Lab 9 - MIPS Multi-Cycle CPU

Chris Escobar, Fabian Torres

Professor Wenjing Rao

ECE 469

Table of Contents

Quartus RTL Schematics

top

mem

mips

datapath

flopr

jumpext

mux2

mux3

signext

regfile

flopenr

sl2

mux4

alu

controller

maindec

state

aludec

Lab Prompt

Introduction

In this part, you will design and build your own multicycle MIPS processor. You will be much more on your own to complete this lab than you have been in the past, but you may reuse any of your hardware (SystemVerilog modules) from previous ones.

Your multicycle processor should match the design from the text, which is reprinted in Figure 1 for your convenience. It should handle the following instructions: {add, sub, and, or, slt, lw, sw, beq, addi, j} The multicycle processor is divided into three units: the *controller, datapath*, and *mem* (memory) units. Note that the *mem* unit contains the shared memory used to hold both data and instructions. Also note that the *controller* unit comprises both the *Main Decoder* that takes $OP_{5:0}$ as inputs and the *ALU Decoder* that takes as inputs $ALUOp_{1:0}$ and the $Funct_{5:0}$ code from the 6 least significant bits of the instruction.The *controller* unit also includes the gates needed to produce the write enable signal, *PCEn*, for the PC register.

Part A: Controller

1. Controller Design

Generating Control Signals

Before you begin developing the hardware for your MIPS multicycle processor, you'll need to determine the correct control signals for each state in the multicycle processor's state transition diagram. This state transition diagram is shown in Figure 7.42 in the book. Complete the output table of the Main Decoder in Table 4. Give the FSM control word in hexadecimal for each state. The first two rows are filled in as examples. Be careful with this step. It takes much longer to debug an erroneous circuit than to design it correctly the first time.

Overall Design

Now you will begin the hardware implementation of your multicycle processor. Read and understand the *mipsmulti.sv* code at the Appendix section, as it provides a framework for your design:

- The mips module instantiates both the datapath and control unit (called the controller module). The controller module in turn instantiates the main decoder module (maindec) and the ALU decoder module (aludec).
- You will be responsible for designing the controller and the datapath.
- The memory is essentially identical to the data memory from Lab 6, as provided for you.

Part B: Datapath Design

Refer to Figure 1 for the hardware modules you need to set up your datapath. In this part, you will design the *datapath* and *mem* units and test your completed MIPS multicycle. Remember that you may reuse hardware from the previous labs (such as the ALU, multiplexers, registers, sign-extension hardware modules, register file, etc.) wherever possible.

All of your registers should take a *Reset* input to reset the initial value to a known state (0). The Instruction Register and PC also require enable inputs. Pay careful attention to bus connections; they are an easy place to make mistakes.

As in Lab 6, it is very helpful to first predict the results of a test program before running the program so that you know what to expect and can discover and track down discrepancies.

Table 5, which is partially completed, lists the expected instruction trace while running the test program. Complete the remainder of the table. Do this before you run simulations so you have a set of expectations to check your results against; otherwise, it is easy to fool yourself into believing that erroneous simulations are correct.

Simulate your processor using the same testbench from lab 6 - below is Fig 7.60 in the textbook as your test bench.

```
#         Assembly              Description                Address    Machine
main:     addi $2, $0, 5        # initialize $2 = 5          0         20020005
          addi $3, $0, 12       # initialize $3 = 12         4         2003000c
          addi $7, $3, -9       # initialize $7 = 3          8         2067fff7
          or   $4, $7, $2       # $4 = (3 OR 5) = 7          c         00e22025
          and  $5, $3, $4       # $5 = (12 AND 7) = 4        10        00642824
          add  $5, $5, $4       # $5 = 4 + 7 = 11            14        00a42820
          beq  $5, $7, end      # shouldn't be taken         18        10a7000a
          slt  $4, $3, $4       # $4 = 12 < 7 = 0            1c        0064202a
          beq  $4, $0, around   # should be taken            20        10800001
          addi $5, $0, 0        # shouldn't happen           24        20050000
around:   slt  $4, $7, $2       # $4 = 3 < 5 = 1             28        00e2202a
          add  $7, $4, $5       # $7 = 1 + 11 = 12           2c        00853820
          sub  $7, $7, $2       # $7 = 12 - 5 = 7            30        00e23822
          sw   $7, 68($3)       # [80] = 7                   34        ac670044
          lw   $2, 80($0)       # $2 = [80] = 7              38        8c020050
          j    end              # should be taken            3c        08000011
          addi $2, $0, 1        # shouldn't happen           40        20020001
end:      sw   $2, 84($0)       # write mem[84] = 7          44        ac020054
```

Part C: MIPS Plus

1. Modifying the MIPS multi-cycle processor

You now need to modify the MIPS multicycle processor by adding the instructions of {bne, xori}. First, modify the MIPS processor datapath to show what changes are necessary. Next, modify the controller FSM, main decoder and ALU decoder as required. Finally, modify the SystemVerilog code as needed to include your modifications.

2. Testing your modified MIPS single-cycle processor

Next, you'll need a test program to verify that your modified processor works. The program should check that your new instructions work properly and that the old ones didn't break. Use the example *test2.asm* for the two additional instructions *bne* and *xori*. At the end of running this program, you should see value 0xFFFF8002 written into M[84]

```
# test2.asm

main:     xori  $8, $0,  0x8000
          addi $9, $0,  -32768
          xori  $10, $8, 0x8001
          beq   $8, $9, there
          slt   $11, $9, $8
          bne   $11, $0,  here
          j     there
here:     bne   $10, $11, there
          xori  $8, $8, 0xFFFF
there:    add   $11, $11, $10
          sub   $8, $10, $8
          sw    $8, 82($11)
```

Convert the program to machine language and put it in a file named *memfile2.dat*. Modify your memory to load this file. Modify the testbench to check for the appropriate address and data value indicating that the simulation succeeded. Run the program and check your results. Debug if necessary. When you are done, print out the waveforms as before and indicate the address and data value written by the *sw* instruction.

When you are finished – congratulations! You have built a microprocessor by yourself and have proven your mastery of microarchitecture, SystemVerilog, FSMs, and logic design!

Completed Tables

Table 4

	14	13	12	11	10	9	8	7	6	5:4	3:2	1:0	
State (Name)	PCWrite	MemWrite	IRWrite	RegWrite	ALUSrcA	Branch	IorD	MemtoReg	RegDst	ALUSrcB[1:0]	PCRsc[1:0]	ALUOp[1:0]	FSM Control Word
0 (Fetch)	1	0	1	0	0	0	0	0	0	01	00	00	0x5010
1 (Decode)	0	0	0	0	0	0	0	0	0	11	00	00	0x0030
2 (MemAdr)	0	0	0	0	1	0	0	0	0	10	00	00	0x0420
3 (MemRd)	0	0	0	0	0	0	1	0	0	00	00	00	0x0100
4 (MemWB)	0	0	0	1	0	0	0	1	0	00	00	00	0x0880
5 (MemWr)	0	1	0	0	0	0	1	0	0	00	00	00	0x2100
6 (RtypeEx)	0	0	0	0	1	0	0	0	0	00	00	10	0x0402
7 (RtypeWB)	0	0	0	1	0	0	0	0	1	00	00	00	0x0840
8 (BeqEx)	0	0	0	0	1	1	0	0	0	00	01	01	0x0605
9 (AddiEx)	0	0	0	0	1	0	0	0	0	10	00	00	0x0420
10 (AddiWB)	0	0	0	1	0	0	0	0	0	00	00	00	0x0809
11 (JEx)	1	0	0	0	0	0	0	0	00	00	10	00	0x4008

Table 5

Cycle	Reset	PC	Instr	(FSM) state	SrcA	SrcB	ALUResult	Zero	Control Word
1	1	00	0	0	00	04	04	0	5010
2	0	04	addi 20020005	1	04	x	x	0	0030
3	0	04	addi 20020005	9	00	05	05	0	0420
4	0	04	addi 20020005	10	x	x	x	0	0800
5	0	04	addi 20020005	0	04	04	08	0	5010
6	0	08	addi 2003000c	1	08	x	x	0	0030
7	0	08	addi 2003000c	9	00	0c	0c	0	0420
8	0	08	addi 2003000c	10	x	x	x	0	0800
9	0	08	addi 2003000C	0	08	04	0C	0	5010
10	0	0C	addi 2067fff7	1	0C	x	x	0	0030
11	0	0C	addi 2067fff7	9	00	F7	F7	0	0420
12	0	0C	addi 2067fff7	10	x	x	x	0	0800
13	0	0C	addi 2067fff7	0	0C	04	10	0	5010
14	0	10	or 00e22025	1	10	x	x	0	0030
15	0	10	or 00e22025	6	03	05	07	0	0402
16	0	10	or 00e22025	7	x	x	x	0	0840
17	0	10	or 00e22025	0	10	04	14	0	5010
18	0	14	and 00642824	1	14	x	x	0	0030
19	0	14	and 00642824	6	0c	07	04	0	0402
20	0	14	and 00642824	7	x	x	x	0	0840
21	0	14	and 00642824	0	14	04	18	0	5010
22	0	18	add 00a42820	1	18	x	x	0	0030
23	0	18	add 00a42820	6	04	07	0b	0	0402
24	0	18	add 00a42820	7	x	x	x	0	0840
25	0	18	add 00a42820	0	18	04	1c	0	5010
26	0	1C	beq 10a7000a	1	1C	X	X	0	0030
27	0	1C	beq 10a7000a	6	B	3	0	1	0605
28	0	1C	beq 10a7000a	0	1C	04	20	0	5010
29	0	20	slt 00642024	1	20	X	X	0	0030
30	0	20	slt 00642024	6	12	7	0	1	0402
31	0	20	slt 00642024	7	X	X	X	0	0840
32	0	20	slt 00642024	0	20	04	24	0	5010
33	0	24	beq 10890001	1	24	X	X	0	0030
34	0	24	beq 10890001	8	0	0	1	0	0605
35	0	24	beq 10890001	0	24	08	2C	0	5010
36	0	2C	slt 00e22024	1	2C	X	X	0	0030
37	0	2C	slt 00e22024	6	3	5	1	0	0402
38	0	2C	slt 00e22024	7	X	X	X		0840
39	0	2C	slt 00e22024	0	2C	04	39	0	5010
40	0	30	add 00853820	1	30	x	x	0	0030
41	0	30	add 00853820	6	01	0b	0c	0	0402
42	0	30	add 00853820	7	x	x	x	0	0840
43	0	30	add 00853820	0	30	04	34	0	5010
44	0	34	sub 00e23822	1	34	x	x	0	0030
45	0	34	sub 00e23822	6	0c	05	07	0	0402
46	0	34	sub 00e23822	7	x	x	x	0	0840
47	0	34	sub 00e23822	0	34	04	38	0	5010
48	0	38	sw ac670044	1	38	x	x	0	0030
49	0	38	sw ac670044	2	0c	44	50	0	0420
50	0	38	sw ac670044	5	x	x	x	0	2100
51	0	38	sw ac670044	0	38	04	3c	0	5010
52	0	3c	lw 8c020050	1	3c	x	x	0	0030
53	0	3c	lw 8c020050	2	00	50	50	0	0420
54	0	3c	lw 8c020050	3	x	x	x	0	0100
55	0	3c	lw 8c020050	4	x	x	x	0	0880
56	0	3c	lw 8c020050	0	3c	04	40	0	5010
57	0	40	j 08000011	1	40	X	X	0	0030
58	0	40	j 08000011	11	x	x	x	0	4008
59	0	40	j 08000011	0	40	04	44	0	5010
60	0	44	sw ac020054	1	44	X	X	0	0030
61	0	44	sw ac020054	2	0	54	54	0	0420
62	0	44	sw ac020054	5	X	X	X	0	2100

Workload Report

How many hours have you spent for this lab entirely?

In total we spent about 10 hours on this lab. About 2 hours for prep, about 3 hours coding, and about 5 hours debugging the code.

Which activity takes the most significant amount of time?

The activity that took the most amount of time was definitely debugging. It was much more time consuming because we had to make sure we knew what results we needed to get, then making changes would sometimes break functionality of the module, or even different modules.

SystemVerilog Programs

mipstop.sv

```
module top(
        input logic clk, reset,
        output logic [31:0] writedata, adr,
        output logic memwrite);

    logic [31:0] readdata;

    // microprocessor (control & datapath)
    mips mips(clk, reset, adr, writedata, memwrite, readdata);

   // memory
    mem mem(clk, memwrite, adr, writedata, readdata);

endmodule
```

mipsmulti.sv

```
//------------------------------------------------------
// mipsmulti.sv
// David_Harris@hmc.edu 8 November 2005
// Update to SystemVerilog 17 Nov 2010 DMH
// Multicycle MIPS processor
//------------------------------------------------

module mips(
        input  logic        clk, reset,
        output logic [31:0] adr, writedata,
        output logic        memwrite,
        input  logic [31:0] readdata);

    logic zero, bne, pcen, irwrite, regwrite, alusrca, iord, memtoreg, regdst;
    logic [1:0]  alusrcb, pcsrc;
    logic [2:0]  alucontrol;
    logic [5:0]  op, funct;
    logic [15:0] controls;

    controller c(clk, reset, op, funct, zero,
        bne, pcen, memwrite, irwrite, regwrite,
        alusrca, iord, memtoreg, regdst,
        alusrcb, pcsrc, alucontrol, controls);

    datapath dp(clk, reset,
        pcen, irwrite, regwrite,
        alusrca, iord, memtoreg, regdst,
        alusrcb, pcsrc, alucontrol,
        op, funct, zero,
        adr, writedata, readdata);
endmodule

module controller(
        input  logic       clk, reset,
        input  logic [5:0] op, funct,
        input  logic       zero,
        output logic       bne, pcen, memwrite, irwrite, regwrite,
        output logic       alusrca, iord, memtoreg, regdst,
        output logic [1:0] alusrcb, pcsrc,
        output logic [2:0] alucontrol,
```

```
        output logic [15:0] controls);

    logic [1:0] aluop;
    logic       branch, pcwrite;

  // Main Decoder and ALU Decoder subunits.
    maindec md(clk, reset, op,
        bne, pcwrite, memwrite, irwrite, regwrite,
        alusrca, branch, iord, memtoreg, regdst,
        alusrcb, pcsrc, aluop, controls);

    aludec  ad(funct, aluop, alucontrol);
    // ADD CODE HERE
    // Add combinational logic (i.e. an assign statement)
    // to produce the PCEn signal (pcen) from the branch,
    // zero, and pcwrite signals
    assign pcen = pcwrite | ((branch | bne) & zero);
endmodule

module maindec(
        input  logic       clk, reset,
        input  logic [5:0] op,
        output logic       bne, pcwrite, memwrite, irwrite, regwrite,
        output logic       alusrca, branch, iord, memtoreg, regdst,
        output logic [1:0] alusrcb, pcsrc,
        output logic [1:0] aluop,
        output logic [15:0] controls);

    parameter   FETCH   = 4'b0000; // State 0
    parameter   DECODE  = 4'b0001; // State 1
    parameter   MEMADR  = 4'b0010;  // State 2
    parameter   MEMRD   = 4'b0011;  // State 3
    parameter   MEMWB   = 4'b0100;  // State 4
    parameter   MEMWR   = 4'b0101;  // State 5
    parameter   RTYPEEX = 4'b0110;  // State 6
    parameter   RTYPEWB = 4'b0111;  // State 7
    parameter   BEQEX   = 4'b1000;  // State 8
    parameter   ADDIEX  = 4'b1001;  // State 9
    parameter   ADDIWB  = 4'b1010;  // state 10
    parameter   JEX     = 4'b1011;  // State 11
    parameter   XORIEX  = 4'b1100;  // State 12
```

```
    parameter   XORIWB  = 4'b1101;  // state 13
    parameter   BNEEX   = 4'b1110;  // State 14

    parameter   LW      = 6'b100011;    // Opcode for lw
    parameter   SW      = 6'b101011;    // Opcode for sw
    parameter   RTYPE   = 6'b000000;    // Opcode for R-type
    parameter   BEQ     = 6'b000100;    // Opcode for beq
    parameter   BNE     = 6'b000101;    // Opcode for bne
    parameter   ADDI    = 6'b001000;    // Opcode for addi
    parameter   XORI    = 6'b001110;    // Opcode for xori
    parameter   J       = 6'b000010;    // Opcode for j

    logic [3:0]  state, nextstate;

    // state register
    always_ff @(posedge clk or posedge reset)
        if(reset) state <= FETCH;
        else state <= nextstate;
    // ADD CODE HERE
    // Finish entering the next state logic below.  We've completed the first
    // two states, FETCH and DECODE, for you.

    // next state logic
    always_comb
        case(state)
            FETCH:   nextstate <= DECODE;
            DECODE:  case(op)
                LW:      nextstate <= MEMADR;
                SW:      nextstate <= MEMADR;
                RTYPE:   nextstate <= RTYPEEX;
                BEQ:     nextstate <= BEQEX;
                BNE:     nextstate <= BNEEX;
                ADDI:    nextstate <= ADDIEX;
                XORI:    nextstate <= XORIEX;
                J:       nextstate <= JEX;
                default: nextstate <= 4'bx; // should never happen
            endcase
            // Add code here
            MEMADR: case(op)
                LW:         nextstate = MEMRD;
                SW:         nextstate = MEMWR;
```

```
            default:    nextstate = 4'bx;
        endcase

        MEMRD:      nextstate = MEMWB;
        MEMWB:      nextstate = FETCH;
        MEMWR:      nextstate = FETCH;
        RTYPEEX:    nextstate = RTYPEWB;
        RTYPEWB:    nextstate = FETCH;
        BEQEX:      nextstate = FETCH;
        BNEEX:      nextstate = FETCH;
        ADDIEX:     nextstate = ADDIWB;
        ADDIWB:     nextstate = FETCH;
        XORIEX:     nextstate = XORIWB;
        XORIWB:     nextstate = FETCH;
        JEX:        nextstate = FETCH;
        default: nextstate <= 4'bx; // should never happen
    endcase

// output logic
assign {bne, pcwrite, memwrite, irwrite,
    regwrite, alusrca, branch, iord,
    memtoreg, regdst, alusrcb,
    pcsrc, aluop} = controls;

// ADD CODE HERE
// Finish entering the output logic below.  We've entered the
// output logic for the first two states, S0 and S1, for you.
always_comb
    case(state)
        FETCH:      controls = 16'h5010;
        DECODE:     controls = 16'h0030;
    // your code goes here
        MEMADR:     controls = 16'h0420;
        MEMRD:      controls = 16'h0100;
        MEMWB:      controls = 16'h0880;
        MEMWR:      controls = 16'h2100;
        RTYPEEX:    controls = 16'h0402;
        RTYPEWB:    controls = 16'h0840;
        BEQEX:      controls = 16'h0605;
        BNEEX:      controls = 16'h8205;
        ADDIEX:     controls = 16'h0420;
```

```
            ADDIWB:     controls = 16'h0800;
            XORIEX:     controls = 16'h0423;
            XORIWB:     controls = 16'h0803;
            JEX:        controls = 16'h4008;
        default: controls <= 16'hxxxx; // should never happen
    endcase
endmodule

module aludec(input  logic [5:0] funct,
        input  logic [1:0] aluop,
        output logic [2:0] alucontrol);

    // ADD CODE HERE
    // Complete the design for the ALU Decoder.
    // Your design goes here.  Remember that this is a combinational
    // module.
    always_comb case(aluop)
        2'b00: alucontrol = 3'b010;  // add
        2'b01: alucontrol = 3'b110;  // sub
        2'b11: alucontrol = 3'b011;  // xori
        default: case(funct)          // RTYPE
            6'b100000: alucontrol = 3'b010; // ADD
            6'b100010: alucontrol = 3'b110; // SUB
            6'b100100: alucontrol = 3'b000; // AND
            6'b100101: alucontrol = 3'b001; // OR
            6'b101010: alucontrol = 3'b111; // SLT
            default:   alucontrol = 3'bxxx; // ???
        endcase
    endcase
    // Remember that you may also reuse any code from previous labs.
endmodule

// Complete the datapath module below for Lab 11.
// You do not need to complete this module for Lab 10

// The datapath unit is a structural verilog module.  That is,
// it is composed of instances of its sub-modules.  For example,
// the instruction register is instantiated as a 32-bit flopenr.
// The other submodules are likewise instantiated.

module datapath(
```

```
    input  logic        clk, reset,
    input  logic        pcen, irwrite, regwrite,
    input  logic        alusrca, iord, memtoreg, regdst,
    input  logic [1:0]  alusrcb, pcsrc,
    input  logic [2:0]  alucontrol,
    output logic [5:0]  op, funct,
    output logic        zero,
    output logic [31:0] adr, writedata,
    input  logic [31:0] readdata);

// Below are the internal signals of the datapath module.
logic [4:0]  writereg;
logic [31:0] pcnext, pc, pcjump;
logic [31:0] instr, data, srca, srcb;
logic [31:0] a, b;
logic [31:0] aluresult, aluout;
logic [31:0] signimm;   // the sign-extended immediate
logic [31:0] signimmsh; // the sign-extended immediate shifted left by 2
logic [31:0] wd3, rd1, rd2;

// op and funct fields to controller
assign op = instr[31:26];
assign funct = instr[5:0];

// Your datapath hardware goes below.  Instantiate each of the submodules
// that you need.  Remember that alu's, mux's and various other
// versions of parameterizable modules are available in mipsparts.sv
// from Lab 9. You'll likely want to include this verilog file in your
// simulation.

// We've included parameterizable 3:1 and 4:1 muxes below for your use.

// Remember to give your instantiated modules applicable names
// such as pcreg (PC register), wdmux (Write Data Mux), etc.
// so it's easier to understand.

// ADD CODE HERE
// datapath

flopenr #(32) instruction_reg(clk, reset, irwrite, readdata, instr);
flopenr #(32) pc_prime(clk, reset, pcen, pcnext, pc);
```

```
    flopr   #(32) data_reg(clk, reset, readdata, data);
    flopr   #(32) alu_reg(clk, reset, aluresult, aluout);
    flopr   #(32) regA(clk, reset, rd1, a);
    flopr   #(32) regB(clk, reset, rd2, b);

    mux2    #(32) adr_mux(pc, aluout, iord, adr);
    mux2    #(5)  a3_mux(instr[20:16], instr[15:11], regdst, writereg);
    mux2    #(32) wd3_reg(aluout, data, memtoreg, wd3);
    mux2    #(32) scrA_mux(pc, a, alusrca, srca);
    mux4    #(32) srcB_mux(b, 32'd4, signimm, signimmsh, alusrcb, srcb);
    mux3    #(32) pcnext_mux(aluresult, aluout, pcjump, pcsrc, pcnext);

    signext sign_ext(instr[15:0], signimm);
    sl2 sign_ext_sl2(signimm, signimmsh);
    jumpext jump_ext(pc[31:28], instr[25:0], pcjump);

    regfile register_file(clk, regwrite, instr[25:21], instr[20:16], writereg, wd3, rd1,
rd2);

    alu mc_alu(srca, srcb, alucontrol, aluresult, zero);

    assign writedata = b;
endmodule

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

    logic [31:0] RAM[63:0];
    // initialize memory with instructions
    initial begin
        $readmemh("memfile.dat", RAM);
        // "memfile.dat" contains your instructions in hex
        // you must create this file
    end

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

    always_ff @(posedge clk)
```

```
        if (we)
            RAM[a[31:2]] <= wd;
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 #1 y = s[1] ? d2 : (s[0] ? d1 : d0);
endmodule

module mux4 #(parameter WIDTH = 8) (
        input  logic [WIDTH-1:0] d0, d1, d2, d3,
        input  logic [1:0]       s,
        output logic [WIDTH-1:0] y);

    always_comb
        case(s)
            2'b00: y <= d0;
            2'b01: y <= d1;
            2'b10: y <= d2;
            2'b11: y <= d3;
        endcase
endmodule
```

mipsparts.sv

```
// Components used in MIPS processor
module regfile(
        input  logic        clk, we3,
        input  logic [4:0]  ra1, ra2, wa3,
        input  logic [31:0] wd3,
        output logic [31:0] rd1, rd2);

    logic [31:0] rf[31:0];
  // three ported register file with register 0 hardwired to 0
  // read two ports combinationally; write third port on rising edge of clock
    always_ff @(posedge clk)
        if (we3) rf[wa3] <= wd3;

    assign rd1 = (ra1 != 0) ? rf[ra1] : 0;
    assign rd2 = (ra2 != 0) ? rf[ra2] : 0;
endmodule

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

    assign y = a + b;
endmodule

module sl2(
        input  logic [31:0] a,
        output logic [31:0] y);

    assign y = {a[29:0], 2'b00};        // shift left by 2
endmodule

module signext(
        input  logic [15:0] a,
        output logic [31:0] y);

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

module jumpext(
        input logic [3:0] PC,
```

```
        input logic [25:0] instr,
        output logic [31:0] y);

    assign y = {PC, instr[25:0], 2'b00};
endmodule

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

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

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

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

alu.sv

```
module alu #(parameter N=32)(
        input logic [N-1:0] A, B,
        input logic [2:0] F,
        output logic [N-1:0] Y,
        output logic zero);

    logic [N-1:0] X, X_ext, XORI, approximate, X11_output;
    wire [N-1:0] newB, AB, A_or_B;

    mux2_1 newB_mux(B, ~B, F[2], newB);
    and_32 allands(A, newB, AB);
    or_32 allors(A, newB, A_or_B);
    add_sub_slt thingy(A, newB, F[2], X, OF);
    assign X_ext = {31'h00000000, X[N-1]};
    assign XORI = A ^ newB;
    mux2_1 #(32) slt_or_xori(XORI, X_ext, F[2], X11_output);  // xori = 011, slt = 111
    mux4_1 finalOut(AB, A_or_B, X, X11_output, F[1:0], Y);
    wide_or zeroz(Y, zero);
endmodule

module add_sub_slt #(parameter N=32)(
        input logic [N-1:0] A, B,
        input logic F2,
        output logic [N-1:0] Y,
        output logic OF);

    logic [N-1:0]temp;
    wire Cout, eq, gt, slt;
    adder_32_b operation(A, B, F2, Y, Cout, OF);
    comparator_32_b slt_finder(A, B, eq, slt, gt);

endmodule

module adder_32_b #(parameter N=32)
        (input logic [N-1:0] A, B,
        input logic Cin,
        output logic [N-1:0] S,
        output logic Cout, OF);
    logic [31:0] w;
```

```
    one_b_FA m0(A[0], B[0], Cin, S[0], w[0]);

    genvar i;
    generate
        for (i = 1; i < N; i = i + 1)
            begin : FA_chain
                one_b_FA mX(A[i], B[i], w[i-1], S[i], w[i]);
            end
    endgenerate

    assign OF = w[30] ^ w[31];
endmodule

module and_32 #(parameter N=32)(
       input logic [N-1:0] A, B,
       output logic [N-1:0] Y);

    assign Y = (A & B);
endmodule

module apm #(parameter N=32)(
        input logic [N-1:0] A, B,
        output logic [N-1:0] Y);

    logic [(N/2):0] map_Y;
    logic [1:0] two_s0, two_s1, two_s2;
    logic [2:0] three_s0;
    logic [3:0] four_s0;
    wire [5:0] S;
    wire [10:0] Cout;
    logic const_zero;

    assign const_zero = 0;

    genvar i;
    generate
        for (i = 0; i < (N/2) + 1; i = i + 1)
            begin : map_chain
                map chain(A[i+15:i], B, map_Y[i]);
            end
```

```
        for (i = 0; i < 5; i = i + 1)
            begin : adder_chain
                one_b_FA chain(map_Y[3*i], map_Y[3*i+1], map_Y[3*i+2], S[i], Cout[i]);
            end
    endgenerate
    one_b_FA one5(map_Y[15], map_Y[16], const_zero, S[5], Cout[5]);

    two_b_FA two0({Cout[0], S[0]}, {Cout[1], S[1]}, const_zero, two_s0, Cout[6]);
    two_b_FA two1({Cout[2], S[2]}, {Cout[3], S[3]}, const_zero, two_s1, Cout[7]);
    two_b_FA two2({Cout[4], S[4]}, {Cout[5], S[5]}, const_zero, two_s2, Cout[8]);

    three_b_FA three0({Cout[6], two_s0}, {Cout[7], two_s1}, const_zero, three_s0,
Cout[9]);

    four_b_FA four0({Cout[9], three_s0}, {const_zero, Cout[8], two_s2}, const_zero,
four_s0, Cout[10]);

    assign Y = {map_Y[16:0], 10'b0000000000, Cout[10], four_s0};
endmodule

module comparator_32_b #(parameter N=32)(
        input logic [N-1:0] A, B,
        output logic eq, lt, gt);

    assign eq = (A == B);
    assign lt = (A < B);
    assign gt = (A > B);
endmodule

module comparator_u32 #(parameter N=32)(    // for SLTU
        input logic unsigned [N-1:0] A, B,
        output logic eq, lt, gt);

    assign eq = (A == B);
    assign lt = (A < B);
    assign gt = (A > B);
endmodule

module four_b_FA #(parameter N=4)(
        input logic [N-1:0] A, B,
```

```
        input logic Cin,
        output logic [N-1:0] S,
        output logic Cout);
    logic [N:0] w_Carry;

    assign w_Carry[0] = Cin;

    genvar i;
    generate
        for (i = 0; i < N; i=i+1)
            begin : adderChain
                one_b_FA chain(A[i], B[i], w_Carry[i], S[i], w_Carry[i+1]);
            end
    endgenerate

    assign Cout = w_Carry[N];
endmodule

module map #(parameter N=32)(
        input logic [(N/2)-1:0] A,
        input logic [N-1:0] B,
        output logic Y);

    logic [(N/2)-1:0] xnor_out, mux_out, B_pattern, B_rule;
    wire const_one = 1;

    assign B_pattern = {B[((N/2)-1):0]};
    assign B_rule = {B[N-1:(N/2)]};

    genvar i;
    generate
        for (i = 0; i < (N/2); i = i + 1)
            begin : xnor_chain
                assign xnor_out[i] = ~(A[i] ^ B_pattern[i]);
                mux2_1_1b dontCare(xnor_out[i], const_one, B_rule[i], mux_out[i]);
            end
    endgenerate

    assign Y = (&mux_out);
endmodule
```

```
module mux2_1 #(parameter N=32)
        (input logic [N-1:0] a0, a1,
        input logic s,
        output [N-1:0] Y);
    assign Y = s ? a1 : a0;
endmodule

module mux2_1_1b #(parameter N=32)
        (input logic a0, a1,
        input logic s,
        output Y);
    assign Y = s ? a1 : a0;
endmodule

module mux4_1 #(parameter N=32)
        (input logic [N-1:0] A, B, C, D,
        input logic [1:0] S,
        output logic [N-1:0] Y);

    always_comb
        begin
            case(S)
                2'b00: Y = A;
                2'b01: Y = B;
                2'b10: Y = C;
                2'b11: Y = D;
            endcase
        end
endmodule

module one_b_FA(
        input logic A, B, Cin,
        output logic S, Cout);
    logic xor_ab, cab, and_ab;
    assign xor_ab = A ^ B;
    assign cab = Cin & xor_ab;
    assign and_ab = A & B;
    assign S = xor_ab ^ Cin;
    assign Cout = cab ^ and_ab;
endmodule
```

```
module or_32 #(parameter N=32)(
       input logic [N-1:0] A, B,
       output logic [N-1:0] Y);

    assign Y = (A | B);
endmodule

module or_bitwise #(parameter N = 32)
        (input logic [N-1:0] A, B,
        output logic [N-1:0] Y,
        output logic Z);
    assign Y = A | B;
    assign Z = ~(|Y);
endmodule

module three_b_FA #(parameter N=3)(
        input logic [N-1:0] A, B,
        input logic Cin,
        output logic [N-1:0] S,
        output logic Cout);
    logic [N:0] w_Carry;

    assign w_Carry[0] = Cin;

    genvar i;
    generate
        for (i = 0; i < N; i=i+1)
            begin : adderChain
                one_b_FA chain(A[i], B[i], w_Carry[i], S[i], w_Carry[i+1]);
            end
    endgenerate

    assign Cout = w_Carry[N];
endmodule

module two_b_FA #(parameter N=2)(
        input logic [N-1:0] A, B,
        input logic Cin,
        output logic [N-1:0] S,
        output logic Cout);
```

```
    logic [N:0] w_Carry;

    assign w_Carry[0] = Cin;

    genvar i;
    generate
        for (i = 0; i < N; i=i+1)
            begin : adderChain
                one_b_FA chain(A[i], B[i], w_Carry[i], S[i], w_Carry[i+1]);
            end
    endgenerate

    assign Cout = w_Carry[N];
endmodule

module wide_or #(parameter N=32)(
       input logic [N-1:0] X,
       output logic Y);

    assign Y = ~(|X);
endmodule
```

testbench.sv

```
module testbench();

    logic        clk;
    logic        reset;
    logic [31:0] writedata, adr;
    logic        memwrite;

    // instantiate device to be tested
    top dut(.*);
    // initialize test
    initial
        begin
            reset <= 1; #9;
            reset <= 0;
        end
    // generate clock to sequence tests
    always
        begin
            clk <= 0; # 5;
            clk <= 1; # 5;
        end
endmodule
```

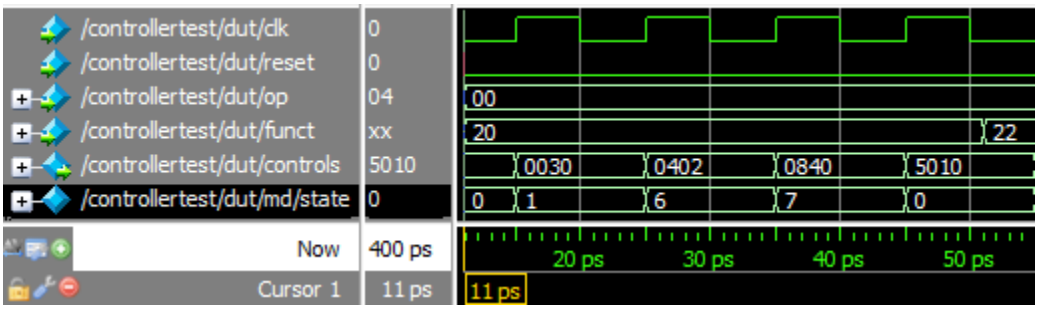
controllertest.sv

```
module controllertest();
    //input logic
    logic clk;
    logic reset;
    logic [5:0] op;
    logic [5:0] funct;
    logic zero;
    //output logic
    logic bne, pcen, memwrite, irwrite, regwrite, alusrca, iord, memtoreg, regdst,
pc_enable;
    logic [1:0] alusrcb, pcsrc;
    logic [2:0] alucontrol;
    logic [3:0] state;
    logic [15:0] controls;

    controller dut(.*);

    always begin
        clk = 0; #5;
        clk = 1; #5;
    end

    initial begin
        reset = 1;                              zero = 0;   #11;
        reset = 0;  op = 6'h0;  funct = 6'h20;              #40;  // add    1, 6, 7, 0
                    op = 6'h0;  funct = 6'h22;              #40;  // sub    1, 6, 7, 0
                    op = 6'h0;  funct = 6'h24;              #40;  // and    1, 6, 7, 0
                    op = 6'h0;  funct = 6'h25;              #40;  // or     1, 6, 7, 0
                    op = 6'h0;  funct = 6'h2a;              #40;  // slt    1, 6, 7, 0
                    op = 6'h08; funct = 6'hx;               #40;  // addi   1, 9, A, 0
                    op = 6'h2b; funct = 6'hx;               #40;  // sw     1, 2, 5, 0
                    op = 6'h23; funct = 6'hx;               #50;  // lw     1, 2, 3, 4,
0
                    op = 6'h04; funct = 6'hx;   zero = 0;   #30;  // beq    1, 8, 0
                    op = 6'h04; funct = 6'hx;   zero = 1;   #30;  // beq    1, 8, 0
                    op = 6'h02; funct = 6'hx;               #30;  // j      1, B, 0
    end
endmodule
```

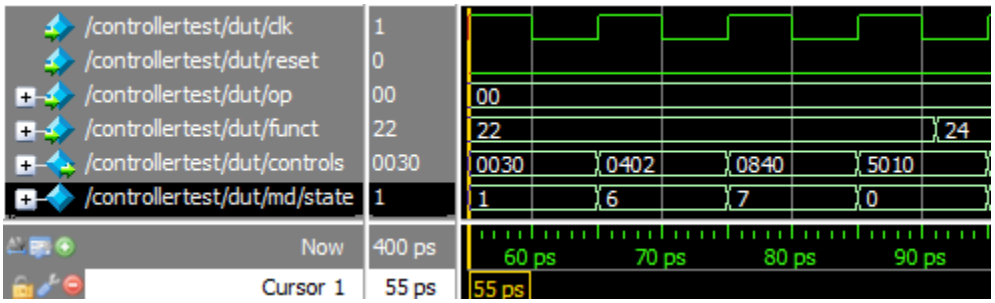
ModelSim Results

controllertest.sv

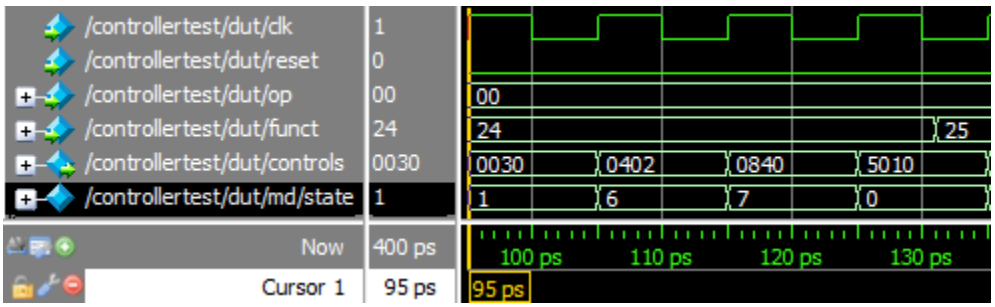
add



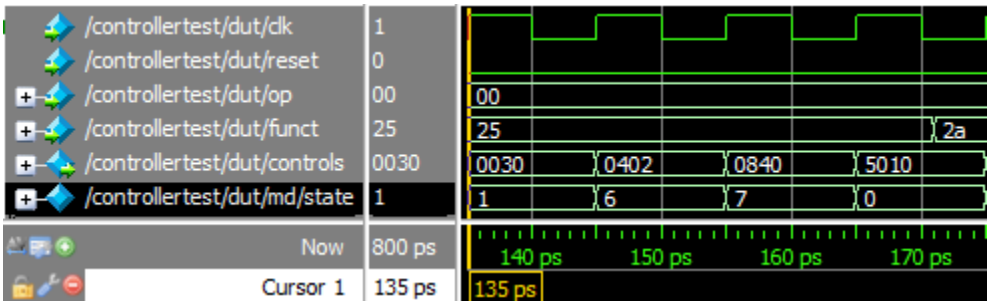
sub



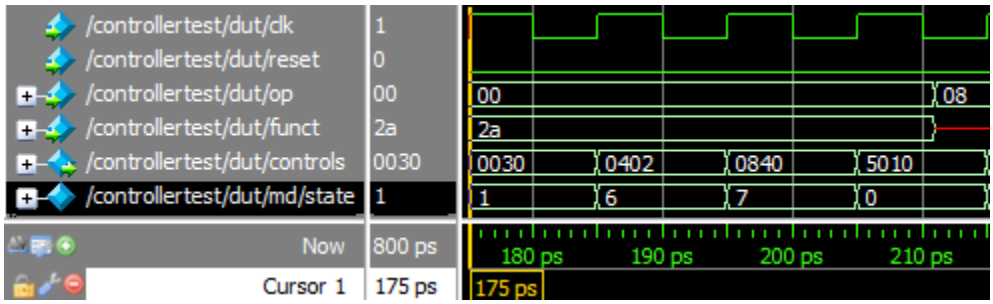
and



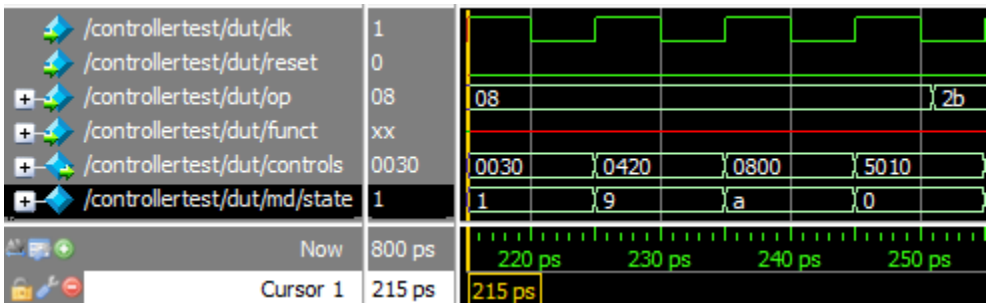
or



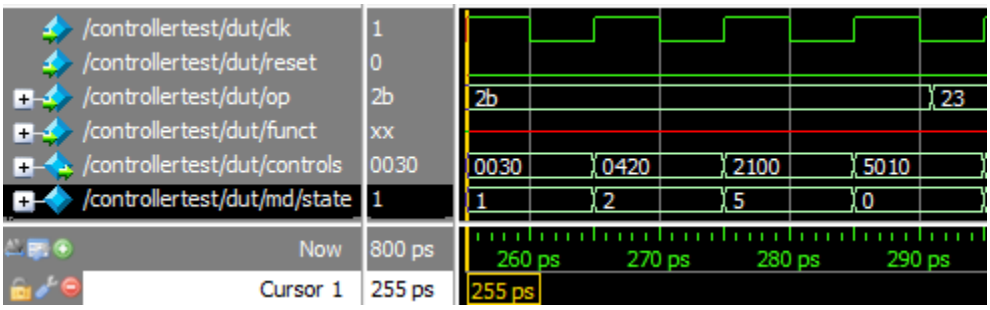
slt



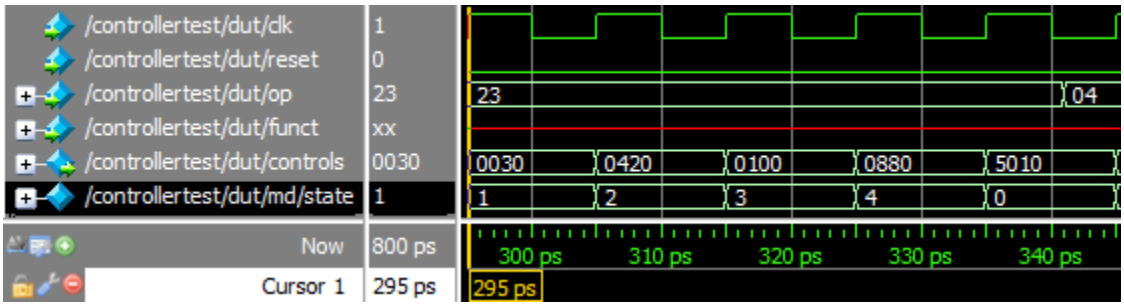
addi



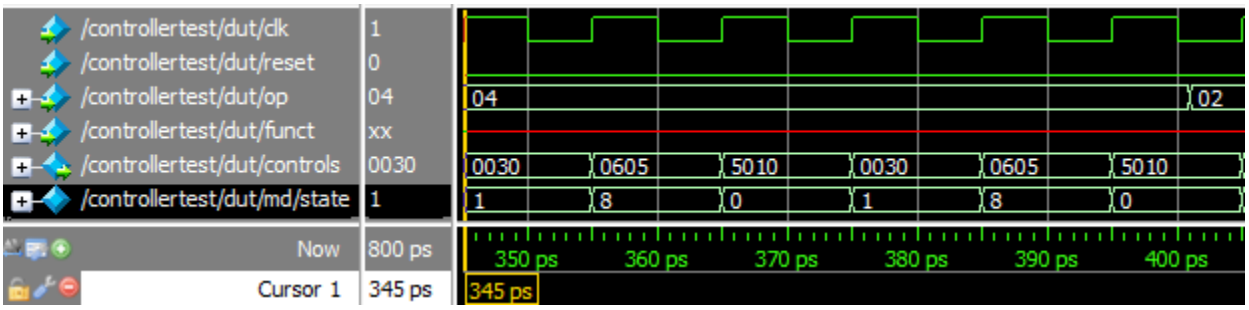
sw



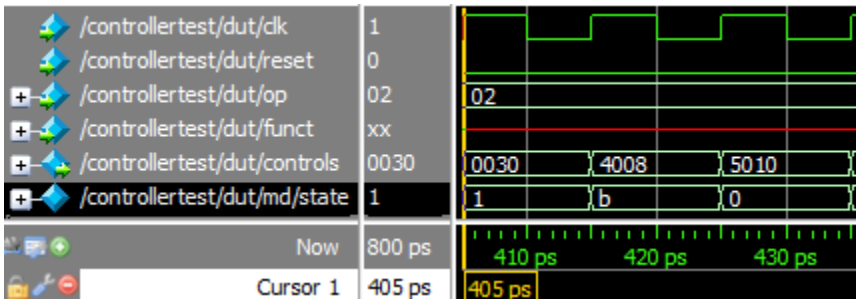
lw



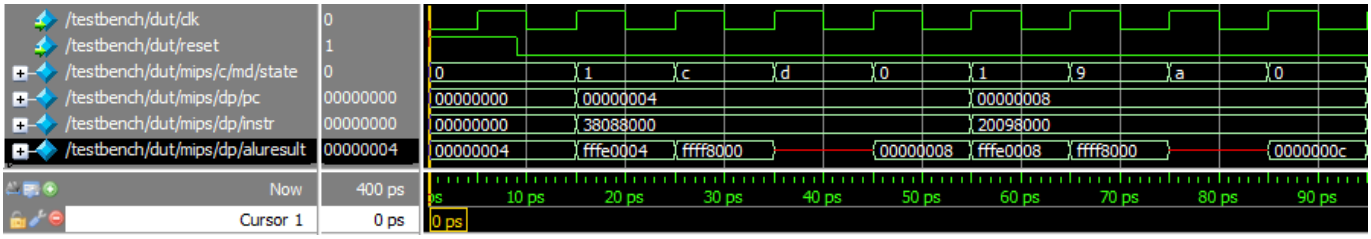
beq

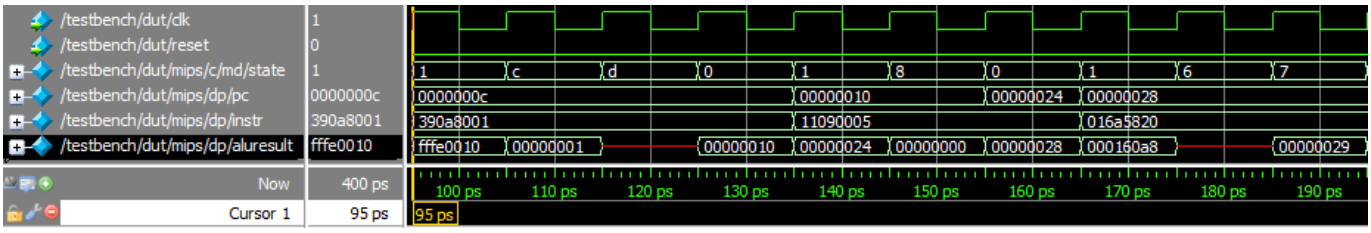


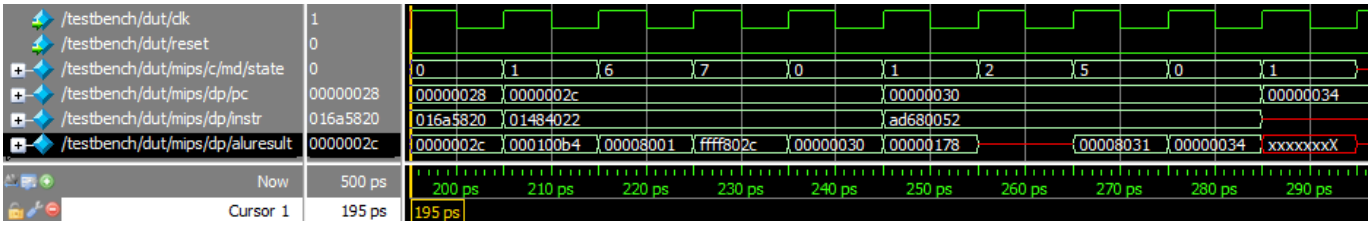
j



testbench.sv (memfile2.dat)

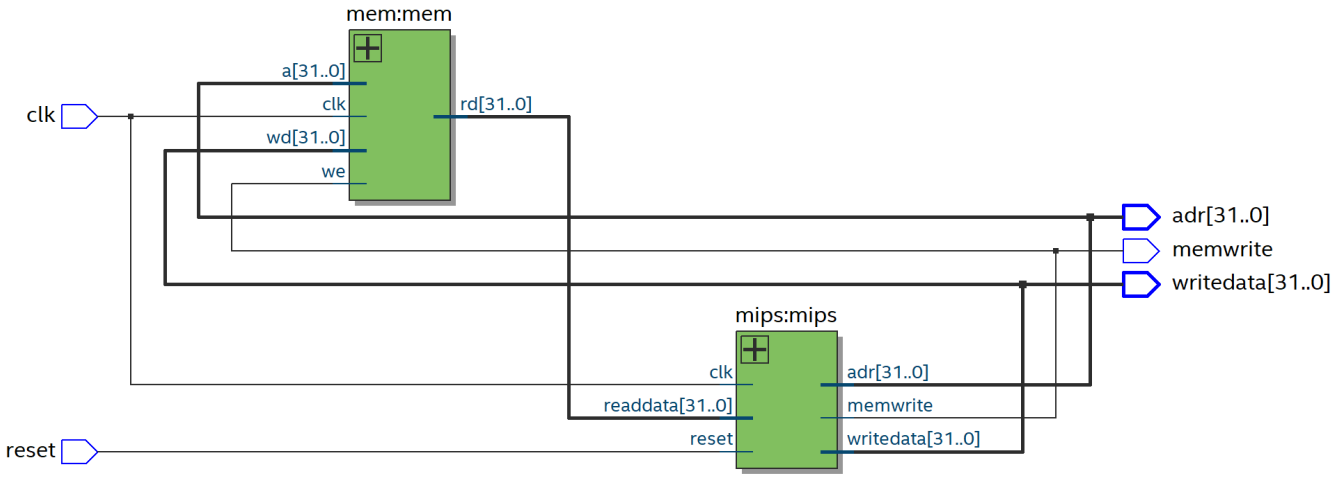




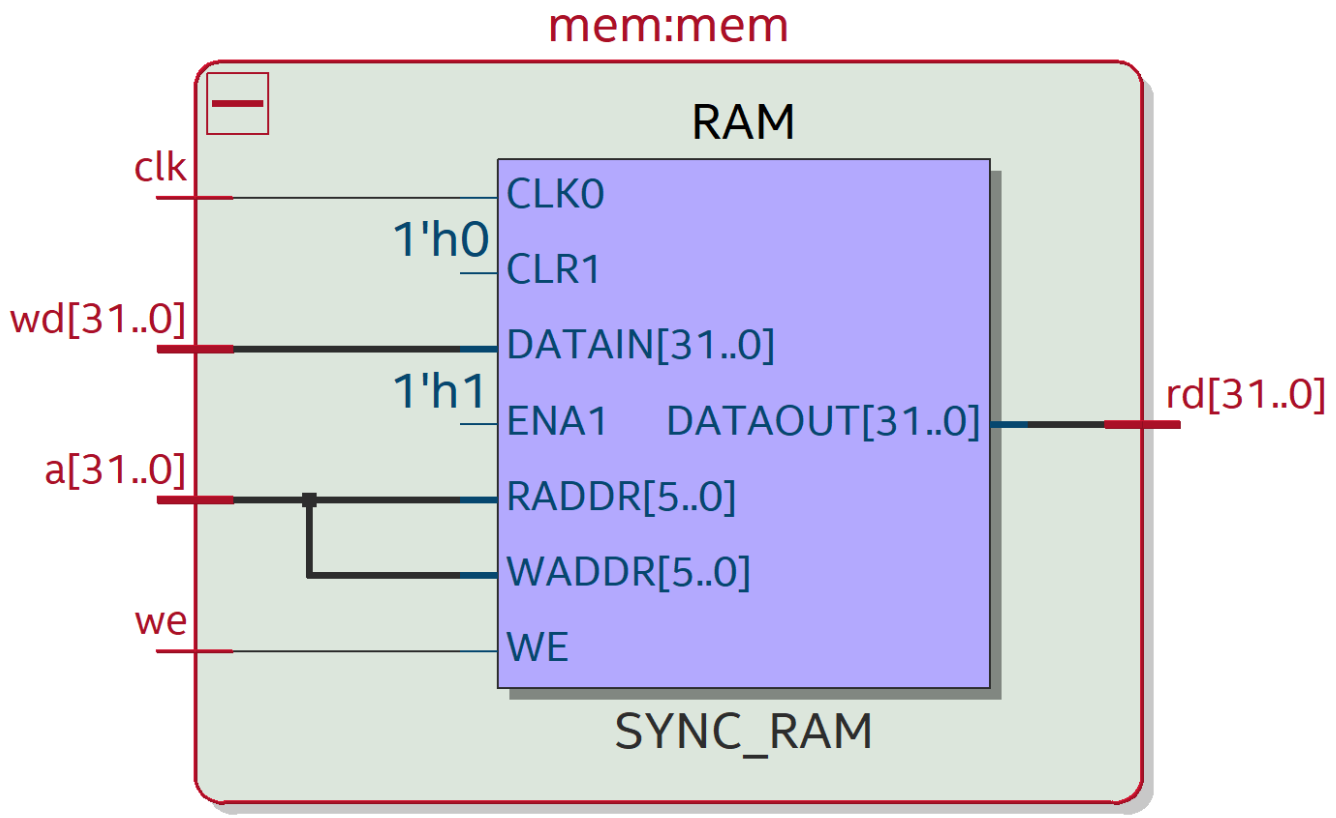


Quartus RTL Schematics

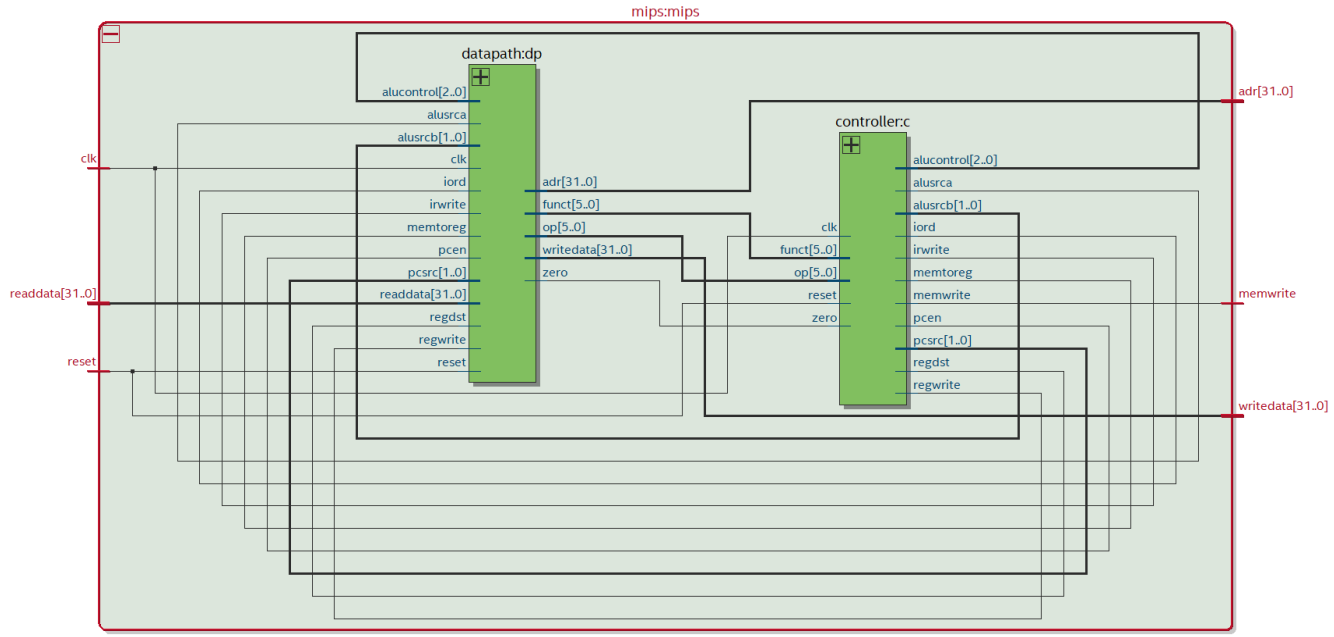
top



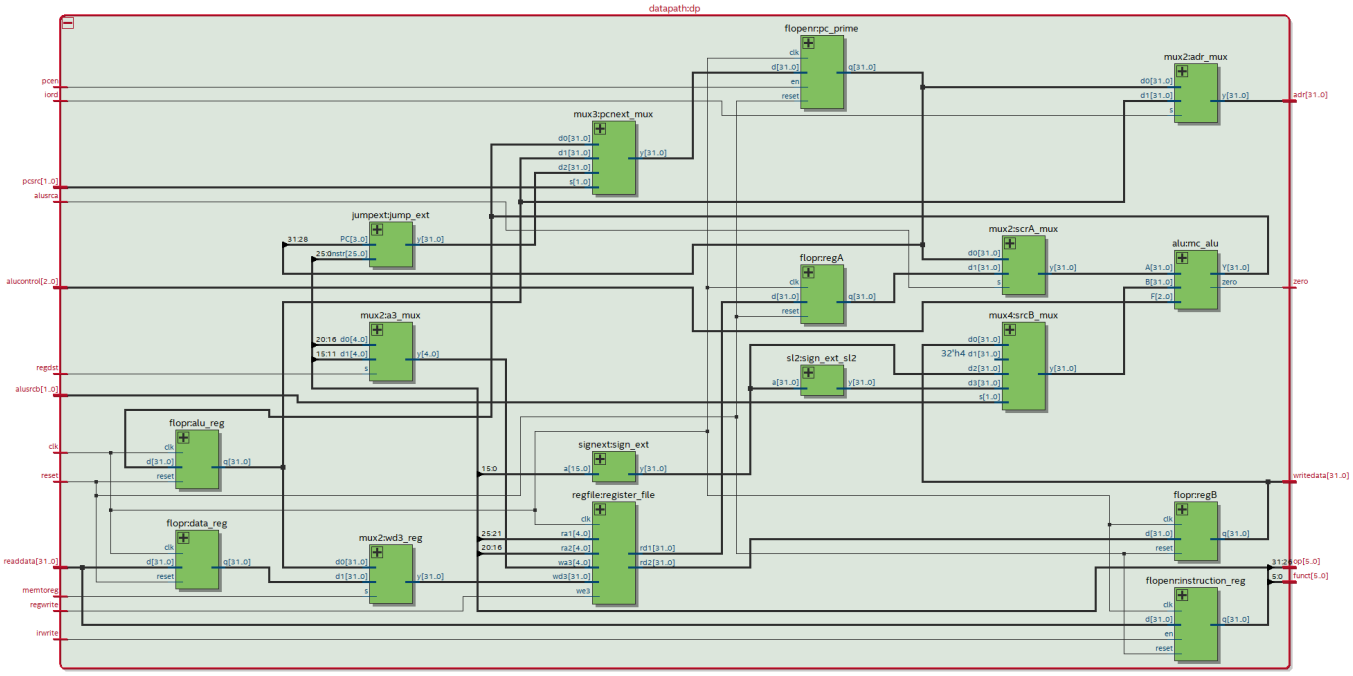
mem



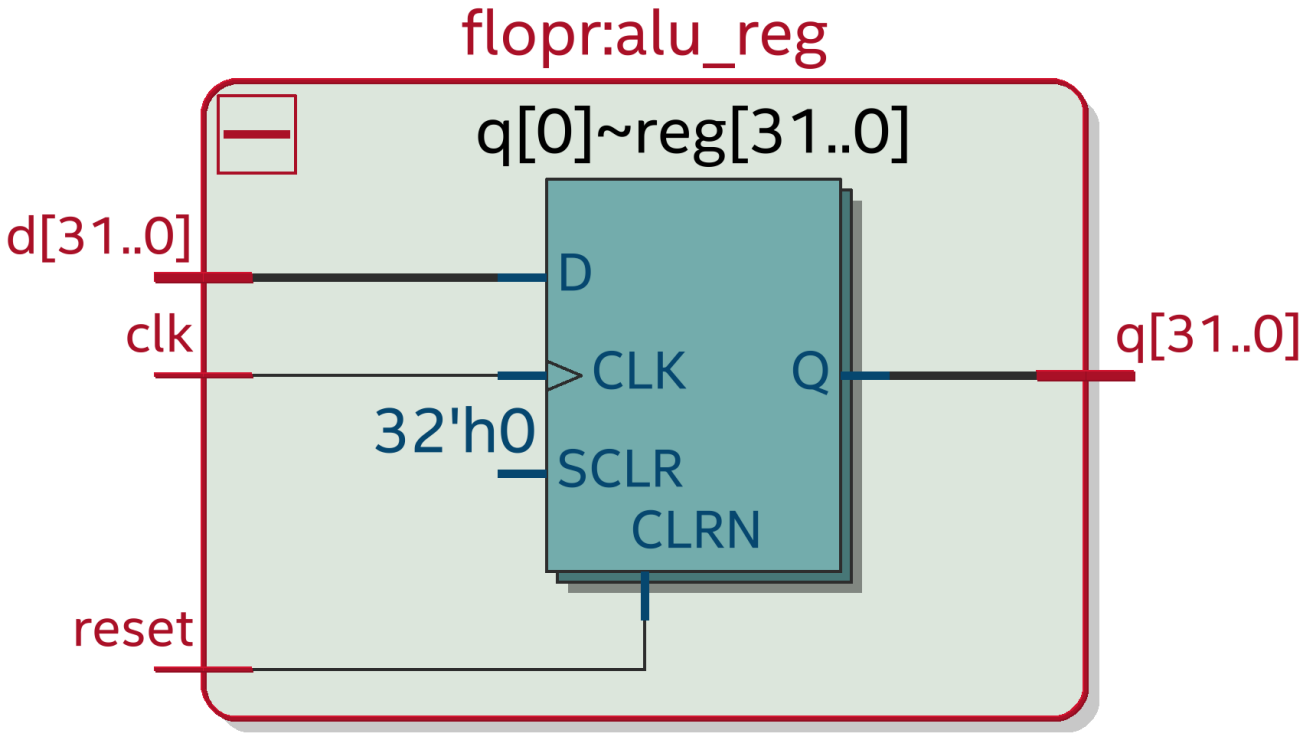
mips



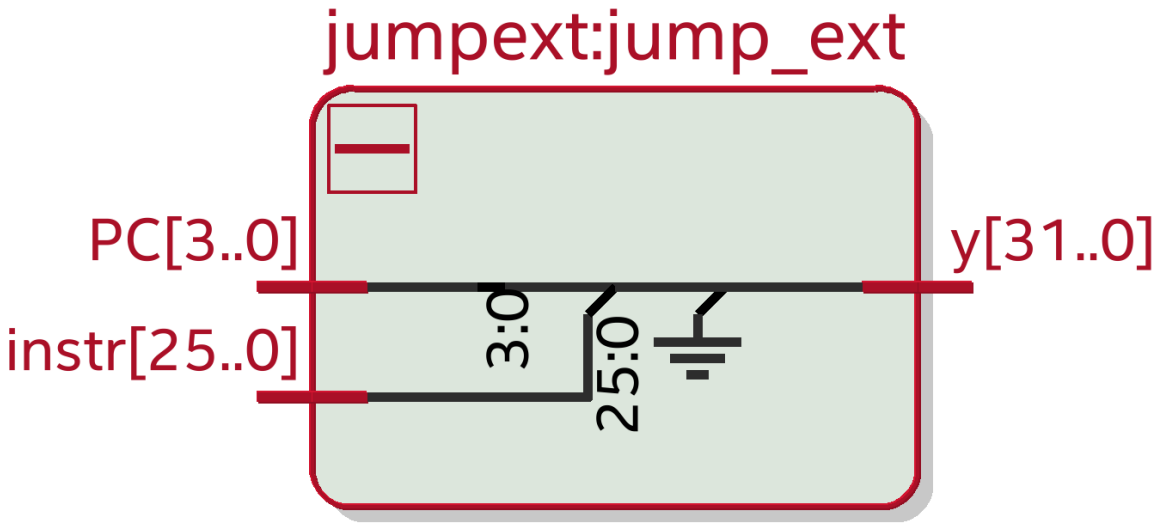
datapath



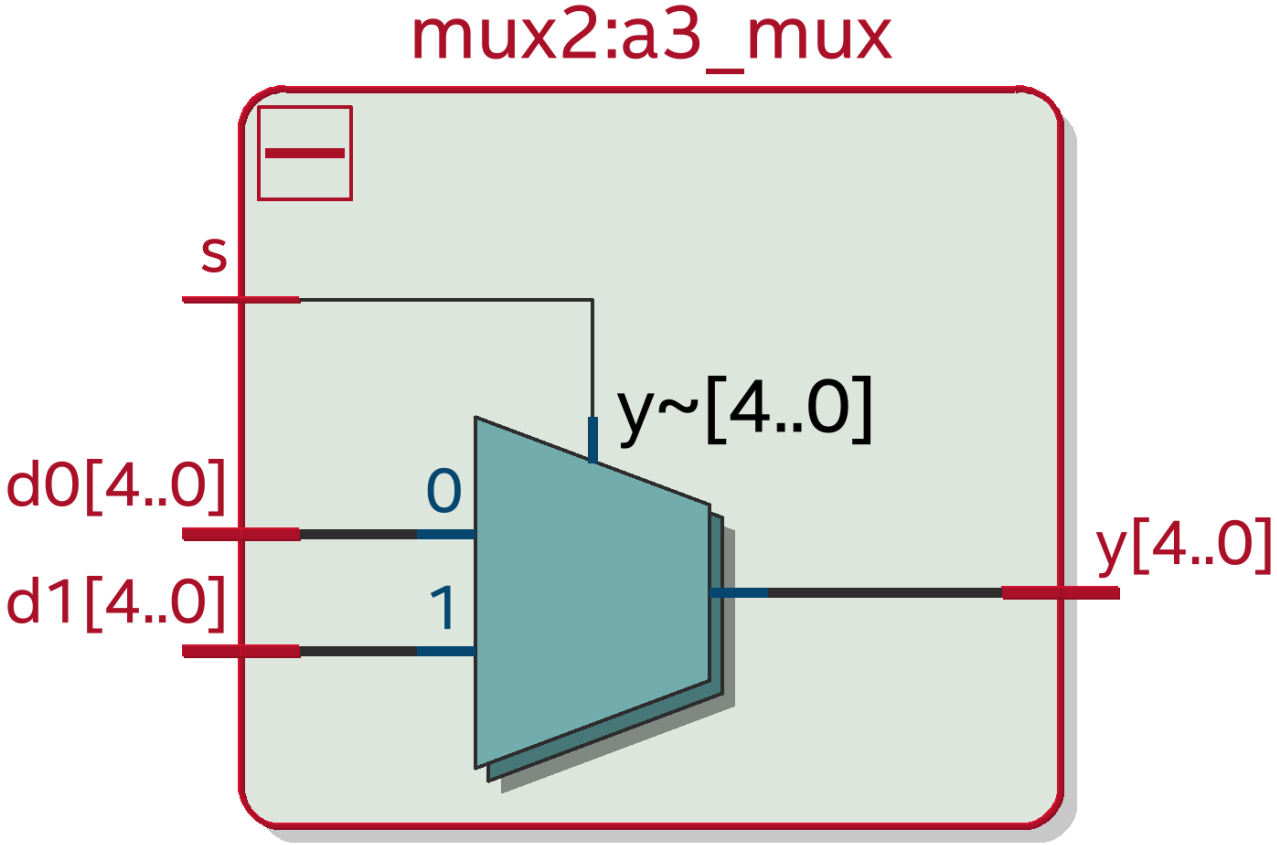
flopr



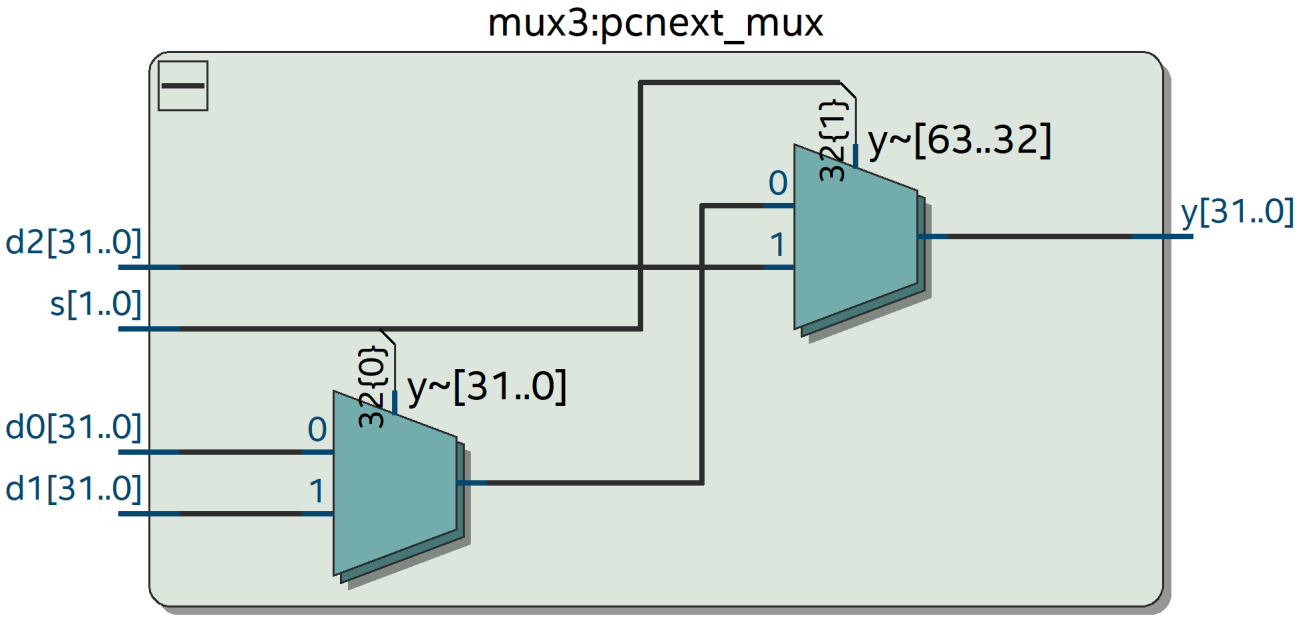
jumpext



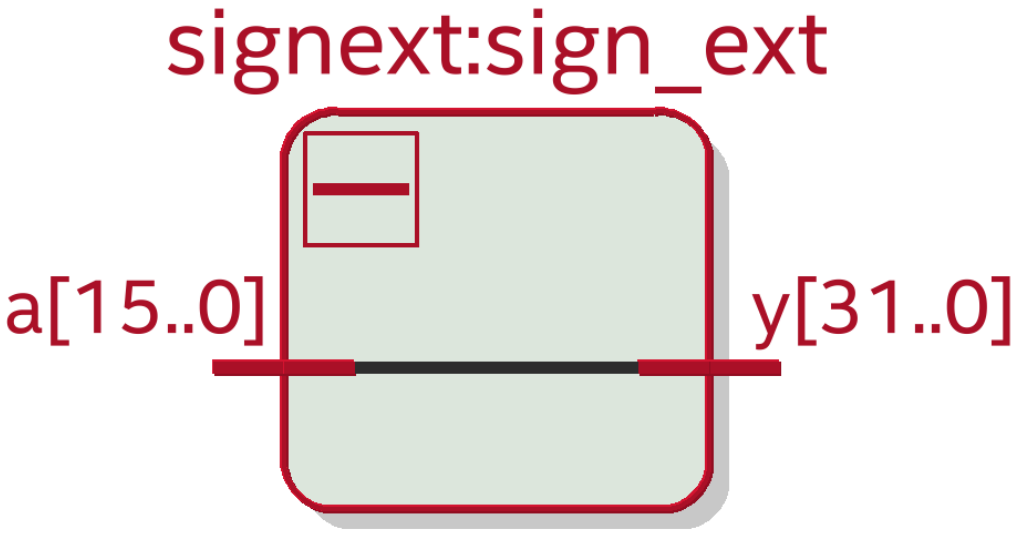
mux2


mux2:a3_mux
s
y~[4..0]
d0[4..0]
0
y[4..0]
d1[4..0]
1

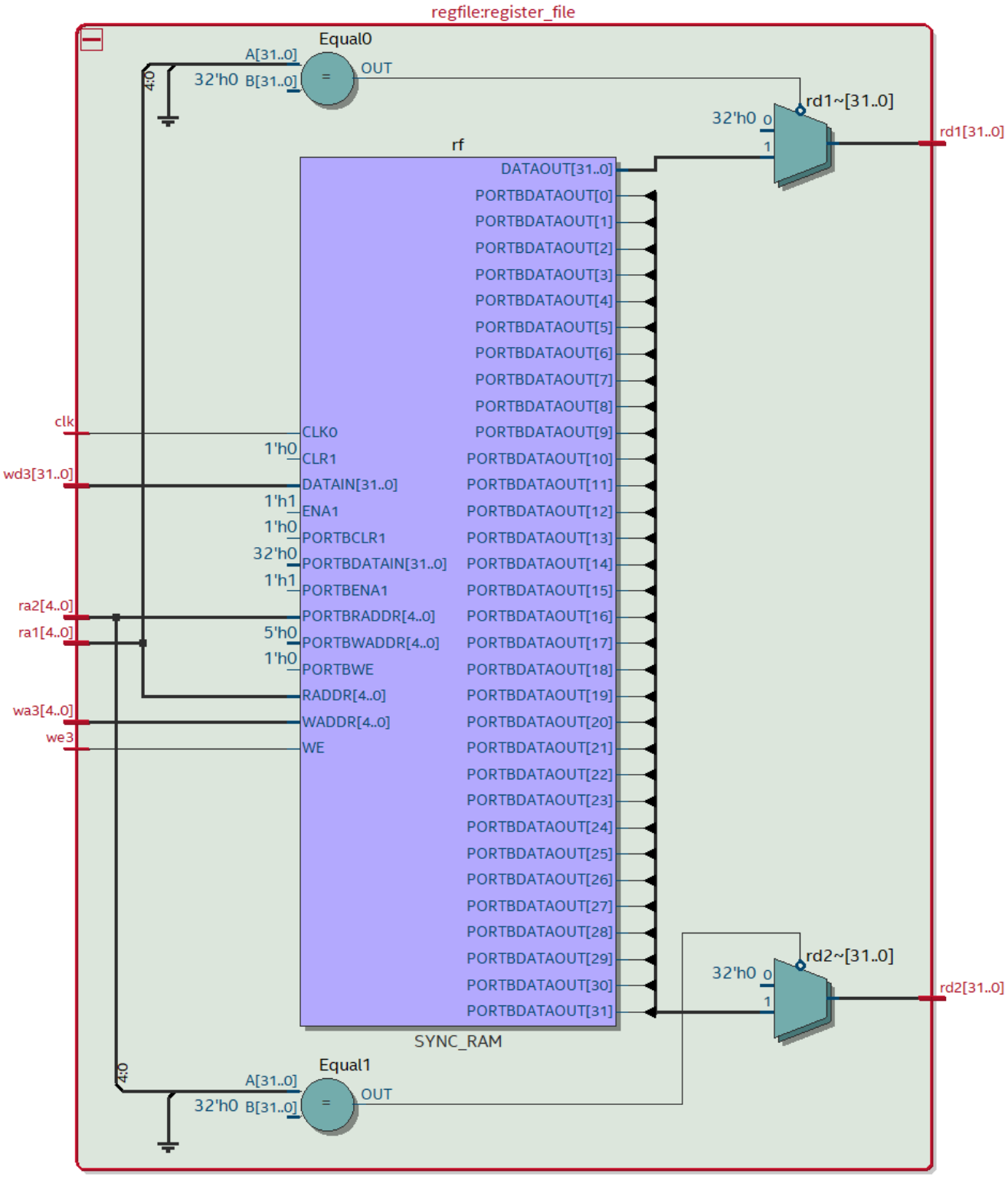
mux3


mux3:pcnext_mux
32{1}
y~[63..32]
0
d2[31..0]
1
y[31..0]
s[1..0]
32{0}
y~[31..0]
d0[31..0]
0
d1[31..0]
1

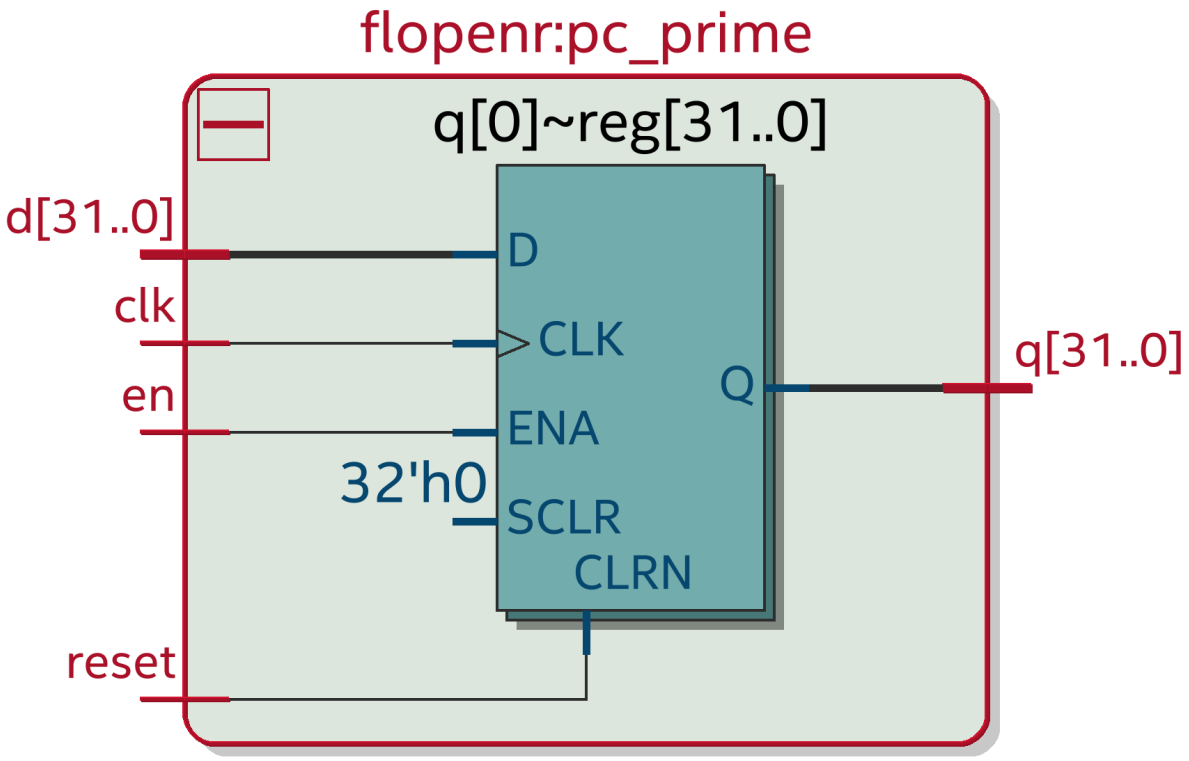
signext



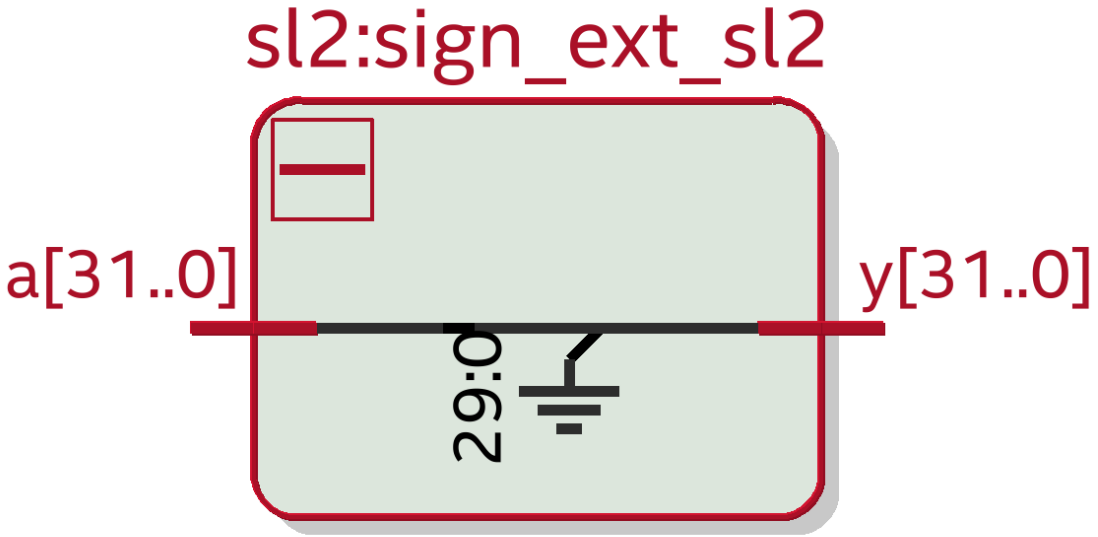
regfile


regfile:register_file
Equal0
A[31..0]
32'h0 B[31..0]
=
OUT
4:0
rd1~[31..0]
32'h0
0
1
rd1[31..0]
rf
DATAOUT[31..0]
PORTBDATAOUT[0]
PORTBDATAOUT[1]
PORTBDATAOUT[2]
PORTBDATAOUT[3]
PORTBDATAOUT[4]
PORTBDATAOUT[5]
PORTBDATAOUT[6]
PORTBDATAOUT[7]
PORTBDATAOUT[8]
PORTBDATAOUT[9]
PORTBDATAOUT[10]
PORTBDATAOUT[11]
PORTBDATAOUT[12]
PORTBDATAOUT[13]
PORTBDATAOUT[14]
PORTBDATAOUT[15]
PORTBDATAOUT[16]
PORTBDATAOUT[17]
PORTBDATAOUT[18]
PORTBDATAOUT[19]
PORTBDATAOUT[20]
PORTBDATAOUT[21]
PORTBDATAOUT[22]
PORTBDATAOUT[23]
PORTBDATAOUT[24]
PORTBDATAOUT[25]
PORTBDATAOUT[26]
PORTBDATAOUT[27]
PORTBDATAOUT[28]
PORTBDATAOUT[29]
PORTBDATAOUT[30]
PORTBDATAOUT[31]
clk
CLK0
1'h0
CLR1
wd3[31..0]
DATAIN[31..0]
1'h1
ENA1
1'h0
PORTBCLR1
32'h0
PORTBDATAIN[31..0]
1'h1
PORTBENA1
ra2[4..0]
PORTBRADDR[4..0]
ra1[4..0]
5'h0
PORTBWADDR[4..0]
1'h0
PORTBWE
RADDR[4..0]
wa3[4..0]
WADDR[4..0]
we3
WE
SYNC_RAM
rd2~[31..0]
32'h0
0
1
rd2[31..0]
4:0
Equal1
A[31..0]
32'h0 B[31..0]
=
OUT

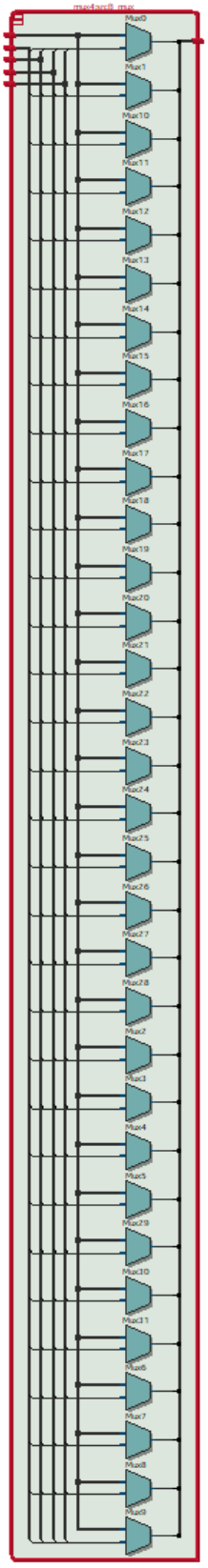
flopenr


flopenr:pc_prime
q[0]~reg[31..0]
d[31..0]
clk
en
32'h0
reset
D
CLK
ENA
SCLR
CLRN
Q
q[31..0]

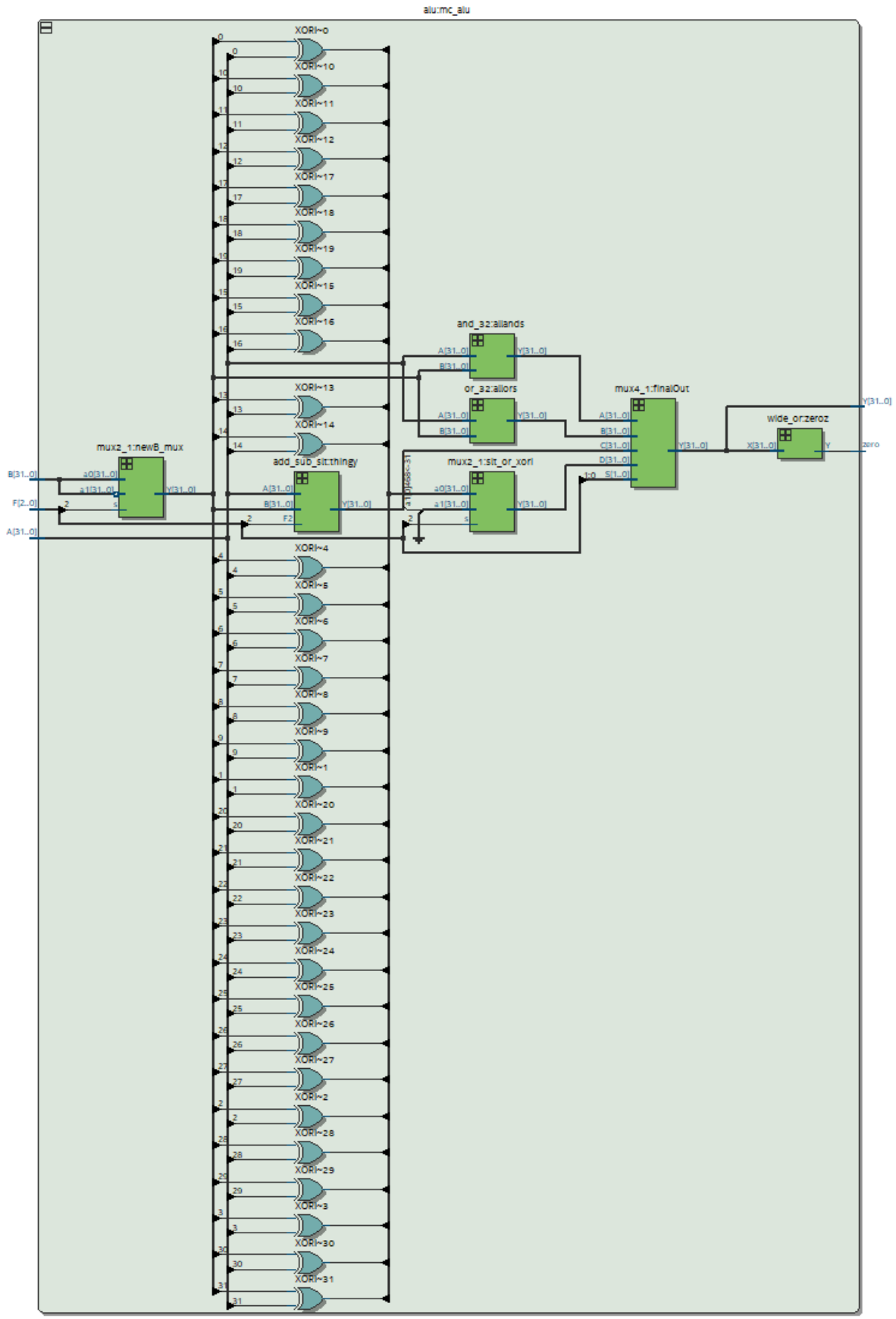
sl2


sl2:sign_ext_sl2
a[31..0]
y[31..0]
29:0

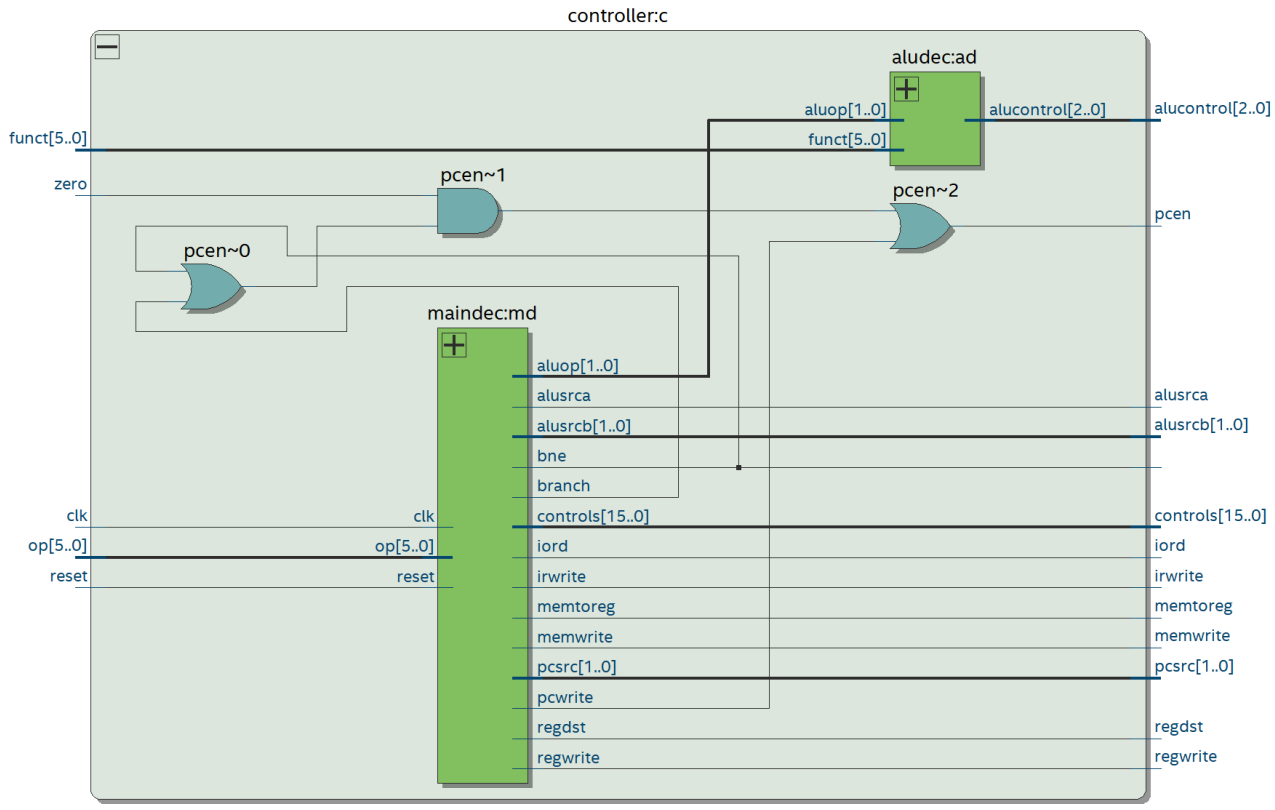
mux4



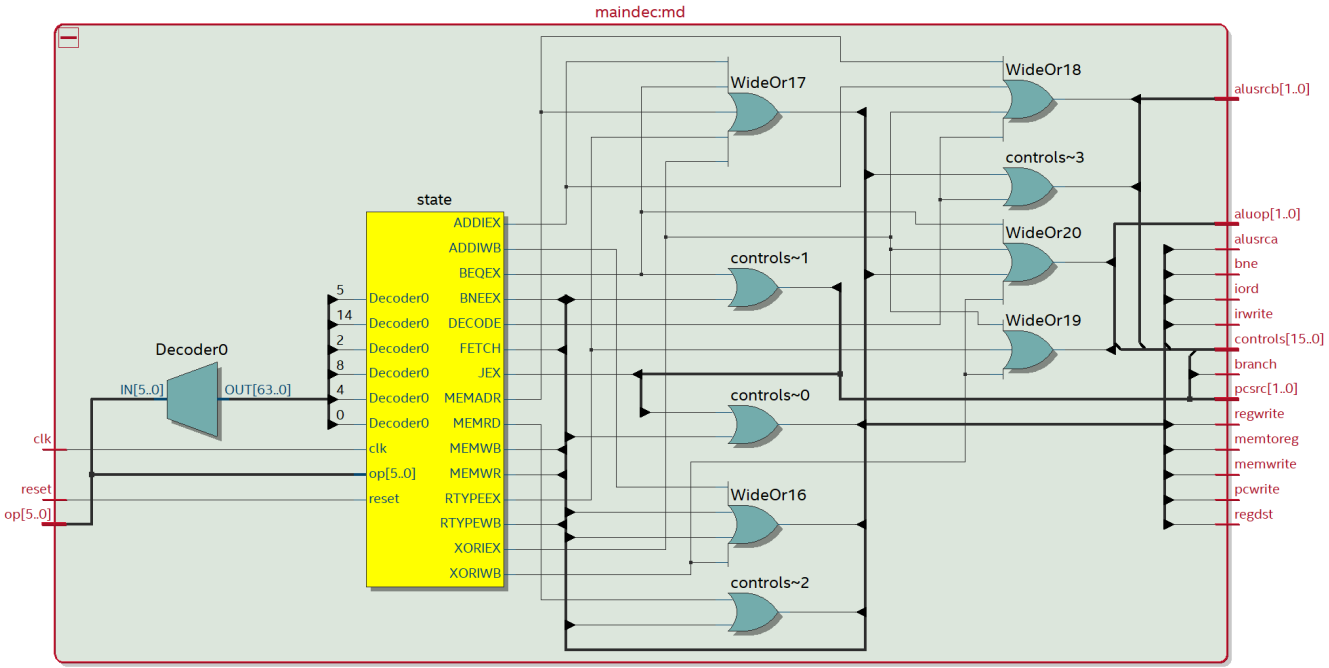
alu


alu:mc_alu
XORI~0
XORI~10
XORI~11
XORI~12
XORI~17
XORI~18
XORI~19
XORI~15
XORI~16
XORI~13
XORI~14
XORI~4
XORI~5
XORI~6
XORI~7
XORI~8
XORI~9
XORI~1
XORI~20
XORI~21
XORI~22
XORI~23
XORI~24
XORI~25
XORI~26
XORI~27
XORI~2
XORI~28
XORI~29
XORI~3
XORI~30
XORI~31
and_32:allands
or_32:allors
mux4_1:finalOut
wide_or:zeroz
mux2_1:newB_mux
add_sub_slt:thingy
mux2_1:slt_or_xori
B[31..0]
F[2..0]
A[31..0]
Y[31..0]
zero

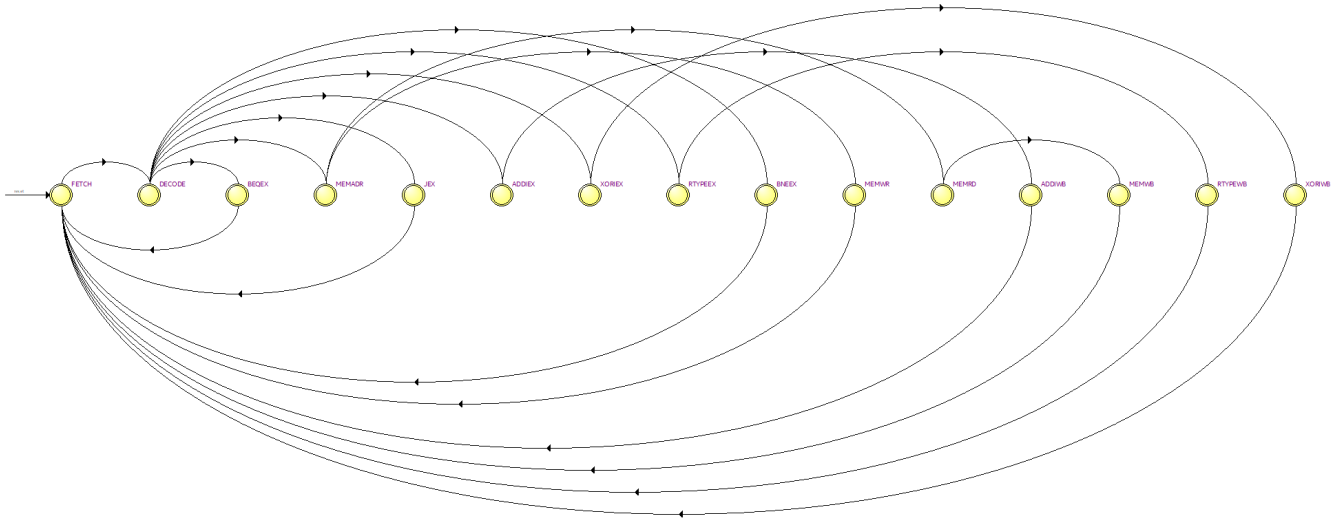
controller



maindec



state



aludec