CHAPTER 4

Datapath Components

4.1 INTRODUCTION

Chapters 2 and 3 introduced increasingly complex building blocks with which to build digital circuits. Those blocks included logic gates, multiplexors, decoders, basic registers, and controllers. Controllers are good for implementing systems having *control* inputs. This chapter instead focuses on creating building blocks for systems having *data* inputs. Control and data inputs differ as follows:

- ***Control:*** A control input is usually one bit and represents a particular event or command that influences or directs the system's mode of operation.
- ***Data:*** A data input is usually multiple bits that collectively represent a single entity, like a 32-bit number from a temperature sensor or an 8-bit ASCII character, and that are operated on or transformed by the system.

As an analogy, a television has control inputs coming from the remote control; those inputs control the TV's mode of operation, such as turning the volume up or changing the channel. A television also has data inputs coming from a video cable; those data inputs are operated on to create the video seen on the TV display. Another example is a hand-held calculator; a user inputs numbers (data input) and issues commands like add or multiply (control inputs) to operate on that data.

Not all inputs are just control or just data—some inputs have features of both, just as humans can't be strictly divided into "tall" and "short" categories.

While a controller is a good building block for systems having control inputs and outputs, new building blocks are needed for systems that operate on data. Such blocks are ***datapath components***. A circuit of datapath components is known as a ***datapath***.

Datapaths can become quite complex, and therefore it is crucial to build datapaths from datapath components that each encapsulates an appropriately high level of functionality. For example, if you had to build a bicycle, you would probably build it by combining tires, a frame, a seat, handlebars, and so on. Each of those components encapsulates a high-level function of part of a bicycle. You started with a tire and not with rubber and glue. Rubber and glue make up the design of a tire, not the design of a bicycle. Likewise, when we design datapaths, we must have appropriately high-level datapath components—logic gates are too low-level. This chapter defines such datapath components and builds some simple datapaths. Chapter 5 will show how to combine datapaths and controllers to build even more capable digital systems.

▶ 4.2 REGISTERS

An ***N-bit register*** is a sequential component that can store *N* bits. *N* is called the register ***width***. Typical register widths are 8, 16, and 32 bits, but can be as small as 1 or can be arbitrarily large. The bits in a register commonly represent data, such as 8 bits representing temperature data in binary.

The common name for storing data into a register is ***loading***; the names *writing* and *storing* are also used. The opposite action of loading a register is known as ***reading*** a register's contents. Reading consists merely of observing a register's outputs and is therefore is not synchronized with the clock. Furthermore, reading a register does not change the bits inside the register, just like reading this book does not change the words inside the book.

Registers come in a variety of styles. The upcoming sections introduce some of the most common styles. Registers are the most fundamental datapath component, so several examples will be presented.

Parallel-Load Register

The most basic type of register, shown in Figure 3.36 in Chapter 3, consists of a set of flip-flops that are loaded on every clock cycle. That basic register is useful as the state register in a controller, because a controller's state register should be loaded on every clock cycle. However, most other uses of registers require some way to control whether or not a register is loaded on a particular clock cycle—on some cycles the register should be loaded, whereas on other cycles the register should keep its previous value.

Control of the loading of a register can be achieved by adding a 2x1 multiplexor in front of each flip-flop of a basic register as shown for the 4-bit register in Figure 4.1(a). When the register's `load` control input is `0` and the clock signal rises, each flip-flop stores its own `Q` value, as shown in Figure 4.1(b). Because `Q` is a flip-flop's present content, the contents of the flip-flops, and hence the register's contents, do not change when `load` is `0`. In contrast, when the `load` input is `1` and the clock signal rises, each flip-flop is loaded with a data input `I0`, `I1`, `I2`, or `I3`—thus, the register is loaded with the data inputs when `load` is `1`.

A register with a load line that controls whether the register is loaded with external inputs, with those inputs loaded in parallel, is called a ***parallel-load register***. Figure 4.1(c) provides a block symbol for a 4-bit parallel-load register. A ***block symbol*** of a component shows a component's inputs and outputs without showing the component's internal details.

▶ WHY THE NAME "REGISTER"?

Historically, the term "register" referred to a sign or chalkboard onto which people could temporarily write out cash transactions, and later perform bookkeeping using those transactions. The term generally refers to a device for storing data. In this context, since a collection of flip-flops stores data, the name "register" seems quite appropriate.

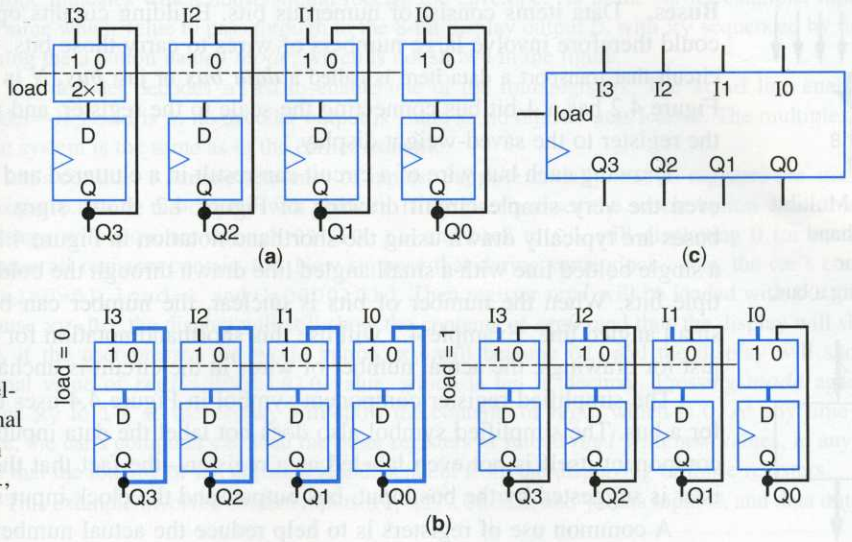


Figure 4.1 4-bit parallel-load register: (a) internal design, (b) paths when `load=0` and `load=1`, (c) block symbol.

Example 4.1 Weight sampler

Consider a scale used to weigh fruit at a grocery store . The scale may have a display that shows the present weight. We want to add a second display and a button that the user can press to remember the present weight (sometimes called "sampling"), so that when the fruit is removed, the remembered weight continues to be displayed on the second display. A block diagram of the system is shown in Figure 4.2.

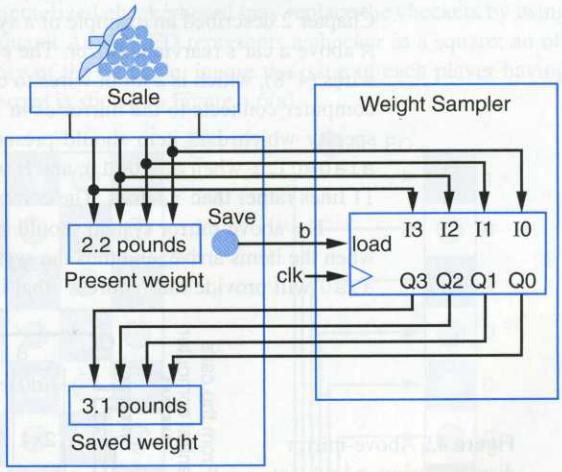


Figure 4.2 Weight sampler implemented using a 4-bit parallel-load register.

Assume that the scale outputs the present weight as a 4-bit binary number, and the "Present weight" and "Saved weight" displays automatically convert their input binary number to the proper displayed value. We can design the *WeightSampler* block using a 4-bit parallel-load register. We connect the button signal `b` to the load input of the register. The output connects to the "Saved weight" display. Whenever `b` is `1`, the weight value gets loaded into the register, and thus appears on the second display. When `b` returns to `0`, the register keeps its value, so the second display continues to show the same weight even if other items are placed on the scale and the first display changes. This example involved a control input `b`, and also two 4-bit data inputs and outputs.

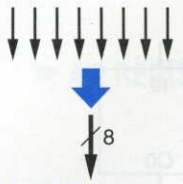


Figure 4.3 Multibit wire shorthand notation for representing a bus.

Buses. Data items consist of numerous bits. Building circuits operating on data items could therefore involve large numbers of wires to carry those bits. A group of N wires in a circuit that transport a data item is called a ***data bus*** or just ***bus***. N is the *width* of the bus. Figure 4.2 has a 4-bit bus connecting the scale to the register, and a 4-bit bus connecting the register to the saved-weight display.

Drawing each bus wire of a circuit can result in a cluttered and hard-to-read drawing; even the very simple circuit drawing of Figure 4.2 shows signs of such clutter. Thus, buses are typically drawn using the shorthand notation in Figure 4.3. The bus is drawn as a single bolded line with a small angled line drawn through the bolded line to denote multiple bits. When the number of bits is unclear, the number can be written next to that small angled line. Example 4.2 will use this shorthand notation for buses. The notation is just for drawings; the actual number of wires in the circuit is unchanged.

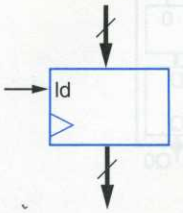


Figure 4.4 Simplified register symbol.

The simplified register component symbol in Figure 4.4 uses the shorthand notation for a bus. The simplified symbol also does not label the data inputs and outputs, and the component itself is not even labeled as a register—the fact that the component is a register is suggested by the bus input, bus output, and the clock input symbol (the triangle).

A common use of registers is to help reduce the actual number of wires in a circuit. Registers help by enabling multiplexing of M different data items over a single bus, with each data item stored in its own register, as will be illustrated in Example 4.2.

Example 4.2 Automobile above-mirror display using parallel-load registers

Chapter 2 described an example of a system that could display one of four 8-bit inputs, `T`, `A`, `I`, and `M` above a car's rearview mirror. The car's central computer was connected to the system using 32 wires (4*8), which is a lot of wires to connect from the computer to the mirror. Instead, assume the computer connects to the mirror as in Figure 4.5, using an 8-bit bus (`C`), 2 control lines `a1a0` that specify which data item should presently appear on `C` (displaying `T` when `a1a0=00`, `A` when `a1a0=01`, `I` when `a1a0=10`, and `M` when `a1a0=11`), and a load control line `load`, for a total of 11 lines rather than 32 lines. The computer can send the data items in any order and at any time.

The above-mirror system should store data items in the appropriate register according to `a1a0` when the items arrive, and thus the system needs four parallel-load registers to store each data item. `a1a0` will provide the "address" that indicates which register to load, much like the address on an

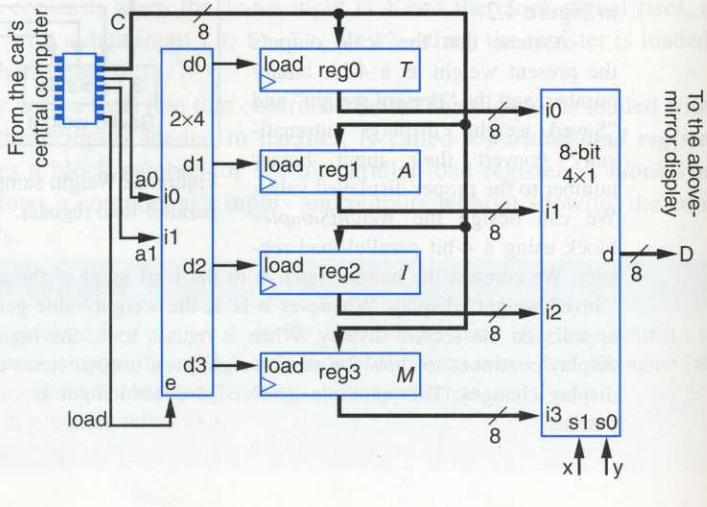


Figure 4.5 Above-mirror display design. `a1a0`, set by the car's central computer, determines which register to load with `C`, while `load=1` enables such loading. `xy`, which are independent of `a1a0` and are set by the user pressing a `mode` button, determine which register to output to the display D.

envelope indicates which house should receive the envelope. As in the earlier example, inputs xy determine which value to pass through to the 8-bit display output D, with xy sequenced by the user pressing the a button named *mode*, which is not shown in the figure.

The decoder decodes a1a0 to enable one of the four registers. The load line enables the decoder—if load is 0, no decoder output is 1 and so no register gets loaded. The multiplexer part of the system is the same as in the earlier example.

Let's consider a sample sequence of inputs. Suppose initially that all registers are storing 0s and xy=00. Thus, the display will show 0. If the user presses the *mode* button four times, the inputs xy will sequence through 01, 10, 11, and back to 00, still displaying 0 for each press (because all registers contain 0s). Now suppose that during some clock cycle, the car's computer sets a1a0=01, load=1, and C=00001010. Then register *reg1* will be loaded with 00001010. Because xy=00, the display will still show the contents of *reg0*, and thus the display will show 0. Now, if the user presses the *mode* button, xy will become 01, and the display will show the decimal value of *reg1*'s 00001010 value, which is ten in decimal. Pressing *mode* again will change xy to 10, so the display will show the contents of *reg2*, which is 0. At any time in the future, the car's computer can load the other registers or reload *reg1* with new values, in any order. Note that the loading of the registers is independent from the displaying of those registers.

This example involved control inputs a0, a1, load, x, and y, data input C, and data output D.

Example 4.3 Computerized checkerboard

Checkers (known in some countries as "draughts") is one of the world's most popular board games. A checkerboard consists of 64 squares, formed from 8 columns and 8 rows. Each player starts with 12 checkers (pieces) on the board. A computerized checkerboard may replace the checkers by using an LED (light-emitting diode) in each square. An on LED represents a checker in a square; an off LED represents no checker. For simplicity of the example, ignore the issue of each player having his own color of checkers. An example board is shown in Figure 4.6(a).

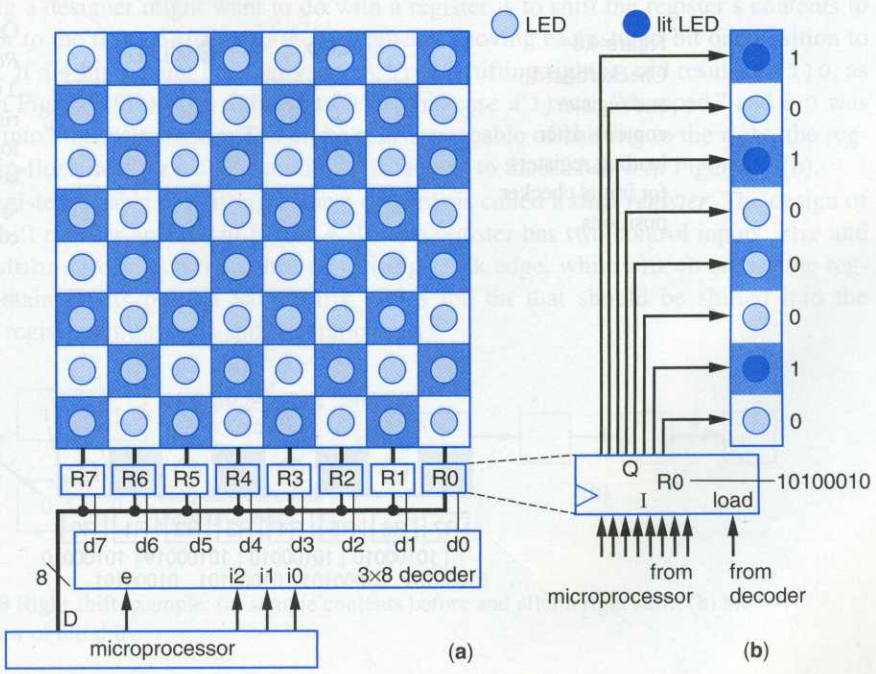


Figure 4.6 An electronic checkerboard: (a) eight 8-bit registers (*R7* through *R0*) can be used to drive the 64 LEDs, using one register per column, (b) detail of how one register connects to a column's LEDs and how the value 10100010 stored in that register would light three LEDs.

A computerized checkerboard typically has a microprocessor that keeps track of where each piece is located, moves pieces according to user commands or according to a checker-playing program (when playing against the computer), and keeps score.

Notice that the microprocessor must set values for 64 bits, one bit for each square. However, the inexpensive type of microprocessor used in such a device typically does not have 64 pins. The microprocessor needs external registers to store those bits that drive the LEDs. The microprocessor will write to those registers one at a time. The sequence of writes to the registers is so fast that an observer would see all the LEDs change at the same time, unable to notice that some LEDs are changing microseconds earlier than others.

Let's use one register per column, meaning eight 8-bit registers will be used in total as shown below the checkerboard in Figure 4.6(a). We named the registers *R7* through *R0*. Each register's 8 bits of data correspond to a particular row in the register's column, indicating whether the respective LED is on or off, as shown in Figure 4.6(b). The eight registers are connected to the microprocessor. The microprocessor uses eight pins (`D`) for data, three pins (`i2, i1, i0`) for addressing the appropriate register (which is decoded into a load line for each of the 8 registers), and one pin (`e`) for the register load line (connected to the decoder's enable), for a total of 12 pins—far fewer than 64 pins if registers were not used. To configure the checkerboard for the beginning of a game, the microprocessor would perform the sequence of register writes shown in Figure 4.7.

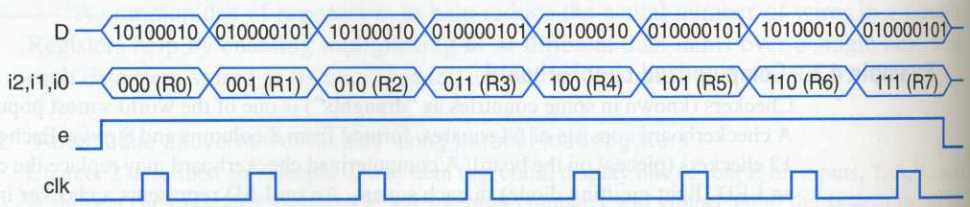
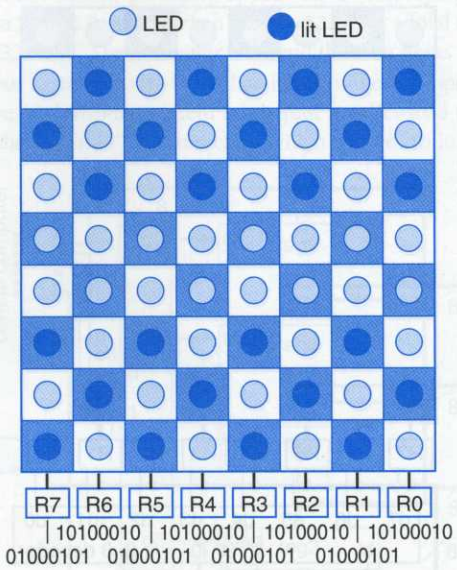


Figure 4.7 Timing diagram showing an input sequence that initializes an electronic checkerboard.

Figure 4.8 Checkerboard and register contents after loading registers for initial checker positions.



On the first rising clock edge, *R0* gets loaded with `10100010`. On the second rising clock edge, *R1* gets loaded with `01000101`. And so on. After eight clock cycles, the registers would contain the desired values, and the board's LEDs would be lit as shown in Figure 4.8.

▶ HOW DOES IT WORK? COMPUTERIZED BOARD GAMES.

Many of you have played a computerized board game, like checkers, backgammon, or chess, either using boards with small displays to represent pieces, or perhaps using a graphics program on a personal computer or website. The main method the computer uses for choosing among possible next moves is called ***lookahead***. For the current configuration of pieces on the board, the computer considers all possible single moves that it might make. For each such move, it might also consider all possible single moves by the opponent. For each new configuration resulting from possible moves, the computer evaluates the configuration's goodness, or quality, and picks a move that may lead to the best configuration. The number of moves that the computer looks ahead (one computer move, one opponent move, another computer move, another opponent move) is called the *lookahead amount*. Good programs might lookahead three, four, five moves, or more. Looking ahead is costly in terms of compute time and memory—if each player has 10 possible moves per turn, then looking ahead two moves results in 10*10 =100 configurations to evaluate; three moves results in 10*10*10=1000 configurations, four moves in 10,000 configurations, and so on. Good game-playing programs will "prune" configurations that appear to be very bad and thus unlikely to be chosen by an opponent, just as humans do, to reduce the configurations to be considered. Computers can examine millions of configurations, whereas humans can only mentally examine perhaps a few dozen. Chess, being perhaps the most complex of popular board games, has attracted extensive attention since the early days of computing. Alan Turing, considered one of the fathers of Computer Science, wrote much about using computers for chess, and is credited as having written the first computer chess program in 1950. However, humans proved better than computer chess programs until 1997, when IBM's Deep Blue computer defeated the reigning world champion in a classic chess match. Deep Blue had 30 IBM RS-6000 SP processors connected to 480 special purpose chess chips, and could evaluate 200 million moves per second, and hence many billions of moves in a few minutes. Today, chess tournaments not only match humans against computer programs, but also programs against programs, many such tournaments hosted by the International Computer Games Association.

(Source: *Computer Chess History*, by Bill Wall).

Shift Register

One thing a designer might want to do with a register is to shift the register's contents to the left or to the right. ***Shifting*** to the right means moving each stored bit one position to the right. If a 4-bit register originally stores `1101`, shifting right would result in `0110`, as shown in Figure 4.9(a). The rightmost bit (in this case a `1`) was "dropped," and a `0` was "shifted into" the leftmost bit. To build a register capable of shifting to the right, the register's flip-flops need to include connections similar to those shown in Figure 4.9(b).

A register capable of shifting its own contents is called a ***shift register***. The design of a right shift register appears in Figure 4.10. The register has two control inputs, `shr` and `shr_in`. `shr=1` causes a right shift on a rising clock edge, while `shr=0` causes the register to maintain its present value. `shr_in` is the bit that should be shifted into the leftmost register bit during a shift operation.

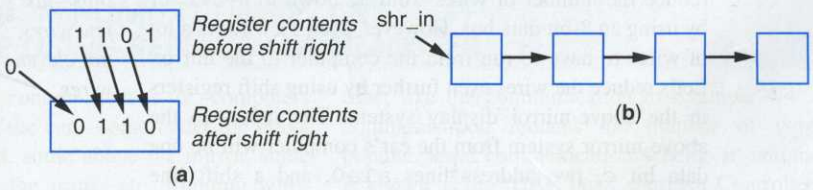


Figure 4.9 Right shift example: (a) sample contents before and after a right shift, (b) bit-by-bit view of the shift.

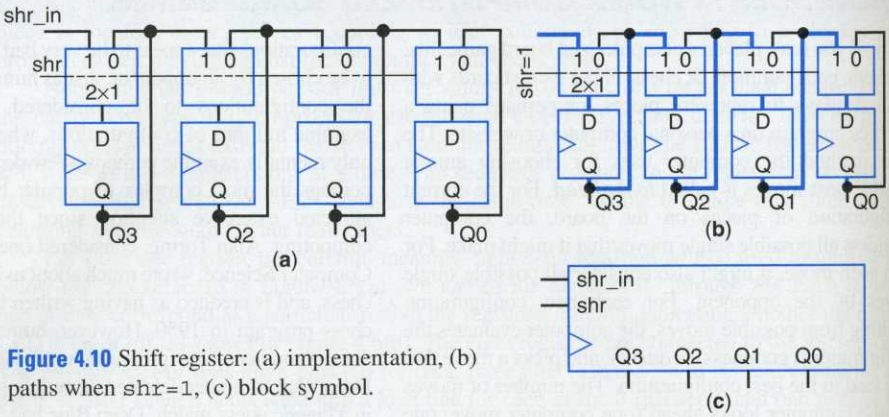


Figure 4.10 Shift register: (a) implementation, (b) paths when `shr=1`, (c) block symbol.

A ***rotate register*** is a slight variation of a shift register in which the outgoing bit gets shifted back in as the incoming bit. So on a right rotate, the rightmost bit gets shifted into the leftmost bit, as shown in Figure 4.11.The design for a rotate register is a slight modification of the design of Figure 4.10. The rightmost flip-flop output, rather than the `shr_in` input, would be connected to the leftmost mux's `i1` input. A rotate register also needs some way to get values into the register—either via a shift, or via parallel load.

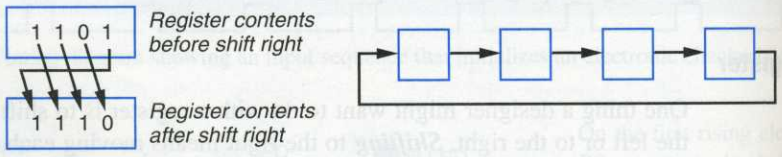
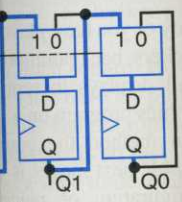


Figure 4.11 Right rotate example: (a) register contents before and after the rotate, (b) bit-by-bit view of the rotate operation.

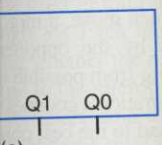
Example 4.4 Above-mirror display using shift registers

Example 4.2 redesigned the connection between a car's central computer and an above-mirror display system to reduce the number of wires from 32 down to 8+2+1=11 by using an 8-bit data bus. However, even 11 wires is a lot of wires to have to run from the computer to the mirror. Let's reduce the wires even further by using shift registers in the above-mirror display system. The inputs to the above-mirror system from the car's computer will be one data bit `c`, two address lines `a1a0`, and a shift line `shift`, for a total of only 4 wires. When the computer is

Figure 4.1 Serial communication enables thin cables.

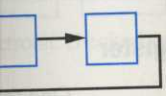


(b)



(c)

ich the outgoing bit gets
most bit gets shifted into
register is a slight modi-
output, rather than the
ut. A rotate register also
, or via parallel load.



the rotate, (b) bit-

dle
e
t a
s,
n

4.1 Serial communication
s thin cables.

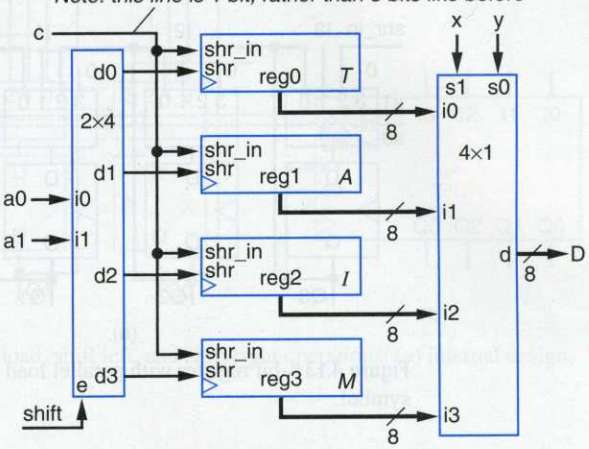


Figure 4.12 Above-mirror display design using shift registers to reduce the number of lines coming from the car's computer. The computer sets `a1a0` to the desired register to load, and then holds `shift=1` for eight clock cycles. During those cycles, it sets `c` to the desired register contents bit-by-bit, one bit per clock cycle. The result is that the desired register is loaded with the sent 8-bit value.

to write to one of the above-mirror system's registers, the computer sets `a1a0` appropriately and then sets `shift` to `1` for exactly eight clock cycles.

For each of those eight clock cycles, the computer will set `c` to one bit of the 8-bit data to be loaded, starting with the least-significant (rightmost) bit on the first clock cycle, and ending with the most-significant (leftmost) bit on the eighth clock cycle. The above-mirror system can thus be designed as shown in Figure 4.12.

When `shift=1`, the appropriate register gets a new value shifted in during the next eight clock cycles. This method achieves the same results as parallel loading but with fewer wires.

This example demonstrates a form of communication between digital circuits known as ***serial communication***, in which the circuits communicate data by sending the data one bit at a time.

Multifunction Registers

Some registers perform a variety of *operations*, also called *functions*, like load, shift right, shift left, rotate right, rotate left, etc. The desired operation can be achieved by setting the register's control inputs. The following section introduces several such ***multifunction registers***.

Register with Parallel Load and Shift Right

A popular combination of operations on a register is parallel load and shift. We can design a 4-bit register capable of parallel load and shift right as shown in Figure 4.13(a). Figure 4.13(b) shows a block symbol of the register.

▶ COMPUTER COMMUNICATIONS IN AN AUTOMOBILE USING SERIAL DATA TRANSFER

Modern automobiles contain dozens of computers distributed throughout the car—some under the hood, some in the dashboard, some above the mirror, some in the door, some in the trunk, etc. Running wires throughout the car so those computers can communicate is a challenge. Most automobile computers communicate *serially*, meaning one bit at a time, like the communication in Example 4.4. Serial communication reduces the number of wires. A popular serial communication scheme in automobiles is known as the "CAN bus," short for Controller Area Network, which is an international standard defined by ISO (International Standards Organization) standard number 11898.

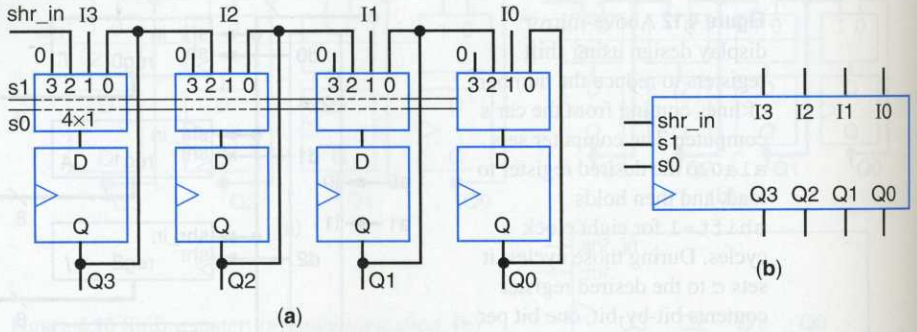


Figure 4.13 4-bit register with parallel load and shift right operations: (a) internal design, (b) block symbol.

The design uses a 4x1 mux rather than a 2x1 mux in front of each flip-flop, because each flip-flop can receive its next bit from one of three locations. The fourth mux input is unused. The table in Figure 4.14 describes the register's behavior. Such a table listing the operation for each combination of a component's control inputs is called an ***operation table***.

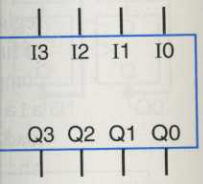
s1	s0	Operation
0	0	Maintain present value
0	1	Parallel load
1	0	Shift right
1	1	(Unused)

Figure 4.14 Operation table of a 4-bit register with parallel load and shift right operations.

Let's examine the mux and flip-flop of the rightmost bit. When `s1s0=00`, the mux passes the current flip-flop value back to the flip-flop, causing the flip-flop to get reloaded with its current value on the next rising clock, thus maintaining the current value. When `s1s0=01`, the mux passes the external `I0` input to the flip-flop, causing the flip-flop to get loaded. When `s1s0=10`, the mux passes the present value of the flip-flop output from the left, `Q1`, thus causing a right shift. `s1s0=11` is not a legal input to the register and thus should never occur; the mux passes `0`s in this case, thus clearing the flip-flop.

▶ UNUSED INPUTS

The circuit in Figure 4.13 included a mux with 4 inputs, of which only 3 inputs were needed. Notice that we actually set the unused input to a particular value, rather than simply leaving the input unconnected. Remember that the input is controlling transistors inside the component—if we don't assign a value to the input, will the internal transistors conduct or not conduct? We don't really know, and so we could get undesired behavior from the mux. Leaving inputs unconnected should not be done. On the other hand, leaving outputs unconnected is no problem—an unconnected output may have a `1` or a `0` that simply doesn't control anything else.



(b)

ternal design, (b) block

Operation
Maintain present value
Parallel load
Shift right
(Unused)

peration table of a 4-bit parallel load and shift ns.

e next rising clock, thus the external `I0` input to `I0`, the mux passes the g a right shift. `s1s0=11` ne mux passes `0`s in this

r not conduct? We don't get undesired behavior nconnected should not be g outputs unconnected is utput may have a `1` or a ything else.

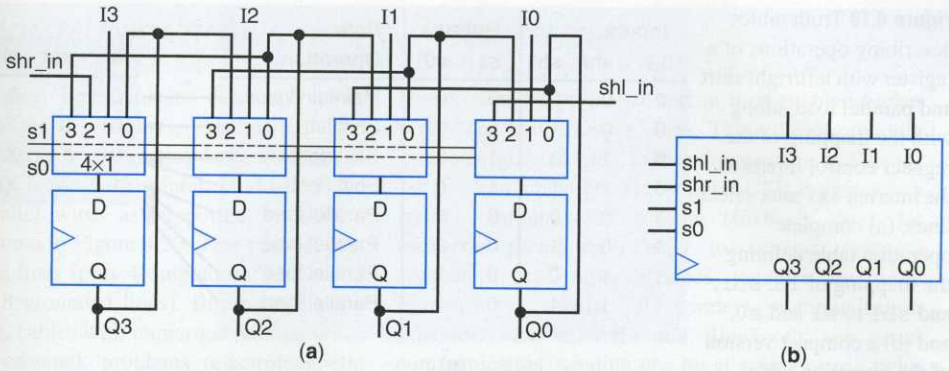


Figure 4.15 4-bit register with parallel load, shift left, and shift right operations: (a) internal design, (b) block symbol.

Register with Parallel Load, Shift Left, and Shift Right

Adding a shift left operation to the above 4-bit register is straightforward and is shown in Figure 4.15. Instead of connecting `0`s to the `I3` input of each 4x1 mux, we instead connect the output from the flip-flop to the right. The rightmost mux's `I3` input would be connected to an additional input `shl_in`.

The register has the operations shown in Figure 4.16.

s1	s0	Operation
0	0	Maintain present value
0	1	Parallel load
1	0	Shift right
1	1	Shift left

Figure 4.16 Operation table of a 4-bit register with parallel load, shift left, and shift right operations.

Load/Shift Register with Separate Control Inputs for Each Operation

Registers typically don't come with control inputs that encode the operation into the minimum number of bits like the control inputs on the registers designed above. Instead, each operation usually has its own control input.

For example, a register with the operations of load, shift left, and shift right might have the control inputs and the operation table shown in Figure 4.17. The four possible operations (maintain, shift left, shift right, and load) require at least two control inputs; the figure shows that the register has three control inputs—`ld`, `shr`, and `shl`.

ld	shr	shl	Operation
0	0	0	Maintain present value
0	0	1	Shift left
0	1	0	Shift right
0	1	1	Shift right – shr has priority over shl
1	0	0	Parallel load
1	0	1	Parallel load – ld has priority
1	1	0	Parallel load – ld has priority
1	1	1	Parallel load – ld has priority

Figure 4.17 Operation table of a 4-bit register with separate control inputs for parallel load, shift left, and shift right.

Notice that the register designer must decide how the

Inputs			Outputs		Note
ld	shr	shl	s1	s0	Operation
0	0	0	0	0	Maintain value
0	0	1	1	1	Shift left
0	1	0	1	0	Shift right
0	1	1	1	0	Shift right
1	0	0	0	1	Parallel load
1	0	1	0	1	Parallel load
1	1	0	0	1	Parallel load
1	1	1	0	1	Parallel load

(a)

ld	shr	shl	Operation
0	0	0	Maintain value
0	0	1	Shift left
0	1	X	Shift right
1	X	X	Parallel load

(b)

Figure 4.18 Truth tables describing operations of a register with left/right shift and parallel load, along with the mapping of the register control inputs to the internal 4x1 mux select lines: (a) complete operation table defining the mapping of `ld`, `shr`, and `shl` to `s1` and `s0`, and (b) a compact version of the operation table.

register will respond if more than one control input is `1`. The operation table shows that if the user sets both `shr` and `shl`, `shr` gets priority. `ld` has priority over `shr` and `shl`.

The internal design of such a register is similar to the load/shift register designed above, except that the three control inputs of `ld`, `shl`, and `shr` need to be mapped to the two control inputs `s1` and `s0` of the earlier register. A simple combinational circuit can be used to perform such mapping, as shown in Figure 4.19.

We can design that combinational circuit starting from a simple truth table shown in Figure 4.18(a). From the table, we derive the following equations for the register's combinational circuit:

```
s1 = ld'*shr'*shl + ld'*shr*shl' + ld'*shr*shl
s0 = ld'*shr'*shl + ld
```

Replacing the combinational circuit box in Figure 4.19 by the gates described by the above equations would complete the register's design.

Register datasheets typically show the register operation table in a compact form, taking advantage of the priorities among the control inputs, as in Figure 4.18(b). A single `X` in a row means that row is actually two rows in the complete table, with one row having `0` in the position of the `X`, the other row having `1`. Two `X`s in a row means that row is actually four rows in the complete table, one row having `00` in the positions of those `X`s, another row having `01`, another `10`, and another `11`. And so on for three `X`s, representing

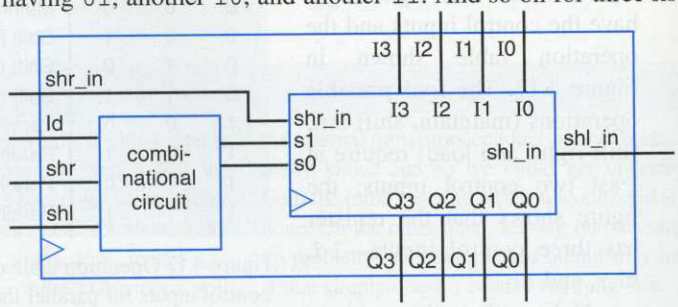


Figure 4.19 A combinational circuit maps the control inputs `ld`, `shr`, and `shl` to the mux select inputs `s1` and `s0`.

shr	shl	Operation
0	0	Maintain value
0	1	Shift left
1	X	Shift right
1	X	Parallel load

(b)

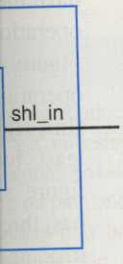
ation table shows that if
over `shr` and `shl`.
d/shift register designed
eed to be mapped to the
binational circuit can be

ple truth table shown in
for the register's combi-

`r*shl`

gates described by the

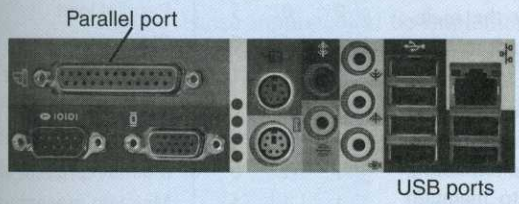
able in a compact form,
Figure 4.18(b). A single
ble, with one row having
a row means that row is
the positions of those `X`s,
for three `X`s, representing



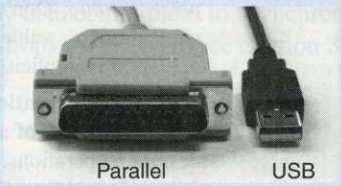
and `shl` to the mux select

▶ SERIAL COMMUNICATION IN DESKTOP COMPUTERS.

A desktop PC must communicate data with other devices like printers, displays, keyboards, and cameras. In the past, communicating large amounts of data, like sending a file to a printer, could be done faster using parallel wires as supported by a PC's parallel port—shown in Figure 4.20. That parallel port has 8 output data lines (plus 4 output control lines, 5 input lines, and 8 grounded lines). But parallel ports and plugs are big, cables with numerous internal wires are costly, and crosstalk problems (electromagnetic fields generated by a changing signal on one wire interfering with the signal on a nearby wire) exist. As higher-speed circuits could be designed over the years, communication clock frequencies could be made faster too. But transmitting data in parallel at high frequencies creates even more crosstalk problems. Transmitting data serially became more appealing, involving smaller ports and plugs, cheaper wires, and fewer crosstalk problems that in turn enabled higher power and hence longer wires. The reduced crosstalk problems also enabled higher frequencies and hence faster communication. The popular USB interface is a serial communication scheme (***USB*** is short for ***universal serial bus***) used to connect personal computers and other devices together by wire. Furthermore, nearly all wireless communication schemes, such as WiFi and BlueTooth, use serial communication, sending one bit at a time over a radio frequency. While data communication between devices may be serial, computations inside devices are typically done in parallel. Thus, shift registers are commonly used inside circuits to convert internal parallel data into serial data to be sent to another device, and to receive serial data and convert that data into parallel data for internal device use.



(a)



(b)

Figure 4.20 Parallel versus serial communication in desktop computers: (a) a PC having a parallel port and six serial USB ports, (b) parallel and USB plugs/cables.

8 rows. Note that putting higher-priority control inputs to the left in the table keeps the table's operations nicely organized.

Register Design Process

Table 4.1 describes a general process for designing a register with any number of functions.

TABLE 4.1 Four-step process for designing a multifunction register.

	Step	Description
1.	*Determine mux size*	Count the number of operations (don't forget the maintain present value operation) and add in front of each flip-flop a mux with at least that number of inputs.
2.	*Create mux operation table*	Create an operation table defining the desired operation for each possible value of the mux select lines.

TABLE 4.1 Four-step process for designing a multifunction register.

3.	*Connect mux inputs*	For each operation, connect the corresponding mux data input to the appropriate external input or flip-flop output (possibly passing through some logic) to achieve the desired operation.
4.	*Map control lines*	Create a truth table that maps external control lines to the internal mux select lines, with appropriate priorities, and then design the logic to achieve that mapping

We'll illustrate the register design process with an example.

Example 4.5 Register with load, shift, and synchronous clear and set

We want to design a register with the following operations: load, shift left, synchronous clear, and synchronous set, with unique control inputs for each operation (`ld`, `shl`, `clr`, `set`). The ***synchronous clear*** operation loads all `0`s into the register on the next rising clock edge. The ***synchronous set*** operation loads all `1`s into the register on the next rising clock edge. The term "synchronous" is included because some registers come with *asynchronous* clear or set operations (see Section 3.5). Following the register design method of Table 4.1, we perform the following steps:

Step 1: **Determine mux size.** There are 5 operations—load, shift left, synchronous clear, synchronous set, and *maintain present value*. Don't forget the maintain present value operation; that operation is implicit.

Step 2: **Create mux operation table.** We'll use the first 5 inputs of an 8x1 mux for the desired 5 operations. For the remaining 3 mux inputs, we choose to maintain the present value, though those mux inputs should never be utilized. The table is shown in Figure 4.20.

s2	s1	s0	Operation
0	0	0	Maintain present value
0	0	1	Parallel load
0	1	0	Shift left
0	1	1	Synchronous clear
1	0	0	Synchronous set
1	0	1	Maintain present value
1	1	0	Maintain present value
1	1	1	Maintain present value

Figure 4.20 Operation table for a register with load, shift, and synchronous clear and set.

Step 3: **Connect mux inputs.** We connect the mux inputs as shown in Figure 4.21, which for simplicity shows only the *n*th flip-flop and mux of the register.

Step 4: **Map control lines.** We'll give `clr` highest priority, followed by `set`, `ld`, and `shl`, so the register control inputs would be mapped to the 8x1 mux select lines as shown in Figure 4.22.

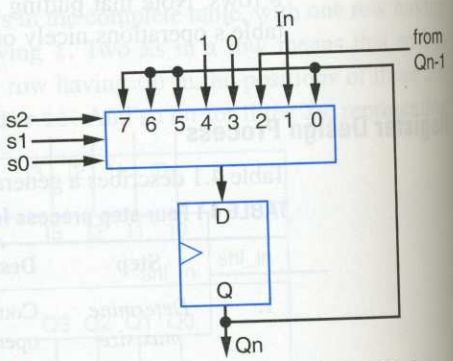


Figure 4.21 *N*th bit-slice of a register with the following operations: maintain present value, parallel load, shift left, synchronous clear, and synchronous set.

▶ 4.3 ADDE

Figure 4.22 Truth table for the control lines of a register with the *N*th bit-slice shown in Figure 4.21.

Inputs				Outputs			
clr	set	ld	shl	s2	s1	s0	Operation
0	0	0	0	0	0	0	Maintain present value
0	0	0	1	0	1	0	Shift left
0	0	1	X	0	0	1	Parallel load
0	1	X	X	1	0	0	Set to all 1s
1	X	X	X	0	1	1	Clear to all 0s

Looking at each output in Figure 4.22, we derive the equations describing the circuit that maps the external control inputs to the mux select lines as follows:

```
s2 = clr'*set
s1 = clr'*set'*ld'*shl + clr
s0 = clr'*set'*ld + clr
```

We could then create a combinational circuit implementing those equations to map the external register control inputs to the mux select lines and hence to complete the register's design.

Some registers come with asynchronous clear and/or asynchronous set control inputs. Those inputs could be implemented by connecting them to asynchronous clear or asynchronous set inputs that exist on the flip-flops themselves (see Section 3.5).

4.3 ADDERS

Adding two binary numbers is perhaps the most common operation performed on data in a digital system. An ***N-bit adder*** is a combinational component that adds two *N*-bit data inputs `A` and `B` representing binary numbers, and generates an *N*-bit data output `S` representing the sum and a 1-bit output `C` representing the carry-out. A 4-bit adder adds two 4-bit numbers and outputs a 4-bit sum and a carry bit. For example, `1111 + 0001` would result in a carry of `1` and a sum of `0000`—or `10000` if you treat the carry bit and sum bits as one 5-bit result. *N* is the ***width*** of the adder. Designing fast yet size-efficient adders is a subject that has received considerable attention for many decades.

Although it appears that an *N*-bit adder could be designed by following the combinational logic design process of Table 2.5, building an *N*-bit adder following that process is not practical when *N* is much larger than 8. To understand why, consider using that process to build a 2-bit adder, which adds two 2-bit numbers. The desired behavior can be captured as the truth table in Figure 4.23. Each output could then be converted to a sum-of-products equation and implemented as a two-level gate-based circuit.

The problem is that the approach results in excessively large truth tables and too many gates for wider adders. A 16-bit adder has 16 + 16 = 32 inputs, meaning the truth table would have over *four billion rows*. A two-level gate-based implementation of that table would likely require millions of gates. To illustrate this point, we performed an experiment that used the standard combinational logic design process to create adders of increasing width, starting with 1-bit adders on up. We used an advanced commercial logic

Inputs				Outputs		
a1	a0	b1	b0	c	s1	s0
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	1	0
0	0	1	1	0	1	1
0	1	0	0	0	0	1
0	1	0	1	0	1	0
0	1	1	0	0	1	1
0	1	1	1	1	0	0

Inputs				Outputs		
a1	a0	b1	b0	c	s1	s0
1	0	0	0	0	1	0
1	0	0	1	0	1	1
1	0	1	0	1	0	0
1	0	1	1	1	0	1
1	1	0	0	0	1	1
1	1	0	1	1	0	0
1	1	1	0	1	0	1
1	1	1	1	1	1	0

Figure 4.23 Truth table for a 2-bit adder.

design tool, and asked the tool to create a design using two levels of logic (one level of AND gates feeding into an OR gate for each output) and using the minimum number of transistors.

The plot in Figure 4.24 summarizes results. A 6-bit adder (N=6) required about 2,000 transistors, a 7-bit adder required 4,000 transistors; and an 8-bit adder required 8,000 transistors. Notice how fast the number of transistors grows as the adder width is increased. This fast growth is an effect of exponential growth—for an adder width of N, the number of truth table rows is proportional to 2^N (more precisely, to 2^{N+N}). We could not complete our experiments for adders larger than 8 bits—the tool simply could not complete the design in a reasonable amount of time. The tool needed 3 seconds to build the 6-bit adder, 40 seconds to build the 7-bit adder, and 30 minutes for the 8-bit adder. The 9-bit adder didn't finish after one full day. Clearly, this exponential growth prohibits using the standard design process for adders wider than perhaps 8 to 10 bits. Looking at this data, can you predict the number of transistors required by a 16-bit adder or a 32-bit adder using two levels of gates? From the figure, it looks like the number of transistors is doubling for each increase in N. Assuming the doubling trend continues for larger adders, then a 16-bit adder would have 8 more doublings beyond the 8-bit adder, meaning multiplying the size of the 8-bit adder by 2^8=256. So a 16-bit adder would require 8000 * 256 = about two million transistors. A 32-bit adder would require an additional 2^{16}= 65,536 doublings, meaning about 2 million * 65,536 = over *100 billion transistors*. That's a ridiculous number of transistors just to add two numbers. We clearly need another approach for designing larger adders.

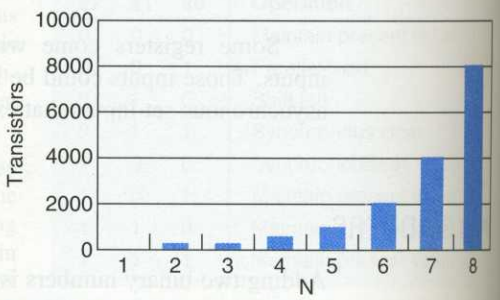
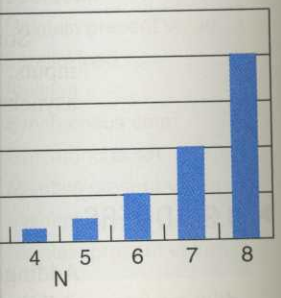


Figure 4.24 Why large adders aren't built using standard two-level combinational logic—notice the exponential growth. How many transistors would a 32-bit adder require?

Outputs		
c	s1	s0
0	1	0
0	1	1
1	0	0
1	0	1
0	1	1
1	0	0
1	0	1
1	1	0

els of logic (one level of
the minimum number of



ders aren't built using
inational logic—notice the
many transistors would a

adder, 40 seconds to build
dder didn't finish after one
tandard design process for
an you predict the number
two levels of gates? From
g for each increase in *N*.
a 16-bit adder would have
the size of the 8-bit adder
t two million transistors. A
s, meaning about 2 million
mber of transistors just to
ning larger adders.

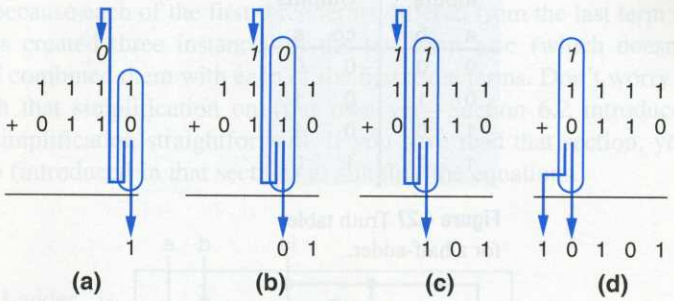


Figure 4.25 Adding two binary numbers by hand, column by column: (a) rightmost column sums to `1`, with no carry (`0`) to the next column, (b) second column sums to `0` with a `1` carried to the next column because `0+1+1 = 10` in base two, (c) third column is `1+1+1 = 11`, so the sum is `1` and a `1` is carried to the next column, (d) leftmost column sums to `0` with a final carry of `1`.

Adder—Carry-Ripple Style

An alternative approach to the standard combinational logic design process for adding two binary numbers is to instead create a circuit that mimics how people add binary numbers by hand—namely one column at a time. Consider the addition of the binary number `A=1111` (15 in base ten) with `B=0110` (6 in base ten), column by column, shown in Figure 4.25.

For each column, three bits are added, resulting in a sum bit for the present column and a carry bit for the next column. The first column is an exception in that only two bits are added, but that column still results in a sum bit and a carry bit. The carry bit of the last column becomes the fifth bit of the sum. The sum for the above numbers is `10101` (21 in base ten).

We can create a combinational component to perform the required addition for a single column, and then use four of those components to add the two 4-bit numbers. The inputs and outputs of such a component for a column are shown in Figure 4.26. Bear in mind that this method of creating an adder is intended to enable efficient design of wide adders such as a 32-bit adder. We illustrate the method using a 4-bit adder because that size adder keeps our figures small and readable, but if all we really needed was a 4-bit adder, the standard combinational logic design process for two-level logic might be sufficient.

We'll now design the components in each column of Figure 4.26.

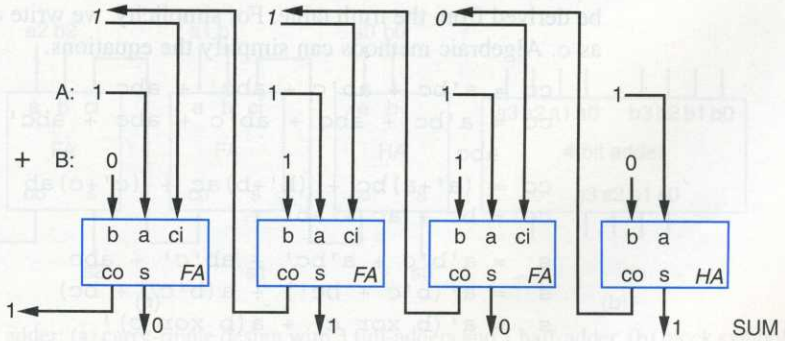


Figure 4.26 Using combinational components to add two binary numbers column by column.

Inputs		Outputs	
a	b	co	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Figure 4.27 Truth table for a half-adder.

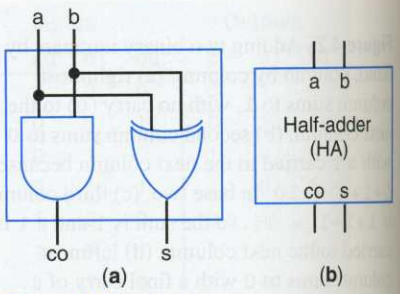


Figure 4.28 Half-adder: (a) circuit, (b) block symbol.

Half-Adder

A ***half-adder*** is a combinational component that adds two bits (`a` and `b`), and outputs a sum bit (`s`) and carry-out bit (`co`). (Note that we did *not* say that a half-adder adds *two 2-bit numbers*—a half-adder merely adds *two bits*.) The component labeled *HA* in Figure 4.26 is a half-adder. A half-adder can be designed using the straightforward combinational logic design process from Chapter 2 as follows:

Step 1: Capture the function. A truth table easily captures the function. The truth table is shown in Figure 4.27.

Step 2A: Create equations. The equations for each truth table output are `co = ab` and `s = a'b + ab'`, which is the same as `s = a xor b`.

Step 2B: Implement as a circuit. The circuit for a half-adder implementing the equations is shown in Figure 4.28(a). Figure 4.28(b) shows a block symbol of a half-adder.

Full-Adder

A ***full-adder*** is a combinational component that adds three bits (`a`, `b`, and `ci`) and outputs a sum bit (`s`) and a carry-out bit (`co`). (A full-adder does *not* add *two 3-bit numbers*—it merely adds *three bits*.) The three components labeled *FA* in Figure 4.26 are full-adders. A full-adder can be designed using the combinational logic design process as follows:

Step 1: Capture the function. A truth table captures the function easily, shown in Figure 4.29.

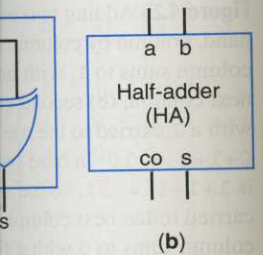
Step 2A: Create equations. Equations for `co` and `s` can be derived from the truth table. For simplicity, we write `ci` as `c`. Algebraic methods can simplify the equations.

```
co = a'bc + ab'c + abc' + abc
co = a'bc + abc + ab'c + abc + abc' +
     abc
co = (a'+a)bc + (b'+b)ac + (c'+c)ab
co = bc + ac + ab

s  = a'b'c + a'bc' + ab'c' + abc
s  = a'(b'c + bc') + a(b'c' + bc)
s  = a'(b xor c) + a(b xor c)'
s  = a xor b xor c
```

Inputs			Outputs	
a	b	ci	co	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Figure 4.29 Truth table for a full-adder.



-adder: (a) circuit,
l.

a and b), and outputs a
a half-adder adds *two 2-*
nt labeled *HA* in Figure
raightforward combina-

function. The truth table

output are co = ab and

implementing the equa-
ymbol of a half-adder.

a, b, and ci) and outputs
dd *two 3-bit numbers*—it
gure 4.26 are full-adders.
gn process as follows:

Inputs			Outputs	
a	b	ci	co	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

ure 4.29 Truth table for a
l-adder.

During algebraic simplification for co, each of the first three terms could be combined with the last term abc, because each of the first three terms differed from the last term in just one literal. We thus created three instances of the last term abc (which doesn't change the function) and combined them with each of the first three terms. Don't worry if you can't come up with that simplification on your own yet—Section 6.2 introduces methods to make such simplification straightforward. If you have read that section, you might try using a K-map (introduced in that section) to simplify the equations.

Step 2B: Implement as a circuit.

The circuit for a full-adder is shown in Figure 4.30(a). The full-adder's block symbol is shown in Figure 4.30(b).

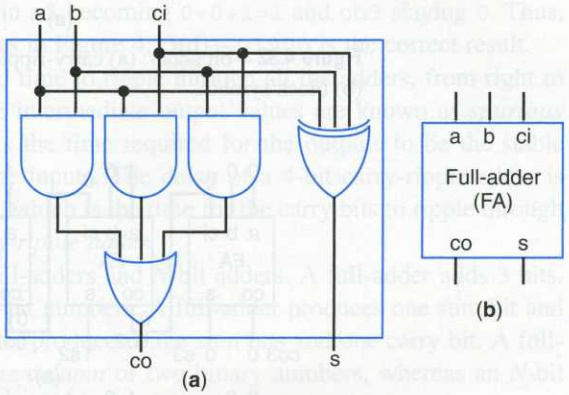


Figure 4.30 Full-adder: (a) circuit, (b) block symbol.

4-Bit Carry-Ripple Adder

Three full-adders and one half-adder can be connected as in Figure 4.31 to implement a 4-bit ***carry-ripple adder***, which adds two 4-bit numbers and generates a 4-bit sum. The 4-bit carry-ripple adder also generates a carry-out bit.

A carry-in bit can be included on the 4-bit adder, which enables connecting 4-bit adders together to build larger adders. The carry-in bit is included by replacing the half-adder (which was in the rightmost bit position) by a full-adder, as in Figure 4.32.

Let's analyze the behavior of this adder. Suppose that all inputs have been 0s for a long time, meaning that S will be 0000, co will be 0, and all ci values of the full adders will also be 0. Now suppose that A becomes 0111 and B becomes 0001 at the same time (whose sum we know should be 01000). Those new values of A and B will propagate through the full-adders. Suppose the delay of a full-adder is 2 ns. So 2 ns after A and B change, the sum outputs of all the full-adders will change, as shown in Figure 4.33(a). So s3 will become 0+0+0=0 (with co3=0), s2 will become 1+0+0=1 (with co2=0), s1 will become 1+0+0=1 (with co1=0), and s0 will become 1+1=0 (with co0=1). But, 1111 + 0110 should *not* be 00110—instead, the sum should be 01000. What went wrong?

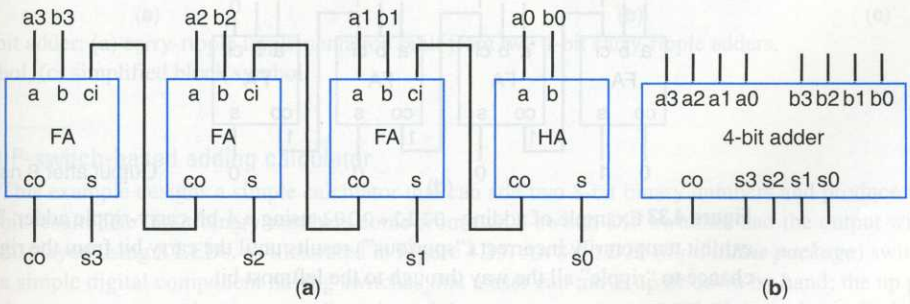


Figure 4.31 4-bit adder: (a) carry-ripple design with 3 full-adders and 1 half-adder, (b) block symbol.

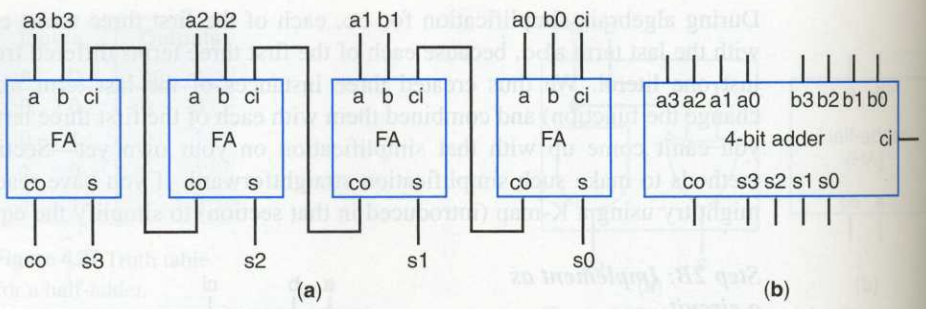


Figure 4.32 4-bit adder: (a) carry-ripple implementation with 4 full-adders and a carry-in input, (b) block symbol.

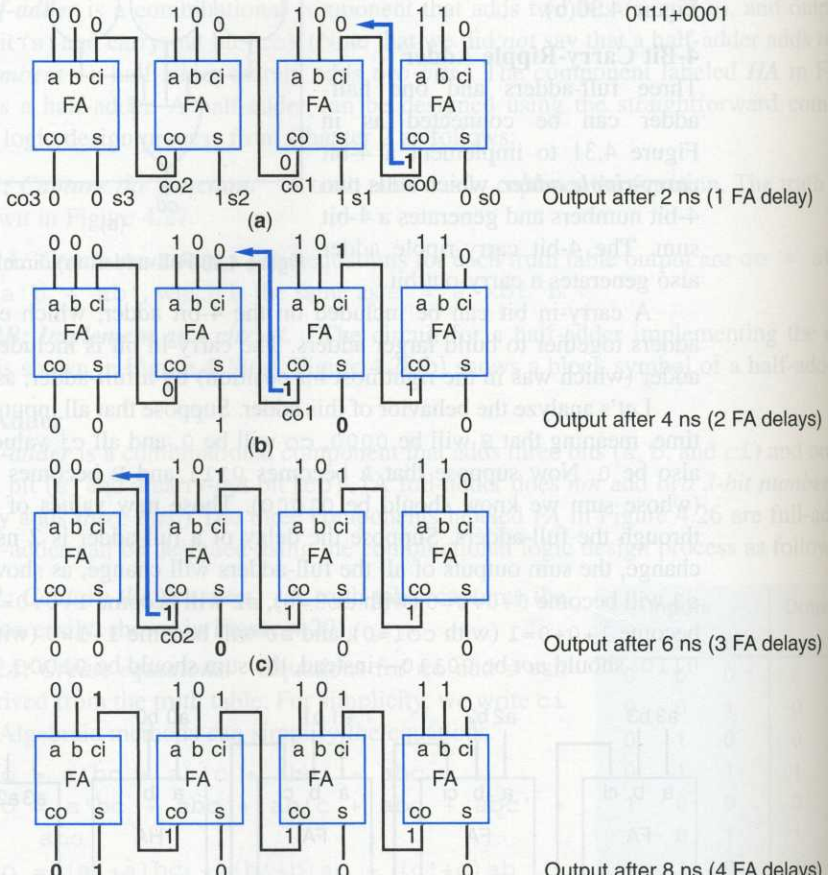
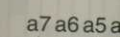
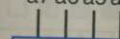
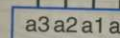


Figure 4.33 Example of adding `0111+0001` using a 4-bit carry-ripple adder. The output will exhibit temporarily incorrect ("spurious") results until the carry bit from the rightmost bit has had a chance to "ripple" all the way through to the leftmost bit.

The term "ripple-carry" adder is actually more common. I prefer the term "carry-ripple" for consistent naming with other adder types, like carry-select and carry-lookahead, which Chapter 6 describes.







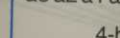


Figure 4.34
(b) block sy

Example 4.6

Nothing went wrong—the carry-ripple adder simply isn't done yet after just 2 ns. After 2 ns, `co0` changed from `0` to `1`. We must allow time for that *new* value of `co0` to proceed through the next full-adder. Thus, after another 2 ns, `s1` will equal `1+0+1=0` and `co2` will become `1`. So after 4 ns the output will be `00100` as shown in Figure 4.33(b).

Keep waiting. After a third full-adder delay, the new value of `co2` will have propagated through the next full-adder, resulting in `s2` becoming `1+0+1=0` and `co2` becoming `1`. So after 6 ns, the output will be `00000` as shown in Figure 4.33(c).

A little more patience. After a fourth full-adder delay, `co2` has had time to propagate through the last full-adder, resulting in `s3` becoming `0+0+1=1` and `co3` staying `0`. Thus, after 8 ns the output will be `01000` as in Figure 4.33(d)—`01000` is the correct result.

To recap, until the carry bits had time to ripple through all the adders, from right to left, the output was not correct. The intermediate output values are known as ***spurious values***. The ***delay*** of a component is the time required for the outputs to be the stable correct value after any change on the inputs. The delay of a 4-bit carry-ripple adder is equal to the delay of four full-adders, which is the time for the carry bits to ripple through all the adders—hence, the term *carry-ripple adder.*

The term "ripple-carry" adder is actually more common. I prefer the term "carry-ripple" for consistent naming with other adder types, like carry-select and carry-lookahead, which Chapter 6 describes.

People often initially confuse full-adders and *N*-bit adders. A full-adder adds 3 bits. In contrast, a 3-bit adder adds two 3-bit numbers. A full-adder produces one sum bit and one carry bit. In contrast, a 3-bit adder produces three sum bits and one carry bit. A full-adder is usually used to add only *one column* of two binary numbers, whereas an *N*-bit adder is used to add two *N*-bit numbers.

An *N*-bit adder often comes with a carry-in bit, so that the adder can be cascaded with other *N*-bit adders to form larger adders. Figure 4.34(a) shows an 8-bit adder built from two 4-bit adders. The carry-in bit (`ci`) on the right would be set to `0` when adding two 8-bit numbers. Figure 4.34(b) shows a block symbol of that 8-bit adder, and Figure 4.34(c) shows a simplified block symbol that is commonly used.

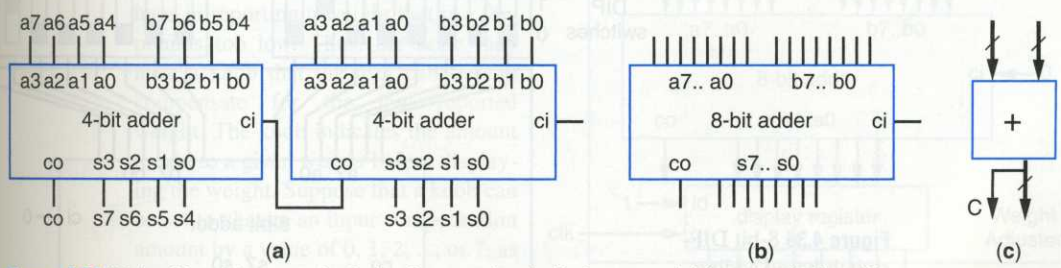


Figure 4.34 8-bit adder: (a) carry-ripple implementation built from two 4-bit carry-ripple adders, (b) block symbol, (c) simplified block symbol.

Example 4.6 DIP-switch-based adding calculator

This example designs a simple calculator that can add two 8-bit binary numbers and produce an 8-bit result. The input binary numbers come from two 8-switch DIP switches and the output will be displayed using 8 LEDs, as illustrated in Figure 4.35. An 8-bit ***DIP*** (***dual inline package***) switch is a simple digital component having switches that a user can move up or down by hand; the up position outputs a `1` on the corresponding pin and down outputs a `0`. An ***LED*** (light-emitting diode) is a small light that illuminates when the LED's input is `1`, and is dark when the input is `0`.

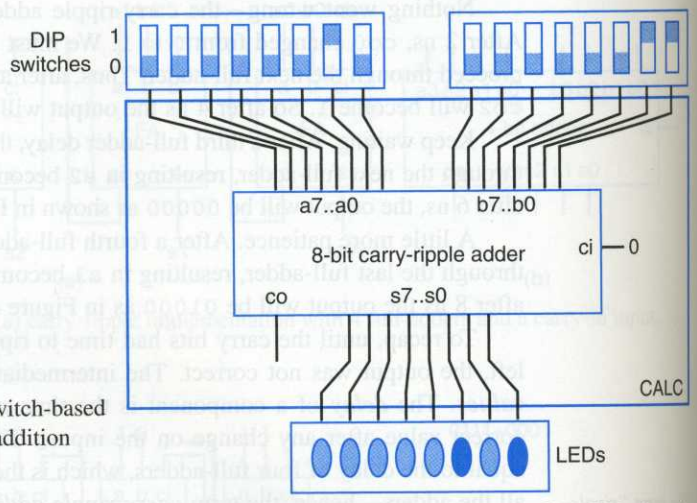


Figure 4.35 8-bit DIP-switch-based adding calculator. The addition 2+3=5 is shown.

The calculator can be implemented using an 8-bit carry-ripple adder for the CALC block, as in Figure 4.35. When a user moves the switches on a DIP switch, the new binary values propagate through the carry-ripple adder's gates, generating spurious values until the carry bits have finally propagated through the entire circuit, at which point the output stabilizes and the LEDs display the correct new sum. The spurious values are likely too fast to be visible on the LEDs by humans.

To avoid the LEDs changing while the user is moving switches, we can introduce a button `e` (for "equals") that indicates when the result should be displayed. The user presses `e` after configuring both DIP switches as the new inputs to be summed. We can connect the `e` input to the `load` input of a parallel load register as in Figure 4.36. When a user moves switches on the DIP switches, intermittent values appear at the adder outputs, but are blocked at the register's inputs, as the register holds its previous value and hence the LEDs display that value. When the `e` button is pressed, then on the next clock edge the register will be loaded, and the LEDs will display the new result.

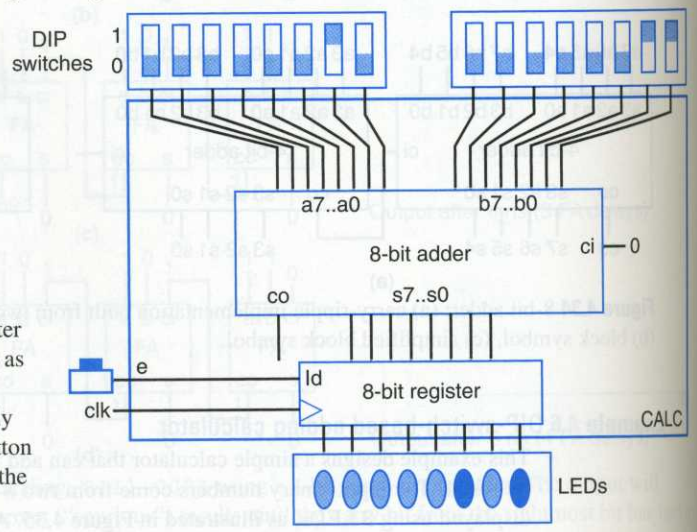
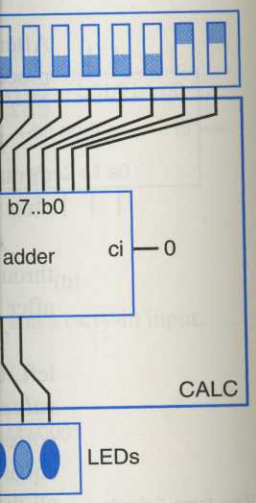
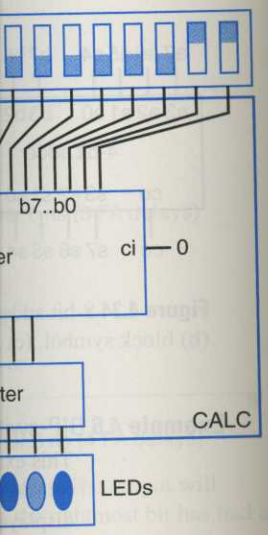


Figure 4.36 8-bit DIP-switch-based adding calculator, using a register to block output changes as the user configures the switches. The LEDs only get updated after the button is pressed, which loads the output register.

Notice that the displayed value will be correct only if the sum is 255 or less. We could connect `co` to a ninth LED to display sums between 256 and 511.



for the CALC block, as in
w binary values propagate
the carry bits have finally
s and the LEDs display the
the LEDs by humans.
e can introduce a button e
ıser presses e after config-
ct the e input to the load
itches on the DIP switches,
egister's inputs, as the reg-
ıen the e button is pressed,
ill display the new result.



55 or less. We could connect

Delay and Size of a 32-Bit Carry-Ripple Adder

Assuming full-adders are implemented using two levels of gates (ANDs followed by an OR) and that every gate has a delay of "1 gate-delay," let's compute the total delay of a 32-bit carry-ripple adder, and also compute the size of such an adder.

To determine the delay, note that the carry must ripple from the first full-adder to the 32nd full-adder (referring to the adder in Figure 4.33 may help). The delay of the first full-adder is 2 gate-delays. The new carry must then ripple through the second full-adder, resulting in another 2 gate-delays. And so on. Thus, the total delay of the 32-bit carry-ripple adder is 64 gate-delays. Supposing a gate-delay is 1 ns, then the total delay would be 64 ns.

To determine the size, note that the full-adder in Figure 4.30 would require about 30 transistors: about 12 transistors for the three 2-input AND gates (4 transistors each), 6 transistors for the 3-input OR gate, and 12 for the 3-input XOR gate. Because the 32-bit adder uses 32 full-adders, the total size of the 32-bit carry-ripple adder is (12 transistors/full-adder)*(32 full-adders)=384 transistors. That's a lot less than the 100 billion transistors predicted from the data in Figure 4.24.

The 32-bit carry-ripple adder has a long delay but a reasonable number of transistors. Section 6.4 shows to build faster adders at the expense of using some more transistors.

Example 4.7 Compensating weight scale using an adder

A scale, such as a bathroom scale, uses a sensor to determine the weight of an object (e.g., a person) on the scale. The sensor's readings for the same object may change over time due to wear and tear on the sensing system (such as a spring losing elasticity), resulting perhaps in reporting a weight that is a few pounds too low. Thus, the scale may have a knob that the user can turn to compensate for the low reported weight. The knob indicates the amount to add to a given weight before displaying the weight. Suppose that a knob can be set to change an input compensation amount by a value of 0, 1, 2, ..., or 7, as shown in Figure 4.37.

We can implement the system using an 8-bit carry-ripple adder as shown in the figure. On every rising clock edge, the display register will be loaded with the sum of the currently sensed weight plus the compensation amount.

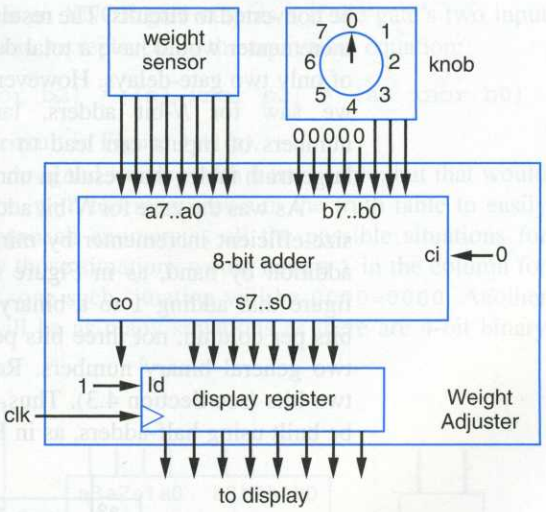


Figure 4.37 Compensating scale: the dial outputs a number from 0 to 7 (000 to 111), which gets added to the sensed weight and then displayed.

Incrementer

Sometimes a designer needs to just add a constant "1" to a number, rather than adding two general numbers. A ***constant*** is a number that does not change in a circuit. For example, a designer may want a register to count up from 0 to 255, which involves adding 1 to the register's current value and loading the result back into the register. The designer could use a carry-ripple adder to perform the addition, but an adder is designed to add any two numbers and thus has more gates than necessary to just add 1. A common component is thus an adder specifically designed just to add 1 to a number, known as an ***incrementer***.

An incrementer can be designed using the combinational logic design process from Chapter 2. Design starts with the truth table shown in Figure 4.38. Each output row's values can be obtained simply by adding `1` to the corresponding input row binary number. We would then derive an equation for each output. It is easy to see from the table that the equation for `c0` is `c0=a3a2a1a0`. It is also easy to see that `s0=a0'`. We would derive equations for the remaining outputs. Finally, the equations would be converted to circuits. The resulting incrementer would have a total delay of only two gate-delays. However, as we saw for *N*-bit adders, larger numbers of inputs can lead to very large truth tables that result in unreasonably large circuits.

Inputs				Outputs				
a3	a2	a1	a0	c0	s3	s2	s1	s0
0	0	0	0	0	0	0	0	1
0	0	0	1	0	0	0	1	0
0	0	1	0	0	0	0	1	1
0	0	1	1	0	0	1	0	0
0	1	0	0	0	0	1	0	1
0	1	0	1	0	0	1	1	0
0	1	1	0	0	0	1	1	1
0	1	1	1	0	1	0	0	0
1	0	0	0	0	1	0	0	1
1	0	0	1	0	1	0	1	0
1	0	1	0	0	1	0	1	1
1	0	1	1	0	1	1	0	0
1	1	0	0	0	1	1	0	1
1	1	0	1	0	1	1	1	0
1	1	1	0	0	1	1	1	1
1	1	1	1	1	0	0	0	0

Figure 4.38 Truth table for four-bit incrementer.

As was the case for *N*-bit adders, we could design a more size-efficient incrementer by mimicking the way humans do addition by hand, as in Figure 4.39. However, note in the figure that adding `1` to a binary number involves only two bits per column, not three bits per column like when adding two general binary numbers. Recall that a half-adder adds two bits (see Section 4.3). Thus, a simple incrementer could be built using half-adders, as in Figure 4.40.

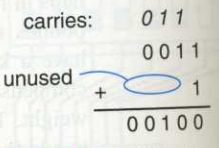


Figure 4.39 Adding 1 to a binary number requires only 2 bits per column.

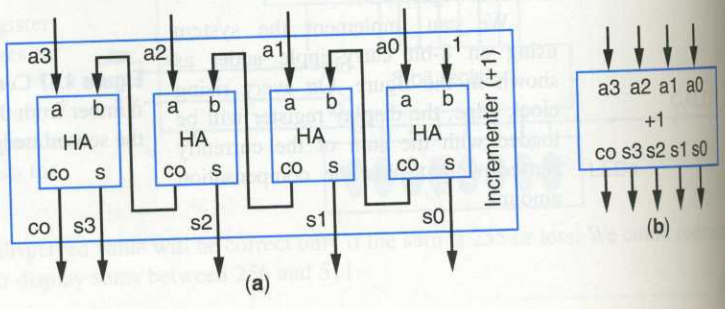


Figure 4.40 4-bit incrementer: (a) internal design, (b) block symbol.

▶ 4.4 COMPARATORS

Designs often need their circuit to compare two binary numbers to determine if the numbers are equal or if one number is greater than the other. For example, a system might sound an alarm if a thermometer measuring human body temperature reports a temperature greater than 103 degrees Fahrenheit (39.4 degrees Celsius). Comparator components perform such comparison of binary numbers.

Equality (Identity) Comparator

An ***N-bit equality comparator*** (sometimes called an ***identity comparator***) is a combinational component that compares two *N*-bit data inputs `A` and `B`, and sets an output control bit `eq` to `1` if those two data inputs are equal. Two *N*-bit inputs, such as two 4-bit inputs `A:a3a2a1a0` and `B:b3b2b1b0`, are equal if each of their corresponding bit pairs are equal. So `A` equals `B` if `a3=b3`, `a2=b2`, `a1=b1`, and `a0=b0`. For example if `A` is `1011` and `B` is `1011`, then `A` equals `B`.

Following the combinational logic design process of Table 2.5, a 4-bit equality comparator can be designed by first capturing the function as an equation:

```
eq = (a3b3+a3'b3') * (a2b2+a2'b2') *
     (a1b1+a1'b1') * (a0b0+a0'b0')
```

Each term detects if the corresponding bits are equal, namely if both bits are `1` or both bits are `0`. The expressions inside each of the parentheses describe the behavior of an XNOR gate (recall from Chapter 2 that an XNOR gate outputs `1` if the gate's two input bits are equal), so the above equation can be replaced by the equivalent equation:

```
eq = (a3 xnor b3) * (a2 xnor b2) * (a1 xnor b1) * (a0 xnor b0)
```

The equation can be converted to the circuit in Figure 4.41(a).

Of course, a comparator could be designed starting with a truth table, but that would be cumbersome for a large comparator, with too many rows in the truth table to easily work with by hand. A truth table approach enumerates all the possible situations for which all the bits are equal, since only those situations would have a `1` in the column for the output `eq`. For two 4-bit numbers, one such situation will be `0000=0000`. Another will be `0001=0001`. Clearly, there will be as many situations as there are 4-bit binary

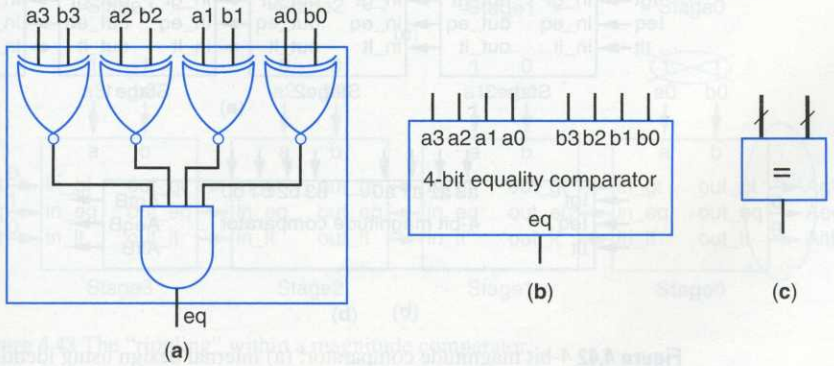
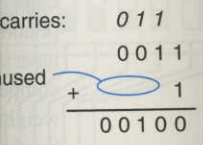


Figure 4.41 Equality comparator: (a) internal design, (b) block symbol, (c) simplified symbol.

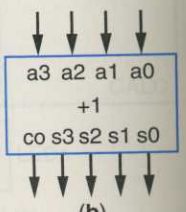
ber, rather than adding
ange in a circuit. For
which involves adding
register. The designer
is designed to add any
A common component
wn as an ***incrementer***.

Outputs			
s3	s2	s1	s0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0
1	0	0	1
1	0	1	0
1	0	1	1
1	1	0	0
1	1	0	1
1	1	1	0
1	1	1	1
0	0	0	0

four-bit incrementer.



gure 4.39 Adding 1 to a
nary number requires
ly 2 bits per column.



numbers—meaning there will be 2^4=16 situations where both numbers are equal. For two 8-bit numbers, there will be 256 equal situations. For two 32-bit numbers, there will be four billion equal situations. A comparator built with such an approach will be large if we don't minimize the equation, and that minimization will be hard with such large numbers of terms. The XNOR-based design is simpler and scales to wider inputs easily—widening the inputs from four bits to five bits involves merely adding one more XNOR gate to Figure 4.41(a).

Magnitude Comparator—Carry-Ripple Style

An ***N-bit magnitude comparator*** is a combinational component that compares two *N*-bit data inputs `A` and `B` representing binary numbers, and outputs whether `A>B`, `A=B`, or `A<B` using three control signals `AgtB`, `AeqB`, and `AltB`.

We have already seen several times that designing certain datapath components by starting with a truth table involves too large of a truth table. Let's instead design a magnitude comparator by considering how humans compare numbers by hand. Consider comparing two 4-bit numbers `A:a3a2a1a0=1011`, `B:b3b2b1b0=1001`. We start by looking at the high-order bits of `A` and `B`, namely, `a3` and `b3`. Since they are equal (both are `1`), we look at the next pair of bits, `a2` and `b2`. Again, since they are equal (both are `0`), we look at the next pair of bits, `a1` and `b1`. Since `a1>b1` (`1>0`), we conclude that `A>B`.

Thus, comparing two binary numbers takes place by comparing from the high bit-pairs down to the low bit-pairs. As long as bit-pairs are equal, comparison continues with the next lower bit-pair. As soon as a bit-pair is different, a conclusion can be made that `A>B` if `ai=1` and `bi=0`, or that `A<B` if `bi=1` and `ai=0`. Based on this comparison concept, we can design a magnitude comparator using the structure shown in Figure 4.42(a).

Each stage works as follows. If `in_gt=1` (meaning a higher stage determined `A>B`), this stage need not compare bits and instead just sets `out_gt=1`. Likewise, if `in_lt=1` (meaning a higher stage determined `A<B`), this stage just sets `out_lt=1`. If `in_eq=1` (meaning higher stages were all equal), this stage must compare bits, setting the output `out_gt=1` if `a=1` and `b=0`, setting `out_lt=1` if `a=0` and `b=1`, and setting `out_eq=1` if `a` and `b` both equal `1` or both equal `0`.

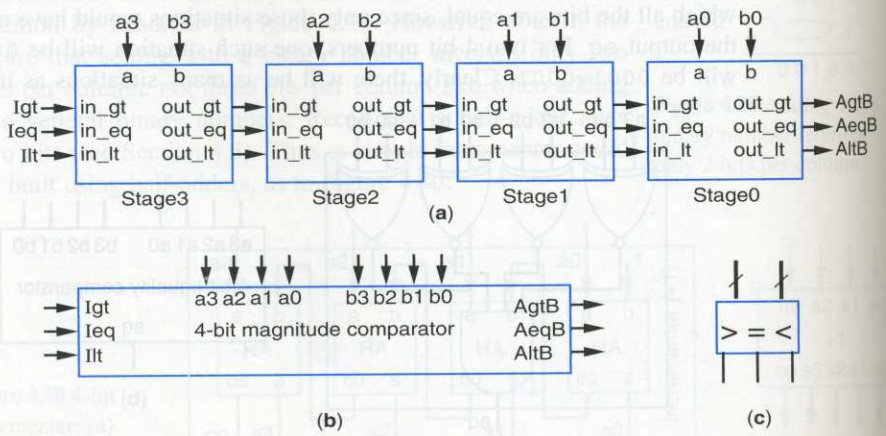


Figure 4.42 4-bit magnitude comparator: (a) internal design using identical components in each stage, (b) block symbol, (c) simplified symbol without ripple inputs.

We could capture the function of a stage's block using a truth table with 5 inputs. A simpler way is to capture the function as the following equations derived from the above explanation of how each stage works; the circuit for each stage would follow directly from these equations:

```
out_gt = in_gt + (in_eq * a * b')
out_lt = in_lt + (in_eq * a' * b)
out_eq = in_eq * (a XNOR b)
```

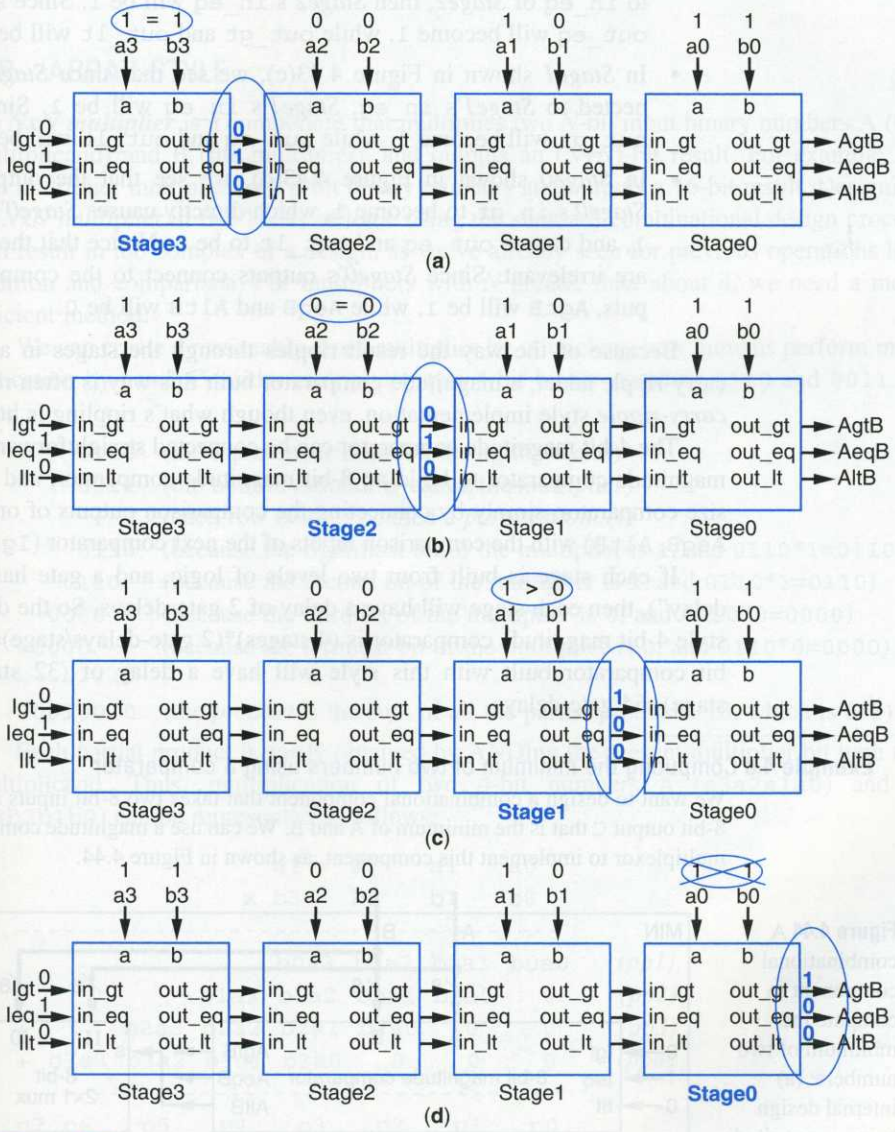


Figure 4.43 The "rippling" within a magnitude comparator.

Figure 4.43 shows how this comparator works for an input of A=1011 and B=1001. We can view the comparator's behavior as consisting of four stages:

- In *Stage3* shown in Figure 4.43(a), we start by setting the external input Ieq=1, to force the comparator to actually do the comparison. *Stage3* has in_eq=1, and since a3=1 and b3=1, then out_eq will become 1, while out_gt and out_lt will become 0.
- In *Stage2* shown in Figure 4.43(b), we see that since out_eq of *Stage3* connects to in_eq of *Stage2*, then *Stage2*'s in_eq will be 1. Since a2=0 and b2=0, then out_eq will become 1, while out_gt and out_lt will be 0.
- In *Stage1* shown in Figure 4.43(c), we see that since *Stage2*'s out_eq is connected to *Stage1*'s in_eq, *Stage1*'s in_eq will be 1. Since a1=1 and b1=0, out_gt will become 1, while out_eq and out_lt will be 0.
- In *Stage0* shown in Figure 4.43(d), we see that the outputs of *Stage1* cause *Stage0*'s in_gt to become 1, which directly causes *Stage0*'s out_gt to become 1, and causes out_eq and out_lt to be 0. Notice that the values of a0 and b0 are irrelevant. Since *Stage0*'s outputs connect to the comparator's external outputs, AgtB will be 1, while AeqB and AltB will be 0.

Because of the way the result ripples through the stages in a manner similar to a carry-ripple adder, a magnitude comparator built this way is often referred to as having a *carry-ripple* style implementation, even though what's rippling is not really a "carry" bit.

The 4-bit magnitude comparator can be connected straightforwardly with another 4-bit magnitude comparator to build an 8-bit magnitude comparator, and likewise to build any size comparator, simply by connecting the comparison outputs of one comparator (AgtB, AeqB, AltB) with the comparison inputs of the next comparator (Igt, Ieq, Ilt).

If each stage is built from two levels of logic, and a gate has a delay of "1 gate-delay"), then each stage will have a delay of 2 gate-delays. So the delay of a carry-ripple style 4-bit magnitude comparator is (4 stages)*(2 gate-delays/stage)=8 gate-delays. A 32-bit comparator built with this style will have a delay of (32 stages)*(2 gate-delays/stage)=64 gate-delays.

Example 4.8 Computing the minimum of two numbers using a comparator

We want to design a combinational component that takes two 8-bit inputs A and B, and outputs an 8-bit output C that is the minimum of A and B. We can use a magnitude comparator and an 8-bit 2x1 multiplexor to implement this component, as shown in Figure 4.44.

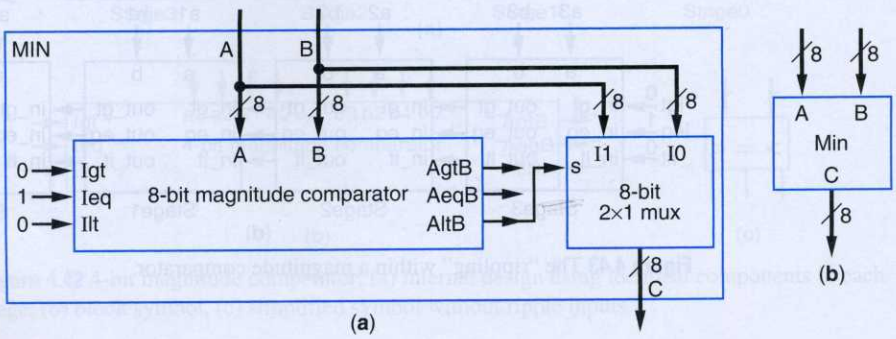


Figure 4.44 A combinational component to compute the minimum of two numbers: (a) internal design using a magnitude comparator, (b) block symbol.

If A<B, the comparator's AltB output will be 1. In this case, we want to pass A through the mux, so we connect AltB to the 8-bit 2x1 mux select input, and A to the mux's I1 input. If AltB is 0, then either AgtB=1 or AeqB=1. If AgtB=1, we want to pass B. If AeqB=1, we can pass either A or B (since they are identical), and so let's pass B. We thus simply connect B to the I0 input of the 8-bit 2x1 mux. In other words, if A<B, we'll pass A, and if A is not less than B, we'll pass B.

Notice that we set the comparator's Ieq control input to 1, and the Igt and Ilt inputs to 0. These values force the comparator to compare its data inputs.

4.5 MULTIPLIER—ARRAY-STYLE

An ***NxN multiplier*** is a component that multiplies two *N*-bit input binary numbers A (the multiplicand) and B (the multiplier), and outputs an (*N*+*N*)-bit result. For example, an 8x8 multiplier multiplies two 8-bit binary numbers and outputs a 16-bit result. Designing an *N*x*N* multiplier in two levels of logic using the standard combinational design process will result in too complex of a design, as we've already seen for previous operations like addition and comparison. For multipliers with *N* greater than about 4, we need a more efficient method.

We can create a reasonably sized multiplier by mimicking how humans perform multiplication by hand. Consider multiplying two 4-bit binary numbers 0110 and 0011 by hand:

```
  0110   (the top number is called the multiplicand)
  0011   (the bottom number is called the multiplier)
  ----   (each row below is called a partial product)
  0110   (because the rightmost bit of the multiplier is 1, and 0110*1=0110)
 0110    (because the second bit of the multiplier is 1, and 0110*1=0110)
0000     (because the third bit of the multiplier is 0, and 0110*0=0000)
+0000    (because the leftmost bit of the multiplier is 0, and 0110*0=0000)
--------
00010010 (the product is the sum of all the partial products: 18, which is 6*3)
```

Each partial product is easily obtained by ANDing the present multiplier bit with the multiplicand. Thus, multiplication of two 4-bit numbers A (a3a2a1a0) and B (b3b2b1b0) can be represented as follows:

```
                        a3   a2   a1   a0
                      x b3   b2   b1   b0
-------------------------------------------
                        b0a3 b0a2 b0a1 b0a0   (pp1)
+                  b1a3 b1a2 b1a1 b1a0   0    (pp2)
+             b2a3 b2a2 b2a1 b2a0   0    0    (pp3)
+ b3a3       b3a2 b3a1 b3a0   0    0    0    (pp4)
-------------------------------------------
p7 p6   p5   p4   p3   p2   p1   p0
```

Note that b0a0 means b0 AND a0. After generating the partial products (*pp1*, *pp2*, *pp3*, and *pp4*) by ANDing the present multiplier bit with each multiplicand bit, we merely

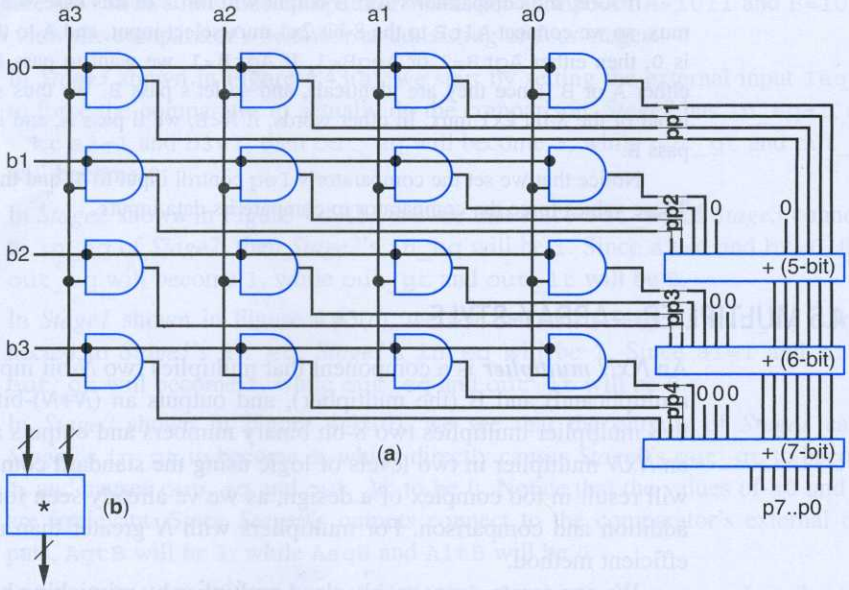


Figure 4.45 4-bit by 4-bit array-style multiplier: (a) internal design, (b) simplified block symbol.

need to sum those partial products together. We can use three adders of varying widths for computing that sum. The resulting design is shown in Figure 4.45(a).

This design for this 4-bit multiplier has a reasonable size, being about three times bigger than a 4-bit carry-ripple adder. The design has reasonable speed. The delay consists of 1 gate-delay for generating the partial products, plus the delay of the adders. If each adder is a carry-ripple adder, then the 5-bit adder delay will be 5*2=10 gate-delays, the 6-bit adder delay will be 6*2=12 gate-delays, and the 7-bit adder delay will be 7*2=14 gate-delays. If we assume that the total delay of the adders is simply the sum of the adder delays, then the total delay would thus be 1+10+12+14=37 gate-delays. However, the total delay of carry-ripple adders when chained together is actually a little less than the sum of delays—see Exercise 4.15.

Delays for larger multipliers, which will have an even longer chain of adders, will be even slower. Faster multiplier designs are possible at the expense of more gates.

▶ 4.6 SUBTRACTORS AND SIGNED NUMBERS

An *N*-bit ***subtractor*** is a combinational component that takes two *N*-bit data inputs `A` and `B`, representing binary numbers, and outputs an *N*-bit result `S` equaling `A-B`.

Subtractor for Positive Numbers Only

For now, let's assume we are only dealing with positive numbers, so the subtractor's inputs are positive, and the result is always positive. Subtraction gets slightly more complex when negative results are considered, like 5–7= –2, and thus far we haven't discussed representation of negative numbers. The result of subtraction always being positive could be the case, for example, when a system only subtracts smaller numbers

from larger numbers, such as when compensating a sampled temperature that will always be greater than 80 using a small compensation value that will always be less than 10.

Designing an *N*-bit subtractor using the standard combinational logic design process suffers from the same exponential size growth problem as an *N*-bit adder, as discussed in Section 4.3. Instead, we can again build an efficient component by mimicking subtraction by hand.

Figure 4.46(a) shows subtraction of 4-bit binary numbers "by hand." Starting with the first column, `a` is less than `b` (`0 < 1`), necessitating a borrow from the previous column. The first column result is then `10-1=1` (stated in base ten: two minus one equals one). The second column has a `0` for `a` because of the borrow by the first column, making `a < b` (`0 < 1`), generating a borrow from the third column—which must itself borrow from the fourth column. The result of the second column is then `10-1=1`. The third column, because of the borrow generated by the second column, has an `a` of `1`, which is not less than `b`, so the result of the third column is `1-1=0`. The fourth column has `a=0` due to the borrow from the third column, and since `b` is also 0, the result is `0-0=0`.

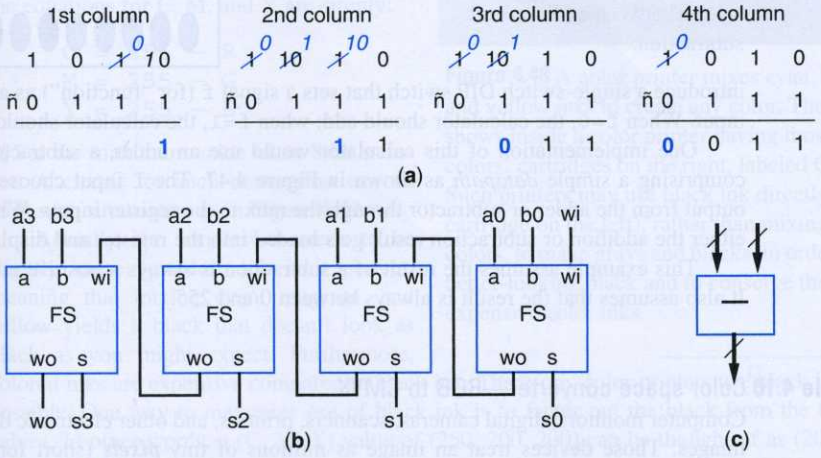


Figure 4.46 Design of a 4-bit subtractor: (a) subtraction "by hand", (b) borrow-ripple implementation with four full-subtractors with a borrow-in input `wi`, (c) simplified block symbol.

Based on the above-described behavior, we could connect four "full-subtractors" in a ripple manner as shown in Figure 4.46(b), similar to a carry-ripple adder. A ***full-subtractor*** has input `wi` representing a borrow by the previous column, an output `wo` representing a borrow from the next column, and of course inputs `a` and `b` and output `s`. We use `w`'s for the borrows rather than `b`'s because `b` is already used for the input; the `w` comes from the end of the word "borrow." We leave the design of a full-subtractor as an exercise for the reader.

Example 4.9 DIP-switch-based adding/subtracting calculator

In Example 4.6, we designed a simple calculator that could add two 8-bit binary numbers and produce an 8-bit result, using DIP switches for inputs, and a register plus LEDs for output. Let's extend that calculator to allow the user to choose among addition and subtraction operations. We'll

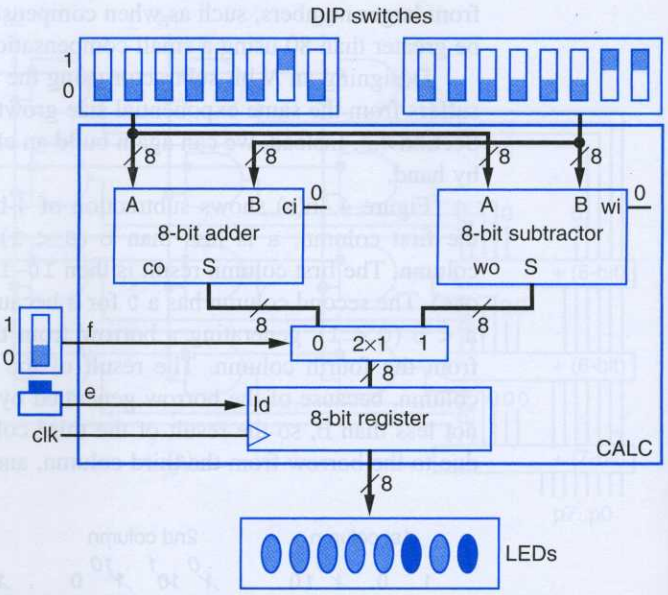


Figure 4.47 8-bit DIP-switch-based adding/subtracting calculator. Control input `f` selects between addition and subtraction.

introduce a single-switch DIP switch that sets a signal `f` (for "function") as another system control input. When `f=0`, the calculator should add; when `f=1`, the calculator should subtract.

One implementation of this calculator would use an adder, a subtractor, and a multiplexor, comprising a simple *datapath* as shown in Figure 4.47. The `f` input chooses whether to pass the output from the adder or subtractor through the mux to the register inputs. When the user presses `e`, either the addition or subtraction result gets loaded into the register and displayed at the LEDs.

This example assumes the result of a subtraction is always a positive number, never negative. It also assumes that the result is always between 0 and 255.

Example 4.10 Color space converter—RGB to CMYK

Computer monitors, digital cameras, scanners, printers, and other electronic devices deal with color images. Those devices treat an image as millions of tiny ***pixels*** (short for "picture elements"), which are indivisible dots representing a tiny part of the image. Each pixel has a color, so an image is just a collection of colored pixels. A good computer monitor may support over 10 million unique colors for each pixel. How does a monitor create each unique color for a pixel? In a method used in what are known as RGB monitors, the monitor has three light sources inside—red, green, and blue. Any color of light can be created by adding specific intensities of each of the three colors. Thus, for each pixel, the monitor shines a specific intensity of red, of green, and of blue at that pixel's location on the monitor's screen, so that the three colors add together to create the desired pixel color. Each subcolor (red, green, or blue) is typically represented as an 8-bit binary number (thus each ranging from 0 to 255), meaning a color is represented by 8+8+8=24 bits. An (R, G, B) value of (0, 0, 0) represents black. (10, 10, 10) represents a very dark gray, while (200, 200, 200) represents a light gray. (255, 0, 0) represents red, while (100, 0, 0) represents a darker (nonintense) red. (255, 255, 255) represents white. (109, 35, 201) represents some mixture of the three base colors. Representing color using intensity values for red, green, and blue is known as an ***RGB color space.***

RGB color space is great for computer monitors and certain other devices, but not the best for some other devices like printers. Mixing red, green, and blue ink on paper will not result in white, but rather in black. Why? Because ink is not light; rather, ink reflects light. So red ink reflects red light, absorbing green and blue light. Likewise, green ink absorbs red and blue light. Blue ink

absorbs red and green light. Mix all three inks together on paper, and the mixture absorbs all light, reflecting none, thus yielding black. Printers therefore use a different color space based on the complementary colors of red/green/blue, namely, cyan/magenta/yellow, known as a ***CMY color space***. Cyan ink *absorbs* red, reflecting green and blue (the mixture of which is cyan). Magenta ink *absorbs* green light, reflecting red and blue (which is magenta). Yellow ink *absorbs* blue, reflecting red and green (which is yellow).

A color printer commonly has three color ink cartridges, one cyan, one magenta, and one yellow. Figure 4.48 shows the ink cartridges for a particular color printer. Some printers have a single cartridge for color instead of three, with that single cartridge internally containing separated fluid compartments for the three colors.

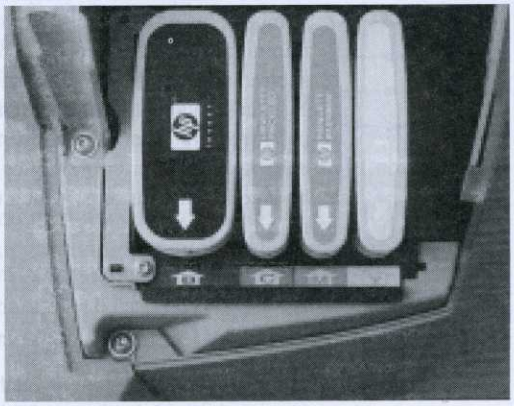


Figure 4.48 A color printer mixes cyan, magenta, and yellow inks to create any color. The picture shows inside a color printer having those three colors' cartridges on the right, labeled C, M, and Y. Such printers may use black ink directly (the big cartridge on the left), rather than mixing the three colors, to make grays and blacks, in order to create a better-looking black and to conserve the more expensive color inks.

A printer must convert a received RGB image into CMY. Let's design a fast circuit to perform that conversion. Given three 8-bit values for R, G, and B for a given pixel, the equations for C, M, and Y are simply:

```
C = 255 - R
M = 255 - G
Y = 255 - B
```

255 is the maximum value of an 8-bit number. A circuit for such conversion can be built with subtractors as in Figure 4.49.

Actually, the conversion needs to be slightly more complex. Ink isn't perfect, meaning that mixing cyan, magenta, and yellow yields a black that doesn't look as black as you might expect. Furthermore, colored inks are expensive compared to black ink. Therefore, color printers use black ink whenever possible. One way to maximize use of black ink is to factor out the black from the C, M, and Y values. In other words, a (C, M, Y) value of (250, 200, 200) can be thought of as (200, 200, 200) plus (50, 0, 0).

The (200, 200, 200), which is a light gray, can be generated using black ink. The remaining (50, 0, 0) can be generated using a small amount of cyan, and using no magenta or yellow ink at all, thus saving precious color ink. A CMY color space extended with black is known as a ***CMYK color space*** (the "K" comes from the last letter in the word "black." "K" is used instead of "B" to avoid confusion with the "B" from "blue").

An RGB to CMYK converter can thus be described as:

```
K = Minimum (C, M, Y)
C2 = C - K
M2 = M - K
Y2 = Y - K
```

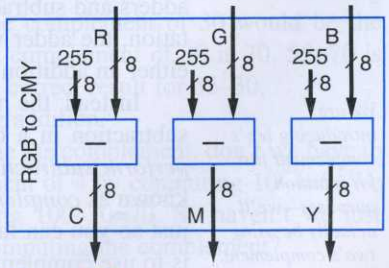


Figure 4.49 RGB to CMY converter.

C, M, and Y are defined as earlier. We thus create the *datapath* circuit in Figure 4.50 for converting an RGB color space to a CMYK color space. We've used the *RGBtoCMY* component from Figure 4.49. We've also used two instances of the MIN component that we created in Example 4.8 to compute the minimum of two numbers; using two such components computes the minimum of three numbers. Finally, we use three more subtractors to remove the K value from the C, M, and Y values. In a real printer, the imperfections of ink and paper require even more adjustments. A more realistic color space converter multiplies the R, G, and B values by a series of constants, which can be described using matrices:

$$\begin{vmatrix} C \\ M \\ Y \end{vmatrix} = \begin{vmatrix} m00 & m01 & m02 \\ m10 & m11 & m12 \\ m20 & m21 & m22 \end{vmatrix} * \begin{vmatrix} R \\ G \\ B \end{vmatrix}$$

Further discussion of such a matrix-based converter is beyond the scope of this example.

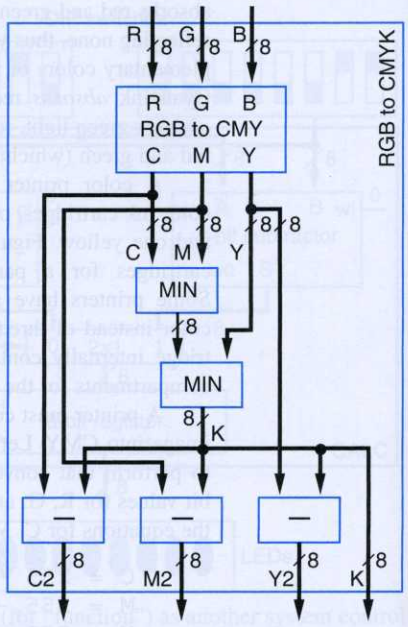


Figure 4.50 RGB to CMKY converter.

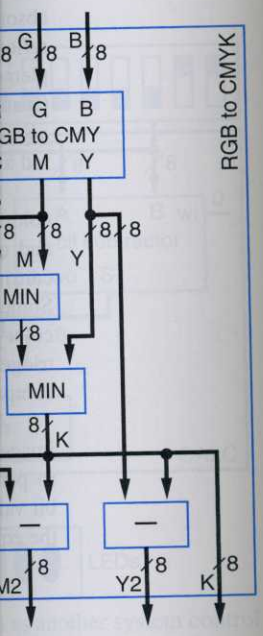
Representing Negative Numbers: Two's Complement Representation

The subtractor design in the previous section assumed positive input numbers and positive results. But in many systems, results may be negative, and in fact the input values may even be negative numbers. We thus need a way to represent negative numbers using bits.

One obvious but not very effective representation is known as ***signed-magnitude representation***. In this representation, the highest-order bit is used only to represent the number's sign, with `0` meaning positive and `1` meaning negative. The remaining low-order bits represent the magnitude of the number. In this representation, and using 4-bit numbers, `0111` would represent +7, while `1111` would represent –7. Thus, four bits could represent –7 to 7. (Notice, by the way, that both `0000` and `1000` would represent 0, the former representing `0`, the latter `-0`.) Signed-magnitude is easy for humans to understand, but doesn't lend itself easily to the design of simple arithmetic components like adders and subtractors. For example, if an adder's inputs use signed-magnitude representation, the adder would have to look at the highest-order bit and then internally perform either an addition or a subtraction, using different circuits for each.

We are introducing ten's complement just for intuition purposes—we'll actually be using two's complement.

Instead, the most common method of representing negative numbers and performing subtraction in a digital system actually uses a trick that allows us to *use an adder to perform subtraction*. The key to performing subtraction using addition lies in what are known as *complements*. We'll first introduce complements in the base ten number system just so you can familiarize yourself with the concept, but bear in mind that the intention is to use complements in base two, not base ten.



GB to CMKY converter.

input numbers and posi-
in fact the input values
t negative numbers using

s ***signed-magnitude rep-***
ed only to represent the
ive. The remaining low-
sentation, and using 4-bit
sent –7. Thus, four bits
`1000` would represent 0,
asy for humans to under-
ithmetic components like
gned-magnitude represen-
d then internally perform
ach.
numbers and performing
vs us to *use an adder to*
addition lies in what are
e base ten number system
in mind that the intention

Consider subtraction involving two single-digit base ten numbers, say 7–4. The result should be 3. Let's define the ***complement*** of a single-digit base ten number `A` as *the number that when added to* `A` *results in a sum of ten*. So the complement of 1 is 9, of 2 is 8, and so on. Figure 4.51 provides the complements for the numbers 1 through 9.

The wonderful thing about a complement is that you can use it to perform subtraction using addition, by replacing the number being subtracted with its complement, then by adding, and then by finally throwing away the carry. For example:

```
7 - 4 —> 7 + 6 = 13 —> ~~1~~3 = 3
```

We replaced 4 by its complement 6, and then added 6 to 7 to obtain 13. Finally, we threw away the carry, leaving 3, which is the correct result. Thus, *we performed subtraction using addition.*

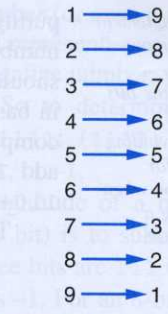
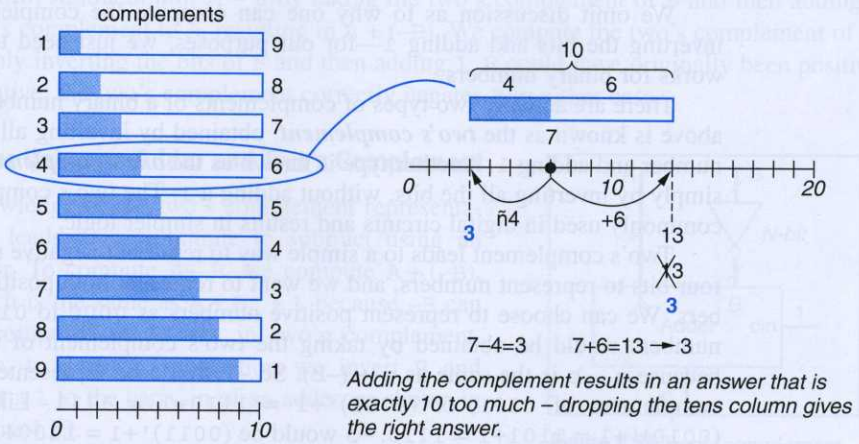


Figure 4.51 Complements in base ten.



Adding the complement results in an answer that is exactly 10 too much – dropping the tens column gives the right answer.

Figure 4.52 Subtracting by adding—subtracting a number (4) is the same as adding the number's complement (6) and then dropping the carry, since by definition of the complement, the result will be exactly 10 too much. After all, that's how the complement was defined—the number plus its complement equals 10.

A number line helps us visualize why complements work, as shown in Figure 4.52. Complements work for any number of digits. Say we want to perform subtraction using two two-digit base ten numbers, perhaps 55–30. The complement of 30 would be the number that when added to 30 results in 100, so the complement of 30 is 70. 55+70 is 125. Throwing away the carry yields 25, which is the correct result for 55–30.

So using complements achieves subtraction using addition.

"Not so fast!" you might say. In order to determine the complement, don't we have to perform subtraction? We know that 6 is the complement of 4 by computing 10–4=6. We know that 70 is the complement of 30 by computing 100–30=70. So haven't we just moved the subtraction to another step—the step of computing the complement?

Yes. Except, it turns out that *in base two, we can compute the complement in a much simpler way—just by inverting all the bits and adding 1*. For example, consider com-

Two's complement can be computed simply by inverting the bits and adding 1—thus avoiding the need for subtraction when computing a complement.

puting the complement of the 3-bit base-two number `001`. The complement would be the number that when added to `001` yields `1000`–you can probably see that the complement should be `111`. To check, we can use the same method for computing the complement as in base ten, computing the two's complement of `001` as `1000–001=111`. So `111` is the complement of `001`. However, it just so happens that if we invert all the bits of `001` and add `1`, we get the same result! Inverting the bits of `001` yields `110`; adding `1` yields `110+1=111`, which is the correct complement.

Thus, to perform a subtraction, say `011–001`, we would perform the following:

```
011 – 001
—> 011 + ((001)'+1)
= 011 + (110+1)
= 011 + 111
= 1010                    (throw away the carry)
—> 010
```

That's the correct answer, and the method didn't involve subtraction—only an invert and additions.

We omit discussion as to why one can compute the complement in base two by inverting the bits and adding `1`—for our purposes, we just need to know that the trick works for binary numbers.

There are actually two types of complements of a binary number. The type described above is known as the ***two's complement***, obtained by inverting all the bits of the binary number and adding `1`. Another type is known as the ***one's complement***, which is obtained simply by inverting all the bits, without adding a `1`. The two's complement is much more commonly used in digital circuits and results in simpler logic.

Two's complement leads to a simple way to represent negative numbers. Say we have four bits to represent numbers, and we want to represent both positive and negative numbers. We can choose to represent positive numbers as `0000` to `0111` (0 to 7). Negative numbers would be obtained by taking the two's complement of the positive numbers, because `a – b` is the same as `a + (–b)`. So –1 would be represented by taking the two's complement of `0001`, or `(0001)'+1 = 1110+1 = 1111`. Likewise, –2 would be `(0010)'+1 = 1101+1 = 1110`. –3 would be `(0011)'+1 = 1100+1 = 1101`. And so on. –7 would be `(0111)'+1 = 1000+1 = 1001`. Notice that the two's complement of `0000` is `1111+1 = 0000`. Two's complement representation has only one representation of 0, namely, `0000` (unlike signed-magnitude representation, which had two representations of 0). Also notice that we can represent –8 as `1000`. So two's complement is slightly asymmetric, representing one more negative number than positive numbers. A 4-bit two's-complement number can represent any number from –8 to +7.

Say you have 4-bit numbers and want to store –5. –5 would be `(0101)'+1 = 1010+1 = 1011`. Now you want to add –5 to 4 (or `0100`). So you simply add `1011 + 0100 = 1111`, which is the correct answer of –1.

The highest-order bit in two's complement acts as a sign bit: 0 means positive, 1 means negative.

Note that negative numbers all have a `1` in the highest-order (leftmost) bit; thus, the highest-order bit in two's complement is often referred to as the ***sign bit***, `0` indicating a positive number, `1` a negative number. An N-bit binary number that only represents positive numbers is called an ***unsigned number*** and can represent numbers from 0 to 2^N-1. For example, an 8-bit unsigned number can represent numbers from 0 to 255. An N-bit binary number that can represent positive or negative numbers is called a ***signed number*** (more specifically, a signed two's-complement number, which is the most common form and the only form this book uses). A signed number can represent numbers from -2^{N-1} to $+2^{N-1}$-1. For example, an 8-bit signed number can represent numbers -128 to +127. You

can't tell whether a number like 1011 is a signed or unsigned number (or even a number at all) just by looking at it; somebody has to tell you what the bits represent.

If you want to know the magnitude of a two's complement negative number, you can obtain the magnitude by taking the two's complement again. So to determine what number 1111 represents, we can take the two's complement of 1111: (1111)'+1 = 0000+1 = 0001. We put a negative sign in front to yield –0001, or –1.

This is a helpful and commonly-used method when learning two's complement; remember it.

A quick method for humans to mentally figure out the magnitude of a negative number in 4-bit two's complement (having a 1 in the high order bit) is to subtract the magnitude of the three lower bits from 8. So for 1111, the low three bits are 111 or 7, so the magnitude is 8–7=1, which in turn means that 1111 represents –1. For an 8-bit two's complement number, we would subtract the magnitude of the lower 7 bits from 128. So 10000111 would be –(128–7) = –121.

To summarize, we can represent negative numbers using two's complement representation. Addition of two's complement numbers proceeds unmodified—we just add the numbers. Even if one or both numbers are negative, we simply add the numbers. We perform subtraction of A – B by taking the two's complement of B and then adding that two's complement to A, resulting in A + (–B). We compute the two's complement of B by simply inverting the bits of B and then adding 1. B could have originally been positive or negative; the two's complement correctly negates B in either case.

Building a Subtractor Using an Adder and Two's Complement

Knowledge of the two's complement representation leads to a technique to subtract using an adder. To compute A – B, we compute A + (–B), which is the same as A + B' + 1 because –B can be computed as B' + 1 in two's complement. Thus, to perform subtraction we invert B and input a 1 to the carry-in of an adder, as shown in Figure 4.53.

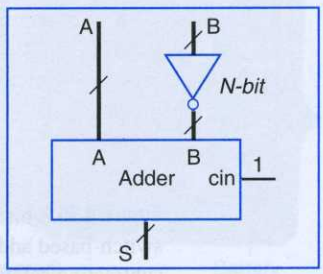


Figure 4.53 Two's complement subtractor built with an adder.

Adder/Subtractor

An adder/subtractor component can be straightforwardly designed, having a control input sub. When sub=1 the component subtracts, but when sub=0 the component adds. The design is shown in Figure 4.54(a). The *N*-bit 2x1 multiplexor passes B when sub=0, and passes B' when sub=1. sub is connected to cin also, so that cin is 1 when subtracting. Actually, XORs can be used instead of the inverters and mux, as shown in Figure 4.54(b). When sub=0, the

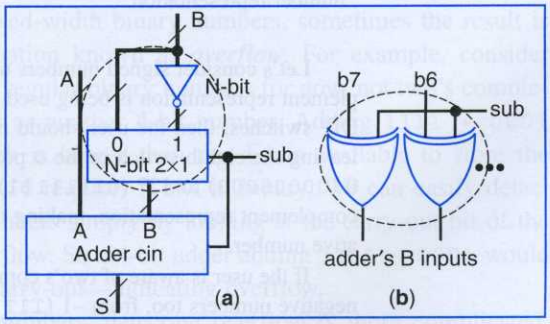


Figure 4.54 Two's complement adder/subtractor using a mux, (b) alternative circuit for B using XOR gates.

output of XOR equals the other input's value. When `sub=1`, the output of the XOR is the inverse of the other input's value.

Example 4.11 DIP-switch-based adding/subtracting calculator (continued)

Let's revisit our DIP-switch-based adding/subtracting calculator of Example 4.9. Observe that at any given time, the output displays the results of either the adder or subtractor, but never both simultaneously. Thus, we really don't need both an adder and a subtractor operating in parallel; instead, we can use a single adder/subtractor component. Assuming DIP switches have been set, setting `f=0` (add) versus `f=1` (subtract) should result in the following computations:

```
00001111 + 00000001 (f=0) = 00010000
00001111 - 00000001 (f=1) = 00001111 + 11111110 + 1 =
  00001110
```

We achieve this simply by connecting `f` to the `sub` input of the adder/subtractor, as shown in Figure 4.55.

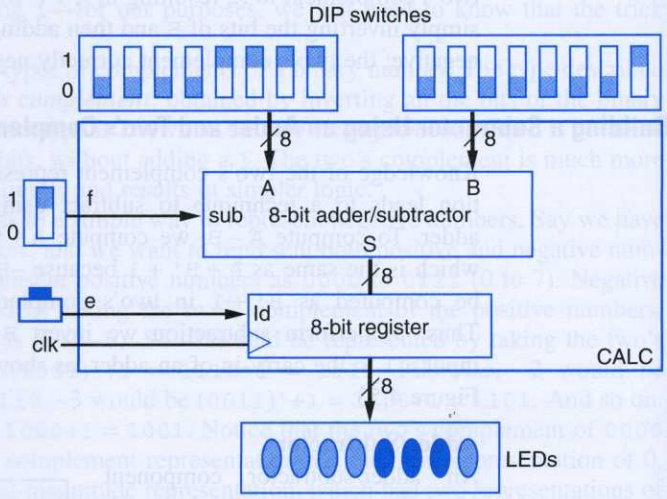


Figure 4.55 8-bit DIP-switch-based adding/subtracting calculator, using an adder/subtractor and two's comlement number representation.

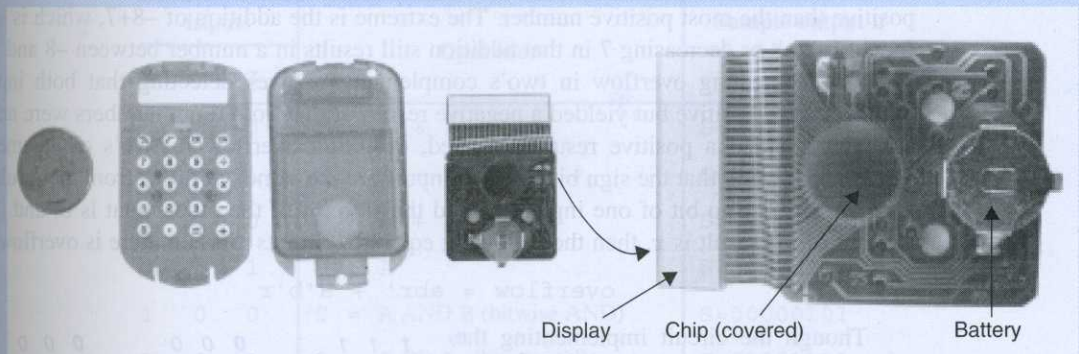
Let's consider signed numbers using two's complement. If the user is unaware that two's complement representation is being used and the user will only be inputting positive numbers using the DIP switches, then the user should only use the low-order 7 switches of the 8-switch DIP inputs, leaving the eighth switch in the `0` position, meaning the user can only input numbers ranging from 0 (`00000000`) to 127 (`01111111`). The reason the user can't use the eighth bit is that in two's complement representation, making the highest-order bit a `1` causes the number to represent a negative number.

If the user is aware of two's complement, then the user could use the DIP switches to represent negative numbers too, from –1 (`11111111`) down to –128 (`10000000`). Of course, the user will need to check the leftmost LED to determine whether the output represents a positive number or a negative number in two's complement form.

▶ WHY SUCH CHEAP CALCULATORS? ECONOMY OF SCALE

Several earlier examples dealt with designing simple calculators. Cheap calculators, costing less than a dollar, are easy to find. Calculators are even given away for free by many companies selling something else. But a calculator internally contains a chip implementing a digital circuit, and chips normally aren't cheap. Why are some calculators such a bargain?

The reason is known as ***economy of scale***, which means that products are often cheaper when produced in large volumes. Why? Because the design and setup costs can be amortized over larger numbers. Suppose it costs $1,000,000 to design a custom calculator chip and to setup the chip's manufacturing (not so unreasonable a number)—design and setup costs are often called ***nonrecurring engineering***, or ***NRE***, costs. If you plan to produce and sell one such chip, then you need to add $1,000,000 to the selling price of that chip if you want to break even (meaning to recover your design and setup costs) when you sell the chip. If you plan to produce and sell 10 such chips, then you need to add $1,000,000/10 = $100,000 to the selling price of each chip. If you plan to produce and sell 1,000,000 such chips, then you need to add only $1,000,000/1,000,000 = $1 to the selling price of each chip. And if you plan to produce and sell 10,000,000,[1] you need to add a mere $1,000,000/10,000,000 = $0.10 = 10 cents to the selling price of each chip. If the actual raw materials only cost 20 cents per chip, and you add another 10 cents per chip for profit, then I can buy the chip from you for a mere 40 cents. And I can then give away such a calculator for free, as many companies do, as an incentive for people to buy something else.



Detecting Overflow

When performing arithmetic using fixed-width binary numbers, sometimes the result is wider than the fixed bitwidth, a situation known as ***overflow***. For example, consider adding two 4-bit binary numbers (just regular binary numbers for now, not two's complement numbers) and storing the result as another 4-bit number. Adding `1111 + 0001` yields `10000`—a 5-bit number, which is bigger than the 4 bits available to store the result. In other words, 15+1=16, and 16 requires 5 bits in binary. We can easily detect overflow when adding two binary numbers simply by looking at the carry-out bit of the adder—a carry-out of `1` indicates overflow. So a 4-bit adder adding `1111 + 0001` would output `1 + 0000`, where the `1` is the carry-out—indicating overflow.

When using two's complement numbers, detecting overflow is more complicated. Suppose we have 4-bit numbers in two's complement form. Consider the addition of two positive numbers, such as `0111` and `0001` in Figure 4.56(a). A 4-bit adder would output `1000`, but that is incorrect—the result of 7+1 should be 8, but `1000` represents –8 in two's complement.

The problem is that the largest positive number that a 4-bit two's complement number can represent is 7. So when adding two positive numbers, overflow can be detected by checking whether the result's most significant bit is `1`.

Likewise, consider adding two negative numbers, such as `1111` and `1000` in Figure 4.56(b). An adder would output a sum of `0111` (and a carry-out of `1`), which is incorrect: –1 + –8 should be –9, but `0111` is +7. The problem is that the most negative number that a 4-bit two's complement can represent is –8. Thus, when adding two negative numbers, overflow can be detected by checking whether the most significant bit is a `0` in the result.

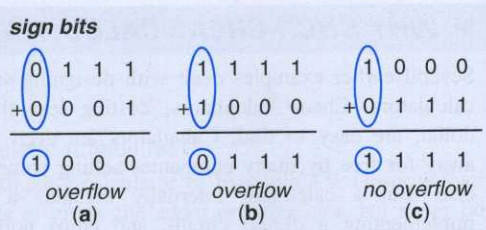


Figure 4.56 Two's complement overflow detection comparing sign bits: (a) when adding two positive numbers, (b) when adding two negative numbers, (c) no overflow.

Adding a positive with a negative, or a negative with a positive, can never result in overflow. The result will always be less negative than the most negative number or less positive than the most positive number. The extreme is the addition of –8+7, which is –1. Increasing –8 or decreasing 7 in that addition still results in a number between –8 and 7.

Thus, detecting overflow in two's complement involves detecting that both input numbers were positive but yielded a negative result, or that both input numbers were negative but yielded a positive result. Restated, detecting overflow in two's complement involves detecting that the sign bits of both inputs are the same but differ from the result's sign bit. If the sign bit of one input is `a` and the sign bit of the other input is `b`, and the sign bit of the result is `r`, then the following equation outputs `1` when there is overflow:

```
overflow = abr' + a'b'r
```

Though the circuit implementing the overflow detection equation is simple and intuitive, we can create an even simpler circuit if the adder generates a carry-out. The simpler method merely compares the carry into the sign bit column with the carry-out of the sign bit column—if the carry-in and carry-out differ, overflow has occurred. Figure 4.57 illustrates this method for several cases. In Figure 4.57(a), the carry into the sign bit is `1`, whereas the carry-out is `0`. Because the carry-in and carry-out differ, overflow has occurred. A circuit detecting whether two bits differ is just an XOR gate, which is slightly simpler than the circuit of the previous method. We omit discussion as to why this method works, but looking at the cases in Figure 4.57 should help provide the intuition.

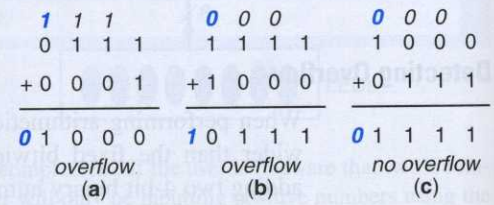


Figure 4.57 Two's complement overflow detection comparing carry into and out of the sign bit column: (a) when adding two positive numbers, (b) when adding two negative numbers, (c) no overflow.

▶ 4.7 ARITHMETIC-LOGIC UNITS—ALUS

An ***N-bit arithmetic-logic unit (ALU)*** is a combinational datapath component able to perform a variety of arithmetic and logic operations on two *N*-bit wide data inputs, generating an *N*-bit data output. Example arithmetic operations include addition and subtraction. Example logic operations include AND, OR, XOR, etc. Control inputs to the ALU indicate which particular operation to perform. To understand the need for an ALU component, consider the following example.

Example 4.12 Multi-function calculator without using an ALU

Let's extend the earlier DIP-switch-based calculator to support eight operations, determined by a three-switch DIP switch that provides three inputs `x`, `y`, and `z` to the system, as shown in Figure 4.58. For each combination of the three switches, we want to perform the operations shown in Table 4.2 on the 8-bit data inputs `A` and `B`, generating the 8-bit output on `S`.

TABLE 4.2 Desired calculator operations

Inputs			Operation	Sample output if A=00001111, B=00000101
x	y	z		
0	0	0	S = A + B	S=00010100
0	0	1	S = A - B	S=00001010
0	1	0	S = A + 1	S=00010000
0	1	1	S = A	S=00001111
1	0	0	S = A AND B (bitwise AND)	S=00000101
1	0	1	S = A OR B (bitwise OR)	S=00001111
1	1	0	S = A XOR B (bitwise XOR)	S=00001010
1	1	1	S = NOT A (bitwise complement)	S=11110000

The table includes several bitwise operations (AND, OR, XOR, and complement). A ***bitwise operation*** applies to each corresponding pair of bits of `A` and `B` separately.

We can design a datapath circuit for the calculator as shown in Figure 4.58, using a separate datapath component to compute each operation: an adder computes the addition, a subtractor computes the subtraction, an incrementer computes the increment, and so on. However, that datapath is very inefficient with respect to the number of wires, power consumption, or delay. There are too many wires that must be routed to all those components, and especially to the mux, which will have 8*8 = 64 inputs. Furthermore, every operation is computed all the time, and that wastes power. Such a design is akin to a restaurant cooking every meal that a customer might order and then serving the customer just the one meal that the customer actually orders.

Furthermore, imagine that the calculator deals not with 8-bit numbers, but instead with 32-bit numbers, and supports not just 8 operations but 32 operations. Then the design would have even

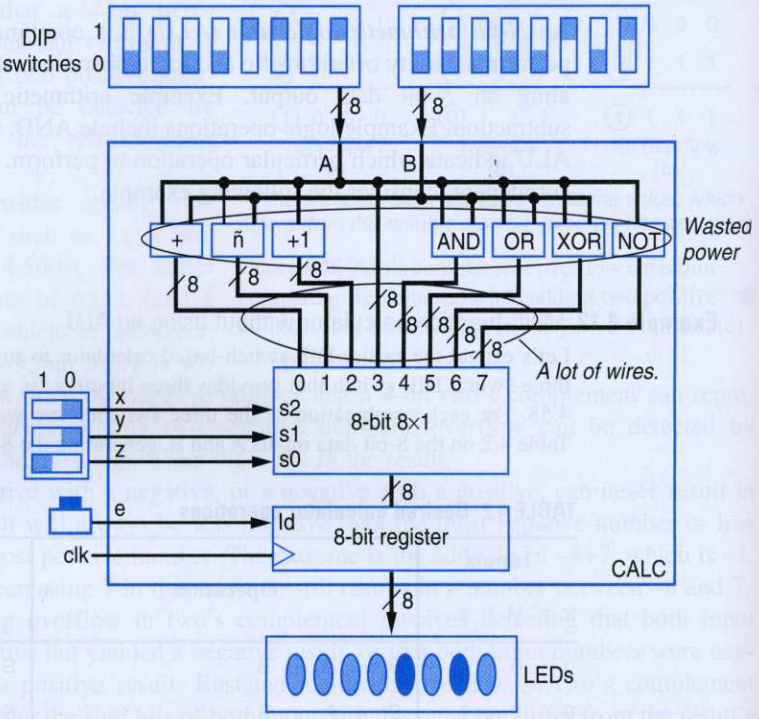


Figure 4.58 8-bit DIP-switch-based multifunction calculator, using separate components for each function.

more wires (32*32 = 1024 wires at the mux inputs), and even more power consumption. Furthermore, a 32x1 mux will require several levels of gates, because due to practical reasons, a 32-input logic gate (inside the mux) will likely need to be implemented using several levels of smaller logic gates.

We saw in the above example that using separate components for each operation is not efficient. To solve the problem, note that the calculator can only be configured to do one operation at a time, so there is no need to compute all the operations in parallel, as was done in the example. Instead, we can create a single component (an ALU) that can compute any of the eight operations. Such a component would be more area- and power-efficient, and would have less delay because a large mux would not be needed.

Let's start with an adder as the base internal ALU design. To avoid confusion, the inputs to the internal adder are named `IA` and `IB`, short for "internal `A`" and "internal `B`," to distinguish those inputs from the external ALU inputs `A` and `B`. We start with the design shown in Figure 4.59(a). The ALU consists of an adder and some logic in front of the adder's inputs, called an arithmetic/logic extender, or *AL-extender*. The purpose of the

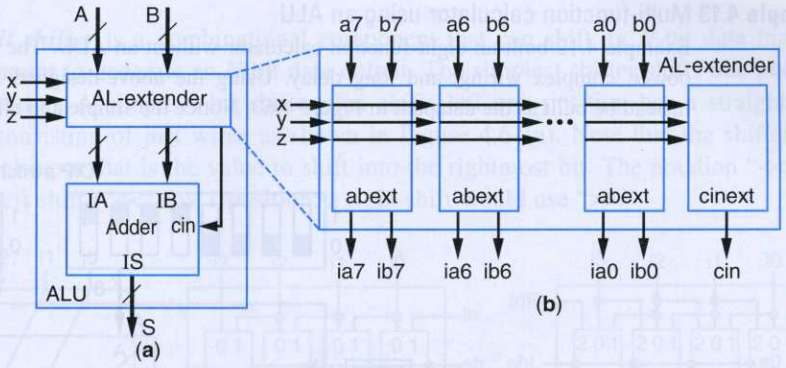


Figure 4.59 Arithmetic-logic unit: (a) ALU design based on a single adder, with an arithmetic/logic extender, (b) arithmetic/logic extender detail.

AL-extender is to set the adder's inputs based on the values of the ALU's control inputs `x`, `y`, and `z`, such that the desired arithmetic or logic result appears at the adder's output. The *AL-extender* actually consists of eight identical components labeled *abext*, one for each pair of bits `ai` and `bi`, as shown in Figure 4.59(b). It also has a component *cinext* to compute the `cin` bit.

Thus, we need to design the *abext* and *cinext* components to complete the ALU design. Consider the first four calculator operations from Table 4.2, which are all arithmetic operations:

- When `xyz=000`, `S=A+B`. So in that case, we want `IA=A`, `IB=B`, and `cin=0`.
- When `xyz=001`, `S=A-B`. So we want `IA=A`, `IB=B'`, and `cin=1`.
- When `xyz=010`, `S=A+1`. So we want `IA=A`, `IB=0`, and `cin=1`.
- When `xyz=011`, `S=A`. So we want `IA=A`, `IB=0`, and `cin=0`. Notice that `A` will pass through the adder, because `A+0+0=A`.

The last four ALU operations are all logical operations. We can compute the desired operation in the *abext* component, and input the result to `IA`. We then set `IB` to `0` and cin to `0`, so that the value on `IA` passes through the adder unchanged.

One possible design of *abext* places an 8x1 mux in front of each output of the *abext* and *cinext* components, with `x`, `y`, and `z` as the select inputs, in which case we would set each mux data input as described above. A more efficient and faster design would create a custom circuit for each component output. We leave the completion of the internal design of the *abext* and *cinext* components as an exercise for the reader.

Example 4.13 redesigns the multifunction calculator of Example 4.12, this time utilizing an ALU.

Example 4.13 Multi-function calculator using an ALU

Example 4.12 built an eight-function calculator without an ALU. The result was wasted area and power, complex wiring, and long delay. Using the above-designed ALU, the calculator could instead be built as the datapath in Figure 4.60. Notice the simple and efficient design.

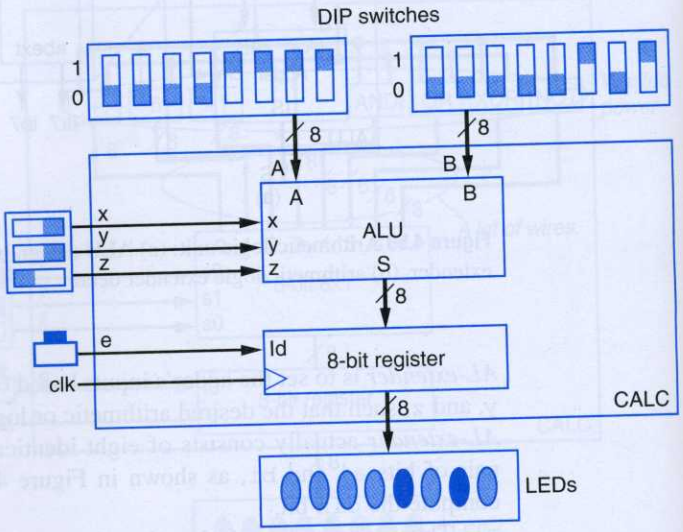


Figure 4.60 8-bit DIP-switch-based multi-function calculator using an ALU.

▶ 4.8 SHIFTERS

Shifting is a common operation applied to data. Shifting can be used to rearrange bits of data. Shifting is useful for communicating data serially as was done in Example 4.4. Shifting is also useful for multiplying or dividing an *unsigned* binary number by a factor of 2. In base ten, you are familiar with the idea that multiplying by 10 can be done by simply appending a 0 to a number. For example, 53 times 10 is 530. Appending a 0 is the same as shifting left one position (and shifting in a 0). Likewise, in base two, multiplying an unsigned binary number by 2 can be done by appending a 0, meaning shifting left one position. So `0101` times 2 is `1010`. Furthermore, in base ten, multiplying by 100 can be done by appending two 0s, meaning shifting left twice. So in base two, multiplying by 4 can be done by shifting left twice. Shifting left three times in base two multiplies by 8. And so on. Because shifting an unsigned binary number left is the same as multiplying by 2, shifting an unsigned binary number right is the same as dividing by 2. So `1010` divided by 2 is `0101`.

Although shifting can be done using a shift register, sometimes we find the need to use a separate combinational component that performs the shift and that can also shift by different numbers of positions and in either direction.

Simple Shifters

An ***N-bit shifter*** is a combinational component that can shift an *N*-bit data input by a fixed amount to generate an *N*-bit data output. The simplest shifter shifts one position in one direction. A simple 4-bit shifter that shifts left one position has a straightforward design consisting of just wires as shown in Figure 4.61(a). Note that the shifter has an additional input that is the value to shift into the rightmost bit. The notation "<<1" indicates a left shift ("<<") by 1 position; a right shift would use ">>."

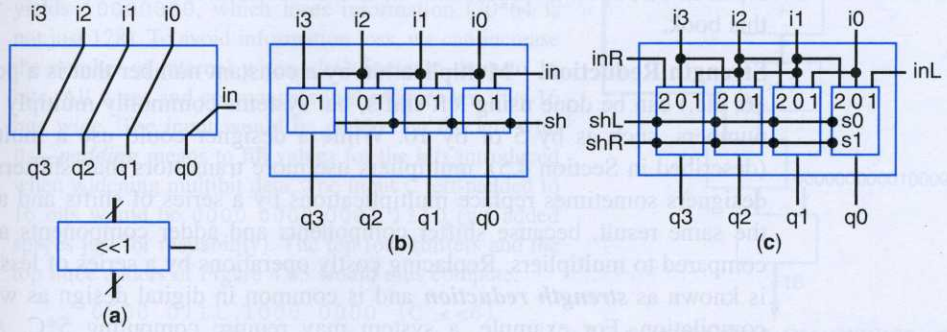


Figure 4.61 Combinational shifters: (a) left shifter design, with block symbol shown at bottom, (b) left shift or pass component, (c) left/right shift or pass component.

A more capable simple shifter can either shift one position when an additional input `sh` is `1` or can pass the inputs through to the outputs unshifted when `sh` is `0`. The design of such a shifter uses 2x1 muxes as shown in Figure 4.61(b). An even more capable simple shifter shown in Figure 4.61(c) can shift left or right one position. When both shift control inputs are `0`, the inputs pass through unchanged. `shL=1` causes a left shift, and `shR=1` causes a right shift. When both those control inputs are `1`, the shifter could be designed to pass the inputs through unchanged (not shown in the figure). A simple shifter's shift amount could be something other than 1; for example, a shifter component might shift right by two places, denoted as ">>2."

Example 4.14 Temperature averager

Consider a system that reads the unsigned binary output `T` of a sensor such as a speed or temperature sensor in an aircraft. The system may sample the sensor several times a second and average recent samples (called a *sliding* average) to help compensate for sensor imperfections that cause occasional spurious sensor readings; e.g., a speed sensor's last four readings may be 202, 203, 235, 202—that 235 reading is probably incorrect.

Let's design a system that computes the average of four registers `Ra`, `Rb`, `Rc`, and `Rd`, storing the result in a register `Ravg` and outputting the result on output `Tavg`. The average is computed as `(Ra+Rb+Rc+Rd) / 4`. Dividing by 4 is the same as shifting right by two. Thus, we can design the system as a

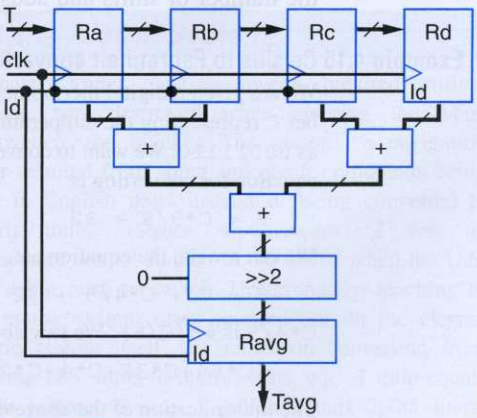


Figure 4.62 Temperature averager using a right-shift-by-2 simple shifter component to divide by 4.

datapath having three adders, a simple right shifter that shifts by two places (with a shift in value of 0), and registers, as shown in Figure 4.62.

This section stated that shifting was the same as multiplication or division by factors of 2 for *unsigned* numbers. Simple shifting would not accomplish such multiplication or division for *signed* numbers. For example, consider the signed 4-bit number 1010, meaning –6. Right shifting to try to divide by 2 would result in 0101 or +5, which is not the correct result of –1, or 1111. Likewise, left shifting to try to multiply by 2 would result in 0100, or 4, rather than –4, or 1100. Solutions exist but are beyond the scope of this book.

Strength Reduction. Multiplication by a constant number that is a power of 2 (2, 4, 8, 16, etc.) can be done using left shifts, but systems commonly multiply by other constant numbers, such as by 5 or by 10. While a designer could use a multiplier component (described in Section 4.5), multipliers use more transistors than shifters or adders. Thus, designers sometimes replace multiplications by a series of shifts and adds that compute the same result, because shifter components and adder components are small and fast compared to multipliers. Replacing costly operations by a series of less costly operations is known as ***strength reduction*** and is common in digital design as well as in software compilation. For example, a system may require computing 5*C. A designer might rewrite that computation as 4*C + C, thus requiring a left-shift-by-2 component (for 4*C) and an adder, which together are still smaller and faster than a multiplier.

Division is slightly harder to do precisely using shifts and adds but can still be done using a similar strength reduction approach. For example, C/5 is C*(1/5), and 1/5 = 0.20. 1/5 could be approximated by converting to a fraction that is close to 0.20 and that has a denominator that is a power of 2, such as 102/512 (0.199). Then the numerator could be computed as the sum of powers of two, i.e., 102/512 is (64+32+4+2)/512. Therefore, C/5 nearly equals C*(64+32+4+2)/512, or (C*64 + C*32 + C*4+C*2)/512. All the multiplications and divisions in that equation can be achieved using left shifts or right shifts, i.e., ((C<<6) + (C<<5) + (C<<2) + (C<<1)) >> 9. For even more accuracy, a larger denominator (and hence numerator) could be used to get closer to 0.20. At some point, though, the number of shifts and adds may exceed the cost of a divider component.

Example 4.15 Celsius to Fahrenheit converter

We are given a digital thermometer that digitizes a temperature into an 8-bit unsigned binary number C representing the temperature in Celsius. For example, 30 degrees Celsius would be digitized as 00011110. We want to convert that temperature to Fahrenheit, again using 8 unsigned bits. The equation for converting is:

```
F = C*9/5 + 32
```

We can rewrite the equation as:

```
F = C + C*4/5 + 32
```

C*4/5 is 4*(C/5). We saw above that C/5 can be closely approximated as:

```
(C*64+C*32+C*4+C*2)/512
```

The multiplication of the above equation by 4 to obtain 4*(C/5) changes the denominator to 128. Thus, the equation for converting can be rewritten as:

```
F = C+(C*64+C*32+C*4+C*2)/128+32
```

The equation for F has been rewritten such that any multiplications or divisions are by a constant power of 2 and are thus replaceable by shifts.

The datapath circuit implementing the rewritten equation is shown in Figure 4.63. Consider an input C=00011110 representing 30 degrees Celsius, which is 86 degrees according to the first conversion equation above. The circuit's "<<6" shifter shifts the input 6 places to the left; shifting 00011110 6 places left yields 10000000, which loses information (30*64 is not just 128). To avoid information loss, we can increase the number of internal wires—let's increase them to 16 bits. All wires and components in the figure are thus 16 bits wide. The input would be padded on the left with 0s—***padding*** means to fill values for the bits introduced when widening multibit data. The input C left-padded to 16 bits would be 0000 0000 0001 1110 (we added spaces just for readability). The top four shifters and the top three adders of Figure 4.63 would thus compute:

```
  0000 0111 1000 0000  (C <<6)
+ 0000 0011 1100 0000  (C <<5)
+ 0000 0000 0111 1000  (C <<2)
+ 0000 0000 0011 1100  (C <<1)
= 0000 1011 1111 0100
```

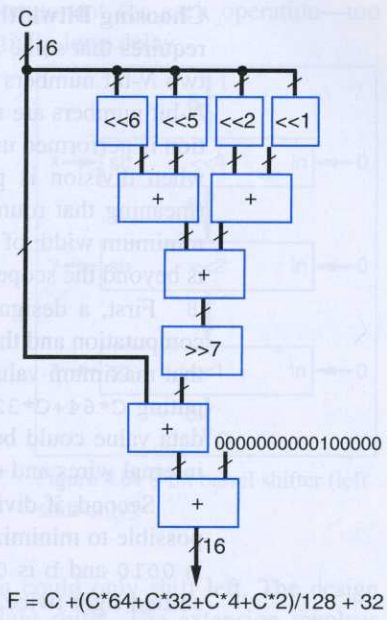


Figure 4.63 Celsius to Fahrenheit converter datapath using shifts and adds.

The shift right by 7 would output 0000 0000 0001 0111. The adder that adds this result to the input C of 0000 0000 0001 1110 would output 0000 0000 0011 0101. The bottom-most adder adds this result to the constant 32 (0000 0000 0010 0000) to yield the final output of 0000 0000 0101 0101. The leftmost 8 bits can be dropped to yield the 8-bit output for F of 01010101, which is 85 in binary—slightly off from the correct value of 86 due to the shift and add approximation approach, but close. A larger denominator (and hence numerator) in the approximation approach would yield higher accuracy at the expense of more shifts and adds.

▶ Fahrenheit versus Celsius—The U.S. and the Metric System

The U.S. usually represents temperature using Fahrenheit, whereas most of the world uses the metric system's Celsius. Presidents and other U.S. leaders have desired to switch to the metric system for almost as long as the U.S. has existed, and several acts have been passed over the centuries, the most recent being the Metric Conversion Act of 1975 (amended several times since). The Act designates the metric system as the preferred system of weights and measures for U.S. trade and commerce. Yet switching to metric has been slow, and few Americans today are comfortable with metric. The problem with such a slow transition was poignantly demonstrated in 1999 when the Mars Climate Orbiter, costing several hundred million dollars, was destroyed when entering the Mars atmosphere too quickly. The reason: "a navigation error resulted from some spacecraft commands being sent in English units instead of being converted to metric units." (Source: www.nasa.gov). I was an elementary school student in the 1970s when the U.S. had a big push to switch. Unfortunately, teaching of the metric system often focused not on the elegant metric system itself, but rather on converting from existing U.S. units to metric units, e.g., 1 mile equals 1.609 kilometers, and 1 gallon equals 3.785 liters. Many Americans thus found the metric system "difficult." No wonder.

Choosing Bitwidths. As alluded to in previous examples, operating on *N*-bit numbers requires that some attention be paid to the bitwidths of internal wires and components. If two *N*-bit numbers are added together, the resulting value could require *N+1* bits. If two *N*-bit numbers are multiplied, the resulting value could require *2N* bits. When multiplication is performed using left shifts, bits may be dropped off of the left (meaning overflow); when division is performed using right shifts, bits may be dropped off of the right (meaning that rounding is occurring). For expected input value ranges, determining the minimum width of all internal wires and components requires mathematical analysis that is beyond the scope of this book. Instead, we describe a few introductory guidelines here.

First, a designer can determine the maximum data value that would occur during computation and then make all the internal wires and components wide enough to support that maximum value. For example, the system described in Example 4.15 involved computing `C*64+C*32+C*4+C*2` for a `C` possibly as large as 255, meaning the maximum data value could be 18,137, which would require 15 bits. In the example, we set all the internal wires and components to 16 bits.

Second, if division will be done, it may be a good idea to do the division as late as possible to minimize the rounding that might occur with each division. For example, if `a` is `0010` and `b` is `0010` and if right shifts are used for division, then `(a+b)/4` is `0100` shifted right twice, or `0001`; instead, `a/4 + b/4` would yield `0010` shifted right twice plus `0010` shifted right twice, or `0000 + 0000 = 0000`; the rounding errors of the earlier divisions caused a poor result. Example 4.15 multiplied the input by four different amounts, added the products, and then divided, i.e., `(C*64+C*32+C*4+C*2)/128`. The division could have instead been applied to each product first followed by addition, i.e., `C*64/128+C*32/128+C*4/128+C*2/128)`, which in fact would reduce to `C/2+C/4+C/32+C/64`. While this equation looks simpler and would even use one less shifter, the result would have more rounding error. Of course, doing the division as late as possible must be balanced with the need to prevent the intermediate sums from getting very large, which could result in very wide components and sets of wires. For example, adding 800 numbers and then dividing by 2 might be better accomplished by adding perhaps 8 numbers at a time and dividing each such sum by 2, then finally adding all the results.

▶ 4.9 (

Barrel Shifter

An ***N-bit barrel shifter*** is a general purpose *N*-bit shifter component that can shift any number of positions. For simplicity, consider only left shifts for the moment. An 8-bit barrel shifter can shift left by 1, 2, 3, 4, 5, 6, or 7 positions (and of course by 0 positions, meaning no shift is done). An 8-bit barrel shifter therefore requires 3 control inputs, say `x`, `y`, and `z`, to specify the distance of the shift. `xyz=000` may mean no shift, `xyz=001` means shift by 1 position, `xyz=010` means shift by 2 positions, etc. A barrel shifter could be useful to replace several shift components (such as the five shifters in Figure 4.63) by a single component to save transistors or wires (later chapters will show how to do that), or when the shift-amount is not known while the circuit is being designed.

We could design such a barrel shifter using 8 distinct shifters: a 1-place left shifter, a 2-place left shifter, and so on. The 8 shifters' outputs would be connected to an 8-bit 8x1 mux having `xyz` connected to its select lines, and the mux output would be the barrel shifter's output. While conceptually straightforward, such a design has problems similar

to building a multifunction ALU using a distinct component for each operation—too many wires, excessive power consumption, and potentially long delay.

A more elegant design for an 8-bit barrel shifter consists of 3 cascaded simple shifters, as shown in Figure 4.64. The first simple shifter can shift left four positions (or none), the second can shift left by two positions (or none), and the third by one position (or none). Notice that the shifts "add" to one another—shifting left by 2, then by 1, results in a total shift of 3 positions. Thus, configuring each shifter appropriately yields a total shift of any amount between zero and seven. Connecting the control inputs `xyz` to the shifters is easy—just think of `xyz` as a binary number representing the amount of the shift: `x` represents shifting by four, `y` shifting by two, and `z` shifting by one. So we just connect `x` to the left-by-four shifter, `y` to the left-by-two shifter, and `z` to the left-by-one shifter, as shown in Figure 4.64.

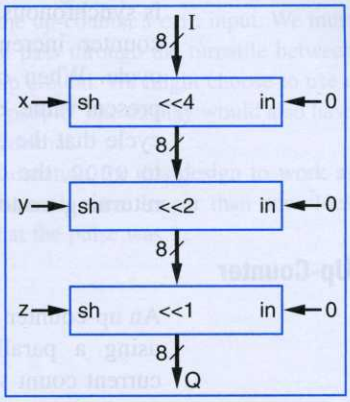


Figure 4.64 8-bit barrel shifter (left shift only).

The above design considered a barrel shifter that could only shift left. The design could easily be extended to support both left and right shifts. The extension involves replacing the internal left shifters by shifters that could shift left or right, each shifter having a control input indicating the direction. The barrel shifter would also have a direction control input, connected to each internal shifter's direction control input.

Finally, the barrel shifter can easily be extended to support rotates as well as shifts. The extension would replace the internal shifters by rotators that could either shift or rotate, each having a control input indicating whether to shift or rotate. The barrel shifter would also have a shift-or-rotate control input, connected to each internal shifter's shift-or-rotate control input.

▶ 4.9 COUNTERS AND TIMERS

An ***N-bit counter*** is a sequential component that can increment or decrement its own value on each clock cycle when a count enable control input is `1`. ***Increment*** means to add 1, and ***decrement*** means to subtract 1. A counter that can increment its value is known as an ***up-counter***, while a ***down-counter*** decrements its value. A 4-bit up-counter would thus count the following sequence: `0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111, 0000, 0001`, etc. Notice that a counter ***wraps around*** (also known as *rolling over*) from the highest value (`1111`) to 0. Likewise, a down-counter would wrap around from 0 to the highest value. A control output on the counter, often called ***terminal count*** or `tc`, becomes `1` during the clock cycle that the counter has reached its terminal (meaning "last" or "end") count value, after which the counter will wrap around.

Figure 4.65 shows the block symbol of a 4-bit up-counter. When `clr=1`, the counter's value is synchronously cleared to 0. When `cnt=1`, the counter increments its value on every clock cycle. When `cnt=0`, the counter maintains its present value. `clr` has priority over `cnt`. On the cycle that the counter wraps around from `1111` to `0000`, the counter sets `tc=1` for that cycle, returning `tc` to `0` on the next cycle.

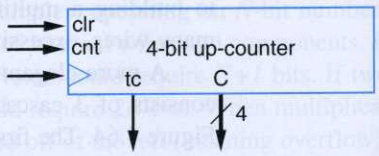


Figure 4.65 4-bit up-counter block symbol.

Up-Counter

An up-counter design is shown in Figure 4.66, using a parallel-load register to store the current count value and using an incrementer component to add `1` to the current value. When `clr=1`, the register will be cleared to 0. When `cnt=0`, the register will maintain its present value due to `ld` being `0`. When `cnt=1`, the register will be loaded with its present value plus `1`. Note that the 4-input AND gate causes terminal count `tc` to become `1` when the counter reaches `1111`.

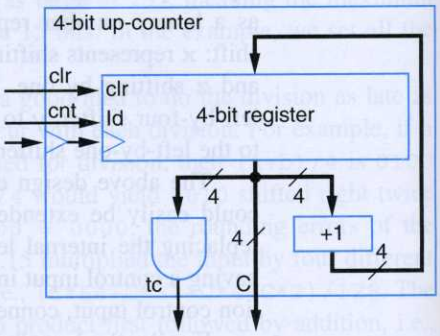


Figure 4.66 4-bit up-counter internal design.

A down-counter can be designed similarly to an up-counter. The incrementer would be replaced by a decrementer. The terminal count `tc` should become `1` when the down-counter reaches `0000` and would thus be implemented using a NOR gate rather than the AND gate in the up-counter—recall that NOR outputs `1` when all its inputs are `0`s. The reason the down-counter detects `0000` for `tc` rather than `1111` like the up-counter is because a down-counter wraps around after `0000`, as in the following count sequence: `0100, 0011, 0010, 0001, 0000, 1111, 1110, ....`

Example 4.16 Turnstile with display

This example designs a system that displays the number of people who have passed through a turnstile. Turnstiles are commonly found at entrances of amusement parks, sports stadiums, and other facilities with controlled entrances. We'll assume the turnstile outputs a pulse on a signal `P` whenever a person passes through. The system should output in binary the number of people that have passed through, and that output is connected to a display that will output that number in decimal. The system should also have a button to reset the display to 0.

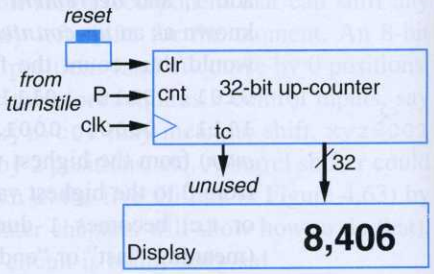


Figure 4.67 Turnstile display using an up-counter.

The system can be straightforwardly designed using an up-counter as shown in Figure 4.67. The reset button connects to the up-counter's clear input. Assuming the pulse on `P` is one clock cycle in duration (see Chapter 3), then `P` can be connected to the up-counter's `cnt` input. We must have some idea of the maximum number of people that might pass through the turnstile between resets of the display, because we don't want the counter to wrap around. We might choose to use a 32-bit counter to be safe, which can count up to about 4 billion people. The display would also have to be able to accept a 32-bit input and display that number in decimal.

Note that a pulse on `P` must be only one clock cycle in duration for this design to work as desired. If a pulse on `P` for one person passing through the turnstile were longer than one clock cycle, the system would count up once for every clock cycle that the pulse was `1`.

Up/Down-Counter

An up/down-counter can count either up or down. It requires a control input `dir` to indicate the count direction, in addition to the count enable control input `cnt`. `dir=0` will mean to count up and `dir=1` to count down. Figure 4.68 shows the design of such a 4-bit up/down-counter. A 4-bit-wide 2x1 mux passes either the decremented or incremented value, with `dir` selecting among the two—`dir=0` (count up) passes the incremented value, and `dir=1` (count down) passes the decremented value. The passed value gets loaded into the 4-bit register if `cnt=1`. `dir` also selects whether to pass the NOR or AND output to the terminal count `tc` external output—`dir=0` (count up) selects the AND, while `dir=1` (count down) selects the NOR.

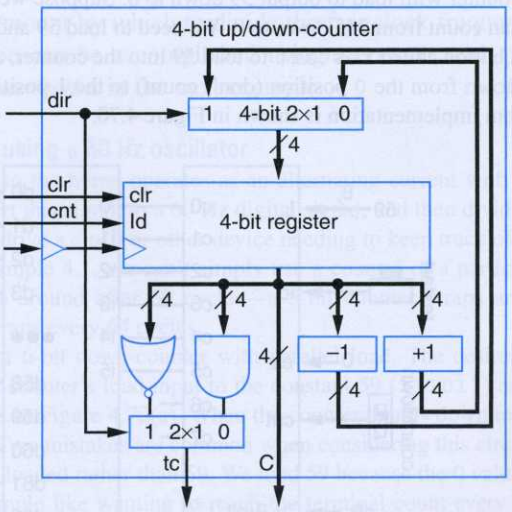


Figure 4.68 4-bit up/down-counter design.

Counter with Load

A well-known counter with load is the "program counter" used in a microprocessor. It holds the address of the current program instruction, normally counting up to go to the next instruction's address, but sometimes being loaded to an entirely new address due to a branch instruction.

Counters often come with the ability to start counting from a particular value, achieved by loading the counter's register with an input value. Figure 4.69 shows the design of a 4-bit up-counter with load. When control input `ld` is `1`, the 2x1 mux passes the input `L` to the register; when `ld` is `0`, the mux passes the incremented value. The design ORs the counter's `ld` and `cnt` signals to generate the load signal for the register. When `cnt` is `1`, the incremented value will be loaded. When `ld` is `1`, the load data will be loaded. Even if `cnt` is `0`, `ld=1` causes the register to be loaded. A synchronous clear input is also provided, which has priority over load and count inputs because the register's clear input has priority over the register's load input. A down-counter or up/down-counter could similarly be extended to have a parallel load.

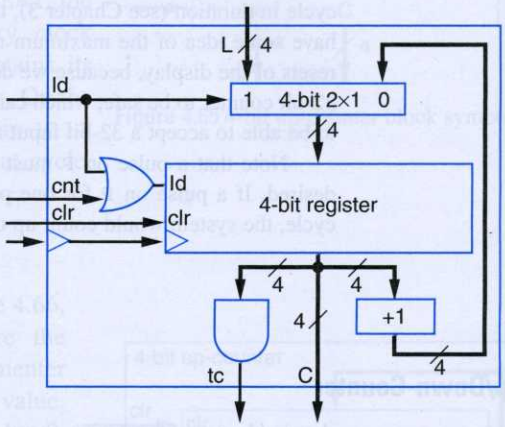


Figure 4.69 Internal design of a 4-bit up-counter with parallel load.

Example 4.17 New Year's Eve countdown display

Example 2.30 utilized a microprocessor to output the numbers 59 down to 0, and a decoder to illuminate one of 60 lights based on that output. This example replaces the microprocessor by a down-counter with load to output 59 down to 0. Suppose we have an 8-bit down-counter available, which can count from 255 down to 0. We need to load 59 and then count down. Assume the user can press a button called `restart` to load 59 into the counter, and then the user can move a switch `count-down` from the `0` position (don't count) to the 1 position (count) to begin the countdown. The system implementation is shown in Figure 4.70.

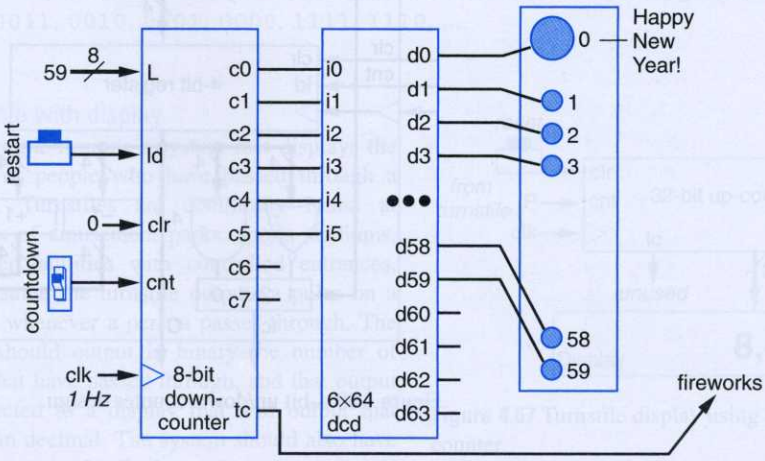


Figure 4.70 Happy New Year countdown system using a down-counter with load.

Some people mistakenly assume that the decoder could be eliminated by using a 60-bit counter instead of an 8-bit counter because there are 60 lights. However, a 60-bit counter would count the following sequence 59 (`000...000111011`), 58 (`000...000111010`), and so on. Those binary outputs connected directly to the lights would illuminate the lights differently than desired; rather than illuminating one light at a time, the approach would light multiple lights at a time.

The `tc` signal Figure 4.70 will be `1` at the same time as the decoder's `d0` output. We've connected `tc` to an output called `fireworks`, which we'll assume activates a device that ignites fireworks. Note that after reaching 0 the counter will wrap around and continue counting from 255, 254, and so on, until the countdown switch is set to the "don't count" position again.

Clock Divider. One use of a counter is to convert one clock frequency into a lower frequency; a component that performs such conversion is known as a ***clock divider***. Suppose we have a 512 MHz input clock (the fast clock), but want a 2 MHz clock signal (the slow clock). The fast clock frequency should thus be divided by 256. We can convert the fast clock into a desired slow clock signal `p` by using an 8-bit counter. The 8-bit counter wraps around every 256 cycles, at which time its terminal count output becomes `1` for one fast-clock cycle, so we simply connect the fast clock oscillator signal to the counter's clock input, set the counter's load input to `1`, and then use the counter's `tc` output as the slow-clock signal, as shown in Figure 4.71.

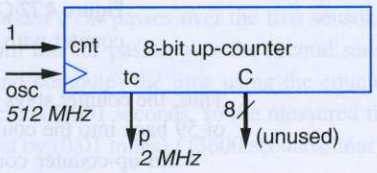


Figure 4.71 Clock divider.

A *clock multiplier* does the opposite of a clock divider, converting an input clock frequency into a higher frequency. Its design is more complex than a clock divider, involving something called a phased-lock loop, and is beyond our scope here.

Sometimes the amount by which to divide the fast clock frequency is not a power of 2. Such clock division can be accomplished using a counter with parallel load, or with some external logic, as shown in the next example.

Example 4.18 1 Hz pulse generator using a 60 Hz oscillator

In the U.S., electricity to the home operates as an alternating current with a frequency of 60 Hz. Some appliances convert this signal to a 60 Hz digital signal, and then divide the 60 Hz digital signal to a 1 Hz signal, to drive a clock or other device needing to keep track of time at the granularity of seconds. Unlike Example 4.1, we can't simply use a counter of a particular bitwidth, since no basic up-counter wraps around after 60 cycles—a 5-bit counter wraps around every 32 cycles, while a 6-bit counter wraps every 64 cycles.

Assume we have a 6-bit down-counter with parallel load. The desired clock divider can be designed by setting the counter's load input to the constant 59 (`111011`) and using the `tc` output to reload the register, as in Figure 4.72(a). When the counter counts down to 0, the circuit automatically reloads with 59. Two mistakes are common when considering this circuit. The first mistake is to believe 60 should be loaded rather than 59. We load 59 because the 0 value is counted too. Think of a smaller count example like wanting to reach the terminal count every 3 cycles; you'd load 2 (not 3) so that the counter would count 2, 1, 0, then 2, 1, 0, etc. The second mistake is to believe that this counter would skip the value 0 because of `tc` being `1` during that clock cycle. Take a moment to understand the timing behavior. When the counter value is 1 and the next rising clock edge arrives, then slightly *after* that clock edge the counter value changes to 0 and `tc` becomes `1`.

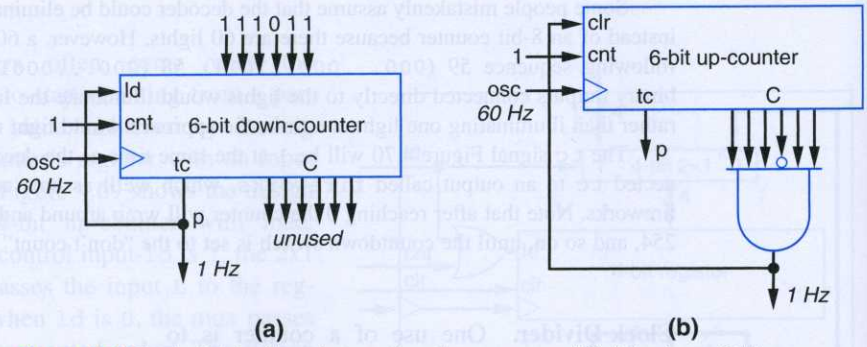


Figure 4.72 Clock divider for a factor other than a power of 2: (a) using a 6-bit up-counter with synchronous clear, (b) using a 6-bit down-counter with parallel load.

Thus, the counter stays at 0 until the *next* rising clock edge, at which time the `1` on `tc` causes a load of 59 back into the counter.

An up-counter could be used instead, again with `tc` connected to load. To obtain 60 cycles between counter wrap arounds, we need to set the counter's load input to 63–59 = 4. People sometimes mistakenly think that 63–60 = 3 should be loaded, but remember that the first value is counted too. Think of a smaller count example like wanting to reach the terminal count every 3 cycles. You'd load 61 (not 60) so that the counter would count 61, 62, 63, then 61, 62, 63, etc.

Alternatively, suppose we only have an up-counter with a synchronous clear input but without parallel load (counters without parallel load exist because they have fewer inputs, which may result in fewer wires or fewer pins). In this case, we could use external logic to generate a signal that will be `1` when the counter's value is 59, and connect that signal to the counter's synchronous clear input so that the value after 59 will be 0, as shown in Figure 4.72(b). Alternatively, a down-counter with synchronous clear could be used, in which case the external logic should detect 63–59=4.

Using a Counter to Measure Time. A designer can use an up-counter to measure the time between events, such as the time between a first press of a button and a second press. The designer can create a circuit that initially clears the counter to 0. When the first event occurs, the circuit should set the counter's control input `cnt` to `1`. When the second event occurs, then the circuit should set `cnt` to `0`. The final value in the counter thus indicates the number of clock cycles that transpired between the first and second event. Multiplying this number by the clock period driving the counter yields the time that transpired. Suppose the counter's clock frequency is 1 kHz, meaning each cycle lasts 1 millisecond. If the final value in the counter is 526, then 526 milliseconds passed between events. Care must be taken to use a counter that is wide enough and/or a clock that is slow enough to ensure that the counter will not wrap around while measuring the time interval of interest.

Example 4.19 Highway speed measuring system

Many highways and freeways have systems that measure the speed of cars at various parts of the highway and upload that speed information to a central computer. Such information is used by law enforcement, traffic planners, and radio and Internet traffic reports.

One technique for measuring the speed of a car uses two sensors embedded under the road, as illustrated in Figure 4.74. When a car is over a sensor, the sensor outputs a `1`; otherwise, the sensor outputs a `0`. A sensor's output travels on underground wires to a speed-measuring computer box, some of which are above the ground and others of which are underground. The speed measurer determines speed by dividing the distance between the sensors (which is fixed and known) by the time taken by a vehicle to travel from the first sensor to the second sensor. If the distance between the sensors is 0.01 miles, and a vehicle takes 0.5 seconds to travel from the first to the second sensor, then the vehicle's speed is 0.01 miles / (0.5 seconds * (1 hour / 3600 seconds)) = 72 miles per hour.

To measure the time between the two sensors separated by 0.01 miles, we can construct a simple FSM that controls a 16-bit up-counter clocked at 1 kHz, as shown in Figure 4.74. State *S0* clears the counter to 0. The FSM transitions to state *S1* when a car passes over the first sensor. *S1* starts the counter counting up. The FSM stays in *S1* until the car passes over the second sensor, causing a transition to state *S2*. *S2* stops the counting and computes the time using the counter's output `C`. The counter's 1 kHz clock means that each cycle is 0.001 seconds, so the measured time would be `C` * 0.001 s. That result would then be multiplied by (0.01 miles) / (3600 seconds/hour) to

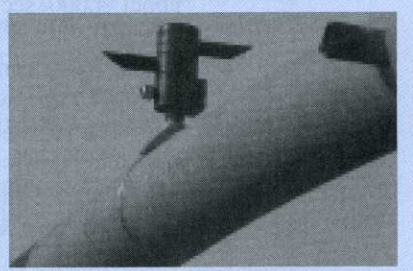
▶ HOW DOES IT WORK? CAR SENSORS IN ROADS.

How does a highway speed sensor or a traffic light car sensor know that a car is present in a particular lane? The main method today uses what's called an *inductive loop*. A loop of wire is placed just under the pavement—you can usually see the cuts, as in Figure 4.73(a). That loop of wire has a particular "inductance," which is an electronics term describing the wire's opposition to a change in electric current—higher inductance means the wire has higher opposition to changes in current. It turns out that placing a big hunk of metal (like a car) near the loop of wire changes the wire's inductance. (Why? Because the metal disrupts the magnetic field created by a changing current in the wire—but that's getting beyond our scope.) The traffic light control circuit keeps checking the wire's inductance (perhaps by trying to change the current and seeing how much the current really changes in a certain time period), and if inductance is more than normal, the circuit assumes a car is above the loop of wire.

Many people think that the loops seen in the pavement are scales that measure weight—I've seen bicyclists jumping up and down on the loops trying to get a light to change. That doesn't work, but it sure is entertaining to watch.

Many others believe that small cylinders attached to a traffic light's support arms, like that in Figure 4.73(b), detect vehicles. Those instead are typically devices that detect a special encoded radio or infrared-light signal from emergency vehicles, causing the traffic light to turn green for the emergency vehicle (e.g., 3M's "Opticom" system). Such systems are another example of digital systems, reducing the time needed by emergency vehicles to reach the scene of an emergency as well as reducing accidents involving the emergency vehicle itself proceeding through a traffic light, thus often saving lives.

(a)



(b)

Figure 4.73 (a) Inductive loop for detecting a vehicle on a road, (b) emergency vehicle signal sensor for changing an intersection's traffic light to green for the approaching emergency vehicle.

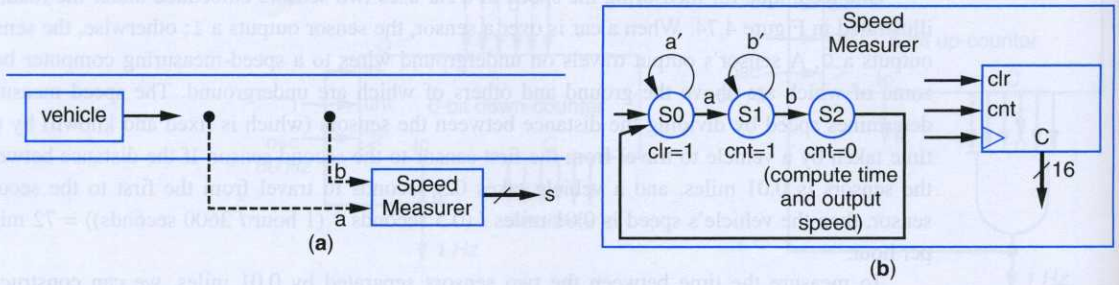


Figure 4.74 Measuring vehicle speeds in a highway speed measuring system: (a) sensors in road feeding into the speed measuring system, (b) state machine controlling an up-counter.

determine a car's speed in miles per hour. We omit the implementation details of the speed computation.

Timers

A ***timer*** is a sequential component that can be programmed to repeatedly generate a short pulse at a user-specified time interval such as every 300 milliseconds. The component is sometimes called a *programmable interval timer.* A timer has a base time unit such as 1 microsecond. A designer programs the timer to a desired time interval by loading a binary number representing the desired multiplication of the base time unit. If a timer's base time unit is 1 microsecond and the designer wants the timer to pulse every 50 microseconds, the designer would load 50 into the timer. If the designer wants a pulse every 300 milliseconds, which means 300,000 microseconds, the designer would load the number 300,000 into the timer. A timer's *width* is the bitwidth of the number than can be loaded; the width defines the maximum time interval of the timer. For example, a 32-bit timer with a base time of 1 microsecond has a maximum interval of $2^{32}*1$ microseconds, or about 4,000 seconds. A timer has an enable control input that activates the timing function.

A timer can be designed using a parallel-load down-counter and a register as in Figure 4.75(a), which shows the design for a 32-bit timer with a 1 microsecond base time unit. The register holds the multiplier number decremented by 1 to compensate for the fact that a down-counter includes 0 in its count; thus, a designer should only load a value greater than 1 (note: most timers actually require the user to subtract 1 from the number before loading; we include the –1 inside the timer to make subsequent designs easier to understand). An input `load` set to `1` causes the register to be loaded with input `M`, and also loads the down-counter with input `M`. The down-counter is clocked by a 1 microsecond oscillator, which could be a standalone oscillator that is internal to the timer, or which could be derived by dividing the clock input of the timer. When `enable` is `1`, the counter counts down once per microsecond until reaching 0, at which time the counter's `tc` output becomes `1`, causing the output `Q` to become `1` for one microsecond. The `tc=1` also causes the counter to be loaded again with the value held in the register.

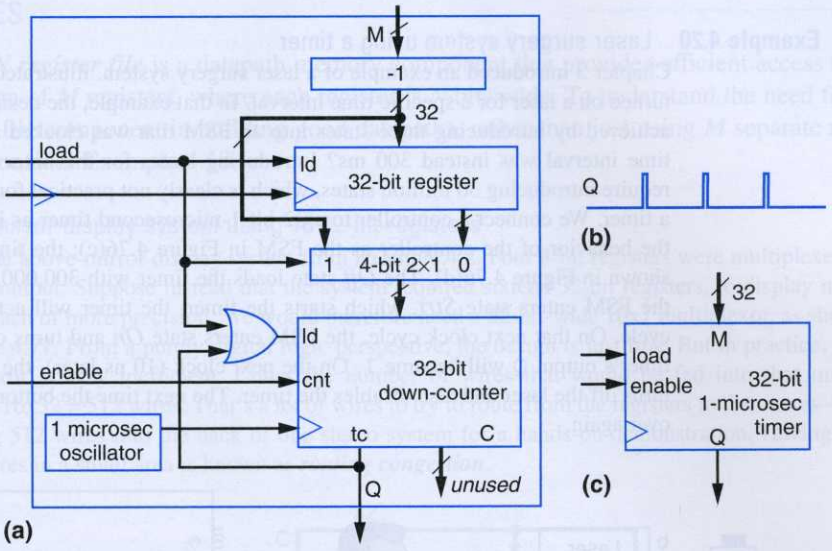


Figure 4.75 Timer: (a) internal design, (b) example timer output when enabled, (c) block symbol.

A timer "ticks" like a clock when enabled as shown in Figure 4.75(b). The period is `M` times the base time unit. The block symbol for a timer is shown in Figure 4.75(c).

A timer is similar to an oscillator. One distinction is that a timer is typically programmable while an oscillator is not, but the distinction is not very rigid and sometimes a timer is called a "programmable timer/oscillator." In terms of usage, designers typically use an oscillator as the clock input of sequential components, whereas designers use a timer to generate events that are detected by an FSM's transition conditions and thus serve as inputs to a controller.

Some variations of timers are commonplace. One variation is a timer with an additional control input `once`; if that input is `1`, the timer stops when it reaches the end of its first interval, thus holding its `Q` output at `1` rather than pulsing `Q` and repeating. Such a timer is referred to as a one-shot timer or simply a ***one-shot***. Another variation is a timer having a second register that can be loaded with the time for which each pulse should be held high. For example, if a timer has a base unit of 1 microsecond, a multiplier register loaded with 500, and a time-high register loaded with 200, then the timer's output would be `1` for 200 microseconds, then `0` for 300 microseconds, then `1` for 200 microseconds again, and so on. This timer variation is known as a ***pulse-width modulator*** or ***PWM***. The percentage of time spent high during each interval is known as the PWM's *duty cycle*; the above example's duty cycle is 200/500 = 40%.

Digital circuits are commonly used in systems that must sample inputs or generate outputs at specified time intervals. For example, an audio recording system (see Chapter 1) may sample an input audio signal 44,000 times per second, a traffic light controller system may keep a green light on for 30 seconds, a laser surgery system may turn on a laser for 500 microseconds, or a video display system may write a new value to a point on the display (a pixel) every 0.1 microseconds. The circuits can use timer components to generate events that indicate when specific time intervals have passed.

Example 4.20 Laser surgery system using a timer

Chapter 3 introduced an example of a laser surgery system, illustrated again in Figure 4.76(a), that turned on a laser for a specific time interval. In that example, the desired time interval of 30 ns was achieved by introducing three states into an FSM that was clocked at 10 ns. What if the desired time interval was instead 300 ms? Introducing states for this interval and a 10 ns clock would require introducing 30 million states, which is clearly not practical for an FSM. Instead, we can use a timer. We connect a controller to a 32-bit 1-microsecond timer as in Figure 4.76(b). We capture the behavior of the controller as the FSM in Figure 4.76(c); the timing behavior of this FSM is shown in Figure 4.76(d). The *Off* state loads the timer with 300,000. When the button is pressed, the FSM enters state *Strt*, which starts the timer; the timer will actually start on the next clock cycle. On that next clock cycle, the FSM enters state *On* and turns on the laser. 300 ms later, the timer's output Q will become 1. On the next clock (10 ns later), the FSM enters state *Off*, which turns off the laser and also disables the timer. The next time the button is pressed, the process starts over again.

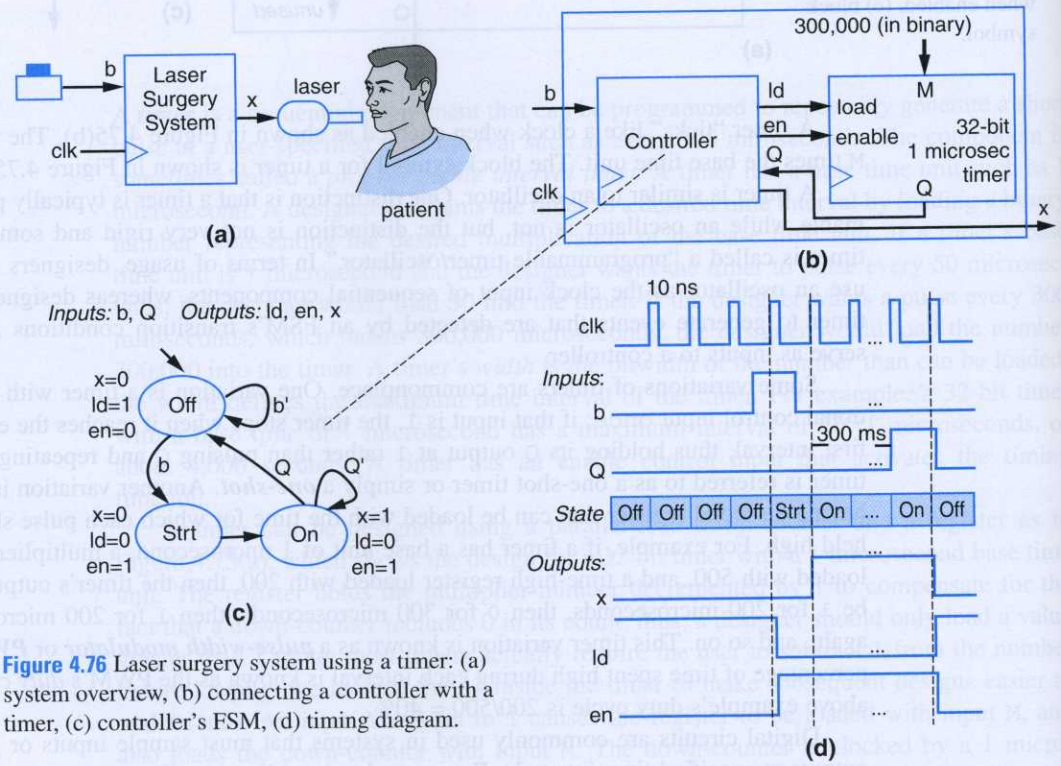


Figure 4.76 Laser surgery system using a timer: (a) system overview, (b) connecting a controller with a timer, (c) controller's FSM, (d) timing diagram.

Note that the system actually keeps the laser on for 300 ms plus 10 ns; this is an effect of using the external timer, and is not likely to be significant for this particular system's desired time interval.

4.10 REGISTER FILES

An ***MxN register file*** is a datapath memory component that provides efficient access to a collection of *M* registers, where each register is *N* bits wide. To understand the need for a register file component in building good datapaths, rather than just using *M* separate registers, consider Example 4.13.

Example 4.21 Above-mirror display system using 16 32-bit registers

Recall the above-mirror display system from Example 4.2. Four 8-bit registers were multiplexed to an 8-bit output. Suppose instead that the system required sixteen 32-bit registers, to display more values, each of more precision. We would therefore need a 32-bit-wide 16x1 multiplexor, as shown in Figure 4.77. From a purely digital logic perspective, the design is just fine. But in practice, that multiplexor is very inefficient. Count the number of wires that would be fed into that multiplexor—16x32 = 512 wires. That's a lot of wires to try to route from the registers to the muxes—try plugging 512 wires into the back of one stereo system for a hands-on demonstration. Having too many wires in a small area is known as ***routing congestion***.

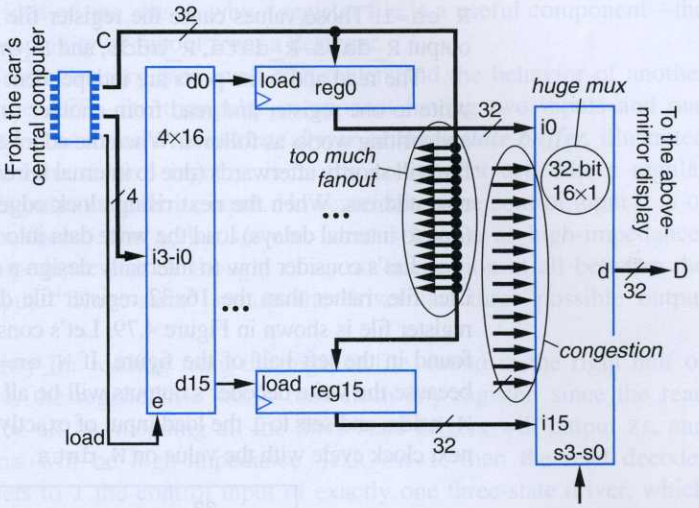


Figure 4.77 Above-mirror display design, assuming sixteen 32-bit registers. The mux has too many input wires, resulting in congestion. Also, the data lines C are fanned out to too many registers, resulting in weak current.

Likewise, consider routing the data input to all sixteen registers. Each data input wire is being branched into sixteen subwires. Imagine electric current being like a river of water—branching a main river into sixteen smaller rivers will yield much less water flow in each smaller river than in the main river. Likewise, branching a wire, known as ***fanout***, can only be done so many times before the branched wires' currents are too small to sufficiently control transistors. Furthermore, low-current wires may be very slow, so fanout can create long delays over wires too.

The fanout and routing congestion problems illustrated in the previous example can be solved by observing that the system never needs to load more than one register at a time, nor read more than one register at a time either. An *MxN* register file solves the fanout and congestion problems by grouping the *M* registers into a single component, with that compo-

nent having a single *N*-bit-wide data input, and a single *N*-bit-wide data output. The wiring inside the component is done carefully to handle fanout and congestion. Figure 4.78 shows a block symbol of a 16x32 register file (16 registers, each 32-bits wide).

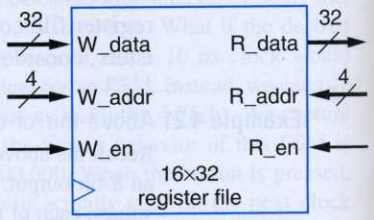


Figure 4.78 16x32 register file symbol.

Consider writing a value to a register in a register file. We would place the data to be written on the input `W_data`. We then need a way to indicate which register to write this data into. Since there are 16 registers, four bits are needed to specify a particular register. Those four bits are called the register's ***address***. We would thus set input `W_addr` to the desired register's address. For example, to write to register 7, we would set `W_addr=0111`. To indicate that we actually want to write on a particular clock cycle (we won't want to write on every cycle), we would set the input `W_en` to `1`. The collection of inputs `W_data`, `W_addr`, and `W_en` is known as a register file's ***write port***.

Reading is similar. We would specify the register to read on input `R_addr`, and set `R_en=1`. Those values cause the register file to output the addressed register's contents onto output `R_data`. `R_data`, `R_addr`, and `R_en` are known as a register file's ***read port***.

The read and write ports are independent of one another. During one clock cycle, we can write to one register and read from another (or the same) register. Such simultaneous reading and writing works as follows. When the addresses appear at the register file's inputs, the register file will shortly afterwards (due to internal wire/gate delays) output the data corresponding to the read address. When the next rising clock edge arrives, the register file will shortly afterwards (due to internal delays) load the write data into the register corresponding to the write address.

Let's consider how to internally design a register file. For simplicity, consider a 4x32 register file, rather than the 16x32 register file described above. One internal design of a 4x32 register file is shown in Figure 4.79. Let's consider the circuitry for writing to this register file, found in the left half of the figure. If `W_en=0`, the register file won't write to any register, because the write decoder's outputs will be all `0`s. If `W_en=1`, then the write decoder decodes `W_addr` and sets to `1` the load input of exactly one register. That register will be written on the next clock cycle with the value on `W_data`.

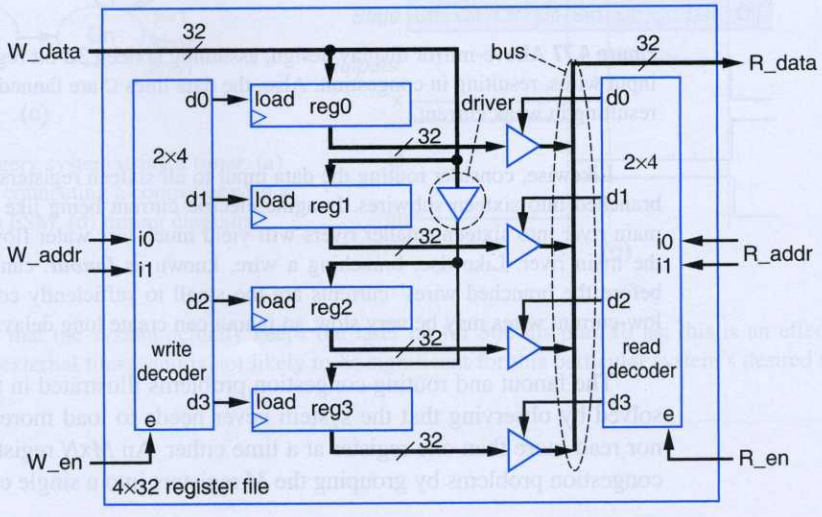
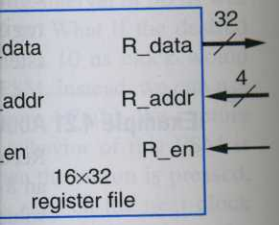


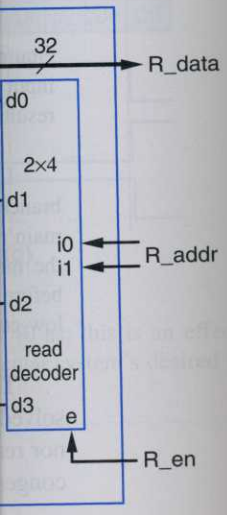
Figure 4.79 One possible internal design of a 4x32 register file.

e data output. The wiring
estion. Figure 4.78 shows
wide).



16x32 register file symbol.

the input `W_en` to `1`. The
gister file's ***write port***.
n input `R_addr`, and set
ed register's contents onto
ster file's ***read port***.
ng one clock cycle, we can
Such simultaneous reading
ster file's inputs, the register
ne data corresponding to the
file will shortly afterwards
nding to the write address.
plicity, consider a 4x32 reg-
e internal design of a 4x32
writing to this register file,
won't write to any register,
n the write decoder decodes
gister will be written on the



Notice the circled triangular one-input one-output component placed on the `W_data` line (there would actually be 32 such components since `W_data` is 32 bits wide). That component is known as a ***driver***, sometimes called a ***buffer***, illustrated in Figure 4.80(a). A driver's output value equals its input, but the output is a stronger (higher current) signal. Remember the fanout problem described in Example 4.21? A driver reduces the fanout problem. In Figure 4.79, the `W_data` lines only fanout to two registers before they go through the driver. The driver's output then fans out to only two more registers. Thus, instead of a fanout of four, the `W_data` lines have a fanout of only two (actually three if you count the driver itself). The insertion of drivers is beyond the scope of this book, and is instead a subject for a VLSI design book or an advanced digital design book. But seeing at least one example of the use of a driver hopefully gives you an idea of one reason why a register file is a useful component—the component hides the complexity of fanout from a designer.

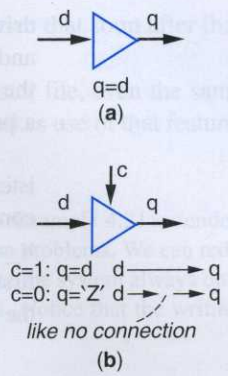


Figure 4.80 (a) driver, (b) three-state driver.

To understand the read circuitry, you must first understand the behavior of another new component in Figure 4.79—the triangular component having two inputs and one output. That component is known as a ***three-state driver*** or ***three-state buffer***, illustrated in Figure 4.80(b). When the control input `c` is `1`, the component acts like a regular driver—the component's output equals its input. However, when the control input `c` is `0`, the driver's output is neither `0` nor `1`, but instead what is known as high-impedance, written as "`Z`." High-impedance can be thought of as no connection at all between the driver's input and output. "Three-state" means the driver has three possible output states—`0`, `1`, and `Z`.

Such components are more commonly known as "tri-state" drivers rather than "three-state." But "tri-state" is a registered trademark of National Semiconductor Corp., so rather than putting the required trademark symbol after every use of the term "tri-state," many documents use the term "three-state."

Consider the circuitry for reading from the register file, found in the right half of Figure 4.79. If `R_en=0`, the register file won't read from any register, since the read decoder's outputs will be all `0`s, meaning all the three-state drivers will output `Z`s, and thus the output `R_data` will be high-impedance. If `R_en=1`, then the read decoder decodes `R_addr` and sets to `1` the control input of exactly one three-state driver, which will pass its register value through to the `R_data` output.

Be aware that each shown three-state driver actually represents a set of 32 three-state drivers, one for each of the 32 wires coming from the 32-bit registers and going to the 32-bit `R_data` output. All 32 drivers in a set are controlled by the same control input.

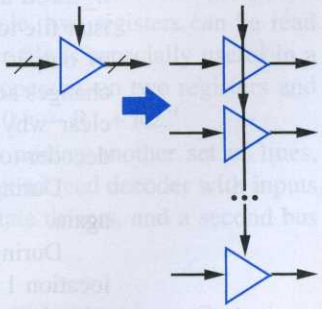


Figure 4.81 Each driver in Figure 4.79 is actually 32 drivers.

The wires fed by the various three-state drivers are known as a ***shared bus***, as indicated in Figure 4.79 and detailed in Figure 4.81. A shared bus is a popular alternative to a multiplexor when each mux data input is many bits wide and/or when there are many mux data inputs, because a shared bus results in less congestion.

Notice that the register file design scales well to larger numbers of registers. The write data lines can be

driven by more drivers if necessary. The read data lines are fed from three-state drivers, and thus there is no congestion at a single multiplexor. The reader may wish to compare the register file design in Figure 4.79 with the design in Figure 4.5, which was essentially a poor design of a register file.

Figure 4.82 provides example timing diagrams describing writing and reading of a register file. During *cycle1*, the contents of the register file are unknown, so the register file's contents are shown as "?." During *cycle1*, we set `W_data`=9 (in binary, of course), `W_addr`=3, and `W_en`=1. Those values cause a write of 9 to register file location 3 on the first clock edge. Notice that we had set `R_en`=0, so the register file outputs nothing ("Z"), and the value we put on `R_addr` does not matter (the value is a "don't care," written as "X").

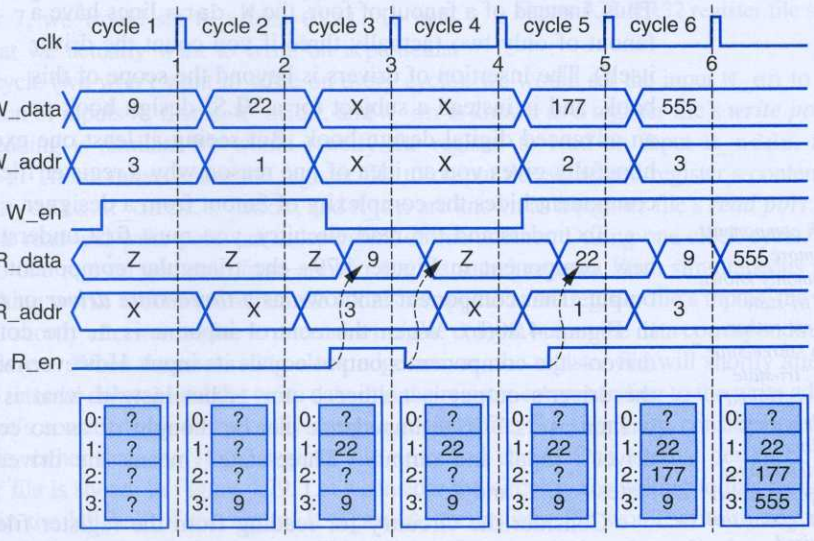


Figure 4.82 Writing and reading a register file.

During *cycle2*, we set `W_data`=22, `W_addr`=1, and `W_en`=1. These values cause a write of 22 to register file location 1 on clock edge 2.

During *cycle3*, we set `W_en`=0, so then it doesn't matter to what values we set `W_data` and `W_addr`. We also set `R_addr`=3 and `R_en`=1. Those values cause the register file to read out the contents of register file location 3 onto `R_data`, causing `R_data` to output 9. Notice that the reading is not synchronized to clock edge 3—`R_data` changes soon after `R_en` becomes `1`. Examining the design of Figure 4.79 should make clear why reading is not synchronous—setting `R_en` to `1` simply enables the output decoder to turn on one set of the three-state buffers.

During *cycle4*, we return `R_en` to `0`. Note that this causes `R_data` to become "Z" again.

During *cycle5*, we want to simultaneously write and read the register file. We read location 1 (which causes `R_data` to become 22) while simultaneously writing location 2 with the value 177.

Finally, during *cycle6*, we want to simultaneously read and write the same register file location. We set `R_addr`=3 and `R_en`=1, causing location 3's contents of 9 to appear on `R_data` shortly after setting those values. We also set `W_addr`=3, `W_data`=555, and

W_en=1. On clock edge 6, 555 thus gets stored into location 3. Notice that soon after that clock edge, R_data also changes to 555.

The ability to simultaneously read and write locations of a register file, even the same location, is a widely used feature of register files. The next example makes use of that feature.

Example 4.22 Above-mirror display system using a 16x32 register file

Example 4.2 used four 8-bit registers for an above-mirror display system. Example 4.21 extended the system to use sixteen 32-bit registers, resulting in fanout and congestion problems. We can redo that example using a register file. The design is shown in Figure 4.83. Since the system always outputs one of the register values to the display, we tied the R_en input to 1. Notice that the writing and reading of particular registers are independent of one another.

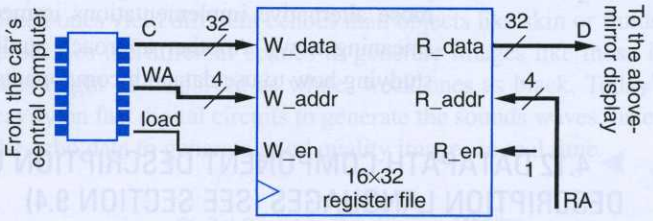


Figure 4.83 Above-mirror display design, using a register file.

A register file having one read port and one write port is sometimes referred to as a ***dual-ported register file***. To make clear that the two ports consist of one read port and one write port, such a register file may be referred to as follows: *dual-ported (1 read, 1 write) register file*.

A register file may actually have just one port, which would be used for both reading and writing. Such a register file has only one set of data lines that can serve as inputs or outputs, one set of address inputs, an enable input, and one more input indicating whether we wish to write or read the register file. Such a register file is known as a ***single-ported register file***.

Multiported (2 Read, 1 Write) Register File. Many register files have three ports: one write port, and two read ports. Thus, in the same clock cycle, two registers can be read simultaneously, and another register written. Such a register file is especially useful in a microprocessor, since a typical microprocessor instruction operates on two registers and stores the result in a third register, like in the instruction "R0 <— R1 + R2."

We can create a second read port in a register file by adding another set of lines, Rb_data, Rb_addr, and Rb_en. We would introduce a second read decoder with inputs Rb_addr and enable input Rb_en, a second set of three-state drivers, and a second bus connected to the Rb_data output.

Other Register File Variations. Register files come in all sorts of configurations. Typical numbers of registers in a register file range from 4 to 1024, and typical register widths range from 8 bits to 64 bits per register, but sizes may vary beyond those ranges. Register files may have one port, two ports, three ports, or even more, but increasing to

The most ports I've seen on a register file in a product was 10 read ports and 5 write ports.

many more than three ports can slow down the register file's performance and increase its size significantly, due to the difficulty of routing all those wires around inside the register file. Nevertheless, you'll occasionally run across register files with perhaps 3 write ports and 3 read ports, when concurrent access is critical.

▶ 4.11 DATAPATH COMPONENT TRADEOFFS (SEE SECTION 6.4)

For each datapath component introduced in previous sections, we created the most basic and easy-to-understand implementation. This section, which physically appears in the book as Section 6.4, describes alternative implementations of several datapath components. Each alternative trades off one design criteria for another—most of those alternatives trade off larger size in exchange for less delay. One approach to using this book involves studying those alternative implementations immediately after studying the basic implementations (meaning now). Another approach studies those alternative implementations later, after studying how to use datapath components during register-transfer level design.

▶ 4.12 DATAPATH COMPONENT DESCRIPTION USING HARDWARE DESCRIPTION LANGUAGES (SEE SECTION 9.4)

This section, which physically appears in the book as Section 9.4, shows how to use HDLs to describe several datapath components. One approach to using this book studies such HDL use now, while another approach studies such HDL use later.

▶ 4.13 PRODUCT PROFILE: AN ULTRASOUND MACHINE

If you or someone you know has ever had a baby, then you may have seen ultrasound images of that baby before he/she was born, like the images of a fetus' head in Figure 4.84(a).

(a)

(b)

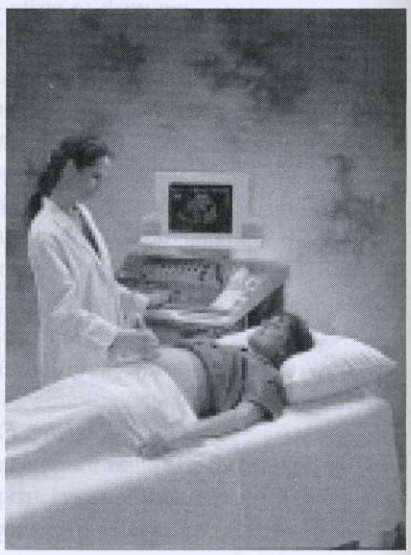


Figure 4.84 (a) Ultrasound image of a fetus, created using an ultrasound device that is simply placed on the mother's abdomen (b) and that forms the image by generating sound waves and listening to the echoes. Photos courtesy of Philips Medical Systems.

That image wasn't taken by a camera somehow inserted into the uterus, but rather by an ultrasound machine pressed against the mother's skin and pointed toward the fetus. Ultrasound imaging is now common practice in obstetrics—mainly helping doctors to track the fetus' progress and correct potential problems early, but also giving parents a huge thrill when they get their first glimpse of their baby's head, hands, and little feet!

Functional Overview

This section briefly describes the key functional ideas of how ultrasound imaging works. Digital designers don't typically work in a vacuum—instead, they apply their skills to particular applications, and thus designers typically learn the key functional ideas underlying those applications. We therefore introduce you to the basic ideas of ultrasound applications. Ultrasound imaging works by sending sound waves into the body and listening to the echoes that return. Objects like bones yield different echoes than objects like skin or fluids, so an ultrasound machine processes the different echoes to generate images like those in Figure 4.84(a)—strong echoes might be displayed as white, weak ones as black. Today's ultrasound machines rely heavily on fast digital circuits to generate the sounds waves, listen to the echoes, and process the echo data to generate good quality images in real time.

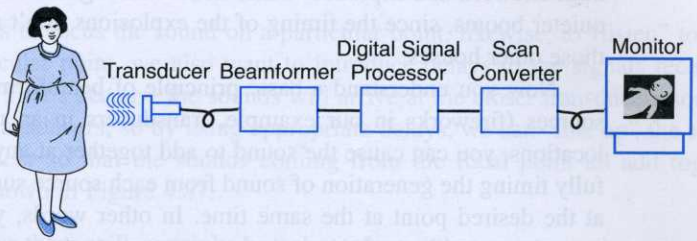


Figure 4.85 Basic components of an ultrasound machine.

Figure 4.85 illustrates the basic parts of an ultrasound machine. Let's discuss each part individually.

Transducer

A ***transducer*** converts energy from one form to another. You're certainly familiar with one type of transducer, a stereo speaker, which converts electrical energy into sound by changing the current in a wire, which causes a nearby magnet to move back and forth, which pushes the air and hence creates sound. Another familiar transducer is a dynamic microphone, which converts sound into electrical energy by letting sound waves move a magnet, which induces current changes in a nearby wire. In an ultrasound machine, the transducer converts electrical pulses into sound pulses, and sound pulses (the echoes) into electrical pulses, but the transducer uses piezoelectric crystals instead of magnets. Applying electric current to such a crystal causes the crystal to change shape rapidly, or vibrate, thus generating sound waves—typically in the 1 to 30 Megahertz frequency range. Humans can't hear much above 30 kilohertz—the term "ultrasound" refers to the fact that the frequency is beyond human hearing. Inversely, sound waves (echoes) hitting the crystal create electric current. An ultrasound machine's transducer component may contain hundreds of such crystals, which we can think of as hundreds of transducers. Each such transducer is considered to form a ***channel***.

Real designers must often learn about the domain for which they will design. Many designers consider such learning about domains, like ultrasound, as one of the fascinating features of the job.

Beamformer

A ***beamformer*** *electronically* "focuses" and "steers" the sound beam of an array of transducers to or from particular focal points, without actually moving any hardware like a dish to obtain such focusing and steering.

To understand the idea of beamforming, we must first understand the idea of additive sound. Consider two loud fireworks exploding at the same time, one 1 mile away from you, and the other 2 miles away. You'll hear the closer firework after about 5 seconds—assuming sound travels 0.2 miles/second (or 1 mile every 5 seconds)—a reasonable approximation. You'll hear the farther firework after about 10 seconds. So you'll hear "boom ... (five seconds pass) ... boom." However, suppose instead that the closer firework exploded 5 seconds later than the farther one. Then you'll hear both at the same time—one big "BOOM!" That's because the two sounds add together. Now suppose there are 100 fireworks spread throughout a city, and you want all the sound from those fireworks to reach one particular house (perhaps somebody you don't like very much) at the same time. You can do this by exploding the closer fireworks later than the farther fireworks. If you time everything just right, that particular house will hear a tremendously loud single "BOOOOOM!!!!" probably rattling the house's walls pretty well, as if one huge firework had exploded. Other houses throughout the city will instead hear a series of quieter booms, since the timing of the explosions don't result in all the sounds adding at those other houses.

Now you understand a basic principle of beamforming: If you have multiple sound sources (fireworks in our example, transducers in an ultrasound machine) in different locations, you can cause the sound to add together at any desired point in space, by carefully timing the generation of sound from each source such that all the sound waves arrive at the desired point at the same time. In other words, you can *electronically* focus and steer the sound beam by introducing appropriate delays. Focusing and steering the sound to a particular point is useful because then *that point will produce a much louder echo than all other points*, so we can easily hear the echo from that point over all the echoes from other points.

Figure 4.86 illustrates the concept of electronic focusing and steering, using two sound sources to focus and steer a beam to a desired point *X*.

At the first time step (Figure 4.86(a)), the bottom source has begun transmitting its sound wave. After two time steps (Figure 4.86(b)), the top source has begun transmitting its sound wave. After three time steps (Figure 4.86(c)), the waves from both sensors reach the focal point, adding together. They'll continue adding as long as the waves from both sources are in phase with one another. We can simplify the drawing by showing only the lines from the sources to the focal point, as shown in Figure 4.86(d).

An ultrasound machine uses this ability to electronically focus and steer sound, in order to scan, point by point, the entire region in front of the transducers. The machine scans each point tens of times per second.

For each focal point, the machine needs to listen to the echo that comes back from whatever object is located at the focal point, to determine if that object is bone, skin, blood, etc., utilizing the fact that each such object generates a different echo. Remember, the echo from the focal point will be louder than echoes from other points, because the sound adds at that point. We can use beamforming to also focus in on a particular point in space that we want to *listen* to. In the same way that we generated sound pulses with par-

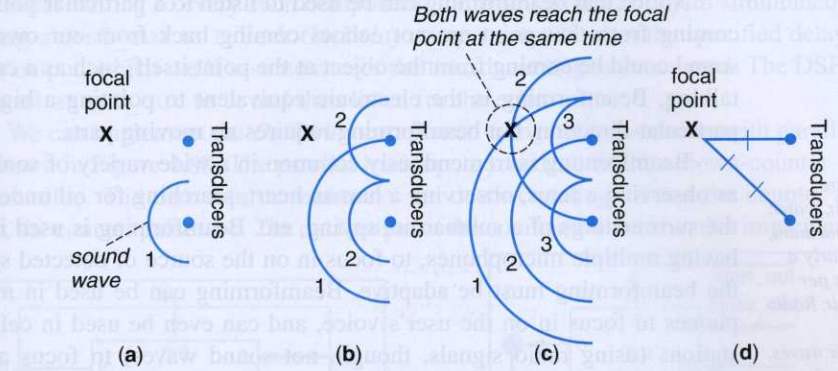


Figure 4.86 Focusing sound at a particular point using beamforming: (a) first time step—only the bottom transducer generates sound, (b) second time step—the top transducer now generates sound too, (c) third time step—the two sound waves add at the focal point, (d) an illustration showing that the top transducer is two time steps away from the focal point, while the bottom transducer is three time steps away, meaning the top transducer should generate sound one time step later than the bottom transducer.

ticular delays to focus the sound on a particular point, likewise, to "listen" to the sounds from a particular point, we also want to introduce delays to the signals received by the transducers. That's because the sounds will arrive at the closer transducers sooner than at the farther transducers, so by using appropriate delays, we can "line up" the signals from each transducer so that the sounds coming from the focal point all add together. This concept is shown in Figure 4.87.

Note that there will certainly be echoes from other points in the region, but those coming from the focal point will be much stronger—hence, the weaker echoes can be filtered out.

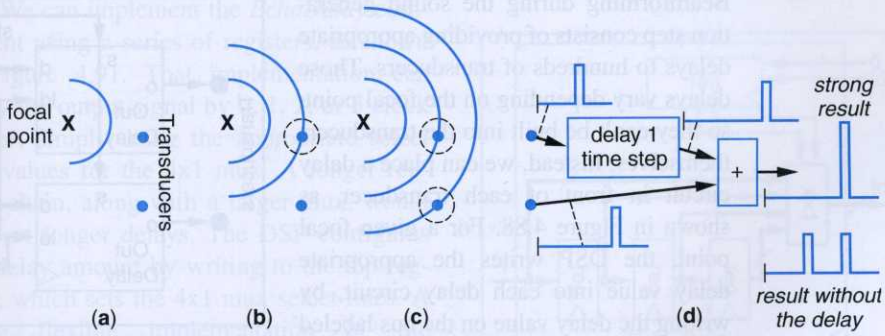


Figure 4.87 Listening to sound from a particular point using beamforming: (a) first time step, (b) second time step—the top transducer has heard the sound first, (c) third time step—the bottom transducer hears the sound at this time, (d) delaying the top transducer by one time step results in the waves from the focal point adding, amplifying the sound.

Notice that beamforming can be used to listen to a particular point even if the sounds coming from that point are not echoes coming back from our own sound pulses—the sound could be coming from the object at the point itself, such as a car engine or a person talking. Beamforming is the electronic equivalent to pointing a big parabolic dish in a particular direction, but beamforming requires no moving parts.

Sound waves are vibrations of air, water, or some other medium, traveling at nearly a thousand miles per hour through air. Radio waves are electromagnetic waves, requiring no such medium (they can travel through space), and traveling at nearly a billion miles per hour (the speed of light in a vacuum).

Beamforming is tremendously common in a wide variety of sonar applications, such as observing a fetus, observing a human heart, searching for oil underground, monitoring the surroundings of a submarine, spying, etc. Beamforming is used in some hearing aids having multiple microphones, to focus in on the source of detected speech—in that case, the beamforming must be adaptive. Beamforming can be used in multimicrophone cell phones to focus in on the user's voice, and can even be used in cellular telephone base stations (using radio signals, though, not sound waves) to focus a signal going to or coming from a cell phone.

Signal Processor, Scan Converter, and Monitor

The signal processor analyzes the echo data of every point in the scanned region, by filtering out noise (see Section 5.13 for a discussion on filtering), interpolating between points, assigning a level of gray to each point depending on the echoes heard (echoes corresponding to bones might be shaded as white, liquid as black, and skin as gray, for example), and other tasks. The result is a gray-scale image of the region. The scan converter steps through this image to generate the necessary signals for a black-and-white monitor, and the monitor displays the image.

Digital Circuits in an Ultrasound Machine's Beamformer

Much of the control and signal processing tasks in an ultrasound machine are carried out using software running on one or more microprocessors, typically special microprocessors specifically designed for digital signal processing, known as digital signal processors, or DSPs. But certain tasks are much more amenable to custom digital circuitry, such as those in the beamformer.

Sound Generation and Echo Delay Circuits

Beamforming during the sound generation step consists of providing appropriate delays to hundreds of transducers. Those delays vary depending on the focal point, so they can't be built into the transducers themselves. Instead, we can place a delay circuit in front of each transducer, as shown in Figure 4.88. For a given focal point, the DSP writes the appropriate delay value into each delay circuit, by writing the delay value on the bus labeled `delay_out`, writing the "address" on the lines labeled `addr`, and enabling the decoder. The decoder will thus set the load line of one of the *OutDelay* components.

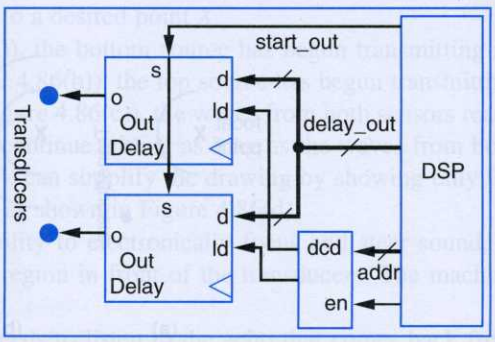


Figure 4.88 Transducer output delay circuits for two channels.

After writing to every such component, the DSP starts all of them simultaneously by setting `start_out` to `1`. Each *OutDelay* component will, after the specified delay, pulse its `o` output, which we'll assume causes the transducer to generate sound. The DSP would then set `start_out` to `0`, and then listen for the echo.

We can implement the *OutDelay* component using a down-counter with parallel load, as shown in Figure 4.89. The parallel load inputs `L` and `ld` load the down-counter with its count value. The `cnt` input commences the down-counting—when the counter reaches zero, the counter pulses `tc`. The data output of the counter is unused in this implementation.

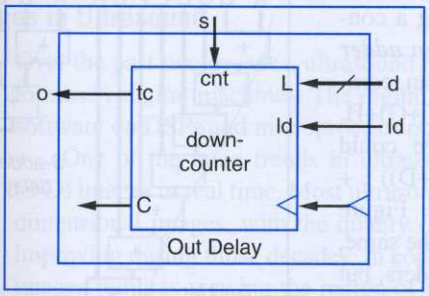


Figure 4.89 OutDelay circuit.

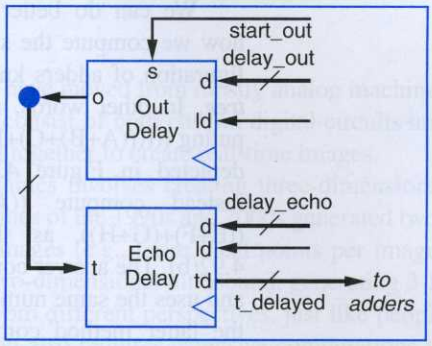


Figure 4.90 Tranducer output and echo delay circuits for one channel.

After the ultrasound machine sends out sound waves focused on a particular focal point, the machine must listen to the echo coming back from that focal point. This listening requires appropriate delays for each transducer to account for the differing distances of each transducer from the focal point. Thus, each transducer needs another delay circuit for delaying the received echo signal, as shown in Figure 4.90. The *EchoDelay* component receives on input `t` the signal from the transducer, which we'll assume has been digitized into a stream of *N*-bit values. The component should output that signal on output `t_delayed`, delayed by the appropriate amount. The delay amount can be written by the DSP using the component's `d` and `ld` inputs.

We can implement the *EchoDelay* component using a series of registers, as shown in Figure 4.91. That implementation can delay the output signal by 0, 1, 2, or 3 clock cycles, simply using the appropriate select line values for the 4x1 mux. A longer register chain, along with a larger mux, would support longer delays. The DSP configures the delay amount by writing to the top register, which sets the 4x1 mux select lines. A more flexible implementation of the *EchoDelay* component would instead use a timer component.

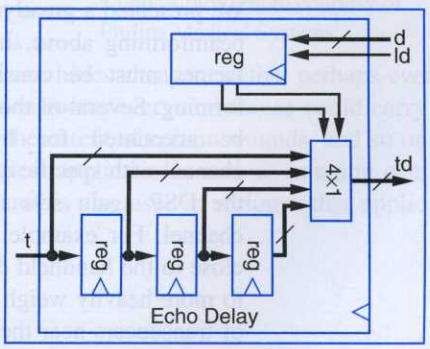


Figure 4.91 EchoDelay circuit.

Summation Circuits—Adder Tree

The output of each transducer, appropriately delayed, should be summed to create a single echo signal from the focal point, as was illustrated in Figure 4.87. That illustration had only two transducers, and thus only one adder. What if we have 256 transducers, as would be more likely in a real ultrasound machine? How do we add 256 values? We could add the values in a linear way, as illustrated on the left side of Figure 4.92(a) for eight values. The delay of that circuit is roughly equal to the delay of seven adders. For 256 values, the delay would roughly be that of 255 adders. That's a very long delay.

We can do better by reorganizing how we compute the sum, using a configuration of adders known as an ***adder tree***. In other words, rather than computing ((((((A+B)+C)+D)+E)+F)+G)+H, depicted in Figure 4.92(a), we could instead compute ((A+B)+(C+D)) + ((E+F)+(G+H)), as shown in Figure 4.92(b). The answer comes out the same, and uses the same number of adders, but the latter method computes four additions in parallel, then two additions in parallel, and then performs a last addition. The delay is thus only that of three adders. For 256 values, the tree's first level would compute 128 additions in parallel, the second level would compute 64 additions, then 32, then 16, then 8, then 4, then 2, and finally 1 last addition. Thus, that adder tree would have eight levels, meaning a total delay equal to eight adder delays. That's a lot faster than 256 adder delays—*32 times faster*, in fact.

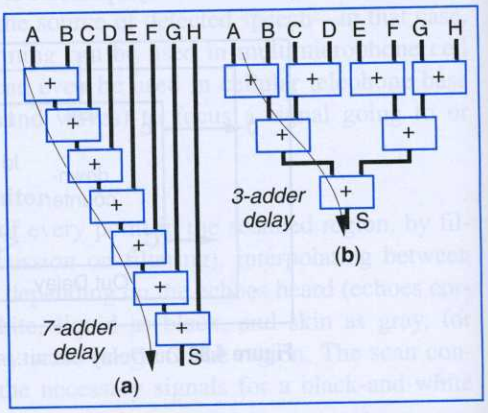


Figure 4.92 Adding many numbers: (a) linearly, (b) using an adder tree. Note that both methods use seven adders.

The output of the adder tree can be fed into a memory to keep track of the results for the DSP, which may access the results sometime after they are generated.

Multipliers

We presented a greatly simplified version of beamforming above. In reality, many other factors must be considered during beamforming. Several of those considerations can be accounted for by multiplying each channel with specific constant values, which the DSP again sets individually for each channel. For example, focusing on a point close to the handheld device may require us to more heavily weigh the incoming signals of transducers near the center of the device. A channel may therefore actually include a multiplier, as shown in Figure 4.93. The DSP

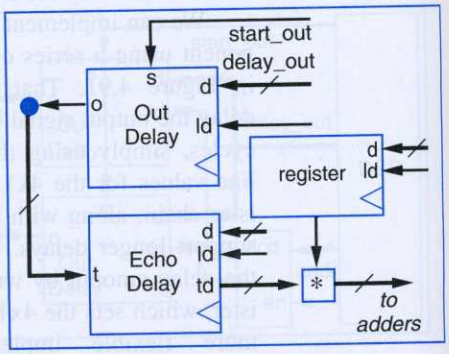


Figure 4.93 Channel extended with a multiplier.

could write to the register shown, which would represent a constant by which the transducer signal would be multiplied.

Our introduction of the ultrasound machine is greatly simplified from a real machine, yet even in this simplified introduction, you can see many of this chapter's datapath components in use. We used a down-counter to implement the *OutDelay* component, and several registers along with muxes for the *EchoDelay* component. We used many adders to sum the incoming transducer signals. And we used a multiplier to weigh those incoming signals.

Future Challenges in Ultrasound

Over the past two decades, ultrasound machines have moved from mostly analog machines to mostly digital machines. The digital systems consist of both custom digital circuits and software on DSPs and microprocessors, working together to create real-time images.

One of the main trends in ultrasound machines involves creating three-dimensional (3-D) images in real time. Most ultrasound machines of the 1990s and 2000s generated two-dimensional images, with the quality of those images (e.g., more focal points per image) improving during those decades. In contrast to two-dimensional ultrasound, generating 3-D images requires viewing the region of interest from different perspectives, just like people view things from their two eyes. Such generation also requires extensive computations to create a 3-D image from the two (or more) perspectives. The result is a picture like that in Figure 4.94.

That's a fetus' face. Impressive, isn't it? Keep in mind that image is made solely from sound waves bouncing into a woman's womb. Color can also be added to distinguish among different fluids and tissues. Those computations take time, but faster processors, coupled with clever custom digital circuits, are bringing real-time 3-D ultrasound closer to reality.

Figure 4.94 3-D ultrasound image of a fetus's face. Photo courtesy of Philips Medical Systems.

Another trend is toward making ultrasound machines smaller and lighter, so that they can be used in a wider variety of health care situations. Early machines were big and heavy, with more recent ones coming on rollable carts. Some recent versions are handheld. A related trend is making ultrasound machines cheaper, so that perhaps every doctor could have a machine in every examination room, every ambulance could carry a machine to help emergency personnel ascertain the extent of certain wounds, and so on.

Ultrasound is used for numerous other medical applications, such as imaging of the heart to detect artery or valve problems. Ultrasound is also used in various other applications, like submarine region monitoring.

▶ 4.14 CHAPTER SUMMARY

This chapter began (Section 4.1) by introducing the idea of new building blocks intended for common operations on multibit data, with those blocks known as datapath components. The chapter then introduced a number of datapath components, including registers,

adders, comparators, multipliers, subtractors, arithmetic-logic units, shifters, counters, timers, and register files. For each component, the chapter examined two aspects: the internal design of the component, and the use of the component as part of a datapath to implement a desired system.

The chapter ended (Section 4.13) by describing some basic principles underlying the operation of an ultrasound machine, and showing how several of the datapath components might be used to implement parts of such a machine. One thing you might notice is how designing a real ultrasound machine would require some knowledge of the domain of ultrasound. The requirement that a software programmer or digital designer have some understanding of an application domain is quite common.

In the coming chapter, you will apply your knowledge of combinational logic design, sequential logic design (controller design), and datapath components, to build digital circuits that can implement general and powerful computations.

▶ 4.15 EXERCISES

An asterisk (*) indicates an especially challenging problem.

For exercises relating to datapath components, each problem may indicate whether the problem emphasizes the component's internal design or the component's use.

SECTION 4.2: REGISTERS

4.1 Trace the behavior of an 8-bit parallel-load register with 8-bit input *I*, 8-bit output *Q*, and load control input `ld` by completing the timing diagram in Figure 4.95.

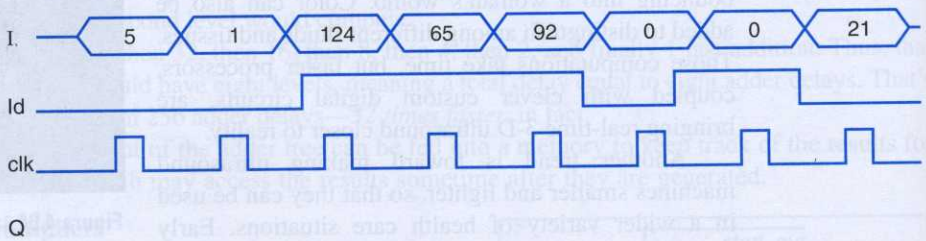


Figure 4.95 Timing diagram.

4.2 Trace the behavior of an 8-bit parallel-load register with 8-bit input *I*, 8-bit output *Q*, load control input `ld`, and synchronous clear input `clr` by completing the timing diagram in Figure 4.96.

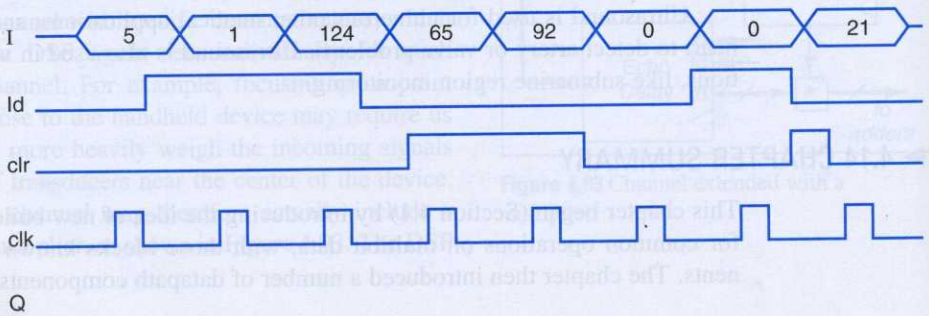


Figure 4.96 Timing diagram.