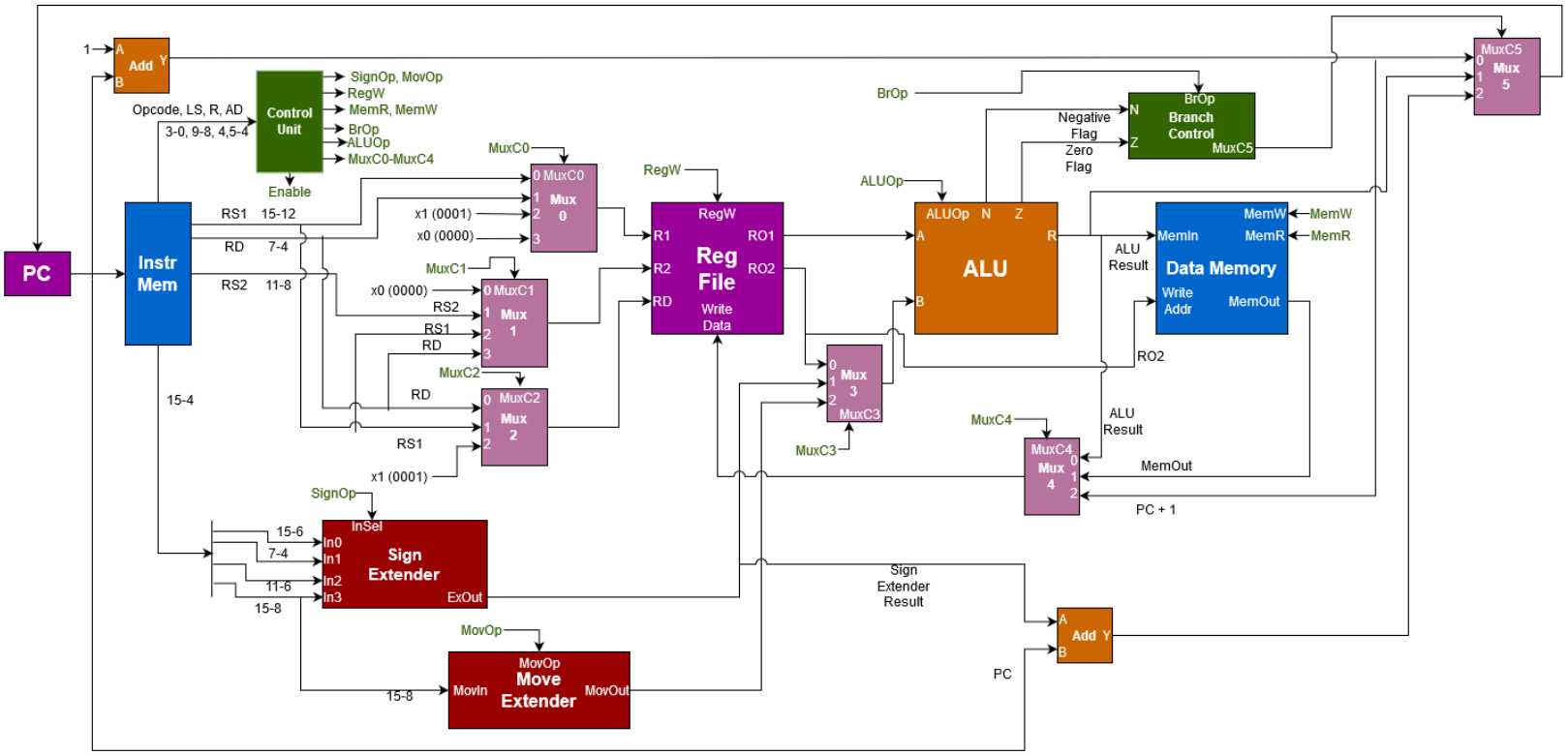
# Datapath and Control Top Component

## 1 Overview

- All datapath and control components must be combined into a top-level component.
- Each instruction in the ISA must be simulated and tested.
- A short program using all types of instructions (except load and store) must be written, compiled into binary, and simulated.
- The top-level component code can be seen in Appendix A.
- Slight modifications have been made to the datapath design and components. An updated datapath diagram can be seen below.



# 2 Instruction Verification

In the sections below, each instruction of the RISC-Z ISA are tested and the simulation results are discussed.
**Note:** In all simulations, the `inr` input selects a register to view. The contents of the register selected by `inr` are outputted as `outvalue`.

## 2.1 Register Instructions

There are 5 R-Type instructions to be tested:

1. Add Register: `addr RD, RS1, RS2`
    - RD = RS1 + RS2
2. AND Register: `andr RD, RS1, RS2`
    - RD = RS1 & RS2
3. NAND Register: `nandr RD, RS1, RS2`
    - RD = RS1 $\sim$ & RS2
4. OR Register: `orr RD, RS1, RS2`
    - RD = RS1 | RS2
5. Subtract Register: `subr RD, RS1, RS2`
    - RD = RS1 - RS2

The register file has been initialized to the following values for testing purposes:

- x0-x4, x9-x15 = 0
- x5 = 100
- x6 = 3000
- x7 = 56
- x8 = -200

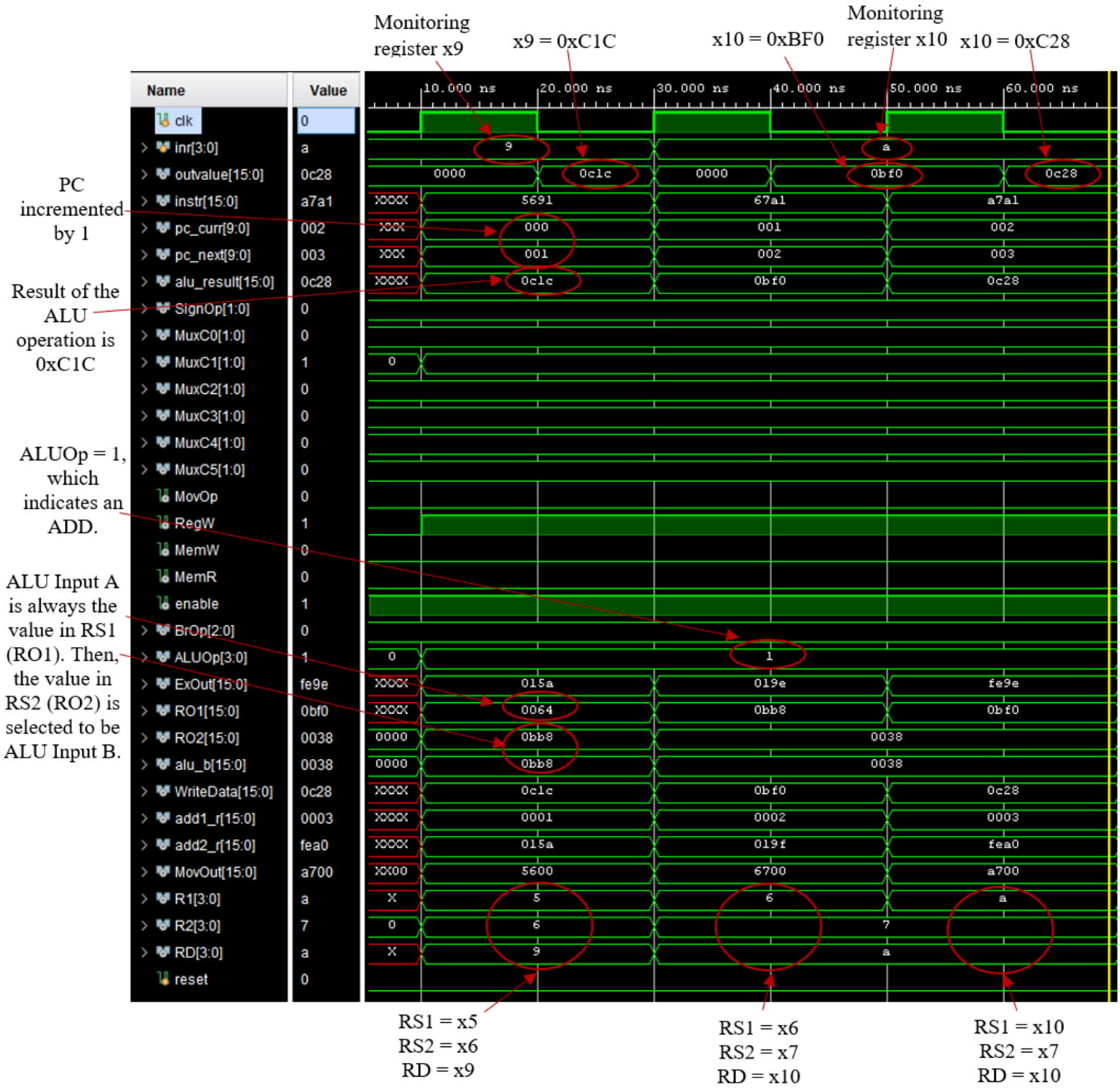
The simulation results for each immediate instruction can be seen below.

### 2.1.1 addr Instruction

Instructions used in simulation:

| Instr. Addr. | Binary | Instr. | Expected Result |
|---|---|---|---|
| 0 | 0101011010010001 | addr x9, x5, x6 | x9 = 0x64 + 0xBB8 = 0xC1C |
| 1 | 0110011110100001 | addr x10, x6, x7 | x10 = 0xBB8 + 0x38 = 0xBF0 |
| 2 | 1010011110100001 | addr x10, x10, x7 | x10 = 0xBF0 + 0x38 = 0xC28 |

In the simulation, the instructions above were placed into the instruction memory. The resulting simulation waveform can be seen below.
The simulation waveform was observed to match the expected results in the above table. Thus, the `addr` instruction was confirmed to function correctly.

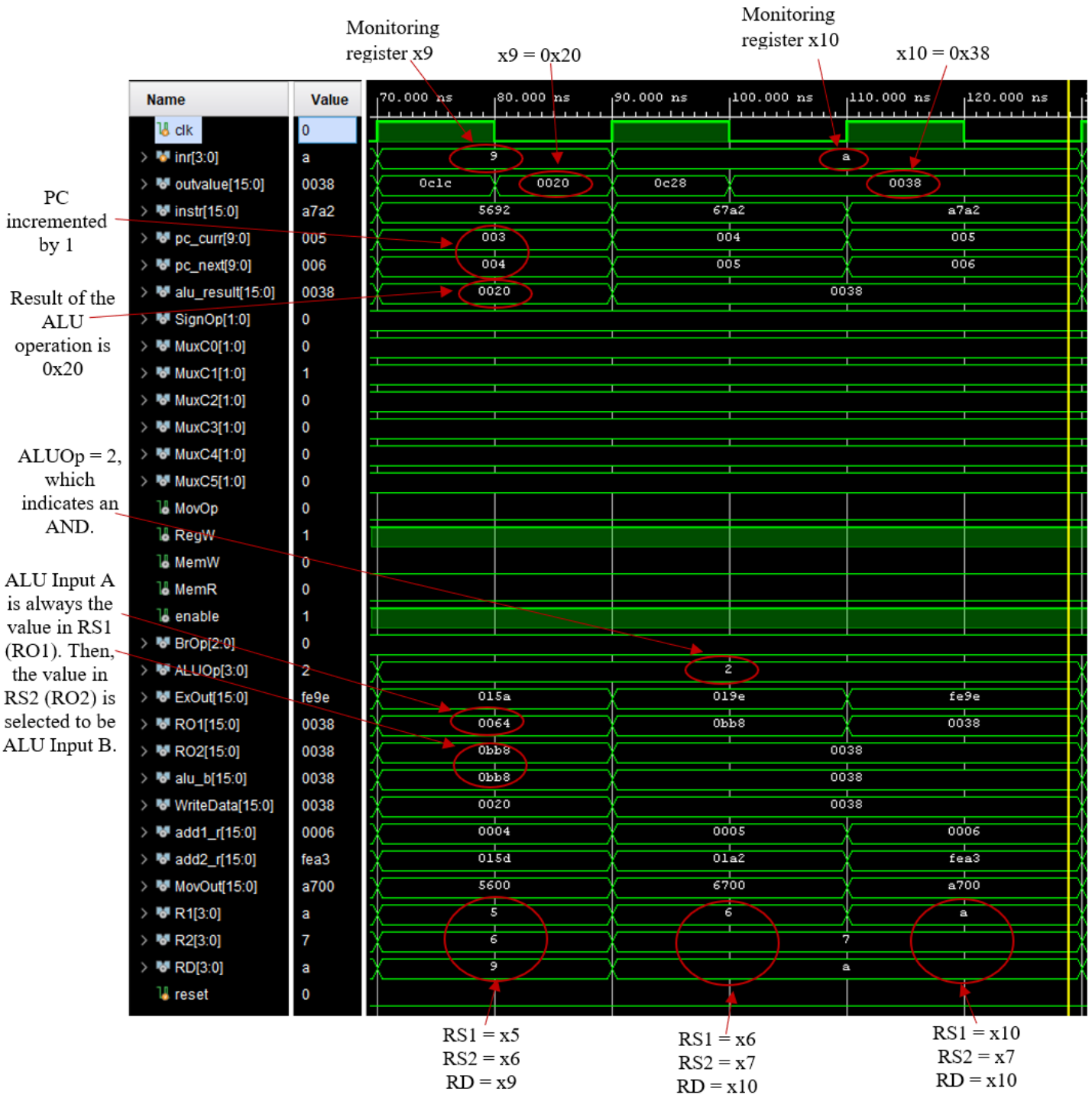

Monitoring register x9
x9 = 0xC1C
x10 = 0xBF0
Monitoring register x10
x10 = 0xC28
PC incremented by 1
Result of the ALU operation is 0xC1C
ALUOp = 1, which indicates an ADD.
ALU Input A is always the value in RS1 (RO1). Then, the value in RS2 (RO2) is selected to be ALU Input B.
RS1 = x5
RS2 = x6
RD = x9
RS1 = x6
RS2 = x7
RD = x10
RS1 = x10
RS2 = x7
RD = x10

### 2.1.2 andr Instruction

Instructions used in simulation:

| Instr. Addr. | Binary | Instr. | Expected Result |
|---|---|---|---|
| 3 | 0101011010010010 | andr x9, x5, x6 | x9 = 0x64 & 0xBB8 = 0x20 |
| 4 | 0110011110100010 | andr x10, x6, x7 | x10 = 0xBB8 & 0x38 = 0x38 |
| 5 | 1010011110100010 | andr x10, x10, x7 | x10 = 0x38 & 0x38 = 0x38 |

In the simulation, the instructions above were placed into the instruction memory. The resulting simulation waveform can be seen below.
The simulation waveform was observed to match the expected results in the above table. Thus, the `andr` instruction was confirmed to function correctly.

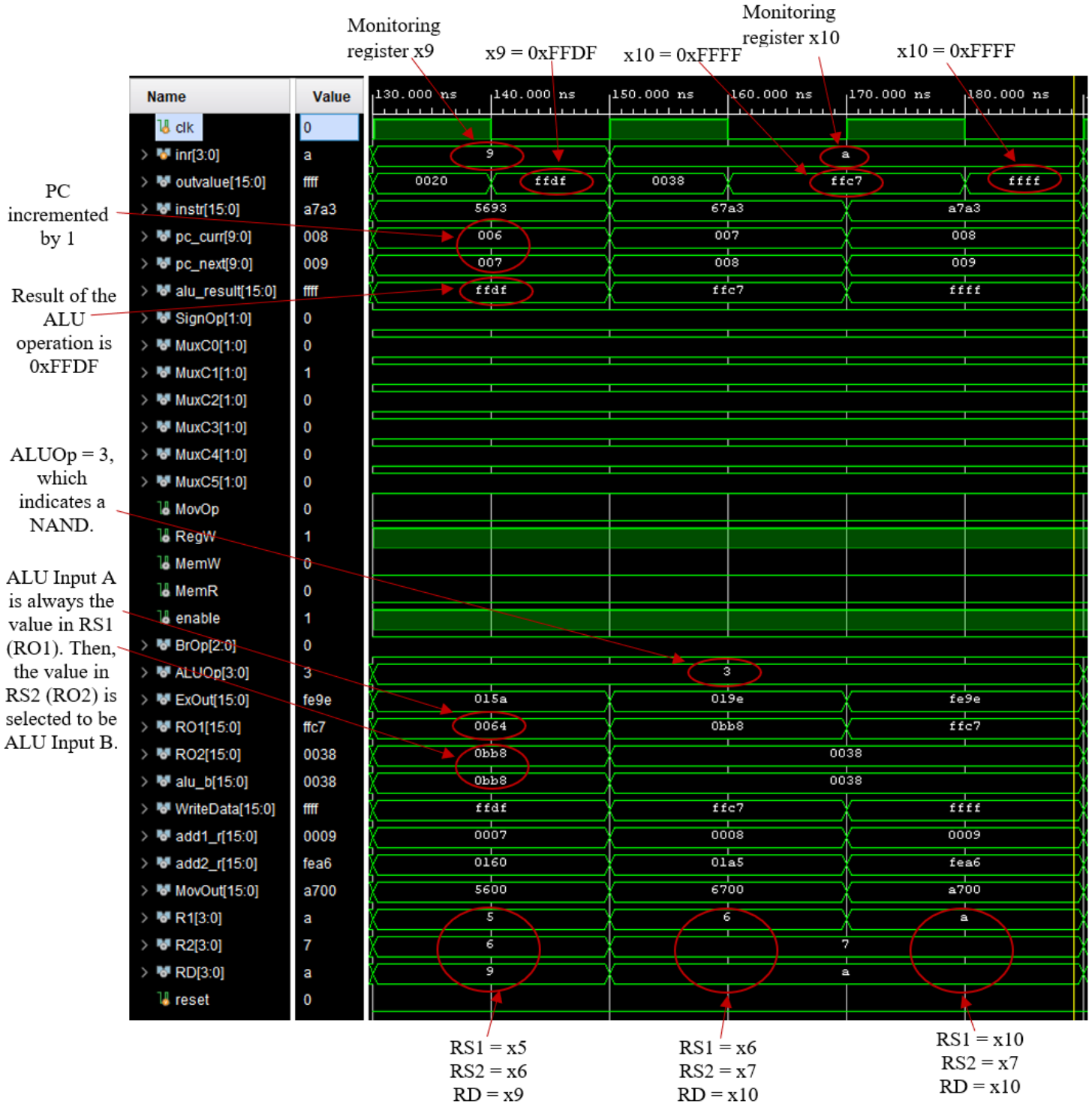

Monitoring register x9
x9 = 0x20
Monitoring register x10
x10 = 0x38
PC incremented by 1
Result of the ALU operation is 0x20
ALUOp = 2, which indicates an AND.
ALU Input A is always the value in RS1 (RO1). Then, the value in RS2 (RO2) is selected to be ALU Input B.
RS1 = x5
RS2 = x6
RD = x9
RS1 = x6
RS2 = x7
RD = x10
RS1 = x10
RS2 = x7
RD = x10

### 2.1.3 nandr Instruction

Instructions used in simulation:

| Instr. Addr. | Binary | Instr. | Expected Result |
|---|---|---|---|
| 6 | 0101011010010011 | nandr x9, x5, x6 | x9 = 0x64 $\sim$ & 0xBB8 = 0xFFDF |
| 7 | 0110011110100011 | nandr x10, x6, x7 | x10 = 0xBB8 $\sim$ & 0x38 = 0xFFC7 |
| 8 | 1010011110100011 | nandr x10, x10, x7 | x10 = 0xFFC7 $\sim$ & 0x38 = 0xFFFF |

In the simulation, the instructions above were placed into the instruction memory. The resulting simulation waveform can be seen below.
The simulation waveform was observed to match the expected results in the above table. Thus, the `nandr` instruction was confirmed to function correctly.

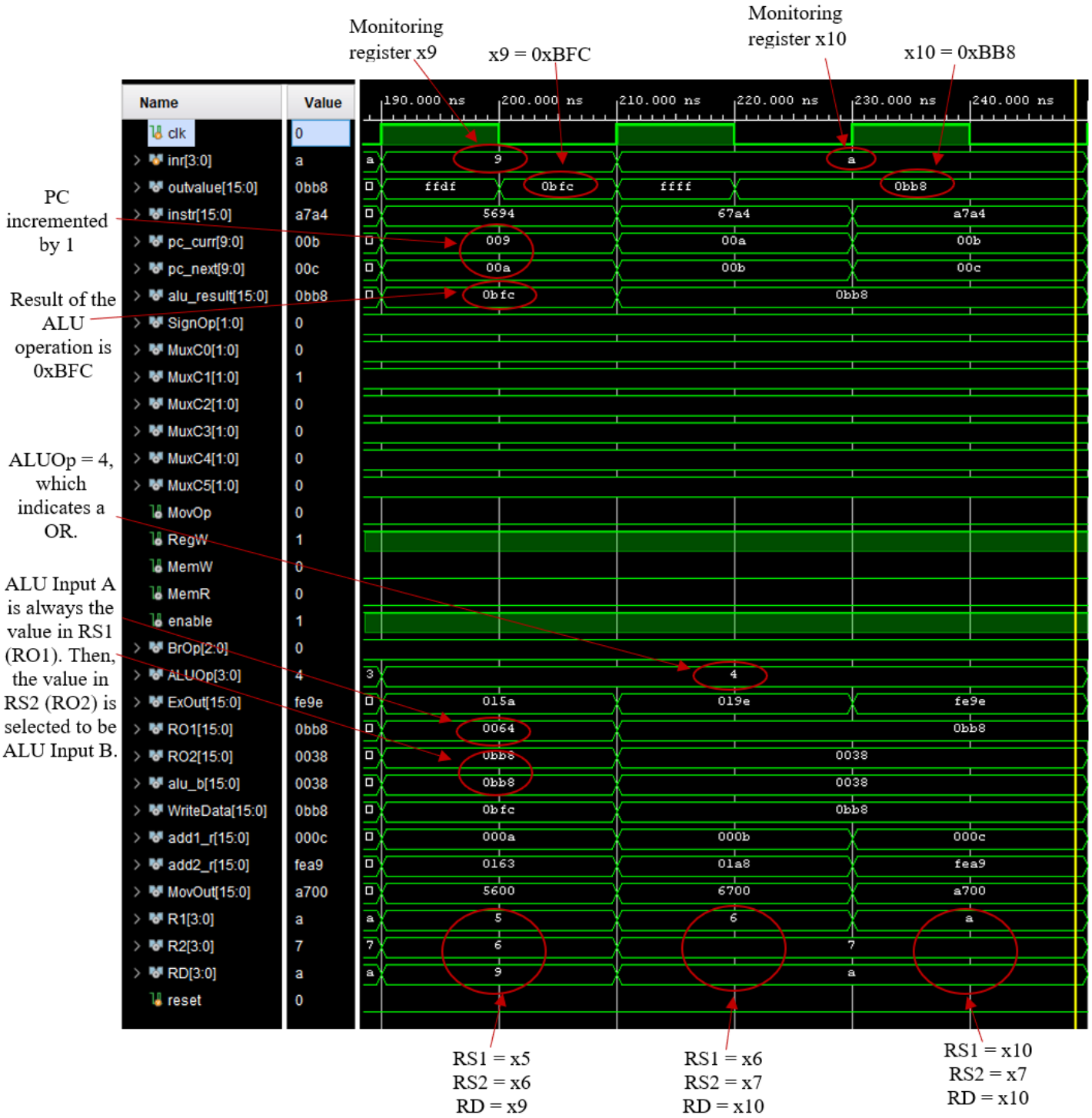

Monitoring register x9
x9 = 0xFFDF
x10 = 0xFFFF
Monitoring register x10
x10 = 0xFFFF
PC incremented by 1
Result of the ALU operation is 0xFFDF
ALUOp = 3, which indicates a NAND.
ALU Input A is always the value in RS1 (RO1). Then, the value in RS2 (RO2) is selected to be ALU Input B.
RS1 = x5
RS2 = x6
RD = x9
RS1 = x6
RS2 = x7
RD = x10
RS1 = x10
RS2 = x7
RD = x10

### 2.1.4 orr Instruction

Instructions used in simulation:

| Instr. Addr. | Binary | Instr. | Expected Result |
|---|---|---|---|
| 9 | 0101011010010100 | orr x9, x5, x6 | x9 = 0x64 \| 0xBB8 = 0xBFC |
| 10 | 0110011110100100 | orr x10, x6, x7 | x10 = 0xBB8 \| 0x38 = 0xBB8 |
| 11 | 1010011110100100 | orr x10, x10, x7 | x10 = 0xBB8 \| 0x38 = 0xBB8 |

In the simulation, the instructions above were placed into the instruction memory. The resulting simulation waveform can be seen below.
The simulation waveform was observed to match the expected results in the above table. Thus, the `orr` instruction was confirmed to function correctly.

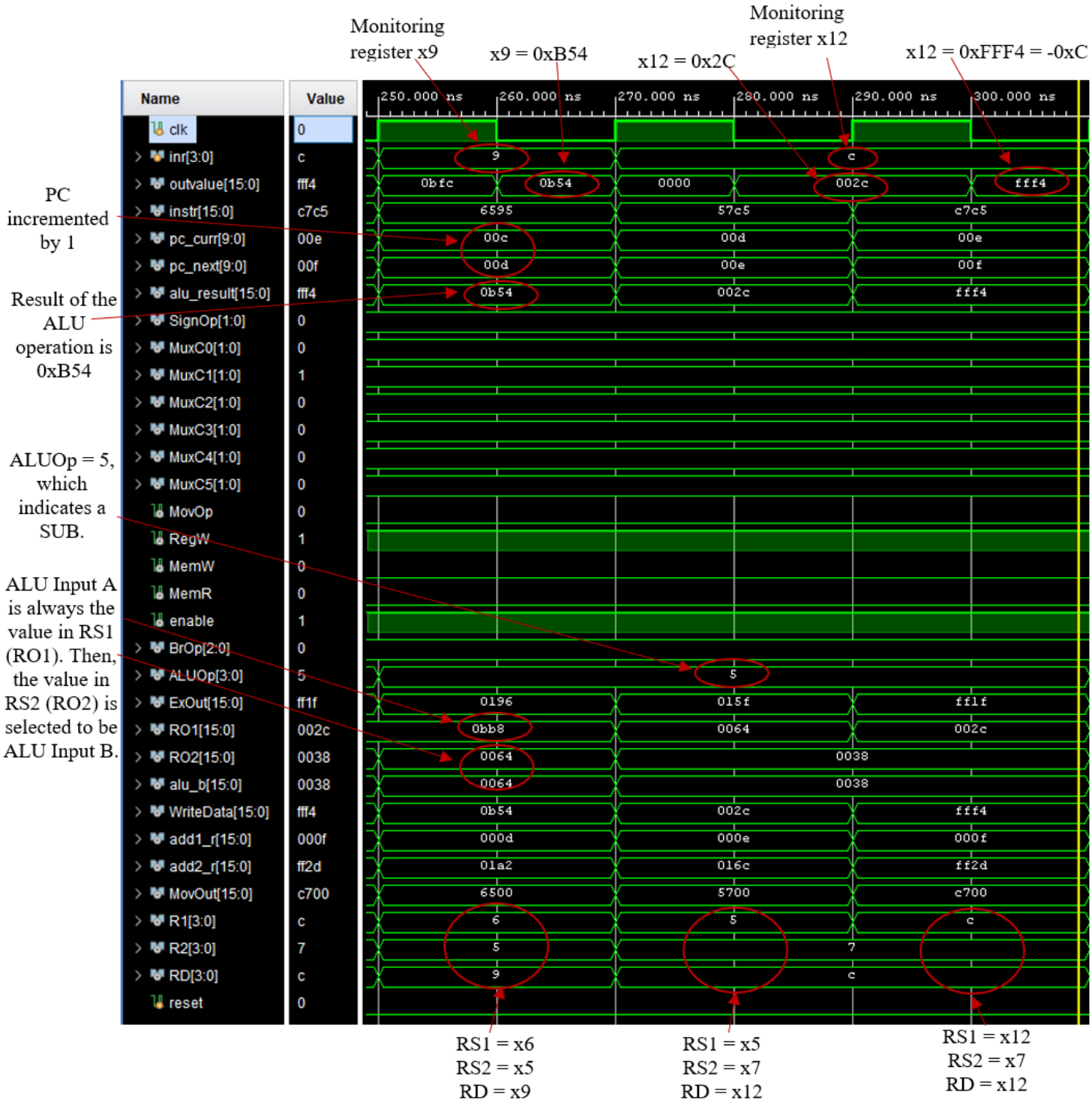

Monitoring register x9
x9 = 0xBFC
Monitoring register x10
x10 = 0xBB8
PC incremented by 1
Result of the ALU operation is 0xBFC
ALUOp = 4, which indicates a OR.
ALU Input A is always the value in RS1 (RO1). Then, the value in RS2 (RO2) is selected to be ALU Input B.
RS1 = x5
RS2 = x6
RD = x9
RS1 = x6
RS2 = x7
RD = x10
RS1 = x10
RS2 = x7
RD = x10

### 2.1.5 subr Instruction

Instructions used in simulation:

| Instr. Addr. | Binary | Instr. | Expected Result |
|---|---|---|---|
| 9 | 0110011110100101 | subr x9, x6, x5 | x9 = 0xBB8 - 0x64 = 0xB54 |
| 10 | 0101011111000101 | subr x12, x5, x7 | x12 = 0x64 - 0x38 = 0x2C |
| 11 | 1100011111000101 | subr x12, x12, x7 | x12 = 0x2C - 0x38 = 0xFFF4 = -0xC |

In the simulation, the instructions above were placed into the instruction memory. The resulting simulation waveform can be seen below.
The simulation waveform was observed to match the expected results in the above table. Thus, the `subr` instruction was confirmed to function correctly.


Monitoring register x9
x9 = 0xB54
x12 = 0x2C
Monitoring register x12
x12 = 0xFFF4 = -0xC
PC incremented by 1
Result of the ALU operation is 0xB54
ALUOp = 5, which indicates a SUB.
ALU Input A is always the value in RS1 (RO1). Then, the value in RS2 (RO2) is selected to be ALU Input B.
RS1 = x6
RS2 = x5
RD = x9
RS1 = x5
RS2 = x7
RD = x12
RS1 = x12
RS2 = x7
RD = x12

## 2.2 Immediate Instructions

There are 3 I-Type instructions to be tested:

1. Add Immediate: `addi RD, Immediate`
    - RD = RD + Immediate
2. Move Upper: `movu RD, Immediate`
    - RD[15:8] = Immediate
3. Move Lower: `movl RD, Immediate`
    - RD[7:0] = Immediate

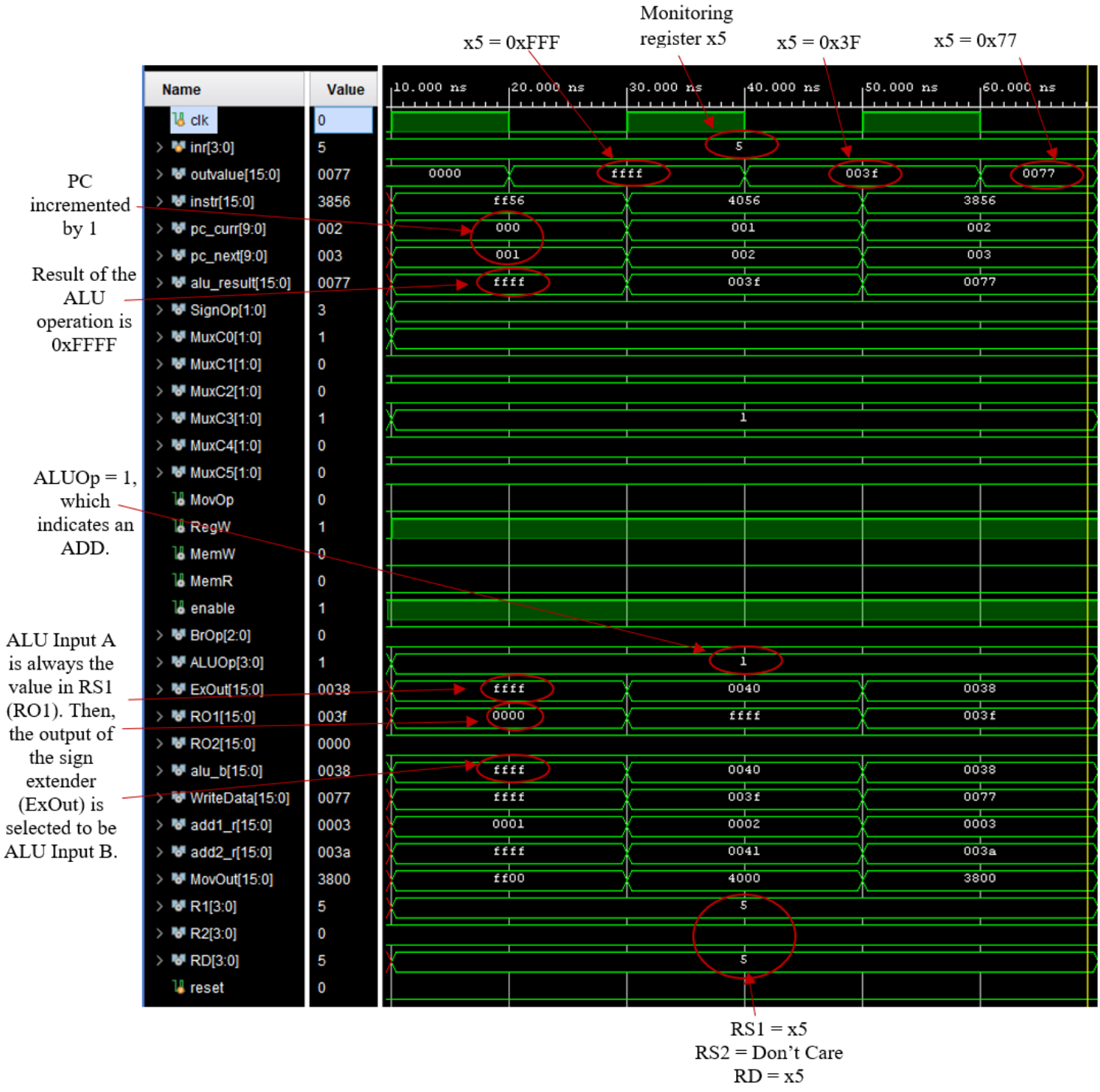
All registers are initialized to a value of 0.
The simulation results for each immediate instruction can be seen below.

### 2.2.1 addi Instruction

Instructions used in simulation:

| Instr. Addr. | Binary | Instr. | Expected Result |
|---|---|---|---|
| 0 | 1111111101010110 | addi x5, -1 | x5 = 0 + -1 = -1 = 0xFFFF |
| 1 | 0100000001010110 | addi x5, 64 | x5 = -1 + 64 = 63 = 0x3F |
| 2 | 0011100001010110 | addi x5, 56 | x5 = 63 + 56 = 119 = 0x77 |

In the simulation, the instructions above were placed into the instruction memory. The resulting simulation waveform can be seen below.
The simulation waveform was observed to match the expected results in the above table. Thus, the `addi` instruction was confirmed to function correctly.

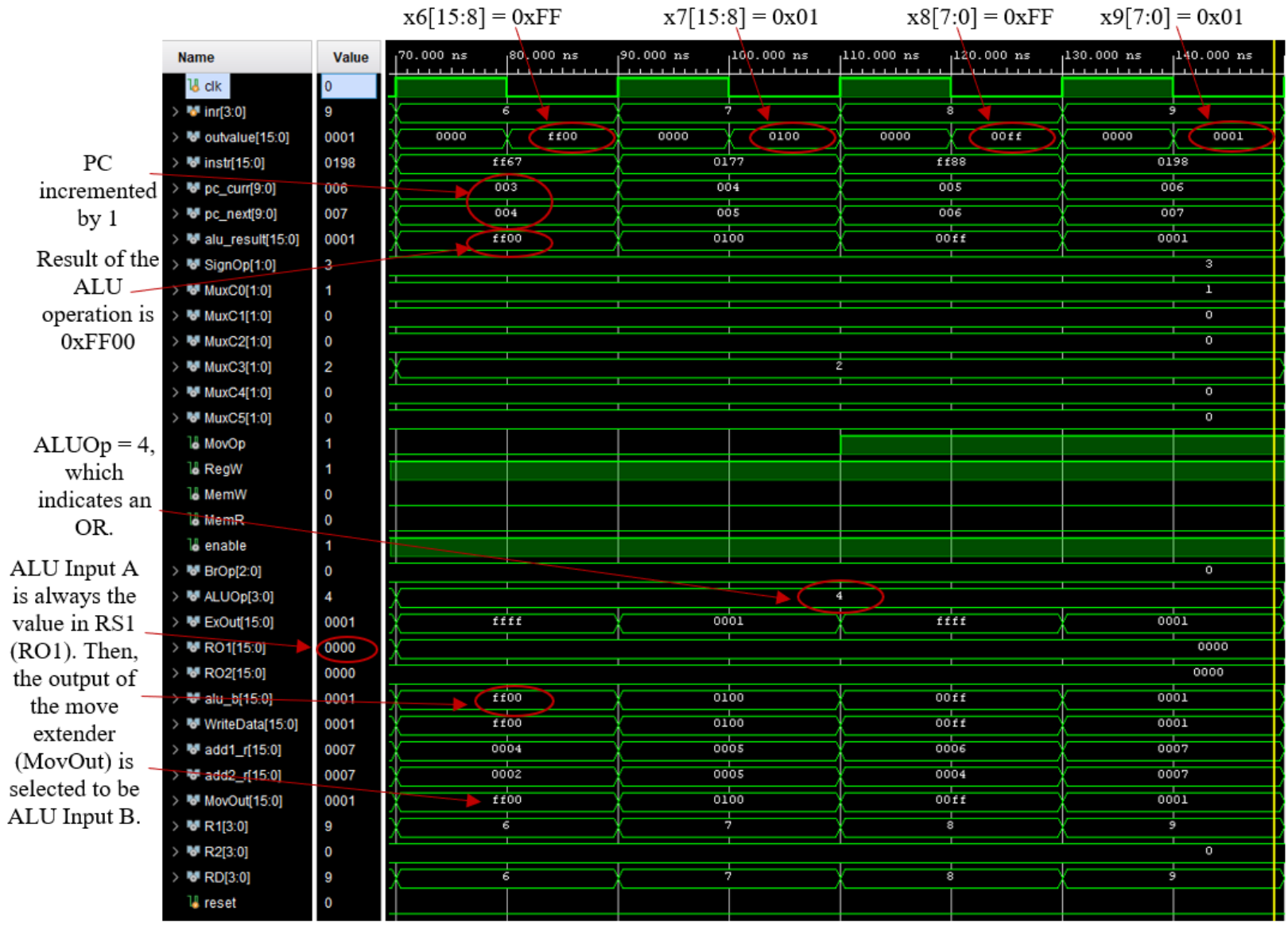

Monitoring register x5
x5 = 0xFFF
x5 = 0x3F
x5 = 0x77
PC incremented by 1
Result of the ALU operation is 0xFFFF
ALUOp = 1, which indicates an ADD.
ALU Input A is always the value in RS1 (RO1). Then, the output of the sign extender (ExOut) is selected to be ALU Input B.
RS1 = x5
RS2 = Don't Care
RD = x5

### 2.2.2 movu and movl Instructions

Instructions used in simulation:

| Instr. Addr. | Binary | Instr. | Expected Result |
|---|---|---|---|
| 3 | 1111111101100111 | movu x6, -1 | x6[15:8] = -1 = 0xFF |
| 4 | 0000000101110111 | movu x8, 1 | x8[15:8] = 1 = 0x01 |
| 5 | 1111111110001000 | movl x7, -1 | x7[7:0] = -1 = 0xFF |
| 6 | 0000000110011000 | movl x9, 1 | x9[7:0] = 1 = 0x01 |

In the simulation, the instructions above were placed into the instruction memory. The resulting simulation waveform can be seen below.
The simulation waveform was observed to match the expected results in the above table. Thus, the `movu` and `movl` instructions were confirmed to function correctly.


x6[15:8] = 0xFF
x7[15:8] = 0x01
x8[7:0] = 0xFF
x9[7:0] = 0x01
PC incremented by 1
Result of the ALU operation is 0xFF00
ALUOp = 4, which indicates an OR.
ALU Input A is always the value in RS1 (RO1). Then, the output of the move extender (MovOut) is selected to be ALU Input B.
Name
Value
clk
inr[3:0]
outvalue[15:0]
instr[15:0]
pc_curr[9:0]
pc_next[9:0]
alu_result[15:0]
SignOp[1:0]
MuxC0[1:0]
MuxC1[1:0]
MuxC2[1:0]
MuxC3[1:0]
MuxC4[1:0]
MuxC5[1:0]
MovOp
RegW
MemW
MemR
enable
BrOp[2:0]
ALUOp[3:0]
ExOut[15:0]
RO1[15:0]
RO2[15:0]
alu_b[15:0]
WriteData[15:0]
add1_r[15:0]
add2_r[15:0]
MovOut[15:0]
R1[3:0]
R2[3:0]
RD[3:0]
reset
70.000 ns
80.000 ns
90.000 ns
100.000 ns
110.000 ns
120.000 ns
130.000 ns
140.000 ns

## 2.3 Jump and Return

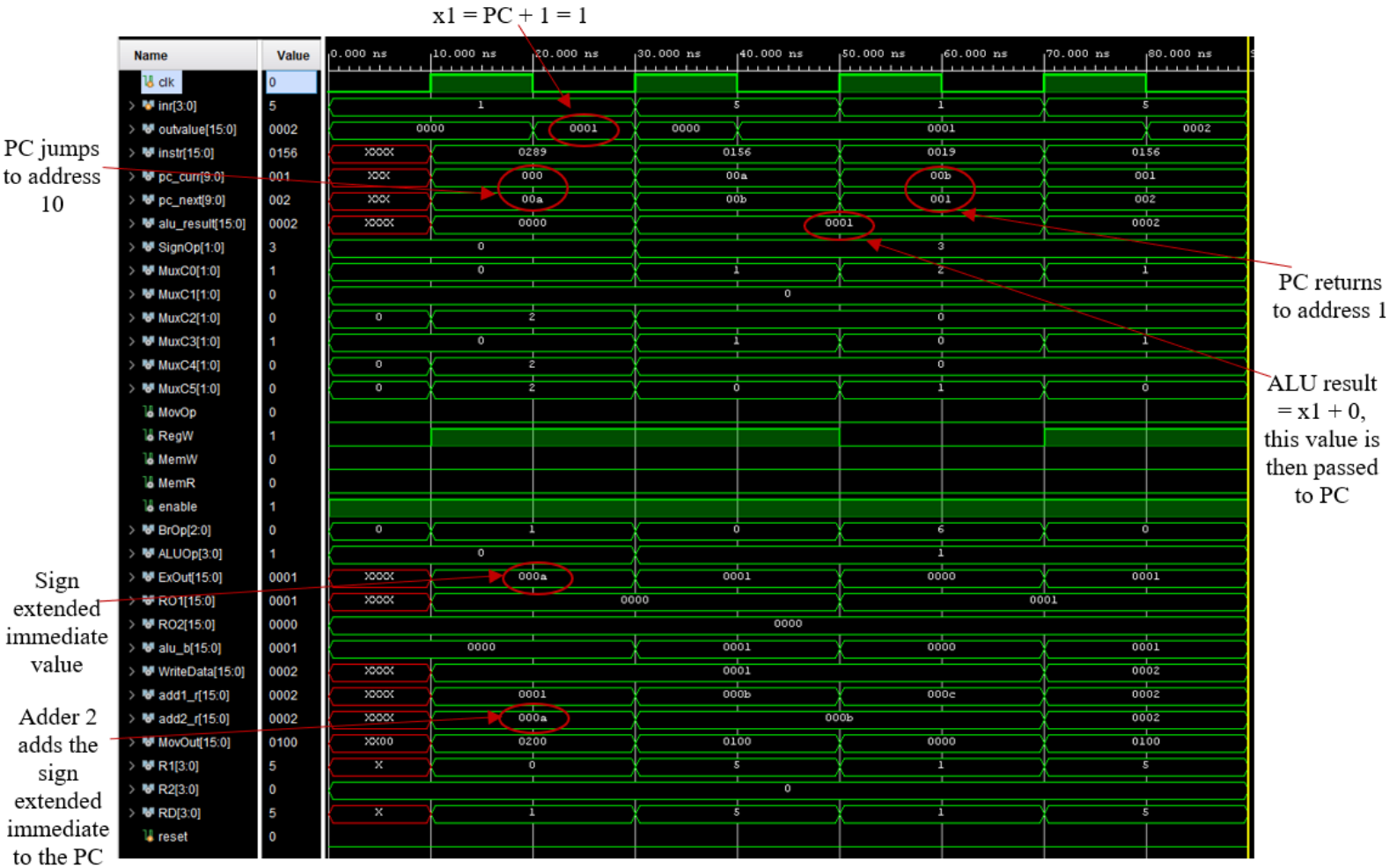
Jump and Return are defined as:

- Jump: `jmp Immediate`
  - x1 = PC + 1
  - PC = PC + Immediate
- Return: `ret`
  - PC = x1

All registers are initialized to a value of 0.
Instructions used in simulation:

| Instr. Addr. | Binary | Instr. | Expected Result |
|---|---|---|---|
| 0 | 0000001010001001 | jmp 10 | PC = PC + 10 = 10, x1 = PC + 1 = 1 |
| 10 | 0000000101010110 | addi x5, 1 | x5 = 0 + 1 = 1 |
| 11 | 0000000000011001 | ret | PC = x1 = 1 |
| 1 | 0000000101010110 | addi x5, 1 | x5 = 1 + 1 = 2 |

In the simulation, the instructions above were placed into the instruction memory. The resulting simulation waveform can be seen below.
The simulation waveform was observed to match the expected results in the above table. Thus, the `jmp` and `ret` instructions were confirmed to function correctly.


x1 = PC + 1 = 1
PC jumps to address 10
PC returns to address 1
ALU result = x1 + 0, this value is then passed to PC
Sign extended immediate value
Adder 2 adds the sign extended immediate to the PC
Name
Value
clk
inr[3:0]
outvalue[15:0]
instr[15:0]
pc_curr[9:0]
pc_next[9:0]
alu_result[15:0]
SignOp[1:0]
MuxC0[1:0]
MuxC1[1:0]
MuxC2[1:0]
MuxC3[1:0]
MuxC4[1:0]
MuxC5[1:0]
MovOp
RegW
MemW
MemR
enable
BrOp[2:0]
ALUOp[3:0]
ExOut[15:0]
RO1[15:0]
RO2[15:0]
alu_b[15:0]
WriteData[15:0]
add1_r[15:0]
add2_r[15:0]
MovOut[15:0]
R1[3:0]
R2[3:0]
RD[3:0]
reset

## 2.4 Shift Instructions

There are 4 shift instructions to be tested:

1. Logical Left Shift: `shl RS1, Immediate`
    - RS1 << Immediate
2. Arithmetic Left Shift: `sal RS1, Immediate`
    - RS1 <<< Immediate
3. Logical Right Shift: `shr RS1, Immediate`
    - RS1 >> Immediate
4. Arithmetic Right Shift: `sar RS1, Immediate`
    - RS1 >>> Immediate

The register file has been initialized to the following values for testing purposes:

- x0-x4, x13-x15 = 0
- x5-x8 = 60
- x9-x12 = -60

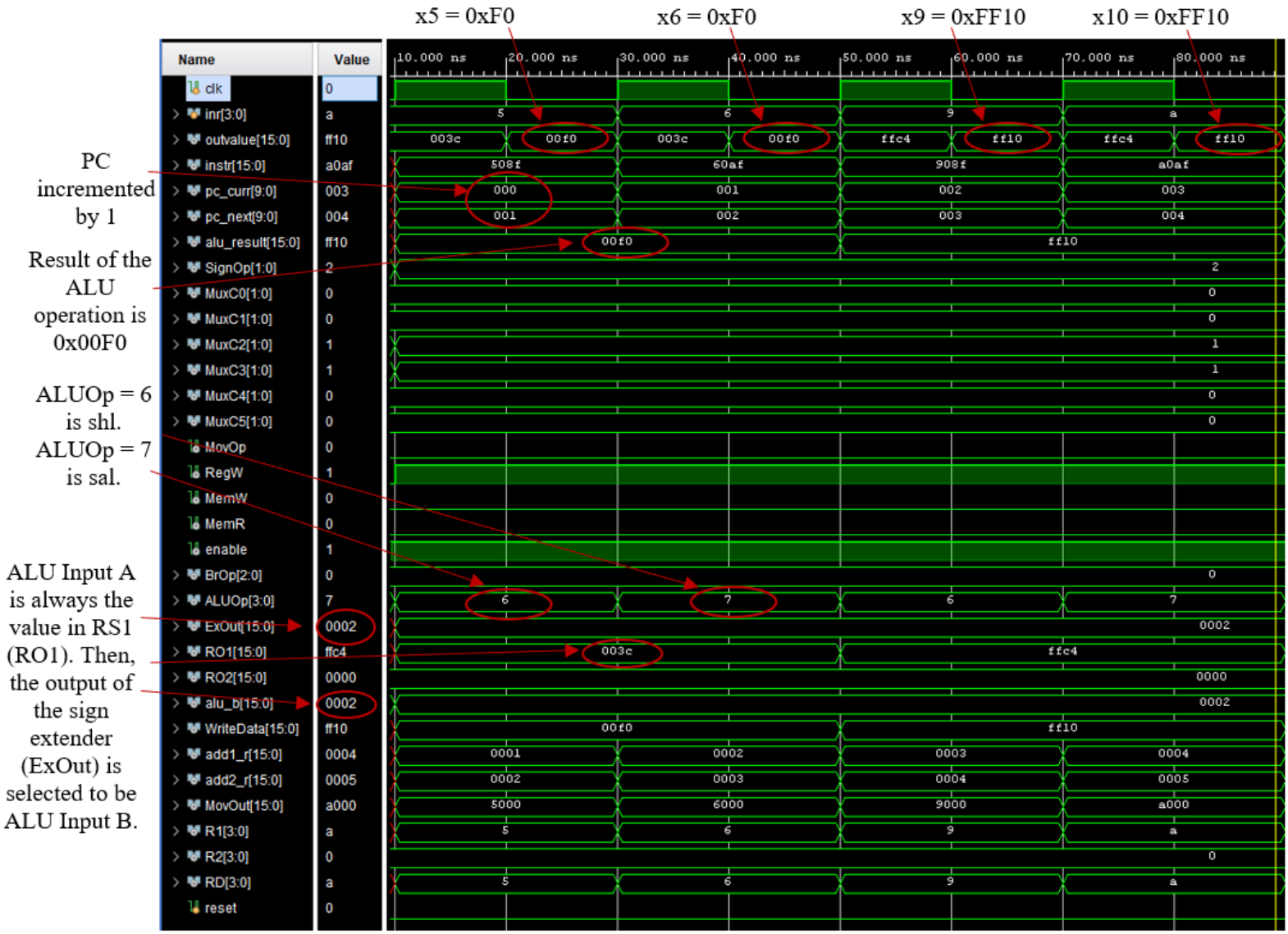
The resulting simulation waveform for each shift instruction can be seen in the sections below.

### 2.4.1 shl and sal Instructions

Instructions used in simulation:

| Instr. Addr. | Binary | Instr. | Expected Result |
|---|---|---|---|
| 0 | 0101000010001111 | shl x5, 2 | x5 = 60 x 4 = 240 = 0xF0 |
| 1 | 0110000010101111 | sal x6, 2 | x6 = 60 x 4 = 240 = 0xF0 |
| 2 | 1001000010001111 | shl x9, 2 | x9 = -60 << 2 = 0xFF10 |
| 3 | 1010000010101111 | sal x10, 2 | x10 = -60 x 4 = -240 = 0xFF10 |

In the simulation, the instructions above were placed into the instruction memory. The resulting simulation waveform can be seen below.
The simulation waveform was observed to match the expected results in the above table. Thus, the `shl` and `sal` instructions were confirmed to function correctly.

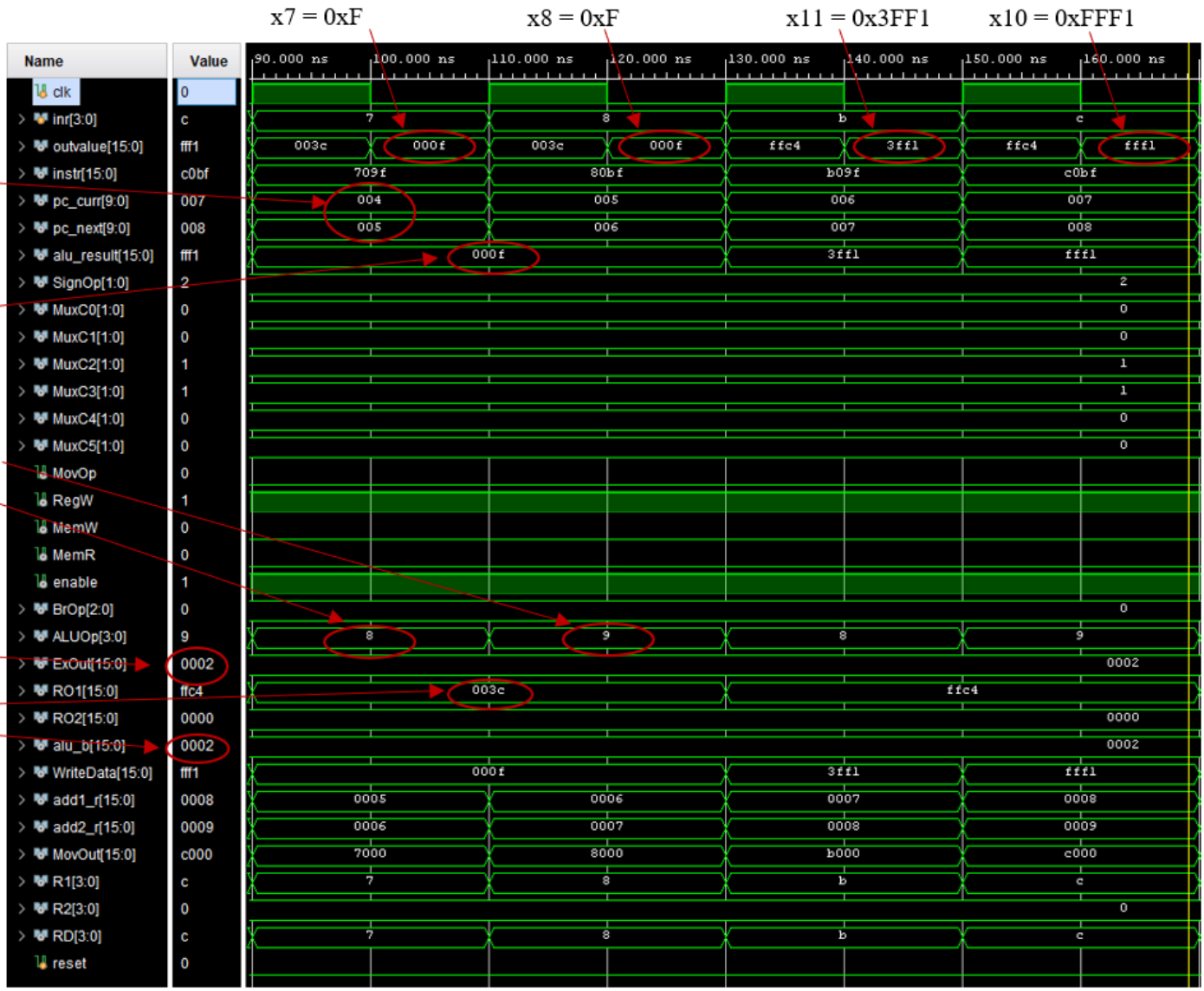


### 2.4.2 shr and sar Instructions

Instructions used in simulation:

| Instr. Addr. | Binary | Instr. | Expected Result |
|---|---|---|---|
| 0 | 0101000010001111 | shr x7, 2 | x7 = 60/4 = 15 = 0xF |
| 1 | 0110000010101111 | sar x8, 2 | x8 = 60/4 = 15 = 0xF |
| 2 | 1001000010001111 | shr x11, 2 | x11 = -60 >> 2 = 0x3FF1 |
| 3 | 1010000010101111 | sar x12, 2 | x12 = -60/4 = -15 = 0xFFF1 |

In the simulation, the instructions above were placed into the instruction memory. The resulting simulation waveform can be seen below.
The simulation waveform was observed to match the expected results in the above table. Thus, the `shr` and `sar` instructions were confirmed to function correctly.


x7 = 0xF
x8 = 0xF
x11 = 0x3FF1
x10 = 0xFFF1
PC incremented by 1
Result of the ALU operation is 0x000F
ALUOp = 8 is shr.
ALUOp = 9 is sar.
ALU Input A is always the value in RS1 (RO1). Then, the output of the sign extender (ExOut) is selected to be ALU Input B.
Name
Value
90.000 ns
100.000 ns
110.000 ns
120.000 ns
130.000 ns
140.000 ns
150.000 ns
160.000 ns
clk
inr[3:0]
outvalue[15:0]
instr[15:0]
pc_curr[9:0]
pc_next[9:0]
alu_result[15:0]
SignOp[1:0]
MuxC0[1:0]
MuxC1[1:0]
MuxC2[1:0]
MuxC3[1:0]
MuxC4[1:0]
MuxC5[1:0]
MovOp
RegW
MemW
MemR
enable
BrOp[2:0]
ALUOp[3:0]
ExOut[15:0]
RO1[15:0]
RO2[15:0]
alu_b[15:0]
WriteData[15:0]
add1_r[15:0]
add2_r[15:0]
MovOut[15:0]
R1[3:0]
R2[3:0]
RD[3:0]
reset

## 2.5 Branches

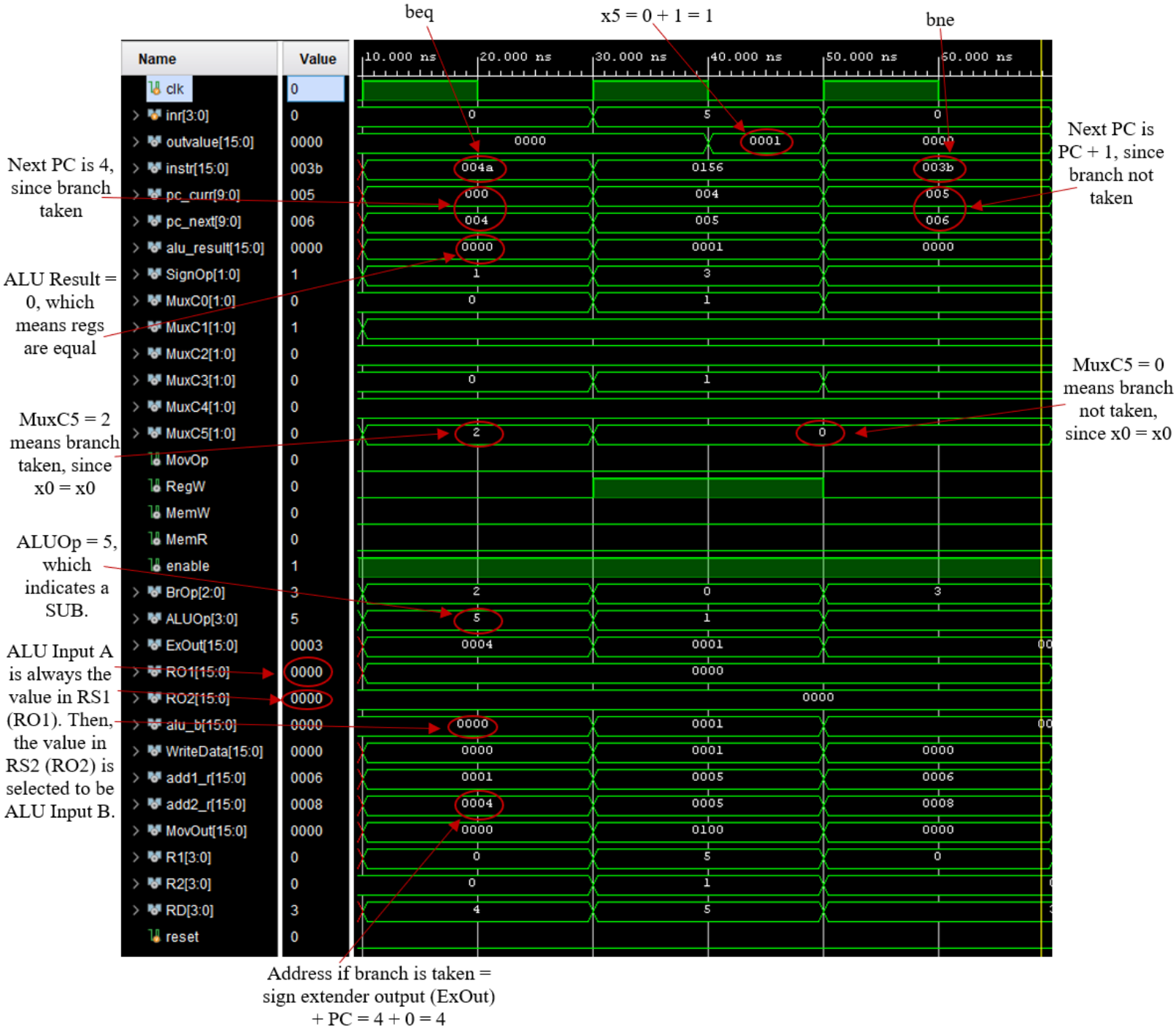
There are 4 branch instructions to be tested:

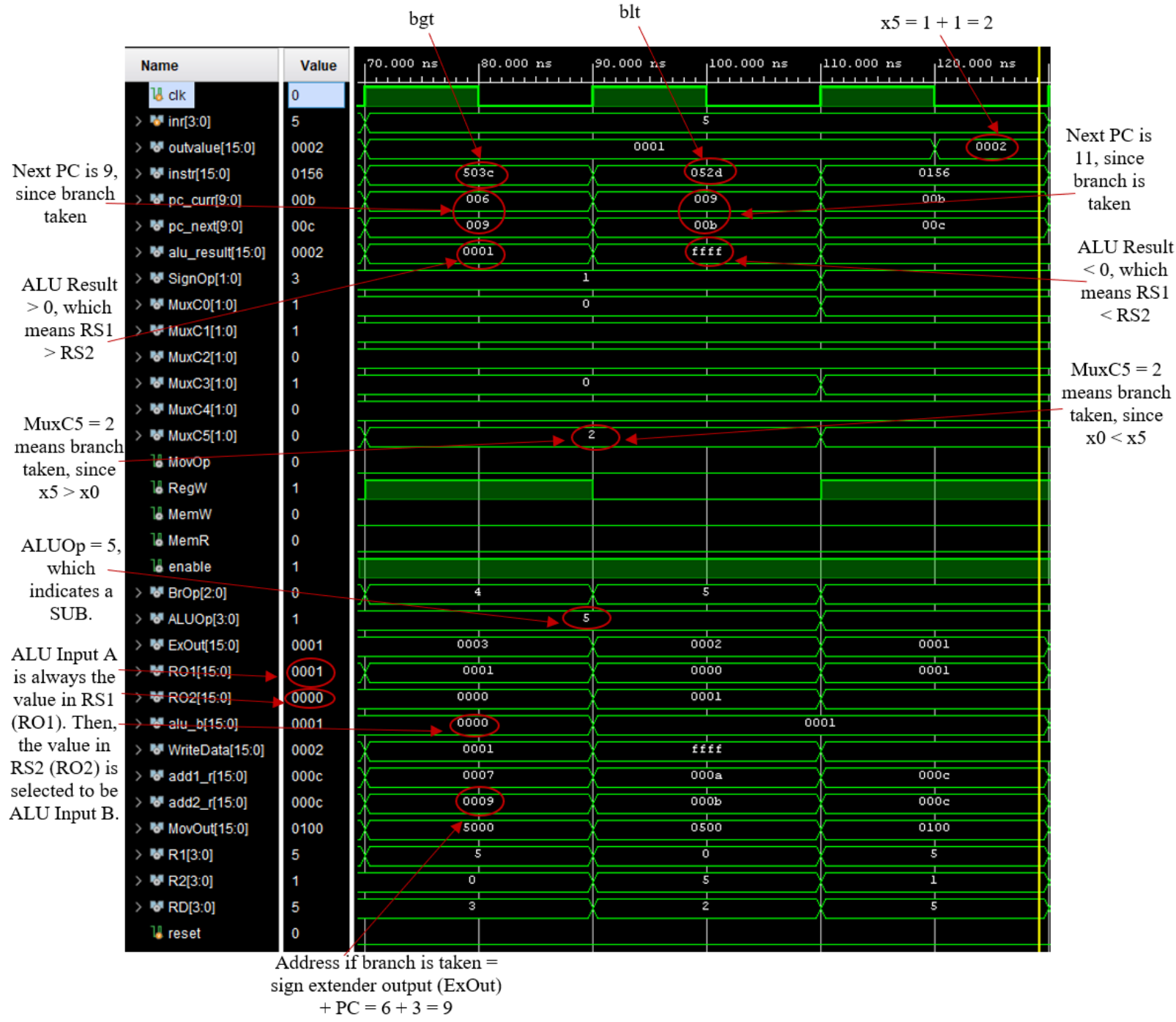
1. Branch Equal: `beq RS1, RS2, Immediate`
   - If RS1 = RS2, PC = PC + Immediate
   - Else, PC = PC + 1
2. Branch Not Equal: `bne RS1, RS2, Immediate`
   - If RS1 != RS2, PC = PC + Immediate
   - Else, PC = PC + 1
3. Branch Greater Than: `bgt RS1, RS2, Immediate`
   - If RS1 > RS2, PC = PC + Immediate
   - Else, PC = PC + 1
4. Branch Less Than: `blt RS1, RS2, Immediate`
   - If RS1 < RS2, PC = PC + Immediate
   - Else, PC = PC + 1

All registers are initialized to a value of 0.
Instructions used in simulation:

| Instr. Addr. | Binary | Instr. | Expected Result |
|---|---|---|---|
| 0 | 0000000001001010 | beq x0, x0, 4 | Branch taken, PC = PC + 4 = 4 |
| 4 | 0000000101010110 | addi x5, 1 | x5 = 0 + 1 = 1 |
| 5 | 0000000000111011 | bne x0, x0, 3 | Branch not taken, PC = PC + 1 = 6 |
| 6 | 0101000000111100 | bgt x5, x0, 3 | Branch taken, PC = PC + 3 = 9 |
| 9 | 0000010100101101 | blt x0, x5, 2 | Branch taken, PC = PC + 2 = 11 |
| 11 | 0000000101010110 | addi x5, 1 | x5 = 1 + 1 = 2 |

In the simulation, the instructions above were placed into the instruction memory. The resulting simulation waveform can be seen below.
The simulation waveform was observed to match the expected results in the above table. Thus, the `beq`, `bne`, `bgt`, and `blt` instructions were confirmed to function correctly.


beq
x5 = 0 + 1 = 1
bne
Name
Value
clk
inr[3:0]
outvalue[15:0]
instr[15:0]
pc_curr[9:0]
pc_next[9:0]
alu_result[15:0]
SignOp[1:0]
MuxC0[1:0]
MuxC1[1:0]
MuxC2[1:0]
MuxC3[1:0]
MuxC4[1:0]
MuxC5[1:0]
MovOp
RegW
MemW
MemR
enable
BrOp[2:0]
ALUOp[3:0]
ExOut[15:0]
RO1[15:0]
RO2[15:0]
alu_b[15:0]
WriteData[15:0]
add1_r[15:0]
add2_r[15:0]
MovOut[15:0]
R1[3:0]
R2[3:0]
RD[3:0]
reset
10.000 ns
20.000 ns
30.000 ns
40.000 ns
50.000 ns
60.000 ns
Next PC is 4, since branch taken
ALU Result = 0, which means regs are equal
MuxC5 = 2 means branch taken, since x0 = x0
ALUOp = 5, which indicates a SUB.
ALU Input A is always the value in RS1 (RO1). Then, the value in RS2 (RO2) is selected to be ALU Input B.
Next PC is PC + 1, since branch not taken
MuxC5 = 0 means branch not taken, since x0 = x0
Address if branch is taken = sign extender output (ExOut) + PC = 4 + 0 = 4


bgt
blt
x5 = 1 + 1 = 2
Name
Value
70.000 ns
80.000 ns
90.000 ns
100.000 ns
110.000 ns
120.000 ns
clk
inr[3:0]
outvalue[15:0]
instr[15:0]
pc_curr[9:0]
pc_next[9:0]
alu_result[15:0]
SignOp[1:0]
MuxC0[1:0]
MuxC1[1:0]
MuxC2[1:0]
MuxC3[1:0]
MuxC4[1:0]
MuxC5[1:0]
MovOp
RegW
MemW
MemR
enable
BrOp[2:0]
ALUOp[3:0]
ExOut[15:0]
RO1[15:0]
RO2[15:0]
alu_b[15:0]
WriteData[15:0]
add1_r[15:0]
add2_r[15:0]
MovOut[15:0]
R1[3:0]
R2[3:0]
RD[3:0]
reset
Next PC is 9, since branch taken
ALU Result > 0, which means RS1 > RS2
MuxC5 = 2 means branch taken, since x5 > x0
ALUOp = 5, which indicates a SUB.
ALU Input A is always the value in RS1 (RO1). Then, the value in RS2 (RO2) is selected to be ALU Input B.
Address if branch is taken = sign extender output (ExOut) + PC = 6 + 3 = 9
Next PC is 11, since branch is taken
ALU Result < 0, which means RS1 < RS2
MuxC5 = 2 means branch taken, since x0 < x5

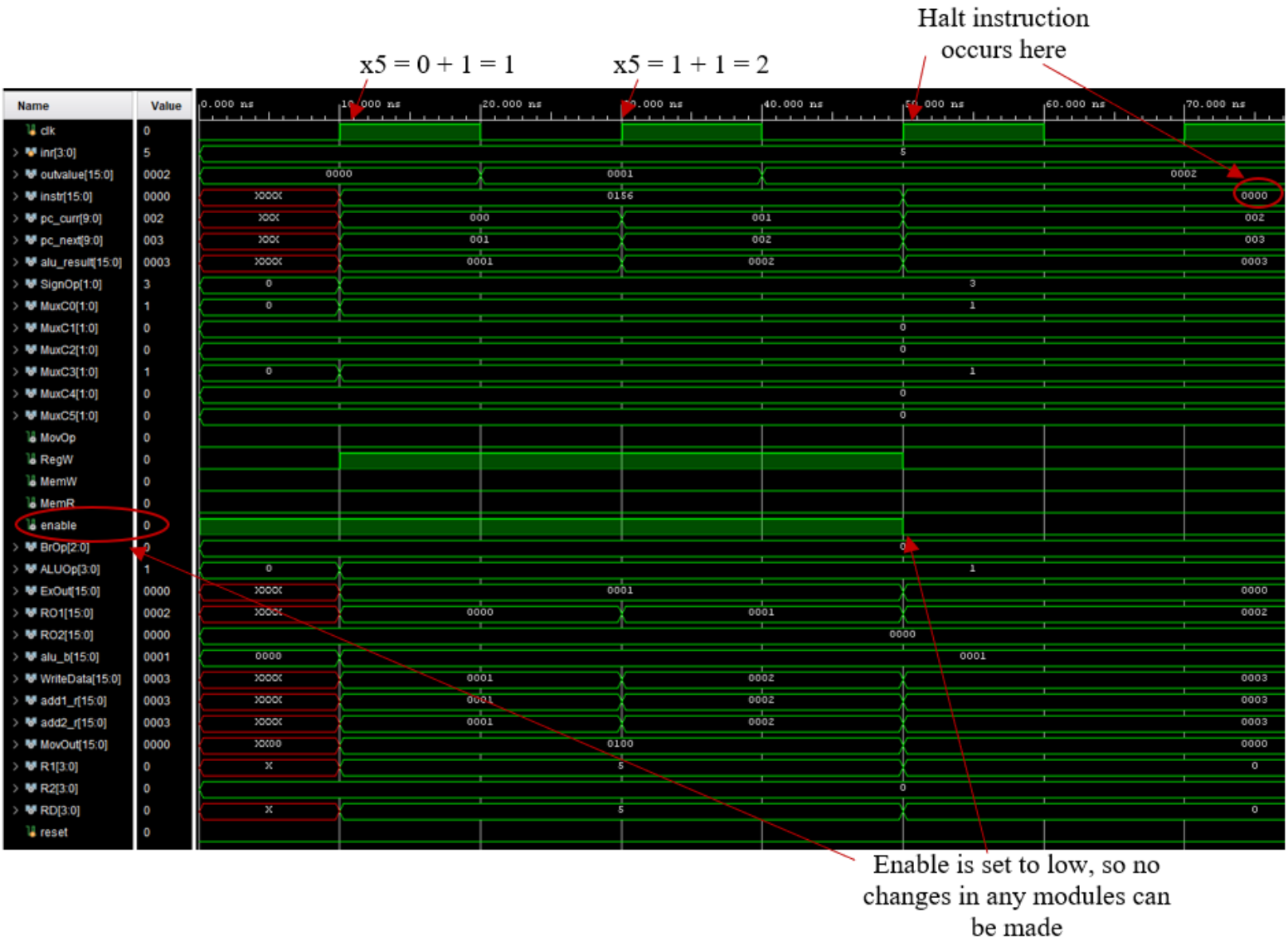
## 2.6 Halt

The `halt` instruction should stop the execution of the program.

All registers are initialized to a value of 0.
Instructions used in simulation:

| Instr. Addr. | Binary | Instr. | Expected Result |
|---|---|---|---|
| 0 | 0000000101010110 | addi x5, 1 | x5 = 0 + 1 = 1 |
| 1 | 0000000101010110 | addi x5, 1 | x5 = 1 + 1 = 2 |
| 2 | 0000000000000000 | halt | Enable = 0, Program stops |

In the simulation, the instructions above were placed into the instruction memory. The resulting simulation waveform can be seen below.
The simulation waveform was observed to match the expected results in the above table. Thus, the `halt` instruction was confirmed to function correctly.


x5 = 0 + 1 = 1
x5 = 1 + 1 = 2
Halt instruction occurs here
Enable is set to low, so no changes in any modules can be made

# 3 Test Program

A short test program including at least one instruction from each type (register, immediate, jump, shift, branch, and halt) was written. The assembly and binary instructions can be seen below.

Assembly:

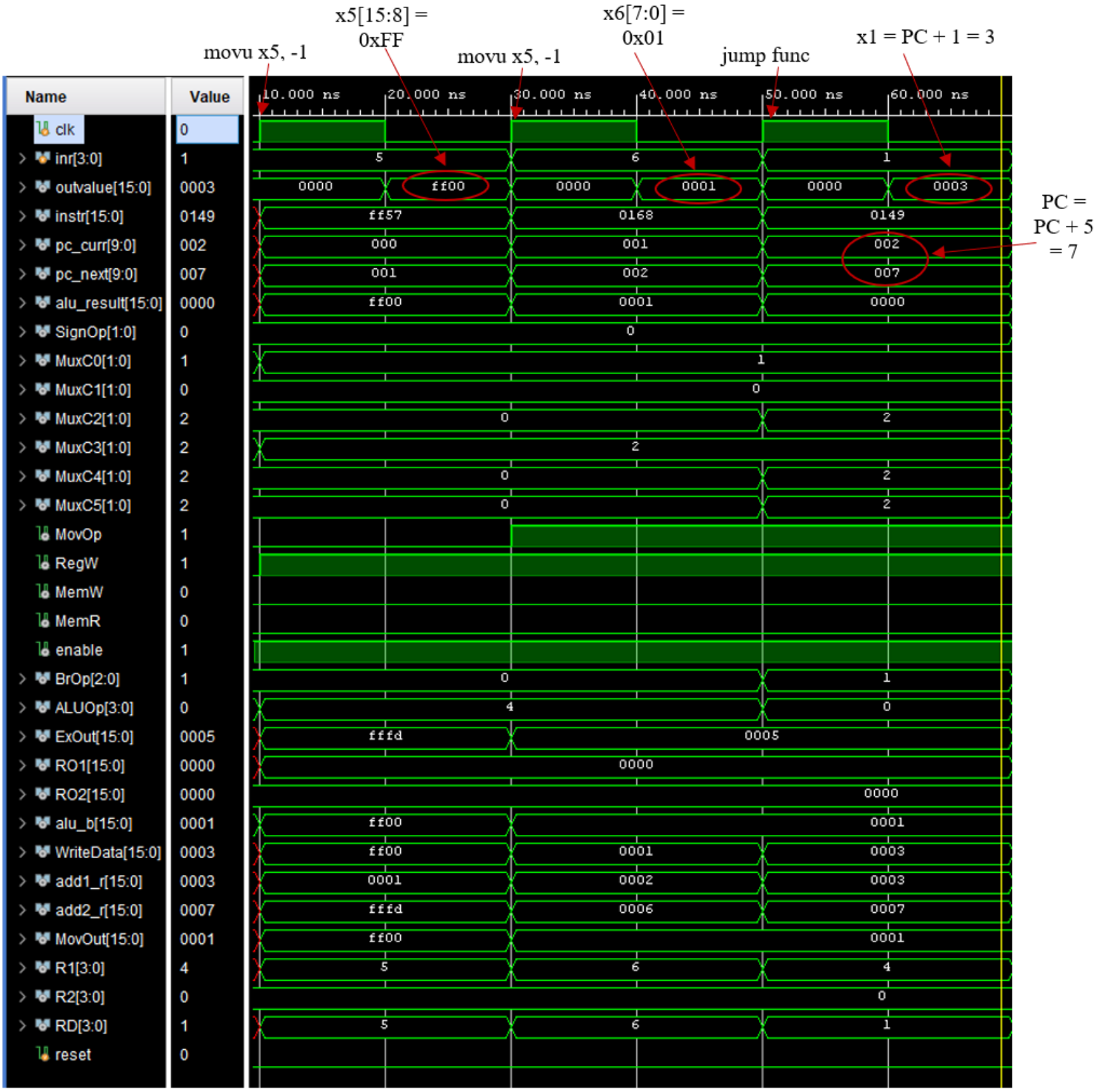
```
movu x5, -1
movl x6, 1
jump func
halt



func:  addi x5, 1
       bgt x6, x0, skip
       addr x6, x0, x0
skip:  addr x7, x6, x5
       andr x7, x5
       shl x5, 2
       ret
```

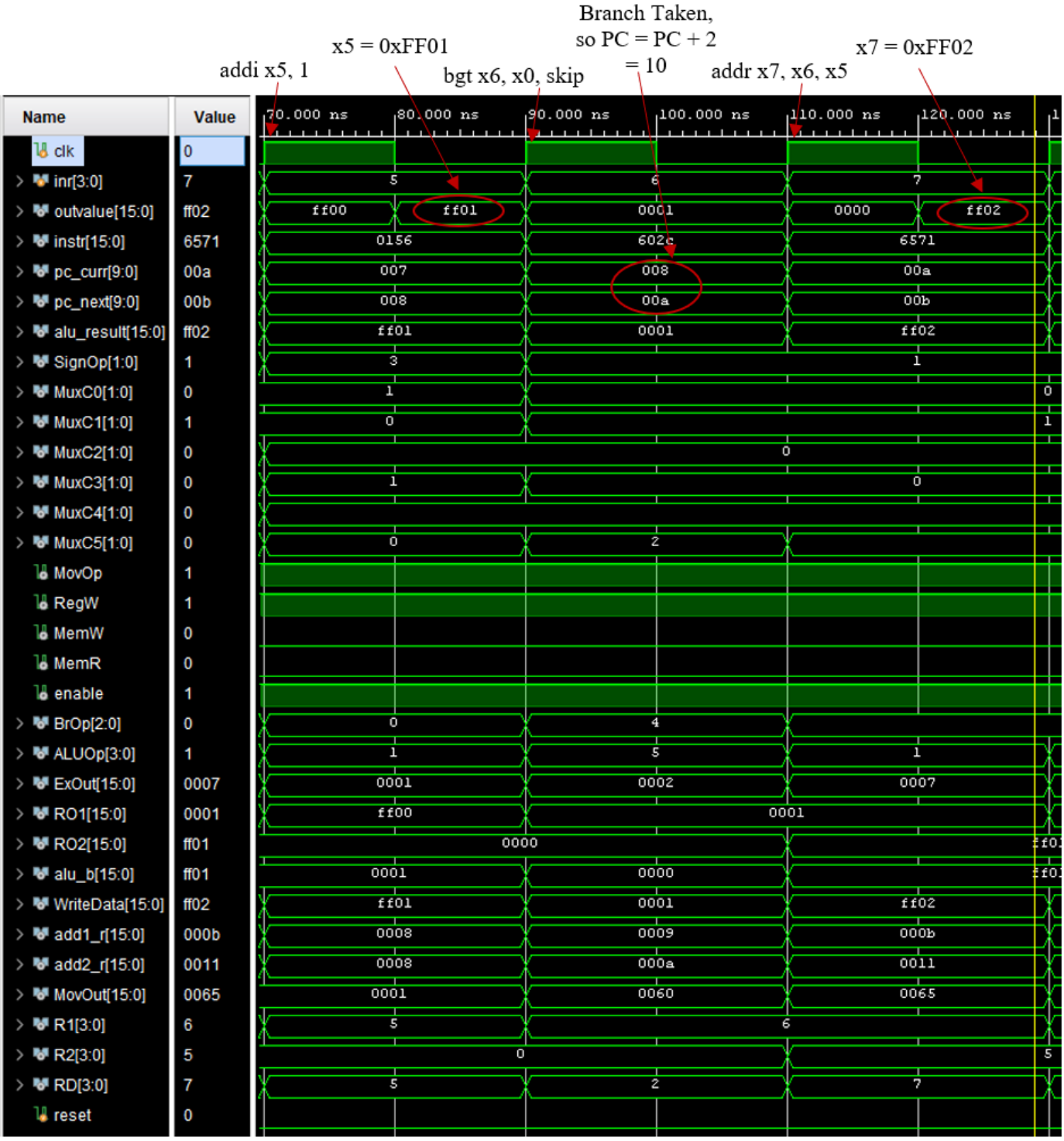
Machine Code:

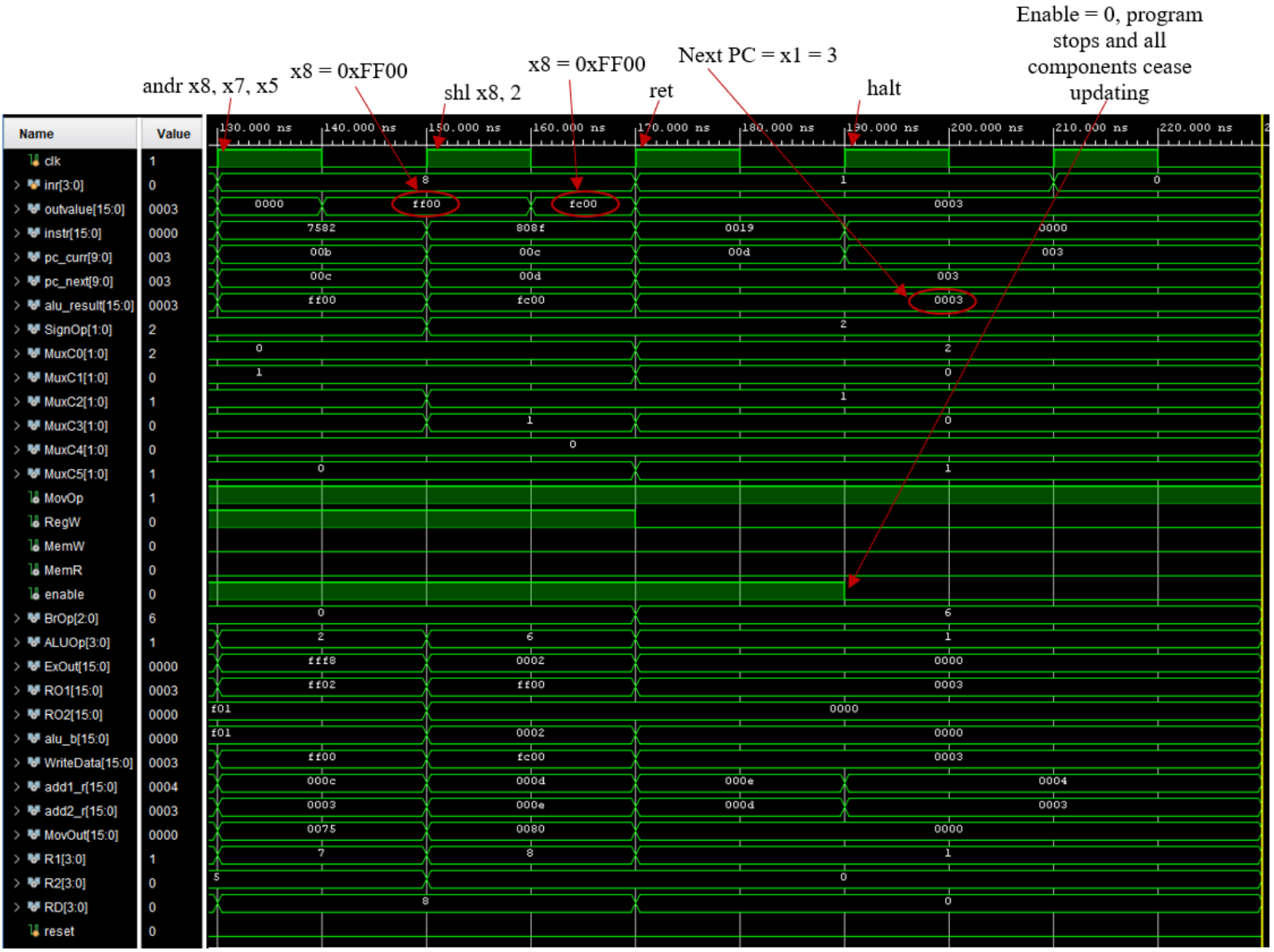
```
1111111101010111
0000000101101000
0000000101001001
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000101010110
0110000000101100
0000000001100001
0110010101110001
0111010110000010
1000000010001111
0000000000011001
```

| Instr. Addr. | Binary | Instr. | Expected Result |
|---|---|---|---|
| 0 | 1111111101010111 | movu x5, -1 | x5[15:8] = 0xFF |
| 1 | 0000000101101000 | movu x6, 1 | x6[7:0] = 0x01 |
| 2 | 0000000101001001 | jmp func | PC = PC + 5 = 7, x1 = PC + 1 = 3 |
| 7 | 0000000101010110 | addi x5, 1 | x5 = x5 + 1 = 0xFF01 |
| 8 | 0110000000101100 | bgt x6, x0, skip | Branch Taken, PC = PC + 2 = 10 |
| 10 | 0110010101110001 | addr x7, x6, x5 | x8 = 0x0001 + 0xFF01 = 0xFF02 |
| 11 | 0111010110000010 | andr x8, x7, x5 | x8 = 0xFF02 & 0xFF01 = 0xFF00 |
| 12 | 1000000010001111 | shl x8, 2 | x8 = 0xFF00 << 2 = 0xFC00 |
| 13 | 0000000000011001 | ret | PC = x1 = 3 |
| 3 | 0000000000000000 | halt | Enable = 0, Program stops |

In the simulation, the instructions above were placed into the instruction memory. The resulting simulation waveform can be seen below.
The simulation waveform was observed to match the expected results in the above table. Thus, the program was confirmed to function correctly.


movu x5, -1
x5[15:8] = 0xFF
movu x5, -1
x6[7:0] = 0x01
jump func
x1 = PC + 1 = 3
PC = PC + 5 = 7
Name
Value
10.000 ns
20.000 ns
30.000 ns
40.000 ns
50.000 ns
60.000 ns
clk 0
inr[3:0] 1
outvalue[15:0] 0003
instr[15:0] 0149
pc_curr[9:0] 002
pc_next[9:0] 007
alu_result[15:0] 0000
SignOp[1:0] 0
MuxC0[1:0] 1
MuxC1[1:0] 0
MuxC2[1:0] 2
MuxC3[1:0] 2
MuxC4[1:0] 2
MuxC5[1:0] 2
MovOp 1
RegW 1
MemW 0
MemR 0
enable 1
BrOp[2:0] 1
ALUOp[3:0] 0
ExOut[15:0] 0005
RO1[15:0] 0000
RO2[15:0] 0000
alu_b[15:0] 0001
WriteData[15:0] 0003
add1_r[15:0] 0003
add2_r[15:0] 0007
MovOut[15:0] 0001
R1[3:0] 4
R2[3:0] 0
RD[3:0] 1
reset 0


addi x5, 1
x5 = 0xFF01
bgt x6, x0, skip
Branch Taken,
so PC = PC + 2
= 10
addr x7, x6, x5
x7 = 0xFF02
Name
Value
70.000 ns
80.000 ns
90.000 ns
100.000 ns
110.000 ns
120.000 ns
clk 0
inr[3:0] 7
outvalue[15:0] ff02
instr[15:0] 6571
pc_curr[9:0] 00a
pc_next[9:0] 00b
alu_result[15:0] ff02
SignOp[1:0] 1
MuxC0[1:0] 0
MuxC1[1:0] 1
MuxC2[1:0] 0
MuxC3[1:0] 0
MuxC4[1:0] 0
MuxC5[1:0] 0
MovOp 1
RegW 1
MemW 0
MemR 0
enable 1
BrOp[2:0] 0
ALUOp[3:0] 1
ExOut[15:0] 0007
RO1[15:0] 0001
RO2[15:0] ff01
alu_b[15:0] ff01
WriteData[15:0] ff02
add1_r[15:0] 000b
add2_r[15:0] 0011
MovOut[15:0] 0065
R1[3:0] 6
R2[3:0] 5
RD[3:0] 7
reset 0


andr x8, x7, x5
x8 = 0xFF00
shl x8, 2
x8 = 0xFF00
ret
Next PC = x1 = 3
halt
Enable = 0, program stops and all components cease updating
Name
Value
130.000 ns
140.000 ns
150.000 ns
160.000 ns
170.000 ns
180.000 ns
190.000 ns
200.000 ns
210.000 ns
220.000 ns
clk 1
inr[3:0] 0
outvalue[15:0] 0003
instr[15:0] 0000
pc_curr[9:0] 003
pc_next[9:0] 003
alu_result[15:0] 0003
SignOp[1:0] 2
MuxC0[1:0] 2
MuxC1[1:0] 0
MuxC2[1:0] 1
MuxC3[1:0] 0
MuxC4[1:0] 0
MuxC5[1:0] 1
MovOp 1
RegW 0
MemW 0
MemR 0
enable 0
BrOp[2:0] 6
ALUOp[3:0] 1
ExOut[15:0] 0000
RO1[15:0] 0003
RO2[15:0] 0000
alu_b[15:0] 0000
WriteData[15:0] 0003
add1_r[15:0] 0004
add2_r[15:0] 0003
MovOut[15:0] 0000
R1[3:0] 1
R2[3:0] 0
RD[3:0] 0
reset 0

# Appendix A Top-level Component Code

```
module control_and_datapath(clk, reset, inr, readBus, outvalue, result,
                            MemR, MemW, RO2);
//readBus will contain data read from data memory
//result, MemR, MemW, and RO2 will be outputted to data memory
    input clk;
    input [3:0] inr;
    input reset;
    input [15:0] readBus;
    output [15:0] outvalue, RO2, result;
    output MemW, MemR;

    wire [15:0] pc_in, pc_out;
    wire [15:0] instr, ExOut, RO1, alu_b, WriteData, add1_r, add2_r, MovOut,
                R1, R2, RD;
    wire [1:0] SignOp, MuxC0, MuxC1, MuxC2, MuxC3, MuxC4, MuxC5;
    wire MovOp, RegW, enable;
    wire [2:0] BrOp;
    wire [3:0] ALUOp;

    pc PC (enable, reset, clk, pc_in, pc_out);
    adder ADD1 (enable, reset, 1, pc_out, add1_r);

    instr_mem INSTR (enable, pc_out[9:0], instr);

    control CON (reset, instr[3:0], instr[4], instr[9:8], instr[5:4], RegW,
                 MemR, MemW, enable, BrOp, ALUOp, MovOp, SignOp, MuxC0,
                 MuxC1, MuxC2, MuxC3, MuxC4);

    mux_4to1 C0 (enable, MuxC0, instr[15:12], instr[7:4], 1, 0, R1);
    mux_4to1 C1 (enable, MuxC1, 0, instr[11:8], instr[15:12],
                 instr[7:4], R2);
    mux_4to1 C2 (enable, MuxC2, instr[7:4], instr[15:12], 1, 0, RD);

    sign_ext SIGN (enable, reset, SignOp, instr[15:6], instr[7:4],
                   instr[11:6], instr[15:8], ExOut);
    adder ADD2 (enable, reset, ExOut, pc_out, add2_r);
    mov_ext MOV (enable, reset, MovOp, instr[15:8], MovOut);

    reg_file REG (enable, reset, clk, inr, R1[3:0], R2[3:0], RD[3:0],
                  RegW, WriteData, RO1, RO2, outvalue);
```

```
41
42      mux_4to1 C3 (enable, MuxC3, RO2, ExOut, MovOut, 0, alu_b);
43
44      alu ALU (enable, reset, ALUOp, RO1, alu_b, result, neg, zero, overflow);
45
46      branch_control BRC (reset, BrOp, neg, zero, MuxC5);
47
48      mux_4to1 C4 (enable, MuxC4, result, readBus, add1_r, 0, WriteData);
49      mux_4to1 C5 (enable, MuxC5, add1_r, result, add2_r, 0, pc_in);
50  endmodule
```