Exercise 4.50 Each of the following VHDL modules contains an error. For brevity, only the architecture is shown; assume that the `library` use clause and `entity` declaration are correct. Explain the error and show how to fix it.

(a)
```
architecture synth of latch is
begin
  process (clk) begin
     if clk = '1' then q <= d;
     end if;
  end process;
end;
```

(b)
```
architecture proc of gates is
begin
  process (a) begin
     y1 <= a and b;
     y2 <= a or b;
     y3 <= a xor b;
     y4 <= a nand b;
     y5 <= a nor b;
  end process;
end;
```

(c)
```
architecture synth of flop is
begin
  process (clk)
    if clk'event and clk = '1' then
      q <= d;
end;
```

(d)
```
architecture synth of priority is
begin
  process (a) begin
    if    a(3) = '1' then y <= "1000";
    elsif a(2) = '1' then y <= "0100";
    elsif a(1) = '1' then y <= "0010";
    elsif a(0) = '1' then y <= "0001";
    end if;
  end process;
end;
```

(e)
```
architecture synth of divideby3FSM is
  type statetype is (S0, S1, S2);
  signal state, nextstate: statetype;
begin
  process (clk, reset) begin
     if reset = '1' then state <= S0;
     elsif clk'event and clk = '1' then
       state <= nextstate;
     end if;
  end process;
```

```
      process (state) begin
       case state is
          when S0 => nextstate <= S1;
          when S1 => nextstate <= S2;
          when S2 => nextstate <= S0;
       end case;
      end process;

    q <= '1' when state = S0 else '0';
  end;
```

(f)
```
  architecture struct of mux2 is
    component tristate
      port (a:  in  STD_LOGIC_VECTOR (3 downto 0);
            en: in  STD_LOGIC;
            y:  out STD_LOGIC_VECTOR (3 downto 0));
    end component;
  begin
    t0: tristate port map (d0, s, y);
    t1: tristate port map (d1, s, y);
  end;
```

(g)
```
  architecture asynchronous of flopr is
  begin
    process (clk, reset) begin
      if reset = '1' then
        q <= '0';
      elsif clk'event and clk = '1' then
        q <= d;
      end if;
    end process;

    process (set) begin
      if set = '1' then
        q <= '1';
      end if;
    end process;
  end;
```

(h)
```
  architecture synth of mux3 is
  begin
    y <= d2 when s(1) else
         d1 when s(0) else d0;
  end;
```

Interview Questions

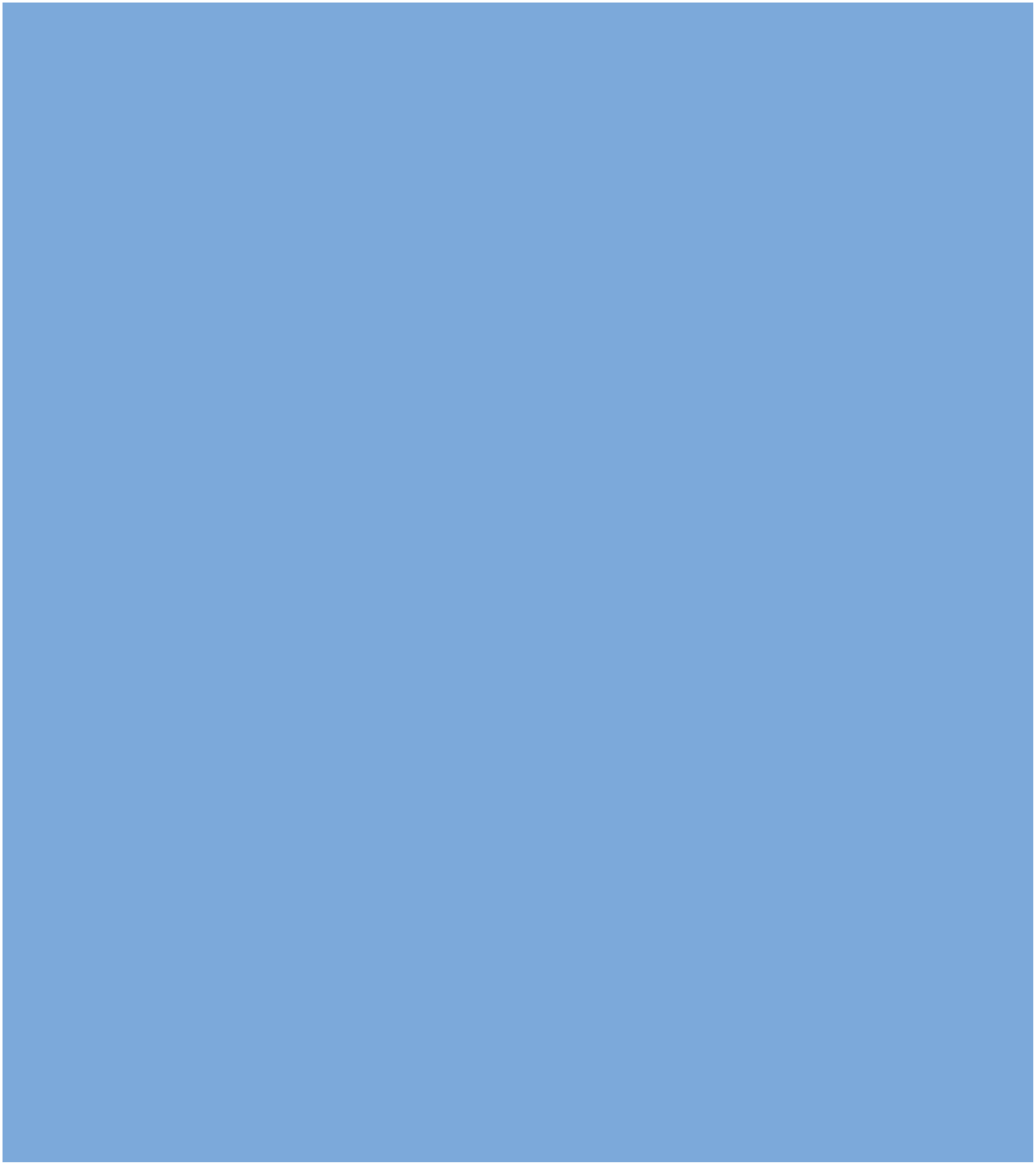
The following exercises present questions that have been asked at interviews for digital design jobs.

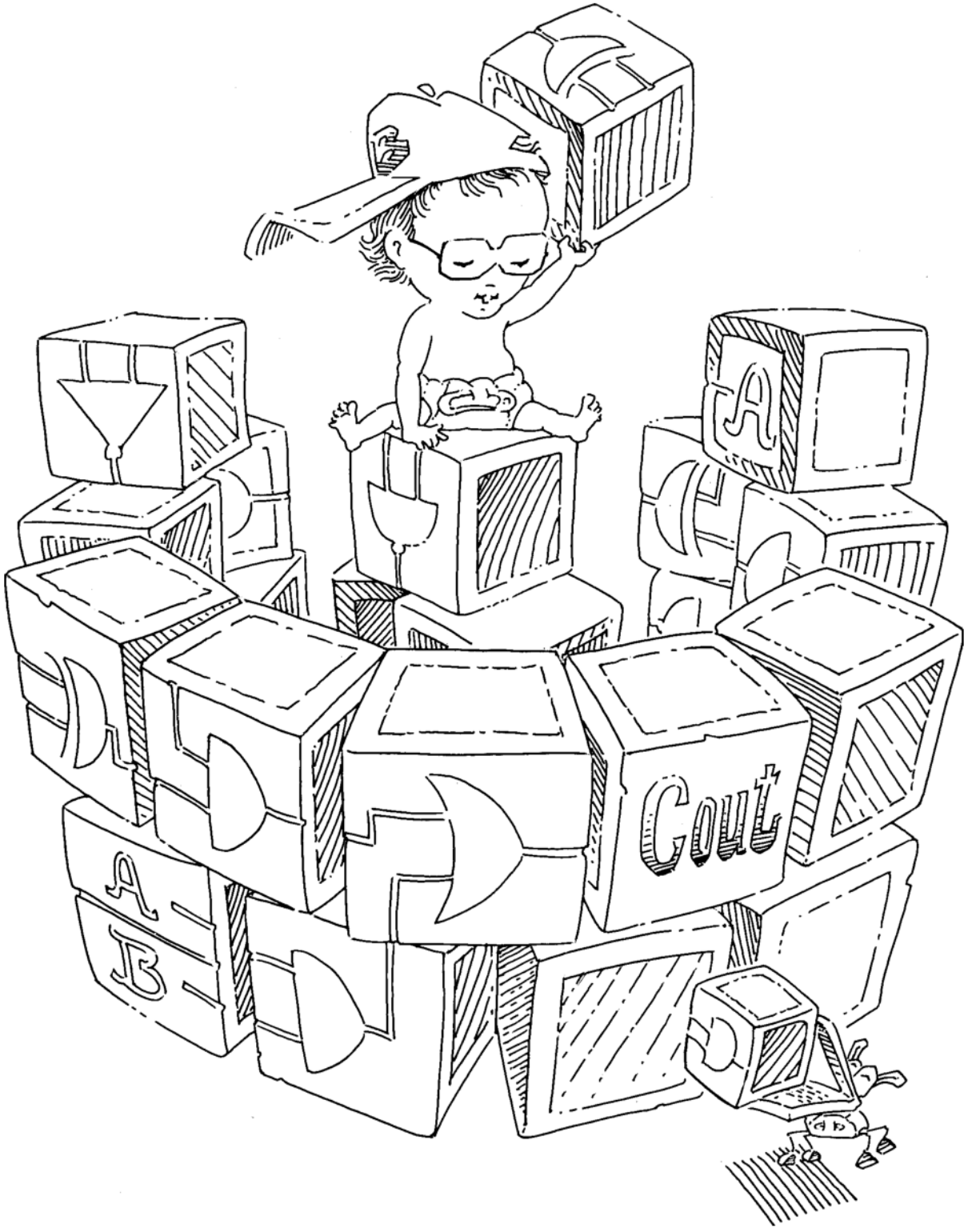
Question 4.1 Write a line of HDL code that gates a 32-bit bus called `data` with another signal called `sel` to produce a 32-bit `result`. If `sel` is TRUE, `result = data`. Otherwise, `result` should be all 0's.

Question 4.2 Explain the difference between blocking and nonblocking assignments in Verilog. Give examples.

Question 4.3 What does the following Verilog statement do?

```
result = | (data[15:0] & 16'hC820);
```




A
Coué
A
B

Digital Building Blocks

5

5.1 INTRODUCTION

Up to this point, we have examined the design of combinational and sequential circuits using Boolean equations, schematics, and HDLs. This chapter introduces more elaborate combinational and sequential building blocks used in digital systems. These blocks include arithmetic circuits, counters, shift registers, memory arrays, and logic arrays. These building blocks are not only useful in their own right, but they also demonstrate the principles of hierarchy, modularity, and regularity. The building blocks are hierarchically assembled from simpler components such as logic gates, multiplexers, and decoders. Each building block has a well-defined interface and can be treated as a black box when the underlying implementation is unimportant. The regular structure of each building block is easily extended to different sizes. In Chapter 7, we use many of these building blocks to build a microprocessor.

5.2 ARITHMETIC CIRCUITS

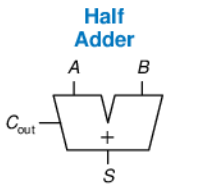
Arithmetic circuits are the central building blocks of computers. Computers and digital logic perform many arithmetic functions: addition, subtraction, comparisons, shifts, multiplication, and division. This section describes hardware implementations for all of these operations.

5.2.1 Addition

Addition is one of the most common operations in digital systems. We first consider how to add two 1-bit binary numbers. We then extend to N-bit binary numbers. Adders also illustrate trade-offs between speed and complexity.

Half Adder

We begin by building a 1-bit *half adder*. As shown in Figure 5.1, the half adder has two inputs, A and B, and two outputs, S and C_{out}. S is the



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

$$S = A \oplus B$$
$$C_{out} = AB$$

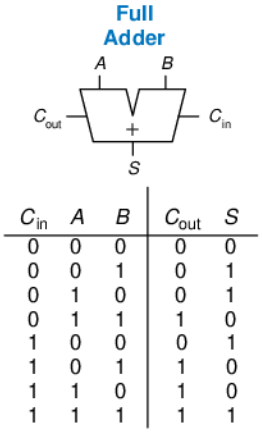
FIGURE 5.1 1-bit half adder

sum of A and B. If A and B are both 1, S is 2, which cannot be represented with a single binary digit. Instead, it is indicated with a carry out, C_{out}, in the next column. The half adder can be built from an XOR gate and an AND gate.

In a multi-bit adder, C_{out} is added or *carried in* to the next most significant bit. For example, in Figure 5.2, the carry bit shown in blue is the output, C_{out}, of the first column of 1-bit addition and the input, C_{in}, to the second column of addition. However, the half adder lacks a C_{in} input to accept C_{out} of the previous column. The *full adder*, described in the next section, solves this problem.

```
  1
 0001
+0101
 0110
```

Figure 5.2 Carry bit



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

$$S = A \oplus B \oplus C_{in}$$
$$C_{out} = AB + AC_{in} + BC_{in}$$

FIGURE 5.3 1-bit full adder

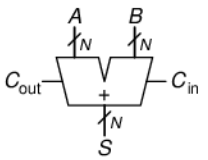


Figure 5.4 Carry propagate adder

Schematics typically show signals flowing from left to right. Arithmetic circuits break this rule because the carries flow from right to left (from the least significant column to the most significant column).

Full Adder

A *full adder*, introduced in Section 2.1, accepts the carry in, C_{in}, as shown in Figure 5.3. The figure also shows the output equations for S and C_{out}.

Carry Propagate Adder

An N-bit adder sums two N-bit inputs, A and B, and a carry in, C_{in}, to produce an N-bit result, S, and a carry out, C_{out}. It is commonly called a *carry propagate adder* (*CPA*) because the carry out of one bit propagates into the next bit. The symbol for a CPA is shown in Figure 5.4; it is drawn just like a full adder except that A, B, and S are busses rather than single bits. Three common CPA implementations are called ripple-carry adders, carry-lookahead adders, and prefix adders.

Ripple-Carry Adder

The simplest way to build an N-bit carry propagate adder is to chain together N full adders. The C_{out} of one stage acts as the C_{in} of the next stage, as shown in Figure 5.5 for 32-bit addition. This is called a *ripple-carry adder*. It is a good application of modularity and regularity: the full adder module is reused many times to form a larger system. The ripple-carry adder has the disadvantage of being slow when N is large. S_{31} depends on C_{30}, which depends on C_{29}, which depends on C_{28}, and so forth all the way back to C_{in}, as shown in blue in Figure 5.5. We say that the carry *ripples* through the carry chain. The delay of the adder, t_{ripple}, grows directly with the number of bits, as given in Equation 5.1, where t_{FA} is the delay of a full adder.

$$t_{ripple} = Nt_{FA} \tag{5.1}$$

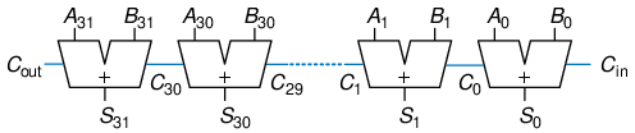


Figure 5.5 32-bit ripple-carry adder

Carry-Lookahead Adder

The fundamental reason that large ripple-carry adders are slow is that the carry signals must propagate through every bit in the adder. A *carry-lookahead* adder is another type of carry propagate adder that solves this problem by dividing the adder into *blocks* and providing circuitry to quickly determine the carry out of a block as soon as the carry in is known. Thus it is said to *look ahead* across the blocks rather than waiting to ripple through all the full adders inside a block. For example, a 32-bit adder may be divided into eight 4-bit blocks.

Carry-lookahead adders use *generate* (G) and *propagate* (P) signals that describe how a column or block determines the carry out. The ith column of an adder is said to *generate* a carry if it produces a carry out independent of the carry in. The ith column of an adder is guaranteed to generate a carry, C_i, if A_i and B_i are both 1. Hence G_i, the generate signal for column i, is calculated as $G_i = A_i B_i$. The column is said to *propagate* a carry if it produces a carry out whenever there is a carry in. The ith column will propagate a carry in, C_{i-1}, if either A_i or B_i is 1. Thus, $P_i = A_i + B_i$. Using these definitions, we can rewrite the carry logic for a particular column of the adder. The ith column of an adder will generate a carryout, C_i, if it either generates a carry, G_i, or propagates a carry in, $P_i\, C_{i-1}$. In equation form,

$$C_i = A_i\, B_i + (A_i + B_i)\, C_{i-1} = G_i + P_i\, C_{i-1} \tag{5.2}$$

The generate and propagate definitions extend to multiple-bit blocks. A block is said to generate a carry if it produces a carry out independent of the carry in to the block. The block is said to propagate a carry if it produces a carry out whenever there is a carry in to the block. We define $G_{i:j}$ and $P_{i:j}$ as generate and propagate signals for blocks spanning columns i through j.

A block generates a carry if the most significant column generates a carry, or if the most significant column propagates a carry and the previous column generated a carry, and so forth. For example, the generate logic for a block spanning columns 3 through 0 is

$$G_{3:0} = G_3 + P_3\,(G_2 + P_2\,(G_1 + P_1\, G_0)) \tag{5.3}$$

A block propagates a carry if all the columns in the block propagate the carry. For example, the propagate logic for a block spanning columns 3 through 0 is

$$P_{3:0} = P_3\, P_2\, P_1\, P_0 \tag{5.4}$$

Using the block generate and propagate signals, we can quickly compute the carry out of the block, C_i, using the carry in to the block, C_j.

$$C_i = G_{i:j} + P_{i:j}\, C_j \tag{5.5}$$

Throughout the ages, people have used many devices to perform arithmetic. Toddlers count on their fingers (and some adults stealthily do too). The Chinese and Babylonians invented the abacus as early as 2400 BC. Slide rules, invented in 1630, were in use until the 1970's, when scientific hand calculators became prevalent. Computers and digital calculators are ubiquitous today. What will be next?

Figure 5.6(a) shows a 32-bit carry-lookahead adder composed of eight 4-bit blocks. Each block contains a 4-bit ripple-carry adder and some lookahead logic to compute the carry out of the block given the carry in, as shown in Figure 5.6(b). The AND and OR gates needed to compute the single-bit generate and propagate signals, G_i and P_i, from A_i and B_i are left out for brevity. Again, the carry-lookahead adder demonstrates modularity and regularity.

All of the CLA blocks compute the single-bit and block generate and propagate signals simultaneously. The critical path starts with computing G_0 and $G_{3:0}$ in the first CLA block. C_{in} then advances directly to C_{out} through the AND/OR gate in each block until the last. For a large adder, this is much faster than waiting for the carries to ripple through each consecutive bit of the adder. Finally, the critical path through the last block contains a short ripple-carry adder. Thus, an N-bit adder divided into k-bit blocks has a delay

$$t_{CLA} = t_{pg} + t_{pg\_block} + \left(\frac{N}{k} - 1\right) t_{AND\_OR} + kt_{FA} \qquad (5.6)$$

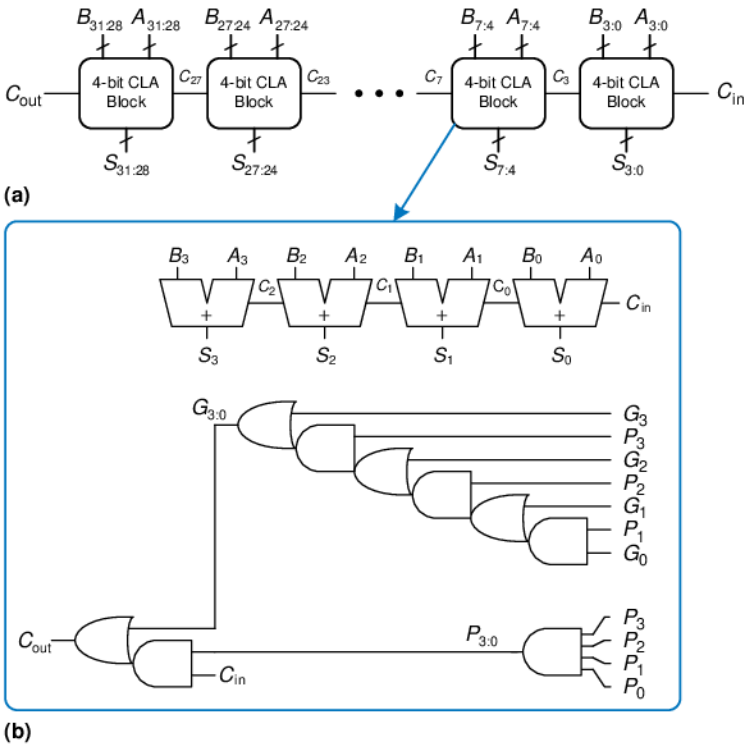


Figure 5.6 (a) 32-bit *carry-lookahead adder* (*CLA*), (b) 4-bit CLA block

where t_{pg} is the delay of the individual generate/propagate gates (a single AND or OR gate) to generate P and G, t_{pg_block} is the delay to find the generate/propagate signals $P_{i:j}$ and $G_{i:j}$ for a k-bit block, and t_{AND_OR} is the delay from C_{in} to C_{out} through the AND/OR logic of the k-bit CLA block. For $N > 16$, the carry-lookahead adder is generally much faster than the ripple-carry adder. However, the adder delay still increases linearly with N.

Example 5.1 RIPPLE-CARRY ADDER AND CARRY-LOOKAHEAD ADDER DELAY

Compare the delays of a 32-bit ripple-carry adder and a 32-bit carry-lookahead adder with 4-bit blocks. Assume that each two-input gate delay is 100 ps and that a full adder delay is 300 ps.

Solution: According to Equation 5.1, the propagation delay of the 32-bit ripple-carry adder is 32×300 ps $= 9.6$ ns.

The CLA has $t_{pg} = 100$ ps, $t_{pg\_block} = 6 \times 100$ ps $= 600$ ps, and $t_{AND\_OR} = 2 \times 100$ ps $= 200$ ps. According to Equation 5.6, the propagation delay of the 32-bit carry-lookahead adder with 4-bit blocks is thus 100 ps + 600 ps + $(32/4 - 1) \times 200$ ps + $(4 \times 300$ ps$) = 3.3$ ns, almost three times faster than the ripple-carry adder.

Prefix Adder*

Prefix adders extend the generate and propagate logic of the carry-lookahead adder to perform addition even faster. They first compute G and P for pairs of columns, then for blocks of 4, then for blocks of 8, then 16, and so forth until the generate signal for every column is known. The sums are computed from these generate signals.

In other words, the strategy of a prefix adder is to compute the carry in, C_{i-1}, for each column, i, as quickly as possible, then to compute the sum, using

$$S_i = (A_i \oplus B_i) \oplus C_{i-1} \tag{5.7}$$

Define column $i = -1$ to hold C_{in}, so $G_{-1} = C_{in}$ and $P_{-1} = 0$. Then $C_{i-1} = G_{i-1:-1}$ because there will be a carry out of column $i-1$ if the block spanning columns $i-1$ through -1 generates a carry. The generated carry is either generated in column $i-1$ or generated in a previous column and propagated. Thus, we rewrite Equation 5.7 as

$$S_i = (A_i \oplus B_i) \oplus G_{i-1:-1} \tag{5.8}$$

Hence, the main challenge is to rapidly compute all the block generate signals $G_{-1:-1}$, $G_{0:-1}$, $G_{1:-1}$, $G_{2:-1}$, . . . , $G_{N-2:-1}$. These signals, along with $P_{-1:-1}$, $P_{0:-1}$, $P_{1:-1}$, $P_{2:-1}$, . . . , $P_{N-2:-1}$, are called *prefixes*.

Early computers used ripple carry adders, because components were expensive and ripple carry adders used the least hardware. Virtually all modern PCs use prefix adders on critical paths, because transistors are now cheap and speed is of great importance.

Figure 5.7 shows an $N = 16$-bit prefix adder. The adder begins with a *precomputation* to form P_i and G_i for each column from A_i and B_i using AND and OR gates. It then uses $\log_2 N = 4$ levels of black cells to form the prefixes of $G_{i:j}$ and $P_{i:j}$. A black cell takes inputs from the upper part of a block spanning bits $i{:}k$ and from the lower part spanning bits $k-1{:}j$. It combines these parts to form generate and propagate signals for the entire block spanning bits $i{:}j$, using the equations.

$$G_{i:j} = G_{i:k} + P_{i:k} G_{k-1:j} \tag{5.9}$$

$$P_{i:j} = P_{i:k} P_{k-1:j} \tag{5.10}$$

In other words, a block spanning bits $i{:}j$ will generate a carry if the upper part generates a carry or if the upper part propagates a carry generated in the lower part. The block will propagate a carry if both the

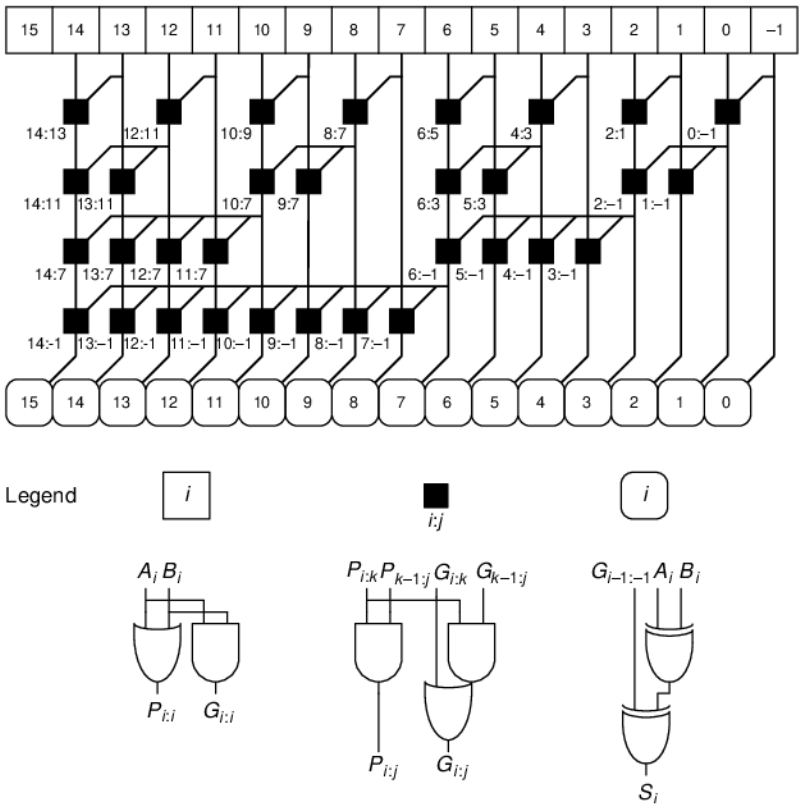


Figure 5.7 16-bit prefix adder

upper and lower parts propagate the carry. Finally, the prefix adder computes the sums using Equation 5.8.

In summary, the prefix adder achieves a delay that grows logarithmically rather than linearly with the number of columns in the adder. This speedup is significant, especially for adders with 32 or more bits, but it comes at the expense of more hardware than a simple carry-lookahead adder. The network of black cells is called a *prefix tree*.

The general principle of using prefix trees to perform computations in time that grows logarithmically with the number of inputs is a powerful technique. With some cleverness, it can be applied to many other types of circuits (see, for example, Exercise 5.7).

The critical path for an N-bit prefix adder involves the precomputation of P_i and G_i followed by $\log_2 N$ stages of black prefix cells to obtain all the prefixes. $G_{i-1:-1}$ then proceeds through the final XOR gate at the bottom to compute S_i. Mathematically, the delay of an N-bit prefix adder is

$$t_{PA} = t_{pg} + \log_2 N(t_{pg\_\text{prefix}}) + t_{\text{XOR}} \tag{5.11}$$

where t_{pg_prefix} is the delay of a black prefix cell.

Example 5.2 PREFIX ADDER DELAY

Compute the delay of a 32-bit prefix adder. Assume that each two-input gate delay is 100 ps.

Solution: The propagation delay of each black prefix cell, t_{pg_prefix}, is 200 ps (i.e., two gate delays). Thus, using Equation 5.11, the propagation delay of the 32-bit prefix adder is $100\text{ ps} + \log_2(32) \times 200\text{ ps} + 100\text{ ps} = 1.2\text{ ns}$, which is about three times faster than the carry-lookahead adder and eight times faster than the ripple-carry adder from Example 5.1. In practice, the benefits are not quite this great, but prefix adders are still substantially faster than the alternatives.

Putting It All Together

This section introduced the half adder, full adder, and three types of carry propagate adders: ripple-carry, carry-lookahead, and prefix adders. Faster adders require more hardware and therefore are more expensive and power-hungry. These trade-offs must be considered when choosing an appropriate adder for a design.

Hardware description languages provide the + operation to specify a CPA. Modern synthesis tools select among many possible implementations, choosing the cheapest (smallest) design that meets the speed requirements. This greatly simplifies the designer's job. HDL Example 5.1 describes a CPA with carries in and out.

HDL Example 5.1 ADDER

Verilog

```
module adder #(parameter N = 8)
              (input  [N-1:0] a, b,
               input          cin,
               output [N-1:0] s,
               output         cout);

  assign {cout, s} = a + b + cin;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity adder is
  generic (N: integer := 8);
  port (a, b:    in  STD_LOGIC_VECTOR(N-1 downto 0);
        cin:     in  STD_LOGIC;
          s:     out STD_LOGIC_VECTOR(N-1 downto 0);
       cout:     out STD_LOGIC);
end;

architecture synth of adder is
  signal result: STD_LOGIC_VECTOR(N downto 0);
begin
  result   <= ("0" & a) + ("0" & b) + cin;
  s        <= result (N-1 downto 0);
  cout     <= result (N);
end;
```

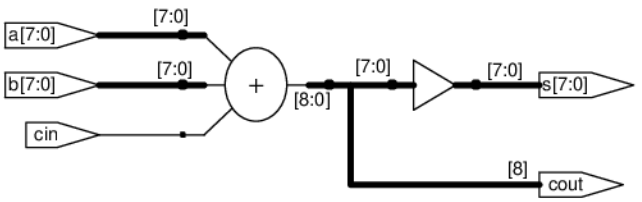


Figure 5.8 Synthesized adder

5.2.2 Subtraction

Recall from Section 1.4.6 that adders can add positive and negative numbers using two's complement number representation. Subtraction is almost as easy: flip the sign of the second number, then add. Flipping the sign of a two's complement number is done by inverting the bits and adding 1.

To compute $Y = A - B$, first create the two's complement of B: Invert the bits of B to obtain $\overline{B}$ and add 1 to get $-B = \overline{B} + 1$. Add this quantity to A to get $Y = A + \overline{B} + 1 = A - B$. This sum can be performed with a single CPA by adding $A + \overline{B}$ with $C_{in} = 1$. Figure 5.9 shows the symbol for a subtractor and the underlying hardware for performing $Y = A - B$. HDL Example 5.2 describes a subtractor.

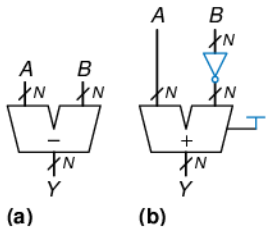


Figure 5.9 Subtractor: (a) symbol, (b) implementation

5.2.3 Comparators

A *comparator* determines whether two binary numbers are equal or if one is greater or less than the other. A comparator receives two N-bit binary numbers, A and B. There are two common types of comparators.

HDL Example 5.2 SUBTRACTOR

Verilog

```
module subtractor #(parameter N = 8)
                   (input  [N-1:0] a, b,
                    output [N-1:0] y);

  assign y = a - b;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity subtractor is
  generic (N: integer := 8);
  port (a, b: in  STD_LOGIC_VECTOR(N-1 downto 0);
            y: out STD_LOGIC_VECTOR(N-1 downto 0));
end;

architecture synth of subtractor is
begin
  y <= a - b;
end;
```

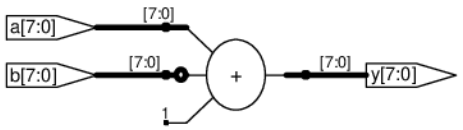


Figure 5.10 Synthesized subtractor

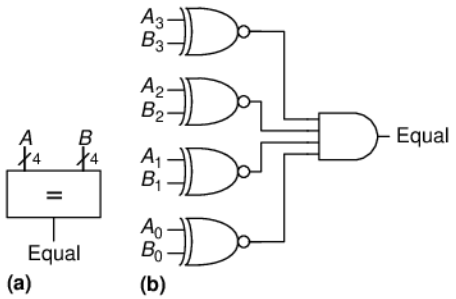


Figure 5.11 4-bit equality comparator: (a) symbol, (b) implementation

An *equality comparator* produces a single output indicating whether A is equal to B ($A == B$). A *magnitude comparator* produces one or more outputs indicating the relative values of A and B.

The equality comparator is the simpler piece of hardware. Figure 5.11 shows the symbol and implementation of a 4-bit equality comparator. It first checks to determine whether the corresponding bits in each column of A and B are equal, using XNOR gates. The numbers are equal if all of the columns are equal.

Magnitude comparison is usually done by computing $A - B$ and looking at the sign (most significant bit) of the result, as shown in Figure 5.12. If the result is negative (i.e., the sign bit is 1), then A is less than B. Otherwise A is greater than or equal to B.

HDL Example 5.3 shows how to use various comparison operations.

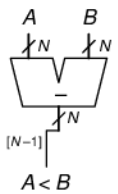


Figure 5.12 *N*-bit magnitude comparator

HDL Example 5.3 COMPARATORS

Verilog

```
module comparators # (parameter N = 8)
                (input [N-1:0] a, b,
                output         eq, neq,
                output         lt, lte,
                output         gt, gte);

  assign eq  = (a == b);
  assign neq = (a != b);
  assign lt  = (a < b);
  assign lte = (a <= b);
  assign gt  = (a > b);
  assign gte = (a >= b);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
entity comparators is
  generic (N: integer := 8);
  port (a, b: in STD_LOGIC_VECTOR(N-1 downto 0);
        eq, neq, lt,
        lte, gt, gte: out STD_LOGIC);
end;

architecture synth of comparators is
begin
  eq  <= '1' when (a = b)  else '0';
  neq <= '1' when (a /= b) else '0';
  lt  <= '1' when (a < b)  else '0';
  lte <= '1' when (a <= b) else '0';
  gt  <= '1' when (a > b)  else '0';
  gte <= '1' when (a >= b) else '0';
end;
```

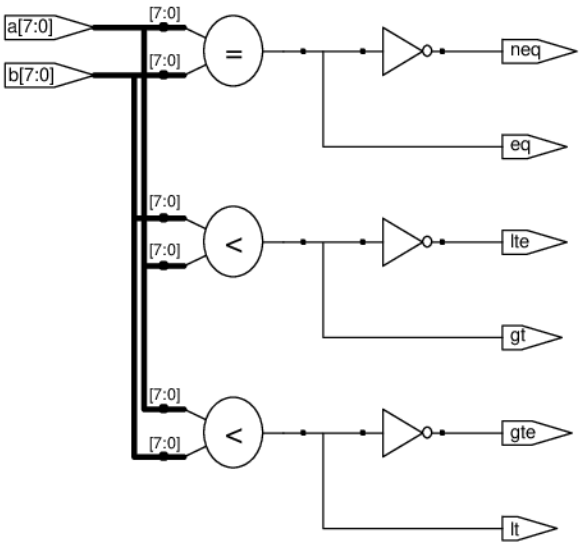


Figure 5.13 Synthesized comparators

5.2.4 ALU

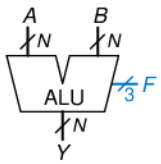


Figure 5.14 ALU symbol

An *Arithmetic/Logical Unit* (*ALU*) combines a variety of mathematical and logical operations into a single unit. For example, a typical ALU might perform addition, subtraction, magnitude comparison, AND, and OR operations. The ALU forms the heart of most computer systems.

Figure 5.14 shows the symbol for an *N*-bit ALU with *N*-bit inputs and outputs. The ALU receives a control signal, *F*, that specifies which

Table 5.1 ALU operations

$F_{2:0}$	Function
000	A AND B
001	A OR B
010	A + B
011	not used
100	A AND $\overline{B}$
101	A OR $\overline{B}$
110	A − B
111	SLT

function to perform. Control signals will generally be shown in blue to distinguish them from the data. Table 5.1 lists typical functions that the ALU can perform. The SLT function is used for magnitude comparison and will be discussed later in this section.

Figure 5.15 shows an implementation of the ALU. The ALU contains an N-bit adder and N two-input AND and OR gates. It also contains an inverter and a multiplexer to optionally invert input B when the F_2 control signal is asserted. A 4:1 multiplexer chooses the desired function based on the $F_{1:0}$ control signals.

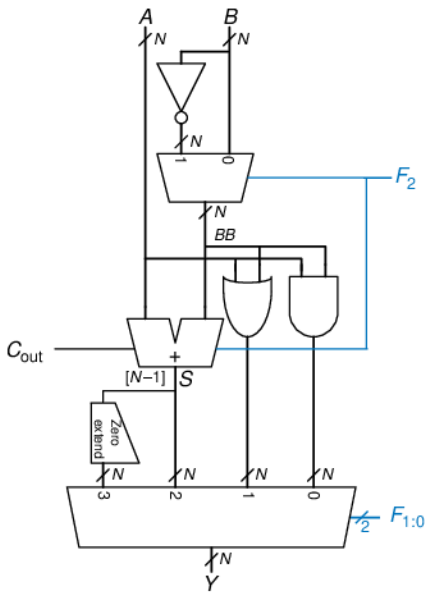


Figure 5.15 *N*-bit ALU

More specifically, the arithmetic and logical blocks in the ALU operate on A and BB. BB is either B or $\overline{B}$, depending on F_2. If $F_{1:0} = 00$, the output multiplexer chooses A AND BB. If $F_{1:0} = 01$, the ALU computes A OR BB. If $F_{1:0} = 10$, the ALU performs addition or subtraction. Note that F_2 is also the carry in to the adder. Also remember that $\overline{B} + 1 = -B$ in two's complement arithmetic. If $F_2 = 0$, the ALU computes $A + B$. If $F_2 = 1$, the ALU computes $A + \overline{B} + 1 = A - B$.

When $F_{2:0} = 111$, the ALU performs the *set if less than* (*SLT*) operation. When $A < B$, $Y = 1$. Otherwise, $Y = 0$. In other words, Y is set to 1 if A is less than B.

SLT is performed by computing $S = A - B$. If S is negative (i.e., the sign bit is set), $A < B$. The *zero extend unit* produces an N-bit output by concatenating its 1-bit input with 0's in the most significant bits. The sign bit (the $N-1^{\text{th}}$ bit) of S is the input to the zero extend unit.

Example 5.3 SET LESS THAN

Configure a 32-bit ALU for the SLT operation. Suppose $A = 25_{10}$ and $B = 32_{10}$. Show the control signals and output, Y.

Solution: Because $A < B$, we expect Y to be 1. For SLT, $F_{2:0} = 111$. With $F_2 = 1$, this configures the adder unit as a subtractor with an output, S, of $25_{10} - 32_{10} = -7_{10} = 1111 \ldots 1001_2$. With $F_{1:0} = 11$, the final multiplexer sets $Y = S_{31} = 1$.

Some ALUs produce extra outputs, called *flags*, that indicate information about the ALU output. For example, an *overflow flag* indicates that the result of the adder overflowed. A *zero flag* indicates that the ALU output is 0.

The HDL for an N-bit ALU is left to Exercise 5.9. There are many variations on this basic ALU that support other functions, such as XOR or equality comparison.

5.2.5 Shifters and Rotators

Shifters and *rotators* move bits and multiply or divide by powers of 2. As the name implies, a shifter shifts a binary number left or right by a specified number of positions. There are several kinds of commonly used shifters:

- **Logical shifter**—shifts the number to the left (LSL) or right (LSR) and fills empty spots with 0's.

  Ex: 11001 LSR 2 = 00110; 11001 LSL 2 = 00100

- **Arithmetic shifter**—is the same as a logical shifter, but on right shifts fills the most significant bits with a copy of the old most significant bit (msb). This is useful for multiplying and dividing signed numbers

(see Sections 5.2.6 and 5.2.7). Arithmetic shift left (ASL) is the same as logical shift left (LSL).

Ex: 11001 ASR 2 = 11110; 11001 ASL 2 = 00100

- **Rotator**—rotates number in circle such that empty spots are filled with bits shifted off the other end.

Ex: 11001 ROR 2 = 01110; 11001 ROL 2 = 00111

An N-bit shifter can be built from N N:1 multiplexers. The input is shifted by 0 to $N-1$ bits, depending on the value of the $\log_2 N$-bit select lines. Figure 5.16 shows the symbol and hardware of 4-bit shifters. The operators <<, >>, and >>> typically indicate shift left, logical shift right, and arithmetic shift right, respectively. Depending on the value of the 2-bit shift amount, $shamt_{1:0}$, the output, Y, receives the input, A, shifted by 0 to 3 bits. For all shifters, when $shamt_{1:0} = 00$, $Y = A$. Exercise 5.14 covers rotator designs.

A left shift is a special case of multiplication. A left shift by N bits multiplies the number by 2^N. For example, $000011_2 << 4 = 110000_2$ is equivalent to $3_{10} \times 2^4 = 48_{10}$.

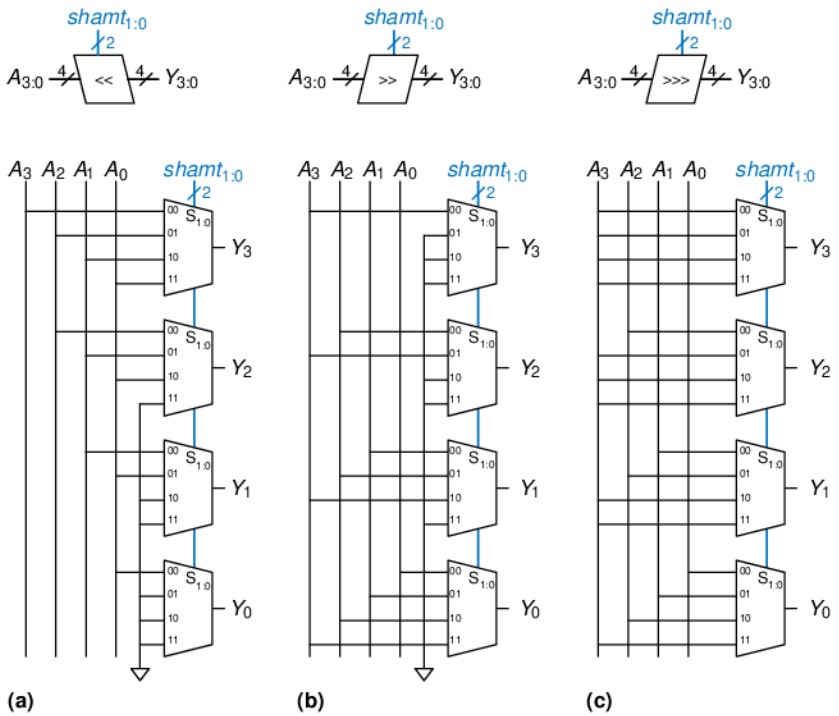


Figure 5.16 4-bit shifters: (a) shift left, (b) logical shift right, (c) arithmetic shift right

An arithmetic right shift is a special case of division. An arithmetic right shift by N bits divides the number by 2^N. For example, $11100_2 >>> 2 = 11111_2$ is equivalent to $-4_{10}/2^2 = -1_{10}$.

5.2.6 Multiplication*

Multiplication of unsigned binary numbers is similar to decimal multiplication but involves only 1's and 0's. Figure 5.17 compares multiplication in decimal and binary. In both cases, *partial products* are formed by multiplying a single digit of the multiplier with the entire multiplicand. The shifted partial products are summed to form the result.

In general, an $N \times N$ multiplier multiplies two N-bit numbers and produces a $2N$-bit result. The partial products in binary multiplication are either the multiplicand or all 0's. Multiplication of 1-bit binary numbers is equivalent to the AND operation, so AND gates are used to form the partial products.

Figure 5.18 shows the symbol, function, and implementation of a 4 $\times$ 4 multiplier. The multiplier receives the multiplicand and multiplier, A and B, and produces the product, P. Figure 5.18(b) shows how partial products are formed. Each partial product is a single multiplier bit (B_3, B_2, B_1, or B_0) AND the multiplicand bits (A_3, A_2, A_1, A_0). With N-bit

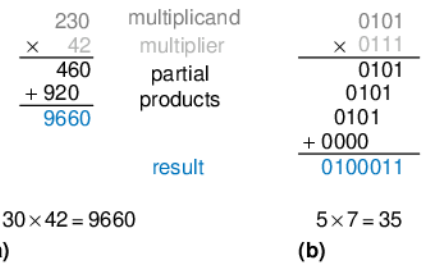


Figure 5.17 Multiplication: (a) decimal, (b) binary

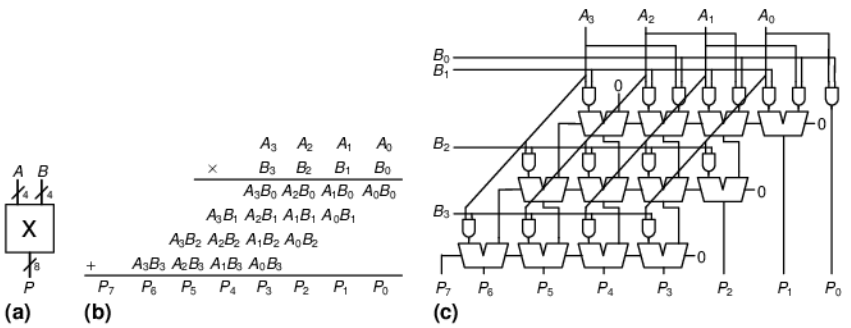


Figure 5.18 4 × 4 multiplier: (a) symbol, (b) function, (c) implementation

operands, there are N partial products and $N - 1$ stages of 1-bit adders. For example, for a 4×4 multiplier, the partial product of the first row is B_0 AND (A_3, A_2, A_1, A_0). This partial product is added to the shifted second partial product, B_1 AND (A_3, A_2, A_1, A_0). Subsequent rows of AND gates and adders form and add the remaining partial products.

The HDL for a multiplier is in HDL Example 5.4. As with adders, many different multiplier designs with different speed/cost trade-offs exist. Synthesis tools may pick the most appropriate design given the timing constraints.

HDL Example 5.4 MULTIPLIER

Verilog

```
module multiplier # (parameter N = 8)
                   (input  [N-1:0]   a, b,
                    output [2*N-1:0] y);

  assign y = a * b;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity multiplier is
  generic (N: integer := 8);
  port (a, b: in  STD_LOGIC_VECTOR(N-1 downto 0);
              y: out STD_LOGIC_VECTOR(2*N-1 downto 0));
end;

architecture synth of multiplier is
begin
  y <= a * b;
end;
```

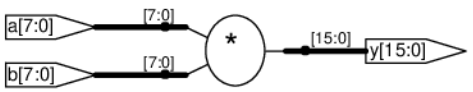


Figure 5.19 Synthesized multiplier

5.2.7 Division*

Binary division can be performed using the following algorithm for normalized unsigned numbers in the range $[2^N - 1, 2^{N-1}]$:

```
R = A
for i = N-1 to 0
  D = R - B
  if D < 0 then    Qi = 0, R' = R   // R < B
  else             Qi = 1, R' = D   // R ≥ B
  if i ≠ 0 then R = 2R'
```

The *partial remainder*, R, is initialized to the dividend, A. The divisor, B, is repeatedly subtracted from this partial remainder to determine whether it fits. If the difference, D, is negative (i.e., the sign bit of D is 1), then the quotient bit, Q_i, is 0 and the difference is discarded. Otherwise, Q_i is 1,

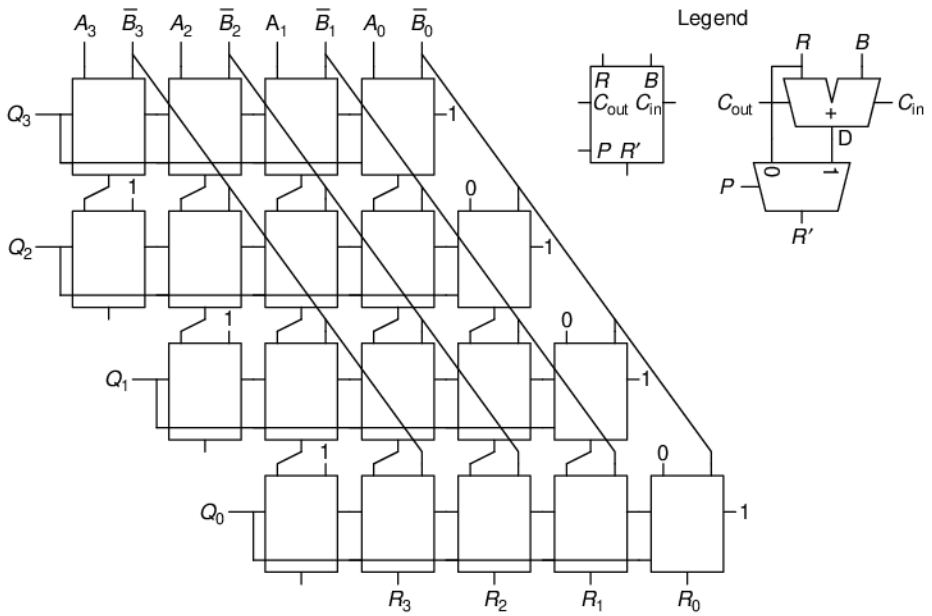


Figure 5.20 Array divider

and the partial remainder is updated to be the difference. In any event, the partial remainder is then doubled (left-shifted by one column), and the process repeats. The result satisfies $\frac{A}{B} = \left(Q + \frac{R}{B}\right)2^{-(N-1)}$.

Figure 5.20 shows a schematic of a 4-bit array divider. The divider computes A/B and produces a quotient, Q, and a remainder, R. The legend shows the symbol and schematic for each block in the array divider. The signal P indicates whether $R - B$ is negative. It is obtained from the C_{out} output of the leftmost block in the row, which is the sign of the difference.

The delay of an N-bit array divider increases proportionally to N^2 because the carry must ripple through all N stages in a row before the sign is determined and the multiplexer selects R or D. This repeats for all N rows. Division is a slow and expensive operation in hardware and therefore should be used as infrequently as possible.

5.2.8 Further Reading

Computer arithmetic could be the subject of an entire text. *Digital Arithmetic,* by Ercegovac and Lang, is an excellent overview of the entire field. *CMOS VLSI Design,* by Weste and Harris, covers high-performance circuit designs for arithmetic operations.

5.3 NUMBER SYSTEMS

Computers operate on both integers and fractions. So far, we have only considered representing signed or unsigned integers, as introduced in Section 1.4. This section introduces fixed- and floating-point number systems that can also represent rational numbers. Fixed-point numbers are analogous to decimals; some of the bits represent the integer part, and the rest represent the fraction. Floating-point numbers are analogous to scientific notation, with a mantissa and an exponent.

5.3.1 Fixed-Point Number Systems

Fixed-point notation has an implied *binary point* between the integer and fraction bits, analogous to the decimal point between the integer and fraction digits of an ordinary decimal number. For example, Figure 5.21(a) shows a fixed-point number with four integer bits and four fraction bits. Figure 5.21(b) shows the implied binary point in blue, and Figure 5.21(c) shows the equivalent decimal value.

(a) 01101100

(b) 0110.1100

(c) $2^2+2^1+2^{-1}+2^{-2}=6.75$

Figure 5.21 Fixed-point notation of 6.75 with four integer bits and four fraction bits

Signed fixed-point numbers can use either two's complement or sign/magnitude notations. Figure 5.22 shows the fixed-point representation of -2.375 using both notations with four integer and four fraction bits. The implicit binary point is shown in blue for clarity. In sign/magnitude form, the most significant bit is used to indicate the sign. The two's complement representation is formed by inverting the bits of the absolute value and adding a 1 to the least significant (rightmost) bit. In this case, the least significant bit position is in the 2^{-4} column.

(a) 0010.0110

(b) 1010.0110

(c) 1101.1010

Figure 5.22 Fixed-point representation of -2.375: (a) absolute value, (b) sign and magnitude, (c) two's complement

Like all binary number representations, fixed-point numbers are just a collection of bits. There is no way of knowing the existence of the binary point except through agreement of those people interpreting the number.

Example 5.4 ARITHMETIC WITH FIXED-POINT NUMBERS

Compute $0.75 + -0.625$ using fixed-point numbers.

Solution: First convert 0.625, the magnitude of the second number, to fixed-point binary notation. $0.625 \geq 2^{-1}$, so there is a 1 in the 2^{-1} column, leaving $0.625 - 0.5 = 0.125$. Because $0.125 < 2^{-2}$, there is a 0 in the 2^{-2} column. Because $0.125 \geq 2^{-3}$, there is a 1 in the 2^{-3} column, leaving $0.125 - 0.125 = 0$. Thus, there must be a 0 in the 2^{-4} column. Putting this all together, $0.625_{10} = 0000.1010_2$

Use two's complement representation for signed numbers so that addition works correctly. Figure 5.23 shows the conversion of -0.625 to fixed-point two's complement notation.

Figure 5.24 shows the fixed-point binary addition and the decimal equivalent for comparison. Note that the leading 1 in the binary fixed-point addition of Figure 5.24(a) is discarded from the 8-bit result.

Fixed-point number systems are commonly used for banking and financial applications that require precision but not a large range.

Figure 5.23 Fixed-point two's complement conversion

```
  0000.1010   Binary Magnitude
  1111.0101   One's Complement
+        1    Add 1
  1111.0110   Two's Complement
```

Figure 5.24 Addition: (a) binary fixed-point (b) decimal equivalent

```
  0000.1100          0.75
+ 1111.0110        + (–0.625)
 10000.0010          0.125
(a)                 (b)
```

5.3.2 Floating-Point Number Systems*

Floating-point numbers are analogous to scientific notation. They circumvent the limitation of having a constant number of integer and fractional bits, allowing the representation of very large and very small numbers. Like scientific notation, floating-point numbers have a *sign*, *mantissa* (M), *base* (B), and *exponent* (E), as shown in Figure 5.25. For example, the number 4.1×10^3 is the decimal scientific notation for 4100. It has a mantissa of 4.1, a base of 10, and an exponent of 3. The decimal point *floats* to the position right after the most significant digit. Floating-point numbers are base 2 with a binary mantissa. 32 bits are used to represent 1 sign bit, 8 exponent bits, and 23 mantissa bits.

$$\pm M \times B^E$$

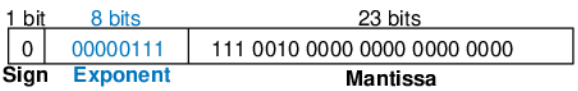
Figure 5.25 Floating-point numbers

Example 5.5 32-BIT FLOATING-POINT NUMBERS

Show the floating-point representation of the decimal number 228.

Solution: First convert the decimal number into binary: $228_{10} = 11100100_2 = 1.11001_2 \times 2^7$. Figure 5.26 shows the 32-bit encoding, which will be modified later for efficiency. The sign bit is positive (0), the 8 exponent bits give the value 7, and the remaining 23 bits are the mantissa.

Figure 5.26 32-bit floating-point version 1



In binary floating-point, the first bit of the mantissa (to the left of the binary point) is always 1 and therefore need not be stored. It is called the *implicit leading one*. Figure 5.27 shows the modified floating-point representation of $228_{10} = 11100100_2 \times 2^0 = 1.11001_2 \times 2^7$. The implicit leading one is not included in the 23-bit mantissa for efficiency. Only the fraction bits are stored. This frees up an extra bit for useful data.

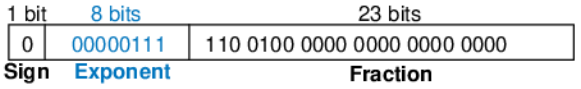


Figure 5.27 Floating-point version 2

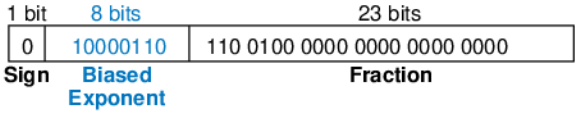


Figure 5.28 IEEE 754 floating-point notation

We make one final modification to the exponent field. The exponent needs to represent both positive and negative exponents. To do so, floating-point uses a *biased* exponent, which is the original exponent plus a constant bias. 32-bit floating-point uses a bias of 127. For example, for the exponent 7, the biased exponent is $7 + 127 = 134 = 10000110_2$. For the exponent -4, the biased exponent is: $-4 + 127 = 123 = 01111011_2$. Figure 5.28 shows $1.11001_2 \times 2^7$ represented in floating-point notation with an implicit leading one and a biased exponent of 134 (7 + 127). This notation conforms to the IEEE 754 floating-point standard.

As may be apparent, there are many reasonable ways to represent floating-point numbers. For many years, computer manufacturers used incompatible floating-point formats. Results from one computer could not directly be interpreted by another computer.

The Institute of Electrical and Electronics Engineers solved this problem by defining the *IEEE 754 floating-point standard* in 1985 defining floating-point numbers. This floating-point format is now almost universally used and is the one discussed in this section.

Special Cases: 0, ±∞, and NaN

The IEEE floating-point standard has special cases to represent numbers such as zero, infinity, and illegal results. For example, representing the number zero is problematic in floating-point notation because of the implicit leading one. Special codes with exponents of all 0's or all 1's are reserved for these special cases. Table 5.2 shows the floating-point representations of 0, ±∞, and NaN. As with sign/magnitude numbers, floating-point has both positive and negative 0. NaN is used for numbers that don't exist, such as $\sqrt{-1}$ or $\log_2(-5)$.

Single- and Double-Precision Formats

So far, we have examined 32-bit floating-point numbers. This format is also called *single-precision*, *single*, or *float*. The IEEE 754 standard also

Table 5.2 IEEE 754 floating-point notations for 0, ±∞, and NaN

Number	Sign	Exponent	Fraction
0	X	00000000	00000000000000000000000
∞	0	11111111	00000000000000000000000
−∞	1	11111111	00000000000000000000000
NaN	X	11111111	non-zero

Table 5.3 Single- and double-precision floating-point formats

Format	Total Bits	Sign Bits	Exponent Bits	Fraction Bits
single	32	1	8	23
double	64	1	11	52

Floating-point cannot represent some numbers exactly, like 1.7. However, when you type 1.7 into your calculator, you see exactly 1.7, not 1.69999. . . . To handle this, some applications, such as calculators and financial software, use *binary coded decimal* (*BCD*) numbers or formats with a base 10 exponent. BCD numbers encode each decimal digit using four bits with a range of 0 to 9. For example the BCD fixed-point notation of 1.7 with four integer bits and four fraction bits would be 0001.0111. Of course, nothing is free. The cost is increased complexity in arithmetic hardware and wasted encodings (A–F encodings are not used), and thus decreased performance. So for compute-intensive applications, floating-point is much faster.

defines 64-bit *double-precision* (also called *double*) numbers that provide greater precision and greater range. Table 5.3 shows the number of bits used for the fields in each format.

Excluding the special cases mentioned earlier, normal single-precision numbers span a range of $\pm 1.175494 \times 10^{-38}$ to $\pm 3.402824 \times 10^{38}$. They have a precision of about seven significant decimal digits (because $2^{-24} \approx 10^{-7}$). Similarly, normal double-precision numbers span a range of $\pm 2.22507385850720 \times 10^{-308}$ to $\pm 1.79769313486232 \times 10^{308}$ and have a precision of about 15 significant decimal digits.

Rounding

Arithmetic results that fall outside of the available precision must round to a neighboring number. The rounding modes are: (1) round down, (2) round up, (3) round toward zero, and (4) round to nearest. The default rounding mode is round to nearest. In the round to nearest mode, if two numbers are equally near, the one with a 0 in the least significant position of the fraction is chosen.

Recall that a number *overflows* when its magnitude is too large to be represented. Likewise, a number *underflows* when it is too tiny to be represented. In round to nearest mode, overflows are rounded up to $\pm\infty$ and underflows are rounded down to 0.

Floating-Point Addition

Addition with floating-point numbers is not as simple as addition with two's complement numbers. The steps for adding floating-point numbers with the same sign are as follows:

1. Extract exponent and fraction bits.
2. Prepend leading 1 to form the mantissa.
3. Compare exponents.
4. Shift smaller mantissa if necessary.
5. Add mantissas.
6. Normalize mantissa and adjust exponent if necessary.
7. Round result.
8. Assemble exponent and fraction back into floating-point number.

Figure 5.29 shows the floating-point addition of 7.875 (1.11111×2^2) and 0.1875 (1.1×2^{-3}). The result is 8.0625 (1.0000001×2^3). After the fraction and exponent bits are extracted and the implicit leading 1 is prepended in steps 1 and 2, the exponents are compared by subtracting the smaller exponent from the larger exponent. The result is the number of bits by which the smaller number is shifted to the right to align the implied binary point (i.e., to make the exponents equal) in step 4. The aligned numbers are added. Because the sum has a mantissa that is greater than or equal to 2.0, the result is normalized by shifting it to the right one bit and incrementing the exponent. In this example, the result is exact, so no rounding is necessary. The result is stored in floating-point notation by removing the implicit leading one of the mantissa and prepending the sign bit.

Floating-point arithmetic is usually done in hardware to make it fast. This hardware, called the *floating-point unit* (*FPU*), is typically distinct from the *central processing unit* (*CPU*). The infamous *floating-point division* (*FDIV*) bug in the Pentium FPU cost Intel $475 million to recall and replace defective chips. The bug occurred simply because a lookup table was not loaded correctly.

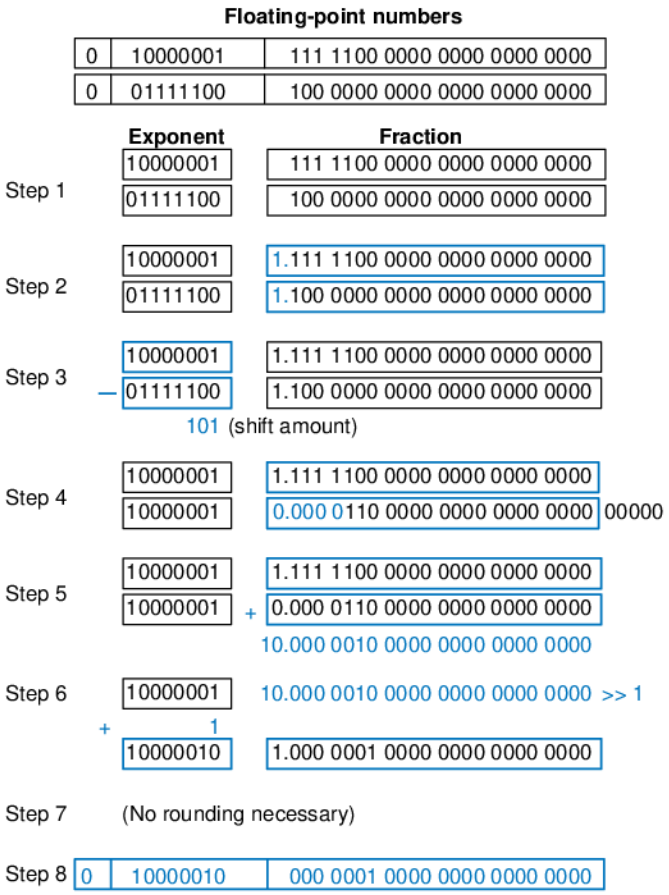


Figure 5.29 Floating-point addition

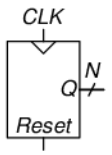


Figure 5.30 Counter symbol

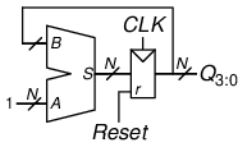


Figure 5.31 *N*-bit counter

5.4 SEQUENTIAL BUILDING BLOCKS

This section examines sequential building blocks, including counters and shift registers.

5.4.1 Counters

An *N*-bit *binary counter*, shown in Figure 5.30, is a sequential arithmetic circuit with clock and reset inputs and an *N*-bit output, *Q*. *Reset* initializes the output to 0. The counter then advances through all 2^N possible outputs in binary order, incrementing on the rising edge of the clock.

Figure 5.31 shows an *N*-bit counter composed of an adder and a resettable register. On each cycle, the counter adds 1 to the value stored in the register. HDL Example 5.5 describes a binary counter with asynchronous reset.

Other types of counters, such as Up/Down counters, are explored in Exercises 5.37 through 5.40.

HDL Example 5.5 COUNTER

Verilog

```
module counter #(parameter N = 8)
               (input              clk,
                input              reset,
                output reg [N-1:0] q);

  always @ (posedge clk or posedge reset)
    if (reset) q <= 0;
    else       q <= q + 1;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;

entity counter is
  generic (N: integer := 8);
  port (clk,
        reset: in STD_LOGIC;
        q: buffer STD_LOGIC_VECTOR(N-1 downto 0));
end;

architecture synth of counter is
begin
  process (clk, reset) begin
    if reset = '1' then
      q <= CONV_STD_LOGIC_VECTOR (0, N);
    elsif clk'event and clk = '1' then
      q <= q + 1;
    end if;
  end process;
end;
```

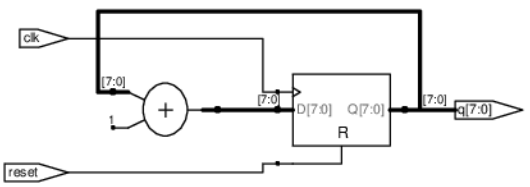


Figure 5.32 Synthesized counter

5.4.2 Shift Registers

A *shift register* has a clock, a serial input, S_{in}, a serial output, S_{out}, and N parallel outputs, $Q_{N-1:0}$, as shown in Figure 5.33. On each rising edge of the clock, a new bit is shifted in from S_{in} and all the subsequent contents are shifted forward. The last bit in the shift register is available at S_{out}. Shift registers can be viewed as *serial-to-parallel converters*. The input is provided serially (one bit at a time) at S_{in}. After N cycles, the past N inputs are available in parallel at Q.

A shift register can be constructed from N flip-flops connected in series, as shown in Figure 5.34. Some shift registers also have a reset signal to initialize all of the flip-flops.

A related circuit is a *parallel-to-serial* converter that loads N bits in parallel, then shifts them out one at a time. A shift register can be modified to perform both serial-to-parallel and parallel-to-serial operations by adding a parallel input, $D_{N-1:0}$, and a control signal, *Load*, as shown in Figure 5.35. When *Load* is asserted, the flip-flops are loaded in parallel from the D inputs. Otherwise, the shift register shifts normally. HDL Example 5.6 describes such a shift register.

Don't confuse *shift registers* with the *shifters* from Section 5.2.5. Shift registers are sequential logic blocks that shift in a new bit on each clock edge. Shifters are unclocked combinational logic blocks that shift an input by a specified amount.

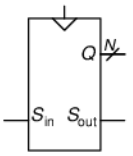


Figure 5.33 Shift register symbol

Scan Chains*

Shift registers are often used to test sequential circuits using a technique called *scan chains*. Testing combinational circuits is relatively straightforward. Known inputs called *test vectors* are applied, and the outputs are checked against the expected result. Testing sequential circuits is more difficult, because the circuits have state. Starting from a known initial condition, a large number of cycles of test vectors may be needed to put the circuit into a desired state. For example, testing that the most significant bit of a 32-bit counter advances from 0 to 1 requires resetting the counter, then applying 2^{31} (about two billion) clock pulses!

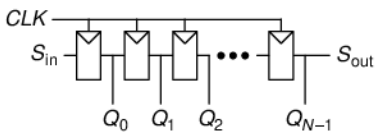


Figure 5.34 Shift register schematic

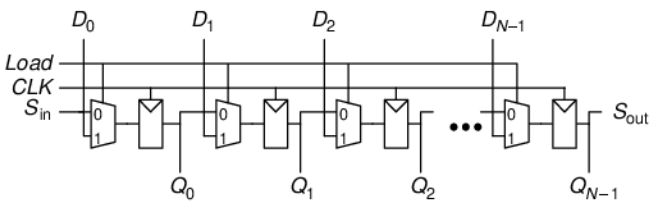


Figure 5.35 Shift register with parallel load

HDL Example 5.6 SHIFT REGISTER WITH PARALLEL LOAD

Verilog

```
module shiftreg # (parameter N = 8)
                (input              clk,
                 input              reset, load,
                 input              sin,
                 input      [N-1:0] d,
                 output reg [N-1:0] q,
                 output             sout);

  always @ (posedge clk or posedge reset)
    if (reset)       q <= 0;
    else if (load)   q <= d;
    else             q <= {q[N-2:0], sin};

  assign sout = q[N-1];
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity shiftreg is
  generic (N: integer := 8);
  port(clk, reset,
       load: in    STD_LOGIC;
       sin:  in    STD_LOGIC;
       d:    in    STD_LOGIC_VECTOR(N-1 downto 0);
       q: buffer   STD_LOGIC_VECTOR(N-1 downto 0);
       sout: out   STD_LOGIC);
end;

architecture synth of shiftreg is
begin
  process (clk, reset) begin
    if reset = '1' then
      q <= CONV_STD_LOGIC_VECTOR (0, N);
    elsif clk'event and clk = '1' then
      if load = '1' then
        q <= d;
      else
        q <= q(N-2 downto 0) & sin;
      end if;
    end if;
  end process;
  sout <= q(N-1);
end;
```

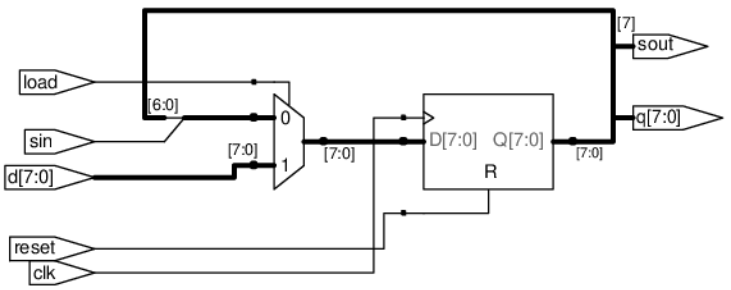


Figure 5.36 Synthesized shiftreg

To solve this problem, designers like to be able to directly observe and control all the state of the machine. This is done by adding a test mode in which the contents of all flip-flops can be read out or loaded with desired values. Most systems have too many flip-flops to dedicate individual pins to read and write each flip-flop. Instead, all the flip-flops in the system are connected together into a shift register called a scan chain. In normal operation, the flip-flops load data from their *D* input and ignore the scan chain. In test mode, the flip-flops serially shift their contents out and shift

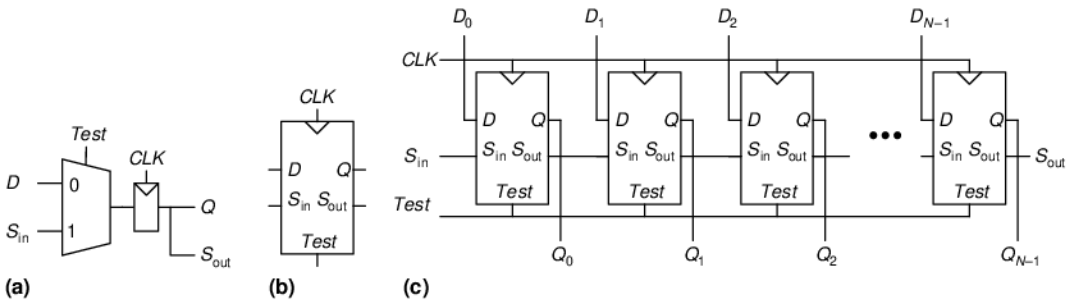


Figure 5.37 Scannable flip-flop: (a) schematic, (b) symbol, and (c) *N*-bit scannable register

in new contents using S_{in} and S_{out}. The load multiplexer is usually integrated into the flip-flop to produce a *scannable flip-flop*. Figure 5.37 shows the schematic and symbol for a scannable flip-flop and illustrates how the flops are cascaded to build an *N*-bit scannable register.

For example, the 32-bit counter could be tested by shifting in the pattern 011111 . . . 111 in test mode, counting for one cycle in normal mode, then shifting out the result, which should be 100000 . . . 000. This requires only 32 + 1 + 32 = 65 cycles.

5.5 MEMORY ARRAYS

The previous sections introduced arithmetic and sequential circuits for manipulating data. Digital systems also require *memories* to store the data used and generated by such circuits. Registers built from flip-flops are a kind of memory that stores small amounts of data. This section describes *memory arrays* that can efficiently store large amounts of data.

The section begins with an overview describing characteristics shared by all memory arrays. It then introduces three types of memory arrays: dynamic random access memory (DRAM), static random access memory (SRAM), and read only memory (ROM). Each memory differs in the way it stores data. The section briefly discusses area and delay trade-offs and shows how memory arrays are used, not only to store data but also to perform logic functions. The section finishes with the HDL for a memory array.

5.5.1 Overview

Figure 5.38 shows a generic symbol for a memory array. The memory is organized as a two-dimensional array of memory cells. The memory reads or writes the contents of one of the rows of the array. This row is

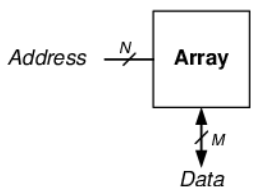


Figure 5.38 Generic memory array symbol

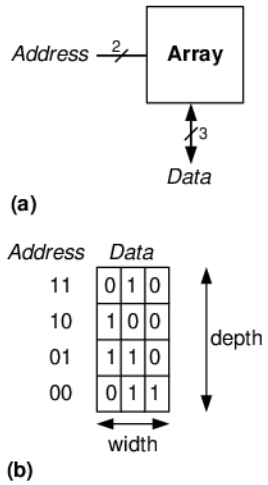


Figure 5.39 **4 × 3 memory array: (a) symbol, (b) function**

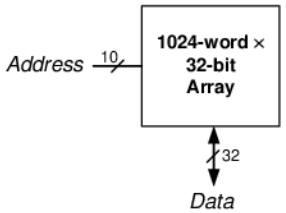


Figure 5.40 **32 Kb array: depth = 2^{10} = 1024 words, width = 32 bits**

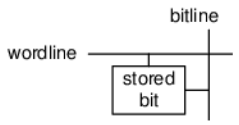


Figure 5.41 **Bit cell**

specified by an *Address*. The value read or written is called *Data*. An array with N-bit addresses and M-bit data has 2^N rows and M columns. Each row of data is called a *word*. Thus, the array contains 2^N M-bit words.

Figure 5.39 shows a memory array with two address bits and three data bits. The two address bits specify one of the four rows (data words) in the array. Each data word is three bits wide. Figure 5.39(b) shows some possible contents of the memory array.

The *depth* of an array is the number of rows, and the *width* is the number of columns, also called the word size. The size of an array is given as *depth* × *width*. Figure 5.39 is a 4-word × 3-bit array, or simply 4 × 3 array. The symbol for a 1024-word × 32-bit array is shown in Figure 5.40. The total size of this array is 32 kilobits (Kb).

Bit Cells

Memory arrays are built as an array of *bit cells*, each of which stores 1 bit of data. Figure 5.41 shows that each bit cell is connected to a *wordline* and a *bitline*. For each combination of address bits, the memory asserts a single wordline that activates the bit cells in that row. When the wordline is HIGH, the stored bit transfers to or from the bitline. Otherwise, the bitline is disconnected from the bit cell. The circuitry to store the bit varies with memory type.

To read a bit cell, the bitline is initially left floating (Z). Then the wordline is turned ON, allowing the stored value to drive the bitline to 0 or 1. To write a bit cell, the bitline is strongly driven to the desired value. Then the wordline is turned ON, connecting the bitline to the stored bit. The strongly driven bitline overpowers the contents of the bit cell, writing the desired value into the stored bit.

Organization

Figure 5.42 shows the internal organization of a 4 × 3 memory array. Of course, practical memories are much larger, but the behavior of larger arrays can be extrapolated from the smaller array. In this example, the array stores the data from Figure 5.39(b).

During a memory read, a wordline is asserted, and the corresponding row of bit cells drives the bitlines HIGH or LOW. During a memory write, the bitlines are driven HIGH or LOW first and then a wordline is asserted, allowing the bitline values to be stored in that row of bit cells. For example, to read *Address* 10, the bitlines are left floating, the decoder asserts $wordline_2$, and the data stored in that row of bit cells, 100, reads out onto the *Data* bitlines. To write the value 001 to *Address* 11, the bitlines are driven to the value 001, then $wordline_3$ is asserted and the new value (001) is stored in the bit cells.

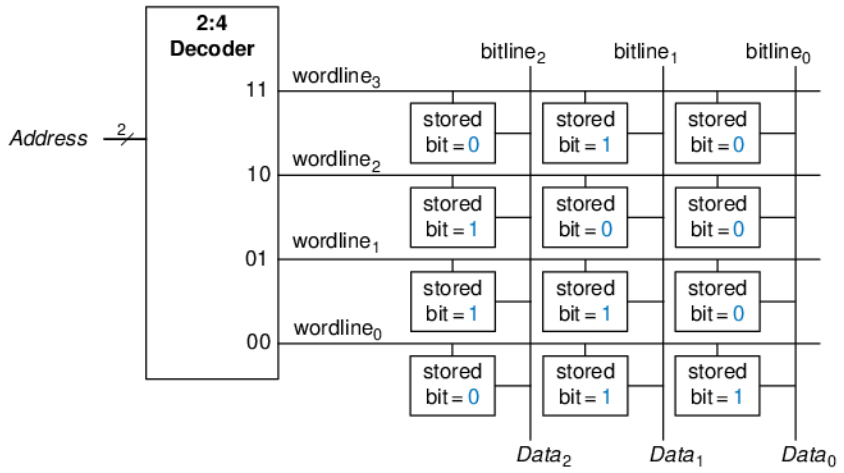


Figure 5.42 4 × 3 memory array

Memory Ports

All memories have one or more *ports*. Each port gives read and/or write access to one memory address. The previous examples were all single-ported memories.

Multiported memories can access several addresses simultaneously. Figure 5.43 shows a three-ported memory with two read ports and one write port. Port 1 reads the data from address *A1* onto the read data output *RD1*. Port 2 reads the data from address *A2* onto *RD2*. Port 3 writes the data from the write data input, *WD3*, into address *A3* on the rising edge of the clock if the write enable, *WE3*, is asserted.

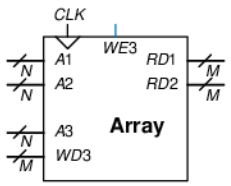


Figure 5.43 Three-ported memory

Memory Types

Memory arrays are specified by their size (depth × width) and the number and type of ports. All memory arrays store data as an array of bit cells, but they differ in how they store bits.

Memories are classified based on how they store bits in the bit cell. The broadest classification is *random access memory* (*RAM*) versus *read only memory* (*ROM*). RAM is *volatile*, meaning that it loses its data when the power is turned off. ROM is *nonvolatile*, meaning that it retains its data indefinitely, even without a power source.

RAM and ROM received their names for historical reasons that are no longer very meaningful. RAM is called *random* access memory because any data word is accessed with the same delay as any other. A sequential access memory, such as a tape recorder, accesses nearby data more quickly than faraway data (e.g., at the other end of the tape).

Robert Dennard, 1932–. Invented DRAM in 1966 at IBM. Although many were skeptical that the idea would work, by the mid-1970s DRAM was in virtually all computers. He claims to have done little creative work until, arriving at IBM, they handed him a patent notebook and said, "put all your ideas in there." Since 1965, he has received 35 patents in semiconductors and microelectronics. (Photo courtesy of IBM.)

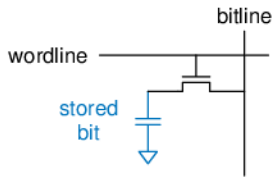


Figure 5.44 DRAM bit cell

ROM is called *read only* memory because, historically, it could only be read but not written. These names are confusing, because ROMs are randomly accessed too. Worse yet, most modern ROMs can be written as well as read! The important distinction to remember is that RAMs are volatile and ROMs are nonvolatile.

The two major types of RAMs are *dynamic RAM* (*DRAM*) and *static RAM* (*SRAM*). Dynamic RAM stores data as a charge on a capacitor, whereas static RAM stores data using a pair of cross-coupled inverters. There are many flavors of ROMs that vary by how they are written and erased. These various types of memories are discussed in the subsequent sections.

5.5.2 Dynamic Random Access Memory

Dynamic RAM (*DRAM*, pronounced "dee-ram") stores a bit as the presence or absence of charge on a capacitor. Figure 5.44 shows a DRAM bit cell. The bit value is stored on a capacitor. The nMOS transistor behaves as a switch that either connects or disconnects the capacitor from the bitline. When the wordline is asserted, the nMOS transistor turns ON, and the stored bit value transfers to or from the bitline.

As shown in Figure 5.45(a), when the capacitor is charged to V_{DD}, the stored bit is 1; when it is discharged to GND (Figure 5.45(b)), the stored bit is 0. The capacitor node is *dynamic* because it is not actively driven HIGH or LOW by a transistor tied to V_{DD} or GND.

Upon a read, data values are transferred from the capacitor to the bitline. Upon a write, data values are transferred from the bitline to the capacitor. Reading destroys the bit value stored on the capacitor, so the data word must be restored (rewritten) after each read. Even when DRAM is not read, the contents must be refreshed (read and rewritten) every few milliseconds, because the charge on the capacitor gradually leaks away.

5.5.3 Static Random Access Memory (SRAM)

Static RAM (*SRAM*, pronounced "es-ram") is *static* because stored bits do not need to be refreshed. Figure 5.46 shows an SRAM bit cell. The data bit is stored on cross-coupled inverters like those described in Section 3.2. Each cell has two outputs, bitline and $\overline{\text{bitline}}$. When the wordline is asserted, both nMOS transistors turn on, and data values are transferred to or from the bitlines. Unlike DRAM, if noise degrades the value of the stored bit, the cross-coupled inverters restore the value.

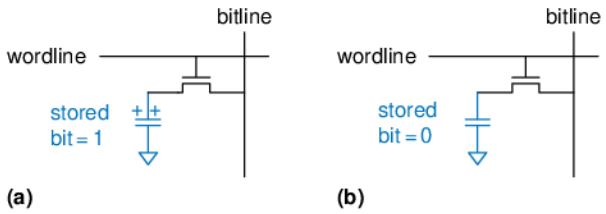


Figure 5.45 DRAM stored values

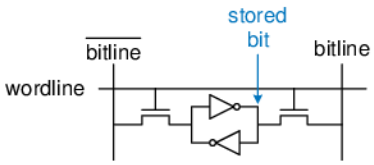


Figure 5.46 SRAM bit cell

Table 5.4 Memory comparison

Memory Type	Transistors per Bit Cell	Latency
flip-flop	~20	fast
SRAM	6	medium
DRAM	1	slow

5.5.4 Area and Delay

Flip-flops, SRAMs, and DRAMs are all volatile memories, but each has different area and delay characteristics. Table 5.4 shows a comparison of these three types of volatile memory. The data bit stored in a flip-flop is available immediately at its output. But flip-flops take at least 20 transistors to build. Generally, the more transistors a device has, the more area, power, and cost it requires. DRAM latency is longer than that of SRAM because its bitline is not actively driven by a transistor. DRAM must wait for charge to move (relatively) slowly from the capacitor to the bitline. DRAM also has lower throughput than SRAM, because it must refresh data periodically and after a read.

Memory latency and throughput also depend on memory size; larger memories tend to be slower than smaller ones if all else is the same. The best memory type for a particular design depends on the speed, cost, and power constraints.

5.5.5 Register Files

Digital systems often use a number of registers to store temporary variables. This group of registers, called a *register file*, is usually built as a small, multiported SRAM array, because it is more compact than an array of flip-flops.

Figure 5.47 shows a 32-register × 32-bit three-ported register file built from a three-ported memory similar to that of Figure 5.43. The

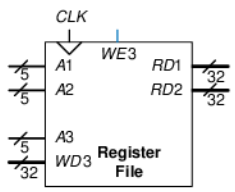


Figure 5.47 32 × 32 register file with two read ports and one write port

register file has two read ports (*A1*/*RD1* and *A2*/*RD2*) and one write port (*A3*/*WD3*). The 5-bit addresses, *A1*, *A2*, and *A3*, can each access all $2^5 = 32$ registers. So, two registers can be read and one register written simultaneously.

5.5.6 Read Only Memory

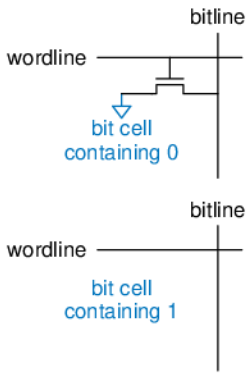


Figure 5.48 ROM bit cells containing 0 and 1

Read only memory (*ROM*) stores a bit as the presence or absence of a transistor. Figure 5.48 shows a simple ROM bit cell. To read the cell, the bitline is weakly pulled HIGH. Then the wordline is turned ON. If the transistor is present, it pulls the bitline LOW. If it is absent, the bitline remains HIGH. Note that the ROM bit cell is a combinational circuit and has no state to "forget" if power is turned off.

The contents of a ROM can be indicated using *dot notation*. Figure 5.49 shows the dot notation for a 4-word × 3-bit ROM containing the data from Figure 5.39. A dot at the intersection of a row (wordline) and a column (bitline) indicates that the data bit is 1. For example, the top wordline has a single dot on $Data_1$, so the data word stored at *Address* 11 is 010.

Conceptually, ROMs can be built using two-level logic with a group of AND gates followed by a group of OR gates. The AND gates produce all possible minterms and hence form a decoder. Figure 5.50 shows the ROM of Figure 5.49 built using a decoder and OR gates. Each dotted row in Figure 5.49 is an input to an OR gate in Figure 5.50. For data bits with a single dot, in this case $Data_0$, no OR gate is needed. This representation of a ROM is interesting because it shows how the ROM can perform any two-level logic function. In practice, ROMs are built from transistors instead of logic gates, to reduce their size and cost. Section 5.6.3 explores the transistor-level implementation further.

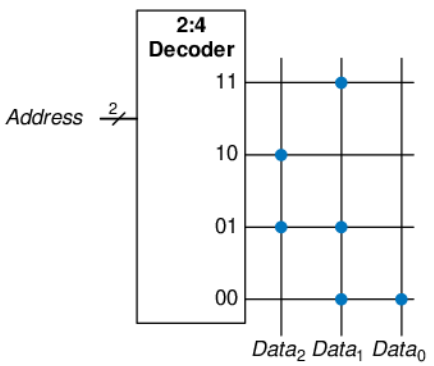


Figure 5.49 4 × 3 ROM: dot notation

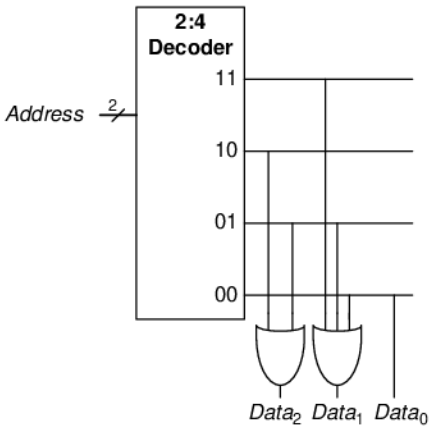


Figure 5.50 4 × 3 ROM implementation using gates

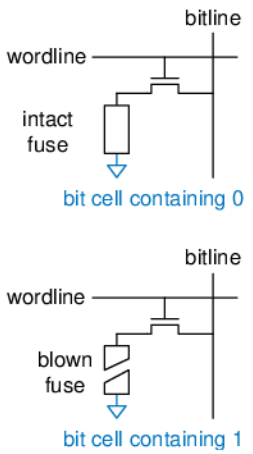


Figure 5.51 Fuse-programmable ROM bit cell

The contents of the ROM bit cell in Figure 5.48 are specified during manufacturing by the presence or absence of a transistor in each bit cell. A *programmable ROM* (*PROM*, pronounced like the dance) places a transistor in every bit cell but provides a way to connect or disconnect the transistor to ground.

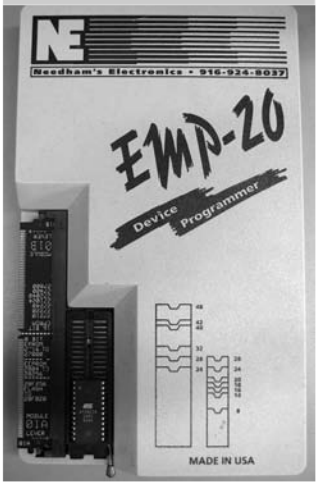
Figure 5.51 shows the bit cell for a *fuse-programmable ROM*. The user programs the ROM by applying a high voltage to selectively blow fuses. If the fuse is present, the transistor is connected to GND and the cell holds a 0. If the fuse is destroyed, the transistor is disconnected from ground and the cell holds a 1. This is also called a one-time programmable ROM, because the fuse cannot be repaired once it is blown.

Reprogrammable ROMs provide a reversible mechanism for connecting or disconnecting the transistor to GND. *Erasable PROMs* (*EPROMs*, pronounced "e-proms") replace the nMOS transistor and fuse with a *floating-gate transistor*. The floating gate is not physically attached to any other wires. When suitable high voltages are applied, electrons tunnel through an insulator onto the floating gate, turning on the transistor and connecting the bitline to the wordline (decoder output). When the EPROM is exposed to intense ultraviolet (UV) light for about half an hour, the electrons are knocked off the floating gate, turning the transistor off. These actions are called *programming* and *erasing*, respectively. *Electrically erasable PROMs* (*EEPROMs*, pronounced "e-e-proms" or "double-e proms") and *Flash* memory use similar principles but include circuitry on the chip for erasing as well as programming, so no UV light is necessary. EEPROM bit cells are individually erasable; Flash memory erases larger blocks of bits and is cheaper because fewer erasing circuits are needed. In 2006, Flash

Fujio Masuoka, 1944–. Received a Ph.D. in electrical engineering from Tohoku University, Japan. Developed memories and high-speed circuits at Toshiba from 1971 to 1994. Invented Flash memory as an unauthorized project pursued during nights and weekends in the late 1970s. Flash received its name because the process of erasing the memory reminds one of the flash of a camera. Toshiba was slow to commercialize the idea; Intel was first to market in 1988. Flash has grown into a $25 billion per year market. Dr. Masuoka later joined the faculty at Tohoku University.

Flash memory drives with Universal Serial Bus (USB) connectors have replaced floppy disks and CDs for sharing files because Flash costs have dropped so dramatically.

Programmable ROMs can be configured with a device programmer like the one shown below. The device programmer is attached to a computer, which specifies the type of ROM and the data values to program. The device programmer blows fuses or injects charge onto a floating gate on the ROM. Thus the programming process is sometimes called *burning* a ROM.



memory costs less than $25 per GB, and the price continues to drop by 30 to 40% per year. Flash has become an extremely popular way to store large amounts of data in portable battery-powered systems such as cameras and music players.

In summary, modern ROMs are not really read only; they can be programmed (written) as well. The difference between RAM and ROM is that ROMs take a longer time to write but are nonvolatile.

5.5.7 Logic Using Memory Arrays

Although they are used primarily for data storage, memory arrays can also perform combinational logic functions. For example, the $Data_2$ output of the ROM in Figure 5.49 is the XOR of the two *Address* inputs. Likewise $Data_0$ is the NAND of the two inputs. A 2^N-word $\times$ M-bit memory can perform any combinational function of N inputs and M outputs. For example, the ROM in Figure 5.49 performs three functions of two inputs.

Memory arrays used to perform logic are called *lookup tables (LUTs)*. Figure 5.52 shows a 4-word $\times$ 1-bit memory array used as a lookup table to perform the function $Y = AB$. Using memory to perform logic, the user can look up the output value for a given input combination (address). Each address corresponds to a row in the truth table, and each data bit corresponds to an output value.

5.5.8 Memory HDL

HDL Example 5.7 describes a 2^N-word $\times$ M-bit RAM. The RAM has a synchronous enabled write. In other words, writes occur on the rising

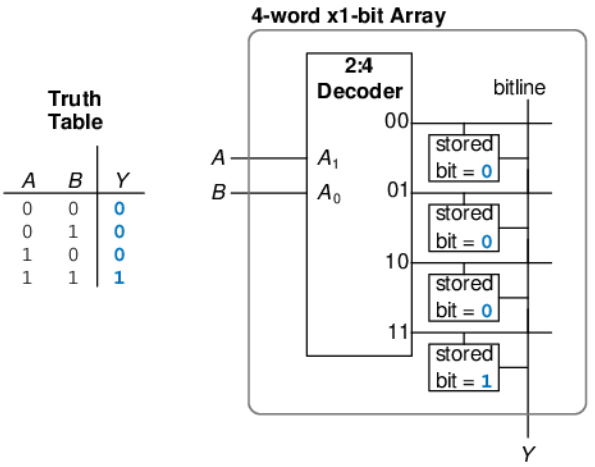


Figure 5.52 4-word $\times$ 1-bit memory array used as a lookup table

HDL Example 5.7 RAM

Verilog

```
module ram # (parameter N = 6, M = 32)
                (input           clk,
                 input           we,
                 input  [N-1:0]  adr,
                 input  [M-1:0]  din,
                 output [M-1:0]  dout);

  reg [M-1:0] mem [2**N-1:0];

  always @ (posedge clk)
    if (we) mem [adr] <= din;

  assign dout = mem[adr];
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity ram_array is
  generic (N: integer := 6; M: integer := 32);
  port (clk,
        we:   in  STD_LOGIC;
        adr:  in  STD_LOGIC_VECTOR(N-1 downto 0);
        din:  in  STD_LOGIC_VECTOR(M-1 downto 0);
        dout: out STD_LOGIC_VECTOR(M-1 downto 0));
end;

architecture synth of ram_array is
  type mem_array is array((2**N-1) downto 0)
       of STD_LOGIC_VECTOR (M-1 downto 0);
  signal mem: mem_array;
begin
  process (clk) begin
    if clk' event and clk = '1' then
      if we = '1' then
        mem (CONV_INTEGER (adr)) <= din;
      end if;
    end if;
  end process;

  dout <= mem (CONV_INTEGER(adr));
end;
```

Figure 5.53 Synthesized ram

edge of the clock if the write enable, *we*, is asserted. Reads occur immediately. When power is first applied, the contents of the RAM are unpredictable.

HDL Example 5.8 describes a 4-word × 3-bit ROM. The contents of the ROM are specified in the HDL `case` statement. A ROM as small as this one may be synthesized into logic gates rather than an array. Note that the seven-segment decoder from HDL Example 4.25 synthesizes into a ROM in Figure 4.22.

HDL Example 5.8 ROM

Verilog

```
module rom (input      [1:0] adr,
            output reg [2:0] dout);

  always @ (adr)
    case (adr)
      2'b00: dout <= 3'b011;
      2'b01: dout <= 3'b110;
      2'b10: dout <= 3'b100;
      2'b11: dout <= 3'b010;
    endcase
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity rom is
  port (adr:  in  STD_LOGIC_VECTOR(1 downto 0);
        dout: out STD_LOGIC_VECTOR(2 downto 0));
end;

architecture synth of rom is
begin
  process (adr) begin
    case adr is
      when "00" => dout <= "011";
      when "01" => dout <= "110";
      when "10" => dout <= "100";
      when "11" => dout <= "010";
    end case;
  end process;
end;
```

5.6 LOGIC ARRAYS

Like memory, gates can be organized into regular arrays. If the connections are made programmable, these *logic arrays* can be configured to perform any function without the user having to connect wires in specific ways. The regular structure simplifies design. Logic arrays are mass produced in large quantities, so they are inexpensive. Software tools allow users to map logic designs onto these arrays. Most logic arrays are also *re*configurable, allowing designs to be modified without replacing the hardware. Reconfigurability is valuable during development and is also useful in the field, because a system can be upgraded by simply downloading the new configuration.

This section introduces two types of logic arrays: *programmable logic arrays* (*PLAs*), and *field programmable gate arrays* (*FPGAs*). PLAs, the older technology, perform only combinational logic functions. FPGAs can perform both combinational and sequential logic.

5.6.1 Programmable Logic Array

Programmable logic arrays (*PLAs*) implement two-level combinational logic in sum-of-products (SOP) form. PLAs are built from an AND array followed by an OR array, as shown in Figure 5.54. The inputs (in true and complementary form) drive an AND array, which produces implicants, which in turn are ORed together to form the outputs. An $M \times N \times P$-bit PLA has M inputs, N implicants, and P outputs.

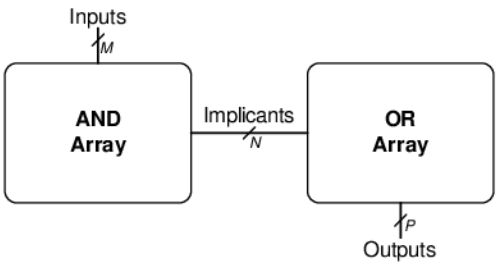


Figure 5.54 ***M*** $\times$ ***N*** $\times$ ***P*-bit PLA**

Figure 5.55 shows the dot notation for a $3 \times 3 \times 2$-bit PLA performing the functions $X = \overline{A}\,\overline{B}C + AB\overline{C}$ and $Y = A\overline{B}$. Each row in the AND array forms an implicant. Dots in each row of the AND array indicate which literals comprise the implicant. The AND array in Figure 5.55 forms three implicants: $\overline{A}\,\overline{B}C$, $AB\overline{C}$, and $A\overline{B}$. Dots in the OR array indicate which implicants are part of the output function.

Figure 5.56 shows how PLAs can be built using two-level logic. An alternative implementation is given in Section 5.6.3.

ROMs can be viewed as a special case of PLAs. A 2^M-word $\times$ N-bit ROM is simply an $M \times 2^M \times N$-bit PLA. The decoder behaves as an AND plane that produces all 2^M minterms. The ROM array behaves as an OR plane that produces the outputs. If the function does not depend on all 2^M minterms, a PLA is likely to be smaller than a ROM. For example, an 8-word $\times$ 2-bit ROM is required to perform the same functions performed by the $3 \times 3 \times 2$-bit PLA shown in Figures 5.55 and 5.56.

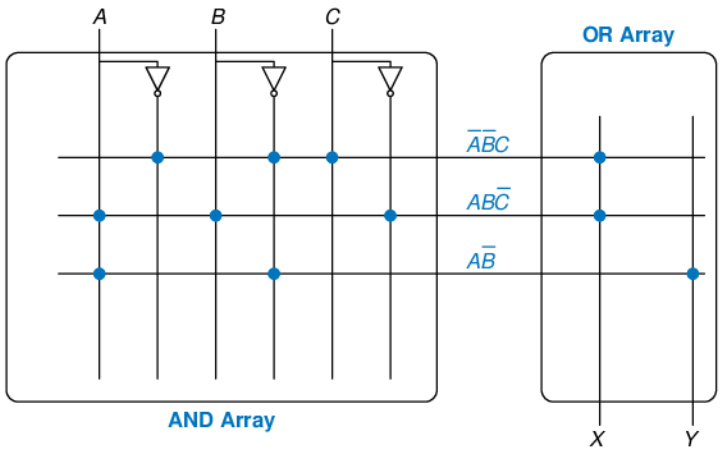


Figure 5.55 **3** $\times$ **3** $\times$ **2-bit PLA: dot notation**

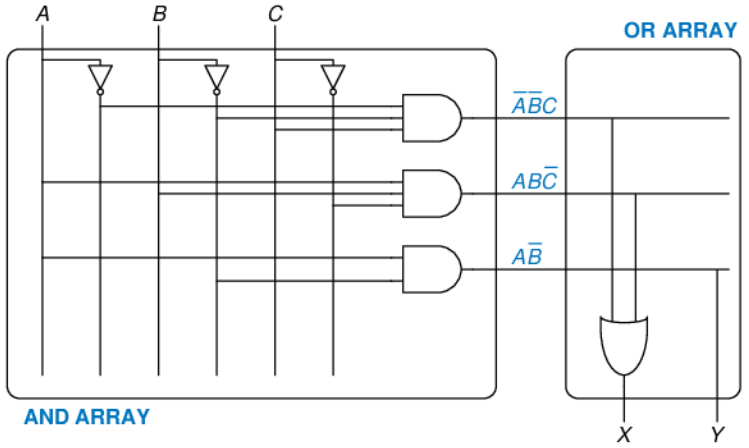


Figure 5.56 $3 \times 3 \times 2$**-bit PLA using two-level logic**

FPGAs are the brains of many consumer products, including automobiles, medical equipment, and media devices like MP3 players. The Mercedes Benz S-Class series, for example, has over a dozen Xilinx FPGAs or PLDs for uses ranging from entertainment to navigation to cruise control systems. FPGAs allow for quick time to market and make debugging or adding features late in the design process easier.

Programmable logic devices (*PLDs*) are souped-up PLAs that add registers and various other features to the basic AND/OR planes. However, PLDs and PLAs have largely been displaced by FPGAs, which are more flexible and efficient for building large systems.

5.6.2 Field Programmable Gate Array

A *field programmable gate array* (*FPGA*) is an array of reconfigurable gates. Using software programming tools, a user can implement designs on the FPGA using either an HDL or a schematic. FPGAs are more powerful and more flexible than PLAs for several reasons. They can implement both combinational and sequential logic. They can also implement multilevel logic functions, whereas PLAs can only implement two-level logic. Modern FPGAs integrate other useful functions such as built-in multipliers and large RAM arrays.

FPGAs are built as an array of *configurable logic blocks (CLBs)*. Figure 5.57 shows the block diagram of the Spartan FPGA introduced by Xilinx in 1998. Each CLB can be configured to perform combinational or sequential functions. The CLBs are surrounded by *input/output blocks (IOBs)* for interfacing with external devices. The IOBs connect CLB inputs and outputs to pins on the chip package. CLBs can connect to other CLBs and IOBs through programmable routing channels. The remaining blocks shown in the figure aid in programming the device.

Figure 5.58 shows a single CLB for the Spartan FPGA. Other brands of FPGAs are organized somewhat differently, but the same general principles apply. The CLB contains lookup tables (LUTs), configurable multiplexers, and registers. The FPGA is configured by specifying the contents of the lookup tables and the select signals for the multiplexers.

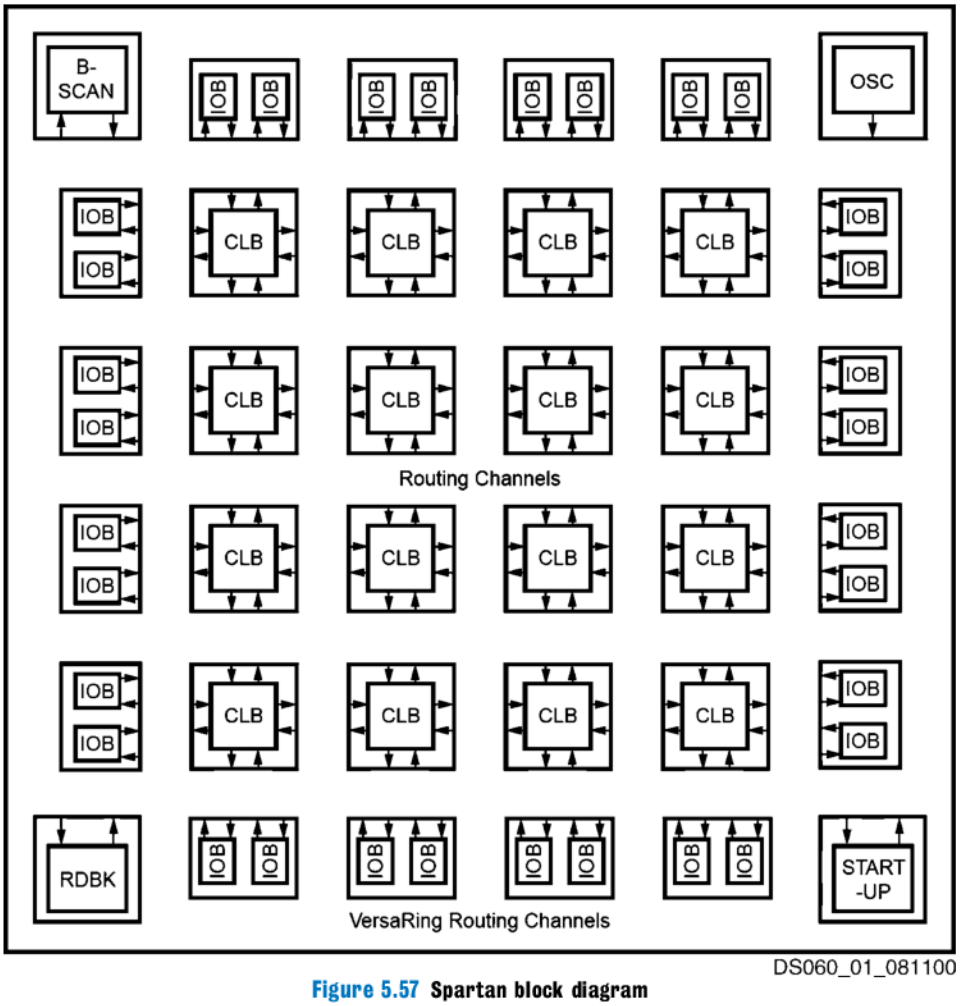


Figure 5.57 Spartan block diagram

Each Spartan CLB has three LUTs: the four-input F- and G-LUTs, and the three-input H-LUT. By loading the appropriate values into the lookup tables, the F- and G-LUTs can each be configured to perform any function of up to four variables, and the H-LUT can perform any function of up to three variables.

Configuring the FPGA also involves choosing the select signals that determine how the multiplexers route data through the CLB. For example, depending on the multiplexer configuration, the H-LUT may receive one of its inputs from either DIN or the F-LUT. Similarly, it receives another input from either SR or the G-LUT. The third input always comes from H1.

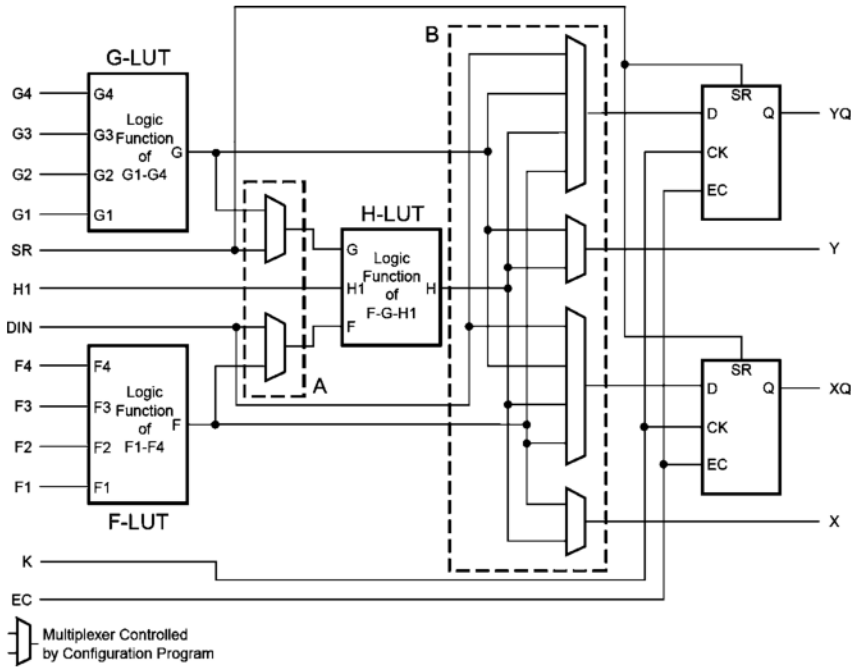


Figure 5.58 Spartan CLB

The FPGA produces two combinational outputs, *X* and *Y*. Depending on the multiplexer configuration, *X* comes from either the F- or H-LUT. *Y* comes from either the G- or H-LUT. These outputs can be connected to other CLBs via the routing channels

The CLB also contains two flip-flops. Depending on the configuration, the flip-flop inputs may come from DIN or from the F-, G-, or H-LUT. The flip-flop outputs, *XQ* and *YQ*, also can be connected to other CLBs via the routing channels.

In summary, the CLB can perform up to two combinational and/or two registered functions. All of the functions can involve at least four variables, and some can involve up to nine.

The designer configures an FPGA by first creating a schematic or HDL description of the design. The design is then synthesized onto the FPGA. The synthesis tool determines how the LUTs, multiplexers, and routing channels should be configured to perform the specified functions. This configuration information is then downloaded to the FPGA.

Because Xilinx FPGAs store their configuration information in SRAM, they can be easily reprogrammed. They may download the SRAM contents from a computer in the laboratory or from an EEPROM chip when the

system is turned on. Some manufacturers include EEPROM directly on the FPGA or use one-time programmable fuses to configure the FPGA.

Example 5.6 FUNCTIONS BUILT USING CLBS

Explain how to configure a CLB to perform the following functions: (a) $X = \overline{A}\,\overline{B}C + AB\overline{C}$ and $Y = A\overline{B}$; (b) $Y = JKLMPQR$; (c) a divide-by-3 counter with binary state encoding (see Figure 3.29(a)).

Solution: (a) Configure the F-LUT to compute X and the G-LUT to compute Y, as shown in Figure 5.59. Inputs $F3$, $F2$, and $F1$ are A, B, and C, respectively (these connections are set by the routing channels). Inputs $G2$ and $G1$ are A and B. $F4$, $G4$, and $G3$ are don't cares (and may be connected to 0). Configure the final multiplexers to select X from the F-LUT and Y from the G-LUT. In general, a CLB can compute any two functions, of up to four variables each, in this fashion.

(b) Configure the F-LUT to compute $F = JKLM$ and the G-LUT to compute $G = PQR$. Then configure the H-LUT to compute $H = FG$. Configure the final multiplexer to select Y from the H-LUT. This configuration is shown in Figure 5.60. In general, a CLB can compute certain functions of up to nine variables in this way.

(c) The FSM has two bits of state ($S_{1:0}$) and one output (Y). The next state depends on the two bits of current state. Use the F-LUT and G-LUT to compute the next state from the current state, as shown in Figure 5.61. Use the two flip-flops to hold this state. The flip-flops have a dedicated reset input from the SR signal in the CLB. The registered outputs are fed back to the inputs using the routing channels, as indicated by the dashed blue lines. In general, another CLB might be necessary to compute the output Y. However, in this case, $Y = S_0'$, so Y can come from the same F-LUT used to compute S_0'. Hence, the entire FSM fits in a single CLB. In general, an FSM requires at least one CLB for every two bits of state, and it may require more CLBs for the output or next state logic if they are too complex to fit in a single LUT.

	(A)	(B)	(C)	(X)
F4	F3	F2	F1	F
X	0	0	0	0
X	0	0	1	1
X	0	1	0	0
X	0	1	1	0
X	1	0	0	0
X	1	0	1	0
X	1	1	0	1
X	1	1	1	0

		(A)	(B)	(Y)
G4	G3	G2	G1	G
X	X	0	0	0
X	X	0	1	0
X	X	1	0	1
X	X	1	1	0

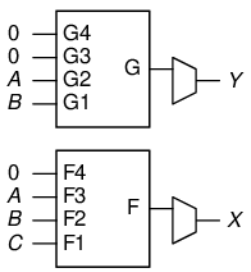


Figure 5.59 **CLB configuration for two functions of up to four inputs each**

(J) F4	(K) F3	(L) F2	(M) F1	F
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

(P) G4	(Q) G3	(R) G2	G1	G
X	0	0	0	0
X	0	0	1	0
X	0	1	0	0
X	0	1	1	0
X	1	0	0	0
X	1	0	1	0
X	1	1	0	0
X	1	1	1	1

H1	F	G	(Y) H
X	0	0	0
X	0	1	0
X	1	0	0
X	1	1	1

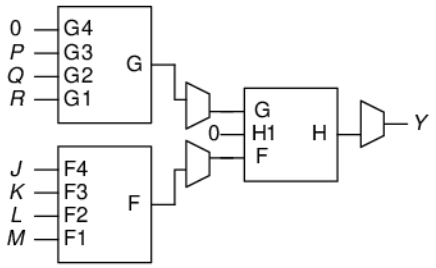


Figure 5.60 CLB configuration for one function of more than four inputs

F4	F3	(S_1) F2	(S_0) F1	(S_0') F
X	X	0	0	1
X	X	0	1	0
X	X	1	0	0
X	X	1	1	0

G4	G3	(S_1) G2	(S_0) G1	(S_1') G
X	X	0	0	0
X	X	0	1	1
X	X	1	0	0
X	X	1	1	0

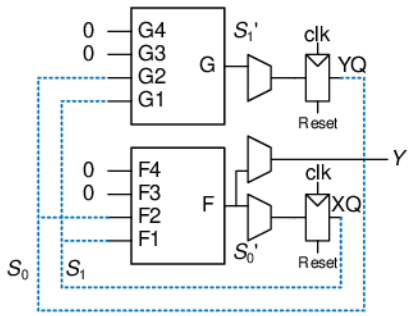


Figure 5.61 CLB configuration for FSM with two bits of state

Example 5.7 CLB DELAY

Alyssa P. Hacker is building a finite state machine that must run at 200 MHz. She uses a Spartan 3 FPGA with the following specifications: t_{CLB} = 0.61 ns per CLB; t_{setup} = 0.53 ns and t_{pcq} = 0.72 ns for all flip-flops. What is the maximum number of CLBs her design can use? You can ignore interconnect delay.

Solution: Alyssa uses Equation 3.13 to solve for the maximum propagation delay of the logic: $t_{pd} \leq T_c - (t_{pcq} + t_{\text{setup}})$.

Thus, $t_{pd} \leq 5$ ns $-$ (0.72 ns + 0.53 ns), so $t_{pd} \leq 3.75$ ns. The delay of each CLB, t_{CLB}, is 0.61 ns, and the maximum number of CLBs, N, is $Nt_{CLB} \leq 3.75$ ns. Thus, $N = 6$.

5.6.3 Array Implementations*

Many ROMs and PLAs use dynamic circuits in place of pseudo-nMOS circuits. Dynamic gates turn the pMOS transistor ON for only part of the time, saving power when the pMOS is OFF and the result is not needed. Aside from this, dynamic and pseudo-nMOS memory arrays are similar in design and behavior.

To minimize their size and cost, ROMs and PLAs commonly use pseudo-nMOS or dynamic circuits (see Section 1.7.8) instead of conventional logic gates.

Figure 5.62(a) shows the dot notation for a 4 × 3-bit ROM that performs the following functions: $X = A \oplus B$, $Y = \overline{A} + B$, and $Z = \overline{AB}$. These are the same functions as those of Figure 5.49, with the address inputs renamed A and B and the data outputs renamed X, Y, and Z. The pseudo-nMOS implementation is given in Figure 5.62(b). Each decoder output is connected to the gates of the nMOS transistors in its row. Remember that in pseudo-nMOS circuits, the weak pMOS transistor pulls the output HIGH *only if* there is no path to GND through the pull-down (nMOS) network.

Pull-down transistors are placed at every junction without a dot. The dots from the dot notation diagram of Figure 5.62(a) are left faintly visible in Figure 5.62(b) for easy comparison. The weak pull-up transistors pull the output HIGH for each wordline without a pull-down transistor. For example, when $AB = 11$, the 11 wordline is HIGH and transistors on X and Z turn on and pull those outputs LOW. The Y output has no transistor connecting to the 11 wordline, so Y is pulled HIGH by the weak pull-up.

PLAs can also be built using pseudo-nMOS circuits, as shown in Figure 5.63 for the PLA from Figure 5.55. Pull-down (nMOS) transistors are placed on the *complement* of dotted literals in the AND array and on dotted rows in the OR array. The columns in the OR array are sent through an inverter before they are fed to the output bits. Again, the blue dots from the dot notation diagram of Figure 5.55 are left faintly visible in Figure 5.63 for easy comparison.

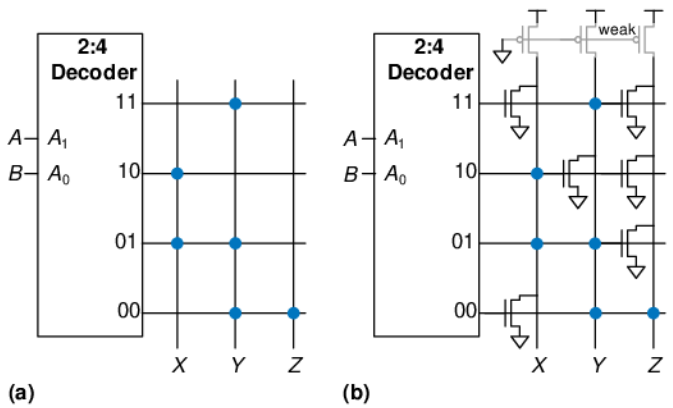


Figure 5.62 ROM implementation: (a) dot notation, (b) pseudo-nMOS circuit

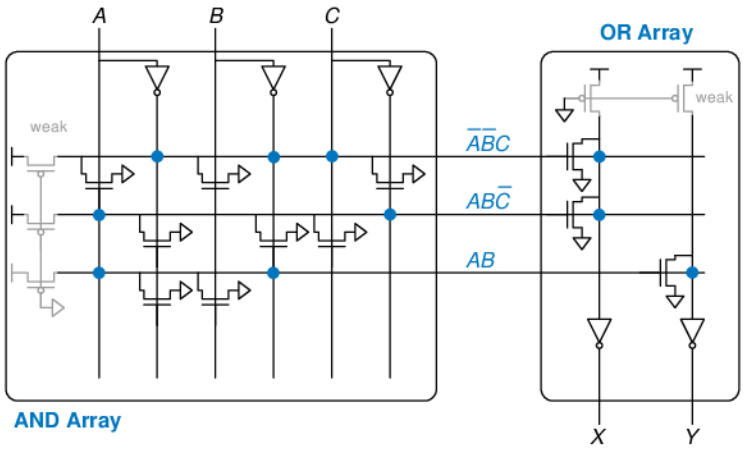


Figure 5.63 $3 \times 3 \times 2$**-bit PLA using pseudo-nMOS circuits**

5.7 SUMMARY

This chapter introduced digital building blocks used in many digital systems. These blocks include arithmetic circuits such as adders, subtractors, comparators, shifters, multipliers, and dividers; sequential circuits such as counters and shift registers; and arrays for memory and logic. The chapter also explored fixed-point and floating-point representations of fractional numbers. In Chapter 7, we use these building blocks to build a microprocessor.

Adders form the basis of most arithmetic circuits. A half adder adds two 1-bit inputs, A and B, and produces a sum and a carry out. A full adder extends the half adder to also accept a carry in. N full adders can be cascaded to form a carry propagate adder (CPA) that adds two N-bit numbers. This type of CPA is called a ripple-carry adder because the carry ripples through each of the full adders. Faster CPAs can be constructed using lookahead or prefix techniques.

A subtractor negates the second input and adds it to the first. A magnitude comparator subtracts one number from another and determines the relative value based on the sign of the result. A multiplier forms partial products using AND gates, then sums these bits using full adders. A divider repeatedly subtracts the divisor from the partial remainder and checks the sign of the difference to determine the quotient bits. A counter uses an adder and a register to increment a running count.

Fractional numbers are represented using fixed-point or floating-point forms. Fixed-point numbers are analogous to decimals, and floating-point numbers are analogous to scientific notation. Fixed-point numbers use ordinary arithmetic circuits, whereas floating-point numbers require more elaborate hardware to extract and process the sign, exponent, and mantissa.

Large memories are organized into arrays of words. The memories have one or more ports to read and/or write the words. Volatile memories, such as SRAM and DRAM, lose their state when the power is turned off. SRAM is faster than DRAM but requires more transistors. A register file is a small multiported SRAM array. Nonvolatile memories, called ROMs, retain their state indefinitely. Despite their names, most modern ROMs can be written.

Arrays are also a regular way to build logic. Memory arrays can be used as lookup tables to perform combinational functions. PLAs are composed of dedicated connections between configurable AND and OR arrays; they only implement combinational logic. FPGAs are composed of many small lookup tables and registers; they implement combinational and sequential logic. The lookup table contents and their interconnections can be configured to perform any logic function. Modern FPGAs are easy to reprogram and are large and cheap enough to build highly sophisticated digital systems, so they are widely used in low- and medium-volume commercial products as well as in education.

EXERCISES

Exercise 5.1 What is the delay for the following types of 64-bit adders? Assume that each two-input gate delay is 150 ps and that a full adder delay is 450 ps.

(a) a ripple-carry adder

(b) a carry-lookahead adder with 4-bit blocks

(c) a prefix adder

Exercise 5.2 Design two adders: a 64-bit ripple-carry adder and a 64-bit carry-lookahead adder with 4-bit blocks. Use only two-input gates. Each two-input gate is 15 μm^2, has a 50 ps delay, and has 20 pF of total gate capacitance. You may assume that the static power is negligible.

(a) Compare the area, delay, and power of the adders (operating at 100 MHz).

(b) Discuss the trade-offs between power, area, and delay.

Exercise 5.3 Explain why a designer might choose to use a ripple-carry adder instead of a carry-lookahead adder.

Exercise 5.4 Design the 16-bit prefix adder of Figure 5.7 in an HDL. Simulate and test your module to prove that it functions correctly.

Exercise 5.5 The prefix network shown in Figure 5.7 uses black cells to compute all of the prefixes. Some of the block propagate signals are not actually necessary. Design a "gray cell" that receives G and P signals for bits $i{:}k$ and $k-1{:}j$ but produces only $G_{i:j}$, not $P_{i:j}$. Redraw the prefix network, replacing black cells with gray cells wherever possible.

Exercise 5.6 The prefix network shown in Figure 5.7 is not the only way to calculate all of the prefixes in logarithmic time. The *Kogge-Stone* network is another common prefix network that performs the same function using a different connection of black cells. Research Kogge-Stone adders and draw a schematic similar to Figure 5.7 showing the connection of black cells in a Kogge-Stone adder.

Exercise 5.7 Recall that an N-input priority encoder has $\log_2 N$ outputs that encodes which of the N inputs gets priority (see Exercise 2.25).

(a) Design an N-input priority encoder that has delay that increases logarithmically with N. Sketch your design and give the delay of the circuit in terms of the delay of its circuit elements.

(b) Code your design in HDL. Simulate and test your module to prove that it functions correctly.

Exercise 5.8 Design the following comparators for 32-bit numbers. Sketch the schematics.

(a) not equal

(b) greater than

(c) less than or equal to

Exercise 5.9 Design the 32-bit ALU shown in Figure 5.15 using your favorite HDL. You can make the top-level module either behavioral or structural.

Exercise 5.10 Add an *Overflow* output to the 32-bit ALU from Exercise 5.9. The output is TRUE when the result of the adder overflows. Otherwise, it is FALSE.

(a) Write a Boolean equation for the *Overflow* output.

(b) Sketch the Overflow circuit.

(c) Design the modified ALU in an HDL.

Exercise 5.11 Add a *Zero* output to the 32-bit ALU from Exercise 5.9. The output is TRUE when $Y == 0$.

Exercise 5.12 Write a testbench to test the 32-bit ALU from Exercise 5.9, 5.10, or 5.11. Then use it to test the ALU. Include any test vector files necessary. Be sure to test enough corner cases to convince a reasonable skeptic that the ALU functions correctly.

Exercise 5.13 Design a shifter that always shifts a 32-bit input left by 2 bits. The input and output are both 32 bits. Explain the design in words and sketch a schematic. Implement your design in your favourite HDL.

Exercise 5.14 Design 4-bit left and right rotators. Sketch a schematic of your design. Implement your design in your favourite HDL.

Exercise 5.15 Design an 8-bit left shifter using only 24 2:1 multiplexers. The shifter accepts an 8-bit input, A, and a 3-bit shift amount, $shamt_{2:0}$. It produces an 8-bit output, Y. Sketch the schematic.

Exercise 5.16 Explain how to build any N-bit shifter or rotator using only $N\log_2 N$ 2:1 multiplexers.

Exercise 5.17 The *funnel shifter* in Figure 5.64 can perform any N-bit shift or rotate operation. It shifts a $2N$-bit input right by k bits. The output, Y, is the N least significant bits of the result. The most significant N bits of the input are