Chapter 6

Spring 2010 Edition

Frequently Used Digital Circuits

A *SUMMARY* of what you learned in Chapter 5:

1- Octal and hexadecimal numbers.
2- Paper-and-pencil algorithm for unsigned addition.
3- Full adders (FAs) and ripple-carry adders: implementation of paper-and-pencil addition.
4- Paper-and-pencil algorithm for unsigned subtraction.
5- Full subtractors (FSs) and ripple-borrow subtractors: implementation of paper-and-pencil subtraction.
6- Addition-based subtraction: subtraction without subtraction!
7- Modified ripple adders to support subtraction without subtraction.
8- Half adders (HA) and half subtractors (HS).
9- Sign-and-magnitude, 1's complement and 2's complement signed number systems.
10- Addition/subtraction in 2's complement system.
11- Sign extension and overflow in 2's complement system.
12- Gray code and ASCII code.

Introduction

In this chapter frequently used digital circuits with some practical examples are presented. In the design of digital systems we do not have to begin everything from scratch. Frequently used digital circuits have already been designed, built and tested. So, we may use them as building blocks larger than basic gates, resulting in significant design/verification time/energy savings. In this chapter three frequently used off-the-shelf *medium-scale integration* (MSI) chips (equivalent to around 20 to 200 gates each) are introduced for the manual design of small systems (as opposed to computerized design of large systems). The chips that you have used in the lab experiments so far are called *small-scale integration* (SSI), typically with the equivalent of around 1-20 gates each. As technology improved *large-scale integration* (LSI) chips, each containing the equivalent of 200 to 1,000,000 gates or more, were introduced. Today's technology offers *very large scale integration* (VLSI) with over-100-million-transistor chips.

In this chapter you also learn the concept of *non-systematic* design. In this technique a (complicated) problem is decomposed into some smaller, hence easily solvable and/or already solved pieces. Then the solutions to these small problems are stitched together to reach a complete solution for the whole problem.

For interested readers there are three extra design examples in three appendixes at the end of this chapter. However, reading and understanding these extra materials are not essential for reading and understanding the rest of this book.

Decoders

We have already seen a couple of codes, such as the normal binary code and Gray code. As an example, consider 3-bit binary numbers: technically speaking, every 3-bit pattern such as 011 is called a 3-bit binary *code word*, and the set of all $2^3 = 8$ possible code words, {000, 001, 010, 011, 100, 101, 110, 111}, is called the 3-bit binary *code*. These two definitions are valid for other codes as well. A circuit that converts an *n*-bit code word (of one code) into an *m*-bit code word (of another code) is usually called a *decoder* if $n < m$, or an *encoder* if $n > m$. In this section we study two types of decoders, namely *binary* decoders and *seven-segment-to-BCD* decoders.

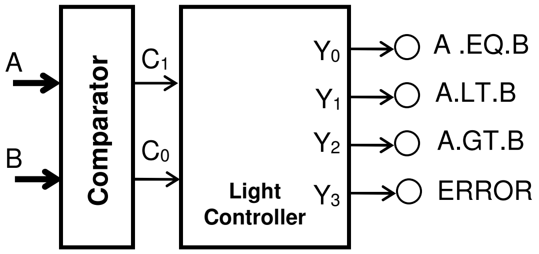
n-to-2^n or Binary Decoders

The following two examples should show you, to some extent, how important binary decoders are. Note that by an input (or output) *vector*, we mean an input (or output) signal that is wider than one bit.

Example 1. Suppose that a comparator with two input vectors, A and B, produces two output bits C_1 and C_0, which are interpreted as follows: the three 2-bit patterns $C_1\ C_0$ = 00, 01 or 10 mean A = B, A < B or A > B respectively; output $C_1\ C_0$ = 11 signifies an error, as listed in Figure 1*a*. There are four LEDs called A.EQ.B, A.LT.B, A.GT.B and ERROR that correspond to these four different situations, respectively. In this example, we are to design a light controller that takes C_1 and C_0 as two inputs, and produces one output for each LED as shown in Figure 1*b*, to turn the right light on. Y_0, Y_1, Y_2 and Y_3 are the four outputs driving the LEDs. Let's assume that in this example an LED turns on with a 1. Therefore, if A happens to be greater than B (i.e., $C_1\ C_0$ = 10), only output Y_2 must go up to turn light A.GT.B on, while the other lights are left off. Or, if an error occurs (i.e., $C_1\ C_0$ = 11), only output Y_3 must be pulled up to turn light ERROR on, and so on. Figure 1*c*, shows a truth table for the four outputs of this controller. Each output is asserted in only one unique row. A circuit that takes $C_1\ C_0$ as two inputs and generates Y_0, Y_1, Y_2 and Y_3 is technically called a 2-to-4 *binary decoder* or DCD for short.

$C_1\ C_0$	Meaning
0 0	A = B
0 1	A < B
1 0	A > B
1 1	Error

(a)



(b)

$C_1\ C_0$	Y_3	Y_2	Y_1	Y_0
0 0	0	0	0	1
0 1	0	0	1	0
1 0	0	1	0	0
1 1	1	0	0	0

(c)

Figure 1. Light controller: (a) meanings of 4 input combinations, (b) I/O signals and LEDs, (c) truth table

In general, an n-to-2^n binary decoder has n input (or select) lines and 2^n output lines. Every input combination corresponds to exactly one output line and vice versa; so that at each instant of time only one output, output number i, is asserted, where i ($0 \leq i \leq 2^n - 1$) is the (binary) number placed on the n select lines of the decoder at that instant of time. (That is why this output code is called *one-out-of-2^n*.) In the truth-table domain, each output of a DCD is asserted in one unique row only; in other words the on-set of each output consists of *exactly one minterm*. The immediate consequences of this specific architecture are as follows:

1) None of the output functions can be simplified, as each function has only one on-set minterm; hence there is no other on-set minterm to be combined with. Therefore, each output needs one n-input AND gate to get realized, where n is the number of input bits. As an example, see Figure 2*a*, which shows a K-map for output Y_3 of the light controller, and the corresponding realization. As we should be expecting, there is only one 1-cell in this four-cell K-map. Therefore, this K-map would be realized with a 2-input AND gate, as depicted in this figure. The K-map/AND-gate pairs for other outputs, Y_2, Y_1 and Y_0, are shown in Figure 2*b*, Figure 2*c*, Figure 2*d*, respectively. We say that the AND gates in Figure 2*a,* Figure 2*b*, Figure 2*c* and Figure 2*d decode* the bit patterns C_1C_0 = 11, 10, 01 and 00, respectively; i.e., the output of each AND gate is only asserted by the corresponding input combination. We also say that inputs C_1C_0 are decoded by this decoder resulting in decoded outputs Y_3, Y_2, Y_1 and Y_0.

2) All possible n-bit minterms have already been realized in an n-to-2^n DCD. This means that every n-variable function can be realized using an n-to-2^n DCD and one proper-type/proper-size gate. I will elaborate on this concept shortly.

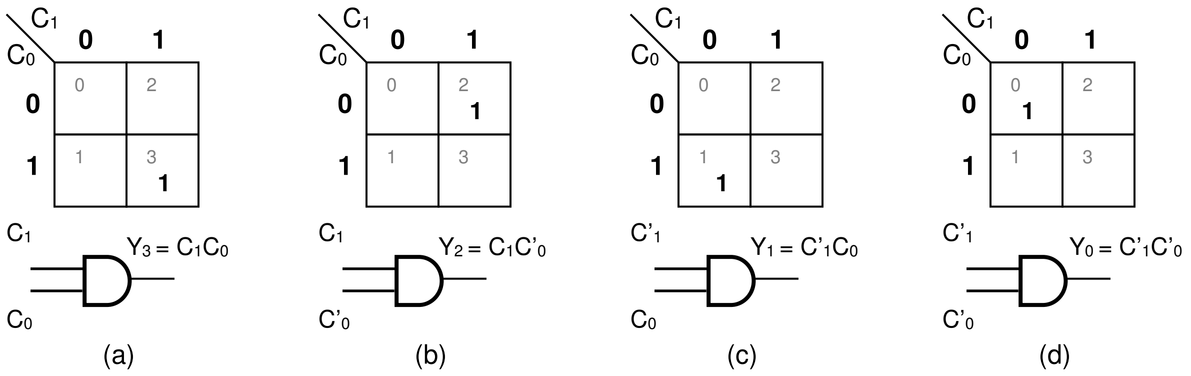


Figure 2. K-maps and logic circuits to realize a 2-to-4 DCD

Example 2. Design a circling light: There are 8 LEDs (LED0 to LED7) fixed around a circle, as shown in Figure 3*a*. Suppose that 3-bit binary code words (000 to 111, and again 000 to 111 and so on) are sequentially generated and placed on three lines, A_2 A_1 A_0, every 100 ms. To get a circling light we need a 3-to-8 DCD that receives A_2 A_1 A_0 as its inputs and drives the eight LEDs by its eight outputs as shown in Figure 3*a*. The DCD takes a 3-bit binary code word from A_2 A_1 A_0 and then asserts exactly one of its eight outputs to turn the corresponding LED on for 100 ms, while the remaining seven LEDs are left off. During the following 100 ms the next output of the DCD is asserted turning the next LED on, while the other seven LEDs are off, and so on. In this way it looks that an on-LED is spinning around the circle.

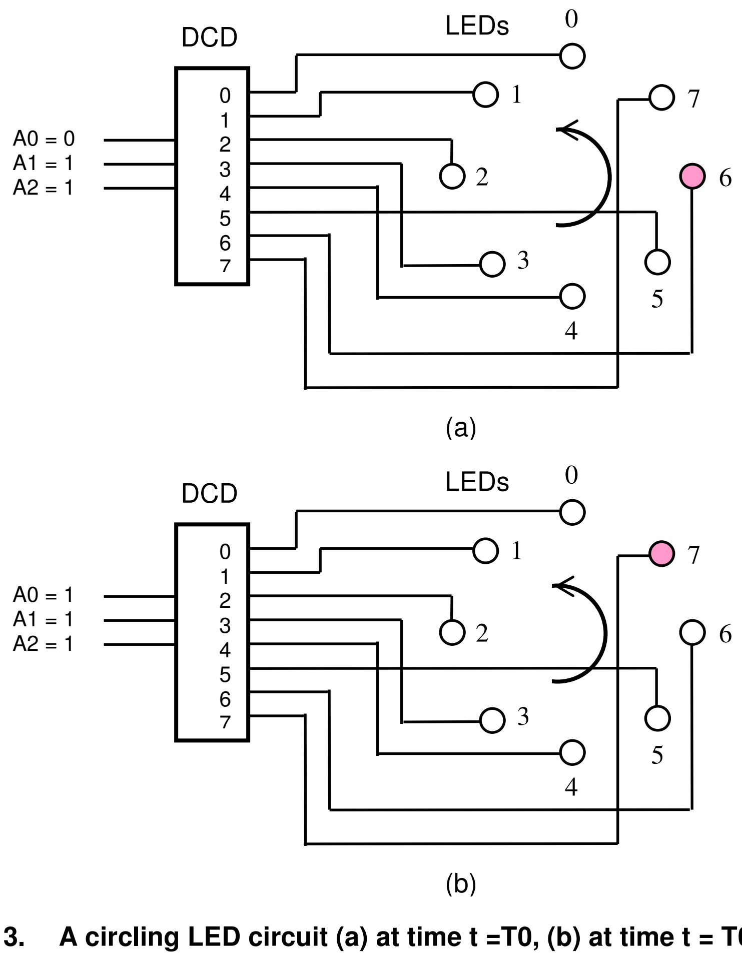


Figure 3. A circling LED circuit (a) at time t =T0, (b) at time t = T0 + 100 ms

In Figure 3*a* the binary number applied to the DCD is A_2 A_1 A_0 = 110 = 6, and therefore only LED 6 is on. One hundred milliseconds later the input bit pattern changes to A_2 A_1 A_0 = 111, turning LED 6 off and LED 7 on, as shown in Figure 3*b*. We do not know yet how to periodically generate 3-bit binary numbers applied to this DCD. Be patient! This and similar topics will be covered in Chapter 8.

A truth table and logic circuit for this DCD are shown in Figure 4 and Figure 5, respectively. In this example we again assume that an LED turns on with a 1; that is why a DCD with active-high outputs have been used. Also notice that in order to improve clarity, inverters (to complement input signals) are not shown in Figure 5 (or similar figures in this section). ◊

Row	A2	A1	A0	Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0	0	0	1	0
2	0	1	0	0	0	0	0	0	1	0	0
3	0	1	1	0	0	0	0	1	0	0	0
4	1	0	0	0	0	0	1	0	0	0	0
5	1	0	1	0	0	1	0	0	0	0	0
6	1	1	0	0	1	0	0	0	0	0	0
7	1	1	1	1	0	0	0	0	0	0	0

Figure 4. Truth table for 3-to-8 active-high DCD

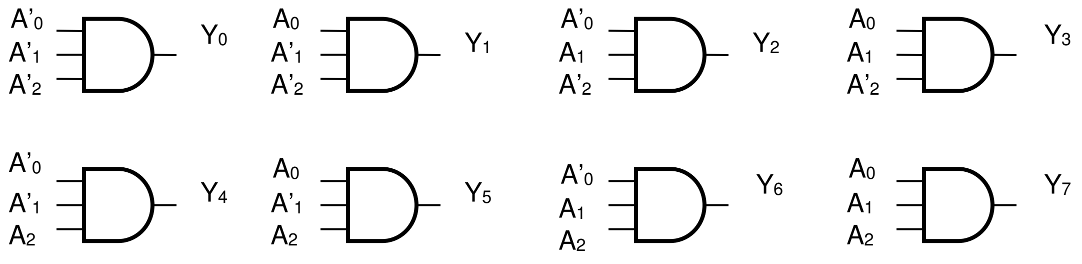


Figure 5. Logic circuit for 3-to-8 active-high DCD

One *enable* input, E, (or more) may be added to a decoder as shown in the truth table of Figure 6. Enable inputs have to be asserted to let the decoder operate as a decoder; otherwise all outputs will remain deasserted, as shown in rows 8-15 of this truth table. In order to realize this feature and add it to the decoder shown in Figure 5, we need to add a forth input, E, to each AND gate of this decoder; so the 3-input AND gates now become 4-input gates, as shown in Figure 7.

Row	E	A2	A1	A0	Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0
0	1	0	0	0	0	0	0	0	0	0	0	1
1	1	0	0	1	0	0	0	0	0	0	1	0
2	1	0	1	0	0	0	0	0	0	1	0	0
3	1	0	1	1	0	0	0	0	1	0	0	0
4	1	1	0	0	0	0	0	1	0	0	0	0
5	1	1	0	1	0	0	1	0	0	0	0	0
6	1	1	1	0	0	1	0	0	0	0	0	0
7	1	1	1	1	1	0	0	0	0	0	0	0
8-15	0	x	x	x	0	0	0	0	0	0	0	0

Figure 6. Truth table for 3-to-8 DCD with one active-high enable input (E)

Remember from Chapter 3 that the E input in Figure 7 is called *active high* because the decoder will be enabled if E is high. We may also design a decoder with an active-low enable input, as shown in the truth table of Figure 8. To realize this truth table, the only change that we need to make to the circuit shown in Figure 7 is to invert input E before it is applied to the eight 4-input AND gates, as shown in Figure 9.

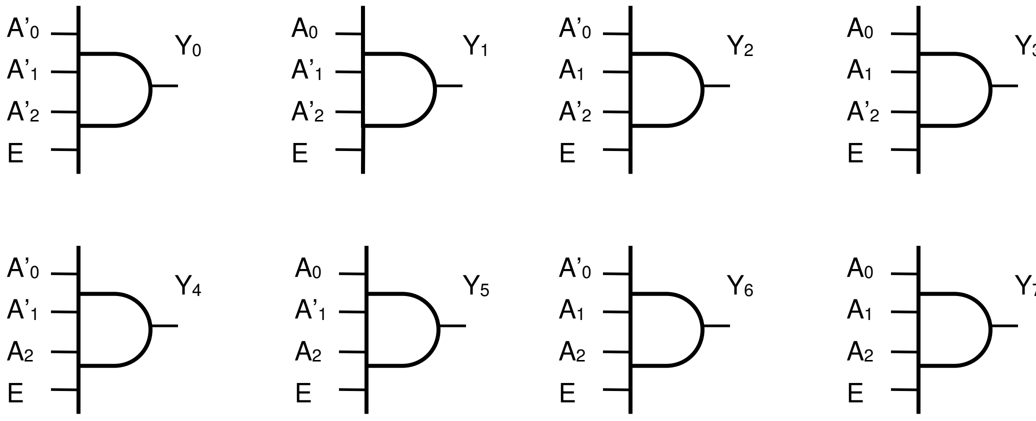


Figure 7. Logic circuit for a 3-to-8 DCD with one active-high enable input (E)

Row	~E	A2	A1	A0	Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0	0	0	0	0	1
1	0	0	0	1	0	0	0	0	0	0	1	0
2	0	0	1	0	0	0	0	0	0	1	0	0
3	0	0	1	1	0	0	0	0	1	0	0	0
4	0	1	0	0	0	0	0	1	0	0	0	0
5	0	1	0	1	0	0	1	0	0	0	0	0
6	0	1	1	0	0	1	0	0	0	0	0	0
7	0	1	1	1	1	0	0	0	0	0	0	0
8-15	1	x	x	x	0	0	0	0	0	0	0	0

Figure 8. Truth table for 3-to-8 DCD with one active-low enable input (~E)

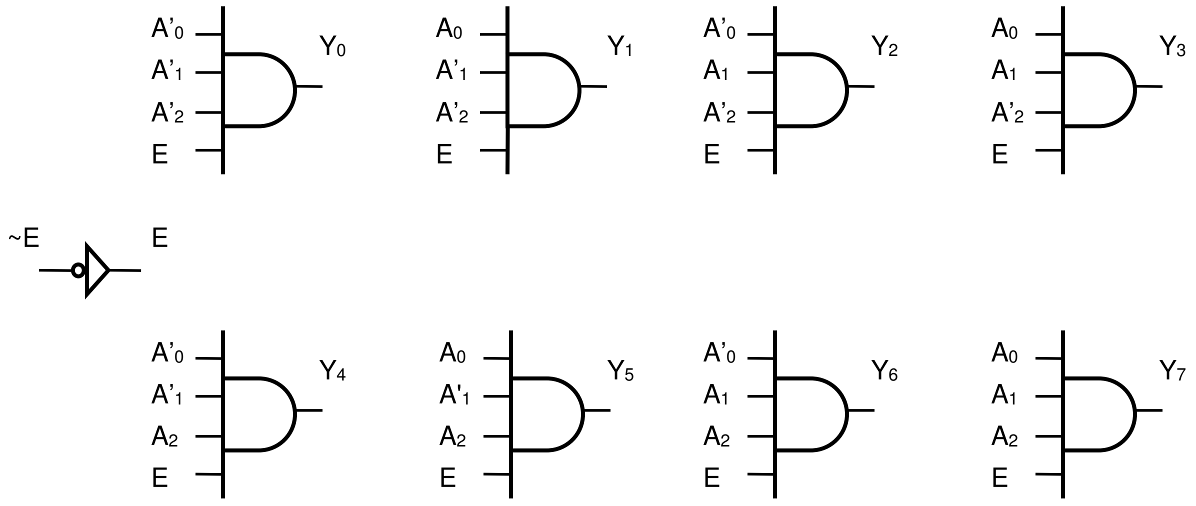


Figure 9. Logic circuit for 3-to-8 DCD with one active-low enable input (~E)

In Figure 9, for every input combination the single output distinguished from the other outputs is *high* (and the other ones are low). Therefore, the outputs (by definition) are active high (similar to the outputs in Figure 5 or Figure 7). We will get a DCD with active-low outputs if the situation is reversed, i.e., the distinguished output is now *low*, and the other ones are high, as shown in Figure 10. In the circuit domain active-low outputs may be realized by simply inverting all active-high outputs of a DCD as shown in Figure 11. Remember that in this book we signify an active-low input/output or signal name with a ~. Additionally, an active-low input/output pin receives a bubble in logic diagrams. ◊

Row	~E	A2	A1	A0	$\sim Y_7$	$\sim Y_6$	$\sim Y_5$	$\sim Y_4$	$\sim Y_3$	$\sim Y_2$	$\sim Y_1$	$\sim Y_0$
0	0	0	0	0	1	1	1	1	1	1	1	0
1	0	0	0	1	1	1	1	1	1	1	0	1
2	0	0	1	0	1	1	1	1	1	0	1	1
3	0	0	1	1	1	1	1	1	0	1	1	1
4	0	1	0	0	1	1	1	0	1	1	1	1
5	0	1	0	1	1	1	0	1	1	1	1	1
6	0	1	1	0	1	0	1	1	1	1	1	1
7	0	1	1	1	0	1	1	1	1	1	1	1
8-15	1	x	x	x	1	1	1	1	1	1	1	1

Figure 10. Truth table for 3-to-8 DCD with active-low outputs and one active-low enable input (~E)

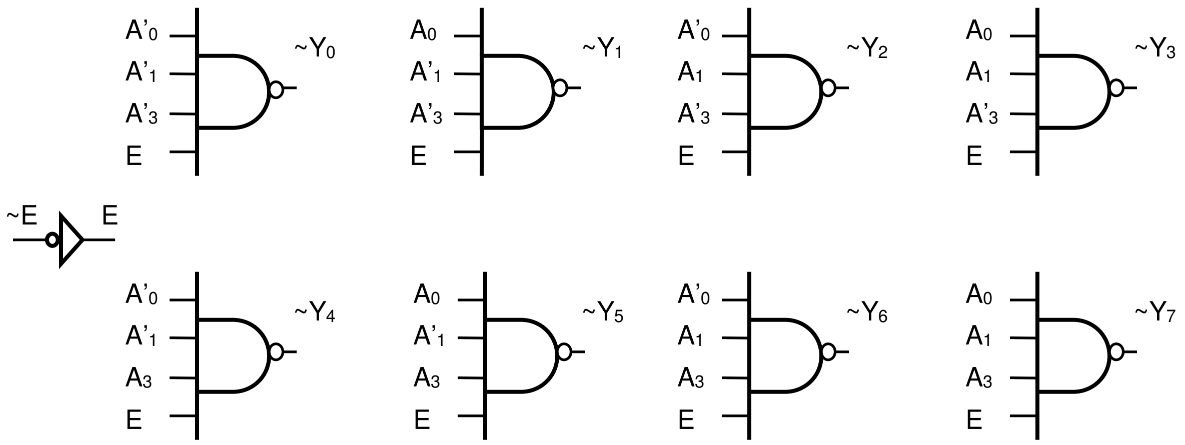


Figure 11. Logic circuit for 3-to-8 DCD with active-low outputs and an active-low enable input (~E)

Considering the definition of a DCD, and as clearly seen in different figures depicted previously such as Figure 5, all (n-variable) minterms have already been realized in an n-to-2^n active-high decoder. Therefore, any n-variable logic function, y, may be realized with an n-to-2^n active-high decoder followed by an m-input OR gate or an $(2^n - m)$-input NOR gate, where m is the number of on-set minterms of y, as shown in the following example for n = 3.

Example 3. Use an appropriate-size active-high decoder to realize $y = \Sigma_{a,b,c}(1, 5, 7)$.

Since y is a function of three variables[1] we need a 3-to-8 active-high decoder to pick the minterms 1, 5 and 7, and then apply them to a 3-input OR gate as shown in Figure 12*a*. (By, say, m0 in this figure we

[1] Notice that writing, say, four variables next to Σ or ∏ does not necessarily mean that the corresponding function is a 4-variable function. For example, consider $y = \Sigma_{a,b,c,d}(0, 2, 5, 7, 9, 11)$. It can be shown that $y = \Sigma_{a,b,d}(0, 3, 5)$, i.e., y is independent of c. We consider this conversion (from four to three variables) as part of the minimization procedure. In general, we should make sure that redundant variables have been removed before proceeding with implementation.

mean minterm0, which is a' . b' . c'.) Alternatively, in Figure 12*b* the off-set minterms of y have been applied to a five-input NOR gate to realize y.

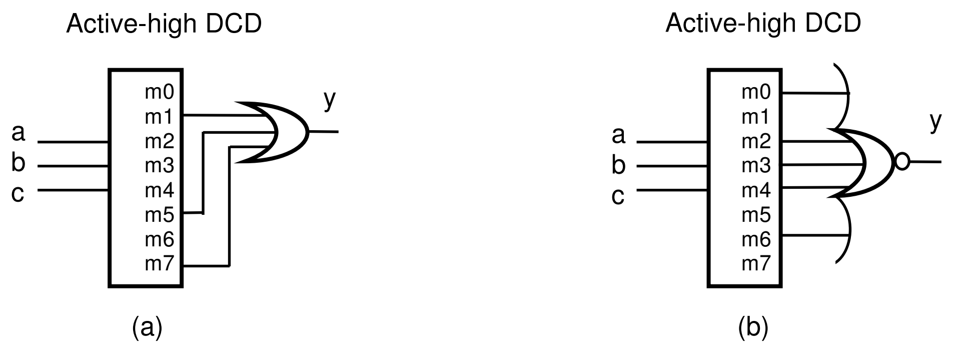


Figure 12. Function y (a, b, c) = Σ (1, 5, 7) realized with an active-high decoder using (a) on-set minterms, (b) offset minterms

On the other hand, an n-to-2^n active-low decoder generates all (n-variable) maxterms. Therefore, any n-variable logic function, y, may be realized with an n-to-2^n active-low decoder, followed by an m input AND gate or an (2^n – m)-input NAND gate, where m is the number of off-set maxterms of y, as shown in the following example for n = 3:

Example 4. Use an appropriate-size active-low decoder to realize function $y = \Sigma_{a,b,c}(1, 5, 7)$.

The off-set maxterm list of this function is $y = \prod_{a,b,c}(0, 2, 3, 4, 6)$. Since y is a function of three variables, we need a 3-to-8 active-low decoder to pick the maxterms 0, 2, 3, 4 and 6, and then apply them to a 5-input AND gate as shown in Figure 13*a*. (A maxterm is the complement of the corresponding minterm; as an example, maxterm0 = m0' = (a'. b'. c')' = (a + b + c).) Alternatively, in Figure 13*b* the on-set maxterms of y have been applied to a three-input NAND gate to realize y.

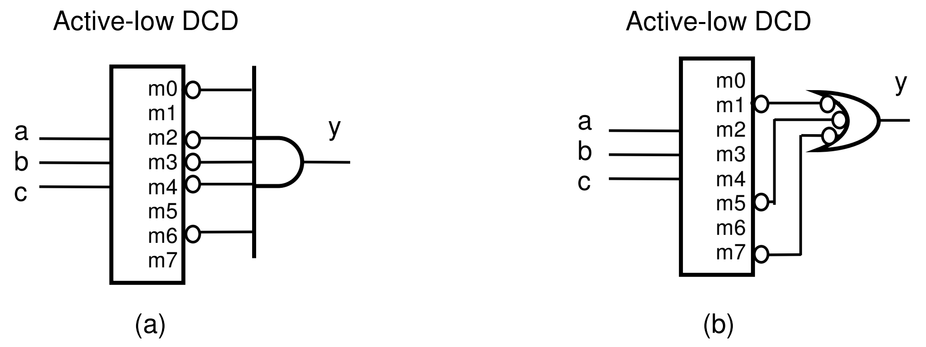
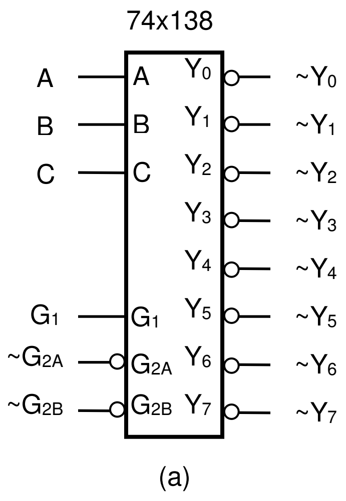


Figure 13. Function y (a, b, c) = Σ (1, 5, 7) realized with an active-low decoder using (a) offset maxterms, (b) on-set maxterms

TTL 74x138:

A logic and a shorthand symbol for the TTL 74x138, an MSI 3-to-8 binary decoder, are shown in Figure 14*a* and Figure 14*b*, respectively. This chip has three active-high inputs, C (MSB), B and A, eight active-low outputs, ~Y7, ~Y6, ~Y5, ~Y4, ~Y3, ~Y2, ~Y1 and ~Y0, two active-low enable inputs, ~G2A and ~G2B, and one active-high enable input, G1. Figure 15 illustrates a truth table for this decoder. All enable inputs have to be asserted to let the 74x138 function properly as a decoder; even one deasserted enable input would disable the decoder resulting in all-high (or deasserted) outputs, as shown in the last three rows of the truth table in Figure 15.



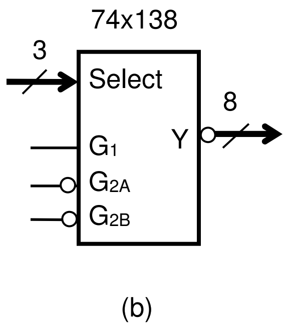


Figure 14. 74x138: (a) logic symbol, (b) shorthand symbol

G	~G_{2A}	~G_{2B}	C	B	A	~Y_7	~Y_6	~Y_5	~Y_4	~Y_3	~Y_2	~Y_1	~Y_0
1	0	0	0	0	0	1	1	1	1	1	1	1	0
1	0	0	0	0	1	1	1	1	1	1	1	0	1
1	0	0	0	1	0	1	1	1	1	1	0	1	1
1	0	0	0	1	1	1	1	1	1	0	1	1	1
1	0	0	1	0	0	1	1	1	0	1	1	1	1
1	0	0	1	0	1	1	1	0	1	1	1	1	1
1	0	0	1	1	0	1	0	1	1	1	1	1	1
1	0	0	1	1	1	0	1	1	1	1	1	1	1
0	x	x	x	x	x	1	1	1	1	1	1	1	1
x	1	x	x	x	x	1	1	1	1	1	1	1	1
x	x	1	x	x	x	1	1	1	1	1	1	1	1

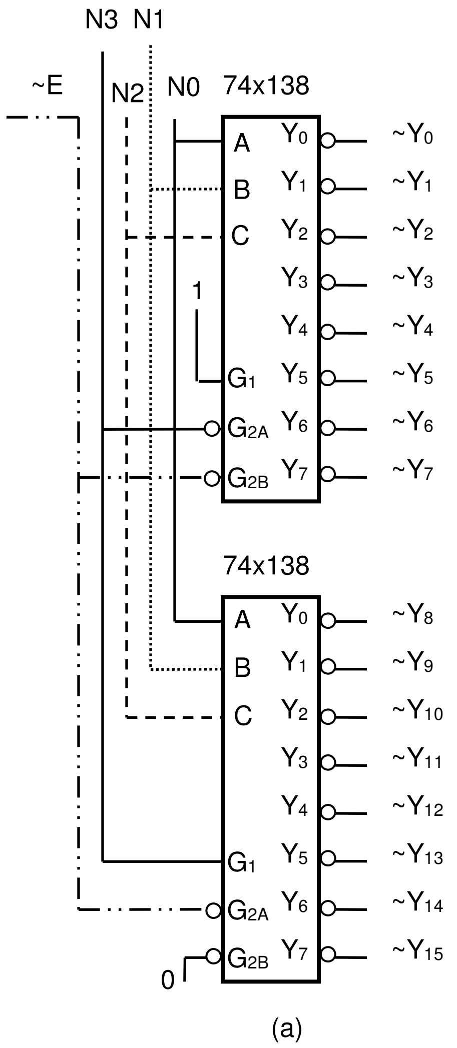
Figure 15. Truth table for 74x138

Pay close attention to the labeling scheme adopted in this book. In Figure 14*a* if we put, say, G_{2A} outside the rectangle representing the chip, the pin name will receive a ~. In other words, a pin name inside the box must not have a ~ while an inversion bubble is attached to that pin. ◊

Different applications may need different-size decoders. However, it is not economical to manufacture decoders in many sizes. So, a couple of reasonable-size decoders are usually manufactured and made available but with some provisions for possible expansions. This is true for other digital building blocks as well. In the following example two 74x138 decoders are cascaded to obtain a 4-to-16 decoder.

Example 5. Use two 74x138 chips to obtain a 4-to-16 DCD.

The design is illustrated in Figure 16*a*. As shown in this figure the unused G1 of the top (least significant) chip and the unused ~G2B of the bottom (most significant) chip have been deactivated by being tied to logic 1 and logic 0, respectively. On the other hand, ~G2B of the top chip and ~G2A of the bottom chip have been connected to each other to become an enable input (~E) for the newly built 4-to-16 decoder. In Figure 16 the four external inputs arriving at this 4-to-16 decoder have been labeled N3 N2 N1 N0. The three LSBs, N2, N1 and N0, go to C, B and A inputs, respectively, of both decoders. The MSB (N3), on the other hand, goes to ~G_{2A} and G_1 of the top chip and the bottom chip, respectively. When N3 is low, then the top chip is enabled but the bottom one is disabled resulting in all-high outputs for the bottom chip. Therefore, the bit pattern on N2, N1 and N0 would be decoded by the top chip whenever N3 = 0. Similarly, when N3 is high the bottom chip is enabled but the top one is disabled resulting in all-high outputs for the top chip. Therefore, the bit pattern on N2, N1 and N0 will be decoded by the bottom chip whenever N3 = 1. For example, N3 N2 N1 N0 = 1011 will be decoded to 11110111 11111111, while N3 N2 N1 N0 = 0011 will result in 11111111 11110111. Figure 16*b* shows the same design using shorthand symbols.


N3 N1
~E
N2
N0
74x138
A
B
C
1
G1
G2A
G2B
Y0 Y1 Y2 Y3 Y4 Y5 Y6 Y7
~Y0 ~Y1 ~Y2 ~Y3 ~Y4 ~Y5 ~Y6 ~Y7
74x138
~Y8 ~Y9 ~Y10 ~Y11 ~Y12 ~Y13 ~Y14 ~Y15
0
(a)

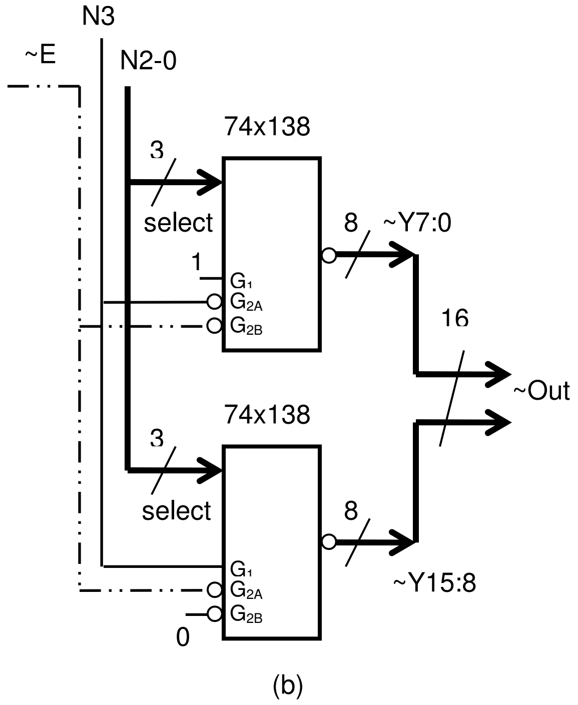


Figure 16. A 4-to-16 DCD using two 74x138 chips: (a) logic diagram, (b) shorthand diagram

In Appendix A four 74x138 chips are used to reach a 5-to-32 DCD.

BCD and 7-segment Codes and BCD to 7-segment Conversion

In this section two more codes, namely *binary coded decimal* (BCD) and *7-segment* are studied and then the BCD to 7-segment decoder is introduced.

BCD: Binary Coded Decimal

You have already realized the inconvenience in converting binary to decimal and vice versa. One solution to overcome this difficulty is to use *binary coded decimal* (or BCD for short), instead of straight binary. BCD maintains the decimal nature of numbers, and only converts each decimal digit (in the range of 0 to 9) *individually* to its 4-bit binary equivalent (0000 to 1001). For example, 793 is now represented as 0111

1001 0011, which is obtained by a trivial conversion. This convenience is achieved at a very high cost: with, say, eight bits, $2^8 = 256$ different numbers can be represented in binary, while using the same number of bits, only 100_{ten} numbers (less than half), 0000 0000 (= 0) to 1001 1001 (= 99), may be represented in BCD.

The four bits comprising one decimal digit in BCD have the same weights (8, 4, 2 and 1) that bits have in the normal 4-bit binary system; that is why BCD is also called the *8421* code. Figure 17 shows the BCD digits and their decimal equivalents.

BCD	Decimal value	BCD	Decimal value
0000	0	0101	5
0001	1	0110	6
0010	2	0111	7
0011	3	1000	8
0100	4	1001	9

Figure 17. Binary coded decimal (BCD) code

7-Segment Code

A seven-segment display shown in Figure 18*a* usually uses seven light-emitting diodes (LEDs) or seven liquid crystal display (LCD) segments called a, b, c, d, e, f and g; so that a decimal digit can be displayed on a seven-segment display by turning on the correct subset of these segments, as shown in Figure 18*b*.

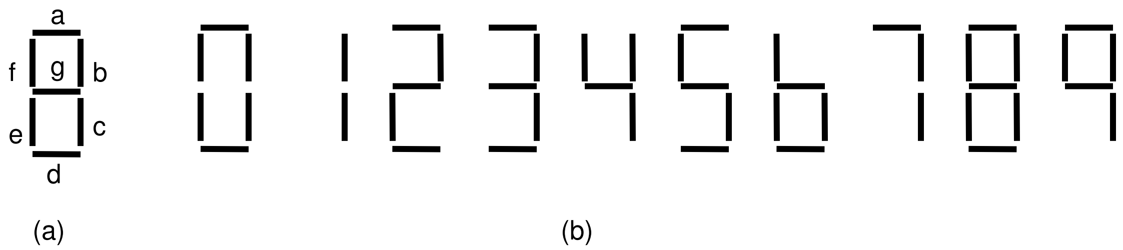


Figure 18. Seven-segment display: (a) segments' names, (b) decimal digits representations

A 7-bit vector that turns on a subset of these segments *meaningfully* is called a 7-segment *code word*, and the set of all ten 7-segment code words is called the *7-segment code*. Whether a 1-bit or a 0-bit is intended to turn the corresponding segment on, the 7-segment code is called active high or active low, respectively. The active-high 7-segment code words are shown in Figure 19*a*.

BCD to 7-segment Decoder

An active-high BCD to 7-segment decoder takes four active-high inputs (say, D (MSB), C, B and A) in BCD and generates the corresponding *7-segment* active-high code word shown in the truth table of Figure 19*b*. We may of course design and/or use an active-low decoder, but now a segment in the display is assumed to turn on with a 0. In other words, the seven-segment display that we choose has to be consistent with the decoder to be used.

The unused six rows (10-15) in Figure 19*b* can be used to display some other meaningful characters such as A-F. This of course entails a more expensive decoder.

Decimal	a	b	c	d	e	f	g
0	1	1	1	1	1	1	0
1	0	1	1	0	0	0	0
2	1	1	0	1	1	0	1
3	1	1	1	1	0	0	1
4	0	1	1	0	0	1	1
5	1	0	1	1	0	1	1
6	0	0	1	1	1	1	1
7	1	1	1	0	0	0	0
8	1	1	1	1	1	1	1
9	1	1	1	0	0	1	1

(a)

Decimal	D	C	B	A	a	b	c	d	e	f	g
0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	0	1	1	0	0	0	0
2	0	0	1	0	1	1	0	1	1	0	1
3	0	0	1	1	1	1	1	1	0	0	1
4	0	1	0	0	0	1	1	0	0	1	1
5	0	1	0	1	1	0	1	1	0	1	1
6	0	1	1	0	0	0	1	1	1	1	1
7	0	1	1	1	1	1	1	0	0	0	0
8	1	0	0	0	1	1	1	1	1	1	1
9	1	0	0	1	1	1	1	0	0	1	1
10-15					x	x	x	x	x	x	x

(b)

Figure 19. (a) Active-high 7-segment code, (b) active-high BCD-to-7-segment decoder: truth table

Encoders
In this section binary and priority encoders are introduced.

Binary Encoders
A 2^n-to-n binary encoder carries out the opposite function of an n-to-2^n binary decoder; so that an 8-to-3 binary encoder (in which $n = 3$), as an example, receives eight bits with one asserted bit at-a-time, and generates a 3-bit binary number corresponding to the asserted input. Figure 20 shows a truth table for an 8-to-3 active-high binary encoder. For example, if input 3 (A3) is asserted, then the output is Y2Y1Y0 = 011, the binary equivalent for 3, as shown in the fourth row of this table.

A7	A6	A5	A4	A3	A2	A1	A0	Y2	Y1	Y0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**1**	**1**
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1
E	L	S	E	E	L	S	E	x	x	x

Figure 20. Truth table for an active-high 8-to-3 binary encoder

By inspecting the truth table in Figure 20 the logic expressions of the three outputs of this encoder are obtained as follows:

$Y_2 = A_7 + A_6 + A_5 + A_4$
$Y_1 = A_7 + A_6 + A_3 + A_2$
$Y_0 = A_7 + A_5 + A_3 + A_1$

Priority Encoders

The concept of binary encoders may be generalized to circumstances in which more than one input out of some *prioritized* inputs may be asserted at-a-time, and we need to identify the asserted input with *the highest priority* among all asserted inputs. For example, suppose that there is only one resource, but many clients (with different priorities and each with its own request line) need to have access to this single resource. In this context more than one client at-a-time may send an access request; therefore, a priority encoder is required to identify the requesting client with the highest priority. Figure 21 and Figure 22 show a logic symbol and a truth table, respectively, for the 74x148, an 8-bit priority encoder.

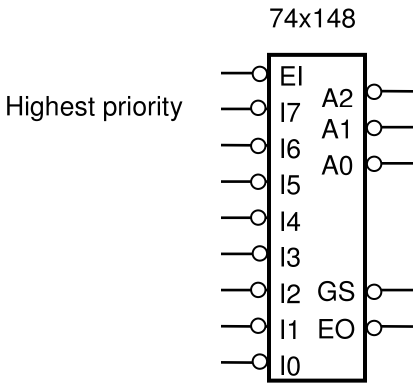


Figure 21. Logic symbol for 74x148, an 8-input priority encoder

~EI	~I0	~I1	~I2	~I3	~I4	~I5	~I6	~I7	~A2	~A1	~A0	~GS	~EO
1	x	x	x	x	x	x	x	x	1	1	1	1	1
0	x	x	x	x	x	x	x	0	0	0	0	0	1
0	x	x	x	x	x	x	0	1	0	0	1	0	1
0	x	x	x	x	x	0	1	1	0	1	0	0	1
0	x	x	x	x	0	1	1	1	0	1	1	0	1
0	**x**	**x**	**x**	**0**	**1**	**1**	**1**	**1**	**1**	**0**	**0**	**0**	**1**
0	x	x	0	1	1	1	1	1	1	0	1	0	1
0	x	0	1	1	1	1	1	1	1	1	0	0	1
0	0	1	1	1	1	1	1	1	1	1	1	0	1
0	1	1	1	1	1	1	1	1	1	1	1	1	0

Figure 22. Truth table for 74x148, an 8-input priority encoder

This chip has three active-low (encoded) output lines (~A2, ~A1, ~A0) and eight active-low input request lines (~I0, ~I1, ~I2 … ~I7) where ~I7 has the highest priority and ~I0 has the lowest priority. For example, row 6 of the truth table in Figure 22 reads "if input ~I3 is asserted while none of the inputs with higher priorities (~I4, ~I5, ~I6, ~I7) are asserted, then input ~I3 will be encoded resulting in ~A2 ~A1 ~A0 = 100, no matter whether or not any inputs with lower priorities (~I2, ~I1, ~I0) are asserted". Output lines ~A2, ~A1 and ~A0 are active-low as well; so, ~A2 ~A1 ~A0 = 100 would read ~A2 ~A1 ~A0 = 3, which is the index of the encoded input, ~I3.

Three outputs (~A2, ~A1 and ~A0) are not sufficient for representing *nine* different situations: one situation for the recognition of each request (2nd row to 9th row in Figure 22) and the 9th situation is when there is no request at all (the last row in Figure 22). The output control line ~GS solves this problem; it is deasserted when no request is received. ~GS will also be deasserted if the encoder is disabled, i.e., when the Enable input is deasserted (~EI = 1); otherwise ~GS will be asserted. In a disabled 74x148 all outputs are deasserted as shown in the first row of Figure 22. By simply "outputs" we mean ~A2, ~A1, ~A0 lines. Output *control* lines are distinguished by the corresponding labels (such as ~GS) and/or by the word "control".

In this encoder there is a second output control signal called ~EO (standing for Enable Output), which is used for expansion purposes together with ~EI and ~GS. As shown in the truth table of Figure 22, ~EO is asserted only in the last row, where the encoder is enabled but no requests have been received. In Appendix B we use four 74x148 chips to design a 32-bit priority encoder, clearly illustrating the roles of different control signals.

Multiplexers or Selectors

Suppose that each client in an 8-member group is supposed to be able to set up a one-way connection with a receiver side, necessarily through a single line. An 8-input 1-bit *multiplexer* lets us reach this goal as illustrated in Figure 23*a*. Figure 23*b* shows an equivalent switch model for this communication path.

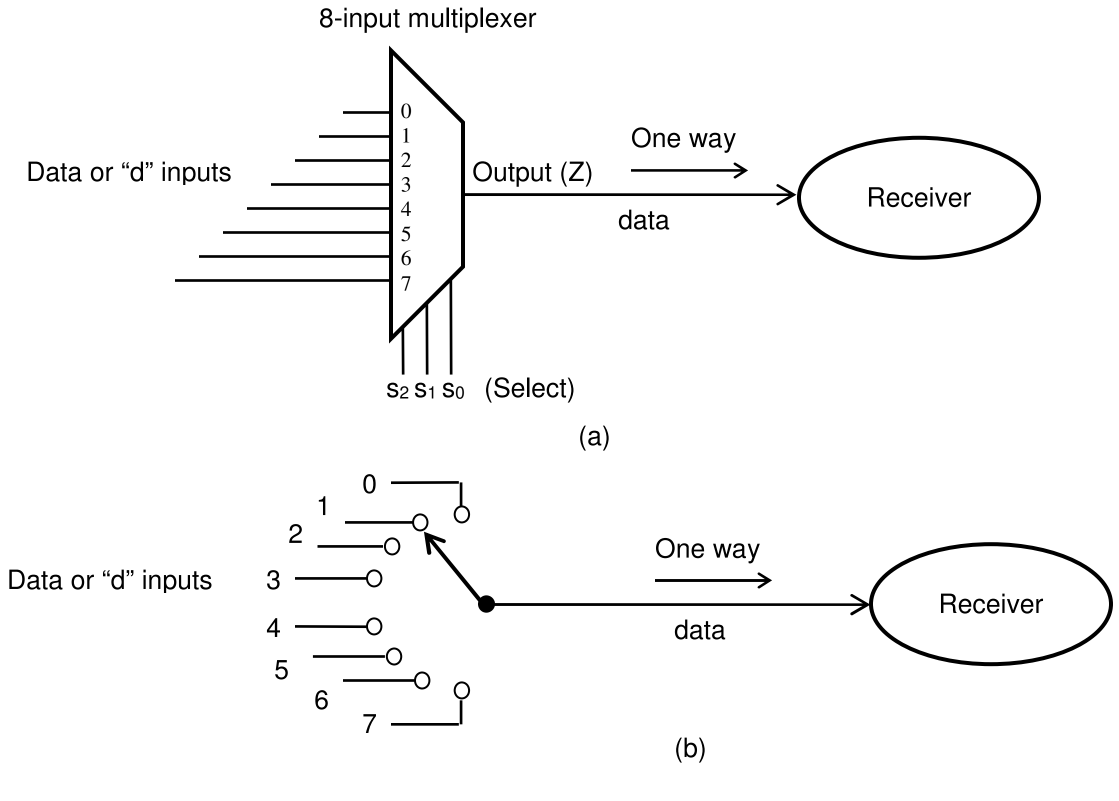


Figure 23. (a) One-way communication link using a mux, (b) switch model of mux

An 8-input 1-bit selector or multiplexer (or mux for short) has 8 data inputs (d_0 to d_7) each 1 bit wide, 3 (= $\log_2 8$) control or select lines (s_2 s_1 s_0), and one 1-bit output (Z), as illustrated in Figure 23*a*. Each bit pattern on the select lines selects or *addresses* one specific input bit. The chosen input bit will *unidirectionally* and *digitally* be connected to the single output bit of the mux; hence the output takes the logic value placed at the selected input, but not vice versa. For example, if $s_2 s_1 s_0 = 011$ in Figure 23*a*, then $Z = d_3$; so that if $d_3 = 1$, then $Z = 1$, and if $d_3 = 0$, then $Z = 0$. This idea can easily be generalized to larger multiplexers with greater number of inputs. In general, a mux with 2^n inputs needs n select lines. As we know from Chapter 01, in order to specify one choice out of 2^n choices, we need $\log_2 2^n = n$ select bits. In other words, the number of choices would be up to $2^{\text{number.of.select.bits}}$.

Figure 24*a*, Figure 24*b* and Figure 24*c* show a symbol, a compressed truth table and a logic circuit, respectively, for a 4-input 1-bit mux. As an example, row 0 in this compressed truth table reads “if s1 s0 = 00, then output Z becomes equal to input d0 (i.e., d0 is copied to Z)”, and row 1 reads “if s1 s0 = 01, then output Z becomes equal to input d1 (i.e., d1 is copied to Z)”, and so on. In this way, the compressed truth table helps us reach the following logic expression for Z:

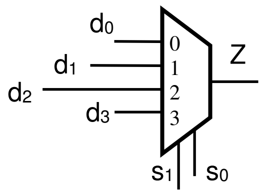
$$Z = s1's0'd0 + s1's0d1 + s1s0'd2 + s1s0d3$$

According to this expression, if s1s0 = 00 (or s1’s0’ = 11), then each of the last three product terms (s1’s0d1, s1s0’d2, s1s0d3) is necessarily 0; therefore Z = s1’s0’d0 = d0. And this is exactly what the truth table reads: “if s1 s0 = 00, output Z equals input d0 (i.e., d0 is copied to Z)”. In a similar way the logic expression can also be verified for other combinations of s1 and s0, i.e., s1s0 = 01, s1s0 = 10 and s1s0 = 11.

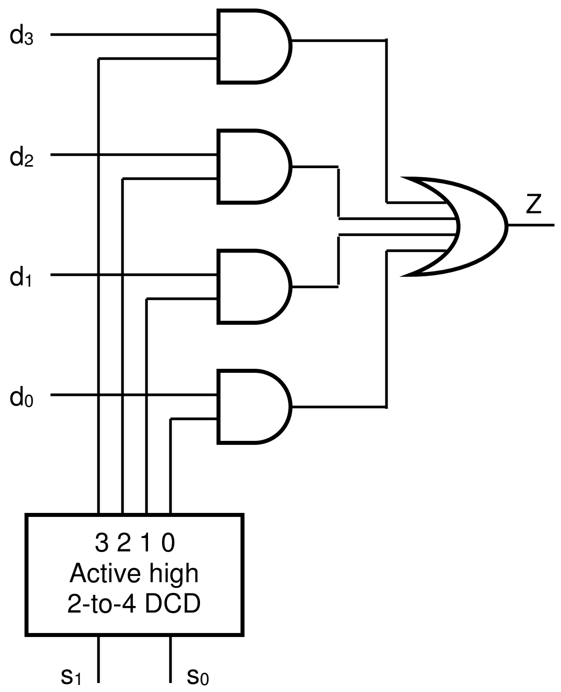
In Figure 24*c* a 2-to-4 binary decoder *opens* exactly one of the four 2-input AND gates at-a-time, letting the corresponding data (d_3, d_2, d_1, or d_0) pass through the open gate (and then the final OR gate) to eventually reach the output, Z. This design illustrates another outstanding application for binary decoders.

Row	s1 s0	Z
0	0 0	d0
1	0 1	d1
2	1 0	d2
3	1 1	d3

(b)



(a)

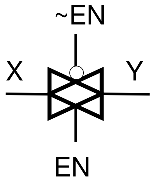


(c)

Figure 24. A four-input 1-bit mux: (a) symbol, (b) truth table, (c) logic circuit

Three-state Devices

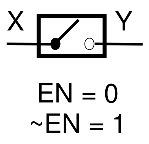
A mux may also be realized with special devices called *transmission gates*. A transmission gate is an *analog* switch, hence the electronic counterpart of a mechanical switch with one ON and one OFF state. A logic symbol and the behavior of such a gate are depicted in Figure 25*a* and Figure 25*b*, respectively. As shown in Figure 25*a* this gate has two data terminals, X and Y, and two normally-complementary enable inputs called EN and ~EN. When EN = 1 and ~EN = 0, then X and Y will be connected to each other in a *bidirectional* fashion (switch ON). And when EN = 0 and ~EN = 1, then X and Y will be disconnected from each other (switch OFF), as summarized in Figure 25*b*. To get a better picture, an OFF and an ON which are modeled in Figure 25*c* and Figure 25*d*, respectively.



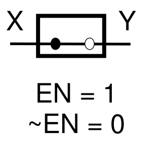
(a)

EN	~EN	Gate's behavior
0	1	Y disconnected from X (switch OFF)
1	0	Y bidirectionally connected to X (switch ON)

(b)



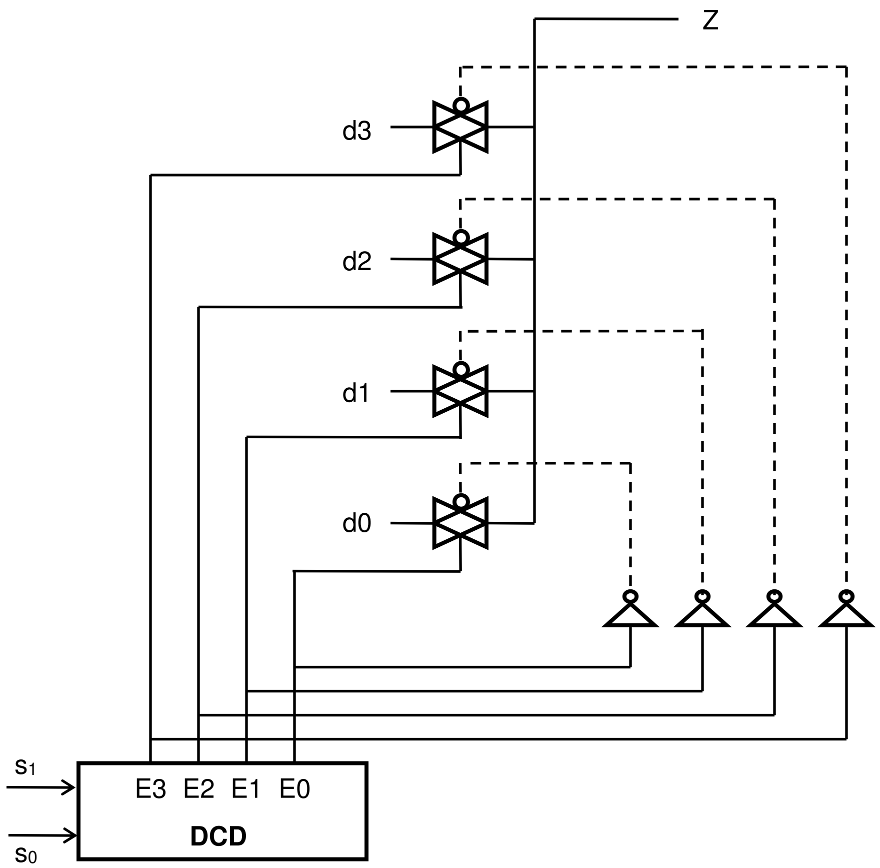
(c)



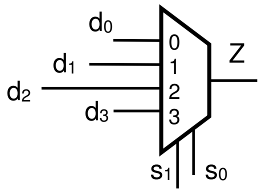
(d)

Figure 25. A transmission gate: (a) symbol, (b) behavior, (c) switch OFF, (d) switch ON

Unlike the outputs of traditional gates, the data terminals of two or more transmission gates could directly be connected to each other (to realize a multiplexer) if at most one transmission gate was enabled at-a-time. A 4-input (transmission-gate-based) single-bit mux is illustrated in Figure 26*a*, and the logic symbol of this size mux is shown again in Figure 26*b*.



(a)



(b)

Figure 26. A 4-input single-bit transmission-gate-based mux: (a) logic circuit, (b) logic symbol

As illustrated in Figure 26*a*, four Y terminals of four transmission gates are tied to each other to form output Z of the mux; now the four X inputs become the four data inputs (d3, d2, d1 and d0) of the mux. On the other hand, the four EN inputs of the four transmission gates are driven by a 2-to-4 active-high binary decoder. Also four inverters driven by the same decoder are used to drive the ~EN inputs of the four transmission gates as shown in Figure 26*a*; so that ~EN of each gate is always the complement of EN of the same gate. Therefore, at each specific instant of time only one transmission gate receives asserted enable inputs to turn ON, letting output Z take the (analog) value of the d input, which has been addressed by the select signals applied to the decoder. As an example, suppose that in Figure 26*a* s1s0 = 01. These select signals will be decoded into E3E2E1E0 = 0010, resulting in one ON and three OFF transmission gates that eventually let Z get connected to d1 as shown in Figure 27. ◊

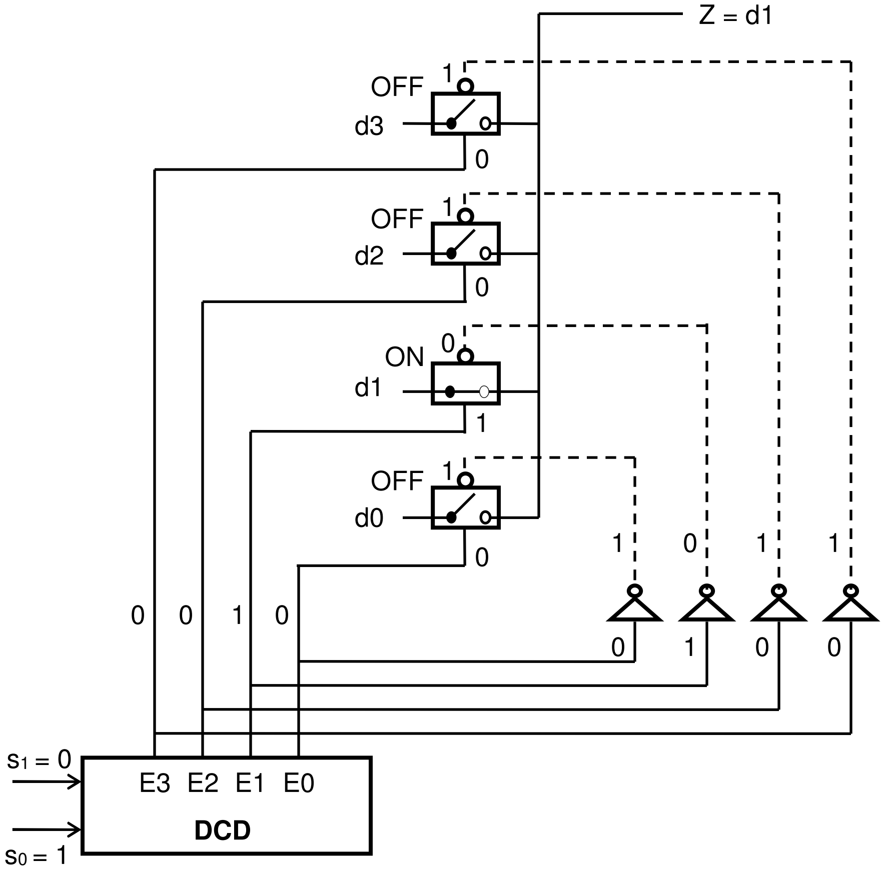
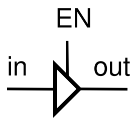


Figure 27. Switch models for 4 transmission gates in 4-input mux when s1s0 = 01

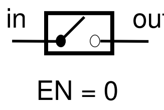
A similar concept is used in a class of gates called *three-state buffers*. A logic symbol and the behavior of such a buffer are shown in Figure 28*a* and Figure 28*b*, respectively. As shown in Figure 28*a*, this gate has a data input, in, a data output, out, and one enable input, EN. When EN = 1, then X and Y will be connected to each other, but unlike a transmission gate this connection is 1) digital and 2) unidirectional, from in to out, only. On the other hand, when EN = 0, then in and out will be disconnected from each other, as summarized in Figure 28*b*. A *disabled* and an *enabled* buffer are modeled in Figure 28*c* and Figure 28*d*, respectively. The buffer shown in Figure 28*a* has an active-high enable input and a non-inverted output. But three-state buffers are also available with active-low enable inputs and/or inverted outputs. Unlike transmission gates, three-state buffers (similar to logic gates) are able to restore signals; but this topic is beyond the scope of this book.



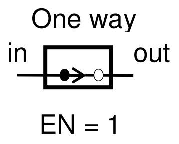
(a)

EN	Buffer's behavior
0	out disconnected from in
1	out connected to in

(b)



(c)



(d)

Figure 28. A three-state buffer: (a) symbol, (b) truth table, (c) buffer disabled, (d) buffer enabled

Expansion of Muxs

Muxs may be expanded (resulting in larger muxs) in two dimensions: 1) width (or number of bits) of each input and 2) number of inputs (or choices).

Figure 29*a* shows the mux expansion in the first dimension, where two 2-input 1-bit muxs are used to reach a 2-input 2-bit mux. The symbol of the resulting mux is shown in Figure 29*b*. If the s input is 1, then data input A (comprised of two bits, a1 and a0) will be chosen, otherwise data input B (made up of bits b1 and b0) will be selected. The same concept can be generalized to obtain larger muxs. In general, we need n 2-input single-bit muxs to obtain a 2-input n-bit mux. When muxs are expanded in this dimension, the number of select lines does not change because the number of choices has not changed.

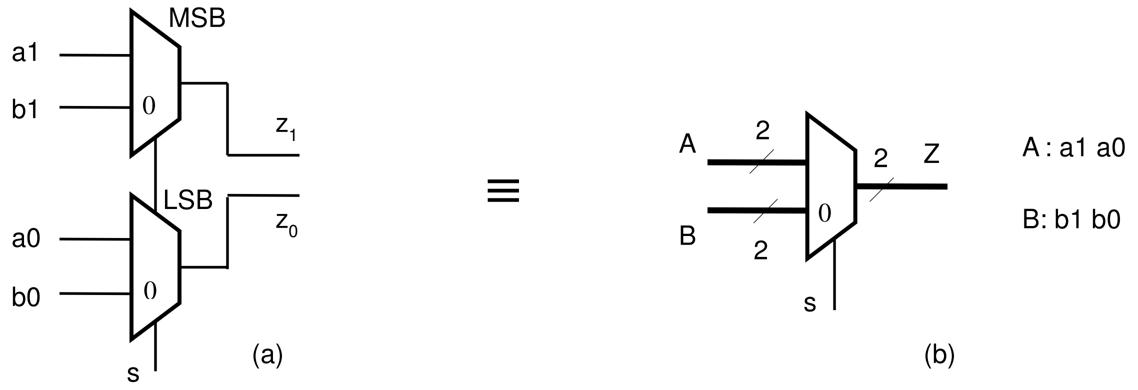
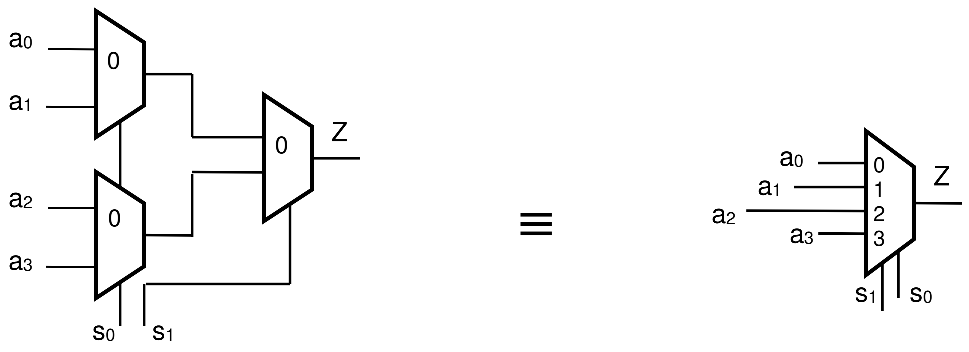


Figure 29. A 2-input 2-bit mux: (a) built up of 2 two-input single-bit muxs, (b) symbol

Figure 30*a* shows the mux expansion in the second dimension where three 2-input 1-bit muxs (each with one select line) are used to reach a 4-input 1-bit mux, which needs two select lines. (The number of select signals is always the smallest integer equal to or greater than $\log_2 n$, where n is the number of choices.) A logic symbol for the resulting mux is shown in Figure 30*b*. A similar concept may be utilized to realize larger muxs.



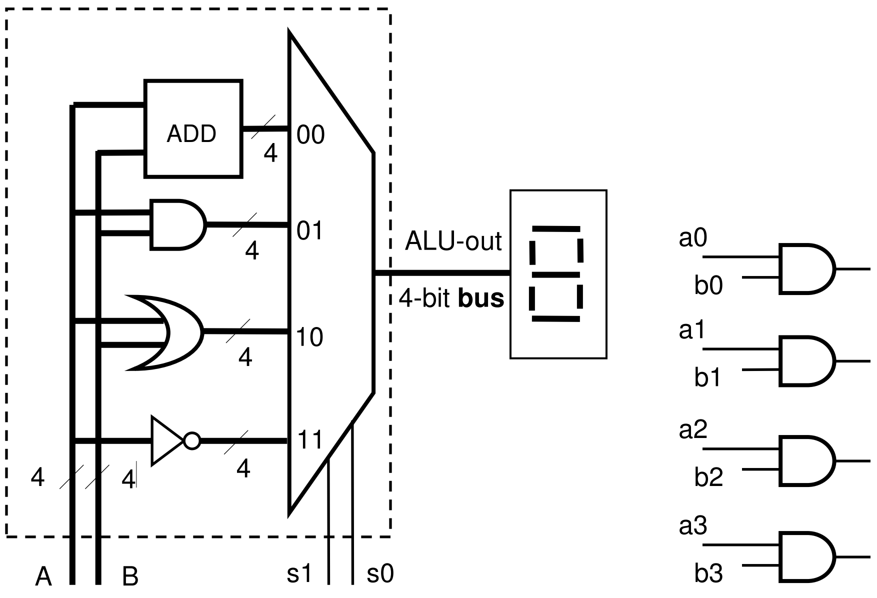
(a) (b)

Figure 30. A 4-input 1-bit mux: (a) built up of 3 two-input one-bit muxs, (b) symbol

A larger mux (in the second dimension) may also be designed from scratch similar to what we did in Figure 24*c* or Figure 26 to obtain a four-bit mux.

Example 6. Arithmetic/logic units (ALUs) are an essential part of microprocessors. A simple ALU is able to perform addition/subtraction and also logical operations, such as AND and OR. In this example we design a four-bit arithmetic/logic unit (ALU) with four basic operations: ADD, AND, OR and NOT, to demonstrate how muxs may be utilized in these widely-used functional units.

Figure 31*a* shows an ALU with 2 four-bit operands, A (a3 a2 a1 a0) and B (b3 b2 b1 b0), and featuring the above four operations. Pay attention to how the logic gates have been abbreviated in this figure. For example, the single AND symbol with two 4-bit inputs and one 4-bit output used in Figure 31*a* stands for a bank of four 2-input AND gates shown in Figure 31*b*. In Figure 31*a* there are four different choices, namely the four-bit output of an adder, the four-bit output of a bank of AND gates, the four-bit output of a bank of OR gates and the four-bit output of a bank of inverters. A four-input four-bit mux selects one choice at a time and places that choice at the output of the mux, which is the output of the ALU as well. Figure 31*c* shows the ALU's function table, where s1 s0 are the mux select inputs and may now be called the *function control lines* of the ALU as well. For example, if s1 s0 = 10, then the output of the OR bank is selected and placed at the output of the ALU. To keep our example simple we assume that the output of the ALU only goes to a seven-segment hexadecimal display (with an appropriate decoder) through a 4-bit *bus*, as shown in Figure 31*a*.



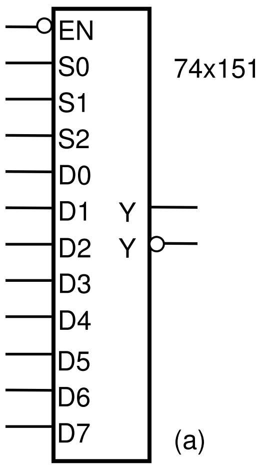
s1 s0	Function
0 0	ADD
0 1	AND
1 0	OR
1 1	NOT

(a) (b) (c)

Figure 31. A simple ALU: (a) logic circuit, (b) bank of four 2-input AND gates, (c) function table

But what is a **bus**? In a simple language a bus is a set of two or more related lines shared by two or more signal *producers* and/or signal *consumers*. For example, in Figure 31*a* there are four signal producers, namely an adder, a bank of four 2-input AND gates, a bank of four 2-input OR gates and a bank of four inverters. These four producers share the same lines (bus), ALU-out, to forward their outputs to a seven-segment display, which is considered the only signal consumer on this bus. ◊

The mux expansion in the second dimension would be easier if the muxs had enable inputs as we see in the 74x151, an 8-input 1-bit mux chip. Figure 32*a* and Figure 32*b* show a logic symbol and a truth table, respectively, for the 74x151. As shown in Figure 32*a,* this chip has one active-low enable input, ~EN, three active-high select inputs, S2, S1 and S0, eight data inputs, D0-D7, one non-inverted output, Y, and one inverted output, ~Y or Y'.



~EN	S2	S1	S0	Y	~Y or Y'
1	x	x	x	0	1
0	0	0	0	D0	D0'
0	0	0	1	D1	D1'
0	0	1	0	D2	D2'
0	0	1	1	D3	D3'
0	1	0	0	D4	D4'
0	1	0	1	D5	D5'
0	1	1	0	D6	D6'
0	1	1	1	D7	D7'

(b)

Figure 32. 74x151: (a) logic symbol, (b) truth table

In Figure 33 two 74x151s are combined to reach a 16-input 1-bit mux with $\log_2 16 = 4$ select lines. Each chip has 8 inputs resulting in a total of 16 inputs. The two outputs from the two chips are merged by a two-input OR gate to generate Xout, the final output of the 16-input mux. On the other hand, S0, S1 and S2 (the select inputs) of both chips are driven by the LSB, 2nd LSB and 3rd LSB, respectively, of the four select inputs of the 16-input mux, as shown in Figure 33. Therefore, for S2 S1 S0 = n ($0 \le n \le 7$), each chip, if enabled, selects its own nth input and applies it to the final OR gate. On the other hand, there is a 1-to-2 decoder comprised of a single inverter shown in Figure 33, which decodes S3, the MSB of the four select inputs of the 16-input mux, and drives the enable inputs (~ENs) of the two chips; so that if S3 = 0, then the upper (most significant) chip is enabled and the lower one is disabled, and similarly if S3 = 1, then the upper chip is disabled and the lower one is enabled. According to the 74x151's truth table shown in Figure 32*b*, output Y of a disabled 74x151 is 0, keeping the OR gate in Figure 33 open for the output of the enabled chip. Therefore, when the lower 74x151 is disabled (by pulling S3 down), the output of the upper chip passes through the OR gate and reaches Xout. Similarly, when S3 = 1 the upper chip is disabled, letting the lower chip forward its output to Xout.

The above concept can be generalized to reach an (8×2^n)-input 1-bit mux using 2^n 74x151 chips. The glue logic that we need to stick these chips together is as follows:

An n-to-2^n active-low decoder is required to select and enable one 74x151 chip at a time: each output out of the 2^n outputs of the decoder drives the ~EN input of exactly one 74x151. Since the total number of data inputs of the new mux is 8×2^n, this mux needs $\log_2 (8 \times 2^n) = (3 + n)$ select lines, out of which the n MSBs drive the decoder, and the 3 LSBs drive the select inputs, S2, S1 and S0, of all 2^n 74x151 chips.

We also need a 2^n-input OR gate to merge the outputs of the 2^n muxs and produce the final output. The binary decoder mentioned above guarantees that at most one mux at a time is enabled; therefore, at any given time $2^n - 1$ inputs to this OR gate are necessarily at logic low, letting the output of the enabled mux reach the output of this OR gate, which is the output of the resulting mux as well.

In a different scenario suppose that we wish to realize an ($8 \times m$)-input mux where m is not a power of 2 anymore. To design this mux we need m 74x151 chips, and an n-to-2^n decoder, where n is the smallest integer that is greater than $\log_2 m$. Now the enable inputs of m 74x151 chips must be driven by m out of

2^n outputs of the decoder. The remaining outputs are not used anymore. And finally we need an m-input OR gate to merge the m outputs of the m 74x151 chips and obtain the final output.

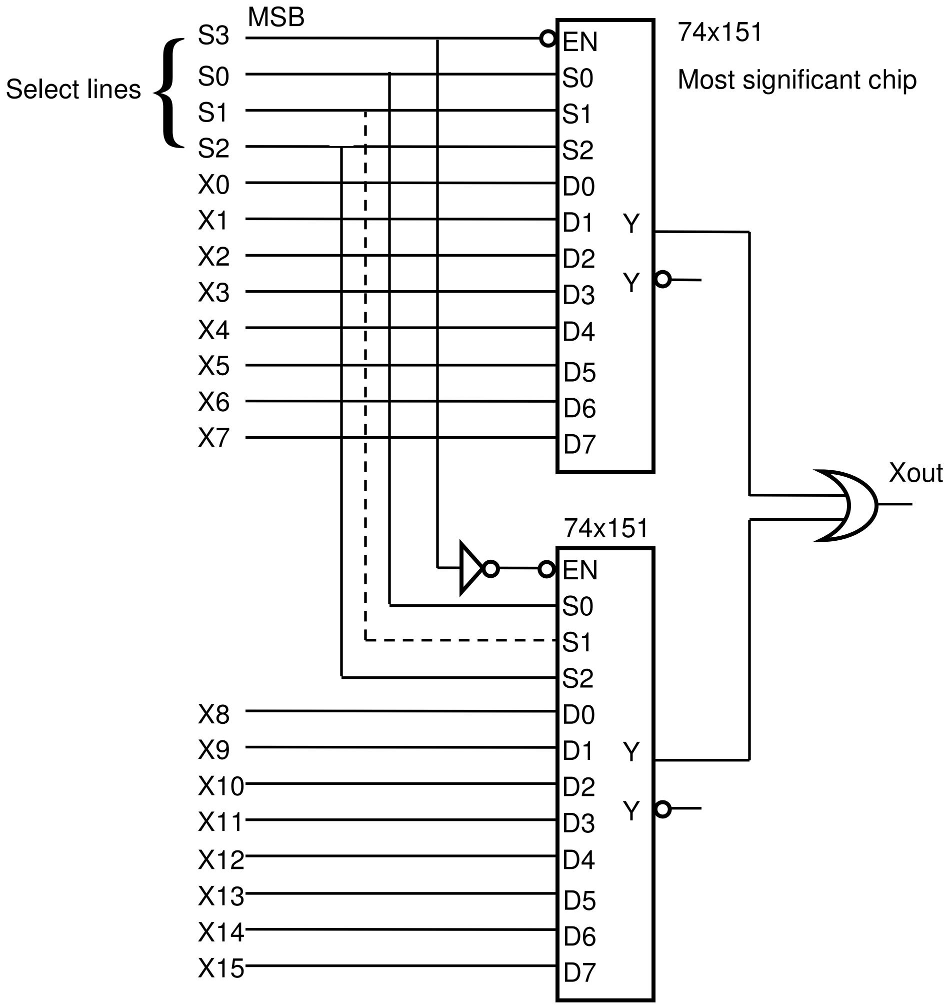


Figure 33. A 16-input 1-bit mux comprised of two 74x151s

Read-Only Memories

An electronic read-only memory (ROM) is a storage device that can be written in (or *programmed*) once, but the stored values may be read out and used forever. Additionally, a ROM retains its contents even if the power goes off. ROMs fall into different categories based on their programming mechanisms, which are out of the scope of this book. What we are interested in here is the *read* mode in which a programmed ROM may be modeled as a matrix or a table of single bits with r rows and c columns, where r and c each are normally a power of 2. Off-the-shelf ROMs come in different sizes. Figure 34 shows an eight-row, four-column (or 8 x 4) ROM, as an example. The content of each bit (in this example) is also shown in this figure. A read operation in a ROM is performed one row at a time: a row address is applied to the ROM and then after some propagation delay time the content of that row appears at the output of the ROM. So, a ROM needs an *address bus* as an input and a *data bus* as an output. Since the number of bits that we need to specify one row out of r rows is $\log_2 r$, the address bus needs to be ($\log_2 r$) bits wide. On the other hand, since one row at a time is read out of the ROM, the data bus has to be c bits wide. Therefore, an 8 x 4 ROM needs a 3-bit address bus ($\log_2 8 = 3$) and a 4-bit data bus as shown in Figure

34. As an example, if address (or input) 101 was applied to this ROM, the output would be 0010, the content of the row corresponding to address 101.

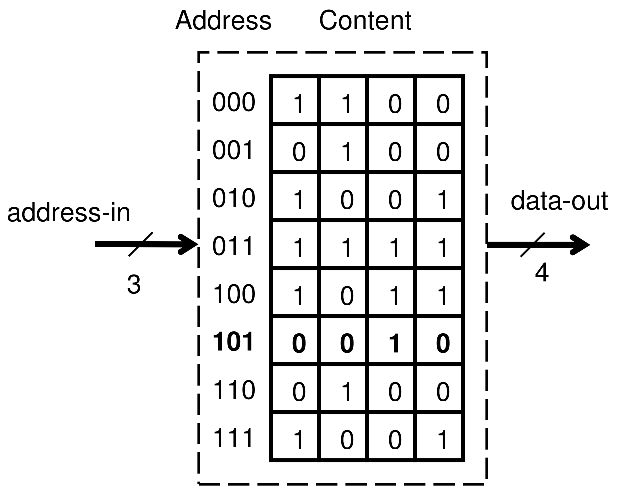
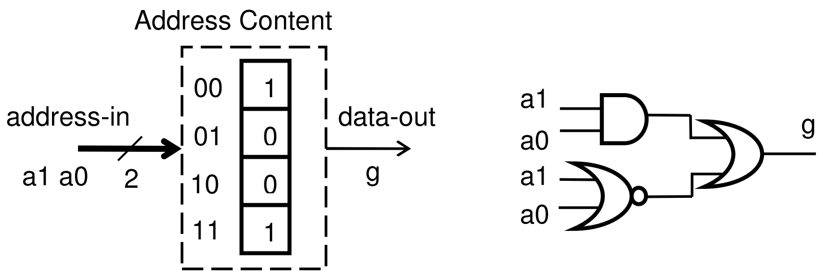


Figure 34. An 8 x 4 programmed ROM with a 3-bit address bus and a 4-bit data bus

Realizing Logic Circuits with ROMs and with Muxs

A 4 x 1 programmed ROM (with address lines a1a0 and output g) is shown in Figure 35*a*. Figure 35*b* and Figure 35*c* show a logic circuit for an XNOR gate (with inputs a1a0 and output g) and its truth table, respectively. It is clearly seen that this circuit and the ROM in Figure 35*a* have identical terminal behaviors; so that we may interpret this circuit (driven by a1a0) as a 4 x 1 ROM (also driven by a1a0) with permanent contents of 1, 0, 0 and 1 located at addresses 00, 01, 10 and 11, respectively, and vice versa.



a1	a0	g
0	0	1
0	1	0
1	0	0
1	1	1

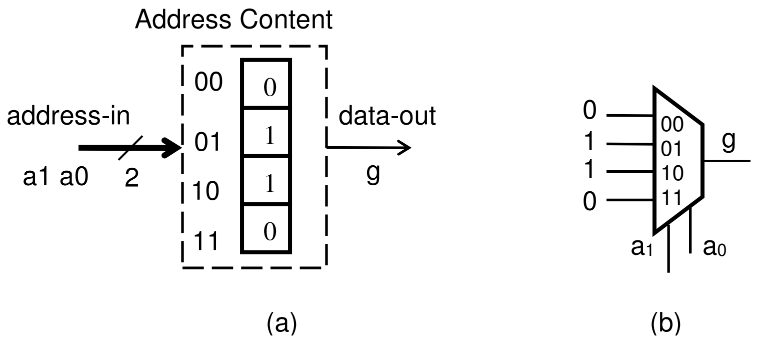
(a) (b) (c)

Figure 35. A second look at logic circuits: (a) four-cell ROM, (b) logic circuit with identical terminal behavior, (c) truth table

Conclusion: Every logic circuit with n inputs and one output is terminally equivalent to a properly-programmed one-column ROM with n address lines. And similarly a one-column programmed ROM is terminally equivalent to a group of logic circuits with the same truth table as the contents of the ROM. More specifically, a one-column ROM with n address lines (or 2^n rows) can be properly programmed to realize any logic function with up to n variables. In general, a 2^n x c ROM can be properly programmed to realize c logic functions each with up to n variables. ◊

A mux with inputs tied to fixed logic levels can interestingly be viewed as a ROM, although not a cost-effective one! If you compare the ROM shown in Figure 36*a*, with the 4-input mux shown in Figure 36*b*, you should realize that they also have identical terminal behaviors. Notice that the inputs of this mux have been fixed at the same logic values that have been stored in the ROM of Figure 36*a*.

Conclusion: To force an r-input single-bit mux to have the same terminal behavior as an r-row single-column programmed ROM, the mux inputs have to be fixed at the same logic levels as stored in the corresponding memory cells of the ROM. Now the select inputs of the mux play the role of the address bus.

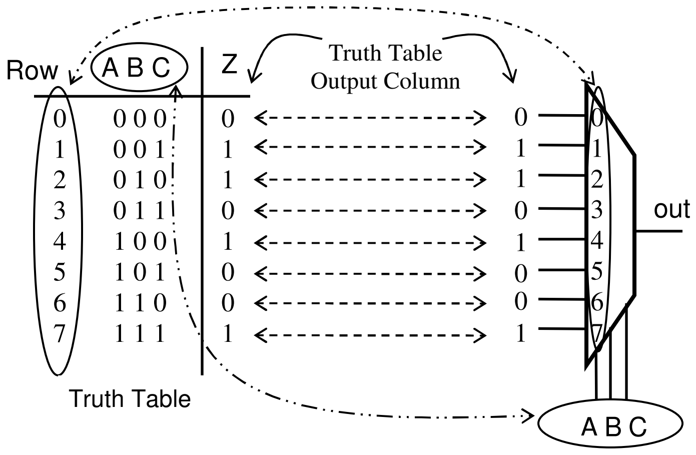


a1	a0	g
0	0	0
0	1	1
1	0	1
1	1	0

(c)

Figure 36. A second look at logic circuits: (a) 4-cell ROM, (b) equivalent mux, (c) truth table

Consider the truth table shown in Figure 36*c*. The output column of this truth table is exactly what has been stored in the ROM shown in Figure 36*a* and also applied to the mux shown in Figure 36*b*. In other words, the truth table in Figure 36*c* has in fact been realized with the programmed ROM illustrated in Figure 36*a* as well as the properly driven mux shown in Figure 36*b*. As another example, Figure 37 shows an 8-row truth table *stored* in an 8-input 1-bit mux to realize the corresponding function.



Row	A B C	Z
0	0 0 0	0
1	0 0 1	1
2	0 1 0	1
3	0 1 1	0
4	1 0 0	1
5	1 0 1	0
6	1 1 0	0
7	1 1 1	1

Figure 37. An 8-input mux used as a ROM to store an 8-row truth table

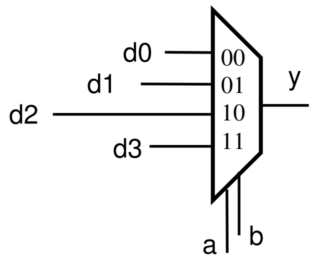
Multiplexers may be used to realize logic functions more efficiently as demonstrated in the following example.

Example 7. Use a four-input single-bit mux to realize the 8-row truth table (with inputs a, b and c and output y) illustrated in Figure 38*a*. Here we assume that input signals are available in non-inverted forms only.

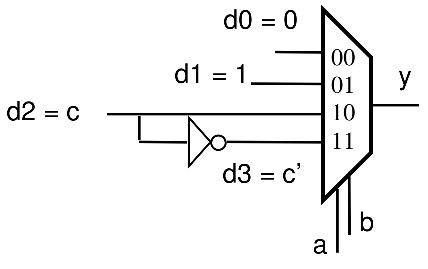
We connect the two control inputs of the mux to two (out of three) input signals, namely a and b, as shown in Figure 38*b*, in which the inputs of this mux have been called d3, d2, d1 and d0. Based on this figure we could say: if a = b = 0, then y = d0, no matter what c is. On the other hand, rows 0 and 1 of the truth table read: if a = b = 0 then y = 0, no matter what c is. This statement is translated into hardware by forcing input d0 to logic 0 as shown in Figure 38*c*. Now similar to the truth table the mux in Figure 38*c* also reads that if a = b = 0, then y = 0, no matter what c is.

Row	a b c	y
0	0 0 0	0
1	0 0 1	0
2	0 1 0	1
3	0 1 1	1
4	1 0 0	0
5	1 0 1	1
6	1 1 0	1
7	1 1 1	0

(a)



(b)



(c)

Figure 38. A 3-variable truth table realized with a 4-input mux: (a) truth table, (b) 4-input mux with control inputs connected to inputs a and b, (c) final logic circuit

Rows 2 and 3 of the truth table read that if a = 0 and b = 1, then y = 1, no matter what c is. So, through a similar reasoning input d1 of the mux is tied to logic 1 as shown in Figure 38*c*.

Rows 4 and 5 of the truth table read that if a = 1 and b = 0, then y = c. Therefore, input d2 of the mux is directly connected to input c, as shown in Figure 38*c*.

Rows 6 and 7 of the truth table read that if a = 1 and b = 1, then y = c', hence, d3 of the mux is tied to c', as illustrated in Figure 38*c*.

In summary, using an n-input single-bit multiplexer (and one inverter) any logic function with up to m variables may be realized, where $m = 1 + \log_2 n$. ◊

Programmable devices in today's technology are far more advanced than the ROMs mentioned above, and fall beyond the scope of this book. These devices supported by state-of-the art computer-aided design (CAD) tools facilitate the realization of logic circuits with different levels of complexities in much more efficient ways than what we studied here using ROMs (or muxs). Interested readers may wish to consult textbooks on *programmable logic design*.

Demultiplexers

A demultiplexer (or demux for short) performs the opposite function of a same-size mux. Remember that a mux has two or more inputs but only a single output; so that one input at a time is selected and placed at the output. A demux, on the other hand, has a single input but two or more outputs; so that one output at a time is selected and then the input is copied to that output. Output selection is done by select signals similar to input selection in muxs. A logic symbol for an 8-output single-bit demux (with 3 (= $\log_2 8$) select inputs, $s_2 s_1 s_0$, and one data input) following an 8-input single-bit mux is shown in Figure 39*a*. At each instant of time the logic value of the single data input (of the demux) is placed at the output (of the demux) specified (or addressed) by the three select bits. For example, if $s_2 s_1 s_0 = 011$, then output 3 takes the value of the input bit.

Figure 39*a* shows how a mux/demux pair facilitates one-way communication from each line in group A (on the mux side) to each line in group B (on the demux side) through a single communication line. Figure 39b illustrates a switch model for this communication link.

Figure 40 shows a compressed truth table for an 8-output demux with outputs Y7, Y6, …Y0 and input In. As shown in this table, In is copied to Y_n ($0 \le n \le 7$) if (binary) number n is placed on the select lines s_2, s_1 and s_0.

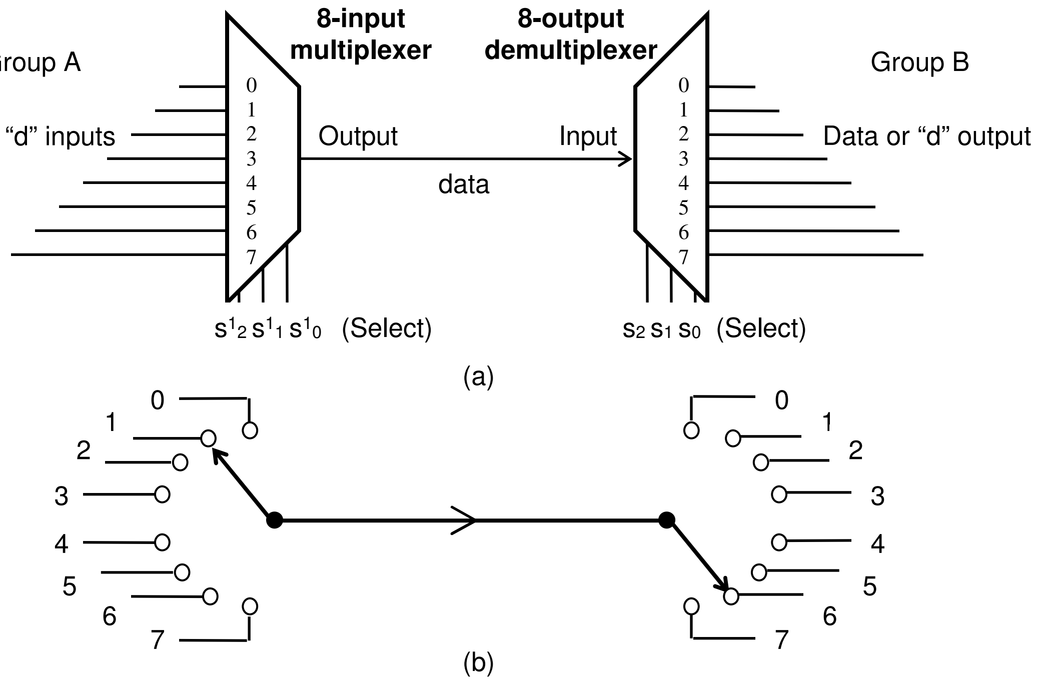


Figure 39. A mux/demux pair: (a) logic symbol, (b) switch model

Row	S2	S1	S0	Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0	0	0	0	In
1	0	0	1	0	0	0	0	0	0	In	0
2	0	1	0	0	0	0	0	0	In	0	0
3	0	1	1	0	0	0	0	In	0	0	0
4	1	0	0	0	0	0	In	0	0	0	0
5	1	0	1	0	0	In	0	0	0	0	0
6	1	1	0	0	In	0	0	0	0	0	0
7	1	1	1	In	0	0	0	0	0	0	0

Figure 40. Compressed truth table for an 8-output demux

Consider the traditional truth table of an 8-output single-bit demux shown in Figure 41, in which variable In has been added to the input column. If In (input line) in this truth table was renamed E (enable input), the table would be identical to the truth table of a 3-to-8 binary decoder shown in Figure 6. Therefore, an n-to-2^n decoder with an enable input is in fact a 2^n-output single-bit demux as well, in which the E input plays the role of the In input, and the A inputs (see Figure 6) play the role of the select inputs.

Row	In	S2	S1	S0	Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0
0	1	0	0	0	0	0	0	0	0	0	0	1
1	1	0	0	1	0	0	0	0	0	0	1	0
2	1	0	1	0	0	0	0	0	0	1	0	0
3	1	0	1	1	0	0	0	0	1	0	0	0
4	1	1	0	0	0	0	0	1	0	0	0	0
5	1	1	0	1	0	0	1	0	0	0	0	0
6	1	1	1	0	0	1	0	0	0	0	0	0
7	1	1	1	1	1	0	0	0	0	0	0	0
8-15	0	x	x	x	0	0	0	0	0	0	0	0

Figure 41. Traditional truth table for an 8-output single-bit demux

Considering the bidirectional nature of transmission gates, a transmission-gate-based mux (in which the output line is labeled "input" and the input lines are labeled "outputs") is in fact a demux as well. So, using a transmission-gate-based mux/demux pair, the link shown in Figure 39 would facilitate a *bidirectional* and *analog* communication between each node in group A and each node in group B.

Demuxs can also be expanded in two different dimensions similar to muxs.

Comparators

An n-bit unsigned comparator compares two n-bit inputs as two unsigned numbers and determines whether A < B, A > B or A = B, by asserting the appropriate output bit. Figure 42*a* shows a logic symbol for such a comparator. But we have already seen a comparator in Chapter 5 implicitly: a subtractor is in fact a comparator as well! When the difference of two operands (A – B) generates a borrow, then B > A, otherwise A > B unless the difference is 0, which signifies A = B as shown in Figure 42*b*. Another technique for comparator design is introduced in Appendix C.

An interesting question is what happens if we apply two signed numbers to an unsigned comparator! For positive numbers the comparator of course generates the right response. This is true for negative numbers as well; for example, consider 1001 (= -7) and 1100 (= -4). As two signed numbers, -4 is greater than -7. Now if these two numbers are interpreted as two unsigned numbers, then 1100 (12) is still greater than 1001 (9). In other words, two negative numbers are still compared correctly by an unsigned comparator. This however, is not the case with two different-sign numbers. Consider 1100 (-4) and 0101 (= 5); as two signed numbers 0101 > 1100, however an unsigned comparator believes that 0101 < 1100. This problem can be solved by swapping the sign bits of the two numbers: now 0101 becomes 1101, and 1100 becomes 0100. With this change an unsigned comparator will make the right decision! Swapping the sign bits does not affect same-sign numbers, as they have anyway identical signs. Therefore, an unsigned comparator can be converted to a signed one by swapping the sign bits of the two numbers as illustrated in Figure 43.

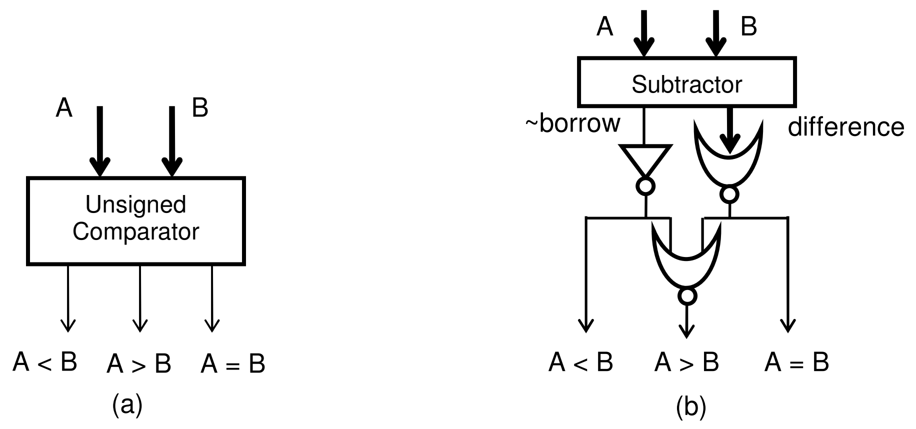


Figure 42. An unsigned comparator: (a) logic symbol, (b) subtractor-based logic circuit

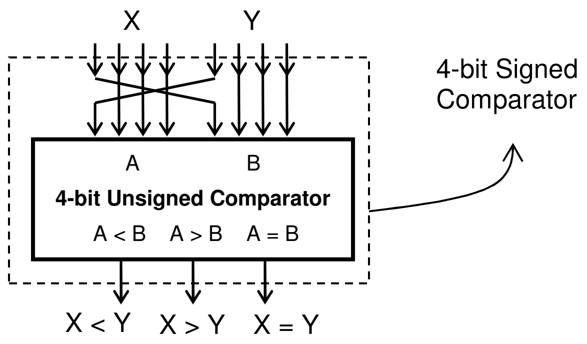


Figure 43. Signed comparison using an unsigned comparator

The above comparators are able to determine whether one number is equal to, less than, or greater than a second number. Let's call them *full* comparators as opposed to *partial* comparators, which are only able to determine whether or not two numbers are equal. The single cell of such a comparator, i.e. an XNOR gate, was introduced in Chapter 2. Figure 44 shows a four-bit partial comparator with two active-high input vectors A (= a_3 a_2 a_1 a_0), B(= b_3 b_2 b_1 b_0) and one active-high output, A.EQ.B. The output is asserted only if A = B, otherwise it would be deasserted. Partial comparators are of course superior in terms of the number of gates and propagation delay time.

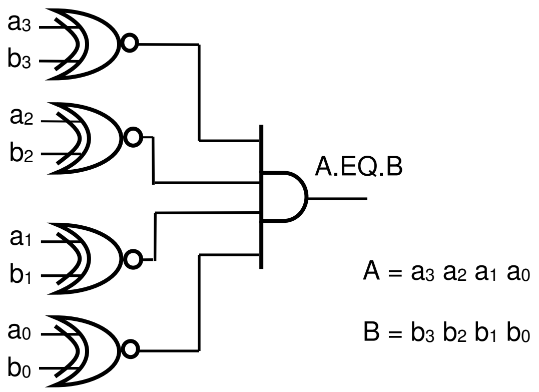


Figure 44. A four-bit partial comparator

A 4-bit partial comparator shown in Figure 44 represents a good example of non-systematic design. In this design methodology a (complicated) problem is decomposed into some smaller, hence easily solvable and/or already solved pieces. Then the solutions to these small problems are stitched together to reach a complete solution for the whole problem. It goes without saying that the designer has to be more creative to efficiently approach a problem non-systematically. For the above comparator we would need a 64-row truth table should we want to use the systematic design procedure; but it is too hard to properly handle such a table, let alone a 2^{64}-row truth table that would have been required if we had decided to apply the systematic design procedure to a 32-bit comparator! In Figure 44 we decomposed the 4-bit comparator into 4 single-bit comparators (with which we are already familiar) and then merged the four outputs through a 4-input AND gate to generate the final output. The same approach works for larger number of inputs. Non-systematic design procedure was also used in some other sections of this chapter such as the design of a 4-input mux shown in Figure 24.

Constant inputs eat the hardware

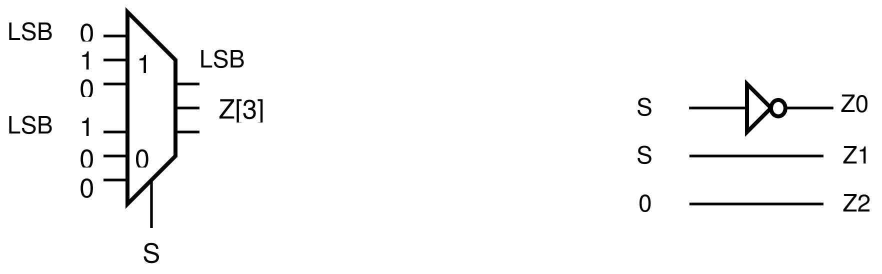
Constant inputs (inputs tied to fixed values, 0 or 1) generally *eat* the hardware resulting in simpler logic circuits. The idea was first examined in Chapter 2, but not under this topic, when a 2-input NOR gate was converted to an inverter by tying one of the inputs to logic 0. In other words, this constant input, 0, simplified a NOR gate to an inverter. In Chapter 5, a half adder was produced by applying a 0 to a full adder. Let's now take a look at another example. By examining the 2-input 3-bit mux shown in Figure 45*a*, the following equations will be derived for the outputs:

Z0 = S'

Z1 = S

Z2 = 0

So, the mux can be replaced with only one inverter as shown in Figure 45*b*!



(a) (b)

Figure 45. (a) 2-input 3-bit mux with constant inputs, (b) simplified logic diagram

Appendix A: A 5-to-32 DCD

Figure 46 shows a 5-to-32 decoder using four 74x138 chips and only one inverter as the glue logic to stick the chips together. The unused G1 from the top (least significant) chip has been fixed at logic 1 and the unused ~G2Bs from the other three chips have been tied to logic 0, in order to deactivate these unused inputs. The five inputs arriving at this 5-to-32 DCD have been called N4 N3 N2 N1 N0, out of which the three LSBs, N2, N1 and N0, drive the C, B and A select inputs, respectively, of all four chips (see also Figure 14*a*). The two most significant inputs, N4 and N3, drive two (out of three) enable inputs of each chip. Pay close attention to how these two enable inputs have been used to decode N4 and N3, hence enable (or select) one chip at-a-time. The three disabled chips at any given time produce 3 x 8 deasserted outputs, while the three least significant input bits, N2, N1 and N0, are decoded by the enabled chip. As an example suppose that N4 N3 N2 N1 N0 = 10 011 are applied to this DCD. Since N4 N3 = 10 chip 2 or the third chip from the top will be enabled, while the other three remain disabled, resulting in the decoded output 1111 1111, 1111 0111, 1111 1111, 1111 1111.

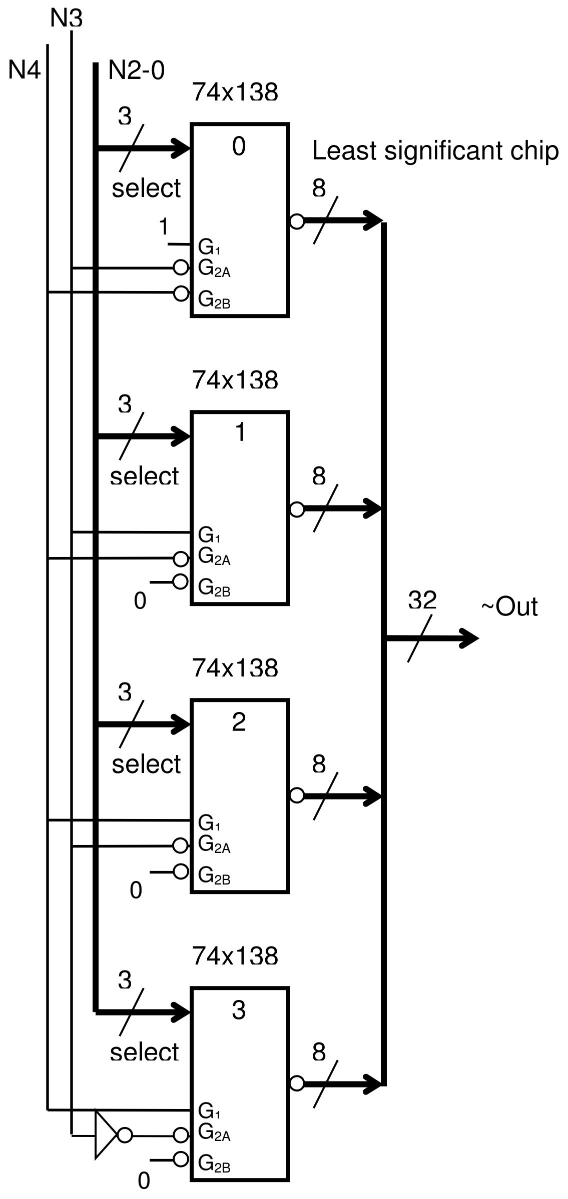


Figure 46. A 5-to-32 DCD using four 74x138s and one inverter

Appendix B: A 32-bit Priority Encoder

Four 74x148 chips are used to design a 32-bit priority encoder. Figure 47 shows the complete design. Let us assume that the chips are located in descending order of priority from top to bottom.

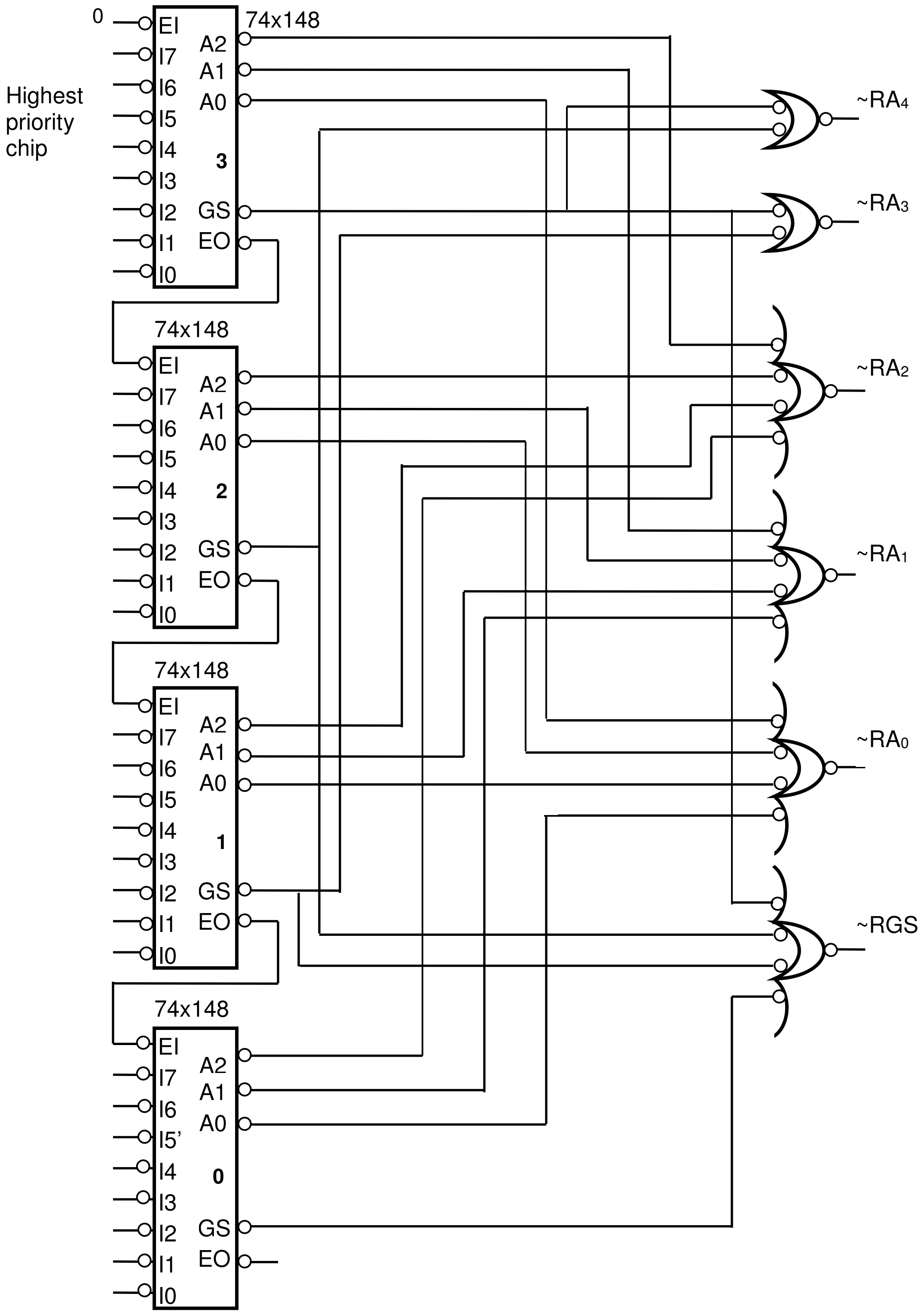


Figure 47. 32-bit priority encoder made up of four 74x148 chips

As shown in Figure 47, the chip with the highest priority (the top one) is always enabled (~EI = 0). Additionally, ~EO of this chip goes to ~EI of the chip with the second highest priority (the 2nd chip from the top). Similarly, ~EO of chip 2 goes to ~EI of chip 1, and ~EO of chip 1 goes to ~EI of chip 0. Considering that ~EO of a 74x148 chip will be asserted only if 1) that chip receives no requests and 2) the chip is enabled (see the last row of the truth table in Figure 22), we come to the conclusion that if an enabled chip receives a request, then all chips (if any) located below this enabled chip (i.e., all chips with lower priorities than this enabled chip has), will be disabled. But a disabled chip has deasserted outputs (see the 1st row of the truth table in Figure 22). On the other hand, an enabled chip with no requests will deassert its outputs as well (see the last row of the truth table in Figure 22).

In order to analyze the circuit in Figure 47 more conveniently I have split it into three subcircuits shown in Figure 48, Figure 49 and Figure 50.

Figure 48 shows how the three least significant output bits (~RA2, ~RA1 and ~RA0) of the 32-bit priority encoder are generated. The outputs of each chip are called ~A2, ~A1 and ~A0. Consider the three 4-input AND gates in this figure: four ~A2s, four ~A1s, and four ~A0s from the four chips go to AND gates 2, 1 and 0, respectively, to generate ~RA2, ~RA1 and ~RA0, respectively. Now consider two different scenarios: 1) No request line is asserted; this will deassert ~A2s, ~A1s, and ~A0s from all four chips, resulting in deasserted ~RA2, ~RA1 and ~RA0. 2) There is at least one request, which means that there is exactly one enabled chip with at least one request. ~A2, ~A1, and ~A0 of this chip will pass through the three AND gates and reach the output lines ~RA2, ~RA1 and ~RA0, respectively, because the other three chips have deasserted ~A2s, ~A1s, and ~A0s. Therefore, at any given time ~A2, ~A1, and ~A0 of the chip with the highest priority among the chips that have received a request, would be copied onto ~RA2, ~RA1 and ~RA0, respectively. And this means that the three LSBs of the final output are generated correctly. The remaining bits (i.e., the two MSBs) are produced in Figure 49.

As we know, the ~GS output of a 74x148 chip will be asserted only if that chip is enabled and also receives a request. This means that at most one output out of four outputs ~GS3, ~GS2 ~GS1, ~GS0 will be asserted at a time. So, what we need here, is a 4-to-2 binary encoder to encode the above ~GS lines and locate the enabled chip that has received at least one request. The two 2-input AND gates in Figure 49*a* comprise this encoder, and the truth table of this encoder is shown in Figure 49*b*. Notice that the no-request scenario is also supported by this design: in case of no requests all ~GS lines will be deasserted resulting in deasserted ~RA4 and ~RA3, as shown in the last row of the truth table in Figure 49*b*.

The last stage in the design of this 32-bit priority encoder is to generate ~RGS, which signifies whether or not at least one request line out of 32 lines has been asserted. To generate this signal, ~GS3, ~GS2, ~GS1 and ~GS0 are simply ANDed as shown in Figure 50.

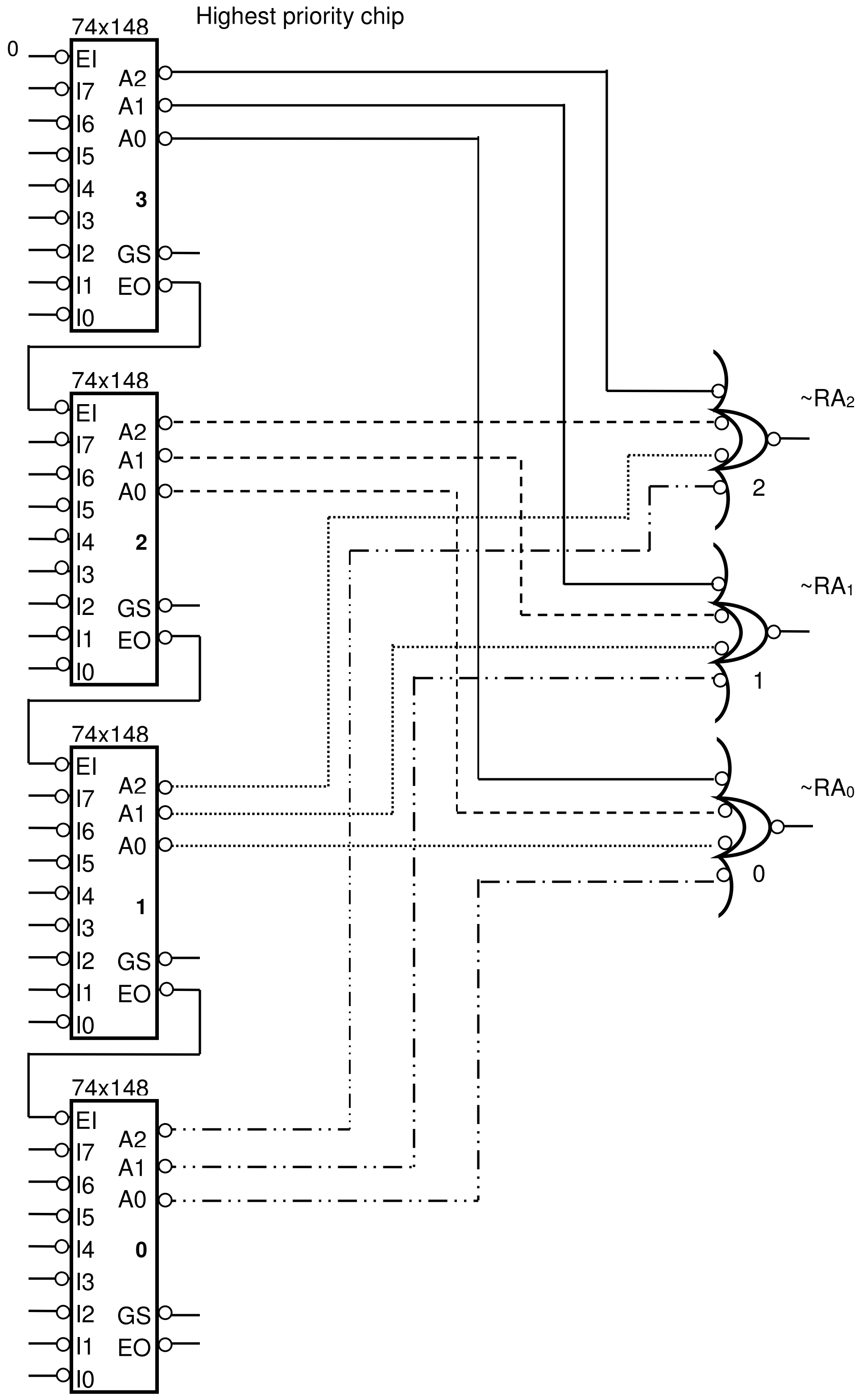
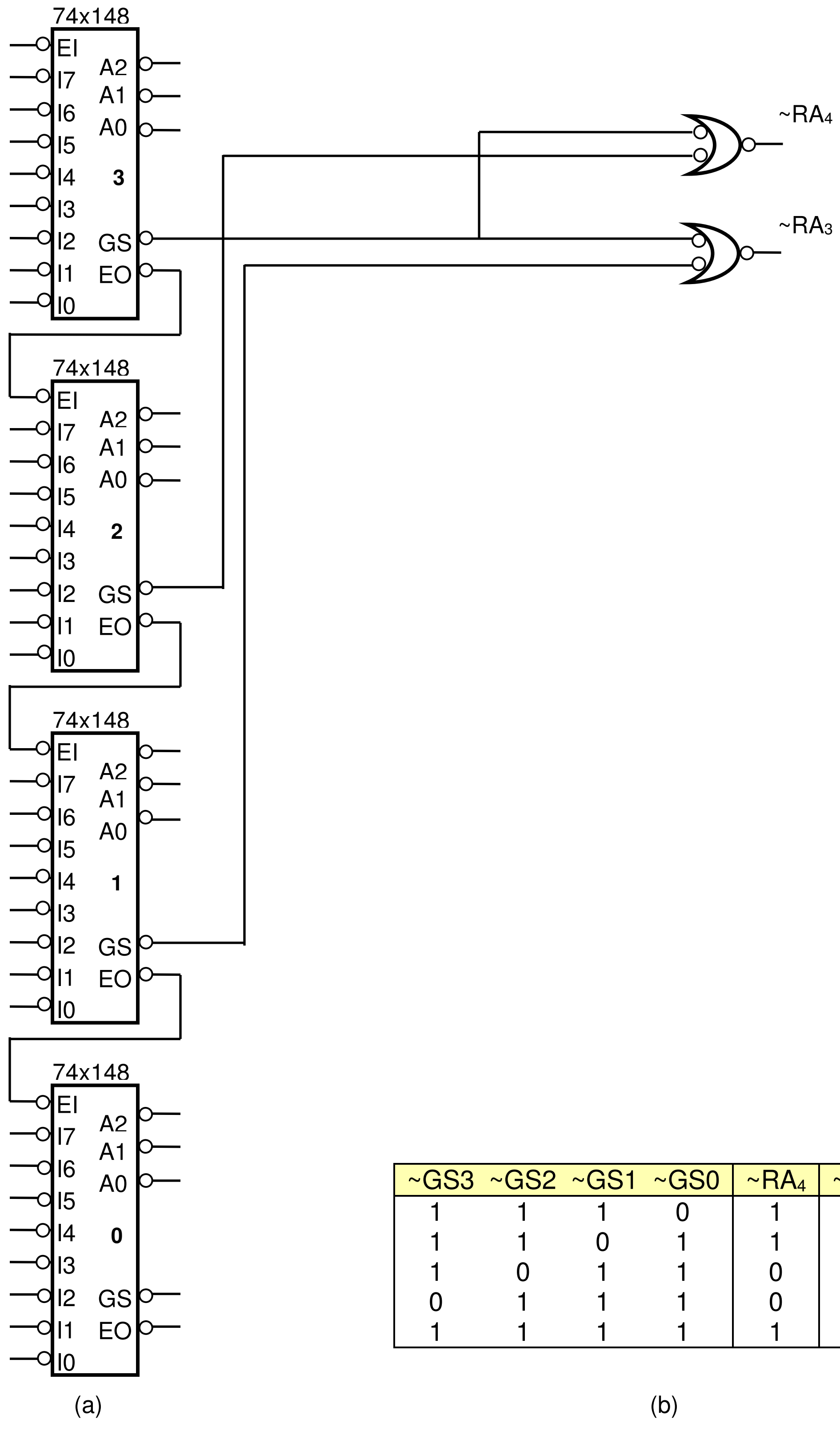


Figure 48. 32-bit priority encoder: three LSBs of output



~GS3	~GS2	~GS1	~GS0	~RA$_4$	~RA$_3$
1	1	1	0	1	1
1	1	0	1	1	0
1	0	1	1	0	1
0	1	1	1	0	0
1	1	1	1	1	1

(b)

Figure 49. Two MSBs of output of 32-bit priority encoder: (a) logic diagram, (b) truth table

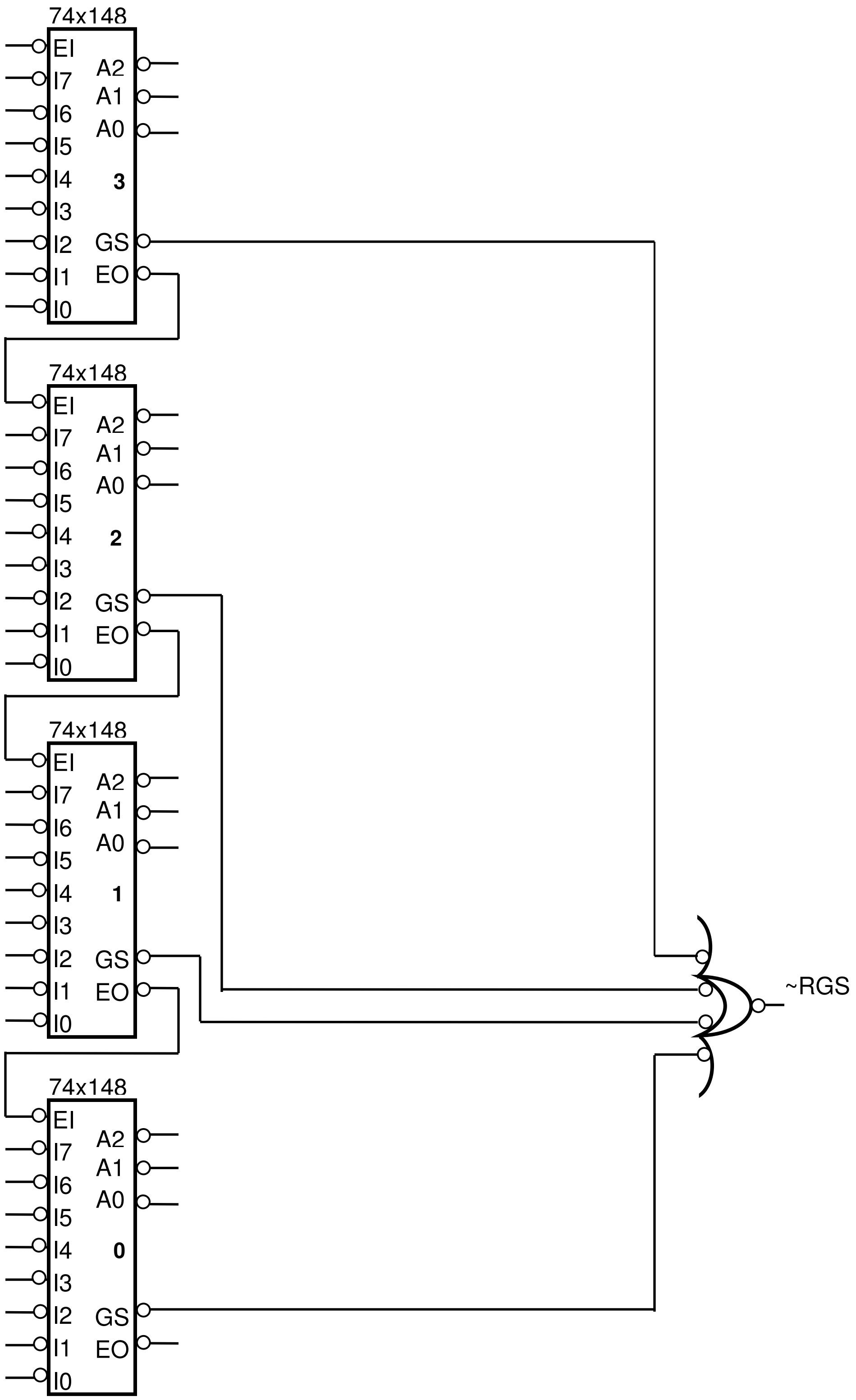


Figure 50. 32-bit priority encoder: GS signal

Appendix C: A Repetitive Approach to Design Comparators

Subtraction-based comparison is not the only method to design comparators. In this appendix a repetitive approach is introduced to design comparators without subtraction. But keep in mind that microprocessors, today's mostly used digital systems, use subtractors as comparators as well because a microprocessor does need a subtractor to do subtraction.

Remember our traditional method to compare two numbers: starting form the left (MSB), we compare two same-weight bits (from the two numbers) one at a time, until the first mismatch is reached, which signifies the end of comparison. The two numbers are equal if no mismatch occurs.

As an example consider two 5-bit numbers 10101 and 10110:

	4 3 2 1 0 ⟵ bit positions or columns
Operand A	1 0 1 0 1
Operand B	1 0 1 1 0

Column 4: $A_4 = B_4$, so no final conclusion may be reached. Tell the next column that "A = B so far".

Column 3 is told that "A = B so far". Additionally, this column realizes that $A_3 = B_3$, so no final conclusion may be reached. Tell the next column that "A = B so far".

Column 2 is told that "A = B so far". Additionally, this column realizes that $A_2 = B_2$, so no final conclusion may be reached. Tell the next column that "A = B so far".

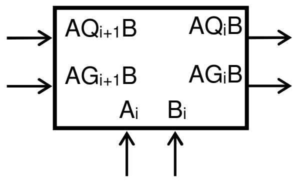
Column 1 is told that "A = B so far"; however, this column realizes that $A_1 < B_1$, so the conclusion is: $A < B$. And this is the final decision, as the following column cannot change it. Tell the next column that $A < B$.

Column 0 is told that "$A < B$"; since this is the final decision, the same message, $A < B$, must be passed on to the outside world.

If the message "A = B so far" had remained valid until all the columns had been processed, then the final decision would have been "A = B".

The above procedure can easily be implemented using a repetitive approach. What we need is to realize one cell of the comparison chain, and then cascade as many cells as we need to reach the required size for the final comparator. Figure 51*a* shows the interface of one individual cell: cell i receives the ith bits, A_i and B_i, from the two numbers A and B, respectively. This cell also receives inputs $AG_{i+1}B$ and $AQ_{i+1}B$ from the previous (more significant) cell, cell i+1, and generates outputs AG_iB and AQ_iB for the following (less significant) cell, cell i – 1. Notice that "AGB = 1" means "A > B", and "AQB = 1" means "A = B so far". These two signals cannot be asserted at the same time; additionally "A < B" is signified by deasserting both AGB and AQB. Based on the above reasoning the logical statements representing the two outputs of cell i are determined as shown in Figure 51*b*. These statements may then be complied into the following logical expressions:

$AG_iB = AG_{i+1}B + AQ_{i+1}B \,.\, A_i \,.\, B'_i.$
$AQ_iB = AQ_{i+1}B \,.\, (A_i \text{ XNOR } B_i)$



(a)

$AG_iB = 1$ iff $(AG_{i+1}B = 1)$ or $(AQ_{i+1}B = 1$ & $A_i > B_i)$

$AQ_iB = 1$ iff $(AQ_{i+1}B = 1)$ & $A_i = B_i$.

(b)

Figure 51. Building block of a repetitive comparator

Figure 52 illustrates a four-bit repetitive comparator comprised of four identical cells shown in Figure 51*a*. Inputs AGB and AQB of the most significant cell must be tied to 0 and 1, respectively, because no partial comparison has been made yet. Also the final outcome of the whole comparison is taken from the least significant cell in the chain, as shown in Figure 52.

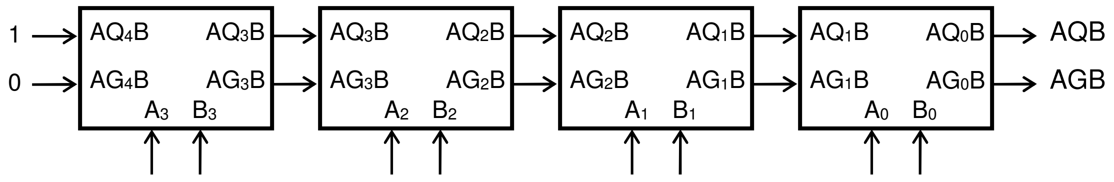


Figure 52. A four-bit repetitive comparator

Although this approach benefits from identical cells, hence ease of design and verification, the resulting comparator has a poor performance for long chains because of the dependence between every two consecutive cells. This is the same problem that ripple adders suffer from. A fast adder/subtractor, hence a fast comparator may be obtained by using other algorithms such as the *carry-look ahead*. Interested readers in this and other techniques are referred to text books on *computer arithmetic*.