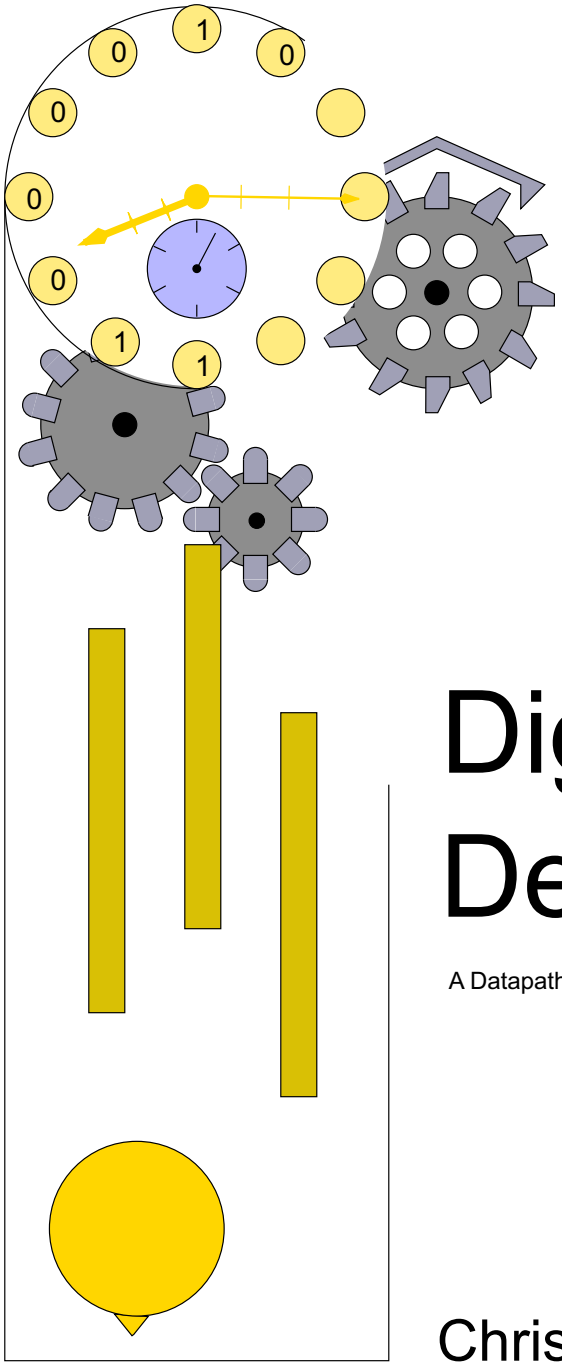


Digital Design

A Datapath and Control Approach

Chris Coulston

October 2, 2024

This document was prepared with LaTeX.

Contents

List of Processes

Digital Design Body Of Knowledge

The focus of this text is very much on the datapath and control approach. To achieve this end in a single semester, sacrafices in coverage must be made.

- Digital Design
  - Numbering Systems
    - Positional Numbering Systems
      - Base 10 - Decimal
      - Base 2 - Binary
      - Base 16 - Hexadecimal
    - Conversion Between Bases
    - Word Size
    - 2's Complement
  - Representation of Logical Function
    - Elementary Logical Functions
    - Word Statement
    - Truth Table
    - Symbolic
    - Circuit Diagram
    - Hardware Description Languages
    - Conversion Between Representations
    - Timing Diagrams
  - Logic Minimization
    - Karnaugh Maps (Kmaps)
    - Kmaps for circuits with multiple outputs
    - Kmaps to find POSmin
    - Logic Minimization Software
  - Combination Logic Building Blocks
    - Decoder
    - Multiplexers
    - Adders
    - Comparators
    - Three-State Buffers
    - Wire Logic
    - Combination
      - Arithmetic Statements
      - Conditional Statements
  - Primitive Sequential Circuits
    - Characteristics
    - Timing
    - Asynchronous set/reset
  - Sequential Logic Building Blocks

Chapter 1

Combinational Logic Building Blocks

In Chapter 1, Figure **??**, a digital system was defined as a box that transforms binary inputs to binary output. This definition of a digital system was sufficient to introduce a wide variety of concepts but is a handicap, now. Following is a more detailed definition of a digital system.

> A digital system transforms the data inputs into data outputs. The transformation performed by the digital system is specified by the control inputs. The status outputs indicate any exceptional events occurring during the processing of the data.

Figure 1.1 shows a digital system with these four types of input and output, namely, data input, control input, data output, status output. This classification of inputs and outputs aids in the construction of complex digital systems using the datapath and control methodology.

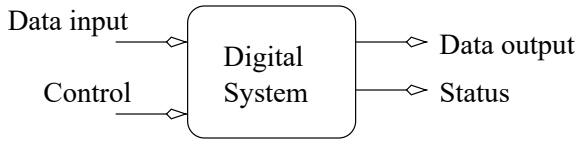


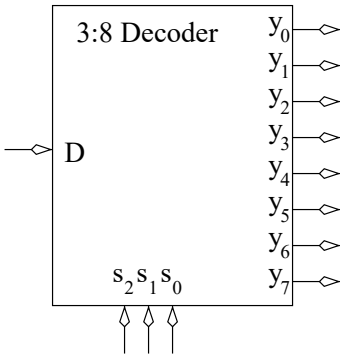
Figure 1.1: A modified diagram of a digital system showing the classification of the inputs and outputs.

Some basic building blocks of digital systems are introduced. While a wide variety of blocks can be considered, those proven to be most useful in the construction of digital circuits are presented. If the right block for a particular task does not exist, create a new block using the methods presented in the last chapter. First in the examination of basic building blocks is a device that routes one bit of data to one of several outputs based on an address.

1.1 Decoder

Nomenclature:	N:M decoder
Data Input:	1-bit D
Data Output:	M-bit vector $y = y_{M-1} \ldots y_1 y_0$
Control:	N-bit vector $s = s_{N-1} \ldots s_1 s_0$
Status:	none
Behavior:	$y_s = D$ all other outputs equal 0

To understand this definition, examine an instance of a 3:8 decoder shown at left in Figure 1.2. Normally, the arrows indicating the direction of the information flow in to and out from the decoder are not drawn. To understand the behavior of the decoder, its truth table is shown to the right of Figure 1.2. Due to space constraints, only part of the entire truth table is shown.



s_2	s_1	s_0	D	y_0	y_1	y_2
0	0	0	0	0	0	0
0	0	0	1	1	0	0
0	0	1	0	0	0	0
0	0	1	1	0	1	0
0	1	0	0	0	0	0
0	1	0	1	0	0	1
0	1	1	0	0	0	0
0	1	1	1	0	0	0
1	0	0	0	0	0	0
1	0	0	1	0	0	0

Figure 1.2: A 3:8 decoder (left) and part of its truth table (right).

From the behavior listed in the description of the decoder, see that the i^{th} output equals the data input, where i is the binary code of the select inputs. In other words, when $S = s_2 s_1 s_0 = 011_2 = 3_{10}$ and $D = 1$, then $y_7 y_6 y_5 y_4 y_3 y_2 y_1 y_0 = 00001000$. If $S = s_2 s_1 s_0 = 011_2 = 3_{10}$ and $D = 0$, then $y_7 y_6 y_5 y_4 y_3 y_2 y_1 y_0 = 00000000$.

The utility of this second case might be questionable because all the outputs are the same and consequently one cannot "see" where the output is being routed. The resolution to this dilemma requires considering the outputs through time. A decoder is a box which sends a "stream" of bits to some destination determined by the select lines. If the destination knows that it is receiving the stream, then it will be expecting both 1s and 0s through time.

The internal organization of a 3:8 decoder must process its four bits of input, consisting of a 3-bit select and a 1-bit data input, and eight bits of outputs. In the previous chapter, each bit of the output could be solved independently of the others. Hence, let's examine the y_0 output first. Examination of the truth table and the behavior of the decoder shows y_0 only equals 1 when $(s_2, s_1, s_0) = (0, 0, 0)$ and $D = 1$. Borrowing the minterm trick from page **??**, $y_0 = s_2' s_1' s_0' D$ results. Every other output shares the characteristic that its output is equal to 1 for a single input. Thus, each output is represented by a minterm as shown in Figure 1.3.

In some digital applications, the need arises to build a larger decoder from multiple smaller decoders. This is done by fanning-out the data input in a tree-like structure. The following 4:16 decoder shows how a larger decoder is built from several smaller 2:4 decoders.

In order to accommodate the 16 outputs, four 2:4 decoders are stacked on top of one another as shown in Figure 1.4. These four 2:4 decoders have a total of four bits of input. These four

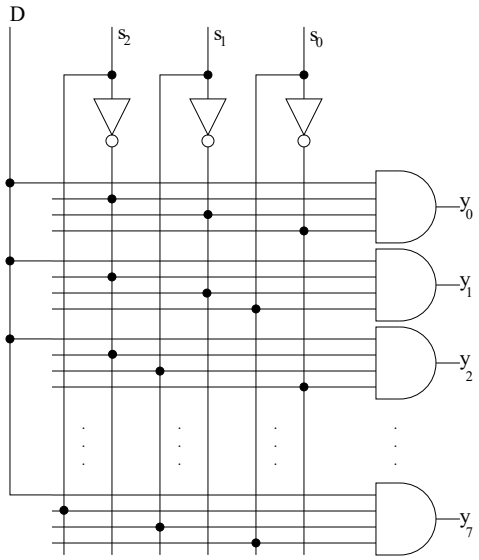


Figure 1.3: The internal organization of a 3:8 decoder.

bits are sourced by the output of a single 2:4 decoder. The single data input of this decoder is the input of the overall 4:16 decoder.

Having organized the structure of the 2:4 decoders, all that remains is to route the select lines. The outputs of the decoder labeled **3** in Figure 1.4 corresponds to outputs $y_{15}y_{14}y_{13}y_{12}$ of the 4:16 decoder. Each of these outputs requires four bits of select with the form $s_3s_2s_1s_0 = 11xx$. Hence, $s_3s_2 = 11$ must route the data input, D, to decoder **3**. Hence, s_3s_2 must be the select to the first level decoder in Figure 1.4. A similar argument for the 2:4 decoders labeled **0,1,2** reinforces the fact that s_3s_2 must be the select to the first level decoder.

Data routed to output y_{12} has $s_3s_2s_1s_0 = 1100$. Thus, routing at the second level of 2:4 decoders seems to be controlled by s_1s_0. Examining all the other outputs reinforces this assumption for all the outputs.

1.2 Multiplexer

A multiplexer, often referred to as a mux, is data routing device which behaves exactly opposite of a decoder. Its structure and behavior is defined in the following table.

Nomenclature:	N:1 multiplexer
Data Input:	M-bit vector $y = y_{M-1} \dots y_1y_0$
Data Output:	1-bit F
Control:	$log_2(N)$-bit vector $s = s_{log_2(N)} \dots s_1s_0$
Status:	none
Behavior:	$F = y_s$

To understand this definition, examine an instance of an 8:1 mux shown on the left in Figure 1.5. From the behavior listed in the description of the multiplexer, the i^{th} data input is routed to the data output where the binary code of the select inputs is i. For example, if $S = s_2s_1s_0 = 101_2 = 5_{10}$ and $y_7y_6y_5y_4y_3y_2y_1y_0 = 00100000$, then $F = 1$.

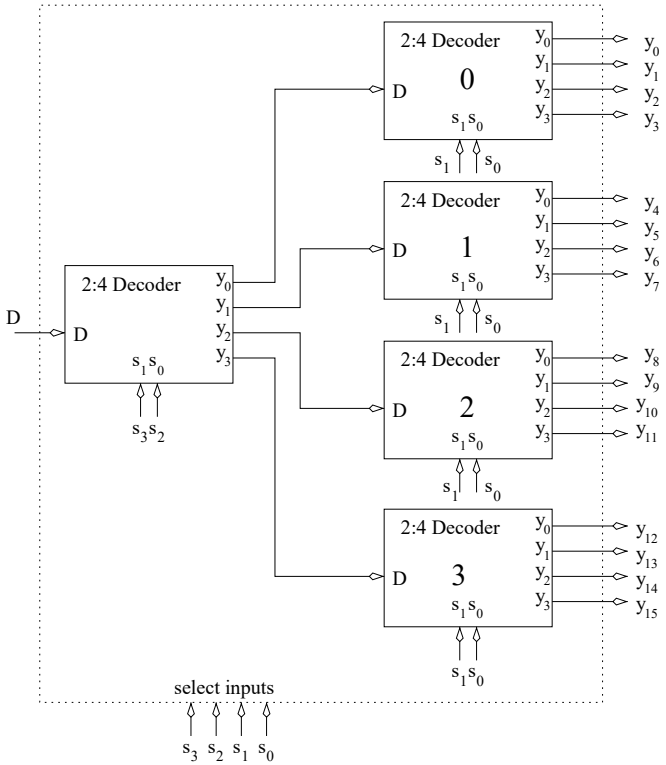
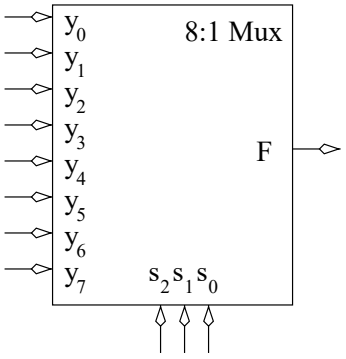


Figure 1.4: A 4:16 decoder built from 2:4 decoders.



s_2	s_1	s_0	F
0	0	0	y_0
0	0	1	y_1
0	1	0	y_2
0	1	1	y_3
1	0	0	y_4
1	0	1	y_5
1	1	0	y_6
1	1	1	y_7

Figure 1.5: An 8:1 mux (left) and its truth table (right).

The unusual form of the truth table shown in Figure 1.5 results from the fact that an 8:1 mux has a total of 11 inputs. There are a total of 2^{11} rows in this truth table making it infeasible to list every combination of the inputs. Consequently, in order to build an 8:1 mux, the structure of the truth table must be determined without listing every combination of inputs. Observe the y_0 input is routed to the output when $s_2 s_1 s_0 = 000$. Consequently, $F = s_2' s_1' s_0' y_0$ for this input. Each of the other outputs has a similar minterm form. Since only one of these "minterms" can equal 1 for a particular select input, the minterms are ORed together to form the output. The resulting internal organization of an 8:1 mux is shown in

Figure 1.6.

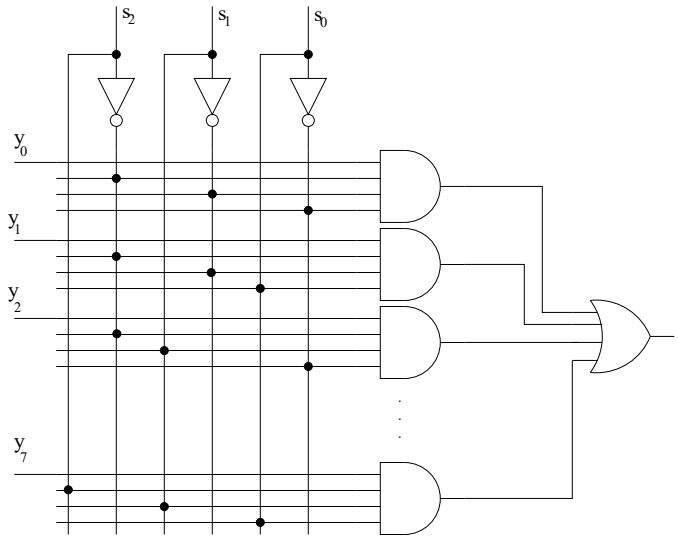


Figure 1.6: The internal organization of an 8:1 mux.

As with decoders, situations arise when a larger mux is built from smaller muxes. The key idea is to funnel down the many data inputs from smaller muxes to a single output. For example, construct a 16:1 mux from several 4:1 muxes.

In order to accommodate the 16 inputs, four 4:1 muxes are stacked on top of one another as shown in Figure 1.7. These four 4:1 muxes have a total of four bits of output. These four bits can nicely be routed by the inputs of a single 4:1 mux. The single data output of this mux is the input of the overall 16:1 mux.

The select lines are assigned to the 4:1 muxes based on the following argument. The inputs of the mux labeled **3** in Figure 1.7 corresponds to inputs $y_{15}y_{14}y_{13}y_{12}$ of the 16:1 mux. Each of these inputs requires four bits of select with the form $s_3s_2s_1s_0 = 11xx$. Hence, $s_3s_2 = 11$ must be the select for the output mux **4**. A similar argument for the 4:1 muxes labeled **0,1,2** reinforces the fact that s_3s_2 must be the select to the output mux.

In order to route y_{12} to the output, the select must equal $s_3s_2s_1s_0 = 1100$. Thus, routing at the input level of 4:1 muxes is controlled by s_1s_0. Examining all the other inputs reinforces this assumption.

Occasionally the need arises to construct a mux to handle "wide" data inputs, that is data inputs which consist of more than a single bit. A mux that can handle many bits is referred to as a *multibit mux*. A M-bit N:1 mux is defined as a N:1 mux whose data inputs and data outputs are M-bits wide. For example, the 4-bit 2:1 mux shown in Figure 1.8 has two data inputs and one data output each 4-bits wide. The internal organization of this mux is shown on the right-hand side of Figure 1.8. Four, 2:1 muxes are needed to accommodate all the data. Since the data inputs are to be handled as a single whole, they must be routed to the same input on each of the 2:1 muxes.

1.3 The Adder

An N-bit adder is a circuit which adds two N-bit binary numbers A, B together generating an N-bit sum S and an overflow indication, Ovf.

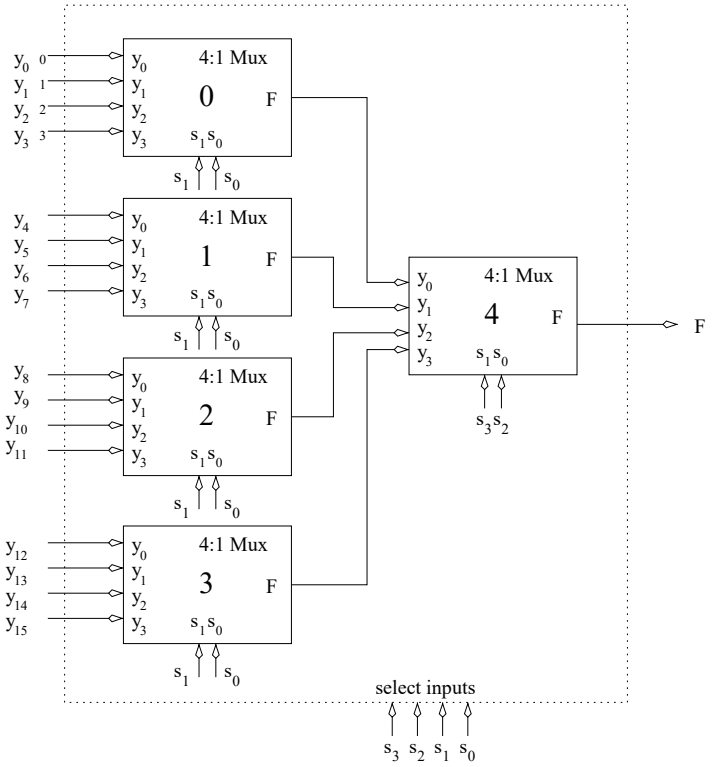


Figure 1.7: The construction of a 16:1 mux from 4:1 muxes.

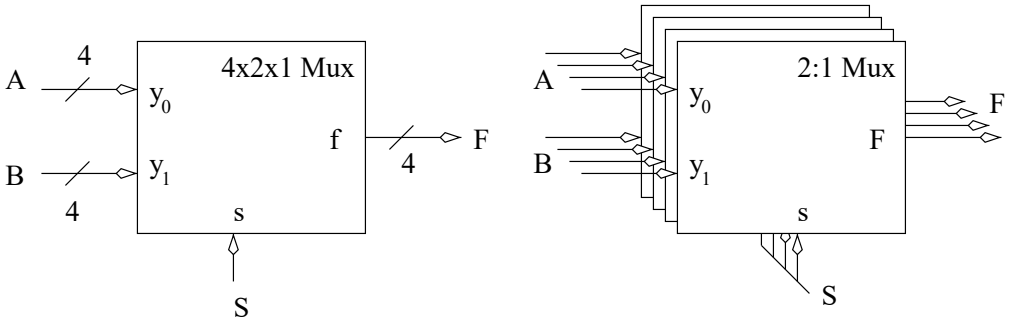


Figure 1.8: The organization of a 4-bit 2:1 mux. The slash labeled "4" denotes the fact that these signals are 4-bits wide.

Nomenclature:	N-bit adder
Data Input:	two N-bit vectors A and B
Data Output:	N-bit vector sum
Control:	none
Status:	1-bit ovf
Behavior:	$sum = A + B$

The behavior of the adder is exactly as expected after reading page **??**. Furthermore, the overflow output is asserted when the adder output is greater than or equal to 2^N. The con-

struction of an adder illustrates an approach that can be utilized for circuits whose function can be broken down into modular pieces.

Approaching the design of a 4-bit adder ($N = 4$) using the design philosophy introduced in Chapter 2, the truth table has eight bits of inputs, 256 rows, and nine bits of outputs. The task would be tedious, error-prone, producing a realization with a large number of gates. Instead of building the circuit as one monolithic piece, a module is designed that adds one set of bits together. Four of these modules are then connected in series to add four bits together. For example, in Figure 1.9 $A = 1011$ and $B = 0001$ are added together using the approach of page **??**.

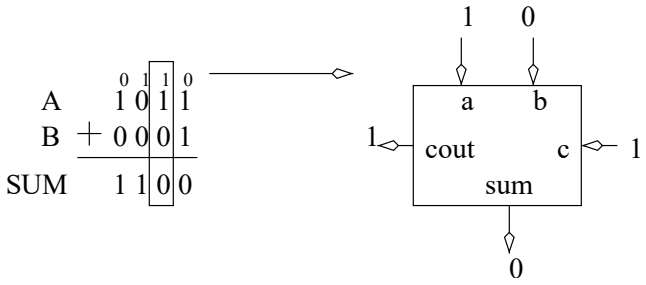


Figure 1.9: An addition problem and the values of one of its bit slices placed on a full adder.

This addition problem contains four *bit-slices* . Each bit-slice adds one bit's worth of the overall addition problem. The addition circuit is then constructed by stringing together four of these bit-slice adders (called full-adders). In the example shown in Figure 1.9, the second bit-slice of the addition problem is outlined. The inputs and outputs of this bit-slice are placed on a full adder. The full adder has three bits of input and two bits of output. To build a 4-bit adder, four full adders are strung together in series using the carry lines. The truth table for the full adder is created by enumerating every combination of inputs and determining the sum and carry-out for each. The carry-out and sum together are the 2-bit sum created by adding together the three input bits.

a	b	c	$cout$	sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$a\backslash bc$	00	01	11	10
0			1	
1		1	1	1

cout = bc + a b + ac

$a\backslash bcin$	00	01	11	10
0		1		1
1	1		1	

sum=a b' c' + a'b'c + abc + a'bc'

The SOP $_{\text{min}}$ expression for the outputs is arrived at by solving Kmaps for each of the outputs. Once, the internal organization of a full adder is complete, four of them are connected together as shown in Figure 1.10 to build a 4-bit adder.

The carry-out of each bit-slice becomes the carry-in of the next, more significant, full adder. This sequence is necessarily broken at the beginning and end of the chain of full adders. Since there is no carry-in to the least significant bit of an addition problem, the carry-in to the least significant full adder is set to 0. Since overflow occurs when the result of the addition process requires more bits than the word size, a carry-out from the most significant full adder indicates overflow.

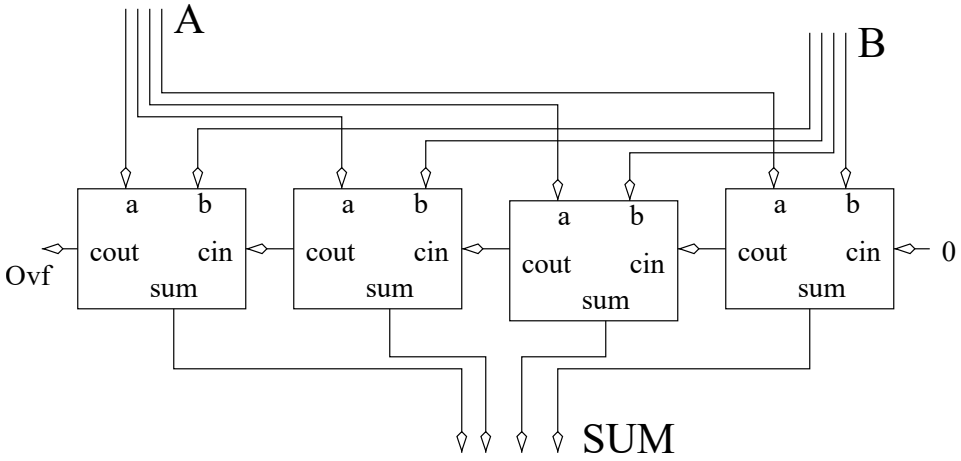


Figure 1.10: The arrangement of full adders to create a multi-bit adder.

1.4 The Adder Subtractor

As its name implies, an adder subtractor can perform two separate functions. From a black-box perspective, the only difference between this module and an adder is the presence of a control input to select which function the adder subtractor performs.

Nomenclature:	N-bit adder subtractor
Data Input:	two N-bit vectors A and B
Data Output:	N-bit vector s
Control:	1-bit f
Status:	1-bit ovf
Behavior:	if c=0 then $s = A + B$ else $s = A - B$

For now, assume the inputs and output of an adder subtractor are 2's-complement numbers. Addition of 2's-complement numbers proceeds like the addition of regular binary numbers. For now, ignore overflow conditions. The subtraction process for 2's-complement numbers, described on page **??**, rewrites the subtraction problem $A - B$ as an addition problem $A + (-B)$. The caveat is that B must be negated. Since both the addition and the subtraction problem require an adder, all that is required is to pass either B or $-B$ to the adder depending on which operation is to be performed. This idea is presented in Figure 1.11.

The "Complement or Pass" box in Figure 1.11 produces either B when $f = 0$ (addition) or $-B$ when $f = 1$ (subtraction). The process of complementing B is to flip the bits and to add 1. To avoid adding a second adder to perform the "add 1" operation the adder shown in Figure 1.11 performs both $A + B$ and the "add 1". This little piece of magic is accomplished by hijacking the carry-in to the least significant bit of the full adder chain shown Figure 1.10. Instead of hardwiring *cin* shown in Figure 1.11 to 0, *cin* is connected to f. When $f = 1$, an extra 1 will be added to sum. The 2's-complement of B is computed in two separate steps: The "Complement or Pass" box negates the bits of B (when $f = 1$) and the adder adds 1 (when $f = 1$). Hence, the circuit performs a subtraction when $f = 1$. When $f = 0$, the "Complement or Pass" box will pass through B unchanged, the adder does not add anything extra to the inputs and, consequently, the circuit performs $A + B$.

The "Complement or Pass" box is broken down into bit-slices, each bit of B is sent to its own slice along with the f control bit. If $f = 0$, then the slice outputs the input B bit. If

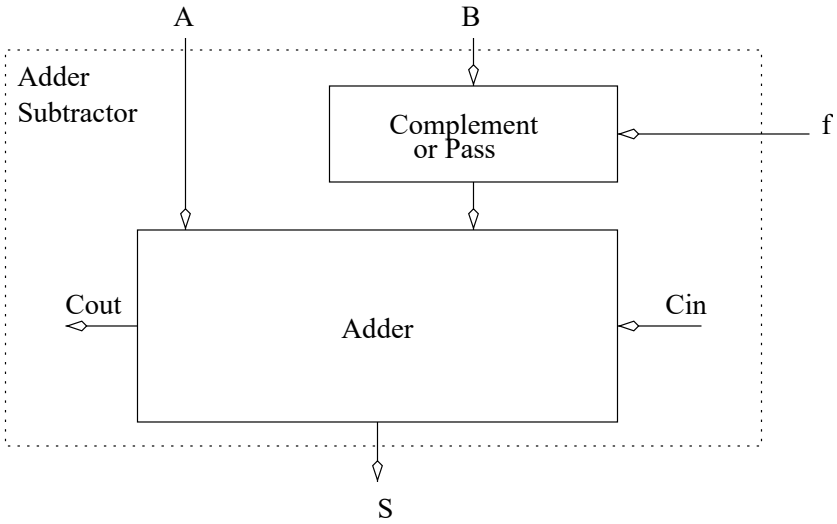


Figure 1.11: The idea behind the creation of an adder subtractor circuit.

$f = 1$, then the slice outputs the complement of the B bit. The truth table is shown below.

b	f	out
0	0	0
0	1	1
1	0	1
1	1	0

Solving the truth table yields $out = b'f + bf' = b \oplus f$. Each of these bit-slices is organized along with the components in Figure 1.11 yielding the circuit diagram in Figure 1.12.

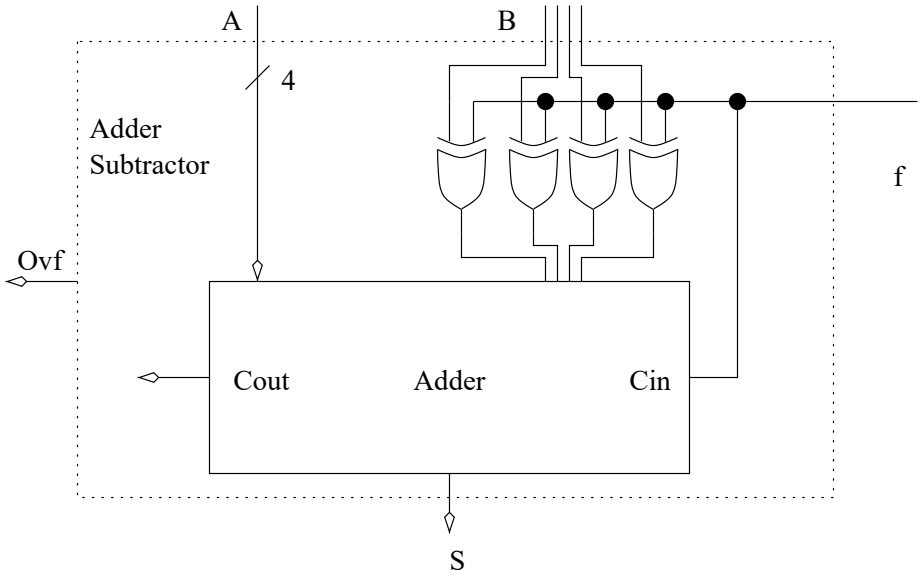


Figure 1.12: The construction of 4-bit adder subtractor.

According to page **??**, overflow in a 2's-complement operation occurs when the carry-in and carry-out to the most significant bit-slice of the addition disagree. In order to build a circuit to check this condition, the "Adder" box shown in Figure 1.12 must be opened revealing the chain of full adders as shown in Figure 1.11. The derivation of the truth table and the circuit realization is left as an exercise at the end of the chapter.

1.5 The Comparator

A comparator is a device which determines the relative magnitude of its two inputs.

Nomenclature:	N-bit comparator
Data Input:	two N-bit vectors X and Y
Data Output:	none
Control:	none
Status:	1-bit G, L, E
Behavior:	

cond	E	L	G
$X = Y$	1	0	0
$X < Y$	0	1	0
$X > Y$	0	0	1

Comparators are rather unique because they lack data output. That is not to say, comparators do not have any output. Rather, their status outputs are quite useful. The comparator examines its two N-bit inputs, denoted X and Y, and outputs three bits describing their relative magnitudes. Each of these three bits, E, L, G, is asserted when X equals Y, X is less than Y, or X is greater than Y, respectively.

Much like the adder circuit the truth table method is not very useful for designing large comparators. For example, a 16-bit comparator has two 16-bit inputs, or 32 bits of inputs, for an astounding 2^{32} rows in the truth table.

The construction of large comparators is based on the method that employed to determine the relative magnitude of two numbers X and Y, working from the most to least significant bits. At each step, compare a bit of X and Y. If the bits are equal, then continue to the next least significant bit. Otherwise, either X or Y is larger, and the comparison is over.

Large comparators are constructed by stringing together a series of modified 1-bit comparators as shown in Figure 1.13. Each bit-slice has as input a pair of bits from X and Y, and *Ein*, *Gin* and *Lin* signals from a more significant bit-slice. *Ein*, *Lin* and *Gin* tell a bit-slice the status of the magnitude of X and Y in the bit positions to its left. Each bit-slice has three bits of output, *Eout*, *Lout*, and *Gout*, communicating the relative magnitude of the inputs so far.

The truth table for a bit-slice of the comparator has five inputs, x, y, Ein, Lin and Gin. Each bit-slice has three outputs, $Eout, Lout$ and $Gout$. A portion of the truth table is shown below.

x	y	*Ein*	*Lin*	*Gin*	*Eout*	*Lout*	*Gout*
0	0	0	0	0	x	x	x
0	0	0	0	1	0	0	1
0	0	0	1	0	0	1	0
0	0	0	1	1	x	x	x
0	0	1	0	0	1	0	0

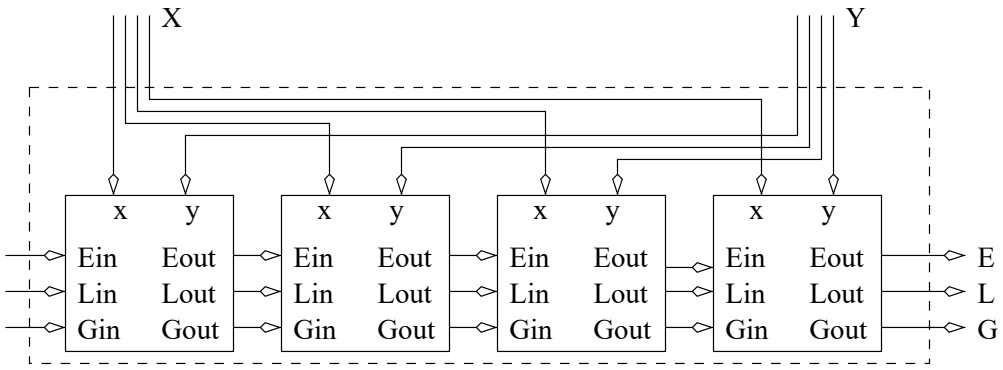


Figure 1.13: The arrangement of the bit-slices for the comparator.

With five bits of input, the truth table has a total of 32 rows, only five of which are shown above. Two rows have their outputs set to "don't cares" because they represent impossible situations. The inputs in the first row, state that the two numbers have no relative magnitude relative to one another. The inputs on the fourth row claim that $X < Y$ and $X > Y$ simultaneously, an impossible situation. The second row states that it has already been determined that $X > Y$, so it does not matter what x and y are because their value cannot effect a decision that has occurred in a more significant bit-slice. Similarly, the third row states that $X < Y$, so the outputs just propagate this fact, regardless of x and y, because the bits of this slice are less significant than those which decided $X < Y$. The fifth row states that so far $X = Y$ and since the bit-slice inputs are equal, the two inputs are still equal. The remainder of the truth table will be completed in an exercise at the end of the chapter.

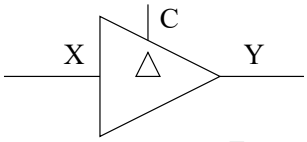
The final question that must be answered is "What should Ein, Lin, and Gin, on the far left of the Figure 1.13, be assigned?" The answer is that the Ein, Lin, Gin inputs to the most significant bit-slice should be set to 1,0,0 because X and Y are initially assumed to be equal. From the preceding paragraph, observe that whenever it is determined that $X > Y$ or $X < Y$, the comparator bit-slices will not change this fact. Hence, starting the circuit off from any other initial condition would cause an irreversible bias. Another way to view this situation is to realize that any binary number can be considered to have an infinite number of leading 0s, all of which are equal.

1.6 Three-State Buffer

Nomenclature:	Three-state buffer
Data Input:	1-bit X
Data Output:	1-bit Y
Control:	1-bit c
Status:	none
Others:	none
Behavior:	The output equals the input when $C = 1$ otherwise the output is disconnected from input.

When several devices share a common signaling pathway, as with a data bus, it is important for every device to be capable of asserting data onto the bus, but only one device at a time actually asserts data. This effect can only be accomplished if each device is capable of electrically disconnecting itself from the bus. The three-state buffer shown in Figure 1.14 does

this under the control of the c signal.



X	C	Y
x	0	Z
0	1	0
1	1	1

Figure 1.14: The symbol and truth table for a three-state buffer.

Notice, when $C = 1$, then $Y = X$; the output equals the input. But, when $C = 0$, the output equals Z. Z is a symbol commonly used in electrical engineering to refer to impedance (reciprocal of resistance). The Z means the output is not connected to anything.

It is common to see N three-state buffers organized together to produce an N-bit three-state buffer. In such an arrangement, each bit of input is routed to its own three-state buffer, and the control input is routed to all of the three-state buffers. Each three-state buffer contributes its one bit of output to the overall output from the N-bit three-state buffer.

Directly connecting together outputs of two gates creates a situation where the two gates may produce different outputs resulting in a short circuit. This connection will eventually lead to the failure of one or both of the gates. Clearly, a system should not be designed with such a flaw. However, cases arise when it makes sense to want to wire together the outputs of several gates.

The bus in a computer is a collection of related signals that move information between any pair of devices on the bus. Allowing a bus to be used by many different devices increases its utilization at the expense of increased complexity. Look at the example bus in Figure 1.15 which connects a CPU, RAM, and Disk. Each of these devices may want data to be sent or received from any of the others. The shaded box in each device (CPU, RAM, Disk) is called out on the right side of Figure 1.15. The input register latches data from the bus and the output register holds the data to be asserted on the bus. The three-state buffer can disconnect the output register from the bus allowing one of the other devices to assert data on the bus. It is important, and fairly obvious, that only one device at a time can assert data on the bus.

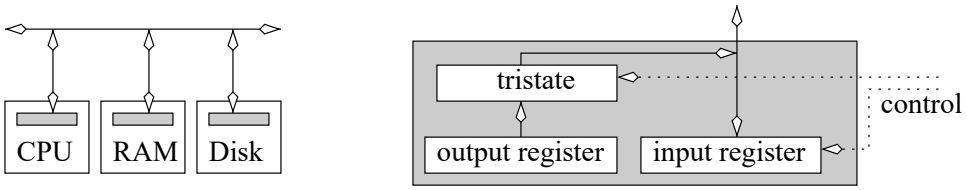


Figure 1.15: Three devices on a bus. The internal circuitry in each device to make it work.

1.7 Wire Logic

When designing a digital system which contains a device such as an adder which manipulates a pair of N-bit inputs representing binary numbers, it is easy to forget that those N bits are really N separate wires being grouped together. As a digital designer, one has access to each of these wires and can manipulate them as necessary. This point is important to remember when designing with the basic building blocks in this chapter because the number of bits representing a value must match the size of the device input. For example, a 4-bit value cannot be run into

an 8-bit-wide input and expected to work without some consideration of how to handle the four, remaining bit positions.

For example, assume a 4-bit binary number needs to be added to an 8-bit number. In order to accommodate the 8-bit number, an 8-bit adder needs to be used. Unfortunately, this decision creates a problem with the 4-bit operand because the upper four bits of its input are unoccupied. The solution is to fill in the upper four bits with 0s. This solution is called padding with 0s because just like when padding is placed around an item being shipped in a container to prevent it from moving around, padding a binary number with 0s keeps it aligned in its word-sized container. Further, the padding operation does not change the value of the 4-bit number.

Padding a 2's-complement number, a process called sign-extension, is slightly more complex. Consider the problem of adding or subtracting a 4-bit 2's-complement number to an 8-bit 2's-complement number using an 8-bit adder subtractor. The important point to keep in mind is that the value of the original, 4-bit, 2's-complement must be the same as the value of the sign-extended 8-bit 2's-complement value. If the 4-bit value were 1111, representing -1 in decimal, then padding the upper four bits with 0s would yield 00001111, which represents +15 in 8-bit 2's-complement. This approach is not the correct way to proceed because the original value of -1 was converted into +15 through the sign-extension process. Instead, the value should have been padded with 1s yielding 11111111, which represents -1 in an 8-bit 2's-complement number. To show that any negative 4-bit value padded with 1s retains its value, show that the values of original and sign-extended numbers are the same when the bits are flipped and 1 is added. Further, any positive 4-bit value should be padded with 0s to retain its value. In general, the 4-bit value needs to be padded with four copies of the most significant bit.

When representing a decimal value, using the base-10 numbering system, multiplication of a number by 10 can be accomplished by moving all the digits one position to the left and inserting a 0 in the vacated digit position. With binary-represented values, multiplication by 2 can be accomplished by moving all the bits one position to the left and inserting a 0 in the vacated position. This manipulation can be handled by padding the least significant bit position with a 0.

Another useful manipulation is to combine signals together. For example, consider a pair of 4-bit binary numbers are to be added together, and their 5-bit sum is to be reported. One alternative might be to pad each of the 4-bit inputs with a 0 and pass them along to a 5-bit adder. Alternatively, the numbers could be used unmodified as inputs to a 4-bit adder, and combine the overflow output of the 4-bit adder to the 4-bit sum output, producing a 5-bit result. Though unconventional, this manipulation is perfectly legal and perfectly correct because, the overflow signal is just the carry-out from the most significant full adder.

1.8 Combinations

The devices introduced in this section have limited utility when used by themselves. Their real potential is realized when combined together. When connecting two components, the data outputs of one device are typically connected to the data inputs of other. Likewise, the status outputs are typically connected to control inputs of another device. But how are the components arranged? A useful starting point is to phrase the solution of the design problem in terms of a simple algorithm. Algorithms are composed of statements. The algorithms to be constructed have several different types of statements.

Arithmetic Statements

Arithmetic statements perform the data manipulations required by design problems. An arithmetic statement consists of two parts, a left-hand side (LHS) and a right-hand side (RHS), related to one another by an equal sign. The RHS describes the operation and its inputs, while the LHS represents the variable denoting the output of the arithmetic operation. For example, the following line of code describes the addition of two values.

```
x = y + 3
```

The hardware realization of this line of code is an adder with `y` and 3 as inputs and `x` as its output. The algorithm does not specify the width of the `x` and `y` signals. These must be determined from accompanying information.

Conditional Statements

Conditional statements arise in programming languages in the form of `if/then/else` statements. All conditional statements consist of three parts, the condition to be checked (the `if` clause), the statement to be evaluated when the condition is true (the `then` clause), and the statement to be evaluated when the condition is false (the `else` clause).

Typically, the condition being evaluated seeks the relative magnitude of two binary numbers. For example, consider checking whether `(a<4)`. This comparison can be realized by routing `a` and `4` into the x and y inputs of a comparator and using the L output.

The consequence of the condition is to cause the evaluation either of the `then` clause or of the `else` clause. For now, these clauses will be arithmetic statements. In order to illustrate the hardware realization of a conditional statement, consider the following example.

```
if (a<4) then x=y+3 else x=y+7
```

The solution to the conditional assignment statement utilizes a comparator to determine the relative magnitudes of `a` and `4`. It is important to note in the solution shown in Figure 1.16 which of the comparator's inputs is x and y. Of the three status outputs, the L signal is used as the select on the multiplexer's inputs. The other two comparator outputs should not be shown since they are not used.

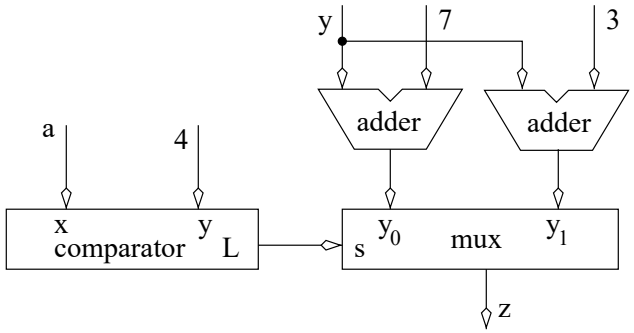


Figure 1.16: A combination of components to realize a conditional assignment statement.

Each of the arithmetic operations in the algorithm is realized by its own adder. Notice in Figure 1.16 that both adders compute their values, and the job of the multiplexer is to select one of the adder outputs based on the results of the L output from the comparator. Since

the LHS of both assignment statements involved the same variable, the output of the mux is labeled with x. When `a<4`, then $L = 1$ and, consequently, the y_1 output of the mux is routed to the output. According to the algorithm, when `a<4`, then $x = y + 3$. Consequently, $y + 3$ should be routed to the y_1 input of the mux. Now, only $y+7$ is left to be routed to the y_0 input of the mux. It is important to annotate the mux inputs with y_1 and y_0 in order demonstrate that the solution works correctly.

The previous solution is more complex than it needs to be. An adder can be removed from the circuit by noting that in the RHS of both assignments, the variable y has either 3 or 7 added to it. Consequently, a mux can be used to switch through either a 3 or 7 and then to add this mux's output to y as shown in Figure 1.17.

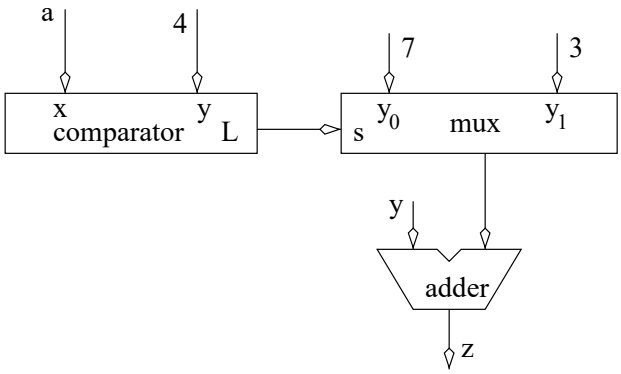


Figure 1.17: A better realization of the conditional assignment statement.

Make it a habit to identify ways to reduce the complexity of a circuit whenever possible. After all, one of the core principles of engineering is always endeavor to do the most with the least.

1.9 Exercises

1. **(2 pts. each)**Short answer:
   a) How many 3:8 decoders would it take to build a 9:512 decoder?
   b) How many AND gates are there in a 2^N:1 mux?
   c) How many AND gates are there in a 2^N : 1 mux which is constructed out of 2:1 muxes?
   d) How many AND gates are there in a 2^N:1 mux which is constructed out of 2^L:1 muxes, assume that 2^N is an integer multiple of 2^L?

2. **(6 pts.)**Determine the SOP $_{\text{min}}$ expression for each of the three outputs of a bit-slice of the comparator.

3. **(2 pts.)**Show how to connect together four 4-bit comparators to construct a 16-bit comparator.

4. **(2 pts.)**Determine the circuitry for the overflow detection circuit for a 2's-complement adder subtractor. See page **??**.

5. **(10 pts.)**Build a BCD to 7-Segment Display converter using Espresso.

Nomenclature:	BCD to 7-segment converter
Data Input:	4-bit vector $D = d_3d_2d_1d_0$
Data Output:	7-bit vector $Y = y_6 \ldots y_1y_0$
Control:	none
Status:	none
Behavior:	The output drives a 7-segment display pattern representing the BCD digit.

A binary coded digit (BCD) is a 4-bit binary number that is constrained to assume the values of 0-9. That is, 1010 ... 1111 are illegal BCD digits.

A 7-segment display is a box with seven inputs and seven output LED bars. Each input is wired to an LED bar that is illuminated when a 1 is applied to its input. Each of the seven LED segments is numbered according to the pattern shown on the left-hand side of Figure 1.18.

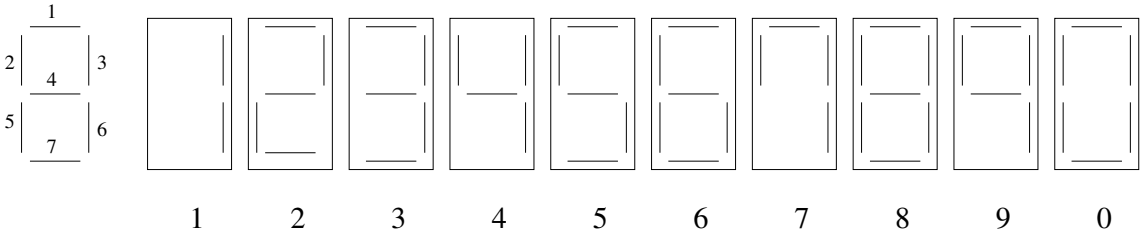


Figure 1.18: The numbering of the segments in a 7-segment display. The patterns of the BCD digits.

The pattern of LEDs to illuminate for each BCD digit is shown on the right-hand side of Figure 1.18. A BCD to 7-segment converter has four inputs, $d_3d_2d_1d_0$ and seven outputs $S_7 \ldots S_1$. Complete the design using Espresso. Make sure to include "Don't cares" in the truth table specification.

a) Use Espresso to determine the SOP $_{\min}$ expression for the outputs $S_7 \ldots S_1$. Underline product terms that are shared. Submit the Espresso source file.

b) Use Espresso to determine the POS $_{\min}$ expression for the outputs $S_7 \ldots S_1$. Underline sum terms that are shared. Submit the Espresso source file

6. **(10 pts.)** Build a box which has one 4-bit input called A and one 4-bit output called T. The output T is the 2's-complement value of the input A. Use the bit slice paradigm to solve this problem. That is, create a building block for one bit of the problem then string four of them together to solve the problem. For the problem at hand this can be done as follows:

   a) Start at the LSB of A.

   b) If this is the first, least significant, 1, flip all bits to the left.

   c) If this is not the first 1, leave the bit alone.

   d) Move one bit to the left.

   e) Goto Step b.

   A bit-slice should communicate whether there has been a 1 to the right, to the more significant bit. Submit:

   - How the above "algorithm" behaves when presented with the inputs A=1100
   - The truth table for one bit slice
   - SOP $_{\min}$ expression and circuit diagram for a bit slice.
   - The organization of four bit slices to solve the problem

7. **(4 pts.)** Build a 7:128 decoder using a minimum number of 4:16, 2:4 and 1:2 decoders. Describe the wiring of the select lines.

8. **(4 pts. each)** Design a circuit with two 8-bit inputs X, Y, an 8-bit output Z and a 1-bit input *sel*. Construct a circuit that yields the correct value of Z using only the basic building blocks presented in this chapter; do NOT show the internal organization of these building blocks. If a mux is used, denote which input is the y_0 and which is y_1. If a comparator is used denote which input is X and which is Y. Do not use any AND or OR gates; it will tempting in the later problems.

   a) `if (sel==0) then Z = X else Z = Y`

   b) `if (sel==0) then Z = X+Y else Z = Y`

   c) `if (sel==0) then Z = X+Y else Z = X-Y`

   d) `if (X==0) then Z = X else Z = Y`

   e) `if (X==Y) then Z = X-Y else Z = Y`

   f) `if (X==Y) then Z = X+Y else Z = X-Y`

   g) `if (X < Y) then Z = X else Z = Y`

   h) `if (X <= Y) then Z = X else Z = Y`

   i) `if (X > Y) then Z = X else Z = Y`

   j) `if (X > Y) then Z = X+X else Z = Y+Y`

9. **(10 pts.)** Build a 4-bit priority encoder.

Nomenclature:	N-bit priority encoder
Data Input:	N-bit vectored $D = d_{N-1} \ldots d_1 d_0$
Data Output:	$\log_2(N)$-bit vector $Y = y_{log_2(N)} \ldots y_1 y_0$
Control:	none
Status:	none
Behavior:	$F = i$ where i is the highest indexed input which equals 1. When all inputs equal 0, the output is a "don't care".

The idea is for the outputs to represent (in binary code) the highest input index which equals 1. For example, a 4-bit priority encoder with input $D = 1010$ has inputs $d_3 = 1$ and $d_0 = 1$. Of these two inputs, the index of d_3 is greater than the index of d_0 so the output, F is equal to 3, or in binary 11. If the input were $D = 0111$ then $F = 10$.

a) Write down the truth table for a 4-bit priority encoder. Hint, the truth table could be structured so that it contains only five rows by using "don't cares" on the inputs.

b) An SOP $_{\text{min}}$ realization of the circuit.

10. **(10 pts.)** Build a 4-bit saturation adder. A saturation adder performs normal 4-bit addition when the resulting sum is less than 15. If the sum is greater than 15, the saturation adders outputs 15. The following table summarizes.

Nomenclature:	4-bit saturation adder
Data Input:	2, 4-bit vectors `A, B`
Data Output:	4-bit vector `sum`
Control:	none
Status:	none
Behavior:	`if (A+B > 15) sum = 15` `else sum = A+B`

Submit a schematic showing the basic building blocks, their data status, and control interconnections. Show any truth tables used to build glue logic.

11. **(10 pts.)** Build a mod-6 adder. The mod-6 adder takes as input two 3-bit (mod 6) numbers and adds them together modulus 6.

Modular arithmetic only operates with a limited portion of the integers. The range of numbers is $\{0, 1, 2, \ldots, m-1\}$ where m is called the *modulus*; note there are m different integers because counting started at 0. For example, when working in mod-6 arithmetic use the integers $\{0, 1, 2, 3, 4, 5\}$. To solve any addition problem in modular arithmetic, it is only necessary to perform regular addition with the special rule that the addition process rolls over from the largest number, $m-1$ to 0 when the result is larger than $m-1$. For example, in mod-6 arithmetic $(5+1) \mod 6 = 0$. The statement " mod 6" is always included in the addition problem to indicate to the reader that mod-6 arithmetic is being performed. Here are a few more examples to help

$2 + 3 \mod 6 = 5$
$3 + 3 \mod 6 = 0$
$4 + 3 \mod 6 = 1$
$5 + 5 \mod 6 = 4$

Nomenclature:	3-bit mod 6 adder
Data Input:	two, 3-bit (mod-6) vectors `A, B`
Data Output:	3-bit (mod-6) vector `sum`
Control:	none
Status:	none
Behavior:	`sum = A+B mod 6`

Submit a schematic showing the basic building blocks, their data status, and control interconnections. Show any truth tables used to build glue logic. Be careful that the word size of the result is handled correctly.

12. **(1pt. each)**Convert the following to 2's-complement assuming a word size of eight bits.

    a) -35

    b) -128

    c) 67

    d) 128

13. **(1 pt. each)** Perform the following operations for the given 2's-complement numbers. Assume a word size of eight bits in all cases. Indicate where overflow occurs. If there is no overflow, convert the result to decimal.

    a) 01011101 + 00110111

    b) 11101011 + 11110001

    c) 01011101 + 10101011

    d) 10111011 - 11110001

    e) 01011101 - 00110111

    f) 01011101 - 10101111

14. **(5 pts.)** Build a 4-bit bus transceiver. A bus transceiver is defined by the following truth table.

Nomenclature:	N-bit bus transceiver.
Data:	two bidirectional N-bits vectors $X = x_{N-1} \ldots x_1 x_0$. $Y = y_{N-1} \ldots y_1 y_0$.
Control:	1-bit F and R
Status:	none

Behavior:

F	R	X	Y	comment
0	0	Z	Z	X and Y tristate
0	1	Y	Y	X = Y
1	0	X	X	Y = X
1	1	x	x	never applied

Flow is determined by the F and R signals denoting forward and reverse respectively. When $F = 1$, data flows from X to Y. In this case, X is acting like and input and Y is acting like an output. When $R = 1$, data flows from the Y input to the X output. This design relies heavily on tristate buffers.

Key	scancode	Key	scancode	Key	scancode	Key	scancode
0	45_{16}	1	16_{16}	2	$1E_{16}$	3	26_{16}
4	25_{16}	5	$2E_{16}$	6	36_{16}	7	$3D_{16}$
8	$3E_{16}$	9	46_{16}	A	$1C_{16}$	B	32_{16}
C	21_{16}	D	23_{16}	E	24_{16}	F	$2B_{16}$
P	$4D_{16}$	L	$4B_{16}$	M	$3A_{16}$	I	43_{16}

Table 1.1: Some keyboard scancodes.

15. **(3 pts.)** Build a 2:1 mux using some tristate buffers and an inverter.

16. **(3 pts.)** Build a 4:1 mux using some tristate buffers and two inverters.

17. **(10 pts.)** Build a flip box. A flip box is defined by the following input, output, and behavior definition.

Nomenclature:	8-bit flip box.
Data Input:	8-bit $D = d_7 \dots d_0$
Data Output:	8-bit $F = f_7 \dots f_0$
Control:	3-bit $S = s_2 s_1 s_0$
Status:	none
Behavior:	The output is the same as the input except for one bit which is inverted. The index of the inverted bit is given by S.

The flip box takes the 8-bit data input, flips a single bit identified by S, then sends the new 8-bit value to the output. For example, if $D = 11110000$ and $S = 010$ then $F = 11110100$. If $D = 11110000$ and $S = 101$ then $F = 11010000$. The solution should rely heavily on the basic building blocks.

18. **(10 pts.)** Build a box which recognizes some keyboard scancode. When a key is pressed on a keyboard, the keyboard transmits (among other things) an 8-bit scancode of the pressed key. Each key has its own scancode listed in Table 1.1. The relationship between the keys and their scancode is not based on ASCII.

Nomenclature:	scancode classifier
Data Input:	8-bit $D = d_7 \dots d_0$
Data Output:	IsP, IsL, IsM, IsI, IsS
Control:	none
Status:	none
Behavior:	IsP =1 when D is the scan code for the letter "P". IsL =1 when D is the scan code for the letter "L". IsM =1 when D is the scan code for the letter "M". IsI =1 when D is the scan code for the letter "I". IsS =1 when D is the scan code for the letter "S".

19. **(10 pts.)** Build a box which converts an 8-bit scancode for a hexadecimal digit into a 4-bit hexadecimal values.

Nomenclature:	scancode classifier
Data Input:	8-bit $D = d_7 \ldots d_0$
Data Output:	4-bit $H = h_3 h_2 h_1 h_0$
Control:	none
Status:	none
Behavior:	Converts the scancode D, representing a the key of a hexadecimal character, into its 4-bit value H.

For example, if $D = 25_{16}$, the scancode for the "4" key, then the converter should output $H = 0100_2$. Assume that the inputs are always legal hexadecimal scancodes.

Appendix A

74LS00 Data Sheets

Since the device documentation for the TI chips is covered by TI's copyright it was decided to leave these pages out of this text. The documents discussed in the text can be found onilne at:
http://focus.ti.com/lit/ds/symlink/sn74ls00.pdf

Index