Adrian Ruiz

CPE 404

HM 4

0. Around 25 hours

Hazard unit Code

```
module hazunit(input[4:0] RsE, RtE, RsD,RtD, WriteRegM, WriteRegW, WriteRegE,
               input RegWriteE, RegWriteM, RegWriteW, MemtoRegE,MemtoRegM,
BranchD, Jump,
               output [1:0] ForwardAE, ForwardBE,
               output StallF, StallD, FlushE, ForwardAD, ForwardBD);

    reg lwstall, branchstall, superstall, jumpstall;
    reg [1:0] ForwardAEReg, ForwardBEReg;
    reg ForwardADReg, ForwardBDReg;

    always @(*) begin  //Forward AE and ForwardBE
        if ((RsE !=0) && (RsE == WriteRegM) && (RegWriteM))
            ForwardAEReg <= 10;
        else if ((RsE !=0) && (RsE == WriteRegW) && (RegWriteW))
            ForwardAEReg <= 01;
        else
            ForwardAEReg <= 00;

        if ((RtE !=0) && (RtE == WriteRegM) && (RegWriteM))
            ForwardBEReg <= 10;
        else if ((RtE !=0) && (RtE == WriteRegW) && (RegWriteW))
            ForwardBEReg <= 01;
        else
            ForwardBEReg <= 00;

        //Forward AD and Forward BD
        ForwardADReg <= (RsD !=0) && (RsD == WriteRegM) && RegWriteM;
        ForwardBDReg <= (RtD !=0) && (RtD == WriteRegM) && RegWriteM;

        //LWstall
        lwstall <= ((RsD == RtE) || (RtD == RtE)) && MemtoRegE;

        //Branch Stall
        branchstall <= (BranchD && RegWriteE && (WriteRegE == RsD ||
WriteRegE == RtD)) ||
                       (BranchD && MemtoRegM && (WriteRegM == RsD ||
WriteRegM == RtD));
        //Jump Stall
        jumpstall <= (Jump && RegWriteE && (WriteRegE == RsD || WriteRegE ==
RtD)) ||
                     (Jump && MemtoRegM && (WriteRegM == RsD || WriteRegM ==
RtD));

        superstall <= lwstall || branchstall || jumpstall;
```

```
    end
    //Assigning Outputs
    assign ForwardAE = ForwardAEReg;
    assign ForwardBE = ForwardBEReg;

    assign ForwardAD = ForwardADReg;
    assign ForwardBD = ForwardBDReg;

    assign StallF = superstall;
    assign StallD = superstall;
    assign FlushE = superstall;

endmodule
```

Hazard Unit Testbench

```
module testbenchhaz();
  reg         clk, reset;
  reg[4:0]    RsE, RtE, RsD, RtD, WriteRegM, WriteRegW, WriteRegE;
  reg         RegWriteE, RegWriteM, RegWriteW, MemtoRegE, MemtoRegM, BranchD;
  wire[1:0]   ForwardAE, ForwardBE;
  wire        StallF, StallD, FlushE, ForwardAD, ForwardBD;

  // instantiate device under test
  hazunit dut(RsE, RtE, RsD, RtD, WriteRegM, WriteRegW, WriteRegE,
                  RegWriteE, RegWriteM, RegWriteW, MemtoRegE, MemtoRegM,
BranchD,
                  ForwardAE, ForwardBE,
                  StallF, StallD, FlushE, ForwardAD, ForwardBD);

  initial begin
    $display("Test all zeros"); #10;
    RsE = 0; RtE = 0; RsD = 0; RtD = 0; WriteRegM = 0; WriteRegW = 0;
    WriteRegE = 0; RegWriteE = 0; MemtoRegE = 0; MemtoRegM = 0; RegWriteM =
0; RegWriteW = 0; BranchD = 0; #10
    $display("ForwardAD = %b, ForwardBD = %b, StallD = %b, StallF = %b,
FlushE = %b, ForwardAE = %b, ForwardBE = %b",
    ForwardAD, ForwardBD, StallD, StallF, FlushE, ForwardAE, ForwardBE); #10;

    $display("Testing RsE Mem Data"); #10;
    RsE = 1; RtE = 2; RsD = 3; RtD = 4; WriteRegM = 1; WriteRegW = 5;
    WriteRegE = 6; RegWriteE = 0; RegWriteM = 1; RegWriteW = 0; BranchD = 0;
#10
    $display("ForwardAD = %b, ForwardBD = %b, StallD = %b, StallF = %b,
FlushE = %b, ForwardAE = %b, ForwardBE = %b Expected ForwardAE = 10",
    ForwardAD, ForwardBD, StallD, StallF, FlushE, ForwardAE, ForwardBE); #10;

    $display("Testing RsE Write Back Data "); #10;
    RsE = 1; RtE = 2; RsD = 3; RtD = 4; WriteRegM = 8; WriteRegW = 1;
    WriteRegE = 6; RegWriteE = 0; RegWriteM = 1; RegWriteW = 1; BranchD = 0;
#10
```

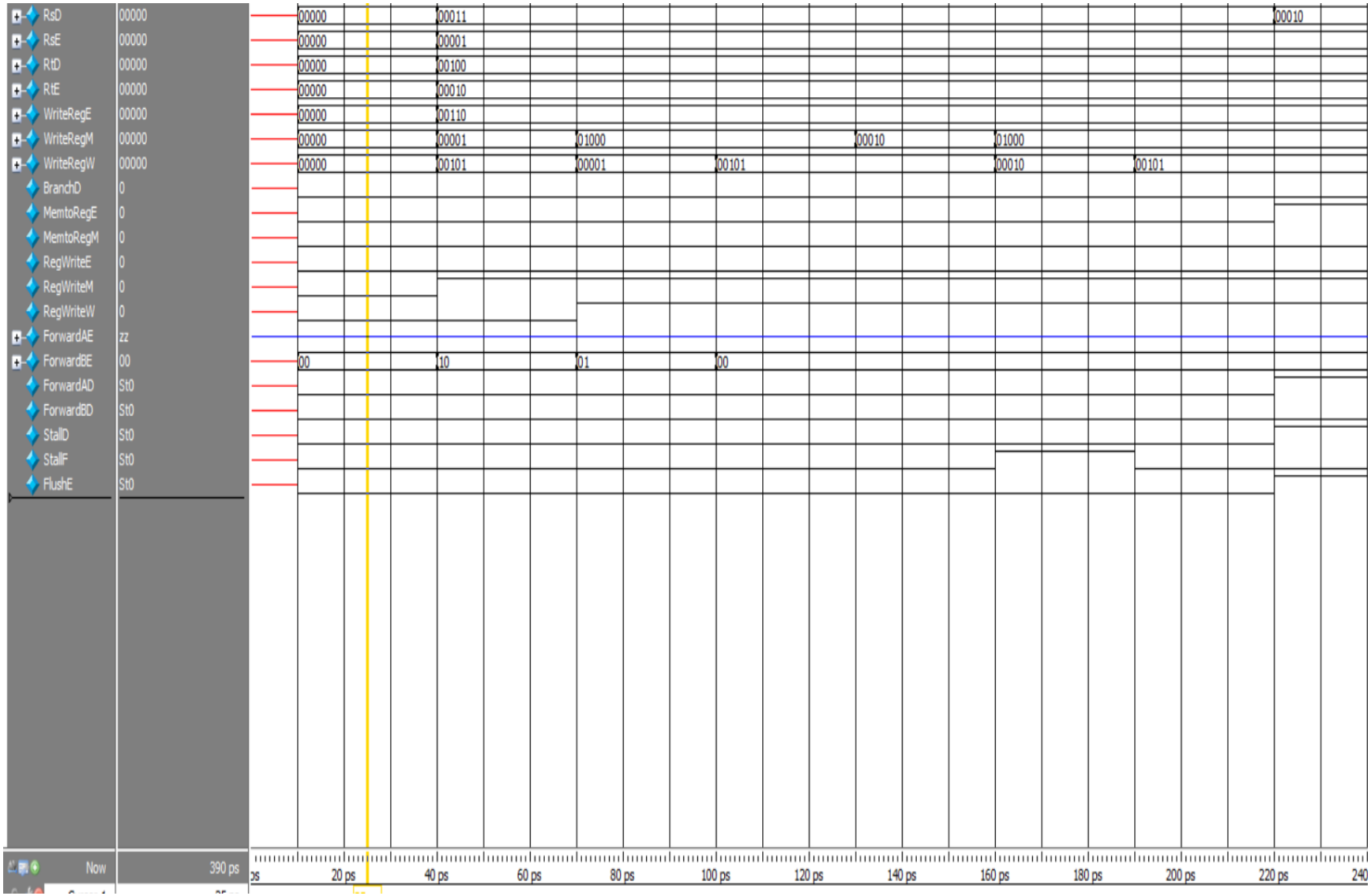
```
    $display("ForwardAD = %b, ForwardBD = %b, StallD = %b, StallF = %b,
FlushE = %b, ForwardAE = %b, ForwardBE = %b Expected ForwardAE = 01",
    ForwardAD, ForwardBD, StallD, StallF, FlushE, ForwardAE, ForwardBE); #10;

    $display("Testing RsE with no Data Hazards "); #10;
    RsE = 1; RtE = 2; RsD = 3; RtD = 4; WriteRegM = 8; WriteRegW = 5;
    WriteRegE = 6; RegWriteE = 0; RegWriteM = 1; RegWriteW = 1; BranchD = 0;
#10
    $display("ForwardAD = %b, ForwardBD = %b, StallD = %b, StallF = %b,
FlushE = %b, ForwardAE = %b, ForwardBE = %b Expected ForwardAE = 00",
    ForwardAD, ForwardBD, StallD, StallF, FlushE, ForwardAE, ForwardBE); #10;

    $display("Testing RtE Mem Back Data Hazard"); #10;
    RsE = 1; RtE = 2; RsD = 3; RtD = 4; WriteRegM = 2; WriteRegW = 5;
    WriteRegE = 6; RegWriteE = 0; RegWriteM = 1; RegWriteW = 1; BranchD = 0;
#10
    $display("ForwardAD = %b, ForwardBD = %b, StallD = %b, StallF = %b,
FlushE = %b, ForwardAE = %b, ForwardBE = %b Expected ForwardBE = 10",
    ForwardAD, ForwardBD, StallD, StallF, FlushE, ForwardAE, ForwardBE); #10;

    $display("Testing RtE Write Back Data Hazard"); #10;
    RsE = 1; RtE = 2; RsD = 3; RtD = 4; WriteRegM = 8; WriteRegW = 2;
    WriteRegE = 6; RegWriteE = 0; RegWriteM = 1; RegWriteW = 1; BranchD = 0;
#10
    $display("ForwardAD = %b, ForwardBD = %b, StallD = %b, StallF = %b,
FlushE = %b, ForwardAE = %b, ForwardBE = %b Expected ForwardBE = 01",
    ForwardAD, ForwardBD, StallD, StallF, FlushE, ForwardAE, ForwardBE); #10;

    $display("Testing RsE No Data Hazard "); #10;
    RsE = 1; RtE = 2; RsD = 3; RtD = 4; WriteRegM = 8; WriteRegW = 5;
    WriteRegE = 6; RegWriteE = 0; RegWriteM = 1; RegWriteW = 1; BranchD = 0;
#10
    $display("ForwardAD = %b, ForwardBD = %b, StallD = %b, StallF = %b,
FlushE = %b, ForwardAE = %b, ForwardBE = %b Expected ForwardBE = 00",
    ForwardAD, ForwardBD, StallD, StallF, FlushE, ForwardAE, ForwardBE); #10;

    $display("Testing LW Stall/ RsD == RtE"); #10;
    RsE = 1; RtE = 2; RsD = 2; RtD = 4; WriteRegM = 8; WriteRegW = 5;
    WriteRegE = 6; RegWriteE = 0; RegWriteM = 1; RegWriteW = 1; MemtoRegE =
1; MemtoRegM = 0; BranchD = 0; #10
    $display("ForwardAD = %b, ForwardBD = %b, StallD = %b, StallF = %b,
FlushE = %b, ForwardAE = %b, ForwardBE = %b Expected Flush  = 1",
    ForwardAD, ForwardBD, StallD, StallF, FlushE, ForwardAE, ForwardBE); #10;

    $display("Testing LW Stall/ RtD == RtE"); #10;
    RsE = 1; RtE = 2; RsD = 3; RtD = 2; WriteRegM = 8; WriteRegW = 5;
    WriteRegE = 6; RegWriteE = 0; RegWriteM = 1; RegWriteW = 1; MemtoRegE =
1; MemtoRegM = 0; BranchD = 0; #10
    $display("ForwardAD = %b, ForwardBD = %b, StallD = %b, StallF = %b,
FlushE = %b, ForwardAE = %b, ForwardBE = %b Expected Flush  = 1",
    ForwardAD, ForwardBD, StallD, StallF, FlushE, ForwardAE, ForwardBE); #10;

    $display("Testing ForwardAD"); #10;
    RsE = 1; RtE = 2; RsD = 2; RtD = 4; WriteRegM = 2; WriteRegW = 5;
    WriteRegE = 6; RegWriteE = 0; RegWriteM = 1; RegWriteW = 1; MemtoRegE =
1; MemtoRegM = 0; BranchD = 0; #10
```

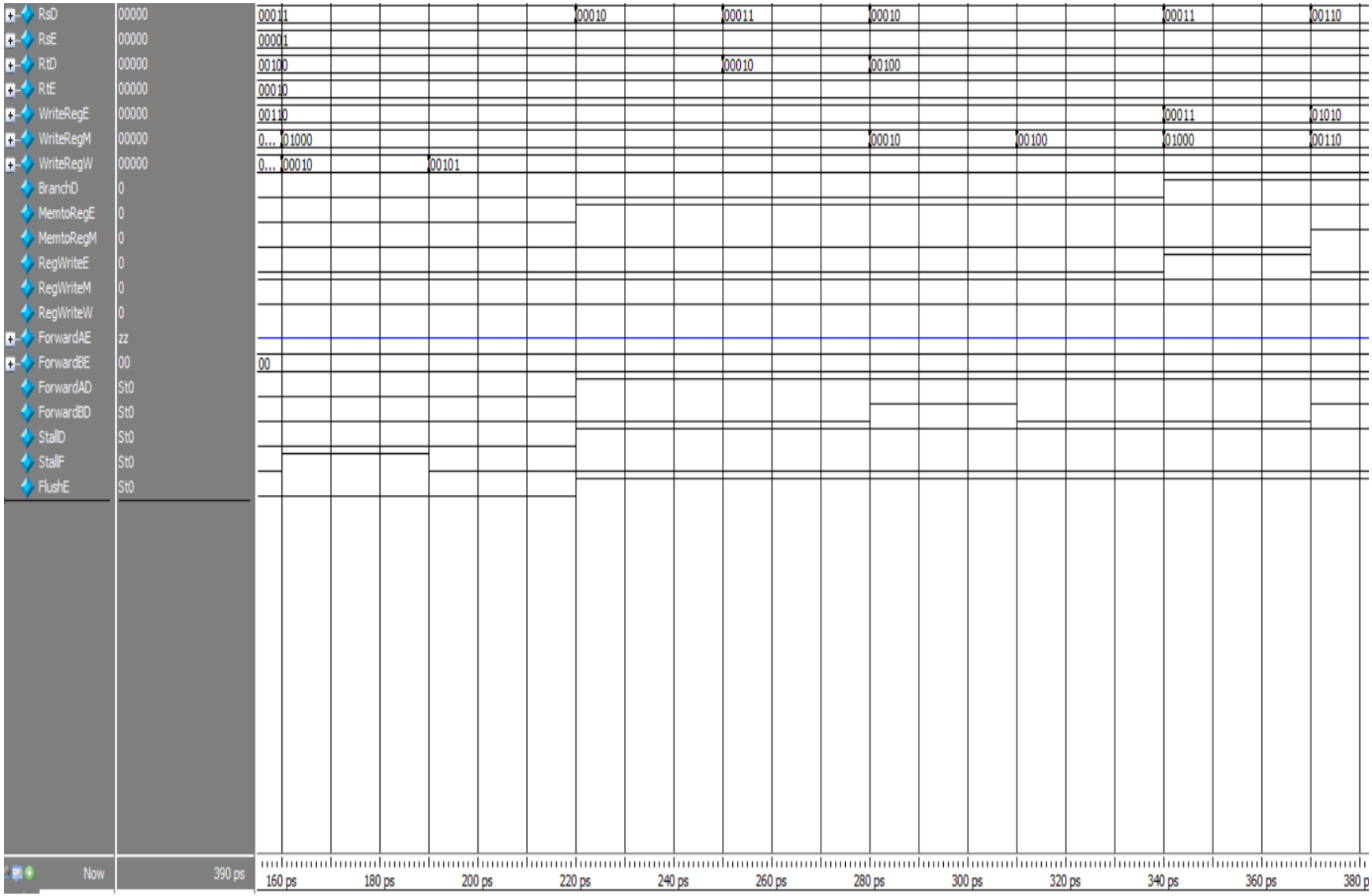
```
    $display("ForwardAD = %b, ForwardBD = %b, StallD = %b, StallF = %b,
FlushE = %b, ForwardAE = %b, ForwardBE = %b Expected ForwardAD  = 1",
    ForwardAD, ForwardBD, StallD, StallF, FlushE, ForwardAE, ForwardBE); #10;

    $display("Testing ForwardBD"); #10;
    RsE = 1; RtE = 2; RsD = 2; RtD = 4; WriteRegM = 4; WriteRegW = 5;
    WriteRegE = 6; RegWriteE = 0; RegWriteM = 1; RegWriteW = 1; MemtoRegE =
1; MemtoRegM = 0; BranchD = 0; #10
    $display("ForwardAD = %b, ForwardBD = %b, StallD = %b, StallF = %b,
FlushE = %b, ForwardAE = %b, ForwardBE = %b Expected ForwardBD  = 1",
    ForwardAD, ForwardBD, StallD, StallF, FlushE, ForwardAE, ForwardBE); #10;

    $display("Testing Branch Stall Execute Stage"); #10;
    RsE = 1; RtE = 2; RsD = 3; RtD = 4; WriteRegM = 8; WriteRegW = 5;
    WriteRegE = 3; RegWriteE = 1; RegWriteM = 1; RegWriteW = 1; MemtoRegE =
1; MemtoRegM = 0; BranchD = 1; #10
    $display("ForwardAD = %b, ForwardBD = %b, StallD = %b, StallF = %b,
FlushE = %b, ForwardAE = %b, ForwardBE = %b Expected Flush  = 1",
    ForwardAD, ForwardBD, StallD, StallF, FlushE, ForwardAE, ForwardBE); #10;


    $display("Testing Branch Memory Stage"); #10;
    RsE = 1; RtE = 2; RsD = 6; RtD = 4; WriteRegM = 6; WriteRegW = 5;
    WriteRegE = 10; RegWriteE = 0; RegWriteM = 1; RegWriteW = 1; MemtoRegE =
1; MemtoRegM = 1; BranchD = 1; #10
    $display("ForwardAD = %b, ForwardBD = %b, StallD = %b, StallF = %b,
FlushE = %b, ForwardAE = %b, ForwardBE = %b Expected Flush  = 1",
    ForwardAD, ForwardBD, StallD, StallF, FlushE, ForwardAE, ForwardBE); #10;

 end
endmodule
```

There is no test vectors because I just tested the cases without a reading in a file.

Hazard Unit Waveform

Below is the waveform generated from the test bench.


RsD 00000
RsE 00000
RtD 00000
RtE 00000
WriteRegE 00000
WriteRegM 00000
WriteRegW 00000
BranchD 0
MemtoRegE 0
MemtoRegM 0
RegWriteE 0
RegWriteM 0
RegWriteW 0
ForwardAE zz
ForwardBE 00
ForwardAD St0
ForwardBD St0
StallD St0
StallF St0
FlushE St0
00000 00011 00010
00000 00001
00000 00100
00000 00010
00000 00110
00000 00001 01000 00010 01000
00000 00101 00001 00101 00010 00101
00 10 01 00
Now 390 ps
20 ps 40 ps 60 ps 80 ps 100 ps 120 ps 140 ps 160 ps 180 ps 200 ps 220 ps 240


RsD
RsE
RtD
RtE
WriteRegE
WriteRegM
WriteRegW
BranchD
MemtoRegE
MemtoRegM
RegWriteE
RegWriteM
RegWriteW
ForwardAE
ForwardBE
ForwardAD
ForwardBD
StallD
StallF
FlushE
00000
00000
00000
00000
00000
00000
00000
0
0
0
0
0
0
zz
00
St0
St0
St0
St0
St0
00011
00001
00100
00010
00110
0... 01000
0... 00010
00101
00010
00011
00010
00011
00110
00010
00100
00011
01010
00010
00100
01000
00110
Now
390 ps
160 ps
180 ps
200 ps
220 ps
240 ps
260 ps
280 ps
300 ps
320 ps
340 ps
360 ps
380 p

Pipelined Processor Code

```
//-------------------------------------------------
// mipssingle.v
// Sarah_Harris@hmc.edu 22 June 2007
// Single-cycle MIPS processor
//-------------------------------------------------

// multi-cycle MIPS processor

module controller(input  [5:0] op, funct,
                  input        zero,
                  output       memtoreg, memwrite,
                  output       alusrc,      // LUI
                  output       regdst,      // JAL
                  output       regwrite,
                  output       Jump,
                  output [3:0] alucontrol,  // 4 bits for SLL
                        input        ltez,        // BLEZ
                        output       jal,         // JAL
                        output       BranchD);         // BranchD

  wire [1:0] aluop;
  wire       branch;
  wire       blez;  // BLEZ and JAL not Used, they wires do connect
                    //any where in the datapath.

  maindec md(op, memtoreg, memwrite, branch,
             alusrc, regdst, regwrite, Jump,
             aluop, blez, jal, BranchD);  // BLEZ, JAL, BranchD
  aludec  ad(funct, aluop, alucontrol);

endmodule

//Maindecoder
//Alu src and regdst have been reduced to one bit
//since JAL and LUI are not used
module maindec(input  [5:0] op,
               output       memtoreg, memwrite,
               output       branch,
               output       alusrc, //
               output       regdst, // JAL
               output       regwrite,
               output       Jump,
               output [1:0] aluop,
                    output       blez,   // BLEZ
                    output       jal,    // JAL
                    output       BranchD);    // BranchD

  reg [11:0] controls;
  assign {regwrite, regdst, alusrc,
          branch, memwrite,
          memtoreg, Jump, aluop,
             blez,   // BLEZ
             jal,    // JAL
             BranchD}     // BranchD
             = controls;
```

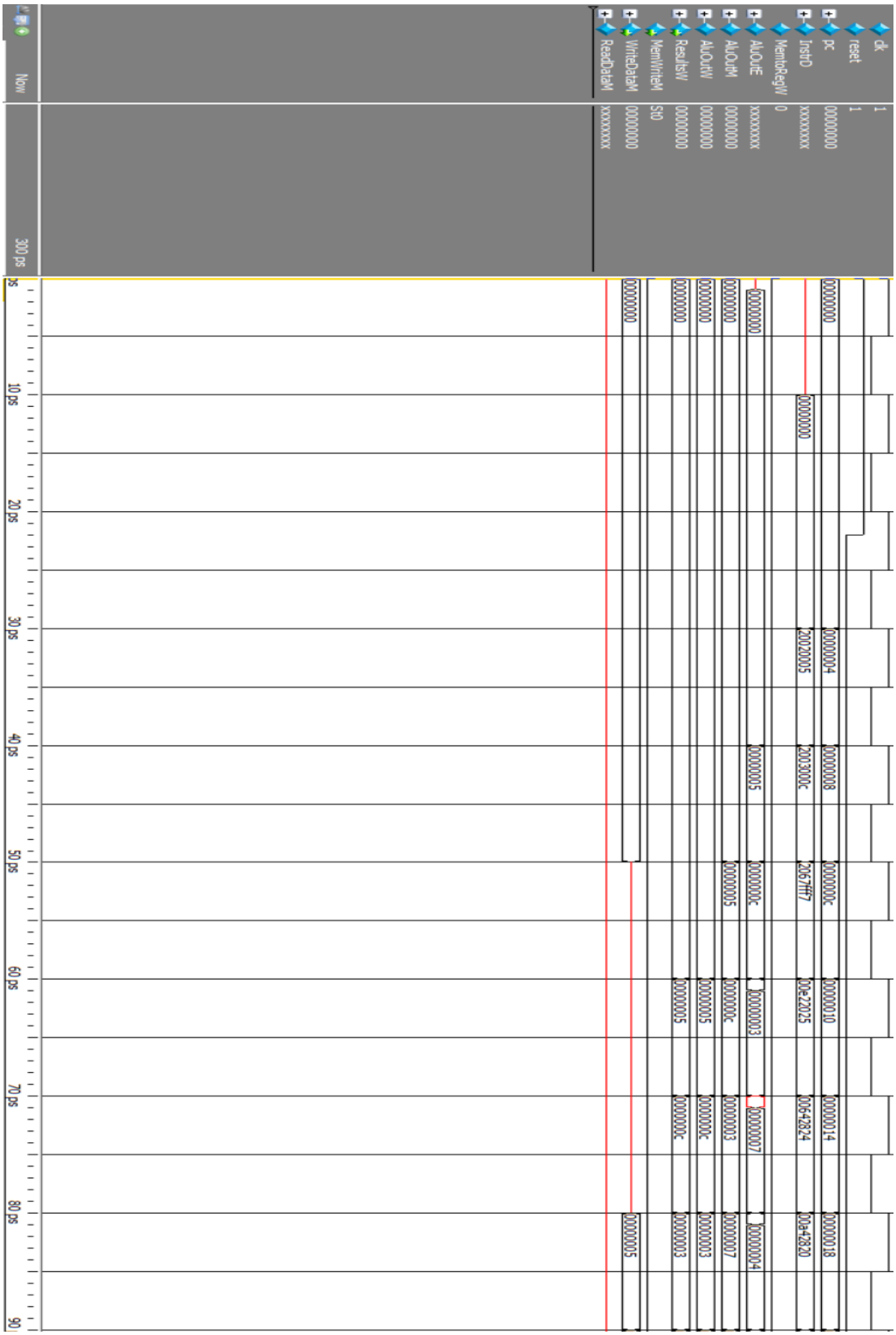
```
  always @(*)
    case(op)
      6'b000000: controls <= 12'b1_1_0_0_0_0_0_10_0_0_0_; //Rtype
      6'b100011: controls <= 12'b1_0_1_0_0_0_0_00_0_0_0_; //LW
      6'b101011: controls <= 12'b0_0_1_0_1_0_0_00_0_0_0_; //SW
      6'b000100: controls <= 12'b0_0_0_0_0_1_0_00_0_1_1_; //BEQ
      6'b001000: controls <= 12'b1_0_1_0_0_0_0_00_0_0_0_; //ADDI
      6'b000010: controls <= 12'b0_0_0_0_0_0_1_00_0_0_0; //J
      //6'b001010: controls <= 12'b1_0_001000011000; //SLTI
      //6'b001111: controls <= 14'b10010000000000; //LUI
      //6'b000110: controls <= 14'b00000000001100; //BLEZ
      //6'b000011: controls <= 14'b11000000100010; //JAL
      //6'b100001: controls <= 14'b10001001000001; // LH
      default:   controls <= 12'bxxxxxxxxxxxx; //???
    endcase
endmodule

module aludec(input      [5:0] funct,
              input      [1:0] aluop,
              output reg [3:0] alucontrol); // 4-bits for SLL

  always @(*)
    case(aluop)
      2'b00: alucontrol <= 4'b0010;  // add
      2'b01: alucontrol <= 4'b1010;  // sub
      2'b11: alucontrol <= 4'b1011;  // slt
      default: case(funct)          // RTYPE
          6'b100000: alucontrol <= 4'b0010; // ADD
          6'b100010: alucontrol <= 4'b1010; // SUB
          6'b100100: alucontrol <= 4'b0000; // AND
          6'b100101: alucontrol <= 4'b0001; // OR
          6'b101010: alucontrol <= 4'b1011; // SLT
          //6'b000000: alucontrol <= 4'b0100; // SLL
          default:   alucontrol <= 4'bxxxx; // ???
        endcase
    endcase
endmodule

module datapath(input         clk, reset,
                output [31:0] pcF,
                output [31:0] ReadDataW, WriteDataM, ResultsW,
                output MemWriteM
                );



    //Wires Next PC Logic
    wire [31:0] pcnext, pcnextbr, pcplus4, pcbranch;
    wire [31:0] InstrF;

    //Not inputted Anywhere Only used for as Place Keeper in ALU
    wire zero;

  //Wire Fetch Register
    wire [31:0] pcplus4F;
```

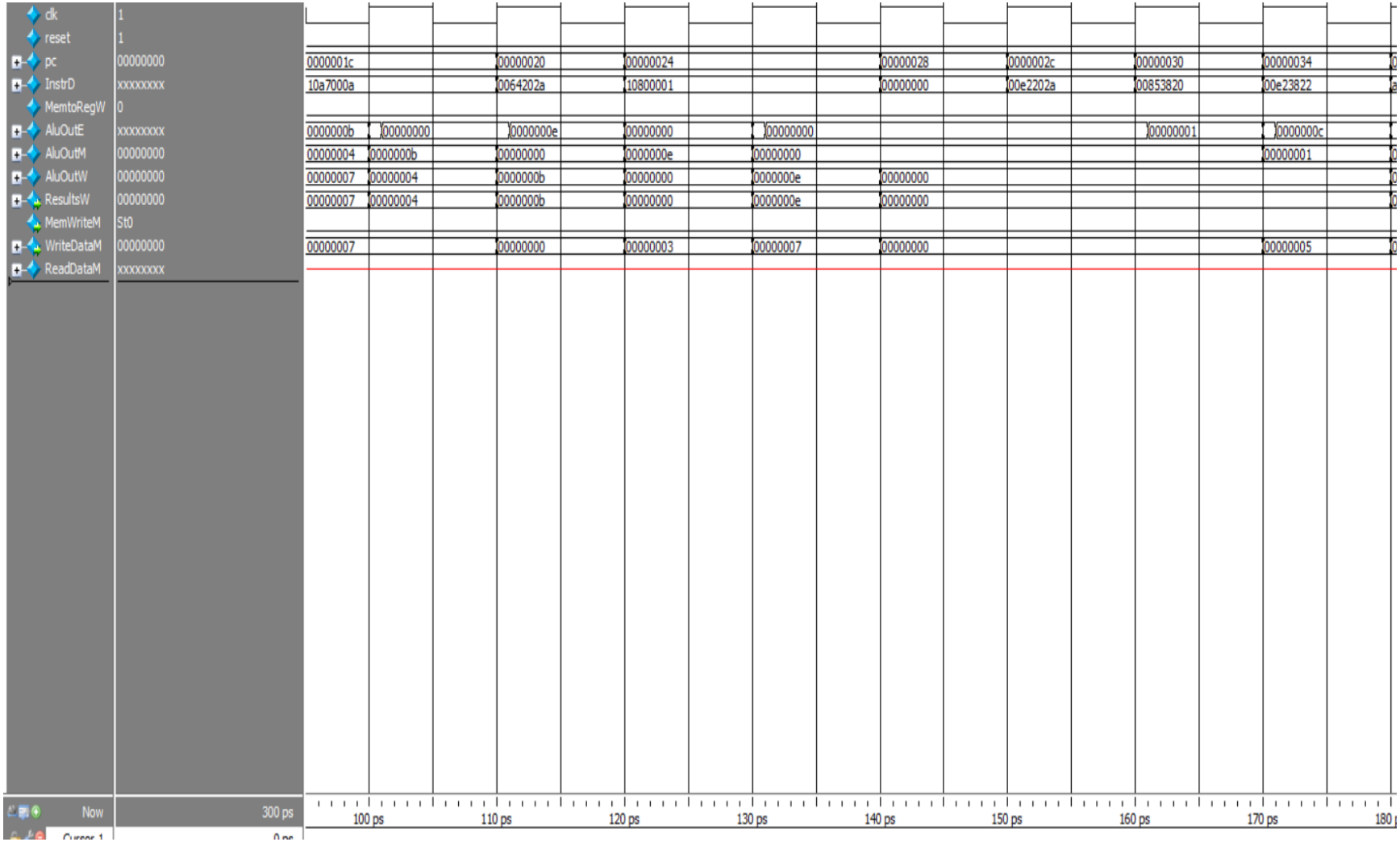
```
  //Wires for Decode Register
    wire [31:0] InstrD, pcplus4D, pcBranchD, signImmD, RD1BMuxD, RD2BMuxD,
RD1AMuxD, RD2AMuxD;
    wire [31:0] signImmshD;
    wire [3:0] AluControlD;
    wire equalD, PCSrcD, RegWriteD, MemtoRegD, MemWriteD, AluSrcD, RegDstD,
BranchD;

    //Wires for Execute
    wire[31:0] RD1AMuxE, RD2AMuxE, AluOutE, SrcAE, SrcBE, WriteDataE,
signImmE;
    wire[4:0]  RsE, RtE, RdE, WriteRegE, ShamtE;
    wire[3:0]  AluControlE;
    wire RegWriteE, MemtoRegE, MemWriteE, AluSrcE, RegDstE;

    //Wire for Memory
    wire[31:0] AluOutM, ReadDataM;
    wire[4:0]  WriteRegM;
    wire RegWriteM, MemtoRegM;

    //Wire For Write Back
    wire[31:0] AluOutW;
    wire[4:0] WriteRegW;
    wire RegWriteW, MemtoRegW;

    //Haz unit wires
    wire[1:0] ForwardAE, ForwardBE;
    wire StallD, StallF, FlushE, ForwardAD, ForwardBD;


    //Hazard Unit
    hazunit hazunit(RsE, RtE, InstrD[25:21],InstrD[20:16], WriteRegM,
WriteRegW, WriteRegE,
                    RegWriteE, RegWriteM, RegWriteW, MemtoRegE, MemtoRegM,
BranchD, Jump,
                    ForwardAE, ForwardBE,
                    StallF, StallD, FlushE, ForwardAD, ForwardBD);


    //Controller
    controller c(InstrD[31:26], InstrD[5:0], zero,
               MemtoRegD, MemWriteD,
               AluSrcD, RegDstD, RegWriteD, Jump,
               AluControlD,
                    ltez,  // BLEZ
                    jal,   // JAL
                    BranchD);   // BranchD


    //Instruction and Data Memory
    imem imem(pcF[7:2], InstrF);
    dmem dmem(clk, MemWriteM, AluOutM, WriteDataM,
            ReadDataM);

    //next PC Logic
    flopenr #(32)   pcreg(clk, reset, StallF, pcnext, pcF);
    adder           pcadd1(pcF, 32'b100, pcplus4F);
```

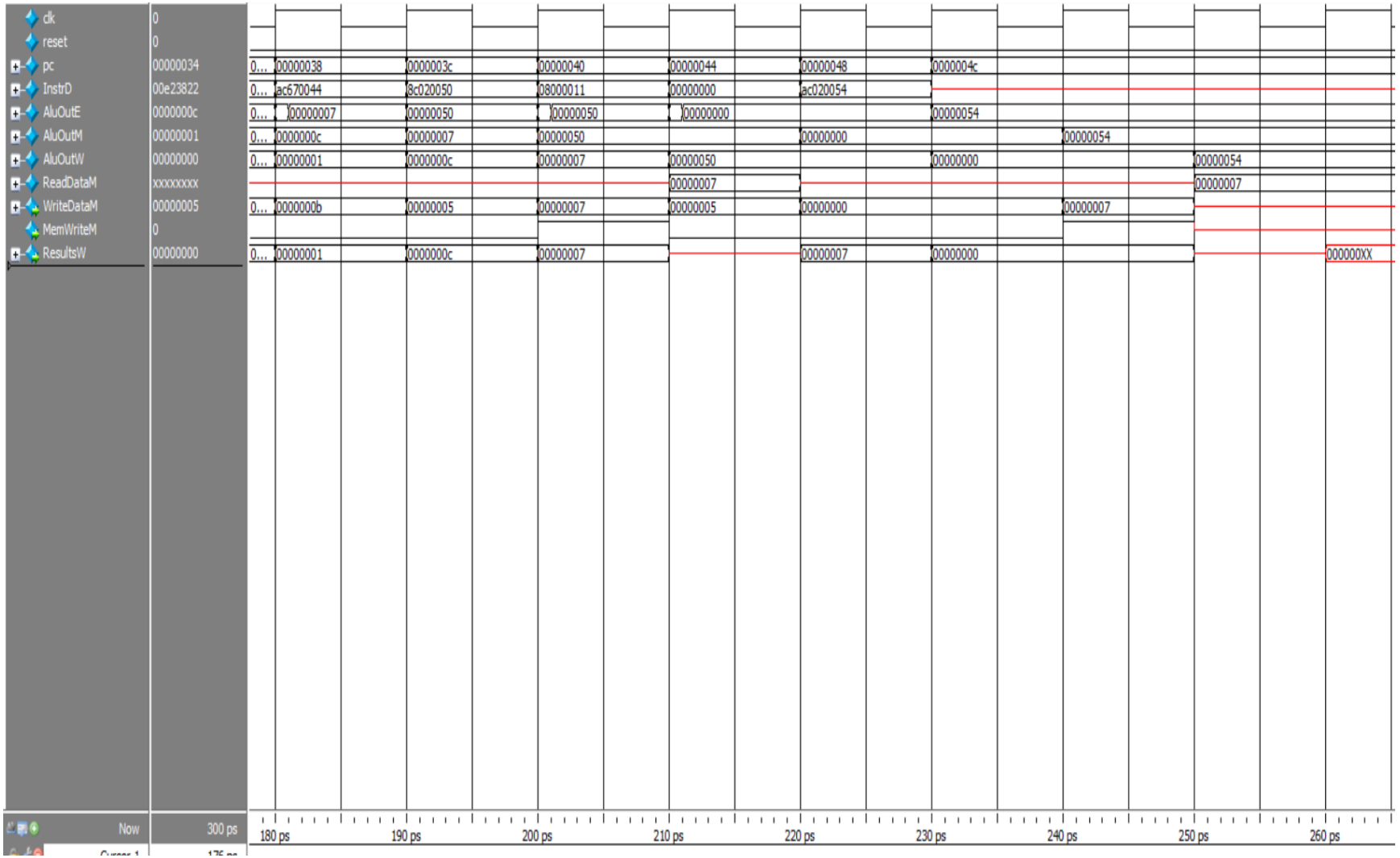
```
    //Branch
    sl2             immsh(signImmD, signImmshD);
    adder           pcadd2(pcplus4D, signImmshD, pcBranchD);
    mux2 #(32)      pcbrmux(pcplus4F, pcBranchD, PCSrcD, pcnextbr);
    //Jump
    mux2 #(32)      pcmux(pcnextbr, {pcplus4D[31:28], InstrD[25:0], 2'b00},
Jump, pcnext);


    //RegisterFile
    regfile         rf(clk, RegWriteW, InstrD[25:21], InstrD[20:16],
WriteRegW, ResultsW,
                       RD1BMuxD, RD2BMuxD);

    //Branch and Jump Data Hazard Mux
    mux2 #(32)      br1mux(RD1BMuxD, AluOutM, ForwardAD, RD1AMuxD);
    mux2 #(32)      br2mux(RD2BMuxD, AluOutM, ForwardBD, RD2AMuxD);
    equalReg        eq(RD1AMuxD, RD2AMuxD, equalD);
    and2            BAnd(BranchD, equalD, PCSrcD);

    //WriteRegE Mux
    mux2 #(5)       wrmux(RtE, RdE, RegDstE, WriteRegE);

    //ForwardAE Mux
    mux3 #(32)      faemux(RD1AMuxE, ResultsW, AluOutM, ForwardAE, SrcAE);

    //ForwardBE Mux
    mux3 #(32)      fbemux(RD2AMuxE, ResultsW, AluOutM, ForwardBE,
WriteDataE);

    //Immediate or Register Mux
    mux2 #(32)      immux(WriteDataE, signImmE, AluSrcE, SrcBE);

    //ALU
    alu             alu(SrcAE, SrcBE, AluControlE, ShamtE,
                        AluOutE, zero, ltez); //last two bits not used

    //Data Memory output Mux
    mux2 #(32)      memmux(AluOutW, ReadDataW, MemtoRegW, ResultsW);

    //signExt
    signext         se(InstrD[15:0], signImmD);

    //Fetch Register
    floprFet        floprFet(clk, reset | PCSrcD | Jump, StallD, PCSrcD,
pcplus4F, InstrF,
                             pcplus4D, InstrD);

    //Decode Register
    floprDec        floprDec(clk, reset | FlushE, FlushE, RD1AMuxD, RD2AMuxD,
signImmD,
                             InstrD[25:21], InstrD[20:16], InstrD[15:11],
InstrD[10:6],
```

```
                            RegWriteD, MemtoRegD, MemWriteD, AluSrcD,
RegDstD,
                            BranchD, AluControlD, RD1AMuxE, RD2AMuxE,
signImmE,
                            RsE, RtE, RdE, ShamtE, RegWriteE, MemtoRegE,
MemWriteE,
                            AluSrcE, RegDstE, AluControlE);
    //Execute Register
    floprExe        floprExe(clk, reset, AluOutE, WriteDataE, WriteRegE,
RegWriteE,
                            MemtoRegE, MemWriteE, AluOutM, WriteDataM,
WriteRegM,
                            RegWriteM, MemtoRegM, MemWriteM);

    //Memory Register
    floprMem        floprMem(clk, reset, AluOutM, ReadDataM, WriteRegM,
RegWriteM,
                            MemtoRegM, AluOutW, ReadDataW, WriteRegW,
RegWriteW,
                            MemtoRegW);

endmodule
```

Pipelined Processor Waveform




clk 1
reset 1
pc 00000000 0000001c 00000020 00000024 00000028 0000002c 00000030 00000034
InstrD xxxxxxxx 10a7000a 0064202a 10800001 00000000 00e2202a 00853820 00e23822
MemtoRegW 0
AluOutE xxxxxxxx 0000000b 00000000 0000000e 00000000 00000000 00000001 0000000c
AluOutM 00000000 00000004 0000000b 00000000 0000000e 00000000 00000001
AluOutW 00000000 00000007 00000004 0000000b 00000000 0000000e 00000000
ResultsW 00000000 00000007 00000004 0000000b 00000000 0000000e 00000000
MemWriteM St0
WriteDataM 00000000 00000007 00000000 00000003 00000007 00000000 00000005
ReadDataM xxxxxxxx
Now 300 ps
100 ps
110 ps
120 ps
130 ps
140 ps
150 ps
160 ps
170 ps


clk
reset
pc
InstrD
AluOutE
AluOutM
AluOutW
ReadDataM
WriteDataM
MemWriteM
ResultsW
00000034
00e23822
0000000c
00000001
00000000
xxxxxxxx
00000005
0
00000000
00000038
0000003c
00000040
00000044
00000048
0000004c
ac670044
8c020050
08000011
00000000
ac020054
00000007
00000050
00000050
00000000
00000054
0000000c
00000007
00000050
00000000
00000054
00000001
0000000c
00000007
00000050
00000000
00000054
00000007
0000000b
00000005
00000007
00000005
00000000
00000007
00000001
0000000c
00000007
00000007
00000000
000000XX
Now
300 ps
180 ps
190 ps
200 ps
210 ps
220 ps
230 ps
240 ps
250 ps
260 ps

In the first waveform picture, after the reset, and on PC 4, the register file gets instruction 2002005. It is addi $2, $0, 5. The results will be written after 4 cycles and is seen in the ResultsW signal. The next instruction is 2003000c, addi $3, $0, 12. It won't be written into the Register File until after 4 cycles. The next instruction 2067fff7, addi $7, $3, -9. This is an instance of having forward the ALU result from the Memory stage to the Execute stage because of $3 hasn't been written. It does get the right value because of 4 cycles, 3 is written into the Register File. The next instruction is 00e22025, or $4, $7, $2. It is another case of forwarding from the Memory to the Execute stage with $7 being the hazard. The correct is written as shown in the second waveform picture. The next two instructions are ANDs with the second instruction being forwarded data from the Memory stage because $5.

At the start of the second waveform picture, the first beq instruction is decoded. Due to the Data hazard of $5 being written in yet, the instruction is stalled to pass it in. That is why this instruction takes two clock cycles as shown in the waveform. It doesn't take the branch as it should. It falls through because the next instruction is 0064202a, slt $4, $3, $4. The next instruction is the next beq instruction. Once again, it the instruction is stalled for $4 from the previous cycle to be calculated. This time it should take the branch. At this point it, stalls and allows the instruction to pass through. Instruction 00e2202a, slt $4, $7, $2. The next instruction, 0085320, goes through no problem.

At the start of the third waveform picture, the next instruction 00e23822, sub $7, $7, $2, does have a data hazard. $7 must be forwarded from memory. The value in $7 can be seen in ReadDataM. The next instruction ac670044, sw $7, 68($3), needs get the value of $7 forwarded from memory. The next instruction is 8c020050, lw $2, 80($0). No stall is need for this instruction. The next instruction is 08000011, j end. It is taken immediately, and the next instruction is cleared. After one cycle, the next instruction, ac020054, is passed in. ac020054, sw $2, 84($0), saves the right value in memory.