(see ASL, ASR)*															
S*cc*	•	-	-	-	-	-	-	-	-	-	•	-	•	•	-	-
SUB, SUBQ	*(see ADD, ADDQ)*															
SWAP	-	•	-	-	-	-	-	-	-	-	•	-	-	-	-	-
TAS	•	-	-	-	-	-	-	-	-	-	•	-	•	•	-	-
TST	•	•	•	-	-	-	-	-	-	-	•	-	•	•	-	-

[a] of the most-used instructions. [b] W or L size only. [c] "( )" = all adr reg indirect modes: (A_n), $(A_n)+$, $-(A_n)$, $d_{16}(A_n)$, $d_8(A_n,X_n)$. [d] 1-bit shift. [e] except $-(A_n)$ and $(A_n)+$. [f] except $(A_n)+$. [g] except $-(A_n)$.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

TABLE 11.3. 68000/8 ADDRESSING MODES

Mode	Syntax	Address generation
REGISTER DIRECT		
Data register direct	Dn	EA = Dn
Address register direct	An	EA = An
ABSOLUTE		
Absolute short	xxx.W	EA = (next word)
Absolute long	xxx.L	EA = (next two words)
PROGRAM COUNTER RELATIVE		
PC relative with offset	d.W(PC)	EA = (PC) + d_{16}
PC relative with index and offset	{ d.B(PC, Xn.W) / d.B(PC, Xn.L) }	EA = (PC) + (Xn) + d_8
REGISTER INDIRECT		
Register indirect	(An)	EA = (An)
Register indirect with postincrement	(An)+	EA = (An); An ← An + N
Register indirect with predecrement	–(An)	An ← An – N; EA = (An)
Register indirect with offset	d.W(An)	EA = (An) + d_{16}
Indexed register indirect with offset	{ d.B(An, Xn.W) / d.B(An, Xn.L) }	EA = (An) + (Xn) + d_8
IMMEDIATE		
Immediate	#xxxx	DATA = next word(s)
Immediate quick	#x	inherent data

Notes:
EA = effective address
An = address register (A0 – A6)
Dn = data register (D0 – D7)
Xn = address or data register used as index register
SR = status register
PC = program counter
( ) = "contents of"
d_8 = 8-bit offset ("displacement")
d_{16} = 16-bit offset ("displacement")
N = 1 for Byte, 2 for Word, and 4 for Long. If An is the stack pointer, and the operand size is Byte, N = 2 to keep the stack pointer on a word boundary.
← = "replaces"

Indirect with offset

Syntax: d_{16}(An)
Example: `MOVE.L (A0),100(A0)`
The operand's address is (An) plus the 16-bit signed displacement, d_{16}

Indexed indirect with offset

Syntax: d_8(An,Xn.W [or .L]) (Xn can be either Dn or An)
Example: `MOVE.L 100(A0),100(A0,D7)`
The operand's address is (An) plus (Xn) plus the 8-bit signed displacement, d_8

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

PC-relative with offset

Syntax: d_{16}(PC)
Example: `LEA 100(PC),A3`
The operand address differs from the address of this instruction by the 16-bit signed displacement

PC-relative with index and offset

Syntax: d_8(PC,Xn.W [or .L])
Example: `MOVE.W 100(PC,D0.W),D1`
The operand address differs from the address of this instruction by the sum of the 8-bit signed displacement and the contents of Xn

A few explanations: The first two modes don't address memory; they address *registers* or *immediate constants* (constants embedded in the instruction stream; they can never be destinations, only sources). All the rest are memory-addressing modes. Memory absolute is good for I/O ports or individual memory accesses. Indirect (particularly with postincrement/predecrement) is good for arrays or the stack; furthermore, if the address is already in an address register, it's faster than absolute addressing, because the (absolute) address doesn't have to be fetched during instruction execution. The PC-relative addressing modes are particularly useful if you want to make "position independent" code, since all addressing is relative to the code itself; note that the 8- or 16-bit displacements are 2's complement (signed) integers, allowing displacements of ±127 or ±32767, respectively. Note that you cannot modify immediate or PC-relative operands (they are "nonalterable").

11.03 Machine-language representation

As we mentioned earlier, the assembly language that we have been using is not the "object code" actually executed by the microprocessor, but rather a mnemonic representation convenient for writing programs. The set of assembly-language instructions that constitutes a program must be converted to a set of binary bytes that the processor actually executes. As with the 8086, each 68000 assembly-language instruction assembles into several bytes of machine code. The op-code is always 2 bytes long, with additional words (2 bytes) following only if needed to complete the addressing mode. Depending on the instruction and addressing modes, a single instruction may be from 2 bytes to 10 bytes in length. For example, the instruction

```
ADD.W (A1)+,D3
```

assembles to the minimum length of 2 bytes, namely $(D6\ 59)_H$, with the register numbers and addressing modes encoded (along with the operation) into the 2-byte instruction. However, the instruction

```
MOVE.W #$FFFF,$A0000
```

assembles to an 8-byte instruction, namely $(33\ FC\ FF\ FF\ 00\ 0A\ 00\ 00)_H$, with the operation and addressing modes specified in the first 2 bytes, the immediate constant in the next 2 bytes, and the absolute long destination address in the last 4 bytes.

The CPU, of course, is genetically programmed to know how to interpret this resulting machine code. Looking at the construction of a particular instruction op-code may help you, too, to get a glimmer of how a CPU thinks. Figure 11.2 shows the anatomy of the 68000's best-selling instruction, "`MOVE`." Let's walk through it. The two leading zeros identify the instruction as a `MOVE` operation (almost), with the next two bits defining the operand size (as listed in the figure). It's interesting to note that, since the bit pair 00 is not a legal size, 0000xxx..xx is not a `MOVE` (don't worry, it's not wasted – Motorola used this combination for some other instructions). The next 6 bits tell the addressing mode and register (if any) for the destination, and the last 6 do the same for the source; the figure shows you how to form those bit

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

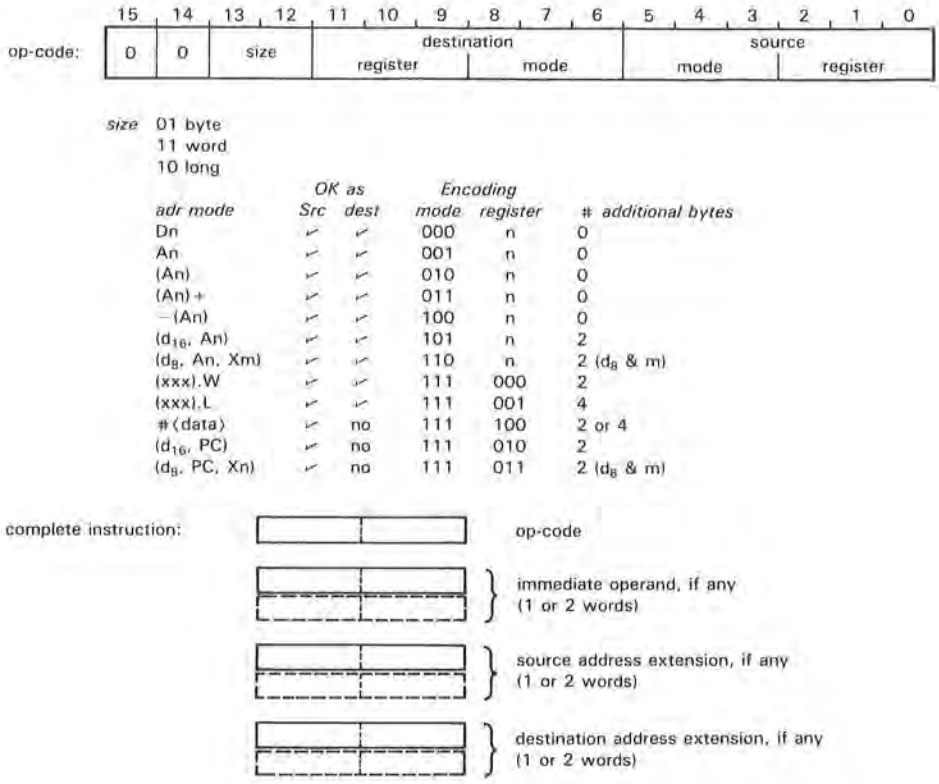


Figure 11.2. The MOVE instruction, dissected.

encodings. Note that the last 5 addressing modes, which don't actually use a register, all share the only remaining mode number (111) and are distinguished by fake "register" numbers. If the addressing mode of either operand requires additional information (immediate data, absolute addresses, or displacements), extra bytes get attached to the instruction, as indicated in the table.

It's interesting to note that the 68000 squanders $1/4 \times 3/4 = 19\%$ of its instruction set on MOVE, in order to accommodate all addressing modes for both source and destination. Motorola couldn't afford to be that profligate with the remaining 50-odd instructions in Table 11.1, so they limited the addressing mode options. For example, using Motorola's term ⟨ea⟩ for the full set of addressing modes, you can form the instructions

```
ADD <ea>,Dn
```

or

```
ADD Dn,<ea>
```

but you can't do the fully general

```
ADD <ea>,<ea>
```

In real life you always use an assembler (running on a computer or a microprocessor "development system") to do the dirty work of constructing these instructions. In order to be able to say we really understand it, however, let's try our luck at "hand assembly": Let's do a

```
MOVE.W #$3FFF,(A1)+
```

It's easy – size is 11 (word); destination is mode 011, register 001; source is mode 111, "register" 100. So the op-code is

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

00 11 001 011 111 100, or $32FC_H$

and the complete instruction is therefore

32 FC 3F FF

It is our belief that if we persist in abstract discussions of the instruction set and addressing modes, you may well shut the book and never open it again! So let's show a simple programming example instead, then move on to the 68008 bus signals. At that point we'll be able to do a complete 68008 circuit design, with software.

As an illustration of 68000 instructions and addressing, Program 11.1 shows two ways to copy a table of 100_H bytes beginning at location $8000 to the memory space just above it (beginning at $8100). In the first program, we used indirect addressing with offset to do the memory-to-memory move (a nice feature of the 68000 not present in the 8086), followed by pointer increment and counter decrement, and finally a test. With a 10MHz clock, the loop takes 6.2μs, and the table is moved in 1.6ms. The second program uses a second address register to point to the destination, which executes faster and also lets us use postincrement, thus eliminating the ADDQ instruction. We also used the more efficient (but tricky) "decrement and branch" (DBcc) instruction. This loop executes almost twice as fast (3.4μs/loop, 0.87ms total).

EXERCISE 11.2

Write a program to calculate the sum of 16-bit words in a table that begins at $10000. Assume that the length of the table, in words, is given as the first table entry (which should not be part of the sum); assume also that the sum will not overflow.

EXERCISE 11.3

Write a program to reverse the order of bytes in a table of $100 bytes beginning at $1000. A straightforward (but slow) way is to reverse the order while copying to a temporary array, then copy the reversed version back. A faster method does the reversal "in place" (but be careful not to step on your own feet as you go). Program both methods.

Program 11.1

```
                                    ;move $100 byte table
                                    ;first method
        MOVE.L   #$8000, A0         ;address of table
        MOVE.W   #$100, D0          ;size of table
LOOP:   MOVE.B   (A0), $100(A0)     ;move byte
        ADDQ.L   #1, A0             ;increment pointer
        SUBQ.W   #1, D0             ;decrement counter
        BHI      LOOP               ;loop until done
          o
          o
          o


                                    ;second method
        MOVE.L   #$8000, A0         ;source table
        MOVE.L   #$8100, A1         ;destination table
        MOVE.W   #$FF, D0           ;size-1
LOOP:   MOVE.B   (A0)+, (A1)+       ;move byte
        DBF      D0, LOOP           ;loop until done
          o
          o
          o
```

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

11.04 Bus signals

If you understood the IBM PC's bus signals, you won't have any trouble with the 68008's, which are similar. We've collected them in Table 11.4 (same format as Table 10.1) and Figure 11.3. The best way to learn about them is to look at the same progression as before: programmed data transfer ("programmed I/O"), interrupts, and DMA. It won't take long.

Programmed data transfer

Programmed data transfer is shown in Figure 11.4; we have also included some signals (CLK and AS′) that you can generally ignore in your circuit design. The 68008 uses a single data strobe (DS′) and a read/write direction line (R/W′), rather than the PC's pair of strobes (IOR′, IOW′).

For WRITE, the CPU asserts R/W′ LOW, and asserts ADDRESS and DATA, then (allowing some setup time) asserts DS′ LOW. The recipient (memory or I/O) latches the data, which (unlike the PC) is guaranteed valid before the leading edge of DS′, and acknowledges by asserting DTACK′. The CPU completes the cycle by disasserting DS′ and then (allowing some hold time, in case the recipient is using transparent latches) disasserting ADDRESS and DATA. Thus the data is guaranteed good during DS′, and for a short time on either side. For READ, the only differences are that the CPU holds the R/W′ line HIGH (to indicate a read cycle), and also asserts DS′ one CPU clock earlier, to give the data source more time to respond to the call for data. That data must be valid before the end of DS′, with actual timing as shown.

TABLE 11.4. 68008 BUS SIGNALS

Signal name	Number	Active	Type[a]	Direction CPU↔BUS	Function
A0–A19	20	H	2S[b]	→	address
D0–D7	8	H	3S	↔	data
AS'	1	L	2S[b]	→	address strobe
DS'	1	L	2S[b]	→	data strobe
R/W'	1	–	2S[b]	→	direction (read/write')
DTACK'	1	L	OC	←	data xfer acknowledge (handshake)
IPL0–IPL2'	2	L	in	←	interrupt request inputs
FC0–FC2	3	H	2S[b]	→	indicates type of cycle
VPA'	1	L	in	←	autovector (or 6800-type I/O)
BERR'	1	L	in	←	bus error signal to CPU
RESET'	1	L	2S[c]	↔	reset
HALT'	1	L	2S[c]	↔	stop
BR'	1	L	OC	←	bus master request
BG'	1	L	2S	→	bus master grant
E	1	H	2S	→	6800-type I/O enable
CLK	1	–	in	←	CPU clock (10MHz typ)

[a] 2S - 2-state (totem-pole); 3S - 3-state. [b] released if not bus master. [c] dual-function: asserted (2-state) by CPU (as output); can be overdriven externally (as input).

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

The business of DTACK′ deserves some explanation. The 68008 bus is what we called "default-wait" (asynchronous) in Section 10.14: Having asserted DS′, the CPU waits for the (wired-OR) acknowledge signal DTACK′ from the addressed device before completing the cycle. If DTACK′ comes back before the end of S4, no wait states are inserted, and the timing is as shown in Figure 11.4; but if DTACK′

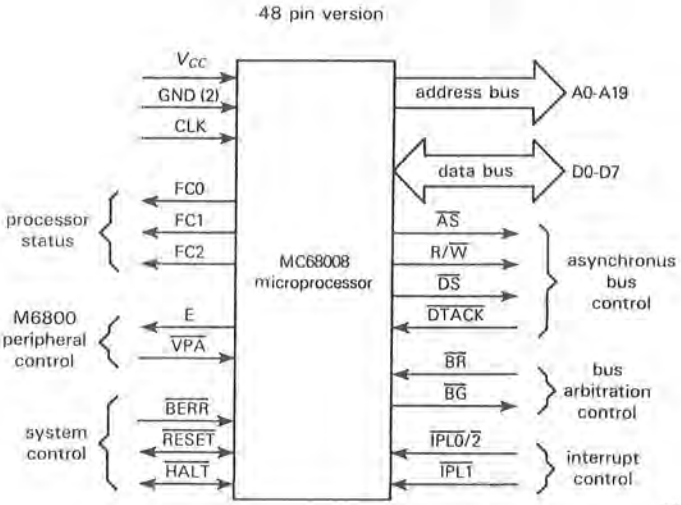


Figure 11.3. 68008 bus.

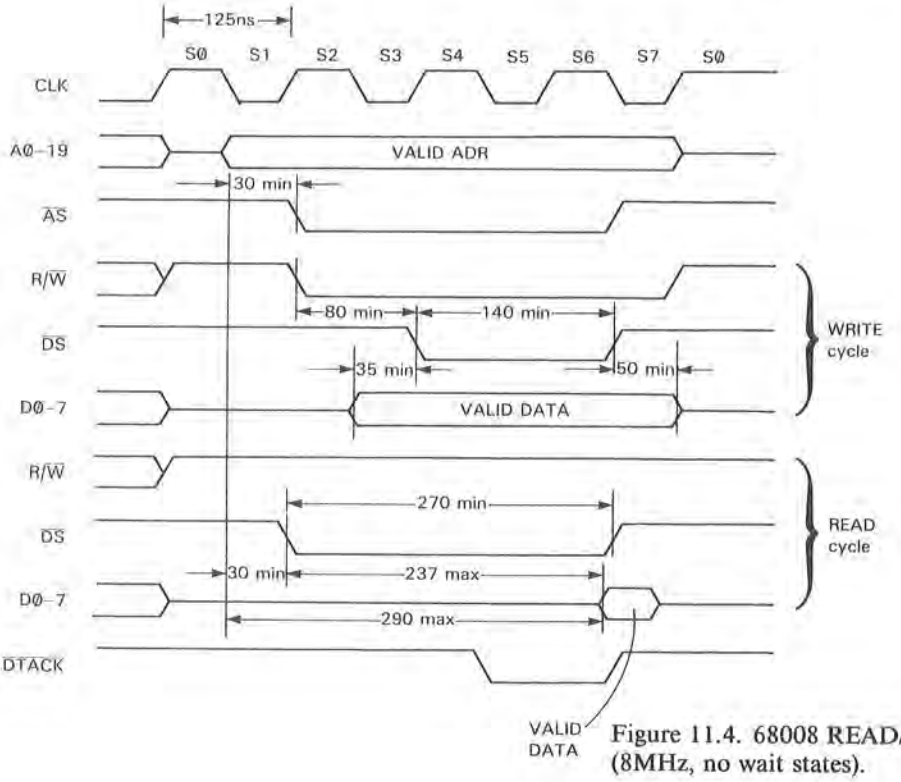


Figure 11.4. 68008 READ/WRITE cycles (8MHz, no wait states).

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

is delayed, the CPU just holds all its outputs stable (inserting "wait states" after S4) until it sees DTACK′, after which it finishes the cycle with S5–S7. Unless the addressed device is very slow, no wait states are needed, so the addressed device should assert DTACK′ as soon as it recognizes its address (it can just assert DTACK′ based on its address decoding, or, if it's compulsive, AND that with AS′, which signals a valid address). In fact, if all devices on the bus are fast, you can live dangerously and tie DTACK′ permanently LOW, thus disabling wait states entirely; this explains the title of a magazine devoted to high-performance 68000-family applications: *DTACK Grounded.*

All this sounds complicated, but in fact interfacing to the 68008 is simple. Figure 11.5 shows the simplest sort of read/write port. The address lines are decoded and qualified with DS′ and R/W′ to generate the *D*-register clock (write) and three-state enable (read). DTACK′ is asserted whenever this port is addressed, since you never need wait states for devices as fast as octal registers; we've used the usual trick of making a three-state driver into an open-collector driver. Note that we've used the trailing edge of DS′ to clock the *D* register; that is because the 35ns minimum "valid data" to the leading edge of DS′ (see Fig. 11.4) is uncomfortably close to the setup time requirement of many octal registers (e.g., both LS and HCT families have t_{setup} = 20ns, min). In fact, if the data bus is *buffered* (for example with '245 octal bidirectional buffers), the additional delay of DATA relative to DS′ might violate minimum setup time for the

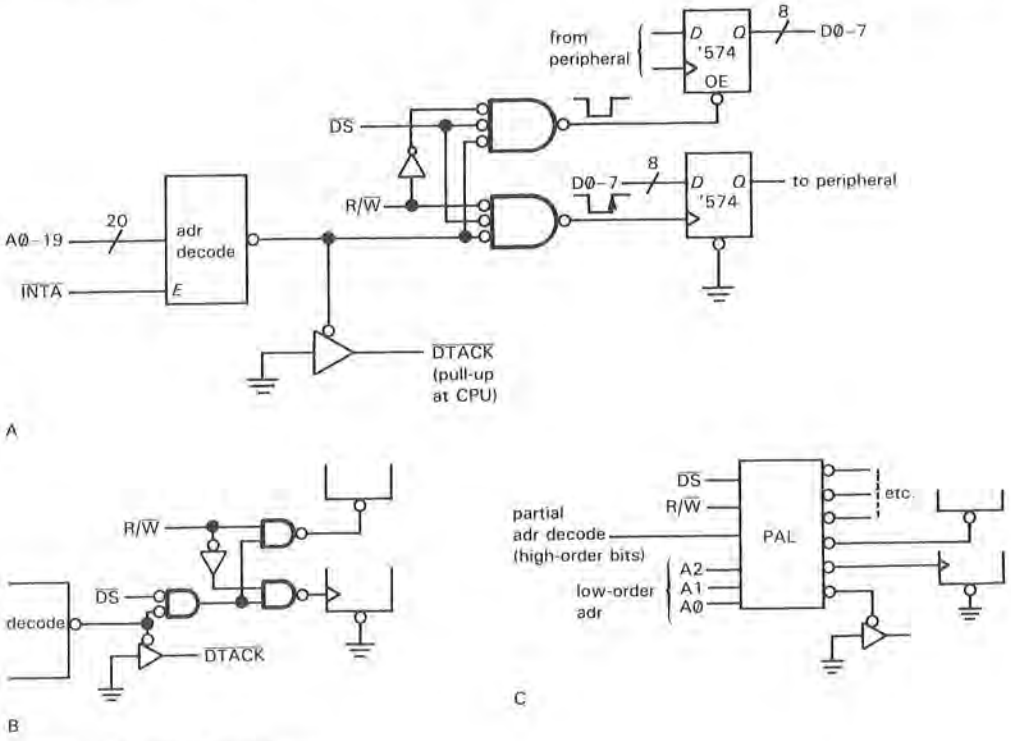


Figure 11.5. Parallel I/O port.
A. Basic implementation.
B. As constrained by available gate types.
C. Implementation with a PAL.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

'574. By using the trailing edge, we gain an extravagant 140ns of additional setup time. (Alternatively, we could have used a transparent latch, e.g., a '573, which freezes the data present at the trailing edge of its latch enable; it requires setup and hold times of about 15ns, min.)

The alternative logic forms shown in Figure 11.5 deal with some practical realities. As it turns out (see Table 8.2), in most logic families you can't get 3-input OR gates! One solution is to use 2-input gates, as shown. A more modern approach is to swallow all the gate logic into a combinational PAL; in addition to reducing the chip count, the PAL's extravagant gate content lets you generate clocking and enabling signals for additional peripheral ports, all with one chip.

In this example we have anticipated the next section by showing one minor detail: During interrupts (discussed next), the 68008 executes an interrupt acknowledge cycle, which looks like a READ from the very top of memory (A4–A19 all 1's). If you actually place any memory or registers there, you must disable it during the interrupt cycle, signaled by the "function code" bits FC0–2. The following section explains all.

Interrupts

The 68008 allows both autovectored and fully vectored (acknowledged) interrupts (see Section 10.11 if you've forgotten how these work), using the bus signals in the second group in Table 11.4. In either case you request an interrupt by pulling some combination of the two prioritized request (IPL′) lines LOW. The two lines define three levels of interrupt (the fourth state – both lines HIGH – corresponds to no interrupt at all). These lines are similar to the IRQ lines of the PC bus, but since they are *level*-sensitive, you can put multiple interrupting devices on each level. (It's worth noting that the 68000, and some versions of the 68008, have 3 IPL′ lines, defining 7 levels of interrupt.)

When the CPU sees an interrupt request (at least one IPL′ line LOW), it executes an acknowledge cycle (Fig. 11.6) to identify the source of the interrupt: It asserts the interrupt level on A1–A3 and asserts A4–A19 HIGH; it also asserts the function code lines FC0–2 all HIGH. Then it does a read cycle (i.e., R/W′ HIGH). Your external circuitry now determines the kind of acknowledge cycle: either autovector (jump according to the IPL′ level) or acknowledge (jump according to a vector asserted onto D0–D7 by the interrupting device).

Autovectoring is the simplest (Fig. 11.7). External circuitry detects the acknowledge cycle by looking at FC0–2, and asserts the VPA′ input, coincident with AS′. The CPU then jumps to the service routine corresponding to the IPL level of the interrupt, using vectors (i.e., 32-bit addresses of the routines) stored in absolute locations $68, $74, or $7C. If you have no more than three devices that can interrupt, autovectoring is fine. In fact, even with more interrupters, all is well as long as you poll the status registers of all eligible devices (i.e., all known devices hooked to the interrupt level being serviced) to find the culprit. It's only when you have many possible interrupters (an unlikely situation in a small gadget with a dedicated 68008), and also insist on minimum latency, that you should resort to vector acknowledgment.

Full interrupt acknowledgment works like this: First, leave the processor's VPA′ permanently disasserted (HIGH). Then arrange circuitry so that each interrupt-capable device can assert a unique vector onto the data lines when the processor (a) does a READ with FC0–2 HIGH, (b) at the IPL level (as seen on A1–3) at which the device interrupted. Your circuitry must ensure that no more than one interrupting device asserts its vector, even if several devices interrupted

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

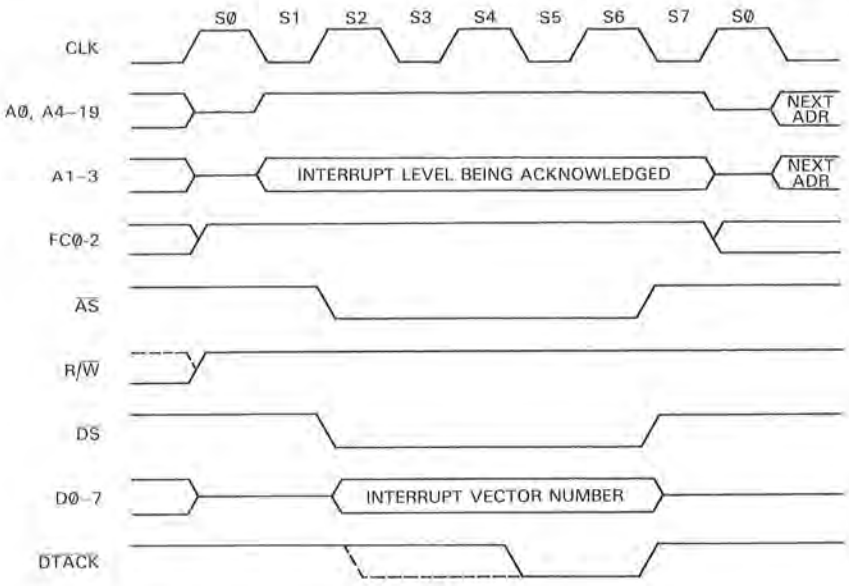


Figure 11.6. Interrupt acknowledge cycle.

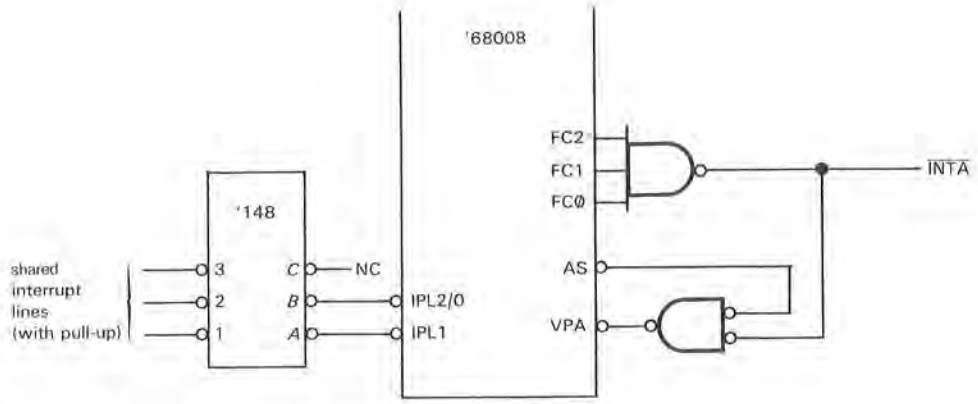


Figure 11.7. Autovectored interrupt.

simultaneously. One method for doing this involves a daisy-chained serial interrupt priority signal, INTP, as discussed in Section 10.11; it guarantees that only the device (of proper IPL level) electrically closest to the CPU acknowledges the interrupt, even if more than one device interrupted on the same IPL level.

An elegant alternative is shown in Figure 11.8. It avoids the need for awkward daisy-chaining, requiring instead a request line from each interrupt-capable device. The state of those request lines is latched at the start of each bus cycle (via the leading edge of AS) and fed to a priority encoder (which generates the binary address of the highest-numbered asserted input, see Section 8.14). The encoder also generates an output (GS′) if *any* inputs are asserted; we use this to initiate a CPU interrupt. For

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

simplicity we put all device interrupts on a single IPL level. The CPU responds to the interrupt by stacking the return address, then initiating the acknowledge cycle of Figure 11.6, during which our circuit asserts both the priority-encoded vector and DTACK′. The CPU now does a vectored jump to the appropriate handler.

This scheme is simple to implement, and for the 68000 family it is actually faster than autovectoring. Furthermore, it is relatively painless to expand the number of interrupters in multiples of 8 with additional '574s and '148s. This scheme does require a dedicated (not bused) line from each peripheral device; although that spoils the symmetry of the data bus, it is probably preferable to the serial daisy-chained scheme, which malfunctions completely if you forget to install jumpers over unused slots! In fact, recent computer buses (e.g. the NuBus in the Macintosh II) seem to be moving toward slot-dedicated interrupt lines.

One interesting (and important) point about this circuit: You might wonder why we need to latch the individual request inputs, since each interrupting device generates its request as a latched bit already (see, e.g., Fig. 10.12). The reason is subtle. Interrupting devices are, in general, asynchronous with respect to the CPU clock and can generate interrupts at any time. If we omitted the latch, and a second peripheral device decided to request an interrupt at precisely the moment the CPU was reading the interrupt vector from the original interrupter, the asserted vector would change in midstream (during the CPU's interrupt acknowledge vector fetch), with unpredictable results. You might object that this scenario is unlikely, and indeed it is; but it *can* happen, and in fact you can estimate about how often. By forcing a decision time for everyone's "intention to interrupt" at the start of each bus cycle, we eliminate the problem (actually, there is still a tiny possibility for an error, due to the "metastability problem"; see Section 8.17 if you need something to worry about).

EXERCISE 11.4

Assume that we choose to live dangerously by omitting the '574 register and that we have two asynchronous devices interrupting at the rate of 1000 interrupts per second each. Assume that the vector fetch cycle has a critical time window of one ns during which a change of asserted vector will lead to an incorrect fetch (i.e., the CPU will read a vector number different from both asserted vectors). Estimate how often the CPU will vector into the wild blue yonder (i.e., crash).

One last point about our circuit. The 68000 series of CPUs has an instruction called HALT, which kills all bus activity but permits restarting via an interrupt (or, of course, a complete reboot). Unfortunately, our circuit won't let you restart via an interrupt (why not?). Thus, you must either abstain from HALTing or use some other timing information (perhaps a variation on CLK) to latch the interrupt requests.

The 68000 allows devices to assert 192 distinct interrupt vectors, numbered 40_H–FF_H; the corresponding jump addresses (i.e., the addresses of the corresponding service routines) are stored in memory locations 100_H–$3FF_H$.

Direct memory access

With the 68000, DMA is not mediated by an on-board DMA controller, with address counters, etc., as it is on the PC bus. Instead, the 68000 is willing to relinquish the bus entirely, in an orderly transfer of bus mastership; the new master (which might be another 68000, or just a humble peripheral interface) can then do anything it wants, including (but not limited to) the classic DMA function of sending data to and from memory.

To become master, any device can make a "bus request" by pulling the wired-OR line BR′ LOW. The CPU really takes this

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

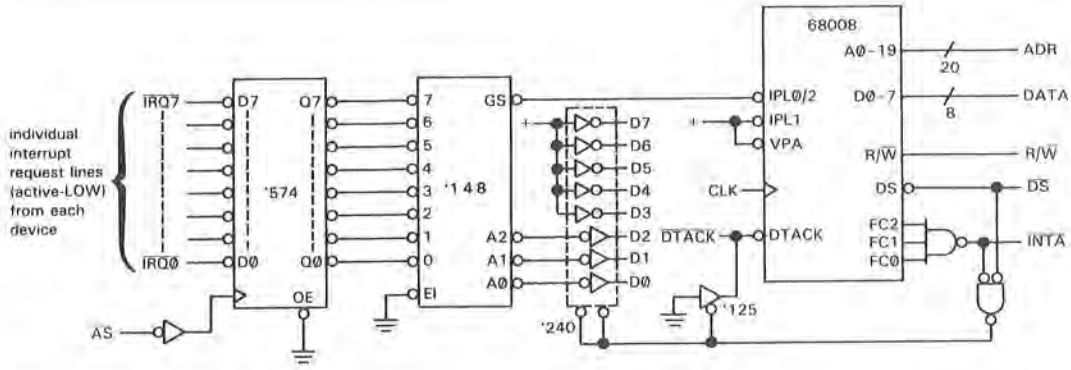


Figure 11.8. Fully vectored interrupt.

seriously, responding as soon as humanly(?) possible by asserting the "bus grant" line BG′ LOW. The CPU also releases control of all bus lines (except BG′), including address, strobes, and other control lines indicated by a superscript "b" in Table 11.4. The external device is now in charge, and it stays that way until it releases BR′, whereupon the CPU regains mastership. The external master is obliged to run the bus under the same rules the CPU normally does, so other bus members don't get confused. In fact, they won't even know anything unusual is happening, unless they happen to look at BR′/BG′.

If there is more than one external bus-master candidate, they have to sort things out ("arbitrate") among themselves. Note, too, that the CPU holds onto control of BG′, thereby retaining a measure of dominance.

Remaining bus signals

Here's what the remaining signals in Table 11.4 are for:

CLK. This is the CPU clock input, which we showed in Figures 11.3 and 11.4. We recommend using inexpensive commercial crystal oscillators in DIP packages for microprocessor clocks, available from companies like CTS, Dale, Motorola, Statek, and Vectron. The 68008 likes a symmetrical clock waveform, which is best generated by dividing the oscillator output with a toggling flip-flop. Maximum clock speeds are usually indicated in microprocessor part numbers (memory, too): Current versions of the 68008 go to 10MHz (MC68008P10). Two-byte instructions generally execute in four clock periods (as in Fig. 11.4), while instructions with the fancier addressing modes have to do more memory fetches and can take up to 70 clock periods or so.

BERR′. You assert this input to tell the CPU that something has gone wrong on the bus. For example, the CPU will be happy to wait forever if no one asserts DTACK′ in response to a data strobe; this could happen if a program tries to access nonexistent memory. Asserting BERR′ causes an interrupt-like jump (officially called an "exception") to a software handler. Look ahead to Figure 11.10 for a simple BERR′ circuit implementation.

RESET′ and HALT′. These signal lines are unusual in serving both as inputs (to reset or halt the processor) and as outputs (by which the CPU can initialize the system). Look again at Figure 11.10 for a straightforward way to handle these lines.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

E (Enable). This output signal sounds important, but isn't. It (in combination with VPA′) makes it easy to hook older 6800-series peripheral chips (designed for the synchronous, and relatively slow, 6800 8-bit microprocessor) directly to the 68008. Otherwise you just ignore it.

A COMPLETE DESIGN EXAMPLE: ANALOG SIGNAL AVERAGER

In the following sections we will design a complete 68008-based instrument – an analog "signal averager" (a subject we'll discuss further in Section 15.13). Our example will include the CPU circuitry (with its DTACKs, BERRs, and so on), memory (both RAM and ROM), and plenty of interfacing: A DIP-switch and LED array, serial and parallel ports, calendar clock/timer, A/D and D/A converters, and a solid-state relay for switching ac loads. In fact, we decided to put a little bit of everything into our design. So it's really a general-purpose microprocessor module, and the software makes it into whatever instrument you want.

We'll take you through the hardware design, pointing out how we chose the parts and designed the circuit. You'll learn how to choose and connect memory and peripherals and put everything in memory space in a sensible way. Once done with the hardware, we'll talk about the overall programming and write some software "modules" (sections of code) to handle the interesting tasks. We won't bore you with every line of code, though, since a design like this includes tedious (but essential) routines, for example to get setup commands from a keyboard. Finally, we'll analyze the instrument's performance – the flexibility that we got by using a microprocessor in the design, and the speed limitations that it implies.

11.05 Circuit design

Block diagram

Figure 11.9 is a block diagram, and Figure 11.10 the schematic, of the microprocessor instrument. Look first at the block diagram, which shows the devices connected to the bus. Viewed in terms of micro*computers*, the memory at first seems lopsided, with four times as much ROM as RAM. But it makes sense when you realize that for a dedicated instrument all the programming and tables reside in ROM, not RAM, which is used only to buffer data and hold temporary results of computations. Besides, the manufacturers of UV-erasable programmable ROMs ("EPROMs") have been discontinuing the smaller ROMs as they get better at their trade; it's hard to find EPROMs smaller than 8K×8 nowadays. In any case, the amount of memory shown is the minimum you could use (one ROM chip and one RAM chip); it's easy to add more, as you'll find out by working the problems.

Next in line on the bus: A calendar clock. *Calendar clock??!!* Isn't that just a luxury, for people too lazy to look at their watch when logging on? No, it's essential for any instrument that might be called upon to make periodic measurements, or keep track of when data came in, or a host of other timekeeping jobs. You can program the calendar clock to make periodic interrupts, at rates from 100 per second to once per day; you can also use it as an alarm clock (no bell, of course; it interrupts instead!), settable to any time in the next hundred years. For our signal averager we'll use the timers in the 8536 parallel port, but it's nice to have the calendar handy, anyway.

The 8530 serial port chip is a high-performance 2-channel USART (universal synchronous/asynchronous receiver/transmitter; see Section 10.19), complete with a pair of on-chip baud-rate generators. It's a very nice chip that can do all the usual

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

RS-232 asynchronous tricks and also has full capabilities for "SDLC/HDLC" synchronous protocols, which include error checking, clock recovery, frame synchronization, etc.; probably overkill for the job, but what the heck. The 8536 is a companion timer and parallel port from the friendly folks at Zilog; it is a powerhouse, also, with a zillion modes of operation. For example, each of its 20 lines can be programmed as either input or output, normal or inverted; each output can be either open-drain or totem-pole, while each input can be normal or "ones-catching" (a momentary HIGH pulse sets an input register). The data sheet goes on and on (26 pages), astounding you with the miracles it can perform.

Moving to the top row in the block diagram, the LEDs are a simple array of 8 lights that are handy for indicating what's going on; they're also great during debugging, if all else fails. We've piggybacked a solid-state ac relay onto one of the latched LED outputs, so you can control some big machine. For example, you could make a temperature-stabilized bath by using an A/D input to sense temperature, and the ac relay to turn on a heater. We'll give you an opportunity to show your stuff with an exercise along these lines. The DIP switch is one of those little 8-station switches, useful for indicating configuration information, e.g., you can tell the processor which serial port (and what baud rate) to use at power-on. Finally, we've hooked up a pair each of A/D and D/A converters so that the instrument can live in an analog world.

Circuit details

Now for the fun part. Let's go through the circuit design (Fig. 11.10).

CPU

CLK. The 68008 needs a CPU clocking signal (CLK), a logic-level square wave in the range 2MHz to 10MHz. The upper limit is set by the intrinsic speed of internal gates and registers; currently you can get 68008s specified for 8, 10, or 12.5MHz maximum clock frequency. The lower limit is dictated by the fact that the CPU uses *dynamic* registers, which have to be periodically "refreshed" because the data is stored on charged capacitors, not flip-flops. The speed of computation is proportional to the clock frequency,

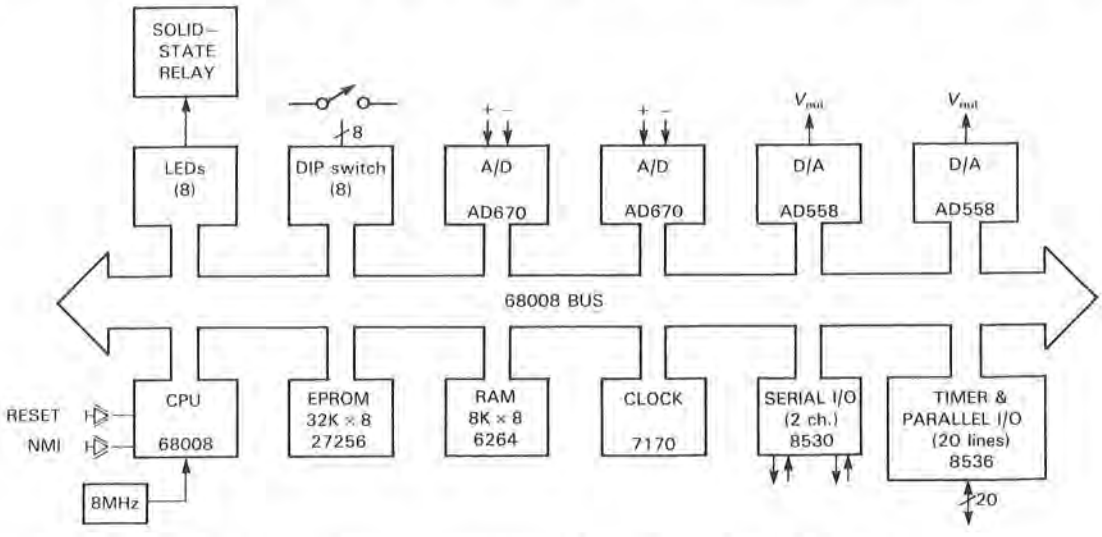


Figure 11.9. General-purpose microprocessor-based instrument – block diagram.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

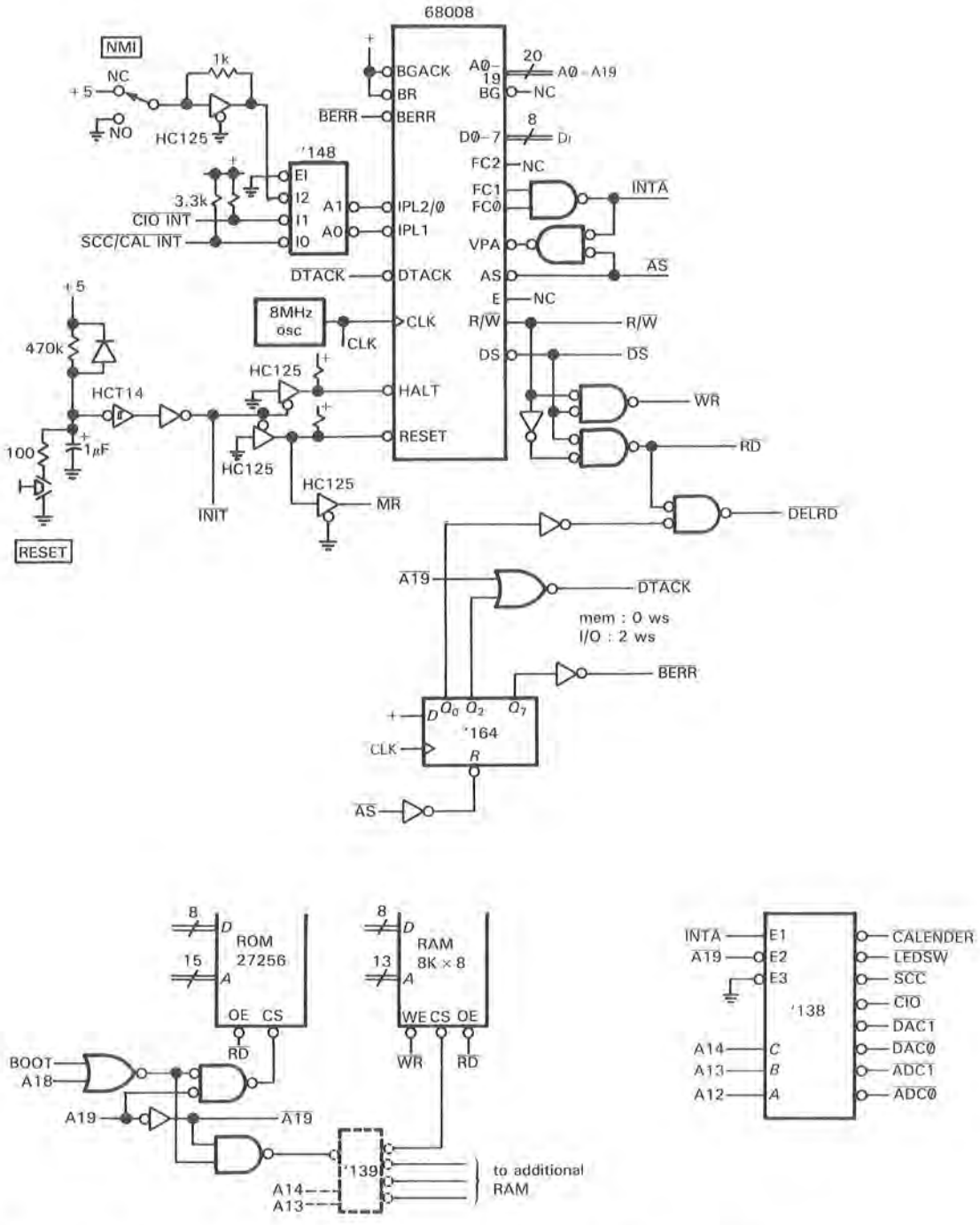


Figure 11.10. General-purpose microprocessor-based instrument – schematic.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

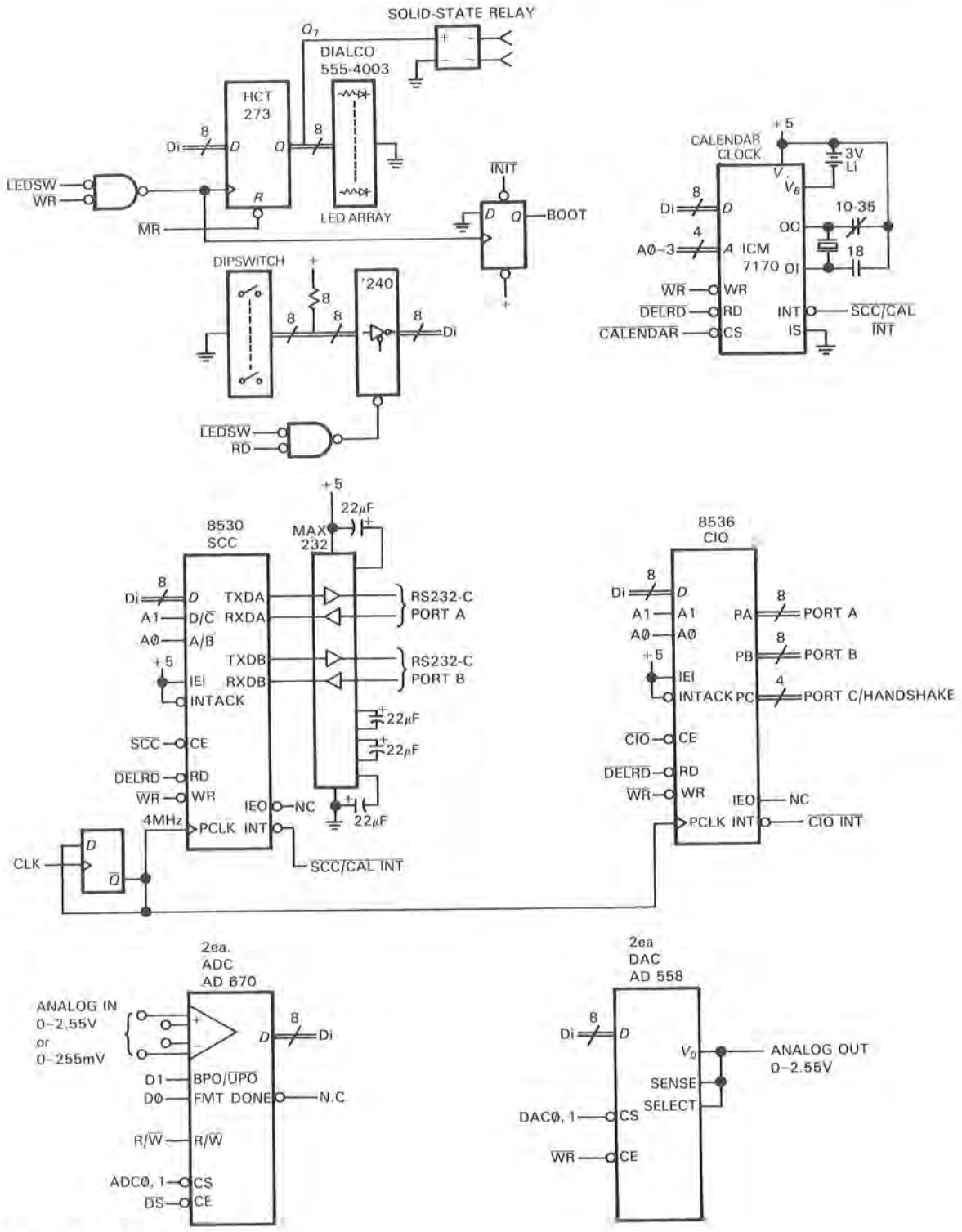


Figure 11.10 (*cont.*)

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

so you generally want to go as fast as you can; the disadvantages of high clock speeds are (a) greater timing demands on memory and peripherals, (b) higher price, and (c) greater power dissipation, particularly for low-power CMOS CPUs and peripherals. You usually don't care about power, except in battery-powered instruments; see Chapter 14. We used an 8MHz clock frequency because with that choice we can use the same clock (divided by two) for the serial port chip; otherwise we would have to use a separate oscillator for the USART, or run at reduced baud rates.

RESET, interrupts, strobes. To boot the 68008 ("boot" means "bootstrap," as in "pull yourself up by your bootstraps"; i.e., start it up from nothing) you assert both RESET′ and HALT′ (these are bidirectional; you must use open-collector with pull-up). We implemented a simple power-on boot with an RC and Schmitt trigger, paralleled with a push button for manual reboot. Note the diode for fast discharge during short supply glitches; a better power-on reset circuit would use something like the MAX692 "microprocessor supervisory" circuit, with well controlled reset output. The line we named MR′ is asserted both at boot and (for 128 clock cycles) when the CPU executes a RESET instruction, while our INIT′ line gets asserted only at boot.

We opted for autovectoring in this simple system; the AND of FC0 and FC1 indicates an interrupt acknowledgment cycle, during which we are therefore obliged to assert VPA′, coincident with address strobe AS′. We will also use our INTA′ signal to inhibit normal I/O decoding (see below). The 68008 permits three levels of autovectored interrupt: We therefore combined, via wired-OR, the slow serial port and calendar interrupts at the lowest level (IPL1) and put the latency-sensitive timer (called a "CIO" chip) interrupt at the next higher level (IPL0/2). We reserved the highest "nonmaskable" level (asserting both IPL lines) for a push-button ("NMI," nonmaskable interrupt), so you can always get the board out of a hung state during program debugging.

We used a few gates to generate a (RD′, WR′) pair of strobes from the (DS′,R/W′) strobe/direction pair that the 68008 provides. They will come in handy for several of the Intel-style peripherals that prefer separate strobes.

DTACK′, BERR′, and slow peripherals. Finally, we used an 8-bit parallel-out shift register ('164) as a state machine to generate several necessary signals in sequence. The shift register is held reset until the CPU asserts AS′, which signals the beginning of a bus cycle (Fig. 11.4), whereupon 1's begin propagating down the register, one stage per CPU-clock rising edge. The Q_0 output is used to generate a delayed RD′ signal for two fussy peripherals (SCC and CIO), as we'll explain shortly. Several I/O devices are slow and require wait states; we therefore used shift register output Q_2 to generate a delayed DTACK, resulting in two wait states for all I/O ports (in our scheme, all I/O is memory-mapped above $80000, i.e., A19 asserted), and no wait states for memory (A19 not asserted). We're in trouble if a 1 ever propagates as far as the last stage of the shift register, since all bus cycles should end (with disassertion of AS) long before that. Therefore we've used the last stage (Q_7) to assert BERR, which forces a vectored jump (through $08) rather than letting the CPU hang forever. Such a bus "time-out" signal is particularly important in a general-purpose computer, where a CPU access to a nonexistent peripheral would otherwise crash the machine.

Memory

When you boot a 68008 (by asserting RESET and HALT), it goes to the bottom of memory to find two crucial addresses:

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

the 32-bit *starting address*, stored in memory bytes $04–$07, and the initial value for the *stack pointer*, stored in bytes $00–$03. Having read these addresses, it initializes the stack pointer, then jumps to the starting address.

Because these low-memory locations are read by the CPU before any other program activity, you've got to have nonvolatile memory down there, i.e., memory that retains its information when power is shut off completely. The most obvious choice is EPROM ("erasable programmable read-only memory"; see Section 11.12), the inexpensive, UV-erasable, non-volatile byte-wide memory that you can recognize by the little glass (actually, quartz) window on the top of each package. EPROMs take a half hour to erase, and a minute or so to program. They come in sizes up to a megabit, and they retain their data longer than the lifetime of the instruments you put them in. The only snag with putting EPROM in low memory is that the various vectors (from interrupts, bus errors, and other "exceptions") also live there, and you like to be able to change them under program control.

You could do that by using a variant of EPROM called "EEPROM" ("electrically erasable programmable read-only memory"). You could also use a two-stage process in which the vectors reside permanently in EPROM, pointing to a set of jumps (a "jump table") in writable memory (RAM, see next). But there's a cleverer way: You arrange things so that at bootup there is ROM at the bottom of memory, but later it is replaced (under program control) by ordinary writable memory, called RAM ("random-access memory," which always means addressable read/write memory).

Look again at Figure 11.10. We've used a 27256 EPROM, which is moderate-sized by present standards, and organized as 32K×8. It has 15 address inputs, 8 three-state data outputs, a chip-select input (CS′), and an output-enable input (OE′). Each addressed byte (written onto the chip previously by an EPROM programmer, and now unchangeable) is asserted onto the data lines only when both enables are asserted. The usual scheme is to assert CS′ as soon as possible with address-decoding logic, then strobe OE′ with the subsequent read pulse. In our case, memory (ROM or RAM) is enabled only if A19 is disasserted; i.e., memory resides in the lower half of address space. In addition, ROM is enabled only when (a) A18 is asserted or (b) the BOOT bit (set at start-up, cleared under program control) is set. RAM also resides in the lower half of address space, but is enabled only when ROM is not. So when you boot the CPU, the BOOT flip-flop is set, and ROM temporarily lives in address space $0000–$7FFF; RAM is nonexistent. The ROM also has a dual existence up in its real home at $40000–$47FFF. The first 8 bytes of ROM are cleverly coded to make a jump into its continuing boot-up code in the higher address space, where (among other things) it clears the LED port (address $86000). Writing to the LED port has the side effect of clearing the BOOT flip-flop, causing RAM to replace the temporary low-memory image of ROM. To make this precise, here are the first 16 bytes of ROM needed to work this trick:

```
0000: 00 00 20 00                  ;initial stack ptr
0004: 00 04 00 08                  ;starting address,
                                     in "real" ROM
0008: 13 FC 00 00 00 08 60 00      ;MOVE.B #0,$86000
                                     - clear boot
```

Note that the last instruction executes at address $40008, due to the starting address fetched from $0004.

The wiring of the 8K×8 RAM chip is simple: It receives the lower 13 bits of address (8K) and is enabled when A19 is

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

disasserted and ROM is disasserted. The RD′ and WR′ strobes are tied to output enable (OE′) and write enable (WE′), respectively. For the time being, imagine that the extra decoding logic shown in dotted outline is omitted. Then RAM resides at the bottom of address space, except during the boot-up, when it is replaced by the temporary image of ROM.

Now there's a peculiar thing happening with our address decoding. Look at RAM. We've *ignored* address bits A13–A17! So, for instance, the memory byte addressed at $0000 has many aliases – you can find it at $2000, $4000, and in fact at any address that has zeros in bits A0–A12 and A18–A19. It makes multiple appearances in address space. To "fix" this we could qualify our RAM CS′ enable by insisting that all those bits also be zero; but it would serve no purpose. It may seem sloppy to have ghost images of memory all over address space, but it is harmless, and in fact saves gates. The same thing is happening with ROM (and with I/O). Figure 11.11 shows the circuit's *memory map*, with the repeating memory shown explicitly.

You do have to look at additional address lines if you want additional memory in the system, of course. The circuit shows the natural way to do that – simply add a 1-of-4 decoder ('139) on the next two address lines, enabled by our existing RAM enable, and, presto, you can add 3 more RAMs. The extension to even larger memory should be evident.

EXERCISE 11.5

Use a 1-of-8 decoder ('138) to expand our original circuit to accommodate eight 8K×8 RAMs.

EXERCISE 11.6

Modify the original circuit to accommodate four 32K×8 RAMs.

EXERCISE 11.7

Now change the circuit to accept a pair of 64K×8 ROMs (27512).

EXERCISE 11.8

Redraw the memory map for each of the previous exercises.

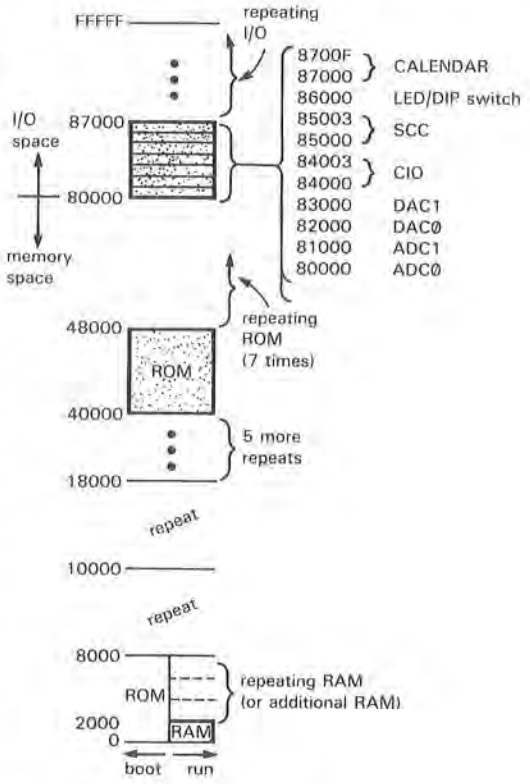


Figure 11.11. Memory map.

Memory timing. Before going on to I/O, it's worth looking at the memory timing situation. We said earlier that our DTACK circuit generated no wait states for memory accesses. That's OK as long as the memory is in fact fast enough to satisfy the timing constraints of the read and write cycles in Figure 11.4. But is it? The way to find out is to start with the 68008 no-wait-state timing diagram, then subtract worst-case delays of the supporting "glue" circuitry to see how much time is left for the memory to respond. Let's try it.

Figure 11.12 shows the situation for a read cycle, which usually presents the more difficult timing. We began with the CPU

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

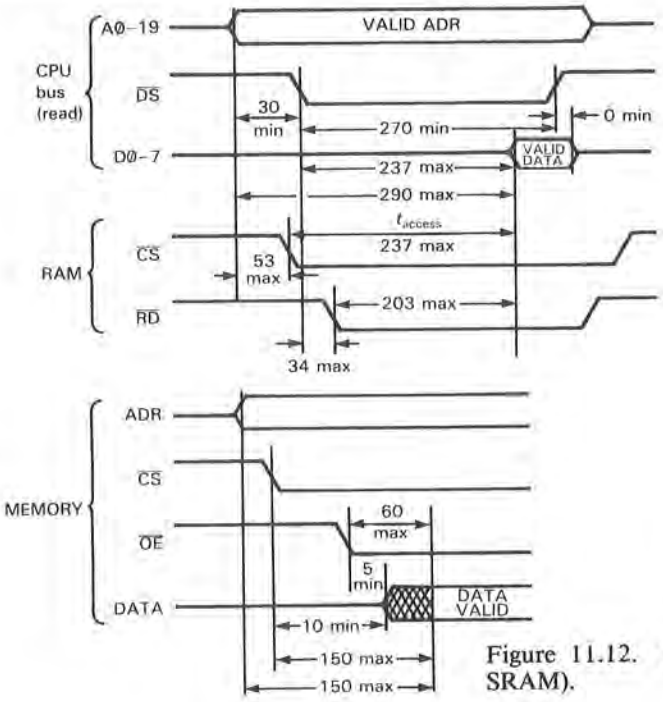


Figure 11.12. Memory READ-cycle timing (150ns SRAM).

timing specifications for the 8MHz chip, since we chose that clock frequency for our circuit. The most important timing is from CPU address valid to memory data valid, because that determines the maximum "address access time" of the memory that you can tolerate. In this case the CPU asserts a valid address at least 290ns before data must be valid; the corresponding figure for DS′ is 237ns. Our RAM CS′ circuit has two cascaded gates, which we'll assume are 74HCT02 and 74HCT00; the worst-case delays are 28ns and 25ns, for a total of 53ns maximum. That leaves 237ns (290−53) for the memory's access time from CS′. By a similar calculation (assuming a single 74HCT32 to generate RD′) the memory has to provide data no later than 203ns after OE′ is asserted. The figure also shows the worst-case timings for the slowest grade (150ns) of 8K×8 static RAM: 150ns access time from address, 150ns access time from CS′, and 60ns access time from OE′. Since our circuit allows 290ns, 237ns, and 203ns, respectively, we have nearly 100ns of margin in the closest case (access from CS′).

We won't go through the analogous calculation for a write cycle, which turns out to be even more relaxed. Evidently there is no RAM timing problem, even with the slowest grade of memory and no CPU wait states.

Unfortunately the same cannot be said of ROM, which tends to be slower than RAM. For example, 32K×8 EPROM comes in standard speeds (address-to-data or CS′-to-data) of 150ns, 200ns, and 250ns. The calculation goes as before, but with an additional 6ns of CS′ delay because of different gating to CS′. Thus, only the two fastest grades of EPROM meet the 231ns CS′-to-data requirement and thus are guaranteed to work without wait states. Instead of using the fast ROMs, we could use faster glue logic, either 74ACT or 74F, which would let us use the 250ns grade of

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

ROM. In reality, they would probably work fine in our circuit anyway, since worst-case calculations tend to be very conservative. They take into account the worst-case combinations of power-supply voltage, temperature, capacitive loading, and batch variation; for example, our worst-case conditions assumed a V_{CC} of 4.5 volts, a temperature range of −40°C to +85°C, and an unrealistically high load capacitance of 50pF. However, if you want to be *sure* your circuits work reliably, particularly if you are shipping production quantities of instruments, you should adhere to worst-case design.

Peripheral circuits

We have nine peripherals in this circuit, so we used a 1-of-8 decoder ('138) as an address decoding switchyard (with the LED indicator and DIP switch sharing one read/write port). The decoder is enabled on A19 asserted, which puts us in the I/O portion (the top half) of address space; it is also disabled during interrupt acknowledge, as we explained earlier. We used address lines A12–A14 into the decoder, which puts the successive peripherals at addresses $80000, $81000, $82000, etc.; we ignored the remaining high-order address lines, as we did with memory, thus producing a "lazy" address decoding, with each peripheral making multiple appearances in address space. In fact, *every* address above $80000, right up to the top of memory at $FFFFF (that's a half million addresses), has some peripheral living in it!

EXERCISE 11.9

Elaborate on this last statement by calculating exactly how many times one of these peripherals appears in memory. Then write down the general form for the LED indicator's address, using x's for "don't cares."

EXERCISE 11.10

The only real disadvantage of our incomplete address decoding is that it wastes a half megabyte of address space (most of which could hold memory) on a few paltry peripherals. Show how a designer who wants to use most of the 1Mbyte address space for memory might decode I/O addresses so that the eight ports are memory-mapped at $FF000, $FF100, . . . , $FF700 and do not respond to lower addresses. You could now install a megabyte of RAM, except for the fact that both memory and I/O would respond to those high I/O port addresses. Figure out a way to solve this problem.

Note that the address decoding assigns a whole set of contiguous addresses to each peripheral, since the low-order address lines are also ignored by the '138. Some peripherals have multiple internal registers, and we'll use a few low-order address lines to address them. Think of the decoder as responding to the peripheral's *base address*. Let's now look at the particular I/O devices in the circuit.

LED and DIP switch. These are the simplest of ports. For output, we drive a set of LEDs with an octal D register, clocked by qualifying the address-decoded LEDSW′ with WR′. Note the use of trailing-edge triggering for relaxed setup timing constraints. We used a '273 octal register with two-state outputs, rather than the usual three-state '574, because it has a RESET′ input, which we assert during boot or processor reset; that's useful here so that the LED register starts up with no lights lit. HCT logic has rail-to-rail saturation and good output sourcing capability (8mA at +4.5V out), so we can drive an LED array that returns to ground (with LS logic you have to return the LEDs to +5V); this is nice because the LEDs then indicate 1's, not 0's. The LED array shown has built-in resistors, limiting the current to 6mA. Note the use of one LED port bit to drive a solid-state ac relay.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

These relays are easily driven by logic levels (3V guaranteed turn-on, 1.5k load resistance), and they use "zero-voltage" switching (Sections 9.08 and 9.10). Note also that the LED port's WRITE strobe does double duty, by clearing the BOOT flip-flop the first time it is asserted; once you've cleared the flip-flop, you can use the LED port as you wish.

The DIP-switch input port is also easy, with a '240 octal inverting three-state buffer driven by the pulled-up switch levels, and enabled by the same LEDSW′ port decoding, this time qualified by RD′. In other words, if you write to $86000 it shows on the LEDs; if you read, you get a byte indicating the DIP-switch settings. We used an inverting buffer so that a switch that is closed reads as a 1, not a 0.

ADCs and DACs. These are easy ports, also. Both converters are advertised as "complete" converters, with internal references and clocks. The AD670 ADC matches the (R/W′, DS′) strobing convention, since it has a direction input and a chip enable input. A write (chip enabled with R/W′ LOW) begins conversion, while a read gets the resulting byte. During the write, the ADC latches the two data bits: BPO/UPO′ controls the input range (HIGH = bipolar, LOW = unipolar), and FMT controls the digital output format (H = 2's complement, L = unsigned binary). The DONE output tells you the ADC has finished a conversion; we ignored it in our design because we think it's easier to execute a few no-ops during the 10μs (max) conversion time than it is to hang on a flag. The AD670, like many peripheral chips, has a sluggish processor interface. This one, in particular, requires a 300ns minimum CE′ strobe during write and has a 250ns access time from assertion of CE′ during read. If you look back at Figure 11.4, you'll see that this violates the fast timing requirements of the 68008 during a normal (0 wait state) bus cycle; but with two wait states (which our circuit generates for all addresses $80000 and up), all is well: the DS′ for a write becomes 390ns wide, and the DS′ setup on read becomes 487ns.

The AD558 DAC is also a complete converter, with convenient single +5 volt supply and *voltage* output. It is write-only, so we used the WR′ strobe for chip enable, with the decoded address used for chip select. Once again the timing is violated without wait states: The AD558 needs 200ns of data setup to the trailing edge of CE′, and 150ns minimum CE′ pulse width. With no wait states you have only 180ns and 140ns, respectively; two wait states increases that to a comfortable 430ns and 390ns.

Serial and parallel ports. The Zilog 8530 SCC (serial port) and 8536 CIO (parallel port and timer) typify large-scale-integration (LSI) peripheral support chips. These tend to be designed with lots of flexibility and a dizzying choice of operating modes, programmed by sending particular bytes to one or more internal registers. Many of these chips are as complicated as microprocessors (see Fig. 11.13), and you should figure on spending up to a half day learning how to program their operation.

Although LSI peripheral chips are usually designed for specific microprocessors, their generality allows you to use one manufacturer's support chips with another's CPU. The Zilog 85*xx* chips actually claim to be "universal" bus-independent peripherals, although in the case of the 68008 there is a minor bus incompatibility involving the RD′ strobe that we will fix by generating a delayed RD′ strobe.

Look first at the 8536 parallel port/timer. It uses the (RD′,WR′) strobing pair, along with a chip enable input, CE′ (which we assert, as usual, from the I/O address decoder). In addition, it requires a clocking input, both to clock its timers and

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

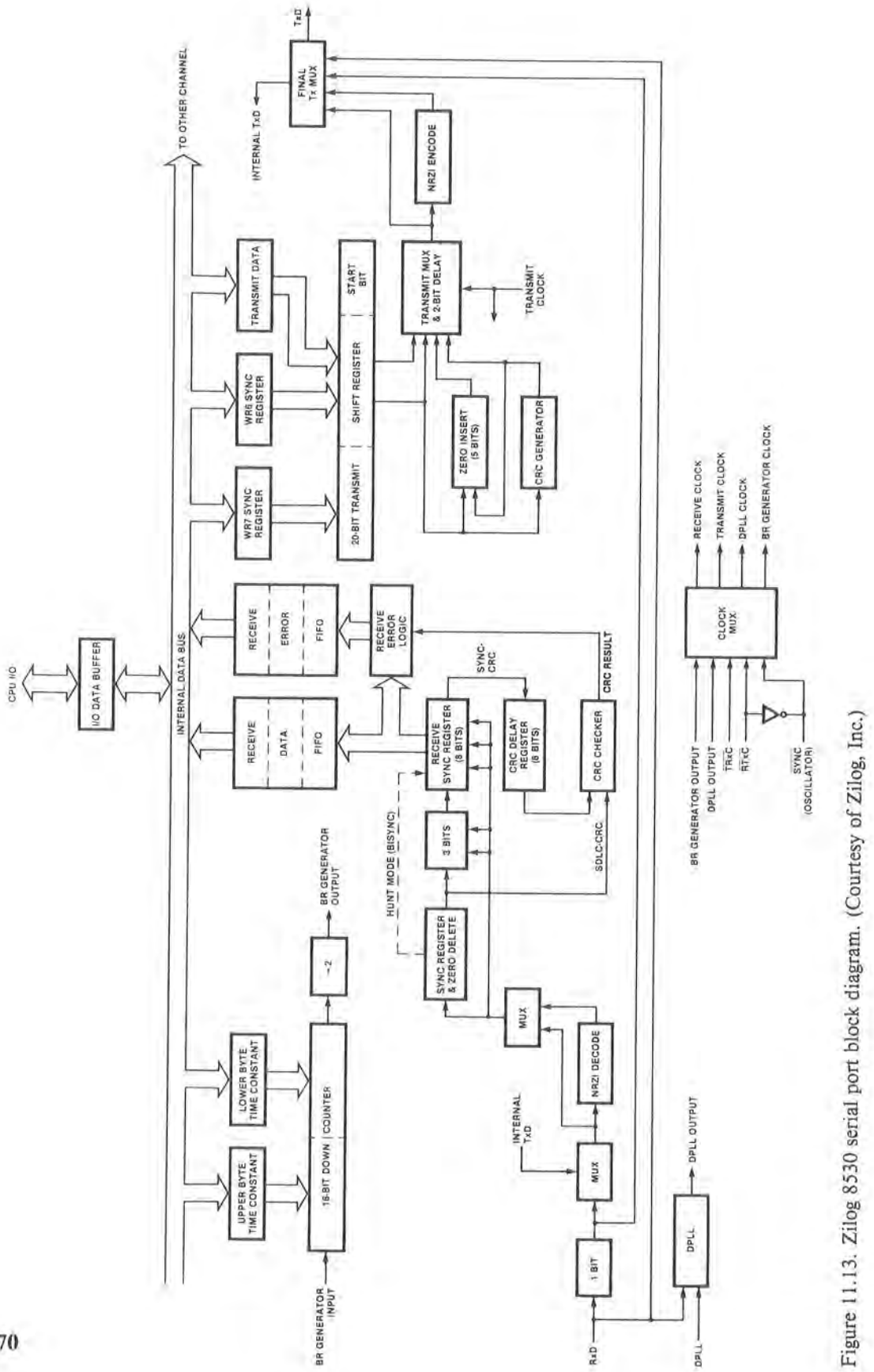


Figure 11.13. Zilog 8530 serial port block diagram. (Courtesy of Zilog, Inc.)

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

to control internal logic. The 8536 has full interrupt acknowledge circuitry, and it can assert a vector onto the data lines during an interrupt acknowledge cycle. We won't be using most of that capability, which includes a daisy-chained priority line (IEI input, IEO output) and the INTACK′ input that signals it to assert its (programmed) vector; we will use the INT′ output, however, which is used to request an interrupt. The only other bus interface lines, besides the data lines D0–D7, are the two inputs (A0, A1) used to address internal registers; you always connect them to the low-order bus address lines, which puts the internal registers into the CPU's address space beginning at the base address. In this case, for example, the internal registers are located in memory space at $84000–$84003. The addressing pins suggest that there are only 4 internal registers, which is quite incorrect: There are actually 41 writable registers and 48 readable registers! (We warned you these chips are hell to program!) You get to them in a two-step process by first writing to the "control" register at BASE+3 ($85003) a byte containing the target register address, then reading or writing that register. The parallel port data registers are a special case, with the addressing streamlined so that you can read/write them directly (at addresses BASE, BASE+1, and BASE+2).

Figure 11.14 shows the RD′ strobe timing problem: The 8536 specifies an 80ns minimum setup time of the A0–A1 address lines before the RD′ strobe leading edge; it also specifies the usual sluggish response – in this case it takes the poor 8536 255ns to produce the data, and it insists on a 390ns (min) width for the RD′ pulse. The slow *response* is by now a familiar problem, solved by wait states. But the address *setup* to RD′ isn't helped by wait states (see Fig. 11.4, which shows that DS′ can come as soon as 30ns after valid address). To solve this problem we have to delay RD′ by a CPU clock cycle; we can do this conveniently with the same shift register circuit we built to generate DTACK′. In this case we simply AND the "prompt" RD′ strobe with the (inverted) Q_0 shift register output, which isn't asserted until the S3–S4 CPU clock edge. This produces a delayed RD′ strobe, which we've called DELRD′, which begins one clock late (at the same time as a normal *write*-cycle DS′). The result is to give the 8536 an extra 125ns of address setup time (i.e., 155ns total). The wait-state generator still introduces 2 wait states, which makes the overall cycle long enough for the slow peripheral.

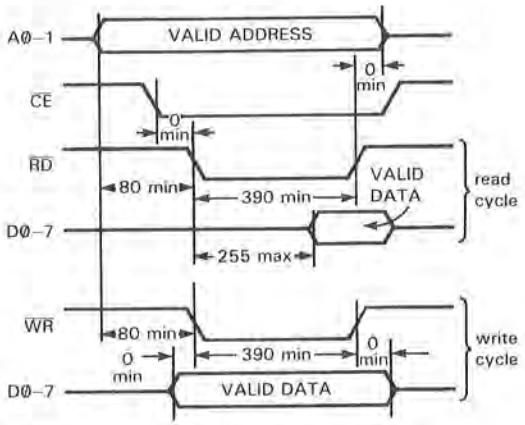


Figure 11.14. Zilog 8536 parallel port timing.

As luck would have it, a similar circuit isn't needed for WR′, because the 68008 considerately provides an extra clock cycle of setup during write cycles (note the delayed DS′ in Fig. 11.4 during write), while the 8536 still requires the same 80ns of setup (Fig. 11.14).

The 8530 serial port interfaces almost identically. The only difference is that the address lines to the internal registers are named differently, with A0 driving A/B′ (which selects channel *A* or *B* of the dual port), and A1 driving D/C′ (which selects data or control registers). This chip is also well endowed with registers: There are 16

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

writable registers and 9 readable registers for each of the two channels, with the same two-step access as the 8536.

The 8530 accepts clock inputs to 6MHz; we chose a 4MHz clock because that permits asynchronous baud rates up to 9600 baud. The asynchronous data lines TxD and RxD are TTL logic levels, not bipolarity RS-232 (see Sections 9.14 and 10.19). Most RS-232 driver chips (typified by the classic 1488) require dual supplies, which would be a real show-stopper in an otherwise single +5 volt system like this. Luckily you can now get clever chips with onboard flying-capacitor voltage converters, originated by Maxim with their MAX232 series of chips (or the LT1080 from LTC). Note that the open-drain interrupt request line from the 8530 is wired-ORed with the corresponding calendar clock line, so either makes an autovectored interrupt at level IPL1. The interrupt handler for this level has to find out which interrupted, by polling each device's status register via programmed read, as we will illustrate.

Calendar clock. The last peripheral is the calendar clock, another LSI chip with some tricky features. Its bus interface is very similar to the Zilog parts, with the Intel-style (RD′,WR′) strobe pair, and five bits of internal register addressing. It may even have the same RD′ timing problem. We say "may" because the data sheet is ambiguous: It specifies a "typical" setup time from address to RD′ (100ns), but doesn't give a *minimum*. We don't know what it is they're trying to say, but why take chances. We've already got the DELRD′ strobe, so let's use it. The ICM7170 is a modern calendar clock chip with internal battery switchover circuitry; you just hang a 3 volt (lithium) cell on, as shown. Earlier calendar chips forced you to worry about disasserting control lines in an orderly sequence at shut-down, but the 7170 also takes care of all that. Of course, you can't talk to the chip when the +5 volts is gone; the battery just keeps the clock ticking during hibernation, so it wakes up with a clear head and a keen sense of time.

Circuit power

Having finished the circuit design, it's tempting to ignore minor details such as power and ground. You shouldn't. Our circuit uses "5 volt logic," which often means 5 volts ±5% (in this example, the CPU and several peripherals require V_{CC} between +4.75V and +5.25V). Furthermore, the supply voltage should be free of large spikes, which is best managed by liberal use of 0.1μF ceramic capacitors, with a sprinkling of larger tantalum electrolytics. The extreme case of a "large spike" is rampant overvoltage caused by outright failure of the +5 volt series regulator in the power supply. For these occasions you should provide an overvoltage crowbar (Section 6.06), either on the motherboard or at the power supply. When thinking about power supplies, keep in mind that you may easily have a dc current drain of an amp or more in a microprocessor card, and perhaps many amps on the motherboard traces that power plug-in cards. Plan ahead, with hefty printed circuit traces and ample connector current capability.

For reasons explained in Section 9.11, ground wiring on and between PC boards is especially important and should be made low-inductance. The best way is to devote a layer of a multilayer PC board, though a "gridded" ground arrangement on a two-layer board often works well enough (more on this in the next chapter). Finally, a word of advice about power-on reset circuits: It's tempting to use a simple *RC* (+ diode) circuit of the kind we showed in Figure 11.10. The problem is that such a circuit may not respond to a short power dip that is sufficient to disrupt the running program. If the microprocessor circuit is part of an instrument, the effect is that the

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

instrument starts malfunctioning, and the only way you may be able to make it regain sanity is by switching the ac power off and back on again! We've had this happen to commercial instruments, as well as to our own designs. The safest solution is to use a good supervisory circuit, for example the MAX690 series from Maxim.

Having disposed of the circuit design, which fell together pretty easily, we'll next proceed to the really hard part, programming.

Warm-up: boiled egg

Our signal-averager example will turn out to be a complex programming task, as in fact most real-world problems are. Close reading of the following sections will reward those readers intending to design their own microprocessor systems, since it will illustrate in detail most of the techniques needed to build serious microprocessor-based instruments.

Instead of jumping immediately into the shark-infested waters, however, let's dip a toe in gently with a simple (and frivolous) example of software for our general-purpose microprocessor circuit. Computers are supposed to spare us unnecessary drudgery in our lives. Let's make this one cook our 5 minute boiled egg each morning at 8AM!

Imagine that the solid-state relay in Figure 11.10 is hooked to our coffee-cup immersion heater, sitting in a cupful of water in which a raw egg is waiting. Look at the code in Program 11.2.

To keep this program simple, we assume the computer is running, and the calendar clock is set. (In the signal-averager example we'll see how to handle those uninteresting, but important, details!) The code begins by sending a zero byte to the LED port, to turn off the heater; it also sets the calendar clock to 24 hour mode and turns off interrupts. It then goes into a loop ("WAKE") that continually tests the hours digit from the calendar clock, until it reads an "8," whereupon it sends a byte of all 1's to the LED port; that turns on the heater and lights all the LEDs.

Now the program falls through to a second loop ("COOK"), which continually tests the *minutes* digit of the calendar clock, until it reads a "5," whereupon it sends a zero byte to shut off the heat (and LEDs). Finally, the program falls

Program 11.2

```
;cook egg 5 minutes at 8AM every day
;assume calendar clock set, computer running
        CLR.B   $86000          ;clear LED and ac relay
        MOVE.B  #$0C, $87011    ;24-hour calendar mode, no int

wake:   MOVE.B  $87001, D0      ;get time -- hours digit
        CMP.B   D0, #8          ;is it 8AM?
        BNE     wake
        MOVE.B  #$FF, $86000    ;yes. start egg, light all LEDs

cook:   MOVE.B  $87002, D0      ;get minutes digit
        CMP.B   D0, #5          ;cooked 5 minutes?
        BNE     cook
        CLR.B   $86000          ;yes. stop egg, turn off LEDs

wait:   MOVE.B  $87001, D0      ;get hours digit
        CMP.B   D0, #8          ;9AM yet?
        BEQ     wait
        BRA     wake            ;yes. look forward to tomorrow's egg
```

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

through to a third loop ("WAIT"), which continually tests the hours digit again, until it *stops* being "8." At that point it branches unconditionally back to the first loop, waiting for 8AM, tomorrow.

We wrote this rough-and-ready program to show how simple it can be. Don't copy our style, though – we did many things crudely, to save space and explanation. If we were willing to expand the code, we could make the timer "smart," for example by using one of the ADC ports to sense when the water reaches boiling; that could initiate the actual egg timing, and even thermostat the heater to save energy! We could allow you to set the wake-up time, the cooking interval, etc., via the NMI button. A DAC port could display the time, perhaps as "walking digits," while the other DAC port (attached to a pillow speaker) could speak to you in a quiet voice, gently waking you with soothing thoughts and memorized music . . . but we digress!

OK, it's time to dive in.

11.06 Programming: defining the task

The surest way to waste a lot of time, and get yourself totally confused in the process, is to begin programming before you've figured out what you want to do. This is especially true when you are programming a dedicated controller in assembly language, because assembly code itself does not have the clean modularity and control flow of a structured high-level language; furthermore, to optimize real-time performance you often resort to tricks and less-than-transparent ways of doing things. The resultant code – full of branches, tasks divided between interrupt handlers and main code, software flags that get modified in unpredictable places, command bytes going to peripherals, etc. – can quickly become a patchwork horror if halfway through you are still discovering how you should have viewed the job at the outset. Just as in painting a house, the preparation may take longer than the actual work, but it's well worth the effort.

The signal averager we're designing is a good example. It is not a very sophisticated instrument, yet a glance ahead at the interrupt handler's flow chart (Fig. 11.21) should convince you that there is considerable complexity in the setting and reading of flags and signals, the real-time alteration of interrupt vectors, and the overall control flow. So it's worth spending the time now to understand how the instrument is intended to work.

What is a signal averager?

A signal averager, sometimes called a multichannel scaler, is an instrument used to improve the signal quality (signal/noise ratio) of a periodic analog input signal that is unavoidably combined with nonperiodic noise (or interfering signals). It does this by measuring the signal amplitude many times during each period, storing these amplitude samples into a set of sequential "bins," then adding to each bin the corresponding samples from many additional periods of the input waveform. In other words, the signal waveform is co-added *modulo its period.* As we explain in detail in Section 15.13, this improves the signal/noise ratio of the signal accumulating in the bins because the (periodic) signal grows linearly with time, while the (random) noise fluctuations grow only as the square root of time. We call each successive periodic accumulation through the bins a "sweep"; a typical data run might consist of a few thousand sweeps.

A good signal averager continually shows you the accumulated waveform (stored in its 1000 or so bins) on a CRT display and gives you a wide choice of dwell time per bin, trigger modes, display scales, etc. We'll put many of these features in ours, but we'll stop short of implementing everything that is possible,

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

in order to fit the example into this chapter. In deciding what to include, we've picked a nice set of functions that lets us illustrate the full range of programming tricks and trade-offs, without getting carried away with gold-plated bells and whistles.

Features

For reasons we'll explain shortly, we chose to use a conventional control panel with labeled switches, rather than the contemporary fashionable keyboard with CRT-screen menus. Our signal averager therefore looks just like a conventional instrument, and the controls have to have predefined functions and ranges. When we planned the chapter, we actually began, as we're beginning here, by deciding what functions to implement, and over what ranges.

We decided to use a fixed number of bins (256), with a large set of selectable dwell times per bin. Since signal averagers get used for two kinds of periodic phenomena – those that have an intrinsic period of their own (e.g., ocean tides) and those that you can trigger or drive periodically (e.g., nerve impulses or resonance scans), we provided two sweeping modes: "triggered," in which the instrument waits for an external trigger before beginning each sweep, and "autolooping," in which it just cycles around periodically. We provided two ways to terminate the signal average: a set of preprogrammed total sweeps, and a "stop" switch that terminates with the next complete sweep. Our design includes analog X and Y outputs (with Z-unblanking pulses) for a continuously refreshed display of the averaged signal on an XY CRT: We provided a set of display scales (progressing by factors of 2) and a smart "autoscale" mode, in which the data are continually rescaled (normalized) according to the number of sweeps completed. Finally, there are LED status indicators (standby, sweeping) and logic-level outputs indicating sweep-in-progress and end-of-sweep. Here are the specifications of our signal averager:

Analog input range: ±5 volts

Number of bins: 256

Internal representation: 32-bit signed integer

Dwell time per bin: 100μs to 1 second, in a 1-2-5 sequence

Sampling: Finite integration (sum of 100μs samples)

Preprogrammed number of sweeps: 1 to 20,000, in a 1-2-5 sequence

Sweep modes: Wait-for-trigger; or autoloop

Display modes: Selectable scale, with wraparound; or autoscale

Display scales: 1 to 16K input range, by factors of 2

Inputs: Analog signal, ext trig

Outputs: X, Y, Z (to CRT), SWEEP, END

Additional controls: START, STOP, RESET (reboot)

Figure 11.15 shows the signals coming in and going out of the microprocessor board. We've used the 8536 parallel port for all the digital signals, with direction and polarity programmed appropriately; all digital inputs that come from switches should have resistor pullups, with switch closure to ground. No debouncing circuitry is needed, because we will do software debouncing.

The analog filter is important and deserves some explanation. If you make periodic brief samples of the amplitude of a continuous analog waveform of finite bandwidth (where the maximum frequency present is f_{max}), you preserve the input information as long as you sample at $2f_{max}$ or faster. If you don't meet this *Nyquist* criterion, funny things can happen; in particular, Figure 11.16 shows the phenomenon of *aliasing* due to undersampling, in this

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

case with frequencies close to the sampling frequency appearing folded back down near zero frequency. In order to prevent aliasing, you must low-pass-filter the incoming waveform at $f_{sample}/2$ or less.

This is simple enough, but how do we handle the fact that the signal averager has adjustable dwell time per bin, i.e., adjustable sampling time? One possibility is to build an adjustable low-pass filter (perhaps a switched-capacitor filter with programmed clock rate) at the input to match the dwell time; this makes sense, because if you choose a long dwell time, you're not interested in high frequencies anyway. But a simpler solution is to notice that if you *integrate* (or average) the signal during the dwell time, you've got an automatically tracking low-pass filter. For this reason, signal averagers sometimes use V/F converters (which are intrinsically integrating) at the input. Our solution amounts to the same thing: We always sample the analog input at 10kHz (which is filtered with a matching 5kHz anti-aliasing filter); for longer dwell times we effectively integrate by adding the appropriate number of successive samples. Figure 11.17 shows what the front panel might look like.

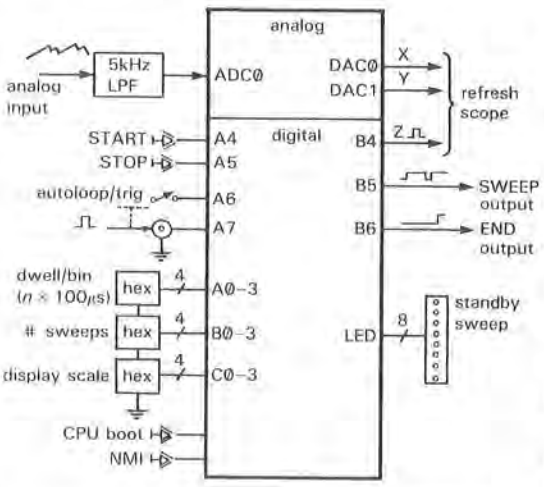


Figure 11.15. Signal-averager controls and inputs/outputs. "LED" is the indicator port in Fig. 11.10; "A," "B," and "C" refer to parallel port bits of the 8536.



Figure 11.16. Aliasing due to undersampling.

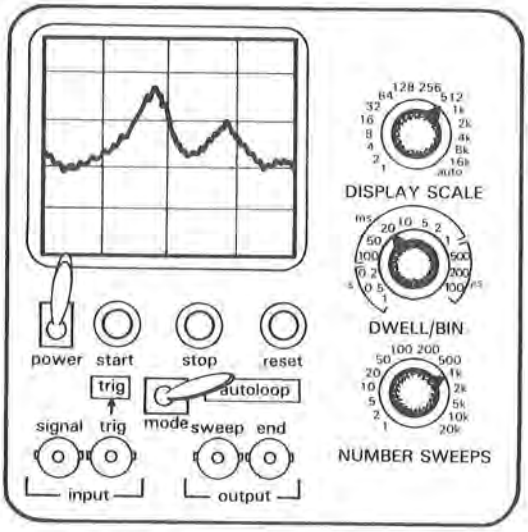


Figure 11.17. Signal-averager panel.

Non-features

It's worth explaining what we left out, and why. We decided not to succumb to the keypad/menu mania, for a few reasons. First, we often find keypad instruments annoying, because each one works differently, and you have to keep relearning how to use them. Second, they're often slower to use; you have to do some *typing* just to change scales! Last, a keypad/menu scheme requires tedious parsing and screen-driving software routines, with no redeeming pedagogical value. So we decided to use the parallel port to read some panel controls, which is quick and simple, even though it sacrifices flexibility by limiting your choices.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

There are quite a few other features that would make a better instrument, but we left them out to keep the program from getting too large for this chapter. We could have had a second analog input channel (there are two ADCs), selectable numbers of bins per sweep, multiple "banks" of data memory, digital outputs telling current bin number, and analog outputs proportional to bin number (for controlling analog quantities). These are all in the nature of desirable enhancements; an *essential* feature that we omitted is the capability to send the final data to a microcomputer, best done through one of the serial ports.

Even the features we included in our design could have additional modes. In particular, microprocessor systems with both A/D and D/A converters can benefit from built-in analog multiplexers that let you loop the DAC outputs back into the ADCs. That way, you can test all the converters with software, as part of a power-on test sequence (that also includes memory and port tests, etc.). You can even check the power-supply voltage(s), using the ADCs.

EXERCISE 11.11

Show how to do this, by connecting analog multiplexers at the input of the ADCs. You'll need to arrange a port address to which the CPU can ship its MUX-select commands.

11.07 Programming: details

Overview

Programs are usually complicated. Time-critical assembly-language programs always are. Rather than show you some fake code that wouldn't do the job, we are going to show you finished code that would work in our hardware microprocessor circuit, in all its beauty (and ugliness). You'll never understand it if we don't ease you in, layer by layer. The best way to get started is in a "top-down" fashion, which means that we begin by identifying the major tasks of the job, the order in which they will be handled, and the protocol for interchange of information between major modules of the program. Having done this, we can proceed with flow charts of the individual program modules. Finally, we write the actual assembly-language code.

Note: The material that follows is necessarily detailed. Readers who simply want to get the "general idea" may wish to skim the discussion of programming, rejoining us in Section 11.08 or 11.09.

Figure 11.18 shows the big picture, considerably simplified. We have assigned three arrays in RAM: a block of 256 32-bit ("long") integers, to hold the current DATA for each bin; a block of 256 16-bit integers ("words"), to hold the number of sweeps completed for each bin, used to NORMalize the data in "autoscale" mode; and an array of 256 bytes, to hold the data that are continually sent to the DISPLAY. Our jobs are to add fresh data from the ADC into the DATA array, while updating the NORM array accordingly; scale those long integers to bytes in the DISPLAY array (using the NORM array, if autoscale mode; otherwise just by shifting); and display those bytes continually.

Here's how the program works, in broad brush strokes: At power-on, the CPU does a complete boot, and begins executing from ROM (whose ghost image at the bottom of memory cleverly provides start-up vectors to its real self, as we described in Section 11.05). The various peripheral chips must be initialized (by sending the right bytes to their command registers), and the program has to initialize arrays, pointers, starting values, etc. An instrument like this has several possible states (waiting for START, taking data, etc.), and we begin in a standby state, waiting for someone to push the START button. The machine isn't totally dead, though – it should also be displaying its data, which we have initialized to all zeros.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

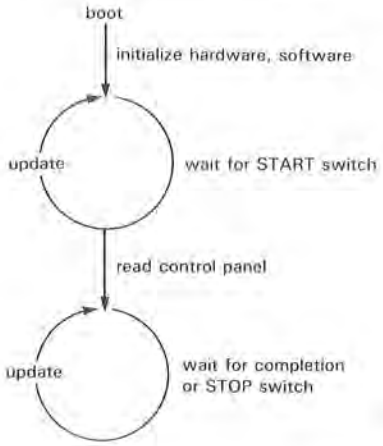
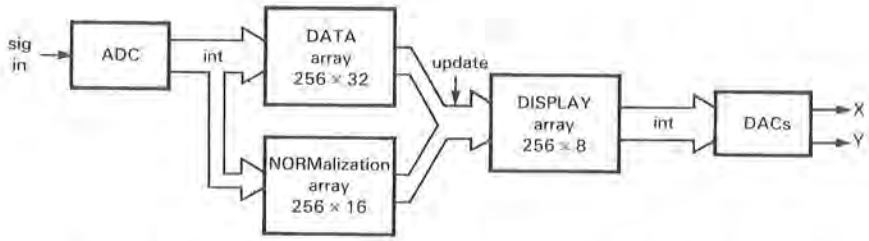


Figure 11.18. Programming: the big picture.



When the START button is pressed, the program proceeds to read the rest of the control panel, to get operating parameters. Then it goes into a data-taking mode (with parameters as specified by the control panel), while displaying the ongoing signal averaging. It jumps back into standby mode when it has completed the specified number of sweeps, or sooner if it senses the STOP button.

Timing; interrupt-driven versus programmed tasks

The most important task is to ensure that the ADC makes conversions every 100μs and that these data get integrated and added into the DATA array. The next most important task is to keep the display refreshed, at a rate of at least 40Hz or so to prevent flicker. Of lesser importance is the job of keeping the DISPLAY memory up to date with what is happening in the DATA array.

You might at first think that the most important task should be tightly coded in the main program loop, with nuisance interrupts occasionally stealing time for minor tasks. You would be wrong. Interrupts take priority over running code, so the time-critical tasks should be in the interrupt handler, and the less important tasks should be in the "main" code, which will get executed with whatever time is left over. (This assumes, of course, that the processor handles interrupts quickly. As we will see, the extravagant number of registers in the 68008 makes it an extremely efficient interrupt handler; the 68008 is well suited to this job.)

As we've indicated in Figure 11.18, ADC data go into the DATA array via interrupts, created at 100μs intervals by a programmable timer in the 8536. The main program keeps busy just continually updating the DISPLAY array, using a routine called "***update***" (since it is oblivious to all the interesting data collection going

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

on under its nose, it is obliged to check a flag that the interrupt routine sets to know when the run is complete). We had originally planned to do the display refreshing in the main loop also. But there was an interesting snag: With each X, Y pair you send to the DACs for display, you have to send a "Z-axis" (trace intensifying) pulse. We showed in Section 10.20 how to make a "software pulse" by sending a 1 followed by a 0 to some port bit. The Z-axis pulses must all have the same duration; otherwise the display will have some points brighter than others. But you can't safely make constant-duration software pulses if interrupts are going on underneath your code.

EXERCISE 11.12
Why not?

One solution is to turn off interrupts, make the pulse, then turn interrupts back on. This is ugly, because it creates undesirable latency for our most important task, the regular collection of ADC samples. Then we hit on the better way: Display one point as an auxiliary task in the interrupt handler. It executes every 100μs, so the complete 256-point graph will be displayed 40 times per second. What's even better, the interrupts go on even when the "main" program is in its other state ("standby" – waiting for START), so the screen display doesn't go away. Finally, there's a wonderful serendipity here, since you have to wait 10μs after starting the ADC before you can read its conversion; that's just enough time to send an X, Y pair to the DACs. In other words, putting the display refresh in the interrupt handler doesn't cost any execution time at all!

□ *Main program: setup*

We've beat around the bush long enough. Let's look at the detailed tasks done by the program. Look first at the main program, which is diagrammed in the unconventional flow chart of Figure 11.19. The chart corresponds very closely to the assembly code, which is shown in Program 11.3.

The program listing begins with RAM address definitions (including our interrupt vector, space for variables and arrays) and port address (and bit) definitions. These definitions will later be used as operands in memory- and port-addressing instructions, causing the assembler to substitute the actual addresses. Even though the result is equivalent, you should always use definitions (rather than the alternative of using the addresses directly in the code), because it not only makes the subsequent code more readable but also makes it easy to change port and bit assignments in later revisions. The port addresses correspond to our schematic diagram and include the peripherals' internal registers, addressable via the low-order address bits or by two-byte transfers.

The program listing also shows how we will use the 68008's registers. At each interrupt we'll be getting data from the ADC, adding it to the current bin count and checking to see if the bin or sweep is complete. We could store the pointers and counters in memory (as you would have to when using a less capable processor like the 8086), but by reserving enough registers for the interrupt handler's needs, we can make the interrupts very efficient. Therefore, we've assigned data registers for the current bin accumulation (D7), remaining dwell count within bin (D6) and bin count within sweep (D5), offset into the display array (D4), and a scratch register (D3). Likewise, we've reserved address registers for the three arrays (NORM, A6; DATA, A5; DISPLAY, A4), and for the most-used ports [ADC0, A3; CIO (parallel port), A2]. The main program promises not to use these whenever interrupts are enabled.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

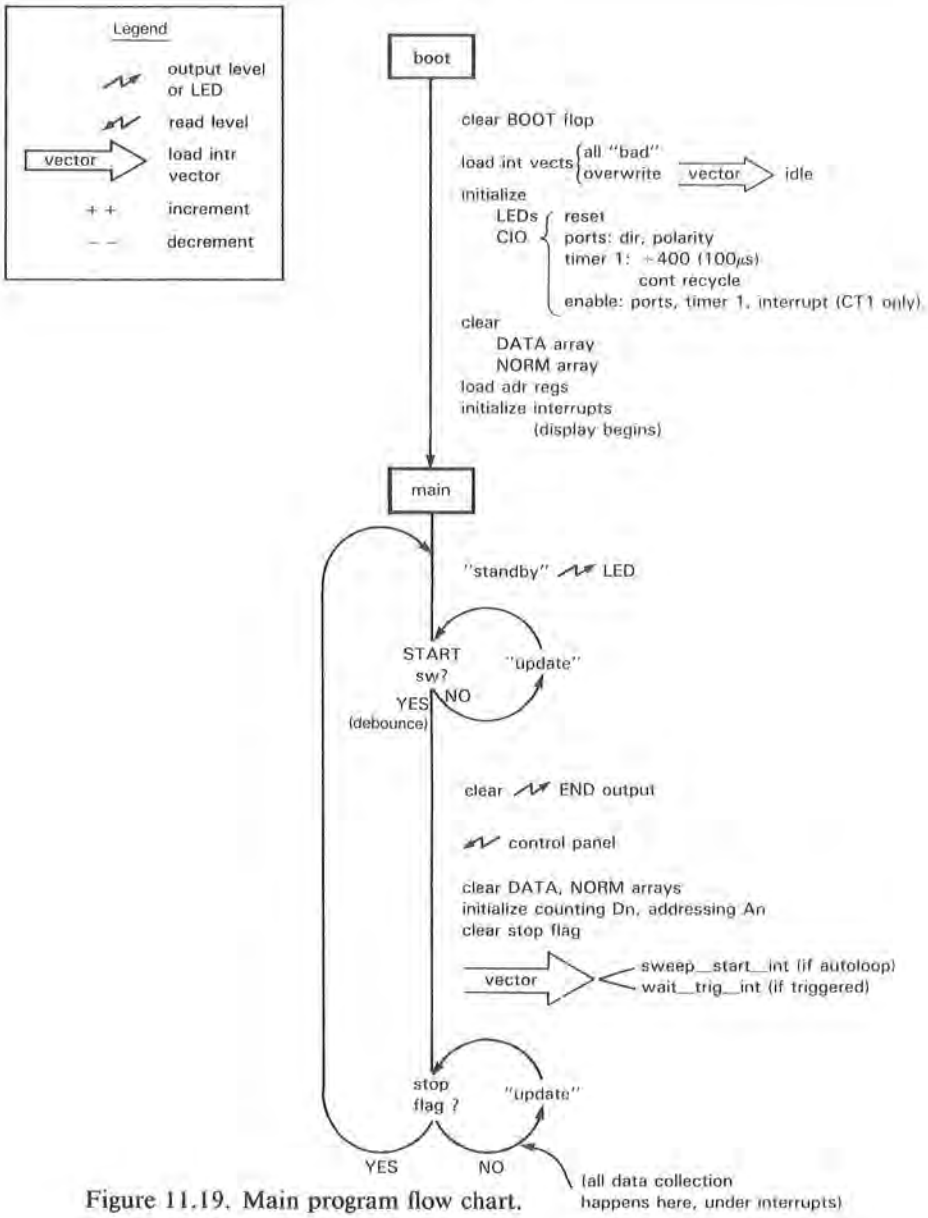


Figure 11.19. Main program flow chart.

You might wonder why we waste addressing registers (with all their fancy array-oriented auto-incrementing features) on fixed port addresses, when absolute addressing will do. The reason is speed. The absolute-addressing instruction

```
MOVE.B ADC0,D0
```

where ADC0 is a long absolute address ($80000 here), takes 28 clocks (3.5μs in our processor), whereas the instruction

```
MOVE.B (A3),D0
```

which uses indirect addressing through A3, takes only 12. The speed difference is caused entirely by the bus traffic, with the

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

Program 11.3

```
;RAM location definitions
   ;vectors
init_stack_top  EQU     $000    ;initial SSP
reset_vect      EQU     $004    ;startup vector
int5_vect       EQU     $074    ;level 5 int
   ;parameters
dwell_per_bin   EQU     $400    ;# ticks per bin
num_sweeps      EQU     $402    ;via decode_table
auto_loop_flag  EQU     $404    ;1=autoloop, 0=trigger
   ;internal flags
stop_flag       EQU     $405    ;set by handler when sweep done
   ;variables
led_store       EQU     $407    ;LED image in memory
update_offset   EQU     $408    ;indexes next point
   ;arrays
data_array      EQU     $1000   ;unscaled data (long)
norm_array      EQU     $1400   ;#sweeps/bin (word)
display_array   EQU     $1600   ;offset binary, scaled (byte)


;Port definitions
ADC0            EQU     $80000  ;kept in A3
DAC0_OFFSET     EQU     $2000   ;offset from ADC0 (index from A3)
DAC1_OFFSET     EQU     $3000
LED             EQU     $86000
   ;parallel port addresses
CIO_CNTRL       EQU     $84003  ;control reg
CIO_PA_DATA     EQU     $84002  ;port A data
CIO_PB_DATA     EQU     $84001  ;port B data -- kept in A2
CIO_PC_DATA     EQU     $84000  ;port C data
CIO_CNTRL_OFFSET EQU    2       ;index from A2
CIO_PA_OFFSET   EQU     1
CIO_PC_OFFSET   EQU     -1
   ;parallel port internal registers (access via CIO_CNTRL)
MAST_CNTRL      EQU     $00
MAST_CONFIG     EQU     $01
PA_CMDSTAT      EQU     $08
PA_MODE         EQU     $20
PA_POLARITY     EQU     $22
PA_DIRECTION    EQU     $23
PA_SPECIAL      EQU     $24
PB_CMDSTAT      EQU     $09
PB_MODE         EQU     $28
PB_POLARITY     EQU     $2A
PB_DIRECTION    EQU     $2B
PB_SPECIAL      EQU     $2C
PC_POLARITY     EQU     $05
PC_DIRECTION    EQU     $06
PC_SPECIAL      EQU     $07
   ;timer1 internal registers (access via CIO_CNTRL)
CT1_CMDSTAT     EQU     $0A
CT1_MODE        EQU     $1C
CT1_FROM_MSB    EQU     $26
CT1_FROM_LSB    EQU     $27
CT3_CMDSTAT     EQU     $0C
```

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

```
   ;parallel port bit assignments
   ;inputs -- port A
START_BIT       EQU     4       ;starts sweep
STOP_BIT        EQU     5       ;stops sweep at end
AUTO_LOOP       EQU     6       ;1=autoloop, 0=ext trig
EXT_TRIGGER     EQU     7       ;trig input if not autoloop
   ;outputs -- port B
Z_BLANK         EQU     4       ;display scope unblank
SWEEP_BIT       EQU     5       ;HIGH while sweeping
END             EQU     6       ;HIGH after last sweep
   ;LED bit assignments
LED_STAND_BY    EQU     7
LED_SWEEP       EQU     6
BOOT_BIT        EQU     0       ;rising edge kills ghost ROM
   ;cal clk and serial port
CAL_CNTRL       EQU     $87011  ;cal clk control reg
SCC             EQU     $85000  ;serial port base address

;Global register usage, for fast interrupt handling
   ;data registers
;D7 - currently accumulating value for one dwell time
;D6 - remaining dwell count (0 is terminal count)
;D5 - remaining bin count (0 is terminal count)
;D4 - index (offset) for X,Y display value each tick
;D3 - temp. reg. for integration; also START debounce
   ;address registers
;A6 - pointer into NORM array
;A5 - pointer into DATA array
;A4 - base pointer into DISPLAY array
;A3 - points at ADC; use offset for DACs
;A2 - points at CIO_PB_DATA port; use offset for other CIO ports

;ROM code begins here
        .ORG $40000             ;assembler directive, defines ROM location
        .long $2000             ;initial SSP -- top of RAM
        .long reset_entry       ;boot-up vector, should be $40008

          ;first executible statement next
reset_entry:
        MOVE.B  #0, LED         ;make sure LED reg is cleared, and turn off
                                ;BOOT flip-flop, since we're in real ROM
          ;now initialize vector table
        MOVE.W  #255, D0                ;table size minus 1
        MOVE.L  #bad_int, D1            ;the vector to load
        MOVE.L  #0, A0                  ;first vector location

vect_init_loop:
        MOVE.L  D1,(A0)+                ;load the vector
        DBF     D0, vect_init_loop      ;and loop

          ;now load int5 with the initial handler entry
        MOVE.L  #idle_int, int5_vect

          ;now initialize ports
          ;LED port first
        MOVE.B  #0, led_store           ;clear LED memory image
        BSET    #LED_STAND_BY, led_store  ;and set standby bit
        MOVE.B  led_store, LED          ;ship to LED port
```

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

```
   ;kill calendar clk and serial port
CLR.B   CAL_CNTRL               ;shut off cal & interrupts
MOVE.B  #09, SCC
MOVE.B  #$C0, SCC               ; shut off SCC & interrupts
   ;parallel port (CIO)
MOVE.B  CIO_CNTRL, D0           ;read to force state 0
MOVE.B  #MAST_CNTRL, CIO_CNTRL  ;prepare to reset
MOVE.B  #$01, CIO_CNTRL         ;in reset state
MOVE.B  #$00, CIO_CNTRL         ;out of reset state
MOVE.B  #MAST_CNTRL, CIO_CNTRL  ;master int cntrl
MOVE.B  #$00, CIO_CNTRL         ;dont enable yet
   ;port A
MOVE.B  #PA_DIRECTION, CIO_CNTRL ;port A direction
MOVE.B  #$FF, CIO_CNTRL         ;all inputs
MOVE.B  #PA_POLARITY, CIO_CNTRL ;port A polarity
MOVE.B  #$7F, CIO_CNTRL         ;invert all switch inputs
MOVE.B  #PA_SPECIAL, CIO_CNTRL  ;port A mode
MOVE.B  #$20, CIO_CNTRL         ;STOP switch is "1's catching"
MOVE.B  #PA_CMDSTAT, CIO_CNTRL
MOVE.B  #$E0, CIO_CNTRL         ;disable port A interrupts
   ;port B
MOVE.B  #PB_DIRECTION, CIO_CNTRL ;port B direction
MOVE.B  #$0F, CIO_CNTRL         ;top 4 bits are output
MOVE.B  #PB_POLARITY, CIO_CNTRL ;port B polarity
MOVE.B  #$0F, CIO_CNTRL         ;invert inputs
MOVE.B  #PB_SPECIAL, CIO_CNTRL  ;port B mode
MOVE.B  #$00, CIO_CNTRL         ;all unlatched
MOVE.B  #PB_CMDSTAT, CIO_CNTRL
MOVE.B  #$E0, CIO_CNTRL         ;disable port B interrupts
MOVE.B  #$00, CIO_PB_DATA       ;clear all outputs
   ;port C
MOVE.B  #PC_DIRECTION, CIO_CNTRL ;port C direction
MOVE.B  #$0F, CIO_CNTRL         ;4 input bits only
MOVE.B  #PC_POLARITY, CIO_CNTRL ;port C polarity
MOVE.B  #$0F, CIO_CNTRL         ;inverted
MOVE.B  #PC_SPECIAL, CIO_CNTRL  ;port C mode
MOVE.B  #$00, CIO_CNTRL         ;all unlatched
   ;timer
MOVE.B  #CT1_FROM_MSB, CIO_CNTRL ;MSByte of counting modulus
MOVE.B  #1, CIO_CNTRL           ;MSByte of 400
MOVE.B  #CT1_FROM_LSB, CIO_CNTRL ;LSByte of modulus
MOVE.B  #144, CIO_CNTRL         ;LSByte of 400
MOVE.B  #CT1_MODE, CIO_CNTRL    ;timer mode
MOVE.B  #$83, CIO_CNTRL         ;continuous, no in/outputs
MOVE.B  #CT1_CMDSTAT, CIO_CNTRL ;interrupts
MOVE.B  #$20, CIO_CNTRL         ;clear interrupts
   ;final doings
MOVE.B  #MAST_CONFIG, CIO_CNTRL
MOVE.B  #$D4, CIO_CNTRL         ;port A,B,C, & timer1 enabled
MOVE.B  #MAST_CNTRL, CIO_CNTRL
MOVE.B  #$80, CIO_CNTRL         ;enable chip's interrupts
MOVE.B  #CT1_CMDSTAT, CIO_CNTRL
MOVE.B  #$23, CIO_CNTRL         ;timer 1 start + interrrupts

   ;port initialization done
   ;set up arrays, registers, pointers, etc.
BSR     clear_arrays            ;zero DATA and NORM arrays
MOVE.L  #display_array, A4      ;initialize screen data pointer,
```

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

```
        MOVE.L  #ADC0, A3               ;ADC pointer,
        MOVE.L  #CIO_PB_DATA, A2        ;port B pointer,
        CLR.B   D5                      ;remaining scan count,
        CLR.L   D4                      ;display offset,
        CLR.W   update_offset           ;and update index to DISPLAY array

        AND.W   #$F8FF, SR              ;enable interrupts
main_loop:                              ;and PLAY BALL!
        BCLR    #LED_SWEEP, led_store
        BSET    #LED_STAND_BY, led_store
        MOVE.B  led_store, LED          ;do good things to LEDs
           ;now wait for START switch
           ;must stay open a while, then close
wait_for_zero:
        BSR     update                  ;update screen, waste 40us
        BTST    #START_BIT, CIO_PA_DATA
        BNE     wait_for_zero           ;loop until START is open
        MOVE.W  #1024, D3               ;debounce delay (update uses D0-2)
check_debounce:
        BSR     update                  ;update screen (40us) while looping
        BTST    #START_BIT, CIO_PA_DATA
        BNE     wait_for_zero           ;if it bounces start over
        DBF     D3, check_debounce      ;must stay open 1K loops
           ;now wait for it to be pressed
wait_press:
        BSR     update                  ;update screen while waiting
        BTST    #START_BIT, CIO_PA_DATA
        BEQ     wait_press              ;loop until START is pressed

           ;START has been pressed. Now read control panel, etc.
        BCLR    #END, CIO_PB_DATA       ;clear END output signal
        MOVE.L  #decode_tbl, A0         ;table for dwell/bin and #sweeps
        MOVE.B  CIO_PA_DATA, D0         ;read autoloop and dwell/bin
        BTST    #AUTO_LOOP, D0          ;autoloop switch set?
        SNE     auto_loop_flag          ;if so set flag
        AND.B   #$0F, D0                ;mask hex switch
        ASL.W   #1, D0                  ;convert to a word offset
        MOVE.W  (A0,D0.W), dwell_per_bin ;get table value and save

        MOVE.B  CIO_PB_DATA, D0         ;read #sweeps hex switch
        AND.B   #$0F, D0                ;mask
        ASL.W   #1, D0
        MOVE.W  (A0, D0.W), num_sweeps  ;get table value and save

           ;final setup before data collection
        BSR     clear_arrays            ;clear DATA and NORM arrays
        MOVE.L  #norm_array, A6         ;load dedicated registers
        MOVE.L  #data_array, A5
        CLR.L   D7                      ;clear accumulating register
        MOVE.W  dwell_per_bin, D6
        CLR.B   D5                      ;clear (=256) bin count
        CLR.B   stop_flag               ;it would be premature
           ;setup which interrupt handler entry is being used
        TST.B   auto_loop_flag          ;check for autoloop
        BEQ     free_run_int
        MOVE.L  #wait_trig_int, int5_vect ;handler entry for triggered
        BRA     update_loop
```

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

```
free_run_int:
        MOVE.L  #sweep_start_int, int5_vect ;handler entry for autoloop
           ;enter "main" loop
update_loop:
        BSR     update                  ;update screen
        TST.B   stop_flag               ;see if handler says it's done
        BEQ     update_loop             ;still sweeping
        BRA     main_loop               ;sweep is done
           ;LOOKUP TABLE for decoding rotary hex switches
           ;used for #sweeps and dwell/bin
decode_tbl:
        .word   1, 2, 5, 10, 20, 50, 100, 200, 500, 1000
        .word   2000, 5000, 10000, 20000, 30000, 0

           ;SUBROUTINES
           ;"clear-arrays" -- clears DATA and NORM arrays
clear_arrays:
        CLR.L   D0                      ;zero lives here
        MOVE.L  #data_array, A0         ;pointers
        MOVE.L  #norm_array, A1
        MOVE.W  #$FF, D1                ;counter
clr_loop:
        MOVE.L  D0, (A0)+
        MOVE.W  D0, (A1)+
        DBF     D1, clr_loop            ;fastest looping primitive
        RTS

           ;"update" -- update one more value to DISPLAY array
           ;checks panel control for scale
           ;also updates LED port from memory image (led_store)
           ;registers -- D0 - update offset  D1 - current data value
           ; D2 - scale factor, storage for normalizers  A0 - array ptr
update:
        MOVE.B  led_store, LED          ;update LEDs
        MOVE.L  #data_array, A0         ;raw data base pointer
        MOVE.W  update_offset, D0       ;indexes next point to update
        ASL.W   #2, D0                  ;make into a long offset
        MOVE.L  (A0, D0.W), D1          ;get data
        ASR.L   #2, D0                  ;restore integer offset
           ;get scale factor
        MOVE.B  CIO_PC_DATA, D2         ;read switch
        AND.B   #$0F, D2                ;mask hex value
        CMP.B   #$0F, D2                ;check for autoscale (n=$F)
        BEQ     auto_scale
        ASR.W   D2, D1                  ;else shift right n bits
        BRA     comp_and_save           ;and convert to offset binary byte
auto_scale:
        MOVE.L  #norm_array, A0
        ASL.W   #1, D0                  ;make a word offset
        MOVE.W  (A0, D0.W), D2          ;get normalizer
        ASR.W   #1, D0                  ;restore integer offset
           ;divide by norm and by dwell/bin
        TST.W   D2                      ;test before dividing
        BEQ     comp_and_save           ;dont divide, data is zero anyway
        DIVS    D2, D1                  ;data/norm
        MOVE.W  dwell_per_bin, D2
        BEQ     comp_and_save           ;data is zero anyway
        DIVS    D2, D1
```

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

```
comp_and_save:
        BCHG    #7, D1                  ;complement MSB of display byte
        MOVE.L  #display_array, A0      ;screen array base pointer
        MOVE.B  D1, (A0, D0.W)          ;store finished value in array
        ADDQ.B  #1, D0                  ;increment index, modulo-256
        MOVE.W  D0, update_offset       ;and save
        RTS


           ;INTERRUPT HANDLER(s)
           ;entered every timer-1 tick (100us)
           ;always refreshes display
        ;five entry points --
           ;"bad" - display LED walking bit forever
           ;"idle" - display refresh only
           ;"get_data" - get ADC data, check for end of bin & sweep
           ;"sweep_start" - initialize, load get_data vector, get_data
           ;"wait_trig" - sweep_start if triggered, else idle

bad_int:
        BCLR    #Z_BLANK, CIO_PB_DATA   ;turn off z-axis
        BCLR    #SWEEP_BIT, CIO_PB_DATA ;and SWEEP output
        BSET    #END, CIO_PB_DATA       ;set END output
           ;now make "walking bit" pattern
           ;and loop forever
        MOVE.B  #$01, D0                ;initialize LED value
flsh_loop:
        MOVE.B  D0, LED                 ;send to LED
        ROL.B   #1, D0                  ;circular shift left
        MOVE.L  #$8000, D1              ;reset delay counter
flsh_delay:
        SUBQ.L  #1, D1
        BNE     flsh_delay              ;make a delay
        BRA     flsh_loop

idle_int:
        MOVE.B  D4, DAC0_OFFSET(A3)     ;send X position
        MOVE.B  (A4, D4.W), DAC1_OFFSET(A3)  ;send Y position
        ADDQ.B  #1, D4                  ;incr index
        BRA     z_pulse                 ;make unblank pulse

wait_trig_int:
        BTST    #STOP_BIT, (A2)         ;check STOP switch
        BNE     stop_sweep
        BTST    #EXT_TRIGGER, (A2)      ;check trigger signal
        BEQ     idle_int

sweep_start_int:
        BSET    #LED_SWEEP, led_store   ;gets displayed by "update"
        BSET    #SWEEP_BIT, (A2)
        BCLR    #STOP_BIT, (A2)         ;clear STOP 1's catcher
        MOVE.L  #get_data_int, int5_vect  ;load "getdata" vector

get_data_int:
        MOVE.B  #$03, (A3)              ;start ADC (bipolar, 2's comp)
           ;update display
        MOVE.B  D4, DAC0_OFFSET(A3)     ;send X
        MOVE.B  (A4, D4.W), DAC1_OFFSET(A3)  ;send Y
```

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

```
        ADDQ.B  #1, D4                  ;incr index
        NOP                             ;waste time so ADC can finish
           ;done sending XY pair
        MOVE.B  (A3), D3                ;read ADC, which must be done
        EXT.W   D3                      ;extend byte->word
        EXT.L   D3                      ;extend word->long
        ADD.L   D3, D7                  ;add data to accumulating register
        SUBQ.W  #1, D6                  ;decrement dwell counter
        BNE     z_pulse                 ;we're still in this bin
           ;next bin
        MOVE.W  dwell_per_bin, D6       ;reset dwell counter
        ADD.L   D7, (A5)+               ;add new value to DATA bin
        ADDQ.W  #1, (A6)+               ;incr normalizer
        CLR.L   D7                      ;clear accumulating register
        SUBQ.B  #1, D5                  ;decr remaining bin count
        BNE     z_pulse                 ;if still in sweep
           ;end of this sweep. bin count is zero already
        MOVE.L  #data_array, (A5)       ;reset array pointers
        MOVE.L  #norm_array, (A6)
        BCLR    #LED_SWEEP, led_store   ;turn off sweep LED
        BCLR    #SWEEP_BIT, (A2)        ;and sweep signal
        BTST    #STOP_BIT, (A2)         ;check for manual abort
        BNE     stop_sweep
           ;now check if at end of sweep count
        MOVE.W  num_sweeps, D3
        BEQ     re_trigger              ;already zero -> go forever
        SUBQ.W  #1, D3                  ;otherwise decr and test
        MOVE.W  D3, num_sweeps
        BEQ     stop_sweep
           ;now check if autoloop
re_trigger:
        TST.B   auto_loop_flag
        BNE     re_trig_auto            ;autoloop, needs vector loaded
        MOVE.L  #wait_trig_int, int5_vect  ;trig -- load vector
        BRA     z_pulse
re_trig_auto:
        MOVE.L  #sweep_start_int, int5_vect  ;load autoloop vector
        BRA     z_pulse
           ;stop sweep, manual or end of sweeps
stop_sweep:
        BSET    #END, (A2)              ;set END output signal
        MOVE.L  #idle_int, int5_vect    ;load idle vector
        ST      stop_flag               ;tell main prog we're done
           ;fall through to Z-axis unblanking pulse
z_pulse:
        BSET    #Z_BLANK, (A2)          ;start unblank software-pulse
        MOVE.B  #CT1_CMDSTAT, CIO_CNTRL_OFFSET(A2)  ;need to clear int
        MOVE.B  #$23, CIO_CNTRL_OFFSET(A2)  ;this does it
        BCLR    #Z_BLANK, (A2)          ;end unblank pulse
        RTE                             ;return from exception (int)
           ;end of program
```

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

68008 using 4 clocks for each byte moved: The first instruction requires the CPU to fetch a 2-byte opcode, 4 bytes of extension for the (long) address, and finally the data byte requested; that's 7 bytes, or 28 clocks. The second instruction is a 2-byte fetch, followed by the data byte requested, for 3 bytes total, or 12 clocks. In general, devices with narrow buses (like our 68008, whose internal 32-bit architecture must feel positively claustrophobic in its 8-bit bus outerwear) suffer the most from fetch-intensive catatonia.

Finally the program begins! The first 8 bytes of ROM are the all-important start-up vectors, for the stack pointer and program entry point, respectively. The entry point is up in "true" ROM (it should be at $40008), so we can immediately clear the BOOT bit, which makes RAM replace the phantom ROM image we needed in order to boot. Now we can load interrupt vectors into low RAM, in the specific locations the 68008 dictates (Table 11.5 shows the full vector area): $68 (INT2), $74 (INT5), and $7C (NMI = INT7). We've used only INT5 (from the 100μs timer in the parallel port chip), which we load with the address of our interrupt handler. Different things need to be done by the handler, depending on the overall state of the machine (standby, waiting for trigger, beginning new sweep, or in the middle of a sweep), so we've written one grand handler, with separate entry points according to what needs to be done. At this point we're not ready to take data yet, so we load the address of the ***idle_int*** entry point into the INT5 vector location. It's a good idea to load all the unused interrupts and other vectors with a ***bad_int*** vector(s), in case something goes wrong (divide-by-zero, spurious interrupt, etc.); we loaded them all with a pointer to a routine that will flash something distinctive on the LED display (you'll find out what later).

Now comes the tedious but essential task of initializing the ports. The price you pay for the wonderful flexibility of LSI peripherals like the 8536 is the need to figure out precisely what control bytes to send to which register, and in what order, in order to make it do what you want. This includes choices of direction, polarity, mode, and interrupts, in the case of simple parallel ports, and choices of modulus, cascading, triggering mode, interrupts, etc., for the timers. We've shown the full initializing code for the parallel port/timer in Program 11.3: Our initialization enables

TABLE 11.5. 68000/8 VECTORS

Vector number	Address[a] (hex)	Assignment
0	000	initial SSP } reset
	004	initial PC } reset
2	008	bus error
3	00C	address error
4	010	illegal instruction
5	014	divide-by-zero
6	018	CHK instruction
7	01C	TRAPV instruction
8	020	privilege violation
9	024	trace
10	028	1010 emulator
11	02C	1111 emulator (fl point)
12–14	030–038	reserved
15	03C	uninitialized intr vector
16–23	040–05C	reserved
24	060	spurious interrupt
25[b]	064	level-1 autovector
26	068	level-2 autovector
27[b]	06C	level-3 autovector
28[b]	070	level-4 autovector
29	074	level-5 autovector
30[b]	078	level-6 autovector
31	07C	level-7 autovector
32–47	080–0BC	TRAP vectors
48–63	0C0–0FC	reserved
64–255	100–3FC	user (acknowledged) intr vectors

[a] all are 4-byte ("long") addresses. [b] not available on the 48-pin 68008, in which IPL0' and IPL2' are combined onto one pin.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

parallel ports A, B, and C, and it makes bits 4–6 of port B outputs, and all the rest inputs (see Fig. 11.15). It initializes timer-0 to divide its 4MHz clock by 400, then retrigger continuously, generating interrupts (on INT5) every 100μs cycle. Note that we've made all switch inputs *inverting*, so that a switch (pulled up to +5V, with common to ground) that is closed reads as a 1, not a 0. We took advantage of the "1's catching" input option for the STOP switch, so a momentary closure is latched until we check it at the end of a sweep.

Finally, we clear the arrays in RAM (note use of a subroutine), initialize registers, enable interrupts, and jump into the "main" loop.

□ *Main program: main loop*

Once things are initialized, we enter the main loop forever. It actually consists of two loops: a loop that waits for the START button to be pressed, and then a second loop that continually updates the display memory while the actual data collection goes on underneath via interrupts. When the interrupt routine has finished its last sweep, it sets a software "stop flag," which the second main loop is continually checking. This signals the main routine to loop back to its first loop, its resting state while waiting for START again. Let's look at the flow chart and code.

The main loop (Fig. 11.19) begins by setting the LEDs to standby. Then it looks for the START button to be pressed, i.e., to go from open to closed. This is trickier than it seems, because the switch is not debounced in hardware, and therefore you typically get a few dozen closely spaced closures, extending over perhaps 25ms. That might be enough time to finish the shortest possible sweep (e.g., if you had selected 1 sweep at 100μs dwell/bin) and then erroneously begin again because the switch is still making transitions from open to closed. So we wrote a simple debouncing routine, which waits until the switch has been open continuously for about 50ms (it keeps busy with the ***update*** routine meanwhile), and then changes to a closed state. Now we've got our marching orders! The routine clears the END output signal, and then reads the control panel and uses the values accordingly (setting software flags like "`auto_loop`," and parameters like "`dwell_per_bin`" and "`num_sweeps`"). Note the use of the *decode_tbl* (and indirect addressing with index) to assign values corresponding to the switch positions.

The program then clears the DATA and NORM arrays, initializes some address and data registers, and clears the stop flag. The last step is to change the INT5 vector location (which currently points to the ***idle_int*** handler entry) to either the "***wait_trigger***" entry or the "***sweep_start***" entry, depending on whether the panel controls specified triggered or autoloop mode.

The main program then enters a tight loop in which it alternately calls ***update*** (update the DISPLAY memory from the DATA memory) and checks the *stop_flag*. Of course, interrupts are secretly making everything interesting happen during this insipid loop.

□ *Main loop: subroutines*

As an interlude before attacking the interrupt handler, which is the most complex code of all, let's look at the two subroutines called by the main program (Fig. 11.20). The ***clear_arrays*** routine sets both DATA and NORM to all zeros; it doesn't even bother with DISPLAY, because DATA's zeros will be rapidly copied to DISPLAY by ***update***. This routine updates one DISPLAY value, using current display parameters from the control panel, and input values from DATA and NORM; it also updates the LED port by copying the memory byte `led_store`.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

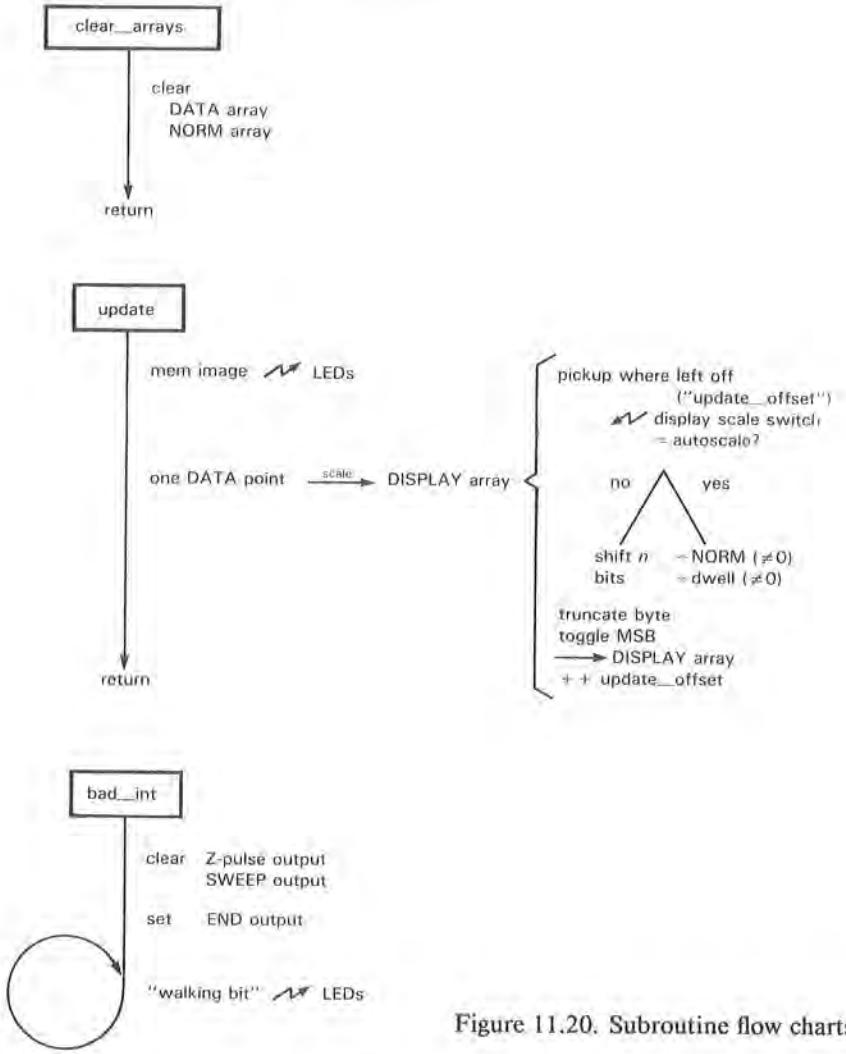


Figure 11.20. Subroutine flow charts.

Look first at the straightforward ***clear_arrays*** subroutine in Program 11.3. Pointers to the two arrays are initialized in A0 and A1, and all 32 bits of D0 are zeroed. D1 is a counter, initialized to the array size *minus one*; we'll see why in a moment. The loop moves either word- or long-sized zeros to the arrays, using indirect addressing (with postincrement); remember that postincrement is smart, and increments the address register by the right amount, in this case adding 2 for a word operation, 4 for a long operation. The DBF instruction is tricky and deserves explanation: It is really the DBcc instruction, with the condition code cc = False. The general `DBcc Dn,label` actually tests two conditions: First it checks the condition cc (i.e., flags set by the previous operation), doing nothing (i.e., falling through to the next instruction) if cc = True; however, if cc = False, it decrements the named register (as a word) and branches to `label` unless the register now equals −1, in which case it falls through. We forced DBcc to become a simple looping primitive by setting cc = F

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

(always false, see Table 11.1), so it always decrements D1, looping until D1 = −1.

In spite of this trickery, and also the restriction that the counter must be word-length, DBcc is handy because it replaces two instructions (SUBQ, Bcc) and is very fast. Since it checks for −1, the counter register must be initialized to one less than the number of loops you want, which explains our initialization. The subroutine ends with the usual RTS (return from subroutine), which restores the old PC (program counter), thus returning to the calling program.

Note that we didn't need to save any registers at the beginning, because the calling program didn't leave anything valuable in D0–D1 or A0–A1. Note also the use of MOVE rather than CLR to zero the arrays; it turns out that MOVE is faster than CLR, because of an idiosyncrasy of the 68000 – to do a CLR, the 68000 does a read followed by a write. The designers did this strange thing in order to simplify the CPU logic.

EXERCISE 11.13

Rewrite *clear_arrays* using SUBQ and Bcc in place of DBF. Rewrite it again using CLR in place of MOVE.

Update is a busier subroutine. Its job is to keep the DISPLAY memory (and also the LEDs) up to date, and it is repeatedly called in both loops of the main program. Since interrupts get priority, their work gets done on schedule (every 100μs), with ***update*** taking essentially all the remaining time. It begins by sending the memory image of the LEDs to the physical LED port. Even this simple operation involves some subtlety. The obvious question you should be asking yourself is why we don't just update the LEDs directly when we want to set or clear a bit (an LED). The answer is twofold: First, we can't just write a new byte to the LED port, because we'll clobber all the other bits; either we need an LED port that we can read as well as write or we need to keep a memory image of the LED. Since we didn't make the LED port readable, we use a memory location ("led_store") to keep a copy of what the LED port was last sent. Second, having been forced to use a memory location anyway, we can save time inside critical loops in the interrupt handler by updating only led_store there, letting the persistent ***update*** get the message out to the panel LEDs in its own good time. When we look at the interrupt handler this will make more sense.

EXERCISE 11.14

Add hardware (very little required) to make the LED port readable. Use cleverness to keep the additional address decoding simple.

The rest of ***update*** updates the DISPLAY memory. First it retrieves from memory the offset (number of points from the beginning of the array) for the next point to update. (It would be nice to use a dedicated address register here, but the speedy interrupt handler got first priority when registers were being given out.) The offset is multiplied by 4 (shift left 2 bits) so that it can be used for indexed addressing within the DATA array of longs. Having fetched the DATA value to D1, we read the panel for current display scale factor, and mask to obtain a number between 0 and 15. The value 15 ($0F) signifies autoscale, while smaller numbers signify fixed scaling by the specified power of 2: We either shift accordingly or branch to the autoscaling code.

For autoscaling we need to divide the current (indexed by update_offset) DATA value by the current NORM value (which tells how many scans are included in the DATA value), and then divide again by dwell-per-bin (which tells how many samples were included in each scan). Before you divide anything, *always test for zero!* Finally, whether we've autoscaled or shifted, we have to convert the resulting signed long to an offset-binary byte.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

In the case of autoscaling, the final long is always in the range ±128; in the case of fixed scaling there will be overflow if you choose a scale smaller than the largest bin; in that case the best thing is to make the points at the top of the display roll over to the bottom on overflow, and vice versa. If you write some numbers down and play around a while, you can convince yourself that the right thing to do is always to truncate to 8 bits, then complement the MSB. We did this by using BCHG (bit change), then doing a byte-length MOVE to the DISPLAY array. Finally, we incremented and saved the update_offset index, and executed the essential RTS.

□ *Interrupt handler*

Now we come to the interrupt handler, which is really the centerpiece of this program. There are four entries to the timer-initiated interrupt; there is also the simple ***bad_int*** handler for spurious interrupts and all manner of other error vectors (Table 11.5). Let's warm up with ***bad_int***, then, with no more excuses left, we'll tackle the timer interrupt handler.

The 68008, as we described earlier, is genetically programmed to recognize both interrupts and the other various "exceptions" listed in the table, and after saving the current PC and status register on the stack, it jumps to the instruction whose address it finds in the vector location corresponding to the exception. Thus, if you try to divide by zero, the CPU will stack the PC and status register, then jump to the instruction whose (32-bit) address is stored in absolute locations $014–$017. Interrupts work the same, with the vectors for fully acknowledged interrupts in locations $100–$3FF, and the vectors for autovectored interrupts in locations $064–$07F. You can do whatever you want in the interrupt handler, and when you're done you execute an RTE instruction (return from exception). In order to prevent pandemonium, the CPU disables interrupts when it jumps to the handler, and reenables them when it executes RTE. If you have a long-winded handler, you may want to re-enable interrupts (at higher-priority levels only) within the handler, which you can do by sending the appropriate byte to the status register.

□ *Bad_int*

In our example, Figure 11.20 and Program 11.3 show the operation of ***bad_int***, whose job is to bring about an orderly shutdown of output signals, then make the LED display do something eye-catching. Its starting address, known to the linker after all the relocatable assembly code has been hooked together, is loaded (by our main program during the boot-up sequence) into all those reserved vector locations (in low RAM) shown in the table. An exception or spurious interrupt (i.e., any except level 5) causes the CPU to do the sequence described above, putting us into the code at ***bad_int***. We first shut off the Z-axis signal, because if we're unlucky enough the exception could have happened right in the middle of the Z-axis software pulse, causing the XY display to stay at full intensity (and in one spot) after the crash. While we're at it, we might as well terminate the SWEEP output and assert the END output, since whatever we were doing is now surely a hopeless cause.

Now for the fun. We send 01_H to the LED, then enter a loop that does a left circular shift, wastes a biologically significant amount of time, then sends the shifted byte out, etc. The net effect is a dazzling "walking bit" display, guaranteed to arouse even the most jaded button pusher. We never do an RTE, so the thing just goes on forever. The operator is obliged to do a total RESET to get things going again.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

EXERCISE 11.15

Think of something smarter, so the operator can find out which exception caused the trouble. Hint: There are slightly fewer than 256 possible exceptions; there are also 8 LED bits. Can you write the code for your brainchild?

□ ***Timer interrupts: four entries***

OK, we've got no more excuses. Let's plunge in. Figure 11.21 and Program 11.3 show the interrupt handler. It has four entry points, according to what state the machine is in. They are called ***idle***, ***wait_trig***, ***sweep_start***, and ***get_data***. The program cleverly causes interrupts to vector to the right entry by changing the interrupt vector (at $074) according to the overall machine state. You enter at ***idle*** if you don't want to collect data; it just displays one point on the screen, then returns. If you enter at ***get_data*** it reads the ADC, checks for end-of-bin or end-of-sweep (processing accordingly), and refreshes the display. ***Sweep_start*** sets appropriate LED and output signals, then falls through to ***get_data***. Finally, ***wait_trig*** checks for a trigger input, entering ***sweep_start*** or ***idle*** accordingly. There are some other labels in the interrupt handler (e.g., ***z_pulse***), but they are not entry points, just targets for loops or branches.

□ *Timer interrupts:* ***idle***. Let's go through the handler in detail, since it is important code. Early in the main program the vector is set to the ***idle*** entry, to create a screen display while waiting to start. Thus, execution commences at the ***idle_int*** label. With our reserved registers, everything is simple. D4 holds the index of the next screen point to be refreshed, so we send it to the X-coordinate converter, DAC0 (addressed indirectly with offset, which is faster than absolute). Then we send the data value (using D4 to index from A4, the DISPLAY base pointer) to the Y-coordinate converter, DAC1. D4 is incremented (but not checked for end-of-array), and control jumps to the Z-axis software pulser.

EXERCISE 11.16

Why can we get away without checking after incrementing the DISPLAY index, D4?

By this time, the X and Y DACs have settled (1μs settling time), so the Z-axis pulse routine uses BSET (bit set) to set the Z_BLANK bit (bit 4, see definitions) of parallel port B, an often-used port whose address we keep at the ready in A2. We could clear the bit next, but the rather short pulse that would result (3μs) would give you a dim display (3μs unblanking every 100μs). However, since all interrupts return via this code, we can use this opportunity to do a good deed while killing some time, namely telling the timer that it can now disassert its interrupt request. Writing to the timer-1 command/status register is a two-step process (as it was in the main program's initialization code), in which we first send the register's internal address ($0A) to the chip's control register ($83000), and then we send the actual controlling byte ($20), which the 8536 interprets as a command to disassert its timer-1 interrupt request. Now there's nothing left to do before returning from the interrupt, so we clear the Z-axis pulse (with a BCLR) and execute an RTE (return from exception). By putting the awkward interrupt acknowledge code within the Z-axis loop, we've managed to stretch the unblanking pulse out to 10μs, repeated every 100μs. Since we had to acknowledge the interrupt anyway, this is a great place to do it. We'll see a similar serendipity when we send the X,Y pairs out during the ADC conversion, coming next.

□ *Timer interrupts:* ***get_data***. This is the entry point used most of the time, namely when the signal averager is in the midst

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

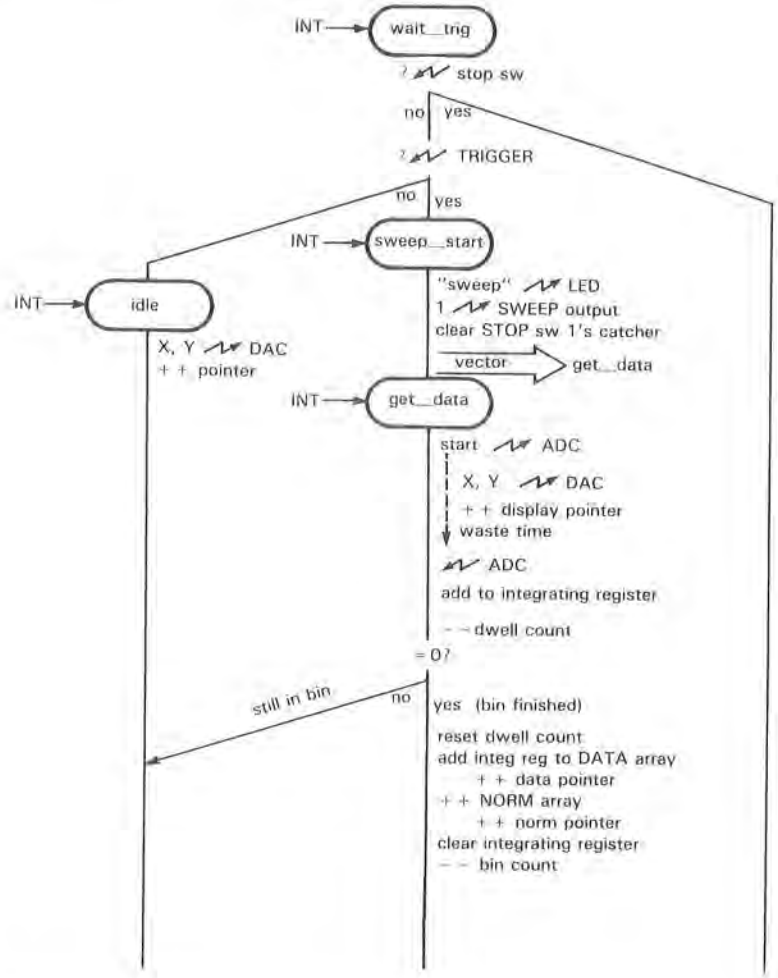
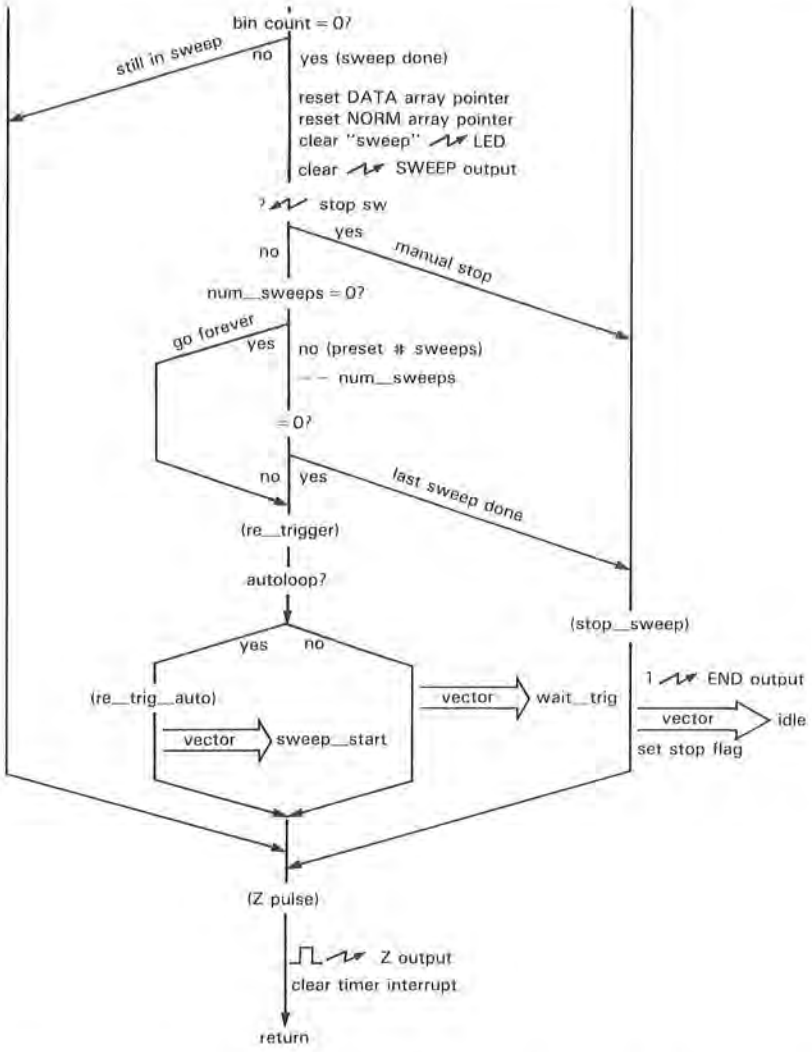


Figure 11.21. Interrupt handler flow chart.

of a sweep. We start the ADC by sending a mode byte ($03) to it; $03 specifies bipolarity 2's-complement conversion. As before, for maximum speed we use indirect addressing through A3 (which holds the ADC's address). Now we have to wait 10μs for the conversion, a perfect opportunity to send a new X,Y pair to the display DACs, with identical code as used in ***idle***. We're done a microsecond early, so we waste time with a NOP (no operation), then read the ADC. Notice how much better this is than having a readable hardware status bit signaling ADC conversion done, as we remarked in the circuit design discussion (Section 11.05); note, however, that we'd have to remember to add some more NOPs if we later decided to use a faster CPU clock.

We've read a 2's-complement byte from the ADC, but our DATA array and bin accumulator (D7) use 2's-complement *longs*. So we use the EXT (sign extend) instruction twice to make a long integer. Sign extension is simply the replication of the MSB leftward until the larger integer word is full; it preserves the value of a signed integer (simple zero-filling does not).

Petitioner Intel Corp., Ex. 1037
IPR2023-00783



The extended integer is now added to the accumulating bin value in D7, and the `dwell_per_bin` counter (D5) is decremented. If it is still nonzero, we return via ***z_pulse***, described above. The total time in the handler in this case is 32.3μs, plus 9μs for the CPU interrupt processing and 5μs more for the RTE, for a grand total of 46.3μs. Thus, the main program gets more than half the CPU processing power for its simple DISPLAY update task.

If the bin accumulation is done, the handler resets the dwell count, adds the accumulated value in D7 to the DATA array (as pointed to by A5), increments the NORM value (through A6), clears the accumulating register (D7), decrements the bin counter, and (if nonzero, i.e., scan not done) jumps to ***z_pulse***. Note the use of auto-incrementing addressing modes. The additional time spent in the handler is 14.8μs.

If the *sweep* is also done, as indicated by zero in the remaining-bin-count register D5, the handler resets pointers, LEDs, and signal outputs. Then it checks to see if the STOP switch has been pushed; you always do the checking at the end (or beginning)

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

of a sweep, so that all data are always averaged for the same number of sweeps. If STOP has been pushed, the routine jumps to ***stop_sweep***, which sets the END output and stop flag, and loads the ***idle*** entry point into the INT5 vector.

If the STOP button hasn't been pushed, the routine checks to see if it should quit anyway, since the panel-selected number of sweeps may have been completed (the number remaining is kept in the memory variable `num_sweeps`). Since the value 0 means "sweep forever," we first check for zero; if `num_sweeps` is zero, we leave it alone and jump to ***re_trigger***, otherwise we decrement and again check for zero. If it is now zero, we have done the last preprogrammed sweep, so we jump to ***stop_sweep***, otherwise we jump to ***re_trigger***.

Re_trigger's job is to worry about how to start the next sweep. If the variable `autoloop`, set by ***main*** after reading the control panel, is true, it loads the ***sweep_start*** entry into the INT5 vector, otherwise it loads the ***wait_trig*** entry. Note that there's no danger of an interrupt occurring while the vector is being changed, because interrupts are turned off by the CPU during interrupt processing; since we don't turn them on in the handler, they stay off.

□ *Timer interrupts:* ***sweep_start*** *and* ***wait_trig.*** These entries are used if the next interrupt should begin a sweep, or if instead we're waiting for an external trigger pulse (at least 100μs duration!). The corresponding vector will have been loaded into the INT5 vector location, either by the main program when the START button is pushed or by the interrupt handler when it finishes a sweep that is not its last (at ***re_trigger***); look at the flow charts to remind yourself where this happens.

Sweep_start begins sweeping immediately, and is easy: It sets the SWEEPing LED and output signal, clears the latching (1's catching) STOP-switch bit, loads the ***get_data*** vector, and falls through to ***get_data***. Subsequent interrupts cause entry at ***get_data***.

Wait_trig is the entry point if the next sweep should not begin until a trigger signal is received (parallel port A, bit 7). Since STOPping should override starting, it looks first at the STOP switch (which causes a jump to ***stop_sweep***), then at the trigger input; if there is no trigger, it jumps to ***idle***, otherwise it falls through to ***sweep_start***.

□ 11.08 Performance

You can design a microprocessor-based instrument so that all the essential data-taking is handled entirely by fast hardware, with the microprocessor taking a role only in setup, display, and readout. In that case the instrument will run as fast as the hardware permits, with the microprocessor adding convenience and flexibility. You pay a price, of course, in hardware complexity and cost; you may also reduce the instrument's flexibility, owing to the fixed hardware architecture. If instead you simplify the hardware and use the processor to handle the data in real time, as we have done in our example, you can reduce the hardware complexity and maintain flexibility, at the probable expense of speed. In many cases speed is not important, though, so the choice is simple.

In our case the basic sampling rate, and therefore the maximum channel-advance (bin-stepping) rate, is limited by the processor speed. Each interrupt service has to be completed before the next one comes along. When we designed this example, we looked at what had to be done and estimated (based on a gut feeling) that 100μs was probably enough time. We didn't know for sure, though, but we decided that we could tolerate a slower sampling rate if that were necessary. Let's look at the numbers.

The 68008 data "sheet," which is really a 100-page book, includes tables of the

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

instruction execution times (in clock cycles). We've used those tables to calculate the execution speeds we've quoted above. Here are the times (including interrupt vectoring and return) for the critical interrupt handler:

Entry	*Time (μs)*
idle	37
get_data	46.3 (within bin)
	61 (end of bin)
	92 (end of sweep, manual STOP)
	105 (end of sweep, programmed stop)
	113 (end of sweep, wait for trig)
	114 (end of sweep, autoloop)
sweep_start	61
wait_trig	46 (no trigger)
	69 (trigger)

Most of the times are less than the 100μs "heartbeat" of the signal averager, which is fine. There are three conditions that lead to a >100μs interrupt service time, however. For the first one (end of sweep, programmed stop) it doesn't matter, since you don't care about an extra few microseconds once all the data have been collected. Likewise, you probably don't care about the second bad case (waiting for an external trigger), since such a trigger would always allow some resting time before beginning each sweep. However, the last bad case (end of sweep, autoloop) could be serious, since in autoloop mode you expect the overall period to be exactly 256 × (dwell/bin). However, the situation is actually OK, for the following reason: When you use a signal averager in autoloop mode, you always trigger the external experiment from the averager (that's why we provided a SWEEP output signal), so you don't care if the period is a fraction of a percent different from what you expect. However, if you insist on getting done in <100μs, just substitute a 68008 running at 10MHz, which will multiply all the times by 0.8, making even the slowest time less than 100μs (91μs, in fact). If you do use a faster processor, be sure to fiddle with the interrupt handler code so that the ADC still has enough time to convert.

In summary, our guess that the 68008 would let us run at a 10kHz sampling rate was correct. The authors are particularly pleased, because we had written everything up to the last paragraph before we learned the good news! It's also obvious that you could not hope to speed up this instrument to a 20kHz rate without going to an all-hardware data collection scheme.

11.09 Some afterthoughts

In this design example we made a number of choices, both in the way we designed the hardware and in the way we implemented the software. In many places we could have done things differently. Most of the time the "best" choice is pretty obvious, but there were some cases where another method would have worked as well; in such situations we generally made our choice to maximize clarity, illustrate the most generally useful methods (avoiding fancy tricks and hardware idiosyncrasies), and minimize the length of assembly code. In real life (as opposed to book-writing) it's OK to use tricks, taking advantage of special features in the hardware, and it's also OK to write a lot of code. Here are some places where a different choice makes sense.

Switch-reading via table

Our program used a block of "in-line" code to read and test the various bits from the control panel, setting software parameters accordingly. That's a perfectly good way to do it. But a nice alternative, and one that is particularly easy to modify, uses instead a short *loop* to cycle through the panel's bits, with the port addresses, bit locations, and corresponding affected software variables specified by *tables*. Because such a scheme requires more explanation, and in our example would probably involve

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

more lines of code, we opted for the simpler preprogrammed in-line switch-reading code. You should consider a table-driven scheme, though, in any application with a large number of parameters, particularly if you are likely to change the assignments or values of the input bits.

☐ *One-shot Z-unblank*

We used a parallel port "software pulse" for the scope unblanking, because we wanted to illustrate this important technique. We particularly wanted to make the point that you cannot safely make software pulses when interrupts are turned on. An alternative is to use a hardware pulse generator, for example a monostable IC (rather than a parallel port bit). Monostables generally leave us with a bad feeling. However, in this case the powerful Zilog 8536 CIO comes to the rescue, with a built-in monostable that can be used to drive any output bit. Its "monostable" is actually one of the three on-board timers, which you program for pulse length (you can even cascade two timers to get longer pulse lengths). In our application there are timers to spare, so this method is ideally suited: By using the 8536 one-shot, you reduce the code inside the interrupt handler, and you can also choose the longest possible Z-unblanking pulse width.

☐ *STOP-switch 1's catcher*

To read the STOP switch, we took advantage of a unique feature of the 8536, namely a built-in "1's catcher" flip-flop. When the 8536 is initialized, any input port bit can be made 1's catching; such a bit is then set by a momentary switch closure and holds that state until reset by a software WRITE to that port bit. This is perfect for our application because we only want to know *at the end of a sweep* if the STOP button has been pushed since the completion of the last sweep. Because a sweep can take many seconds, the memory feature saves us the trouble of repetitively reading the STOP switch; in our code, therefore, we look at the STOP bit only at the end of a sweep (see Fig. 11.21, the interrupt handler flow chart).

Since most parallel port chips don't have input memory, you may have to do the hard work we avoided. Here's what you would do: First, define an internal software flag, which you might call "`stop_at_end`"; in our code you would put the definition next to `stop_flag`. Be sure to clear this software flag before entering the data-taking loop; a good place would be after reading the control panel. Then add a few instructions in ***update_loop*** to check the STOP_BIT input repeatedly, setting `stop_at_end` if the STOP button is pressed. Finally, change the code in the interrupt handler to check this software flag, rather than the STOP switch, at the end of each sweep.

EXERCISE 11.17

Pencil in these changes to the assembly-code listing we've provided.

☐ *Interrupt handler: multiple entries versus flags*

We used multiple entry points into our interrupt handler, one for each of four possible machine states (idle, waiting for trigger, beginning sweep, getting data). Since the handler is not a called subroutine, but rather a vectored entry at each interrupt, the program altered the entry point by loading the corresponding interrupt vector (in low RAM) each time it changed state. An obvious alternative is to have a single handler entry point, with flag-checking code in the handler to decide what to do. The program then tells the handler what to do by changing this software flag (rather than by fiddling with the handler entry vector). This scheme has the virtue of simplicity, but it takes longer to execute, since every entry involves tests and branches.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

The difference isn't all that significant, though, so you should feel free to use flags to alter handler function if you like that method better.

□ *Serial port: data dump and slave control*

As we remarked in the "non-features" section (11.06), our signal averager lacks the essential capability of sending its averaged data out to an external computer! The code for this isn't terribly difficult, just tedious, involving both an initialization procedure analogous to that for the 8536 and a parsing and handshaking procedure so that the data recipient can trigger and acknowledge the transfer of data.

Assuming that the serial port has been set up for connection to a computer, it makes sense to use the same port as an alternative control panel, so that the external computer can set parameters and trigger data collection. To do this, the parsing routine would look for certain bytes that the computer sends to tell the averager that it wants control. Additional bytes would specify parameters (dwell per bin, number of sweeps, etc.), unconstrained by the limited number of switch positions in our panel-reading design. The software should be designed, of course, so that the panel switches take over if the computer does not ask for control. This gives us the best of both worlds, namely the simplicity of front-panel switches with the optional flexibility of computer control.

Reading panel knobs

In our microprocessor instrument we managed to avoid the complexity of panel *knobs*, sticking with the simpler option of panel *switches*, each of which drove a single parallel port bit. This kind of laziness on the part of instrument designers has led to an unfortunate tendency to omit knobs entirely, substituting instead (for example on a microprocessor-controlled oscillator) a pair of "up" and "down" buttons. If you're like us, you miss the nostalgic feel of a real rotary control. We could make our signal averager more civilized by having a knob that selects a bin on the screen, causing a display of the address and accumulated count.

The easiest way to regain control in a microprocessor instrument is to use an ADC input to convert the voltage from a panel potentiometer that is connected between +5 volts (or some better reference) and ground. There are inexpensive and small 8-bit ADCs with on-chip 8-input multiplexers and S/H; typically you have a few inputs left over, and you can use them to read several panel controls. In fact, you can even use an ADC input to read out the state of an n-position rotary switch – just string a resistive divider chain of n-1 equal resistors along the switch contacts, and use the ADC to read out the voltage!

If you need more analog readout resolution than a simple 8-bit ADC provides, consider instead a *rotary encoder*. This typically consists of a panel-mounting control no larger than an ordinary panel potentiometer, containing a pair of optical interrupters that provide quadrature pulses (90° out of phase) as the knob is turned. By providing pulses in quadrature, the encoder lets you determine which way the knob is being turned (see Fig. 8.97). Unlike an ordinary potentiometer, a rotary encoder has no limit stop, so you can turn the shaft through multiple rotations. A typical unit like the Bourns EN series produces 256 pulses per revolution.

MICROPROCESSOR SUPPORT CHIPS

In our microprocessor circuit there are 22 ICs, of which 10 are powerful LSI functions (CPU, memory, parallel and serial ports, calendar clock, and ADC/DAC converters), with the remaining 12 (a clear majority) consisting of lowly gates, flip-flops,

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

buffers, and latches. The latter small- and medium-scale chips are needed to stick the big chips together and are sometimes called "glue logic." You can reduce the glue substantially with PALs, or, in large-production situations, with custom or semicustom chips designed for the task. Nevertheless, latches and three-state buffers, in particular, are used extensively in all microprocessor systems, and so it is worth taking a quick look at some of the choices you have. After that we'll deal briefly with some common LSI support chips (serial and parallel ports, and converters), then conclude the section with a discussion of memory.

11.10 Medium-scale integration

Latches and D registers

We mentioned latches and registers briefly in Section 8.24. The term "latch" strictly refers only to a *transparent latch*, whose outputs follow the respective inputs while enabled. A so-called *edge-triggered latch* is properly called a *D*-type register and consists of an array of *D* flip-flops with common clock. The difference has important consequences when latching data from a bus, because of the relative timing of DATA and its corresponding WRITE strobe. In particular, with some microprocessor buses (e.g., the IBM PC) the DATA is not necessarily valid at the leading edge of the write strobe, but it is guaranteed to be valid (and to have been valid for some minimum setup time) by the trailing edge of the strobe; see Figures 10.6 and 11.22. If you use a transparent latch, enabled during the entire strobe, you will most likely get transient states at the output, as shown. By comparison, the outputs of a *D*-type register (clocked on the trailing edge in this case) change state at the clock edge and are guaranteed not to have any glitches. It's an important fact that an output bit that has not changed state since the previous WRITE will have no momentary spikes or transients; this means you can safely use the various output lines of a latched byte to generate data and strobes for following circuitry.

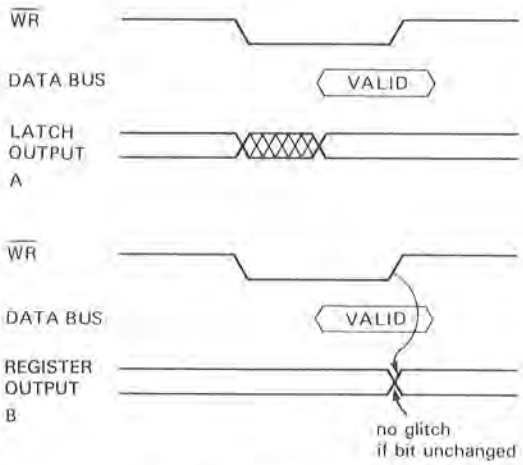


Figure 11.22. WRITE-cycle timing.
A. Transparent latch.
B. Edge-triggered *D* register.

When choosing between latches and registers, note that valid output data is available sooner with a transparent latch, which is sometimes important. Note also that many buses (e.g., the 68008) provide valid data throughout the strobe, including some setup and hold time, and with these you can clock a *D* register on the leading edge if there is enough setup time. On such a bus, of course, a transparent latch will not have transient output states.

There is a nice variety available in both D registers and transparent latches, with features such as RESET input, "broadside" pinouts (all inputs on one side, outputs on the other), inverted outputs, three-state outputs (useful for driving buses), and separate input enable. The latter lets you simplify external gating by driving the clock input with the write strobe and the input enable with the address decoding logic. Look back to Table 8.9 for a listing of features. Some perennial favorites

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

are the octal three-state '373 (latch) and '374 (*D*-type), now modernized into the broadside '573 and '574. These are all packaged in 20-pin DIPs. In that same package the '273 is a '374 with RESET (but not three-state), and the '377 is a '374 with ENABLE (but not three-state). The newer 24-pin "skinny-DIP" packaging maintains the convenient 0.3 inch width while freeing up some extra pins. Thus, the modern '821 series includes 8- and 9-bit registers and latches with ENABLE, RESET, *and* three-state outputs, all in broadside pinout.

Note that a few of these 20-pin or 24-pin narrow chips may be a better choice than a fancy 40-pin (0.6 inch wide) LSI parallel port chip in many applications. For example, in our microprocessor design we used a '273 octal register for the LED port, and a '240 octal buffer for the DIP-switch port. The alternative would be a parallel port LSI chip like the Zilog 8536 (though we'd probably use the simpler Intel 8255 for such a trivial application), which costs more, uses up more space and power, and requires extra programming. In fact, the LSI chip also has poorer output drive (1.7mA sink and negligible source for the 8255, compared with 8mA source/sink for the 'HCT273). Some of the MSI latch/register chips are real brutes when it comes to output drive: The 'AC(T) family can sink/source 24mA (with rail-to-rail swings), and the 'AS821 series can source 24mA and sink 48mA. The LSI chips show their true colors, on the other hand, when you need fancy operating modes (interrupts, tricky input and output modes, etc.) or lots of programmable flexibility, rather than sheer muscle.

Buffers

Another chip used by the bucketful in microprocessor system design is the three-state buffer. You use it for asserting data and address information onto the bus. Most often you're simply sending data to the CPU, as with our simple DIP-switch input. As with latches, there are 20- and 24-pin versions with 8-bit (or more) width. Various features (see Table 8.4) include input hysteresis (to suppress noise), inverted output, broadside pinout, and separate enable inputs for bidirectional use. Special bidirectional buffers with a DIRECTION input and an ENABLE input (rather than a pair of enable inputs) are also available and are usually called *transceivers*; see Table 8.5 for a listing. Figure 11.23 shows a bidirectional buffer used to fortify the relatively feeble (~5mA) microprocessor data bus so it can drive the wiring and input capacitance of a board full of chips; such buffers are mandatory in a micro*computer*, in which a CPU board has to drive relatively high currents into a system bus (backplane) of high capacitance.

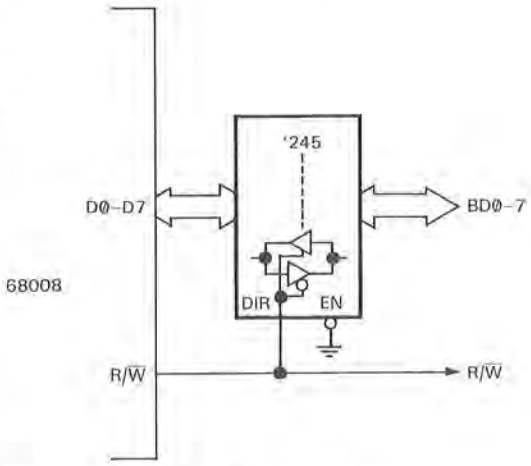


Figure 11.23. Data bus transceiver.

In many cases you can find MSI chips that combine a three-state output buffer with some other function you need; for example, you can connect a counter or latch, or even an A/D converter with three-state outputs, directly to a microprocessor bus. Our microprocessor example illustrated that method with its A/D converters.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

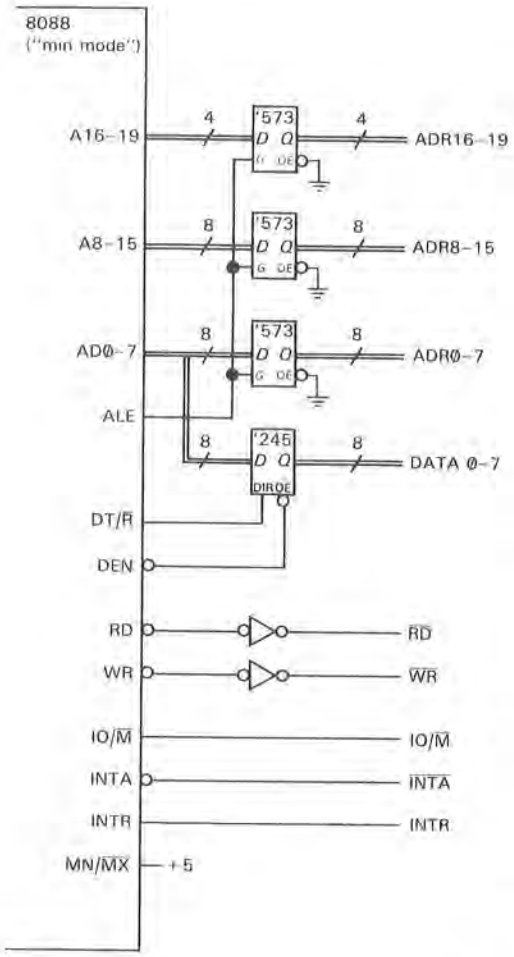


Figure 11.24. Multiplexed DATA/ADR bus.

In Figure 11.24 we show another example of bus buffering: Some microprocessors (e.g., the 8086 and 8088) combine pin functions to save space, multiplexing both DATA and low-order ADDRESS lines onto the same pins. An output called *address latch enable* (ALE) signals valid address and is used to enable a set of latches, as shown. You don't need to latch the DATA, however, because valid RD′ and WR′ strobes occur only when DATA is valid anyway. Note the use of a '245 transceiver to buffer the bidirectional DATA lines.

11.11 Peripheral LSI chips

General characteristics

As we mentioned earlier, LSI chips intended for microprocessor support are usually constructed with NMOS or CMOS technology, and they are usually supplied in large packages with 28 or 40 pins. They tend to be designed with lots of flexibility, often with programmable parameters of operation. Although they are usually designed for specific microprocessors, their generality allows you to use one manufacturer's support chips with another's CPU; thus we combined an Intersil calendar clock and two Zilog ports with a Motorola CPU in our microprocessor circuit. LSI peripherals tend to be expensive at the time of their introduction (the 8530 costs about $25, for example), often costing more than the CPU itself, but they show the usual exponential decay in price that is characteristic of IC technology (and precious little else in this world!). Figure 8.87 illustrated this apparently universal law of "Silicon Valley" (a section of the San Andreas Fault between San Francisco and San Jose).

Although we cast some disparaging remarks in their direction, many LSI support chips are virtually indispensable: Disk and video controllers are obvious examples. Another widely used type of support chip is the universal synchronous/asynchronous receiver/transmitter, USART for short.

How to use a USART

A USART is a microprocessor-controlled serial port chip, for example the Zilog 8530 in our design. Good USARTs include a programmable baud-rate generator, flexible control of bit format (number of bits, parity, etc.), and, in the most advanced USARTs, powerful synchronous modes of operation (with names like HDLC and SDLC), along with a choice of modulation methods (NRZ, FM, Manchester), clock

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

recovery, error checking, etc. Most USARTs include interrupt hardware, and some even support block data transfer to the CPU via DMA. Most CPU families have their own USART, though only a small effort is involved in adapting a different USART. For example, IBM chose the National 8250 UART (asynchronous only), rather than Intel's 8251 USART, to go with the Intel 8088 in their PC. We chose Zilog's 8530 (also used in the Macintosh computer) because of flexibility, availability, and price, and we will use it to illustrate USART interfacing and programming.

USARTs are most often used to send data to and from terminals, modems, or hard-copy devices (printers, plotters), or directly between computers, where the major requirement is universal compatibility and simplicity of interconnection. The usual method is to use serial ASCII transmitted via bipolarity RS-232 levels, as described in Section 9.14 or 10.19. For this simplest sort of communication the USART is operated in the asynchronous mode, with each 8-bit character sandwiched between a START and STOP bit, and transmitted as a 10-bit serial string at a standard baud rate; for this job the 8530 is overqualified.

The 8530 comes in a 40-pin package (Fig. 11.25), and it communicates with the CPU via a set of *processor interface* lines, while independently communicating with the outside via a set of *communication interface* lines.

Processor interface. The 8530 uses 8 bidirectional DATA lines to connect to the CPU's bus, and the usual pair of strobes (RD′, WR′) and chip enable (CE′) for programmed I/O. The A/B′ input tells which of the two USART channels is being accessed, and the D/C′ input tells if the transfer is data (D/C′ HIGH) or control/status information (D/C′ LOW). As with the 8536, there are actually multiple control/status registers, accessed via a pair of successive transfers (look back at the 8536 initialization code). In the usual circuit implementation, the A/B′ and D/C′ lines are simply connected to the low-order CPU address lines, which map them into address space starting at the base address (as determined by the device address decode logic) of the USART. Finally, the processor interface includes four interrupt lines.

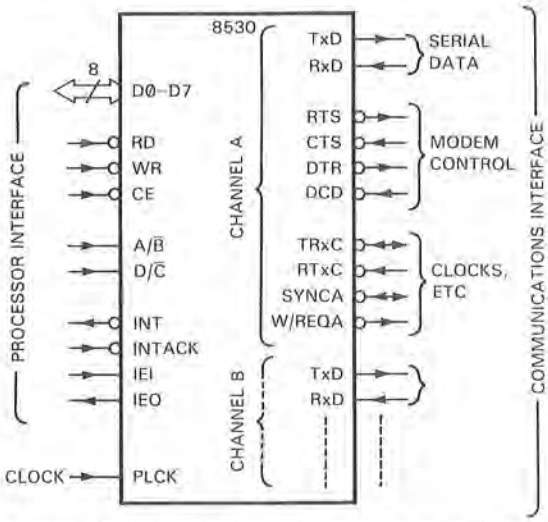


Figure 11.25. Zilog 8530 "serial communications controller" (UART) signals.

Communication interface. Each USART channel (labeled A and B) has transmitted and received serial data (TxD, RxD), and "modem control" handshaking lines (RTS, CTS, etc.), just like you find on the back of a computer. In addition, there are less familiar clock lines used for synchronous communication only (TRxC, RTxC). Finally, the USART requires an external oscillator signal at an integral multiple of 32 times the highest baud rate.

The USART doesn't know anything about bipolarity RS-232 levels, so you have to use RS-232C drivers and receivers on all these lines. For decades the classic RS-232 interface chips have been the bipolar 1488

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

(quad driver) and 1489 (quad receiver); however, in our microprocessor design we used the CMOS MAX233 (dual driver + receiver) because it conveniently has on-chip flying-capacitor voltage doublers and inverters, therefore running from a single +5 volt supply. Note that we didn't bother with the modem control lines (RTS, CTS, DSR, DTR); they're ignored by most of the world anyway, which uses "soft" handshaking (ctrl-S, ctrl-Q) embedded into the data stream instead.

Software. As we said at the outset, the operating modes of the USART are controlled by software commands. A byte sent to the USART in command mode (D/C′ LOW) is interpreted by the USART as a control command, setting the mode of operation. You can choose, for example, synchronous versus asynchronous operation, the number of STOP bits, type of parity, and so on. The simple USARTs of yesteryear were easy to program, with a single control register; fancy chips like the 8530 have literally dozens of registers and require the services of a PhD to program. Unfortunately this sort of complexity is the price you pay for the extreme flexibility of powerful LSI microprocessor support chips.

To illustrate, let's look at the command sequence to initialize an 8530 for asynchronous serial communication on channel A, at 1200 baud, 8 bits, no parity, and one stop bit; we'll also disable interrupts. The full initialization is somewhat long-winded; we'll therefore display the full sequence, but only show in gory detail how we figured out one or two of the critical command bytes. Table 11.6 lists the writable and readable registers in the 8530, which, as we've explained, are accessed by first writing the register *number* as a command (D/C′ LOW), then writing (or reading) from that register. This two-step process is waived for the transmit/receive buffers (WR8 and RR8), because they are needed for each transmitted byte; for these, a simple read or write with D/C′ HIGH does the trick. Likewise, the buffer status byte needs streamlined access, since you probably want to read its flag for each transmitted or received byte; the 8530 accommodates you by letting you read RR0 as a simple command/status read (D/C′ LOW). We'll see how this all works shortly, with some simple assembly language routines.

TABLE 11.6. ZILOG 8530 REGISTERS

Register	Function
Read registers	
RR0	xmit/recv buff status and extnl status
RR1	special receive condition status
RR2	unmodified intr vector (ch A); modified intr vector (ch B)
RR3	interrupt pending bits (ch A)
RR8	receive buffer
RR10	miscellaneous status
RR12	baud rate gen count (lower byte)
RR13	baud rate gen count (upper byte)
RR15	external/status interrupt info
Write registers	
WR0	initialization, pointers
WR1	intr and xfer mode definition
WR2	interrupt vector
WR3	receive parameters and control
WR4	misc parameters and modes
WR5	xmit parameters and control
WR6	sync chars or SDLC address field
WR7	sync char or SDLC flag
WR8	xmit buffer
WR9	master intr control and reset
WR10	misc xmit/recv control bits
WR11	clock mode control
WR12	baud rate gen count (lower byte)
WR13	baud rate gen count (upper byte)
WR14	misc control bits
WR15	external/status interrupt control

Each bit of each register signifies something. For example, Figure 11.26 shows WR3 and WR4, which are used to set up various communication options. The only bits of WR3 that affect asynchronous operation are D0, which enables the receiver, D5, which enables hardware

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

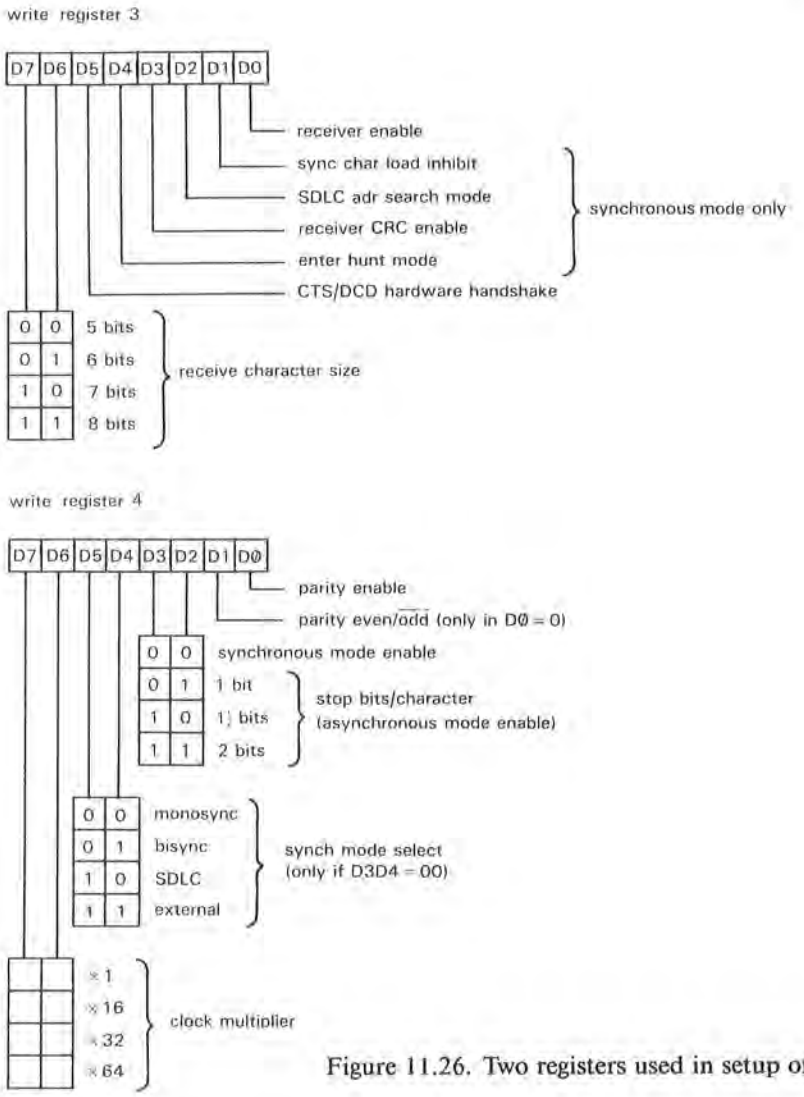


Figure 11.26. Two registers used in setup of Zilog 8530.

handshaking (see next section) via the "modem control" signals CTS and DCD, and the two high-order bits, which select the number of bits/character. The rest have to do with synchronous modes, which we're going to disable by appropriate choice of D2–D3 in WR4. So we make (D7,D6) = (1,1), D5 = 0, and D0 = 1, i.e., we send the hex byte C1 to WR3. For WR4 we select ×16 clock mode (the minimum divisor for asynchronous operation – the USART has to sample in the middle of each bit cell, so it needs the input clock to be a multiple of the baud rate), 1 stop bit/character (the standard choice, except for the obsolete 110 baud teletype standard), and no parity; thus we send the hex byte 44. Note that D5–D4 don't matter, having to do with synchronous options; also, D1 (parity choice) is irrelevant if D0 (parity enable) is 0. Note also that choosing 1 stop bit/character (which makes sense only in asynchronous mode) automatically disabled the synchronous mode, and with

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

it all control register bits that specify synchronous options (e.g., D4–D1 of WR3).

In similar manner you navigate the remaining control registers. It's pretty boring work, and it's easy to make a mistake. Table 11.7 lists what we believe are the correct bytes (we tried it out to be sure!). Note the XXs ("don't care") for registers that are ignored because we aren't using interrupts or synchronous mode. WR9 does a total chip reset, which must precede all other commands. WR12 and WR13 set the 16-bit divisor for the baud rate generator, which for the 8530 is given by

$$\text{baud rate} = f_{\text{clock}} / [2(\text{clock mode})(\text{divisor} + 2)]$$

Thus with a 4MHz clock and ×16 clock mode, we need a divisor of 102_{10} for 1200 baud (this actually produces 1201.92 baud, which is close enough). Our choice of clock frequency allows any of the standard baud rates up to 9600 baud (for which the divisor should be set to 13).

TABLE 11.7. 8530 SERIAL PORT INITIALIZATION[a]

Register	Byte[b]	Result
WR0	(reg pntr)	use to access WR1-WR15
WR1	00	disable interrupts
WR2	XX	interrupt vector (if enabled)
WR3	C1	8 bits, Rx enable
WR4	44	x16 clk, 1 stop bit, no parity
WR5	68	8 bits, Tx enable
WR6	XX	synch char (sync mode only)
WR7	XX	synch char (sync mode only)
WR8	(xmit buf)	Tx data (direct access via D/C'=1)
WR9	C0	reset
WR10	XX	misc sync mode options
WR11	50	Tx, Rx clock from baud generator
WR12	102_{10}	baud rate divider – high byte
WR13	00	baud rate divider – low byte
WR14	03	enable baud rate generator
WR15	XX	interrupt options (if enabled)

[a] for 1200 baud asynchronous, 8 bits, no parity.
[b] XX = "don't care"; all are hex except WR12.

Note that there are numerous other possible operating modes under your control, set by corresponding sets of initialization control bytes. The 8530, like all USARTs, also permits synchronous communication in various modes at rates to 1Mbit/s; this is particularly useful for communication between a pair of processors. Such an application doesn't make much sense for our processor design, because of its small RAM, though it might come in handy for attaching a hard-disk drive.

Note that the correct initialization bytes must be sent to the USART by the CPU before any serial data are transmitted, just as with the 8536 parallel port chip; since we didn't use the serial port in our example earlier, we omitted the initialization. Program 11.4 shows how the initialization would go in this case. This sort of complexity (including deciphering the data sheet in order to figure out those crucial bytes) is the price you pay for the extreme flexibility of these microprocessor support chips.

After setting up the USART operating mode via the control registers, actual bytes of data are sent and received by CPU writes and reads with D/C′ HIGH. The status register must also be interrogated (D/C′ LOW) to determine when the USART has a new received data byte to be picked up by the CPU (D0 of RR0 set) or when it can accept a new byte to be transmitted (D2 of RR0 set). In addition, other bits of the status registers tell if a parity error was detected, if incoming data were lost, etc. You often ignore these latter dire indications of doom and plunge boldly ahead. Program 11.5 is an example, with the same register mapping into address space as above.

Note that these are the simplest forms of handlers, using programmed I/O to check for status information (see Sections 10.06–10.08). These hold up CPU operation by looping on the status flags. Input, in particular, would benefit from an

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

Program 11.4

```
   ;serial port initialization
   ;serial port addresses
CTRL_A  EQU     $85001  ;channel A control
CTRL_B  EQU     $85000  ;channel B control
DATA_A  EQU     $85003  ;channel A data
DATA_B  EQU     $85002  ;channel B data
   ;initialize port A (but see warning in text)
        MOVE.L  #CTRL_A, A0     ;port address, used often

        MOVE.B  #9, (A0)        ;first WR9, to reset chip
        MOVE.B  #$C0, (A0)      ;reset both channels

        MOVE.B  #4, (A0)
        MOVE.B  #$44, (A0)      ;x16 clk, 1 stop bit, no parity

        MOVE.B  #1, (A0)
        MOVE.B  #0, (A0)        ;disable interrupts

        MOVE.B  #3, (A0)
        MOVE.B  #$C1, (A0)      ;Rx 8 bits/char, enable Rx

        MOVE.B  #5, (A0)
        MOVE.B  #$68, (A0)      ;Tx 8 bits/char, enable Tx

        MOVE.B  #11, (A0)
        MOVE.B  #$50, (A0)      ;Tx and Rx clock from baud gen

        MOVE.B  #12, (A0)
        MOVE.B  #102, (A0)      ;baud rate divider, low byte

        MOVE.B  #13, (A0)
        MOVE.B  #0, (A0)        ;baud rate divider, high byte

        MOVE.B  #14, (A0)
        MOVE.B  #$03, (A0)      ;enable baud rate generator
   ;all remaining WR registers affect synchronous operation only
```

Program 11.5

```
   ;transmit routine
   ;enter here, with outgoing data byte in D0
trans:  BTST.B  #2, CTRL_A      ;Tx buffer empty?
        BEQ     trans           ;if not, keep checking
        MOVE.B  D0, DATA_A      ;if so, send byte
        RTS                     ;and return

   ;receive routine
   ;incoming data byte returned in D0
recv:   BTST.B  #0, CTRL_A      ;Rx character available?
        BEQ     recv            ;if not, keep checking
        MOVE.B  DATA_A, D0      ;if so, get byte
        RTS                     ;and return
```

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

interrupt-driven routine. The 8530 would be happy to oblige by making interrupts on any specified condition; it can even acknowledge with an 8-bit vector of your choosing, if you connect the interrupt acknowledgment lines (IEI, IEO, INTACK). These options are, of course, specified through initialization control bytes.

Parallel I/O chips

We showed an example of these versatile parallel port chips (often combined with one or more timers) in our microprocessor example. The 8536 we used there is a companion to the 8530 serial port chip, and it uses a similar processor interface and setup protocol. Good parallel port chips let you individually program the direction and mode (latching, open-drain, inverting, etc.) of each bit. The data-transfer protocol can also be programmed; for example, with the 8536 you can enable vectored interrupts on any *pattern* of input bits. You can also select one of four *handshake* modes, as we'll describe shortly.

PIO chips, like all LSI peripheral chips, are constructed with NMOS or CMOS technology, with the latter being favored in new designs. Outputs can generally sink a few milliamps, but NMOS outputs, unlike CMOS, can source only a fraction of a milliamp. As a result, they are usually used in conjunction with power driver chips to drive loads requiring significant amounts of current. Don't try to turn on a relay directly with a PIO output (but see our comments on MSI ports under "Latches and *D* registers," above).

Figure 11.27 suggests the sort of circuit you might use, in this case to refresh a 6-digit display. You would, of course, have to write software to repetitively output the successive digit values, along with a "walking bit" in the A port, taking care that interrupts are disabled during display to prevent flicker. An easier way to handle multidigit LED displays in microprocessor systems is to use something like the Siemens "intelligent display" series of memory-mapped display sticks, which conveniently look like memory to the

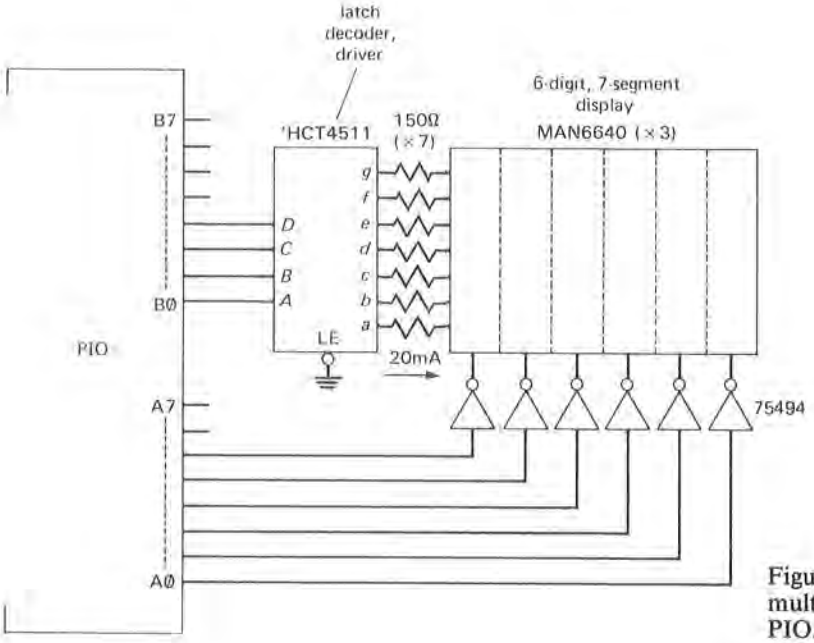


Figure 11.27. Driving a multiplexed display with a PIO.

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

CPU (Fig. 9.24); since they latch their displayed values, you need to write to them only when you want to change the display.

Handshaking

The business of handshaking deserves a bit more explanation. Suppose you are an external device sending bytes of data to a processor via its PIO port. You want to know when the PIO is ready to accept the next byte, i.e., when the previous byte has been picked up by the processor. The natural way to handle this is to have a "ready for data" (RFD) output from the PIO, which is disasserted by the PIO when you give it a byte, and reasserted after that byte has been picked up by the CPU. In other words, you can strobe your data at any time RFD is asserted.

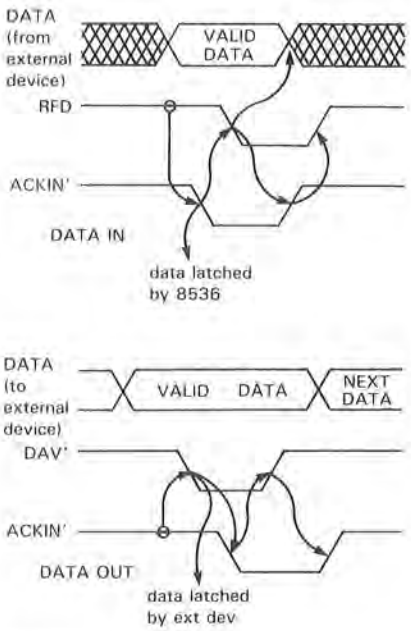


Figure 11.28. Handshaking.

Figure 11.28 shows how you use this "interlocked handshake," which is actually one of four possible handshaking modes provided by the 8536. For data IN, the external source may assert data, but waits for RFD true before asserting the strobe ACKIN′ (that's Zilog's name for this pin!). It releases the strobe (and may disassert data) when it sees RFD disasserted, then waits for RFD again. For data OUT the procedure is similar, with the PIO asserting valid data, then asserting DAV′ (data available). The external device latches the data, then asserts the acknowledgment ACKIN′. This causes the PIO to disassert both the data and DAV′. The latter is a signal for the external device to disassert ACKIN′, beginning a new cycle. Note that the handshake is fully interlocked, in that each side in the transaction waits for the other to complete each step. This fully interlocked data exchange guarantees that no data are lost. It's sometimes nice to simplify things, however. The 8536 therefore has a "pulsed handshake" mode in which the ACKIN′ signal doesn't have to wait for permission to disassert; instead, it is a pulse of at least 250ns duration, asserted when RFD or DAV′ is true, as above.

As you might expect, the handshaking modes are selected by sending those all-important control bytes during initialization. When handshaking is selected, some or all of the four port-C lines are used for ACKIN′, DAV′, and RFD. If you don't tell the chip to use handshaking, you can use the port-C lines instead for ordinary I/O bits, just like ports A and B.

Warning: A general disease of many LSI peripheral chips, particularly those with a clock input, is that they can have a surprisingly large latency between allowed accesses. Our experience with the 8530 is instructive: It was necessary to put some NOPs between successive WRITEs when using a 68000 running at 10MHz, because the speed with which the CPU could send successive bytes (0.8μs) was faster than the 1.7μs minimum "valid access recovery time" of the 8530. Watch out, also, for peculiar processor-interface timing

Petitioner Intel Corp., Ex. 1037
IPR2023-00783

requirements. Recall that in our microprocessor circuit, for example, the Zilog peripherals had an unusual requirement of 80ns setup time from address to leading edge of the *read* strobe, forcing us to add circuitry to generate a delayed RD′ pulse. These idiosyncrasies, which you never see in plain MSI digital logic, are the result of the chip's clocked state-machine logic, the relatively slow NMOS speeds, or both.

The only safe cure for these LSI pathologies is careful design, which implies studying the data sheets carefully and completely, talking to other users, and exhaustive testing. If you notice an anomaly while testing a prototype, don't assume that it will go away in the more carefully constructed final version. In fact, these are the very clues you are seeking. Often you can test a circuit more thoroughly by artificially varying the clock rate and supply voltage, and perhaps also the temperature (use a heat gun), in the prototype.

Microprocessor-bus A/D and D/A converters

With the increasing use of microprocessor-controlled (as opposed to stand-alone) data acquisition, newer A/D and D/A converters have been designed with data buses in mind. "Microprocessor-compatible" D/A converters, for example, have byte-wide input data paths and double buffering so that you can latch a 12-bit quantity from an 8-bit bus in two write cycles; the double buffering ensures that the entire 12-bit quantity is presented to the converter simultaneously, to minimize output glitches. Likewise, microprocessor-compatible A/D converters have three-state output drivers, organized in 8-bit-wide groups. Both A/D and D/A converters tend to have familiar bus-oriented signals such as RD′, WR′, and CS′. These converters can generally be connected directly to the microprocessor bus, with only a small amount of "glue" needed for address decoding. You don't need to fool around with the usual interface paraphernalia such as external latches and three-state drivers. Take a look back at our microprocessor circuit, which has both ADCs and DACs, to see how easy it is.

As examples, the AD7537 from Analog Devices is a dual 12-bit DAC with 8-bit loading (i.e., each 12-bit quantity is loaded as 8 + 4, in two bus cycles), with provision for simultaneous updating of the two double-buffered 12-bit channels (Fig. 11.29); the companion AD7547 has a 12-bit-wide input, for 16-bit buses. The AD7572 from the same manufacturer is a no-glue 12-bit ADC with choice of byte-wide or 12-bit-wide parallel output via three-state drivers; it can thus be used efficiently with microprocessor buses of 8-bit or 16-bit width. Some bus-compatible converters even allow left- or right-justification when the word is smaller than the bus. When choosing a bus-compatible converter, watch out for devices with sluggish processor-interface timing (which has nothing to do with the *conversion* speed), which can force you to insert wait states, delayed strobes, etc. For example, the AD558, an otherwise admirable self-contained 8-bit ADC, has a 200ns minimum data setup to trailing edge of WR′, requiring a wait state with the 68008.

In any ADC microprocessor interface with resolution of 12 bits or more, consider isolating the ADC chip outputs from the system bus with buffers (or even opto-isolators); otherwise the digital current transients and microprocessor noise are likely to degrade the resolution. For the highest resolutions (16 bits or more), it may be best to put the converter outside the box containing the digital electronics. As an example of what can happen, we offer our experience with a commercial 16-bit ADC board designed for the IBM PC: The converter module is on the PC board, inside the computer. We were skeptical that it would attain a full 16 bits of resolution, so before buying one we asked what

Petitioner Intel Corp., Ex. 1037
IPR2023-00783