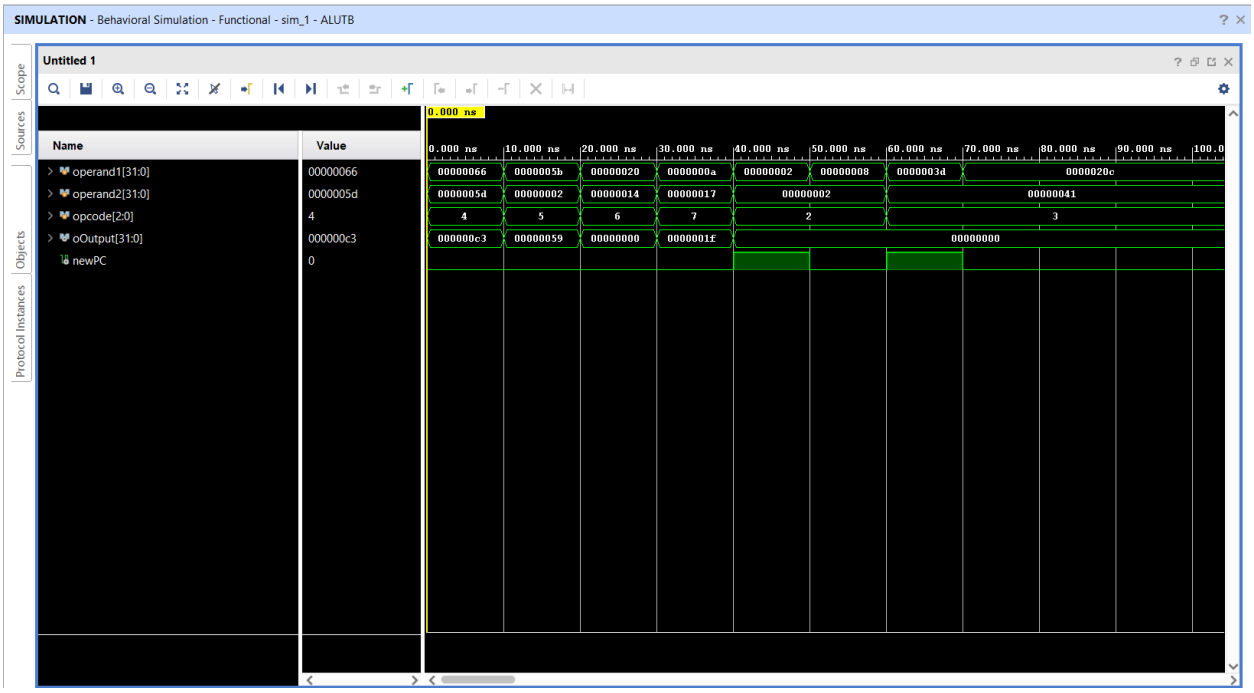
Francisco Soriano
5119989

Final Project Report

Waveforms

ALU



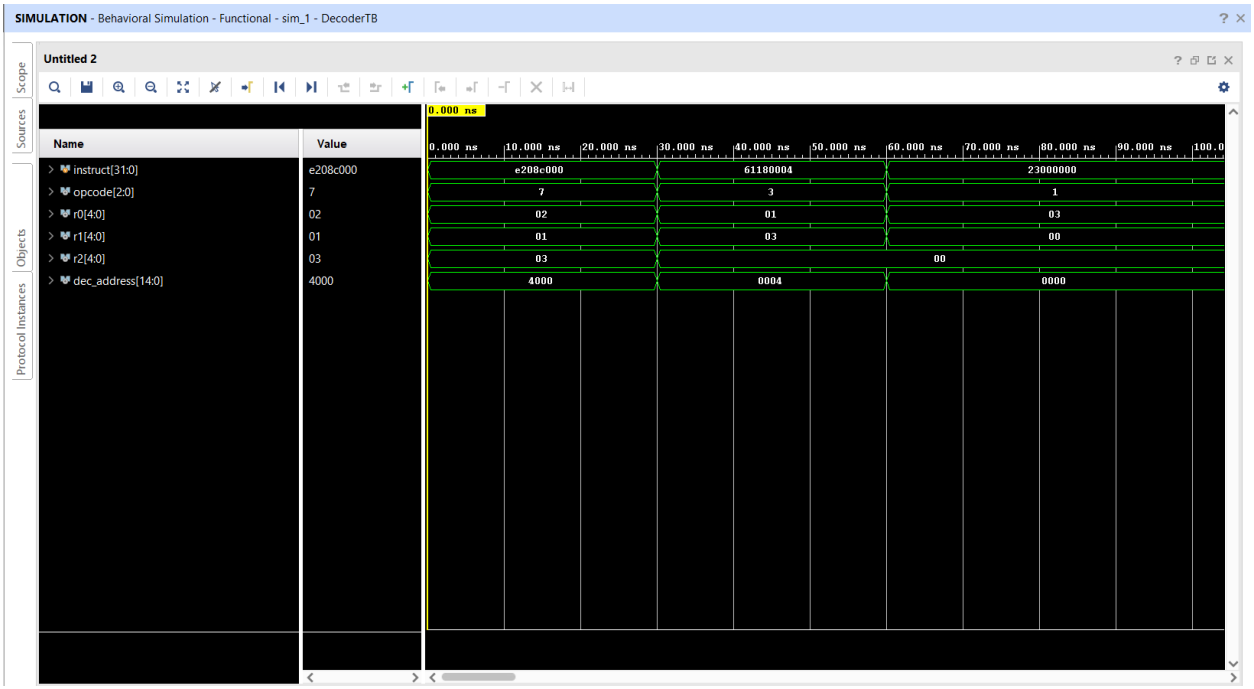
The waveform above corresponds to a testbench made to simulate an Arithmetic Logic Unit (ALU) module. This ALU is a fundamental building block of the CPU and can calculate the results of a wide variety of basic arithmetical computations.

The testbench initializes two 32-bit operands (operand1 and operand2) and a 3-bit opcode. The opcode determines the operation to be performed on the operands. The ALU_UUT (ALU Under Test) module takes these operands and the opcode as inputs and outputs a 32-bit result (oOutput) and a newPC signal.

The testbench then sequentially assigns different values to the operands and the opcode, with a delay of 10 time units between each operation. For example, in the first operation, operand1 is assigned a value of 102, operand2 a value of 93, and opcode a value of 4. After a delay of 10 time units, the values of the operands and the opcode are changed, and this process is repeated several times.

The waveform resulting from this testbench shows the changes in the values of operand1, operand2, and opcode over time, as well as the corresponding changes in oOutput and newPC. Each operation is clearly separated by the delay which makes it overall easier to observe the effect of different operations on the output.

Decoder



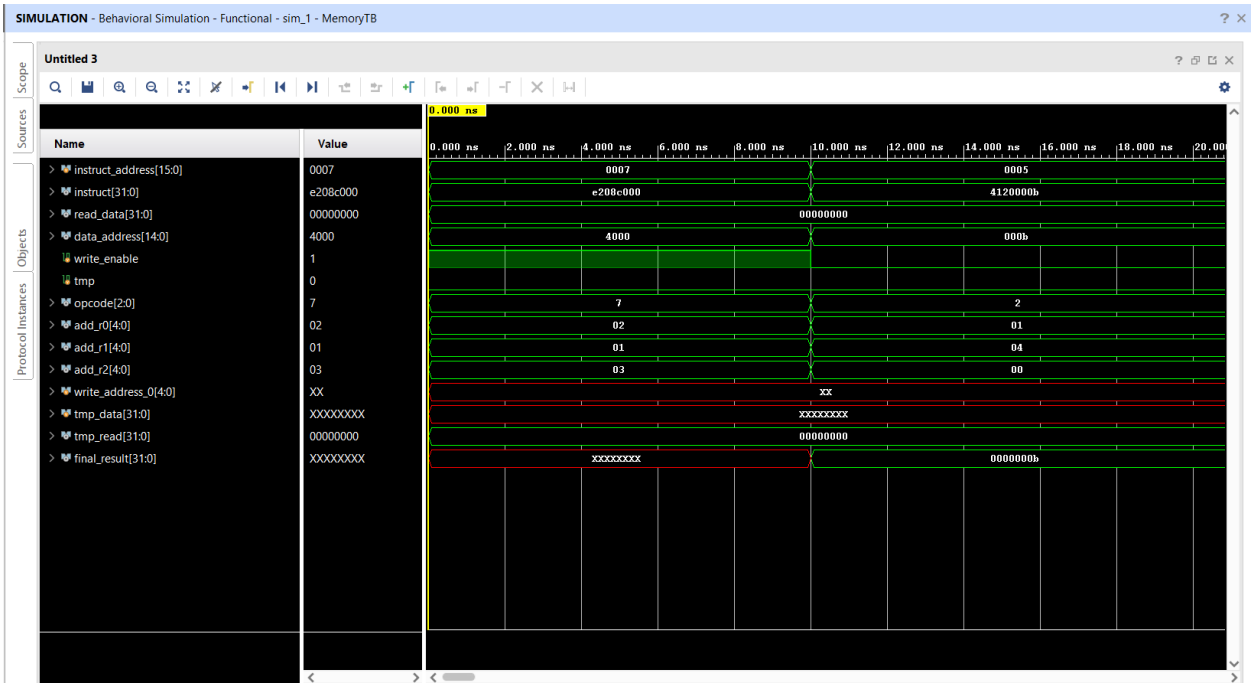
The waveform above corresponds to a testbench made to simulate a Decoder module. A decoder is a combinational circuit that converts binary information from ‘n’ input lines to a maximum of 2^n unique output lines. A decoder is used to determine the operation to be performed and the operands to be used based on the instruction fetched from memory.

The testbench initializes a 32-bit instruction (instruct). The Decoder_UUT (Decoder Under Test) module takes this instruction as input and outputs a 3-bit opcode, three 5-bit register addresses (r0, r1, r2), and a 15-bit decoded address (dec_address).

The testbench then sequentially assigns different values to the instruction, with a delay of 30 time units between each operation. For example, in the first operation, instruct is assigned a value of 32'he208_c000. After a delay of 30 time units, the value of the instruction is changed, and this process is repeated several times.

The waveform resulting from the testbench shows the changes in the value of ‘instruct’ over time, as well as the corresponding changes in opcode, r0, r1, r2, and dec_address. Each operation is clearly separated by the delay, making it easy to observe the effect of different instructions on the output of the Decoder.

Memory



The waveform above corresponds to a testbench made to simulate a Memory module. This memory module includes an Instruction Memory, a Decoder, a Register File, and a Data Memory. These components work together to fetch, decode, and execute instructions, and to read and write data from and also to memory.

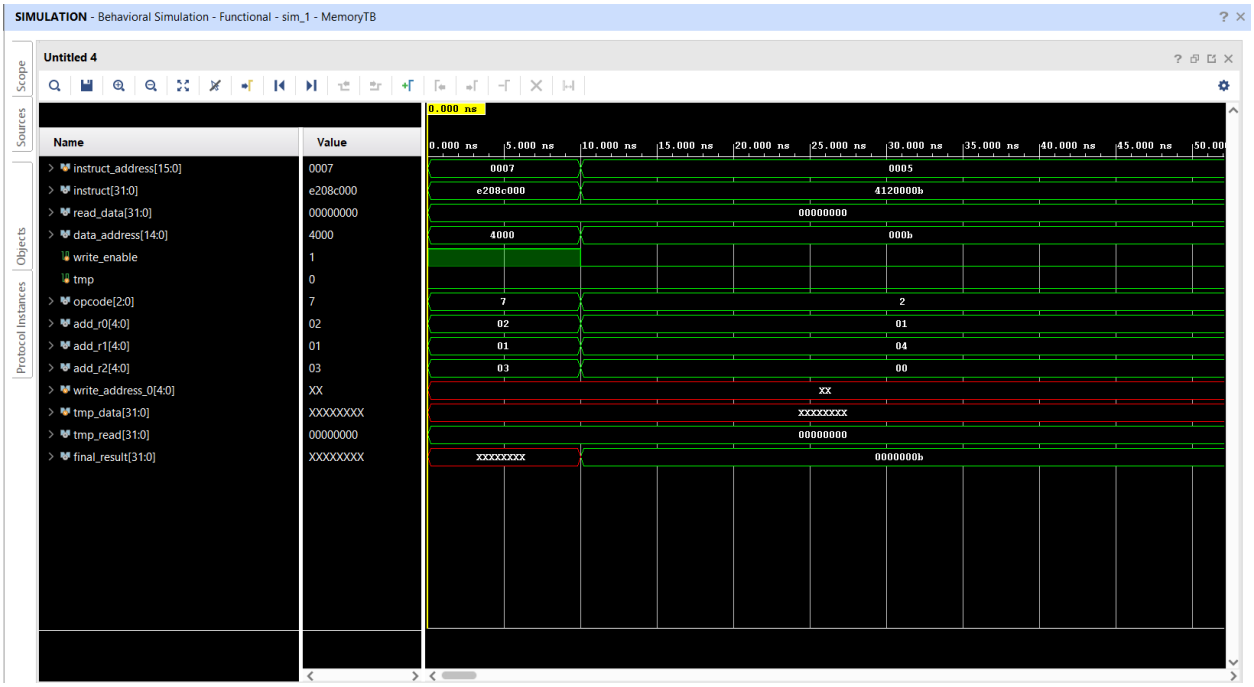
The testbench initializes a 16-bit instruction address (instruct_address), a write enable signal (write_enable), and a temporary register (tmp). The InstructionMemory_UUT (Instruction Memory Under Test) module takes the instruction address as input and outputs a 32-bit instruction (instruct). The Decoder_UUT (Decoder Under Test) module takes this instruction as input and outputs a 3-bit opcode and three 5-bit register addresses (add_r0, add_r1, add_r2), and a 15-bit decoded address (data_address). The RegisterFile_UUT (Register File Under Test) module takes these register addresses, the write enable signal, and a 32-bit temporary data (tmp_data) as inputs, and outputs a total of two 32-bit read data (read_data, tmp_read). The DataMemory_UUT (Data Memory Under Test) module takes the decoded address, the write enable signal, and the read data as inputs, and outputs a 32-bit final result (final_result).

The testbench then sequentially assigns different values to the instruction address and the write enable signal, with a delay of 10 time units between each operation. For example, in the first operation, instruct_address is assigned a value of 7, write_enable is set to 1, and tmp is set to 0. After a delay of 10 time units, the values of the instruction address and the write enable signal are changed, and this process is repeated.

The waveform resulting from this testbench shows the changes in the values of instruct_address, write_enable, and tmp over time, as well as the corresponding changes in instruct, opcode, add_r0, add_r1, add_r2, data_address, write_address_0, tmp_data, read_data, tmp_read, and

final_result. Each operation is clearly separated by the delay, making it easy to observe the effect of different operations on the output of the Memory module.

Register



The waveform above corresponds to a testbench made to simulate a Register File module. A Register File is a small set of registers that are visible at the programmer level of a computer processor and is a key component of the processor as it is used to quickly access data during computation.

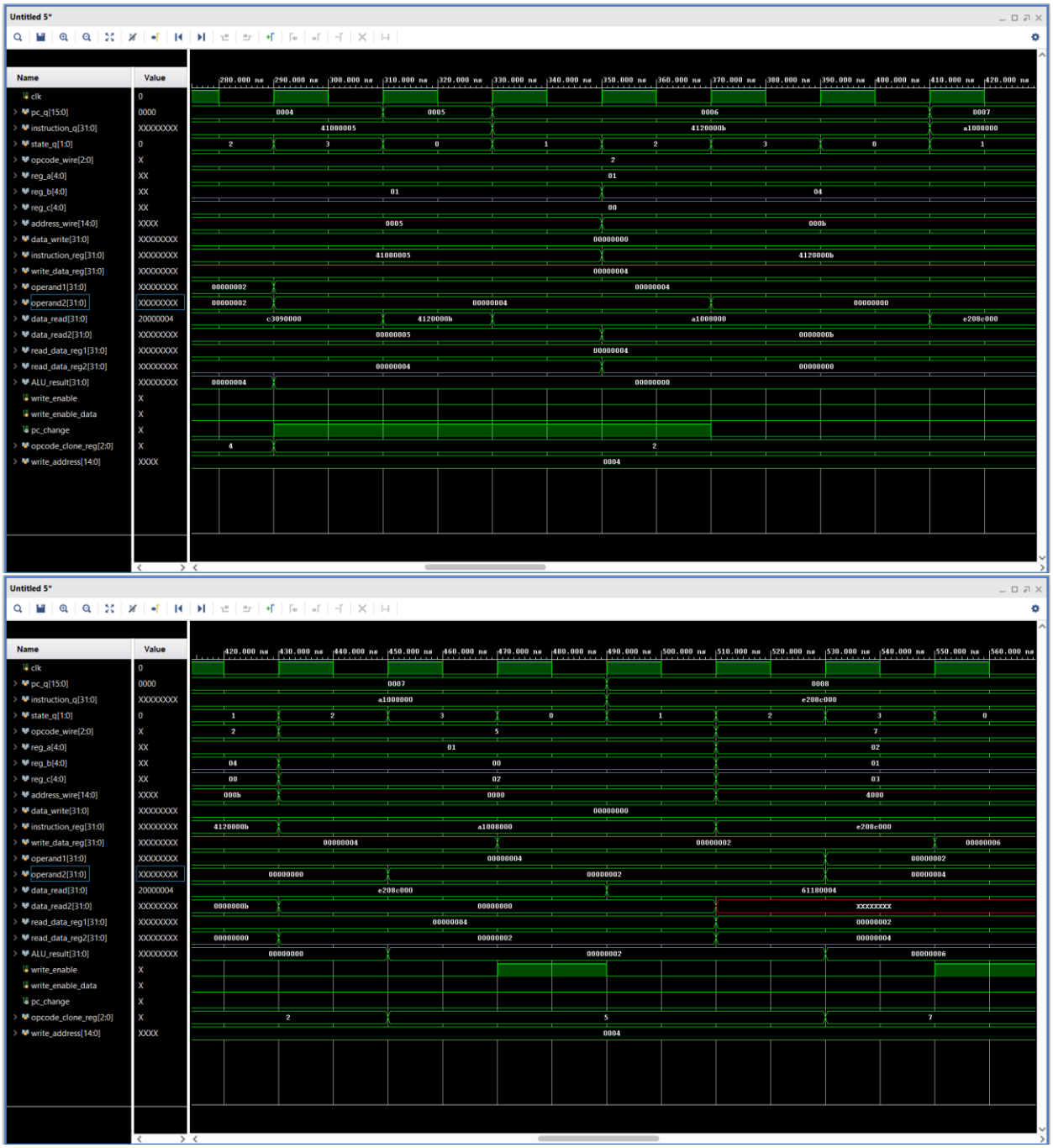
The testbench initializes two 5-bit read addresses (rd0 and rd1), a 5-bit write address (write), a write enable signal (write_enable), and a 32-bit write data (write_data). The RegisterFile_UUT (Register File Under Test) module takes these as inputs and outputs two 32-bit read data (read_data1 and read_data2).

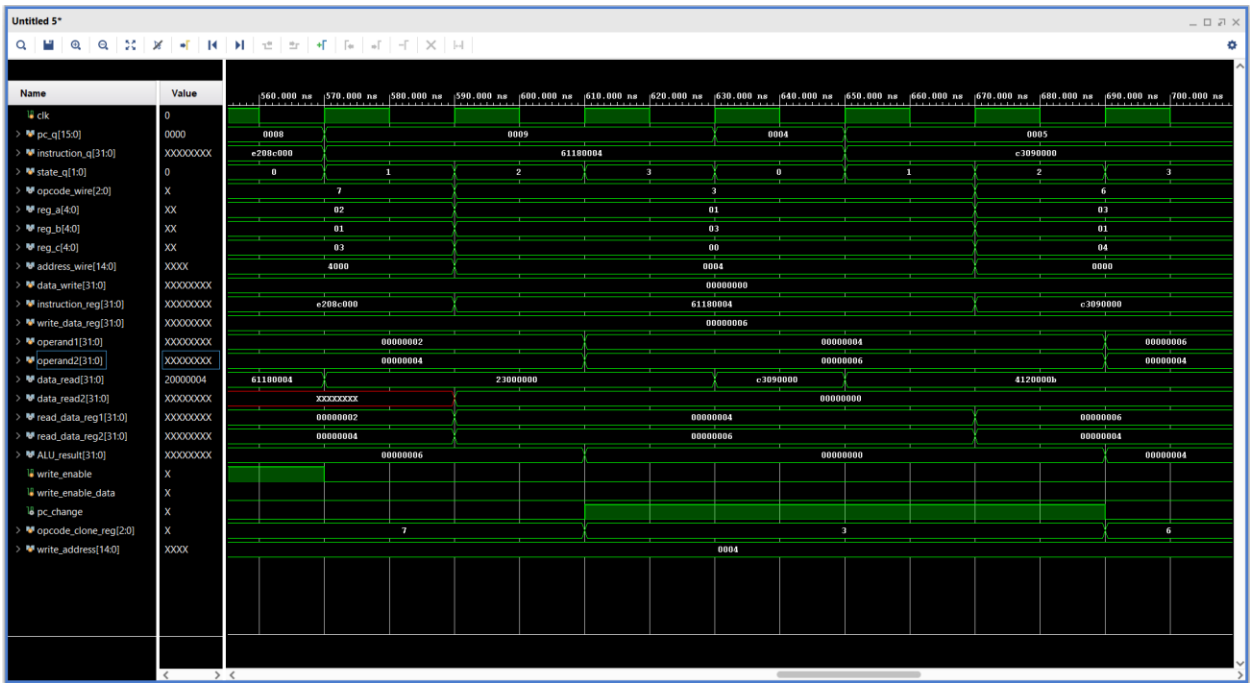
The testbench then sequentially assigns different values to the read addresses, the write address, the write enable signal, and the write data, with a delay of 10 time units between each operation. For example, in the first operation, rd0 is assigned a value of 9, rd1 is assigned a value of 13, and write_enable is set to 0. After a delay of 10 time units, the values of the write address, the write data, and the write enable signal are changed, and then the process repeats.

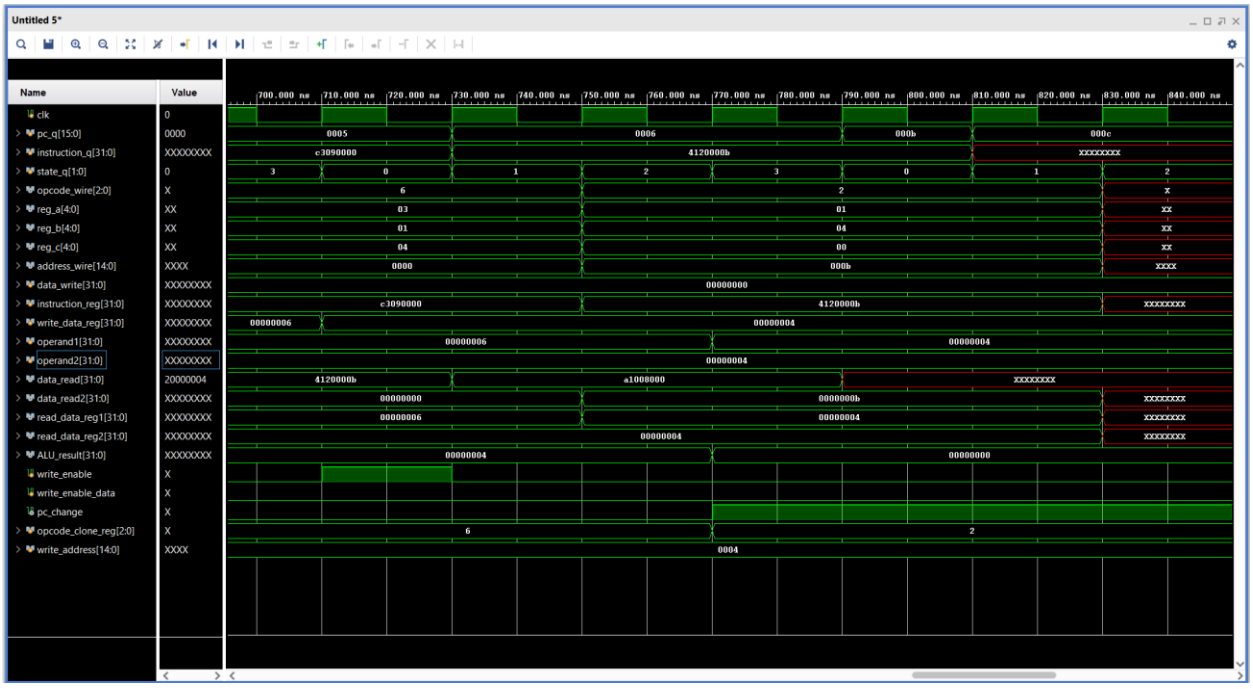
The waveform resulting from this testbench shows the changes in the values of rd0, rd1, write, write_enable, and write_data over time, as well as the corresponding changes in read_data1 and read_data2. Each operation would be clearly separated by the delay, making it easy to observe the effect of different operations on the output of the Register File.

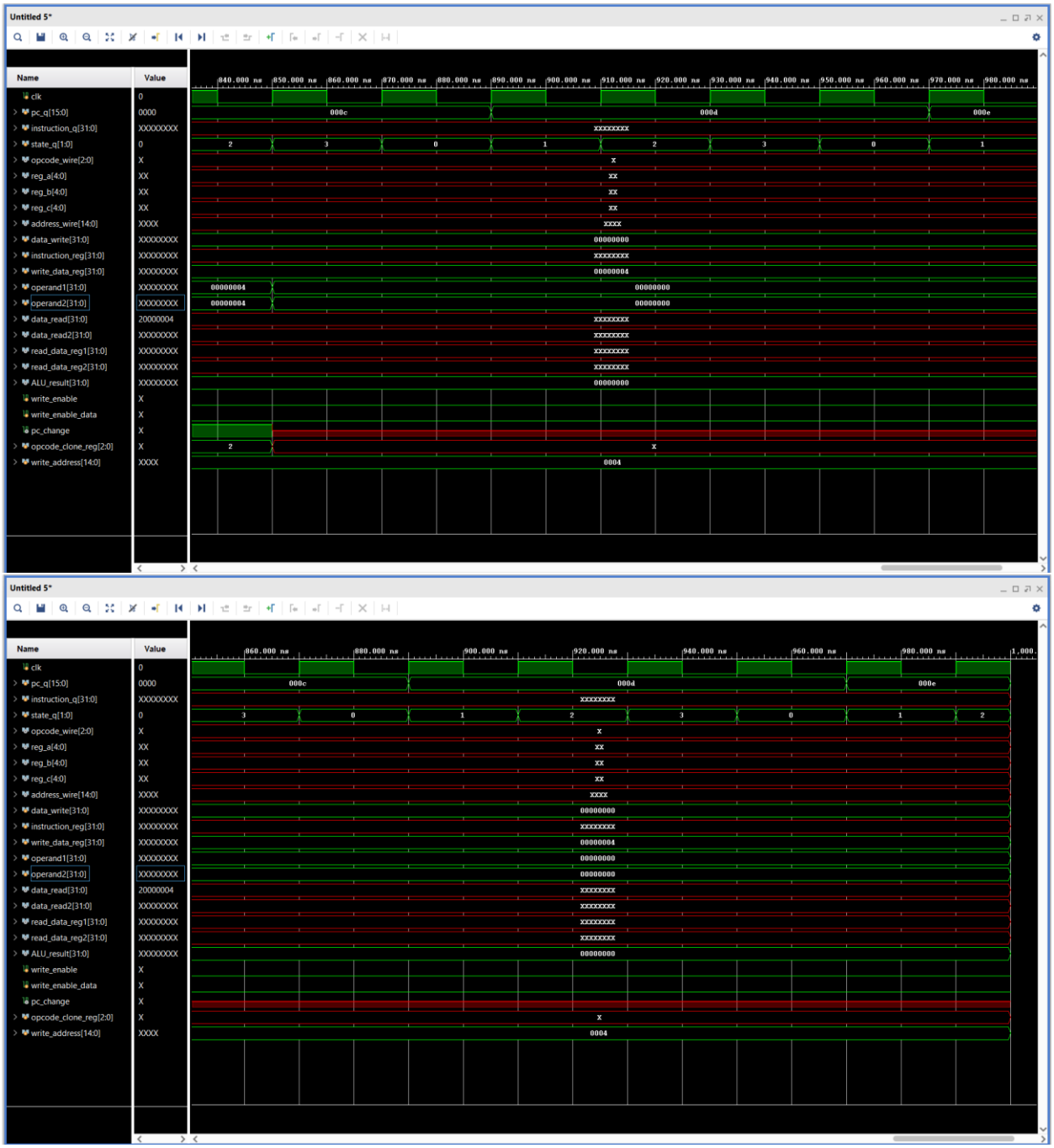
CPU


Untitled 5*
Name
Value
clk
pc_q[15:0]
instruction_q[31:0]
state_q[1:0]
opcode_wire[2:0]
reg_a[4:0]
reg_b[4:0]
reg_c[4:0]
address_wire[14:0]
data_write[31:0]
instruction_reg[31:0]
write_data_reg[31:0]
operand1[31:0]
operand2[31:0]
data_read[31:0]
data_read2[31:0]
read_data_reg1[31:0]
read_data_reg2[31:0]
ALU_result[31:0]
write_enable
write_enable_data
pc_change
opcode_clone_reg[2:0]
write_address[14:0]
Untitled 5*
Name
Value
clk
pc_q[15:0]
instruction_q[31:0]
state_q[1:0]
opcode_wire[2:0]
reg_a[4:0]
reg_b[4:0]
reg_c[4:0]
address_wire[14:0]
data_write[31:0]
instruction_reg[31:0]
write_data_reg[31:0]
operand1[31:0]
operand2[31:0]
data_read[31:0]
data_read2[31:0]
read_data_reg1[31:0]
read_data_reg2[31:0]
ALU_result[31:0]
write_enable
write_enable_data
pc_change
opcode_clone_reg[2:0]
write_address[14:0]


Untitled 5*
Name
Value
clk
pc_q[15:0]
instruction_q[31:0]
state_q[1:0]
opcode_wire[2:0]
reg_a[4:0]
reg_b[4:0]
reg_c[4:0]
address_wire[14:0]
data_write[31:0]
instruction_reg[31:0]
write_data_reg[31:0]
operand1[31:0]
operand2[31:0]
data_read[31:0]
data_read2[31:0]
read_data_reg1[31:0]
read_data_reg2[31:0]
ALU_result[31:0]
write_enable
write_enable_data
pc_change
opcode_clone_reg[2:0]
write_address[14:0]
Untitled 5*
Name
Value
clk
pc_q[15:0]
instruction_q[31:0]
state_q[1:0]
opcode_wire[2:0]
reg_a[4:0]
reg_b[4:0]
reg_c[4:0]
address_wire[14:0]
data_write[31:0]
instruction_reg[31:0]
write_data_reg[31:0]
operand1[31:0]
operand2[31:0]
data_read[31:0]
data_read2[31:0]
read_data_reg1[31:0]
read_data_reg2[31:0]
ALU_result[31:0]
write_enable
write_enable_data
pc_change
opcode_clone_reg[2:0]
write_address[14:0]


Untitled 5*
Name
Value
clk
pc_q[15:0]
instruction_q[31:0]
state_q[1:0]
opcode_wire[2:0]
reg_a[4:0]
reg_b[4:0]
reg_c[4:0]
address_wire[14:0]
data_write[31:0]
instruction_reg[31:0]
write_data_reg[31:0]
operand1[31:0]
operand2[31:0]
data_read[31:0]
data_read2[31:0]
read_data_reg1[31:0]
read_data_reg2[31:0]
ALU_result[31:0]
write_enable
write_enable_data
pc_change
opcode_clone_reg[2:0]
write_address[14:0]


Untitled 5*
Name
Value
clk
pc_q[15:0]
instruction_q[31:0]
state_q[1:0]
opcode_wire[2:0]
reg_a[4:0]
reg_b[4:0]
reg_c[4:0]
address_wire[14:0]
data_write[31:0]
instruction_reg[31:0]
write_data_reg[31:0]
operand1[31:0]
operand2[31:0]
data_read[31:0]
data_read2[31:0]
read_data_reg1[31:0]
read_data_reg2[31:0]
ALU_result[31:0]
write_enable
write_enable_data
pc_change
opcode_clone_reg[2:0]
write_address[14:0]



The simulation reveals that the clock signal undergoes inversion every 10 ns, marking the initiation of a new CPU state with each positive edge occurring at the same interval. During simulation, the InstructionMemory source code was utilized to test all of the CPU instructions, which total up to 8. The results illustrate that with each state transition, a new stage is executed, facilitating the transfer of data to subsequent registers or modules for CPU operations.

In the initial state (state 0), the CPU retrieves instructions from the InstructionMemory file, proceeding to decode them in the subsequent stage. Decoded instructions are then sent to a register designated for holding instruction inputs, enabling the CPU to determine its next course

of action. Concurrently, data is fetched from the register file to ensure availability for ALU operations, if deemed required. Moving to state 2, the CPU engages in ALU operations by transmitting data from the register file to input registers of the ALU. Additionally, the ALU communicates changes in the program counter (PC) variable for branching instruction scenarios, facilitating adjustments to the PC counter as necessary.

The final stage of the CPU encompasses memory operations, wherein lw/sw instructions entail fetching or storing data between data memory and register addresses. Furthermore, this stage involves writing ALU results to destination registers within the register file or also modifying the PC counter based on the different execution outcomes.

Challenges

In the process of building a simplified multi-cycle CPU using Verilog, I experienced a couple of challenges alongside experience that gave me genuine insight into this key component.

Given that I am currently taking Senior Design 1, I am working daily on contributing to my team's prototype. Therefore, my biggest challenge was time. I was having a challenge to allocate time to both completing the project and understanding it. I overcame this challenge by reaching a good stopping point for Senior Design 1 and then clearing my headspace for a full concentration session on this project.

Another brief challenge that I encountered was within the Instruction Memory module where branching back was being executed until I dealt with this by ensuring the instructions account for this issue. In doing so, I was left trying to test out instructions such that they would branch and eventually end with all eight different instructions tested on the CPU simulation.

Additionally, designing a CPU with multiple Verilog files introduces specific challenges to the debugging process. The primary challenge that I personally encountered was coordinating and synchronizing the interactions between various modules responsible for different CPU functionalities, such as memory access. Errors in one module can (and most likely will) propagate throughout the CPU, affecting its overall operation in a negative way. In general, debugging becomes more intricate due to the distributed nature of the design across multiple files, making it harder to track the flow of data and control signals, especially in the context of this project.