CHAPTER 4

DATAPATH COMPONENTS

4.1 EXCERCISES

Exercises marked with an asterisk (*) represent especially challenging problems.

For exercises relating to datapath components, each problem indicates 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.

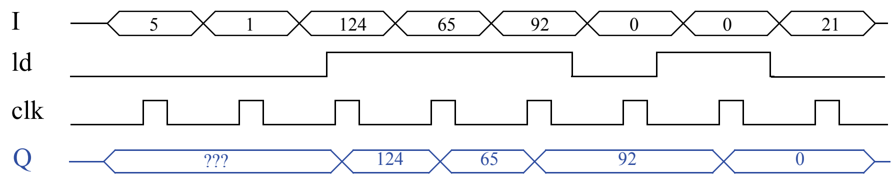


Figure 4.95

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.

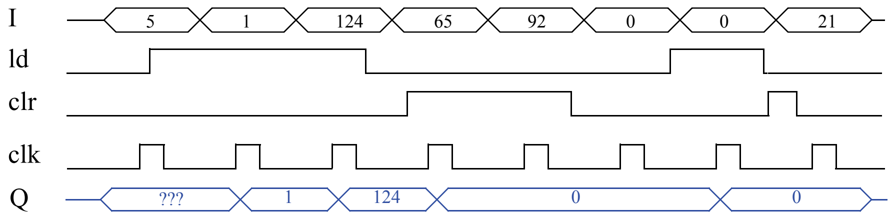
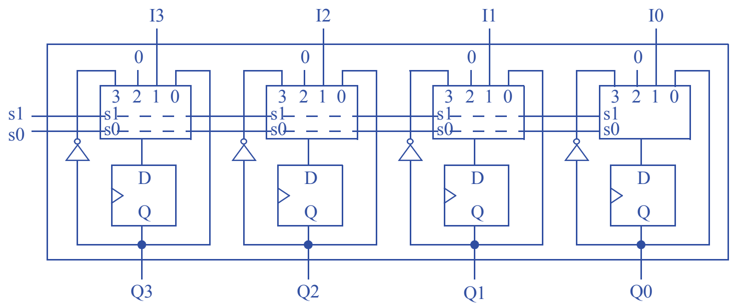
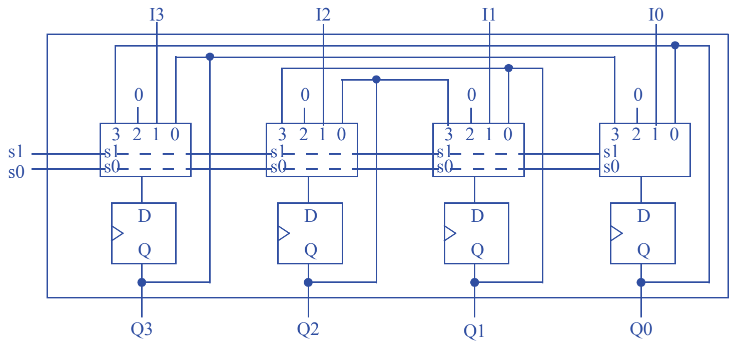


Figure 4.96

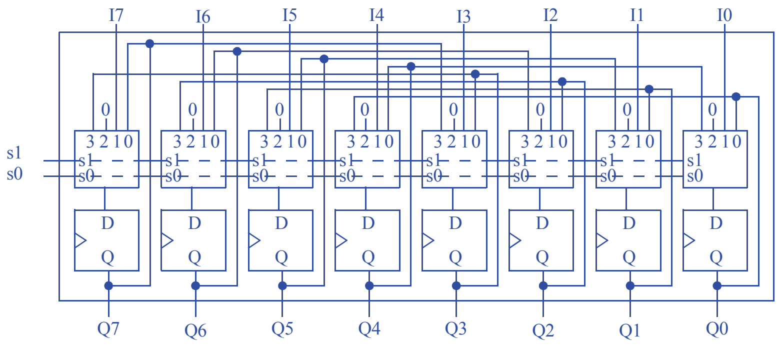
4.3 Design a 4-bit register with 2 control inputs s1 and s0, 4 data inputs I3, I2, I1 and I0, and 4 data outputs Q3, Q2, Q1 and Q0. When s1s0=00, the register maintains its value. When s1s0=01, the register loads I3..I0. When s1s0=10, the register clears itself to 0000. When s1s0=11, the register complements itself, so for example 0000 would become 1111, and 1010 would become 0101. *(Component design problem)*.



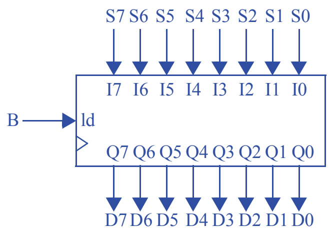
4.4 Repeat the previous problem, but when s1s0=11, the register reverses its bits, so 1110 would become 0111, and 1010 would become 0101. *(Component design problem)*.



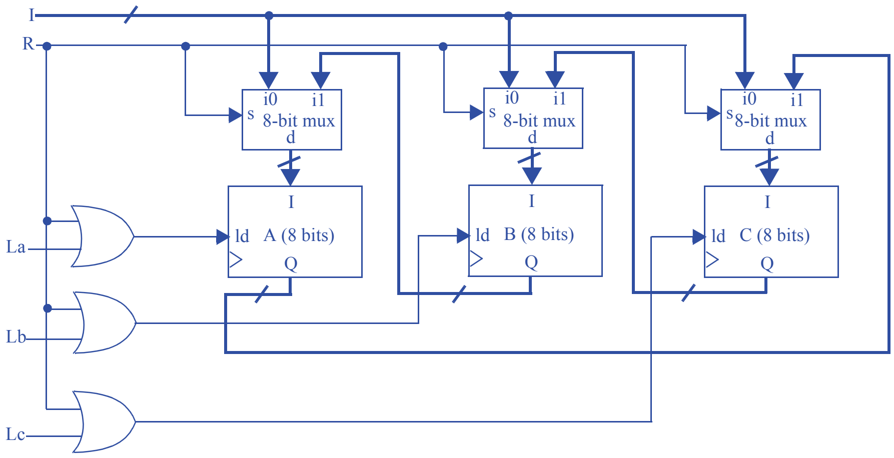
4.5 Design an 8-bit register with 2 control inputs s1 and s0, 8 data inputs I7..I0, and 8 data outputs Q7..Q0. s1s0=00 means maintain the present value, s1s0=01 means load, and s1s0=10 means clear. s1s0=11 means to swap the high nibble with the low nibble (a nibble is 4 bits), so 11110000 would become 00001111, and 11000101 would become 01011100. *(Component design problem)*.



4.6 The radar gun used by a police officer outputs a radar signal and measures the speed of the cars as they pass. However, when the officer wants to ticket an individual for speeding, he must save the measured speed of the car on the radar unit. Build a system to implement a speed save feature for the radar gun. The system has an 8-bit speed input S, an input B from the save button on the radar gun, and an 8-bit output D that will be sent to the radar gun's speed display. *(Component use problem)*.

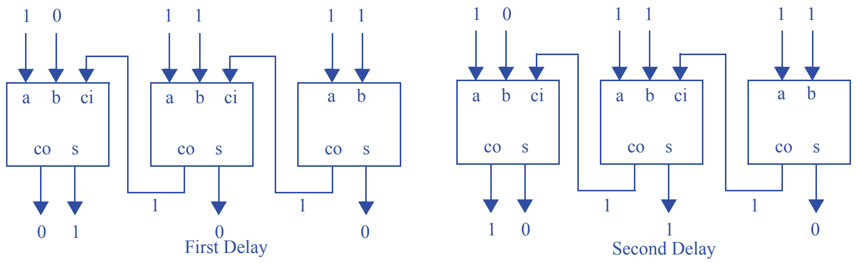


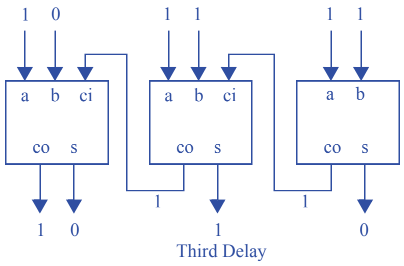
4.7 Design a system with an 8-bit input I that can be stored in 8-bit registers A, B, and/or C when input La, Lb, and/or Lc is 1, respectively. So if inputs La and Lb are 1, then registers A and B will be loaded with input I, but register C will keep its current value. Furthermore, if input R is 1, then the register values swap such that A=B, B=C, and C=A. Input R has priority over the L inputs. The system has one clock input also. (Component use problem.)



Section 4.3: Adders

4.8 Trace the values appearing at the outputs of a 3-bit carry-ripple adder for every one-full-adder-delay time period when adding 111 with 011. Assume all inputs were previously 0 for a long time.

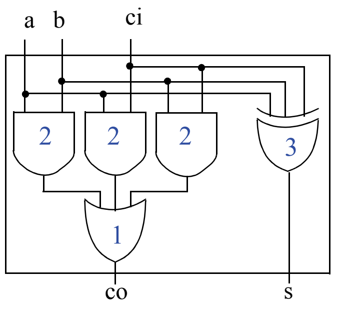




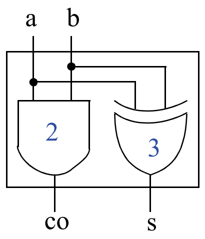
4.9 Assuming all gates have a delay of 1 ns, compute the longest time required to add two numbers using an 8-bit carry-ripple adder.

An 8-bit carry-ripple adder contains 7 full adders and 1 half adder. Each full adder has 2 gate delays and the half adder has 1 gate delay. Therefore a minimum of (7 FA * 2 gate delay/FA * + 1 HA * 1 gate delay/HA) * 1ns/gate delay = 15 ns is required to ensure that the carry-ripple adder's sum is correct.

4.10 Assuming AND gates have a delay of 2 ns, OR gates have a delay of 1 ns, and XOR gates have a delay of 3 ns, compute the longest time required to add two numbers using an 8-bit carry-ripple adder.



Full Adder

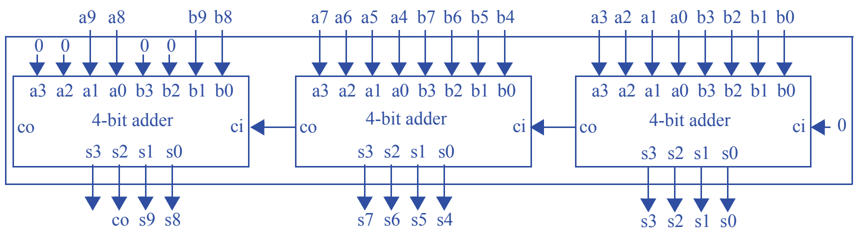


Half Adder

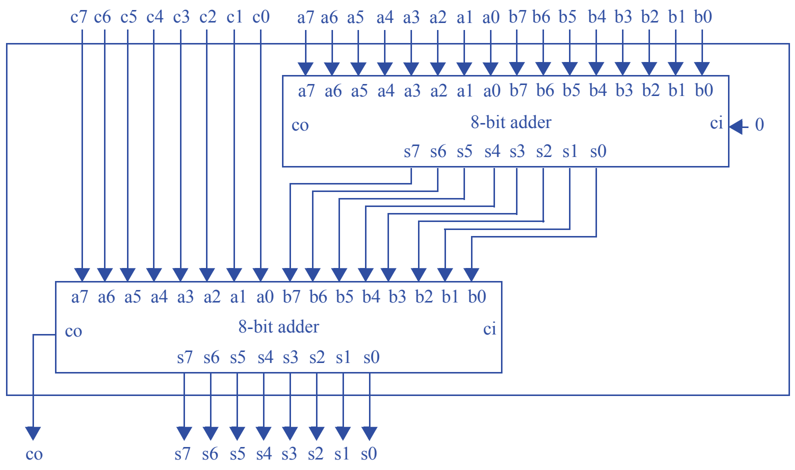
From the illustration above, we see that both the FA and HA have a maximum gate delay of 3 ns. Therefore, 8 adders * 3 ns/adder = 24 ns is required for an 8-bit carry-ripple adder to ensure a correct sum is on the adder's output.

An answer of 23 ns is also acceptable since the carry out of a half-adder will be correct after 2 ns, not 3 ns, and a half-adder may be used for adding the first pair of bits (least significant bits) if the 8-bit adder has no carry-in.

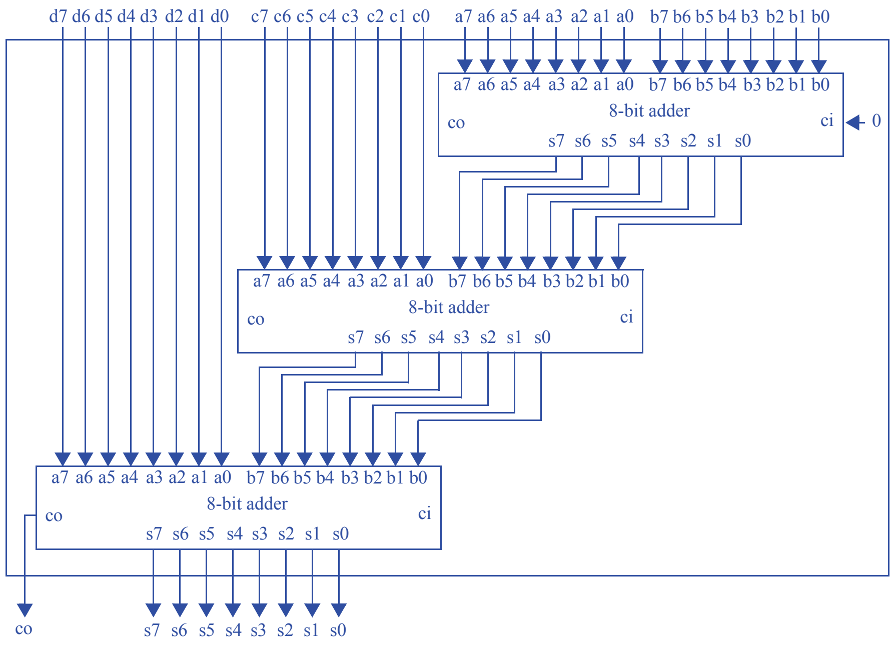
4.11 Design a 10-bit carry-ripple adder using 4-bit carry-ripple adders. *(Component use problem)*.



4.12 Design a system that computes the sum of three 8-bit numbers using 8-bit carry-ripple adders. *(Component use problem)*.

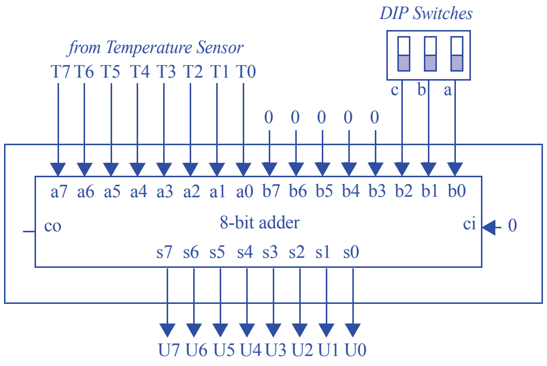


4.13 Design an adder that computes the sum of four 8-bit numbers, using 8-bit carry-ripple adders. *(Component use problem)*.

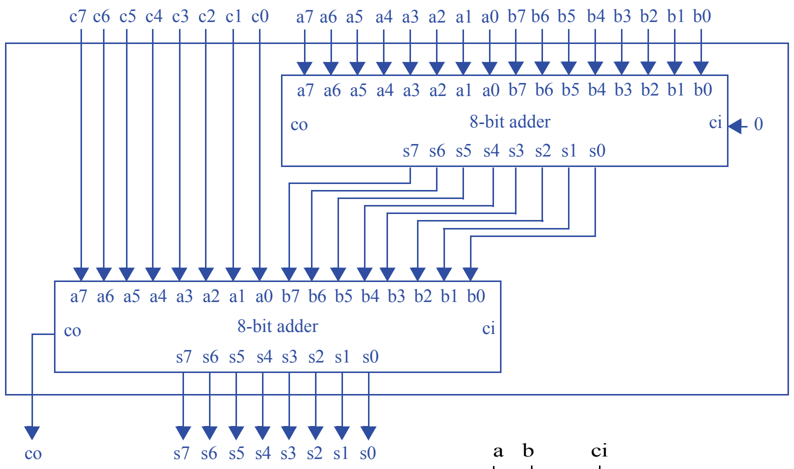


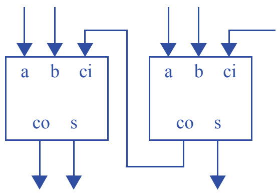
Another correct solution would add C+D, and then add the results to the result of A+B. That solution also uses just three adders, but actually has less delay.

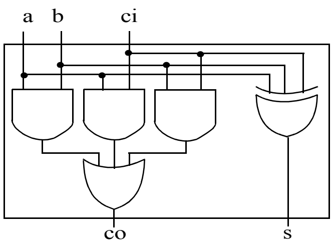
4.14 Design a digital thermometer system that can compensate for errors in the temperature sensing device's output T, which is an 8-bit input to the system. The compensation amount can be positive only and comes to the system as a 3-bit binary number c, b, and a (a is the least significant bit), which come from a 3-pin DIP switch. The system should output the compensated temperature on an 8-bit output U. *(Component use problem)*.



4.15 We can add three 8-bit numbers by chaining one 8-bit carry-ripple adder to the output of another 8-bit carry-ripple adder. Assuming every gate has a delay of 1 time-unit, compute the longest delay of this three 8-bit number adder. Hint: you may have to look carefully inside the carry-ripple adders, even inside the full-adders, to correctly compute the longest delay *(Component use problem)*.


c7 c6 c5 c4 c3 c2 c1 c0
a7 a6 a5 a4 a3 a2 a1 a0 b7 b6 b5 b4 b3 b2 b1 b0
a7 a6 a5 a4 a3 a2 a1 a0 b7 b6 b5 b4 b3 b2 b1 b0
co
8-bit adder
ci
0
s7 s6 s5 s4 s3 s2 s1 s0
a7 a6 a5 a4 a3 a2 a1 a0 b7 b6 b5 b4 b3 b2 b1 b0
co
8-bit adder
ci
s7 s6 s5 s4 s3 s2 s1 s0
co
s7 s6 s5 s4 s3 s2 s1 s0





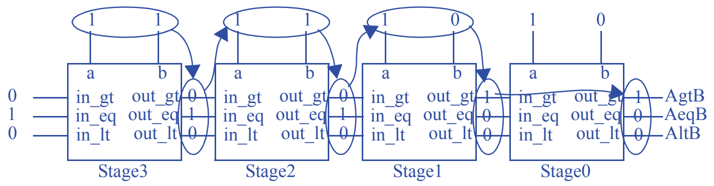
Full Adder

The above shows two 8-bit adders chained together to form a three 8-bit number adder. Each adder is made from eight full adders, whose configuration is shown at the bottom left. The bottom right shows the internal design of a full adder. Thus, the carry out of each stage requires 2 time units (following the problem's assumption of 1 time unit per gate), and the sum output requires 1 time unit.

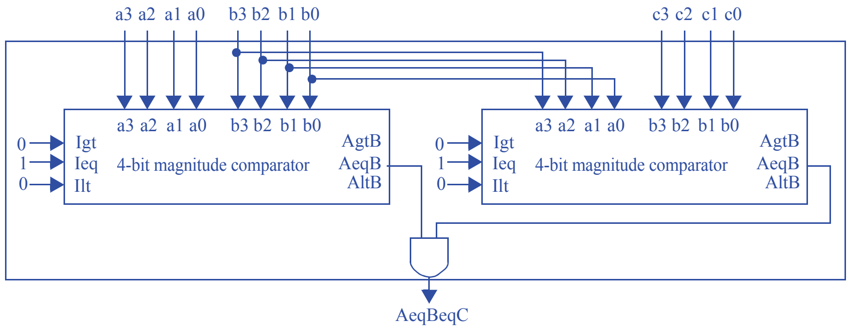
The longest delay in a full adder is 2 time units, from carry-in to carry-out. Since only 1 of the 8 full-adders in the top 8-bit adder has its carry-out unconnected (for a delay of 1 time unit), the delay from the top adder is 7*2 + 1 = 15 time units. The lower adder has its carry-out connected, however, giving the lower adder a delay of 8*2 = 16 time units. Thus, our adder has a total delay of 15 + 16 = 31 time units.

Section 4.4: Comparators

4.16 Trace through the execution of the 4-bit magnitude comparator shown in Figure 4.45 when a=15 and b=12. Be sure to show how the comparisons propagate thought the individual comparators.

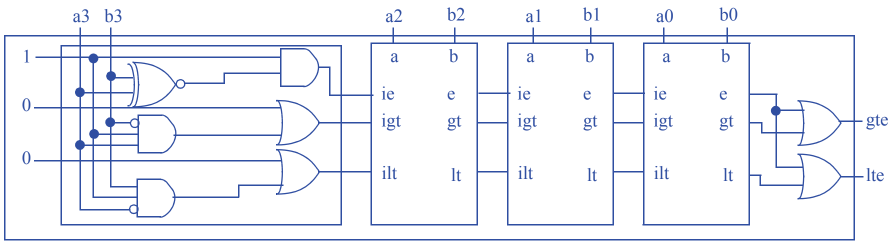


4.17 Design a system that determines if three 4-bit numbers are equal, by connecting 4-bit magnitude comparators together and using additional components if necessary. *(Component use problem)*.

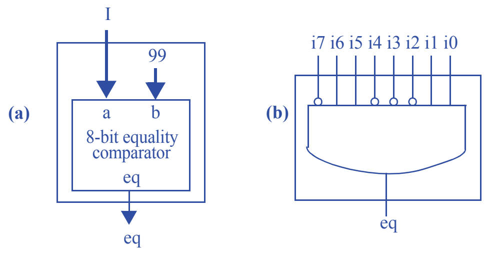


4.18 Design a 4-bit carry-ripple style magnitude comparator that has two outputs, a greater-than or equal-to output *gte*, and a less-than or equal-to output *lte*. Be sure to clearly show the equations used in developing the individual 1-bit comparators and how they are connected to form the 4-bit circuit. *(Component design problem)*.

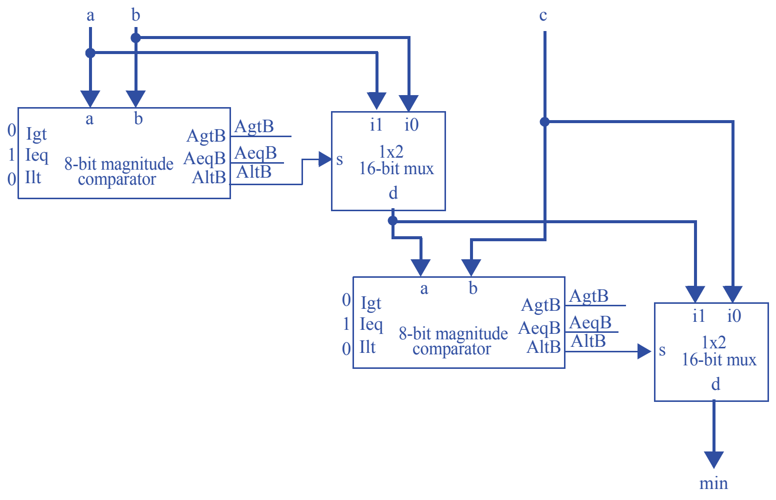
For each 1-bit comparator, assuming gte means "a >= b" and lte means "a <= b", gt = igt+((a XNOR b)*a*b'), lt = ilt+((a XNOR b)*a'*b), e = ie*(a XNOR b). Recall that XNOR detects equality. a*b' detects a>b. a'*b detects a<b.



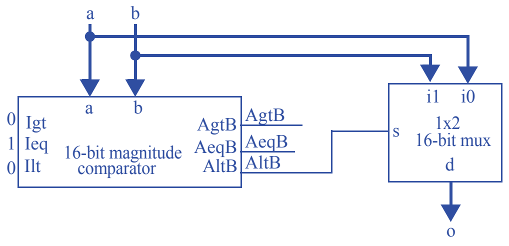
4.19 Design a circuit that outputs 1 if the circuit's 8-bit input equals 99: (a) using an equality comparator, (b) using gates only. *Hint*: In the case of (b), you need only 1 AND gate and some inverters. *(Component use problem)*.



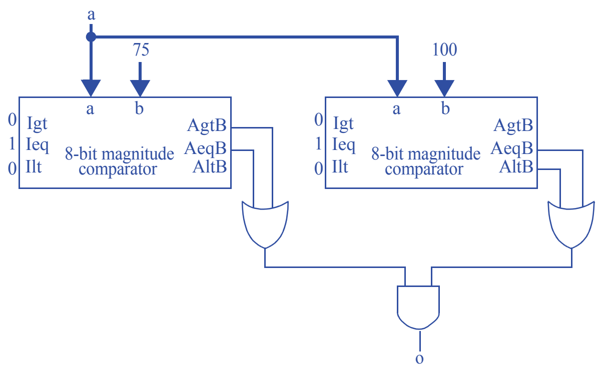
4.20 Use magnitude comparators and logic to design a circuit that computes the minimum of three 8-bit numbers. *(Component use problem)*.



4.21 Use magnitude comparators and logic to design a circuit that computes the maximum of two 16-bit numbers. *(Component use problem)*.

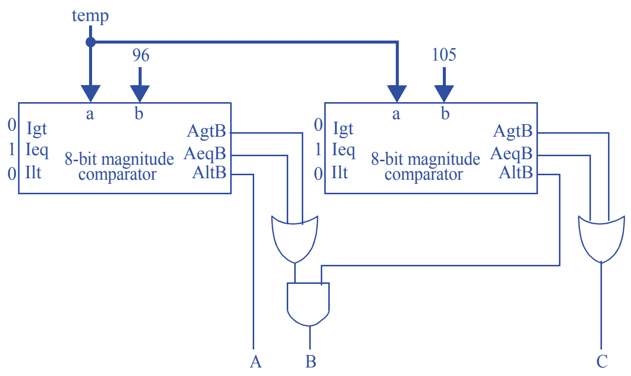


4.22 Use magnitude comparators and logic to design a circuit that outputs 1 when an 8-bit input *a* is between 75 and 100, inclusive. *(Component use problem)*.



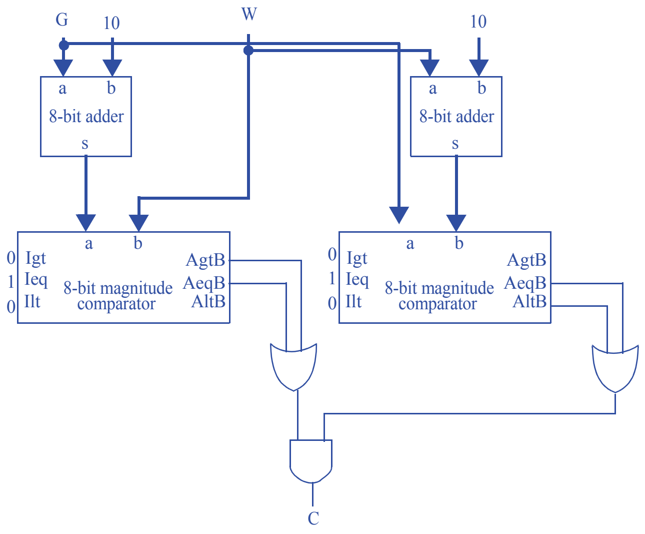
4.23 Design a human body temperature indicator system for a hospital bed. Your system takes an 8-bit input representing the temperature, which can range from 0 to 255. If the measured temperature is 95 or less, set output A to 1. If the temperature is 96 to 104, set output B to 1. If the temperature is 105 or above, set output C to 1. Use 8-bit magnitude comparators and additional logic as required. *(Component use problem)*.

A being 95 or less is the same as being less than 96. B should be 1 if the input is equal or greater than 96, AND if the input is less than 105. C is 1 if the input is equal to 105 OR if the output is greater than 105.



4.24 You are working as a weight guesser in an amusement park. Your job is to try to guess the weight of an individual before they step on a scale. If your guess is not within ten pounds of the individual's actual weight (higher or lower), the individual wins a prize. So if you guess 85 and the actual weight is 95, the person does not win; if you'd guessed 84, the person wins. Build a weight guess analyzer system that outputs whether the guess was within ten pounds. The weight guess analyzer has an 8-bit guess input G, an 8-bit input from the scale W with the correct weight, and a bit output C that is 1 if the guessed weight was within the defined limits of the game. Use 8-bit magnitude comparators and additional logic and components as required. *(Component use problem.)*

The solution checks if the guess plus 10 is greater than or equal to the actual weight, AND if guess is less than or equal to the actual weight plus 10. An alternative solution would would use a subtractor instead of the adder on the left, comparing G with W-10 rather than comparing G+10 with W.

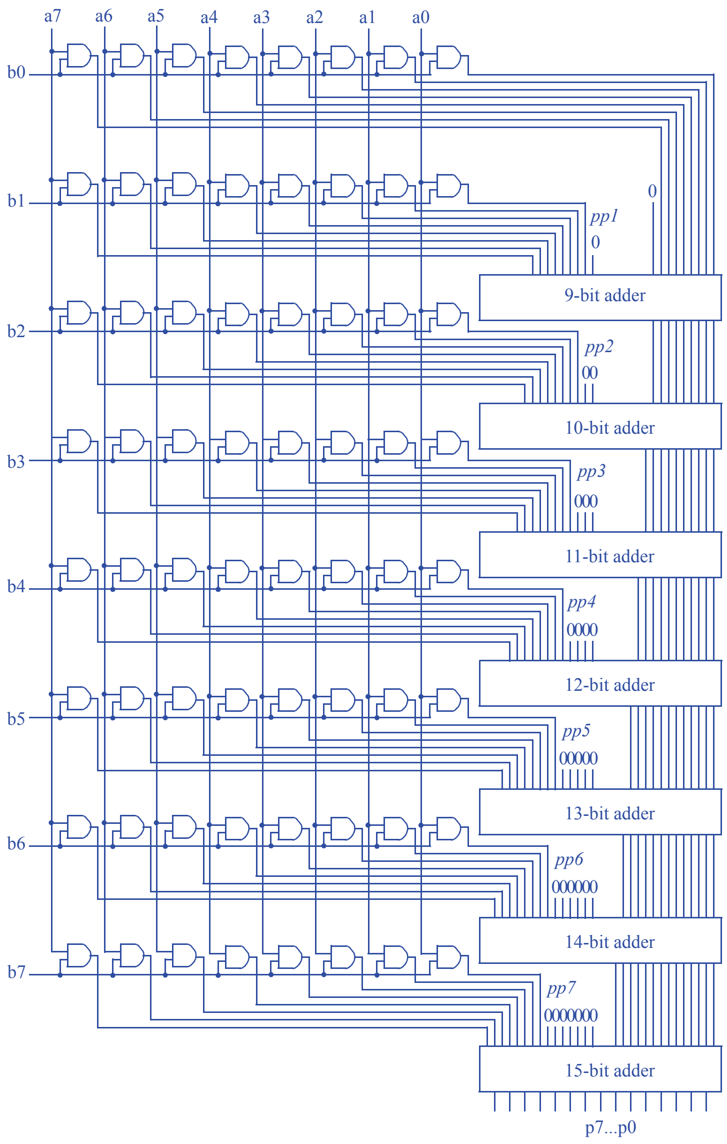


Section 4.5: Multiplier—Array Style

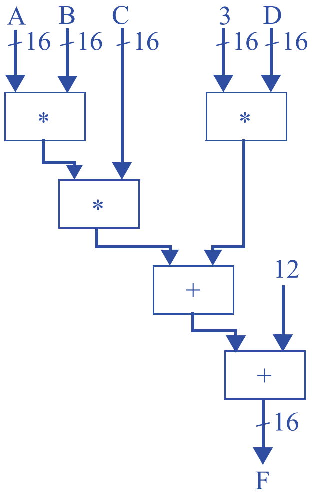
4.25 Assuming all gates have a delay of 1 time-unit, which of the following designs will compute the 8-bit multiplication A*9 faster: (a) a circuit as designed in Exercise 4.45 or (b) an 8-bit array style multiplier with one of its inputs connected to a constant value of nine.

(a) The circuit designed in Exercise 4.45 requires 16 time-units (all for the adder's computation)
(b) An 8-bit array style multplier requires 1 time-unit to compute the partial products (9 + 10 + 11 + 12 + 13 + 14 + 15) * 2 = 168 time-units to add the partial products, for a total of 169 time-units. Clearly, the circuit designed in Exercise 4.45 will compute the multiplication faster.

4.26 Design an 8-bit array-style multiplier. *(Component design problem)*.



4.27 Design a circuit to compute F = (A * B * C) + 3*D + 12. A, B, C, and D are 16-bit inputs, and F is a 16-bit output. Use 16-bit multiplier and adder components, and ignore overflow issues.



Section 4.6: Subtractors

4.28 Convert the following two's complement binary numbers to decimal numbers:

a. 00001111

b. 10000000

c. 10000001

d. 11111111

e. 10010101

a) 15
b) -128
c) -127
d) -1
e) -107

4.29 Convert the following two's complement binary numbers to decimal numbers:

a. 01001101

b. 00011010

c. 11101001

d. 10101010

e. 11111100

a) 77
b) 26
c) -23
d) -86
e) -4

4.30 Convert the following two's complement binary numbers to decimal numbers:

a. 11100000
b. 01111111
c. 11110000
d. 11000000
e. 11100000

a) -32
b) 127
c) -16
d) -64
e) -32

4.31 Convert the following 9-bit two's complement binary numbers to decimal numbers:

a. 011111111
b. 111111111
c. 100000000
d. 110000000
e. 111111110

a) 255
b) -1
c) -256
d) -128
e) -2

4.32 Convert the following decimal numbers to 8-bit two's complement binary form:

a. 2
b. -1
c. -23
d. -128
e. 126
f. 127
g. 0

a) 00000010
b) 1111111
c) 11101001
d) 10000000
e) 01111110
f) 01111111
g) 00000000

4.33 Convert the following decimal numbers to 8-bit two's complement binary form:

a. 29
b. 100
c. 125
d. -29
e. -100
f. -125
g. -2

a) 00011101
b) 01100100
c) 01111101
d) 11100011
e) 10011100
f) 10000011
g) 11111110

4.34 Convert the following decimal numbers to 8-bit two's complement binary form:

a. 6
b. 26
c. -8
d. -30
e. -60
f. -90

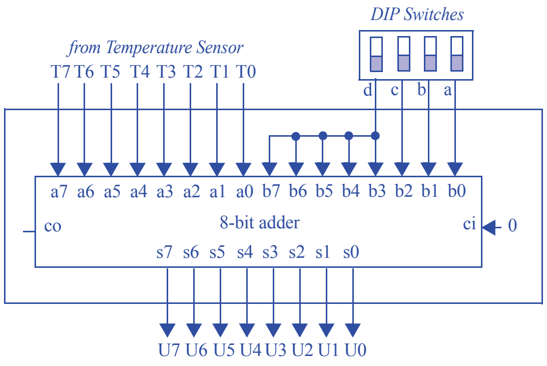
a) 00000110
b) 00011010
c) 11111000
d) 11100010
e) 11000100
f) 10100110

4.35 Convert the following decimal numbers to 9-bit two's complement binary form:

a. 1
b. -1
c. -256
d. -255
e. 255
f. -8
g. -128

a) 000000001
b) 111111111
c) 100000000
d) 100000001
e) 011111111
f) 111111000

4.36 Repeat Exercise 4.14, except that the compensation amount can be positive or negative, coming to the system via four inputs d, c, b, and a from a 4-pin DIP switch (d is the most significant bit). The compensation amount is in two's complement form (so the person setting the DIP switch must know that). Design the circuit. What is the range by which the input temperature can be compensated? *(Component use problem)*.

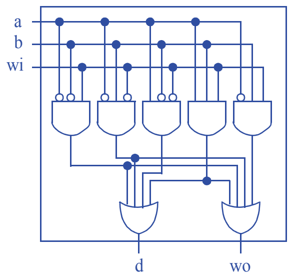


The 4-bit input must be extended to the 8-bit input of the adder. If the high-order bit d of the 4-bit input is 0, then b7-b3 should all be 0. If the high-order bit d is 1, then b7-b3 should all be 1. The temperature can be compensated from -8 to +7 degrees.

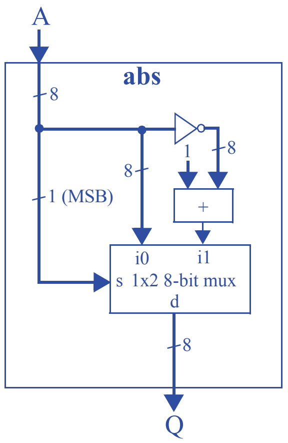
4.37 Create the internal design of a full-subtractor. *(Component design problem)*.

Inputs			Outputs	
a	b	wi	d	wo
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

d = a’b’wi + a’bwi’ + ab’wi’ + abwi
wo = a’b’wi + a’bwi’ + a’bwi + abwi

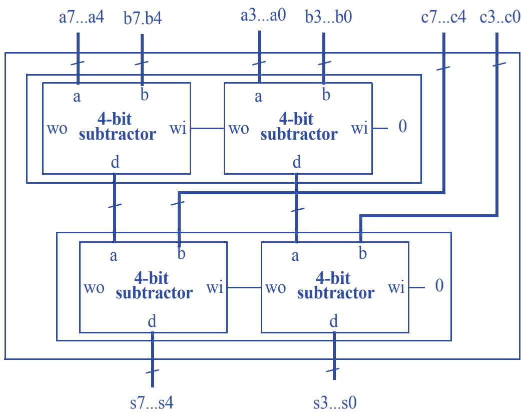


4.38 Create an absolute value component abs with an 8-bit input A that is a signed binary number, and an 8-bit output Q that is unsigned and that is the absolute value of A. So if the input is 00001111 (+15) then the output is also 00001111 (+15), but if the input is 11111111 (-1) then the output is 00000001 (+1).



4.39 Using 4-bit subtractors, build a circuit that has three 8-bit inputs, A, B, and C, and a single 8-bit output F, where F=(A-B)-C. *(Component use problem.)*

First compose the 4-bit subtractors into an 8-bit subtractor, then use 8-bit subtractors in the design.

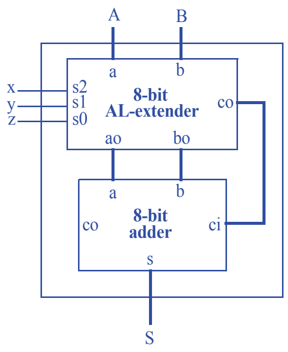


Section 4.7: Arithmetic-Logic Units—ALUs

4.40 Design an ALU with two 8-bit inputs A and B, and control inputs x, y, and z. The ALU should support the operations described in Table 4.3. Use an 8-bit adder and an arithmetic/logic extender. *(Component design problem)*.

Table 4.3

Inputs			Operation
x	y	z	
0	0	0	S = A - B
0	0	1	S = A + B
0	1	0	S = A * 8
0	1	1	S = A / 8
1	0	0	S = A NAND B (bitwise NAND)
1	0	1	S = A XOR B (bitwise XOR)
1	1	0	S = Reverse A (bit reversal)
1	1	1	S = NOT A (bitwise complement)

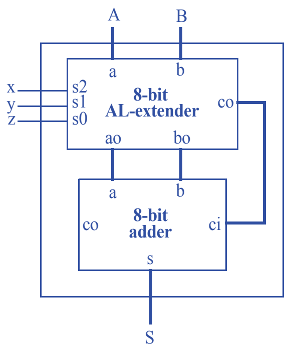


Operation of the AL-extender:
When xyz=000, ao=a, bo=b’, co=1
When xyz=001, ao=a, bo=b, co=0
When xyz=010, ao=a<<3, bo=0, co=0
When xyz=011, ao=a>>3, bo=0, co=0
When xyz=100, ao=a NAND b, bo=0, co=0
When xyz=101, ao=a XOR b, bo=0, co=0
When xyz=111, ao=a reversed, bo=0, co=0
When xyz=111, ao=NOT a, bo=0, co=0

4.41 Design an ALU with two 8-bit inputs A and B, and control signals x, y, and z. The ALU should support the operations described in Table 4.4. Use an 8-bit adder and an arithmetic/logic extender. *(Component design problem)*.

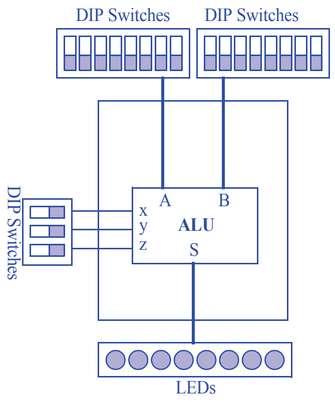
Table 4.4

Inputs			**Operation**
x	y	z	
0	0	0	S = A + B
0	0	1	S = A AND B (bitwise AND)
0	1	0	S = A NAND B (bitwise NAND)
0	1	1	S = A OR B (bitwise OR)
1	0	0	S = A NOR B (bitwise NOR)
1	0	1	S = A XOR B (bitwise XOR)
1	1	0	S = A XNOR B (bitwise XNOR)
1	1	1	S = NOT A (bitwise complement)



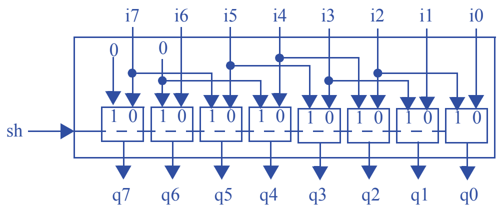
Operation of the AL-extender:
When xyz=000, ao=a, bo=b', co=1
When xyz=001, ao=a AND b, bo=0, co=0
When xyz=010, ao=a NAND b, bo=0, co=0
When xyz=011, ao=a OR b, bo=0, co=0
When xyz=100, ao=a NOR b, bo=0, co=0
When xyz=101, ao=a XOR b, bo=0, co=0
When xyz=110, ao=a XNOR b, bo=0, co=0
When xyz=111, ao=NOT a, bo=0, co=0

4.42 An instructor teaching Boolean algebra wants to help her students learn and understand basic Boolean operators by providing the students with a calculator capable of performing bitwise AND, NAND, OR, NOR, XOR, XNOR, and NOT operations. Using the ALU specified in Exercise 4.41, build a simple logic calculator using DIP-switches for input and LEDs for output. The logic calculator should have three DIP-switch inputs to select which logic operation to perform. *(Component use problem)*.



Section 4.8: Shifters

4.43 Design an 8-bit shifter that shifts its inputs two bits to the right (shifting in 0s) when the shifter's shift control input is 1 (*Component design problem*).



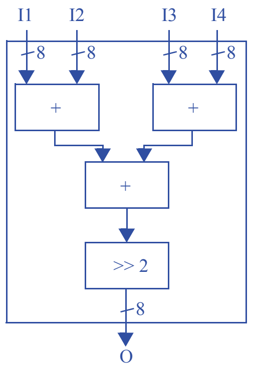
4.44 Design a circuit that outputs the average of four 8-bit inputs representing unsigned binary numbers:

a. Ignoring overflow issues.

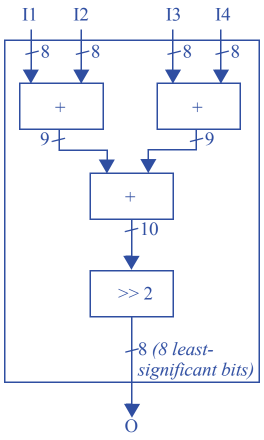
b. Using wider internal components or wires to avoid losing information due to overflow.

(Component use problem.).

a.)

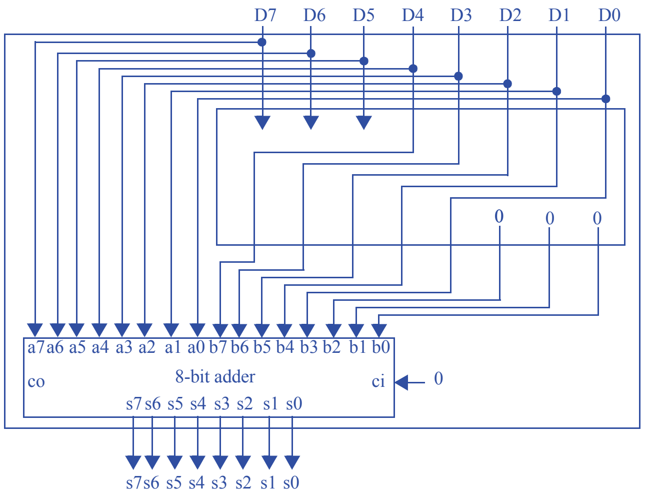


b.)We can use the same circuit from a), but now we prefix the output bus of each adder with the carry-out bit of that adder, thus adding one bit of precision at each level of additions..


I1
I2
I3
I4
8
8
8
8
+
+
9
9
+
10
>> 2
8 (8 least-significant bits)
O

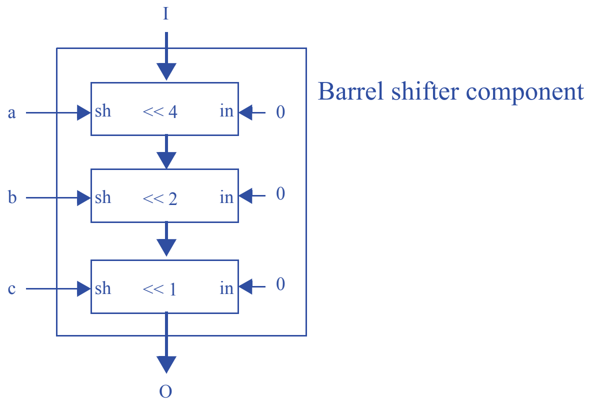
4.45 Design a circuit whose 16-bit output is nine times its 16-bit input D representing an unsigned binary number. Ignore overflow issues. (Component use problem.)

Use a left shift by 3 to obtain 8D, then add D to the result to obtain 8D+D=9D.



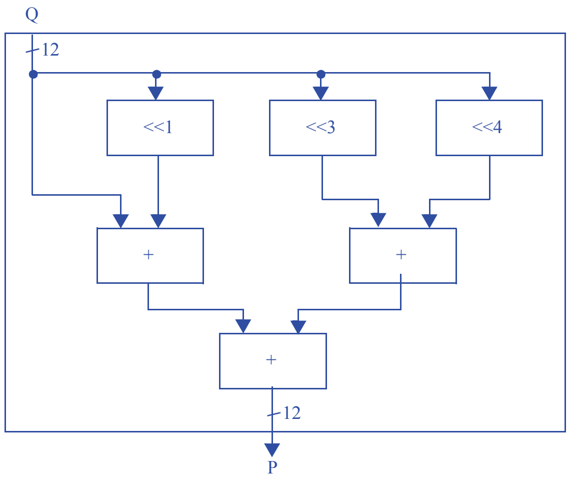
4.46 Design a special multiplier circuit that can multiply its 16-bit input by 1, 2, 4, 8, or 16, or 32, specified by three inputs a, b, c (abc=000 means no multiply, abc=001 means multiply by 2, abc=010 means by 4, abc=011 means by 8, abc=100 means by 16, abc=101 means by 32). *Hint*: A simple solution consists entirely of just one copy of a component from this chapter. *(Component use problem)*.

The solution just uses a single barrell shifter component. The internals of such a component are shown below for convenience.



4.47 Use strength reduction to create a circuit that computes P = 27*Q using only shifts and adds. P is a 12-bit output and Q is a 12-bit input. Estimate the transistors in the circuit and compare to the estimated transistors in a circuit using a multiplier.

We can implement 27*Q as (16 + 8 + 2 + 1)*Q = (Q*16 + Q*8 + Q*2 + Q), which could be accomplished using only shifts and adds as (Q<<4 + Q<<3 + Q<<1 + Q):



Since each shifter can be implemented with only wires, each shifter uses 0 transistors. We have 3 12-bit adders, which means 3*12 = 36 full-adders. If each full-adder requires approximately 12 transistors, this means 12*36 = 432 transistors in the shift-and-add implementation.

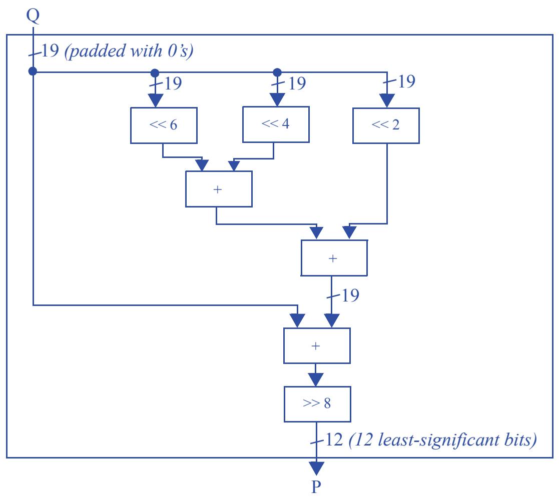
Since the smallest power of two which is greater than or equal to 27 is 32, the smallest multiplier we could use is a 12x5 multiplier. Assuming the multiplier is an array-style multiplier, this means 12*5 = 60 AND gates, a 13-bit adder, a 14-bit adder, a 15-bit adder, and a 16-bit adder. Each AND gate is ~6 transistors, so we have 360 transistors from the AND gates alone. The 13-bit adder has (13 * 12) = 156 transistors, the 14-bit adder (14 * 12) = 168 transistors, the 15-bit adder (15 * 12) = 180 transistors, and the 16-bit adder (16 * 12) = 192 transistors. In total, the multiplier would consist of (360 + 156 + 168 + 180 + 192) = 1052 transistors.

It’s easy to see how the use of strength reduction can drastically reduce the number of transistors required.

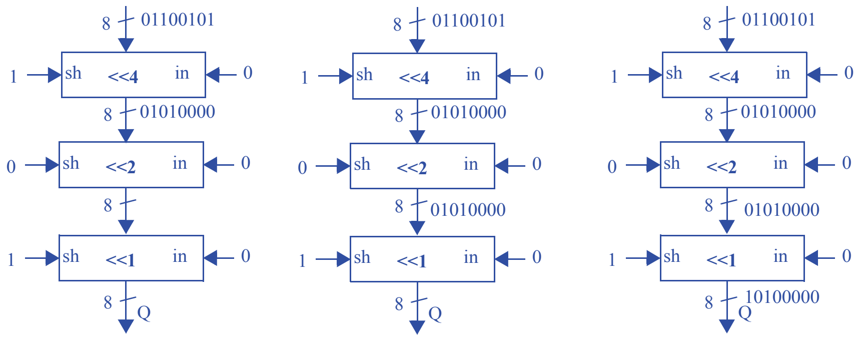
4.48 Use strength reduction to create a circuit that approximately computes P = (1/3)*Q using only shifters and adders. Strive for accuracy to the hundredths place (0.33). P is a 12-bit output and Q is a 12-bit input. Use wider internal components and wires as necessary to prevent internal overflow.

Our goal here is essentially to find a fraction whose denominator is a power of two and whose value approximates 1/3 to the hundredths place. For instance, we might choose the approximation 85/256, whose value is ~0.332.

The multiplication could thus be approximated by Q*(64 + 16 + 4 + 1) / 256 = (Q*64 + Q*16 + Q*4 + Q) / 256, which could be accomplished using only shifters and adders as (Q<<6 + Q<<4 + Q<<2 + Q)>>8:



4.49 Show the internal values of the barrel shifter of Figure 4.64, when I=01100101, x = 1, y = 0, and z = 1. Be sure to show how the input I is shifted after each internal shifter stage. (Component design problem).

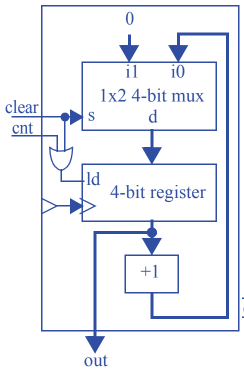


4.50 Using the barrel shifter shown in Figure 4.42, what settings of the inputs x, y, and z are required to shift the input I left by six positions.

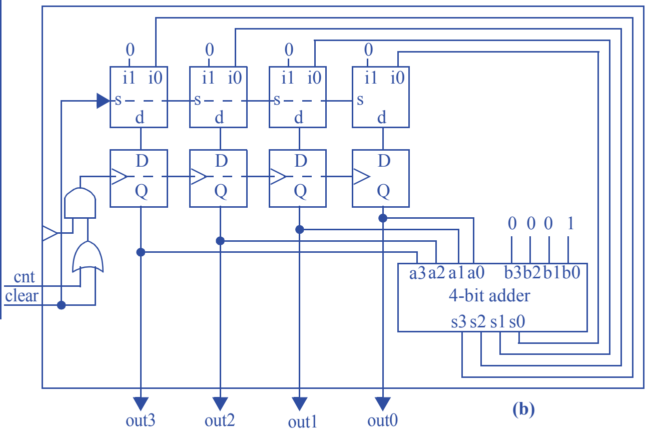
x = 1, y = 1, z = 0

Section 4.9: Counters

4.51 Design a 4-bit up-counter that has two control inputs: *cnt* enables counting up, while *clear* synchronously resets the counter to all 0s, (a) using a parallel load register as a building block, (b) using flip-flops and muxes directly by following the register design process of Section 4.2. *(Component design problem)*.

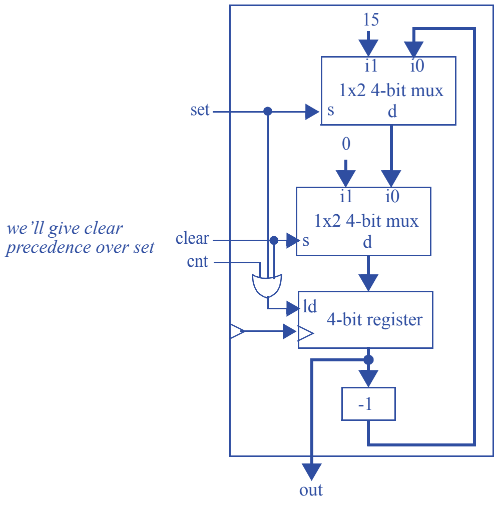


(a)

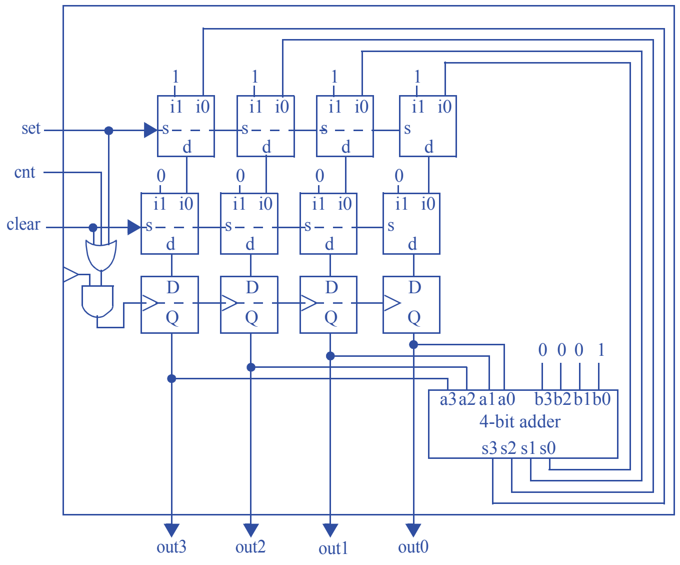


(b)

4.52 Design a 4-bit down-counter that has three control inputs: *cnt* enables counting up, *clear* synchronously resets the counter to all 0s, and *set* synchronously sets the counter to all `1`s, (a) using a parallel load register as a building block, (b) using flip-flops and muxes directly by following the register design process of Section 4.2. *(Component design problem)*.



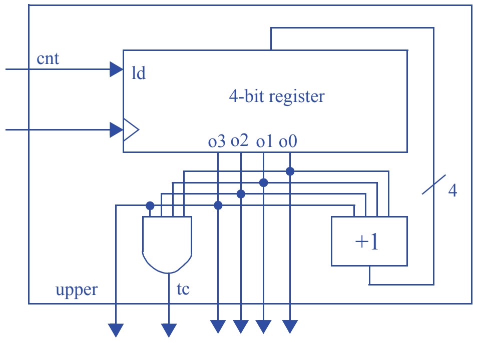
(a)



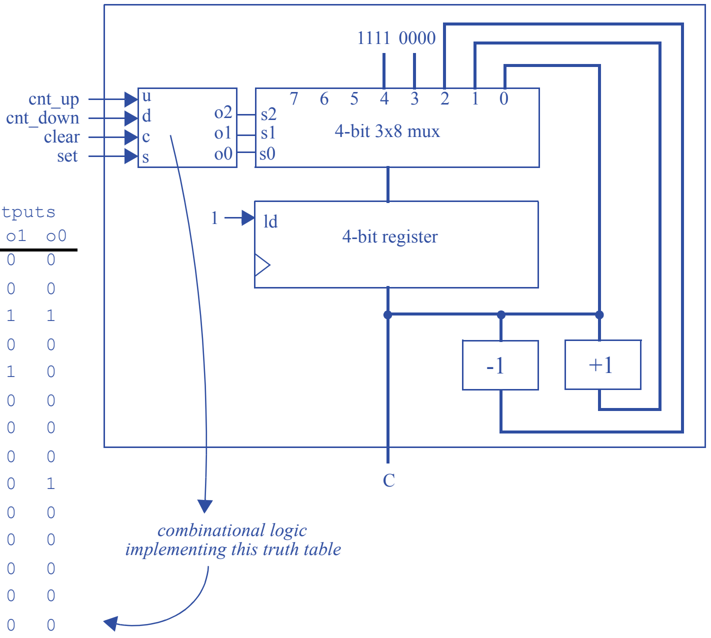
(b)

4.53 Design a 4-bit up-counter with an additional output *upper*. *upper* outputs a 1 whenever the counter is within the upper half of the counter’s range, 8 to 15. Use a basic 4-bit up-counter as a building block. *(Component design problem)*

Upper is obtained simply from the 4th bit of the counter, which will be 1 for values 8 to 15. The internals of the up-counter are shown below for convenience.

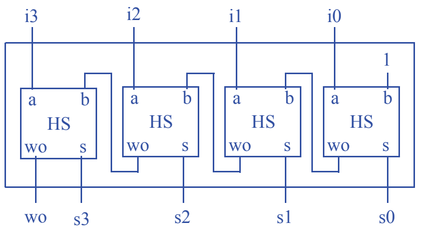


4.54 Design a 4-bit up/down-counter that has four control inputs: cnt_up enables counting up, cnt_down enables counting down, clear synchronously resets the counter to all 0s, and set synchronously sets the counter to all 1s. If two or more control inputs are 1, the counter retains its current count value. Use a parallel load register as a building block. (Component design problem.)



Inputs				Outputs		
u	d	c	s	o2	o1	o0
0	0	0	0	0	0	0
0	0	0	1	1	0	0
0	0	1	0	0	1	1
0	0	1	1	0	0	0
0	1	0	0	0	1	0
0	1	0	1	0	0	0
0	1	1	0	0	0	0
0	1	1	1	0	0	0
1	0	0	0	0	0	1
1	0	0	1	0	0	0
1	0	1	0	0	0	0
1	0	1	1	0	0	0
1	1	0	0	0	0	0
1	1	0	1	0	0	0
1	1	1	0	0	0	0
1	1	1	1	0	0	0

4.55 Design a circuit for a 4-bit decrementer. *(Component design problem)*.



4.56 Assume an electronic turnstile internally uses a 64-bit counter that counts up once for each person that passes through the turnstile. Knowing that California's Disneyland park attracts about 15,000 visitors per day, and assuming they all pass that one turnstile, how many days would pass before the counter would roll over? (Component use problem.)

2^{64}/15000 = 1,229,782,938,247,303 days. That's a long time.

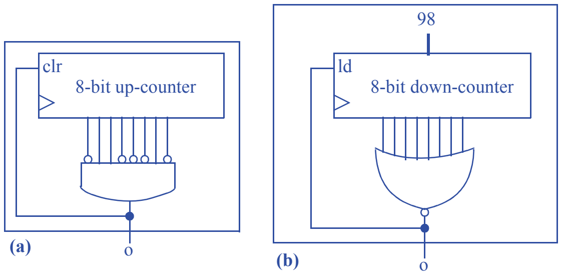
4.57 Design a circuit that outputs a 1 every 99 clock cycles:

a. Using an up-counter with a synchronous clear control input, and using extra logic,

b. Using a down-counter with parallel load, and using extra logic.

c. What are the tradeoffs between the two designs from parts (a) and (b)?

(Component use problem.)



(c) The circuit implemented in (a) is smaller, while the circuit implemented in (b) is easier to modify to pulse at a different rate.

4.58 Give the count range for the following sized up-counters:

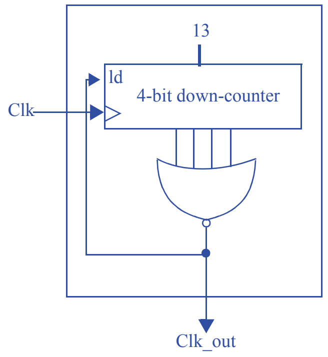
a. 8-bits, 12-bits, 16-bits, 20-bits, 32-bits, 40-bits, 64-bits, and 128-bits.

b. For each size of counter in part (a), assuming a 1 Hz clock, indicate how much time would pass before the counter wraps around; use the most appropriate units for each answer (seconds, minutes, hours, days, weeks, months, or years).

(Component use problem.)

8 bits: 0-255 (4 mins, 16 secs)
12 bits: 0-4,095 (1 hour, 8 mins, 16 secs)
16 bits: 0-65,535 (18 hours, 12 mins, 16 secs)
20 bits: 0-1,048,575 (12 days, 3 hours, 16 mins, 16 secs)
32 bits: 0-4,294,967,295 (136 years, 70 days, 6 hours, 28 mins, 16 secs)
40 bits: 0-1,099,511,627,775 (34,865 years, 104 days, 36 mins, 16 secs)
64 bits: 0-1.845E19 (5.849E11 years)
128 bits: 0-3.403E38 (1.079E31 years)
(For comparison, the universe is approximately 14 billion or 14E9 years old)

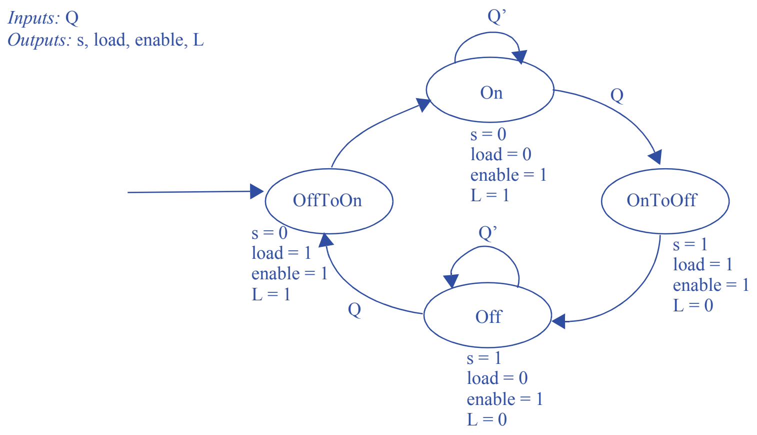
4.59 Create a clock divider that converts a 14 MHz clock into a 1 MHz clock. Use a down-counter with parallel load. Clearly indicate the width of the down counter and the counter's load value. (Component use problem.)



Note that this is technically a pulse generator, but it still divides the clock by 14. If a 50% duty cycle is required, we can change the down-counter load value to 6, add a register whose ld signal is Clk_out and whose input is a 1x2 mux, where i0 is 1, i1 is 0, and the select line is the output of the register. The output of the register would then also be the divided clock signal.

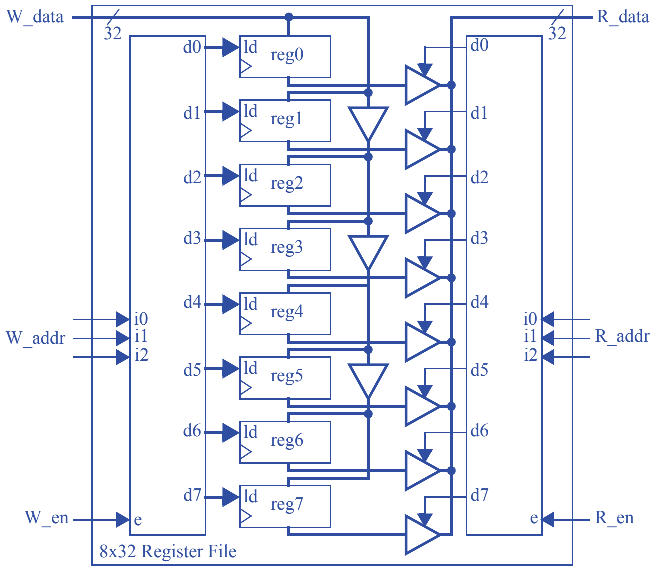
4.60 Assuming a 32-bit microsecond timer is available to a controller and a controller clock frequency of 100 MHz, create a controller FSM that blinks an LED by setting an output L to 1 for 5 ms and then to 0 for 13 ms, and then repeats. Use the timer to achieve the desired timing (i.e., do not use a clock divider). For this example, the blinking rate can vary by a few clock cycles. (Component use problem.)

Assuming the timer's input is connected to a 1x2 32-bit mux whose i0 is 5000 and whose i1 is 13000, the mux's select line is called 's', one possible FSM would be:

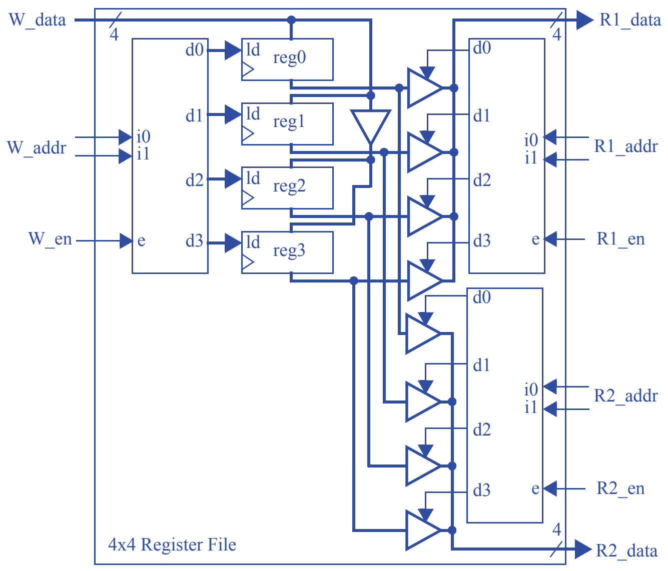


Section 4.10: Register Files

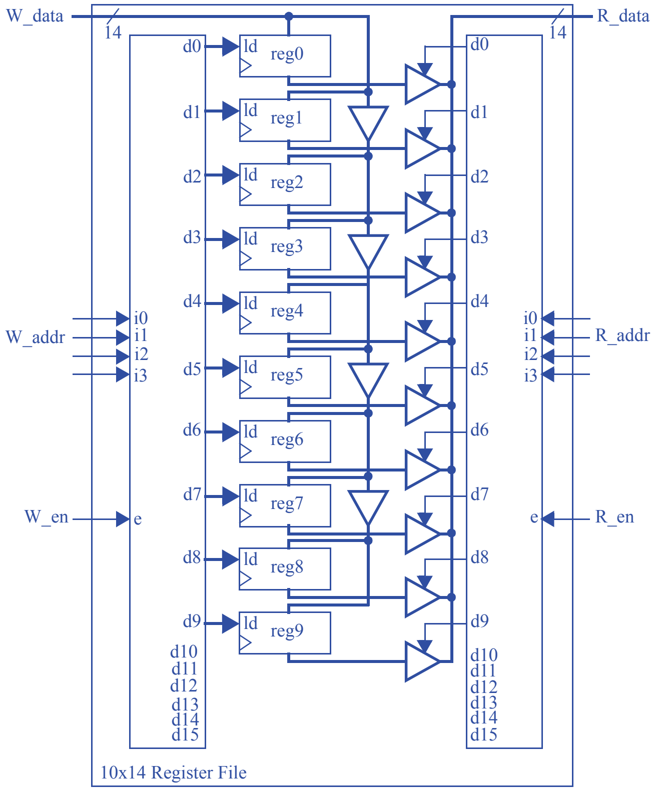
4.61 Design an 8x32 two port (1 read, 1 write) register file. *(Component design problem)*.



4.62 Design a 4x4 three port (2 read, 1 write) register file. *(Component design problem)*.



4.63 Design a 10x14 register file (one read port, one write port). *(Component design problem)*.



4.64 A 4x4 register file's four registers initially each contain 0101.

a. Show the input values necessary to read register 3 and to simultaneously write register 3 with the value 1110.

b. With these values, show the register file's register values and output values before the next rising clock edge, and after the next rising clock edge.

a.)W_data = 1110, W_addr = 11, W_en = 1, R_addr = 11, R_en = 1.

b.) Before rising edge:

R0 = 0101

R1 = 0101

R2 = 0101

R3 = 0101

R_data = 0101

After rising edge:

R0 = 0101

R1 = 0101

R2 = 0101

R3 = 1110

R_data = 1110