VHDL Programming Assignment #2

Architecture of Computers (CSI3102-01), Spring 2020

Welcome to the second programming assignment for the Architecture of Computers (CSI3102-01) course at Yonsei University! By solving the last assignment, we believe you now have some sense of how to write digital circuits using VHDL, and how to compile and test your VHDL code using Xilinx Vivado.

In this assignment, we will implement the Register File (RF) and memory which play a crucial role in RISC-V Instruction Set Architecture (ISA). The RF serves as the working space for the datapath (i.e., the ALUs inside the CPU core), and the memory serves as a storage for instructions and data. The problem descriptions will heavily use the terms used by RISC-V ISA, so I suggest you review "Instructions: Language of the Computer" lectures and textbook contents before working on this assignment.

As we did for the previous assignment, we provide a `.zip` file containing `.vhd` files which provide skeleton VHDL code for the problems. You should modify/extend the `.vhd` files to solve the problems. After completing this programming assignment, you should upload your own `.zip` file containing the modified/extended `.vhd` files. When making your own `.zip` file, ***DO NOT* change the filenames of the `.vhd` files**.

The deadline for this assignment is **11:59pm on April 22nd, 2020 (Wed)**. No extension will be given.

Problem #1: Register File (20 Points)

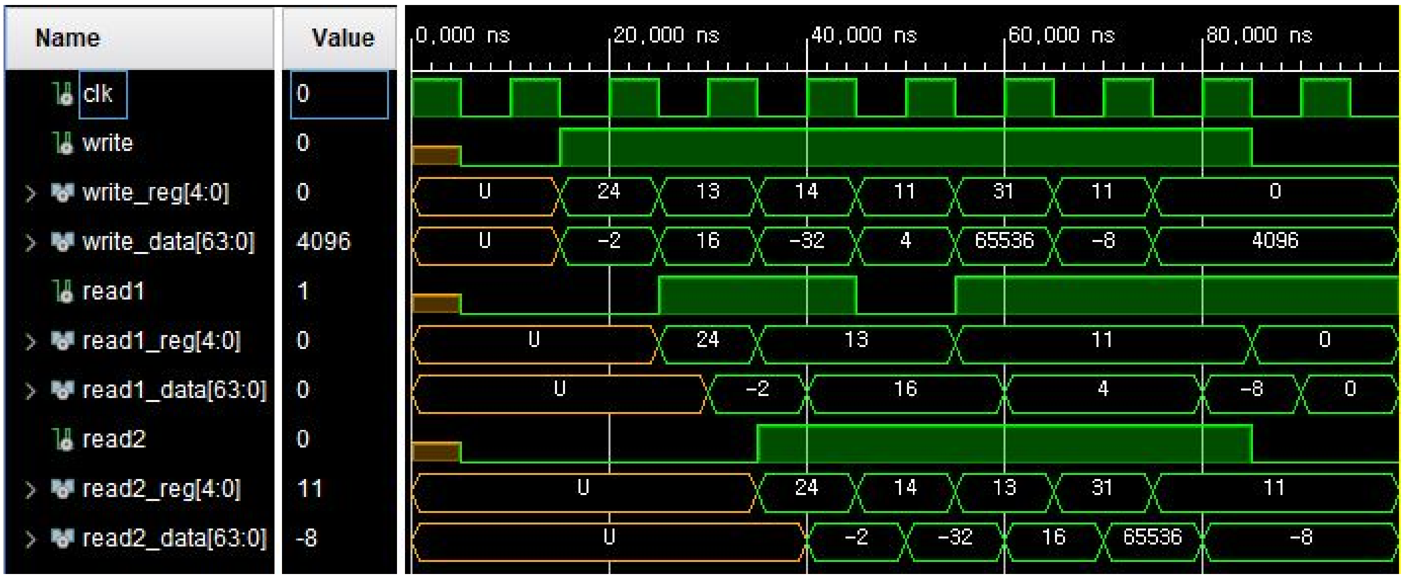
Implement a Register File (RF) as specified by the RISC-V ISA by extending `RegisterFile.vhd`. According to the ISA, the RF consists of 32 registers, named as `x0`, `x1`, ..., and `x31`, and `x0` is hardwired to a value of zero. Each register is 64-bit wide. Initially, all the registers should have a value of zero.

The RF should have two read ports and one write port. That is, the RF should be able to read data from two different registers and write data to another register at the same time. The three ports should be implemented with nine VHDL signals: `read1`, `read1_reg`, `read1_data`, `read2`, `read2_reg`, `read2_data`, `write`, `write_reg`, and `write_data`. Signals `read1`, `read2`, and `write` indicate whether we are reading data from (`read1`, `read2`) and/or writing data to (`write`) the RF.

When reading data from the RF, `read1` and/or `read2` are set to 1, and `read1_reg` and/or `read2_reg` denote the registers to read the data from. With respect to the values of the read-relevant signals, `read1_data` and `read2_data` should contain the values stored in the target registers. When writing data to the RF, `write` is set to 1, `write_reg` denotes the register to write the data to, and `write_data` contains the data. The RF should update the value of the target register so that it contains the value of `write_data` signal. That is, any following reads to the target register should return the updated value, not the previous value stored in the target register. Any writes to register `x0` should be ignored as `x0` is hardwired to a value of zero.

All the behaviors of the RF should be driven by the rising edges of the clock signal (`clk`).

Example:


Name
Value
0,000 ns
20,000 ns
40,000 ns
60,000 ns
80,000 ns
clk 0
write 0
write_reg[4:0] 0: U, 24, 13, 14, 11, 31, 11, 0
write_data[63:0] 4096: U, -2, 16, -32, 4, 65536, -8, 4096
read1 1
read1_reg[4:0] 0: U, 24, 13, 11, 0
read1_data[63:0] 0: U, -2, 16, 4, -8, 0
read2 0
read2_reg[4:0] 11: U, 24, 14, 13, 31, 11
read2_data[63:0] -8: U, -2, -32, 16, 65536, -8

Signals `{read1,read2,write}_reg` are shown in unsigned decimal numbers, and signals `{read1,read2,write}_data` are shown in signed decimal numbers.

Problem #2: Byte-Addressable Memory (30 Points)

Extend `Memory.vhd` and implement a 4-KiB byte-addressable memory. As the size of the memory is 4 KiB (= 2^{12} bytes), the range of the valid memory addresses is from 0 to $111111111111_{(2)}$.

In real-world scenarios, the memory typically takes many cycles to read/write data. That is, the memory may take more than 1 cycle to complete a read/write request. We will take such a behavior into account. Our memory takes 1 cycle to read/write 1-byte data, 2 cycles to read/write 2-byte data, 4 cycles to read/write 4-byte data, and 8 cycles to read/write 8-byte data.

Due to the size-dependent response times, our memory indicates whether it is ready to serve a new read/write request through `ready` signal. When `ready` = 0, the memory does not accept a new read/write request; however, when `ready` = 1, the memory accepts a new read/write request at the upcoming rising edge of the clock signal `clk`. Therefore, your implementation of the memory should ignore any new requests until it completes the currently-serving request.

When our memory is not busy (i.e., `ready` = 1), our memory accepts a new read/write request through `ivalid`, `rdwr`, `addr`, `size`, and `(i|o)data` signals. A user of the memory sets `ivalid` to 1 when it wishes to send a new request to the memory. In other words, the memory should simply ignore the input signals when `ivalid` = 0.

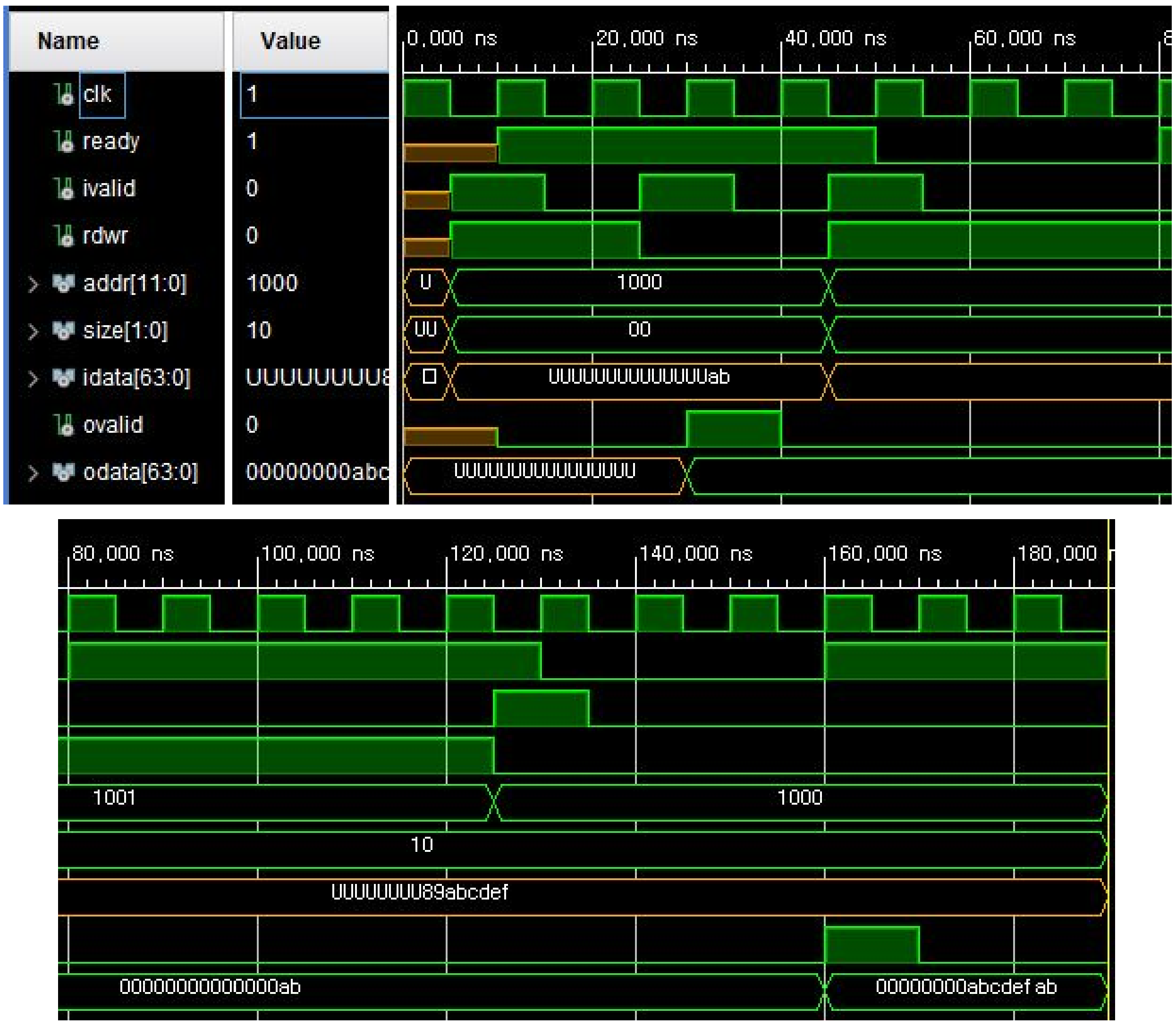
The `rdwr` signal indicates whether the new request is a read request (`rdwr` = 0) or a write request (`rdwr` = 1). First, when the new request is a write request, `addr` indicates the memory address the memory should start writing data from, `size` indicates the amount of data to write (`size` = 00 for writing 1 byte, `size` = 01 for writing 2 bytes, `size` = 10 for writing 4 bytes, and `size` = 11 for writing 8 bytes), and `idata` stores the data to write to the memory. Then, the memory completes the write request after spending a few cycles for updating the destination memory address(es). After that, the memory should accept a new read/write request by setting `ready` to 1.

Second, when the new request is a read request, `addr` indicates the memory address the memory should start reading data from and `size` indicates the amount of data to read; however, the memory utilizes the `odata` signal as an output signal for returning the requested data. The memory then reads the requested amount of data, saves the loaded data to the `odata` signal, and specifies that the output is valid by setting `ovalid` to 1. At the same time, the memory sets the value of the `ready` signal to 1 as it has completed the read request and is now able to accept a new memory request. Note that the data stored in the `odata` signal should be valid for only one cycle; when one clock cycle elapses after setting `ovalid` to 1, `ovalid` should be updated to 0 to indicate that the value of the `odata` signal is no longer valid.

When accessing the memory with smaller-than-8-bytes data sizes, the smaller-than-8-bytes value is stored in the right-most bits (i.e., 7/15/31/63-downto-0 for 1/2/4/8-byte data). The remaining left-most bits should be filled with zeros.

All the behaviors of the memory should be driven by the rising edges of the clock signal (`clk`).

Example:



'size' is shown in binary numbers, and 'idata' and 'odata' are shown in hexadecimal numbers.

Problem #3: RF-Augmented Arithmetic Logic Unit (50 Points)

Extend `ALUWithRF.vhd` and implement an Arithmetic Logic Unit (ALU) whose source and destination operands are registers, not input/output signals.

The ALU has four input signals for performing arithmetic operations with the RF: op for specifying which arithmetic operation to perform, `rs1` and `rs2` for specifying the source registers, and `rd` for specifying the destination register. In addition, the ALU provides two signals which we can use to populate new values to registers and inspect the current values of the registers: `idata` for supplying new value to a register, `odata` for inspecting the value of a register. How the ALU behaves with respect to the input signals is as follows:

op	**Operation**	**Latency**	**Notes**
`001`	`rd = rs1 + rs2`	4 cycles	Add two signed integers
`010`	`rd = rs1 - rs2`	4 cycles	Subtract one signed integer from another
`011`	`rd = rs1 * rs2`	4 cycles	Multiply one signed integer with another
`100`	`rd = rs1 / rs2`	4 cycles	Divide one signed integer from another
`101`	`rd = idata`	2 cycles	Assign a new value to a register
`110`	`odata = rs1`	3 cycles	Inspect the value stored in a register

'Latency' column indicates how many clock cycles the ALU takes to complete the operation. It is similar to the memory's 1/2/4/8-cycle latency.

When solving this problem, you should utilize the VHDL code you wrote to solve the first problem, the RF. You will quickly realize that the RF having two read ports and one write port is suitable for solving this problem. The skeleton `ALUWithRF.vhd` we provide already instantiates the RF for your convenience, so please do not implement a separate RF for solving this problem.

All the behaviors of the ALU should be driven by the rising edges of the clock signal `clk`. You may assume that no overflow occurs for any operations.

Example #1: x31 = x1 + x2 takes 4 clock cycles to complete.

- At rising edge #1,
  - The ALU reads its input signals op = 001, rd = 11111, rs1 = 00001, and rs2 = 00010.
- Between rising edges #1 and #2,
  - The ALU sets up the RF's input signals to read x1 and x2.
- At rising edge #2,
  - The RF reads the input signals as configured by the ALU.
- Between rising edges #2 and #3,
  - The RF fetches data from the registers.
  - The RF configures its output signal.
- At rising edge #3,
  - The ALU reads the source registers from the RF.
- Between rising edges #3 and #4,
  - The ALU adds the source registers.
  - The ALU configures the RF's input signals to write the result to x31.
- At rising edge #4,
  - The RF reads its input signals.
- Between rising edges #4 and #5,
  - The RF writes the result to register x31.
  - The ALU configures its output signal to indicate that it's ready to perform a new operation.

Example #2: Writing a new value to register x24 takes 2 clock cycles to complete.

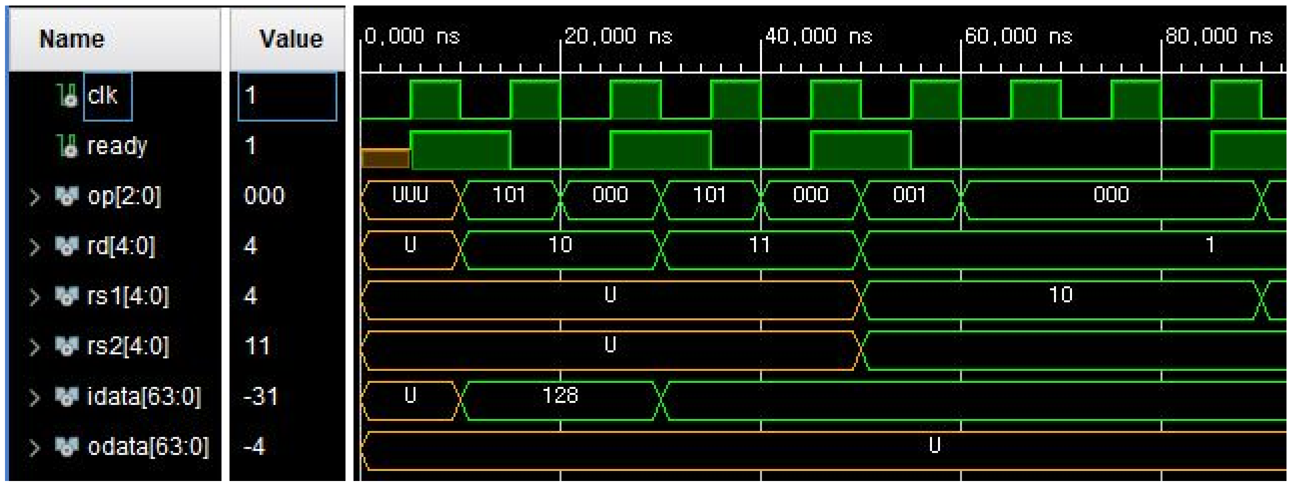
- At rising edge #1,
  - The ALU reads its input signals.
- Between rising edges #1 and #2,

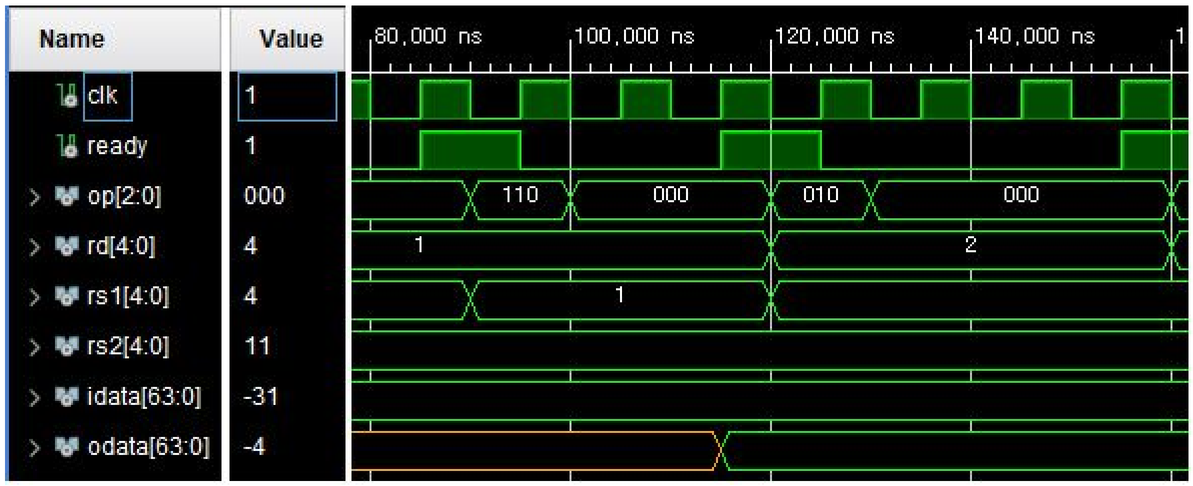
    - The ALU sets up the RF's input signals to write a new value to register x24.
- At rising edge #2,
    - The RF reads the input signals as configured by the ALU.
- Between rising edges #2 and #3,
    - The RF writes the new data to register x24.
    - The ALU configures its output signal to indicate that it's ready to perform a new operation.

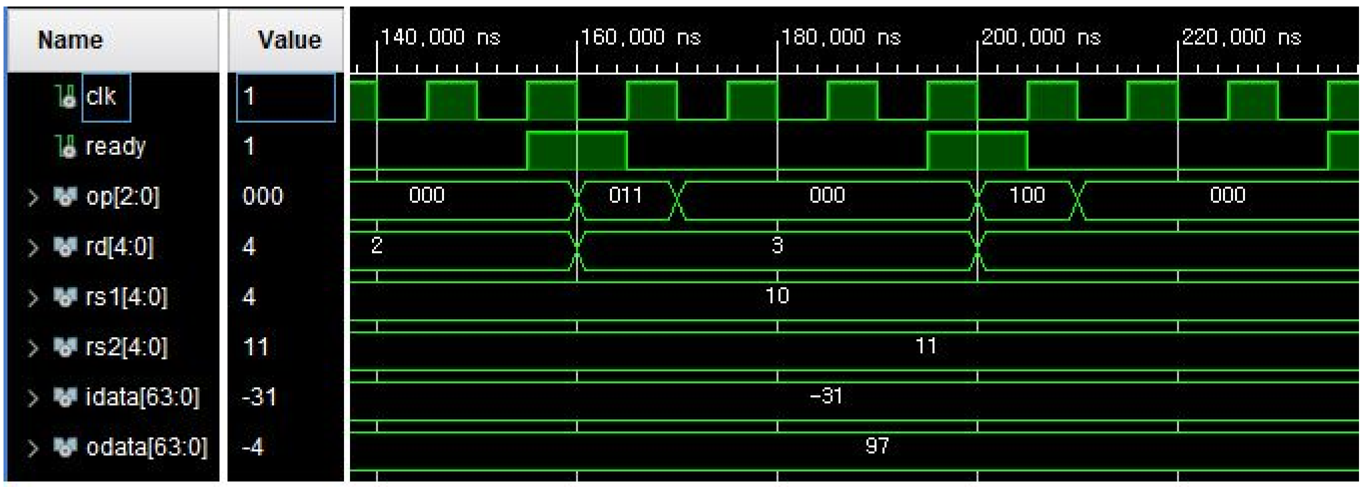
Example #3: Reading the value of register x20 takes 3 cycles to complete.

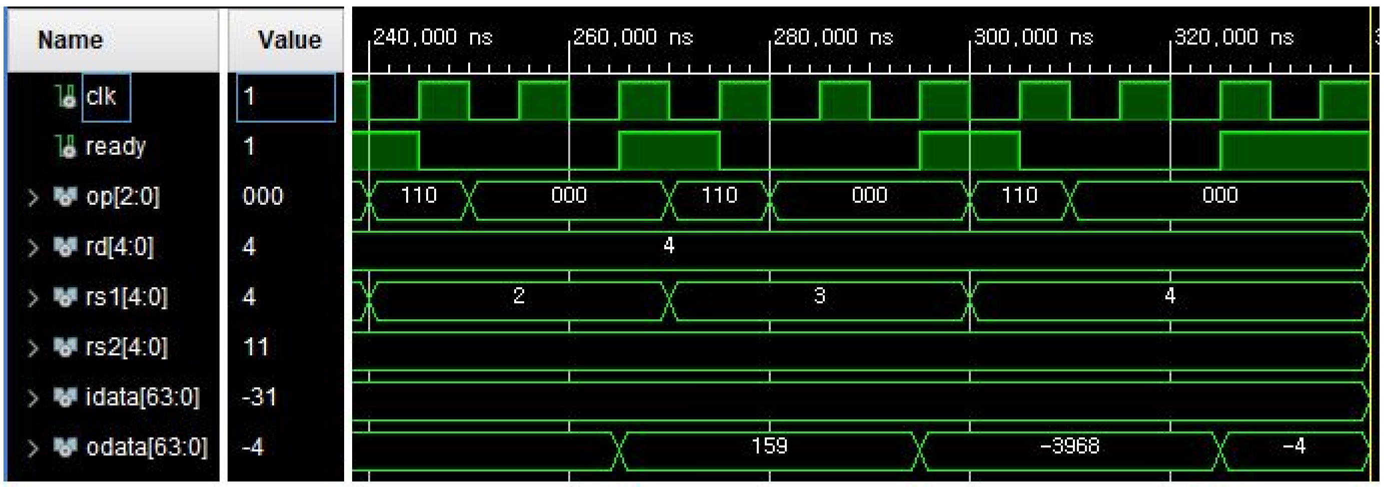
- At rising edge #1,
    - The ALU reads its input signals.
- Between rising edges #1 and #2,
    - The ALU sets up the RF's input signals to read the value stored in register x20.
- At rising edge #2,
    - The RF reads the input signals as configured by the ALU.
- Between rising edges #2 and #3,
    - The RF reads the value stored in register x20.
    - The RF configures its output signals to contain the value stored in register x20.
- At rising edge #3,
    - The ALU reads the value of register x20 from the RF's output signals.
- Between rising edges #3 and #4,
    - The ALU configures its output signals to expose the value of register x20.
    - The ALU configures its output signals to indicate that it's ready to perform a new operation.

Example #4: Waveforms









‘op’ signal is shown in binary numbers,
‘rd’, ‘rs1’, and ‘rs2’ signals are shown in unsigned decimal numbers,
and ‘idata’ and ‘odata’ are shown in signed decimal numbers.

Signal the ALU to write 128 to register x10 @ 15 ns (rising edge #2)
Signal the ALU to write -31 to register x11 @ 35 ns (rising edge #4)
Signal the ALU to perform x1 = x10 + x11 @ 55 ns (rising edge #6)
Signal the ALU to read the value of x1 @ 95 ns (rising edge #10)
Signal the ALU to perform x2 = x10 - x11 @ 125 ns (rising edge #13)
Signal the ALU to perform x3 = x10 * x11 @ 165 ns (rising edge #17)
Signal the ALU to perform x4 = x10 / x11 @ 205 ns (rising edge #21)
Signal the ALU to read the value of x2 @ 245 ns (rising edge #25)
Signal the ALU to read the value of x3 @ 275 ns (rising edge #28)
Signal the ALU to read the value of x4 @ 305 ns (rising edge #31)