Lab 6 - MIPS Single-Cycle CPU

Fabian Torres, Chris Escobar

Professor Wenjing Rao

ECE 469

Table of Contents

aludec
datapath
signext
adder
sl2
mux2
regfile
flopr
alu
mux2_1
apm
add_sub_slt
and_32
or_32
mux4_1
wide_or
dmem
imem

After Modification

top
mips
controller
maindec
aludec
datapath
signext
adder
sl2
mux2
regfile
flopr
alu
mux2_1
comparator_u32
add_sub_slt
and_32
or_32
mux4_1
wide_or
dmem
imem

Workload report

How many hours have you spent for this lab?
Which activity takes the most significant amount of time?

Lab Prompt

Part A MIPS Basic

Introduction

In this part of the project you will build a simplified MIPS single-cycle processor using SystemVerilog. You will combine your ALU from the previous project with the code for the rest of the processor taken from the textbook. Then you will load a test program and confirm that the system works. Next, you will implement two new instructions, and then write a new test program that confirms the new instructions work as well. By the end, you should thoroughly understand the internal operation of the MIPS single-cycle processor.

Please read and follow the instructions. In the past, many students have lost points for silly errors like not printing all the signals requested.

Before starting this project, you should be very familiar with the single-cycle implementation of the MIPS processor described in Ch 7.3 of your textbook. The single-cycle processor schematic from the text is repeated at the end of this assignment for your convenience. This version of the MIPS single-cycle processor can execute the following instructions:

{add, sub, and, or, slt, lw, sw, beq, addi, j}

Our model of the single-cycle MIPS processor divides the machine into two major units: the control and the datapath. Each unit is constructed from various functional blocks. For example, as shown in the figure on the last page, the datapath contains the 32-bit ALU that you designed in project 1, the register file, the sign extension logic, and five multiplexers to choose appropriate operands.

MIPS Single-Cycle Processor

The SystemVerilog single-cycle MIPS module is given in Ch 7.6 of the textbook.

Study the code until you are familiar with their contents. Look into the mips module, which instantiates two sub-modules, controller and datapath. Then take a look at the controller module and its two submodules: maindec and aludec. The maindec module produces all control signals except those for the ALU. The aludec module produces the control signal, alucontrol[2:0], for the ALU. Make sure you thoroughly understand the controller module. Correlate signal names in the SystemVerilog code with the wires on the schematic.

After you thoroughly understand the controller module, take a look at the datapath module. The datapath has quite a few submodules. Make sure you understand why each submodule is there and where each is located on the MIPS single-cycle processor schematic. You'll notice that the alu module is not defined. Use your ALU module from your previous project here. Be sure the module name matches the instance module name (alu), and make sure the inputs and outputs are in the same order as they are expected in the datapath module.

The highest-level module, top, includes the instruction and data memories as well as the processors. Each of the memories is a 64-word × 32-bit array. The instruction memory needs to contain some initial values representing the program. The test program is given in Figure 7.60 of the textbook. Study the program until you understand what it does. The machine language code for the program should be stored in a file called memfile.dat

Testing the single-cycle MIPS processor

In a complex system, if you don't know what to expect the answer should be, you are unlikely to get the right answer. Begin by predicting what should happen on each cycle when running the program. Complete the chart in Table 1 at the end of the assignment with your predictions. What address will the final sw instruction write to and what value will it write? Below is Fig 7.60 on memfile.dat 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
```

Simulate your processor with ModelSim. Be sure to add all of the .sv files, including the one containing your ALU. Add all of the signals from Table 1 to your waves window. (Note that many are not at the top level; you'll have to drill down into the appropriate part of the hierarchy to find them.)

Run the simulation. If all goes well, the testbench will print "Simulation succeeded." Look at the waveforms and check that they match your predictions in Table 1. If they don't, the problem is likely in your ALU or because you didn't properly add all of the files.
If you need to debug, you'll likely want to view more internal signals.

However, on the final waveform that you turn in, show ONLY the following signals:
clk, reset, pc, instr, aluout, writedata, memwrite, and readdata.

All the values need to be output in hexadecimal, in the above order, and must be readable to get full credit.

Note: You can use the MIPS simulator "MARS" to help generate hex code for assembly code using its "memory dump" functionality. To use MARS to run the assembly code, remember to activate the Settings -> Memory Configuration -> Compact, Data at Address 0
option

Part B: MIPS Plus

Implement the following 2 instructions:

{bne, andi}

bne has op = 000101
andi has op = 001100

Modifying the MIPS processor

You now need to modify the MIPS single-cycle processor by adding the andi and bne instructions. First, modify the MIPS processor schematic (at the last page) to show what changes are necessary. You can draw your changes directly onto the schematic. Then modify the main decoder and ALU decoder as required. Show your changes in the tables at the end too. Finally, modify the SystemVerilog code as needed to include your modifications.

Testing your modified MIPS single-cycle processor

```
# test_b.asm

addi  $8, $0,  -1
andi $8, $8, 7
addi $9, $0, 40
loop:
andi $10, $8, 3
sw $10, 0($9)
addi $8, $8, -1
addi $9, $9, 4
bne $8, $0, loop
lw $10, 44($0)
```

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. The above provides an example test_b.asm for the two additional instructions bne and andi.
Convert the program to machine language and put it in a file named memfile_b.dat. Modify imem 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.

Part C: MIPS Plus Plus

In addition to Part B, add the following instructions:

{sltu, bgt}

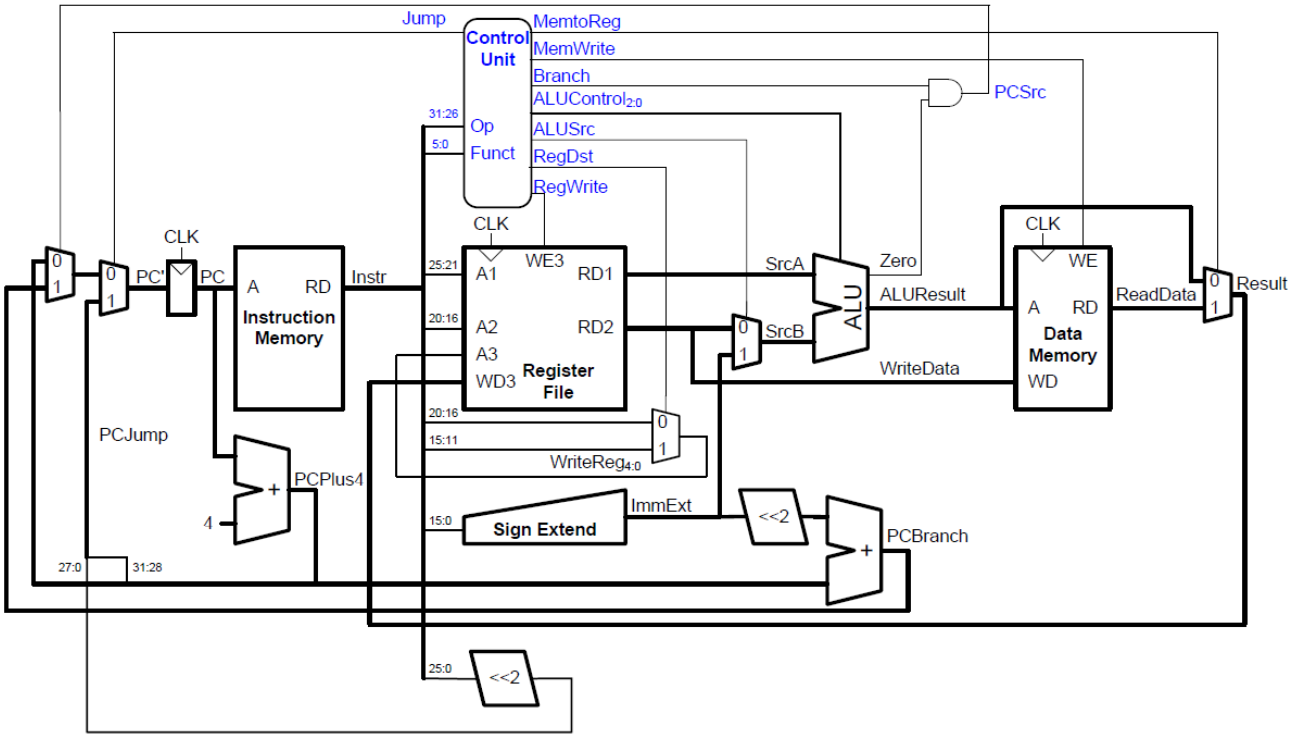
sltu should work like slt except assuming numbers to be unsigned. For example, if $8 = 0xFFFFFFFF, $9 = 0x00000000, then
slt $10, $8, $9 will result in $10 = 1 ($8 treated as negative number).
sltu $10, $8, $9 will result in $10 = 0 ($8 treated as a large positive number).

sltu has op = 000000, func = 101011

bgt does not exist in standard MIPS instruction, thus you can choose any unused op and format for its machine code. It should function as follows:
“bgt $8, $7, loop” will go to loop when $8>$7. Otherwise ($8<$7 or $8=$7) go to the next instruction.

Use test_c.asm to verify your design similar to that of Part B above.

```
# test_c.asm
addi  $8, $0,  -1
andi $8, $8, 3
addi $9, $0, 40
addi $7, $0, -4
loop:
sltu $10, $0, $8
slt $11, $0, $8
sub $10, $10, $7
sub $11, $11, $7
sw $10, 0($9)
sw $11, 4($9)
addi $8, $8, -3
addi $9, $9, 8
bgt $8, $7, loop
sw $8, 36($0)
```



Single-cycle MIPS processor

Extended functionality. Main Decoder:

Inst.	$Op_{5:0}$	RegWrite	RegDst	AluSrc	Branch	bne	bgt	MemWrite	MemtoReg	Jump	$ALUOp_{1:0}$
R-Type	000000	1	1	0	0	0	0	0	0	0	10
lw	100011	1	0	1	0	0	0	0	1	0	00
sw	101011	0	X	1	0	0	0	1	X	0	00
beq	000100	0	X	0	1	0	0	0	X	0	01
addi	001000	1	0	1	0	0	0	0	0	0	00
j	000010	0	X	X	X	0	0	0	X	1	XX
andi	001100	1	0	1	0	0	0	0	0	0	11
bne	000101	0	X	0	1	1	0	0	X	0	01
sltu	000000	1	1	0	0	0	0	0	0	0	10
bgt	010101	0	0	0	0	0	1	0	0	0	01

Extended functionality. ALU Decoder:

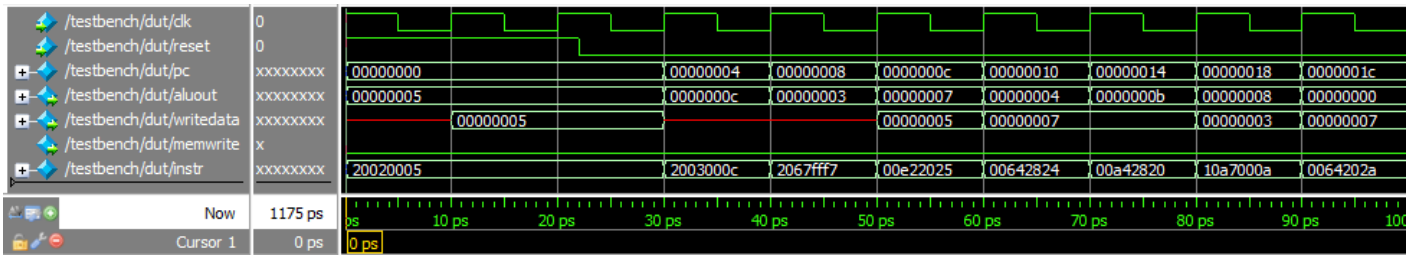
$ALUOp_{1:0}$	Meaning
00	Add
01	Subtract
10	Look at *funct* field (R-Type)
11	Andi

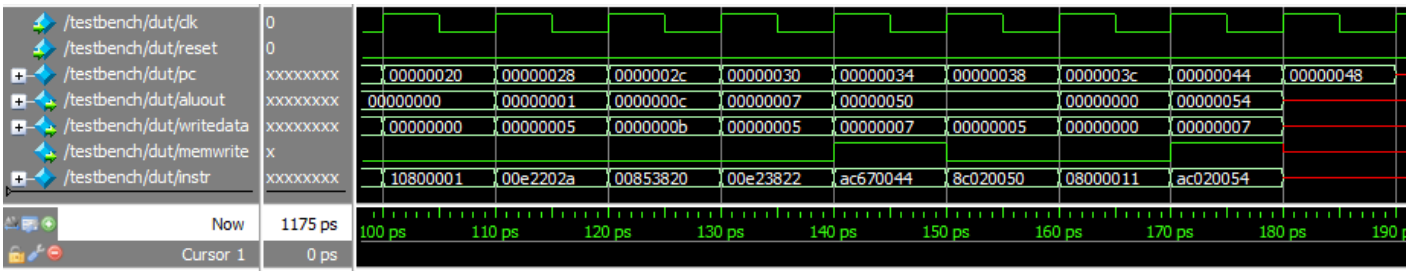
Part A

Table 1

cycle	reset	pc	instr	branch	srca	srcb	aluout	zero	pcsrc	write data	mem write	read data
1	1	00	addi $2, $0, $5	0	0	5	5	0	0	0	0	X
2	0	04	addi $3, $0, 12	0	0	c	0	0	0	0	0	X
3	0	0C	addi $7, $3, -9	0	c	-9	3	0	0	0	0	X
4	0	10	or $4, $7, $2	0	3	5	7	0	0	0	0	X
5	0	14	and $5, $3, $4	0	c	7	4	0	0	0	0	X
6	0	18	add $5, $3, $4	0	4	7	11	0	0	0	0	X
7	0	1C	beq $5, $7, end	1	11	3	X	0	0	0	0	X
8	0	20	slt $4, $3, $4	0	C	7	0	1	0	0	0	X
9	0	24	beq $4, $0, around	1	0	0	0	1	1	0	0	X
10	0	28	slt $4, $7, $2	0	3	5	1	0	0	0	0	X
11	0	2C	add $7, $4, $5	0	1	11	C	0	0	0	0	X
12	0	30	sub $7, $7, $2	0	C	5	7	0	0	0	0	X
13	0	34	sw $7, 68($3)	0	12	68	80	0	0	7	1	X
14	0	38	lw $2, 80($0)	0	0	80	80	0	0	0	0	7
15	0	3C	j end	0	X	X	X	X	X	X	0	X
16	0	44	sw $2, 84($0)	0	0	84	84	0	0	7	1	X

Simulation Waveforms of unmodified processor





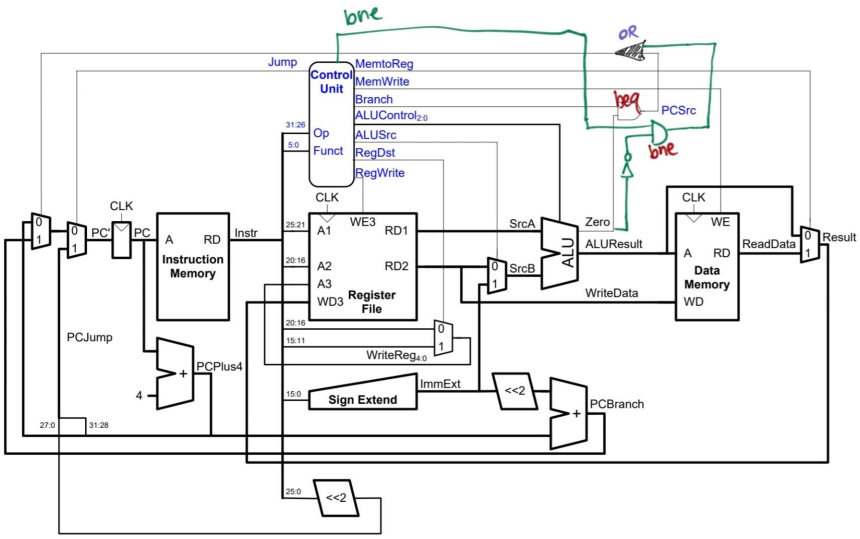

```
VSIM 39> run
# Simulation succeeded
# ** Note: $stop    : C:/intelFPGA_lite/MIPS Single-Cycle CPU/mipstest.sv(28)
#    Time: 175 ps  Iteration: 1  Instance: /testbench
# Break in Module testbench at C:/intelFPGA_lite/MIPS Single-Cycle CPU/mipstest.sv line 28
```

Yes, the correct value of 7 is written to address 84 (0x54).

Part B/C

List of the new instructions with encodings and datapath schematics

bne



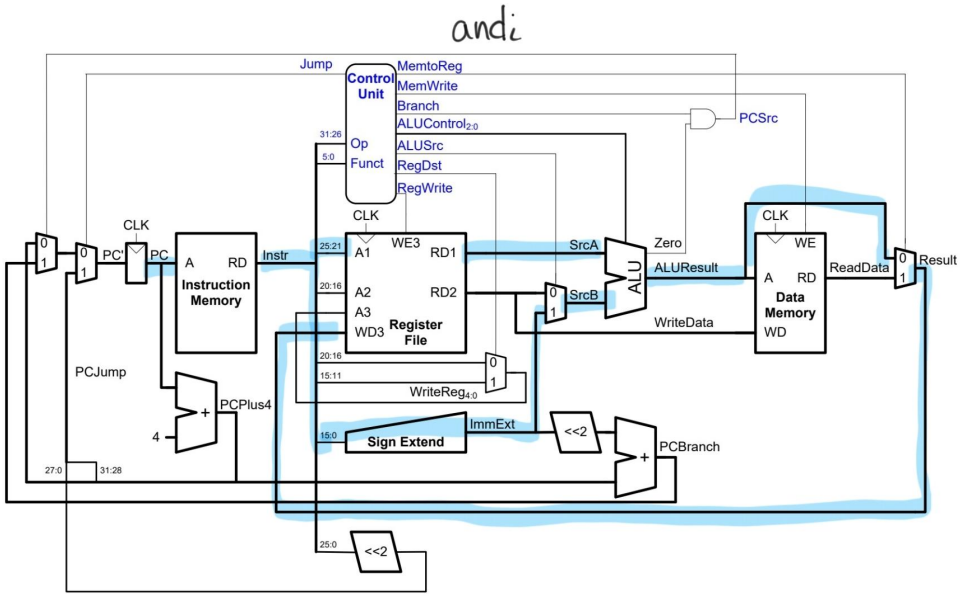
Single-cycle MIPS processor

controls: 11'b00001000001

aluop: 2’b01

alucontrol: 3'b110

andi



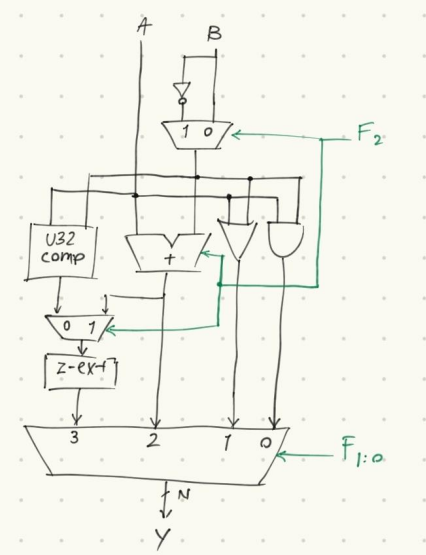
controls: 11'b10100000011

aluop: 2'b11

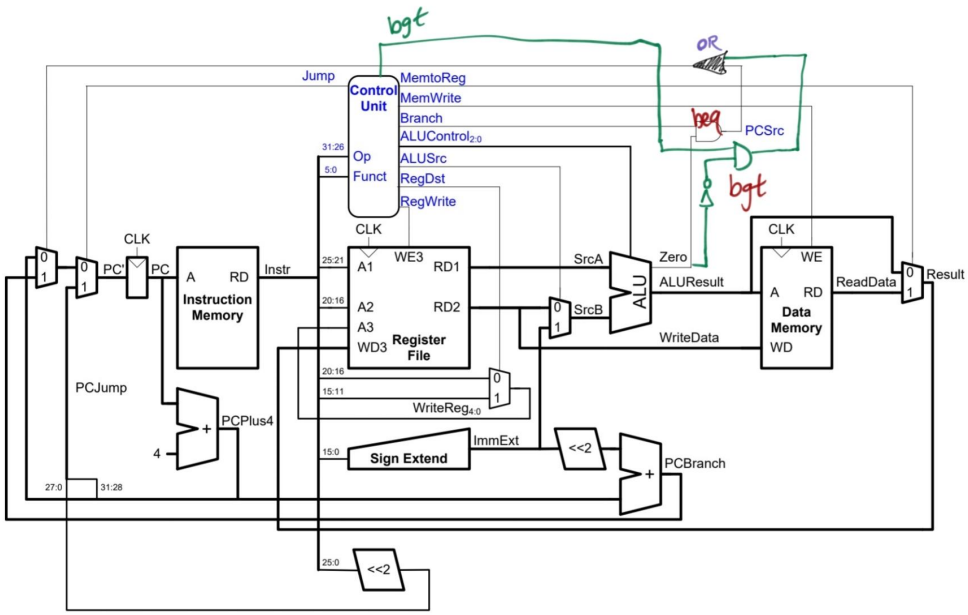
alucontrol: 3'b000

sltu

controls: 11'b11000000010
funct: 6'b101011
alucontrol: 3'b011



bgt



Single-cycle MIPS processor

controls: 11'b00000100001;
aluop: 2’b01
alucontrol: 3'b110

SystemVerilog code with changes highlighted in the code

bne

```
module controller(
        input logic [5:0] op, funct,
        input logic zero, neg_pos,
        output logic memtoreg, memwrite, pcsrc, alusrc, regdst, regwrite, jump,
        output logic [2:0] alucontrol);
    logic [1:0] aluop;
    logic branch, beq_check, bne_check, bne, bgt;
    maindec md(op, memtoreg, memwrite, branch, bne, bgt, alusrc, regdst, regwrite,
jump, aluop);
    aludec ad(funct, aluop, alucontrol);
    assign beq_check = branch & zero;
    assign bne_check = bne & ~zero;
    assign bgt_check = bgt & neg_pos;
    assign pcsrc = beq_check | bne_check | bgt_check;
endmodule
module maindec(
        input logic [5:0] op,
        output logic memtoreg, memwrite, branch, bne, bgt, alusrc, regdst, regwrite,
jump,
        output logic [1:0] aluop);
    logic [10:0] controls;
    assign {regwrite, regdst, alusrc, branch, bne, bgt, memwrite, memtoreg, jump,
aluop} = controls;
    always_comb
        case(op)
            6'b000000: controls = 11'b11000000010;  // R-Type
            6'b100011: controls = 11'b10100001000; // LW
            6'b101011: controls = 11'b00100010000; // SW
            6'b000100: controls = 11'b00010000001; // BEQ
            6'b001000: controls = 11'b10100000000; // ADDI
            6'b000010: controls = 11'b00000000100; // J
            6'b000101: controls = 11'b00001000001; // BNE
            6'b001100: controls = 11'b10100000011; // ANDI
            6'b010101: controls = 11'b00000100001; // BGT
            default:   controls = 11'bxxxxxxxxxxx; // ???
        endcase
endmodule
```

andi

```
module maindec(
input logic [5:0] op,
output logic memtoreg, memwrite, branch, bne, bgt, alusrc, regdst, regwrite, jump,
        output logic [1:0] aluop);
        logic [10:0] controls;
        assign {regwrite, regdst, alusrc, branch, bne, bgt, memwrite, memtoreg, jump,
aluop} = controls;
        always_comb
           case(op)
               6'b000000: controls = 11'b11000000010;  // R-Type
               6'b100011: controls = 11'b10100001000; // LW
               6'b101011: controls = 11'b00100010000; // SW
               6'b000100: controls = 11'b00010000001; // BEQ
               6'b001000: controls = 11'b10100000000; // ADDI
               6'b000010: controls = 11'b00000000100; // J
               6'b000101: controls = 11'b00001000001; // BNE
               6'b001100: controls = 11'b10100000011; // ANDI
               6'b010101: controls = 11'b00000100001; // BGT
               default:   controls = 11'bxxxxxxxxxxx; // ???
         endcase
endmodule
```

```
module aludec(
        input logic [5:0] funct,
        input logic [1:0] aluop,
        output logic [2:0] alucontrol);
    always_comb
        case(aluop)
            2'b00: alucontrol = 3'b010; // add
            2'b01: alucontrol = 3'b110; // sub
            2'b11: alucontrol = 3'b000; // and
        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
            6'b101011: alucontrol = 3'b011; // SLTU
            default:   alucontrol = 3'bxxx; // ???
        endcase
    endcase
endmodule
```

sltu

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

    always_comb
        case(aluop)
            2'b00: alucontrol = 3'b010; // add
            2'b01: alucontrol = 3'b110; // sub
            2'b11: alucontrol = 3'b000; // andi
        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
            6'b101011: alucontrol = 3'b011; // SLTU
            default:   alucontrol = 3'bxxx; // ???
        endcase
    endcase
endmodule
```

```
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, approximate, X11_output, sltu;
    wire [N-1:0] newB, AB, A_or_B;
    logic lt;
    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]};
    comparator_u32 unsigned_comp(A, B, lt);
    assign sltu = {31'h00000000, lt};
    mux2_1 slt_or_sltu(sltu, X_ext, F[2], X11_output);
    mux4_1 finalOut(AB, A_or_B, X, X11_output, F[1:0], Y);
    wide_or zeroz(Y, zero);
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
```

bgt

```
module controller(
        input logic [5:0] op, funct,
        input logic zero, neg_pos,
        output logic memtoreg, memwrite, pcsrc, alusrc, regdst, regwrite, jump,
        output logic [2:0] alucontrol);

    logic [1:0] aluop;
    logic branch, beq_check, bne_check, bne, bgt;

    maindec md(op, memtoreg, memwrite, branch, bne, bgt, alusrc, regdst, regwrite,
jump, aluop);
    aludec ad(funct, aluop, alucontrol);
    assign beq_check = branch & zero;
    assign bne_check = bne & ~zero;
    assign bgt_check = bgt & neg_pos;
    assign pcsrc = beq_check | bne_check | bgt_check;
endmodule
```

```
module maindec(
        input logic [5:0] op,
        output logic memtoreg, memwrite, branch, bne, bgt, alusrc, regdst, regwrite,
jump,
        output logic [1:0] aluop);
    logic [10:0] controls;
    assign {regwrite, regdst, alusrc, branch, bne, bgt, memwrite, memtoreg, jump,
aluop} = controls;

    always_comb
        case(op)
            6'b000000: controls = 11'b11000000010;  // R-Type
            6'b100011: controls = 11'b10100001000; // LW
            6'b101011: controls = 11'b00100010000; // SW
            6'b000100: controls = 11'b00010000001; // BEQ
            6'b001000: controls = 11'b10100000000; // ADDI
            6'b000010: controls = 11'b00000000100; // J
            6'b000101: controls = 11'b00001000001; // BNE
            6'b001100: controls = 11'b10100000011; // ANDI
            6'b010101: controls = 11'b00000100001; // BGT
            default:   controls = 11'bxxxxxxxxxxx; // ???
        endcase
endmodule
```

Contents of memfile_b.dat

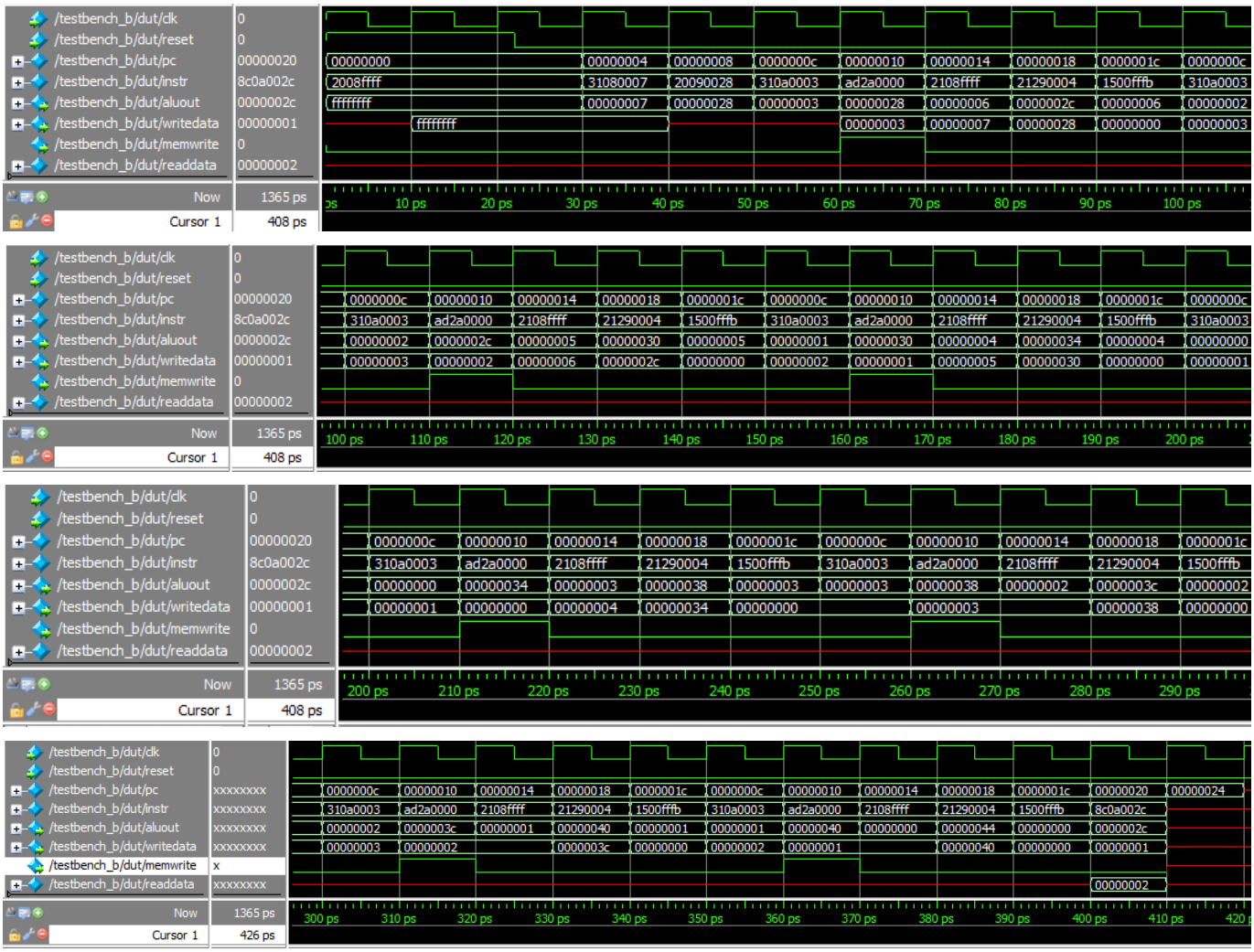
2008ffff
31080007
20090028
310a0003
ad2a0000
2108ffff
21290004
1500fffb
8c0a002c

Contents of memfile_c.dat

2008ffff
31080003
20090028
2007fffc
0008502b
0008582a
01475022
01675822
ad2a0000
ad2b0004
2108fffd
21290008
00e8082a
1420fff6
ac080024

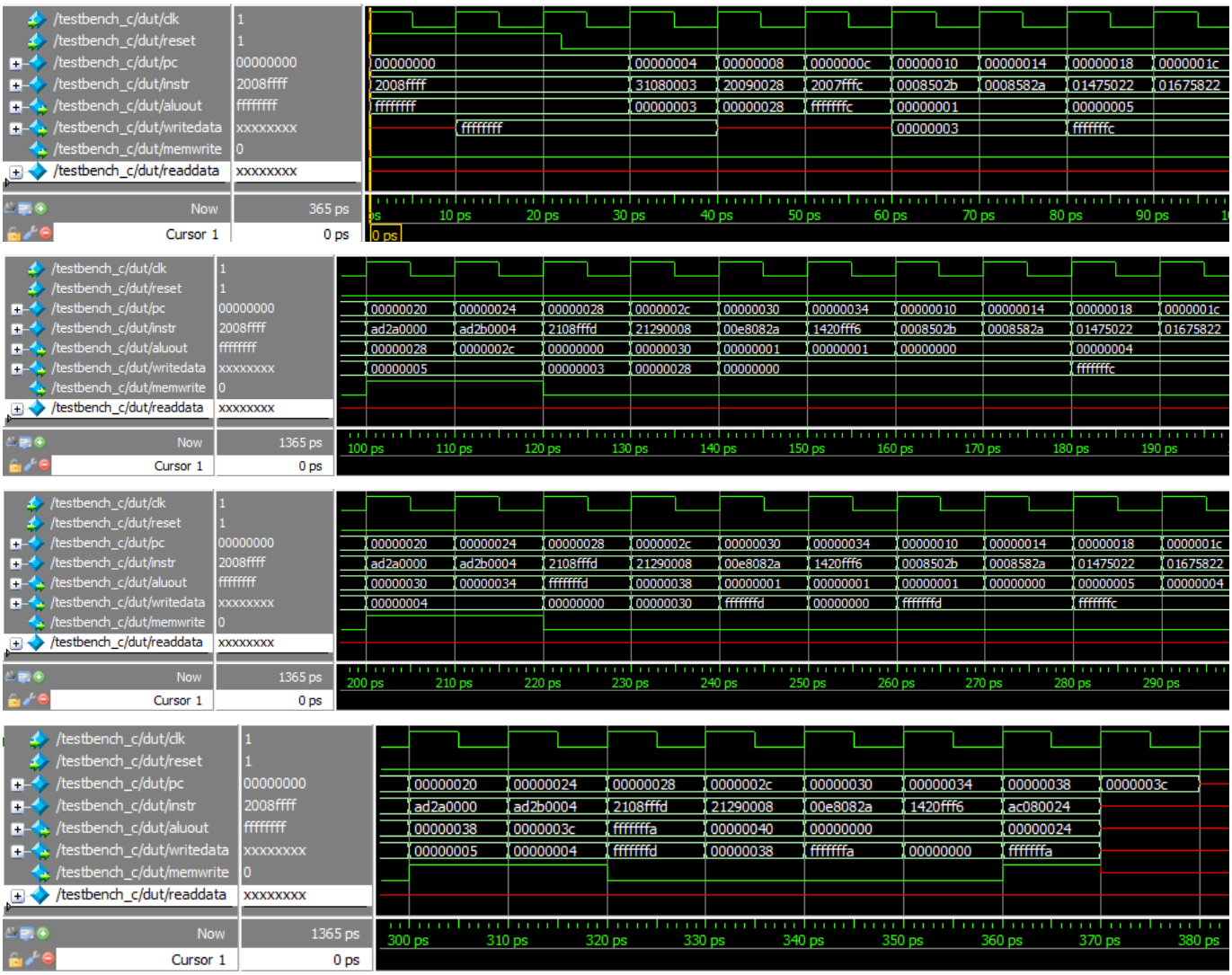
Simulation waveforms and testbenches test_b, test_c

test_b



```
module testbench_b();
    logic clk, reset;
    logic [31:0] writedata, aluout;
    logic memwrite;

    // instantiate device to be tested
    top dut(clk, reset, writedata, aluout, memwrite);

    // initialize test
    initial
        begin
            reset <= 1; #22; reset <= 0;
        end

    // generate clock to sequence tests
    always
        begin
            clk <= 1; # 5; clk <= 0; # 5;
        end
    //
    always@(negedge clk) begin
        if(memwrite) begin
            if(aluout === 64 && writedata === 1) begin
                $display("Simulation succeeded");
                $stop;
            end
            else if(aluout !== 40 || aluout !== 44 || aluout !== 48 || aluout !== 52
|| aluout !== 56 || aluout !== 60) begin
                $display("Simulation failed");
                $stop;
            end
        end
    end
endmodule
```

test_c


/testbench_c/dut/clk
/testbench_c/dut/reset
/testbench_c/dut/pc
/testbench_c/dut/instr
/testbench_c/dut/aluout
/testbench_c/dut/writedata
/testbench_c/dut/memwrite
/testbench_c/dut/readdata
Now
Cursor 1
00000000 00000004 00000008 0000000c 00000010 00000014 00000018 0000001c
2008ffff 31080003 20090028 2007fffc 0008502b 0008582a 01475022 01675822
ffffffff 00000003 00000028 fffffffc 00000001 00000005
ffffffff 00000003 fffffffc
00000020 00000024 00000028 0000002c 00000030 00000034 00000010 00000014 00000018 0000001c
ad2a0000 ad2b0004 2108fffd 21290008 00e8082a 1420fff6 0008502b 0008582a 01475022 01675822
00000028 0000002c 00000000 00000030 00000001 00000001 00000000 00000004
00000005 00000003 00000028 00000000 fffffffc
00000020 00000024 00000028 0000002c 00000030 00000034 00000010 00000014 00000018 0000001c
ad2a0000 ad2b0004 2108fffd 21290008 00e8082a 1420fff6 0008502b 0008582a 01475022 01675822
00000030 00000034 fffffffd 00000038 00000001 00000001 00000001 00000000 00000005 00000004
00000004 00000000 00000030 fffffffd 00000000 fffffffd fffffffc
00000020 00000024 00000028 0000002c 00000030 00000034 00000038 0000003c
ad2a0000 ad2b0004 2108fffd 21290008 00e8082a 1420fff6 ac080024
00000038 0000003c fffffffa 00000040 00000000 00000024
00000005 00000004 fffffffd 00000038 fffffffa 00000000 fffffffa

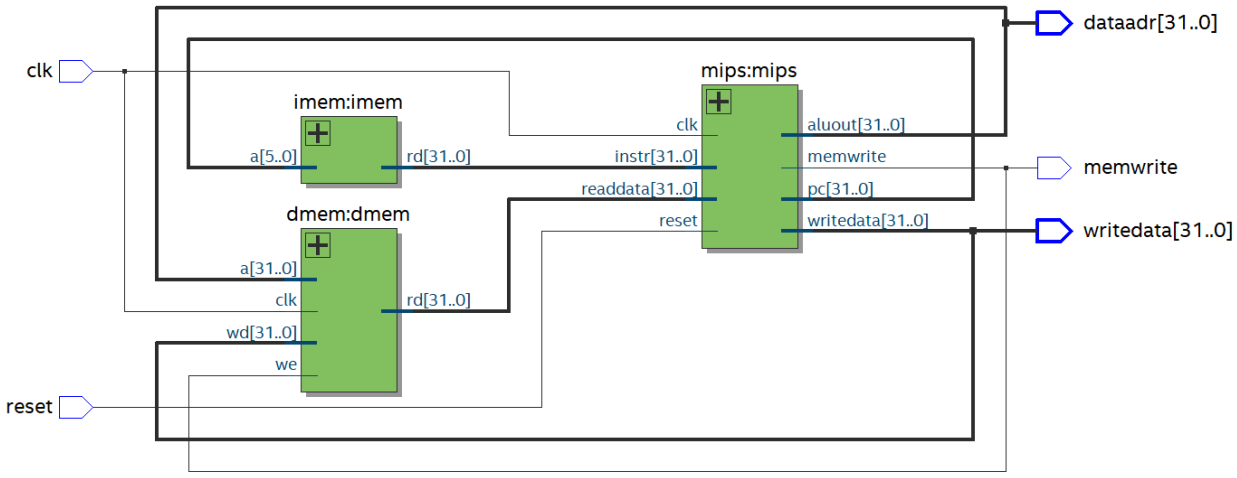
```
module testbench_c();
    logic clk, reset;
    logic [31:0] writedata, aluout;
    logic memwrite;

    // instantiate device to be tested
    top dut(clk, reset, writedata, aluout, memwrite);

    // initialize test
    initial
        begin
            reset <= 1; #22; reset <= 0;
        end

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

    always@(negedge clk) begin
        if(memwrite) begin
            if(aluout === 36 & writedata === -6) begin
                $display("Simulation succeeded");
                $stop;
            end
            else if (aluout !== 40 || aluout !== 44 || aluout !== 48 || aluout !== 52
|| aluout !== 56 || aluout !== 60) begin
                $display("Simulation failed");
                $stop;
            end
        end
    end
endmodule
```

Overall

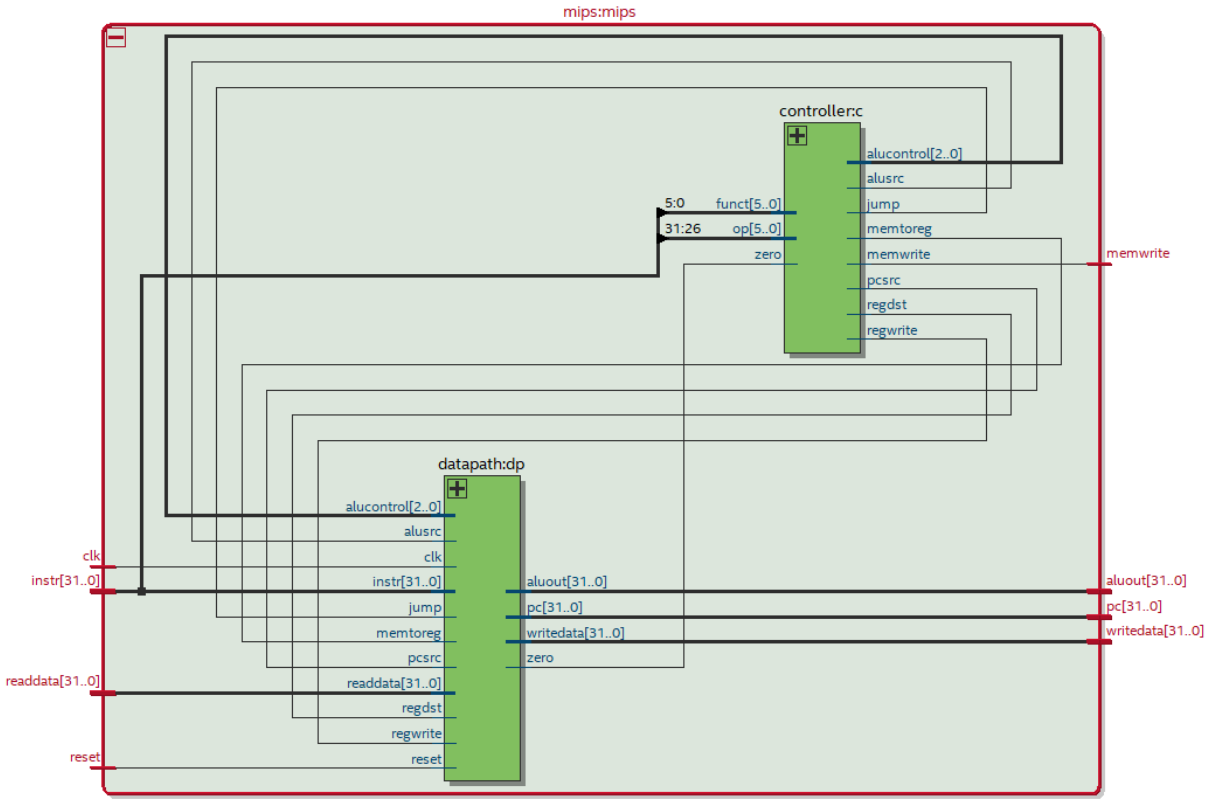
Quartus RTL Schematics

Before Modification

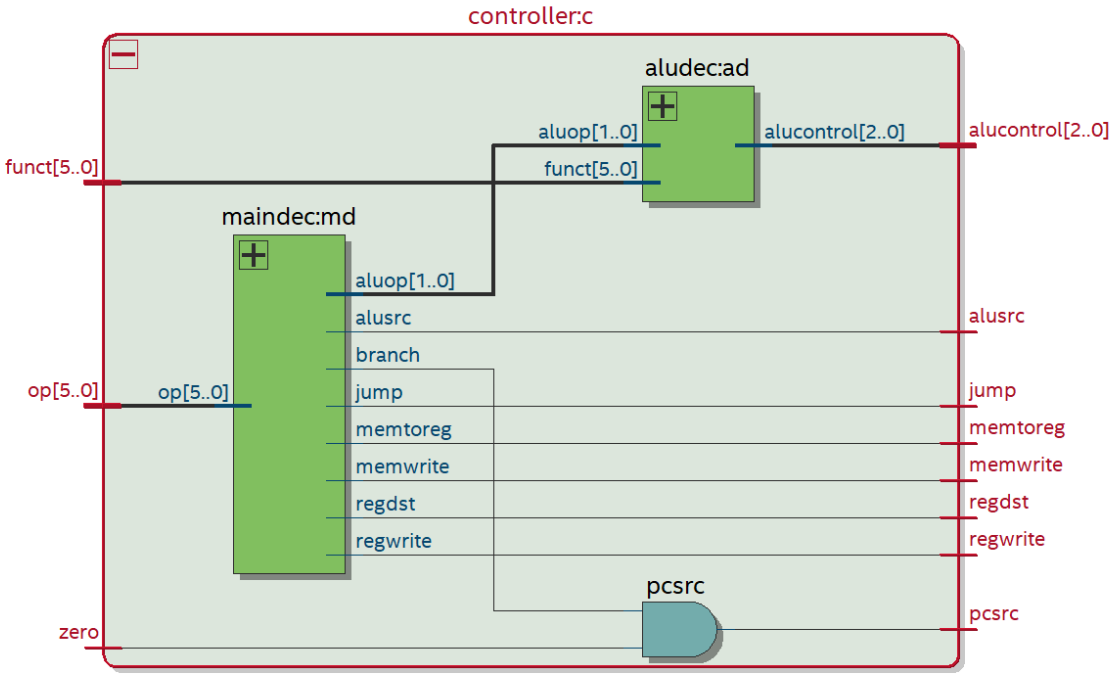
top



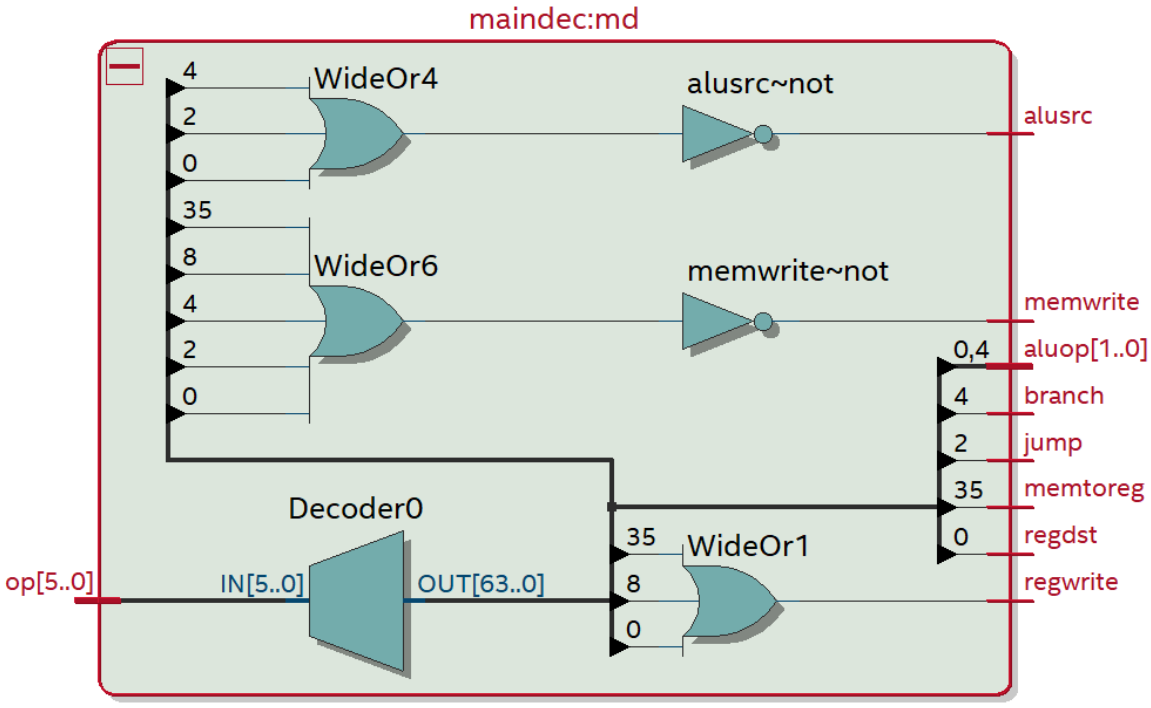
mips



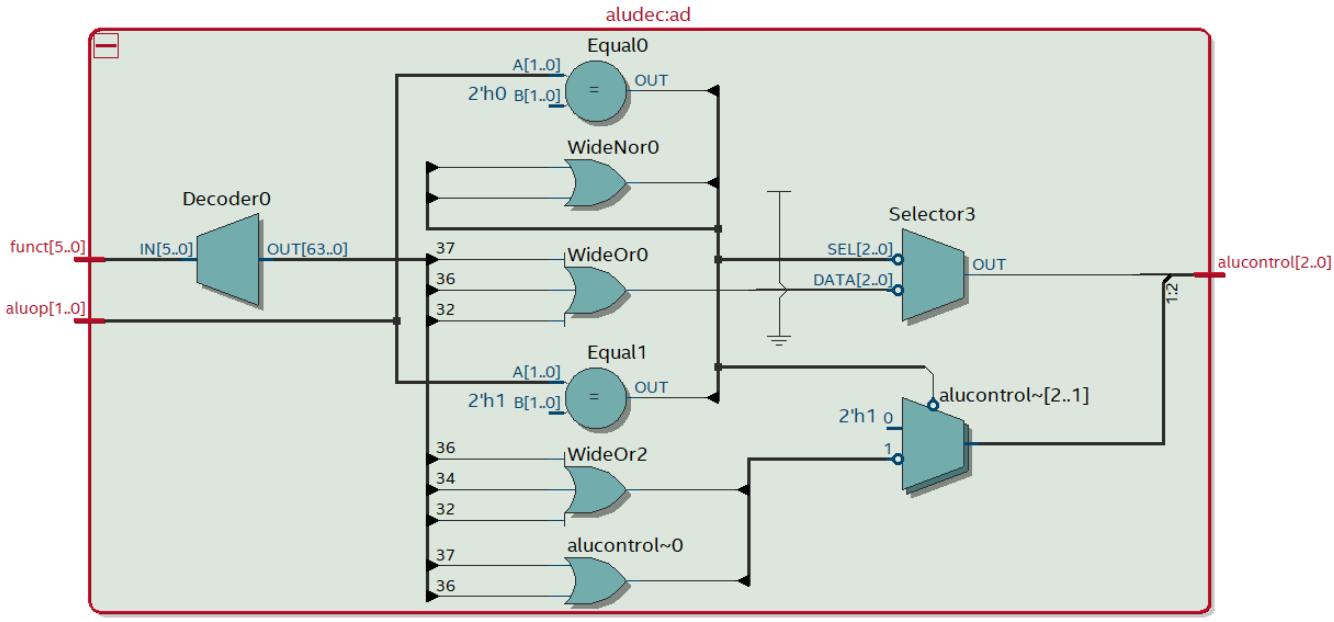
controller



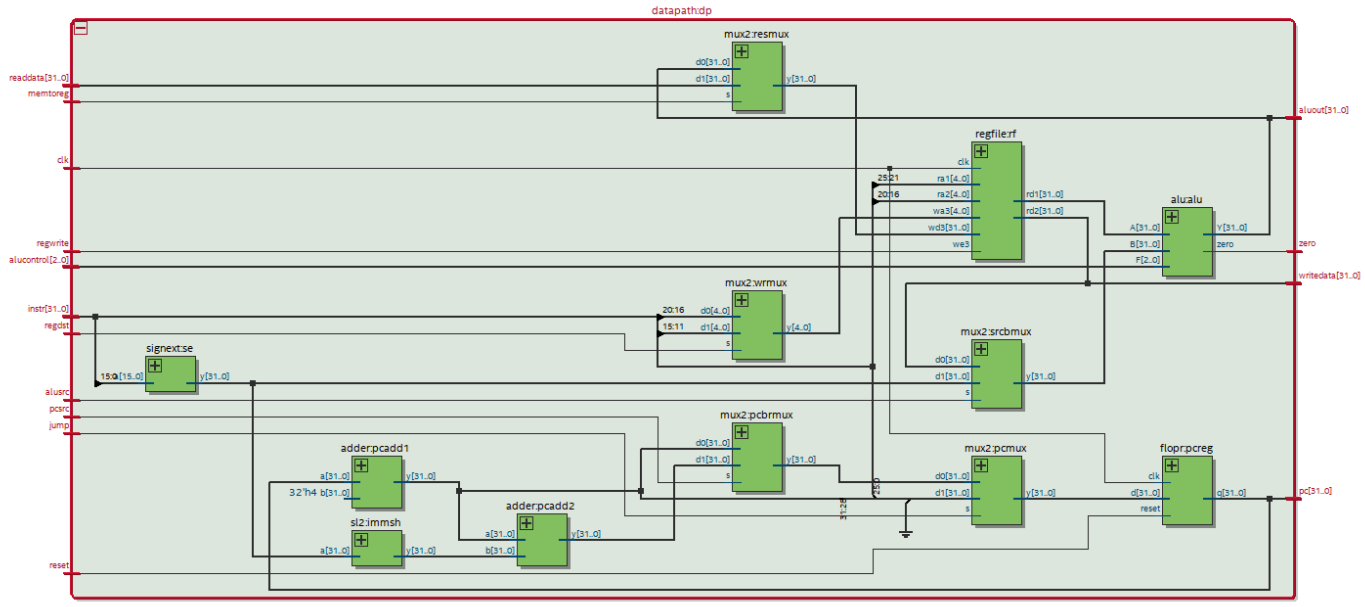
maindec



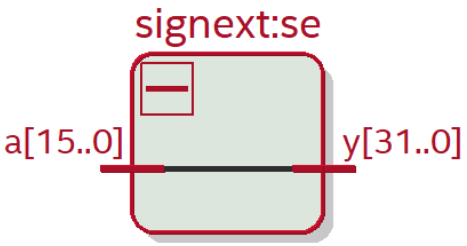
aludec



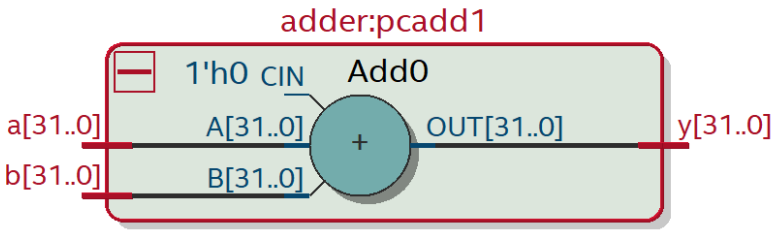
datapath



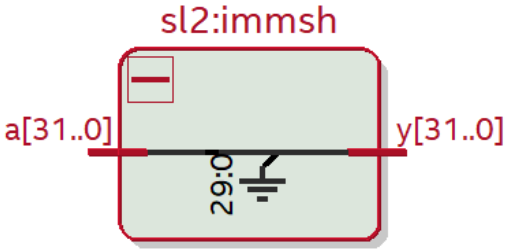
signext



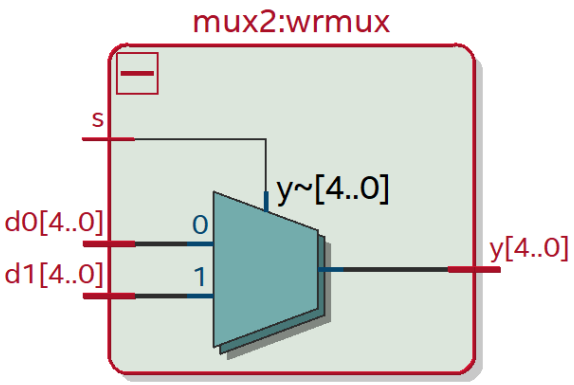
adder



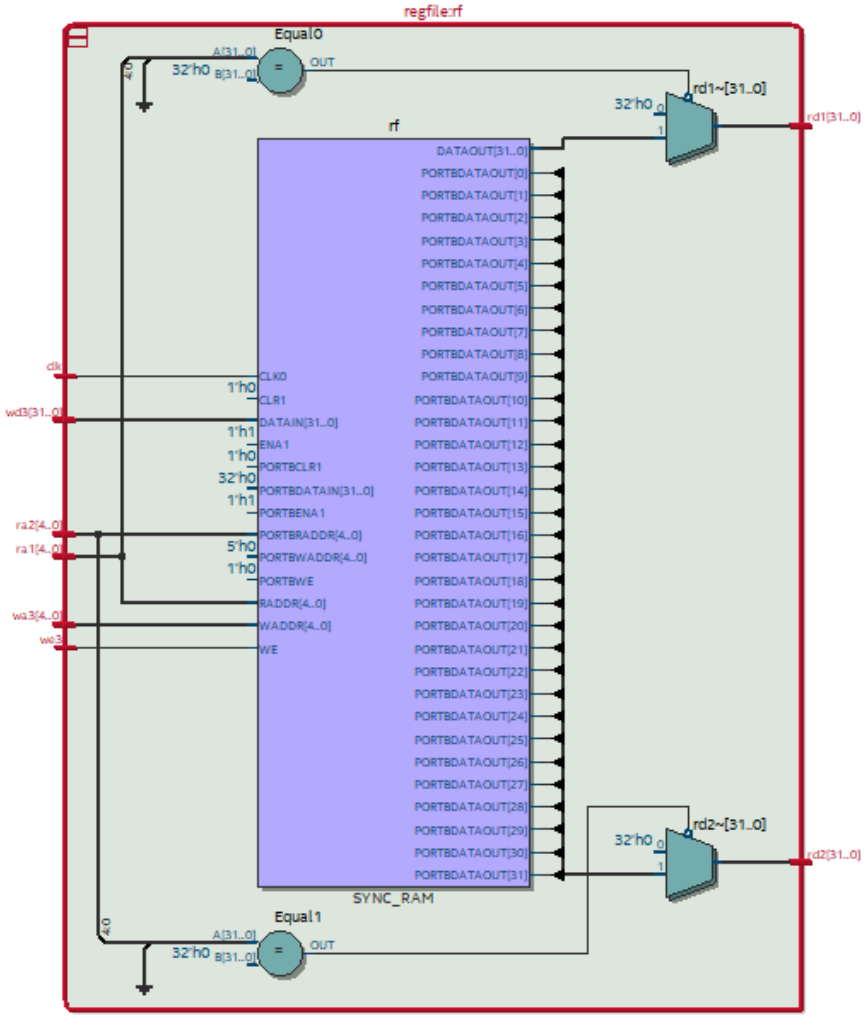
sl2



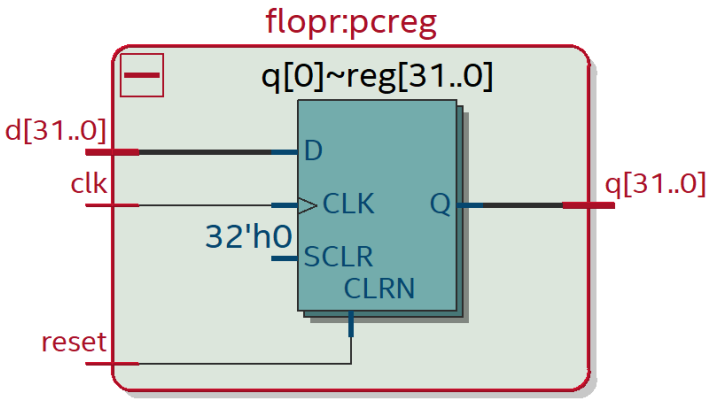
mux2



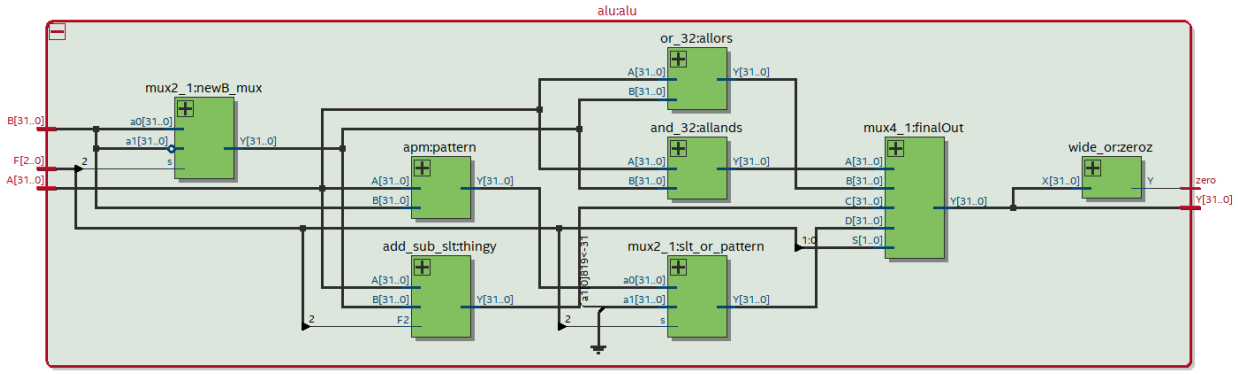
regfile



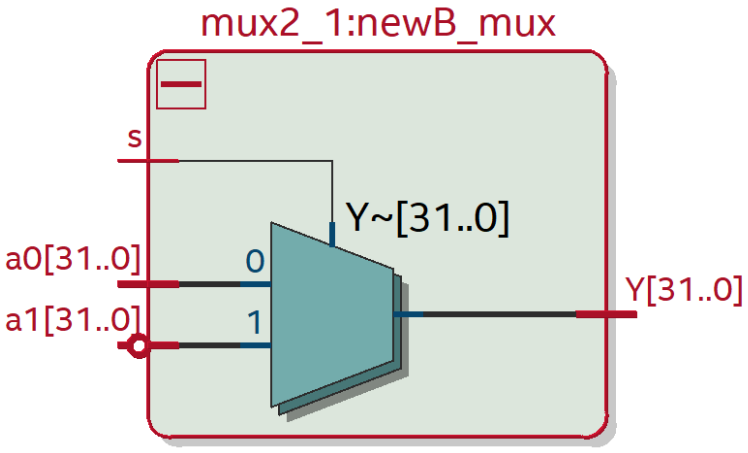
flopr



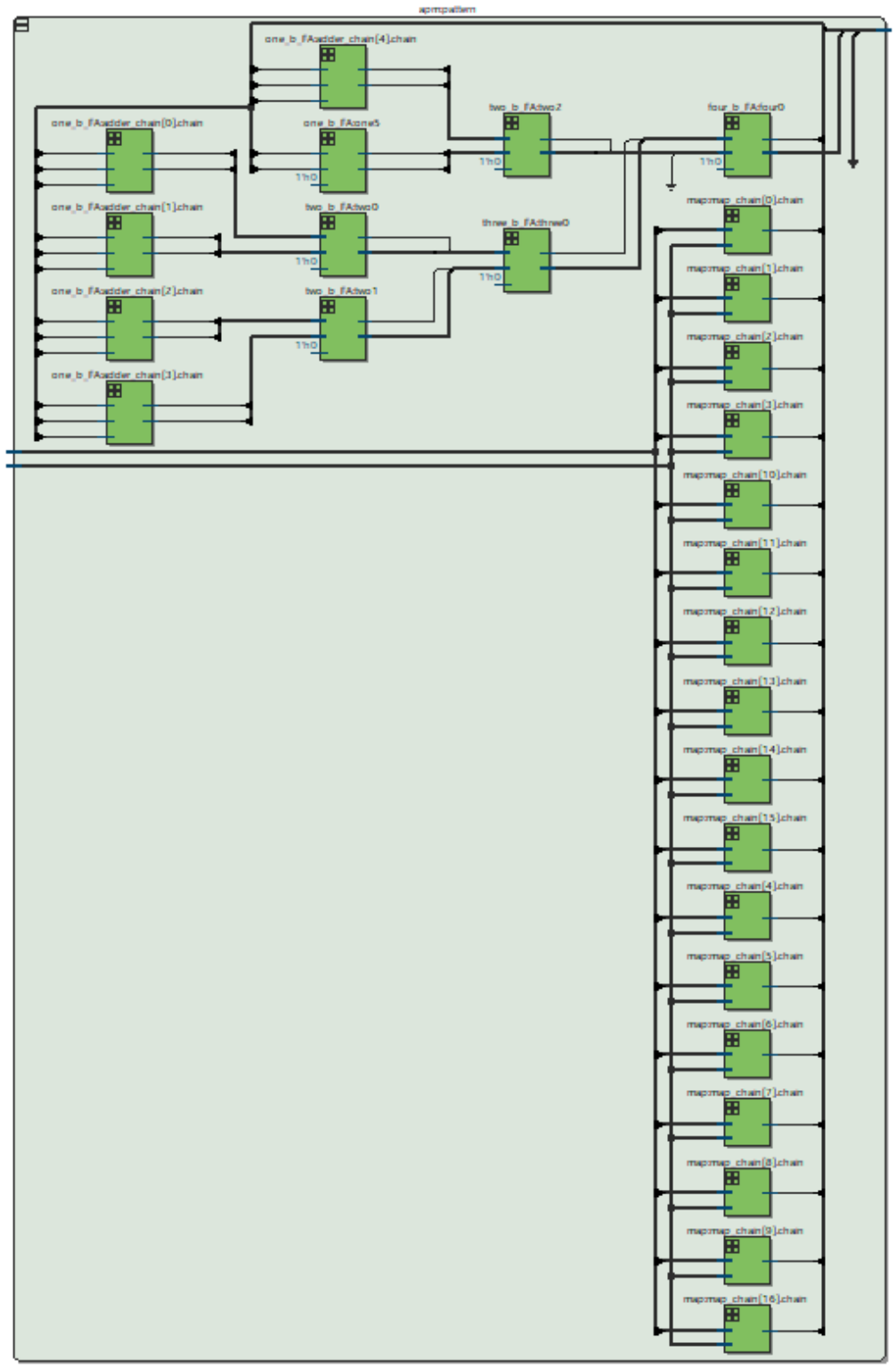
alu



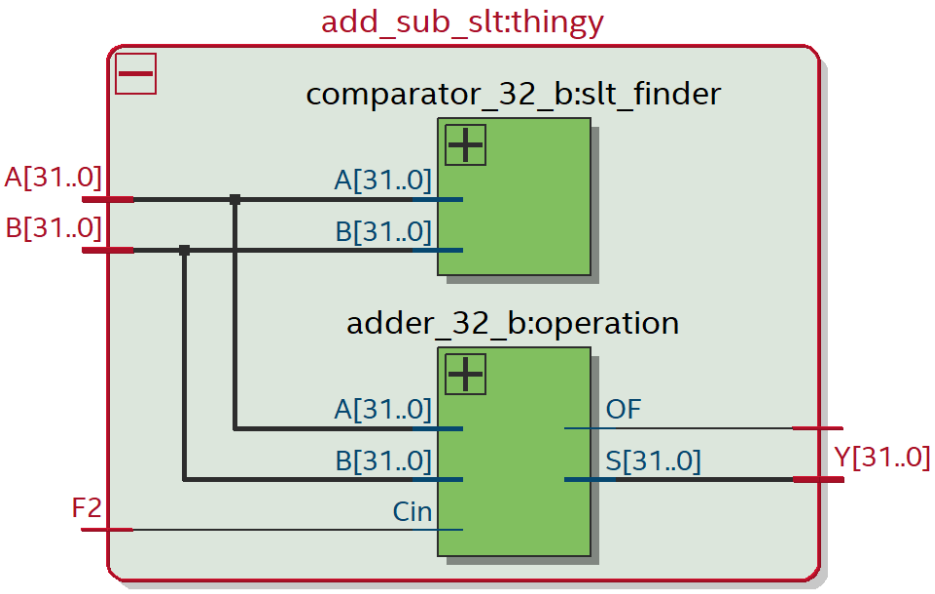
mux2_1



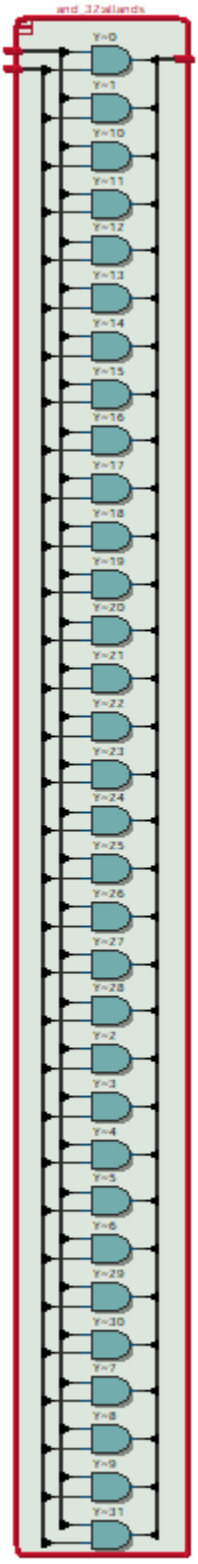
apm


apmtpathtem
one_b_FAadder_chain[4].chain
one_b_FAadder_chain[0].chain
one_b_FAone5
two_b_FAtwo2
four_b_FAfour0
1'h0
one_b_FAadder_chain[1].chain
two_b_FAtwo0
three_b_FAthree0
one_b_FAadder_chain[2].chain
two_b_FAtwo1
one_b_FAadder_chain[3].chain
mapmap_chain[0].chain
mapmap_chain[1].chain
mapmap_chain[2].chain
mapmap_chain[3].chain
mapmap_chain[10].chain
mapmap_chain[11].chain
mapmap_chain[12].chain
mapmap_chain[13].chain
mapmap_chain[14].chain
mapmap_chain[15].chain
mapmap_chain[4].chain
mapmap_chain[5].chain
mapmap_chain[6].chain
mapmap_chain[7].chain
mapmap_chain[8].chain
mapmap_chain[9].chain
mapmap_chain[16].chain

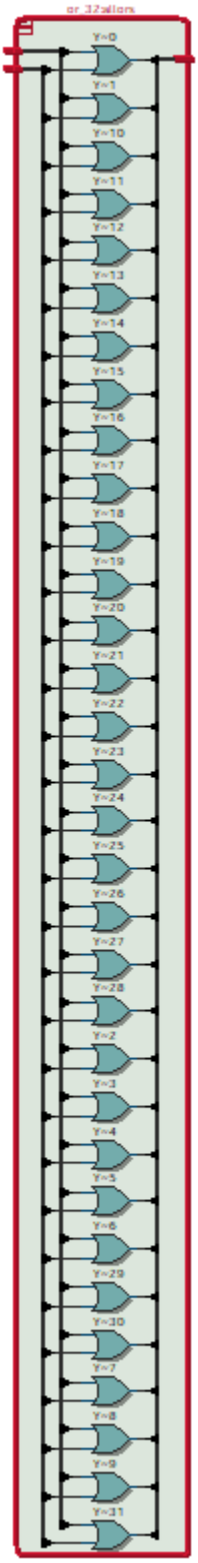
add_sub_slt


add_sub_slt:thingy
comparator_32_b:slt_finder
A[31..0]
B[31..0]
A[31..0]
B[31..0]
adder_32_b:operation
A[31..0]
B[31..0]
Cin
OF
S[31..0]
F2
Y[31..0]

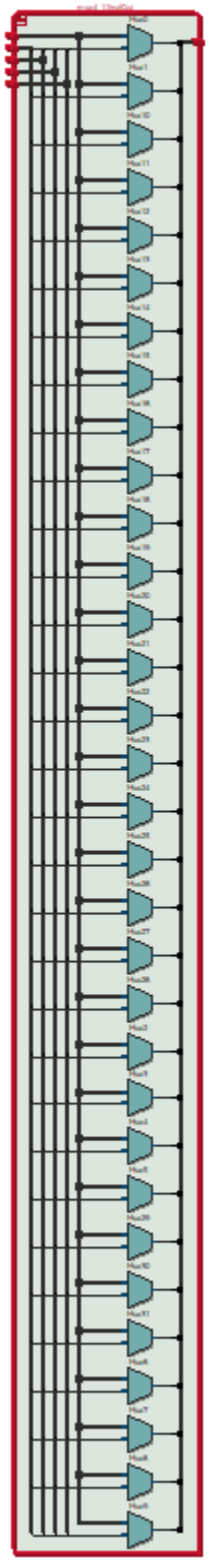
and_32

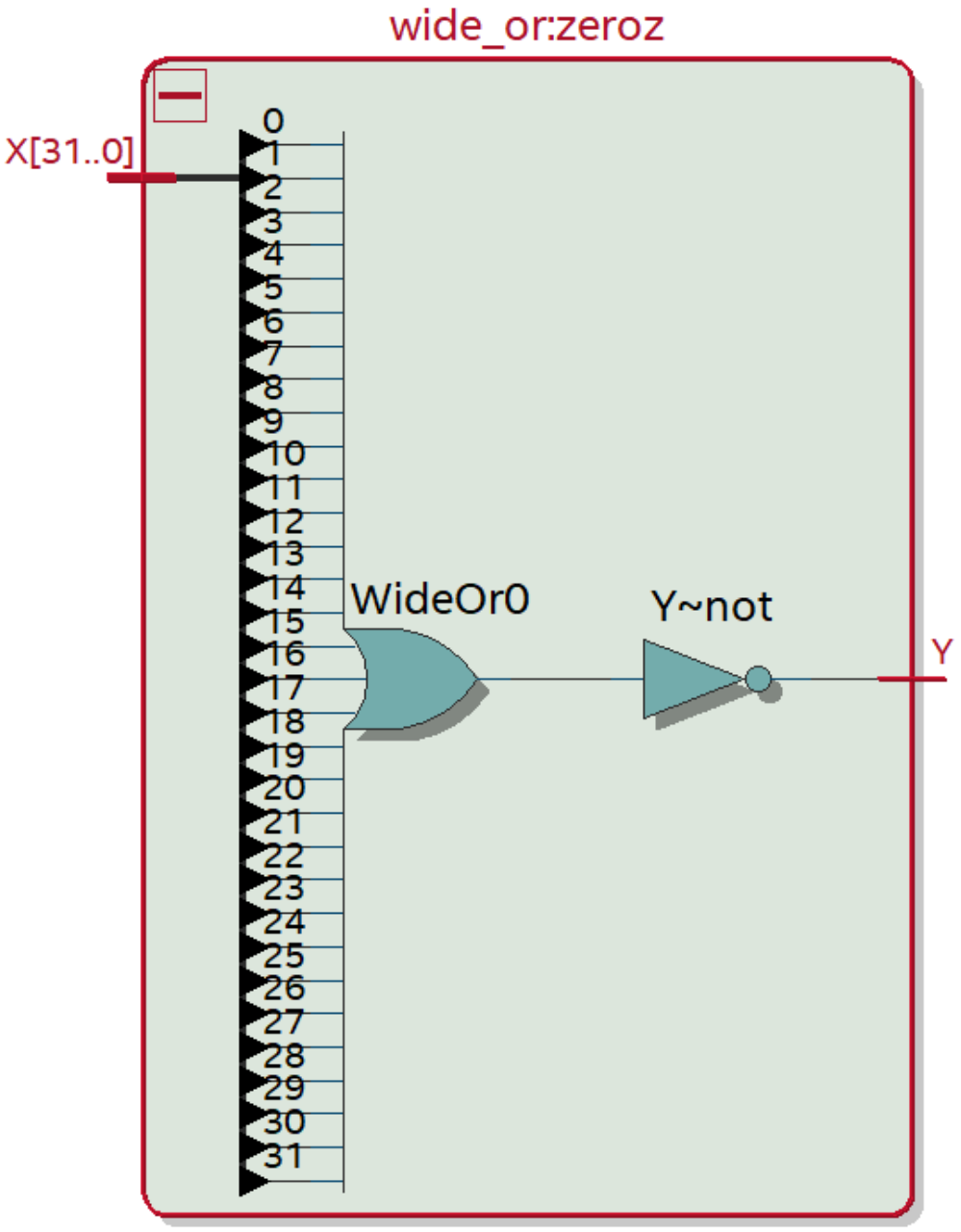

and_32allandx
Y=0
Y=1
Y=10
Y=11
Y=12
Y=13
Y=14
Y=15
Y=16
Y=17
Y=18
Y=19
Y=20
Y=21
Y=22
Y=23
Y=24
Y=25
Y=26
Y=27
Y=28
Y=2
Y=3
Y=4
Y=5
Y=6
Y=29
Y=30
Y=7
Y=8
Y=9
Y=31

or_32

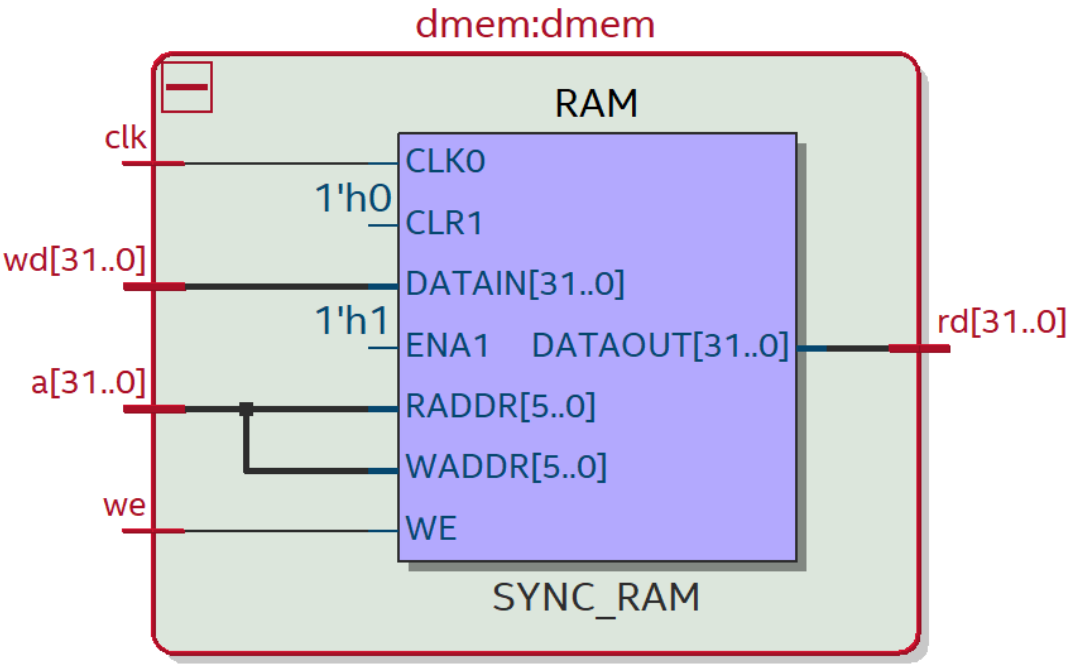

or_32allors
Y=0
Y=1
Y=10
Y=11
Y=12
Y=13
Y=14
Y=15
Y=16
Y=17
Y=18
Y=19
Y=20
Y=21
Y=22
Y=23
Y=24
Y=25
Y=26
Y=27
Y=28
Y=2
Y=3
Y=4
Y=5
Y=6
Y=29
Y=30
Y=7
Y=8
Y=9
Y=31

mux4_1

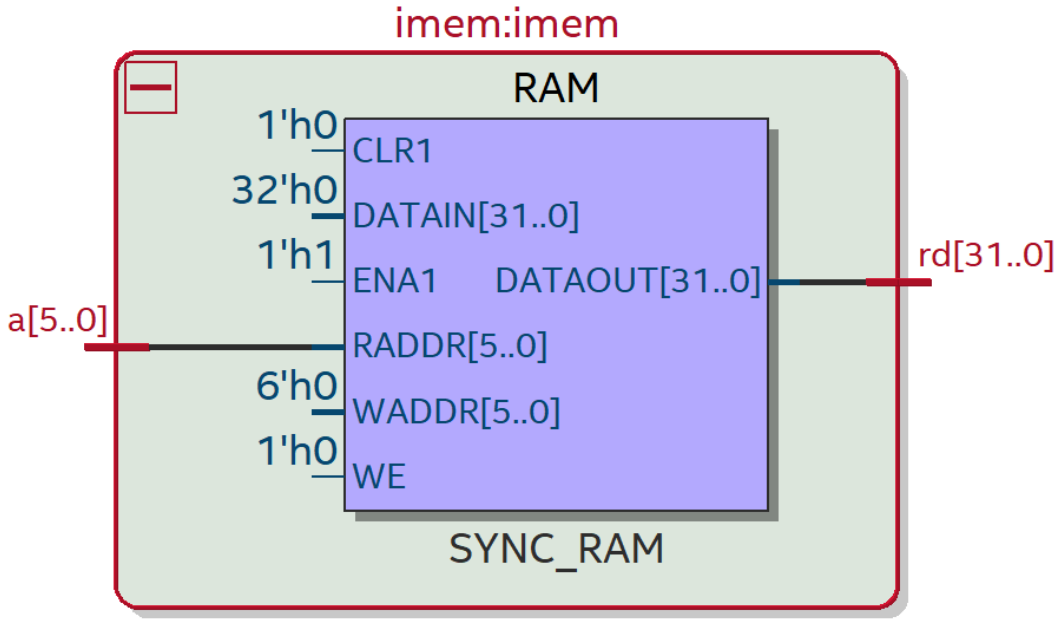



wide_or:zeroz
X[31..0]
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
WideOr0
Y~not
Y

dmem

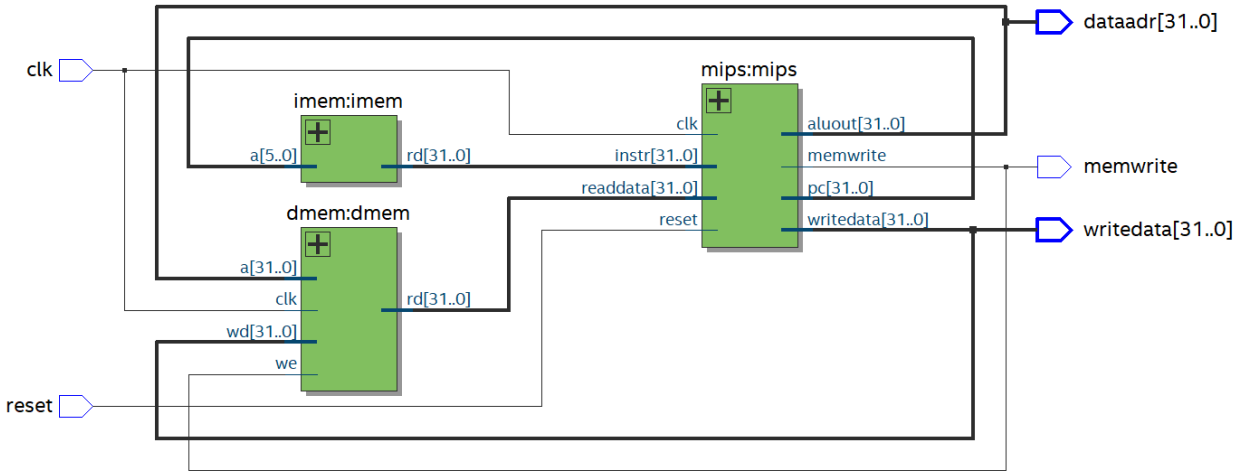

dmem:dmem
RAM
clk
CLK0
1'h0
CLR1
wd[31..0]
DATAIN[31..0]
1'h1
ENA1
DATAOUT[31..0]
rd[31..0]
a[31..0]
RADDR[5..0]
WADDR[5..0]
we
WE
SYNC_RAM

imem

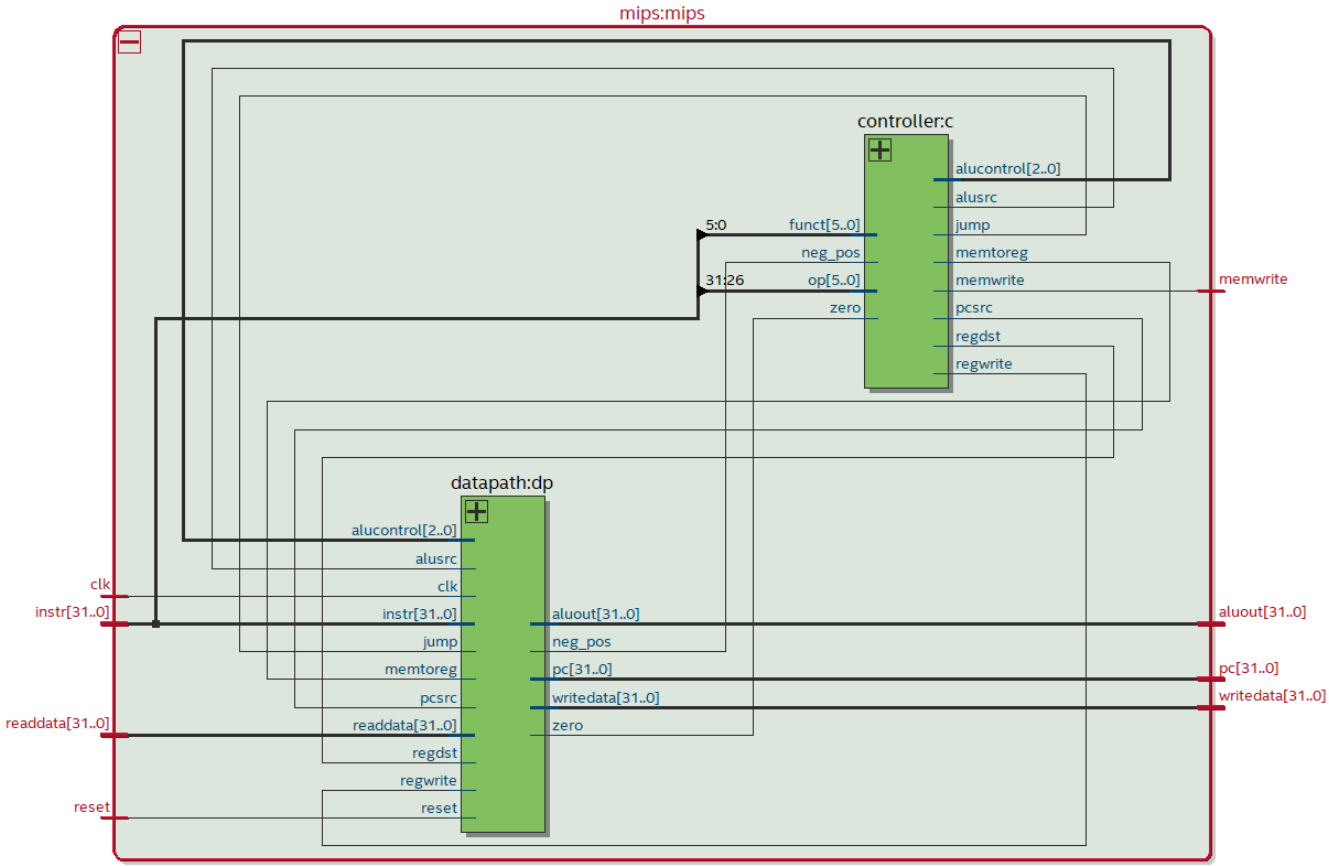

imem:imem
RAM
1'h0
CLR1
32'h0
DATAIN[31..0]
1'h1
ENA1
DATAOUT[31..0]
rd[31..0]
a[5..0]
RADDR[5..0]
6'h0
WADDR[5..0]
1'h0
WE
SYNC_RAM

After Modification

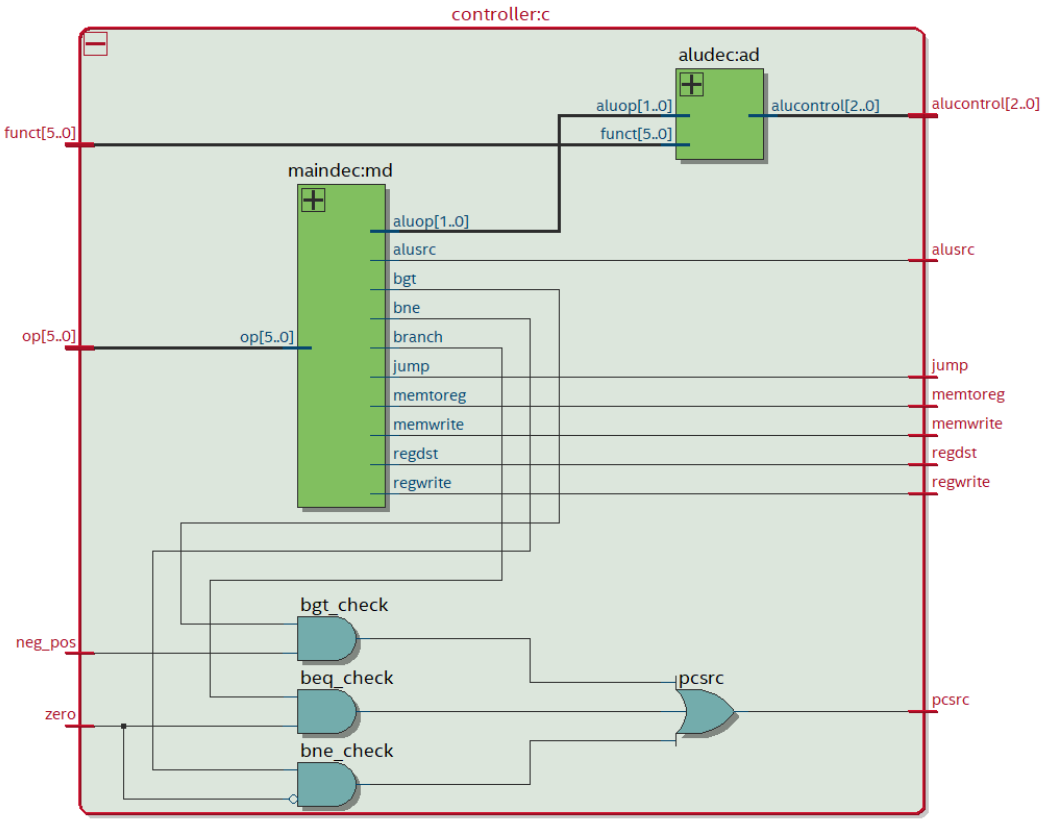
top



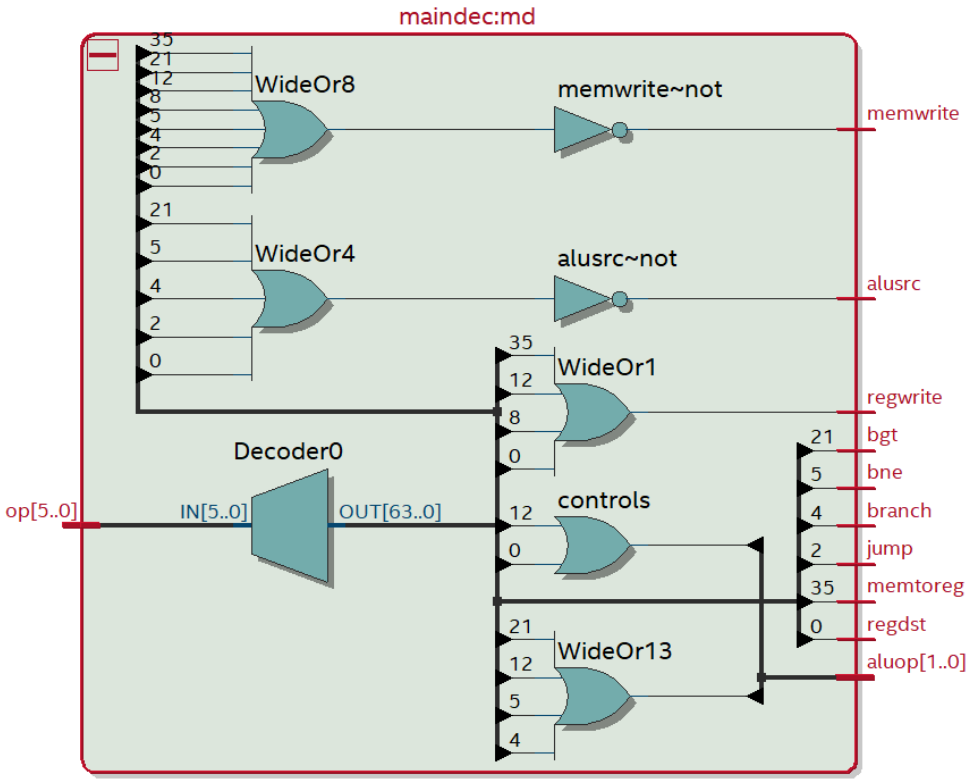
mips



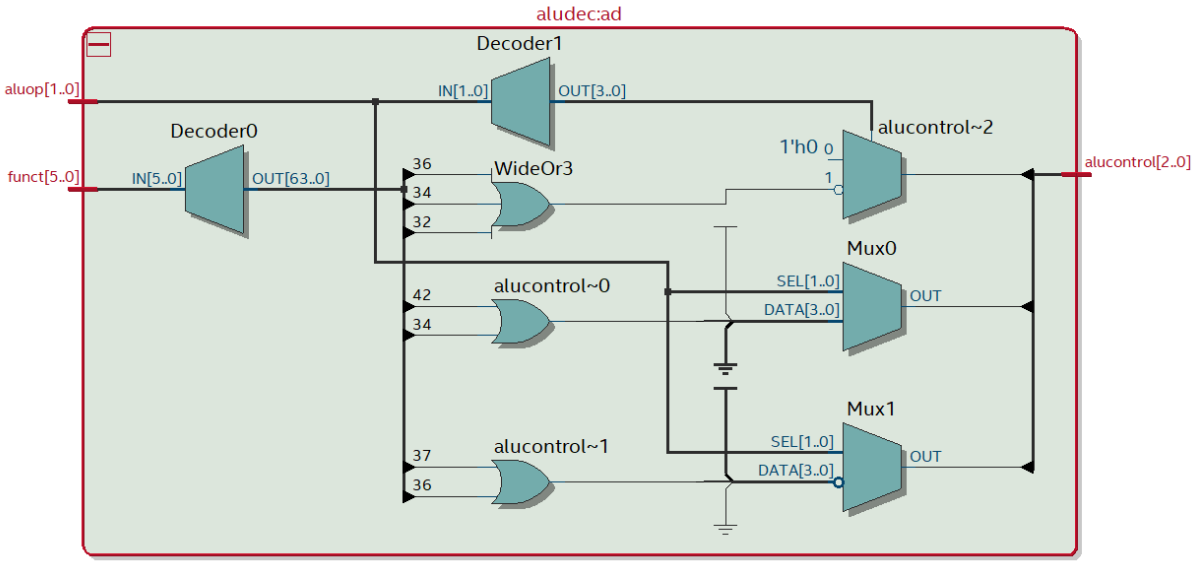
controller



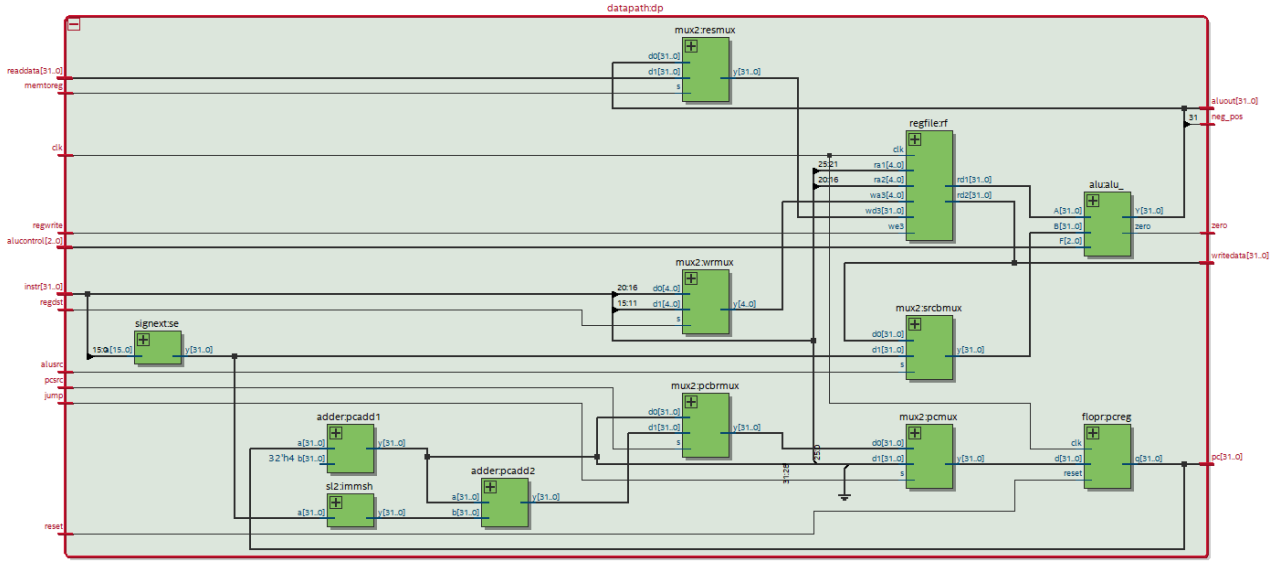
maindec



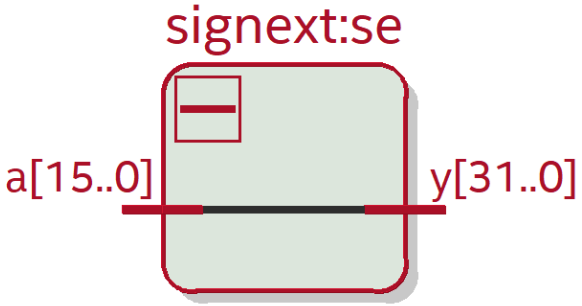
aludec



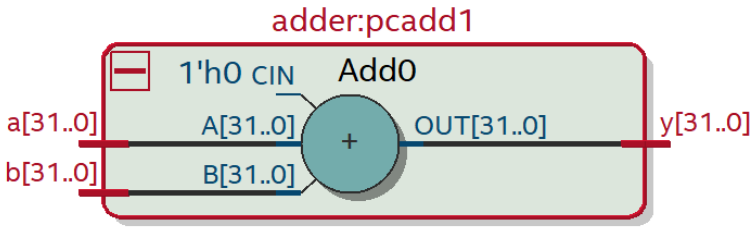
datapath



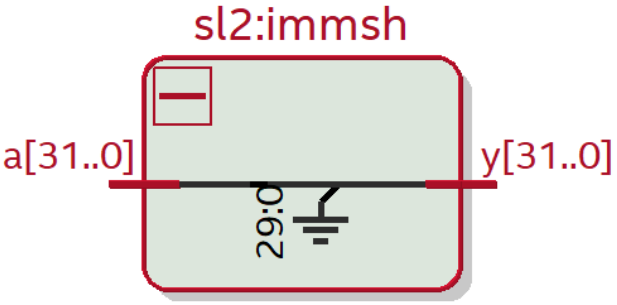
signext



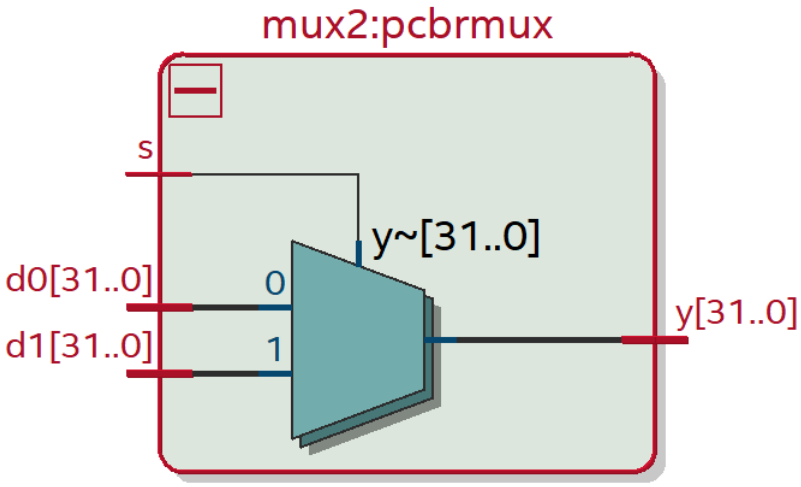
adder



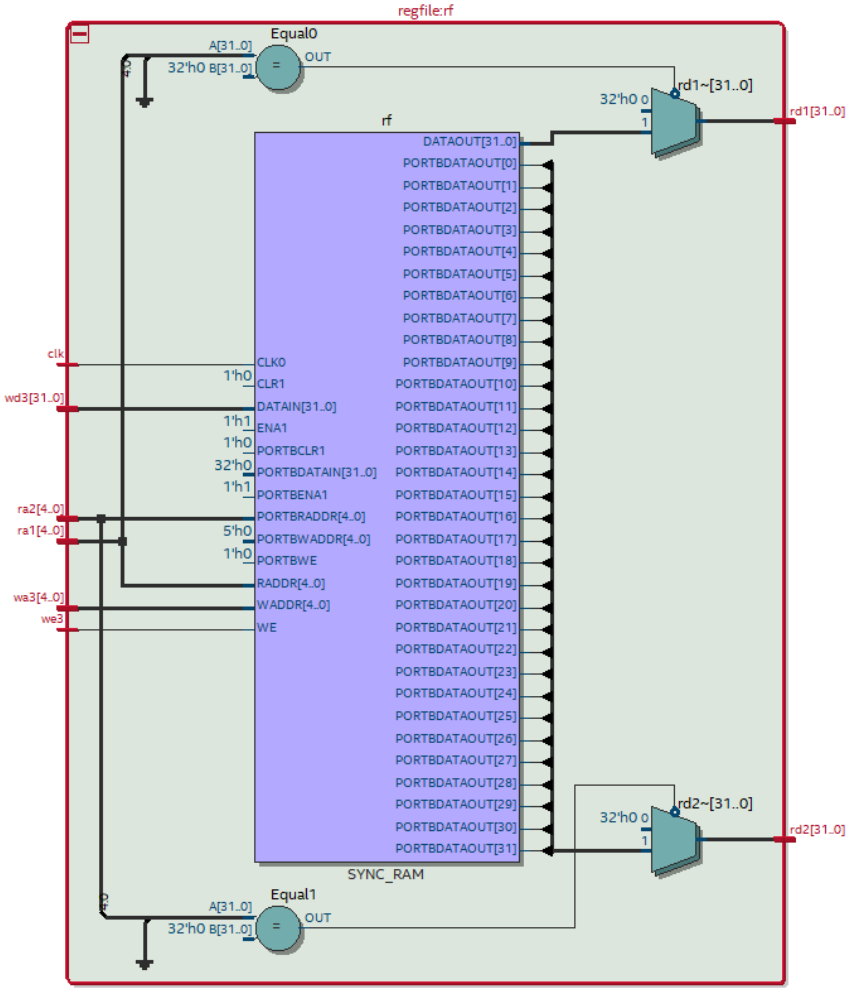
sl2



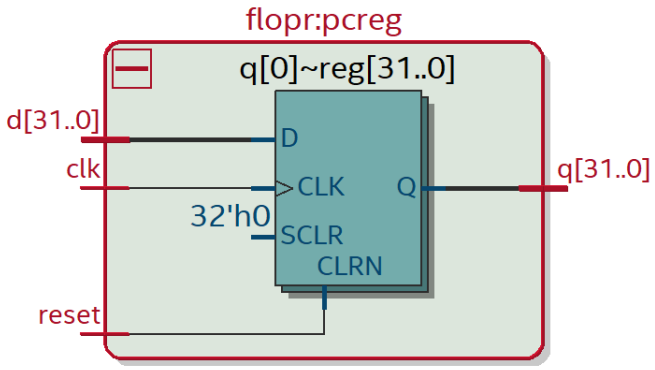
mux2



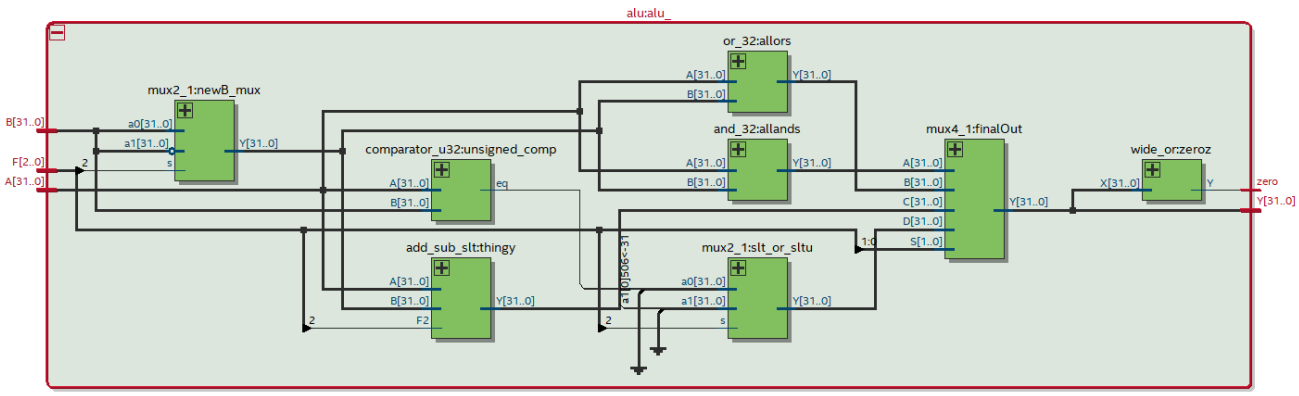
regfile



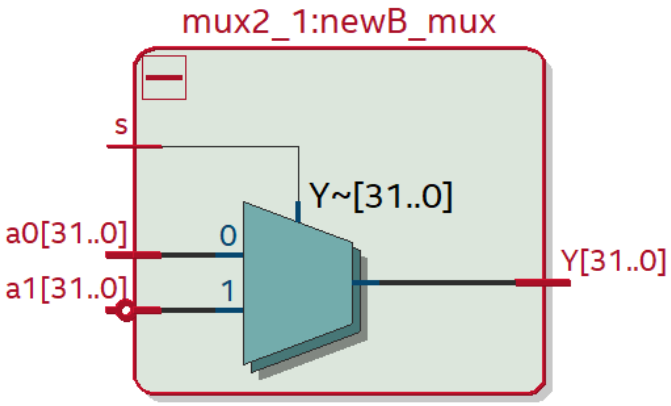
flopr



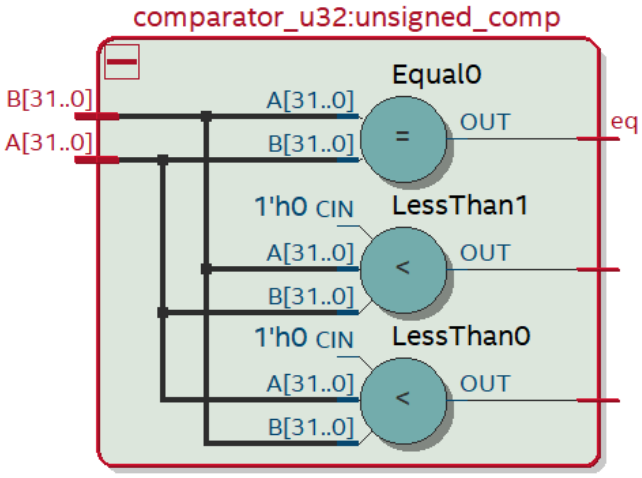
alu



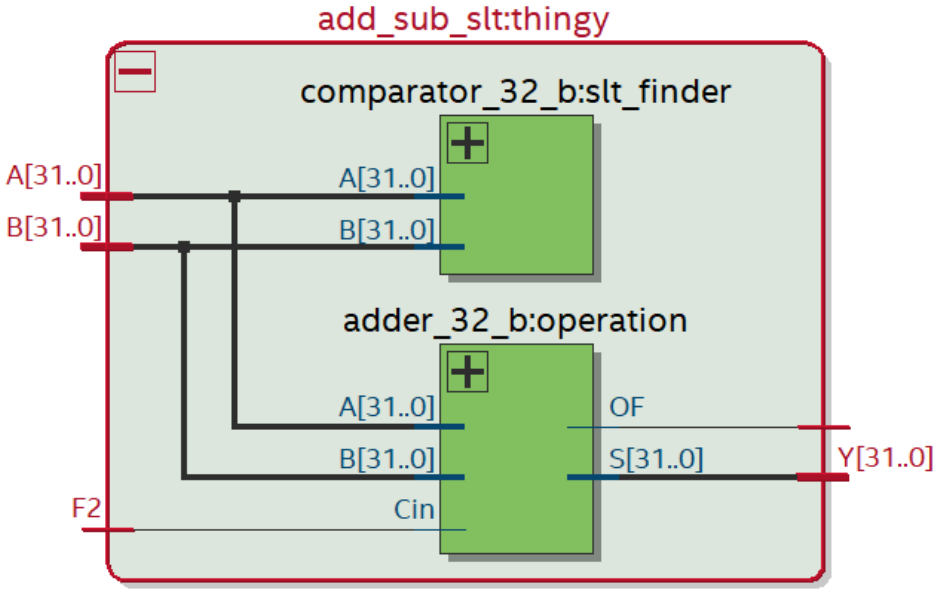
mux2_1



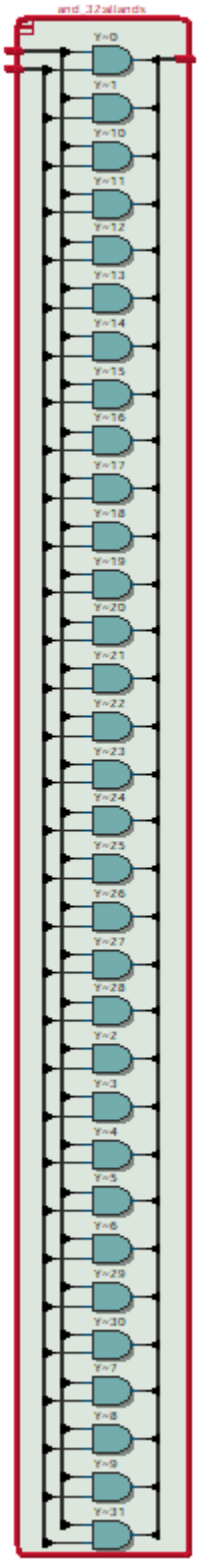
comparator_u32



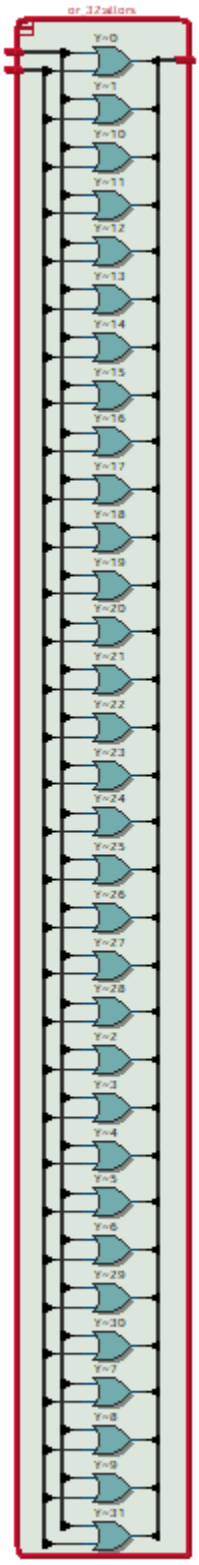
add_sub_slt


add_sub_slt:thingy
comparator_32_b:slt_finder
A[31..0]
B[31..0]
A[31..0]
B[31..0]
adder_32_b:operation
A[31..0]
B[31..0]
F2
Cin
OF
S[31..0]
Y[31..0]

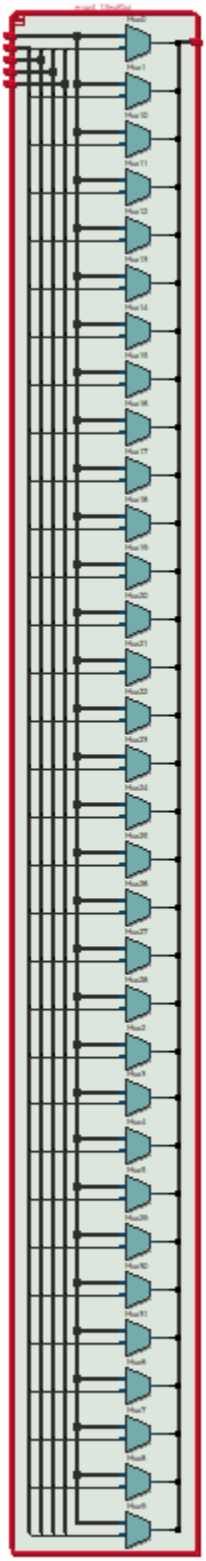
and_32


and_32allands
Y=0
Y=1
Y=10
Y=11
Y=12
Y=13
Y=14
Y=15
Y=16
Y=17
Y=18
Y=19
Y=20
Y=21
Y=22
Y=23
Y=24
Y=25
Y=26
Y=27
Y=28
Y=2
Y=3
Y=4
Y=5
Y=6
Y=29
Y=30
Y=7
Y=8
Y=9
Y=31

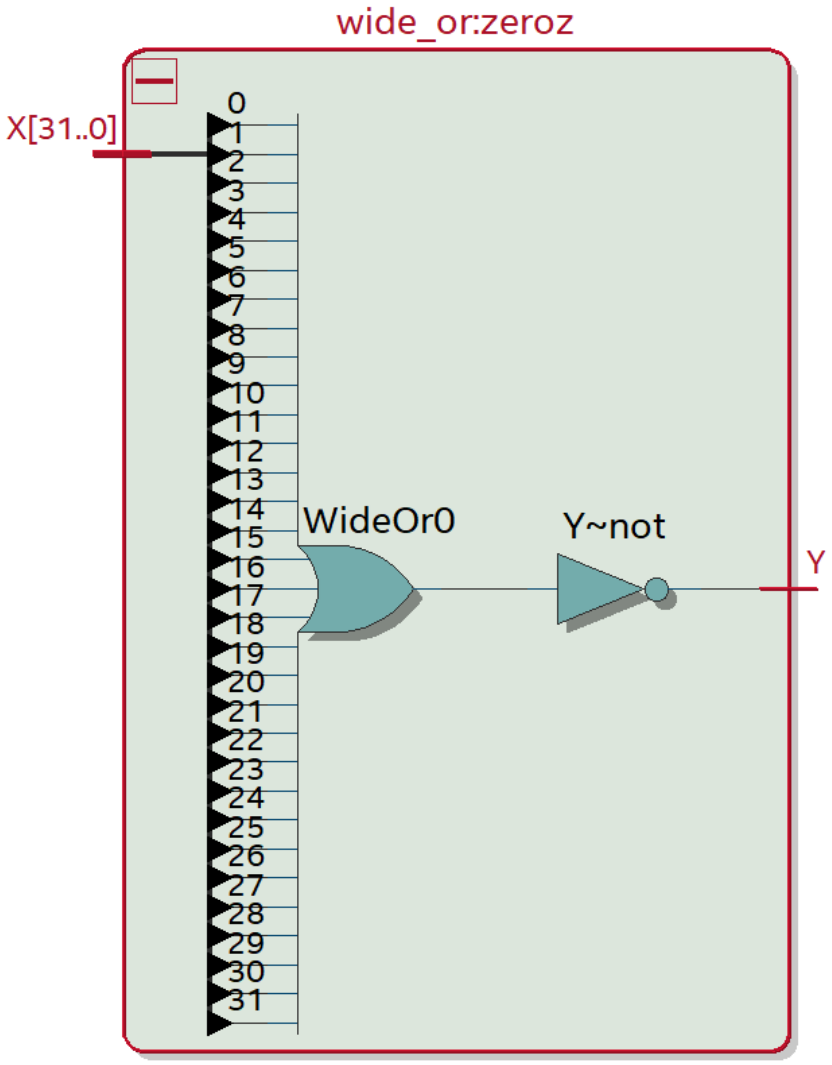
or_32


or_32alljpg
Y=0
Y=1
Y=10
Y=11
Y=12
Y=13
Y=14
Y=15
Y=16
Y=17
Y=18
Y=19
Y=20
Y=21
Y=22
Y=23
Y=24
Y=25
Y=26
Y=27
Y=28
Y=2
Y=3
Y=4
Y=5
Y=6
Y=29
Y=30
Y=7
Y=8
Y=9
Y=31

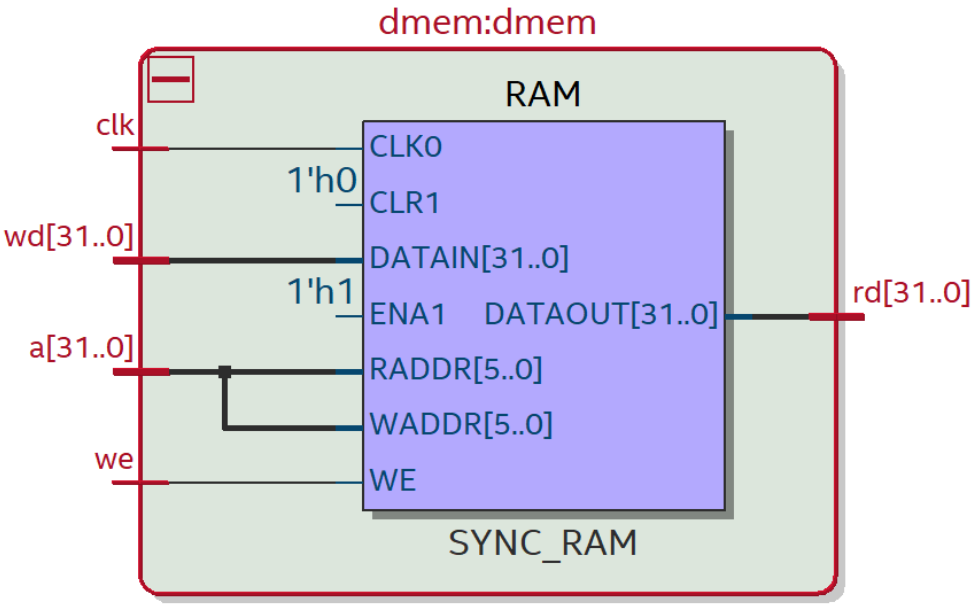
mux4_1



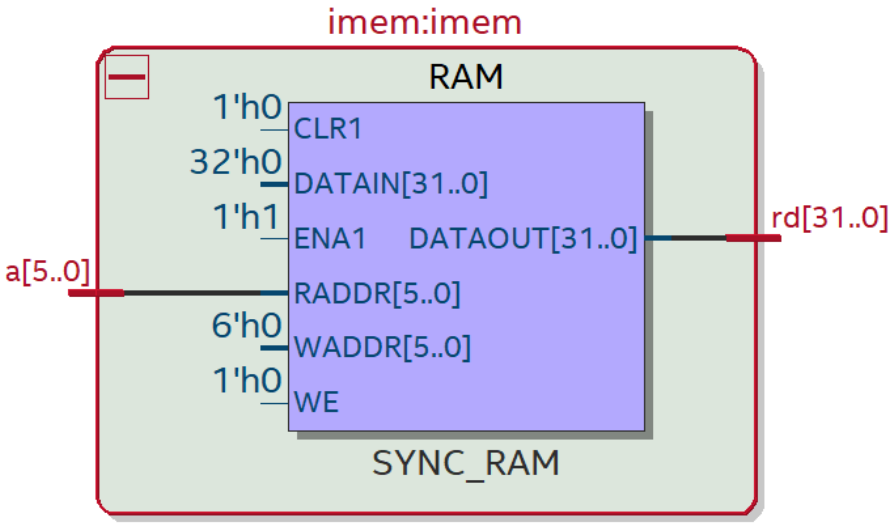
wide_or


wide_or:zeroz
X[31..0]
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
WideOr0
Y~not
Y

dmem


dmem:dmem
RAM
clk
CLK0
1'h0
CLR1
wd[31..0]
DATAIN[31..0]
1'h1
ENA1
DATAOUT[31..0]
rd[31..0]
a[31..0]
RADDR[5..0]
WADDR[5..0]
we
WE
SYNC_RAM

imem


imem:imem
RAM
1'h0
CLR1
32'h0
DATAIN[31..0]
1'h1
ENA1
DATAOUT[31..0]
rd[31..0]
a[5..0]
RADDR[5..0]
6'h0
WADDR[5..0]
1'h0
WE
SYNC_RAM

Workload report

How many hours have you spent for this lab?

We spent about 15 hours total to do this lab, including parts A - C, testbenches, and the report as well.

Which activity takes the most significant amount of time?

Creating the testbenches and simulating the CPU takes the most time, as ModelSim is very slow and takes forever to restart simulations after making small tweaks to the .sv code. Another good portion of the lab is creating the report, getting all of the Quartus screenshots and making everything look nice.