浙江大学

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　6 月 18 日

Final Report– Multi-Cycle CPU Implementation

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: 2020-06-18 **Instructor:** 洪奇军

1. Experiment Objectives and Requirements

- Understand and implement a multi-cycle CPU design that supports at least the following instructions:
  - R-Type: add, sub, and, or, xor, nor, slt, srl, jr, jalr
  - I-Type: addi, andi, ori, xori, lui, lw, sw, beq, bne, slti
  - J-Type: j, jal
- Datapath Design
  - Mandate memory management and ALU operations
- Controller Design
  - Set to send the appropriate control signals to the datapath for each instruction
- Design testing procedures

2. Content and Principles of the Experiment

- **CPU Organization**

  The central processing unit (CPU), consist of two main components – the control unit and datapath. As depicted, the datapath follows the program instructions and performs arithmetic operations to get the result. The controller tells the datapath what to do and what components to use, by asserting different control signals.

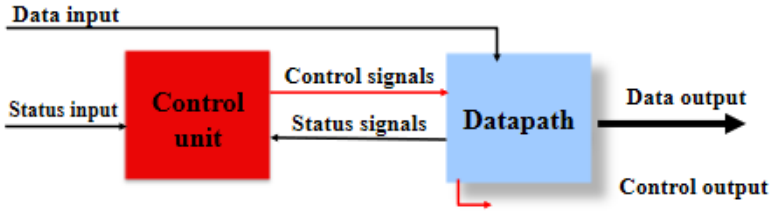


Figure 1 - CPU Organization

- **MIPS Instructions**

  A MIPS instruction is broken up into different fields, specifying the registers and operations used for when it is processed. R-type, I-type and J-type do not share the same format. An instruction consists of 32-bits and contains all the information needed to be processed in the CPU.

Note the different destination registers, where R-type would use the rd field and I-type would use the rt field.

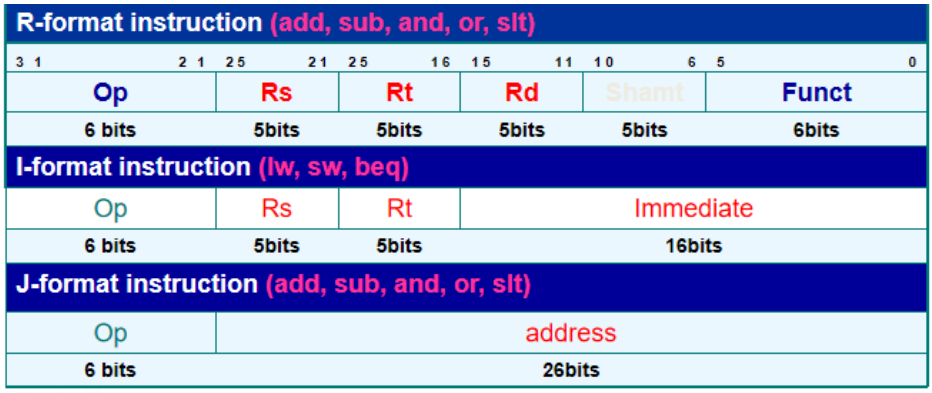


Figure 2 - MIPS Instruction Field

- **Differences Between Multi-Cycle and Single-Cycle**

  The multi-cycle CPU is more efficient, because it can work with smaller and multiple clock cycles for each instruction, while the single-cycle implementation only has one set cycle length. Many instructions could finish execution in a shorter clock cycle, and this could potentially build up wasted time. Multi-cycle implementation allows units to be used multiple times, but not always simultaneously in the same clock cycle. Here, we assume that an instruction takes up multiple clock cycles, and each step will take one clock cycle to completion.

  Hardware is significantly reduced in the multi-cycle implementation, most noticeably the lack of adders and the memory units. Functional units are now "shared", but multiplexers are widened to support routing the proper data to other parts of the CPU. Additional registers are added to every major unit, and data is held until used in a subsequent clock cycle.

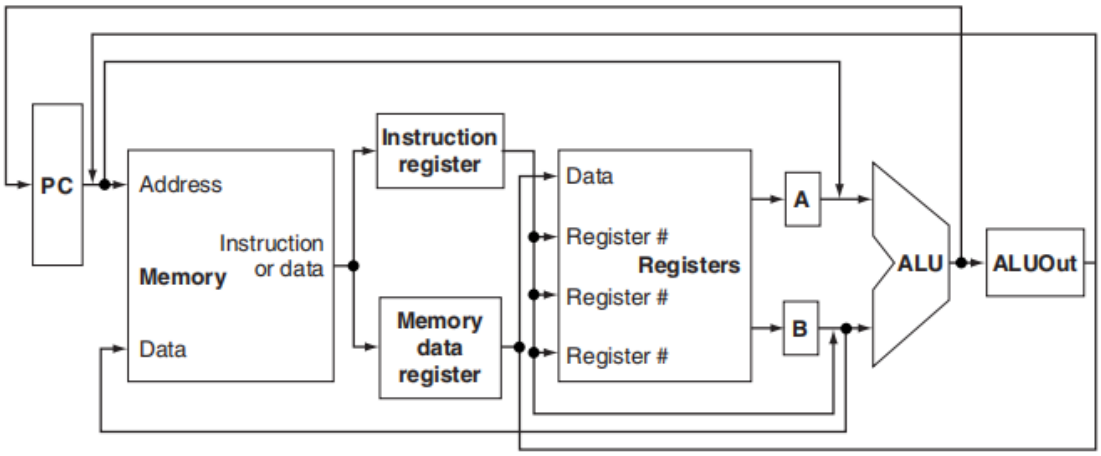


Figure 3 - Top-level of MCPU

- **Instruction and Data Memory**

  Unlike the single-cycle CPU, the instruction memory, and data memory units here are merged into one single RAM component. Instructions are still fetched from this unit, and data memory can be accessed and written when necessary signals are asserted. This is solely referred as the memory unit of the multi-cycle CPU.

- **The Datapath**

  The datapath implemented in this course has the following components – instruction and data memory, instruction register, memory data register, register file, ALU, program counter, shifters and a 32-bit sign extender.

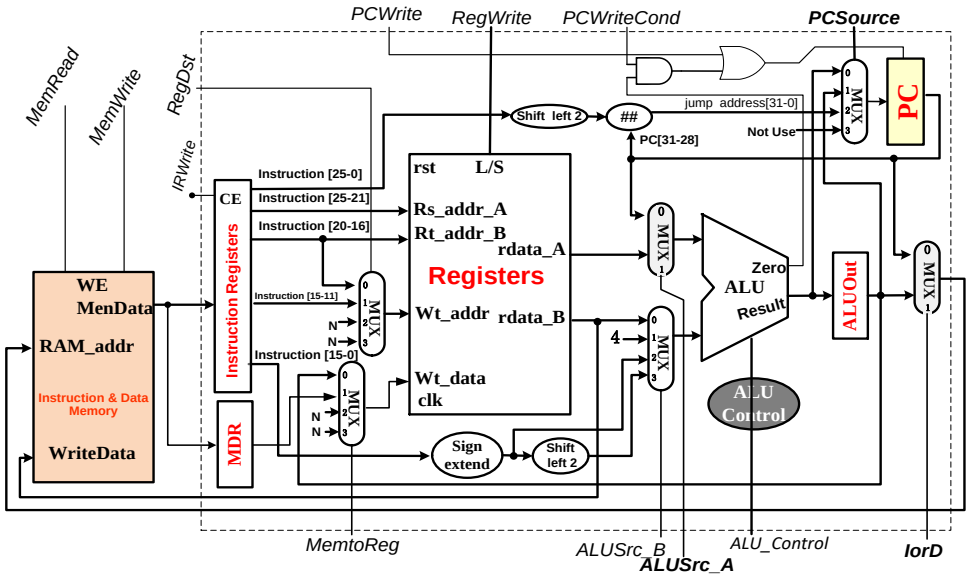


Figure 4 - Top Diagram of Lab MCPU

  - **Instruction Register (IR)** - The IR is one of the additional temporary registers that are implemented in a multi-cycle CPU. Its purpose is to hold the instruction until completion. It receives the 32-bit instruction as input from the memory unit, and outputs each field needed by the other components.

  - **Memory Data Register (MDR)** - The MDR is one of the additional temporary registers that are implemented in a multi-cycle CPU. It holds data only between a pair of adjacent clock cycles. This is where data from the memory unit gets held, until it gets written into the register file for R-type and memory reference instructions.

  - **ALU Input Registers (A&B)** - Registers A and B store values from the two outputs of register files. These would store the data of registers rs and rt, for a R-type instruction. They only get routed to the ALU for input if permitted by the control signals.

However, the labs do not require ALU input registers to be implemented. Instead, the contents of rdata_A and rdata_B are routed to the ALU if selected by the multiplexer.

- **ALU Output Register (ALUOut) -** ALUOut simply stores the resulting output produced by the ALU. Depending on what signals are asserted, the data in this register could get routed to memory, the register file or to the PC. It could store an immediate value, an address for the PC or a memory offset.

- **Program Counter (PC)** – This is where the address of the current instruction is stored. The address gets sent to memory, where the instruction is fetched and processed. In a R-type instruction, the PC gets updated with the next sequential instruction address while a jump or branching instruction would update the PC with another target address.

- **Arithmetic Logic Unit (ALU) -** All R-type and I-type instructions use the ALU for processing. The memory reference instructions would use it for address calculations, branching for comparisons, and executing arithmetic-logic operations. The type of arithmetic operation is done based on what signal ALU_Control provides.

  In a multi-cycle CPU, the ALU takes care of all the tasks that the auxiliary adders are assigned to do. This includes incrementing the PC, and computing the target addresses as well as offsets. Widened multiplexers are used to select the appropriate inputs for the operation.

  The overflow signal indicates whether there is an overflow and the zero signal is asserted when a branch is taken by a branching instructed. Lastly, the 32-bit ALU_output contains the address to be written into the data memory or the data to be written into a destination register.

  The ALU supports 8 different operations – And, Or, Add, Sub, Slt, Nor, Srl, and Xor.

- **Register File –** Like the name suggests, this component contains a set of registers that can be read and written by supplying a register number to be accessed. There is a total of 32 32-bit registers in the CPU. The registers are implemented as an array of D flip-flops, decoders and multiplexers are used for reading and writing data. There are four main inputs; R_addr_A, R_addr_B, Wt_addr and Wt_data. The first two are the numbers registers to be read, while the third one provides the number of the destination register and the last one contains the data to be written into the destination. Outputs rdata_A and rdata_B returns the contents of the operand registers and routes them to the ALU. There are three signals used by the register; clk, rst and L_S (asserted if Wt_addr is written with the value on the Wt_data input).

  Note that inputs and outputs are 32-bits, while register numbers are 5-bits.

- **Sign Extender –** A 32-bit sign extender unit is used for branching, memory reference and I-type instructions. Addresses, offsets, and immediate values are given in 16-bit

values, so it is necessary to extend them to 32-bits for further processing. The extended output (Imm_32) gets sent to the B input of the ALU, only if selected by a multiplexer to do so.

- **Multiplexers –** Six multiplexers were used in the single-cycle CPU design, and they all serve the same purpose of selecting one of its inputs to be routed out to another unit as their input.

  MUX1: Selected the input for the write address for the register file (reg_Wt_addr). It chooses from either the rd field or rt field.

  MUX2: Selected the input for the write data port of the register file (w_reg_data). It chooses from either the MDR, or ALU_out.

  MUX3: Selected the input for the B port of the ALU. It chooses from either an immediate value of 4, the branch offset, the B output value from the register file, or jump offset.

  MUX4: Selected the input for the A port of the ALU. It chooses from either the A output value from the register file or the current PC address.

  MUX5: Selected the input for the memory unit address (M_addr). It chooses from either the current PC or the contents of ALUOut.

  MUX6: Selected the input for the PC. It chooses from either the ALU output (res[31:0]), ALUOut or the computed jump address.

- **The Controller**

The controller unit controls the flow of information with the use of several signals. It determines the components that need to be used, and which MUX signals to assert with its own controls. The ALU relies on the ALU_Control signal to determine which operation to execute for a specific instruction.

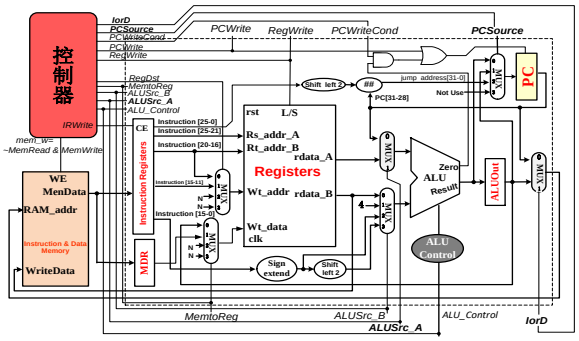


Figure 5 - Controller of Lab MCPU

Signal	Function	Asserted	Not Asserted
ALUSrc_A	Selects the A input of the ALU	Input comes from rdata_A	Input comes from the PC
ALUSrc_B[1:0]	Selects the B input of the ALU	**01** – constant 4 **10** – sign-extended IR[15:0] **11** - sign-extended IR[15:0] >> 2	**00** – Input comes from rdata_B
RegDst[1:0]	Selects the register write address	**01** – Write to rd	**00** – Write to rt
MemtoReg[1:0]	Selects source of the data to be written into the registers	**01** – Use data from MDR	**00** – Use data from ALUOut
IorD	Selects source of the address to be sent to the memory	Address comes from memory	Address comes from ALUOut
PCSource[3:0]	Selects what overwrites the PC	**01** – ALUOut (branch address) **10** – Jump target address from ALU	**00** – Default PC + 4
PCWriteCond	Used for conditional instructions	Update PC with branch address if zero == 1	Branch not taken
PCWrite	Determines if PC needs to be written to	PC needs to be written according to PCSource	None.
Branch	Used for BNE and BEQ	Branch taken	Branch not taken
RegWrite	Determines if a register needs to be written to.	Register indicated by Wt_addr is written with Wt_data	None.
MemWrite	Determines if the data memory needs to be written to.	Memory at address Ram_addr is overwritten.	None.
MemRead	Determines if the data memory needs to be read.	Memory at address Ram_addr is read.	None.
IRWrite	Determines whether if the memory output is written into the IR	Update IR with the newly fetched instruction	None.
ALU_Control[2:0]	Sets the appropriate ALU function, using ALU_OP. (Refer to chart)	N/A	N/A

Table 1 - MCPU Control Signals

- **Main Controller Truth Table**

状态	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001
输出信号	IF	ID	MEM-Ex	MEM-RD	LW_WB	MEM_W	R_Exc	R_WB	Beq_Exc	J
PCWrite	1	0	0	0	0	0	0	0	0	1
PCWriteCond	0	0	0	0	0	0	0	0	1	0
IorD	0	0	0	1	0	1	0	0	0	0
MemRead	1	0	0	1	0	0	0	0	0	0
MemWrite	0	0	0	0	0	1	0	0	0	0
IRWrite	1	0	0	0	0	0	0	0	0	0
MemtoReg	00	00	00	00	01	00	00	00	00	00
PCSource1	0	0	0	0	0	0	0	0	0	1
PCSource0	0	0	0	0	0	0	0	0	1	0
ALUSrcA	0	0	1	0	0	0	1	0	1	0
ALUSrcB1	0	1	1	0	0	0	0	0	0	0
ALUSrcB0	1	1	0	0	0	0	0	0	0	0
RegWrite	0	0	0	0	1	0	0	1	0	0
RegDst	00	00	00	00	00	00	00	01	00	00
Branch	0	0	0	0	0	0	0	0	1	0
ALUOp1	0	0	0	0	0	0	1	0	0	0
ALUOp0	0	0	0	0	0	0	0	0	1	0
MEM_IO	0	0	0	1	0	1	0	0	0	0

Table 2 - FSM Value Signals Pt.1

状态 输出信号	1010 I_Exc	1100 I_WB	1011 Lui_Exc	1101 Bne_Exc	1110 Jr	1111 Jal	10000 Jalr
PCWrite	0	0	0	0	1	1	1
PCWriteCond	0	0	0	1	0	0	0
IorD	0	0	0	0	0	0	0
MemRead	0	0	0	0	0	0	0
MemWrite	0	0	0	0	0	0	0
IRWrite	0	0	0	0	0	0	0
MemtoReg	00	00	10	00	00	11	11
PCSource1	0	0	0	0	1	1	1
PCSource0	1	0	0	1	1	0	1
ALUSrcA	1	0	1	1	0	0	0
ALUSrcB1	0	0	1	0	0	0	0
ALUSrcB0	0	0	1	0	0	0	0
RegWrite	0	1	1	0	0	1	1
RegDst	00	00	00	00	00	10	00
Branch	0	0	0	0	0	0	0
ALUOp1	1	0	0	0	0	0	0
ALUOp0	1	0	0	1	0	0	0
CPU_IO	0	0	0	0	0	0	0

Table 3 - FSM Value Signals Pt.2 Extentsion

- **Finite State Machine**

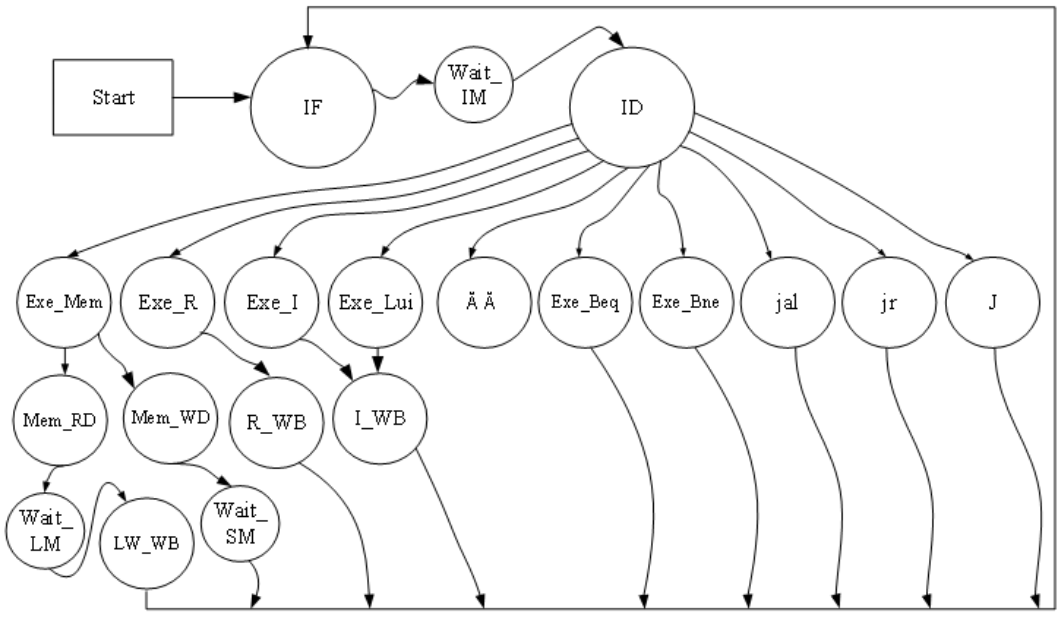
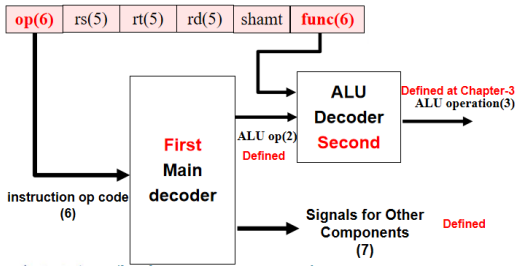


Figure 6 - Extended FSM Diagram

- **ALU Decoder**

The opcode field of an instruction are first decoded and sets the signals for the processes of other units. As for the specified ALU operation, the funct field (instruction[5:0]) is separately decoded to ALU_Control.



ALU Control Lines	Function
000	And
001	Or
010	Add
110	Sub
111	Set on less than
100	nor
101	srl
011	xor

Figure 7 - Decoder Organization *Table 4 - ALU Control Signal Values*

ALU_Control signals are broken down from ALU_OP, bnegate signal and the instruction funct field, as follows.

Table 5 - ALU Signals and Funct Fields

Opcode	ALU_OP	Instruction Operation	Funct	ALU_Control	ALU Operation
LW 100011	00	Load word	XXXXXX	010	Add
SW 101011	00	Store word	XXXXXX	010	Add
BEQ 000100	01	Branch equal	XXXXXX	110	Subtract
R-type 000000	10	Add	100000	010	Add
R-type 000000	10	Subtract	100010	110	Subtract
R-type 000000	10	And	100100	000	And
R-type 000000	10	Or	100101	001	Or
R-type 000000	10	Set on less than	101010	111	SLT
J-type 000010	XX	Jump	N/A	N/A	N/A

*The J-type instructions do not use the controller.

- **Executing Instructions of All Instruction Classes: Summary**

Step	R-type	Memory Reference	Branching	Jumps
Instruction Fetch **(IF)**	IR = Memory[PC] PC = PC + 4			
Instruction Decode **(ID)** Register Fetch	A = Reg[rs] B = Reg[rt] ALUOut = PC + (sign-extend(instruction[15:0] << 2)			
Execution Compute Address Branch/Jump Finish	ALUOut=A op B	ALUOut = A + (sign-extend(instruction[15:0])	If (A == B): PC = ALUOut	PC = address + PC[31:28] + "00"
Memory Access R-Type Completion	Reg(rd)=ALUOUT	**Load:** MDR = Memory[ALUOut] OR **Store:** Memory[ALUOut] <= B		
Memory Read Finish		**Load:** Reg[rt] = MDR		

Table 6 - MCPU Execution Summary

3. Equipment

Instruments:

1. Computer with Xilinx ISE 14.7 1 unit
2. SWORD Experimental Box 1 unit

4. Methods and Procedures

1. Construct the top-level of the single-cycle CPU with Verilog.

topMod.v

```
module topMod(
      input RSTN,
      input [3:0] BTN_y,
      input [4:0] BTN_x,
      input [15:0] SW,
      input clk_100mhz,
      output CR,
      output RDY,
      output readn,
      output seg_clk,
      output seg_sout,
      output seg_clrn,
      output SEG_PEN,
      output led_clk,
      output led_sout,
      output LED_PEN,
      output led_clrn,
      output [7:0] SEGMENT,
      output [3:0] AN,
```

```
      output [7:0] LED,
      output Buzzer
);

      wire V5, N0;

      assign V5 = 1'b1;
      assign N0 = 1'b0;
      assign Buzzer = 1'b1;

   wire Clk_CPU, mem_w, data_ram_we, IO_clk, GPIOE0, GPIOF0, counter0_out,
counter1_out, counter2_out, counter_we;
   wire[1:0] counter_set;
   wire[3:0] BTN_OK, Pulse;
   wire[4:0] Key_out, state;
   wire[7:0] point_out, LE_out, blink;
   wire[9:0] ram_addr;
   wire[15:0] SW_OK, LED_out;
   wire[31:0] inst, PC, Addr_out, Data_in, Data_out, ram_data_in, ram_data_out,
CPU2IO, Counter_out, Div, Disp_num, Ai, Bi;

      assign IO_clk = ~Clk_CPU;

      Multi_CPU U1(
        .clk(Clk_CPU),
        .reset(rst),
        .inst_out(inst),
        .INT(counter0_out),
        .PC_out(PC),
        .mem_w(mem_w),
        .Addr_out(Addr_out),
        .Data_in(Data_in),
        .Data_out(Data_out),
        .state(state),
        .CPU_MIO(),
        .MIO_ready(V5)
    );

    RAM_B U3(
        .addra(ram_addr),
        .wea(data_ram_we),
        .dina(ram_data_in),
        .clka(clk_100mhz),
        .douta(ram_data_out)
    );

    MIO_BUS U4(
        .clk(clk_100mhz),
        .rst(rst),
        .BTN(BTN_OK),
        .SW(SW_OK),
        .mem_w(mem_w),
        .Cpu_data2bus(Data_out),
        .addr_bus(Addr_out),
        .ram_data_out(ram_data_out),
        .led_out(LED_out),
        .counter_out(Counter_out),
        .counter0_out(counter0_out),
```

```
        .counter1_out(counter1_out),
        .counter2_out(counter2_out),
        .Cpu_data4bus(Data_in),
        .ram_data_in(ram_data_in),
        .ram_addr(ram_addr),
        .data_ram_we(data_ram_we),
        .GPIOf0000000_we(GPIOF0),
        .GPIOe0000000_we(GPIOE0),
        .counter_we(counter_we),
        .Peripheral_in(CPU2IO)
    );

    Multi_8CH32 U5(
        .clk(IO_clk),
        .rst(rst),
        .EN(GPIOE0),
        .Test(SW_OK[7:5]),
        .point_in({Div, Div[31:13], state, N0, N0, N0, N0, N0, N0, N0, N0}),
        .LES(64'b0),
        .Data0(CPU2IO),
        .data1({N0,N0,PC[31:2]}),
        .data2(inst),
        .data3(Counter_out),
        .data4(Addr_out),
        .data5(Data_out),
        .data6(Data_in),
        .data7(PC),
        .point_out(point_out),
        .LE_out(LE_out),
        .Disp_num(Disp_num)
    );

    SSeg7_Dev U6(
        .clk(clk_100mhz),
        .rst(rst),
        .Start(Div[20]),
        .SW0(SW_OK[0]),
        .flash(Div[25]),
        .Hexs(Disp_num),
        .point(point_out),
        .LES(LE_out),
        .seg_clk(seg_clk),
        .seg_sout(seg_sout),
        .SEG_PEN(SEG_PEN),
        .seg_clrn(seg_clrn)
    );

    SPIO U7(
        .clk(IO_clk),
        .rst(rst),
        .Start(Div[20]),
        .EN(GPIOF0),
        .GPIOf0(),
        .P_Data(CPU2IO),
        .counter_set(counter_set),
        .LED_out(LED_out),
        .led_clk(led_clk),
        .led_sout(led_sout),
```

```
        .led_clrn(led_clrn),
        .LED_PEN(LED_PEN)
    );

    clk_div U8(
        .clk(clk_100mhz),
        .rst(rst),
        .SW2(SW_OK[2]),
        .clkdiv(Div),
        .Clk_CPU(Clk_CPU)
    );

    SAnti_jitter U9(
        .clk(clk_100mhz),
        .RSTN(RSTN),
        .readn(readn),
        .Key_y(BTN_y),
        .Key_x(BTN_x),
        .SW(SW),
        .Key_out(Key_out),
        .Key_ready(RDY),
        .pulse_out(Pulse),
        .BTN_OK(BTN_OK),
        .SW_OK(SW_OK),
        .CR(CR),
        .rst(rst)
    );

    Counter_x U10(
        .clk(IO_clk),
        .rst(rst),
        .clk0(Div[8]),
        .clk1(Div[9]),
        .clk2(Div[10]),
        .counter_we(counter_we),
        .counter_val(CPU2IO),
        .counter_ch(counter_set),
        .counter0_OUT(counter0_out),
        .counter1_OUT(counter1_out),
        .counter2_OUT(counter2_out),
        .counter_out(Counter_out)
    );

    SEnter_2_32 M4(
        .clk(clk_100mhz),
        .BTN(BTN_OK[2:0]),
        .Ctrl({SW_OK[7:5],SW_OK[15],SW_OK[0]}),
        .D_ready(RDY),
        .Din(Key_out),
        .readn(readn),
        .Ai(Ai),
        .Bi(Bi),
        .blink(blink)
    );

    Seg7_Dev U61(
        .Scan({SW_OK[1],Div[19:18]}),
        .SW0(SW_OK[0]),
```

```
        .flash(Div[25]),
        .Hexs(Disp_num),
        .point(point_out),
        .LES(LE_out),
        .SEGMENT(SEGMENT),
        .AN(AN)
    );

    PIO U71(
        .clk(IO_clk),
        .rst(rst),
        .EN(GPIOF0),
        .counter_set(),
        .GPIOf0(),
        .PData_in(CPU2IO),
        .LED_out(LED)
    );

endmodule
```

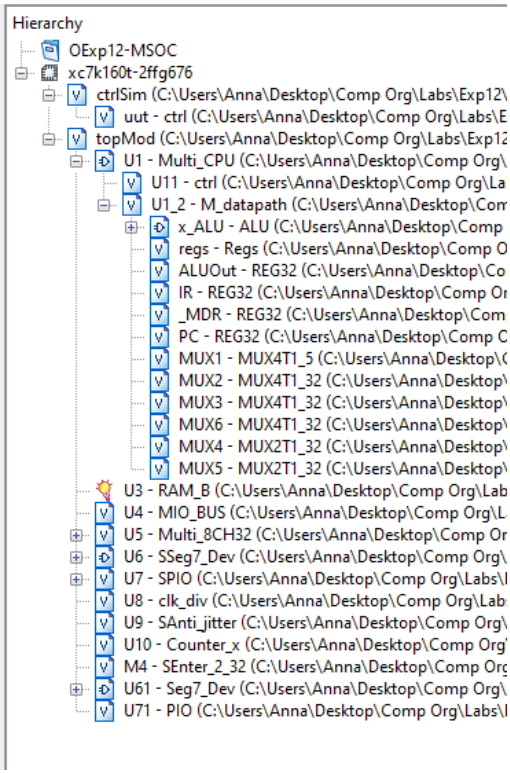


Figure 8 - MCPU file hierarchy

2. SSeg7_Dev, and Seg7_Dev are the seven-segment displays for the SWORD board. They were constructed in the earlier labs (1-4).

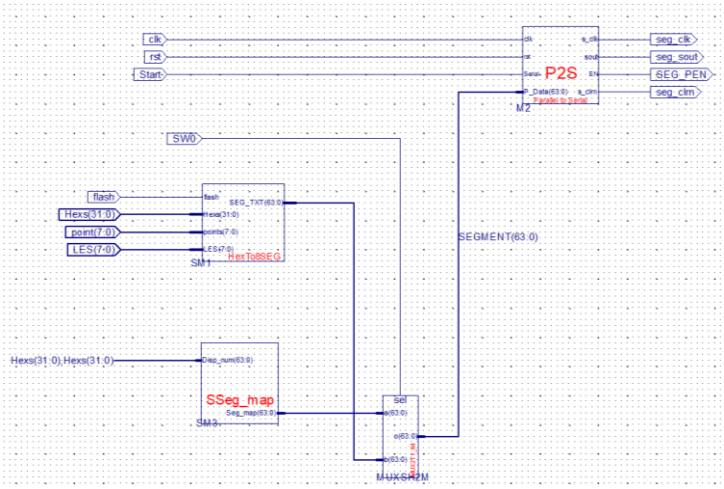


Figure 9 - SSeg7_Dev.sch

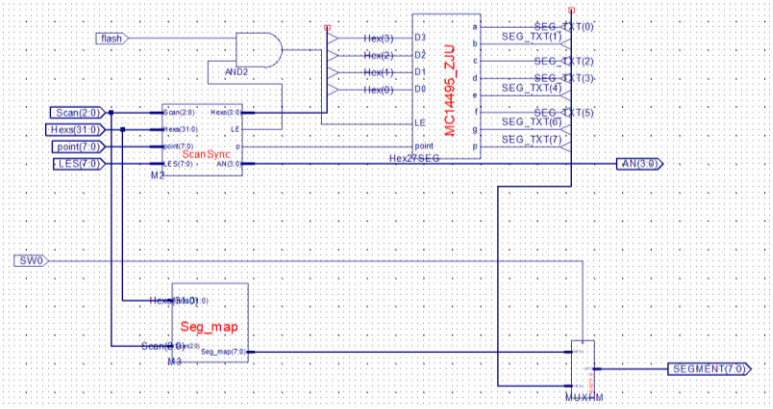


Figure 10 - Seg7_Dev.sch

3. Other than Multi_CPU, the rest of the modules can be directly added as a source and linked to the top-level.

4. Construct the top-level of the CPU with its main two components – the datapath and the controller. Link the components and set the I/Os accordingly as illustrated.

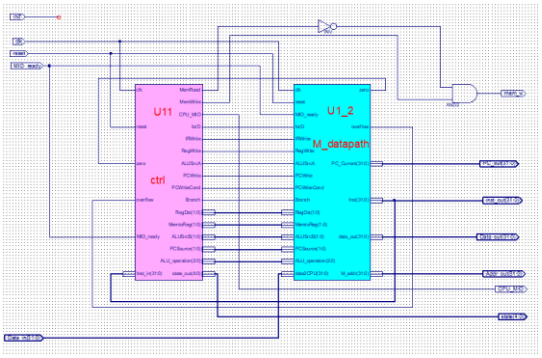


Figure 11 - Multi_CPU.sch

5. Construct the top-level of the datapath using Verilog. The provided schematic in the courseware can be used as a reference. A schematic was used in the first MCPU labs but was later implemented in HDL.

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
```

6. Use the ALU from lab 4.

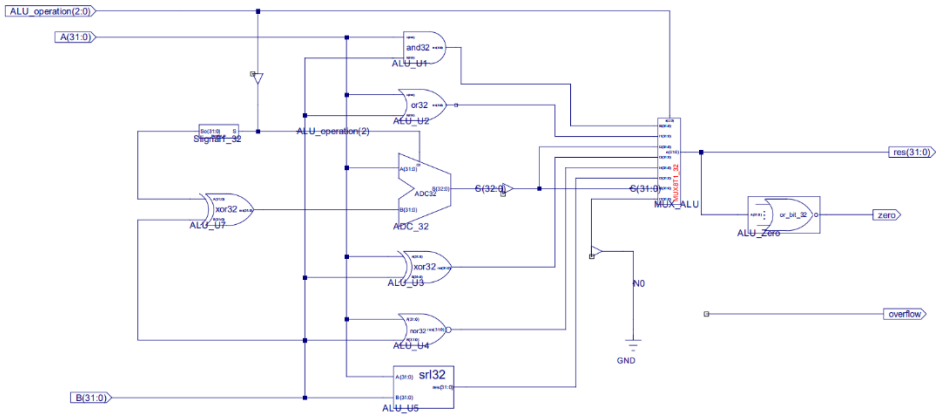


Figure 12 - ALU.sch

7. Implement the register file using Verilog. This was also taken from lab 4 and was provided in the courseware.

Regs.v

```
module  Regs(input clk, rst, L_S,
             input [4:0] R_addr_A, R_addr_B, Wt_addr,
             input [31:0] Wt_data,
             output [31:0] rdata_A,  rdata_B
```

```
            );

reg [31:0] register [1:31];             // r1 - r31
integer i;

assign rdata_A = (R_addr_A == 0) ? 0 : register[R_addr_A];         // readassign
rdata_B = (R_addr_B == 0) ? 0 : register[R_addr_B];          // read

always @(posedge clk or posedge rst)  begin
      if (rst == 1) begin
                  for (i=1; i<32; i=i+1) begin
                        register[i] <= 0;          // reset
                  end
            end else if ((Wt_addr != 0) && (L_S == 1)) begin
                  register[Wt_addr] <= Wt_data;      // write
            end
      end

endmodule
```

8. Implement the program counter (PC), ALUOut register, instruction register, and memory data register. These are just 32-bit registers, that holds a value. This was provided in the courseware.

REG32.v

```
module REG32(input clk, rst, CE, [31:0] D,
            output reg[31:0] Q
    );

      always @(posedge clk or posedge rst) begin
            if (rst==1) Q <= 32'h00000000;
            else if (CE) Q <= D;
      end


endmodule
```

9. Implement the six multiplexers. For the MCPU, we will need one 8-bit 4-1 MUX, two 32-bit 2-1 MUX, and three 32-bit 4-1 MUX. These select the input for the unit, and more detail about their implementation can be found in section 2.

10. Implement the 32-bit signal extender. This was also taken from lab 4 and was provided in the courseware.

Ext_32.v

```
module Ext_32(input [15:0] imm_16,
            output[31:0] Imm_32
            );

      assign Imm_32 = {{16{imm_16[15]}},imm_16};

endmodule
```

11. Implement the controller using Verilog. Construction started from lab 11 and had extensions added in for lab 12. This was done using the finite state machine as reference.

mulit_ctrl_IO.v

```
module ctrl(input  clk,
                          input  reset,
                          input  [31:0] Inst_in,
                          input  zero,
                          input  overflow,
                          input  MIO_ready,
                          output reg MemRead,
                          output reg MemWrite,
                          output reg[2:0]ALU_operation,
                          output [4:0]state_out,

                          output reg CPU_MIO,
                          output reg IorD,
                          output reg IRWrite,
                          output reg [1:0]RegDst,
                          output reg RegWrite,
                          output reg [1:0]MemtoReg,
                          output reg ALUSrcA,
                          output reg [1:0]ALUSrcB,
                          output reg [1:0]PCSource,
                          output reg PCWrite,
                          output reg PCWriteCond,
                          output reg Branch
                          );

wire Rtype, LS, IBeq, Jump, Load, Store;
wire[5:0] OP = Inst_in[31:26];
reg[3:0] state;
reg[1:0] ALUop;

parameter IF = 4'b0000,  ID = 4'b0001, Mem_Ex = 4'b0010, Mem_RD = 4'b0011,
       LW_WB = 4'b0100,  Mem_W = 4'b0101,  R_Exc = 4'b0110, R_WB = 4'b0111,
       Beq_Exc = 4'b1000, J = 4'b1001, I_Exc = 5'b01010, I_WB = 5'b01011,
       Lui_Exc = 5'b01100, Bne_Exc = 5'b01101, Jr = 5'b01110, Jal = 5'b01111,
       Jalr = 5'b10000,    Error = 4'b1111;

`define Datapath_signals {PCWrite, PCWriteCond,IorD, MemRead, MemWrite,IRWrite,
MemtoReg, PCSource, ALUSrcA, ALUSrcB, RegWrite, RegDst, Branch, ALUop, CPU_MIO}

parameter  value0 = 20'b10010100000010000000,
       value1 = 20'b00000000000110000000,
       value2 = 20'b00000000001100000000,
       value3 = 20'b00110000000000000001,
       value4 = 20'b00000001000001000000,
       value5 = 20'b00101000000000000001,
       value6 = 20'b00000000001000000100,
       value7 = 20'b00000000000001010000,
       value8 = 20'b01000000011000001010,
       value9 = 20'b10000000100000000000,
       value10 = 20'b00000000001100000110,
       value11 = 20'b00000000000001000000,
       value12 = 20'b00000010001111000000,
       value13 = 20'b01000000011000000010,
      value14 = 20'b10000000110000000000,
       value15 = 20'b10000011100001100000,
       value16 = 20'b10000011110001000000,
       value17 = 20'b10000011100001100000;

parameter AND=3'b000, OR=3'b001, ADD=3'b010, SUB=3'b110, NOR=3'b100,
SLT=3'b111, XOR=3'b011, SRL=3'b101;
```

```
always @ (posedge clk or posedge reset)
      if (reset==1) state <= IF;
      else
            case(state)
                  IF: if(MIO_ready) state <= ID;
                         else state <= IF;
                  ID: case (Inst_in[31:26])
                              6'b000000:
           begin
               case(Inst_in[5:0])
                   6'b001000:  state <= Jr;       //Jr
                   6'b001001:  state <= Jalr;     //Jalr
                   default:    state <= R_Exc;          //R-type OP
               endcase
           end
           6'b100011: state <= Mem_Ex;        //Lw
           6'b101011: state <= Mem_Ex;        //Sw
           6'b001000: state <= I_Exc;         //Addi
           6'b001100: state <= I_Exc;         //Andi
           6'b001101: state <= I_Exc;         //Ori
           6'b001110: state <= I_Exc;         //Xori
           6'b001010: state <= I_Exc;         //Slti
           6'b001111: state <= Lui_Exc;       //Lui
           6'b000100: state <= Beq_Exc;       //Beq
           6'b000101: state <= Bne_Exc;       //Bne
           6'b000010: state <= J;           //Jump
           6'b000011: state <= Jal;         //Jal
           default:   state <= Error;
                       endcase
                  Mem_Ex: if(Inst_in[29]) state <= Mem_W;
                                 else state <= Mem_RD;
                  Mem_RD: state <= LW_WB;
                  LW_WB: state <= IF;
                  Mem_W: state <= IF;
                  R_Exc: state <= R_WB;
                  R_WB: state <= IF;
                  I_Exc: state <= I_WB;
                  I_WB:  state <= IF;
                  Lui_Exc: state <= IF;
                  Beq_Exc: state <= IF;
                  Bne_Exc:     state <= IF;
                  Jal: state <= IF;
                  Jr: state <= IF;
                  J: state <= IF;
                  Error: state <= Error;
                  default: state <= Error;
            endcase

always @ * begin
    case(state)                          //state
        IF:     `Datapath_signals = value0;
        ID:         `Datapath_signals = value1;
        Mem_Ex:   `Datapath_signals = value2;
        Mem_RD:     `Datapath_signals = value3;
        LW_WB:         `Datapath_signals = value4;
        Mem_W:         `Datapath_signals = value5;
        R_Exc:         `Datapath_signals = value6;
        R_WB:    `Datapath_signals = value7;
        Beq_Exc:  `Datapath_signals = value8;
        J:          `Datapath_signals = value9;
        I_Exc:      `Datapath_signals = value10;
```

```
        I_WB:       `Datapath_signals = value11;
        Lui_Exc:  `Datapath_signals = value12;
        Bne_Exc:  `Datapath_signals = value13;
        Jr:         `Datapath_signals = value14;
        Jal:     `Datapath_signals = value15;
        Jalr:    `Datapath_signals = value16;
        default:  `Datapath_signals = value0;
    endcase
end

always @ * begin
    case(ALUop)
        2'b00: ALU_operation = 3'b010;  //add????
        2'b01: ALU_operation = 3'b110;  //sub????
        2'b10:
        case (Inst_in[5:0])
            6'b100000: ALU_operation = ADD;
            6'b100010: ALU_operation = SUB;
            6'b100100: ALU_operation = AND;
            6'b100101: ALU_operation = OR;
            6'b100111: ALU_operation = NOR;
            6'b101010: ALU_operation = SLT;
            6'b000010: ALU_operation = SRL;       //shfit 1bit right
            6'b000000: ALU_operation = XOR;
            default:   ALU_operation = ADD;
        endcase
        2'b11:
               case (Inst_in[31:26])
                   6'b001010: ALU_operation = SLT;   //slti
            6'b001000: ALU_operation = ADD;    //addi
            6'b001100: ALU_operation = AND;    //andi
            6'b001101: ALU_operation = OR;     //ori
            6'b001110: ALU_operation = XOR;    //xori
            default:   ALU_operation = ADD;
        endcase
    endcase
end

endmodule
```

12. Implement the memory unit (RAM_B) by generating an IP Core. Load the .coe file provided by the courseware.

13. Attach the provided .ucf file to the top module.

14. Simulate the register file, ALU, datapath, controller, and seven-segment displays.

15. Generate the programmable file (.bit) by synthesizing and implementing the top module design.

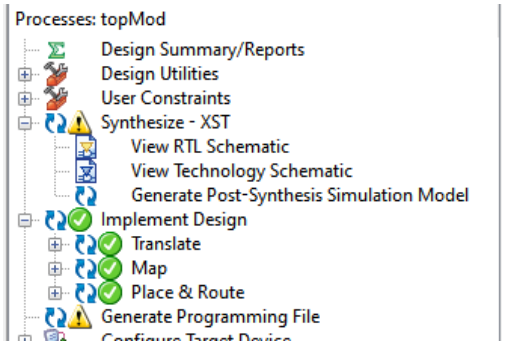


Figure 13 - Successfully generated programming file

Figure 14 - .bit file generated

16. Implement .bit file onto the SWORD board and observe results.

5. Experimental Results and Data Analysis

Due to the given circumstances of this semester, we were unable to verify the function of these labs and implement it onto the SWORD board. Observations and photos will be omitted. A total of five components in the multi-cycle CPU were simulated – ALU, datapath, controller, register file and 7-segment display. Note that the same ALU and register file simulations were used in the SCPU labs.

- Datapath

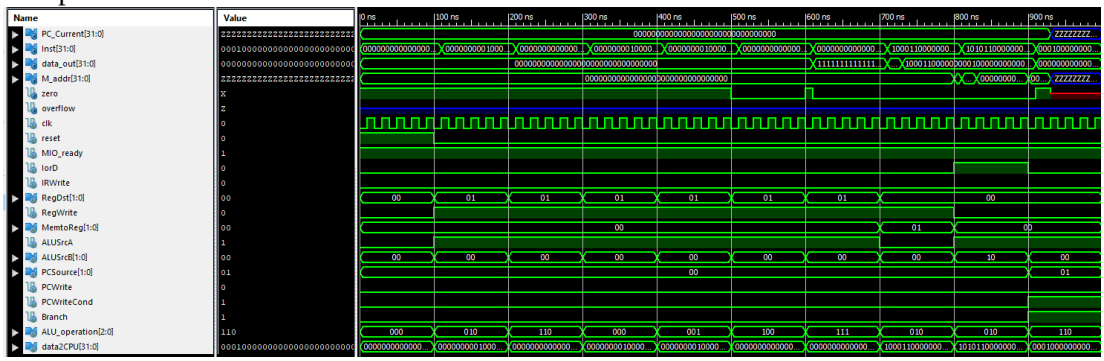


Figure 15 - MCPU datapath simulation

The datapath was simulated by providing the 32-bit instruction (data2CPU) and manually setting the different control signals (`signals) and ALU_Operation for each state represented in the finite state machine. States 0 and 1 are already preinitialized to simulate the instruction decoding, and if we were to simulate the process of a R-type instruction, we would assert the appropriate signals for states 6, 7 and back to 0. I-type, R-type, branch, and memory reference instructions were simulated. Consistent results were produced.

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
```

- Controller

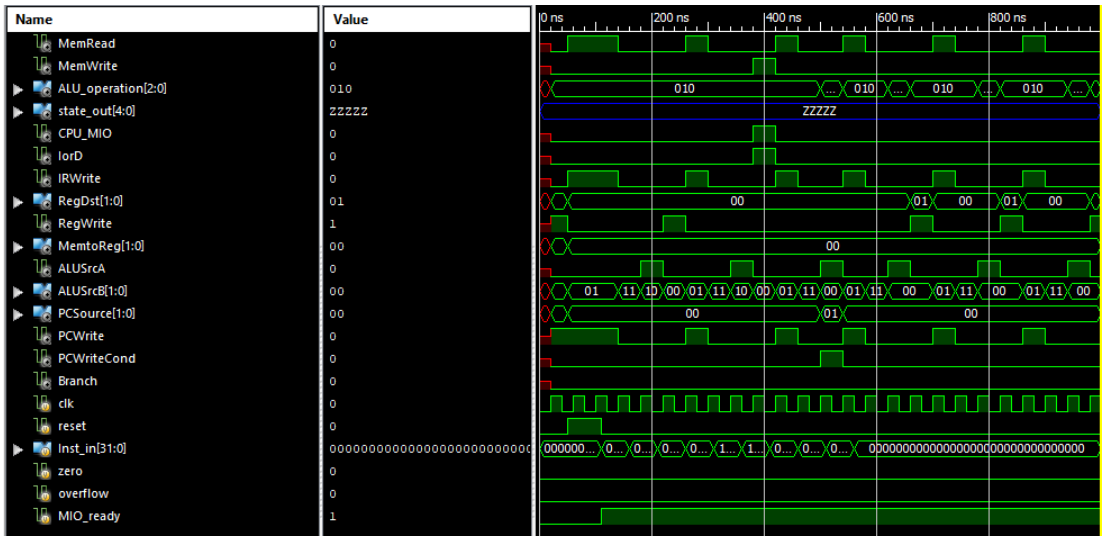


Figure 16 - MCPU controller simulation

The controller input was simulated with 32-bit instructions as input. Several R-type, I-type, branch and jump instructions were used for testing. Consistent results were produced.

ctrlSim.v

```
module ctrlSim;

	// Inputs
	reg clk;
	reg reset;
	reg [31:0] Inst_in;
	reg zero;
	reg overflow;
	reg MIO_ready;

	// Outputs
	wire MemRead;
	wire MemWrite;
	wire [2:0] ALU_operation;
	wire [4:0] state_out;
	wire CPU_MIO;
	wire IorD;
	wire IRWrite;
	wire [1:0] RegDst;
	wire RegWrite;
	wire [1:0] MemtoReg;
	wire ALUSrcA;
	wire [1:0] ALUSrcB;
	wire [1:0] PCSource;
	wire PCWrite;
	wire PCWriteCond;
	wire Branch;

	// Instantiate the Unit Under Test (UUT)
	ctrl uut (
		.clk(clk),
		.reset(reset),
```

```
		.Inst_in(Inst_in),
		.zero(zero),
		.overflow(overflow),
		.MIO_ready(MIO_ready),
		.MemRead(MemRead),
		.MemWrite(MemWrite),
		.ALU_operation(ALU_operation),
		.state_out(state_out),
		.CPU_MIO(CPU_MIO),
		.IorD(IorD),
		.IRWrite(IRWrite),
		.RegDst(RegDst),
		.RegWrite(RegWrite),
		.MemtoReg(MemtoReg),
		.ALUSrcA(ALUSrcA),
		.ALUSrcB(ALUSrcB),
		.PCSource(PCSource),
		.PCWrite(PCWrite),
		.PCWriteCond(PCWriteCond),
		.Branch(Branch)
	);

	initial begin
		// Initialize Inputs
		clk = 0;
		reset = 0;
		Inst_in = 0;
		zero = 0;
		overflow = 0;
		MIO_ready = 0;

		// Wait 100 ns for global reset to finish
		#50;
		reset=1;
		#60;
		reset=0;
		MIO_ready=1;
		Inst_in = 32'h014B4820; //add t1, t2, t3
		#50;
		Inst_in = 32'h2014003f; //addi s4, zero, 3f
		#50;
		Inst_in = 32'h11600005; //beq t3, zero, 5
		#50;
		Inst_in = 32'h0800000c; //j 12
		#50;
		Inst_in = 32'h8D69FFFF; //lw t1, 0xffff(t3)
		#50;
		Inst_in = 32'hAD71FFFF; //sw s1, 0xffff(t3)
		#50;
		Inst_in = 32'h0C00BFAF; //jal bfaf
		#50;
		Inst_in = 32'h15700005; //bne s0 5
		#50;
		Inst_in = 32'h3C0B0001; //lui t3 1
		#50;
		Inst_in = 32'h00000000;
		#50;
	end
```

```
	always begin
		clk=0;
		#20;
		clk=1;
		#20;
	end
endmodule
```

- ALU

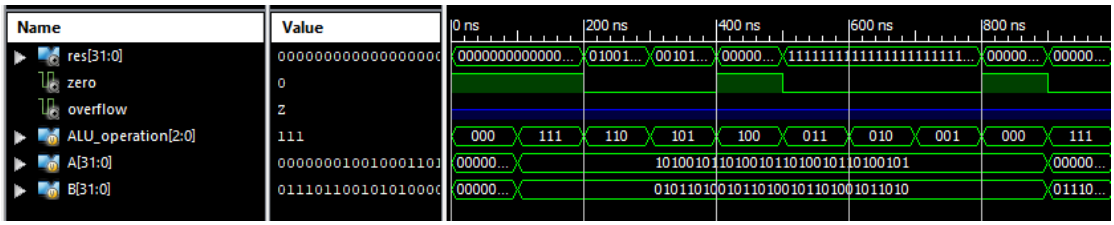


Figure 17 - MCPU ALU simulation

The ALU unit was simulated by setting two arbitrary input values for A and B, and changing the opcodes to toggle the different operations. ALU operations slt, sub, srl, nor, xor, add, or and logical and were simulated. Consistent results were produced in the res output. The Verilog module for this simulation was provided in the courseware.

aluSim.v

```
`timescale 1ns / 1ps

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; //slt
        #100;
        ALU_operation =3'b110; //sub
        #100;
        ALU_operation =3'b101; //srl
        #100;
        ALU_operation =3'b100; //nor
        #100;
        ALU_operation =3'b011; //xor
        #100;
        ALU_operation =3'b010; //add
        #100;
        ALU_operation =3'b001; //or
        #100;
        ALU_operation =3'b000; //and
        #100;
        A=32'h01234567;
        B=32'h76543210;
        ALU_operation =3'b111; //slt
end
```

- Register File

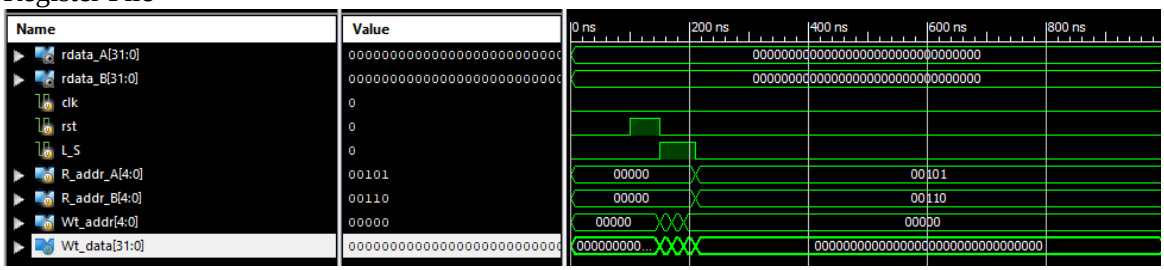


Figure 18 - MCPU Regs simulation

The register file is tested by asserting and deasserting the RegWrite signal, and providing random parameters to all four of its input ports. Either it is given two operand values, or a value and register address to write to.

regSim.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
		// Initialize Inputs
		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
```

- 7-Segment Display

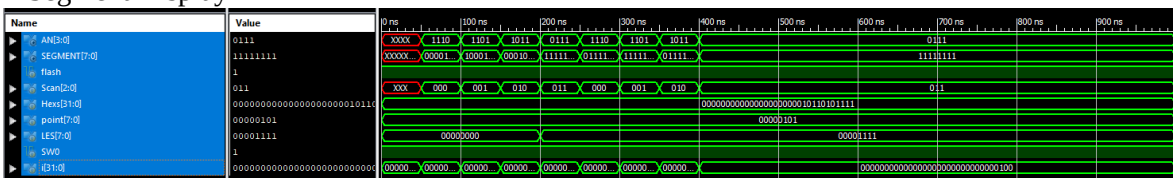


Figure 19 - Seg7Dev simulation

The 7-segment display was simulated back in lab 2. The basic task of this simulation was to traverse each segment of the display. Consistent results were produced.

Seg7Dev_Sim.v

```
`timescale 1ns / 1ps

module Seg7_Dev_Seg7_Dev_sch_tb();

// Inputs
   reg flash;
   reg [2:0] Scan;
   reg [31:0] Hexs;
   reg [7:0] point;
   reg [7:0] LES;
   reg SW0;

// Output
   wire [3:0] AN;
   wire [7:0] SEGMENT;

// Bidirs

// Instantiate the UUT
   Seg7_Dev UUT (
              .flash(flash),
              .Scan(Scan),
              .Hexs(Hexs),
              .point(point),
              .LES(LES),
              .AN(AN),
              .SEGMENT(SEGMENT),
              .SW0(SW0)
   );
// Initialize Inputs
   `ifdef auto_init
       initial begin
              flash = 0;
              Scan = 0;
              Hexs = 0;
              point = 0;
              LES = 0;
              SW0 = 0;
   `endif
```

```
      integer i;
      initial begin
            Hexs = 16'h05AF;
            point = 4'b0101;
            LES = 4'b0000;
            SW0 = 1;
            flash = 1;
            for(i = 0; i < 4; i = i + 1) begin
                  #50;
                  Scan = i;
            end
            LES = 4'b1111;
            for(i = 0; i < 4; i = i + 1) begin
                  #50;
                  Scan = i;
            end
      end
endmodule
```

6. Discussion and Conclusion

To modify the simultaneous output of control signals for the finite state machine with HDL, I needed to extend the controller unit of the MCPU processor. Each of the 16 value parameters represented a state of the FSM, and what signals (defined by Datapath_signals) were asserted. Representing the main decoder, a case-switch statement was used and assigned each state a name and its respective opcode. Following that, the ALU decoder (also represented by a case-switch statement) assigned the ALU_Operation signal controls to their respective opcode and ALU operation. Using the FSM to model the controller is how one would expand the different instructions supported. The BNE instruction has different signals than the BEQ instruction because the branch is taken in the event of an inequality. Thus, ALUop = 11 and ALU_Operation = 110. This datapath is modified to support I-type arithmetic instructions, since immediate values are handled before it gets routed to the B input port of the ALU. Secondary decoding has many advantages, such as improving performance and enabling parallel decoding. It can directly and efficiently route the information that the ALU needs to operate. By doing the decoding in a parallel matter, the input values of the ALU would not have to stall and waste clock cycles by waiting for ALU_Operation. The temporary ALU output register is needed to hold the output of a computed value, and route it to another functional unit if needed. Implementing an additional register is more cost-efficient than adding extra adders and reduces clock cycles since the value is immediately available if needed. Having the ALU output register prevents any conflicts, since the output of the ALU could be directly routed from output to the PC or stored for use in the register in the subsequent cycle.

This marks the conclusion of the multi-cycle CPU design labs of the Computer Organization course! Although I am disappointed that I am unable to first-hand experience these labs in person, I still found it to be very rewarding. Not being able to physically do SoC verification made debugging the CPU tricky, because we could not tell what worked what did not work. I am also disappointed that I am unable to run demo MIPS program using this CPU, and this makes the whole lab experience seem incomplete. I think it would have been fun to implement a project, or a game to wrap up all these labs. Doing these labs were not a complete lost though, because it greatly supplemented the material that I learned in the theoretical portion of this course. It helped me better understand how control signals

played a role in instruction execution, and how instructions were processed throughout each component. It made learning about the CPU a whole lot easier and I immensely enjoyed doing these labs. These labs have helped me gain confidence with writing testing modules and interpreting simulation results. From this, I had an easier time debugging and it was a way to keep reiterating information. I really liked how these labs were structured because it progressively built up my knowledge by implementing the top module first, then designing each of the components separately. I was able to tell how and why the MCPU was more efficient than the SCPU, by noticing the hardware changes and observing differences in the simulations. I look forward to learning more about computer architecture in the future and exploring its applications.

浙江大学

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　6 月 10 日

Final Report– Single-Cycle CPU Implementation

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: 2020-06-10 **Instructor:** 洪奇军

1. Experiment Objectives and Requirements

- Understand and implement a single-cycle CPU design that supports the following instructions:
  - R-Type: add, sub, and, or, xor, nor, slt, srl, jr, jalr
  - I-Type: addi, andi, ori, xori, lui, lw, sw, beq, bne, slti
  - J-Type: j, jal
- Datapath Design
  - Mandate memory management and ALU operations
- Controller Design
  - Set to send the appropriate control signals to the datapath for each instruction
- Design testing procedures

2. Content and Principles of the Experiment

- **CPU Organization**

  The central processing unit (CPU), consist of two main components – the control unit and datapath. As depicted, the datapath follows the program instructions and performs arithmetic operations to get the result. The controller tells the datapath what to do and what components to use, by asserting different control signals.

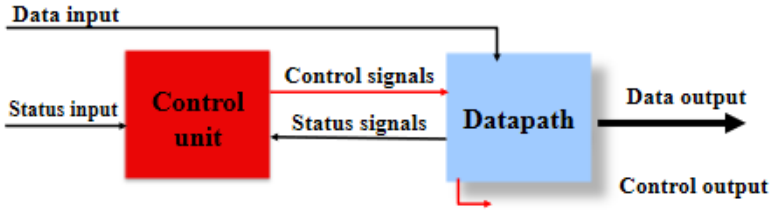


Figure 1 - CPU Organization

- **MIPS Instructions**

  A MIPS instruction is broken up into different fields, specifying the registers and operations used for when it is processed. R-type, I-type and J-type do not share the same format. An instruction consists of 32-bits and contains all the information needed to be processed in the CPU.

Note the different destination registers, where R-type would use the rd field and I-type would use the rt field.

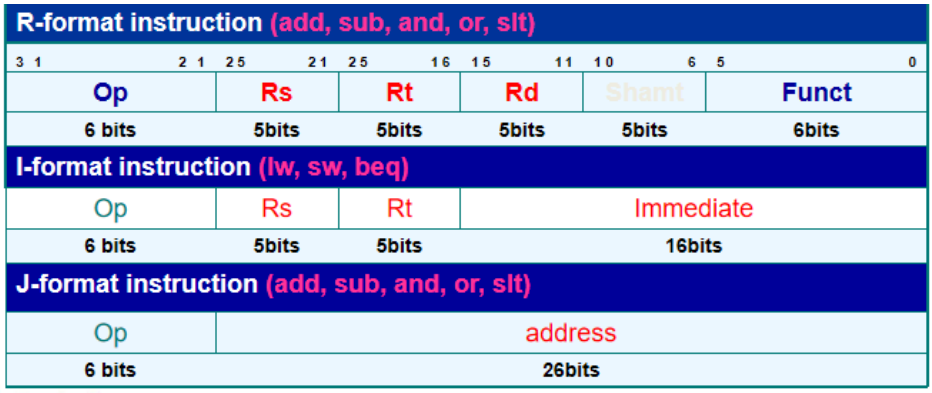


Figure 2 - MIPS Instruction Field

- **Instruction Memory**

  For these labs, the instruction memory was implemented as a ROM. Like the name suggests, this was where the instructions where stored. The program counter (PC) would send the address to this component, where the instruction would be fetched, where it would be sent to the datapath for decoding and processing. Simultaneously, the PC would send its current address to an adder and increment it by 4 to the next instruction address, getting it ready to be sent to the instruction memory once again.

- **Data Memory**

  The data memory was implemented as a RAM. Typically used in memory access instructions, this is where the datapath would retrieve data and write it to the registers. Similarly, the datapath can overwrite a memory segment with the contents of the register.

- **The Datapath**

  The datapath implemented in this course has the following components – program counter, register file, ALU, an adder for the program counter, an adder for branch instructions and a 32-bit sign extender.

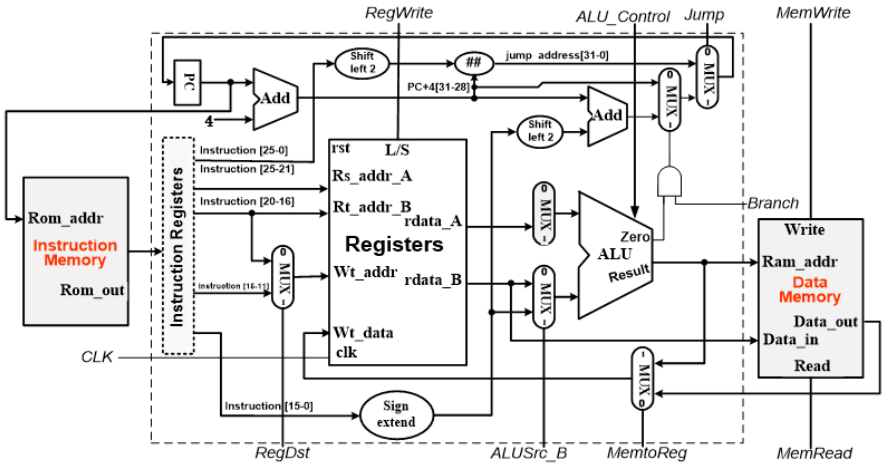


Figure 3 - Top Diagram of Lab MCPU

- **Program Counter (PC)** – This is where the address of the current instruction is stored. The address gets sent to the instruction memory, where the instruction is fetched and processed. In a R-type instruction, the PC gets updated with the next sequential instruction address while a jump or branching instruction would update the PC with another target address.

- **Arithmetic Logic Unit (ALU) –** All R-type and I-type instructions use the ALU for processing. The memory reference instructions would use it for address calculations, branching for comparisons, and executing arithmetic-logic operations. The type of arithmetic operation is done based on what signal ALU_operation provides. For lab purposes, the ALU takes in 2 32-bit inputs A and B from the register file, a 3-bit ALU_operation control signal, and 2 signal outputs zero and overflow and finally the output ALU_Out.

  The overflow signal indicates whether there is an overflow and the zero signal is asserted when a branch is taken by a branching instructed. Lastly, the 32-bit ALU_output contains the address to be written into the data memory or the data to be written into a destination register.

  The ALU supports 8 different operations – And, Or, Add, Sub, Slt, Nor, Srl, and Xor.

- **Register File –** Like the name suggests, this component contains a set of registers that can be read and written by supplying a register number to be accessed. There is a total of 32 32-bit registers in the CPU. The registers are implemented as an array of D flip-flops, decoders and multiplexers are used for reading and writing data. There are four main inputs; R_addr_A, R_addr_B, Wt_addr and Wt_data. The first two are the numbers registers to be read, while the third one provides the number of the destination register and the last one contains the data to be written into the destination. Outputs rdata_A and rdata_B returns the contents of the operand registers and routes them to the ALU. There

are three signals used by the register; clk, rst and L_S (asserted if Wt_addr is written with the value on the Wt_data input).

Note that inputs and outputs are 32-bits, while register numbers are 5-bits.

- **Sign Extender –** A 32-bit sign extender unit is used for branching, memory reference and I-type instructions. Addresses, offsets, and immediate values are given in 16-bit values, so it is necessary to extend them to 32-bits for further processing. The extended output (Imm_32) gets sent to the B input of the ALU, only if selected by a multiplexer to do so.

- **Adders –** Only two adders are used in our single-cycle CPU implementation.

  One is used to increment the contents of the PC to retrieve the next instruction. It receives the 32-bit current instruction as the input and adds 4 to it. Afterwards, it gets sent to a multiplexer and routes it to the PC.

  The other adder is used for branching and jump instructions. This adds the incremented PC and the sign-extended offset and sends the resulting address to the PC if selected by the multiplexer. For jumps, it takes the upper four bits of PC+4, adds it to the 26-bit offset from the instruction, and shifts it left by two.

- **Multiplexers –** Five multiplexers were used in the single-cycle CPU design, and they all serve the same purpose of selecting one of its inputs to be routed out to another unit as their input.

  MUXD1 & MUXD2: Selected which segment of the instruction should be interpreted as the destination register; either instruction bits 11-15 or bits 16-20. This then gets sent as the input of the register file, Wt_addr.

  MUXD3: Selected the input source for B of the ALU; either from rdata_B (register file) or the sign-extended offset.

  MUXD4: Selected the input source of Wt_data of the register file (data to be written to a register), either from data input, the last 16 bits of the instruction, contents of ALU_Out or the incremented PC.

  MUXD5: Selected the address source for the next instruction (PC); either from PC+4 adder, branch adder or jump destination.

- **The Controller**

The controller unit controls the flow of information with the use of several signals. It determines the components that need to be used, and which MUX signals to assert with its own controls. The

ALU relies on the ALU_Control signal to determine which operation to execute for a specific instruction.

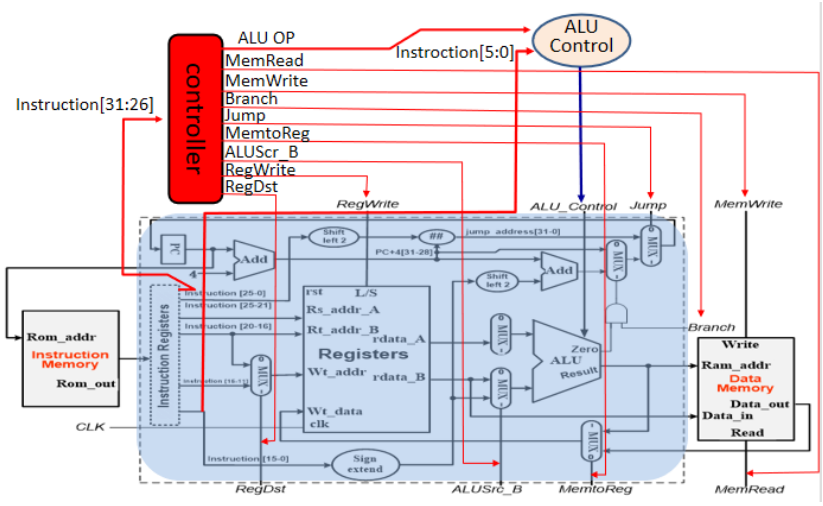


Figure 4 - Controller of Lab MCPU

Signal	Function	Asserted	Not Asserted
ALU_Src_B[1:0]	Determines what goes into the B input of the ALU by controlling MUXD3.	Select sign-extended offset (Imm_32)	Select register B (rdata_B)
RegDst[1:0]	Determines the destination register for the register file (Wt_addr) by controlling MUXD1.	Choose rd (inst_field(15:11))	Choose rt (inst_field(20:16))
MemtoReg[1:0]	Determines the source of the data to be written into the registers	Data to be written comes from the data memory	Data comes from the ALU (ALU_out)
Branch[1:0]	Determines if the branch should be taken (update the PC).	Take branch and update PC with branch address (zero == 1 as well)	PC = PC + 4
Jump[1:0]	Determines if a jump instruction is present (update the PC).	Update PC with target address.	PC = PC + 4
RegWrite	Determines if a register needs to be written to.	Register indicated by Wt_addr is written with Wt_data	None.
MemWrite	Determines if the data memory needs to be written to.	Memory at address Ram_addr is overwritten.	None.
MemRead	Determines if the data memory needs to be read.	Memory at address Ram_addr is read.	None.
ALU_Control[2:0]	Sets the appropriate ALU function, using ALU_OP. (Refer to chart)	N/A	N/A

Table 1 - CPU Control Signals

- **Main Controller Truth Table**

OP	RegDst	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	Jump	ALU Op1	ALU Op0
R	1	0	0	1	0	0	0	0	1	0
Lw	0	1	1	1	1	0	0	0	0	0
Sw	X	1	X	0	0	1	0	0	0	0
Beq	X	0	X	0	0	0	1	0	0	1
J	X	X	X	0	0	0	0	1	X	X

Table 2 - FSM Value Signals

- **ALU Decoder**

The opcode field of an instruction are first decoded and sets the signals for the processes of other units. As for the specified ALU operation, the funct field (instruction[5:0]) is separately decoded to ALU_Control.

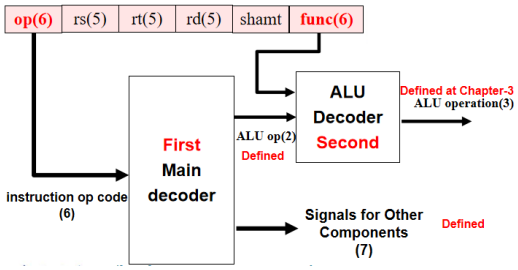


Figure 5 - Decoder Organization

ALU Control Lines	Function
000	And
001	Or
010	Add
110	Sub
111	Set on less than
100	nor
101	srl
011	xor

Table 3 - ALU Control Signal Values

ALU_Control signals are broken down from ALU_OP, bnegate signal and the instruction funct field, as follows.

Table 4 - ALU Signals and Funct Fields

Opcode	ALU_OP	Instruction Operation	Funct	ALU_Control	ALU Operation
LW 100011	00	Load word	XXXXXX	010	Add
SW 101011	00	Store word	XXXXXX	010	Add
BEQ 000100	01	Branch equal	XXXXXX	110	Subtract
R-type 000000	10	Add	100000	010	Add
R-type 000000	10	Subtract	100010	110	Subtract
R-type	10	And	100100	000	And

000000					
R-type 000000	10	Or	100101	001	Or
R-type 000000	10	Set on less than	101010	111	SLT
J-type 000010	XX	Jump	N/A	N/A	N/A

*The J-type instructions do not use the controller.

- **Datapath Operations: R-type Instructions**

OP	RegDst	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	Jump	ALU Op1	ALU Op0
R	1	0	0	1	0	0	0	0	1	0

Instruction address is first fetched from the PC, and the PC is simultaneously incremented to the next one. The address is used to retrieve the instruction from the instruction memory, where it is then decoded by the main control unit. At the same time, the instruction fields are read and registers rs and rt are accessed. The control unit tells the register file that rd will be used as the destination, with the RegDst signal. ALU detects that rs is one of its inputs, and ALUSrc tells the connecting MUX that it should take rt as the second input. The ALU control unit takes the funct field and ALUOp to determine what operation the ALU needs to perform, and after execution the result gets routed to a MUX and eventually to the input port of the register. The MemtoReg signal tells the MUX to send ALU_Out to the register file, and RegWrite signals that a register needs to be written into.

- **Datapath Operations: Memory Access Instructions**

OP	RegDst	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	Jump	ALU Op1	ALU Op0
Lw	0	1	1	1	1	0	0	0	0	0
Sw	X	1	X	0	0	1	0	0	0	0

Instruction address is first fetched from the PC, and the PC is simultaneously incremented to the next one. The address is used to retrieve the instruction from the instruction memory, where it is then decoded by the main control unit. At the same time, the instruction fields are read and register rs is accessed.

For the lw instruction, rt is set as destination register since RegDst is deasserted. ALUSrc is asserted to use the sign-extended offset as the B input for the ALU. ALUOp instructs the ALU to compute the sum of the offset and rs and sends ALU_out (memory address) to the data memory. ALU_out contains the address of the data memory, and this is used to retrieve the data and route it to the register file, with the assertion of the MemRead, MemtoReg and RegWrite signals.

No register is written to for the sw instruction, so RegDst is omitted. ALUOp instructs the ALU to compute the sum of the offset and rs and sends ALU_out out (memory address) to the data

memory. We are only writing to memory and not reading it, so only the MemWrite signal is asserted. Nothing is written to the register file, so MemtoReg is omitted.

- **Datapath Operations: Branch Instructions**

OP	RegDst	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	Jump	ALU Op1	ALU Op0
Beq	X	0	X	0	0	0	1	0	0	1

Instruction address is first fetched from the PC, and the PC is simultaneously incremented to the next one. The address is used to retrieve the instruction from the instruction memory, where it is then decoded by the main control unit. No destination register is used, so RegDst is omitted. Registers rs and rt are taken as inputs A and B for the ALU, as determined by the ALUSrc signal. The address offset goes through the 32-bit sign extender, and an adder computes the sum of the branch address. ALUOp tells the ALU to subtract A from B and asserts the zero signal if A – B = 0. This indicates that the condition is met, and coupled with the Branch signal, it tells the PC to update its value with the new branch address. Memory and registers are not written to.

- **Datapath Operations: J-Type Instructions**

OP	RegDst	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	Jump	ALU Op1	ALU Op0
J	X	X	X	0	0	0	0	1	X	X

Instruction address is first fetched from the PC, and the PC is simultaneously incremented to the next one. The address is used to retrieve the instruction from the instruction memory, where it is then decoded by the main control unit. For J-type instructions, only the opcode portion is decoded. Registers and memory are not read nor written to, so those respective signals are either deasserted or omitted. ALUOp is also omitted, since the ALU is not used. The adder computes the sum of the target address with the provided 26-bit offset from the instructions. When Jump is asserted, it overwrites the PC with the target address to go to.

3. Equipment

Instruments:

1. Computer with Xilinx ISE 14.7 1 unit
2. SWORD Experimental Box 1 unit

4. Methods and Procedures

1. Construct the top-level of the single-cycle CPU with a schematic, using the courseware provided.

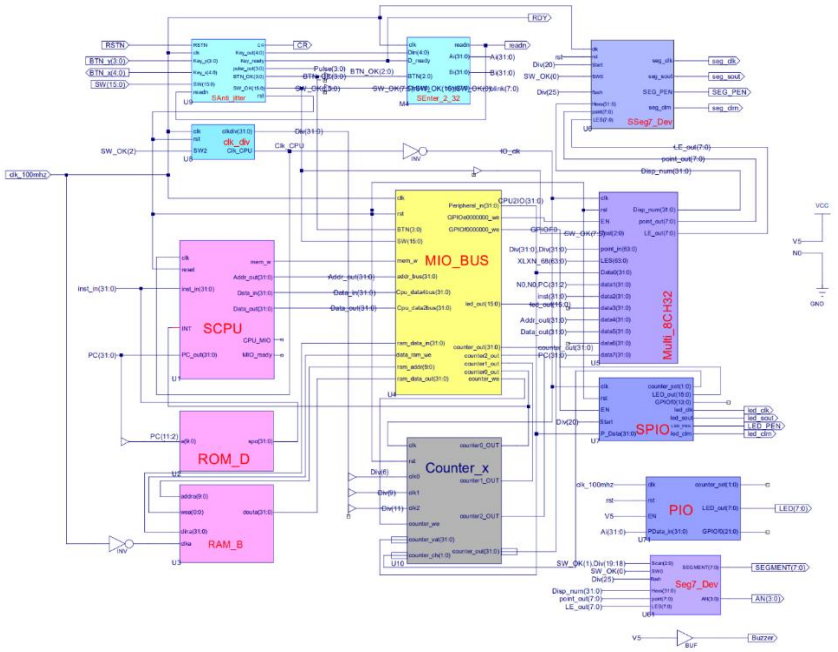


Figure 6 - topMod.sch

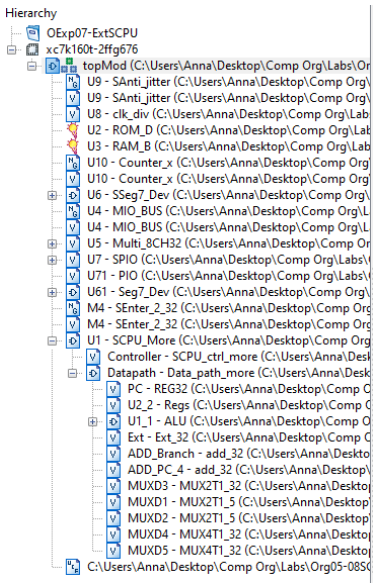


Figure 7 - SCPU file hierarchy

2. SSeg7_Dev, and Seg7_Dev are the seven-segment displays for the SWORD board. They were constructed in the earlier labs (1-4).

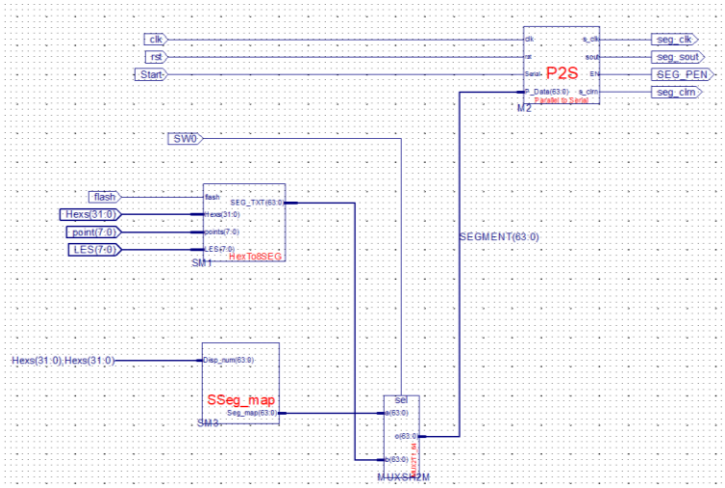


Figure 8 - SSeg7_Dev.sch

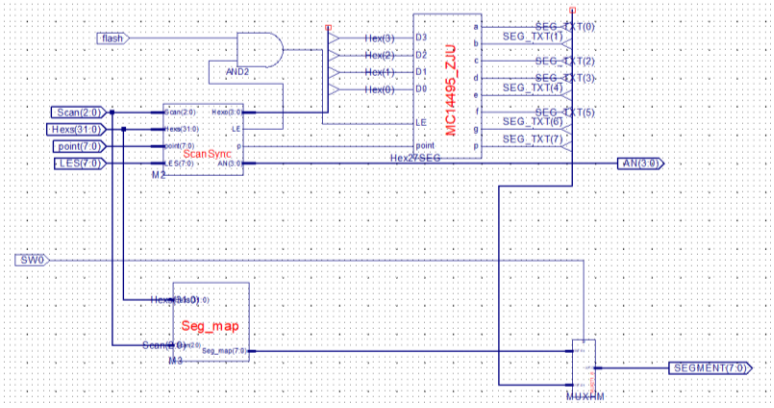


Figure 9- Seg7_Dev.sch

3. Other than SCPU_More, the rest of the modules can be directly added as a source and linked to the top-level.

4. Construct the top-level of the CPU with its main two components – the datapath and the controller. Link the components and set the I/Os accordingly as illustrated.

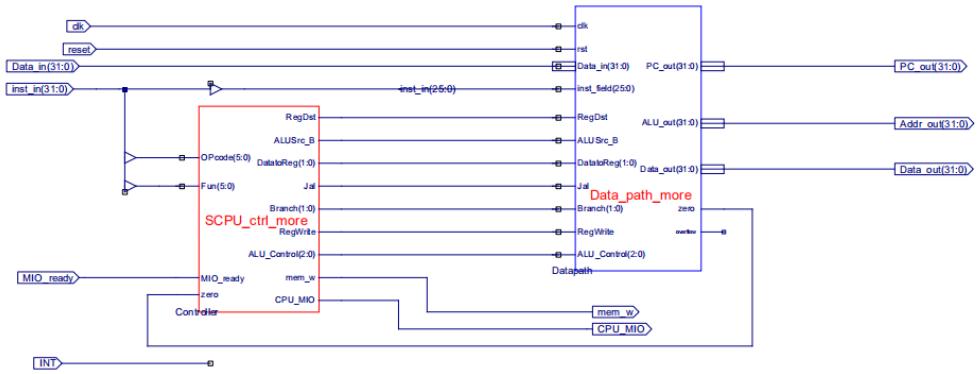


Figure 10 - SCPU.sch

5. Start by constructing the schematic for the datapath. Note the different control signals and components that were discussed in section two. Datapath design begun in lab 4 and was steadily improved to support more instructions.

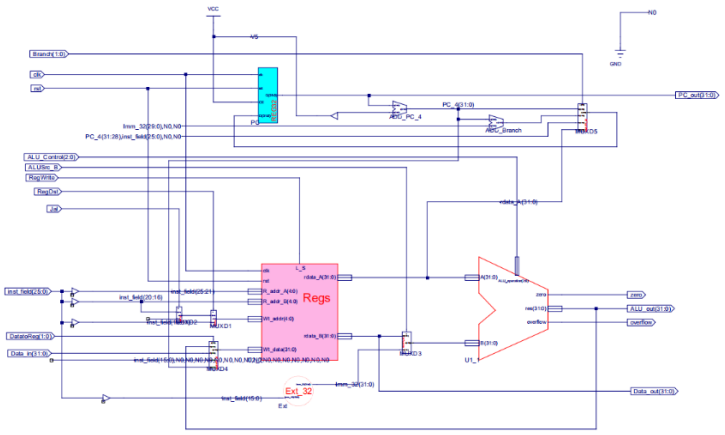


Figure 11 - Data_path_more.sch

6. Design the ALU using a schematic. This step was completed in lab 4.

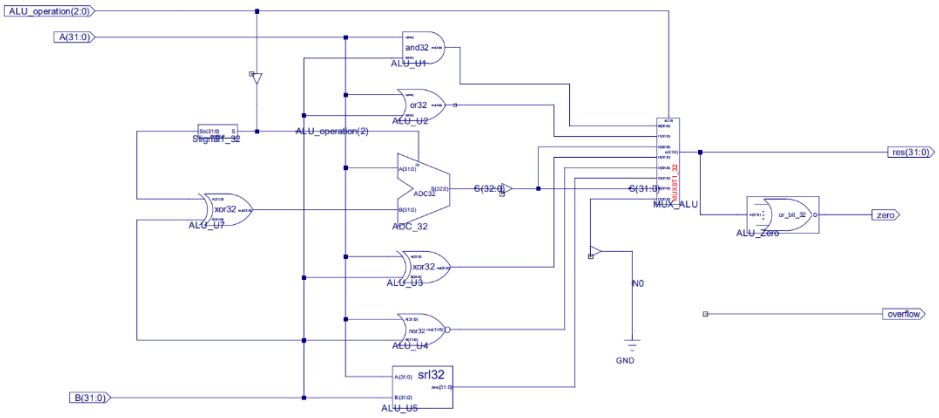


Figure 12 - ALU.sch

7. Implement the register file using Verilog. This was also taken from lab 4 and was provided in the courseware.

Regs.v

```
module  Regs(input clk, rst, L_S,
             input [4:0] R_addr_A, R_addr_B, Wt_addr,
             input [31:0] Wt_data,
             output [31:0] rdata_A,  rdata_B
             );

reg [31:0] register [1:31];              // r1 - r31
integer i;

assign rdata_A = (R_addr_A == 0) ? 0 : register[R_addr_A];         // readassign
rdata_B = (R_addr_B == 0) ? 0 : register[R_addr_B];          // read

always @(posedge clk or posedge rst)  begin
       if (rst == 1) begin
                     for (i=1; i<32; i=i+1) begin
```

```
                        register[i] <= 0;          // reset
                    end
                end else if ((Wt_addr != 0) && (L_S == 1)) begin
                    register[Wt_addr] <= Wt_data;      // write
                end
        end

endmodule
```

8. Implement the program counter (PC). This is just a 32-bit register, that holds the address value. This was also taken from lab 4 and was provided in the courseware.

REG32.v

```
module REG32(input clk, rst, CE, [31:0] D,
             output reg[31:0] Q
    );

        always @(posedge clk or posedge rst) begin
             if (rst==1) Q <= 32'h00000000;
             else if (CE) Q <= D;
        end


endmodule
```

9. Implement the adders for branch addresses and PC incrementing. These are just 32-bit adders that take in two 32-bit inputs and produce a 32-bit sum as an output. This was also taken from lab 4 and was provided in the courseware.

add_32.v

```
module add_32(input [31:0] a,
              input [31:0] b,
              output [31:0]c
              );

        assign c=a+b;

endmodule
```

10. Implement the five multiplexers. For the SCPU, we will need one 32-bit 2-1 MUX, two 5-bit 2-1 MUX, and two 32-bit 4-1 MUX. These select the input for the unit, and more detail about their implementation can be found in section 2.

11. Implement the 32-bit signal extender. This was also taken from lab 4 and was provided in the courseware.

Ext_32.v

```
module Ext_32(input [15:0] imm_16,
              output[31:0] Imm_32
              );

        assign Imm_32 = {{16{imm_16[15]}},imm_16};

endmodule
```

12. Implement the controller using Verilog. This was completed in lab 7.

SCPU_ctrl_more.v

```
module SCPU_ctrl_more(
                    input[5:0]OPcode,   //Opcode
                    input[5:0]Fun,      //Function
                    input MIO_ready,  //CPU Wait
                    input zero,
                    output reg RegDst,
                    output reg ALUSrc_B,
                    output reg [1:0]DatatoReg,
                    output reg Jal,
                    output reg [1:0]Branch,
                    output reg RegWrite,
                    output reg [2:0]ALU_Control,
                    output reg mem_w,
                    output reg CPU_MIO
    );

`define CPU_ctrl_signals
{RegDst,ALUSrc_B,DatatoReg,Jal,Branch,RegWrite,ALU_Control,mem_w,CPU_MIO}
       always @* begin
        case(OPcode)
            6'b000000: begin
               case(Fun)
                    6'b100000: `CPU_ctrl_signals = 13'b1000000101000; //add
                  6'b100010: `CPU_ctrl_signals = 13'b1000000111000; //sub
                  6'b100100: `CPU_ctrl_signals = 13'b1000000100000; //and
                  6'b100101: `CPU_ctrl_signals = 13'b1000000100100; //or
                  6'b100110: `CPU_ctrl_signals = 13'b1000000101100; //xor
                  6'b100111: `CPU_ctrl_signals = 13'b1000000110000; //nor
                  6'b101010: `CPU_ctrl_signals = 13'b1000000111100; //slt
                  6'b000010: `CPU_ctrl_signals = 13'b1100000110100; //srl
                  6'b001000: `CPU_ctrl_signals = 13'b1000011000000; //jr
                  6'b001001: `CPU_ctrl_signals = 13'b1011111100000; //jalr
               endcase
              end

       6'b100011: begin `CPU_ctrl_signals = 13'b0101000101000; end //load
       6'b101011: begin `CPU_ctrl_signals = 13'b0101000001010; end //store

       6'b000100: begin
             if (zero == 1'b1) `CPU_ctrl_signals = 13'b0000001011000; //beq
             else `CPU_ctrl_signals = 13'b0000000011000;
             end

       6'b000101: begin
             if (zero == 1'b0) `CPU_ctrl_signals = 13'b0000000011000; //bne
             else `CPU_ctrl_signals = 13'b0000001011000;
             end

       6'b000010: begin `CPU_ctrl_signals = 13'b0000010000000; end //jump

       6'b001010: begin `CPU_ctrl_signals = 13'b0100000111100; end //slti

       6'b001110: begin `CPU_ctrl_signals = 13'b0100000101100; end //xori
```

```
        6'b000011: begin `CPU_ctrl_signals = 13'b0011110100000; end //jal

        default: begin `CPU_ctrl_signals = 13'b0000000000000; end
        endcase
        end

endmodule
```

13. Implement ROM_D and RAM_B by generating an IP Core. Load the .coe file provided by the courseware.

14. Attach the provided .ucf file to the top module.

15. Simulate the register file, ALU, datapath, controller, and seven-segment displays.

16. Generate the programmable file (.bit) by synthesizing and implementing the top module design.

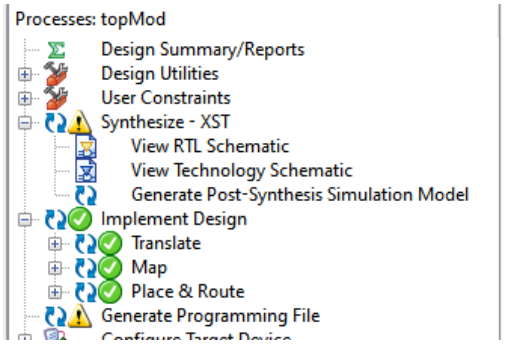


Figure 3 - Successfully generated programming file

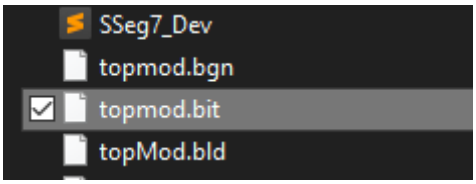


Figure 4 - .bit file generated

17. Implement .bit file onto the SWORD board and observe results.

5. Experimental Results and Data Analysis

Due to the given circumstances of this semester, we were unable to verify the function of these labs and implement it onto the SWORD board. Observations and photos will be omitted. A total of five components in the single-cycle CPU were simulated – ALU, datapath, controller, register file and 7-segment display.

- ALU

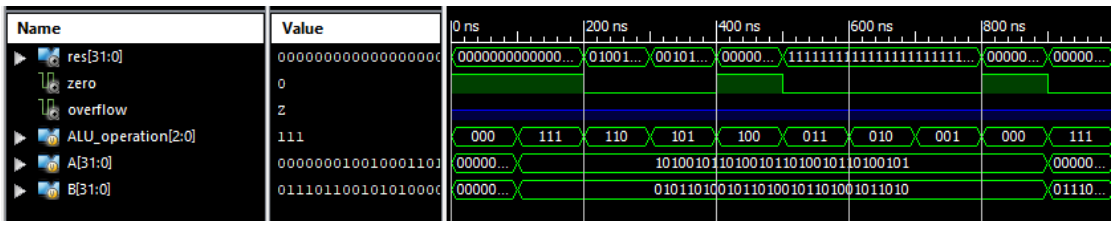


Figure 5 – ALU simulation

The ALU unit was simulated by setting two arbitrary input values for A and B, and changing the opcodes to toggle the different operations. ALU operations slt, sub, srl, nor, xor, add, or and logical and were simulated. Consistent results were produced in the res output. The Verilog module for this simulation was provided in the courseware.

aluSim.v

```
`timescale 1ns / 1ps

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; //slt
       #100;
       ALU_operation =3'b110; //sub
       #100;
       ALU_operation =3'b101; //srl
       #100;
       ALU_operation =3'b100; //nor
       #100;
       ALU_operation =3'b011; //xor
       #100;
       ALU_operation =3'b010; //add
       #100;
       ALU_operation =3'b001; //or
       #100;
```

```
        ALU_operation =3'b000; //and
        #100;
        A=32'h01234567;
        B=32'h76543210;
        ALU_operation =3'b111; //slt
end
```

- Datapath

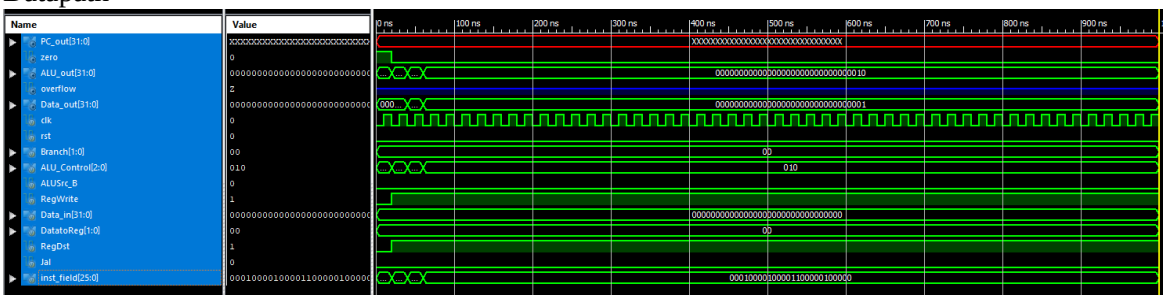


Figure 16- SCPU datapath simulation

The datapath was simulated by providing an instruction (without the opcode) for it to be processed. A simple simulation was done, where two R-type instructions (nor and slt) and beq is tested. The register fields (last 26 bits of an instruction) are provided, and the control signals are manually put into the testing module. Consistent results were produced.

datapath_moreSim.v

```
`timescale 1ns / 1ps

module Data_path_more_Data_path_more_sch_tb();

// Inputs
   reg clk;
   reg rst;
   reg [1:0] Branch;
   reg [2:0] ALU_Control;
   reg ALUSrc_B;
   reg RegWrite;
   reg [31:0] Data_in;
   reg [1:0] DatatoReg;
   reg RegDst;
   reg Jal;
   reg [25:0] inst_field;

// Output
   wire [31:0] PC_out;
   wire zero;
   wire [31:0] ALU_out;
   wire overflow;
   wire [31:0] Data_out;

// Bidirs

// Instantiate the UUT
   Data_path_more UUT (
              .PC_out(PC_out),
              .clk(clk),
```

```
            .rst(rst),
            .Branch(Branch),
            .ALU_Control(ALU_Control),
            .zero(zero),
            .ALU_out(ALU_out),
            .overflow(overflow),
            .Data_out(Data_out),
            .ALUSrc_B(ALUSrc_B),
            .RegWrite(RegWrite),
            .Data_in(Data_in),
            .DatatoReg(DatatoReg),
            .RegDst(RegDst),
            .Jal(Jal),
            .inst_field(inst_field)
   );
// Initialize Inputs
      initial begin
            clk = 0;
            rst = 0;
            Branch = 0;
            ALU_Control = 0;
            ALUSrc_B = 0;
            RegWrite = 0;
            Data_in = 0;
            DatatoReg = 0;
            RegDst = 0;
            Jal = 0;
            inst_field = 0;

            #20

            rst = 0;

            ALU_Control = 3'b100;
            RegWrite = 1;
            RegDst = 1;
            inst_field = 26'b00000_00000_00001_00000_100111;
            #20;

            ALU_Control = 3'b111;
            inst_field = 26'b00000_00001_00010_00000_101010;
            #20;

            Branch = 0;
            ALU_Control = 3'b010;
            ALUSrc_B = 0;
            RegWrite = 1;
            RegDst = 1;
            inst_field = 26'b00010_00010_00011_00000_100000;
            #20;

            end

            always begin
                  clk=0;#10;
                  clk=1;#10;
            end
```

```
endmodule
```

- Controller

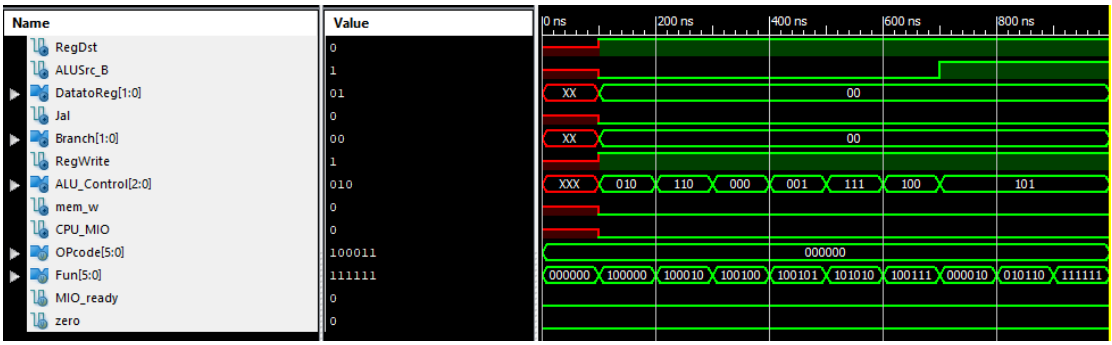


Figure 17 - SCPU controller simulation

The Fun input is the funct field of a R-type instruction, and it aids in determining the ALU operation to be used. The OPcode input is as the name suggests. All the supported ALU operations are simulated. Consistent results were produced. The Verilog module for this simulation was provided in the courseware.

SCPU_ctrl_moreSim.v

```
module SCPU_ctrl_moreSim;

	// Inputs
	reg [5:0] OPcode;
	reg [5:0] Fun;
	reg MIO_ready;
	reg zero;

	// Outputs
	wire RegDst;
	wire ALUSrc_B;
	wire [1:0] DatatoReg;
	wire Jal;
	wire [1:0] Branch;
	wire RegWrite;
	wire [2:0] ALU_Control;
	wire mem_w;
	wire CPU_MIO;

	// Instantiate the Unit Under Test (UUT)
	SCPU_ctrl_more uut (
		.OPcode(OPcode),
		.Fun(Fun),
		.MIO_ready(MIO_ready),
		.zero(zero),
		.RegDst(RegDst),
		.ALUSrc_B(ALUSrc_B),
		.DatatoReg(DatatoReg),
		.Jal(Jal),
		.Branch(Branch),
		.RegWrite(RegWrite),
		.ALU_Control(ALU_Control),
		.mem_w(mem_w),
```

```
            .CPU_MIO(CPU_MIO)
      );

      initial begin
            // Initialize Inputs
            OPcode = 0;
            Fun = 0;
            MIO_ready = 0;
            zero = 0;

            // Wait 100 ns for global reset to finish
            #100;

            // Add stimulus here
            OPcode = 6'b000000; //ALU??,?? ALUop=2'b10; RegDst=1; RegWrite=1
            Fun = 6'b100000;    //add,??ALU_Control=3'b010
            #100;
            Fun = 6'b100010;    //sub,??ALU_Control=3'b110
            #100;
            Fun = 6'b100100;    //and,??ALU_Control=3'b000
            #100;
            Fun = 6'b100101;    //or,??ALU_Control=3'b001
            #100;
            Fun = 6'b101010;    //slt,??ALU_Control=3'b111
            #100;
            Fun = 6'b100111;    //nor,??ALU_Control=3'b100
            #100;
            Fun = 6'b000010;    //srl,??ALU_Control=3'b101
            #100;
            Fun = 6'b010110;    //xor,??ALU_Control=3'b011
            #100;
            Fun = 6'b111111;    //??
            #100;
            OPcode = 6'b100011;//load??,?? ALUop=2'b00, RegDst=0,
            #100;           // ALUSrc_B=1, MemtoReg=1, RegWrite=1
            OPcode = 6'b101011;
            #100; //store??,??ALUop=2'b00, mem_w=1, ALUSrc_B=1
            OPcode = 6'b000100;//beq??,?? ALUop=2'b01, Branch=1
            #100;
            OPcode = 6'b000010;//jump??,?? Jump=1
            #100;

            OPcode = 6'h3f;           //??
            Fun = 6'b000000;          //??
      end

endmodule
```

- Register File

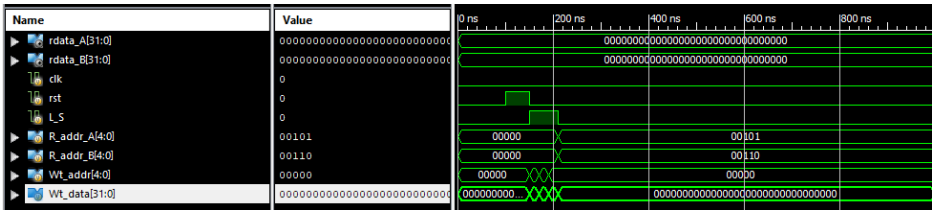


Figure 6 - SCPU Regs simulation

The register file is tested by asserting and deasserting the RegWrite signal, and providing random parameters to all four of its input ports. Either it is given two operand values, or a value and register address to write to.

regSim.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
		// Initialize Inputs
		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
```

- 7-Segment Display

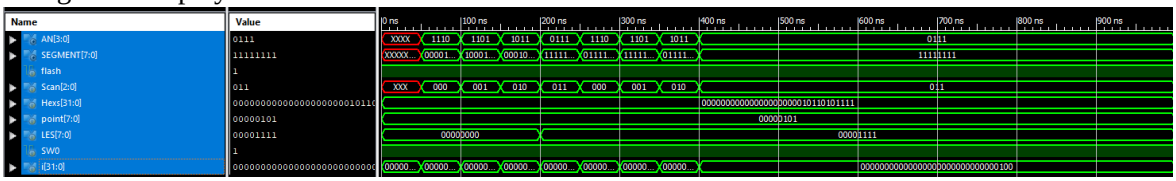


Figure 7 - Seg7Dev simulation

The 7-segment display was simulated back in lab 2. The basic task of this simulation was to traverse each segment of the display. Consistent results were produced.

Seg7Dev_Sim.v

```
`timescale 1ns / 1ps

module Seg7_Dev_Seg7_Dev_sch_tb();

// Inputs
   reg flash;
   reg [2:0] Scan;
   reg [31:0] Hexs;
   reg [7:0] point;
   reg [7:0] LES;
   reg SW0;

// Output
   wire [3:0] AN;
   wire [7:0] SEGMENT;

// Bidirs

// Instantiate the UUT
   Seg7_Dev UUT (
            .flash(flash),
            .Scan(Scan),
            .Hexs(Hexs),
            .point(point),
```

```
            .LES(LES),
            .AN(AN),
            .SEGMENT(SEGMENT),
            .SW0(SW0)
   );
// Initialize Inputs
   `ifdef auto_init
       initial begin
            flash = 0;
            Scan = 0;
            Hexs = 0;
            point = 0;
            LES = 0;
            SW0 = 0;
   `endif

      integer i;
      initial begin
            Hexs = 16'h05AF;
            point = 4'b0101;
            LES = 4'b0000;
            SW0 = 1;
            flash = 1;
            for(i = 0; i < 4; i = i + 1) begin
                  #50;
                  Scan = i;
            end
            LES = 4'b1111;
            for(i = 0; i < 4; i = i + 1) begin
                  #50;
                  Scan = i;
            end
      end
endmodule
```

6. Discussion and Conclusion

The multi-level decoding scheme has the advantage of reduced hardware cost (several smaller units instead of one big decoder) and improves the performance of the controller. Having a separate decoder for the ALU may be a hardware disadvantage as well though, with the fact that another functional unit needs to be managed. The BNE instruction has different signals than the BEQ instruction because the branch is taken in the event of an inequality. Thus, ALUop = 11 and ALU_Operation = 110. Regarding andi and other I-type instructions, this can be easily implemented with 16-bit to 32-bit sign-extenders and routed to the B port of the ALU via multiplexer. Support for other instructions can be done by extending the controller's function by adding in extra cases for the decoder. Single-cycle CPU is not very efficient and generates a lot of potentially wasted clock cycles. Compared to multi-cycle CPU, it takes more time and hardware and it enforces a single clock cycle length to be used.

This marks the conclusion of the single-cycle CPU design labs of the Computer Organization course! Although I am disappointed that I am unable to first-hand experience these labs in person, I still found it to be very rewarding. I am also disappointed that I am unable to run demo MIPS program using this CPU, and this makes the whole lab experience seem incomplete. Doing these labs were not

a complete lost though, because it greatly supplemented the material that I learned in the theoretical portion of this course. It helped me better understand how control signals played a role in instruction execution, and how instructions were processed throughout each component. Overall, it made learning about the CPU a whole lot easier and I immensely enjoyed doing these labs. These labs have helped me gain confidence with writing testing modules and interpreting simulation results. From this, I had an easier time debugging and it was a way to keep reiterating information. I really liked how these labs were structured because it progressively built up my knowledge by implementing the top module first, then designing each of the components separately.