VHDL Programming Assignment #1

Architecture of Computers (CSI3102-01), Spring 2020

Welcome to the very first programming assignment for the Architecture of Computers (CSI3102-01) lecture at Yonsei University! As we discussed earlier in the semester, our ultimate goal is to implement a simple Central Processing Unit (CPU) using VHDL Hardware Description Language (HDL). This programming assignment serves as the first step toward achieving our ultimate goal.

The purpose of this programming assignment is to make you get familiar with VHDL itself and the tools we use to compile and test VHDL code. Before working on this programming assignment, I suggest you review the "VHDL Basics" and the "Compiling and Testing Your VHDL Code" lectures.

As the starting point, 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**.

The deadline for submitting your VHDL code is **11:59pm on April 8th, 2020 (Wed)**. No extension will be given.

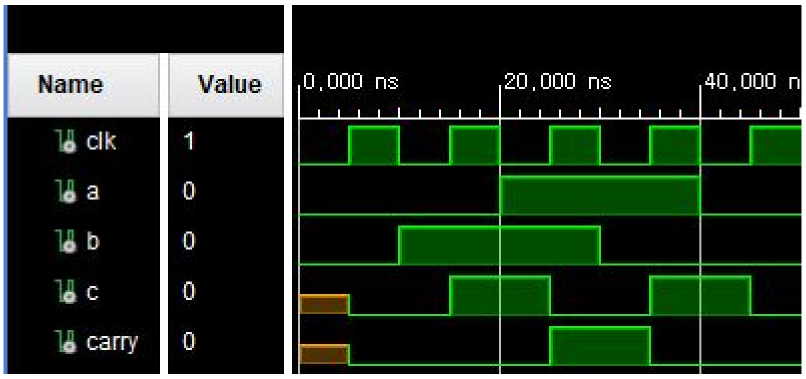
Without further ado, let's move onto the problems!

Problem #1: Single-Bit Adder (10 Points)

Implement a single-bit adder by extending `SingleBitAdder.vhd`.

The adder should produce two outputs: `c` and `carry`. If the addition of two input signals a and b incurs an overflow, `carry` should be set to 1; otherwise, `carry` should be set to `0`.

The adder should be driven by the clock signal `clk` and update the output signals when `clk` changes from `0` to 1. Assume that both input signals are unsigned integers.

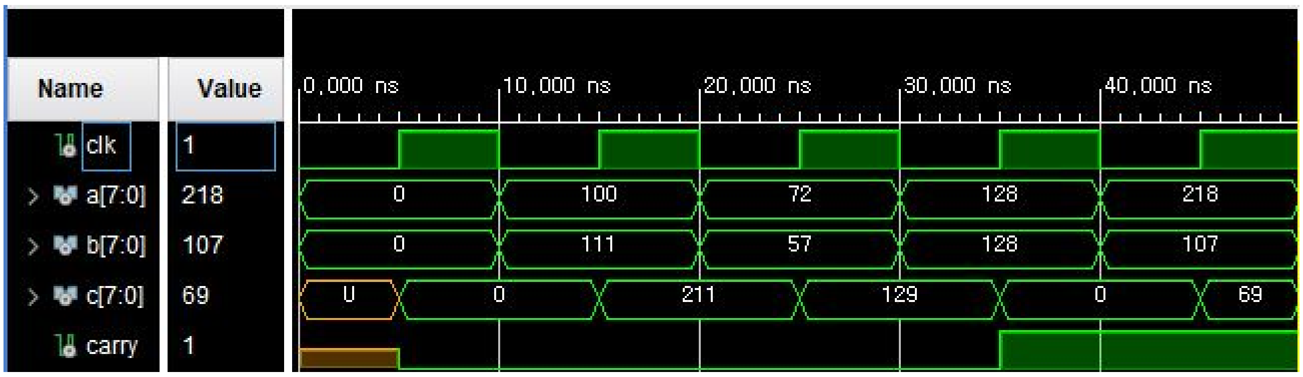


Problem #2: Multi-Bit Adder (10 Points)

Implement a multi-bit adder by extending `MultiBitAdder.vhd`. In particular, you should implement an 8-bit adder.

The adder should produce two outputs: c and `carry`. If the addition of two input signals a and b incurs an overflow, `carry` should be set to 1; otherwise, `carry` should be set to `0`.

The adder should be driven by the clock signal `clk` and update the output signals when `clk` changes from `0` to 1. Assume that both input signals are unsigned integers.



Name	Value	0 ns	10 ns	20 ns	30 ns	40 ns
clk	1					
a[7:0]	218	0	100	72	128	218
b[7:0]	107	0	111	57	128	107
c[7:0]	69	U, 0	0, 211	211, 129	129, 0	0, 69
carry	1					

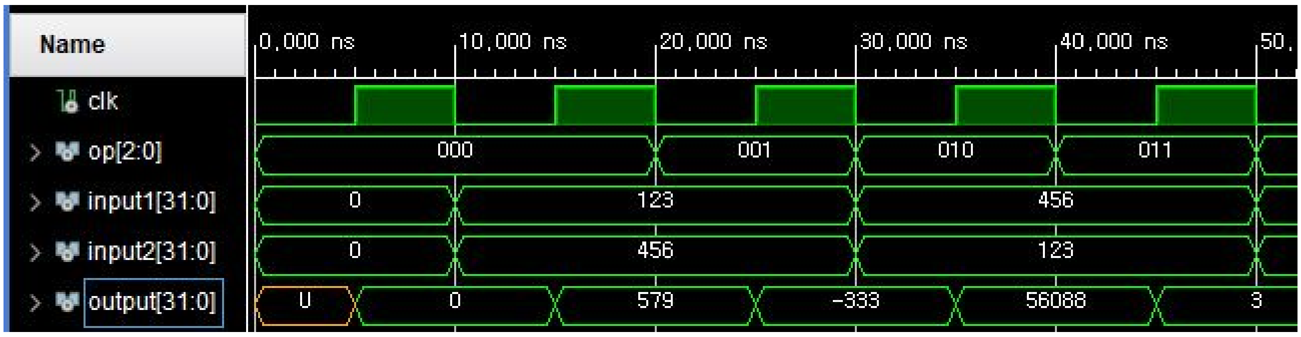
Problem #3: Arithmetic Logic Unit (20 Points)

Implement an ALU by extending `ALU.vhd`. Your ALU should support basic arithmetic and logic operations.

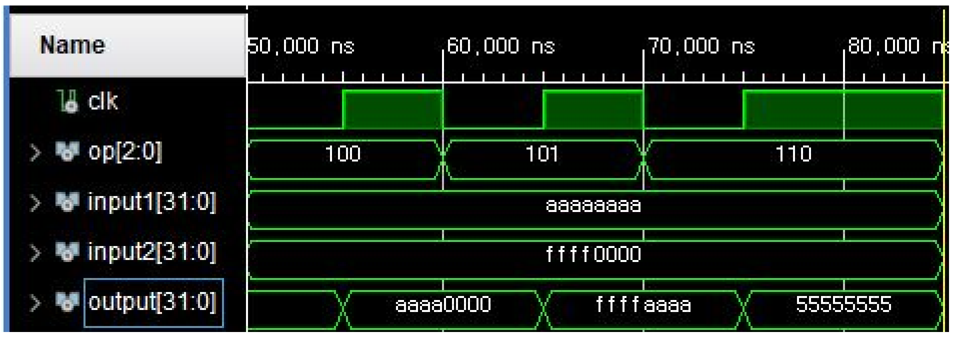
The ALU has four input signals: the clock signal `clk`, the operation specifier op, and input operations `input1` and `input2`. With respect to the values of the input signals, your ALU should evaluate its output signal `output` as follows:

op	Operation	Notes
`000`	`output = input1 + input2`	Add two signed integers
`001`	`output = input1 - input2`	Subtract one signed integer from another
`010`	`output = input1 * input2`	Multiply two signed integers
`011`	`output = input1 / input2`	Divide two signed integers
`100`	`output = input1 AND input2`	Element-wise logical AND
`101`	`output = input1 OR input2`	Element-wise logical OR
`110`	`output = NOT input1`	Element-wise logical NOT

The ALU should be driven by the clock signal and update the output signal when `clk` changes from `0` to 1. Assume that no overflow/underflow or division-by-zero occurs.



*input1, input2, input3 are shown in decimal from 0 ns to 50 ns.

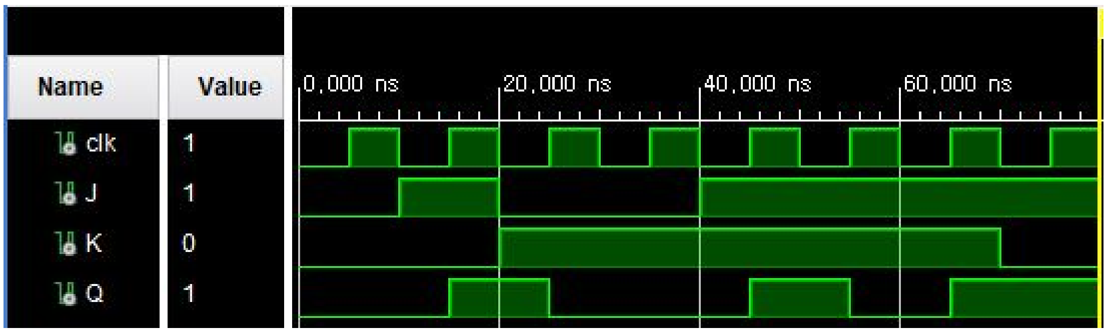


*input1, input2, input3 are shown in hexadecimal from 50 ns to 80 ns.

Problem #4: JK Flip-Flop (10 Points)

Implement the JK flip-flop by extending `JKFlipFlop.vhd`. The characteristic equation of the JK flip-flop is $Q_{next} = J\overline{Q} + \overline{K}Q$ where `Q` denotes the current internal state of the JK flip-flop, `Q_next` denotes the next internal state (i.e., at the next rising edge of the clock signal `clk`), and J and K denote the input signals to the flip-flop.

The flip-flop should be driven by the clock signal `clk` and update the output signal `Q` when `clk` changes from `0` to 1. The flip-flop's initial `Q` value should be `0`.



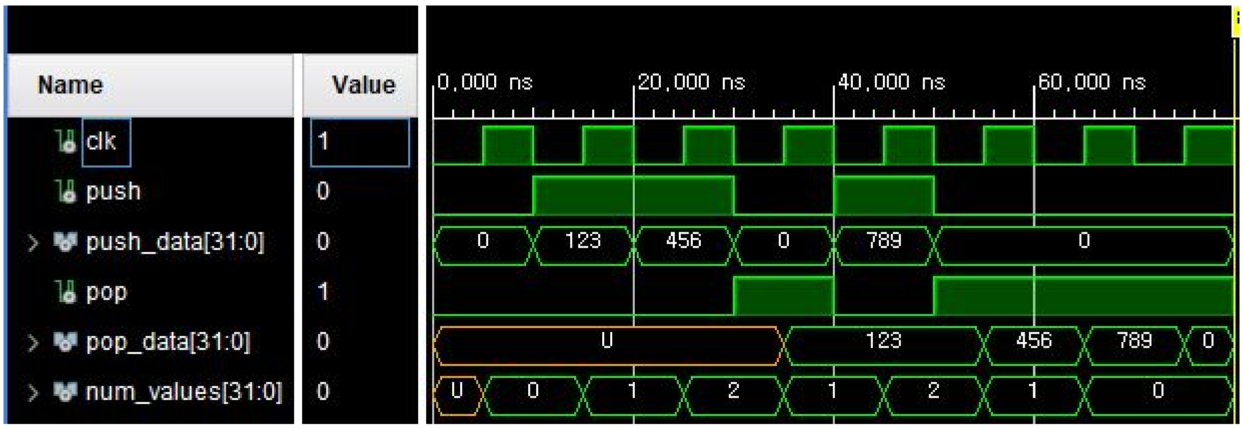
Problem #5: Queue (10 Points)

Implement a queue by extending `Queue.vhd`.

The queue should support two operations: push and pop. To push a value into the queue, the values of input signals `push` and `push_data` will be set to 1 and the value, respectively. When popping a value from the queue, the value of input signal `pop` will be set as 1, and the queue should evaluate output signal `pop_data` as the value being popped. When `pop` is set to 1, but the queue doesn't have any value, `pop_data` should be set as 0. You may assume that there will be no more than 32 elements in the queue at any given time. You may also assume that there are no cases where both `push` and `pop` are set to 1.

To monitor the dynamic status of the queue, there is an additional output signal called `num_values` which shows the number of values stored in the queue. Whenever the clock signal incurs a rising edge, `num_values` should be updated to a proper value.

The queue should be driven by the clock signal `clk` and update the output signals when `clk` changes from `0` to 1. The queue should be empty at the beginning of a simulation.



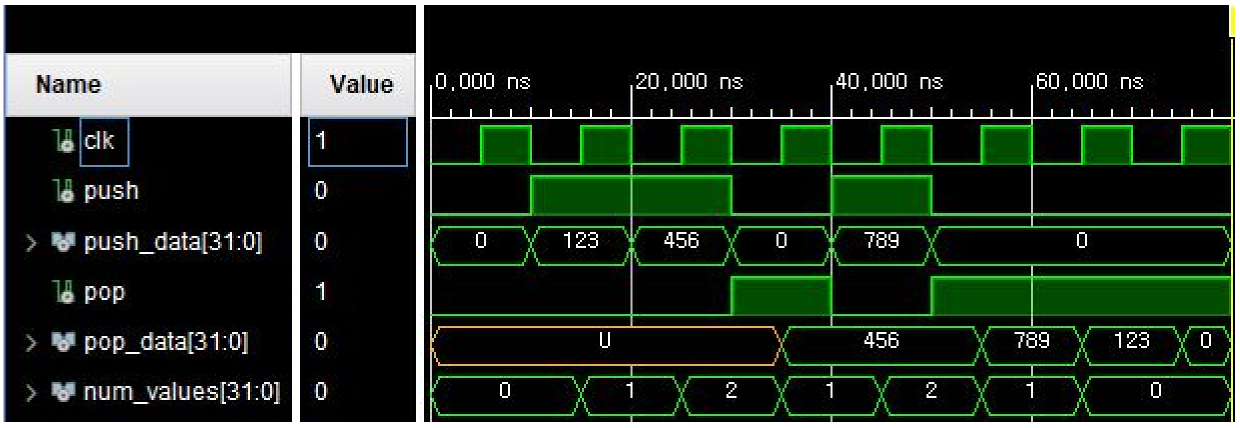
Problem #6: Stack (10 Points)

Implement a stack by extending `Stack.vhd`.

The stack should support two operations: push and pop. To push a value into the stack, the values of input signals `push` and `push_data` will be set to 1 and the value, respectively. When popping a value from the stack, the value of input signal `pop` will be set as 1, and the stack should evaluate output signal `pop_data` as the value being popped. When `pop` is set to 1, but the stack doesn't have any value, `pop_data` should be set as 0. You may assume that there will be no more than 32 elements in the stack at any given time. You may also assume that there are no cases where both `push` and `pop` are set to 1.

To monitor the dynamic status of the stack, there is an additional output signal called `num_values` which shows the number of values stored in the stack. Whenever the clock signal incurs a rising edge, `num_values` should be updated to a proper value.

The stack should be driven by the clock signal `clk` and update the output signals when `clk` changes from `0` to 1. The stack should be empty at the beginning of a simulation.



Problem #7: Elevator (30 Points)

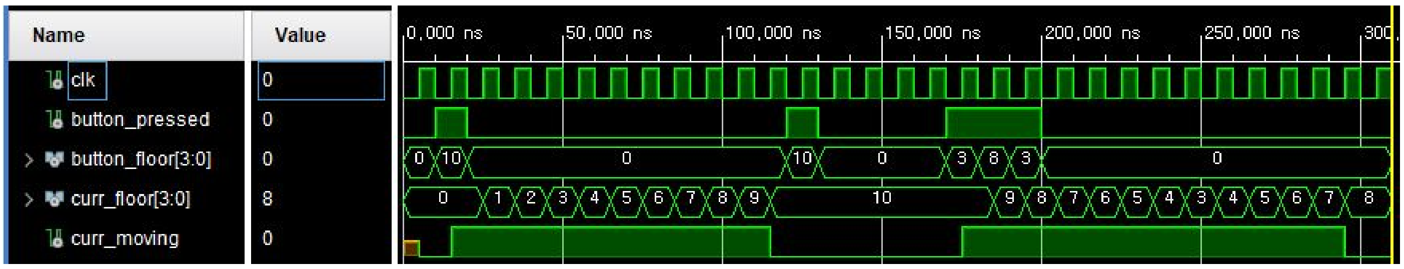
Implement a simple elevator by extending `Elevator.vhd`.

To emulate a person pressing the elevator's button, our elevator has two input signals `button_pressed` and `button_floor`. When the elevator's button is pressed at some floor, `button_pressed` is set to 1 and `button_floor` is set to which of the floors the button is pressed at. Assume that only one button gets pressed at a time.

Our elevator works in a very simple way; it visits the floors in the order of button presses. For example, if the button at floor 3 is pressed, then the button at floor 5, and the button at floor 8 later, our elevator visits the third floor first, the fifth floor second, and the eighth floor last regardless of which floor our elevator is currently at.

Our elevator also shows which floor it is currently at and whether it is moving through two output signals `curr_floor` and `curr_moving`. `curr_floor` should be set as the current floor of the elevator. `curr_moving` should be set to 1 if the elevator is currently moving to the next floor, and 0 if the elevator is not moving. Assume that the elevator can move one floor per clock cycle. For example, when the elevator is currently at the third floor at a rising edge of the clock signal `clk`, the elevator can be at either the second floor or the fourth floor at the next rising edge of the clock signal. When there are no further floors to visit, `curr_moving` should be set as 0. When the elevator is not moving and the button at the elevator's current floor is pressed, the elevator ignores the button press.

The elevator should be driven by the clock signal and update the output signals when `clk` changes from `0` to 1. The elevator should be at the 0-th floor at the beginning of a simulation.



- Button press @ 10th floor, and the elevator starts moving.
- The elevator reaches the 10th floor, so it stops.
- Button press @ 10th floor; however, the elevator is at the 10th floor and is not moving, so ignored.
- Button press @ 3rd floor, and the elevator starts moving.
- Button press @ 8th floor. The floors to serve become: 3rd, 8th.
- Button press @ 3rd floor; however, the 3rd floor is already registered to be served, so ignored.
- The elevator reaches the 3rd floor. The remaining floor to serve is the 8th floor.
- The elevator serves the 8th floor, and stops moving as there are no more floors to serve.