Lab 4 - R-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 2/7/2021

Lab 4- R-Type CECS 341 Spring 2021

Goal/Objective:

Our goal is to design an R-Type MIPS datapath with an ALU, Instruction Memory, and Register file.

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

To make an R-Type MIPS datapath we needed to connect the separate components in our program(ALU, Instruction Memory..) and add a slt(shift less than) to our ALU.

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

    wire [31:0] pcaOutputWire;
    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 N;
    wire Z;
    wire C;
    wire V;
    control ctrl(.Op(instructionMemOutputWire[31:26]), .Func(instructionMemOutputWire[5:0]), .RegWrite(controlRegRWriteOutputWire), .ALUCtrl(controlALUCntlOutputWire));
    Instruction_Memory im(.Addr(pcrOutputWire), .Inst_out(instructionMemOutputWire));
    regfile32 rf(.clk(clk), .reset(reset), .D_En(controlRegRWriteOutputWire), .D_Addr(instructionMemOutputWire[15:11]), .S_Addr(instructionMemOutputWire[25:21]), .T_Addr(instructionMemOutputWire[20:16]),
        .D(ALUOutputWire), .S(rfRSOutputWire), .T(rtRTOutputWire));
    PCADD pca(.Din(pcrOutputWire), .PCADD_out(pcaOutputWire));
    PC pcr(.clock(clk), .Reset(reset), .Din(pcaOutputWire), .PC_out(pcrOutputWire));
    ALU alu(.A(rfRSOutputWire), .B(rtRTOutputWire), .ALUctrl(controlALUCntlOutputWire), .ALUout(ALUOutputWire), .N(N), .C(C), .Z(Z), .V(V));
    assign Dout = ALUOutputWire;
endmodule
```

This Datapath interface includes connections to the control unit, instruction memory, register files, the PC register and adder, and the ALU that simulates the entire Datapath hardware.

```
module control(
    input [5:0] Op,
    input [5:0] Func,
    output reg RegWrite,
    output reg [3:0] ALUCtrl
    );

    always@(*) begin
       if (Op == 6'b0) begin
            RegWrite = 1'b1;
            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
            RegWrite = 1'b0;
            ALUCtrl = 4'bxxxx;
       end
    end
endmodule
```

This control unit takes in the first and last 6 bits of the instruction code outputted from the instruction memory and decodes it to 4-bit control signals and an enabler RegWrite for the register files.

```
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 stores instruction code in these registers with a total of 4096 addresses.

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

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

This PC Adder adds 4 bits to the previous PC output in order to transition to the next address.

```
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 holds the current address for the instruction memory. Whenever a clock cycle turns to

1, a new instruction address is outputted. However, if Reset is activated, then the instruction address will turn to all zeros.

```
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
```

These register files are instantiated and assigned to hold specific contents in an array of 32 32-bit registers, where 32-bit values are written to and read from in 5-bit addresses.

```
        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
```

These are the additional control signal inputs added into the ALU to calculate signed and unsigned set less than.

```
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_RegFile; begin
            $timeformat( -9, 1, " ns", 9);
            for ( i = 0; i < 32; i =i+1) begin
                @(posedge clk)
                $display("t=%t rf[%0d]: %h",
                $time, i, uut.rf.regArray[i]);
                end
            end
         endtask

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

            reset = 0; #200;
            Dump_RegFile;
            $finish;
     end
endmodule
```

This testbench instantiates our datapath simulation by utilizing the instructions and registers given to us from imem.dt and regfile.dt, and simulates it according to the reset and clock specifications presented and displays the calculations of each instruction in the imem.dt file via Dump_RegFile.

Results: (what we would do differently next time)

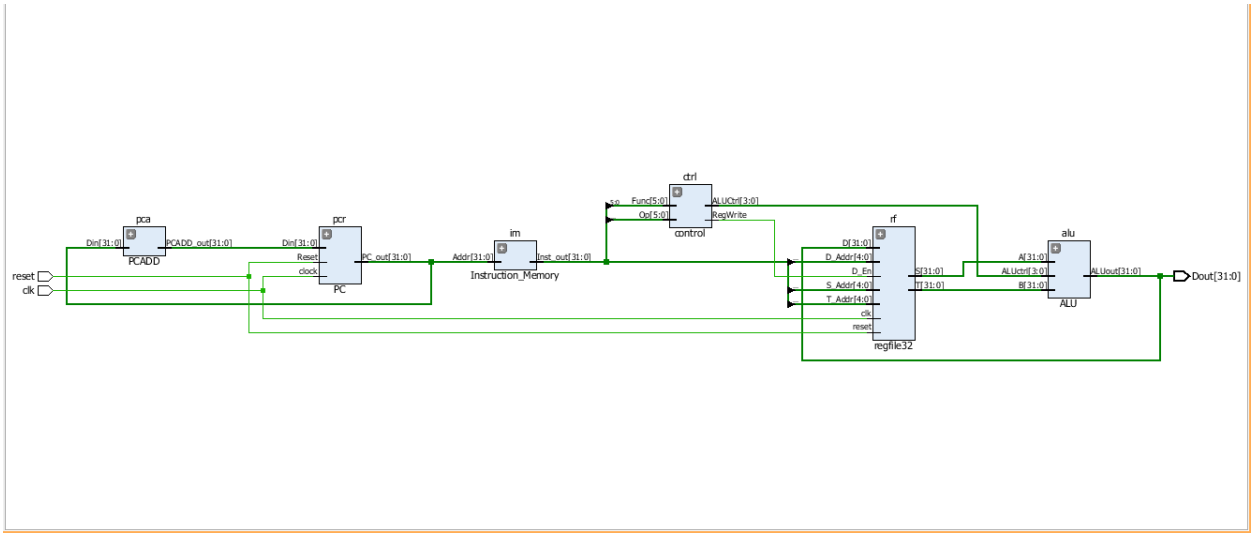
Next time I work on a lab for this class I want to go to office hours sooner to make sure I understand

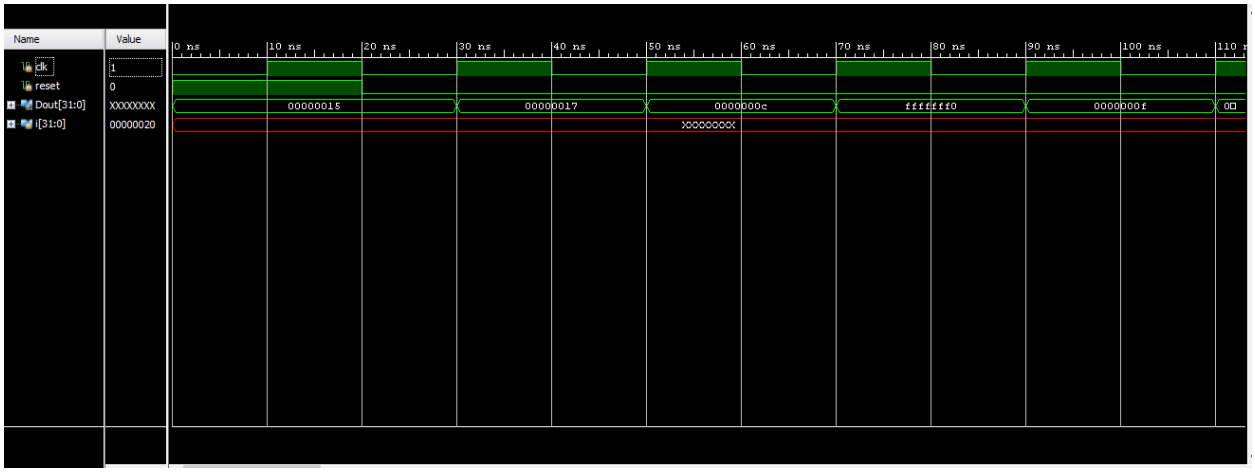
the goal of the lab and all of the components that go into making the lab.

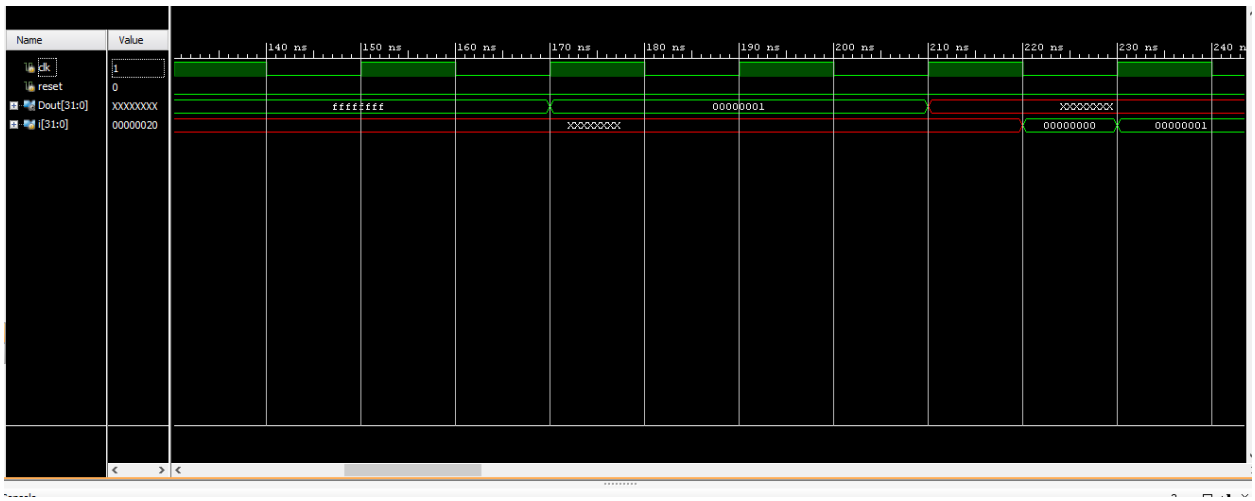
Conclusion:

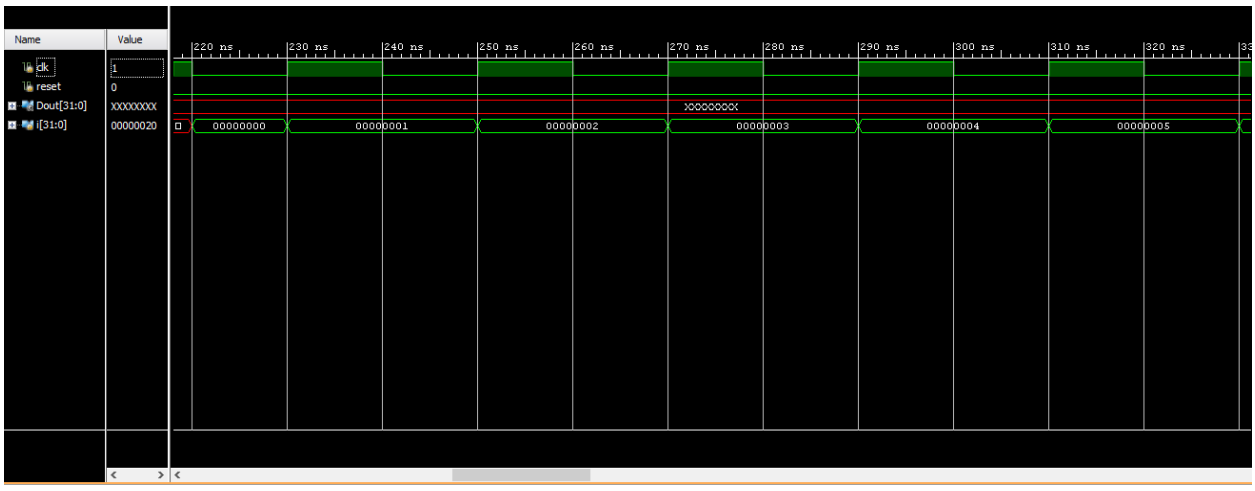
Building the MIPS R-Type datapath has given me a better understanding of how the computer works, and how many different parts of the computer need to be working perfectly for the programs to run correctly. Seeing the different connections between different parts of computers has also made me appreciate more how complex computers are and how much work goes into making them run smoothly.

RTL Schematic, Captured WaveForm:




Name
Value
clk
reset
Dout[31:0]
i[31:0]
1
0
XXXXXXXX
00000020
0 ns
10 ns
20 ns
30 ns
40 ns
50 ns
60 ns
70 ns
80 ns
90 ns
100 ns
00000015
00000017
0000000c
fffffff0
0000000f
XXXXXXXX


Name
Value
clk
reset
Dout[31:0]
i[31:0]
1
0
XXXXXXXX
00000020
140 ns
150 ns
160 ns
170 ns
180 ns
190 ns
200 ns
210 ns
220 ns
230 ns
240 n
ffffffff
00000001
XXXXXXXX
00000000
00000001


Name
Value
clk
reset
Dout[31:0]
i[31:0]
1
0
XXXXXXXX
00000020
220 ns
230 ns
240 ns
250 ns
260 ns
270 ns
280 ns
290 ns
300 ns
310 ns
320 ns
XXXXXXXX
00000000
00000001
00000002
00000003
00000004
00000005

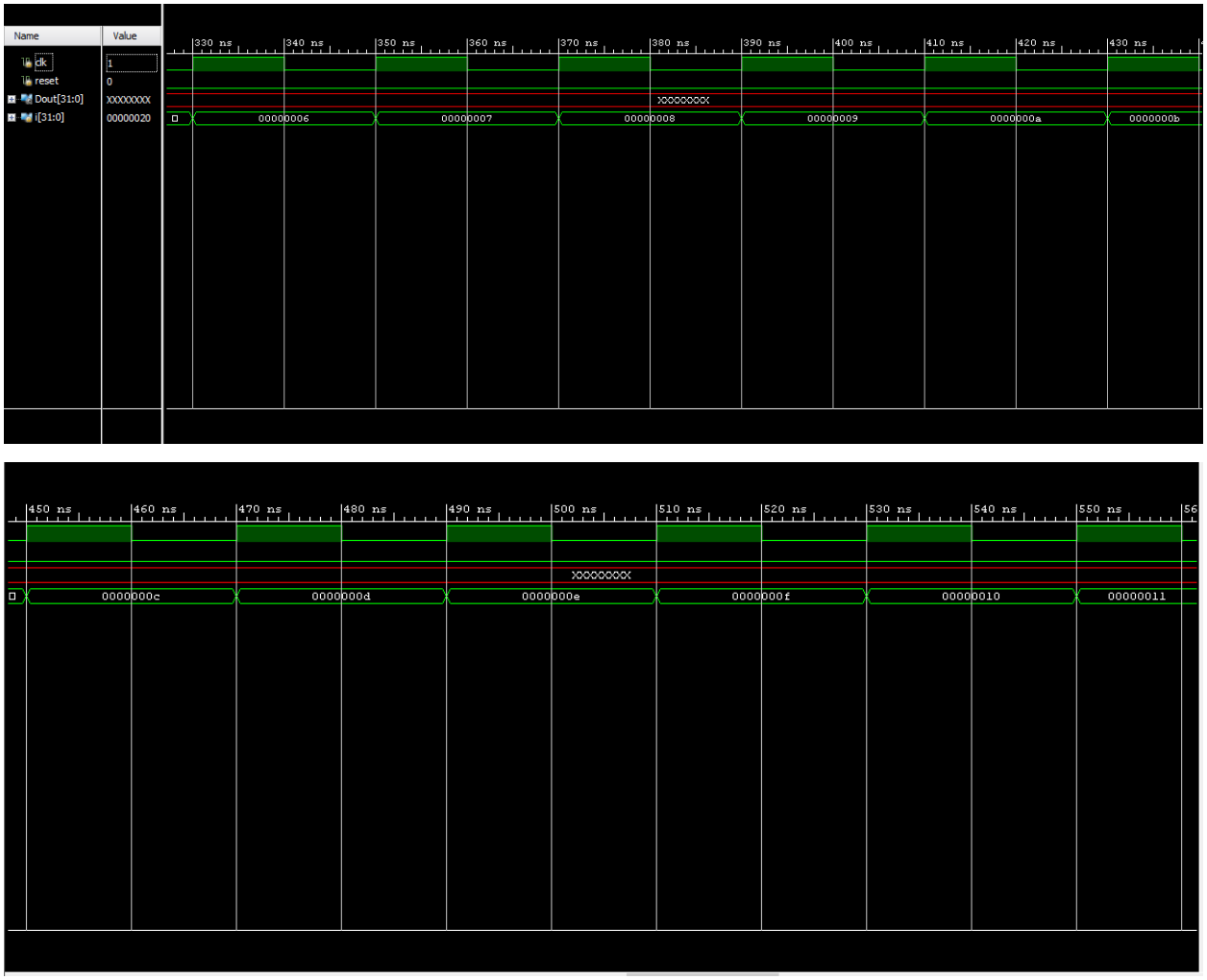


Table 3 - Initial Values of Registers:

No.	Reg.	Calculated		Simulated	
		$Initial_{hex}$	$Final_{hex}$	$Initial_{hex}$	$Final_{hex}$
0	$Zero	00000000	00000000	00000000	00000000
1	$at	00000000	00000000	00000000	00000000
2	$v0	00000000	00000000	00000000	00000000
3	$v1	00000000	00000000	00000000	00000000
4	$a0	00000000	00000000	00000000	00000000

5	$a1	00000000	00000000	00000000	00000000
6	$a2	00000000	00000000	00000000	00000000
7	$a3	00000000	00000000	00000000	00000000
8	$t0	00000009	00000015	00000009	00000015
9	$t1	0000000A	00000017	0000000A	00000017
10	$t2	0000000B	0000000C	0000000B	0000000C
11	$t3	0000000C	FFFFFFF0	0000000C	FFFFFFF0
12	$t4	0000000D	0000000F	0000000D	0000000F
13	$t5	0000000E	0000001F	0000000E	0000001F
14	$t6	0000000F	0000000F	0000000F	0000000F
15	$t7	00000010	00000010	00000010	00000010
16	$s0	00000011	11111111	00000011	11111111
17	$s1	00000012	11111111	00000012	11111111
18	$s2	00000013	00000001	00000013	00000001
19	$s3	00000014	00000001	00000014	00000001
20	$s4	00000015	00000015	00000015	00000015
21	$s5	00000016	00000016	00000016	00000016
22	$s6	00000017	00000017	00000017	00000017
23	$s7	00000018	00000018	00000018	00000018
24	$t8	00000019	00000019	00000019	00000019
25	$t9	0000001A	0000001A	0000001A	0000001A
26	$k0	00000000	00000000	00000000	00000000

27	$k1	00000000	00000000	00000000	00000000
28	$gp	00000000	00000000	00000000	00000000
29	$sp	00000000	00000000	00000000	00000000
30	$fp	00000000	00000000	00000000	00000000
31	$ra	00000000	00000000	00000000	00000000