CS 4341 Digital Logic and Computer Design Semester Project
Assigned; March 22, 2021
Due: May 3rd, 2021

COVID-19

Normally, I would have the class break up into teams of four, and have class time for project meetings. This management plan does not work with social distancing. I have tried this style of project online, and I have found that fast digital communication allows teams to fall apart as fast as to put them together. Each student will be turning in their own copy of their project. Students may form a study group, and work together so long as they acknowledge they are working together. I refer to these study groups as cohorts. There will be a special 0-point quiz online for you to name your cohort. If you do not have a cohort, then you will use your netid.

Objective:

For this semester, you are being asked to create **behavioral-style data path circuit** using the Verilog hardware design language.

What is a data path circuit?

A data path circuit receives binary inputs, and a command. The data path circuit then processes the command to generate a binary output. In other words, when the system starts, data and commands are loaded into the data path. Time then passes, on a clock tick, and the results are then available as an output.

What is Behavioral-Style?

In Verilog, the program is made up of objects called modules. Every module can be either structural or behavioral. Structural coding means that all the wires and gates must be present in the code. In a behavioral style, which is what is required, you can use built-in operations instead of building the circuits down to the finest detail.

Topics

The primary topic for this project is the creation of an Arithmetic Logic Unit, which is the core calculation module in a computer processor. An ALU is a piece of ***sequential logic*** that uses ***combinational logic*** components to do work. The secondary topic for this project is to come up with a data path circuit of your own, that has the same level of complexity. Past examples have been robots and lightsabers.

For the ALU

An Arithmetic Logic Circuit contains a 32-bit Accumulator register, that holds the current value. The 32-bit accumulator registers is made up of 32 one-bit D Flip-flops working synchronously. On each instruction, the ALU will take a 16-bit integer input and the lower 16-bits of the accumulator register, and send them as inputs to 8 different modules. These modules will be a 16-bit adder, a 16-bit subtractor, a 16-bit multiplier, a 16-bit divider, a 16-bit AND, a 16-bit OR, a 16-bit NOT, and a 16-bit XOR. Each of these modules operate in **parallel**, and each of their outputs goes to a different input channel of a 16x32 bit multiplexer. (The output of a 16-bit multiplier can go up to 32 bits) The output of the ALU will be the current contents of the Accumulator register. The Subtractor will have an Overflow check, which will be an additional output. Also, the multiplexer has to handle the "No Operation" when no results of any module is used. And also, the D-Flip Flops must be able to be either preset to 1 or reset to 0. No-op, preset, and reset, will be handled by unused multiplexer channels.

Required Documentation

1. Description
2. Parts List
3. CIrcuit DIagram
4. OpCode Table
5. State Machine

The Description

Your project may be the one-input ALU described above, a multi-input ALU, or even a multi-input ALU with temporary registers for the input. All of these versions are fine, but you must write one up. Alternatively, you would have a description of an alternate project of equal complexity.

The Parts List

Your project should have a list of parts.

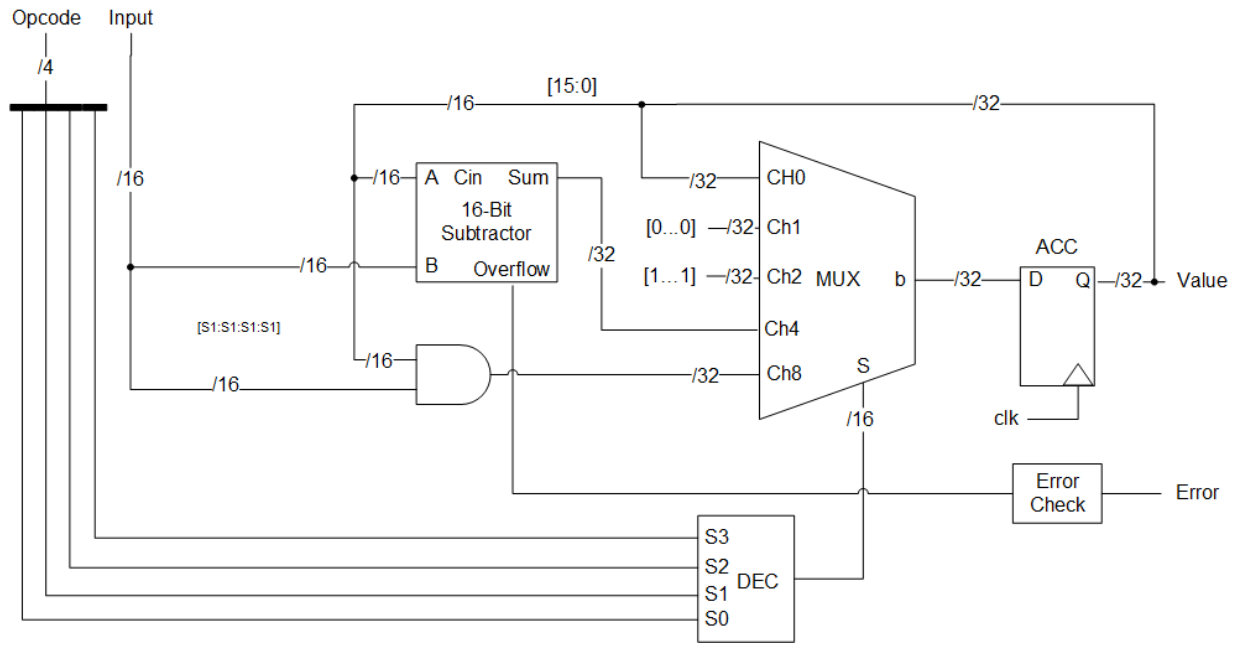
First the wires: All the inputs, and all the outputs, and all the interfaces should be listed. Inputs include both the data and the command. Outputs include the current results and any status messages. Interfaces include all the wires that connect between the components.

Second, gates and combinational logic components. All the gates and combinational logic components must be listed. Gates include all the AND, OR, NOT, XOR and their compliments. The combinational logic components include arbiters, comparators, multiplexors, decoders, encoders, adders, multipliers, subtractors, divisors, and any other combinational logic components.

Third, sequential logic components. Sequential logic components will include any flip-flops or registers made of flip-flops in your design

The Circuit Diagram

Your project should have a circuit diagram. Inputs should be on the left side of the diagram, and the outputs on the right. For spacing, I recommend putting sequential logic components in the middle, and filling the rest of the space with the combinational and gate logic components. Your circuit diagram should be complete, and its labels should match the descriptions in your parts list. If the grader cannot find your components, they cannot give you the points.



OpCode Table

The data path circuit you are creating will have a list of operations it will perform. They should have the name of the operation, a description of what the operation does, a op-code (digital abstraction and design) for the operation, and which channel will be handled by the binary code. The basic ALU in the description would have 4 math, 4 logic, and 3 support commands. So at least 11 operations.

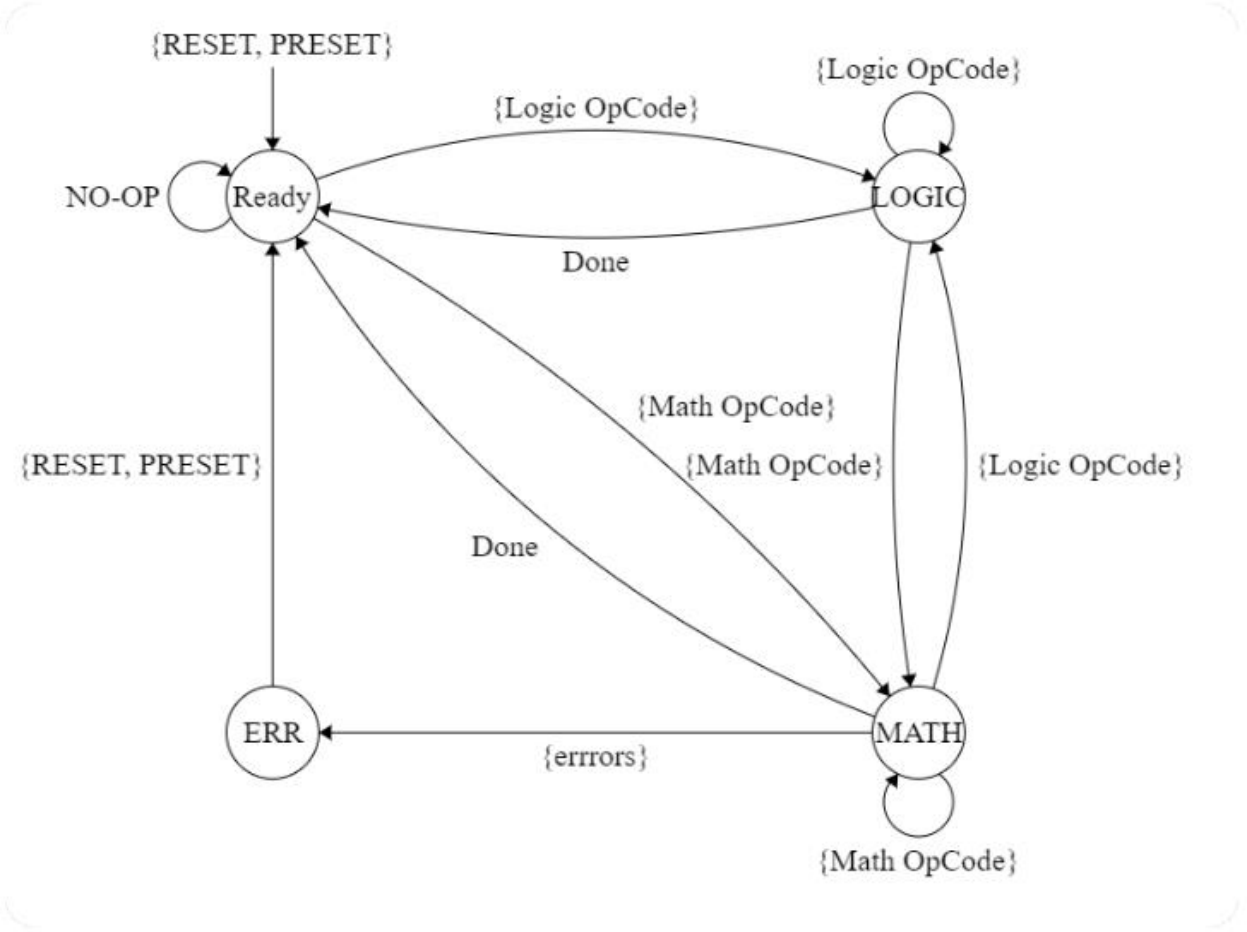
For example:

Command	Description	Op-Code	Multiplexer Channel
NO-OP	Refreshes Accumulator with feedback	0000	Channel 0
Done	When operation ends, return to a ready state	0000	Channel 0
Reset	Sets Accumulator to all 0's	0001	Channel 1
Preset	Sets Accumulator to all 1's	0010	Channel 2
Subtract	Subtract the input from the lower 16 bits of the accumulator	0100	Channel 4
AND	ANDs the input with the lower sixteen bits of the accumulator	1000	Channel 8

The State Machine Diagram

State Machines, Finite State Machines (FSM), Automata, Directed Graphs…. so many names. For this project, I will refer to it as the state machine. Your project does not require to have a programmed state machine, but the command structure will indicate a behavior your project should follow.

For Example, an ALU might have this state machine for its control.



Verilog Code

The next section is your Verilog code for your project. One thing you have to make sure is that your previous sections match your code: parts, operational codes, and modules actually match the code that you turn in. You will not be allowed to turn in a single module with everything written like one giant C or Java program. That will earn you a zero on the project.

On Verilog

The first reaction many new programmers have when they see Verilog is that the language is impossible. Actually, Verilog is not impossible. The language is just like the other programming languages you have seen, but with a time delay added to allow a parallel processing approach. Where can you find the information about the language? All three text books, including the MANO book, have chapters dedicated to Verilog.

Working with Dr. Becker's examples and using the textbook for reference will give you the material you need for the project. Students in the past have gone to the web to get an overcomplicated example, and then are unable to finish the project. Researching an example online is perfectly fine, but grabbing something huge and trying to make it work is a mistake. I strongly recommend starting small and building bigger as you make progress. ***Take an agile software engineering approach.***

Verilog uses modules, which behave like a class object of one function. Each main component in our circuit diagram (multiplexor, flip-flop, etc.) must be in their own module. Your Verilog code has a main module, called **testbench**, in which you set up the clock and inputs for the simulation. Between the two, I strongly recommend creating another module called Breadboard. Use the breadboard module to construct the circuit, and have the input go from the testbench to the bread board, and then the output from the breadboard back to the testbench.

On Output

Your output for your system has to be complete enough to show your system in operation. The testbench stimulus would start to run, and the output would show the behavior of the system in operation. The output should be readable. Various ASCII and formatting may be used to make an output that can be read. Your output should also use enough operations to show a complete run of the circuit. For example, a rover would have to start, stop, start, turn left, turn right, go back to stop.

Example:

In this example, the clock is shown to demonstrate the timing behavior. Input is the variable from the testbench, shown as an integer and as a binary. ACC is the current value of the register, shown as an integer and a brinaey. Instruction shows the command as a string label, and the opcode as a binary number. Next is the result, the next state of the system at the end of the clock tick. And the final column is a one-bit display to show if an error has occurred.

```
C|      |      |           |      |
L|Input |ACC   |Instruction|Next  |
K|#|BIN |#|BIN |CMD|OpCode |#|BIN |Error
-|-|----|-|----|------|----|-|----|-----
0|0|0000|X|XXXX|Reset | 100|X|XXXX| X
1|0|0000|X|XXXX|Reset | 100|0|0000| 0
0|0|0000|0|0000|No-Op | 000|0|0000| 0
1|0|0000|0|0000|No-Op | 000|0|0000| 0
0|5|0101|0|0000|ADD   | 001|0|0000| 0
1|0|0000|5|0101|ADD   | 001|5|0101| 0
0|0|0000|5|0101|No-Op | 000|5|0101| 0
1|F|1111|5|0101|AND   | 011|5|O101| 0
0|A|1010|5|0101|ADD   | 001|5|0101| 0
1|A|1010|5|0101|ADD   | 100|F|1111| 0
```

The Bonus

Verilog and building a circuit, and being able to generate an output that uses time constraints is all well and good. But… it is plain text on a screen. But I want to see something better. Take your project and turn it into something cool. Different library packaged exist to connect Verilog to other languages. Some versions of Verilog have libraries that connect to Javavscript. Other methods include file dumps and screen scrapes. Other bonuses have been to create an ALU out of Minecraft© Redstone.

For this semester, build something cool on top of Verilog, (Or at least using your design in a super-cool medium) and create a video that can be uploaded into Blackboard. It would be best if you turned in your submission before the video conferences on May 3rd.

What to turn in:

Each student will turn in the following. If you are doing this solo, replace <CohortName> with your netid.

1. A PDF named <COHORTNAME>.SystemDesign.pdf containing
    a. The description
    b. The parts list
    c. The circuit diagram
    d. The OpCode table
    e. The state machine
2. A text file named <COHORTNAME>.code.zip containing an archive of your Verilog code.
3. A text file named <COHORTNAME>.output.txt containing the output of a run of your program.

In the BONUS Turn-in location

1. A file named <COHORTNAME>.Bonus.<playable extension> that is a movie of your presentation about your bonus assignment.

Structural vs. Behavioral: An Example.

A structural module for a Full Adder would have the module header, which would include the inputs: a, b, and carry-in, and the outputs: sum and carry-out. Inside the module, the code would be written to have an XOR gate for the sum, and an AND gate for the carry-out. To combine the full adder into a 16-bit adder, sixteen full-adders would have to be instantiated and connected to produce the correct output. The sixteen sum outputs would be concatenated into a single output, and the final carry would become the carry out.

In a behavioral module for a full adder, the three inputs would be added together and stored in a 17-bit register. The lower 16 bits would become the sum, and the high bit would become the cout.

Both of these solutions are correct.

Example: Structural 16-bit adder

```
//================================
// Proper Half Adder
//================================
module HalfAdder1Bit(a,b,c,s) ;
  input a,b ;
  output c,s ;  // carry and sum
  assign s = a ^ b ;
  assign c = a & b ;
endmodule
```

```
//================================
// Proper Full Adder
//================================
module FullAdder1Bit(a,b,cin,cout,s) ;
  input a,b,cin ;
  output cout,s ;  // carry and sum
  wire g,p ;       // generate and propagate
  wire cp ;
  HalfAdder1Bit ha1(a,b,g,p) ;
  HalfAdder1Bit ha2(cin,p,cp,s) ;
  assign cout = g | cp;
endmodule
```

```
module SixteenBitAdder(a,b,cin,s,cout);
input [15:0] a;
input [15:0] b;
input cin;
output [15:0] s;
output [16:0] temp;
output cout;

wire  sum[15:0];
wire  car[15:0];


FullAdder1Bit fa0(a[ 0],b[ 0],cin    ,car[ 0],sum[ 0]);
FullAdder1Bit fa1(a[ 1],b[ 1],car[ 0],car[ 1],sum[ 1]);
FullAdder1Bit fa2(a[ 2],b[ 2],car[ 1],car[ 2],sum[ 2]);
FullAdder1Bit fa3(a[ 3],b[ 3],car[ 2],car[ 3],sum[ 3]);
FullAdder1Bit fa4(a[ 4],b[ 4],car[ 3],car[ 4],sum[ 4]);
FullAdder1Bit fa5(a[ 5],b[ 5],car[ 4],car[ 5],sum[ 5]);
FullAdder1Bit fa6(a[ 6],b[ 6],car[ 5],car[ 6],sum[ 6]);
FullAdder1Bit fa7(a[ 7],b[ 7],car[ 6],car[ 7],sum[ 7]);
FullAdder1Bit fa8(a[ 8],b[ 8],car[ 7],car[ 8],sum[ 8]);
FullAdder1Bit fa9(a[ 9],b[ 9],car[ 8],car[ 9],sum[ 9]);
FullAdder1Bit faA(a[10],b[10],car[ 9],car[10],sum[10]);
FullAdder1Bit faB(a[11],b[11],car[10],car[11],sum[11]);
FullAdder1Bit faC(a[12],b[12],car[11],car[12],sum[12]);
FullAdder1Bit faD(a[13],b[13],car[12],car[13],sum[13]);
FullAdder1Bit faE(a[14],b[14],car[13],car[14],sum[14]);
FullAdder1Bit faF(a[15],b[15],car[14],car[15],sum[15]);

assign cout=car[15];
assign s[ 0]=sum[ 0];
assign s[ 1]=sum[ 1];
assign s[ 2]=sum[ 2];
assign s[ 3]=sum[ 3];
assign s[ 4]=sum[ 4];
assign s[ 5]=sum[ 5];
assign s[ 6]=sum[ 6];
assign s[ 7]=sum[ 7];
assign s[ 8]=sum[ 8];
```

```
assign s[ 9]=sum[ 9];
assign s[10]=sum[10];
assign s[11]=sum[11];
assign s[12]=sum[12];
assign s[13]=sum[13];
assign s[14]=sum[14];
assign s[15]=sum[15];

endmodule
```

Example Behavioral 16-bit adder:

```
module SixteenBitAdder(a,b,cin,s,cout);
input [15:0] a;
input [15:0] b;
input cin;
output [15:0] s;
output cout;

//Here is the sneaky bit.
//You have to think by increasing the bus by 1 bit.
wire[16:0] temp;

assign temp=a+b; //Wow. This is short.
assign s[15:0]=temp[15:0];
assign cout=temp[16];
endmodule
```