XCbb: Combinational Building-Block Examples

In earier chapters of this book, we looked at basic principles in several areas—number systems, digital circuits, and combinational logic—and we described many of the basic building blocks of combinational design—decoders, multiplexers, and the like. All of that is a needed foundation, but the ultimate goal of studying digital design is eventually to be able to solve real problems by designing digital systems (well, duh…). That usually requires experience beyond what you can get by reading a textbook. We'll try to get you started by presenting a number of larger combinational design examples in this and other example sections.

This section gives design examples using combinational building blocks. While the examples are written in terms of MSI functions, the same functions are widely used in ASIC and schematic-based CPLD and FPGA design. The idea of these examples is to show that you can often express a combinational function using a collection of smaller building blocks. This is important for a couple of reasons: a hierarchical approach usually simplifies the overall design task, and the smaller building blocks often have a more efficient, optimized realization in FPGA and ASIC cells than what you'd get if you wrote a larger, monolithic description in an HDL and then just hit the "synthesize" button.

XCbb.1 Barrel Shifter

barrel shifter

A *barrel shifter* is a combinational logic circuit with n data inputs, n data outputs, and a set of control inputs that specify how to shift the data between input and output. A barrel shifter that is part of a microprocessor CPU can typically specify the direction of shift (left or right), the type of shift (circular, arithmetic, or logical), and the amount of shift (typically 0 to $n-1$ bits, but sometimes 1 to n bits).

In this subsection we'll look at the design of a simple 16-bit barrel shifter that does left circular shifts only, using a 4-bit control input S[3:0] to specify the amount of shift. For example, if the input word is ABCDEFGHIJKLMNOP (where each letter represents one bit), and the control input is 0101 (5), then the output word is FGHIJKLMNOPABCDE.

From one point of view, this problem is deceptively simple. Each output bit can be obtained from a 16-input multiplexer controlled by the shift-control inputs, where each multiplexer data input is connected to the appropriate data bit. On the other hand, when you look at the details of the design, you'll see that there are trade-offs in the speed and size of the circuit.

Let us first consider a design that uses off-the-shelf MSI multiplexers. A 16-input, one-bit multiplexer can be built using two 74x151s by applying S3 and its complement to the EN_L inputs and combining the Y_L data outputs with a NAND gate, as we showed in Figure 6-62 for a 32-input multiplexer. The

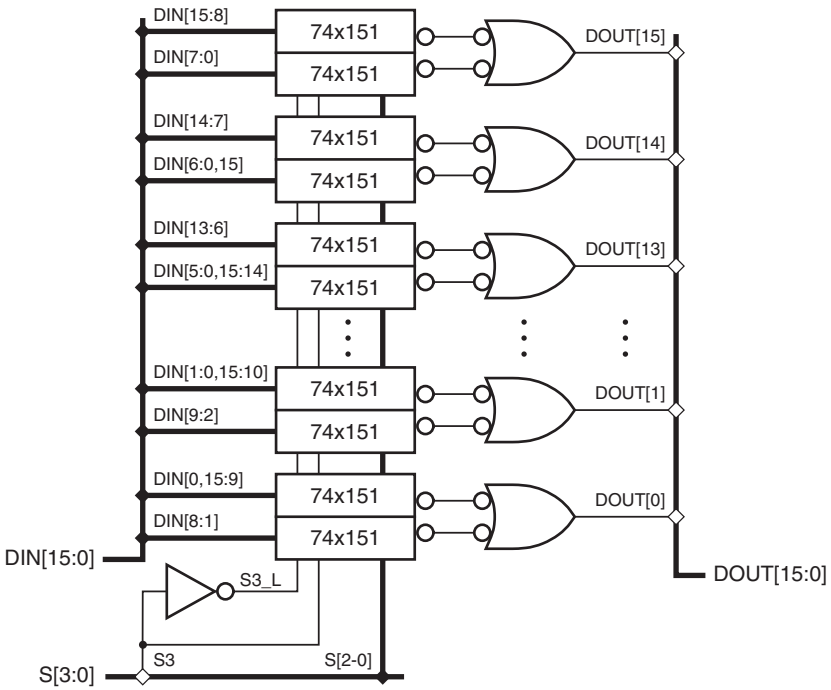


Figure XCbb-1 One approach to building a 16-bit barrel shifter.

low-order shift-control inputs, S2–S0, connect to the like-named select inputs of the ’151s.

We complete the design by replicating this 16-input multiplexer 16 times and hooking up the data inputs appropriately, as shown in Figure XCbb-1. The active-low enable input of the top ’151 of each pair should be connected to S3, and the bottom one to S3_L; the remaining select bits are connected to all 32 ’151s. Data inputs D0–D7 of each ’151 are connected to the DIN inputs in the listed order from left to right.

The first row of Table XCbb-1 shows the characteristics of this first approach. About 36 chips (32 74x151s, 4 74x00s, and 1/6 74x04) are used in the MSI/SSI realization. We can reduce this to 32 chips by replacing the 74x151s with 74x251s and tying their three-state Y outputs together, as tabulated in the second row. Both of these designs have very heavy loading on the control inputs; each of the control bits S[2:0] must be connected to the like-named select input of all 32 multiplexers. The data inputs are also fairly heavily loaded; each data bit must connect to 16 different multiplexer data inputs, corresponding to the 16 possible shift amounts. However, assuming that the heavy control and data loads don’t slow things down too much, the 74x251-based approach yields the shortest data delay, with each data bit passing through just one multiplexer.

Alternatively, we could build the barrel shifter using 16 74x157 2-input, 4-bit multiplexers, as tabulated in the last row of the table. We start by using four

Multiplexer Component	Data Loading	Data Delay	Control Loading	Total ICs
74x151 (8-in, 1-bit)	16	2	32	36
74x251 (8-in, 1-bit, 3-state)	16	1	32	32
74x153 (4-in, 2-bit)	4	2	8	16
74x157 (2-in, 4-bit)	2	4	4	16

Table XCbb-1 Properties of four different barrel-shifter design approaches.

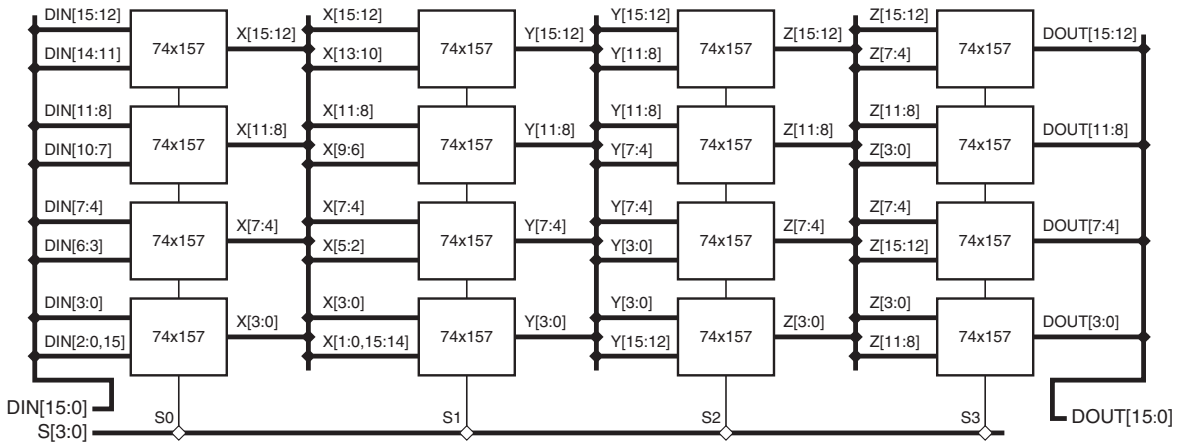
74x157s to make a 2-input, 16-bit multiplexer. Then, we can hook up a first set of four ’157s controlled by S0 to shift the input word left by 0 or 1 bit. The data outputs of this set are connected to the inputs of a second set, controlled by S1, which shifts its input word left by 0 or 2 bits. Continuing the cascade, a third and fourth set are controlled by S2 and S3 to shift selectively by 4 and 8 bits, as shown in Figure XCbb-2. Here, the 1A–4A and 1B–4B inputs and the 1Y–4Y outputs of each ’157 are connected to the indicated signals in the listed order from left to right.

The ’157-based approach requires only half as many MSI packages and has far less loading on the control and data inputs. On the other hand, it has the longest data-path delay, since each data bit must pass through four 74x157s.

Halfway between the two approaches, we can use eight 74x153 4-input, 2-bit multiplexers to build a 4-input, 16-bit multiplexer. Cascading two sets of these, we can use S[3:2] to shift selectively by 0, 4, 8, or 12 bits, and S[1:0] to shift by 0–3 bits. This approach has the performance characteristics shown in the third row of Table XCbb-1 and would appear to be the best compromise if you don’t need to have the absolutely shortest possible data delay.

The same kind of considerations would apply if you were building the barrel shifter out of ASIC cells instead of MSI parts, except you’d be counting

Figure XCbb-2 A second approach to building a 16-bit barrel shifter.



chip area instead of MSI/SSI packages. Typical ASIC cell libraries have 1-bit-wide multiplexers, usually realized with CMOS transmission gates, with two to eight inputs. To build a larger multiplexer, you have to put together the appropriate combination of smaller cells.

Besides the kind of choices we encountered in the MSI example, you have the further complication that CMOS delays are highly dependent on loading. Thus, depending on the approach, you must decide where to add buffers to the control lines, the data lines, or both to minimize loading-related delays. An approach that looks good on paper, before analyzing these delays and adding buffers, may actually turn out to have poorer delay or more chip area than another approach. To find the optimal approach in a particular ASIC technology, you'll typically have to design and analyze the performance of all of them.

XCbb.2 Simple Floating-Point Encoder

The previous example used multiple copies of a single building block, a multiplexer, and it was pretty obvious from the problem statement that a multiplexer was the appropriate building block. The next example, a "fixed-point to floating-point encoder," shows that you sometimes have to look a little harder to see the solution in terms of known building blocks.

An unsigned binary integer B in the range $0 \le B < 2^{11}$ can be represented by 11 bits in "fixed-point" format, $B = b_{10}b_9 \ldots b_1b_0$. We can represent numbers in the same range with less precision using only 7 bits in a floating-point notation, $F = M \cdot 2^E$, where M is a 4-bit mantissa $m_3m_2m_1m_0$ and E is a 3-bit exponent $e_2e_1e_0$. The smallest integer in this format is $0 \cdot 2^0$ and the largest is $(2^4-1) \cdot 2^7$.

Given an 11-bit fixed-point integer B, we can convert it to our 7-bit floating-point notation by "picking off" four high-order bits beginning with the most significant 1, for example,

$$\begin{aligned}
11010110100 &= 1101 \cdot 2^7 + 0110100 \\
00100101111 &= 1001 \cdot 2^5 + 01111 \\
00000111110 &= 1111 \cdot 2^2 + 10 \\
00000001011 &= 1011 \cdot 2^0 + 0 \\
00000000010 &= 0010 \cdot 2^0 + 0
\end{aligned}$$

The last term in each equation is a truncation error that results from the loss of precision in the conversion. Corresponding to this conversion operation, we can write the specification for a fixed-point to floating-point encoder circuit:

- A combinational circuit is to convert an 11-bit unsigned binary integer B into a 7-bit floating-point number M,E, where M and E have 4 and 3 bits, respectively. The numbers have the relationship $B = M \cdot 2^E + T$, where T is the truncation error, $0 \le T < 2^E$.

Starting with a problem statement like the one above, it takes some creativity to come up with an efficient circuit design—the specification gives no clue. However, we can get some ideas by looking at how we converted numbers by hand earlier. We basically scanned each input number from left to right to find the first position containing a 1, stopping at the b_3 position if no 1 was found. We picked off four bits starting at that position to use as the mantissa, and the starting position number determined the exponent. These operations are beginning to sound like MSI building blocks.

"Scanning for the first 1" is what a generic priority encoder does. The output of the priority encoder is a number that tells us the position of the first 1. The position number determines the exponent; first-1 positions of $b_{10} - b_3$ imply exponents of 7–0, and positions of $b_2 - b_0$ or no-1-found imply an exponent of 0. Therefore, we can scan for the first 1 with an 8-input priority encoder with inputs I7 (highest priority) through I0 connected to $b_{10} - b_3$. We can use the priority encoder's A2–A0 outputs directly as the exponent, as long as the no-1-found case produces A2–A0 = 000.

"Picking off four bits" sounds like a "selecting" or multiplexing operation. The 3-bit exponent determines which four bits of B we pick off, so we can use the exponent bits to control an 8-input, 4-bit multiplexer that selects the appropriate four bits of B to form M.

An MSI circuit that results from these ideas is shown in Figure XCbb-3 on the next page. It contains several optimizations:

- Since the available MSI priority encoder, the 74x148, has active-low inputs, the input number B is assumed to be available on an active-low bus B_L[10:0]. If only an active-high version of B is available, then eight inverters can be used to obtain the active-low version.
- If you think about the conversion operation a while, you'll realize that the most significant bit of the mantissa, m_3, is always 1, except in the no-1-found case. The '148 has a GS_L output that indicates this case, allowing us to eliminate the multiplexer for m_3.
- The '148 has active-low outputs, so the exponent bits (E0_L–E2_L) are produced in active-low form. Naturally, three inverters could be used to produce an active-high version.
- Since everything else is active-low, active-low mantissa bits are used, too. Active-high bits are also readily available on the '148 EO_L and the '151 Y_L outputs.

Strictly speaking, the multiplexers in Figure XCbb-3 are drawn incorrectly. The 74x151 symbol can be drawn alternatively as shown in Figure XCbb-4. In words, if the multiplexer's data inputs are active low, then the data outputs have an active level opposite that shown in the original symbol. The "active-low-data" symbol should be preferred in Figure XCbb-3, since the active levels of the

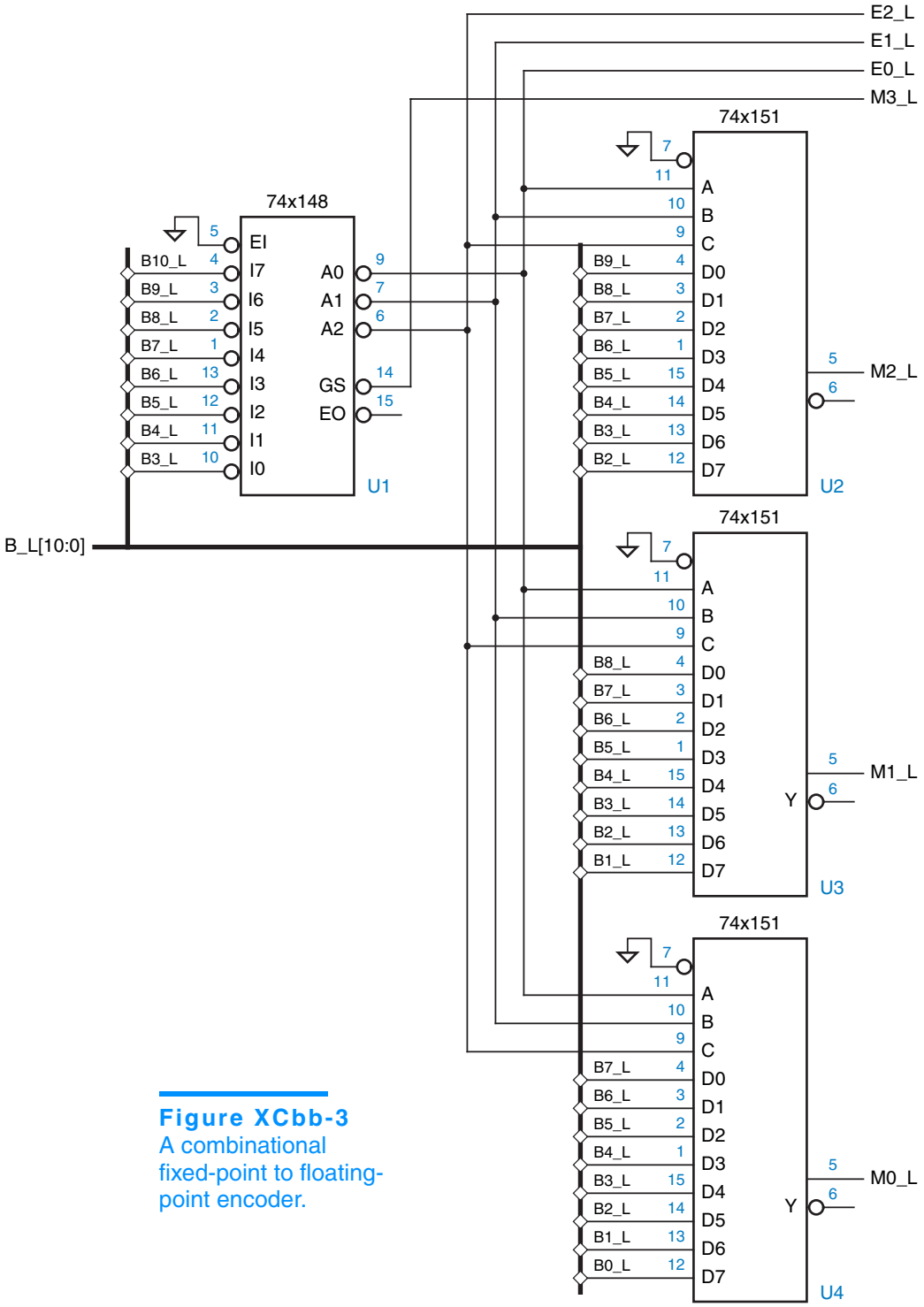


Figure XCbb-3
A combinational fixed-point to floating-point encoder.

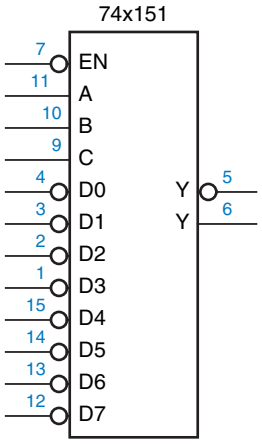


Figure XCbb-4
Alternate logic symbol for the 74x151 8-input multiplexer.

'151 inputs and outputs would then match their signal names. However, in data transfer and storage applications, designers (and the book) don't always "go by the book." It is usually clear from the context that a multiplexer (or a multibit register, in Section 8.2.5) does not alter the active level of its data.

XCbb.3 Dual-Priority Encoder

Quite often MSI building blocks need a little help from their friends—ordinary gates—to get the job done. In this example, we'd like to build a priority encoder that identifies not only the highest but also the second-highest-priority asserted signal among a set of eight request inputs.

We'll assume for this example that the request inputs are active low and are named R_L[0:7], where R_L0 has the highest priority. We'll use A[2:0] and AVALID to identify the highest-priority request, where AVALID is asserted only if at least one request input is asserted. We'll use B[2:0] and BVALID to identify the second-highest-priority request, where BVALID is asserted only if at least two request inputs are asserted.

Finding the highest-priority request is easy enough, we can just use a 74x148. To find the second-highest-priority request, we can use another '148, but only if we first "knock out" the highest-priority request before applying the request inputs. This can be done using a decoder to select a signal to knock out, based on A[2:0] and AVALID from the first '148. These ideas are combined in the solution shown in Figure XCbb-5. A 74x138 decoder asserts at most one of its eight outputs, corresponding to the highest-priority request input. The outputs are fed to a rank of NAND gates to "turn off" the highest-priority request.

A trick is used in this solution to get active-high outputs from the '148s, as shown in Figure XCbb-6. We can rename the address outputs A_L[2:0] to be active high if we also change the name of the request input that is associated with each output combination. In particular, we complement the bits of the request number. In the redrawn symbol, request input I0 has the highest priority.

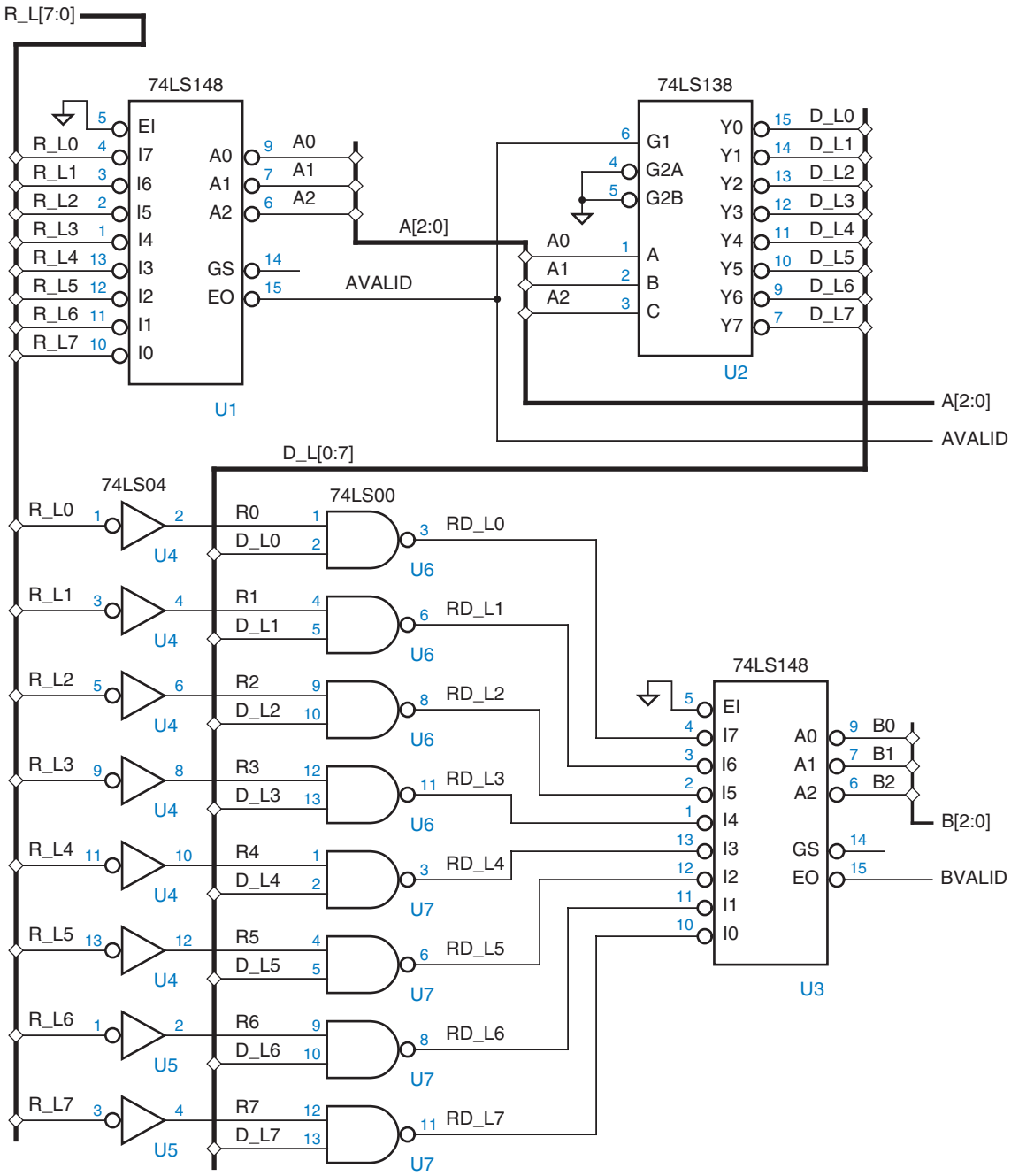


Figure XCbb-5 First- and second-highest-priority encoder circuit.

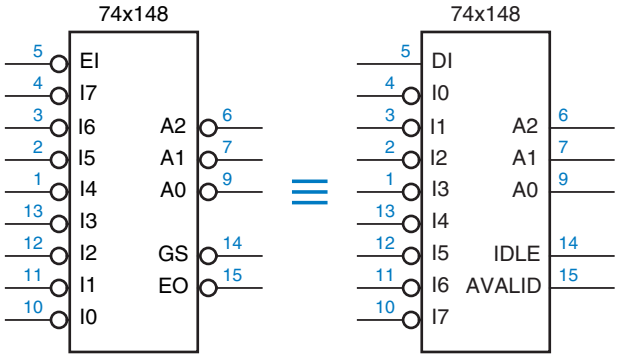


Figure XCbb-6 Alternate logic symbols for the 74x148 8-input priority encoder.

XCbb.4 Cascading Comparators

In Section 6.9.4, we showed how 74x85 4-bit comparators can be cascaded to create larger comparators. Since the 74x85 uses a serial cascading scheme, it can be used to build arbitrarily large comparators. The 74x682 8-bit comparator, on the other hand, doesn't have any cascading inputs and outputs at all. Thus, at first glance, you might think that it can't be used to build larger comparators. But that's not true.

If you think about the nature of a large comparison, it is clear that two wide inputs, say 32 bits (four bytes) each, are equal only if their corresponding bytes are equal. If we're trying to do a greater-than or less-than comparison, then the corresponding most-significant bytes that are not equal determine the result of the comparison.

Using these ideas, Figure XCbb-7 on the next page uses three 74x682 8-bit comparators to do equality and greater-than comparison on two 24-bit operands. The 24-bit results are derived from the individual 8-bit results using combinational logic for the following equations:

$$PEQQ = EQ2 \cdot EQ1 \cdot EQ0$$
$$PGTQ = GT2 + EQ2 \cdot GT1 + EQ2 \cdot EQ1 \cdot GT0$$

This "parallel" expansion approach is actually faster than the 74x85's serial cascading scheme, because it does not suffer the delay of propagating the cascading signals through a cascade of comparators. The parallel approach can be used to build very wide comparators using two-level AND-OR logic to combine the 8-bit results, limited only by the fan-in constraints of the AND-OR logic. Arbitrary large comparators can be made if you use additional levels of logic to do the combining.

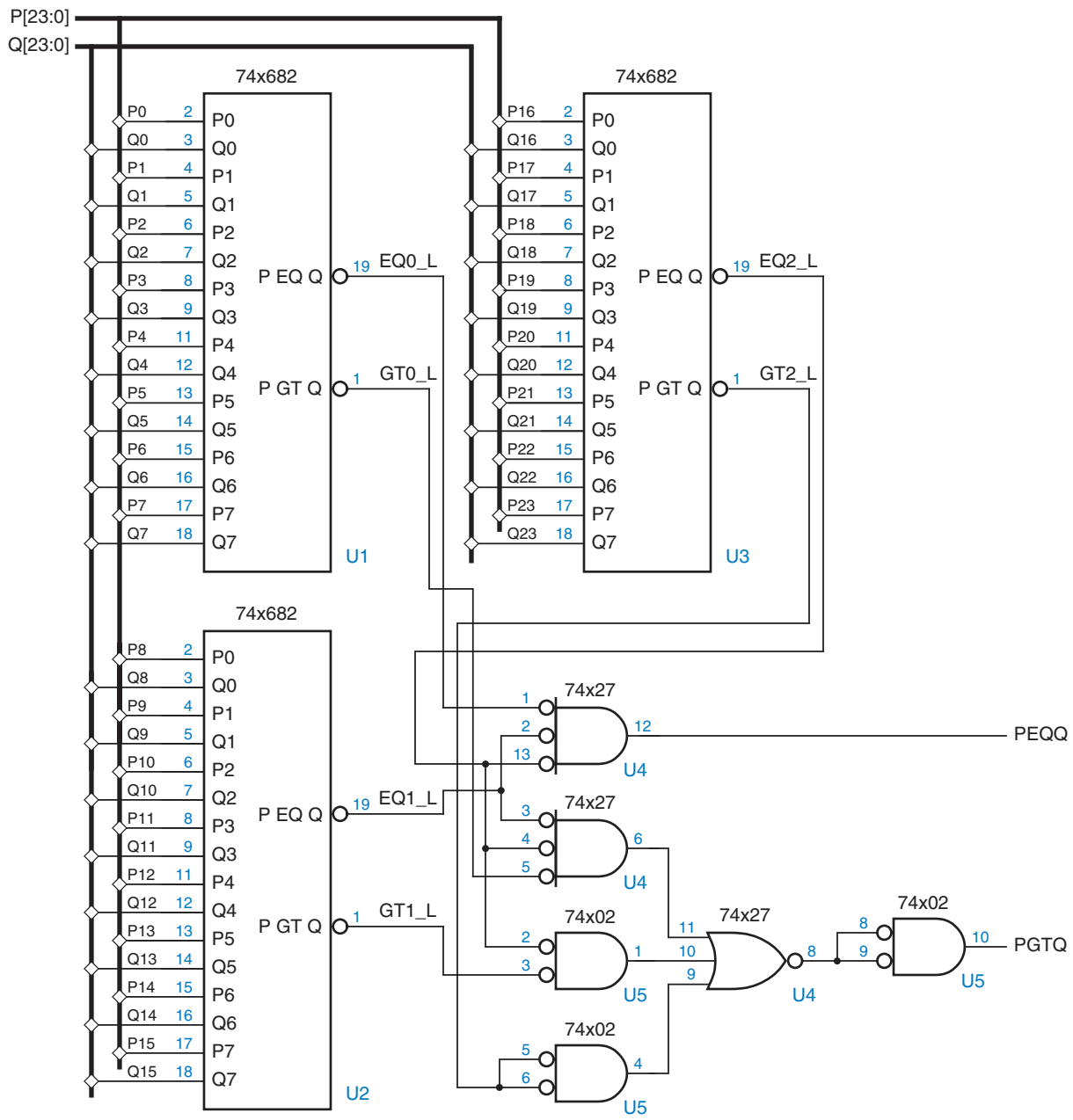


Figure XCbb-7 24-bit comparator circuit.

XCbb.5 Mode-Dependent Comparator

Quite often, the requirements for a digital-circuit application are specified in a way that makes an MSI or other building-block solution obvious. For example, consider the following problem:

- Design a combinational circuit whose inputs are two 8-bit unsigned binary integers, X and Y, and a control signal MIN/MAX. The output of the circuit is an 8-bit unsigned binary integer Z such that Z = min(X,Y) if MIN/MAX = 1, and Z = max(X,Y) if MIN/MAX = 0.

This circuit is fairly easy to visualize in terms of MSI functions. Clearly, we can use a comparator to determine whether X > Y. We can use the comparator's output to control multiplexers that produce min(X,Y) and max(X,Y), and we can use another multiplexer to select one of these results depending on MIN/MAX. Figure XCbb-8(a) on the next page is the block diagram of a circuit corresponding to this approach.

Our first solution approach works, but it's more expensive than it needs to be. Although it has three 2-input multiplexers, there are only two input words, X and Y, that may ultimately be selected and produced at the output of the circuit. Therefore, we should be able to use just a single 2-input mux, and use some other logic to figure out which input to tell it to select. This approach is shown in Figure XCbb-8(b) and (c). The "other logic" is very simple indeed, just a single XOR gate.

DON'T BE A BLOCKHEAD

The wastefulness of our original design approach in Figure XCbb-8(a) may have been obvious to you from the beginning, but it demonstrates an important approach to designing with building blocks:

- Use standard building blocks to handle data, and look for ways that a single block can perform different functions at different times or in different modes. Design control circuits to select the appropriate functions as needed, to reduce the total parts count of the design.

As Figure XCbb-8(c) dramatically shows, this approach can save a lot of chips. When designing with IC chips, you should *not* heed the slogan, "Have all you want, we'll make more"!

Exercises

XCbb.1 Explain how the 16-bit barrel shifter of Section XCbb.1 can be realized with a combination of 74x157s and 74x151s. How does this approach compare with the others in delay and parts count?

XCbb.2 Show how the 16-bit barrel shifter of Section XCbb.1 can be realized in eight identical GAL22V10s.

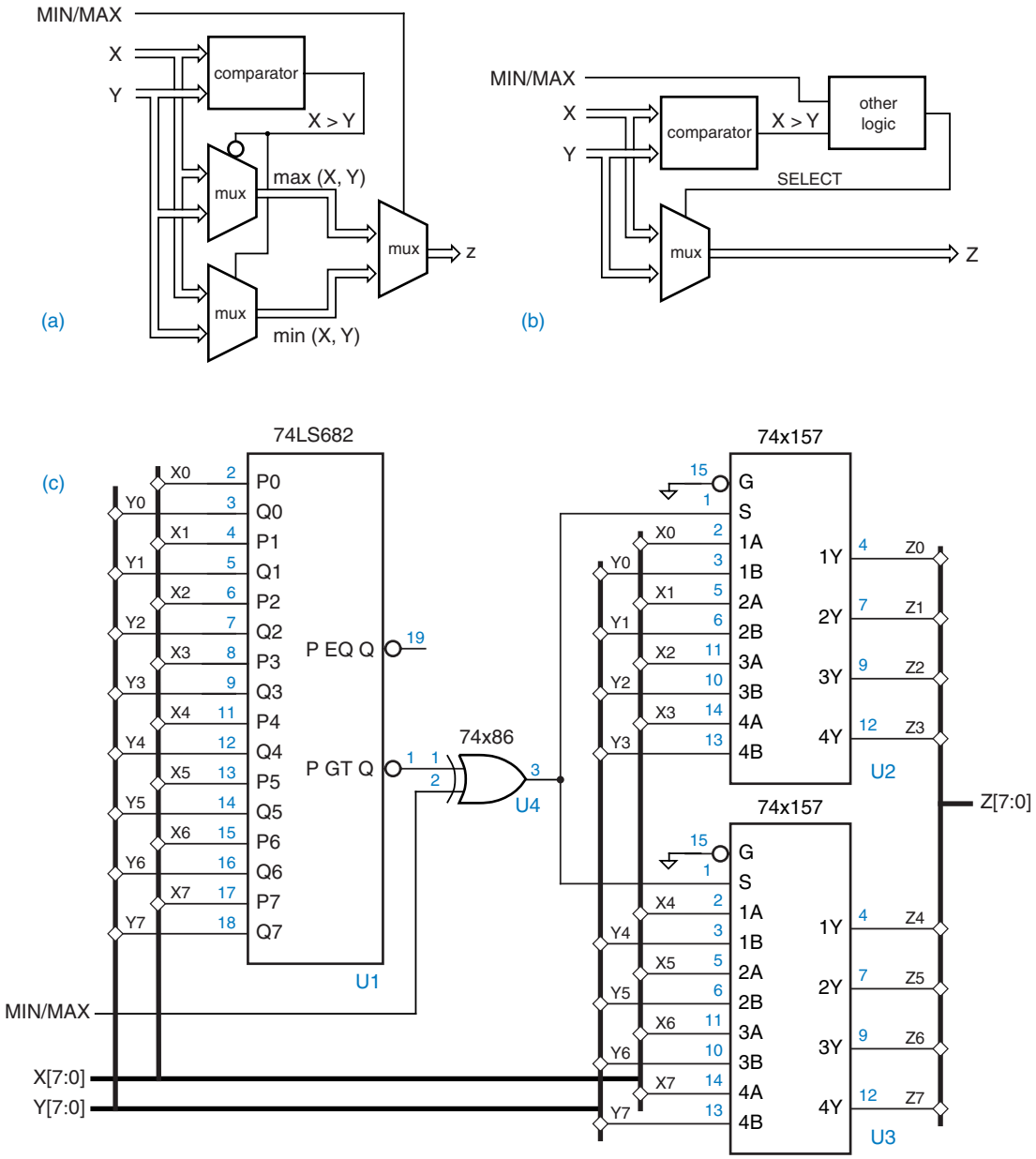


Figure XCbb-8 Mode-dependent comparator circuit: (a) block diagram of a "first-cut" solution; (b) block diagram of a more cost-effective solution; (c) logic diagram for the second solution.