# Lab 6 - Datapath for I,J,R type

CECS 341 – Computer Architecture Organization

Student Antonio Ojeda SID 026076048
Student Bridget Naylor SID 025531413
Professor:Jose Aceves


California State University, Long Beach

College of Engineering
1250 Bellflower Blvd, Long Beach, CA 90840
Lab 2/12/2021

Lab 6-I-Type CECS 341 Spring 2021

**Goal/Objective:**

Our goal is to modify our existing lab to make it able to execute jump instructions.

**Technical Description/Steps: (include screenshots and description here)**

To make the existing program execute jump instructions, we needed another mux and a shift left. We also needed to add a jump variable to our ALU to switch depending on the instruction.

```
wire Branchsrc;
wire jump;
wire [31:0] SignExtend;

//Output wires for mux
wire [4:0] RegDstMux;
wire [31:0] MemtoRegMux;
wire [31:0] ALUsrcMux;
wire [31:0] BranchSrcMux;
wire [31:0] JumpMux;


//Create logic for muxes
assign RegDstMux = (RegDst == 1) ? instructionMemOutputWire[15:11] : instructionMemOutputWire[20:16];
assign MemtoRegMux = (MemtoReg == 1) ? DataMemOut : ALUOutputWire;
assign ALUsrcMux = (ALUsrc == 1) ? SignExtend : rtRTOutputWire;
assign BranchSrcMux = (Branchsrc == 1) ? pcaOutputWire1 : pcaOutputWire;
assign JumpMux = (jump == 1) ? {pcaOutputWire[31:28], instructionMemOutputWire[25:0], 2'b00} : BranchSrcMux;
```

This image includes the addition of the jump wire, as well as the JumpMux wire and the assignment of JumpMux between the new jump address and the output of the branch mux. The new jump address is a concatenation of the bits [31:28] of the first PC adder output, bits [25:0] of the instruction, and 2 additional bits as a result of shifting left 2 times.

```
control ctrl(.Op(instructionMemOutputWire[31:26]),
             .Func(instructionMemOutputWire[5:0]),
             .RegWrite(controlRegRWriteOutputWire),
             .ALUCtrl(controlALUCntlOutputWire),
             .RegDst(RegDst),
             .Branch(Branch),
             .MemRead(MemRead),
             .MemWrite(MemWrite),
             .MemtoReg(MemtoReg),
             .ALUsrc(ALUsrc),
             .jump(jump));
```

The last connection port of ctrl is connected to the jump wire.

```
PC pcr(.clock(clk),
       .Reset(reset),
       .Din(JumpMux),
       .PC_out(pcrOutputWire));
```

This PC register has an additional port where instead of the output of the branch mux going into it, the output of the jump mux contains the new instruction address.

```
6'h02: begin    //j
       ALUCtrl = 4'b0;
       RegWrite = 1'b0;
       RegDst = 1'b0;
       Branch = 2'b0;
       MemRead = 1'b0;
       MemWrite = 1'b0;
       MemtoReg = 1'b0;
       ALUsrc = 1'b0;
       jump = 1'b1;
       end
```

This new jump control case outputs signals to no other output signals except for jump.

```
00 00 08 20
00 00 09 20
05 00 0A 34
00 00 0B 20
2A 60 0A 01
05 00 80 11
00 00 6C 8D
20 48 2C 01
01 00 08 21
04 00 6B 21
04 00 00 08
00 00 68 AD
04 00 69 AD
08 00 6A AD
0C 00 6B AD
```

This is the instruction memory file we translated from the MIPS instructions. It is set in the Little Endian format.

The following images are unchanged except for the relating to the jump modification.

Datapath:

```
module Datapath(
    input clk,
    input reset,
    output [31:0] Dout
    );

    wire [31:0] pcaOutputWire;
    wire [31:0] pcaOutputWire1;
    wire [31:0] pcrOutputWire;
    wire [31:0] rfRSOutputWire;
    wire [31:0] rtRTOutputWire;
    wire [31:0] instructionMemOutputWire;
    wire controlRegRWriteOutputWire;
    wire [3:0] controlALUCntlOutputWire;
    wire [31:0] ALUOutputWire;
    wire [31:0] DataMemOut;
    wire N;
    wire Z;
    wire C;
    wire V;
    wire RegDst;
    wire MemWrite;
    wire MemRead;
    wire [1:0] Branch;
    wire MemtoReg;
    wire ALUsrc;
    wire Branchsrc;
    wire jump;
    wire [31:0] SignExtend;

    //Output wires for mux
    wire [4:0] RegDstMux;
    wire [31:0] MemtoRegMux;
    wire [31:0] ALUsrcMux;
    wire [31:0] BranchSrcMux;
    wire [31:0] JumpMux;


    //Create logic for muxes
    assign RegDstMux = (RegDst == 1) ? instructionMemOutputWire[15:11] : instructionMemOutputWire[20:16];
    assign MemtoRegMux = (MemtoReg == 1) ? DataMemOut : ALUOutputWire;
    assign ALUsrcMux = (ALUsrc == 1) ? SignExtend : rtRTOutputWire;
    assign BranchSrcMux = (Branchsrc == 1) ? pcaOutputWire1 : pcaOutputWire;
    assign JumpMux = (jump == 1) ? {pcaOutputWire[31:28], instructionMemOutputWire[25:0], 2'b00} : BranchSrcMux;
```

```
control ctrl(.Op(instructionMemOutputWire[31:26]),
             .Func(instructionMemOutputWire[5:0]),
             .RegWrite(controlRegRWriteOutputWire),
             .ALUCtrl(controlALUCntlOutputWire),
             .RegDst(RegDst),
             .Branch(Branch),
             .MemRead(MemRead),
             .MemWrite(MemWrite),
             .MemtoReg(MemtoReg),
             .ALUsrc(ALUsrc),
             .jump(jump));
Instruction_Memory im(.Addr(pcrOutputWire),
                      .Inst_out(instructionMemOutputWire));
regfile32 rf(.clk(clk),
             .reset(reset),
             .D_En(controlRegRWriteOutputWire),
             .D_Addr(RegDstMux),
             .S_Addr(instructionMemOutputWire[25:21]),
             .T_Addr(instructionMemOutputWire[20:16]),
             .D(MemtoRegMux),
             .S(rfRSOutputWire),
             .T(rtRTOutputWire));
PCADD pca(.Din(pcrOutputWire),
          .PCADD_out(pcaOutputWire));
PCADD1 pca1(.PCADDIn(pcaOutputWire),
            .SignEx(SignExtend),
            .NextPC(pcaOutputWire1));
BranchSrc bsrc(.BranchIn(Branch),
               .Zero(Z),
               .BranchOut(Branchsrc));
PC pcr(.clock(clk),
       .Reset(reset),
       .Din(JumpMux),
       .PC_out(pcrOutputWire));
ALU alu(.A(rfRSOutputWire),
        .B(ALUsrcMux),
        .ALUctrl(controlALUCntlOutputWire),
        .ALUout(ALUOutputWire),
        .N(N), .C(C), .Z(Z), .V(V));

   DataMem dm(.clk(clk),
              .mem_wr(MemWrite),
              .mem_rd(MemRead),
              .addr(ALUOutputWire),
              .wr_data(rtRTOutputWire),
              .rd_data(DataMemOut));
   SignExtension se(.SignExtIn(instructionMemOutputWire[15:0]),
                    .SignExtOut(SignExtend));
   assign Dout = MemtoRegMux;
endmodule
```

Control:

```
module control(
    input [5:0] Op,
    input [5:0] Func,
    output reg RegWrite,
    output reg [3:0] ALUCtrl,
    output reg RegDst,
    output reg [1:0] Branch,
    output reg MemRead,
    output reg MemWrite,
    output reg MemtoReg,
    output reg ALUsrc,
    output reg jump
    );

    always@(*) begin
       if (Op == 6'b0) begin    //Detects R-Type Instruction
            RegWrite = 1'b1;    //Wrutes back to register file
            RegDst = 1'b1;      //Inst[15:11] as write back address
            Branch = 2'b00;
            MemRead = 1'b0;
            MemWrite = 1'b0;
            MemtoReg = 1'b0;
            ALUsrc = 1'b0;
            jump = 1'b0;
            case(Func)
                6'h20: ALUCtrl = 4'b1010; //Add signed
                6'h21: ALUCtrl = 4'b0010; //Add unsigned
                6'h22: ALUCtrl = 4'b1110; //Subtract signed
                6'h23: ALUCtrl = 4'b0110; //Subtract unsigned
                6'h24: ALUCtrl = 4'b0000; //AND
                6'h25: ALUCtrl = 4'b0001; //OR
                6'h26: ALUCtrl = 4'b0011; //XOR
                6'h27: ALUCtrl = 4'b1100; //NOR
                6'h2a: ALUCtrl = 4'b1111; //Set less than signed
                6'h2b: ALUCtrl = 4'b0100; //Set less than unsigned
            default: ALUCtrl = 4'bxxxx; //default to AND
            endcase
       end
```

```
else begin
     case(Op)
         6'h08: begin    //addi
                 ALUCtrl = 4'b1010;   //Add ALU content
                 RegWrite = 1'b1;     //Write back to register file
                 RegDst = 1'b0;      //Inst[20:16] write back address
                 Branch = 2'b00;      //No branch performed
                 MemRead = 1'b0;    //No read from data memory
                 MemWrite = 1'b0;    //No write to data memory
                 MemtoReg = 1'b0;    //Write back data comes from ALU
                 ALUsrc = 1'b1;      //Source B comes from SE immediate
                 jump = 1'b0;
                end
         6'h09: begin    //addiu
                 ALUCtrl = 4'b0010;
                 RegWrite = 1'b1;
                 RegDst = 1'b0;
                 Branch = 2'b00;
                 MemRead = 1'b0;
                 MemWrite = 1'b0;
                 MemtoReg = 1'b0;
                 ALUsrc = 1'b1;
                 jump = 1'b0;
                end
         6'h0c: begin    //andi
                 ALUCtrl = 4'b0000;
                 RegWrite = 1'b1;
                 RegDst = 1'b0;
                 Branch = 2'b00;
                 MemRead = 1'b0;
                 MemWrite = 1'b0;
                 MemtoReg = 1'b0;
                 ALUsrc = 1'b1;
                 jump = 1'b0;
                end
         6'h0d: begin    //ori
                 ALUCtrl = 4'b0001;
                 RegWrite = 1'b1;
                 RegDst = 1'b0;
                 Branch = 2'b00;
                 MemRead = 1'b0;
                 MemWrite = 1'b0;
                 MemtoReg = 1'b0;
                 ALUsrc = 1'b1;
                 jump = 1'b0;
                end
```

```
6'h23: begin    //lw
        ALUCtrl = 4'b0010;
        RegWrite = 1'b1;
        RegDst = 1'b0;
        Branch = 2'b00;
        MemRead = 1'b1;
        MemWrite = 1'b0;
        MemtoReg = 1'b1;
        ALUsrc = 1'b1;
        jump = 1'b0;
       end
6'h2b: begin    //sw
        ALUCtrl = 4'b0010;
        RegWrite = 1'b0;
        RegDst = 1'b0;
        Branch = 2'b00;
        MemRead = 1'b0;
        MemWrite = 1'b1;
        MemtoReg = 1'b0;
        ALUsrc = 1'b1;
        jump = 1'b0;
       end
6'h04: begin    //beq
        ALUCtrl = 4'b0110;
        RegWrite = 1'b0;
        RegDst = 1'b0;
        Branch = 2'b01;
        MemRead = 1'b0;
        MemWrite = 1'b0;
        MemtoReg = 1'b0;
        ALUsrc = 1'b0;
        jump = 1'b0;
       end
6'h05: begin    //bnq
        ALUCtrl = 4'b0110;
        RegWrite = 1'b0;
        RegDst = 1'b0;
        Branch = 2'b10;
        MemRead = 1'b0;
        MemWrite = 1'b0;
        MemtoReg = 1'b0;
        ALUsrc = 1'b0;
        jump = 1'b0;
       end
6'h0a: begin    //slti
        ALUCtrl = 4'b1111;
        RegWrite = 1'b1;
        RegDst = 1'b0;
        Branch = 2'b00;
        MemRead = 1'b0;
        MemWrite = 1'b0;
        MemtoReg = 1'b0;
        ALUsrc = 1'b1;
        jump = 1'b0;
       end
6'h0b: begin    //sltiu
        ALUCtrl = 4'b0100;
        RegWrite = 1'b1;
        RegDst = 1'b0;
        Branch = 2'b00;
        MemRead = 1'b0;
        MemWrite = 1'b0;
        MemtoReg = 1'b0;
        ALUsrc = 1'b1;
        jump = 1'b0;
       end
```

```
                6'h02: begin    //j
                        ALUCtrl = 4'b0;
                        RegWrite = 1'b0;
                        RegDst = 1'b0;
                        Branch = 2'b0;
                        MemRead = 1'b0;
                        MemWrite = 1'b0;
                        MemtoReg = 1'b0;
                        ALUsrc = 1'b0;
                        jump = 1'b1;
                        end
                default: begin      //andi default
                            ALUCtrl = 4'b0000;
                            RegWrite = 1'b1;
                            RegDst = 1'b0;
                            Branch = 2'b00;
                            MemRead = 1'b0;
                            MemWrite = 1'b0;
                            MemtoReg = 1'b0;
                            ALUsrc = 1'b1;
                            jump = 1'b0;
                         end
                endcase
        end
    end
endmodule
```

Instruction Memory:

```
module Instruction_Memory(
    input [31:0] Addr,
    output [31:0] Inst_out
    );

    reg [7:0] imem[0:4095]; //2^12 byte addresses

    //Read Memory Contents, 2 bit offset for alignment
    assign Inst_out = {
                        imem[{Addr[11:2],2'b0}+2'd3],
                        imem[{Addr[11:2],2'b0}+2'd2],
                        imem[{Addr[11:2],2'b0}+2'd1],
                        imem[{Addr[11:2],2'b0}+2'd0]
                        };
endmodule
```

Register Files:

```
module regfile32(
    input clk,
    input reset,
    input D_En,
    input [4:0] D_Addr,
    input [4:0] S_Addr,
    input [4:0] T_Addr,
    input [31:0] D,
    output wire [31:0] S,
    output wire [31:0] T
    );

    //Instantiate 32 32-bit registers
    reg [31:0] regArray [0:31];

    //Assign S and T, specific contents of regArray
    assign S = regArray[S_Addr];
    assign T = regArray[T_Addr];

    //Write to regArray
    //regArray[0] inaccesible to overwriting
    always@(posedge clk, posedge reset) begin
        if(reset)
            regArray[0] <= 32'b0; else
        if(D_En && D_Addr)
            regArray[D_Addr] <= D;
    end
endmodule
```

First PC adder:

```
module PCADD(
    input [31:0] Din,
    output [31:0] PCADD_out
    );

    assign PCADD_out = Din + 3'b100;
endmodule
```

Second PC adder:

```
module PCADD1(
    input [31:0] PCADDIn,
    input [31:0] SignEx,
    output [31:0] NextPC
    );
    wire [31:0] SignSL;
    assign SignSL = SignEx << 2;
    assign NextPC = SignSL + PCADDIn;
endmodule
```

Branch:

```
module BranchSrc(
    input [1:0] BranchIn,
    input Zero,
    output BranchOut
    );
    wire notz, and1, and2;
    not
        n1(notz, Zero);

    and
        a1(and1, BranchIn[0], Zero),
        a2(and2, BranchIn[1], notz);
    or
        o1(BranchOut, and1, and2);
endmodule
```

PC Register:

```
module PC(
    input clock,
    input Reset,
    input [31:0] Din,
    output reg [31:0] PC_out
    );
    always@(posedge clock, posedge Reset) begin
        if(Reset)
            PC_out <= 32'b0;
        else
            PC_out <= Din;
    end
endmodule
```

ALU:

```
module ALU(
    input [31:0] A,
    input [31:0] B,
    input [3:0] ALUctrl,
    output reg [31:0] ALUout,
    output reg N,
    output reg C,
    output Z,
    output reg V
    );

    assign Z = (ALUout == 0) ? 1'b1 : 1'b0;
    reg signed [31:0] A_s, B_s;
    always@(*) begin
        A_s = A; B_s = B;
        case(ALUctrl)
            4'b0000: begin  //AND
                        ALUout = A & B;
                        C = 1'bx;
                        V = 1'bx;
                        N = ALUout[31];
                    end
            4'b0001: begin  //OR
                        ALUout = A | B;
                        C = 1'bx;
                        V = 1'bx;
                        N = ALUout[31];
                    end
            4'b0011: begin  //XOR
                        ALUout = A ^ B;
                        C = 1'bx;
                        V = 1'bx;
                        N = ALUout[31];
                    end
            4'b0010: begin  //Add Unsigned
                        {C, ALUout} = A + B;
                        V = C;
                        N = 0;
                    end
            4'b0110: begin  //Subtract unsigned
                        {C, ALUout} = A - B;
                        V = C;
                        N = 0;
                    end
```

```
4'b1100: begin  //NOR
            ALUout = ~(A | B);
            C = 1'bx;
            V = 1'bx;
            N = ALUout[31];
        end
4'b0111: begin  //NOT
            ALUout = ~A;
            C = 1'bx;
            V = 1'bx;
            N = ALUout[31];
        end
4'b1101: begin  //SLL
            ALUout = A << 1;
            C = 1'bx;
            V = 1'bx;
            N = ALUout[31];
        end
4'b1010: begin  //add signed
            {C, ALUout} = A + B;
            if ((A[31] & B[31] & ~ALUout[31]) || (~A[31] & ~B[31] & ALUout[31]))
                V = 1'b1;
            else
                V = 1'b0;
            N = ALUout[31];
        end
4'b1110: begin  //subtract signed
            {C, ALUout} = A - B;
            if ((A[31] & ~B[31] & ~ALUout[31]) || (~A[31] & B[31] & ALUout[31]))
                V = 1'b1;
            else
                V = 1'b0;
            N = ALUout[31];
        end
 4'b0111: begin  //SLL
            ALUout = A << 1;
            C = 1'bx;
            V = 1'bx;
            N = ALUout[31];
        end
```

```
            4'b1111: begin  //SLT signed
                        if ( A_s < B_s )
                             ALUout = 32'b1;
                        else
                             ALUout = 32'b0;
                        C = 1'bx;
                        V = 1'bx;
                        N = ALUout[31];
                    end
            4'b0100: begin  //SLT unsigned
                        if ( A < B )
                            ALUout = 32'b1;
                        else
                            ALUout = 32'b0;
                        C = 1'bx;
                        V = 1'bx;
                        N = ALUout[31];
                    end
            default: begin
                        ALUout = 32'bx;
                        {C, V, N} = 3'bxxx;
                     end
        endcase
    end
endmodule
```

Data Memory:

```
module DataMem(
    input clk,
    input mem_wr,
    input mem_rd,
    input [31:0] addr,
    input [31:0] wr_data,
    output [31:0] rd_data
    );

    reg [7:0] dmem [0:4095];

    // write
    always@(posedge clk) begin
        if(mem_wr) begin
            dmem[addr[11:0] + 2'd3] <= wr_data[7:0];
            dmem[addr[11:0] + 2'd2] <= wr_data[15:8];
            dmem[addr[11:0] + 2'd1] <= wr_data[23:16];
            dmem[addr[11:0] + 2'd0] <= wr_data[31:24];
        end
    end

    //read
    assign rd_data = (mem_rd) ? {   dmem[addr[11:0] + 2'd0],
                                    dmem[addr[11:0] + 2'd1],
                                    dmem[addr[11:0] + 2'd2],
                                    dmem[addr[11:0] + 2'd3] }
                                    : 32'hz;


endmodule
```

Sign Extension:

```
module SignExtension(
    input [15:0] SignExtIn,
    output [31:0] SignExtOut
    );
    assign SignExtOut = {{16{SignExtIn[15]}}, SignExtIn};
endmodule
```

**Results: (what we would do differently next time)**

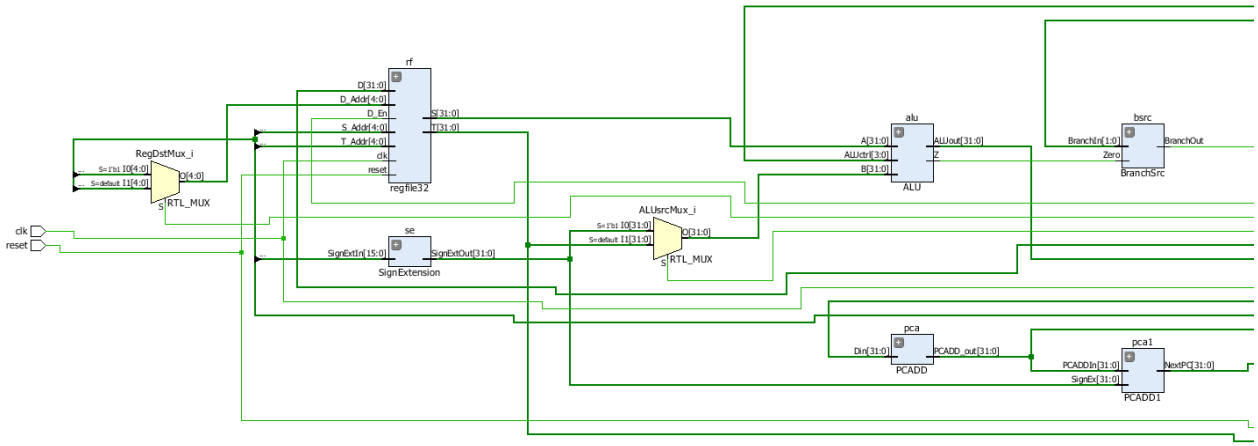
We feel like this time we really started early on the project and got a good feel for how the project worked. I noticed that for this project it was a lot easier to understand the diagram we were given and to translate it into Verilog. I think next time

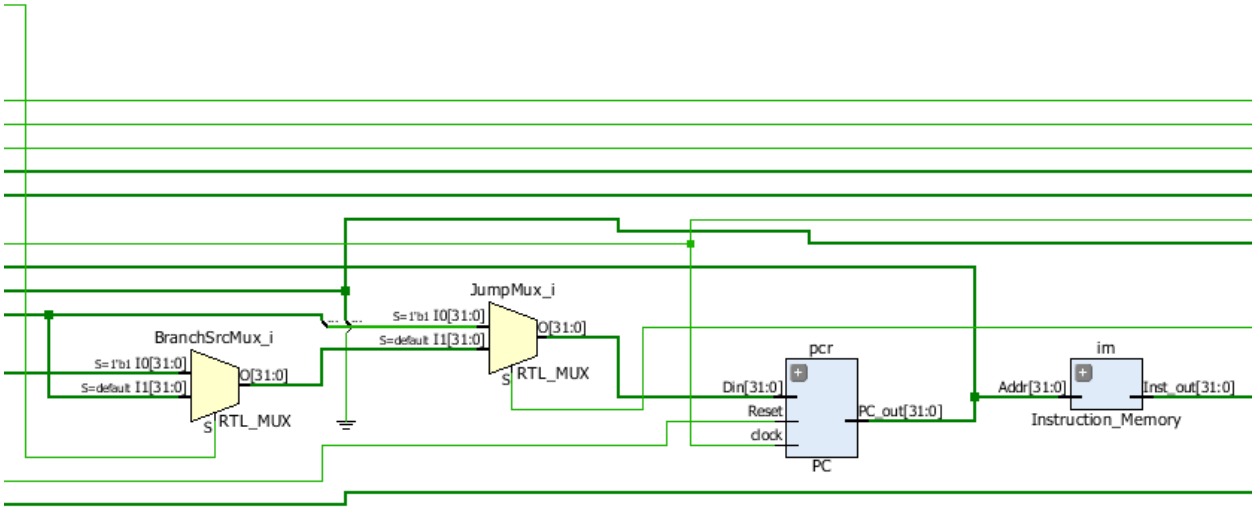
I’m in a class learning a new language like Verilog I’m going to watch some introduction videos on Youtube.

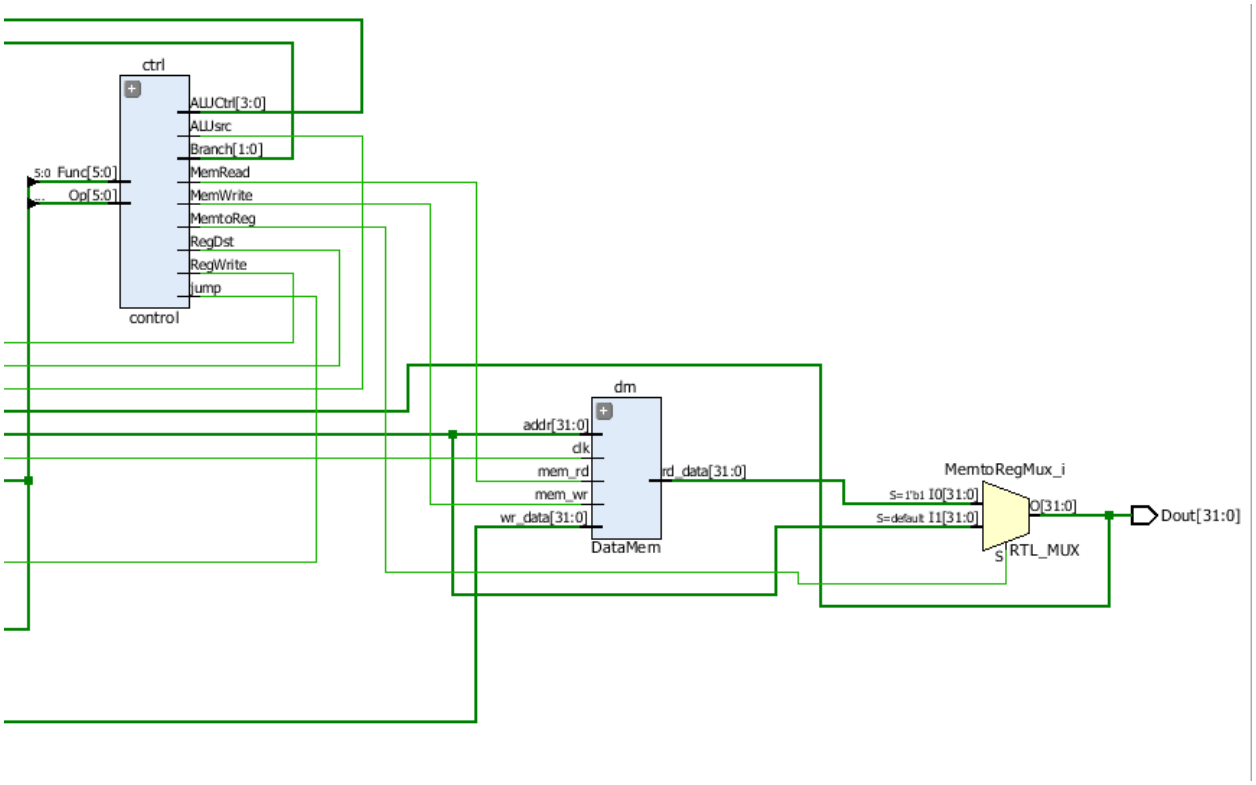
**Conclusion:**

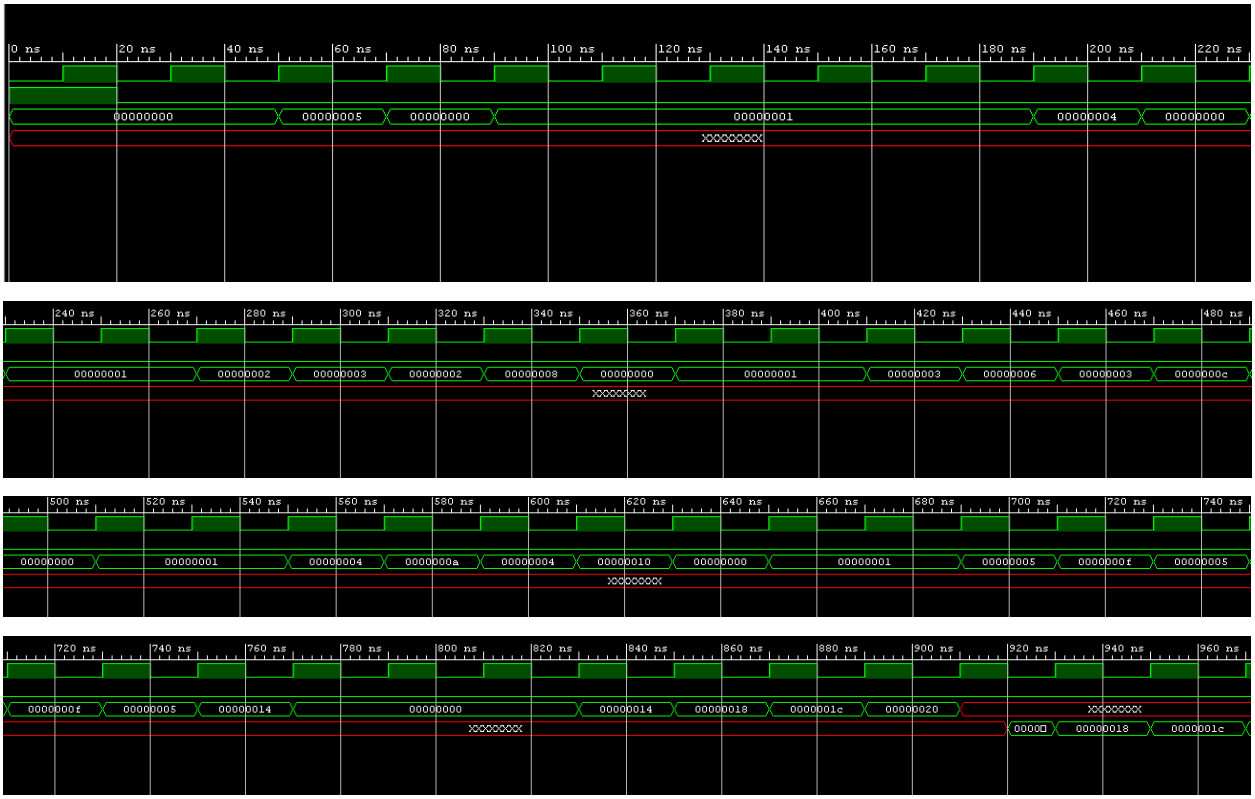
Adding functionality for the J-Type instruction has taught us how interconnected the different R, I and J-Type instructions are in a computer. It was really cool to see how all the labs of the semester came together in the end to form a final project with R,I,and J-Type instructions.

## RTL Schematic, Captured WaveForm:




JumpMux_i
S=1'b1 I0[31:0]
S=default I1[31:0]
O[31:0]
S
RTL_MUX
BranchSrcMux_i
S=1'b1 I0[31:0]
S=default I1[31:0]
O[31:0]
S
RTL_MUX
pcr
Din[31:0]
Reset
clock
PC_out[31:0]
PC
im
Addr[31:0]
Inst_out[31:0]
Instruction_Memory


ctrl
ALUCtrl[3:0]
ALUsrc
Branch[1:0]
MemRead
MemWrite
MemtoReg
RegDst
RegWrite
Jump
5:0 Func[5:0]
Op[5:0]
control
dm
addr[31:0]
clk
mem_rd
mem_wr
wr_data[31:0]
rd_data[31:0]
DataMem
MemtoRegMux_i
S=1'b1 I0[31:0]
S=default I1[31:0]
O[31:0]
S
RTL_MUX
Dout[31:0]




```
# run 1000ns
t= 930.0 ns dm[20]: 00000005
t= 950.0 ns dm[24]: 0000000f
t= 970.0 ns dm[28]: 00000005
t= 990.0 ns dm[32]: 00000014
$finish called at time : 990 ns
```