UM–SJTU Joint Institute

VE370 - Introduction to Computer Organization Project 2 Report

Group	32
Name	**ID**
Han Liying	517370910069
Gracia Stefani	517370990022
Hu Jiaoyang	517370910140

November 5, 2019

Contents

1 Introduction

MIPS structure is commonly seen to implement the processor of computers. In this course VE370, Introduction to Computer Organization, we have learned a lot of instructions supported by MIPS structure, and also the fundamental hardware structure to support those instructions. In this project, we will design the hardware structure to support several MIPS instructions. We will model and simulate in Verilog HDL, and synthesize by using Xilinx synthesis tool.

2 Objective

Our goal is to model both single cycle and pipelined implementation of MIPS computer in Verilog that support a subset of MIPS instruction set including:

- The memory-reference instructions load word (`lw`) and store word (`sw`).
- The arithmetic-logical instructions `add`, `addi`, `sub`, `and`, `andi`, `or`, and `slt`.
- The jumping instructions branch equal (`beq`), branch not equal (`bne`), and jump (`j`),

For single cycle implementation, we will mainly design and use 11 modules, which are modules for PC, Mux, Adder, ALU, ALU Control, Control, Register File, Instruction Memory, Data Memory, Sign Extension, and Shift Left. We write testbenches to simulate our single cycle implementation.

For pipelined implementation, we will mainly design and use 14 modules, which are modules for PC, Mux, Adder, ALU, ALU Control, Control, Register File, Instruction Memory, Data Memory, Sign Extension, Shift Left, Hazard Detection, Forwarding Unit, and Pipeline Register. We write testbenches to simulate our pipelined implementation, and then use FPGA board to demonstrate our result by SSD display.

3 Single Cycle Implementation

3.1 Top Level Diagram

Our diagram of top-level block for single cycle implementation is similar to the one shown in Figure 1, but we add one more control signal `nBranch` for detecting `bne`, and then add one `or` gate whose input is `nBranch` and `eBranch` (for `beq`). The output of this `or` gate is then forwarded to the **Mux module** which selects from branch address or PC + 4 address. Figure 2 shows the schematic generated by **Vivado** software.

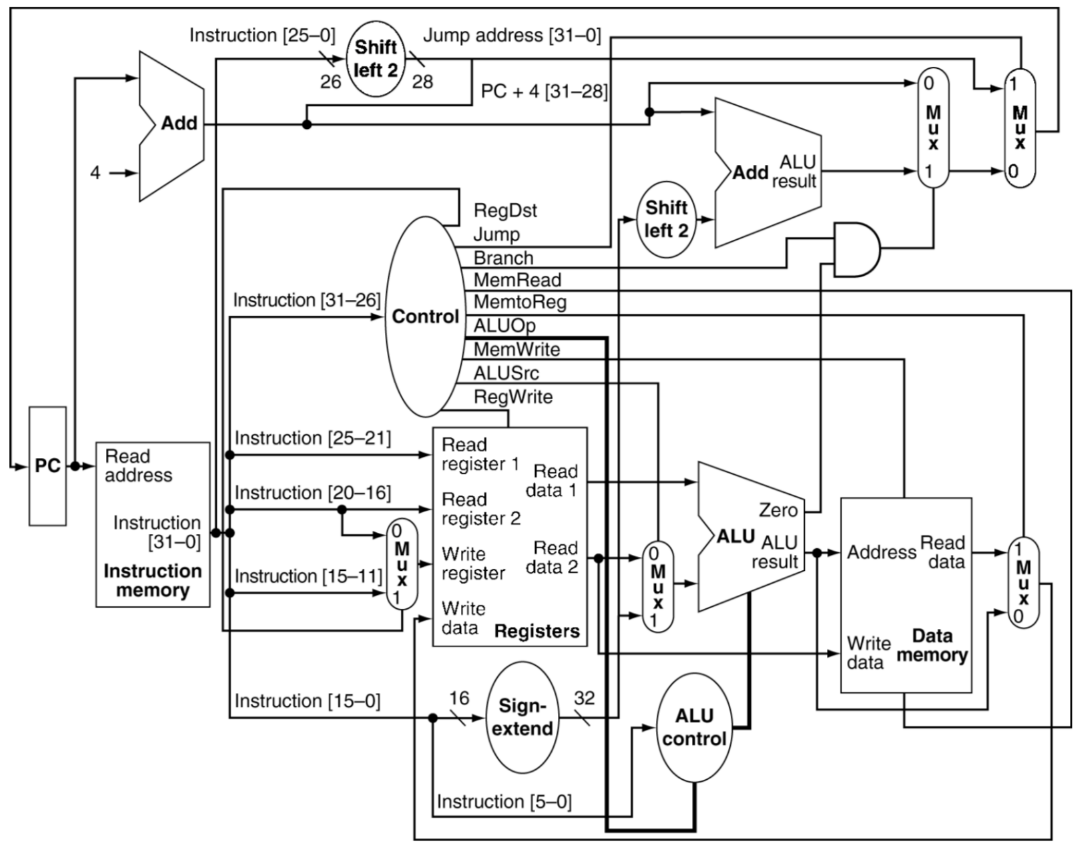


Figure 1: The top-level block diagram for single cycle implementation [1].

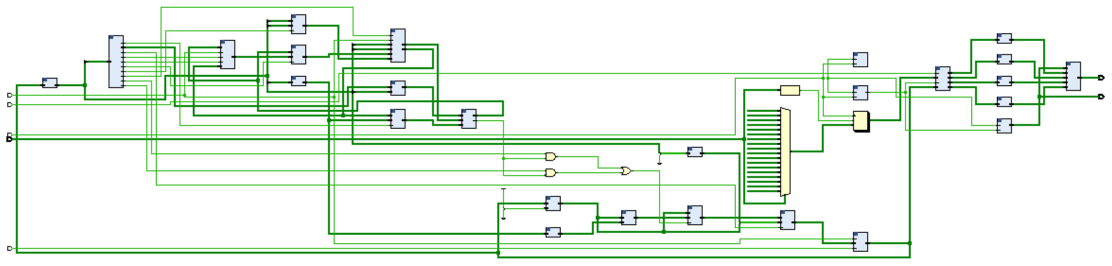


Figure 2: The schematic diagram for single cycle implementation.

3.2 Modules Design

We mainly need to design and use 11 component modules, which are modules for PC, Mux, Adder, ALU, ALU Control, Control, Register File, Instruction Memory, Data Memory, Sign Extension, and Shift Left. Finally write a main module to combine them together so that we can fulfill the function of single cycle structure.

3.2.1 Components Design

PC module is used to calculate the address of PC + 4, we write the module in `pc.v`. The output is `outAddrs`, and input is `pcAddrs`, `clock` and `reset`. At first use we initialize the output of this module to be 0.

```
module PC (pcAddrs, clock, outAddrs, reset);
  input [31:0] pcAddrs;
  input clock,hold,reset;
  output [31:0] outAddrs;

  reg[31:0] pcReg;

  initial begin
    pcReg = 32'b0;
  end

  always @(negedge clock) begin
    if(reset == 1'b1) pcReg = 32'b0;
    else pcReg = pcAddrs;
  end
  assign outAddrs = pcReg;
endmodule
}
```

ALU module is written in `ALU.v`. The ALU has two input ports, `in2` and `in2` which takes in two 32 bits data. It also takes control signal in the `control` port to tell the ALU which operation to execute. The control signal 4'b0000 tells the ALU to do the `and` operation, 4'b0001 for the `or` operation, 4'b0010 for the `add` operation, 4'b0110 for the `substract` operation, 4'b0111 for the `set-on-less-than` operation and 4'b1100 for the nor operation. If the control signal is not recognized by the ALU, it will display "ALU invalid control signal [control]" for the simulation. The Verilog code of the module is as follows:

```
module ALU (in1,in2, control, zero, aluRes);
  input signed [31:0] in1, in2; //Signed input
  input [3:0] control; //Control signal input telling ALU
             //Which instruction to execute
  output [31:0] aluRes; //The result output of the ALU
  output zero;

  reg [31:0] aluRes;
  reg zero;

  always @ (*) begin
    case (control)
```

```
      4'b0000: aluRes <= in1 & in2; //and
      4'b0001: aluRes <= in1 | in2; //or
      4'b0010: aluRes <= in1 + in2; //add
      4'b0110: aluRes <= in1 - in2; //substract
      4'b0111: begin //set-on-less-than
        if(in1 < in2) begin
          aluRes <= 32'b1;
        end
        else begin
          aluRes <= 32'b0;
        end
      end
      4'b1100: aluRes <= ~(in1 | in2); //nor
      default begin
        \$display("ALU␣invalid␣control␣signal␣\%b", control);
      end
    endcase
    if (aluRes == 32'b0) zero = 1'b1;
    else zero = 1'b0;
  end
endmodule
```

Data Memory module is written in `dataMem.v` . The Data Memory module has 5 inputs: `clock`, `memRead`, `memWrite`, `address`, `wData`. The Data Memory module is triggered by the negative edge of the clock such that it will only write to the memory at the negative edge of the clock. It will only write the data `wData` to the memory in the address `address` if `memWrite == 1`. In this design, it will always read from the memory, however the `memRead` signal is used later in the main implementation. It also has a parameter memWords, indicating how many words wants to be stored in the memory, which is set to 32 by default. The Data Memory has a private register `dataMem` which can hold up to `4*memWords` of 8-bit data, which means the data memory can store `memWords` words. The data memory is initially cleared to 0. The Verilog code is as follows:

```

module dataMem(clock, memRead, memWrite, address, wData, rData);
  input clock, memRead, memWrite;
  input [31:0] address, wData;
  output [31:0] rData;

  parameter memWords = 32; //can store 32 32-bits data (32x32 memory)
  reg [7:0] dataMem [4*memWords - 1:0];

  integer i;
  initial begin //clear the memory
    //rData <= 8'b0; //read data will initially be 0
    for (i=0; i < 4*memWords; i = i+1) begin
      dataMem[i] = 8'b0;
    end
  end

  assign rData = \{ dataMem[address],
    dataMem[address+1],
    dataMem[address+2],
   dataMem[address+3]\};

  always @(negedge clock) begin
    if (memWrite == 1'b1) begin
     \{dataMem[address],
      dataMem[address+1],
      dataMem[address+2],
      dataMem[address+3] \}<= wData;
      \$display("writing␣to␣dataMem[\%h],␣supposed␣value␣=␣0x\%h", address,
         wData);
    end

    if(memRead == 1'b1) begin
      \$display("Data␣read␣from␣dataMem[\%h]␣=␣0x\%h", address, rData);
    end

end

endmodule
```

ALU control module is written in ALUcontrol.v. It receives one input signal ALUop from **Control module**, and another input signal funct from the lower 6 bits of instruction code. Then the output ALUsel is used to control the calculation of **ALU module**. The following code shows how it is implemented:

```
module ALUcontrol(ALUop, funct, ALUsel);

  input [1:0] ALUop;     // 2-bit ALUop from opcode
  input [5:0] funct;     // 5-bit funct from instruction
  output reg [3:0] ALUsel;   // 4-bit output of ALUcontrol unit

  initial begin
    ALUsel = 4'b0000;        // initialize ALUsel
  end

  always @(*) begin
    if (ALUop == 2'b00) begin   // lw, sw, addi
      ALUsel <= 4'b0010;
    end else
    if (ALUop == 2'b01) begin   // beq, bne
      ALUsel <= 4'b0110;
    end else
    if (ALUop == 2'b11) begin   // andi
      ALUsel <= 4'b0000;
    end
    else if (ALUop == 2'b10) begin
      if (funct == 6'b100000) begin     //add
        ALUsel <= 4'b0010;
      end
      else if (funct == 6'b100010) begin    //subtract
        ALUsel <= 4'b0110;
      end
      else if (funct == 6'b100100) begin    //and
        ALUsel <= 4'b0000;
      end
      else if (funct == 6'b100101) begin    //or
        ALUsel <= 4'b0001;
      end
      else if (funct == 6'b101010) begin    //slt
        ALUsel <= 4'b0111;
      end
    end
  end

endmodule
```

Control module is written in `Control.v`. It receives input signal `opcode` from the highest 6 bits of instruction code, and then output 10 control signals `RegDst`, `Jump`, `eBranch`, `nBranch`, `MemRead`, `MemtoReg`, `ALUop`, `ALUSrc`, `MemWrite`, `RegWrite`. The following code shows how it is implemented:

```
// Get control signal from opcode
// Support: 'lw', 'sw', 'add', 'addi', 'sub', 'and', 'andi', 'or', 'slt',
// 'beq', 'bne', 'j'

module Control(
       opcode,          // The opcode we get from insturction.
       RegDst,          // Determine the destination register to wirte,
                        // rt or rd.
       Jump,            // Do 'jump' instruction.
```

```
      eBranch,         // Do 'beq' instruction.
      nBranch,         // Do 'bne' instruction
      MemRead,         // Whether read data from data memory.
      MemtoReg,        // Which data to write into RF, the output of ALU
                       // or data read from data memory.
      ALUop,           // Get ALUop from opcode for ALU control unit.
      ALUSrc,          // Determine the source to use for ALU,
                       //the data from RF or extended immediate.
      MemWrite,        // Whether write data into data memory.
      RegWrite         // Whether write data into RF.
);

  input [5:0] opcode;
  output reg [1:0] ALUop;
  output reg RegDst, Jump, eBranch, nBranch, MemRead, MemtoReg, ALUSrc,
     MemWrite, RegWrite;

  initial begin
    // Initialize
    Jump = 1'b0;
    ALUSrc = 1'b0;
    RegDst = 1'b0;
    ALUop = 2'b00;
    MemWrite = 1'b0;
    MemRead = 1'b0;
    eBranch = 1'b0;
    nBranch = 1'b0;
    MemtoReg = 1'b0;
    RegWrite = 1'b0;
  end

  always @ (*) begin
    Jump <= 1'b0;
    ALUSrc <= 1'b0;
    RegDst <= 1'b0;
    ALUop <= 2'b00;
    MemWrite <= 1'b0;
    MemRead <= 1'b0;
    eBranch <= 1'b0;
    nBranch <= 1'b0;
    MemtoReg <= 1'b0;
    RegWrite <= 1'b0;
    if (opcode == 6'h0) begin  // add/and/sub/or/slt
      RegDst <= 1'b1;
      ALUop <= 2'b10;
```

```verilog
      RegWrite <= 1'b1;
    end else
      if (opcode == 6'h8) begin  // addi
        ALUSrc <= 1'b1;
        ALUop <= 2'b00;
        RegWrite <= 1'b1;
      end else
      if (opcode == 6'hc) begin  // andi
        ALUSrc <= 1'b1;
        ALUop <= 2'b11;
        RegWrite <= 1'b1;
      end else
      if (opcode == 6'h4) begin  // beq
        ALUop <= 2'b01;
        eBranch <= 1'b1;
      end else
      if (opcode == 6'h5) begin  // bne
        ALUop <= 2'b01;
        nBranch <= 1'b1;
      end else
      if (opcode == 6'h23) begin  // lw
        ALUSrc <= 1'b1;
        MemRead <= 1'b1;
        MemtoReg <= 1'b1;
        RegWrite <= 1'b1;
      end else
      if (opcode == 6'h2b) begin  // sw
        ALUSrc <= 1'b1;
        MemWrite <= 1'b1;
      end else
      if (opcode == 6'h2) begin  // j
        Jump <= 1'b1;
      end
    end

endmodule
```

Register File module is written in `RegisterFile.v`. It is triggered by clock signal `clk`, and receives input signal `RegWrite` from **Control module**. The input signal also includes `readRegister1`, `readRegister2`, `writeRegister`, `writeData`. Then the output of this module are `readData1` and `readData2`. Only when the `RegWrite` signal is 1, the register file begins to write data into register file. The following code shows how it is implemented:

```verilog
// 32*32 Register File
```

```

module RegisterFile(
       clk,             // Clock signal
       RegWrite,        // Control signal to write
       readRegister1,   // First register of read
       readRegister2,   // Second register of read
       writeRegister,   // Register to write
       writeData,       // Data to write
       readData1,       // Data read from the first register
       readData2       // Data read from the second register
);

  input clk, RegWrite;
  input [4:0] readRegister1, readRegister2, writeRegister;
  input [31:0] writeData;
  output [31:0] readData1, readData2;


  reg [31:0] RegisterFile [0:31];
  // initialize
  integer i;
  initial begin
    for (i=0; i<32; i=i+1)  // initialize the register file
      RegisterFile[i] = 32'b0;
  end

    // always read
  assign readData1 = RegisterFile[readRegister1];
  assign readData2 = RegisterFile[readRegister2];

  always @ (posedge clk) begin
    if (RegWrite == 1'b1) begin
      RegisterFile[writeRegister] <= writeData;
    end
  end

endmodule
```

ShiftLeft is written in `ShiftLeft2.v` . The ShiftLeft module has 1 input: `Input`, and 1 output:`ShiftedOutput`. The ShiftLeft module is asynchronous, and is activated only when there is input. There are two uses for shiftleft2, first is in branch instructions where the $address = PC + offset * 4 + 4$, with offset being a 4 digit number.The second usage is in jump instructions, where the 26 digits of immediate is shifted by left 2 to 28 digits, before the top four digits of PC is appended to the address. Since in these two usages, the digits of the number can be different, we set the digits as a parameter. The Verilog code is as follows:

```verilog
module ShiftLeft2(
Input,
ShiftedOutput
);
//by HU Jiaoyang 517370910140 in VE370 Project 2 in 19FA
//There are two uses for shiftleft2, first is in branch instructions where
   the address = PC+offset*4+4, with offset being a 4 digit number
//The second usage is in jump instructions, where the 26 digits of immediate
    is shifted by left 2 to 28 digits, before the top four digits of PC is
   appended to the address
parameter ShiftAmount = 2;
parameter Digits = 32;//this is subject to change depending on which path
   this component is on

input [Digits - 1:0] Input;
output [Digits - 1:0] ShiftedOutput;

assign ShiftedOutput = {Input[Digits - 1-ShiftAmount:0],{ShiftAmount{1'b0
   }}};//this instruction takes the lower 30(could be less 30) digits and
   adds two 00 after the digits,
//which times the digits by 4 and shifts two digits
endmodule
```

SignExtend module is written in `SignExtend.v`. The SignExtend module has 1 input: `Input`, and 1 output:`Output`. The SignExtend module is asynchronous, and is activated only when there is input. The module is normally used in I-type instructions, like branch. This component extends the 16-bit immediate field to 32 full bits to obtain the final address in branch instructions, the extended parts have the same digit as the most significant digit of the input. The Verilog code is as follows:

```verilog
module SignExtend(
Input,
Output
);
//by HU Jiaoyang 517370910140 in VE370 Project 2 in 19FA
//This component extends the 16-bit immediate field to 32 full bits to obtain
   the final address in branch instructions, the extended parts have the same
   digit as the most significant digit of the input
parameter Digits = 32;
parameter ImmeDigits = 16;
input [ImmeDigits-1:0] Input;
```

```
output [Digits-1:0] Output;

assign Output = {{(Digits-ImmeDigits){Input[ImmeDigits-1]}},Input};

endmodule
```

Instruction Memory module is written in `InstructionMemory.v` . The Instruction Memory module has 1 input: `Address`, and 1 output:`Instruction`. The Instruction-Memory module is asynchronous, and is activated only when there is input. This component takes addresses of instructions as input and stores the instruction codes in two dimemsional arrays. The module has a private register, which is the memory storing all the instructions. We set the size of a unit of the instruction codes to be 8-digit, because computer instructions are based on 8-bit storage, and size of the memory 256 bits because the Memory needs to be large enough. Inside the codes is a sample instruction memory provided by TAs. The Verilog code is as follows:

```
 module InstructionMemory(
 Address,
 Instruction
 );
 //by HU Jiaoyang 517370910140 in VE370 Project 2 in 19FA
 //This component takes addresses of instructions as input and output the
    instruction codes
 input [31:0] Address;
 output [31:0] Instruction;
 reg [7:0] Memory[0:255];//This is the whole Memory, storing all the
    instrutions. 8 bit because computer instructions are based on 8-bit
    storage, and 256 bits because the Memory needs to be large enough

 initial $readmemb("Instruction.txt",Memory);//this statement loads up the
    Memory at simulation time 0 starting at the Memory address 1

   end
 assign Instruction = {Memory[Address],Memory[Address+1],Memory[Address+2],
    Memory[Address+3]};//This is straightforward,each slot stores eight
    hexadecimal numbers, and we combine four of them to form a complete
    instruction
 endmodule
```

Adder module is written in `Adder.v` . The Adder module has 2 inputs: `Input1,Input2` to be added, and 1 output:`Output`. The Adder module is asynchronous, and is activated only when there are inputs. This module is used to compute address for program counter to move on to the next instruction, or to obtain the final address in jump and branch instructions. The Verilog code is as follows:

```
`timescale 1ns/100ps
module Adder(
Input1,
Input2,
Output
);
//by HU Jiaoyang 517370910140 in VE370 Project 2 in 19FA
//This is used to compute address for program counter to move on to the next
    instruction, or to obtain the final address in jump and branch
    instructions
parameter Digits = 32;

input [Digits-1:0] Input1,Input2;
output [Digits-1:0] Output;

assign Output = Input1+Input2;

endmodule
```

Mux module is written in `Mux.v` . The Mux module has 2 inputs: `Input1,Input2` to be selected, and 1 output:`Output`. The Mux module is asynchronous, and is activated only when there are inputs. This module is used to output input1 or input2 depending on the selection signal, selecting sources for the ALU; or it can select whether to use the data from Data Memory or from registers. The Verilog code is as follows:

```
module Mux(
Selection,
Input1,
Input2,
Output
);
//by HU Jiaoyang 517370910140 in VE370 Project 2 in 19FA
//this does so much things,it outputs input1 or input2 depending on the
    selection signal, selecting sources for the ALU, selecting whether to use
    the data from DM or from registers...
parameter Digits = 32;//this is subject to change depending on which path this
     component is on
input [Digits-1:0] Input1,Input2;
output reg [Digits-1:0] Output;
input Selection;

```

```
always @ (*) begin
if(Selection == 0) begin Output = Input1; end
if(Selection == 1) begin Output = Input2; end
end
endmodule
```

3.2.2 Top Module Design

After completing all necessary modules, we work on the **Top Module** to implement the whole function of single cycle. The structure is the same as the top level diagram shown as Figure 2. This module is written in `main_single.v`.

```
`include "ShiftLeft2.v"
`include "Mux.v"
`include "Adder.v"
`include "SignExtend.v"
`include "ALUcontrol.v"
`include "Control.v"
`include "RegisterFile.v"
`include "ALU.v"
`include "dataMem.v"
`include "pc.v"
`include "InstructionMemory.v"
module main(clk, reset);
  input clk, reset;
  wire [31:0] pc_out, add1_out, add2_out, mux2_out, mux3_out, mux4_out,
     mux5_out, ins_out, signextend_out, reg_read1, reg_read2, ALU_result,
     dataMem_out,  shift2_out, jumpAddress;
  wire [27:0] shift1_out;
  wire [25:0] ins_except_op;         // instruction code without opcode
  wire [15:0] ins_imm;               // immediate part of instruction
  wire [5:0] ins_opcode, ins_funct; // opcode and funct part of instruction
  wire [4:0] mux1_out, ins_rs, ins_rt, ins_rd;
  wire [3:0] ALUsel;
  wire [1:0] ALUop;
  wire RegDst, Jump, eBranch, nBranch, MemRead, MemtoReg, MemWrite, ALUSrc,
     RegWrite, and1_out, and2_out, or_out, ALU_zero, reset;

  reg [31:0] add_amount;
  //Let the modules initialize on the first clock cycle
  initial begin add_amount = 32'b0; end

  Adder adder1(pc_out, add_amount, add1_out); // add1 for pc+4.
  Adder adder2(add1_out, shift2_out, add2_out); // add2 for branch
  ALU alu(reg_read1, mux2_out, ALUsel, ALU_zero, ALU_result);
  ALUcontrol alucontrol(ALUop, ins_funct, ALUsel);
  dataMem dataMemory(clk, MemRead, MemWrite, ALU_result, reg_read2,
     dataMem_out);
  InstructionMemory insMem(pc_out, ins_out);

  Control control(ins_opcode, RegDst, Jump, eBranch, nBranch, MemRead,
     MemtoReg, ALUop, ALUSrc, MemWrite, RegWrite);

  Mux #(.Digits(5))mux1(RegDst, ins_rt, ins_rd, mux1_out);
  // mux1 for choosing rt or rd
  Mux mux2(ALUSrc, reg_read2, signextend_out, mux2_out);
  // mux2 for choosing ALU_b
  Mux mux3(MemtoReg, ALU_result, dataMem_out, mux3_out);
  // mux3 for choosing write-back data
  Mux mux4(or_out, add1_out, add2_out, mux4_out); // mux4 for branch
  Mux mux5(Jump, mux4_out, jumpAddress, mux5_out); // mux5 for jump

  PC pc (mux5_out, clk, pc_out, reset);
  RegisterFile regFile(clk, RegWrite, ins_rs, ins_rt, mux1_out, mux3_out,
     reg_read1, reg_read2);
  ShiftLeft2 shift2(signextend_out, shift2_out); // shift2 for brach address
  ShiftLeft2 #(.Digits(28)) shift1 ({2'b0, ins_except_op}, shift1_out);
  // shift1 for jump address
  SignExtend #(.Digits(32), .ImmeDigits(16)) ext (.Input(ins_imm), .Output(
     signextend_out));

  and(and1_out, eBranch, ALU_zero);     // and1 for beq
  and(and2_out, nBranch, ~ALU_zero);    // and2 for bne
  or(or_out, and1_out, and2_out);       // or for branch or not

  initial begin #5 add_amount = 32'h4; end

  // instruction decoding
  assign ins_opcode = ins_out[31:26];
  assign ins_rs = ins_out[25:21];
  assign ins_rt = ins_out[20:16];
  assign ins_rd = ins_out[15:11];
  assign ins_funct = ins_out[5:0];
  assign ins_imm = ins_out[15:0];
  assign ins_except_op = ins_out[25:0];
  assign jumpAddress = {add1_out[31:28], shift1_out};

endmodule
```

3.3 Simulation

We write several testbenches to test whether each module works well. After making sure that they all work well, we write the final testbench for **Top Module** in `test_single.v`. We use the MIPS instructions provided by TA to do the simulation.

3.3.1 Simulation Scenario

The content of the Instruction Memory used in this simulation is as follows:

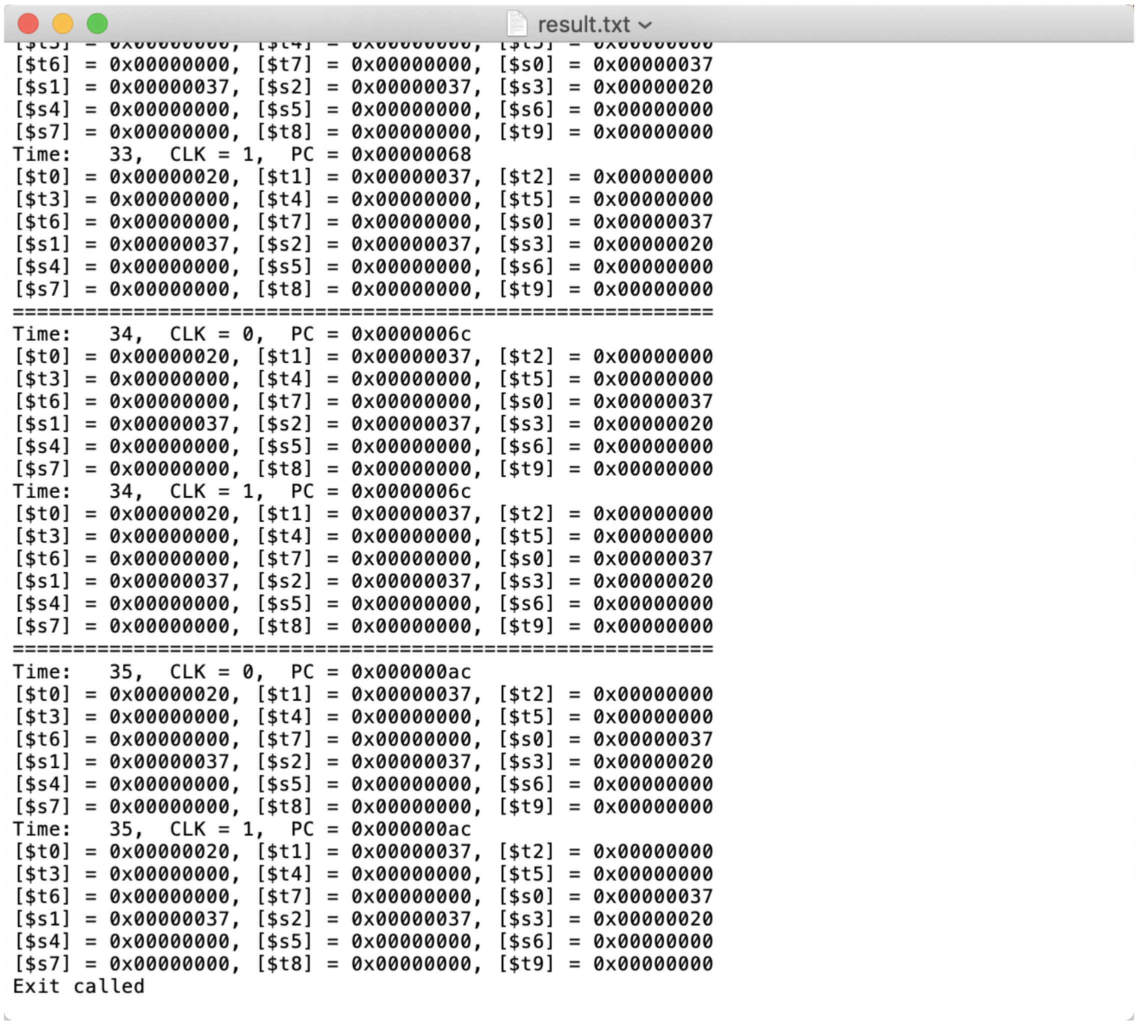
```
memory[0] = 32’b00100000000010000000000000100000; //addi $t0, $zero, 0x20
memory[1] = 32’b00100000000010010000000000110111; //addi $t1, $zero, 0x37
memory[2] = 32’b00000001000010011000000000100100; //and $s0, $t0, $t1
memory[3] = 32’b00000001000010011000000000100101; //or $s0, $t0, $t1
memory[4] = 32’b10101100000100000000000000000100; //sw $s0, 4($zero)
memory[5] = 32’b10101100000010000000000000001000; //sw $t0, 8($zero)
memory[6] = 32’b00000001000010011000100000100000; //add $s1, $t0, $t1
memory[7] = 32’b00000001000010011001000000100010; //sub $s2, $t0, $t1
memory[8] = 32’b00100000000010000000000000100000; //addi $t0, $zero, 0x20
memory[9] = 32’b00100000000010000000000000100000; //addi $t0, $zero, 0x20
memory[10] = 32’b00100000000010000000000000100000; //addi $t0, $zero, 0x20
memory[11] = 32’b00010010001100100000000000010010; //beq $s1, $s2, error0
memory[12] = 32’b10001100000100010000000000000100; //lw $s1, 4($zero)
memory[13]= 32’b00110010001100100000000001001000; //andi $s2, $s1, 0x48
memory[14] = 32’b00100000000010000000000000100000; //addi $t0, $zero, 0x20
memory[15] = 32’b00100000000010000000000000100000; //addi $t0, $zero, 0x20
memory[16] = 32’b00100000000010000000000000100000; //addi $t0, $zero, 0x20
memory[17] =32’b00010010001100100000000000001111; //beq $s1, $s2, error1
memory[18] =32’b10001100000100110000000000001000; //lw $s3, 8($zero)
memory[19] = 32’b00100000000010000000000000100000; //addi $t0, $zero, 0x20
memory[20] = 32’b00100000000010000000000000100000; //addi $t0, $zero, 0x20
memory[21] = 32’b00100000000010000000000000100000; //addi $t0, $zero, 0x20
memory[22] =32’b00010010000100110000000000001101; //beq $s0, $s3, error2
memory[23] =32’b00000010010100011010000000101010; //slt $s4, $s2, $s1 (Last)
memory[24] = 32’b00100000000010000000000000100000; //addi $t0, $zero, 0x20
memory[25] = 32’b00100000000010000000000000100000; //addi $t0, $zero, 0x20
memory[26] = 32’b00100000000010000000000000100000; //addi $t0, $zero, 0x20
memory[27] =32’b00010010100000000000000000001111; //beq $s4, $0, EXIT
memory[28] =32’b00000010001000001001000000100000; //add $s2, $s1, $0
memory[29] =32’b00001000000000000000000000010111; //j Last
memory[30] =32’b00100000000010000000000000000000; //addi $t0, $0, 0(error0)
memory[31] =32’b00100000000010010000000000000000; //addi $t1, $0, 0
memory[32] =32’b00001000000000000000000000111111; //j EXIT
memory[33] =32’b00100000000010000000000000000001; //addi $t0, $0, 1(error1)
memory[34] =32’b00100000000010010000000000000001; //addi $t1, $0, 1
```

```
memory[35] =32'b00001000000000000000000000111111; //j EXIT
memory[36] =32'b00100000000010000000000000000010; //addi $t0, $0, 2(error2)
memory[37] =32'b00100000000010010000000000000010; //addi $t1, $0, 2
memory[38] =32'b00001000000000000000000000111111; //j EXIT
memory[39] =32'b00100000000010000000000000000011; //addi $t0, $0, 3(error3)
memory[40] =32'b00100000000010010000000000000011; //addi $t1, $0, 3
memory[41] =32'b00001000000000000000000000111111; //j EXIT}
```

The code for testbench is shown in the Appendix section.

3.3.2 Textual Result

We use the command `iverilog -o test_single.vvp test_single.v` in terminal to compile the testbech, and then run the `test_single.vvp` file and redirect the result into `result.txt` file. The following shows the textual result before exiting the single cycle. **For the whole simulation result you can refer to section Appendix.**


result.txt

```
[$t6] = 0x00000000, [$t7] = 0x00000000, [$s0] = 0x00000037
[$s1] = 0x00000037, [$s2] = 0x00000037, [$s3] = 0x00000020
[$s4] = 0x00000000, [$s5] = 0x00000000, [$s6] = 0x00000000
[$s7] = 0x00000000, [$t8] = 0x00000000, [$t9] = 0x00000000
Time:   33,  CLK = 1,  PC = 0x00000068
[$t0] = 0x00000020, [$t1] = 0x00000037, [$t2] = 0x00000000
[$t3] = 0x00000000, [$t4] = 0x00000000, [$t5] = 0x00000000
[$t6] = 0x00000000, [$t7] = 0x00000000, [$s0] = 0x00000037
[$s1] = 0x00000037, [$s2] = 0x00000037, [$s3] = 0x00000020
[$s4] = 0x00000000, [$s5] = 0x00000000, [$s6] = 0x00000000
[$s7] = 0x00000000, [$t8] = 0x00000000, [$t9] = 0x00000000
============================================================
Time:   34,  CLK = 0,  PC = 0x0000006c
[$t0] = 0x00000020, [$t1] = 0x00000037, [$t2] = 0x00000000
[$t3] = 0x00000000, [$t4] = 0x00000000, [$t5] = 0x00000000
[$t6] = 0x00000000, [$t7] = 0x00000000, [$s0] = 0x00000037
[$s1] = 0x00000037, [$s2] = 0x00000037, [$s3] = 0x00000020
[$s4] = 0x00000000, [$s5] = 0x00000000, [$s6] = 0x00000000
[$s7] = 0x00000000, [$t8] = 0x00000000, [$t9] = 0x00000000
Time:   34,  CLK = 1,  PC = 0x0000006c
[$t0] = 0x00000020, [$t1] = 0x00000037, [$t2] = 0x00000000
[$t3] = 0x00000000, [$t4] = 0x00000000, [$t5] = 0x00000000
[$t6] = 0x00000000, [$t7] = 0x00000000, [$s0] = 0x00000037
[$s1] = 0x00000037, [$s2] = 0x00000037, [$s3] = 0x00000020
[$s4] = 0x00000000, [$s5] = 0x00000000, [$s6] = 0x00000000
[$s7] = 0x00000000, [$t8] = 0x00000000, [$t9] = 0x00000000
============================================================
Time:   35,  CLK = 0,  PC = 0x000000ac
[$t0] = 0x00000020, [$t1] = 0x00000037, [$t2] = 0x00000000
[$t3] = 0x00000000, [$t4] = 0x00000000, [$t5] = 0x00000000
[$t6] = 0x00000000, [$t7] = 0x00000000, [$s0] = 0x00000037
[$s1] = 0x00000037, [$s2] = 0x00000037, [$s3] = 0x00000020
[$s4] = 0x00000000, [$s5] = 0x00000000, [$s6] = 0x00000000
[$s7] = 0x00000000, [$t8] = 0x00000000, [$t9] = 0x00000000
Time:   35,  CLK = 1,  PC = 0x000000ac
[$t0] = 0x00000020, [$t1] = 0x00000037, [$t2] = 0x00000000
[$t3] = 0x00000000, [$t4] = 0x00000000, [$t5] = 0x00000000
[$t6] = 0x00000000, [$t7] = 0x00000000, [$s0] = 0x00000037
[$s1] = 0x00000037, [$s2] = 0x00000037, [$s3] = 0x00000020
[$s4] = 0x00000000, [$s5] = 0x00000000, [$s6] = 0x00000000
[$s7] = 0x00000000, [$t8] = 0x00000000, [$t9] = 0x00000000
Exit called
```

Figure 3: The simulation result before exit.

We can see that the result shown in Fig 3 accords the theoretical result. Our implementation of single cycle is successful.

4 Pipelined Implementation

4.1 Top Level Diagram

The top level diagram is as follows:

We provide a zoomed in image of each module below:

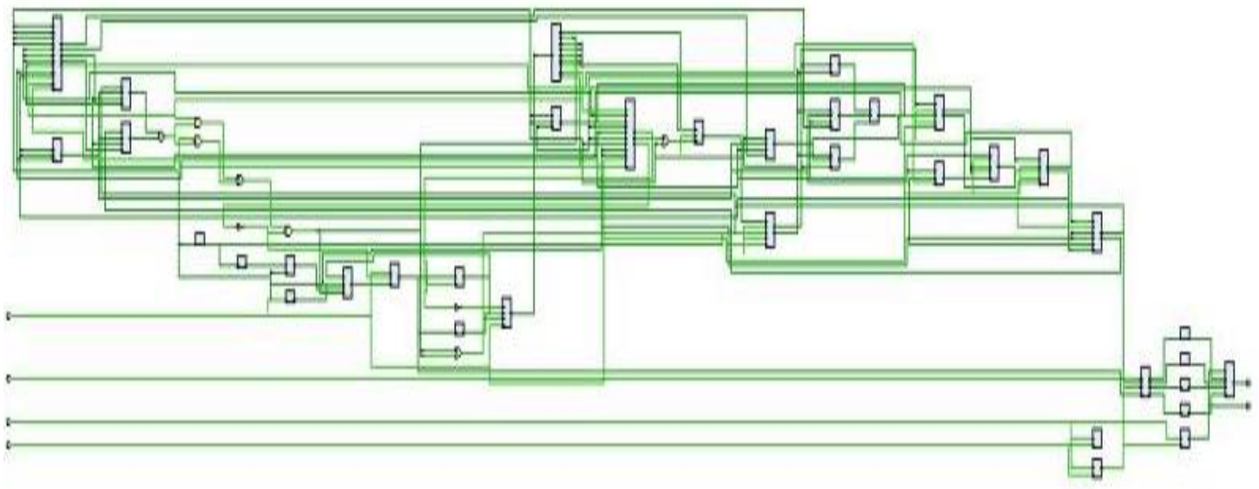


Figure 4: Top level diagram of pipelined implementation

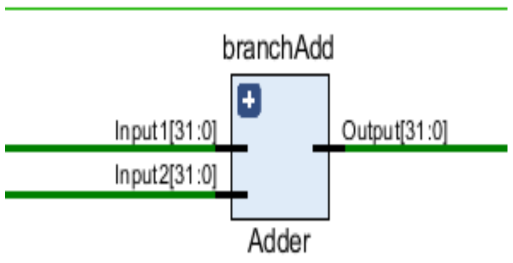


Figure 5: Adder

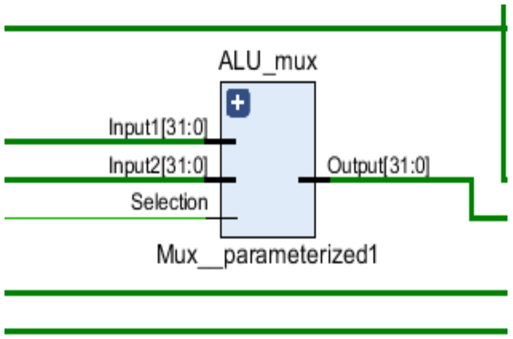


Figure 6: ALU mux

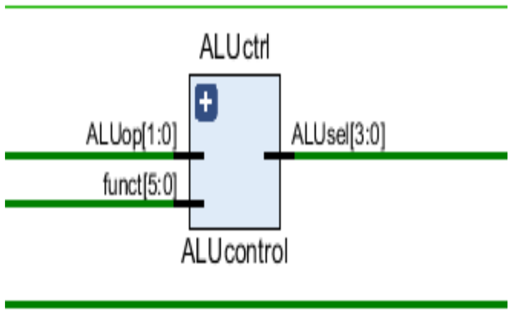


Figure 7: ALU Control

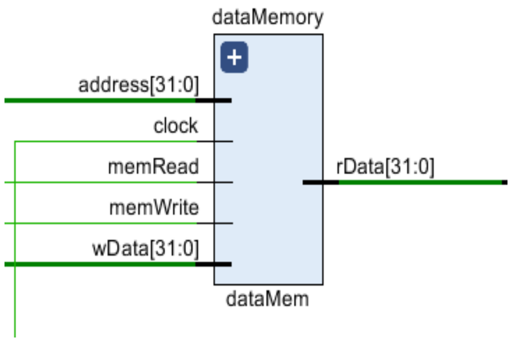


Figure 8: Data Memory

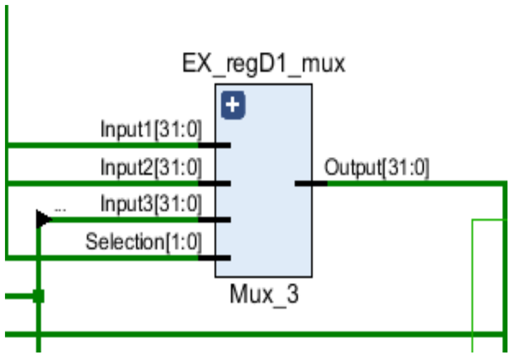


Figure 9: EX_regD1 mux

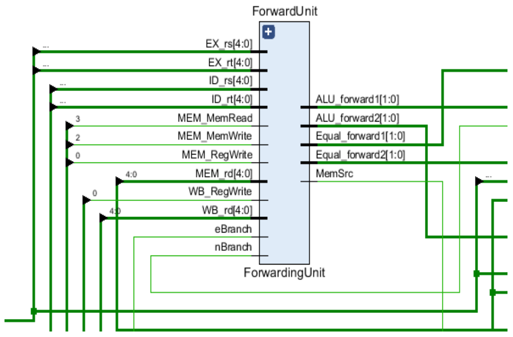


Figure 10: Forwarding Unit

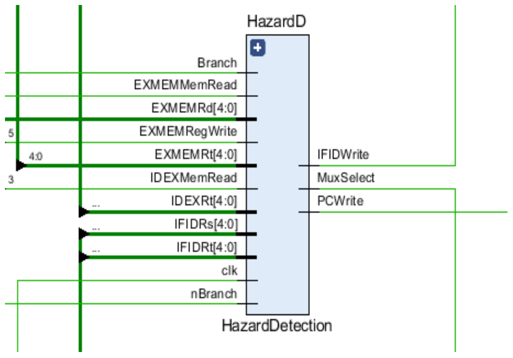


Figure 11: HazardDetection

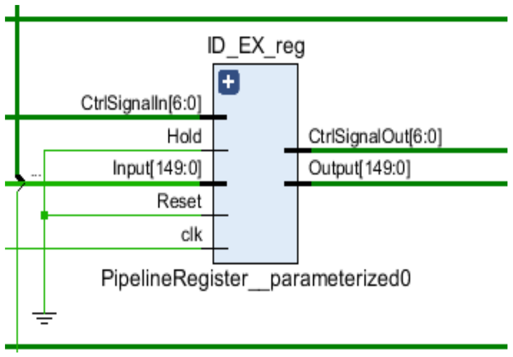


Figure 12: ID_EX pipelined register

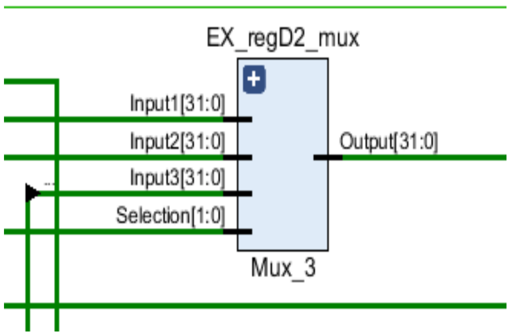


Figure 13: 3 input mux

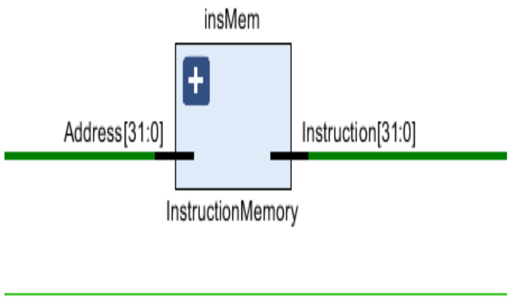


Figure 14: Instruction Memory

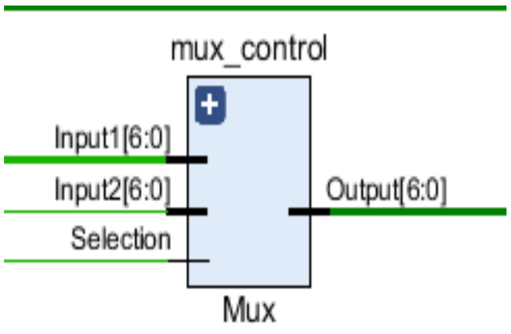


Figure 15: mux_control module

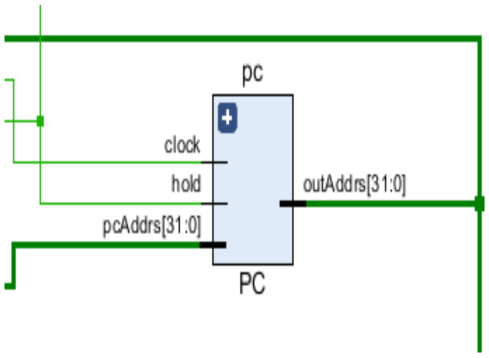


Figure 16: PC

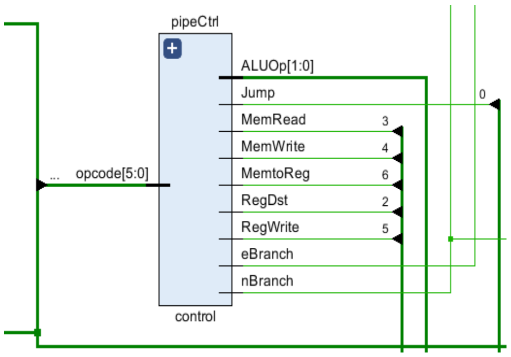


Figure 17: PipeCtrl module

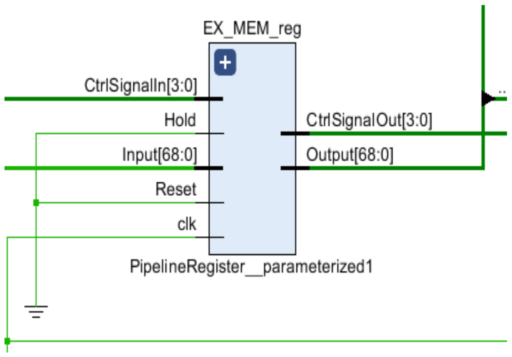


Figure 18: Pipeline Register

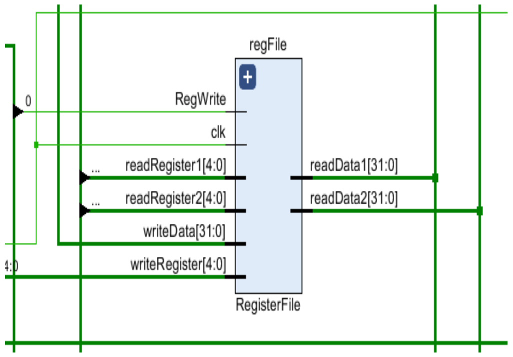


Figure 19: RegFile moduke (register file)

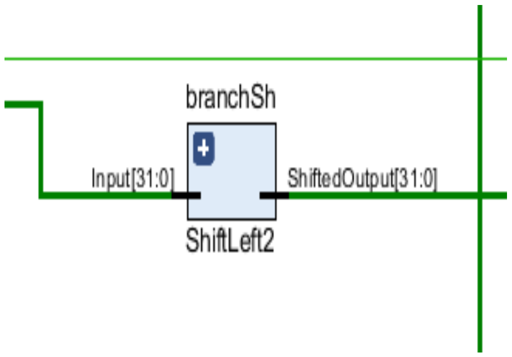


Figure 20: ShiftLeft2 module

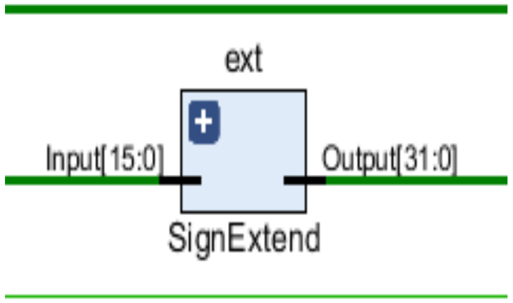


Figure 21: SignExtend module

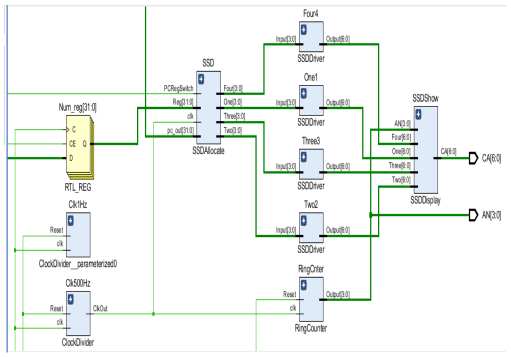


Figure 22: SSD module

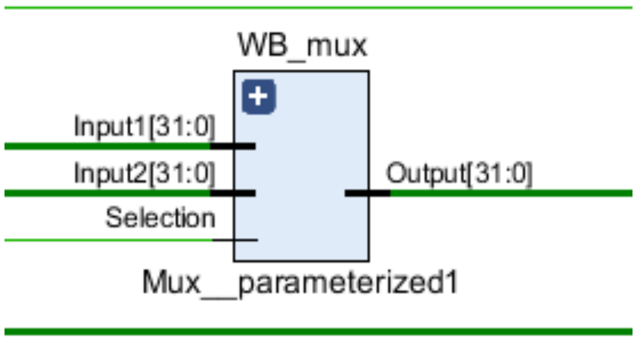


Figure 23: WB_Mux module

4.2 Modules Design

There are 15 modules used in the pipelined design. The modules are: PC, ALU, ALU Control, Control, Adder, Data Memory, Instruction Memory, Mux(2 inputs and 3 inputs mux), Shift Left, Register File, Sign Extender, Forwarding Unit, Hazard Detection and Pipeline Register. The modules are put together in the Main module, which implements the pipelined MIPS computer. The implementation supports the following instructions: load word(lw), store word(sw), add, addi, sub, and, andi, or, slt, branch equal(beq), branch not equal(bne) and jump(j). This implementation takes on the **assume branch not taken** approach.

4.2.1 Components Design

PC module is used to generate the PC address. This module is written in the `pcPipeline.v` file. This PC module has an additional input `hold` from the one used in the single cycle implementation. `Hold` allows the PC to hold its PC value, such that it will only update the PC output if `Hold == 1`. The module is as follows:

```
module PC (pcAddrs, clock, outAddrs, hold);
  input [31:0] pcAddrs; // Input address of PC
  input clock,hold;     // Hold tells the PC whether to hold the address
                        //or output the input address
  output [31:0] outAddrs;   // The output address of the PC

  reg[31:0] pcReg;


  initial begin
    pcReg <= 32'b0;
  end
  assign outAddrs = pcReg;

  always @(negedge clock) begin //PC updates at negedge clock
    if(hold == 1'b0) begin
        pcReg <= pcAddrs;
    end
  end

endmodule


endmodule
```

The ALU, ALU control, Adder, Data Memory, Instruction Memory, Mux (2 inputs), Shift Left, Register FIle and Sign Extender module is the same as the one used in the single cycle implementation. Please refer to the single cycle implementation Components Design section.

Mux (3 inputs) is just an upgrade from the 2 input mux. It is written in the `Mux_3.v` file. It takes three inputs (`Input1, Input2, Input3`) and a select signal (`Selection`). When `Selection` is 2'b00, the mux will output `Input1`. When `Selection` is 2'b01, the mux will output `Input2`. When `Selection` is 2'b10, the mux will output `Input3`. Else, it will output `Input1` by default. Considering there are only 3 inputs, the `Select` should never be 2'b11. The Verilog code is as follows:

```

module Mux_3(
Selection,
Input1,
Input2,
Input3,
Output
);
//based on HU Jiaoyang 517370910140 2 input Mux module in VE370 Project 2 in
    19FA

parameter Digits = 32;//this is subject to change depending on which path this
     component is on
  input [Digits-1:0] Input1,Input2,Input3;
output reg [Digits-1:0] Output;
  input [1:0] Selection;

always @ (*) begin
  if(Selection == 2'b00) begin Output <= Input1; end
  else if(Selection == 2'b01) begin Output <= Input2; end
  else if(Selection == 2'b10) begin Output <= Input3; end
  else begin Output <= Input1; end
end
endmodule
```

4.2.2 Data Hazards Handling

Hazard Detection Unit is written in `HazardDetection.v`, mainly to deal with Load-use Hazard andBranch Hazard. The HazardDetection module has 11 inputs: `clk`, `IDEXMemRead`, `Branch`, `nBranch`, `EXMEMMemRead`, `EXMEMRegWrite`, `IFIDRt`, `IFIDRs`,

`IDEXRt`, `EXMEMRt`, `EXMEMRd`, and 3 output: `PCWrite`, `IFIDWrite`, `MuxSelect`. The module is synchronous, and works in ID stage of the processor. The module looks for data hazards that cannot be solved with forwarding(unlike EX/MEM hazards), and it inserts stalls into the processor to prevent the hazards from happening.
In the case of `Load-use Hazard`, the result of lw is obtained in the MEM stage of the instruction, however, the next instruction uses its result as either Rs or Rt, and require it in the EX stage. Therefore, the processor has to delay for one cycle before forwarding the result of lw from MEM/WB pipeline register to ID/EX pipeline register.
The way to detect Load-use Hazard is to first confirm that the first instruction is lw, which is to check IDEXMemRead. Next is to check that the result of lw is required in the next instruction, which is to check whether the Rt in the ID/EX pipeline register(executing lw instruction), is the same as either Rs or Rt in the next instruction's IF/ID pipeline register. Next is confirm that the next instruction is not branch instruction, because otherwise it would be a branch hazard, this is done by checking branch signal that we placed in the processor. After all is confirmed, we hold the processor by making PCWrite and IF/IDWrite 0, and select mux result to be 1, and flush the pipeline register.
A typical example of triggering the hazard looks like this:

```
addi $t0,$t0,6
addi $t1,$t1,16//address for $s0
sw $t0,$t1($zero)//store 6 in $s0
addi $t0,$zero,0
addi $t0,$zero,0//stalls
lw $t2,$t1($zero)//loading 6 from $s0 and place into $t2
add $t3,$t2,$t1//adding 6 in $t2, 16 in $t1 and place result into $t3
```

Translating these codes into machine code we get:

```
00100001000010000000000000000110//addi $t0,$t0,6
00100001001010010000000000010000//addi $t1,$t1,16
10101101001010000000000000000000//sw $t0,0($t1)
00100001000010000000000000000000//addi $t0,$t0,0
00100001000010000000000000000000//addi $t0,$t0,0
10001101001010100000000000000000//lw $t2,0($t1)
00000001001010100101100000100000//add $t3,$t2,$t1
```

It should be staightfoward that the result will be:
[\$t0] = 0,[\$t1] = 16,[\$t2] = 6,[\$t3] = 22. And after simulation, we see in the TCL Concole:

```
Exit called, executing final instructions
Final register state:
And after
[$t0] = 0x00000006, [$t1] = 0x00000010, [$t2] = 0x00000006
[$t3] = 0x00000016, [$t4] = 0x00000000, [$t5] = 0x00000000
```

We can see that the results are the same with what we expected, which indicate that we have successful solved the Load-use Hazard.
In the case of `Branch Hazards`, there are a total of three instance that we need to consider.
Firstly, we need to consider the case in which lw is immediately followed by a branch instruction, like so:

```
//lw $1,addr
//beq $1,$0,target(Here we need to insert two bubbles)
```

Since a branch instruction requires its input in ID stage, and we can only obtain the result of lw in its MEM stage, we have to insert two stalls before the forwarding unit can send the result from the MEM/WB pipeline register of lw to the IF/ID register of branch.
We can identify this type of branch hazard by firstly confirming that the first signal is lw, which is checking ID/EXMemRead. After that we check the branch signal that it's branch, and then compare the Rs,Rt in branch with Rt in lw to confirm whether there is a hazard.
Since we need to insert two bubbles, Hazard Detection Unit has a register called DoubleBubble. The register is changed to 1 when this instance of the branch hazard occurs. And in every clock cycle, the unit checks the DoubleBubble, and insert a second bubble when the value stored inside the register is 1.
The second instance is very similar to the first, in which an irrelevant instruction is inserted between lw and branch instruction, and we only need to insert one bubble.
Here's an instance of this type of branch hazard.

```
//lw $1,addr
//addi $2,$2,0
//beq $1,$0,target(Here we need to insert two bubbles)
```

The third instance is an R-type instruction followed by a branch instruction. The ID stage of branch instruction needs the output of the first instruction, which is not avaibable until the EX stage of the first instruction. Therefore, the processor inserts one bubble, and after the bubble the forwarding unit can forward the result from EX/MEM pipeline register to ID stage.
To check for this type of hazard, we need to make sure the first instruciton is not lw, which means checking that EXMEMRead is 0 and EXMEMRegWrire is 1. Next is checking that the second instruction is branch, which is done by checking the branch signal that we placed inside the processor. Finally, to confirm the hazard, we compare the Rd of the first instruction with the Rs and Rt of the branch instruction. With the above confirmed, we can safely say that there is the branch hazard, and we flush the pipeline registers and hold PC and IF/ID pipeline.
Here's an instance of this type of branch hazard.

```
//add $4,$5,$6
//beq $1,$4,target(Here we need to insert one bubble)
```

The Verilog code of **Hazard Detection Unit** is as follows:

```
`timescale 1ns / 100ps

module HazardDetection(
clk,
IDEXRt,
IFIDRt,
IFIDRs,
IFIDWrite,
PCWrite,
MuxSelect,
IDEXMemRead,
Branch,
nBranch,
EXMEMMemRead,
EXMEMRt,
EXMEMRd,
EXMEMRegWrite
);
//by HU Jiaoyang 517370910140 in VE370 19FA Project2
//we are checking data hazards that cannot be solved with forwarding(not like
    some data hazards)
//Therefore, this module adds bubbles who stalling the processor is the only
    solution
input [4:0] IFIDRt,IFIDRs,IDEXRt,EXMEMRt,EXMEMRd;
input clk,IDEXMemRead,Branch,nBranch,EXMEMMemRead,EXMEMRegWrite;
output reg PCWrite,IFIDWrite,MuxSelect;

reg DoubleBubble;
//We start by initializing the variables
initial begin
PCWrite <= 1'b1;
IFIDWrite <= 1'b1;
MuxSelect <= 1'b0;
end

//Sometimes we need to insert double bubble, so in this cycle we might need to
     continue inserting bubbles following the command from last cycle
always @ (posedge clk) begin
if(DoubleBubble == 1'b1) begin
DoubleBubble <= 1'b0;
```

```verilog
PCWrite <= 1'b0;
IFIDWrite <= 1'b0;
MuxSelect <= 1'b1;
end
//Insert another bubble, reset the DoubleBubble variable

//Now after dealing with DoubleBubble issue, we now initialize the variable
    again before detecting hazards
else begin
DoubleBubble <= 1'b0;
PCWrite <= 1'b1;
IFIDWrite <= 1'b1;
MuxSelect <= 1'b0;
//Here we consider a Load-use hazard that also requires one stall
//Hazard 01
//lw $1,addr
//nop
//add $0,$1,$4(Here we need to insert one bubble)
//Here we just need to stall one cycle so the EX of add comes after MEM of lw
if (IDEXMemRead == 1'b1) begin
if((IFIDRs == IDEXRt || IFIDRt == IDEXRt) && !(Branch || nBranch))begin
DoubleBubble <= 1'b0;
PCWrite <= 1'b0;
IFIDWrite <= 1'b0;
MuxSelect <= 1'b1;
end
end
//Here we consider three instances of branch hazard
//Hazard 02
//lw $1,addr
//beq stalled
//beq stalled
//beq $1,$0,target(Here we need to insert two bubbles)
//beq reads its rt in ID stage, so we need to: 1. make sure that it is branch
    2. make sure ID of beq comes after the MEM of lw
if (IDEXMemRead == 1'b1) begin
if((IFIDRs == IDEXRt || IFIDRt == IDEXRt)&&(Branch || nBranch))begin
DoubleBubble <= 1'b1;
PCWrite <= 1'b0;
IFIDWrite <= 1'b0;
MuxSelect <= 1'b1;
end
end

//Hazard 03
```

```
//lw $1, addr
//...
//beq stalled
//beq $1,$4,target(Here we need to insert one bubble)
if(EXMEMMemRead && (IFIDRs == EXMEMRt || IFIDRt == EXMEMRt)&&(Branch ||
    nBranch))begin
DoubleBubble <= 1'b0;
PCWrite <= 1'b0;
IFIDWrite <= 1'b0;
MuxSelect <= 1'b1;
end

//Hazard 04
//add $4,$5,$6
//beq stalled
//beq $1,$4,target(Here we need to insert one bubble)
if (!EXMEMMemRead && EXMEMRegWrite && (Branch || nBranch) && (EXMEMRd ==
    IFIDRs || EXMEMRd == IFIDRt)) begin
DoubleBubble <= 1'b0;
PCWrite<=1'b0;
IFIDWrite<=1'b0;
MuxSelect <=1'b1;
end
end
end
endmodule
```

Forwarding Unit Module is designed to resolve the problem of data dependency. It is written in `ForwardingUnit.v` file, and is mainly designed to solve 5 kinds of data dependency, which are MEM/EX hazard, WB/EX hazard, `beq/bne` with R/I type instruction, `beq/bne` with R/I type and `lw` instruction, and instruction that writes into register file following with `sw`.

In this project, we just use the function of solving 3 kinds of data dependency, MEM/EX hazard, WB/EX hazard and instruction that writes into register file following with sw. The example for MEM/EX hazard is:

```
add $4,$5,$6
add $5,$4,$0 // here $4 is in data dependency (MEM/EX)
```

The example for WB/EX hazard is:

```
add $4,$5,$6
addi $6, $5, 0
add $5,$4,$0 // here $4 is in data dependency (WB/EX)
```

In our design, the calculated result of register `$4` from `add $4,$5,$6` in WB stage will be directly forward to one of the input of register file `RegRead1` or `RegRead2` in EX stage, depending on where we detect data hazard, at `rd` or `rs`. Especially our Forwarding Unit can solve WB/EX hazard only when MEM/EX hazard doesn't happen. Hence in the example below, forwarding for WB/EX hazard won't be triggered.

```
add $4,$5,$6
addi $4, $5, 0
add $5,$4,$0 // here $4 is in data dependency (MEM/EX)
```

The example for the hazard where instruction writing into register file following with sw:

```
add $4,$5,$6
sw $5,4($4) // here $4 is in data dependency
```

In our design, the result coming from `add` in WB stage will be directly forward to the input of memory file in MEM stage. The following is the Verilog code of **Forwarding Unit**:

```
module ForwardingUnit(
       ALU_forward1,
       ALU_forward2,
       Equal_forward1,
       Equal_forward2,
       MemSrc,
       MEM_rd,
       WB_rd,
       EX_rs,
       EX_rt,
       MEM_RegWrite,
       WB_RegWrite,
       MEM_MemRead,
       MEM_MemWrite,
       eBranch,
       nBranch,
       ID_rs,
       ID_rt
);

    input [4:0] MEM_rd, WB_rd, EX_rs, EX_rt, ID_rs, ID_rt;
    input MEM_RegWrite, WB_RegWrite, MEM_MemRead, MEM_MemWrite, eBranch,
       nBranch;
    output reg [1:0] ALU_forward1, ALU_forward2, Equal_forward1,
       Equal_forward2;
    output reg MemSrc;

    //initialize control signals
    initial begin
        ALU_forward1 <= 2'b00;
        ALU_forward2 <= 2'b00;
        Equal_forward1 <= 2'b00;
        Equal_forward2 <= 2'b00;
        MemSrc <= 1'b0;
    end

    always @ (*) begin
    //default value
    ALU_forward1 <= 2'b00;
        ALU_forward2 <= 2'b00;
        Equal_forward1 <= 2'b00;
        Equal_forward2 <= 2'b00;
        MemSrc <= 1'b0;
    if (MEM_RegWrite == 1'b1) begin // For MEM hazard
        if (MEM_rd == EX_rs)  begin ALU_forward1 <= 2'b10;
            $display("Forwarding␣unit::␣MEM/EX␣hazard,␣ForwardA␣=␣10");
        end
        if (MEM_rd == EX_rt) begin ALU_forward2 <= 2'b10;
            $display("Forwarding␣unit::␣MEM/EX␣hazard,␣ForwardB␣=␣10");
        end
    end
    if (WB_RegWrite == 1'b1) begin // For WB hazard
        if (WB_rd == EX_rs && !(MEM_RegWrite == 1'b1 && MEM_rd == EX_rs))
           begin ALU_forward1 <= 2'b01;
        $display("Forwarding␣unit::␣WB/EX␣hazard,␣ForwardA␣=␣01"); end
        if (WB_rd == EX_rt && !(MEM_RegWrite == 1'b1 && MEM_rd == EX_rt))
           begin  ALU_forward2 <= 2'b01;
        $display("Forwarding␣unit::␣WB/EX␣hazard,␣ForwardA␣=␣01"); end
    end
    if (WB_RegWrite == 1'b1 && MEM_MemWrite == 1'b1) begin
    // For sw immediately after operations that write in register file
        if (WB_rd == MEM_rd) begin MemSrc <= 1'b1;
    $display("Forwarding␣unit::␣WB/sw␣hazard,␣Forward_load␣=␣1"); end
    end
    if (WB_RegWrite == 1'b1 && (eBranch || nBranch)) begin
    // For branch and lw/R/I type operation


        if (WB_rd == ID_rs) begin  Equal_forward1 <= 2'b01;
        $display("Forwarding␣unit::␣WB/branch␣hazard,␣Forward_branch1␣=␣01");
           end

        if (WB_rd == ID_rt) begin Equal_forward2 <= 2'b01;
        $display("Forwarding␣unit::␣WB/branch␣hazard,␣Forward_branch2␣=␣01");
            end

    end
    if (MEM_RegWrite == 1'b1 && (eBranch || nBranch) && MEM_MemRead == 1'b0)
        begin
    // For branch and R/I type operation (no lw)
        if (MEM_rd == ID_rs) begin Equal_forward1 <= 2'b10;
        $display("Forwarding␣unit::␣MEM/branch␣hazard,␣Forward_branch1␣=␣10");
             end
        if (MEM_rd == ID_rt) begin Equal_forward2 <= 2'b10;
 $display("Forwarding␣unit::␣MEM/branch␣hazard,␣Forward_branch1␣=␣01"); end
    end


    end
endmodule
```

4.2.3 Top Module Design

The top module of the pipelined MIPS computer is written in the `main_pipeline.v` file. In this implementation, the assume branch not taken approach was used, which means the pc will always increase by 4 and if a branch is taken (decided in the ID stage), the IF/ID pipeline registered will be flushed (all data set to 0 and all controls signals set to 0). The instruction supplied to the PC to be executed next will the be the target address of the branch. In the IF stage, the main task being executed is choosing the correct PC address and fetching the instruction at that address from the Instruction Memory. A 3 input mux, `branch_mux` outputs either PC + 4 (`PC_add4`), target address of the jump instruction (`jump_addrs`) or target address of branch instruction (`branch_addrs`). The output of the mux is called the `pc_addrs`, which will be the address sent to the Instruction Memory. The Instruction Memory, `insMem`, fetches the instruction from the address `pc_addrs` and outputs the instruction (`insmem_out`). The IF/ID pipeline register takes in the following inputs from the IF stage: `PC_add4, insmem_out` and outputs them as `ID_pc4, ins`.

The ID stage decodes the instruction in the IF/ID register into `ins_op` (ins[31:26), `ins_rs` (ins[25:21], `ins_rt` (ins[20:16]), `ins_rd`(ins[15:11], `ins_funct` (ins[5:0]), `ins_imm` (ins[15:0]), `ins_jump` (ins[25:0]). The usage of each signal will be further explained. A control module, `pipeCtrl` is available in the ID stage, which takes it `ins_op` as input and outputs 9 control signals namely `ID_regDst, jump, Branch, nBranch, ID_memRead, ID_MemtoReg, ALUop, ID_memWrite, ID_regWrite`.There is also a register file, `regFile`, in the ID stage. `in_rs` will always be connected to the `readRegister1` input and `in_rt` will always be connected to the `readRegister2` input of the register file. When the `RegWrite` input of the register file is 1, it will write to the register. This input comes from the WB stage (`WB_regWrite`). The `writeData` and `writeRegister` inputs also come from the WB stage, which are `WB_data` and `WB_dest` respectively. Note that pipeline and pc is updated during negative edge of clock while information is written to register during the positive edge of clock.

The register file will then output the data from the register, namely `reg_D1` and `reg_D2`. The decision to branch is also made in this stage. It compares the two registers in the beq instruction with a comparator and outputs `equal = 1` if the data are equal, 0 otherwise. The forwarding and any possible hazards related to the input data and output data are dealt with by the Hazard Detection Unit and Forwarding Unit. If the instruction is a beq instruction, which means the `Branch` signal will be one, `branch_eq` = 1 if equal = 1. For bne instruction, `branch_neq` will be 1 if equal = 0. This can be done by simple or & and logic gates. The wire that decides whether if a branch equal or branch not equal instruction will occur is the `final_branch` wire. A jump instruction is always executed. The branch and jump target addresses will be sent as the inputs of `branch_mux` in IF stage and the {`final_branch, jump`} signals will be the selection signal of the mux.

In the EX stage, the ALU executes the execution required by each instructions. The first input of the ALU (`ALU_in1`) comes from a mux which choose whether to output the value previously read from register (`EX_reg_D1`) or forwarded inputs from MEM or WB stage. For the second input of the ALU (`ALU_in2`), a mux similar to the one for the first input is also present. However, the output of this mux (`D2_m1_out)` serves as an input to another mux which chooses between this input or `EX_imm`, which is the extended immediate number. The control signals of the ALU comes from the ALU Control module, which is also present in the EX stage.

In the MEM stage, the main operation is done by the Data Memory module. If `MEM_memRead == 1` data is fetched from the memory at the address supplied by `MEM_ALUres`, which is the ALU result previously obtained in the EX stage. If (MEM_memWrite == 1), data is stored to that address. A mux is present to choose the write data. The data might come from either the output of the register file previously obtained in the ID stage and and stored in the ID/EX pipeline register or the data forwarded from the WB stage.

In the WB stage, the data is written back to the register if `WB_regWrite.`== 1. A mux is present to choose the data to be written back. The data might either be from the output of the Data Memory in the previous stage, which is stored in the MEM/WB pipeline register as `WB_MEM_data` , or from ALU result previously obtain in the EX stage, which is stored in the pipeline register as `WB_reg_data`. The signal controlling the mux is `WB_MemtoReg`.

The Verilog code of the main module is as follows:

```verilog

//2019 Fall VE370 project 2 Group 32, main module by: Gracia Stefani
    517370990022
//This module assumes branch is not taken
`include "ShiftLeft2.v"
`include "Mux.v"
`include "Adder.v"
`include "SignExtend.v"
`include "InstructionMemory.v"
`include "ALU.v"
`include "pcPipeline.v"
`include "dataMem.v"
`include "Control.v"
`include "ALUcontrol.v"
`include "RegisterFile.v"
`include "PipelineRegister.v"
`include "HazardDetection.v"
`include "ForwardingUnit.v"
`include "Mux_3.v"

module main(clk);

  input clk;

  //declaration of wires
  wire [31:0] pc_out, insmem_out, ins, reg_D1, reg_D2, comp_1, comp_2,
      ins_imm_ext,ins_sh_emm, ID_pc4,jump_addrs, PC_add4, pc_addrs,branch_addrs
      ,EX_imm, EX_reg_D2, EX_reg_D1,ALU_in1, ALU_in2, D2_m1_out, ALU_res,
      MEM_D1, MEM_ALUres, MEM_WD,MEM_readData, WB_MEM_data, WB_reg_data,
      WB_data , EX_pc4, branch_mux_out;
  wire [4:0] ins_rs, ins_rt, ins_rd,EX_rs, EX_rt, EX_rd, EX_dest, MEM_dest,
      WB_dest;
  wire [5:0] ins_op, ins_funct, EX_funct;
  wire [25:0] ins_jump;
  wire [15:0] ins_imm;
  wire [27:0] ins_sh_out;
  wire IF_hold, PC_hold, Bubble,  MEM_memRead,MEM_MemtoReg, MEM_memWrite,
      go_branch, jump, IF_flush, EX_flush, branch_neq, branch_eq, nBranch,
      Branch, WB_MemtoReg, WB_regWrite, Forward_load,ID_regDst, ID_memWrite,
      ID_MemtoReg, ID_memRead, ID_regWrite, equal,  EX_memWrite, EX_MemtoReg,
      EX_regDst, EX_memRead, EX_regWrite,zero, BubCtrl, final_branch,nothing1,
      nothing2, BubCheck, or_branch, or_JB, branch_xnor, branch_check,
      flush_check, jump_check, IF_Write, PC_Write; //BubCheck sets the value of
       BubCtrl to 0 if EX_flush is x
  wire [1:0] ALUop, EX_ALUop, ForwardA, ForwardB, Forward_branch1,
      Forward_branch2;
  wire [3:0] ALU_control;
  wire [6:0] control_out;

  //declaration of modules

  Adder pcAdd (.Input1(pc_out), .Input2(32'h4), .Output(PC_add4));
  Adder branchAdd (.Input1(ins_sh_emm), .Input2(ID_pc4), .Output(branch_addrs)
      );
  ALU MEM_ALU (.in1(ALU_in1), .in2(ALU_in2), .control(ALU_control), .zero(zero
      ), .aluRes(ALU_res));
  ALUcontrol ALUctrl(.ALUop(EX_ALUop), .funct(EX_funct), .ALUsel(ALU_control))
      ;
  dataMem dataMemory (.clock(clk), .memRead(MEM_memRead), .memWrite(
      MEM_memWrite), .address(MEM_ALUres), .wData(MEM_WD), .rData(MEM_readData)
      );
  InstructionMemory insMem (.Address(pc_out), .Instruction(insmem_out));

  control pipeCtrl (.opcode(ins_op), .RegDst(ID_regDst), .Jump(jump), .eBranch
      (Branch), .nBranch(nBranch), .MemRead(ID_memRead), .MemtoReg(ID_MemtoReg)
      , .ALUOp(ALUop), .MemWrite(ID_memWrite), .RegWrite(ID_regWrite));

  Mux #(.Digits(7)) mux_control (.Selection(BubCtrl), .Input1({ID_MemtoReg,
      ID_regWrite,ID_memWrite, ID_memRead,ID_regDst, ALUop}), .Input2(7'b0), .
      Output(control_out));
  Mux #(.Digits(5)) mux_dest (.Selection(EX_regDst),.Input1(EX_rt), .Input2(
      EX_rd), .Output(EX_dest));
  Mux MEM_mux (.Selection(Forward_load), .Input1(MEM_D1), .Input2(WB_data), .
      Output(MEM_WD));
  Mux WB_mux (.Selection(WB_MemtoReg), .Input1(WB_reg_data), .Input2(
      WB_MEM_data), .Output(WB_data));
  Mux ALU_mux (.Selection(EX_regDst), .Input1(EX_imm), .Input2(D2_m1_out), .
      Output(ALU_in2));

  Mux_3 branch_mux (.Selection({branch_check,jump_check}), .Input1(PC_add4), .
      Input2(jump_addrs), .Input3(branch_addrs), .Output(pc_addrs));
  //Mux pc_mux (.Selection(flush_check), .Input1(branch_mux_out), .Input2(
      ID_pc4), .Output(pc_addrs));
  Mux_3 regD1_mux (.Selection(Forward_branch1), .Input1(reg_D1), .Input3(
      MEM_ALUres), .Input2(WB_data), .Output(comp_1));
  Mux_3 regD2_mux (.Selection(Forward_branch2), .Input1(reg_D2), .Input3(
      MEM_ALUres), .Input2(WB_data), .Output(comp_2));
  Mux_3 EX_regD1_mux (.Selection(ForwardA), .Input1(EX_reg_D1), .Input3(
      MEM_ALUres), .Input2(WB_data), .Output(ALU_in1));
  Mux_3 EX_regD2_mux (.Selection(ForwardB), .Input1(EX_reg_D2), .Input3(
      MEM_ALUres), .Input2(WB_data), .Output(D2_m1_out));

  PC pc (.pcAddrs(pc_addrs), .clock(clk), .outAddrs(pc_out),.hold(PC_hold));
  RegisterFile regFile (.clk(clk), .RegWrite(WB_regWrite), .readRegister1(
      ins_rs), .readRegister2(ins_rt), .writeRegister(WB_dest), .writeData(
      WB_data), .readData1(reg_D1), .readData2(reg_D2));

  ShiftLeft2 branchSh (.Input(ins_imm_ext), .ShiftedOutput(ins_sh_emm));
  ShiftLeft2 #(.Digits(28)) jumpSh (.Input({2'b0,ins_jump}), .ShiftedOutput(
      ins_sh_out));
  SignExtend ext (.Input(ins_imm), .Output(ins_imm_ext));

  and (branch_eq, Branch, equal);
  and (branch_neq, nBranch, ~equal);
  or (go_branch, branch_eq, branch_neq);
  and (final_branch, go_branch, ~PC_hold);
  or (IF_flush, final_branch, jump);
  or(BubCtrl, Bubble, BubCheck); //if the IF/ID register is to be flushed,
      control signals should also be flushed, pass this flush control signal to
       EX stage since decision is available to be passed before the instruction
       to be flushed move to ID stage

  PipelineRegister #(.Digits(64), .CtrlDigits(1)) IF_ID_reg (.clk(clk), .Input
      ({PC_add4, insmem_out}), .Output({ID_pc4,ins}), .CtrlSignalIn(nothing1),
      .CtrlSignalOut(nothing2) ,.Hold(IF_hold), .Reset(IF_flush));

  PipelineRegister #(.Digits(150), .CtrlDigits(7)) ID_EX_reg (.clk(clk), .
      Input({reg_D1,reg_D2, ins_rs, ins_rt, ins_rd, ins_funct, ins_imm_ext,
      IF_flush, ID_pc4}), .Output({EX_reg_D1, EX_reg_D2, EX_rs, EX_rt, EX_rd,
      EX_funct, EX_imm, EX_flush, EX_pc4}), .CtrlSignalIn(control_out), .
      CtrlSignalOut({EX_MemtoReg, EX_regWrite,EX_memWrite, EX_memRead,EX_regDst
      , EX_ALUop}), .Hold(1'b0), .Reset(1'b0));

  PipelineRegister #(.Digits(69), .CtrlDigits(4)) EX_MEM_reg (.clk(clk), .
      Input({ALU_res, D2_m1_out, EX_dest}), .Output({MEM_ALUres, MEM_D1,
      MEM_dest}), .CtrlSignalIn({EX_memRead, EX_memWrite, EX_MemtoReg,
      EX_regWrite}), .CtrlSignalOut({MEM_memRead, MEM_memWrite, MEM_MemtoReg,
      MEM_regWrite}), .Hold(1'b0), .Reset(1'b0));

  PipelineRegister #(.Digits(69), .CtrlDigits(2)) MEM_WB_reg (.clk(clk), .
      Input({MEM_readData, MEM_ALUres, MEM_dest}), .Output({WB_MEM_data,
      WB_reg_data, WB_dest}), .CtrlSignalIn({MEM_MemtoReg, MEM_regWrite}), .
      CtrlSignalOut({WB_MemtoReg, WB_regWrite}), .Hold(1'b0), .Reset(1'b0));

  HazardDetection HazardD (.PCWrite(PC_Write), .IFIDWrite(IF_Write), .
      MuxSelect(Bubble), .IDEXMemRead(EX_memRead), .EXMEMMemRead(MEM_memRead),
      .Branch(Branch), .nBranch(nBranch), .IFIDRs(ins_rs), .IFIDRt(ins_rt), .
      IDEXRt(EX_rt), .EXMEMRegWrite(EX_regWrite), .EXMEMRt(MEM_dest), .EXMEMRd(
      EX_dest), .clk(clk));

  ForwardingUnit ForwardUnit ( .ALU_forward1(ForwardA), .ALU_forward2(ForwardB
      ), .Equal_forward1(Forward_branch1), .Equal_forward2(Forward_branch2), .
      MemSrc(Forward_load), .MEM_RegWrite(MEM_regWrite), .WB_RegWrite(
      WB_regWrite), .MEM_rd(MEM_dest), .WB_rd(WB_dest), .EX_rs(EX_rs), .EX_rt(
      EX_rt), .MEM_MemRead(MEM_memRead), .MEM_MemWrite(MEM_memWrite), .eBranch(
      Branch), .nBranch(nBranch), .ID_rs(ins_rs), .ID_rt(ins_rt));

  assign ins_op = ins[31:26];
  assign ins_rs = ins[25:21];
  assign ins_rt = ins[20:16];
  assign ins_imm = ins[15:0];
  assign ins_rd = ins[15:11];
  assign ins_funct = ins[5:0];
  assign ins_jump = ins[25:0];
  assign equal = (comp_1 == comp_2) ? 1 : 0;
  assign jump_addrs = {ID_pc4[31:28], ins_sh_out};
  assign PC_hold = ~PC_Write;
  assign IF_hold = ~ IF_Write;

  //Allows initialization of necessary control signals to 0
  assign BubCheck = (EX_flush === 1'bx) ? 0 : EX_flush;
  assign flush_check = (IF_flush === 1'bx) ? 0 : IF_flush;
      assign branch_check = (final_branch === 1'bx) ? 0 : final_branch;
  assign jump_check = (jump === 1'bx) ? 0 : jump;
endmodule
```

4.3 Simulation

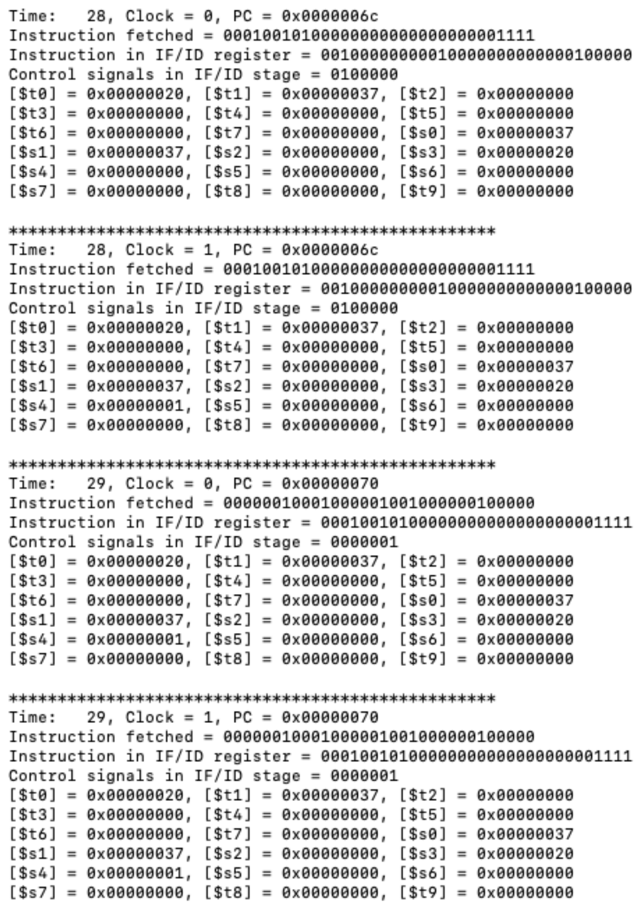
4.3.1 Simulation Scenario

The content of the Instruction Memory used in this simulation is as follows:

```
memory[0] = 32'b00100000000010000000000000100000; //addi $t0, $zero, 0x20
memory[1] = 32'b00100000000010010000000000110111; //addi $t1, $zero, 0x37
memory[2] = 32'b00000001000010011000000000100100; //and $s0, $t0, $t1
memory[3] = 32'b00000001000010011000000000100101; //or $s0, $t0, $t1
```

```
memory[4] = 32'b10101100000100000000000000000100; //sw $s0, 4($zero)
memory[5] = 32'b10101100000010000000000000001000; //sw $t0, 8($zero)
memory[6] = 32'b00000001000010011000100000100000; //add $s1, $t0, $t1
memory[7] = 32'b00000001000010011001000000100010; //sub $s2, $t0, $t1
memory[8] = 32'b00100000000010000000000000100000; //addi $t0, $zero, 0x20
memory[9] = 32'b00100000000010000000000000100000; //addi $t0, $zero, 0x20
memory[10] = 32'b00100000000010000000000000100000; //addi $t0, $zero, 0x20
memory[11] = 32'b00010010001100100000000000010010; //beq $s1, $s2, error0
memory[12] = 32'b10001100000100010000000000000100; //lw $s1, 4($zero)
memory[13]= 32'b00110010001100100000000001001000; //andi $s2, $s1, 0x48
memory[14] = 32'b00100000000010000000000000100000; //addi $t0, $zero, 0x20
memory[15] = 32'b00100000000010000000000000100000; //addi $t0, $zero, 0x20
memory[16] = 32'b00100000000010000000000000100000; //addi $t0, $zero, 0x20
memory[17] =32'b00010010001100100000000000001111; //beq $s1, $s2, error1
memory[18] =32'b10001100000100110000000000001000; //lw $s3, 8($zero)
memory[19] = 32'b00100000000010000000000000100000; //addi $t0, $zero, 0x20
memory[20] = 32'b00100000000010000000000000100000; //addi $t0, $zero, 0x20
memory[21] = 32'b00100000000010000000000000100000; //addi $t0, $zero, 0x20
memory[22] =32'b00010010000100110000000000001101; //beq $s0, $s3, error2
memory[23] =32'b00000010010100011010000000101010; //slt $s4, $s2, $s1 (Last)
memory[24] = 32'b00100000000010000000000000100000; //addi $t0, $zero, 0x20
memory[25] = 32'b00100000000010000000000000100000; //addi $t0, $zero, 0x20
memory[26] = 32'b00100000000010000000000000100000; //addi $t0, $zero, 0x20
memory[27] =32'b00010010100000000000000000001111; //beq $s4, $0, EXIT
memory[28] =32'b00000010001000001001000000100000; //add $s2, $s1, $0
memory[29] =32'b00001000000000000000000000010111; //j Last
memory[30] =32'b00100000000010000000000000000000; //addi $t0, $0, 0(error0)
memory[31] =32'b00100000000010010000000000000000; //addi $t1, $0, 0
memory[32] =32'b00001000000000000000000000111111; //j EXIT
memory[33] =32'b00100000000010000000000000000001; //addi $t0, $0, 1(error1)
memory[34] =32'b00100000000010010000000000000001; //addi $t1, $0, 1
memory[35] =32'b00001000000000000000000000111111; //j EXIT
memory[36] =32'b00100000000010000000000000000010; //addi $t0, $0, 2(error2)
memory[37] =32'b00100000000010010000000000000010; //addi $t1, $0, 2
memory[38] =32'b00001000000000000000000000111111; //j EXIT
memory[39] =32'b00100000000010000000000000000011; //addi $t0, $0, 3(error3)
memory[40] =32'b00100000000010010000000000000011; //addi $t1, $0, 3
memory[41] =32'b00001000000000000000000000111111; //j EXIT}
```

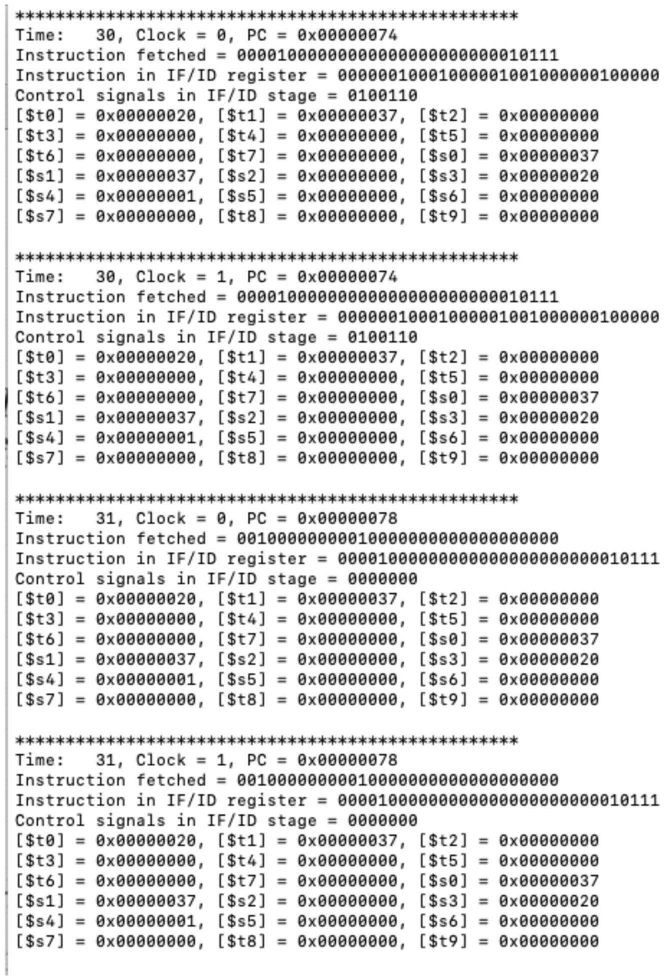
4.3.2 Textual Result



```
Time:   28, Clock = 0, PC = 0x0000006c
Instruction fetched = 00010010100000000000000000001111
Instruction in IF/ID register = 00100000000100000000000000100000
Control signals in IF/ID stage = 0100000
[$t0] = 0x00000020, [$t1] = 0x00000037, [$t2] = 0x00000000
[$t3] = 0x00000000, [$t4] = 0x00000000, [$t5] = 0x00000000
[$t6] = 0x00000000, [$t7] = 0x00000000, [$s0] = 0x00000037
[$s1] = 0x00000037, [$s2] = 0x00000000, [$s3] = 0x00000020
[$s4] = 0x00000000, [$s5] = 0x00000000, [$s6] = 0x00000000
[$s7] = 0x00000000, [$t8] = 0x00000000, [$t9] = 0x00000000

**************************************************
Time:   28, Clock = 1, PC = 0x0000006c
Instruction fetched = 00010010100000000000000000001111
Instruction in IF/ID register = 00100000000100000000000000100000
Control signals in IF/ID stage = 0100000
[$t0] = 0x00000020, [$t1] = 0x00000037, [$t2] = 0x00000000
[$t3] = 0x00000000, [$t4] = 0x00000000, [$t5] = 0x00000000
[$t6] = 0x00000000, [$t7] = 0x00000000, [$s0] = 0x00000037
[$s1] = 0x00000037, [$s2] = 0x00000000, [$s3] = 0x00000020
[$s4] = 0x00000001, [$s5] = 0x00000000, [$s6] = 0x00000000
[$s7] = 0x00000000, [$t8] = 0x00000000, [$t9] = 0x00000000

**************************************************
Time:   29, Clock = 0, PC = 0x00000070
Instruction fetched = 00000010001000001001000000100000
Instruction in IF/ID register = 00010010100000000000000000001111
Control signals in IF/ID stage = 0000001
[$t0] = 0x00000020, [$t1] = 0x00000037, [$t2] = 0x00000000
[$t3] = 0x00000000, [$t4] = 0x00000000, [$t5] = 0x00000000
[$t6] = 0x00000000, [$t7] = 0x00000000, [$s0] = 0x00000037
[$s1] = 0x00000037, [$s2] = 0x00000000, [$s3] = 0x00000020
[$s4] = 0x00000001, [$s5] = 0x00000000, [$s6] = 0x00000000
[$s7] = 0x00000000, [$t8] = 0x00000000, [$t9] = 0x00000000

**************************************************
Time:   29, Clock = 1, PC = 0x00000070
Instruction fetched = 00000010001000001001000000100000
Instruction in IF/ID register = 00010010100000000000000000001111
Control signals in IF/ID stage = 0000001
[$t0] = 0x00000020, [$t1] = 0x00000037, [$t2] = 0x00000000
[$t3] = 0x00000000, [$t4] = 0x00000000, [$t5] = 0x00000000
[$t6] = 0x00000000, [$t7] = 0x00000000, [$s0] = 0x00000037
[$s1] = 0x00000037, [$s2] = 0x00000000, [$s3] = 0x00000020
[$s4] = 0x00000001, [$s5] = 0x00000000, [$s6] = 0x00000000
[$s7] = 0x00000000, [$t8] = 0x00000000, [$t9] = 0x00000000
```

Figure 24: State of registers at time = 28

The instruction at 0x0000006c is beq $s4, 0, EXIT. Since this implementation assumes branch not taken, the PC will still increment by 4 after fetching the beq instruction. When PC = 0x00000070, it is able to decide that branch is not taken since $s4 = 0x00000001, so it doesn't have to take any penalty.



```
**************************************************
Time:   30, Clock = 0, PC = 0x00000074
Instruction fetched = 00001000000000000000000000010111
Instruction in IF/ID register = 00000010001000001001000000100000
Control signals in IF/ID stage = 0100110
[$t0] = 0x00000020, [$t1] = 0x00000037, [$t2] = 0x00000000
[$t3] = 0x00000000, [$t4] = 0x00000000, [$t5] = 0x00000000
[$t6] = 0x00000000, [$t7] = 0x00000000, [$s0] = 0x00000037
[$s1] = 0x00000037, [$s2] = 0x00000000, [$s3] = 0x00000020
[$s4] = 0x00000001, [$s5] = 0x00000000, [$s6] = 0x00000000
[$s7] = 0x00000000, [$t8] = 0x00000000, [$t9] = 0x00000000

**************************************************
Time:   30, Clock = 1, PC = 0x00000074
Instruction fetched = 00001000000000000000000000010111
Instruction in IF/ID register = 00000010001000001001000000100000
Control signals in IF/ID stage = 0100110
[$t0] = 0x00000020, [$t1] = 0x00000037, [$t2] = 0x00000000
[$t3] = 0x00000000, [$t4] = 0x00000000, [$t5] = 0x00000000
[$t6] = 0x00000000, [$t7] = 0x00000000, [$s0] = 0x00000037
[$s1] = 0x00000037, [$s2] = 0x00000000, [$s3] = 0x00000020
[$s4] = 0x00000001, [$s5] = 0x00000000, [$s6] = 0x00000000
[$s7] = 0x00000000, [$t8] = 0x00000000, [$t9] = 0x00000000

**************************************************
Time:   31, Clock = 0, PC = 0x00000078
Instruction fetched = 00100000000100000000000000000000
Instruction in IF/ID register = 00001000000000000000000000010111
Control signals in IF/ID stage = 0000000
[$t0] = 0x00000020, [$t1] = 0x00000037, [$t2] = 0x00000000
[$t3] = 0x00000000, [$t4] = 0x00000000, [$t5] = 0x00000000
[$t6] = 0x00000000, [$t7] = 0x00000000, [$s0] = 0x00000037
[$s1] = 0x00000037, [$s2] = 0x00000000, [$s3] = 0x00000020
[$s4] = 0x00000001, [$s5] = 0x00000000, [$s6] = 0x00000000
[$s7] = 0x00000000, [$t8] = 0x00000000, [$t9] = 0x00000000

**************************************************
Time:   31, Clock = 1, PC = 0x00000078
Instruction fetched = 00100000000100000000000000000000
Instruction in IF/ID register = 00001000000000000000000000010111
Control signals in IF/ID stage = 0000000
[$t0] = 0x00000020, [$t1] = 0x00000037, [$t2] = 0x00000000
[$t3] = 0x00000000, [$t4] = 0x00000000, [$t5] = 0x00000000
[$t6] = 0x00000000, [$t7] = 0x00000000, [$s0] = 0x00000037
[$s1] = 0x00000037, [$s2] = 0x00000000, [$s3] = 0x00000020
[$s4] = 0x00000001, [$s5] = 0x00000000, [$s6] = 0x00000000
[$s7] = 0x00000000, [$t8] = 0x00000000, [$t9] = 0x00000000
```

Figure 25: State of registers at time = 30

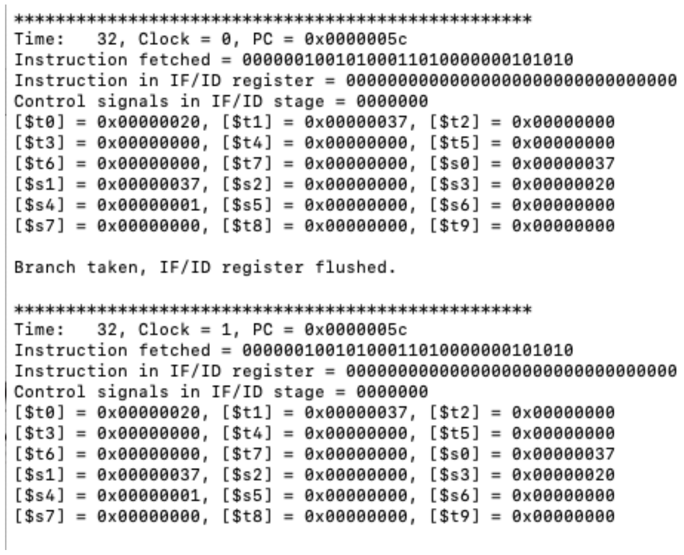
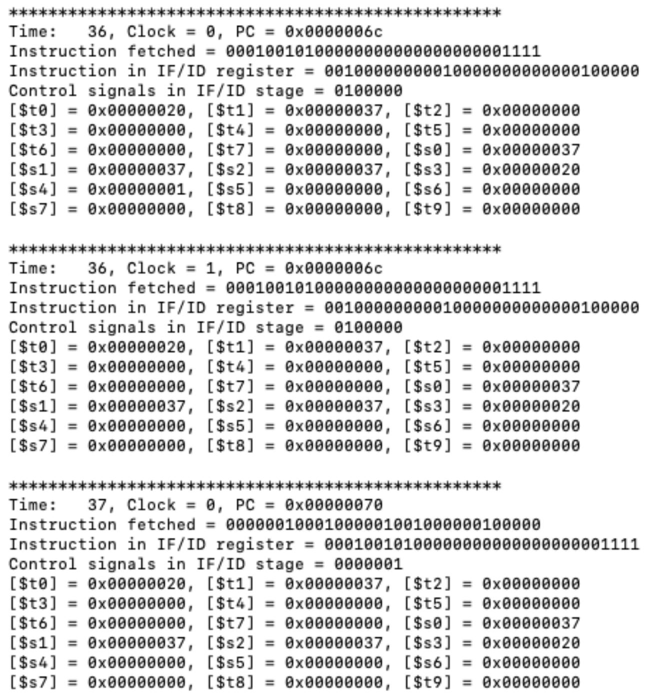

```
**************************************************
Time:   32, Clock = 0, PC = 0x0000005c
Instruction fetched = 00000010010100011010000000101010
Instruction in IF/ID register = 00000000000000000000000000000000
Control signals in IF/ID stage = 0000000
[$t0] = 0x00000020, [$t1] = 0x00000037, [$t2] = 0x00000000
[$t3] = 0x00000000, [$t4] = 0x00000000, [$t5] = 0x00000000
[$t6] = 0x00000000, [$t7] = 0x00000000, [$s0] = 0x00000037
[$s1] = 0x00000037, [$s2] = 0x00000000, [$s3] = 0x00000020
[$s4] = 0x00000001, [$s5] = 0x00000000, [$s6] = 0x00000000
[$s7] = 0x00000000, [$t8] = 0x00000000, [$t9] = 0x00000000

Branch taken, IF/ID register flushed.

**************************************************
Time:   32, Clock = 1, PC = 0x0000005c
Instruction fetched = 00000010010100011010000000101010
Instruction in IF/ID register = 00000000000000000000000000000000
Control signals in IF/ID stage = 0000000
[$t0] = 0x00000020, [$t1] = 0x00000037, [$t2] = 0x00000000
[$t3] = 0x00000000, [$t4] = 0x00000000, [$t5] = 0x00000000
[$t6] = 0x00000000, [$t7] = 0x00000000, [$s0] = 0x00000037
[$s1] = 0x00000037, [$s2] = 0x00000000, [$s3] = 0x00000020
[$s4] = 0x00000001, [$s5] = 0x00000000, [$s6] = 0x00000000
[$s7] = 0x00000000, [$t8] = 0x00000000, [$t9] = 0x00000000
```

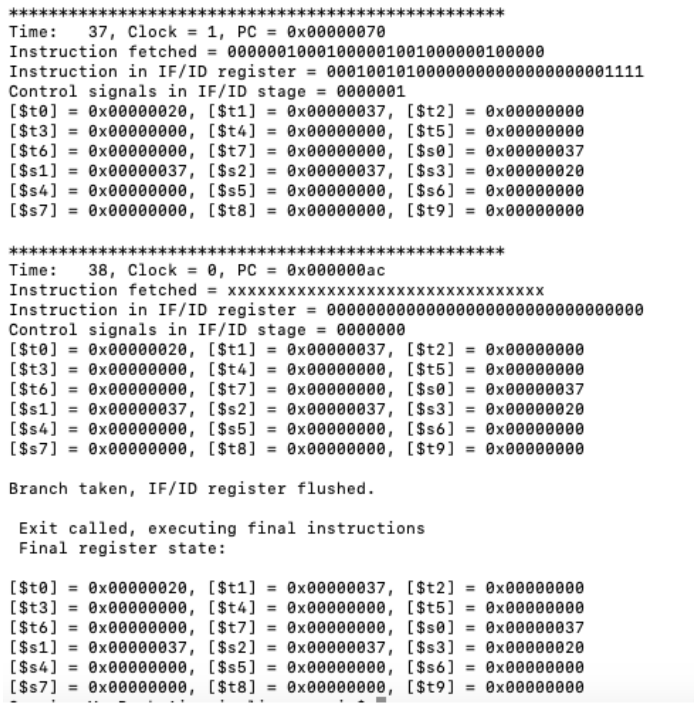
Figure 26: State of registers at time = 32

The instruction at 0x00000074 is j Last. Here, branch is taken, thus the IF/ID register has to be flushed. As can be seen, it will jump to 0x0000005c after it realizes that branch should be taken, which takes one clock cycle from the time the instruction is fetched. Since branch was taken, it flushed the IF/ID register containing the instruction at 0x00000078. It can be seen that the register has been flushed successfully since the instruction in the register is 32'b0 and control signals to be passed to ID/EX register is 7'b0.



```
**************************************************
Time:   36, Clock = 0, PC = 0x0000006c
Instruction fetched = 00010010100000000000000000001111
Instruction in IF/ID register = 00100000000100000000000000100000
Control signals in IF/ID stage = 0100000
[$t0] = 0x00000020, [$t1] = 0x00000037, [$t2] = 0x00000000
[$t3] = 0x00000000, [$t4] = 0x00000000, [$t5] = 0x00000000
[$t6] = 0x00000000, [$t7] = 0x00000000, [$s0] = 0x00000037
[$s1] = 0x00000037, [$s2] = 0x00000037, [$s3] = 0x00000020
[$s4] = 0x00000001, [$s5] = 0x00000000, [$s6] = 0x00000000
[$s7] = 0x00000000, [$t8] = 0x00000000, [$t9] = 0x00000000

**************************************************
Time:   36, Clock = 1, PC = 0x0000006c
Instruction fetched = 00010010100000000000000000001111
Instruction in IF/ID register = 00100000000100000000000000100000
Control signals in IF/ID stage = 0100000
[$t0] = 0x00000020, [$t1] = 0x00000037, [$t2] = 0x00000000
[$t3] = 0x00000000, [$t4] = 0x00000000, [$t5] = 0x00000000
[$t6] = 0x00000000, [$t7] = 0x00000000, [$s0] = 0x00000037
[$s1] = 0x00000037, [$s2] = 0x00000037, [$s3] = 0x00000020
[$s4] = 0x00000000, [$s5] = 0x00000000, [$s6] = 0x00000000
[$s7] = 0x00000000, [$t8] = 0x00000000, [$t9] = 0x00000000

**************************************************
Time:   37, Clock = 0, PC = 0x00000070
Instruction fetched = 00000010001000001001000000100000
Instruction in IF/ID register = 00010010100000000000000000001111
Control signals in IF/ID stage = 0000001
[$t0] = 0x00000020, [$t1] = 0x00000037, [$t2] = 0x00000000
[$t3] = 0x00000000, [$t4] = 0x00000000, [$t5] = 0x00000000
[$t6] = 0x00000000, [$t7] = 0x00000000, [$s0] = 0x00000037
[$s1] = 0x00000037, [$s2] = 0x00000037, [$s3] = 0x00000020
[$s4] = 0x00000000, [$s5] = 0x00000000, [$s6] = 0x00000000
[$s7] = 0x00000000, [$t8] = 0x00000000, [$t9] = 0x00000000
```

Figure 27: State of registers at time = 36



```
**************************************************
Time:   37, Clock = 1, PC = 0x00000070
Instruction fetched = 00000010001000001001000000100000
Instruction in IF/ID register = 00010010100000000000000000001111
Control signals in IF/ID stage = 0000001
[$t0] = 0x00000020, [$t1] = 0x00000037, [$t2] = 0x00000000
[$t3] = 0x00000000, [$t4] = 0x00000000, [$t5] = 0x00000000
[$t6] = 0x00000000, [$t7] = 0x00000000, [$s0] = 0x00000037
[$s1] = 0x00000037, [$s2] = 0x00000037, [$s3] = 0x00000020
[$s4] = 0x00000000, [$s5] = 0x00000000, [$s6] = 0x00000000
[$s7] = 0x00000000, [$t8] = 0x00000000, [$t9] = 0x00000000

**************************************************
Time:   38, Clock = 0, PC = 0x000000ac
Instruction fetched = xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Instruction in IF/ID register = 00000000000000000000000000000000
Control signals in IF/ID stage = 0000000
[$t0] = 0x00000020, [$t1] = 0x00000037, [$t2] = 0x00000000
[$t3] = 0x00000000, [$t4] = 0x00000000, [$t5] = 0x00000000
[$t6] = 0x00000000, [$t7] = 0x00000000, [$s0] = 0x00000037
[$s1] = 0x00000037, [$s2] = 0x00000037, [$s3] = 0x00000020
[$s4] = 0x00000000, [$s5] = 0x00000000, [$s6] = 0x00000000
[$s7] = 0x00000000, [$t8] = 0x00000000, [$t9] = 0x00000000

Branch taken, IF/ID register flushed.

 Exit called, executing final instructions
 Final register state:

[$t0] = 0x00000020, [$t1] = 0x00000037, [$t2] = 0x00000000
[$t3] = 0x00000000, [$t4] = 0x00000000, [$t5] = 0x00000000
[$t6] = 0x00000000, [$t7] = 0x00000000, [$s0] = 0x00000037
[$s1] = 0x00000037, [$s2] = 0x00000037, [$s3] = 0x00000020
[$s4] = 0x00000000, [$s5] = 0x00000000, [$s6] = 0x00000000
[$s7] = 0x00000000, [$t8] = 0x00000000, [$t9] = 0x00000000
```

Figure 28: Final state of register

At Time = 36, the instruction fetches the beq \$s4, 0, EXIT at 0x0000006c for the second time. However this time, it is supposed to jump since \$s4 should be 0. Note that the pipeline registers and PC are updated at negative edge of clock while the registers are updated as positive edge of clock, hence \$s4 changes at positive edge of clock in the graph above.

As expected, at Time = 37 it realizes that branch was taken, thus it sets the IF/ID register to be flushed at the next clock cycle and sends the instruction to branch in the next clock cycle. Thus, at Time = 38, the IF/ID register containing the instruction at 0x00000070 is flushed and the PC address is the target address of the branch instruction. The final state of the register after executing the remaining instruction in EX,MEM, and WB stages is also shown.

Please view the Appendix section for the full textual result

4.4 FPGA Board Implementation

To implement the pipeline processor on the board, the first thing is to write SSD display modules to display the lower four hexadecimal digits of the output of program counter or any register file.
SSD is written in `SSDDisplay.v, SSDDriver.v, SSDAllocate.v, RingCounter.v, ClockDivider.v`. These modules work together to display the address from program counter, and the values in any register file.
In the top module we need to define these things: `PCRegSwitch, RegLocation, Portclk`.
We write a sections of codes in main to access the content of the registers. The location of the register is indicated by the input RegLocation.We have a total of 18 registers who content we may change, therefore, we used five switches to input the location of register whose values we wish to read, and they combine into one input RegLocation.
Next is to choose to either display PC or register file. SSD Allocate reads the input PCRegSwitch, and decides which should be the output.
Finally, the modules generates output onto SSD.The port clock is connected to W5 of the FPGA board, which has $100MHz$ frequency. We then use the clockdivider to make it into frequency $500Hz$, which is used to turn the AN so we can show all four numbers.We can show on SSD four hexadecimal digits at a time. SSD Driver can transform the four digits from either PC or register into four 7-bit variable which can be used to control cathodes in SSD. SSD Display and RingCounter are the modules that control the anodes, and they work with the rest of the modules to show the digits.
The codes in The Verilog code is as follows:
ClockDivider.v

```verilog
`timescale 1ns / 100ps

module ClockDivider#(
parameter Width = 3, // Width of the register required
parameter N = 6// We will divide by 12 for example in this case
)
(clk,Reset, ClkOut);
//By HU Jiaoyang 517370910140 at VE370 Project 2 19FA
input clk;
input Reset;
output ClkOut;

reg [Width-1:0] RReg;
wire [Width-1:0] RNext;
reg tmp;

always @(posedge clk or posedge Reset)

begin
  if (Reset)
     begin
        RReg <= 0;
    tmp <= 1'b0;
     end

      else if (RNext == N)
       begin
         RReg <= 0;
         tmp <= ~tmp;
       end

  else
      RReg <= RNext;
end

 assign RNext = RReg + 1;
 assign ClkOut = tmp;
endmodule
```

RingCounter.v

```verilog
`timescale 1ns / 100ps

module RingCounter(
clk,
Reset,
Output
);
//by HU Jiaoyang VE370 Project 2 19FA
input clk,Reset;
output reg [3:0] Output;

always @ (posedge clk or posedge Reset) begin
  if(Reset) Output <= 4'b1110;
  else begin
  Output[3] <= Output[0];
  Output[2] <= Output[3];
  Output[1] <= Output[2];
  Output[0] <= Output[1];
  end
end
endmodule
```

SSDAllocate.v

```
`timescale 1ns / 100ps
module SSDAllocate(
clk,PCRegSwitch,pc_out,Reg,Four,Three,Two,One
    );
    input clk,PCRegSwitch;
    input [31:0] pc_out,Reg;
    output reg [3:0] Four,Three,Two,One;
   always @ (clk) begin
    if (PCRegSwitch)begin
    Four <= pc_out[15:12];
    Three <= pc_out[11:8];
    Two <= pc_out[7:4];
    One <= pc_out[3:0];
    end
    else begin
    Four <= Reg[15:12];
    Three <= Reg[11:8];
    Two <= Reg[7:4];
    One <= Reg[3:0];
    end
    end
endmodule
```

SSDDisplay.v

```
`timescale 1ns / 100ps

```

```
module SSDDisplay(
input [6:0] Four,
input [6:0] Three,
input [6:0] Two,
input [6:0] One,
input [3:0] AN,
output [6:0] CA
    );
    //By HU Jiaoyang 517370910140 at VE370 Project 2 19FA
    assign CA = (AN == 4'b1110) ? One : 7'bzzzzzzz,
    CA = (AN == 4'b1101) ? Two : 7'bzzzzzzz,
    CA= (AN == 4'b1011) ? Three : 7'bzzzzzzz,
    CA = (AN == 4'b0111) ? Four : 7'bzzzzzzz;
endmodule
```

SSDDriver.v

```
`timescale 1ns / 100ps

module SSDDriver(
input [3:0] Input,
output reg [6:0] Output
    );
//by HU Jiaoyang at 19FA VE370 Project 2
always @ (Input) begin
    case(Input)
    0: Output <= 7'b0000001;
    1: Output <= 7'b1001111;
    2: Output <= 7'b0010010;
    3: Output <= 7'b0000110;
    4: Output <= 7'b1001100;
    5: Output <= 7'b0100100;
    6: Output <= 7'b0100000;
    7: Output <= 7'b0001111;
    8: Output <= 7'b0000000;
    9: Output <= 7'b0000100;
    10: Output <= 7'b0001000;
    11:Output <= 7'b1100000;
    12: Output <= 7'b0110001;
    13: Output <= 7'b1000010;
    14: Output <= 7'b0110000;
    15: Output <= 7'b0111000;
    default Output <= 7'b1111111;// When there is no input, show nothing
    endcase
end
endmodule
```

With SSD display finished, the next issue is I/O planning. Anodes(W4,V4,U4,U2) and cathodes(W7,W6,U8,V8,U5,V5,U7) of the SSD are connected to AN,CA of the program as output. As for the input, the pipeline processor takes the following inputs: clk, Port-clk,PCRegSwitch,Reset,RegLocation.clk is the switch that we are using as clock cycle input, therefore it is connected to V17 of the FPGA board.PCRegSwitch is the switch that decides whether SSD outputs the content of register file or the output of the program counter, we connect it to switch V16.RegLocation is a 5-bit input from switches that tells the processor to access $\$t0 - \$t9$ and $\$s0 - \$s7$. We connect the Reglocation to R2,T1,V1,W2,R3. We connect reset to button U18. To make SSD work, we need a clock to switch between the anodes and update the SSDs. Therefore, we take in the $100MHz$ clock from pin W5, and use clock divider to make it to $500Hz$.

5 Conclusion & Discussion

In this project, we successfully provided single cycle implementation and pipelined implementation for MIPS instruction. We tested our correctness by writing testbenches and compare the result of simulation with the theoretical ones. For pipelined implementation, we also run the program on FPGA board to check the value of registers after each clock cycle. We get to know all the details of how to implement hardware for MIPS computer, and understand the knowledge we have learned in class much better.

At first we have encountered with problems. When implementing single cycle case, we first run into the situation where the `pc` result is always `0xxxxxxxxx`. After careful debugging we found that the error came from no delay of pc assignment. Hence we change the assignment style in **PC module** from " $<=$ " to " $=$ " to fix this problem.

After solving all those problems in the project, we learn more about how to deal with imperfectness in the real world, and grasp the theory better.

6 References

1. Patterson and Hennessy, Morgan Kaufmann Publishers, Computer Organization and Design.