Andres Martinez Paz

819505668

CompE 470L

Project Progress Report

My final project consists of a 16bit MIPS microarchitecture, with an interface module for demonstration purposes. The microarchitecture is capable of processing 16 different instructions. The main modules for the microarchitecture are, the ControlUnit, the InstructionMemory, the RegisterFile, and the DataMemory. For the InterfaceModule, there is a multiplexer module to display the values in specific registers or a specific memory location on the seven segment display. Also, the dipswitches will denote the address of the register or memory location to be displayed, while the IO buttons will be used to flip the display between the register file, the data memory file, or the instruction memory. Another button is used to advance the program counter. The LEDs on the IO board will give the current program counter value. Additionally, I included a clock divider that will slow down the clock signal for the memory files of the microarchitecture. This is needed for my implementation in order to reduce the bouncing from the IO push buttons.

My microarchitecture is a modified version of the 32-bit, single-cycle, MIPS microarchitecture. I included support for a total of 15 instructions, eight of which are R-type instructions, three are I-type, and two are J-type, as well as jump register (jr) and Jump and Link (jal) instructions. The encoding for each instruction is as follows:

$$R - type$$

000	sss	ttt	ddd	x	fff
OP code	**Source reg**	**t register**	**Dest reg**	**Shft amt**	**Funct code**

$$I - type$$

xxx	sss	ddd	xxxxxxx
OP code	**Source reg**	**Dest reg**	**7-bit immediate**

$$J - type$$

xxx	xxxxxxxxxxxxx
OP code	**Jump destination**

The available instructions and their details are as follows:

And ---

R-type

Op code = 000

Funct code = 000

Or ---

R-type

Op code = 000

Funct code = 001

Add ---

R-type

Op code = 000

Funct code = 010

Xor ---

R-type

Op code = 000

Funct code = 011

Shift Left Logical –-

R-type

Op code = 000

Funct code = 100

Mult --- (Result must be within 16-bit, lower 16-bit will be result of operation)

R-type

Op code = 000

Funct code = 101

Sub ---

R-type

Op code = 000

Funct code = 110

Set on Less Than ---

R-type

Op code = 000

Funct code = 111

Load Word ---

I-type

Op code = 001

Store Word ---

I-type

Op code = 010

Jump ---

J-type

Op code = 011

Branch Equal ---

I-type

Op code = 100

Add Immediate ---

I-type

Op code = 101

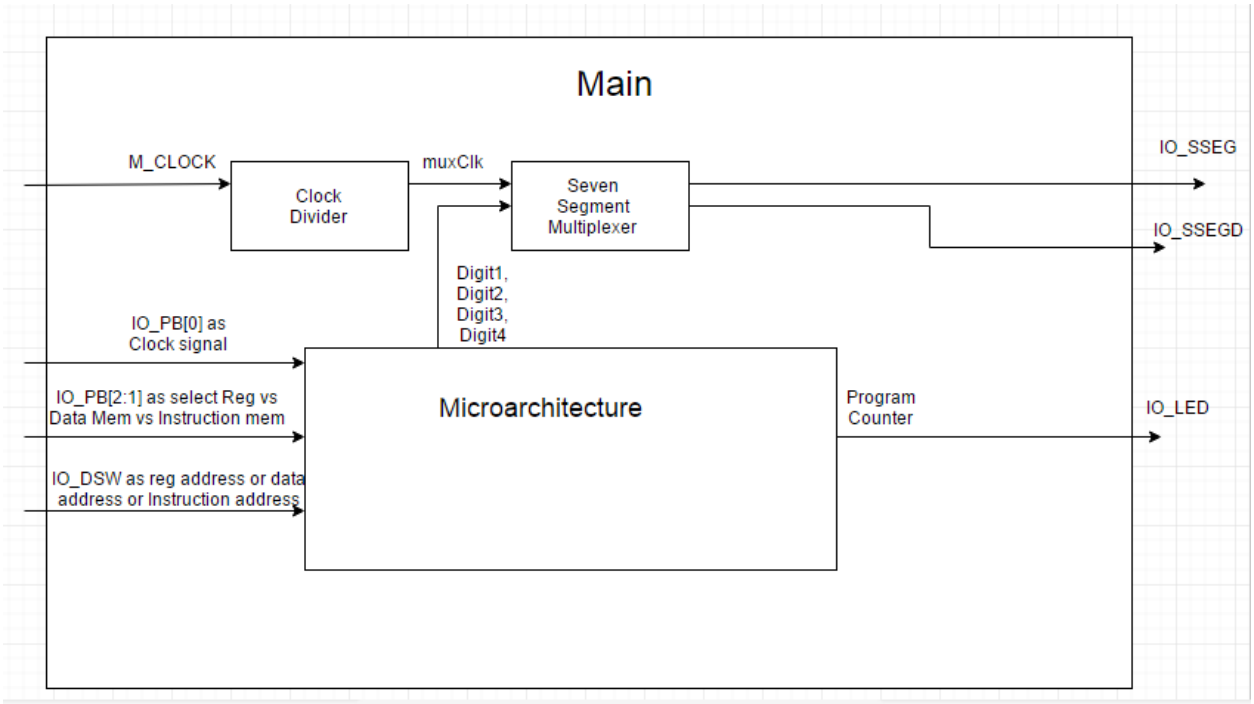
Jump and Link

Op code = 110

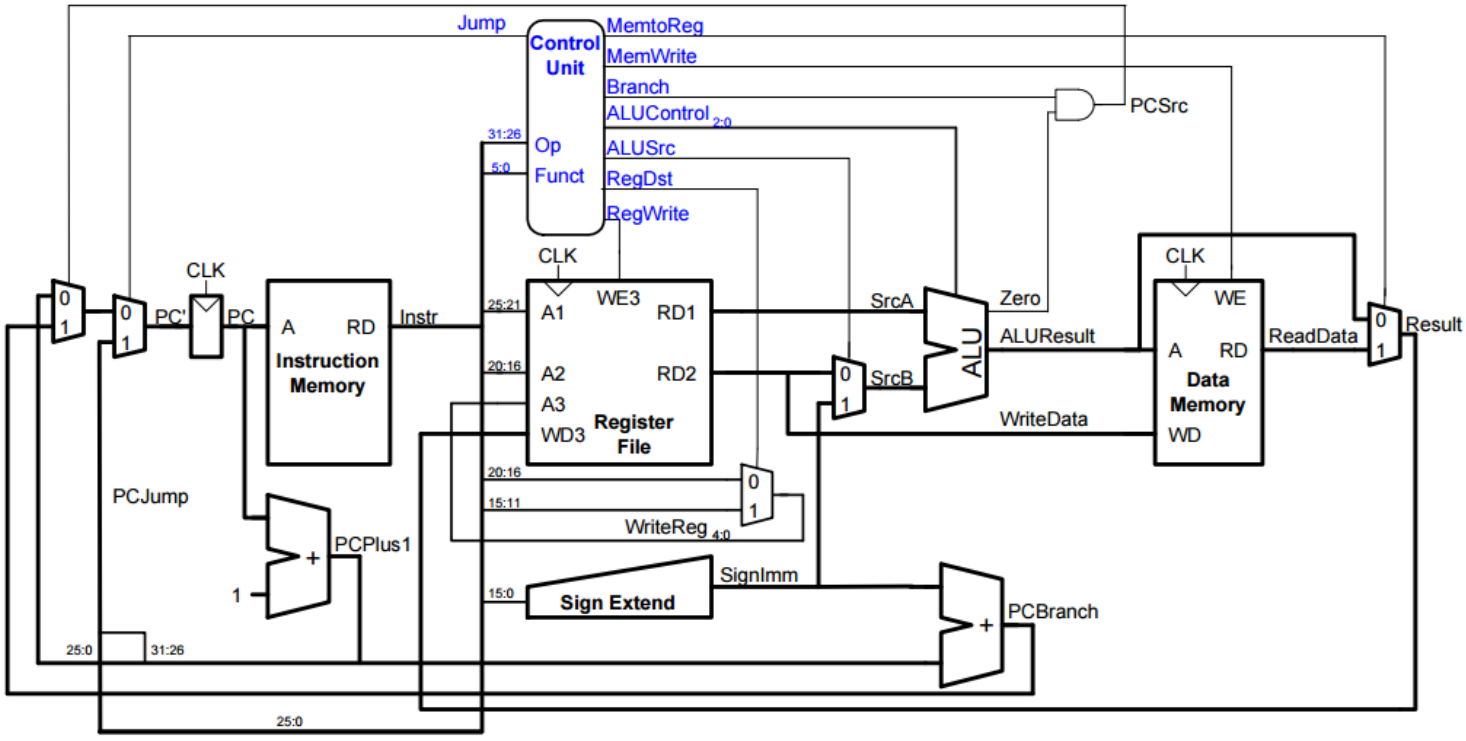
Jump Register

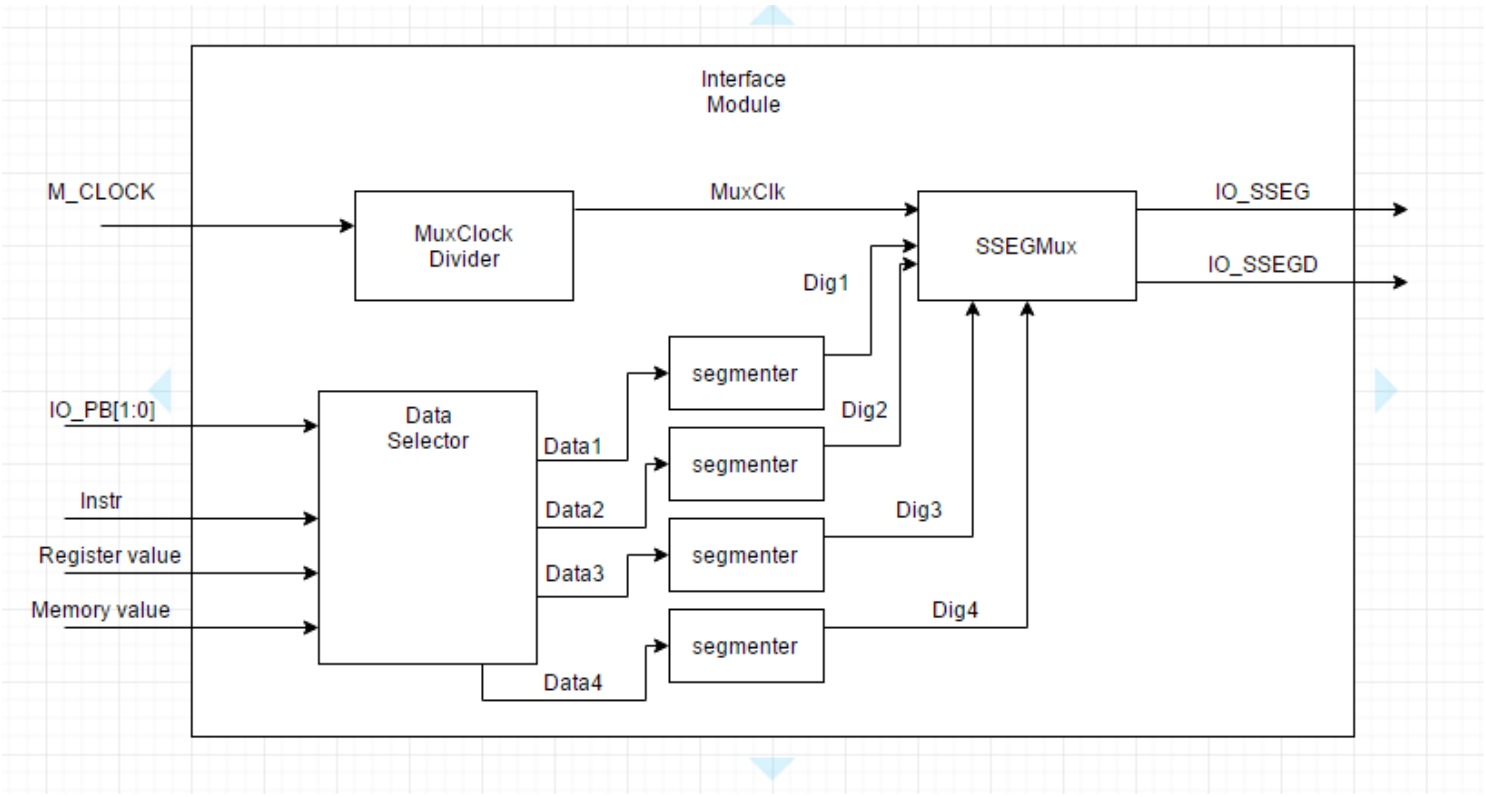
Op code = 111

Below I show the block diagram for the microarchitecture, for the InterfaceModule, and another one for the main module.



Microarchitecture





Instructions for use

To use this microarchitecture, one must load the instruction data to the InstructionMemory ROM through the file “prog.dat” in the main project directory. This file must be loaded with the instruction data prior to synthesizing and implementing the main module. One must also change the “NumOfInstructions” parameter in InstructionMemory module accordingly (Program might malfunction otherwise). Once the Instructions have been loaded, the parameter changed, and the main module synthesized and loaded to the FPGA, the user can interact with the program as follows:

Seven segment display:

The display shows the value of several addresses in each of the three main memories (Instruction memory, Register file, and Data memory) At its default state, the display shows values in the Register File. One can change the state of the display through the IO push buttons (Read below) to show values of the instruction memory, or the values of the Data memory. The addresses of the values are determined by the IO dip switches, except for the Instruction to be displayed, the seven segment display will show the following instruction to be executed.

IO push buttons:

The push buttons (From left to right, 3 2 1 0) serve different functions. IO_PB[0] serves as a single stepper clock. This button advances the program counter one cycle at a time.

IO_PB[1] serves as the program counter reset signal. For the reset signal to take effect, one must send a positive edge of the clock (IO_PB[0]) while reset signal is active. In other words, to reset the program counter, press both IO_PB[1] and IO_PB[0].

IO_PB[2] changes the state of the seven segment display, to show values of the Data memory. IO_PB[3] changes the state of the seven segment display to show values of the Instruction memory. The display will only change states while the buttons are pressed, if the user stops pressing the buttons, the display will go back to showing values of the register memory (same will happen if both IO_PB[3] and IO_PB[2] are pressed at the same time).

IO Dip switches:

The dip switches on the IO board determine the address of the register being shown. They also determine the address of the Data memory value being displayed. The address of the register being shown is determined by IO_DSW[2:0], and the address of the data memory being shown is determined by IO_DSW[3:0]. IO_DSW[7:4] are added to the design but are not used.

IO LEDs:

The LEDs on the IO board show the current Program Count.

FPGA LEDs:

The LEDs on the FPGA board show the state of the seven segment display. If no LEDs are on, the display is showing the Register file. If LED[0] is on, the display is showing Data memory. Finally, if LEDs [0] and [1] are both on, the display is showing Instruction memory.

Demo program data file

```
// Registers and Data memory are initialized to zero
A08F  // addi R1, R0, 7'b0001111    R1 = 000F
A17F  // addi R2, R0, 7'b1111111    R2 = FFFF
0530  // and R3, R1, R2      R3 = 000F
0541  // or R4, R1, R2      R4 = FFFF
11D2  // add R5, R4, R3      R5 = 000E
1563  // xor R6, R5, R2      R6 = FFF1
0022  // add R2, R0, R0      R2 = 0000
A102  // addi R2, R0, 7'b0000010       R2 = 0002
0D34  // sll R3, R3, R2       R3 = 003C
0D35  // mult R3, R3, R2      R3 = 0078
0D46  // sub R4, R3, R2      R4 = 0076
0E57  // slt R5, R3, R4     R5 = 0000
11D7  // slt R5, R4, R3     R5 = 0001
4180  // sw R3, 0(R0)     DM[0] = 0078
2300   // lw R6, 0(R0)     R6 = 0078
8E92  // beq R3, R5, 0012    Program counter = 16
0000  // nop
8F02  // beq R3, R6, 0002    Program counter = 19
0000  // nop
0000  // nop
6018  // j 0017
0000  // nop
0000  // nop
0000  // nop
C022  // jal 0022     Program Counter = 34
A241  // addi R4, R0, 0041    R4 = 0041
4203  // sw R4, 3(R0)     DM[3] = 0041
```

0000 // nop

0000 // nop

0000 // nop

0000 // nop

0000 // nop

0000 // nop

0000 // nop

0000 // nop

0000 // nop

0000 // nop

0000 // nop

0000 // nop

FC00 // jr R7 Program counter = 23

Final Register Values:

R0 = 0000

R1 = 000F

R2 = 0002

R3 = 0078

R4 = 0041

R5 = 0001

R6 = 0078

R7 = 0019

Final Data memory values:

DM[0] = 0078

DM[3]

Source Code

```
module InterfaceModule( input M_CLOCK,
                        input PB_DataMem,
                        input PB_InstrMem,
                        input [15:0] Instr,
                        input [15:0] ProgramCounter,
                        input [15:0] RegValue,
                        input [15:0] MemValue,
                        output [6:0] IO_SSEG,
                        output [3:0] IO_SSEGD,
                        output [7:0] IO_LED);

    wire muxClk;
    reg [3:0] data1;
    reg [3:0] data2;
    reg [3:0] data3;
    reg [3:0] data4;
    wire [6:0] dig1;
    wire [6:0] dig2;
    wire [6:0] dig3;
    wire [6:0] dig4;

    assign IO_LED = ProgramCounter[7:0];

    //instantiate clock divider
    muxClock u00 (.inClk(M_CLOCK),
                  .outClk(muxClk));

    //instantiate seven segment multiplexer
    SSEGMux u01 (.clk(muxClk),
```

```
                .digit1(dig1),
                .digit2(dig2),
                .digit3(dig3),
                .digit4(dig4),
                .sseg(IO_SSEG),
                .ssegd(IO_SSEGD));

segmenter s01 (.in(data1),
                .segs(dig1));

segmenter s02 (.in(data2),
                .segs(dig2));

segmenter s03 (.in(data3),
                .segs(dig3));

segmenter s04 (.in(data4),
                .segs(dig4));

always @(*)
        case ({PB_DataMem, PB_InstrMem})
                2'b00: begin   // default case, both pressed. Display register
                        data1 = RegValue[15:12];
                        data2 = RegValue[11:8];
                        data3 = RegValue[7:4];
                        data4 = RegValue[3:0];
                end
                2'b01: begin  // DataMem pressed, display Data Memory value
                        data1 = MemValue[15:12];
```

```
                        data2 = MemValue[11:8];
                        data3 = MemValue[7:4];
                        data4 = MemValue[3:0];
                    end
                    2'b10: begin  // InstrMem pressed, display Instruction
                        data1 = Instr[15:12];
                        data2 = Instr[11:8];
                        data3 = Instr[7:4];
                        data4 = Instr[3:0];
                    end
                    2'b11: begin  // no button pressed, display register
                        data1 = RegValue[15:12];
                        data2 = RegValue[11:8];
                        data3 = RegValue[7:4];
                        data4 = RegValue[3:0];
                    end
                endcase
endmodule

module ClockDivider(input inClk,
                    output reg outClk);

    reg [12:0] counter = 0;

    always @(posedge inClk) begin
        if (counter >= 8000) begin
            outClk = ~outClk;
            counter = 0;
        end else counter = counter + 1;
```

```
	end
endmodule



module muxClock(input inClk,
		output reg outClk);

	reg [9:0] counter = 0;

	always @(posedge inClk) begin
		if (counter >= 1000) begin
			outClk = ~outClk;
			counter = 0;
		end else counter = counter + 1;
	end
endmodule



module SSEGMux(input clk,
		input [6:0] digit1,
		input [6:0] digit2,
		input [6:0] digit3,
		input [6:0] digit4,
		output reg [6:0] sseg,
		output reg [3:0] ssegd);

	always @(posedge clk) begin
```

```
                case (ssegd)
                        4'b0111: begin
                                sseg = digit2;
                                ssegd = 4'b1011;
                        end
                        4'b1011: begin
                                sseg = digit3;
                                ssegd = 4'b1101;
                        end
                        4'b1101: begin
                                sseg = digit4;
                                ssegd = 4'b1110;
                        end
                        4'b1110: begin
                                sseg = digit1;
                                ssegd = 4'b0111;
                        end
                        default: begin
                                sseg = digit1;
                                ssegd = 4'b0111;
                        end
                endcase
        end
endmodule


module segmenter(input [3:0] in,
                 output reg [6:0] segs);
```

```
	always @(*) begin
		case (in)
			4'b0000: segs = 7'b1000000; // 0
			4'b0001: segs = 7'b1111001; // 1
			4'b0010: segs = 7'b0100100; // 2
			4'b0011: segs = 7'b0110000; // 3
			4'b0100: segs = 7'b0011001; // 4
			4'b0101: segs = 7'b0010010; // 5
			4'b0110: segs = 7'b0000010; // 6
			4'b0111: segs = 7'b1011000; // 7
			4'b1000: segs = 7'b0000000; // 8
			4'b1001: segs = 7'b0010000; // 9
			4'b1010: segs = 7'b0001000; // A
			4'b1011: segs = 7'b0000011; // b
			4'b1100: segs = 7'b1000110; // C
			4'b1101: segs = 7'b0100001; // d
			4'b1110: segs = 7'b0000110; // E
			4'b1111: segs = 7'b0001110; // F
			default: segs = 7'b1111111; // empty
		endcase
	end
endmodule


module Mux16(  input [15:0] in0,
		input [15:0] in1,
		input sel,
		output reg [15:0] out);
```

```
	always @(*)
		case (sel)
			1'b0: out = in0;
			1'b1: out = in1;
		endcase
endmodule



module DataFile( input clk,
		input WE,
		input [3:0] Address,
		input [15:0] WriteData,
		input [3:0] ShowAddress,
		output reg [15:0] ReadData,
		output reg [15:0] ShowData);

	reg [3:0] a;
	wire [15:0] spo;

	always @(posedge clk) begin
		a = ShowAddress;
		ShowData = spo;
		a = Address;
		ReadData = spo;
	end

	DataMemory dm00 (
			.a(a),
```

```
                .d(WriteData),
                .clk(clk),
                .we(WE),
                .spo(spo));
endmodule


module PCBranch(input signed [15:0] SignImm,
                input signed [15:0] PCPlus1,
                output signed [15:0] PCBranch);

        assign PCBranch = SignImm + PCPlus1;
endmodule

module ALU(input [2:0] ALUControl,
                input [15:0] srcA,
                input [15:0] srcB,
                output reg [15:0] ALUResult,
                output reg Zero);

        wire signed [15:0] sgnSrcA;
        wire signed [15:0] sgnSrcB;
        reg [31:0] multResult;
        assign sgnSrcA[15:0] = srcA[15:0];
        assign sgnSrcB[15:0] = srcB[15:0];

        always @* begin
                case (ALUControl)
                        3'b000: begin           //Code for bitwise and
```

```
            ALUResult = srcA & srcB;
            Zero = 1'bx;
    end
    3'b001: begin               // Code for bitwise or
            ALUResult = srcA | srcB;
            Zero = 1'bx;
    end
    3'b010: begin               //code for add
            ALUResult = sgnSrcA + sgnSrcB;
            Zero = 1'bx;
    end
    3'b011: begin               //Code for xor
            ALUResult = srcA ^ srcB;
            Zero = 1'bx;
    end
    3'b100: begin    //Code for shift left
            ALUResult = srcA << srcB;
            Zero = 1'bx;
    end
    3'b101: begin  //Code for mult. Result must be within 32bits, rest will be lost
            multResult = srcA*srcB;
            ALUResult = multResult[15:0];
            Zero = 1'bx;
    end
    3'b110: begin               //code for subtract
            if (sgnSrcA == sgnSrcB) begin
                    ALUResult = sgnSrcA - sgnSrcB;
                    Zero = 1'b1;
            end else begin
```

```
                                        ALUResult = sgnSrcA - sgnSrcB;
                                        Zero = 1'b0;
                                end
                        end
                        3'b111: begin               //code for slt
                                if (sgnSrcA <= sgnSrcB) ALUResult = 1'b1;
                                else ALUResult = 1'b0;
                                Zero = 1'bx;
                        end
                        default: begin
                                ALUResult = 0;
                                Zero = 1'bx;
                        end
                endcase
        end
endmodule



module Mux3( input [2:0] in0,
                 input [2:0] in1,
                 input sel,
                output reg [2:0] out);

        always @(*)
                case(sel)
                        1'b0: out = in0;
                        1'b1: out = in1;
                endcase
```

```
endmodule



module SignExtend(input [6:0] imm,
                  output reg [15:0] signImm);

    always @* begin
        signImm[6:0] = imm[6:0];
        signImm[15:7] = {9{imm[6]}};
    end
endmodule

module RegisterFile( input clk,
        input [2:0] Address1,
        input [2:0] Address2,
        input [2:0] Address3,
        input WE,
        input [15:0] WriteData,
        input [2:0] ShowAddress,
        output reg [15:0] RD1,
        output reg [15:0] RD2,
        output reg [15:0] ShowData);

    // declare the nets types for the instantiation of memory block
    reg [2:0] addra;
    wire [15:0] douta;

    always @(posedge clk or negedge clk) begin
```

```
		case (clk)
			1'b1: begin
				addra = Address1;
				RD1 = douta;
				addra = Address2;
				RD2 = douta;
				addra = ShowAddress;
				ShowData = douta;
			end
			1'b0: begin
				addra = Address3;
				RD1 = RD1;
				RD2 = RD2;
				ShowData = ShowData;
			end
		endcase
	end

	RegisterMem m00 ( .clka(clk), // input clka
			.wea(WE), // input [0 : 0] wea
			.addra(addra), // input [3 : 0] addra
			.dina(WriteData), // input [15 : 0] dina
			.douta(douta)); // output [15 : 0] douta

endmodule


module ControlUnit( input [2:0] Op,
		input [2:0] Funct,
```

```
        output MemToReg,
        output MemWrite,
        output Branch,
        output [2:0] ALUControl,
        output ALUSrc,
        output RegDst,
        output RegWrite,
        output Jump,
        output JAL,
        output JumpReg,
        output RegRA,
        output LoadUpper);

wire [1:0] ALUOp;

//instantiate main decoder for control unit
MainDecoder u00 (.Op(Op),
                 .RegWrite(RegWrite),
                  .RegDst(RegDst),
                 .ALUSrc(ALUSrc),
                 .Branch(Branch),
                 .MemWrite(MemWrite),
                 .MemToReg(MemToReg),
                  .Jump(Jump),
                  .JAL(JAL),
                  .JumpReg(JumpReg),
                  .RegRA(RegRA),
                 .LoadUpper(LoadUpper),
                  .ALUOp(ALUOp) );
```

```
        //instantiate Alu decoder for control unit
        ALUDecoder u01 (.Funct(Funct),
                        .ALUOp(ALUOp),
                        .ALUControl(ALUControl) );

endmodule




module MainDecoder( input [2:0] Op,
                        output reg RegWrite,
                        output reg RegDst,
                        output reg ALUSrc,
                        output reg Branch,
                        output reg MemWrite,
                        output reg MemToReg,
                        output reg Jump,
                        output reg JAL,
                        output reg JumpReg,
                        output reg RegRA,
                        output reg [1:0] ALUOp);
        always @* begin
                case (Op)
                        3'b000: begin    //R-type instructions
                                RegWrite = 1'b1;
```

```
            RegDst = 1'b1;
            ALUSrc = 1'b0;
            Branch = 1'b0;
            MemWrite = 1'b0;
            MemToReg = 1'b0;
            ALUOp = 2'b10;
            Jump = 1'b0;
            JAL = 1'b0;
            JumpReg = 1'b0;
            RegRA = 1'b0;
        end
        3'b001: begin      //lw instruction
            RegWrite = 1'b1;
            RegDst = 1'b0;
            ALUSrc = 1'b1;
            Branch = 1'b0;
            MemWrite = 1'b0;
            MemToReg = 1'b1;
            ALUOp = 2'b00;
            Jump = 1'b0;
            JAL = 1'b0;
            JumpReg = 1'b0;
            RegRA = 1'b0;
        end
        3'b010: begin      //sw instruction
            RegWrite = 1'b0;
            RegDst = 1'bx;
            ALUSrc = 1'b1;
            Branch = 1'b0;
```

```
            MemWrite = 1'b1;
            MemToReg = 1'bx;
            ALUOp = 2'b00;
            Jump = 1'b0;
            JAL = 1'b0;
            JumpReg = 1'b0;
            RegRA = 1'b0;
        end
        3'b011: begin      //j instruction
            RegWrite = 1'b0;
            RegDst = 1'bx;
            ALUSrc = 1'bx;
            Branch = 1'b0;
            MemWrite = 1'b0;
            MemToReg = 1'bx;
            ALUOp = 2'bxx;
            Jump = 1'b1;
            JAL = 1'b0;
            JumpReg = 1'b0;
            RegRA = 1'b0;
        end
        3'b100: begin      //beq instruction
            RegWrite = 1'b0;
            RegDst = 1'bx;
            ALUSrc = 1'b0;
            Branch = 1'b1;
            MemWrite = 1'b0;
            MemToReg = 1'bx;
            ALUOp = 2'b01;
```

```
            Jump = 1'b0;
            JAL = 1'b0;
            JumpReg = 1'b0;
            RegRA = 1'b0;
    end
    3'b101: begin      //addi instruction
            RegWrite = 1'b1;
            RegDst = 1'b0;
            ALUSrc = 1'b1;
            Branch = 1'b0;
            MemWrite = 1'b0;
            MemToReg = 1'b0;
            ALUOp = 2'b00;
            Jump = 1'b0;
            JAL = 1'b0;
            JumpReg = 1'b0;
            RegRA = 1'b0;
    end
    3'b110: begin      //jal instruction
            RegWrite = 1'b1;
            RegDst = 1'bx;
            ALUSrc = 1'bx;
            Branch = 1'b0;
            MemWrite = 1'b0;
            MemToReg = 1'bx;
            ALUOp = 2'bxx;
            Jump = 1'b1;
            JAL = 1'b1;
            JumpReg = 1'b0;
```

```
                RegRA = 1'b1;
            end
            3'b111: begin      //jr instruction
                RegWrite = 1'b0;
                RegDst = 1'bx;
                ALUSrc = 1'b0;
                Branch = 1'b0;
                MemWrite = 1'b0;
                MemToReg = 1'b0;
                ALUOp = 2'b00;
                Jump = 1'b0;
                JAL = 1'b0;
                JumpReg = 1'b1;
                RegRA = 1'b0;
            end
        endcase
    end
endmodule


module ALUDecoder( input [2:0] Funct,
                   input [1:0] ALUOp,
                   output reg [2:0] ALUControl);

    ////////////////////////////
    // Funct codes to ALU operation
    //
    // 3'b000 -> and
    // 3'b001 -> or
```

```
        // 3'b010  ->  add
        // 3'b011  ->  xor
        // 3'b100  ->  sll
        // 3'b101  ->  mult
        // 3'b110  ->  sub
        // 3'b111  ->  slt
        //
        ////////////////////////
        always @*
                case (ALUOp)
                        2'b00: ALUControl = 3'b010;  // send addition code to ALUControl signal
                        2'b01: ALUControl = 3'b110;  // send subtraction code to
                        default: ALUControl = Funct; // send Funct code to ALUControl
                endcase
endmodule


module PCPlus1( input [15:0] PC,
                 output [15:0] PCPlus1);

        assign PCPlus1 = PC + 1;
endmodule


module ProgramCounter( input [15:0] PCPrime,
                        input rst,
                        input clk,
                        output reg [15:0] PC);
```

```
	always @(posedge clk)
		case (rst)
			1'b0: PC = PCPrime;
			1'b1: PC = 0;
		endcase
endmodule



module CPU( input M_CLOCK,
		 input PB_CLK,
		input PB_RST,
		input PB_DataMem,
		input PB_InstrMem,
		input [7:0] IO_DSW,
		output [6:0] IO_SSEG,
		output IO_SSEG_DP,
		output [3:0] IO_SSEGD,
		output IO_SSEG_COL,
		output [7:0] IO_LED,
		output [3:0] F_LED);

	assign IO_SSEG_COL = 1'b1;
	assign IO_SSEG_DP = 1'b1;

	wire [15:0] PC;
	wire [15:0] PCPrime;
	wire PCSrc;
	wire [15:0] PCPlus1;
	wire [15:0] PCBranch;
```

```
wire [15:0] Mux0Out;
wire [15:0] Mux1Out;
wire [2:0] Mux3Out;
wire [15:0] Mux6Out;
wire [15:0] Mux7Out;
wire [15:0] Instr;
wire Jump;
wire [2:0] Op;
wire [2:0] Funct;
wire MemToReg;
wire MemWrite;
wire Branch;
wire [2:0] ALUControl;
wire ALUSrc;
wire RegDst;
wire RegWrite;
wire JAL;
wire JumpReg;
wire RegRA;
wire [2:0] Address1;
wire [2:0] Address2;
wire [2:0] Address3;
wire [15:0] WriteData;
wire [2:0] ShowAddress;
wire [15:0] RD1;
wire [15:0] RD2;
wire [15:0] ShowData;
wire [6:0] imm;
wire [15:0] signImm;
```

```
wire [15:0] ALUSrcA;
wire [15:0] ALUSrcB;
wire [15:0] ALUResult;
wire Zero;
wire [15:0] ReadData;

wire [3:0] ShowDataAddress;
wire [15:0] ShowDataMemory;

assign Op = Instr[15:13];
assign Funct =  Instr[2:0];
assign Address1 = Instr[12:10];
assign Address2 = Instr[9:7];
assign imm = Instr[6:0];

assign ShowAddress = ~IO_DSW[2:0];
assign ShowDataAddress = ~IO_DSW[3:0];
assign ALUSrcA = RD1;
assign clk = ~PB_CLK;

//instantiate clock divider to drive memories and registers of CPU (to eliminate PB glitches)
ClockDivider cd00 (
                    .inClk(M_CLOCK),
                    .outClk(CPUClk));

//instantiate PCSrc mux (16bit)
Mux16 mux00 (
                    .in0(PCPlus1),
                    .in1(PCBranch),
```

```
                .sel(PCSrc),
                .out(Mux0Out));


//instantiate Jump mux
Mux16 mux01 (
                .in0(Mux0Out),
                .in1({PCPlus1[15:13], Instr[12:0]}),
                .sel(Jump),
                .out(Mux1Out));

//instantiate JumpReg mux
Mux16 mux02 (
                .in0(Mux1Out),
                .in1(ALUResult),
                .sel(JumpReg),
                .out(PCPrime));

//instantiate ProgramCounter
ProgramCounter pc00 (
                .PCPrime(PCPrime),
                .rst(~PB_RST),
                .clk(CPUClk),
                .write(clk),
                .PC(PC));

//instantiate PCPlus1
PCPlus1 pcp00 (
                .PC(PC),
```

```
                .PCPlus1(PCPlus1));

//instantiate InstructionMemory
InstructionMem im00 (
                .Address(PC[7:0]),
                .ReadData(Instr));

//instantiate Control Unit
ControlUnit cu00 (
                .Op(Op),
                .Funct(Funct),
                .MemToReg(MemToReg),
                .MemWrite(MemWrite),
                .Branch(Branch),
                .ALUControl(ALUControl),
                .ALUSrc(ALUSrc),
                .RegDst(RegDst),
                .RegWrite(RegWrite),
                .Jump(Jump),
                .JAL(JAL),
                .JumpReg(JumpReg),
                .RegRA(RegRA));

//instantiate RegisterFile
RegisterFile rf00 (
                .clk(CPUClk),
                .Address1(Address1),
                .Address2(Address2),
                .Address3(Address3),
```

```
                .WE(RegWrite),
                .write(clk),
                .WriteData(WriteData),
                .ShowAddress(ShowAddress),
                .RD1(RD1),
                .RD2(RD2),
                .ShowData(ShowData));

//instantiate SignExtend
SignExtend se00 (
                .imm(imm),
                .signImm(signImm));

//instantiate RegDst mux
Mux3 mux03 (
                .in0(Instr[9:7]),
                .in1(Instr[6:4]),
                .sel(RegDst),
                .out(Mux3Out));

//instantiate RegRA mux
Mux3 mux04 (
                .in0(Mux3Out),
                .in1(3'b111),  //address for RA register
                .sel(RegRA),
                .out(Address3));

//instantiate ALUSrc mux
Mux16 mux05 (
```

```
                .in0(RD2),
                .in1(signImm),
                .sel(ALUSrc),
                .out(ALUSrcB));

//Instantiate ALU
ALU alu00 (
                .ALUControl(ALUControl),
                .srcA(ALUSrcA),
                .srcB(ALUSrcB),
                .ALUResult(ALUResult),
                .Zero(Zero));

//Instantiate PCBranch
PCBranch pcb00 (
                .SignImm(signImm),
                .PCPlus1(PCPlus1),
                .PCBranch(PCBranch));

//instantiate AND gate for PCSrc
and and00 (PCSrc, Branch, Zero);

//instantiate DataFile
DataFile df00 (
                .clk(CPUClk),
                .WE(MemWrite),
                .write(clk),
                .Address(ALUResult[3:0]),
                .WriteData(RD2),
```

```
                .ShowAddress(ShowDataAddress),
                .ReadData(ReadData),
                .ShowData(ShowDataMemory));

//instantiate MemToReg mux
Mux16 mux06 (
                .in0(ALUResult),
                .in1(ReadData),
                .sel(MemToReg),
                .out(Mux6Out));


//instantiate JAL mux
Mux16 mux07 (
                .in0(Mux6Out),
                .in1(PCPlus1),
                .sel(JAL),
                .out(WriteData));


//instantiate InterfaceModule
InterfaceModule inm00 (
                .M_CLOCK(M_CLOCK),
                .PB_DataMem(PB_DataMem),
                .PB_InstrMem(PB_InstrMem),
                .Instr(Instr),
                .ProgramCounter(PC),
                .RegValue(ShowData),
                .MemValue(ShowDataMemory),
```

```
                    .IO_SSEG(IO_SSEG),
                    .IO_SSEGD(IO_SSEGD),
                    .IO_LED(IO_LED),
                    .F_LED(F_LED));
endmodule
```