**San Jose State University**

**Department of Computer Engineering**

**CMPE 140 Lab Report**

**==========================================================**

**Lab \_\_7\_\_ Report**

**Title: Enhanced Single-cycle MIPS Processor**

**Semester \_\_\_\_\_\_\_\_\_\_\_\_\_ 4/23/2016 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**by**

**Name \_\_\_\_\_LiXin Li\_\_\_\_\_\_\_ SID \_\_007702539\_\_**

**Name \_\_\_\_\_Mandy Phi\_\_\_\_\_\_SID \_\_008544952\_\_\_**

**Lab Record**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Performed by (print name)** | **Checked by (print name)** | **Successfully Completed** | **Partially Completed\*** | **Failed or Not Performed\*** |
| **Lixin Li** | **Sang** | **yes** |  |  |
| **Mandy Phi** | **Sang** | **yes** |  |  |

**\* Detailed descriptions must be given in the report.**

Purpose

The purpose of this lab assignment was to enhance the single cycle MIPs processor functionality by extending its instruction set, which includes the following instructions: MULT, MFHI, MFLO, JR, JAL. The design must be tested via both functional verification and FPGA validation using Nexys 3 board.

**Materials and Softwar**e

1. MIPS Assembler
2. MIPS Test Log
3. CmpE 140 Lab 7 Assignment Sheet
4. Xilinx ISE Design Suite version 14.7
5. Nexys 3 FPGA board

**Tasks**

**Week 1.** The main task of this week was to design the given data path to extended one which include all the new instructions. Rough draft of the schematic was drawn in Visio.

**Week 2.** The main task of week 2 was to finish the implementation of all the added instructions into the extended data path schematic. Final draft of the schematic was drawn in Visio. The final draft of the schematic can be found in Figure 1 in Appendix.

**Week 3.**

*Task 1.* There were three tasks to be completed in week 3. The first task was to complete the extended MIPS processor design in Verilog.

There are few new modules and new initializations added into the extended MIPS processor design.

* **Multiply module for “multu” instruction**

module multiply(input [31:0] a, b, output [31:0] hi, lo);

wire [63:0] product;

assign product = a \* b;

assign hi = product[63:32];

assign lo = product[31:0];

endmodule

* **An extra register for storing hi and lo multiplying values**

module reg2 #(parameter WIDTH = 32)(

input clk, reset, en,

input [WIDTH-1:0] d,

output reg [WIDTH-1:0] q);

always@(posedge clk)

begin

if(reset) q = 32'b0;

else

begin

if(en) q = d;

else q = q;

end

end

endmodule

* **Extended next PC logic, initialized in datapath module**

mux2 #(32) JRtopcmux(pcplus4, srca, JR, pcnext\_jr);

//choosing between JR instruction or regular pcplus4 logic

mux2 #(32) pcbrmux(pcnext\_jr, pcbranch, pcsrc, pcnextbr);

// choosing between the result from JR mux and Branch instruction

mux2 #(32) pcmux(pcnextbr, {pcplus4[31:28], instr[25:0], 2'b00}, jump, pcnext);

// choossing between the result from Branch mux and JUMP instruction

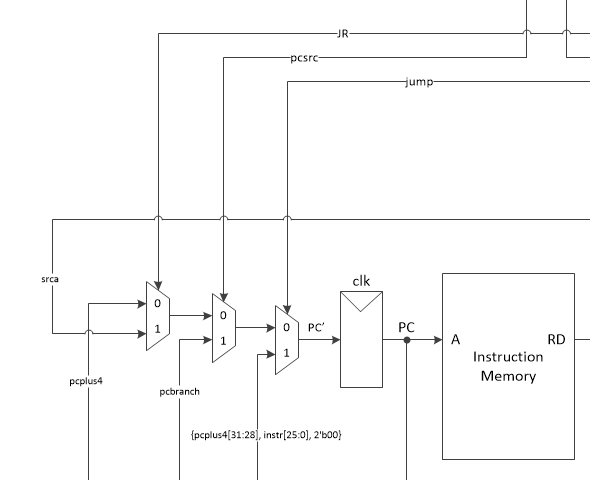


Figure 2. Next PC logic schematic

* **Writedata logic**

mux2 #(32) resmux(aluout, readdata, memtoreg, result);

//original mux

mux2 #(32) linkmux2(result, pcplus4, link, link\_mult);

// choose between result and pcplus4 , activate/link signal = 1 when link instruction is called

mux2 #(32) multu(link\_mult, output\_hiorlo, mult, mult\_WD);

// choose between the link mux result and the output of hilomux, activate/hilo signal = 1 when mult signal is called

* **Multu logic**

multiply multip(srca, writedata, mult\_hi, mult\_lo);

// multiplying srca and writedata; this unit will take srca and writedat, two unsigned 32 bits integers, multiply them to a 64 bits product, the product will split and send to the HI and LO register below.

reg2 HI(clk, reset, we\_mult, mult\_hi, output\_hi);

// store $hi content

reg2 LO(clk, reset, we\_mult, mult\_lo, output\_lo);

// store $lo content

mux2 #(32) hilomux(output\_hi, output\_lo, hilo, output\_hiorlo);

// MFHI or MFLO and put it into the result bus/WD

Control unit signals were also extended due to extra instructions implemented into the design. Below is the complete signal table. (X indicates don’t care, which all zero in the actual signals)

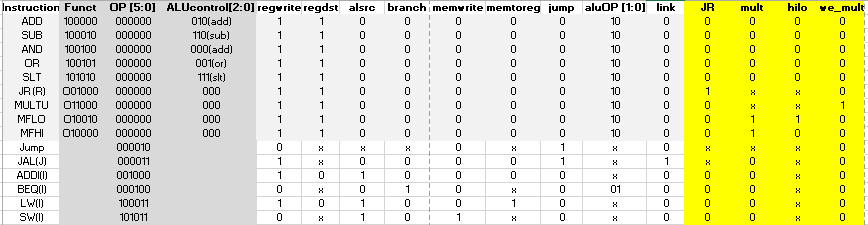


Figure 2. Control Unit signals

regwrite, regdst, alusrc, branch, memwrite, memtoreg, jump, aluop, link are the control signals output from the **main decoder**.

6'b000000: controls <= 10'b1100000100; //Rtype

6'b100011: controls <= 10'b1010010000; //LW

6'b101011: controls <= 10'b0010100000; //SW

6'b000100: controls <= 10'b0001000010; //BEQ

6'b001000: controls <= 10'b1010000000; //ADDI

6'b000010: controls <= 10'b0000001000; //J

6'b000011: controls <= 10'b1000001001; //jal

JR, MFHI, MFLO and MULTU are all R type instructions and they cannot be decode by the main decoder. Their function fields are all decoded inside the **ALU decoder** and set their controls JR, mult, hilo, we\_mult as the output control signals.

//The Rtypecontrols signals are all zero for the below instructions because they are only used in ALU.

6'b100000: begin alucontrol <= 3'b010; Rtypecontrols <= 4'b0000; end // ADD

6'b100010: begin alucontrol <= 3'b110; Rtypecontrols <= 4'b0000; end // SUB

6'b100100: begin alucontrol <= 3'b000; Rtypecontrols <= 4'b0000; end // AND

6'b100101: begin alucontrol <= 3'b001; Rtypecontrols <= 4'b0000; end // OR

6'b101010: begin alucontrol <= 3'b111; Rtypecontrols <= 4'b0000; end // SLT

//added cases; below alucontrols are all zero because they are not for the controls inside the ALU

6'b011001: begin alucontrol <= 3'b000; Rtypecontrols <= 4'b0001; end //MULT

6'b010000: begin alucontrol <= 3'b000; Rtypecontrols <= 4'b0100; end //MFHI

6'b001000: begin alucontrol <= 3'b000; Rtypecontrols <= 4'b1000; end //JR

*Task 2.* The second task was to write a testbench to simulate the extended design.

*Task 3.* The third task was to implement the design onto the Nexys3 FPGA board.

**Discussion and Results**

First, the team execute the given assembly and found out that the final result is 18 in hex stored in register s0, which is register 16.

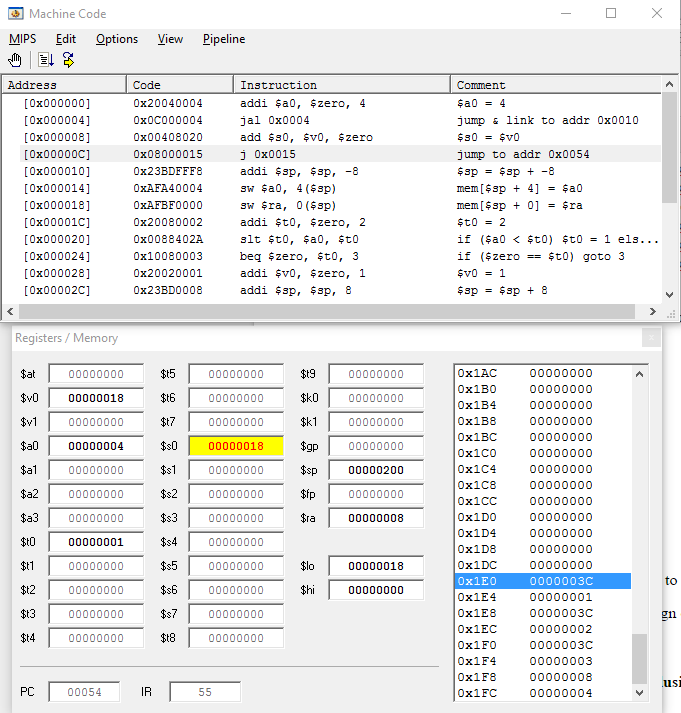


Figure 3. Executed given assembly code

A simple testbench was written to functionally testing the team’s verilog design. The testbench was not self checking because the team was not able to finish it. However, the team was able to use the simple testbench to validate design. The testbench used three modules, mips, imem and dmem and when PC reaches to the end, the test bench will stop. As shown, when PC hits 54, the content in register 16 is 18. This verified the functionality of the design.

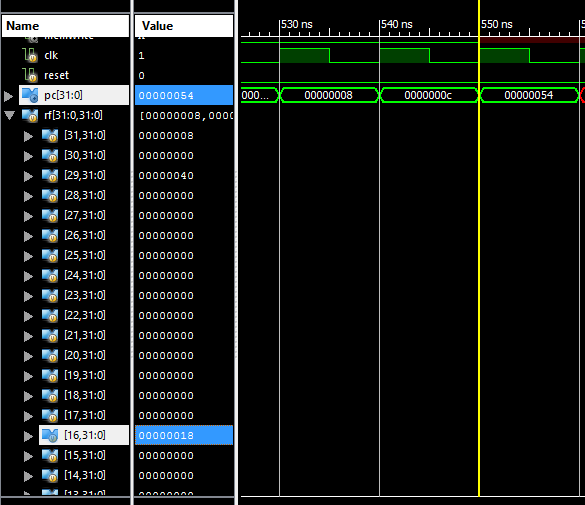


Figure 4. Testbench simulation

The team implemented the code onto the Nexys 3 FPGA board and set the switch to be 000 and set the register display to be 16. The team also waited until the last PC 54 and inded the display show 18. This successfully verify the functionality of the design and prove that the design can do the added instructions include Jal, JR, addi, mult, mflo, mfhi.

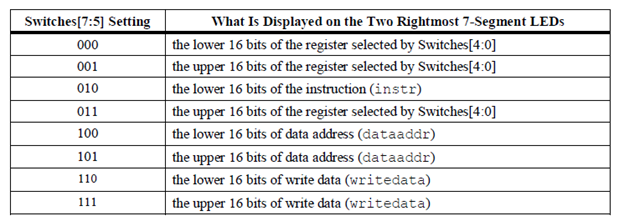


Figure 5. On board switches table

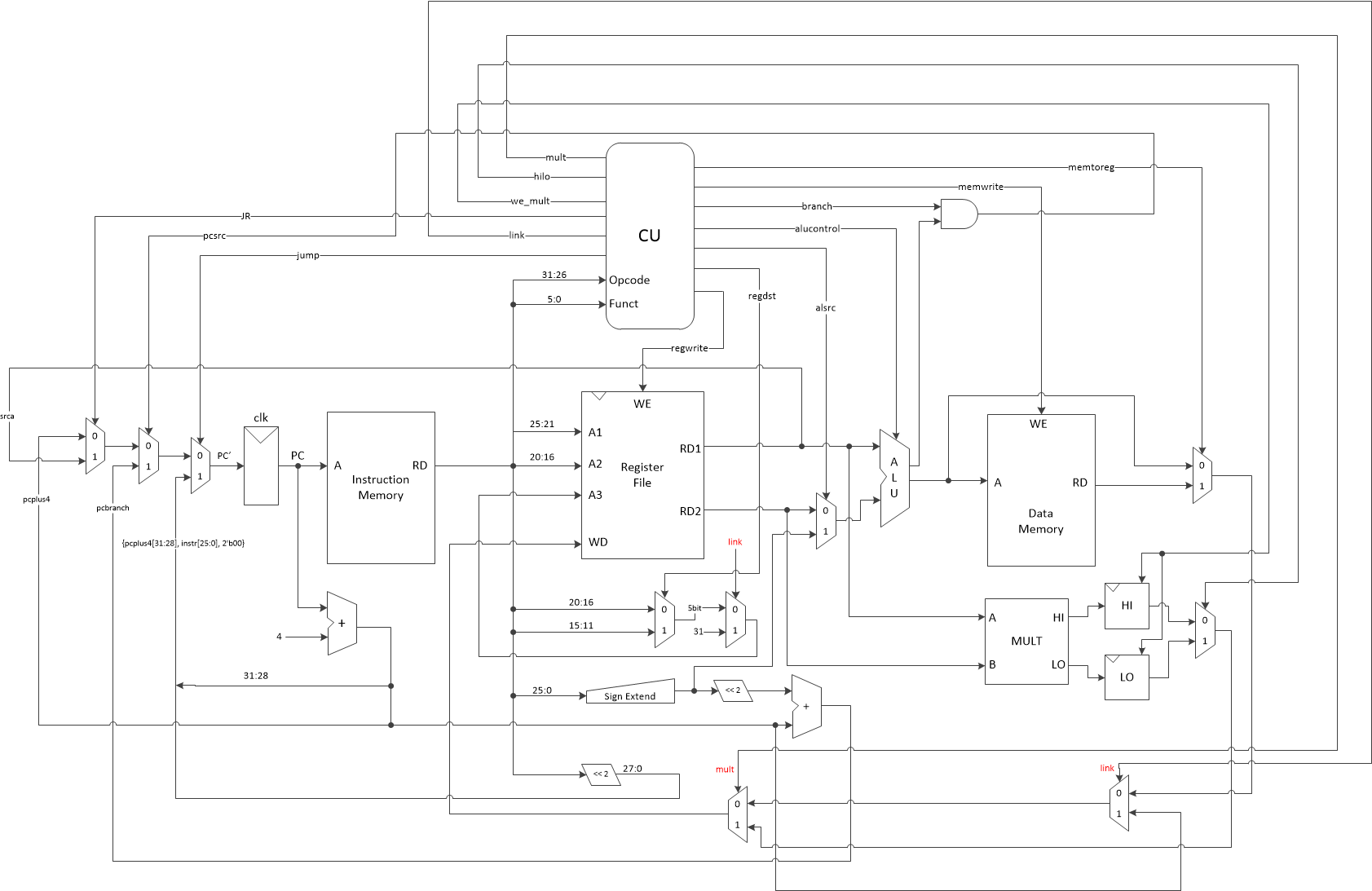


Figure 6. On board FPGA testing

**Conclusion**

The team was successfully finished the three weeks tasks assignment for this lab, the lab was completed and executed without any problems and fulfilled all the requirements. Through this lab, the team gained a better understanding of the MULT, MFLO, JR, JAL instruction design of single-cycle MIPS processor and learned the hardware validation processor on the FPGA board. This lab provided the team with fundamental skills and knowledge needed for future lab assignments.

**Appendix**



**Figure 1. Overall extended datapath schematic**

Main:

# addi $sp, $0, 48 not for SPIM

addi $a0, $0, 4 # set arg

jal factorial # compute the factorial

add $s0, $v0, $0 # move result into $s0

j end

Factorial:

addi $sp, $sp, -8 # make room on stack

sw $a0, 4($sp) # store $a0

sw $ra, 0($sp) # store $ra

addi $t0, $0, 2 # $t0 = 2

slt $t0, $a0, $t0 # a <= 1 ?

beq $t0, $0, else # no - goto else

addi $v0, $0, 1 # yes - return 1

addi $sp, $sp, 8 # restore $sp

jr $ra # return

Else:

addi $a0, $a0, -1 # n = n - 1

jal factorial # recursive call

lw $ra, 0($sp) # restore $ra

lw $a0, 4($sp) # restore $a0

addi $sp, $sp, 8 # restore $sp

multu $a0, $v0 # n \* factorial(n-1)

mflo $v0 # mv result into $v0

jr $ra

End:

**File 1. Given assembly code**

//-----------------------------------------------------------------

// Module Name : clk\_gen

// Description : Generate 4 second and 5KHz clock cycle from

// the 50MHz clock on the Nexsys2 board

//------------------------------------------------------------------

module clk\_gen(

input clk50MHz, reset,

output reg clksec );

reg clk\_5KHz;

integer count, count1;

always@(posedge clk50MHz) begin

if(reset) begin

count = 0;

count1 = 0;

clksec = 0;

clk\_5KHz =0;

end else begin

if (count == 50000000) begin

// Just toggle after certain number of seconds

clksec = ~clksec;

count = 0;

end

if (count1 == 20000) begin

clk\_5KHz = ~clk\_5KHz;

count1 = 0;

end

count = count + 1;

count1 = count1 + 1;

end

end

endmodule

//------------------------------------------------

// Source Code for a Single-cycle MIPS Processor (supports partial instruction)

// Developed by D. Hung, D. Herda and G. Gerken,

// based on the following source code provided by

// David\_Harris@hmc.edu (9 November 2005):

// mipstop.v

// mipsmem.v

// mips.v

// mipsparts.v

//------------------------------------------------

// Main Decoder

module maindec(

input [5:0] op,

output memtoreg, memwrite, branch, alusrc, regdst, regwrite, jump, link,

output [1:0] aluop );

reg [9:0] controls;

assign {regwrite, regdst, alusrc, branch, memwrite, memtoreg, jump, aluop, link} = controls;

always @(\*)

case(op)

6'b000000: controls <= 10'b1100000100; //Rtype

6'b100011: controls <= 10'b1010010000; //LW

6'b101011: controls <= 10'b0010100000; //SW

6'b000100: controls <= 10'b0001000010; //BEQ

6'b001000: controls <= 10'b1010000000; //ADDI

6'b000010: controls <= 10'b0000001000; //J

6'b000011: controls <= 10'b1000001001; //jal

default: controls <= 10'bxxxxxxxxx; //???

endcase

endmodule

// ALU Decoder

module aludec(

input [5:0] funct,

input [1:0] aluop,

output reg [2:0] alucontrol,

output JR, mult, hilo, we\_mult);

reg [3:0] Rtypecontrols;

assign {JR, mult, hilo, we\_mult} = Rtypecontrols;

always @(\*)

case(aluop)

2'b00: begin alucontrol <= 3'b010; Rtypecontrols <= 4'b0000; end // add

2'b01: begin alucontrol <= 3'b110; Rtypecontrols <= 4'b0000; end // sub

default: case(funct) // RTYPE

6'b100000: begin alucontrol <= 3'b010; Rtypecontrols <= 4'b0000; end // ADD

6'b100010: begin alucontrol <= 3'b110; Rtypecontrols <= 4'b0000; end // SUB

6'b100100: begin alucontrol <= 3'b000; Rtypecontrols <= 4'b0000; end // AND

6'b100101: begin alucontrol <= 3'b001; Rtypecontrols <= 4'b0000; end // OR

6'b101010: begin alucontrol <= 3'b111; Rtypecontrols <= 4'b0000; end // SLT

//added cases

6'b011001: begin alucontrol <= 3'b000; Rtypecontrols <= 4'b0001; end //MULT

6'b010000: begin alucontrol <= 3'b000; Rtypecontrols <= 4'b0100; end //MFHI

6'b010010: begin alucontrol <= 3'b000; Rtypecontrols <= 4'b0110; end //MFLO

6'b001000: begin alucontrol <= 3'b000; Rtypecontrols <= 4'b1000; end //JR

default: begin alucontrol <= 3'bxxx; Rtypecontrols <= 4'b0000; end // ???

endcase

endcase

endmodule

// ALU

module alu(

input [31:0] a, b,

input [ 2:0] alucont,

output reg [31:0] result,

output zero );

wire [31:0] b2, sum, slt;

assign b2 = alucont[2] ? ~b:b;

assign sum = a + b2 + alucont[2];

assign slt = sum[31];

always@(\*)

case(alucont[1:0])

2'b00: result <= a & b;

2'b01: result <= a | b;

2'b10: result <= sum;

2'b11: result <= slt;

endcase

assign zero = (result == 32'b0);

endmodule

//multiplier

module multiply(input [31:0] a, b, output [31:0] hi, lo);

wire [63:0] product;

assign product = a \* b;

assign hi = product[63:32];

assign lo = product[31:0];

endmodule

// Adder

module adder(

input [31:0] a, b,

output [31:0] y );

assign y = a + b;

endmodule

// Two-bit left shifter

module sl2(

input [31:0] a,

output [31:0] y );

// shift left by 2

assign y = {a[29:0], 2'b00};

endmodule

// Sign Extension Unit

module signext(

input [15:0] a,

output [31:0] y );

assign y = {{16{a[15]}}, a};

endmodule

// Parameterized Register

module flopr #(parameter WIDTH = 8) (

input clk, reset,

input [WIDTH-1:0] d,

output reg [WIDTH-1:0] q);

always @(posedge clk, posedge reset)

if (reset) q <= 0;

else q <= d;

endmodule

// commented out since flopenr is not used

//module flopenr #(parameter WIDTH = 8) (

// input clk, reset,

// input en,

// input [WIDTH-1:0] d,

// output reg [WIDTH-1:0] q);

//

// always @(posedge clk, posedge reset)

// if (reset) q <= 0;

// else if (en) q <= d;

//endmodule

// Parameterized 2-to-1 MUX

module mux2 #(parameter WIDTH = 8) (

input [WIDTH-1:0] d0, d1,

input s,

output [WIDTH-1:0] y );

assign y = s ? d1 : d0;

endmodule

// register file with one write port and three read ports

// the 3rd read port is for prototyping dianosis

module regfile(

input clk,

input we3,

input [4:0] ra1, ra2, wa3,

input [31:0] wd3,

output [31:0] rd1, rd2,

input [4:0] ra4,

output [31:0] rd4);

reg [31:0] rf[31:0];

integer n;

//initialize registers to all 0s

initial

for (n=0; n<32; n=n+1)

rf[n] = 32'h00;

rf[29] = 32'h0000003C;

//write first order, include logic to handle special case of $0

always @(posedge clk)

if (we3)

if (~ wa3[4])

rf[{0,wa3[3:0]}] <= wd3;

else

rf[{1,wa3[3:0]}] <= wd3;

// this leads to 72 warnings

//rf[wa3] <= wd3;

// this leads to 8 warnings

//if (~ wa3[4])

// rf[{0,wa3[3:0]}] <= wd3;

//else

// rf[{1,wa3[3:0]}] <= wd3;

assign rd1 = (ra1 != 0) ? rf[ra1[4:0]] : 0;

assign rd2 = (ra2 != 0) ? rf[ra2[4:0]] : 0;

assign rd4 = (ra4 != 0) ? rf[ra4[4:0]] : 0;

endmodule

//reg2

module reg2 #(parameter WIDTH = 32)(

input clk, reset, en,

input [WIDTH-1:0] d,

output reg [WIDTH-1:0] q);

always@(posedge clk)

begin

if(reset) q = 32'b0;

else

begin

if(en) q = d;

else q = q;

end

end

endmodule

// Control Unit

module controller(

input [5:0] op, funct,

input zero,

output memtoreg, memwrite, pcsrc, alusrc, regdst, regwrite, jump, link, JR, mult, hilo, we\_mult,

output [2:0] alucontrol );

wire [1:0] aluop;

wire branch;

maindec md(op, memtoreg, memwrite, branch, alusrc, regdst, regwrite, jump, aluop, link);

aludec ad(funct, aluop, alucontrol, JR, mult, hilo, we\_mult);

assign pcsrc = branch & zero;

endmodule

// Data Path (excluding the instruction and data memories)

module datapath(

input clk, reset, memtoreg, pcsrc, alusrc, regdst, regwrite, jump, link, JR, mult, hilo, we\_mult,

input [2:0] alucontrol,

output zero,

output [31:0] pc,

input [31:0] instr,

output [31:0] aluout, writedata,

input [31:0] readdata,

input [4:0] dispSel,

output [31:0] dispDat );

wire [4:0] writereg;

wire [31:0] pcnext, pcnextbr, pcplus4, pcbranch, signimm, signimmsh, srca, srcb, result;

//added wires

wire [31:0] pcnext\_jr, link\_mult, mult\_WD, mult\_hi, mult\_lo, output\_hi, output\_lo;

wire [4:0] link\_A3;

// next PC logic

flopr #(32) pcreg(clk, reset, pcnext, pc);

adder pcadd1(pc, 32'b100, pcplus4);

sl2 immsh(signimm, signimmsh);

adder pcadd2(pcplus4, signimmsh, pcbranch);

//JR

mux2 #(32) JRtopcmux(pcplus4, srca, JR, pcnext\_jr);

//

mux2 #(32) pcbrmux(pcnext\_jr, pcbranch, pcsrc, pcnextbr);

mux2 #(32) pcmux(pcnextbr, {pcplus4[31:28], instr[25:0], 2'b00}, jump, pcnext);

// register file logic////////////

regfile rf(clk, regwrite, instr[25:21], instr[20:16], link\_A3, mult\_WD, srca, writedata, dispSel, dispDat);

//data address mux

mux2 #(5) wrmux(instr[20:16], instr[15:11], regdst, writereg);

mux2 #(5) linkmux1(writereg, 5'd31, link, link\_A3);

// write data mux

mux2 #(32) resmux(aluout, readdata, memtoreg, result);

mux2 #(32) linkmux2(result, pcplus4, link, link\_mult);

mux2 #(32) multu(link\_mult, output\_hiorlo, mult, mult\_WD);

//immediate value sign extention

signext se(instr[15:0], signimm);

//multiplier -- module multiply(input [31:0] a, b, output [31:0] hi, lo);

multiply multip(srca, writedata, mult\_hi, mult\_lo);

reg2 HI(clk, reset, we\_mult, mult\_hi, output\_hi);

reg2 LO(clk, reset, we\_mult, mult\_lo, output\_lo);

mux2 #(32) hilomux(output\_hi, output\_lo, hilo, output\_hiorlo);

// ALU logic

mux2 #(32) srcbmux(writedata, signimm, alusrc, srcb);

alu alu(srca, srcb, alucontrol, aluout, zero);

endmodule

// The MIPS (excluding the instruction and data memories)

module mips(

input clk, reset,

output [31:0] pc,

input [31:0] instr,

output memwrite,

output [31:0] aluout, writedata,

input [31:0] readdata,

input [4:0] dispSel,

output [31:0] dispDat );

// deleted wire "branch" - not used

wire memtoreg, pcsrc, zero, alusrc, regdst, regwrite, jump;

wire [2:0] alucontrol;

controller c(instr[31:26], instr[5:0], zero,

memtoreg, memwrite, pcsrc,

alusrc, regdst, regwrite, jump, link, JR, mult, hilo, we\_mult,

alucontrol); //added 5 signals

datapath dp(clk, reset, memtoreg, pcsrc,

alusrc, regdst, regwrite, jump, link, JR, mult, hilo, we\_mult,

alucontrol, zero, pc, instr, aluout,

writedata, readdata, dispSel, dispDat); //added 5 signals

endmodule

// Instruction Memory

module imem (

input [5:0] a,

output [31:0] dOut );

reg [31:0] rom[0:63];

//initialize rom from memfile\_s.dat

initial

$readmemh("memfile.dat", rom);

//simple rom

assign dOut = rom[a];

endmodule

// Data Memory

module dmem (

input clk,

input we,

input [31:0] addr,

input [31:0] dIn,

output [31:0] dOut );

reg [31:0] ram[63:0];

integer n;

//initialize ram to all FFs

initial

for (n=0; n<64; n=n+1)

ram[n] = 8'hFF;

assign dOut = ram[addr[31:2]];

always @(posedge clk)

if (we)

ram[addr[31:2]] = dIn;

Endmodule

**File 2. MIPS**

module mips\_top(

input clk, reset,

output memwrite,

output [ 3:0] top\_an,

output [ 7:0] top\_sseg,

input [ 7:0] switches,

output sinkBit);

wire [31:0] pc, instr, dataadr, writedata, readdata, dispDat;

wire clksec;

reg [ 7:0] reg\_hex1, reg\_hex0;

// Clock (1 second) to slow down the running of the instructions

clk\_gen top\_clk(.clk50MHz(clk), .reset(reset), .clksec(clksec));

// Instantiate processor and memories

mips mips (clksec, reset, pc, instr,

memwrite, dataadr, writedata, readdata, switches[4:0], dispDat);

imem imem (pc[7:2], instr);

dmem dmem (clk, memwrite, dataadr, writedata, readdata);

// Instantiate 7-seg LED display module

disp\_hex disp\_unit (.clk(clk), .reset(1'b0),

.hex3(reg\_hex1[7:4]), .hex2(reg\_hex1[3:0]),

.hex1(reg\_hex0[7:4]), .hex0(reg\_hex0[3:0]),

.dp\_in(4'b1111), .an(top\_an), .sseg(top\_sseg));

// contents displayed on the 7 segment LEDs depending on DIP switches 7:5

// 7:5 = 000: display PC & LSB of register selected by DIP switches 4:0

// 7:5 = 001: display PC & LSB of instr

// 7:5 = 010: display PC & LSB of dataadr

// 7:5 = 011: display PC & LSB of writedata

// 7:5 = 100: display PC & instr byte 0

// 7:5 = 101: display PC & instr byte 1

// 7:5 = 110: display PC & instr byte 2

// 7:5 = 111: display PC & instr byte 3

always @ (posedge clk) begin

reg\_hex1 = pc[7:0];

case ({switches[7],switches[6], switches[5]})

3'b000: begin

reg\_hex0 = dispDat[ 7:0];

end

3'b001: begin

reg\_hex0 = instr[ 7:0];

end

3'b010: begin

reg\_hex0 = dataadr[ 7:0];

end

3'b011: begin

reg\_hex0 = writedata[ 7:0];

end

3'b100: begin

reg\_hex0 = instr[ 7:0];

end

3'b101: begin

reg\_hex0 = instr[ 15:8];

end

3'b110: begin

reg\_hex0 = instr[ 23:16];

end

3'b111: begin

reg\_hex0 = instr[ 31:24];

end

endcase

end

//sink unused bit(s) to knock down the number of warning messages

assign sinkBit = (pc > 0) ^ (instr > 0) ^ (dataadr > 0) ^ (writedata > 0) ^

(readdata > 0) ^ (dispDat > 0);

endmodule

**File 3. MIPS TOP**

module testbench();

reg clk;

reg reset;

reg [7:0] switches;

wire [31:0] writedata, dataadr, pc, instr, dispDat, readdata;

wire memwrite;

// instantiate device to be tested

mips mips (clk, reset, pc, instr,

memwrite, dataadr, writedata, readdata, switches[4:0], dispDat);

imem imem (pc[7:2], instr);

dmem dmem (clk, memwrite, dataadr, writedata, readdata);

// initialize test

initial

begin

clk = 0;

reset <= 1;

#1;

clk = 1;

#1;

clk = 0;

reset <= 0;

#1;

end

// generate clock to sequence tests

always

begin

clk <= 1; # 5; clk <= 0; # 5;

end

endmodule

**File 4. Simple non self testing testbench**