TABLE 3-11: Examples of FPGAs with Dedicated Multipliers

FPGA	Dedicated Multipliers
Xilinx Virtex-4, Virtex-II Pro/X, Spartan-3E, Spartan 3/3L	18 × 18 multipliers
Xilinx Virtex-5 and Virtex-6	25 × 18 multipliers
Altera Stratix II, III, IV	18 × 18 multipliers
Cyclone II, III, IV	

dedicated multipliers (See Table 3-11). Thus, instead of mapping a multiplier into several logic blocks, dedicated multipliers provided on the FPGA fabric can be used. These dedicated multipliers are more efficient than a multiplier one could implement using the programmable logic in the FPGA. As indicated in Table 3-11, many Xilinx and Altera FPGAs provide 18-bit × 18-bit multipliers. Xilinx Virtex-5 and Virtex-6 series contain 25 × 18 multipliers, which are touted to be highly useful for digital signal processing (DSP) applications.

Digital Signal Processing (DSP) Blocks

Multiplication is a common operation in DSP. Hence the dedicated multipliers help DSP applications. Xilinx Virtex 5 FPGAs provide DSP slices which contain 25 × 18 multipliers, 48-bit addition, and so forth. Most FPGAs contain carry chains to facilitate addition. The Altera Stratix IV FPGAs contain built-in adders in each logic module. As with adders and multipliers, an FPGA vendor can provide DSP building blocks such as hardware for fast Fourier transforms (FFTs), finite impulse response (FIR) filters, infinite impulse response (IIR) filters, and so forth. Encryption/decryption, compression/decompression, and security functions can also be provided. Once a large amount of specialized components have been provided, a large part of an FPGA may be unused in applications that do not warrant such specialized components. In some FPGAs, DSP support is limited to the dedicated multipliers.

TABLE 3-12: Examples of FPGAs with Embedded Microprocessors

FPGA	Embedded Processor
Xilinx Virtex-4, Virtex-II Pro/X	IBM PowerPC
Xilinx Spartan-3E, Spartan 3/3l	MicroBlaze PicoBlaze
Xilinx Kintex	MicroBlaze
Altera Stratix II, Cyclone II	Nios II
Altera APEX, APEX II	ARM, MIPS, Nios
Altera Excalibur	ARM 9
Altera Arria	ARM 9
Microsemi Fusion	ARM7

Embedded Processors

Many modern FPGAs contain an entire processor core. This is extremely useful when designers use hybrid solutions, where part of a system is in a programmable

processor but part of the system is implemented in hardware. Circuitry that needs a large amount of flexibility can be implemented in the microprocessor, but circuit parts that need better performance than that of a programmable processor can be implemented in the FPGA logic blocks. Some FPGAs include the core of a small MIPS processor such as the MIPS R 4000, and some include an embedded version of the IBM PowerPC processor. Some FPGAs include custom processors designed by the FPGA vendors such as the MicroBlaze from Xilinx and the Nios processor from Altera.

Content Addressable Memories (CAM)

In some FPGAs, the memory blocks can be used as **content addressable memories** (CAMs). The general concept of a memory is that the user provides a memory address and the memory unit responds with the content. A CAM is a special kind of memory in which the content, not the address, is used to search the memory. A user provides a data element and the CAM responds with addresses where that data was found. CAMs contain more logic than RAMs because all locations of the memory have to be searched simultaneously to see whether the particular content is in any of the locations. Some FPGAs allow embedded CAM (e.g., Altera APEX II).

The **Microsemi Fusion** architecture shown in Figure 3-40 provides several specialized components, including embedded RAM, decryption, and A/D converters. At the core of the chip are tiles of logic blocks (VersaTiles in Microsemi terminology). The embedded RAM is in the form of rows of SRAM blocks above and below the tiles of logic blocks. Several specialized components appear below the SRAM blocks in the bottom. There is a dedicated decryption unit that implements the AES decryption algorithm. (AES stands for advanced encryption standard and has been the cryptographic standard for the U.S. government since 2001.) There is an analog-to-digital converter (ADC) that accepts inputs from several analog quads, which are circuitry to condition analog signals received by the FPGA. The analog quads contain circuitry to monitor and condition signals according to voltage, current, and temperature.

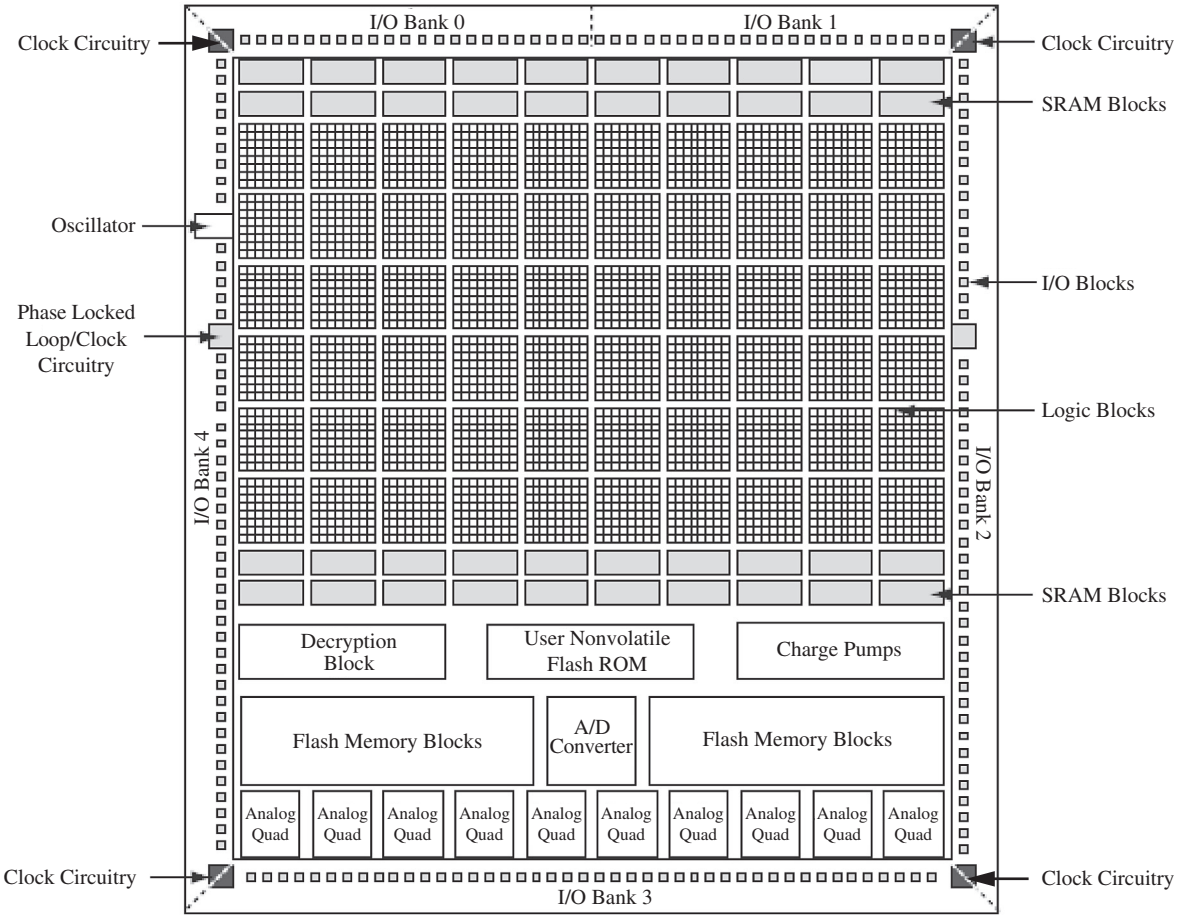
3.4.7 Applications of FPGAs

FPGAs have become a popular mode of circuit implementation for various applications.

Rapid Prototyping

FPGAs are very useful for building rapid prototypes of large systems. A designer can build proof-of-concept systems very quickly using field-programmable gate arrays. Since FPGAs are large enough to contain 5 million or more gates, many large real-world systems can be prototyped using a single FPGA. If a single FPGA will not suffice, multiple FPGAs can be interconnected to realize large systems.

FIGURE 3-40: Overview of the Microsemi Fusion Chip. Based on Xilinx.



Rapid prototyping of large systems is done by using boards with multiple FPGAs and plugging multiple boards into a backplane (motherboard).

FPGAs As Final Product in Medium-Speed Systems

Circuits realized using FPGAs typically operate in the 150–200 MHz clock rate. For applications where this speed is sufficient, FPGAs can be used for the final product itself as opposed to the prototype. When an FPGA is used as the final product, enhancements to the system can be done as software updates rather than as hardware changes. Modern FPGA speeds are adequate for many applications.

Reconfigurable Circuits and Systems

The reprogrammability of FPGAs lends itself to building dynamically reconfigurable circuits and systems. SRAM-based FPGAs make it possible to implement

"soft" hardware. FPGAs have been used to design circuits and systems that need multiple functionalities at various times.

As an example, consider a reprogrammable Tomahawk missile that the U.S. Navy designed using FPGAs. [TomaHawk reference]. The conventional Tomahawk is a long-range Navy cruise missile designed to perform a variety of missions. The Navy designed a reconfigurable Tomahawk, which can operate in one of two modes, depending on the mission at hand. Rather than designing separate logic for each mode, the missile designers used FPGAs so that the configuration for each mode can be kept on-board in ROM. Depending on the mode of operation, the FPGA could be configured in mid-flight.

Glue Logic

FPGAs have become the medium of choice for implementing interface or glue logic between modules and components. Small changes in interface protocols or formats would conventionally necessitate building new interface logic. With SRAM FPGAs, the new interface logic can be implemented on the same FPGA as in a software update.

Hardware Accelerators/Coprocessors

A software application running on a conventional system can be accelerated if a coprocessor/accelerator can implement some key routines/kernels from the application in hardware. An FPGA can be used to implement the key kernel. An SRAM-based, reconfigurable FPGA is well suited for this use, because depending on the application running, different kernels can dynamically be programmed into the FPGA. This approach has been demonstrated for applications such as pattern matching. FPGA-based hardware is used for several applications, including computer architecture simulator acceleration, emulation boards, hardware test/verification, among others.

3.4.8 Design Flow for FPGAs

Sophisticated CAD tools are available to assist with the design of systems using programmable gate arrays. Designs can be entered in many ways.

In the early days of FPGAs, designs were entered using schematic entry or even lower levels of design entry tools. Low-level design entry means less abstraction, whereas high-level means entering designs at a higher level of abstraction (e.g., behavioral Verilog/VHDL description). Early FPGA tools allowed low-level utilities to enter logic equations, Karnaugh maps, and other objects into specific logic blocks in the FPGA. Schematic capture technique means that the designer develops a schematic of the design. Schematic diagrams utilizing standard hardware components are created and entered into the CAD software.

Currently, automatic synthesis tools are available that will take a Verilog description of the system as an input and generate an interconnection of gates and flip-flops to realize the system. Behavioral models can be translated into design

implementations reasonably efficiently. Synthesis tools have advanced significantly in the last decade. The most common method of designing a digital system with an FPGA uses the following steps:

1. Create a behavioral, RTL, or structural model of the design in a hardware description language such as Verilog or VHDL.
2. Simulate and debug the design.
3. Synthesize the design targeting the desired device.
4. Run a mapping/partitioning program. This program will break the logic diagram into pieces that will fit into the configurable logic blocks.
5. Run an automatic place-and-route program. This will place the logic blocks in appropriate places in the FPGA and will then route the interconnections between the logic blocks.
6. Run a program that will generate the bit pattern necessary to program the FPGA.
7. Download the bit pattern into the internal configuration cells in the FPGA and test the operation of the FPGA.

Steps 3, 4, and 5 are often integrated in modern CAD tools. However, the processes mentioned in the steps are happening whether presented as one step or several steps. This is analogous to how general-purpose compilers have integrated compiling and assembling steps. In the early days of high-level language compilers, the term compiling meant only translation into an assembly language format. Converting from assembly language to machine language code was considered the assembler's job. Currently, the steps are integrated in most high-level language compilation environments.

In SRAM-based FPGAs, when the final system is built, the bit pattern for programming the FPGA is normally stored in an EPROM and automatically loaded into the FPGA when the power is turned on. The EPROM is connected to the FPGA, as shown in Figure 3-41. The FPGA resets itself after the power has been applied. Then it reads the configuration data from the EPROM by supplying a sequence of addresses to the EPROM inputs and storing the EPROM output data in the FPGA internal configuration memory cells. This is not required in flash memory based FPGAs because the flash technology is non-volatile. In antifuse FPGAs, the configuration bits permanently alter the switches.

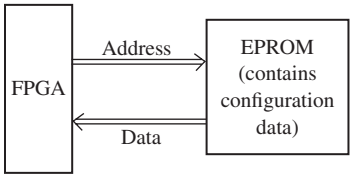


FIGURE 3-41: EPROM Connections for SRAM FPGA Initialization

In this chapter we introduced several different types of programmable logic devices and used them for designing circuits. The technology underlying early

programmable logic devices, such as ROMs, PALs and PLAs, was presented first. Simple PLDs and GALs were presented next. Examples were presented to illustrate implementations of simple logic functions in these devices. CPLDs and FPGAs were presented next. The discussion of FPGAs was limited to an overview of the general technology underlying this class of devices. General organization of FPGAs, general structure of logic blocks, typical programming techniques, and do forth were discussed. More details on FPGAs are presented in Chapter 6.

Problems

3.1 What is the size of the smallest ROM that is needed to implement the following?
- **(a)** An 8-bit full adder (assume carry-in and carry-out)
- **(b)** A BCD-to-binary converter (2 BCD digits)
- **(c)** A 4-to-1 MUX
- **(d)** A 32-bit adder (adds two 32-bit numbers to give a 33-bit sum)
- **(e)** A 3-to-8 decoder
- **(f)** A 32-bit adder (no carry in or carry out)
- **(g)** A 16 × 16 bit multiplier
- **(h)** A 16-bit full adder (with carry-in and carry-out)
- **(i)** An 8-to-3 priority encoder
- **(j)** A 10-to-4 priority encoder
- **(k)** An 8-to-1 multiplexer

3.2 Given $F = A'B' + BC'$ and $G = AC + B'$, write a complete Verilog module that realizes the functions F and G using an 8-word × 2-bit ROM. Include the array type declaration and the constant declaration that defines the contents of the ROM.

3.3 Implement the following state table using a ROM and two D flip-flops. Use a straight binary state assignment.
- **(a)** Show the block diagram and the ROM truth table. Truth table column headings should be in the order $Q_1\ Q_0\ X\ D_1\ D_0\ Z$
- **(b)** Write Verilog code for the implementation. Use an array to represent the ROM table, and use two processes.

Present State	Next State		Output (Z)	
	$X = 0$	$X = 1$	$X = 0$	$X = 1$
S_0	S_0	S_1	0	1
S_1	S_2	S_3	1	0
S_2	S_1	S_3	1	0
S_3	S_3	S_2	0	1

3.4 The following state table is implemented using a ROM and two D flip-flops (falling edge triggered):

Q_1Q_2	$Q_1^+Q_2^+$		Z	
	$X = 0$	$X = 1$	$X = 0$	$X = 1$
00	01	10	0	1
01	10	00	1	1
10	00	01	1	0

(a) Draw the block diagram.
(b) Write Verilog code that describes the system. Assume that the ROM has a delay of 10 ns and each flip-flop has a propagation delay of 15 ns.

3.5 Find a minimum-row PLA to implement the following three functions:

$$f(A, B, C, D) = \Sigma m(3, 6, 7, 11, 15)$$
$$g(A, B, C, D) = \Sigma m(1, 3, 4, 7, 9, 13)$$
$$h(A, B, C, D) = \Sigma m(4, 6, 8, 10, 11, 12, 14, 15)$$

(a) Use Karnaugh maps to find common terms. Give the logic equations with common terms underlined, the PLA table, and also a PLA diagram similar to Figure 3-15.
(b) Use the Espresso multiple output simplification routine that is in *LogicAid*. Compare the *LogicAid* results with part **(a)**. They might not be exactly the same since *LogicAid* Espresso finds only minimum row tables; it does not necessarily minimize the number of variables in each AND term. *Note*: enter the variable names A, B, C, D, F, G, and H in *LogicAid*. Printouts with variable names X_1, X_2, X_3, X_4, and so forth are not acceptable.

3.6 Find a minimum-row PLA table to implement the following sets of functions:
(a) $f_1(A, B, C, D) = \Sigma m(0, 2, 3, 6, 7, 8, 9, 11, 13)$,
$f_2(A, B, C, D) = \Sigma m(3, 7, 8, 9, 13)$,
$f_3(A, B, C, D) = \Sigma m(0, 2, 4, 6, 8, 12, 13)$

(b) $f_1(A, B, C, D) = cd + ad + a'bc'd'$
$f_2(A, B, C, D) = bc'd' + ac' + ad'$

3.7 **(a)** Find a minimum-row PLA table to implement the following equations:

$$x(A, B, C, D) = \Sigma m(0, 1, 4, 5, 6, 7, 8, 9, 11, 12, 14, 15)$$
$$y(A, B, C, D) = \Sigma m(0, 1, 4, 5, 8, 10, 11, 12, 14, 15)$$
$$z(A, B, C, D) = \Sigma m(0, 1, 3, 4, 5, 7, 9, 11, 15)$$

(b) Indicate the connections that will be made to program a PLA to implement your solution to part **(a)** on a diagram similar to Figure 3-15.

3.8 Write VHDL code that describes the output macrocell of a 22V10 (the part enclosed by dashed lines on Figure 3-20). The entity should include S_1 and S_0. Note that the flip-flop has an asynchronous reset (AR) and a synchronous preset (SP).

3.9 An N-bit bidirectional shift register has N parallel data inputs, N outputs, a left serial input (LSI), a right serial input (RSI), a clock input, and the following control signals:

Load: Load the parallel data into the register (load overrides shift).
Rsh: Shift the register right (LSI goes into the left end).
Lsh: Shift the register left (RSI goes into the right end).

(a) If the register is implemented using a 22V10, what is the maximum value of N?
(b) Give equations for the rightmost two cells.

3.10 Show how the left shift register of Figure 2-41 could be implemented using a CPLD. Draw a diagram similar to Figure 3-25. Give the equations for the flip-flop D inputs.

3.11 A Mealy sequential circuit with four output variables is realized using a 22V10. What is the maximum number of input variables it can have? What is the maximum number of states? Can any Mealy circuit with these numbers of inputs and outputs be realized with a 22V10? Explain.

3.12 **(a)** What is the difference between a traditional gate array and an FPGA?
(b) What are the different types of FPGAs based on architecture (organization)?
(c) What are the different programming technologies for FPGAs?
(d) What is the main advantage of SRAM FPGAs?
(e) What is the main advantage of antifuse FPGAs?
(f) What are the major programmable elements in an FPGA?
(g) What are the disadvantages of SRAM FPGAs?
(h) What are the disadvantages of antifuse FPGAs?
(i) How many transistors are typically required to make an SRAM cell?
(j) What is an MPGA?
(k) What is difference between a CPLD and an FPGA?
(l) What is an advantage of a CPLD over an FPGA?
(m) What is the advantage of an FPGA over a CPLD?
(n) Name three vendors of CPLDs.
(o) Name three vendors of FPGAs.

3.13 **(a)** In which applications should a designer use a CPLD rather than an FPGA?
(b) In which applications should a designer use an MPGA rather than an FPGA?
(c) In which applications should a designer use an FPGA rather than an MPGA?
(d) A company is designing an experimental product that is in version 1 now. It is expected that the product will undergo several revisions. The company's plan is to use an FPGA for the actual design. What type of FPGA (SRAM or antifuse) should be used?
(e) A company is designing a product using an FPGA. The company's plan is to use an FPGA for the actual design. The product has undergone several revisions

and is fairly stable. Minimizing area, power, and cost is important for the company. Which type of FPGA (SRAM or antifuse) should be used?

(f) A company is designing a product. It expects to sell 1000 copies of the product. Should the company use an MPGA or an FPGA for this product?

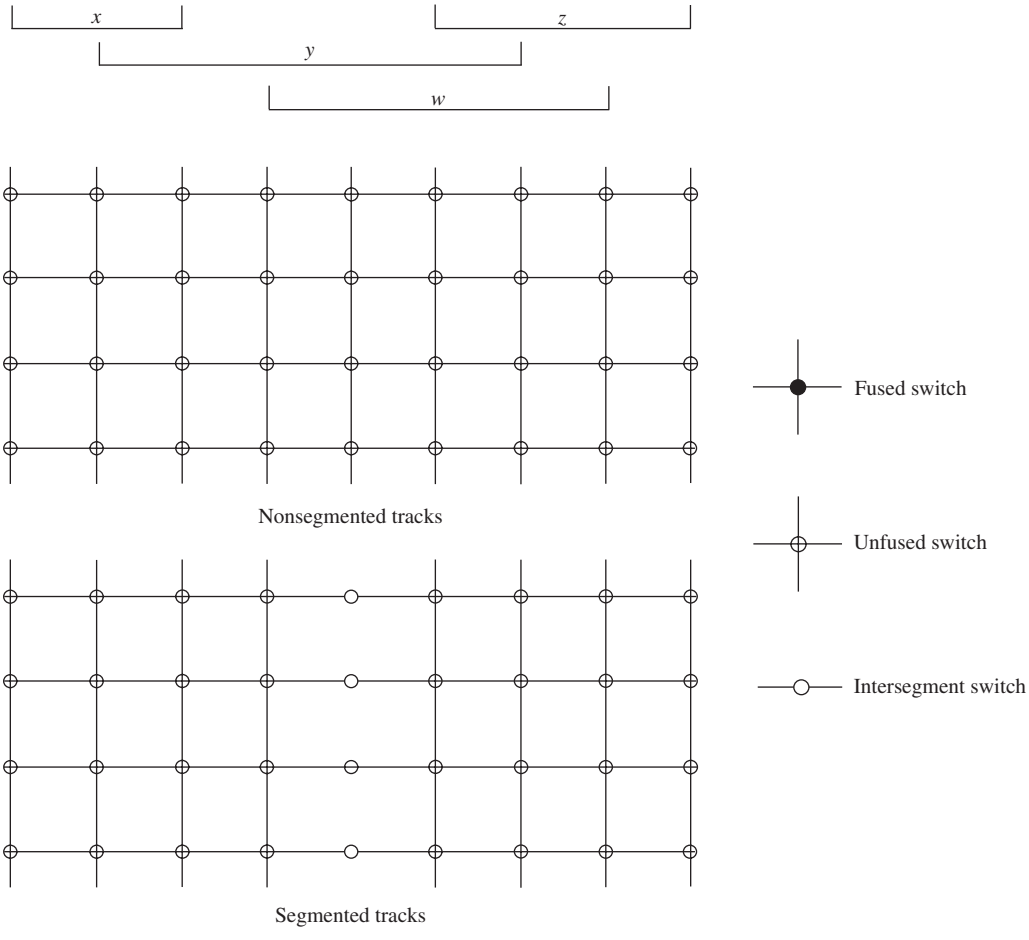
(g) A company is designing a product. It expects to sell 100 million copies of the product. Should the company use an MPGA or an FPGA for this product?

3.14 **(a)** Implement the function $F_1 = A'BC + B'C + ABC$ using an FPGA with programmable logic blocks consisting of 4-to-1 multiplexers. Assume inputs and their complements are available as shown in Figure 3-34.

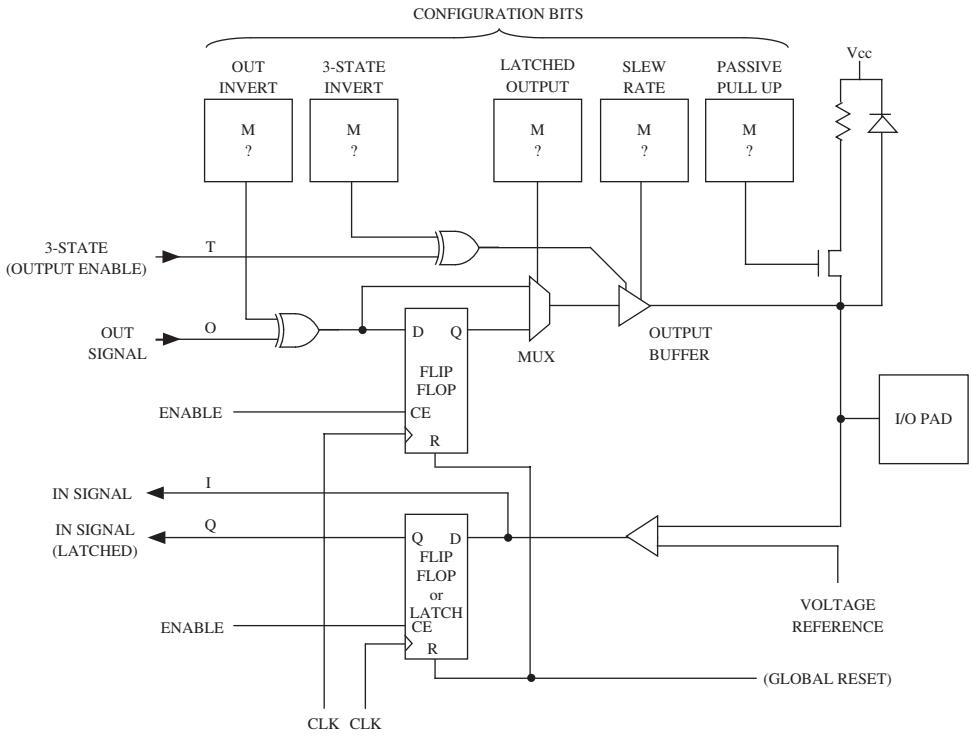
(b) Implement the function $F_1 = A'B + AB' + AC' + A'C$ using a multiplexer. What is the size of the smallest multiplexer needed, assuming inputs and their complements are available?

3.15 **(a)** Route the "w", "x", "y", and "z" nets on the non-segmented tracks shown in the diagram that follows. Use the minimum number of tracks possible.

(b) Route the "w", "x", "y", and "z" nets on the segmented tracks shown in the diagram that follows. Use the minimum number of tracks possible.



3.16 Consider the following programmable I/O block:



Highlight the connections to configure this I/O block as an INPUT pin. Specify the five configuration bits.

CHAPTER
4

Design Examples

In this chapter, we present several Verilog design examples to illustrate the design of small digital systems. We present the concept of dividing a design into a controller and a data path and using the control circuit to control the sequence of operations in a digital system. We use Verilog to describe a digital system at the behavioral level so that we can simulate the system to test the algorithms used. We also show how designs have to be coded structurally if specific hardware structures are to be generated.

In any design, first, one should understand the problem and the design specifications clearly. If the problem has not been stated clearly, try to get the specification of the design clarified. In real-world designs, if another team or a client company is providing your team with the specifications, getting the design specifications clarified properly can save you a lot of grief later. Good design starts with a clear specification document.

Once the problem has been stated clearly, often designers start thinking about the basic blocks necessary to accomplish what is specified. Designers often think of standard building blocks, such as adders, shift-registers, counters, and the like. Traditional design methodology splits a design into a "data path" and a "controller." The term "data path" refers to the hardware that actually performs the data processing. The controller sends control signals or commands to the data path, as shown in Figure 4-1. The controller can obtain feedback in the form of status signals from the data path.

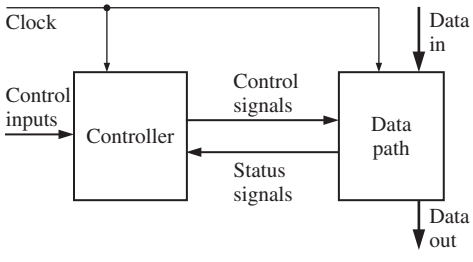


FIGURE 4-1: Separation of a Design into Data Path and Controller

In the context of a microprocessor, the data path is the ALU (Arithmetic and Logic Unit) that performs the core of the processing. The controller is the control logic that sends appropriate control signals to the data path, instructing it to

perform addition, multiplication, shifting, or whatever action is called for by the instruction. Many users have a tendency to mistakenly consider the term "data path" to be synonymous with the data bus, but "data path" in traditional design terminology refers to the actual data processing unit.

Maintaining a distinction between data path and controller helps in debugging (i.e., finding errors in the design). It also helps while modifying the design. Many modifications can be accomplished by changing only the control path because the same data path can still support the new requirements. The controller can generate the new sequence of control signals to accomplish the functionality of the modified design. Design often involves refining the data path and controller in iterations.

In this chapter, we will discuss various design examples. Several arithmetic and non-arithmetic examples are presented. Non-arithmetic examples include a 7-segment decoder, a traffic light, a scoreboard, and a keypad scanner. Arithmetic circuits such as adders, multipliers, and dividers are also presented.

4.1 BCD to 7-Segment Display Decoder

Seven segment displays are often used to display digits in digital counters, watches, and clocks. A digital watch displays time by turning on a combination of the segments on a 7-segment display. For this example, the segments are labeled as follows, and the digits have the forms as indicated in Figure 4-2.

FIGURE 4-2: 7-Segment Display

Let us design a BCD to 7-segment display decoder. BCD stands for "binary coded decimal." In this format, each digit of a decimal number is encoded into 4-bit binary representation. This decoder is a purely combinational circuit; hence, no state machine is involved here. A block diagram of the decoder is shown in Figure 4-3. The decoder for one BCD digit is presented.

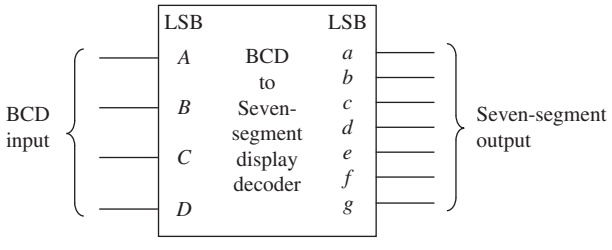


FIGURE 4-3: Block Diagram of a BCD to 7-Segment Display Decoder

We will create a behavioral Verilog architectural description of this BCD to 7-segment decoder by using a single process with a case statement to model this

combinational circuit, as in Figure 4-4. The sensitivity list of the process consists of the BCD number (4 bits).

FIGURE 4-4: Behavioral Verilog Code for BCD to 7-Segment Decoder

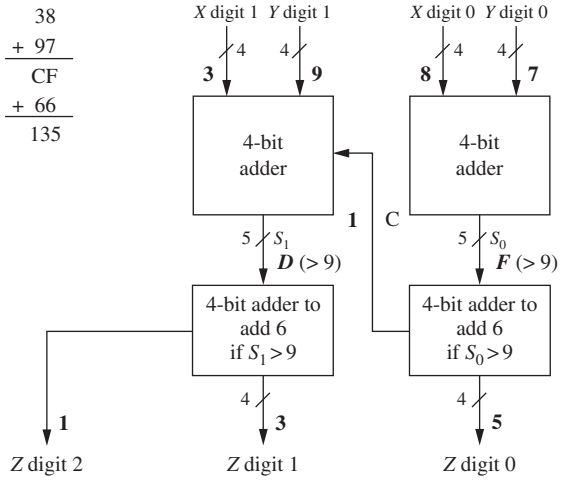
```
module bcd_seven (bcd, seven);
  input [3:0] bcd;
  output[7:1] seven;

  reg   [7:1] seven;

  always @(bcd)
  begin
    case (bcd)
      4'b0000 : seven = 7'b0111111 ;
      4'b0001 : seven = 7'b0000110 ;
      4'b0010 : seven = 7'b1011011 ;
      4'b0011 : seven = 7'b1001111 ;
      4'b0100 : seven = 7'b1100110 ;
      4'b0101 : seven = 7'b1101101 ;
      4'b0110 : seven = 7'b1111101 ;
      4'b0111 : seven = 7'b0000111 ;
      4'b1000 : seven = 7'b1111111 ;
      4'b1001 : seven = 7'b1101111 ;
      default : seven = 7'b0000000 ;
    endcase
  end
endmodule
```

4.2 A BCD Adder

In this example, we design a 2-digit BCD adder, which will add two BCD numbers and produce the sum in BCD format. In BCD representation, each decimal digit is encoded into binary. For instance, decimal number 97 will be represented as 1001 0111 in the BCD format, where the first 4 bits represent digit 9 and the next 4 bits represent digit 7. One may note that the BCD representation is different from the binary representation of 97, which is 110001. It takes 8 bits to represent 97 in BCD, whereas the binary representation of 97 (110001) only requires 6 bits. The 4-bit binary combinations 1010, 1011, 1100, 1101, 1110, and 1111 corresponding to hexadecimal numbers *A* to *F* are not used in the BCD representation. Since 6 out of 16 representations possible with 4 binary bits are skipped, a BCD number will take more bits than the corresponding binary representation.

When BCD numbers are added, each sum digit should be adjusted to skip the six unused codes. For instance, if 6 is added with 8, the sum is 14 in decimal form. A binary adder would yield 1110, but the lowest digit of the BCD sum should read 4. In order to obtain the correct BCD digit, 6 should be added to the sum whenever it is greater than 9. Figure 4-5 illustrates the hardware that will be required to

FIGURE 4-5: Addition of Two BCD Numbers



perform the addition of 2 BCD digits. A binary adder adds the least significant digits. If the sum is greater than 9, an adder adds 6 to yield the correct sum digit along with a carry digit to be added with the next digit. The addition of the higher digits is performed in a similar fashion.

The Verilog code for the BCD adder is shown in Figure 4-6. The input BCD numbers are represented by *X* and *Y*. The BCD sum of two 2-digit BCD numbers can exceed two digits and hence three BCD digits are provided for the sum, which is represented by *Z*. The compiler directive 'define' can be used to denote each digit of each BCD number. For example, the upper digit of *X* can be denoted by *Xdig1* by using the Verilog statement:

```
`define      Xdig1      X[7:4]
```

This statement allows us to use the name *Xdig1* whenever we wish to refer to the upper digit of *X*. If BCD numbers 97 and 38 are added, the sum is 135; hence, *Zdig2*

FIGURE 4-6: Verilog Code for BCD Adder

```
`define Xdig1 X[7:4]
`define Xdig0 X[3:0]
`define Ydig1 Y[7:4]
`define Ydig0 Y[3:0]
`define Zdig2 Z[11:8]
`define Zdig1 Z[7:4]
`define Zdig0 Z[3:0]

 module BCD_Adder (X, Y, Z);

   input[7:0] X;
   input[7:0] Y;
   output[11:0] Z;
```

```
    wire[4:0] S0;
    wire[4:0] S1;
    wire C;

    assign S0 = `Xdig0 + `Ydig0 ;
    assign `Zdig0 = (S0 > 9) ? S0[3:0] + 6 : S0[3:0] ;
    assign C = (S0 > 9) ? 1'b1 : 1'b0 ;

    assign S1 = `Xdig1 + `Ydig1 + C ;
    assign `Zdig1 = (S1 > 9) ? S1[3:0] + 6 : S1[3:0] ;
    assign `Zdig2 = (S1 > 9) ? 4'b0001 : 4'b0000 ;
endmodule
```

equals 1, *Zdig1* equals 3 and *Zdig0* equals 5. In Verilog code, the defined name should be used with ` (e.g., `Xdig1 or `Zdig2).

During the addition of the second digit, the carry digit from the addition of the *XDig0* and *Ydig0* is also added. Thus, the addition of the second digit is accomplished by the statement:

```
assign S1 = `Xdig1 + `Ydig1 + C ;
```

4.3 32-Bit Adders

Let us assume that we have to design a 32-bit adder. A simple way to construct an adder is to build a **ripple-carry adder**, as shown in Figure 4-7. In this type of adder, 32 copies of a one-bit full adder are connected in succession to create the 32-bit adder. The carry "ripples" from the least significant bit to the most significant bit. If gate delays are t_g, a one-bit adder delay is $2*t_g$ (assuming a sum-of-products expression for sum and carry, and ignoring delay for inverters), and a 32-bit ripple-carry adder will take approximately 64 gate delays. For instance, if gate delays are 1 ns, the maximum frequency at which the 32-bit ripple-carry adder can operate is approximately 16 MHz! This is inadequate for many applications. Hence, designers often resort to faster adders.

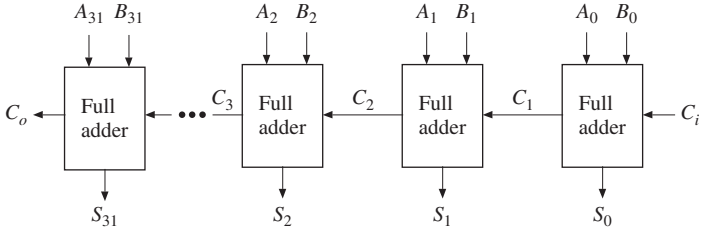


FIGURE 4-7: A 32-Bit Ripple-Carry Adder

Carry Look-Ahead Adders

A popular fast-addition technique is carry look-ahead (CLA) addition. In the carry look-ahead adder, the carry signals are calculated in advance, based on the input signals. For any bit position *i*, one can see that a carry will be generated if the

corresponding input bits (i.e., A_i, B_i) are 1 or if there was a carry-in to that bit and at least one of the input bits are 1. In other words, bit i has carry-out if A_i and B_i are 1 (irrespective of carry-in to bit i); bit i also has a carry-out if $C_i = 1$ and either A_i or B_i is 1. Thus, for any stage i, the carry-out is

$$C_{i+1} = A_iB_i + (A_i \oplus B_i) \cdot C_i \tag{4-1}$$

The "⊕" stands for the exclusive OR operation. Equation 4-1 simply expresses that there is a carry out from a bit position if it **generated** a carry by itself (e.g., $A_iB_i = 1$) or it simply **propagated** the carry from the lower bit forwarded to it (i.e., $(A_i \oplus B_i) \cdot C_i$).

Since $A_iB_i = 1$ indicates that a stage generated a carry, a general **generate (Gi) function** may be written as

$$G_i = A_iB_i \tag{4-2}$$

Similarly, since $(A_i \oplus B_i)$ indicates whether a stage should propagate the carry it receives from the lower stage, a general **propagate (P_i) function** may be written as

$$P_i = A_i \oplus B_i \tag{4-3}$$

Notice that the propagate and generate functions depend only on the input bits and can be realized with one or two gate delays. Since there will be a carry whether one of A_i or B_i is 1 or both are 1, one can also write the propagate expression as

$$P_i = A_i + B_i \tag{4-4}$$

where the OR operation is substituted for the XOR operation. Logically this propagate function also results in the correct carry-out; however, traditionally it has been customary to define the propagate function as the XOR; that is, the bit position simply propagates a carry (without generating a carry by itself). Also, typically, the sum signal is expressed as

$$S_i = A_i \oplus B_i \oplus C_i = P_i \oplus C_i \tag{4-5}$$

The expression $P_i \oplus C_i$ can be used for sum only if P_i is defined as $A_i \oplus B_i$.

The carry-out equation can be rewritten by substituting (4-2) and (4-3) in (4-1) for G_i and P_i as:

$$C_{i+1} = G_i + P_iC_i \tag{4-6}$$

In a 4-bit adder, the $C_i s$ can be generated by repeatedly applying Equation 4-6 as shown here

$$C_1 = G_0 + P_0C_0 \tag{4-7}$$

$$C_2 = G_1 + P_1C_1 = G_1 + P_1G_0 + P_1P_0C_0 \tag{4-8}$$

$$C_3 = G_2 + P_2C_2 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0 \tag{4-9}$$

$$C_4 = G_3 + P_3C_3 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0 \tag{4-10}$$

These carry bits are the look-ahead carry bits. They are expressed in terms of $P_i s$, $G_i s$, and C_0. Thus, the sum and carry from any stage can be calculated without

waiting for the carry to ripple through all the previous stages. Since G_is and P_is can be generated with one or two gate delays, the C_is will be available in three or four gate delays. The advantage is that these delays will be the same, independent of the number of bits one needs to add, in contrast to the ripple counter. Of course this is achieved with the extra gates to generate the look-ahead carry bits. A 4-bit carry look-ahead adder can now be built, as illustrated in Figure 4-8. The output of partial full adder will be **propagate** (P) and **generate** (G) in addition to **sum** (S), and **carry** (C) will be generated by carry look-ahead logic rather than the partial full adder. C_4 can also be generated with the following equation:

$$C_4 = G_G + P_G C_0 \tag{4-11}$$

This is accomplished by computing a **group propagate** (P_G) and **group generate** (G_G) signal, which is produced by **carry look-ahead logic**:

$$P_G = P_3 P_2 P_1 P_0 \tag{4-12}$$

$$G_G = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 \tag{4-13}$$

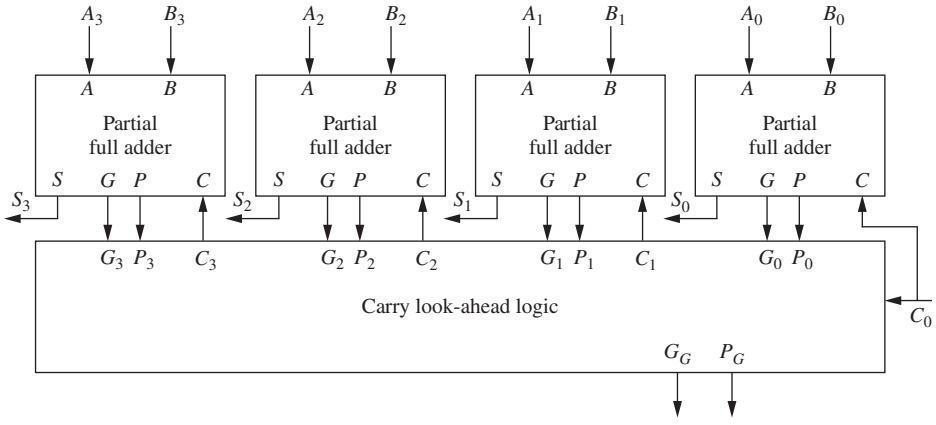


FIGURE 4-8: Block Diagram of a 4-Bit CLA

The disadvantage of the carry look-ahead adder is that the look-ahead carry logic as shown in Equations 4-7 through 4-13, is not simple. It gets quite complicated for more than 4 bits. For that reason, carry look-ahead adders are usually implemented as 4-bit modules and are used in a hierarchical structure to realize adders that have multiples of 4 bits. Figure 4-9 shows the block diagram for a 16-bit carry look-ahead adder. Four carry look-ahead adders, similar to the ones introduced previously, are used. Instead of relying on each 4-bit adder to send its carry-out to the next 4-bit adder, the **block carry look-ahead logic** generates input carry bits to be fed to each 4-bit adder using a group propagate (P_G) and group generate (G_G) signal, which is produced by each 4-bit adder. The next level of carry look-ahead logic uses these group propagates/generates and generates the required carry bits in parallel. The propagate for a group is true if all the propagates in that group are true. The generate for a group is true if the MSB generated a carry or if a lower bit generated a carry and every higher bit in the group propagated it.

The group propagate P_G and generate G_G will be available after three and four gate delays, respectively (one or two additional delays than the P_i and G_i signals,

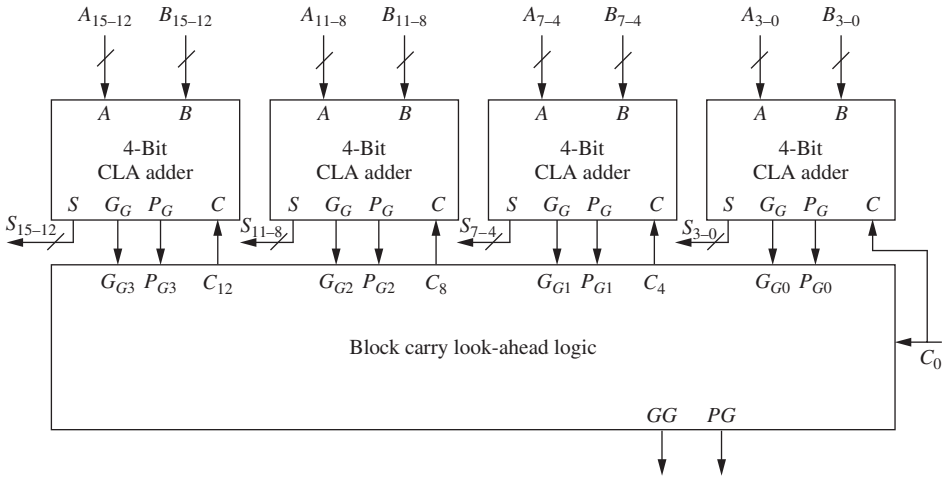


FIGURE 4-9: Block Diagram of a 16-Bit CLA

respectively). The carry equations for the block carry look-ahead logic are as follows:

$$C_4 = G_{G0} + P_{G0}C_0 \tag{4-14}$$

$$C_8 = G_{G1} + P_{G1}G_{G0} + P_{G1}P_{G0}C_0 \tag{4-15}$$

$$C_{12} = G_{G2} + P_{G2}G_{G1} + P_{G2}P_{G1}G_{G0} + P_{G2}P_{G1}P_{G0}C_0 \tag{4-16}$$

C_{16}, which is a final carry of 16-bit CLA, will be

$$C_{16} = GG + PG\ C_0 \tag{4-17}$$

One can derive the propagate (PG) and generate (GG) equation for block carry look-ahead logic in a manner similar to equation 4-12 and 4-13.

Figure 4-10 illustrates the Verilog description of a 4-bit carry look-ahead adder.

FIGURE 4-10: Verilog Description of a 4-Bit Carry Look-Ahead Adder

```
module CLA4 (A, B, Ci, S, Co, PG, GG);
   input[3:0] A;
   input[3:0] B;
   input Ci;
   output[3:0] S;
   output Co;
   output PG;
   output GG;

   wire[3:0] G;
   wire[3:0] P;
   wire[3:1] C;
   CLALogic CarryLogic (G, P, Ci, C, Co, PG, GG);
   GPFullAdder FA0 (A[0], B[0], Ci, G[0], P[0], S[0]);
   GPFullAdder FA1 (A[1], B[1], C[1], G[1], P[1], S[1]);
   GPFullAdder FA2 (A[2], B[2], C[2], G[2], P[2], S[2]);
   GPFullAdder FA3 (A[3], B[3], C[3], G[3], P[3], S[3]);
endmodule
```

```
module CLALogic (G, P, Ci, C, Co, PG, GG);
   input[3:0] G;
   input[3:0] P;
   input Ci;
   output[3:1] C;
   output Co;
   output PG;
   output GG;

   wire GG_int;
   wire PG_int;

   assign C[1] = G[0] | (P[0] & Ci) ;
   assign C[2] = G[1] | (P[1] & G[0]) | (P[1] & P[0] & Ci) ;
   assign C[3] = G[2] | (P[2] & G[1]) | (P[2] & P[1] & G[0]) | (P[2] & P[1] &
                 P[0] & Ci) ;
   assign PG_int = P[3] & P[2] & P[1] & P[0] ;
   assign GG_int = G[3] | (P[3] & G[2]) | (P[3] & P[2] & G[1]) | (P[3] & P[2] &
                   P[1] & G[0]) ;
   assign Co = GG_int | (PG_int & Ci) ;
   assign PG = PG_int ;
   assign GG = GG_int ;
endmodule

module GPFullAdder (X, Y, Cin, G, P, Sum);
   input X;
   input Y;
   input Cin;
   output G;
   output P;
   output Sum;

   wire P_int;

   assign G = X & Y ;
   assign P = P_int ;
   assign P_int = X ^ Y ;
   assign Sum = P_int ^ Cin ;
endmodule
```

Verilog code for a 16-bit carry look-ahead adder can be developed by instantiating four copies of the 4-bit carry look-ahead adder and one additional copy of the carry look-ahead logic. A 64-bit adder can be built by one more level of block carry look-ahead logic. The delay increases by only two gate delays when the adder size increases from 16 bits to 64 bits. Developing Verilog code for 16- and 64-bit carry look-ahead logic is left as subjects of exercise problems.

Figure 4-11 illustrates behavioral Verilog code for a 32-bit adder using the "+" operator. If this code is synthesized, depending on the tools used and the target technology, an adder with characteristics in between a ripple-carry adder and a fast 2-level adder will be obtained. The various topologies result in different area, power, and delay characteristics.

FIGURE 4-11: Behavioral Model for a 32-Bit Adder

```
module Adder32 (A, B, Ci, S, Co);

    input[31:0] A;
    input[31:0] B;
    input Ci;
    output[31:0] S;
    output Co;

    wire[32:0] Sum33;

    assign Sum33 = A + B + Ci ;
    assign S = Sum33[31:0] ;
    assign Co = Sum33[32] ;
endmodule
```

Example

If gate delays are t_g, what is the delay of the fastest 32-bit adder? Assume that the amount of hardware consumed is not a constraint. Only speed is important.

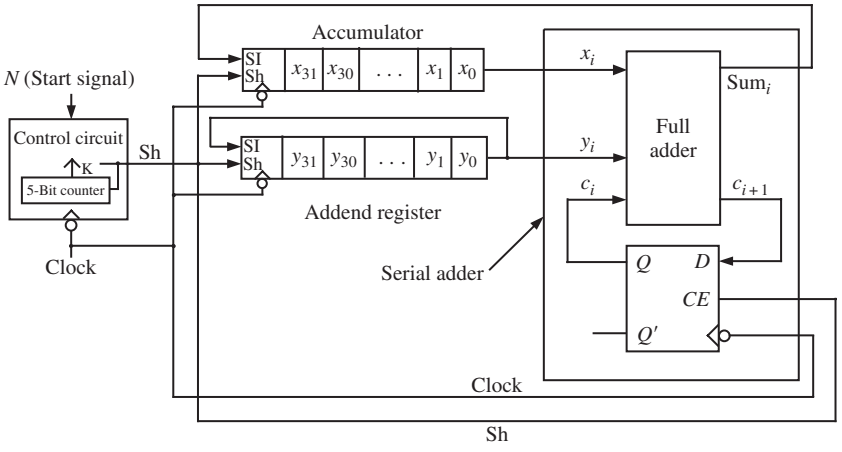
Answer: One can express each sum bit of a 32-bit adder as a sum-of-products expression of the input bits. There will be 33 such equations, including one for the carry out bit. These equations will be very long, and some of them could include 60+ variables in the product term. Nevertheless, if gates with any number of inputs are available, theoretically a 2-level adder can be made. Although it is not very practical, theoretically, the delay of the fastest adder will be $2t_g$ if gate delays are t_g.

Example

Is ripple-carry adder the smallest 32-bit adder?

Answer: A 32-bit ripple-carry adder uses 32 1-bit adders. One could design a 32-bit serial adder using a single 1-bit full adder. The input numbers are shifted into the adder, one bit at a time, and carry output from addition of each pair of bits is saved in a flip-flop and fed back to the next addition. The hardware illustrated in Figure 4-12 accomplishes this. The delay

FIGURE 4-12: A 32-Bit Serial Adder Built from a Single 1-Bit Adder



of adder will be 32 * ($2t_g + t_{ff}$), where $2t_g$ is the delay of the one-bit full adder and t_{ff} is the delay of the flip-flop (including set up time). If a flip-flop delay is at least two gate delays, the delay of the 32-bit serial adder will be at least $128t_g$. The adder hardware is simple; however, there is also the control circuitry needed to generate 32 shift signals. The registers storing the operands must have shift capability as well.

Even if you write Verilog code based on data flow equations, as shown in Figure 4-10, that does not guarantee that the synthesizer will produce a carry look-ahead adder with the delay characteristics we have discussed. The software might optimize the synthesis output depending on the specific hardware components available in the target technology. For instance, if you are using an FPGA with fast adder support, the software may map some of the functions into the fast adder circuitry. Depending on the number of FPGA logic blocks and interconnects used, the delays will be different from the manual calculations. The delays of a ripple-carry, carry look-ahead, and serial adder for a gate-based implementation are presented in Table 4-1 for various adder sizes. One can see that the carry look-ahead adder is very attractive for large adders.

TABLE 4-1: Comparison of Ripple-Carry and Carry Look-Ahead Adders

Adder size	Ripple-carry adder delay	CLA delay	Serial adder delay
4 bit	8 t_g	5-6 t_g	16 t_g
16 bit	32 t_g	7-8 t_g	64 t_g
32 bit	64 t_g	9-10 t_g	128 t_g
64 bit	128 t_g	9-10 t_g	256 t_g

4.4 Traffic Light Controller

Let us design a sequential traffic light controller for the intersection of street "A" and street "B." Each street has traffic sensors, which detect the presence of vehicles approaching or stopped at the intersection. $Sa = 1$ means a vehicle is approaching on street "A," and $Sb = 1$ means a vehicle is approaching on street "B." Street "A" is a main street and has a green light until a car approaches on "B." Then the lights change, and "B" has a green light. At the end of 50 seconds, the lights change back unless there is a car on street "B" and none on "A," in which case the "B" cycle is extended for 10 additional seconds. If cars continue to arrive on street "B" and no car appears on street "A," "B" continues to have a green light. When "A" is green, it remains green at least 60 seconds, and then the lights change only when a car approaches on "B." Figure 4-13 shows the external connections to the controller. Three of the outputs (*Ga*, *Ya*, and *Ra*) drive the green, yellow, and red lights on

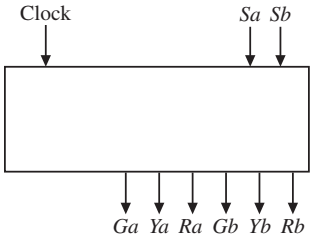


FIGURE 4-13: Block Diagram of Traffic Light Controller

street "A." The other three (*Gb*, *Yb*, and *Rb*) drive the corresponding lights on street "B."

Figure 4-14 shows a Moore state graph for the controller. For timing purposes, the sequential circuit is driven by a clock with a 10-second period. Thus, a state change can occur at most once every 10 seconds. The following notation is used: *GaRb* in a state means that $Ga = Rb = 1$ and all the other output variables are 0. *Sa'Sb* on an arc implies that $Sa = 0$ and $Sb = 1$ will cause a transition along that arc. An arc without a label implies that a state transition will occur when the clock occurs, independent of the input variables. Thus, the green "A" light will stay on for six clock cycles (60 seconds) and then change to yellow if a car is waiting on street "B."

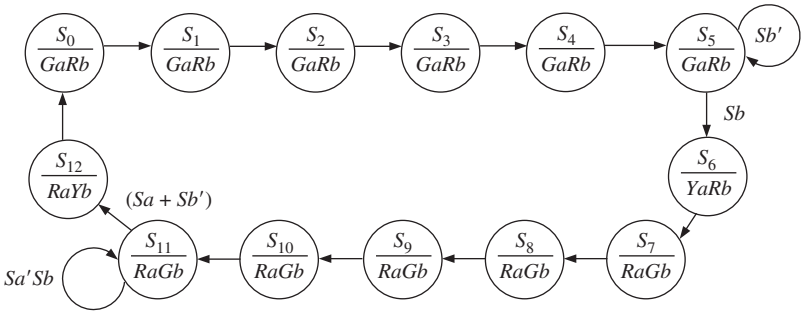


FIGURE 4-14: State Graph for Traffic Light Controller

The Verilog code for the traffic light controller (Figure 4-15) represents the state machine with two always statements. Whenever the state—*Sa* or *Sb*—changes, the first always statement updates the outputs and *nextstate*. When the rising edge of the clock occurs, the second always statement updates the state register. The case statement illustrates the use of multiple case numbers. Since states S_0 through S_4 have the same outputs and the next states are in numeric sequence, we use multiple case numbers together instead of five separate case behaviors.

```
0, 1, 2, 3, 4 :
        begin
            Ga_tmp = 1'b1 ;
            Rb_tmp = 1'b1 ;
            nextstate = state + 1 ;
        end
```

FIGURE 4-15: Verilog Code for Traffic-Light Controller

```
module traffic_light (clk, Sa, Sb, Ra, Rb, Ga, Gb, Ya, Yb);
input clk;
input Sa;
input Sb;
inout Ra;
inout Rb;
inout Ga;
inout Gb;
inout Ya;
inout Yb;

reg Ra_tmp;
reg Rb_tmp;
reg Ga_tmp;
reg Gb_tmp;
reg Ya_tmp;
reg Yb_tmp;

reg[3:0] state;
reg[3:0] nextstate;
parameter[1:0] R = 0;
parameter[1:0] Y = 1;
parameter[1:0] G = 2;
wire[1:0] lightA;
wire[1:0] lightB;

assign Ra = Ra_tmp;
assign Rb = Rb_tmp;
assign Ga = Ga_tmp;
assign Gb = Gb_tmp;
assign Ya = Ya_tmp;
assign Yb = Yb_tmp;

initial
begin
   state = 0;
end

always @(state or Sa or Sb)
begin
   Ra_tmp = 1'b0 ;
   Rb_tmp = 1'b0 ;
   Ga_tmp = 1'b0 ;
   Gb_tmp = 1'b0 ;
   Ya_tmp = 1'b0 ;
   Yb_tmp = 1'b0 ;
   nextstate = 0;

   case (state)
      0, 1, 2, 3, 4 :
                begin
```

```
                    Ga_tmp = 1'b1 ;
                    Rb_tmp = 1'b1 ;
                    nextstate = state + 1 ;
                end
        5 :
                begin
                    Ga_tmp = 1'b1 ;
                    Rb_tmp = 1'b1 ;
                    if (Sb == 1'b1)
                    begin
                        nextstate = 6 ;
                    end
                    else
                    begin
                        nextstate = 5 ;
                    end
                end
        6 :
                begin
                    Ya_tmp = 1'b1 ;
                    Rb_tmp = 1'b1 ;
                    nextstate = 7 ;
                end
        7, 8, 9, 10 :
                begin
                    Ra_tmp = 1'b1 ;
                    Gb_tmp = 1'b1 ;
                    nextstate 5 state + 1 ;
                end
        11 :
                begin
                    Ra_tmp = 1'b1 ;
                    Gb_tmp = 1'b1 ;
                    if (Sa == 1'b1 | Sb == 1'b0)
                    begin
                        nextstate = 12 ;
                    end
                    else
                    begin
                        nextstate = 11 ;
                    end
                end
        12 :
                begin
                    Ra_tmp = 1'b1 ;
                    Yb_tmp = 1'b1 ;
                    nextstate = 0 ;
                end
    endcase
end
```

```
always @(posedge clk)
begin
   state <= nextstate ;
end

assign lightA = (Ra==1'b1) ? R : (Ya==1'b1) ? Y : (Ga==1'b1) ? G : lightA;
assign lightB = (Rb==1'b1) ? R : (Yb==1'b1) ? Y : (Gb==1'b1) ? G : lightB;

endmodule
```

For each state, only the signals that are 1 are listed within the case statement. Since in Verilog a signal will hold its value until it is changed, we should turn off each signal when the next state is reached. In state 6 we should set *Ga* to 0, in state 7 we should set *Ya* to 0, and so forth. This could be accomplished by inserting appropriate statements in each case. For example, we could insert `Ga <= '0'` in the **`case of state=6 -> case 6.`** An easier way to turn off the outputs is to set them all to 0 before the case statement, as shown in Figure 4-15. At first, it seems that a glitch might occur in the output when we set a signal to 0 that should remain 1. However, this is not a problem, because the sequential statements within an always statement execute instantaneously. For example, suppose that at time = 20 ns a state change from S_2 to S_3 occurs. *Ga* and *Rb* are 1, but as soon as the always statement starts executing, the first line of code is executed and *Ga* and *Rb* are scheduled to change to 0 at time 20 + D. The case statement then executes, and *Ga* and *Rb* are scheduled to change to 1 at time 20 + D. Since this is the same time as before, the new value (1) preempts the previously scheduled value (0), and the signals never change to 0.

One may also notice the statement

```
nextstate = 0;
```

before the case statement. This statement helps to avoid latches for `nextstate` even if all cases are not specified. In this example, only 13 of the 16 cases are specified and latches will be generated for `nextstate` unless a default clause is created where `nextstate` is set to 0. The basic thing to remember is that `nextstate` is a combinational signal and `state` is the actual signal that holds the state value in flip-flops.

Before completing the design of the traffic controller, we will test the Verilog code to see that it meets specifications. As a minimum, our test sequence should cause all of the arcs on the state graph to be traversed at least once. We may want to perform additional tests to check the timing for various traffic conditions, such as heavy traffic on both "A" and "B," light traffic on both, heavy traffic on "A" only, heavy traffic on "B" only, and special cases such as a car failing to move when the light is green, a car going through the intersection when the light is red and so forth.

To make it easier to interpret the simulator output, we define a type named light with the values R, Y, and G and two signals, *lightA* and *lightB*, which can assume these values. Then we add code to set *lightA* to R when the light is red, to Y when

the light is yellow, and to G when the light is green. The following simulator command file first tests the case where both self-loops on the graph are traversed and then the case where neither self-loop is traversed:

```
add wave clk Sa Sb state lightA lightB
force clk 0 0, 1 5 sec -repeat 10 sec
force Sa 1 0, 0 40, 1 170, 0 230, 1 250 sec
force Sb 0 0, 1 70, 0 100, 1 120, 0 150, 1 210, 0 250, 1 270 sec
run 300 sec
```

The test results in Figure 4-16 verify that the traffic lights change at the specified times.

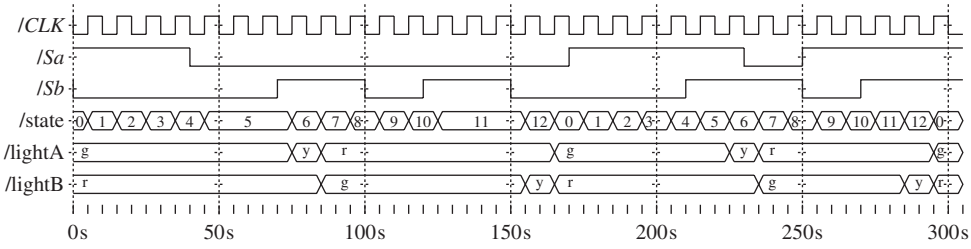


FIGURE 4-16: Test Results for Traffic Light Controller

4.5 State Graphs for Control Circuits

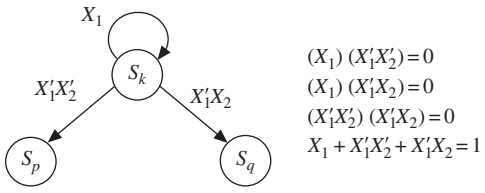
Before continuing with additional examples, we describe the notation we use on control state graphs and then state the conditions that must be satisfied to have a proper state graph. We usually label control state graphs using variable names instead of 0s and 1s. This makes the graph easier to read, especially when the number of inputs and outputs is large. If we label an arc on a Mealy state graph X_iX_j/Z_pZ_q, this means if inputs X_i and X_j are 1 (we don't care what the other input values are), the outputs Z_p and Z_q are 1 (and the other outputs are 0), and we will traverse this arc to go to the next state. For example, for a circuit with four inputs (X_1, X_2, X_3, X_4) and four outputs (Z_1, Z_2, Z_3, Z_4), the label X_1X_4'/Z_2Z_3 is equivalent to 1–0/0110. In general, if we label an arc with an input expression, I, we will traverse the arc when $I = 1$. For example, if the input label is $AB + C'$, we will traverse the arc when $AB + C' = 1$.

In order to have a completely specified proper state graph in which the next state is always uniquely defined for every input combination, we must place the following constraints on the input labels for every state S_k:

1. If I_i and I_j are any pair of input labels on arcs exiting state S_k, then $I_iI_j = 0$ if $i \neq j$.
2. If n arcs exit state S_k and the n arcs have input labels I_1, I_2, . . ., I_n, respectively, then $I_1 + I_2 + \cdots + I_n = 1$.

Condition 1 assures us that at most one input label can be 1 at any given time, and condition 2 assures us that at least one input label will be 1 at any given time. Therefore, exactly one label will be 1, and the next state will be uniquely defined for every input combination. For example, consider the partial state graph in Figure 4-17 where $I_1 = X_1$, $I_2 = X_1'X_2'$, and $I_3 = X_1'X_2$.

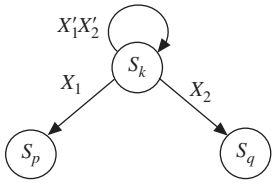
FIGURE 4-17: Example Partial State Graph



Conditions 1 and 2 are satisfied for S_k.

An incompletely specified proper state graph must always satisfy condition 2, and it must satisfy condition 1 for all combinations of values of input variables that can occur for each state S_k. Thus, the partial state graph in Figure 4-18 represents part of a proper state graph only if input combination $X_1 = X_2 = 1$ cannot occur in state S_k.

FIGURE 4-18: Example Partial State Graph



If there are three input variables (X_1, X_2, X_3), the preceding partial state graph represents the following state table row:

	000	001	010	011	100	101	110	111
S_k	S_k	S_k	S_q	S_q	S_p	S_p	–	–

4.6 Scoreboard and Controller

In this example, we will design a simple scoreboard, which can display scores from 0 to 99 (decimal). The input to the system should consist of a reset signal and control signals to increment or decrement the score. The 2-digit decimal count gets incremented by 1 if increment signal is true and is decremented by 1 if decrement signal is true. If increment and decrement are true simultaneously, no action occurs.

The current count is displayed on 7-segment displays. In order to prevent accidental erasure, the reset button must be pressed for five consecutive cycles in order to erase the scoreboard. Scoreboard should allow down counts to correct a mistake (in case of accidentally incrementing more than required).

Data Path

At the core of the design will be a 2-digit BCD counter to perform the counting. Two 7-segment displays will be needed to display the current score. One will also require BCD to 7-segment decoders to facilitate the display of each BCD digit. Figure 4-19 illustrates a block diagram of the system. Since true reset should happen

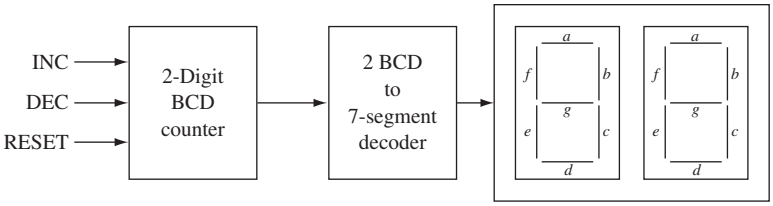


FIGURE 4-19: Overview of Simple Scoreboard

only after pressing reset for five clock cycles, we will also use a 3-bit reset counter called *rstcnt*.

Controller

The controller for this circuit works as follows. There are two states in this finite state machine (FSM), as indicated in Figure 4-20. In the initial state (S_0), the BCD counter is cleared. The reset counter is also made equal to 0. Essentially, S_0 is an initialization state where all the counters are cleared. After the initial start state, the FSM moves to the next state (S_1), which is where counting gets done. In this state, in every clock cycle, incrementing or decrementing is done according to the input signals. If reset signal *rst* arrives, the *rstcnt* is incremented. If reset count has already reached 4, and reset command is still persisting in the fifth clock cycle, a transition to state S_0 is made. If the *inc* signal is present and *dec* is not present, the BCD counter is incremented. The notation *add1* on the arc on the top-right is used to indicate that the BCD counter is incremented. If the *dec* signal is present and *inc* is not present, the BCD counter is decremented. The notation *sub1* on the arc on the bottom-right is used to indicate that the BCD counter is decremented. In any cycle that the reset signal is not present, the *rstcnt* is cleared. If both the *inc* and *dec* signals are true, or neither are true, the rest counter (*rstcnt*) is cleared and the BCD counter is left unchanged.

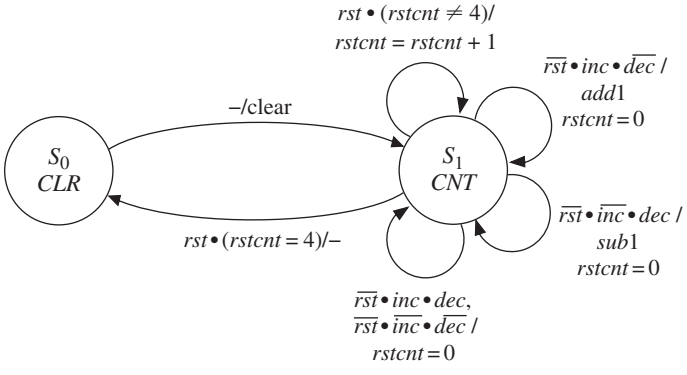


FIGURE 4-20: State Graph for Scoreboard

Verilog Model

The Verilog code for the scoreboard is given in Figure 4-21. The two 7-segment displays, *seg7disp1* and *seg7disp2*, are declared as unsigned 7-bit vectors. The segments of the 7-segment display are labeled *a* through *g*, as in Figure 4-19. The unsigned

FIGURE 4-21: Verilog Code for Scoreboard

```
module Scoreboard (clk, rst, inc, dec, seg7disp1, seg7disp0);

   input clk;
   input rst;
   input inc;
   input dec;
   output[6:0] seg7disp1;
   output[6:0] seg7disp0;

   reg[0:0] State; //Only 2 states; So 1 bit sufficient
   reg[3:0] BCD1;
   reg[3:0] BCD0;
   reg[2:0] rstcnt;
   reg[6:0] seg7Rom[0:9];

   initial
   begin
      BCD1 = 4'b0000;
      BCD0 = 4'b0000;
      rstcnt = 0;
      State = 0;

      seg7Rom[0] = 7'b0111111;
      seg7Rom[1] = 7'b0000110;
      seg7Rom[2] = 7'b1011011;
      seg7Rom[3] = 7'b1001111;
      seg7Rom[4] = 7'b1100110;
      seg7Rom[5] = 7'b1101101;
      seg7Rom[6] = 7'b1111100;
      seg7Rom[7] = 7'b0000111;
      seg7Rom[8] = 7'b1111111;
      seg7Rom[9] = 7'b1100111;
   end

   always @(posedge clk)
   begin
          case (State)
             0 :
                       begin
                          BCD1 <=4'b0000 ;
                          BCD0 <= 4'b0000 ;
                          rstcnt <= 0 ;
                          State <= 1 ;
                       end
             1 :
                       begin
                          if (rst == 1'b1)
                          begin
                             if (rstcnt == 4)
                             begin
```

```
                                State <= 0 ;
                            end
                            else
                            begin
                                rstcnt <= rstcnt + 1 ;
                            end
                        end
                        else if (inc == 1'b1 && dec == 1'b0)
                        begin
                            rstcnt <= 0 ;
                            if (BCD0 < 4'b1001)
                            begin
                                BCD0 <= BCD0 + 1 ;
                            end
                            else if (BCD1 < 4'b1001)
                            begin
                                BCD1 <= BCD1 + 1 ;
                                BCD0 <= 4'b0000 ;
                            end
                        end
                        else if (dec == 1'b1 && inc == 1'b0)
                        begin
                            rstcnt <= 0 ;
                            if (BCD0 > 4'b0000)
                            begin
                                BCD0 <= BCD0 - 1 ;
                            end
                            else if (BCD1 > 4'b0000)
                            begin
                                BCD1 <= BCD1 - 1 ;
                                BCD0 <= 4'b1001 ;
                            end
                        end
                        else if (inc == 1'b1 && dec == 1'b1)
                        begin
                          rstcnt <= 0 ;
                        end
                    end
        endcase
    end
    assign seg7disp0 = seg7Rom[BCD0] ;
    assign seg7disp1 = seg7Rom[BCD1] ;

endmodule
```

type is used so that the overloaded "+" operator can be used for incrementing the counter by one. The decoder for the 7-segment display can be implemented as an array or look-up table. The look-up table consists of 10 7-bit vectors. A reg datatype *seg7Rom* is declared for the array of the 7-segment values corresponding to each BCD digit. It is a two-dimensional array with 10 elements, each of which is a 7-bit

unsigned vector. The look-up table must be addressed with a BCD digit, and the statement

```
assign seg7disp0 = seg7rom[BCD0];
```

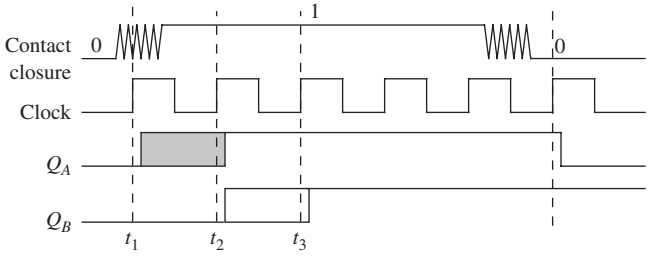
accesses the appropriate element from the array *seg7rom* to convert the decimal digit to the 7-segment form. BCD addition is accomplished with adjustment of the sum if the sum is greater than 9. If the current count is less than 9, it is incremented. If it is 9, adding 1 results in a 0, but the next digit should be incremented. Similarly, decrementing from 0 is performed by borrowing a 1 from the next higher digit.

4.7 Synchronization and Debouncing

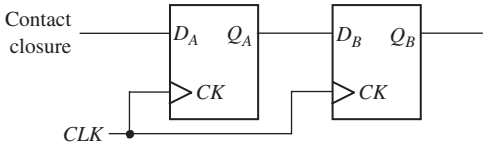
The *inc*, *dec*, and *rst* signals to the scoreboard in the previous design are external inputs. An issue in systems involving external inputs is synchronization. Outputs from a keypad or push button switches are not synchronous to the system clock signal. Since they will be used as inputs to a synchronous sequential circuit, they should be synchronized.

Another issue in systems involving external inputs is switch bounce. When a mechanical switch is closed or opened, the switch contact will bounce, causing noise in the switch output, as shown in Figure 4-22(a). The contact may bounce for several milliseconds before it settles down to its final position. After a switch closure has been detected, we must wait for the bounce to settle before reading the key. In any circuit involving mechanical switches, one should debounce the switches. Debouncing means removing the transients in the switch output.

FIGURE 4-22:
Debouncing Mechanical Switches



(a) Switch bouncing



(b) Debouncing and synchronizing circuit

Flip-flops are very useful devices when contacts must be synchronized and debounced. Figure 4-22(b) shows a proposed debouncing and synchronizing circuit. In this design, the clock period is greater than the bounce time. If the rising edge of

the clock occurs during the bounce, either a 0 or 1 will be clocked into the flip-flop at t_1. If a 0 was clocked in, a 1 will be clocked in at the next active clock edge (t_2). Consequently, it appears that Q_A will be a debounced and synchronized version of the contact closure. However, a possibility of failure exists if *the switch* changes very close to the clock edge such that the setup or hold time is violated. In this case the flip-flop output Q_A may oscillate or otherwise malfunction. Although this situation will occur very infrequently, it is best to guard against it by adding a second flip-flop. We will choose the clock period so that any oscillation at the output of Q_A will have died out before the next active edge of the clock so that the input D_B will always be stable at the active clock edge. The debounced signal, Q_B, will always be clean and synchronized with the clock, although it may be delayed up to two clock cycles after the switch is pressed.

Single Pulser

One assumption in the scoreboard design is that each time the *inc* and *dec* signals are provided, they last only for one clock cycle. Digital systems generally run at speeds higher than actions by humans, and practically, it is very difficult for humans to produce a signal that lasts only for a clock pulse. If the pressing of the button lasted longer than a clock cycle, the counters will continue to be incremented in the foregoing design. A solution to the problem is to develop a circuit that generates a single pulse for a human action of pressing a button or switch. Such a circuit can be used in a variety of applications involving humans, push buttons, and switches.

Now, let us design a single pulser circuit that delivers a synchronized pulse that is a single clock cycle long when a button is pressed. The circuit must sense the pressing of a button and assert an output signal for one clock cycle. Then the output stays inactive until the button is released.

Let us create a state diagram for the single pulser. The single pulser circuit must have two states: one in which it will detect the pressing of the key and one in which it will detect the release of the key. Let us call the first state S_0 and the second state S_1. Let us use the symbol SYNCPRESS to denote the synchronized key press. When the circuit is in state S_0 and the button is pressed, the system produces the single pulse and moves to state S_1. The single pulse is a Mealy output as the state changes from S_0 to S_1. Once the system is in state S_1, it waits for the button to be released. As soon as it is released, it moves to the start state S_0 waiting for the next button press. The single pulse output is true only during the transition from S_0 to S_1. The state diagram is illustrated in Figure 4-23.

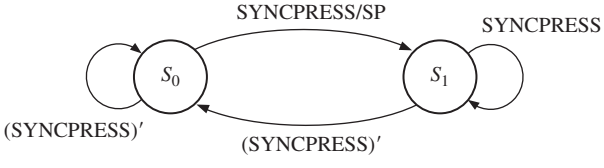


FIGURE 4-23: State Diagram of Single Pulse

Since there are only two states for this circuit, it can be implemented using one flip-flop. A single pulser can be implemented as in Figure 4-24. The first block consists of the circuitry in Figure 4-22 and generates a synchronized button press, SYNCPRESS. The flip-flop implements the two states of the state machine. Let us

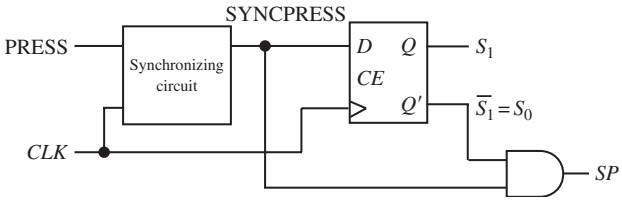


FIGURE 4-24: Single Pulser and Synchronizer Circuit

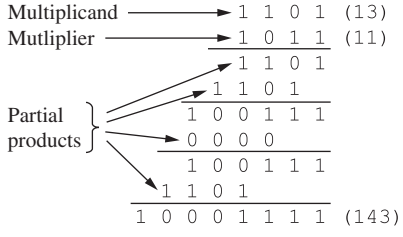
assume the state assignments are $S_0 = 0$ and $S_1 = 1$. In such a case, the Q output of the second flip-flop is synonymous with S_1, and the Q' output of the second flip-flop is synonymous with S_0. The equation for the single pulse SP is

$$SP = S_0 \cdot \text{SYNCPRESS}$$

It may also be noted that $S_0 = S_1'$. Including the 2 flip-flops inside the synchronizing block, three flip-flops can provide debouncing, synchronization, and single-pulsing. If button pushes can be passed through such a circuit, a single pulse that is debounced and synchronized, with respect to the system clock, can be obtained. It is a good practice to feed external push-button signals through such a circuit in order to obtain controlled and predictable operation.

4.8 A Shift-and-Add Multiplier

In this section, we will design a multiplier for unsigned binary numbers. When we form the product $A \times B$, the first operand (A) is called the *multiplicand* and the second operand (B) is called the *multiplier*. As illustrated here, binary multiplication requires only shifting and adding. In the following example, we multiply 13_{10} by 11_{10} in binary



Note that each partial product is either the multiplicand (1101) shifted over by the appropriate number of places or zero. Instead of forming all the partial products first and then adding, each new partial product is added in as soon as it is formed, which eliminates the need for adding more than two binary numbers at a time.

Multiplication of two 4-bit numbers requires a 4-bit multiplicand register, a 4-bit multiplier register, a 4-bit full adder, and an 8-bit register for the product. The product register serves as an accumulator to accumulate the sum of the partial products. If the multiplicand were shifted left each time before it was added to the accumulator, as was done in the previous example, an 8-bit adder would be needed. Therefore, it is better to shift the contents of the product register to the right each time, as shown in the block diagram of Figure 4-25.

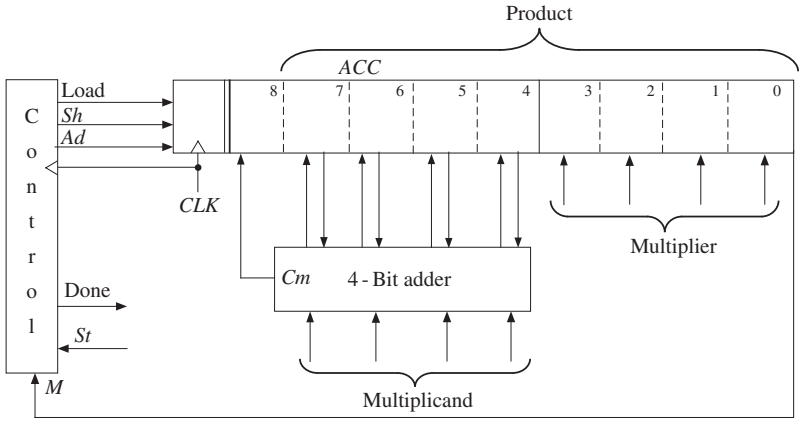


FIGURE 4-25: Block Diagram for Binary Multiplier

This type of multiplier is sometimes referred to as a serial-parallel multiplier, since the multiplier bits are processed serially, but the addition takes place in parallel. As indicated by the arrows on the diagram, 4 bits from the accumulator (ACC) and 4 bits from the multiplicand register are connected to the adder inputs; the 4 sum bits and the carry output from the adder are connected back to the accumulator. When an add signal (*Ad*) occurs, the adder outputs are transferred to the accumulator by the next clock pulse, thus causing the multiplicand to be added to the accumulator. An extra bit at the left end of the product register temporarily stores any carry that is generated when the multiplicand is added to the accumulator. When a shift signal (*Sh*) occurs, all 9 bits of ACC are shifted right by the next clock pulse.

Since the lower 4 bits of the product register are initially unused, we will store the multiplier in this location instead of in a separate register. As each multiplier bit is used, it is shifted out the right end of the register to make room for additional product bits. A shift signal (*Sh*) causes the contents of the product register (including the multiplier) to be shifted right one place when the next clock pulse occurs. The control circuit puts out the proper sequence of add and shift signals after a start signal ($St = 1$) has been received. If the current multiplier bit (M) is 1, the multiplicand is added to the accumulator followed by a right shift; if the multiplier bit is 0, the addition is skipped and only the right shift occurs. The multiplication example (13 × 11) is reworked as follows showing the location of the bits in the registers at each clock time:

initial contents of product register	0 0 0 0 0 | 1 0 1 1 ← M	(11)
(add multiplicand since $M = 1$)	1 1 0 1 |	(13)
after addition	0 1 1 0 1 | 1 0 1 1	
after shift	0 0 1 1 0 1 | 1 0 1 ← M	
(add multiplicand since $M = 1$)	1 1 0 1 |	
after addition	1 0 0 1 1 1 | 1 0 1	
after shift	0 1 0 0 1 1 1 | 1 0 ← M	
(skip addition since $M = 0$)		
after shift	0 0 1 0 0 1 1 1 | 1 ← M	
(add multiplicand since $M = 1$)	1 1 0 1 |	
after addition	1 0 0 0 1 1 1 1 | 1	
after shift (final answer)	0 1 0 0 0 1 1 1 | 1	(143)

dividing line between product and multiplier

The control circuit must be designed to output the proper sequence of add and shift signals. Figure 4-26 shows a state graph for the control circuit. In Figure 4-26, S_0 is the reset state, and the circuit stays in S_0 until a start signal ($St = 1$) is received. This generates a *Load* signal, which causes the multiplier to be loaded into the lower 4 bits of the accumulator (ACC) and the upper 5 bits of the accumulator to be cleared. In state S_1, the low-order bit of the multiplier (M) is tested. If $M = 1$, an add signal is generated, and if $M = 0$, a shift signal is generated. Similarly, in states S_3, S_5, and S_7, the current multiplier bit (M) is tested to determine whether to generate an add or shift signal. A shift signal is always generated at the next clock time following an add signal (states S_2, S_4, S_6, and S_8). After four shifts have been generated, the control network goes to S_9 and a done signal is generated before returning to S_0.

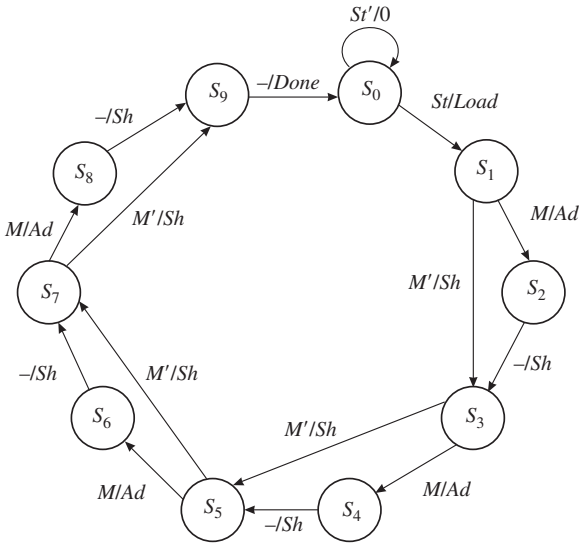


FIGURE 4-26: State Graph for Binary Multiplier Control

The behavioral Verilog model (Figure 4-27) corresponds directly to the state graph. Since there are 10 states, we have declared an integer ranging from 0 to 9 for the state signal. The signal *ACC* represents the 9-bit accumulator output. The statement

```
`define M ACC[0]
```

allows us to use the name *M* in place of *ACC(0)*. The notation 1, 3, 5, 7 : means when the state is 1 or 3 or 5 or 7, the action that follows occurs. All register operations and state changes take place on the rising edge of the clock. For example, in state 0, if *St* is 1, the multiplier is loaded into the accumulator at the same time the state changes to 1. The expression `{1'b0, ACC[7:4]} + Mcand` is used to compute the sum of two 4-bit unsigned vectors to give a 5-bit result. This represents the adder output, which is loaded into *ACC* at the same time the state counter is incremented. The right shift on *ACC* is accomplished by loading *ACC* with 0 concatenated with the upper 8 bits of *ACC*. The expression `{1'b0, ACC[8:1]}` could be replaced with `ACC >> 1`.

FIGURE 4-27: Behavioral Model for 4 × 4 Binary Multiplier

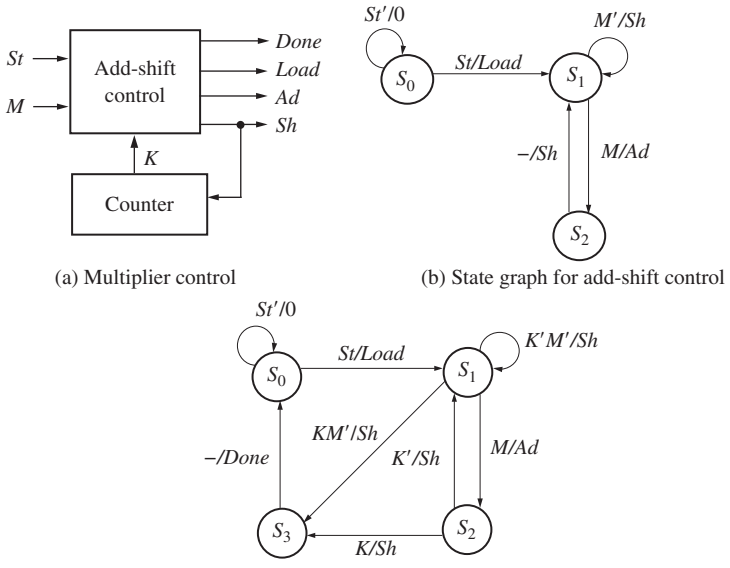
```
// This is a behavioral model of a multiplier for unsigned
// binary numbers.  It multiplies a 4-bit multiplicand
// by a 4-bit multiplier to give an 8-bit product.

// The maximum number of clock cycles needed for a
// multiply is 10.

`define M ACC[0]

module mult4X4 (Clk, St, Mplier, Mcand, Done, Result);

   input Clk;
   input St;
   input[3:0] Mplier;
   input[3:0] Mcand;
   output Done;
   output[7:0] Result;

   reg[3:0] State;
   reg[8:0] ACC;

   initial
   begin
      State = 0;
      ACC   = 0;
   end

   always @(posedge Clk)
   begin
         case (State)
            0 :
                     begin
                        if (St ==1'b1)
                        begin
                           ACC[8:4] <= 5'b00000 ;
                           ACC[3:0] <= Mplier ;
                           State <= 1 ;
                        end
                     end
            1, 3, 5, 7 :
                     begin
                      if (`M == 1'b1)
                      begin
                         ACC[8:4] <= {1'b0, ACC[7:4]} + Mcand ;
                           State <= State + 1 ;
                        end
                        else
                        begin
                          ACC <= {1'b0, ACC[8:1]} ;
```

```
                        State <= State + 2 ;
                    end
                end
            2, 4, 6, 8 :
                begin
                    ACC <= {1'b0, ACC[8:1]} ;
                    State <= State + 1 ;
                end
            9 :
                begin
                    State <= 0 ;
                end
        endcase
    end

    assign Done = (State == 9) ? 1'b1 : 1'b0 ;
    assign Result = (State == 9) ? ACC[7:0] : 8'b01010101 ;

endmodule
```

The *Done* signal should be turned on only in state 9. If we had used the statement `State <= 0; Done <= '1';` for the behavior of State = 9, *Done* would be turned on at the same time *State* changes to 0. This is too late, since we want *Done* to turn on when *State* becomes 9. Therefore, we used a separate concurrent assignment statement. This statement is placed outside the process so that *Done* will be updated whenever *State* changes.

As the state graph for the multiplier indicates, the control performs two functions—generating add or shift signals as needed and counting the number of shifts. If the number of bits is large, it is convenient to divide the control circuit into a counter and an add-shift control, as shown in Figure 4-28(a). First, we will derive a state graph for the add-shift control that tests *St* and *M* and outputs the proper sequence of add and shift signals (Figure 4-28(b)). Then we will add a completion signal (*K*) from the counter that stops the multiplier after the proper number of shifts have been completed. Starting in S_0 in Figure 4-28(b), when a start signal $St = 1$ is received, a load signal is generated and the circuit goes to state S_1. Then if $M = 1$, an add signal is generated and the circuit goes to state S_2; if $M = 0$, a shift signal is generated and the circuit stays in S_1. In S_2, a shift signal is generated since a shift always follows an add. The graph of Figure 4-28(b) will generate the proper sequence of add and shift signals, but it has no provision for stopping the multiplier.

To determine when the multiplication is completed, the counter is incremented each time a shift signal is generated. If the multiplier is *n* bits, *n* shifts are required. We will design the counter so that a completion signal (*K*) is generated after $n - 1$ shifts have occurred. When $K = 1$, the circuit should perform one more addition, if necessary, and then do the final shift. The control operation in Figure 4-28(c) is the same as Figure 4-28(b) as long as $K = 0$. In state S_1, if $K = 1$, we test *M* as usual. If $M = 0$, we output the final shift signal and go to the done state (S_3); however, if $M = 1$, we add before shifting and go to state S_2. In state S_2, if $K = 1$, we output one



(a) Multiplier control

(b) State graph for add-shift control

(c) Final state graph for add-shift control

FIGURE 4-28: Multiplier Control with Counter

more shift signal and then go to S_3. The last shift signal will increment the counter to 0 at the same time the add-shift control goes to the done state.

As an example, consider the multiplier of Figure 4-25, but replace the control circuit with Figure 4-28(a). Since $n = 4$, a 2-bit counter is needed to count the four shifts, and $K = 1$ when the counter is in state 3 (11_2). Table 4-2 shows the operation of the multiplier when 1101 is multiplied by 1011. S_0, S_1, S_2, and S_3 represent states of the control circuit (Figure 4-28(c)). The contents of the product register at each step are the same as given on page 233 of this chapter.

TABLE 4-2: Operation of Multiplier Using a Counter

Time	State	Counter	Product Register	St	M	K	Load	Ad	Sh	Done
t_0	S_0	00	000000000	0	0	0	0	0	0	0
t_1	S_0	00	000000000	1	0	0	1	0	0	0
t_2	S_1	00	000001011	0	1	0	0	1	0	0
t_3	S_2	00	011011011	0	1	0	0	0	1	0
t_4	S_1	01	001101101	0	1	0	0	1	0	0
t_5	S_2	01	100111101	0	1	0	0	0	1	0
t_6	S_1	10	010011110	0	0	0	0	0	1	0
t_7	S_1	11	001001111	0	1	1	0	1	0	0
t_8	S_2	11	100011111	0	1	1	0	0	1	0
t_9	S_3	00	010001111	0	1	0	0	0	0	1

At time t_0, the control is reset and waits for a start signal. At time t_1, the start signal *St* is 1, and a *Load* signal is generated. At time t_2, $M = 1$, so an *Ad* signal is generated. When the next clock occurs, the output of the adder is loaded into the

accumulator and the control goes to S_2. At t_3, an *Sh* signal is generated, so at the next clock shifting occurs and the counter is incremented. At t_4, $M = 1$ so $Ad = 1$, and the adder output is loaded into the accumulator at the next clock. At t_5 and t_6, shifting and counting occur. At t_7, three shifts have occurred and the counter state is 11, so $K = 1$. Since $M = 1$, addition occurs and control goes to S_2. At t_8, $Sh = K = 1$, so at the next clock the final shift occurs and the counter is incremented back to state 00. At t_9, a *Done* signal is generated.

The multiplier design given here can easily be expanded to 8, 16, or more bits simply by increasing the register size and the number of bits in the counter. The add-shift control would remain unchanged.

4.9 Array Multiplier

An array multiplier is a parallel multiplier that generates the partial products in a parallel fashion. The various partial products are added as soon as they are available. Consider the process of multiplication as illustrated in Table 4-3. Two 4-bit unsigned numbers, $X_3X_2X_1X_0$ and $Y_3Y_2Y_1Y_0$, are multiplied to generate a product that is possibly 8 bits. Each of the X_iY_j product bits can be generated by an AND gate. Each partial product can be added to the previous sum of partial products using a row of adders. The sum output of the first row of adders, which adds the first two partial products, is $S_{13}S_{12}S_{11}S_{10}$, and the carry output is $C_{13}C_{12}C_{11}C_{10}$. Similar results occur for the other two rows of adders. (We have used the notation S_{ij} and C_{ij} to represent the sums and carries from the ith row of adders.)

TABLE 4-3: 4-Bit Multiplier Partial Products

			X_3	X_2	X_1	X_0		Multiplicand
			Y_3	Y_2	Y_1	Y_0		Multiplier
			X_3Y_0	X_2Y_0	X_1Y_0	X_0Y_0		partial product 0
		X_3Y_1	X_2Y_1	X_1Y_1	X_0Y_1			partial product 1
		C_{12}	C_{11}	C_{10}				1st row carries
	C_{13}	S_{13}	S_{12}	S_{11}	S_{10}			1st row sums
	X_3Y_2	X_2Y_2	X_1Y_2	X_0Y_2				partial product 2
	C_{22}	C_{21}	C_{20}					2nd row carries
C_{23}	S_{23}	S_{22}	S_{21}	S_{20}				2nd row sums
X_3Y_3	X_2Y_3	X_1Y_3	X_0Y_3					partial product 3
C_{32}	C_{31}	C_{30}						3rd row carries
C_{33}	S_{33}	S_{32}	S_{31}	S_{30}				3rd row sums
P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0	final product

Figure 4-29 shows the array of AND gates and adders to perform this multiplication. If an adder has three inputs, a full adder (FA) is used, but if an adder has only two inputs, a half-adder (HA) is used. A half-adder is the same as a full adder with one of the inputs set to 0. This multiplier requires 16 AND gates, 8 full adders,

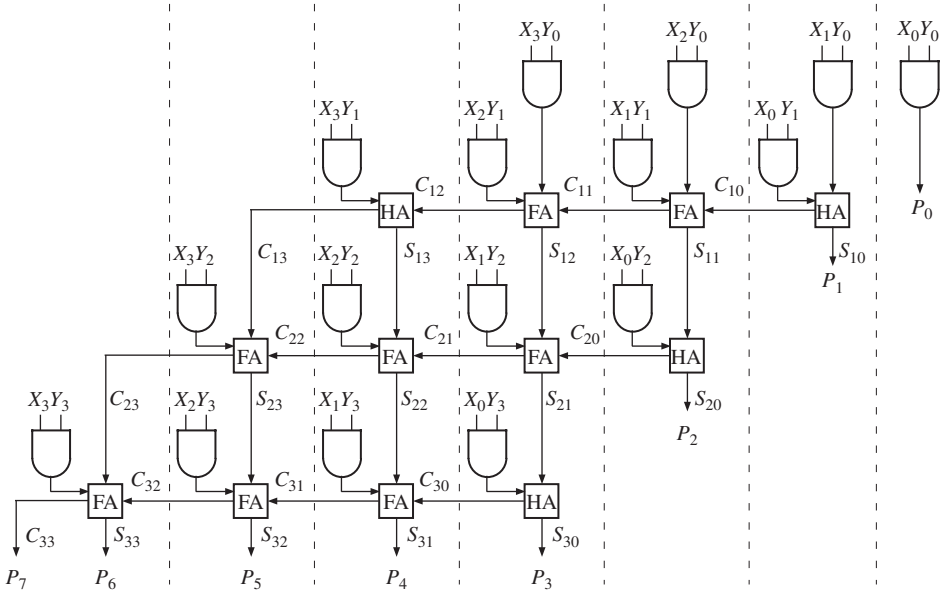


FIGURE 4-29: Block Diagram of 4 × 4 Array Multiplier

and 4 half-adders. After the *X* and *Y* inputs have been applied, the carry must propagate along each row of cells, and the sum must propagate from row to row. The time required to complete the multiplication depends primarily on the propagation delay in the adders. The longest path from input to output goes through 8 adders. If t_{ad} is the worst-case (longest possible) delay through an adder, and t_g is the longest AND gate delay, then the worst-case time to complete the multiplication is $8t_{ad} + t_g$.

In general, an *n*-bit-by-*n*-bit array multiplier would require n^2 AND gates, $n(n - 2)$ full adders, and n half-adders. So the number of components required increases quadratically. For the serial-parallel multiplier previously designed, the amount of hardware required in addition to the control circuit increases linearly with *n*.

For an $n \times n$ array multiplier, the longest path from input to output goes through *n* adders in the top row, *n-1* adders in the bottom row and *n-3* adders in the middle rows. The corresponding worst-case multiply time is $(3n - 4)t_{ad} + t_g$. The longest delay in a circuit is called a critical path. The worst-case delay can be improved to $2nt_{ad} + t_g$ by forwarding carry from each adder to the diagonally lower adder rather than the adder on the left side. When $n = 4$, both expressions are the same; however, for larger values of *n*, it is beneficial to pass carry diagonally as opposed to rippling it to the left. One may note that this multiplier has no sequential logic or registers.

The shift-and-add multiplier that we previously designed requires $2n$ clocks to complete the multiply in the worst case, although this can be reduced to n clocks using a technique discussed in the following section. The minimum clock period depends on the propagation delay through the *n*-bit adder as well as the propagation delay and setup time for the accumulator flip-flops.

Verilog Coding

If the topology has to be exactly what the designer wants, one needs to do structural coding. If one made a behavioral model of a multiplier without specifying the

topology, the topology generated by the synthesizer will depend on the synthesis tool. Here, we present a structural model for an array multiplier in Figure 4-30. Full-adder and half-adder modules are created and used as components for the array multiplier. The full adders and half adders are interconnected according to the array multiplier topology. Several instantiation statements are used for this purpose.

FIGURE 4-30: Verilog Code for 4 × 4 Array Multiplier

```
module Array_Mult (X, Y, P);

   input[3:0] X;
   input[3:0] Y;
   output[7:0] P;

   wire[3:0] C1;
   wire[3:0] C2;
   wire[3:0] C3;
   wire[3:0] S1;
   wire[3:0] S2;
   wire[3:0] S3;
   wire[3:0] XY0;
   wire[3:0] XY1;
   wire[3:0] XY2;
   wire[3:0] XY3;

   assign XY0[0] = X[0] & Y[0] ;
   assign XY1[0] = X[0] & Y[1] ;
   assign XY0[1] = X[1] & Y[0] ;
   assign XY1[1] = X[1] & Y[1] ;
   assign XY0[2] = X[2] & Y[0] ;
   assign XY1[2] = X[2] & Y[1] ;
   assign XY0[3] = X[3] & Y[0] ;
   assign XY1[3] = X[3] & Y[1] ;
   assign XY2[0] = X[0] & Y[2] ;
   assign XY3[0] = X[0] & Y[3] ;
   assign XY2[1] = X[1] & Y[2] ;
   assign XY3[1] = X[1] & Y[3] ;
   assign XY2[2] = X[2] & Y[2] ;
   assign XY3[2] = X[2] & Y[3] ;
   assign XY2[3] = X[3] & Y[2] ;
   assign XY3[3] = X[3] & Y[3] ;

   FullAdder FA1 (XY0[2], XY1[1], C1[0], C1[1], S1[1]);
   FullAdder FA2 (XY0[3], XY1[2], C1[1], C1[2], S1[2]);
   FullAdder FA3 (S1[2], XY2[1], C2[0], C2[1], S2[1]);
   FullAdder FA4 (S1[3], XY2[2], C2[1], C2[2], S2[2]);
   FullAdder FA5 (C1[3], XY2[3], C2[2], C2[3], S2[3]);
   FullAdder FA6 (S2[2], XY3[1], C3[0], C3[1], S3[1]);
   FullAdder FA7 (S2[3], XY3[2], C3[1], C3[2], S3[2]);
   FullAdder FA8 (C2[3], XY3[3], C3[2], C3[3], S3[3]);
   HalfAdder HA1 (XY0[1], XY1[0], C1[0], S1[0]);
```

```
    HalfAdder HA2 (XY1[3], C1[2], C1[3], S1[3]);
    HalfAdder HA3 (S1[1], XY2[0], C2[0], S2[0]);
    HalfAdder HA4 (S2[1], XY3[0], C3[0], S3[0]);

    assign P[0] = XY0[0] ;
    assign P[1] = S1[0] ;
    assign P[2] = S2[0] ;
    assign P[3] = S3[0] ;
    assign P[4] = S3[1] ;
    assign P[5] = S3[2] ;
    assign P[6] = S3[3] ;
    assign P[7] = C3[3] ;
endmodule

// Full Adder and half adder modules
// should be in the project

module FullAdder (X, Y, Cin, Cout, Sum);

    input X;
    input Y;
    input Cin;
    output Cout;
    output Sum;

    assign Sum = X ^ Y ^ Cin ;
    assign Cout = (X & Y) | (X & Cin) | (Y & Cin) ;
endmodule

module HalfAdder (X, Y, Cout, Sum);

    input X;
    input Y;
    output Cout;
    output Sum;

    assign Sum = X ^ Y ;
    assign Cout = X & Y ;
endmodule
```

4.10 A Signed Integer/Fraction Multiplier

Several algorithms are available for multiplication of signed binary numbers. The following procedure is a straightforward way to carry out the multiplication:

1. Complement the multiplier if negative.
2. Complement the multiplicand if negative.
3. Multiply the two positive binary numbers.
4. Complement the product if it should be negative.

Although this method is conceptually simple, it requires more hardware and computation time than some of the other available methods.

The next method we describe requires only the ability to complement the multiplicand. Complementation of the multiplier or product is not necessary. Although the method works equally well with integers or fractions, we illustrate the method with fractions, since we will later use this multiplier as part of a multiplier for floating-point numbers. Using 2's complement for negative numbers, we will represent signed binary fractions in the following form:

$$0.101 \qquad +5/8 \qquad 1.011 \qquad -5/8$$

The digit to the left of the binary point is the sign bit, which is 0 for positive fractions and 1 for negative fractions. In general, the 2's complement of a binary fraction F is $F^* = 2 - F$. Thus, $-5/8$ is represented by $10.000 - 0.101 = 1.011$. (This method of defining 2's complement fractions is consistent with the integer case ($N^* = 2^n - N$), since moving the binary point $n - 1$ places to the left is equivalent to dividing by 2^{n-1}.) The 2's complement of a fraction can be found by starting at the right end and complementing all the digits to the left of the first 1, the same as for the integer case. The 2's complement fraction 1.000 . . . is a special case. It actually represents the number -1, since the sign bit is negative and the 2's complement of 1.000. . . is $2 - 1 = 1$. We cannot represent $+1$ in this 2's complement fraction system, since 0.111 . . . is the largest positive fraction.

Binary Fixed Point Fractions

Fixed point numbers are number formats in which the decimal or binary point is at a fixed location. One can have a fixed-point 8-bit number format where the binary point is assumed to be after 4 bits (i.e., 4 bits for the fractional part and 4 bits for the integral part). If the binary point is assumed to be located 2 more bits to the right, there will be 6 bits for the integral part and 2 bits for the fraction. The range and precision of the numbers that can be represented in the different formats depend on the location of the binary point. For instance, if there are 4 bits for the fractional part and 4 bits for the integer, the range, assuming unsigned numbers, is 0.00 to 15.925. If only 2 bits are allowed for the fractional part and 6 bits for the integer, the range increases but the precision reduces. Now, the range would be 0.00 to 63.75, but the fractional part can be specified only as a multiple of 0.25.

Let us say we need to represent -13.45 in a 2's complement fixed-point number representation with 4 fractional bits. To convert any decimal fraction into the binary fraction, one technique is to repeatedly multiply the fractional part (only the fractional part in each intermediate step) by 2. So, starting with 0.45, the repeated multiplication results in

0.90

1.80

1.60

1.20
0.40
0.80
1.60
1.20

Now, the binary representation can be obtained by considering the digits in bold. An appropriate representation can be obtained depending on the number of bits available (e.g., 0111 if 4 bits are available, 01110011 if 8 bits are available, and so on). The representation for decimal number 13.45 in the fixed point format with 4 binary places will be as follows:

13.45: 1101.0111

One may note that the represented number is only an approximation of the actual number. The represented number can be converted back to decimal and seen to be 13.4375 (slightly off from the number we started with). The representation approaches the actual number as more and more binary places are added to the representation.

Negative fractions can be represented in 2's complement form. Let us represent −13.45 in 2's complement form. This cannot be done if we have only four places for the integer. We need to have at least 5 bits for the integer in order to handle the sign. Assuming 5 bits are available for the integer, in a 9 bit format:

13.45:	01101.0111
1's complement	10010.1000
2's complement	10010.1001

Hence, −13.45 = 10010.1001 in this representation.

When multiplying signed binary numbers, we must consider four cases:

Multiplicand	Multiplier
+	+
−	+
+	−
−	−

When both the multiplicand and the multiplier are positive, standard binary multiplication is used. For example,

0.1 1 1	(+7/8)	←	Multiplicand
× 0.1 0 1	(+5/8)	←	Multiplier
(0. 0 0)0 1 1 1	(+7/64)	←	*Note*: The proper representation of the
(0.)0 1 1 1	(+7/16)	←	fractional partial products requires extension
0. 1 0 0 0 1 1	(+35/64)		of the sign bit past the binary point, as indicated in parentheses. (Such extension is not necessary in the hardware.)

When the multiplicand is negative and the multiplier is positive, the procedure is the same as in the previous case, except that we must extend the sign bit of the multiplicand so that the partial products and final product will have the proper negative sign. For example,

1.1 0 1	(−3/8)		
× 0.1 0 1	(+5/8)		
(1. 1 1)1 1 0 1	(−3/64)	←	*Note*: The extension of the
(1.)1 1 0 1	(−3/16)	←	sign bit provides proper representation of the
1. 1 1 0 0 0 1	(−15/64)		negative products.

When the multiplier is negative and the multiplicand is positive, we must make a slight change in the multiplication procedure. A negative fraction of the form 1.*g* has a numeric value $-1 + 0.g$; for example, $1.011 = -1 + 0.011 = -(1 - 0.011) = -0.101 = -5/8$. Thus, when multiplying by a negative fraction of the form 1.*g*, we treat the fraction part (.*g*) as a positive fraction, but the sign bit is treated as −1. Hence, multiplication proceeds in the normal way as we multiply by each bit of the fraction and accumulate the partial products. However, when we reach the negative sign bit, we must add in the 2's complement of the multiplicand instead of the multiplicand itself. The following example illustrates this:

0.1 0 1	(+5/8)		
× 1.1 0 1	(−3/8)		
(0.0 0)0 1 0 1	(+5/64)		
(0.)0 1 0 1	(+5/16)		
(0.)0 1 1 0 0 1			
1. 0 1 1	(−5/8)	←	*Note*: The 2's complement of
1. 1 1 0 0 0 1	(−15/64)		the multiplicand is added at this point.

When both the multiplicand and the multiplier are negative, the procedure is the same as before. At each step, we must be careful to extend the sign bit of the partial product to preserve the proper negative sign, and at the final step we must add in the 2's complement of the multiplicand, since the sign bit of the multiplier is negative. For example,

1.1 0 1	(−3/8)		
× 1.1 0 1	(−3/8)		
(1. 1 1) 1 1 0 1	(−3/64)	←	*Note*: Extend sign bit
(1.)1 1 0 1	(−3/16)		
1. 1 1 0 0 0 1			
0. 0 1 1	(+3/8)	←	Add the 2's complement of the multiplicand.
0. 0 0 1 0 0 1	(+9/64)		

In summary, the procedure for multiplying signed 2's complement binary fractions is the same as for multiplying positive binary fractions, except that we must be careful to preserve the sign of the partial product at each step, and if the sign of the multiplier is negative, we must complement the multiplicand before adding it in at the last step. The hardware is almost identical to that used for multiplication of positive numbers, except a complementer must be added for the multiplicand.

Figure 4-31 shows the hardware required to multiply two 4-bit fractions (including the sign bit). A 5-bit adder is used so the sign of the sum is not lost due to a carry into the sign bit position. The *M* input to the control circuit is the currently active bit of the multiplier. Control signal *Sh* causes the accumulator to shift right one place with sign extension. *Ad* causes the ADDER output to be loaded into the left 5 bits of the accumulator. The carry out from the last bit of the adder is discarded, since we are doing 2's complement addition. *Cm* causes the multiplicand (Mcand) to be complemented (1's complement) before it enters the adder inputs. *Cm* is also connected to the carry input of the adder so that when $Cm = 1$, the adder adds 1 plus the 1's complement of Mcand to the accumulator, which is equivalent to adding the

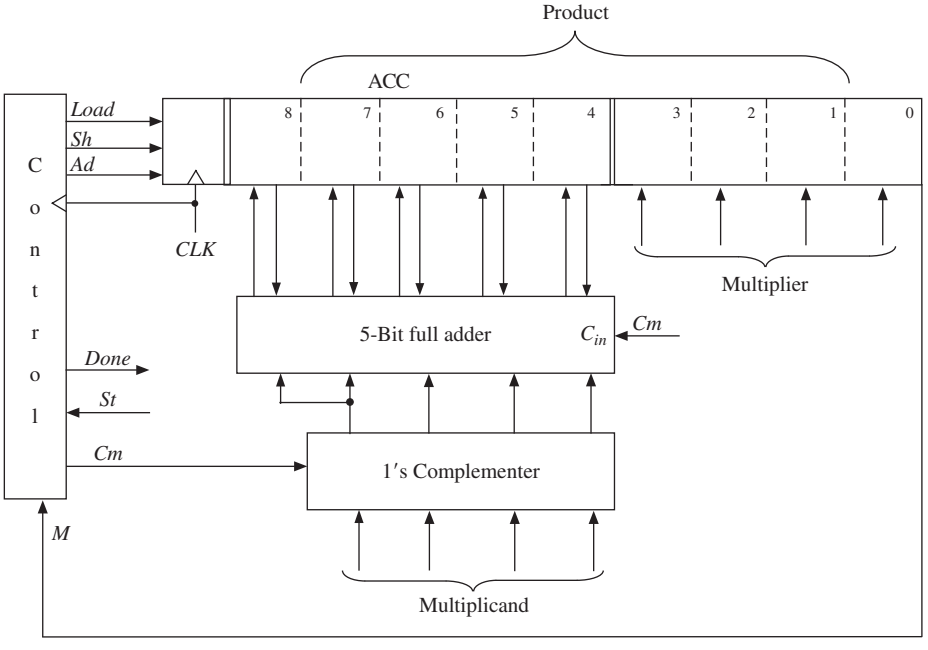


FIGURE 4-31: Block Diagram for 2's Complement Multiplier

2's complement of Mcand. Figure 4-32 shows a state graph for the control circuit. Each multiplier bit (M) is tested to determine whether to add and shift or whether to just shift. In state S_7, M is the sign bit, and if $M = 1$, the complement of the multiplicand is added to the accumulator.

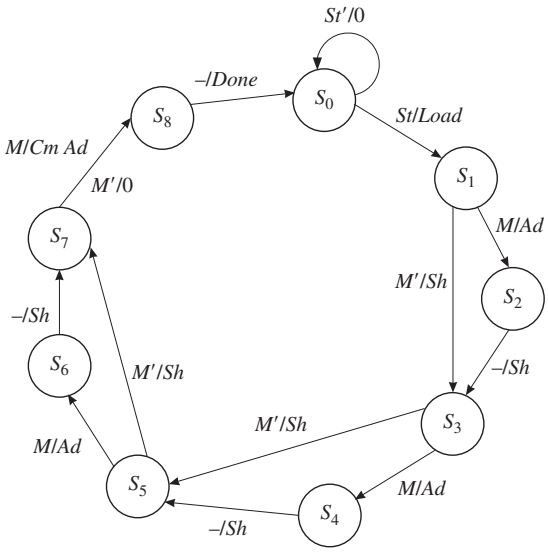


FIGURE 4-32: State Graph for 2's Complement Multiplier

When the hardware in Figure 4-31 is used, the add and shift operations must be done at two separate clock times. We can speed up the operation of the multiplier by moving the wires from the adder output one position to the right (Figure 4-33) so that the adder output is already shifted over one position when it is loaded into the accumulator. With this arrangement, the add and shift operations can occur at the same clock time, which leads to the control state graph of Figure 4-34. When the multiplication is complete, the product (6 bits plus sign) is in the lower 3 bits

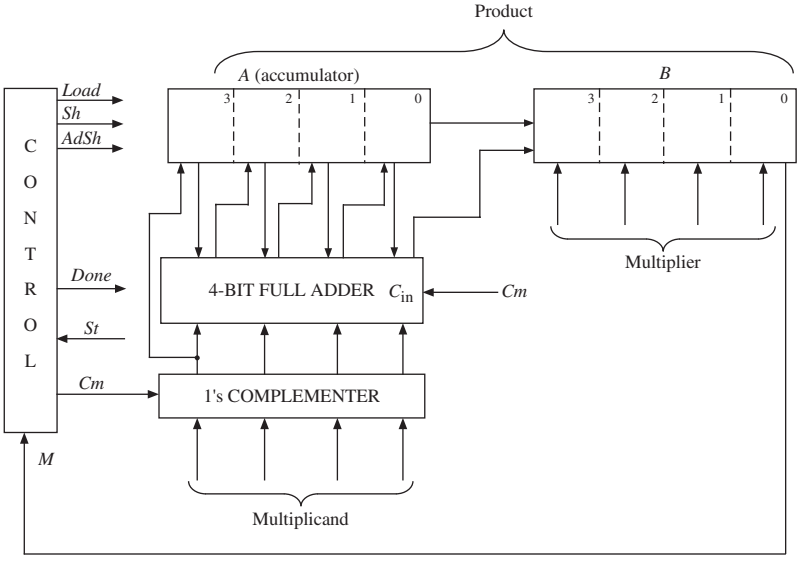


FIGURE 4-33: Block Diagram for Faster Multiplier

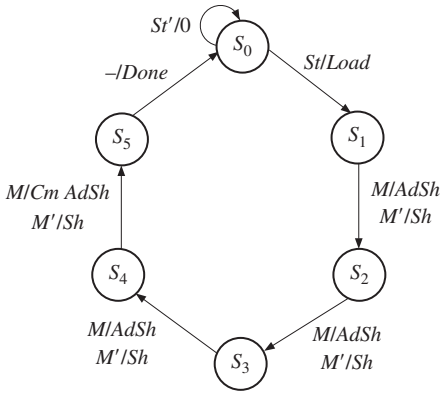


FIGURE 4-34: State Graph for Faster Multiplier

of *A* followed by *B*. The binary point then is in the middle of the *A* register. If we wanted it between the left two bits, we would have to shift *A* and *B* left one place.

A behavioral Verilog model for this multiplier is shown in Figure 4-35. Shifting the *A* and *B* registers together is accomplished by the sequential statements

```
A <= {A[3], A[3:1]} ;
B <= {A[0], B[3:1]} ;
```

FIGURE 4-35: Behavioral Model for 2's Complement Multiplier

```
`define M B[0]

module mult2C (CLK, St, Mplier, Mcand, Product, Done);

   input CLK;
   input St;
   input[3:0] Mplier;
   input[3:0] Mcand;
   output[6:0] Product;
   output Done;

   reg[2:0] State;
   reg[3:0] A;
   reg[3:0] B;
   reg[3:0] addout;

   initial
   begin
      State = 0;
   end

   always @(posedge CLK)
   begin
         case (State)
            0 :
                     begin
                        if (St == 1'b1)
                        begin
                           A <= 4'b0000 ;
                           B <= Mplier ;
```

```
                        State <= 1 ;
                    end
                    else
                        State <= 0;
                end
            1, 2, 3 :
                begin
                    if (`M == 1'b1)
                    begin
                        addout = A + Mcand;
                        A <= {Mcand[3], addout[3:1]} ;
                        B <= {addout[0], B[3:1]} ;
                    end
                    else
                    begin
                        A <= {A[3], A[3:1]} ;
                        B <= {A[0], B[3:1]} ;
                    end
                    State <= State + 1 ;
                end
            4 :
                begin
                    if (`M == 1'b1)
                    begin
                        addout = A + ~Mcand + 1;
                        A <= {~Mcand[3], addout[3:1]} ;
                        B <= {addout[0], B[3:1]} ;
                    end
                    else
                    begin
                        A <= {A[3], A[3:1]} ;
                        B <= {A[0], B[3:1]} ;
                    end
                    State <= 5 ;
                end
            5 :
                begin
                  State <= 0 ;
                end

        default :
                begin
                    State <= 0 ;
                end

        endcase
        end

    assign Done = (State == 5) ? 1'b1 : 1'b0 ;
    assign Product = {A[2:0], B} ;

endmodule
```

Although these statements are executed sequentially, *A* and *B* are both scheduled to be updated at the same delta time. Therefore, the old value of *A[0]* is used when computing the new value of *B*.

A register *addout* has been defined to represent the 5-bit output of the adder. In states 1 through 4, if the current multiplier bit *M* is 1, then the sign bit of the multiplicand followed by 3 bits of *addout* are loaded into *A*. At the same time, the low-order bit of *addout* is loaded into *B* along with the high-order 3 bits of *B*. The *Done* signal is turned on when control goes to state 5, and then the new value of the product is outputted.

Before continuing with the design, we will test the behavioral level Verilog code to make sure that the algorithm is correct and consistent with the hardware block diagram. At early stages of testing, we will want a step-by-step printout to verify the internal operations of the multiplier and to aid in debugging, if required. When we think that the multiplier is functioning properly, then we will only want to look at the final product's output so that we can quickly test a large number of cases.

Figure 4-36 shows the command file and test results for multiplying +5/8 by −3/8. A clock is defined with a 20-ns period. The *St* signal is turned on at 2 ns and turned off one clock period later. By inspection of the state graph, the multiplication requires six clocks, so the run time is set at 120 ns.

FIGURE 4-36: Command File and Simulation Results for (+5/8 by −3/8)

```
// command file to test signed multiplier
add list CLK St State A B Done Product
force St 1 2, 0 22
force CLK 1 0, 0 10 – repeat 20
// (5/8 * -3/8)
force Mcand  0101
force Mplier 1101
run 120
```

ns	delta	CLK	St	State	A	B	Done	Product
0	+1	1	0	0	0000	0000	0	0000000
2	+0	1	1	0	0000	0000	0	0000000
10	+0	0	1	0	0000	0000	0	0000000
20	+1	1	1	1	0000	1101	0	0000000
22	+0	1	0	1	0000	1101	0	0000000
30	+0	0	0	1	0000	1101	0	0000000
40	+1	1	0	2	0010	1110	0	0000000
50	+0	0	0	2	0010	1110	0	0000000
60	+1	1	0	3	0001	0111	0	0000000
70	+0	0	0	3	0001	0111	0	0000000
80	+1	1	0	4	0011	0011	0	0000000
90	+0	0	0	4	0011	0011	0	0000000
100	+2	1	0	5	1111	0001	1	1110001
110	+0	0	0	5	1111	0001	1	1110001
120	+1	1	0	0	1111	0001	0	1110001

To thoroughly test the multiplier, we need to test not only the four standard cases (+ +, + −, − +, and − −) but also special cases and limiting cases. Test values for the multiplicand and multiplier should include 0, the largest positive fraction, the most negative fraction, and all 1s. We will write a Verilog test bench to test the multiplier. The **test bench** will provide a sequence of values for the multiplicand and the multiplier. Thus, it provides stimuli to the system under test, the multiplier. The test bench can also check for the correctness of the multiplier output. The multiplier we are testing will be treated as a component and embedded in the test bench program. The signals generated within the test bench are interfaced to the multiplier as shown in Figure 4-37.

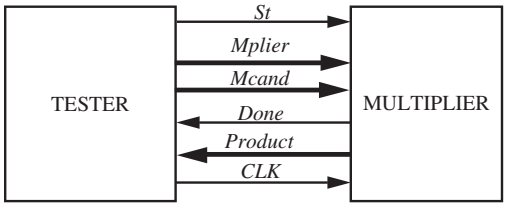


FIGURE 4-37: Interface between Multiplier and Its Test Bench

Figure 4-38 shows the Verilog code for the multiplier test bench. The test sequence consists of 11 sets of multiplicands and multipliers, provided in the *Mcandarr* and *Mplierarr* arrays. The expected outputs from the multiplier

FIGURE 4-38: Test Bench for Signed Multiplier

```
module testmult ();

   parameter N = 11;
   reg[3:0] Mcandarr[1:N];
   reg[3:0] Mplierarr[1:N];
   reg[6:0] Productarr[1:N];
   reg CLK;
   reg St;
   wire Done;
   reg[3:0] Mplier;
   reg[3:0] Mcand;
   wire[6:0] Product;
   integer i;

   initial
   begin
      CLK = 1'b1;
      Mcandarr[1] = 4'b0111;
      Mcandarr[2] = 4'b1101;
      Mcandarr[3] = 4'b0101;
      Mcandarr[4] = 4'b1101;
      Mcandarr[5] = 4'b0111;
      Mcandarr[6] = 4'b1000;
```

```
        Mcandarr[7] = 4'b0111;
        Mcandarr[8] = 4'b1000;
        Mcandarr[9] = 4'b0000;
        Mcandarr[10] = 4'b1111;
        Mcandarr[11] = 4'b1011;
        Mplierarr[1] = 4'b0101;
        Mplierarr[2] = 4'b0101;
        Mplierarr[3] = 4'b1101;
        Mplierarr[4] = 4'b1101;
        Mplierarr[5] = 4'b0111;
        Mplierarr[6] = 4'b0111;
        Mplierarr[7] = 4'b1000;
        Mplierarr[8] = 4'b1000;
        Mplierarr[9] = 4'b1101;
        Mplierarr[10] = 4'b1111;
        Mplierarr[11] = 4'b0000;
        Productarr[1] = 7'b0100011;
        Productarr[2] = 7'b1110001;
        Productarr[3] = 7'b1110001;
        Productarr[4] = 7'b0001001;
        Productarr[5] = 7'b0110001;
        Productarr[6] = 7'b1001000;
        Productarr[7] = 7'b1001000;
        Productarr[8] = 7'b1000000;
        Productarr[9] = 7'b0000000;
        Productarr[10] = 7'b0000001;
        Productarr[11] = 7'b0000000;
    end

    always
    begin
        #10 CLK <= ~CLK ;
    end

    always @(posedge CLK)
    begin
        for(i = 1; i <= N; i = i + 1)
        begin
            Mcand <= Mcandarr[i] ;
            Mplier <= Mplierarr[i] ;
            St <= 1'b1 ;
            @(posedge CLK);
            St <= 1'b0 ;
            @(negedge Done);
            if (~(Product == Productarr[i])) //compare with expected answer
                $display("Incorrect Product (error)");
        end
        $display("TEST COMPLETED (ERROR)");
    end

    mult2C mult1 (CLK, St, Mplier, Mcand, Product, Done);

endmodule
```

are provided in another array, the *Productarr*, in order to test the correctness of the multiplier outputs. The test values and results are placed in constant arrays in the Verilog code. The multiplier is instantiated and all signals are mapped with the test sequences.

```
mult2C mult1 (CLK, St, Mplier, Mcand, Product, Done);
```

The tester also generates the clock and start signal. The for loop reads values from the *Mcandarr* and *Mplierarr* arrays and then sets the start signal to 1. After the next clock, the start signal is turned off. Then, the test bench waits for the *Done* signal. When the trailing edge of *Done* arrives, the multiplier output is compared against the expected output in the array *Productarr.* An error is reported if the answers do not match. Since the *Done* signal is turned off at the same time the multiplier control goes back to S_0, the process waits for the falling edge of *Done* before looping back to supply new values of *Mcand* and *Mplier*. One may note that the **multiplier instatiation** is outside the always statement which generates the stimulus. The multiplier constantly receives some set of inputs and generates the corresponding set of outputs.

Figure 4-39 shows the command file and simulator output. We have annotated the simulator output to interpret the test results. The −NOtrigger together with the −Trigger done in the list statement causes the output to be displayed only when the *Done* signal changes. Without the −NOtrigger and −Trigger, the output would be displayed every time any signal on the list changed. All the product outputs are correct, except for the special case of −1 × −1 (1.000 × 1.000), which gives 1.000000 (−1) instead of +1. This occurs because no representation of +1 is possible without adding another bit.

FIGURE 4-39: Command File and Simulation of Signed Multiplier

```
// Command file to test results of signed multiplier
add list -NOtrigger Mplier Mcand Product -Trigger Done
run 1320

  ns   delta  Mplier Mcand  Product Done
   0      +0    xxxx  xxxx xxxxxxx    x
   0      +2    0101  0111 xxxxxxx    0
 100      +2    0101  0111 0100011    1  5/8 * 7/8 = 35/64
 120      +2    0101  1101 0100011    0
 220      +2    0101  1101 1110001    1  5/8 * -3/8 = -15/64
 240      +2    1101  0101 1110001    0
 340      +2    1101  0101 1110001    1  -3/8 * 5/8 = -15/64
 360      +2    1101  1101 1110001    0
 460      +2    1101  1101 0001001    1  -3/8 * -3/8 = 9/64
 480      +2    0111  0111 0001001    0
 580      +2    0111  0111 0110001    1  7/8 * 7/8 = 49/64
 600      +2    0111  1000 0110001    0
 700      +2    0111  1000 1001000    1  7/8 * -1 = -7/8
 720      +2    1000  0111 1001000    0
```

```
 820      +2    1000   0111 1001000     1   -1 * 7/8 = -7/8
 840      +2    1000   1000 1001000     0
 940      +2    1000   1000 1000000     1   -1 * -1 = -1 (error)
 960      +2    1101   0000 1000000     0
1060      +2    1101   0000 0000000     1   -3/8 * 0 = 0
1080      +2    1111   1111 0000000     0
1180      +2    1111   1111 0000001     1   -1/8 * -1/8 = 1/64
1200      +2    0000   1011  0000001    0
1300      +2    0000   1011  0000000    1   0 * -3/8 = 0
1320      +2    0000   1011  0000000    0
```

Next, we refine the Verilog model for the signed multiplier by explicitly defining the control signals and the actions that occur when each control signal is asserted. The Verilog code (Figure 4-40) is organized in a manner similar to the Mealy machine model of Figure 1-17. In the first process, the *Nextstate* and output control signals are defined for each present *State*. In the second process, after waiting for the rising edge of the clock, the appropriate registers are updated and the *State* is updated. We can test the Verilog code of Figure 4-40 using the same test file we used previously and verify that we get the same product outputs.

FIGURE 4-40: Model for 2's Complement Multiplier with Control Signals

```
`define M B[0]

// This Verilog model explicitly defines control signals.
module mult2C2 (CLK, St, Mplier, Mcand, Product, Done);

   input CLK;
   input St;
   input[3:0] Mplier;
   input[3:0] Mcand;
   output[6:0] Product;
   output Done;

   reg Done;
   reg[2:0] State;
   reg[2:0] Nextstate;
   reg[3:0] A;
   reg[3:0] B;
   wire[3:0] compout;
   wire[3:0] addout;
   reg AdSh;
   reg Sh;
   reg Load;
   reg Cm;

   always @(State or St or `M)
   begin
      Load = 1'b0 ;
```

```
        AdSh = 1'b0 ;
        Sh = 1'b0 ;
        Cm = 1'b0 ;
        Done = 1'b0 ;
        Nextstate = 1'b0;
        case (State)
           0 :
                     begin
                        if (St == 1'b1)
                        begin
                           Load = 1'b1 ;
                           Nextstate = 1 ;
                        end
                        else
                        begin
                           Load = 1'b0 ;
                           Nextstate = 0 ;
                        end
                     end
           1, 2, 3 :
                     begin
                        if (`M == 1'b1)
                        begin
                          AdSh = 1'b1 ;
                        end
                        else
                        begin
                           Sh = 1'b1 ;
                        end
                       Nextstate = State + 1 ;
                     end
           4 :
                     begin
                        if (`M == 1'b1)
                        begin
                           Cm = 1'b1 ;
                           AdSh = 1'b1 ;
                        end
                        else
                        begin
                           Sh = 1'b1 ;
                        end
                        Nextstate = 5 ;
                     end
           5 :
                     begin
                        Done = 1'b1 ;
                        Nextstate = 0 ;
                     end
        default:
```

```
                    begin
                        Done = 1'b0 ;
                        Nextstate = 0 ;
                    end

        endcase
    end

    assign compout = (Cm == 1'b1) ? ~Mcand : Mcand ;
    assign addout = A + compout + Cm ;

    always @(posedge CLK)
    begin
        if (Load == 1'b1)
        begin
            A <= 4'b0000 ;
            B <= Mplier ;
        end
        if (AdSh == 1'b1)
        begin
            A <= {compout[3], addout[3:1]} ;
            B <= {addout[0], B[3:1]} ;
        end
        if (Sh == 1'b1)
        begin
            A <= {A[3], A[3:1]} ;
            B <= {A[0], B[3:1]} ;
        end
        State <= Nextstate ;
    end

    assign Product = {A[2:0], B} ;
endmodule
```

4.11 Keypad Scanner

In this example, we design a scanner for a keypad with three columns and four rows as in Figure 4-41. The keypad is wired in matrix form with a switch at the intersection of each row and column. Pressing a key establishes a connection between a row and column. The purpose of the scanner is to determine which key has been pressed and to output a binary number $N = N_3N_2N_1N_0$, which corresponds to the key number. For example, pressing key 5 must output 0101, pressing the * key must output 1010, and pressing the # key must output 1011. When a valid key has been detected, the scanner should output a signal V for one clock time. Assume that only one key

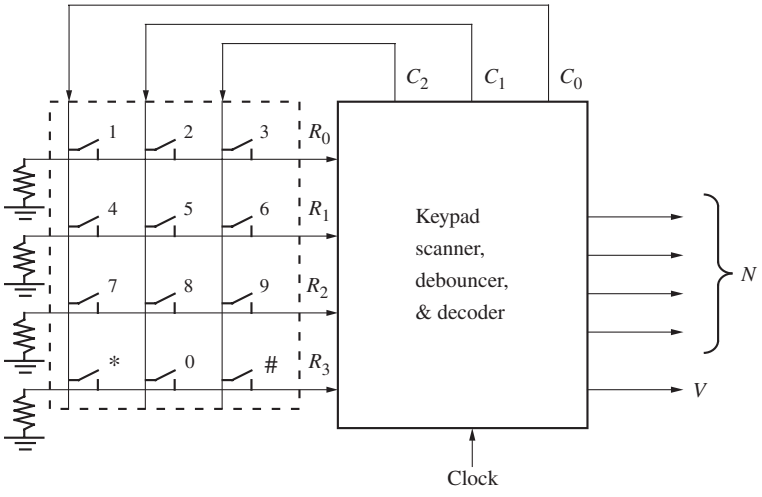
FIGURE 4-41: Keypad with Three Columns and Four Rows

1	2	3
4	5	6
7	8	9
*	0	#

is pressed at a time. The design must include hardware to protect the circuitry from malfunction due to keypad bounces.

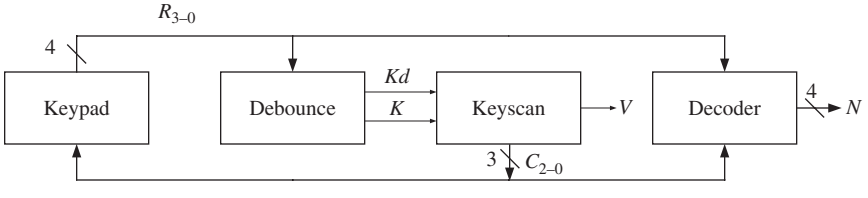
The overall block diagram of the circuit is presented in Figure 4-42. The keypad contains resistors that are connected to ground. When a switch is pressed, a path is established from the corresponding column line to the ground. If a voltage can be applied on the column lines C_0, C_1, and C_2, then the voltage can be obtained on the row line corresponding to the key that is pressed. One among the rows R_0, R_1, R_2, or R_3 will have an active signal.

FIGURE 4-42: Block Diagram for Keypad Scanner



We will divide the design into several modules, as shown in Figure 4-43. The first part of the design will be a scanner that scans the rows and columns of the keypad. The keyscan module generates the column signals to scan the keyboard. The debounce module generates a signal *K* when a key has been pressed and a signal *Kd* after it has been debounced. When a valid key is detected, the decoder determines the key number from the row and column numbers.

FIGURE 4-43: Scanner Modules



Scanner

We will use the following procedure to scan the keyboard: First apply logic 1*s* to columns C_0, C_1, and C_2 and wait. If any key is pressed, a 1 will appear on R_0, R_1, R_2, or R_3. Then apply a 1 to column C_0 only. If any of the R_is is 1, a valid key is detected. If R_0 is received, one knows that switch 1 was pressed. If R_1, R_2, or R_3 is received, it indicates switch 4, 7, or * was pressed. If so, set $V = 1$ and output the corresponding N. If no key is detected in the first column, apply a 1 to C_1 and repeat. If no key is detected in the second column, repeat for C_2. When a valid key is detected, apply 1_s to C_0, C_1, and C_2 and wait until no key is pressed. This last step is necessary so that only one valid signal is generated each time a key is pressed.

Debouncer

As discussed in the scoreboard example, we need to debounce the keys to avoid malfunctions due to switch bounce. Figure 4-44 shows a proposed debouncing and synchronizing circuit. The four row signals are connected to an OR gate to form signal *K,* which turns on when a key is pressed and a column scan signal is applied. The debounced signal *Kd* will be fed to the sequential circuit.

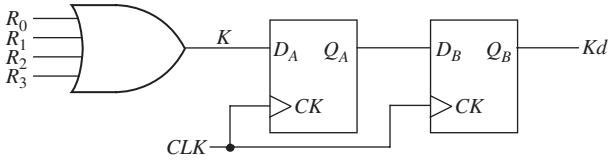


FIGURE 4-44: Debouncing and Synchronizing Circuit

Decoder

The decoder determines the key number from the row and column numbers using the truth table given in Table 4-4. The truth table has one row for each of the 12 keys. The remaining rows have don't care outputs since we have assumed that only one key is pressed at a time. Since the decoder is a combinational circuit, its output will

TABLE 4-4: Truth Table for Decoder

R_3	R_2	R_1	R_0	R_0	R_1	R_2	N_3	N_2	N_1	N_0	
0	0	0	1	1	0	0	0	0	0	1	
0	0	0	1	0	1	0	0	0	1	0	
0	0	0	1	0	0	1	0	0	1	1	
0	0	1	0	1	0	0	0	1	0	0	
0	0	1	0	0	1	0	0	1	0	1	
0	0	1	0	0	0	1	0	1	1	0	
0	1	0	0	1	0	0	0	1	1	1	
0	1	0	0	0	1	0	1	0	0	0	
0	1	0	0	0	0	1	1	0	0	1	
1	0	0	0	1	0	0	1	0	1	0	(*)
1	0	0	0	0	1	0	0	0	0	0	
1	0	0	0	0	0	1	1	0	1	1	(#)

Logic Equations for Decoder

$$N_3 = R_2C_0' + R_3C_1'$$

$$N_2 = R_1 + R_2C_0$$

$$N_1 = R_0C_0' + R_2'C_2 + R_1'R_0'C_0$$

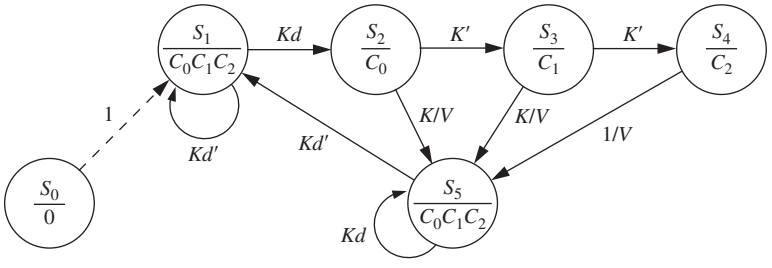
$$N_0 = R_1C_1 + R_1'C_2 + R_3'R_1'C_1'$$

change as the keypad is scanned. At the time a valid key is detected ($K = 1$ and $V = 1$), its output will have the correct value and this value can be saved in a register at the same time the circuit goes to S_5.

Controller

Figure 4-45 shows the state diagram of the controller for the keypad scanner. It waits in S_1 with outputs $C_0 = C_1 = C_2 = 1$ until a key is pressed. In S_2, $C_0 = 1$, so if the key that was pressed is in column 0, $K = 1$ and the circuit outputs a valid signal and goes to S_5. Signal K is used instead of Kd, since the key press is already debounced. If no key press is found in column 0, column 1 is checked in S_3, and if necessary, column 2 is checked in S_4. In S_5, the circuit waits until all keys are released and Kd goes to 0 before resetting.

FIGURE 4-45: State Graph for Keypad Scanner



The foregoing state diagram as presented works for many cases, but it does have some timing problems. Let us analyze the following situations.

1. **Is *K* true whenever a button is pressed?**
   No. Although K is true if any one of the row signals R_1, R_2, R_3, or R_4 is true, if the column scan signals are not active, none of R_1–R_4 can be true, although the button is pressed.
2. **Can *Kd* be false when a button is continuing to be pressed?**
   Yes. Signal Kd is nothing but K delayed by two clock cycles. K can go to 0 during the scan process even when the button is being pressed. For instance, consider the case when a key in the rightmost column is pressed. During scan of the first two columns, K goes to 0. If K goes to 0 at any time, Kd will go to zero two cycles later. Hence, neither K nor Kd is the same as pressing the button.
3. **Can you go from S_5 to S_1 when a button is still pressed?**
   In the state diagram in Figure 4-45, the S_4-to-S_5 transition could happen when Kd is false. Kd might have become false while scanning C_0 and C_1. Hence, it is possible that one reaches back to S_1 when the key is still being pressed. As an example, let us assume that a button is pressed in column C_2. This is to be detected in S_4. However, during the scanning process in S_2 and S_3, K is 0; hence, two cycles later Kd will be 0 even if the button stays pressed. During the scan in S_4, the correct key can be found; however, the system can reach S_5 when Kd is still 0 and a malfunction can happen. S_5 is intended to sense the release of the key. However, Kd is not synonymous to pressing the button and Kd' does not truly indicate that the button got released. Since Kd' can appear when the button is still pressed, if you reach S_5 when Kd' is true due to scanning activity in a previous state, the system

can go from S_5 to S_1 without a key release. In such a case, the same key may be read multiple times.

4. **What if a key is pressed for only one or two clock cycles?**
If the key is pressed and released very quickly, there would be problems especially if the key is in the third column. By the time the scanner reaches state S_4, the key might have been released already. The key should be pressed long enough for the scanner to go through the longest path in the state graph from S_0 to S_5. This may not be a serious problem, because usually, the digital system clock is much faster than any mechanical switch.

These problems can be fixed by assuring that one can reach S_5 only if Kd is true. A modified state diagram is presented in Figure 4-46. Before transitioning to state S_5, this circuit waits in state S_2, S_3, and S_4 until Kd also becomes 1.

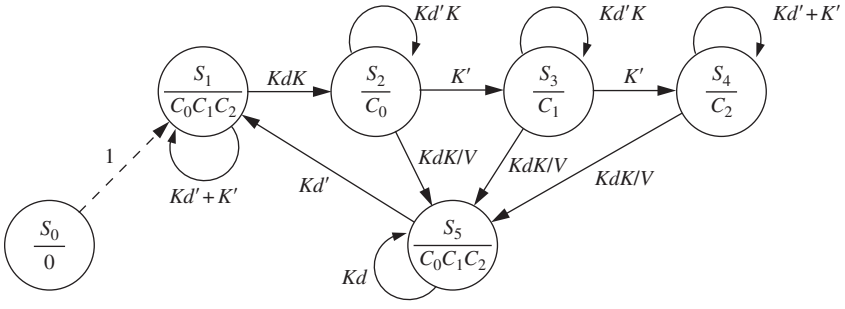


FIGURE 4-46: Modified State Graph for Keypad Scanner

Verilog Code

The Verilog code used to implement the design is shown in Figure 4-47. The decoder equations as well as the equations for K and V are implemented by concurrent statements. The process implements the next state equations for the keyscan and debounce flip-flops.

FIGURE 4-47: Verilog Code for Scanner

```
module scanner (R0, R1, R2, R3, CLK, C0, C1, C2, N0, N1, N2, N3, V);
    input R0;
    input R1;
    input R2;
    input R3;
    input CLK;
    inout C0;
    inout C1;
    inout C2;
    output N0;
    output N1;
    output N2;
    output N3;
    output V;

    reg V;
    reg C0_tmp, C1_tmp, C2_tmp;
```

```
reg QA;
wire K;
reg Kd;
reg[2:0] state;
reg[2:0] nextstate;

assign C0 = C0_tmp;
assign C1 = C1_tmp;
assign C2 = C2_tmp;

assign K = R0 | R1 | R2 | R3 ;
assign N3 = (R2 & ~C0) | (R3 & ~C1) ;
assign N2 = R1 | (R2 & C0) ;
assign N1 = (R0 & ~C0) | (~R2 & C2) | (~R1 & ~R0 & C0) ;
assign N0 = (R1 & C1) | (~R1 & C2) | (~R3 & ~R1 & ~C1) ;

initial
begin
   state = 0;
   nextstate = 0;
end

always @(state or R0 or R1 or R2 or R3 or C0 or C1 or C2 or K or Kd or QA)
begin
  C0_tmp = 1'b0 ;
  C1_tmp = 1'b0 ;
  C2_tmp = 1'b0 ;
  V = 1'b0 ;
  case (state)
      0 :
                begin
                   nextstate = 1 ;
                end
      1 :
                begin
                  C0_tmp = 1'b1 ;
                  C1_tmp = 1'b1 ;
                  C2_tmp = 1'b1 ;
                   if ((Kd & K) == 1'b1)
                   begin
                      nextstate = 2 ;
                   end
                   else
                   begin
                      nextstate = 1 ;
                   end
                end
     2 :
                begin
                  C0_tmp = 1'b1 ;
                   if ((Kd & K) == 1'b1)
                  begin
                      V = 1'b1 ;
```

```
                    nextstate = 5 ;
                end
                else if (K == 1'b0)
                begin
                    nextstate = 3 ;
                end
                else
                begin
                  nextstate = 2 ;
                end
            end
        3 :
            begin
                C1_tmp = 1'b1 ;
                if ((Kd & K) == 1'b1)
                begin
                    V = 1'b1 ;
                    nextstate = 5 ;
                end
                else if (K == 1'b0)
                begin
                    nextstate = 4 ;
                end
                else
                begin
                    nextstate = 3 ;
                end
            end
        4 :
            begin
                C2_tmp <= 1'b1 ;
                if ((Kd & K) == 1'b1)
                begin
                    V <= 1'b1 ;
                    nextstate = 5 ;
                end
                else
                begin
                    nextstate = 4 ;
                end
            end
        5 :
            begin
                C0_tmp = 1'b1 ;
                C1_tmp = 1'b1 ;
                C2_tmp = 1'b1 ;
                if (Kd == 1'b0)
                 begin
                    nextstate = 1 ;
                end
                else
```

```
                        begin
                           nextstate = 5 ;
                        end
                     end
     endcase
   end
   always @(posedge CLK)
   begin
         state <= nextstate ;
         QA <= K ;
         Kd <= QA ;
   end
endmodule
```

Test Bench for Keypad Scanner

This Verilog code would be very difficult to test by supplying waveforms for the inputs R_0, R_1, R_2, and R_3, since these inputs depend on the column outputs (C_0, C_1, C_2). A much better way to test the scanner is by using a test bench in Verilog. The scanner we are testing will be treated as a component and embedded in the test bench program. The signals generated within the test bench are interfaced to the scanner as shown in Figure 4-48. The test bench simulates a key press by supplying the appropriate R signals in response to the C signals from the scanner. When test bench receives $V = 1$ from the scanner, it checks to see whether the value of N corresponds to the key that was pressed.

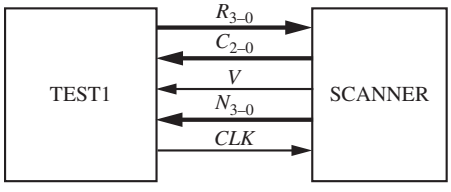


FIGURE 4-48: Interface for Test Bench

The Verilog code for the keypad test bench is shown in Figure 4-49. A copy of the scanner is instantiated within the TEST1 module, and connections to the scanner are made by the input/output mapping. The sequence of key numbers used for testing is stored in the array *KARRAY*. The tester simulates the keypad operation using concurrent statements for R_0, R_1, R_2, and R_3. Whenever C_0, C_1, C_2, or the key number (KN) changes, new values for the Rs are computed. For example, if $KN = 5$ (to simulate pressing key 5), then $R_0R_1R_2R_3 = 0100$ is sent to the scanner when $C_0C_1C_2 = 010$. The test process is as follows:

1. Read a key number from the array to simulate pressing a key.
2. Wait until $V = 1$ and the rising edge of the clock occurs.
3. Verify that the N output from the scanner matches the key number.
4. Set $KN = 15$ to simulate no key pressed. (Since 15 is not a valid key number, all Rs will go to 0.)
5. Wait until $Kd = 0$ before selecting a new key.

FIGURE 4-49: Verilog for Scanner Test Bench

```
module scantest ();

    integer KARRAY[0:23];
    wire C0;
    wire C1;
    wire C2;
    wire V;
    reg CLK;
    wire R0;
    wire R1;
    wire R2;
    wire R3;
    wire[3:0] N;
    integer KN;
    integer i;

    initial
    begin
        CLK = 0;
       KARRAY[0] = 2;
       KARRAY[1] = 5;
       KARRAY[2] = 8;
       KARRAY[3] = 0;
       KARRAY[4] = 3;
       KARRAY[5] = 6;
       KARRAY[6] = 9;
       KARRAY[7] = 11;
       KARRAY[8] = 1;
       KARRAY[9] = 4;
       KARRAY[10] = 7;
       KARRAY[11] = 10;
       KARRAY[12] = 1;
       KARRAY[13] = 2;
       KARRAY[14] = 3;
       KARRAY[15] = 4;
       KARRAY[16] = 5;
       KARRAY[17] = 6;
       KARRAY[18] = 7;
       KARRAY[19] = 8;
       KARRAY[20] = 9;
       KARRAY[21] = 10;
       KARRAY[22] = 11;
       KARRAY[23] = 0;
    end

    always #20 CLK = ~CLK ;
    assign R0 = ((C0 == 1'b1 & KN == 1) | (C1 == 1'b1 & KN == 2)
                | (C2 == 1'b1 & KN == 3)) ? 1'b1 : 1'b0 ;
    assign R1 = ((C0 == 1'b1 & KN == 4) | (C1 == 1'b1 & KN == 5)
                | (C2 == 1'b1 & KN == 6)) ? 1'b1 : 1'b0 ;
```

```
    assign R2 = ((C0 == 1'b1 & KN == 7) | (C1 == 1'b1 & KN == 8)
                | (C2 == 1'b1 & KN == 9)) ? 1'b1 : 1'b0 ;
    assign R3 = ((C0 == 1'b1 & KN == 10) | (C1 == 1'b1 & KN == 0)
                | (C2 == 1'b1 & KN == 11)) ? 1'b1 : 1'b0 ;

    always @(posedge CLK)
    begin
       for(i = 0; i <= 23; i = i + 1)
       begin
          KN <= KARRAY[i] ;
          @(posedge CLK);
          if (V == 1'b1)
             if (~(N == KN)) $display("Numbers don't match.");

          KN <= 15 ;
          @(posedge CLK);
          @(posedge CLK);
          @(posedge CLK);
       end

       $display("Test Complete.");
    end

    scanner scanner1(R0, R1, R2, R3, CLK, C0, C1, C2, N[0],N[1],
    N[2],N[3],V);
endmodule
```

Key presses in row order and column order are tried using the various numbers in KARRAY. The test bench will test whether the reported number matches the key pressed. The `display` statement will report "Numbers don't match" if the scanner generates the wrong key number, and it will report "Testing Complete." when all keys have been tested.

4.12 Binary Dividers

Unsigned Divider

We will consider the design of a parallel divider for positive binary numbers. As an example, we will design a circuit to divide an 8-bit dividend by a 4-bit divisor to obtain a 4-bit quotient. The following example illustrates the division process:

```
                      1010      quotient
Divisor  1101 ) 10000111        dividend
                 1101
(135 ÷ 13 = 10 with  0111
  a remainder of 5)  0000
                     1111
                    1101
                     0101
                     0000
                     0101       Remainder
```

Just as binary multiplication can be carried out as a series of add and shift operations, division can be carried out by a series of subtract and shift operations. To construct the divider, we will use a 9-bit dividend register and a 4-bit divisor register, as shown in Figure 4-50. During the division process, instead of shifting the divisor right before each subtraction, we will shift the dividend to the left. Note that an extra bit is required on the left end of the dividend register so that a bit is not lost when the dividend is shifted left. Instead of using a separate register to store the quotient, we will enter the quotient bit by bit into the right end of the dividend register as the dividend is shifted left.

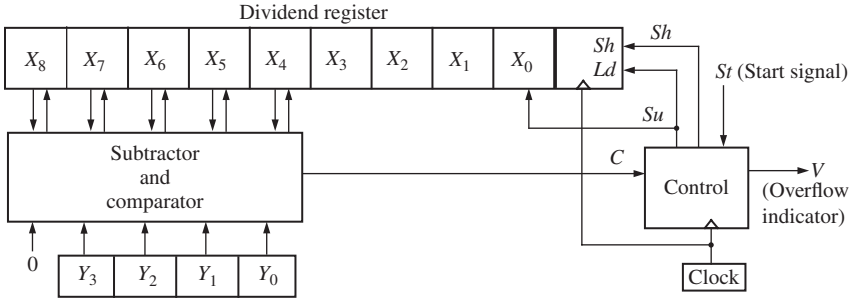


FIGURE 4-50: Block Diagram for Parallel Binary Divider

The preceding division example (135 divided by 13) is reworked next, showing the location of the bits in the registers at each clock time. Initially, the dividend and divisor are entered as follows:

0	1	0	0	0	0	1	1	1

1	1	0	1

Subtraction cannot be carried out without a negative result, so we will shift before we subtract. Instead of shifting the divisor one place to the right, we will shift the dividend one place to the left.

1 0 0 0 0 1 1 1 | 0 ← Dividing line between dividend and quotient

1 1 0 1 ← Note that after the shift, the right most position in the dividend register is "empty"

Subtraction is now carried out and the first quotient digit of 1 is stored in the unused position of the dividend register:

0 0 0 1 1 1 1 1 | 1 ← first quotient digit

Next we shift the dividend one place to the left.

0 0 1 1 1 1 1 | 1 0

1 1 0 1

Since subtraction would yield a negative result, we shift the dividend to the left again, and the second quotient bit remains zero.

0 1 1 1 1 1 | 1 0 0

1 1 0 1

Subtraction is now carried out, and the third quotient digit of 1 is stored in the unused position of the dividend register.

0 0 0 1 0 1 | 1 0 1 ← third quotient digit

A final shift is carried out and the fourth quotient bit is set to 0.

$$\underbrace{0\quad 0\quad 1\quad 0\quad 1}_{\text{remainder}}\ \vdots\ \underbrace{1\quad 0\quad 1\quad 0}_{\text{quotient}}$$

The final result agrees with that obtained in the first example.

If, as a result of a division operation, the quotient contains more bits than are available for storing the quotient, we say that an *overflow* has occurred. For the divider of Figure 4-50, an overflow would occur if the quotient is greater than 15, since only 4 bits are provided to store the quotient. It is not actually necessary to carry out the division to determine whether an overflow condition exists, since an initial comparison of the dividend and divisor will tell if the quotient will be too large. For example, if we attempt to divide 135 by 7, the initial contents of the registers are

$$\begin{array}{ccccccccc} 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ & 0 & 1 & 1 & 1 & & & & \end{array}$$

Since subtraction can be carried out with a nonnegative result, we should subtract the divisor from the dividend and enter a quotient bit of 1 in the rightmost place in the dividend register. However, we cannot do this because the rightmost place contains the least significant bit of the dividend, and entering a quotient bit here would destroy that dividend bit. Therefore, the quotient would be too large to store in the 4 bits we have allocated for it, and we have detected an overflow condition. In general, for Figure 4-50, if initially $X_8X_7X_6X_5X_4 \geq Y_3Y_2Y_1Y_0$ (i.e., if the left 5 bits of the dividend register exceed or equal the divisor), the quotient will be greater than 15 and an overflow occurs. Note that if $X_8X_7X_6X_5X_4 \geq Y_3Y_2Y_1Y_0$, the quotient is

$$\frac{X_8X_7X_6X_5X_4X_3X_2X_1X_0}{Y_3Y_2Y_1Y_0} \geq \frac{X_8X_7X_6X_5X_40000}{Y_3Y_2Y_1Y_0} = \frac{X_8X_7X_6X_5X_4 \times 16}{Y_3Y_2Y_1Y_0} \geq 16$$

The operation of the divider can be explained in terms of the block diagram of Figure 4-50. A shift signal (*Sh*) will shift the dividend one place to the left. A subtract signal (*Su*) will subtract the divisor from the 5 leftmost bits in the dividend register and set the quotient bit (the rightmost bit in the dividend register) to 1. If the divisor is greater than the 5 leftmost dividend bits, the comparator output is $C = 0$; otherwise, $C = 1$. Whenever $C = 0$, subtraction cannot occur without a negative result, so a shift signal is generated. Whenever $C = 1$, a subtract signal is generated, and the quotient bit is set to 1. The control circuit generates the required sequence of shift and subtract signals.

Figure 4-51 shows the state diagram for the control circuit. When a start signal (*St*) occurs, the 8-bit dividend and 4-bit divisor are loaded into the appropriate registers. If *C* is 1, the quotient would require 5 or more bits. Since space is provided only for a 4-bit quotient, this condition constitutes an overflow, so the divider is stopped and the overflow indicator is set by the *V* output. Normally, the initial value of *C* is 0, so a shift will occur first and the control circuit will go to state S_2. Then, if $C = 1$, subtraction occurs. After the subtraction is completed, *C* will always be 0, so the next clock pulse will produce a shift. This process continues until four shifts

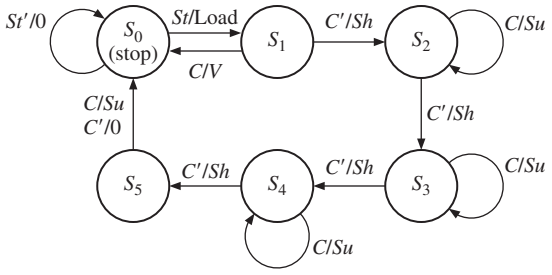


FIGURE 4-51: State Diagram for Divider Control Circuit

have occurred and the control is in state S_5. Then a final subtraction occurs, if necessary, and the control returns to the stop state. For this example, we will assume that when the start signal (*St*) occurs, it will be 1 for one clock time and then it will remain 0 until the control circuit is back in state S_0. Therefore, *St* will always be 0 in states S_1 through S_5.

Table 4-5 gives the state table for the control circuit. Since we assumed that $St = 0$ in states S_1, S_2, S_3, and S_4, the next states and outputs are "don't cares" for these states when $St = 1$. The entries in the output table indicate which outputs are 1. For example, the entry *Sh* means $Sh = 1$ and the other outputs are 0.

TABLE 4-5: State Table for Divider Control Circuit

	StC				StC			
State	00	01	11	10	00	01	11	10
S_0	S_0	S_0	S_1	S_1	0	0	Load	Load
S_1	S_2	S_0	–	–	*Sh*	*V*	–	–
S_2	S_3	S_2	–	–	*Sh*	*Su*	–	–
S_3	S_4	S_3	–	–	*Sh*	*Su*	–	–
S_4	S_5	S_4	–	–	*Sh*	*Su*	–	–
S_5	S_0	S_0	–	–	0	*Su*	–	–

This example illustrates a general method for designing a divider for unsigned binary numbers, and the design can easily be extended to larger numbers such as 16 bits divided by 8 bits or 32 bits divided by 16 bits. Using a separate counter to count the number of shifts is recommended if more than four shifts are required.

Signed Divider

We now design a divider for signed (2's complement) binary numbers that divides a 32-bit dividend by a 16-bit divisor to give a 16-bit quotient. Although algorithms exist to divide the signed numbers directly, such algorithms are rather complex. So we take the easy way out and complement the dividend and divisor if they are negative; when division is complete, we complement the quotient if it should be negative.

Figure 4-52 shows a block diagram for the divider. We use a 16-bit bus to load the registers. Since the dividend is 32 bits, two clocks are required to load the upper and lower halves of the dividend register, and one clock is needed to load

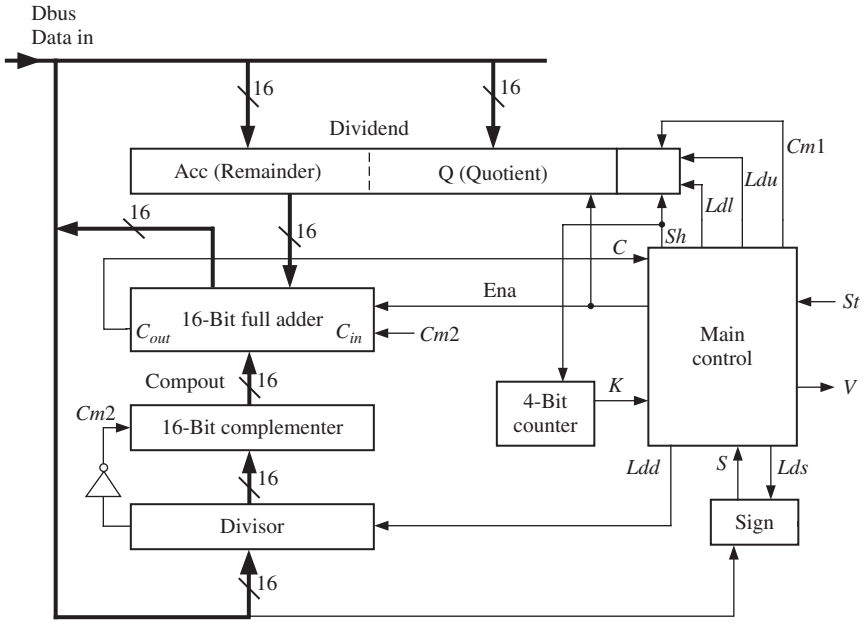


FIGURE 4-52: Block Diagram for Signed Divider

the divisor. An extra sign flip-flop is used to store the sign of the dividend. We will use a dividend register with a built-in 2's complementer. The subtractor consists of an adder and a complementer, so subtraction can be accomplished by adding the 2's complement of the divisor to the dividend register. If the divisor is negative, using a separate step to complement it is unnecessary; we can simply disable the complementer and add the negative divisor instead of subtracting its complement. The control circuit is divided into two parts—a main control, which determines the sequence of shifts and subtracts, and a counter, which counts the number of shifts. The counter outputs a signal $K = 1$ when 15 shifts have occurred. Control signals are defined as follows:

LdU	Load upper half of dividend from bus.
LdL	Load lower half of dividend from bus.
Lds	Load sign of dividend into sign flip-flop.
S	Sign of dividend.
Cm1	Complement dividend register (2's complement).
Ldd	Load divisor from bus.
Su	Enable adder output onto bus (*Ena*), and load upper half of dividend from bus.
Cm2	Enable complementer. (*Cm2* equals the complement of the sign bit of the divisor, so a positive divisor is complemented and a negative divisor is not.)
Sh	Shift the dividend register left one place and increment the counter.

C	Carry output from adder. (If $C = 1$, the divisor can be subtracted from the upper dividend.)
St	Start.
V	Overflow.
Qneg	Quotient will be negative. ($Qneg = 1$ when the sign of the dividend and divisor are different.)

The procedure for carrying out the signed division is as follows:

1. Load the upper half of the dividend from the bus, and copy the sign of the dividend into the sign flip-flop.
2. Load the lower half of the dividend from the bus.
3. Load the divisor from the bus.
4. Complement the dividend if it is negative.
5. If an overflow condition is present, go to the done state.
6. Else carry out the division by a series of shifts and subtracts.
7. When division is complete, complement the quotient if necessary and go to the done state.

Testing for overflow is slightly more complicated than for the case of unsigned division. First, consider the case of all positive numbers. Since the divisor and quotient are each 15 bits plus sign, their maximum value is 7FFFh. Since the remainder must be less than the divisor, its maximum value is 7FFEh. Therefore, the maximum dividend for no overflow is

$$\text{divisor} \times \text{quotient} + \text{remainder} = 7FFFh \times 7FFFh + 7FFEh = 3FFF7FFFh$$

If the dividend is 1 larger (3FFF8000h), division by 7FFFh (or anything smaller) will give an overflow. We can test for the overflow condition by shifting the dividend left one place and then comparing the upper half of the dividend (divu) with the divisor. If divu $\geq$ divisor, the quotient would be greater than the maximum value, which is an overflow condition. For the preceding example, shifting 3FFF8000h left once gives 7FFF0000h. Since 7FFFh equals the divisor, there is an overflow. On the other hand, shifting 3FFF7FFFh left gives 7FFEFFFEh, and since 7FFEh $<$ 7FFFh, no overflow occurs when dividing by 7FFFh.

Another way of verifying that we must shift the dividend left before testing for overflow is as follows. If we shift the dividend left one place and then divu $\geq$ divisor, we could subtract and generate a quotient bit of 1. However, this bit would have to go in the sign bit position of the quotient. This would make the quotient negative, which is incorrect. After testing for overflow, we must shift the dividend left again, which gives a place to store the first quotient bit after the sign bit. Since we work with the complement of a negative dividend or a negative divisor, this method for detecting overflow will work for negative numbers, except for the special case where the dividend is 80000000h (the largest negative value). Modifying the design to detect overflow in this case is left as an exercise.

Figure 4-53 shows the state graph for the control circuit. When $St = 1$, the registers are loaded. In S_2, if the sign of the dividend (S) is 1, the dividend is

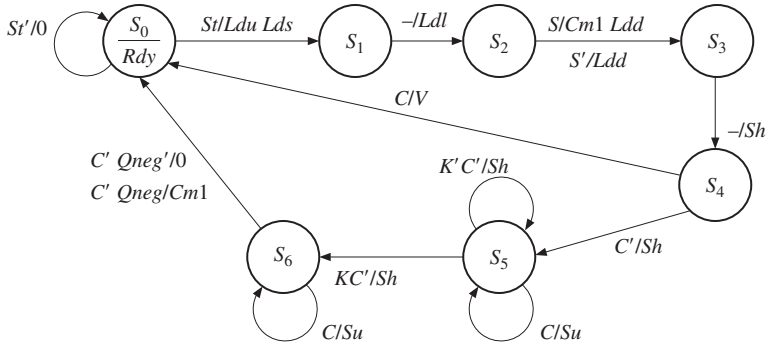


FIGURE 4-53: State Graph for Signed Divider Control Circuit

complemented. In S_3, we shift the dividend left one place and then we test for overflow in S_4. If $C = 1$, subtraction is possible, which implies an overflow, and the circuit goes to the done state. Otherwise, the dividend is shifted left. In S_5, C is tested. If $C = 1$, then $Su = 1$, which implies *Ldu* and *Ena*, so the adder output is enabled onto the bus and loaded into the upper dividend register to accomplish the subtraction. Otherwise, $Sh = 1$ and the dividend register is shifted. This continues until $K = 1$, at which time the last shift occurs if $C = 0$, and the circuit goes to S_6. Then if the sign of the divisor and the saved sign of the dividend are different, the dividend register is complemented so that the quotient will have the correct sign.

The Verilog code for the signed divider is shown in Figure 4-54. Since the 1's complementer and adder are combinational circuits, we have represented their operation by concurrent statements. All the signals that represent register outputs are updated on the rising edge of the clock, so these signals are updated in the process after waiting for *CLK* to change to 1. The counter is simulated by a signal, *count*. For convenience in listing the simulator output, we have added a ready signal (*Rdy*), which is turned on in S_0 to indicate that the division is completed.

FIGURE 4-54: Verilog Model of 32-Bit Signed Divider

```
`define Acc Dividend[31:16]

module sdiv (CLK, St, Dbus, Quotient, Remainder, V, Rdy);

   input CLK;
   input St;
   input[15:0] Dbus;
   output[15:0] Quotient;
   output[15:0] Remainder;
   output V;
   output Rdy;

   reg V;
```

```
reg[2:0] State;
reg[3:0] Count;
reg Sign;
wire C;
wire Cm2;
reg[15:0] Divisor;
wire[15:0] Sum;
wire[15:0] Compout;
reg[31:0] Dividend;

assign Cm2 = ~Divisor[15] ;
assign Compout = (Cm2 == 1'b0) ? Divisor : ~Divisor ;
assign Sum = `Acc + Compout + Cm2;
assign C = ~Sum[15] ;
assign Quotient = Dividend[15:0] ;
assign Remainder = Dividend[31:16];
assign Rdy = (State == 0) ? 1'b1 : 1'b0 ;

initial
begin
  State = 0;
end

always @(posedge CLK)
begin

      case (State)
         0 :
                   begin
                      if (St == 1'b1)
                      begin
                          `Acc <= Dbus ;
                          Sign <= Dbus[15] ;
                          State <= 1 ;
                          V <= 1'b0 ;
                          Count <= 4'b0000 ;
                      end
                  end
        1 :
                   begin
                      Dividend[15:0] <= Dbus ;
                      State <= 2 ;
                   end
         2 :
                   begin
                      Divisor <= Dbus ;
                      if (Sign == 1'b1)
                      begin
                         Dividend <= ~Dividend + 1 ;
                      end
                      State <= 3 ;
                   end
```

```
            3 :
                    begin
                        Dividend <= {Dividend[30:0], 1'b0} ;
                        Count <= Count + 1 ;
                        State <= 4 ;
                    end
             4 :
                    begin
                        if (C == 1'b1)
                        begin
                            V <= 1'b1 ;
                            State <= 0 ;
                        end
                        else
                        begin
                            Dividend <= {Dividend[30:0], 1'b0} ;
                            Count <= Count + 1 ;
                            State <= 5 ;
                        end
                    end
             5 :
                    begin
                        if (C == 1'b1)
                        begin
                            `Acc <= Sum ;
                            Dividend[0] <= 1'b1 ;
                        end
                        else
                        begin
                            Dividend <= {Dividend[30:0], 1'b0} ;
                            if (Count == 15)
                            begin
                                State <= 6 ;
                            end
                            Count <= Count + 1 ;
                        end
                    end
             6 :
                    begin
                        State <= 0 ;
                        if (C == 1'b1)
                        begin
                            `Acc <= Sum ;
                            Dividend[0] <= 1'b1 ;
                            State <= 6 ;
                        end
                        else if ((Sign ^ Divisor[15]) == 1'b1)
                        begin
                          Dividend[15:0] <= ~Dividend[15:0] + 1 ;
                          if(Sign == 1)
                          begin
```

```
                        Dividend [31:16] <= ~Dividend[31:16] + 1;
                        end
                      end
                      else begin
                        if(Sign && Divisor[15])
                        begin
                        Dividend [31:16] <= ~Dividend[31:16] + 1;
                        end
                    end
                  end
        endcase
        end

endmodule
```

We are now ready to test the divider design by using the Verilog simulator. We will need a comprehensive set of test examples that will test all the different special cases that can arise in the division process. To start with, we need to test the basic operation of the divider for all the different combinations of signs for the divisor and dividend (+ +, + −, − +, and − −). We also need to test the overflow detection for these four cases. Limiting cases must also be tested, including largest quotient, zero quotient, and so forth. Use of a Verilog test bench is convenient, because the test data must be supplied in sequence at certain times and the length of time to complete the division is dependent on the test data. Figure 4-55 shows a test bench for the divisor. The test bench contains a dividend array and a divisor array for the test data. The notation `32'h07FF00BB` is the hexadecimal representation of a bit string. The process in `testsdiv` first puts the upper dividend on *Dbus* and supplies a start signal. After waiting for the clock, it puts the lower dividend on *Dbus*. After the next clock, it puts the divisor on *Dbus*. It then waits until the *Rdy* signal indicates that division is complete before continuing. *Count* is set equal to the loop-index, so that the change in *Count* can be used to trigger the listing output.

FIGURE 4-55: Test Bench for Signed Divider

```
module testsdiv;
  parameter N = 12;
  reg[31:0] dividendarr[1:N];
  reg[15:0] divisorarr[1:N];

  //inputs to sdiv module should be reg types
  reg CLK;
  reg St;
  reg[15:0] Dbus;

  //outputs from sdiv module should be wire types
  wire[15:0] Quotient;
  wire[15:0] Remainder;
  wire V;
  wire Rdy;
```

```
reg[15:0] Divisor;
reg[31:0] Dividend;
reg[3:0] Count;
reg[31:0] dividendarr_tmp;
reg [15:0] quotientarr[1:N];
reg [15:0] remainderarr[1:N];

integer i;

always
begin
  #10 CLK <= ~CLK;
end

initial
begin
  //initialization of dividend array
  dividendarr[1] = 32'h0000006F;
  dividendarr[2] = 32'h07FF00BB;
  dividendarr[3] = 32'hFFFFFE08;
  dividendarr[4] = 32'hFF80030A;
  dividendarr[5] = 32'h3FFF8000;
  dividendarr[6] = 32'h3FFF7FFF;
  dividendarr[7] = 32'hC0008000;
  dividendarr[8] = 32'hC0008000;
  dividendarr[9] = 32'hC0008001;
  dividendarr[10] = 32'h00000000;
  dividendarr[11] = 32'hFFFFFFFF;
  dividendarr[12] = 32'hFFFFFFFF;

  //initialization of divisor array
  divisorarr[1] = 16'h0007;
  divisorarr[2] = 16'hE005;
  divisorarr[3] = 16'h001E;
  divisorarr[4] = 16'hEFFA;
  divisorarr[5] = 16'h7FFF;
  divisorarr[6] = 16'h7FFF;
  divisorarr[7] = 16'h7FFF;
  divisorarr[8] = 16'h8000;
  divisorarr[9] = 16'h7FFF;
  divisorarr[10] = 16'h0001;
  divisorarr[11] = 16'h7FFF;
  divisorarr[12] = 16'h0000;

  //initialization of quotient array
  quotientarr[1] = 16'h000F;
  quotientarr[2] = 16'hBFFE;
  quotientarr[3] = 16'hFFF0;
  quotientarr[4] = 16'h07FC;
  quotientarr[5] = 16'h0000;
  quotientarr[6] = 16'h7FFF;
  quotientarr[7] = 16'h0000;
```

```
    quotientarr[8] = 16'h7FFF;
    quotientarr[9] = 16'h8001;
    quotientarr[10] = 16'h0000;
    quotientarr[11] = 16'h0000;
    quotientarr[12] = 16'h0002;

    //initialization of remainder array
    remainderarr[1] = 16'h0006;
    remainderarr[2] = 16'h00C5;
    remainderarr[3] = 16'hFFE8;
    remainderarr[4] = 16'hF2F2;
    remainderarr[5] = 16'h7FFF;
    remainderarr[6] = 16'h7FFE;
    remainderarr[7] = 16'h7FFF;
    remainderarr[8] = 16'h0000;
    remainderarr[9] = 16'h8002;
    remainderarr[10] = 16'h0000;
    remainderarr[11] = 16'hFFFF;
    remainderarr[12] = 16'h0000;

    CLK = 0;
    Count = 0;

  @(posedge CLK);
  @(negedge CLK);

  for(i = 1 ; i <= N ; i = i + 1)
  begin
    St = 1'b1;
    dividendarr_tmp = dividendarr[i];
    Dbus = dividendarr_tmp[31:16];

    @(posedge CLK);
    Dbus = dividendarr_tmp[15:0];

    @(posedge CLK);
    Dbus = divisorarr[i];
    St = 1'b0;
    Dividend = dividendarr_tmp[31:0];
    Divisor = divisorarr[i];

    @(posedge Rdy);
    Count = i;
     if(quotientarr[i] == Quotient)
      begin
      $display("quotient[%d] is correct",i);
      end
     else
      begin
      $display("quotient[%d] is wrong",i);
      end
     if(remainderarr[i] == Remainder)
      begin
      $display("remainder[%d] is correct",i);
      end
```

```
    else
    begin
       $display("remainder[%d] is wrong",i);
    end
   end
   $display("TESTS DONE");
end

   sdiv sdiv1(CLK, St, Dbus, Quotient, Remainder, V, Rdy);
endmodule
```

Figure 4-56 shows the simulator command file and output. The `-notrigger`, together with the `-trigger count` in the list statement, causes the output to be displayed only when the *count* signal changes. Examination of the simulator output shows that the divider operation is correct for all of the test cases, except for the following case:

$$\text{C0008000h} \div \text{7FFFh} = -\text{3FFF8000} \div \text{7FFFh} = -\text{8000h} = \text{8000h}$$

In this case, the overflow is turned on and division never occurs. In general, the divider will indicate an overflow whenever the quotient should be 8000h (the most negative value). This occurs because the divider basically divides positive numbers and the largest positive quotient is 7FFFh. If it is important to be able to generate the quotient 8000h, the overflow detection can be modified so it does not generate an overflow in this special case.

FIGURE 4-56: Simulation Test Results for Signed Divider

```
// Command file to test results of signed divider
add list -hex -notrigger Dividend Divisor Quotient Remainder V -trigger Count
run 5300

  ns   delta  dividend  divisor  quotient remainder  v  count
   0      +0  xxxxxxxx     xxxx      xxxx      xxxx  x      x
 490      +3  0000006F     0007      000F      0006  0      1
 930      +3  07FF00BB     E005      BFFE      00C5  0      2
1350      +3  FFFFFE08     001E      FFF0      FFE8  0      3
1930      +3  FF80030A     EFFA      07FC      F2F2  0      4
2030      +3  3FFF8000     7FFF      0000      7FFF  1      5
2730      +3  3FFF7FFF     7FFF      7FFF      7FFE  0      6
2830      +3  C0008000     7FFF      0000      7FFF  1      7
3530      +3  C0008000     8000      7FFF      0000  0      8
4230      +3  C0008001     7FFF      8001      8002  0      9
4630      +3  00000000     0001      0000      0000  0      A
5030      +3  FFFFFFFF     7FFF      0000      FFFF  0      B
5130      +3  FFFFFFFF     0000      0002      0000  1      C
```

In this chapter, we have presented several design examples. The examples included several arithmetic and non-arithmetic circuits. A 7-segment display, a BCD adder, a traffic light controller, a scoreboard, and a keypad scanner are examples of non-arithmetic circuits presented in the chapter. We also described

algorithms for addition, multiplication, and division of unsigned and signed binary numbers. Specific designs such as the carry look-ahead adder and the array multiplier were presented. We designed digital systems to implement these algorithms. After developing a block diagram for such a system and defining the required control signals, we used state graphs to define a sequential machine that generates control signals in the proper sequence. We used Verilog to describe the systems at several different levels so that we can simulate and test for correct operation of the systems we have designed.

Problems

4.1 Design the correction circuit for a BCD adder that computes Zdigit 0 and C for S_0 (see Figures 4-5 and 4-6). This correction circuit adds 0110 to S_0 if $S_0 > 9$. This is the same as adding $0AA0$ to S_0, where $A = 1$ if $S_0 > 9$. Draw a block diagram for the correction circuit using one full adder, three half adders, and a logic circuit to compute A. Design a circuit for A using a minimum number of gates. Note that the maximum possible value of S_0 is 10010.

4.2 **(a)** If gate delays are 5 ns, what is the delay of the fastest 4-bit ripple carry adder? Explain your calculation.
(b) If gate delays are 5 ns, what is the delay of the fastest 4-bit adder? What kind of adder will it be? Explain your calculation.

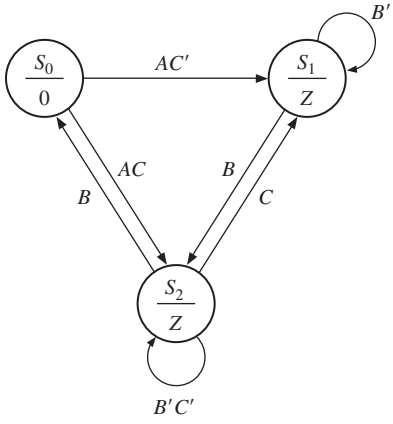
4.3 Develop a Verilog model for a 16-bit carry look-ahead adder utilizing the 4-bit adder from Figure 4-10 as a module.

4.4 Derive generates, propagates, group generates, group-propagates, and the final sum and carry out for the 16-bit carry look-ahead adder of Figure 4-9, while adding 0101 1010 1111 1000 and 0011 1100 1100 0011.

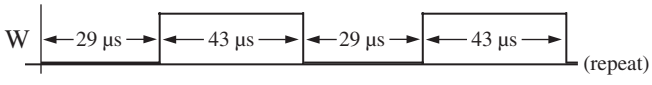
4.5 **(a)** Write a Verilog module that describes one bit of a full adder with accumulator. The module should have two control inputs, Ad and L. If $Ad = 1$, the Y input (and carry input) are added to the accumulator. If $L = 1$, the Y input is loaded into the accumulator.
(b) Using the module defined in **(a)**, write a Verilog description of a 4-bit subtracter with accumulator. Assume negative numbers are represented in 1's complement. The subtracter should have control inputs *Su* (subtract) and *Ld* (load).

4.6 **(a)** Implement the traffic-light controller of Figure 4-14 using a module 13 counter with added logic. The counter should increment every clock, with two exceptions. Use a ROM to generate the outputs.
(b) Write a Verilog description of your answer to **(a)**.
(c) Write a test bench for part **(b)** and verify that your controller works correctly. Use concurrent statements to generate test inputs for *Sa* and *Sb*.

4.7 Make the necessary additions to the following state graph so that it is a proper, completely specified state graph. Demonstrate that your answer is correct. Convert the graph to a state table using 0's and 1's for inputs and outputs.



4.8 Write synthesizable Verilog code that will generate the given waveform (*W*). Use a single always block. Assume that a clock with a 1 μs period is available as an input.



4.9 A BCD adder adds two BCD numbers (each of range 0 to 9) and produces the sum in BCD form. For example, if it adds 9 (1001) and 8 (1000) the result would be 17 (1 0111). Implement such a BCD adder using a 4-bit binary adder and appropriate control circuitry. Assume that the two BCD numbers are already loaded into two 4-bit registers (*A* and *B*) and there is a 5-bit sum register (*S*) available. You need some kind of correction to get the sum in the BCD form, because the binary adder produces results in the range 0000 to 1111 (plus a carry in some cases). If any addition is required for this correction, use the same adder (i.e., you can use only one adder). Use multiplexers at the adder inputs to steer the appropriate numbers to the adder in each cycle. Assume a start signal to initiate the addition and a done signal to indicate completion.

(a) Draw a block diagram of the system. Label each component appropriately to indicate its functionality and size.
(b) Describe step by step the algorithm that you would use to perform the addition. Explain and illustrate the correction step.
(c) Draw a state graph for the controller.

4.10 Write Verilog code for a shift register module that includes a 16-bit shift register, a controller, and a 4-bit down counter. The shifter can shift a variable number of bits depending on a count provided to the shifter module. Inputs to the module are a number *N* (indicating shift count) in the range 1 to 15, a 16-bit vector *par_in*, a clock,

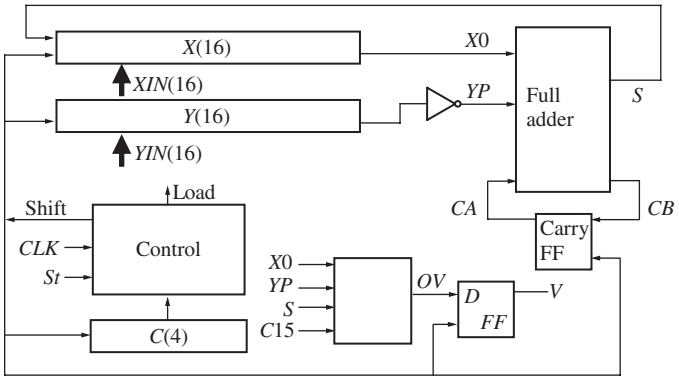
and a start signal, *St*. When *St* = 1, *N* is loaded into the down counter, and *par_in* is loaded into the shift register. Then the shift register does a cycle left shift *N* times, and the controller returns to the start state. Assume that *St* is only 1 for one clock time. All operations are synchronous on the falling edge of the clock.

(a) Draw a block diagram of the system and define any necessary control signals.
(b) Draw a state graph for the controller (two states).
(c) Write Verilog code for the shift-register module. Use two always blocks (one for the combinational part of the circuit and one for updating the registers).

4.11 **(a)** Figure 4-12 shows the block diagram for a 32-bit serial adder with accumulator. The control circuit uses a 5-bit counter, which outputs a signal *K* = 1 when it is in state 11111. When a start signal (*St*) is received, the registers should be loaded. Assume that *St* will remain 1 until the addition is complete. When the addition is complete, the control circuit should go to a stop state and remain there until *St* is changed back to 0. Draw a state diagram for the control circuit (excluding the counter).
(b) Write the Verilog for the complete system and verify its correct operation.

4.12 A block diagram for a 16-bit 2's complement serial subtracter is given here. When *St* = 1, the registers are loaded and then subtraction occurs. The shift counter, *C*, produces a signal *C15* = 1 after 15 shifts. *V* should be set to 1 if an overflow occurs. Set the carry flip-flop to 1 during load in order to form the 2's complement. Assume that *St* remains 1 for one clock time.

(a) Draw a state diagram for the control (two states).
(b) Write Verilog code for the system. Use two always blocks. The first always block should determine the next state and control signals; the second always block should update the registers on the rising edge of the clock.



4.13 This problem involves the design of a BCD-to-binary converter. Initially a 3-digit BCD number is placed in the *A* register. When an *St* signal is received, conversion to binary takes place and the resulting binary number is stored in the *B* register. At each step of the conversion, the entire BCD number (along with the binary number) is shifted one place to the right. If the result in a given decade is greater than or

equal to 1000, the correction circuit subtracts 0011 from that decade. (If the result is less than 1000, the correction circuit leaves the contents of the decade unchanged.) A shift counter is provided to count the number of shifts. When conversion is complete, the maximum value of B will be 999 (in binary). *Note*: B is 10 bits.

(a) Illustrate the algorithm starting with the BCD number 857, showing A and B at each step.
(b) Draw the block diagram of the BCD-to-binary converter.
(c) Draw a state diagram of the control circuit (three states). Use the following control signals: *St*: start conversion; *Sh*: shift right; *Co*: subtract correction if necessary; and *C9*: counter is in state 9, or *C10*: counter is in state 10. (Use either *C9* or *C10* but not both.)
(d) Write a Verilog description of the system.

4.14 This problem involves the design of a circuit that finds the square root of an 8-bit unsigned binary number N using the method of subtracting out odd integers. To find the square root of N, we subtract 1, then 3, then 5, and so on, until we can no longer subtract without the result going negative. The number of times we subtract is equal to the square root of N. For example, to find $\sqrt{27}$: $27 - 1 = 26; 26 - 3 = 23; 23 - 5 = 18; 18 - 7 = 11; 11 - 9 = 2; 2 - 11$ (can't subtract). Since we subtracted five times, $\sqrt{27} = 5$. Note that the final odd integer is $11_{10} = 1011_2$, and this consists of the square root ($101_2 = 5_{10}$) followed by a 1.

(a) Draw a block diagram of the square rooter that includes a register to hold N, a subtracter, a register to hold the odd integers, and a control circuit. Indicate where to read the final square root. Define the control signals used on the diagram.
(b) Draw a state graph for the control circuit using a minimum number of states. The N register should be loaded when $St = 1$. When the square root is complete, the control circuit should output a done signal and wait until $St = 0$ before resetting.

4.15 This problem concerns the design of a multiplier for unsigned binary numbers that multiplies a 4-bit number by a 16-bit number to give a 20-bit product. To speed up the multiplication, a 4-by-4 array multiplier is used so that we can multiply by 4 bits in one clock time instead of by only 1 bit at each clock time. The hardware includes a 24-bit accumulator register that can be shifted right 4 bits at a time using a control signal *Sh4*. The array multiplier multiplies 4 bits by 4 bits to give an 8-bit product. This product is added to the accumulator using an *Ad* control signal. When an *St* signal occurs, the 16-bit multiplier is loaded into the lower part of the A register. A done signal should be turned on when the multiplication is complete. Since both the array multiplier and adder are combinational circuits, the 4-bit multiply and the 8-bit add can both be completed in the same clock cycle. Do NOT include the array multiplier logic in your code; just use the overloaded "*" operator. If D and E are 4-bit unsigned numbers, $D * E$ will compute an 8-bit product.

(a) Draw a state graph for the controller (10 states)
(b) Write Verilog code for the multiplier. Use two always blocks (a combinational always block and a clocked always block).