PART 2

SYSTEMS DESIGN AND DESIGN METHODS

This Part starts with Chapter 6 examining a structured design approach and its application to CMOS system design. This is followed by a discussion of CMOS chip implementation options ranging from Field Programmable Gate Arrays (FPGAs) to full custom layout, which illustrates the trade-offs between implementation, design complexity, and time to market. The chapter then discusses a variety of CMOS design automation options and the various design tools used for CMOS design.

Chapter 7 deals with the important problem of testing CMOS circuits by introducing the reader to the test process, followed by an explanation of test nomenclature. Various methods of designing a testable CMOS circuit are then treated. This treatment centers on a structured approach to testing.

Chapter 8 provides an extensive set of subsystem examples starting with coverage of datapaths. Adders are treated at great depth. The chapter continues with a treatment of the design of memories and concludes by examining various techniques for implementing CMOS control logic.

Micron Ex. 1037, p. 419
Micron v. YMTC
IPR2025-00119

CMOS DESIGN METHODS

6

6.1 Introduction

In Chapter 1 we found that the design description for an integrated circuit may be described in terms of three domains, namely: (1) the behavioral domain, (2) the structural domain, and (3) the physical domain. In each of these domains there are a number of design options that may be selected to solve a particular problem. For instance, at the behavioral level, the freedom to choose, say, a sequential or a parallel algorithm is available. In the structural domain, the decision about which particular logic family, clocking strategy, or circuit style to use is initially unbound. At the physical level, how the circuit is implemented in terms of chips, boards, and cabinets also provides many options to the designer. These domains may be hierarchically divided into levels of design abstraction. Classically these have included the following:

- Architectural or functional level.
- Register-transfer level (RTL).
- Logic level.
- Circuit level.

The relationship between description domains and levels of design abstraction are elegantly shown by the Y-chart[1,2] in Fig. 6.1, which was introduced in Chapter 1. In this diagram, the three radial lines represent the three description domains, namely the behavioral, structural, and physical

Micron Ex. 1037, p. 421
Micron v. YMTC
IPR2025-00119

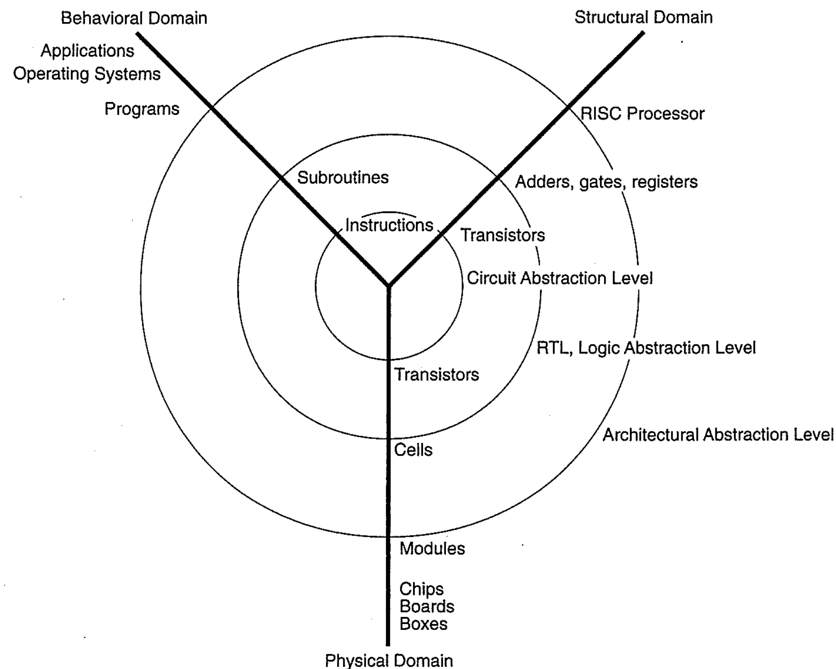


FIGURE 6.1 Y-chart showing description domains and levels of design abstraction

domains. Along each line are enumerated types of objects in that domain. Circles represent levels of similar design abstraction: the architectural, logic, and circuit levels. The particular abstraction levels and design objects may differ slightly, depending on the design method.

In this chapter we will examine the means by which we transform a description in one domain into a description in another domain. We begin by discussing some of the guiding principles that apply to most engineering projects. Then the various design strategies available to the CMOS IC designer are surveyed, ranging from very fast prototyping or small-volume approaches to the more labor-intensive custom design approaches. The CAD tools necessary to achieve the design strategies are then summarized. Finally, we examine the economics of design, which can guide us to the right selection of an implementation strategy.

6.2 Design Strategies

6.2.1 Introduction

The economic viability of an IC is in large part affected by the productivity that can be brought to bear on the design. This in turn depends on the efficiency with which the design may be converted from concept to architecture, to logic and memory, to circuit and hence to a physical layout. A good VLSI design system should provide for consistent descriptions in all three description domains

Micron Ex. 1037, p. 422
Micron v. YMTC
IPR2025-00119

(behavioral, structural, and physical) and at all relevant levels of abstraction (architecture, RTL, logic, circuit). The means by which this is accomplished may be measured in various terms that differ in importance based on the application. These design parameters may be summarized in terms of

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

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

Given that the process of designing a system on silicon is complicated, the role of good VLSI-design aids is to reduce this complexity, increase productivity, and assure the designer of a working product. A good method of simplifying the approach to a design is by the use of constraints and abstractions. By using constraints the tool designer has some hope of automating procedures and taking a lot of the "legwork" out of a design. By using abstractions, the designer can collapse details and arrive at a simpler concept with which to deal.

In this chapter we will examine design methodologies that allow a variation in the freedom available in the design strategy. The choice, assuming all styles are equally available, should be entirely economic. According to function, suitable design methods are selected. It may be found that due to inefficiencies in layout, some styles will not be capable of implementing the function. Following these steps, the required die cost is estimated and the quickest means of achieving that die should be chosen. We will focus on structured approaches to design since they offer the best prospects of dealing with large and diverse VLSI problems of the present and future.

6.2.2 Structured Design Strategies

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

Micron Ex. 1037, p. 423
Micron v. YMTC
IPR2025-00119

consideration is a small chip designed by a single designer or a large system designed by a team of designers, the basic principles of structured design will improve the prospects of success. In the following sections some of the classical techniques for reducing the complexity of IC design will be summarized.[3,4]

6.2.3 Hierarchy

The use of hierarchy, or "divide and conquer," involves dividing a module into submodules and then repeating this operation on the submodules until the complexity of the submodules is at an appropriately comprehensible level of detail. This parallels the software case where large programs are split into smaller and smaller sections until simple subroutines, with well-defined functions and interfaces, can be written. As we have seen, a design may be expressed in terms of three domains. We can employ a "parallel hierarchy" in each domain to document the design. For instance, an adder may have a subroutine that models the behavior, a gate-connection diagram that specifies the circuit structure, and a piece of layout that specifies the physical nature of the adder. Composing the adder into other structures can proceed in parallel for all three domains, with domain-to-domain comparisons ensuring that the representations are consistent.

At a system level, the use of hierarchy allows one to specify single-designer projects, at which level the schedule is proportional to the number of available personnel.

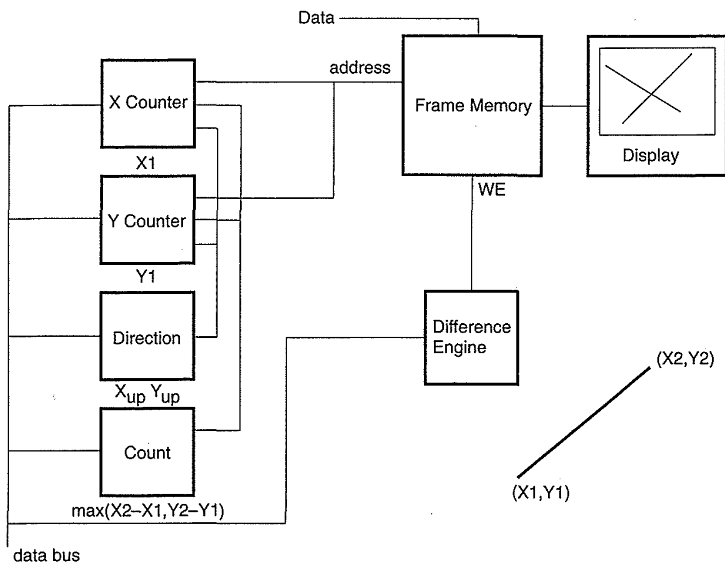
Example

To illustrate the principle of hierarchy consider the top-level diagram of a raster-graphics vector generator that includes an 8-bit difference engine, shown in Fig. 6.2(a). This engine may be used for a variety of graphics algorithms, including line drawing and linear shading. Operation in the case of drawing a line on a raster display consists of loading the *X*, *Y*, count, and direction registers with the initial (*X*,*Y*) point, length of the line, and up/down-count control data for the *X* and *Y* counters. The difference engine block is loaded with three values—*A*, *B*, and *C*—which are derived from the parameters of the line to be drawn.

A diagram of the difference engine is shown in Fig. 6.2(b). It consists of an *A*, *B*, and *C* register, an adder, and two multiplexers or muxes. The multiplexers, registers, and adder may be decomposed into 1-bit units. The hierarchy is stopped at the level where modules are defined in terms of simulation models and physical layouts. For instance, the adder, multiplexer, and register might be standard cells. Similar decompositions could be completed for the other modules in Fig. 6.2(a).

The hierarchy defined above is a structural hierarchy that reflects functionality, such as the adding, multiplexing, or storing state. An alternative hierarchy for the difference engine is shown in Fig. 6.3, where 8 identical "bit-slices" have been built. Each bit slice has one element of the engine

Micron Ex. 1037, p. 424
Micron v. YMTC
IPR2025-00119



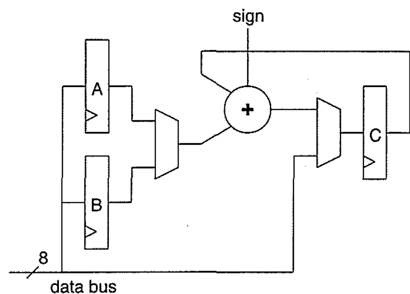
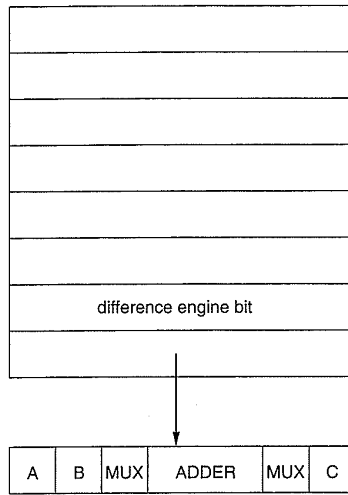


FIGURE 6.2 A difference engine: (a) system diagram; (b) implementation

shown in Fig. 6.2(b). This is known as a "physical hierarchy" because it might be the decomposition used to build an n-bit difference engine layout. Thus there are at least two "disjoint" hierarchies describing the same structure. The hierarchies "join" at the difference-engine level. Generally, it is good practice to maintain identical hierarchies between the function, structure, and physical aspects of a design because this allows consistent checks between description domains from the lowest level of the hierarchy to the very top levels. Frequently, if the physical hierarchy is designed first without a structural or functional hierarchy, it will be found that the resulting hierarchy is cumbersome. On the other hand structural hierarchies may be defined that do not map well to physical constraints. For instance, consider the floorplan shown in Fig. 6.4 where module *A* has to fit within a certain area constraint. Module *B* has space for some of the contents of Module *A* but, due to the structural hierarchy, the floorplan in Fig. 6.4(a) results. Usually, after a few iterations the physical and structural hierarchies may be reconciled (Fig. 6.4b). Many times the issue is moot because an automatic layout system is able to take the structural hierarchy and create a layout that meets both timing and area requirements.

Micron Ex. 1037, p. 425
Micron v. YMTC
IPR2025-00119



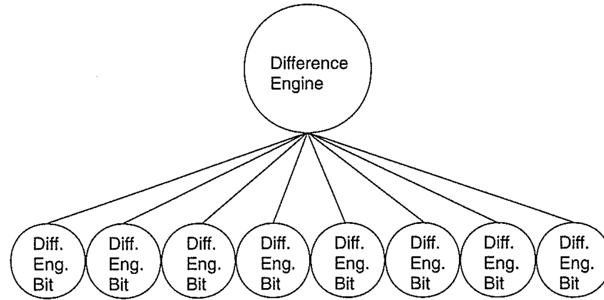
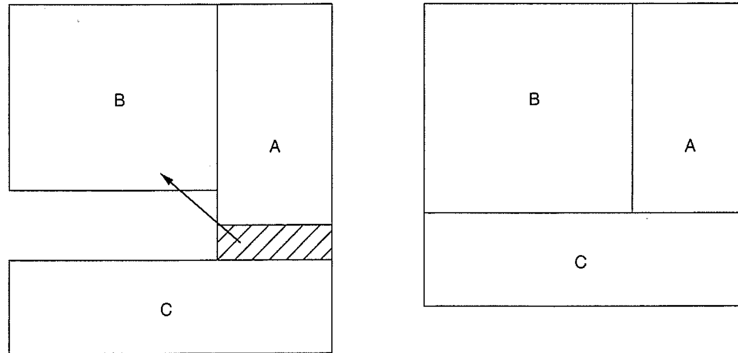
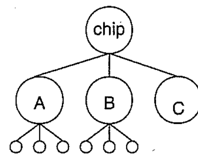


FIGURE 6.3 A physical hierarchy for the difference engine





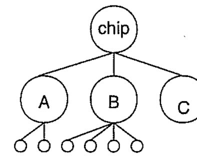


FIGURE 6.4 Repartitioning the structural hierarchy to suit the physical hierarchy

Micron Ex. 1037, p. 426
Micron v. YMTC
IPR2025-00119

6.2.4 Regularity

Hierarchy involves dividing a system into a set of submodules. However, hierarchy alone does not necessarily solve the complexity problem. For instance, we could repeatedly divide the hierarchy of a design into different submodules but still end up with a large number of different submodules. With regularity as a guide, the designer attempts to divide the hierarchy into a set of similar building blocks. The use of iteration to form arrays of identical cells is an illustration of the use of regularity in an IC design. However, extended use may be made of regular structures to simplify the design process. For instance, if the designer were constructing a "datapath," the interface between modules (power, ground, clocks, busses) might be common but the internal details of modules may differ according to function. Regularity can exist at all levels of the design hierarchy. At the circuit level, uniformly sized transistors might be used rather than manually optimizing of each device. At the logic-module level, identical gate structures might be employed. At higher levels, one might construct architectures that use a number of identical processor structures. By using regularity in the ways mentioned, a design may be judged correct by construction. Methods for formally proving the correctness of a design may also be aided by regularity.

Regularity allows an improvement in productivity by reusing specific designs in a number of places, thereby reducing the number of different designs that need to be completed.

Example

Continuing the example of the difference engine in Fig. 6.2(b), the multiplexer, adder, and register modules may be defined in terms of identical CMOS inverters and tristate inverters as illustrated in Fig. 6.5. The counters shown in Fig. 6.2(a) might use the same adder, register, and multiplexer used in the difference engine. For every different module that is used (no matter what level), a variety of design checks have to be performed—functional verification, timing verification, layout-connectivity verification, etc. By identifying common operations at a high level, regularization can reduce the number of different modules that need to be designed and verified (i.e., counting = adding 1 = adding). This principle applies at all levels of hierarchy.

6.2.5 Modularity

The tenet of modularity adds to hierarchy and regularity the condition that submodules have well-defined functions and interfaces. If modules are "well-formed," the interaction with other modules may be well-characterized. The notion of "well-formed" may differ from situation to situation, but a good starting point is the criteria placed on a "well-formed" software subroutine. First of all, a well-defined interface is required. In the case of software this is an argument list with typed variables. In the IC case this

Micron Ex. 1037, p. 427
Micron v. YMTC
IPR2025-00119

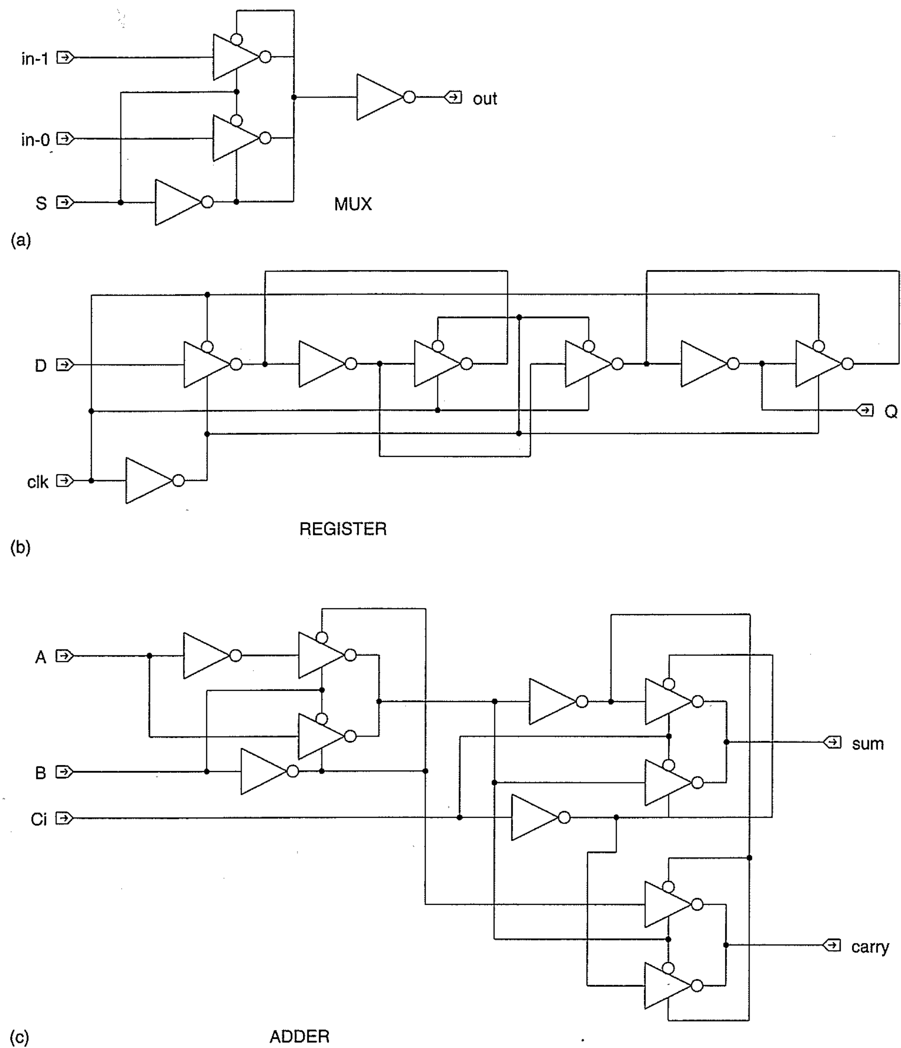


FIGURE 6.5 Regularity at the circuit level by using inverters and tristate buffers: (a) multiplexer; (b) register; (c) adder

corresponds to a well-defined behavioral, structural, and physical interface that indicates the position, name, layer type, size, and signal type of external interconnections, along with logic function and electrical characteristics. For instance, connection points may indicate the power and ground, inputs and outputs to a module. The function must also be defined in an unambiguous manner. Modularity helps the designer to clarify and document an approach to a problem, and also allows a design system to more easily check the attributes of a module as it is constructed. The ability to divide a task into a set of well-defined modules also aids in a team design where each of a number of designers has a portion of a complete chip to design.

In structured programming, proponents advise the use of only three basic constructs. These are concatenation, iteration, and conditional selec-

Micron Ex. 1037, p. 428
Micron v. YMTC
IPR2025-00119

tion. In the IC design world these constructs have parallels. For instance, concatenation is mirrored by cell abutment, where IC cells (in the physical domain) are connected by placing them adjacent to each other and intercell connections are formed on the common boundary. Iteration is handled in the IC case by one- and two-dimensional arrays of identical cells, typified by a memory. The use of conditional selection is typified in a programmable logic array (PLA), the function of which is determined by the location of transistors in an array. When combined with the ability to parametize designs, these three programming notions can greatly aid the designer in modularizing a design. Of course in a Hardware Description Language (HDL) these constructs are used directly.

At a system level, the correct decisions regarding modularity allows one to break up a system with the confidence that when the parts are combined, the whole system will function as specified.

Example

A good example of the use (or ill use) of modularity is the use of transmission gates as inputs to logic modules (especially those using ratioed logic). Normally in CMOS circuits the inputs to logic blocks connect to the gates of MOS transistors. In these cases, the internal behavior of the modules is entirely determined by the arrival time and shape of the input waveform. Consider the case where the inputs are connected to transmission gates that are in turn connected to ratioed circuitry (for instance, consider the situation when the 2-input multiplexer shown in Fig. 6.6(a) is used for the multiplexer blocks in the difference engine example). The internal signal condition is now determined by the source impedance in addition to the input timing. A module-to-module cross-check has to occur to ensure that the driving circuit can adequately drive the mux. This is an example of a poorly modularized circuit. The fix is to use the tristate-inverter-based mux shown in Fig. 6.5(a) or the buffered mux shown in Fig. 6.6(b).

Modules can also be poorly modularized on a temporal basis. Consider a module for the difference engine that uses dynamic CMOS logic but fails to latch or register the inputs. Because external inputs might arrive at various times with respect to the clock, erroneous results might occur unless the timing of each input is individually checked. A modular approach to clocking where all module inputs are registered on entering the module and all outputs are the outputs of registers is the first step in ensuring module-to-module timing consistency.

6.2.6 Locality

By defining well-characterized interfaces for a module, we are effectively stating that the other internals of the module are unimportant to any exterior

Micron Ex. 1037, p. 429
Micron v. YMTC
IPR2025-00119

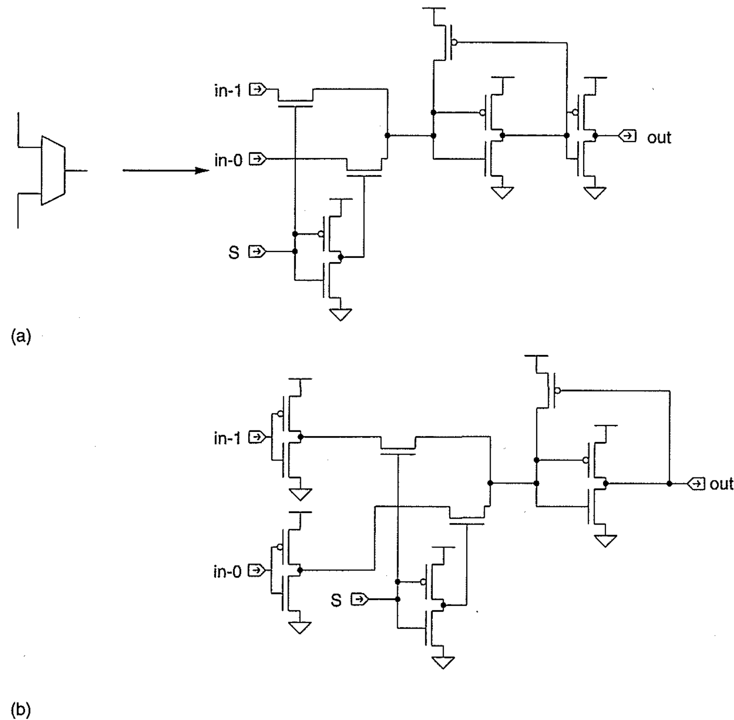


FIGURE 6.6 An example of poor modularity: (a) mux with transmission-gate inputs; (b) a solution—a buffered mux

interface. In this way we are performing a form of "information hiding" that reduces the apparent complexity of that module. In the software world this is paralleled by the reduction of global variables to a minimum (hopefully to zero).

Example:

Increasingly, locality has come to mean "time locality"; that is, modules see a common clock, and hence synchronous-timing methods apply. The first way of ensuring time locality is to pay attention to the clock generation and distribution network (see Chapter 4). Having done this, critical paths, if possible, should be kept within module boundaries. Any global module-to-module signal should have the entire clock cycle to traverse the chip. Repeated chip crossings of critical signals will rapidly lead to inferior timing characteristics. Many times in modern designs, logic is replicated to alleviate cross-chip crossings.

Modules can also be located to minimize the "global wiring" that may be necessary to connect a number of modules in a system. A common imperative in design systems today that applies for both gate-arrays and custom design is use "wires first, then modules"—rather than the more common "place modules, then route them together."

Micron Ex. 1037, p. 430
Micron v. YMTC
IPR2025-00119

TABLE 6.1 Structured Software and VLSI Hardware Design

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

6.2.7 Summary

There are strong parallels between the methods of design for software systems and for hardware systems. Table 6.1 summarizes some of these parallels for the principles outlined above.

As stated previously the use of HDLs to describe hardware systems in essence merges these two disciplines and the software methods above are used to define hardware. At some level the hardware aspects become relevant as a physical chip is the end product.

6.3 CMOS Chip Design Options

In this section we will examine a range of design options that may be used to implement a CMOS system design. These are arranged in order of "increased design investment," which loosely relates to the time it takes to design the device. The sequence is also somewhat in order of complexity of device that may be implemented.

6.3.1 Programmable Logic

As the investment made in any chip design is significant, designers search for ways in which to amortize the design effort over a large number of devices. This might result from a huge single market for one device or, more probably, multiple smaller markets for a more adaptable device. The larger the unit volume for a part the lower its cost will be to the end user.

Programmability is one way to achieve a wider use for a particular part. This is epitomized by the microprocessor. Often, though, the cost or speed of

Micron Ex. 1037, p. 431
Micron v. YMTC
IPR2025-00119

a microprocessor may not meet system goals and an alternative solution is required. In CMOS, one may divide this spectrum of programmable devices into three areas:

- Chips with programmable logic structures.
- Chips with programmable interconnect.
- Chips with reprogrammable gate arrays.

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

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

6.3.2 Programmable Logic Structures

The first broad class of programmable CMOS devices are represented by the programmable logic devices referred to as PALs® (Programmable Array Logic, ®Advanced Micro Devices, Inc.) or PLDs (Programmable Logic Devices).[5,6] Generally, these devices are implemented as AND-OR plane devices as shown in Fig. 6.7. In the design shown a number of inputs feed vertical wires, which are selectively connected to an AND-OR gate. Each AND-OR gate has a variable number of product terms that feed the gate. This gate in turn feeds an I/O cell, which allows registering of the AND-OR signal and the feedback of the registered result into the AND-OR plane. PAL devices come in a large range of sizes with a variable number of inputs, outputs, product terms, and I/O-cell complexity. The 22V10 is an industry-standard device with the following characteristics:

12 inputs
10 I/Os
#product terms 9 10 12 14 16 14 12 10 8
24 pins

The I/O structure for a 22V10 is shown in the inset in Fig. 6.7. It consists of a register, an output 4:1 mux, a tristate buffer, and a 2:1 input mux. The tristate buffer is used to enable the output. Alternatively, the pin may be used as an input to the array. The 4:1 mux routes the true or complemented ver-

Micron Ex. 1037, p. 432
Micron v. YMTC
IPR2025-00119

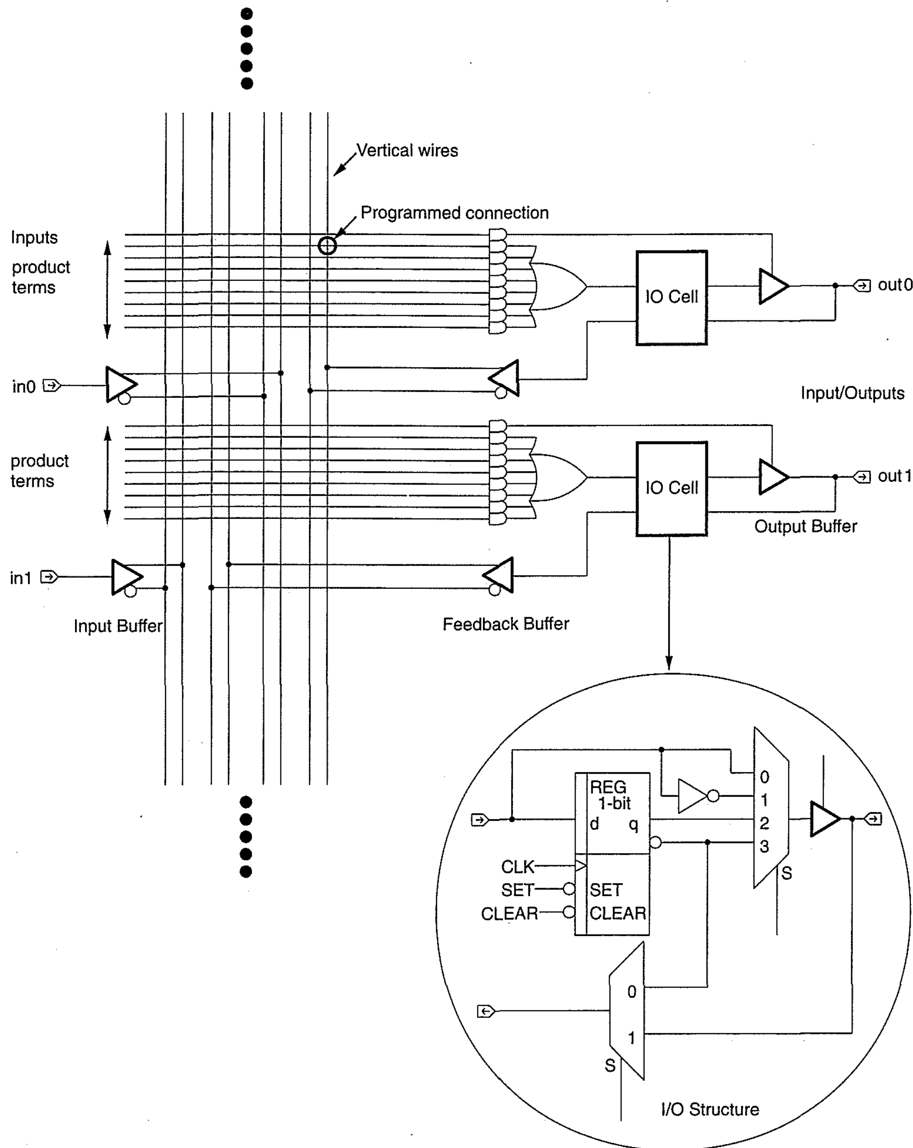


FIGURE 6.7 A typical PAL architecture

sion of the product term or register output to the output. The 2:1 input mux may also select the register output. The register is provided with a global synchronous preset and asynchronous reset.

Typical speeds for a 22V10 in high-speed CMOS are:

- CLK to output—8 *ns*.
- Input to combinational output—15 *ns*.

Typical toggle frequencies with feedback are around 40 MHz.

Micron Ex. 1037, p. 433
Micron v. YMTC
IPR2025-00119

The programming of PALs is done in three main ways:

- Fusible links.
- UV-erasable EPROM.
- EEPROM (E^2ROM)—Electrically Erasable Programmable ROM.

Fusible links use a metal such as platininum silicide or titanium tungsten to form links that are blown when a certain current is exceeded in the fuse. This is normally accomplished by using a higher than normal programming voltage applied to the device. This technology is normally used in conjunction with a bipolar process (as opposed to a CMOS process) where the small devices can readily sink the current needed to blow the fuses. Programming is a one-time operation. As an alternative to current, a laser can be used to cut aluminum fuses in normal CMOS technologies. Frequently this is used in redundant memory techniques where a spare column may be switched in to replace a failing one.

UV-erasable memories typically use a floating gate structure as shown in Fig. 6.8. Here a floating gate is interposed between the regular MOS transistor gate and the channel (see also Chapter 3). To program the cell, a voltage around 13–14 volts is applied to the control gate while the drain of the transistor to be programmed is held at around 12 volts. This results in the floating gate becoming charged negatively. This increases the threshold of the transistor (to around 7 volts), thus rendering it permanently "off" for all normal circuit voltages (maximum 5–6 volts). The process can be reversed by illuminating the gate with UV light.

"Permanently" means at least 10 years at 125°C. At elevated temperatures the storage time will be reduced. Programming may be completed numerous times. The chips are usually housed in glass-lidded packages to allow illumination by UV light.

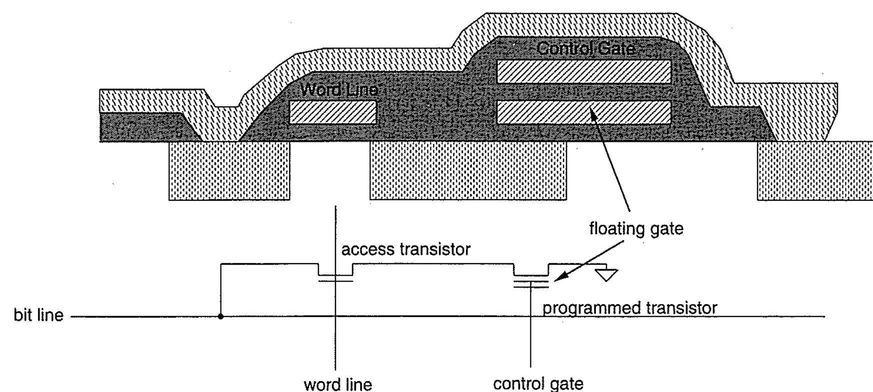


FIGURE 6.8 UV- erasable EPROM structure

Micron Ex. 1037, p. 434
Micron v. YMTC
IPR2025-00119

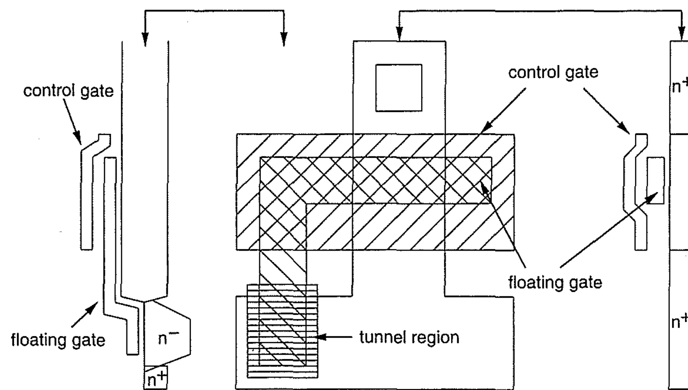


FIGURE 6.9 EEPROM structure (© IEEE 1992)

EEPROM technology allows the electrical programming and erasure of CMOS ROM cells. This type of programming forms the most popular in use today for CMOS and is the one most likely encountered by the IC-system designer in today's foundry processes. A typical structure is shown in Fig. 6.9.[7] Two transistors are typically used in a ROM cell. One is an access transistor, while the other is the programmed transistor. A two-poly sandwich is again used in the programmed transistor with the control gate on the top. A very thin oxide between the floating gate and the drain of the device allows electrons to "tunnel" to or from the floating gate (thus charging the gate oxide) to turn the cell off or on respectively. The series-access transistor allows programming of cells. EEPROM has a testability advantage over fused technologies. Each device can be fully tested before shipment. A range of ROM architectures have been used, including the normal NOR ROM structure[8] and NAND structures.[9]

By way of comparison, in a custom .8 μm CMOS chip a PLA (Programmable Logic Array) of the complexity of the 22V10 (programmed via mask) would be roughly 200μ wide by 500μ tall or .01 mm^2, and would be approximately the same speed or faster in a given technology. On a 100 mm^2 square chip one could fit 5,000 such PLAs (assuming 50% overhead for routing).

6.3.3 Programmable Interconnect

In a PAL the device is programmed by changing the characteristics of the switching element. An alternative would be to program the routing. This has been demonstrated via a number of techniques including Laser Pantography, where a laser lays down paths of metal under computer control. Commercially, programmable routing approaches are represented by products from Actel, QuickLogic, and other companies.

The Actel Field Programmable Gate Arrays[10] are based on an element called a PLICE™ (Programmable Low-Impedance Circuit Element) or anti-

Micron Ex. 1037, p. 435
Micron v. YMTC
IPR2025-00119

fuse. An antifuse is normally high resistance (>100 MΩ). On application of appropriate programming voltages, the antifuse is changed permanently to a low-resistance structure (200–500Ω). The structure of an antifuse is shown in Fig. 6.10(a). It consists of an ONO (oxide-nitride-oxide) layer sandwiched between a polysilicon layer on top and an n+ diffusion on the bottom. The QuickLogic array is based on a structure called a ViaLink®, which consists of a sandwich of material between metal1 and metal2.[11] This is illustrated in Fig. 6.10(b). The "on" resistance of this structure is somewhat lower than that in Fig. 6.10(a).

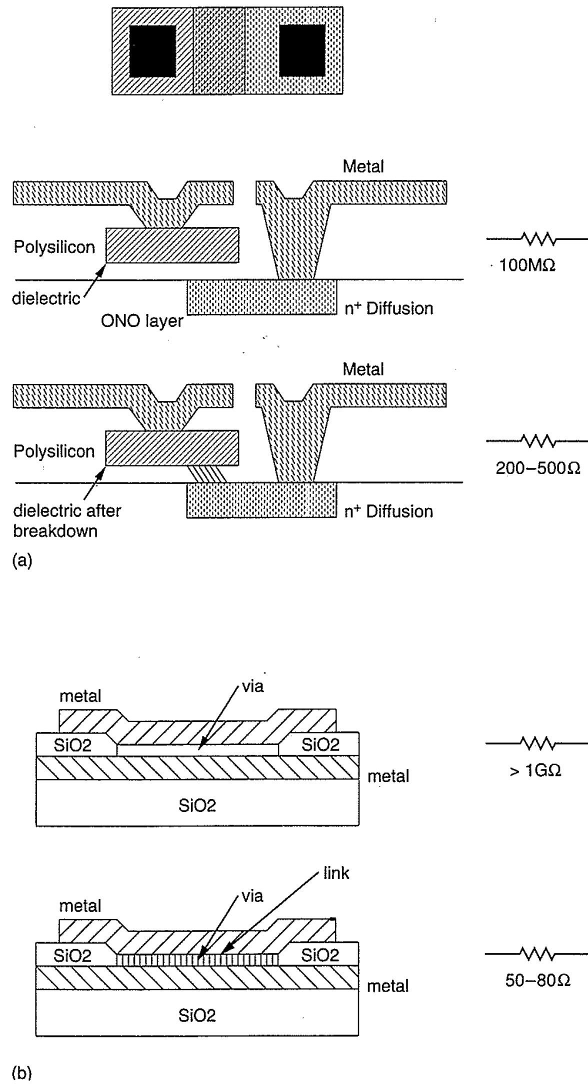


FIGURE 6.10 Programmable interconnect structures: (a) Antifuse®; (b) ViaLink®

Micron Ex. 1037, p. 436
Micron v. YMTC
IPR2025-00119

One chip architecture that uses the antifuse is shown in Fig. 6.11.[12,13,14] Logic elements are arranged in rows separated by horizontal interconnect. Interconnect permanently connected to the logic elements passes vertically. Both horizontal and vertical segments are segmented into a variety of lengths. Segments may be joined by programming antifuses. Certain special signals such as power and a clock line are routed globally to all logic. The logic elements are surrounded by I/O pads and programming and diagnostic logic. Note the similarity to a gate-array (Section 6.3.5).

A more detailed representation of the interconnect scheme is shown in Fig. 6.12. Pass transistors are used to connect wire segments for the purpose of programming. These may be bypassed by antifuses if the links are required permanently. In the figure transistors N_1, N_2, N_3, N_4, N_9, N_{10}, N_{11}, and N_{12} are the column-access transistors, while transistors N_5, N_6, N_7, and N_8 are the row-access transistors. These are used during programming or may be used for diagnostic purposes to check the state of any signal. In the example shown, the antifuse at the conjunction of N_2's column and N_8's row has been programmed (denoted by a solid dot). This connects the signal in logic module A to the segments that are bolded in the diagram. In addition, the bypass antifuse on N_8 has been programmed, thereby extending the horizontal segment to the next set of logic cells. To program the antifuse at N_2-N_8, all pass transistors in series with N_2 are turned on and the top end is connected to the programming voltage. In addition, all transistors in series with N_8 are turned on and the end connected to the ground supply. When the programming sequence is applied, the antifuse so selected is "blown." Similar addressing techniques allow for the sampling of signals for testing or debugging. The sequencing of the antifuse blowing is carefully determined to ensure that all fuses can be blown.

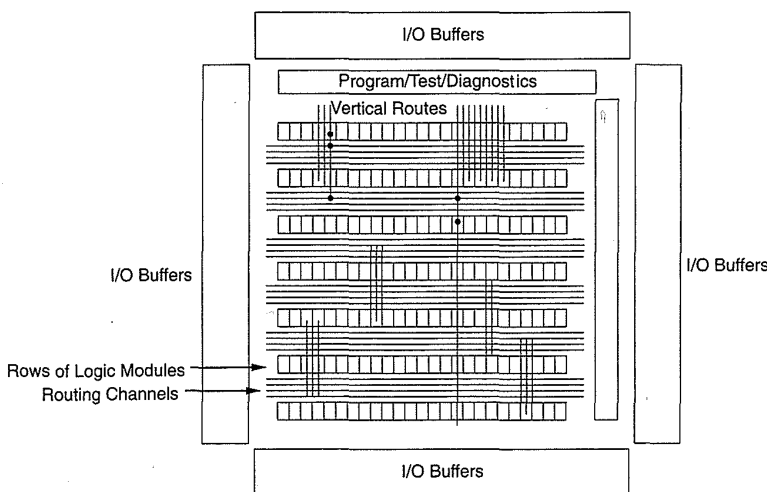


FIGURE 6.11 Actel FPGA chip architecture

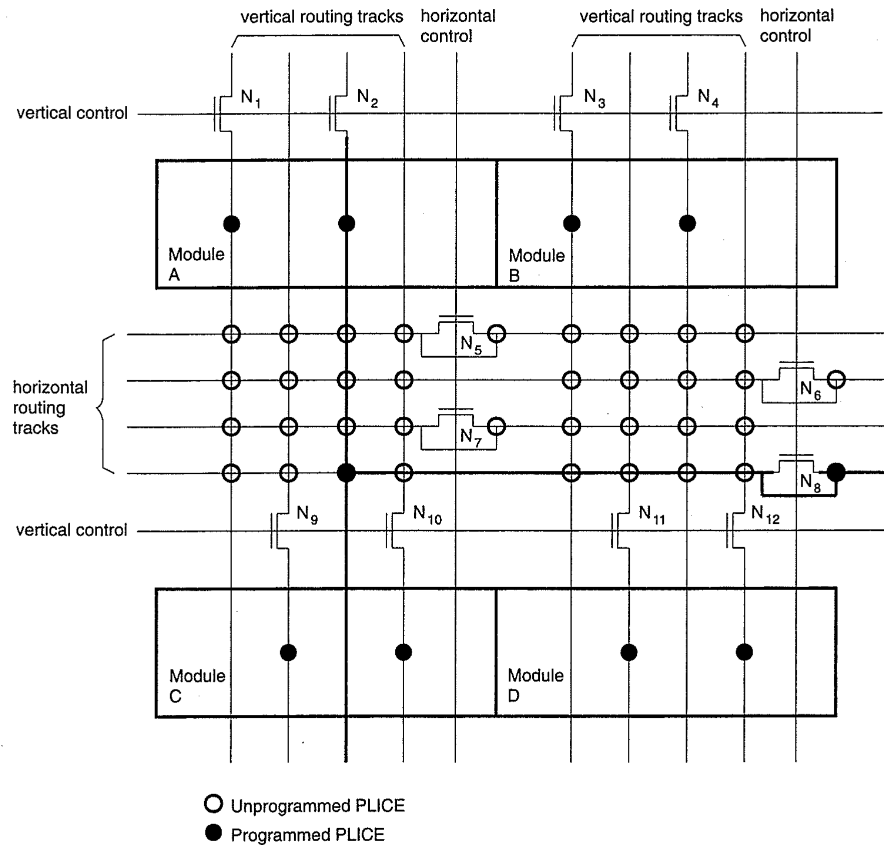


FIGURE 6.12 Actel interconnect example

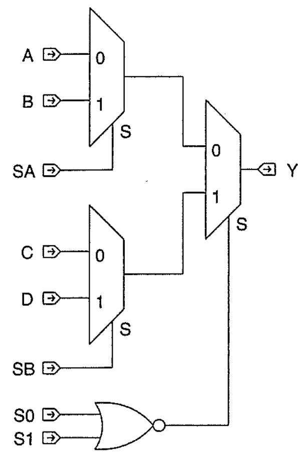


FIGURE 6.13 Actel logic cell

The structure of the Actel logic element is shown in Fig. 6.13. It consists of three 2-input muxes and a NOR gate. This structure can implement all 2- and 3-input logic functions and some 4-input functions. A latch may be implemented with one logic element, while a register requires two elements. The QuickLogic cell is shown in Fig. 6.14. In addition to the structure shown in Fig. 6.13, it includes a resettable register and numerous logic gates. An interesting trade-off in these types of arrays is the granularity of the logic cell versus the amount of routing.

The Actel programmable I/O pad is shown in Fig. 6.15. Two antifuses allow the configuration to operate as an input pad, output pad, or bidirectional pad. If the *ENABLE* pin is not programmed, then the pad is bidirectional. If the *ENABLE* antifuse to V_{DD} is blown, the pad is an output, whereas if the V_{SS} antifuse is blown, the pad is an input. The isolation devices isolate the pad if necessary during programming and testing. (A highly desirable feature of the Actel architecture is the ability to observe any node in the chip using the series pass-transistors that are used for programming.)

Micron Ex. 1037, p. 438
Micron v. YMTC
IPR2025-00119

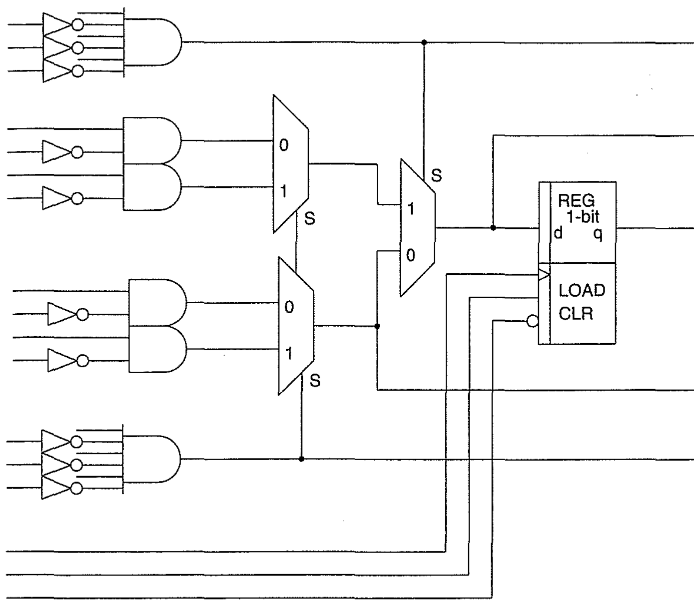


FIGURE 6.14 QuickLogic logic cell

At the time of writing, these arrays could implement 550 logic modules and 70 I/O modules. The speed of a particular circuit depends on the logic element speed and the delay through antifuse elements in any routing. A single logic module exhibits a delay from 7 *ns* to 14 *ns* (5V and 25°C) depending on fan-out in a 2 μm technology. Long route delays through many antifuses can range from 15 *ns* to 35 *ns*. With smaller technologies the logic module delays would decrease while the routing delays might decrease

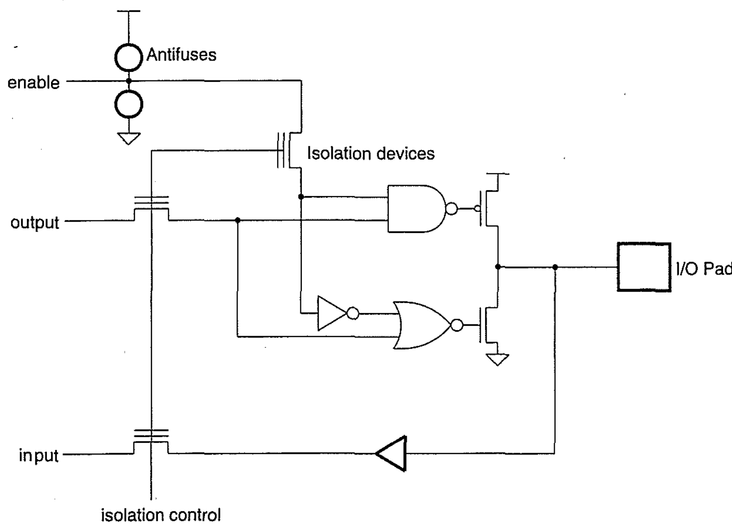


FIGURE 6.15 Actel I/O pad

Micron Ex. 1037, p. 439
Micron v. YMTC
IPR2025-00119

somewhat. More drastic reductions in the routing delays would come with lower "on-resistance" antifuses.

If a 32-bit adder were implemented in an Actel array, 160 logic modules would be needed, and it would add in approximately 65 *ns*. Thus roughly 3.5 32-bit adders would fit in a single FPGA chip. Currently in a 1μ custom chip 7 mm on a side (6 mm-by-6 mm active area), we could fit 1300 adders (each 560μ by 50μ) running twice as fast, or 600 adders running ten times as fast. Bear in mind that if we only wanted two 32-bit adders and a bit of logic running at 10 MHz, we could get in in an afternoon and for about $5–10 if we used an FPGA; compared with 6 months and $200,000 that an application-specific chip would require.

6.3.4 Reprogrammable Gate Arrays

A further class of programmable device is the programmable (or reprogrammable) gate array. These may be further categorized into ad-hoc and structured arrays.

6.3.4.1 The XILINX Programmable Gate Array

An example of an ad-hoc array is a set of products from the XILINX company.[15] The architecture of the XC3000 series is depicted in Fig. 6.16. An array of Configurable Logic Blocks (CLBs) is embedded within a set of horizontal and vertical channels that contain routing that can be personalized to interconnect CLBs. The configuration of the interconnect is achieved by turning on n-channel pass transistors. The state that determines a given interconnect pattern is held in static RAM cells distributed across the chip close

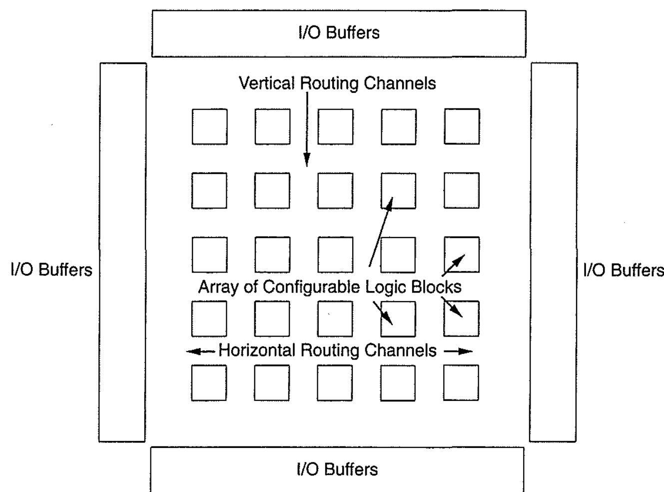


FIGURE 6.16 XILINX FPGA architecture

to the controlled elements. The CLBs and routing channels are surrounded by a set of programmable I/Os.

In detail, the structure of a CLB is shown in Fig. 6.17. It consists of two registers, a number of muxes, and a combinatorial function unit. The latter can generate two functions of four variables, any function of five variables, or a selection between two functions of four variables. The function bit and each mux is controlled by a number of RAM state bits. More recent CLBs feature enhanced table lookup function generators which can be used to build logic functions or used as register storage. Inbuilt support for carry chains means that datapaths can be conveniently built (XC4000 series). Each input and output on a CLB has a particular local interconnect pattern (called direct interconnect by XILINX), which allows most local interconnection between adjacent CLBs to take place. At the junction of the horizontal and vertical routing channels (where the general-purpose interconnect runs),

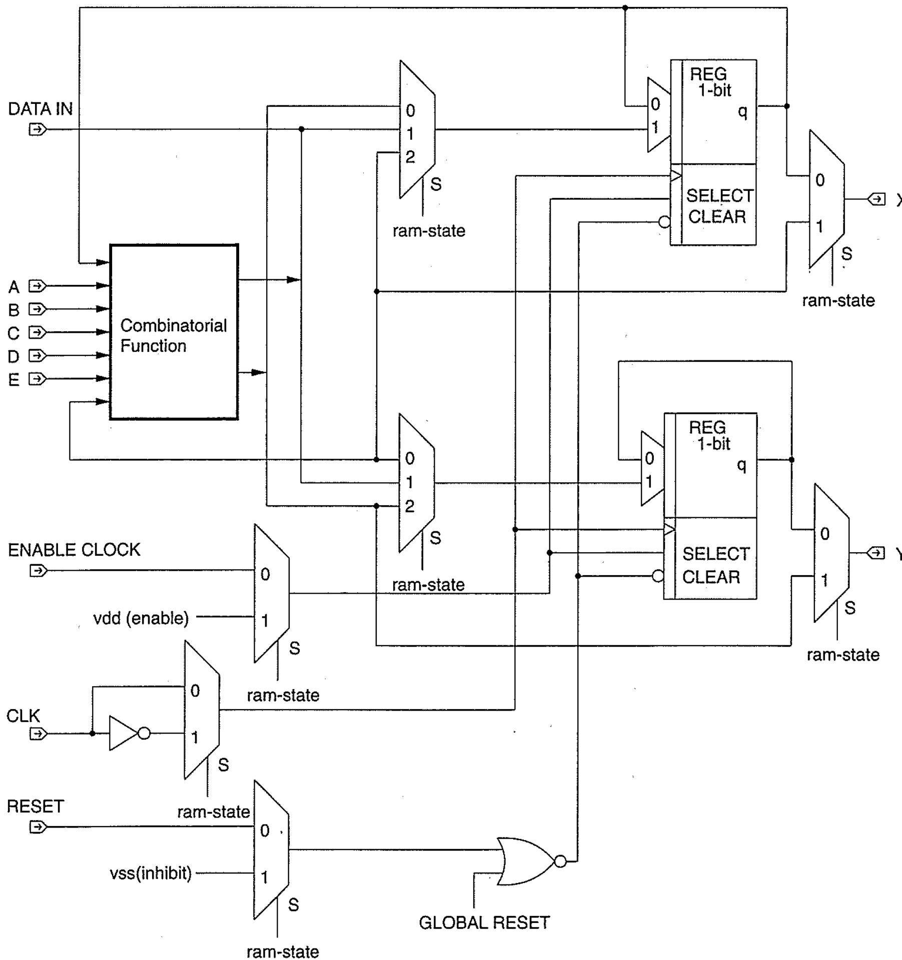


FIGURE 6.17 XILINX Configurable Logic Block® (CLB)

Micron Ex. 1037, p. 441
Micron v. YMTC
IPR2025-00119

programmable switching matrices are employed to redirect routes. Fig. 6.18 shows a typical CLB surrounded by switching matrices. The switching matrices perform crossbar switching of the global interconnect, which runs both vertically and horizontally. Programmable Interconnect Points or PIPs interconnect the global routing to CLBs. Both PIPs and the switching matrices are implemented as n-channel pass gates controlled by 1-bit RAM cells. Extra special long-distance interconnect is used to route important timing signals with low skew.

Assuming one has a board design finished, design proceeds by mapping the logic design to the CLBs and thence to one or more programmable gate-arrays. Software then "places and routes" the CLBs by loading the internal state RAM with the codes needed to program the I/Os, the CLBs, and the routing. The design is then ready to be tested or used.

Currently, the largest array holds ≈500 CLBs and has approximately 100K bits of state RAM (this will increase with time as processes shrink). In common with the Actel approach, timing is dependent on the basic CLB speed and a routing delay term. Users seem to be able to achieve system-clock rates that are 30–50% of the speed grade. Thus with 250 MHz parts an 80 MHz clock frequency is feasible.

A 32-bit adder would require approximately 62 CLBs, enabling roughly 8 to be implemented on the largest CLB available in 1993. The speed would be approximately 20–50 MHz. Thus the reprogrammable arrays implemented in a more advanced (but standard) process and with probably larger die sizes (hence cost) are roughly of the same complexity as

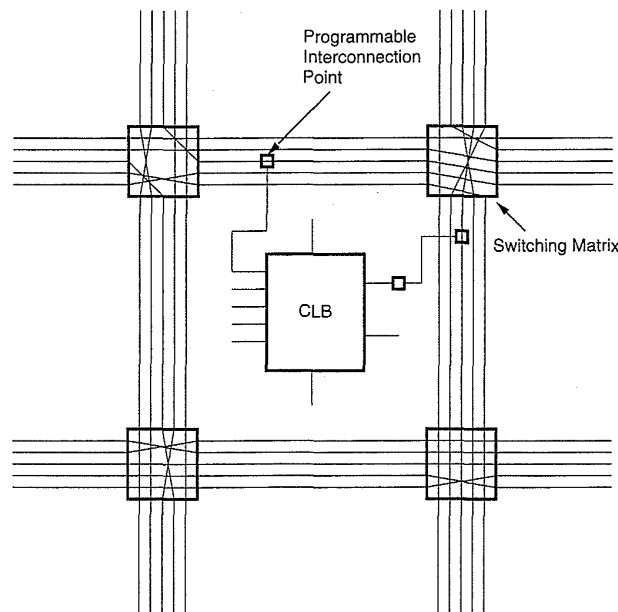


FIGURE 6.18 XILINX cross-bar connect and CLB local connect example

Micron Ex. 1037, p. 442
Micron v. YMTC
IPR2025-00119

the programmable example given in the previous section implemented in a less dense process.

While the XILINX arrays are stand-alone programmable gate arrays, the ideas may be of use to the IC-system designer who wishes to embed some reprogrammable logic within a larger system. In addition, the IC designer may find that prototyping a design in such an array might aid in system debug of a chip function. A significant advantage of the reprogrammable gate array is the ability to redesign the internals of a chip by changing software. This can be of considerable advantage in a product that has to undergo field updates.

6.3.4.2 Algotronix

An example of a regular programmable array is the CAL1024 (Configurable Array Logic) from Algotronix.[16,17] This architecture contains 1024 identical logic cells arranged in a 32-by-32 matrix. At the boundary of the chip, 128 programmable I/O pins allow cascading the chips in even larger arrays. The cell interconnect is shown in Fig. 6.19. Each cell is connected to the East, South, West, and North neighbor. In addition two global-interconnect signals connect to each cell. These are used to supply a low-skew signal to all cells for clocking. Each cell also receives row select lines and bit lines that are used to program RAM bits within the logic cells that dynamically customize the logic cell.

The cell design is shown in Fig. 6.20. It consists of four "through" multiplexers to route single-bit signals entering from the North, South, East, and West. In addition two multiplexers route a selection of signals to a function

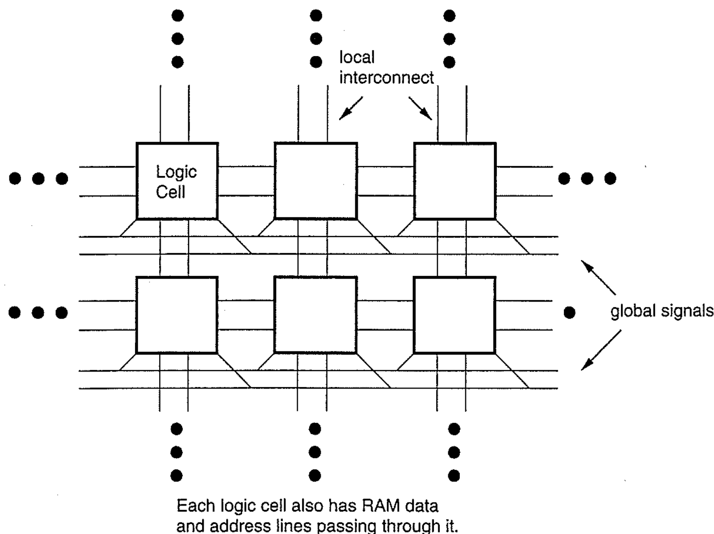


FIGURE 6.19 Algotronix FPGA chip architecture

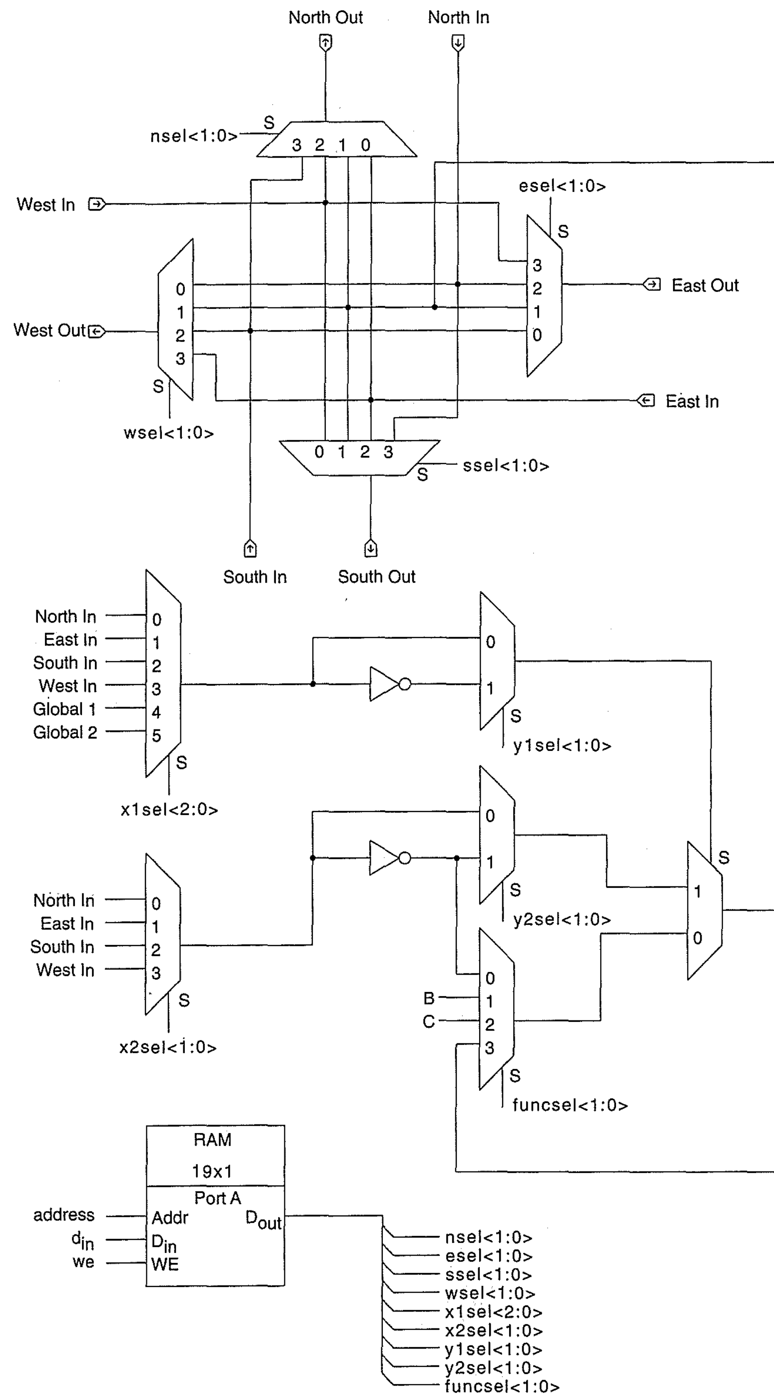


FIGURE 6.20 Algotronix cell design

Micron Ex. 1037, p. 444
Micron v. YMTC
IPR2025-00119

TABLE 6.2 CAL Logic Cell Functions

NUMBER	FUNCTION	NUMBER	FUNCTION
0	ZERO	8	–X1.X2
1	ONE	9	–X1.–X2
2	X1	10	X1+X2
3	–X1	11	X1+–X2
4	X2	12	–X1+X2
5	–X2	13	xnor(X1,X2)
6	X1.X2	14	–X1+–X2
7	X1.–X2	15	xor(X1,X2)
16	D Clk Latch	17	–D Clk Latch
18	D –Clk Latch	19	–D –Clk Latch

unit. These signals include the signals entering on the orthogonal edges of the cell, two global "clock" signals, and the output of the function block for feedback situations (latches). The muxes are controlled (as in the XILINX array) by small 5-transistor static RAM cells. The functions that the logic cell can implement are detailed in Table 6.2.

The I/O pads are very interesting. The trick is to use only one pin for I/O into and out of the array but have the communicating chips automatically deal with two pins that are outputs. The pads achieve this by using a ternary (three-level) logic scheme to sense when two outputs are driving each other via a contention circuit. This is then used with an XOR gate, as shown in Fig. 6.21, to deduce the correct input value.

Design is similar to both the XILINX and Actel approaches, where substantially automatic techniques can place and route a CAL chip. Unlike both other approaches, however, all routing (save the global clock lines) must pass through cells to get from one point to another. Thus, in the worst case, a signal may have to travel through 64 cells. Although implemented with fast transmission gates, this still can result in a substantial delay. For instance the through routing delay is in the range of 0.5 *ns*–2 *ns*, resulting in a delay of 32 *ns*–128 *ns*. However, an intriguing option with this type of array is that the programming can be changed almost in real time. Thus one can think of

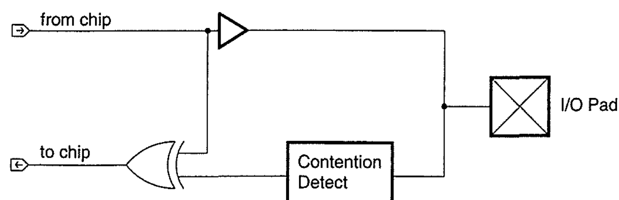


FIGURE 6.21 Algotronix I/O circuit

Micron Ex. 1037, p. 445
Micron v. YMTC
IPR2025-00119

actually having a computer program that "executes" on such an array many times faster than conventional machines.[18]

From a complexity viewpoint, a single-bit adder would take 4 cells. Thus a 32-bit ripple-carry adder would take 128 cells, and 8 adders could fit on a 1024-cell chip (1.5 μm CMOS). The speed would be in the 500 *ns* range. In all of these array architectures serial arithmetic may be preferable to parallel arithmetic because low-delay connections can be made between adjacent cells.

6.3.4.3 Concurrent Logic

The CLi6000 series is another example of a regular array style FPGA.[19] Current designs have between 1000 and 3136 cells, with prospects of up to 10,000 cells per chip in the next few years. As an example the CLi6005 consists of a 7-by-7 array of superblocks. Each superblock has an array of 8-by-8 logic cells. Each logic cell connects to the four nearest neighbors and to a local and express bus (Fig. 6.22). The cell structure is shown in Fig. 6.23.

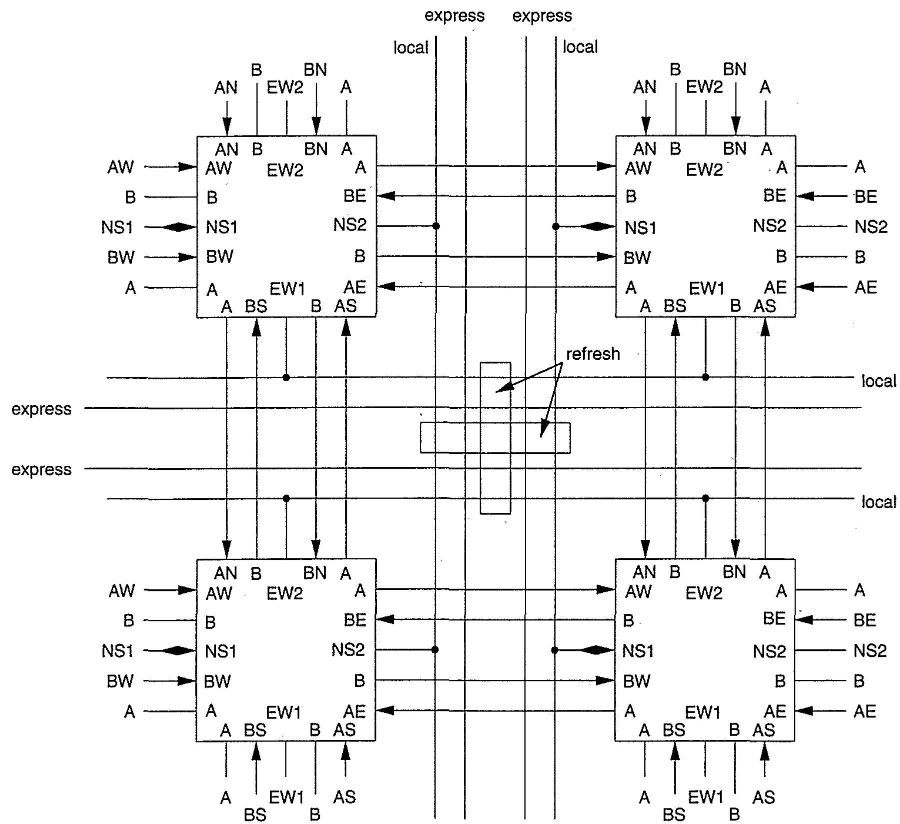


FIGURE 6.22 Concurrent logic array details

Micron Ex. 1037, p. 446
Micron v. YMTC
IPR2025-00119

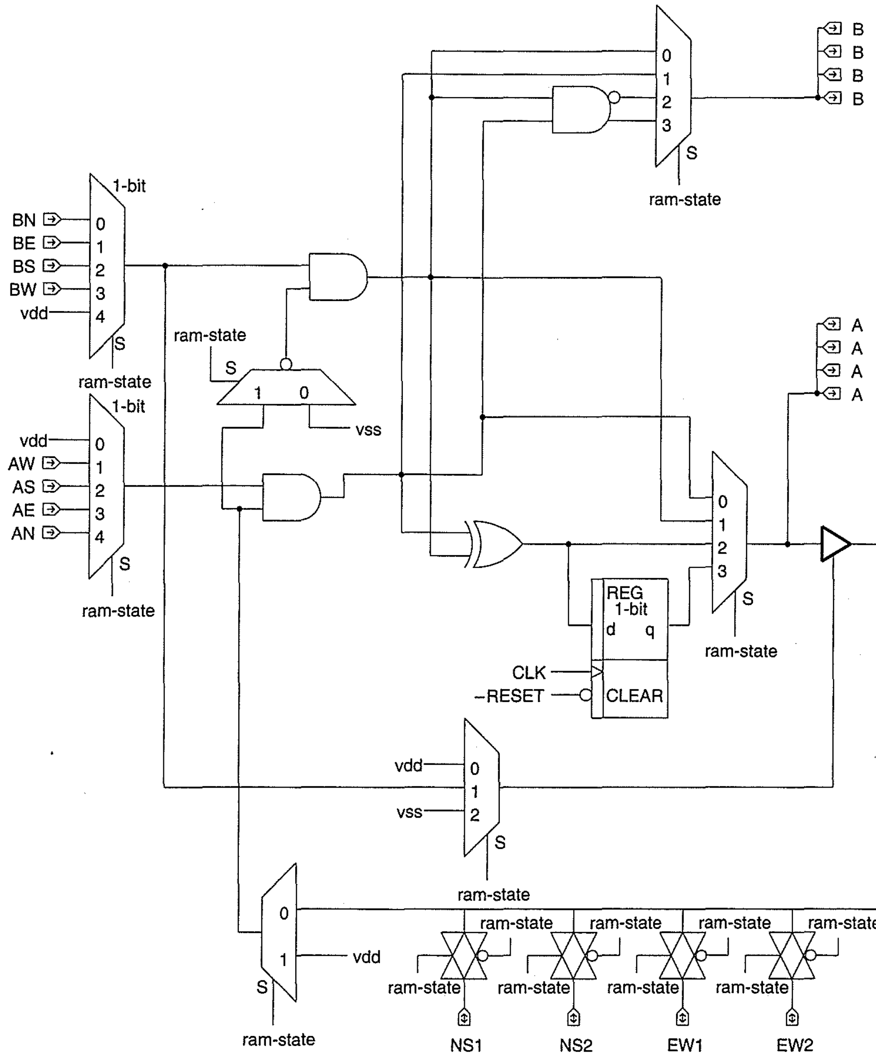


FIGURE 6.23 Concurrent Logic array logic cell

Compared with the Algotronix cell it has considerably more functionality within a cell. A resettable register, XOR, and an AND gate are included. Thus, for instance, a single-counter bit can be implemented in a single cell.

6.3.5 Sea-of-Gate and Gate Array Design

Programming interconnect on chips is a method of reducing the design cost of an integrated circuit. For small-volume chips this can have a direct impact on the part price. The most popular style in use for the implementation of general logic functions is the Sea-of-Gates (SOG) or Gate Array structure, in which

the core of the chip contains a continuous array of n- and p-transistors. A vendor stocks what are called master or base wafers that have been processed up to the stage of laying down polysilicon (i.e., the transistors have been formed). Personalization is then achieved by using design-specific metalization and contacts. The cost is kept down because of the following factors:

- The wafer cost is kept low because large numbers of base wafers may be used for many different designs.
- Only 2–5 masks need to be generated, thus keeping mask costs low.
- Design time is small due to highly automated tools for placement, routing, and testing.
- Packaging cost is kept low due to standard bond-outs and packages.
- Processing time is kept to a minimum because only the top metalization steps need be run.
- Testing costs are kept low because common test fixtures are used for multiple designs.

A typical SOG structure is shown in Fig. 6.24. It consists of a continous strip of n- and p-transistor diffusions adjacent to substrate diffusions. Poly-

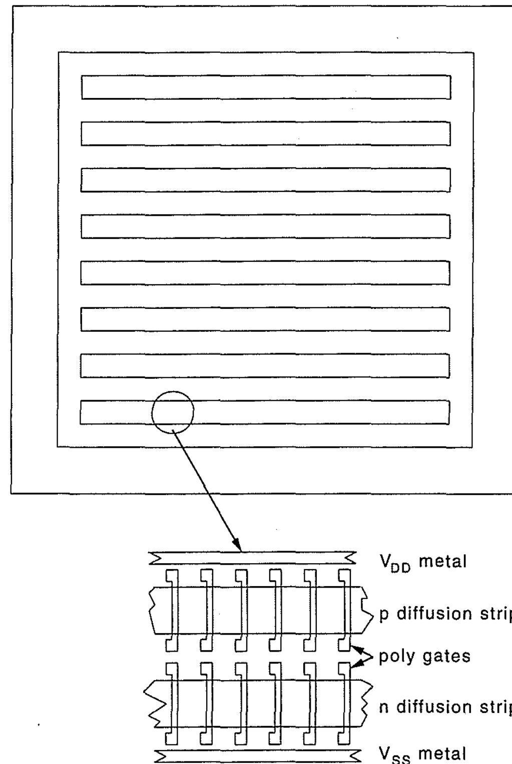


FIGURE 6.24 Sea-of-gates (SOG) chip layout architecture

Micron Ex. 1037, p. 448
Micron v. YMTC
IPR2025-00119

silicon crossing the n and p diffusions forms a continuous horizontal array of transistors. These rows are repeated vertically. The core of an SOG chip so constructed is surrounded by an array of I/O cells that can also be programmed by metalization. Routing channels are formed by routing over the top of unused transistors. Gate arrays, which predate SOG structures, used fixed-height routing tracks. Wiring between active logic rows in an SOG chip occurs over the top of unused transistors, while in a Gate Array the routing is constrained to a routing channel. Fig. 6.25 shows a collection of gates wired together illustrating the routing over the top of the transistor rows. The necessity to pick a number for the Gate Array routing track density thereby constraining the number of horizontal routes gave way eventually to the continuous-array SOG approach.

A number of design decisions have to be made when designing the base array.[20] These include the following:

- The overall size of the core array.
- The macro structure of the strips:
  - –How many n rows and p rows there are per horizontal strip, and how they are routed.
- The micro architecture:
  - –The size and ratio of the n- and p-transistors.
  - –The number, direction, and layer of routing tracks.
  - –The method of logic-gate isolation.
  - –The personalization method.

Usually, the core-array sizes vary from small to large die sizes. When a given system is being planned, the actual density of transistors is mapped to an equivalent raw gate number and then to an effective usable gate level that reflects the cost of routing and placement overheads. For instance, if the column pitch of the array is 10μ and the row pitch is 100μ, then an 8 mm-by-

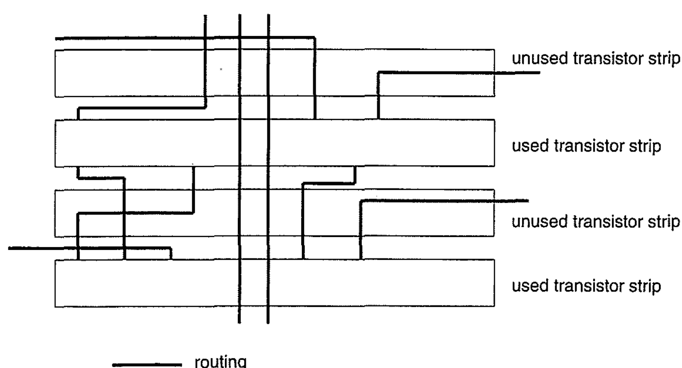


FIGURE 6.25 SOG gates wired together showing routes over unused transistors

Micron Ex. 1037, p. 449
Micron v. YMTC
IPR2025-00119

8 mm core chip would contain 800 × 80 = 64,000 transistor pairs. This corresponds roughly to 16,000 2-input NAND gates. At a 40% usability index (the number of gates that may be used/the total number of gates), this means we can expect to use 6400 gates on this base core.

Most SOG structures have a single row of n- and p-transistors. Some designers have found it advantageous to use a double row of n-transistors and a double row of p-transistors to aid in the implementation of memories and dynamic logic.[21] Other designers, wishing to implement analog circuits, choose arrays of transistors that are suitable for those applications.[22]

Most designs choose equally sized transistors, presumably because unequal rise and fall times tend to even out. The absolute size of transistors is a trade-off between drive capability, fan-in loading, and the array density required. The size of the transistors also affects the granularity of routing tracks.

Typical examples of geometrically isolated and gate-isolated designs are shown in Fig. 6.26 and Fig. 6.27. In Fig. 6.26 a geometrically isolated design typical of early gate-arrays is shown in which three n-p pairs are coupled to form a cell.[23] N- and p-transistors are equally sized. The polysilicon gates are commoned. The "dog-bone" poly connections on the transistor gates allow for routing. Substrate connections are placed below the n-transistor strips and above the p-transistor strips. A typical SOG design is

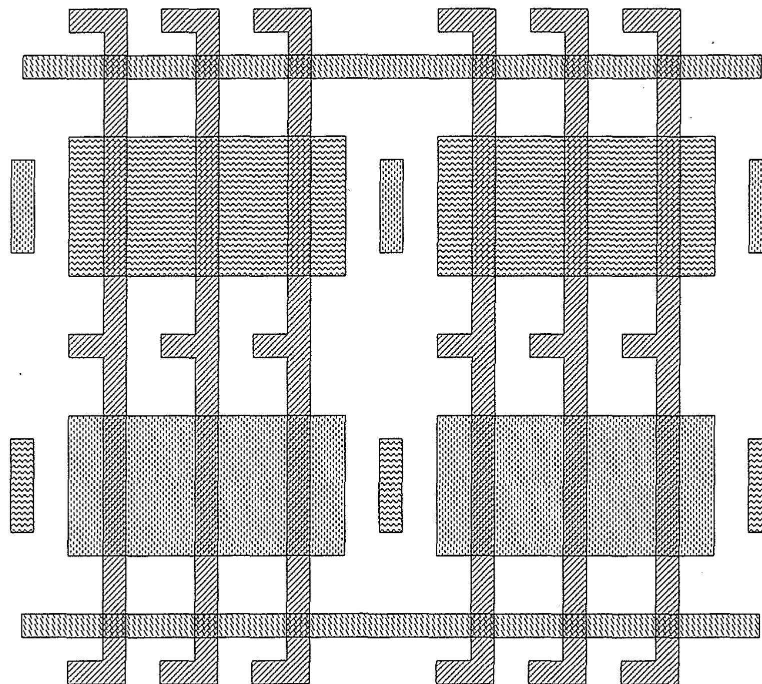


FIGURE 6.26 Geometry isolated SOG or Gate-array base cell

Micron Ex. 1037, p. 450
Micron v. YMTC
IPR2025-00119

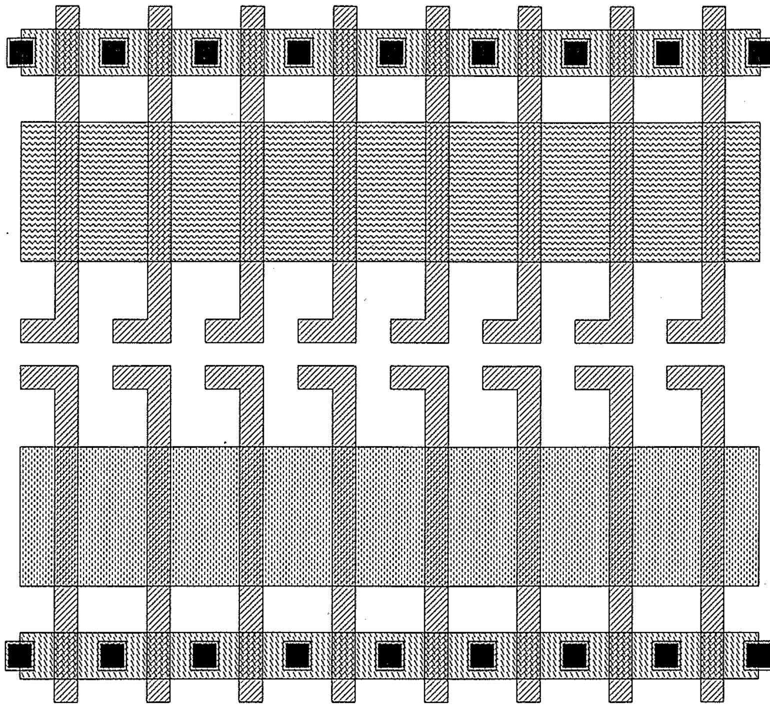


FIGURE 6.27 Transistor- or gate-isolated SOG cell

shown in Fig. 6.27. The key point about the SOG structure is that the transistors at the end of a gate serve to isolate adjacent gates. This is achieved by tying the gate of the n-isolation transistor to V_{SS} and the gate of the p-isolation transistor to V_{DD}. Where adjacent gates share a V_{SS}- or V_{DD}-connected transistor, the isolation transistor is not required. Substrate and well connections run under power busses at the bottom and top of the cell.

The personalization may be completed in a number of ways. For instance, possible methods are:

- Single-layer metal.
- Single-layer metal and contacts.
- Double-layer metal and contacts and vias.
- Triple-layer metal, vias, and contacts.

The personalization of the arrays shown in Fig. 6.26 and 6.27 for a 3-input NAND gate and a D latch are shown in Fig. 6.28. Both cells are arranged to have metal1 running horizontally and metal2 vertically. Note how for the NAND gate the geometrically isolated gate is smaller but for a more complex structure the transistor isolated array is much smaller.

Micron Ex. 1037, p. 451
Micron v. YMTC
IPR2025-00119

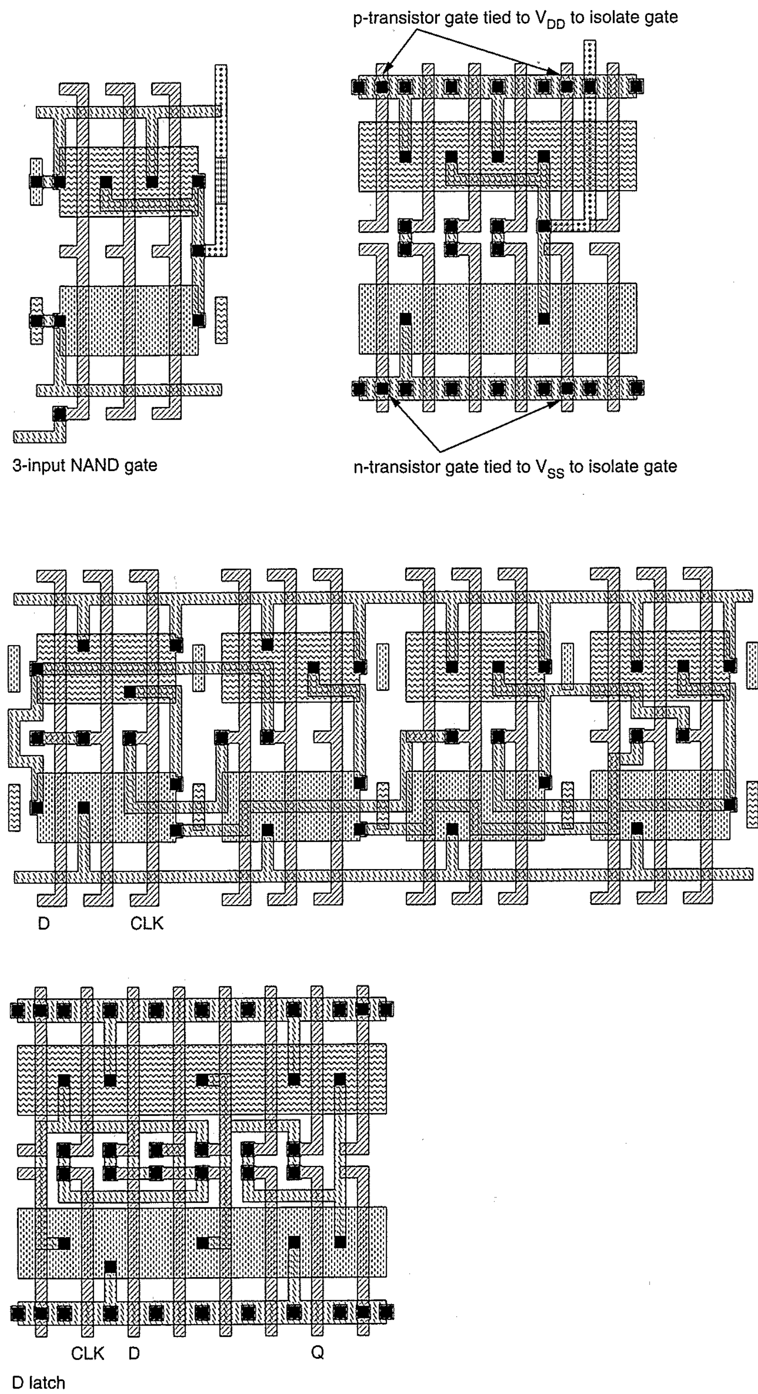


FIGURE 6.28 Personalization of a 3-input NAND gate and a D latch in geometry isolated and transistor isolated SOG structures

Micron Ex. 1037, p. 452
Micron v. YMTC
IPR2025-00119

The routing style of most SOG orients the macrocells (i.e., NANDs, NORs, REGISTERs) along the rows as illustrated in Fig. 6.27, with routing running horizontally between rows of macrocells. In an alternative strategy, the cells are grouped in columns[24] with routing tracks running vertically between columns of macrocells on the chip. The latter approach has been shown to provide 1.08 to 1.31 higher gate density than the row-based approach.

6.3.6 Standard-cell Design

Whereas gate-array architectures standardize at the chip geometry level, it is possible to standardize at the logic or function level. That is, a specific design for each logic gate in a library can be created. This is the basis for what is termed standard-cell or cell-based design. Library cells are normally created for the following classes of circuits:

- SSI logic (nand, nor, xor, aoi, oai, inverters, buffers, registers).
- MSI logic (decoders, encoders, parity trees, adders, comparators).
- Datapath (alus, adders, register files, shifters, bus extractors, and inserters).
- Memories (RAM, ROM, CAM).
- System-level blocks (multipliers, microcontrollers, UARTs, RISC cores).

A design is captured using the standard cells available in a library via schematic or HDL. The layout is then normally automatically placed and routed by CAD software. For SSI and MSI blocks, the layout style is usually identifiable as rows of constant or near-constant height blocks separated by rows of routing. As the complete layout is being done, optimization of the height of routing channels may be completed by good placement. Most manufacturers have extended the SSI/MSI standard-cell technique to the design of datapaths and other higher-level functions such as microprocessors and their peripherals. Another fundamental component of a standard cell system is a selection of memories. Often these are available as a set of parameterizable modules based on word width, number of words, and number of read- and write-ports.

Compared to gate-arrays, standard-cell designs provide a density advantage at the cost of increased prototype costs and possibly increased design complexity. However, where manufacturers have implemented sizable circuit blocks, the productivity of a standard-cell approach might in fact be better that that of a gate-array because the function does not need to be designed.

Micron Ex. 1037, p. 453
Micron v. YMTC
IPR2025-00119

6.3.6.1 A Typical Standard-cell Library

The LSI Logic standard cell library[25] is representative of a large number of libraries that are available. Frequently the SSI logic blocks come in a density-optimized version and a speed-optimized version. Figure 6.29 shows possible layouts for a low-power and normal-power 2-input NAND gate. Metal power busses run horizontally at the top and bottom of the cells. Connections to the cells are available at the top and bottom of the cells. As these hypothetical cells are implemented on a salicided process, connections to the inputs of the gates are made in polysilicon. The density-optimized versions use minimum-sized transistors to achieve the smallest-height standard-cell while the speed-optimized versions use large transistors to provide good driving capability. A summary of typically available cell types is summarized in Table 6.3.

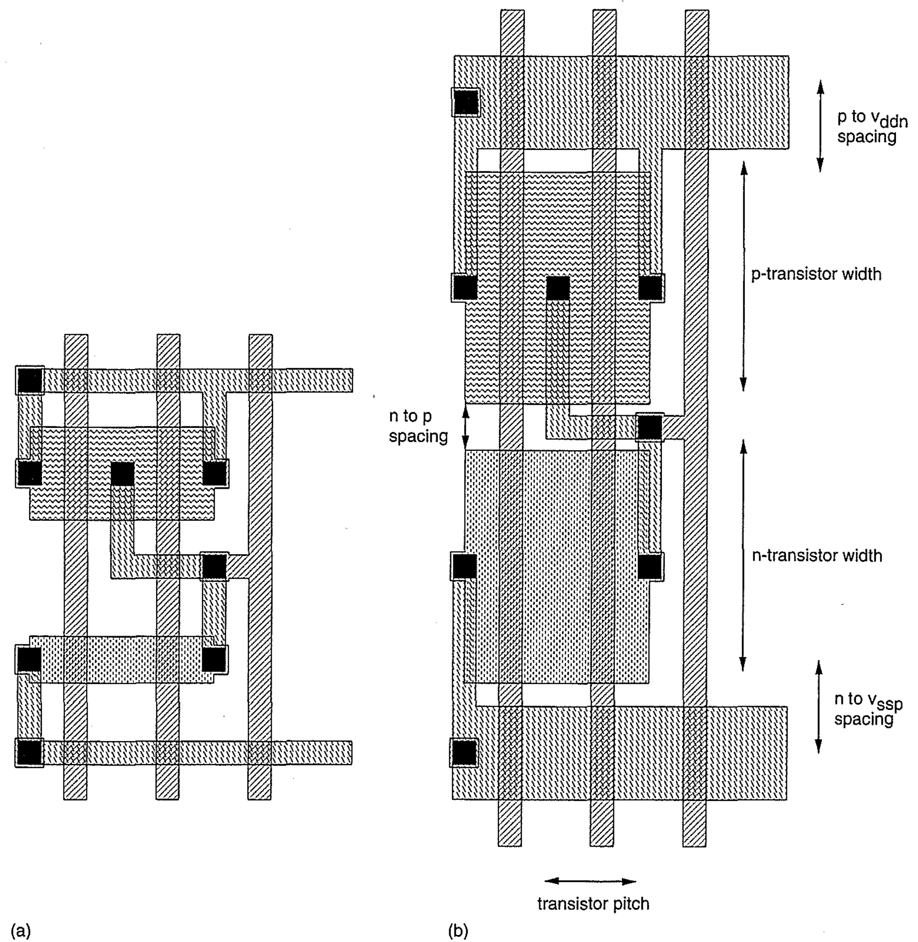


FIGURE 6.29 Typical standard-cell structures showing low-power and regular-power cells

Micron Ex. 1037, p. 454
Micron v. YMTC
IPR2025-00119

TABLE 6.3 Typical SSI Standard-cell Library Summary

GATE TYPE	VARIATIONS	OPTIONS
inverter/buffer/tristate buffers		High-, Normal-, Low-power
nand/and	2–8 inputs	High-, Normal-, Low-power
nor/or	2–8 inputs	High-, Normal-, Low-power
xor	2–3 inputs	High-, Normal-, Low-power
xnor		High-, Normal-, Low-power
aoi (and-or-invert)		High-, Normal-, Low-power
oai (or-and-invert)		High-, Normal-, Low-power
multiplexers	2–8 inputs, inverting/noninverting	High-, Normal-, Low-power
schmitt trigger	inverting/noninverting	High-, Normal-, Low-power
adder/half-adder	normal, fast	High-, Normal-, Low-power
latches	D, asych/synch clear/set, scan	High-, Normal-, Low-power
registers	D, JK, asych/synch clear/set, scan	High-, Normal-, Low-power
I/O pads	in, out, tristate, bidirect, boundary scan, limited slew rate, crystal oscillator	Various current options 1–16 mA

In addition various parameterizable macro cells such as register files, FIFOs, RAMs and ROMs may be provided.

Wide varieties in standard-cell topologies exist. An example of a 3μm library may be found in *CMOS3 Cell Library* by Dennis V. Heinbruch.[26] These cells are very intricate and designed to minimize parasitics and to maximize performance within a given area. Such libraries take a long time to create. Another approach is to abstract the geometry of the cells to allow rapid redeployment in a new technology. Other variations are shown in Fig. 5.15. Where no salicide is available, the polysilicon gates might be strapped in metal2, to eliminate any *RC* delays that might occur in routes that pass through a number of rows of cells.

Figure 6.30 (also Plate 6) shows a three-level-metal standard-cell strategy used at TLW (for a 3-input NAND gate). In this example, internal cell connections are completed in metal1. Connections to gates occur at the center of the cell with a double-via structure from poly to metal2. Metal3 runs horizontally and metal2 runs vertically; thus the cells may be completely covered with routing. With this kind of a cell combined with good automatic placement, very good densities can be achieved. With a library where the

Micron Ex. 1037, p. 455
Micron v. YMTC
IPR2025-00119

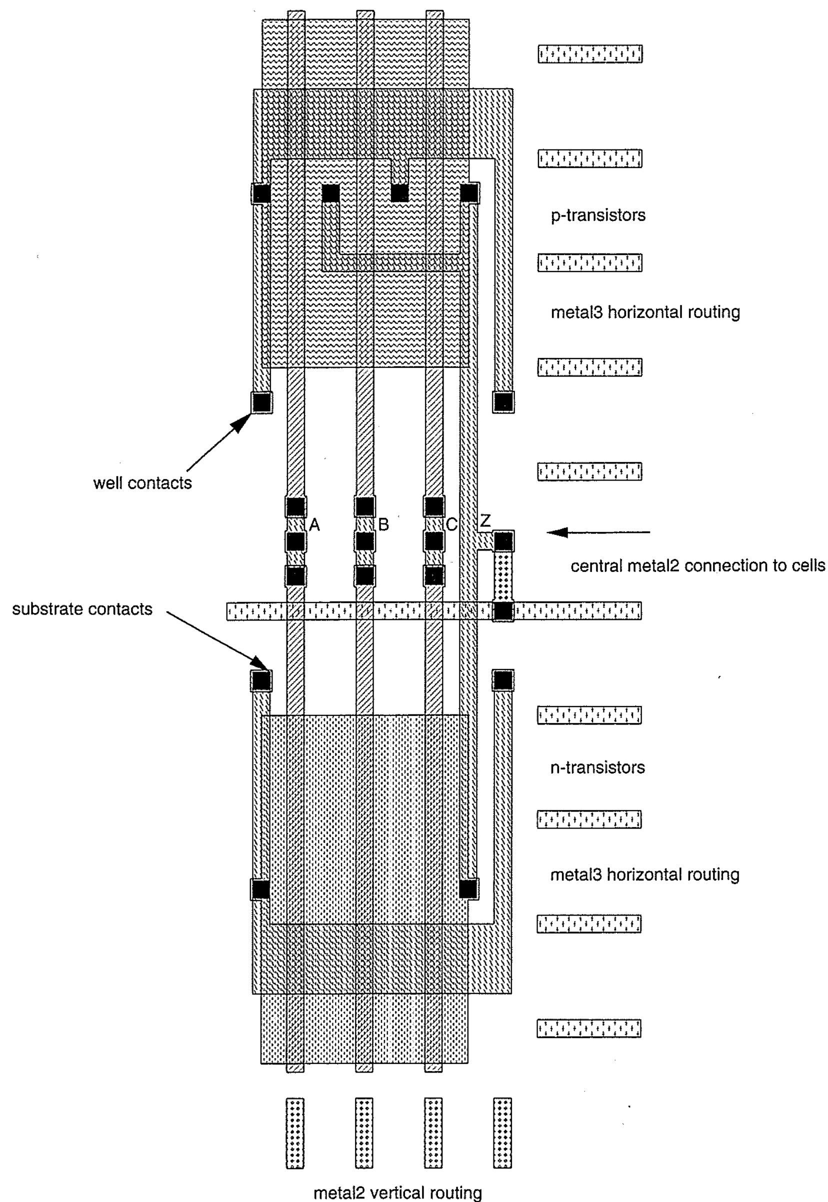


FIGURE 6.30 Three-level-metal standard cell

size of the transistors may be parameterized and some generator support for regular-array structures such as datapaths and memories is available, the density can rival that done by hand. This means that other than for special analog, memory, or I/O blocks, all layout can be compiled with little impact on die cost but a big impact on productivity.

Micron Ex. 1037, p. 456
Micron v. YMTC
IPR2025-00119

6.3.7 Full-custom Mask Design

Full-custom design is the name given to the technique where the function and layout of practically every transistor is optimized. Traditionally, this is how most commercial designs have been done from the beginning of IC-design history. Many times, nonconventional circuit forms or clocking methods will be used in an effort to decrease size or increase speed. Design involves detailed manipulation of the geometric layout ("polygon pushing") and detailed circuit simulation of every circuit structure. As a historical point, it is interesting to note that even in the mid-1970s custom design was long on geometry and short on any kind of verification (due to lack of compute cycles). Design entry might have included cutting your own mask from Rubylith®, entering the geometry via a text editor (having drawn it by hand), or digitizing the same hand-drawn layout with a large digitizer. If you were lucky you may have seen the layout briefly on a monochrome storage display. Frequently, using a library cell consisted of deft use of a pair of scissors, some tape, and an old layout plot. Design-verification tools consisted of a large room, knee pads, and coloring pencils to color layout plots. Of course, circuits consisted of 10*s* to 100*s* of transistors.

In these times, for digital CMOS circuits, companies rarely use full custom design due to the high labor content and low productivity. Exceptions to this include the design of memory and commodity parts such as FPGAs and the design of quasi-analog components such as phase-locked loops. In addition, large mega-cells such as RISC microprocessors may be custom-designed for speed and cost reasons.

In 1989, Fey[27] found that for full-custom designs the productivity ranged between 6 and 17 transistors per day for logic transistors and 60 to 230 transistors per day for ROM transistors.

6.3.8 Symbolic Layout

Because a major component of custom design is the physical layout of new modules, IC-designers sought methods of reducing the time of entry of the physical layout. In the early 1970s MOS logic designers frequently used manually drawn shorthand notations for layout structures (the author included). As computers became more prevalent, this practice led to *symbolic* layout systems. These systems attempt to abstract the layout in some manner in order to reduce the complexity of the task, thus increasing productivity. Over the last 15 years a number of strategies have evolved.

6.3.8.1 Coarse-grid Symbolic Layout

The idea behind coarse-grid symbolic layout involves dividing the chip surface into a uniformly spaced grid in both the X and Y directions. The grid size

Micron Ex. 1037, p. 457
Micron v. YMTC
IPR2025-00119

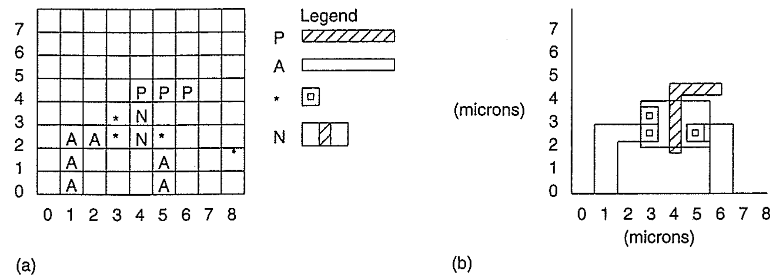


FIGURE 6.31 Coarse-grid symbolic layout

represents the minimum feature or placement tolerance that is desired in a given process, and is usually selected by close consultation between design-tool developers and semiconductor-process engineers. For each combination of mask layers that exist at a grid location, a symbol is defined. Figure 6.31 shows a typical symbol set and layout. Given a particular design system, these symbols are then placed on the grid to construct the desired circuit, much the same way as you would tile a floor. Symbol sets could be defined as characters or graphical symbols, which was invaluable in the early days of color displays because character-only color displays were a lot cheaper than color graphics displays.

American Microsystems International (AMI)[28,29] and Rockwell International[30] pioneered the use of character-based symbolic layout.

This style of symbolic layout provided for first-generation symbolic layout with low-cost design entry. In general these systems have been supplanted by more modern approaches.

6.3.8.2 Gate-matrix Layout

A character-based symbolic layout style was developed at Bell Labs[31] specifically for custom CMOS circuitry. It improved on coarse-grid symbolic layout by providing a regular layout style where a matrix of intersecting transistor diffusion rows and polysilicon columns is employed. The intersection of a row and a column is a potential transistor site (poly crossing diffusion). A related style is featured in Piguet et al.[32]

The evolution of this technique from a standard-cell viewpoint is shown in Fig. 6.32. Figure 6.32(a) shows a circuit implemented in terms of standard cells (four 2-input NANDs and one inverter). Note that intercell connections are in metal. Rather than running these connections in metal, we can run vertical polysilicon columns corresponding to each gate signal. The transistors may then be placed on the polysilicon signals and interconnected, as shown in Fig. 6.32(b). Note that vertical columns may be either polysilicon or diffusion. Horizontal rows are transistors and/or metal routing tracks. Metal may also run vertically. A character-symbolic layout for the layout may be

Micron Ex. 1037, p. 458
Micron v. YMTC
IPR2025-00119

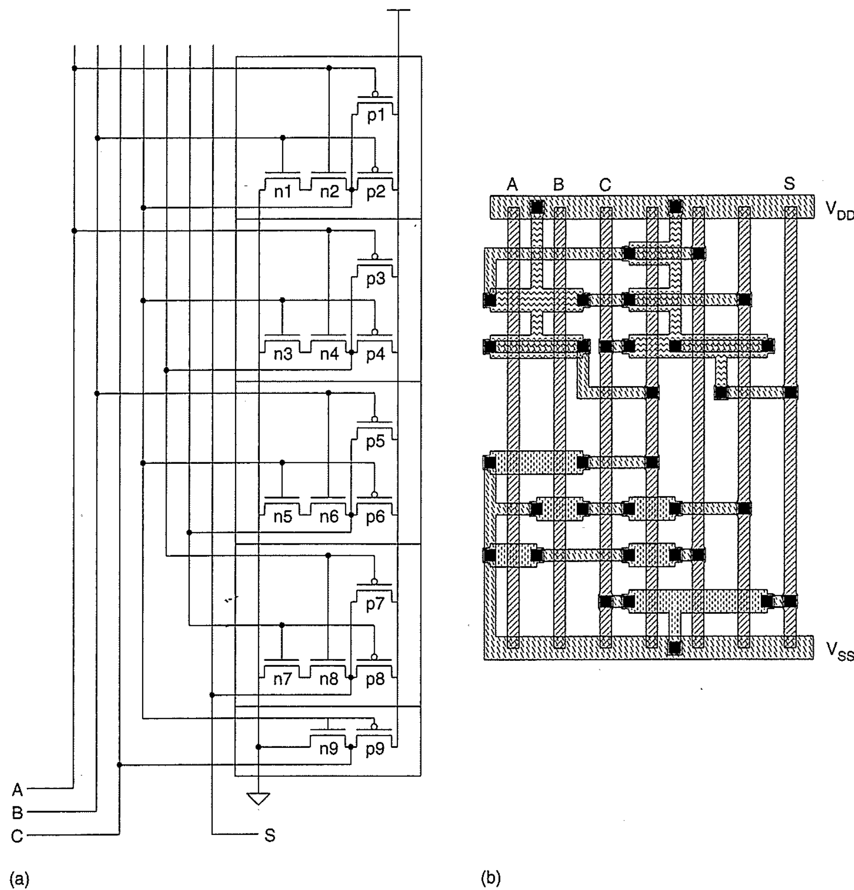


FIGURE 6.32 Evolution of gate-matrix layout: (a) standard cell layout (in schematic form); (b) gate-matrix layout

created (in fact this is how the layouts were first captured). The following rules summarize the gate-matrix technique:

1. Polysilicon runs only in one direction and is of constant width and pitch.
2. Diffusion wires (of constant width) may run vertically between polysilicon columns.
3. Metal may run horizontally and vertically. Any pitch departures from minimum (e.g., power rails) are manually specified.
4. Transistors can only exist on polysilicon columns.

To convert from a character symbolic description to mask artwork, the character matrix is examined and the symbols are expanded to their equivalent mask entities. Operations such as merging horizontal dashes into one metal wire and merging adjacent devices are performed during this phase. Obvi-

Micron Ex. 1037, p. 459
Micron v. YMTC
IPR2025-00119

ously very simple software and limited computer resources are needed to capture designs in this manner.

In common with coarse-grid symbolic layout, gate-matrix-symbolic layout systems have been largely replaced, but the style of layout is still of interest for small- to medium-sized modules.

6.3.8.3 Sticks Layout and Compaction

A popular method of symbolic design is termed "sticks" layout. Here the designer draws a freehand sketch of a layout, using colored lines to represent the various process layers such as diffusion, metal, and polysilicon. Where polysilicon crosses diffusion, transistors are created and where metal wires join diffusion or polysilicon, contacts are formed. Alternatively, specific primitives such as transistors are drawn and interconnected with lines representing conductors. Following this rapid capture of the rough topology, a spacing program or compactor determines the correct spacing between all wires, transistors, and contacts created. The most popular compactor is what is termed a graph compactor. The compactor creates a directed-constraint graph. The nodes of the graph are the primitives, and the branches are used to connect groups that have potential design-rule violations. The weights of the branches are the minimum separations necessary between two nodes. An example of the mapping of a symbolic circuit to a graph is depicted in Fig. 6.33. If there is no spacing necessary between two groups, an edge will not be created between the two groups. Once the graph has been established, the critical path (i.e., the

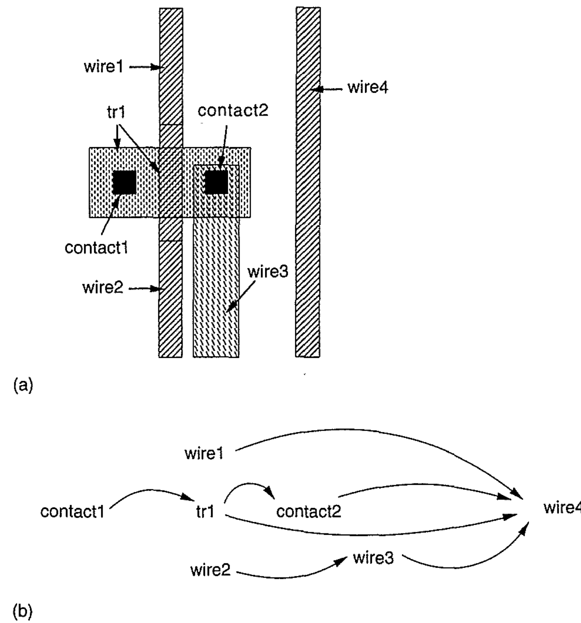


FIGURE 6.33 "Sticks"-symbolic layout compaction: (a) layout; (b) horizontal constraint graph

Micron Ex. 1037, p. 460
Micron v. YMTC
IPR2025-00119

path with the greatest spacing requirement) through the graph can be determined. The nodes in the path can then be placed sequentially. For any given node, there may be a number of paths to it. The critical path to a given node will determine its minimum placement consistent with all design rules. *X* and *Y* passes through the graph are completed to compress the layout.

During the 1980s a large body of CAD research was devoted to "sticks" symbolic layout systems.[33–43] These systems and their commercial derivatives have met with varied success and acceptance.

6.3.8.4 Virtual-grid Symbolic Layout

Virtual-grid symbolic layout[44] is a symbolic layout method that draws on the experience gained in coarse-grid symbolic systems, gate-matrix, "sticks"-type systems, and other approaches, such as ICSYS,[45] developed at the University of Edinburgh and Caltech. In essence, the system approaches design at the layout level by manipulating circuit elements such as transistors and wires as opposed to any form of geometric mask description. These elements are placed on a grid to facilitate easy design capture and use of simplified tools, with the final geometric spacing between grid lines determined by the density and interference of circuit elements on neighboring grid locations. This leads to the notion of a *virtual grid.*

The concept is best illustrated by a simple example, as shown in Fig. 6.34(a). Three vertical wires are shown centered on a grid. The result of using a fixed grid of 10 units and a wire width and separation of 10 units leads to the mask description shown in Fig. 6.34(b). By using a grid in which the spacing varies according to topology, the mask description in Fig. 6.34(c) is constructed. The end result for the designer is that placement on the grid may be done without regard to any design rules. In addition to eliminating design rules, the grid is also used to define circuit connectivity in a manner similar to that employed in schematic capture systems. Here, the notion of a "coordinode," as introduced by Buchanan, is used to capture physical location, structural connectivity, and behavioral state. As its name suggests, a coordinode has the properties of a coordinate, namely some *xy* position that will eventually map to the silicon surface. In addition, it may possess the properties of a node in a circuit, such as voltage or simulation state. Structurally, a coordinode defines the nodes in the network being designed. In the virtual-grid context, a coordinode is mapped to a discrete set of grid points rather than a quasi-continuous set of *xy* coordinates. The grid coordinates form the lines of action in a circuit, defining the essential communication paths in and through a circuit. Local geometric perturbations are handled by software skilled in the art of manipulating geometry.

MULGA is an example of an integrated design system based on these principles still in use.[46,47] The NS system is a VLSI system that has been in use since 1984, which employs the same principles.[48] Further systems have

Micron Ex. 1037, p. 461
Micron v. YMTC
IPR2025-00119

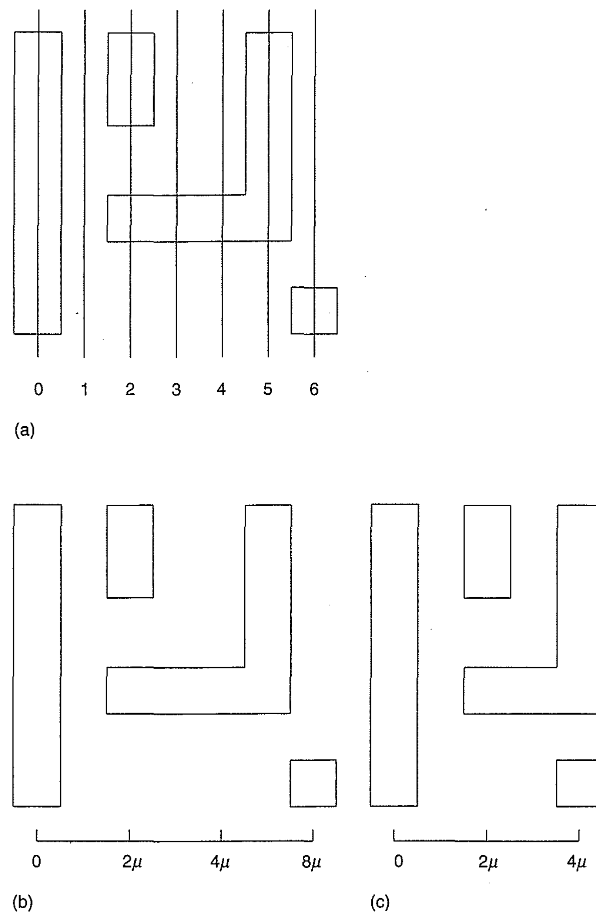


FIGURE 6.34 Virtual-grid symbolic layout

reported the use of similar techniques.[49,50] A virtual-grid-circuit capture system yields the following benefits:

- Design-rule-free topology capture.
- Rapid design capture through the use of point interconnect.
- Fast grid-based algorithms for connectivity audit, compaction, and other design processes.
- Ability to allow simplified parametized cells with automatic geometry generation.
- Hierarchical module assembly.
- Natural target for higher-level silicon compilers (geometry free).

The use of the virtual grid allows a very simple compaction strategy to be used. However, recent virtual-grid systems also use graph based compaction.

Micron Ex. 1037, p. 462
Micron v. YMTC
IPR2025-00119

In terms of productivity, based on roughly 20 chip designs, by using known circuits and just altering the geometry, productivities similar to standard-cell designs are seen while densities comparable with hand layout result.

Symbolic-layout systems are an acquired taste. Many people (such as the author) could not imagine designing without them at the layout level, while many others design quite happily without them. Recently, there has been somewhat of a renaissance of symbolic layout when designers with fixed-geometry designs are confronted with the effort of porting an old CMOS design to a completely new process.

6.3.9 Process Migration—Retargetting Designs

Of concern to the system designer are methods of retaining the investment in engineering large systems where ever-improving process densities mean that the system cost can be reduced over time by incorporating the five ASICs designed in a given year onto one chip in two years, and onto one-fourth of a chip in four years. The following approaches are possible:

- Recast the architecture.
- Recast the logic design.
- Recast the layout in the new process.

The first option requires a complete reengineering of the problem with the associated system and production test vectors. This frequently provides the best technical solution with a varied saving on previous engineering. In the software domain, this is equivalent to rewriting the program to suit a faster computer with, say, a parallel architecture.

The second option allows a design implemented in today's standard cells to be fairly painlessly translated to the next generation standard cells. Timing analysis still has to be completed, but the process can reuse previous engineering efforts. While this process works well for logic, specialized VLSI structures such as memories may be a problem. Some ASIC vendors counter this by providing n-port memory compilers that work in the target processes. The counterpart in the software world is the use of a new compiler to produce code for a new fast processor.

Finally for the "dusty deck" problem, researchers have turned their attention to the problem of migrating mask designs implemented in old technologies to newer smaller processes. Some success has been recorded with systems that extract symbols from old geometry and then recompact the symbols with the new design rules of the target process.[51]

In all cases, the design effort would still contain a large proportion of simulation and timing analysis, which usually constitutes the major portion of the design effort in today's designs.

Micron Ex. 1037, p. 463
Micron v. YMTC
IPR2025-00119

6.4 Design Methods

When starting a design project, the designer has a number of options with regard to the specification level of the chip. Usually the designer starts at the behavioral level and progresses to the RTL level, then to the logic level, and then possibly to the structural level, and finally to the layout level. Depending on the complexity of the design, tools exist to synthesize a chip layout from any of these levels of specification.

6.4.1 Behavioral Synthesis

At the behavioral level, the operation of the system is captured without having to specify the implementation. For instance, the pipelining required to meet a certain speed may not be specified. This is the level that provides the fastest emulation of the system and the one that is best used to debug the operation of the complete system. Obviously, this level is technology-independent.

For the synthesis of complex behavioral descriptions including signal-processing architectures, a rich research literature is available. Researchers have had success with high-level synthesis by building systems to synthesize constrained architectures. Good examples are the Cathedral series of silicon compilers, Cathedral I, which concentrated on bit-serial digital filters[52]; Cathedral II, which compiled collections of communicating sequential DSP processors[53]; and Cathedral III, which was aimed at video-signal-processing architectures.[54] Another example is the LAGER compiler for signal processing architectures.[55] These targeted systems are sometimes called Silicon Compilers, because they take a design from the behavioral (code) to the mask level (silicon).

In principle, a behavioral compiler must perform the following operations:

- Decide upon and assign resources based on area and timing requirements.
- Insert pipeline registers to achieve timing constraints.
- Create microcode and/or control logic.

For instance, consider the following behavioral code fragment:

```
a = a + b*c;
```

This specifies a multiply-accumulate step. Depending on the required speed and word size, this may be implemented as anything from a bit-serial multiplier to a fully parallel Booth-encoded Wallace tree multiplier (see Chapter 8).

Micron Ex. 1037, p. 464
Micron v. YMTC
IPR2025-00119

The vector-drawing architecture shown in Fig. 6.2 is derived from the equation for a straight line,

```
Y = a*x + b;
```

It is not intuitively obvious how the structure in Fig. 6.2 might evolve from the equation above. It took a clever human and difference-equation mathematics to produce the implementation shown.

Behavioral-synthesis systems currently provide very good silicon implementations for narrow (but very useful) classes of problems, and will continue to gain ground as they become more generalized and commercially available.

6.4.2 RTL Synthesis

RTL-synthesis programs take an RTL description and convert it to a set of registers and combinational logic. At this stage of the design process, the architecture has been captured. One research system pioneering this approach to design (and aimed at CMOS) is the Yorktown Silicon Compiler System.[56] There are also a number of commercial systems now available.

Commonly, RTL descriptions are captured using a Hardware Description Language (HDL). In general, RTL HDLs have to capture the following attributes of a design:

- Control flow using if-then-else and case statements.
- Iteration.
- Hierarchy.
- Word widths, bit vectors, and bit fields.
- Sequential versus parallel operations.
- Register specification and allocation.
- Arithmetic, logic, and comparison operations.

An RTL compiler is responsible for converting a description in an HDL into a set of registers and combinational logic. Logic optimization is then used to improve the logic to meet timing or area constraints (Section 6.4.3).

As examples of available commercial systems, some of the transformations that allow RTL descriptions to be synthesized will be given in terms of the VHDL language[57–59] and the Synopsy VHDL Compiler®.[60] Consider the following (sketchy) VHDL description of the difference engine shown in Fig. 6.2(b).

```
package types is
type OP_CODE is (NOP, LOADA, LOADB, LOADF, RUN);
```

Micron Ex. 1037, p. 465
Micron v. YMTC
IPR2025-00119

```
    attribute OP_CODE_ENCODING of OP_CODE:
          type is "000 001 010 011 100";
    end types;
```

The **type** section defines a user-defined type called `OP_CODE`, which will be used to control the difference engine. It states that the 3-bit field has five operation codes to load the three registers, to run the difference engine, or to do nothing (`NOP`). An optional encoding has been assigned via the `attribute` keyword. Thus one operation that the HDL synthesizer does is to assign values to unspecified type fields.

```
    entity DIFF_ENGINE is
    port(
         DATA : in BIT_VECTOR (0 to 7);
         OP : in OP_CODE;
         CLOCK : in BIT;
         SIGN : out BIT;
         );
    end DIFF_ENGINE;
```

The **entity** section defines the name of the design (`DIFF_ENGINE`) and denotes a port interface to the module. For instance, in this case `DATA` is defined to be an 8-bit bit-vector that is an input.

```
architecture DIFF_ENGINE_1 of DIFF_ENGINE is
signal A,B,F : BIT_VECTOR (0 to 7);
begin
     process
     begin
          wait until (not CLOCK'stable and CLOCK = '1');
               case OP is
                    when LOADA  => A <= DATA;
                    when LOADB  => B <= DATA;
                    when LOADF  => F <= DATA;
                    when RUN  =>
                         if (SIGN = '1') then F <= F+A
                         else F <= F+B
                         end if;
                    when NOP => F <= F
               end case;
          SIGN <= (F<0);
     end process;
end DIFF_ENGINE_1
```

In the above RTL description, a number of statements are illustrated. First, the signal statement defines some internal signals that have local scope within the module. The **process** statement indicates a section of code to be implemented sequentially. The **wait** statement indicates the presence of

Micron Ex. 1037, p. 466
Micron v. YMTC
IPR2025-00119

clocked registers and specifies that the registers are triggered on the rising edge of CLOCK. The **case** operator indicates a multiplexer, as does the if operator. Finally, the "+" operator indicates an addition while the "<" operator indicates a comparison operator. When combined with an appropriate substrate (simulation, logic library, operator definition), the above description may be compiled into a set of logic gates and registers.

In the case of state-machines, RTL compilers need to provide for automatic state-assignment[61] and minimization.

6.4.3 Logic Optimization

Logic optimization programs take logic descriptions as generated by an RTL synthesis (with the registers stripped out) or which are generated directly at the logic level and optimize the network of gates that are required to implement the function specified by the logic description for a given logic library. The registers are then reunited with the optimized logic, and the physical layout for the system may be implemented using largely automatic techniques. The methods for this are well understood[62] and there are a number of very successful commercial systems.

A typical flow through a typical logic synthesis system is shown in Fig. 6.35. The design is commenced with a logic description. This may be in the form of Boolean equations or a schematic netlist of logic gates. The objec-

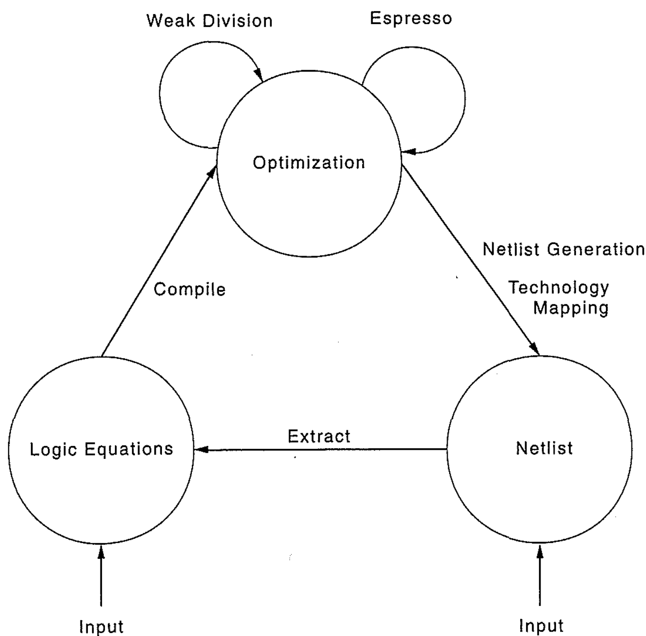


FIGURE 6.35 Logic-optimization flow

Micron Ex. 1037, p. 467
Micron v. YMTC
IPR2025-00119

tive of a logic optimization scheme is to manipulate the logic to meet speed or area constraints or a combination of both goals. Generally, logic-optimization systems divide the problem into two stages:

- A technology-independent phase in which the logic is optimized according to algebraic and/or Boolean techniques.
- A technology-mapping phase, which translates the technology-independent description derived in the previous step to specific library standard cells, FPGA elements, or other implementable logic gates.

The technology-independent aspect of logic optimization uses a large body of algorithms that operate on logic networks, using both Boolean and algebraic techniques. Most often logic-optimization systems provide a means to read logic networks, manipulate them, perform a technology mapping, and save the resulting structure to be used by an automatic layout program or some other tool. A typical flow through an optimization script is as follows[63]:

- Network organization.
- Two-level minimization.
- Algebraic decomposition.
- Iterative improvement.

Having read the design in, the first step might be to perform tasks such as eliminating constant nodes and redundant inverters or converting the logic to a two-level PLA sum-of-products form (see Chapter 8). Next, a two-level minimization might be invoked. Espresso[64] is an example of a widely used two-level minimization program. Next, algebraic decomposition may be used that introduces new nodes into the logic network in a manner that minimizes the cost. One technique used is known as "weak division."[65] This is used to decompose two-level logic expressions into multiple-level logic expressions. It operates by repeatedly "dividing" the expressions by subexpressions that appear more than once in the set of expressions that constitute the design. The most suitable subexpression is chosen by evaluating a cost function that may be based on reducing the number of literals (area) or other functions related to the levels of logic (speed). Consider the following equations:

```
f1 = aef + bef + ceg
f2 = aeg + bg + def
```

The common subexpressions are `ef`, `ae`, `eg`, and `a+b`. Of these, `ef` saves the most literals. When `ef` is divided into all subexpressions, the result is

```
f1 = (a+b)t1 + ceg
```

Micron Ex. 1037, p. 468
Micron v. YMTC
IPR2025-00119

```
f2 = aeg + bg + dt1
t1 = ef
```

After this pass the literal eg might be chosen, yielding

```
f1 = (a+b)t1 + ct2
f2 = at2 + bg + dt1
t1 =ef
t2 = eg
```

Finally, algorithms are used to iteratively improve the logic structure. This may employ the algebraic techniques of extraction, factoring, and substitution in addition to decomposition.

Following the technology-independent step, a technology mapper is then used to optimize the gates for a particular technology.[66] Two kinds of optimizers are in popular use. The first consists of a rule base consisting of rules in the form

if (antecedent) *then* (precedent).[67]

They are used to map over small sections of circuitry to choose suitable logic gates for an implementation. Figure 6.36 illustrates some typical rules. Figure 6.36(a) eliminates cascaded inverters, while Fig. 6.36(b) converts NOR and INVERT logic gates into an OAI gate. Other rules might bias gate selection toward faster gates such as NAND gates. Another approach is termed Directed-Acyclic-Graph(DAG) covering.[68] In this approach, what is called a *base-function set* is chosen. This might be a 2-input NAND gate and an inverter. All logic gates in the target library are then described in terms of the base-function elements. These are known as *pattern gates*. The logic network is optimized using the base-function set. This creates a *subject graph.* Graph optimization techniques are then used to find an optimized set of target gates. Figure 6.37(a) shows a base function set. Figure 6.37(c) shows examples of pattern graphs. For the 4-input NAND gate shown in Fig. 6.37(c), two possible pattern graphs are shown. Figure 6.37(d) shows a subject graph in which a particular mapping has been identified. Finally, Fig. 6.37(e) shows the resultant mapped logic implementation.

The MIS[69] and BOLD[70] systems are examples of research-based tools that provide logic minimization. These systems provide an environment that contains a number of minimization techniques. These are used to construct minimization scripts that can be adapted for varying styles of logic minimi-

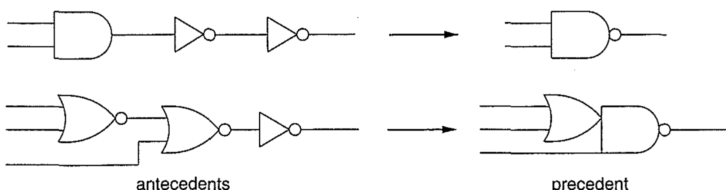


FIGURE 6.36 Rule-based technology mapping

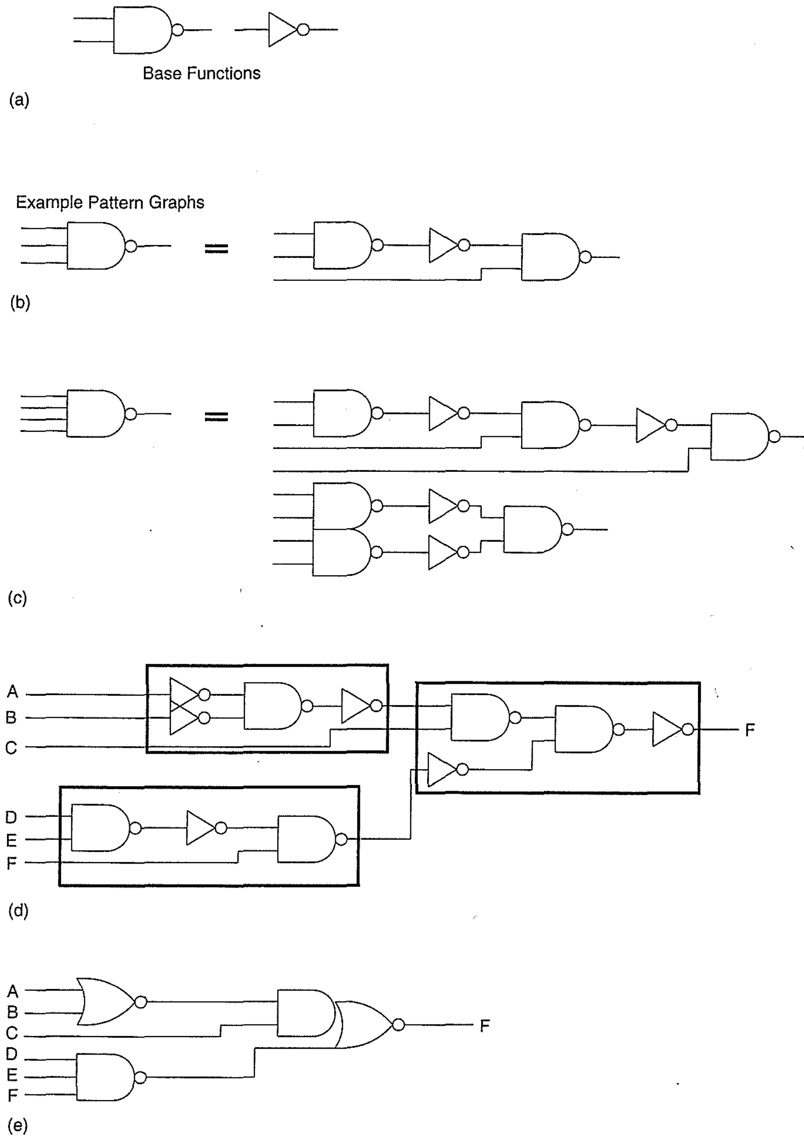


FIGURE 6.37 DAG technology mapping

zation. Figure 6.38 shows the typical inputs to MIS that might consist of a minimization script, a set of equations, and a logic-library definition. The output is a netlist implementing the equations in terms of the logic library. An example of the use of this program is presented in Chapter 8. The EDIF language is frequently used as a common netlist format between design systems.[71]

Apart from increasing design productivity, logic synthesis systems are very useful for transforming between technologies. For instance, a designer might synthesize a circuit in terms of multiple FPGAs, and construct a pro-

Micron Ex. 1037, p. 470
Micron v. YMTC
IPR2025-00119

totype. This might be used to verify the operation of the circuit under real-world conditions and then a single-chip version may be compiled using a gate-array library and the original logic description.

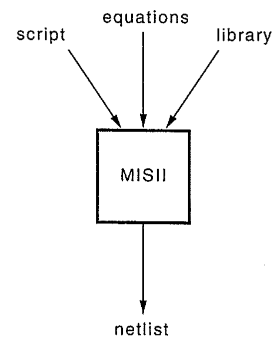


FIGURE 6.38 Logic optimization using MISII

6.4.4 Structural-to-Layout Synthesis

Once a network of logic gates and registers is available, these may be automatically converted to a layout. Software for this task is very well developed, having been refined over the last 15 to 20 years. Gate arrays and standard-cell designs use this approach. There are two main phases that are required: placement and routing.

6.4.4.1 Placement

Placement is the task of placing modules adjacent to each other to minimize area or cycle time (timing-directed placement). Two main automated algorithms have been developed. The Min-cut algorithm[72] takes the blocks at the top level of the chip or module to be placed and finds two approximately equal area-groupings of subblocks with the minimum number of signal interconnections. These two blocks are then placed in the top and bottom half of a conceptual final layout. This process is repeated for these two halves, splitting the conceptual layout into quarters and so on until the leaf cells are reached. This algorithm is very fast and gives good placements. Another popular technique in which the movement of modules is likened to thermal annealing is also used.[73] Modules are initially allowed to move randomly, and the "temperature" of the layout is evaluated by applying some measure such as routing area or timing. As the layout "cools" the routing and/or timing improves. For each proposed subblock movement, the resulting temperature is calculated. If it is higher than the current temperature, the move is not completed. To avoid local minima, the "melt" is reheated and then recooled according to an "annealing schedule." This process is used in the TimberWolf program that was developed at the University of California, Berkeley[74] and refined at Yale University.

6.4.4.2 Routing

A router takes a module placement and a list of connections and connects the modules with wires. This technology is very mature. Types of routers include channel routers, switchbox routers, and maze routers. Channel routers are typified by the YACR2 router[75] and the Greedy router.[76] These routers route rectangular channels. Switchbox routers can route more complex channel shapes than channel routers. Maze routers[77] can route just about any configuration but have comparatively long running times. They are usually reserved for really tough routing problems.

Micron Ex. 1037, p. 471
Micron v. YMTC
IPR2025-00119

A global router[78] is a special router that works during a placement algorithm to try to plan where routes will travel when the layout is finally placed.

6.4.4.3 An Automatic Placement Example

The standard-cell (constant height cells) placement part of TimberWolf takes as inputs the following:

- The .cel file.
- The .blk file.
- The .par file.
- The .net file.

The .cel file describes the connectivity of standard-cells port locations and signal names. A partial example is shown below:

```
cell 2 INVERTER-2
left -36 right 36 bottom -225 top 225
pin name Z_top signal NEXT_STATE<2> 18 225
equiv name Z_bottom 18 -225
pin name A_top signal [169] -18 225
equiv name A_bottom -18 -225
.
pad 0 %%%pad_TOP_0 orient 3
padside T
left -1 right 1 bottom -1 top 1
pin name top_%io-port signal STATE<3> 0 1
.
pad 1 %%%pad_TOP_1 orient 3
padside T
left -1 right 1 bottom -1 top 1
pin name top_%io-port signal STATE<2> 0 1
```

The first `cell` statement denotes an inverter, `INVERTER-2` with output `NEXT_STATE<2>` and input `[169]`. The bounding box of the inverter is specified by the second line. Each successive line specifies a port and its location. Ports that feed through cells may also be specified. Finally, the ordering and location of I/O pads are specified by the `pad` statements.

The .blk file contains information pertaining to the structure of each row in the layout. An example follows:

```
block height 450 class 1
block height 450 class 1 mirror
block height 450 class 1
```

This defines a three-row layout with the middle row mirrored in *Y*.

Micron Ex. 1037, p. 472
Micron v. YMTC
IPR2025-00119

The .par file contains various global parameters to be applied to the layout. The following is an example:

```
rowSep 1.1111112
addFeeds
feedThruWidth 36
implicit.feed.thru.range 0.25
do.global.route
do.global.route.cell.swaps
```

For instance, this specifies to add feedthroughs if necessary.

Finally the .net file specifies information about the nets to be routed. The following is an example:

```
allnets HVweights 2.5 1.0
```

This specifies that all nets are to be routed with an equal weighting of 2.5 for horizontal and 1 for vertical routes.

TimberWolf returns the following files

- The .pl1 and .pl2 files which describe the placement of modules.
- The .pin file, which describes the segment list of routes.
- The .twf file.
- The .out file, which is a summary of the program execution.
- The .sv2 and .sav files, which allow restart of the program.

For example, a portion of .pl2 file example appears as follows:

```
.
D-REG-MUX-8 0 502  612 952  2 1
.
INVERTER-2 1188 1452  1260 1902  1 2
.
D-REG-MUX-31 1332 502  1944 952  2 1
.
```

A portion of a .pin file appears as follows:

```
.
[212] 3 NOR2-1 A_bottom 702 502 1 1 0
.
STATE<0> 12 AOI21-35 B_top 774 2852 4 -1 0
.
```

Using this information, a channel router may be called to complete the channel routes specified by TimberWolf. With the addition of power feeds on the two ends of the layout, the standard cells and routing blocks may be placed to create the final layout. A typical standard-cell layout in outline form is shown in Fig. 6.39 and Plate 9.

Micron Ex. 1037, p. 473
Micron v. YMTC
IPR2025-00119

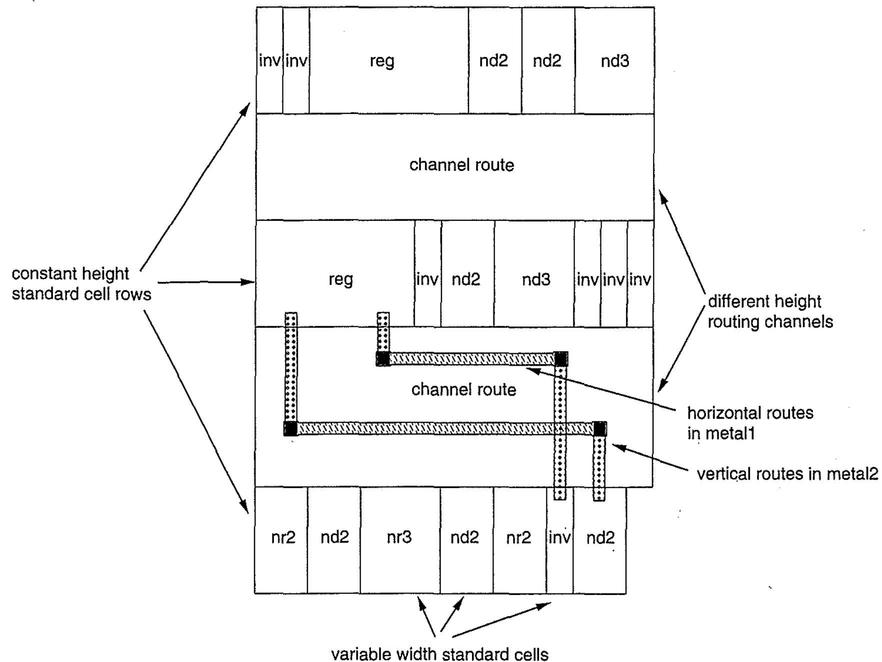


FIGURE 6.39 A typical standard-cell layout style

A large number of mature research, proprietary, and commercial place-and-route systems are available.

6.4.5 Layout Synthesis

The layout of regular structures such as RAMs, ROMs, PLAs, register files, multipliers, and general datapaths may be synthesized by software generators. These programs take a number of parameters as input and automatically create a custom physical layout. Some systems create actual mask layout tuned to a particular process, while others create symbolic layouts that may be compacted to suit a particular technology.[79]

The following is an example of a virtual-grid symbolic description of an inverter from the NS design system. It is specified in Common Lisp.

```
(defaspect-generator ("USER:INVERTER" :VIRTUAL-GRID)
                     (w ratio pw)
;;; transistors
  (part N-CHANNEL-MOSFET :origin (pt 1 3) :WIDTH w)
  (part P-CHANNEL-MOSFET :origin (pt 1 7)
                         :WIDTH (* ratio w))
;;; contacts
  (part VG-TERMINAL :x 0 :y 3
                        :CONNECTED-LAYERS '(N-DIFF METAL))
```

Micron Ex. 1037, p. 474
Micron v. YMTC
IPR2025-00119

```
  (part VG-TERMINAL :x 2 :y 3
                    :CONNECTED-LAYERS '(N-DIFF METAL))
  (part VG-TERMINAL :x 3 :y 5
                    :CONNECTED-LAYERS '(METAL POLY))
  (part VG-TERMINAL :x 0 :y 7
                    :CONNECTED-LAYERS '(P-DIFF METAL))
  (part VG-TERMINAL :x 2 :y 7
                    :CONNECTED-LAYERS '(P-DIFF METAL))
;;; wires
  (part VG-LOG :from (pt 0 10) :to (pt 4 10)
                    :LAYER 'METAL :WIDTH pw)
  (part VG-LOG :from (pt 0 0) :to (pt 4 0)
                    :LAYER 'METAL :WIDTH pw)
  (part VG-LOG :from (pt 0 0) :to (pt 0 3) :LAYER 'METAL)
  (part VG-LOG :from (pt 0 7) :to (pt 0 10) :LAYER 'METAL)
  (part VG-LOG :from (pt 2 5) :to (pt 2 3) :LAYER 'METAL)
  (part VG-LOG :from (pt 1 0) :to (pt 1 10) :LAYER 'POLY)
  (part VG-LOG :from (pt 3 0) :to (pt 3 10) :LAYER 'POLY)
  (part VG-LOG :from (pt 2 7) :to (pt 2 5) :LAYER 'METAL)
  )
```

The first two statements specify transistors. The next four statements specify inter-layer contacts (the type is specified by the list of layers following the :CONNECTED-LAYERS keyword). The final eight statements specify wires with position keywords (:from :to), a size keyword (:WIDTH), and the :LAYER keyword, specifying the layer on which the wire is routed.

In this example, the width of the n- and p-transistors has been specified in terms of the variables w and ratio. The keyword :WIDTH passes this to the transistor generator. In addition, the power bus width has been specified in terms of pw. Figure 6.40 illustrates a few instances of this layout generator.

At a higher level, the following is the top-level call to a PLA generator (see Chapter 8):

```
(defaspect-generator ("PLA:PLA" :VIRTUAL-GRID) (pla-
                          filename)
  (let* ((inputs (get-number-of-inputs pla-filename))
         (outputs (get-number-of-outputs pla-filename))
         (cells
          (list
          `("PLA-EDGE" :inputs ,inputs :outputs ,outputs
              :edge :bottom)
          `("PLA-MIDDLE" :pla-file ,pla-filename)
          `("PLA-EDGE"  :inputs ,inputs :outputs ,outputs
              :edge :top))))
     (vertically-abut cells)
     (import-all-ports)))
```

This hierarchically calculates some parameters (inputs,outputs) from the file specifying the PLA and calls some other generators (PLA-EDGE and

Micron Ex. 1037, p. 475
Micron v. YMTC
IPR2025-00119

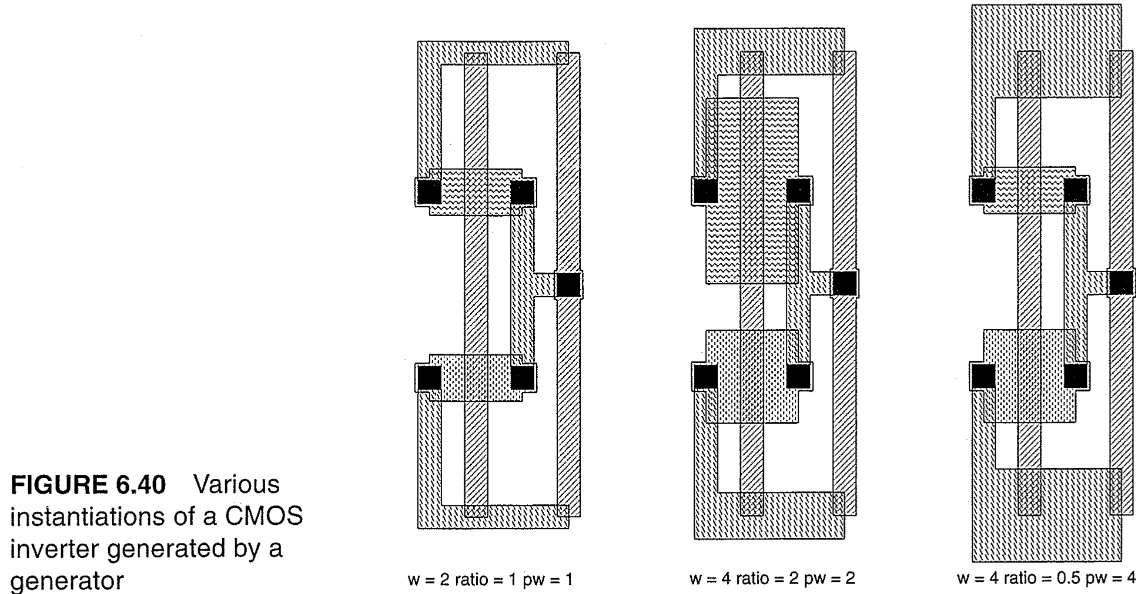


FIGURE 6.40 Various instantiations of a CMOS inverter generated by a generator

PLA-MIDDLE) and then vertically abuts these cells. Following this, the ports are imported from the lower level of the design. As an example of the next level down, the generator that performs the PLA-MIDDLE function is reproduced below:

```
(defaspect-generator ("PLA:PLA-MIDDLE" :VIRTUAL-GRID)
                     (pla-filename)
  (let ((cells (list
            `("PLA-LEFT" :pla-file ,pla-filename)
            `("AND-PLANE" :pla-file ,pla-filename)
            `("AND-OR-JOIN" :pla-file ,pla-filename)
            `("OR-PLANE" :pla-file ,pla-filename)
            `("PLA-RIGHT" :pla-file ,pla-filename))))
  (horizontally-abut cells)
  (import-all-ports)))
```

This horizontally abuts the PLA-LEFT, AND-PLANE, AND-OR-JOIN, OR-PLANE, and PLA-RIGHT cells. Finally at the AND and OR plane level, the generator places transistors according to the PLA personality matrix.

It may be seen that with the combination of symbolic layout, a powerful language and a good CAD substrate, powerful layout generators may be created with minimum effort.

As opposed to the creation of random logic, which the previous section illustrated, layout generators are used for regular arrays or places where a simple algorithm can specify the layout.

Micron Ex. 1037, p. 476
Micron v. YMTC
IPR2025-00119

6.5 Design-capture Tools

6.5.1 HDL Design

The behavior and/or structure of a system may be captured in a Hardware Description Language. There are a wide variety of proprietary, commercial, and public domain languages including those specifically designated hardware description languages (HDLs), such as VHDL, ELLA, Verilog®, and modified high-level languages, such as C, Pascal, and Lisp. Languages like VHDL allow for the capture of both structure and behavior. For example, Chapter 1 used an example of a structural design coded in Verilog®, while this chapter used an RTL VHDL description.

The popular standard HDLs differ from high-level languages by catering for hardware notions such as bit vectors, signals, and time within the native language. This is reflected in the syntax of the language and the underlying runtime operating support, which includes compilers, debuggers, and simulators. In common with high-level languages, HDLs usually provide all of the elements of modern computer languages—structure, parametization, conditionals, looping, and hierarchy.

6.5.2 Schematic Design

The traditional method of capturing a digital system design is via an interactive schematic editor. Actually, preferences have cycled from textual netlists (when graphics hardware was expensive) to interactive graphic editors to textual HDLs. Many design systems allow a free mix of code and diagrams so that designers can choose. In general, diagrams are more quickly understood ("a picture is worth a thousand words"), but HDLs are more easily modified.

Schematic editors provide a means to draw and connect components. A collection of components may be collected into a module for which an icon may be defined. The icon is a diagram that stands for the collection of components within the module. The shape might suggest the function of the module, while the I/O connections of the module are represented by stubs with signal names. This icon may then be used in another module, and so on, hierarchically, throughout the design. Figure 6.41 shows a typical schematic for a module and its schematic icon.

Primarily, schematic editors are menu-based graphic editors with operations such as:

- Creating, selecting, and deleting parts by pointing or area inclusion.
- Changing the graphic view by panning, zooming, or other means.

Micron Ex. 1037, p. 477
Micron v. YMTC
IPR2025-00119

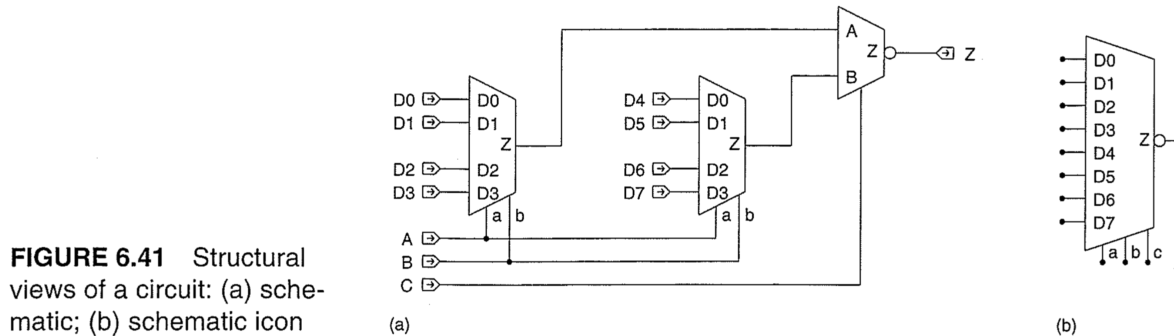


FIGURE 6.41 Structural views of a circuit: (a) schematic; (b) schematic icon

To a basic graphic editor, operations are added that pertain to the electrical nature of the schematic, such as:

- Selecting an electrical node and interrogating it for state, connections, capacitance, etc.
- Running an attached simulator or other electrical network-based tools.

6.5.3 Layout Design

Layout too can be captured via code (in the case of generators) or interactive graphics editors. However, to maintain one's sanity, a good color editor is a strong requirement if substantial layout editing is to be performed. Layout editors, like schematic editors, are based on drawing editors (for instance see Rubin[80]). Differences occur in the way color is treated and sometimes in the way detail is thresholded (although in advanced design systems one editor is usually used for all diagram editing[81]). Because there is usually a large amount of data present, various means of turning off detail are required to alleviate long redraw times. A layout editor might interface to a Design Rule Checking program to allow interactive checking of DRC errors, and to a layout-extraction program to examine circuit-connectivity issues.

6.5.4 Floorplanning

Floorplanning[82,83] is the exercise of arranging blocks of layout within a chip to minimize area or maximize speed. The latter is increasingly the main reason for performing this activity. Floorplan editors provide graphical feedback about the size and placement of modules without showing internal layout details. In addition, the editors show connectivity information between modules in the form of "rat's-nest" wiring diagrams, where the connected ports of modules are connected by straight lines. These kinds of diagrams indicate the relative density of wiring and whether, for instance, ports line up between adjacent modules. Figure 6.42 shows a simple example.

Micron Ex. 1037, p. 478
Micron v. YMTC
IPR2025-00119

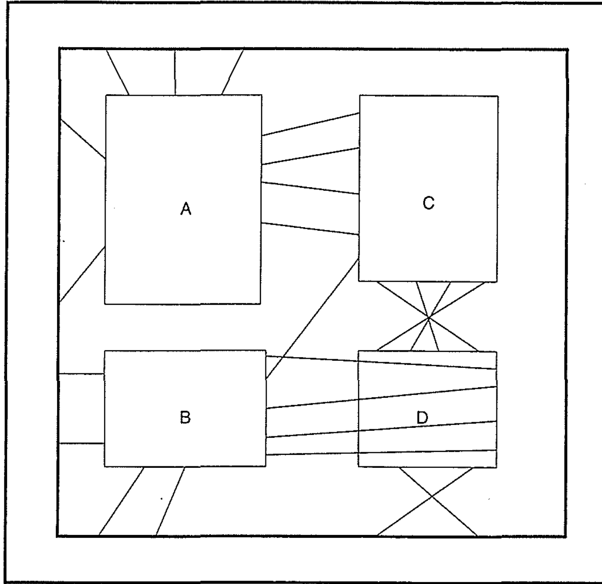


FIGURE 6.42 A floorplan example

This shows that module *D* should be flipped around the *Y* axis to improve the routing. Some editors provide shaded color displays of routing density that allows designers to re-place and "rip-up-and-reroute" congested areas of the chip. While floorplanning may be done automatically, many times a much better job can be done manually. Usually, the task is not that time consuming, given the right interactive tools and a knack for doing it.

6.5.5 Chip Composition

Similar to structural synthesis, chip composition, or "block-place-and-route," is the term that is applied to wiring the top level modules in a design. At this point a good placement of modules is assumed. The task consists mainly of routing modules together and then placing a pad ring around the completed chip core. Usually there is a routing strategy that is followed. For instance, the technique of binary composition,[84] has been widely used. Here modules are combined alternately in horizontal and vertical strips from the bottom up until the complete chip is routed. Figure 6.43 illustrates an example that shows the progression of steps of grouping modules and adding routing channels. Figure 6.43(a) shows the unrouted, relatively placed modules. Figure 6.43(b) shows a horizontal composition where, for instance, *A* and *B* are routed together by routing cell *AB*. A vertical composition step is shown in Fig. 6.43(c), where all modules are now connected. Routing block *DEF* routes module *D* and the composed module (*E,EF,F*). The advantage of this

Micron Ex. 1037, p. 479
Micron v. YMTC
IPR2025-00119

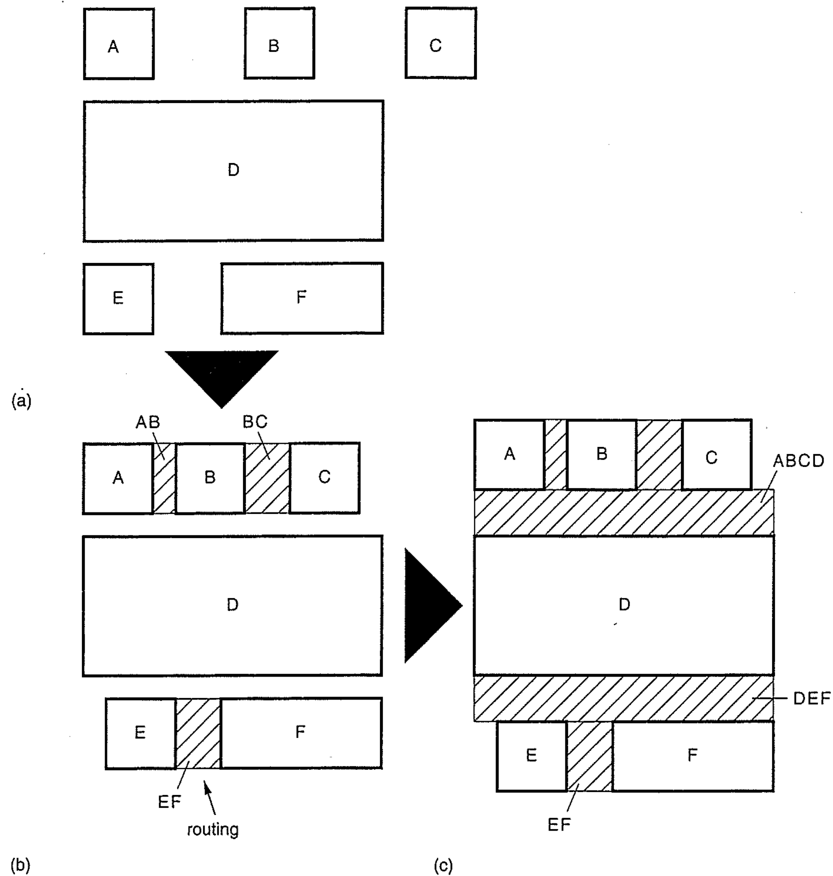


FIGURE 6.43 Binary composition for chip routing

approach is that only channel routes need to be routed, which is a well-solved problem.

6.6 Design Verification Tools

Figure 6.44 shows a conventional flow through a set of design tools to produce a working CMOS chip from a functional specification. Depending on the tools, some steps may be automatic and hidden from the designer but usually are performed by some agent. The design process is commenced with a clock-cycle-accurate functional specification (say, in a high-level language such as C). This is used to verify that the system performs as required. This is translated to a structural RTL or logic description. If done manually, the functionality of the two descriptions has to be proved isomorphic. This is done by applying a stimulus to the functional description and to a logic simulation of the RTL description and comparing the outputs of both forms on a

Micron Ex. 1037, p. 480
Micron v. YMTC
IPR2025-00119

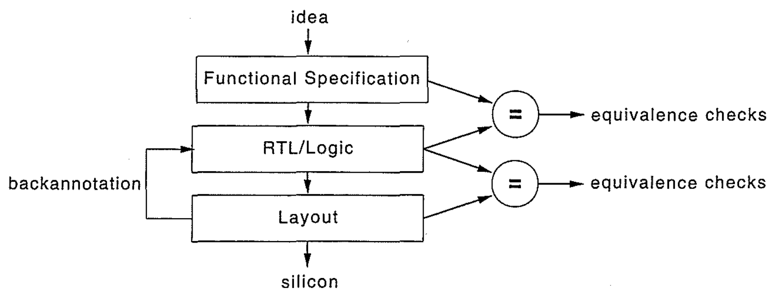


FIGURE 6.44 Design flow through typical CMOS VLSI tools

clock-cycle by clock-cycle basis. RTL simulations may be done with the actual clock timing by estimating the layout loading capacitances. Once the functional equivalence has been satisfied, the structural description is transformed into a physical form (i.e., a layout). Again, this might be automatic for a gate array or standard-cell layout or might be done manually. The problem now is to prove that the layout is a faithful reproduction of the structure of the RTL description (i.e., all signals are routed correctly). In addition, we have to prove that the functionality is still maintained in the temporal domain (timing). This is done by extracting the parasitic routing capacitances introduced by the physical layout and applying them to the RTL simulation model.

Each one of these steps requires a certain set of verification tools. In this section we summarize these tools.

6.6.1 Simulation

Probably the software tool that designers most frequently encounter is some form of simulator that is used to predict and verify the performance of a given circuit. Simulators come in a wide variety depending on the accuracy and speed of simulation required.[85]

6.6.1.1 Circuit-level Simulation

The most detailed and accurate simulation technique is referred to as Circuit Analysis. As the name suggests the simulators operate at the circuit level. Circuit-analysis programs are typified by the SPICE program developed at the University of California at Berkeley[86] and ASTAP developed at IBM.[87] Commercially available versions are typified by the HSPICE program.[88] The basis for this type of program is the solution of the matrix equations relating the circuit voltages, currents, and resistances (or conductances). This type of simulator is characterized by high accuracy but long simulation times. Simulation time is typically proportional to N^m, where N is the number of nonlinear devices in the circuit, and m is between 1 to 2. This type of

program is used to verify in detail small circuits or to verify the simulation results of faster but less accurate simulators, such as timing simulators. It is unrealistic to use this type of program for the verification of large VLSI chips.

Circuit simulators used to verify performance of CMOS circuits should not be assumed to accurately predict the performance of designs. There are three basic sources of error. These are as follows:

- Inaccuracies in the MOS model parameters.
- The use of an inappropriate MOS model.
- Inaccuracies in parasitic capacitances and resistances.

Usually, contemporary circuit simulators related to SPICE provide different levels of modeling (specified by the LEVEL parameter). Simple models are optimized for speed of processing, while more complex models are used for more accurate simulation. If possible one should verify actual transistors from known process corners against the DC characteristics predicted by the simulator. A good practice is to include test transistors of both polarities with various widths with the lengths used in the design (usually the minimum and slightly longer for I/O transistors). Because processes are shrunk, the models used by a simulator may no longer be able to accurately predict the performance of the transistors.

Assuming that one has verified the DC performance of the transistors, the AC performance can now be in question. A significant source of error in predicting performance can be the parasitic capacitances that are applied to nodes in the circuit description used in the circuit-analysis program. The gate capacitance is part of the MOS model and should be subtracted from the total capacitance predicted by a layout-extraction program. One should check how drain and source capacitances are added to transistors—often they are added as diodes or as part of the MOS model. In this case they should not be added as stray capacitance on the node. A good practice is to create a check layout with known areas and peripheries on each layer and then check the SPICE deck produced by any extraction program. The bottom line is to be aware of the means of process calibration (i.e., that someone is responsible for it).

6.6.1.2 Timing Simulation

It is possible to simplify the general circuit analysis approach used above to allow simple nonmatrix calculations to be employed to solve for circuit behavior. This usually involves making some approximations about the circuit. Typical of an early simulator using this approach is the MOTIS simulator.[89] More recent examples may be found in White and Sangiovanni-

Micron Ex. 1037, p. 482
Micron v. YMTC
IPR2025-00119

Vincentelli.[90] The accuracy of such simulators is less than that of SPICE-type simulators, but the execution time is almost two orders of magnitude less.

Implementations may use MOS-model equations to calculate device currents or may use table look-up methods. Calibration of any simulator is advisable, using the techniques described above for circuit simulators. Usually the relative accuracy of such simulators is good; that is, inherently high speed circuits will demonstrate better performance than slower circuits. However, sometimes the absolute accuracy may not be as good as a circuit simulator, especially if no real silicon has been used to check simulation results.

Absolute accuracy is somewhat of a red herring anyway, because process variation, temperature variation, and allowable supply-voltage excursions may vary by a range of three or four to one. The designer is usually trying to predict the slowest the circuit will operate. It is unwise to do this with no margin unless the whole design system is known to accurately predict worst-case performance. Designers generally allow a 10–20% margin in assessing speeds.

6.6.1.3 Logic-level Simulation

Many simulators have evolved to deal with simulation at the logic level. They use primitive models such as NOT, AND, OR, NAND, and NOR gates. Some operate in a "unit delay" mode, where every gate is assumed to have a delay of one time-unit. This type of simulator can be highly optimized for execution speed. Alternatively, timing parameters may be assigned to the logic models based on prior circuit simulation and known circuit parasitics. Because all logic circuits are rarely active simultaneously, logic events may be scheduled on a queue. This means that the state of the network is evaluated on an event-driven basis, rather than on a timing-substep basis, as are most of the implementations of the two previous simulators.

Timing is normally specified in terms of an inertial delay and a load dependent delay for the appropriate edge transitions, as follows:

$$T_{gate} = T_{intrinsic} + C_{load} \times T_{load}, \qquad \textbf{(6.1)}$$

where

T_{gate} = the delay of the gate

$T_{intrinsic}$ = the intrinsic gate delay (no load)

C_{load} = the actual load in some units (i.e., *pF* or # normalized gates)

T_{load} = the delay per load in some units (i.e., *ns/pF* or *ns/#normalized gates*)

Micron Ex. 1037, p. 483
Micron v. YMTC
IPR2025-00119

(A normalized gate might be the minimum gate load of the smallest inverter in a standard-cell library—all other gate inputs would be characterized in terms of this unit.)

Logic simulators with such timing information are quite accurate for CMOS logic configurations or other circuits where the function has been well characterized at the gate level. Nowadays this characterization can be done automatically by running scripts that perform the circuit simulations and extract the relevant data. Where this capability is not available, a considerable manual simulation effort is required to create a new standard-cell library in a new process.

Logic simulators are adequate for well-characterized CMOS circuits that have regular logic counterparts. They are relatively fast and are thus suitable for large circuits. This has been also aided by hardware engines that compute the simulation algorithm. Early logic simulators were not suitable for circuits with transistors used as transmission gates, such as transmission gate multiplexers, memories, or pass-gate logic. However, recent logic simulators do deal with transistor circuits in a limited manner.

6.6.1.4 Switch-level Simulation

Switch-level simulators merge logic-simulator techniques with some circuit-simulation techniques by modeling transistors as switches. RSIM[91] is an example of a switch-level simulator with timing. CMOS gates are modeled as either pull-up or pull-down structures, for which RSIM dynamically calculates a resistance to power or ground. This resistance is used with the output capacitance of the gate to predict rise or fall times.

Switch-level simulators alleviate the need for circuit analysis calibration of CMOS gates, but do have some accuracy limitations when evaluating transmission-gate circuits (they are usually overly pessimistic). In addition, some circuit structures present pathological topology cases, which confuse the simulation algorithms (the "tiny XOR" gate used in the transmission-gate adder in Chapter 8 is an example).

If you design at the transistor level, a switch-level simulator provides a first line of defense as far as simulation. One should probably back up any simulations with a reduced set of simulations using a timing simulator.

6.6.1.5 Mixed-mode Simulators

There now exist very good commercial simulators that merge the good points of functional simulation, logic simulation, switch simulation, timing simulation, and circuit simulation. Each circuit block can be simulated in the appropriate mode. For instance a standard-cell logic block might be simulated at the logic level, a memory might be simulated at the functional level, and a phase-locked loop might be simulated at the circuit level. In this way

Micron Ex. 1037, p. 484
Micron v. YMTC
IPR2025-00119

only those circuits requiring detailed simulation expend expensive compute cycles.

6.6.1.6 Summary

A good simulator is crucial to modern CMOS design. The style of simulator determines the level to which one can safely design. With a logic simulator, one can accurately model well characterized gates and functional blocks. A timing simulator allows design down to the transistor level for most digital circuits and some limited-accuracy analog circuits. Finally, a circuit-analysis program provides enough accuracy (when calibrated to a process) for the most complicated analog circuitry. The simulation times and therefore the amount of circuitry that may be simulated with each kind of simulator varies widely. Logic simulators (particularly unit-delay) are of use at the system level. Timing simulators are useful for modules into the 100–100K transistor range and have been used for 1M+ transistor circuits for a few hundred vectors. Circuit simulators are useful for 10 to 1,000–transistor complexities for short simulation periods. Modern mixed-mode simulators allow a trade-off in simulation accuracy and time of simulation.

6.6.2 Timing Verifiers

Classically, designers simulated with unit-delay simulators to verify functionality. Then they ran simulations with delays to check for timing problems. The detection of such problems is pattern dependent. In other words, if the critical timing vector is not exercised, the critical path will not be found. A timing verifier takes a different approach to temporal verification. Here, the delays through all paths in a circuit are evaluated in a pattern-independent manner and the user is provided with information about these delays. CMOS verifiers in common with simulators may work at the gate or the transistor level.[92] The circuit to be analyzed is first statically examined to determine the direction of signal flow in all transistors. This is necessary to evaluate only those delays that will be critical in actual circuit operation. A recent example of this type of analyzer is the Pearl program.[93] Each transistor is examined and the direction of signal flow is calculated using nine rules. These rules may be determined from:

- Circuit-design methodology rules.
- Electrical rules.
- User-supplied rules.

The Pearl program calculates an RC delay for each node using RSIM. These are then evaluated in a breadth-first manner. Delay paths are qualified by appropriate clocks.

Micron Ex. 1037, p. 485
Micron v. YMTC
IPR2025-00119

A timing analyzer implemented at the transistor level can provide a designer with rapid feedback about critical paths. Combined with a switch-level simulator for rapid global functional simulation, a timing simulator for detailed module verification, and a circuit-analysis program for critical-path evaluation, the timing analyzer completes a set of powerful verification tools. Timing analyzers implemented at the gate level allow the same quality of design down to the gate level, which is sufficient for a wide range of CMOS systems.

Pitfalls of timing analyzers are false paths and sneak paths. False paths can occur because the timing analyzer does not know how the circuit is used. For instance, that a bus is only used to read or write during a cycle and not for both. False paths are dealt with by blocking them as they are recognized. Sneak paths are paths that for some reason the timing analyzer does not recognize. These can occur in complicated clocking schemes that may be beyond aging timing analyzers. For this reason it may be prudent to timing simulate circuits as a backup unless you are confident that your timing analyzer catches everything. (Many have believed the latter point, only to be ushered back to reality by the outcome of the silicon.)

6.6.3 Network Isomorphism

An electrical network may be represented by a graph where the vertices of the graph are devices such as MOS transistors, bipolar transistors, diodes, resistors, and capacitors. The arcs are the connections between devices. These are the electrical nodes in the circuit. This graph may be in turn represented by some data structure that may be accessed by a variety of software routines interested in the electrical connectivity properties of the circuit. Two electrical circuits are identical if the graphs representing them are isomorphic; that is, each graph has the same number of devices and for every device in one circuit there is a matching device in the other circuit. The matching devices have identical properties such as:

- Transistor width and length.
- Resistance value.
- The number of connections on each terminal (i.e., gate, drain, source).

Each node in one circuit has a matching node in the other circuit. They have identical properties such as:

- The same number of source and drains attached to them.
- And the same number of gates (MOS gate).

Network isomorphism is used to prove that two networks are equivalent and therefore should function equivalently. This is used most often to prove

Micron Ex. 1037, p. 486
Micron v. YMTC
IPR2025-00119

that a layout is equivalent to a network extracted from a schematic schematic or HDL structural netlist. Other uses include proving that two schematics or two layouts are equivalent.

The process of comparing two networks is commonly called "netlist comparison," "network isomorphism," or LVS (layout versus schematic).

Electrical networks may use subnetworks as devices. For instance, in a chip layout standard-cell blocks may be represented by bounding boxes (for a vendor's proprietary library). The layout extract operation then extracts only the routing. This is compared with to the network obtained by expanding the structural description down to the level of gates (but not transistors). Frequently the notion of "logical equivalency" is used. This allows a layout-design system to swap the order of signals on series transistors in logic gates with respect to the structural specification for layout convenience. While this is fine for logic circuits, some problems can occur if it is used in high-performance and mixed-signal circuits. Consider an analog-bias circuit consisting of series transistors where the order of the transistors dictates the behavior—these cannot be swapped.

6.6.4 Netlist Comparison

If a schematic or circuit description is entered to define an IC, at some stage a physical layout is generated. This may be completed automatically, as in the case of a gate array or place-and-route standard-cell system. Alternatively, the physical layout may have a manual component. Ideally, the signal names between parallel representations would be the same, allowing easy comparison between desired and actual circuit by matching node names and the number and type of components connected to each node in the schematic and the layout. In reality, signal names are often omitted from internal nodes in a circuit (especially in layouts) and only applied to I/O ports. Thus there is the problem of comparing two graphs that are labeled in a limited manner. Programs that verify the equivalence or lack thereof of two unnamed circuit graphs are thus needed.

Typical of a program that performs this function is GEMINI.[94] Signatures are calculated for each transistor in the test and reference circuit. Signatures include:

- Fan-in.
- Fan-out.
- Transistor type.
- Bound nets connected to the transistor.

Test and reference circuits are then repeatedly checked to correlate transistors. Discrepancies are either indicated interactively or by a listing of the matched and unmatched nodes.

Micron Ex. 1037, p. 487
Micron v. YMTC
IPR2025-00119

6.6.5 Layout Extraction

Layout extractors examine the interrelationship of mask layers to infer the existence of transistors and other components. They are related to design rule checkers (Section 6.6.7). Various approaches have been implemented to approach this problem.[95,96] Commonly, parasitic capacitances and resistances are reported in addition to transistor connectivity. Algorithms commonly use geometric-shape intersections to recognize active devices (see Chapter 3). The need for such tools by the system designer will decrease as higher-level design techniques provide "correct-by-construction" modules. However, some form of layout extraction is usually done to create data for the back annotation step described in the next section.

6.6.6 Back-Annotation

Once a layout has been constructed and there is isomorphism between the schematic network and the layout network, one can correlate extracted capacitances from the layout with the schematic and perform simulation or timing analysis to verify performance. This is done by moving the capacitance that appears on a layout node to the corresponding schematic node while accounting for existing capacitance on the schematic node. For instance, the schematic may already have the source-drain and gate load due to the gates connected to the node and only the routing capacitance is required to be added. This operation is known as "back-annotation" (Fig. 6.44).

6.6.7 Design-rule Verification

If mask design is completed manually (and even automatically), it is necessary to verify that the layout conforms to the geometric design rules. This is achieved with a design-rule checker. Many variations exist, but typical approaches are found in Szymanski and Van Wyk,[97] Baker and Terman,[98] and Baird[99] (see also Chapter 3).

Hierarchical design-rule checkers are necessary for large circuits.[100] These design-rule checkers use the hierarchical nature of a design to reduce the number of cells that have to be individually checked.

6.6.8 Pattern Generation

Pattern generation is the last step in the sequence that starts at the architecture for a chip and ends with a database suitable for manufacture. It is the operation of creating the data that is used for maskmaking. Over the years, the format of this data has changed as the methods of generating masks have changed. Originally, the data drove flatbed plotters that cut Rubylith® (a red

Micron Ex. 1037, p. 488
Micron v. YMTC
IPR2025-00119

plastic "mask" layer backed by a clear Mylar® plastic backing). Nowadays, most semiconductor operations use electron-beam–generated masks (i.e., generated by exposing a resist-coated metal film with a focused electron beam). These machines expose the masks in a raster-scan style similar to a television.

A common format is the Electron Beam Exposure System, EBES format.[101] Data is composed of rectangles, parallelograms, or trapezoids. Given a layout captured in a design system, the following steps must be completed to create an EBES file:

- Combine layers to form required mask (i.e., all n^+ and vddn regions for an nplus mask).
- Size-resulting data (i.e., shrink or bloat to account for processing effects such as under-etching or sideways diffusion).
- Canonicalize resulting geometry in terms of base figures (i.e., rectangles).
- Sort the resulting shapes in scanline order.
- Determine polarity of mask (i.e., dark field or light field).
- Output data in suitable format.

Because this is the last step in the design process and because it is hard to detect defects on the masks, manufacturers frequently pattern two or more die patterns on a single-mask reticle and then use differencing techniques to detect differences between pairs of like die patterns to detect mask defects.

6.7 Design Economics

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

In this study we will concentrate on the cost of a single IC, although one should consider the overall system when making such decisions. System level issues such as packaging and power dissipation may affect the cost of an IC.[102]

The selling price of an integrated circuit may be given by

$$S_{total} = \frac{C_{total}}{1-m}, \qquad (6.2)$$

Micron Ex. 1037, p. 489
Micron v. YMTC
IPR2025-00119

where

C_{total} = the manufacturing cost of a single IC to the vendor

m = the desired profit margin.

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

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

- Nonrecurring costs (NREs).
- Recurring costs.
- Fixed Costs.

6.7.1 Nonrecurring Engineering Costs (NREs)

The nonrecurring costs are those costs that are spent once during the design of an integrated circuit. They include

- The engineering design cost.
- The prototype manufacturing cost.

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

$$F_{total} = E_{total} + P_{total}, \tag{6.3}$$

where

E_{total} = the engineering cost

P_{total} = the prototype manufacturing cost.

Normally the recurring costs are viewed as an investment for which there is a required rate of return. For instance, if $100K is invested in NRE for a chip then $1M might have to be generated as profit for a rate of return of 10.

6.7.1.1 Engineering Costs

The costs of designing the IC (E_{total}) hopefully happen only once during the chip design process. The costs include:

- Personnel costs.
- Support costs.

Micron Ex. 1037, p. 490
Micron v. YMTC
IPR2025-00119

The personnel costs might include the labor for:

- Architectural design.
- Logic capture.
- Simulation for functionality.
- Layout of modules and chip.
- Timing verification.
- DRC and tapeout procedures.
- Test generation.

The support costs, amortized over the life of the equipment for the length of the design project, include:

- Computer costs.
- CAD program costs.
- Education or reeducation costs.

6.7.1.2 Prototype Manufacturing Costs

These costs (P_{total}) are the fixed cost to get the first ICs from the vendor. They include:

- The mask cost.
- Test fixture costs.
- Package tooling.

The photo-mask cost is proportional to the number of steps used in the process. Mask costs increase as the process dimensions are reduced, so while newer, smaller processes generally have increased mask costs, masks on the metalization layers can be less expensive than the lower layers. A mask can currently cost between $500 and $1500.

A test fixture consists of a printed wiring board-probe assembly to probe individual die at the wafer level and the interface to the tester. Costs range from $1000 to $5000 depending on the complexity of the interface electronics.

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

Micron Ex. 1037, p. 491
Micron v. YMTC
IPR2025-00119

6.7.2 Recurring Costs

Once the development cost of an IC has been determined, the IC manufacturer will arrive at a price for the specific IC. This includes a recurring cost; that is, one that recurs every time an IC is sold.

The IC manufacturer will determine a part price for an IC based on the cost to produce that IC and a profit margin. The margin can fall as the revenue increases. An expression for the cost to process an IC follows.

The total cost is

$$C_{total} = C_{process} + C_{package} + C_{test}, \tag{6.4}$$

where

$C_{package}$ = package cost

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

$$C_{process} = \frac{W + P}{NY_w Y_{pa} Y_{ft}}, \tag{6.5}$$

where

W = wafer cost

P = processing cost

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

Y_w = die yield per wafer

Y_{pa} = packaging yield

Y_{ft} = final test yield.

The wafer yield, Y_w, was dealt with in Chapter 4. The packaging yield is the percentage of successfully diced, bonded and packaged parts. The final test yield is the percentage of packaged parts that pass a final packaged part test sequence.

6.7.3 Fixed Costs

Once a chip has been designed and put into manufacture, the cost to support that chip from an engineering viewpoint may have a few sources. In order for the part to be effectively used, Data Sheets describing the characteristics of the IC have to be written. A data sheet is probably always required, even

Micron Ex. 1037, p. 492
Micron v. YMTC
IPR2025-00119

for application specific ICs that are not sold outside the company that developed them. From time to time Application Notes describing how to use the IC may be needed. In addition specific application support may have to be provided to help particular users. This is particularly true for ASICs, where the designer usually becomes the walking, talking data sheet and application note. Another ongoing task may be failure analysis if the part is in high volume and you desire to increase the yield.

Finally there is what is called "the cost of sales," which is the marketing, sales force, and overhead costs associated with selling each IC. In a captive situation this might be zero.

6.7.4 Schedule

At the outset of a system-design project involving newly designed ICs it is important to be able to estimate the design cost and design time for that system. Estimating the cost can guide the designer as to the method by which the ICs will be designed. Estimating the time is essential to be able to select a strategy by which the ICs will be available in the right timescale and at the right price. This second task (estimating schedules) is usually the least well specified and requires some experience to accurately predict design timescales.

If we assume that for a given IC size $C_{process}$ is constant, the variables left in determining the return on investment of an IC are, E_{total}, the engineering design cost, P_{total} the prototype-manufacturing cost. P_{total} depends on the way in which the IC is implemented. We examined a variety of strategies for the design of CMOS systems in Section 6.2. The fixed costs of prototyping P_{total} are relatively constant, given an implementation technology. The engineering costs depend on the complexity of the chip and the design strategy. For this reason, it is important to be able to estimate a schedule for the design of an IC and then manage the available resources to bring the project to a successful conclusion.

Studies on schedule management for ICs have been carried out by analyzing many IC design projects implemented in a variety of ways.[104] These show that schedule is only a function of personpower, that is, the number of people working on the project. The study showed that below 30 weeks, schedule is proportional to personpower, while beyond about 30 weeks, the schedules become proportional to the cube root of the personpower. Fey and Paraskevopoulos[102] suggest a number of methods for increasing productivity, thereby improving schedules. They include the following:

- Using a high-productivity design method.
- Improving the productivity of a given technique.
- Decreasing the complexity of the design task by partitioning.

Micron Ex. 1037, p. 493
Micron v. YMTC
IPR2025-00119

A range of various design methods were examined in Section 6.3 which form the basis for achieving some of these goals.

As a final point one should note that increasing the personpower is a poor way of improving a schedule and likely will have the opposite effect.

6.7.5 Personpower

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

- Architectural design.
- Logic capture.
- Simulation for functionality.
- Layout of modules and chip.
- Timing verification.
- DRC and tapeout procedures (ERC, LVS, MEBES).
- Test generation.

If we take each of these activities and apply a productivity figure for a given complexity of design, we may have the basis for the manpower, or personpower, required to complete the project.

Fey has completed productivity studies for custom-chip designs[103, 104] and gate-array designs.[105]

6.7.6 An Example—Gate-array Productivity

Fey's productivity models for gate arrays lead to the following empirically determined equation:

$$P = 16.2\, mG^{0.6} \quad 0.5 < G < 25, \tag{6.6}$$

where

P = the productivity in gates/person-day

G = the number of gates in thousands

$m = (0.61^{I})(0.86^{U})(0.64^{R})(1.17^{D})$,

where

$$I = \text{the adjusted I/O} = \frac{(inputs + outputs + bidirects)^{0.5}}{K} \tag{6.7}$$

Micron Ex. 1037, p. 494
Micron v. YMTC
IPR2025-00119

K = the number of gates
R = the complexity (from 1 (lowest) to 5)
U = the number of gates used = max(0, %gates used – 90%)
D = the design experience = number of previous designs completed by designer

As with automobile advertising, these formulae are for comparison purposes only; your mileage may vary. Other variables that were studied included the number of test vectors per gate, the quality of the design specification and the year of the design.

By normalizing the productivity, we obtain

$$P_N = \frac{P}{m} = 16.2G^{0.6} \quad 0.5 < G < 25 \qquad \textbf{(6.8)}$$

where P_N is the normalized productivity in gates per person-day.

The personpower (M) may be calculated from the productivity by

$$M = \frac{200G}{P_B} \qquad 0.5 < G < 25, \qquad \textbf{(6.9)}$$

where

$$P_B = 17.1G^{0.61} \text{ (Eq. 6.8 fitted to experimental data),}$$

Thus

$$M = 11.7G^{0.39} \qquad \textbf{(6.10)}$$

By evaluating M one may then estimate the schedule (T), using the following equation:

$$T = M \qquad M < 29 \qquad \textbf{(6.11)}$$

$$T = 9.1M^{0.34} \quad M \geq 29 \qquad \textbf{(6.12)}$$

Thus for a 1000-gate design these equations would suggest that it takes 9 weeks while a 10K-gate design would take 29 weeks. These equations are included as an analytical guide for estimating schedules. They represent the result of one piece of research work aimed at quantifying design productivity. Nothing replaces experience when it comes to estimating the real thing (and even then that does not always help!).

Micron Ex. 1037, p. 495
Micron v. YMTC
IPR2025-00119

The general outcome of this research suggests keeping design schedules below 6–7 months. Increasing productivity can increase the number of gates that can be designed in this time. With HDL based synthesis, this should reach 50–100K gates. Other design methods should lead to higher productivities.

6.8 Data Sheets

A data sheet for an IC describes what it does and outlines the specifications for making the IC work in a system. These specifications would include power-supply voltages, currents, input setup times, output-delay times, and clock-cycle times. Also included would be pin loadings and package and pinning details. While commercially produced chips are accompanied by data sheets (and this is a good place to look for examples), chips produced in small volumes internally in an organization may often be introduced into the world without the advantage of a data sheet.

A good habit to acquire is that of compiling a data sheet for any chip you might design. A data sheet is the interface between the chip designer and the board-level designer. In particular, it is good practice and mandatory in industry to compile the data sheet for the chip and give it to the ultimate customer before it is fabricated. This prevents many undesirable scenarios that can arise when perfectly designed chip meets perfectly designed system and creates product nightmare. In this section an outline of a typical data sheet will be reviewed by way of example.

6.8.1 The Summary

A summary of the chip includes the following details:

- The designation and descriptive name of the chip (i.e., ABC1478—FIR Filter Chip).
- A short description of what the chip does.
- A features list (optional for an internal product—but good for your ego).
- A very high level block diagram of the chip function.

This serves to orient the user to the chip and the function it performs.

6.8.2 Pinout

The pinout section should contain a description of the following pin attributes:

- The name of the pin.

Micron Ex. 1037, p. 496
Micron v. YMTC
IPR2025-00119

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

This documents the external interface of the chip.

6.8.3 Description of Operation

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

6.8.4 DC Specifications

The absolute maximum ratings should be stated for the following:

- Supply voltage.
- Pin voltages.
- Junction temperature.

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

- The V_{IL} and V_{IH} for each input.
- The V_{OL} and V_{OH} for each output (at a given drive level).
- The input loading for each input.
- The output drive capability of each output.
- Quiescent current.
- Leakage current.
- Power-down current (if applicable).
- Any other relevant voltages and currents.

This section communicates the power dissipation and required voltages for the chip to correctly operate.

6.8.5 AC Specifications

The following timing specifications should be presented:

- Setup and hold times on all inputs (slowest and fastest).

Micron Ex. 1037, p. 497
Micron v. YMTC
IPR2025-00119

- Clock (and all other relevant inputs) to output delay times (slowest and fastest).
- Other critical timing, such as minimum pulse widths,.

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

6.8.6 Package Diagram

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

6.9 Summary

This chapter has covered a broad spectrum of design issues that may be encountered when designing CMOS chips. The structured design strategies that were introduced early in the chapter are useful for any kind of CMOS-chip design method. A range of implementation options was given to give the reader an appreciation for the wide spectrum of solutions that are available today. In addition a summary of the design styles was given. Increasingly, the level of design is being pushed upward as logic synthesizers are refined, compilers are optimized, and knowledge is captured from libraries of reusable components. You as a designer must keep abreast of such techniques to ensure that you can bring to bear a productivity that results in timely, cost-effective, and reliable silicon that may be shipped after the first manufacturing run.

6.10 Exercises

1. Explain how you would assess the required design-method for a function that has to be performed by a single chip. Draw a decision chart that shows the various questions that have to be answered, and the resulting actions.
2. Explain the following terms with respect to CMOS-chip design: hierarchy, regularity, modularity, and locality. Give an example of each.

Micron Ex. 1037, p. 498
Micron v. YMTC
IPR2025-00119

3. Summarize the differences between a SOG chip and a standard-cell chip. What benefits does each implementation style have?

6.11 References

1. Daniel D. Gajski, *Silicon Compilation,* Reading, Mass.: Addison-Wesley, 1988.
2. D. D. Gajski and R. H. Kuhn, "New VLSI tools," *IEEE Computer,* vol. 16, no. 12, 1983, pp. 11–14.
3. Carver Mead and Lynn Conway, *Introduction to VLSI Systems,* Reading, Mass.: Addison-Wesley, 1980.
4. Irene Buchanan, "Modelling and verification in structured integrated circuit design," Ph.D. thesis, Dept. of Computer Science, University of Edinburgh, Scotland, 1980.
5. *PAL Device Data Book,* Sunnyvale, Calif.: Advanced Micro Devices Inc., 1988.
6. *GAL Data Book,* Hillsboro, Ore.: Lattice Semiconductor Corp., 1990.
7. Clinton Kuo, Mark Weidner, Thomas Toms, Henry Choe, Ko-Min Chang, Ann Harwood, Joseph Jelemensky and Philip Smith, "A 512-kb Flash EEPROM Embedded in a 32-b Microcontroller," *IEEE Journal of Solid State Circuits,* vol. 27, no. 4, Apr. 1992, pp. 574–582.
8. Takaaki Nozaki, Toshiaki Tanaka, Yoshiro Kijiya, Eita Kinoshita, Tatsuo Tsuchiya, and Yutaka Hayashi, "A 1-Mb EEPROM with MONOS memory cell for semiconductor disk application," *IEEE JSSC,* vol. 26, no. 4, Apr. 1991, pp. 497–501.
9. Masaki Momodomi, Tomoharu Tanaka, Yoshihisa Iwata, Yoshiyuki Tanaka, Hideko Oodaira, Yasuo Itoh, Riichiro Shirota, Kazunori Ohuchi and Fujio Masuoka, "A 4-Mb NAND EEPROM with tight programmable V_t distribution," *IEEE JSSC,* vol. 26, no. 4, Apr. 1991, pp. 492–496.
10. *ACT Family Field Programmable Gate Array DATABOOK,* San Jose, Calif.: Actel Corporation, 1990.
11. "Very-High-Speed FPGAs," *pASIC 1 Family Data Book,* Santa Clara, Calif.: QuickLogic Corporation, 1992.
12. Esmat Hamdy, John McCollum, Shih-ou Chen, Steve Chiang, Shafy Eltoukhy, Jim Chang, Ted Speers and Amr Mohsen, "Dielectric-based antifuse for logic and memory ICs," *Proceedings of the International Electron Devices Meeting,* 1988, pp. 786–789, Washington, D.C.
13. Abbas El Gamal, Jonathan Greene, Justin Reyneri, Eric Rogoyski, Khaled A. El-Ayat, and Amr Mohsen, "An architecture for electrically configurable gate arrays," *IEEE JSSC,* vol. 24, no. 2, Apr. 1989, pp. 394–398.
14. Khaled A. El-Ayat, Abbas El Gamal, Richard Guo, John Chang, Ricky K. H. Mak, Frederick Chiu, Esmat Z. Hamdy, John McCollum, and Amr Mohsen, "A CMOS electrically configurable gate array," *IEEE JSSC,* vol. 24, no. 3, Jun. 1989, pp. 752–762.
15. *The Programmable Gate Array Data Book,* San Jose, Calif.: XILINX, Inc., 1990.
16. Thomas Andrew Kean, "Configurable logic: a dynamically programmable cellular architecture and its VLSI implementation," Ph.D. thesis, Department of Computer Science, University of Edinburgh, Scotland, 1989.

Micron Ex. 1037, p. 499
Micron v. YMTC
IPR2025-00119

17. *CAl1024 Datasheet,* Edinburgh, Scotland: Algotrinix Ltd., 1990.
18. J. P. Gray and T. A. Kean, "Configurable hardware: a new paradigm for computation," *Proceedings of the 1989 Decennial Caltech Conference, Pasadena, CA,* Cambridge, Mass.: MIT Press, pp. 1-17.
19. *CLi6000 Series Field-Programmable Gate Arrays,* (data sheet), Sunnyvale, Calif.: Concurrent Logic Inc., 1992.
20. Michiel A. Beunder, Juergen P. Kernhof, and Bernd Hoefflinger, "The CMOS gate forest: an efficient and flexible high-performance ASIC design environment," *IEEE JSSC,* vol. 23, no. 2, Apr. 1988, pp. 387–399.
21. Harry J. M. Veendrick, Dré A. J. M. Van Den Elshout, Dick W. Harberts, and Teus Brand, "An efficient and flexible architecture for high-density gate arrays," *IEEE JSSC,* vol. 25, no. 5, Oct. 1990, pp. 1153–1157.
22. Philippe Duchene and Michel J. Declercq, "A highly flexible sea-of-gates structure for digital and analog applications," *IEEE JSSC,* vol. 24, no. 3, Jun. 1989, pp. 576–584.
23. *1.5 Micron Compacted Array Technology Databook,* Milpitas, Calif.: LSI Logic Corp., 1987.
24. Masatomi Okabe, Yoshihiro Okuno, Takahiko Arakawa, Ichiro Tomioka, Takio Ohno, Tomoyoshu Noda, Masahiro Hatanaka, and Yoichi Kuramitsu, "A 400K-transistor CMOS sea-of-gates array with continuous track allocation," *IEEE JSSC,* vol. 24, no. 5, Oct. 1989, pp. 1280–1286.
25. *1.0-Micron Cell-Based Products Databook,* Milpitas, Calif.: LSI Logic, Feb. 1991.
26. Dennis V. Heinbruch, *CMOS3 Cell Library,* Reading, Mass.: Addison-Wesley, 1988.
27. Curt F. Fey and Demetris E. Paraskevopoulos, "Studies in LSI technology economics IV: models for gate array design productivity," *IEEE JSSC,* vol. SC-24, no. 4, Aug. 1989, pp. 1085–1091.
28. D. Gibson and S. Nance, "SLIC—symbolic layout of integrated circuits," *IEEE Proceedings of the 13th Design Automation Conference,* Jun. 1976, pp. 434–440.
29. D. Clary, R. Kirk, and S. Sapiro," SIDS—a symbolic interactive design system," *IEEE/ACM Proceedings of the 17th Design Automation Conference,* Jun. 1980, Minneapolis, Minnesota, pp. 292–295.
30. R. P. Larson, "Versatile mask generation techniques for customer microelectronic devices," *IEEE/ACM Proceedings of the 15th Design Automation Conference*, Jun. 1978, Las Vegas, Nev., pp. 193–198.
31. A. D. Lopez and H-F. S. Law, "A defense gate matrix layout style for MOS LSI," *IEEE JSSC,* vol. SC-15, no. 4, Aug. 1980, pp. 736–740.
32. C. Piguet, J. Zahnd, A. Stauffer, and M. Bertarionne, "A metal-oriented layout for CMOS logic," *IEEE JSSC*, vol. SC-19, no. 3, Jun. 1984, pp. 425–436.
33. J. Williams, "STICKS—a graphical compiler for high level LSI design," *Proceedings of the National Computer Conference,* May 1978, pp. 289–295.
34. A. Dunlop, "SLIM—the translation of symbolic layouts into mask data," *IEEE/ACM Proceedings of the 17th Design Automation Conference,* Jun. 1980, Minneapolis, Minnesota, pp. 595–602.
35. M. Y. Hsueh and D. O. Pederson, "Computer-aided layout of LSI building blocks," *IEEE Proceedings of the 1979 International Symposium on Circuits and Systems,* Jul. 1979, Tokyo, Japan, pp. 474–477.

Micron Ex. 1037, p. 500
Micron v. YMTC
IPR2025-00119

36. G. Kedem and H. Watanabe, "Graph optimization techniques for IC layout and compaction," *IEEE/ACM Proceedings of the 20th Design Automation Conference,* Jun. 1983, Miami Beach, Fla., pp. 113–120.
37. R. C. Mosteller, "Rest—a leaf cell design system," *Proceedings of IFIP VLSI '81* (J. Gray, ed.), Edinburgh 1981, pp. 163–172.
38. Werner Bonath and Manfred Glesner, "Process-independent 2D-compaction in a symbolic design environment," *Proceedings of IFIP VLSI '89* (G. Musgrave and V. Lauther, eds.), Munich, 1989, pp. 433–443.
39. David G. Boyer, "Symbolic layout compaction review," *IEEE/ACM Proceedings of the 25th Design Automation Conference,* Jun. 1988, Anaheim, Calif., pp. 383–389.
40. W. H. Crocker, C. Y. Lo, and R. Varadarahan, "MACS: a module assembly and compaction system," *Proceedings of the IEEE International Conference on Computer Design,* Nov. 1985, Santa Clara, Calif., pp. 205–208.
41. J. L. Burns and A. R. Newton, "SPARCS: a new constraint-based IC layout symbolic spacer," *Proc. IEEE Custom Integrated Circuits Conference,* May 1986, Rochester, N.Y., pp. 534–539.
42. H. Shin, A. Sangiovanni-Vincentelli, and C. Sequin, "Two-dimensional module compactor based on zone-refining," *Proceedings of the IEEE International Conference on Computer design,* Oct. 1987, Port Chester, N.Y., pp. 201–204.
43. Johan K. J. Van Ginderdeuren, Hugo J. De Man, Bart J. S. De Loore, Hilbradb Vanden Winjingaert, Atoine DeLaruelle, and Guy R. J. Van Den Audenaerde, "A high-quality digital audio filter set designed by silicon compiler CATHEDRAL-I," *IEEE JSSC,* vol. SC-21, no. 6, Dec. 1986, pp. 1067–1075.
44. N. Weste, "Virtual grid symbolic layout," *IEEE/ACM Proc. of the 18th Design Automation Conference*, Nashville, Tenn., Jun. 1981, pp. 225–233.
45. I. Buchanan, "Modelling and verification in structured integrated circuit design," Ph.D. thesis, Dept. of Computer Science, University of Edinburgh, Scotland, 1980.
46. N. Weste, "MULGA—an interactive symbolic layout system for the design of integrated circuits," *Bell System Technical Journal*, vol. 60, no. 6, Jul.-Aug. 1981, pp. 823–858.
47. N. Weste and B. Ackland, "A pragmatic approach to topological symbolic IC design," *IFIP Proc. VLSI '81* (J. Gray, ed.), Edinburgh, Scotland, August 1981, pp. 117–129.
48. James J. Cherry, "CAD programming in an object oriented programming environment," *VLSI CAD Tools and Applications,* (Wolfgang Fichtner and Martin Morf, eds.), Norwell, Mass.: Kluwer Academic Publishers, 1987, Chapter 9.
49. K. Ramachandran, R. R. Cordell, D. F. Daly, D. N. Deutsch and A. F. Kwan, "SYMCELL—a symbolic standard cell system," *IEEE JSSC,* vol. 26, no. 3, Mar. 1991, pp. 449–452.
50. M. C. Revett and P. A. Ivey, "ASTRA—a CAD system to support a structured approach to IC design," *IFIP VLSI '83*, (F. Anceav and E. J. Aas, eds.), 1983, pp. 413–422.
51. Bill Lin and A. Richard Newton, "A circuit disassembly technique for synthesizing symbolic layouts from mask descriptions," *IEEE Transactions on CAD,* vol. 9, no. 9, Sept. 1990, pp. 959–969.
52. Rajeev Jain, Francky Catthoor, Jan VanHoof, Bart J. S. De Loore, Gert Goossens, Nelson F. Goncalvez, Luc. J. M. Claesen, Johan K. J. Van Gindereuren, Joos

Micron Ex. 1037, p. 501
Micron v. YMTC
IPR2025-00119

VanDeWalle, and Hugo J. De Man, "Custom design of a VLSI PCM-FDM transmultiplexer from system specifications to circuit layout using a computer-aided design system," *IEEE JSSC,* vol. SC-21, no. 1, Feb. 1986, pp. 73–85; and Johan K. J. Van Ginderdeuren et al., *op. cit.*

53. J. Rabaey, H. De Man, J. Vanhoof, G. Goossens, and F. Catthor, "Cathedral II: a synthesis system for multiprocessor DSP systems," *Silicon Compilation* (Daniel D. Gajski, ed.), Reading, Mass.: Addison-Wesley, 1988.
54. F. Catthoor and H. De Man, "Application-specific architectural methodologies for high-throughput digital signal and image processing," *IEEE Transactions on Acoustics, Speech and Signal Processing,* vol. 38, Feb. 1990, pp. 339–349.
55. C. Bernard Shung, Rajeev Jain, Ken Rimey, Edward Wang, Mani B. Srivastava, Brian C. Richards, Erik Lettang, S. Khalid Azim, Lars Thon, Paul N. Hilfinger, Jan M. Rabaey, and Robert W. Brodersen, "An integrated CAD system for algorithm-specific IC design," *IEEE Transactions on CAD,* vol. 10, no. 4, Apr. 1991, pp. 447–463.
56. R. K. Brayton, R. Campansano, G. De Micheli, R. H. J. M. Otten, J. van Eijndhoven, "The Yorktown Silicon Compiler system," *Silicon Compilation* (Daniel D. Gajski, ed.), Reading, Mass.: Addison-Wesley, 1988.
57. David R. Coelho, "The VHDL handbook," Norwell, Mass.: Kluwer Academic Press, 1989.
58. Jean-Michel Bergé, Alain Fonkova, Serge Maginot, and Jacques Roilland, "VHDL designer's reference," Norwell, Mass.: Kluwer Academic Press, 1992.
59. Douglas L. Perry, "VHDL," Summitt, Penn.: McGraw-Hill, 1992.
60. Steve Carlson, *Introduction to HDL-Based Design Using VHDL,* Mountain View, Calif.: Synopsys, 1991.
61. Srinivas Devadas, Hi-Keung Ma, Richard Newton, and A. Sangiovanni-Vincentelli, "MUSTANG: state assignment of finite state machines targeting multilevel logic implementations," *IEEE Transactions on CAD,* vol. 27, no. 12, Dec. 1988, pp. 1290–1300.
62. R. K. Brayton, G. D. Hachtel, and A. L. Sangiovanni-Vincentelli, "Multilevel logic synthesis," *Proceedings of the IEEE,* vol. 78, no. 2, Feb. 1990, pp. 264–300.
63. Richard Rudell, "Logic Synthesis," Custom Integrated Circuits Conference '91 Educational Session; Session III: Design Automation (lecture notes).
64. R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. Sangiovanni-Vincentelli, *ESPRESSO-IIC: Logic Minimization Algorithms for VLSI Synthesis, The Netherlands*, Norwell, Mass.: Kluwer Academic, 1984.
65. R. K. Brayton and C. McMullen, "The decomposition and factorization of Boolean expressions," *Proc. IEEE Int. Symposium on Circuits and Systems,* 1982, pp. 49–54.
66. Kurt Keutzer, "DAGON: technology binding and local optimization by DAG matching," *IEEE Proc. 24th DAC,* 1987, pp. 341–347.
67. David Gregory, Karen Bartlett, Aart De Geus, and Gary Hachtel, "Socrates: a system for automatically synthesizing and optimizing combinational logic," *Proceedings of the 23rd DAC,* Jun.-Jul., 1986, pp. 79–85.
68. Kurt Keutzer, op. cit.
69. R. Brayton, E. Detjens, S. Krishna, T. Ma, P. McGeer, L. Pei, N. Phillips, R. Rudell, R. Segal, A Wang, R. Yung, and A. Sangiovanni-Vincentelli, "Multiple-level logic optimization system," *Proc. IEEE ICCAD 1986,* pp. 356–359; and R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang, "MIS: a multi-

Micron Ex. 1037, p. 502
Micron v. YMTC
IPR2025-00119

ple-level logic optimization system," *IEEE Transactions on CAD of Integrated Circuits and Systems,* vol. CAD-6, no. 6, Nov. 1987, pp. 1062–1081.

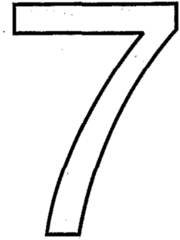
70. D. Bostick, G. D. Hachtel, R. Jacoby, M. R. Lightner, P. Moceyunas, C. R. Morrison, and D. Ravenscroft, "The Boulder Optimal Logic Design system," *Proc. Int. Conf. on Computer-Aided Design,* Nov. 1987, pp. 62–65.
71. *EDIF Electronic Design Interchange Format Version 2 0 0* (Paul Stanford and Paul Mancuso, eds.), Washington, D.C.: Electronic Industries Association, 1989.
72. Ulrich Lauther, "A min-cut placement algorithm for general cell assemblies based on a graph," *Proceedings of the 16th Design Automation Conference,* 1979, pp. 1–10.
73. S. Kirkpatrick, C. Gelatt, and M. Vecchi, "Optimization by simulated annealing," *Science,* vol. 220, no. 4598, May 13, 1983, pp. 671–680.
74. C. Sechen and A. Sangiovanni-Vincentelli, "TimberWolf3.2: a new standard cell placement and global routing package," *Proceedings of the 23rd Design Automation Conference,* Las Vegas, Nev., 1986, pp. 432–439.
75. A. Sangiovanni-Vincentelli, A. Santamauro, and J. Reed, "A new gridless channel router: Yet Another Channel Router the second (YACRII)," *Proceedings of the International Conference on Computer-Aided Design,* 1984, pp. 72–75.
76. Ronald L. Rivest and Charles Fiducia, "A 'greedy' channel router," *Proceedings of the 19th Design Automation Conference,* 1982, pp. 418–424.
77. J. Soukup, "Fast maze router," *Proceedings of the 15th Design Automation Conference,* Las Vegas, Nev., 1978, pp. 100–102.
78. J. Soukup, "Global router," *Proceedings of the 16th Design Automation Conference,* Las Vegas, Nev., 1978, pp. 481–484.
79. Hung-fai Steven Law, Graham Wood, and Mindy Lam, "An Intelligent Composition Tool for Regular and Semiregular VLSI Structures," *Silicon Compilation* (Daniel D. Gajski, ed.), Reading, Mass.: Addison-Wesley, 1988.
80. Steven M. Rubin, *Computer Aids for VLSI Design,* Reading, Mass.: Addison-Wesley, 1987, Chapter 11.
81. James J. Cherry, *op. cit.*
82. William R. Heller, G. Sorkin, and Klim Maling, "The planar package planner for system designers," *Proceedings of the 19th Design Automation Conference,* Jun. 1982, pp. 253–260.
83. Ralph H. J. M. Otten, "Automatic floorplan design," *Proceedings of the 19th Design Automation Conference,* Jun. 1982, pp. 261–267.
84. C. L. Wardle, C. R. Watson, C. A. Wilson, J. C. Mudge, and B. J. Nelson, "A declarative design approach for combining macrocells by directed placement and constructive routing," *Proceedings of the 21st Design Automation Conference,* 1984, pp. 594–601.
85. Christopher J. Terman, "Simulation tools for VLSI," *VLSI CAD Tools and Applications* (Wolfgang Fichtner and Martin Morf, eds.), Norwell, Mass.: Kluwer Academic, 1987, Chapter 3.
86. L. W. Nagel, "SPICE2: a computer program to simulate semiconductor circuits," *Memo ERL-M520,* Dept. Electrical Engineering and Computer Science, University of California at Berkeley, May 9, 1975.
87. W. T. Weeks, A. J. Jimenez, G. W. Mahoney, D. Mehta, H. Qasemzadeh, and T. R. Scott, "Algorithms for ATSAP—a network analysis program," *IEEE Transactions on Circuit Theory,* vol. CT-20, Nov. 1973, pp. 628–634.
88. *HSPICE User's Manual H9001,* Campbell, Calif.: Meta-Software, 1990.

Micron Ex. 1037, p. 503
Micron v. YMTC
IPR2025-00119

89. B. R. Chawla, H. K. Gummel, and P. Kozak, "MOTIS—an MOS timing simulator," *IEEE Transactions on Circuits and Systems,* vol. 22, no. 12, Dec. 1975, pp. 901–910.
90. J. White and A. Sangiovanni-Vincentelli, *Relaxation Techniques for the Simulation of VLSI Circuits,* Hingham, Mass.: Kluwer Academic, 1987.
91. C. Terman, "Timing simulation for large digital MOS circuits," in *Advances in Computer-Aided Engineering Design,* Volume 1 (A. Sangiovanni-Vincentelli, ed.), JAI Press, 1984, pp. 1–91.
92. Thomas G. Szymanski, "LEADOUT: a static timing analyzer for MOS circuits," *IEEE International Conference on Computer-Aided Design,* Santa Clara, Calif., Nov. 1986, pp. 130–133.
93. James J. Cherry, "Pearl: a CMOS timing analyzer," *IEEE/ACM Proceedings of the 25th Design Automation Conference,* Anaheim, Calif., 1988, pp. 148–153, and "Pearl User's Guide," Parsec, Inc., Palo Alto, Calif. 1992.
94. C. Ebeling and O. Zajicek, "Validating VLSI circuit layout by wirelist comparison," *Proceedings of IEEE Int. Conf. on CAD,* Sept. 1983, pp. 172–173.
95. M. Hofmann and V. Lauther, "HEX: an instruction driven approach to feature extraction," *Proceedings of the 20th Design Automation Conference,* Jun. 1983, pp. 331–336.
96. C. M. Baker and C. J. Terman, "Tools for verifying integrated circuit designs," *Lambda Magazine (VLSI Design),* 4th quarter 1980, pp. 22–30.
97. T. G. Szymanski and C. J. Van Wyk, "Space efficient algorithms for VLSI artwork analysis," *Proceedings of the 20th Design Automation Conference,* June 1983, pp. 734–739.
98. C. M. Baker and C. J. Terman, *op. cit.*
99. H. S. Baird, "Fast algorithms for LSI artwork analysis," *Proceedings of the 14th Design Automation Conference,* 1977, pp. 303–311.
100. T. Whitney, "A Hierarchial Design Analysis Front End," *VLSI '81,* 1981, pp. 217–225.
101. Steven M. Rubin, *op. cit.,* Appendix E.
102. Curt F. Fey and Demetris E. Paraskevopoulos, "Studies in LSI technology economics II: a comparison of product costs using MSI, gate arrays, standard cells, and full custom VLSI," *IEEE JSSC,* vol. SC-21, no. 2, Apr. 1986, pp. 297–303.
103. Demetris E. Paraskevopoulos and Curt F. Fey, "Studies in LSI technology economics III: design schedules for application-specific integrated circuits," *IEEE JSSC,* vol. SC-22, no. 2, Apr. 1987, pp. 223–229.
104. Curt F. Fey, "Custom LSI/VLSI chip design productivity," *IEEE JSSC,* vol. SC-20, no. 2, Apr. 1985, pp. 555–561.
105. Curt F. Fey and Demetris E. Paraskevopoulos, "Studies in LSI technology economics IV: models for gate array design productivity," *IEEE JSSC,* vol. SC-24, no. 4, Aug. 1989, pp. 1085–1091.

Micron Ex. 1037, p. 504
Micron v. YMTC
IPR2025-00119

CMOS TEST METHODS



7.1 The Need for Testing

While in real estate the refrain is "Location! Location! Location!," the comparable advice in IC design should be "Testing! Testing! Testing!" While most problems in VLSI design have been reduced to algorithms in readily available software, the responsibilities for the various levels of testing and testing methodology can be a significant burden on the designer.

In Chapter 4 we noted that the yield of a particular IC was the number of good die divided by the total number of die per wafer. Due to the complexity of the manufacturing process not all die on a wafer correctly operate. Small imperfections in starting material, processing steps, or in photomasking may result in bridged connections or missing features. It is the aim of a test procedure to determine which die are good and should be used in end systems.

Testing a die (chip) can occur:

- at the wafer level.
- at the packaged-chip level.
- at the board level.
- at the system level.
- in the field.

Micron Ex. 1037, p. 505
Micron v. YMTC
IPR2025-00119

By detecting a malfunctioning chip at an earlier level, the manufacturing cost may be kept low. For instance, the approximate cost to a company of detecting a fault at the above levels is[1]:

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

Obviously, if faults can be detected at the wafer level, the cost of manufacturing is kept the lowest. In some circumstances, the cost to develop adequate tests at the wafer level, mixed signal requirements or speed considerations may require that further testing be done at the packaged-chip level or the board level. A component vendor can only test at the wafer or chip level. Special systems, such as satellite-borne electronics, might be tested exhaustively at the system level.

Tests may fall into two main categories. The first set of tests verifies that the chip performs its intended function; that is, that it performs a digital filtering function, acts as a microprocessor, or communicates using a particular protocol. In other words, these tests assert that all the gates in the chip, acting in concert, achieve a desired function. These tests are usually used early in the design cycle to verify the functionality of the circuit. These will be called *functionality tests* in this book. They may be lumped into the verification activity. The second set of tests verifies that every gate and register in the chip functions correctly. These tests are used after the chip is manufactured to verify that the silicon is intact. They will be called *manufacturing tests* in this book. In many cases these two sets of tests may be one and the same, although the natural flow of design usually has a designer considering function before manufacturing concerns.

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

7.1.1 Functionality Tests

Functionality tests are usually the first tests a designer might construct as part of the design process. Does this adder add? Does this counter count? Does this state-machine yield the right outputs at the right clock cycles?

For most systems, functionality tests involve proving that the circuit is functionally equivalent to some specification. That specification might be a verbal description, a plain-language textual specification, a description in

Micron Ex. 1037, p. 506
Micron v. YMTC
IPR2025-00119

some high-level computer language such as C, FORTRAN, Pascal, or Lisp or in a hardware-description language such as VHDL, ELLA, or Verilog®, or simply a table of inputs and required outputs. Functional equivalence involves running a simulator at some level on the two descriptions of the chip (say, one at the gate level and one at a functional level) and ensuring for all inputs applied that the outputs are equivalent at some convenient checkpoints in time. The most detailed check might be on a cycle-by-cycle basis.

Functional equivalence may be carried out at various levels of the design hierarchy. If the description is in a behavioral language (such as the last two categories mentioned), the behavior at a system level may be verifiable. For instance, in the case of a microprocessor, the operating system might be booted and key programs might be run for the behavioral description. However, this might be impractical (due to long simulation times) for a gate-level model and extremely impractical for a transistor-level model. The way out of this impasse is to use the hierarchy inherent within a system to verify chips and modules within chips. That, combined with well-defined modular interfaces, goes a long way in increasing the likelihood that a system composed of many VLSI chips will be first-time functional. Remember too, at the lowest levels of the hierarchy, timing tests must be run to validate that a particular function such as addition is achieved at a given clock frequency.

There is no good theory on how to ensure that good functional tests be written. The best advice is to simulate the chip or system as closely as possible to the way it will be used in the real world. Often this is impractical due to slow simulation times and very long verification sequences. One approach is to move up the simulation hierarchy as modules become verified at lower levels. For instance, the gate-level adder and register modules in a video filter might be replaced by functional models and then the filter itself might be replaced by a functional model. At each level, small tests are written to verify the equivalence between the new higher-level functional model and the lower-level gate or functional level. At the top level, the filter functional-model may be surrounded with a software environment that models the real-world use of the filter. For instance, a carefully selected subsample of a video frame might be fed to the filter and the output of the functional model compared with what is expected theoretically. The video output might also be observed on a video frame buffer to check that it looks correct (by no means an exhaustive test, but a confidence builder). Finally, if enough time is available, all or part of the functional test may be appplied to the gate level and even the transistor level if transistor primitives have been used. One approach that is becoming more popular and feasible is to model chips as collections of reprogrammable gate arrays (see Chapter 6). Commercial hardware is available to aid this activity.

Remember the following statement, culled from many years of IC-design experience, whenever you are tempted to give test work short shrift:

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

Micron Ex. 1037, p. 507
Micron v. YMTC
IPR2025-00119

7.1.2 Manufacturing Tests

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

- layer-to-layer shorts (i.e., metal to metal).
- discontinous wires (i.e., metal thins when crossing vertical topology jumps).
- thin-oxide shorts to substrate or well.

These in turn lead to particular circuit maladies, including:

- nodes shorted to power or ground.
- nodes shorted to each other.[2]
- inputs floating/outputs disconnected.

Tests are required to verify that each gate and register is operational and has not been compromised by a manufacturing defect. Tests are normally carried out at the wafer level to cull out bad die, and then on the packaged parts. The length of the tests at the wafer level might be shortened to reduce test time based on experience with the test sequence.

Apart from the verification of internal gates, I/O integrity is also tested through completing the following tests:

- I/O-level test (i.e., checking the noise margin for TTL, ECL, or CMOS I/O pads).
- Speed test.
- I_{DD} test.

The last of these tests checks the leakage if the circuit is composed of complementary logic. Any value markedly above the expected value for a given wafer normally indicates an internal shorting failure (or very bad leakage). Wafer tests may be done at high speed or low speed (1 MHz) due to possible power and ground bounce effects that may be present in older testers.

In general, manufacturing-test generation assumes that the circuit/chip functions correctly, and ways of exercising all gate inputs and of monitoring all gate outputs are required.

To illustrate the difference between a functional test and a manufacturing test, consider the testing of a microprocessor at a functional level, which might

Micron Ex. 1037, p. 508
Micron v. YMTC
IPR2025-00119

be the first concern of the designer (to see whether the microprocessor worked as a whole). To test any instruction, a sequence of instructions that use that instruction might be used (i.e., does the ADD instruction add?). While this might prove that the control logic that yields that instruction is intact, it does not, for instance, prove that the instruction works for all possible addresses and data. At this level of test it is assumed that the adders, muxes, gates, and registers in the microprocessor datapaths operate correctly.

Tests that exercise all bits in the datapaths have to be written to verify the chip at the manufacturing level. These tests might include a test to check that registers can store a 1 and a 0 and a test that exercises each bit in any adder and ensures that the carry chain is not broken (i.e., does the adder add for all inputs?). The inputs have to be chosen carefully to check for all possible manufacturing defects. The manufacturing tests may be the only tests applied to a microprocessor prior to its being placed in a socket and booted.

7.1.3 A Walk Through the Test Process

As a designer you may be responsible for part or all of the tests that are written to test a particular chip (often called the stimulus). "Written" might include a number of methods of test specification from applying waveforms or logic values manually to a simulator to, more probably, writing a program in a high-level language to apply stimuli to a description of the circuit. When the stimulus is applied to a circuit via a simulator, the output of the simulator may be dumped to a file (often called an *activity file*). If this output is filtered so that only the chip inputs and chip outputs are retained and further filtered so that only the quiescent signal values are kept after an input or inputs change, then the resulting file may be used to generate a "test program."

Depending on whether you are testing a wafer or a packaged part, a probe card or "device-under-test" (DUT) board would be needed to connect the tester outputs and inputs to the die I/O pads or chip package pins. Probe cards are normally constructed by experts, while DUT boards are well within the capabilities of the electronic hobbyist.

The next requirement for a chip tester is the existence of this "test program." This is a file with a format of inputs and outputs that suit the chip tester that is to be used to test your chip. A simple format is shown below for the case of a single-bit adder:

```
      III   OO
            SC
            UA
            MR
             R
      ABC    Y
0     000   00
```

```
1    001  10
2    010  10
3    011  01
4    100  10
5    101  01
6    110  01
7    111  11
```

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

Sundry other setup files are normally required by the tester. Normally a mapping file is required that maps a given input or output in the test program to a physical connection (pin) in the tester. This pin may be programmed to be an input, output, tristate, bidirect, or, in some cases, a multiplexed data pin. Each pin on the tester is driven by a function memory that is used either to assert a value or to check a value at a DUT pin. In addition, various control memories may be present to control the drive on the tester pin (i.e., to control a tristate pin) or to mask data from the chip (i.e., to ignore certain pins at certain times). These memories have finite length, so sometimes with older testers more than one vector load has to be used to test a part. This normally slows testing because the reload procedure may be slow. Modern testers seldom suffer from this problem.

The clock speed is specified (by specifying a test cycle time, T_C), as are supply-voltage levels and pins on the tester and probe card or DUT board. The time at which outputs are asserted or inputs are sampled is also specified on a pin-by-pin basis (T_S). The format of the test data may usually be chosen from Non Return to Zero (NRZ), Return To Zero (RTZ), or other formats, such as Surround By Zero (SBZ). For instance, an RTZ output would transition (if the pin were driven high) at T_S and return to zero at T_C.

The probe card or DUT board is connected to the tester. The test program is compiled and downloaded into the tester, and the tests are applied to the circuit. The tester samples the chip outputs and compares the values with

Micron Ex. 1037, p. 510
Micron v. YMTC
IPR2025-00119

those provided by the test program. If there are any differences, the chip is marked as faulty (with an ink dot) and the miscomparing vectors may be displayed for reference. In the case of a probe card, the card is raised, moved to the next die on the wafer, and lowered, and the test procedure is repeated. In the case of a DUT board with automatic part handling, the tested part is binned into a good or bad bin and a new part is fed to the DUT board, and the test is repeated. In most cases these procedures take a few seconds for each part tested.

The ability to vary the voltage and timing on a per-pin basis with a tester allows a process known as "schmooing" to be carried out. For instance, one might vary the V_{DD} voltage from 3 to 6 volts on a 5-volt part while varying the tester cycle time.This yields a graph that shows the speed sensitivity of the part with respect to voltage. Another "schmoo" test that is frequently exercised is to skew the timing on inputs with respect to the chip clock to look for setup and hold variations.

7.2 Manufacturing Test Principles

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

Figure 7.1(a) shows a combinational circuit with n-inputs. To test this circuit exhaustively a sequence of 2^n inputs (or test vectors) must be applied and observed to fully exercise the circuit. This combinational circuit is converted to a sequential circuit with addition of m-storage registers, as shown in Fig. 7.1(b). The state of the circuit is determined by the inputs and the pre-

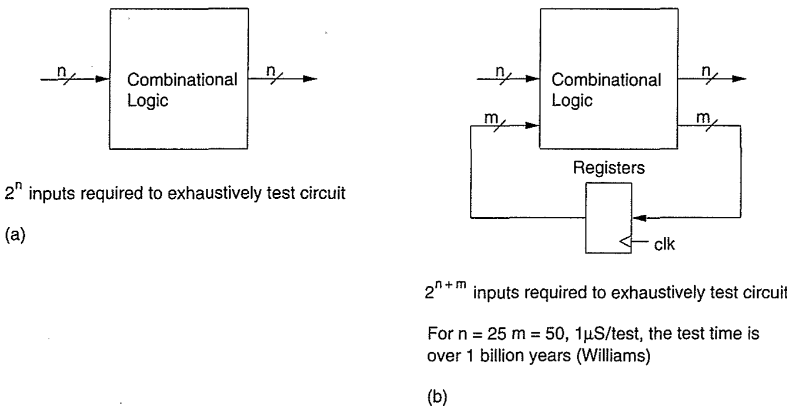


FIGURE 7.1 The combinational explosion in test vectors

Micron Ex. 1037, p. 511
Micron v. YMTC
IPR2025-00119

vious state. A minimum of $2^{(n+m)}$ test vectors must be applied to exhaustively test the circuit. To quote Williams[3]:

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

Clearly, this is an important area of design that has to be well understood.

7.2.1 Fault Models

7.2.1.1 Stuck-At Faults

In order to deal with the existence of good and bad parts it is necessary to propose a "fault model," that is, a model for how faults occur and their impact on circuits. The most popular model is called the "Stuck-At" model. With this model, a faulty gate input is modeled as a "stuck at zero" (Stuck-At-0, S-A-0, SA0) or "stuck at one" (Stuck-At-l, S-A-l, SA1). This model dates from board-level designs where this was determined to be an adequate set of models for modeling faults. Figure 7.2 illustrates how an S-A-0 or S-A-1 fault might occur. These faults most frequently occur due to thin-oxide shorts (the n-transistor gate to V_{SS} or the p-transistor gate to V_{DD}) or metal-to-metal shorts.

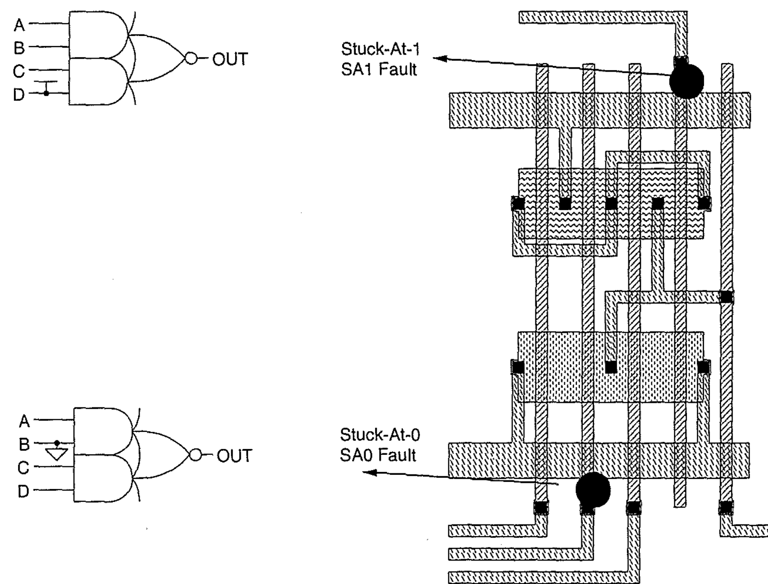


FIGURE 7.2 CMOS stuck-at faults

Micron Ex. 1037, p. 512
Micron v. YMTC
IPR2025-00119

7.2.1.2 Short-Circuit and Open-Circuit Faults

Other models include "stuck-open"[4] or "shorted" models. Two shorted faults are shown in Fig. 7.3. Considering the faults shown in Fig. 7.3, the short *S1* is modeled by an S-A-0 fault at input *A,* while short *S2* modifies the function of the gate. What becomes evident is that to ensure the most accurate modeling, faults should be modeled at the transistor level, because it is only at this level that the complete circuit structure is known. For instance, in the case of a simple NAND gate, the intermediate node in the series n-pair is "hidden" by the schematic. What this implies is that test generation must be done in such a way as to take account of possible shorts and open circuits at the switch level.[5] Although the switch level may be the most appropriate level, expediency dictates that most existing systems rely on Boolean logic representations of circuits and S-A-0 and S-A-1 fault modeling.

A particular problem that arises with CMOS is that it is possible for a fault to convert a combinational circuit into a sequential circuit. This is illustrated for the case of a 2-input NOR gate in which one of the transistors is rendered ineffective (stuck open or stuck closed) in Fig. 7.4. This might be due to a missing source, drain, or gate connection. If one of the n-transistors (*A* connected to gate) is stuck open, then the function displayed by the gate will be

$$F = (not\ (A + B)) + (A\ .\ (not\ B)\ .\ F_n),$$

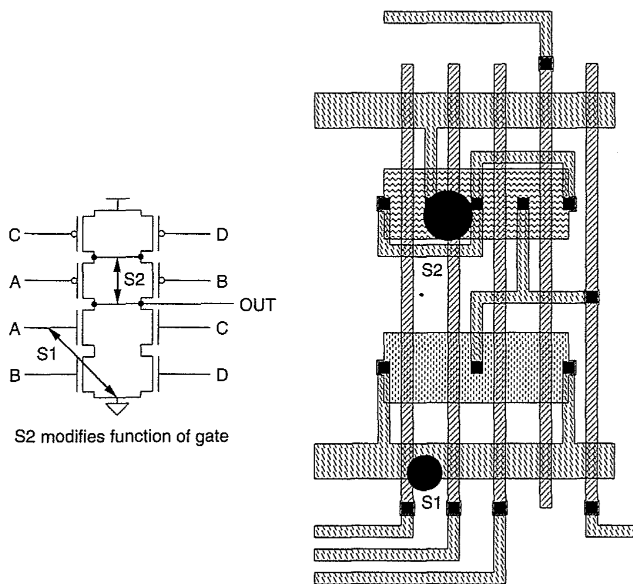


FIGURE 7.3 CMOS bridging faults

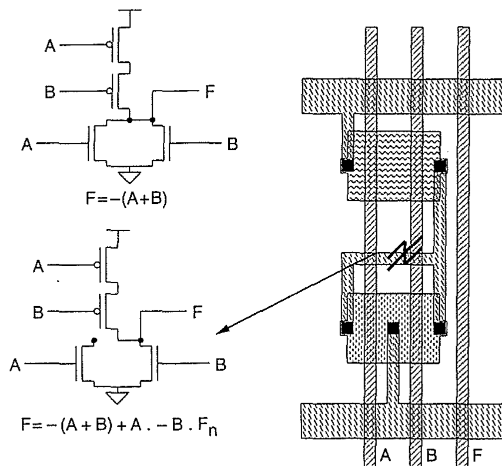


FIGURE 7.4 A CMOS open fault that causes sequential faults

where F_n is the previous state of the gate. Similarly if the B n-transistor drain connection is missing, the function is

$$F = (not\ (A + B)) + ((not\ A) \ .\ B \ .\ F_n).$$

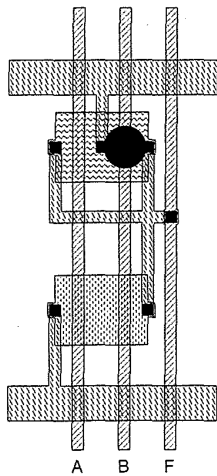


FIGURE 7.5 A defect that causes static I_{DD} current

If either p-transistor is open, the node would be arbitrarily charged (i.e., it might be high due to some weird charging sequence) until one of the n-transistors discharged the node. Thereafter it would remain at zero, bar charge-leakage effects. This problem has caused researchers to search for new methods of test generation to detect such behavior.[6]

Currently debate ranges over whether an SA0/SA1 approach to testing is adequate for testing CMOS. It is also possible to have switches (transistors) exhibit a "stuck-open" or "stuck-closed" state. Stuck-closed states can be detected by observing the static V_{DD} current (I_{DD}) while applying test vectors. Consider the gate fault shown in Fig. 7.5, where a p-transistor in a 2-input NAND gate is shorted. This could physically occur if stray metal overlapped the source and drain connections or if the source and drain diffusions shorted. If we apply test vector 11 to the A and B input and measure the static I_{DD} current, we will notice that it rises to some value determined by the β ratios of the n- and p-transistors. While the debate continues and test cycles are at a premium, the SA0/SA1 model will suffice for some time to come.

7.2.2 Observability

The observability of a particular internal circuit node is the degree to which one can observe that node at the outputs of an integrated circuit (i.e., the pins). This measure is of importance when a designer/tester desires to measure the output of a gate within a larger circuit to check that it operates cor-

Micron Ex. 1037, p. 514
Micron v. YMTC
IPR2025-00119

rectly. Given a limited number of nodes that may be directly observed, it is the aim of well-designed chips to have easily observed gate outputs, and the adoption of some basic test design techniques can aid tremendously in this respect. Ideally, one should be able to observe directly or with moderate indirection (i.e., one may have to wait a few cycles) every gate output within an integrated circuit. While at one time this aim was hindered by limited gate-count processes and a lack of design methodology, current design practices and processes allow one to approach this ideal. Section 7.3 examines a range of methods for increasing observability.

7.2.3 Controllability

The controllability of an internal circuit node within a chip is a measure of the ease of setting the node to a 1 or 0 state. This measure is of importance when assessing the degree of difficulty of testing a particular signal within a circuit. An easily controllable node would be directly settable via an input pad. A node with little controllability might require many hundreds or thousands of cycles to get it to the right state. Often one finds it impossible to generate a test sequence to set a number of poorly controllable nodes into the right state. It should be the aim of a well-designed circuit to have all nodes easily controllable. In common with observability, the adoption of some simple design for test techniques can aid tremendously in this respect.

7.2.4 Fault Coverage

A measure of goodness of a test program is the amount of fault coverage it achieves; that is, for the vectors applied, what percentage of the chip's internal nodes were checked. Conceptually, the way in which the fault coverage is calculated is as follows. Each circuit node is taken in sequence and held to 0 (S-A-0), and the circuit is simulated, comparing the chip outputs with a known "good machine"—a circuit with no nodes artificially set to 0 (or 1). When a discrepancy is detected between the "faulty machine" and the good machine, the fault is marked as detected and the simulation is stopped. This is repeated for setting the node to 1 (S-A-1). In turn, every node is stuck at 1 and 0, sequentially. The total number of nodes that, when set to 0 or 1, do result in the detection of the fault, divided by the total number of nodes in the circuit, is called the percentage-fault coverage.

The above method of fault analysis is called sequential fault grading. While this might be practical for small circuits, or by using hardware simulation accelerators on medium circuits, the time to complete the fault grading may be very long. On average KN cycles (assuming that, on average, N/2 cycles are needed to detect each fault) need to be simulated, where K is the number of nodes in the circuit and N is the length of the test sequence. For $K = 1000$ and $N = 12{,}000$, 12 million cycles are required. At 1 ms per cycle, this yields 12,000 seconds or 3 hrs 20 minutes. For $K = 100{,}000$ and

Micron Ex. 1037, p. 515
Micron v. YMTC
IPR2025-00119

$N = 360{,}000$, 3.6×10^9 cycles are required. At 1 s per cycle, 1040 years would be required to do sequential fault grading.

To overcome these long simulation times many ingenious techniques have been invented to deal with fault simulation.

7.2.5 Automatic Test Pattern Generation (ATPG)

Historically in the IC industry, designers designed circuits, layout drafts-people completed the layout, and the test engineer wrote the tests. In many ways, the test engineers were the Sherlock Holmes of the industry, reverse engineering circuits and devising tests that would test the circuits in an adequate manner. For the longest time, test engineers implored circuit designers to include extra circuitry to ease the burden of test generation. Happily, as processes have increased in density and chips have increased in complexity, the inclusion of test circuitry has become less of an overhead for both the designer and the manager worried about the cost of the die. In addition, as tools have improved, more of the burden for generating tests has fallen on the circuit/logic designer. To deal with this burden, methods for automatically generating tests have been invented. Collectively these are known as ATPG, for Automatic Test Pattern Generation. This section summarizes one approach to ATPG to provide background for the reader. In practice, one may find that ATPG is of great use in the generation of test vectors or that for a variety of reasons it is not applicable.

Historically, most ATPG approaches have been based on simulation. A five-valued logic[7] form is commonly used to implement test generation algorithms (more advanced algorithms use up to 10 level logic). This consists of the states 1, 0, D, $\overline{D}$, and X. 0 and 1 represent logical zero and logical one respectively. X represents the unknown or DON'T-CARE state. D represents a logic 1 in a good machine and a logic 0 in a faulty machine while $\overline{D}$ represents a logic 0 in a good machine and a logic 1 in a faulty machine. The truth tables for inverters, AND, and OR gates are shown in Tables 7.1, 7.2, and 7.3.

TABLE 7.1 Inverter Z = NOT A

A	Z
0	1
1	0
X	X
D	$\overline{D}$
D	D

Micron Ex. 1037, p. 516
Micron v. YMTC
IPR2025-00119

TABLE 7.2 2-input AND gate Z = A AND B

A	B	0	1	X	D	$\overline{D}$
0		0	0	0	0	0
1		0	1	X	D	D
X		0	X	X	X	X
D		0	D	X	D	0
D		0	D	X	0	D

We can examine the use of this five-valued logic by considering the circuit shown in Fig. 7.6 where an S-A-0 fault is to be detected at node *h*. We will alternatively call a circuit a *machine,* which is customary in test nomenclature. Thus node *h* would have value *D*. There are two objectives. The first is to propagate the *D* on node *h* to one or more *primary outputs* (POs). A primary output is a directly observable signal, such as a pad or, as we shall learn later, a scan output. This path to the primary output (or outputs) is called the *sensitized path.* The second objective is to set node *h* to state *D* via a set of *primary inputs* (PIs). A primary input is one that can be directly set via a pad or some other means. The gate driving node *h* is the *Gate Under Test* or GUT. From node *h* we backtrack to the primary inputs (*a, b, c, d, e*) to find the necessary input vector required to set node *h* to a 1. Because the gate driving node *h* is an AND gate from the above definition (a *D* is a 1 in a good machine), both inputs (*f, g*) have to be set to 1 to set *h* to 1. Proceeding further toward the inputs, to assert node *f* as a 1, both nodes *a* and *b* have to be set to a 1. Because node *g* is driven by an OR gate, either node *c* or node *d* need to be set to a 1 to assert node *g*. Thus a vector {*a,b,c,d*} of {1,1,1,0} or {1,1,0,1} is required to control node *h*. To observe that node *g* has been set to a *D,* input node *e* has to be set to a 1. Thus the resultant test vector is

TABLE 7.3 2-input OR gate Z = A OR B

A	B	0	1	X	D	$\overline{D}$
0		0	1	X	D	$\overline{D}$
1		1	1	1	1	1
X		X	1	X	X	X
D		D	1	X	D	1
$\overline{D}$		$\overline{D}$	1	X	1	$\overline{D}$

Micron Ex. 1037, p. 517
Micron v. YMTC
IPR2025-00119

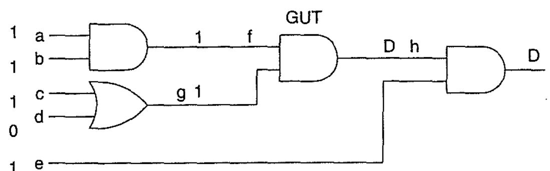


FIGURE 7.6 The *D* algorithm—sensitization step

$\{a,b,c,d,e\} = \{1,1,0,1,1\}$ or $\{1,1,1,0,1\}$. If we are checking for an S-A-1 fault at node *h*, we must be able to set it to 0. By similar reasoning to that for the S-A-0 case the test vector would be $\{a,b,c,d,e\} = \{0,1,X,X,1\}$ or $\{1,0,X,X,1\}$ or $\{0,0,X.X,1)$ or $\{1,1,0,0,1\}$. Similarly, for other nodes a summary of the vectors is as in Table 7.4.

The next step is to collapse the vectors into the least set that covers all nodes. A possible set is $\{1,1,0,1,1\}$, $\{0,0,1,0,1\}$, $\{1,1,0,0,1\}$.

The reason for using a five-valued logic is shown in Fig. 7.7. Here an additional AND gate and INVERT gate have been added to the circuit. We can see that a fault at node *h* is essentially unobservable (due here to the nonsensical logic). This circuit suffers from what is called reconvergent fan-out.

The usual basis for manual generation of tests by test engineers and many current automatic test-pattern generation programs is the *D*-algorithm (DALG).[8] PODEM[9] and PODEM-X[10] are improved algorithms that are more efficient than the original DALG and in addition treat error-correcting circuits composed of XOR gates with reconvergent fan-out. Another ATPG algorithm is called FAN[11] and an improved efficiency algorithm[12] dealing with tristate drivers called ZALG has been developed.[13] Other work has concentrated on dealing at a module level rather that a gate level.[14] In basis, these algorithms start by propagating the *D* value on an internal node to a primary output. This is called the *D-propagation* phase. The selection of which gates to pass through to the output is guided by observability indexes assigned to gates. At any particular gate input, the gate with the highest observability is selected. Once the *D* value is observable at a primary output, the next step is to determine the primary input values that are required to

TABLE 7.4 Node-vector Summary of *D* Algorithm (Fig. 7.6)

NODE	TEST	VECTOR $\{a,b,c,d,e\}$
h	S-A-0	$\{1,1,0,1,1\}$, $\{1,1,1,0,1\}$
h	S-A-1	$\{0,1,X,X,1\}$, $\{1,0,X,X,1\}$, $\{0,0,X,X,1\}$, $\{1,1,0,0,1\}$
f	S-A-0	$\{1,1,0,1,1\}$, $\{1,1,1,0,1\}$
f	S-A-1	$\{0,0,0,1,1\}$, $\{0,0,1,0,1\}$
g	S-A-0	$\{1,1,0,1,1\}$, $\{1,1,1,0,1\}$, $\{1,1,1,1,1\}$
g	S-A-1	$\{1,1,0,0,1\}$

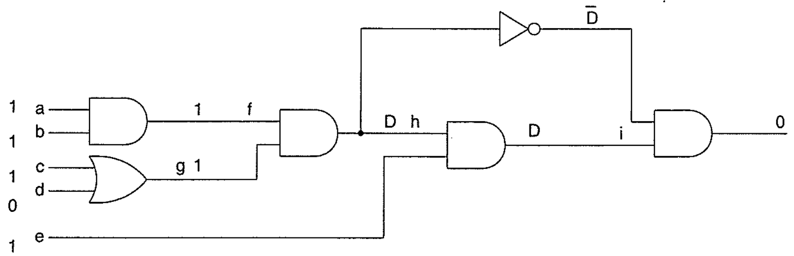


FIGURE 7.7 Reconvergent fan-out with *D* notation

enable the fault to be observed and tested. This proceeds by backtracking from the faulted signal and sensitized path-enables toward the primary inputs. The selection of which path to proceed along toward the inputs is aided by controllability indices assigned to nodes. This is known as the *backtrace* step.

Controllabilities and observabilities can be assigned statically (that is, without regard to the logic state of the network) or dynamically[15] (that is, according to the current state of the network). The SCOAP[16] algorithm is one method of assigning controllabilities and observabilities. In the SCOAP system the following six testability measures (TMs) are defined for each circuit node:

- $CC0(n)$—combinatorial 0 controllability of node n (i.e., the extent to which a combinatorial node can be forced to a zero).
- $CC1(n)$—combinatorial 1 controllability of node n.
- $CO(n)$—combinatorial observability of node n.
- $SC0(n)$—sequential 0 controllability of node n.
- $SC1(n)$—sequential 1 controllability of node n.
- $SO(n)$—sequential observability of node n.

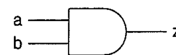


FIGURE 7.8 NAND gate

The combinatorial measures are applied to the ouputs of logic gates, while the sequential measures apply to registers and other "sequential" modules. As an example, for the AND gate shown in Fig. 7.8 the $CC1$ value is

$$CC1(z) = CC1(a) + CC1(b) + 1.$$

That is, the 1-controllability of the output of the AND gate is the sum of the 1-controllabilities of each input because each input has to be set to 1 to set the output to 1. The 1 is added at the end because the AND gate represents one stage of combinatorial logic. The sequential 1-controllability is given by

$$SC1(z) = SC1(a) + SC1(b).$$

Micron Ex. 1037, p. 519
Micron v. YMTC
IPR2025-00119

The combinatorial 0-controllability is given by

$$CC0(z) = min[CC0(a), CC0(b)] + 1.$$

This arises due to the fact that either a 0 on a or b forces a 0 at the output. Therefore the easiest controllable input may be used (the lowest combinatorial controllability). The sequential controllability is given by

$$SC0(z) = min[SC0(a), SC0(b)].$$

The combinatorial observability of a is given by

$$CO(a) = CO(z) + CC1(b) + 1;$$

that is, the observability of z added to the combinatorial 1-controllability of b. This occurs because b has to be forced to a 1 to make a observable. The sequential observability of a is given by

$$SO(a) = SO(z) + SO(b).$$

Similar equations may be derived for other gate types. The SCOAP algorithm proceeds by first calculating the circuit controllabities by propagating controllabilites from the logic inputs. Following this, the observabilities are propagated from the logic outputs. Figure 7.9(a) shows a logic circuit with the 1-controllabities annotated. Figure 7.9(b) shows the observabilities.

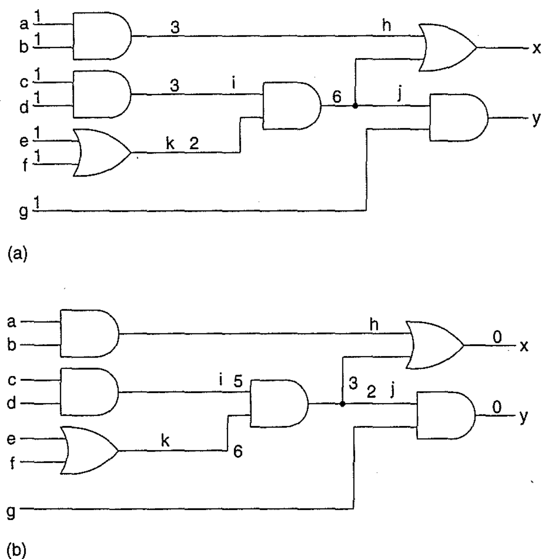


FIGURE 7.9 SCOAP testability measure example: (a) controllabilities; (b) observabilities

Micron Ex. 1037, p. 520
Micron v. YMTC
IPR2025-00119

In cases of multiple fan-out, the minimum observability measure is used. The presence of high controllability numbers indicates a node that is difficult to control, while the presence of high observability numbers indicates nodes that are difficult to observe. As mentioned above, the testability measures are used to guide the selection of paths in the D-propagation and backtrace phase of the D-algorithm–based ATPG procedures.

Other testability measures, such as COP[17] and LEVEL,[18] are also used. COP testability measures are probablistic in nature.

More recently authors have proposed the use of massively parallel methods for ATPG.[19] Methods have also been developed that model faults as changes to a Boolean network. Equivalence checking is used to prove that the two networks are not equivalent. These methods, when combined with random-fault generation and fault simulation, have demonstrated a great deal of success.[20]

7.2.6 Fault Grading and Fault Simulation

Fault grading consists of two steps. First, the node to be faulted is selected. Normally global nodes such as reset lines and clock lines are excluded because faulting them can lead to unnecessary simulation (i.e., if the reset or clock line is stuck, then not much is going to happen in the circuit). A simulation is run with no faults inserted, and the results of this simulation (that is, the primary output responses for each input test vector) are saved. Following this process, in principle, each node or line to be faulted is set to 0 and then 1 and the test vector set is applied. If, and when, a discrepancy is detected between the faulted circuit response and the good circuit response, the fault is said to be detected and the simulation is stopped, and the process is repeated for the next node to be faulted. If the number of nodes to be faulted is K, and the average number of test vectors is N, the number of simulation cycles, S_K, is approximately given by

$$\begin{aligned} S_K &= 2\frac{N}{2}K + N \\ &= K(N+1) \approx KN. \end{aligned} \tag{7.1}$$

This serial fault simulation process is therefore running K sets of the test vector set. With a small vector set, simple circuit, or very fast simulator, this approach is feasible. However, for large test sets and circuits, it is highly impractical.

To deal with this problem, a number of ideas have been developed to increase the speed of fault simulation.

Parallel Simulation is one method for speeding up simulation of multiple machines. In this method m words in an n-bit computer are used to

Micron Ex. 1037, p. 521
Micron v. YMTC
IPR2025-00119

encode the state of n "machines" for a 2^m-state simulator. Two n-bit words may be used to encode n machines for a three-state simulator. More computer words may be used to encode simulators with more states. Moreover this principle has been extended to special-purpose hardware where the computer word length could be optimized to deal with substantially more circuits in parallel. Now if M circuits can be simulated in parallel, then

$$S_K = \frac{KN}{M}. \quad (7.2)$$

Concurrent Simulation[21] is currently the most popular method for software-based fault simulators. The technique uses a nonfaulted version of the circuit to create a "good" machine model. Each fault creates a new faulty machine that is simulated in parallel with the good machine. Thus $N + 1$ simulations may have to be completed, where N is the number of faults. Concurrent simulators rely on a number of heuristics to reduce the amount of simulation. For instance, when a difference is noted between a faulted machine and a good machine at an externally observable point (i.e., the pads), the faulty machine is dropped from the simulation queue and the fault is "detected." If the bad machine has an X or Z compared to a 1 or 0 for the good machine, the fault is a "possible detect." Obviously, the more externally observable nodes a circuit has, the quicker bad machines get dropped from the simulation. Normally, only the good machine state is stored, with each node listing the fault machines that differ with the good machine. The different state is often small, which implies that there is a small amount of extra simulation to be done. In other words, most simulation for a faulty machine is exactly the same as the good machine. This is what concurrent simulation exploits. Fault collapsing occurs when two different faults result in the same faulty machine. This is noted, and one of the faulty machines may be dropped. Some machines perform static fault collapsing prior to simulation. For instance, an SA0 fault on the input of an inverter is the same as an SA1 fault at the output of the same inverter. With some fault simulators it is possible to create a fault dictionary. This is a cross reference that maps an observed fault to a set of possible internal faults. It is of use when the tester wishes to track down the actual internal failure (such as to perform yield improvement) rather than just cull the part.

Apart from software-based simulations, hardware-fault simulation accelerators that can provide a speedup over software-based simulators are also available.

7.2.7 Delay Fault Testing

The fault models we have dealt with to this point have neglected timing. Failures that occur in CMOS could leave the functionality of the circuit untouched, but affect the timing. For instance, consider the layout shown in

Micron Ex. 1037, p. 522
Micron v. YMTC
IPR2025-00119

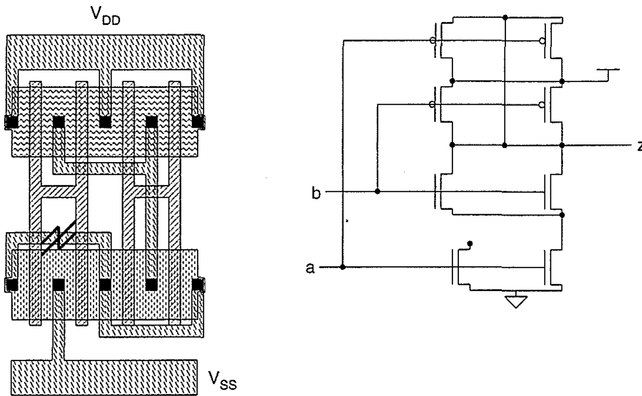


FIGURE 7.10 An example of a delay fault

Fig. 7.10 for a high-power NAND gate composed of paralleled n- and p-transistors. If the link illustrated was opened, the gate would still function, but with increased pull-down time. In addition, the fault now becomes sequential because the detection of the fault depends on the previous state of the gate and the simulation clock speed.

7.2.8 Statistical Fault Analysis

Conventional fault analysis can consume large CPU resources and take a long time. An alternative to this is what is called statistical fault analysis (STAFAN).[22] This method of fault analysis relies on estimating the probability that a fault will be detected. In summary, a fault free simulation is performed on a circuit in which some extra statistics are gathered by a modified simulator on a per-input vector basis. These are as follows:

- Zero-counter—The 0 count on each gate input when a 1→0 change of the output is detected.
- One-counter—The 1 count on each gate input when a 0→1 change of the output is detected.
- Sensitization-counter—incremented if the input change causes the output to be sensitized.
- Loop-counter—used to detect and deal with feedback.

The one-controllability of line l is given by

$$C1(l) = one\text{-}count/N,$$

where N is the number of vectors.

The zero-controllability is given by

$$C0(l) = zero\text{-}count/N.$$

TABLE 7.5 Statistical Fault Analysis 1 And 0 Observabilities

GATE TYPE	$B1(l)$	$B0(l)$
AND	$B1(m) \cdot \frac{C1(m)}{C1(l)}$	$B0(m) \cdot \frac{S(l) - C1(m)}{C0(l)}$
OR	$B1(m) \cdot \frac{S(l) - C0(m)}{C1(l)}$	$B0(m) \cdot \frac{C0(m)}{C0(l)}$
NAND	$B0(m) \cdot \frac{C0(m)}{C1(l)}$	$B1(m) \frac{S(l) - C1(m)}{C0(l)}$
NOR	$B0(m) \frac{S(l) - C1(m)}{C1(l)}$	$B1(m) \cdot \frac{C1(m)}{C0(l)}$
NOT	$B0(m)$	$B1(m)$

The one-level sensitization probability is

$$S(l) = \textit{sensitization-count}/N.$$

The observabilities are calculated by propagating from gate outputs to gate inputs. For common gates, Jain and Agrawal derive the one-observabilities ($B1$) and zero-observabilities ($B0$) for common gates, as shown in Table 7.5.

Methods also exist to deal with fan-out where two observabilities must be combined. Once these observability and controllability measures have been determined, the probability of fault detection may be calculated as follows:

$$D1(l) = B0(l) \,.\, C0(l),$$

where $D1(l)$ is the probability of detection that line l is SA1.

$$D0(l) = B1(l) \,.\, C1(l),$$

where $D0(l)$ is the probability of detection that line l is SA0.

From these values the fault coverage of the circuit may be calculated. The results of using this technique follow very closely the results generated by conventional fault simulation.

7.2.9 Fault Sampling

Another approach to fault analysis is known as fault sampling. This is used in circuits where it is impossible to fault every node in the circuit. Nodes are randomly selected and faulted. The resulting fault-detection rate may be sta-

Micron Ex. 1037, p. 524
Micron v. YMTC
IPR2025-00119

tistically inferred from the number of faults that are detected in the fault set and the size of the set. As with all probabalistic methods it is important that the randomly selected faults be unbiased. Although this approach does not yield a specific level of fault coverage, it will determine whether the fault coverage exceeds a desired level. The level of confidence may be increased by increasing the number of samples.

7.3 Design Strategies for Test

7.3.1 Design for Testability

The key to designing circuits that are testable are the two concepts that we have introduced called controllability and observability. Restated, controllability is the ability to set (to 1) and reset (to 0) every node internal to the circuit. Observability is the ability to observe either directly or indirectly the state of any node in the circuit.

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

- ad-hoc testing.
- scan-based approaches.
- self-test and built-in testing.

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

- random logic (multilevel standard-cell, two-level PLA).
- regular logic arrays (datapaths).
- memories (RAM, ROM).

7.3.2 Ad-Hoc Testing

Ad-hoc test techniques, as their name suggests, are collections of ideas aimed at reducing the combinational explosion of testing. Common techniques involve:

- partitioning large sequential circuits.
- adding test points.
- adding multiplexers.
- providing for easy state reset.

Micron Ex. 1037, p. 525
Micron v. YMTC
IPR2025-00119

Long counters are good examples of circuits that can be tested by ad-hoc techniques. For instance imagine you have designed an 8-bit counter and want to test it. Figure 7.11(a) shows a naive implementation in which the counter only has a *RESET* and a *CLOCK* input, with the terminal count (*TC*) being observable. The designer probably thought that a reset and 256 clock cycles, followed by the observation of *TC*, would be adequate for testing purposes. Apart from the nonobservability of the count value (*Q*<7:0>), the

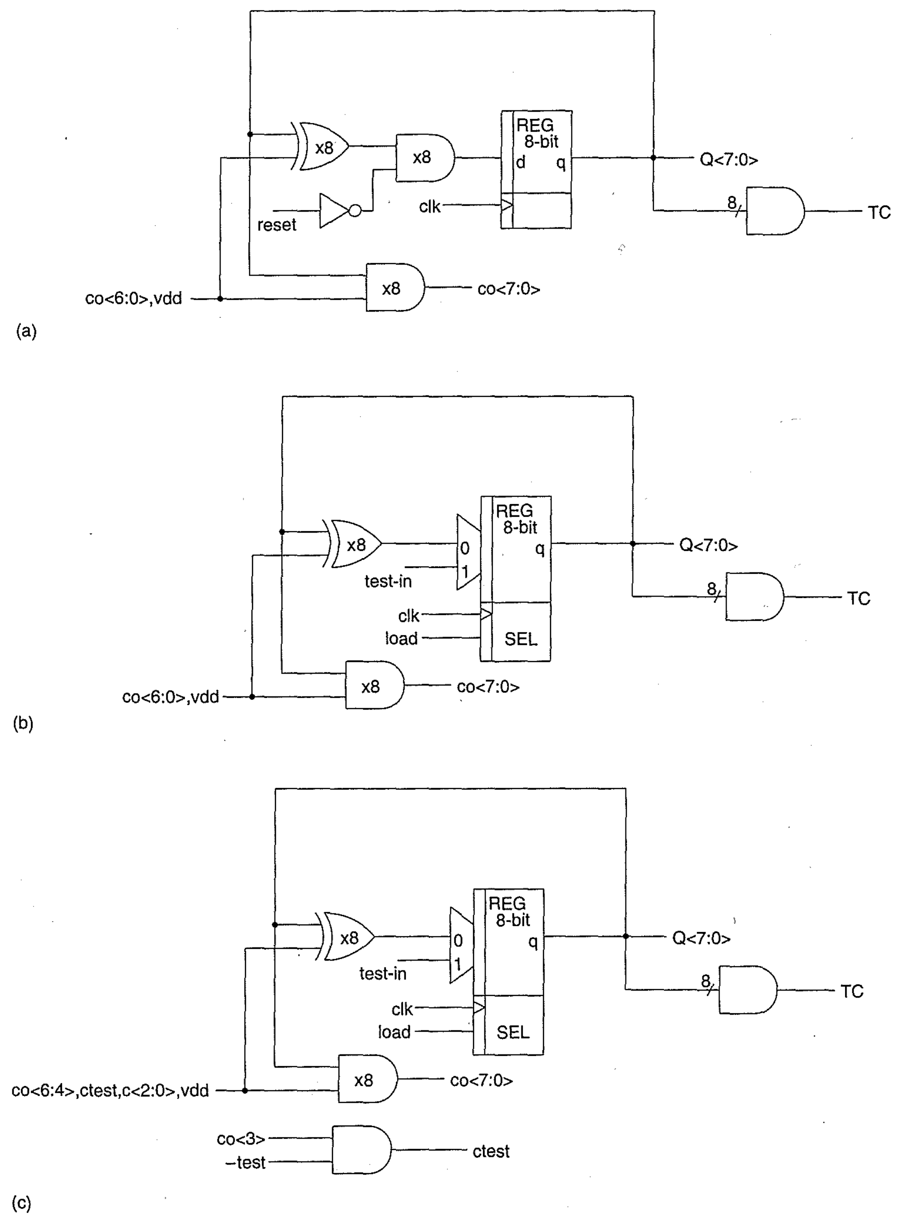
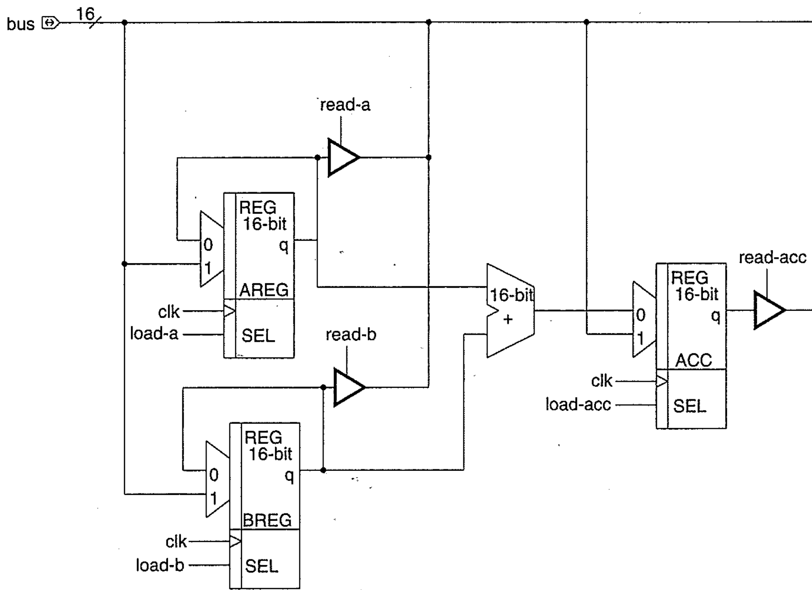


FIGURE 7.11 Ad-hoc test techniques applied to a counter

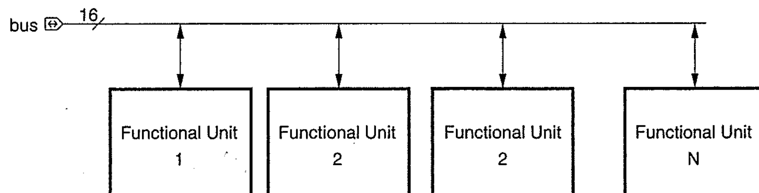
Micron Ex. 1037, p. 526
Micron v. YMTC
IPR2025-00119

main problem is the number of cycles required to test a single counter. Possible ad-hoc test techniques are shown in Fig. 7.11(b) and Fig. 7.11(c). In Fig. 7.11(b), a parallel-load feature is added to the counter. This enables the counter to be preloaded with appropriate values to check the carry propagation within the counter. Another technique is to reduce the length of each counter to, say, 4 bits, as shown in Fig. 7.11(c). This is achieved by having the test signal block the carry propagate at every 4-bit boundary. With this method 16 vectors exhaustively can test each 4-bit section. The carry propagate between 4-bit sections may be tested with a few additional vectors.

Another technique classified in this category is the use of the bus in a bus-oriented system for test purposes. This is shown on Fig. 7.12(a) for a very simple accumulator. Each register has been made loadable from the bus and capable of being driven onto the bus. Here the internal logic values that exist on a data bus are enabled onto the bus for testing purposes. A more general scheme is illustrated in Fig. 7.12(b), where the normally inaccessible inputs are set and the outputs are observed via the bus.



(a)



(b)

FIGURE 7.12 Bus-oriented test techniques

Micron Ex. 1037, p. 527
Micron v. YMTC
IPR2025-00119

Frequently, multiplexers may be used to provide alternative signal paths during testing. In CMOS, transmission gate multiplexers provide low area and speed overhead. Figure 7.13(a) shows a scheme called a Design for Autonomous Test[23], which uses multiplexers. Figure 7.13(b) shows the circuit configured for normal use, while Fig. 7.13(c) shows the circuit configured to test module A.

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

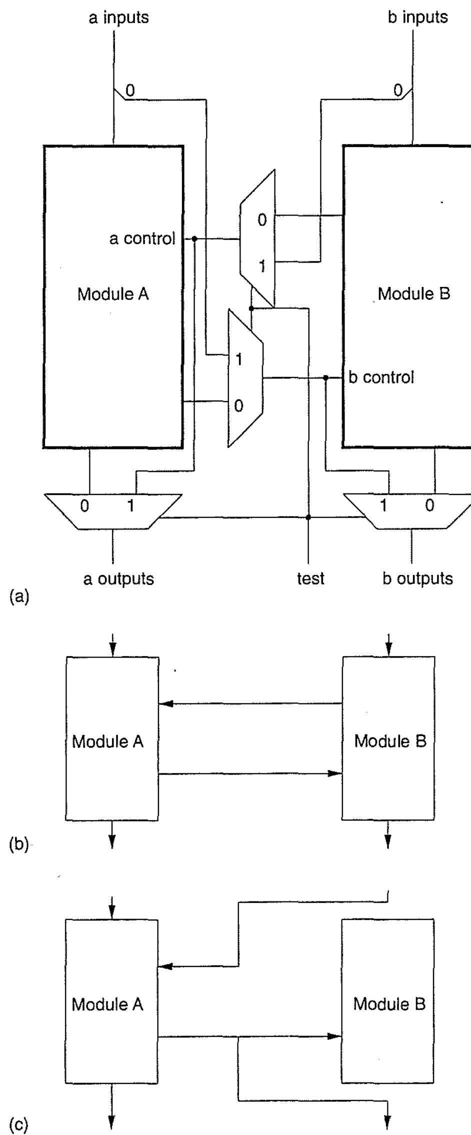


FIGURE 7.13 Multiplexer based testing

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

7.3.3 Scan-Based Test Techniques

A collection of approaches have evolved for testing that lead to a structured approach to testability. The approaches stem from the basic tenets of controllability and observability outlined earlier in this chapter.

7.3.3.1 Level Sensitive Scan Design (LSSD)

A popular approach is called Level Sensitive Scan Design, or the LSSD approach, introduced by IBM.[24,25,26] This is based on two tenets. First, that the circuit is level sensitive. According to Williams[27],

> A logic system is *level-sensitive* if, and only if, the steady state response to any allowed input state change is independent of the circuit and wire delays within the system. Also, if an input state change involves the changing of more than one input signal, then the response must be independent of the order in which they change. Steady state response is the final value of all logic gate outputs after all change activity has terminated.

The second principle of LSSD is that each register may be converted to a serial shift register.

The basic building block in LSSD is the Shift Register Latch, or SRL. A block-level implementation of a polarity-hold SRL is shown in Fig. 7.14(a). It consists of two latches, L_1 and L_2. L_1 has a serial data port, I, and an enable, A. It also has a data port, D, and an enable, C. When A is high, the value of L_1 (T_1) is set by the value of I, while when C is high, L_1 is set by D. A and C can not be simultaneously high. When signal B in L_2 is high, T_1 is passed to T_2. A gate-level implementation of the SRL is shown in Figs. 7.14(b) and 7.14(c). In normal operation, the D input is the normal input to the register, while the T_2 signal is the output. L_1 is the master while L_2 is the slave. SRLs may be connected in series by using the T_2 output and the I input of successive latches. During normal system operation, A is held low and C and B may be thought of as a two-phase nonoverlapping clock. When data is to be loaded into the SRLs or dumped out of the SRLs, A and B are used as a two-phase shift clock.

Figure 7.15(a) shows a typical LSSD scan system. An expanded view is shown in Fig. 7.15(b). The first rank of SRLs have inputs driven from a preceding stage and have outputs $QA1$, $QA2$, and $QA3$. These outputs feed a block of combinational logic. The output of this logic block feeds a second

Micron Ex. 1037, p. 529
Micron v. YMTC
IPR2025-00119

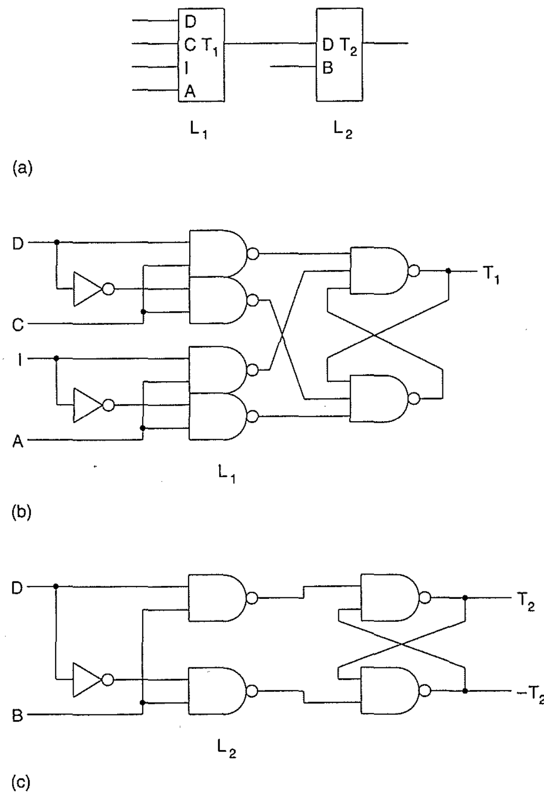


FIGURE 7.14 A shift register latch

rank of SRLs with outputs $QB1$, $QB2$, and $QB3$. Figure 7.15(c) shows a typical clocking sequence. Initially the *shift-clk* and $c2$ are clocked three times to shift data into the first rank of SRLs ($QA1$–3). $c1$ is asserted, and then $c2$ is asserted, clocking the output of the logic block into the second rank of SRLs ($QB1$–3). *shift-clk* and $c2$ are then clocked three times to shift $QB1$, $QB2$, and $QB3$ out via the serial-data-out line. Testing proceeds in this manner of serially clocking the data through the SRLs to the right point in the circuit, running a single "system" clock cycle and serially clocking the data out for observation. In this scheme, every input to the combinational block may be controlled and every output may be observed. In addition, running a serial sequence of 1's and 0's (such as 110010) through the SRLs can test them.

Test generation for this type of test architecture may be highly automated. ATPG techniques may be used for the combinational blocks, and as mentioned, the SRLs are easily tested. The prime disadvantage is the complexity of the SRLs (i.e., impacting density and speed).

7.3.3.2 Serial Scan

Level Sensitive Scan went to great pains to provide a hazard-free latching scheme. Faster clock speeds and design for smaller overhead in the registers

Micron Ex. 1037, p. 530
Micron v. YMTC
IPR2025-00119

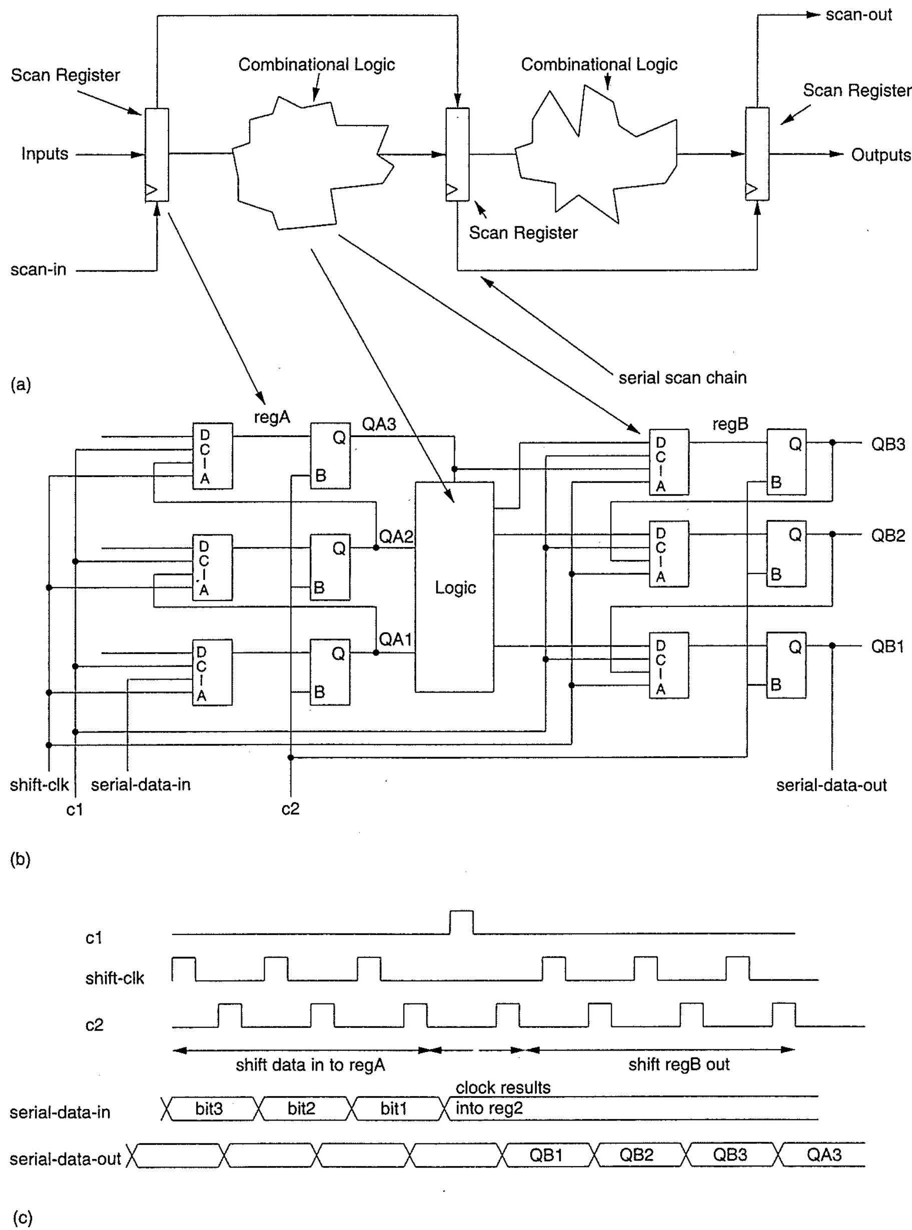


FIGURE 7.15 An LSSD scan chain: (a) basic architecture; (b) example circuit; (c) example timing

has led to simplifications in the SRL that give up a little on the hazard front but retain the scan principles mentioned above. (The hazard is moved inside the register, which with careful design can be guaranteed to be race free for a particular process and environmental characteristics.)

A schematic for a commonly used CMOS edge-sensitive scan-register is shown in Fig. 7.16. A MUX is added before the master latch in a conventional *D* register. *TE* is the Test Enable pin, and *TI* is the Test Input pin. When *TE* is enabled, *TI* is clocked into the register by the rising edge of *CLK*. Figure 7.17

Micron Ex. 1037, p. 531
Micron v. YMTC
IPR2025-00119

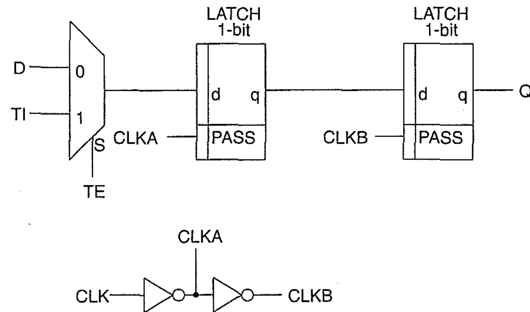


FIGURE 7.16 A typical CMOS scan-register

shows some circuit-level diagrams of CMOS SRL implementations. Figure 7.17(a) shows a frequently used implementation, which uses transmission gates to implement the multiplexers. The layout density overhead for this latch is minimal. In addition, because the addition of the testability MUX places two transmission gates in series, the increase in delay is minimized. Two further implementations of the input MUX are shown in Figs. 7.17(b) and 7.17(c). Figure 7.17(b) shows the addition of only two transistors and a single control line. A register so implemented does have the normal problems associated

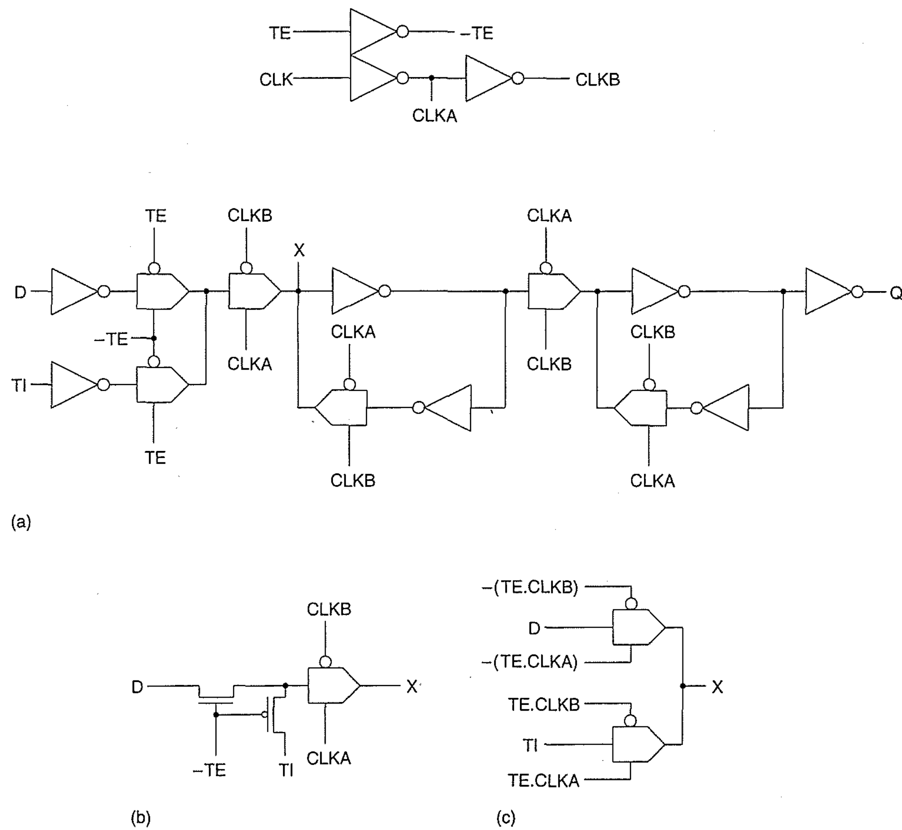


FIGURE 7.17 Various CMOS scan-latch options

Micron Ex. 1037, p. 532
Micron v. YMTC
IPR2025-00119

with used single-polarity transmission gates (see Chapter 5). Alternatively, the clocks may be gated, as shown in Fig. 7.17(c). While this minimizes transistors, it may lead to unacceptable hold-time constraints on the register. Because the signals applied to the master latch are delayed with respect to the main clock, the data has to be held for a longer time at the input.

7.3.3.3 Partial Serial Scan

Quite often in a design, one may not find it area- and speed-efficient to implement scan registers in every location where a register is used. This occurs, for instance, in signal-processing circuits where many pipeline registers might be used to achieve high speed. If these are in the data-flow section of the chip, then one can think of the logic that has to be tested as the logic with the pipeline registers removed. In this case only the input and output registers need be made scannable.[28] This technique of testing is known as partial scan, and depends on the designer making decisions about which registers need to be made scannable.

Consider the design shown in Fig. 7.18 (from Gupta et al.[29]). In a full-scan test strategy all registers would have to be scannable. A partial-scan design is shown in Fig. 7.18(a) where only two registers have been made scannable (*R*6 and *R*3). In addition, these registers have the ability to hold their state dependent on a HOLD control. The part of the circuit that is being tested and monitored by the scan registers (known as the *kernel*) is shown in Fig. 7.18(b). It may be proven that, by holding the vectors at the input of the kernel for three clock cycles, the kernel may be represented by the combinational-equivalent circuit shown in Fig. 7.18(c). This circuit may be used by an ATPG program to generate test vectors.

7.3.3.4 Parallel scan

One can imagine that serial-scan chains can become quite long, and the loading and unloading sequence can dominate testing time. An extension of serial scan is called random-access or parallel scan.[30]

The basic idea is shown in Fig. 7.19. Each register in the design is arranged on an imaginary (or real) grid where registers on common rows receive common data lines and registers in common columns receive common read- and write-control signals. In the figure, an array of 2-by-2 registers is shown. The *D* and *Q* signals of the registers are connected to the normal circuit connections. Any register output may be observed by enabling the appropriate column read line and setting the appropriate address on an output data multiplexer. Similarly, data may be written to any register.

Figure 7.20 shows a *D*-register implementation called a Cross-Controlled Latch.[31] It consists of a normal CMOS master-slave edge-triggered register augmented by two small n-transistors, N_1 and N_2. When *-test-write-enable* is high, *Probe[j]* is high, and *clk* is low, the value of node *Y* (*D*) may

Micron Ex. 1037, p. 533
Micron v. YMTC
IPR2025-00119

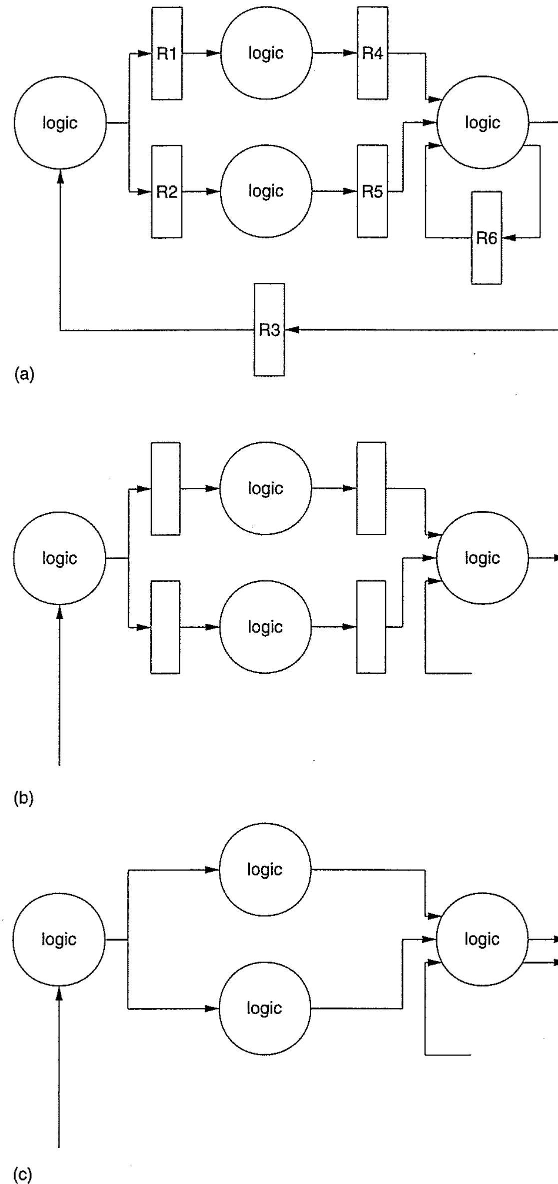


FIGURE 7.18 The application of scan techniques to employ partial scan: (a) pipeline circuit; (b) kernel of pipeline circuit; (c) combinational equivalent of kernel

be sensed on *Sense[i]* via transistor N_2. When *-test-write-enable* is low, *Probe[j]* is high, and *clk* is high, the value on *Sense[i]* can be driven onto node *Y*. This is seen immediately at the output of the register. The net effect on the register-timing parameters of the extra transistors is to slightly increase the minimum clock-pulse width. The area impact for an ASIC-based register is around 3%.

The large number of observable outputs (one for every register in the design) are compressed using signature analysis (see Section 7.3.4.1). The large number of observable outputs leads to very efficient concurrent-fault simulation.

Micron Ex. 1037, p. 534
Micron v. YMTC
IPR2025-00119

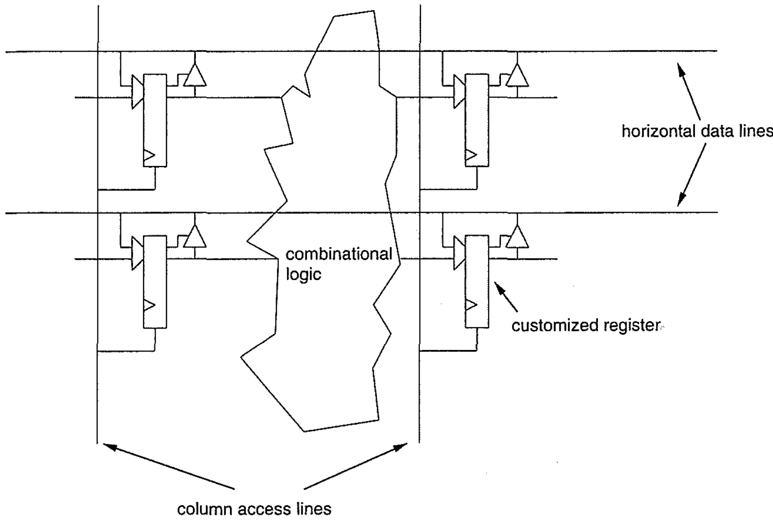


FIGURE 7.19 Parallel scan—basic structure

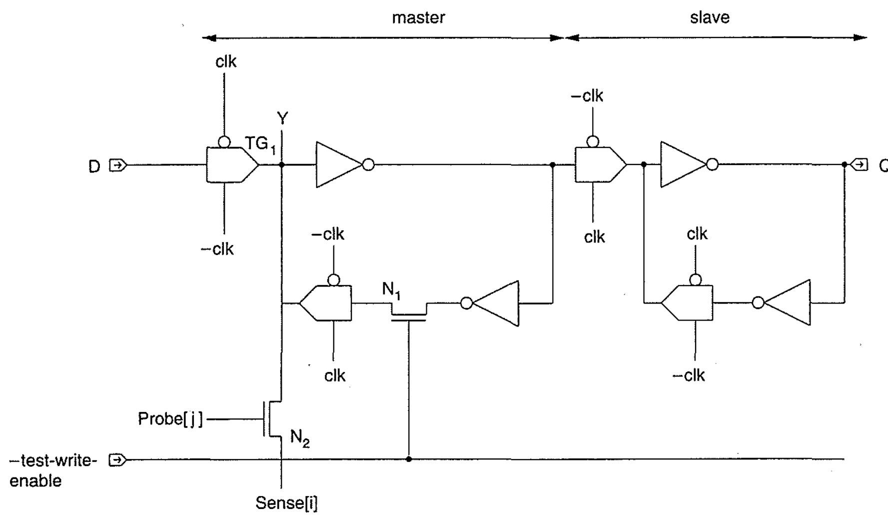


FIGURE 7.20 Parallel scan register (a cross-controlled latch)

7.3.4 Self-Test Techniques

Self-test and built-in test techniques, as their names suggest, rely on augmenting circuits to allow them to perform operations on themselves that prove correct operation.

7.3.4.1 Signature Analysis and BILBO

One method of incorporating a built-in test module is to use signature analysis[32,33] or cyclic-redundancy checking. This involves the use of a pseudo–

Micron Ex. 1037, p. 535
Micron v. YMTC
IPR2025-00119

random sequence generator (PRSG) to generate the input signals for a section of combinational circuitry and then using a signature analyzer to observe the output signals.

A PRSG implements a polynomial of some length N. It is constructed from a linear feedback shift register (LFSR), which is constructed, in turn, from a number of 1-bit registers connected in a serial fashion, as shown in Fig. 7.21. The outputs of certain shift bits are XORed and fed back to the input of the LFSR to calculate the required polynomial. For instance, in Fig. 7.21, the 3-bit shift register is computing the polynomial $f(x) = 1 + x + x^3$. For an n-bit LFSR, the output will cycle through $2^n - 1$ states before repeating the sequence. Tables for determining suitable shift registers may be found in Golumb.[34] A complete feedback shift register (CFSR) includes the zero state, which may be required in some test situations. Methods for designing these may be found in Wang and McCluskey.[35]

A signature analyzer is constructed by cyclically adding the outputs of a circuit to a shift register or an LFSR if successive logic blocks are to be tested in a like manner. A typical circuit is shown in Fig. 7.22(a). As each test vector is run, the incoming data is XORed with the contents of the LFSR. At the end of a test sequence, the LFSR contains a number, known as the *syndrome,* which is a function of the current output and all previous outputs. This can be compared with the correct syndrome (derived by running a test program on the good logic) to determine whether the circuit is good or bad.

Signature analysis can be merged with the scan technique to create a structure known as BILBO—for Built-In Logic Block Observation.[36]

A 3-bit register is shown with the associated circuitry. In mode D ($C0 = C1 = 1$), the registers act as conventional parallel registers. In mode A ($C0 = C1 = 0$), the registers act as scan registers. In mode C ($C0 = 1$, $C1 = 0$), the registers act as a signature analyzer or pseudo–random sequence generator (PRSG). The registers are reset if $C0 = 0$ and $C1 = 1$. Thus a complete test-generation and observation arrangement can be implemented, as shown in Fig. 7.22(b). In this case two sets of registers have been added in addition to some random logic to effect the test structure.

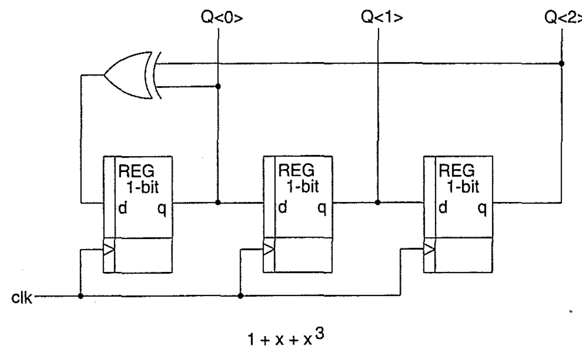
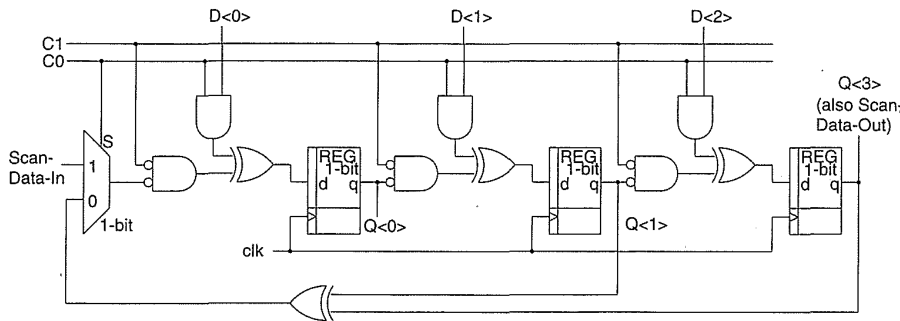


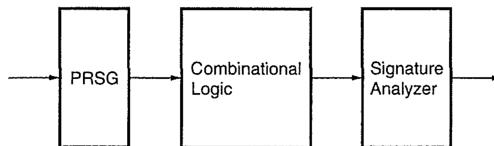
FIGURE 7.21 Pseudo–random sequence generator (PRSG)

Micron Ex. 1037, p. 536
Micron v. YMTC
IPR2025-00119



(a)

MODE	C_0	C_1	
A	0	0	Scan Mode
B	0	1	Reset
C	1	0	PRSG or Signature Analyzer
D	1	1	Parallel Registers



(b)

FIGURE 7.22 Built-in logic block observation (BILBO): (a) individual register; (b) use in a system

A chip set for FFT applications was designed with local testing based on pseudo–random pattern generation and signature analysis.[37] With a 28-bit pattern generator and a 17-bit signature at 10 MHz it took 26 seconds to test the part.

7.3.4.2 Memory Self-Test

Embedding self-test circuits for memories in higher-speed circuits not only may be the way of testing the structures at speed but can save on the number of external test vectors that have to be run. A typical read/write memory (RAM) test program for an M-bit address memory might be as follows[38,39]:

FOR i=0 to M-1 write($\overline{\texttt{data}}$)
FOR i=0 to M-1 read($\overline{\texttt{data}}$) then write(data)
FOR i=0 to M-1 read(data) then write($\overline{\texttt{data}}$)
FOR i=M-1 to 0 read($\overline{\texttt{data}}$) then write(data)
FOR i=M-1 to 0 read(data) then write($\overline{\texttt{data}}$)

`data` is 1 and $\overline{\texttt{data}}$ is 0 for a 1-bit memory or a selected set of patterns for an n-bit word. For an 8-bit memory `data`, might be x00, x55, x33, and x0F. An address counter, some multiplexers, and a simple-state machine result in a fairly low overhead self-test structure for read/write memories. Oshawa et al.[40] describe a 4-Mbit RAM with self-test. The self-test consists of 256K cycles that input a checkerboard pattern to test for cell-to-cell interference.

Micron Ex. 1037, p. 537
Micron v. YMTC
IPR2025-00119

This is followed by 256K cycles in which the data is read out. Then a complemented checkerboard is written and read. A total of 1 million cycles provide a test sufficient for system maintenance.

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

7.3.4.3 Iterative Logic Array Testing

Arrays of logic[41,42] present an interesting problem to the test architect because the replication can be used to advantage in reducing the number of tests. In addition, by augmenting the logic extremely high fault coverage rates are possible. An iterative logic array (ILA) is a collection of identical logic modules (such as an n-bit adder). An ILA is C-testable if it can be tested with a constant number of input vectors independent of the iteration count. An ILA is I-testable if a particular fault that occurs in any module as a result of an applied input vector is identical for all modules in the ILA. Assuming that only one module is faulty, the detection of a fault may be made by using an equality test on the ILA outputs.

7.3.5 IDDQ Testing

An increasingly popular method of testing for bridging faults is called IDDQ (V_{DD} supply current Quiescent) or current-supply monitoring.[43,44] This relies on the fact that when a complementary CMOS logic gate is not switching, it draws no DC current (except for leakage). When a bridging fault occurs, for some combination of input conditions a measurable DC I_{DD} will flow. Testing consists of applying the normal vectors, allowing the signals to settle, and then measuring I_{DD}. To be effective any circuits that draw DC power such as pseudo-nMOS gates or analog circuits have to be disabled. Because many circuits now require SLEEP modes to reduce power, this may not be a substantial additional overhead.

Because current measuring is slow, the tests must be run slower than normal, thus increasing the test time. However, this technique gives a form of indirect massive observability at little circuit overhead.

7.4 Chip-Level Test Techniques

In this chapter we have discussed the principles behind testing ICs, and covered some techniques aimed at making testing easier. In the past the design

Micron Ex. 1037, p. 538
Micron v. YMTC
IPR2025-00119

process was frequently divided between a designer who designed the circuit and a test engineer who designed the test to apply to that circuit. The advent of the ASIC, small design teams, the desire for reliable ICs, and rapid times to market have all forced the "test problem" earlier in the design cycle. In fact, the designer who is only thinking about what functionality has to be implemented and not about how to test the circuit will quite likely cause product deadlines to be slipped and in extreme cases products to be stillborn. In this section we will examine some practical methods of incorporating test requirements into a design. This discussion is structured around the main types of circuit structure that will be encountered in a digital CMOS chip.

7.4.1 Regular Logic Arrays

Partial serial scan or parallel scan is probably the best approach for structures such as datapaths. One approach that has been used in a Lisp microprocessor is shown in Fig. 7.23.[45] Here the input busses may be driven by a serially loaded register. These in turn may be used to load the internal datapath registers. The datapath registers may be sourced onto a bus, and this bus may be loaded into a register that may be serially accessed. All of the control signals to the datapath are also made scannable.

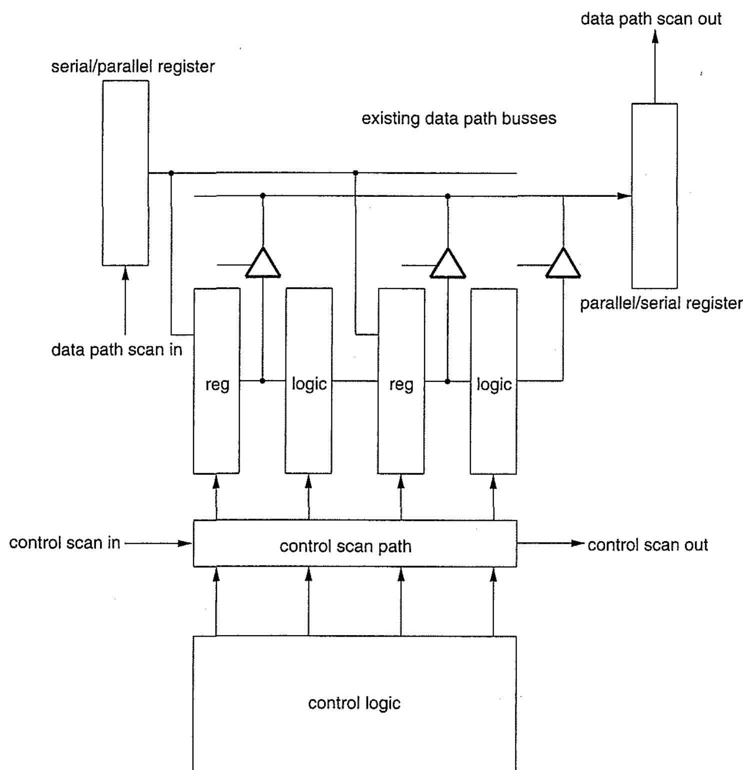


FIGURE 7.23 Datapath test scheme

7.4.2 Memories

Memories may use the self-testing techniques mentioned in Section 7.3.4.2. Alternatively, the provision of multiplexers on data inputs and addresses and convenient external access to data outputs enables the testing of embedded memories. It is a mistake to have memories indirectly accessible (i.e., data is written by passing through logic, data is observed after passing through logic, addresses can not be conveniently sequenced). Because memories have to be tested exhaustively, any overhead on writing and reading the memories can substantially increase the test time and, probably more significantly, turn the testing task into an effort in inscrutability.

7.4.3 Random Logic

Random logic is probably best tested via full serial scan or parallel scan.

7.5 System-Level Test Techniques

Up to this point we have concentrated on the methods of testing individual chips. Traditionally at the board level, "bed-of-nails" testers have been used to test boards. In this type of a tester, the board under test is lowered onto a set of test points (nails) that probe points of interest on the board. These may be sensed (the observable points) and driven (the controllable points) to test the complete board. At the chassis level, software programs are frequently used to test a complete board set. For instance, when a computer boots, it might run a memory test on the installed memory to detect possible faults.

The increasing complexity of boards and the movement to technologies like Multichip Modules (MCMs) and surface-mount technologies (with an absence of through-board vias) resulted in system designers agreeing on a unified scan-based methodology for testing chips at the board (and system level). This is called Boundary Scan.

7.5.1 Boundary Scan

7.5.1.1 Introduction

The IEEE 1149 Boundary Scan architecture[46] is shown in Fig. 7.24. In essence it provides a standardized serial scan path through the I/O pins of an IC. At the board level, ICs obeying the standard may be connected in a variety of series and parallel combinations to enable testing of a complete board or, possibly, collection of boards. The description here is a precis of the pub-

Micron Ex. 1037, p. 540
Micron v. YMTC
IPR2025-00119

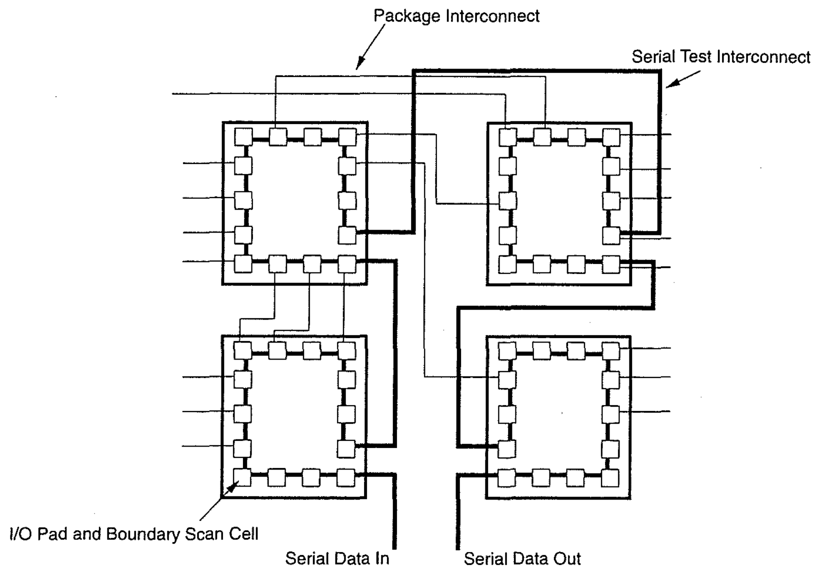


FIGURE 7.24 Boundary scan architecture

lished standard. The standard allows for the following types of tests to be run in a unified testing framework:

- Connectivity tests between components.
- Sampling and setting chip I/Os.
- Distribution and collection of self-test or built-in-test results.

7.5.1.2 The Test Access Port (TAP)

The Test Access Port (or TAP) is a definition of the interface that needs to be included in an IC to make it capable of being included in a Boundary-Scan architecture. The port has four or five single-bit connections, as follows:

- *TCK* (The Test Clock Input)—used to clock tests into and out of chips.
- *TMS* (The Test Mode Select)—used to control test operations.
- *TDI* (The Test Data Input)—used to input test data to a chip.
- *TDO* (The Test Data Output) used to output test data from a chip.

It also has an optional signal

- *TRST** (The Test Reset Signal) used to asynchronously reset the TAP controller; also used if a power-up reset signal is not available in the chip being tested.

Micron Ex. 1037, p. 541
Micron v. YMTC
IPR2025-00119

The *TDO* signal is defined as a tristate signal that is only driven when the TAP controller is outputting test data.

7.5.1.3 The Test Architecture

The basic test architecture that must be implemented on a chip is shown in Fig. 7.25. It consists of:

- the TAP interface pins.
- a set of test-data registers to collect data from the chip.
- an instruction register to enable test inputs to be applied to the chip.
- a TAP controller, which interprets test instructions and controls the flow of data into and out of the TAP.

Data that is input via the *TDI* port may be fed to one or more test data registers or an instruction register. An output MUX selects between the instruction register and the data registers to be output to the tristate *TDO* pin.

7.5.1.4 The TAP Controller

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

The state diagram is shown in Fig. 7.26. The state adjacent to each state transition is that of the *TMS* signal at the rising edge of *TCK*.

The reader is referred to the standard for complete descriptions of these states. It is probably best to understand them by examining a typical test sequence. Starting initially in the Test-Logic-Reset state, a low on TMS tran-

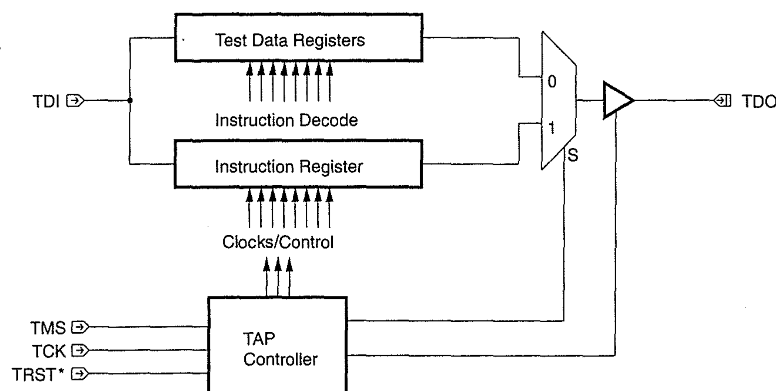


FIGURE 7.25 TAP architecture

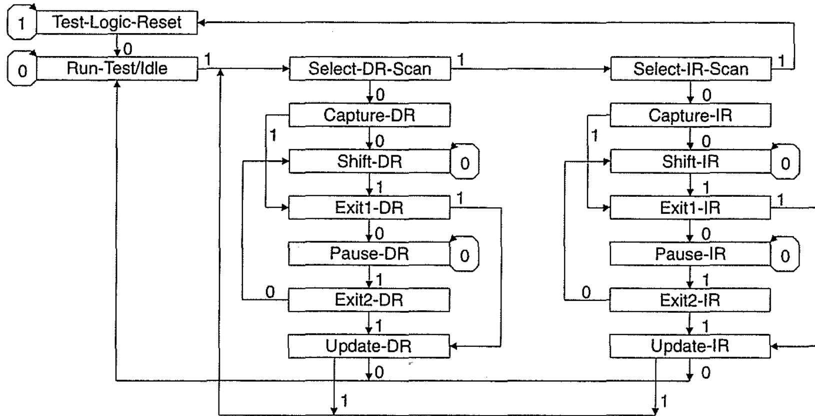


FIGURE 7.26 TAP controller state diagram

sitions the FSM to the Run-Test/Idle mode. Holding *TMS* high for the next three *TCK* cycles places the FSM in the Select-DR-Scan, Select-IR-Scan, and finally Capture-IR mode. In this mode two bits are input to the *TDI* port and shifted into the instruction register. Asserting *TMS* for a cycle allows the instruction register to pause while serially loading to allow tests to be carried out. Asserting *TMS* for two cycles, allows the FSM to enter the Exit2-IR mode on exit from the Pause-IR state and then to enter the Update-IR mode where the Instruction Register is updated with the new IR value. Similar sequencing is used to load the data registers.

A CMOS implementation of the Tap Controller based on that in the standard is shown in Fig. 8.89.

7.5.1.5 The Instruction Register (IR)

The instruction register has to be at least two bits long, and logic detecting the state of the instruction register has to decode at least three instructions, which are as follows:

- BYPASS—This instruction is represented by an IR having all zeroes in it. It is used to bypass any serial-data registers in a chip with a 1-bit register. This allows specific chips to be tested in a serial-scan chain without having to shift through the accumulated SR stages in all the chips.
- EXTEST—This instruction allows for the testing of off-chip circuitry and is represented by all ones in the IR.
- SAMPLE/PRELOAD—This instruction places the boundary-scan registers (i.e., at the chips' I/O pins) in the DR chain, and samples or preloads the chips I/Os.

Micron Ex. 1037, p. 543
Micron v. YMTC
IPR2025-00119

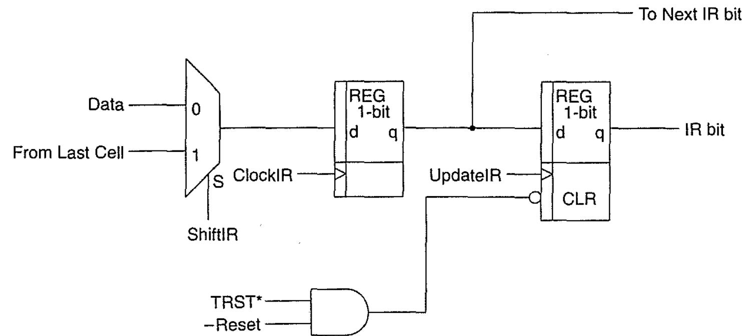


FIGURE 7.27 Instruction-register bit implementation

In addition to these instructions, the following are also recommended:

- INTEST—This instruction allows for single-step testing of internal circuitry via the boundary-scan registers.
- RUNBIST—This instruction is used to run internal self-testing procedures within a chip.

Further instructions may be defined as needed to provide other testing functions.

A typical IR bit is shown in Fig. 7.27.

7.5.1.6 Test-Data Registers (DRs)

The test-data registers are used to set the inputs of modules to be tested, and to collect the results of running tests. The simplest data-register configuration would be a boundary-scan register (passing through all I/O pads) and a bypass register (1-bit long). Figure 7.28 shows a generalized view of the data registers where one internal data register has been added. A multiplexer under the control of the Tap controller selects which particular data register is routed to the *TDO* pin.

7.5.1.7 Boundary Scan Registers

The boundary scan register is a special case of a data register. It allows circuit-board interconnections to be tested, external components tested, and the state of chip digital I/Os to be sampled. Apart from the bypass register, it is the only data register required in a Boundary Scan compliant part.

A single structure (in addition to the existing I/O circuitry) can be used for all I/O pad types, depending on the connections made to the cell. It consists of two multiplexers and two edge-triggered registers. Figure 7.29(a) shows this cell used as an input pad. Two register bits allow the serial shifting of data through the boundary-scan chain and the local storage of a data

Micron Ex. 1037, p. 544
Micron v. YMTC
IPR2025-00119

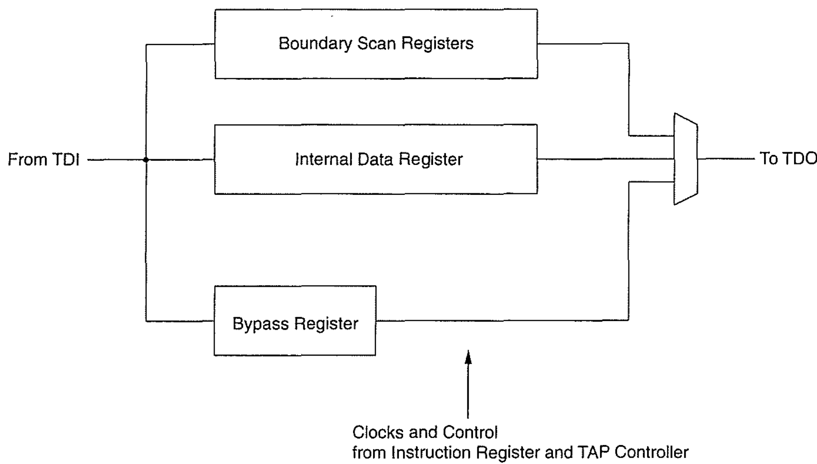


FIGURE 7.28 TAP data registers

bit. This data bit may be directed to internal circuitry in the INTEST or RUNBIST modes (*Mode* = 1). When *Mode* = 0, the cell is in EXTEST or SAMPLE/PRELOAD mode. A further multiplexer under the control of *ShiftDR* controls the serial/parallel nature of the cell. The signal *ClockDR* and *UpdateDR* generated by the Tap Controller load the serial and parallel register, respectively.

An output cell is shown in Fig. 7.29(b). When *Mode* = 1, the cell is in EXTEST, INTEST, or RUNBIST modes, communicating the internal data to

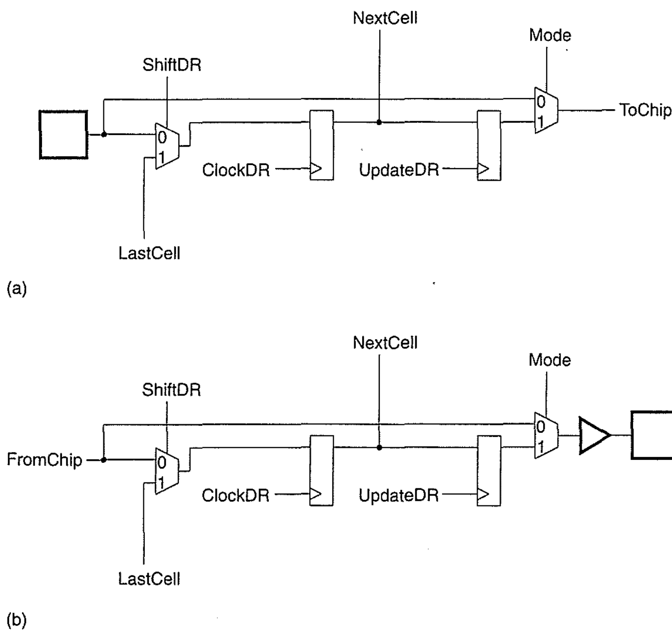


FIGURE 7.29 Boundary scan (a) input and (b) output cells

Micron Ex. 1037, p. 545
Micron v. YMTC
IPR2025-00119

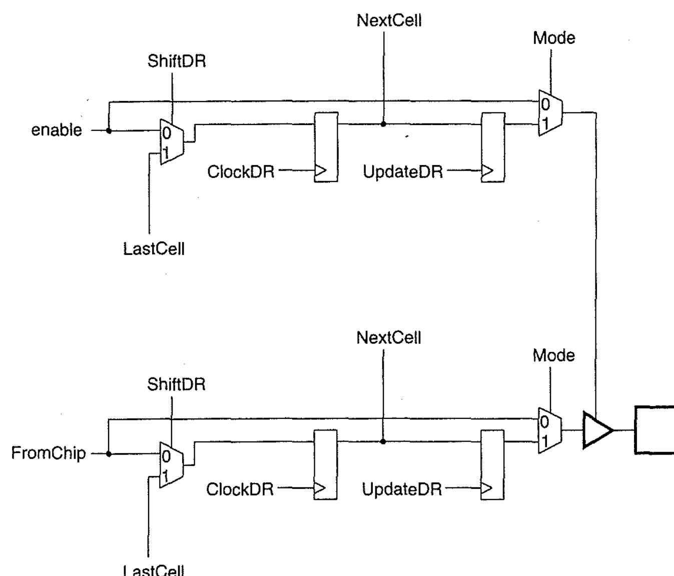


FIGURE 7.30 Boundary scan tristate cell

the output pad. When *Mode* = 0, the cell is in the SAMPLE/PRELOAD mode.

Two output cells may be combined to form a tristate boundary-scan cell, as shown in Fig. 7.30. The output signal and tristate-enable each have their own muxes and registers. The *Mode* control is the same for the output-cell example.

Finally, a bidirectional pin combines an input and tristate cell, as shown in Fig. 7.31.

7.5.2 Summary

At the system level, the Boundary Scan–Test Access Port approach has been summarized. There are, however, other related methods of dealing with testing at the system level. For instance, a boundary-scan method used in a multichip workstation, which uses a central controller rather than implementing the controller in each chip, has been reported.[47] A system designer has to trade off aspects, such as chip area versus implementation time, when deciding on a test strategy. However, the important thing is to *have* a strategy.

7.6 Layout Design for Improved Testability

In this chapter a number of models for failure were postulated and methods for detecting the faults in working circuits were proposed. We have already

Micron Ex. 1037, p. 546
Micron v. YMTC
IPR2025-00119

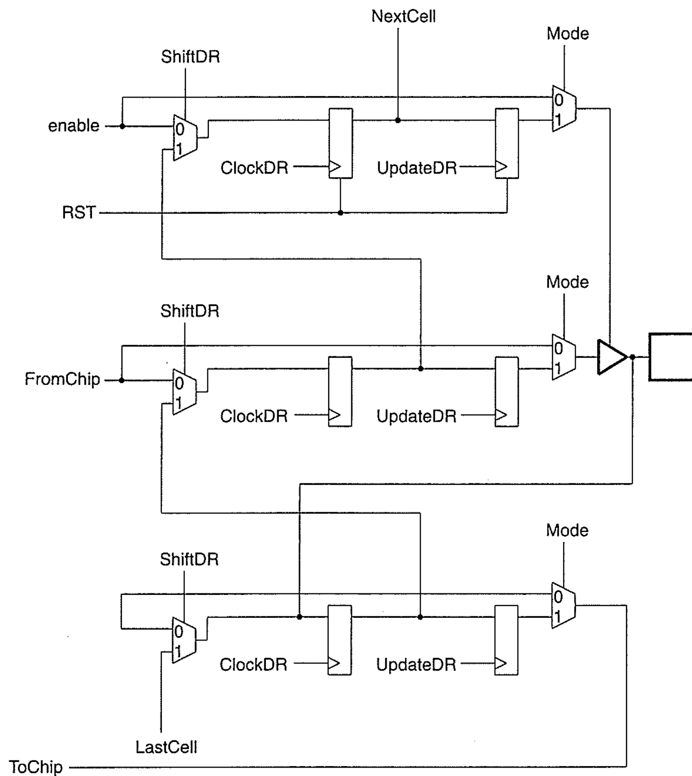


FIGURE 7.31 Boundary scan bidirectional cell

discussed a circuit-design technique to detect stuck-open faults in CMOS in Section 7.3.3.2. An interesting question arises. Can we construct the physical layouts to reduce the likelihood of such failures? This question has interested many researchers, and there is a body of literature that discusses the possible answers.[48–52]

In order to predict layout styles that improve testability, a designer has to have some idea of the nature and frequency of defects for a particular process. The types of defects that commonly occur may be divided into those that short together conductors and those that create open circuits. Shorts are possible intralayer in all layers used for connections, i.e., diffusion, polysilicon, metal1, metal2, and metal3, if used. The gate oxide may also short to the substrate or to either the source or drain. The source and drain regions may also short. Similarly for open circuits, all conducting layers might have open circuits. In addition, contacts may be misaligned, missing, or badly etched, leading to interlayer opens.

For open circuits, the ideas proposed in the literature to increase the immunity to open-circuit faults usually involve incorporating connection redundancy.

Micron Ex. 1037, p. 547
Micron v. YMTC
IPR2025-00119

7.7 Summary

This chapter has summarized the important issues in CMOS chip testing and has provided some methods for incorporating test considerations into chips from the start of the design. The importance of writing adequate tests for both functional verification and manufacturing verification can not be understated. It is probably the single most important activity in any CMOS chip design cycle and usually takes the longest time no matter what design methodology is used. If a single message should be left in the reader's mind after reading this chapter, it should be that a chip designer should be absolutely rigorous about the testing activity surrounding a chip project and that testing should rank first in any design trade-offs.

7.8 Exercises

1. Explain what is meant by a Stuck-At-1(SA1) fault and a Stuck-At-0 (SA0) fault.
2. How are sequential faults caused in CMOS? Give an example.
3. Explain the different kinds of physical faults that can occur on a CMOS chip, and relate them to typical circuit failures.
4. Explain the terms controllability, observability, and fault coverage.
5. Explain how serial-scan testing is implemented.
6. Explain how a pseudo–random sequence generator (PRSG) may be used to test a 16-bit datapath. How would the outputs be collected and checked?
7. Design a block diagram of a test generator for an $8 \times 4K$ static RAM.

7.9 References

1. T. W. Williams, "Design for testability," in *Computer Design Aids for VLSI Circuits* (P. Antogneti, D. O. Pederson, and H de Man, eds.), NATO AIS Series, The Netherlands: Martinus Nijhoff Publishers, 1986, pp. 359–416.
2. G. S. Visweswaran, Akhtar-uz-zaman M. Ali, Parang K. Lala, and Carlos R. P. Hartmann, "The effects of transistor source-to-gate bridging faults in complex CMOS gates," *IEEE Journal of Solid State Circuits,* vol. 26, no. 6, Jun. 1991, pp. 893–896.

Micron Ex. 1037, p. 548
Micron v. YMTC
IPR2025-00119

3. Thomas W. Williams and Kenneth P. Parker, "Design for testability—a survey," *Proceedings of the IEEE,* vol. 71, no. 1, Jan. 1983, pp. 98–112.
4. Anura P. Jayasumana, Yashwant K. Malaiya, and Rochit Rajsuman, "Design of CMOS circuits for stuck-open fault testability," *IEEE JSSC,* vol. 26, no. 1, Jan. 1991, pp. 58–61.
5. J. Galiay, Y. Crouzet, and M. Verginiault, "Physical versus logical fault models MOS LSI circuits: impact on their testability," *IEEE Transactions on Computers,* vol. C-29, no. 6, Jun. 1980, pp. 527–531.
6. Y. M. El-ziq and R. J. Cloutier, "Functional level test generation for stuck-open faults in CMOS VLSI," *Digest of Papers, IEEE International Test Conference,* Oct. 1981, pp. 536–546.
7. J. P. Roth, "Diagnosis of automata failures: a calculus and a method," *IBM Journal of Research and Development,* vol. 10, Jul. 1966, pp. 278–291.
8. *Ibid.*
9. Prabhakar Goel, "An implicit enumeration algorithm to generate tests for combinational logic circuits," *IEEE Transactions on Computers,* vol. c-30, no. 3, Mar. 1981, pp. 215–222.
10. Prabhakar Goel and Barry C. Rosales, "PODEM-X—an automatic test generation system for VLSI logic structures," *IEEE Proceedings of the 18th Design Automation Conference,* Jun. 1981, pp. 260–268.
11. H. Fuijiwara and T. Shimono, "On the acceleration of test generation algorithms," *IEEE Transactions on Computers,* vol. C-32, no. 12, Dec. 1983, pp. 1137–1144.
12. Michael H. Schulz, Erwin Trischler, and Thomas M. Sarfert, "SOCRATES: a highly efficient automatic test pattern generation system," *IEEE Transactions on CAD,* vol. 7, no. 1, Jan. 1988, pp. 126–137.
13. Noriyoshi Itazaki and Kozo Kinoshita, "Test pattern generation for circuits with tri-state modules by Z-algorithm," *IEEE Transactions on CAD,* vol. 8, no. 12, Dec. 1989, pp. 1327–1333.
14. John D. Calhoun and Franc Brglez, "A framework and method for hierarchical test generation," *IEEE Transactions on CAD,* vol. 11, no. 1, Jan. 1992, pp. 45–67.
15. Andre Ivanov and Vinod K. Agarwal, "Dynamic testability measures for ATPG," *IEEE Transactions on CAD,* vol. 7, no. 5, May 1988, pp. 598–608.
16. Lawrence H. Goldstein and Evelyn L. Thigpen, "SCOAP: Sandia controllability/observability analysis program," *Proceedings of the 17th Design Automation Conference,* Jun. 1980, pp. 190–196.
17. F. Brglez, P. Pownall, and R. Hum, "Application of testability analysis: from ATPG to critical delay path tracing," *Proceedings 1984 Test Conference,* Oct. 1984, pp. 705–712.
18. A. Lioy and M. Mezzalama, "On parameters affecting ATPG performance," *Proc. CompEuro 1987,* May 1987, pp. 394–397.
19. Srimat T. Chakradhar, Michael L. Bushnell, and Vishwani D. Agrawal, "Toward massively parallel automatic test generation," *IEEE Transactions on CAD,* vol. 9, no. 9, Sept. 1990, pp. 981–994.
20. Michael H. Schulz et al., *op. cit.*
21. E. G. Ulrich and T. Baker, "The concurrent simulation of nearly identical digital networks," *IEEE/ACM Proceedings of the 10th Design Automation Conference,* Jun. 1973, pp. 145–150.

Micron Ex. 1037, p. 549
Micron v. YMTC
IPR2025-00119

22. Sunil K. Jain and Vishwani D. Agrawal, "STAFAN: an alternative to fault simulation," *Proceedings of the ACM IEEE 21st Design Automation Conference,* June 1984, Albuquerque, N.M., pp. 18–23.
23. E. J. McCluskey and S. Bozorgui-Nesbat, "Design for autonomous test," *IEEE Transactions on Computers,* vol. C-30, no. 11, Nov. 1981, pp. 866–875.
24. E. B. Eichelberger and T. W. Williams, "A logic design structure for LSI testing," *IEEE/ACM Proceedings of the 14th Design Automation Conference,* June 1977, New Orleans, Louisiana, pp. 462–468.
25. E. B. Eichelberger and T. W. Williams, "A logic design structure for LSI testing," *Journal of Design Automation and Fault Tolerant Computing,* vol. 2, no. 2, May 1978, pp. 165–178.
26. S. DasGupta, E. B. Eichelberger, and T. W. Williams, "LSI chip design for testability," *Digest of Technical Papers, IEEE International Solid State Circuits Conference,* San Francisco, Feb. 1978, pp. 216–217.
27. T. W. Williams, *op. cit.*
28. Rajesh Gupta, Rajiv Gupta, and Melvin A. Breuer, "An efficient implementation of the BALLAST partial scan architecture," *IFIP Proceedings of the International VLSI '89 Conference,* Aug. 1990, Munich, pp. 133–142.
29. *Ibid.*
30. H. Ando, "Testing VLSI with random access scan," *IEEE/ACM Digest of Papers COMPCON 80,* Feb. 1980, pp. 50–52.
31. Susheel J. Chandra, Tom Ferry, Tushar Gheewala, and Kerry Pierce, "ATPG based on a novel grid-addressable latch element," *IEEE/ACM Proceedings of the 28th IEEE Design Automation Conference,* June 1991, San Francisco, Calif., pp. 282–286.
32. R. A. Frowerk, "Signature analysis—a new digital field service method," *Hewlett Packard Journal,* May 1977, pp. 2–8.
33. H. J. Nadig, "Signature analysis—concepts, examples and guidelines," *Hewlett Packard Journal,* May 1977, pp. 15–21.
34. S. W. Golumb, *Shift Register Sequences,* Revised Edition, Laguna Hills, Calif.: Aegean Park Press, 1982.
35. Laung-Terng Wang and Edward J. McCluskey, "Complete feedback shift register design for built-in self test," *Proceedings of 1986 IEEE International Conference on Computer-Aided Design (ICCAD-86),* Nov. 1986, Santa Clara, Calif., pp. 56–59.
36. B. Koenemann, J. Mucha, and G. Zwiehoff, "Built-in logic block observation techniques," *Digest 1979 IEEE Test Conference,* 79CH1509-9C, Oct. 1979, pp. 37–41.
37. John Fox, Giuseppe Surace, and Paul A. Thomas, "A self-testing 2-μm CMOS chip set for FFT applications," *IEEE JSSC,* vol. SC-22, no. 1, Feb. 1987, pp. 15–19.
38. Ravindra Nair, Staish M. Satte, and Jacob A. Abraham, "Efficient algorithms for testing semiconductor random-access memories," *IEEE Transactions on Computers,* vol. C-27, no. 6, June 1978, pp. 572–576.
39. Rob Dekker, Frans Beenker, and Loek Thijssen, "A realistic fault model and test algorithms for static random access memories," *IEEE Transactions on CAD,* vol. 9, no. 6, June 1990, pp. 567–572.
40. Takashi Oshawa, Tohru Furuyama, Yohji Watanabe, Hiroto Tanaka, Natsuki Kushiyama, Kenji Tsuchida, Yohsei Nagahama, Satoshi Yamano, Takeshi Tanaka, Satoshi Shinozaki, and Kenji Natori, "A 60ns 4-Mbit CMOS DRAM

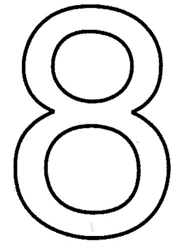
Micron Ex. 1037, p. 550
Micron v. YMTC
IPR2025-00119

with built-in self-test function," *IEEE JSSC,* vol. SC-22, no. 5, Oct. 1987, pp. 663–668.

41. W. H. Kautz, "Testing for faults in cellular logic arrays," *Proceedings of the 8th Annual Symposium on Switching and Automation Theory,* 1967, pp. 161–174.
42. Thirumalai Sridar and John P. Hayes, "Design of easily testable bit-sliced systems," *IEEE Transactions on Computers,* vol. C-30, no. 11, Nov. 1981, pp. 842–854.
43. John M. Acken, "Testing for bridging faults (shorts) in CMOS circuits," *Proceedings of the 20th IEEE/ACM Design Automation Conference,* June 1983, Miami Beach, Fla., pp. 717–718.
44. Kuen-Jong Lee and Melvin A. Breuer, "Design and test rules for CMOS circuits to facilitate IDDQ testing of bridging faults," *IEEE Transactions on CAD,* vol. 11, no. 5, May 1992, pp. 659–670.
45. Patrick Bosshart and Thirumalai Sridhar, "Test methodology for a 32-bit processor chip," *IEEE Digest of Technical Papers,* ICCAD-86, Nov. 1986, pp. 12–14.
46. IEEE Standard 1149.1: "IEEE standard test access port and boundary-scan architecture," New York: IEEE Standards Board.
47. Bulent I. Dervisoglu, "Application of scan hardware and software for debug and diagnostics in a workstation environment," *IEEE Transactions on CAD,* vol. 9, no. 6, June 1990, pp. 612–620.
48. J. Galiay, Y. Crouzet, and M. Vergniault, "Physical versus logical fault models MOS LSI circuits: impact on their testability," *IEEE Transactions on Computers,* vol. C-29, no. 6, pp. 527–531.
49. Wojciech Malay, "Realistic fault modeling for VLSI testing," *IEEE/ACM Proceedings of the 24th IEEE Design Automation Conference,* Miami Beach, Fla., 1987, pp. 173–180.
50. Siegmar Koeppe, "Optimal layout to avoid CMOS stuck-open faults," *IEEE/ACM Proceedings of the 24th Design Automation Conference,* Miami Beach, Florida, 1987, pp. 829–835.
51. Jose Joao H. T. de Sousa, Fernando M. Goncalves, and J. Paulo Teixeira, "Physical design of testable CMOS digital integrated circuits," *IEEE JSSC,* vol. 26, no. 7, July 1991, pp. 1064–1072.
52. Marc E. Levitt and Jacob A. Abraham, "Physical design of testable VLSI: techniques and experiments," *IEEE JSSC,* vol. 25, no. 2, April 1990, pp. 474–481.

Micron Ex. 1037, p. 551
Micron v. YMTC
IPR2025-00119

SUBSYSTEM DESIGN



8.1 Introduction

Most digital functions can be divided into the following categories:

- datapath operators.
- memory elements.
- control structures.
- I/O cells.

CMOS system design consists of partitioning the system to be designed into components that may be categorized into the above groups. Once those groupings have been determined, CMOS subsystems that implement those functions are designed. Many options exist that trade speed, density, programmability, ease of design, and many other variables. In this chapter we present a number of subsystems built with the circuits developed in Chapter 5. These subsystems may be used to build systems (chips, chip sets, or boards) of considerable complexity.

8.2 Datapath Operators

Datapath operators form an important subclass of VLSI circuit design that benefit from the structured design principles of hierarchy, regularity, modular-

Micron Ex. 1037, p. 552
Micron v. YMTC
IPR2025-00119

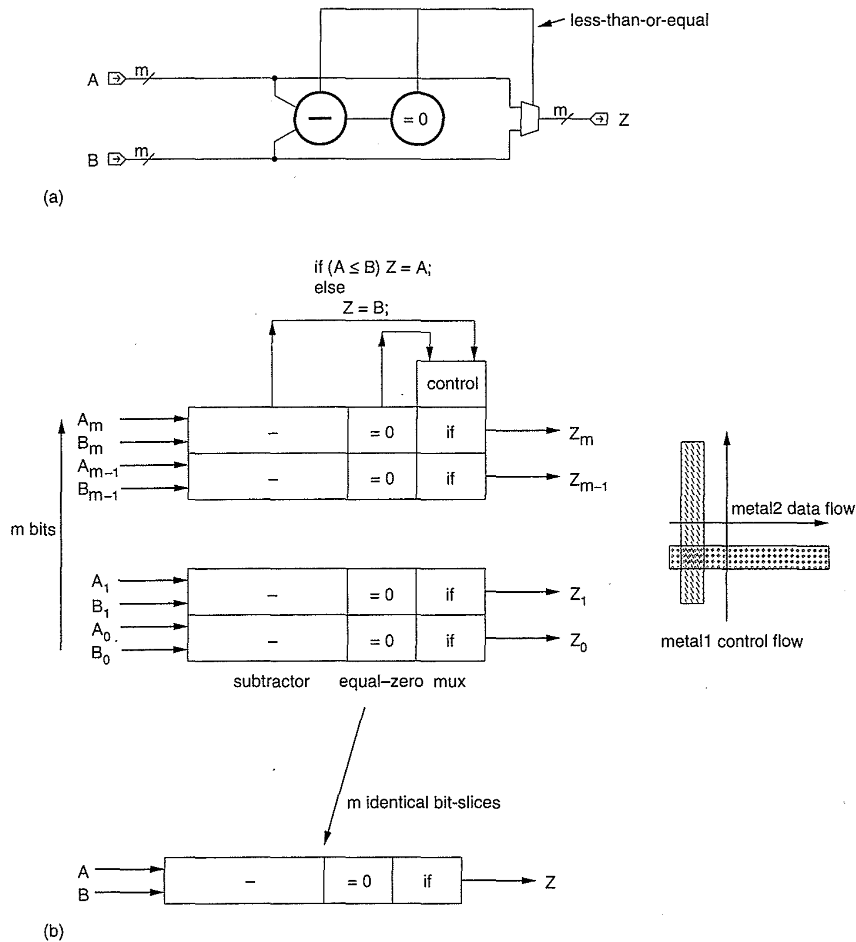


FIGURE 8.1 Datapath Example

ity, and locality. This arises because n-bit data is generally processed, which naturally leads to the ability to use n identical circuits to implement the function. In addition, data operations may generally be sequenced in time or space, which leads to the notion of physically placing linked data operators adjacent to each other. Generally, data may be arranged to flow in one direction, while any control signals are introduced in an orthogonal direction to the dataflow. This mirrors the physical reality of a CMOS chip, which usually has at least two good routing layers (i.e., metal1 and metal2, or metal2 and metal3).

Consider the magnitude comparator shown in Fig. 8.1(a). This may be implemented by the layout shown in Fig. 8.1(b), where data operators are arranged horizontally and data bits are arranged vertically. Data is relayed from operator to operator by horizontal wires (say, in metal2), while control information is routed vertically (say, in metal1). Datapaths allow optimization of the area of the layout by incorporating the regular routing strategy into the operator cell design. Usually, the data routing may be passed over

Micron Ex. 1037, p. 553
Micron v. YMTC
IPR2025-00119

the active circuitry, while the control signals are passed over or through the cells. Little area more than the basic area that the transistors take to implement a function is consumed. This efficiency is hard to achieve in random logic. The VLSI designer can exploit the regularity of datapaths by having to design one "bit-slice" of the design, which is a horizontal slice through the structure, shown in Fig. 8.1(b).

The rest of this section is devoted to describing a variety of data-processing elements that can be cast as datapaths.

8.2.1 Addition/Subtraction

Addition forms the basis for many processing operations from counting to multiplication to filtering. As a result, adder circuits that add two binary numbers are of great interest to digital system designers. A wide variety of adder implementations are available to serve different speed/density requirements. The truth table for a binary full adder was introduced in Chapter 1 and is reproduced in Table 8.1, along with some functions that will be of use during the discussion of adders.

A and *B* are the adder inputs, *C* is the carry input, *SUM* is the sum output, and *CARRY* is the carry output. The generate signal, *G* (*A.B*), occurs when a carry output (*CARRY*) is internally generated within the adder. When the propagate signal, *P* (*A* + *B*), is true, the carry-in signal (*C*) is passed to the carry output (*CARRY*) when *C* is true. (In some adders $A \oplus B$ is used as the *P* term because it may be reused to generate the sum term.)

8.2.1.1 Single-Bit Adders

Probably the simplest approach to designing an adder is to implement gates to yield the required majority logic functions. From the truth table these are:

$$SUM = ABC + A\overline{B}\,\overline{C} + \overline{A}\,\overline{B}C + \overline{A}B\overline{C}, \quad \textbf{(8.1)}$$

TABLE 8.1 Adder Truth Table

C	A	B	A.B(G)	A + B(P)	A ⊕ B	SUM	CARRY
0	0	0	0	0	0	0	0
0	0	1	0	1	1	1	0
0	1	0	0	1	1	1	0
0	1	1	1	1	0	0	1
1	0	0	0	0	1	1	0
1	0	1	0	1	1	0	1
1	1	0	0	1	1	0	1
1	1	1	1	1	0	1	1

Micron Ex. 1037, p. 554
Micron v. YMTC
IPR2025-00119

which may be factored as follows:

$$= C\,(AB + \bar{A}\,\bar{B}) + \bar{C}\,(A\bar{B} + \bar{A}B)$$
$$= A \oplus B \oplus C \qquad \textbf{(8.2)}$$

$$CARRY = AB + AC + BC,$$

which may be factored as follows

$$= AB + C\,(A + B). \qquad \textbf{(8.3)}$$

The gate schematic for the direct implementation of Eqs. (8.2) and (8.3) is shown in Fig. 8.2(a). This implementation uses a 3-input XOR gate. A transistor-level implementation is shown in Fig. 8.2(b). This uses a total of 32 transistors. An implementation that does not use XOR gates is shown in Fig. 8.3(a). This uses an alternative implementation that is achieved by realizing that the *CARRY* term may be reused in the *SUM* term as a common subexpression. In this implementation, shown in Fig. 8.3(b),

$$SUM = ABC + (A + B + C)\,\overline{CARRY}$$
$$= ABC + (A + B + C)\,\overline{(AB + C\,(A + B))}\,. \qquad \textbf{(8.4)}$$

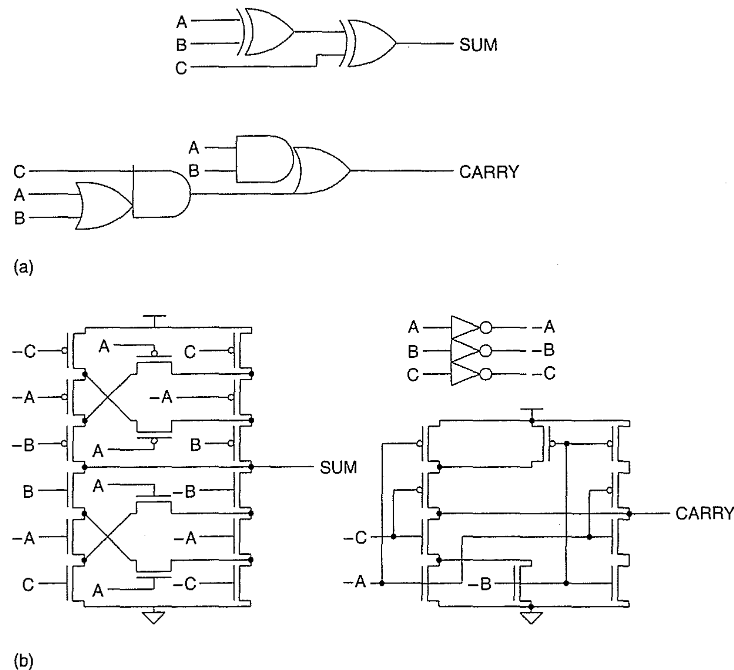


FIGURE 8.2 Single-bit adder schematic (3-input XOR)

Micron Ex. 1037, p. 555
Micron v. YMTC
IPR2025-00119

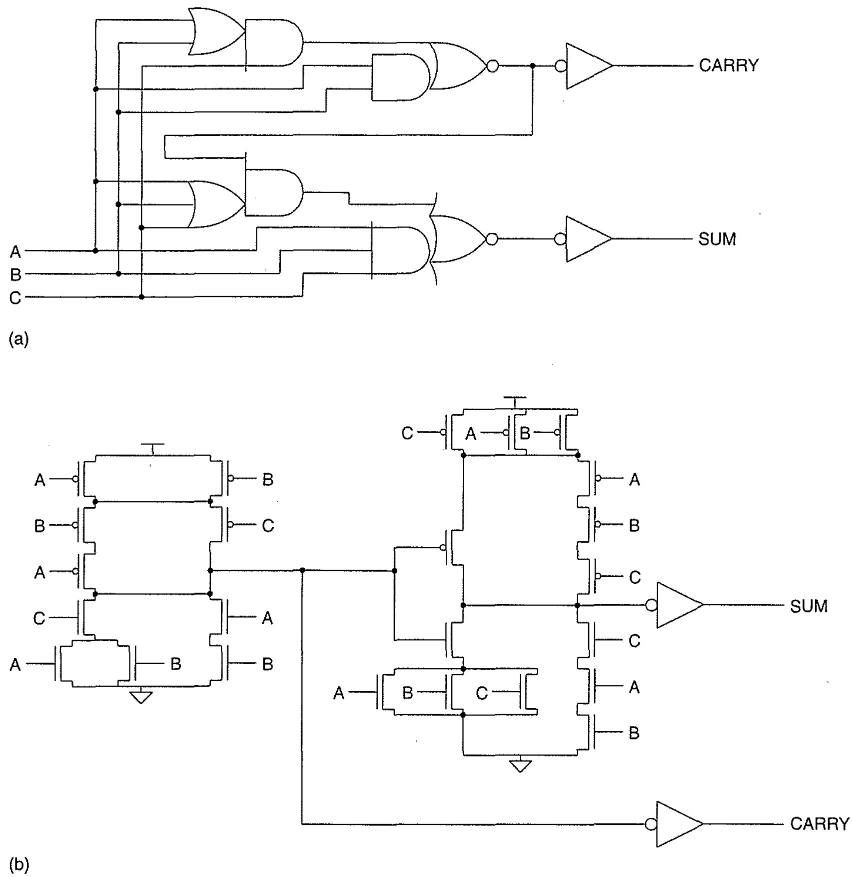


FIGURE 8.3 Single-bit adder schematic (cascaded logic gates)

The transistor schematic for this implementation is shown in Fig. 8.3(b). It uses 28 transistors.

8.2.1.2 Bit-Parallel Adder

An n-bit adder may be constructed by cascading n 1-bit adders, as shown in Fig. 8.4(a). This is called a Ripple Carry Adder. The inputs are n-bit A and B values. The *CARRY* signal of stage i is fed to the C signal of stage $i + 1$ and the *SUM* signal forms the n-bit output. The nth bit of the *SUM* indicates the sign of the result, while the nth *CARRY* signal indicates whether an overflow condition has occurred. Because the carry-output signal (*CARRY*) is used in the generation of *SUM* in the circuit shown in Fig. 8.3(a), *SUM* will be delayed with respect to *CARRY*. In the case of an n-bit parallel adder, the carry delay has to be minimized, because the delay associated with the adder is $T_n = nT_c$, where T_n is the total add time, n is the number of stages, and T_c is the delay of one carry stage. To optimize the carry delay, the inverter at the output of the carry gate can be omitted. In this case, every other stage oper-

Micron Ex. 1037, p. 556
Micron v. YMTC
IPR2025-00119

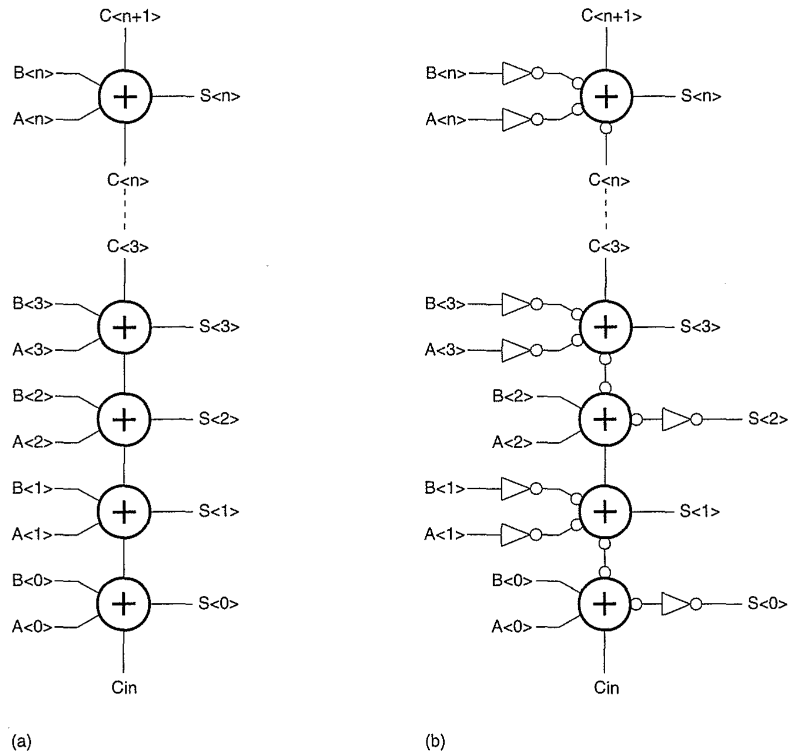


Figure 8.4 Parallel adder implementations

ates on complement data, as shown in Fig. 8.4(b). This may result in a significant decrease in carry delay. The delay in inverting the adder inputs or sum outputs is finessed out of the critical-ripple carry path.

An n-bit subtractor may be constructed by inverting one operand to an n-bit adder and adding 1 to the adder via the carry input, as shown in Fig. 8.5(a). An adder/subtractor may be constructed from XOR gates and adders, as shown in Fig. 8.5(b).

The transistor schematic for the adder in Fig. 8.3(a) is shown redrawn in Fig. 8.6(a). The propagate term ($A + B$) and generate term ($A.B$) can be clearly seen. To aid in a uniform layout, the p-chain is not the exact dual of the n-chain. It is left to the reader to verify the equivalence. Figure 8.6(b) shows how the transistors in the carry stage might be sized to optimize the delay through the carry stage. Sizing up the transistors in the carry gate while keeping the other transistors small decreases the effective load of these transistors and any parasitic routing capacitance. Using the styles of layout presented so far, two possible mask layouts for the combinational adder are depicted in Fig. 8.7 (also Plate 7). The choice of aspect ratio would depend very much on the environment. In a standard-cell environment, the layout in Fig. 8.7(a) might be appropriate where a single row of n- and p-transistors is used. The routing for the A, B, and C inputs is shown inside the cell although it is quite possible it could be placed outside the cell because external routing

Micron Ex. 1037, p. 557
Micron v. YMTC
IPR2025-00119

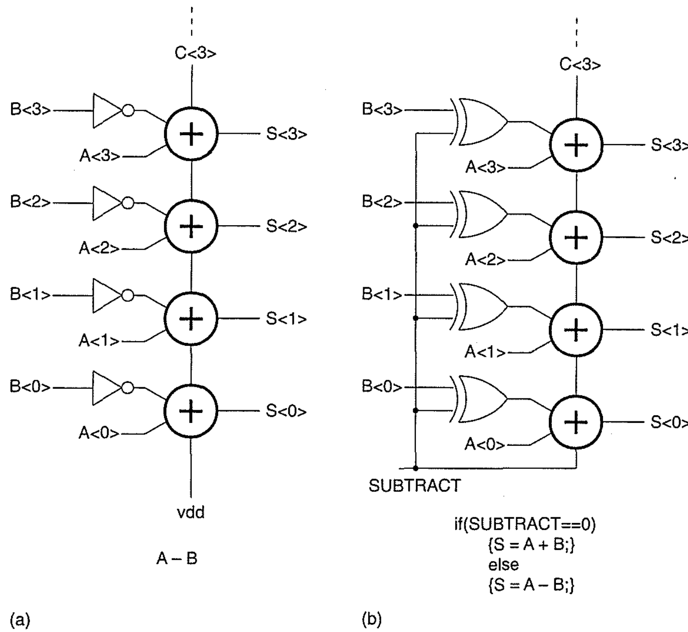


FIGURE 8.5 Arithmetic operators: (a) subtractor; (b) adder/subtractor

tracks have to be assigned to these signals anyway. Figure 8.7(b) shows a layout that might be appropriate for a datapath. Here the transistors are rotated and all of the wiring is completed in polysilicon and metal1. This allows metal2 bus lines to pass over the cell horizontally. In addition, the size of the transistors in the adder may be increased without impacting the bit-pitch (height) of the datapath. The following optimizations may be made to the combinational adder (Fig. 8.6):

1. Arrange the transistors switched by the carry in signal (C) close to the output. This will enable the input signals to settle the gate such that the C transistors are least influenced by body effect.
2. Make all transistors in the sum gate whose gate signals are connected to *CARRY* minimum size. This minimizes the capacitive load on this signal. Keep routing on this signal to a minimum and minimize the use of diffusion as a routing layer.
3. Sizing of series transistors can be determined by simulation. It may or may not pay to increase the size of the series n-transistors and p-transistors. For instance, it may not pay to increase the size of the series transistors connected to A and B in the carry gate in a ripple-carry adder, because these signals will have time to settle in the upper bits of the adder while the carry is rippling. It may be of advantage to increase the size of the C transistors in the carry gate to override the effects of stray capacitance. For a parallel adder, the *SUM*

Micron Ex. 1037, p. 558
Micron v. YMTC
IPR2025-00119

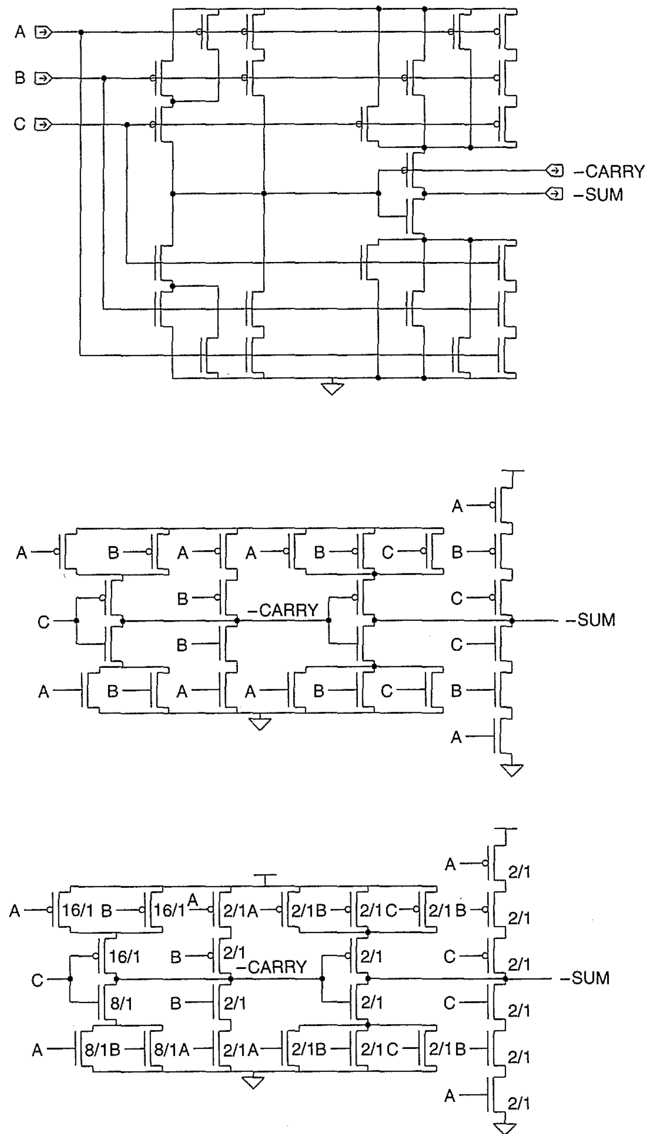


Figure 8.6 Optimized combinational adder schematic

gate transistors may be made minimum size, while for a serial adder the *CARRY* and *SUM* delays would have to be more balanced.

8.2.1.3 Bit-Serial Adders, Carry-save Addition, and Pipelining

Rather than construct a ripple carry adder, a serial adder, shown in Fig. 8.8, may be constructed. This uses a single adder and constructs the *SUM*

Micron Ex. 1037, p. 559
Micron v. YMTC
IPR2025-00119

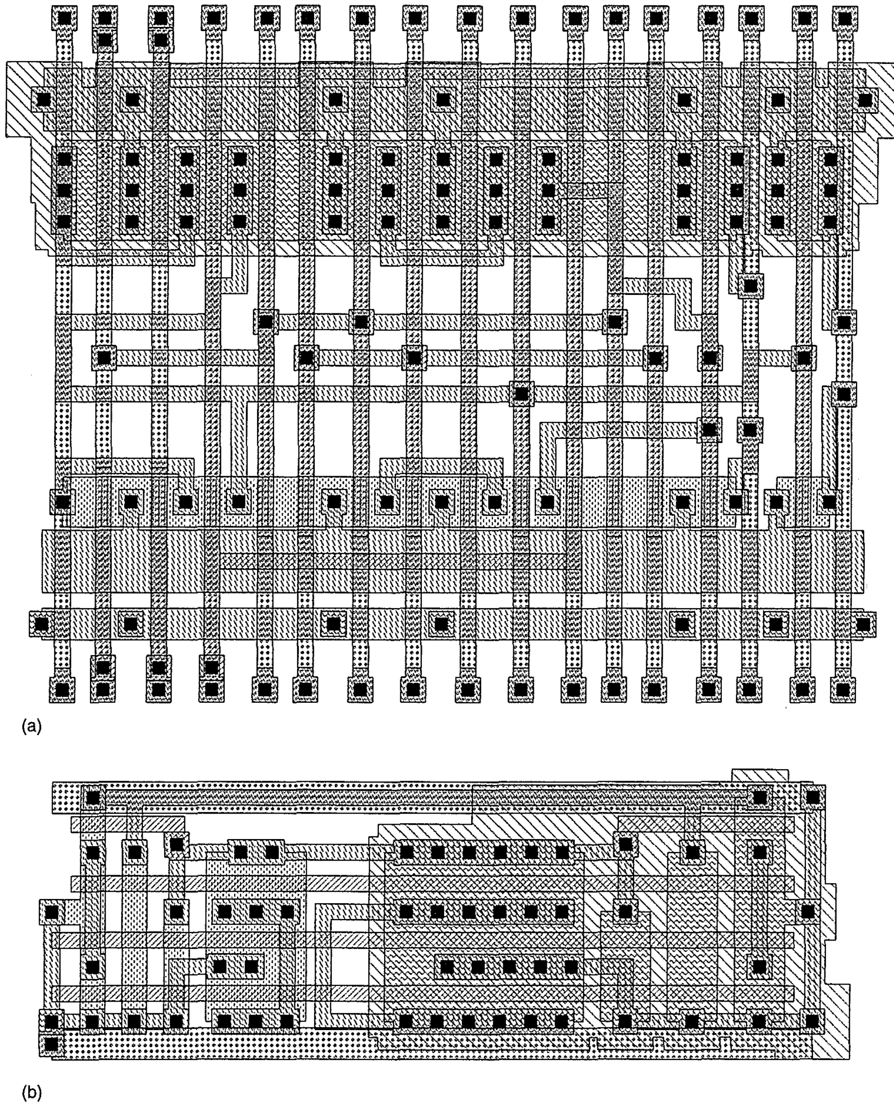


FIGURE 8.7 Combinational adder layouts: (a) standard cell; (b) datapath

sequentially. At time *t*, the *SUM* is calculated and the *CARRY* stored in a register. At time $t + 1$, the sum uses *CARRY[t]* to calculate a new *SUM*.

$$CARRY[t+1] = A[t+1].B[t+1] + C[t].(A[t+1] + B[t+1])$$

$$SUM[t+1] = CARRY[t+1].(A[t+1] + B[t+1] + C[t]) + A[t+1].B[t+1].C[t] \quad \textbf{(8.5)}$$

The two inputs to the adder are stored in n-bit registers. The *SUM* output is stored in an n-bit result register. An illustrative add cycle is shown in Fig. 8.8. Addition is commenced by clearing the carry register. Then the

Micron Ex. 1037, p. 560
Micron v. YMTC
IPR2025-00119

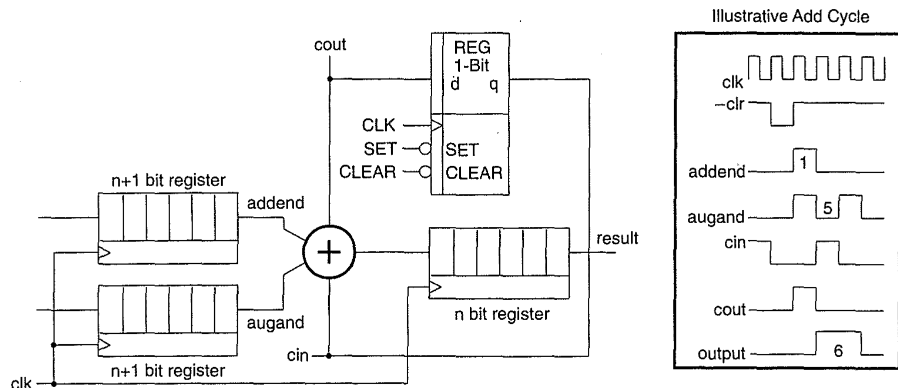


Figure 8.8 Bit-serial adder implementation

operands are serially applied to the inputs of the adder, the least significant bit first. The example shows 1 added to 5 to form 6 at the output. It takes n clock cycles to complete an n-bit add. In a serial adder, equal *SUM* and *CARRY* delays are advantageous, because these delays determine the fastest clock frequency at which the adder can operate.

Bit-serial architectures have been used successfully for a variety of signal-processing applications, especially with technologies in the 2–5µ range.[1] Reasons for using bit-serial architectures include reduced signal routing (1-bit signals instead of n-bit signals), reduced module sizes, and higher-speed operation (one adder and a register rather than an n-bit adder). Multilevel-metal CMOS technologies have largely solved the signal routing problems while more advanced processes have drastically reduced the size and increased the speed of adders and registers to the point where the design problems lie elsewhere (for instance, in correctly completing a large design on schedule). However, the general principle of breaking an n-bit addition into smaller additions may be applicable to current design situations. Apart from bit-serial adders, nibble (4-bit) and byte (8-bit) adders are frequently used. The reason for using reduced-size adders might range from size to power dissipation considerations.

Adders, such as the 1-bit serial adder shown in Fig. 8.8, where both the carry and sum are registered on each cycle, are often called *carry-save adders* (CSAs).[2] This can be extended to an n-bit adder by registering n carries and n sums. The carries are left shifted, with a new carry input introduced to the D of the LSB carry register and the carry output available at the MSB carry register Q. An n-bit CSA would have $2n$ registers. Figure 8.9 illustrates a circuit which uses two 4-bit CSAs, which is representative of a structure that might be used in a digital filter. The inputs *SIN<3:0>* and *CIN<2:0>* are added to constant *A<3:0>* in the left column (first rank) of CSA bits, and then *B<3:0>* is added in the next column (second rank). Each bit of a CSA has the binary output encoded in the sum and carry of each bit. The carry output of each CSA stage is left shifted to feed the carry input of

Micron Ex. 1037, p. 561
Micron v. YMTC
IPR2025-00119

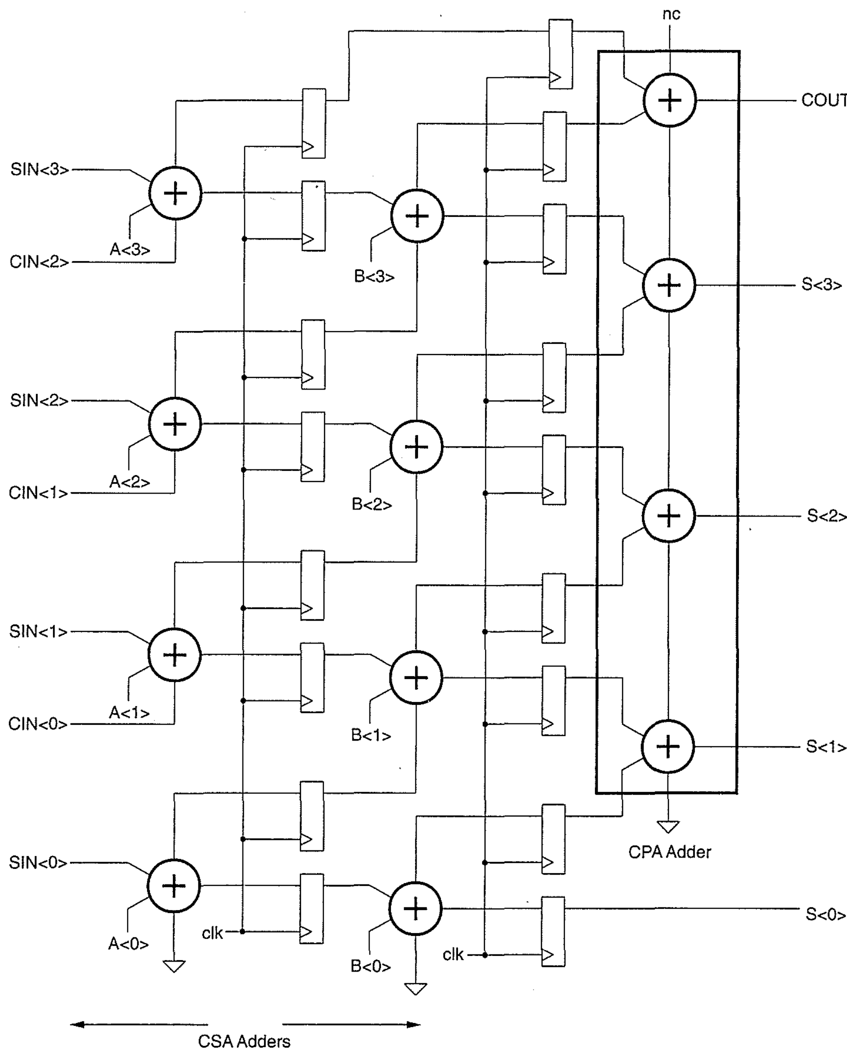


FIGURE 8.9 Carry-save adder (CSA) example

the next rank. The binary output may be extracted by feeding the sum and carry of each bit of the CSA to the inputs of a *carry-propagate adder* (CPA), as illustrated by the ripple-carry adder in Fig. 8.9 (or Fig. 8.4a). Usually a different, faster, architecture is used for the final carry-propagate adder. In applications such as filtering where many additions have to occur and many n-bit adders have to be used, use of cascaded carry-save adders reduces the critical path to the sum of the clock to Q time of the register, the adder delay, and the setup time into the register. Current CMOS processes allow operation in excess of 200 MHz, and operation above 1 GHz is very close. Figure 8.10 shows a CPA adder structure that can be used for the CPA shown in Fig. 8.9. Registers are used at the input and output of the CPA to ensure that the inputs arrive at the same time as the carry and that the outputs all appear

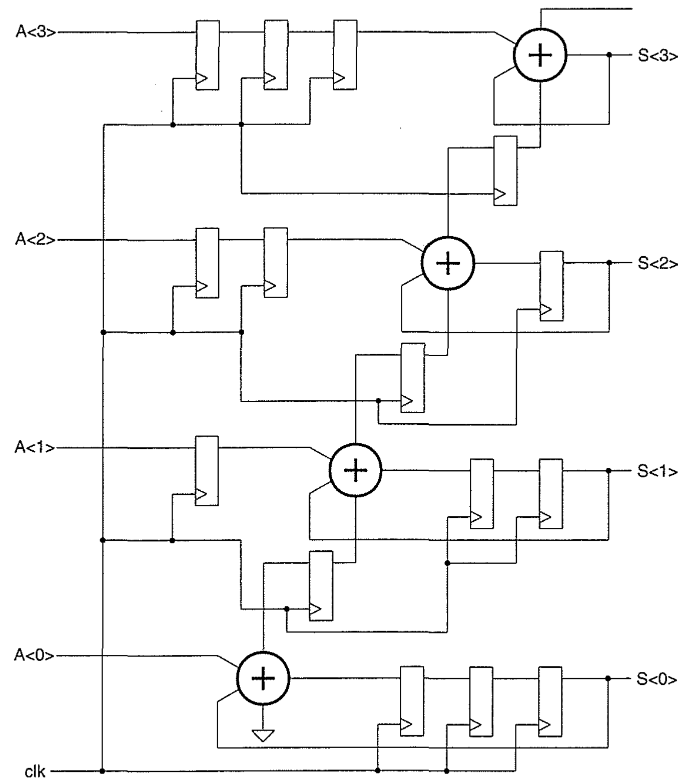


Figure 8.10 Pipelined carry-propagate (CPA) example

at the correct time. Figures 8.9 and 8.10 illustrate two methods of increasing the speed of a basic n-bit adder by the use of *pipelining*. The cost of pipelining is *latency;* that is, the time it takes from when operands are introduced to the data processing element to when outputs are available from the module. The adder in Fig. 8.9 has a latency of two clock cycles, while the adder in Fig. 8.10 has a latency of three cycles. A filter built with k CSAs would have a latency of $k + 3$ cycles. The *throughput* is k adds/cycle. Latency is usually not important in DSP applications, such as filtering, but is important in applications such as microprocessors where for a variety of reasons (including control) an add operation (32 bits or more) has to occur in a single clock cycle. On the other hand, throughput is all important to DSP applications. In the rest of this section some alternative techniques for improving adder speed will be introduced. In so doing, we will discover some classical examples of trading space for time. In other words, by increasing the size of a data element, we can often improve the speed.

8.2.1.4 Transmission-Gate Adder

A rather different implementation of an adder uses a novel exclusive-or (XOR) gate. The schematic for this XOR gate is shown in Fig. 8.11. As a

Micron Ex. 1037, p. 563
Micron v. YMTC
IPR2025-00119

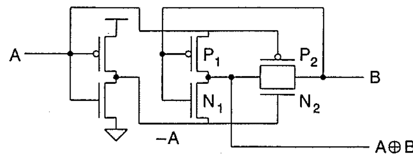


FIGURE 8.11 Transmission-gate XOR (tiny XOR)

point to note, switch-level simulators have problems with this gate. The operation of the gate is explained as follows:

1. When signal A is high, $-A$ is low. Transistor pair P_1 and N_1 thus act as an inverter, with $-B$ appearing at the output. The transmission gate formed by transistor pair P_2 and N_2 is open.
2. When signal A is low, $-A$ is high. The transmission gate (P_2, N_2) is now closed, passing B to the output. The inverter (P_1, N_1) is partially disabled (level reduced B passed to output by P_1, N_1).

Thus this transistor configuration forms a 6-transistor XOR gate. By reversing the connections of A and $-A$, an exclusive-nor (XNOR) gate is constructed.

By using four transmission gates, four inverters, and two XOR gates, an adder may be constructed according to Fig. 8.12.[3] $A \oplus B$ and the complement are formed using the TG XOR gate shown in Fig. 8.11. The *SUM* $(A \oplus B \oplus C)$ is formed by a multiplexer controlled by $A \oplus B$ (and complement). Examining the adder truth-table reveals that $CARRY = C$ when $A \oplus B$ is true. When $A \oplus B$ is false, $CARRY = A$ (or B). This adder has 24 transistors, the same as the combinational adder, but has the advantage of having equal *SUM* and *CARRY* delay times. In addition, the *SUM* and *CARRY* signals are non-inverted. The number of transistors may be reduced if speed is not the ulti-

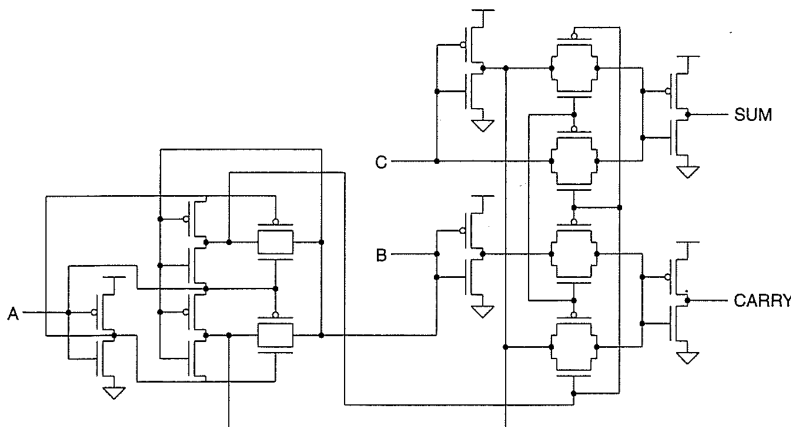


FIGURE 8.12 Transmission-gate adder

Micron Ex. 1037, p. 564
Micron v. YMTC
IPR2025-00119

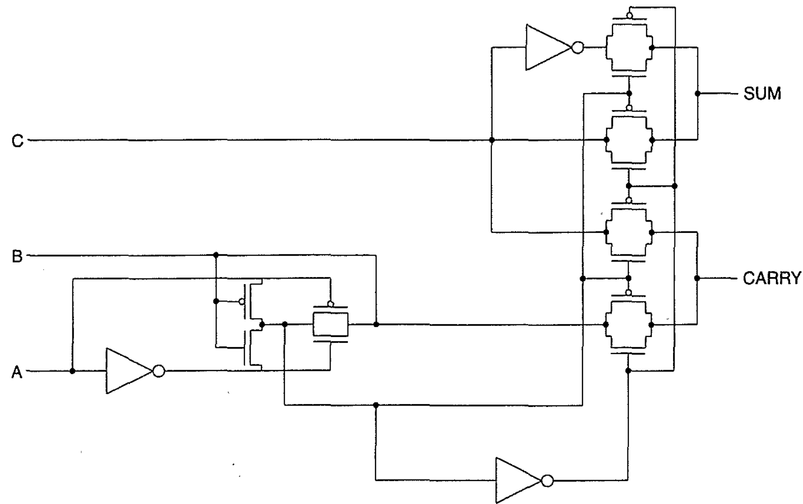


Figure 8.13 Optimized-area TG adder

mate goal. Two transistors may be eliminated by using an inverter on the output of the XOR gate. In addition with some optimization, the output buffers may be eliminated, as shown in Fig. 8.13.[4]

8.2.1.5 Carry-Lookahead Adders

The linear growth of adder carry-delay with the size of the input word for an n-bit adder may be improved by calculating the carries to each stage in parallel. The carry of the ith stage, C_i, may be expressed as

$$C_i = G_i + P_i \,.\, C_{i-1}, \tag{8.6}$$

where

$$G_i = A_i \,.\, B_i \qquad \textit{generate signal}$$
$$P_i = A_i + B_i \qquad \textit{propagate signal.}$$

Expanding this yields

$$C_i = G_i + P_iG_{i-1} + P_iP_{i-1}G_{i-2} + \ldots + P_i \ldots P_1C_0. \tag{8.7}$$

The sum S_i is generated by

$$S_i = C_{i-1} \oplus A_i \oplus B_i$$
$$\text{or } C_{i-1} \oplus P_i \text{ (if } P_i = A_i \oplus B_i). \tag{8.8}$$

The size and fan-in of the gates needed to implement this carry-lookahead scheme can clearly get out of hand. As a result, the number of stages of look-

Micron Ex. 1037, p. 565
Micron v. YMTC
IPR2025-00119

ahead is usually limited to about four. For four stages of lookahead, the appropriate terms are

$$C_0 = G_0 + P_0CI$$
$$C_1 = G_1 + P_1G_0 + P_1P_0CI$$
$$C_2 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0CI$$
$$C_3 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0CI. \quad \textbf{(8.9)}$$

Figure 8.14 shows a generic carry-lookahead adder. The *PG* generation and *SUM* generation circuits surround a carry-generate block. A possible implementation of the carry gate for this kind of carry-lookahead adder for 4 bits is shown in Fig. 8.15. Note that the gates have been partitioned to keep the number of inputs less than or equal to four. This is typical of the type of carry lookahead that would be used in a gate-array or standard-cell design. The circuit and layout are quite irregular. Taking the term of C_3, we note that it may be expressed as

$$C_3 = G_3 + P_3 \cdot (G_2 + P_2 \cdot (G_1 + P_1 \cdot (G_0 + P_0 \cdot CI))). \quad \textbf{(8.10)}$$

This function may be implemented as a domino CMOS (nMOS) gate, as shown in Fig. 8.16(d). Carry $C_0 - C_2$ are generated similarly. Note that the worst-case delay path in this circuit has six n-transistors in series. A high-speed static version of the carry-lookahead gate for C_3 is shown in Fig. 8.17.[5]

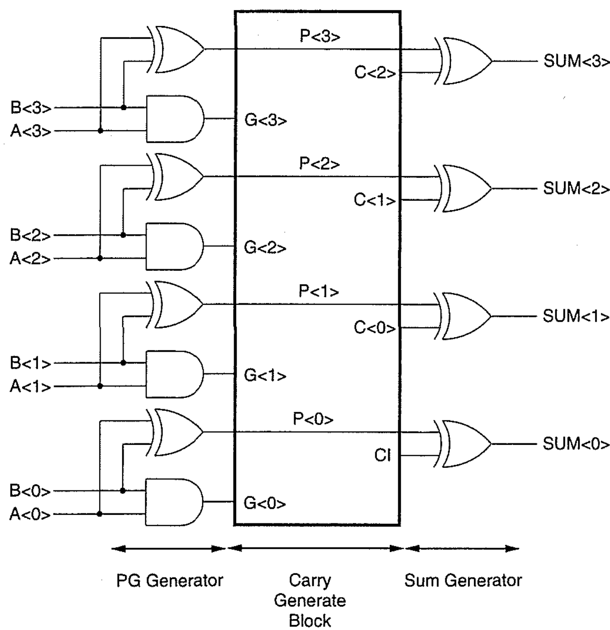


FIGURE 8.14 Generic carry-lookahead adder (CLA)

Micron Ex. 1037, p. 566
Micron v. YMTC
IPR2025-00119

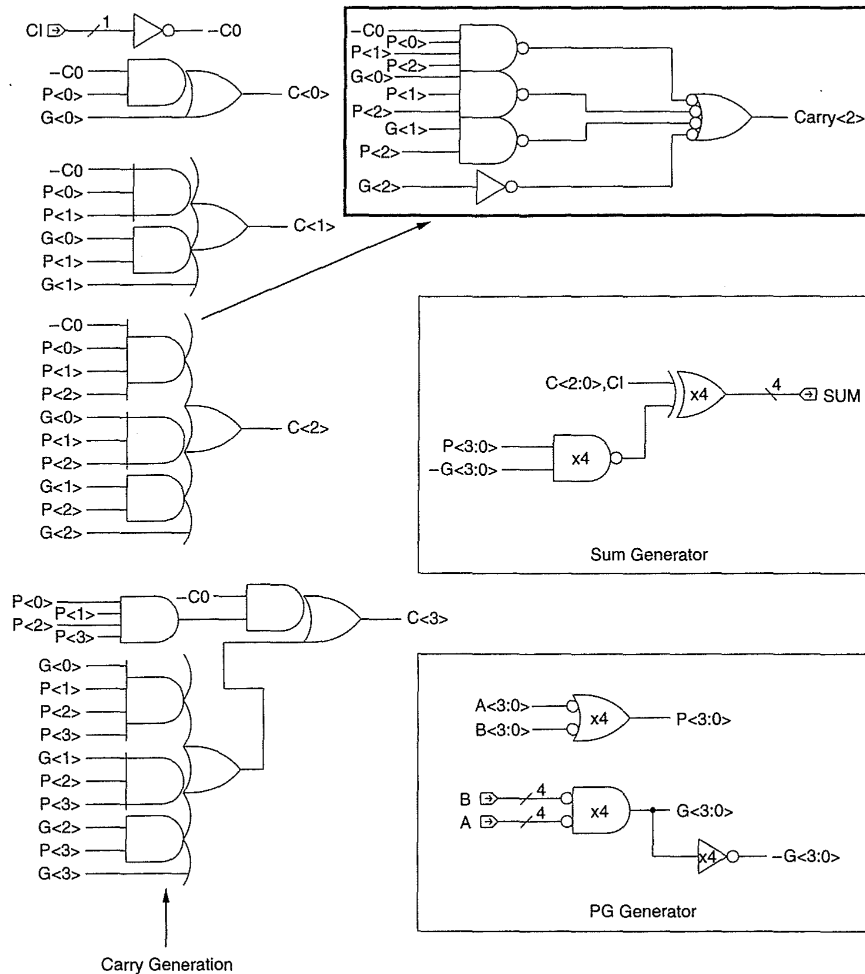


Figure 8.15 4-bit CLA

This uses pseudo-nMOS gates to achieve high-speed static operation. An adder using this stage may be constructed by using a 4-bit adder block with local ripple carry and this gate as the block carry generator.

The Manchester adder stage improves on the carry-lookahead implementation by using a single C_3 gate, as shown in Fig. 8.16(d). A selection of the elemental carry stages is shown in Fig. 8.18. The first, shown in Fig. 8.18(a), is a dynamic stage. Operation proceeds as follows. When *CLK* is low, the output node is precharged by the p pull-up transistor. When *CLK* goes high, the n pull-down transistor turns on. If carry generate $(A.B)$ is true, then the output node discharges. If carry propagate $(A + B)$ is true, then a previous carry may be coupled to the output node, conditionally discharging it. Note that in this circuit *CARRY* is actually propagated. A static stage is shown in Fig. 8.18(b). This requires P to be generated as $A \oplus B$. A multiplexer-based

Micron Ex. 1037, p. 567
Micron v. YMTC
IPR2025-00119

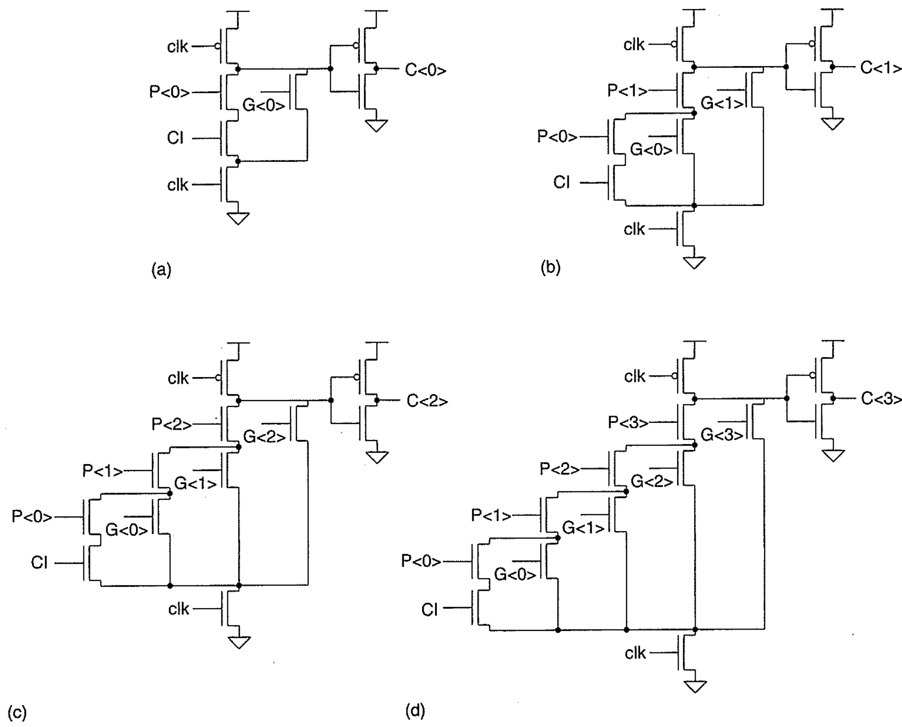


FIGURE 8.16 Dynamic carry gates

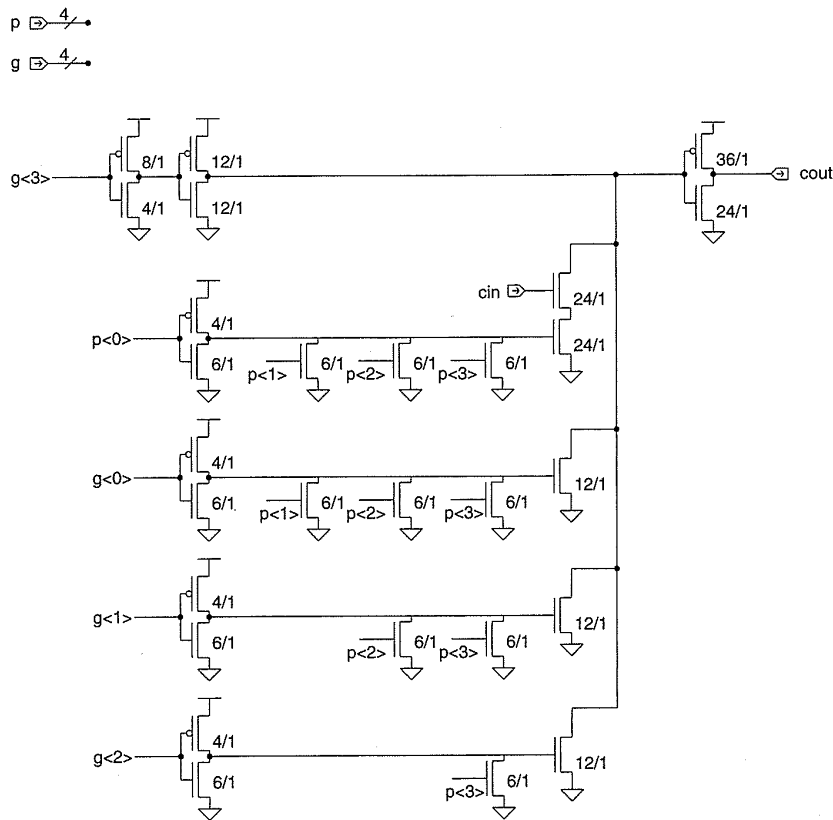


FIGURE 8.17 High-speed carry lookahead logic

Micron Ex. 1037, p. 568
Micron v. YMTC
IPR2025-00119

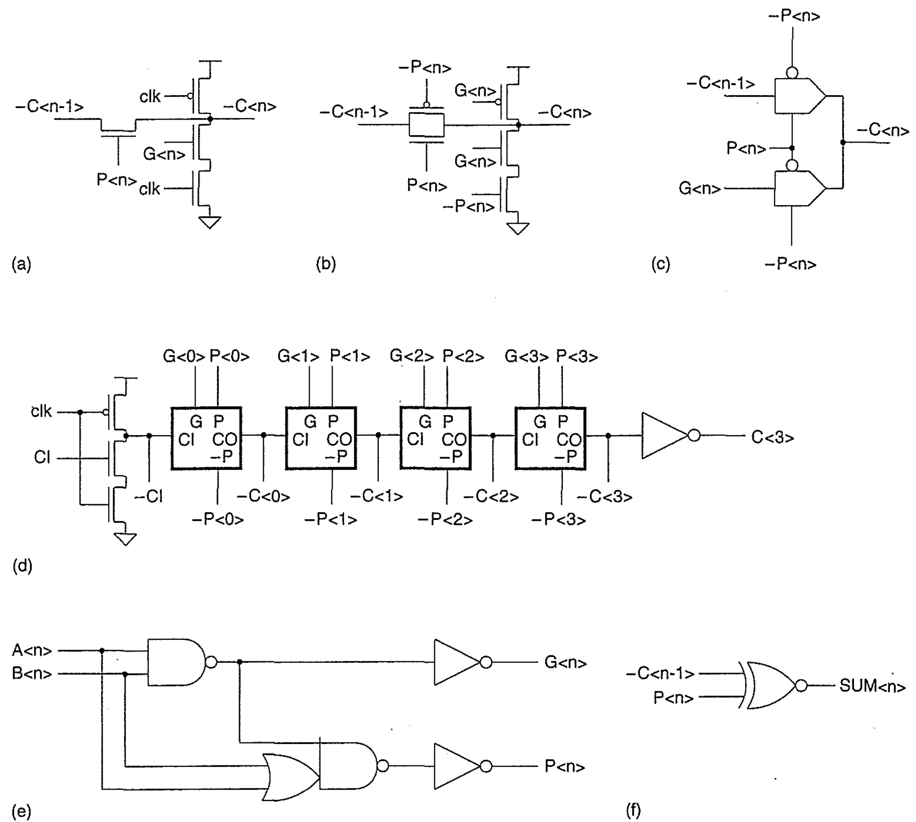
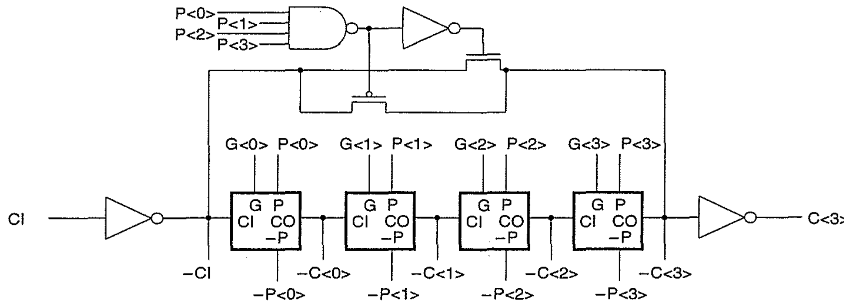


Figure 8.18 Manchester-adder circuits: (a) dynamic stage; (b) static stage; (c) MUX stage; (d) 4-bit section; (e) PG logic; (f) SUM logic

implementation is shown in Fig. 8.18(c). A 4-bit adder may be constructed by cascading four such stages and adding the circuitry to supply the required signals. This is commonly called a Manchester carry adder. Thus a 4-bit adder would be constructed as shown in Fig. 8.18(d). There is some similarity with the domino carry circuit. However, the intermediate carry gates are no longer needed, because the carry values are available in a distributed fashion. The 4-bit adder is chosen to reduce the number of series-propagate transistors, which improves the speed. Note that if all propagate signals are true, and *CI* is high, six series n-transistors pull the output node low in the case of the dynamic gate while five transistors are in series in the static gate. In addition to four Manchester stages, the adder requires four *PG* generator blocks, one representative implementation being shown in Fig. 8.18(e). Four *SUM* generate blocks (an XNOR gate), shown in Fig. 8.18(f), complete the adder. This worst-case propagation time can be improved by bypassing the four stages if all carry-propagate signals are true.[6] The additional circuitry needed to achieve this is shown in Fig. 8.19(a). It consists of an AND gate, which turns on a carry-bypass signal if all carry propagates are true. The optimum number of cascaded stages may be calculated for a given technology by simulation. A final implementation of a 4-bit Manchester adder is shown in Fig. 8.19(b).

Micron Ex. 1037, p. 569
Micron v. YMTC
IPR2025-00119



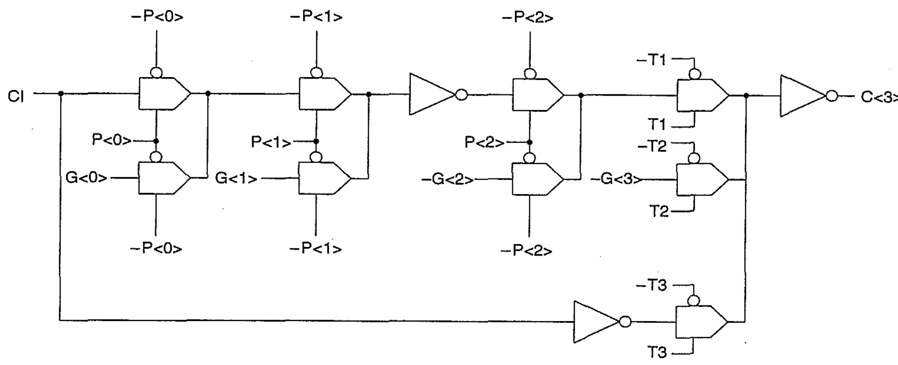


FIGURE 8.19 Manchester adder with carry bypass: (a) simple; (b) conflict free

This implementation[7] uses a "conflict-free" bypass circuit, which improves the speed by using a 3-input multiplexer that prevents conflicts at the wired OR node in the adder, shown in Fig. 8.19(a). The control signals T_1, T_2, and T_3 are respectively generated by

$$T_1 = -(P_0 P_1 P_2).P_3$$
$$T_2 = -P_3$$
$$T_3 = P_0 P_1 P_2 P_3.$$

Note that in this version the inverter present on the c_{in} signal has been moved to the center of the carry chain to improve speed (there are now a maximum of two transmission gates in series with an inverter). Very wide, fast adders may be constructed by extending the carry bypass shown in Fig. 8.19(b).[8]

Micron Ex. 1037, p. 570
Micron v. YMTC
IPR2025-00119

8.2.1.6 Carry-Select Adder

An additional approach to increase the speed of a parallel adder that expends area in favor of speed is to use a carry-select adder. The basic scheme is shown in Fig. 8.20(a).[9] Usually, two ripple-carry-adder structures are built (although any adder structure may be used), one with a zero carry-in and the other with a one carry-in. This is repeated for a certain sized adder, say, of 4-bits. The previous carry then selects the appropriate sum using a multiplexer or tristate adder gates. The stage carries and the previous carry are gated to form the carry for the succeeding stage. As a further optimization, each succeeding ripple adder may be extended by one stage to account for the delay in the carry-lookahead gate. Thus for a 32-bit adder, the stage numbers are 4-4-5-6-7-6, as shown in Fig. 8.20(b). This yields an adder with approximately (4 + 1 + 1 + 1 + 1 + 1), or 9, gate delays for a 32-bit addition.

8.2.1.7 Conditional-Sum Adder

A CMOS implementation of a conditional-sum adder[10] is shown in Fig. 8.21.[11] A conditional block generates C_0, C_1, S_0, and S_1 signals, as

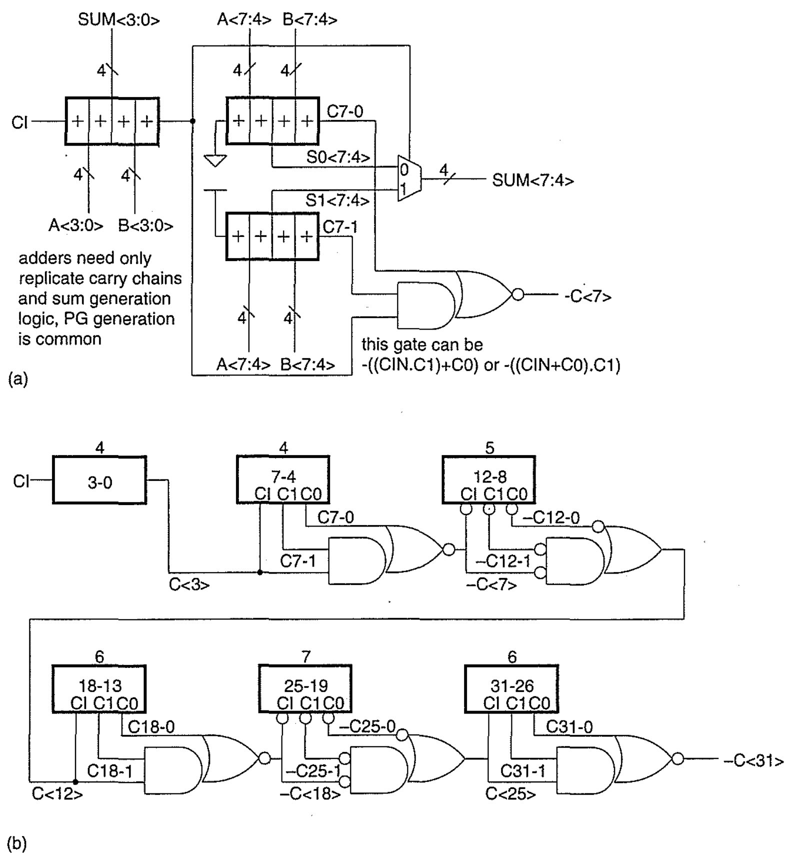
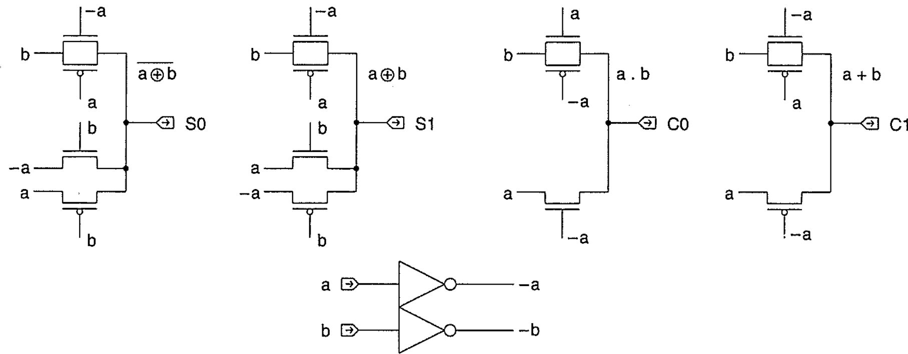
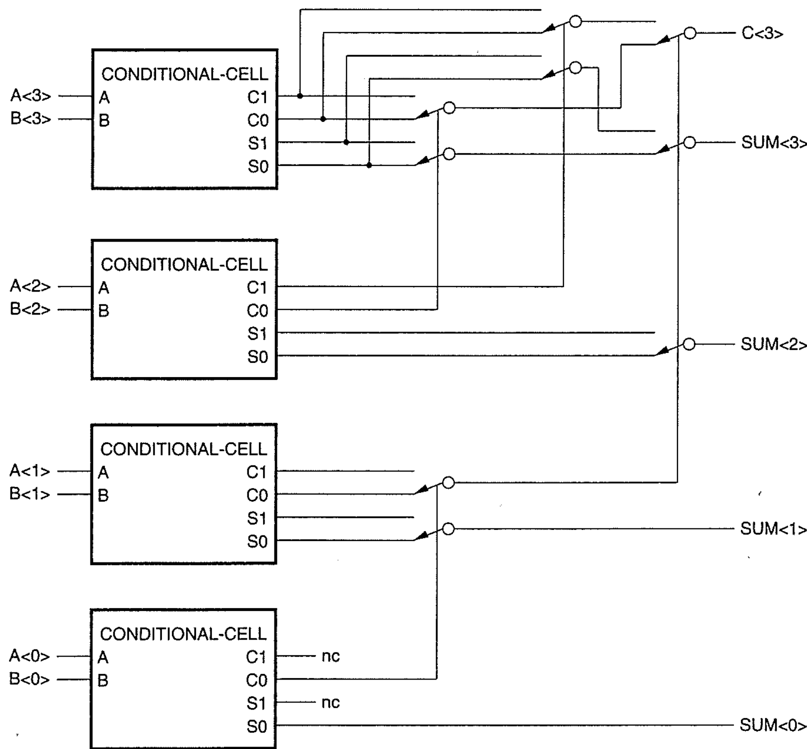


Figure 8.20 Carry-select adder: (a) basic architecture; (b) 32-bit carry-select adder example

Micron Ex. 1037, p. 571
Micron v. YMTC
IPR2025-00119



(a)



(b)

FIGURE 8.21 Conditional-sum adder: (a) basic circuits; (b) 4-bit adder example

shown in Fig. 8.21(a). Here pass logic has been used to generate

$$S_0 = -(A \oplus B)$$

$$S_1 = A \oplus B$$

$$C_0 = A \,.\, B$$

$$C_1 = A + B.$$

Micron Ex. 1037, p. 572
Micron v. YMTC
IPR2025-00119

The C_0 and C_1 signals are fed to successive stages, selecting between the S_0 and S_1 signals using transmission gate multiplexers. *SUM*s and *CARRY*s are generated in a tree-like fashion as shown in Fig. 8.21(b). For a 32-bit adder there are six transmission gates in series. If the stray capacitance in the series-transmission gates can be minimized, it is claimed that this adder can be quite fast.

8.2.1.8 Very Wide Adders

Adders with very large word sizes (>32 bits) can be constructed hierarchically by combining smaller "block" adders typically with a word width of 16 bits. Figure 8.22 shows a 64-bit adder composed of four 16-bit blocks. Each 16-bit block outputs a block *P* and block *G* signal that are fed to a block-carry generator. This module in turn feeds the carry-in to each 16-bit block.

A single bit of a typical block adder[12] is shown logically in Fig. 8.23(a), while a transmission-gate and inverter implementation is shown in Fig. 8.23(b). This is divided into three sections, which generate the local propagate signal (*P*), the block propagate (P_{out}), and the block generate (G_{out}) signals and the sum signal (*SUM*). These single bits are cascaded to form a 16-bit adder block. The block propagate and block-generate signals pass through series connections of transmission gates and inverters. These can be accelerated by using bypass techniques similar to that shown in Fig. 8.19(b). Figure 8.24(a) shows a representation of the block-generate chain for a 16-bit adder with bypassing. For example the final transmission gates are controlled by signals S_1, S_2, and S_3. These are generated as follows:

$S_1 = -T_2.P_{15}$ (passes $G<15>$ to output)

$S_2 = -T_1.T_2.P_{15}$ (passes $G<10>$ to output)

$S_3 = T_1.T_2.P_{15}$ (passes $G<5>$ to output),

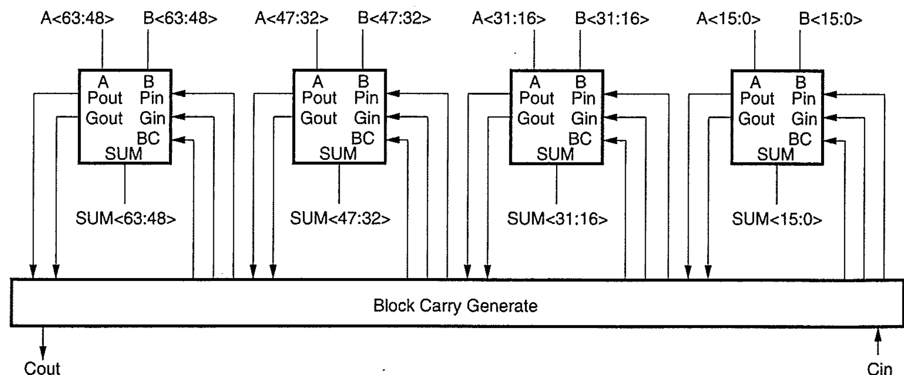


Figure 8.22 A 64-bit adder block diagram

Micron Ex. 1037, p. 573
Micron v. YMTC
IPR2025-00119

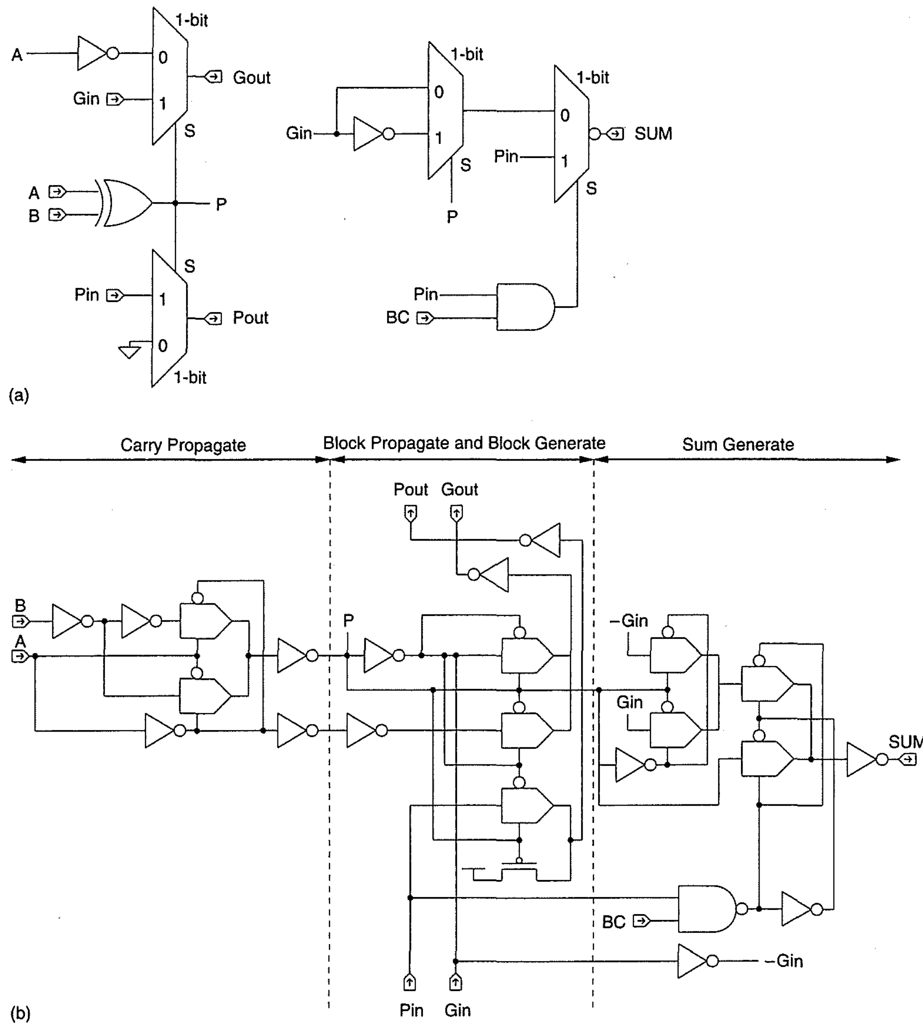


FIGURE 8.23 The cells in a 16-bit adder block used in the 64-bit adder: (a) gate diagram; (b) circuit diagram

where

$$T_1 = P_6.P_7.P_8.P_9.P_{10}$$
$$T_2 = P_{11}.P_{12}.P_{13}.P_{14}.$$

The block-carry generator for the 112-bit adder is shown in Fig. 8.24(b). The block generates and propagates from seven 16-bit adders are combined into seven carry-bypass multiplexers. These in turn are bypassed by the transmission gates, reducing the maximum number of series transmission gates from seven to four. The $-BC$ signals are the block carries that are fed to each 16-bit adder. The 112-bit adder that Figs. 8.23 and 8.24 are based on yielded a 8.5 *ns* 112-bit adder in a 0.8μ three-level-metal technology.

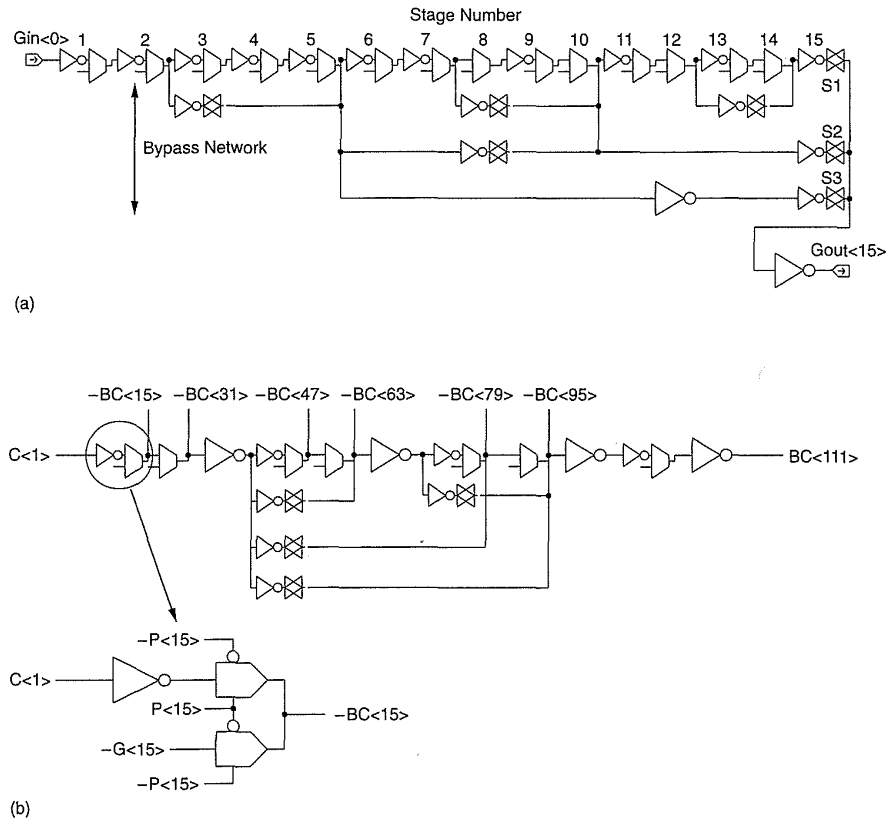


Figure 8.24 Bypass techniques used in a 112-bit adder: (a) 16-bit adder generate bypass logic; (b) 112-bit carry-bypass logic

8.2.1.9 Summary

With the number of adders presented, the natural question is "What adder should be used where?" In general, a ripple adder (Fig. 8.2, Fig. 8.3, Fig. 8.6, or Fig. 8.12) should be used as the first choice because they are small, simple, and relatively fast. The adder shown in Figs. 8.6 and 8.7 is one of the smallest that can be designed and is especially suited to pipelined adders. If a faster nonpipelined adder is required, standard-cell libraries frequently supply the adder shown in Fig. 8.15 or some other 4-bit adder with a lookahead generator. The carry-select adder is a good choice for a faster *n*-bit adder because it can easily be assembled from ripple adders and multiplexers and the speed can be improved by adding adders and multiplexers. This ease of construction comes at the expense of area. The Manchester adders are good choices for custom-designed datapaths with word widths from 16 to 32 bits because they are regular, small and fast. The transmission-gate adder (Fig. 8.12 or 8.13) is of use where the *SUM* and *CARRY* propagation times must be similar (e.g., in multiplier arrays). For adders used in floating-point ALUs, the adder outlined in Figs. 8.23 and 8.24 may be suitable.

Micron Ex. 1037, p. 575
Micron v. YMTC
IPR2025-00119

8.2.2 Parity Generators

A function related to binary addition is parity generation, that is, detecting whether the number of ones in an input word is odd or even. Frequently it is necessary to generate the parity of, say, a 16- or 32-bit word. The function is

$$PARITY = A_0 \oplus A_1 \oplus A_2 \oplus A_3 \ldots \oplus A_n. \quad \textbf{(8.11)}$$

Figure 8.25(a) shows a conventional implementation. A dynamic dual-rail logic version is shown in Fig. 8.25(b). A number of these may be cascaded to perform a 32-bit parity function.[13] A static 4-input XOR that could be used is shown in Fig. 8.25(c).[14] In a data path, Fig. 8.25(a) may be implemented as a linear column with a tree-routing channel connecting the XOR gates.

8.2.3 Comparators

A magnitude comparator is useful to compare the magnitude of two binary numbers. One can build a comparator from an adder and a complementer, as shown in Fig. 8.26. A zero detect (NOR gate) provides the $A = B$ signal while the final carry output provides the $B > A$ signal. Other signals—such as $A < B$ or $A \leq B$—may be generated by logical combinations of these signals. The generation of $B < A$ is shown in Fig. 8.26.

If equality comparison is required, then XNOR gates and an AND gate are all that is required, as shown in Fig. 8.27(a). Rather than a gate implementation, a pass-gate logic structure may be used, as shown in Fig. 8.27(b). Single-polarity transmission gates have been used here as might be appropriate in a low-power circuit, but of course complementary transmission gates may also be used. This structure does not draw any DC current but may be slow for long comparators. The final circuit shown in Fig. 8.27(c) is a merged XNOR/NOR gate using pseudo-nMOS. This gate draws DC current but is very small and very fast.

8.2.4 Zero/One Detectors

Detecting all ones or all zeros on wide words requires large fan-in AND or OR gates. One can build a tree of AND gates, as shown in Fig. 8.28(a). Here alternate NAND and NOR gates have been used. The delay to the output is proportional to log N, where N is the bit width of the word. If the word being checked has a natural skew in the outputs (such as at the output of a ripple adder), the designer might consider mimicking the adder delay in the zero or one detect as shown in Fig. 8.28(b). Here the delay from the last changing output to the zero/one detect is a constant one gate delay. Similar to the comparator example in the last section, a small and fast ONE/ZERO detection

Micron Ex. 1037, p. 576
Micron v. YMTC
IPR2025-00119

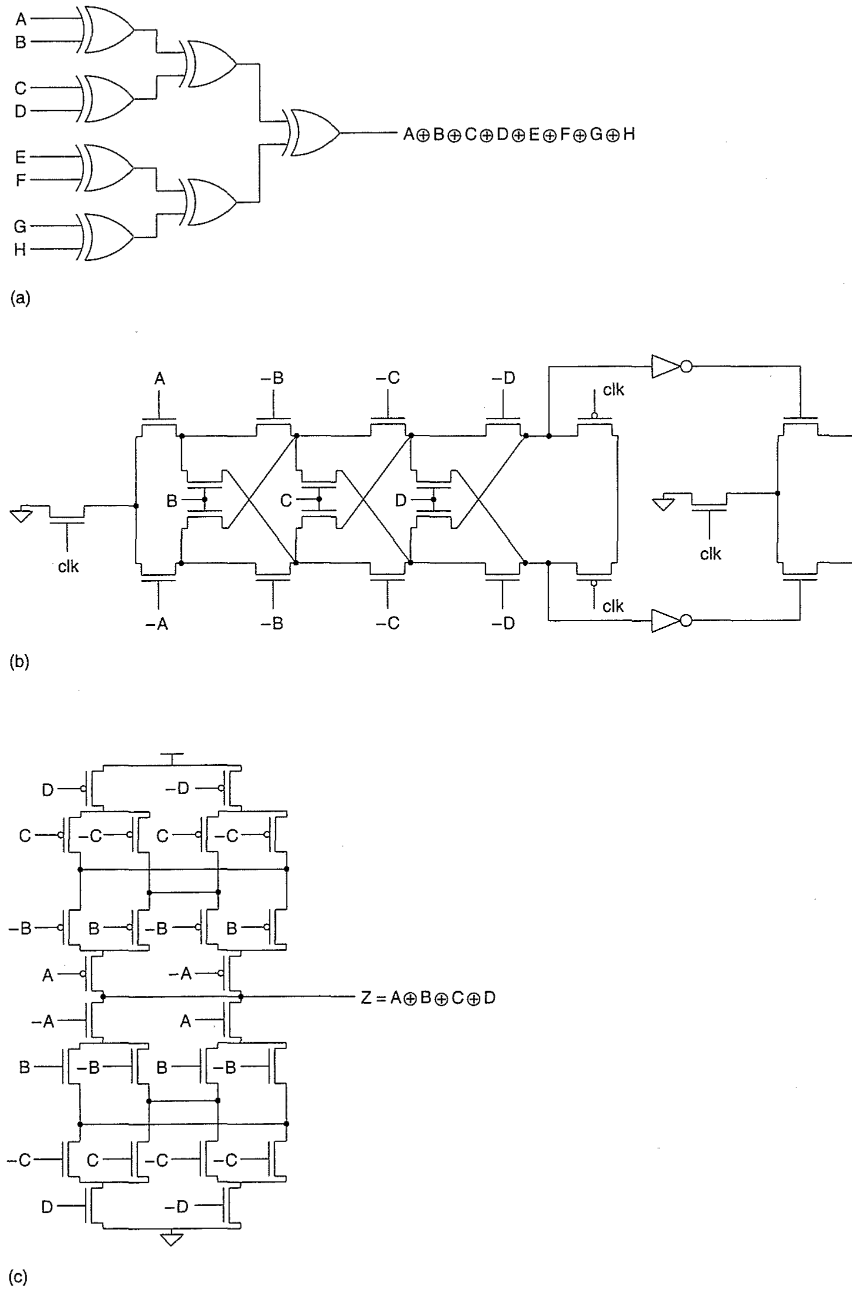


Figure 8.25 Parity generation: (a) static XOR tree; (b) dynamic version; (c) static 4-input XOR

circuit for word widths of less that 32 bits is the pseudo-nMOS NOR gate. At large word widths, self-loading may require the pseudo-nMOS gate to be split into 8- or 16-bit chunks.

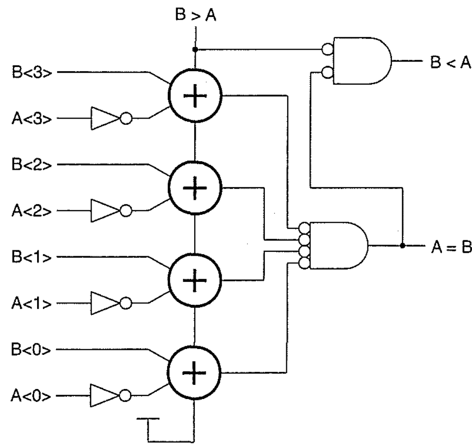


FIGURE 8.26 Comparator using an adder

8.2.5 Binary Counters

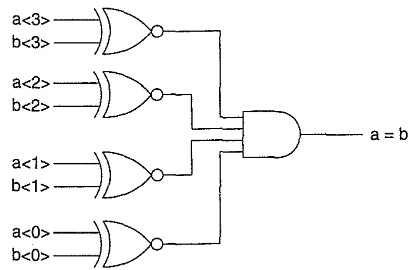
Binary counters are used to cycle through a sequence of binary numbers. An asynchronous counter has outputs that change at varying times with respect to the clock edge, whereas a synchronous counter has outputs that change at substantially the same time.

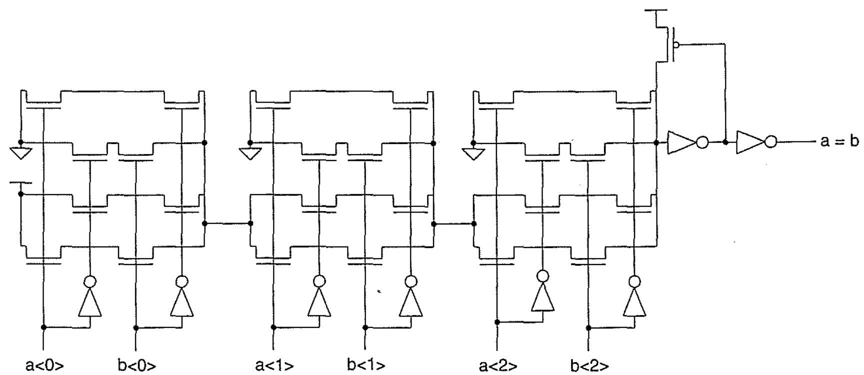
8.2.5.1 Asynchronous Counters

A "ripple-carry" binary counter is shown in Fig. 8.29. This is based on the toggle register introduced in Chapter 5. The *T* register (which is a single-counter stage) is reproduced in Fig. 8.29(a). This counter stage may be cascaded, as shown in Fig. 8.29(b). Note that the clocking of each stage is carried out by the previous counter stage, and thus the time it takes the last counter stage to settle can be quite large for a long counter chain. This counter is shown mainly for historical and reference purposes and should not be used as shown. Note that it has no reset signal, thus making it extremely difficult to test.

8.2.5.2 Synchronous Counters

A general synchronous up/down counter is shown in Fig. 8.30. It uses an adder and a D register per bit position. The speed that this counter can operate is determined by the ripple-carry time from the LSB to the MSB. This can be improved using any of the carry-lookahead techniques discussed in Section 8.2. If only an incrementer is required, the adder circuit degenerates into a synchronous counter stage, comprising an XOR gate, an AND gate,





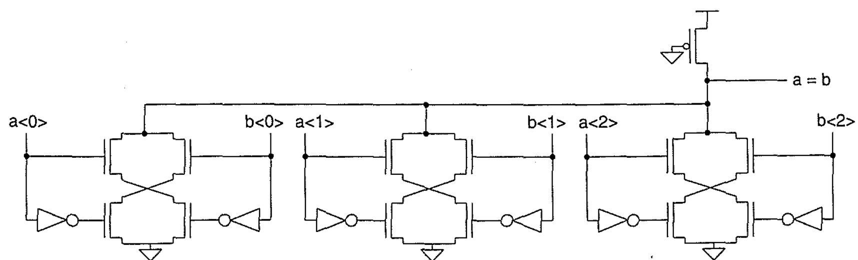


Figure 8.27 Comparator circuits: (a) XNOR based; (b) pass gate based; (c) pseudo-nMOS based

and a D register, as shown in Fig. 8.31. A multiplexer on the D input of the register allows a value to be loaded into the register for initialization. Remembering that an XOR can be implemented with a multiplexer yields the counter structure shown in Fig. 8.32(a). A reset register allows initialization, while the XOR function is provided by the multiplexer on the register D input. The multiplexer selects between the true and complement values of the register, based on the carry-input value. A more detailed version of the counter cell is shown in Fig. 8.32(b).

Micron Ex. 1037, p. 579
Micron v. YMTC
IPR2025-00119

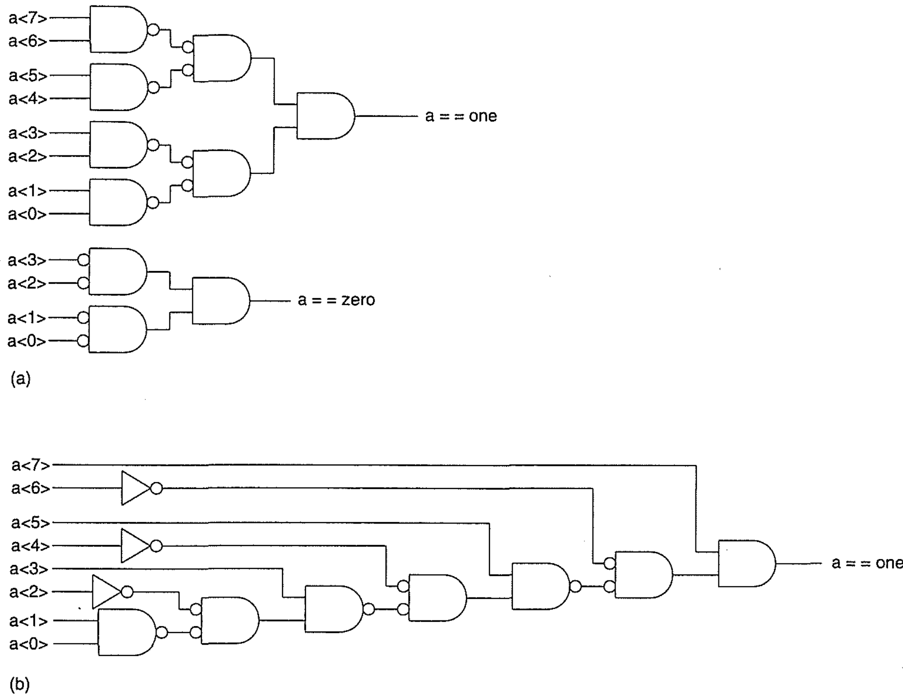


FIGURE 8.28 One- and zero-detect circuits: (a) tree; (b) "ripple"

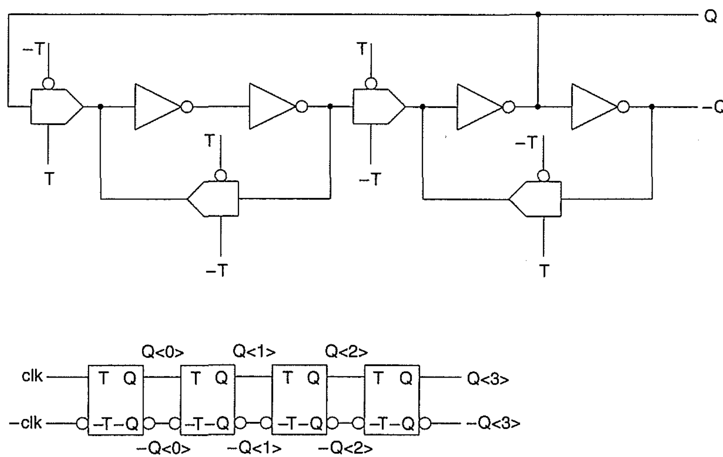


FIGURE 8.29 Asynchronous counter

8.2.6 Boolean Operations—ALUs

Boolean operations are most easily accomplished by using the multiplexer-based circuit shown in Fig. 5.35. This is shown in multiplexer format in Fig. 8.33. An Arithmetic Logic Unit (ALU) requires both arithmetic (add,

Micron Ex. 1037, p. 580
Micron v. YMTC
IPR2025-00119

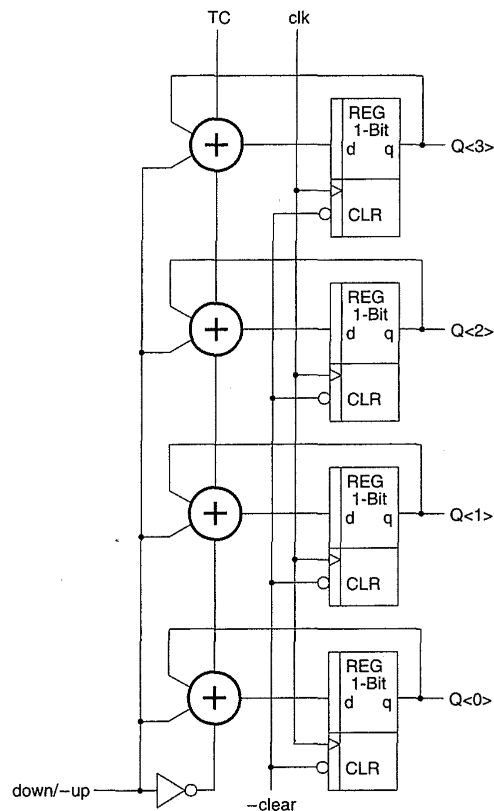


Figure 8.30 Synchronous up/down counter using adders and registers

subtract) and Boolean operations. One may either multiplex between an adder and a Boolean unit or merge the Boolean unit into the adder as in the classic TTL 181 ALU.[15] A 1-bit CMOS implementation of the latter circuit that uses a Manchester carry stage is shown in Fig. 8.34. Signal *mode* is false for arithmetic operations and true for Boolean operations. Signals *S*<3:0> control the operation type. For instance, in Boolean mode for *S*<3>=0, *S*<2>=0, *S*<1>=0, *S*<1>=1, *b* is passed to the output.

8.2.7 Multiplication

In many digital signal processing operations—such as correlations, convolution, filtering, and frequency analysis—one needs to perform multiplication. Multiplication algorithms will be used to illustrate methods of designing different cells so that they fit into a larger structure. In order to introduce these designs, simple serial and parallel multipliers will be introduced. The appropriate texts should be consulted for more definitive system architectures. The most basic form of multiplication consists of forming the product of two

Micron Ex. 1037, p. 581
Micron v. YMTC
IPR2025-00119

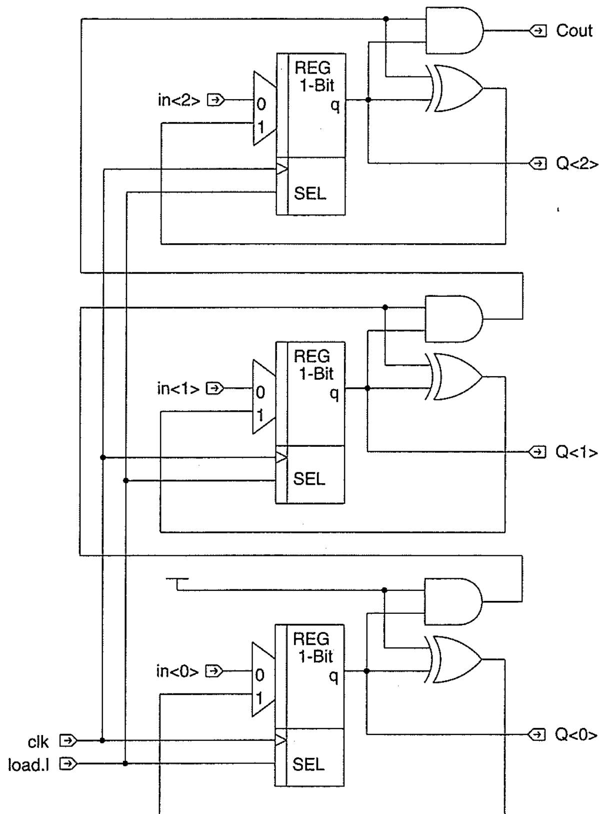


FIGURE 8.31 Incrementer

positive binary numbers. This may be accomplished through the traditional technique of successive additions and shifts in which each addition is conditional on one of the multiplier bits. For example, the multiplication of two positive binary integers, 12_{10} and 5_{10}, may proceed using the shift-and-add method in the following manner:

```
multiplicand:          1100 : 12₁₀
multiplier             0101 : 5₁₀
                       1100
                      0000
                     1100
                    0000
                    0111100 : 60₁₀
```

Therefore, the multiplication process may be viewed to consist of the following two steps:

1. Evaluation of partial products.
2. Accumulation of the shifted partial products.

Micron Ex. 1037, p. 582
Micron v. YMTC
IPR2025-00119

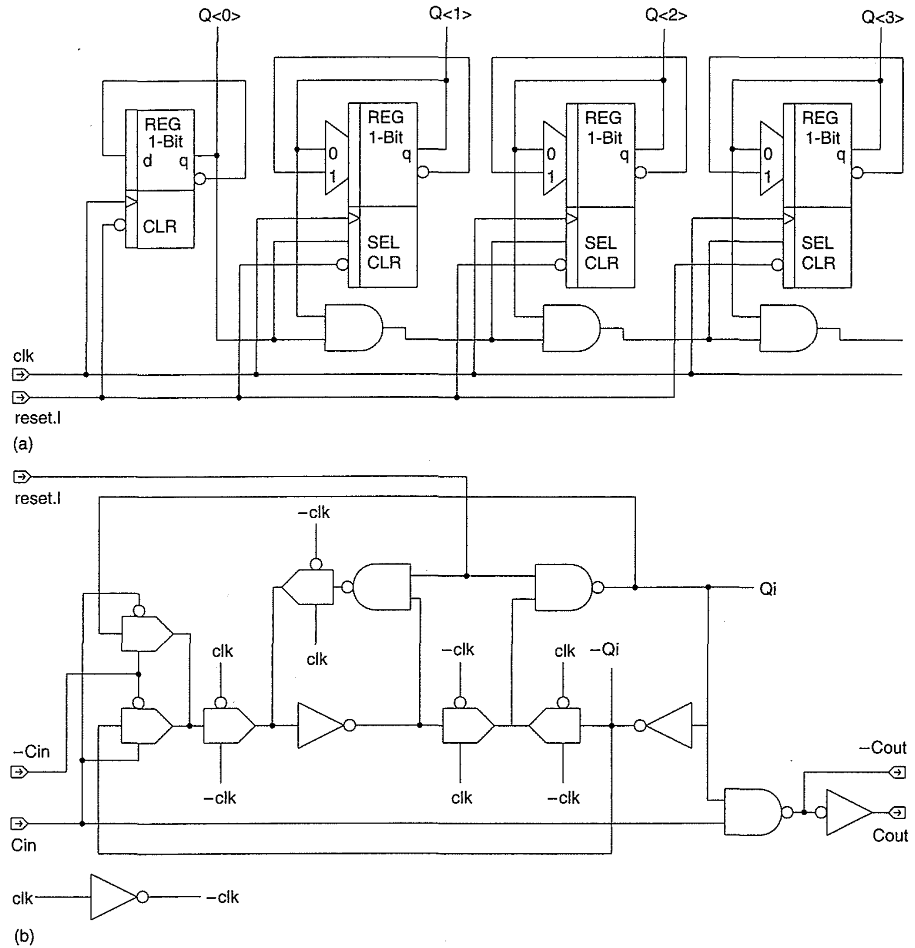


Figure 8.32 Compact synchronous counter

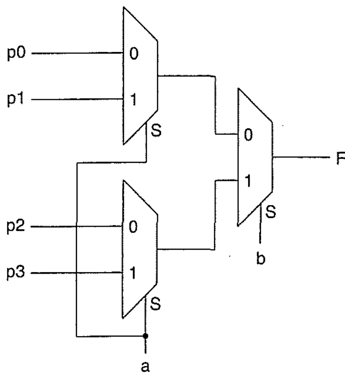


FIGURE 8.33 Boolean logic unit as MUXes

It should be noted that binary multiplication is equivalent to a logical AND operation. Thus evaluation of partial products consists of the logical ANDing of the multiplicand and the relevant multiplier bit. Each column of partial products must then be added and, if necessary, any carry values passed to the next column. There are a number of techniques that may be used to perform multiplication. In general, the choice is based on factors such as speed, throughput, numerical accuracy, and area. As a rule, multipliers may be classified by the format in which data words are accessed, namely:

- serial form.
- serial/parallel form.
- parallel form.

Micron Ex. 1037, p. 583
Micron v. YMTC
IPR2025-00119

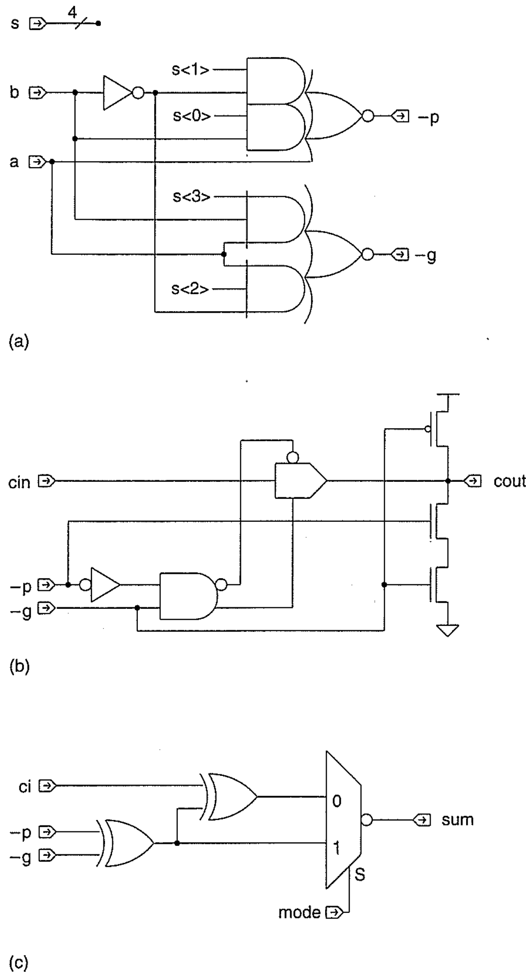


FIGURE 8.34 181 ALU

8.2.7.1 Array Multiplication

A parallel multiplier is based on the observation that partial products in the multiplication process may be independently computed in parallel. For example, consider the unsigned binary integers X and Y.

$$X = \sum_{i=0}^{m-1} X_i 2^i$$

$$Y = \sum_{j=0}^{n-1} Y_j 2^j$$

The product is found by

$$P = X \times Y = \sum_{i=0}^{m-1} X_i 2^i \cdot \sum_{j=0}^{n-1} Y_j 2^j$$

$$= \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} (X_i Y_j)\, 2^{i+j}$$

$$= \sum_{k=0}^{m+n-1} P_k 2^k.$$

Thus P_k are the partial product terms called summands. There are mn summands, which are produced in parallel by a set of mn AND gates. For 4-bit numbers, the expression above may be expanded as in Table 8.2.

An $n \times n$ multiplier requires $n(n-2)$ full adders, n half adders, and n^2 AND gates. The worst-case delay associated with such a multiplier is $(2n+1)\tau_g$, where τ_g is the worst-case adder delay. Figure 8.35 shows a cell that may be used to construct a parallel multiplier. The X_i term is propagated diagonally from top right to bottom left, while the Y_j term is propagated horizontally. Incoming partial products enter at the top. Incoming *CARRY IN* values enter at the top right of the cell. The bit-wise AND is performed in the cell, and the *SUM* is passed to the next cell below. The *CARRY OUT* is passed to the bottom left of the cell. Figure 8.36 shows the multiplier array with the partial products enumerated. This arrangement may be drawn as a square array, as shown in Fig. 8.37, which is the most convenient for implementation. In this version the degeneracy of the first two rows of the multiplier are shown. The first row of the multiplier adders has been replaced with AND gates while the second row employs half-adders rather than full adders. This optimization might not be done if a completely regular multiplier were required (i.e., one

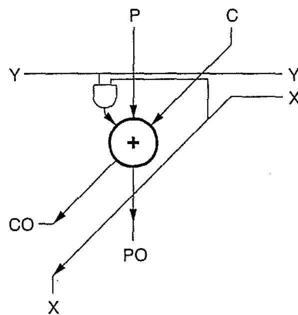


FIGURE 8.35 Array multiplier cell

TABLE 8.2 4-bit Multiplier Partial Products

				X3	X2	X1	X0	Multiplicand
				Y3	Y2	Y1	Y0	Multiplier
				X3Y0	X2Y0	X1Y0	X0Y0	
			X3Y1	X2Y1	X1Y1	X0Y1		
		X3Y2	X2Y2	X1Y2	X0Y2			
	X3Y3	X2Y3	X1Y3	X0Y3				
P7	P6	P5	P4	P3	P2	P1	P0	Product

Micron Ex. 1037, p. 585
Micron v. YMTC
IPR2025-00119

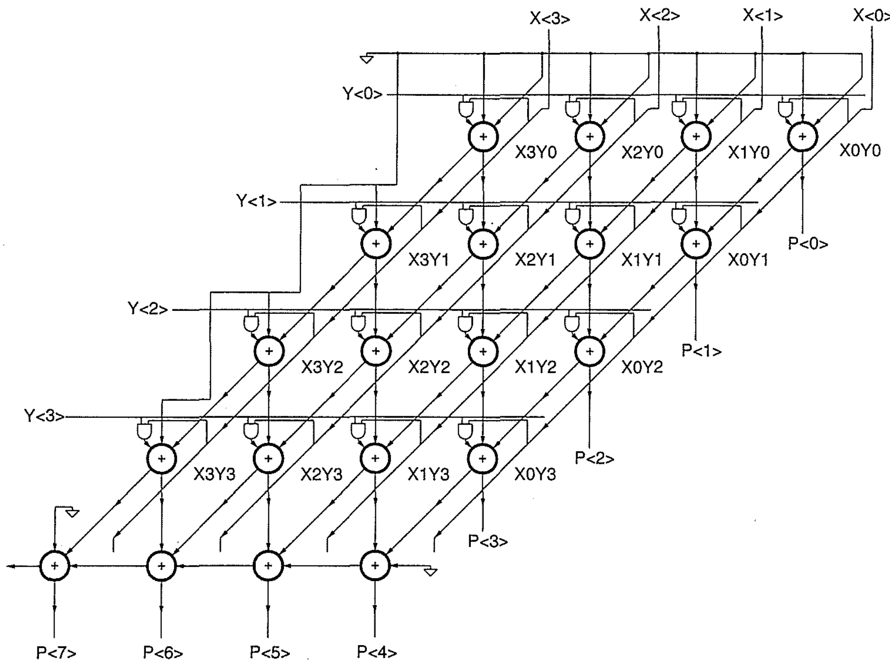


FIGURE 8.36 A 4 × 4 array multiplier

array cell). In this case the appropriate inputs to the first and second row would be connected to ground, as shown in Fig. 8.36.

The cell design for this multiplier is relatively straightforward, with the main attention paid to the adder. An adder with equal carry and sum propagation times is advantageous, because the worst-case multiply time depends on both paths.

*8.2.7.2 Radix-*n *Multiplication*

The structure shown in Figs. 8.36 and 8.37 computes the partial-products in a Radix-2 manner, that is by observing one bit of the multiplicand at a time. Higher radix multipliers may be designed to reduce the number of adders and hence the delay required to compute the partial sums. The best known method is called Booth recoding, which is a Radix-4 multiplication scheme.

A Booth-recoded multiplier examines three bits of the multiplicand at a time to determine whether to add zero, 1*, –1*, 2*, or –2* of that rank of the multiplicand. Table 8.3 shows the operation to be performed based on the current two bits of the multiplicand and the previous bit. In addition three control values are shown: *ZERO* zeroes the operand, *NEG* inverts the operand, and *TWO* multiplies the value by 2 (left shift).

Figure 8.38 shows a 16 × 16 Booth-recoded multiplier. Figure 8.38(a) shows the top level schematic and a possible floorplan. The schematic shows the multiplier divided into two parts—one the Booth array and the other a

Micron Ex. 1037, p. 586
Micron v. YMTC
IPR2025-00119

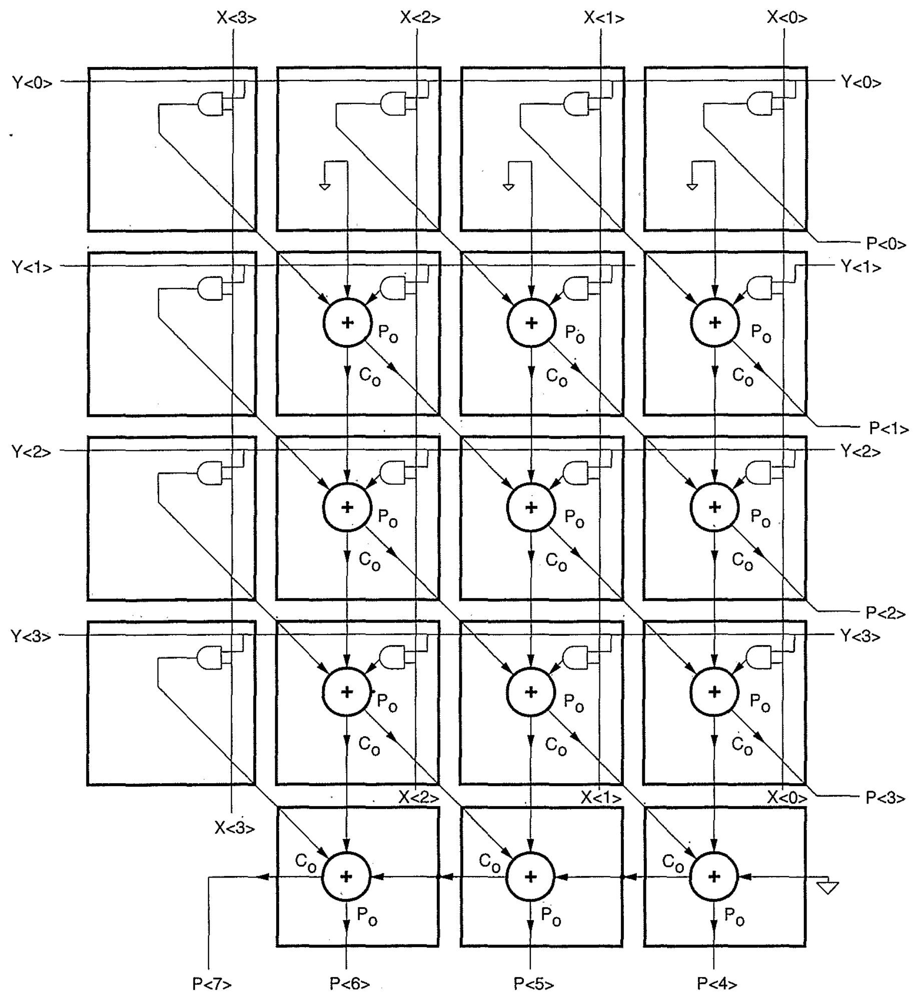


Figure 8.37 A square version of the 4 × 4 array multiplier

TABLE 8.3 Booth-recoding Values

X_{i-1}	X_i	X_{i+1}	OPERATION	NEG	ZERO	TWO
0	0	0	add 0	1	1	0
0	0	1	add 2	0	0	1
0	1	0	sub 1	1	0	0
0	1	1	add 1	0	0	0
1	0	0	sub 1	1	0	0
1	0	1	add 1	0	0	0
1	1	0	sub 2	1	0	1
1	1	1	add 0	0	1	0

Micron Ex. 1037, p. 587
Micron v. YMTC
IPR2025-00119

carry propagate adder (CPA). The Booth array accepts two 16-bit inputs, *MIER*<15:0> (the multiplier) and *MCAND*<15:0> (the multiplicand) and feeds the CPA. The CPA also accepts a 32-bit input (*SUM-IN*<31:0>), which is used to perform multiple-accumulates. The floorplan divides the layout according to the schematic hierarchy using an array block and a CPA block. As the only 32-bit datapath is the final CPA, this structure is folded to obtain a set of datapaths that are roughly 16 bits high. Figure 8.38(b) shows the schematic and floorplan for the array section of the multiplier. It consists of 8 ranks of adders each 17 bits wide (or tall for the floorplan shown). The first rank (Booth-First-16) degenerates to the schematic shown in Fig. 8.38(c), while the remaining ranks are represented by the schematic in Fig. 8.38(e). Both ranks use a Booth decode cell which is shown in Fig. 8.38(d). This cell observes 3 bits of the multiplier (*MIER*) and produces the control signals *X*1, *X*2 and *N*<1:0> which are used in the array adders (Figs. 8.38f) and

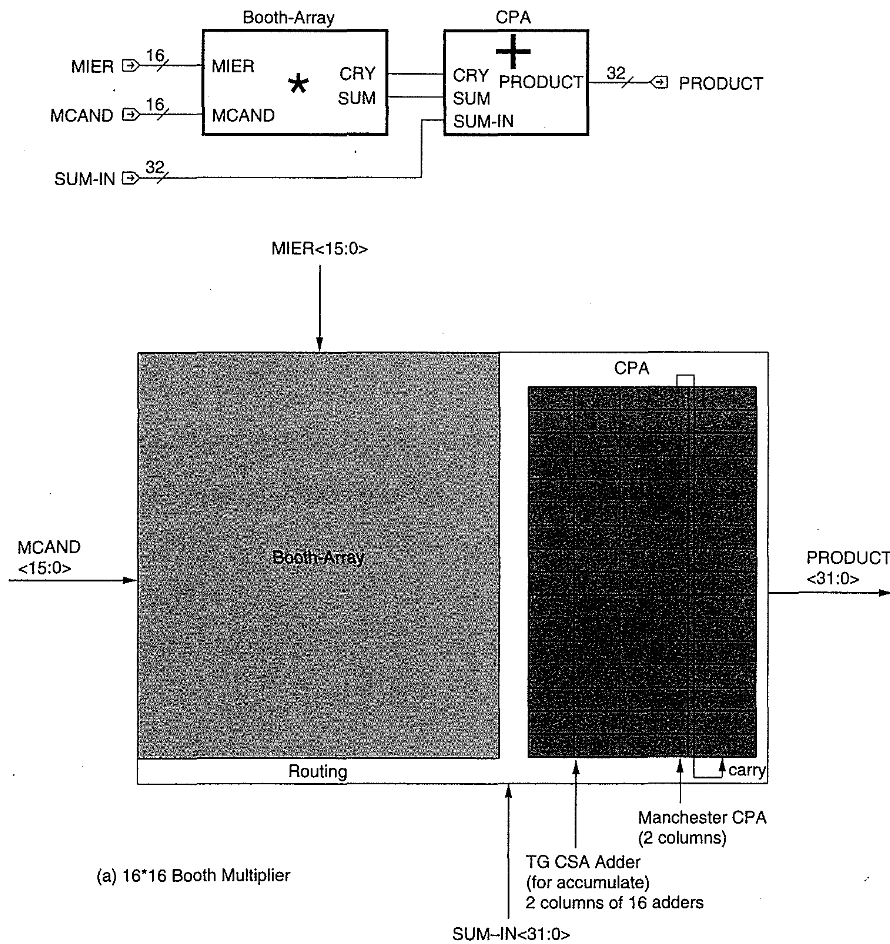
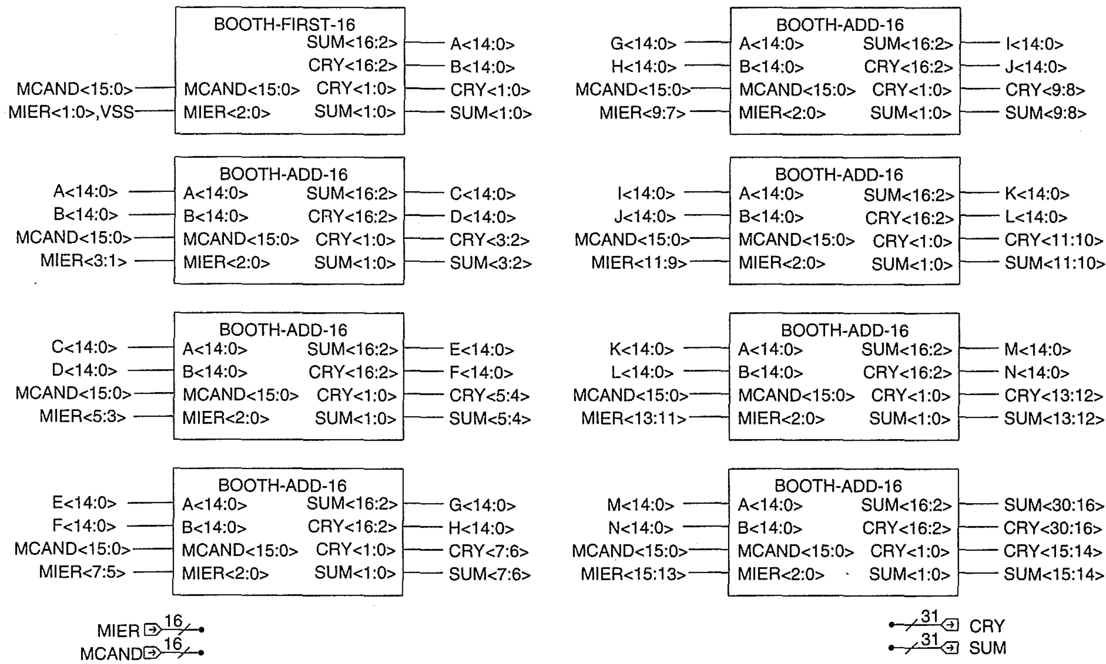
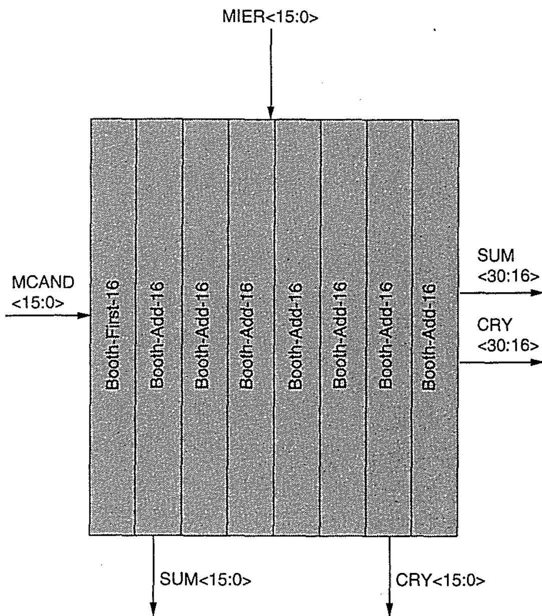


FIGURE 8.38 Radix-2 multiplier (Booth-recoded): (a) 16 × 16 multiplier top level schematic and floorplan; (b) array schematic and floorplan; (c) first rank schematic; (d) Booth decoder; (e) adder rank schematic, rank floorplan, and bit floorplan; (f) Booth gate; (g) array adder schematic and mask layout; (h) final adder

Micron Ex. 1037, p. 588
Micron v. YMTC
IPR2025-00119

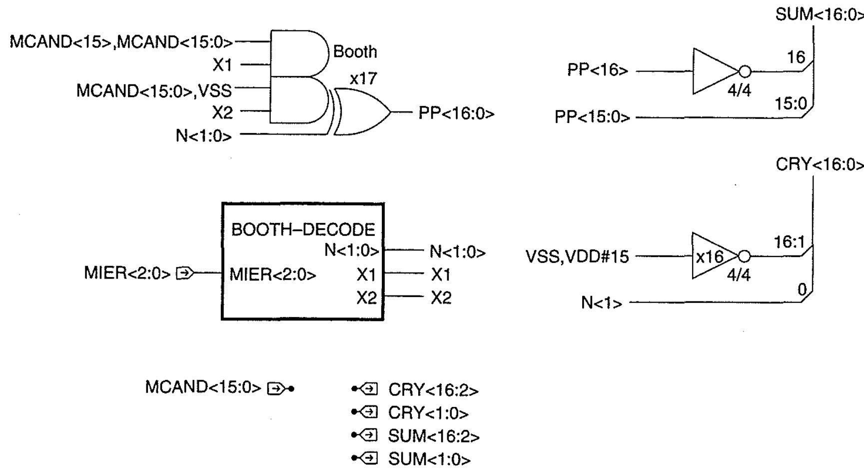




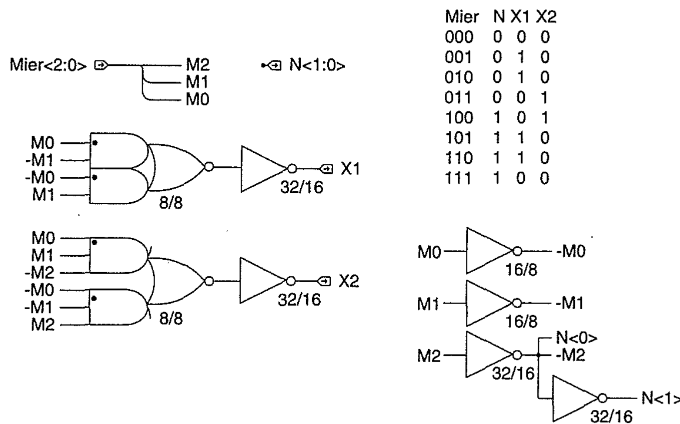
(b) Booth Array

Figure 8.38 *(continued)*

Micron Ex. 1037, p. 589
Micron v. YMTC
IPR2025-00119



(c) Booth-First-16



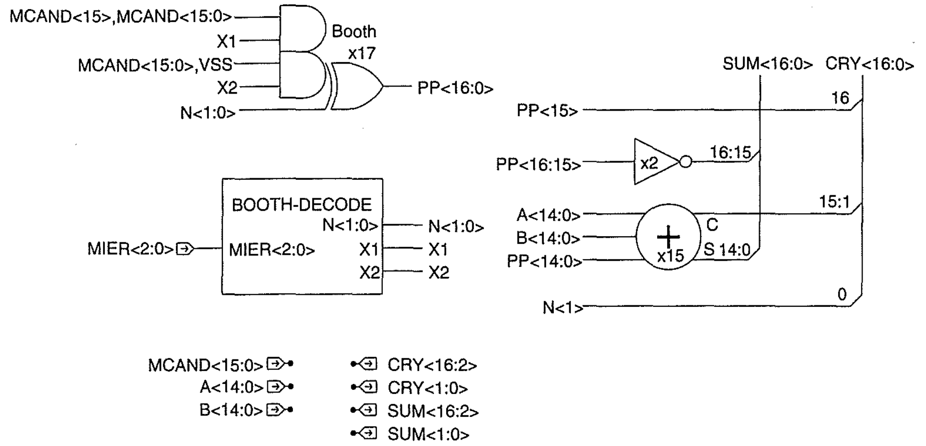
Mier	N	X1	X2
000	0	0	0
001	0	1	0
010	0	1	0
011	0	0	1
100	1	0	1
101	1	1	0
110	1	1	0
111	1	0	0

(d) Booth-Decode

FIGURE 8.38 *(continued)*

8.38g). From Fig. 8.38(b) it may be seen that *MIER*<1:0> and V_{SS} are fed to the first rank (Booth-First-16), *MIER*<3:1> to the second rank and so on. Each rank "retires" two bits of the partial product sum (*SUM*) and carry (*CRY*) so by the last adder rank (lower right of schematic in Fig. 8.38b) 31 *SUM, CRY* pairs have been produced. These are used by the CPA to produce a 32-bit result. A possible floorplan of the Booth array is shown in Fig. 8.38(b). It consists of the 8 ranks of adder abutted horizontally. The circuit diagram for an adder rank appears in Fig. 8.38(e). It consists of a Booth decode (Fig. 8.38d), 17 Booth gates (Fig. 8.38f), and a 17-bit carry-save adder. The latter circuit consists of a 15-bit CSA for the LSBs and two inverters for the top 2 bits. The floorplan of the adder rank and adder bit is shown in Fig. 8.38(e). The adder rank consists of 15 Booth-Adder modules,

Micron Ex. 1037, p. 590
Micron v. YMTC
IPR2025-00119



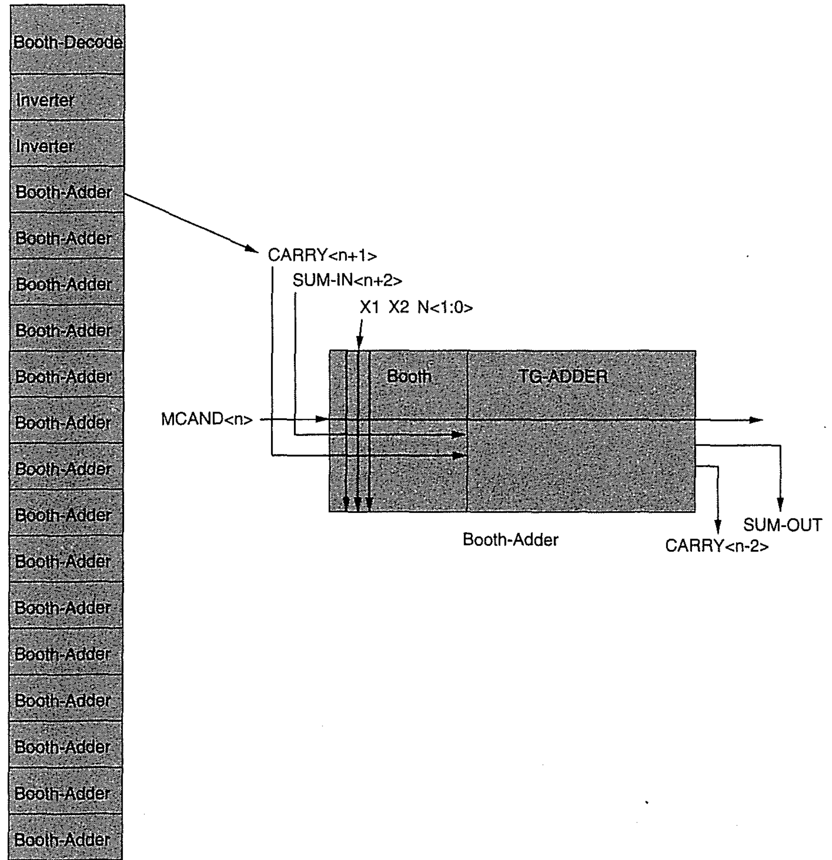
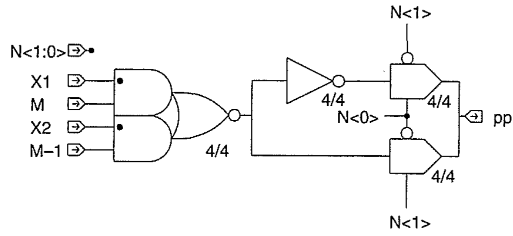
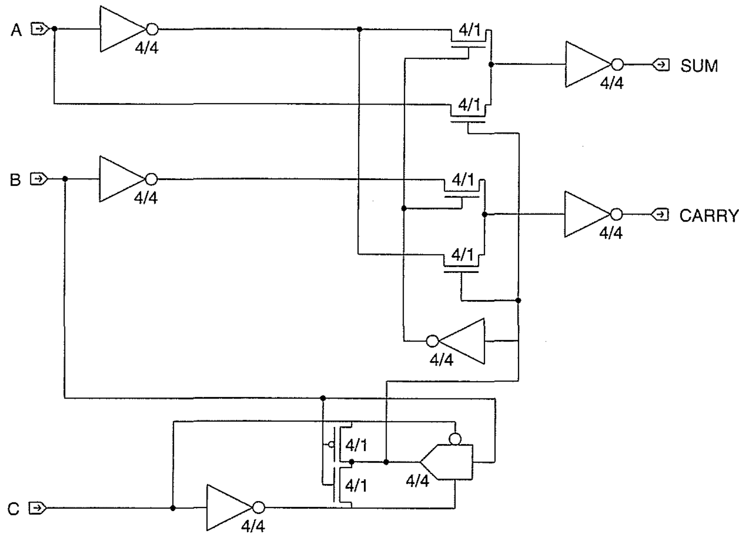


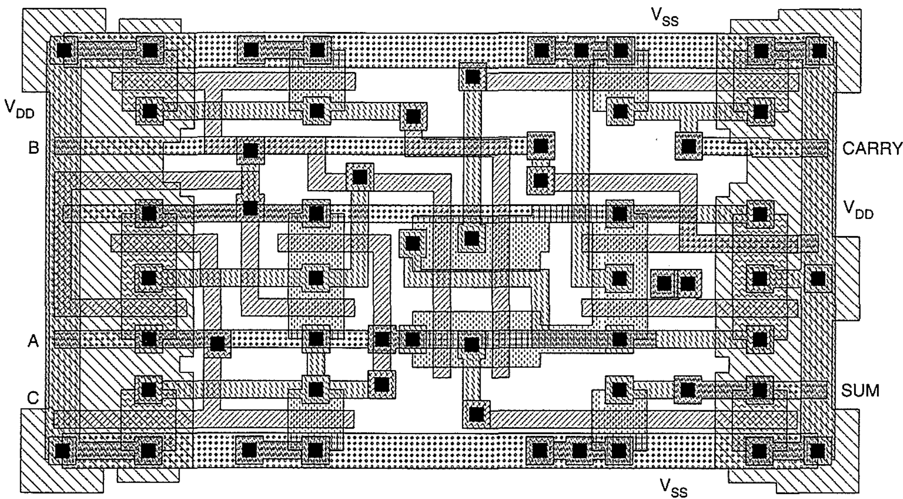
Figure 8.38 *(continued)* (e) Booth-Add-16

Micron Ex. 1037, p. 591
Micron v. YMTC
IPR2025-00119



(f) Booth





(g)

FIGURE 8.38 *(continued)*

Micron Ex. 1037, p. 592
Micron v. YMTC
IPR2025-00119

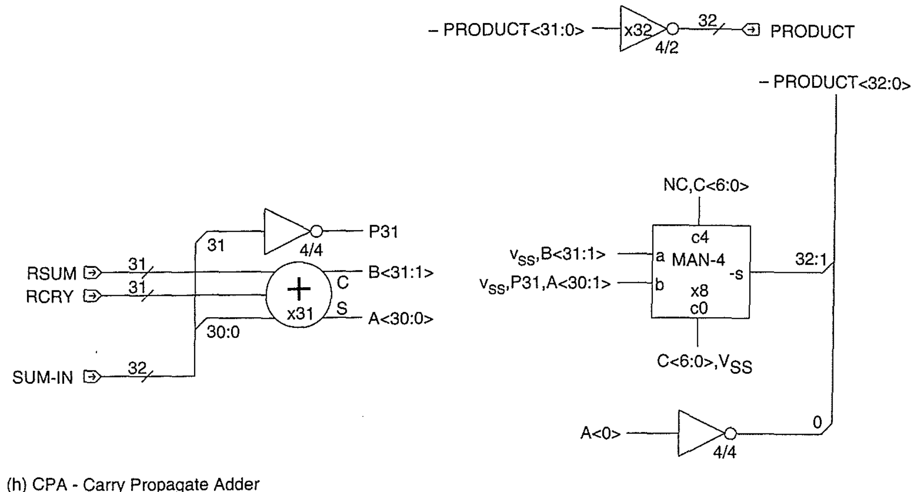


Figure 8.38 *(continued)* (h) CPA - Carry Propagate Adder

2 inverters and the Booth Decoder vertically abutted. The Booth-Adder consists of a Booth gate and adder stage horizontally abutted. The *X*1, *X*2 and *N*<1:0> lines feed vertically from the Booth-Decode logic to the Booth gate in each adder (or inverter). At each rank the *SUM* shifts right by two bit positions while the *CRY* shifts right by one bit position. The adder shown in Fig. 8.38(g) shows one possible adder implementation that has been optimized for size by using the transmission-gate adder with n-pass transistors. A possible mask layout is also shown. Figure 8.38(h) shows the final adder (CPA). It consists of a CSA to add in the *SUM-IN* signal and 32 stages of Manchester adder (8 MAN-4) to produce the final output. Any fast 32-bit CPA could be used here. Of course this multiplier may be made faster by including appropriate pipeline registers.

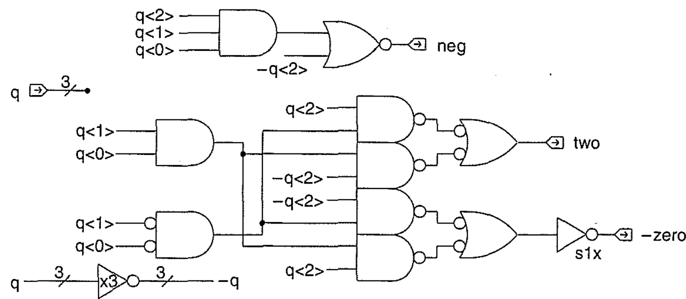
Figure 8.39 shows some alternative Booth related circuits. Figure 8.39(a) shows an alternative Booth-decoder stage along with a generic multiplier cell (Fig. 8.39b). It is implemented with a multiplexer, an XOR gate, an AND gate, and an adder. This circuit may be highly optimized at the circuit level. Figure 8.39(c) shows one particular implementation for the pre-adder gating that uses n-channel pass transistors.

Radix-8 multiplication carries Radix-4 multiplication one step further by requiring that +1,–1,+2,–2,+3,–3,+4,–4 and 0 times the multiplicand need to be calculated. The *3 is the hard term to calculate, requiring an adder. However, in some circumstances a Radix-8 multiplier might be appropriate.

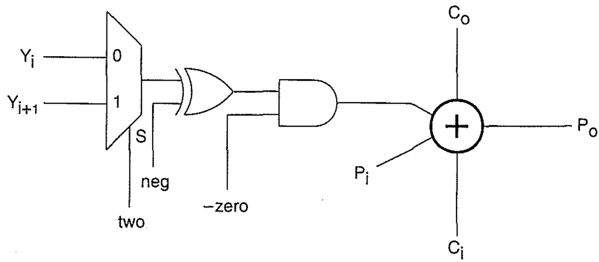
8.2.7.3 Wallace Tree Multiplication

If Table 8.1, showing the truth table for an adder, is examined, it may be seen that an adder is in effect a "one's counter" that counts the number of 1's on the *A, B,* and *C* inputs and encodes them on the *SUM* and *CARRY* outputs. Table 8.4 below summarizes this.

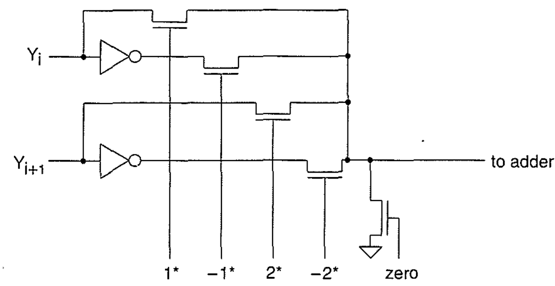
Micron Ex. 1037, p. 593
Micron v. YMTC
IPR2025-00119



(a)



(b)



(c)

FIGURE 8.39 Booth-recoded multiplier cells

TABLE 8.4 An Adder as a 1's Counter

ABC	CS	Number of 1's
000	00	0
001	10	1
010	10	1
011	01	2
100	01	1
101	10	2
110	10	2
111	11	3

Micron Ex. 1037, p. 594
Micron v. YMTC
IPR2025-00119

A 1-bit adder provides a 3:2 compression in the number of bits. The addition of partial products in a column of an array multiplier may be thought of as totaling up the number of 1's in that column, with any carry being passed to the next column to the left. Consider the 6 × 6 multiplication table shown in Table 8.5.

Considering the product P5, it may be seen that it requires the summation of six partial products and a possible column carry from the summation of P4. Figure 8.40 enumerates the adders required in a multiplier based on this style of addition. The adders have been arranged vertically into ranks that indicate the time at which the adder output becomes available. While this small example shows the general Wallace addition technique, it does not show the real speed advantage of a Wallace tree. In Fig. 8.40 there is an identifiable "array part" and a CPA part, which is at the top right. While this has been shown as a ripple-carry adder, any fast CPA can be used here. The delay through the array addition (not including the CPA) is proportional to log (base3/2) n, where n is the width of the Wallace tree. In a simple array multiplier it is proportional to n. So in a 32-bit multiplier where the maximum number of partial products is 32, the compressions (3:2 compressors) are

$$32 \rightarrow 22 \rightarrow 16 \rightarrow 12 \rightarrow 8 \rightarrow 6 \rightarrow 4 \rightarrow 3 \rightarrow 2.$$

Thus there are 9 adder delays in the array. In an array multiplier (Booth-recoded) there are 16 (note that the Booth recoding may also be used with a Wallace tree adder). To get the total addition time, the final CPA time has to be added to the array propagation times. For a 64-bit multiplier the comparison is 11 for a Wallace tree versus 32 for an array.

Apart from 3:2 compression, 4:2 compression (really 5:3) is often used. An improvement over two cascaded adders may be achieved by using the 4:2 compressor shown in Fig. 8.41. This has three XOR delays in the sum path rather than the four that would be present if two adders were used. A regular

TABLE 8.5 A 6 × 6 Multiplier

						X5	X4	X3	X2	X1	X0	Multiplicand
						Y5	Y4	Y3	Y2	Y1	Y0	Multiplier
						X5Y0	X4Y0	X3Y0	X2Y0	X1Y0	X0Y0	
					X5Y1	X4Y1	X3Y1	X2Y1	X1Y1	X0Y1		
				X5Y2	X4Y2	X3Y2	X2Y2	X1Y2	X0Y2			
			X5Y3	X4Y3	X3Y3	X2Y3	X1Y3	X0Y3				
		X5Y4	X4Y4	X3Y4	X2Y4	X1Y4	X0Y4					
	X5Y5	X4Y5	X3Y5	X2Y5	X1Y5	X0Y5						
P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0	Product

Micron Ex. 1037, p. 595
Micron v. YMTC
IPR2025-00119

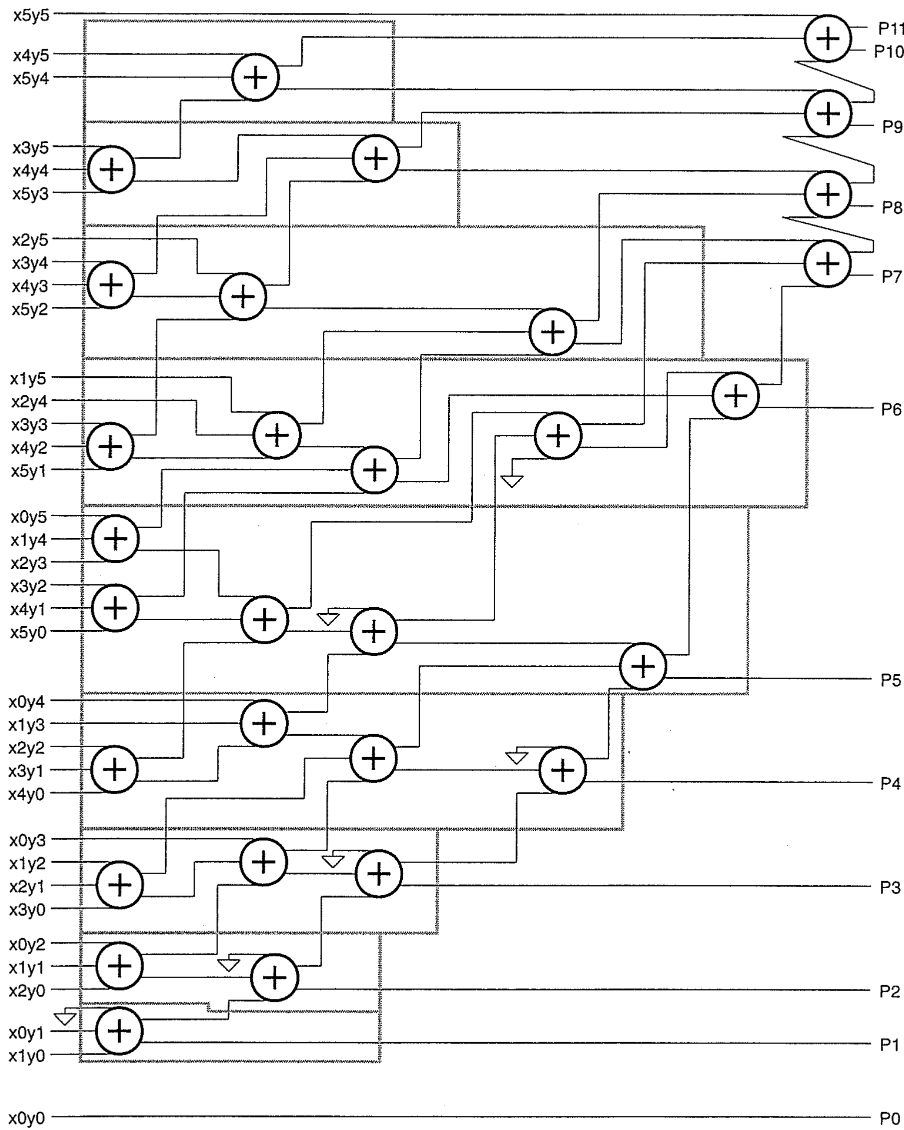


FIGURE 8.40 Wallace adder tree (for 6 × 6 multiplier)

layout for a 54-by-54 bit multiplier using the compressor shown in Fig. 8.41 may be found in Goto et al.[16]

8.2.7.4 Serial Multiplication

Multiplication may be performed serially. The simplest form of serial multiplier, shown in Fig. 8.42, uses the successive addition algorithm and is implemented using a full adder, a logical AND circuit, a delay element (i.e., either static or dynamic flip-flop), and a serial-to-parallel register.

Micron Ex. 1037, p. 596
Micron v. YMTC
IPR2025-00119

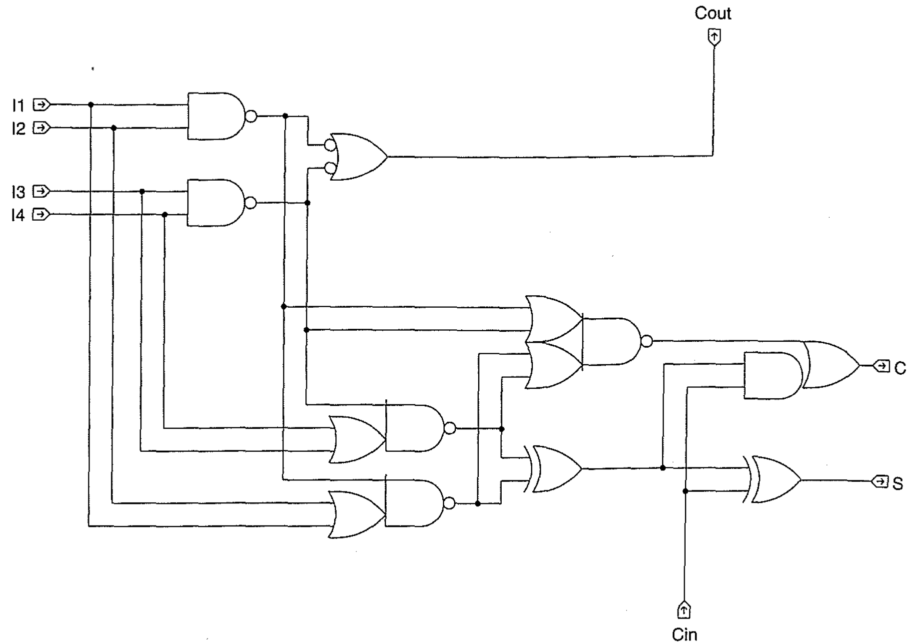


Figure 8.41 A 4:2 (5:3) compressor circuit

The two numbers X and Y are presented serially to the circuit (at different rates to account for multiplier and multiplicand word-lengths). The partial product is evaluated for every bit of the multiplier, and a serial addition is performed with the partial additions already stored in the register. The AND gate ($G2$) between the input to the adder and the output of the register is used to reset the partial sum at the beginning of the multiplication cycle. If the register is made of $N-1$ stages, then the 1-bit shift required for each partial product is obtained automatically. As far as the speed of operation is concerned, the complete product of $M+N$ bits can be obtained in MN intervals of the multiplicand clock.

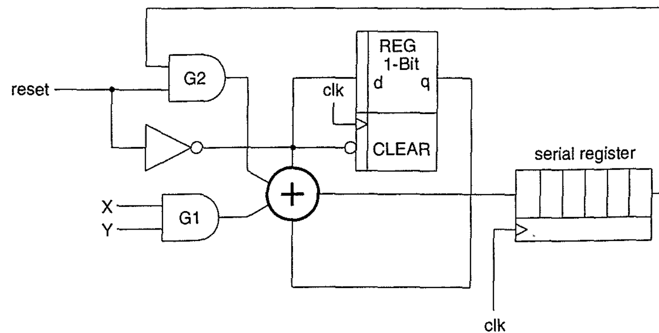


FIGURE 8.42 Serial multiplier

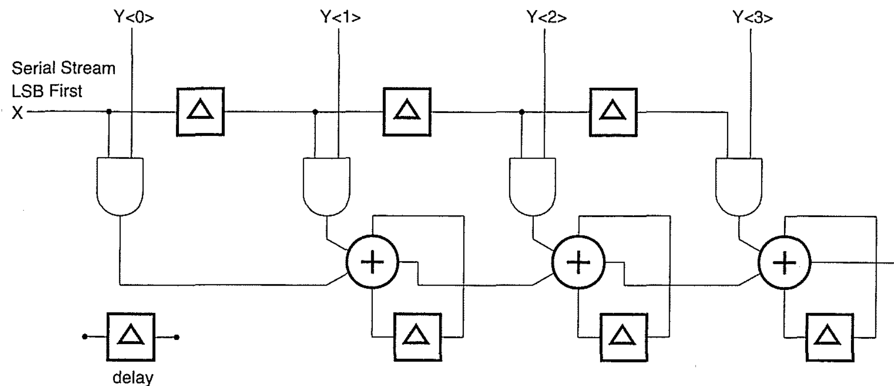


FIGURE 8.43 Serial/parallel multiplier

Using the general approach discussed previously, it is possible to realize a serial/parallel multiplier with a very modular structure that can easily be modified to obtain a pipelined system. The basic implementation is illustrated by Fig. 8.43. In this structure, the multiplication is performed by means of successive additions of columns of the shifted partial products matrix. As left-shifting by one bit in serial systems is obtained by a l-bit delay element, the multiplier is successively shifted and gates the appropriate bit of the multiplicand. The delayed, gated instances of the multiplicand must all be in the same column of the shifted partial-product matrix. They are then added to form the required product bit for the particular column.

This structure requires $M + N$ clock cycles to produce a product. The main limitation is that the maximum frequency is limited by the propagation through the array of adders. The structure of Fig. 8.43 can be pipelined with the introduction of two delay elements in each cell, as shown in Fig. 8.44. If rounding or truncation of the product term to the same word length as the input is tolerated, then the time necessary to produce a product is 2M clock cycles. In this case the multiplier accumulates partial product sums, starting with the least significant partial product. After each addition, the result is an

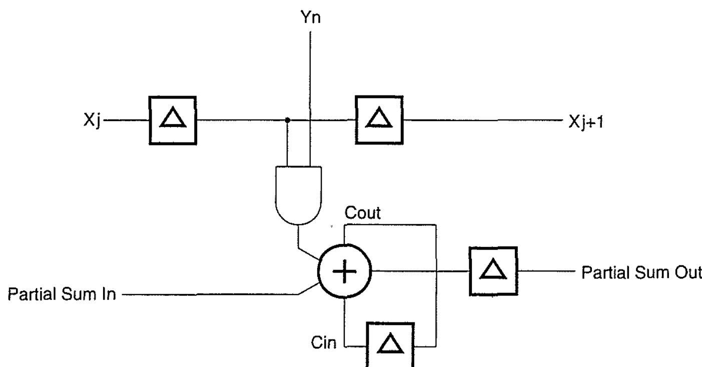


FIGURE 8.44 Pipelined serial/parallel multiplier

Micron Ex. 1037, p. 598
Micron v. YMTC
IPR2025-00119

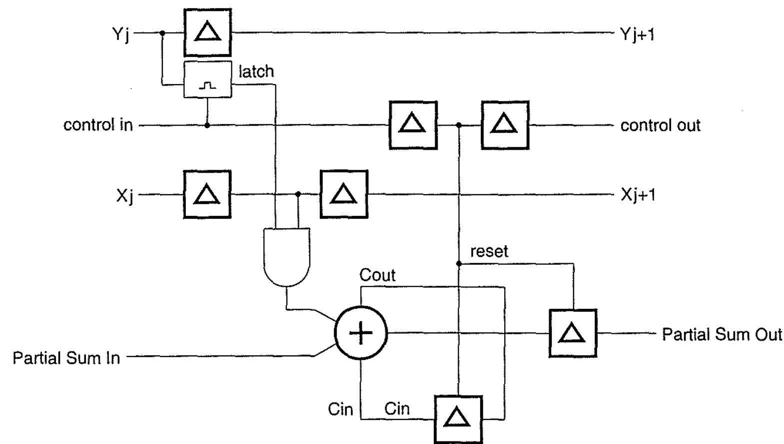


Figure 8.45 Lyon serial multiplier

This multiplier uses LSB first because this is the format that more naturally caters for addition and multiplication.

N-bit number that shortens to N-1 bits before the next partial product is added. Here, it can be noted that the chip area increases linearly with the length of the multiplier.

Figure 8.45 shows a schematic of a two-stage serial-multiplier stage based on the work of Lyon[17], in which the basic solution described so far has been modified so that both words are in serial form.[18] Multipliers of this type are frequently useful in FPGAs.

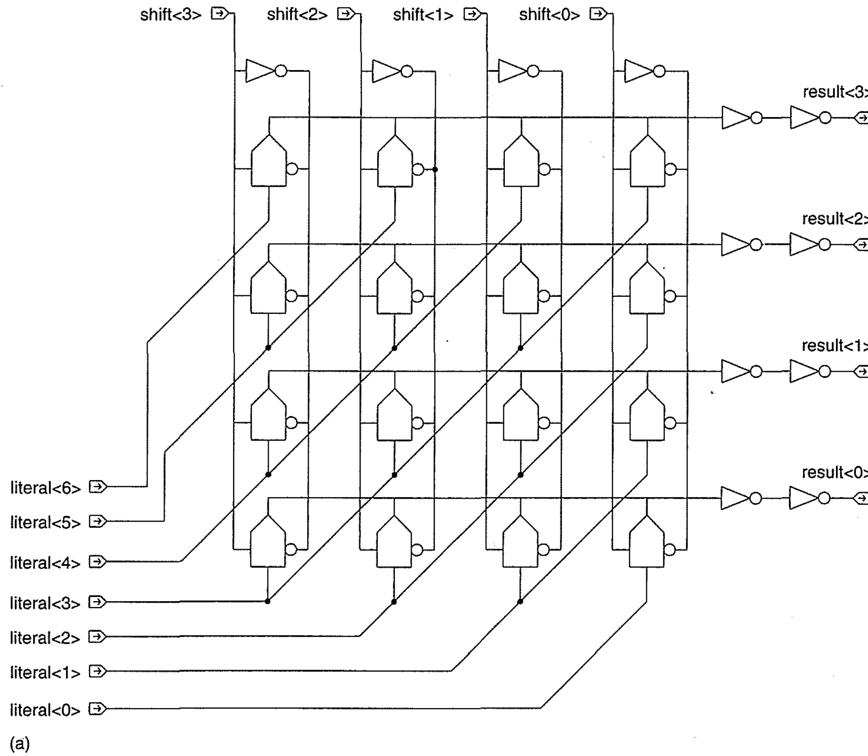
8.2.8 Shifters

Shifters are important elements in many microprocessor designs for arithmetic shifting, logical shifting, and rotation functions. A 4-by-4 barrel shifter is shown in Fig. 8.46(a), constructed from complementary transmission gates. The input to the shifter is the value to be shifted a (*literal<6:0>*) and the shift amount (*shift<3:0>*). Table 8.6 shows the value of the output (*result<3:0>*) for various values of shift and literal.

The function performed depends on the connections of the literal bus. These connections may be made with an additional multiplexer on the front of the shift matrix. Table 8.7 shows the functions.

Both arithmetic and logical shifts are implemented as well as rotates. Figure 8.46(b) shows a symbolic layout for the core transmission gate. The control lines have been run in polysilicon, assuming either that silicided poly is used or that these signals are set up well in advance of the literal input. Other layouts that do not use polysilicon are of course possible. While the

Micron Ex. 1037, p. 599
Micron v. YMTC
IPR2025-00119



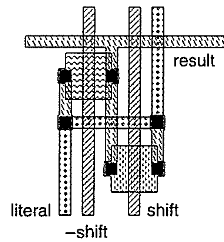


FIGURE 8.46 Array shifter using transmission gates: (a) circuit; (b) cell layout

TABLE 8.6 Shifter Operations

SHIFT	RESULT
1	LITERAL<3:0>
2	LITERAL<4:1>
4	LITERAL<5:2>
8	LITERAL<6:3>

TABLE 8.7 Modified Shifter Operations

LITERAL<6:0>	OPERATION
VSS,VSS,VSS,A<3:0>	LOGICAL RIGHT SHIFT
A<3>,A<3>,A<3>,A<3:0>	ARITHMETIC RIGHT SHIFT
A<3:0>,VSS,VSS,VSS	LOGICAL LEFT SHIFT
A<3:0>,A<2:0>	LEFT ROTATE
A<2:0>,A<3:0>	RIGHT ROTATE

circuit shown in Fig. 8.46(a) is fine for transistor level design, it is not really appropriate for a gate-level implementation.

Figure 8.47 shows a shifter that uses multiplexers (which of course can be transmission gates). An implementation for a logical left shift, arithmetic right shift is shown. The shifter is divided into two halves, one of which

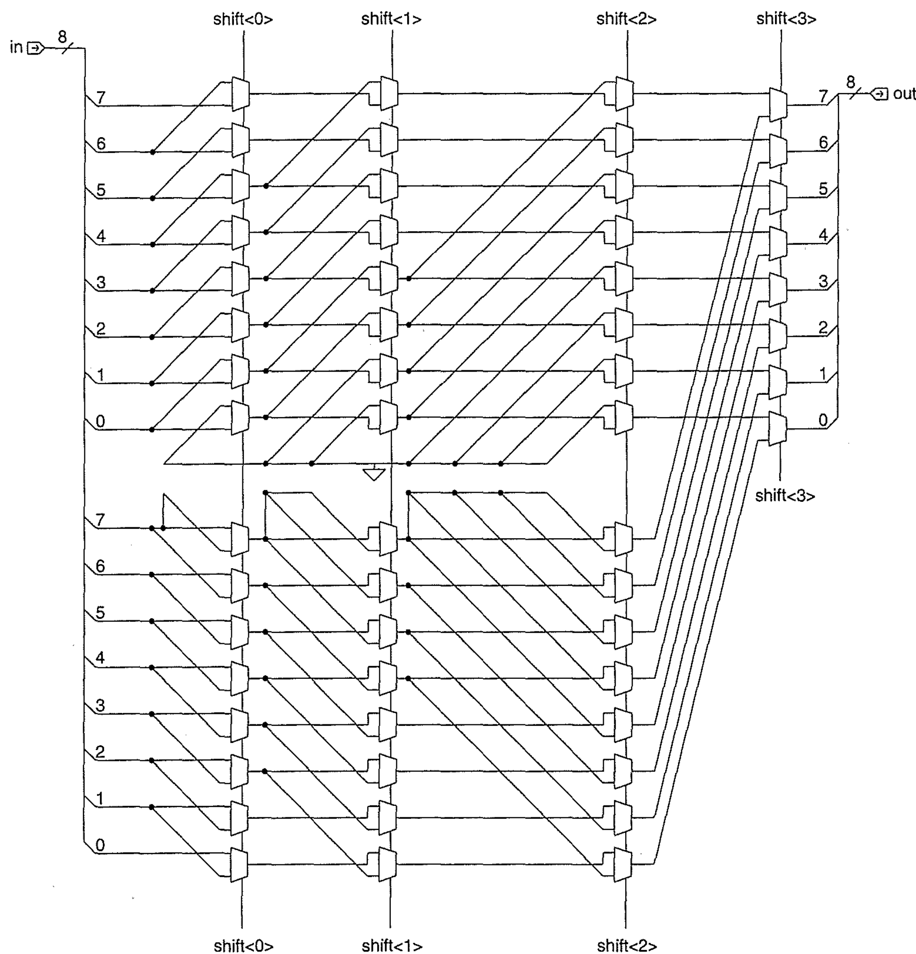


Figure 8.47 Multiplexer-based shifter

Micron Ex. 1037, p. 601
Micron v. YMTC
IPR2025-00119

shifts right and one of which shifts left. The fill values can be set by appropriate connections at the ends of the shifter ranks. The output of the two shifters is muxed to form a final result. The value of *SHIFT*<2:0> gives the amount of the shift with *SHIFT*<3> = 1 producing a left shift, while *SHIFT*<3> = 0 produces an arithmetic right shift. Left and right rotates may be implemented by wrapping the end connections conditionally to the opposite end bits.

Shifters implemented with transmission gates are notorious for fooling timing analyzers unless the directionality of the pass transistors are somehow communicated to the timing analyzer. The multiplexer shifter may use buffered (inverting, if need be) multiplexers, which can aid in speeding up the long lines in large shifters. The multiplexer version directly takes the shift amount as control, while the array version requires an $n{:}m$ decoder (2:4 for the one shown in Fig. 8.46a). For these reasons the multiplexer version may be favored in CMOS although the version shown in Fig. 8.46 can be compact.

Other shift options are frequently required, for instance, shuffles, bit-reversals, and interchanges. One can either use the complementary transmission gate, static single-pass transistors (usually n-channel). Precharged versions of single-pass transistor shifter circuits are generally cumbersome. Large capacitances can be associated with the intermediate mux nodes and these must all be precharged to prevent charge-sharing problems. The speed of an n-bit shifter is proportional to $\log(n)$, so combined with the fast speed of transmission gates, shifting can be a fast operation.

8.3 Memory Elements

Memory elements form critical components in the implementation of CMOS systems. While off-the-shelf memories are limited by the number of I/O pins, the speed of driving into the chip, and large off-chip output nodes, on-chip memories can be engineered to be very fast and to have unique access paths. In general, CMOS ASIC processes will not compete with the density of state-of-the-art DRAM memory, but may be very competitive with high-speed static memories. Memory elements may be divided into the following categories:

- Random access memory.
- Serial access memory.
- Content access memory.

Random access memory at the chip level is classed as memory that has an access time independent of the physical location of the data. This is contrasted with serial-access memories, which have some latency associated

Micron Ex. 1037, p. 602
Micron v. YMTC
IPR2025-00119

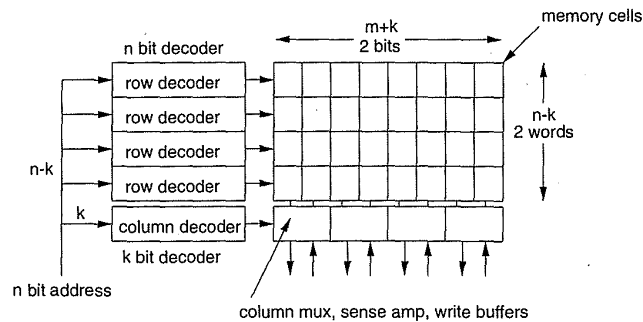


FIGURE 8.48 Memory-chip architecture

with the reading or writing of a particular datum and with content-addressable memories. Within the general classification of random access memory, we can consider read only memory (ROM) or read/write memory (commonly called RAM). ROMs usually have a write time much greater than their read time (programmable ROMs have write times of the order of milliseconds), while RAMs have very similar read and write times. Both types of memory may be further divided into static-load, synchronous, and asynchronous categories. Static-load memories require no clock. Synchronous RAMs or ROMs require a clock edge to enable memory operation. The address to a synchronous memory only needs to be valid for a certain setup time after the clock edge. Asynchronous RAMs recognize address changes and output new data after any such change. Static-load and synchronous memories are easier to design and usually form the best choice for a system-level building block, because they can generally be clocked by the system clock.

The memory cells used in RAMs can further be divided into static structures and dynamic structures. Static cells use some form of latched storage, while dynamic cells use dynamic storage of charge on a capacitor. We will concentrate on static RAMs because they are easier to design and potentially less troublesome than dynamic RAMs. Static RAMs tend to be faster (but much larger) than dynamic RAMs.

A typical memory-chip architecture is shown in Fig. 8.48. Central to the design is a memory array consisting of 2^n by 2^m bits of storage (actually 2^{n-k} by 2^{m+k}). A row (or word) decoder addresses one word of 2^m bits out of 2^{n-k} words. The column (or bit) decoder addresses 2^k of 2^m bits of the accessed row. This column decoder accesses a multiplexer, which routes the addressed data to and from interfaces to the external world.

8.3.1 Read/Write Memory

8.3.1.1 RAM

Figure 8.49 shows one row and one column of a generic RAM architecture with the support circuits required by the RAM cell. The row decoder is a 1 of $n-k$ decoder which may generally be thought of as an AND gate. One of the 2^{n-k}

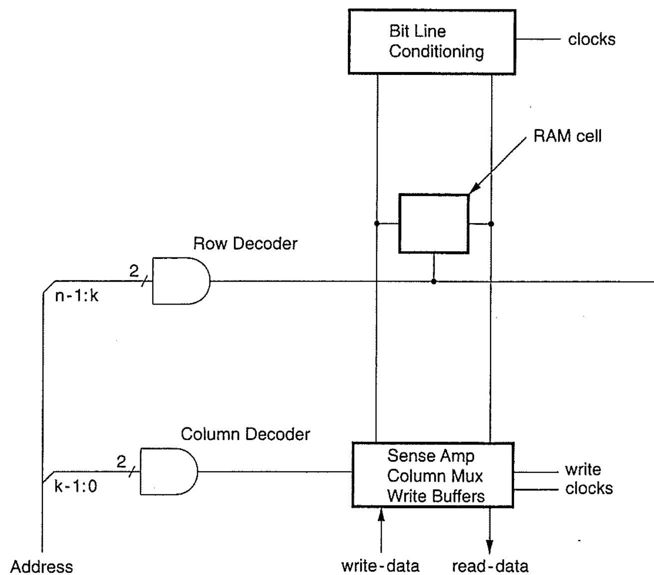


FIGURE 8.49 Generic RAM circuit

row lines is accessed at one time. The bit-line-conditioning circuitry, the ram-cell, the sense amplifiers, column multiplexers, and the write buffers form a tightly coupled circuit that provides for the hazard-free reading and writing of the memory cell. The bit lines are normally run as complementary signals. There are many variations of these circuits to achieve varying density/speed/noise–margin requirements. We shall look at a variety of schemes for implementing static RAMs. The column decoder is similar to the row decoder but is a 1 of k decoder. k is normally less than n and the decoder drives a multiplexer (rather than a selector). Frequently, the column decoder may be merged with the column multiplexer.

Starting with the RAM cell itself, various circuits are shown in Fig. 8.50.[19] The most commonly used in ASIC memories is the 6-transistor, cross-coupled inverter circuit shown in Fig. 8.50(a). A typical mask-level layout for a 6-transistor circuit is shown in Fig. 8.51 (also Plate 8). The p-transistors may be replaced with high-value polysilicon resistors if the process supports this option (Fig. 8.50b). The value of the resistor has to be such that it prevents leakage from changing any value stored in the RAM cell. Generally the resistors are in the 100's to 1000's of Megaohms. Deleting one of the bit-line pass transistors results in a 5-transistor RAM cell. Writing such a cell has to be considered carefully (see later in this section). A 4-transistor dynamic RAM cell may be achieved by deleting the p loads of the static cell, as shown in Fig. 8.52(a). This cell and the other dynamic cells have to be refreshed to retain the contents of the memory. A 3-transistor cell is shown in Fig. 8.52(b). The cell stores data on the gate of the storage transistor. Separate read and write control lines are used. Multiple read-ports may be added easily, by adding read transistors. In addition, separate or

Micron Ex. 1037, p. 604
Micron v. YMTC
IPR2025-00119

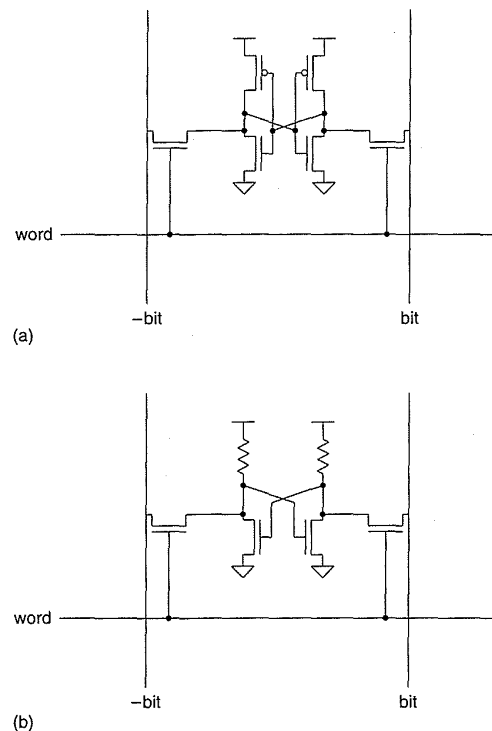


Figure 8.50 Static RAM cell circuits

merged read and write data busses may be used. A 1-transistor cell is shown in Fig. 8.52(c).[20] The memory value is again stored on a capacitor. The capacitor can be implemented as a transistor as shown in Figs. 8.52(d) and 8.52(e). Sense amplifiers sense the small change in voltage that results when a particular cell is switched onto the *bit* line. This type of cell (Fig. 8.52c) forms the basis for most high-density DRAMs.[21–28] The cell shown in Fig. 8.52(d) can be implemented in a conventional two metal, single poly process. The dominant problem that arises with this type of memory when used in an ASIC process is the loss of the stored charge due to leakage or stray substrate currents created by surrounding digital logic.

As far as the average CMOS-system design is concerned, the static 6-transistor cell should be used since it involves the least amount of detailed circuit design and process knowledge and is the safest with respect to noise and other effects that may be hard to estimate before silicon is available. In addition, current processes are dense enough to allow large static RAM arrays. As a general system-design principle, large amounts of memory should only be included in a design if the performance of the system is affected. Commercial RAM manufacturers are much better at designing RAMs than the average system designer. If dense memory can be partitioned

Micron Ex. 1037, p. 605
Micron v. YMTC
IPR2025-00119

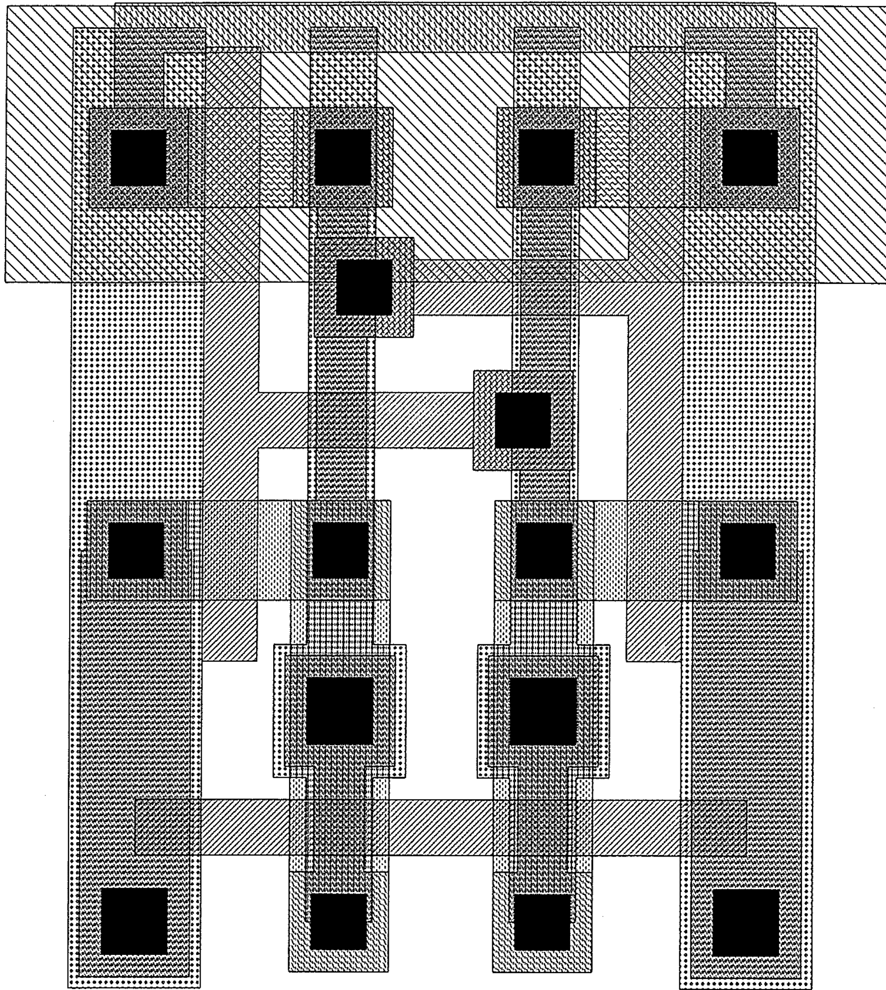


FIGURE 8.51 Mask layout for 6-transistor static RAM

off-chip with no performance degradation or cost impact, then this is a good approach to take.

8.3.1.1.1 Static RAM—read

We will begin our examination of CMOS static RAMs by considering a read operation. Imagine that the bit lines of the circuit shown in Fig. 8.49 are at some value and that the word line is asserted. The one node on the memory cell will attempt to pull the bit line up through the access transistor and the p load. The zero node will attempt to pull the bit line down through the access transistor and the n channel pull-down. As an n-channel transistor is poor at passing a one and the p-channel transistors in the RAM cell are generally small (or in the case of a resistive load, the resistors are very large), design of

Micron Ex. 1037, p. 606
Micron v. YMTC
IPR2025-00119

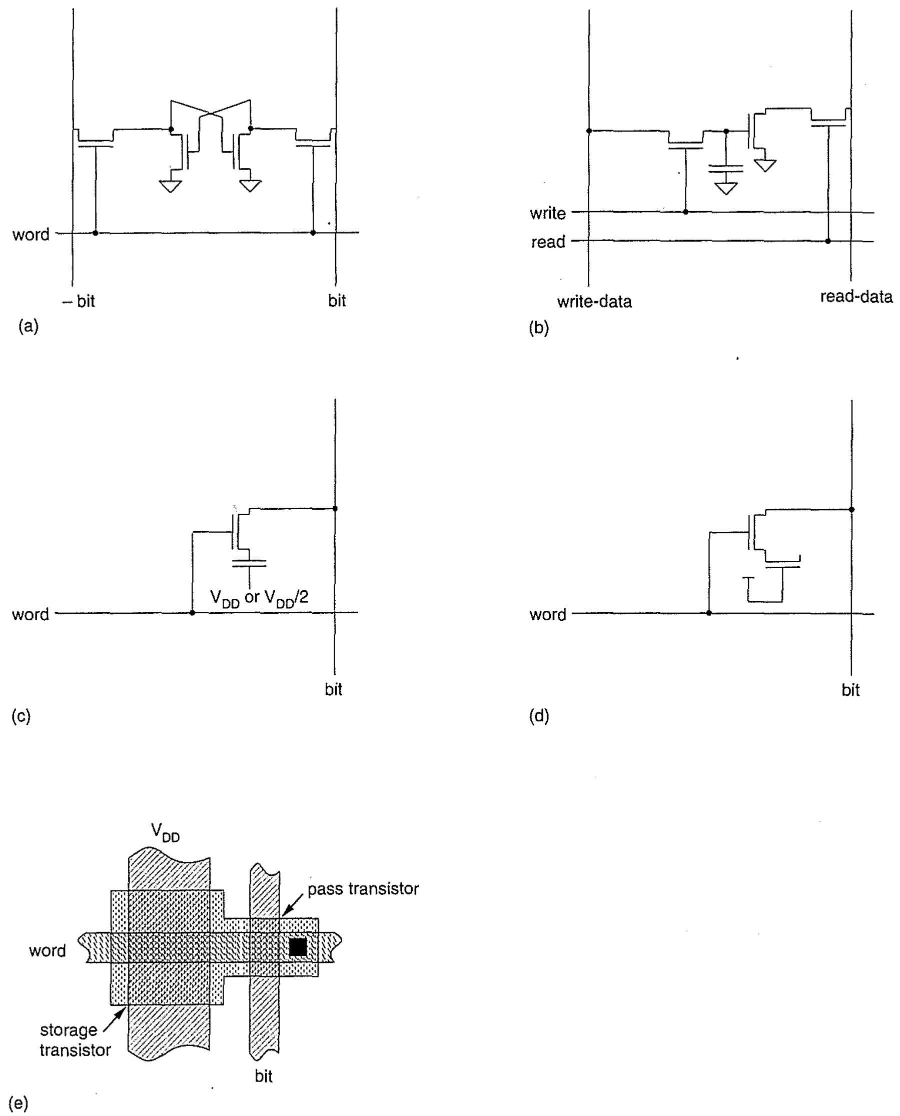


Figure 8.52 Dynamic RAM circuits: (a) 4-transistor; (b) 3 transistor; (c) 1 transistor with capacitor; (d) 1 transistor with transistor capacitor; (e) representative layout for (d)

the RAM circuit concentrates on pulling the bit line from high to low. Thus one method of reading a RAM cell would be to precharge the bit lines high and then enable the word-line decoder. For a given pair of bit lines, one RAM cell will attempt to pull down either the *bit* or *–bit* line depending on the stored data. The bit-line pull-up circuit may use p-channel transistors to precharge each bit line (Fig. 8.53a). In this example, the sense amplifier is an inverter that forms a single-ended sense amplifier. The sense time is roughly

Micron Ex. 1037, p. 607
Micron v. YMTC
IPR2025-00119

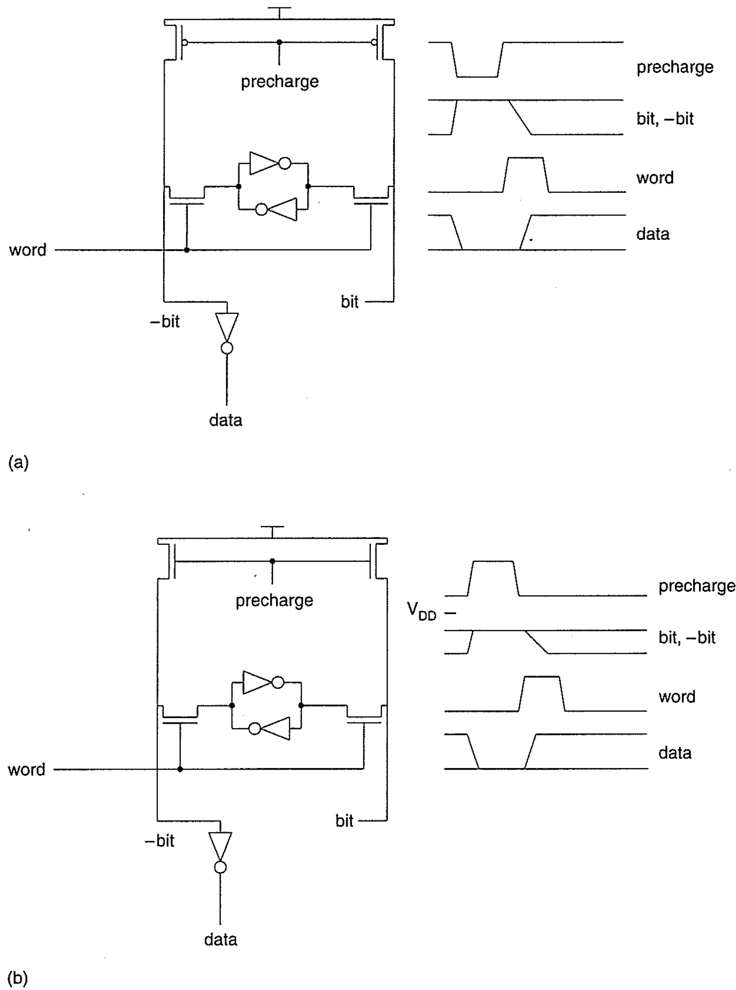


FIGURE 8.53 RAM read options: (a) V_{DD} precharge; (b) V_{DD}–V_{tn} precharge

the time it takes one RAM cell pull-down and access transistor to reach the inverter threshold. To optimize speed, one might set the inverter threshold above the V_{DD} midpoint, but below an adequate noise margin down from the V_{DD} rail. Alternatively, one can precharge the bit lines with n-channel transistors, which results in the bit lines being precharged to an n threshold down from V_{DD} (Fig. 8.53b). This can dramatically improve the speed of the RAM cell access. In addition, it reduces power dissipation because the bit lines do not change by the supply voltage. The key aspect of the precharged RAM read cycle is the timing relationship between the RAM addresses, the precharge pulse, and the enabling of the row decoder. If the word-line assertion precedes the end of the precharge cycle, the RAM cells on the active word-line will see both bit lines pulled high and the RAM cells may flip state. If

the addresses change after the precharge cycle has finished, more than one word line will be accessed and more than one RAM cell will have the chance to pull the bit lines down, leading to erroneous READ data. Normally, RAM designers generate a carefully designed timing chain than ensures the correct temporal relationships between precharge, row access, and sense operations.

A RAM access method that does not require precharge is shown in Fig. 8.54(a). Here n-channel load transistors pull up the bit lines statically. When the word line is asserted, the bit line being pulled down by the RAM cell falls to a value that is a function of the pull-up size, the pass-transistor size, and the RAM inverter pull-down size. At the same time, the pull-up must not be able to flip the RAM cell. A differential amplifier is used to amplify the bit-line difference. Figure 8.54(b) shows the equivalent circuit of the pull-down circuit during a read operation. Voltage V_1 must safely clear the input threshold of the RAM cell inverters. A value of .5–1V is appropriate. Voltage V_2 yields the bit-line difference voltage, which must be amplified to detect a transition on the bit line. The size of the bit-line load determines how fast the bit line can recover (to prevent false writes) after a write operation where the bit line may have been driven to V_{SS}. The sense amplifier is designed in conjunction with the bit-line pull-up and RAM cell to amplify this bit-line change. Design margins must be valid over all process, temperature, and voltage extremes. Figure 8.55 shows the zero bit voltage ($V_{bit(0)}$) and the pull-down voltage ($V_{pulldown}$) for various ratios of pull-up beta to pull-down betas. As the pull-up becomes weaker, the $V_{bit(0)}$ voltage approaches V_{SS} and the differential voltage between a high and a low on the bit lines increases. However, as the pull-down transistors are limited in size by the desire to keep the RAM cell small, a design trade-off has to be made between speed and the differential bit voltage, which affects the noise

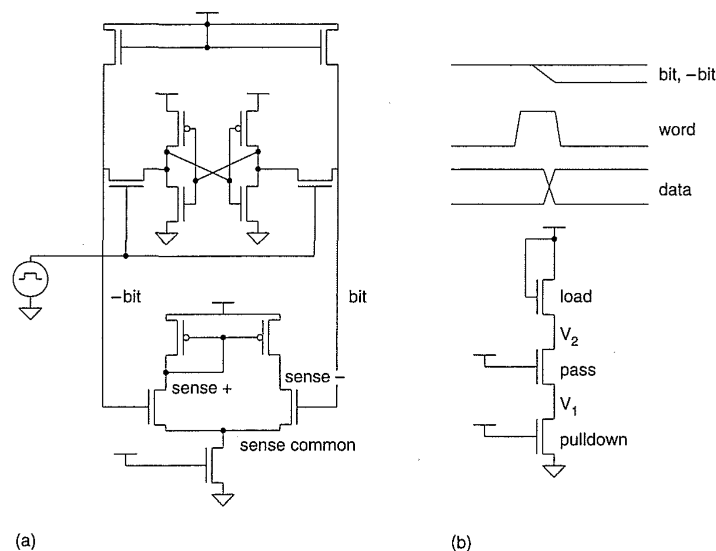


Figure 8.54 RAM read operation model

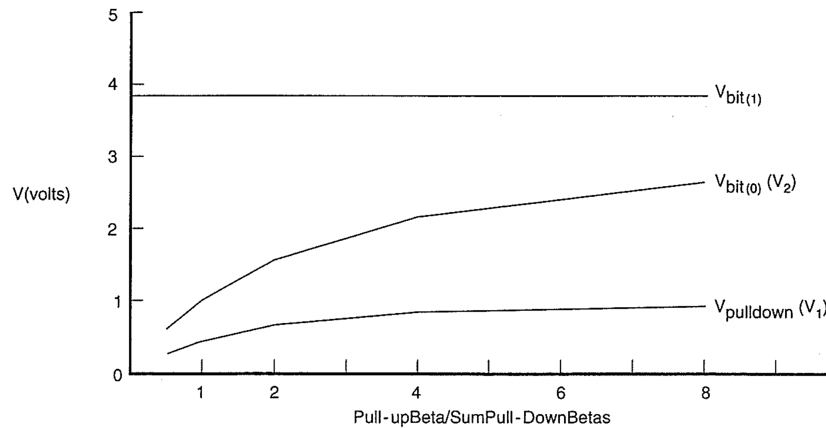


FIGURE 8.55 RAM bit-line voltage levels versus transistor size for static pull-up RAM

immunity of the cell and the write characteristics. To a first order, the bit-line voltage (V_2) is given by

$$V_2 = (V_{DD} - V_{tn}) \left(1 - \sqrt{\frac{1}{1 + \frac{\beta_{pullup}}{\beta_{driver\text{-}eff}}}} \right),$$

where β_{pullup} is the gain of the load and $\beta_{driver\text{-}eff}$ is the gain of the combination of the pass and pull-down transistor in series. When the gain of the pull-up is high compared with the pull-down path, the bit-line voltage rises towards $V_{DD} - V_{tn}$. When the gain of the pull-up is very small, the bit-line voltage approaches zero. The pull-down voltage V_1 is a result of resistive divider action between the word-access transistor and the RAM-cell pull-down. While these transistors are in the linear region, V_1 is roughly given by

$$V_1 = V_2 \frac{\beta_{pass}}{\beta_{pass} + \beta_{pulldown}}.$$

The RAM cell and the sense amplifier draw static current, which affects power dissipation. Figure 8.56 shows typical SPICE waveforms for the word line, bit lines, and sense amplifier. In this design the bit line pulls down to about 2 volts, while the bit-line high level is about 4 volts. During access, the RAM cell low value is pulled up to about 1 volt, leaving about 1 volt of margin to the switching point of the RAM cell inverter. The sense amplifier can be seen starting to switch just as the bit lines start diverging. The period between word line deassertion and *bit* nearing-*bit* is the *recovery time* (during which no other word line should be asserted in order to prevent false writes).

Micron Ex. 1037, p. 610
Micron v. YMTC
IPR2025-00119

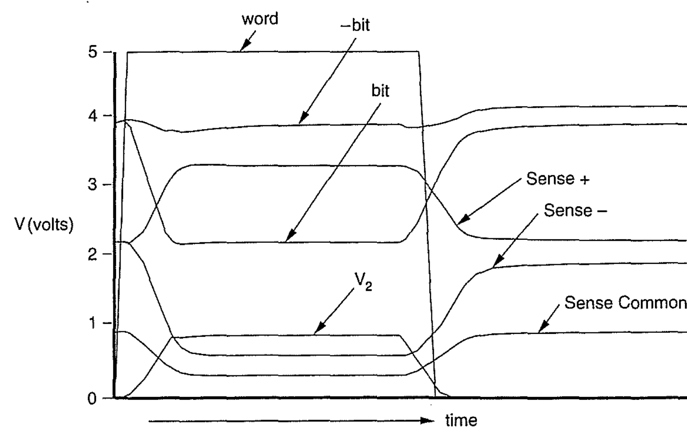


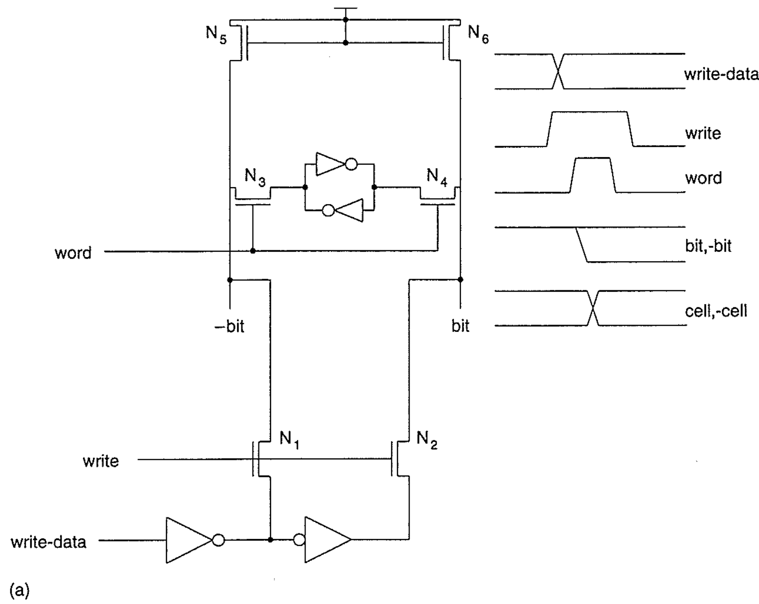
Figure 8.56 Static RAM–read waveforms

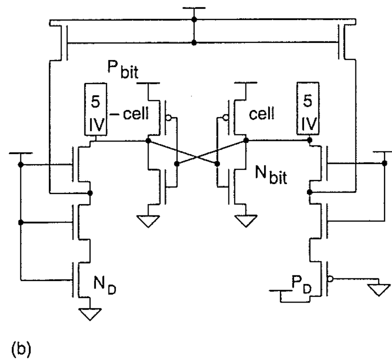
Current mode sensing may also be used.[29,30,31] In this technique, the current change in the bit lines is detected using special circuits. The theory is that by using low-impedance circuits, the RC delay inherent in driving the bit lines may be decreased.

8.3.1.1.2 Static RAM—write

The objective of the RAM write operation is to apply voltages to the RAM cell such that it will flip state (a condition we do not desire during the read operation). Figure 8.57(a) shows a straightforward write circuit. In this circuit, the write-enable transistors (N_1,N_2) are enabled to allow the data and complement to move to the bit lines. The word line is then asserted (actually the turn-on order is not important). Either the *bit* or *–bit* line is driven to V_{SS}, while the other bit line is driven to a threshold down from V_{DD}. Figure 8.57(b) shows a more detailed view of the situation. The figure shows a zero stored in the cell. During a WRITE cycle where a one is to be written, node *–Cell* has to be pulled below the RAM-cell inverter threshold and at the same time node *Cell* has to be pulled above the RAM-cell inverter threshold. In the former case, n-transistors N_D (the driver n-transistor), N_1 (the write-access transistor), and N_3 (the word-access transistor) have to pull P_{bit} (the RAM inverter pull-up) below the inverter threshold. In addition N_5 (the bit-line pull-up) has to be pulled low by N_1 and N_D. On the other bit-line side, P_D, N_2 and N_4 have to pull N_{bit} as high as possible. To augment the write operations it may be necessary to use complementary write-access transistors, as shown in Fig. 8.57(c). Correct WRITE operation must be verified over all process, temperature, and voltage extremes. Figure 8.58 shows a plot of the waveforms during a WRITE operation. The SPICE circuit used to model the RAM write operation is shown at the top of the figure. *write-data* and *–write-data* were driven antiphase into the write transistors N_2 and N_6. The cell switches when *–write-data* = 3V and *write-data* = 2V.

Micron Ex. 1037, p. 611
Micron v. YMTC
IPR2025-00119





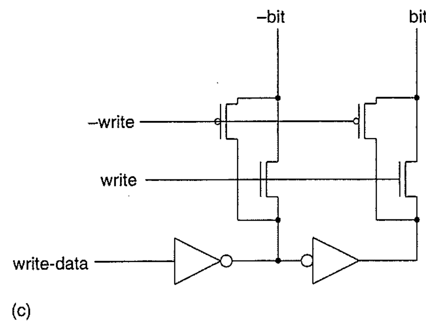


FIGURE 8.57 Static RAM–write circuits: (a) n-channel pass transistors; (b) circuit model during write; (c) complementary transmission gate version

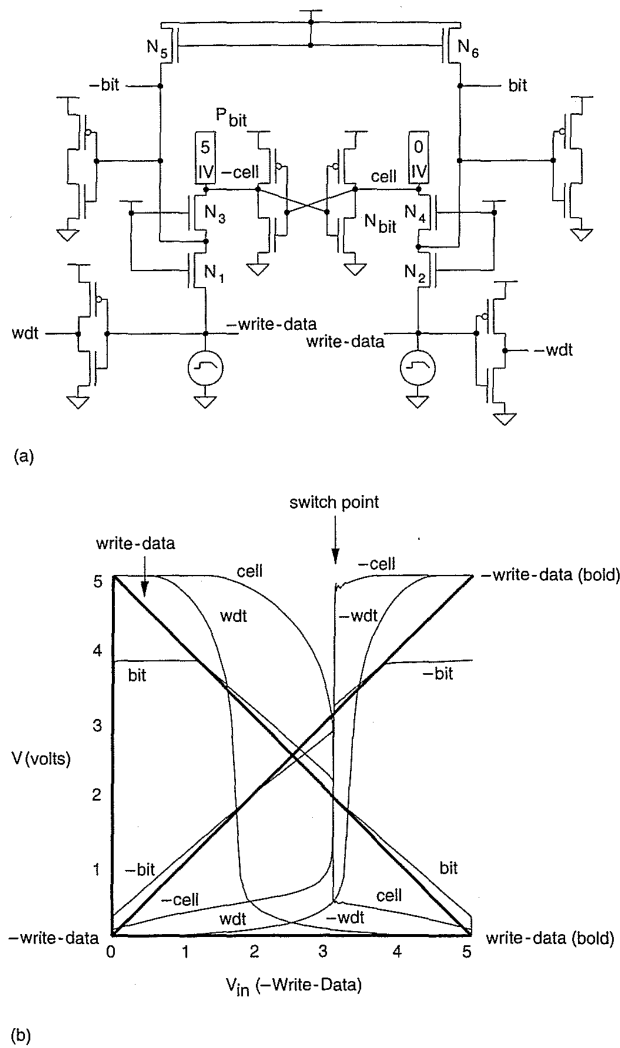


Figure 8.58 Static RAM–write waveforms and circuit model

8.3.1.1.3 Row decoders

The simplest row decoder is an AND gate. Figure 8.59 shows two straightforward implementations. The first in Fig. 8.59(a) is a static complementary NAND gate followed by an inverter. This structure is useful for up to 5–6 inputs or more if speed is not critical. The NAND transistors are usually made minimum size to reduce the load on the buffered address lines because there are 2^{n-k} $(N_{load} + P_{load})$'s on each address line. The second implementation, shown in Fig. 8.59(b), uses a pseudo-nMOS NOR gate buffered with two inverters. The NOR gate transistors can be made minimum size, and the inverters can be scaled appropriately to drive the word line. Large fan-in AND gates can also be constructed from smaller NAND and NOR gates, as shown in Fig. 8.59(c). Figure 8.60 shows two possible layout styles (in sym-

Micron Ex. 1037, p. 613
Micron v. YMTC
IPR2025-00119

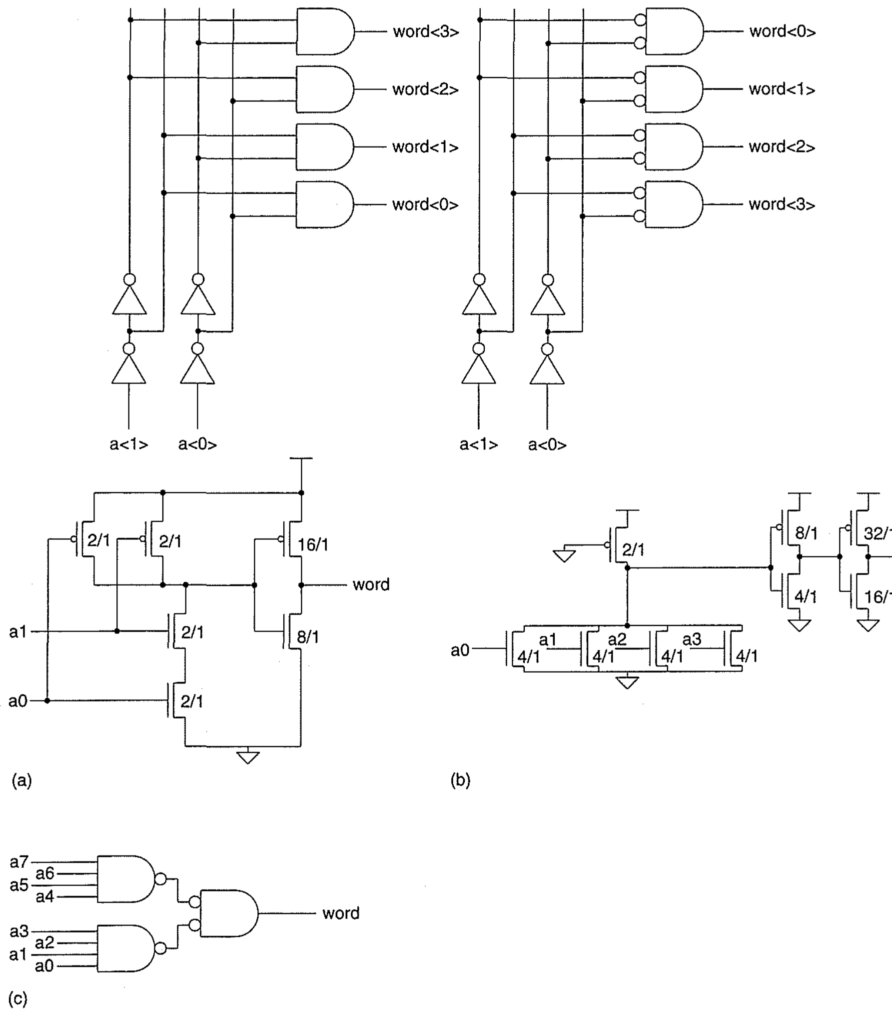


FIGURE 8.59 Row-decoder circuits: (a) complementary AND gate; (b) pseudo-nMOS gate; (c) cascaded NAND, NOR gates

bolic form) for the row decoders. One passes the address lines over the decode gates, while the other uses a more standard cell style. Choice would depend on the size of the decoder in relation to the size of the RAM cell. Often, speed requirements or size restrict the use of single-level decoding, such as that shown in Fig. 8.59. The alternative is a predecoding scheme, which is illustrated in Fig. 8.61(a). Here the (*n*-*k*) row address lines are split into a *p*-bit predecode field and a *q*-bit direct decode field. The *q*-bit decode field requires a gate per word line, so *q* is chosen to suit the pitch of the RAM cell. The *p*-bit predecode field generates 2^p predecode lines (4 in this example), each of which is fed vertically to 2^{n-k}-row decode gates (8 in this example). Figure 8.61(b) shows a possible implementation of a predecode scheme, where the predecode gate is a NAND gate and the word-decode gate is a NOR gate. An additional input (*–clk*) has been included in the NOR gate

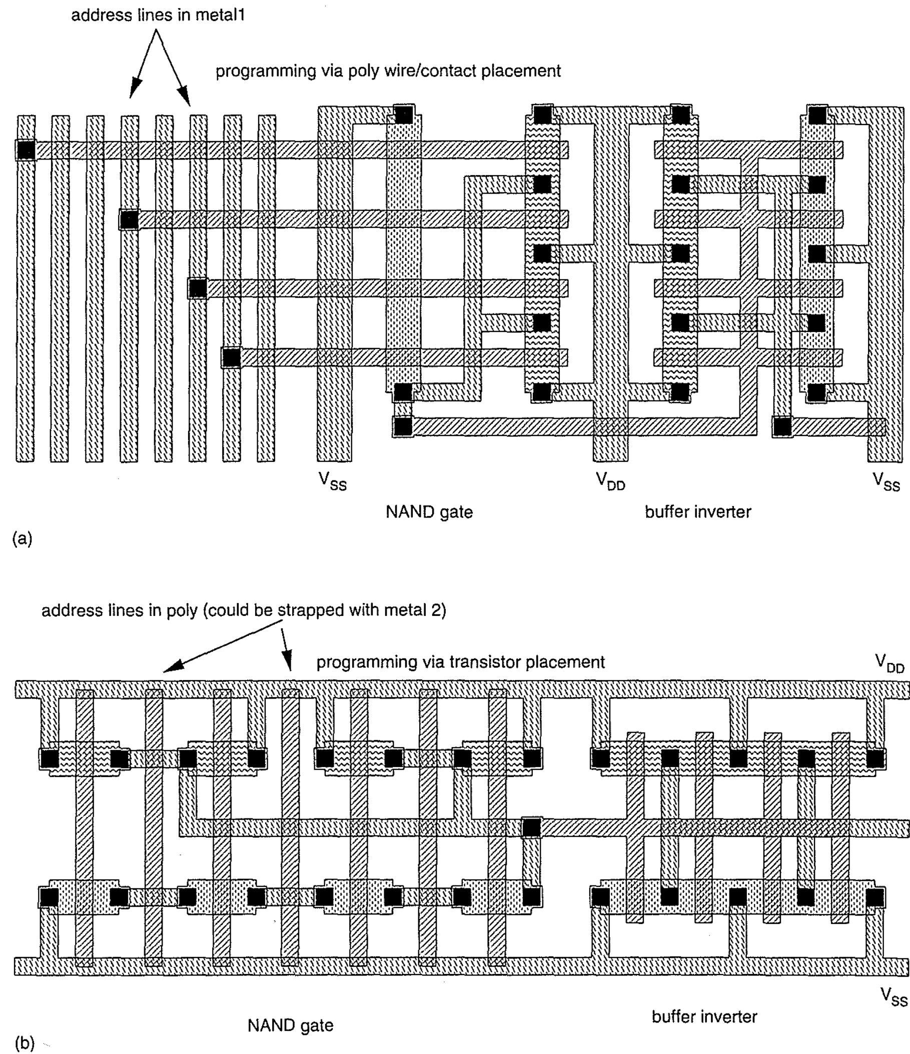


Figure 8.60 Typical symbolic layouts of row decoders

to allow the enabling of the gate, which is necessary to ensure correct timing of the word signal. A slow rise time and fast fall time on a word-decode gate might be advantageous because it ensures that any RAM cells on a word line transitioning low are isolated before RAM cells on a high-transitioning word line are accessed. Figure 8.61(c) shows a pseudo-nMOS AND row decode gate. Finally, Fig. 8.62 shows a few more row decoder circuits. Figure 8.62(a) shows some obvious ways of building large fan-in AND gates from smaller fan-in gates. Figure 8.62(b) is a pseudo-nMOS decoder that minimizes draw static power. Figure 8.62(c) shows a predecode scheme where the predecode gates power the word-line driver.[32] Figure 8.62(d) shows a domino dynamic AND gate implementation.

8.3.1.1.4 Column decoders

The column decoder is responsible for selecting 2^k out of 2^m bits of the accessed row. A tree decoder is shown in Fig. 8.63. Here the data is routed

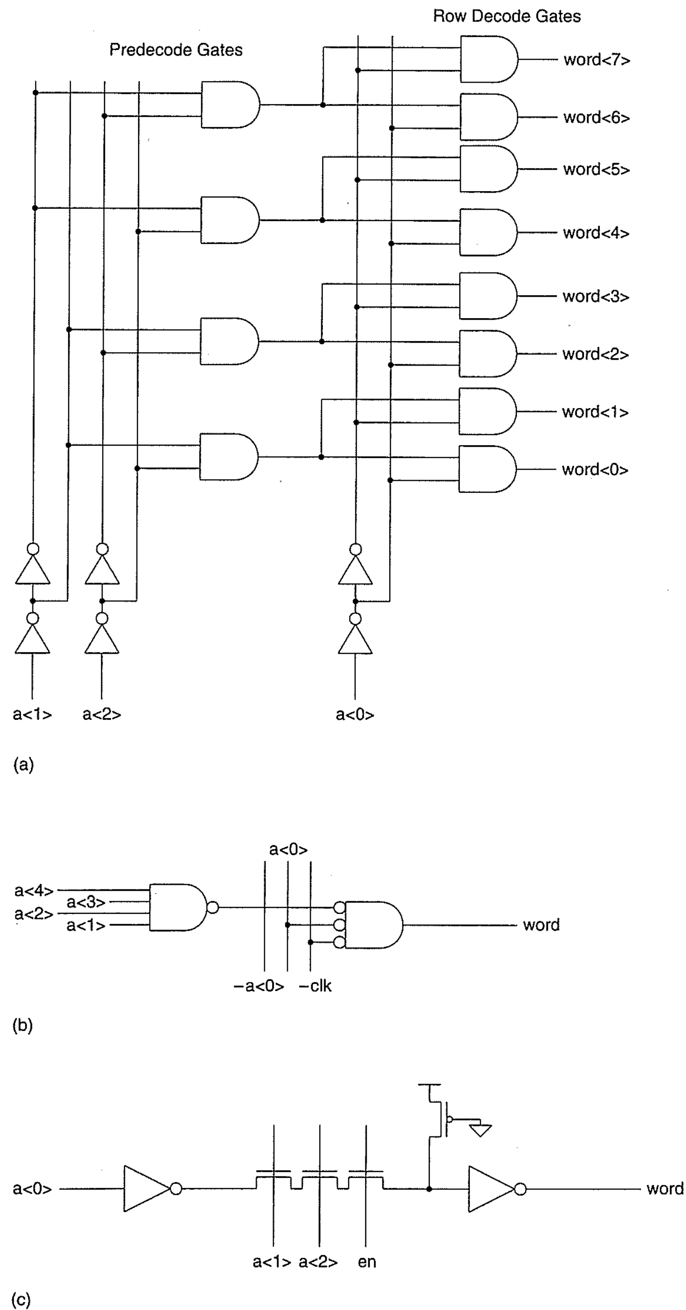


FIGURE 8.61 Predecode circuits: (a) basic approach; (b) actual implementation; (c) pseudo-nMOS example

via pass gates enabled by the column-address lines. The address decoding is in essence distributed. Decoders for *bit* and *–bit* lines are shown, although one of these may be omitted for single-ended read operations. The read (and, usually of lesser importance, write) operations are somewhat delayed by the series-transmission gates. However, in comparison with gate delays these

Micron Ex. 1037, p. 616
Micron v. YMTC
IPR2025-00119

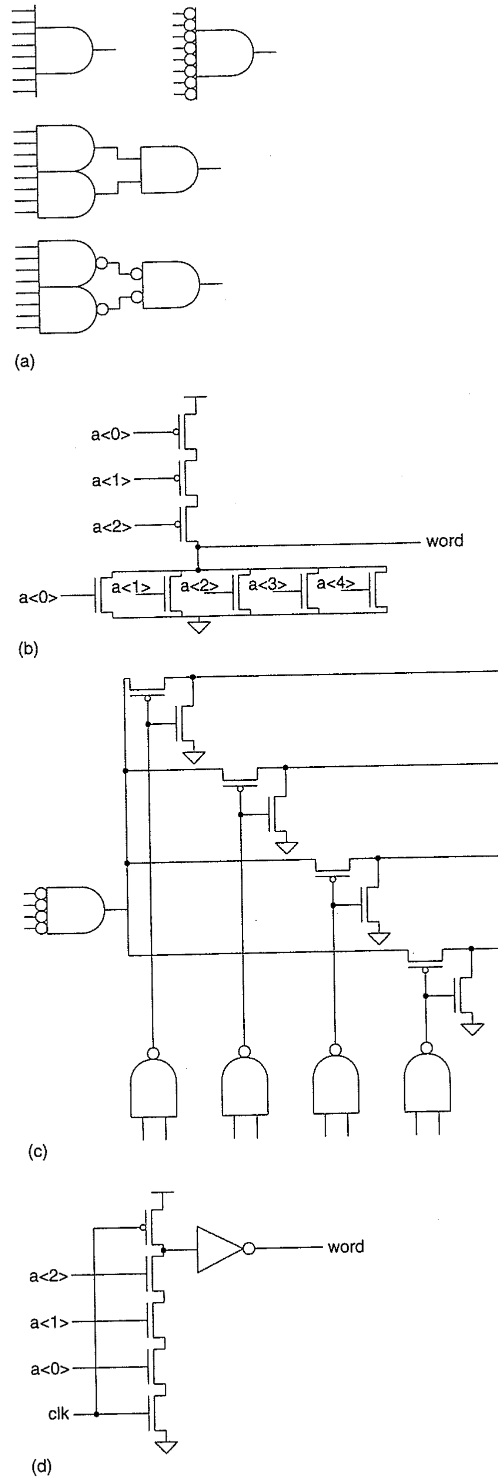


Figure 8.62 Various other row decoder circuits: (a) methods of building large fan-in AND gates; (b) power saving pseudo-nMOS gate; (c) decoder powered; (d) domino

usually are small for a low number of series transistors (2 to 4). Complementary transmission gates may also be used, if required, by either the read operation or write operation.

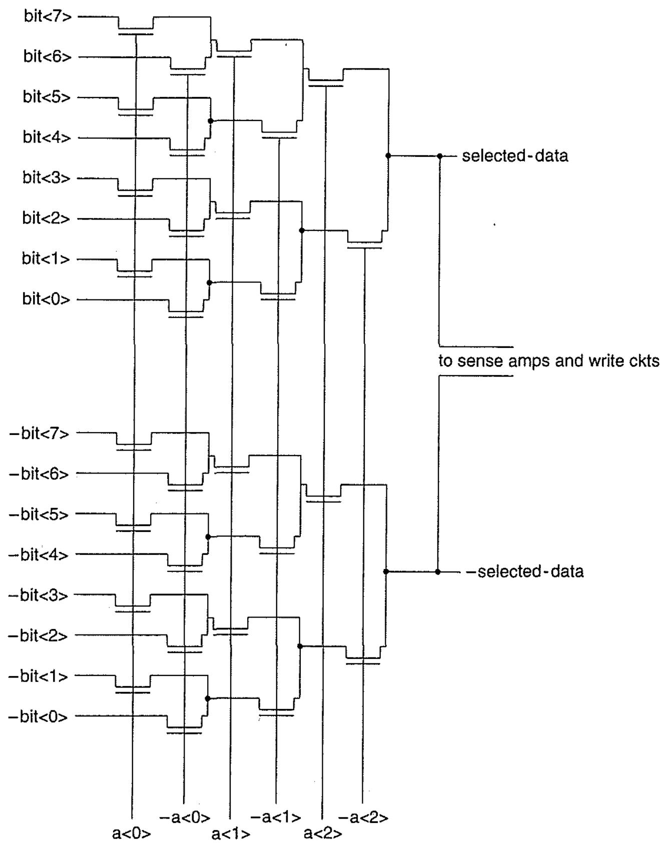


FIGURE 8.63 Tree-style column decoder

If the delay of the series-pass gates was troublesome, the decoder shown in Fig. 8.64 could be used. Here a NAND decoder is employed on a bit-by-bit basis to enable complementary transmission gates (single transistors may be used where possible) onto a common pair of data lines. These are then routed to a sense amplifier and write circuitry.

8.3.1.1.5 Sense amplifiers

Many sense amplifiers have been invented to provide faster sensing, smaller layouts, and lower power-dissipation sensing.[33] The simple inverter sense amplifier provides for low power sensing at the expense of speed. The differential sense amplifier can consume a significant amount of DC power (Fig. 8.54). Alternatively, one can employ clocked sense amplifiers similar to the SSDL gate shown in Fig. 5.40.

8.3.1.1.6 RAM timing budget

The critical path in a static RAM read cycle includes the clock to address delay time, the row address driver time, row decode time, bit-line sense time, and the setup time to any data register. The column decode is usually not in

Micron Ex. 1037, p. 618
Micron v. YMTC
IPR2025-00119

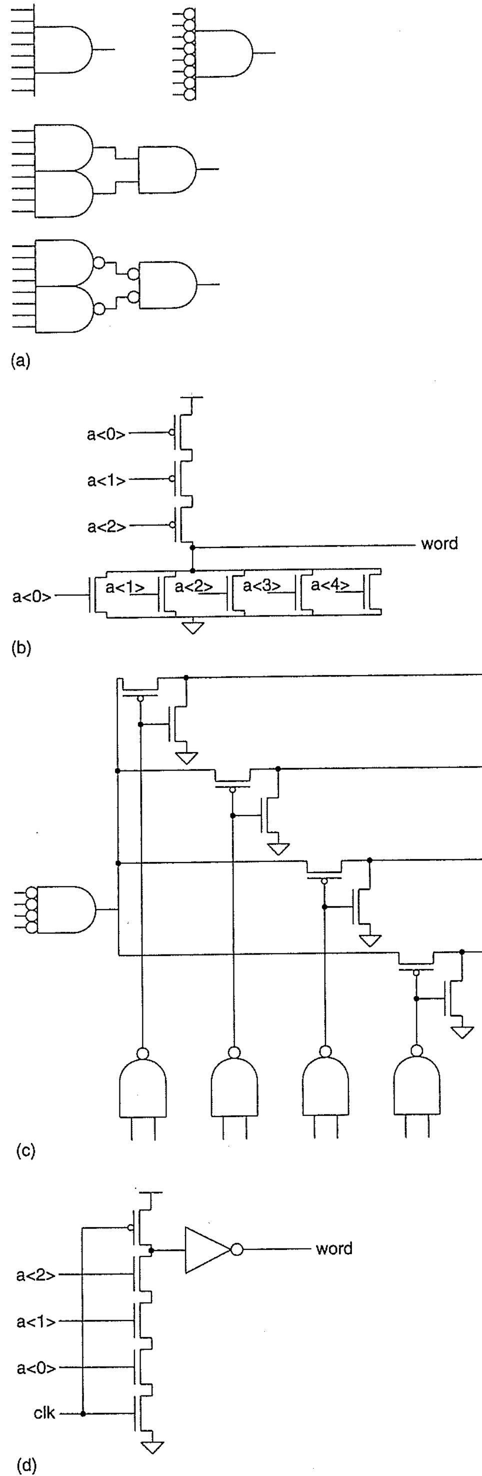


Figure 8.62 Various other row decoder circuits: (a) methods of building large fan-in AND gates; (b) power saving pseudo-nMOS gate; (c) decoder powered; (d) domino

usually are small for a low number of series transistors (2 to 4). Complementary transmission gates may also be used, if required, by either the read operation or write operation.

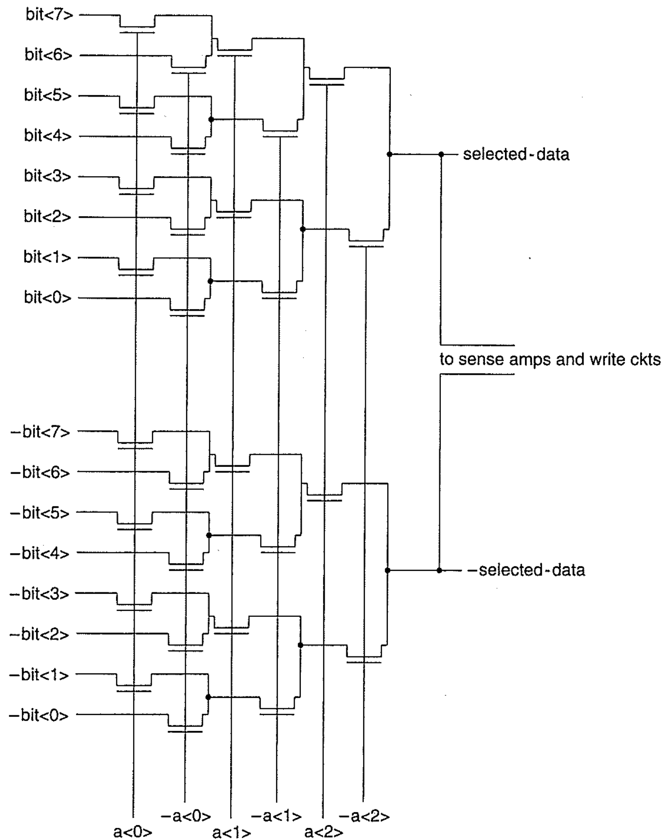


FIGURE 8.63 Tree-style column decoder

If the delay of the series-pass gates was troublesome, the decoder shown in Fig. 8.64 could be used. Here a NAND decoder is employed on a bit-by-bit basis to enable complementary transmission gates (single transistors may be used where possible) onto a common pair of data lines. These are then routed to a sense amplifier and write circuitry.

8.3.1.1.5 Sense amplifiers

Many sense amplifiers have been invented to provide faster sensing, smaller layouts, and lower power-dissipation sensing.[33] The simple inverter sense amplifier provides for low power sensing at the expense of speed. The differential sense amplifier can consume a significant amount of DC power (Fig. 8.54). Alternatively, one can employ clocked sense amplifiers similar to the SSDL gate shown in Fig. 5.40.

8.3.1.1.6 RAM timing budget

The critical path in a static RAM read cycle includes the clock to address delay time, the row address driver time, row decode time, bit-line sense time, and the setup time to any data register. The column decode is usually not in

Micron Ex. 1037, p. 620
Micron v. YMTC
IPR2025-00119

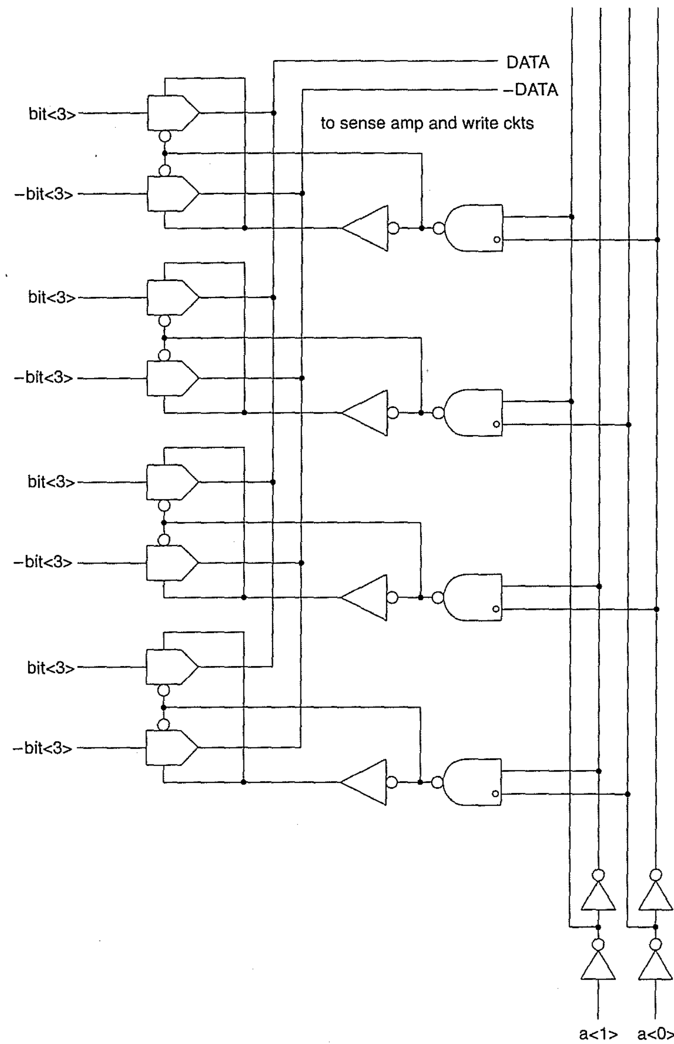


Figure 8.64 Decoded column decoder

the critical path because the decoder is usually smaller and the decoder has the row access time and bit-line sense time to operate. The write operation is usually faster than the read cycle because the bit lines are being actively driven by larger transistors than the memory cell transistors. However, the bit lines may have to be allowed to recover to their quiescent values before any more access cycles take place. In the static load RAM, this speed depends on the size of the static pull-up. Apart from carefully sizing transistors, the RAM speed may be increased by pipelining the row decode signal.

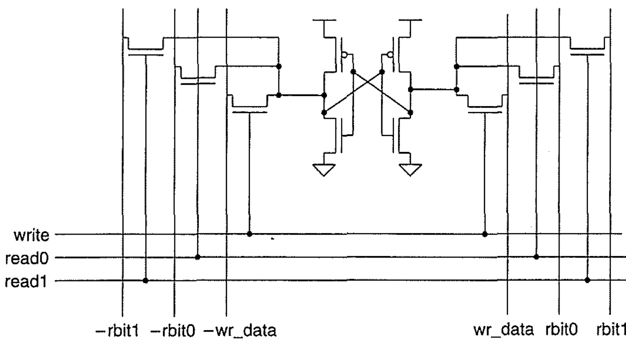
8.3.1.2 Register Files

Register files are generally fast RAMs with multiple read and write ports. Conventional RAM cells may be made multiported by adding pass transis-

Micron Ex. 1037, p. 621
Micron v. YMTC
IPR2025-00119

tors. Such a configuration is shown in Fig. 8.65(a). A single-write-port, double-read-port memory is shown. For a cross-coupled inverter RAM cell, the write lines generally have to be differential. However, the read lines can be single ended. Figure 8.65(b) shows a modified RAM cell with a single write port and two read ports. This general technique has been used on a 17-port register file that had an overall bandwidth of 1.4 Gigabytes/s.[34]

An alternative register file structure that can be easily changed for a wide variety of read and write ports is shown in Fig. 8.66.[35] Figure 8.66(a) shows a single-write-port, double-read-port cell. The write port is a single-ended implementation where the write pass transistor (N_1) is used to overdrive a weak feedback inverter (N_3, P_3). The threshold of storage inverter (N_2, P_2) is biased towards V_{SS} by increasing the size of N_2 with respect to P_2 to aid in the writing of the cell. The storage inverters drive a buffer inverter (N_4, P_4) which in turn drives two read lines through pass transistors (N_5, N_6). Additional read ports are constructed by adding transistors at the output of inverter (N_4, P_4) and adding additional read-row decode lines. Additional write lines are added by adding transistors that drive inverters (N_2, P_2) and (N_3, P_3). A benefit of this design is that no matter what load appears on the output of the buffer inverter, the state of the memory cell can not be flipped.



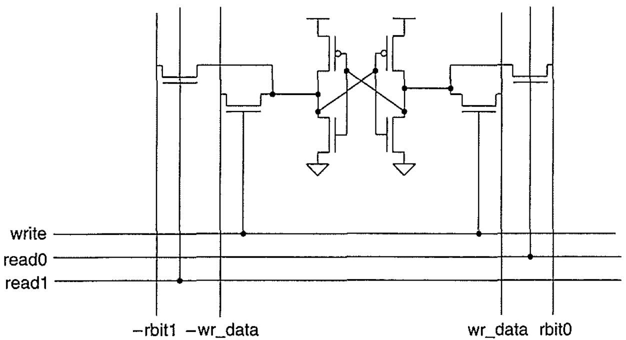
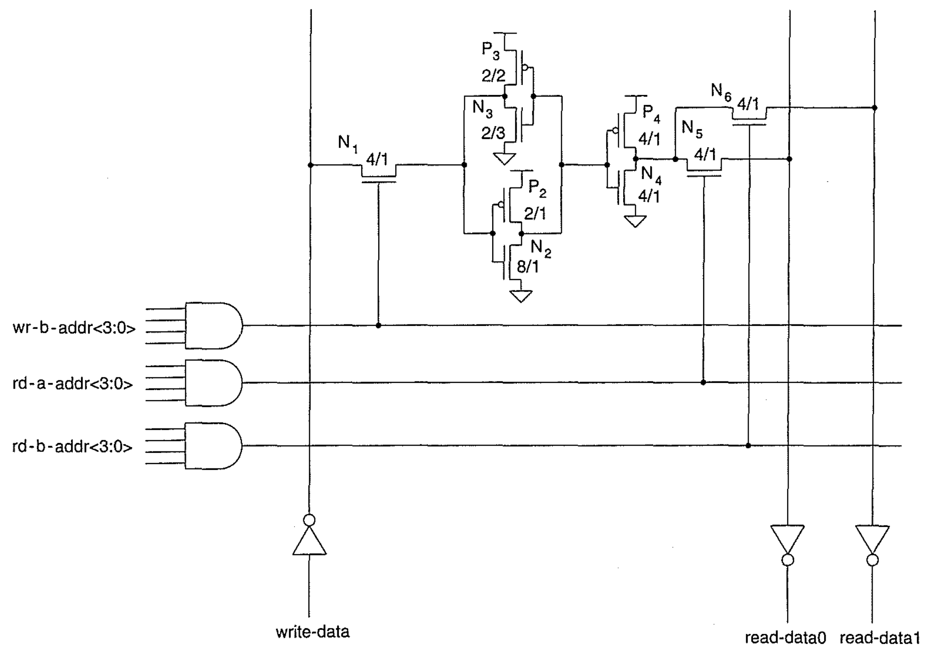


FIGURE 8.65 Multiported (2R-1W) RAM cell: (a) fully differential; (b) single-ended read

Micron Ex. 1037, p. 622
Micron v. YMTC
IPR2025-00119



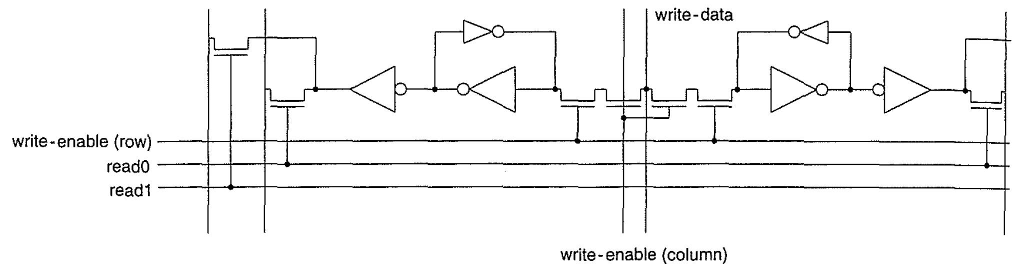


Figure 8.66 Expandable register file cell: (a) row accessed; (b) column accessed and row accessed

The design in Fig. 8.66(a) is used where there is no column multiplexing. The version shown in Fig. 8.66(b) is used where column multiplexing is required. An additional transistor (two in this design for symmetry) is added per column to enable a column for writing.

8.3.1.3 FIFOs, LIFOs, SIPOs

Using the basic RAM memory cell, multiport register cells, or variations of these, a variety of special-purpose memories can be constructed.

A First In First Out (FIFO) memory is useful for buffering data between two asynchronous data streams. Figure 8.67 shows a block diagram that outlines the operation of a FIFO. A stream writes into the FIFO when a *WRITE*

Micron Ex. 1037, p. 623
Micron v. YMTC
IPR2025-00119

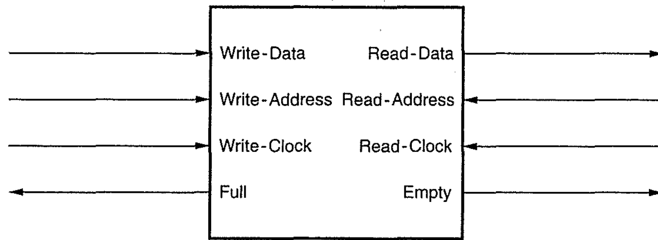


FIGURE 8.67 FIFO interface signals

clock is asserted and observes a *FULL* flag, which is raised when the FIFO can accept no more input data. Another stream reads data when a *READ* clock is asserted until an *EMPTY* flag is asserted. Ideally, the two ports can read and write independently. Due to other system delays and latencies it may be desirable to have *ALMOST-FULL* and *ALMOST-EMPTY* flags so that impending fullness or emptiness can be communicated. The simplest implementation of a FIFO uses a dual port RAM or register file with a read and write counter. An example design of the addressing logic that is useful for synchronous read and write signals is shown in Fig. 8.68. Two counters control the read pointer (*RP*) and the write pointer (*WP*) that are addresses to the dual port memory. A further difference circuit is incremented, in the case of a write, or decremented, in the case of a read. The output of this counter is examined to determine the *EMPTY* and *FULL* flags. In this case a *ZERO* detect determines when the FIFO is *EMPTY.* Alternative implementations of FIFOs may use distributed forms of row decoders, where full and empty bits are propagated by serial shift registers in the word direction of the memory.

A Last In First Out (LIFO) memory, or push-down stack, is of use in such applications as subroutine stacks in microcontrollers. In common with FIFOs, regular RAMs or register files may be used or special distributed row decoders may be designed as the address pointer moves sequentially from row to row. The former usually are more straightforward to design, while the latter may save some space. (See also Section 9.2.4.3.)

A Serial In Parallel Out (SIPO) memory is of use to convert serial data to a parallel form. These memories are often of use in signal-processing applications. An example of the memory cell used in this type of memory is shown in Fig. 8.69. Data is shifted in at a high rate via the complementary clocks *clk* and *–clk,* which should be nonoverlapping to prevent data feedthrough. Data may be read in parallel through access transistor N_1 with an appropriately timed clock pulse (i.e., when the Q data is valid).

8.3.1.4 Serial-Access Memory

Serial-access memories (shift registers) are also of use in signal-processing applications for storage and delaying signals. A serial-access memory may

Micron Ex. 1037, p. 624
Micron v. YMTC
IPR2025-00119

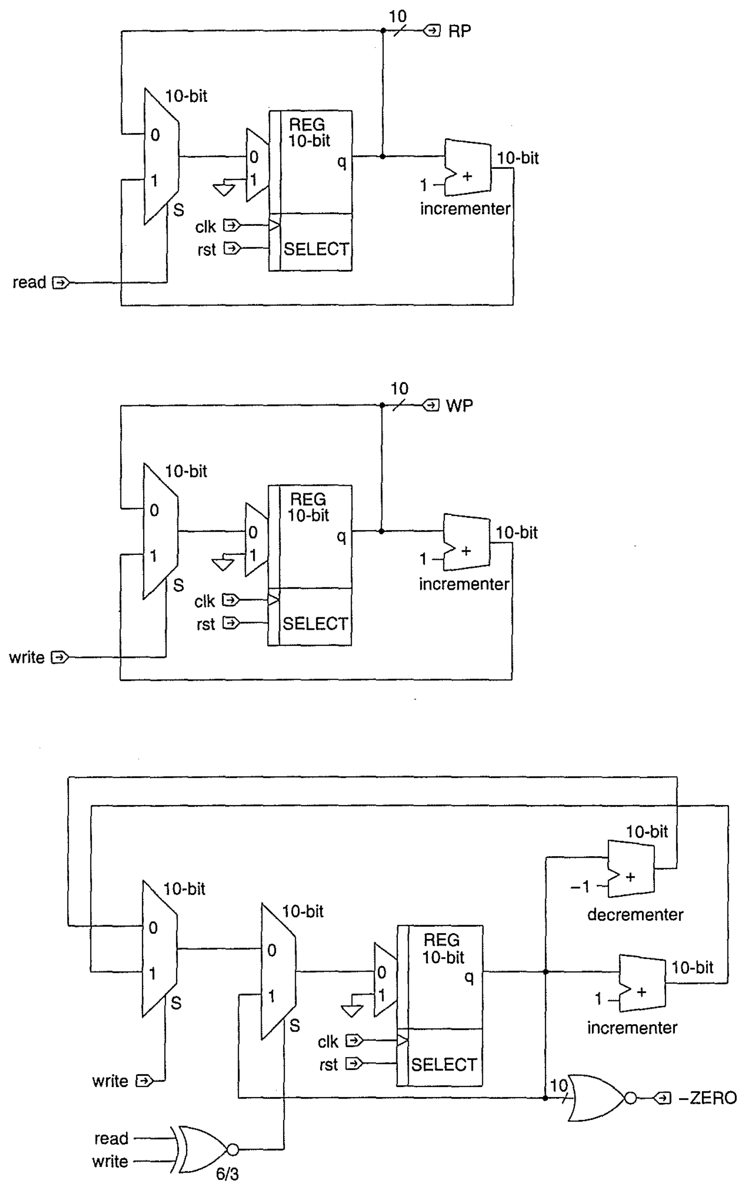


Figure 8.68 FIFO address control design

be simulated by a RAM, and probably for most applications this provides the smallest implementation because the CMOS static RAM cell is a very area-efficient structure. However, the RAM is surrounded by peripheral circuits, such as row and column decoders and sense amplifiers, and in the case of a serial-access memory, a counter. In some circumstances, a dedicated shift-register memory may be appropriate from a density, speed, or floorplanning viewpoint. (See Chapter 9.)

Micron Ex. 1037, p. 625
Micron v. YMTC
IPR2025-00119

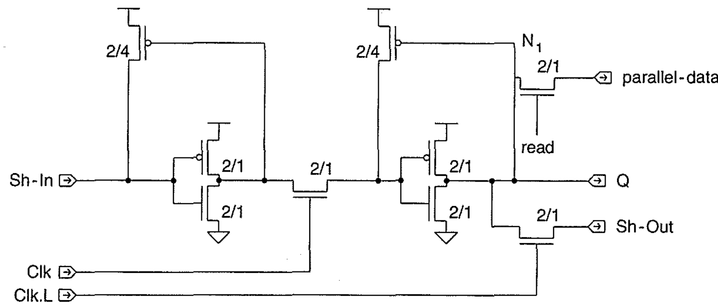
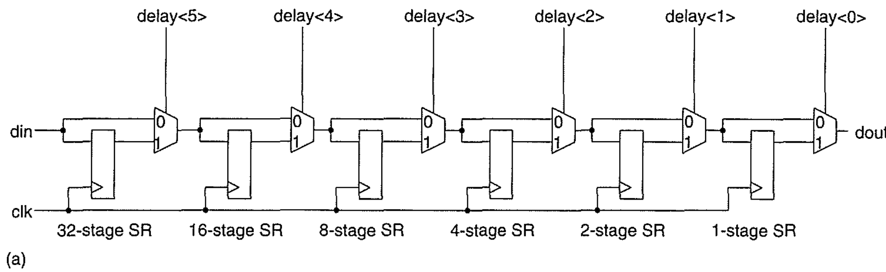
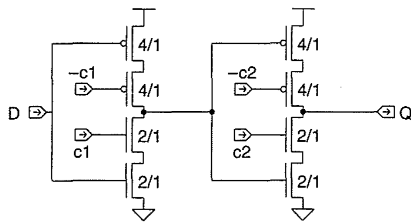


FIGURE 8.69 SIPO cell design





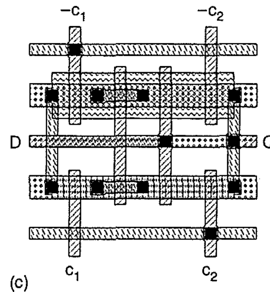


FIGURE 8.70 Tapped delay line: (a) architecture; (b) circuit; (c) symbolic layout

Figure 8.70(a) shows an example of a 64-byte tapped delay line that might be used in a video processing system. Blocks of byte-wide shift registers are delayed by 32, 16, 8, 4, 2, and 1 clock cycles, and multiplexers control the pass-around of the delay blocks to yield the appropriate delay amount. Each memory cell is a shift register, as shown in Fig. 8.70(b). A typical layout for the shift register cell is shown in Fig. 8.70(c). Here the 2-phase clocks are run horizontally between bits of the shift register. The horizontal metal2 power busses are run over the transistors.

8.3.2 Read Only Memory

Read Only Memory cells may be implemented with only one transistor per bit of storage. A ROM is a static memory structure in that the state is retained

Micron Ex. 1037, p. 626
Micron v. YMTC
IPR2025-00119

indefinitely—even without power. A ROM array is usually implemented as a NOR array, as shown in Fig. 8.71. Note that a NAND array may be used if ultra-small ROMs[36,37] are required, but as discussed in Chapter 5, these implementations will be quite slow.

The electrical details of the NOR structure may embody any of the NOR gate structures studied so far, including the pseudo-nMOS NOR and the domino NOR gate. One problem with the domino gate is that the pull-down path passes through two transistors, one the programmed transistor and the other the virtual ground pull-down. This can slow the bit-line transition for large ROMs. A dynamic CMOS alternative to the domino NOR is shown in Fig. 8.72. Here the word lines are forced low while the bit lines are being precharged. This ensures that DC current does not flow. After the bit-line pull-ups have been turned off, the word-line drivers are asserted and one word line is active. The timing chain to ensure this sequence of events has to be carefully designed and simulated. Where DC power dissipation is acceptable and the speed is sufficient, the pseudo-nMOS ROM is the easiest to design, requiring no timing. The DC power dissipation may be significantly reduced by turning the pull-ups on according to the column address decoding. Figure 8.73 shows an example of this where only one bit line in four is being pulled up at any one time. Row decoders for ROMs are similar to those for RAMs except that they are usually very constrained by the ROM bit pitch. This usually means that some form of a predecode structure is required. Column decoders for ROMs are usually simpler than those for

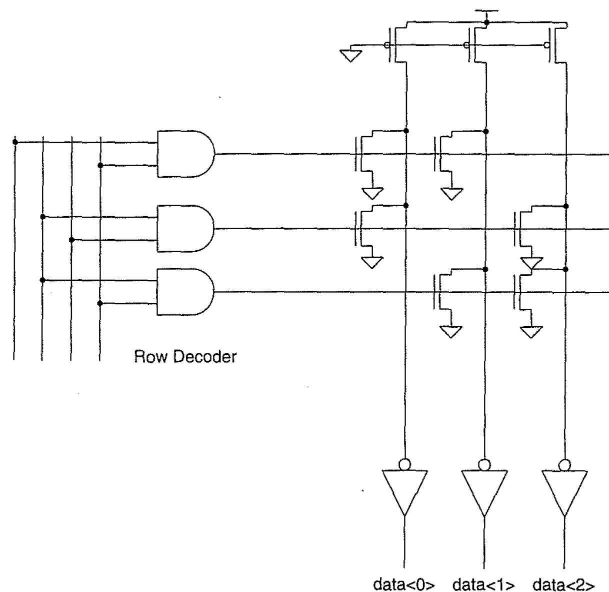


Figure 8.71 Basic ROM architecture

Micron Ex. 1037, p. 627
Micron v. YMTC
IPR2025-00119

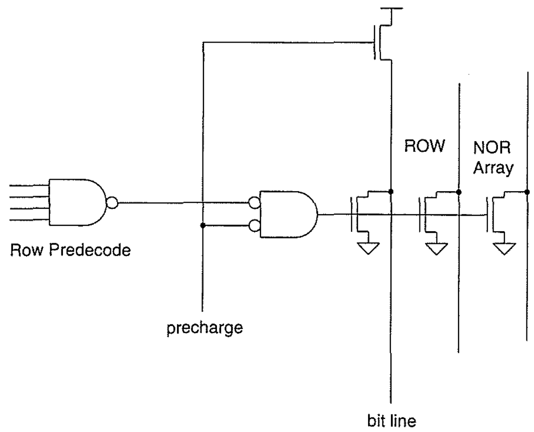


Figure 8.72 Dynamic ROM circuitry

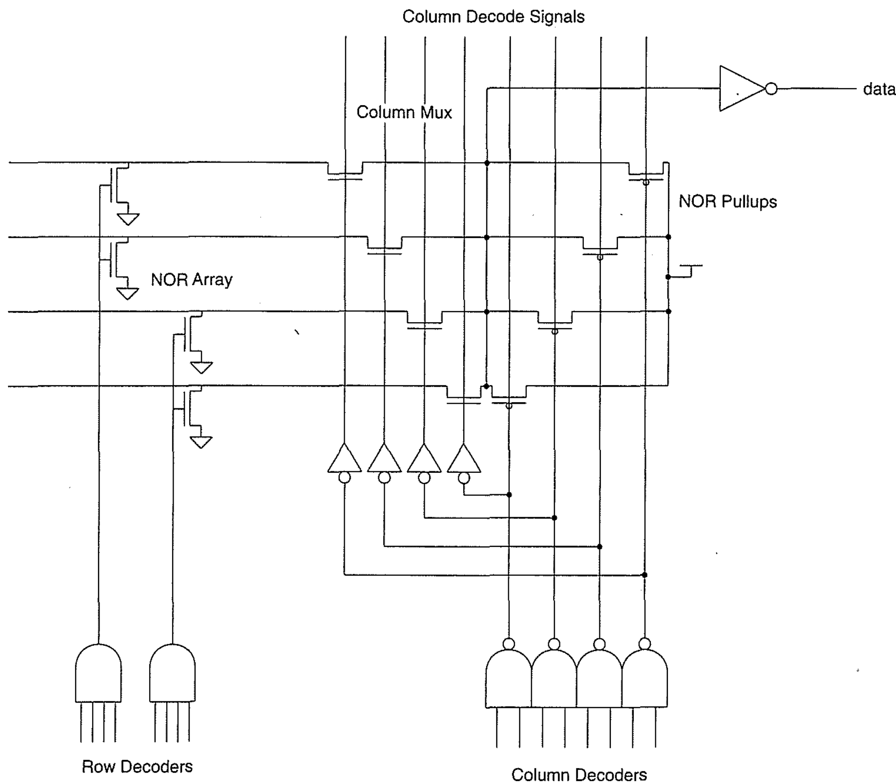


FIGURE 8.73 A power-saving ROM circuit

RAMs because only read operations are employed and single-ended sensing is usually employed.

Mask programmability may be achieved via contact programming, presence or absence of a transistor, or an implant to turn a transistor permanently

Micron Ex. 1037, p. 628
Micron v. YMTC
IPR2025-00119

off or on. Other technology options may be possible, such as electrically erasable random access memories.

Several symbolic layouts for ROM cells are shown in Fig. 8.74, along with a programming techinque. Running word lines in polysilicon is only appropriate for slow speed ROMS or (perhaps) silicided poly. In a microcode ROM in a microprocessor, transistor programming would be preferable, because this would minimize the dynamic power dissipation (less capacitance on word lines). It can also affect speed if the load on word lines can be balanced in a sparse ROM. In a generic circuit that is mask-programmable, metal programming may be desirable. Strapping the poly with metal2 every 4 to 8 ROM sites is appropriate for higher speed ROMs (Fig. 8.74c).

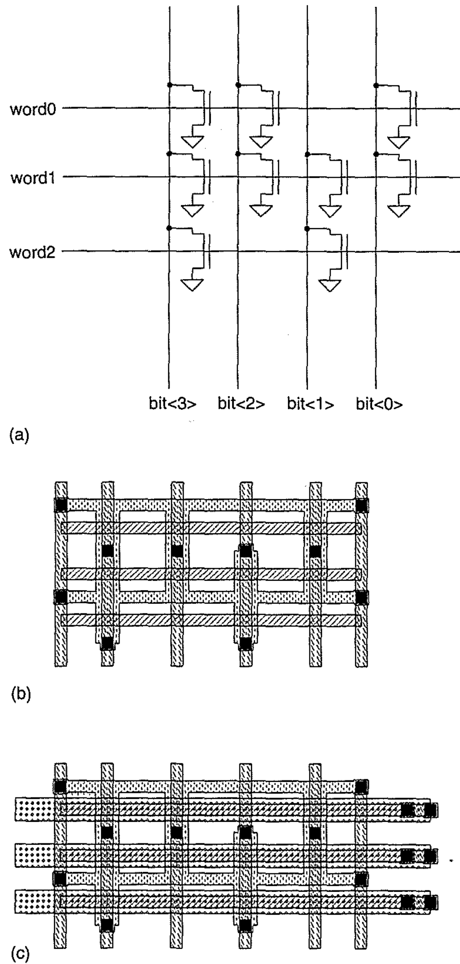


Figure 8.74 ROM layouts: (a) circuit; (b) poly word lines; (c) poly and metal2 strapped word lines

Micron Ex. 1037, p. 629
Micron v. YMTC
IPR2025-00119

8.3.3 Content-Addressable Memory

A content addressable memory[38,39] is shown in Fig. 8.75(a). The CAM portion examines a data word and compares this data with internally stored data. If any data word internally matches the input data word, the CAM signals that there is a match. These match signals can be passed as word lines to a RAM to enable a specific data word to be output (Fig. 8.75b). This structure may be used as a translation look-aside buffer in the virtual memory lookup in a microprocessor.

A typical CMOS CAM memory cell is shown in Fig. 8.76(a). It consists of a normal static RAM cell with additional transistors N_1 and N_2, which form an XOR gate, and N_3, which is a distributed NOR pull-down. The memory cell may be written and read in the conventional manner. Writes are used to store the match data in the cells, whereas reads are used for testing purposes. A MATCH operation proceeds by placing the data to be matched on the bit lines but not asserting the word line. A 1 appears on the gate of N_3 if the data in the cell is not equal to the data on the bit lines. The drains of N_3 transistors of cells in the same row are commoned, as shown in Fig. 8.76(b). These form a distributed NOR gate, which may be dynamic (with appropriate timing) or pseudo-nMOS (if speed is not critical). Each match line (*match*<3:0>) remains high if the data in the row matches the data placed on the bit lines. These lines may be used to assert the word lines on a RAM.

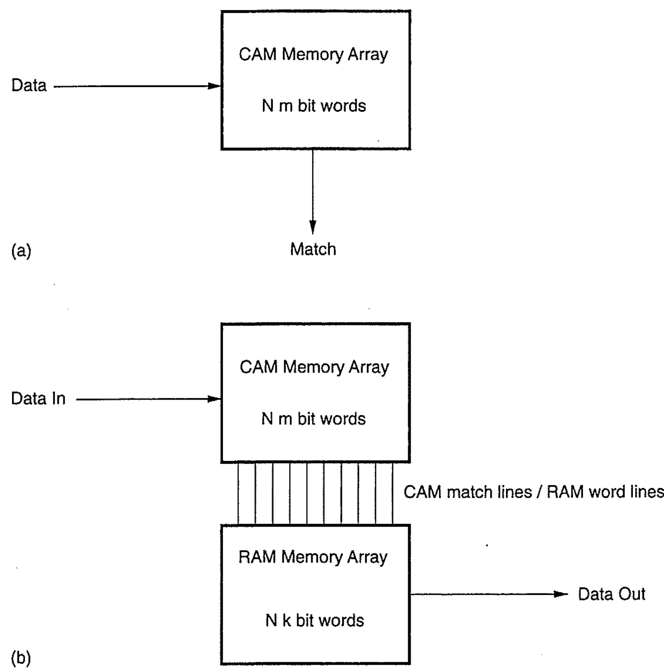
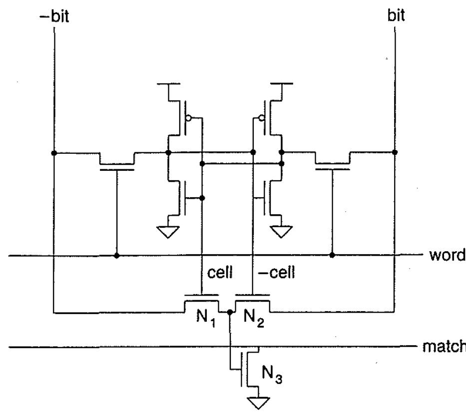


FIGURE 8.75 CAM architecture: (a) basic CAM; (b) typical application as translation lookaside buffer

Micron Ex. 1037, p. 630
Micron v. YMTC
IPR2025-00119



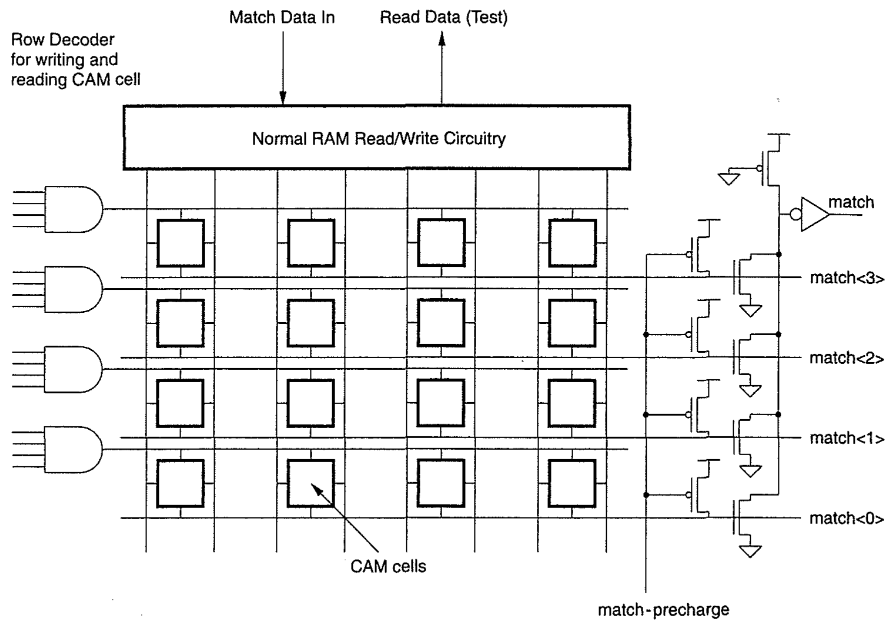


Figure 8.76 CAM: (a) cell; (b) array circuit

Another NOR gate, which looks at all the match lines, yields an overall match signal.

8.4 Control

While arithmetic and memory structures benefit from regularity, control structures usually do not. They perennially form the really hard part of a design—the part that takes the longest time to design, verify, and test. Usu-

Micron Ex. 1037, p. 631
Micron v. YMTC
IPR2025-00119

ally the control portion of a design is also the last to solidify in the design cycle. Thus it is prudent to look for methods of designing control structures that are highly automated and therefore quick to design. This section begins with a discussion of finite-state machines (FSMs) and then examines various methods for implementing these and other control structures.

8.4.1 Finite-State Machines

A finite-state machine (Fig. 5.43a) provides an organized structure for capturing control sequencing and operation. Diagrammatically, a state machine may be represented by a state-transition diagram (or graph) in which the labelled nodes of the graph represent states and the labelled directed arcs represent transitions between states. A state-transition diagram can be constructed in which the nodes are drawn as circles and the transitions are drawn as lines with arrows. Two basic types of state machines can be designed. A Mealy state machine uses logic to determine the outputs from the inputs and the current state, stored in state registers. A Moore machine determines the outputs from the current state alone. Figure 8.77 shows the two types of machine.

8.4.1.1 FSM Design Procedure

While the design of complex state machines is liable to be machine assisted, small state machines may be designed by hand. As an example, a state machine that might control a toll-booth on a highway will be used. In the *idle* state, the tollbooth, with its gate lowered and its green "proceed" light off, awaits a car. When a car enters the tollbooth, a pressure sensor detects the car and passes a signal to the controller. The controller then awaits the *correct toll* signal and on receiving this, raises the gate and turns the green light on. When the car exits the tollbooth, the controller reenters the *idle* state (green light off and gate down).

The following steps are illustrative of the design of a state machine to perform this function.

1. Draw the state-transition diagram

First the state machine is captured in a state-transition diagram. The inputs to the controller are three signals: a *RESET* (*R* for short) signal, a *CAR-IN-BOOTH* (*A* for short) signal (indicating the car is in the tollbooth), and the *CHANGE-OK* (*C* for short) signal indicating the correct toll has been tendered. There is one output from the controller, the *GREEN-LIGHT* signal used to raise the gate and turn the green light on. The example above may be represented by three states: the *IDLE* state, the *WAIT-FOR-COIN* state and the *WAIT-FOR-CAR-TO-EXIT* state. These are represented as circles in Fig. 8.78(a). Arcs are drawn between states to represent the state transitions. For example, when the controller is in the *IDLE* state it

Micron Ex. 1037, p. 632
Micron v. YMTC
IPR2025-00119

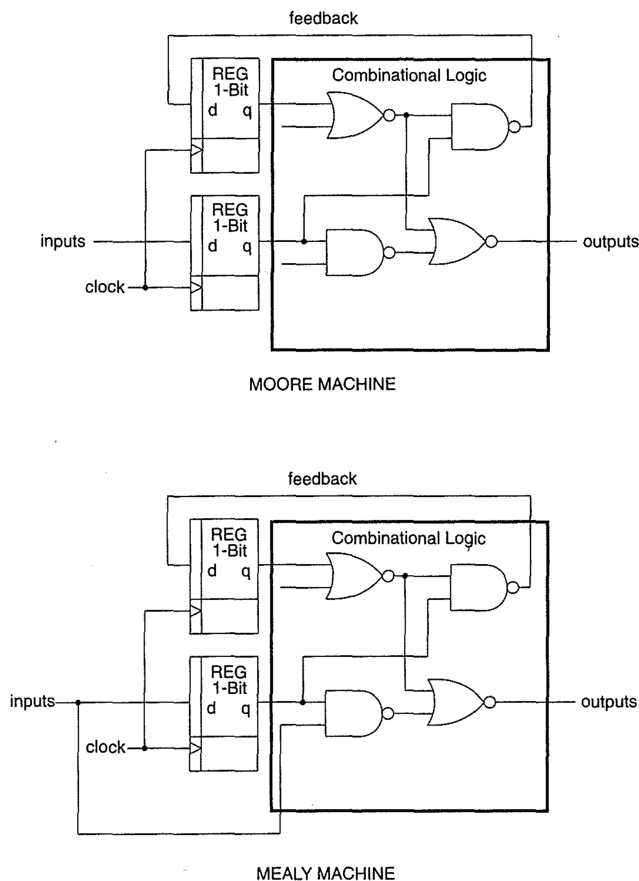


Figure 8.77 Mealy and Moore state machines

transitions to the *WAIT-FOR-COIN* state when there is a car in the booth (i.e., the input *CAR-IN-BOOTH* or *A*, is true). When *CAR-IN-BOOTH* is false, the *IDLE* state loops to itself (–*A*). Additionally, the *RESET* signal causes the *IDLE* state to be entered.

2. Check the state diagram

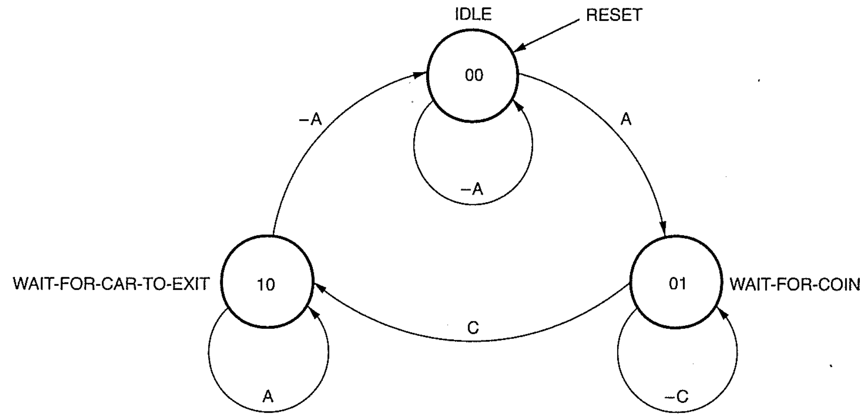
There are some simple checks that may be made on the state-transition diagram. These are as follows:

A. Ensure that all states are represented, including the *IDLE* state.

B. Check that the OR of all transitions leaving a state is TRUE. This is a simple method of determining that there is a way out of a state once entered.

C. Verify that the pairwise XOR of all exit transitions is TRUE. This ensures that there are not conflicting conditions that would lead to more than one exit-transition becoming active at any one time.

Micron Ex. 1037, p. 633
Micron v. YMTC
IPR2025-00119



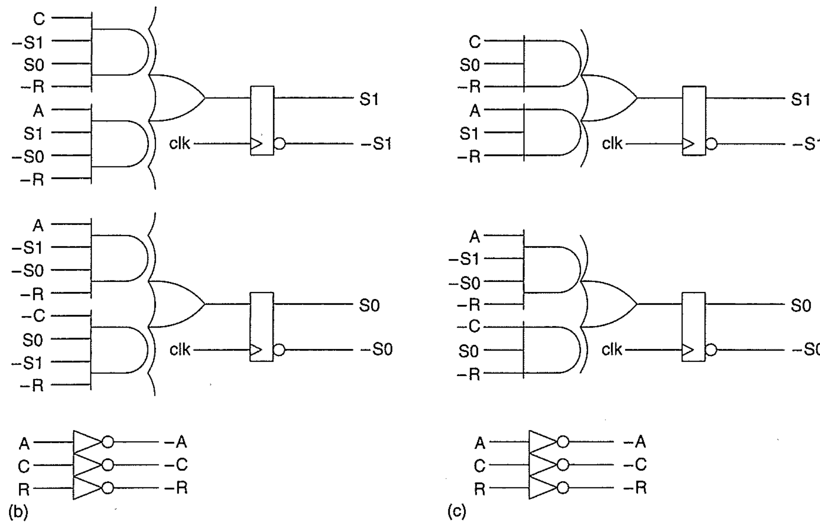


FIGURE 8.78 State-machine transition diagram

D. Insert loops into any state if it is not guaranteed to otherwise change on each cycle. In other words, if a machine enters a state and stays there until some condition occurs, insert the appropriate transition, which is a loop to the state itself. For instance, in the *WAIT-FOR-COIN* state, if the toll is not correct, the controller stays in the *WAIT-FOR-COIN* state.

3. Write the state equations

For each transition, the state equation may be represented as follows:

if(state == oldstate & condition) next-state = newstate

Micron Ex. 1037, p. 634
Micron v. YMTC
IPR2025-00119

For instance, for the state transition diagram shown in Fig. 8.78, the state equations may be written as

```
next-state IDLE when state==IDLE & !CAR-IN-BOOTH
            OR state==WAIT-FOR-CAR-TO-EXIT & !CAR-IN-BOOTH
next-state WAIT-FOR-COIN when state==IDLE & CAR-IN-BOOTH
            OR state==WAIT-FOR-COIN & !CHANGE-OK
next-state WAIT-FOR-CAR-TO-EXIT
            when state==WAIT-FOR-COIN & CHANGE-OK
            OR state==WAIT-FOR-CAR-TO-EXIT & CAR-IN-BOOTH
```

The first state equation states that the next state is *IDLE* if any of the following conditions are true:

The current state is *IDLE* and the input *CAR-IN-BOOTH* is false.

The current state is *WAIT-FOR-CAR-TO-EXIT* and the input *CAR-IN-BOOTH* is false.

4. Assign the states

The minimum number of state bits that can represent a state machine is $\log_2 K$ where K is the number of states. To ease the problem of state assignment, more states than the minimum may be used. In this example, there are three states, so two state bits (S_1, S_0) are required. Alternatively, unary state assignment would require three bits. The following design guidelines are useful:

A. Assign the *ZERO* state (S_1,S_0=0,0) to the most complex state (state *IDLE*).

B. Assign the adjacent states in a Gray code manner (such that they differ by one bit).

C. Otherwise assign states to minimize logic.

Thus for this example an assignment would be:

```
                IDLE = 0 0
       WAIT-FOR-COIN = 0 1
WAIT-FOR-CAR-TO-EXIT = 1 0
```

The state assignments are labelled at the center of each state (the bolded circles) in Fig. 8.78. At this point a truth table may be constructed that describes the state machine (see Table 8.8).

Each line of Table 8.8 describes an arc on the state-transition diagram. For instance, the first row shows that the *IDLE* state is entered (00) when *RESET* is set to 1. The second line shows that when in state *IDLE*, the next state is *IDLE* if there is no car in the tollbooth. The *CHANGE-OK* is a don't-

Micron Ex. 1037, p. 635
Micron v. YMTC
IPR2025-00119

TABLE 8.8 Tollbooth-state table

		current state			next state	
reset	car-in-booth	change-ok	state<1>	state<0>	state<1>	state<0>
1	x	x	x	x	0	0
0	0	x	0	0	0	0
0	1	x	0	0	0	1
0	x	0	0	1	0	1
0	x	1	0	1	1	0
0	1	x	1	0	1	0
0	0	x	1	0	0	0

care in this case. Examining the state bits one by one, logic equations may be written for each bit. For instance,

```
next-state<0> =  !reset & car-in-booth & !state<0> & !state<1>
                 + !reset & !change-ok & state<0> & !state<1>
next-state<1> =  !reset & !change-ok & state<0> & !state<1>
                 + !reset & car-in-booth & !state<0> & !state<1>
```

The logic for any outputs must also be generated. In this example, the output *GREEN-LIGHT* is simply *state*<1>. Notice that the state assignment may have been done in a way that would have necessitated some logic to decode this signal (i.e., state *WAIT-FOR-CAR-TO-EXIT* = 11).

5. Construct the resulting logic and registers

The resulting logic and registers are shown in Fig. 8.78(b). This may be simplified to yield the design in Fig. 8.78(c).

8.4.2 Control Logic Implementation

Control logic in CMOS is constructed in two main ways, with two-level sum-of-products logic and with multilevel logic. Two-level sum-of-products representations have a straightforward geometric implementation in the form of a Programmable Logic Array (PLA). Both two-level and multilevel-logic may be implemented in terms of CMOS logic gates (either static or dynamic).

8.4.2.1 PLA Control Implementation

A programmable logic array (PLA) is a structure that provides a regular structure for implementing combinatorial and sequential logic functions. A

PLA may be used to take inputs and perform some combinatorial function of these inputs to yield outputs, or additionally some of the outputs may be fed back to the inputs via registers, thus forming a finite-state machine. Two-level logic minimization is a well-understood problem. The program Espresso[40] is representative of programs that minimize sum-of-products forms.

A typical PLA uses a two-level sum-of-products AND-OR structure similar to that shown in Fig. 8.79. This implementation also shows clocks to latch inputs and outputs. The basis of a PLA is a sum-of-products form of representation of binary expressions. For example, consider the following expressions that have to be evaluated:

$$z_0 = x_0$$
$$z_1 = x_1 + (-x_0 \,.\, -x_1 \,.\, -x_2)$$
$$z_2 = -x_1 \,.\, -x_2$$
$$z_3 = (-x_0 \,.\, -x_1 \,.\, x_2) + (-x_0 \,.\, x_1 \,.\, -x_2)$$

where z_0, z_1, z_2, and z_3 are the four output terms (or sums) and x_0, x_1, and x_2 are the input variables. There are five product terms, namely, x_0, x_1, $-x_0 \,.\, -x_1 \,.\, x_2$, $-x_1 \,.\, x_2$, and $-x_0 \,.\, x_1 \,.\, -x_2$. Thus these terms would be formed in the AND array of the PLA, as shown in Fig. 8.80. The four outputs are formed by ORing the appropriate product terms. Normally, high-speed PLAs are implemented as two NOR arrays, as shown in Fig. 8.79 (although NAND arrays may be used for slow applications). By using inverting inputs and outputs, the AND-OR structure is maintained.

The electrical design of a CMOS PLA depends on the generic style of PLA. A straightforward physical implementation for a PLA is represented

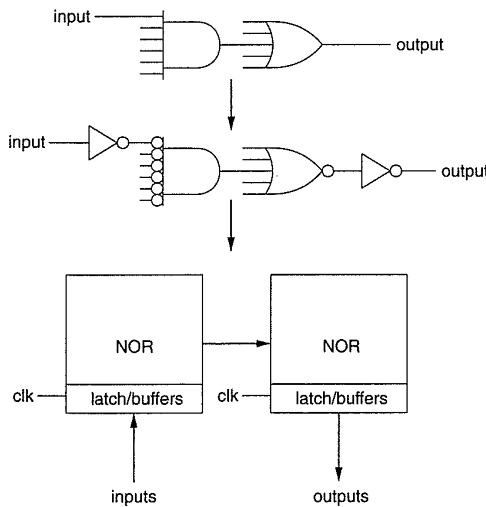


Figure 8.79 PLA Architecture

Micron Ex. 1037, p. 637
Micron v. YMTC
IPR2025-00119

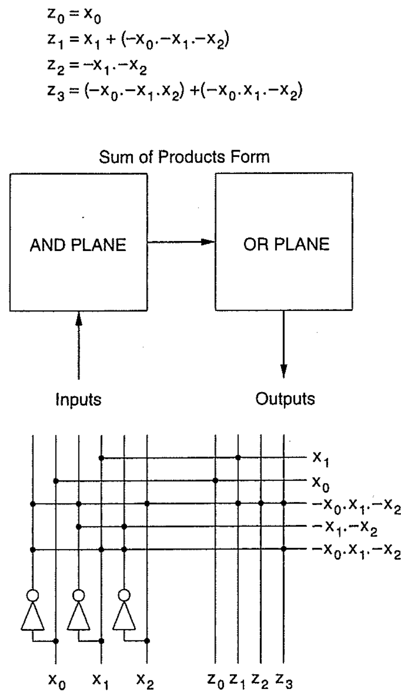


FIGURE 8.80 PLA example

by Fig. 8.80. Variations of this involve multiple-sided access (Fig. 8.81) and various folded structures.

A generic floorplan for a "simple" PLA is shown in Fig. 8.82. This has been designed as a set of tiles, designated by letters. In the treatment of various circuit options this naming convention will be used to designate particular cells. Brief descriptions of the cells are as follows:

AN AND-plane programming cell
OR OR-plane programming cell

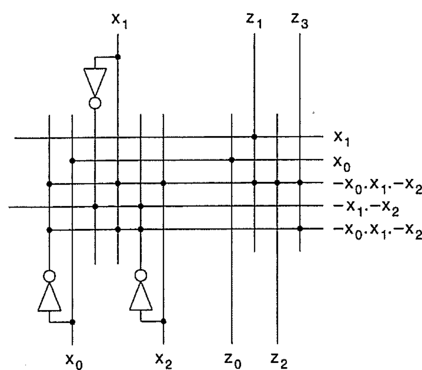


FIGURE 8.81 Multisided PLA access

Micron Ex. 1037, p. 638
Micron v. YMTC
IPR2025-00119

TL	TI	TI	TM	TO	TO	TR
LA	AN	AN	AO	OR	OR	RO
LA	AN	AN	AO	OR	OR	RO
BL	BI	BI	BM	BO	BO	BR

FIGURE 8.82 Generic PLA floorplan

AO AND-OR communication cell
TI Top AND-plane input cell
BI Bottom-AND plane input cell
TO Top OR-plane output cell
BO Bottom OR-plane output cell
LA Left AND-plane cell
RO Right OR-plane cell
BL Bottom-left cell
BM Bottom-middle cell
BR Bottom-right cell
TL Top-left cell
TA Top AND cell
TM Top-middle cell
TO Top OR cell
TR Top-right cell

The most straightforward PLA design uses a pseudo-nMOS NOR gate. Figure 8.83 shows the circuit diagram with the key cell positions identified. Cell AO can either be a layer-change cell or can be used to buffer the AND array outputs. Figure 8.84 shows a PLA for the tollbooth example. Design of

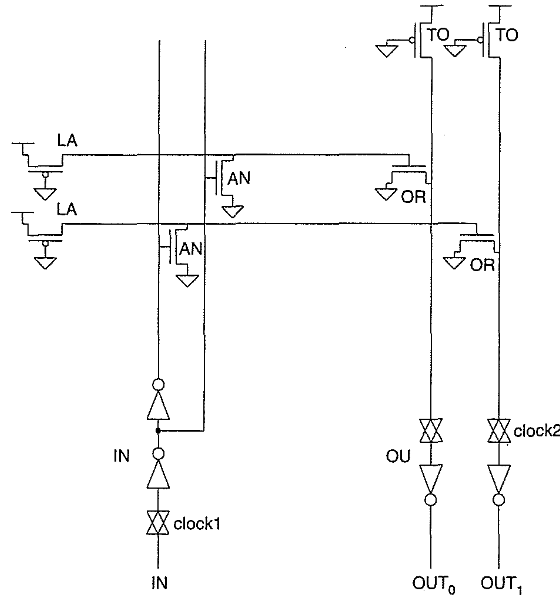


Figure 8.83 Pseudo-nMOS PLA

Micron Ex. 1037, p. 639
Micron v. YMTC
IPR2025-00119

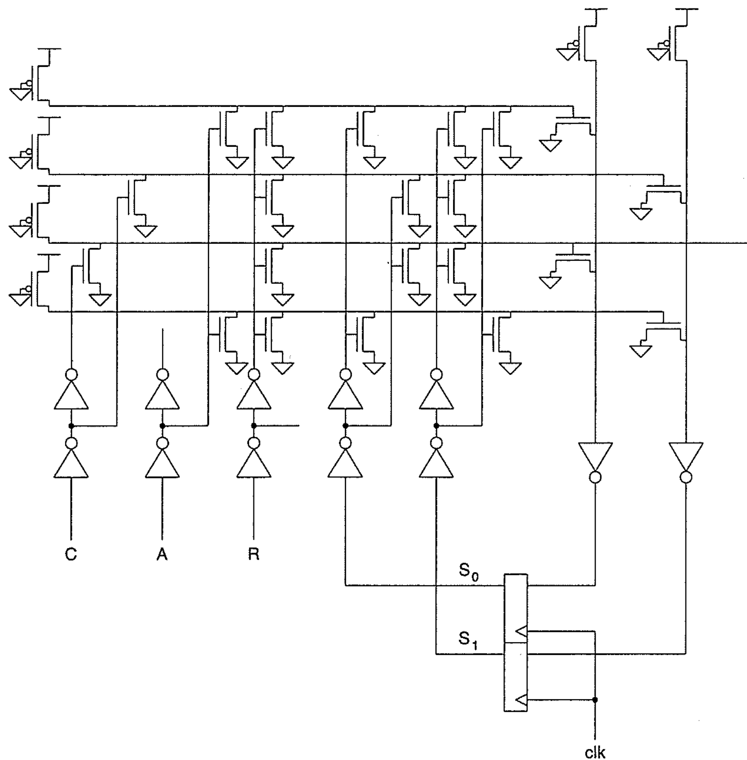
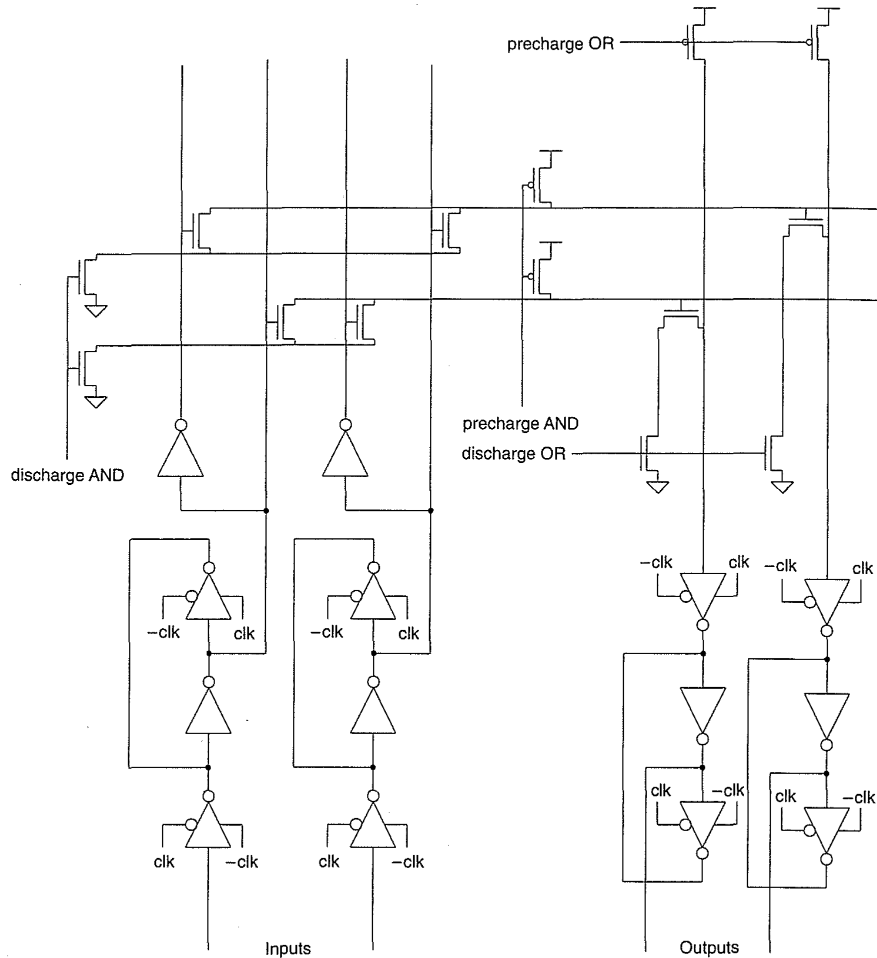


FIGURE 8.84 PLA for toll-booth example

the pseudo-nMOS NOR gates would follow the guidelines given in previous chapters. Advantages of this PLA include simplicity and small size. Disadvantages occur due to the static power dissipation of the NOR gates and possible speed problems (the pull-ups may become slow on large terms). Any convenient register may be used; a static register is shown. This PLA could be fairly independent of the overall system-clocking strategy. Cells TL, TA, BL, BM, TM, TR, RO, and BR are used to route power and clocks as necessary.

By using dynamic CMOS, the circuit shown in Fig. 8.85(a) may be used. Both the AND plane and OR plane have to be supplied with clocks similar to those shown at the bottom of the diagram (Fig. 8.85b). On the rising edge of the clock the input latches store the input data. Following this the AND and OR planes are precharged and then evaluated. When the AND plane outputs are valid, the OR plane may be evaluated. The waveforms in Fig. 8.85(b) may be generated from a multiphase clock or, more probably, in a single-phase clocking scheme by self-timed circuits. Figure 8.86 shows some possible circuits for self timing the PLA operation. The AND precharge may be timed of the rising edge of the clock and the worst case time it takes an AND line to pullup. This may be accomplished by using the circuit shown in Fig. 8.86(a), which uses a dummy AND row. This row has every AND programming transistor inserted (*NPD*) to ensure the load capac-

Micron Ex. 1037, p. 640
Micron v. YMTC
IPR2025-00119



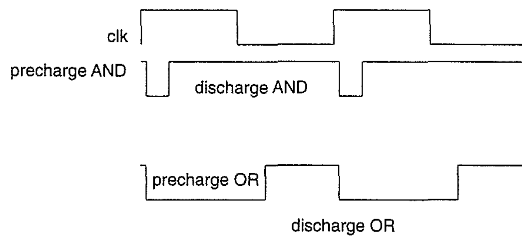


Figure 8.85 Dynamic PLA

itance (C_{load}) is a maximum. In addition, the p pull-up (PU) is made smaller than the normal p pull-ups to give some timing margin. An inverter and a NAND gate complete the timing circuit. The OR precharge clock may be self-timed using the circuit shown in Fig. 8.86(b). Here a dummy AND row

Micron Ex. 1037, p. 641
Micron v. YMTC
IPR2025-00119

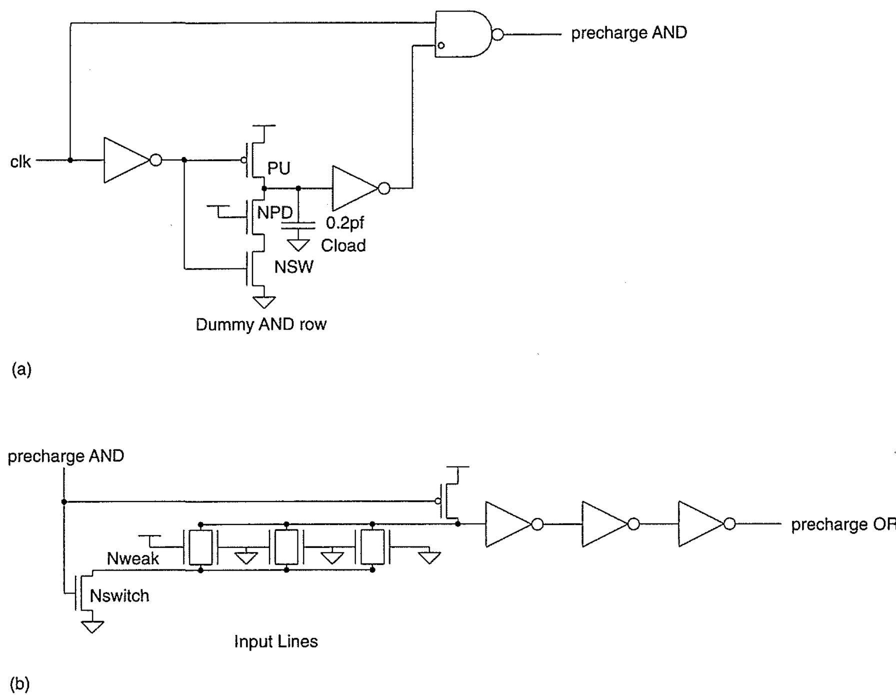


FIGURE 8.86 Self-timed PLA circuits: (a) AND precharge; (b) OR precharge

is used to determine the worst-case fall time of an AND row. The row is fully populated, with transistors turned off. One smaller-than-normal n pull-down (N_{weak}) in conjunction with a smaller-than-usual ground switch (N_{switch}) is used to pull down the heavily loaded AND line. This is delayed by a few inverters and fed to the OR-place precharge/discharge.

A single-clock PLA that combines a pseudo-nMOS AND plane and a dynamic OR plane is shown in Fig. 8.87.[41] When the clock is high, the OR plane is precharged while all the product terms are forced low by the clocked n-transistor in the AND plane. When the clock transitions low, the product terms conditionally evaluate and then the precharged OR-plane outputs evaluate. The AND plane–transistor ratios are designed according to normal pseudo-nMOS techniques. The inputs must be held constant during the period when the clock is low. This PLA cuts down on the DC dissipation of a fully pseudo-nMOS PLA, while requiring only one clock.

In general PLAs have not found as much acceptance in CMOS as in nMOS technologies. This is due to a number of reasons, some due to CMOS technology and some to the passage of time:

- PLAs have a fixed floorplan and fairly fixed I/O, so extra routing often overshadows any area benefit.
- Dynamic PLAs are cumbersome to design in CMOS, whereas pseudo-nMOS PLAs dissipate DC power.

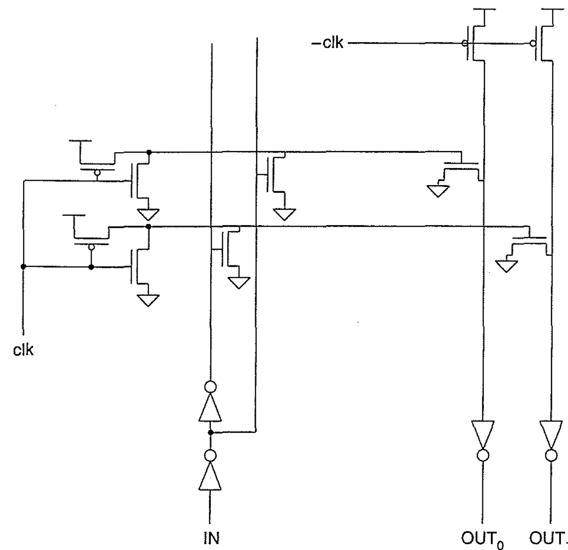


Figure 8.87 Hybrid dynamic pseudo-nMOS PLA

- PLAs are not very compatible with Gate Array technologies, which are popular for ASICs.
- Automatic multilevel logic synthesis has improved dramatically.
- Processes are smaller, hence logic gates are cheaper.
- Large PLAs can be slow.

However, PLAs are compact in themselves and provide a very straightforward way to automate the generation of control logic.

8.4.2.2 ROM Control Implementation

Frequently, control structures may be implemented as a sequenced ROM. A ROM is a special case of a PLA where the AND plane is fully populated. Figure 8.88 shows a simple example of a ROM controller that has a condition-code input and a jump capability implemented by a mux. The ROM has five fields: a next-address field, which provides the next address to the ROM if a branch is not taken; a jump-address field, which is the address taken if the condition code is true; a condition-code select field, which selects which of a number of external conditions to select and the polarity; and an output field, which provides control outputs. Programming consists of writing a microprogram that controls the values of different fields of the ROM. For instance, for the tollbooth example the symbolic microcode might be given as in Table 8.9.

Micron Ex. 1037, p. 643
Micron v. YMTC
IPR2025-00119

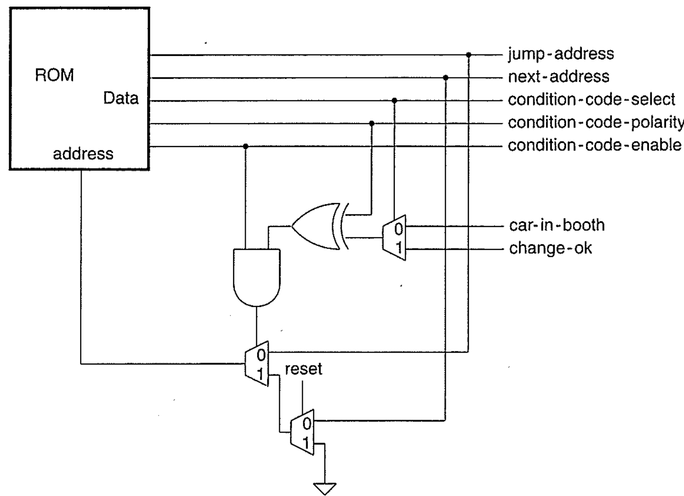


FIGURE 8.88 ROM microcontroller

Here an instruction field has been constructed from the condition-code field as follows:

nop = select true input 00
car-in-booth = select car-in-booth input 01
change-ok = select change-ok input 10

The polarity of the condition controls the XOR gate.

If a metal-programmed ROM is constructed, then a readily changed microsequencer may be constructed. With the addition of a simple datapath, the microsequencer can be extended into a more general microcontroller, which can be used in many low-speed control applications. Megacell libraries frequently contain core microcontrollers that implement standard instruction sets that are supported by a wealth of software.

TABLE 8.9 Symbolic Microcode for the Tollbooth Example

ADDRESS	LABEL	INSTRUCTION	JUMP-ADDRESS	OUTPUT
0	idle:	nop		
1		!car-in-booth	jmp idle	
2	cib:	change-ok	jmp exit	
3		nop	jmp cib	
4	exit:	!car-in-booth	jmp idle	green
5		car-in-booth	jmp exit	green

Micron Ex. 1037, p. 644
Micron v. YMTC
IPR2025-00119

8.4.2.3 Multilevel Logic

The most commonly used method for implementing control logic in CMOS is to use multilevel logic, that is, cascaded groups of regular gates such as INVERTERS, BUFFERS, NANDs, NORs, XORS, and AOIs. There are many CAD systems available today that will automatically minimize the logic for a set of Boolean equations or other algorithmic description. Furthermore, some design systems can do state assignment and can synthesize state machines from a high-level description. After a set of gates have been generated, automatic layout programs can produce a layout in gate-array or standard-cell technology. Even in full-custom chips, this is now a preferred method of generating control logic for the following reasons:

- Standard-cell logic is fluid in shape and can be "reflowed" into gaps that occur in chip layouts due to fixed blocks like memories and datapaths.
- The designer (or synthesis program) has a large amount of control over speed through basically fast gates, gate sizing, and the ability to trade area for speed.
- The automatic logic-gate layout generation is a mature technology—in double-level metal it might be half as dense as a customized layout, but in triple-level metal the density difference is even less.

Most standard-cell control-logic layout is composed of rows of predefined logic and storage cells separated by routing. Programs have been written to automate the generation of control logic from the transistor level. Many of these use the gate-matrix layout style (see Chapter 6). While for small sections of control logic this technique works, there are a number of problems for large sections of control logic. In particular, the layouts get sparse, and internal gate connections completed by long horizontal metal lines tend to produce low-performance gates. Other techniques for custom generating the required logic gates on the fly have included generating dynamic CVSL gate layouts and connecting them in a standard cell style.

8.4.2.4 An Example of Control-Logic Implementation

Figure 8.89 shows the logic schematic for the Boundary Scan–state machine described in Chapter 7. The state-transition diagram appears in Fig. 7.26. A target cycle time of 100 *ns* was desired.

A state-machine description was written that was automatically fed to the MISII[42] logic-synthesis program. This Lisp-based state-machine language description is shown below:

```
(defpal TAP-FSM-AOI-MUX prototype
  (ipin 2 reset)
```

Micron Ex. 1037, p. 645
Micron v. YMTC
IPR2025-00119

```
(ipin 3 TMS)
(ipin 4 -TCK)
(opin 100 clockir)
(opin 101 updateir)
(rpin 102 shiftir)
(opin 103 clockdr)
(opin 104 updatedr)
(rpin 105 shiftdr)
(rpin 106 enable)
(opin 107 select)
(rpin 108 -reset)
(rpin 20 state<0> h :polarity-fuse)
(rpin 21 state<1> h :polarity-fuse)
(rpin 22 state<2> h :polarity-fuse)
(rpin 23 state<3> h :polarity-fuse)

(setq updateir (and state-update-ir  -TCK))
(setq updatedr (and state-update-dr  -TCK))
(setq clockir (not (and (or state-shift-ir state-capture-ir)
             -TCK)))
(setq clockdr (not (and (or state-shift-dr state-capture-dr)
             -TCK)))
(setq shiftir state-shift-ir :clock -tck )
(setq -reset (not state-test-logic-reset) :clock -tck)
(setq shiftdr state-shift-dr :clock -tck)
(setq enable (or state-shift-ir state-shift-dr) :clock -tck)
(setq select (not (or state-exit2-ir state-exit1-ir
             state-shift-ir
             state-pause-ir state-run-test-idle
                   state-update-ir
             state-capture-ir state-test-logic-reset)))

(macro
  (state-machine
    ;;State transitions
    '(;;Idle—wait for car
            (test-logic-reset
              (:next run-test-idle (not tms))
              (:next test-logic-reset tms)
              (:reset reset))
            (run-test-idle
              (:next select-dr-scan tms)
              (:next run-test-idle (not tms)))
            (select-dr-scan
              (:next select-ir-scan tms)
              (:next capture-dr (not tms)))
            (capture-dr
              (:next exit1-dr tms)
              (:next shift-dr (not tms)))
```

Micron Ex. 1037, p. 646
Micron v. YMTC
IPR2025-00119

```
(shift-dr
  (:next exit1-dr tms)
  (:next shift-dr (not tms)))
(exit1-dr
  (:next update-dr tms)
  (:next pause-dr (not tms)))
(pause-dr
  (:next exit2-dr tms)
  (:next pause-dr (not tms)))
(exit2-dr
  (:next update-dr tms)
  (:next shift-dr (not tms)))
(update-dr
  (:next select-dr-scan tms)
  (:next run-test-idle (not tms)))
(select-ir-scan
  (:next test-logic-reset tms)
  (:next capture-ir (not tms)))
(capture-ir
  (:next exit1-ir tms)
  (:next shift-ir (not tms)))
(shift-ir
  (:next exit1-ir tms)
  (:next shift-ir (not tms)))
(exit1-ir
  (:next update-ir tms)
  (:next pause-ir (not tms)))
(pause-ir
  (:next exit2-ir tms)
  (:next pause-ir (not tms)))
(exit2-ir
  (:next update-ir tms)
  (:next shift-ir (not tms)))
(update-ir
  (:next select-dr-scan tms)
(:next run-test-idle (not tms)))
  )
;;State number assignments
  '((exit2-dr 0)
  (exit1-dr 1)
  (shift-dr 2)
  (pause-dr 3)
  (select-ir-scan 4)
  (update-dr 5)
  (capture-dr 6)
  (select-dr-scan 7)
  (exit2-ir 8)
```

Micron Ex. 1037, p. 647
Micron v. YMTC
IPR2025-00119

```
  (exit1-ir 9)
  (shift-ir #xA)
  (pause-ir #xB)
  (run-test-idle #xC)
  (update-ir #xD)
  (capture-ir #xE)
  (test-logic-reset #xF))
;;Base name for state variables (optional)
"state"))
  )
```

This simple language defines the inputs and outputs and then lists the states and their transitions. State assignment is done manually, although in general, software is available to do this task.

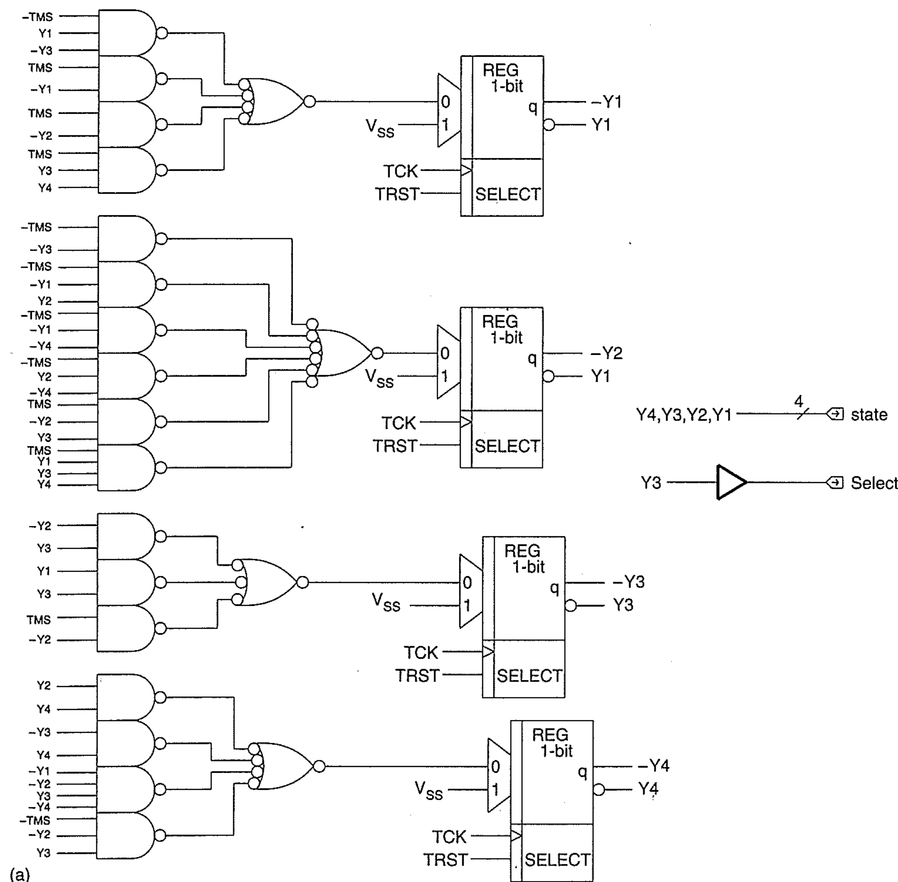


FIGURE 8.89 Boundary-scan tap controller design

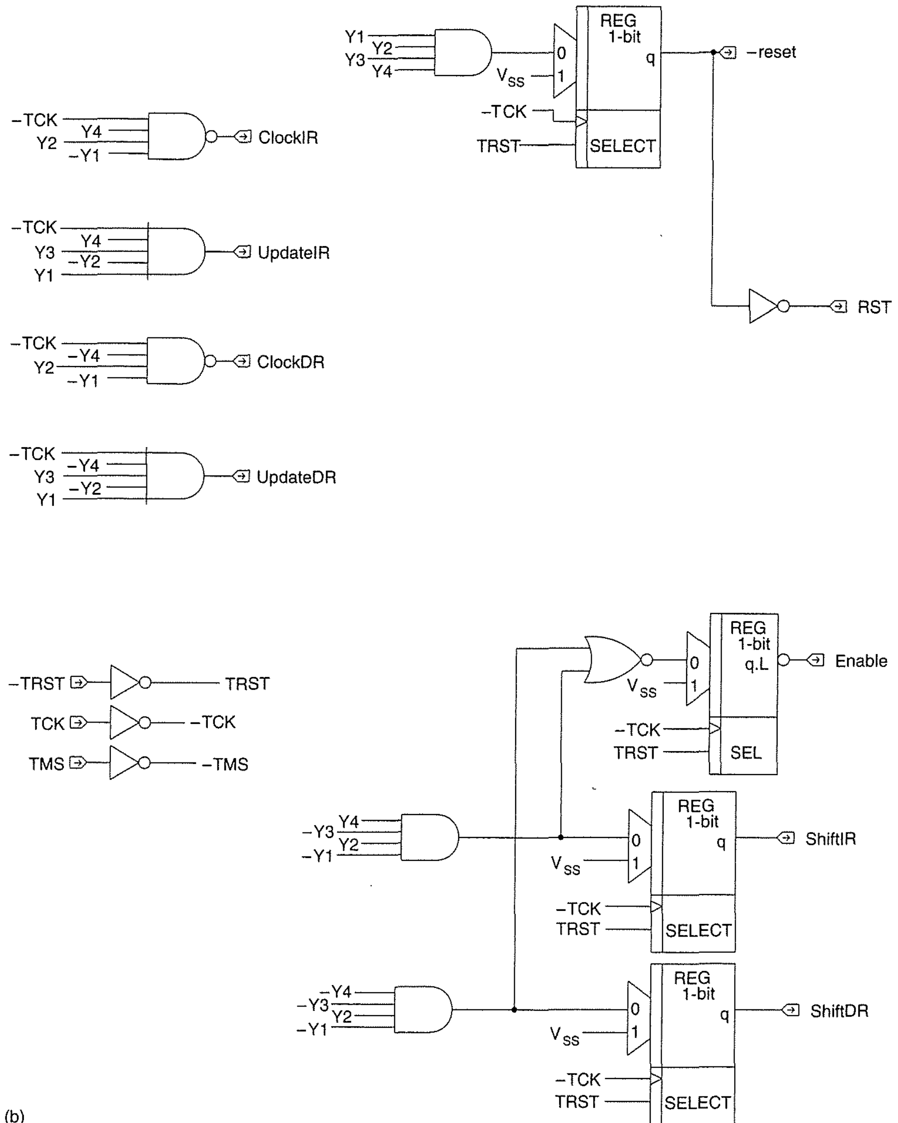


FIGURE 8.89 *(continued)*

When compiled, this state-machine description built the following set of logic equations that were fed to MISII.

```
INORDER = RESET TMS STATE<2> -TCK STATE<0> STATE<1> STATE<3> ;
OUTORDER = NEXT_STATE<3> NEXT_STATE<2> NEXT_STATE<1> NEXT_STATE<0>
NEXT_-RESET SELECT NEXT_ENABLE NEXT_SHIFTDR UPDATEDR CLOCKDR
NEXT_SHIFTIR UPDATEIR CLOCKIR ;
CLOCKIR = !(-TCK * !STATE<0> * STATE<1> * STATE<3>);
UPDATEIR = !(!-TCK + !STATE<0> + STATE<1> + !STATE<2> + !STATE<3>);
NEXT_SHIFTIR = !(STATE<0> + !STATE<1> + STATE<2> + !STATE<3>);
CLOCKDR = !(-TCK * !STATE<0> * STATE<1> * !STATE<3>);
UPDATEDR = !(!-TCK + !STATE<0> + STATE<1> + !STATE<2> + STATE<3>);
NEXT_SHIFTDR = !(STATE<0> + !STATE<1> + STATE<2> + STATE<3>);
NEXT_ENABLE = !(STATE<0> + !STATE<1> + STATE<2>);
```


Micron Ex. 1037, p. 649
Micron v. YMTC
IPR2025-00119

```
SELECT = !(STATE<3>);
NEXT_-RESET = !(STATE<0> * STATE<1> * STATE<2> * STATE<3>);
     NEXT_STATE<0> = (!STATE<1>*TMS + !STATE<0>*STATE<1>*TMS +
                          STATE<0>*!STATE<2>*!TMS + RESET +
STATE<0>*STATE<1>*STATE<2>*STATE<3>*TMS);
NEXT_STATE<1>=(!STATE<2>*!TMS+STATE<1>*STATE<2>*!STATE<3>*!TMS+
              !STATE<0>*!STATE<1>*STATE<2>*!STATE<3> + RESET +
                     STATE<0>*STATE<1>*STATE<2>*STATE<3>*TMS +
                  !STATE<0>*STATE<1>*STATE<2>*STATE<3>*!TMS +
                          STATE<0>*!STATE<1>*STATE<2>*TMS +
!STATE<0>*!STATE<1>*STATE<2>*STATE<3>*TMS);
NEXT_STATE<2>=(!STATE<0>*!STATE<1>*STATE<2>+STATE<0>*STATE<2>+
!STATE<1>*!STATE<2>*TMS + RESET);
           NEXT_STATE<3> = (!STATE<0>*STATE<2>*STATE<3>*!TMS +
             STATE<1>*STATE<3>*TMS + !STATE<2>*STATE<3>*!TMS +
!STATE<0>*!STATE<1>*STATE<2>*!STATE<3> + RESET +
!STATE<1>*!STATE<2>*STATE<3>*TMS+STATE<0>*!STATE<1>*STATE<2>*TMS+
STATE<0>*STATE<1>*STATE<2>*STATE<3>*!TMS);
```

This description is read into MISII, converted to sum-of-products form and written out as a PLA description. The following is the expanded input in sum of products form:

```
.i 7
.o 13
.ilb RESET TMS STATE<2> -TCK STATE<0> STATE<1>
STATE<3>
.ob NEXT_STATE<3> NEXT_STATE<2> NEXT_STATE<1> NEXT_-
STATE<0> NEXT_-RESET SELECT NEXT_ENABLE NEXT_SHIFTDR
UPDATEDR CLOCKDR NEXT_SHIFTIR UPDATEIR CLOCKIR
.p 41
1------ 1000000000000
-0----1 1000000000000
--0---1 1000000000000
-----11 1000000000000
--1-000 1000000000000
001-10- 1000000000000
1------ 0100000000000
--1-1-- 0100000000000
-1---0- 0100000000000
--1--0- 0100000000000
1------ 0010000000000
000---- 0010000000000
-00--1- 0010000000000
-0--01- 0010000000000
-11--0- 0010000000000
-0---10 0010000000000
-11-1-1 0010000000000
--1-000 0010000000000
1------ 0001000000000
```

```
-1--0-- 0001000000000
-1---0- 0001000000000
000-1-- 0001000000000
-11-111 0001000000000
--0---- 0000100000000
----0-- 0000100000000
------0 0000100000000
-----0- 0000100000000
------0 0000010000000
--0-01- 0000001000000
--0-010 0000000100000
--11100 0000000010000
---0--- 0000000001000
----1-- 0000000001000
-----0- 0000000001000
------1 0000000001000
--0-011 0000000000100
--11101 0000000000010
---0--- 0000000000001
----1-- 0000000000001
-----0- 0000000000001
------0 0000000000001
.e
```

The initial statements indicate the number of inputs (`.i`), outputs (`.o`), and product terms (`.p`). For each product term (line), the six inputs on the left are coded in terms of zero (`0`), one (`1`) or don't care (`-`). The inputs are ordered according to the `.ilb` statement. The outputs on the right are ordered according to the `.ob` statement. This description is then minimized by the Espresso sum-of-products minimizer. The output in sum-of-products form is shown below:

```
.i 7
.o 13
.ilb RESET TMS STATE<2> -TCK STATE<0> STATE<1> STATE<3>
.ob NEXT_STATE<3> NEXT_STATE<2> NEXT_STATE<1> NEXT_STATE<0>
NEXT_-RESET SELECT NEXT_ENABLE NEXT_SHIFTDR UPDATEDR CLOCKDR
NEXT_SHIFTIR UPDATEIR CLOCKIR
.p 21
--11100 0000000010000
--11101 0000000000010
--0-010 0000001100000
--0-011 0000001000100
-11-1-1 0011000000000
--1-000 1010000000000
-0---10 0010000000000
-00-1-- 0011000000000
-0--01- 0010100000000
```

Micron Ex. 1037, p. 651
Micron v. YMTC
IPR2025-00119

```
-11--0- 0010100000000
-00--0- 0010000001001
--1-1-- 0100000000000
-1--0-- 0001100000000
--0---1 1000100000000
-----11 1000000001000
-01--0- 1100100001001
-1---0- 0101000001001
---0--- 0000000001001
------0 0000110000001
----1-- 0000000001001
1------ 1111000000000
.e
```

This reduced sum-of-products form has 21 product terms. A representative pseudo-nMOS PLA layout is shown in Fig. 8.90. The registers have been arrayed across the bottom of the PLA because this allows the inputs and outputs to the PLA to be placed on the bottom of the PLA. If the registers are incorporated into the PLA itself, the inputs and outputs have to be placed on the top and bottom of the PLA to achieve a small pitch. In this implementation, the input pitch was 7.5µ, the minterm pitch was 5.25µ, and the output pitch was 4.5µ. The layout shown was 181µ wide by 270µ high. Minimum-sized n-transistors were used, although these could be increased in size to improve the speed if necessary. As designed, the state machine implemented with a PLA could operate at a worst-case cycle time of 10 *ns*. The PLA dissipates 8 mW at 10 MHz. Of this around 5 to 6 mW are due to DC dissipation in the p pull-ups. A dynamic version was estimated to dissipate about 3 mW.

A manual schematic design was also completed for the design shown in Fig. 8.89. In addition, a library file, describing the standard-cell library, was input to MISII. Each gate denotes its name, size, logic equation, and timing behavior. The library is shown below:

```
# Area is approximate virtual grid squares
# Name Area Equation
# <phase> <input load> <max load>
# <rise-block-delay> <rise-fanout-delay> <fall-block-delay>
<fall-fanout-delay>
GATE ZERO     0    O=CONST0;
GATE ONE      0    O=CONST1;
GATE XOR      420  O=A*!B+!A*B;       PIN * UNKNOWN 2 50
                                      0.52  0.4  .45  .54
GATE XNOR     420  O=A*B+!A*!B;       PIN * UNKNOWN 2 50
                                      0.5  0.4  .31  0.4
GATE OR3      300  O=A+B+C;           PIN * NONINV 1 50
                                      0.33  0.14  .81  0.12
GATE OR2      240  O=A+B;             PIN * NONINV 1 50
                                      0.36  0.13  .49  0.10
GATE NOR4     300  O=!(A+B+C+D);      PIN * INV 1 50
                                      0.31  0.5  .35  0.12
```

Micron Ex. 1037, p. 652
Micron v. YMTC
IPR2025-00119

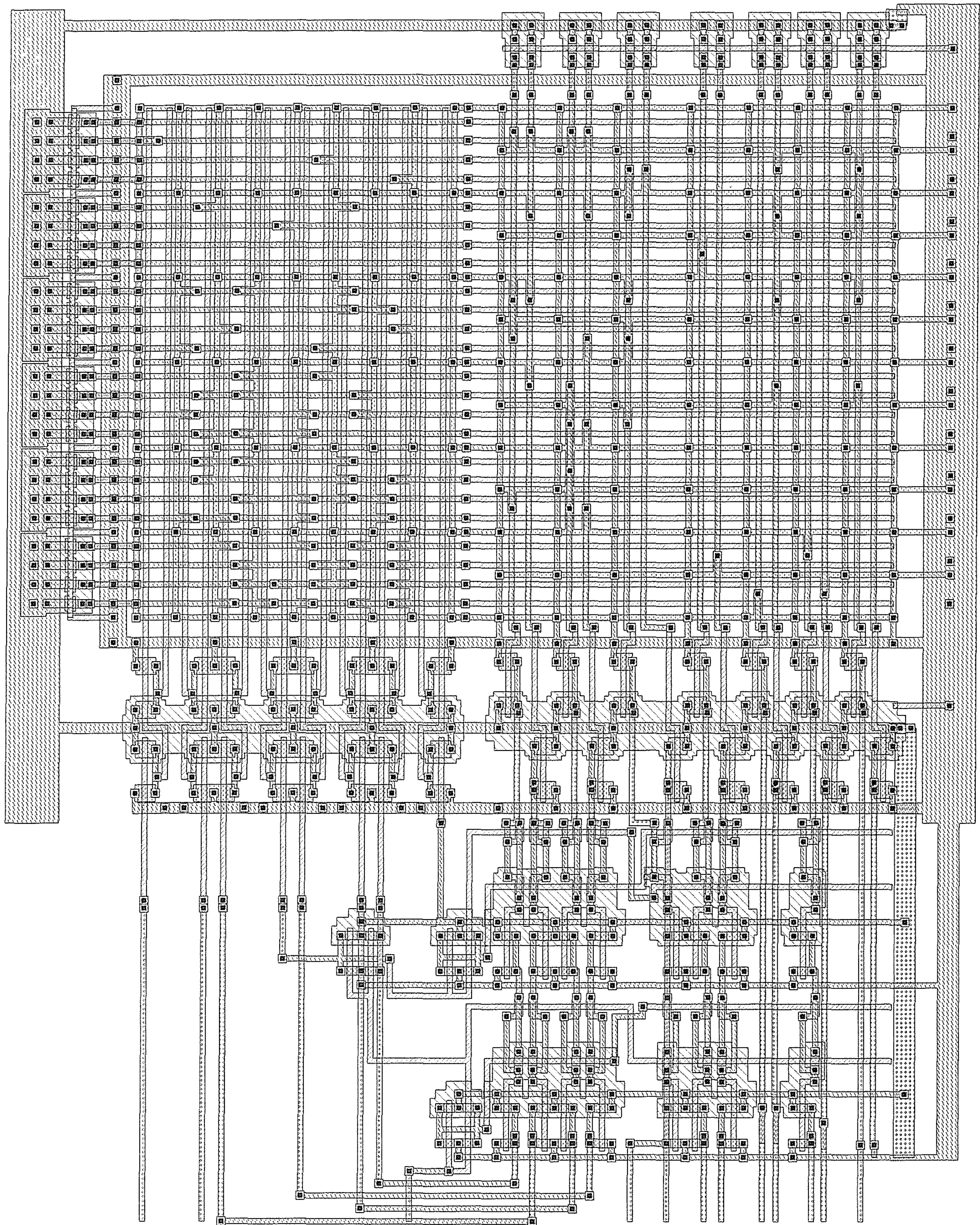


FIGURE 8.90 PLA layout for boundary-scan tap controller design

Micron Ex. 1037, p. 653
Micron v. YMTC
IPR2025-00119

```
GATE NOR3      240  O=!(A+B+C);        PIN * INV 1 50
                                       0.29  0.4  .35  0.12
GATE NOR2      180  O=!(A+B);          PIN * INV 1 50
                                       0.17  0.26  .29  0.12
GATE NAND4     300  O=!(A*B*C*D);      PIN * INV 1 50
                                       0.16  0.14  .36  0.4
GATE NAND3     240  O=!(A*B*C);        PIN * INV 1 50
                                       0.14  0.14  .33  0.3
GATE NAND2     180  O=!(A*B);          PIN * INV 1 50
                                       0.18  0.14  .28  0.2
GATE INVERTER 120  O=!A;               PIN * INV 1 50
                                       0.29  0.14  .1  0.12
GATE BUFFER    180  O=A;               PIN * NONINV 1 50
                                       0.6  0.14  .2  0.12
GATE AND4      360  O=A*B*C*D;         PIN * NONINV 1 50
                                       0.73  0.14  .37  0.2
GATE AND3      300  O=A*B*C;           PIN * NONINV 1 50
                                       0.37  0.12  .64  0.1
GATE AND2      240  O=A*B;             PIN * NONINV 1 50
                                       0.47  0.12  .31  0.1
GATE AOI21     240  O=!(A+(B*C));      PIN * INV 1 50
                                       0.17  0.2  .28  0.2
GATE OAI21     240  O=!(A*(B+C));      PIN * INV 1 50
                                       0.17  0.2  .28  0.2
GATE MUX2      480  O=((A*!S)+(B*S)); PIN * NONINV 1 50
                                       0.14  0.2  .28  0.3
GATE MUX2-INV 420  O=!((A*!S)+(B*S));PIN * INV 1 50
                                       0.45  0.14  .6 0.1
```

MISII examined the logic equations, minimized the logic, and created a netlist in terms of the library gates. The netlist output of MISII is shown below:

```
.model tap-fsm-aoi.eqn
.inputs RESET TMS STATE<2> -TCK STATE<0> STATE<1> STATE<3>
.outputs NEXT_STATE<3> NEXT_STATE<2> NEXT_STATE<1> NEXT_-
STATE<0> NEXT_-RESET \
SELECT NEXT_ENABLE NEXT_SHIFTDR UPDATEDR CLOCKDR NEXT_-
SHIFTIR UPDATEIR CLOCKIR
.default_input_arrival 0.00 0.00
.default_output_required 0.00 0.00
.default_input_drive 0.14 0.12
.default_output_load 1.00
.gate INVERTER A=STATE<0> O=[324]
.gate INVERTER A=STATE<2> O=[323]
.gate NOR2 A=STATE<0> B=[323] O=[312]
.gate INVERTER A=STATE<3> O=SELECT
.gate INVERTER A=STATE<1> O=[327]
.gate OR2 A=STATE<0> B=[327] O=[329]
.gate NAND2 A=[329] B=STATE<2> O=[481]
.gate INVERTER A=RESET O=[350]
```

Micron Ex. 1037, p. 654
Micron v. YMTC
IPR2025-00119

```
.gate INVERTER A=TMS O=[326]
.gate OAI21 A=[350] B=STATE<1> C=[326] O=[328]
.gate INVERTER A=[328] O=[449]
.gate NAND2 A=[481] B=[449] O=NEXT_STATE<2>
.gate NAND3 A=[312] B=SELECT C=NEXT_STATE<2> O=[471]
.gate NOR2 A=[323] B=[324] O=[302]
.gate NOR2 A=SELECT B=[327] O=[299]
.gate NAND2 A=[302] B=[299] O=NEXT_-RESET
.gate NAND2 A=NEXT_-RESET B=STATE<0> O=[461]
.gate NAND2 A=[461] B=TMS O=[479]
.gate NAND2 A=[326] B=STATE<0> O=[331]
.gate OAI21 A=[479] B=NEXT_STATE<2> C=[331] O=[334]
.gate OR2 A=[328] B=[334] O=NEXT_STATE<0>
.gate NAND2 A=[327] B=STATE<0> O=[335]
.gate OAI21 A=[471] B=NEXT_STATE<0> C=[335] O=[336]
.gate NOR3 A=STATE<1> B=[323] C=[326] O=[202]
.gate OAI21 A=[350] B=SELECT C=[202] O=[338]
.gate OR2 A=[336] B=[338] O=NEXT_STATE<3>
.gate AND2 A=NEXT_STATE<2> B=STATE<2> O=[272]
.gate AND2 A=NEXT_STATE<0> B=[272] O=[205]
.gate NOR2 A=STATE<0> B=STATE<3> O=[268]
.gate AOI21 A=[205] B=NEXT_STATE<3> C=[268] O=[258]
.gate NAND2 A=NEXT_-RESET B=STATE<1> O=[469]
.gate NAND2 A=NEXT_STATE<2> B=[469] O=[459]
.gate AOI21 A=RESET B=[326] C=[459] O=[255]
.gate NAND2 A=[258] B=[255] O=NEXT_STATE<1>
.gate OR2 A=STATE<2> B=[329] O=[343]
.gate INVERTER A=[343] O=NEXT_ENABLE
.gate OR2 A=STATE<3> B=[343] O=[433]
.gate INVERTER A=[433] O=NEXT_SHIFTDR
.gate INVERTER A=-TCK O=[344]
.gate NOR2 A=[323] B=[344] O=[248]
.gate NOR2 A=STATE<1> B=[324] O=[245]
.gate NAND2 A=[248] B=[245] O=[347]
.gate NOR2 A=STATE<3> B=[347] O=UPDATEDR
.gate NAND2 A=SELECT B=-TCK O=[348]
.gate OR2 A=[329] B=[348] O=CLOCKDR
.gate AND2 A=[433] B=NEXT_ENABLE O=NEXT_SHIFTIR
.gate NOR2 A=SELECT B=[347] O=UPDATEIR
.gate NAND2 A=STATE<3> B=-TCK O=[349]
.gate OR2 A=[329] B=[349] O=CLOCKIR
.end
```

This was fed to the NS Design System[43], where a program-generated schematic was created. This schematic was extracted, and netlist information and physical cell details were fed to the TimberWolf[44] placement program. The placement information was then returned to the NS VLSI design system where the circuit was automatically routed and a symbolic standard-cell lay-

Micron Ex. 1037, p. 655
Micron v. YMTC
IPR2025-00119

out created. After compaction to a set of CMOS-process design rules, the final mask layouts were available for backannotation, simulation, timing analysis, and size comparison. This complete integrated process (Fig. 8.91) completes automatically from changing any of the primary inputs in about five minutes. A number of "knobs" may be turned to affect the size of the layout. Starting from the top, a number of different logic-synthesis scripts and a variety of standard-cell libraries, varying from a 2-input NAND, a two-input NOR, and an INVERTER to more extensive collections of gates were tried. The number of rows that TimberWolf used was also varied.

Finally, the standard-cell height may be varied to create "performance" (large) or "area" (small) conscious layouts. Alternatively with the system described above, additional characterized symbolic standard cells may be added in about 5 minutes. The process may also be changed, thus creating a further dimension for optimization. Because all the characterization tools (simulation, timing analysis) work at the transistor level, new cells may be added with ease. Moreover with sophisticated placement programs, standard-cell layouts may be "reflowed" into unused space between larger fixed blocks. Table 8.10 summarizes the results.

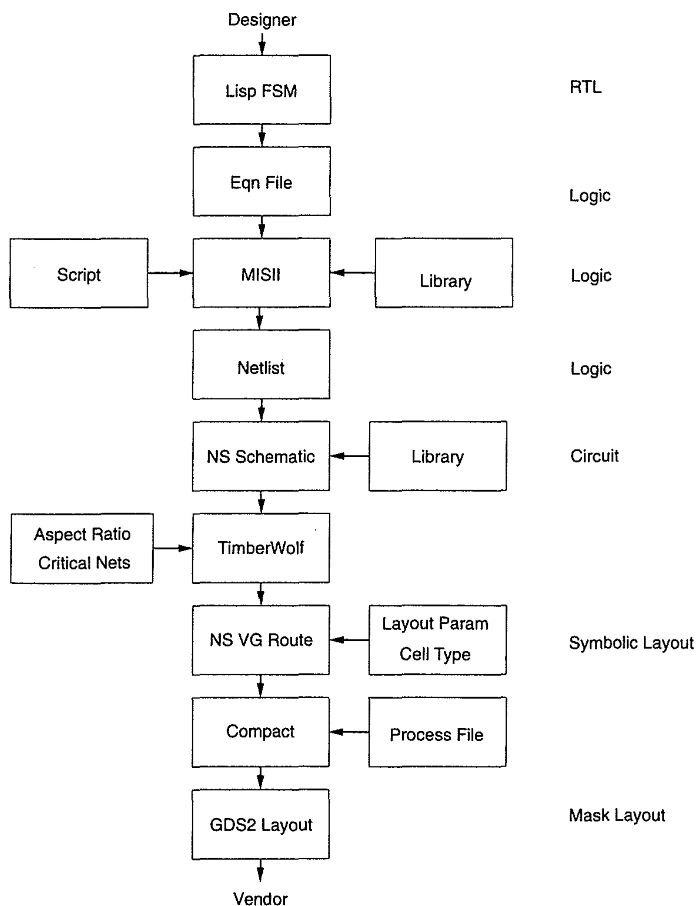


FIGURE 8.91 Control-logic design process

Micron Ex. 1037, p. 656
Micron v. YMTC
IPR2025-00119

In Table 8.10, the *basic* library had (in addition to registers):

inverter
nand2
nor2.

The *tlw* library had the following gates (in addition to registers):

inverter
nand2,nand3,nand4,and2,and3,and4
nor2,nor3,nor4,or2,or3,or4
xor,xnor.

The *tlw-aoi* library had, in addition to the *tlw* library, the following four gates:

mux2,mux-inverting
and-or-invert $Z = !(A \,.\, (B + C))$
or-and-invert $Z = !(A + (B \,.\, C))$.

Table 8.10 shows the flexibility in aspect ratio that can be gained with standard-cell layouts. It also illustrates that metal3 provides layouts that are about half the size of the metal2 counterparts. Combined with RTL synthesis and logic optimization, the standard-cell approach provides an excellent means of capturing control logic (and other logic styles also). The final layout after compaction for one of the standard-cell layouts for a two-level-metal process is shown in Fig. 8.92. A metal3 layout is shown in Fig. 8.93 (also Plate 9).

Finally, because this state machine was very small, a gate-matrix layout was completed by hand (assuming that an automatic synthesis program was

TABLE 8.10 Standard-cell Layout Options

LOGIC	# ROWS	SIZE (mm^2)	LIBRARY	STD-CELL HEIGHT	METAL LAYERS	DIM $X \times Y$ (mm)
Fig. 8.89	2	.120	tlw	50μ	2	.67 × .18
MISII	2	.118	basic	50μ	2	.62 × .19
MISII	2	.099	tlw-aoi	50μ	2	.52 × .19
MISII	3	.1	tlw-aoi	50μ	2	.4 × .25
MISII	4	.122	tlw-aoi	50μ	2	.34 × .33
MISII	2	.058	tlw-aoi	50μ	3	.49 × .12
MISII	3	.058	tlw-aoi	50μ	3	.34 × .17
MISII	3	.074	tlw-aoi	30μ	2	.4 × .19

Micron Ex. 1037, p. 657
Micron v. YMTC
IPR2025-00119

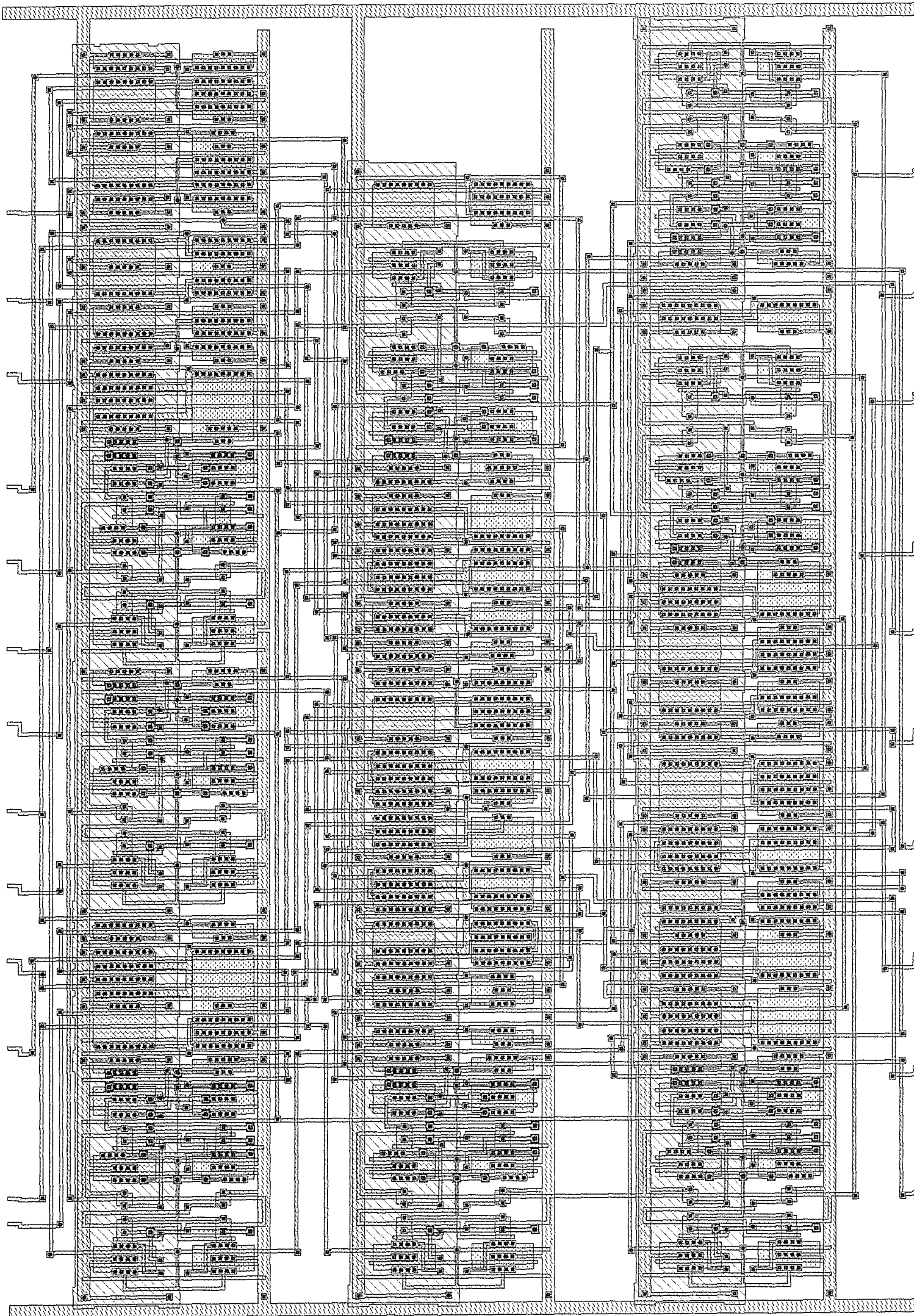


FIGURE 8.92 Metal2 standard-cell layout for boundary-scan tap controller

Micron Ex. 1037, p. 658
Micron v. YMTC
IPR2025-00119

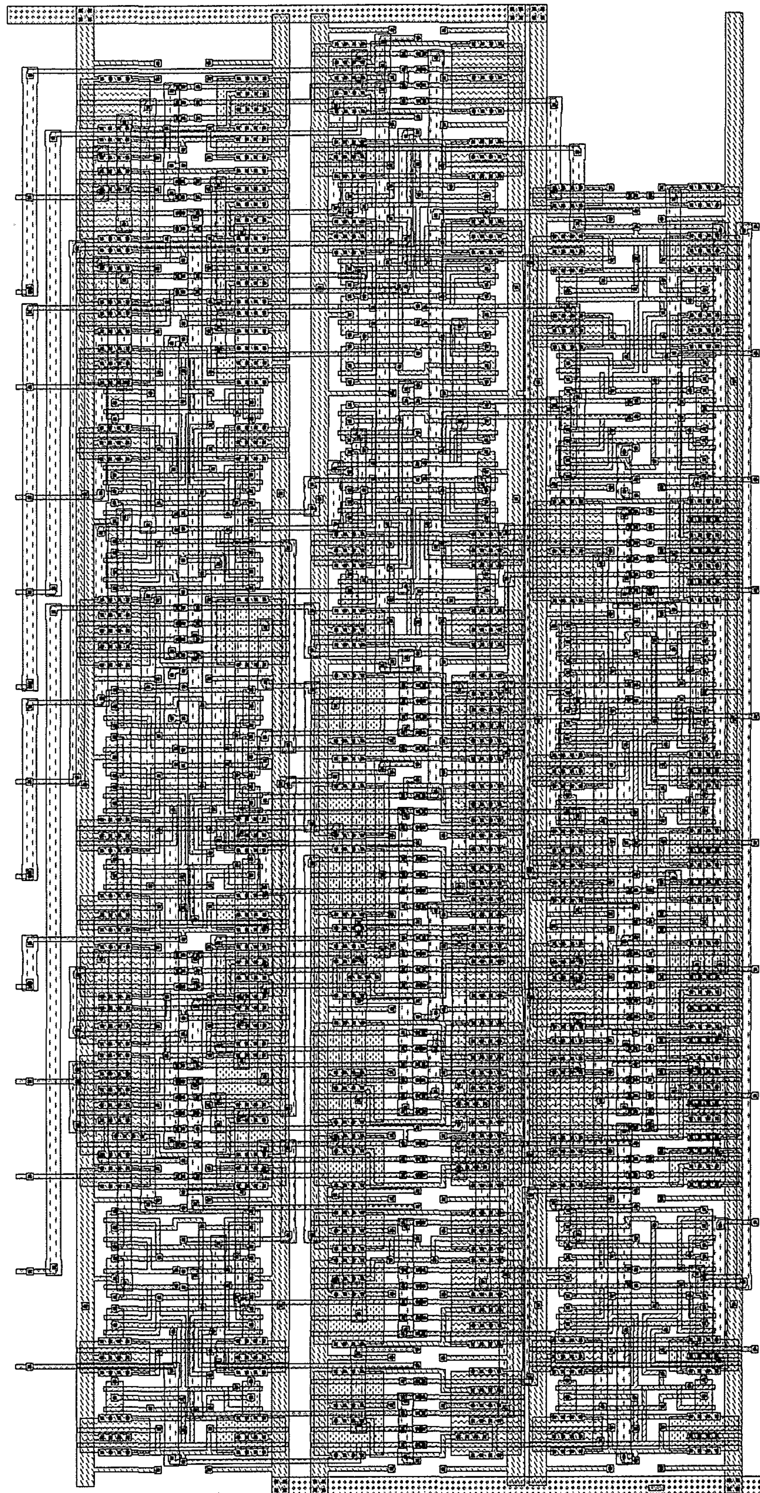


FIGURE 8.93 Metal3 standard-cell layout for boundary-scan tap controller

Micron Ex. 1037, p. 659
Micron v. YMTC
IPR2025-00119

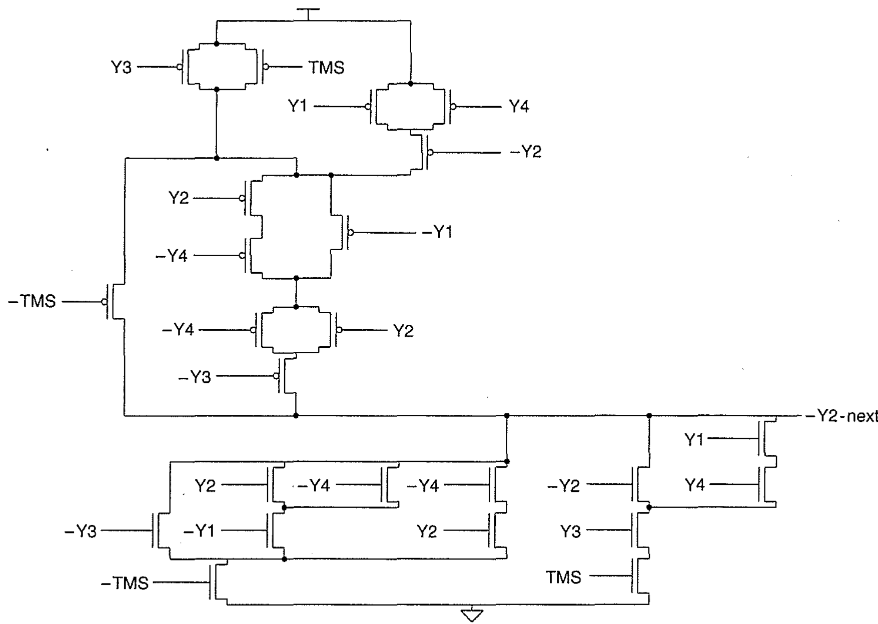


FIGURE 8.94 An example of a complex CMOS gate used in the gate-matrix implementation

available for production designs). In this design minimum-size transistors were used and large AOI gates were used to implement the more complex terms. Figure 8.94 shows the gate used to implement the Y_2-*next* term. The design could use dynamic logic and dynamic registers to reduce the size and power even further. The registers were placed to the right of the gate-matrix logic section. The layout is shown in Fig. 8.95.

Table 8.11 summarizes the area, speed, and power of the three implementations. The various implementations exhibit their strong points. The PLA is easy to design in a fully automated manner, is small, and for this application is fast enough. A dynamic version would have lower power dissipation. It is, however, fairly fixed in size. The standard-cell design may also be fully automated, as with the PLA, from a state-machine description. The speed and size may be varied over some range by logic-synthesis techniques and the avail-

Table 8.11 Area, Speed, and Power of Control Implementations

STYLE	AREA	SPEED	POWER
PLA	.050 mm^2	10 ns	8 mW
Standard Cell (DLM)	.099 mm^2	10 ns	6.3 mW
Standard Cell (TLM)	.058 mm^2	~10 ns	~6 mW
Gate Matrix	.032 mm^2	15 ns	1.5 mW

Micron Ex. 1037, p. 660
Micron v. YMTC
IPR2025-00119

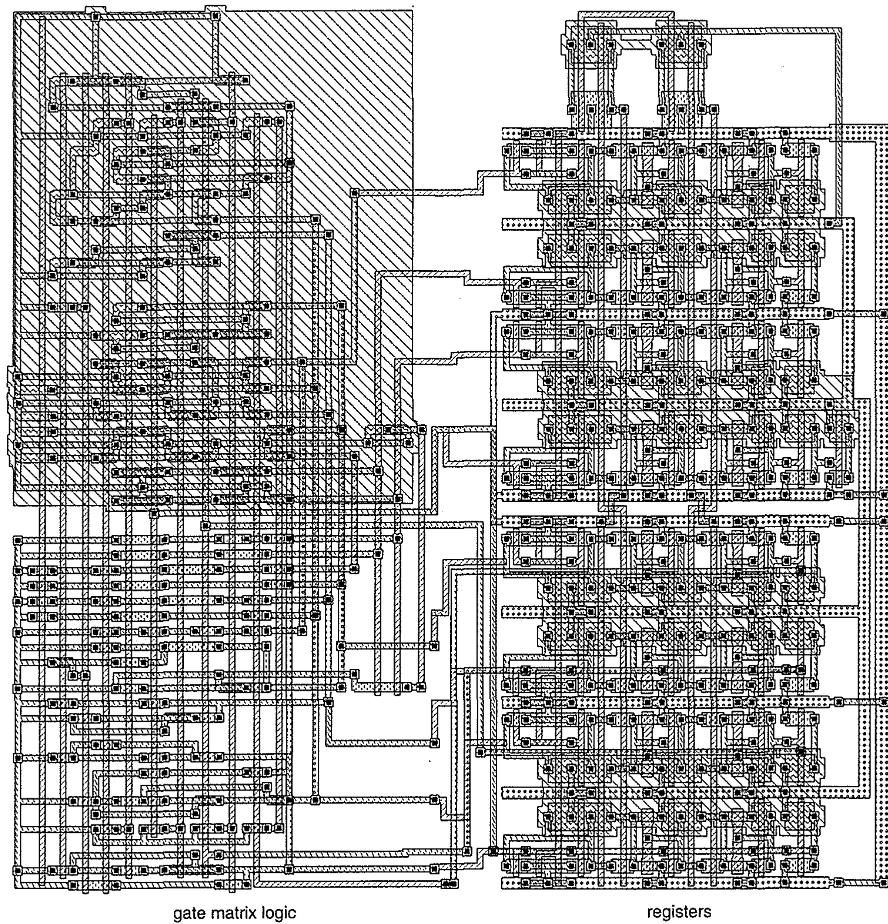


Figure 8.95 Gate-matrix layout for boundary-scan tap controller

ability of a continuously variable symbolic layout library. The aspect ratio and shape of the resulting layout is primarily dictated by the placement program and in theory may be varied with a granularity down to the size of a gate (~20μ × 50μ). Finally, the gate-matrix layout represents the smallest and lowest power-dissipation case (which is largely due to the fact that this example is so simple). This comes at the cost of speed (although this is not important in this application) and time to design because an automated approach was not available. This approach is not recommended when suitable automation is not available except under extreme power or size restrictions.

8.5 Summary

This chapter has presented a range of sub-system designs in terms of datapath, memory, and control elements. Coupled with I/O structures, these form the basic building blocks from which larger systems may be hierarchically

Micron Ex. 1037, p. 661
Micron v. YMTC
IPR2025-00119

structured. How one goes about designing and implementing a given CMOS chip is largely affected by the availability of tools, the schedule, the complexity of the system, and the final cost goals of the chip. In general, the simplest and least expensive (in terms of time and money) approach that meets the target goals should be chosen. Given an ideal tool set that can draw on large libraries of predefined components, synthesis tools, and integrated VLSI backend tools, an almost continuous trade-off may be made to select a particular approach. The following rules may be applied. Where the design is of moderate complexity (a description that changes with time) and time to silicon is of paramount importance, an FPGA approach is probably suitable. If speed or complexity eliminate this approach, a gate-array is the next logical choice. To cost-reduce a gate-array or include sizable memories, a standard-cell approach is the next option. The next step to symbolic layout of regular arrays (i.e., datapaths, special memories, cellular arrays) and standard-cell control logic, is a big one. This step is usually taken on large-volume chips where the complexity, speed, and area dictate a custom approach. At this time in the evolution of VLSI, the final step of a full-custom, micron-tweaked design should almost never occur except for small, highly optimized circuits. As for capture methods, both HDLs and schematics have their strong points. It is likely that an increasing number of designers will move to HDLs, but schematics will be around for some time yet.

8.6 Exercises

1. Design an 8-bit parallel accumulator (adder and register) that is optimized for low power and has a power-down capability. Show how your circuit would retain the stored state.
2. Design a 32-bit parallel adder optimized for speed, single-cycle operation, and regularity of layout. Repeat the exercise with no layout restrictions.
3. Show how the layout of the parity generator in Fig. 8.25(a) may be designed as a linear column of XOR gates with a tree-routing channel.
4. Design an 8-bit barrel shifter (i.e., arbitrary left or right rotate) using multiplexers. Explain what the performance limitations of your design might be.
5. A four-section Finite Impulse Response (FIR) filter employs fixed coefficients and implements the function

$$Y = [2 \times X(t)] + [4 \times X(t-1)] + [8 \times X(t-2)] + [2 \times X(t-2)] + [4 \times X(t-3)],$$

Micron Ex. 1037, p. 662
Micron v. YMTC
IPR2025-00119

where $X(t)$ is the sampled value of an 8-bit input at time t. Design a circuit to implement this function.

6. Design an FSM to control the stop lights at a four-way intersection with pedestrian crossing. Implement the control logic as a PLA, as multiple-level logic, and as a ROM microcontroller.
7. Tong and Jha[45] describe a binary divider. Design the base cell and show how you would complete a layout for the divider.
8. Complete a floorplan (showing clock and power and ground routing) and the circuit design for key cells (i.e., memory cell, row decoder, column decoder, sense amplifier) in a three-read-port, two-write-port register file. What simulations would you carry out to ensure the performance and justify the selection of power and ground bus widths?

8.7 References

1. Peter Denyer and David Renshaw, *VLSI Signal Processing: A Bit-Serial Approach,* Reading, Mass.: Addison-Wesley, 1985.
2. P. B. Denyer and D. J. Myers, "Carry-save arrays for VLSI signal processing," *VLSI 81* (John Gray, ed.), Edinburgh, Scotland: Academic Press, pp. 151–160.
3. Yasoji Suzuki, Kaichiro Odagawa, and Toshio Abe, "Clocked CMOS Calculator Circuitry," *IEEE Journal of Solid State Circuits,* vol. SC-8, no. 6, Dec. 1973, pp. 734–739.
4. Nan Zhuang and Haomin Wu, "A new design of the CMOS full adder," *IEEE JSSC,* vol. 27, no. 5, May 1992, pp. 840–844.
5. Kazuo Yano, Toshiaki Yamanaka, Takashi Nishida, Masayoshi Saito, Katsuhiro Shimohigashi, and Akihiro Shimizu, "A 3.8-ns CMOS 16*16-b multiplier using complementary pass-transistor logic," *IEEE JSSC,* vol. 25, no. 2, Apr. 1990, pp. 388–395.
6. M. Pomper, W. Biefuss, K. Horinger, and W. Kaschite, "A 32-bit execution unit in an advanced nMOS technology," *IEEE JSSC,* vol. SC-17, no. 3, Jun. 1982, pp. 533–538.
7. T. Sato, M. Sakate, H. Okada, T. Sukemara, and G. Goto, "An 8.5ns 112-b transmission gate adder with a conflict-free bypass circuit," *IEEE JSSC,* vol. 27, no. 4, Apr. 1992, pp. 657–659.
8. T. Sato, et al., *op. cit.*
9. M. Uya, K. Kaneko, and J. Yasui, "A CMOS floating point multiplier, IEEE International Solid-State Circuits Conference, *Digest of Technical Papers,* Feb. 1984, pp. 90–91.
10. J. Slansky, "Conditional-sum addition logic," *IRE Transactions on Electronic Computers,* vol. EC-9, Jun. 1960, pp. 226–231.
11. Albrecht Rothermel, Bedrich J. Hosticka, Gerhard Troster, and Juergen Arndt, "Realization of transmission-gate conditional-sum (TGCS) adders with low latency time," *IEEE JSSC,* vol. 24, no. 3, Jun. 1989, pp. 558–561.

Micron Ex. 1037, p. 663
Micron v. YMTC
IPR2025-00119

12. T. Sato et al., *op. cit.*
13. M. Y. Tsai, "High density parity-checking circuits with pass transistors," *IBM Technical Disclosure Bulletin,* vol. 26, no. 3A, Aug. 1983, pp. 959–960.
14. W. R. Grifton and J. A. Hiltebeitel, "CMOS four-way XOR circuit," *IBM Technical Disclosure Bulletin,* vol. 25, no. 11B, Apr. 1983, pp. 6066–6067.
15. "TTL Databook," Texas Instruments, Dallas, Tex.
16. Gensuke Goto, Tomio Sato, Masao Nakajima, and Takao Sukemura, "A 54 * 54 regularly structured tree multiplier," *IEEE JSSC,* vol. 27, no. 9, Sept. 1992, pp. 1229–1236.
17. R. F. Lyon, "Two's complement pipeline multiplier," *IEEE Transactions on Communications,* vol. COM-24, Apr. 1976, pp. 418–425.
18. Peter Denyer and David Renshaw, *op. cit.*
19. Kunihiko Yamaguchi, Hiroaki Nambu, Kazuo Kanetani, Youji Idei, Noriyuki Homma, Toshiro Hiramoto, Nobuo Tamba, Kunihiko Watanbe, Masanori Odaka, Takahide Ikeda, Kenichi Ohhata, and Yoshiaki Sakurai, "A 1.5-ns access time, 78-μm^2 memory-cell size, 64-kb ECL-CMOS SRAM," *IEEE JSSC,* vol. 27, no. 2, Feb. 1992, pp. 167–174.
20. V. L. Rideout, "One-device cells for dynamic random-access memories," *IEEE Transactions on Electron Devices,* vol. ED-26, Jun. 1979, pp. 839–852.
21. Hideto Hidaka, Kazutami Arimoto, and Kazuyasu Fujishima, "A high density dual-port memory cell operation and array architecture for ULSI DRAM's," *IEEE JSSC,* vol. 27, no. 4, Apr. 1992, pp. 610–617.
22. Dong-Sun Min, Sooin Cho, Dong Soo Jun, Dong-Jae Lee, Yongsik Seok, and Daeje Chin, "Temperature-compensation circuit techniques for high-density CMOS DRAM's, *IEEE JSSC,* vol. 27, no. 4, Apr. 1992, pp. 626–631.
23. Hideto Hidaka, Kazutami Arimoto, Kazutoshi Hirayama, Masanori Hayashikoshi, Mikio Asakura, Masaki Tsukude, Tsukasa Oishi, Shinji Kawai, Katsuhiro Suma, Yasuhiro Konishi, Koji Tanaka, Wataru Wakamiya, Yoshikazu Ohno, and Kazuyasu Fujishima, "A 34-ns 16-Mb DRAM with controllable voltage down-converter," *IEEE JSSC,* vol. 27, no. 7, Jul. 1992, pp. 1020–1027.
24. Toshiaki Kirihata, Sang H. Dhong, Koji Kitamura, Toshio Sunaga, Yasuano Katayama, Roy E. Scheuerlein, Akashi Satoh, Yoshinori Sakaue, Kentaroh Tobimatsu, Koji Hosokawa, Takaki Saitoh, Takefumi Yoshikawa, Hideki Hashimoto, and Michiya Kazusawa, "A 14-ns 4-Mb CMOS DRAM with 300-mW active power," *IEEE JSSC,* vol. 27, no. 9, Sept. 1992, pp. 1222–1228.
25. Hideto Hidaka, Yoshio Matsuda, and Kazuyasu Fujishima, "A divided/shared bit-line sensing scheme for ULSI DRAM cores," *IEEE JSSC,* vol. 26, no. 4, Apr. 1991, pp. 473–478.
26. Toshio Yamada, Yoshiro Nakata, Junko Hasegawa, Noriaki Amano, Akinori Shibayama, Masaru Sasago, Naoto Matsuo, Toshiki Yabu, Susumu Matsumoto, Shozo Okada, and Michihiro Inoue, "A 64-Mb DRAM with meshed power line," *IEEE JSSC,* vol. 26, no. 11, Nov. 1991, pp. 1506–1510.
27. Walter H. Henkels, Duen-Shun Wen, Rick L. Mohler, Robert L. Franch, Thomas J. Bucelot, Christopher W. Long, John A. Bracchitta, W. J. Cote, Gary B. Bronner, Yuan Taur, and Robert H. Dennard, "A 4-Mb low-temperature DRAM," *IEEE JSSC,* vol. 26, no. 11, Nov. 1991, pp. 1519–1529.
28. Takeshi Nagai, Kenji Numata, Masaki Ogihara, Mitsuru Shimizu, Kimimasa Imai, Takahiko Hara, Munehiro Yoshida, Yoshikazu Saito, Yoshiaki Asao, Shizuo Sawada, and Syuso Fujii, "A 17-ns 4-Mb CMOS DRAM," *IEEE JSSC,* vol. 26, no. 11, Nov. 1991, pp. 1538–1543.

Micron Ex. 1037, p. 664
Micron v. YMTC
IPR2025-00119

29. Travis N. Blalock and Richard C. Jaeger, "A high-speed sensing scheme for 1T dynamic RAM's utilizing the clamped bit-line sense amplifier," *IEEE JSSC,* vol. 27, no. 4, Apr. 1992, pp. 618–625.
30. Evert Seevinck, Petrus J. van Beers, and Hana Ontrop, "Current-mode techniques for high-speed VLSI circuits with application to current sense amplifier for CMOS SRAM's," *IEEE JSSC,* vol. 26, no. 4, Apr. 1991, pp. 525–536.
31. Travis N. Blalock and Richard C. Jaeger, "A high-speed clamped bit-line current-mode sense amplifier," *IEEE JSSC,* vol. 26, no. 4, Apr. 1991, pp. 542–548.
32. Shingo Aizaki, Toshiyuki Shimizu, Masayoshi Ohkawa, Kazuhiko Abe, Akane Aizaki, Manabu Ando, Osamu Kudoh, and Isao Sasaki, "A 15-ns 4-Mb CMOS SRAM," *IEEE JSSC,* vol. 25, no. 5, Oct. 1990, pp. 1063–1067.
33. Heribert Geib, Werner Weber, Erdi Wohlrab, and Lothar Risch, "Experimental investigation of the minimum signal for reliable operation of DRAM sense amplifiers," *IEEE JSSC,* vol. 27, no. 7, Jul. 1992, pp. 1028–1035.
34. Richard D. Jolly, "A 9-ns, 1.4-Gigabyte/s, 17-ported CMOS register file," *IEEE JSSC,* vol. 26, no. 10, Oct. 1991, pp. 1407, 1412.
35. Hirofumi Shinohara, Noriaki Matsumoto, Kumiko Fijimori, Yoshiki Tsujihashi, Hiroomi Nakao, Shuishi Kato, Yasutaka Horiba, and Akiharu Tada, "A flexible multiport RAM compiler for data path," *IEEE JSSC,* vol. 26, no. 3, Mar. 1991, pp. 343–349.
36. Alfredo R. Linz, "A low-power PLA for a signal processor," *IEEE JSSC,* vol. 26, no. 2, Feb. 1991, pp. 107–115.
37. Yoshihisha Iwata, Masaki Momodomi, Tomoharu Tanaka, Hideko Oodaira, Yasuo Itoh, Ryozo Makayama, Ryouhei Kirisawa, Seiichi Aritome, Tetsuo Endoh, Riichiro Shirota, Kazunori Ohuchi, and Fujio Masuoka, "A high-density NAND EEPROM with block-page programming for microcomputer applications," *IEEE JSSC,* vol. 25, no. 2, Apr. 1990, pp. 417–424.
38. Sateh M. S. Jalaledine and Louis G. Johnson, "Associative IC memories with relational search and nearest-match capabilities," *IEEE JSSC,* vol. 27, no. 6, Jun. 1992, pp. 892–900.
39. Sargur N. Srihari, "A special-purpose content addressable memory chip for real-time image processing," *IEEE JSSC,* vol. 27, no. 5, May 1992, pp. 737–744.
40. R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. Sangiovanni-Vincentelli, *ESPRESSO-IIC: Logic Minimization Algorithms for VLSI Synthesis,* The Netherlands: Kluwer Academic, 1984.
41. Gerard M. Blair, "PLA design for single-clock CMOS," *IEEE JSSC,* vol. 27, no. 8, Aug. 1992, pp. 1211–1213.
42. R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang, "MIS: a multiple-level logic optimization system," *IEEE Transactions on CAD of Integrated Circuits and Systems,* vol. CAD-6, no. 6, Nov. 1987, pp. 1062–1081.
43. James J. Cherry, "CAD programming in an object oriented programming environment," in VLSI CAD Tools and Applications (Wolfgang Fichtner and Martin Morf, eds.), Boston, Mass.: Kluwer Academic, 1987, Chapter 9.
44. C. Sechen and A. Sangiovanni-Vincentelli, "TimberWolf3.2: a new standard cell placement and global routing package," *Proceedings of the 23rd Design Automation Conference,* Las Vegas, Nev., 1986, pp. 432–439.
45. Qiao Tong and Niraj K. Jha, "Design of C-Testable DCVS Binary Array Dividers," *IEEE JSSC,* vol. 26, no. 2, Feb. 1991, pp. 134–141.

Micron Ex. 1037, p. 665
Micron v. YMTC
IPR2025-00119

CMOS SYSTEM CASE STUDIES

PART 3

Part 3 comprises three case studies of CMOS chips or modules. They are included as examples to show the use of sub-systems introduced in Chapter 8 and Chapter 5. In addition, the designs were chosen to illustrate three different levels of design.

The first example describes an embeddable RISC microcontroller. This was implemented as a custom chip but could be implemented as a standard cell, gate array, or even an FPGA. The design emphasis here is from the architectural level down to the gate level.

In the second example, describing a television ghost cancellation chip, the architecture is also important but the detailed circuit and layout design have a huge impact on the commercial viability of the chip. The emphasis extends from the architectural level to the circuit level.

The final example illustrates a simple analog-to-digital converter where an individual inverter is the ultimate focus of attention.

The three case studies represent decreasing logic complexity and increasing emphasis on circuit design. As the complexity of a given CMOS system increases, the ability to individually address individual transistors, gates and sub-systems decreases. This trend is illustrated by these examples.

Micron Ex. 1037, p. 666
Micron v. YMTC
IPR2025-00119

CMOS SYSTEM DESIGN EXAMPLES

9

9.1 Introduction

Many times VLSI design mirrors board-level system design, where standard components such as ALUs, memories, and logic gates are combined to form a specified function. This reality is reflected in CMOS standard-cell or gate-array libraries. However, the VLSI medium affords the designer the possibility of creating new components that break the barriers created by packaging. This might be a section of logic, phase-locked to a lower external clock, that operates at extremely high speed. Or it might be a special memory structure that merges logic and the structure of an algorithm to meet a speed, power, or cost objective.

In the first edition of this book, this chapter contained a number of examples that illustrated this principle. Implemented in today's technology many of these examples would form small components of much larger chips.

In this chapter, three examples are given that illustrate how the components developed in Chapter 8 and previous chapters are used in larger systems. The first is a contemporary high-speed RISC microcontroller that may be used for a variety of high-speed DSP applications. It provides an illustration of the flow of constraints from high-level decisions to the low-level implementation that results. The example also provides an example of three kinds of CMOS layout—datapath, memory, and control logic. While the implementation style used in this example is symbolic custom layout, the

Micron Ex. 1037, p. 667
Micron v. YMTC
IPR2025-00119

design could also be implemented (at an area and perhaps performance cost) as a standard-cell, sea-of-gates design or even an FPGA.

The second example might be thought of as a "classical" regularly structured CMOS VLSI design, in which one basic core cell is replicated many times to form the major portion of the chip. Many times such structures find application in high-speed digital-signal processing applications, such as video filtering and image processing.

The final example bridges the analog/digital gap by describing a simple 6-bit flash A/D converter. Each example demonstrates increasing emphasis on the detail of circuit and layout.

9.2 A Core RISC Microcontroller †

The first example presented is a RISC microcontroller that was designed as part of a much larger image processing chip. A block diagram of the processor is shown in Fig. 9.1. The processor had an on-chip instruction RAM and connected via a system address and data bus to the rest of the chip and special function units. As it is shown in Fig. 9.1, the processor is typical of

†This processor was designed by B. Edwards, C. Terman and N. Weste of TLW.

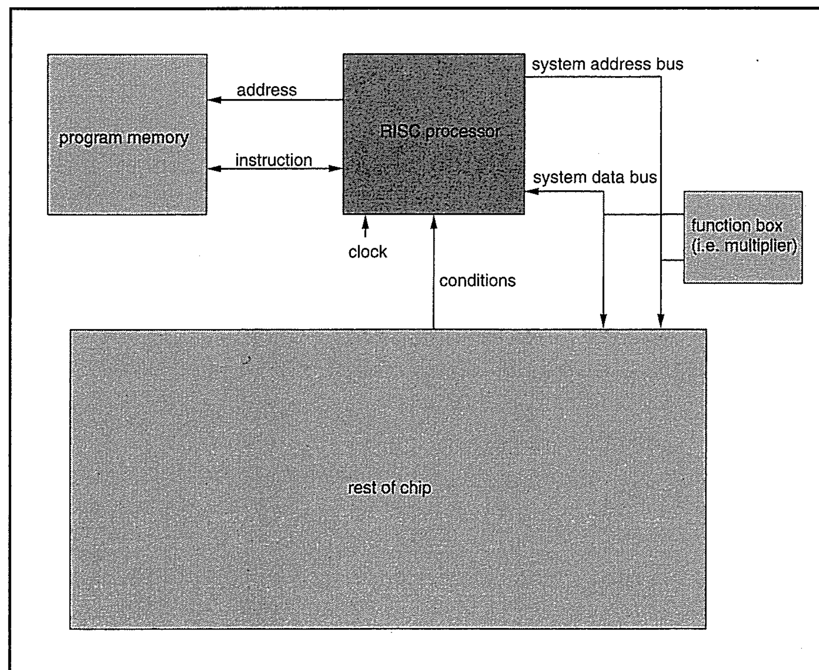


FIGURE 9.1 RISC microcontroller system use block diagram

Micron Ex. 1037, p. 668
Micron v. YMTC
IPR2025-00119

embedded programmable cores which provide control for other dedicated processing elements.

The 16-bit processor had to run at 40MHz and affords a good example of an embedded processor that can be used for a wide variety of control and signal-processing applications. While a little more complicated than a minimal microcontroller, the concepts demonstrated in this example can be applied to a wide range of CMOS VLSI system problems.

We begin with the design of the instruction set, which in turn must be able to implement the operations required of the processor. We follow with a description of the pipeline architecture or the arrangement of memory and logic to enable the implementation of the instruction set in the required cycle time. The major logic blocks are then summarized. The layout of these blocks is then summarized. Finally, the methods of testing and verifying the processor are described.

9.2.1 Instruction Set

The instruction set defines what basic operations are possible with the processor and forms a high-level specification for the processor. The instructions in this instance are divided into two groups, namely,

- the control-transfer class.
- the ALU class.

The control-transfer class includes jump and call instructions. The ALU class includes arithmetic and logic operations. Other types of instructions might include operating system instructions (for a full microprocessor) and specialized I/O instructions (say, for a graphics accelerator).

9.2.1.1 Address Architecture

At this point there is a decision to make concerning the type of address architecture that the microcontroller is to implement. Options include a stack-address architecture, an accumulator architecture, and a multiple-register (two- or three-register) address architecture. Each dictates a particular register architecture. Figure 9.2 indicates some of the possibilities.

An accumulator architecture has a special register associated with the ALU that holds the intermediate results of computation (Fig. 9.2a). The computation for

```
c = a + b
```

would be

```
load a
plus b
store c
```

Micron Ex. 1037, p. 669
Micron v. YMTC
IPR2025-00119

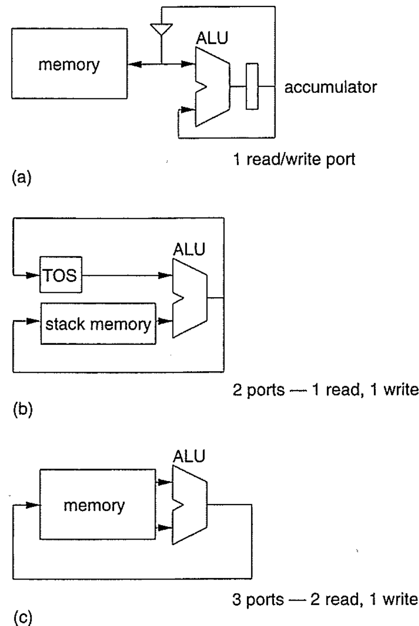


Figure 9.2 Alternative processor address architectures

In this example the register file has to be able to do one read or write at a time. At any one time only a single port is required, at the expense of taking three clock cycles to complete the add. However, a single-port register file can be implemented with a six-transistor static RAM cell, the smallest all-transistor static memory that can be implemented.

In a stack architecture (Fig. 9.2b), the register file implements a stack. The ALU uses the stack and a special register called the top-of-stack (TOS) to perform arithmetic operations. For instance, to complete the operation

```
c = a + b;
```

the following operations would be completed:

```
push a    pushes a onto stack (TOS=a)
push b    pushes b onto stack (TOS=b)
pop add   pops a and b, pushes a+b onto stack (TOS = a+b = c)
```

This requires that the register be read and written in a single cycle as the pop operation reads (a) and writes (c= a+b). The static RAM cell shown in Figs. 8.53 and 8.65 can achieve this by single-ended sensing of the bit lines and differential driving for write operations. This RAM cell is not much larger than the RAM cell that would be used in the stack architecture. Two decoders are needed, so the overall register file would be slightly larger than for the stack case. Although the add operation takes three cycles in the example above, in general as one operand may already be on the top of the stack, the add can be completed in two cycles.

Micron Ex. 1037, p. 670
Micron v. YMTC
IPR2025-00119

A three-address architecture is what most modern RISC processors use (Fig. 9.2c). It completes the operation

```
c = a + b
```

(and most other operations) in one cycle. To do this requires a three-ported register file that can independently read two operands and write a third. This register file can be implemented using the structures shown in Fig. 8.65 (a regular RAM cell with multiple ports) or Fig. 8.66 (a register file). This register file is larger than the static RAMs for the accumulator and stack architectures due to the increase in memory-cell size and the three address-decoders that are required, but the architecture is potentially two or three times faster (for various reasons this speed increase is not always reflected in real programs).

The size (in words) of the register file has to be estimated. This is related to the maximum number of intermediate results that need to be held at one time. This might be estimated by implementing the chip function in a high-level language (C, FORTRAN, Pascal, Lisp) first and using this as a basis for estimation. In this case, 128 registers were dictated by the system architects.

At this point the designer might assess the areas of each type of register structure (if area was of importance) by completing a layout for each memory cell and decoder or, in the case of a gate-array or standard-cell design, finding the most appropriate register architecture. In the case at hand, the three-port address register architecture is chosen for the following reasons:

- Potentially provides the fastest architecture.
- Easy to program.
- Increase in area deemed not important (a custom layout is assumed).
- No complicated clocking required (minimizes design time).

An equal address space was allowed for random "external registers," yielding an 8-bit address field for reads and writes. A 1 in the MSB of the read or write address indicates a read or a write to a register on an external bus. This provides for a register-mapped I/O space for communication with external devices through the implementation of extra hardware if required.

9.2.1.2 ALU Class Instructions

With the register architecture fixed, an encoding for the instruction set may be proposed (Hex numbers are used). Because there are three addresses of 8-bits required, the following encoding for the ALU class was used, resulting in a 32-bit instruction. In the following description,

`WR` (write address) is one of
`00-7F` register file address

Micron Ex. 1037, p. 671
Micron v. YMTC
IPR2025-00119

80-FF external interface address
RA (read port A address) is one of
00-7F register file address
80-FF external interface address
RB (read port B address) is one of
00-7F register file address
80-FF external interface address
OP is the op-code for the instruction.
IT is the instruction type.

The first type of instruction is used for general arithmetic and logic operations.

IT = 1 three-address arithmetic instruction

#bits = 2	6	8	8	8
IT=1	OP	WR	RA	RB

The general operation is as follows:

```
WR = RA op RB
```

So for instance, a specific operation might include

```
WR[10] = RA[5] + RB[4]
```

In the instruction above, location 10 in the register file is replaced with the sum of locations 5 and 4.

A two-address literal instruction is provided as follows:

IT = 2 two-address with sign-extended 8-bit literal

IT=2	OP	WR	RA	LITERAL

The general operation is as follows:

```
WR = RA op LITERAL
```

A specific operation might be

```
WR[100] = RA[20] + 24
```

In the instruction above, location 100 in the register file is replaced with the sum of location 10 and the constant 24.

A single-address literal instruction is also provided. This allows a 16-bit literal to be loaded into the register file.

IT = 3 one-address with sign-extended 12-bit literal

IT=3	OP	WR	LITERAL

Micron Ex. 1037, p. 672
Micron v. YMTC
IPR2025-00119

The general operation is as follows:

```
WR = op LITERAL (B is undefined)
```

A specific operation might be

```
WR[100] = -1 (#xFFFF)
```

that is, placing the constant FFFF in location 100.

The opcode, OP, is defined as follows

```
00      A+B              04    A-B-1
01      A+B+1            05    A-B
02      A                06    A-1
03      A+1              07    A
08-0F   Undefined
10      all zeros        14    A and notB
11      A and B          15    A
12      notA and B       16    A xor B
13      B                17    A or B
18      notA and notB    1C    notB
19      A xnor B         1D    A or notB
1A      notA             1E    A nand B
1B      notA or B        1F    all ones
20      logical left shift by SHIFTR
21-2F   logical left shift by 1 to 15 bits
30      arithmetic right shift by SHIFTR
31-3F   arithmetic right shift by 1 to 15 bits
```

SHIFTR is a special register (external address = 21) that allows a shift amount to be externally specified.

Although these assignments may seem random, they are linked to the implementation. For instance, bit 0 of the ALU opcode is the carry-in to the adder in the ALU, bit 1 forces the B bus to zero, and bit 3 inverts the B bus. Similarly, bits 3–0 are used directly by the Boolean logic to implement the functions outlined above (see Section 9.2.3.1). These assignments are used to eliminate control logic.

9.2.1.3 Control Transfer Instructions

The control transfer instructions implement jumps, call, and return. They are defined as follows:

IT=0	OP 6 bits	COND 8 bits	JA 12 bits

where

```
OP=20 Jump True
OP=22 Jump False
```

Micron Ex. 1037, p. 673
Micron v. YMTC
IPR2025-00119

```
OP=10 Call
OP=08 Return
```

and

COND defines a condition code as follows:

```
00      ALU result negative
01      ALU result zero
02      Adder result had carry
03      ALU result was negative or zero
04      Boolean/Shifter result zero
05-0F   <Illegal>
1F-34   External conditions selectable by multiplexer
3F      True
40-FF   <Illegal>
```

JA specifies a 12-bit jump or call address. In the case of a CALL, the current program counter (PC) is pushed onto a stack, while a RETURN pops the PC from the stack.

9.2.2 Pipeline Architecture

The design was partitioned into six major blocks (Fig. 9.3a). The ALU_DP is responsible for performing arithmetic and logic operations. It also contains an interface to an external system-address and data bus. The register file is responsible for providing operands for the ALU_DP and storing the results of ALU operations. The PC (Program Counter) data path is responsible for calculating the next program counter value. It therefore has to deal with JUMPs, CALLs, and RETURNs. For the latter instructions, it implements an eight-deep stack. The Instruction Pipe datapath stores the instruction for a number of pipeline stages and performs comparisons to permit pass-around (see Section 9.2.2.1). The instruction RAM provides instructions for the processor and the control section provides for instruction decode and various other control operations.

The ALU_DP, register file, instruction pipe, and PC datapath were constructed using datapath techniques although they could be implemented as standard cells or sea-of-gates structures. A high-speed static RAM was used for program storage. A separate write port was provided to load the program RAM. A single control block was used to control all sections.

The operation of the microcontroller is as follows:

1. The PC presents an address to the instruction RAM, which in turn looks up an instruction to be applied to the machine. This includes the opcode for the ALU and the addresses to the register file.
2. The register file accesses the operands addressed by the instruction

Micron Ex. 1037, p. 674
Micron v. YMTC
IPR2025-00119

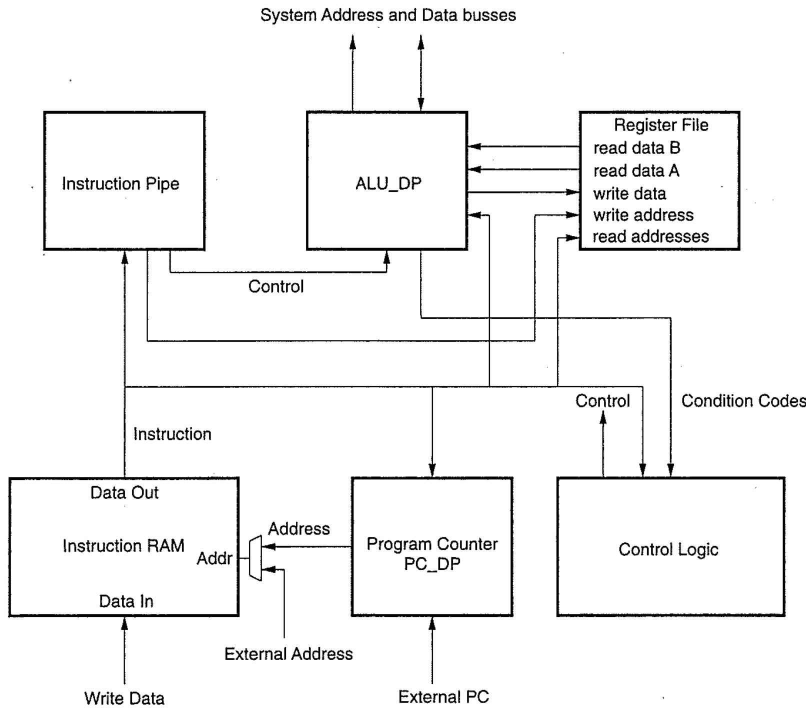


FIGURE 9.3 Processor-block diagram: (a) overview; (b) schematic

(either from the register file or external registers), and places them on the ALU input busses.

3. The ALU calculates the result including condition flags such as carry and zero.
4. The result is written to the register file or external registers.

If all of these operations were placed in one cycle, the cycle could be quite long. For instance, consider the following representative times:

Instruction RAM access	15 *ns*
Register file read	10 *ns*
ALU operation	15 *ns*
Register write	10 *ns*

which results in a total cycle time of around 50 *ns*. This exceeds the desired cycle time by more than a factor of two. Fortunately, this problem may be solved by the use of pipelining.

The microcontroller may be conveniently pipelined according to the steps outlined above, calling the stages the I, R, E, and W stages for Instruction fetch, register Read, Execute, and Write operations. One may of course choose to pipeline the machine differently; this scheme is chosen based on

Micron Ex. 1037, p. 675
Micron v. YMTC
IPR2025-00119

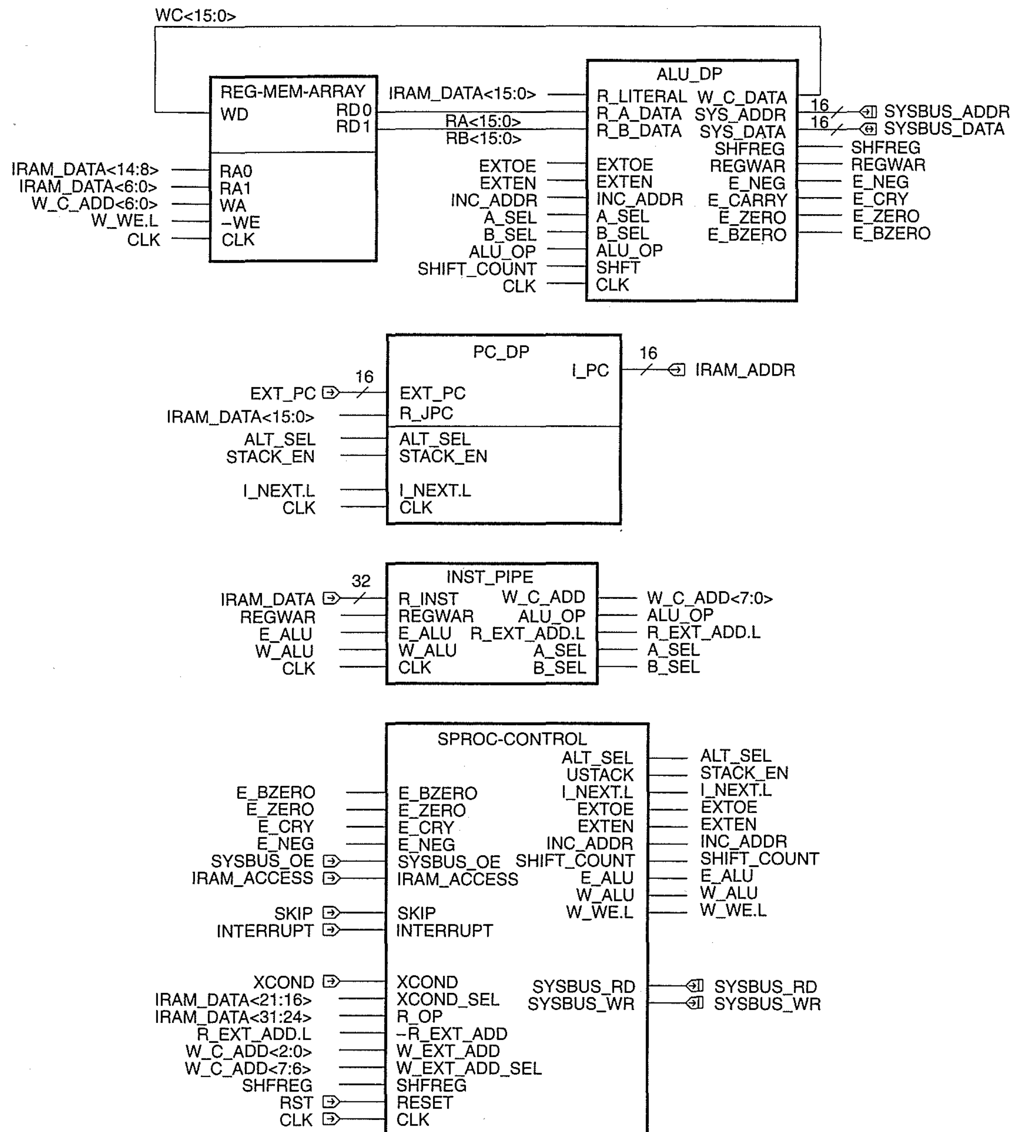


Figure 9.3 *(continued)*

the fact that the microcontroller has to run at 40 MHz and on some experience of how fast modules will run. In practice, a number of pipeline schemes might be explored, with the one that meets the speed requirement with the least design effort selected.

In the signal nomenclature, I_, R_, E_, or W_ preceding a name indi-

cates that the data is valid in that pipeline stage. The pipeline operation can be visualized using the following pipeline diagram:

Cycle	I stage	R stage	E stage	W stage
0	Inst 1			
1	Inst 2	Inst 1		
2	Inst 3	Inst 2	Inst 1	
3	Inst 4	Inst 3	Inst 2	Inst 1
4	Inst 5	Inst 4	Inst 3	Inst 2

In cycle 0, instruction 1 is in the I stage, having been fetched from the instruction RAM. In cycle 1, it moves to the R stage and is used to address the register file to access the read operands in the instruction (RA and RB). Instruction 2 enters the I stage. Instruction 1 enters the E-stage in cycle 2, where it presents the opcode to the ALU. The read operands have also been fetched from the register file. Instruction 3 enters the I stage, and Instruction 2 moves to the R stage. Finally, in cycle 3 Instruction 1 enters the W stage, where the WR address is used to write the result of the ALU operation into the register file.

9.2.2.1 Bypassing, Result Forwarding, or Pass-around

In the above pipeline diagram it may be seen that the operands for instruction 2 and 3 have been read by the time that the result of instruction 1 is written back to the register file. This requires that the W stage write data be forwarded to the R stage or E stage if these stages require the data that is being written to the register file. For instance, if the result of instruction 1 were used in instruction 2, then the data being written to the register file in cycle 3 would also have to be passed to the E stage for instruction 2. This is done by comparing the operand addresses in the R stage and E stage with those in the W stage and controlling a set of multiplexers that feed the appropriate operand to the ALU.

Consider the following code fragment:

```
ADD A,B,C     Add A, B and place in C
SUB D,C,F     Subtract C from D and place in F
XXX
YYY
```

The following pipeline diagram represents the code sequence:

Cycle	I stage	R stage	E stage	W stage
0	ADD A,B,C			
1	SUB D,C,F	ADD A,B,C		
2	XXX	SUB D,C,F	ADD A,B,C	
3	YYY	XXX	SUB D,C,F	ADD A,B,C

Micron Ex. 1037, p. 677
Micron v. YMTC
IPR2025-00119

In cycle 3, the result of adding *A* and *B* is being written to location *C*. However, *C* is required in the E stage to compute *D* – *C*. Thus the normal read path from the register file has to be bypassed to allow the current (W stage) value of *C* to be passed to the ALU.

9.2.2.2 Conditional Branching

The condition code (if coming from the ALU) is calculated late in the E stage. Any jump occurs in the cycle after the condition is calculated and, due to the pipelining, the instruction after a jump is executed.

Consider the following code sequence that implements a branch based on the result of an ADD instruction:

```
     ADD A,B,C          Add A, B and place in C
     JMP ZERO,FOO       Jump to location FOO if the result of
                        A+B(C)=0
     SUB E,F,G          Subtract F from E and place in G
     AND X,Y,Z          AND X and Y and place in Z
FOO: OR P,Q,R           Branch Target FOO
     ADD A,R,C
```

The following pipeline diagram represents the code sequence where the branch is not taken:

Cycle	I stage	R stage	E stage	W stage
0	SUB E,F,G	JMP ZERO	ADD C,A,B	
1	AND D,G,H	SUB E,F,G	JMP ZERO,FOO	ADD C,A,B
2		AND D,G,H	SUB E,F,G	JMP ZERO
3			AND D,G,H	SUB E,F,G

In cycle 0 the ADD instruction is executed and the ZERO condition is calculated. The JMP instruction is executed in cycle 1, the condition being registered to the W stage. In cycle 2 the SUB instruction is unconditionally executed, and in cycle 3 the AND instruction is executed.

The following pipeline diagram represents the code sequence where the branch is taken:

Cycle	I stage	R stage	E stage	W stage
0	SUB E,F,G	JMP ZERO	ADD C,A,B	
1	OR P,Q,R	SUB E,F,G	JMP ZERO,FOO	ADD C,A,B
2	ADD A,R,C	OR P,Q,R	SUB E,F,G	JMP ZERO
3		ADD A,R,C	OR P,Q,R	SUB E,F,G

Micron Ex. 1037, p. 678
Micron v. YMTC
IPR2025-00119

The sequence here is the same except that in cycle 1 the OR instruction moves to the I stage. In cycle 3 this instruction is executed. Note that the SUB instruction is still executed in cycle 2.

9.2.2.3 Subroutine Call and Return

Consider the following code sequence, which demonstrates a Call and Return Sequence:

```
      ADD A,B,C
      CALL FOO
      SUB E,F,G
BAZ:  XXX
      YYY
      ...
FOO:  ADD J,K,L
      RETURN
      SUB X,Y,Z
```

The execution of this sequence is shown below:

Cycle	I stage	R stage	E stage	W stage
0	SUB E,F,G	CALL FOO	ADD A,B,C	
1	ADD J,K,L	SUB E,F,G	CALL FOO	ADD A,B,C
2	RETURN	ADD J,K,L	SUB E,F,G	CALL FOO
3	SUB X,Y,Z	RETURN	ADD J,K,L	SUB E,F,G
4	XXX	SUB X,Y,Z	RETURN	ADD J,K,L
5	YYY	XXX	SUB X,Y,Z	RETURN
6		YYY	XXX	SUB X,Y,Z
7			YYY	XXX

As with the JUMP instruction, the instructions after the CALL and RETURN instructions are also executed.

9.2.2.4 I/O Architecture

The I/O architecture in this example is a condensed version of the real I/O architecture to keep the example simple. Five registers are provided in the I/O space of the processor (seven are listed below but some are read/write or otherwise utilize the same physical register). They are as follows:

- SBRAR (address = 0) Sysbus Read Address Register. When this register is written, it causes a read on the system bus. If an SBRAR is written

Micron Ex. 1037, p. 679
Micron v. YMTC
IPR2025-00119

on cycle i, then on cycle $i + 1$, the SBRAR address is output to the SYS-ADDR bus. On cycle $i + 2$, the data is returned on the SYS-DATA lines. On cycle $i + 3$, the data may be used in the ALU. Thus a code segment to read data from the system bus and increment it might be

```
WR[20] = RA[80](SBRAR)
XX
YY
WR[20] = RA[20] + 1
```

- SBWAR (address = 81 (hex)) Sysbus Write Address Register. This register holds the next address of a Sysbus write. It uses the same register as SBRAR.
- SBWDR (WR = 82) Sysbus Write Data Register. When this register is written, it causes a write on the system bus.
- SBWDR-INC (address = 83) This causes an autoincrement in the SBWAR register, which provides for a simple DMA capability.
- SBRDR (address = 84-read-only) This register contains the data from the last system bus read.
- REGWAR (address = 85) This register is used as a register-file write address in ALU class instructions when WR = C0 – DF.
- SHFREG (address = 86) This register allows a shift amount to be specified for the shifter when the ALU-OP is 20 or 30.

9.2.3 Major Logic Blocks

With the instruction set defined, and address and pipeline architectures decided, the next step is to define logic and storage elements that will realize these architectures. Just how one translates architecture into an RTL design varies. It usually requires a stepwise refinement process where a design is proposed, simulated, and modified to correspond more closely to the required behavior. Most often, previous experience will aid in determining good directions. It is a skill that improves as more designs are completed.

In this section, an RTL design will be presented in schematic form as a finished design. Because the style of processor design is fairly generic, it is hoped that this example will provide readers with a starting point for their own designs. The complete schematic for design for the controller core (sans instruction memory and I/O devices) is shown in detail in Fig. 9.3(b).

9.2.3.1 ALU_DP

The ALU_DP module is responsible for computing collecting operands, collecting the results, and interfacing with external modules. It is divided into

Micron Ex. 1037, p. 680
Micron v. YMTC
IPR2025-00119

three main sections:

- The I/O-REGS.
- The ALU proper.
- The EXT-BUS-DP.

The module is shown in Fig. 9.4.

The inputs are:

R_A_DATA and *R_B_DATA* busses—These are the read data busses from the register file.

R_LITERAL—Literal bus from the instruction RAM.

A_SEL and *B_SEL*—A and B operand select control.

ALU_OP—The ALU op-code.

SHFT—The shift amount for the shifter.

EXTOE, EXTEN controls—Various controls to the EXT-BUS-DP module for loading registers and tristating busses.

INC_ADDR—An increment control for the autoincrementing address register (SBWDR_INC) in the external-bus module.

CLK—The clock.

The outputs are:

SYS_ADDR (system address bus)—Used to address external modules.

SYS_DATA (system data bus)—Used to transfer data to and from external modules.

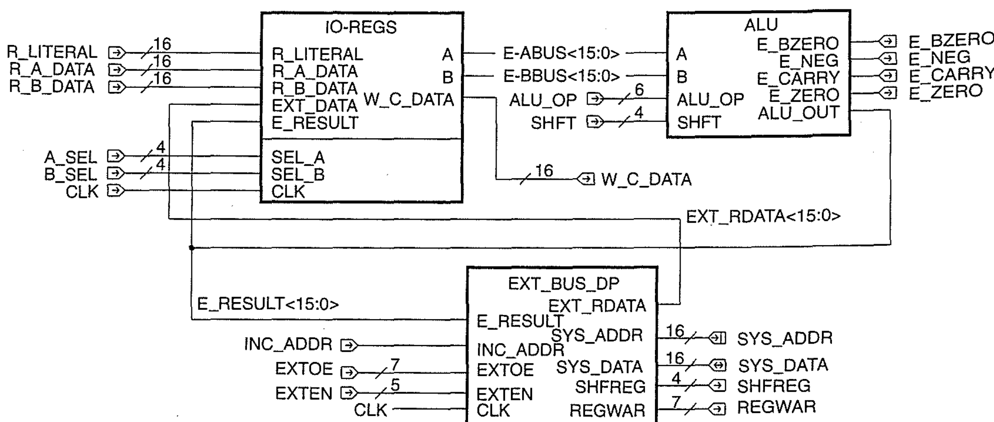


FIGURE 9.4 ALU_DP-block diagram

Micron Ex. 1037, p. 681
Micron v. YMTC
IPR2025-00119

W_C_DATA—Write data bus to the register file.

SHFREG —Shifts amount to control block.

REGWAR—Alternates register write address to INST_PIPE.

CONDITION-CODES—Flags from ALU operation such as ZERO (*E_BZERO and E_ZERO*), NEGATIVE (*E_NEG*), and CARRY (*E_CARRY*).

First we will deal with the two modules that surround the ALU.

9.2.3.1.1 IO-REGS

The IO-REGS are responsible for providing the appropriate operands to the ALU. The A and B port of the ALU have identical structures (IO-REG) that provide operands from:

- the register file.
- the literal bus (from the instruction RAM).
- W-stage data (for pass-around) and E-stage data.
- external data from the external bus (*EXT_DATA*).

The IO-REGS circuit is shown in Fig. 9.5 and the IO-REG subblock is shown in Fig. 9.6. The modules are composed of multiplexers, registers, and

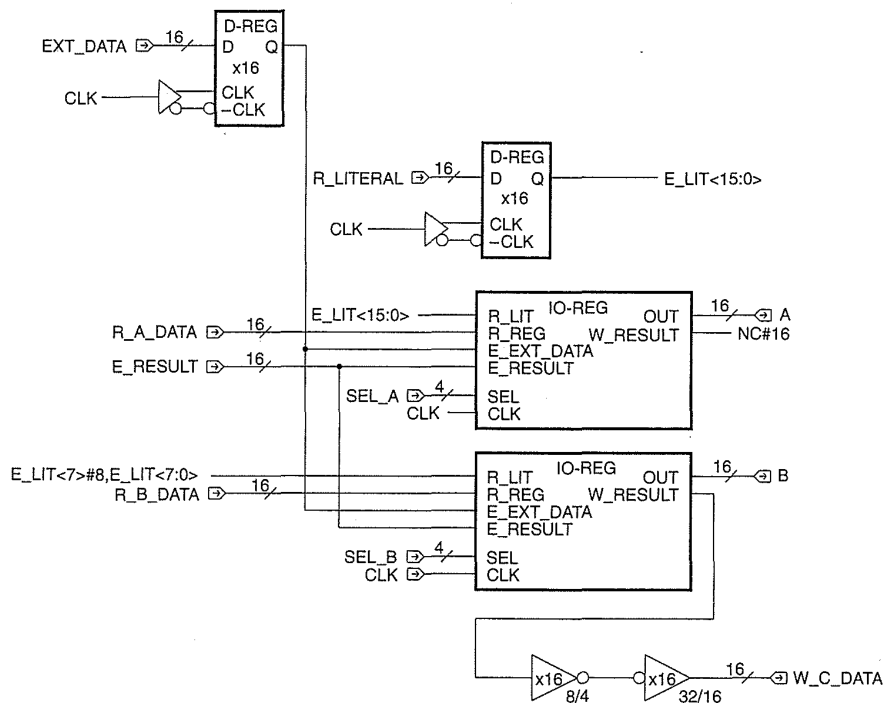


Figure 9.5 IO-REGS module schematic

Micron Ex. 1037, p. 682
Micron v. YMTC
IPR2025-00119

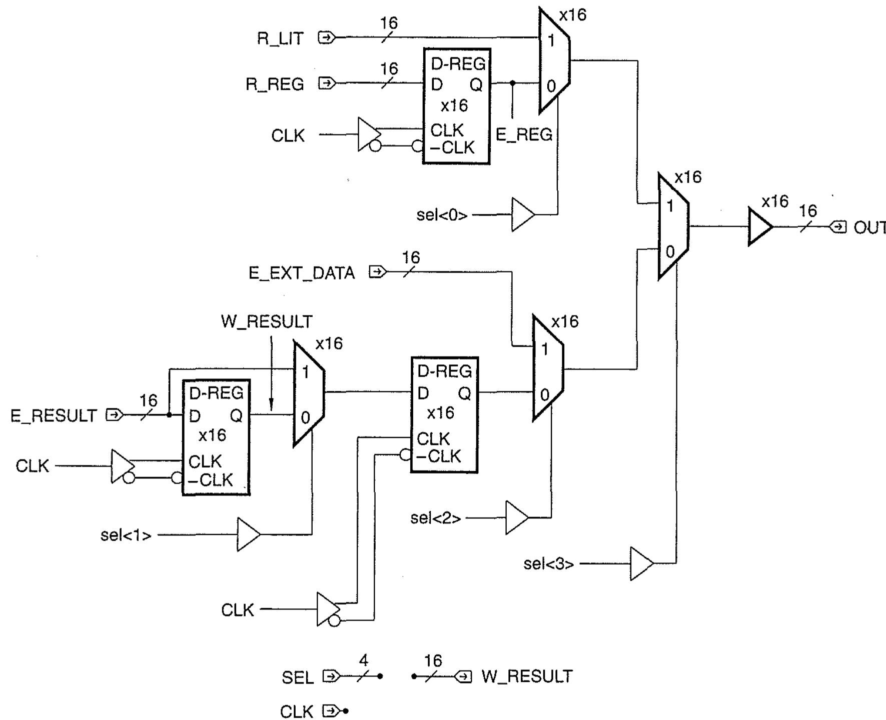


FIGURE 9.6 IO-REG module schematic

buffers. As a single clock is being used, the register structure in Fig. 9.25(a) was chosen for the following reasons:

- Static (A habit; dynamic registers could be used but if area is not that affected, static allows the clock to be stopped and the static registers are not as susceptible to noise). Static registers also allow IDDQ testing (see Chapter 7).
- Small number of transistors.

In Fig. 9.5 the literal field from the instruction RAM, *R_LITERAL<15:0>*, is registered to create the E-stage version of this data, *E_LIT<15:0>*. Note that port A receives the full 16-bit field of the literal, while the B port receives an 8-bit sign-extended version (*E_LIT<7>#7, E_LIT<7:0>*). The register-file write data is taken from the B IO-REG module. A further register stores the *EXT_DATA* bus. In the IO-REG module (Fig. 9.6), a register stores the R-stage data from the register file. The decision to place this register here rather than in the register file relates to the difference in critical path between the ALU and the read access of the register file. In this case, the relatively long delay of driving a bus from the register file to the ALU is placed in the R stage. If the register were placed in the register file, this delay would be in the E stage (because it would be added to the

Micron Ex. 1037, p. 683
Micron v. YMTC
IPR2025-00119

clock-to-Q delay of the register). Because the register-file read time was shorter than the ALU critical path, this placement was used. Two other registers store the ALU result (*E_RESULT*) and the subsequent *W_RESULT*. The latter represents the data that will be stored in the register file in the W stage. A large buffer drives the A or B data to the ALU. There is some replication in registers and muxes between the two IO-REG sections. This was done for layout regularity and bus optimization reasons.

9.2.3.1.2 EXT-BUS-DP

The EXT-BUS-DP (Fig. 9.7) provides an interface between the external system bus and the processor. A write-data register (SBWDR) is provided to

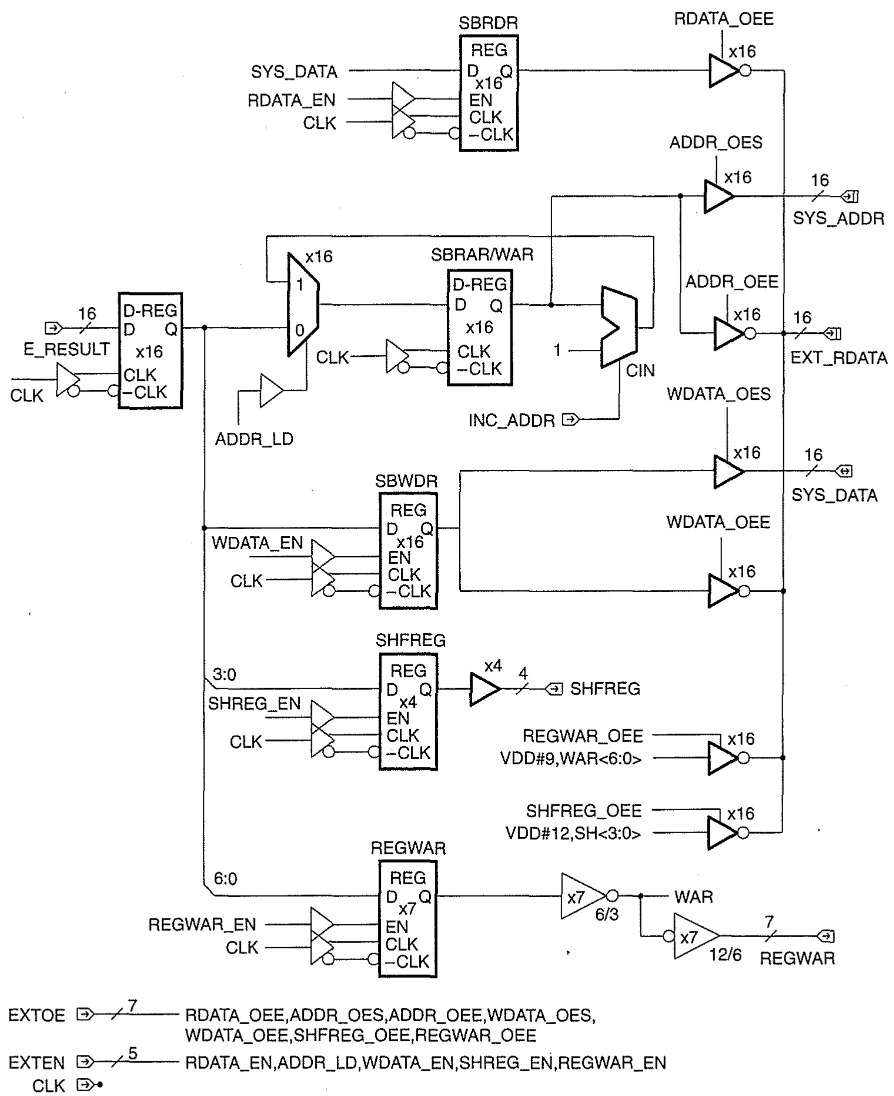


Figure 9.7 EXT-BUS-DP module schematic

Micron Ex. 1037, p. 684
Micron v. YMTC
IPR2025-00119

which data may be written from the ALU to the *SYS_DATA* bus. An address register (SBRAR/WAR) is provided to supply addresses to the system address bus (*SYS_ADDR*). An incrementer is provided with the address register to aid in simple DMA-type operations. The *SYS_DATA* may also be registered (SBRDR) for reads from the *SYS_DATA* bus. These registers may be tristated onto the *EXT_RDATA* bus, which is an input of the IO-REGS module. The SHFREG stores a shift amount that may be used by the shifter. In the actual chip, this module had a few extra registers for other special operations. As mentioned previously, REGWAR provides a write address for the register file for ALU operations with addresses in the range C0–DF.

9.2.3.1.3 ALU

The ALU is divided into three subsections, namely,

- the adder.
- the Boolean unit.
- the shifter.

While it would be possible to merge the Boolean unit with the adder (à la the 181 ALU), the designers happened to like this partitioning because the instruction decode is simple and the modularity allows the Boolean and shifter to be dropped if those blocks are not required. Also a range of adders may be used to achieve different size and speed requirements. As shown in Fig. 9.8, each functional unit takes its inputs from the *A* and *B* buses and conditionally tristates the result onto the *ALU_OUT* bus.

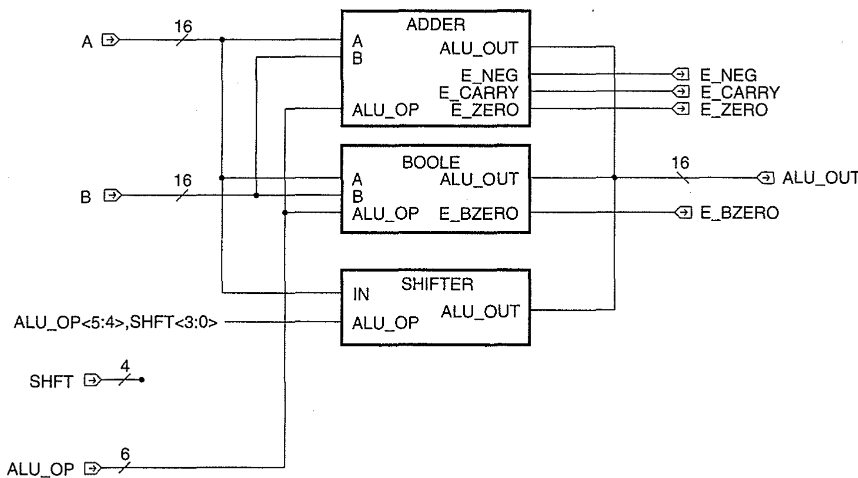


FIGURE 9.8 ALU module schematic

Micron Ex. 1037, p. 685
Micron v. YMTC
IPR2025-00119

The adder module chosen for this design is the static Manchester carry stage shown in Fig. 8.19(a). The reasons are as follows:

- It is a static design requiring no clocking (straightforward design).
- It could be designed to have a very small pitch, which is important in keeping datapath height small and delays low.
- It had been used in previous designs successfully.
- It provides a better speed trade-off for the adder (simpler PG and SUM generators—see Figs. 8.18 and 8.34).

The adder module is designed in 4-bit sections and consists of a PG generator, a carry lookahead section, and a SUM generator. Figure 9.9 shows a schematic of a single-bit section that also represents the physical layout—i.e., the carry section is sandwiched between the PG generator on the left and the SUM logic on the right. Figure 9.10 shows the schematic for a 4-bit section. Complementary CMOS logic is used throughout the adder except for the carry lookahead gate, which is a pseudo-nMOS NAND gate. Variations of this adder might include the pseudo-nMOS carry-lookahead gate shown in Fig. 8.17. Other adders that might be used if the speed is not too stringent might include the adder shown in Fig. 8.7(b). This is potentially the smallest adder that might be used, and the speed can be altered somewhat by adjusting the size of the CARRY gate transistors. This adder, used as a carry-select adder, might also prove useful. If in doubt, the designer might complete some initial simulations of various adders at this point. If possible, you should actually design the layouts and backannotate the schematics or HDL when simulating because this will give the best indication of the final speed that might be attained. Remember to simulate at the Worst Case Speed corner of the process. Figure 9.11 shows the complete 16-bit adder complete with control circuitry. A zero and negate circuit (called BUS-OP in this

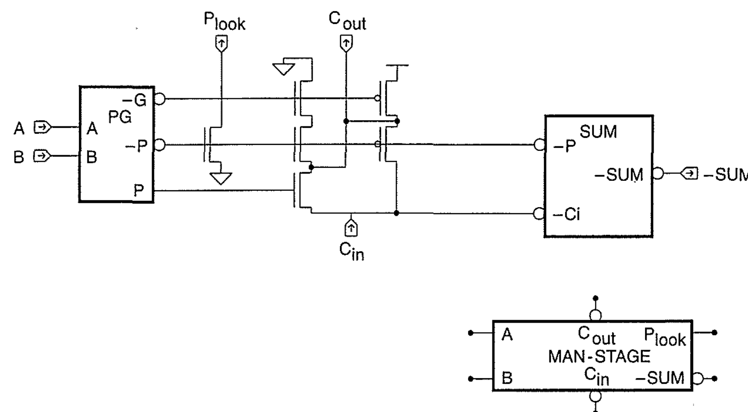


Figure 9.9 Manchester stage used in adder

Micron Ex. 1037, p. 686
Micron v. YMTC
IPR2025-00119

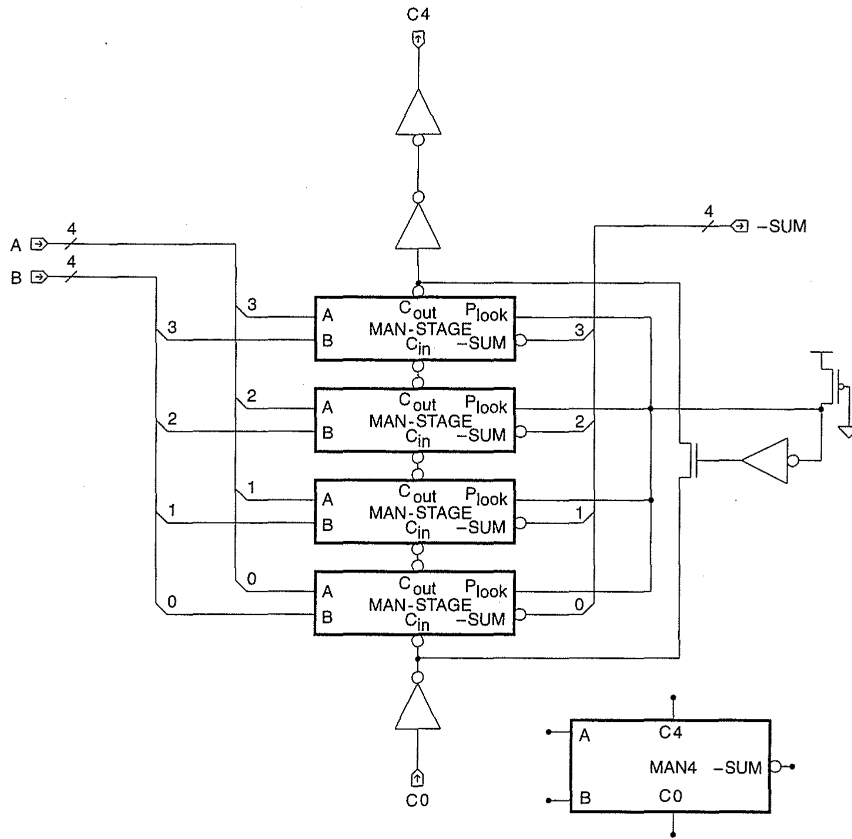


FIGURE 9.10 4-bit Manchester adder

design) is required on one input (B) to zero the operand or set B to all ones (for adding zero, –1 or just passing A) and conditionally inverting the operand for subtraction operations. While this could be implemented using an XOR gate and an AND gate as shown at the bottom of Fig. 9.11, the pass-gate implementation shown in Fig. 9.25(b) is faster and smaller and was used in this design. A zero detect is placed on the output of the adder (*E_ZERO*). It is placed here rather than on the *ALU_OUT* bus in Fig. 9.8 because this implementation is faster by the time it takes the tristate buffers in the adder and Boolean unit to drive that bus. As a result, the zero circuit is replicated in the Boolean unit (*E_BZERO*). The circuit used for this is a pseudo-nMOS NOR gate. Other complementary CMOS solutions could also be used (see Fig. 8.28). The ALU-OP instruction decode is also shown. This is fairly straightforward. For instance, `ALU-OP=5` is `A-B`. Thus *INVERT* = 1, *CIN* = 1, and *ALU-OP*<*5:4*> = 0. This inverts the B operand, adds 1 to the adder via the carry in, and enables the adder tristate buffer.

The module used for the Boolean unit is a transmission-gate circuit based on the structure shown in Fig. 8.33, and is shown in Fig. 9.12. The

Micron Ex. 1037, p. 687
Micron v. YMTC
IPR2025-00119

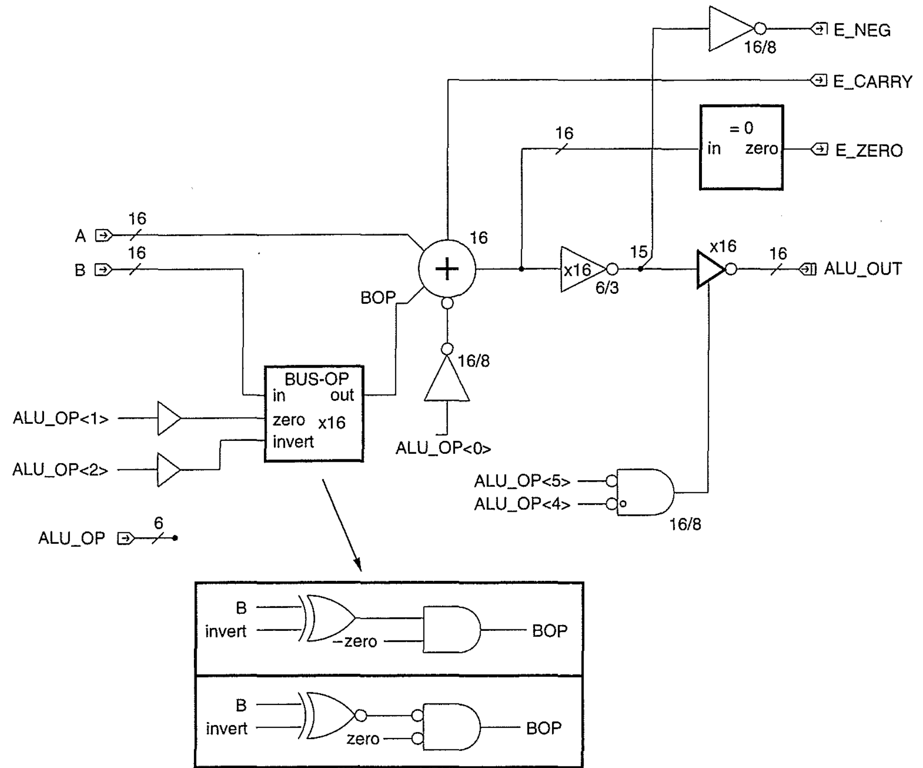


Figure 9.11 16-bit adder schematic

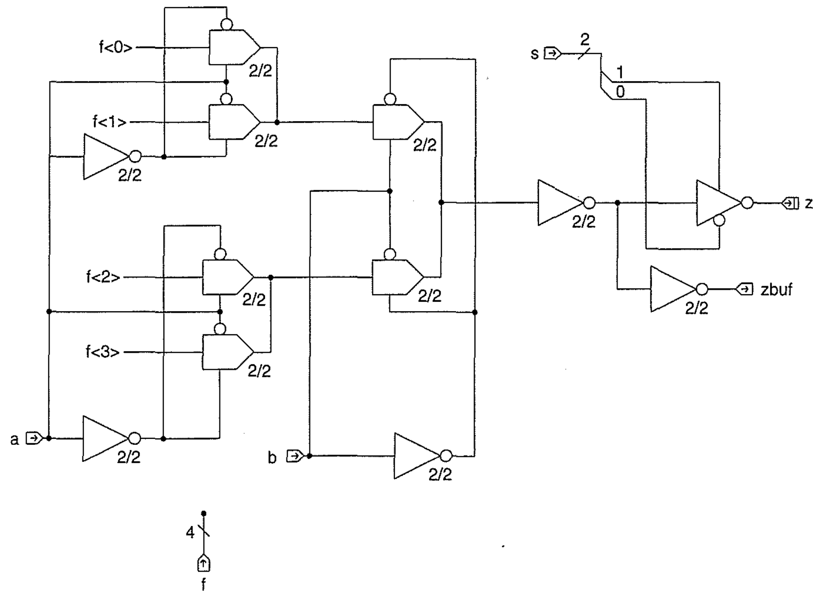


Figure 9.12 Boolean bit

complete Boolean unit is shown in Fig. 9.13 that includes a tristate output buffer, decoding logic, and a zero detect. As an example of a Boolean instruction, consider that ALU-OP=16 is A⊕B. In Fig. 9.12, *f*<*0*>=0, *f*<*1*>=1, *f*<*2*>=1, and *f*<*3*>=0. Thus the truth table is

A	B	OP
0	0	0
1	0	1
0	1	1
1	1	0

which implements the XOR operation. *ALU-OP*<*5:4*>=01 enables the Boolean unit onto the *ALU_OUT* bus.

The shifter is a tristate-buffer multiplexer structure using a left shifter and a right shifter with direct decoding of the shift amount (see Fig. 8.47 with tristate-buffers replacing transmission gate muxes). Tristate buffers were used to achieve the desired speed.

A word about the critical path. In the ALU section, the critical path starts in a register, passes through the IO-REG muxes, through the BUS-OP circuit (conditionally negating and zeroing the operand), through the adder and into the condition code logic where the conditions are registered. The critical path as reported by the timing analyzer is shown below.

Summary of path-clock rising to clock rising delay of 22.7 *ns* due to a delay of 21.8 *ns* at node 3836 and a .9 *ns* setup time into the register:

```
CK to CK 22.7ns (setup time 0.9) data node 3836 at CK + 21.8
 S>sproc>SPROC-CONTROL-1>REG-17>D-REG-MUXSTANDARD
```

The delays have the form:

```
Node name cumulative-delay (this-node-delay)
node-path
```

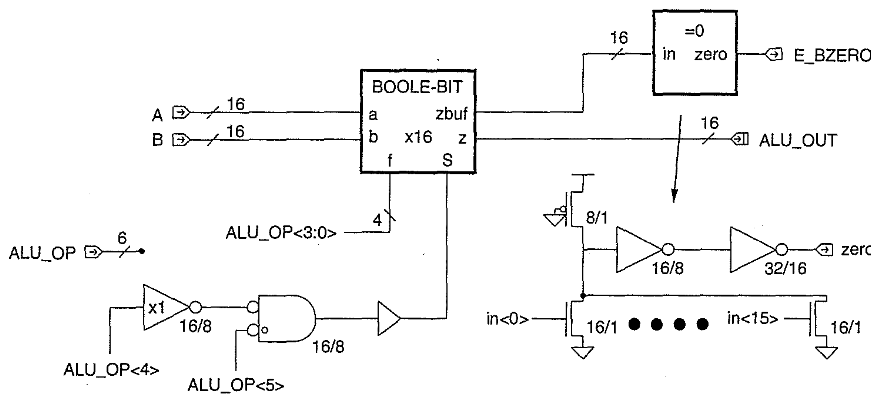


FIGURE 9.13 Boolean unit schematic

Micron Ex. 1037, p. 689
Micron v. YMTC
IPR2025-00119

Clock buffer delay:

```
Node CK 0.0ns (0.9ns)
S>sproc>INST_PIPE-2>CLOCK-BUFFER-3>CG-INVERTER-3>NMOS-4
Node 4329 0.9ns (0.6ns)
S>sproc>INST_PIPE-2>CLOCK-BUFFER-3>CG-INVERTER-4>PMOS-4
```

Clock-to-Q delay of register:

```
Node 4338 1.5ns (1.6ns)
S>sproc>INST_PIPE-2>DP-D-REG-10
```

Instruction Pipe logic—Figure 9.18:

```
Node 4277 3.1ns (0.9ns)
S>sproc>INST_PIPE-2>CG-INVERTER-47>PMOS-4
```

IO-REGS delay (Fig. 9.6):

```
Node 1364 4.0ns (1.6ns)
S>sproc>SP-DP-1>IO-REGS-1>IO-REG-2>MUX-DRIVER-1>
     CG-INVERTER-3>NMOS-4
Node 3007 5.6ns (1.0ns)
S>sproc>SP-DP-1>IO-REGS-1>IO-REG-2>MUX-DRIVER-1>
     CG-INVERTER-4>PMOS-4
Node 3008 6.7ns (1.0ns)
S>sproc>SP-DP-1>IO-REGS-1>IO-REG-2>DP-MUX2-3#2>
     CG-TG-2>NMOS-1
Node 3082 7.6ns (0.7ns)
S>sproc>SP-DP-1>IO-REGS-1>IO-REG-2>BUS-DRV-2#2>
     CG-INVERTER-9>PMOS-4
Node 3126 8.3ns (0.6ns)
S>sproc>SP-DP-1>IO-REGS-1>IO-REG-2>BUS-DRV-2#2>
     CG-INVERTER-10>NMOS-4
```

BUS-OP gate (Figure 9.11):

```
Node 2372 9.0ns (0.9ns)
S>sproc>SP-DP-1>ALU-1>ADDER-1>BUS-OP-2#2>
     CG-INVERTER-14>PMOS-4
```

Manchester-adder delay (Figure 9.11):

```
Node 2313 9.9ns (4.0ns)
S>sproc>SP-DP-1>ALU-1>ADDER-1>MAN-16-2>MAN-4-LSB-1
Node 2381 13.9ns (0.7ns)
S>sproc>SP-DP-1>ALU-1>ADDER-1>MAN-16-2>MAN-4-1#0
Node 2379 14.6ns (0.7ns)
S>sproc>SP-DP-1>ALU-1>ADDER-1>MAN-16-2>MAN-4-1#1
Node 2380 15.3ns (4.0ns)
S>sproc>SP-DP-1>ALU-1>ADDER-1>MAN-16-2>MAN-4-1#2
```

Micron Ex. 1037, p. 690
Micron v. YMTC
IPR2025-00119

```
Node 2306 19.3ns (1.6ns)
S>sproc>SP-DP-1>ALU-1>ADDER-1>CG-INVERTER-15#15>NMOS-4
Node 2343 20.9ns (0.7ns)
S>sproc>SP-DP-1>ALU-1>ADDER-1>CG-INVERTER-7>PMOS-4
```

LE gate in control logic (Figure 9.21):

```
Node 1436 21.7ns (0.1ns)
S>sproc>SPROC-CONTROL-1>OR-17>NOR2STANDARD>
     N-CHANNEL-MOSFET-94
Node 3836 21.8ns
```

From this, it may be seen that ~48% of the timing budget is used by the 16-bit adder, 18% is spent in the register, and 22% is spent in the input-operand switching. Because this was not the overall worst path in the design and the timing was close to the design goal of 40 MHz, this timing was deemed sufficient. If improvements were required, the adder and IO-REGS could be further scrutinized (or the design could be transferred to a smaller process).

9.2.3.2 Register File

The register is arranged as a 32 × 64 memory with 4:1 column multiplexing. The register-file structure shown in Fig. 8.66(a) is used as the three-port register file. Figure 9.14(a) shows the transistor sizes used in the memory cell along with the read- and write-row decoders. The ratios of the write-path transistors and the storage inverter are chosen to ensure correct writes over all process corners. Two read-access transistors are used to provide dual read ports. Two read-row decoders and a write-row decoder that are based on a complementary 5-bit AND gate are used.

A 2-bit write address performs a column select for write operations, while 2-bit read addresses employ a multiplexer decoder (Fig. 8.63) to yield a 16-bit result. The column circuit, which is a 4-bit section, is shown in Fig. 9.15(a). It includes two 4:1 multiplexers (using single n-channel transistors), a sense inverter, and a buffer, which select the data to be routed to the IO-REGS. The write circuitry consists of a register to hold the write data and buffers for the four write-data lines (*WD<3:0>*) and write-strobe logic to allow the selective writing of any of the four columns (*WAS<3:0>*).

A write operation proceeds by placing an address on the write-address lines and then deasserting the clock. This causes one of four column WRITE-STROBE (*WAS*) signals to be asserted, which writes data into the cells with the word line asserted. Latches were added to the WRITE-ADDRESS (*WA*) lines to improve the speed.

Reads are totally static. For read operations, the critical path begins in the Instruction RAM, the output of which is passed to the word-line decoder of the register file. This in turn drives the row line of the register file, accessing a register. This triggers bit-line changes, which are demultiplexed and

Micron Ex. 1037, p. 691
Micron v. YMTC
IPR2025-00119

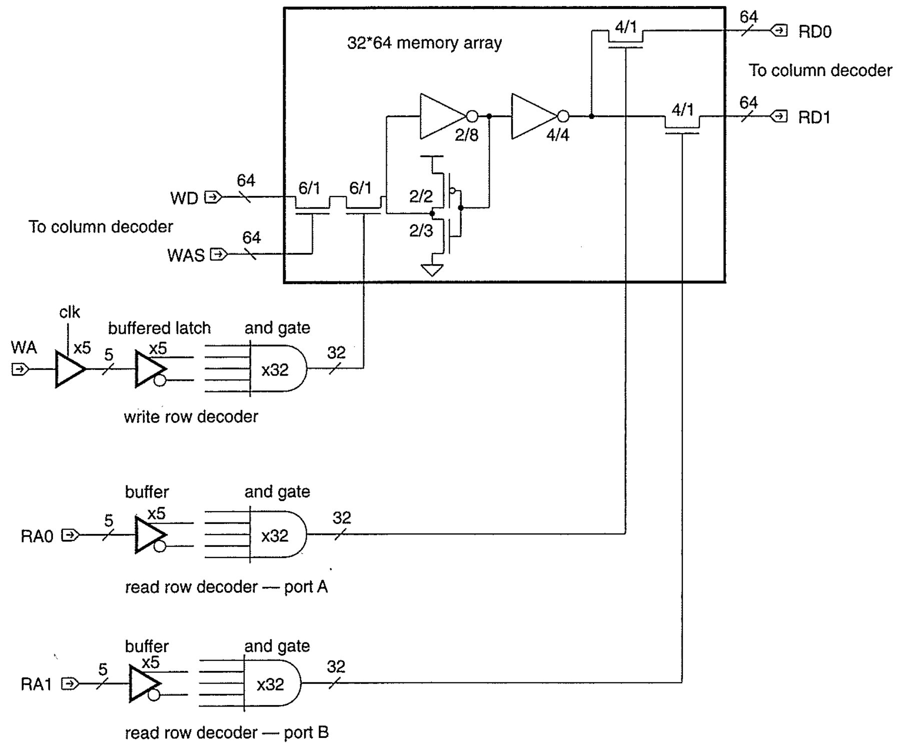


Figure 9.14 Register file partial schematic

driven to the IO-REGS module in the ALU-DP. There is a large spectrum over which the speed (and power) of the register file may be changed by sizing the row-decoder drivers, the decoder, AND gates, and the word-line drivers. In addition, moving the read registers into the register file could also be used to improve speed if necessary. Additionally, the bit lines could have been precharged to improve speed. In this case none of these improvements were required. A typical SPICE simulation for verifying the read-access time is shown in Fig. 9.15(b). This includes parasitic capacitances that are determined from a mask extraction of the layout of key cells of the register file (for instance, from a layout of the register file memory cell, the word-line and bit-line capacitances may be estimated). Figure 9.15(c) shows an address-input changing and the word-line response and the bit-line change. Figure 9.15(d) shows the bit-line response, the delay through the n-channel column multiplexer, and the final output. This shows that the delay from the address change to valid output is around 10 *ns*. The bit-line sense amplifier is ratioed to move the threshold voltage toward V_{SS}, which improves the sense time. For simplicity, the bit-line is not precharged, although some speed increase could be achieved by precharging and using a more exotic sensing scheme. However, this met the speed goals by almost half a clock cycle, so no further design effort was required.

Micron Ex. 1037, p. 692
Micron v. YMTC
IPR2025-00119

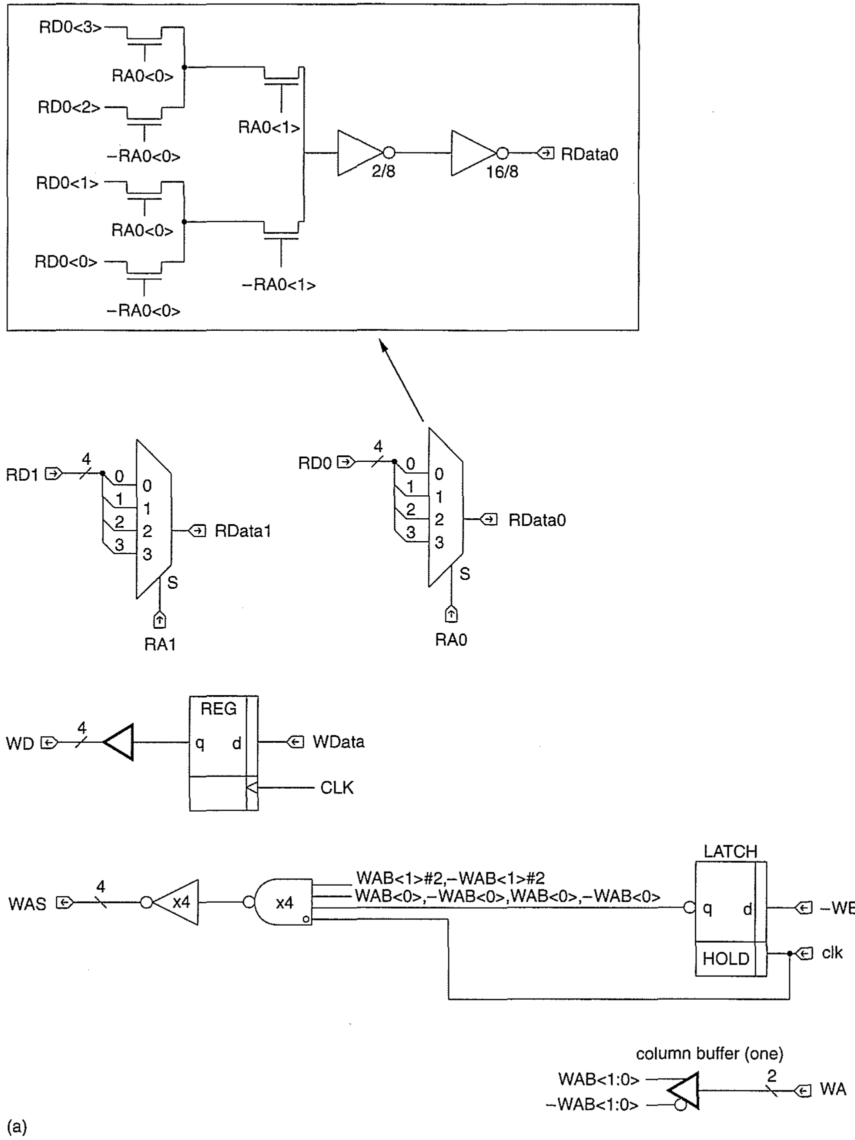
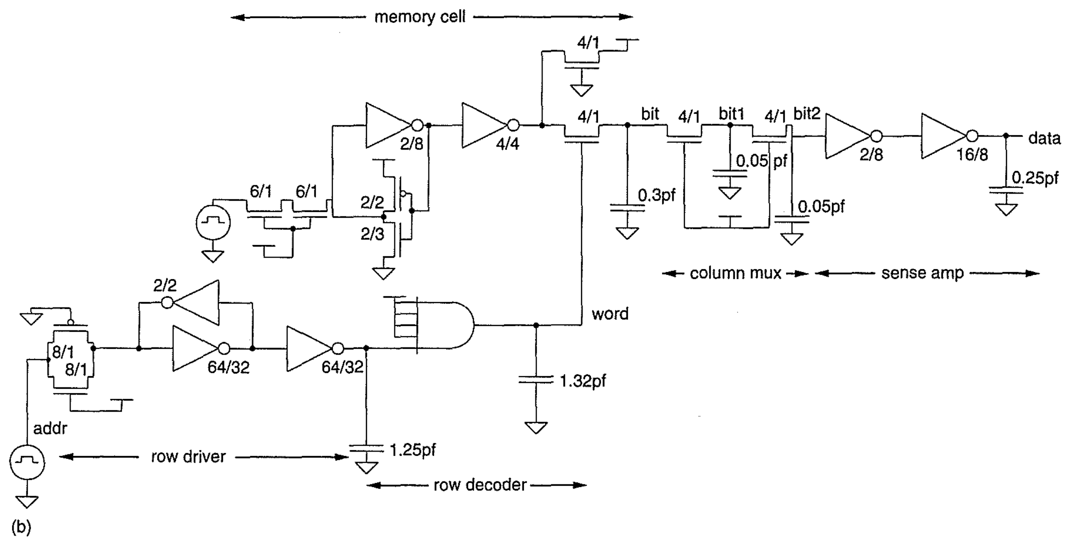


FIGURE 9.15 Register file read: (a) column decoder circuit; (b) SPICE model; (c) waveforms; (d) waveforms

A point that arises here is the importance of being able to rapidly prototype a design to assess where the speed bottlenecks are. Frequently this can be done with pencil and paper or more conveniently with a good top-to-bottom VLSI CAD system. Completing a rough "first draft" of a design can often highlight critical parts of the design ahead of time. This prevents work on areas that do not affect the performance of the overall system. Frequently, designers tend to take a myopic view of the design and can spend unnecessary time optimizing something that does not matter ("disappearing down the optimization rat-hole").

Micron Ex. 1037, p. 693
Micron v. YMTC
IPR2025-00119



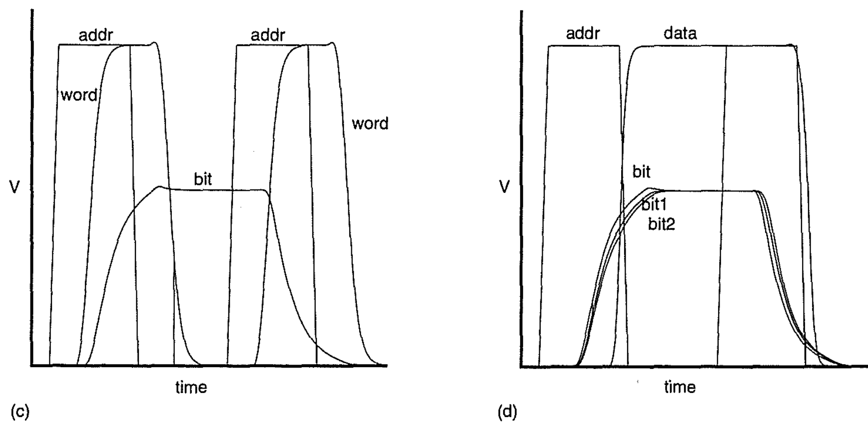
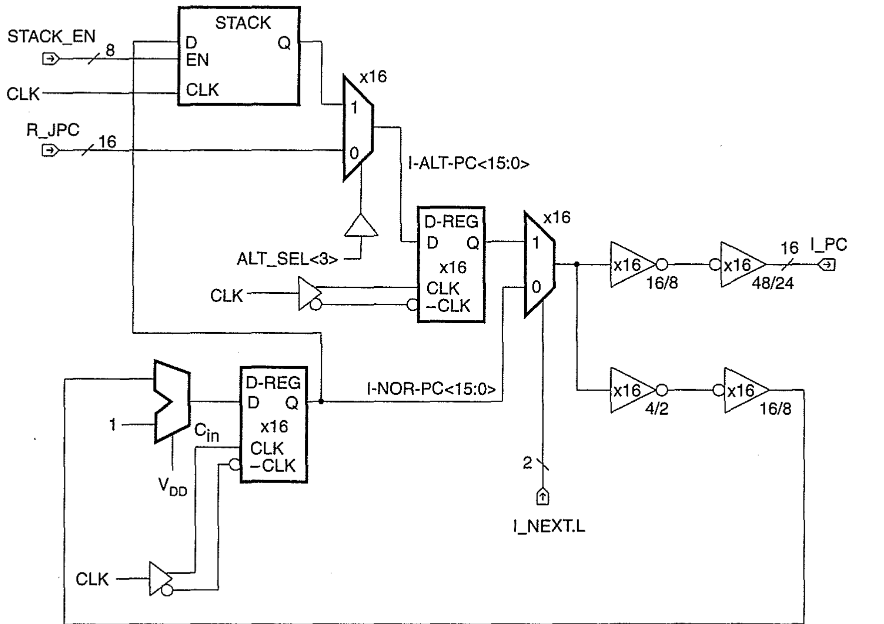


Figure 9.15 *(continued)*

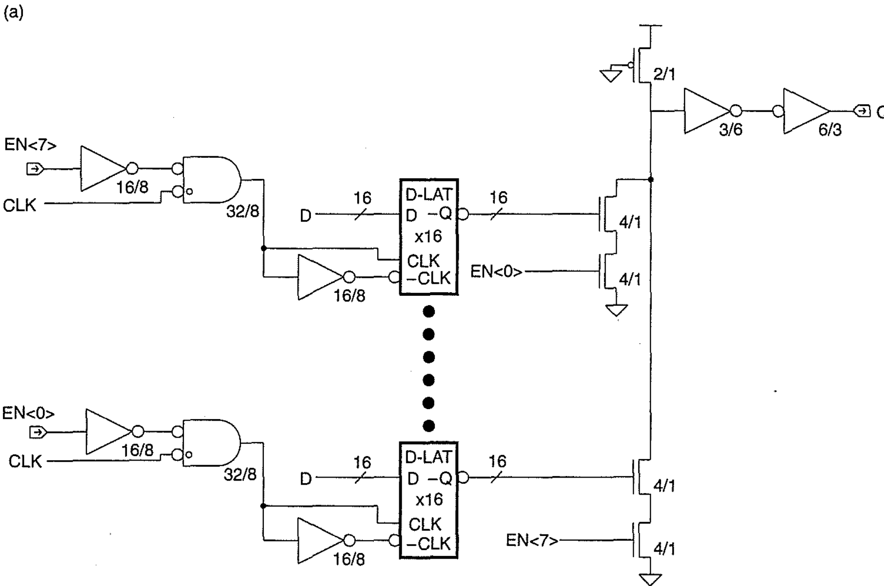
9.2.3.3 PC Datapath (PC_DP)

The PC datapath computes the next program-counter address (*I_PC*) (Fig. 9.16a). During normal operation the *I_PC* selects the incremented version of the PC (*I-NOR-PC<15:0>*). When a JUMP, CALL, or RETURN operation occurs, the I_PC register is loaded with the R_JPC (from the instruction) in the case of a JUMP or a CALL, or with the stack in the case of a RETURN (*I-ALT-PC<15:0>*). The multiplexers to achieve this can be seen in the figure. The signal *I_NEXT* selects either the "normal" PC (*I-NOR-PC*) on ALU operations or conditional jumps that fail. It selects the "alternate" PC (*I-ALT-PC*) on subroutine call, return, and conditional jumps that are taken. Note that the *I_PC* signal is duplicated, feeding the *I_PC* to the incrementer. This was done to improve speed because a critical path exists from the *I_NEXT.L* signal, through the incrementer and into register NPC, that switches the *I_PC* multiplexer. The regular *I_PC* output of the

Micron Ex. 1037, p. 694
Micron v. YMTC
IPR2025-00119



(a)



(b)

FIGURE 9.16 PC_DP

module is heavily loaded because it drives external modules. The duplicated path provides a faster *I_PC* to the incrementer, thereby improving speed.

The subroutine stack with a depth of eight is constructed from eight 16-bit latches. The stack design is shown in Fig. 9.16(b). One of eight latches may be conditionally written when the clock is low depending on an enable

Micron Ex. 1037, p. 695
Micron v. YMTC
IPR2025-00119

signal generated by the control logic. The outputs of the 8 registers are selected using a wired OR structure. One has to carefully design this circuit as there is a race that can occur between the data and the gated clock.

The module can be a 10- to 16-bit datapath, depending on the size of the program RAM or ROM.

9.2.3.4 Instruction Memory

In the first design that employed this processor, static RAM was used as the program memory. This decision was made because a number of algorithms were to be implemented by the processor and the algorithms were in a state of flux at the time of design. The RAM was a conventional, fast, static RAM employing a six-transistor cell similar to that described in Chapter 8.

9.2.3.5 Instruction Pipe

The Instruction Pipe (INST_PIPE) is shown in Figs. 9.17 and 9.18. This module registers certain parts of the instruction and generates the select con-

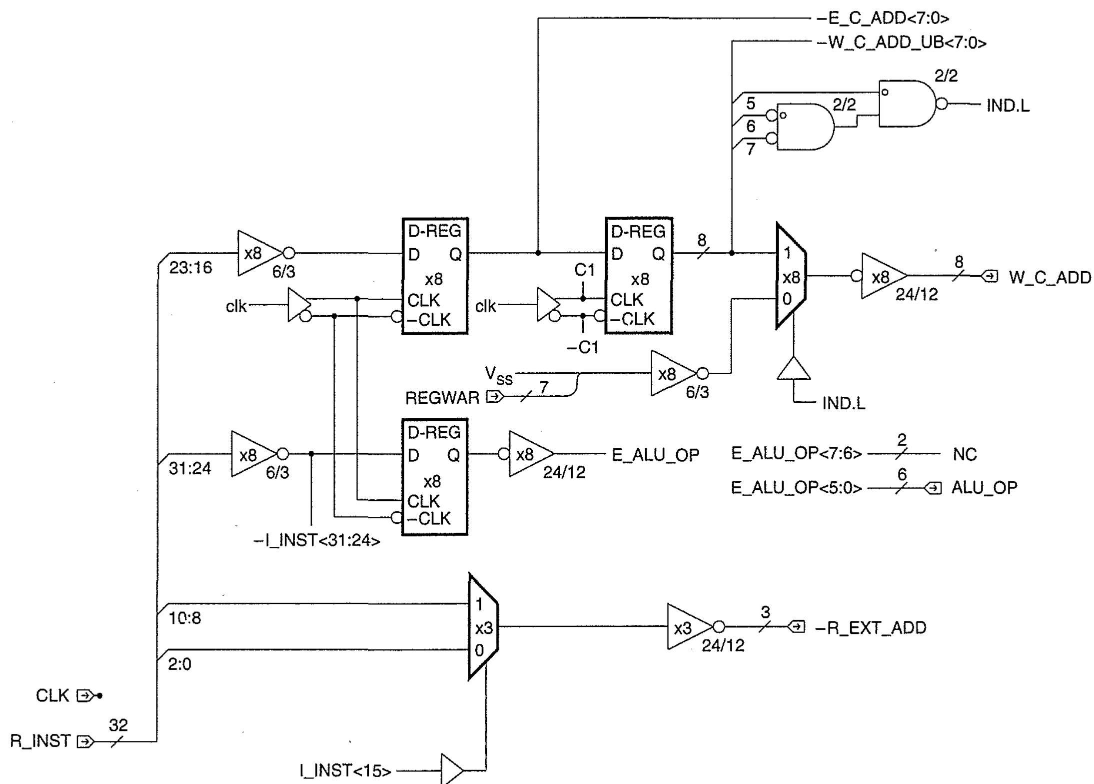


Figure 9.17 INST_PIPE module registers

Micron Ex. 1037, p. 696
Micron v. YMTC
IPR2025-00119

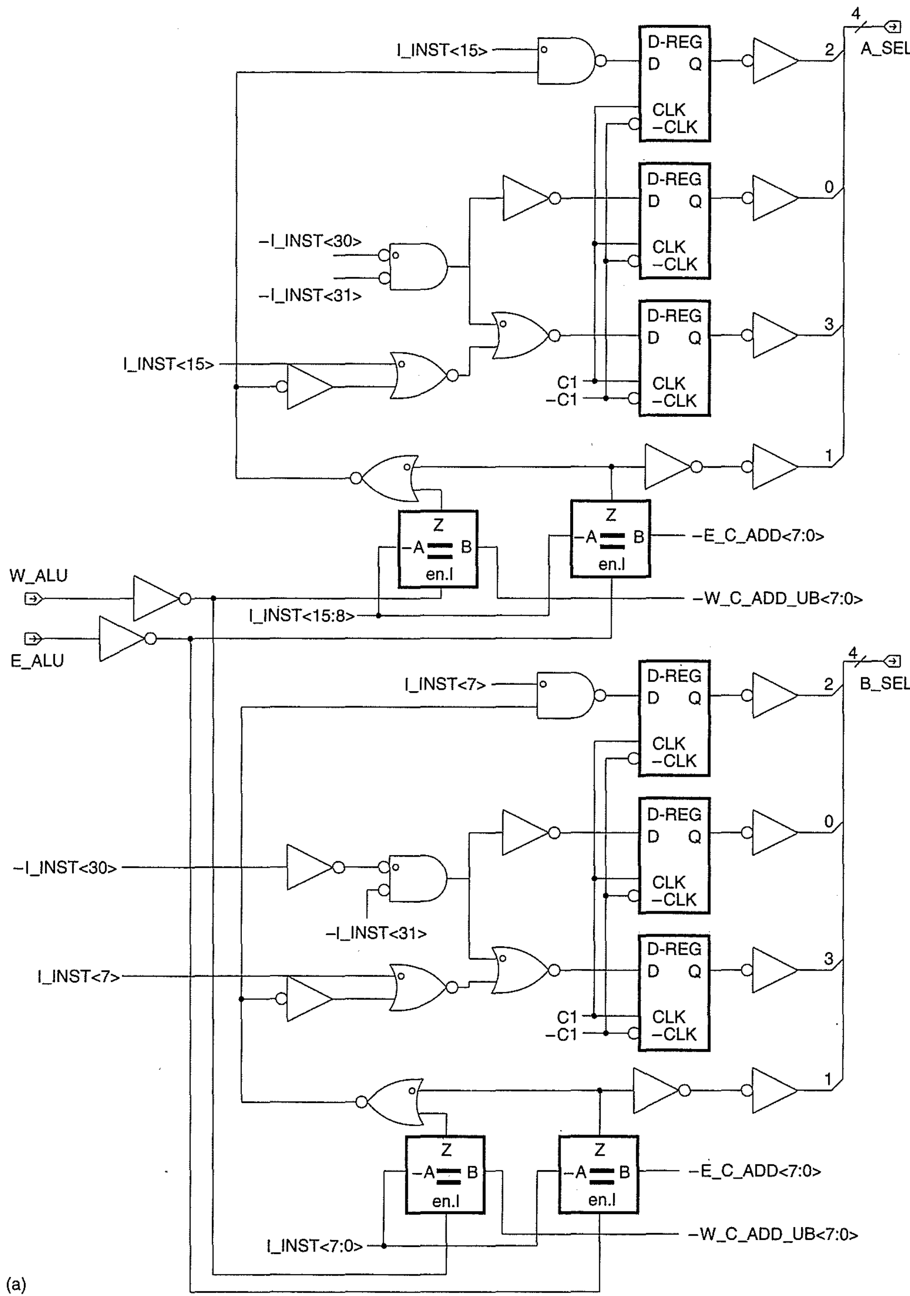


FIGURE 9.18 INST_PIPE address comparators: (a) schematic; (b) equality gate

trols for the IO-REGS module that determine which operands are fed to the ALU.

The Instruction RAM has a register that stores the I-stage instruction *I_INST <31:0>* and outputs *R_INST<31:0>*. In INST_PIPE, the *R_INST<31:24>* bits are registered to form *E_ALU_OP<7:0>*, which is fed

Micron Ex. 1037, p. 697
Micron v. YMTC
IPR2025-00119

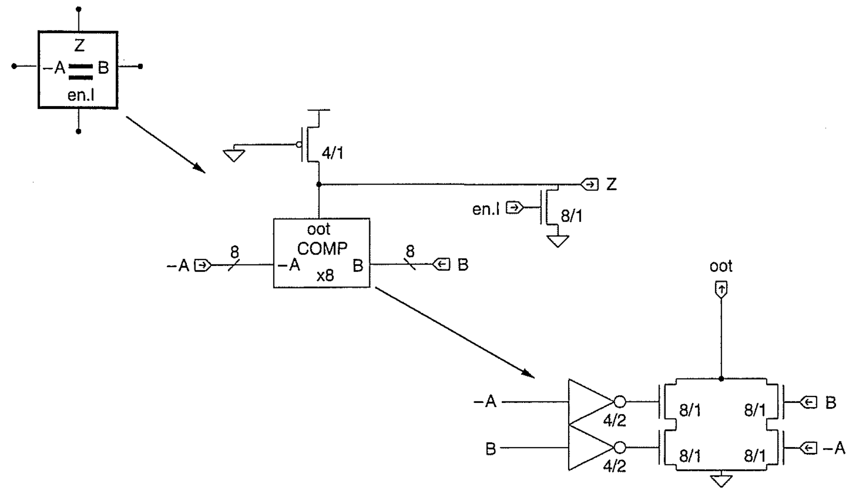


Figure 9.18 *(continued)* (b)

to the ALU_DP to control the ALU operation. While only six bits are required, a full 8-bit register was used because it maintains the regularity of the layout (and does not take up any more space in a data path).

R_INST<23:16> is stored in two successive registers to form *E_C_ADD* and *W_C_ADD,* the E-stage and W-stage register-file write addresses. These addresses are compared with the R-stage RA and RB addresses to determine whether bypassing is required (Fig. 9.18a). The select signals generated in this module are routed to the IO-REGS module in the ALU-DP. For instance by referring to Fig. 9.18(a), it may be seen that *A_SEL<1>* is set when the E-stage write address equals the R-stage address. Figures 9.4 and 9.5 show that when this signal is asserted, the E-stage data is forwarded to the bypass register. When *A_SEL<3>* is asserted, it selects between the register-file and the literal field (*A_SEL<0>*). When *A_SEL<2>* is asserted, it selects external data; when deasserted, it selects between *E_RESULT* and *W_RESULT,* as discussed above. *B_SEL* is similarly generated. The logic required to achieve this is shown in Fig. 9.18(a). Because this was a small amount of logic, it was placed in the datapath under the metal2 bus signals.

The address comparators use an enabled pseudo-nMOS XNOR gate as shown in Fig. 9.18(b). This gate is small and fast, and it fits unobtrusively into the datapath.

9.2.3.6 Control Logic

The control-logic block is responsible for four main control functions:

- instruction decode.
- microstack address control.

Micron Ex. 1037, p. 698
Micron v. YMTC
IPR2025-00119

- condition-code control.
- I/O control.

Figure 9.19 shows the instruction-decode logic. In addition to the *R-RET* signal the signals,

E-CALL, a Call instruction
E-JFALSE, a Jump False instruction
E-JTRUE, a Jump True instruction
E-ALU, an ALU instruction

are generated. The write enable for the register file (*W_WE*) is generated when there is a *W-ALU* signal while the *W_EXT_ADD_SEL<1>* signal is generated if an external register is not addressed. The *SHIFT-COUNT* signal is used by the shifter and is either derived from the op-code or an external register in the EXT_BUS_DP section.

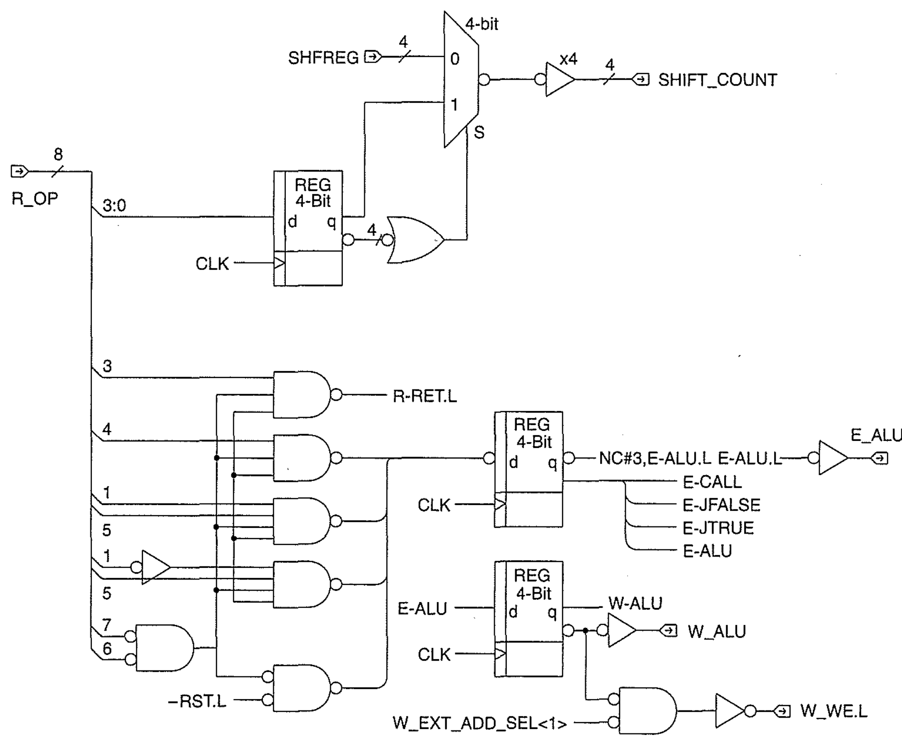


FIGURE 9.19 Control—instruction decode

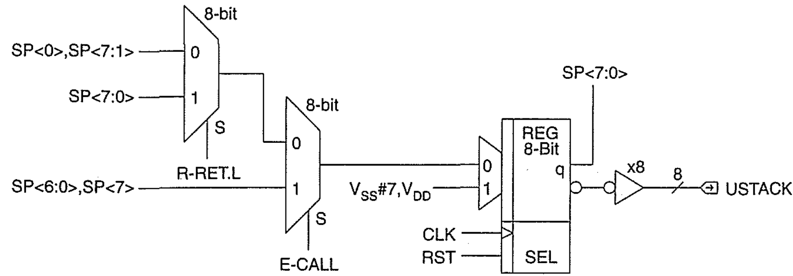


Figure 9.20 Control—stack address

The PC_DP-stack (microstack) address is generated using an 8-bit shift register (Fig. 9.20). The shift register is reset with a 1 in the LSB. When *E-CALL* is true, the register left-shifts, while when *R-RET* is true the register right-shifts. Thus this implements a pointer that points to the current return address. This could have been implemented as a 3-bit counter and a set of row decoders in the stack. However, this was deemed simpler and smaller for this size stack.

The condition-code logic is responsible for collecting the conditions, selecting the appropriate condition, and then controlling I_NEXT multiplexer in the PC_DP, as shown in Fig. 9.18. The registers are shown in Fig. 9.21, and the condition-code logic is shown in Fig. 9.22. In Fig. 9.21 the con-

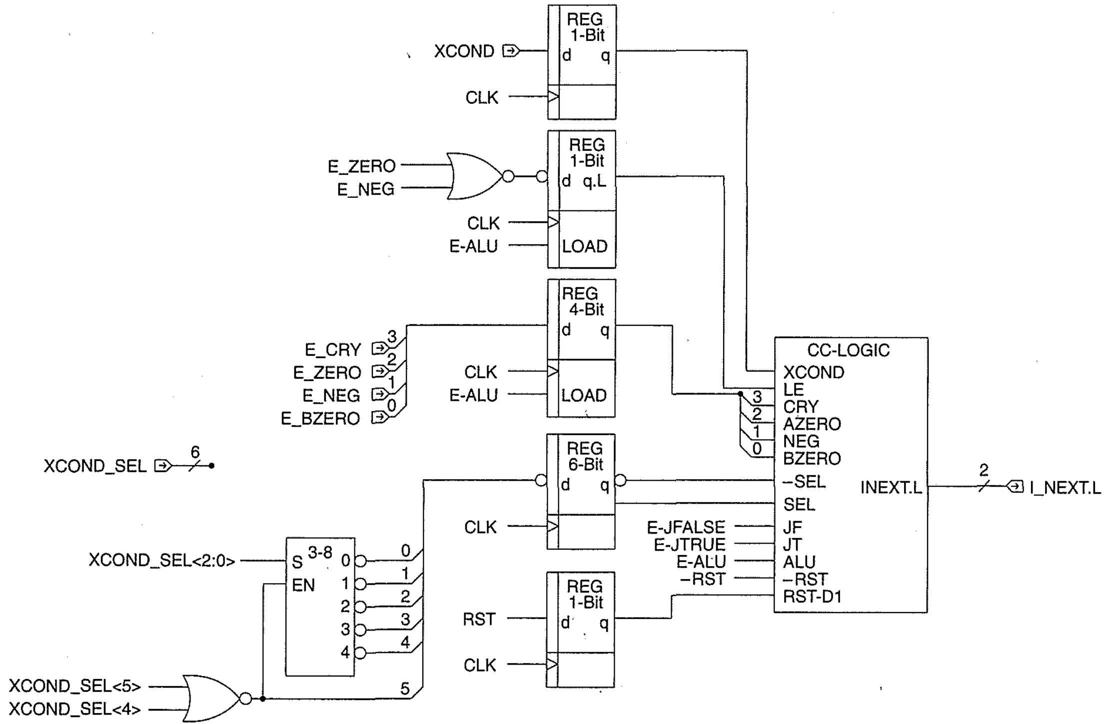


Figure 9.21 Control—condition code registers

Micron Ex. 1037, p. 700
Micron v. YMTC
IPR2025-00119

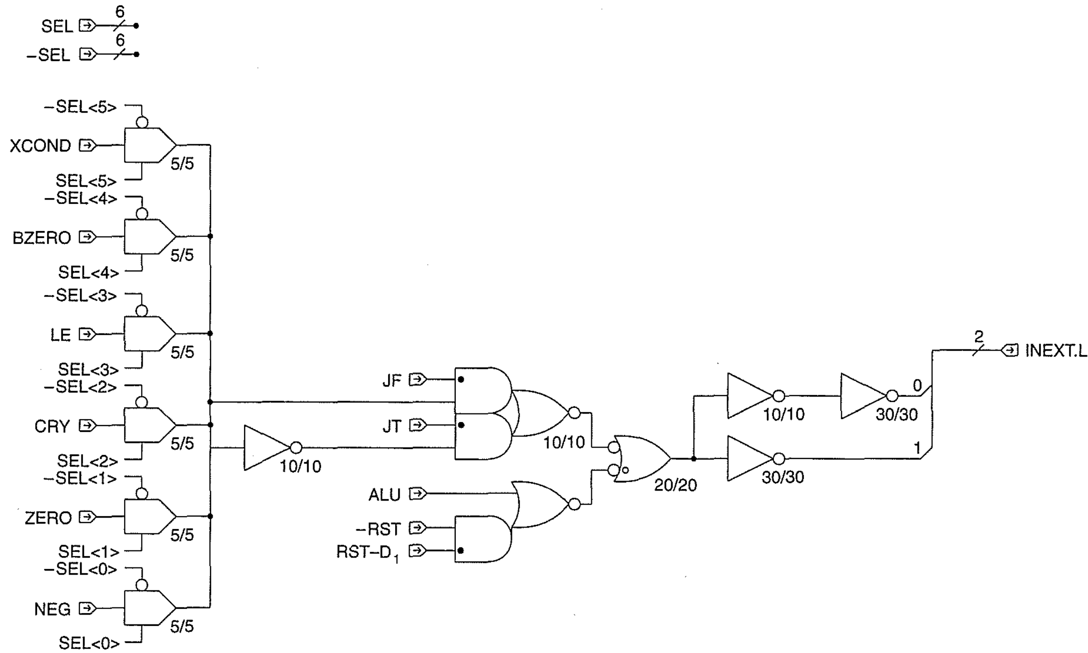


FIGURE 9.22 Control—condition code selection

ditions and the selection field (*XCOND_SEL*) are registered. Conditions that come from the ALU are

E_CRY—carry from adder

E_ZERO—adder zero

E_NEG—high bit of ALU

E_BZERO—Boolean zero.

In addition extra conditions are generated. For instance, *LE* (less-than-or-equal-to) is *E_ZERO* ORed with *E_NEG*. A condition may also be passed from an external source (*XCOND*).

Figure 9.22 shows the condition-code logic. First, a 6-input multiplexer selects the appropriate condition. This is then passed to a set of gates that control the *INEXT* signal dependent on *JF* (Jump False), *JT* (Jump True), *ALU* (an ALU instruction—no jump can be taken), and reset.

Finally, the I/O control logic is shown in Fig. 9.23. This controls the writing of registers and the tristating of busses in the EXT_BUS_DP. The *EXTOE* bus controls the enabling of external registers onto the external bus for reading by the ALU. The *EXTEN* signals control the loading of various external registers caused by writes by the ALU or the returning *SYSBUS* data.

Micron Ex. 1037, p. 701
Micron v. YMTC
IPR2025-00119

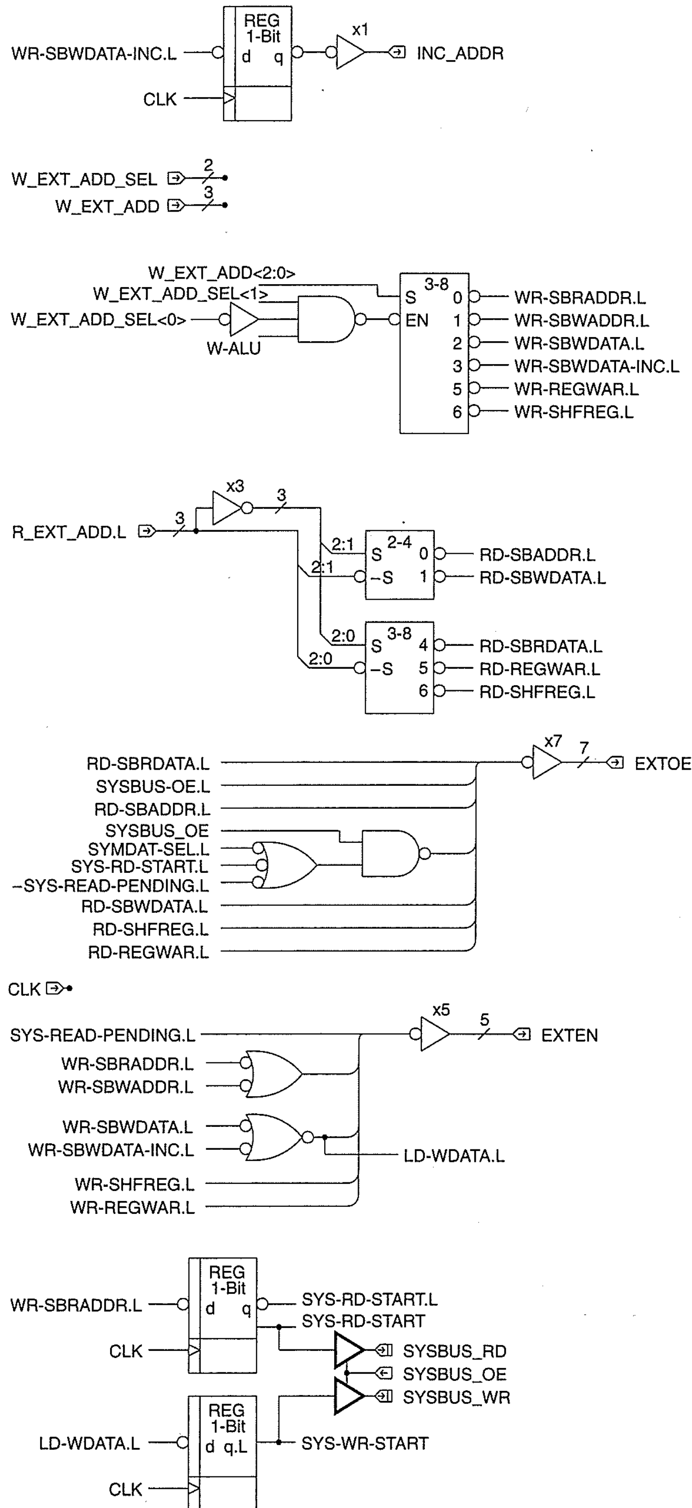


FIGURE 9.23 Control—I/O path

Micron Ex. 1037, p. 702
Micron v. YMTC
IPR2025-00119

9.2.4 Layout

The microcontroller was constructed using three styles of layout, namely,

- datapath elements
- standard-cell layout
- memory layout

and of course, routing.

A datapath layout strategy was selected for this design, which is based on one its designers have used for a large Lisp microprocessor and which has proved useful on numerous other datapath chip layouts (Fig. 9.24a). Metal2 power busses run at the bottom and top of the cell. Sometimes these may be omitted and the power busses run vertically in metal1. Space is allowed for four metal2 busses to run through the cell (or five without metal2 power busses). The choice of four busses was originally made for

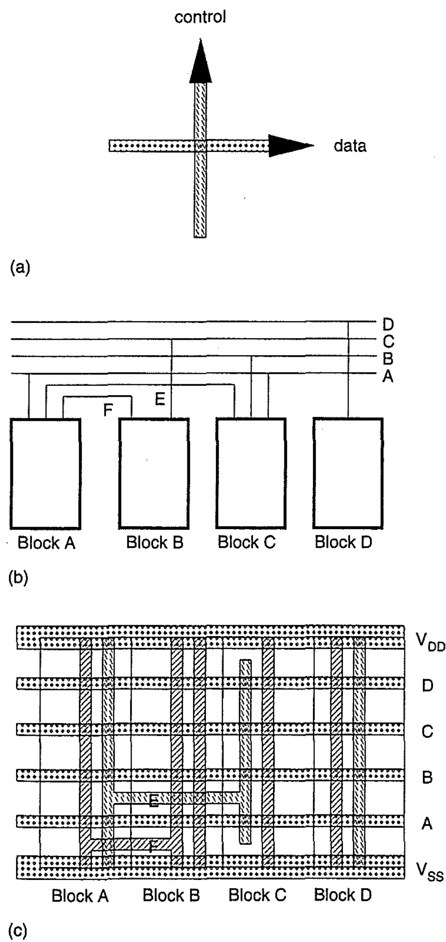


FIGURE 9.24 Datapath strategy

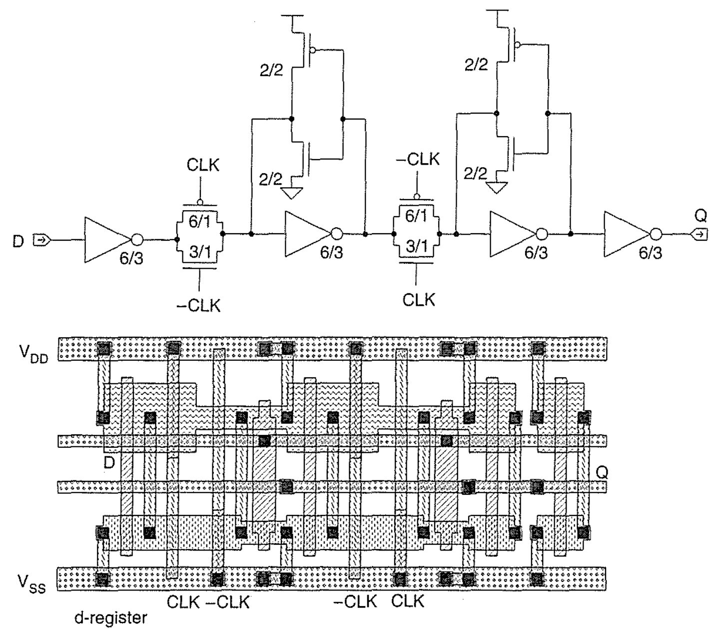
Micron Ex. 1037, p. 703
Micron v. YMTC
IPR2025-00119

the microprocessor, and most datapaths have been shoehorned into this structure. A designer of course could choose more route-through busses but probably not fewer. Active circuitry is placed under the route-throughs in diffusion, polysilicon, and metal1. In a nonsilicided process, metal control lines are run vertically, while in a silicided process, the polysilicon may be used as control lines as long as the delay does not impact speed (be warned: in high-speed circuits silicide is really not adequate). More than four busses may be run through cells for short distances. For instance, adjacent connects between cells may be made in polysilicon or metal. Metal (or poly) can pass over an intervening cell if metal (or poly) transparency is provided in the cell. For instance, to achieve metal transparency, all vertical connections are made in polysilicon. This normally elongates the cell but may provide the right trade-off of increasing the length of one cell rather than adding an extra bus for every bit for the complete length of the datapath. Figures 9.24(b) and (c) show an example of 5 to 6 busses being routed. The height of the cell is determined by a combination of the metal2 pitch and the n-to-p spacing of the transistors. There is a maximum width of the horizon-

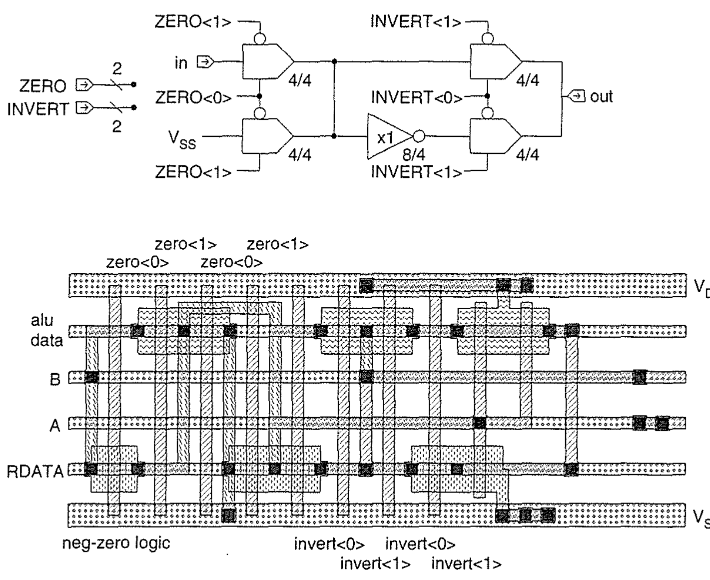
Table 9.1 Standard Cells

INVERTING FUNCTIONS	
INVERTER	
2-input NAND	2-input NOR
3-input NAND	3-input NOR
4-input NAND	
NONINVERTING FUNCTIONS	
2-input AND	2-input OR
3-input AND	3-input OR
4-input AND	BUFFER 1X, 2X, 4X, 8X drive
STORAGE ELEMENTS	
D REGISTER	D LATCH
D REG/CLEAR	D LATCH/CLEAR
D REG/SET	D LATCH/SET
D REG/MUXED	
OTHER LOGIC FUNCTIONS	
2-input MUX	Tristate-Buffer
4-input MUX	
XOR	
XNOR	
AND-OR-INVERT 221	
OR-AND-INVERT 221	

Micron Ex. 1037, p. 704
Micron v. YMTC
IPR2025-00119



(a)



(b)

FIGURE 9.25 Datapath layouts: (a) register; (b) BUS-OP gate

tally oriented n- and p-transistor that determines the minimum height of the cell given a certain number of route-throughs. If wider transistors are required, then they must be rotated or composed of multiple smaller transistors. Figure 9.25 shows a number of examples of symbolic layouts of cells designed using this style of layout. Figure 9.25(a) shows the *D*-register cell, and Fig. 9.25(b) shows the BUS-OP cell used for conditionally negating and zeroing the *B* operand to the ALU. Figure 9.26 shows 4 bits of the Manchester adder (also Plate 10).

Micron Ex. 1037, p. 705
Micron v. YMTC
IPR2025-00119

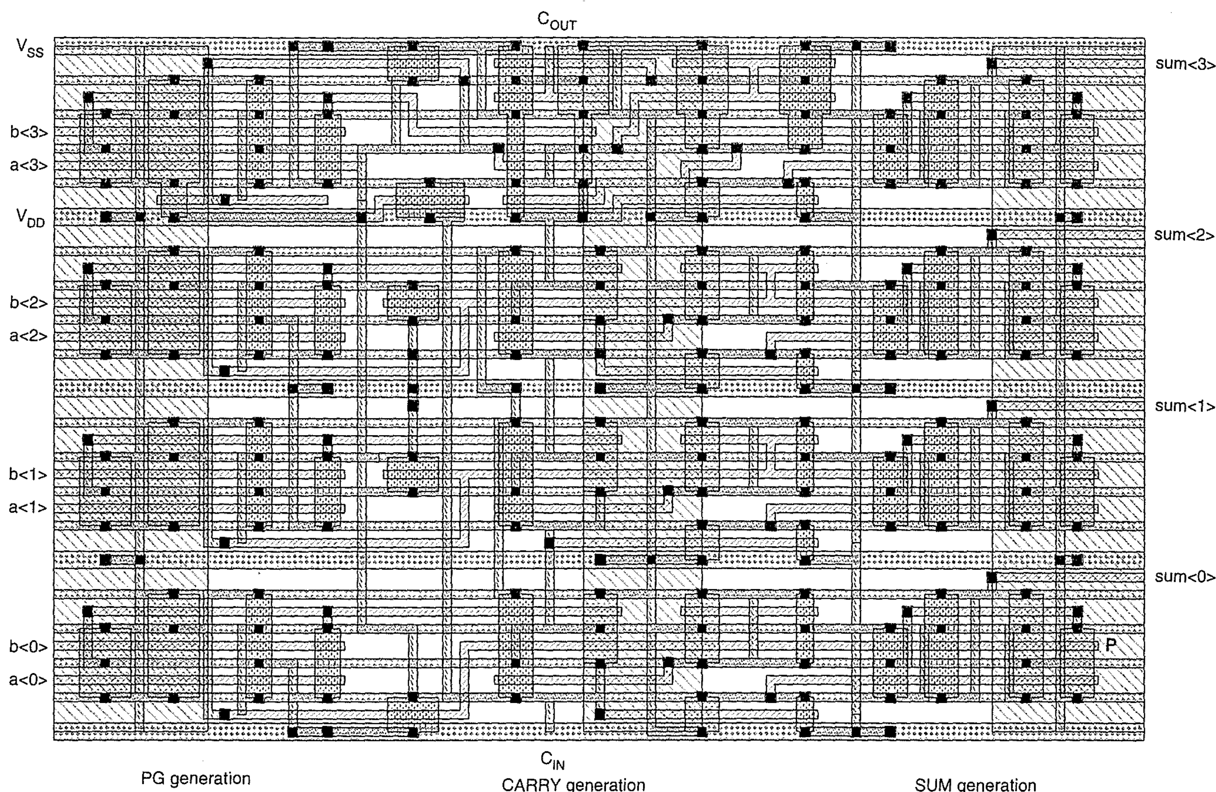


FIGURE 9.26 Manchester adder layout

The standard cells in the control section use a fairly standard two-level-metal routing strategy (see Fig. 6.29b). Metal1 power busses run horizontally, and polysilicon (silicide) runs vertically. Routing channels may be metal1/metal2 or metal1/poly. Table 9.1 gives a list of all the standard cells used.

The standard cells are placed by the TimberWolf program[1] and routed symbolically. The resulting layout is compacted to form a mask layout. Of course, any place-and-route program may be used to build the standard-cell logic.

9.2.4.1 Datapath Floorplans

When using datapath elements with a constrained number of route-throughs, the designer needs to determine an ordering of the functions on the datapath that does not require the number of feedthroughs to be exceeded. The ordering of functions on a datapath may be determined by permuting the order of the blocks and counting the number of connections between the blocks. This is done in a top-down manner from the highest level at which a single datapath is required. Common tricks employed include using tristate drivers as multiplexers (thus only requiring one common wire), replicating logic to reduce route-throughs, or using routing layers other than metal2 on adjacent or nearly adjacent modules. As noted previously, adjacent connections may be made in polysilicon or metal.

Micron Ex. 1037, p. 706
Micron v. YMTC
IPR2025-00119

As an example of a typical floorplan, the ALU_DP module will be examined. The floorplan follows the schematic hierarchy, employing a layout block for the IO-REGS, ALU, and EXT_BUS_DP. These are placed adjacent to each other, as shown in Fig. 9.27. The register-file and literal ports enter on the left, and the system-address and data ports are accessed at the right of the ALU_DP. The control signals and clock enter at the bottom. From Fig. 9.4 it may also be seen that the *R_EXT_BUS* and *ALU_OUT* bus have to connect to all three modules. Just at this level it may be seen that IO-REGS is going to have at least six horizontal busses passing through it, so some thought has to be given to this potential problem.

Examining Fig. 9.28(a), the floorplan of the IO-REGS module may be seen. This module consists of 16 bit-slice sections vertically abutted on top of a control section. At this level *R_A_DATA, R_B_DATA, R_LITERAL,* and *W_C_DATA* enter on the left of the module. *A*, *B*, and *EXT_DATA* enter on the right. The bit-slice is shown in Fig. 9.28(b). It does not follow the schematic hierarchy in order to meet the four-bus constraint. With the arrangement shown in Fig. 9.28(b), no more than four busses are required in the datapath bit-slices.

Figure 9.29(a) shows the floorplan of the EXT_BUS_DP module. It consists of a number of 16-bit modules abutted to a common control section. Each vertical 16-bit section is associated with a register, which is shown in Fig. 9.7. Some registers (e.g., SHFREG and REGWAR) have bus-throughs in their upper bits. Figure 9.29(b) shows the RDDATA register-block bit-slice, which is composed of a register and tristate buffer abutted. The controls run vertically.

Figure 9.30(a) shows a floorplan of the ALU. The ALU datapath is split according to the schematic hierarchy (Fig. 9.8), that is, an adder, a Boolean unit, and a shifter block. Figure 9.30(b) shows the floorplan of the adder. It consists of a 16-bit BUS-OP, Manchester adder, zero detect, and bus-driver horizontally abutted, with a control block at the bottom. The adder is, in turn, four 4-bit Manchester sections (Fig. 9.26). The Boolean-unit bit floorplan is shown in Fig. 9.30(c). Finally, the shifter floorplan is shown in Fig. 9.30(d). The shifter was

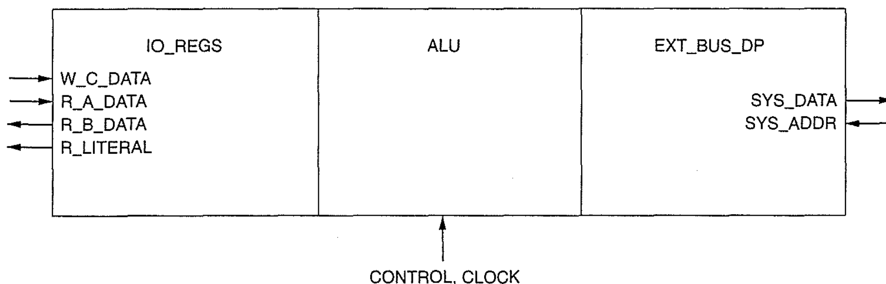


Figure 9.27 ALU_DP floorplan

Micron Ex. 1037, p. 707
Micron v. YMTC
IPR2025-00119

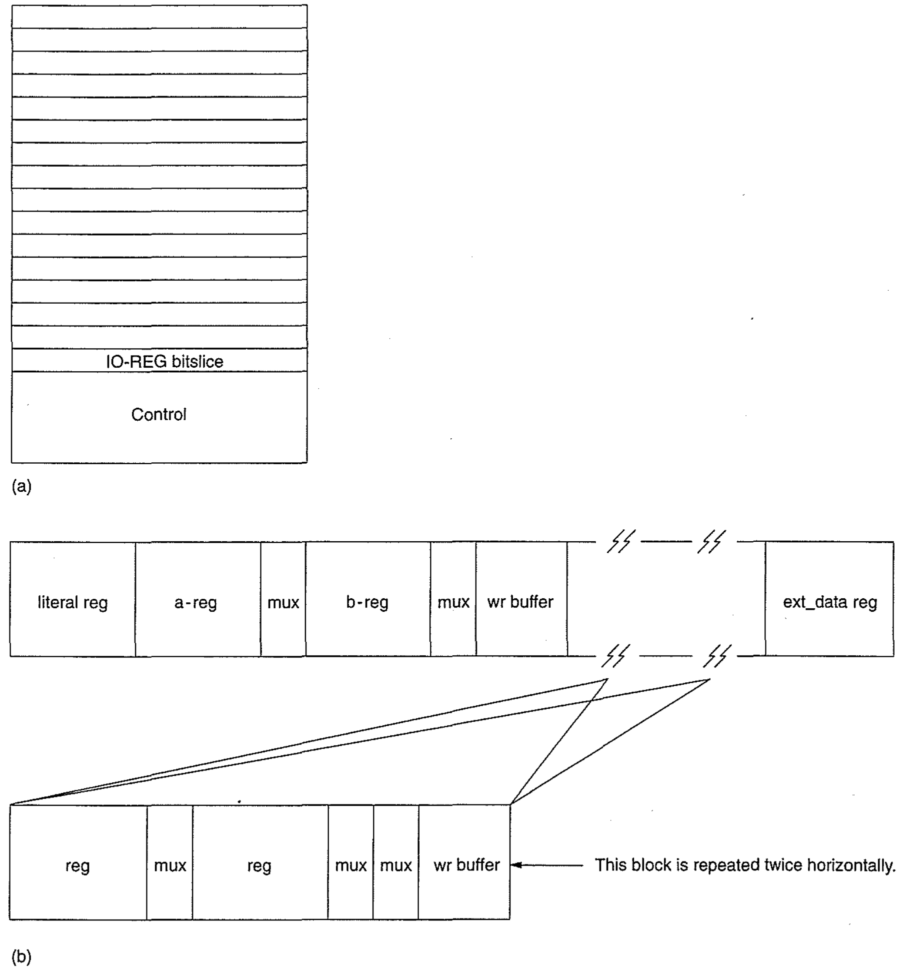


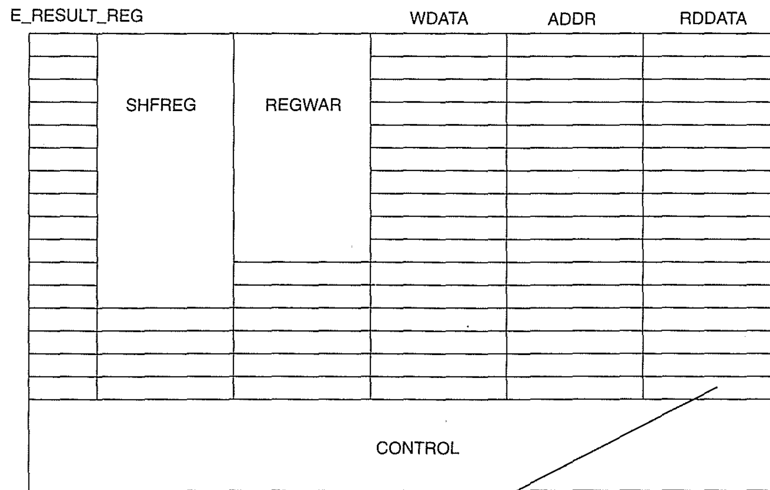
FIGURE 9.28 IO-REGS floorplan: (a) complete module; (b) 1-bit slice

designed so that one module shifted left and the same module, reflected in *X* and *Y* and abutted to a common central bus-buffer module, shifted right.

The other datapath modules are similarly constructed. As an example of a memory structure, the floorplan of the register file is shown in Fig. 9.31. The storage array (the register file shown in Fig. 9.14a) is abutted at the right by 16 4-bit column circuits, represented in Fig. 9.14(b). The three row-decoders are arrayed at the bottom of the memory array. Buffered address lines for the read and write ports run horizontally and are buffered in the bottom-right corner. Column-address buffers are placed above these drivers.

The control standard cell does not have an ordered floorplan but was specified as a two-row standard-cell layout that turns out to be about as long as the ALU_DP datapath so that it may be conveniently placed adjacent to this module. Figure 9.32 shows a possible floorplan of the processor.

Micron Ex. 1037, p. 708
Micron v. YMTC
IPR2025-00119



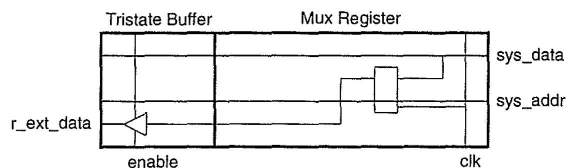


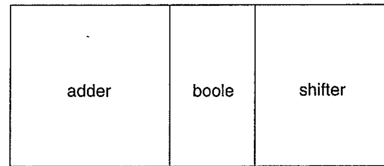
Figure 9.29 EXT_BUS_DP floorplan: (a) complete module; (b) an example bit

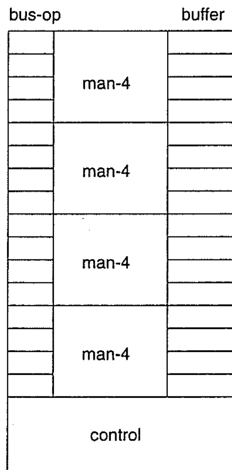
9.2.5 Functional Verification and Testing

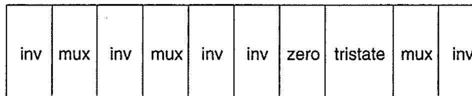
A schematic for the design was first captured and the functionality of various modules checked. For instance, the adder, Boolean unit, and shifter had tests written for them. In addition for this design a C register transfer model of the processor was written as the RTL schematic was being developed. Once the overall processor was captured at the schematic and RTL levels, an assembler was written so that programs could be written using the instruction set of the machine to verify the functionality. A suite of tests were written to check each instruction class and type of operation. For instance, all arithmetic instructions were checked, a test was written to check the register file, the pass-around logic was checked, and the condition-code logic was checked. These vectors were enhanced to increase fault coverage by running them on a fault simulator.

Apart from these initial functional tests, timing simulations were run on the backannotated schematics to verify the performance of modules such as the adder. A timing analyzer was then used to report overall worst-case timing paths for the processor (and its peripherals) as a whole. This was first done with the layout incomplete, and as layouts were completed, the accu-

Micron Ex. 1037, p. 709
Micron v. YMTC
IPR2025-00119







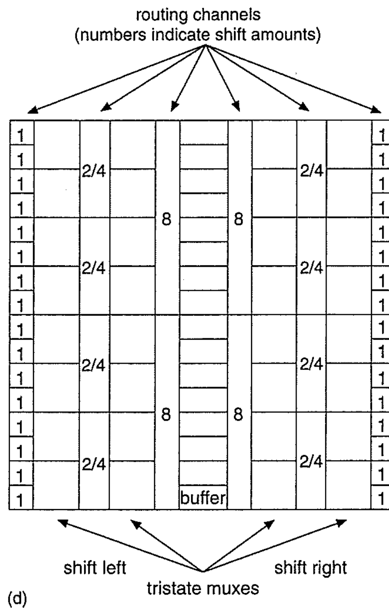


FIGURE 9.30 ALU floorplans: (a) complete module; (b) adder; (c) Boolean unit bit; (d) shifter

rate backannotated schematics were used to achieve increasingly accurate timing analyses. Frequently, gates had to be moved between modules and between pipeline stages to achieve the desired speed.

Micron Ex. 1037, p. 710
Micron v. YMTC
IPR2025-00119

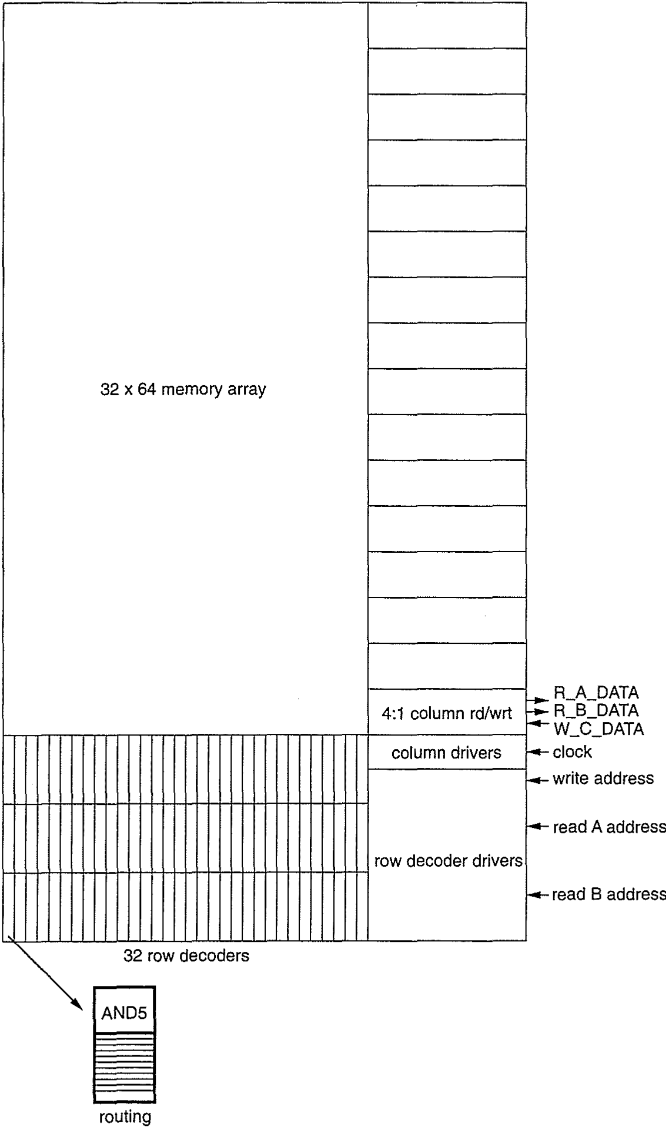


Figure 9.31 Register-file floorplan

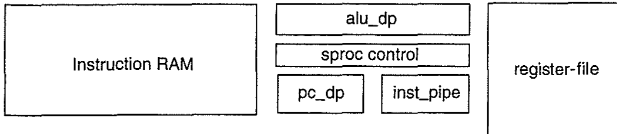


Figure 9.32 Possible processor floorplan

In addition, extensive tests were written for the C functional model. The vectors at the boundary of the symbol processor were captured and applied to the schematic version and the corresponding outputs checked for consistency.

The C RTL model was merged with other C-level models for the complete chip, and the system as a whole was tested by pumping data though the complete chip, and comparing the output values with a "golden" software model. This kind of testing verifies that the processor can be programmed to perform the required signal processing operations, but it does not give very much insight into whether all instructions work for all operands in the processor.

So in summary there were a number of levels of verification and testing:

- At the individual submodule level (functionality (C RTL simulator and transistor level simulator) and timing tested (transistor level simulator)).
- At the processor level (functionality (C RTL simulator) and timing tested (transistor level simulator and timing analyzer)).
- At the chip level (functionality and timing tested; C RTL simulator, module level timing analyzer used).

A good strategy to have is a set of regression tests that are run anytime a change is made to a module. These can be run in bottom-up mode so that bugs in low-level modules are found without having to find the bugs in time-consuming high-level simulations.

9.3 A TV Echo Canceller

The chip described in this section[2] (designed by A. Corry, B. Edwards, and N. Weste of TLW, and C. Greenberg of Philips Laboratories) is presented as an example of the kind of structure that lends itself to implementation as a regular structure. Because the chip is dominated by this regular structure, a high proportion of the engineering of the chip may be directed at the repeated structure, thereby providing effective use of the chip area.

9.3.1 Ghost Cancellation

This application is in the area of video-ghost cancellation.[3,4] Terrestrial and cable TV transmissions are subject to multiple-path propagation and transmission-line impedance discontinuities. Both of these imperfections in the communication channel lead to what is termed "ghosting," or echoes, which is familiar to most TV viewers.

Micron Ex. 1037, p. 712
Micron v. YMTC
IPR2025-00119

Figure 9.33(a) shows a representation of the transmission path subject to ghosting. The signal at the receiver is given by

$$S_R = S_T(1 + H_G),$$

where

S_R = the received signal
S_T = the transmitted signal
H_G = the contribution of ghosts to the signal at the receiver.

If the receiver incorporates a filter structure, as shown in Fig. 9.33(b), then

$$S_C = \frac{S_R}{(1 + H_{GC})}$$

$$= S_T \frac{1 + H_G}{1 + H_{GC}},$$

where

S_C = the processed signal
H_{GC} = the response of the filter.

If $H_{GC} = H_G$, then the original signal is restored.

A typical ghosted signal is shown in the time domain in Fig. 9.34(a). It consists of the main signal and a set of ghosts that precede the main signal (pre-ghosts) and a set of ghosts that follow the main signal (post-ghosts). The filter structure shown in Fig. 9.34(b) may be used to cancel the ghosts shown in Fig. 9.34(a). A filter with characteristic H_{GC1} prior to the adder cancels the pre-ghosts, and the filter with characteristic H_{GC2} is used to cancel the post-ghosts. A delay line adds the main signal at the required point in time via a three input adder.

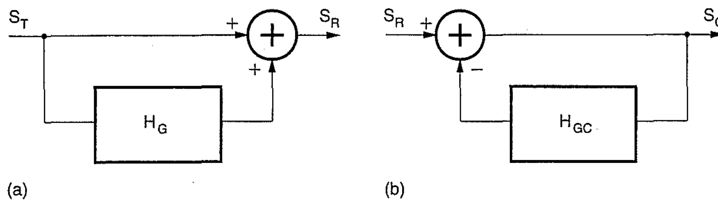
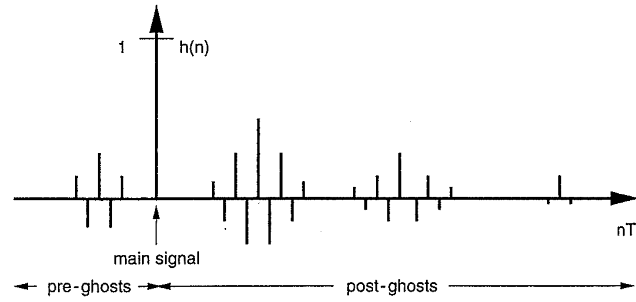


FIGURE 9.33 Ghosts: (a) transmission channel; (b) receive channel

Micron Ex. 1037, p. 713
Micron v. YMTC
IPR2025-00119



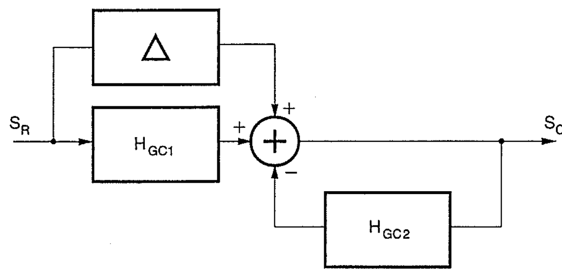


Figure 9.34 Ghosted signal: (a) time domain; (b) possible cancellation filter

The filters shown in Fig. 9.34(b) may be implemented with filters that are called Finite Impulse Response (FIR) and Infinite Impulse Response (IIR) filters. By sending a known "training signal" as part of a regular TV signal, the filter coefficients of the two filters may be determined and the ghosts canceled.[5,6]

9.3.2 FIR and IIR Filters

Mathematically, a sampled data FIR filter is represented by

$$y(t) = \sum_{i=0}^{n} h_i x(t), \tag{9.1}$$

where

$y(t)$ = the filtered signal stream at time t

$x(t)$ = the input signal stream at time t

h_i = the filter coefficients

n = the order or length of the filter.

An IIR filter may be constructed by using an FIR filter with feedback.

Micron Ex. 1037, p. 714
Micron v. YMTC
IPR2025-00119

There are a number of well-known forms for the sampled data FIR filter. Some are shown in Fig. 9.35. Figures 9.35(a) and (b) show two straightforward implementations of a 4-tap FIR filter. In the first implementation a delayed version of the incoming signal (X) is fed to a set of taps comprised of a multiplier and an adder. The multiplier multiplies the coefficient (H_n) by the delayed version of X. The adders are cascaded to form the final sum Y. In the second implementation, the delay is placed between adders in the taps. Each filter tap requires a register, an adder, and a multiplier. The precision of

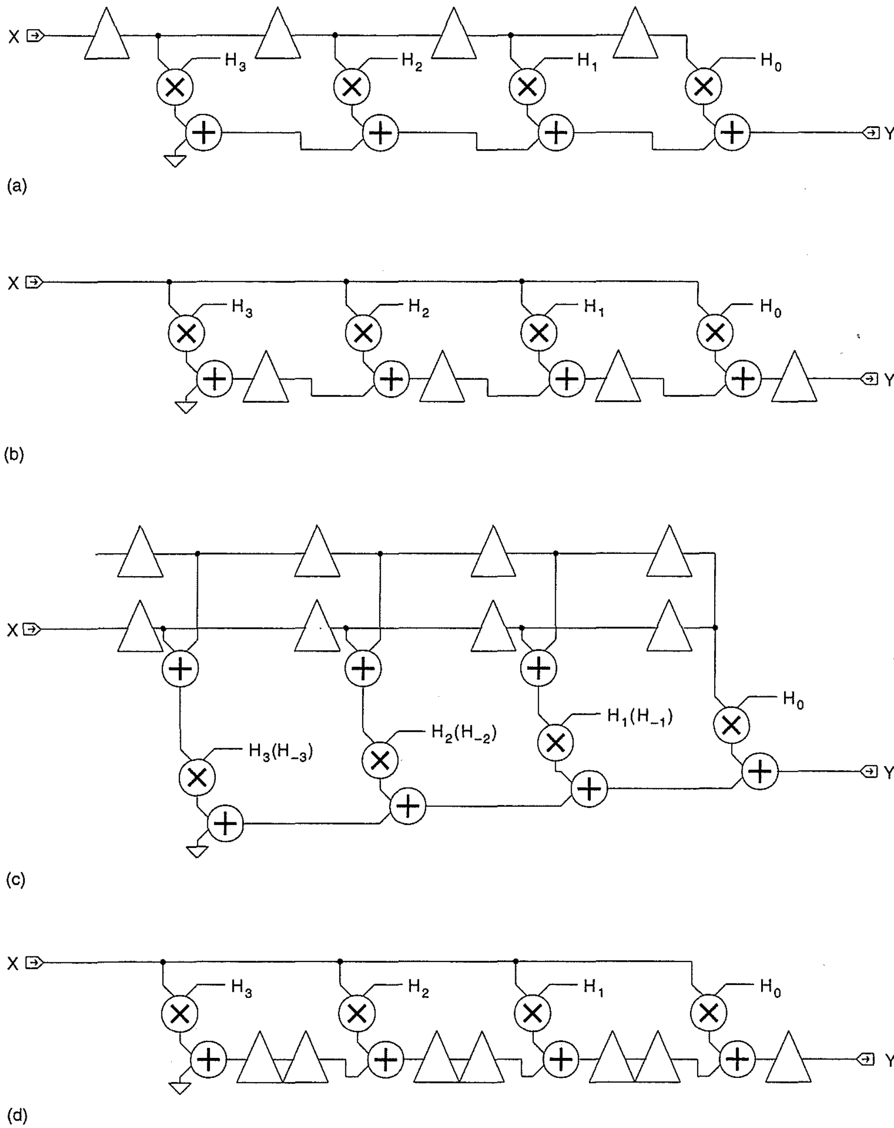


FIGURE 9.35 FIR architectures

the adder and multiplier depend on the precision of the coefficient, the length of the filter, and the desired precision or result. Implementations vary from just writing code on a microprocessor or DSP processor (for speech applications) to full hardware implementations. The multiplier can be a costly component in high throughput situations (such as video). If the filter response is fixed and the coefficients are therefore fixed, the multiplier may be simplified to contain only the product terms required. For instance, if an 8-bit coefficient is 00110011, then only four adders are required to perform this fixed multiplication. For 01111110, only two adders are required (the MSB and LSB are subtracted from the signal). This is known as Canonic Signed Digit representation.[7] In fixed FIR filters, a great deal of signal-processing expertise goes into designing the coefficients so that the number of adders is reduced. Unfortunately, in the ghost-cancellation application, the filter response is adaptive (i.e., it has to be programmable) and other methods have to be sought to reduce the size of the filters.

Many times the coefficients are symmetric, that is $H_{-n} = H_n$. A symmetric filter is shown in Fig. 9.35(c). The cost of adding two taps in this structure is only two adders, a multiplier, and two registers. Frequently, every other coefficient is zero, in which case the structure in Fig. 9.35(d) is useful.

A single stage of an FIR filter requires one add and one multiply (called a multiply-accumulate operation) at the appropriate resolution, which is dictated by the data and coefficients. For m-bit data, if the data frequency is f_D, and n filter taps are required, then the number of m-bit multiply-accumulates per second is

$$N_{mult\text{-}acc} = n \times f_D$$

For data sizes and speeds ranging from 1 bit at 100 Hz (for sigma-delta A/D converters) to 8 to 12 bits at 40 MHz (HDTV video) and beyond, the architecture and implementation styles for FIR filters vary widely.

9.3.3 System Architecture

Figure 9.36 shows a block diagram of a typical ghost cancellation system. Analog baseband video is converted to digital form, and the synchronization and phase-locked sample clock are extracted. The digital video is fed to the ghost cancellation chip and to a DSP processor. The DSP processor examines a single captured TV line in which the training signal is embedded and runs an algorithm that calculates the filter coefficients required to cancel any imperfections in the transmission channel. These are downloaded into the ghost-canceller chip, and the ghosts are canceled. The output of the ghost-cancellation chip is fed to a D/A converter and hence to the baseband port of a TV receiver.

Micron Ex. 1037, p. 716
Micron v. YMTC
IPR2025-00119

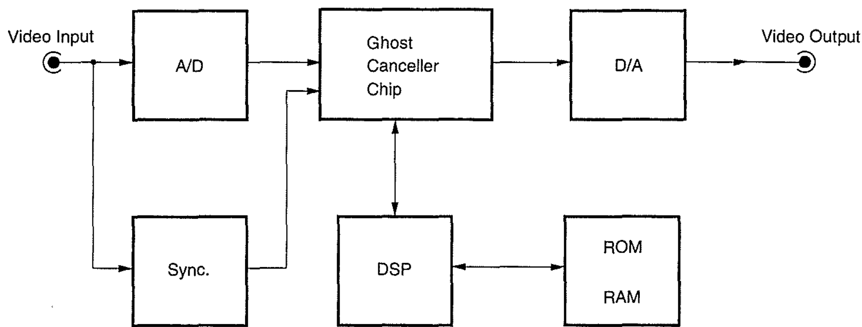


FIGURE 9.36 Ghost cancellation system

9.3.4 Chip Architecture

9.3.4.1 Filter Considerations

In this application, the following specifications are applicable:

Sampling rate	$= 4 \times$ Fsc
	= 14.32 MHz
Ghost delays	$= -10\ \mu s$ to $+40\ \mu s$
	= –150 samples to +580 samples
Coefficient precision	= 8 bits
Sample precision	= 8 or 9 bits

Ideally then, a total of 730 8-bit filter taps would be required. Each tap would require an 8-bit multiply-accumulate block operating at 14.32 MHz. As an example, a 50 MHz DSP processor (16 to 32 bits), could deal with ~3.5 taps. Based on some early area estimates, it was decided that a full implementation of 730 taps was too large to provide economic die sizes in current technology. In addition, most of the filter taps would be zero, making this direct implementation very inefficient from a hardware resource point of view.

It was realized that if each tap could be positioned independently within the time domain, the filter would require one tap for each nonzero filter coefficient. The disadvantage of this approach is that a separate delay line is required for each filter tap. An intermediate approach is to group a number of successive taps into sections, with a delay line for each section. This yields reasonably effective use of the taps, since a single ghost generally requires many taps to cancel it. This architectural trade-off was verified at Philips Research Labs with a prototype system with simulated and real echo situations. As a result, the number of taps required was reduced to around 150. This was the first step toward creating a practical implementation.

The second optimization to improve chip size involves the filter-tap design because it dominates the chip area. As a starting point we might consider an 8×8 multiplier-accumulator based on the designs given in Chapter 8.

Micron Ex. 1037, p. 717
Micron v. YMTC
IPR2025-00119

As a rough size estimate, an 8×8 Booth-recoded multiplier with an 18-bit accumulator would require 16 adders, 8 half-adders, and an 18-bit CPA adder. An alternative to a parallel multiplier is a word-serial multiplier. This requires four cycles to compute an 18-bit product for 8-bit coefficients and employs a single 18-bit adder. The former parallel implementation requires a cycle time of ~70 *ns*, while the latter requires a cycle time of 17.5 *ns*. While these decisions are justified within a few sentences, they actually required quite a bit of prototyping of various multipliers, which involved completing the layouts and simulating backannotated schematics with a transistor-level timing simulator. Simulation without the layout is usually not a good idea, because with compact structures, delays are normally due to self-loading and are hence highly affected by the actual layout.

9.3.4.2 Chip Overview

A block diagram of the chip is shown in Fig. 9.37. In essence it is the same structure as shown in Fig. 9.34(b). The main modules consist of the IIR and FIR filter sections, a 3-input adder, a main signal-variable-delay line, and various scaling and rounding logic.

The filter sections were divided into nine sections of twenty contiguous filter taps each, resulting in a total of 180 taps. Each section was designed so it could be part of the FIR- or the IIR-filter response. In addition, each section may receive an input that has a fixed delay of 0–128 samples for a section used in the FIR filter or 0–448 samples for a section used in the IIR filter. The signal may then be delayed by a 0–63 stage programmable delay line.

In the FIR and IIR filters, a section may be placed at an arbitrary temporal location by virtue of the programmable delay lines. Figure 9.38 shows an example where there are 40 FIR stages (Filter Blocks[8–7]) and 140 IIR stages (Filter Block[6–0]). The bold lines show the signal flow in this configuration.

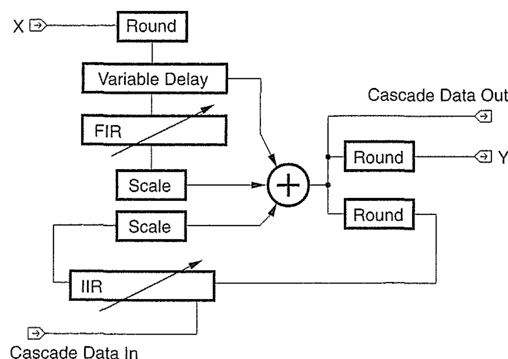


Figure 9.37 Ghost canceller chip architecture

Micron Ex. 1037, p. 718
Micron v. YMTC
IPR2025-00119

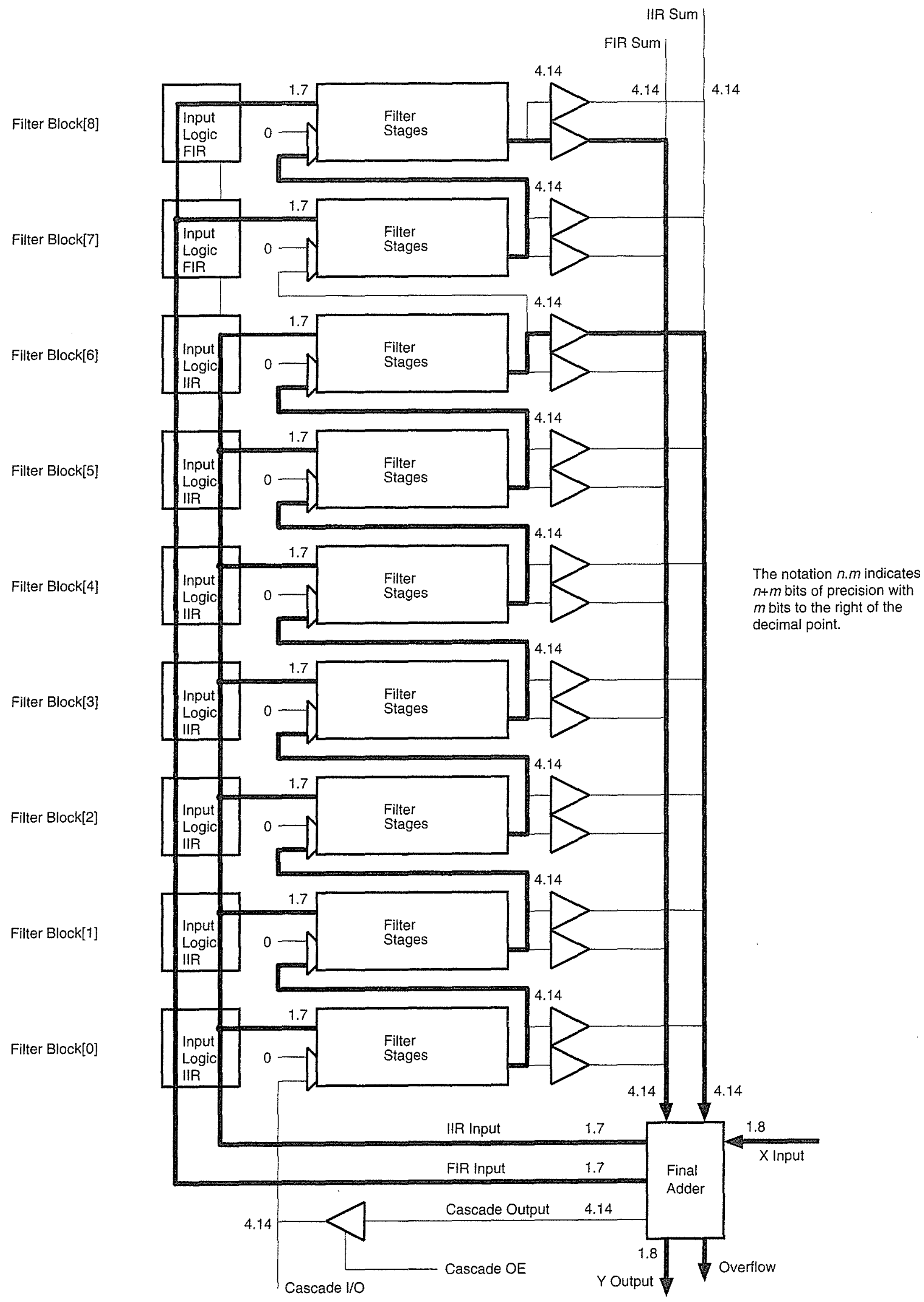


FIGURE 9.38 An example of use of the chip with 40 FIR stages and 140 IIR stages

Micron Ex. 1037, p. 719
Micron v. YMTC
IPR2025-00119

A 3-input adder is required to add in the main signal, the FIR signal, and the IIR signal. In addition, it allows cascading of more than one chip, and associated circuitry performs various rounding, limiting, and overflow-detection operations.

A main signal-delay line is used to delay the main signal (by convention the strongest) so that the FIR filter may be placed before this signal in time.

The control logic is required to interface with an external microprocessor to allow the configuration of the filters and the loading of coefficients.

A phase-locked loop is used to multiply a 4 × Fsc clock (14.32 MHz) by 4 to achieve the required clock frequency for the filter. A 2-phase clock generator uses the PLL clock or an external clock to generate a 57.28 MHz 2-phase clock for the chip.

9.3.5 Submodules

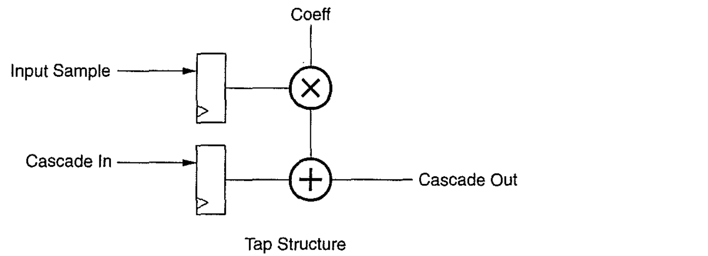
9.3.5.1 Filter Taps

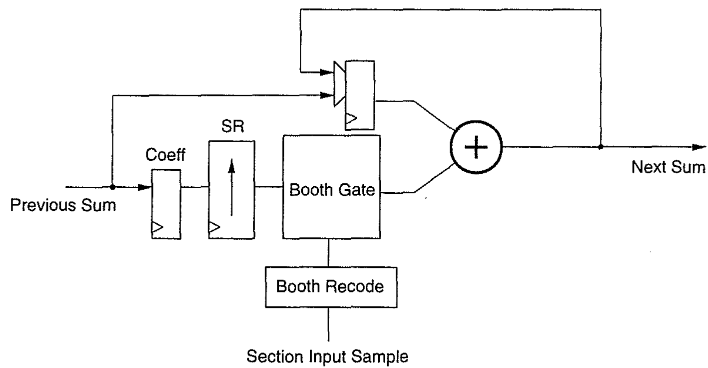
The basic filter tap structure is reviewed in Fig. 9.39(a). An 8-bit resolution and an accumulation to 18 bits was required. A number of filter architectures were investigated, resulting in a tap based on a serial Modified-Booth Recoded multiplier that requires four clock cycles to complete a multiply-accumulate.

This basic filter tap is shown at an RTL level in Fig. 9.39(b). It consists of an 18-bit adder and accumulator, a register to hold the coefficient, a shift register to shift the coefficient, a Booth recoder, and a Booth gate. Initially, the previous cascaded sum is loaded into the accumulator. On the next four cycles, the coefficient shift register shifts left by two bits in each cycle. The Booth recoder operates on the incoming sample and passes 0, 1, –1, 2, or –2 times the coefficient to the adder. The loading of the next tap accumulator may be pipelined in the last cycle so that only four clock cycles are required.

Clearly some choices have to be made between circuit implementations of elements such as adders, registers, and multiplexers. Three parameters drove this design—size, speed and power dissipation. Turning to the adder first, size constraints dictated a ripple-style adder (Figs. 8.6 and 8.7b). This style of adder is the smallest that can be built with decent performance. The registers could be static or dynamic. For size considerations, dynamic registers were selected except for the coefficient storage. This in part led to the next decision which was the 2-phase clocking strategy. Again consistent with reduced size and power consumption, n-channel pass gates with p-feedback inverters were used where possible. For instance, the logic that implements the Booth gating is shown in Fig. 9.40. As a matter of interest, one can compare this with the BUS-OP gate shown in Fig. 9.25(b) (which is a subset of this gate).

Micron Ex. 1037, p. 720
Micron v. YMTC
IPR2025-00119





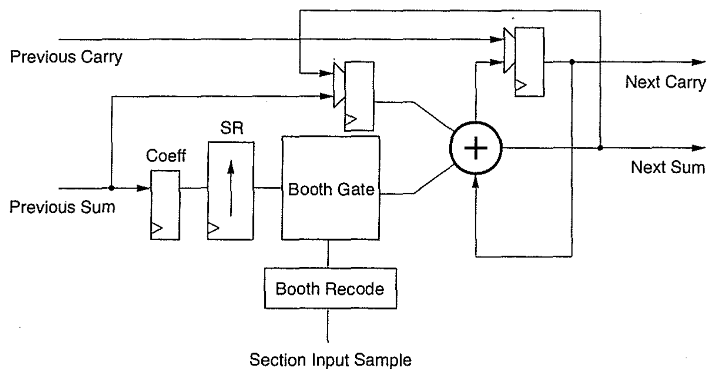


FIGURE 9.39 Tap design: (a) architecture; (b) RTL circuit; (c) carry save version

Why was one design used in one instance and not in the other? In the design in Fig. 9.40 area was of paramount importance, so the extra design time spent in verifying the correct operation of this gate (it is a ratioed gate) was well spent. In the processor example, the BUS-OP gate is one of many different modules that had to be designed, so a "fire and forget" philosophy—that is, an approach that is guaranteed to function correctly and whose speed is verified during the normal course of transistor-level timing analy-

Micron Ex. 1037, p. 721
Micron v. YMTC
IPR2025-00119

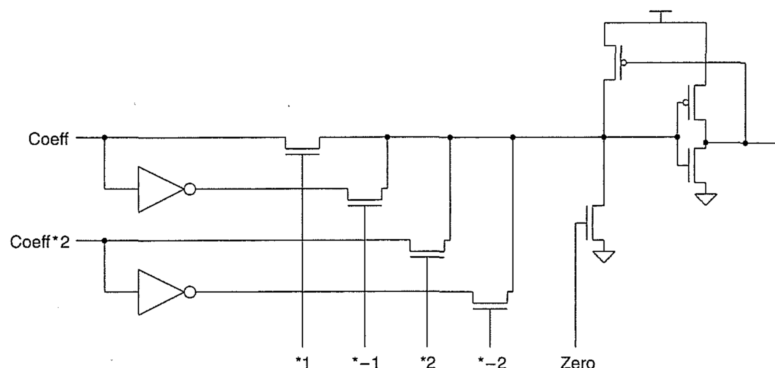


Figure 9.40 Booth gate

sis—was used. If no transistor-level tools were available, a gate-level implementation (XOR and AND) gate would be used. This microexample provides a model for the kinds of optimization that may occur in a CMOS VLSI design. It is important to achieve a balance between optimization and project completion time.

Power dissipation estimations showed that a 5-volt part would dissipate around 2–3 watts. While this is satisfactory for a ceramic package, something nearer one watt was required for a plastic package. Thus the decision was made to operate the chip at either 3.3 volts or 5 volts. As a result it was found that an 18-bit ripple adder was marginal at the slow corner of the 3.3-volt operating conditions (110°C, 3.0 volts). This forced a reevaluation of the ripple-adder strategy and was solved by employing a carry-save approach. This is shown in Fig. 9.39(c). While this increases the overall power dissipation somewhat, it allows operation at the lower voltage, which resulted in a power dissipation near one watt for a 3.3-volt operation. Both sum and carry are pipelined, resulting in a critical path that is close to a clock-to-Q delay, a worst-case adder delay, and a register setup delay. In the process in which this chip was implemented, this was below 5 *ns*. A side benefit of this architecture is that it is capable of much higher frequency operation, thus extending the application of the basic filter to HDTV video rates.

Figure 9.41 shows the final circuit diagram for a single bit of the filter tap. The complete filter tap consists of 18 of these bits with a clock driver and qualification block driving this datapath. An example of the symbolic layout of the datapath is shown in Fig. 9.42 (also Plate 11). Typical simulations that would be run on this datapath would include speed tests to ensure that the stage operated at 17.5 *ns* at all process corners. In addition, the power dissipation could be estimated to aid in overall chip planning and power distribution.

The basic floorplan of a section is shown in Fig. 9.43. A U-shaped structure, ten taps wide, is used so that the input and output from the section are

Micron Ex. 1037, p. 722
Micron v. YMTC
IPR2025-00119

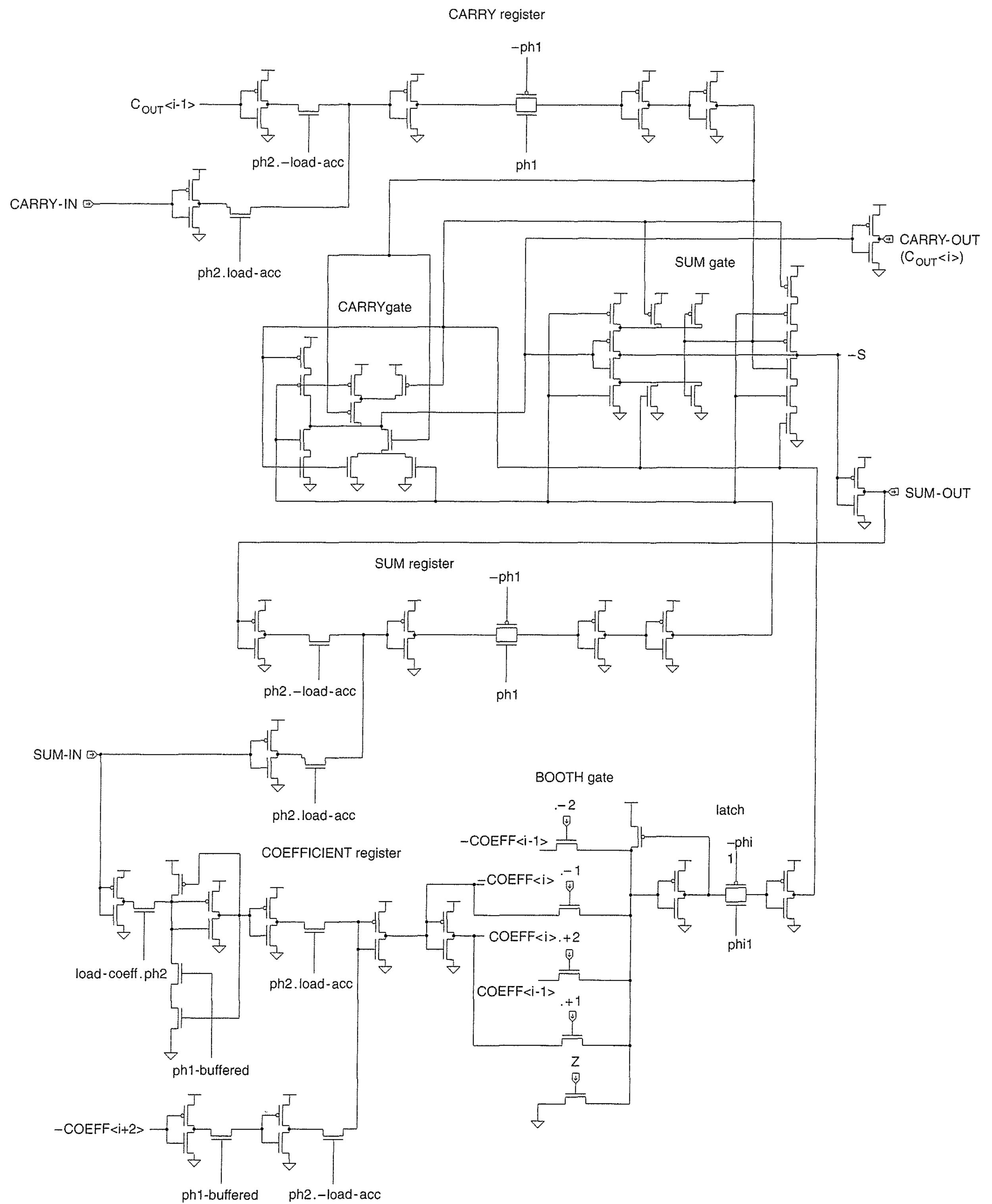


FIGURE 9.41 Tap circuit

Micron Ex. 1037, p. 723
Micron v. YMTC
IPR2025-00119

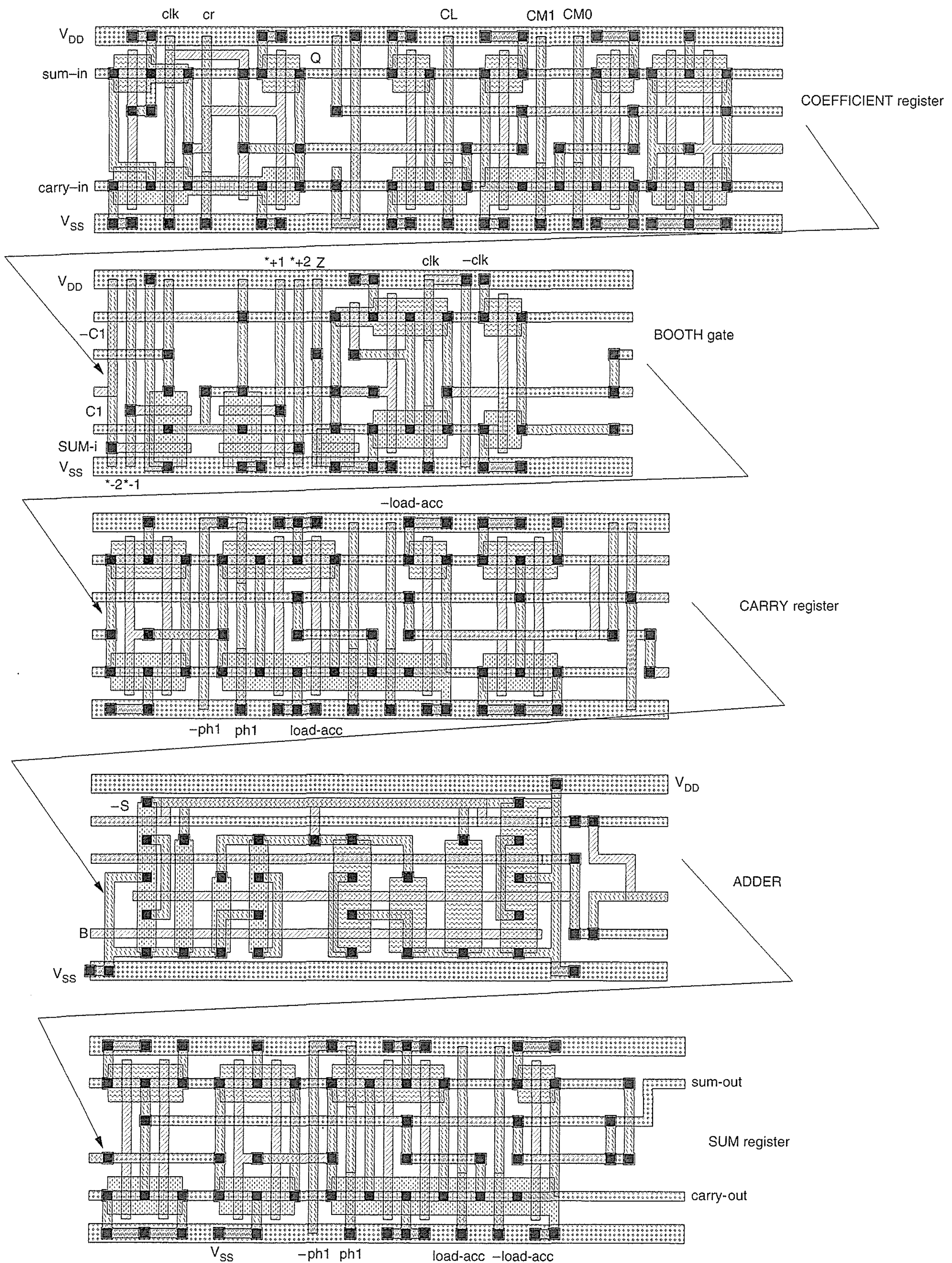


FIGURE 9.42 Representative tap layouts

Micron Ex. 1037, p. 724
Micron v. YMTC
IPR2025-00119

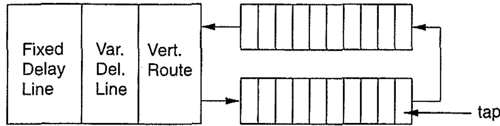


FIGURE 9.43 Section floorplan

available at a central routing trunk. Because the Booth-recoding logic is common to all taps, one recoder is used for all twenty taps in a section. An I/O section provides for driving the output of a section onto global IIR-sum and FIR-sum busses. Each section is connected to its neighbors via a cascade connection.

9.3.5.2 Delay Lines

There are at least two approaches to implementing the variable delay lines: a multiport RAM and shift register. Although the per-bit area of the RAM implementation is smaller than the shift register, the shift-register approach was chosen because it provided a more compact floorplan when considered in concert with the layout of the sections.

The programmable delay line for each section is placed at the left of each section (Fig. 9.43). To the left of the variable delay line, the fixed-delay block is situated (Fig. 9.43). This consists of the actual delay line and a distributed multiplexer. The distributed multiplexer consists of 11 8-bit busses that run vertically for the height of the nine sections. Each fixed delay takes its input from one of these busses and outputs to another. The multiplexer consists of tristate buffers placed under the vertical routing.

The shift-register bit is a simple 2-phase dynamic register and is shown in Fig. 8.70(b), and the variable delay line is shown in Fig. 8.70(a).

9.3.5.3 Phase-locked Loop- and Clock-generation

The Phase-Locked Loop is a charge-pump type PLL[8,9]; it was first introduced in Fig. 5.61 and is repeated in Fig. 9.44. A divide-by-4 counter in the feedback loop provides one 16 × Fsc clock for a 4 × Fsc input-clock. The phase detector measures the difference between the PLL VCO frequency (divided by 4 in this case) and the incoming reference frequency (4 × Fsc—14.32 MHz). The phase

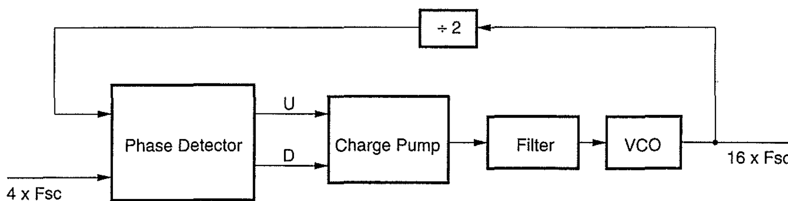


FIGURE 9.44 PLL

Micron Ex. 1037, p. 725
Micron v. YMTC
IPR2025-00119

detector produces a sequence of UP or DOWN pulses, which are used to switch a charge pump. The charge pump either charges or discharges a capacitor with voltage or current pulses, as determined by the phase detector. A filter is used to limit the rate of change of the capacitor voltage, and the result is a slowly rising or falling voltage that depends on the frequency difference between the PLL VCO and the reference frequency. The VCO increases or decreases its frequency of operation as the control voltage is increased or decreased. Together, the components form a closed-loop feedback system whose phase and frequency response are determined by the characteristics of the charge pump, the filter, and the VCO.

The phase-detector implementation is shown in Fig. 9.45. This is a conventional phase detector implemented in static CMOS logic. If F_2 falls before F_1 falls, the signal *DN* is asserted. If F_1 falls before F_2 falls, *UP* is asserted.

The charge pump is shown in Fig. 9.46. The charge pump feeds pulses of current to a filter and capacitor, which has the effect of charging the capacitor up or down. This results in a voltage that rises or falls, and controls the VCO, either increasing or decreasing the frequency. The charge pump consists of a resistively biased constant current source (N_1, N_2, N_3) with n (N_4) and p (P_1) current mirror sources. These feed current mirror transistors N_5 and P_2. These in turn are switched to the filter via CMOS transmission gates, with complementary clocks balanced for equal delay. The current-source transistors are double the minimum length to improve the drain conductance.

Because the filter had to be monolithic (with no external components), an RC filter was constructed from MOS transistors. A CMOS transmission gate is used as a resistor, and MOS transistors are used as capacitors (Fig. 9.47a).

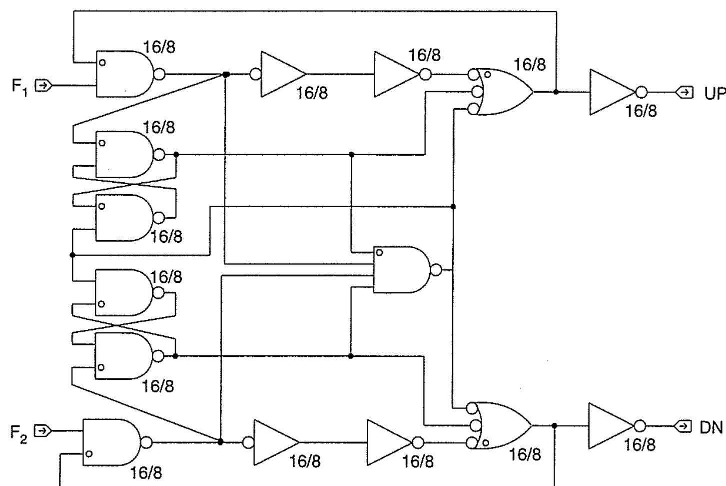


Figure 9.45 Phase detector

Micron Ex. 1037, p. 726
Micron v. YMTC
IPR2025-00119

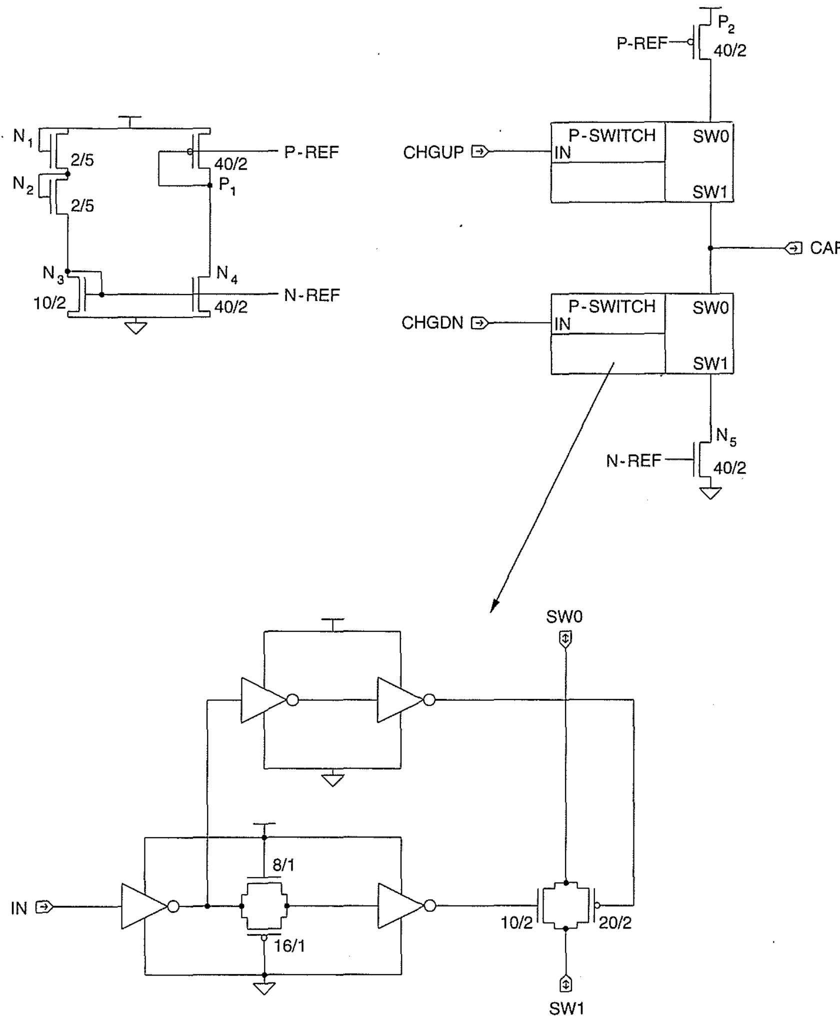


FIGURE 9.46 Charge pump

These are not ideal components but are adequate for this application. P-diffusion resistors might also be used for the resistor, while the capacitors can be poly-poly capacitors if a two-poly process is available. Alternatively, the filter can be implemented off-chip at the expense of a pad and of possible noise injection. The equivalent RC filter is shown in Fig. 9.47(b).

The VCO consists of 13 stages of a current-starved oscillator with a buffered output (Fig. 9.48). This was chosen because it had a wide range of operation and was verified to operate correctly over all process corners. The main parameter of interest for the VCO is the frequency range and the oscillator sensitivity in terms of MHz/volts. The transistors were made larger than minimum to reduce the effects of geometry biases and to swamp the routing

Micron Ex. 1037, p. 727
Micron v. YMTC
IPR2025-00119

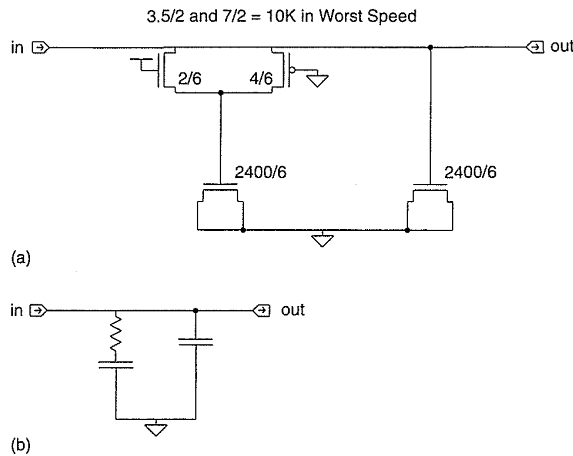


Figure 9.47 Filter

capacitance (thereby ensuring high-speed performance). The n- and p-transistors are sized to achieve equal rise and fall time.

The filter values and the VCO sensitivity vary widely over process corners. The lockup times for the VCO have a much longer time constant than the VCO itself. Thus it is time consuming (and possibly impractical) to completely verify the VCO using circuit simulation (SPICE). In order to ensure that the chosen parameters would work over the process corners, a simple analytical model was constructed that modeled the PLL. This was used to verify the VCO lock times. Various of these functional simulations were spot-checked against full circuit simulations for accuracy in the analytical model.

The output of the PLL is fed to the 2-phase clock generator, which is shown in Fig. 9.49. This consists of a conventional cross-coupled 2-phase clock generator with the final driver transistors being 4000μ (p) and 2000μ (n) wide. The clock load was 100pF per phase. A multiplexer allowed the on-chip PLL to be bypassed. This was done as a system requirement and also as a safety mechanism in case the analog PLL was inadequate (it wasn't).

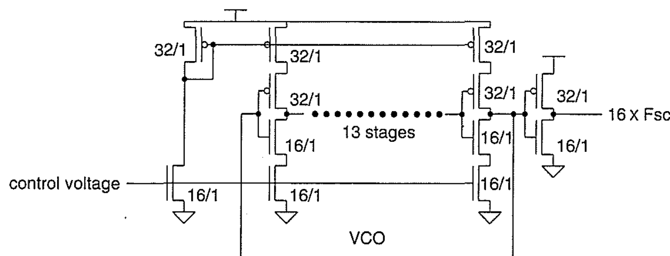


Figure 9.48 VCO

Micron Ex. 1037, p. 728
Micron v. YMTC
IPR2025-00119

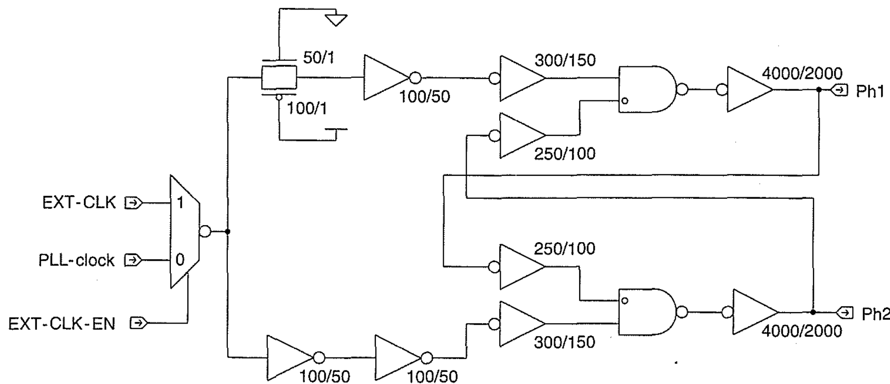


FIGURE 9.49 Clock generator

9.3.5.4 Peripheral Processing

A number of support modules surround the filters. These include

- a microprocessor interface.
- the final filter adder.
- an I/O block.

The microprocessor block is used to load the filter-block coefficients, configure the filter sections, and monitor filter overflow. The I/O block is responsible for converting the filter output to signed or unsigned form and for managing the I/O communication busses for multiple chips. The final adder combines the main pulse of the FIR with the output of the FIR and IIR filters. In addition it rounds the filter output to 8 or 9 bits and can scale the outputs or the FIR and IIR filters by factors of 2. In many ways it is this kind of "glue" and support logic that can dominate the design after the core signal processing section has been designed. One must always remember to count in the "control" portions of a design when estimating design times.

9.3.6 Power Distribution

At 5-volt operation, the chip dissipates around 2.5 watts and draws 500 mA. On average this is 55 mA per section with the peak current being much higher than this. To both achieve metal migration requirements and reduce switching noise, each section was effectively provided with a power and ground pad, shown in Fig. 9.50(a). These connections were routed in metal3. In addition, to reduce power-supply noise on-chip bypass capacitors were placed under the power lines. These consist of large gate-area n-transistors, with their gates connected to V_{DD} and source and drains connected to V_{SS} (substrate).

Micron Ex. 1037, p. 729
Micron v. YMTC
IPR2025-00119

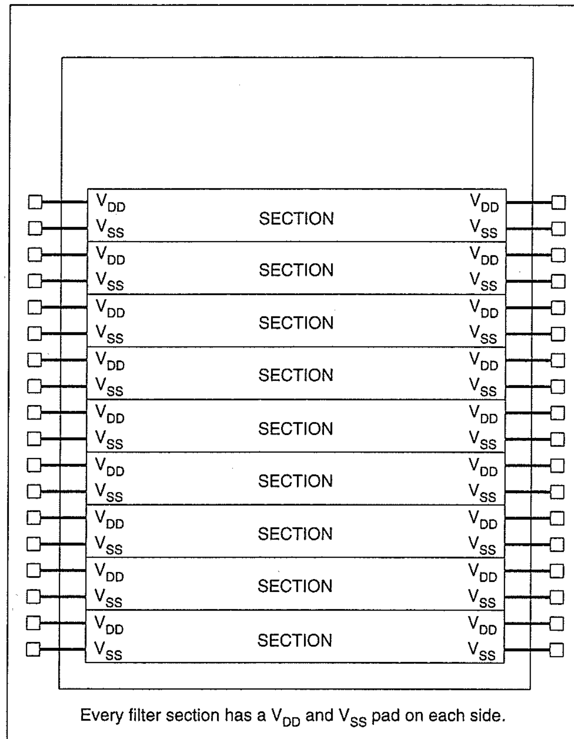


Figure 9.50 Power distribution to chip

9.3.7 Chip Floorplan

The overall chip floorplan is shown in Fig. 9.51(a) (also Plate 12). The sections occupy the majority of the chip. The final datapath and I/O sections are at the top of the design. The clock driver is placed in the pad ring at the top left, while the PLL is placed in the pad ring at the top right. The PLL has its own 5-volt supply.

Figure 9.51(b) shows the bond diagram for the chip for a 144-pin plastic PGA package. This is a step of the design process in which the placement of

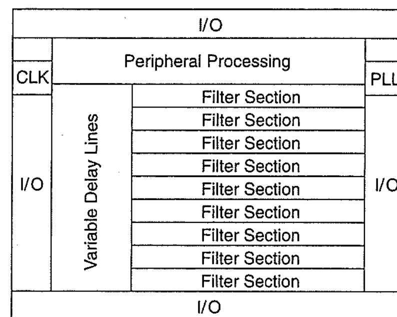
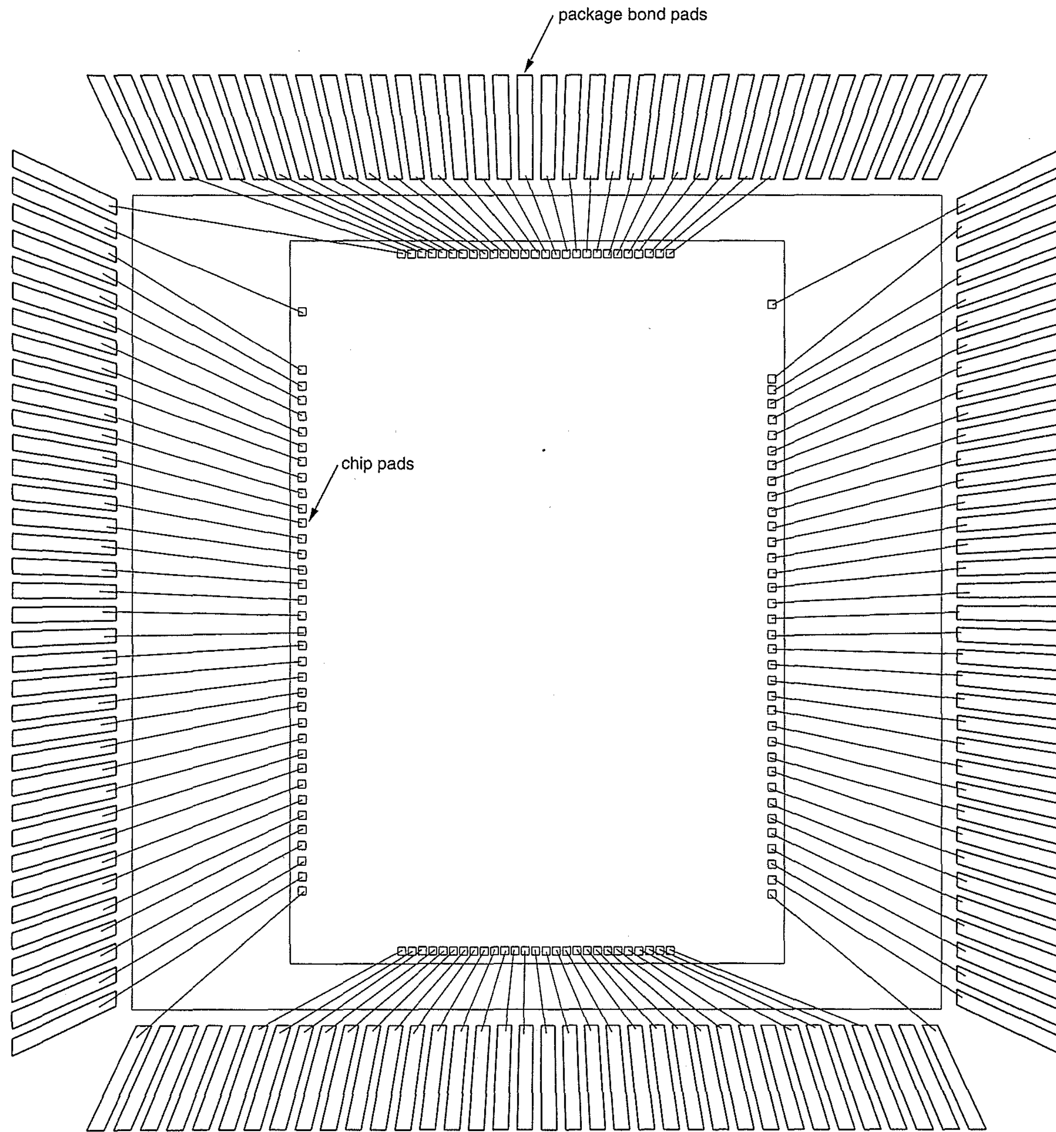


Figure 9.51 Ghost chip: (a) floorplan; (b) bond diagram

Micron Ex. 1037, p. 730
Micron v. YMTC
IPR2025-00119



(b)

FIGURE 9.51 *(continued)*

the I/O pads of the chip and the package bond leads are checked for bond-lead compliance. Usually, this means checking for correct bond angles (the angle of the bond wire with respect to the center of the package, and checking that bonds do not cross each other).

Micron Ex. 1037, p. 731
Micron v. YMTC
IPR2025-00119

9.3.8 Testing and Verification

Similarly to the testing of the RISC microcontroller, testing of this chip was completed at various levels. Low-level SPICE simulations were completed for key modules, such as the adder bit and registers. Timing simulation was completed at the section level with backannotated schematics. Full chip simulation was completed at the gate level using a unit-delay logic simulator. Functional models were also written for a filter tap, and simulations were completed at a mixed transistor/functional level. The Lisp functional model for a tap is shown below.

```
(deffunctional-model csa-filter-stage-dp
 :inputs (("sum-in<17:0>" :capacitance .11)
        ("carry-in<17:0>" :capacitance .09)
        (Z :capacitance .26)
        (*+1 :capacitance .26)
        (*+2 :capacitance .26)
        (*-1 :capacitance .26)
        (*-2 :capacitance .26)
        (load-coeff :capacitance .08)
        (-load-acc :capacitance .18)
        (ph1 :capacitance .18)
        (ph2 :capacitance .26))
 :outputs (("sum-out<17:0>" :capacitance .08)
        ("carry-out<17:0>" :capacitance .11))
 :local-state ((*-2or*-1 :initform 'X)
           (creg-master :initform '0) (creg :initform '0)
           (sreg-master :initform '0) (sreg :initform '0)
           (recode-reg :initform '0)
           (coeff-reg :initform '0)
           (coeff-shifter :initform '0)
           (coeff-shifter-reg :initform '0))
 :model (progn
; most outputs advance on phase1
           (when (eq ph1 1)
           (setq *-2or*-1 (sim-or *-1 *-2)
             creg creg-master
             sreg sreg-master
             coeff-shifter (if (eq coeff-shifter-reg 'X) 'X
               (lsh coeff-shifter-reg 2)))
;; recoding the coefficient
           (setq recode-reg
             (cond
               ((eq Z 1)   0)
               ((eq coeff-shifter-reg 'X) 'X)
               ((eq *+1 1) coeff-shifter-reg)
               ((eq *-1 1) (logxor coeff-shifter-reg #o777777))
               ((eq *+2 1) (lsh coeff-shifter-reg 1))
            ((eq *-2 1)(logxor(lsh coeff-shifter-reg 1) #o777777))
               (t 'X))))
;; compute sum/carry output bits of the adder
```

Micron Ex. 1037, p. 732
Micron v. YMTC
IPR2025-00119

```
    (if (or (eq recode-reg 'X)(eq sreg 'X)
          (eq creg 'X)(eq *-2or*-1 'X))
          (setq sum-out   'X
              carry-out 'X)
          (loop with sum = 0 and carry = 0
              for i below 18.
              for coeff-bit = (ldb (byte 1 i) recode-reg)
              for sum-bit = (ldb (byte 1 i) sreg)
              for carry-bit first *-2or*-1 then
                 (ldb (byte 1 (1- i)) creg)
              for adder = (+ coeff-bit sum-bit carry-bit)
             do (setq sum (dpb (logand adder 1) (byte 1 i) sum)
                    carry (dpb (lsh adder -1) (byte 1 i) carry))
            finally (setq sum-out sum
            carry-out carry)))
;; phase 2 clock asserted
          (when (eq ph2 1)
;; coefficient load cycle
            (when (eq load-coeff 1)
              (setq coeff-reg sum-in))
;; accumulator-load cycle
          (case -load-acc
;reload coefficient
        · (0 (setq coeff-shifter-reg coeff-reg
                 creg-master carry-in
                 sreg-master sum-in))
;shift coefficient
          (1 (setq coeff-shifter-reg coeff-shifter
                 creg-master carry-out
                 sreg-master sum-out))))
          )
 :delays
    ((ph1↑ sum-out↑ :delay 11.4 :driver-size 12/1)
    (ph1↑ sum-out↓ :delay 11.4 :driver-size 8/1)
    (ph1↑ carry-out↑ :delay 11.4 :driver-size 12/1)
    (ph1↑ carry-out↓ :delay 11.4 :driver-size 8/1))
:timing-constraints ((-load-acc ph2↑:setup 1.0 :hold 3.0)
          (Z  ph1↑ :setup 8.4 :hold 3.0)
          (*+1  ph1↑ :setup 8.4 :hold 3.0)
          (*-1  ph1↑ :setup 8.4 :hold 3.0)
          (*+2  ph1↑ :setup 8.4 :hold 3.0)
          (*-2  ph1↑ :setup 8.4 :hold 3.0)
          (sum-in  ph2↑ :setup -3.0 :hold 3.0)
          (carry-in  ph2↑ :setup -3.0 :hold 3.0)
          (load-coeff ph↑ :setup 3.25)) ;write pulse setup
)
```

At the start of the model, inputs and outputs are defined by the keywords `:inputs` and `:outputs`. Each input is denoted by a name and a load capacitance, while each output has a name, a capacitive load, and a drive strength. The `:local-state` keyword denotes the internal registers in the model and their initialization values. The functionality is specified within the

Micron Ex. 1037, p. 733
Micron v. YMTC
IPR2025-00119

`:model` section. Finally, the `:delay` section specifies important signal-to-signal delays, while the `:timing-constraints` section specifies `:setup` and `:hold` times for the registers in the design. The timing values are derived from SPICE or timing simulations. The model above is useful for both simulation and timing analysis. In the latter case, the modularity of the tap is well specified by the I/O and timing specifications that are included. Extensive timing analysis was completed, using backannotated schematics with parasitics extracted from compacted mask layouts.

The critical path in this design consisted of the clock generator and distributed skew in the clock lines (remember that the design also had to operate at 3.3 volts).

9.3.9 Summary

This section has presented an example of a CMOS VLSI design that employs a high degree of regularity. Accordingly, the design style changes from the processor design, with much more emphasis being placed on circuit and layout design for the key replicated cells. A key talent in CMOS-system design is knowing when and where to optimize. For example, had the ghost-cancellation chip been implemented as a standard-cell or gate array, it would have been from 4 to 10 times larger, resulting in a chip cost that would have made the implementation economically infeasible.

9.4 A 6-bit Flash A/D†

9.4.1 Introduction

This final example has been included as an example of a simple analog circuit that is almost digital (hence its inclusion in a digital CMOS text). The circuit is a 6-bit A/D converter implemented as a "flash" converter. This provides for a very fast converter at the expense of area. Although limited to about 8 bits of resolution, as CMOS circuits are becoming smaller over time, this circuit architecture is also becoming smaller, and faster. This style of converter or variants is of particular use at video-sample rates.

Figure 9.52 shows the basic architecture. An analog input is presented to a sample-and-hold circuit, which feeds one input of 2^N comparators, where N is the desired digital precision. A clock input samples the input and strobes the sample-and-hold. The other input to the comparators is connected to a resistor string connecting a reference voltage ($+V_{REF}$ and $-V_{REF}$). For a given input voltage, after the sampling process and the comparators have

†Designed by N. Weste.

Micron Ex. 1037, p. 734
Micron v. YMTC
IPR2025-00119

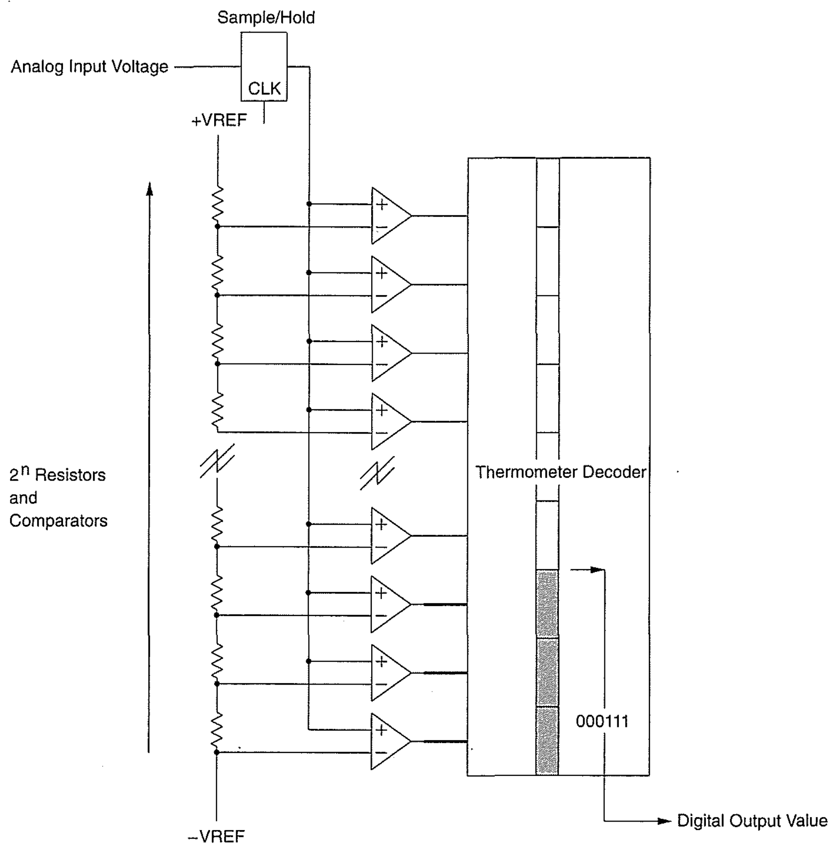


FIGURE 9.52 Flash A/D architecture

switched, the output of the comparators is a string of 1s where the highest 1 represents the highest comparator that switched. This is represented in the diagram by a gray bar. A decoder may be used to convert this "thermometer code" to an N-bit number.

9.4.2 Basic Architecture

A CMOS implementation of a 6-bit flash A/D converter is shown in Fig. 9.53. It consists of a polysilicon resistor string, 64 sampling comparators, 64 registers, and a thermometer decoder that consists of 64 3-input NOR gates

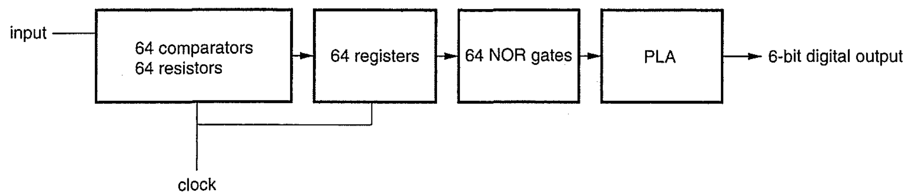


FIGURE 9.53 Flash circuit

and a 64-term PLA. The analog parts that have to be designed include the resistor string and the comparators. The digital parts consist of the register, the NOR gate, and the PLA.

9.4.3 Resistor String

The resistors may be implemented as diffusion resistors, polysilicon resistors, or metal in very high speed flash converters. In this design, polysilicon resistors were used. The value of the resistor and the reference voltage determine the DC current drawn by the reference ladder. Choosing the value of the resistors is a trade-off between limiting the DC power dissipation and achieving a low impedance reference to supply the comparator. The value chosen in this design was 20 Ω, which is approximately 1 square of polysilicon. This yields a nominal resistance of 1280 Ω for the string. For a 1-volt reference the current drawn is approximately 0.8 mA.

9.4.4 The Comparator

The comparator is shown in Fig. 9.54. It consists of two cascaded, capacitively coupled, auto-zeroed inverters[10] followed by a dynamic register. During the sample time, the input value is stored on the capacitor C_1 via pass gate P_1, while the inverters are auto-zeroed via pass gates P_3 and P_4. The auto-zero step is used to reduce the effect of any offset voltages present in the comparator. The transistors in each inverter have slightly different characteristics (such as threshold voltage and beta). This in turn leads to slightly different transfer characteristics. If the inverters were not auto-zeroed, the comparators would switch at slightly different voltages. The auto-zero step reduces this offset to below the precision of the converter.

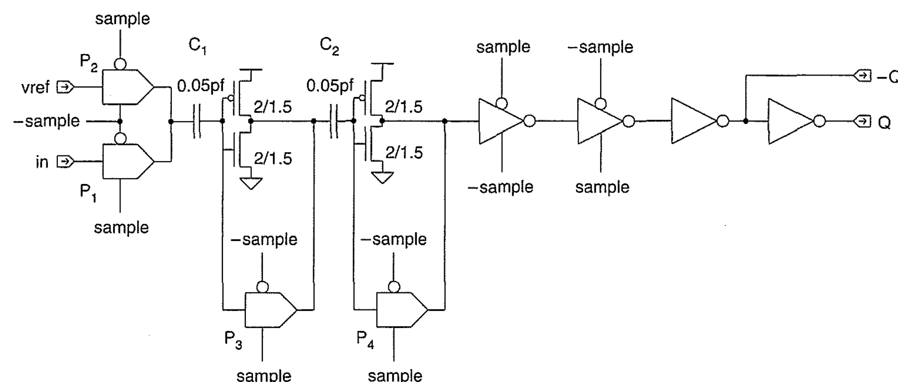
When *sample* is false, the reference input is connected to the capacitor, thereby transferring charge to or from the capacitor. This causes a voltage change at the input of the first comparator, which causes the output of the comparator to rise or fall by an amount proportional to the gain of the inverter (and the capacitive divider formed by C_1 and the input capacitance of the inverter). This signal is further amplified by the second comparator. This is then passed to a register.

As stated, the gain of the comparator depends on the gain of the inverter and the ratio of $C_1(C_2)$ to the input capacitance of the inverter. The input capacitance of the inverter is approximately two minimum-size transistor gate capacitances. In a two-polysilicon process, the coupling capacitor (C_1) can be made large compared to the gate capacitance. However in a single poly process, this capacitor has to be made from a relatively low capacitance

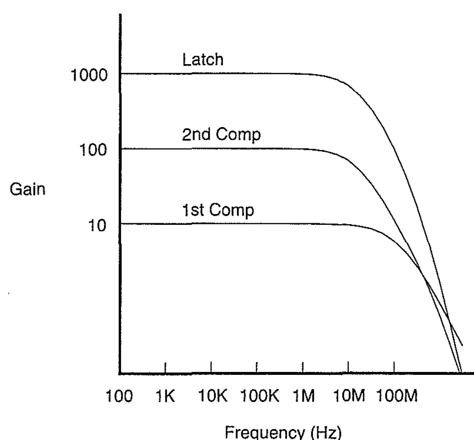
Micron Ex. 1037, p. 736
Micron v. YMTC
IPR2025-00119

poly-metal-metal2 sandwich (poly and metal2 connected to one plate, metal to the other). In this design this capacitor was made $.05pF$. The combined gain of the capacitor and an inverter was around 10 (the capacitor value used roughly halves the gain). Thus two comparators result in a gain of about 100. Figure 9.54(b) shows the frequency response of two capacitively coupled comparators. This would indicate a sensitivity of around 10mV. In practice the converter demonstrated 6-bit resolution with an input reference of 600mV, which agrees with these simulated results.

The transistor size and n/p β ratio of the inverter transistors affect the DC-transfer characteristic (gain, linearity, and dynamic range) and the AC performance (bandwidth and phase response). A design often involves mak-



(a)



(b)

FIGURE 9.54 Comparator: (a) circuit; (b) frequency response

Micron Ex. 1037, p. 737
Micron v. YMTC
IPR2025-00119

ing a compromise between conflicting requirements. For instance, the inverter gain is improved by using transistors with slightly longer than minimum length at the expense of input-loading capacitance. Increasing the width of the p with respect to the n moves the quiescent point ($V_{in} = V_{out}$) of the DC-transfer characteristic toward midrail but increases the input capacitance, which reduces the gain. The main point to note here is that with this extremely simple analog circuit, a relatively large effort goes into the design of one inverter compared with that of a logic inverter.

The maximum sample rate of the converter is dependent on the bandwidth of the comparators (and reference resistor string) and the maximum clock frequency of the digital circuitry. A SPICE-frequency analysis (Fig. 9.54b) revealed that the comparator used here (for the process used) had a 3dB bandwidth of 10 MHz. Measured devices were successfully operated at 10 MHz with supply voltages of both 3 and 5 volts.

Complementary transmission gates are used as analog switches. Charge injection occurs when a changing gate signal couples charge to the source/drain node via C_{gd} or C_{gs}. This can be minimized by using complementary switches and delay-balanced clocks. The sample clock and its complement are buffered with an equalized delay clock generator (shown in Fig. 5.52b). The CMOS switches have to be made large enough to achieve speed goals.

The final piece of circuitry in the comparator is a dynamic register. This consists of two cascaded tristate inverters followed by a pair of buffer inverters.

9.4.5 Thermometer Code Logic

The output of the comparators is a thermometer code, which has to be converted to a binary number. This is achieved using a logic gate, which checks for a 011 code using a 3-input NOR gate (Fig. 9.55). This indicates the upper boundary of the comparator output. This is then passed to a PLA-style decoder, which has a term for every comparator. For instance, the 8th bit is shown. When the signal at the output of the NOR gate is asserted, the PLA NOR gate transistors are turned on, causing the code 000111 to appear at the output. The PLA is implemented with a pseudo-nMOS NOR gates.

9.4.6 Floorplan and Layout

A basic cell horizontally joins a resistor, two comparators, the register, and the thermometer gate. This structure is then arrayed 64 times vertically. The decoder PLA is abutted horizontally on the right of this structure, and clock and I/O buffers are placed on the top and bottom of the structure. This floor-

Micron Ex. 1037, p. 738
Micron v. YMTC
IPR2025-00119

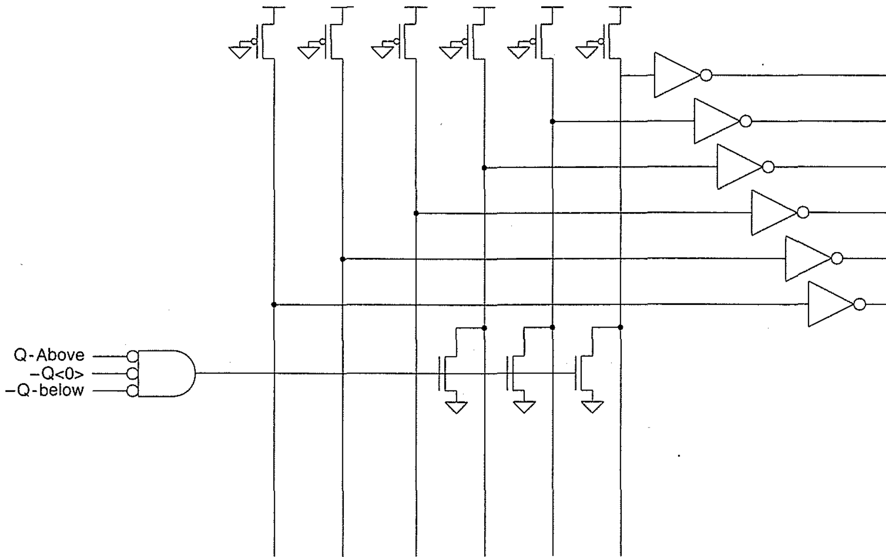


FIGURE 9.55 Thermometer decode logic

plan is shown in Fig. 9.56. A generator was written to automatically generate the layout for a given resolution. A chip micrograph is shown in Plate 13.

A portion of a 2-bit section of the converter layout is shown in Fig. 9.57. The resistor string may be seen at the bottom, with the resistors placed vertically. The input/reference switches are above the resistors. The comparator capacitor is the long structure in the center of the layout. The capacitor is

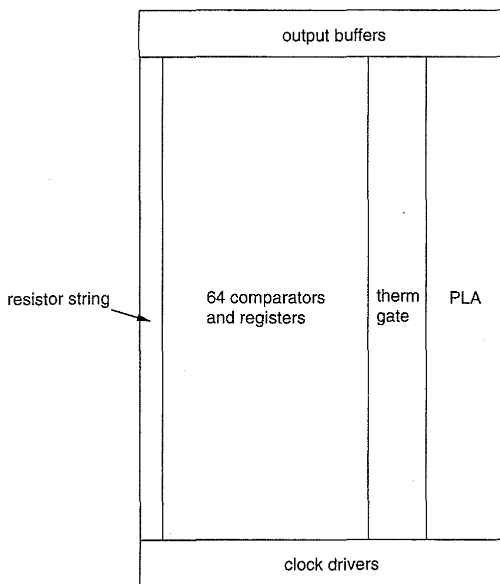


FIGURE 9.56 Floorplan of A/D

Micron Ex. 1037, p. 739
Micron v. YMTC
IPR2025-00119

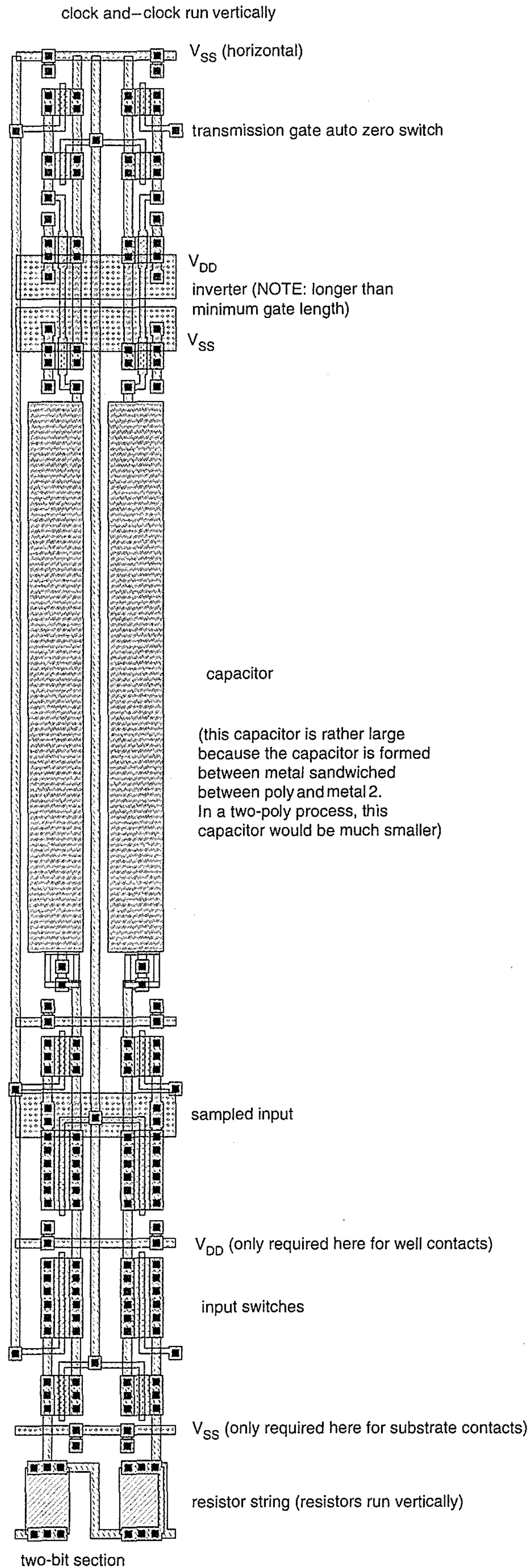


FIGURE 9.57 Partial cell layout for A/D

Micron Ex. 1037, p. 740
Micron v. YMTC
IPR2025-00119

about 100μ long. This would be much smaller in a double polysilicon process. Above the capacitor is the comparator inverter and auto-zero switch. Another comparator, register, NOR gate, and PLA gate are placed above this section in the converter.

9.4.7 Summary

A modest implementation of a flash A/D converter has been presented in order to illustrate some of the issues that are addressed as we leave the digital world. There is a wealth of literature on A/D conversion, and the reader is encouraged to investigate these sources if this subject is of interest. For a more efficient 8-bit A/D converter employing a similar technique see Dingwall and Zazzu[11] and Tsukada et al.[12]

9.5 Summary

This chapter has presented three case examples of CMOS designs. The first described a contemporary RISC microcontroller with a mix of datapath, memory, and control logic. This design could be implemented in a wide range of CMOS logic styles and design methods. The second described a high-performance signal-processing circuit that is representative of video-rate architectures. To achieve commercial viability this design required custom layout and innovative architecture and circuit design. The final design featured a straightforward flash A/D converter representing the interface between the analog world and the CMOS digital domain. In this design the focus extends to basically one inverter which is extensively simulated. From these examples it may be seen that the more complex a system becomes, the less time is available to spend on low-level details; notwithstanding, it is possible to create denser, faster designs in a given technology if the appropriate amount of design effort is invested. In these days of short product cycles, time to market is almost always the dominant concern. This leads to the requirement for short design times. This in turn is achieved by the use of highly automated design systems, the use of libraries, and the reuse of other components of interest.

9.6 Exercises

1. In order to achieve a short cycle time, the RISC microcontroller in Section 9.2 used four pipeline stages. Redesign the processor to run in a single (albeit longer) cycle. How does each module change? What simplifications may be made?

Micron Ex. 1037, p. 741
Micron v. YMTC
IPR2025-00119

2. If the critical path of the processor is that shown in Section 9.2.3.1.3, what changes to the architecture would you make to improve this speed?
3. Sketch out the logic design for a multiplication-function block that can be attached to the processor in the EXT_BUS_DP module. (Could you implement multiplication on the processor as described?)
4. Examine alternative architectures to implement the FIR/IIR filters in the ghost cancellation chip (i.e., 8×8 multiplier running 4 taps).

9.7 References

1. C. Sechen and A. Sangiovanni-Vincentelli, "TimberWolf3.2: a new standard cell placement and global routing package," *Proceedings of the 23rd Design Automation Conference,* 1986, Las Vegas, Nev., pp. 432–439.
2. Bruce Edwards, Alan Corry, Neil Weste, and Craig Greenberg, "A Single Chip Ghost Canceller," *Proceedings of the IEEE 1992, Custom Integrated Circuits Conference, May 1992,* pp. 26.5.1–26.5.4.
3. Walter Ciciora, Gary Sgrignoli, and William Thomas, "A tutorial on ghost cancelling in television systems," *IEEE Transactions on Consumer Electronics,* vol. CE-25, Feb. 1979, pp. 9–44.
4. Stephen Herman, "The development of commercial echo cancellers for television," *National Association of Broadcasters, 1991 Broadcast Engineering Conference Proceedings,* pp. 102–106.
5. David Koo, "Developing a new class of high energy ghost cancellation reference signals," *International Conference on Consumer Electronics, Digest of Technical Papers,* Jun. 1992, Boston, Mass., pp. 76–77.
6. Craig B. Greenberg, "Ghost cancellation system for high energy GCE," *International Conference on Consumer Electronics Digest of Technical Papers,* June 1992, Boston, Mass. pp. 78–79.
7. H. Samueli, "An improved search algorithm for the optimization of the FIR filter coefficients represented by a canonic signed digit code," *IEEE Transactions on Circuits and Systems,* vol. CAS-34, Sept. 1987, pp. 1192–1202.
8. F. A. Gardner, "Charge-pump phase-locked loops," *IEEE Transactions in Communications,* vol. COM-28, Nov. 1980, pp. 1849–1858.
9. Deog-Kyoon Jeong, Gaetano Borriello, David A. Hodges, and Randy H. Katz, "Design of PLL-based clock generation circuits," *IEEE Journal of Solid State Circuits,* vol. SC-22, no. 2, Apr. 1987, pp. 255–261.
10. Andrew G. F. Dingwall, "Monolithic expandable 6 bit 20 MHz CMOS/SOS A/D converter," *IEEE JSSC,* vol. SC-14, no. 6, Dec. 1979, pp. 926–932.
11. Andrew G. F. Dingwall and Victor Zazzu, "An 8-MHz CMOS subranging 8-bit A/D converter, *IEEE JSSC,* vol. SC-20, no. 6, Dec. 1985, pp. 1138–1143.
12. Toshiro Tsukada, Yuuischi Nakatani, Eiki Imaizumi, Yoshitomi Toba, and Seiichi Ueda, "CMOS 8b 25MHz flash ADC," *Proceedings IEEE International Solid State Circuits Conference,* New York, N.Y., Feb. 1985, pp. 34–35.

Micron Ex. 1037, p. 742
Micron v. YMTC
IPR2025-00119

INDEX

.MODEL- SPICE MOS model
 call, 189
22V10, 392
3-D CMOS, 140
A/D converter, 694
abstraction levels, 21
AC specifications, 457
accelerated lifetime testing, 250
acceptor, 110
acceptor silicon, 112
accumulation, 181
accumulation mode, 44
accumulator architecture, 629
Actel, 395
active mask, 118
active-matrix LCDs, 140
activity file, 469
AD - SPICE MOS model parameter,
 190
ad-hoc testing, 485
adder as a compressor, 556
 bit-serial, 520
 block, 534
 carry lookahead, 526
 carry propagate, 523
 carry select, 532
 carry-save, 522
 conditional sum, 532
 Manchester, 528, 646, 667
 one bit, 515
 parallel, 517
 ripple, 517
 transmission gate, 524
 wide, 534
 truth table, 23
adders, 515-536
address architecture, 629
 comparator, 658
algebraic decomposition, 428
Algotronix, 403
ALPHA microprocessor, 333
ALU, 640, 645
ALU 181, 542
 instructions, 631
ambient temperature, 243
amorphous silicon, 140
amplifier, 71
AND function, 10

Micron Ex. 1037, p. 743
Micron v. YMTC
IPR2025-00119

gate 16
anisotropic etch, 126, 128
antifuse, 395
arithmetic logic units ALUs, 542
array multiplication, 545
AS - SPICE MOS model parameter, 190
ASTAP, 441
asychronous counters, 539
asynchronous RAM, 564
ATPG, 476
automatic gate layout, 280
test pattern generation ATPG, 476
avalanche breakdown, 47
phenomena, 362

back-annotation, 37, 448
backtrace, 479
barrel shifter, 560
base, 93
base-function set, 429
behavioral description, 31
domain, 21, 22
synthesis, 424
beta ratio, 68-69
BiCMOS, 96
BiCMOS logic, 297
processes, 136-138
bidirectional I/O, 364
BILBO, 494
binary adder truth table, 23
composition, 439
counters, 539
bipolar inverter, 94
transistor 93
transistor DC characteristics, 95
bird's beak, 121
bit-parallel adders, 517
bit-serial adders, 520
bloating masks, 165
block adder, 534
body effect, 51, 54, 223
ties, 123
BOLD, 429
bond diagram, 690
Boolean function unit, 305-306
operations, 541
unit, 647
Booth recoded multiplication, 547
bootstrapping, 217
Boron, 112
Boundary Scan techniques, 500-507
Bresenham algorithm, 23
built in logic block observation
BILBO, 494
bulk potential, 48
buried channel, 121
contact, 152
bypass capacitors on-chip, 689
bypassing, 637

C-SWITCH, 8
C^2MOS, 301-303
C^2MOS latch, 342
CAL1024, 403
call instruction, 634
Calma Stream format, 155
CAM, 589
Canonic Signed Digit CSD, 676
capacitance models, 192
values for 1µm process, 202
capacitor chip, 696
capacitors chip, 134
carrier concentration, 49
carry function, 23
lookahead adders, 526
propagate adder CPA, 523
save adders, 522
select adder, 532
cascade voltage switch logic, 311
cascode inverter, 80
Cathedral I II III, 424
C_{db}, 183
C_{gb}, 183
CGBO - SPICE MOS model
parameter, 189
C_{gd}, 183
CGDO - SPICE MOS model
parameter, 189
C_{gs}, 183
CGSO - SPICE MOS model parameter,
189
channel length modulation, 55
resistance, 60, 177
router, 431
stop, 116
stop implant, 119
charge pump, 334, 686
sharing, 240
storage, 333
chemical vapor deposition, 132
chip composition, 439
circuit extraction, 166

Micron Ex. 1037, p. 744
Micron v. YMTC
IPR2025-00119

circuit-level simulation, 441
CJ - SPICE MOS model parameter, 190
CJSW - SPICE MOS model parameter, 190
CLB, 400
clean V_{DD} and V_{SS}, 361
CLi6000, 406
clock buffer design, 364
buffers, 325
design, 333
distribution, 356
enabling, 350
generato,r 688
multiplication, 334
race, 323
skew, 201, 324, 325, 344, 345
synchronization, 334
tree, 356
clocking schemes, 334
CMOS delays, 206-216
cell array, 286
complimentary logic, 295
gate design, 262
inverter, 9
inverter - DC characteristics, 61-68
inverter amplifier, 71
inverter layout, 273
layout guidelines, 287
multi-drain logic, 299
process enhancements, 130
transistor construction - mask, 150
transmission gate - DC operation, 86-90
CMRR - Common Mode Rejection Ratio, 81
coarse grid symbolic-layout, 417
collector, 93
column decoders, 576
combinational logic, 10
common mode gain, 81
compaction, 420
comparator, 658
comparator analog, 696
comparators, 537
complex gate delays, 214
gate layout, 279
complimentary switch, 8
compound gates, 15
compressors 3:2 4:2, 556
concurrent fault simulation, 482
Concurrent Logic 406
condition-code logic, 660
conditional branching, 638
conditional-sum adder, 532
conductor capacitance as a function of spacing, 198
sheet resistance, 177
conductors sizing, 238
configurable array logic CAL, 403
logic block, CLB 400
conflict-free carry bypass, 531
constant current load inverter, 77
field scaling, 251
voltage scaling, 251
contact replication, 240
resistance, 179
rules, 151
content addressable memory CAM, 589
control logic, 590-620
logic example, 659
transfer, 633
controllability, 475
controllability measures, 479
COP, 481
core limited, 357
counter design, 487
counters, 539
CPA, 523
critical path, 644
paths, 263
cross-controlled latch, 494
crowbarred, 10
C_{sb}, 183
CSD, 676
current density, 238
mirror, 84, 336
starved inverter, 336
CVD, 132
CVSL, 311
cyclic redundancy checking, 494
Czochralski method, 110

D register, 325, 643
D-algorithm, 478
D-propagation, 478
DALG, 478
dark field, 147
data sheets, 456
datapath layout, 663, 684
datapaths, 513
DC specifications, 457
defects-manufacturing, 468

Micron Ex. 1037, p. 745
Micron v. YMTC
IPR2025-00119

delay fault testing, 482
  lines, 685
  model, 222
  time, 206, 211
depletion, 181
depletion capacitance, 182
  layer, 181
  lod inverter, 79
  mode, 44
  mode transistors, 42
deposition, 112
design corners, 246
  economics, 449
  flow, 441
  for testability, 485
  margining, 243
  rule verification, 448
  rule waiver, 142
device failure, 57
  under test DUT, 469
difference engine, 383
differential gain, 81
  inverter, 81-86
  split-level cascode logic, 314
diffusion capacitance, 186
  capacitance values, 188
  equation, 199
diode, 91
diode clamps, 362
  current, 91
directed-acyclic-graph DAG technology mapping, 429
dirty V_{DD} and V_{SS}, 361
disjoint hierarchy, 384
distributed *RC* effects, 198-202
domino logic, 308, 341
donor, 110
donor silicon, 112
donut transistor, 278
dopant, 110
double guard ring, 162
double-edge triggered register, 328
DRACULA, 164
drain engineering, 122
DRAM cells, 135
DRC, 164
dummy collectors, 162
DUT, 469
dynamic CMOS logic, 301
  D latches, 330
  logic, 310
  power dissipation, 233
  RAM cells, 566

EAROM, 136
EBES, 449
EBL, 113
ECL I/O, 367
edge-triggered register, 19-20, 319
EDIF, 430
EEPROM, 394
EEROM, 136
effective resistance, 219
E_g band gap energy of silicon, 49
electon beam lithography, 113
electrically alterable ROM, 136
Electron Beam Exposure System EBES, 449
ELLA, 22
Elmore delay, 219
emitter, 93
emphirical delays, 213
enhancement mode transistors, 42
environmental characteristics temperature, 243
  characteristics voltage 244
epitaxial, 140
epitaxial layer, 124, 160
epitaxy, 111
Espresso, 428
estimating design schedules, 453
Euler path, 280
evaluate, 301
externally induced latchup, 159

fall time, 206, 208
FAN, 478
fan-in, 264
fan-out, 264
fat-metal, 154
fault coverage, 475
  grading, 481
  machine, 475
  models, 472
  sampling, 484
  simulation, 481
field aiding, 163
  device, 116
field-induced junction, 45
field-oxide, 116
field-programmable gate-array FPGA, 400

Micron Ex. 1037, p. 746
Micron v. YMTC
IPR2025-00119

FIFO, 582
final test yield, 452
Finite Impulse Response, 674
state machine, FSM 317
finned capacitor, 135
FIR filter, 674
filter tap design, 680
first in first out memory FIFO, 582
fixed costs, 452
flash A/D converter, 694
flat-band voltage, 48
floorplan, 690, 698
floorplan examples, 665-671
floorplanning, 438
four-phase clocking, 351
Fowler-Nordheim tunneling, 57,136
FPGA, 400
frequency response, inverter, 698
fringing fields, 191
FSM, 317
full custom mask design, 417
fully restored, 15
function memory, 470
functional model, 692
verification, 669
functionality tests, 466
fusible links, 394

GaAs technology, 38
ganged CMOS, 300
gate capacitance, 181
extension, 150
isolation, 411
matrix symbolic-layout, 418
oxide, 43
oxide capacitance, 182
gate-array design, 407-413
layout, 285
gate-under-test, GUT, 477
gates compound, 15
GEMINI, 447
generate signal G, 515
generators, 434
geometric isolation, 410
Ghost canceller chip, 672
GND, 7
good machine, 475
graph compaction, 420
graphical editor, 437
gross die per wafer, 452
GROUND, 7
ground bounce, 239
GTL, 368
guard rings, 152, 361
Gunning I/O logic, 368

hardware description language, 437
stack, 655
HDL, 425, 437
Heil, O. , 3
hierarchy, 383
high-impedance, 8, 10
hold time, 318, 323
holding voltage, 156
hot electrons, 57
HSPICE, 441

I/O latchup prevention, 162
pads, 357
structures, 357
IDD testing, 474
IDDQ testing, 74, 498
IIR filter, 674
impact ionization, 57
impurity, 110
incrementer, 539
inductance, 205
inertial delay, 443
input, 7
input pad, 361
protection, 362
waveform slope, 216
instruction set, 629
inter-layer contacts, 151
interdigitated pads, 357
internal latchup prevention, 161
intrinsic delay, 443
silicon, 109
inversion, 181
inversion mode, 44
notation, 19-20
inverter, 9
inverter BiCMOS, 96-98
bipolar, 94
current load, 72
layout, 273
resistive load, 72
threshold voltage, 228
ion implantation, 112
island, 118
isolation of MOS transistors, 50

Micron Ex. 1037, p. 747
Micron v. YMTC
IPR2025-00119

isomorphism network, 446
isotropic etch, 128
iterative logic array testing, 498

jamb latch, 326
JK register, 320
jump instruction, 634
junction capacitance, 187
  diode, 91
  pn, 110
  temperature, 243
k Boltzmann's constant, 49
kink effect, 129

LAGER, 424
lambda design rules, 142
  SPICE parameter, 51
laser pantography, 395
last in first out memory LIFO, 582
latch, 19-20, 318
latchup, 156-163
latchup prevention, 160
  triggering, 158
latency, 524
lateral scaling, 251
layer assignments CIF, 155
  assignments GDS2, 155
  representations, 143
layout design rules, 142-156
  extraction, 448
  multiplexer, 294
  synthesis, 434
  transmission-gate, 291
layout-versus-schematic LVS, 447
LDD, 122
leaf cell, 35
leakage current, 231
LEVEL, 481
Level 3 SPICE model, 99-105
  sensitive scan design LSSD, 489
level-sensitive latch, 19-20, 319
LFSR, 496
LIFO, 582
light field, 147
lightly doped drain LDD, 122
Lilienfeld, J. , 3
linear feedback shift register LFSR, 496
  operating region, 45
Lisp, 692
literal, 632

local interconnect, 133
locality, 389
LOCOS, 121
logic levels, 8
  optimization, 427
  simulation, 443
loop filter, 334, 686
low power logic, 368-370
  power standard cells, 414
LSSD, 489
LVS, 447

machine as applied in testing, 475
macromodeling, 221
Manchester adder, 528, 646, 667
manufacturing defects, 468
  tests, 468
maze router, 431
memory, 563-590
memory test, 497
merged contact, 152
mesa, 118
metal interconnect, 130
  rules, 154
  tab, 131
metal2 process steps, 132
  rules, 154
metal3 rules, 154
metallization, 122
metallurgical junction, 45
metal migration, 238-239, 361
metastability, 337
Miller effect, 218
min-cut algorithm, 431
MIS, 429
mixed-mode simulation, 444
MJ - SPICE MOS model parameter, 190
MJSW - SPICE MOS model parameter, 190
mobility, 52
mobility variation, 56
modularity, 387
MOS capacitor, 180
  DC device equations, 51
  device capacitances ,183
  gate capacitance, 184
  switches, 7
  transistor DC characteristics, 53
  transistor intoduction, 4
  transistor invention, 3

Micron Ex. 1037, p. 748
Micron v. YMTC
IPR2025-00119

transistor symbols, 41
MOTIS, 442
MTBU, 339
MULGA, 421
multiple conductor capacitances, 192
multiplexer, 17-18, 140, 304
multiplexer layout, 294
multipliers, 542-560
multiport memories, 580

N-switch, 7
n-well CMOS process, 117-123
CMOS process flow, 168-172
construction, 118
rules, 150
NAND gate, 11
gate delay, 265
NAND-NOR delays, 267
layout, 278
native substrate, 117
net-list, 35
netlist comparison, 447
network isomorphism, 446
nichrome, 134
NM_H, 70
NM_L, 70
nMOS transistor, 43
noise margin, 69-71
non-recurring costs NREs, 450
nonsaturated region, 45
NOR, 586
NOR gate, 12
not gate, 9
notation, 19-20
notation inversion, 19-20
NP domino logic, 310
dynamic logic, 341
NREs, 450
NRZ, 470
NS, 421, 434

observability, 474
observability measures, 479
ohmic contact, 121
one detectors, 537
one-bit adders, 515
ONO, 396
open-circuit faults, 473
operating conditions temperature, 243
OR function, 10
output, 7
output condunctance, 60
pads, 360
overglass, 155
oxidation of SiO_2, 111
oxide breakdown, 361
capacitance, 182
oxide-nitride-oxide ONO, 396

p-well CMOS process, 123-124
packaging, 247
packaging yield, 452
pad limited, 357
pullups and pulldowns 365
PALs, 392
parallel fault simulation, 481
hierarchy, 383
plate capacitance model, 191
scan testing, 494
switches, 10
parasistic transistors, 116
parasitic capacitance, 183
parity generators, 537
partial products, 546
scan testing, 493
pass transistor logic, 304
pass-around, 637
passivation, 155
pattern gates, 429
generation, 448
PB - SPICE MOS model parameter, 190
PC, *see* program counter, 654
PCMs, 225
PD - SPICE MOS model parameter, 190
Pearl, 445
Penfield-Rubenstein delay model, 219
slope delay mode, 220
peripheral capacitance, 187
personalization, 411
PG logic, 530
PGA, 247
phase detector, 686
Locked Loop PLL, 685
phased-locked loops, 334
Phosphorous, 112
phosphorous glass, 126
photoresist, 113
physical description, 28, 35-36
domain, 21

Micron Ex. 1037, p. 749
Micron v. YMTC
IPR2025-00119

hierarchy, 384
origin of latchup, 156
PIDL, 168
pinched off, 45
pipeline diagram, 637
pipelined system, 317
pipelining, 524, 634
PLA generator example, 435
placement, 431
planarization, 121
PLDs, 392
PLICE, 395
PLL, 685
PLLs, 334
pMOS transistor, 47
pn diode, 91
juntion, 45
pnp transistor layout, 153
PODEM, 478
PODEM-X, 478
poly2, 134
polycide, 132
polysilicon, 113
POWER, 7
power and ground bounce, 239
bounce, 239
dissipation, 231-237
dissipation dynamic, 233
dissipation short-circuit, 235
dissipation static, 231
distribution, 689
economy, 237
saving pseudo-nMOS, 587
PQFP, 247
PR, 113
precharge, 301
predecode gates, 575
preferential etch, 128
primary inputs, 477
outputs, 477
probe card, 470
process control monitors PCMs, 225
gain factor, 54
Input Desciption Language, 168
migration, 423
variation, 245
program counter, 634, 654
programmable array logic PALs, 392
interconnect, 395
logic devices PLDs, 392
logic structures, 392
propagate signal P, 515
PRSG, 496
PS - SPICE MOS model parameter, 190
pseudo random sequence generator PRSG, 496
pseudo-nMOS, 537, 586, 646, 658, 698
pseudo-nMOS inverter, 73-77, 228
logic, 298
PSG, 137
PSWITCH, 8
punch-through, 47
punch-through devices, 362
punchthrough, 57
PWR, 7

q electronic charge, 49
QuickLogic, 396

race clock, 323
radiation tolerance, 129
radio frequency interference RFI pads, 366
radix-2 multiplication, 547
RAM, 564
RAM read operation, 567
sense amplifier, 84
write operation, 572
rapid prototyping, 653
rats-nest, 438
RC delay clock, 334
read-only-memory, 585
recurring costs, 450, 452
refractory metal, 132
register, 19-20, 318
register file, 651
files, 580
registered pads, 365
regularity, 387
reliability, 250
resettable register, 330
resistance, 176
resistance extimation, 176
of nonrectangular regions, 178
resistivity, 176
resistor string, 696
resistors chip, 134
result forwarding, 637
return instruction, 634
reverse breakdown voltage, 93

Micron Ex. 1037, p. 750
Micron v. YMTC
IPR2025-00119

rip-up-and-reroute, 439
ripple adders, 517
RISC microcontroller, 628
rise time, 206, 210
ROM, 585
ROM layout, 588
round transistor, 278
routing, 431
routing capacitance, 191-198
row decoders, 574
RS-latch, 319
RSIM, 444
RTL synthesis, 425
RTZ, 470
Rubylith®, 448
rule based logic optimizer, 429

SA0, 472
SA1, 472
salicide, 132
sample-set differential logic SSDL, 313
sapphire, 126
saturated load inverters, 78-79
  region, 45
SBZ, 470
scaling, 250
schematic design, 437
  icon, 437
schmooing, 471
SCOAP, 479
scribe line, 155
sea-of-gates design, 407-413
  layout, 286
seed layer, 140
select mask, 122
selective diffusion, 113
self test, 494
self-aligned process, 116
sense amplifier, 84
  amplifiers, 579
sensitized path, 477
sequential fault grading, 475
  faults, 473
serial in parallel out memory SIPO, 582
  multiplication, 557
  scan testing, 490
serial-access memory, 583
serial/parallel multiplication, 559
series switches, 10
settable register, 330
setup time, 318
  time, 323
SFPL, 314
sheet resistance, 176
shift register, 685
  register latch SRL , 489
shifter, 649
shifters, 560
short-circuit dissipation, 235
  faults, 473
sidewall capacitance, 187
signature analysis, 494
silicide, 132
silicon gate process, 113
  nitride, 50
Silicon-on-insulator, 125-130
simulation, 441-445
simulation circuit-level, 441
  logic, 443
  mixed-mode, 444
  switch-level, 444
  timing, 442
simulator delay model, 222
SiN, 112, 120
single dynamic clock latches, 331
  wire capacitance, 191
sinker layer, 140
SiO2, 112
SIPO, 582
site, 285
sizing conductors, 238
slope delay model, 220
  effect on delay, 216
small signal characteristics, 59
SOG, 407-413
SOI, 125-130
SOI advantages, 129
  rules 156
source follower pull-up logic SFPL, 314
source-drain extension, 150
SPICE, 441
SPICE characterization example, 652
  circuit description language, 27-28
  modeling of capacitance, 188
  MOS Model call, 189
  MOS parameters - typical, 59
split contact, 152
SRAM cell, 133, 139
SRL, 489
SSDL, 313

Micron Ex. 1037, p. 751
Micron v. YMTC
IPR2025-00119

stack address generator, 660
architecture, 630
hardware, 655
STAFAN, 483
stage ratio, 229
stage-ratio, 265
standard cell design, 413-416
cells 283
static load, 72
power dissipation 231
RAM cells 565
statistical fault modeling, 483
step coverage, 121, 131
sticks symbolic-layout, 420
strong 0, 8
1, 8
structural description, 32-35
domain, 21
representation, 24
synthesis, 431
structured design methods, 383
stuck-at faults, 472
stuck-at-0 fault, 472
stuck-at-1 fault, 472
subject graph, 429
submicron processes, 147
subroutine call and return, 639
substrate contacts, 123
resistance, 157
substrate-bias effect, 54
subthreshold region, 55
subtractor, 518
SUM logic, 530
sum-of-products, 428
summands, 546
surface state charge, 49
SWAMI, 121
switch-level RC models, 218
simulation, 444
switchbox router, 431
switches parallel, 10
series, 10
symbolic layout, 417-423
symbolic-layout coarse grid, 417
gate matrix, 418
sticks, 420
virtual-grid, 421
symmetric NOR gate, 300
synchronizer failure, 337
synchronizers, 337
synchronous counters, 539
Synopsys VHDL compiler, 425
T latch or register, 320
Tantalum, 132
tantalum silicide, 132
tapped delay line, 585
technology mapping, 428, 429
temperature effect on inverter transfer characteristic, 69
variation - theshold voltage, 48
variation of resistance, 178
ternary, 405
test program format, 469
tester, 470
testing, 669
TG adder, 553
XOR gate, 525
thermal annealing, 431
impedance, 243
resistance, 253
thermometer code, 698
thin-film transistors, 139
thinox, 118
thinoxide, 118
three level metal standard cells, 415
threshold adjust, 49-50
voltage, 47–50, 47
throughput, 524
TimberWolf, 431, 432
timing analysis example, 649
analyzers, 445
budget, 651
generator, 470
simulation, 442
verifiers, 445
TiN, 133
topside connection, 123
transconductance, 60
transistor rules, 150
sizing, 226, 271
transmission gate - DC operation, 86-90
transmission-gate adder, 524
layout, 291
transparent routing, 289
trench capacitor, 135
triangle function, 31
trigger point, 156, 158
tristate I/O, 364
inverter, 91
tristate-buffer latch, 326
TTL interface inverter, 80
load, 361
tunnel oxide, 136

Micron Ex. 1037, p. 752
Micron v. YMTC
IPR2025-00119

twin-well CMOS process, 124-125
two phase clock generator, 349
  phase clocking, 344
  phase dynamic registers, 347
two-level minimization, 428
unsaturated load inverters, 77-78
  region, 45

UV light, 113
UV-PROM, 394

V_{BE}, 93
VCDL, 336
V_{CE}, 93
VCO, 334, 687
V_{DD}, 7
V_{DD} and V_{SS} pads, 360
  contact, 123, 152
VHDL, 425
via construction, 131
  resistance, 179
  rules, 154
via2 rules, 154
ViaLink, 396
V_{IH}, 70
V_{IL}, 70
virtual-grid symbolic-layout, 421,434
VJ - SPICE MOS model parameter, 190
V_{OH}, 70
V_{OL}, 70
voltage controlled delay line VCDL, 336
  controlled oscillator, VCO 687
  regulator, 296
voltage-controlled oscillator, 334
V_{SS}, 7
V_{SS} contact, 123, 152

wafer, 110
wafer processing, 109-110
waiver-design rule, 142
Wallace tree multiplication, 554
Wanlass, Frank , 3
wave pipelining, 324
weak 0, 8
  1, 8
  division, 428
weak-feedback inverter, 326
Weimer, P.K. , 3
well contacts, 123
  resistance, 157
  ties, 123
white space, 288
wide adders, 534
wire length design guide, 204
work function, 49
worst-power corner, 246
worst-speed corner, 246

XC3000, 400
XC4000, 401
XILINX, 400
XNOR, gate 282, 304, 305, 658
XOR, 312, 525, 540

Y chart, 382
yield, 248
Yorktown Silicon Compiler, 425

zero and negate, 646
  detect, 647
zero/one detectors, 537
Zipper CMOS, 310

Micron Ex. 1037, p. 753
Micron v. YMTC
IPR2025-00119

TK7874/.W522/1993
Weste, Neil H. E.

Principles of CMOS VLSI design : a systems perspective / Neil H.E. bimb
c. 1 SVO

DATE	ISSUED TO
~~7/28/07~~	~~Conard~~
~~7/28/10~~	~~Shev~~
~~11/18/2011~~	~~E. Schuerlein~~

DEMCO RE-ORDER P802514

TK7874/.W522/1993
Weste, Neil H. E.

Principles of CMOS VLSI design : a systems perspective / Neil H.E. bimb
c. 1 SVO