ECE120: Introduction to Computer Engineering

Notes Set 2.5 Using Abstraction to Simplify Problems

In this set of notes, we illustrate the use of abstraction to simplify problems, then introduce a component called a multiplexer that allows selection among multiple inputs. We begin by showing how two specific examples—integer subtraction and identification of letters in ASCII—can be implemented using logic functions that we have already developed. We also introduce a conceptual technique for breaking functions into smaller pieces, which allows us to solve several simpler problems and then to compose a full solution from these partial solutions.

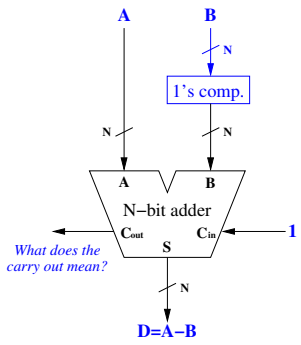
Together with the idea of bit-sliced designs that we introduced earlier, these techniques help to simplify the process of designing logic that operates correctly. The techniques can, of course, lead to less efficient designs, but *correctness is always more important than performance*. The potential loss of efficiency is often acceptable for three reasons. First, as we mentioned earlier, computer-aided design tools for optimizing logic functions are fairly effective, and in many cases produce better results than human engineers (except in the rare cases in which the human effort required to beat the tools is worthwhile). Second, as you know from the design of the 2's complement representation, we may be able to reuse specific pieces of hardware if we think carefully about how we define our problems and representations. Finally, many tasks today are executed in software, which is designed to leverage the fairly general logic available via an instruction set architecture. A programmer cannot easily add new logic to a user's processor. As a result, the hardware used to execute a function typically is not optimized for that function. The approaches shown in this set of notes illustrate how abstraction can be used to design logic.

2.5.1 Subtraction

Our discussion of arithmetic implementation has focused so far on addition. What about other operations, such as subtraction, multiplication, and division? The latter two require more work, and we will not discuss them in detail until later in our class (if at all).

Subtraction, however, can be performed almost trivially using logic that we have already designed. Let's say that we want to calculate the difference D between two N-bit numbers A and B. In particular, we want to find $D = A - B$. For now, think of A, B, and D as 2's complement values. Recall how we defined the 2's complement representation: the N-bit pattern that we use to represent $-B$ is the same as the base 2 bit pattern for $(2^N - B)$, so we can use an adder if we first calculate the bit pattern for $-B$, then add the resulting pattern to A. As you know, our N-bit adder always produces a result that is correct modulo 2^N, so the result of such an operation, $D = 2^N + A - B$, is correct so long as the subtraction does not overflow.

How can we calculate $2^N - B$? The same way that we do by hand! Calculate the 1's complement, $(2^N - 1) - B$, then add 1. The diagram to the right shows how we can use the N-bit adder that we designed in Notes Set 2.3 to build an N-bit subtracter. New elements appear in blue in the figure—the rest of the logic is just an adder. The box labeled "1's comp." calculates the 1's complement of the value B, which together with the carry in value of 1 correspond to calculating $-B$. What's in the "1's comp." box? One inverter per bit in B. That's all we need to calculate the 1's complement. You might now ask: does this approach also work for unsigned numbers? The answer is yes, absolutely. However, the overflow conditions for both 2's complement and unsigned subtraction are different than the overflow condition for either type of addition. What does the carry out of our adder signify, for example? The answer may not be immediately obvious.



Let's start with the overflow condition for unsigned subtraction. Overflow means that we cannot represent the result. With an N-bit unsigned number, we have $A - B \notin [0, 2^N - 1]$. Obviously, the difference cannot be larger than the upper limit, since A is representable and we are subtracting a non-negative (unsigned) value. We can thus assume that overflow occurs only when $A - B < 0$. In other words, when $A < B$.

 © 2000-2022 Steven S. Lumetta. All rights reserved.

To calculate the unsigned subtraction overflow condition in terms of the bits, recall that our adder is calculating $2^N + A - B$. The carry out represents the 2^N term. When $A \geq B$, the result of the adder is at least 2^N, and we see a carry out, $C_{out} = 1$. However, when $A < B$, the result of the adder is less than 2^N, and we see no carry out, $C_{out} = 0$. *Overflow for unsigned subtraction is thus inverted from overflow for unsigned addition*: a carry out of 0 indicates an overflow for subtraction.

What about overflow for 2's complement subtraction? We can use arguments similar to those that we used to reason about overflow of 2's complement addition to prove that subtraction of one negative number from a second negative number can never overflow. Nor can subtraction of a non-negative number from a second non-negative number overflow.

If $A \geq 0$ and $B < 0$, the subtraction overflows iff $A - B \geq 2^{N-1}$. Again using similar arguments as before, we can prove that the difference D appears to be negative in the case of overflow, so the product $\overline{A_{N-1}}\ B_{N-1}\ D_{N-1}$ evaluates to 1 when this type of overflow occurs (these variables represent the most significant bits of the two operands and the difference; in the case of 2's complement, they are also the sign bits). Similarly, if $A < 0$ and $B \geq 0$, we have overflow when $A - B < -2^{N-1}$. Here we can prove that $D \geq 0$ on overflow, so $A_{N-1}\ \overline{B_{N-1}}\ \overline{D_{N-1}}$ evaluates to 1.

Our overflow condition for N-bit 2's complement subtraction is thus given by the following:

$$\overline{A_{N-1}}\ B_{N-1}\ D_{N-1} + A_{N-1}\ \overline{B_{N-1}}\ \overline{D_{N-1}}$$

If we calculate all four overflow conditions—unsigned and 2's complement, addition and subtraction—and provide some way to choose whether or not to complement B and to control the C_{in} input, we can use the same hardware for addition and subtraction of either type.

2.5.2 Checking ASCII for Upper-case Letters

Let's now consider how we can check whether or not an ASCII character is an upper-case letter. Let's call the 7-bit letter $C = C_6C_5C_4C_3C_2C_1C_0$ and the function that we want to calculate $U(C)$. The function U should equal 1 whenever C represents an upper-case letter, and should equal 0 whenever C does not.

In ASCII, the 7-bit patterns from 0x41 through 0x5A correspond to the letters A through Z in alphabetic order. Perhaps you want to draw a 7-input K-map? Get a few large sheets of paper! Instead, imagine that we've written the full 128-row truth table. Let's break the truth table into pieces. Each piece will correspond to one specific pattern of the three high bits $C_6C_5C_4$, and each piece will have 16 entries for the four low bits $C_3C_2C_1C_0$. The truth tables for high bits 000, 001, 010, 011, 110, and 111 are easy: the function is exactly 0. The other two truth tables appear on the left below. We've called the two functions T_4 and T_5, where the subscripts correspond to the binary value of the three high bits of C.

C_3	C_2	C_1	C_0	T_4	T_5
0	0	0	0	0	1
0	0	0	1	1	1
0	0	1	0	1	1
0	0	1	1	1	1
0	1	0	0	1	1
0	1	0	1	1	1
0	1	1	0	1	1
0	1	1	1	1	1
1	0	0	0	1	1
1	0	0	1	1	1
1	0	1	0	1	1
1	0	1	1	1	0
1	1	0	0	1	0
1	1	0	1	1	0
1	1	1	0	1	0
1	1	1	1	1	0

T_4

C_1C_0 \ C_3C_2	00	01	11	10
00	0	1	1	1
01	1	1	1	1
11	1	1	1	1
10	1	1	1	1

$$T_4 = C_3 + C_2 + C_1 + C_0$$

T_5

C_1C_0 \ C_3C_2	00	01	11	10
00	1	1	0	1
01	1	1	0	1
11	1	1	0	0
10	1	1	0	1

$$T_5 = \overline{C_3} + \overline{C_2}\ \overline{C_1} + \overline{C_2}\ \overline{C_0}$$

As shown to the right of the truth tables, we can then draw simpler K-maps for T_4 and T_5, and can solve the K-maps to find equations for each, as shown to the right (check that you get the same answers).

How do we merge these results to form our final expression for U? We AND each of the term functions (T_4 and T_5) with the appropriate minterm for the high bits of C, then OR the results together, as shown here:

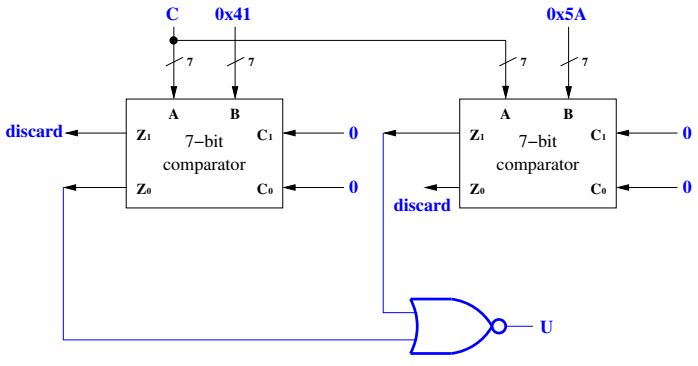
$$\begin{aligned} U &= C_6\ \overline{C_5}\ \overline{C_4}\ T_4 + C_6\ \overline{C_5}\ C_4\ T_5 \\ &= C_6\ \overline{C_5}\ \overline{C_4}\ (C_3 + C_2 + C_1 + C_0) + C_6\ \overline{C_5}\ C_4\ (\overline{C_3} + \overline{C_2}\ \overline{C_1} + \overline{C_2}\ \overline{C_0}) \end{aligned}$$

Rather than trying to optimize by hand, we can at this point let the CAD tools take over, confident that we have the right function to identify an upper-case ASCII letter.

Breaking the truth table into pieces and using simple logic to reconnect the pieces is one way to make use of abstraction when solving complex logic problems. In fact, recruiters for some companies often ask questions that involve using specific logic elements as building blocks to implement other functions. Knowing that you can implement a truth table one piece at a time will help you to solve this type of problem.

Let's think about other ways to tackle the problem of calculating U. In Notes Sets 2.3 and 2.4, we developed adders and comparators. Can we make use of these components as building blocks to check whether C represents an upper-case letter? Yes, of course we can: by comparing C with the ends of the range of upper-case letters, we can check whether or not C falls in that range.

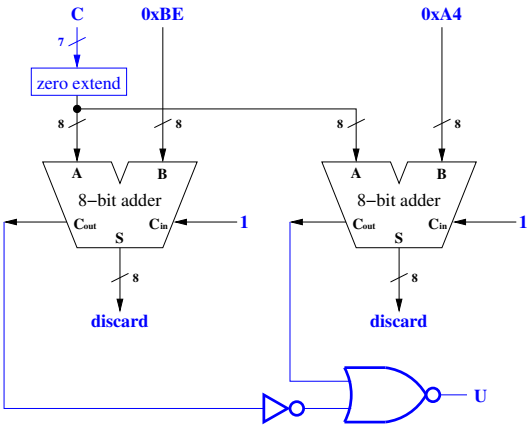
The idea is illustrated on the left below using two 7-bit comparators constructed as discussed in Notes Set 2.4. The comparators are the black parts of the drawing, while the blue parts represent our extensions to calculate U. Each comparator is given the value C as one input. The second value to the comparators is either the letter A (0x41) or the letter Z (0x5A). The meaning of the 2-bit input to and result of each comparator is given in the table on the right below. The inputs on the right of each comparator are set to 0 to ensure that equality is produced if C matches the second input (B). One output from each comparator is then routed to a NOR gate to calculate U. Let's consider how this combination works. The left comparator compares C with the letter A (0x41). If $C \geq$ 0x41, the comparator produces $Z_0 = 0$. In this case, we may have a letter. On the other hand, if $C <$ 0x41, the comparator produces $Z_0 = 1$, and the NOR gate outputs $U = 0$, since we do not have a letter in this case. The right comparator compares C with the letter Z (0x5A). If $C \leq$ 0x5A, the comparator produces $Z_1 = 0$. In this case, we may have a letter. On the other hand, if $C >$ 0x5A, the comparator produces $Z_1 = 1$, and the NOR gate outputs $U = 0$, since we do not have a letter in this case. Only when 0x41 $\leq C \leq$ 0x5A does $U = 1$, as desired.



Z_1	Z_0	meaning
0	0	$A = B$
0	1	$A < B$
1	0	$A > B$
1	1	not used

 © 2000-2022 Steven S. Lumetta. All rights reserved.

What if we have only 8-bit adders available for our use, such as those developed in Notes Set 2.3? Can we still calculate U? Yes. The diagram shown to the right illustrates the approach, again with black for the adders and blue for our extensions. Here we are actually using the adders as subtracters, but calculating the 1's complements of the constant values by hand. The "zero extend" box simply adds a leading 0 to our 7-bit ASCII letter. The left adder subtracts the letter A from C: if no carry is produced, we know that $C <$ 0x41 and thus C does not represent an upper-case letter, and $U = 0$. Similarly, the right adder subtracts 0x5B (the letter Z plus one) from C. If a carry is produced, we know that $C \geq$ 0x5B, and thus C does not represent an upper-case letter, and $U = 0$. With the right combination of carries (1 from the left and 0 from the right), we obtain $U = 1$.



Looking carefully at this solution, however, you might be struck by the fact that we are calculating two sums and then discarding them. Surely such an approach is inefficient?

We offer two answers. First, given the design shown above, a good CAD tool recognizes that the sum outputs of the adders are not being used, and does not generate logic to calculate them. The logic for the two carry bits used to calculate U can then be optimized. Second, the design shown, including the calculation of the sums, is similar in efficiency to what happens at the rate of about 10^{15} times per second, 24 hours a day, seven days a week, inside processors in data centers processing HTML, XML, and other types of human-readable Internet traffic. Abstraction is a powerful tool.

Later in our class, you will learn how to control logical connections between hardware blocks so that you can make use of the same hardware for adding, subtracting, checking for upper-case letters, and so forth.

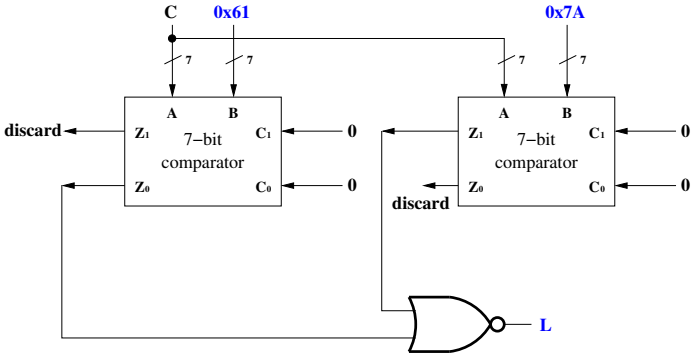
2.5.3 Checking ASCII for Lower-case Letters

Having developed several approaches for checking for an upper-case letter, the task of checking for a lower-case letter should be straightforward. In ASCII, lower-case letters are represented by the 7-bit patterns from 0x61 through 0x7A. One can now easily see how more abstract designs make solving similar tasks easier. If we have designed our upper-case checker with 7-variable K-maps, we must start again with new K-maps for the lower-case checker. If instead we have taken the approach of designing logic for the upper and lower bits of the ASCII character, we can reuse most of that logic, since the functions T_4 and T_5 are identical when checking for a lower-case character. Recalling the algebraic form of $U(C)$, we can then write a function $L(C)$ (a lower-case checker) as shown on the left below.

$$U = C_6\ \overline{C_5}\ \overline{C_4}\ T_4 + C_6\ \overline{C_5}\ C_4\ T_5$$

$$L = C_6\ C_5\ \overline{C_4}\ T_4 + C_6\ C_5\ C_4\ T_5$$

$$= C_6\ C_5\ \overline{C_4}\ (C_3 + C_2 + C_1 + C_0) + C_6\ C_5\ C_4\ (\overline{C_3} + \overline{C_2}\ \overline{C_1} + \overline{C_2}\ \overline{C_0})$$



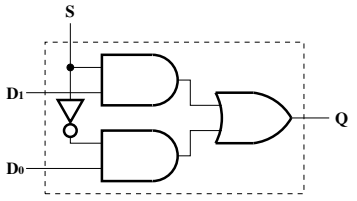
Finally, if we have used a design based on comparators or adders, the design of a lower-case checker becomes trivial: simply change the numbers that we input to these components, as shown in the figure on the right above for the comparator-based design. The only changes from the upper-case checker design are the inputs to the comparators and the output produced, highlighted with blue text in the figure.

2.5.4 The Multiplexer

Using the more abstract designs for checking ranges of ASCII characters, we can go a step further and create a checker for both upper- and lower-case letters. To do so, we add another input S that allows us to select the function that we want—either the upper-case checker $U(C)$ or the lower-case checker $L(C)$.

For this purpose, we make use of a logic block called a **multiplexer**, or **mux**. Multiplexers are an important abstraction for digital logic. In general, a multiplexer allows us to use one digital signal to select which of several others is forwarded to an output.

The simplest form of the multiplexer is the 2-to-1 multiplexer shown to the right. The logic diagram illustrates how the mux works. The block has two inputs from the left and one from the top. The top input allows us to choose which of the left inputs is forwarded to the output. When the input $S = 0$, the upper AND gate outputs 0, and the lower AND gate outputs the value of D_0. The OR gate then produces $Q = 0 + D_0 = D_0$. Similarly, when input $S = 1$, the upper AND gate outputs D_1, and the lower AND gate outputs 0. In this case, the OR gate produces $Q = D_1 + 0 = D_1$.

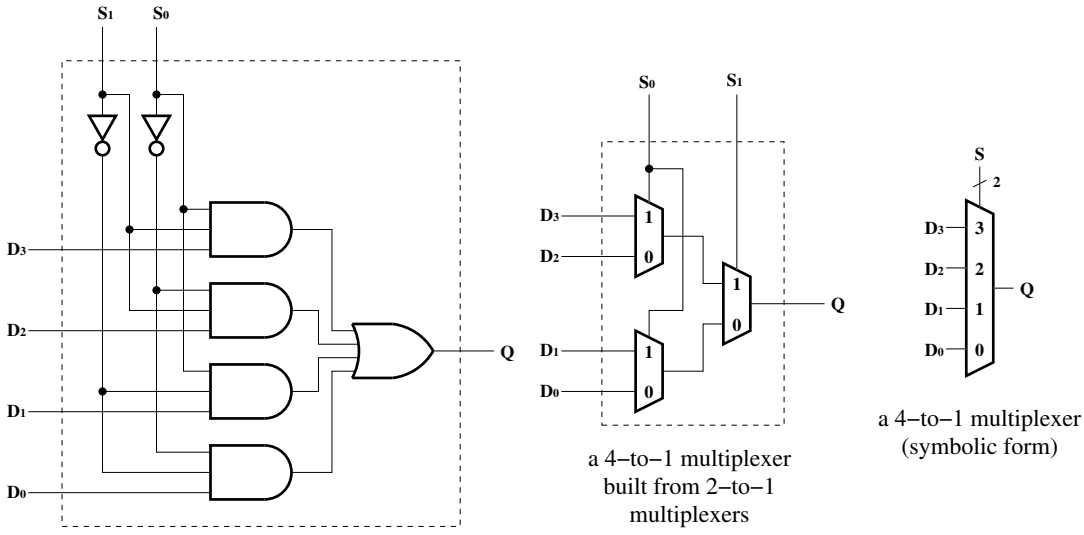


a 2–to–1 multiplexer (logic diagram)

a 2–to–1 multiplexer (symbolic form)

The symbolic form of the mux is a trapezoid with data inputs on the larger side, an output on the smaller side, and a select input on the angled part of the trapezoid. The labels inside the trapezoid indicate the value of the select input S for which the adjacent data signal, D_1 or D_0, is copied to the output Q.

We can generalize multiplexers in two ways. First, we can extend the single select input to a group of select inputs. An N-bit select input allows selection from amongst 2^N inputs. A 4-to-1 multiplexer is shown below, for example. The logic diagram on the left shows how the 4-to-1 mux operates. For any combination of S_1S_0, three of the AND gates produce 0, and the fourth outputs the D input corresponding to the interpretation of S as an unsigned number. Given three zeroes and one D input, the OR gate thus reproduces one of the D's. When $S_1S_0 = 10$, for example, the third AND gate copies D_2, and $Q = D_2$.



a 4–to–1 multiplexer (logic diagram)

a 4–to–1 multiplexer built from 2–to–1 multiplexers

a 4–to–1 multiplexer (symbolic form)

As shown in the middle figure, a 4-to-1 mux can also be built from three 2-to-1 muxes. Finally, the symbolic form of a 4-to-1 mux appears on the right in the figure.

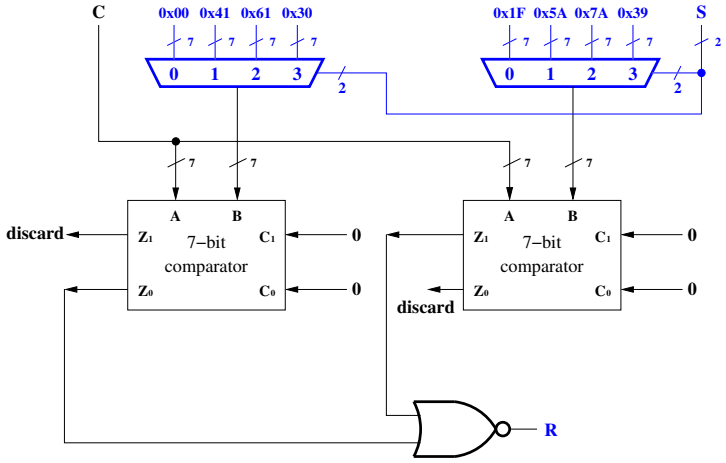
 © 2000-2022 Steven S. Lumetta. All rights reserved.

The second way in which we can generalize multiplexers is by using several multiplexers of the same type and using the same signals for selection. For example, we might use a single select bit T to choose between any number of paired inputs. Denote input pair by i D_1^i and D_0^i. For each pair, we have an output Q_i. When $T = 0$, $Q_i = D_0^i$ for each value of i. And, when $T = 1$, $Q_i = D_1^i$ for each value of i. Each value of i requires a 2-to-1 mux with its select input driven by the global select signal T.

Returning to the example of the upper- and lower-case checker, we can make use of two groups of seven 2-to-1 muxes, all controlled by a single bit select signal S, to choose between the inputs needed for an upper-case checker and those needed for a lower-case checker.

Specific configurations of multiplexers are often referred to as N**-to-**M **multiplexers**. Here the value N refers to the number of inputs, and M refers to the number of outputs. The number of select bits can then be calculated as $\log_2(N/M)$—N/M is generally a power of two—and one way to build such a multiplexer is to use M copies of an (N/M)-to-1 multiplexer.

Let's extend our upper- and lower-case checker to check for four different ranges of ASCII characters, as shown below. This design uses two 28-to-7 muxes to create a single checker for the four ranges. Each of the muxes in the figure logically represents seven 4-to-1 muxes.



The table to the right describes the behavior of the checker. When the select input S is set to 00, the left mux selects the value 0x00, and the right mux selects the value 0x1F, which checks whether the ASCII character represented by C is a control character. When the select input $S = 01$, the muxes produce the values needed to check whether C is an upper-case letter. Similarly, when the select input $S = 10$, the muxes produce the values needed to check whether C is a lower-case letter. Finally, when the select input $S = 11$, the left mux selects the value 0x30, and the right mux selects the value 0x39, which checks whether the ASCII character represented by C is a digit (0 to 9).

S_1S_0	left comparator input	right comparator input	$R(C)$ produced
00	0x00	0x1F	control character?
01	0x41	0x5A	upper-case letter?
10	0x61	0x7A	lower-case letter?
11	0x30	0x39	numeric digit?