Tyler Thompson

ECE 3570 Lab 3a
10-bit Instruction Set Architecture CPU - Single-Cycle CPU

March 12th, 2018

Critical Path Delay

Register File: 6.8ns
ALU: 7.9ns (Longest Case)
Fetch Unit: 3.9ns
Control Unit: 6.02ns
CPU Module: 5.17ns

Clock Period

Due to the ALU having a the longest critical path delay, the clock period was chosen to be **8ns**. This is the minimum clock period time. The clock cycle time was minimized by having the positive edge of the clock start the clock period. Register are read at the positive edge, so their contents can be sent to the ALU immediately. Since the ALU is the slowest component, feeding it input at the beginning of the clock cycle allows the cycle time to be minimized to the critical path delay of the ALU. This may change once memory is implemented.

Design Changes

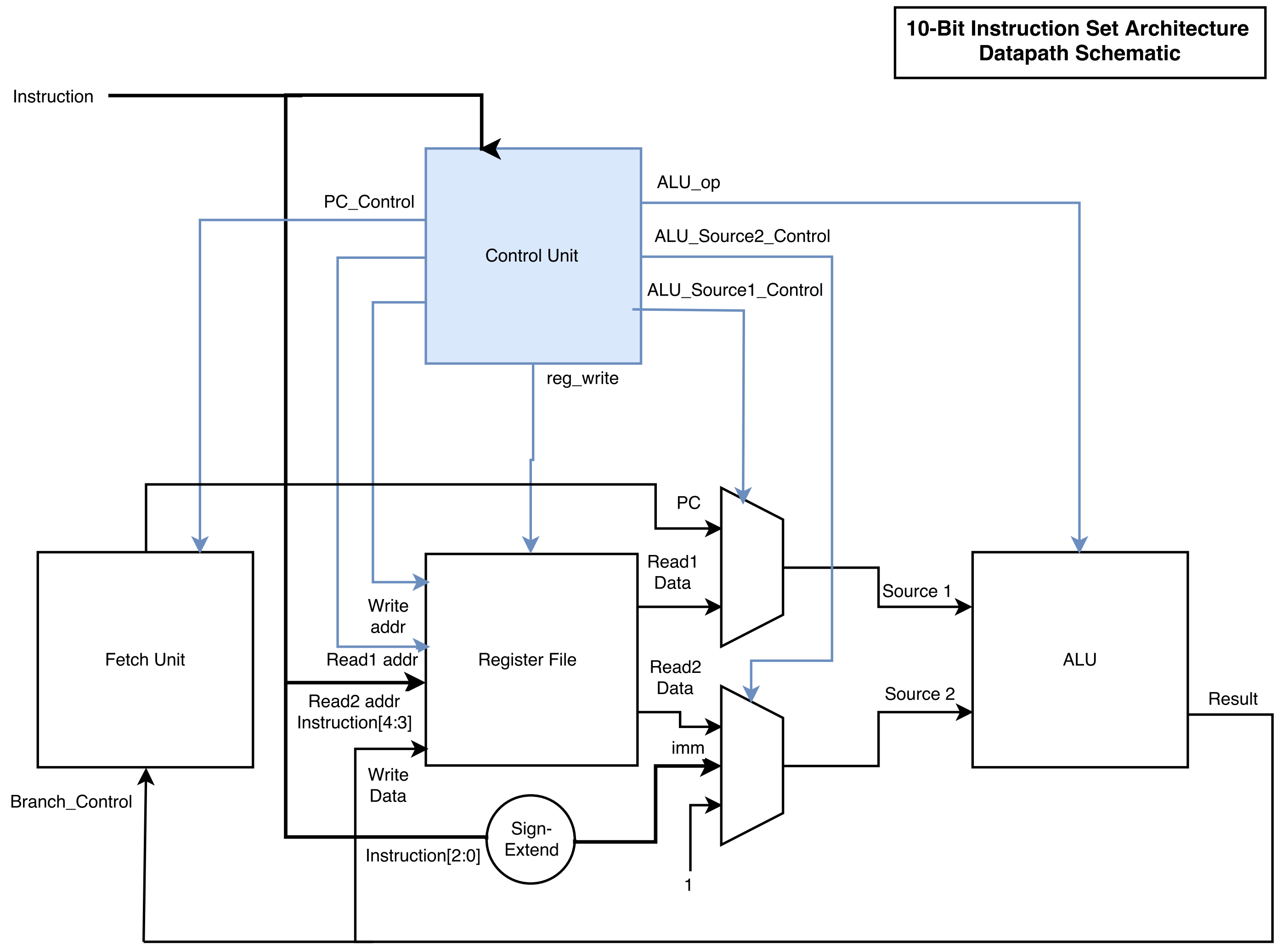
1. The set less than (slt) instruction was changed to compare (cmp). The compare instruction can determine if a number is greater than, less than, or equal to another number.
2. The jump register (jr) instruction was added to help support function call returning.
3. The two's complement (tcp) instruction was added to help deal with negative numbers. It uses the same op code as the addi instruction, but is only called if the immediate value of the instruction is set to negative zero (100).
4. All of the modules in Lab 2 were slightly modified in order to remove latches from their schematics. These changes did not affect the behavior of the modules in the simulator.
5. An input signal was added to the fetch unit that is fed by the result of the ALU. This allows the fetch unit to know when branch instructions execute successful and is more efficient than adding more control lines to the control unit.

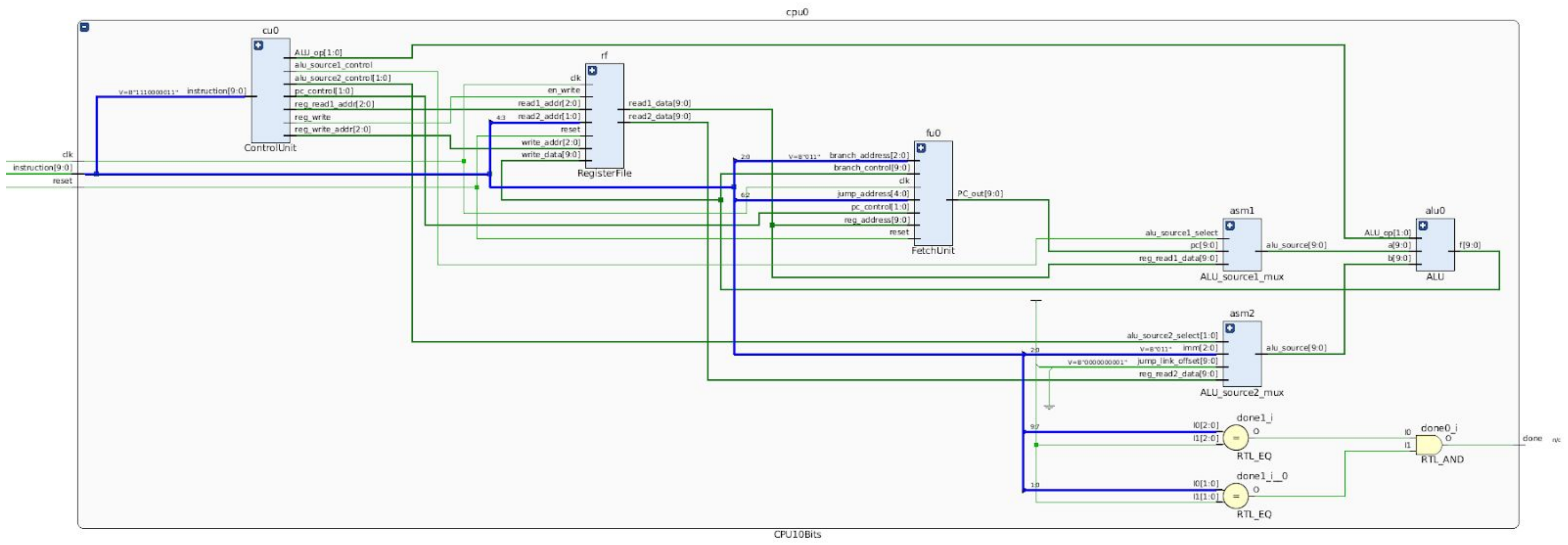
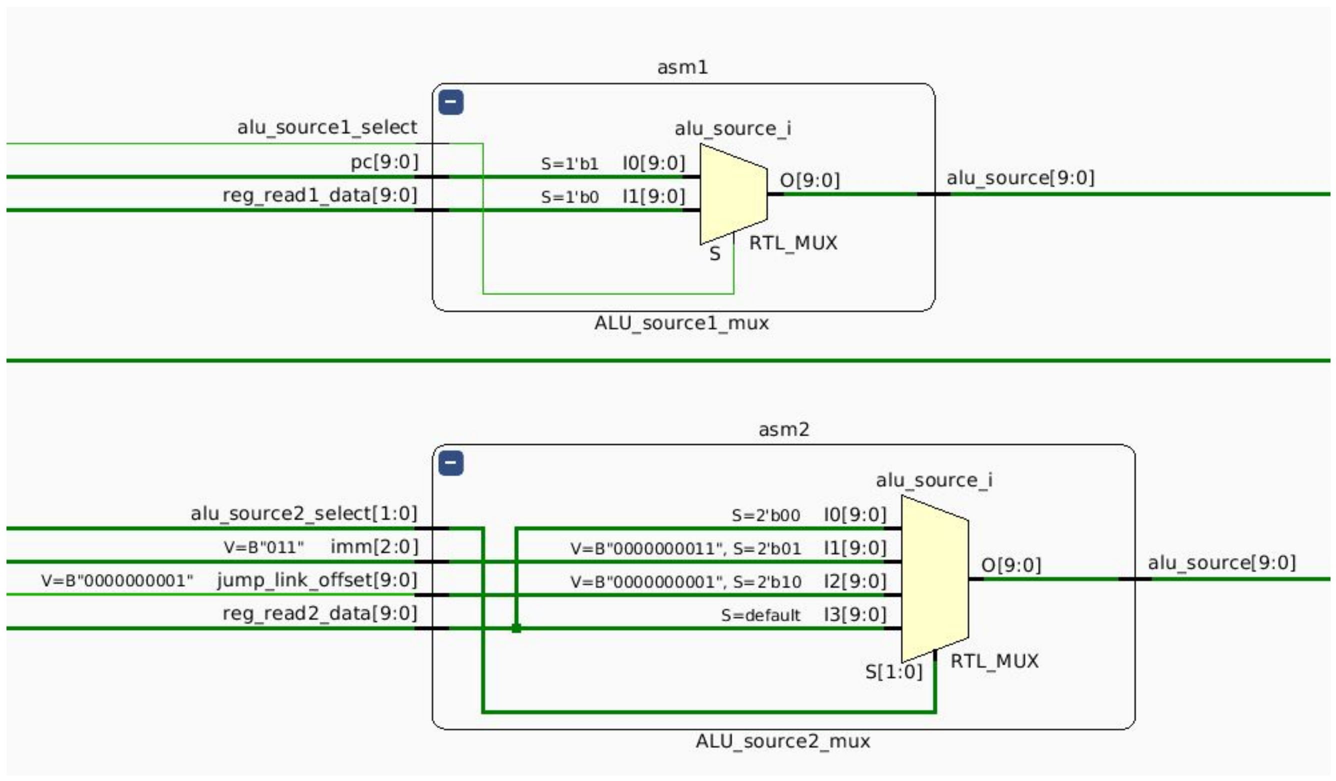
10-Bit ISA Reference Sheet

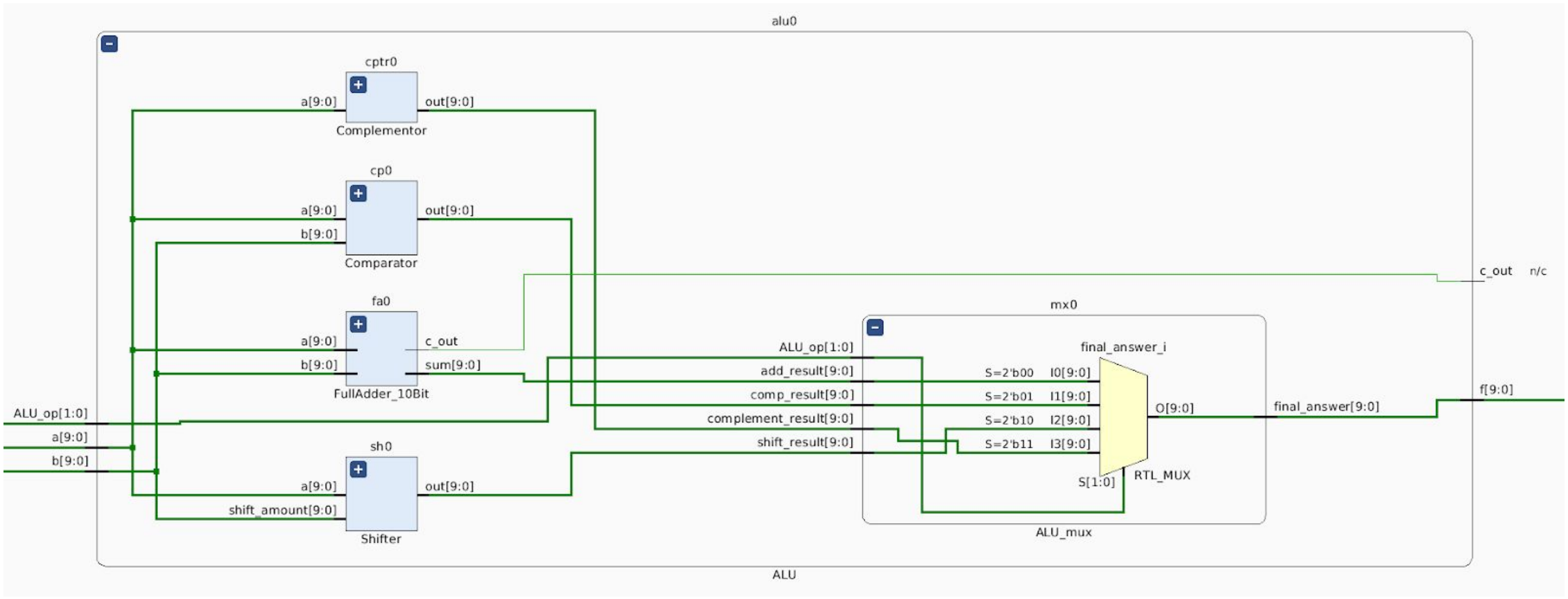
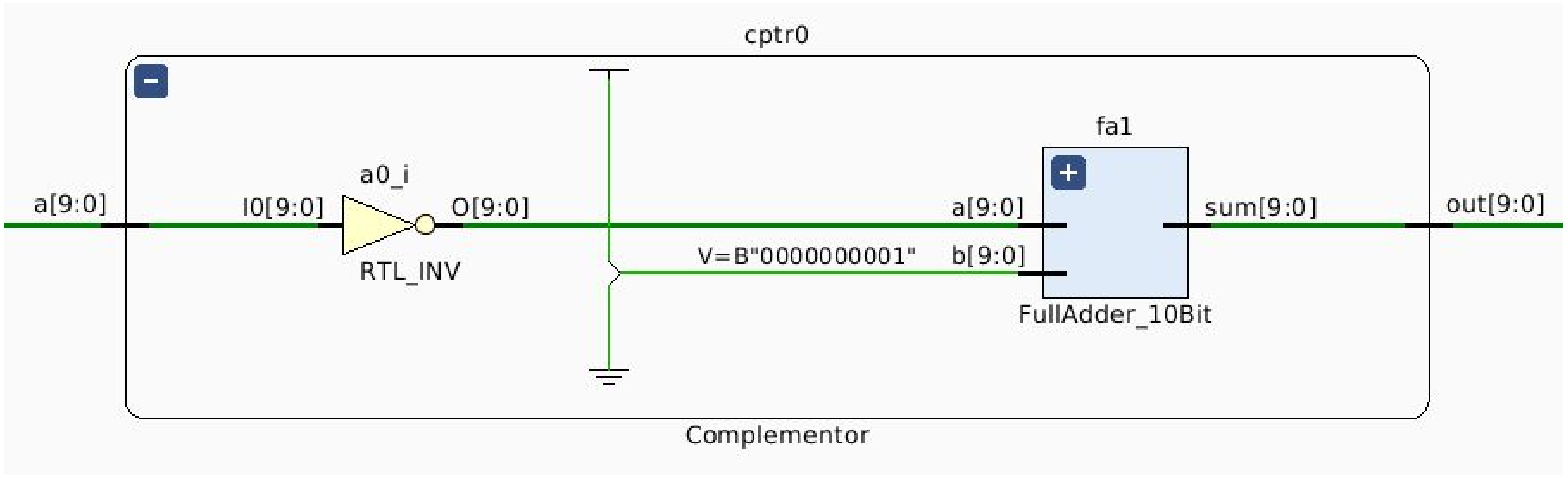
Instructions						
Name	**Description**	**Format**	**RTL**	**Syntax**	**OP Code (hex)**	**Function Code (hex)**
add	Add two values in registers together	R	if MEM[PC] == ADD rs rt rd [rs] <= [rt] + [rd] PC <= PC + 1	add $rs, $rt, $rd	0x0	
addi	Add a 3-bit signed contant to a value in a register	I	if MEM[PC] == ADDI rs rt imm [rs] <= [rt] + sign-ext(imm) PC <= PC + 1	addi $rs, $rt, imm	0x4	
tcp	Take the two's complement of a value in a registe Imm value should be set to 100 (-0) to differentiate from addi	I	if MEM[PC] == TCP rs rt [rs] <= ~[rt] + 1 PC <= PC + 1	tcp $rs, $rt	0x4	
sw	Store a word in memory	I	if MEM[PC] == SW rt offset(base) address = sign-extend(offset) + [base] MEM[address] <= [rt] PC <= PC + 1	sw $rs, M($rt)	0x5	
lw	Load a word from memory	I	if MEM[PC] == LW rt offset(base) address = sign-extend(offset) + [base] [rt] <= MEM[address] PC <= PC + 1	lw $rs, M($rt)	0x6	
sll	Shift left logical	R	if MEM[PC] == SLL rd rs rt [rd] <= [rs] << [rt] PC = PC + 1	sll $rd, $rs, $rt	0x1	
cmp	Compare	R	if MEM[PC] == CMP rd rs rt [rd] <= [rs] < [rt] - 1 [rd] <= [rs] == [rt] - 0 [rd] <= [rs] > [rt] - 2 PC = PC + 1	cmp $rd, $rs, $rt	0x2	

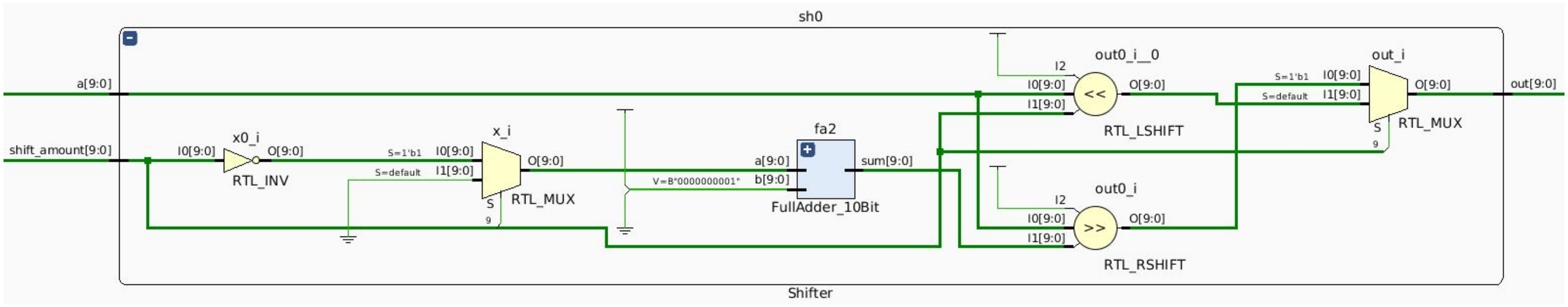
beq	Branch equal	R	if MEM[PC] == BEQ rd rs rt
			if [rs] == [rt]
			PC <= PC + 4 + [rd]
				beq $rd, $rs, $rt	0x3	

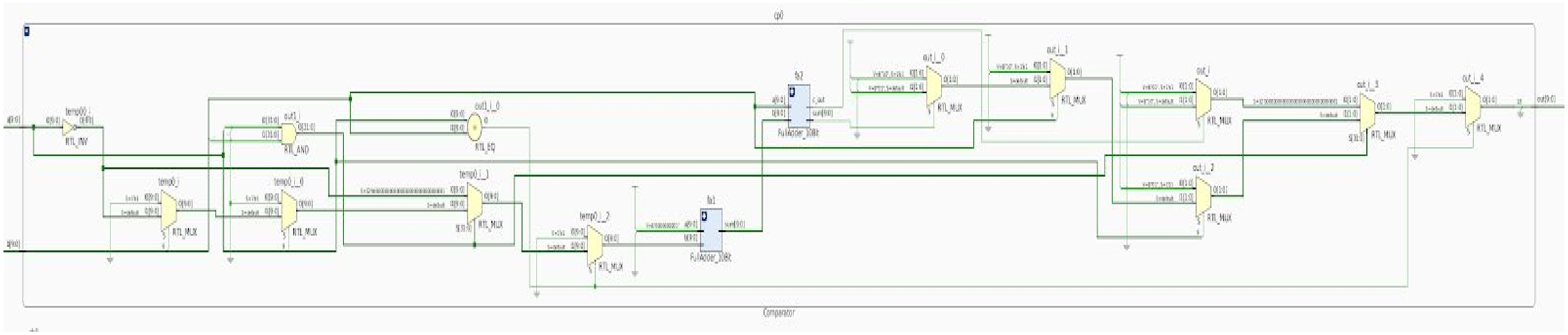
jal	Jump and link	J	if MEM[PC] == JAL address [ra] <= PC + 1 PC <= PC + address	jal Label	0x7	0x2
j	Jump	J	if MEM[PC] == J address PC <= PC + address	j Label	0x7	0x1
jr	Jump Register	J	if MEM[PC] == JR $reg PC <= $reg	jr $reg	0x7	0x0
halt	Halt the machine	J	if MEM[PC] == HALT PC <= 0x0	halt	0x7	0x3


10-Bit Instruction Set Architecture
Datapath Schematic
Instruction
Control Unit
PC_Control
ALU_op
ALU_Source2_Control
ALU_Source1_Control
reg_write
PC
Read1
Data
Source 1
Fetch Unit
Write
addr
Read1 addr
Register File
ALU
Read2
Data
Source 2
Result
Read2 addr
Instruction[4:3]
imm
Write
Data
Branch_Control
Sign-
Extend
Instruction[2:0]
1

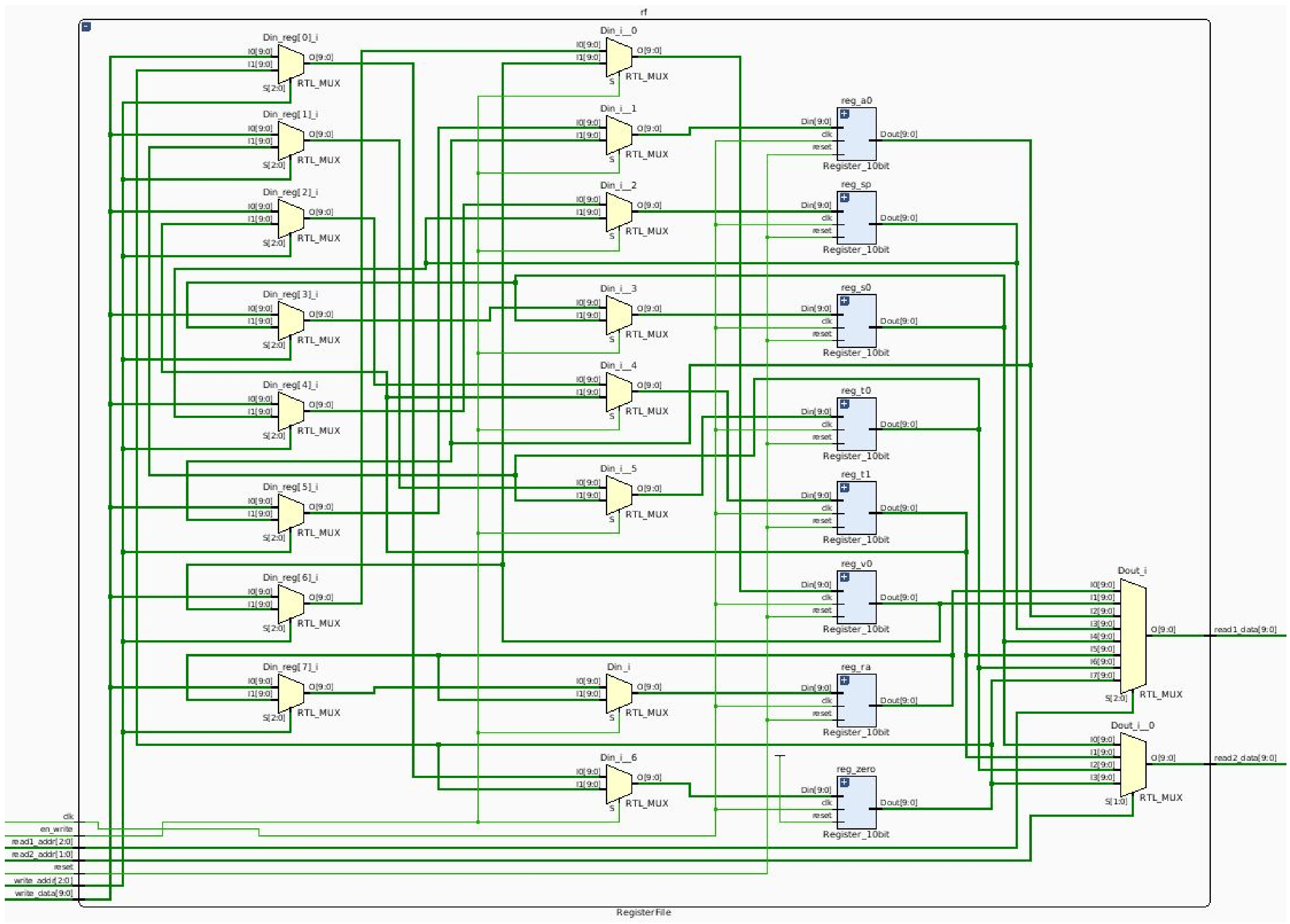

cpu0
cu0
ALU_op[1:0]
alu_source1_control
alu_source2_control[1:0]
pc_control[1:0]
reg_read1_addr[2:0]
reg_write
reg_write_addr[2:0]
instruction[9:0]
ControlUnit
rf
clk
en_write
read1_addr[2:0]
read2_addr[1:0]
reset
write_addr[2:0]
write_data[9:0]
read1_data[9:0]
read2_data[9:0]
RegisterFile
fu0
branch_address[2:0]
branch_control[9:0]
clk
jump_address[4:0]
pc_control[1:0]
reg_address[9:0]
reset
PC_out[9:0]
FetchUnit
asm1
alu_source1_select
pc[9:0]
reg_read1_data[9:0]
alu_source[9:0]
ALU_source1_mux
alu0
ALU_op[1:0]
a[9:0]
b[9:0]
f[9:0]
ALU
asm2
alu_source2_select[1:0]
imm[2:0]
jump_link_offset[9:0]
reg_read2_data[9:0]
alu_source[9:0]
ALU_source2_mux
done1_i
RTL_EQ
done1_i__0
RTL_EQ
done0_i
RTL_AND
done
clk
instruction[9:0]
reset
CPU10Bits

asm1
alu_source1_select
pc[9:0]
reg_read1_data[9:0]
alu_source_i
S=1'b1 I0[9:0]
S=1'b0 I1[9:0]
O[9:0]
S
RTL_MUX
alu_source[9:0]
ALU_source1_mux
asm2
alu_source2_select[1:0]
V=B"011" imm[2:0]
V=B"0000000001" jump_link_offset[9:0]
reg_read2_data[9:0]
alu_source_i
S=2'b00 I0[9:0]
V=B"0000000011", S=2'b01 I1[9:0]
V=B"0000000001", S=2'b10 I2[9:0]
S=default I3[9:0]
O[9:0]
S[1:0]
RTL_MUX
alu_source[9:0]
ALU_source2_mux


alu0
cptr0
a[9:0]
out[9:0]
Complementor
cp0
a[9:0]
b[9:0]
out[9:0]
Comparator
fa0
a[9:0]
b[9:0]
c_out
sum[9:0]
FullAdder_10Bit
c_out
n/c
mx0
ALU_op[1:0]
add_result[9:0]
comp_result[9:0]
complement_result[9:0]
shift_result[9:0]
final_answer_i
S=2'b00 I0[9:0]
S=2'b01 I1[9:0]
S=2'b10 I2[9:0]
S=2'b11 I3[9:0]
O[9:0]
S[1:0]
RTL_MUX
ALU_mux
final_answer[9:0]
f[9:0]
ALU_op[1:0]
a[9:0]
b[9:0]
sh0
a[9:0]
shift_amount[9:0]
out[9:0]
Shifter
ALU

cptr0
a[9:0]
I0[9:0]
a0_i
O[9:0]
RTL_INV
V=B"0000000001"
fa1
a[9:0]
b[9:0]
sum[9:0]
FullAdder_10Bit
out[9:0]
Complementor

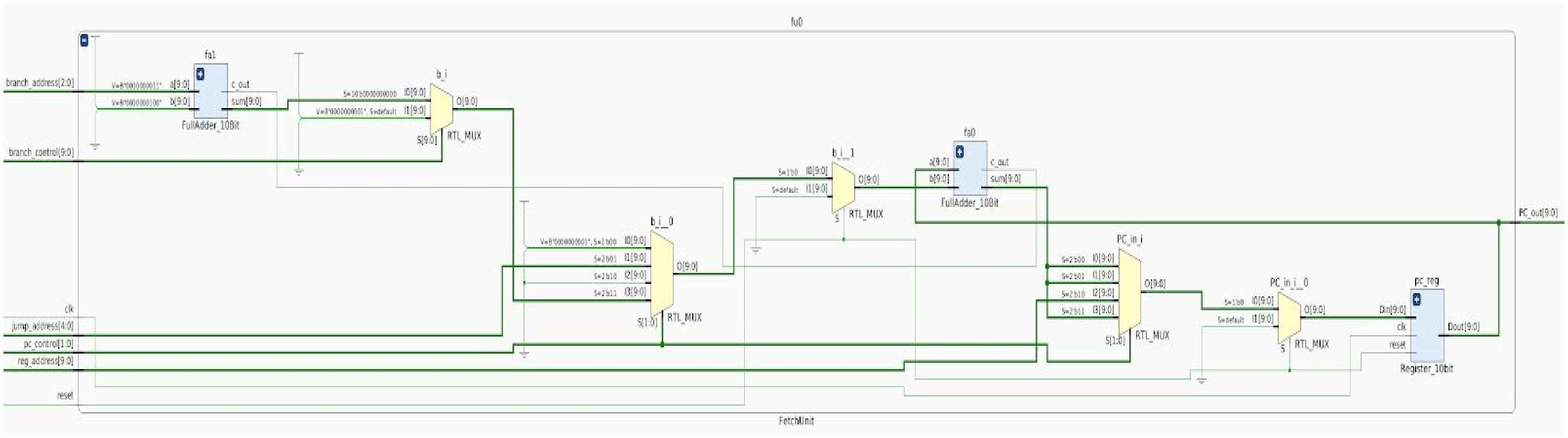

sh0
a[9:0]
shift_amount[9:0]
x0_i
I0[9:0]
O[9:0]
RTL_INV
x_i
S=1'b1
S=default
I0[9:0]
I1[9:0]
O[9:0]
S
9
RTL_MUX
V=B"0000000001"
fa2
a[9:0]
b[9:0]
sum[9:0]
FullAdder_10Bit
out0_i__0
I2
I0[9:0]
I1[9:0]
O[9:0]
RTL_LSHIFT
out0_i
RTL_RSHIFT
out_i
out[9:0]
Shifter


cp0
RTL_INV
RTL_AND
RTL_EQ
RTL_MUX
fa1
fa2
FullAdder_10Bit
Comparator

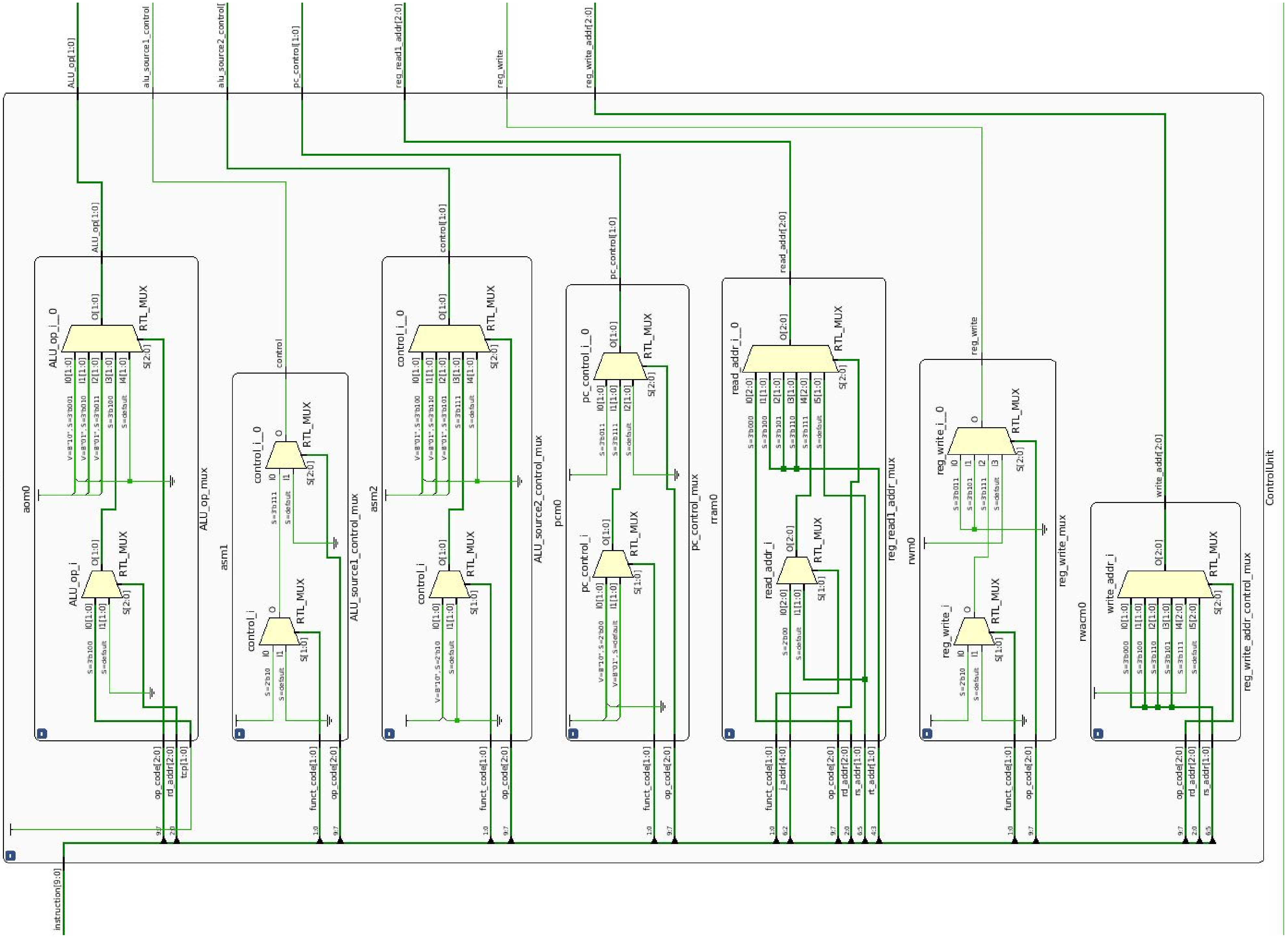
Comparator



rf
Din_reg[0]_i
Din_reg[1]_i
Din_reg[2]_i
Din_reg[3]_i
Din_reg[4]_i
Din_reg[5]_i
Din_reg[6]_i
Din_reg[7]_i
I0[9:0]
I1[9:0]
O[9:0]
S[2:0]
RTL_MUX
Din_i__0
Din_i__1
Din_i__2
Din_i__3
Din_i__4
Din_i__5
Din_i
Din_i__6
S
reg_a0
reg_sp
reg_s0
reg_t0
reg_t1
reg_v0
reg_ra
reg_zero
Din[9:0]
clk
reset
Dout[9:0]
Register_10bit
Dout_i
I2[9:0]
I3[9:0]
I4[9:0]
I5[9:0]
I6[9:0]
I7[9:0]
Dout_i__0
S[1:0]
read1_data[9:0]
read2_data[9:0]
clk
en_write
read1_addr[2:0]
read2_addr[1:0]
reset
write_addr[2:0]
write_data[9:0]
RegisterFile

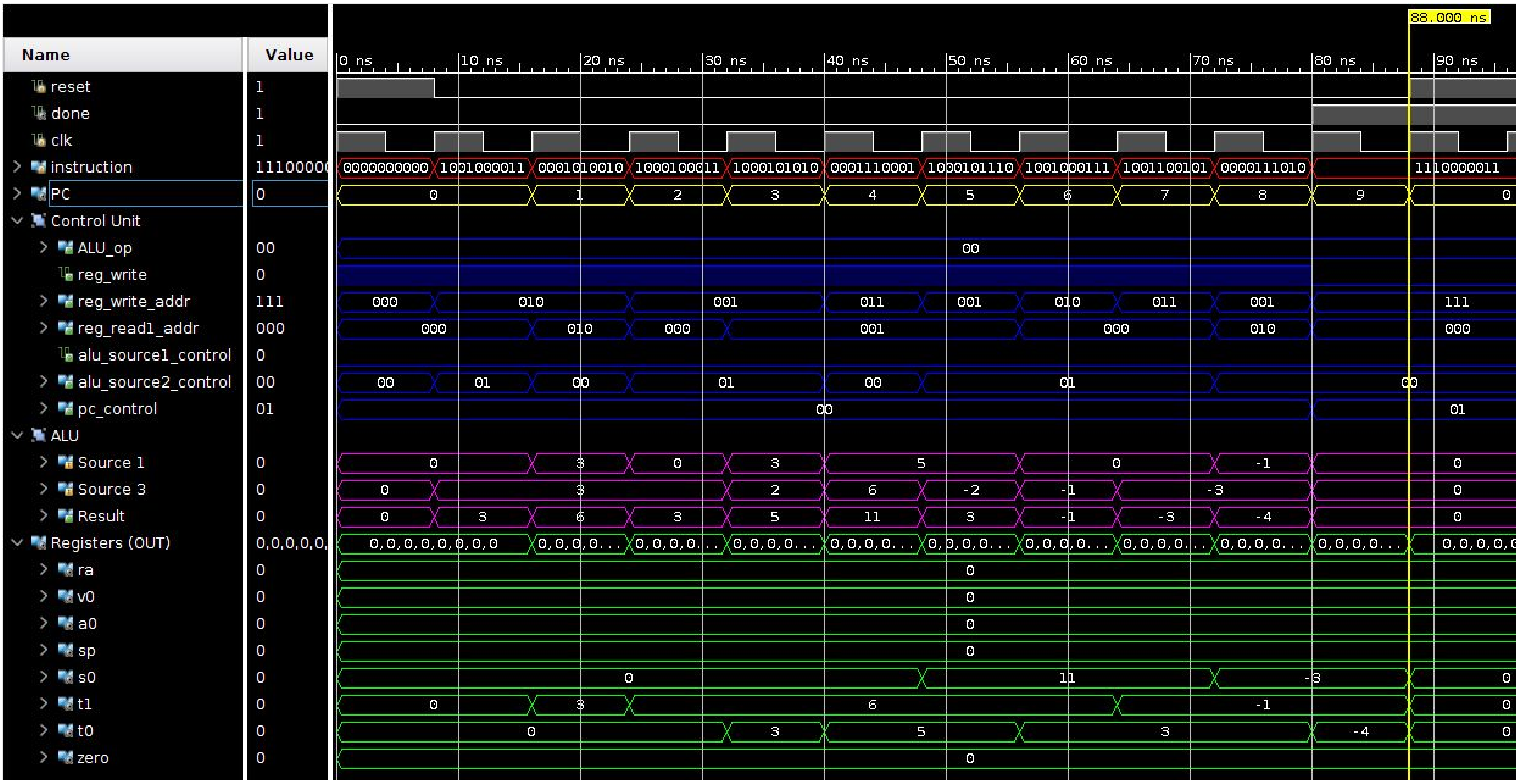

fu0
fa1
FullAdder_10Bit
a[9:0]
b[9:0]
c_out
sum[9:0]
b_i
RTL_MUX
branch_address[2:0]
branch_control[9:0]
clk
jump_address[4:0]
pc_control[1:0]
reg_address[9:0]
reset
b_i__0
b_i__1
fa0
PC_in_i
PC_in_i__0
pc_reg
Register_10bit
Din[9:0]
Dout[9:0]
PC_out[9:0]
FetchUnit

Fetch Unit


ALU_op[1:0]
alu_source1_control
alu_source2_control[
pc_control[1:0]
reg_read1_addr[2:0]
reg_write
reg_write_addr[2:0]
cu0
ControlUnit
aom0
ALU_op_mux
ALU_op_i__0
ALU_op_i
asm1
ALU_source1_control_mux
control_i__0
control_i
asm2
ALU_source2_control_mux
pcm0
pc_control_mux
pc_control_i__0
pc_control_i
rram0
reg_read1_addr_mux
read_addr_i__0
read_addr_i
read_addr[2:0]
rwm0
reg_write_mux
reg_write_i__0
reg_write_i
rwacm0
reg_write_addr_control_mux
write_addr_i
write_addr[2:0]
RTL_MUX
op_code[2:0]
rd_addr[2:0]
tcp[1:0]
funct_code[1:0]
j_addr[4:0]
rs_addr[1:0]
rt_addr[1:0]
instruction[9:0]

ADD and ADDI Instruction Test

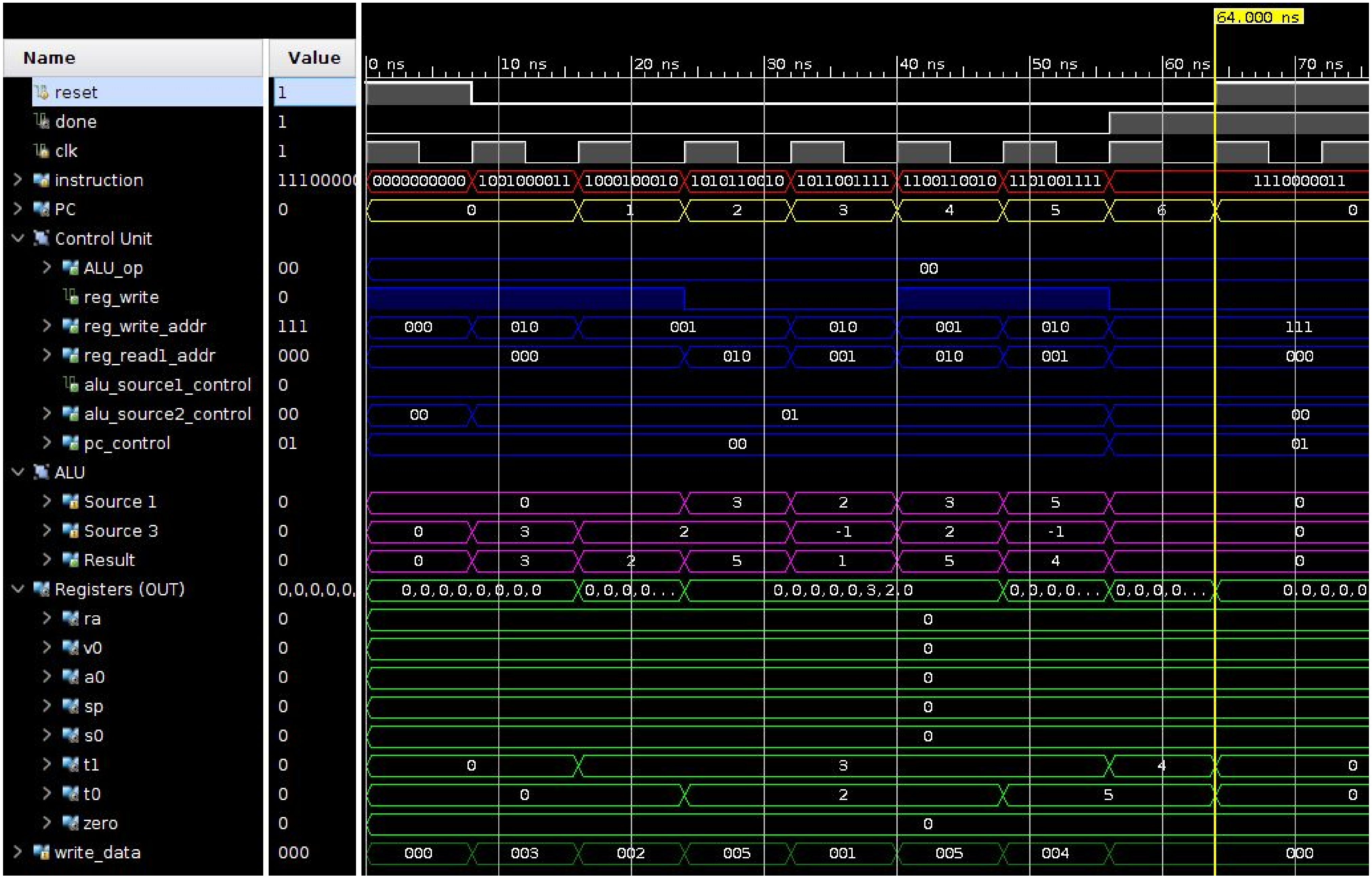
☐Clock/Reset/Done 🟥Instruction 🟨Program Counter 🟦Control Signals 🟪ALU Signals 🟩Register Contents



LW and SW Instruction Test

(Because these is no memory, lw stores the memory address in the destination register and sw attempts to do the same, but fails to write to the register because the reg_write signal disables writing.)

☐Clock/Reset/Done 🟥Instruction 🟨Program Counter 🟦Control Signals 🟪ALU Signals 🟩Register Contents

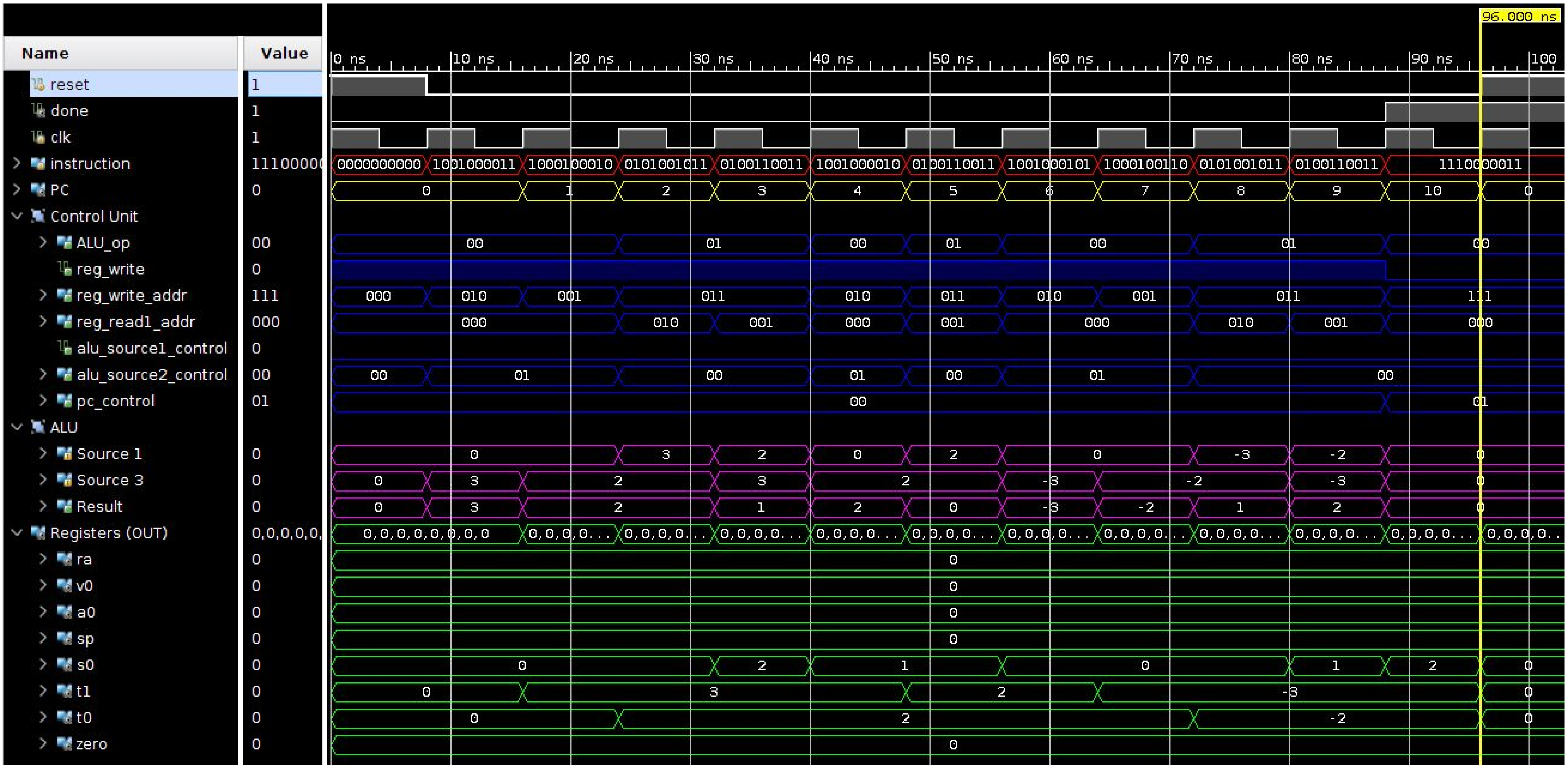


```
instruction:     0000000000 | 1001000011 | 1000100010 | 1010110010 | 1011001111 | 1100110010 | 1101001111 | 1110000011
PC:              0 | 1 | 2 | 3 | 4 | 5 | 6 | 0
ALU_op:          00
reg_write_addr:  000 | 010 | 001 | 010 | 001 | 010 | 111
reg_read1_addr:  000 | 010 | 001 | 010 | 001 | 000
alu_source2_control: 00 | 01 | 00
pc_control:      00 | 01
Source 1:        0 | 3 | 2 | 3 | 5 | 0
Source 3:        0 | 3 | 2 | -1 | 2 | -1 | 0
Result:          0 | 3 | 2 | 5 | 1 | 5 | 4 | 0
Registers (OUT): 0,0,0,0,0,0,0,0 | 0,0,0,0... | 0,0,0,0,0,3,2,0 | 0,0,0,0... | 0,0,0,0... | 0,0,0,0,0
ra:              0
v0:              0
a0:              0
sp:              0
s0:              0
t1:              0 | 3 | 4 | 0
t0:              0 | 2 | 5 | 0
zero:            0
write_data:      000 | 003 | 002 | 005 | 001 | 005 | 004 | 000
```

CMP (Compare) Instruction Test

ALU Compare Result: (1): a < b (0): a == b (2): a > b

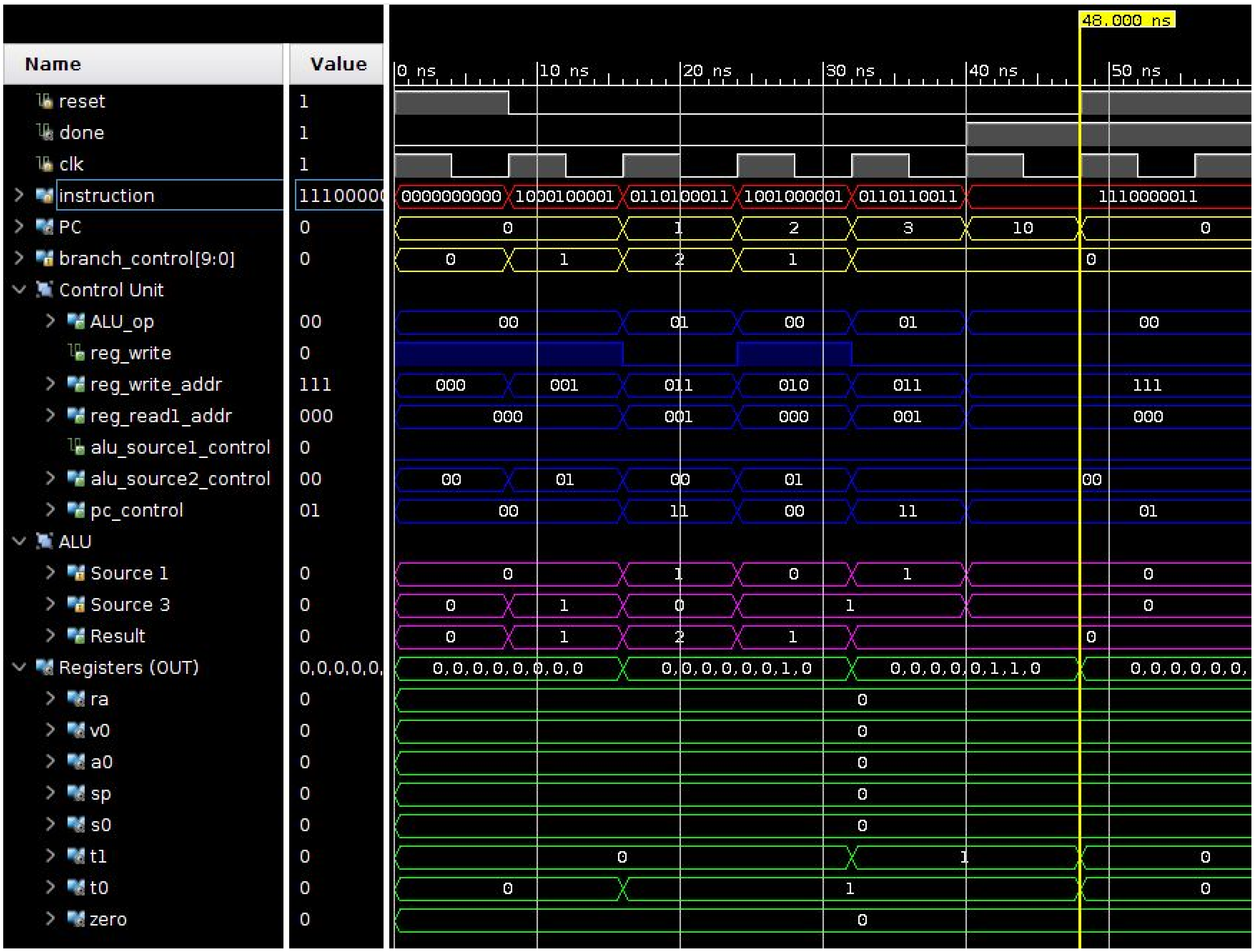
⬜Clock/Reset/Done 🟥Instruction 🟨Program Counter 🟦Control Signals 🟪ALU Signals 🟩Register Contents



Name	Value
reset	1
done	1
clk	1
instruction	11100000
PC	0
Control Unit	
ALU_op	00
reg_write	0
reg_write_addr	111
reg_read1_addr	000
alu_source1_control	0
alu_source2_control	00
pc_control	01
ALU	
Source 1	0
Source 3	0
Result	0
Registers (OUT)	0,0,0,0,0,
ra	0
v0	0
a0	0
sp	0
s0	0
t1	0
t0	0
zero	0

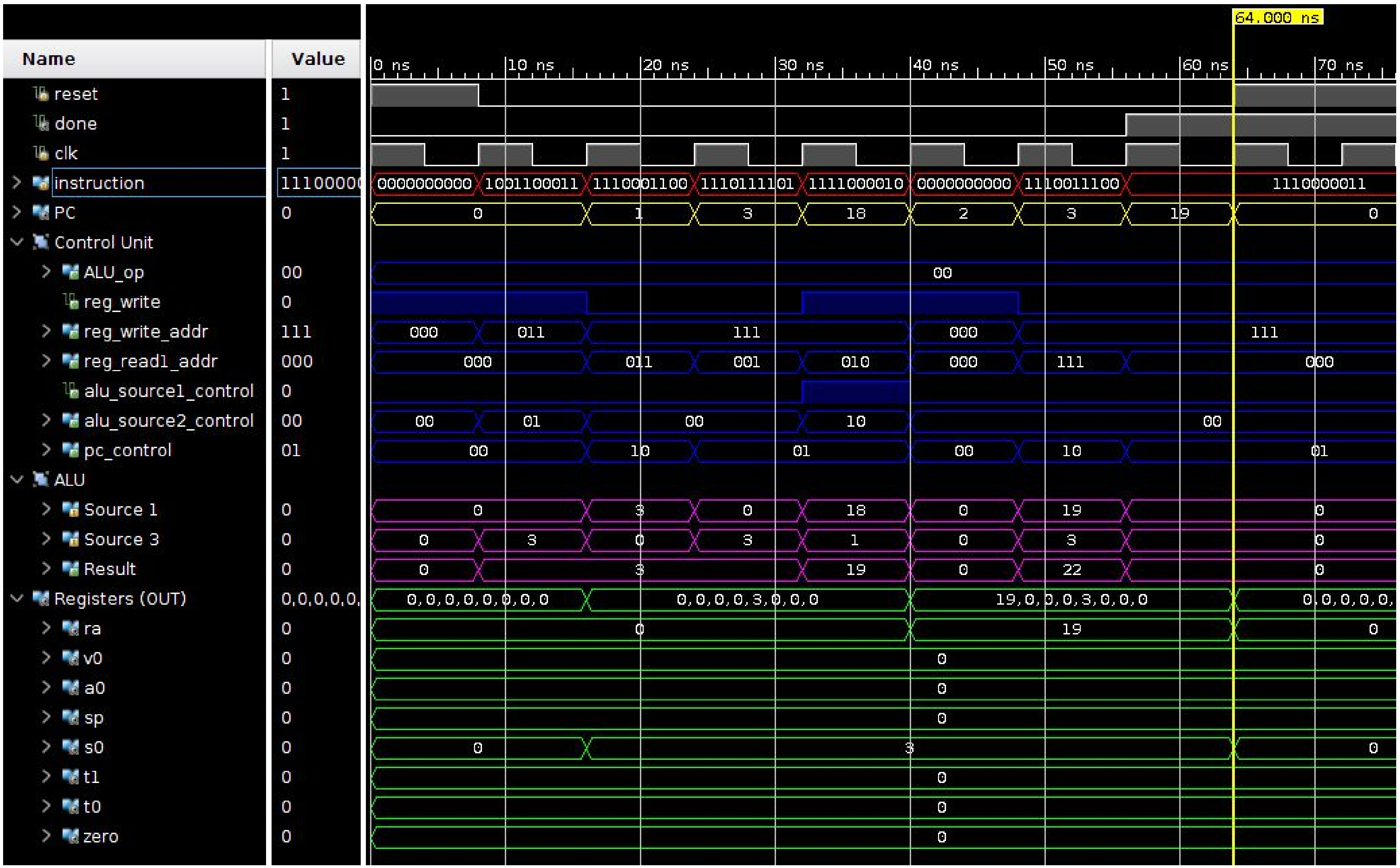
BEQ Instruction Test

⬜Clock/Reset/Done 🟥Instruction 🟨Program Counter 🟦Control Signals 🟪ALU Signals 🟩Register Contents



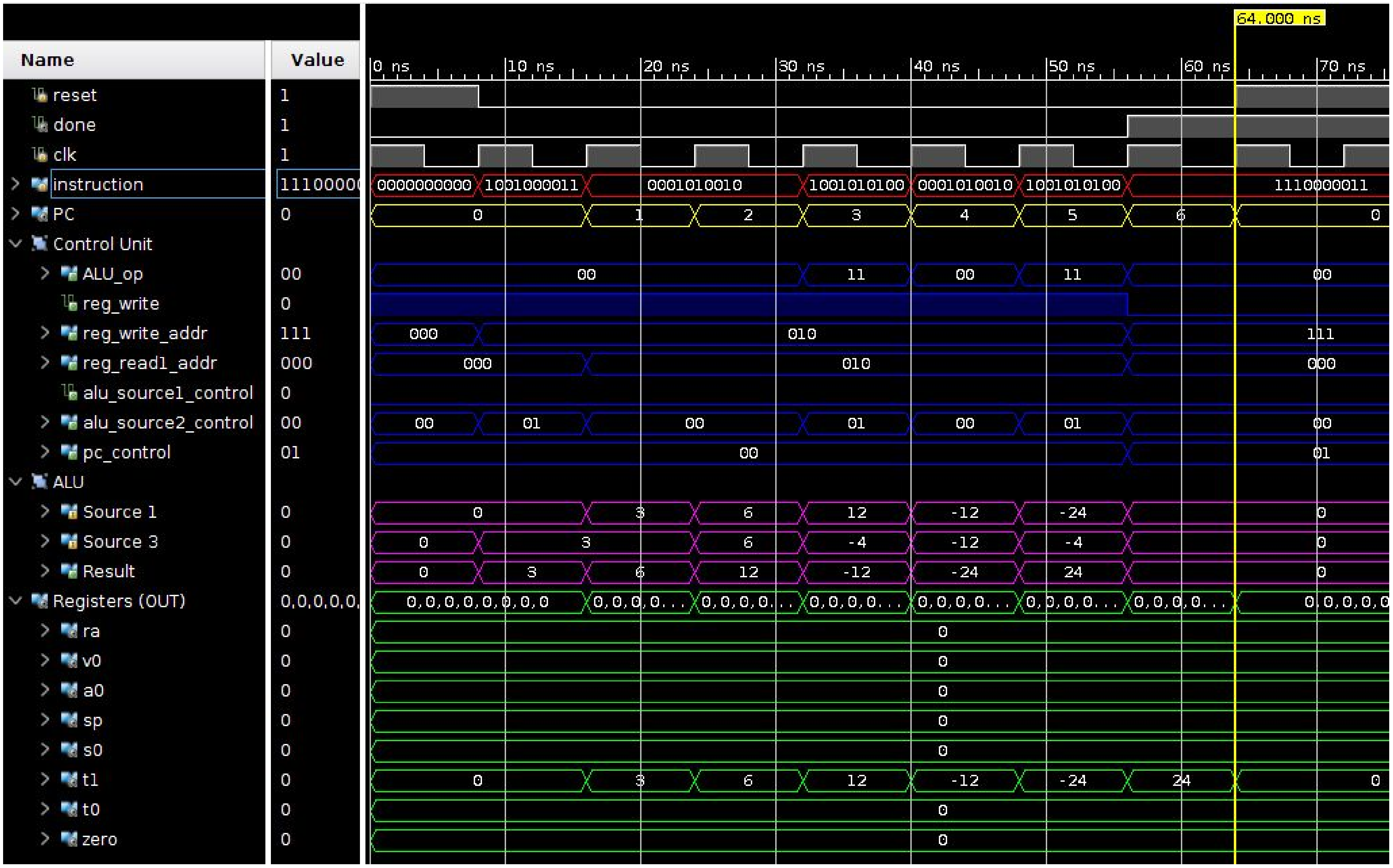
J, JR, and JAL Instruction Test

☐Clock/Reset/Done 🟥Instruction 🟨Program Counter 🟦Control Signals 🟪ALU Signals 🟩Register Contents



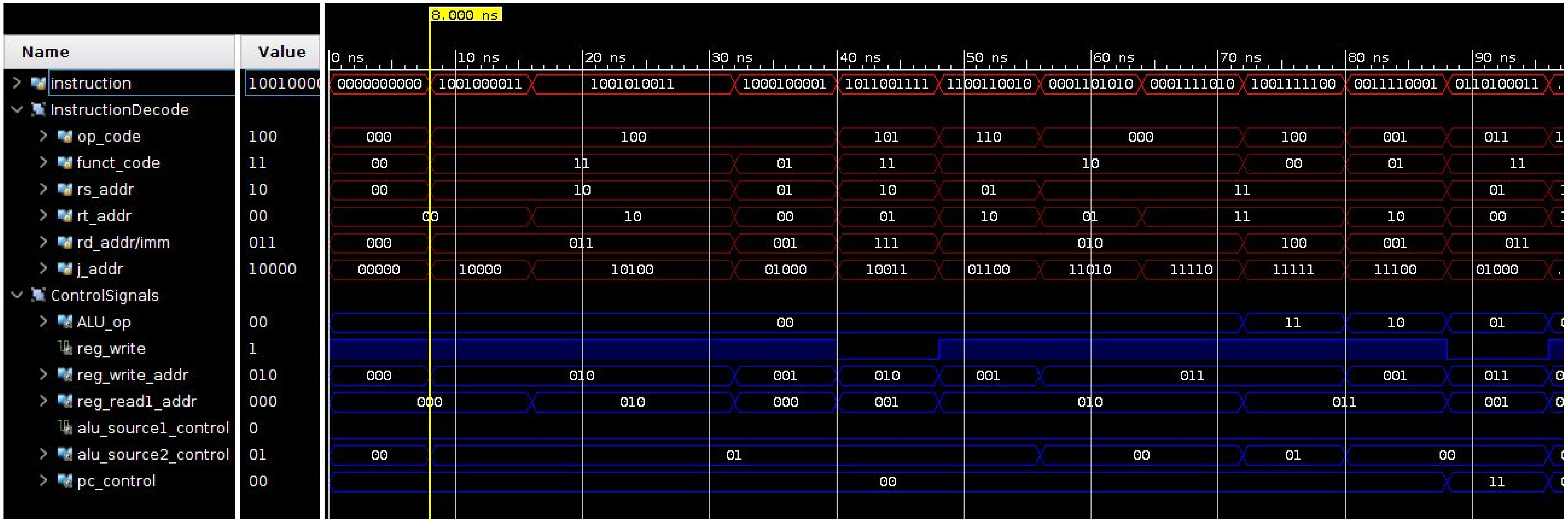
TCP (Two's Complement) Instruction Test

☐Clock/Reset/Done 🟥Instruction 🟨Program Counter 🟦Control Signals 🟪ALU Signals 🟩Register Contents



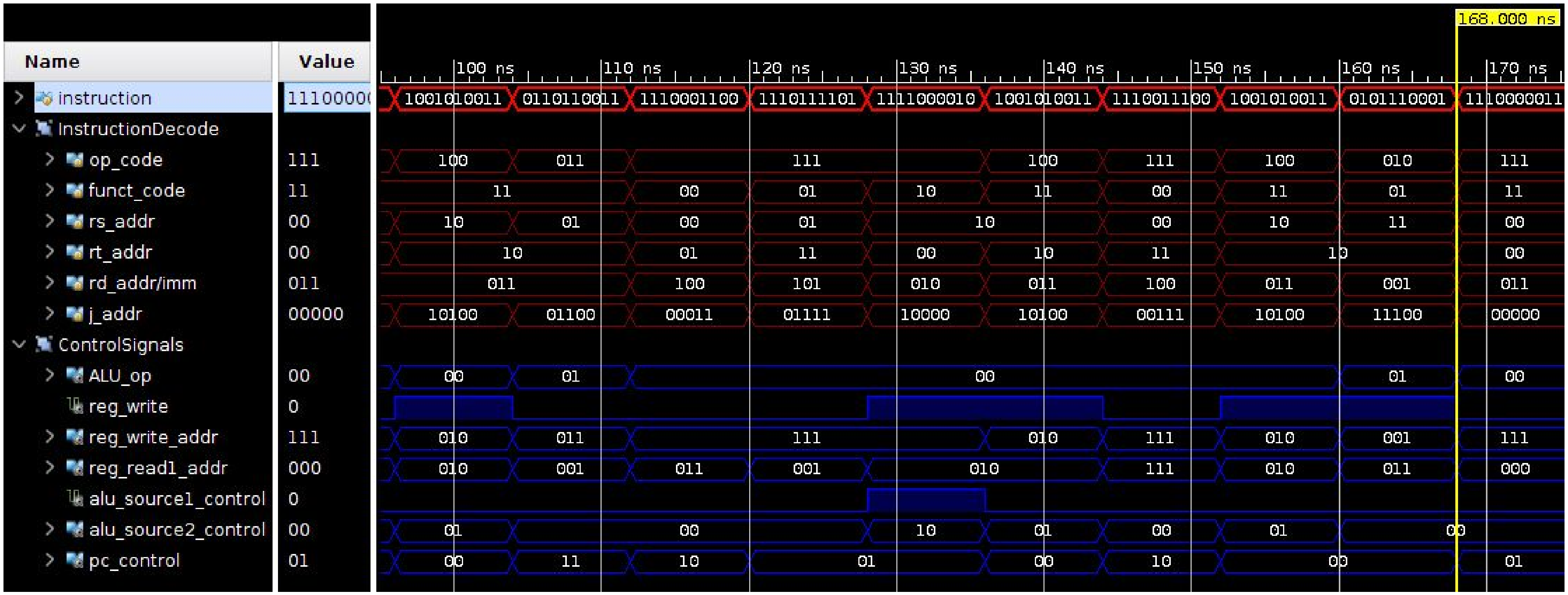
Control Unit Module Test (1 of 2)

🟥Instruction 🟥Instruction Decode 🟦Control Signals



Control Unit Module Test (2 of 2)

🟥Instruction 🟥Instruction Decode 🟦Control Signals

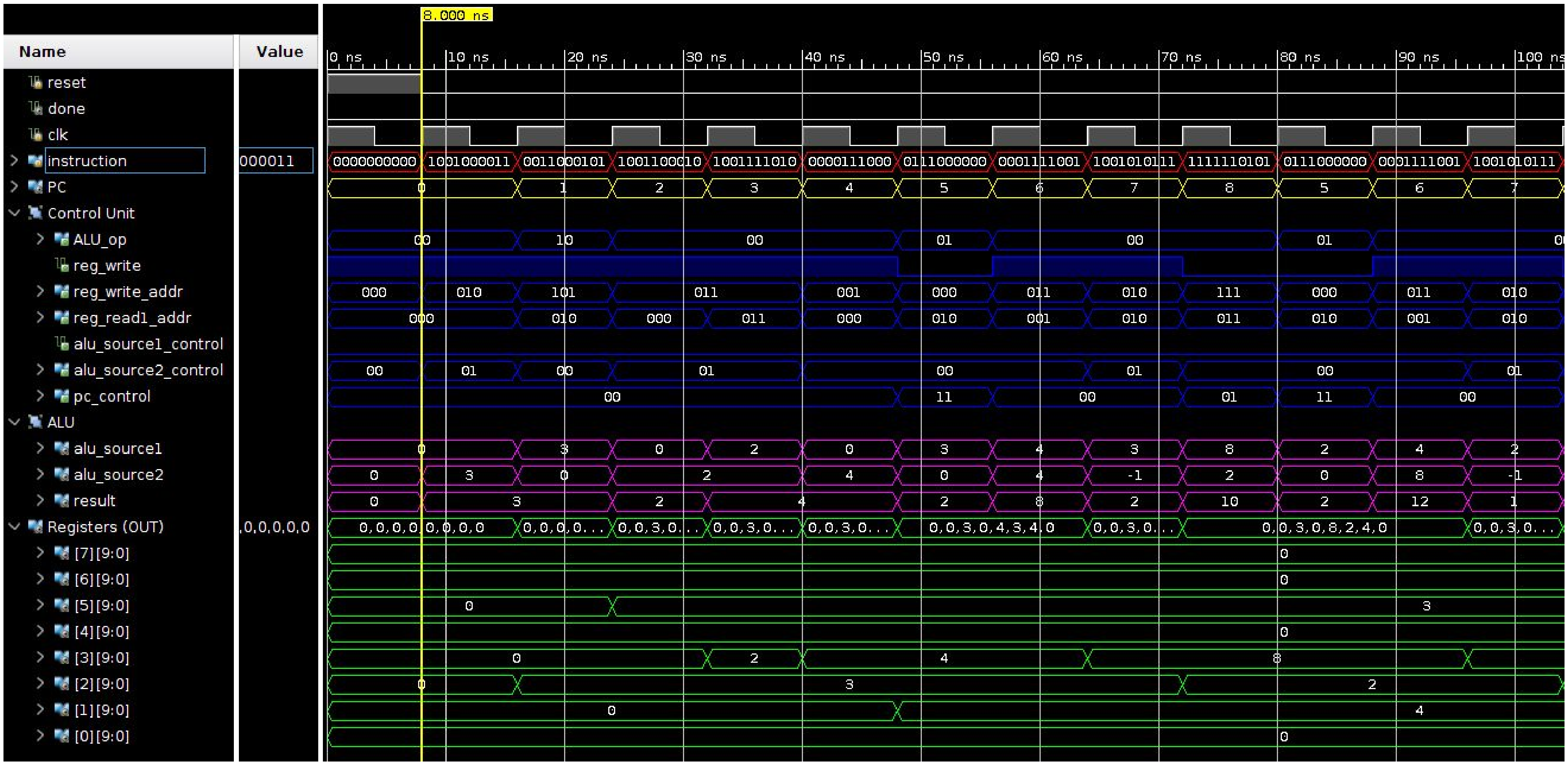


```
; Tyler Thompson
; ECE3570 Lab3a
; 2/6/2018
; Program 3 re-written for memoryless execution
; This program was modified for each test to accommodate
; different values of x and y

1001000011      addi $t1, $zero, 3      ;$t1=x=3
0011000101      sll $a0, $t1, $zero     ;store original x in $a0
1001100010      addi $s0, $zero, 2
1001111010      addi $s0, $s0, 2        ;$s0=y=4
0000111000      add $t0, $s0, $zer0     ;$t0=4

0111000000      beq $t1, $zer0, 0       ;if x==0, pc=pc+4
0001111001      add $s0, $s0, $t0       ;y=y+4
1001010111      addi $t1, $t1, -1       ;x--
1111110101      j -3                    ;pc=pc-3

0001000101      add $t1, $zero, $a0     ;load original x
0101000010      cmp $t1, $t1, $zer0
1000100001      addi $t0, $zero, 1
0110110101      beq $t0, $t1, -2        ;if x is pos, pc=pc+2
1001111100      tcp $s0, $s0            ;two comp of y
1001111110      addi $s0, $s0, -2
1001111110      addi $s0, $s0, -2       ;y=y-4
0011100110      sll $v0, $s0, $zero     ;f=(x*y)-4

1110000011      halt
```

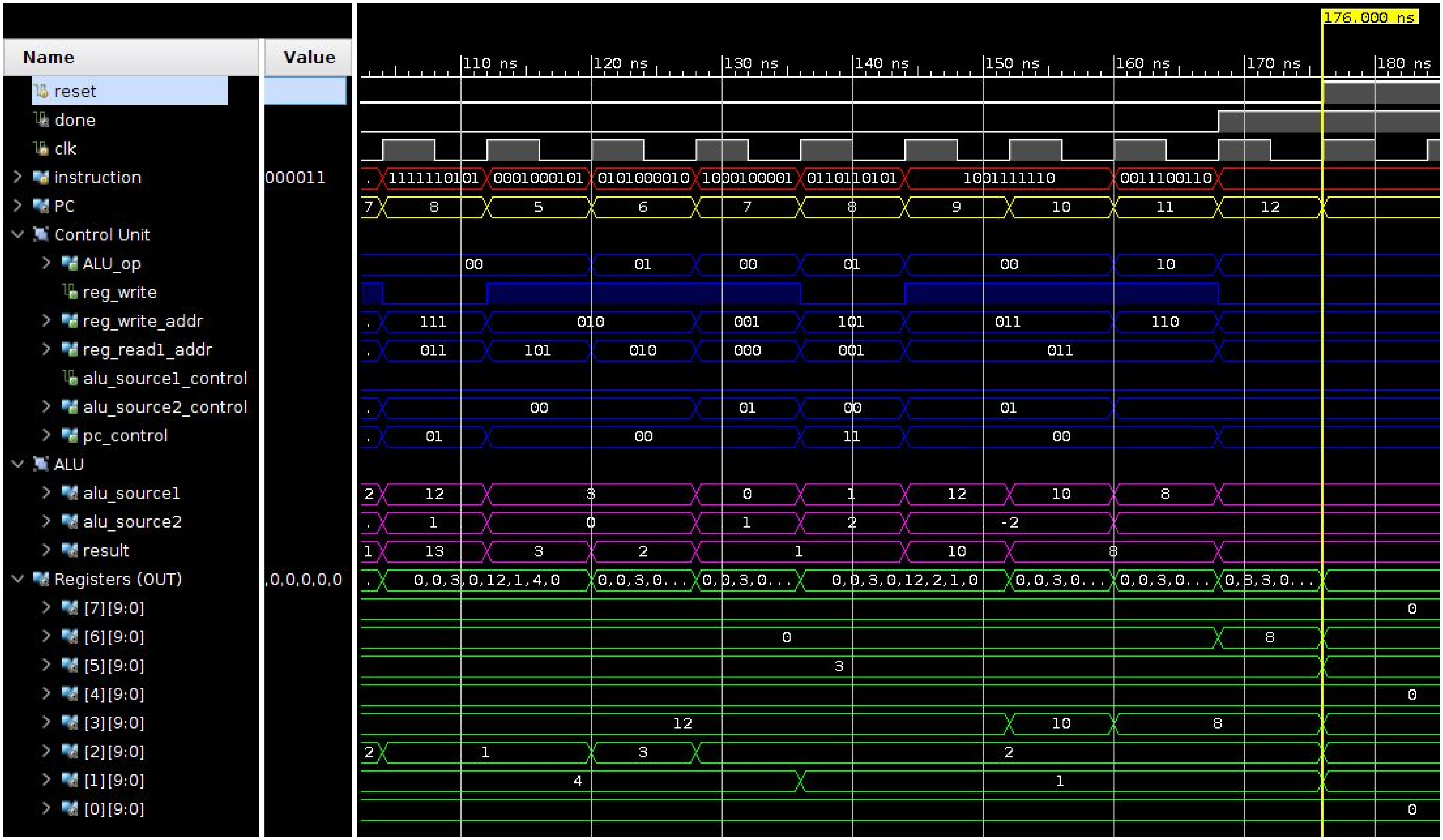
Program 3 Test Run 1 (1 of 2)

X = 3 Y = 4 F = 8

F is stored into v0 at the end of the program



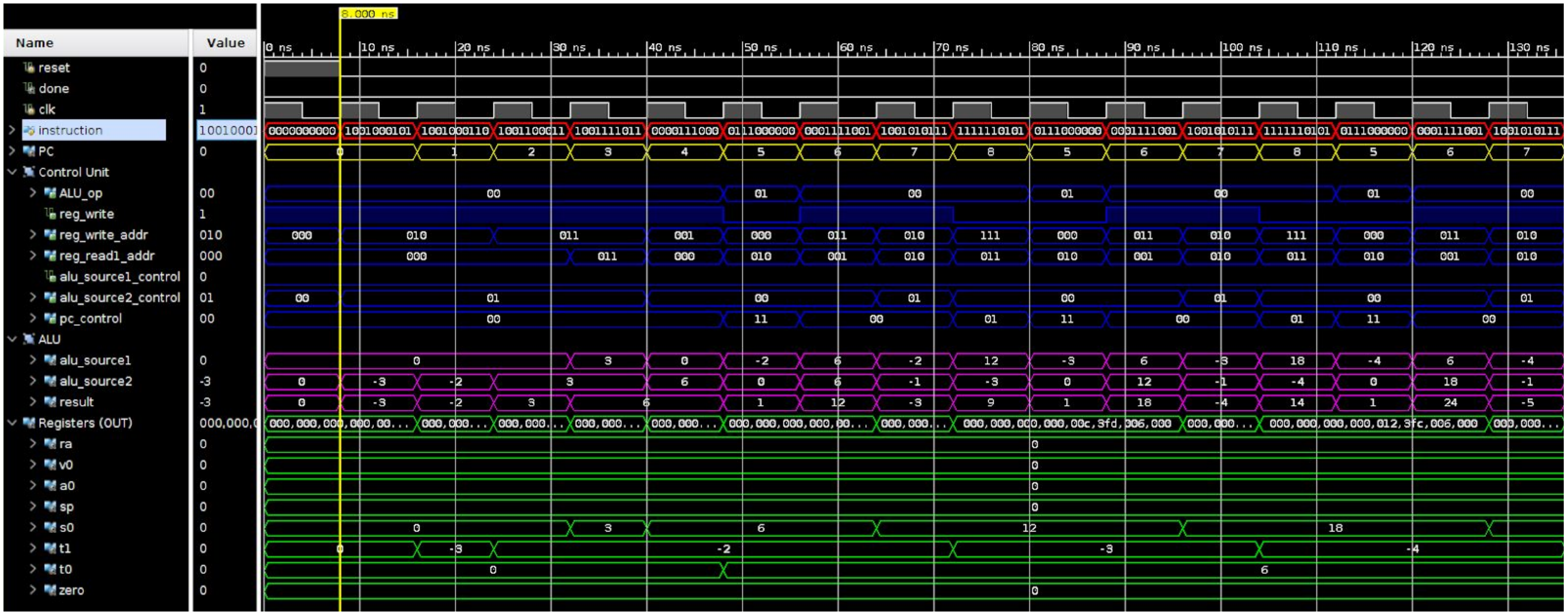
Program 3 Test Run 1 (2 of 2)



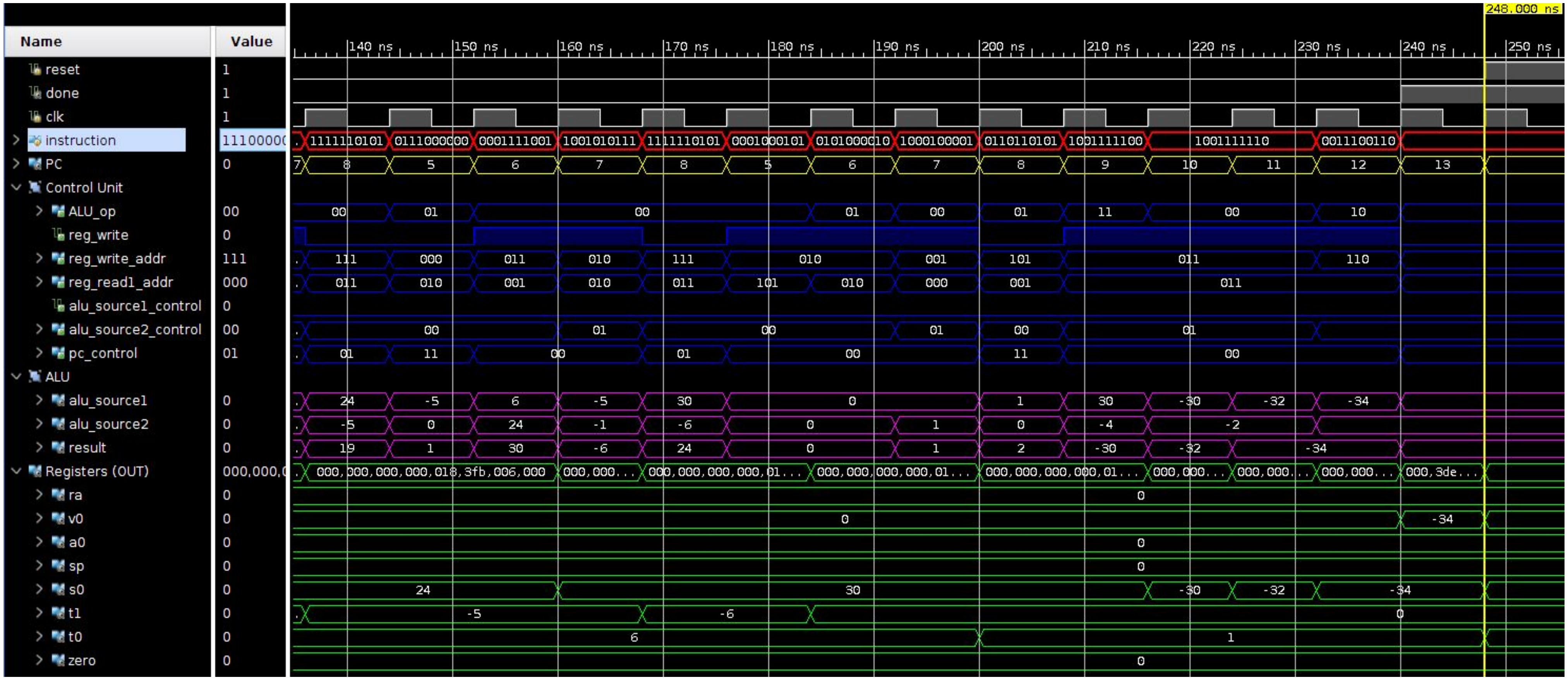
Program 3 Test Run 2 (1 of 2)

X = -5 Y = 6 F = -34

F is stored into v0 at the end of the program



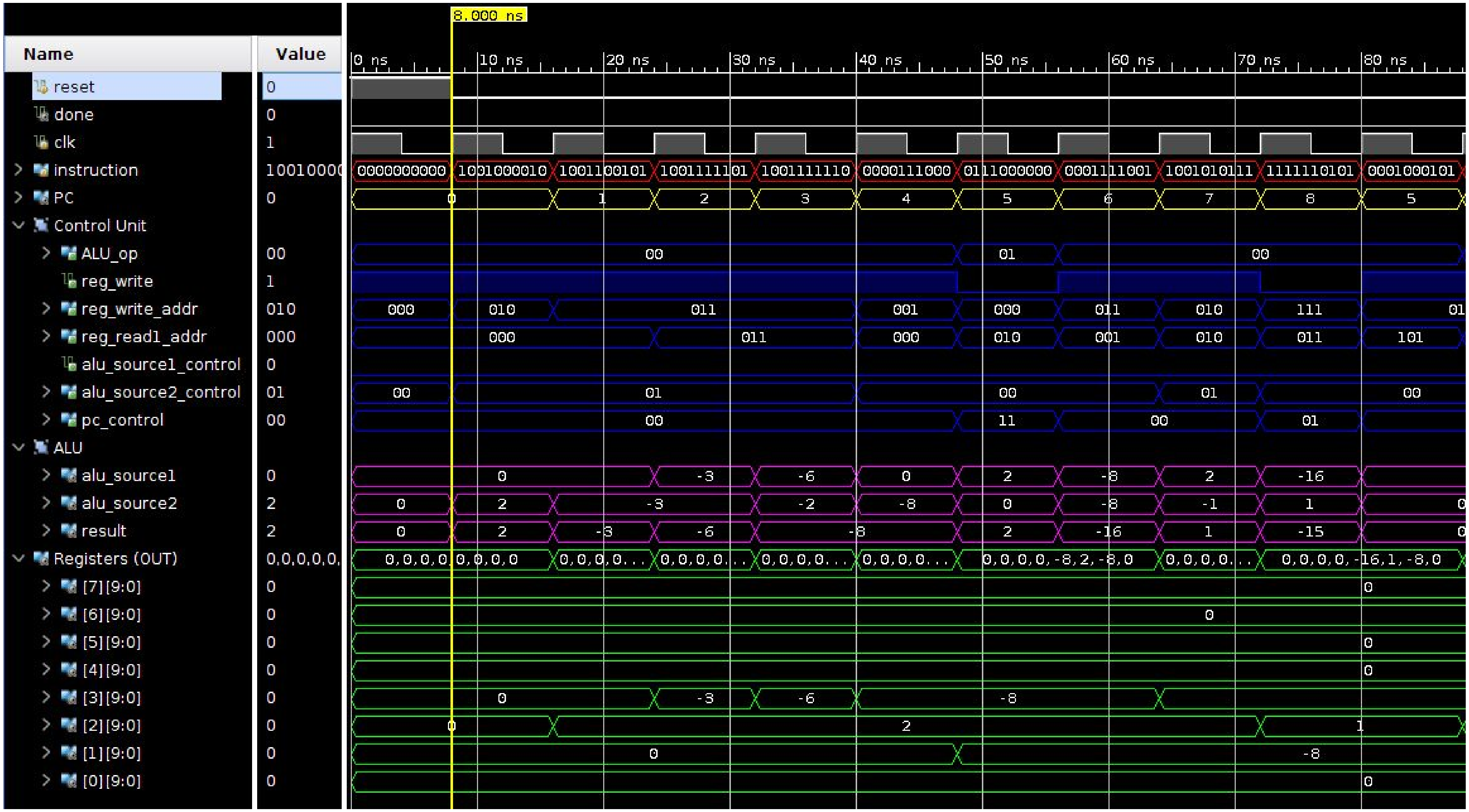
Program 3 Test Run 2 (2 of 2)



Program 3 Test Run 3 (1 of 2)

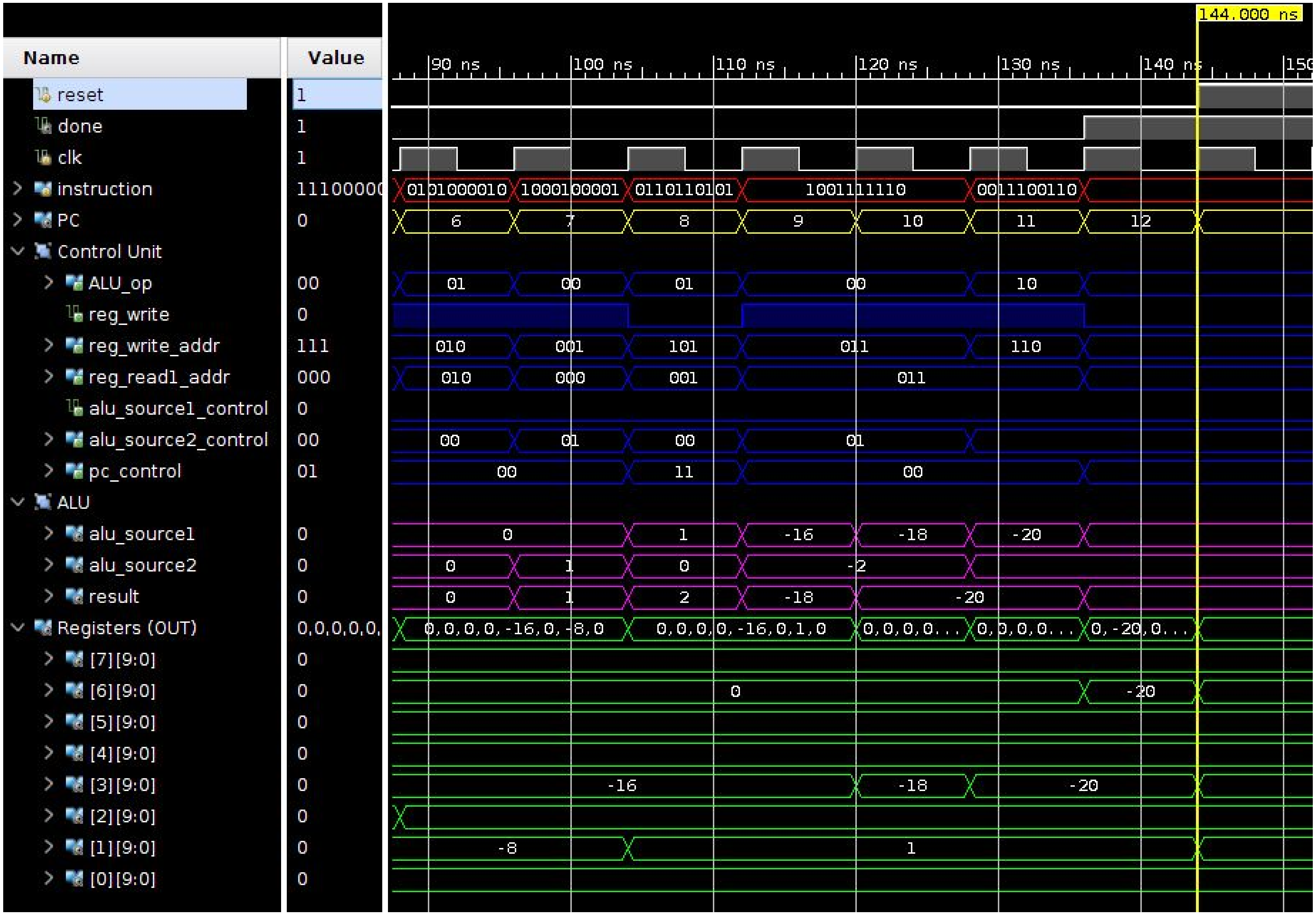
X = 2 Y = -8 F = -20

F is stored into v0 at the end of the program



Name	Value
reset	0
done	0
clk	1
instruction	10010000
PC	0
Control Unit	
ALU_op	00
reg_write	1
reg_write_addr	010
reg_read1_addr	000
alu_source1_control	0
alu_source2_control	01
pc_control	00
ALU	
alu_source1	0
alu_source2	2
result	2
Registers (OUT)	0,0,0,0,0,
[7][9:0]	0
[6][9:0]	0
[5][9:0]	0
[4][9:0]	0
[3][9:0]	0
[2][9:0]	0
[1][9:0]	0
[0][9:0]	0

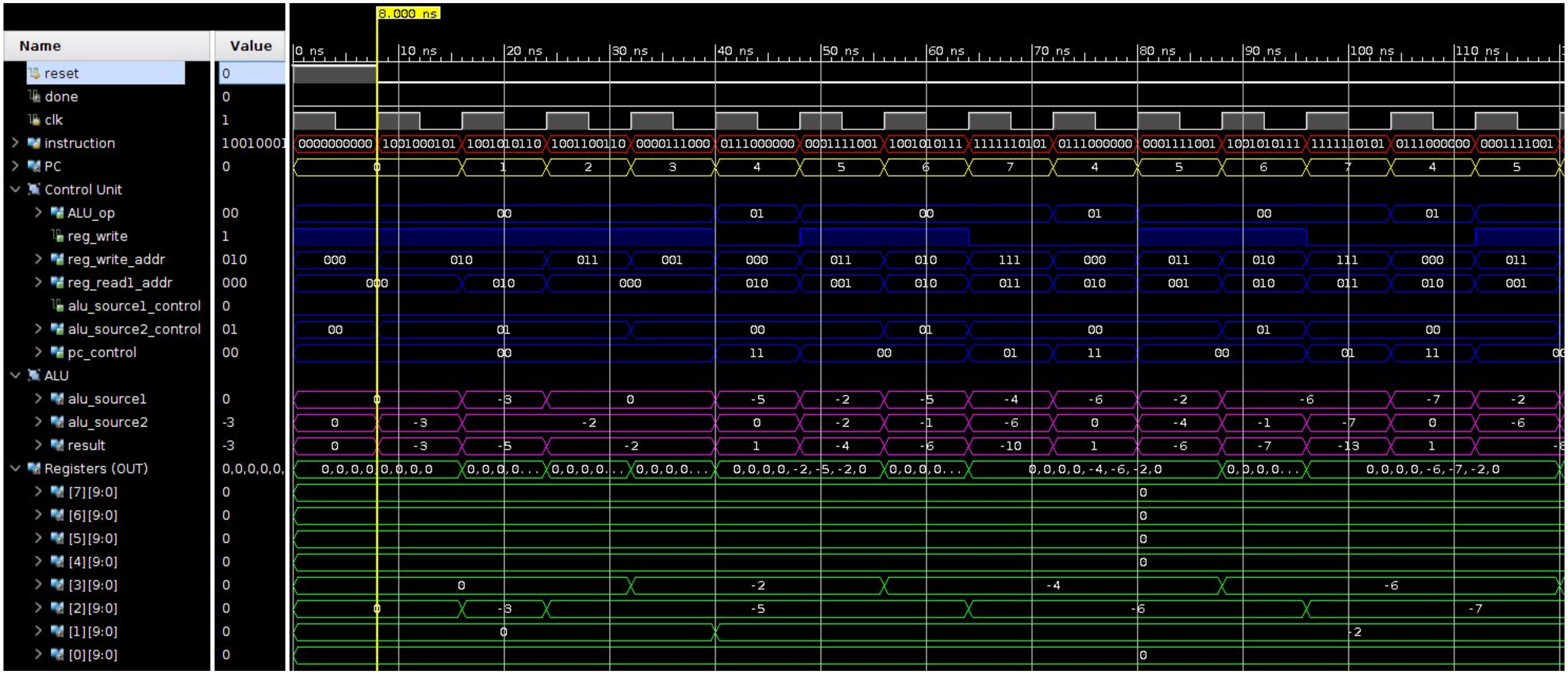
Program 3 Test Run 3 (2 of 2)



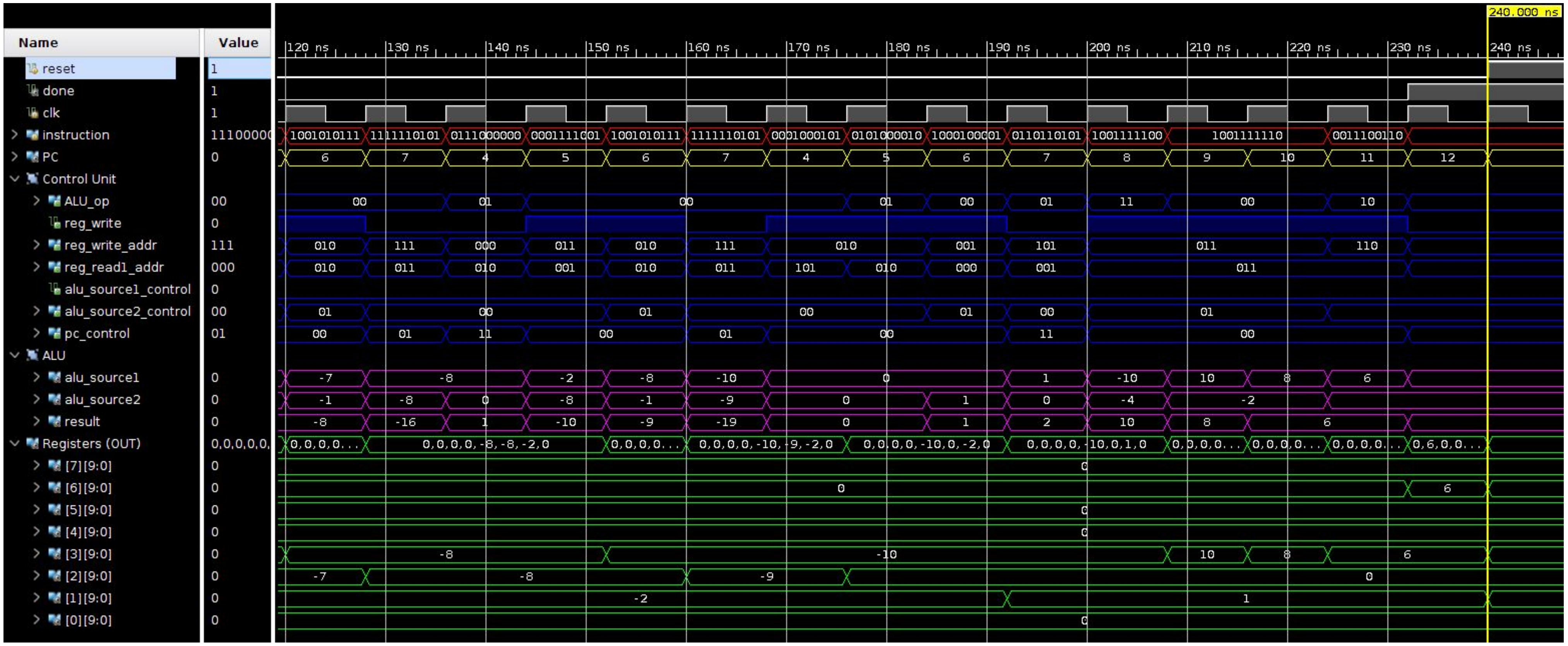
Program 3 Test Run 4 (1 of 2)

X = -5 Y = -2 F = 6

F is stored into v0 at the end of the program



Program 3 Test Run 4 (2 of 2)



```
`timescale 1ns / 1ns
`include "ALU.v"
`include "Registers.v"
`include "ControlUnit.v"
`include "FetchUnit.v"
//////////////////////////////////////////////////////////////////////////////////
// Company: Western Michigan University
// Engineer: Tyler Thompson
//
// Create Date: 02/22/2018 04:42:19 PM
// Design Name:
// Module Name: CPU
// Project Name: ECE 3570 Lab 3a
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

module CPU10Bits_jump_test();
    reg clk;
    reg reset;
    reg [9:0] instruction;
    wire done;

    CPU10Bits cpu0( .instruction(instruction), .clk(clk), .reset(reset), .done(done)
);

    //toggle clock every 4ns
    always #4 clk = ~clk;

    initial begin
        clk = 1;
        reset = 1;
        instruction = 10'b0000000000; // NOP
        #8;
        reset = 0;

        instruction = 10'b1001100011; // addi $s0, $zero, 3
        #8;
        instruction = 10'b1110001100; // jr $s0
        #8;
        instruction = 10'b1110111101; // j 01111 (15)
        #8;
        instruction = 10'b1111000010; // jal 10000 (-16)
        #8;
        instruction = 10'b0000000000; // NOP
        #8;
        instruction = 10'b1110011100; // jr $ra
        #8;
        instruction = 10'b1110000011; // halt
        #8;
        reset = 1;
    end
endmodule

module CPU10Bits_branch_test();
    reg clk;
    reg reset;
    reg [9:0] instruction;
    wire done;

    CPU10Bits cpu0( .instruction(instruction), .clk(clk), .reset(reset), .done(done)
);

    //toggle clock every 4ns
    always #4 clk = ~clk;

    initial begin
        clk = 1;
        reset = 1;
        instruction = 10'b0000000000; // NOP
        #8;
        reset = 0;

        instruction = 10'b1000100001; // addi $t0, $zero, 1
        #8;
        instruction = 10'b0110100011; // beq $t0, $zero, PC=PC+4+3
        #8;
        instruction = 10'b1001000001; // addi $t1, $zero, 1
        #8;
        instruction = 10'b0110110011; // beq $t0, $t1, PC=PC+4+3
        #8;
        instruction = 10'b1110000011; // halt
        #8;
        reset = 1;
    end
endmodule

module CPU10Bits_twos_complement_test();
    reg clk;
    reg reset;
    reg [9:0] instruction;
    wire done;

    CPU10Bits cpu0( .instruction(instruction), .clk(clk), .reset(reset), .done(done)
);

    //toggle clock every 4ns
    always #4 clk = ~clk;

    initial begin
        clk = 1;
        reset = 1;
        instruction = 10'b0000000000; // NOP
        #8;
        reset = 0;

        instruction = 10'b1001000011; // addi $t1, $zero, 3
        #8;
        instruction = 10'b0001010010;  // add $t1, $t1, $t1
        #8;
        instruction = 10'b0001010010;  // add $t1, $t1, $t1
        #8;
        instruction = 10'b1001010100; // tcp $t1, $t1
        #8;
        instruction = 10'b0001010010;  // add $t1, $t1, $t1
        #8;
        instruction = 10'b1001010100; // tcp $t1, $t1
        #8;
        instruction = 10'b1110000011; // halt
        #8;
        reset = 1;
    end
endmodule

module CPU10Bits_compare_test();
    reg clk;
    reg reset;
    reg [9:0] instruction;
    wire done;

    CPU10Bits cpu0( .instruction(instruction), .clk(clk), .reset(reset), .done(done)
);

    //toggle clock every 4ns
    always #4 clk = ~clk;

    initial begin
        clk = 1;
        reset = 1;
        instruction = 10'b0000000000; // NOP
        #8;
        reset = 0;

        instruction = 10'b1001000011; // addi $t1, $zero, 3
        #8;
        instruction = 10'b1000100010;  // addi $t0, $zero, 2
        #8;
        instruction = 10'b0101001011; // cmp $s0, $t1, $t0
        #8;
        instruction = 10'b0100110011; // cmp $s0, $t0, $t1
        #8;
        instruction = 10'b1001000010; // addi $t1, $zero, 2
        #8;
        instruction = 10'b0100110011; // cmp $s0, $t0, $t1
        #8;
        instruction = 10'b1001000101; // addi $t1, $zero, -3
        #8;
        instruction = 10'b1000100110;  // addi $t0, $zero, -2
        #8;
        instruction = 10'b0101001011; // cmp $s0, $t1, $t0
        #8;
        instruction = 10'b0100110011; // cmp $s0, $t0, $t1
        #8;
        instruction = 10'b1110000011; // halt
        #8;
        reset = 1;
    end
endmodule

module CPU10Bits_store_load_test();
    reg clk;
    reg reset;
    reg [9:0] instruction;
    wire done;

    CPU10Bits cpu0( .instruction(instruction), .clk(clk), .reset(reset), .done(done)
);

    //toggle clock every 4ns
    always #4 clk = ~clk;

    initial begin
        clk = 1;
        reset = 1;
        instruction = 10'b0000000000; // NOP
        #8;
        reset = 0;

        instruction = 10'b1001000011; // addi $t1, $zero, 3
        #8;
        instruction = 10'b1000100010;  // addi $t0, $zero, 2
        #8;
        instruction = 10'b1010110010; // sw $t0, 2($t1)
        #8;
        instruction = 10'b1011001111; // sw $t1, -1($t0)
        #8;
        instruction = 10'b1100110010; // lw $t0, 2($t1)
        #8;
        instruction = 10'b1101001111; // lw $t1, -1($t0)
        #8;
        instruction = 10'b1110000011; // halt
        #8;
        reset = 1;
    end
endmodule

module CPU10Bits_shift_test();
    reg clk;
    reg reset;
    reg [9:0] instruction;
    wire done;

    CPU10Bits cpu0( .instruction(instruction), .clk(clk), .reset(reset), .done(done)
);

    //toggle clock every 4ns
    always #4 clk = ~clk;

    initial begin
        clk = 1;
        reset = 1;
        instruction = 10'b0000000000; // NOP
        #8;
        reset = 0;

        instruction = 10'b1001000011; // addi $t1, $zero, 3
        #8;
        instruction = 10'b0001010010;  // add $t1, $t1, $t1
        #8;
        instruction = 10'b0001010010;  // add $t1, $t1, $t1
        #8;
        instruction = 10'b1000100011;  // addi $t0, $zero, 3
        #8;
        instruction = 10'b1001100101;  // addi $s0, $zero, -3
        #8;
        instruction = 10'b0011001010; // sll $t1, $t1, $t0
        #8;
        instruction = 10'b0011011010; // sll $t1, $t1, $s0
        #8;
        instruction = 10'b1110000011; // halt
        #8;
        reset = 1;
    end
endmodule

module CPU10Bits_add_addi_test();
    reg clk;
    reg reset;
    reg [9:0] instruction;
    wire done;

    CPU10Bits cpu0( .instruction(instruction), .clk(clk), .reset(reset), .done(done)
);

    //toggle clock every 4ns
    always #4 clk = ~clk;

    initial begin
        clk = 1;
        reset = 1;
        instruction = 10'b0000000000; // NOP
        #8;
        reset = 0;

        instruction = 10'b1001000011; // addi $t1, $zero, 3
        #8;
        instruction = 10'b0001010010;  // add $t1, $t1, $t1
        #8;
        instruction = 10'b1000100011;  // addi $t0, $zero, 3
        #8;
        instruction = 10'b1000101010;  // addi $t0, $t0, 2
        #8;
        instruction = 10'b0001110001;  // add $s0, $t1, $t0
        #8;
        instruction = 10'b1000101110;  // addi $t0, $t0, -2
        #8;
        instruction = 10'b1001000111;  // addi $t1, $zero, -1
        #8;
        instruction = 10'b1001100101;  // addi $s0, $zero, -3
        #8;
        instruction = 10'b0000111010;  // add $t0, $s0, $t1
        #8;
        instruction = 10'b1110000011; // halt
        #8;
        reset = 1;
    end
endmodule


// Program 3, f = (x * y) - 4
// x = 3 -> $t1, y = 4 -> $s0, f = 8 -> $v0
// TEST 1
module CPU10Bits_program_test1();
    reg clk;
    reg reset;
    reg [9:0] instruction;
    wire done;

    integer x = 3;
    integer index = 0;
    integer x_neg = 0;

    CPU10Bits cpu0( .instruction(instruction), .clk(clk), .reset(reset), .done(done)
);

    //toggle clock every 4ns
    always #4 clk = ~clk;

    initial begin
        clk = 1;
        reset = 1;
        instruction = 10'b0000000000; // NOP
        #8;
        reset = 0;
        instruction = 10'b1001000011; // addi $t1, $zero, 3; $t1 = x
        #8;
        instruction = 10'b0011000101;   //   sll $a0, $t1, $zero
        #8;
        instruction = 10'b1001100010; // addi $s0, $zero, 2; $s0 = y
        #8;
        instruction = 10'b1001111010; // addi $s0, $s0, 2;
        #8;
        instruction = 10'b0000111000; // add $t0, $s0, $zero
        #8;

        for ( index = 0; index < (x - 1); index = index + 1 ) begin
            instruction = 10'b0111000000; //beq $t1, $zer0, 0
            #8;
            instruction = 10'b0001111001; //add $s0, $s0, $t0
            #8;
            instruction = 10'b1001010111; //addi $t1, $t1, -1
            #8;
            instruction = 10'b1111110101; //j 11101 (-3)
            #8;
        end
        instruction = 10'b0001000101;    //  add $t1, $zero, $a0     ;load original x
        #8;
        instruction = 10'b0101000010;    //  cmp $t1, $t1, $zer0
        #8;
        instruction = 10'b1000100001;    //  addi $t0, $zero, 1
        #8;
        instruction = 10'b0110110101;    //  beq $t0, $t1, -2        ;if x is pos,
pc=pc+2
        #8;
        if ( x_neg == 1 ) begin
            instruction = 10'b1001111100;    //  tcp $s0, $s0            ;two comp of
y
            #8;
        end
        instruction = 10'b1001111110; // addi $s0, $s0, -2;
        #8;
        instruction = 10'b1001111110; // addi $s0, $s0, -2;
        #8;
        instruction = 10'b0011100110; // sll $v0, $s0, $zer0
        #8;
        instruction = 10'b1110000011; // halt
        #8;
        reset = 1;


    end
endmodule

// Program 3, f = (x * y) - 4
// x = -5 -> $t1, y = 6 -> $s0, f = -34 -> $v0
// TEST 2
module CPU10Bits_program_test2();
   reg clk;
    reg reset;
    reg [9:0] instruction;
    wire done;

    integer x = 5;
    integer index = 0;
    integer x_neg = 1;
    CPU10Bits cpu0( .instruction(instruction), .clk(clk), .reset(reset), .done(done)
);

    //toggle clock every 4ns
    always #4 clk = ~clk;

    initial begin
        clk = 1;
        reset = 1;
        instruction = 10'b0000000000; // NOP
        #8;
        reset = 0;
        instruction = 10'b1001000101; // addi $t1, $zero, -3; $t1 = x
        #8;
        instruction = 10'b1001010110; // addi $t1, $t1, -2; $t1 = x = -5
        #8;
        instruction = 10'b1001100011; // addi $s0, $zero, 3; $s0 = y =6
        #8;
        instruction = 10'b1001111011; // addi $s0, $s0, 3;
        #8;
        instruction = 10'b0000111000; // add $t0, $s0, $zero
        #8;

        for ( index = 0; index < (x - 1); index = index + 1 ) begin
            instruction = 10'b0111000000; //beq $t1, $zer0, 0
            #8;
            instruction = 10'b0001111001; //add $s0, $s0, $t0
            #8;
            instruction = 10'b1001010111; //addi $t1, $t1, -1
            #8;
            instruction = 10'b1111110101; //j 11101 (-3)
            #8;
        end
        instruction = 10'b0001000101;    //  add $t1, $zero, $a0     ;load original x
        #8;
        instruction = 10'b0101000010;    //  cmp $t1, $t1, $zer0
        #8;
        instruction = 10'b1000100001;    //  addi $t0, $zero, 1
        #8;
        instruction = 10'b0110110101;    //  beq $t0, $t1, -2        ;if x is pos,
pc=pc+2
        #8;
        if ( x_neg == 1 ) begin
            instruction = 10'b1001111100;    //  tcp $s0, $s0            ;two comp of
y
            #8;
        end
        instruction = 10'b1001111110; // addi $s0, $s0, -2;
        #8;
        instruction = 10'b1001111110; // addi $s0, $s0, -2;
        #8;
        instruction = 10'b0011100110; // sll $v0, $s0, $zer0
        #8;
        instruction = 10'b1110000011; // halt
        #8;
        reset = 1;


    end

endmodule

// Program 3, f = (x * y) - 4
// x = 2 -> $t1, y = -8 -> $s0, f = -20 -> $v0
// TEST 3
module CPU10Bits_program_test3();
    reg clk;
    reg reset;
    reg [9:0] instruction;
    wire done;

    integer x = 2;
    integer index = 0;
    integer x_neg = 0;

    CPU10Bits cpu0( .instruction(instruction), .clk(clk), .reset(reset), .done(done)
);

    //toggle clock every 4ns
    always #4 clk = ~clk;

    initial begin
        clk = 1;
        reset = 1;
        instruction = 10'b0000000000; // NOP
        #8;
        reset = 0;
        instruction = 10'b1001000010; // addi $t1, $zero, 2; $t1 = x
        #8;
        instruction = 10'b1001100101; // addi $s0, $zero, -3; $s0 = y
        #8;
        instruction = 10'b1001111101; // addi $s0, $s0, -3;
        #8;
        instruction = 10'b1001111110; // addi $s0, $s0, -2;
        #8;
        instruction = 10'b0000111000; // add $t0, $s0, $zero
        #8;

        for ( index = 0; index < (x - 1); index = index + 1 ) begin
            instruction = 10'b0111000000; //beq $t1, $zer0, 0
            #8;
            instruction = 10'b0001111001; //add $s0, $s0, $t0
            #8;
            instruction = 10'b1001010111; //addi $t1, $t1, -1
            #8;
            instruction = 10'b1111110101; //j 11101 (-3)
            #8;
        end
        instruction = 10'b0001000101;    //  add $t1, $zero, $a0     ;load original x
        #8;
        instruction = 10'b0101000010;    //  cmp $t1, $t1, $zer0
        #8;
        instruction = 10'b1000100001;    //  addi $t0, $zero, 1
        #8;
        instruction = 10'b0110110101;    //  beq $t0, $t1, -2        ;if x is pos,
pc=pc+2
        #8;
        if ( x_neg == 1 ) begin
            instruction = 10'b1001111100;    //  tcp $s0, $s0            ;two comp of
y
            #8;
        end
        instruction = 10'b1001111110; // addi $s0, $s0, -2;
        #8;
        instruction = 10'b1001111110; // addi $s0, $s0, -2;
        #8;
        instruction = 10'b0011100110; // sll $v0, $s0, $zer0
        #8;
        instruction = 10'b1110000011; // halt
        #8;
        reset = 1;


    end
endmodule

// Program 3, f = (x * y) - 4
// x = -5 -> $t1, y = -2 -> $s0, f = 6 -> $v0
// TEST 4
module CPU10Bits_program_test4();
   reg clk;
    reg reset;
    reg [9:0] instruction;
    wire done;

    integer x = 5;
    integer index = 0;
    integer x_neg = 1;

    CPU10Bits cpu0( .instruction(instruction), .clk(clk), .reset(reset), .done(done)
);

    //toggle clock every 4ns
    always #4 clk = ~clk;

    initial begin
        clk = 1;
        reset = 1;
        instruction = 10'b0000000000; // NOP
        #8;
        reset = 0;
        instruction = 10'b1001000101; // addi $t1, $zero, -3; $t1 = x
        #8;
        instruction = 10'b1001010110; // addi $t1, $t1, -2; $t1 = x=-5
        #8;
        instruction = 10'b1001100110; // addi $s0, $zero, -2; $s0 = y
        #8;
        instruction = 10'b0000111000; // add $t0, $s0, $zero
        #8;

        for ( index = 0; index < (x - 1); index = index + 1 ) begin
            instruction = 10'b0111000000; //beq $t1, $zer0, 0
            #8;
            instruction = 10'b0001111001; //add $s0, $s0, $t0
            #8;
            instruction = 10'b1001010111; //addi $t1, $t1, -1
            #8;
            instruction = 10'b1111110101; //j 11101 (-3)
            #8;
        end
        instruction = 10'b0001000101;    //  add $t1, $zero, $a0     ;load original x
        #8;
        instruction = 10'b0101000010;    //  cmp $t1, $t1, $zer0
        #8;
        instruction = 10'b1000100001;    //  addi $t0, $zero, 1
        #8;
        instruction = 10'b0110110101;    //  beq $t0, $t1, -2        ;if x is pos,
pc=pc+2
        #8;
        if ( x_neg == 1 ) begin
            instruction = 10'b1001111100;    //  tcp $s0, $s0            ;two comp of
y
```

```
            #8;
        end
        instruction = 10'b1001111110; // addi $s0, $s0, -2;
        #8;
        instruction = 10'b1001111110; // addi $s0, $s0, -2;
        #8;
        instruction = 10'b0011100110; // sll $v0, $s0, $zer0
        #8;
        instruction = 10'b1110000011; // halt
        #8;
        reset = 1;


    end
endmodule

module CPU10Bits(
    input wire [9:0] instruction,
    input wire clk,
    input wire reset,
    output reg done
    );

    wire [1:0]ALU_op;
    wire reg_write;
    wire [2:0] reg_write_addr; // destination select
    wire [9:0]alu_source1; // ALU input a
    wire [9:0]alu_source2; // ALU input b
    wire alu_source1_control;
    wire [1:0]alu_source2_control;
    wire [1:0]pc_control;
    wire [9:0]PC;
    wire c_out;
    wire [9:0]result;
    wire [2:0]reg_read1_addr; // source select
    wire [9:0]read1_data;
    wire [9:0]read2_data;

    ControlUnit cu0( .instruction(instruction), .ALU_op(ALU_op), .reg_write
(reg_write), .reg_write_addr(reg_write_addr), .reg_read1_addr
(reg_read1_addr), .alu_source1_control(alu_source1_control), .alu_source2_control
(alu_source2_control), .pc_control(pc_control) );

    FetchUnit fu0( .clk(clk), .reset(reset), .pc_control(pc_control), .branch_control
(result), .jump_address(instruction[6:2]), .branch_address(instruction
[2:0]), .reg_address(read1_data), .PC_out(PC) );

    RegisterFile rf( .clk(clk), .reset(reset), .en_write(reg_write), .write_addr
(reg_write_addr), .write_data(result), .read1_addr(reg_read1_addr), .read2_addr
(instruction[4:3]), .read1_data(read1_data), .read2_data(read2_data) );

    ALU alu0( .ALU_op(ALU_op), .a(alu_source1), .b(alu_source2), .f(result), .c_out
(c_out) );

    ALU_source1_mux asm1( .alu_source1_select(alu_source1_control), .reg_read1_data
(read1_data), .pc(PC), .alu_source(alu_source1) );

    ALU_source2_mux asm2( .alu_source2_select(alu_source2_control), .reg_read2_data
(read2_data), .imm(instruction[2:0]), .jump_link_offset(10'b0000000001), .alu_source
(alu_source2) );

    always@(*) begin
        // halt
       if ( instruction[9:7] == 3'b111 && instruction[1:0] == 2'b11 ) begin
```

```
                done <= 1'b1;
            end
            else begin
                done <= 1'b0;
            end
        end

    endmodule

    // ALU Source 1 Mux
    // alu_source1_select:
    //  0 -> reg_read1_data
    //  1 -> PC
    module ALU_source1_mux(
        input wire alu_source1_select,
        input wire [9:0]reg_read1_data,
        input wire [9:0]pc,
        output reg [9:0]alu_source
        );
        always@(*)begin
            case(alu_source1_select)
                1'b1: alu_source <= pc;
                1'b0: alu_source <= reg_read1_data;
                default: alu_source <= reg_read1_data;
            endcase
        end
    endmodule

    // ALU Source 2 Mux
    // alu_source2_select:
    //  00 -> reg_read2_data
    //  01 -> imm - sign extend
    //  10 -> jump_link_offset
    module ALU_source2_mux(
        input wire [1:0]alu_source2_select,
        input wire [9:0]reg_read2_data,
        input wire [2:0]imm,
        input wire [9:0]jump_link_offset,
        output reg [9:0]alu_source
        );
        always@(*)begin
            case(alu_source2_select)
                2'b00: alu_source <= reg_read2_data;
                2'b01: alu_source <= { {7{imm[2]}}, imm[2:0] }; //sign extend imm
                2'b10: alu_source <= jump_link_offset;
                default: alu_source <= reg_read2_data;
            endcase
        end
    endmodule
```

```
`timescale 1ns / 1ns
//////////////////////////////////////////////////////////////////////////////////
// Company: Western Michigan University
// Engineer: Tyler Thompson
//
// Create Date: 02/22/2018 03:55:32 PM
// Design Name:
// Module Name: ControlUnit
// Project Name: ECE 3570 Lab 3a
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

module ControlUnitTest();
   reg [9:0] instruction;
   wire [1:0]ALU_op;
   wire reg_write;
   wire [2:0] reg_write_addr;
   wire [2:0] reg_read1_addr;
   wire alu_source1_control;
   wire [1:0]alu_source2_control;
   wire [1:0]pc_control;

    ControlUnit cu0( .instruction(instruction), .ALU_op(ALU_op), .reg_write
(reg_write), .reg_write_addr(reg_write_addr), .reg_read1_addr
(reg_read1_addr), .alu_source1_control(alu_source1_control), .alu_source2_control
(alu_source2_control), .pc_control(pc_control) );

    initial begin
         instruction = 10'b0000000000;
           #8;

           instruction = 10'b1001000011; // addi $t1, $zero, 3
           #8;
           instruction = 10'b1001010011; // addi $t1, $t1, 3
           #8;
           instruction = 10'b1001010011; // addi $t1, $t1, 3
           #8;
           instruction = 10'b1000100001; // addi $t0, $zero, 1
           #8;
           instruction = 10'b1011001111; // sw $t1, -1($t0)
           #8;
           instruction = 10'b1100110010; // lw $t0, 2($t1)
           #8;
           instruction = 10'b0001101010; // add $s0, $t0, $t1
           #8;
           instruction = 10'b0001111010; // add $s0, $s0, $t1
           #8;
           instruction = 10'b1001111100; // tcp $s0, $s0
           #8;
           instruction = 10'b0011110001; // sll $t0, $s0, $t1
           #8;
           instruction = 10'b0110100011; // beq $t0, $zero, PC=PC+4+3
           #8;
           instruction = 10'b1001010011; // addi $t1, $t1, 3
           #8;
```

```
            instruction = 10'b0110110011; // beq $t0, $t1, PC=PC+4+3
            #8;
            instruction = 10'b1110001100; // jr $s0
            #8;
            instruction = 10'b1110111101; // j 01111 (15)
            #8;
            instruction = 10'b1111000010; // jal 10000 (-16)
            #8;
            instruction = 10'b1001010011; // addi $t1, $t1, 3
            #8;
            instruction = 10'b1110011100; // jr $ra
            #8;
            instruction = 10'b1001010011; // addi $t1, $t1, 3
            #8;
            instruction = 10'b0101110001; // cmp $t0, $s0, $t1
            #8;
            instruction = 10'b1110000011; // halt
            #8;

    end



endmodule

module ControlUnit(
   input wire [9:0] instruction,
   output reg [1:0]ALU_op,
   output reg reg_write,
   output reg [2:0] reg_write_addr,
   output reg [2:0] reg_read1_addr,
   output reg alu_source1_control,
   output reg [1:0]alu_source2_control,
   output reg [1:0]pc_control
    );

    reg [2:0]op_code;
    reg [1:0]funct_code;
    reg [1:0]rs_addr;
    reg [1:0]rt_addr;
    reg [2:0]rd_addr; //also hold immediate value for I type instructions
    reg [4:0]j_addr;

    wire reg_write_mux_val;
    reg_write_mux rwm0( .op_code(op_code), .funct_code(funct_code), .reg_write
(reg_write_mux_val) );

    wire [1:0]ALU_op_mux_val;
    ALU_op_mux aom0( .op_code(op_code), .rd_addr(rd_addr), .tcp(2'b11), .ALU_op
(ALU_op_mux_val) );

    wire [1:0]pc_control_mux_val;
    pc_control_mux pcm0( .op_code(op_code), .funct_code(funct_code), .pc_control
(pc_control_mux_val) );

    wire [2:0]reg_write_addr_control_val;
    reg_write_addr_control_mux rwacm0( .op_code(op_code), .rd_addr(rd_addr), .rs_addr
(rs_addr), .write_addr(reg_write_addr_control_val) );

    wire [2:0]reg_read1_addr_mux_val;
    reg_read1_addr_mux rram0( .op_code(op_code), .funct_code(funct_code), .rd_addr
(rd_addr), .rs_addr(rs_addr), .rt_addr(rt_addr), .j_addr(j_addr), .read_addr
(reg_read1_addr_mux_val) );
```

```
    wire alu_source1_control_mux_val;
    ALU_source1_control_mux asm1( .op_code(op_code), .funct_code(funct_code), .control
(alu_source1_control_mux_val) );

    wire [1:0] alu_source2_control_mux_val;
    ALU_source2_control_mux asm2( .op_code(op_code), .funct_code(funct_code), .control
(alu_source2_control_mux_val) );

    always@(*) begin
        // instruction decode
        op_code <= instruction[9:7];
        funct_code <= instruction[1:0];
        rs_addr <= instruction[6:5];
        rt_addr <= instruction[4:3];
        rd_addr <= instruction[2:0];
        j_addr <= instruction[6:2];


       reg_write <= reg_write_mux_val;
       ALU_op <= ALU_op_mux_val;
       pc_control <= pc_control_mux_val;
       reg_write_addr <= reg_write_addr_control_val;
       reg_read1_addr <= reg_read1_addr_mux_val;
       alu_source1_control <= alu_source1_control_mux_val;
       alu_source2_control <= alu_source2_control_mux_val;

    end

endmodule

// read1_addr
module reg_read1_addr_mux(
        input wire [2:0] op_code,
        input wire [1:0] funct_code,
        input wire [2:0] rd_addr,
        input wire [1:0] rs_addr,
        input wire [1:0] rt_addr,
        input wire [4:0] j_addr,
        output reg [2:0] read_addr
    );
    always@(*)begin
            case(op_code)
                3'b000: read_addr <= rd_addr; // add
                3'b100: read_addr <= rt_addr; // addi
                3'b101: read_addr <= rt_addr; // sw
                3'b110: read_addr <= rt_addr; // lw
                3'b111: begin
                            case(funct_code)
                                2'b00: read_addr <= j_addr; // jr
                                default: read_addr <= rs_addr;
                            endcase
                        end
                default: read_addr <= rs_addr;
            endcase
        end
endmodule

module reg_write_addr_control_mux(
    input wire [2:0] op_code,
    input wire [2:0] rd_addr,
    input wire [1:0] rs_addr,
    output reg [2:0] write_addr
    );
    always@(*)begin
```

```
        case(op_code)
            3'b000: write_addr <= rs_addr; // add
            3'b100: write_addr <= rs_addr; // addi
            3'b110: write_addr <= rs_addr; // lw
            3'b101: write_addr <= rs_addr; // sw
            3'b111: write_addr <= 3'b111; // jal
            default: write_addr <= rd_addr;
        endcase
    end
endmodule

// ALU Source 1 Control
// control:
//  0 -> reg_read1_data
//  1 -> PC
module ALU_source1_control_mux(
    input wire [2:0] op_code,
    input wire [1:0] funct_code,
    output reg control
    );
    always@(*)begin
        case(op_code)
            3'b111: begin
                        case(funct_code)
                            2'b10: control <= 1'b1; // jal
                            default: control <= 1'b0;
                        endcase
                    end
            default: control <= 0;
        endcase
    end
endmodule

// ALU Source 2 Control
// control:
//  00 -> reg_read2_data
//  01 -> imm - sign extend
//  10 -> jump_link_offset
module ALU_source2_control_mux(
    input wire [2:0] op_code,
    input wire [1:0] funct_code,
    output reg [1:0]control
    );
    always@(*) begin
        case(op_code)
            3'b100: control <= 1'b01; // addi
            3'b110: control <= 1'b01; // lw
            3'b101: control <= 1'b01; // sw
            3'b111: begin
                        case(funct_code)
                            2'b10: control <= 2'b10; // jal
                            default: control <= 1'b00;
                        endcase
                    end
            default: control <= 1'b00;
        endcase
    end
endmodule

module pc_control_mux(
    input wire [2:0] op_code,
    input wire [1:0] funct_code,
    output reg [1:0] pc_control
    );
    always@(*) begin
        case(op_code)
            3'b011: pc_control <= 2'b11; // beq - branch
            3'b111: begin // J-type
                        case(funct_code)
                            2'b00: pc_control <= 2'b10; // jump_reg
                            default: pc_control <= 2'b01;  // jump
                        endcase
                    end
            default: pc_control <= 2'b00;
        endcase
    end

endmodule

module ALU_op_mux(
    input wire [2:0] op_code,
    input wire [2:0] rd_addr,
    input wire [1:0] tcp,
    output reg [1:0] ALU_op
    );

    always@(*) begin
        case(op_code)
            //3'b000: ALU_op <= 2'b00; //add
            3'b001: ALU_op <= 2'b10; //sll
            3'b010: ALU_op <= 2'b01; //cmp
            3'b011: ALU_op <= 2'b01; //beq
            3'b100: begin
                        case(rd_addr)
                            3'b100: ALU_op <= tcp; //tcp Twos complement
                            default: ALU_op <= 2'b00; //addi
                        endcase
                    end
            default: ALU_op <= 2'b00;
        endcase
    end

endmodule

module reg_write_mux(
    input wire [2:0] op_code,
    input wire [1:0] funct_code,
    output reg reg_write
    );
    always@(*) begin
            case(op_code)
                3'b011: reg_write <= 0; //beq
                3'b101: reg_write <= 0; //sw:
                3'b111: begin   // j-type
                            case(funct_code)
                                2'b10: reg_write <= 1; // jal
                                default: reg_write <= 0;
                            endcase
                        end
                default: reg_write <= 1;
            endcase
    end
endmodule
```

```
`timescale 1ns / 1ns
`include "ALU.v"
`include "Registers.v"

//////////////////////////////////////////////////////////////////////////////////
// Company: Western Michigan University
// Engineer: Tyler Thompson
//
// Create Date: 02/13/2018 02:47:56 PM
// Design Name:
// Module Name: FetchUnit
// Project Name: ECE 3570 Lab 2
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies: ALU.v, Registers.v
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////



module FetchUnitTest();
    reg clk, reset;
    reg [1:0] pc_control;
    reg [9:0] branch_control;
    reg [4:0] jump_address;
    reg [2:0] branch_address;
    reg [9:0] reg_address;
    wire [9:0] PC;
    //toggle clock every 5ns
    always #5 clk = ~clk;

    FetchUnit fu0( .clk(clk), .reset(reset), .pc_control(pc_control), .branch_control
(branch_control), .jump_address(jump_address), .branch_address
(branch_address), .reg_address(reg_address), .PC_out(PC) );

    initial begin
        branch_control = 1;
        clk = 0;
        reset = 1; //initialize the program counter
        #20
        reset = 0;
        #15

        pc_control = 2'b00; // increment program counter by 1
        #20
        pc_control = 2'b01; jump_address = 5'b00100; // increment program count by 4
        #20
        pc_control = 2'b10; reg_address = 10'b0001000000; // set program counter to 64
        #20
        pc_control = 2'b11; branch_address = 3'b010; branch_control = 0;// branch to
address
        #20
        pc_control = 2'b00;
    end
endmodule


// Program Counter control unit
// pc_control:  00 -> increment program count by 1
//              01 -> add jump_address to program counter
//              10 -> set program counter to reg_address
//              11 -> add 4 to program count and then add the branch_address
module FetchUnit(
    input wire clk,
    input wire reset,
    input wire [1:0] pc_control,
    input wire [9:0] branch_control,
    input wire [4:0] jump_address,
    input wire [2:0] branch_address,
    input wire [9:0] reg_address,
    output wire [9:0] PC_out
    );
    reg [9:0]PC_in;
    reg [9:0] b;
    wire c_out;
    wire [9:0] sum;
    wire [9:0] branch_offset;
    reg [9:0] branch_ext;

    //Create 10 bit register for program counter
    Register_10bit pc_reg( .clk(clk), .reset(reset), .Din(PC_in), .Dout(PC_out) );

    //adder for calculating branch offset
    FullAdder_10Bit fa1( .a( branch_ext ), .b(10'b0000000100), .sum
(branch_offset), .c_out(c_out) );

    //adder for calculating pc address
    FullAdder_10Bit fa0( .a(PC_out), .b(b), .sum(sum), .c_out(c_out) );

    //update the program counter on each clock cycle
    always @(*) begin
            //sign extend branch address
            branch_ext <= { {7{branch_address[2]}}, branch_address[2:0] };
            PC_in <= sum;
            b <= 0;
            if ( reset == 0 ) begin
                case(pc_control)
                    2'b00: b <= 10'b0000000001;
                    2'b01: b <= { {5{jump_address[4]}}, jump_address[4:0] }; // sign
extend jump address
                    2'b10: PC_in <= reg_address;
                    2'b11: begin // branch
                            if ( branch_control == 0 ) begin
                                b <= branch_offset;
                            end
                            else begin
                                b <= 10'b0000000001;
                            end
                           end
                    default: PC_in <= PC_out;
                endcase
            end
            else begin
                PC_in <= 0;
            end

    end



endmodule
```

```verilog
`timescale 1ns / 1ns
//////////////////////////////////////////////////////////////////////////////////
// Company: Western Michigan University
// Engineer: Tyler Thompson
//
// Create Date: 02/13/2018 02:44:18 PM
// Design Name:
// Module Name: ALU
// Project Name: ECE 3570 Lab 2
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module ALUTest();

    wire [9:0] result;
    wire c_out;

    reg [9:0] a;
    reg [9:0] b;
    reg [1:0] ALU_op;

    ALU alu0( .ALU_op(ALU_op), .a(a), .b(b), .f(result), .c_out(c_out) );

    initial begin
        ALU_op = 2'b00;
        a = 10'b0000000000;
        b = 10'b0000000000;
    end

    always @(a or b) begin
        //adder test
         a = 10'b0000000001; b = 10'b0000000001; ALU_op = 2'b00; // 1 + 1
         #8;
         a = 10'b0000000011; b = 10'b0000000001; ALU_op = 2'b00; // 3 + 1
         #8;
         a = 10'b0000000010; b = 10'b0000000001; ALU_op = 2'b00; // 2 + 1
         #8;
         a = 10'b0010110010; b = 10'b0011001101; ALU_op = 2'b00; // 178 + 205 = 383
         #8;
         a = 10'b1101001110; b = 10'b0011001101; ALU_op = 2'b00; // -178 + 205 = 27
         #8;
         a = 10'b1101001110; b = 10'b1111111110; ALU_op = 2'b00; // -178 + -2 = -180
         #8;

        //shifter test
        a = 10'b0001000000; b = 10'b1111111111; ALU_op = 2'b10; //shift a right by 1
        #8;
        a = 10'b0000000001; b = 10'b0000000001; ALU_op = 2'b10; //shift a left by 1
        #8;
        a = 10'b0001000000; b = 10'b1111111100; ALU_op = 2'b10; //shift a right by 4
        #8;
        a = 10'b0000000001; b = 10'b0000000100; ALU_op = 2'b10; //shift a left by 4
        #8;
        //comparator test
        a = 10'b0000000010; b = 10'b0000000001; ALU_op = 2'b01; //test if 2 is
greater than 1
        #8;
        a = 10'b0000000001; b = 10'b0000000010; ALU_op = 2'b01; //test if 1 is
greater than 2
        #8;
        a = 10'b0000000001; b = 10'b0000000001; ALU_op = 2'b01; //test if 1 is
greater than 1
        #8;
        a = 10'b1111111111; b = 10'b0000000001; ALU_op = 2'b01; //test if -1 is
greater than 1
        #8;
        a = 10'b0000000001; b = 10'b1111111111; ALU_op = 2'b01; //test if 1 is
greater than -1
        #8;
        a = 10'b1111111110; b = 10'b1111111111; ALU_op = 2'b01; //test if -2 is
greater than -1
        #8;
        a = 10'b1111111111; b = 10'b1111111110; ALU_op = 2'b01; //test if -1 is
greater than -2
        #8;
        a = 10'b1111110000; b = 10'b1111111100; ALU_op = 2'b01; //test if -16 is
greater than -4
        #8;
        a = 10'b1111111000; b = 10'b1111101100; ALU_op = 2'b01; //test if -8 is
greater than -20
        #8;
        a = 10'b1000000000; b = 10'b1000000010; ALU_op = 2'b01; //test if -512 is
greater than -510
        #8;
        a = 10'b1000000010; b = 10'b1000000000; ALU_op = 2'b01; //test if -510 is
greater than -512
        #8;
        a = 10'b0111111111; b = 10'b0111111110; ALU_op = 2'b01; //test if 511 is
greater than 510
    end

endmodule


// Arithmetic Logic Unit
// Supports add, shift, and compare operations
// ALU_op: 00 -> adder
//         01 -> comparator
//         10 -> shifter
//         11 -> Twos Complement
module ALU(
    input wire [1:0] ALU_op,
    input wire [9:0] a,
    input wire [9:0] b,
    output wire [9:0] f,
    output c_out
    );
    wire [9:0] add_result;
    wire [9:0] shift_result;
    wire [9:0] comp_result;
    wire [9:0] complement_result;

    FullAdder_10Bit fa0( .a(a), .b(b), .sum(add_result), .c_out(c_out) );
    Comparator cp0( .a(a), .b(b), .out(comp_result) );
    Shifter sh0( .a(a), .shift_amount(b), .out(shift_result) );
    Complementor cptr0( .a(a), .out(complement_result) );
    ALU_mux mx0( .ALU_op(ALU_op), .add_result(add_result), .comp_result
(comp_result), .shift_result(shift_result), .complement_result
(complement_result), .final_answer(f) );

endmodule


// Multiplexer for ALU output selection
// ALU_op: 00 -> adder
//         01 -> comparator
//         10 -> shifter
//         11 -> complementor
module ALU_mux(
     input wire [1:0] ALU_op,
         input wire [9:0] add_result,
         input wire [9:0] comp_result,
         input wire [9:0] shift_result,
         input wire [9:0] complement_result,
         output reg [9:0] final_answer
         );

         always@(*) begin
                case(ALU_op)
                         2'b00: final_answer <= add_result;
                         2'b01: final_answer <= comp_result;
                 2'b10: final_answer <= shift_result;
                 2'b11: final_answer <= complement_result;
                 default: final_answer <= 10'b0;
                endcase
        end
endmodule

// takes the twos complement of an input.
module Complementor(
    input wire [9:0] a,
    output reg [9:0] out
    );
   reg [9:0] not_a;
   wire c_out;
   wire [9:0] sum;
   FullAdder_10Bit fa1( .a(not_a), .b(10'b0000000001), .sum(sum), .c_out(c_out) );

   always@(*)begin
        not_a <= ~a;
        out <= sum;
   end
endmodule


// Compares a to b
// out = 2 if a is grater than b and 0 if a is equal to b and 1 if a is less than b
module Comparator(
    input wire [9:0] a,
    input wire [9:0] b,
    output reg [9:0] out //2 -> a is greater than b, 0 -> a equal to b, 1-> a is less
than b
    );

    reg [9:0]temp0;
    wire [9:0] temp1;
    wire c_out;
    wire [9:0]sum;
    wire c_out1;

    //adder for twos complement
    FullAdder_10Bit fa1( .a(10'b0000000001), .b(temp0), .sum(temp1), .c_out(c_out) );

    FullAdder_10Bit fa2( .a(b), .b(temp1), .sum(sum), .c_out(c_out1) );
    always@(*) begin
        temp0 <= 0;
        // a and b are equal
        if ( a == b ) begin
            out = 10'b0000000000;
        end
        //both a and b are negative
        else if ( (a[9] & b[9]) == 1 ) begin
           //twos complement of a
           temp0 <= ~a;

           if (c_out1 == 1) begin
                out = 10'b0000000001;

           end
           else begin
                out = 10'b0000000010;

           end
        end
        //a is negative and b is positive
        else if ( a[9] == 1 ) begin
            out = 10'b0000000001;
        end
        //a is positive and b is negative
        else if ( b[9] == 1 ) begin
            out = 10'b0000000010;
        end
        //a is positive and b is positive
        else begin
            //twos completment of a
            temp0 <= ~a;

            if ( sum[9] == 1 ) begin
                out = 10'b0000000010;
            end
            else begin
                out = 10'b0000000001;
            end
         end

    end
endmodule


// Shift a number left or right
// a positive shift_amount value will shift the number left
// a negative shift_amount value will shift the number right
module Shifter(
    input [9:0] a,
    input [9:0] shift_amount,
    output reg [9:0] out
    );
    reg [9:0] x;
    wire [9:0] sum;
    wire c_out;
    FullAdder_10Bit fa2( .a(x), .b(10'b0000000001), .sum(sum), .c_out(c_out) );
   always @(*) begin
        //if shift_amount is negative, take twos complement and shift right
        if ( shift_amount[9] == 1'b1 ) begin
           x <= ~shift_amount ;
```

```
            out = a >> sum;
        end
        else begin
            out = a << shift_amount;
            x <= 0;
        end
    end
endmodule

// Ripple adder for 10 bit numbers
module FullAdder_10Bit(
    input [9:0] a,
    input [9:0] b,
    output [9:0] sum,
    output c_out
    );
        wire [8:0]ripple;

        FullAdder_1Bit f0( .a( a[0] ), .b( b[0] ), .c_in( 1'b0 ), .s( sum[0]), .c_out
( ripple[0] ) );
        FullAdder_1Bit f1( .a( a[1] ), .b( b[1] ), .c_in( ripple[0] ), .s( sum
[1]), .c_out(ripple[1]) );
        FullAdder_1Bit f2( .a( a[2] ), .b( b[2] ), .c_in( ripple[1] ), .s( sum
[2]), .c_out(ripple[2]) );
        FullAdder_1Bit f3( .a( a[3] ), .b( b[3] ), .c_in( ripple[2] ), .s( sum
[3]), .c_out(ripple[3]) );
        FullAdder_1Bit f4( .a( a[4] ), .b( b[4] ), .c_in( ripple[3] ), .s( sum
[4]), .c_out(ripple[4]) );
        FullAdder_1Bit f5( .a( a[5] ), .b( b[5] ), .c_in( ripple[4] ), .s( sum
[5]), .c_out(ripple[5]) );
        FullAdder_1Bit f6( .a( a[6] ), .b( b[6] ), .c_in( ripple[5] ), .s( sum
[6]), .c_out(ripple[6]) );
        FullAdder_1Bit f7( .a( a[7] ), .b( b[7] ), .c_in( ripple[6] ), .s( sum
[7]), .c_out(ripple[7]) );
        FullAdder_1Bit f8( .a( a[8] ), .b( b[8] ), .c_in( ripple[7] ), .s( sum
[8]), .c_out(ripple[8]) );
        FullAdder_1Bit f9( .a( a[9] ), .b( b[9] ), .c_in( ripple[8] ), .s( sum
[9]), .c_out(c_out) );

    endmodule

//simple adder for adding two bits
module FullAdder_1Bit(
    input wire a,
    input wire b,
    input wire c_in, //carry in
        output wire s, //sum
        output wire c_out //carry out
    );

        assign s = a ^ b ^ c_in;
        assign c_out = ((a ^ b) & c_in) | (a & b);

endmodule
```

```
`timescale 1ns / 1ns
//////////////////////////////////////////////////////////////////////////////////
// Company: Western Michigan University
// Engineer: Tyler Thompson
//
// Create Date: 02/05/2018 07:30:26 PM
// Design Name:
// Module Name: Registers
// Project Name: ECE 3570 Lab 2
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

module RegisterFileTest();
     reg clk;
     reg en_write;
     reg [2:0] write_addr;
     reg [9:0] write_data;
     reg [2:0] read1_addr;
     reg [2:0] read2_addr;
     wire [9:0] read1_data;
     wire [9:0] read2_data;

    //toggle clock every 4ns
    always #4 clk = ~clk;

    RegisterFile rf( .clk(clk), .en_write(en_write), .write_addr
(write_addr), .write_data(write_data), .read1_addr(read1_addr), .read2_addr
(read2_addr), .read1_data(read1_data), .read2_data(read2_data) );

    initial begin
        en_write = 0;
        clk = 0;
        #4;
        //prove you can't write to the zero register
        en_write = 1; write_addr = 3'b000; write_data = 10'b1111111111; read1_addr =
3'b000; read2_addr = 3'b000;
        #8;
        //write to $t0 and read $zero twice
        en_write = 1; write_addr = 3'b001; write_data = 10'b0000001111; read1_addr =
3'b000; read2_addr = 3'b000;
        #8;
        //write to $t1 and read $t0 and $zero
        en_write = 1; write_addr = 3'b010; write_data = 10'b1010101010; read1_addr =
3'b001; read2_addr = 3'b000;
        #8;
        //write to $s0 and read $t1 and $t0
        en_write = 1; write_addr = 3'b011; write_data = 10'b0101010101; read1_addr =
3'b010; read2_addr = 3'b001;
        #8;
        //write to $sp and read $s0 and $t1
        en_write = 1; write_addr = 3'b100; write_data = 10'b1111000000; read1_addr =
3'b011; read2_addr = 3'b010;
        #8;
         //write to $a0 and read $sp and $s0
        en_write = 1; write_addr = 3'b101; write_data = 10'b0000111111; read1_addr =
3'b100; read2_addr = 3'b011;
        #8;
        //write to $v0 and read $a0 and $sp
        en_write = 1; write_addr = 3'b110; write_data = 10'b1111111111; read1_addr =
3'b101; read2_addr = 3'b100;
        #8;
        //write to $ra and read $v0 and $a0
        en_write = 1; write_addr = 3'b111; write_data = 10'b1100110011; read1_addr =
3'b110; read2_addr = 3'b101;
        #8;
        //read from $ra and $v0
        en_write = 0; read1_addr = 3'b111; read2_addr = 3'b110;

    end
endmodule


// Register file, read and write data to registers
// addressing:  000 -> $zero
//              001 -> $t0
//              010 -> $t1
//              011 -> $s0
//              100 -> $sp
//              101 -> $a0
//              110 -> $v0
//              111 -> $ra
module RegisterFile(
    input wire clk,
    input wire reset,
    input wire en_write,
    input wire [2:0] write_addr,
    input wire [9:0] write_data,
    input wire [2:0] read1_addr,
    input wire [1:0] read2_addr,
    output reg [9:0] read1_data,
    output reg [9:0] read2_data
    );

      //Array of data input for registers
      reg [9:0]Din [7:0];

      //Array of data outputs for registers
      wire [9:0]Dout[7:0];

      //reg [2:0] read1_addr_ext;
      //wire reset = 0;
      //Initialize all general purpose registers
      Register_10bit reg_zero(.clk(clk), .reset(1'b1), .Din(Din[0]), .Dout(Dout[0]));
      Register_10bit reg_t0(.clk(clk), .reset(reset), .Din(Din[1]), .Dout(Dout[1]));
      Register_10bit reg_t1(.clk(clk), .reset(reset), .Din(Din[2]), .Dout(Dout[2]));
      Register_10bit reg_s0(.clk(clk), .reset(reset), .Din(Din[3]), .Dout(Dout[3]));
      Register_10bit reg_sp(.clk(clk), .reset(reset), .Din(Din[4]), .Dout(Dout[4]));
      Register_10bit reg_a0(.clk(clk), .reset(reset), .Din(Din[5]), .Dout(Dout[5]));
      Register_10bit reg_v0(.clk(clk), .reset(reset), .Din(Din[6]), .Dout(Dout[6]));
      Register_10bit reg_ra(.clk(clk), .reset(reset), .Din(Din[7]), .Dout(Dout[7]));

    always@(*) begin
        Din[0] <= Dout[0]; Din[1] <= Dout[1]; Din[2] <= Dout[2]; Din[3] <= Dout[3];
Din[4] <= Dout[4]; Din[5] <= Dout[5]; Din[6] <= Dout[6]; Din[7] <= Dout[7];
        read1_data <= Dout[read1_addr];
        read2_data <= Dout[read2_addr];
        if ( en_write == 1 ) begin
            Din[write_addr] <= write_data;
        end
      end

endmodule


module RegisterTest();
  reg clk, reset;
  //toggle clock every 4ns
  always #4 clk = ~clk;

  //Array of data input for registers
  reg [9:0]Din [7:0];

  //Array of data outputs for registers
  wire [9:0]Dout[7:0];

  //Initialize all general purpose registers
  Register_10bit reg_zero(.clk(clk), .reset(1'b1), .Din(Din[0]), .Dout(Dout[0]));
  Register_10bit reg_t0(.clk(clk), .reset(reset), .Din(Din[1]), .Dout(Dout[1]));
  Register_10bit reg_t1(.clk(clk), .reset(reset), .Din(Din[2]), .Dout(Dout[2]));
  Register_10bit reg_s0(.clk(clk), .reset(reset), .Din(Din[3]), .Dout(Dout[3]));
  Register_10bit reg_sp(.clk(clk), .reset(reset), .Din(Din[4]), .Dout(Dout[4]));
  Register_10bit reg_a0(.clk(clk), .reset(reset), .Din(Din[5]), .Dout(Dout[5]));
  Register_10bit reg_v0(.clk(clk), .reset(reset), .Din(Din[6]), .Dout(Dout[6]));
  Register_10bit reg_ra(.clk(clk), .reset(reset), .Din(Din[7]), .Dout(Dout[7]));

  //run a test on the registers
  integer i;
  always @(posedge clk) begin
      for( i = 0; i < 8; i = i + 1) begin
          Din[i] = ~Dout[i];
          #4;
      end
  end

  initial begin
      clk = 0;
      reset = 0;

      //initialize all registers with zeros
      { Din[0], Din[1], Din[2], Din[3], Din[4], Din[5], Din[6], Din[7] } = { 10'b0,
10'b0, 10'b0, 10'b0, 10'b0, 10'b0, 10'b0, 10'b0 };
  end

endmodule


module Register_10bit(
    input wire clk,
    input wire reset,
    input wire [9:0] Din,
    output reg [9:0] Dout
    );

    always @(posedge clk) begin
        if (reset == 1'b1) begin
            Dout <= 10'b0000000000;
        end
        else begin
            Dout <= Din;
        end
    end

endmodule
```