

**本科实验报告**

|  |  |
| --- | --- |
| 课程名称： | 计算机组成 |
| 姓 名： | TANG ANNA YONGQI |
| 学 院： | 计算机科学与技术学院 |
| 专 业： | 计算机科学与技术（中加班）留学生 |
| 学 号： | 3180300155 |
| 生活照： |  |
| 指导教师： | 刘海风，洪奇军 |

2020年 5月24日

Lab 10 – Multicycle CPU Datapath Design

**Name:** Anna Yongqi Tang **ID:** 3180300155 **Major:** 计算机科学与技术（中加班）留学生

**Course:** Computer Organization

**Date:** 2020-05-24 **Instructor:** 洪奇军

1. **Method and Experimental Steps**

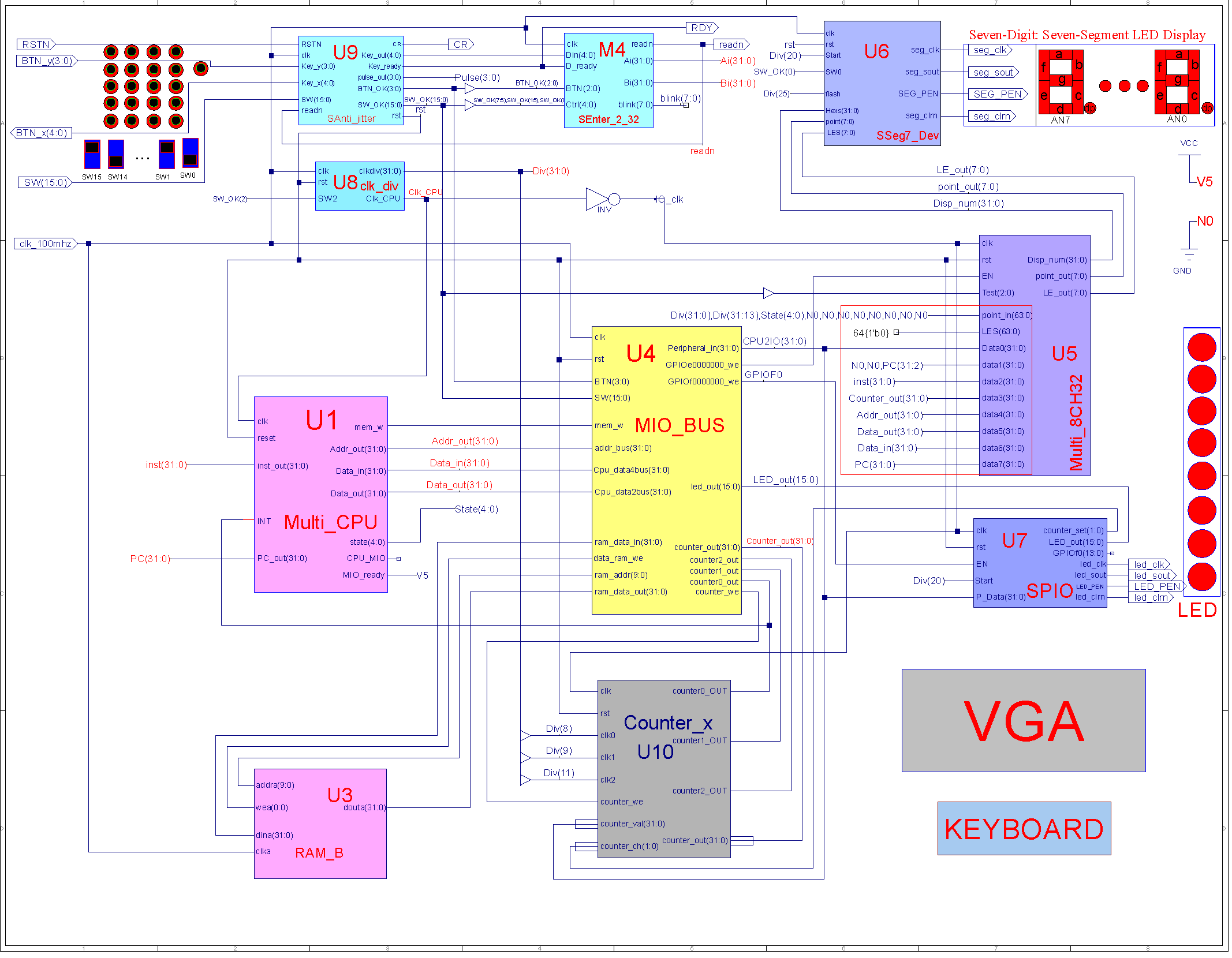


Figure 1 - topMod.sch

Lab 10 is a continuation of the multicycle CPU design from lab 9. Like before, an HDL implementation was used to implement the top module instead of the schematic pictured above. The main task of this experiment was to replace the Multi\_CPU’s M\_datapath ngc file with an actual implementation. The .ucf for this program came from the provided courseware, and is linked to topMod.v. Synthesis had minimal warnings, and implementation was successful. A programmable file has been generated and is ready for testing on the SWORD board. The above diagram was retrieved from the lab PowerPoint. No schematic has been drawn for this experiment.

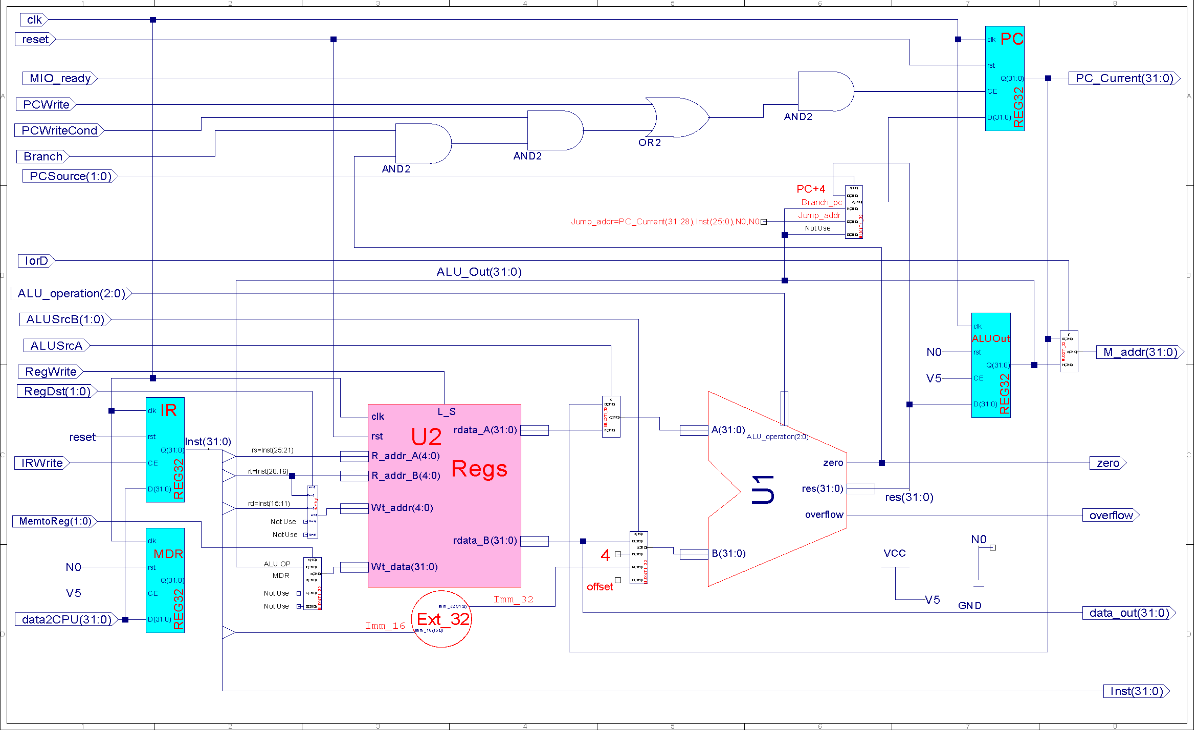


Figure 2 – M\_datapath.sch

Instead of using the suggested schematic (pictured above), the M\_datapath was constructed in HDL as well. The big differences of the multicycle datapath are the IR, MDR, and ALUOut. The ALU and Regs modules come from the single cycle implementation in lab 4. This datapath also supports 9 instructions: add, sub, and, or, slt, nor, lw, sw, beq and j. Lastly, M\_datapath is a module of the multi\_cpu component.



Figure 3 – 3180300155\_TANGANNAYONGQI\_10

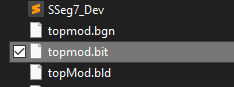
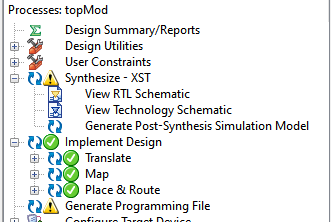


Figure 4 - .bit file generation Figure 5 - .bit file generated in directory

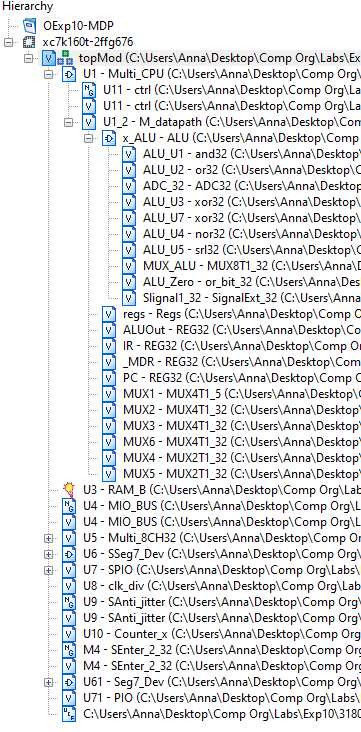


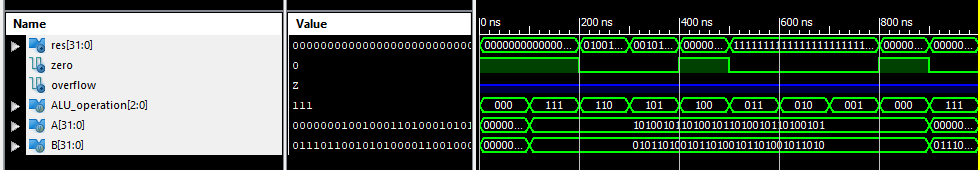
Figure 6 - file hierarchy

1. **Simulations and Observations**

This lab requires a MIPS program (MCPU\_DEMO9.coe) to be tested on the multicyle CPU. The demo will be performed later in a future exercise. The ALU, Regs and M\_datapath modules were simulated in this lab.

*ALU Simulation*

The code used for this simulation was taken from previous labs. The ALU\_operation shows that all nine instructions have been tested.



*aluSim.v*

module ALU\_ALU\_sch\_tb();

// Inputs

reg [2:0] ALU\_operation;

reg [31:0] A;

reg [31:0] B;

// Output

wire [31:0] res;

wire zero;

wire overflow;

// Bidirs

// Instantiate the UUT

ALU UUT (

.ALU\_operation(ALU\_operation),

.res(res),

.zero(zero),

.overflow(overflow),

.A(A),

.B(B)

);

// Initialize Inputs

initial begin

A = 0;

B = 0;

ALU\_operation = 0;

#100;

// Wait 100 ns for global reset to finish

// Add stimulus here

A=32'hA5A5A5A5;

B=32'h5A5A5A5A;

ALU\_operation =3'b111;

#100;

ALU\_operation =3'b110;

#100;

ALU\_operation =3'b101;

#100;

ALU\_operation =3'b100;

#100;

ALU\_operation =3'b011;

#100;

ALU\_operation =3'b010;

#100;

ALU\_operation =3'b001;

#100;

ALU\_operation =3'b000;

#100;

A=32'h01234567;

B=32'h76543210;

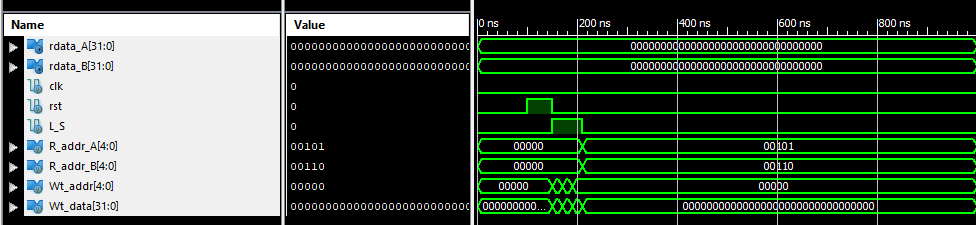
ALU\_operation =3'b111;

end

endmodule

*Regs Simulation*

The code used for this simulation was also taken from a previous lab.



*regsSim.v*

module regsSim;

// Inputs

reg clk;

reg rst;

reg L\_S;

reg [4:0] R\_addr\_A;

reg [4:0] R\_addr\_B;

reg [4:0] Wt\_addr;

reg [31:0] Wt\_data;

// Outputs

wire [31:0] rdata\_A;

wire [31:0] rdata\_B;

// Instantiate the Unit Under Test (UUT)

Regs uut (

.clk(clk),

.rst(rst),

.L\_S(L\_S),

.R\_addr\_A(R\_addr\_A),

.R\_addr\_B(R\_addr\_B),

.Wt\_addr(Wt\_addr),

.Wt\_data(Wt\_data),

.rdata\_A(rdata\_A),

.rdata\_B(rdata\_B)

);

initial begin

clk = 0;

rst = 0;

L\_S = 0;

R\_addr\_A = 0;

R\_addr\_B = 0;

Wt\_addr = 0;

Wt\_data = 0;

// Wait 100 ns for global reset to finish

#100;

// Add stimulus here

rst = 1;

#50;

rst = 0;

L\_S = 1;

R\_addr\_A = 0;

R\_addr\_B = 0;

Wt\_addr = 5;

Wt\_data = 32'hA5A5A5A5;

#20;

L\_S = 1;

R\_addr\_A = 0;

R\_addr\_B = 0;

Wt\_addr = 6;

Wt\_data = 32'h55AA55AA;

#20;

L\_S = 1;

R\_addr\_A = 0;

R\_addr\_B = 0;

Wt\_addr = 0;

Wt\_data = 32'hAAAA5555;

#20;

L\_S = 0;

R\_addr\_A = 5;

R\_addr\_B = 6;

Wt\_addr = 0;

Wt\_data = 0;

#20;

end

endmodule

*M\_Datapath Simulation*

Eight instructions; add, sub, and, or, nor, slt, lw, sw and beq were simulated. The instruction encoding was first fed into the data2CPU port, and the control signals were set by `signals. For each state/cycle of the multicycle FSM, the control signals and ALU\_operation were updated. This can be observed with how different instruction classes require a different number of cycles.



*M\_DatapathSim.v*

module M\_datapathSim;

// Inputs

reg clk;

reg reset;

reg MIO\_ready;

reg IorD;

reg IRWrite;

reg [1:0] RegDst;

reg RegWrite;

reg [1:0] MemtoReg;

reg ALUSrcA;

reg [1:0] ALUSrcB;

reg [1:0] PCSource;

reg PCWrite;

reg PCWriteCond;

reg Branch;

reg [2:0] ALU\_operation;

reg [31:0] data2CPU;

// Outputs

wire [31:0] PC\_Current;

wire [31:0] Inst;

wire [31:0] data\_out;

wire [31:0] M\_addr;

wire zero;

wire overflow;

// Instantiate the Unit Under Test (UUT)

M\_datapath uut (

.clk(clk),

.reset(reset),

.MIO\_ready(MIO\_ready),

.IorD(IorD),

.IRWrite(IRWrite),

.RegDst(RegDst),

.RegWrite(RegWrite),

.MemtoReg(MemtoReg),

.ALUSrcA(ALUSrcA),

.ALUSrcB(ALUSrcB),

.PCSource(PCSource),

.PCWrite(PCWrite),

.PCWriteCond(PCWriteCond),

.Branch(Branch),

.ALU\_operation(ALU\_operation),

.PC\_Current(PC\_Current),

.data2CPU(data2CPU),

.Inst(Inst),

.data\_out(data\_out),

.M\_addr(M\_addr),

.zero(zero),

.overflow(overflow)

);

initial begin

// Initialize Inputs

`define signals {PCWrite, PCWriteCond, IorD, IRWrite, MemtoReg, PCSource, ALUSrcB, ALUSrcA, RegWrite, RegDst}

clk = 0;

reset = 1;

MIO\_ready = 1;

IorD = 0;

IRWrite = 0;

RegDst = 0;

RegWrite = 0;

MemtoReg = 0;

ALUSrcA = 0;

ALUSrcB = 0;

PCSource = 0;

PCWrite = 0;

PCWriteCond = 0;

Branch = 0;

ALU\_operation = 0;

data2CPU = 0;

#100;

reset = 0;

//add r3, r2, r2

data2CPU = 32'b000000\_00010\_00010\_00011\_00000\_100000;

`signals = 14'b1\_00\_1000\_0010\_000;

ALU\_operation = 3'b000;

`signals = 14'b0\_00\_0000\_0110\_000;

`signals = 14'b0\_00\_0000\_0001\_000;

ALU\_operation = 3'b010;

`signals = 14'b0\_00\_0000\_0001\_101;

#100;

//sub r4, r0, r3

data2CPU = 32'b000000\_00000\_00011\_00100\_00000\_100010;

`signals = 14'b1\_00\_1000\_0010\_000;

ALU\_operation = 3'b000;

`signals = 14'b0\_00\_0000\_0110\_000;

`signals = 14'b0\_00\_0000\_0001\_000;

ALU\_operation = 3'b110;

`signals = 14'b0\_00\_0000\_0001\_101;

#100;

//and r5, r3, r4

data2CPU = 32'b000000\_00100\_00011\_00101\_00000\_100100;

`signals = 14'b1\_00\_1000\_0010\_000;

ALU\_operation = 3'b000;

`signals = 14'b0\_00\_0000\_0110\_000;

`signals = 14'b0\_00\_0000\_0001\_000;

`signals = 14'b0\_00\_0000\_0001\_101;

#100;

//or r6, r2, r4

data2CPU = 32'b000000\_00100\_00010\_00110\_00000\_010110;

`signals = 14'b1\_00\_1000\_0010\_000;

ALU\_operation = 3'b000;

`signals = 14'b0\_00\_0000\_0110\_000;

`signals = 14'b0\_00\_0000\_0001\_000;

ALU\_operation = 3'b001;

`signals = 14'b0\_00\_0000\_0001\_101;

#100;

//nor r1, r0, r0

data2CPU = 32'b000000\_00000\_00000\_00001\_00000\_100111;

`signals = 14'b1\_00\_1000\_0010\_000;

ALU\_operation = 3'b000;

`signals = 14'b0\_00\_0000\_0110\_000;

`signals = 14'b0\_00\_0000\_0001\_000;

ALU\_operation = 3'b100;

`signals = 14'b0\_00\_0000\_0001\_101;

#100;

//slt r2, r0, r1

data2CPU = 32'b000000\_00000\_00001\_00010\_00000\_101010;

`signals = 14'b1\_00\_1000\_0010\_000;

ALU\_operation = 3'b000;

`signals = 14'b0\_00\_0000\_0110\_000;

`signals = 14'b0\_00\_0000\_0001\_000;

ALU\_operation = 3'b111;

`signals = 14'b0\_00\_0000\_0001\_101;

#100;

//lw r1, 4(r0)

data2CPU = 32'b100011\_00000\_00001\_00000\_00000\_000100;

`signals = 14'b1\_00\_1000\_0010\_000;

ALU\_operation = 3'b000;

`signals = 14'b0\_00\_0000\_0110\_000;

`signals = 14'b0\_00\_0000\_0101\_000;

ALU\_operation = 3'b010;

`signals = 14'b0\_01\_0000\_0101\_000;

`signals = 14'b0\_00\_0010\_0000\_100;

#100;

//sw r1, 8(r0)

data2CPU = 32'b101011\_00000\_00001\_00000\_00000\_001000;

`signals = 14'b1\_00\_1000\_0010\_000;

ALU\_operation = 3'b000;

`signals = 14'b0\_00\_0000\_0110\_000;

`signals = 14'b0\_00\_0000\_0101\_000;

ALU\_operation = 3'b010;

`signals = 14'b0\_01\_0000\_0101\_000;

#100;

//beq r0, r0, 4

data2CPU = 32'b000100\_00000\_00000\_00000\_00000\_000100;

`signals = 14'b1\_00\_1000\_0010\_000;

ALU\_operation = 3'b000;

`signals = 14'b0\_00\_0000\_0110\_000;

`signals = 14'b0\_10\_0000\_1001\_000;

ALU\_operation = 3'b110;

Branch = 1;

#100;

end

always begin

clk=0;

#10;

clk=1;

#10;

end

endmodule

**3. Conclusion**

This week’s lab was an overview to a multicycle CPU’s datapath. It reinforced the material that was taught in class, but there were some differences such as the lack of the A and B registers. Comparing the simulations of a single cycle datapath to the one in this lab, we can tell that this implementation is a lot more efficient in runtime. I look forward to continuing recreating the other modules from scratch, and adjusting to using HDL instead of schematics for the top module.

1. **Source Code**

All modules and components were either retrieved from previous labs or directly taken from the given files.

*M\_datapath – M\_datapath\_IO.v*

module M\_datapath(input clk,

input reset,

input MIO\_ready,

input IorD,

input IRWrite,

input[1:0] RegDst,

input RegWrite,

input[1:0]MemtoReg,

input ALUSrcA,

input[1:0]ALUSrcB,

input[1:0]PCSource,

input PCWrite,

input PCWriteCond,

input Branch,

input[2:0]ALU\_operation,

output[31:0]PC\_Current,

input[31:0]data2CPU,

output[31:0]Inst,

output[31:0]data\_out,

output[31:0]M\_addr,

output zero,

output overflow

);

wire [31:0] rdata\_A, rdata\_B, ALU\_Out, MDR, w\_reg\_data, Alu\_A, Alu\_B, res, PC\_Next;

wire[4:0] reg\_Rs\_addr\_A = Inst[25:21];

wire[4:0] reg\_Rt\_addr\_B = Inst[20:16];

wire[4:0] reg\_rd\_addr = Inst[15:11];

wire[4:0] reg\_Wt\_addr;

wire[15:0] imm = Inst[15:0];

wire[31:0] imm\_32 = {{16{imm[15]}},imm};

wire N0 = 1'b0, V5 = 1'b1;

wire CE;

assign CE = MIO\_ready && (PCWrite || (PCWriteCond && zero&&Branch));

assign data\_out = rdata\_B;

ALU x\_ALU(.A(Alu\_A),

.B(Alu\_B),

.ALU\_operation(ALU\_operation),

.res(res),

.zero(zero),

.overflow(overflow)

);

Regs regs(.clk(clk),

.rst(reset),

.R\_addr\_A(reg\_Rs\_addr\_A), //Inst(25:21)

.R\_addr\_B(reg\_Rt\_addr\_B), //Inst(20:16)

.Wt\_addr(reg\_Wt\_addr),

.Wt\_data(w\_reg\_data),

.L\_S(RegWrite),

.rdata\_A(rdata\_A),

.rdata\_B(rdata\_B)

);

REG32 ALUOut(.clk(clk),

.rst(N0),

.CE(V5),

.D(res),

.Q(ALU\_Out)

);

REG32 IR (.clk(clk),

.rst(reset),

.CE(V5),

.D(data2CPU),

.Q(Inst)

);

REG32 \_MDR(.clk(clk),

.rst(N0),

.CE(V5),

.D(data2CPU),

.Q(MDR)

);

REG32 PC (.clk(clk),

.rst(reset),

.CE(CE),

.D(PC\_next),

.Q(PC\_Current)

);

MUX4T1\_5 MUX1(.I0(reg\_Rt\_addr\_B), //reg addr=IR[21:16]

.I1(reg\_rd\_addr), //reg addr=IR[15:11]

.I2(5'b11111), // not use

.I3(5'b00000), // not use

.s(RegDst),

.o(reg\_Wt\_addr)

);

MUX4T1\_32 MUX2(.I0(ALU\_Out), //ALU OP

.I1(MDR),

.I2(32'h00000000), // not use

.I3(32'h00000000), // not use

.s(MemtoReg),

.o(w\_reg\_data)

);

MUX4T1\_32 MUX3(.I0(data\_out), //reg out B

.I1(32'h00000004), //4 for PC+4

.I2(imm\_32[31:0]),

.I3({imm\_32[29:0],N0,N0}),

.s(ALUSrcB),

.o(Alu\_B)

);

MUX4T1\_32 MUX6(.I0(res[31:0]),

.I1(ALU\_Out[31:0]),

.I2({PC\_Current[31:28],Inst[25:0],N0,N0}),

.I3(32'h00000000),

.s(PCSource),

.o(PC\_Next)

);

MUX2T1\_32 MUX4(.I0(rdata\_A), // reg out A

.I1(PC\_Current), // PC

.s(ALUSrcA),

.o(Alu\_A)

);

MUX2T1\_32 MUX5(.I0(PC\_Current), //IF

.I1(ALU\_Out), //access memory

.s(IorD),

.o(M\_addr)

);

endmodule