

# CS 210: Quartus II Part 3

## Help, Buses, Naming, & Parameterized Modules

### Project: Design of an ALU

#### Overview

In this lab we will learn a final set of techniques for circuit design in QuartusII, including the use of buses (multiple wires), how to avoid drawing too many wires by using the right bus names, and the use of predefined circuits such as multiplexers. However, since this is the last lab that will include training in QuartusII, we will simply describe the techniques briefly and explain why they are useful, and then use the Help system to learn about them—hereafter you will have to use the Help system to learn more about QuartusII, and you should get used to its features. You will then start the actual design of a processor by designing the ALU. As usual, the files implementing this ALU will be submitted as the last problem for the upcoming Homework.

#### Help in Quartus II

The help system in Quartus II is well-designed and informative, and the best source of information on particular features; you should get used to using it as you proceed. There are two modes to know: one is the normal Help menu, from which you can do the expected lookup of keywords like any other application; the other is the Context-Sensitive Help button. This is the button on the toolbar with an arrow and question mark. If you click on this button and then click on an object (say, a gate in your graphic design file, or something you don't understand in the Waveform Editor), it will bring up an appropriate help window with information on that object.

You **MUST** get used to the help system in Quartus II, as it is the best way to learn advanced techniques; even the long manual from Altera does not go into enough detail to really teach you all you need to know.

#### Buses and Naming

One of the most important things to learn about graphic design when creating complex circuits is how to route not just single bits, but sequences of bits. For example, our CPU will not manipulate single bits, but 16-bit *words* of data and instructions. Drawing individual wires for each of the 16 bits would be absurdly difficult, so you need to learn how to use buses, or multiple wires, in your design. The basic idea is that when you make a connection between two devices, you can draw a multi-bit wire, called a bus, by selecting the thicker line from the selection of wire styles in the drop-down menu in the upper-right corner of the Graphic Editor, and then drawing the wire as usual. You must give buses names, which look something like array names of either one or two dimensions. To name a bus (or any wire), simply select it with the Selection Tool, and then type in the name; you

can then drag the name around if you wish. To specify a bus called A that is 8 bits wide, you would use

A[7..0]

to specify that the bus has bits 7 through 0. To refer to the MSb, you can then use “A[7]” or simply “A7.” To create an input or output that is a bus, you simply give it a bus name.

One of the nicest features of names is that they can help you avoid drawing too many wires or buses. Having given a bus a name, if you want to make a connection to the bus or to some sequence of bits or just to one bit inside the bus, you can simply give another wire or bus elsewhere in the system the appropriate name, and there will be an (invisible) connection between the two. For example, to break the bits coming down A into three separate wires, you could do this:



A comprehensive use of this technique will make your design much cleaner, as for example, you never need to actually connect any bus input or output with an actual wire—just give it the appropriate name and use those names elsewhere to make the connection.

One small but important note on bus names: names of buses are local to a design file, similar to local variables inside a C function. This means that you may not give two buses the same name inside one file, but it also means that a name has no meaning outside the design file. But, when using a symbol in your design (say, using the adder1 circuit to build your adder4 circuit), the names you gave to the inputs and outputs in the adder1 graphic design file will appear in the icon to help you remember what the different inputs and outputs are. However, these names are just “comments” have no meaning in the new design; you can’t refer to these names in the new design. Thus, you must give new names to the wires attached to the icon.

You need to read more about buses and naming than we have room for here. Look in the Search of the Help system for “buses” and double-click on the subtopic you are interested in. From this, you can (and should) follow links to read about Bus Names and anything else that looks useful. Look at all the examples carefully.

## Library of Parameterized Modules

We have to this point used only basic gates and devices<sup>1</sup> from the **primitives** library. Another important library to know about, however, is the **gates** library under **megafunctions** library, which contains predefined devices such as adders, decoders, and multiplexers. Before proceeding further, use the Help system to look up “megafunctions” then “Megafunctions/LPM” and look over the list of available devices. In particular, we will use the **lpm\_mux** device, so you can also search for lpm\_mux.

---

<sup>1</sup> We have learned about various gates and devices such as input and output, but you should explore the **prim** library for other useful devices. **In particular, learn about VCC, which outputs a constant logical 1 value, and GND, which outputs a constant 0 value.**

To use a LPM device, simply insert it in your design from the **megafunctions/gates** library. In the Symbol window, on the bottom left, “Launch Megawizard Plug-in” is selected. Unselect this option. Instead select the “Repeat insert mode” option. Then click OK. A window attached to the multiplexer will appear on the top right. You can set the parameters by clicking on this window. For example, if you insert **lpm\_mux**, you must set the parameters **lpm\_size** (the number of data inputs, e.g., a 8-to-1 MUX has size 8) and the **lpm\_width** (the width of each line, e.g., our designs will have data paths of width 16). Simply click on the name of the parameters and set the value in the Parameter Value input box. For MUX’s, you need only set the size and width.

## Design of an ALU: Basic Circuits

Opcodes	Instructions where used	Opcodes in Binary	Operation	Meaning in C++
0	srl	000	Shift A right (logical)	
1	sra	001	Shift A right (arithmetic)	Res = A >> 1;
2	add	010	Addition	Res = A + B;
3	sub	011	Subtraction	Res = A - B;
4	and	100	Bit-wise AND	Res = A & B;
5	or	101	Bit-wise OR	Res = A   B;
6	not	110	Bit-wise NOT	

Figure 1

The instructions that you will implement in your CPU design will include basic arithmetic and logic instructions. The ALU must support these instructions. We have chosen a small subset to implement, and so you will only have to implement the seven operations given in Figure 1; since addition and subtraction use the same adder circuit, there are actually only six basic circuits that are necessary. Later you will see how they are used in the assembly language instructions of the processor we will implement.

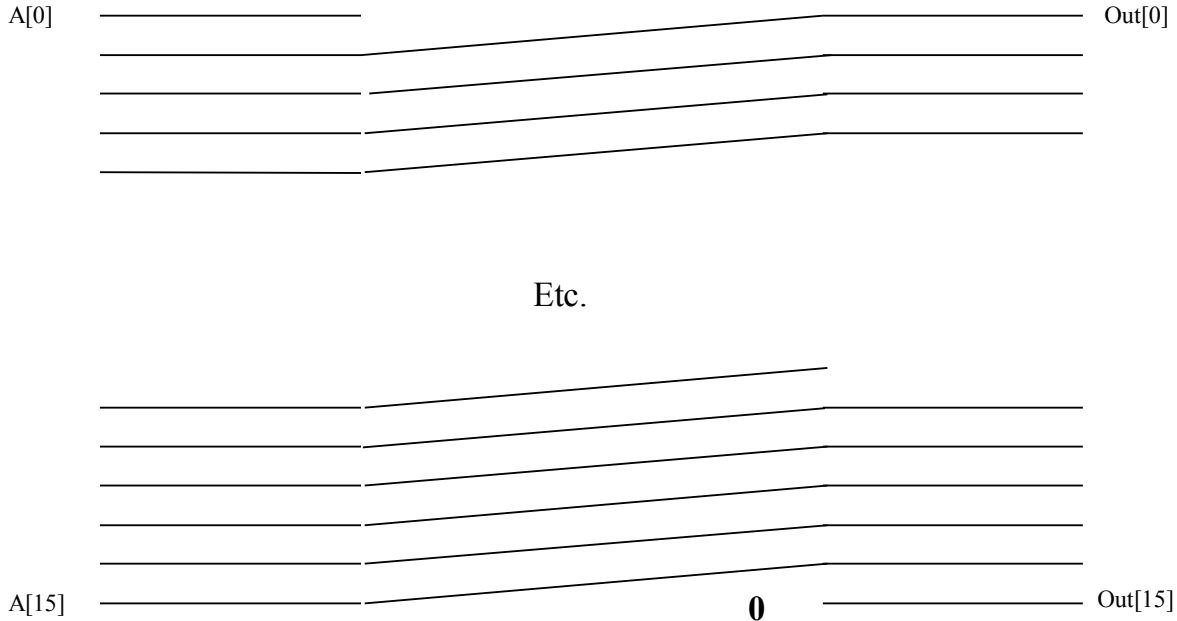
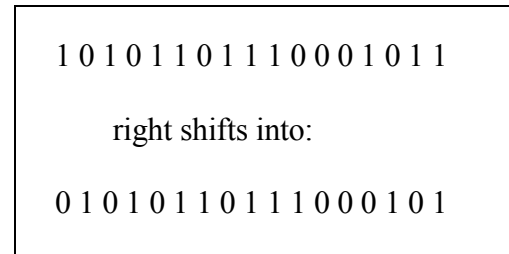
The ALU must perform the following operations shown in Figure 1 on two 16-bit inputs A and B; the opcode 7 corresponds to an instruction that does not use the ALU. The basic design of the ALU (see the circuit diagram on the last page) provides functional units (the ovals in the diagram) for each of these functions, and then has the opcodes control a large multiplexer which routes the appropriate result to the ALU Result output bus. Note that all of these units will compute their own function regardless of the opcode, but that the multiplexer will ensure that only the selected result will end up on the output bus. In the case of opcode 7, we can tie this MUX input to VCC (constant 1), or GND (constant 0), or, indeed, to any of the devices, since the result of the ALU is ignored by the rest of the CPU for this opcode.

Your task in this lab is to start to create the circuits for each of the basic operations given in Figure 1. Each should be a separate circuit with separate .bdf, .bsf, and .vwf files, and then incorporated into the ALU design on the last page. We will now go through each functional unit and explain how it works and what role it plays in implementing the instructions listed in the table. Please refer to the ALU diagram on the last page as you read through these descriptions.

## Right Shifter (Logical)

This takes a 16 bit binary number as input, and moves all the bits right one position; the low-order bit of the input is thrown away, and the high-order bit of the output is a 0 (i.e., a 0 is shifted in from the left), as shown in the box on the right.

To create this circuit, you need to have a single bus input (perhaps called “A[15..0]”) and a single output



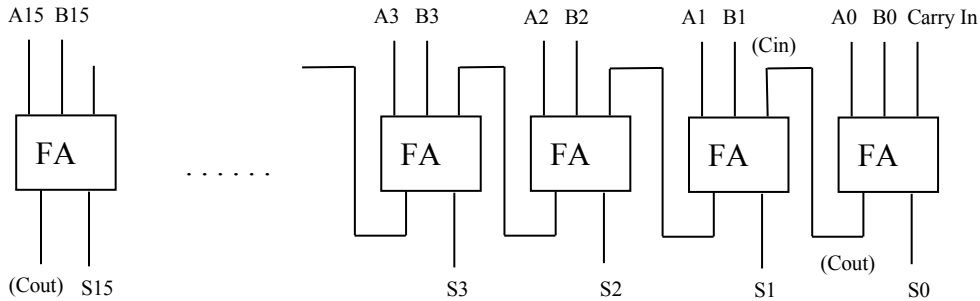
(perhaps “Out[15..0]”). If you had separate inputs and outputs to connect, you could simply draw connections between them as shown above; since we will be using names to connect buses together, you need to use the **wire** device (from the **prim** library) instead. Simply insert a wire (which does nothing but pass values along unchanged) and draw buses in and out of it with appropriate names (e.g., A[15..1] to Out[14..0]). Use the **Gnd** device from the **primitives/other** library to produce a constant 0 value. Note that the input A0 is not used; this is ok, but will produce a warning when you compile the circuit. Ignore it. When creating this circuit, call it **srl.bdf**, **srl.bsfc**, and **srl.vwf**.

## Right Shifter (Arithmetic)

This is the same as the previous, except that you must duplicate the sign bit (bit 15) in the high-order bit of the output. Call this circuit **sra.bdf** and so on.

## Ripple-Carry Adder

This is built by staging 16 Full Adders in a row:



This was covered in Lab Two, where you created a 1-bit version. For this version, you can either continue to add 15 more full adders to your 1-bit design, or (better) you can simply build your 16-bit version from 8 of 2-bit adders, connecting carry-outs to carry-ins, or you can package two 4-bit adders to make an 8-bit adder, then use 2 of these. You get the idea... Call this circuit **adder16.bdf**.

## Bit-wise AND and Bit-Wise NOT

These are simple to implement: the first one applies the AND gate to all the bits of A and B in parallel, and the last applies the NOT gate to all the bits of B in parallel. Note in the ALU design on the last page how the Bit-wise NOT is used both as a primitive operation, and used as input to the adder in the case of subtraction. Call these **bwand.bdf** and **bwnot.bdf**.

## Design of an ALU: Putting it all together

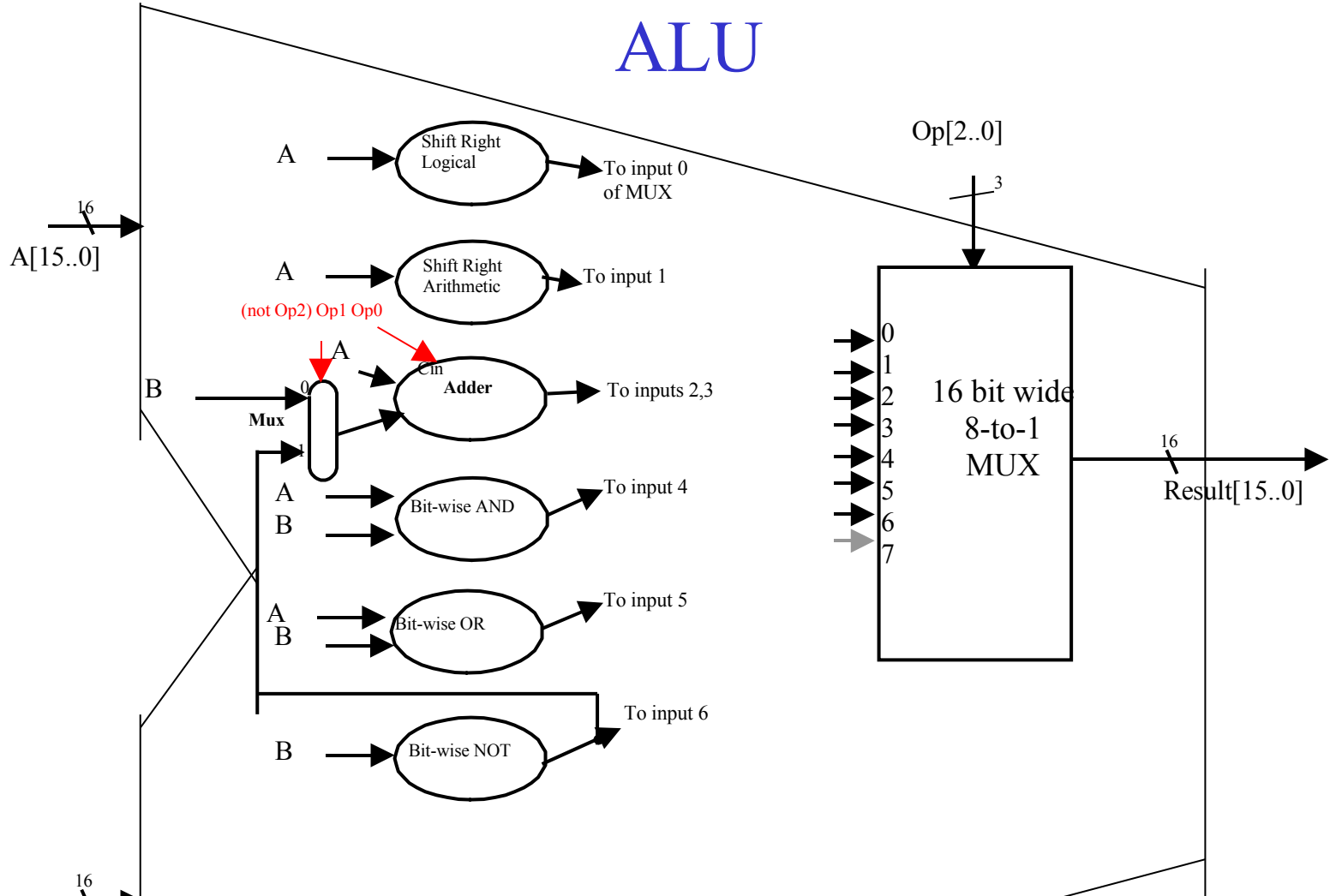
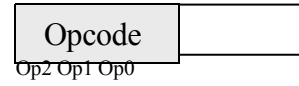
Finally, you must put all these together with the multiplexer as shown in the diagram at the end of this document. The inputs and outputs to the ALU module should be as shown on the diagram, i.e., the inputs should be **A[15..0]**, **B[15..0]**, and **Op[2..0]**, and the output must be **Result[15..0]**. The design of the basic components of the ALU has been given above, as well as the method for connecting them all together with buses. Note that you will need to use two multiplexers, one for the input to the adder (depending on whether you are adding or subtracting), and a big one for doing the selection of the output. Call this final circuit **alu.bdf**.

Instructions for submission and testing of the ALU design will be given in the upcoming Homework

### How about subtraction?

For addition (**add**), the Carry In line to the low-order bit (which is the Carry In noted on the ALU diagram) is set to 0. For subtraction (**sub**), according to the rules for two's complement arithmetic, we can calculate A-B by adding 1 plus A plus the bit-wise inverse of B. The 1 can be introduced via the Carry In line to the adder. Thus, by examination of the opcodes, we see that the logical formula (**not Op2**) and **Op1 and Op0** will produce the appropriate Carry In value. The Cout of the high-order bit is ignored. Do not worry about overflow.

### Instruction Format:



### Notes:

- All arrows (except for red control lines and opcode input) are 16 bits wide.
- Unused data input for MUX (7) should be tied to Gnd or to an arbitrary device—it is not used, so it doesn't matter, but all inputs to a device must be hooked to something or you will get a compilation error in QuartusII.
- **(not Op2) and Op1 and Op0**; you will need to create this small circuit (not shown), which outputs a 1 when the opcode indicates a subtraction, as part of the ALU design. Note how this circuit both inputs a 1 to the carry-in of the adder, and selects the bit-wise NOT of B as the input to the adder.