5 Digital Building Blocks

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

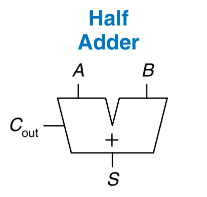
Arithmetic circuits are the central building blocks of computers. Computers and digital logic perform many arithmetic functions: addition, subtraction, comparisons, shifts, multiplication, and division. This section describes hardware implementations for all of these operations.

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.

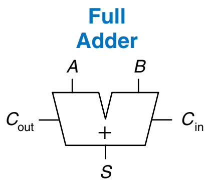
In a multi-bit adder, C_{out} is added or *carried in* to the next most significant bit. For example, in Figure 5.2, the carry bit shown in blue is the output, C_{out}, of the first column of 1-bit addition and the input, C_{in}, to the second column of addition. However, the half adder lacks a C_{in}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

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}.



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

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.

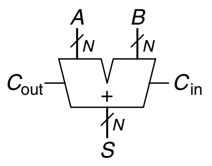


Figure 5.4 Carry propagate adder

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).

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.

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

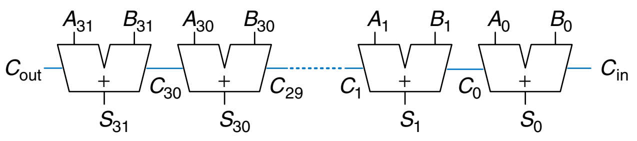


Figure 5.5 32-bit ripple-carry adder

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 *i*th column of an adder is said to *generate* a carry if it produces a carry out independent of the carry in. The *i*th 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 $G_i = A_iB_i$. The column is said to *propagate* a carry if it produces a carry out whenever there is a carry in. The *i*th 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 *i*th 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,

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

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?

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.

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

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

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

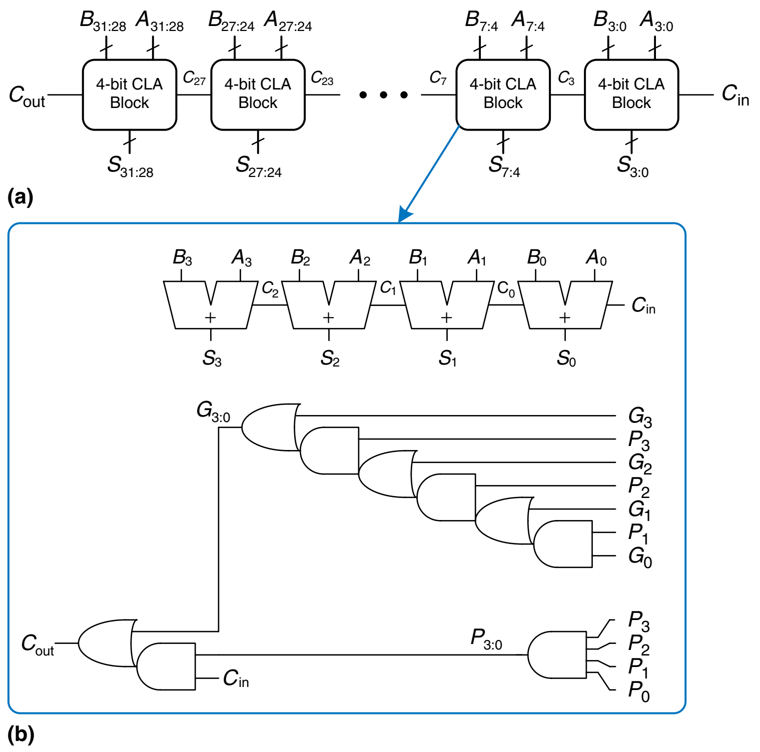


Figure 5.6 (a) 32-bit ***carry-lookahead adder*** (***CLA***), (b) 4-bit CLA block

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 $t_{\text{AND\_OR}}$ 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 $t_{\text{AND\_OR}} = 2 \times 100$ 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 $G_{-1} = C_{\text{in}}$ 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 $G_{-1:-1}, G_{0:-1}, G_{1:-1}, G_{2:-1}, \ldots, G_{N-2:-1}$. These signals, along with $P_{-1:-1}, P_{0:-1}, P_{1:-1}, P_{2:-1}, \ldots, P_{N-2:-1}$, 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

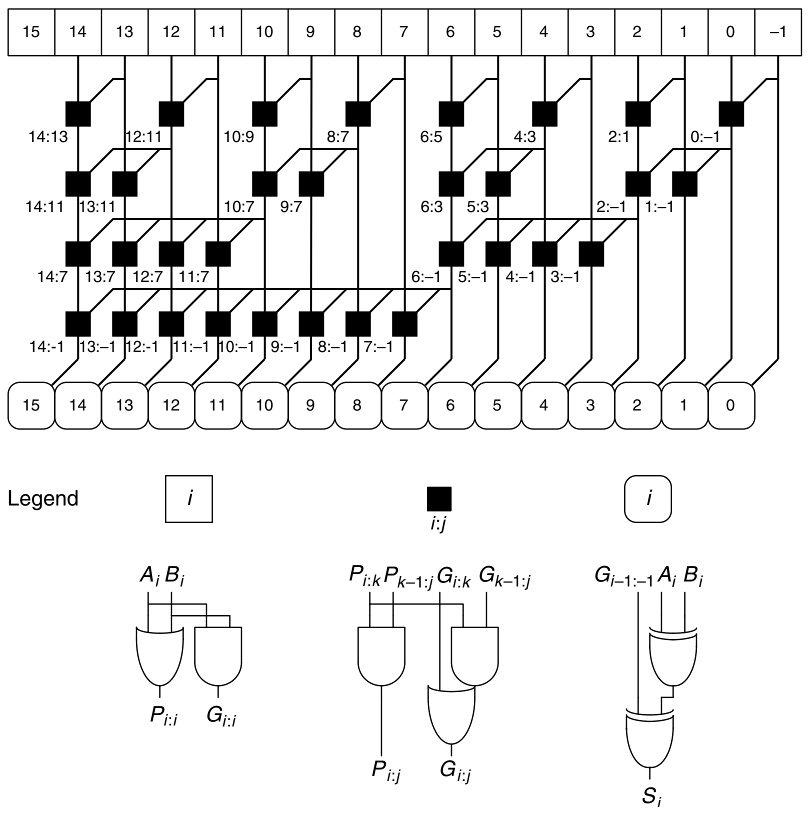


Figure 5.7 16-bit prefix adder

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 ps + $\log_2(32) \times 200$ ps + 100 ps = 1.2 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;
```

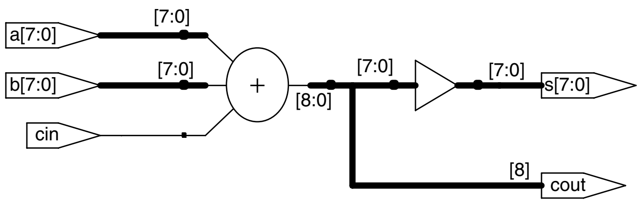


Figure 5.8 Synthesized adder

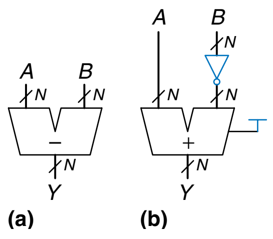


Figure 5.9 Subtractor: (a) symbol, (b) implementation

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.

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;
```

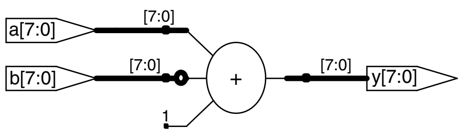


Figure 5.10 Synthesized subtractor

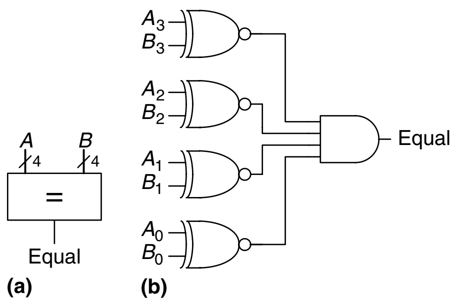


Figure 5.11 4-bit equality comparator: (a) symbol, (b) implementation

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.

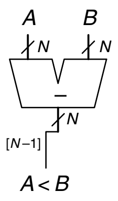


Figure 5.12 *N*-bit magnitude comparator

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;
```

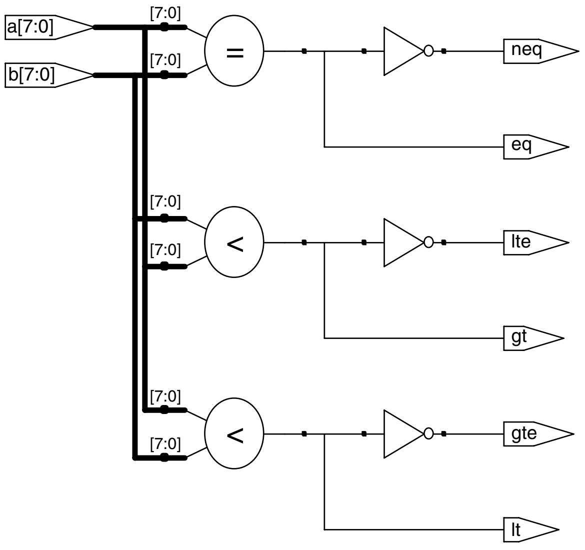


Figure 5.13 **Synthesized comparators**

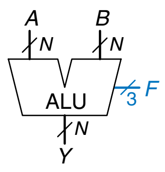


Figure 5.14 **ALU symbol**

5.2.4 ALU

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 $\overline{\text{B}}$
101	A OR $\overline{\text{B}}$
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.

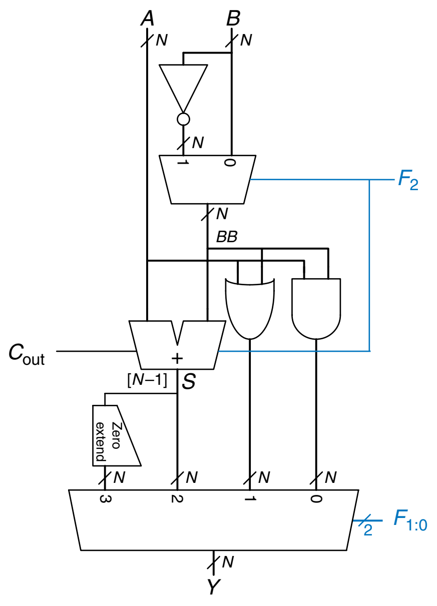


Figure 5.15 ***N*-bit ALU**

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 $N-1^{th}$ 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 $3_{10} \times 2^4 = 48_{10}$.

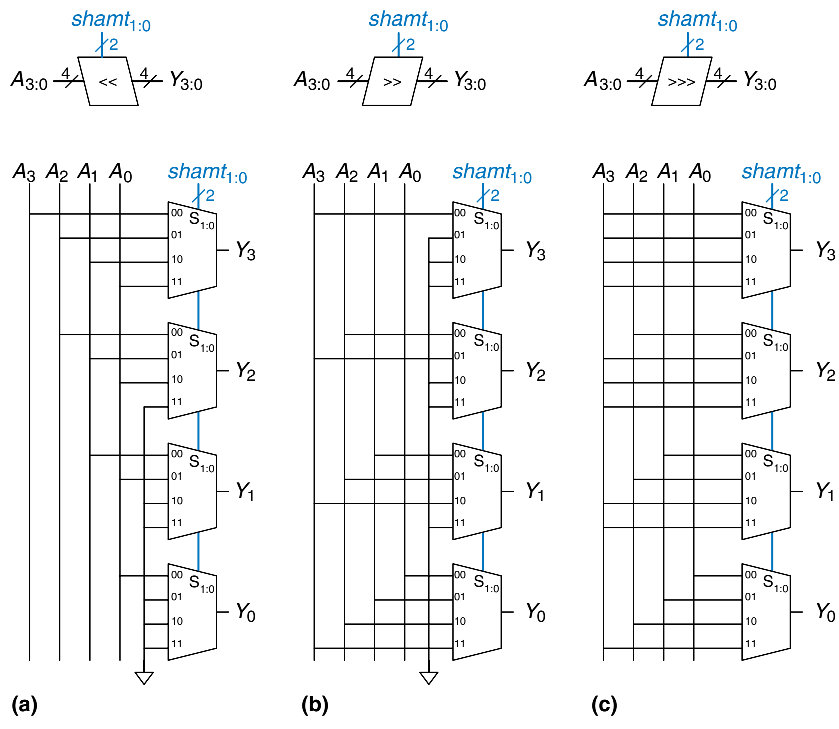


Figure 5.16 **4-bit shifters: (a) shift left, (b) logical shift right, (c) arithmetic shift right**

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

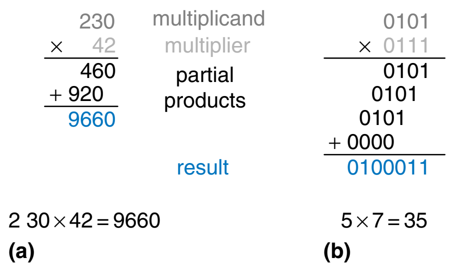


Figure 5.17 **Multiplication: (a) decimal, (b) binary**

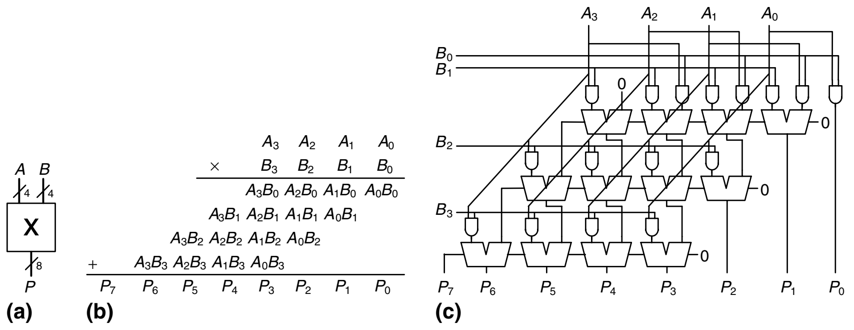


Figure 5.18 **4 × 4 multiplier: (a) symbol, (b) function, (c) implementation**

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;
```

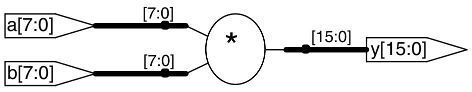


Figure 5.19 **Synthesized multiplier**

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,

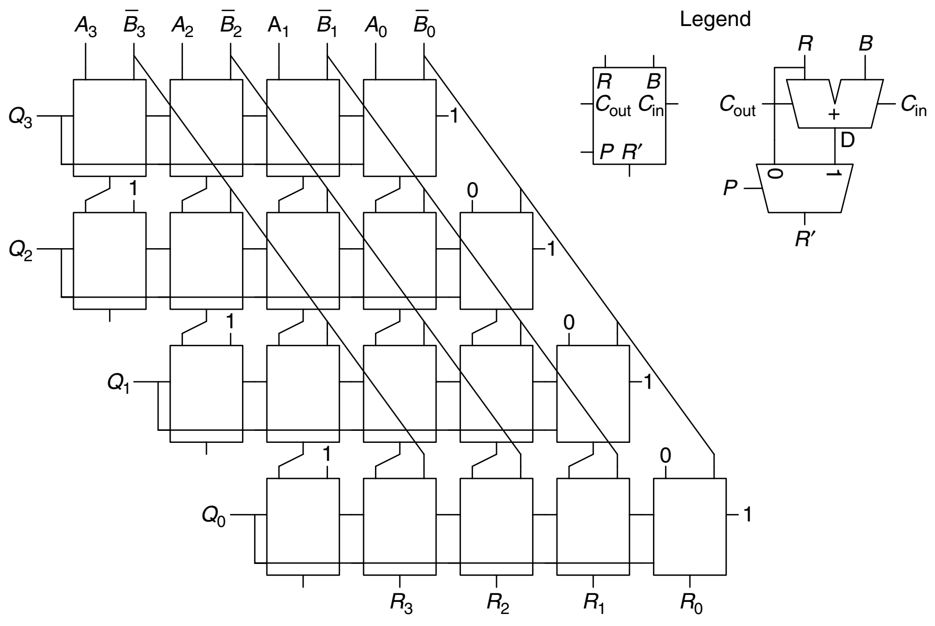

A_3 $\overline{B}_3$ A_2 $\overline{B}_2$ A_1 $\overline{B}_1$ A_0 $\overline{B}_0$
Q_3
Q_2
Q_1
Q_0
1
0
R_3 R_2 R_1 R_0
Legend
R B
C_{out} C_{in}
P R'
+
D
0 1
P
R'

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.

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

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.
