**San Jose State University**

**Department of Computer Engineering**

**CMPE 200 Report**

**Assignment 6 Report**

**Title** Enhanced Single-cycle MIPS Processor

**Semester:** Fall 2023 **Date:** 11/4/2023

**by**

**Name:** Sharath Chandra Mummadi **SID** 017400019

**Name:** Jincy Jose Thottathil **SID:** 016903978

**Name:** Sai Kashyap Kurella **SID:** 016018925

**INTRODUCTION:**

This assignment outlines the extension of the single-cycle MIPS processor, originally designed in Lab #5. The objective was to enhance the processor's capabilities by incorporating additional MIPS instructions, namely MULTU, MFHI, MFLO, JR, JAL, SLL, and SLR, alongside the existing instructions (add, sub, and, or, slt, lw, sw, beq, j, and addi). The assignment necessitated functional verification testing to ensure the seamless integration and reliable performance of the extended functionalities. The provided initial MIPS design schematics and the test program (fatorial.asm) were utilized as the foundation for this exploration and validation process. This report details the design modifications, implementation strategies, testing methodologies, and challenges encountered during the enhancement of the processor's functionalities.

**DESIGN METHODOLOGY:**

An examination of the source code prompted modifications to the Control Unit and Datapath to accommodate specific instructions, including MULTU, MFHI, MFLO, JR, JAL, SLL, and SLR. For the MULTU, MFHI, and MFLO instructions, the system incorporated a multiplier module, high low register, and a multiplexer. During the execution of MFHI and MFLO instructions, a multiplexer was employed to enable the output of the multiplier(mult) to be rerouted back to the register file. The JR instruction was supported by integrating a multiplexer and introducing a new signal in the auxiliary decoder. The SLL and SLR instructions were facilitated through the use of the ALU and auxiliary decoder. Additionally, two extra multiplexers were added to the datapath to handle JAL instructions. These multiplexers, controlled by the main decoder in the control unit, allowed the transfer of the program counter's address into the register file. The modified parts of the Datapath in the extended MIPS microarchitecture are depicted in red in Figure 1.

A diagram of a computer

Description automatically generated

Figure 1. *Extended MIPS microarchitecture*

![A computer screen shot of a computer

Description automatically generated]()

Figure 2. *Complete Processor Schematic from Vivado*

Below are the block level diagrams of the control unit and data path of the extended processor.

![A computer screen shot of a computer

Description automatically generated]()

Figure 3. *Data Path with extended signals*

A computer screen shot of a diagram

Description automatically generated

Figure 4. *Control Unit with extended signals*

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Main Decoder** | | | | | | | | | | | |
| **Instruction** | **Op** | **branch** | **jump** | **reg\_dst** | **we\_reg** | **alu\_src** | **we\_dm** | **dm2reg** | **alu\_op** | **rf\_wd\_jal\_sel** | **rf\_wa\_jal\_sel** |
| R-Type | 00\_0000 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 10 | 0 | 0 |
| addi | 00\_1000 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 00 | 0 | 0 |
| beq | 00\_0100 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 01 | 0 | 0 |
| j | 00\_0010 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 00 | 0 | 0 |
| jal | 00\_0011 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 00 | 1 | 1 |
| sw | 10\_1011 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 00 | 0 | 0 |
| lw | 10\_0011 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 00 | 0 | 0 |

Shown below are the truth tables of the control unit modules. (i.e., Main Decoder and Auxiliary Decoder)

Figure 5. *Main decoder truth table of Control Unit*

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Auxilary Decoder** | | | | | | | | |
| **Instruction** | **funct** | **alu\_ctrl** | **pc\_jr\_sel** | **rf\_mult\_shift\_wd\_sel** | **mult\_we** | **mult\_hilo\_sel** | **mult\_shift\_sel** | **sl\_or\_sr** |
| and | 10\_0100 | 000 | 0 | 0 | 0 | 0 | 0 | 0 |
| or | 10\_0101 | 001 | 0 | 0 | 0 | 0 | 0 | 0 |
| add | 10\_0000 | 010 | 0 | 0 | 0 | 0 | 0 | 0 |
| sub | 10\_0010 | 110 | 0 | 0 | 0 | 0 | 0 | 0 |
| slt | 10\_1010 | 111 | 0 | 0 | 0 | 0 | 0 | 0 |
| multu | 01\_1000 | 000 | 0 | 0 | 1 | 0 | 0 | 0 |
| mfhi | 01\_0000 | 000 | 0 | 1 | 0 | 0 | 0 | 0 |
| mflo | 01\_0010 | 000 | 0 | 1 | 0 | 1 | 0 | 0 |
| sll | 00\_0000 | 000 | 0 | 1 | 0 | 0 | 1 | 0 |
| srl | 00\_0010 | 000 | 0 | 1 | 0 | 0 | 1 | 1 |
| jr | 00\_1000 | 000 | 1 | 0 | 0 | 0 | 0 | 0 |

Figure 6. *Auxiliary decoder truth table of Control Unit*

**FUNCTIONAL VERIFICATION AND SIMULATED WAVEFORMS:**

The designed extended processor was verified with the factorial mips assembly code by converting it into hexadecimal machine code and dumping it into the memory file (***memfile.dat***). Assembly program used for verifications is shown below.

A screenshot of a computer program

Description automatically generated

Figure 7. *MIPS Assembly factorial program used for verification.*

A screenshot of a computer

Description automatically generated

Figure 8. *Machine code dumped into memfile.dat memory file.*

![A computer screen shot of a computer program

Description automatically generated]()*.*

Figure 9. *Simulated waveforms of the Extended Processor running the factorial program.*

![A screenshot of a computer

Description automatically generated]()



Figure 10.  *waveforms of the Extended Processor displaying last executed instructions.*

**OBSERVATIONS FROM THE WAVEFORMS:**

The provided assembly program performs a **4!** which should result 24(0x00000018 in hexadecimal) which should be the output of ALU. This output should be driven back to register files and stored in the specified register. Clearly the designed extended mips processor worked as expected as marked in the above waveform snippet.

**COLLABORATION:**

We discussed about the extension that is to be provided to Assignment 5 to incorporate additional instructions required in this assignment. We discussed on additional modules and signals that is to be added in the control unit and datapath. We divided the work equally between three of us. We divided the diagrams, tables, simulation logs, testbench waveforms, schematics and source code. Jincy did the extended microarchitecture diagram. Kashyap and Sharath worked on Verilog coding to get the simulation done right. All the tables and vivado schematic was done by Kashyap. Report was done by all three of us together.

**CONCLUSION:**

In this assignment, we successfully extended the single-cycle MIPS processor's functionality by incorporating additional MIPS instructions. This expansion included operations such as MULTU, MFHI, MFLO, JR, JAL, SLL, and SLR, in addition to the existing instructions (add, sub, and, or, slt, lw, sw, beq, j, and addi).We performed functional verification to ensure the reliability and seamless integration of the extended functionalities. The provided initial MIPS design schematics and the test program (fatorial.asm) were instrumental in guiding our modifications and validating the new instruction set.

This assignment provided invaluable insights into processor design and testing, highlighting the significance of rigorous validation in ensuring the robustness of a processor. The successful extension of the processor's capabilities underscores the adaptability and versatility required in the field of digital design.

**APPENDIX:**

1. **Source Code**

**mips.v**

module mips (

input wire clk,

input wire rst,

input wire [4:0] ra3,

input wire [31:0] instr,

input wire [31:0] rd\_dm,

output wire we\_dm,

output wire [31:0] pc\_current,

output wire [31:0] alu\_out,

output wire [31:0] wd\_dm,

output wire [31:0] rd3,

// Outputs for simulation

output wire [31:0] mult\_result,

output wire [31:0] shift\_result,

output wire pc\_jr\_sel,

output wire rf\_mult\_shift\_wd\_sel,

output wire mult\_we,

output wire mult\_hilo\_sel,

output wire mult\_shift\_sel,

output wire sl\_or\_sr,

output wire rf\_wa\_jal\_sel,

output wire rf\_wd\_jal\_sel

);

wire branch;

wire jump;

wire reg\_dst;

wire we\_reg;

wire alu\_src;

wire dm2reg;

wire [2:0] alu\_ctrl;

// New wires Moved all control signals to output of this module to see in waveforms. To remove, uncomment below and remove outputs from module definition

// wire pc\_jr\_sel;

// wire rf\_mult\_shift\_wd\_sel;

// wire mult\_we;

// wire mult\_hilo\_sel;

// wire mult\_shift\_sel;

// wire sl\_or\_sr;

// wire rf\_wa\_jal\_sel;

// wire rf\_wd\_jal\_sel;

datapath dp (

.clk (clk),

.rst (rst),

.branch (branch),

.jump (jump),

.reg\_dst (reg\_dst),

.we\_reg (we\_reg),

.alu\_src (alu\_src),

.dm2reg (dm2reg),

.alu\_ctrl (alu\_ctrl),

.ra3 (ra3),

.instr (instr),

.rd\_dm (rd\_dm),

.pc\_current (pc\_current),

.alu\_out (alu\_out),

.wd\_dm (wd\_dm),

.rd3 (rd3),

.pc\_jr\_sel (pc\_jr\_sel),

.rf\_mult\_shift\_wd\_sel (rf\_mult\_shift\_wd\_sel),

.mult\_we (mult\_we),

.mult\_hilo\_sel (mult\_hilo\_sel),

.mult\_shift\_sel (mult\_shift\_sel),

.sl\_or\_sr (sl\_or\_sr),

.rf\_wa\_jal\_sel (rf\_wa\_jal\_sel),

.rf\_wd\_jal\_sel (rf\_wd\_jal\_sel),

.mult\_result (mult\_result),

.shift\_result (shift\_result)

);

controlunit cu (

.opcode (instr[31:26]),

.funct (instr[5:0]),

.branch (branch),

.jump (jump),

.reg\_dst (reg\_dst),

.we\_reg (we\_reg),

.alu\_src (alu\_src),

.we\_dm (we\_dm),

.dm2reg (dm2reg),

.alu\_ctrl (alu\_ctrl),

.pc\_jr\_sel (pc\_jr\_sel),

.rf\_mult\_shift\_wd\_sel (rf\_mult\_shift\_wd\_sel),

.mult\_we (mult\_we),

.mult\_hilo\_sel (mult\_hilo\_sel),

.mult\_shift\_sel (mult\_shift\_sel),

.sl\_or\_sr (sl\_or\_sr),

.rf\_wa\_jal\_sel (rf\_wa\_jal\_sel),

.rf\_wd\_jal\_sel (rf\_wd\_jal\_sel)

);

endmodule

**imem.v**

module imem (

input wire [5:0] a,

output wire [31:0] y

);

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

initial begin

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

end

assign y = rom[a];

endmodule

**dmem.v**

module dmem (

input wire clk,

input wire we,

input wire [5:0] a,

input wire [31:0] d,

output wire [31:0] q,

input wire rst

);

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

integer n;

initial begin

for (n = 0; n < 64; n = n + 1) ram[n] = 32'hFFFFFFFF;

end

always @ (posedge rst) begin

end

always @ (posedge clk, posedge rst) begin

if (rst) for (n = 0; n < 64; n = n + 1) ram[n] = 32'hFFFFFFFF;

else if (we) ram[a] <= d;

end

assign q = ram[a];

endmodule

**datapath.v**

module datapath (

input wire clk,

input wire rst,

input wire branch,

input wire jump,

input wire reg\_dst,

input wire we\_reg,

input wire alu\_src,

input wire dm2reg,

input wire [2:0] alu\_ctrl,

input wire [4:0] ra3,

input wire [31:0] instr,

input wire [31:0] rd\_dm,

// New inputs

input wire pc\_jr\_sel,

input wire rf\_mult\_shift\_wd\_sel,

input wire mult\_we,

input wire mult\_hilo\_sel,

input wire mult\_shift\_sel,

input wire sl\_or\_sr,

input wire rf\_wd\_jal\_sel,

input wire rf\_wa\_jal\_sel,

output wire [31:0] pc\_current,

output wire [31:0] alu\_out,

output wire [31:0] wd\_dm,

output wire [31:0] rd3,

// Outputs for simulation (to remove, remove from here, remove from mips.v outputs and port connect, mips\_top.v outputs and port connect, tb\_mips\_top.v

output wire [31:0] mult\_result,

output wire [31:0] shift\_result

);

wire [4:0] rf\_wa;

wire pc\_src;

wire [31:0] pc\_plus4;

wire [31:0] pc\_pre;

wire [31:0] pc\_next;

wire [31:0] sext\_imm;

wire [31:0] ba;

wire [31:0] bta;

wire [31:0] jta;

wire [31:0] alu\_pa;

wire [31:0] alu\_pb;

wire [31:0] wd\_rf;

wire zero;

// Newly added wires

wire [31:0] pc\_final;

wire [63:0] mult\_out;

wire [31:0] mfhi;

wire [31:0] mflo;

// wire [31:0] mult\_result; // Moving to output to display in simulation

// wire [31:0] shift\_result; // Moving to output to display in simulation

wire [31:0] mult\_shift\_result;

wire [31:0] rf\_wd\_jal\_mux\_out;

wire [31:0] mux\_alu\_dm\_out;

wire [4:0] rf\_wa\_mux\_out;

assign pc\_src = branch & zero;

assign ba = {sext\_imm[29:0], 2'b00};

assign jta = {pc\_plus4[31:28], instr[25:0], 2'b00};

// --- PC Logic --- //

dreg pc\_reg (

.clk (clk),

.rst (rst),

.d (pc\_final),

.q (pc\_current)

);

adder pc\_plus\_4 (

.a (pc\_current),

.b (32'd4),

.y (pc\_plus4)

);

adder pc\_plus\_br (

.a (pc\_plus4),

.b (ba),

.y (bta)

);

mux2 #(32) pc\_src\_mux (

.sel (pc\_src),

.a (pc\_plus4),

.b (bta),

.y (pc\_pre)

);

mux2 #(32) pc\_jmp\_mux (

.sel (jump),

.a (pc\_pre),

.b (jta),

.y (pc\_next)

);

// --- RF Logic --- //

mux2 #(5) rf\_wa\_mux (

.sel (reg\_dst),

.a (instr[20:16]),

.b (instr[15:11]),

.y (rf\_wa\_mux\_out)

);

regfile rf (

.clk (clk),

.we (we\_reg),

.ra1 (instr[25:21]),

.ra2 (instr[20:16]),

.ra3 (ra3),

.wa (rf\_wa),

.wd (wd\_rf),

.rd1 (alu\_pa),

.rd2 (wd\_dm),

.rd3 (rd3),

.rst (rst)

);

signext se (

.a (instr[15:0]),

.y (sext\_imm)

);

// --- ALU Logic --- //

mux2 #(32) alu\_pb\_mux (

.sel (alu\_src),

.a (wd\_dm),

.b (sext\_imm),

.y (alu\_pb)

);

alu alu (

.op (alu\_ctrl),

.a (alu\_pa),

.b (alu\_pb),

.zero (zero),

.y (alu\_out)

);

// --- MEM Logic --- //

mux2 #(32) rf\_wd\_mux (

.sel (dm2reg),

.a (alu\_out),

.b (rd\_dm),

.y (mux\_alu\_dm\_out)

);

// -- JR Logic --- //

mux2 #(32) pc\_jr\_mux (

.sel (pc\_jr\_sel),

.a (pc\_next),

.b (alu\_pa),

.y (pc\_final)

);

// -- MULTU, MFLO, MFHI Logic -- //

mult multu (

.a (alu\_pa),

.b (wd\_dm),

.y (mult\_out)

);

we\_dreg #(32) hi (

.d (mult\_out[63:32]),

.q (mfhi),

.we (mult\_we),

.clk (clk)

);

we\_dreg #(32) lo (

.d (mult\_out[31:0]),

.q (mflo),

.we (mult\_we),

.clk (clk)

);

mux2 #(32) mult\_hilo\_mux (

.sel (mult\_hilo\_sel),

.a (mfhi),

.b (mflo),

.y (mult\_result)

);

// -- SLL, SLR Logic -- //

shifter #(32) sll\_slr\_shifter (

.sl\_or\_sr (sl\_or\_sr),

.shamt (instr[10:6]),

.d (wd\_dm),

.y (shift\_result)

);

// -- MULTU or SHIFT MUX -- //

mux2 #(32) mult\_shift\_mux (

.sel (mult\_shift\_sel),

.a (mult\_result),

.b (shift\_result),

.y (mult\_shift\_result)

);

mux2 #(32) rf\_mult\_shift\_wd\_mux (

.sel (rf\_mult\_shift\_wd\_sel),

.a (rf\_wd\_jal\_mux\_out),

.b (mult\_shift\_result),

.y (wd\_rf)

);

// -- JAL Logic -- //

mux2 #(32) rf\_wd\_jal\_mux (

.sel (rf\_wd\_jal\_sel),

.a (mux\_alu\_dm\_out),

.b (pc\_plus4),

.y (rf\_wd\_jal\_mux\_out)

);

mux2 #(5) rf\_wa\_jal\_mux (

.sel (rf\_wa\_jal\_sel),

.a (rf\_wa\_mux\_out),

.b (5'd31),

.y (rf\_wa)

);

endmodule

**dmem.v**

module dmem (

input wire clk,

input wire we,

input wire [5:0] a,

input wire [31:0] d,

output wire [31:0] q,

input wire rst

);

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

integer n;

initial begin

for (n = 0; n < 64; n = n + 1) ram[n] = 32'hFFFFFFFF;

end

always @ (posedge rst) begin

end

always @ (posedge clk, posedge rst) begin

if (rst) for (n = 0; n < 64; n = n + 1) ram[n] = 32'hFFFFFFFF;

else if (we) ram[a] <= d;

end

assign q = ram[a];

endmodule

**controlunit.v**

module controlunit (

input wire [5:0] opcode,

input wire [5:0] funct,

output wire branch,

output wire jump,

output wire reg\_dst,

output wire we\_reg,

output wire alu\_src,

output wire we\_dm,

output wire dm2reg,

output wire [2:0] alu\_ctrl,

// New outputs

output wire pc\_jr\_sel,

output wire rf\_mult\_shift\_wd\_sel,

output wire mult\_we,

output wire mult\_hilo\_sel,

output wire mult\_shift\_sel,

output wire sl\_or\_sr,

output wire rf\_wd\_jal\_sel,

output wire rf\_wa\_jal\_sel

);

wire [1:0] alu\_op;

maindec md (

.opcode (opcode),

.branch (branch),

.jump (jump),

.reg\_dst (reg\_dst),

.we\_reg (we\_reg),

.alu\_src (alu\_src),

.we\_dm (we\_dm),

.dm2reg (dm2reg),

.alu\_op (alu\_op),

.rf\_wd\_jal\_sel (rf\_wd\_jal\_sel),

.rf\_wa\_jal\_sel (rf\_wa\_jal\_sel)

);

auxdec ad (

.alu\_op (alu\_op),

.funct (funct),

.alu\_ctrl (alu\_ctrl),

.pc\_jr\_sel (pc\_jr\_sel),

.rf\_mult\_shift\_wd\_sel (rf\_mult\_shift\_wd\_sel),

.mult\_we (mult\_we),

.mult\_hilo\_sel (mult\_hilo\_sel),

.mult\_shift\_sel (mult\_shift\_sel),

.sl\_or\_sr (sl\_or\_sr)

);

endmodule

**maindec.v**

module controlunit (

input wire [5:0] opcode,

input wire [5:0] funct,

output wire branch,

output wire jump,

output wire reg\_dst,

output wire we\_reg,

output wire alu\_src,

output wire we\_dm,

output wire dm2reg,

output wire [2:0] alu\_ctrl,

// New outputs

output wire pc\_jr\_sel,

output wire rf\_mult\_shift\_wd\_sel,

output wire mult\_we,

output wire mult\_hilo\_sel,

output wire mult\_shift\_sel,

output wire sl\_or\_sr,

output wire rf\_wd\_jal\_sel,

output wire rf\_wa\_jal\_sel

);

wire [1:0] alu\_op;

maindec md (

.opcode (opcode),

.branch (branch),

.jump (jump),

.reg\_dst (reg\_dst),

.we\_reg (we\_reg),

.alu\_src (alu\_src),

.we\_dm (we\_dm),

.dm2reg (dm2reg),

.alu\_op (alu\_op),

.rf\_wd\_jal\_sel (rf\_wd\_jal\_sel),

.rf\_wa\_jal\_sel (rf\_wa\_jal\_sel)

);

auxdec ad (

.alu\_op (alu\_op),

.funct (funct),

.alu\_ctrl (alu\_ctrl),

.pc\_jr\_sel (pc\_jr\_sel),

.rf\_mult\_shift\_wd\_sel (rf\_mult\_shift\_wd\_sel),

.mult\_we (mult\_we),

.mult\_hilo\_sel (mult\_hilo\_sel),

.mult\_shift\_sel (mult\_shift\_sel),

.sl\_or\_sr (sl\_or\_sr)

);

endmodule

**auxdec.v**

module auxdec (

input wire [1:0] alu\_op,

input wire [5:0] funct,

output wire [2:0] alu\_ctrl,

// New outputs

output wire pc\_jr\_sel,

output wire rf\_mult\_shift\_wd\_sel,

output wire mult\_we,

output wire mult\_hilo\_sel,

output wire mult\_shift\_sel,

output wire sl\_or\_sr

);

reg [8:0] ctrl;

assign {alu\_ctrl, pc\_jr\_sel, rf\_mult\_shift\_wd\_sel, mult\_we, mult\_hilo\_sel, mult\_shift\_sel, sl\_or\_sr} = ctrl;

always @ (alu\_op, funct) begin

case (alu\_op)

2'b00: ctrl = 9'b010\_0\_0\_0\_0\_0\_0; // ADD (LW or SW)

2'b01: ctrl = 9'b110\_0\_0\_0\_0\_0\_0; // SUB (BEQ)

default: case (funct)

6'b10\_0100: ctrl = 9'b000\_0\_0\_0\_0\_0\_0; // AND

6'b10\_0101: ctrl = 9'b001\_0\_0\_0\_0\_0\_0; // OR

6'b10\_0000: ctrl = 9'b010\_0\_0\_0\_0\_0\_0; // ADD

6'b10\_0010: ctrl = 9'b110\_0\_0\_0\_0\_0\_0; // SUB

6'b10\_1010: ctrl = 9'b111\_0\_0\_0\_0\_0\_0; // SLT

6'b01\_1001: ctrl = 9'b000\_0\_0\_1\_0\_0\_0; // NEW: MULTU

6'b01\_0000: ctrl = 9'b000\_0\_1\_0\_0\_0\_0; // NEW: MFHI

6'b01\_0010: ctrl = 9'b000\_0\_1\_0\_1\_0\_0; // NEW: MFLO

6'b00\_1000: ctrl = 9'b000\_1\_0\_0\_0\_0\_0; // NEW: JR

6'b00\_0000: ctrl = 9'b000\_0\_1\_0\_0\_1\_0; // NEW: SLL

6'b00\_0010: ctrl = 9'b000\_0\_1\_0\_0\_1\_1; // NEW: SLR

default: ctrl = 9'bxxx\_x;

endcase

endcase

end

endmodule

**dreg.v**

module dreg # (parameter WIDTH = 32) (

input wire clk,

input wire rst,

input wire [WIDTH-1:0] d,

output reg [WIDTH-1:0] q

);

always @ (posedge clk, posedge rst) begin

if (rst) q <= 0;

else q <= d;

end

endmodule

**adder.v**

module adder (

input wire [31:0] a,

input wire [31:0] b,

output wire [31:0] y

);

assign y = a + b;

endmodule

**mux2.v**

module mux2 #(parameter WIDTH = 8) (

input wire sel,

input wire [WIDTH-1:0] a,

input wire [WIDTH-1:0] b,

output wire [WIDTH-1:0] y

);

assign y = (sel) ? b : a;

endmodule

**regfile.v**

module regfile (

input wire clk,

input wire we,

input wire [4:0] ra1,

input wire [4:0] ra2,

input wire [4:0] ra3,

input wire [4:0] wa,

input wire [31:0] wd,

output wire [31:0] rd1,

output wire [31:0] rd2,

output wire [31:0] rd3,

input wire rst

);

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

integer n;

initial begin

for (n = 0; n < 32; n = n + 1) rf[n] = 32'h0;

rf[29] = 32'h100; // Initialze $sp

end

always @ (posedge clk, posedge rst) begin

if (rst) begin

for (n = 0; n < 32; n = n + 1) rf[n] = 32'h0;

rf[29] = 32'h100; // Initialze $sp

end

else if (we) rf[wa] <= wd;

end

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

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

assign rd3 = (ra3 == 0) ? 0 : rf[ra3];

endmodule

**signext.v**

module signext (

input wire [15:0] a,

output wire [31:0] y

);

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

endmodule

**alu.v**

module alu (

input wire [2:0] op,

input wire [31:0] a,

input wire [31:0] b,

output wire zero,

output reg [31:0] y

);

assign zero = (y == 0);

always @ (op, a, b) begin

case (op)

3'b000: y = a & b;

3'b001: y = a | b;

3'b010: y = a + b;

3'b110: y = a - b;

3'b111: y = (a < b) ? 1 : 0;

endcase

end

endmodule

**mult.v**

module mult (

input wire [31:0] a, b,

output wire [63:0] y

);

assign y = a \* b;

endmodule

**we\_dreg.v**

module we\_dreg # (parameter WIDTH = 32) (

input wire clk,

input wire we,

input wire [WIDTH-1:0] d,

output reg [WIDTH-1:0] q

);

always @ (posedge clk) begin

if (we) q <= d;

else q <= q;

end

endmodule

**shifter.v**

module shifter # (parameter WIDTH = 32) (

input wire [4:0] shamt,

input wire sl\_or\_sr,

input wire [WIDTH-1:0] d,

output reg [WIDTH-1:0] y

);

always @ (shamt, sl\_or\_sr, d) begin

if (sl\_or\_sr) y = d >> shamt;

else y = d << shamt;

end

endmodule

**mips\_top.v**

module mips\_top (

input wire clk,

input wire rst,

input wire [4:0] ra3,

output wire we\_dm,

output wire [31:0] pc\_current,

output wire [31:0] instr,

output wire [31:0] alu\_out,

output wire [31:0] wd\_dm,

output wire [31:0] rd\_dm,

output wire [31:0] rd3,

// Outputs for simulation

output wire [31:0] mult\_result,

output wire [31:0] shift\_result,

output wire pc\_jr\_sel,

output wire rf\_mult\_shift\_wd\_sel,

output wire mult\_we,

output wire mult\_hilo\_sel,

output wire mult\_shift\_sel,

output wire sl\_or\_sr,

output wire rf\_wa\_jal\_sel,

output wire rf\_wd\_jal\_sel

);

wire [31:0] DONT\_USE;

mips mips (

.clk (clk),

.rst (rst),

.ra3 (ra3),

.instr (instr),

.rd\_dm (rd\_dm),

.we\_dm (we\_dm),

.pc\_current (pc\_current),

.alu\_out (alu\_out),

.wd\_dm (wd\_dm),

.rd3 (rd3),

// For simulation

.mult\_result (mult\_result),

.shift\_result (shift\_result),

.pc\_jr\_sel (pc\_jr\_sel),

.rf\_mult\_shift\_wd\_sel (rf\_mult\_shift\_wd\_sel),

.mult\_we (mult\_we),

.mult\_hilo\_sel (mult\_hilo\_sel),

.mult\_shift\_sel (mult\_shift\_sel),

.sl\_or\_sr (sl\_or\_sr),

.rf\_wa\_jal\_sel (rf\_wa\_jal\_sel),

.rf\_wd\_jal\_sel (rf\_wd\_jal\_sel)

);

imem imem (

.a (pc\_current[7:2]),

.y (instr)

);

dmem dmem (

.clk (clk),

.we (we\_dm),

.a (alu\_out[7:2]),

.d (wd\_dm),

.q (rd\_dm),

.rst (rst)

);

endmodule