EEL3705L, Digital Logic Design Lab, Spring 2011

**Pre-Lab #4**

For

Lab Assignment #4:

Design a Modular Arithmetic-Logic Unit

**Document Description:** This report details the steps taken to design a four-function Arithmetic Logic Unit (ALU). This ALU is to perform addition, subtraction, bitwise OR, and bitwise AND on two eight-bit binary numbers, and display them into a human readable form. This design is for lab number four of the spring 2011 EEL3705 class at Florida State University.

* **Introduction**

Students are required to design and implement an Arithmetic Logic Unit that they will design from scratch. The logic that is to be implemented by these ALUs will be decided by the teacher. For our section, the ALU must perform addition, subtraction, bitwise OR, and bitwise AND of the two input numbers. Both input numbers will be eight-bits long, and there will be a two-bit selector number for the ALU to use to select which operation is to be done.

* **Requirements**

1. The two input numbers must be eight-bit two’s complement binary numbers.
2. The ALU must perform four separate operations on the two input numbers. In this case, it is addition, subtraction, bitwise OR, and bitwise AND.
3. Students cannot use the megafunction lpm\_add\_sub for their addition and subtraction of the two input numbers.
4. The numbers must be displayed correctly in a human-readable form, and there should be an indication if a signed overflow occurred.

* **Theory and Design**

The table for the function of the ALU with reference to the selector number, made up of bits s1s0, is shown below.

|  |  |  |
| --- | --- | --- |
| s1 | s0 | ALU operation |
| 0 | 0 | +(addition) |
| 0 | 1 | -(subtraction) |
| 1 | 0 | &(AND) |
| 1 | 1 | |(OR) |

The AND and OR functions of the ALU will be the simplest to design and implement. For two input numbers a[7..0] and b[7..0], we would take a significant bit of the first one, and AND it with the corresponding bit of the second one, if we were doing the AND operation. For example, a0 would be joined with b0 in an AND gate. We would do the same process for the OR operation, replacing the AND gate with an OR gate. We could then use a 2-to-1 multiplexer with the selector bit as s0 to determine which operation will be going forward to be multiplexed with the addition/subtraction operation.

In class, we learned that we can design a ripple-carry adder to that it can add or subtract these numbers if there is a bit controlling the function. We can see here that the bit controlling whether the number will be added or subtracted is s0. So, we can use such a design here, with components coming afterwards to convert the binary to a decimal-readable notation.

The top-level block diagram for the ALU is given on the next page.

8-bit binary to HEX display decimal output

Selector for bitwise operations

Module for & operation

First 8-bit input binary number

Module for | operation

Second 8-bit input binary number

Selector for bitwise or arithmetic operation

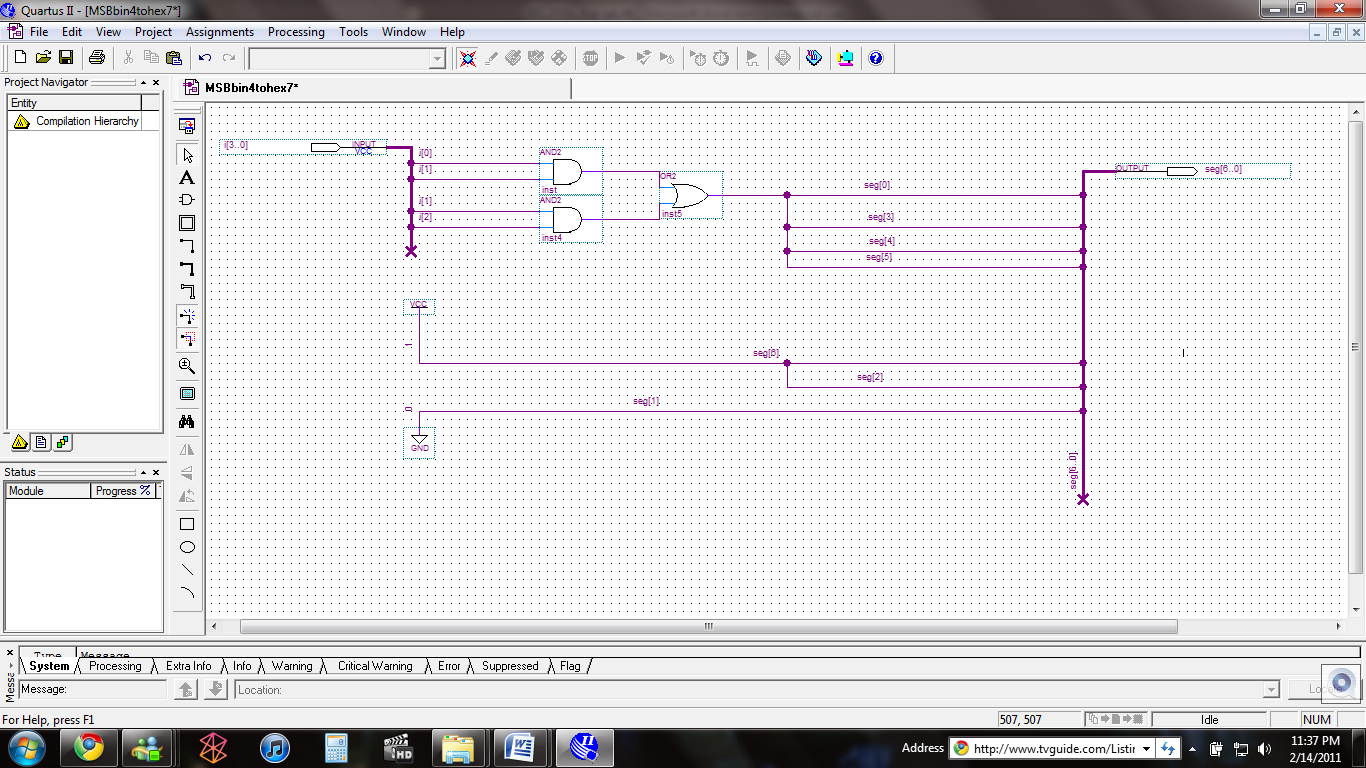
HEX displays

Module to display either error message or the binary number in decimal notation

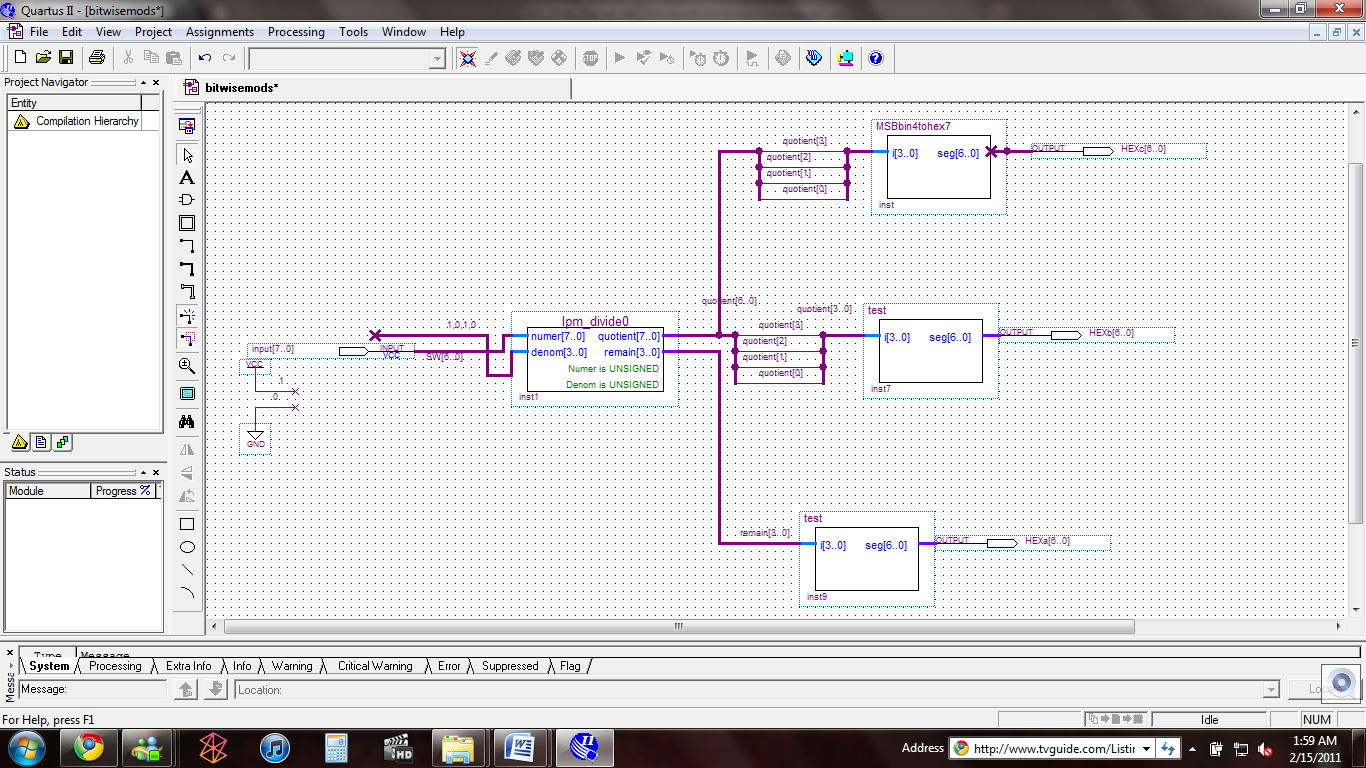
Addition/Subtraction Unit

Because we have selector bits, the “selector for …” blocks can be implemented with a multiplexer. For the bitwise functions, the module to convert the binary to decimal is very close to what was designed in the last lab. However, there is a third decimal number that will show up, so a function will need to be added to handle that. Because the largest number that can be produced is 12710, the new decimal bit will only be a one or a zero. For this, segment 6 will always be off, and segments 1 and 2 will always be on. The K-map below shows that segments 0, 2, 3, 4, and 5 will be controlled by where i0 is the least significant bit of the input number and 12 is the third least significant bit.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| i0i1  i2i3 | 00 | 01 | 11 | 10 |
| 00 | 0 | 0 | 1 | 0 |
| 01 | 0 | 0 | 1 | 0 |
| 11 | 0 | 0 | 1 | 1 |
| 10 | 0 | 0 | 1 | 1 |

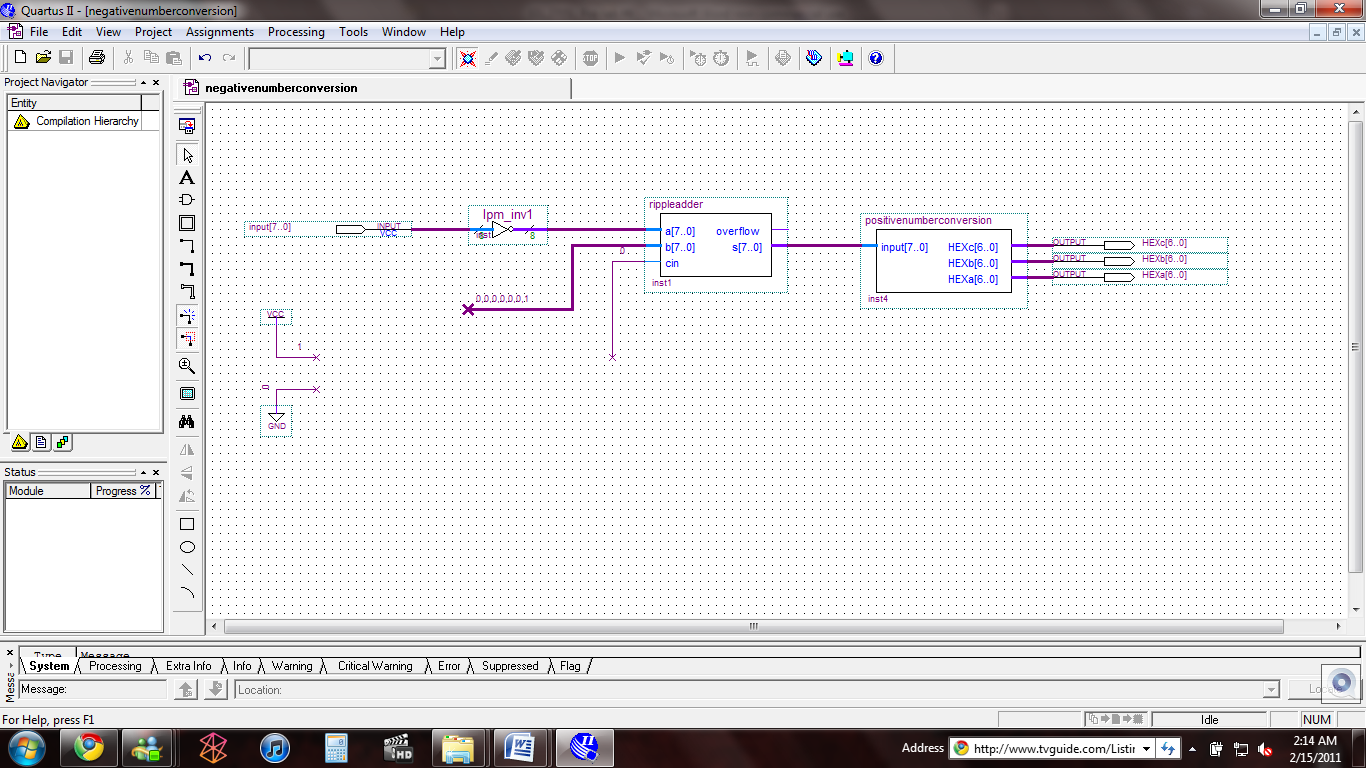


Because the result could be either a positive or a negative number, we can split these two cases apart from one another by a 2-to-1 mux with the most significant bit in as the control bit. For the positive number, we can re-use some of the designs from the last lab, with the exception that we need to add another Hex display, since the numbers can now go up to 12710. The Quartus representation for this function is given below.



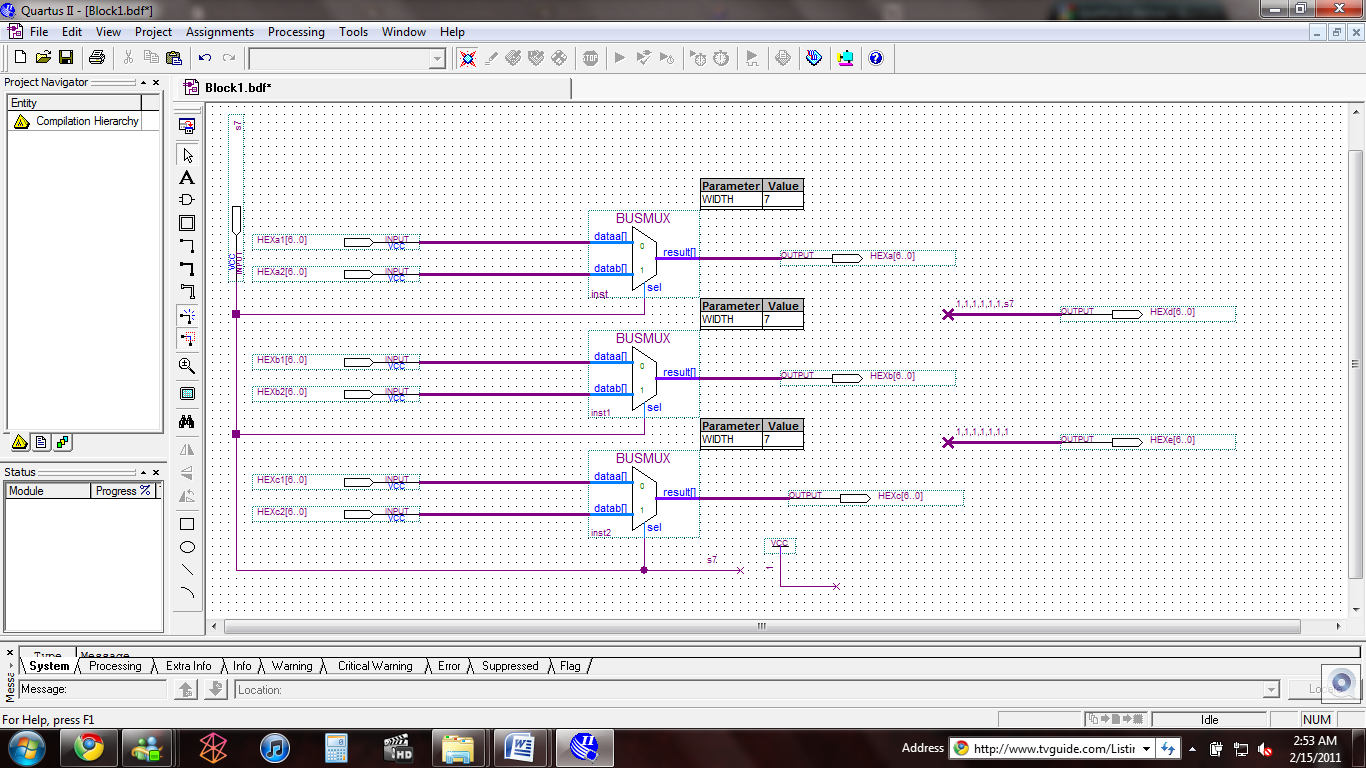
This was made into a symbol block called “positivenumberconversion”.

If the number was negative, we could get the absolute value of the 2’s complement number by complementing the number and then adding 1. We would then feed this into a “positivenumberconversion” block. This is shown in Quartus below.



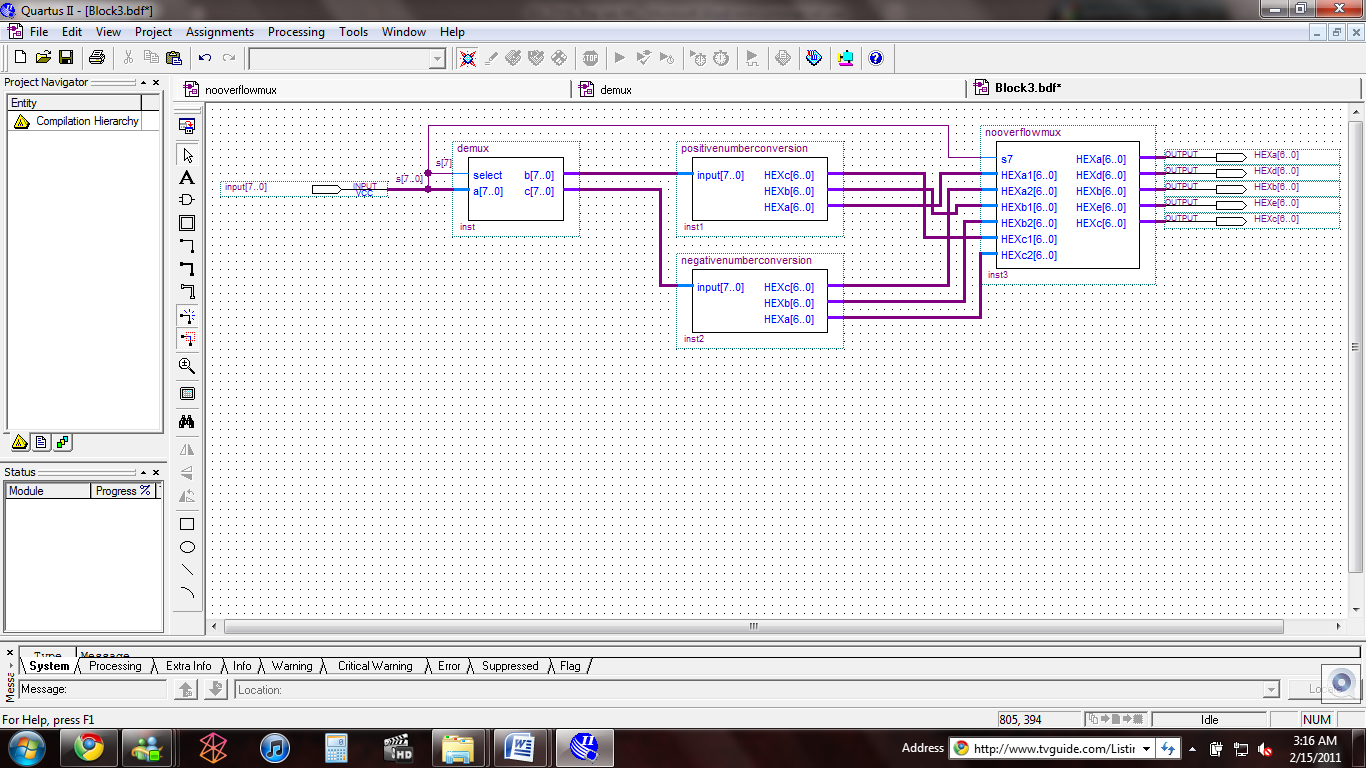
This was made into a block called “negativenumberconversion”.

Now all that is needed is to have a 2-to-1 mux for HEXa, HEXb, and HEXc. We can notice that HEXd will only be affected by the most significant bit of the bitwise result, and that HEXe will always be off. Below is the Quartus representation of this.



A block was made for this called “nooverflowmux”.

Now, we just need to use a demux with the input’s most significant digit as the select bit to split the number into positive and negative numbers. The outputs of the demux will go to “positivenumberconversion” and “negativenumberconversion” respectively. We can then use the “nooverflowmux” to re-combine them, with input’s most significant digit as the select bit, and give us the correct outputs to the HEX displays. This is shown in the image below.

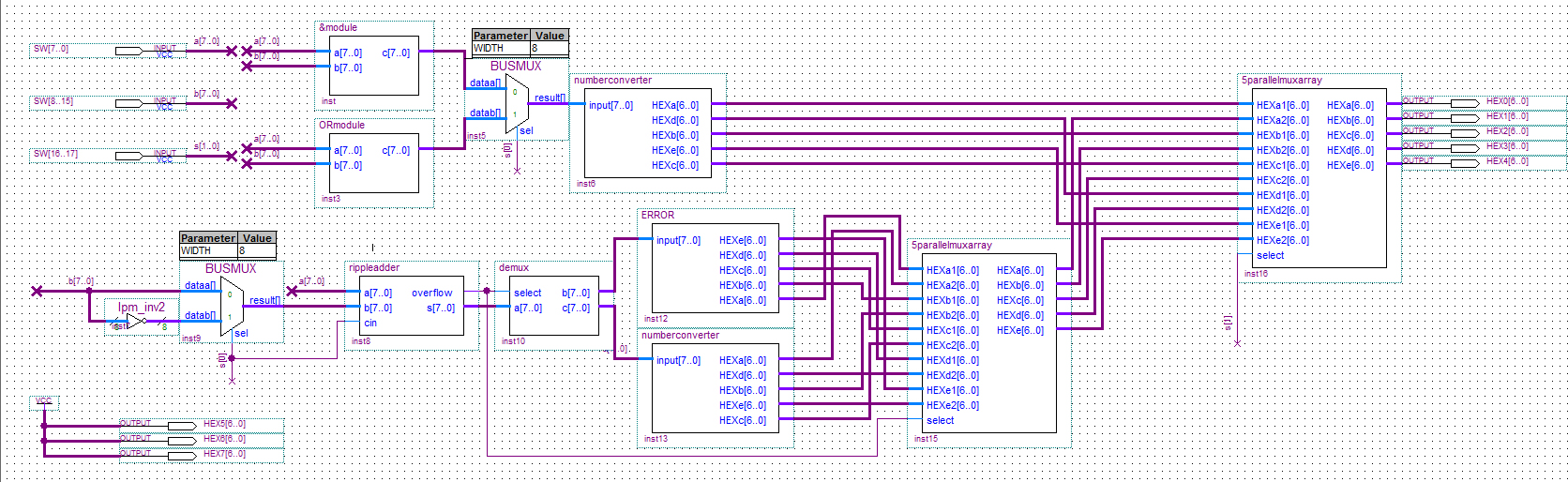


This was put into its own block called “numberconverter”.

The addition/subtraction module we will use was demonstrated in class, and is made of a megafunction inverter, a 2-to-1 mux, and a 8-bit ripple-carry adder made up of eight full adders whose carry is input into the next highest significant bits’ input. We will output the last carry bit and the XOR of the last two carry bits, the signed overflow indicator, as they will be needed later.

If our circuit was to have a signed overflow, we decided that the simplest method would be to simply display “Error” on the seven-segment displays rather than extending the binary number one bit and translating it. This can be accomplished with a 1-to-2 demultiplexer with the signed overflow indicator bit as the selector bit, and the first output going to the module to create the message. The second input can go into another “numberconverter” symbol file. In order to have only one, correct output for either an overflow or no overflow possibilities, we must mux the corresponding outputs of the “ERROR” module and “numberconverter” module with the signed overflow indicator bit as the select bit.

Now, with the arithmetic and bitwise functions done, we just mux the HEX outputs of each with s1 as the select bit. The complete Quartus circuit is shown below.



* **Design of Prototype Testing Experiments**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| s1 | s0 | Test at least two different inputs, one result with an overflow (if applicable) and one without. Draw out results by hand and compare. Did they match? (Y/N) | What were the input numbers? | If there was not a match, what output was observed? |
| 0 | 0 |  |  |  |
| 0 | 1 |  |  |  |
| 1 | 0 |  |  |  |
| 1 | 1 |  |  |  |

* **Answers to Questions**

1. Appropriate inputs for each of the eight-bit input numbers will be sixteen of the DE2 switches, eight for each number.
2. A human-readable output would be signed decimal representation on the seven-segment HEX displays of the DE2 board. We will actually only use five of them at most.
3. One of the seven segment displays will light up the middle segment for a negative number, or leave the whole display blank for a positive number.
4. If the result is out of the range of the eight-bit output, the seven segment displays will show the word “Error” on them.
5. If a signed overflow is detected, “Error” will appear on the displays, since the eight-bit result will be wrong. Other overflows will be truncated.
6. There was nothing to be answered on this question.
7. There will be a two-bit binary selector code for the operations, with the code containing s1 and s0. The truth table for the operations is below.

|  |  |  |
| --- | --- | --- |
| s1 | s0 | ALU operation |
| 0 | 0 | +(addition) |
| 0 | 1 | -(subtraction) |
| 1 | 0 | &(AND) |
| 1 | 1 | |(OR) |

1. The top-level design for the circuit is shown in the Theory/Design section. Multiplexers and demultiplexers will allow us to pick and choose what functions to output.
2. Our adder/subtractor is a simple 8-bit ripple carry adder with a multiplexed second input to differentiate between adding or subtracting.

* **Conclusion**

This lab really showed the real complexity that just adding a sign creates. And since we were using 2’s complement, it was as simple of a representation that we could use for simple hardware. Having to track whether a number is positive or negative not only added difficulty in the arithmetic, but also in the translation to a human-readable form.

I also noticed just how convenient a mux or demux really is. They seem to perform like a programming IF statement, and really simplify a circuit.