Leila Scola & Story DeWeese
June 5, 2020
Final Project Report

**Abstract:**
Throughout this project, we created the data path as seen in **Figure 1: Data Path**. To do this, we first create each of the individual components. We were able to use and modify some of the work from previous labs as a starting point. Then, we altered that previous code to work for the specific data path for this project. This first step included finalizing the design for the *Mux*, *Program Counter, Instruction Memory, PC Buffer, IF/IX Buffer, Control, Register File, Sign Extend*, *ALU*, *ID/EX Buffer*, *Data Memory* and the *EX/WB Buffer*. Then, we created the *Testbench CPU* to utilize all of these components together. We used the output wires from one component as the input wires on the next according to **Figure 1: Data Path**. Next, from the given assembly instructions, we optimized and translated them to binary to test and debug our final data path.

**Detailed description of the CPU design including the datapath:**
If we begin at the *Program Counter* the *Mux* before it passes an address based on the previous instruction and this is passed as is. The *PC Buffer* stalls for one cycle before passing the address into the *IF/ID Buffer* to account for the time it takes for the *Adder* and the *Instruction Memory* to function. The *Adder* increases the address that was just passed by four, and offers it as a select to the *Mux* that feeds into the *Program Counter* to move the code sequentially if the instruction is not a branch or jump. The *Instruction Memory* will fetch the actual data stored at the address' point in memory and pass it to the *IF/ID Buffer*.

The *IF/ID Buffer* breaks up the passed memory and address and feeds it into many different modules. It feeds bits 0-21 of memory into the *Sign Extender*, bits 10-15 becomes the address for *Rt*, bits 16-21 becomes the address for *Rs*, bits 22-27 becomes the address for *Rd*, bits 28-31 of the memory into the *Control* module, and the address from the *PC Buffer* is stalled and output as is.

The *Control* unit takes its four-bit input and chooses its outputs bits (*RegWrt, MemToReg, PCtoReg, BranchNeg, BranchZero, Jump, JumpMem, MemRead, MemWrt,* and *ALUOP*) based on that. Based on the instruction, different bits are set to make sure the rest of the modules will work as expected to complete the instruction appropriately.

The *Register File* takes the address for *Rs* and *Rt* and reads them, outputting what is stored in memory at that point (the output will be 32 bits in this case). It may also receive *Rd* as a write address, decoding the address to write to that point in memory if specified by the *Write* bit being

high, which is an output from the *RegWrt* from the *Control* unit. The *Data* written comes from *Data Memory*.

While this is occurring, the *Sign Extend* module takes its 22-bit input and turns it into a 32-bit output by extending the MSB. This is passed to another *Adder* along with the *PC Buffer* address. The *Adder* combines these two inputs and passes it along to the *ID/EX Buffer*. This output is an input to the *Mux* seen at the right of the Data Path as a potential address to be passed back as the *Data* to be written to the Register File.

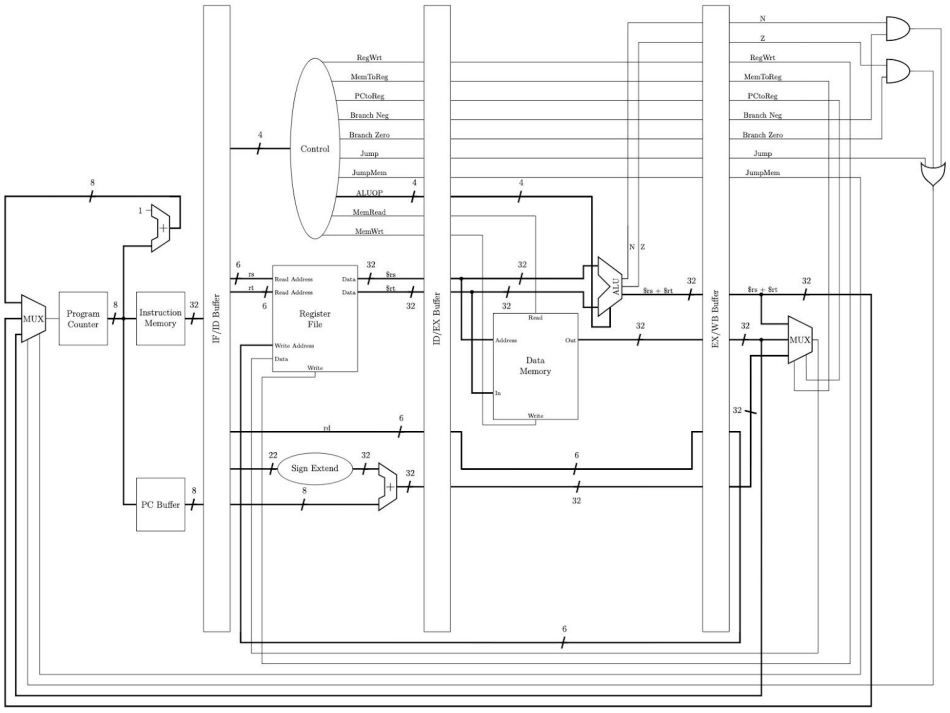
The *ID/EX Buffer* takes all the outputs from the *Control* unit, the *Register File*, the unused *Rd*, and the *Sign Extend Adder* and buffers them for one cycle. The *Data Memory* unit receives the *Rs* memory as an address, the *Rt* address as *Data* coming in, and *MemWrt* from the *Control* unit to choose whether to write to the data coming in to the address specified or not, applied on a LDUR or STUR instruction. It also receives a *MemRead* from the *Control* Unit, to read *Data Memory* from the given address. It outputs memory and passes it to the *EX/WB Buffer*. At this point, an *ALU* receives *Rs* and Rt as inputs and the *ALUOP* as a select to choose what arithmetic operation to perform on the two pieces of data. The result of the calculation is also passed to the *EX/WB Buffer*, as well as a *Negative* and a *Zero* indicator. The *EX/WB Buffer* stalls for one cycle, then passes its inputs.

There is an *AND* gate that takes the *Negative* and *Zero* as an input, and an *AND* gate that takes the *RegWrt* and *BranchZero Control* outputs as inputs. The result of the *AND* gates feeds into an *OR* gate that also takes the *Jump* bit from the *Control* unit. The result of the *OR* operation becomes a select for the *Mux* seen before the *Program Counter*. If asserted, it means that instructions will not be executed sequentially, but must *Branch* or *Jump* to a new location before returning to sequential execution.

There is a final *Mux* on the right side of the *Data Path* that receives the *ALU* output, *Data Memory* output, and the *Sign Extend Adder* outputs as inputs. It's selects come from *MemToReg* and *PCtoReg*. This *Mux* selects whether we are writing to memory or not and its output feeds into the *Register File* as "*Data*." The *Data* varies on the instruction given. If the instruction was an arithmetic operation, the *ALU* output is chosen, if the instruction was to access *Data Memory* (such as a LDUR/STUR) then the *Data Memory* output is chosen, and if there is an address passed that needs to be altered.

To wrap back to the beginning, that initial *Mux* on the left of the Data Path receives inputs of the *Program Counter Adder*, the *ALU* output, and the *Data Memory* output. Its selects are *JumpMem* from the *Control* unit and the *OR* gate output. This *Mux* outputs a value to be taken by the *Program Counter*, deciding whether we are branching or executing instructions sequentially.

**Figure 1: Data Path**

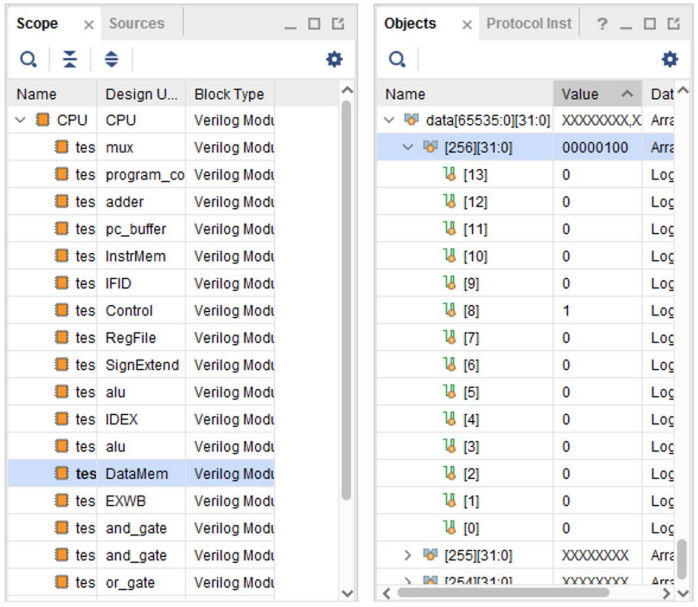


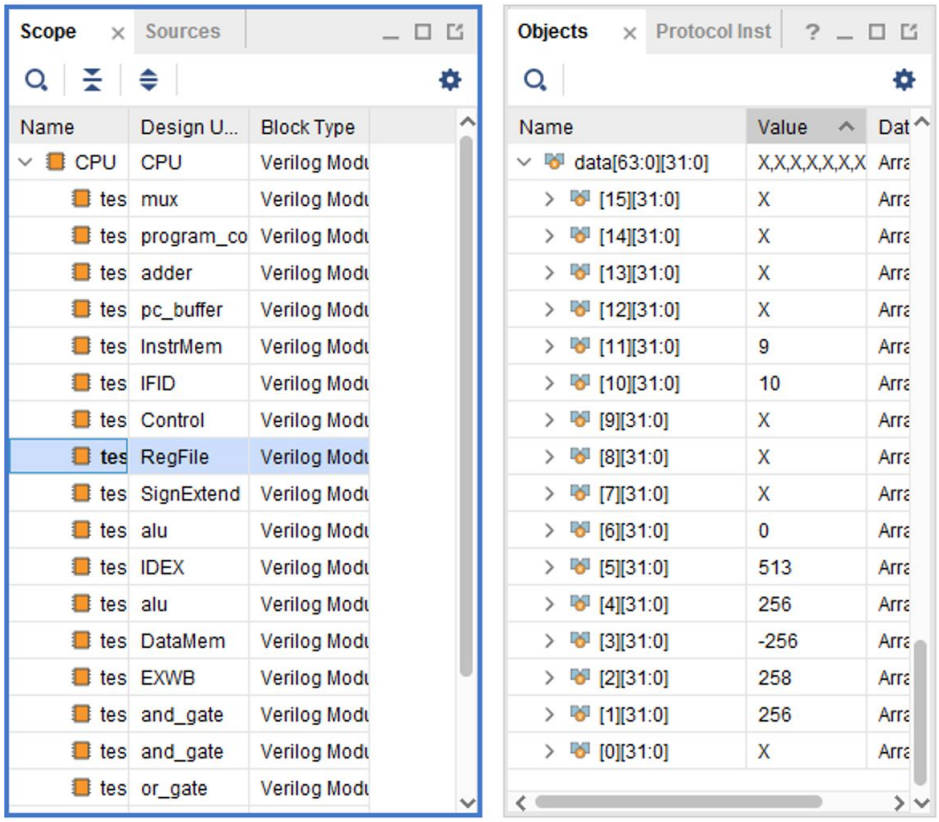

**Control Truth Table:**

| Opcode | RegWrt | MemToReg | PCtoReg | Branch Neg | Branch Zero | Jump | JumpMem | MemRead | MemWrt | ALUOP |
|---|---|---|---|---|---|---|---|---|---|---|
| 0000 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1111 |
| 1111 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0000 |
| 1110 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0100 |
| 0011 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0100 |
| 0100 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0000 |
| 0101 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0001 |
| 0110 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0010 |
| 0111 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0011 |
| 1000 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0100 |
| 1001 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0100 |
| 1010 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0100 |

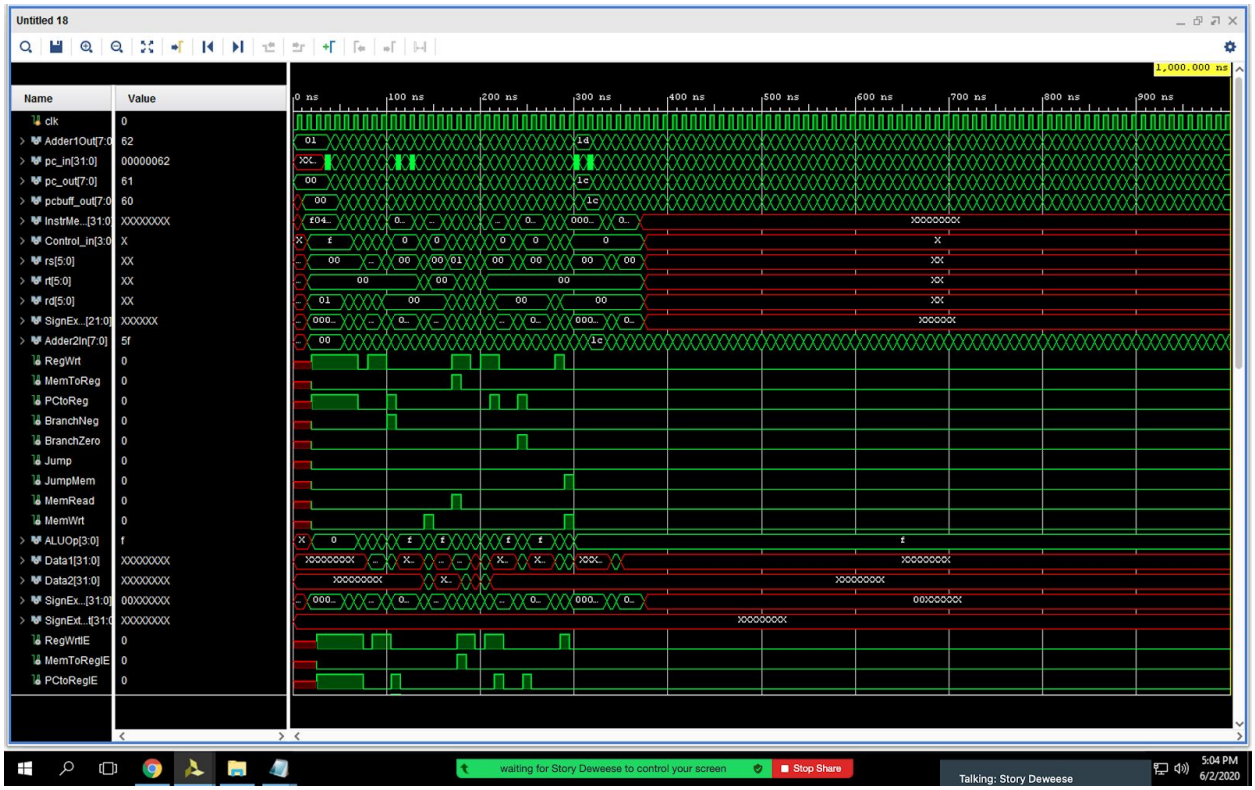
| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 1011 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0100 |
| 0001 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0000 |

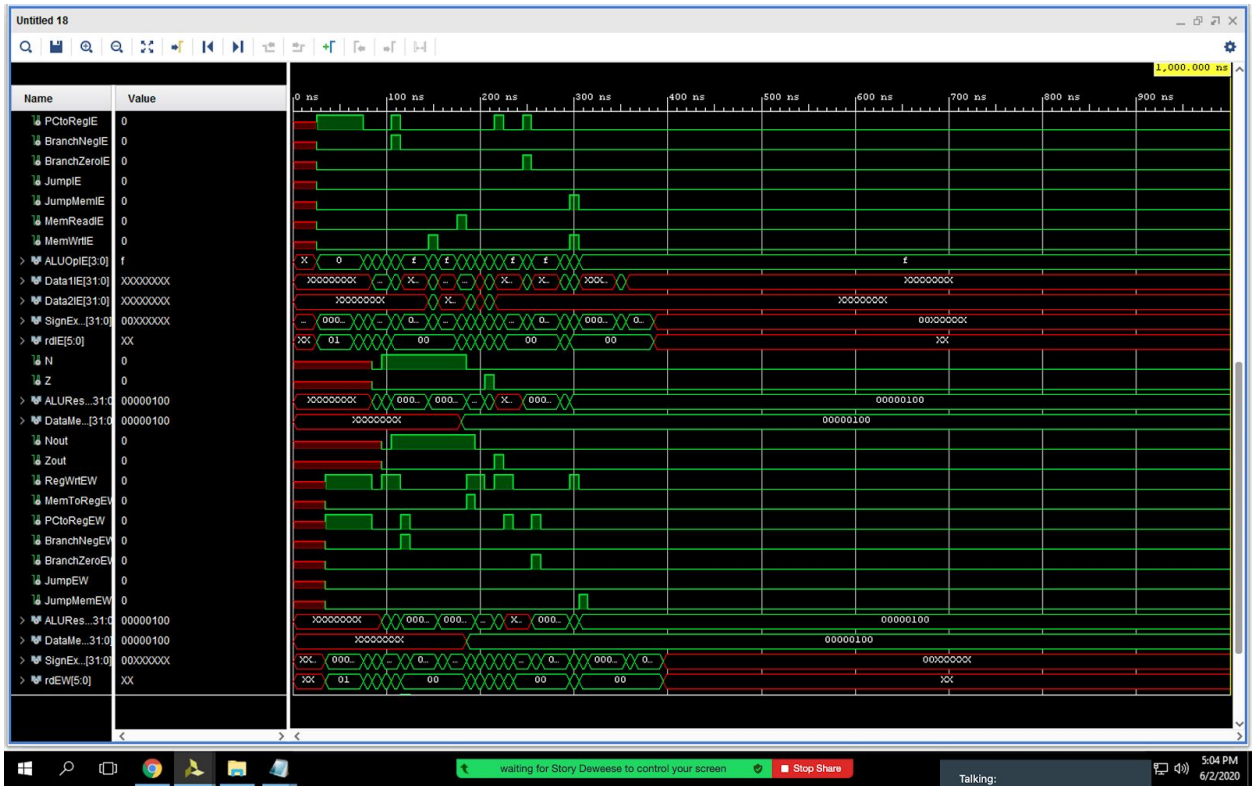
**Test benchmarks:**

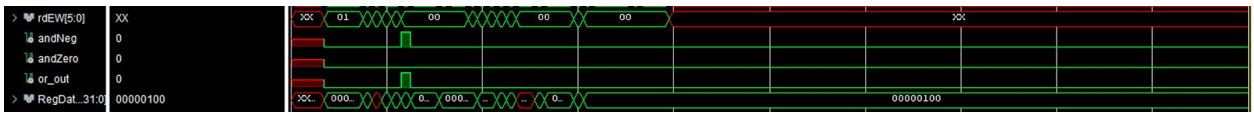


**Waveforms verifying the functions:**


Untitled 18
Name
Value
clk 0
Adder1Out[7:0] 62
pc_in[31:0] 00000062
pc_out[7:0] 61
pcbuff_out[7:0] 60
InstrMe...[31:0] XXXXXXXX
Control_in[3:0] X
rs[5:0] XX
rt[5:0] XX
rd[5:0] XX
SignEx...[21:0] XXXXXX
Adder2In[7:0] 5f
RegWrt 0
MemToReg 0
PCtoReg 0
BranchNeg 0
BranchZero 0
Jump 0
JumpMem 0
MemRead 0
MemWrt 0
ALUOp[3:0] f
Data1[31:0] XXXXXXXX
Data2[31:0] XXXXXXXX
SignEx...[31:0] 00XXXXXX
SignExt...t[31:0] XXXXXXXX
RegWrtIE 0
MemToRegIE 0
PCtoRegIE 0
1,000.000 ns
0 ns
100 ns
200 ns
300 ns
400 ns
500 ns
600 ns
700 ns
800 ns
900 ns
waiting for Story Deweese to control your screen
Stop Share
Talking: Story Deweese
5:04 PM
6/2/2020


Untitled 18
Name
Value
PCtoRegIE 0
BranchNegIE 0
BranchZeroIE 0
JumpIE 0
JumpMemIE 0
MemReadIE 0
MemWrtIE 0
ALUOpIE[3:0] f
Data1IE[31:0] XXXXXXXX
Data2IE[31:0] XXXXXXXX
SignEx...[31:0] 00XXXXXX
rdIE[5:0] XX
N 0
Z 0
ALURes...31:0 00000100
DataMe...[31:0] 00000100
Nout 0
Zout 0
RegWrtEW 0
MemToRegEW 0
PCtoRegEW 0
BranchNegEW 0
BranchZeroEW 0
JumpEW 0
JumpMemEW 0
ALURes...31:0 00000100
DataMe...31:0 00000100
SignEx...[31:0] 00XXXXXX
rdEW[5:0] XX
1,000.000 ns
0 ns
100 ns
200 ns
300 ns
400 ns
500 ns
600 ns
700 ns
800 ns
900 ns
waiting for Story Deweese to control your screen
Stop Share
Talking:
5:04 PM
6/2/2020


rdEW[5:0] XX
andNeg 0
andZero 0
or_out 0
RegDat...31:0 00000100

**Appendix:**

## Code:

Adder

```
module adder(A1, A2, Result);

// define inputs to adder, including 8-bit input A
input [7:0] A1;
input [7:0] A2;

// define out to be a reg meaning a value can be assigned to it
output reg [7:0] Result;

// the 'always@' means that every time an input changes, run this body of code
again
// (so our and gate reruns whenever we change the inputs!)
always@(A1, A2)
begin
      Result = A1 + 1;

end
endmodule
```

ALU

```
module alu(A, B, opcode, Result, neg, Z);

// define inputs to ALU, including 32-bit inputs A and B, and 4-bit opcode
input [31:0] A;
input [31:0] B;
input [3:0] opcode;

// define out to be a reg meaning a value can be assigned to it
output reg [31:0] Result;
//define negative and zero busts so value can be assigned to them
output reg neg;
output reg Z;

// the 'always@' means that every time an input changes, run this body of code
again
// (so our and gate reruns whenever we change the inputs!)
always@(A, B, opcode)
begin
//neg is always 1 if the MSB of Result is 1 (0 otherwise)
//Zero is always 1 if the result is all 0's

//if the opcode is 0000, it indicates ADD
```

```
      if(opcode == 4'b0000)
      begin
         //the result should be A+B
            Result = A+B;
      end
//if the opcode is 0001, it indicates INCREMENT (of A)
      if(opcode == 4'b0001)
      begin
      //The result is A + 1 in the LSB
            Result = A+1;
      end
//if the opcode is 0010, it indicates NEGATE A
      if(opcode == 4'b0010)
      begin
      //Result is equal to flipping all the bits of A and adding 1, which
creates a 2's complement
            Result = (~A)+32'b1;
      end
//if the opcode is 0011, it indicates SUBTRACT A and B
      if(opcode == 4'b0011)
      begin
      //to subtract A and B, add the 2's complement of B to A
            Result = A + (~B+32'b1);
      end
//if the opcode is 0100, it indicates PASS A
      if(opcode == 4'b0100)
      begin
      //Result = A
            Result = A;
      end
end
always@(Result)
begin
    if (opcode != 4'b0100)
    begin
        // if the result is equal to 0, set the zero flag to 1
        if (Result == 32'd0)
            Z= 1;
        else
            Z= 0;
        // if the most significant bit is 1, set the neg flag to 1
        if (Result[31] == 1)
            neg = 1;
        else
            neg = 0;
    end
end
endmodule
```

## And Gate

```
module and_gate(d1, d2, out);

// define d1, d2 as inputs
input d1 ,d2;

// define out to be a reg meaning a value can be assigned to it
output reg out;

// the 'always@' means that every time an input changes, run this body of code
again
// (so our and gate reruns whenever we change the inputs!)
always@(d1, d2)
begin
    out = d1 & d2;
end

endmodule
```

## Or Gate

```
module or_gate(d1, d2, d3, out);

    // define d1, d2 as inputs
    input d1, d2, d3;

    // define out to be a reg meaning a value can be assigned to it
    output reg out;

    // the 'always@' means that every time an input changes, run this body of
code again
    // (so our and gate reruns whenever we change the inputs!)
    always@(d1, d2, d3)
    begin
        out = d1 || d2 || d3;
    end

endmodule
```

## Control

```
module Control(CLK, Opcode, RegWrt, MemToReg, PCtoReg, BranchNeg, BranchZero,
Jump, JumpMem, ALUOP, MemRead, MemWrt);

// define input as the opcode
```

```
input [3:0] Opcode;
input CLK;

// define out to be a reg meaning a value can be assigned to it
//create output registers to be bits to control rest of logic operations based
on opcode
output reg RegWrt, MemToReg, PCtoReg, BranchNeg, BranchZero, Jump, JumpMem,
MemRead, MemWrt;
output reg [3:0] ALUOP;

// the 'always@' means that every time clock hits a positive edge, run this
body of code again
always@(negedge CLK)
begin
	//no operation
	if(Opcode == 4'b0000)
	begin
		RegWrt = 0;
		MemToReg = 0;
		PCtoReg = 0;
		BranchNeg = 0;
		BranchZero = 0;
		Jump = 0;
		JumpMem = 0;
		MemRead = 0;
		MemWrt = 0;
		ALUOP = 4'b1111;
	end
	//load PC ---> LDPC rd, X ---> $rd = PC + X
	else if(Opcode == 4'b1111)
	begin
		RegWrt = 1;
		MemToReg = 0;
		PCtoReg = 1;
		BranchNeg = 0;
		BranchZero = 0;
		Jump = 0;
		JumpMem = 0;
		MemRead = 0;
		MemWrt = 0;
		ALUOP = 4'b0000;
	end
	//load ---> LD rd, rs ---> $rd = M[$rs]
	else if(Opcode == 4'b1110)
	begin
		RegWrt = 1;
		MemToReg = 1;
		PCtoReg = 0;
```

```
        BranchNeg = 0;
        BranchZero = 0;
        Jump = 0;
        JumpMem = 0;
        MemRead = 1;
        MemWrt = 0;
        ALUOP = 4'b0100;
end
//store ---> ST rt, rs ---> M[$rs] <= $rt
else if(Opcode == 4'b0011)
begin
        RegWrt = 0;
        MemToReg = 0;
        PCtoReg = 0;
        BranchNeg = 0;
        BranchZero = 0;
        Jump = 0;
        JumpMem = 0;
        MemRead = 0;
        MemWrt = 1;
        ALUOP = 4'b0100;
end
//add ---> ADD rd, rs, rt ---> $rd <= $rs + $rt
else if(Opcode == 4'b0100)
begin
        RegWrt = 1;
        MemToReg = 0;
        PCtoReg = 0;
        BranchNeg = 0;
        BranchZero = 0;
        Jump = 0;
        JumpMem = 0;
        MemRead = 0;
        MemWrt = 0;
        ALUOP = 4'b0000;
end
//increment ---> INC rd, rs ---> $rd <= $rs + 1
else if(Opcode == 4'b0101)
begin
        RegWrt = 1;
        MemToReg = 0;
        PCtoReg = 0;
        BranchNeg = 0;
        BranchZero = 0;
        Jump = 0;
        JumpMem = 0;
        MemRead = 0;
        MemWrt = 0;
```

```
      ALUOP = 4'b0001;
end
//negate ---> NEG rd, rs ---> $rd <= -$rs
else if(Opcode == 4'b0110)
begin
      RegWrt = 1;
      MemToReg = 0;
      PCtoReg = 0;
      BranchNeg = 0;
      BranchZero = 0;
      Jump = 0;
      JumpMem = 0;
      MemRead = 0;
      MemWrt = 0;
      ALUOP = 4'b0010;
end
//subtract ---> SUB rd, rs, rt ---> $rd <= $rs - $rt
else if(Opcode == 4'b0111)
begin
      RegWrt = 1;
      MemToReg = 0;
      PCtoReg = 0;
      BranchNeg = 0;
      BranchZero = 0;
      Jump = 0;
      JumpMem = 0;
      MemRead = 0;
      MemWrt = 0;
      ALUOP = 4'b0011;
end
//jump ---> J rs ---> PC <= $rs
else if(Opcode == 4'b1000)
begin
      RegWrt = 0;
      MemToReg = 0;
      PCtoReg = 1;
      BranchNeg = 0;
      BranchZero = 0;
      Jump = 1;
      JumpMem = 0;
      MemRead = 0;
      MemWrt = 0;
      ALUOP = 4'b0100;
end
//Branch if Zero ---> BRZ rs ---> PC <= $rs
else if(Opcode == 4'b1001)
begin
      RegWrt = 0;
```

```
        MemToReg = 0;
        PCtoReg = 1;
        BranchNeg = 0;
        BranchZero = 1;
        Jump = 0;
        JumpMem = 0;
        MemRead = 0;
        MemWrt = 0;
        ALUOP = 4'b0100;
end
//jump memory ---> JM rs ---> PC <= M[$rs]
else if(Opcode == 4'b1010)
begin
        RegWrt = 0;
        MemToReg = 0;
        PCtoReg = 0;
        BranchNeg = 0;
        BranchZero = 0;
        Jump = 0;
        JumpMem = 1;
        MemRead = 0;
        MemWrt = 1;
        ALUOP = 4'b0100;
end
//branch if negative ---> BRN rs ---> PC <= $rs
else if(Opcode == 4'b1011)
begin
        RegWrt = 0;
        MemToReg = 0;
        PCtoReg = 1;
        BranchNeg = 1;
        BranchZero = 0;
        Jump = 0;
        JumpMem = 0;
        MemRead = 0;
        MemWrt = 0;
        ALUOP = 4'b0100;
end
//Sum ---> SUM rd, rs, rt ---> $rd = ($rs+$rt) memory [i]
else if(Opcode == 4'b0001)
begin
        RegWrt = 1;
        MemToReg = 1;
        PCtoReg = 0;
        BranchNeg = 0;
        BranchZero = 0;
        Jump = 0;
        JumpMem = 0;
```

```
            MemRead = 1;
            MemWrt = 0;
            ALUOP = 4'b0000;
      end
end

endmodule
```

## Data Memory

```
module DataMem(CLK, read, write, Addr, data_in, data_out);


// define inputs to Data Memory, including Clock, read and write signals, the
Address, and data-in
input CLK;
input read, write;
input [31:0] Addr, data_in;

// define out to be a reg meaning a value can be assigned to it
output reg [31:0] data_out;

//create array of 65536 elements with 32 bits each
reg [31:0] data [65535:0];

// the 'always@' means that every time clock hits a positive edge, run this
body of code again
always@(negedge CLK)
begin

      //if read bit is high, set the data out to data in the address location
from the test bench
      if (read == 1)
            data_out = data[Addr[15:0]];   //use first 16 bits so we don't
crash Vivado
      //if write bit is high, write to the data address from test bench the
data in given
      if (write == 1)
            data[Addr[15:0]] = data_in;
      //data[256] = 256;
end

endmodule
```

## IF/ID Buffer

```
module IFID(CLK, InstrMem, PCBuff, control_in, rs, rt, rd, SignExtendIn,
PCBuffOut); // control_in = instruction_out[38:24],

// define inputs to IF/ID Buffer, including Clock, Instruction Memory, and PC
Buffer
input [31:0] InstrMem;
input [7:0] PCBuff;
input CLK;

// define out to be a reg meaning a value can be assigned to it
//create output registers as reciprocals of inputs
output reg [3:0] control_in;
output reg [5:0] rs, rt, rd;
output reg [21:0] SignExtendIn;
output reg [7:0] PCBuffOut;

// the 'always@' means that every time clock hits a positive edge, run this
body of code again
always@(posedge CLK)
begin
//set outputs equal to inputs at next clock cycle
      control_in = InstrMem [31:28];
      rt = InstrMem[15:10];
      rs = InstrMem[21:16];
      rd = InstrMem[27:22];
      SignExtendIn = InstrMem[21:0];
      PCBuffOut = PCBuff;

end

endmodule
```

## ID/EX Buffer

```
module IDEX(CLK, RegWrtIn, MemtoRegIn, PCtoRegIn, BranchNeg, BranchZero, Jump,
JumpMem, ALUOpIn, MemRead, MemWrite, rs, rt, rd, addALUout, RegWrtOut,
MemtoRegOut, PCtoRegOut, BranchNegOut, BranchZeroOut, JumpOut, JumpMemOut,
Read, Write, ALUOpOut, rsout, rtout, rdout, addALUout2);


// define inputs to IDEX Buffer, including Clock, Writes, Reads, and Data
Inputs
input CLK, RegWrtIn, MemtoRegIn, PCtoRegIn, BranchNeg, BranchZero, Jump,
JumpMem, MemRead, MemWrite;
input [3:0] ALUOpIn;
input [31:0] rs, rt;
input [5:0] rd;
```

```
input [31:0] addALUout;

// define out to be a reg meaning a value can be assigned to it
//create output registers as reciprocals of inputs
output reg RegWrtOut, MemtoRegOut, PCtoRegOut, BranchNegOut, BranchZeroOut,
JumpOut, JumpMemOut, Read, Write;
output reg [3:0] ALUOpOut;
output reg [31:0] rsout, rtout;
output reg [5:0] rdout;
output reg [31:0] addALUout2;

// the 'always@' means that every time clock hits a positive edge, run this
body of code again
always@(posedge CLK)
begin
//set outputs equal to inputs at next clock cycle
      RegWrtOut = RegWrtIn;
      MemtoRegOut = MemtoRegIn;
      PCtoRegOut = PCtoRegIn;
      BranchNegOut = BranchNeg;
      BranchZeroOut = BranchZero;
      JumpOut = Jump;
      JumpMemOut = JumpMem;
      ALUOpOut = ALUOpIn;
      Read = MemRead;
      Write = MemWrite;
      rsout = rs;
      rtout = rt;
      rdout = rd;
      addALUout2 = addALUout;

end

endmodule
```

## EX/WB Buffer

```
module EXWB(CLK, Nin, Zin, RegWrtIn, MemtoRegIn, PCtoRegIn, BranchNeg,
BranchZero, Jump, JumpMem, REGin, DataMemIn, WrtAddrIn, ALUin, Nout, Zout,
RegWrtOut, MemtoRegOut, PCtoRegOut, BranchNegOut, BranchZeroOut, JumpOut,
JumpMemOut, REGout, DataMemOut, WrtAddrOut, ALUout);


// define inputs to EXWB, including Clock, Flags, Writes, Data coming in, etc.
input CLK, Nin, Zin, RegWrtIn, MemtoRegIn, PCtoRegIn, BranchNeg, BranchZero,
Jump, JumpMem;
input [5:0] WrtAddrIn;
```

```
input [31:0] DataMemIn, ALUin, REGin;


// define out to be a reg meaning a value can be assigned to it
//create output branches as reciprocals of inputs
output reg Nout, Zout, RegWrtOut, MemtoRegOut, PCtoRegOut, BranchNegOut,
BranchZeroOut, JumpOut, JumpMemOut;
output reg [5:0] WrtAddrOut;
output reg [31:0] DataMemOut, ALUout, REGout;

// the 'always@' means that every time clock hits a positive edge, run this
body of code again
always@(posedge CLK)
begin
//set outputs equal to the inputs at the next clock cycle
      Nout = Nin;
      Zout = Zin;
      RegWrtOut = RegWrtIn;
      MemtoRegOut = MemtoRegIn;
      PCtoRegOut = PCtoRegIn;
      BranchNegOut = BranchNeg;
      BranchZeroOut = BranchZero;
      JumpOut = Jump;
      JumpMemOut = JumpMem;
      REGout = REGin;
      DataMemOut = DataMemIn;
      WrtAddrOut = WrtAddrIn;
      ALUout = ALUin;

end
```

## Instruction Memory

```
module InstrMem(CLK, InstrAddr, Instruction);

// define inputs to Instruction Memory, including 8-bit input address and Clock
(CLK)
input [7:0] InstrAddr;
input CLK;

// define out to be a reg meaning a value can be assigned to it, it will be the
output instruction
output reg [31:0] Instruction;

//256x32 bits instruction memory array
reg [31:0] data [511:0]; // increased from 256 to 512
```

```
// hard-code in the machine code (binary) for the instructions that are given
in the demo
// each element represents a single instruction in this format: 4-bit opcode,
6-bit rd, 6-bit rs, 6-bit rt, 10-bit unused


//to assign bits to data to monitor functionality
initial begin
    // LDPC $r1, 0xFF (diff than lab handout, which will be updated shortly!)
note there are no rs & rd values needed for the Load PC instruction. The last
value is 255 in binary
    data [0] = {4'b1111, 6'b000001, 6'b000000, 6'b000000, 10'b0100000000}; //
start at 256!
     // LDPC $r10, 8 //rs & rd are don't cares. unused = 8
    data [1] = {4'b1111, 6'b001010, 6'b000000, 6'b000000, 10'b0000001010}; //
move up to second instruction to reduce the no-ops needed
    // 1 no-op should go here
    data [2] = {4'b0000, 6'b000000, 6'b000000, 6'b000000, 10'b0000000000};

    // INC $r2, $r1 // rd = 2, rs = 1, rt is a don't care
    data [3] = {4'b0101, 6'b000010, 6'b000001, 6'b000000, 10'b0000000000};
    // NEG $r3, $r1 // rd = 3, rs = 1, rt is a don't care
    data [4] = {4'b0110, 6'b000011, 6'b000001, 6'b000000, 10'b0000000000};

    // BRN $r10 // rs = 10, rd & rt are don't cares
    data [5] = {4'b1011, 6'b000000, 6'b001010, 6'b000000, 10'b0000000000};

    // 3 no-ops should go here
    data [6] = {4'b0000, 6'b000000, 6'b000000, 6'b000000, 10'b0000000000};
    data [7] = {4'b0000, 6'b000000, 6'b000000, 6'b000000, 10'b0000000000};
    data [8] = {4'b0000, 6'b000000, 6'b000000, 6'b000000, 10'b0000000000};

    // INC $r2, $r2 // rd = 2, rs = 2, rt is a don't care
    data [9] = {4'b0101, 6'b000010, 6'b000010, 6'b000000, 10'b0000000000};
    // ST $r1, $r1 // rt & rs = 1, rd is a don't care
    data [10] = {4'b0011, 6'b000000, 6'b000001, 6'b000001, 10'b0000000000};

    // 2 no-ops should go here
    data [11] = {4'b0000, 6'b000000, 6'b000000, 6'b000000, 10'b0000000000};
    data [12] = {4'b0000, 6'b000000, 6'b000000, 6'b000000, 10'b0000000000};

    // LD $r4, $r1 // rd = 4, rs = 1, rt is a don't care
    data [13] = {4'b1110, 6'b000100, 6'b000001, 6'b000000, 10'b0000000000};
    // ADD $r5, $r1, $r2 // rd = 5, rs = 1, rt = 2
    data [14] = {4'b0100, 6'b000101, 6'b000001, 6'b000010, 10'b0000000000};
    // 1 no-op should go here
    data [15] = {4'b0000, 6'b000000, 6'b000000, 6'b000000, 10'b0000000000};
```

```
    // SUB $r6, $r4, $r1 // rd = 6, rs = 4, rt = 1
    data [16] = {4'b0111, 6'b000110, 6'b000100, 6'b000001, 10'b0000000000};
    // LDPC $r11, 5 //rs & rd are don't cares. unused = 5
    data [17] = {4'b1111, 6'b001011, 6'b000000, 6'b000000, 10'b0000001001};
    // 2 no-ops should go here
    data [18] = {4'b0000, 6'b000000, 6'b000000, 6'b000000, 10'b0000000000};
    data [19] = {4'b0000, 6'b000000, 6'b000000, 6'b000000, 10'b0000000000};
    // BRZ $r11 // rs = 11, rd & rt are don't cares
    data [20] = {4'b1001, 6'b000000, 6'b001011, 6'b000000, 10'b0000000000};

    // 3 no-ops should go here
    data [21] = {4'b0000, 6'b000000, 6'b000000, 6'b000000, 10'b0000000000};
    data [22] = {4'b0000, 6'b000000, 6'b000000, 6'b000000, 10'b0000000000};
    data [23] = {4'b0000, 6'b000000, 6'b000000, 6'b000000, 10'b0000000000};

    // INC $r2, $r2 // rd = 2, rs = 2, rt is a don't care
    data [24] = {4'b0101, 6'b000010, 6'b000010, 6'b000000, 10'b0000000000};
    // JM $r1 // rs = 1, rd & rt are don't cares
    data [25] = {4'b1010, 6'b000000, 6'b000001, 6'b000000, 10'b0000000000};

    // 3 no-ops should go here
    data [26] = {4'b0000, 6'b000000, 6'b000000, 6'b000000, 10'b0000000000};
    data [27] = {4'b0000, 6'b000000, 6'b000000, 6'b000000, 10'b0000000000};
    data [28] = {4'b0000, 6'b000000, 6'b000000, 6'b000000, 10'b0000000000};

    // J $r1 // rs = 256, rd & rt are don't cares
    data [29] = {4'b0000, 6'b000000, 6'b000001, 6'b000000, 10'b0100000000};

    // 3 no-ops should go here
    data [30] = {4'b0000, 6'b000000, 6'b000000, 6'b000000, 10'b0000000000};
    data [31] = {4'b0000, 6'b000000, 6'b000000, 6'b000000, 10'b0000000000};
    data [32] = {4'b0000, 6'b000000, 6'b000000, 6'b000000, 10'b0000000000};
end


// the 'always@' means that every time input changes, run this body of code
again
//whenever the CLK hits a positive edge, this code repeats/begins
always@(negedge CLK)
begin
      //set the output instruction to the location in data memory of the
address given
      Instruction = data[InstrAddr];
end

endmodule
```

## Mux

```
module mux(d1, d2, d3, select1, out); // this select will take 2 1-bit inputs
(when testing, you can use this syntax: {sel1, sel2) to concatenate these into
1 input!
    // inputs declared
    input [31:0] d1, d2, d3; // different than the mux made in Lab 1, this mux
needs to be promoted to 32 bits
    input [1:0] select1;
    // output declared
    output reg [31:0] out;

    always@(d1, d2, d3, select1)
    begin
        // for select = 00, the output should be d1
        if(select1 == 2'b00)
            out = d1;
        // for select = 01, the output should be d2
        if(select1 == 2'b01)
            out = d2;
        // for select = 10, the output should be d3
        if(select1 == 2'b10)
            out = d3;
     end

endmodule
```

## Program Counter

```
module program_counter(clk, pc_in, pc_out); // you only need 8 bits for this!
   // inputs declared
   input [7:0] pc_in;
   input clk;
   // outputs declared
   output reg [7:0] pc_out;

   initial begin
      pc_out = 0;
   end

  always@(negedge clk)
   begin
   if(pc_in) // ensure that the input is a valid value before setting the
output equal to it
   begin
        pc_out = pc_in;
```

```
   end
   end

endmodule
```

## PC Buffer

```
module pc_buffer(CLK, pc_in, pc_out);
   // inputs declared
   input CLK;
   input [7:0] pc_in;
   // outputs declared
   output reg [7:0] pc_out;

   always@(negedge CLK)
   begin
        pc_out = pc_in;
   end

endmodule
```

## Register File

```
module RegFile(CLK, write, Rd, Rs, Rt, data_in, rsout, rtout);

// define inputs to Register File, including Clock, write bit, Input Registers
-- Rs, Rd, Rt --, and Data-In
input CLK;
input write;
input [5:0] Rd, Rs, Rt;
input [31:0] data_in;

// define outputs to be a reg meaning a value can be assigned to it
output reg [31:0] rsout;
output reg [31:0] rtout;

//64 x 32 array (64 elements, 32 bits each)
reg [31:0] data [63:0];


// the 'always@' means that every time clock changes, run this body of code
again
//at the clock's positive edge, this code will begin
always@(negedge CLK)
begin
      //if write is high, set data at the output destination to data in
```

```
	if(write == 1)
		data[Rd] = data_in;

	//set the Rs register to the location in data memory of Rs
	rsout = data[Rs];
	//set the Rt register to the location in data memory of Rt
	rtout = data[Rt];
end

endmodule
```

## Sign Extend

```
module SignExtend(SignIn, SignOut);

// define input to be incoming data.
input [21:0] SignIn;

// define out to be a reg meaning a value can be assigned to it
//create output as sign extension of input
output reg [31:0] SignOut;

// the 'always@' means that every time clock hits a positive edge, run this
body of code again
always@(SignIn)
begin
//set outputs equal to the inputs at the next clock cycle
	if(SignIn[21] == 1)
		SignOut = {10'b1111111111, SignIn};
	else
		SignOut = {10'b0000000000, SignIn};

end
endmodule
```

## CPU Testbench

```
module CPU();

// regs can store data
//create registers for clock input
reg clk;

//create wires for MUX1
wire [7:0] Adder1Out;
//create wire to output for MUX1/input PC
```

```
wire [31:0] pc_in;


//create wires for PC
wire [7:0] pc_out; //adder1in, pcbuff_in, instrmem_in


//create wires for PC Buffer
wire [7:0] pcbuff_out;


//create wires for Instruction Memory
wire [31:0] InstrMemVal;

//create wire to output for IFID
//wire [31:0] InstrMemValOut;
wire [3:0] Control_in;
wire [5:0] rs, rt, rd;
wire [21:0] SignExtendIn;
wire [7:0] Adder2In;

//create wire to output for Control
wire RegWrt, MemToReg, PCtoReg, BranchNeg, BranchZero, Jump, JumpMem, MemRead,
MemWrt;
wire [3:0] ALUOp;

//create wire to output for Register File
wire [31:0] Data1, Data2;

//create wire to output for Sign Exend
wire [31:0] SignExtended;

//create wire for SignExtendAdder
wire [31:0] SignExtendAdderOut;

//create wire for IDEX
wire RegWrtIE, MemToRegIE, PCtoRegIE, BranchNegIE, BranchZeroIE, JumpIE,
JumpMemIE, MemReadIE, MemWrtIE;
wire [3:0] ALUOpIE;
wire [31:0] Data1IE, Data2IE, SignExtendedIE;
wire [5:0] rdIE;

//create wire for ALU
wire N, Z;
wire [31:0] ALUResult;

//wire for Data Memory
wire [31:0] DataMemOut;
```

```
//wire for EXWB
wire Nout, Zout, RegWrtEW, MemToRegEW, PCtoRegEW, BranchNegEW, BranchZeroEW,
JumpEW, JumpMemEW;
wire [31:0] ALUResultEW, DataMemEW, SignExtendedEW;
wire [5:0] rdEW;

//wire for BranchNeg and Neg AND gate
wire andNeg;

//wire for BranchZero and Zero AND gate
wire andZero;

//wire for AND gates and JUMP OR gate
wire or_out;

//wire for final mux
wire [31:0] RegDataIn;


mux test0({24'b0, Adder1Out}, ALUResultEW, DataMemEW, {JumpMemEW, or_out},
pc_in); // first mux

program_counter test(clk, pc_in[7:0], pc_out);

adder test1(pc_out, 1, Adder1Out);

pc_buffer test2(clk, pc_out, pcbuff_out);

InstrMem test3(clk, pc_out, InstrMemVal);

IFID test4(clk, InstrMemVal, pcbuff_out, Control_in, rs, rt, rd, SignExtendIn,
Adder2In);

Control test5(clk, Control_in, RegWrt, MemToReg, PCtoReg, BranchNeg,
BranchZero, Jump, JumpMem, ALUOp, MemRead, MemWrt);

RegFile test6(clk, RegWrtEW, rdEW, rs, rt, RegDataIn, Data1, Data2);

SignExtend test7(SignExtendIn, SignExtended);

alu test8(SignExtended, Adder2In, 4'b00, SignExtendAdderOut, , );

IDEX test9 (clk, RegWrt, MemToReg, PCtoReg, BranchNeg, BranchZero, Jump,
JumpMem, ALUOp, MemRead, MemWrt, Data1, Data2, rd, SignExtended, RegWrtIE,
MemToRegIE, PCtoRegIE, BranchNegIE, BranchZeroIE, JumpIE, JumpMemIE, MemReadIE,
MemWrtIE, ALUOpIE, Data1IE, Data2IE, rdIE, SignExtendedIE);
```

```
alu test10(Data1IE, Data2IE, ALUOpIE, ALUResult, N, Z);

DataMem test11(clk, MemReadIE, MemWrtIE, Data1IE, Data2IE, DataMemOut);

EXWB test12 (clk, N, Z, RegWrtIE, MemToRegIE, PCtoRegIE, BranchNegIE,
BranchZeroIE, JumpIE, JumpMemIE, SignExtendedIE, DataMemOut, rdIE, ALUResult,
Nout, Zout, RegWrtEW, MemToRegEW, PCtoRegEW, BranchNegEW, BranchZeroEW, JumpEW,
JumpMemEW, SignExtendedEW, DataMemEW, rdEW, ALUResultEW);

and_gate test13(Nout, BranchNegEW, andNeg);

and_gate test14(Zout, BranchZeroEW, andZero);

or_gate test15(andNeg, andZero, JumpEW, or_out);

mux test16(ALUResultEW, DataMemEW, SignExtendedEW, {PCtoRegEW, MemToRegEW},
RegDataIn);

//set the clock to create a square wave
initial begin
      clk = 0;
      //forever, every 5 ns, clock switches from high to low
      forever #5 clk = ~clk;
end

// 'initial' means just to do it once (unlike 'always')
initial
begin
      #2000;//wait 2000 ns to execute
    $finish;
end
endmodule
```

The assembly code was provided by the TA. It was then optimized and translated into binary for the purpose of inputs for this lab:

```
LDPC $r1,0x100 # r1 = 0x100
LDPC $r10, 10 #please replace 'label1' with a number # moved up to act as NO OP
NO OP

INC $r2,$r1 # r2 = 0x101
NEG $r3,$r1# r3 = 0xFFFF_FF00
BRN $r10 # jump to 'label1'
NO OP # 3 needed after Branch
NO OP
NO OP

INC $r2,$r2 # r2 = r2+1 should NOT be executed.
```

```
10:
        ST $r1,$r1 # mem[0x100] = 0x100
        NO OP
        NO OP
        LD $r4, $r1 # r4 = 0x100
        ADD $r5, $r1, $r2 # r5 = 0x201 # act as NO OP for r4
        NO OP
        SUB $r6, $r4, $r1 # r6 = 0x0
        LDPC $r11, 9 #please replace 'label2' with a number
        NO OP
        NO OP
        BRZ $r11 # jump to 'label2'
        NO OP # 3 needed after Branch
        NO OP
        NO OP
        INC $r2, $r2 # r2 = r2+1 should NOT be executed.
9:
        JM $r1 # jump to 0x100
        NO OP # 3 needed after Jump
        NO OP
        NO OP
        0x100:
        J $r1 #Please put this instruction at IM address 0x100; endless loop to
        0x100
        NO OP # 3 needed after Jump
        NO OP
        NO OP
```