VHDL Programming Assignment #4

Architecture of Computers (CSI3102-01), Spring 2020

Welcome to the fourth programming assignment of the Architecture of Computers (CSI3102-01) course at Yonsei University! In the previous assignment, you implemented a single-cycle processor which supports seven representative RISC-V instructions. The seven instructions cover various issues which we must carefully deal with when implementing a processor, and we believe you now know how we should deal with those issues when implementing the single-cycle processor.

In this programming assignment, you will extend the single-cycle processor you implemented for the previous assignment to support 5-stage pipelining. As discussed during prior lectures, pipelining is implemented in virtually every modern processor as it greatly improves throughput.

In addition, as an accurate single-cycle processor implementation is necessary to implement the 5-stage pipelining, you will be given an extra opportunity to complete your single-cycle processor implementation. Please use the extra opportunity to strengthen your processor implementation knowledge and implementation techniques.

The deadline for this assignment is **11:59pm on June 10th, 2020 (Wed)**. No extension will be given.

1. A Single-Cycle Processor (30 Points)

Changes vs. VHDL Programming Assignment #3

- The opcode for the `beq` instruction has changed to **`1100011`** from `1100111`.
- ***DO NOT*** change the labels of the pre-defined VHDL components (i.e., `RF` for the register file, `DataMem` for `DataMemory`) when extending `SingleCycleProcessor.vhd`.

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 size)	Field 7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	Comments
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	1100011	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

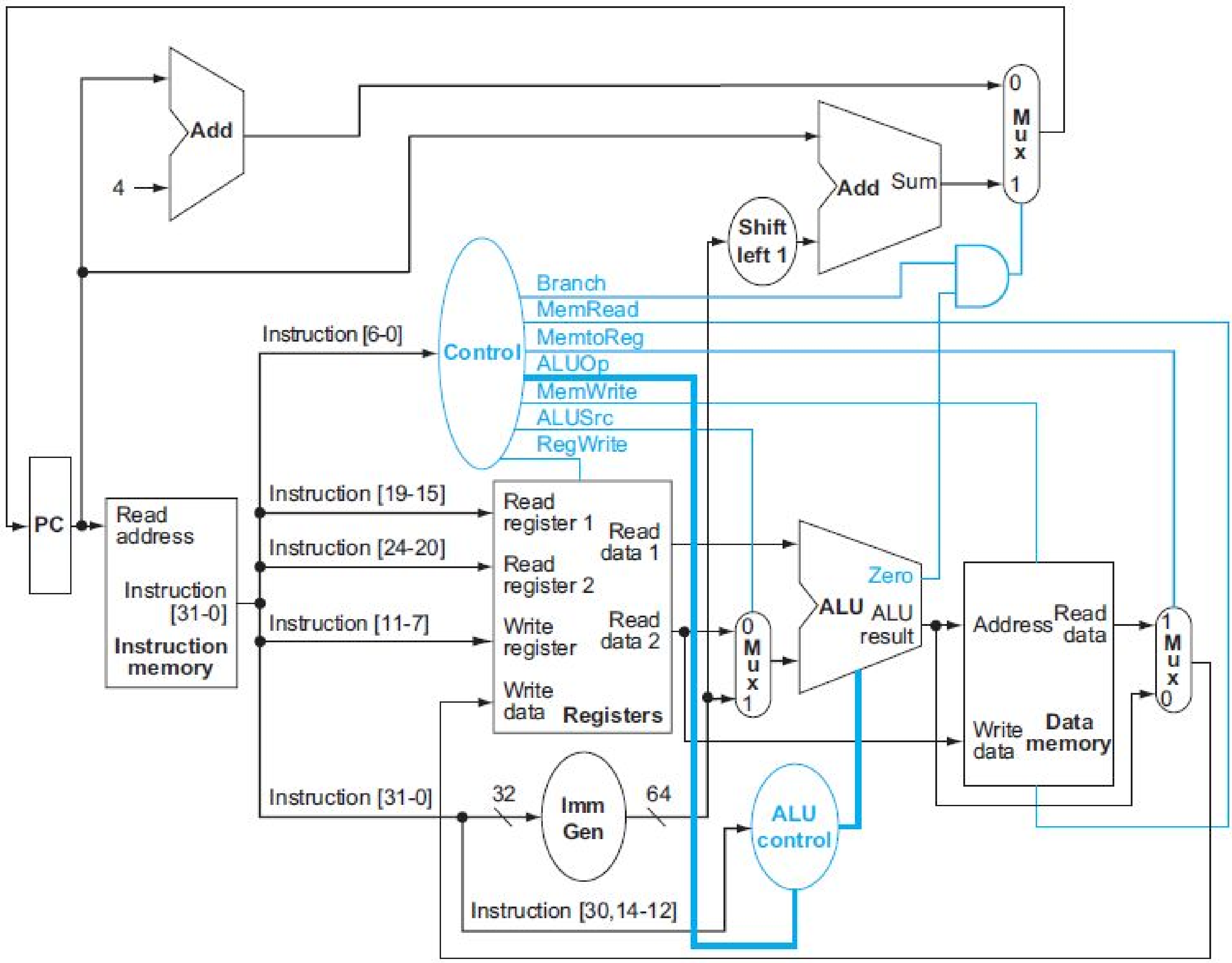
To implement the ALU control unit, refer to the following two tables (Figure 4.12 in the textbook):

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):



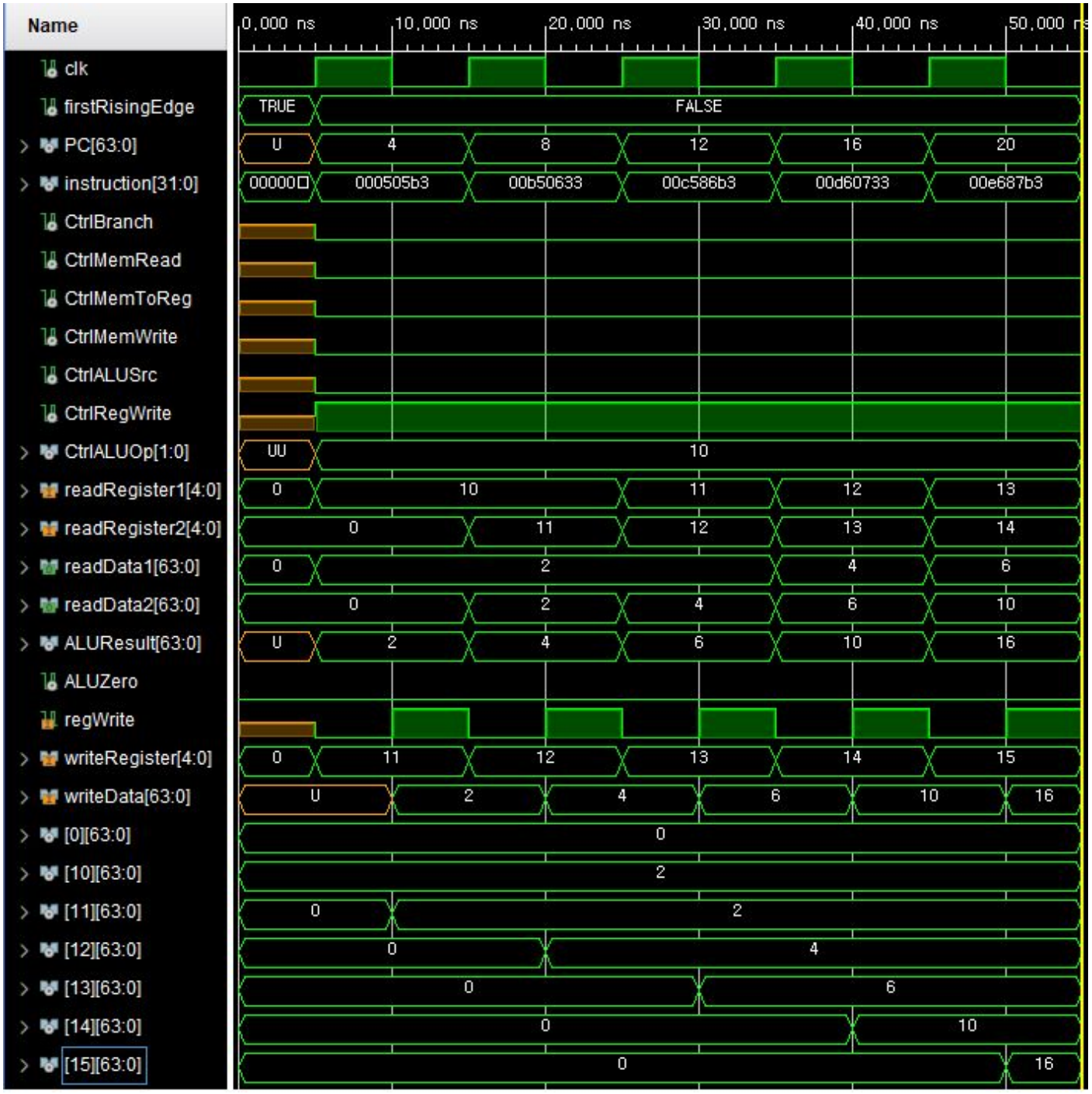
Example:

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;
```

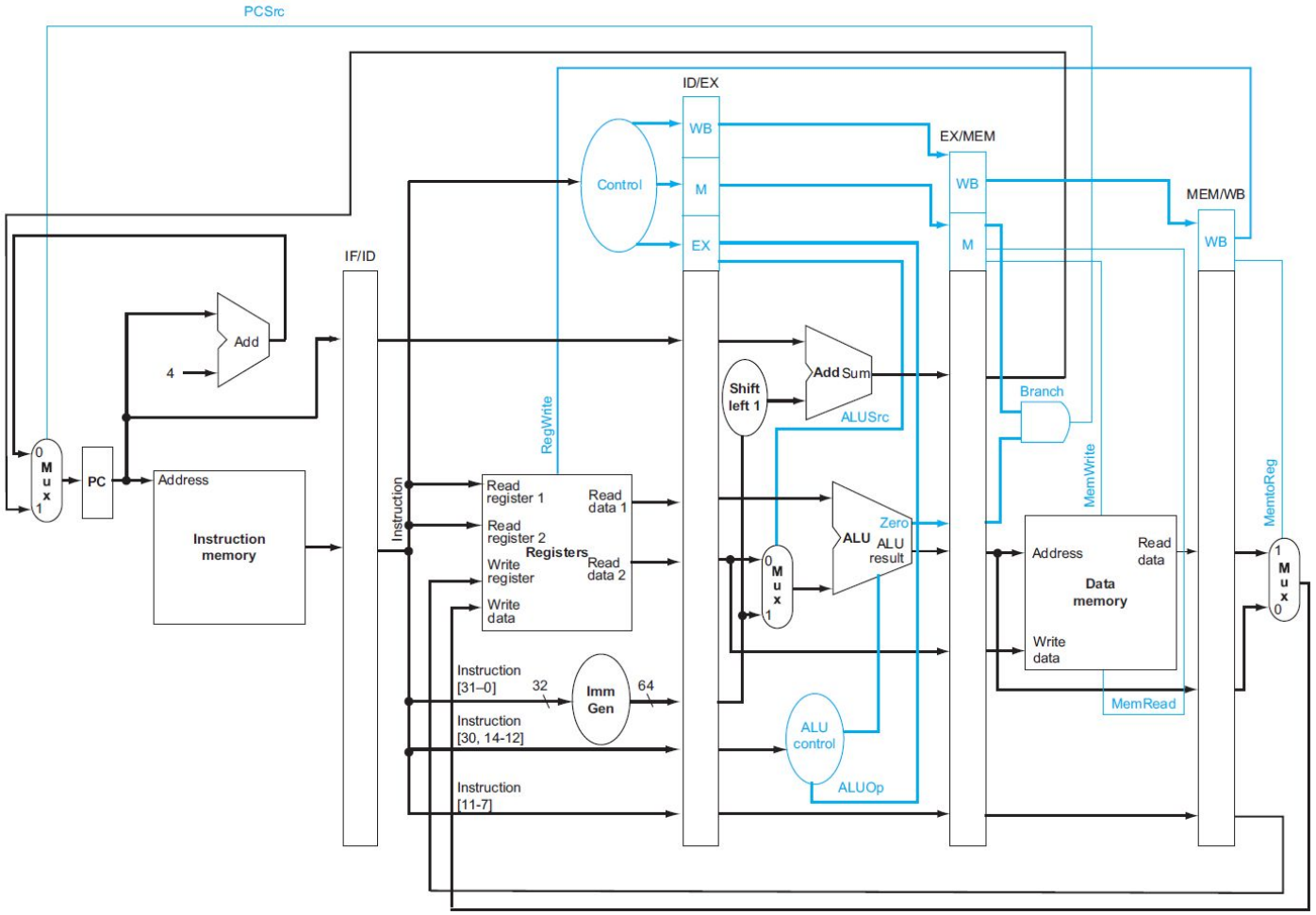
InstructionMemory.vhd:

```
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;
```

2. A 5-Stage Pipelined Processor (70 Points)

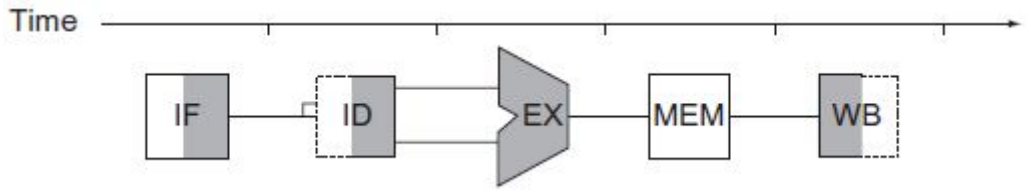
Implement a 5-stage pipelined processor by extending `PipelinedProcessor.vhd`. The instructions the pipelined processor should support and the basic building blocks for implementing the pipelined processor are the same to those of the single-cycle processor you implemented in the previous assignment, so I recommend you to start with your `SingleCycleProcessor.vhd`.

Unlike the single-cycle processor, the execution of an instruction should take five clock cycles on the pipelined processor; however, the pipelined processor should be able to complete one instruction per clock cycle when the pipeline is fully filled with five independent instructions.



A 5-stage pipelined datapath to support the seven target RISC-V instructions (Figure 4.49)

Use VHDL's rising_edge and falling_edge functions to make the timing of each of the pipeline stages to be:

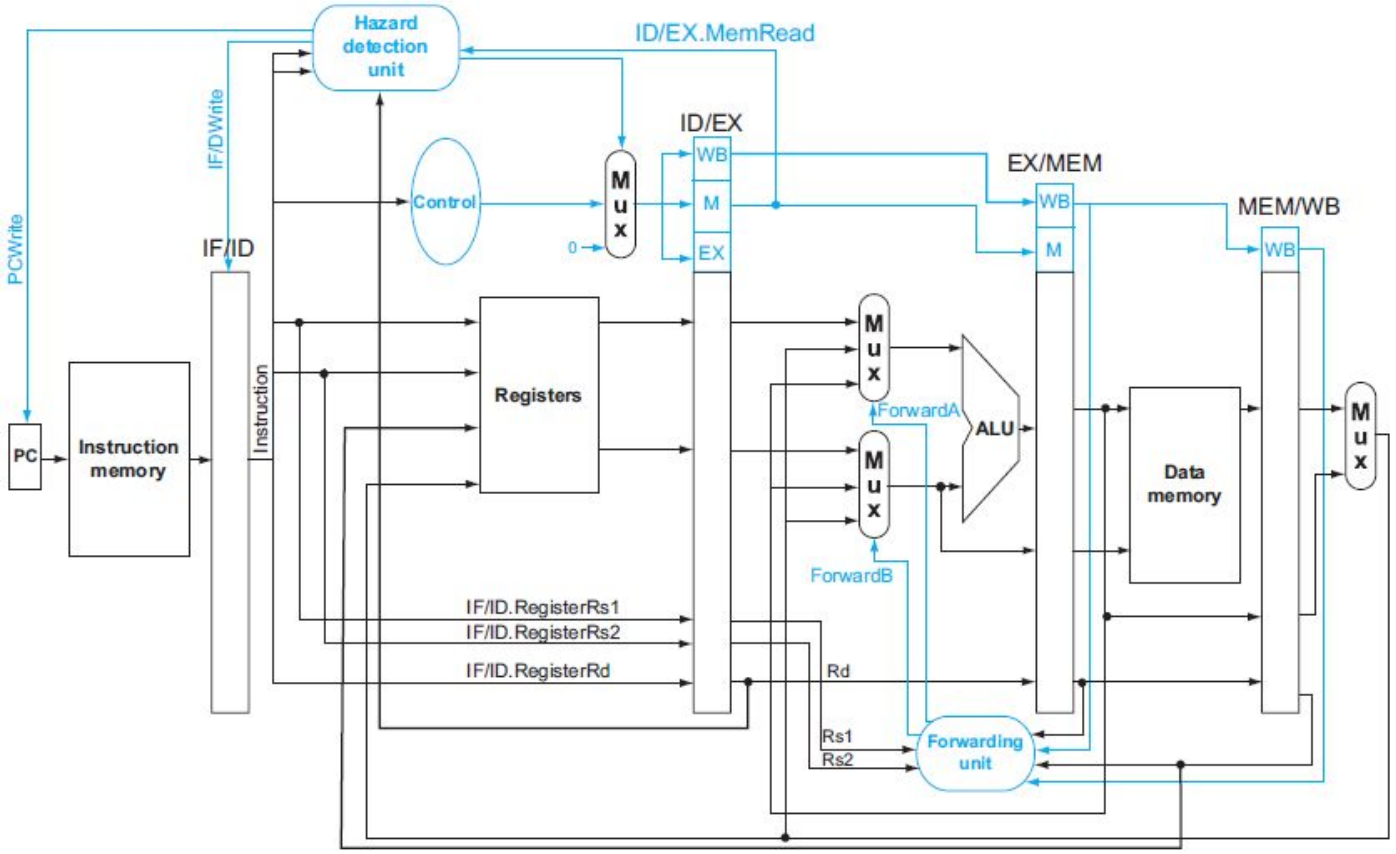


where any writes to memory occur in the first half of a clock cycle, and any reads from memory occur in the second half of the clock cycle. For example, the Write-Back (WB) stage writes a new value to its destination register upon a rising edge, and the Instruction Fetch (IF) stage reads data from the instruction memory and the Instruction Decode (ID) stage reads the value(s) of its source operand(s) upon a falling edge.

>>>>>> UPDATE #1

Unfortunately, the default 5-stage pipelining may not guarantee the functionally-correct execution of instructions due to data hazards and the lack of stalls. To ensure that your pipelined processor achieves functional correctness, you should resolve data hazards by implementing data forwarding and inserting no-ops (nops) when necessary.

To do so, you should extend the default 5-stage pipelining to have a forwarding unit and a hazard detection unit. When the 5-stage pipelining is augmented with the forwarding unit and the hazard detection unit, the overall pipeline should look like:



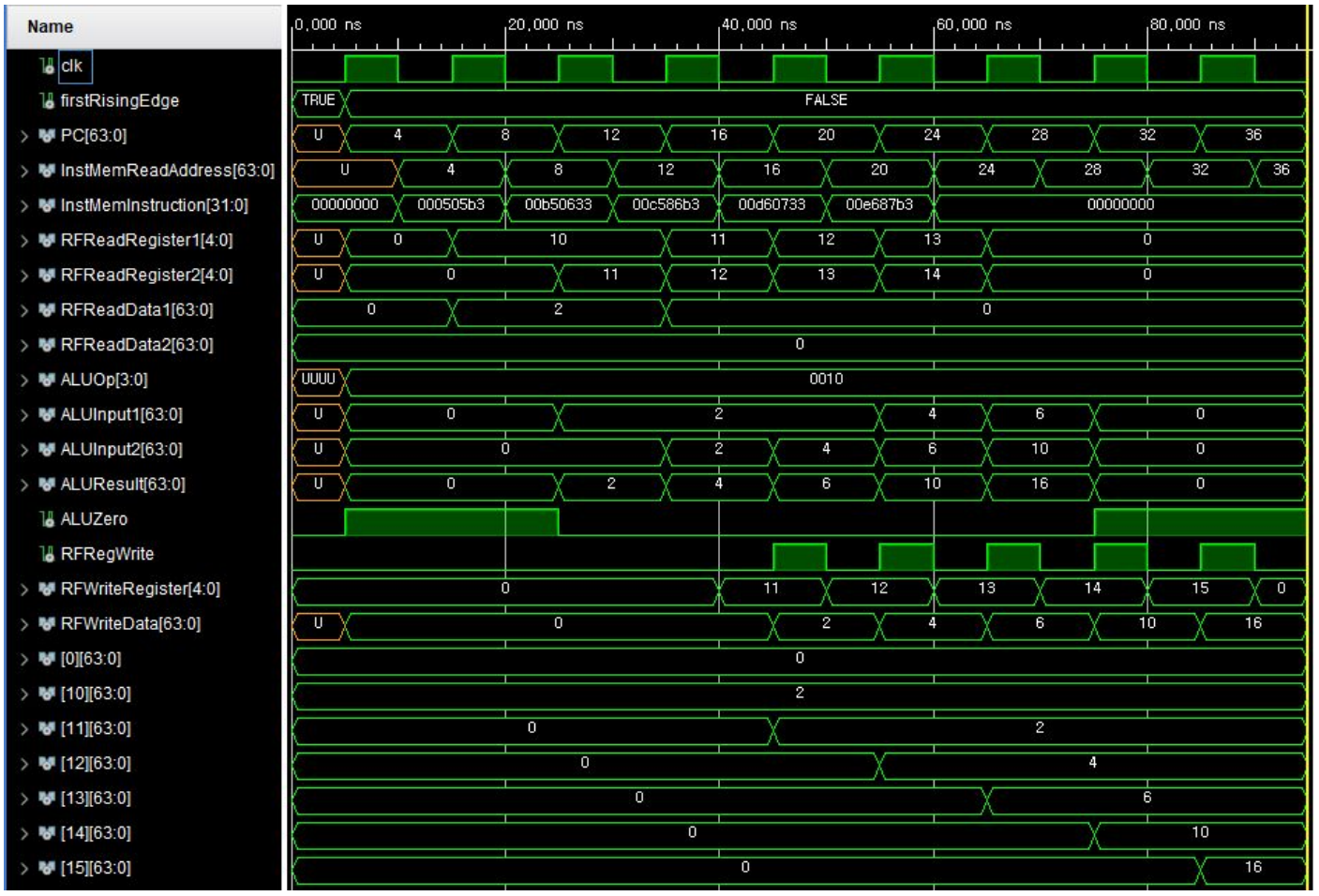
A 5-stage pipelined processor with the forwarding and the hazard detection units (Figure 4.58)

Refer to the textbook's Chapter 4.7 for more details on the forwarding and the hazard detection units.

<<<<<< UPDATE #1

All the specifications for the instructions are the same to those of the single-cycle processor. In addition, when extending `PipelinedProcessor.vhd`, do not change the labels of the pre-defined VHDL components (i.e., RF for the register file, `DataMem` for `DataMemory`).

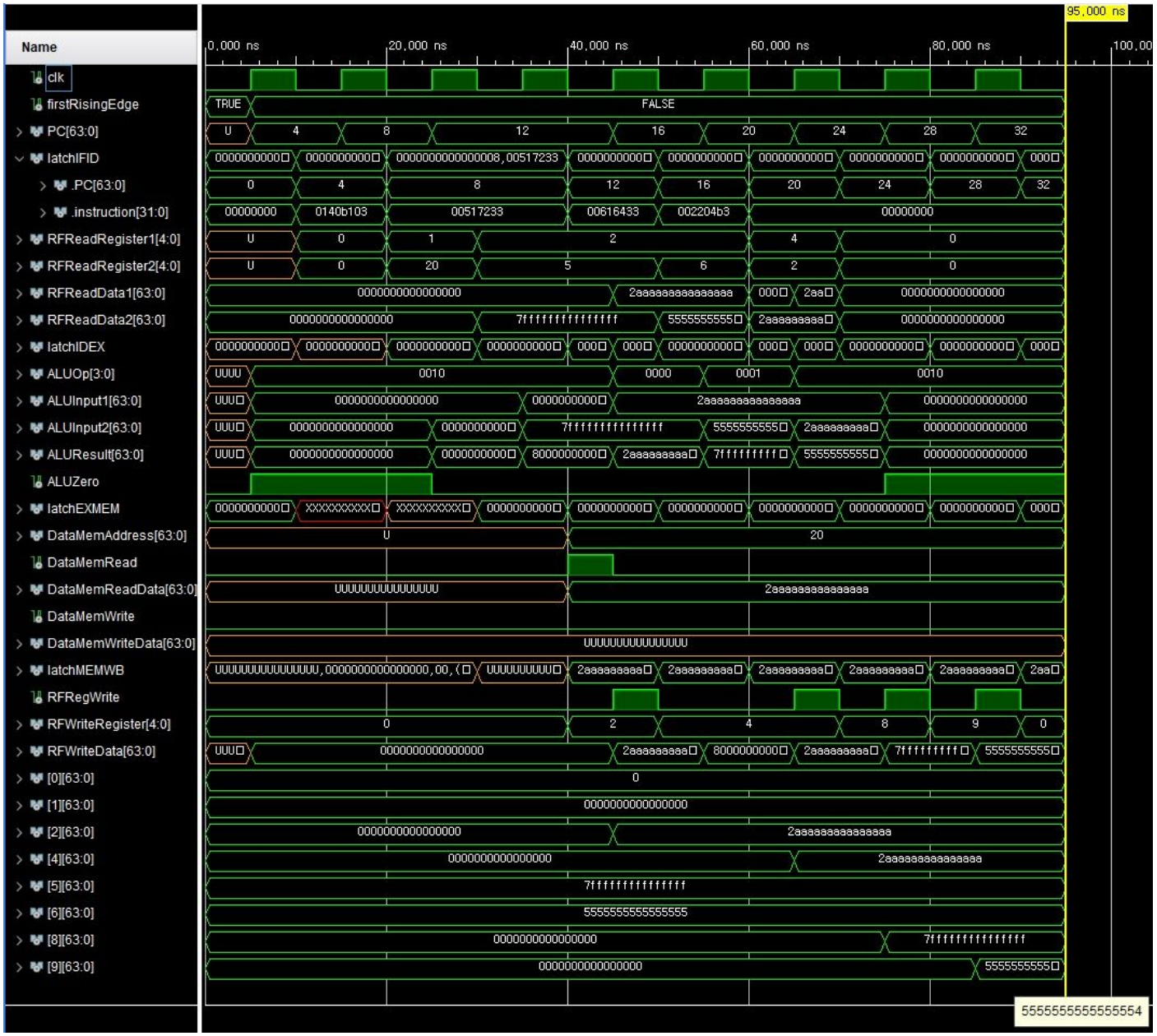
Example #1:



Refer to “csi3102-assn4-prob2-ex1.png” file uploaded to YSCEC for a more clear image. The example uses the same instructions to those of problem #1. Take the following notices into account when analyzing the example waveform:

- PC, InstMemReadAddress, RFReadRegister1, RFReadRegister2, RFWriteRegister signals are shown in unsigned decimal numbers.
- InstMemInstruction signal is shown in hexadecimal numbers.
- ALUOp signal is shown in binary numbers.
- RFReadData1, RFReadData2, ALUInput1, ALUInput2, ALUResult, RFWriteData, and registers x0, x10, ..., x15 ([0], [10]~[15]) are shown in signed decimal numbers.

Example #2:



Refer to “csi3102-assn4-prob2-ex2.jpg” file uploaded to YSCEC for a more clear image. This example demonstrates a pipeline stall example (Figure 4.57 in the textbook).

In this example, we execute the following instructions:

```
ld x2, 20(x1)     // 0x0140B103
and x4, x2, x5    // 0x00517233
or x8, x2, x6     // 0x00616433
add x9, x4, x2    // 0x002204B3
```

with the following initial values for registers and data memory:

```
x1 = 0x0000000000000000
x5 = 0x7FFFFFFFFFFFFFFF
x6 = 0x5555555555555555
DataMemory[27:20] = 0x2AAAAAAAAAAAAAAA
```

According to the instructions and the initial values, the programmer-visible state should be changed to:

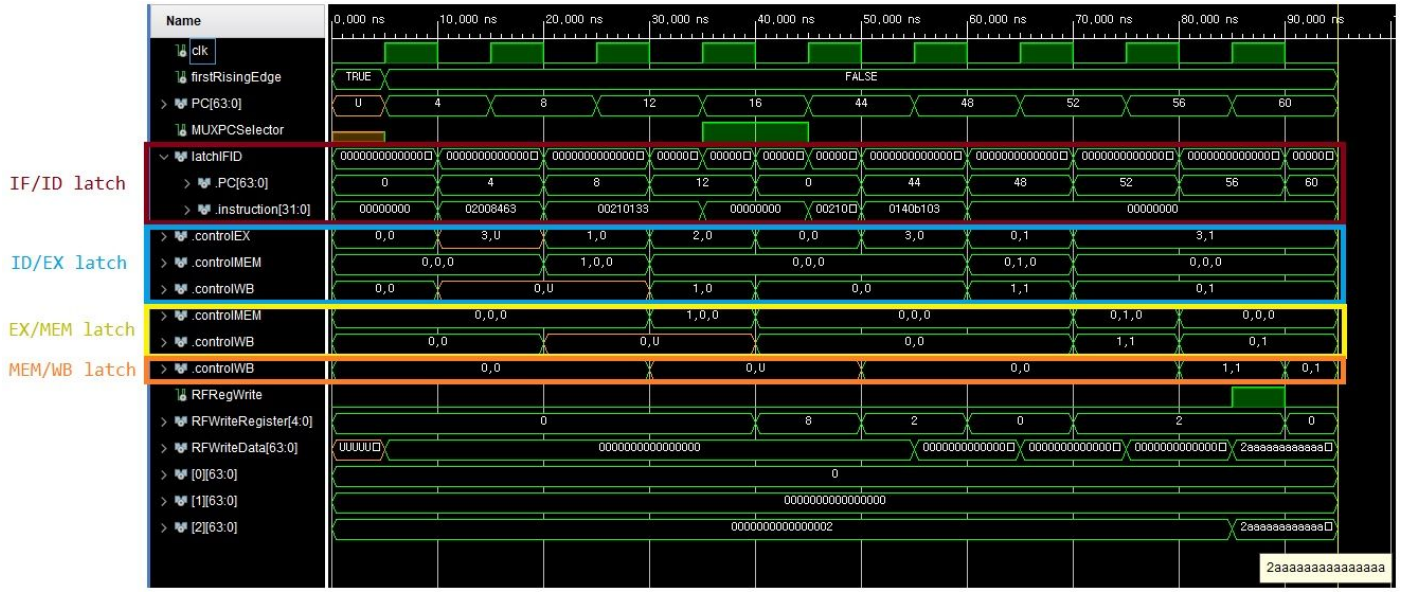
```
x4 = 0x2AAAAAAAAAAAAAA
x8 = 0x7FFFFFFFFFFFFFF
x9 = 0x5555555555555554
```

Note that a stall cycle has been inserted between instructions `ld x2, 20(x1)` and `and x4, x2, x5` as the data memory is accessed in the MEM stage and the source registers are read in the ID stage. Thus, the execution of the four instructions takes 9 clock cycles, not 8 clock cycles.

For your convenience, update `InstructionMemory.vhd` to have the following code to simulate the four instructions:

```
-- pipeline example #2
signal memory: t1KBMemory := (
    x"00", x"00", x"00", x"00",
    x"03", x"B1", x"40", x"01", -- ld x2, 20(x1)
    x"33", x"72", x"51", x"00", -- and x4, x2, x5
    x"33", x"64", x"61", x"00", -- or x8, x2, x6
    x"B3", x"04", x"22", x"00", -- add x9, x4, x2
    others => (others => '0')
);
```

Example #3:



Refer to “csi3102-assn4-prob2-ex3.jpg” file uploaded to YSCEC for a more clear image. This example demonstrates a pipeline flush due to branch penalty/misprediction (similar to Figure 4.59 in the textbook).

In this example, we execute the following instructions:

```
beq x1, x0, 40    // 0x02008463
add x2, x2, x2    // 0x00210133
add x2, x2, x2    // 0x00210133
add x2, x2, x2    // 0x00210133
add x2, x2, x2    // 0x00210133
add x2, x2, x2    // 0x00210133
add x2, x2, x2    // 0x00210133
add x2, x2, x2    // 0x00210133
add x2, x2, x2    // 0x00210133
add x2, x2, x2    // 0x00210133
ld  x2, 20(x1)    // 0x0140B103
```

with the following initial values for registers and data memory:

```
x2 = 0x0000000000000002
DataMemory[27:20] = 0x2AAAAAAAAAAAAAA
```

According to the instructions and the initial values, the only change in the programmer-visible state (except the PC) is:

```
x2 = 0x2AAAAAAAAAAAAAA
```

Note that although the three add instructions have been inserted into the 5-stage pipeline due to a branch misprediction, the processor flushes the latches of the instructions which should not have been executed. The flush results in a three-cycle penalty, so an update to register x2 occurs at clock cycle 9, not clock cycle 6. Also note that the value of register x2 does not change before clock cycle 9 as the programmer-visible state should not be affected by the branch misprediction.