VHDL Programming Assignment #3

Architecture of Computers (CSI3102-01), Spring 2020

Welcome to the third programming assignment for the Architecture of Computers (CSI3102-01) course at Yonsei University! Through programming assignments #1 and #2, we believe you are now quite familiar with how to write clock-driven digital circuits.

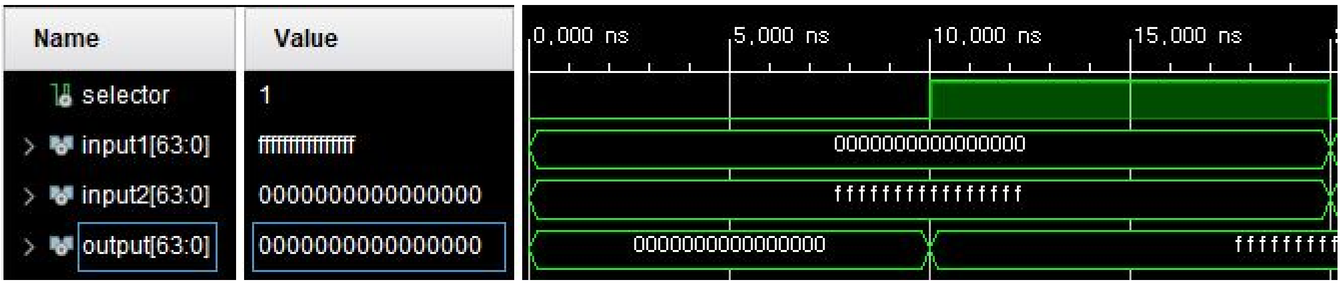
In this programming assignment, you will implement a single-cycle processor which supports a small subset of the RISC-V instructions. As the basic building blocks of the single-cycle processor, you will first implement various combinational circuits which correspond to the components of the single-cycle processor.

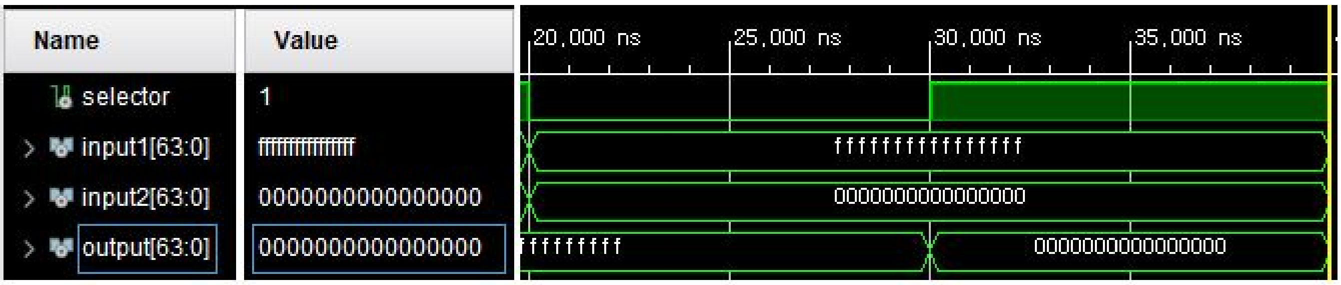
The deadline for this assignment is **11:59pm on May 13th, 2020 (Wed)**. No extension will be given.

1. 2-to-1 Multiplexer (10 Points)

Implement a two-to-one multiplexer by extending `Multiplexer.vhd`. The multiplexer has three input signals: `selector`, `input1`, and `input2`. If the `selector` signal is set to `0`, the multiplexer assigns `input1` to the `output` signal. Otherwise, `input2` is assigned to the `output` signal.

Example:



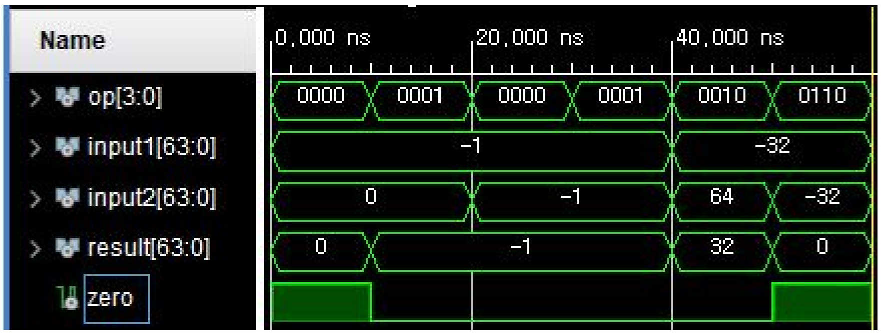


2. Arithmetic Logic Unit (10 Points)

Implement an Arithmetic Logic Unit (ALU) by extending `ArithmeticLogicUnit.vhd`. The ALU has three input signals: `op`, `input1`, and `input2`. The ALU sets its two output signals, `result` and `zero`, with respect to the following table:

`op[3:0]`	Function	result	zero
`0000`	Bitwise AND	`input1[63:0] AND input2[63:0]`	1 (if `result == 0`) `0` (otherwise)
`0001`	Bitwise OR	`input1[63:0] OR input2[63:0]`	
`0010`	Integer addition	`input1[63:0] + input2[63:0]`	
`0110`	Integer subtraction	`input1[63:0] - input2[63:0]`	

Example:



*the values of the ‘op’ signal are shown in binary numbers.
*the values of the ‘input1’, ‘input2’, and ‘result’ signals are shown in signed decimal numbers.

3. Register File (10 Points)

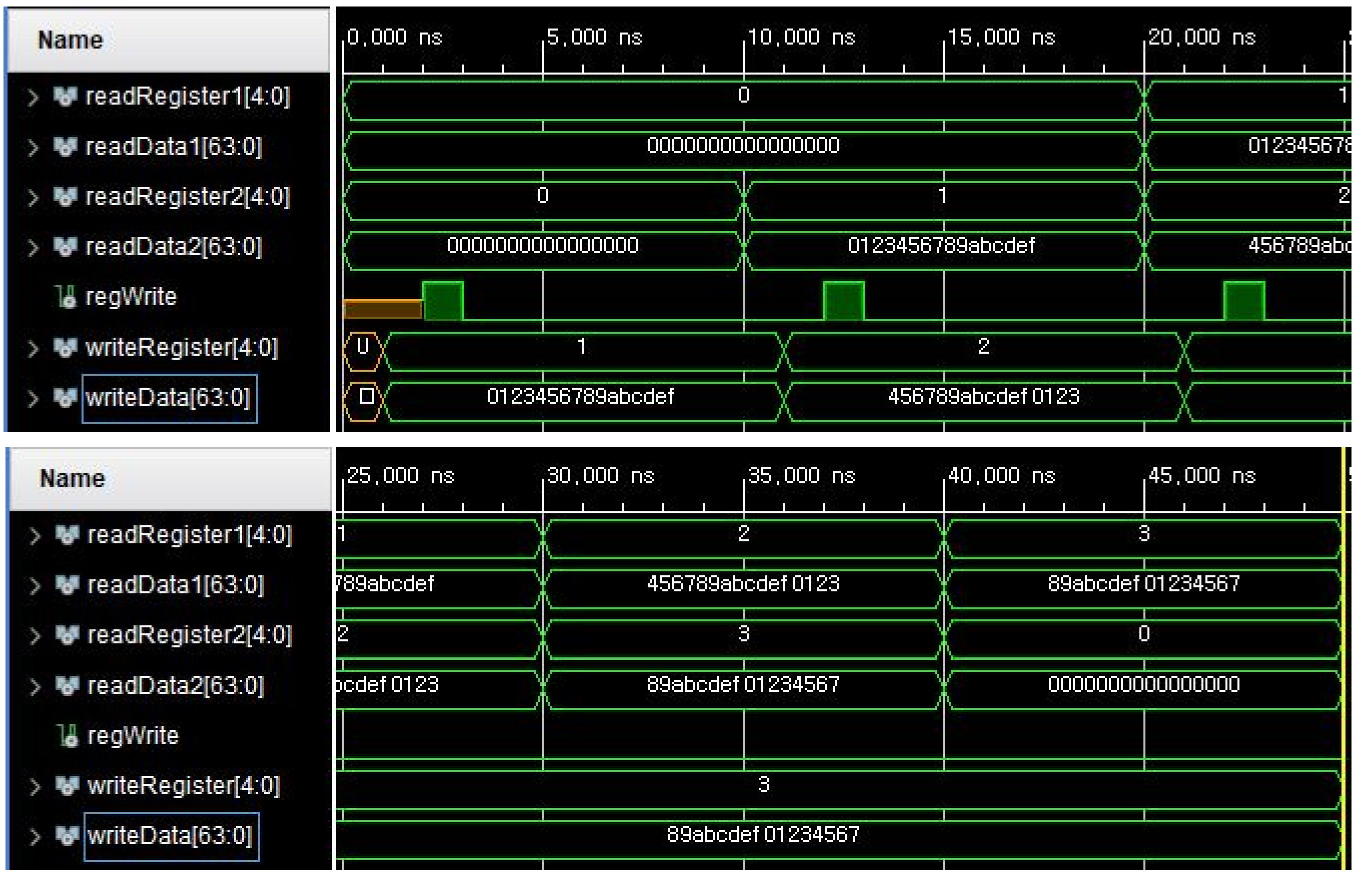
Implement a Register File (RF) as a sequential circuit by extending `RegisterFile.vhd`. The RF should model the integer registers of the RISC-V ISA: have 32 registers named `x0` ~ `x31`, hardware a value of `0` to register `x0`.

The RF has five input ports and two output ports. The input ports are: `readRegister1`, `readRegister2`, `regWrite`, `writeRegister`, `writeData`. The output ports are: `readData1` and `readData2`.

When reading a value from a source register, either `readRegister1` or `readRegister2` is set as the index of the source register. Then, the RF updates the value of the corresponding output signal (i.e., either `readData1` or `readData2`) to be the current value stored in the source register.

When writing a value to a destination register, the `regWrite` signal is set to 1 after assigning the index of the destination register to `writeRegister` and the value to `writeData`. When `regWrite` changes to 1, the RF writes the data (i.e., the value of `writeData`) to the destination register. After the RF writes the value to the destination register, the user of the RF should set `regWrite` back to `0`.

Example:



*the values of readRegister1, readRegister2, and writeRegister are shown in unsigned decimal numbers

**the values of readData1, readData2, and writeData signals are shown in hexadecimal numbers

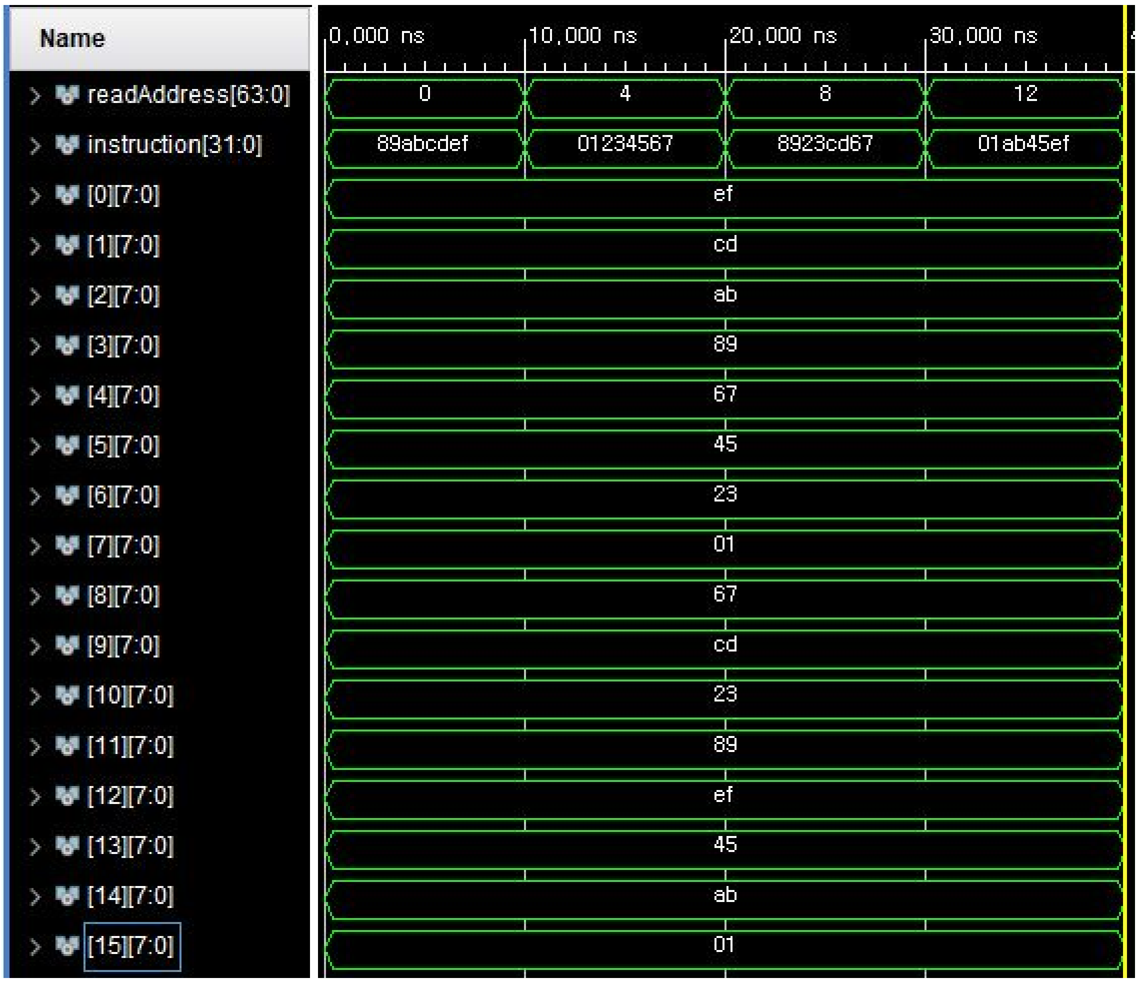
4. Instruction Memory (10 Points)

Implement a little-endian instruction memory by extending `InstructionMemory.vhd`. You may assume that only the byte addresses from `0x0` to `0x3FF` will be accessed. As the instruction memory is little-endian, the least-significant byte of the 4-byte instruction should be placed at the lowest address among the four target byte addresses.

The instruction memory has one input signal named `readAddress`. The instruction memory should update the value of its output signal named `instruction` by assigning the 4-byte value stored in the target memory address to the output signal.

You may assume that the unsigned decimal value of `readAddress` signal is always a multiple of four.

Example:



*The values of readAddress are shown in unsigned decimal numbers
*The values of instruction signal are shown in hexadecimal numbers
*[0] ~ [15] refer to the hexadecimal values stored in memory addresses 0 ~ 15, respectively.

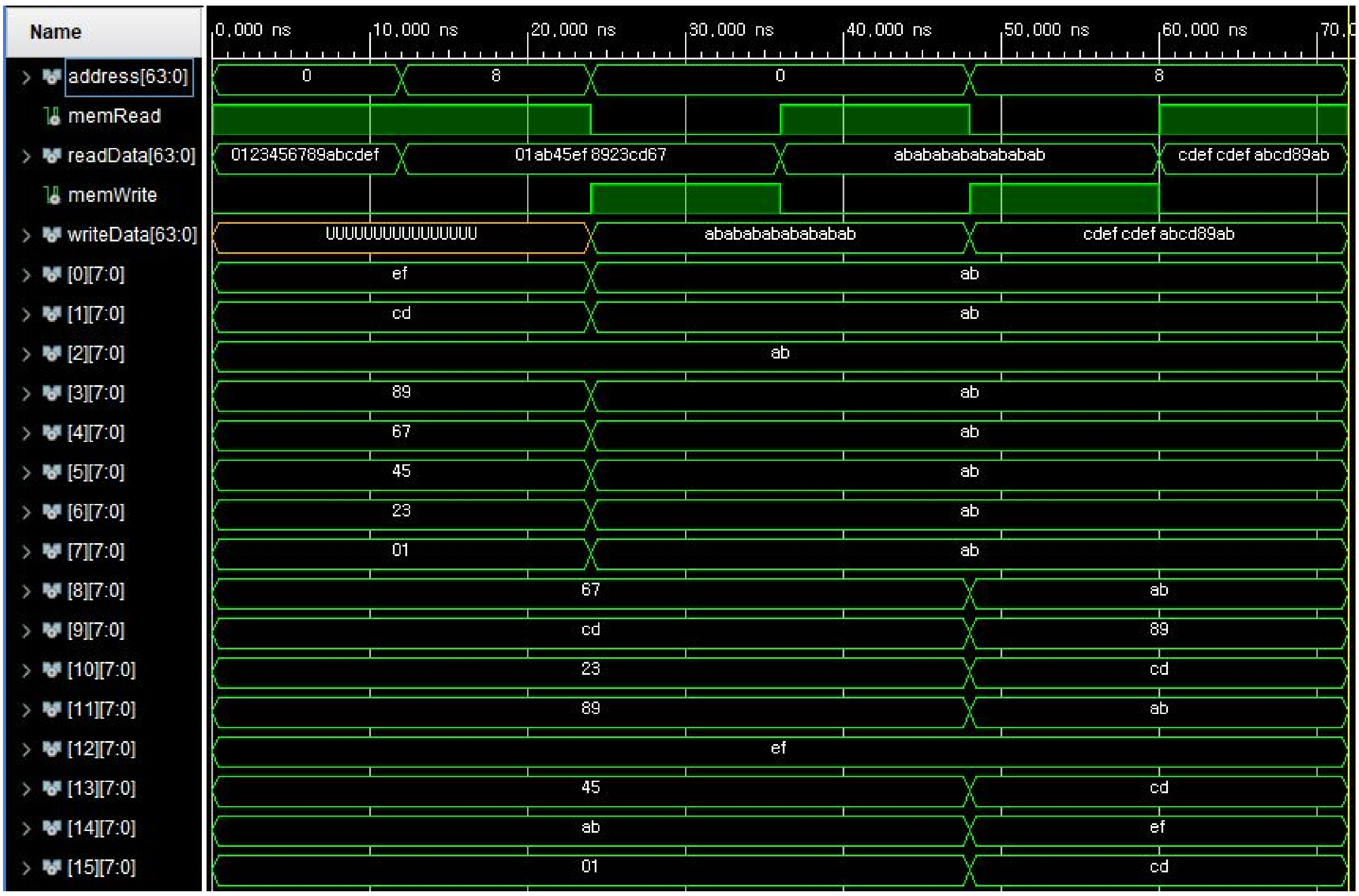
5. Data Memory (10 Points)

Implement a little-endian data memory by extending `DataMemory.vhd`. You may assume that only the byte addresses from `0x0` to `0x3FF` will be accessed. As the data memory is little-endian, the least-significant byte of the 8-byte doubleword data should be placed at the lowest address among the eight target byte addresses.

The data memory has four input signals: `address`, `memRead`, `memWrite`, and `writeData`. `address` specifies the target byte address we want to access. If `memRead` is 1, then we're performing a read operation. Similarly, if `memWrite` is 1, then we're performing a write operation. When we're performing a write operation, `writeData` specifies the 8-byte data to write from the target memory address.

You may assume that the unsigned decimal value of `address` signal is always a multiple of eight.

Example:



*the values of the address signal are shown in unsigned decimal numbers
*the values of all the other signal values are shown in hexadecimal numbers
*[0] ~ [15] correspond to the bytes stored in memory addresses 0 ~ 15, respectively.

6. A Single-Cycle Processor (50 Points)

Extend `SingleCycleProcessor.vhd` and implement a single-cycle processor which supports the following RISC-V instructions:

- The memory-reference instructions *load doubleword* (**`ld`**) and *store doubleword* (**`sd`**)
- The arithmetic-logical instructions **`add`**, **`sub`**, **`and`**, and **`or`**
- The conditional branch instruction *branch if equal* (**`beq`**)

FYI, the seven instructions utilize different instruction formats as follows:

- R-type for **`add`**, **`sub`**, **`and`**, **`or`**
- I-type for *load doubleword* (**`ld`**)
- S-type for *store doubleword* (**`sd`**)
- SB-type for *branch if equal* (**`beq`**)

and the four instruction formats are defined as:

Name	Field						Comments
(Field size)	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format

Using the opcode, funct3, and funct7 fields, the seven target instructions are defined as:

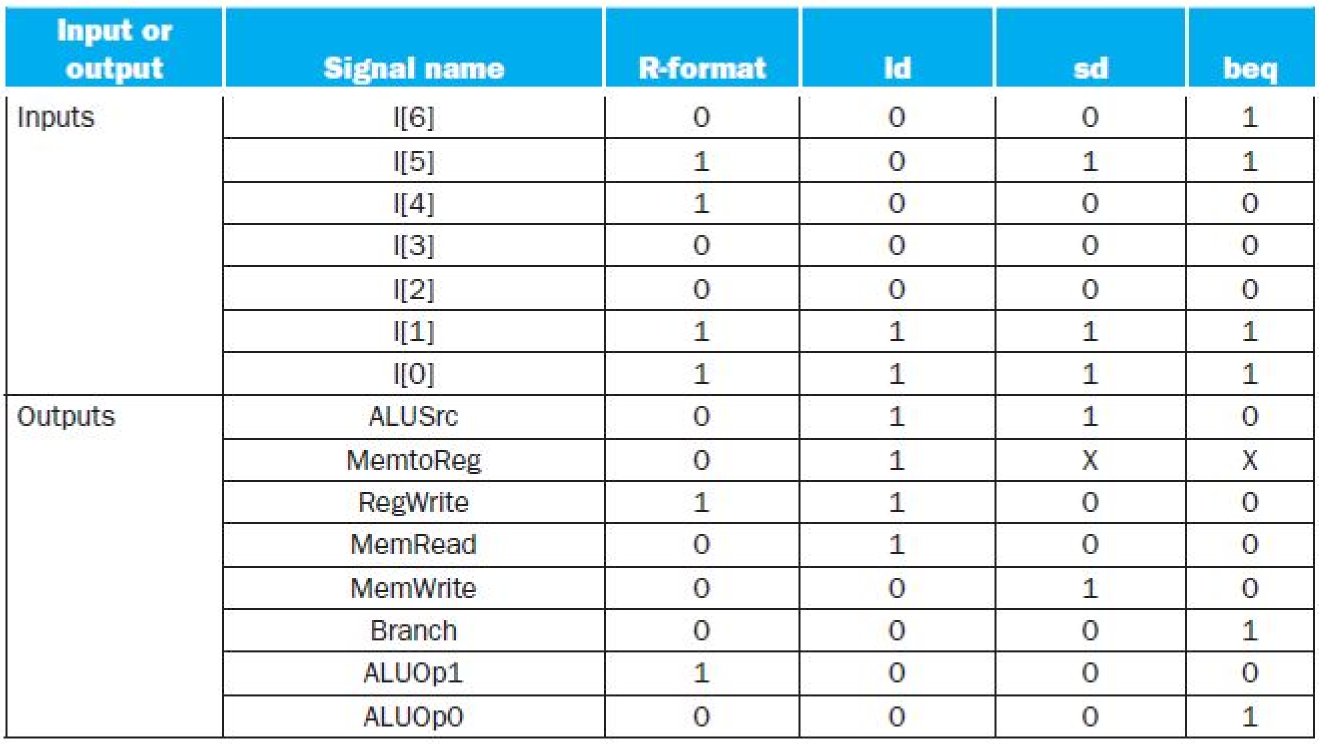
Instruction	opcode	funct3	funct7
add	0110011	000	0000000
sub	0110011	000	0100000
and	0110011	111	0000000
or	0110011	110	0000000
ld	0000011	011	X
sd	0100011	111	X
beq	1100111	000	X

The `rs1`, `rs2`, and `rd` fields specify the 1st source, 2nd source, and destination registers, respectively, using unsigned binary numbers. For example, if the first source register is `x10`, the `rs1` field is set to `01010`. On the other hand, the immediate values embedded within instructions utilize the two's complement representation for signed binary numbers. As an example, the `immed[11:0]` field for `beq x1, x2, -4` will contain `111111111100` which is the two's complement representation of `-4`.

When implementing the single-cycle processor, you should utilize the sequential hardware units you implemented for problems 1~5. You should also implement and utilize an immediate generator unit by extending `ImmediateGenerator.vhd`. The immediate generator unit takes in a RISC-V instruction as input, extracts the immediate value embedded in the instruction, sign-extends the immediate value, and produces the sign-extended 64-bit immediate value as its output.

In addition, you should implement two control units: one for controlling the ALU and the other for controlling the entire datapath, i.e., the Main Control Unit (MCU). You should implement the two control units by extending `ALUControlUnit.vhd` and `MainControlUnit.vhd`. The MCU takes in the seven least-significant bits of a RISC-V instruction (i.e., the `opcode` field) and produces the control signals for the instruction. The ALU control unit takes `ALUOp`, which is one of the output signals of the MCU, as input, and produces an appropriate op signal for the ALU.

When implementing the MCU, refer to the following table (Figure 4.22 in the textbook):



Input or output	Signal name	R-format	ld	sd	beq
Inputs	I[6]	0	0	0	1
	I[5]	1	0	1	1
	I[4]	1	0	0	0
	I[3]	0	0	0	0
	I[2]	0	0	0	0
	I[1]	1	1	1	1
	I[0]	1	1	1	1
Outputs	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

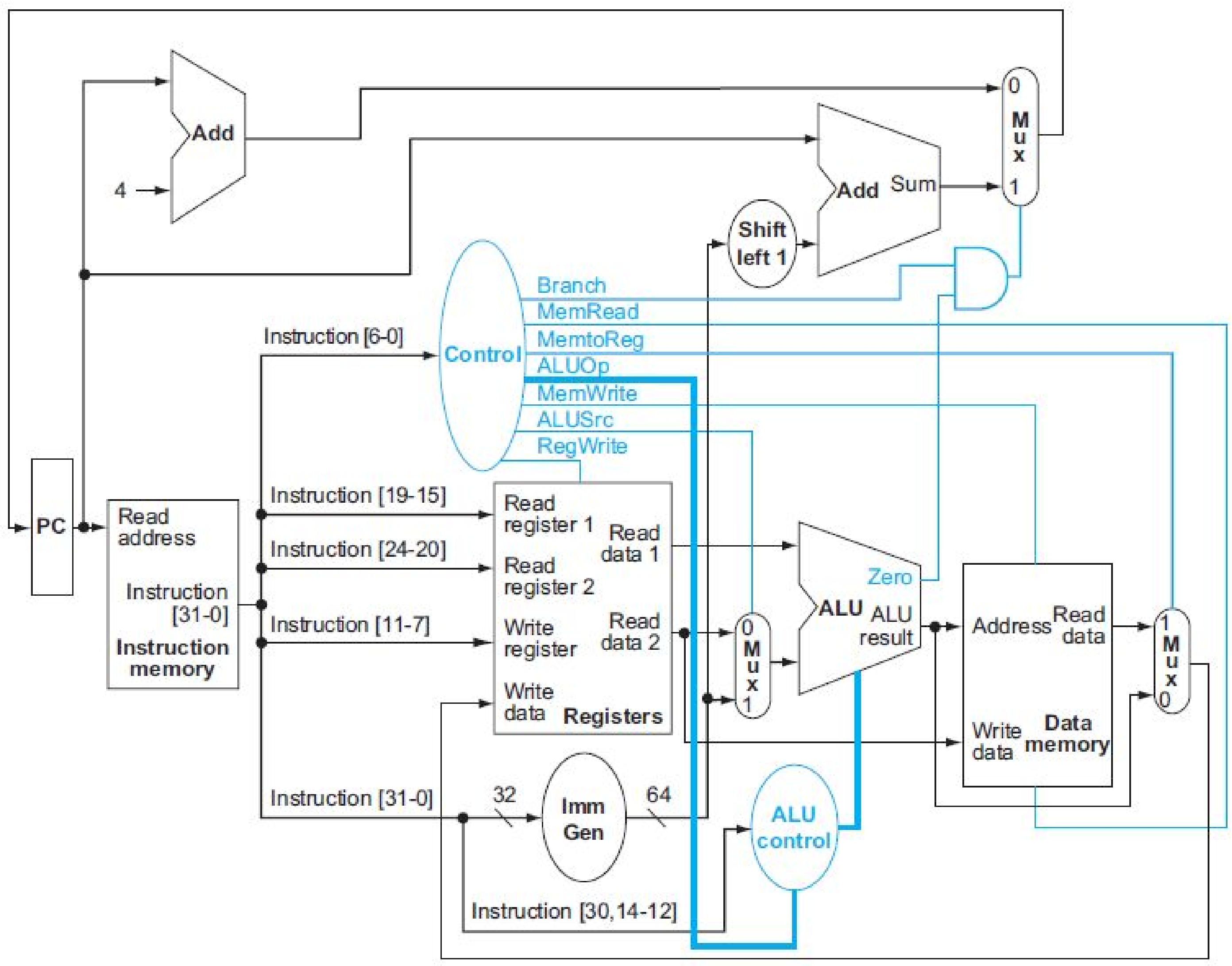
To implement the ALU control unit, refer to the following two tables (Figure 4.12 in the textbook):

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract

Instruction opcode	ALUOp	Operation	Funct7 field	Funct3 field	Desired ALU action	ALU control input
ld	00	load doubleword	XXXXXXX	XXX	add	0010
sd	00	store doubleword	XXXXXXX	XXX	add	0010
beq	01	branch if equal	XXXXXXX	XXX	subtract	0110
R-type	10	add	0000000	000	add	0010
R-type	10	sub	0100000	000	subtract	0110
R-type	10	and	0000000	111	AND	0000
R-type	10	or	0000000	110	OR	0001

Assume that the first instruction to execute is stored in the instruction memory starting from address 0x4 in little-endian. That is, the first 4-byte instruction is stored across instruction memory addresses `0x7` downto `0x4`. You may also assume that, in simulations, the initial value of the clock signal `clk` is always 0. When the first rising edge of the clock signal `clk` occurs, the value of the PC should be updated to 0x4, and your single-cycle processor implementation should execute the instructions stored in the instruction memory.

In summary, here's the complete hardware structure of the single-cycle processor you should implement (Figure 4.17 in the textbook):



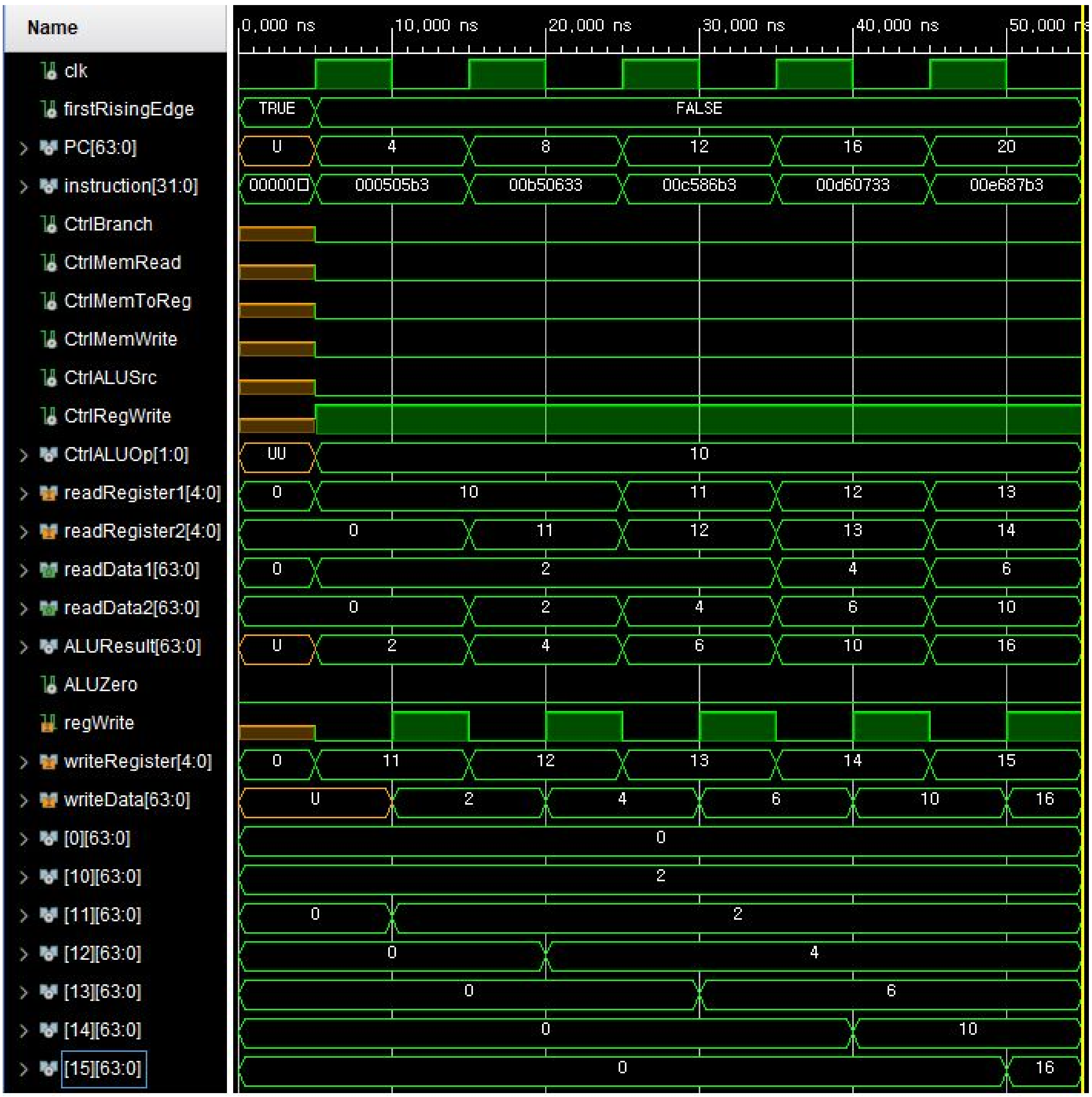
Example #1:

This example executes the following instructions:

```
                              <- MSB ->                         <- LSB ->
                              func7    rs2    rs1    func3  rd     opcode
  add x11, x10, x0        →  0000000 00000 01010 000    01011 0110011
  add x12, x10, x11       →  0000000 01011 01010 000    01100 0110011
  add x13, x11, x12       →  0000000 01100 01011 000    01101 0110011
  add x14, x12, x13       →  0000000 01101 01100 000    01110 0110011
  add x15, x13, x14       →  0000000 01110 01101 000    01111 0110011
```

when the initial value of register x10 is 2.

With respect to the instructions, the waveforms should look like:



1) PC, readRegister1, readRegister2, writeRegister: unsigned decimal numbers
2) instruction: hexadecimal numbers
3) CtrlALUOp: binary numbers
4) readData1, readData2, ALUResult, writeData, [0] (x0), [10]~[15] (x10~x15): signed decimal numbers

In order to simulate this example, you should modify InstructionMemory.vhd file to make the instruction memory have the proper values (i.e., instructions), and RegisterFile.vhd so that register x10 has 2 as its initial value.

RegisterFile.vhd:

```
architecture Behavioral of RegisterFile is
  type tRegister is array (0 to 31) of std_logic_vector(63 downto 0);
  -- problem #6 example #1
  signal registers: tRegister := (
    10 => std_logic_vector(to_signed(2, 64)),
    others => (others => '0')
  );
begin
  ...
end Behavioral;
```

`InstructionMemory.vhd`:

```
  architecture Behavioral of InstructionMemory is
    subtype tByte is std_logic_vector(7 downto 0);
    type t1KBMemory is array (0 to 1023) of tByte;
    -- problem #6 example #1
    signal memory: t1KBMemory := (
        x”00”, x”00”, x”00”, x”00”,
        x"B3", x"05", x"05", x"00", -- NOTE: little-endian
        x"33", x"06", x"B5", x"00",
        x"B3", x"86", x"C5", x"00",
        x"33", x"07", x"D6", x"00",
        x"B3", x"87", x"E6", x"00",
        others => (others => '0')
    );
  begin
    ...
  end Behavioral;
```

Example #2:

This example first sets the initial values of registers x10, x11, and x12 as 5, 0, and 1, respectively. Then, the example executes the following instructions:

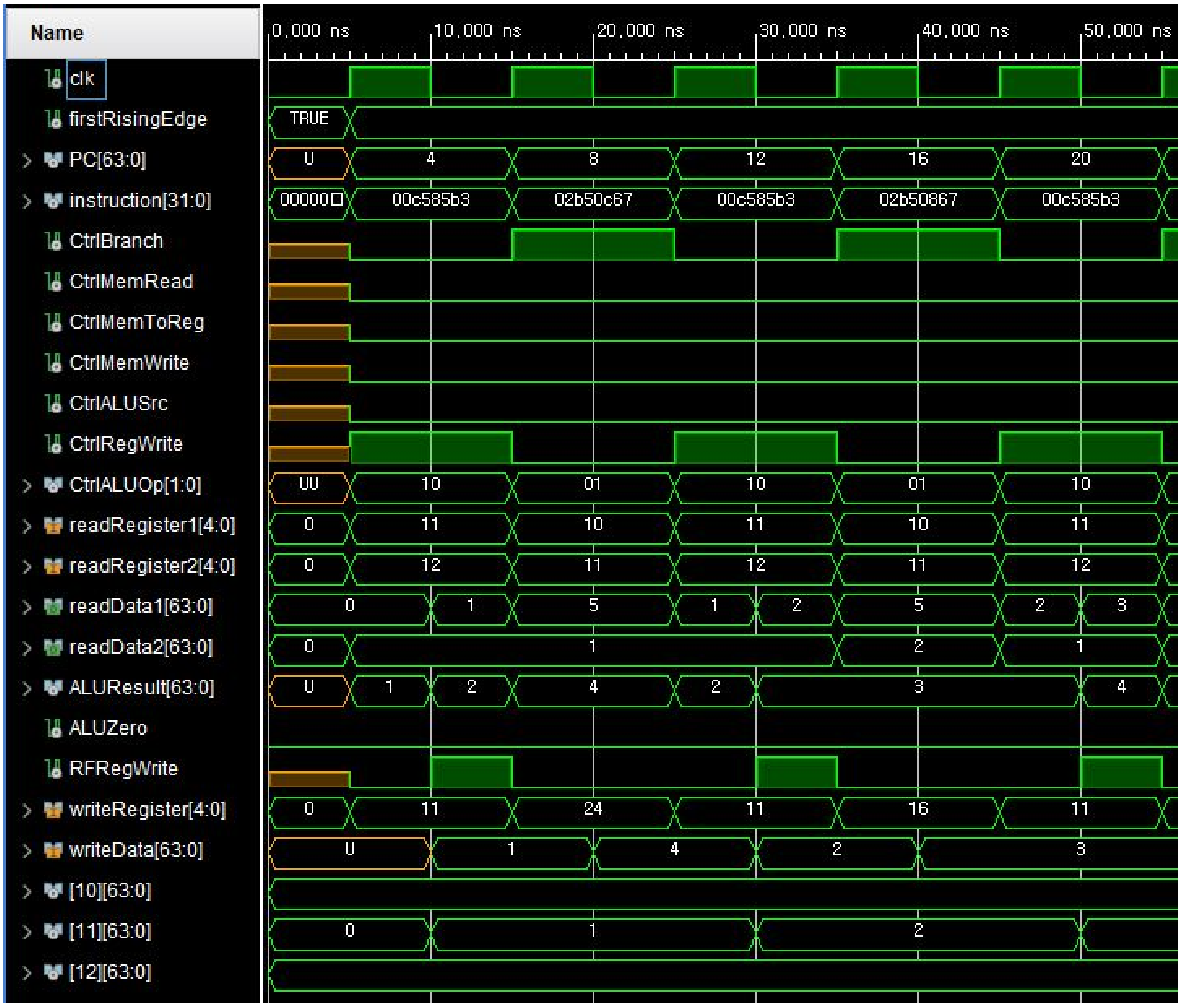
```
0x04: add x11, x11, x12     // x11 = x11 + x12 = 0 + 1 = 1
0x08: beq x10, x11, 0x38    // (x10 = 5) != (x11 = 1) --> not taken
0x0C: add x11, x11, x12     // x11 = x11 + x12 = 1 + 1 = 2
0x10: beq x10, x11, 0x30    // (x10 = 5) != (x11 = 2) --> not taken
0x14: add x11, x11, x12     // x11 = x11 + x12 = 2 + 1 = 3
0x18: beq x10, x11, 0x28    // (x10 = 5) != (x11 = 3) --> not taken
0x1C: add x11, x11, x12     // x11 = x11 + x12 = 3 + 1 = 4
0x20: beq x10, x11, 0x20    // (x10 = 5) != (x11 = 4) --> not taken
0x24: add x11, x11, x12     // x11 = x11 + x12 = 4 + 1 = 5
0x28: beq x10, x11, 0x18    // (x10 = 5) == (x11 = 5) --> taken, goto 0x40
0x2C: sub x11, x11, x12     // not executed
0x30: sub x11, x11, x12     // not executed
0x34: sub x11, x11, x12     // not executed
0x38: sub x11, x11, x12     // not executed
0x3C: sub x11, x11, x12     // not executed
0x40: add x11, x11, x11     // x11 = x11 + x11 = 5 + 5 = 10
```

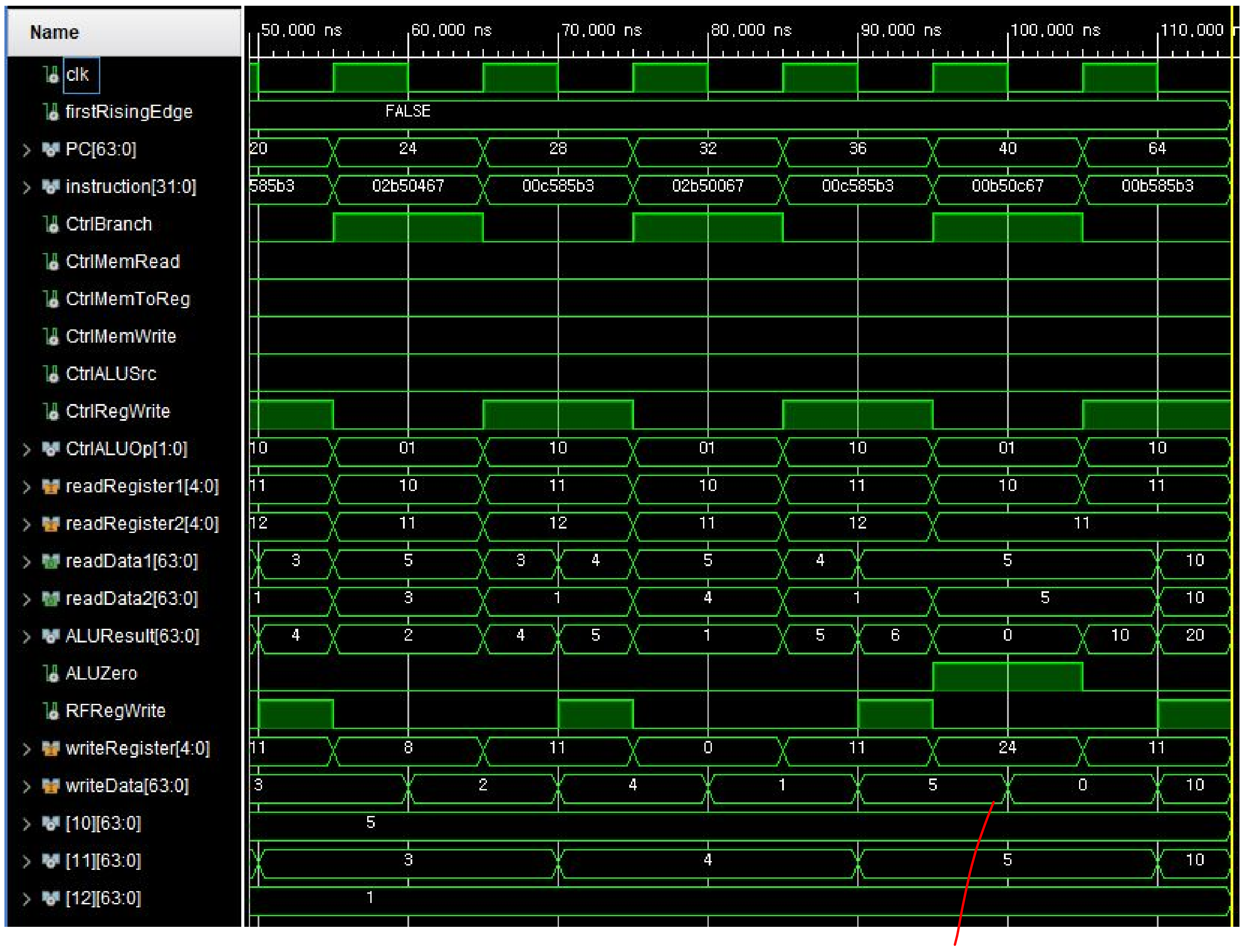
According to the instruction formats, the instructions in this example are encoded as:

```
                      func7   rs2   rs1   funct3 rd    opcode
```

```
add x11, x11, x12  --> 0000000 01100 01011 000    01011 0110011
sub x11, x11, x12  --> 0100000 01100 01011 000    01011 0110011
add x11, x11, x11  --> 0000000 01011 01011 000    01011 0110011

                       immed[12,10:5] rs2   rs1   funct3 immed[4:1,11] opcode
beq x10, x11, 0x38 --> 0000001        01011 01010 000    11000         1100111
beq x10, x11, 0x30 --> 0000001        01011 01010 000    10000         1100111
beq x10, x11, 0x28 --> 0000001        01011 01010 000    01000         1100111
beq x10, x11, 0x20 --> 0000001        01011 01010 000    00000         1100111
beq x10, x11, 0x18 --> 0000000        01011 01010 000    11000         1100111
```

Then, the waveform would look like:





Note that the values of the ‘CtrlALUOp’ (i.e., ALUOp control signal in the hardware diagram) is shown in binary numbers, the values of the ‘PC’, ‘readRegister1’, ‘readRegister2’, and ‘writeRegister’ are shown in unsigned decimal numbers, the values of the ‘instruction’ are shown in hexadecimal numbers, and the values of ‘readData1’, ‘readData2’, and ‘writeData’, [10]/[11]/[12] (i.e., registers x10/x11/x12) are shown in signed decimal numbers.

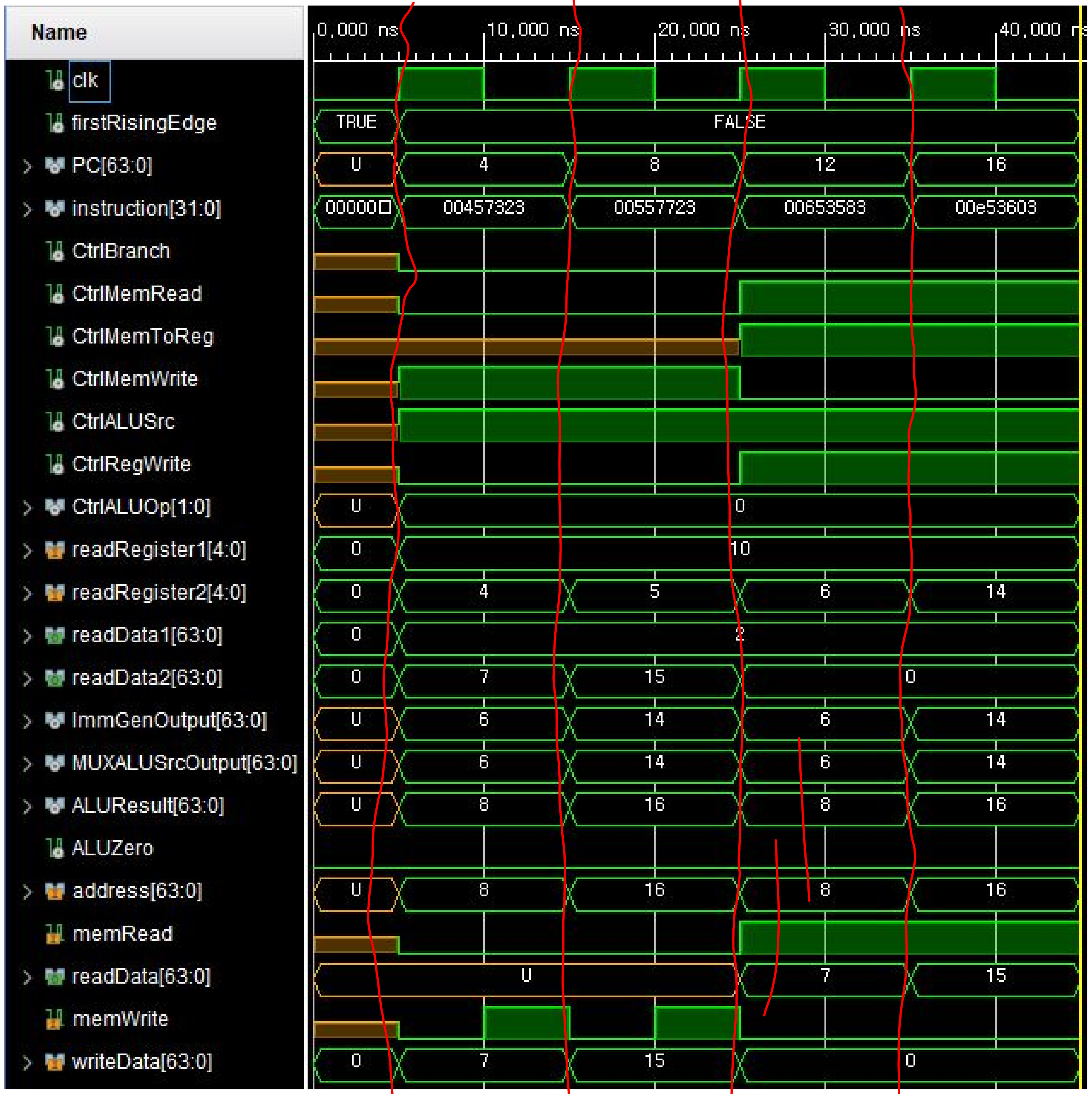
Example #3:

This example instructs the single-cycle processor to execute the following instructions:

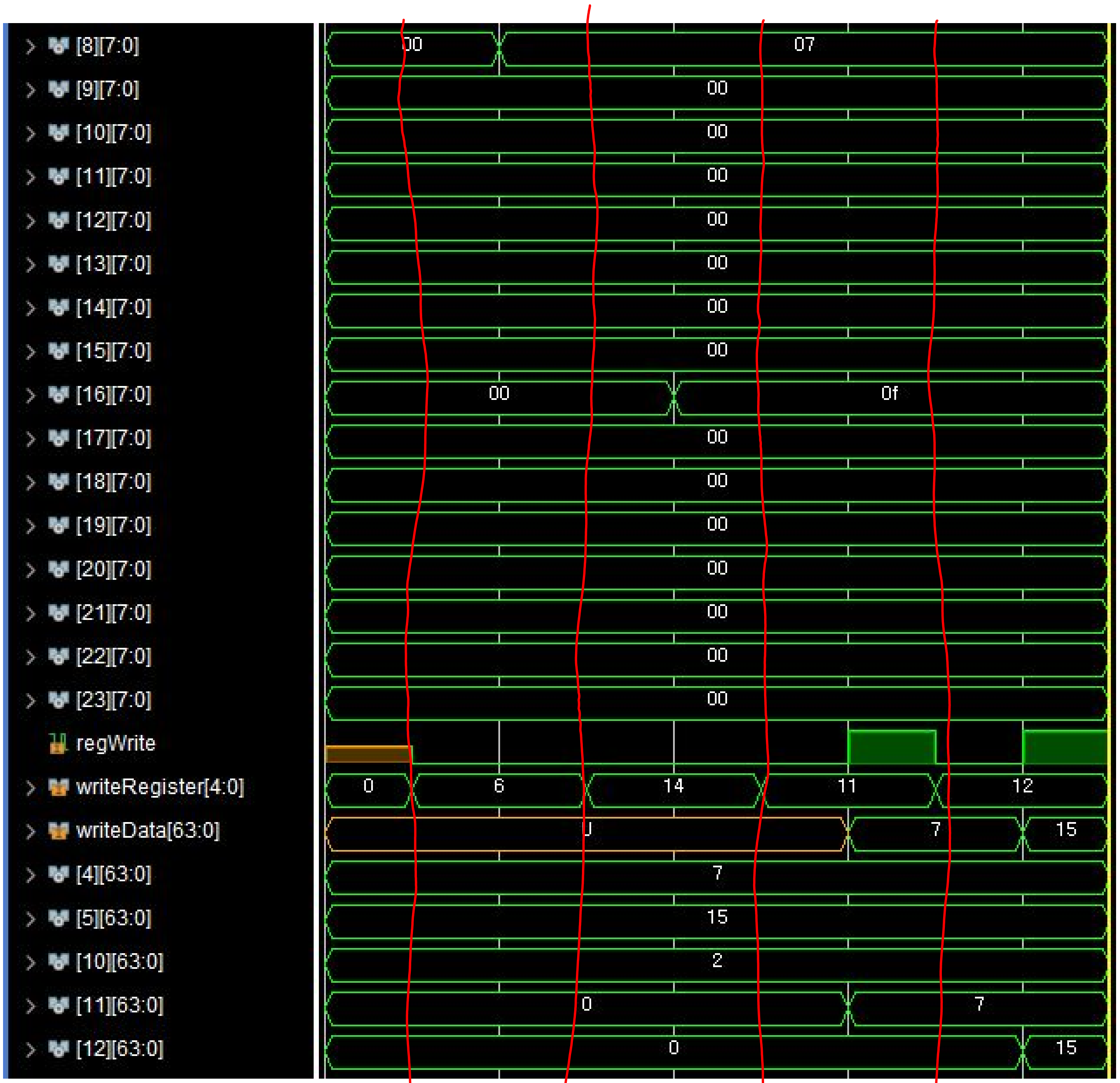
```
                              immed[11:5]  rs2   rs1   funct3 immed[4:0] opcode
0x04: sd x4, 0x6(x10)   -->  0000000      00100 01010 111    00110      0100011
0x08: sd x5, 0xE(x10)   -->  0000000      00101 01010 111    01110      0100011

                              immed[11:0]  rs1   funct3 rd    opcode
0x0C: ld x11, 0x6(x10) -->  000000000110 01010 011    01011 0000011
0x10: ld x12, 0xE(x10) -->  000000001110 01010 011    01100 0000011
```

when the registers are initialized as: `x4 = 7`, `x5 = 15`, and `x10 = 2`.

A valid waveform would look like:



* the values of PC, readRegister1, readRegister2, and address are shown in unsigned decimal numbers.
** the values of readData1, readData2, ImmGenOutput, MUXALUSrcOutput, ALUResult, readData, and writeData are shown in signed decimal numbers.



* [8]~[23] correspond to the data memory at addresses 8~23, respectively.
** the values of writeRegister are shown in unsigned decimal numbers.
*** the values of writeData are shown in unsigned decimal numbers.
**** [4], [5], [10], [11], and [12] correspond to registers x4, x5, x10, x11, and x12, respectively.

* Acknowledgement:
Thanks to Hyemi No for providing an example.

Example #4:

In this example, the single-cycle processor executes the following instructions:

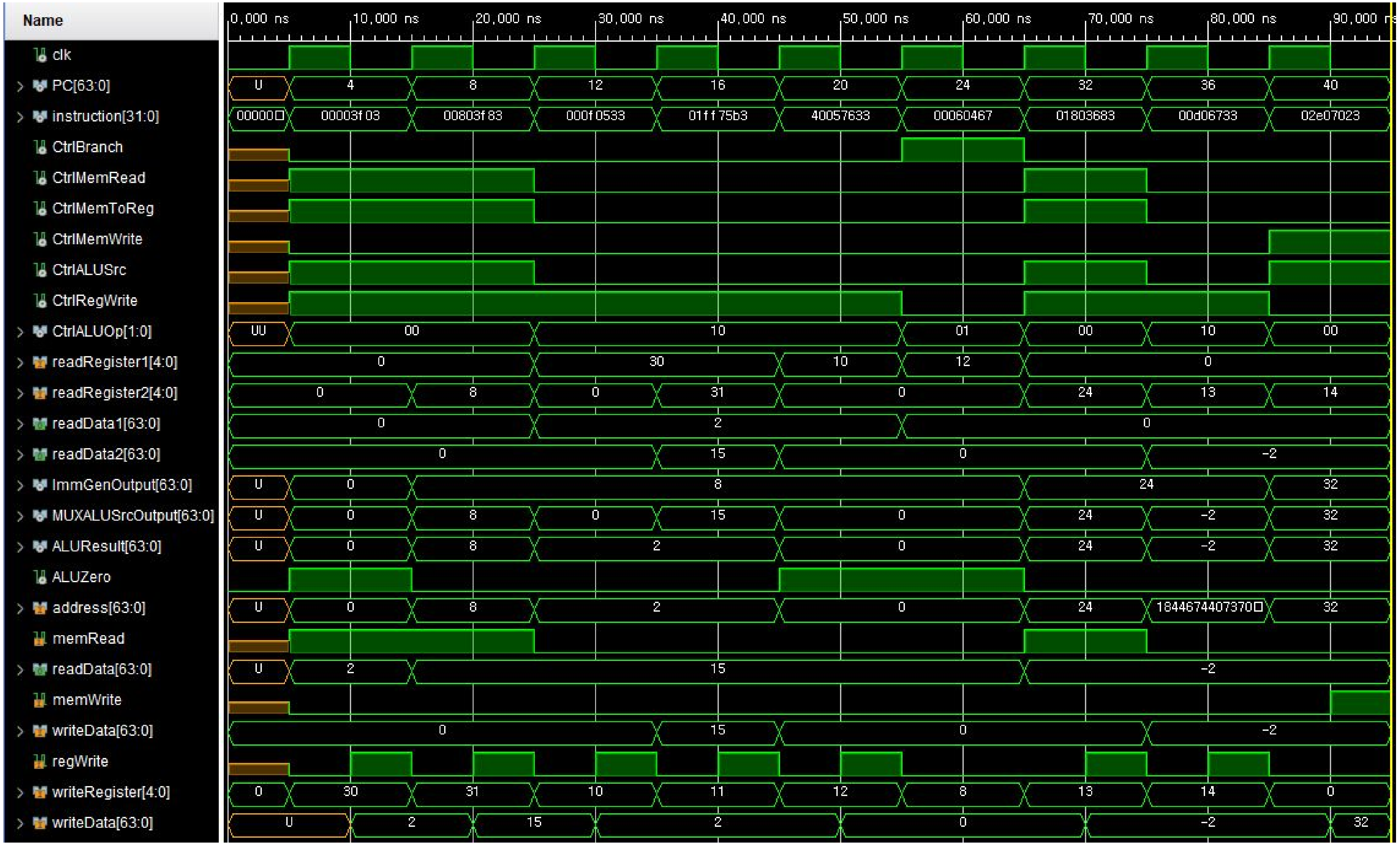
```
add, sub, and, or, ld, sd, beq

0x04: ld  x30, 0x0(x0)    // x30 = DataMemory[ 7:0] = 0x2 =  2
0x08: ld  x31, 0x8(x0)    // x31 = DataMemory[15:8] = 0xF = 15
0x0C: add x10, x30, x0    // x10 = x30 + x0 = 2 + 0 = 2
0x10: and x11, x30, x31   // x11 = x30 AND x31 = 0x2 AND 0xF = 2
0x14: sub x12, x10, x11   // x12 = x11 - x10 = 0x2 - 0x2 = 0
0x18: beq x12, x0, 0x8    // if x12(=0) == x0(=0), then nextPC = currPC + 0x8
```

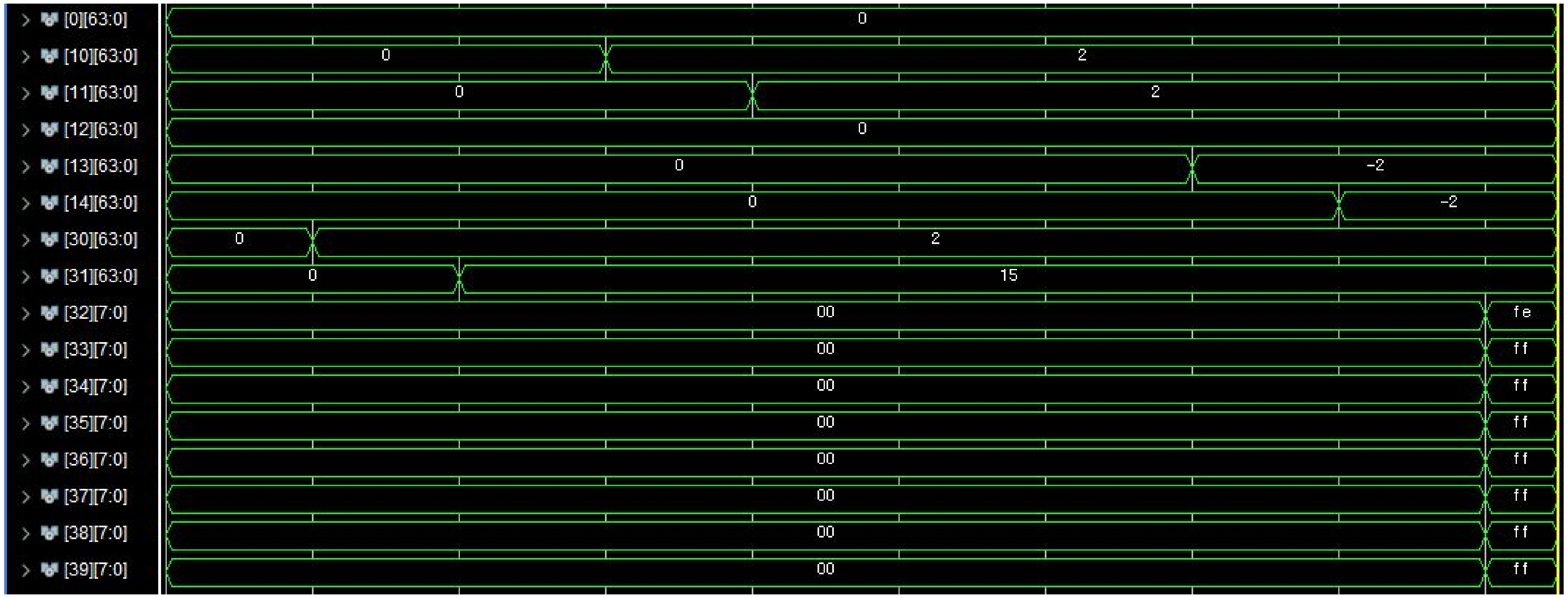
```
0x1C: ld  x13, 0x10(x0)  // NOT EXECUTED; x13 = DataMemory[15:8] = 0xF
0x20: ld  x13, 0x18(x0)  // x13 = DataMemory[23:15] = -2
0x24: or  x14, x0, x13   // x14 = x0 OR x13(=-2) = -2
0x28: sd  x14, 0x20(x0)  // DataMemory[39:32] = x14 = -2
```

assuming that the data memory has been initialized as follows:

```
DataMemory[31:24] = 0xFFFFFFFFFFFFFFFE  // = -2
DataMemory[23:16] = 0x0000000000000002
DataMemory[15: 8] = 0x000000000000000F
DataMemory[ 7: 0] = 0x0000000000000002
```

A valid waveform should look like:


Name
0,000 ns
10,000 ns
20,000 ns
30,000 ns
40,000 ns
50,000 ns
60,000 ns
70,000 ns
80,000 ns
90,000 ns
clk
PC[63:0] U 4 8 12 16 20 24 32 36 40
instruction[31:0] 00000 00003f03 00803f83 000f0533 01ff75b3 40057633 00060467 01803683 00d06733 02e07023
CtrlBranch
CtrlMemRead
CtrlMemToReg
CtrlMemWrite
CtrlALUSrc
CtrlRegWrite
CtrlALUOp[1:0] UU 00 10 01 00 10 00
readRegister1[4:0] 0 30 10 12 0
readRegister2[4:0] 0 8 0 31 0 24 13 14
readData1[63:0] 0 2 0
readData2[63:0] 0 15 0 -2
ImmGenOutput[63:0] U 0 8 24 32
MUXALUSrcOutput[63:0] U 0 8 0 15 0 24 -2 32
ALUResult[63:0] U 0 8 2 0 24 -2 32
ALUZero
address[63:0] U 0 8 2 0 24 1844674407370 32
memRead
readData[63:0] U 2 15 -2
memWrite
writeData[63:0] 0 15 0 -2
regWrite
writeRegister[4:0] 0 30 31 10 11 12 8 13 14 0
writeData[63:0] U 2 15 2 0 -2 32



* [0], [10]~[14], [30], [31] correspond to registers x0, x10~x14, x30, and x31

** [32]~[39] correspond to the data memory addresses 32~39.