Lab 7 - VHDL

Arithmetic and Logic Unit

What to hand in

1. VHDL file for full Adder (.vhdl)
2. VHDL file for 16 bit ripple carry adder (.vhdl)
3. Archive of ALU project (.qar) - Must be called Team"Number"_lab7.qar
   For example, team 6 would call theirs Team6_lab7.qar
4. .do files from each task
5. .bmp for each task of the simulation

Pre-lab

1. When adding two binary numbers, we know that for each bit, the sum and carry out can be determined with the following expressions:

$$S = A_i \oplus B_i \oplus C_{in}$$

$$C_{out} = A_i * B_i + A_i * C_{in} + B_i * C_{in}$$

   Draw the gate diagram for both S and C_{out}.

2. Congratulations, you just made a 1-bit full adder. Now, given inputs A[3..0], B[3..0], and Cin, use multiple full adders to create a 4-bit adder. The outputs should be S[3..0] and C_{out}. This will require a though process like last weeks final pre-lab question.

3. What should C_{in} be in each of the following cases. You will have to think of the purpose of C_{in}

   (a) Computing A + B

   (b) Computing A - B, or A + -B

Task 1

Given that you wrote a full adder for prelab, implement a full adder in VHDL. Make sure to write a .do file and to test it to make sure your logic is correct.

Part 2

Now we are going to use a VHDL file as part of another VHDL file. Some VHDL files require the use of other entity's. An example, albeit a completely unnecessary one, is a 16 bit AND. Let's assume that AND in VHDL can only take 1 bit on either side. If this is true, then to AND 16 bits we would have to write the logic of 16 different AND commands like so:

```
output(0) <= A(0) AND b(0);
output(1) <= A(1) AND b(1);
output(2) <= A(2) AND b(2);
....
....
output(15) <= A(15) AND b(15);
```

Now instead of using functions like above, we can instead use that AND gate we created back in lab 5. Remember that the AND gate was an entity call andGate. Now to use that andGate inside of another VHDL file, we have to declare it as a component inside that file. A component is declared inside another entity inside of the ARCHITECTURE sections. If you are confused about this, look at the below example.

```
ARCHITECTURE behavior OF and16 IS
  COMPONENT andGate
    PORT(
      a,b :IN STD_LOGIC;
      f   :OUT STD_LOGIC
    );
  END COMPONENT;
BEGIN
...
END
```

Now you can see that the component is declared where you normally declare signals. You can declare as many components and signals as you see fit. The actual declaration requires a few things. It must start with COMPONENT, the component name **must** match the name of the other file, the PORT names should be the same, and it must end with END COMPONENT. **NOTE:** this component's .VHDL file must be located in the same directory as the main .VHDL file.

Now actually using these components is not difficult. In VHDL you do something called PORT Mapping. All this means is that in the main VHDL, where something is declared as a Component, you will make the inputs and outputs of a certain component be I/O's from that file. An example to further explain this is on the next page:

```
ARCHITECTURE behavior OF and16 IS
  COMPONENT andGate
    PORT(
      a,b :IN STD_LOGIC;
      f :OUT STD_LOGIC
    );
  END COMPONENT;
BEGIN
  and0: andGate PORT MAP(A(0), B(0), F(0));
  and1: andGate PORT MAP(A(1), B(1), F(1));
  ...
  ...
  and15: andGate PORT MAP(A(15), B(15), F(15));
END
```

Port mapping is only done between the BEGIN and END of the ARCHITECTURE. The first word, e.g. andO, is a name that you choose. You can call this whatever you want. This name ends with a colon. The next word is the name of the component you want to use. The above example uses andGate as that is the component we care about. The next two words are PORT and MAP. PORT and MAP are seperated by a space. Immediately following MAP, with no spaces, is a set of parenthesis. Inside that parenthesis' is what you want to map to. The order you map to is top to bottom. Since we declared the inputs of andGate first, those are the things we map to first. Then we will map to the outputs. It follows from left to right if you list all the I/O's on one line. This means we map to a first and then b.

Of course, this example is a little bit unnecessary. The true power of components comes with complex gate diagrams. The ability to abstract away the complex code makes the VHDL much easier to write. You can keep abstracting upwards if you start with a simple entity. This is akin to how the processor can be built in VHDL. The first file you could write would be a multiplexor. This can be used in the Register File, which is then used in the datapath, which is then included in stage 1, and finally this is included in the main processor file.

Task 2

Using your knowledge of components, you should be able to create a 16 bit ripple carry adder. Your 16 bit adder should have the inputs A, B, and Cin with A and B being 16 bits and Cin being 1 bit. It should also have the output F, C14, and C15 where F is the output of A+B and is 16 bits, and C14/C15 are the last two carry outs. These will be useful later on to have already outputted. **You should use signals as the carry outs between two bits.**

Part 3

For this lab, since we are just learning about components, we will use a block diagram for the rest of the ALU. What we must do first is create a symbol file for the 16 bit adder we just made. You can create a symbol file for any VHDL file by using the following steps:

1. Open the .VHDL file in quartus.
2. Make sure it is located in the same area as your project
3. With the file you want to create a .bsf of open and selected do the following steps:
   (a) Click on file
   (b) Click on create/update
   (c) Click on create symbol file for current file
   (d) Wait for compilation to finish and you are done

   Please note that if there is an error with your .VHDL file when you try to create a symbol file, it will stop making it and let you know the problem

After you have created the file, you can use the symbol inside the a block diagram by clicking on the the component button, see figure 1, and then clicking on project, see figure 2, and finally clicking on the diagram and placing it on your block diagram, see figure 3.

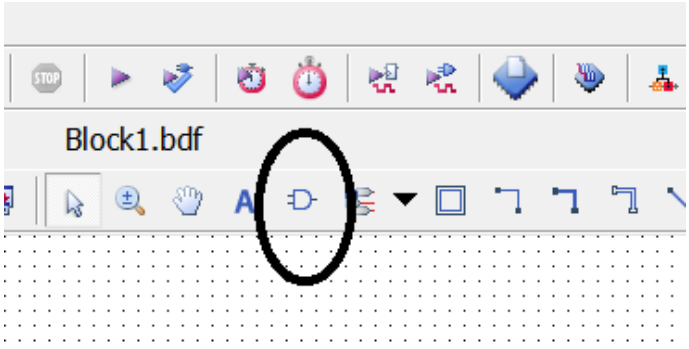

Figure 1: The component button

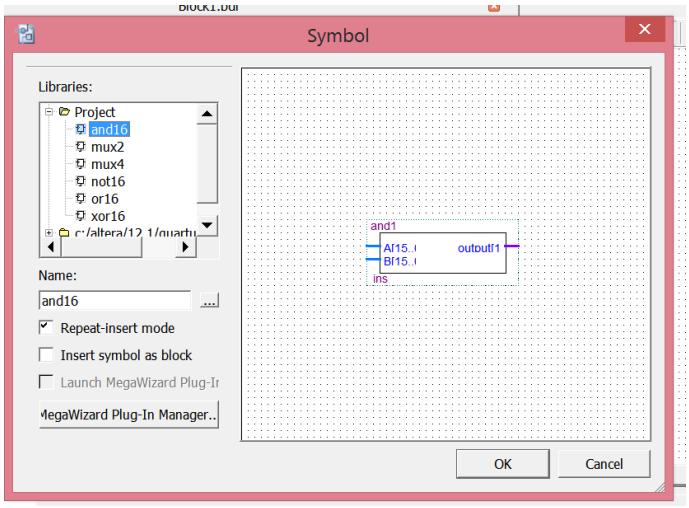

Figure 2: And16 selected

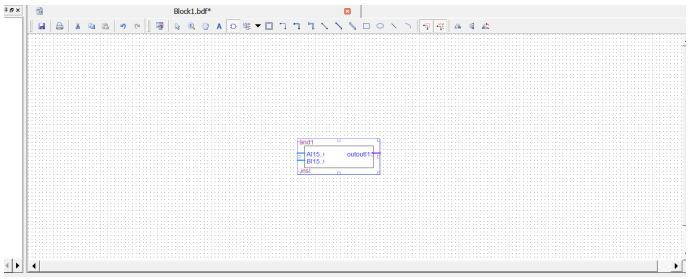

Figure 3: The block symbol in the file

Task 3

For the main part of this lab you must create a fully operational 16 bit ALU. We will be using quartus to handle most of the wire inter-conenctions this week as components are a relatively new concept to most people. The ALU must be able to perform Add, Sub, AND, OR, and XOR. The ALU must also be able to output the status of the flags N,C,V,Z. The logic of the various flags is in the table below:

N	Bit 15 is 1
C	Carry 15 is 1
Z	All bits are zero
V	Carry14 XOR Carry15

You can implement the various flags in a .VHDL file that has inputs S, 16 bit output of 16 bit adder, and C14 and C15, 1 bit output from 16 bit adder. The outputs should be N, C, V, and Z.

Figure 4 shows what your ALU should look like in the end.

As a quick reference, the inputs and outputs of the ALU are below:

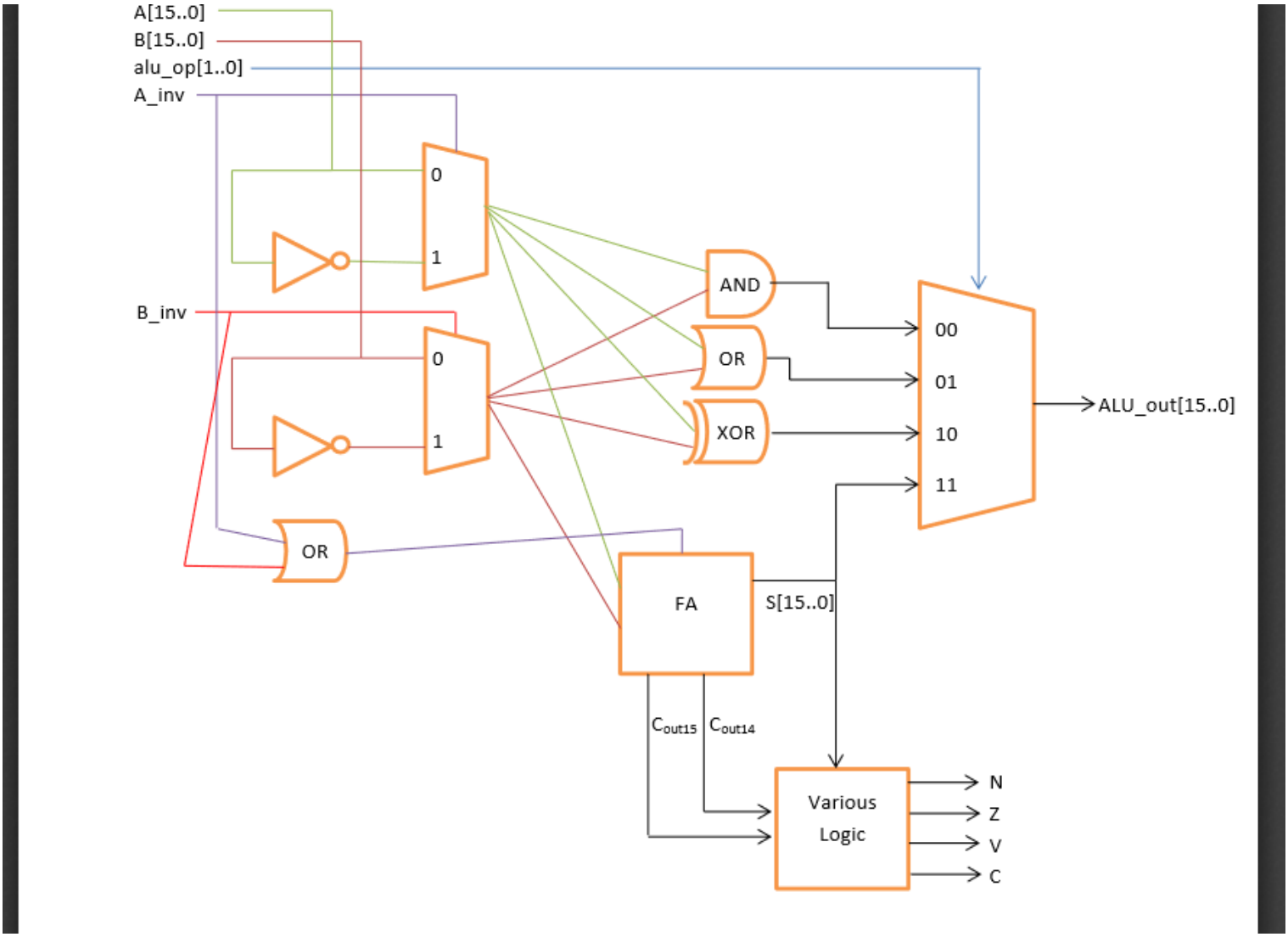

Figure 4: The ALU as a block diagram

INPUTS	OUTPUTS
A[15..0]	ALU_out[15.0]
B[15..0]	N
A_inv	Z
B_inv	C
alu_op[1..0]	V

Make sure to test you ALU with different values of A and B testing each operation. That is, if we give A and B a specific value, make sure to test each possible output of those two. Test different values of A and B with all possible outputs. Alu_op is the selector for the final mux. This should vary from 00 to 11.