CSC/CpE 142

Term project

Phase 2

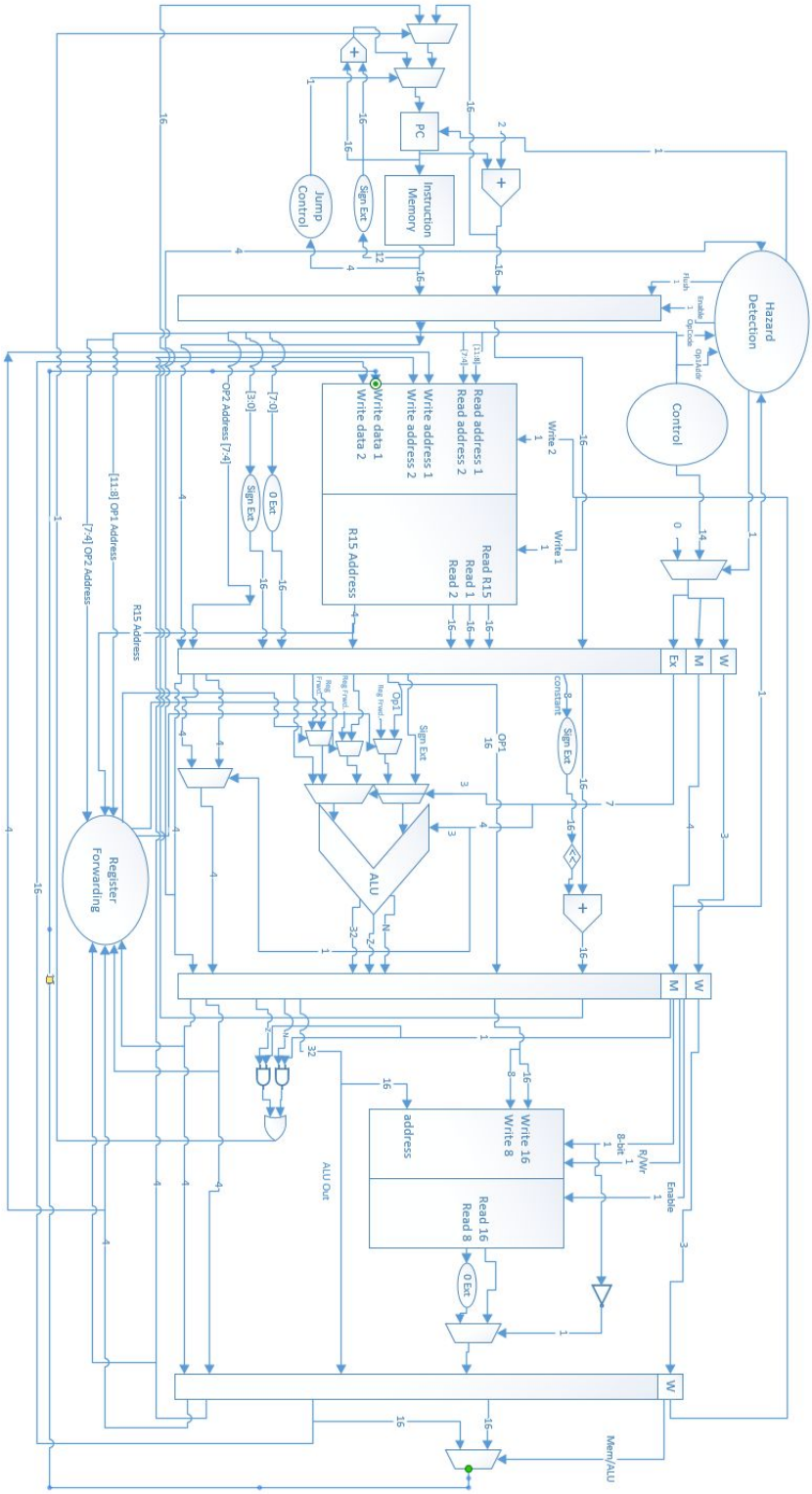
Andrew Robertson & Ehsan Halterman
(50% contribution from each)

Table of contents

Status report

Currently it seems instructions that don't require hazard detection or register forwarding are working. Our strategy was to get the essentials working first then fix register forwarding and hazard detection. The final step was going to be to improve branch performance by moving it to the ID stage and introducing a comparator. All submodules appear to be working on their own - their testbenches and screen captures of them functioning will be included with the digital files.

Diagram


PC
Instruction Memory
Jump Control
Sign Ext
Hazard Detection
Control
Write data 1
Write data 2
Write address 1
Write address 2
Read address 1
Read address 2
R15 Address
Read R15
Read 1
Read 2
Write 1
Write 2
OP2 Address [7:4]
[3:0]
[7:0]
Sign Ext
0 Ext
[11:8] OP1 Address
[7:4] OP2 Address
R15 Address
Ex
M
W
Op1
Reg Fwd
constant
Sign Ext
OP1
ALU
N
Z
Register Forwarding
address
Write 16
Write 8
Read 16
Read 8
R/Wr
8-bit
Enable
0 Ext
ALU Out
Mem/ALU

Control logic

Instruction	Jump control	W			M				EX						
add	1	1	1	0	1	x	x	1	1	0	0	0	0	1	x
sub	1	1	1	0	1	x	x	1	1	0	0	1	0	1	x
mul	1	1	0	0	1	x	x	1	1	0	1	0	0	1	1
div	1	1	0	0	1	x	x	1	1	0	1	1	0	1	1
mv	1	1	1	0	1	x	x	1	1	1	1	1	0	1	x
swp	1	1	0	0	1	x	x	1	1	1	0	0	0	1	0
andi	1	1	1	0	1	x	x	1	1	1	0	1	1	0	x
or	1	1	1	0	1	x	x	1	1	1	1	0	1	0	x
lbu	1	0	1	0	1	0	1	0	0	0	0	0	0	1	x
sb	1	1	1	1	1	0	0	0	0	0	0	0	0	1	x
lw	1	0	1	0	1	1	1	0	0	0	0	0	0	1	x
sw	1	1	1	1	1	1	0	0	0	0	0	0	0	1	x
blt	1	1	1	1	0	x	x	1	1	0	0	1	0	0	x
bgt	1	1	1	1	0	x	x	1	1	0	0	1	0	0	x
beq	1	1	1	1	0	x	x	1	1	0	0	1	0	0	x
jmp	0	1	1	1	1	x	x	1	x	x	x	x	x	x	x
halt	1	1	1	1	1	x	x	1	x	x	x	x	x	x	x
	(Active low)	Memoral u	Write 2 (activelow)	Write 1 (activelow)	Branch (activelow)	Eightbit (activelow)	R/~WR (activelow)	Memenable (activelow)	signext	ALU			Secondin		WBop2

	ALU		
add	0	0	0
sub	0	0	1
mul	0	1	0
div	0	1	1
swp	1	0	0
and	1	0	1
or	1	1	0
OP2	1	1	1

Hazard detect:

If lw destination address = source address of next instruction

Deassert control, pc, IF buffer

Else

Normal operation

Reg forward:

		MUX A
If D/EX OP1 addr matches (& being written)	EX/M OP1 addr	001
	M/WB OP1 addr	010
	EX/M OP2 addr	011
	M/WB OP2 addr	100
	EX/M R15 addr	101
	M/WB R15 addr	110
	O.W.	000

MUX B

If D/EX OP2 addr matches (& being written)	EX/M OP1 addr	001
	M/WB OP1 addr	010
	EX/M OP2 addr	011
	M/WB OP2 addr	100
	EX/M R15 addr	101
	M/WB R15 addr	110
	O.W.	000

MUX C

If D/EX R15 addr matches (& being written)	EX/M OP1 addr	001
	M/WB OP1 addr	010
	EX/M OP2 addr	011
	M/WB OP2 addr	100
	EX/M R15 addr	101
	M/WB R15 addr	110
	O.W.	000

Test assembly program

```
ADD R1, R2
SUB R1, R2
ORi R3, FF
ANDi R3, 4C
MUL R5, R6
DIV R1, R5
SUB R15, R15
MOV R4, R8
SWP R4, R6
ORi R4, 2
LBU R6, 4(R9)
SB  R6, 6(R9)
LW R6, 6(R9)
```

```
LW R6, 6(R9)
BEQ R7, 4
ADD R11, R1
BLT R7, 5
ADD R11, R2
BGT R7, 2
ADD R1, R1
ADD R1, R1
LW R8, 0(R9)
ADD R8, R8
SW R8, 2 (R9)
LW R10, 2 (R9)
ADD R12, R12
SUB R13, R13
ADD R12, R13
HALT
```

(Read top to bottom left column followed by the right)
Following the above code, these are the results per line, d = decimal, h = hex:
R1 = 3920d or 0F50h
R1 = 3840d or
R3 = 255d
R3 = 76
R5 = 9980h, R15 = 0019h
R15 = 0, R1 = 0
R15 = 0
R4 = 65399d
R4 = 26214d , R6 = 65399d
R4 = 26214d
R6 = 4930d
Data memory 6h = 66d
R6 = 66d

Branch not taken
R11 = 0
Branch not taken
R11 = 80d
Branch taken , PC = (PC+2)+2(offset)
R1 = 0
R8 = 15068d
R8 = 30136d
Mem location 2 = 30136d
R10 = 30136d
R12 = 39186d
R13 = 0
R12 = 39186d
FINISH

Stimulus module

```
// Top level stimulus module

`include "cpu.v"

module cpu_fixture;

// Declare variables for stimulating input
reg clk, reset;

initial
        $vcdpluson;

initial
//       $monitor($time, " clk = %b reset = %b", clk, reset);
                        $monitor($time, "\n",
//"\npre ifid buff outputs -- IM: %b\tPC: %d\thaztopc: %b\nleftmuxsel: %b\trightmuxsel: %b\n\n",
//uut.imout, uut.pcout, uut.haztopc, uut.muxenresult, uut.IFjumpout,

"////////// IFID buffer output /////////\n\ninstruction: %b\tpc+2: %d\n\n",
uut.imset, uut.IDpc,

//"pre idex buff outputs -- r15: %d\top1: %d\top2:%d\n\n",
//uut.reg15, uut.read1, uut.read2,

"////////// IDEX buffer output /////////\npc: %d\tcontrol: %b\nR15: %d\tOP1: %d\tOP2: %d\nop1
address: %b\top2 address: %b\n\n",
uut.adda, uut.muxcontrolbuffout,uut.r15, uut.op1, uut.op2, uut.op1address, uut.op2address,

//"pre exmem buff outputs -- adder: %d\n\n",
//uut.exaddout,

"////////// EXMEM buffer output ////////\ncontrol: %b\tpc: %d\nALU output: %b\nALU dec high:
%d\nALU dec low: %d\nnflag: %b\tzflag: %b\nmemory write 16: %d\tmemory write 8: %d\nwrite
1 address: %b\twrite 2 address: %b\n\n",
{uut.writethrough, uut.memenable, uut.memR_WR, uut.eightbit, uut.branchenable},
uut.Mempc, uut.ALUout, uut.ALUout[31:16], uut.ALUout[15:0],uut.nflag, uut.zflag, uut.write16,
uut.write16[7:0],
uut.memwriteadd1, uut.memwriteadd2,
```

```
"////////// MEMWB buffer output ////////\ncontrol: %b\nALU output: %b\nmemory read 16:
%d\nwrite 1 address: %b\twrite 2 address: %b\n\n",
{uut.wbenable1, uut.wbenable2, uut.memalu}, uut.aludata, uut.memdata,
uut.writeadd1, uut.writeadd2);

// Instantiate the design block
cpu uut(clk, reset);

// Stimulate the Clock Signal
always
#5 clk = ~clk;

initial
begin
clk = 1'b0 ; reset = 1'b0;
#7 reset = 1'b1;
end

// Finish the simulation at time 100
initial
begin
  #300 $finish;
end

endmodule
```

Source code

CPU

```
`include "termproj41mux.v"
`include "termprojadder.v"
`include "termprojALU.v"
`include "termprojbuffone.v"
`include "termprojdatamem.v"
`include "termprojhazdet.v"
`include "termprojinstrmem.v"
`include "termprojjumpcontrol.v"
`include "termprojleftshift.v"
`include "termprojpathcontrol.v"
`include "termprojpc.v"
`include "termprojregblock.v"
`include "termprojregforward.v"
`include "termprojeightonemux.v"
`include "termprojzeroext.v"
`include "termprojtwoonemux.v"
`include "termprojsignext.v"

module cpu(input clk, reset);

//wires
wire            haztopc, haztobuff, wbenable1, wbenable2;
reg             muxenresult;
wire [15:0] Mempc;
wire [15:0] memdata, writedata1;
wire [31:0] aludata;
wire            memalu;
wire [15:0] muxtopcmux, muxtopc, pcout, imout, IFextout, IFaddtopout, IFaddbotout;
wire            IFjumpout;
wire            memR_WR, memenable, eightbit, branchenable, zflag, nflag;
wire [15:0] IDpc, imset;
wire [3:0]      writeadd1, writeadd2, reg15address;
wire [15:0] reg15, read1, read2, typebextout, zeroregout;
wire            zerocontrol;
wire [13:0] muxcontrolout;
wire [13:0] muxcontrolbuffout;
wire [3:0]      r15address, op2address, op1address;
```

```
wire [15:0] input1, input2, yhigh, ylow, alusignext, signextalu, op1regforward, exsignout,
                        r15regforward, op2regforward, exzeroext, adda, addb, exaddout,op1,
op2, r15;
wire            neg, zero,input1sel, input2sel, wbsel, wbsel2, wbsel3;
wire [2:0]      alusel, op1sel, op2sel, r15sel;
wire [7:0]      exsignin;
wire [31:0] ALUout;
wire [3:0]      memwriteadd1, memwriteadd2;
wire [15:0] write16, read16, read8ext, memmuxout, branchtarget;
wire [2:0]      writethrough;
wire [7:0]      read8;
wire [3:0]      memwrite1, wbwrite1, memwrite2, wbwrite2, muxaddout;
wire [13:0] control;



//IF////////////////////////////////////////////////////////////////////////////////////////////////

adder IFaddtop(16'd2, pcout, IFaddtopout);

always@*
muxenresult = 1'b0 ;//branch not doing well((!zflag&!branchenable)|(!nflag&!branchenable));

twoonemux #(16) IFleft(IFaddtopout, Mempc, muxenresult, muxtopcmux);

twoonemux #(16) IFright(muxtopcmux, IFaddbotout, IFjumpout, muxtopc);

adder IFaddbot(IFextout,pcout,IFaddbotout);

signext #(12) IFsigned(imout[11:0],IFextout);

jumpcontrol j1(imout[15:12],IFjumpout);

instrmem i1(pcout, 16'd0, clk, 1'b1, 1'b0, reset, imout);

pc p1(clk, reset, 1'b0,muxtopc, pcout); //haztopc temp removed, forcing enable

buffone #(32) IFID (clk, 1'b0, 1'b1, {IFaddtopout , imout}, {IDpc, imset}); //haztobuff temp
removed, forcing enable



//ID////////////////////////////////////////////////////////////////////////////////////////////////
```

```
regblock r1(imset[11:8], imset[7:4], writeadd1, writeadd2,writedata1, aludata[31:16],
                    clk, wbenable1, wbenable2, reset, reg15, read1, read2, reg15address);

zeroext #(8) z1(imset[7:0], zeroregout);

signext #(4) s1(imset[3:0], typebextout);

twoonemux #(14) t1(control ,14'd0 ,zerocontrol , muxcontrolout);

buffone #(130) IDEX (clk, 1'b0, 1'b1, {imset[7:0],typebextout, zeroregout, imset[7:4], imset[11:8],
muxcontrolout, IDpc, reg15, read1, read2, reg15address}, {exsignin,signextalu,
exzeroext,op2address, op1address, muxcontrolbuffout, adda, r15, op1, op2, r15address});



//EX////////////////////////////////////////////////////////////////////////////////////////////////

ALU a1(input1, input2, muxcontrolbuffout[5:3], yhigh, ylow, neg, zero);

twoonemux #(16) t2(signextalu, op1regforward, muxcontrolbuffout[6],input1);

fouronemux f1(r15regforward,op2regforward, exzeroext,16'b0,muxcontrolbuffout[2:1],input2);

twoonemux #(4) t3(op2address, r15address, muxcontrolbuffout[0], muxaddout);

adder a2(adda, addb, exaddout);

leftshift l1(exsignout,addb);

signext #(8) s2(exsignin,exsignout);

eightonemux e1(op1,
ALUout[15:0],writedata1,ALUout[31:16],aludata[15:0],ALUout[31:16],aludata[15:0],op1,op1sel,o
p1regforward);

eightonemux e2(r15,
ALUout[15:0],writedata1,ALUout[31:16],aludata[15:0],ALUout[31:16],aludata[15:0],r15,r15sel,r1
5regforward);

eightonemux e3(op2,
ALUout[15:0],writedata1,ALUout[31:16],aludata[15:0],ALUout[31:16],aludata[15:0],op2,op2sel,o
p2regforward);
```

```
buffone #(82) EXM (clk, 1'b0, 1'b1, {wbsel,muxcontrolbuffout[13:7], exaddout, op1, neg, zero,
yhigh, ylow, muxaddout, op1address}, {wbsel2,writethrough, memenable,
                                        memR_WR, eightbit, branchenable, Mempc, write16, zflag,
nflag, ALUout, memwriteadd2, memwriteadd1});



//MEM////////////////////////////////////////////////////////////////////////////////////////////

datamem d3(ALUout[15:0], write16, write16[7:0], clk, memR_WR, memenable, eightbit, reset,
read16, read8);

twoonemux #(16) memmux(read16, read8ext,!eightbit, memmuxout);

zeroext #(8) z2(read8, read8ext);

buffone #(60) MW (clk, 1'b0, 1'b1, {wbsel2, writethrough, memmuxout, ALUout, memwriteadd2,
memwriteadd1}, {wbsel3, wbenable1, wbenable2, memalu, memdata, aludata, writeadd2,
writeadd1});



//WB////////////////////////////////////////////////////////////////////////////////////////////

twoonemux #(16) wb(memdata, aludata[15:0], memalu, writedata1);



//Hazard////////////////////////////////////////////////////////////////////////////////////////

pathcontrol p2(imset[15:12], imset[3:0], control);

regforward r2(r15address, op1address, op2address, memwriteadd1,
                                writeadd1, memwriteadd2, writeadd2,wbsel2, wbsel3, op1sel, op2sel,
r15sel);

hazdet h1(op1address, imset[11:8], imset[7:4], muxcontrolbuffout[1], haztobuff, zerocontrol,
haztopc);


endmodule
```

4-1 mux

```
module fouronemux(input [15:0] a,b,c,d,
                                input [1:0] sel,
                                output reg [15:0] y);

always @*
        begin
                case (sel)
                        2'd0 :  y = a;
                        2'd1 :  y = b;
                        2'd2 :  y = c;
                        2'd3 :  y = d;
                endcase
        end

endmodule
```

Adder

```
module adder(input signed [15:0] a,b,
                                output reg signed [15:0] y);

always @*
        y = a+b;
endmodule
```

ALU

```
module ALU(input signed [15:0] a, b,
        input [2:0] s,
                output reg signed [15:0] yhigh, ylow,
                output reg n,z);

        always@*
        begin
                case (s)
                        3'd0 :
                                begin
                                yhigh = 16'd0;
                                ylow = a + b;
```

```
                end
        3'd1 :
                begin
                yhigh = 16'd0;
                ylow = a - b;
                end
        3'd2 :
                {yhigh,ylow} = a*b;
        3'd3 :
                begin
                yhigh = a % b;
                ylow = a / b;
                end
        3'd4 :
                begin
                ylow = b;
                yhigh = a;
                end
        3'd5 :
                begin
                yhigh = 16'd0;
                ylow = a & b;
                end
        3'd6 :
                begin
                yhigh = 16'd0;
                ylow = a | b;
                end
        3'd7 :
                begin
                yhigh = 16'd0;
                ylow = b;
                end
    endcase

    if (yhigh==16'd0 && ylow==16'd0) z=1'b0;        //active low
    else z=1'b1;

    if (ylow<0) n=1'b0;                                     //active low
    else n=1'b1;

  end
endmodule
```

Buffer

```
module buffone #(parameter SIZE = 134) (input clk, enable, reset, input [SIZE-1:0] read_in,
                                        output reg [SIZE-1:0] data_out);

reg [SIZE-1:0] data_in;

        always @(posedge clk)
        begin
                if (!enable) data_in <= read_in;
        end

        always @(*)
        begin

                        if (!reset) data_in <= {SIZE{1'b0}};

                data_out <= data_in;
        end

endmodule
```

Data memory

```
module datamem(input [15:0]address, write16,
                                    input [7:0] write8,
                                    input               clk, R_WR, enable, eightbit, reset,
                                    output reg [15:0] read16,
                                    output reg [7:0]  read8);

//data block

reg [7:0] data [511:0];
integer mufasa;
//write rising edge

always @(posedge clk)
begin
        if (!enable & !R_WR)
        begin
```

```
                if(!eightbit) data[(address%128)] <= write8;
                else {data[(address%128)] , data[(address%128 + 1)]} <= write16;
        end
end

//read falling edge

always @*
begin
        if (!enable & R_WR)
        begin
                read16 <= {data[(address%128)] , data[(address%128 + 1)]};
                read8 <= data[(address%128)];
        end
end

always @*
begin
        if (!reset)
                for(mufasa = 0; mufasa<128; mufasa = mufasa +1)
                begin
                        data[mufasa] <= 0;
                end

                data[(16'h0000)] <= 8'h3A;
                data[(16'h0001)] <= 8'hDC;
                data[(16'h0002)] <= 8'h00;
                data[(16'h0003)] <= 8'h00;
                data[(16'h0004)] <= 8'h13;
                data[(16'h0005)] <= 8'h42;
                data[(16'h0006)] <= 8'hAD;
                data[(16'h0007)] <= 8'hDE;
                data[(16'h0008)] <= 8'hEF;
                data[(16'h0009)] <= 8'hBE;
                data[(16'h000A)] <= 8'hFF;
                data[(16'h000B)] <= 8'hFF;
                data[(16'h000E)] <= 8'hAA;
                data[(16'h000F)] <= 8'hAA;
end

endmodule
```

8-1 mux

```
module eightonemux(input [15:0] a,b,c,d,e,f,g,h,
                                input [2:0] sel,
                                output reg [15:0] y);

always @*
        begin
                case (sel)
                        3'd0 :  y = a;
                        3'd1 :  y = b;
                        3'd2 :  y = c;
                        3'd3 :  y = d;
                        3'd4 :  y = e;
                        3'd5 :  y = f;
                        3'd6 :  y = g;
                        3'd7 :  y = h;
                endcase
        end

endmodule
```

Hazard detection

```
module hazdet(input [3:0] EXwriteAddr, DEop1Addr, DEop2Addr,
                        input EXreadbit,
                        output reg fetchbuffenable, zerocontrol, pcenable);

reg test;
always @*
begin
if (!EXreadbit & ((EXwriteAddr == DEop1Addr) |
            (EXwriteAddr == DEop2Addr))) test = 1'b1;
else test = 1'b0;
end

always@*
begin
case (test)
        1'b0:
        begin
```

```
                fetchbuffenable = 1'b0;         //enable (active low)
                zerocontrol  = 1'b0;            //enable (active low)
                pcenable  = 1'b0;                       //enable (active low)
        end
        1'b1:
        begin
                fetchbuffenable = 1'b1;         //disable (active low)
                zerocontrol  = 1'b1;            //disable (active low)
                pcenable  = 1'b1;                       //disable (active low)
        end
        1'bx:
        begin
                fetchbuffenable = 1'b0;         //enable (active low)
                zerocontrol  = 1'b0;            //enable (active low)
                pcenable  = 1'b0;                       //enable (active low)
        end
endcase
end

endmodule
```

Instruction memory

```
module instrmem(input [15:0]address, setup,
                                input           clk, R_WR, enable, reset,
                                output reg [15:0] instruction);

//data block

reg [7:0] data [511:0];

//write rising edge

always @(posedge clk)
begin
        if (!enable & !R_WR)
                {data[(address%128)] , data[(address%128 + 1)]} <= setup;
end

//read falling edge
```

```
always @(*)
begin
	if (!enable & R_WR)
	begin
		instruction <= {data[(address%128)] , data[(address%128 + 1)]};
	end
end

always @(negedge reset)
begin
	if (!reset)


			data[(16'h0000)] <= 8'h01;
			data[(16'h0001)] <= 8'h20;
			data[(16'h0002)] <= 8'h01;
			data[(16'h0003)] <= 8'h21;
			data[(16'h0004)] <= 8'h23;
			data[(16'h0005)] <= 8'hFF;
			data[(16'h0006)] <= 8'h13;
			data[(16'h0007)] <= 8'h4C;
			data[(16'h0008)] <= 8'h05;
			data[(16'h0009)] <= 8'h64;
			data[(16'h000A)] <= 8'h01;
			data[(16'h000B)] <= 8'h58;
			data[(16'h000C)] <= 8'h0F;
			data[(16'h000D)] <= 8'hF1;
			data[(16'h000E)] <= 8'h04;
			data[(16'h000F)] <= 8'h8E;
			data[(16'h0010)] <= 8'h04;
			data[(16'h0011)] <= 8'h6F;
			data[(16'h0012)] <= 8'h24;
			data[(16'h0013)] <= 8'h02;
			data[(16'h0014)] <= 8'h86;
			data[(16'h0015)] <= 8'h94;
			data[(16'h0016)] <= 8'h96;
			data[(16'h0017)] <= 8'h96;
			data[(16'h0018)] <= 8'hA6;
			data[(16'h0019)] <= 8'h96;
			data[(16'h001A)] <= 8'h67;
			data[(16'h001B)] <= 8'h04;
			data[(16'h001C)] <= 8'h0B;
			data[(16'h001D)] <= 8'h10;
```

```
                data[(16'h001E)] <= 8'h47;
                data[(16'h001F)] <= 8'h02;
                data[(16'h0020)] <= 8'h0B;
                data[(16'h0021)] <= 8'h20;
                data[(16'h0022)] <= 8'h57;
                data[(16'h0023)] <= 8'h02;
                data[(16'h0024)] <= 8'h01;
                data[(16'h0025)] <= 8'h10;
                data[(16'h0026)] <= 8'h01;
                data[(16'h0027)] <= 8'h10;
                data[(16'h0028)] <= 8'hA8;
                data[(16'h0029)] <= 8'h90;
                data[(16'h002A)] <= 8'h08;
                data[(16'h002B)] <= 8'h80;
                data[(16'h002C)] <= 8'hB8;
                data[(16'h002D)] <= 8'h92;
                data[(16'h002E)] <= 8'hAA;
                data[(16'h002F)] <= 8'h92;
                data[(16'h0030)] <= 8'h0B;
                data[(16'h0031)] <= 8'hB0;
                data[(16'h0032)] <= 8'h0D;
                data[(16'h0033)] <= 8'hD1;
                data[(16'h0034)] <= 8'h0B;
                data[(16'h0035)] <= 8'hC0;
                data[(16'h0036)] <= 8'hF0;
                data[(16'h0037)] <= 8'h00;
end

endmodule
```

Jump control

```
module jumpcontrol(input [3:0] opcode,
                   output reg jump);

always @*
begin
        if (opcode == 4'b1100) jump=1'b1;
        else jump=1'b0;
end
endmodule
```

Left shift

```
module leftshift(input [15:0] a,
                                output reg [15:0] y);

always @*
        y = {a[13:0],2'b0};
endmodule
```

Path control

```
module pathcontrol(input [3:0] opcode, functcode,
                   output reg [13:0] control);

        always@*
        begin
                case (opcode)
                        4'd0 :
                begin
                if (functcode == 4'b0000) control = 14'b11011111000010;       //add
                else if (functcode == 4'b0001) control = 14'b11011111001010;  //sub
                else if (functcode == 4'b0100) control = 14'b10011111010011;  //mul
                else if (functcode == 4'b1000) control = 14'b10011111011011;  //div
                 else if (functcode == 4'b1110) control = 14'b11011111111011;  //mv
                else if (functcode == 4'b1111) control = 14'b10011111100010;  //swp
                else control = 14'bxxxxxxxxxxxxxx;
                end
                        4'd1 :
      control = 14'b11011111101101;       //andi
                        4'd2 :
      control = 14'b11011111110101;       //ori
                        4'd8 :
      control = 14'b11011111000010;       //lbu
                        4'd9 :
      control = 14'b11110000000011;       //sb
                        4'd10 :
      control = 14'b01011100000011;       //lw
                        4'd11 :
      control = 14'b11111000000011;       //sw
                        4'd4 :
      control = 14'b11101111001001;       //blt
```

```
                        4'd5 :
        control = 14'b11101111001001;       //bgt
                        4'd6 :
        control = 14'b11101111001001;       //beq
                        4'd12 :
        control = 14'b11011111000010;       //jmp
                        4'd15 :
        control = 14'b11111111111111;       //halt
                        default:
                                control = 14'bxxxxxxxxxxxxxx;
                endcase
        end
endmodule
```

PC

```
module pc (input clk, clr, load, input [15:0] da, output reg [15:0] qa);

always@(negedge clr or  posedge clk)
begin
   if(!clr) qa <= 0;
   else if (!load)
      qa <= da;
end

endmodule
```

Register block

```
module regblock(input [3:0] readaddress1, readaddress2,
                                                writeaddress1, writeaddress2,
                        input [15:0]writedata1, writedata2,
                        input           clk, enable1, enable2, reset,
                        output reg [15:0] reg15, read1, read2,
                        output reg [3:0]  reg15address);

//data block

reg [15:0] registers [15:0];
```

```
//write rising edge

always @(posedge clk)
begin
	if (!enable1) registers[writeaddress1] <= writedata1;
	if (!enable2) registers[writeaddress2] <= writedata2;
end

//read falling edge

always @*
begin
	reg15 <= registers[15];
	read1 <= registers[readaddress1];
	read2 <= registers[readaddress2];
	reg15address <= 4'b1111;
end

always @*
begin
	if (!reset)


		registers[(0)] <= 16'd0;
		registers[(1)] <= 16'h0F00;
		registers[(2)] <= 16'h0050;
		registers[(3)] <= 16'hFF0F;
		registers[(4)] <= 16'hF0FF;
		registers[(5)] <= 16'h0040;
		registers[(6)] <= 16'h6666;
		registers[(7)] <= 16'h00FF;
		registers[(8)] <= 16'hFF77;
		registers[(9)] <= 16'h0000;
		registers[(10)] <=16'h0000;
		registers[(11)] <=16'h0000;
		registers[(12)] <= 16'hCC89;
		registers[(13)] <= 16'h0002;
		registers[(14)] <= 16'h0000;
		registers[(15)] <= 16'h0000;
end

endmodule
```

Register forward

```
module regforward (input [3:0]                  EXr15, EXOP1, EXOP2, memwrite1,
                                                                    wbwrite1, memwrite2,
wbwrite2,
                                        input                   memmux, wbmux,
        output reg [2:0] MUXA, MUXB, MUXC);

always @ *
begin



//MUXA
if (EXOP1 == memwrite1) MUXA = 3'b001;
else if (EXOP1 == wbwrite1) MUXA = 3'b010;
else if (EXOP1 == memwrite2)
        begin
          if(memmux) MUXA = 3'b101;
          else MUXA = 3'b011;
        end
else if (EXOP1 == wbwrite2)
        begin
          if(wbmux) MUXA = 3'b110;
          else MUXA = 3'b100;
        end
else MUXA = 0;



//MUXB
if (EXOP2 == memwrite1) MUXB = 3'b001;
else if (EXOP2 == wbwrite1) MUXB = 3'b010;
else if (EXOP2 == memwrite2)
        begin
          if(memmux) MUXB = 3'b101;
          else MUXB = 3'b011;
        end
else if (EXOP2 == wbwrite2)
        begin
          if(wbmux) MUXB = 3'b110;
          else MUXB = 3'b100;
```

```
        end
else MUXB = 0;



//MUXC
if (EXr15 == memwrite1) MUXC = 3'b001;
else if (EXr15 == wbwrite1) MUXC = 3'b010;
else if (EXr15 == memwrite2)
        begin
          if(memmux) MUXC = 3'b101;
          else MUXC = 3'b011;
        end
else if (EXr15 == wbwrite2)
        begin
          if(wbmux) MUXC = 3'b110;
          else MUXC = 3'b100;
        end
else MUXC = 0;



end
endmodule
```

Sign extend

```
module signext #(parameter SIZE = 8)(input [SIZE-1:0] a,
                                     output reg [15:0] y);

always @*
        begin
                if(a[SIZE-1] == 1'b0)
                y = {{16-SIZE{1'b0}},a};
                else
                y = {{16-SIZE{1'b1}},a};
        end
endmodule
```

2-1 mux

```
module twoonemux #(parameter SIZE = 16) (input [SIZE-1:0] a,b,
                                         input sel,
                                         output reg [SIZE-1:0] y);

always @*
        begin
                case (sel)
                        1'd1 :  y = b;
                        default : y = a;
                endcase
        end

endmodule
```

Zero extend

```
module zeroext #(parameter SIZE = 8)(input [SIZE-1:0] a,
                                     output reg [15:0] y);

always @*
        y = {{16-SIZE{1'b0}},a};
endmodule
```