

Lab 3 Lecture Notes

Introduction to Arithmetic Circuits and IP Cores

I. Introduction to Arithmetic Circuits

Part 1. Introduction to Multiplexers

The multiplexer is one of the basic building blocks of any digital design system. It takes a number of inputs and multiplexes them onto a single output line. That is, it selects one of the input lines, and passes its state to the output line.

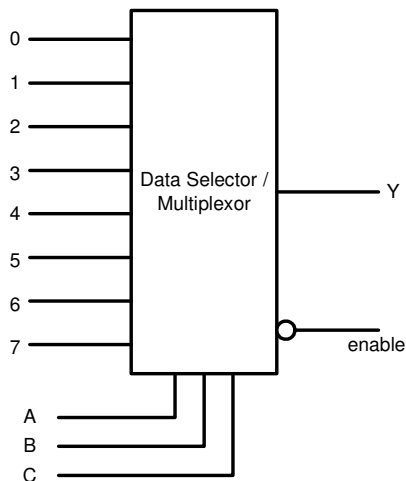


Figure 1: A Multiplexer

The input on the A, B, and C lines tells the multiplexer which one of the inputs 0-8 to pass to the output. In the image above if $A=0$, $B=1$, and $C=0$, then the input line selected to be passed to the output would be line 2.

This property of multiplexers is very useful. As you will see later on, we can use a multiplexer to implement complex logic functions. In terms of practical lab experience, it allows one chip, the multiplexer, to do the job of several simple logic gates. Later on, you will see how to use the multiplexer to implement binary addition and subtraction. However, before you can do this, you need to know something about binary arithmetic.

Part 2. Digital Arithmetic - Addition

This section discusses the basics of digital addition. Here is a block diagram of a binary adder:

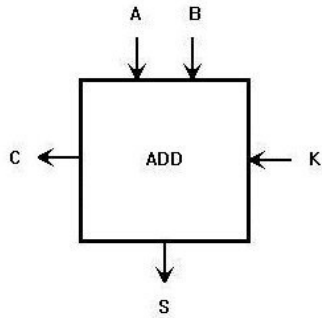


Figure 2. Adder block diagram

From this diagram, you can see that the adder has three inputs and two outputs. This will mean that one must generate two different K-maps, one for each output. Here are the functions each line performs:

- A,B - The two numbers to be added
- K- Carry-in
- C- Carry-out
- S- Sum

A	B	K	S	C
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Figure 3. Full adder truth table

Using Multiplexers to Design a Binary Adder

Instead of using the 8-to-1 MUX introduced in Part 1, the 4-to-1 MUX will be used to implement the binary adder. The following is the logic diagram of the 4-to-1 MUX.

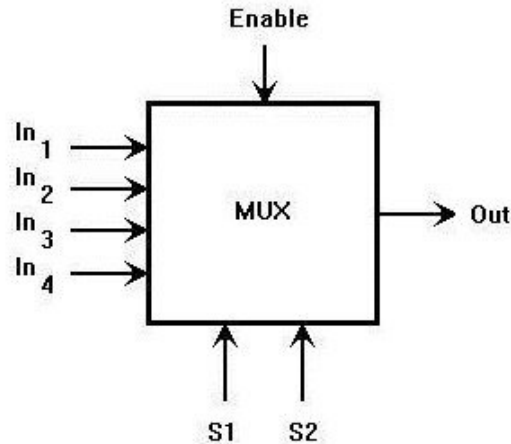


Figure 4. Adder implemented using a mux

Design process

Using the truth table shown in figure 3, generate two K-maps; one K-map for the Sum, and one for the Carry Out.

		B K			
		00	01	11	10
A	0	0	1	0	1
	1	1	0	1	0

Figure 5. K-map for the Sum

Let's take some time to inspect this Karnaugh map. Note that the Sum is A when B and K are both true and false. The Sum is the complement of A when B or K is true, but not both.

Now, how can we use this knowledge to create the Sum from a 4-to-1 MUX? Recall that the output is either A or the complement of A. This makes things easier. Also, notice how the 4-to-1 MUX has two control lines, S1, and S2. Recall that the value on the control lines determines which input line is passed to the output.

Here is the solution:

If we tie B to S1 and K to S2, then we can tie A to input lines 0 and 3 (00 and 11). Then we must tie the complement of A to input lines 1 and 2 (01 and 10). This allows us to use a 4-to-1 MUX to produce the Sum. Figure 6

shows a block diagram for the sum implemented using 4x1 Mux.

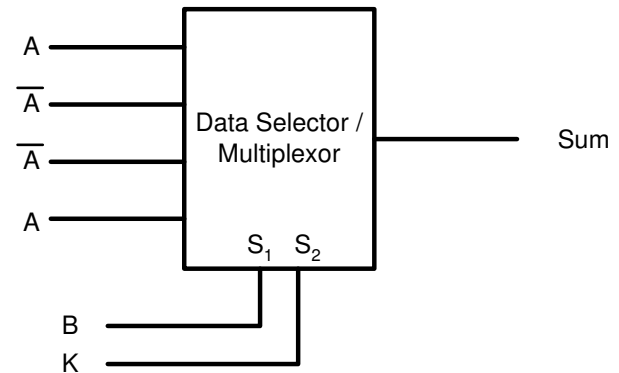


Figure 6. Block diagram for SUM

Now, we have to address the Carry Out output.

		B K			
		00	01	11	10
A	0	0	0	1	0
	1	0	1	1	1

Figure 7. K-map for the Carry-Out.

Now, inspect the K-map shown in figure 7 for a moment. Think of how we could use this K-map to produce the Carry-Out using a 4-to-1 MUX.

Note that when B and K are both false, then the output is also false. Note that when B and K are both true, then the output is also true. Finally, note that when B or K are true, but not both, the output is A.

So, if we again apply B to S1 and K to S2, we can generate the correct Carry-Out by tying input line 0 to ground, input line 3 to Vcc, and both input lines 1 and 2 to A. This will generate the correct output. And, here, you can see that we can build a simple one-bit adder using just two 4-to-1 MUX. The block diagram for the carry out is shown in figure 8.

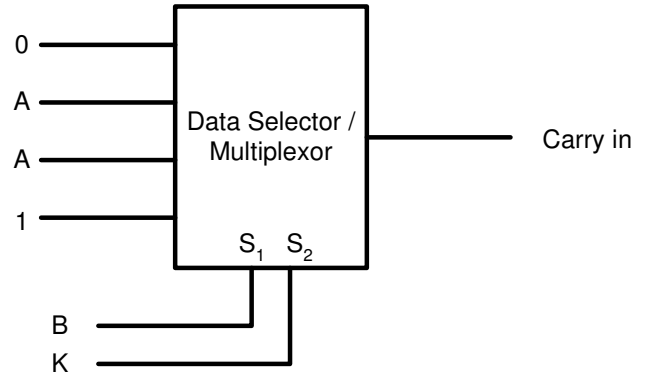


Figure 8. Carry out block diagram

If you want to add bigger numbers, you can cascade several binary adders together to produce the sum.

Part 3. Digital Arithmetic - Subtraction

This section discusses the basics of digital subtraction. Here is a block diagram of a binary subtractor.

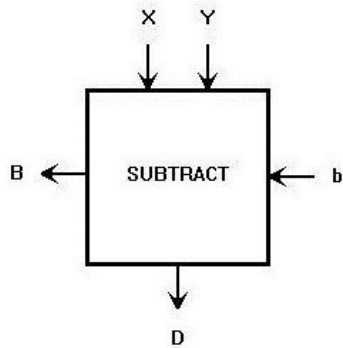


Figure 8. Subtractor block diagram.

From figure 8, you can see that the subtractor has three inputs and two outputs. This will mean that one must generate two different K-maps, one for each output. Here are the functions each line performs.

- X,Y - The two numbers to be subtracted.
- b - Borrow-in.
- B - Borrow-out.
- D – Difference.

This subtractor will work on two one-bit numbers X and Y. Figure 9 show the truth table for the subtractor.

X	Y	b	D	B
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Figure 9. Subtractor truth table.

II. Introduction to IP Cores

Part 1: Digital Design -- Using IP Cores to Simplify Design

In the world of digital design, one uses Hardware Description Languages to describe complex logic functions. These are encapsulated into design suites such as Xilinx's ISE and similar tools. However, if a digital engineer were to code an adder, or create a cosine lookup table each time he or she were on a project, the engineer would be wasting his or her time by reinventing the wheel. Alternatively, if the design engineer had to continually re-code commonly used complex digital circuits in large projects; they would also end up wasting money.

Because of this, a digital design engineer may just use an IP (Intellectual Property) core. An IP core is a block of HDL code that other engineers have already written to perform a specific function. It is a specific piece of code designed to do a specific job. One can use these cores in a complex design where an engineer wants to save the time he or she might have otherwise used to design the given function from scratch.

As with any engineering tool, IP cores have their advantages and disadvantages. Though an IP core may simplify a given design, the engineer has to design the whole project around the requirements of the IP core. And while an IP core may reduce design time, the engineer frequently has to pay for the right to use the core, since it is someone else's intellectual property.

Part 2: VHDL – Structural description

Structural description is the lowest level of abstraction possible to describe a circuit or system using VHDL. The description will basically consist on the specification of the connection between the different components of the subsystem. Consider the circuit shown in figure 1.

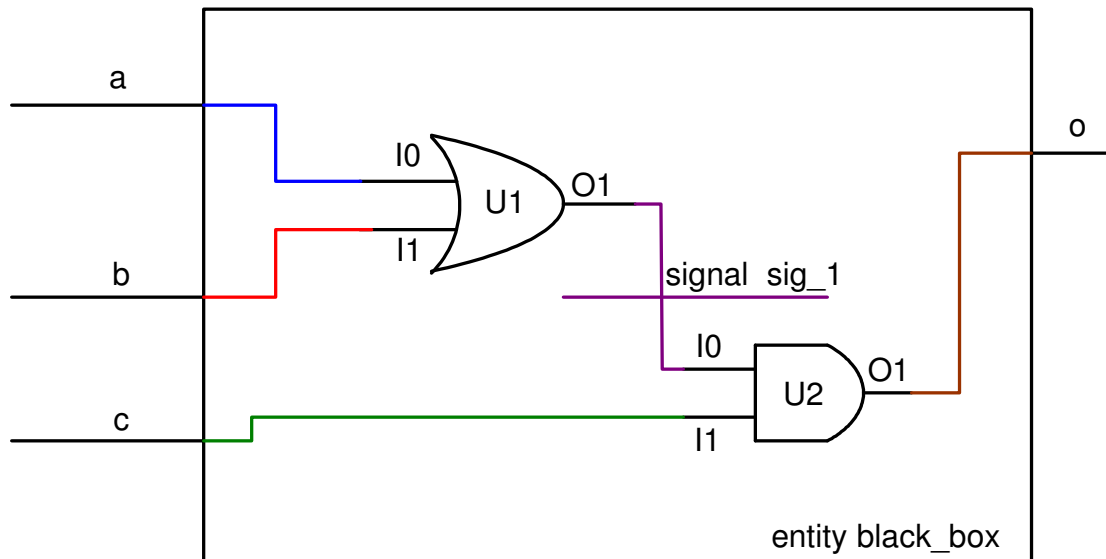


Figure 1. Circuit to be described using structural level of abstraction

The system **black_box** is made out of two components: **U1** and **U2**. The first step is to declare these components as part of the system. This is done as shown in the piece of code below.

```
architecture STRUCTURAL of black_box is           (line 1)

component U1                                       (line 2)
  port (                                           (line 3)
    I0, I1: IN std_logic;                          (line 4)
    O1: OUT std_logic;                             (line 5)
  );                                               (line 6)
end component;                                    (line 7)

component U2                                       (line 8)
  port (                                           (line 9)
    I0, I1: IN std_logic;                          (line 10)
    O1: OUT std_logic;                             (line 11)
  );                                               (line 12)
end component;                                    (line 13)

begin                                             (line 14)
```

Note that the components are declared between the architecture declaration line (line 1 in the code above) and the begin (line 14). The declaration of the component consists simply on the description of its inputs and outputs.

After the component declaration, follows the component instantiation which is the specification of the connections between the components. Component instantiation for the circuit shown in figure 1 would be as described by the code below:

```
begin                                             (line 1)
your_instance_name : U1                          (line 2)
  port map (                                       (line 3)
    I0 => A,                                       (line 4)
    I1 => B,                                       (line 5)
    O1 => sig_1                                    (line 6)
  );                                               (line 7)

your_instance_name2 : U2                         (line 8)
  port map (                                       (line 9)
    I0 => sig_1,                                   (line 10)
    I1 => C,                                       (line 11)
    O1 => o                                        (line 12)
  );                                               (line 13)
end structural;                                   (line 14)
```

Line 1 is the begin line for the structural description. Note that in the name of the inputs and outputs of the components are declared in the left column while the name of the connections is declared in the right column. For instance line 4 says that the input line I0 of the component U1 is connected to the input of the circuit A. Similarly, line 10 says that the input line I1 of the component U2 is connected to the input C of the circuit.

Now, from the schematic of the circuit shown in figure 1 we know that we want to connect the output O1 of component U1 with the input I0 of component U2. This connection cannot be made directly because one point of the connection is an input (I1) and the other point is an output (O1). We cannot mix apples and potatoes (inputs and outputs) together. That is why we need to declare a **signal** which will act as a bridge between the input and output that we want to connect together. That is why, line 6 of the code above shows a connection between O1 of component U1

with the signal sig_1, and line 11 shows a connection between O1 of the component U2 with the same signal. The connection has now been made.

The signal we used must be declared in the same section of the code as the component declaration as follows:

```
signal sig_1 : std_logic;
```

Finally, note that the code above shows a tag “***your_instance_name***” before the name of the component U1 or U2. That tag will carry a name for the component instantiation. For instance, if our system has more than one component of the same kind (lets say two components U1), that tag will allow us to differentiate between the instantiation of the two similar components.

Structural description is very useful when piece of code representing black boxes, is given to you to put then together to build a system.