Lab 5 - I-Type Data Path

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 4/7/2021

Lab 5- I-Type CECS 341 Spring 2021

Goal/Objective:

Our goal is to design an I-Type datapath based on the previously constructed R-Type datapath.

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

To make an I-Type datapath with our current R-Type design, we needed to add an additional PC adder to determine whether we need to transition towards a branching instruction or not, as well as a sign extension module to change part of the instruction from 16 bits to 32 bits, additional control signals in case the opcode is something other than 000000, and some logic gates to make our branch signal function.

```
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 [31:0] SignExtend;

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

```
//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;


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));
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(BranchSrcMux),
           .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
```

This is the modified datapath that is designed to perform both R- and I-Type instructions, including muxes, additional wires, and new module connections.

```
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
    );

    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;
            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
                   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;
                   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;
                   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;
       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;
       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;
       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;
       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;
       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;
       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;
                       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;
                         end
                endcase
        end
    end
endmodule
```

This is the modified version of the control unit, designed to perform R- and I-Type instructions by including other options if the opcode is not 6’b0. AlUCtrl sends out control signals to the ALU to perform the necessary operations. RegWrite enables data traveling from the ALU or data memory to be written inside the register files. RegDst determines whether the destination register is either Rt or Rd using a mux. Branch uses two bits to determine the branching operations (00 is for neither operations, 01 is for branch on equal, 10 is for branch on not equal). MemRead enables data from the data memory to be read from it and travels to the MemtoReg mux. MemWrite enables the results coming from the register files to be written inside the data memory. MemtoReg determines whether what

data is going to the register files, either data from the data memory or the ALU result. ALUsrc determines what data is going to the second input of the ALU, either R[Rt] or SignExtImmediate.

```
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
```

This instruction memory is the same as the R-Type design.

```
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
```

The same goes for the register files.

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

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

As well as the PC adder; however, the output will pass through another adder to determine the next address. It will also encounter a branch mux.

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

This is an additional PC adder that shifts the SignExtImmediate and adds the current PC with the new immediate value. If the branch mux enables the branch option, the result of this adder will be the new address.

```
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
```

This module is designed to determine the enabler for the branch mux using the 2-bit control signal and the zero flag from the ALU.

```
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
```

This PC register is the same as the R-Type datapath.

```
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
```

The same goes for the ALU.

```
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
```

The Data Memory module creates an array for data memory in which each value is stored in words (4 bytes).

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

The Sign Extension module extends the immediate value from 16 bits to 32 bits.

```
`timescale 1ns / 1ps

module Datapath_tb();
    reg clk;
    reg reset;
    wire [31:0] Dout;

    integer i;
    Datapath uut(.clk(clk), .reset(reset), .Dout(Dout));

    always
        #10 clk = ~clk;//makes clk change from rising to falling or vice versa
        task Dump_Dmem; begin
            $timeformat( -9, 1, " ns", 9);
            for ( i = 24; i < 48; i =i+4) begin
                @(posedge clk)
                $display("t=%t dm[%0d]: %h",
                $time, i, {uut.dm.dmem[i], uut.dm.dmem[i+1], uut.dm.dmem[i+2], uut.dm.dmem[i+3]});
                end
            end
         endtask

        initial begin
            clk = 0;
            $readmemh("imem1.dat", uut.im.imem);
            $readmemh("DataMem.dat",uut.dm.dmem);
            //Create testbench here based on lab guide specs
            reset = 1; #20;

            reset = 0; #600;
            Dump_Dmem;
            $finish;
     end
endmodule
```

This new version of the testbench instantiates our new datapath design but displays the data within the data memory instead of the register files.

Results: (what we would do differently next time)

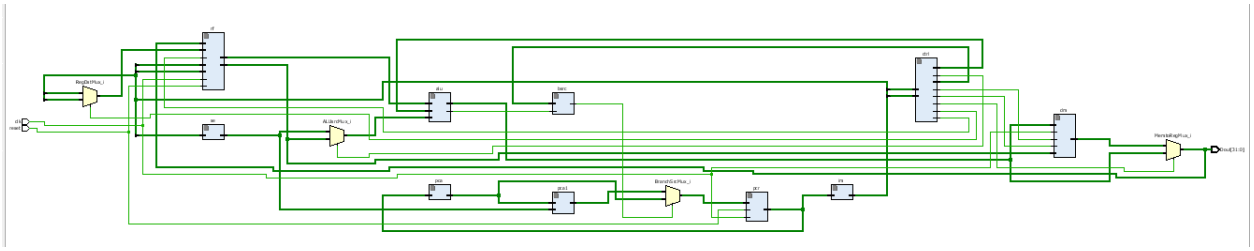
Next time I want to pay more attention to the selections in the book that go over Data path designs. Also, in the next lab we want to start working and collaborating on it more early on.

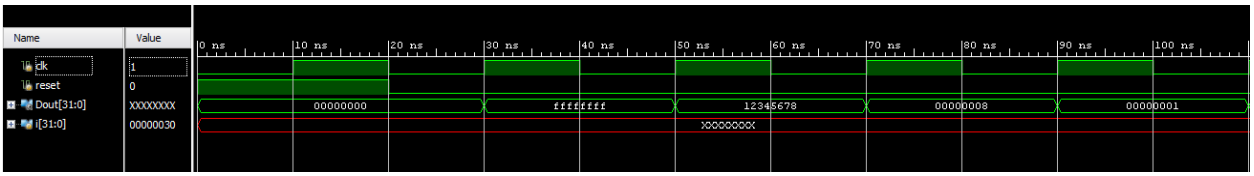
Conclusion:

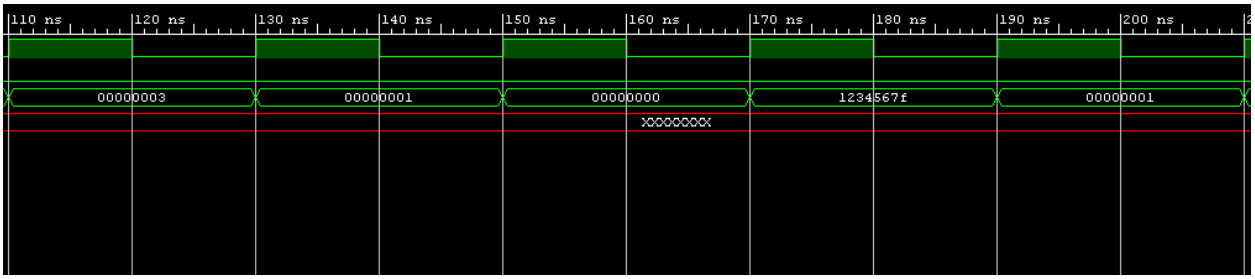
Building the I-type Data path has taught us how the different Data Paths look in a design.Seeing how similar the design of an R-type and I-type was very

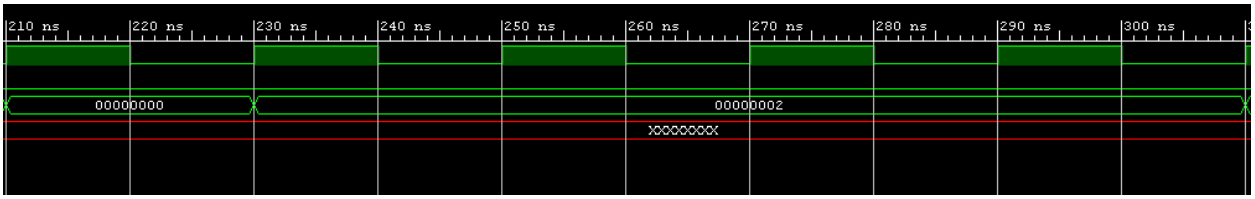
interesting. It was interesting how much of the code we could reuse from the R-Type Data path.

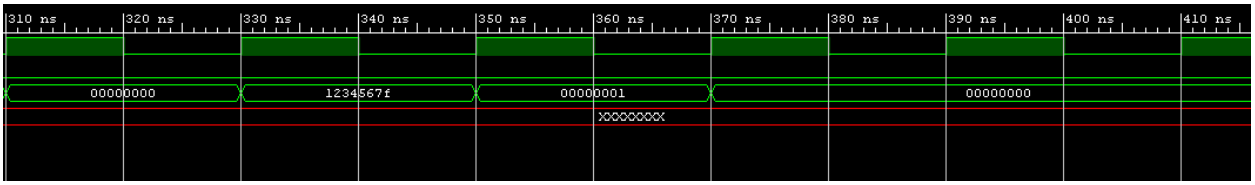
RTL Schematic, Captured WaveForm, Console Output:

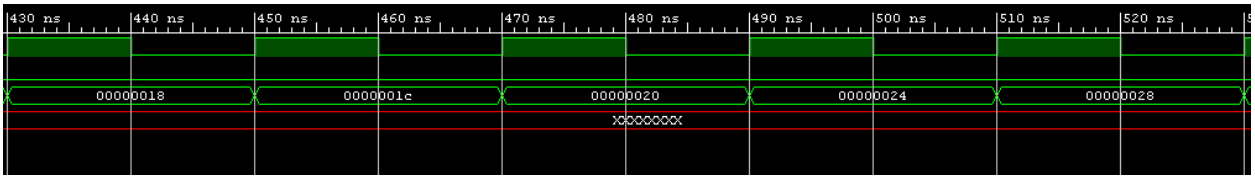


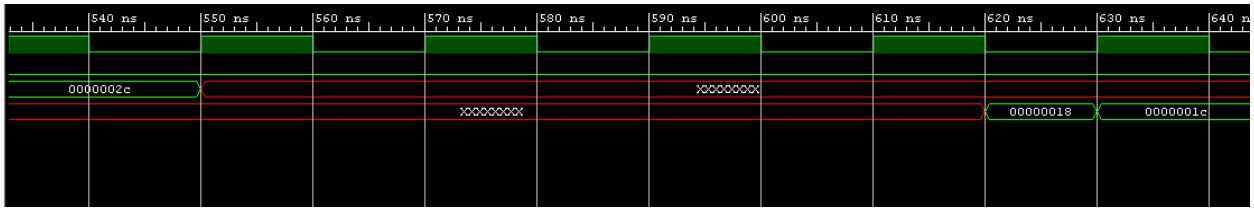


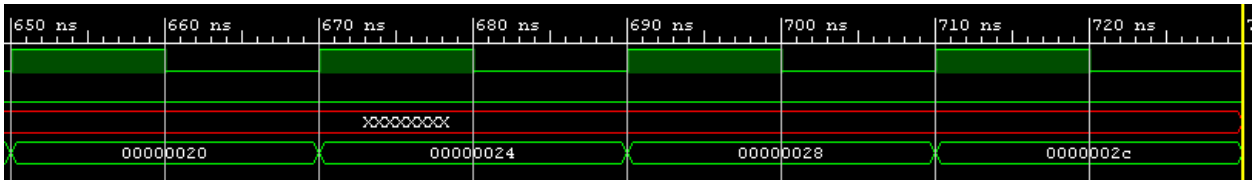













```
t= 630.0 ns dm[24]: 00000002
t= 650.0 ns dm[28]: 00000000
t= 670.0 ns dm[32]: 1234567f
t= 690.0 ns dm[36]: 00000001
t= 710.0 ns dm[40]: 00000000
t= 730.0 ns dm[44]: 00000000
```

Table 2 - Initial Contents of Data Memory:

Memory Location	Initial Value
0	00
1	00
2	00
3	00
4	FF
5	FF
6	FF
7	FF
8	12
9	34
10	56

11	78
12	00
13	00
14	00
15	08
16	00
17	00
18	00
19	01
20	00
21	00
22	00
23	03

Register Location	Data
$t0	00 00 00 02
$t1	00 00 00 00
$t2	12 34 56 7F
$t3	00 00 00 01

$t4	00 00 00 00
$t5	00 00 00 00

Memory Location	Final Value
24	00
25	00
26	00
27	02
28	00
29	00
30	00
31	00
32	12
33	34
34	56
35	7F
36	00
37	00
38	00
39	01
40	00
41	00
42	00

43	00
44	00
45	00
46	00
47	00

341 Lab 5 Work

```
lw $t0, 0($zero)     $t0 ← M[0] + offset 0
lw $t1, 4($zero)     $t1 ← M[0] + offset 4
lw $t2, 8($zero)     $t2 ← M[0] + offset 8
lw $t3, 12($zero)    $t3 ← M[0] + offset 12
lw $t4, 16($zero)    $t4 ← M[0] + offset 16
lw $t5, 20($zero)    $t5 ← M[0] + offset 20

insts: addi $t0, $t0, 1      $t0 ← value at $t0 + 1
       andi $t1, $t1, 0      $t1 ← value at $t1 AND 0
       ori $t2, $t2, 7       $t2 ← value at $t2 OR 7
       slti $t3, $t3, 9      $t3 ← value at $t3 < 9 if true $t3 ← 1 else $t3 ← 0
       sltiu $t4, $t4, 0     $t4 ← value at $t4 < 0 if true $t4 ← 1 else $t4 ← 0
       sub $t5, $t5, $t0     $t5 ← value of $t5 - value of $t0
       beq $t5, $zero, store $t5 == $zero → go to store else cont.
       bne $t5, $zero, insts $t5 != $zero → go to insts else go to store

store: sw $t0, 24($zero)     $t0 ← $zero + 24 offset into memory
       sw $t1, 28($zero)     $t1 ← $zero + 28 offset into memory
       sw $t2, 32($zero)     $t2 ← $zero + 32 offset into memory
       sw $t3, 36($zero)     $t3 ← $zero + 36 offset into memory
       sw $t4, 40($zero)     $t4 ← $zero + 40 offset into memory
       sw $t5, 42($zero)     $t5 ← $zero + 42 offset into memory
```