ECSE 4770 – Computer Hardware Design

Lab 9 Report – RISC-V Pipelined Processor

Paul Nieves (RIN: 61999083)

Professor Liu Liu, Garrett Gagnon

December 10 2024

Table of Contents

Survey..3
Modifications for LUI..3
Amended Table 1 and Table 2 for changes made to support lui....................................3
Modified test program and testbench for lui...4
Simulation waveforms...5
Demo of passing testbench to TAs during lab sections...6
RTL Viewer schematics...6
Hierarchical SystemVerilog..7

Survey

I spent about 30 Hrs on this lab

Modifications for LUI

The lui function will require the introduction of a new controller output signal, also named lui. This signal will pass through the datapath, modifying the behavior of both the extend module and the ALU module. In the ALU, lui will activate the control logic for U-type instructions, ensuring that the ALUControl is set to produce the correct output. In the extend module, lui will configure ImmExtD to use the first 5 bits of the input data, appending three zeroes at the end to generate the desired output. This processed signal will then propagate to SrcB and subsequently into the ALU, where the ALU will set the result to the value of SrcB, effectively implementing the lui function.

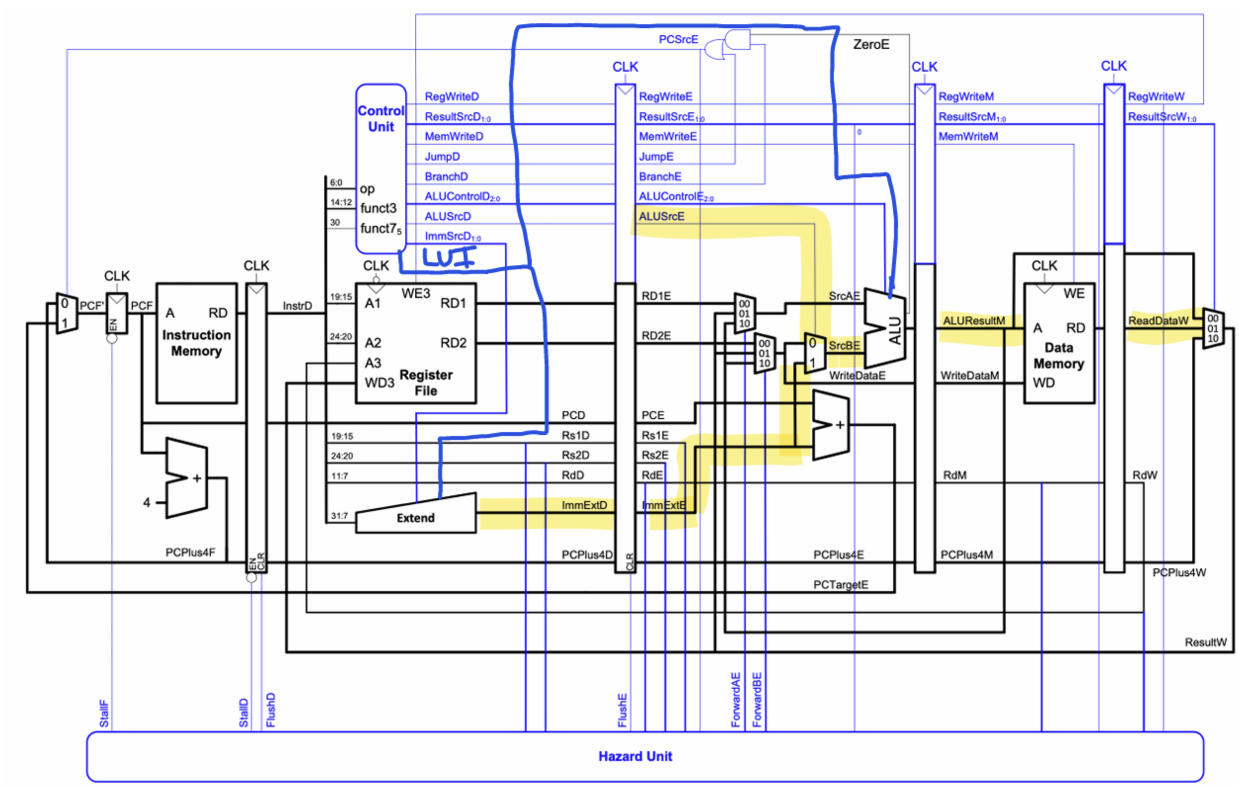

Figure 1: Modified Pipelined Processor with a Hazard Unit

Amended Table 1 and Table 2 for changes made to support lui

Instruction	Opcode	RegWrite	ImmSrc	ALUSrcA	ALUSrcB	MemWrite	ResultSrc	Branch	ALUOp	Jump	LUI
lw	0000011	1	00	0	1	0	01	0	00	0	0
sw	0100011	0	01	0	1	1	xx	0	00	0	0
R-type	0110011	1	xx	0	0	0	00	0	10	0	0
Beq	1100011	0	10	0	0	0	xx	1	01	0	0
I-type ALU	0010011	1	00	0	1	0	00	0	10	0	0
Jal	1101111	1	11	x	x	0	10	0	xx	1	0

LUI (U-type)	0110111	1	00	0	1	0	00	0	10	0	1

Table 1. Main Decoder Truth Table

ALUOp[1:0]	funct3[2:0]	{op5, funct7[5]}	ALUControl[2:0]	Operation
00	x	x	000	Add
01	x	x	001	Subtract
10	000	00, 01, 10	000	Add
10	000	11	001	Subtract
10	010	x	101	SLT
10	110	x	011	OR
10	111	x	010	AND
10	001	x	110	LUI

Table 2. ALU Decoder Truth Table

Modified test program and testbench for lui

```
#       RISC-V Assembly         Description                 Address   Machine Code
main:   addi x2, x0, 5          # x2 = 5                    0         00500113
        addi x3, x0, 12         # x3 = 12                   4         00C00193
        addi x7, x3, -9         # x7 = (12 - 9) = 3         8         FF718393
        or   x4, x7, x2         # x4 = (3 OR 5) = 7         C         0023E233
        and  x5, x3, x4         # x5 = (12 AND 7) = 4       10        0041F2B3
        add  x5, x5, x4         # x5 = (4 + 7) = 11         14        004282B3
        beq  x5, x7, end        # shouldn't be taken        18        02728863
        slt  x4, x3, x4         # x4 = (12 < 7) = 0         1C        0041A233
        beq  x4, x0, around     # should be taken           20        00020463
        addi x5, x0, 0          # shouldn't happen          24        00000293
around: slt  x4, x7, x2         # x4 = (3 < 5)  = 1         28        0023A233
        add  x7, x4, x5         # x7 = (1 + 11) = 12        2C        005203B3
        sub  x7, x7, x2         # x7 = (12 - 5) = 7         30        402383B3
        sw   x7, 84(x3)         # [96] = 7                  34        0471AA23
        lw   x2, 96(x0)         # x2 = [96] = 7             38        06002103
        add  x9, x2, x5         # x9 = (7 + 11) = 18        3C        005104B3
        jal  x3, end            # jump to end, x3 = 0x44    40        008001EF
        addi x2, x0, 1          # shouldn't happen          44        00100113
end:    add  x2, x2, x9         # x2 = (7 + 18)  = 25       48        00910133
        lui  x2, 0x11111        # x2 = 0x11111000           4C        11111137
        sw   x2, 0x20(x3)       # mem[100] = 25             50        0221A023
done:   beq  x2, x2, done       # infinite loop             54        00210063
```

Figure 2: Modified Testvector

```
#       RISC-V Assembly         Description                 Address   Machine Code
main:   addi x2, x0, 5          # x2 = 5                    0         00500113
        addi x3, x0, 12         # x3 = 12                   4         00C00193
        addi x7, x3, -9         # x7 = (12 - 9) = 3         8         FF718393
        or   x4, x7, x2         # x4 = (3 OR 5) = 7         C         0023E233
```

```
         and  x5, x3, x4         # x5 = (12 AND 7) = 4     10         0041F2B3
         add  x5, x5, x4         # x5 = (4 + 7) = 11       14         004282B3
         beq  x5, x7, end        # shouldn't be taken      18         02728863
         slt  x4, x3, x4         # x4 = (12 < 7) = 0       1C         0041A233
         beq  x4, x0, around     # should be taken         20         00020463
         addi x5, x0, 0          # shouldn't happen        24         00000293
around:  slt  x4, x7, x2         # x4 = (3 < 5)  = 1       28         0023A233
         add  x7, x4, x5         # x7 = (1 + 11) = 12      2C         005203B3
         sub  x7, x7, x2         # x7 = (12 - 5) = 7       30         402383B3
         sw   x7, 84(x3)         # [96] = 7                34         0471AA23
         lw   x2, 96(x0)         # x2 = [96] = 7           38         06002103
         add  x9, x2, x5         # x9 = (7 + 11) = 18      3C         005104B3
         jal  x3, end            # jump to end, x3 = 0x44  40         008001EF
         addi x2, x0, 1          # shouldn't happen        44         00100113
end:     add  x2, x2, x9         # x2 = (7 + 18)  = 25     48         00910133
         sw   x2, 0x20(x3)       # mem[100] = 25           4C         0221A023
done:    beq  x2, x2, done       # infinite loop           50         00210063
```

Figure 3: Unmodified Testvector

Simulation waveforms

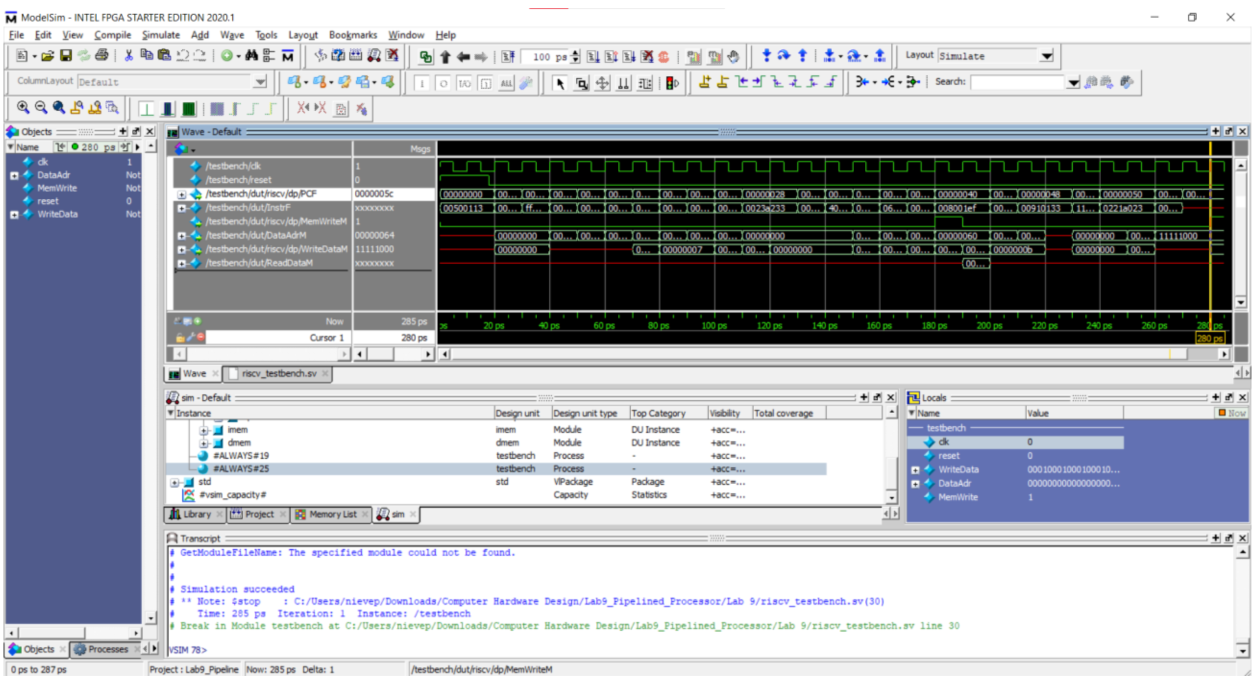

Figure 3: Simulation with LUI Test Vector Modification

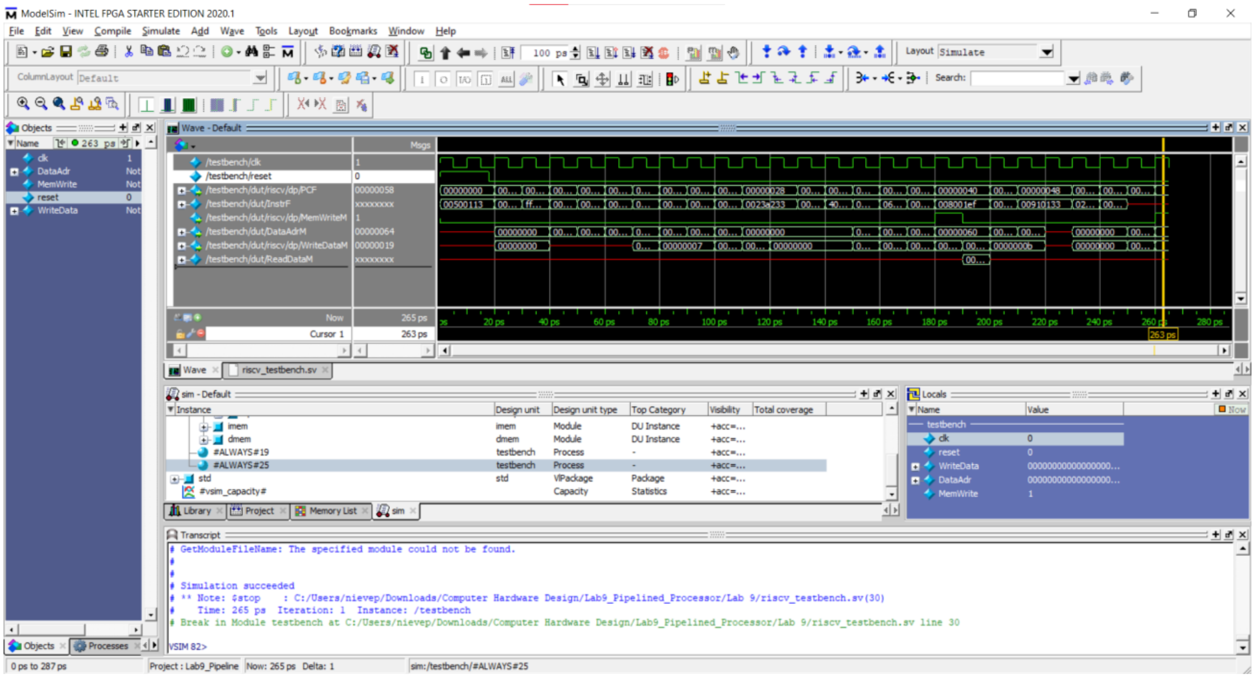

Figure 4: Simulation without LUI Test Vector Modification

Demo of passing testbench to TAs during lab sections.

The testbenches were demoed to the TA in CII 6118 on 12/10/2024

RTL Viewer schematics

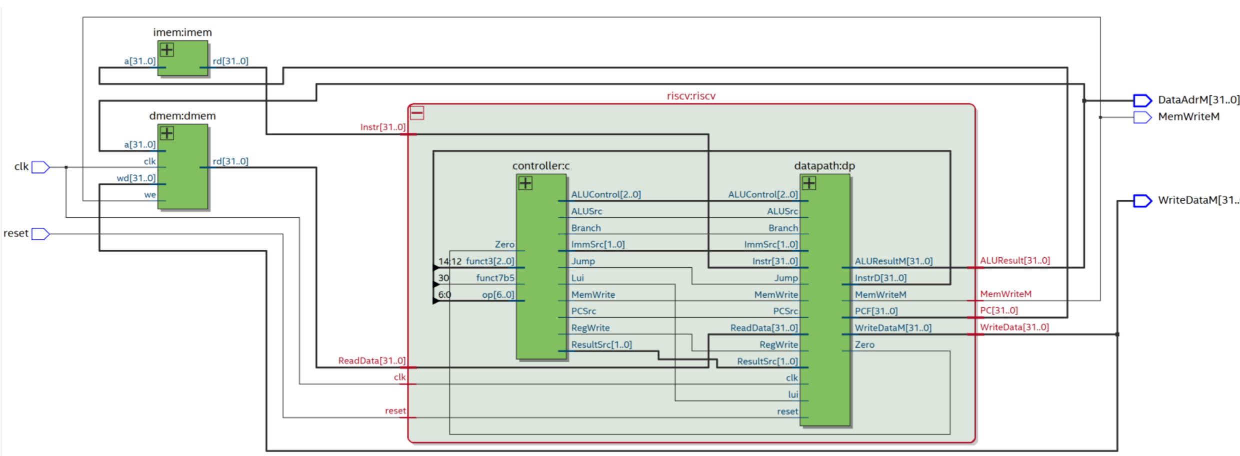

Figure 5: RTL View Datapath and Controller Overview

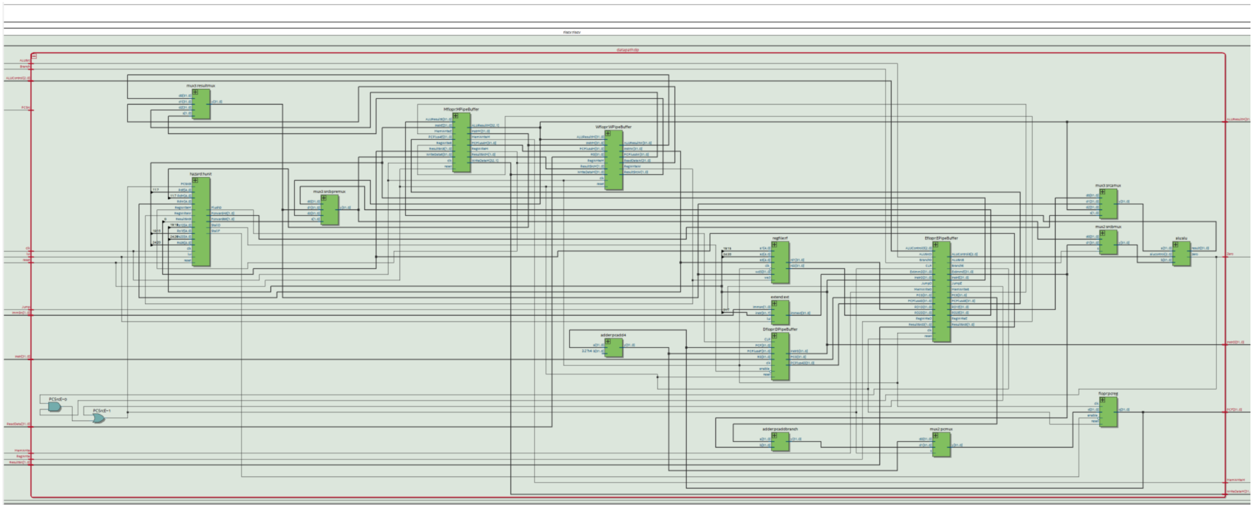

Figure 6: RTL View - Datapath Overview

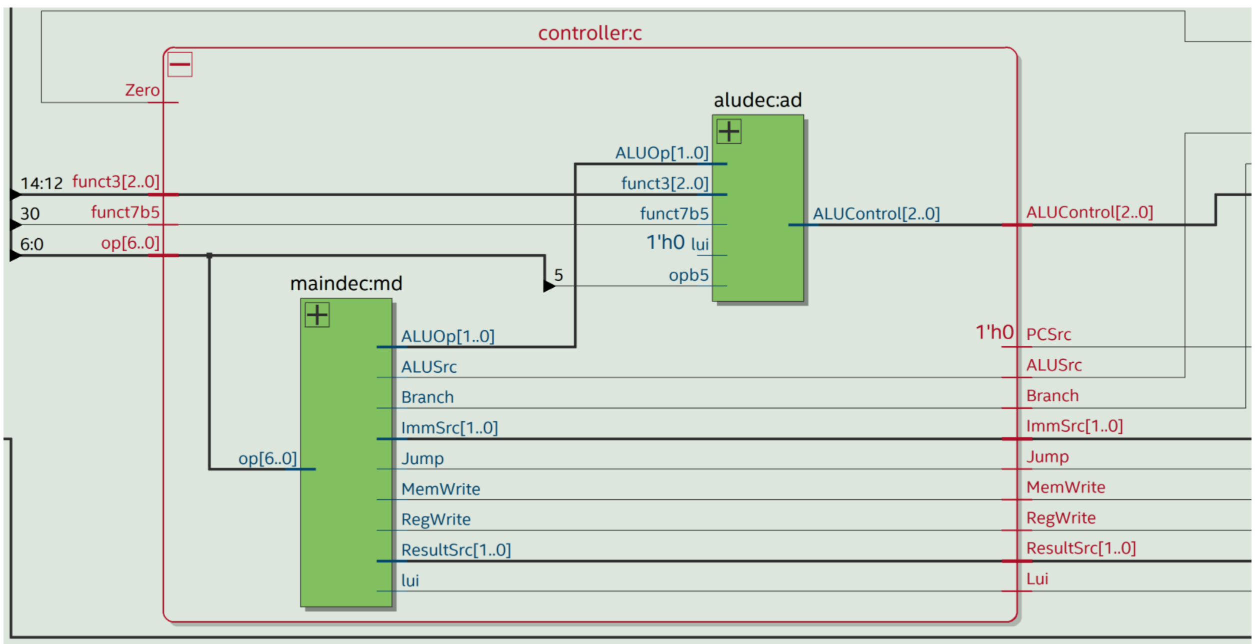

Figure 7: RTL View – Controller Overview

Hierarchical SystemVerilog

```
module testbench();

  logic        clk;
  logic        reset;

  logic [31:0] WriteData, DataAdr;
  logic        MemWrite;

  // instantiate device to be tested
```

```
  top dut(clk, reset, WriteData, DataAdr, MemWrite);

  // initialize test
  initial
    begin
      reset <= 1; # 18; reset <= 0;
    end

  // generate clock to sequence tests
  always
    begin
      clk <= 1; # 5; clk <= 0; # 5;
    end

  // check results
  always @(negedge clk)
    begin
      if(MemWrite) begin
        if(DataAdr === 100 & WriteData === 286330880) begin //25 normal case,
286330880 lui case
          $display("Simulation succeeded");
          $stop;
        end else if (DataAdr !== 96) begin
          $display("Simulation failed");
          $stop;
        end
      end
    end
endmodule
```

Figure 8: risc_v_testbench.sv

```
module top(input  logic        clk, reset,
           output logic [31:0] WriteDataM, DataAdrM,
           output logic        MemWriteM);

  logic [31:0] PCF, InstrF, ReadDataM;

  // instantiate processor and memories
  riscv riscv(clk, reset, PCF, InstrF, MemWriteM, DataAdrM,
                        WriteDataM, ReadDataM);
  imem imem(PCF, InstrF);
  dmem dmem(clk, MemWriteM, DataAdrM, WriteDataM, ReadDataM);
endmodule

module imem(input  logic [31:0] a,
```

```
           output logic [31:0] rd);

  logic [31:0] RAM[63:0];

  initial
      $readmemh("riscvtest.txt",RAM);

  assign rd = RAM[a[31:2]]; // word aligned
endmodule

module dmem(input  logic clk, we,
            input  logic [31:0] a, wd,
            output logic [31:0] rd);

  logic [31:0] RAM[63:0];
  assign rd = RAM[a[31:2]]; // word aligned

  always_ff @(posedge clk)
    if (we) RAM[a[31:2]] <= wd;
endmodule
```

Figure 9: top.sv

```
module riscv(input  logic        clk, reset,
                   output logic [31:0] PC,
                   input  logic [31:0] Instr,
                   output logic        MemWriteM,
                   output logic [31:0] ALUResult, WriteData,
                   input  logic [31:0] ReadData);

  logic       ALUSrc, RegWrite, Jump, Zero, Branch, lui;
  logic [1:0] ResultSrc, ImmSrc;
  logic [2:0] ALUControl;
  logic [31:0] InstrD;

  controller c(InstrD[6:0], InstrD[14:12], InstrD[30], Zero,
               ResultSrc, MemWrite, PCSrc,
               ALUSrc, RegWrite, Jump,
               ImmSrc, Branch, lui, ALUControl);
  datapath dp(clk, reset, ResultSrc, PCSrc,
              ALUSrc, RegWrite, Jump, Branch,
              ImmSrc, ALUControl,
              Zero, PC, Instr, InstrD,
              ALUResult, WriteData, ReadData, MemWrite, lui, MemWriteM);
endmodule
```

Figure 10: riscv.sv

```
module datapath(input  logic        clk, reset,
                input  logic [1:0]  ResultSrc,
                input  logic        PCSrc, ALUSrc,
                input  logic        RegWrite, Jump, Branch,
                input  logic [1:0]  ImmSrc,
                input  logic [2:0]  ALUControl,
                output logic        Zero,
                output logic [31:0] PCF,
                input  logic [31:0] Instr,
                output logic [31:0] InstrD,
                output logic [31:0] ALUResultM, WriteDataM,
                input  logic [31:0] ReadData,
                input  logic        MemWrite, lui,
                output logic        MemWriteM);

  logic [31:0] PCTargetE, PCNext;
  logic [31:0] PCD, PCE;
  logic [31:0] PCPlus4F, PCPlus4D, PCPlus4E, PCPlus4M, PCPlus4W;
  logic [31:0] ImmExtD, ImmExtE;
  logic [31:0] SrcA, SrcB;
  logic [31:0] RD1E, RD2E;
  logic [31:0] RD1D, RD2D;
  logic [31:0] ALUResultE, ALUResultW, ResultW;
  logic [31:0] InstrE, InstrM, InstrW;
  logic [31:0] WriteDataE, ReadDataW;
  logic        RegWriteE, RegWriteM, RegWriteW;
  logic [1:0]  ResultSrcE, ResultSrcM, ResultSrcW;
  logic        MemWriteE;
  logic        JumpE;
  logic        BranchE;
  logic [2:0]  ALUControlE;
  logic        ALUSrcE;
  logic        StallD, FlushD, StallF, FlushE, FakeFlushE;
  logic        PCSrcE;
  logic [1:0]  ForwardAE, ForwardBE;

  assign PCSrcE = ((BranchE & Zero) || JumpE);
  assign FlushE = PCSrcE;
  hazard hunit(clk, reset, InstrD[19:15], InstrE[19:15], InstrD[24:20], 
InstrE[24:20],
               InstrE[11:7], InstrM[11:7], InstrW[11:7], PCSrcE, RegWriteM, 
RegWriteW, ResultSrcE[0], lui,
               StallF, StallD, FlushD, FakeFlushE, ForwardAE, ForwardBE);
  // next PC logic
  //Fetch Stage
```

```
  flopr #(32)  pcreg(clk, reset, !StallF, PCNext, PCF);
  mux2 #(32)   pcmux(PCPlus4F, PCTargetE, PCSrcE, PCNext);

  //Decode Stage
  Dflopr #(32) DPipeBuffer(clk, reset, !StallD, FlushD, Instr, PCF, PCPlus4F,
PCD, PCPlus4D, InstrD);

  adder       pcadd4(PCF, 32'd4, PCPlus4F);



  // register file logic
  regfile     rf(!clk, RegWriteW, InstrD[19:15], InstrD[24:20],
                 InstrW[11:7], ResultW, RD1D, RD2D);
  extend      ext(InstrD[31:7], ImmSrc, lui, ImmExtD);

  //Execute Stage
  Eflopr #(32) EPipeBuffer(clk, reset, FlushE, ResultSrc, MemWrite, Branch,
ALUSrc, RegWrite, Jump, ALUControl, RD1D, RD2D, PCD, InstrD, ImmExtD, PCPlus4D,
                          RD1E, RD2E, PCE, InstrE, ImmExtE, PCPlus4E, ResultSrcE,
ALUControlE, RegWriteE, MemWriteE, JumpE, BranchE, ALUSrcE);
  // ALU logic
  mux3 #(32)  srcbpremux(RD2E, ResultW, ALUResultM, ForwardBE, WriteDataE);
  mux3 #(32)  srcamux(RD1E, ResultW, ALUResultM, ForwardAE, SrcA);
  mux2 #(32)  srcbmux(WriteDataE, ImmExtE, ALUSrcE, SrcB);

  adder       pcaddbranch(PCE, ImmExtE, PCTargetE);
  alu         alu(SrcA, SrcB, ALUControlE, ALUResultE, Zero);

  //Memory Stage
  Mflopr #(32) MPipeBuffer(clk, reset, ResultSrcE, MemWriteE, RegWriteE,
ALUResultE, WriteDataE, InstrE, PCPlus4E,
                          InstrM, PCPlus4M, ALUResultM, WriteDataM, ResultSrcM,
RegWriteM, MemWriteM);

  //Writeback Stage
  Wflopr #(32) WPipeBuffer(clk, reset, ResultSrcM, RegWriteM, ALUResultM,
ReadData, WriteDataM, InstrM, PCPlus4M,
                          ALUResultW, ReadDataW, InstrW, PCPlus4W, ResultSrcW,
RegWriteW);


  mux3 #(32)  resultmux(ALUResultW, ReadDataW, PCPlus4W, ResultSrcW, ResultW);
```

```
endmodule

module regfile(input  logic        clk,
               input  logic        we3,
               input  logic [ 4:0] a1, a2, a3,
               input  logic [31:0] wd3,
               output logic [31:0] rd1, rd2);

  logic [31:0] rf[31:0];

  // three ported register file
  // read two ports combinationally (A1/RD1, A2/RD2)
  // write third port on rising edge of clock (A3/WD3/WE3)
  // register 0 hardwired to 0

  always_ff @(posedge clk)
    if (we3) rf[a3] <= wd3;

  assign rd1 = (a1 != 0) ? rf[a1] : 0;
  assign rd2 = (a2 != 0) ? rf[a2] : 0;
endmodule

module adder(input  [31:0] a, b,
             output [31:0] y);

  assign y = a + b;
endmodule

module extend(input  logic [31:7] instr,
              input  logic [1:0]  immsrc,
              input  logic        lui,
              output logic [31:0] immext);

  always_comb
    if (lui) begin
      immext = {instr[31:12], 12'b0};
    end
    else
    case(immsrc)
      2'b00:   immext = {{20{instr[31]}}, instr[31:20]}; // I-type
      2'b01:   immext = {{20{instr[31]}}, instr[31:25], instr[11:7]}; // S-type
(stores)
      2'b10:   immext = {{20{instr[31]}}, instr[7], instr[30:25], instr[11:8],
1'b0}; // B-type (branches)
```

```
      2'b11:   immext = {{12{instr[31]}}, instr[19:12], instr[20], instr[30:21],
1'b0};  // J-type (jal)
      default: immext = 32'bx; // undefined
    endcase
endmodule

module flopr #(parameter WIDTH = 8)
              (input  logic             clk, reset, enable,
               input  logic [WIDTH-1:0] d,
               output logic [WIDTH-1:0] q);

  always_ff @(posedge clk, posedge reset)
    if (reset) q <= 0;
    else if (enable)      q <= d;
endmodule

module Dflopr #(parameter WIDTH = 8)
              (input  logic             clk, reset, enable, CLR,
               input  logic [WIDTH-1:0] RD, PCF, PCPlus4F,
               output logic [WIDTH-1:0] PCD, PCPlus4D, InstrD);

  always_ff @(posedge clk, posedge reset)
    if (reset)
    begin
      PCD <= 0;
      PCPlus4D <= 0;
      InstrD <= 0;
    end
   else if(CLR)
    begin
      PCD <= 0;
      PCPlus4D <= 0;
      InstrD <= 0;
    end
    else if (enable)
      begin
        PCD <= PCF;
        PCPlus4D <= PCPlus4F;
        InstrD <= RD;
      end
endmodule

module Eflopr #(parameter WIDTH = 8)
              (input  logic             clk, reset, CLR,
```

```
             input logic [1:0] ResultSrcD,
             input logic       MemWriteD, BranchD,
             input logic       ALUSrcD,
             input logic       RegWriteD, JumpD,
             input logic [2:0] ALUControlD,

             input logic [WIDTH-1:0] RD1D, RD2D,
             input logic [WIDTH-1:0] PCD, InstrD, ExtImmD, PCPlus4D,

             output logic [WIDTH-1:0] RD1E, RD2E,
             output logic [WIDTH-1:0] PCE, InstrE, ExtImmE, PCPlus4E,
             output logic [1:0] ResultSrcE,
             output logic [2:0] ALUControlE,
             output logic RegWriteE, MemWriteE, JumpE, BranchE, ALUSrcE);

always_ff @(posedge clk, posedge reset)
  if (reset)
    begin
      RD1E <= 0;
      RD2E <= 0;
      PCE <= 0;
      InstrE <= 0;
      ExtImmE <= 0;
      ResultSrcE <= 0;
      ALUControlE <= 0;
      RegWriteE <= 0;
      MemWriteE <= 0;
      JumpE <= 0;
      BranchE <= 0;
      ALUSrcE <= 0;
      PCPlus4E <= 0;
    end
  else if (CLR)
    begin
      RD1E <= 0;
      RD2E <= 0;
      PCE <= 0;
      InstrE <= 0;
      ExtImmE <= 0;
      ResultSrcE <= 0;
      ALUControlE <= 0;
      RegWriteE <= 0;
      MemWriteE <= 0;
      JumpE <= 0;
      BranchE <= 0;
```

```
        ALUSrcE <= 0;
        PCPlus4E <= 0;
      end
    else
      begin
        RD1E <= RD1D;
        RD2E <= RD2D;
        PCE <= PCD;
        InstrE <= InstrD;
        ExtImmE <= ExtImmD;
        ResultSrcE <= ResultSrcD;
        ALUControlE <= ALUControlD;
        RegWriteE <= RegWriteD;
        MemWriteE <= MemWriteD;
        JumpE <= JumpD;
        BranchE <= BranchD;
        ALUSrcE <= ALUSrcD;
        PCPlus4E <= PCPlus4D;
      end
endmodule

module Mflopr #(parameter WIDTH = 8)
              (input  logic             clk, reset,

               input logic [1:0] ResultSrcE,
               input logic       MemWriteE,
               input logic       RegWriteE,

               input logic [WIDTH-1:0] ALUResultE, WriteDataE, InstrE, PCPlus4E,

               output logic [WIDTH-1:0] InstrM, PCPlus4M,
               output logic [32:1] ALUResultM, WriteDataM,
               output logic [1:0] ResultSrcM,
               output logic RegWriteM, MemWriteM);

  always_ff @(posedge clk, posedge reset)
    if (reset)
      begin
        InstrM <= 0;
        PCPlus4M <= 0;
        ResultSrcM <= 0;
        RegWriteM <= 0;
        MemWriteM <= 0;
      end
    else
```

```
      begin
        InstrM <= InstrE;
        PCPlus4M <= PCPlus4E;
        ALUResultM <= ALUResultE;
        WriteDataM <= WriteDataE;
        ResultSrcM <= ResultSrcE;
        RegWriteM <= RegWriteE;
        MemWriteM <= MemWriteE;
      end
endmodule

module Wflopr #(parameter WIDTH = 8)
              (input  logic             clk, reset,

               input logic [1:0] ResultSrcM,
               input logic       RegWriteM,

               input logic [WIDTH-1:0] ALUResultM, RD, WriteDataM, InstrM,
PCPlus4M,

               output logic [WIDTH-1:0] ALUResultW, ReadDataW, InstrW, PCPlus4W,
               output logic [1:0] ResultSrcW,
               output logic RegWriteW);

  always_ff @(posedge clk, posedge reset)
    if (reset)
      begin
        ALUResultW <= 0;
        ReadDataW <= 0;
        InstrW <= 0;
        PCPlus4W <= 0;
        ResultSrcW <= 0;
        RegWriteW <= 0;
      end
    else
      begin
        ALUResultW <= ALUResultM;
        ReadDataW <= RD;
        InstrW <= InstrM;
        PCPlus4W <= PCPlus4M;
        ResultSrcW <= ResultSrcM;
        RegWriteW <= RegWriteM;
      end
endmodule
```

```
module hazard(input clk, reset,
              input logic [4:0] Rs1D, Rs1E, //Rs1D, Rs1E
              input logic [4:0] Rs2D, Rs2E, //Rs2D, Rs2E
              input logic [4:0] RdE, RdM, RdW, //RdE, RdM, RdW
              input logic       PCSrcE, RegWriteM, RegWriteW,
              input logic       ResultSrcE, lui,

              output logic      StallF, StallD, FlushD, FlushE,
              output logic [1:0] ForwardAE, ForwardBE);

   logic lui_last;
   always @(posedge clk)
   begin
        lui_last <= lui;
   end
   always_comb
    begin
        ForwardAE[1] = ( ((Rs1E == RdM) && RegWriteM) && (Rs1E != 0));
        ForwardAE[0] = ( ((Rs1E == RdW) && RegWriteW) && (Rs1E != 0) &&
(!ForwardAE[1]));

        ForwardBE[1] = ( ((Rs2E == RdM) && RegWriteM) && (Rs2E != 0));
        ForwardBE[0] = ( ((Rs2E == RdW) && RegWriteW) && (Rs2E != 0) &&
(!ForwardBE[1]));

        StallF = (((Rs1D == RdE) || (Rs2D == RdE)) && ResultSrcE) || (lui &
!lui_last);
        StallD = (((Rs1D == RdE) || (Rs2D == RdE)) && ResultSrcE) || (lui &
!lui_last);

        FlushD = PCSrcE;
        FlushE = StallF || FlushD;
    end
endmodule

module mux2 #(parameter WIDTH = 8)
             (input  logic [WIDTH-1:0] d0, d1,
              input  logic             s,
              output logic [WIDTH-1:0] y);

  assign y = s ? d1 : d0;
endmodule

module mux3 #(parameter WIDTH = 8)
             (input  logic [WIDTH-1:0] d0, d1, d2,
```

```
             input  logic [1:0]       s,
             output logic [WIDTH-1:0] y);

  assign y = s[1] ? d2 : (s[0] ? d1 : d0);
endmodule

module alu(input  logic [31:0] a, b,
           input  logic [2:0]  alucontrol,
           output logic [31:0] result,
           output logic        zero);

  logic [31:0] condinvb, sum;
  logic        v;              // overflow
  logic        isAddSub;       // true when is add or subtract operation
  logic [31:0] set_less_than;

  assign condinvb = alucontrol[0] ? ~b : b;
  assign sum = a + condinvb + alucontrol[0];
  assign isAddSub = ~alucontrol[2] & ~alucontrol[1] |
                    ~alucontrol[1] & alucontrol[0];
  assign set_less_than[0] = a < b;

  always_comb
    case (alucontrol)
      3'b000:  result = sum;                 // add
      3'b001:  result = sum;                 // subtract
      3'b010:  result = a & b;               // and
      3'b011:  result = a | b;               // or
      3'b100:  result = a ^ b;               // xor
      3'b101:  result = {31'b0, a < b};      // slt
      //3'b110:  result = a << b;            // sll
     3'b110:  result = b;             //lui

      3'b111:  result = a >> b;              // srl
      default: result = 32'bx;
    endcase

  assign zero = (result == 32'b0);
  assign v = ~(alucontrol[0] ^ a[31] ^ b[31]) & (a[31] ^ sum[31]) & isAddSub;

endmodule
```

Figure 11: Datapath.sv

```
module controller(input  logic [6:0] op,
```

```
                  input  logic [2:0] funct3,
                  input  logic       funct7b5,
                  input  logic       Zero,
                  output logic [1:0] ResultSrc,
                  output logic       MemWrite,
                  output logic       PCSrc, ALUSrc,
                  output logic       RegWrite, Jump,
                  output logic [1:0] ImmSrc,
                  output logic       Branch, Lui,
                  output logic [2:0] ALUControl);

  logic [1:0] ALUOp;

  maindec md(op, ResultSrc, MemWrite, Branch,
             ALUSrc, RegWrite, Jump, Lui, ImmSrc, ALUOp);
  aludec  ad(op[5], funct3, funct7b5, ALUOp, lui, ALUControl);

endmodule

module maindec(input  logic [6:0] op,
               output logic [1:0] ResultSrc,
               output logic       MemWrite,
               output logic       Branch, ALUSrc,
               output logic       RegWrite, Jump,
               output logic       lui,
               output logic [1:0] ImmSrc,
               output logic [1:0] ALUOp);

  logic [11:0] controls;

  assign {RegWrite, ImmSrc, ALUSrc, MemWrite,
          ResultSrc, Branch, ALUOp, Jump, lui} = controls;

  always_comb
    case(op)
    // RegWrite_ImmSrc_ALUSrc_MemWrite_ResultSrc_Branch_ALUOp_Jump_Lui
      7'b0000011: controls = 12'b1_00_1_0_01_0_00_0_0; // lw
      7'b0100011: controls = 12'b0_01_1_1_xx_0_00_0_0; // sw
      7'b0110011: controls = 12'b1_xx_0_0_00_0_10_0_0; // R-type
      7'b1100011: controls = 12'b0_10_0_0_xx_1_01_0_0; // beq
      7'b0010011: controls = 12'b1_00_1_0_00_0_10_0_0; // I-type ALU
      7'b1101111: controls = 12'b1_11_x_0_10_0_xx_1_0; // jal
      7'b0000011: controls = 12'b1_00_1_0_01_0_00_0_0; // lbu
      7'b0110111: controls = 12'b1_00_1_0_00_0_10_0_1; // U-Type Instruction
```

```
      default:     controls = 12'b0_00_0_0_00_0_00_0_0; // non-implemented
instruction
    endcase
endmodule

module aludec(input  logic       opb5,
              input  logic [2:0] funct3,
              input  logic       funct7b5,
              input  logic [1:0] ALUOp,
          input  logic lui,
              output logic [2:0] ALUControl);

  logic  RtypeSub;
  assign RtypeSub = funct7b5 & opb5;  // TRUE for R-type subtract instruction

  always_comb
      case(ALUOp)
        2'b00:                ALUControl = 3'b000; // addition
        2'b01:                ALUControl = 3'b001; // subtraction
        default: case(funct3) // R-type or I-type ALU
                3'b000:  if (RtypeSub)
                       ALUControl = 3'b001; // sub
                      else
                       ALUControl = 3'b000; // add, addi
                3'b001:    ALUControl = 3'b110; //lui
                //3'b100:    ALUControl = 3'b000; //lbu
                3'b101:    ALUControl = 3'b111; //srl
                3'b010:    ALUControl = 3'b101; // slt, slti
                3'b110:    ALUControl = 3'b011; // or, ori
                3'b111:    ALUControl = 3'b010; // and, andi
                default:   ALUControl = 3'bxxx; // ???
              endcase
      endcase

endmodule
```
Figure 12: controller.sv