# Digital Logic: Project 1

Anderson, Anna anna.anderson@colorado.edu
Cuthbertson, Sam samuel.cuthbertson@colorado.edu

March 9, 2017

## 1 Introduction

Beginning to understand how to code Verilog and start thinking as a digital designer is a large task. The point of this lab was to begin to gain a familiarity with a broad range of Verilog commands, explore the differences between procedural versus continuous assignment, and learn about good practices when it comes to digital design. Some of these practices included starting well. Constructing detailed block diagrams from the very beginning not only helped our personal understanding or brought to light conceptual places we were struggling, but it implicitly outlined the code that we ended up writing. This helped the development process.

Beginning at Figure 1, we began working through the lab conceptually. Though this changed slightly after we began coding, it is still largely accurate. Our first seven switches go into the specific modules that we made: arithmetic, logical, and comparison. These switches enabled us to perform each of the functions that the individual modules required. However, switches 8 and 9 determine which state displays on the seven segment display and LED. For example, in comparison, these switches chose between equal to, greater than, less than, and max.

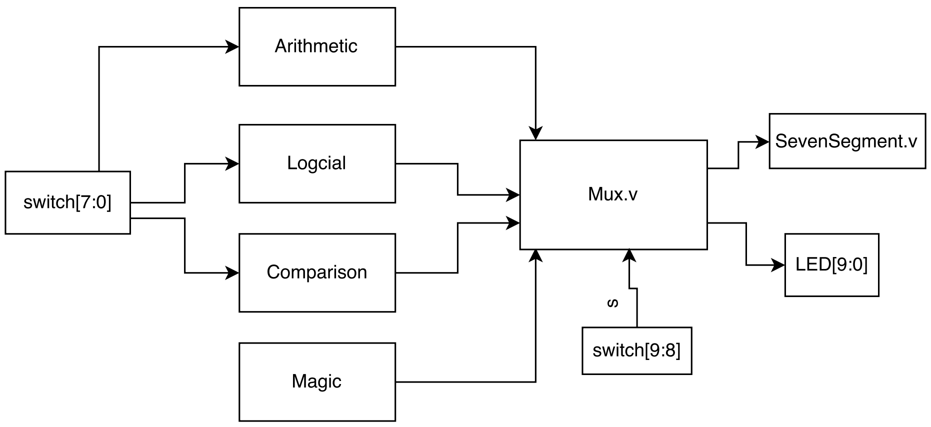

Figure 1: Block Diagram of Top Module

## 2 Procedures

### 2.1 Conceptual to Generated

#### 2.1.1 Logical and General Verilog Design

We began with the logical module (See Figure 15). Drawing this module out was very straight forward which is why, after drawing the rest of the modules, we began by coding this one first. This proved to be a good choice as it brought to light some important conceptual ideas that propagated through to the other modules. For one, figuring out how to pass information out of the module and have it received by other modules turned out to be a sticking point. Realizing that we could and should treat anything outside of the module as though we were writing the module as a black box is what ended up making the development process much easier. Pass in 8 bits of information. Pass out 8 bits of information, or whatever was necessary for the particular module. This simplified and streamlined our process a great deal, and is very apparent when viewing our code in the form of a block diagram in Figure 18.

### 2.1.2 Comparison, Arithmetic and Block Diagramming Efficiently

Returning to drawing our block diagrams, we moved on to comparison next (In Figure 16). We opted for drawing boxed indicating the function that we wanted, as it was advised that Verilog had built in commands for things such as greater than and less than. This allowed us to not over-complicate our process. Thus moving on to arithmetic (In Figure 14), we needed to take into account the LED that should turn on if a carry or remainder was produced. This resulted in a slightly more complicated block diagram.

In this process, it was determined that we should continue to opt for the blocks containing functions such as full adder and subtractor as we realized that the equations given in class were our best bet to constructing this module, and that spending time on block diagramming them in detail would simply be copying down our lecture notes. Additionally, it should be mentioned that we did not build a seven segment display module. As we were thinking through it, we wanted to use case statements in an always statement and did not know how to make a block diagram of it. Under this same reasoning we decided not to make a block diagram of our multiplexer. We still talked about the inputs and outputs that went into it, and being able to physically code it proved to be much more conceptually helpful for us.

### 2.1.3 Coding our Diagrams

After we finished diagramming we moved on to coding in Quartus. From our initial ideas, we did find that some things ended up changing. A significant one ended up being how we processed wanting to use all eight switches under a variable z. Instead of taking z as an input, z was assigned a wire in which the beginning bits were taken from the switches associated with x and the ending ones from those associated with y. This can be seen rather clearly in our generated block diagrams for Logical (Figure 18) and for Arithmetic (Figure 19), as well as our code in `Logical.v` (lines 7-9) and `Arithmetic.v` (also lines 7-9).

Our arithmetic module (`Arithmetic.v`) largely looked as we expected it to. The addition and subtraction modules inside of it this followed the expected format that we went over in class. In the `fullAdd` module (lines 39-57) we have a `generate` statement that creates a new instantiation of `module bAdd` until we have our N-bit adder, where the `module bAdd` utilizes the equations covered in class. Inside of `fullAdd`, we have a wire for our carries in order to pass them down the line of `bAdd` instantiations. This also allows us to return our carry out, as it will be the most significant bit in this wire (line 56).

However, our subtraction modules exploited `integer i` in it's for loop (line 69). Subtraction proved to be conceptually harder on the front end. However, it's module is shorter. Inside the always block, the difference and borrow wire are iterated over in the `for` loop on line 75. This is extremely clear as gates in Figure 19. In this way we still create an N bit subtractor, however it's all contained within the single module.

The multiplication and division modules were much simpler. Each represented a a bit shift in either direction. For the multiplication module, we assigned the carry to the last bit of our input, `z`, and the first bit of our output to zero, representing the shift (lines 91-92). The rest of output bits were assigned to bits six through zero of our input (line 93). The opposite was done for division. The lowest bit of the input was set to the remainder, while the highest was set to zero (lines 102-103).

The comparison module exactly followed the format that we had outlined in our block diagrams. Less than, greater than, and equal to all represented a keyword for a circuit in Verilog, greatly simplifying the process. Inside comparison, the `bMax` module was the only one that we ended up needing to modify from the simple box represented in the diagram, as an always statement made much more sense to us after some experience with Verilog. To implement `bMax`, we made an instantiation of our `bGreater` (line 48), assigning it's output to a wire. That wire is then referenced within a case statement that allows us to output the value of either x or y, depending on whether x is greater than y. This is much clearer in the genereated block diagram in Figure 20

However, Magic was not implemented in the same way that we originally had demonstrated in Figure 1. Magic ended up bypassing that selector multiplexer altogether, which we'll cover later.

## 2.2 Encoding Schemes

### 2.2.1 Button Modes

For the buttons to swap between Arithmetic/Comparison/Logical/Magic, we created a `state` register which would toggle between 0 and 1 on `posedge` of the corresponding button: i.e, `KEY[0]` would toggle `state[0]` and `KEY[1]` would toggle `state[1]`. For the mode selectors within Arithmetic/Comparison/Logical, we used the direct value from `SW[9:8]`.

| Top | |
|---|---|
| state | Output |
| 00 | Arithmetic |
| 01 | Logical |
| 10 | Comparison |
| 11 | Magic |

| Arithmetic | |
|---|---|
| selector | Output |
| 00 | Addition(x,y) |
| 01 | Subtraction(x,y) |
| 10 | Multiply by 2(z) |
| 11 | Divide by 2(z) |

| Comparison | |
|---|---|
| selector | Output |
| 00 | Equal(x,y) |
| 01 | XGreater(x,y) |
| 10 | XLess(x,y) |
| 11 | Max(x,y) |

| Logical | |
|---|---|
| selector | Output |
| 00 | And(x,y) |
| 01 | Or(x,y) |
| 10 | xOr(x,y) |
| 11 | Not(z) |

### 2.2.2 Wire Arrangements in Top.v

For the output from each of Arithmetic/Comparison/Logical, we expected the ordering shown below.

| Bit Count | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| Output Value | Lower 7seg | | | | Higher 7seg | | | | LED9 |

We also hard coded the input from magic to our Mux to be $1'b0$, which would set both seven segment displays to 0 for the magic mode. Additionally, the output from the magic module was sent to the LEDs only when our `state` register was 11.

Finally, our `state` register is also used as input to our `SevenSegmentMode` module in order to display the current mode we are in, as one of A, L, C or M for magic.

## 3 Results

### 3.1 Output Pictures

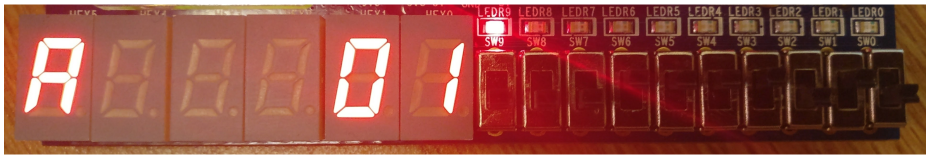

Figure 2: Our output for adding 0xA and 0x7, showing that it is 0x11

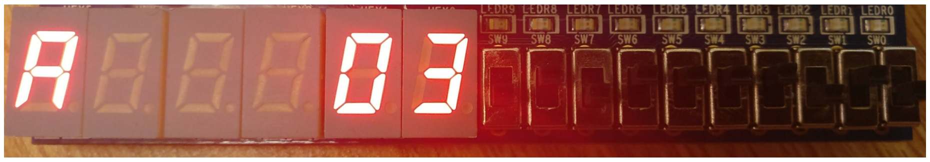

Figure 3: Our output for subtracting 0x7 from 0xA, showing that it is 0x3

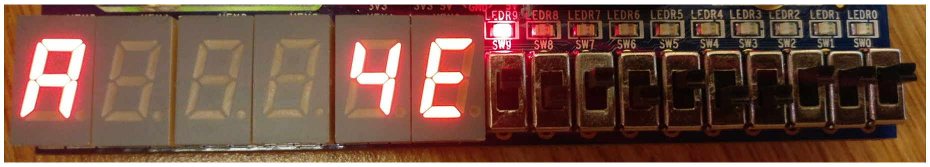

Figure 4: Our output for multiplying 0xA7 by 2, showing that it is 0x14E

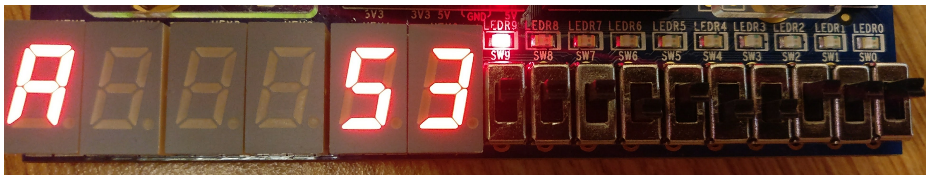

Figure 5: Our output for dividing 0xA7 by 2, showing that it is 0x53 and a half

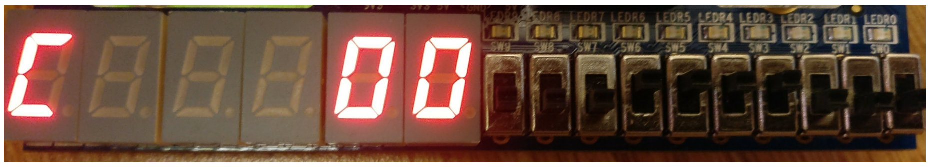

Figure 6: Our output for seeing if 0x8 and 0x7 are equal, showing that they are not.

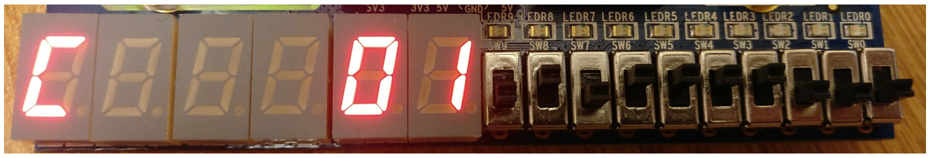

Figure 7: Our output for seeing if 0x8 is greater than 0x7, showing that it is.

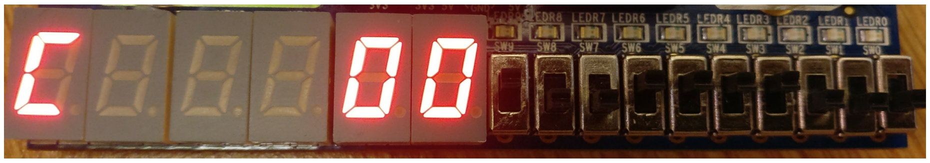

Figure 8: Our output for seeing if 0x8 is less than 0x7, showing that it is not.

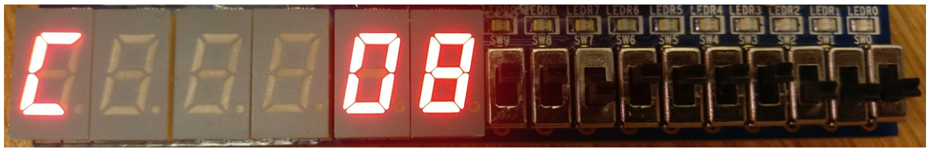

Figure 9: Our output for returning the max of 0x8 and 0x7, showing that it is 0x8.

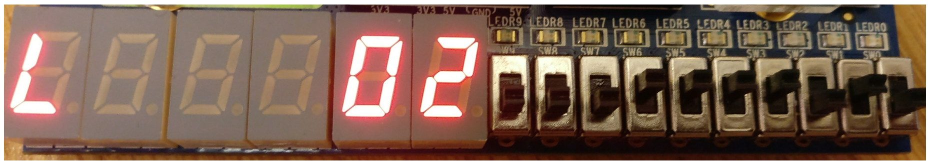

Figure 10: Our output for the bitwise and of 0xA and 0x7, showing that it is 0x2

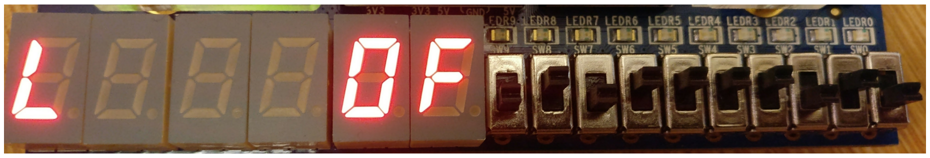

Figure 11: Our output for the bitwise or of 0xA and 0x7, showing that it is 0xF

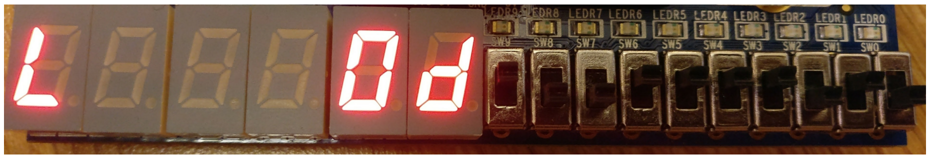

Figure 12: Our output for the bitwise exclusive or of 0xA and 0x7, showing that it is 0xd

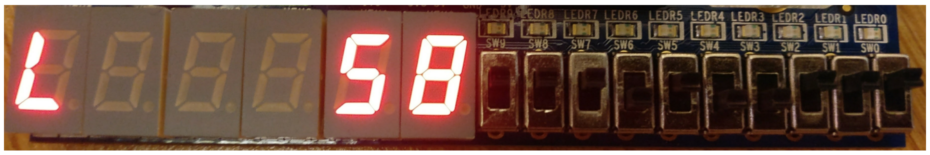

Figure 13: Our output for the bitwise not of 0xA7, showing that it is 0x53

## 4 Conclusions

This project was quite challenging in that we had no previous exposure to writing Verilog, using Quartus, or even necessarily having developed a solid intuition for digital design. An important part of the intuition we ended up developing was the idea that information is always being updated. We as a team struggled in that our default intuition is related to high level languages. It didn't matter if we pumped the switches into the code before we'd set a mode because they would be constantly updating anyway. However, this was a very good introductory project to expose us to a lot of really fascinating and useful functions. Displaying the generated block diagrams and exploring through them was quite satisfying and proved to solidify some of the concepts. The biggest bug we ran into was in making Magic. While it does work, in our time limit we were not able to move away from higher level thinking. Thus, magic is highly inefficient and "hacky", to quote Professor Fosdick. If given more time, we would redo magic to more explicitly take into account the system we are programming for. In general, we did not run into any glaring error or bugs that we could not resolve. The largest error that we worked on were syntax related. For example, the code written on the board during class had a couple very significant syntax errors. The order given in class read input/output, variable name, and then size. However, as it turns out, it actually is really important that size comes before the variable names.

## 5 Conceptual Block Diagrams Appendix

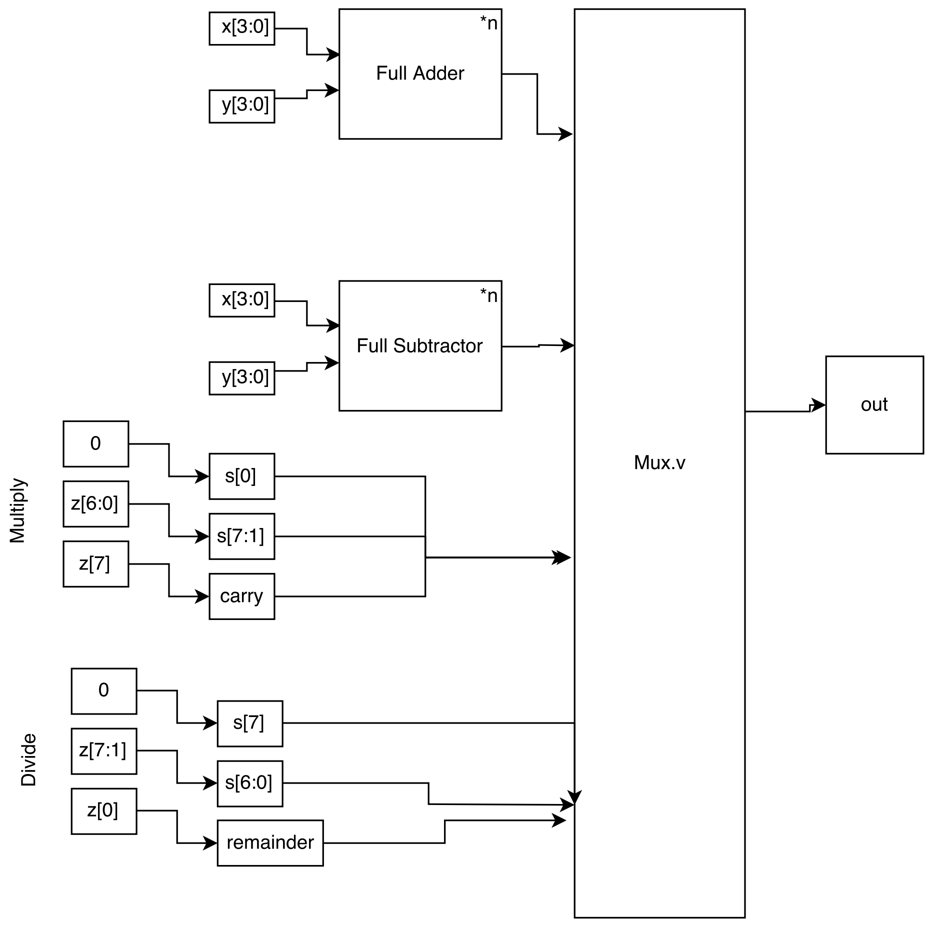

Figure 14: Block Diagram of Arithmetic Mode

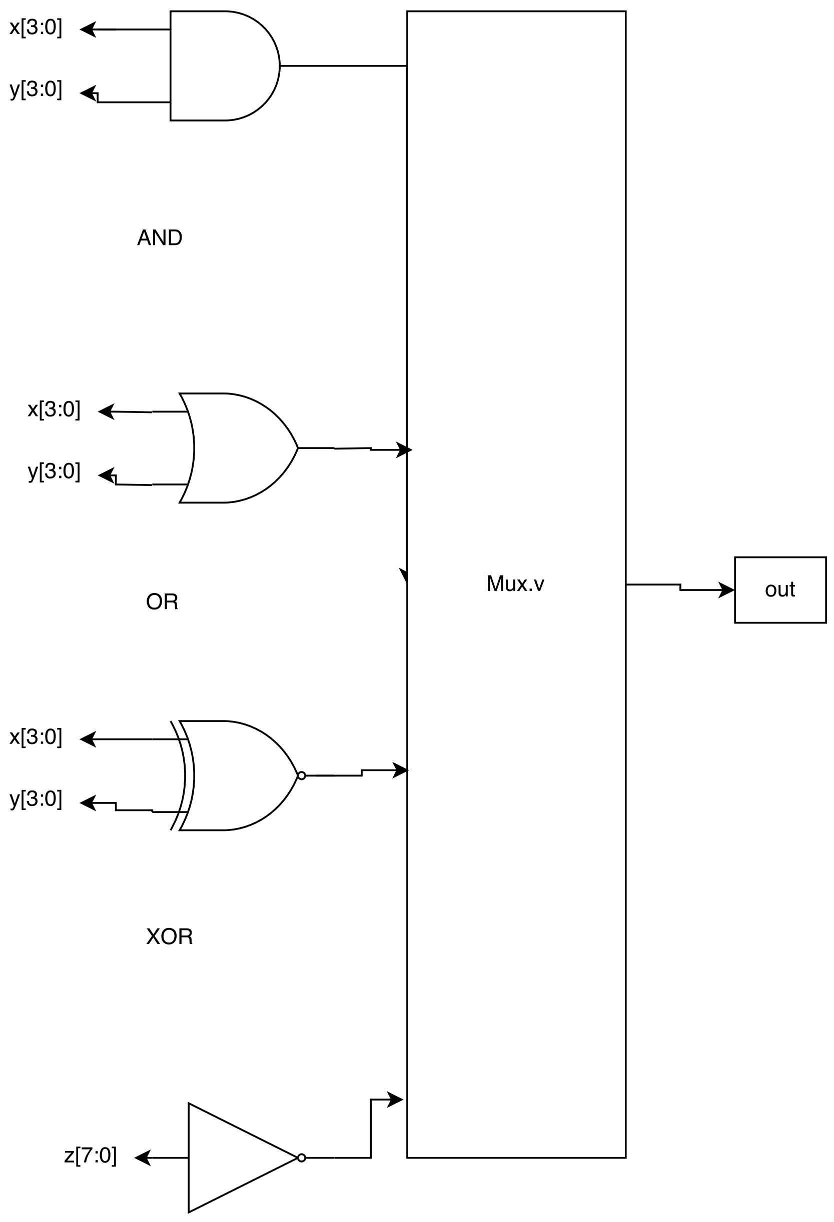

Figure 15: Block Diagram of Logical Mode

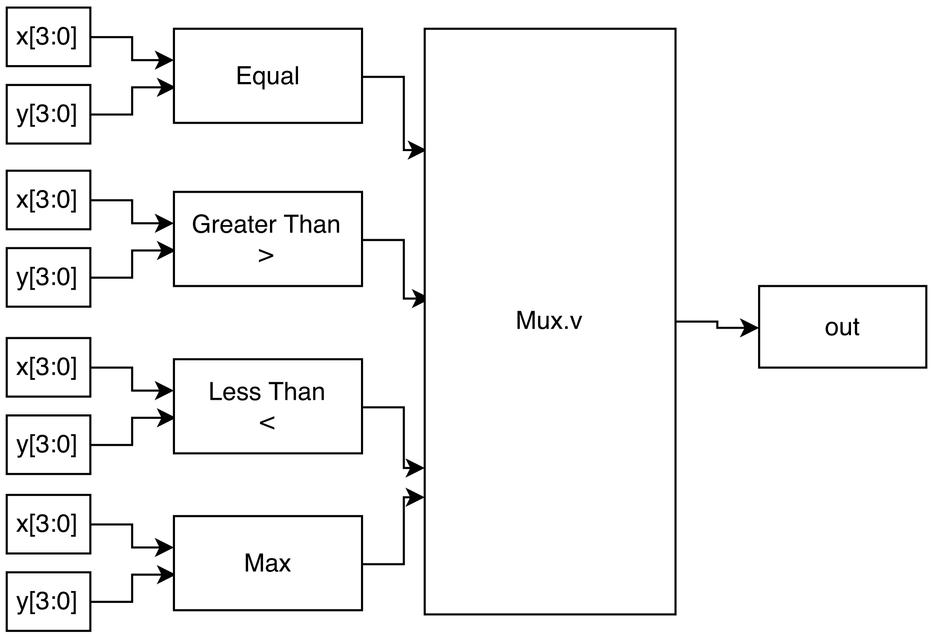

Figure 16: Block Diagram of Comparison Mode

## 6 Generated Block Diagrams Appendix

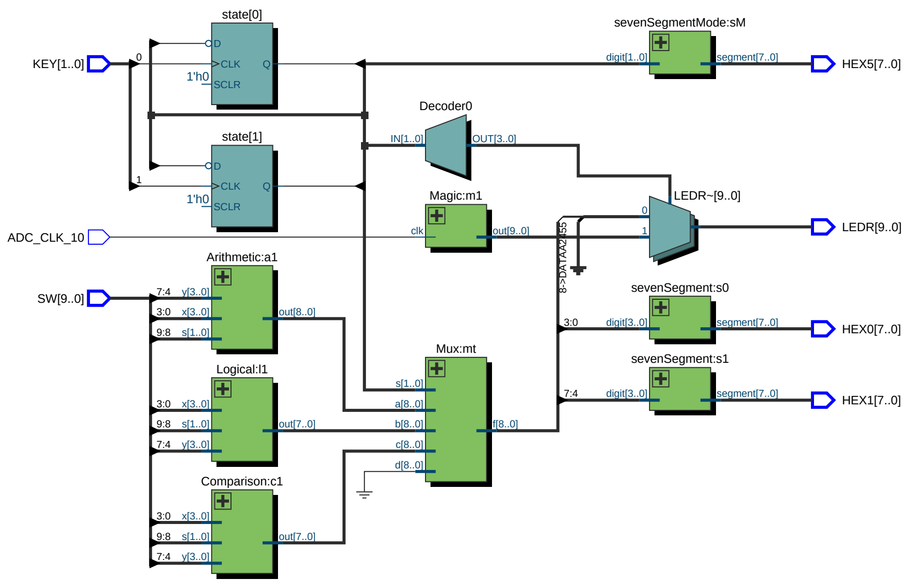

Figure 17: Generated Block Diagram of our Top.v

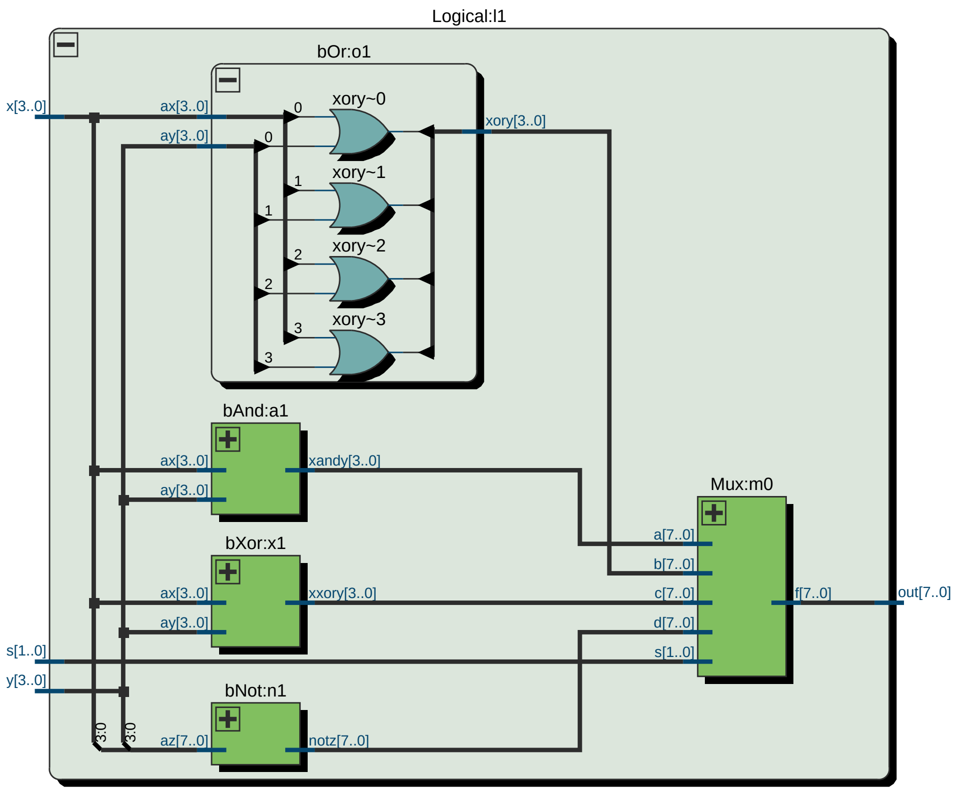

Figure 18: Generated Block Diagram of our Logical.v, with our 'or' module expanded as an example of how the other modules are laid out internally.

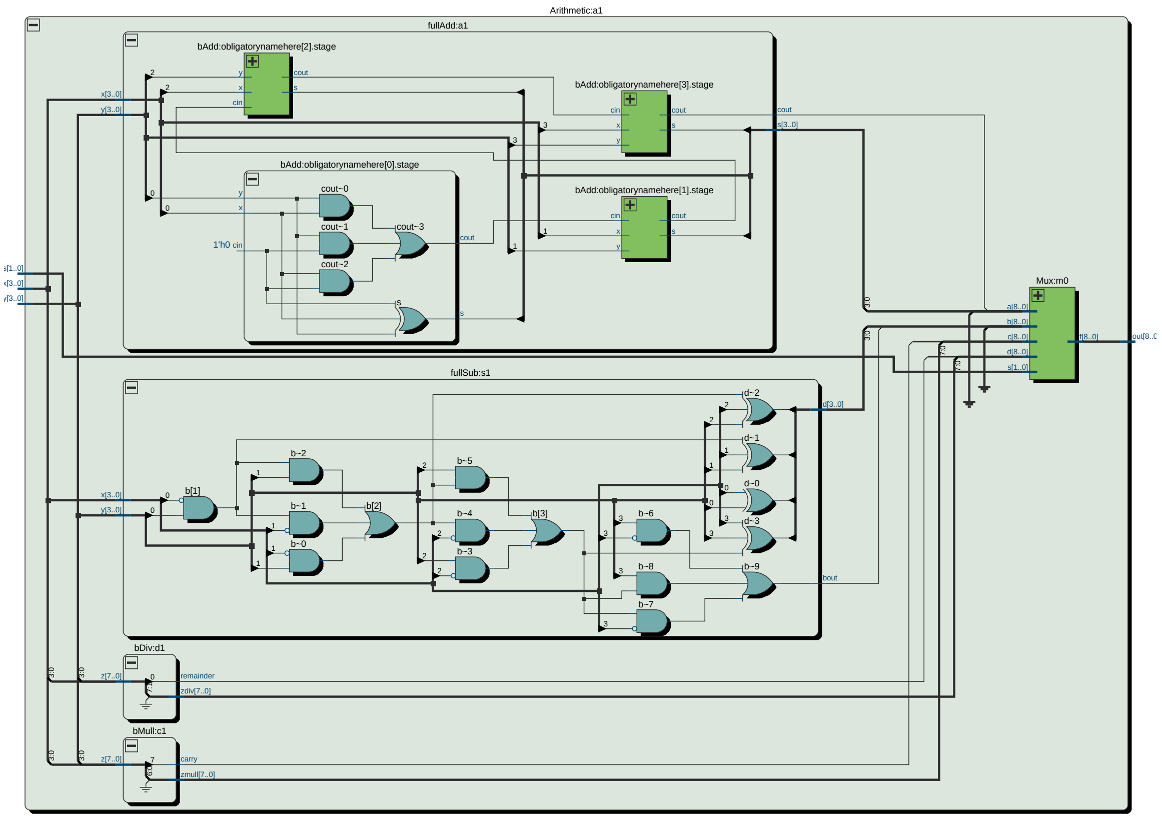

Figure 19: Generated Block Diagram of our Arithmetic.v, showing the 4 bit adder with one module expanded and the full 4 bit subtractor.

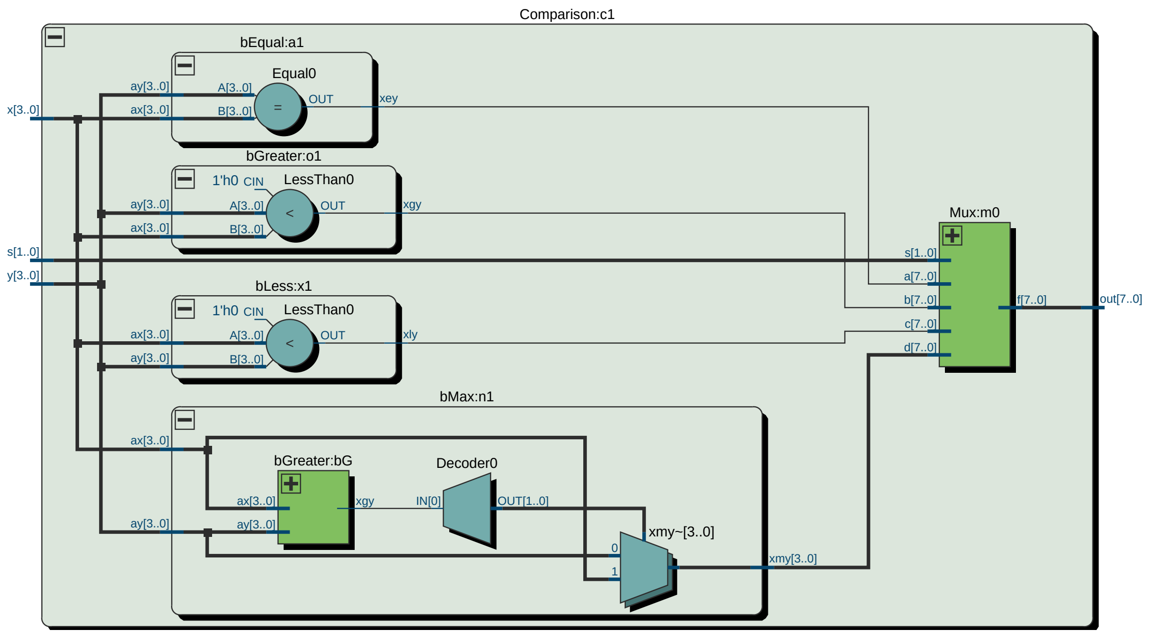

Figure 20: Generated Block Diagram of our Comparison.v, with Max expanded to show its use of bGreater.

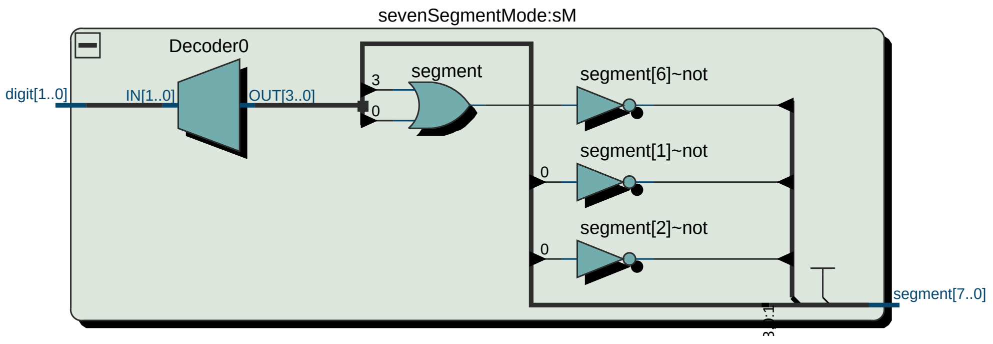

Figure 21: Generated Block Diagram of our SevenSegmentMode.v, showing how Quartus compiled our always/case statments for the mode indicator.

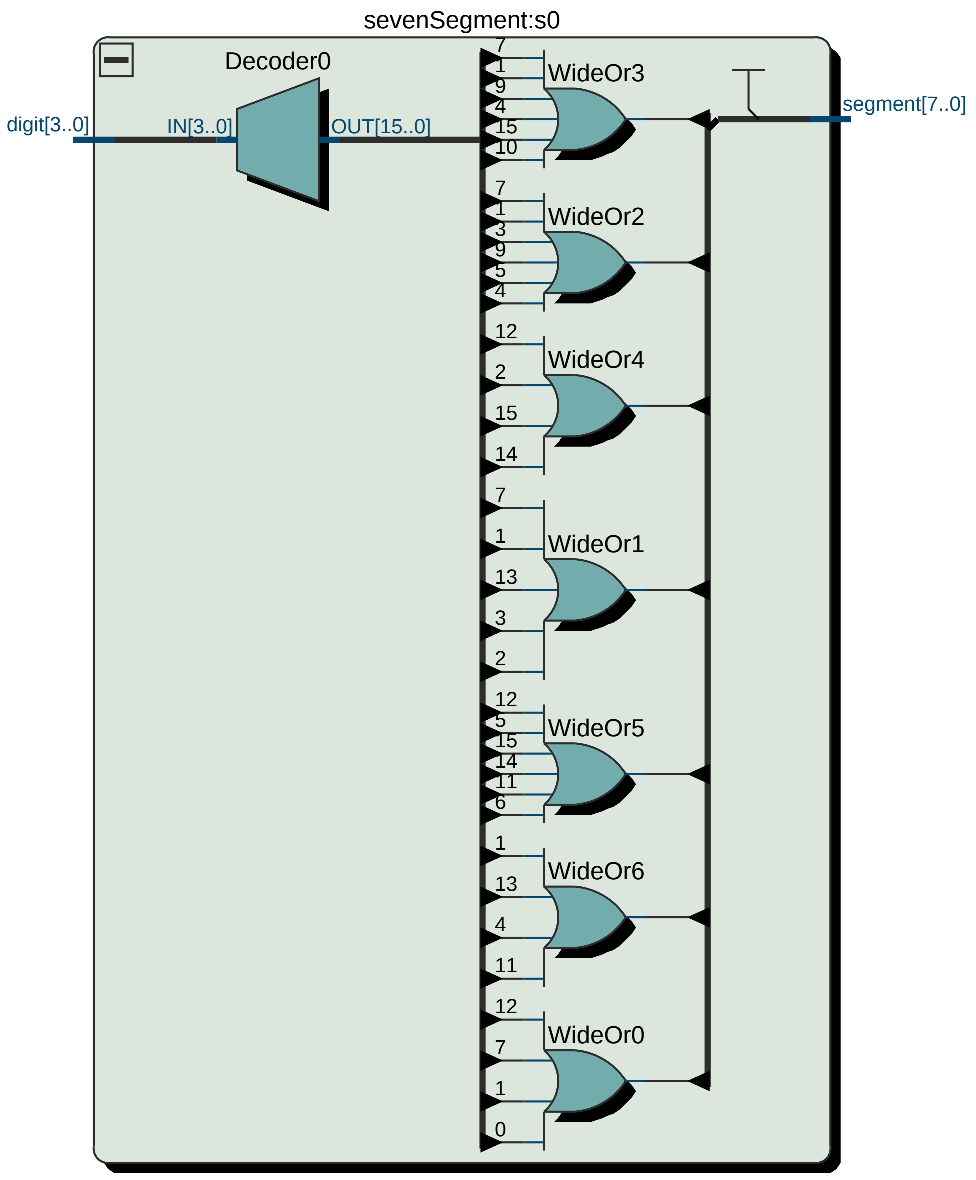

Figure 22: Generated Block Diagram of our SevenSegment.v, showing how Quartus compiled our always/case statments.

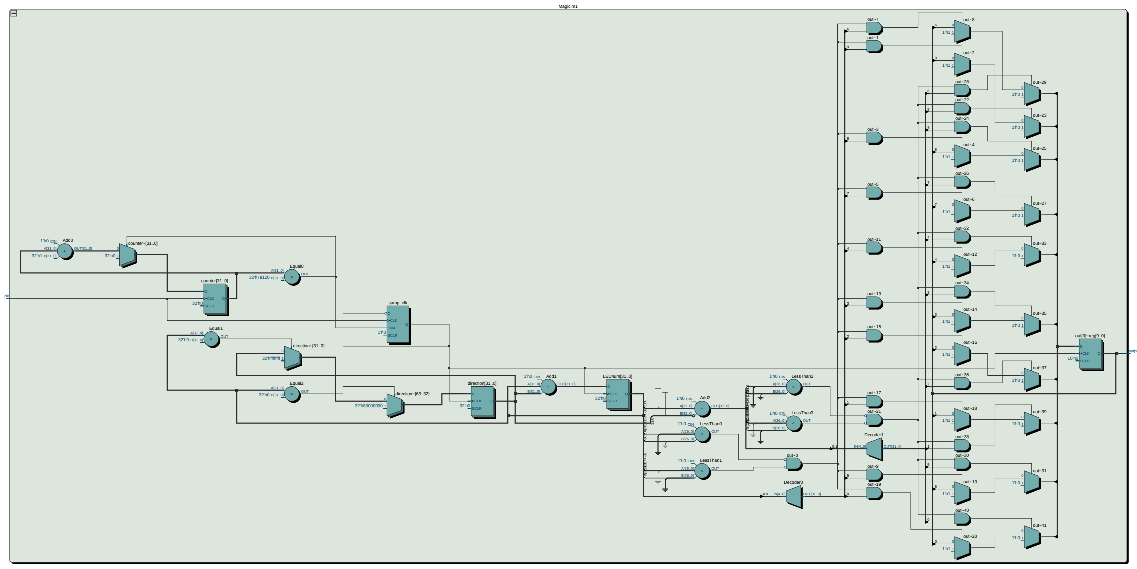

Figure 23: Generated Block Diagram of our Magic.v, showing how Quartus compiled our always/case statements and integer arithmetic from some very C-esqe code down into a very inefficient gate setup.