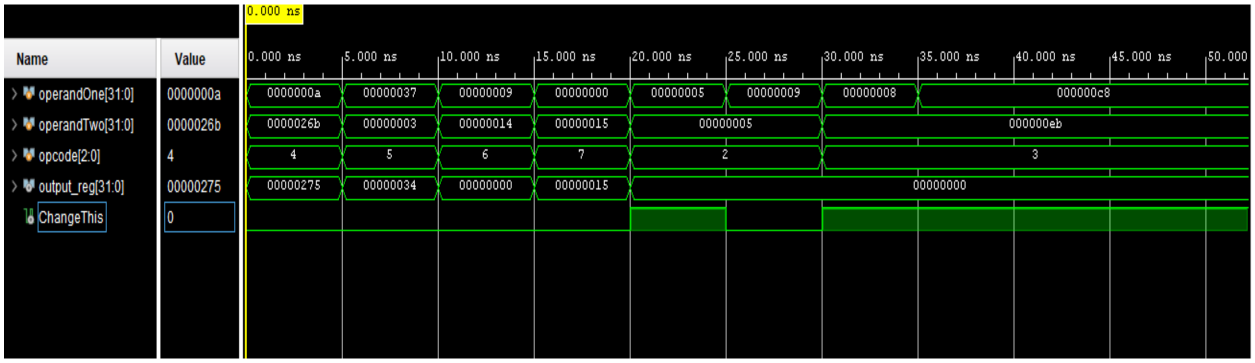
RAUL GRATEROL
4/18/24

EE4783: Final Project

ALU Waveform:

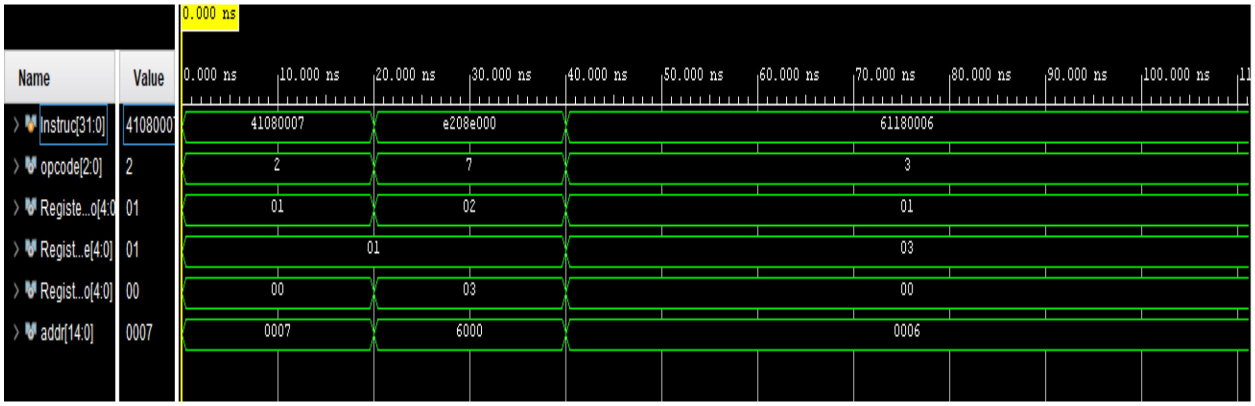


The ALU is a fundamental building block of the central processing unit (CPU) in a computer. This specific ALU module is designed to perform arithmetic operations such as addition, subtraction, and logical operations like AND and OR based on the opcode input. It takes two 32-bit inputs, ip_0 and ip_1, and an opcode. The output, op_0, is determined by the opcode. For instance, if the opcode is 4, the ALU performs the addition of ip_0 and ip_1. If the opcode is 5, it performs subtraction, and so on for the AND and OR operations with opcodes 6 and 7, respectively.

The change_pc output is another significant aspect of this ALU. It is used for branch operations and goes high under two conditions. First, when the opcode is 2 (which corresponds to the 'beq' instruction in assembly language) and ip_0 equals ip_1, Second, when the opcode is 3 (corresponding to the 'blt' instruction) and ip_0 is less than ip_1.

The second module in the code is the testbench for the ALU. This module, named ALUTB, instantiates the ALU module and provides it with test inputs for different operations. It changes the operands and the opcode over time to test the functionality of the ALU under different conditions. The results of the operations can be observed in the output_reg and ChangeThis wires. This testbench is crucial for verifying the correctness of the ALU design as it provides different operand values and opcodes to the ALU and observes the outputs. In this way, we can ensure that our ALU design is functioning as expected under various conditions.

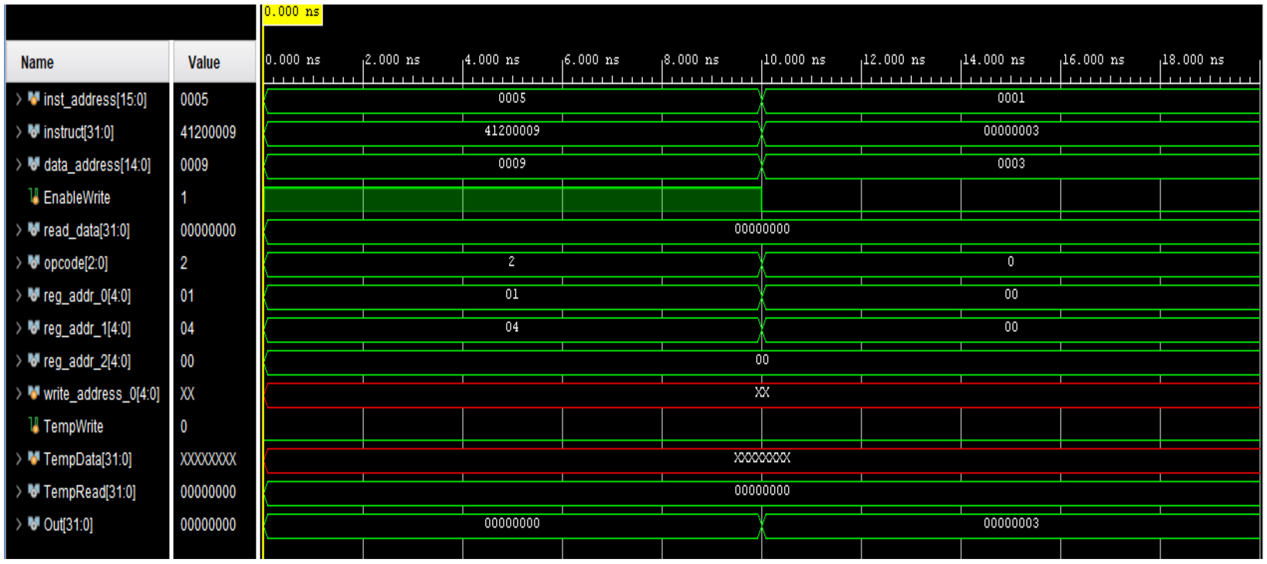
Decoder Waveform:



The Decoder is a crucial component in a computer's central processing unit (CPU). It takes a 32-bit instruction as input and drives each of the outputs based on the Instruction Set Architecture given in the project instructions. The Decoder module has five outputs: opcode, reg_addr_0, reg_addr_1, reg_addr_2, and addr. The opcode is assigned the three most significant bits of the instruction, which determine the operation to be performed. The addr output is assigned the 15 least significant bits of the instruction. The reg_addr_0, reg_addr_1, and reg_addr_2 outputs are assigned specific 5-bit sections of the instruction, which are used to address the registers involved in the operation.

The second module in the code is the testbench for the Decoder. This module, named DecoderTB, instantiates the Decoder module and provides it with test instructions. It changes the instruction over time to test the functionality of the Decoder under different conditions. The results of the decoding can be observed in the opcode, RegisterZero, RegisterOne, RegisterTwo, and addr wires. This testbench is essential for verifying the correctness of the Decoder design as it provides different instructions to the Decoder and observes the outputs. In this way, we can ensure that our Decoder design is functioning as expected under various conditions. In this specific testbench, three different instructions are tested. The first instruction is 32'h4108_0007, the second is 32'he208_E000, and the third is 32'h6118_0006. After each instruction is assigned to Instruc, there is a delay of 20-time units before the next instruction is assigned. This sequence of instructions and delays simulates a series of instructions being decoded by the Decoder over time.

Memory Waveform:

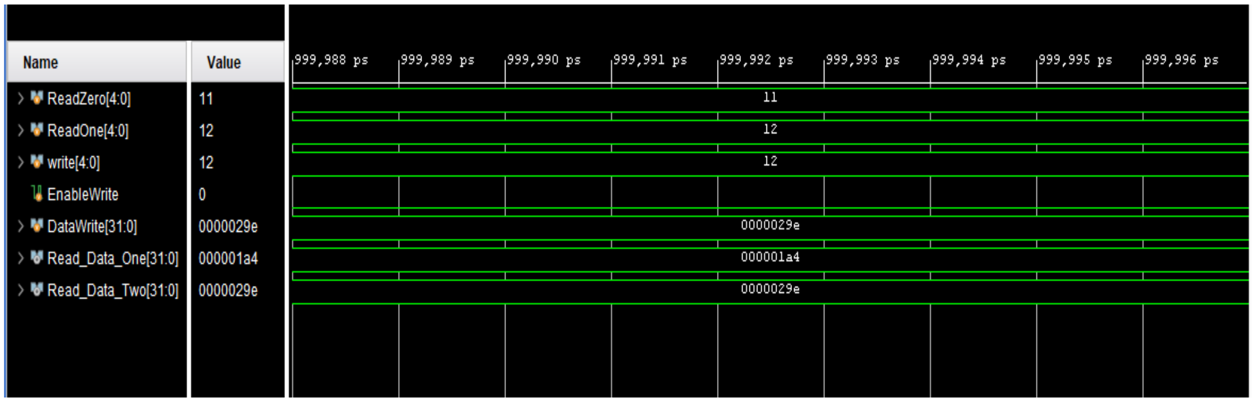

0.000 ns
Name
Value
0.000 ns
2.000 ns
4.000 ns
6.000 ns
8.000 ns
10.000 ns
12.000 ns
14.000 ns
16.000 ns
18.000 ns
inst_address[15:0] 0005 — 0005, 0001
instruct[31:0] 41200009 — 41200009, 00000003
data_address[14:0] 0009 — 0009, 0003
EnableWrite 1
read_data[31:0] 00000000 — 00000000
opcode[2:0] 2 — 2, 0
reg_addr_0[4:0] 01 — 01, 00
reg_addr_1[4:0] 04 — 04, 00
reg_addr_2[4:0] 00 — 00
write_address_0[4:0] XX — XX
TempWrite 0
TempData[31:0] XXXXXXXX — XXXXXXXX
TempRead[31:0] 00000000 — 00000000
Out[31:0] 00000000 — 00000000, 00000003

In this testbench, the InstructionMemory module named IMTest takes an instruction address (inst_address) as input and provides a 32-bit instruction (instruct) as output. The Decoder module named decoding takes this instruction as input and decodes it into an opcode and three register addresses (reg_addr_0, reg_addr_1, reg_addr_2), and a data address (data_address).

The RegisterFile module named RFing takes the register addresses and a write enable signal (TempWrite) as inputs. It also takes a 32-bit data input (TempData) for writing to a register. It provides two 32-bit data outputs (read_data and TempRead) corresponding to the data in the addressed registers.
The DataMemory module named DMTest takes a data address, a write enable signal (EnableWrite), and a 32-bit data input (read_data) for writing to the data memory. It provides a 32-bit data output (Out) corresponding to the data at the addressed location in the data memory. The initial begin...end block in the

testbench is where the simulation is defined. Inside this block, different values are assigned to the inst_address, EnableWrite, and TempWrite signals at different times, simulating the operation of the modules under different conditions. The #10 after the first assignment statement introduces a delay of 10-time units before the next statement is executed. This allows enough time for the modules to process each instruction and for the outputs to stabilize. The outputs of the modules can be observed to verify that they are functioning correctly. By comparing these outputs with the expected results for each instruction and each set of control signals, we can verify that the modules are functioning correctly.

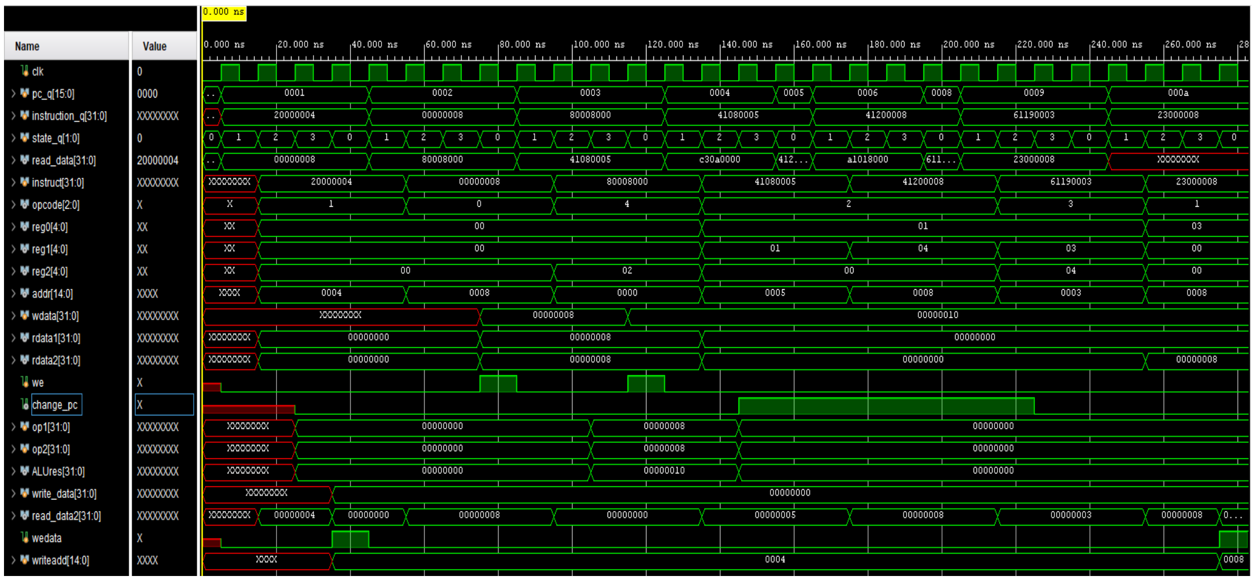
RegisterFile Waveform:

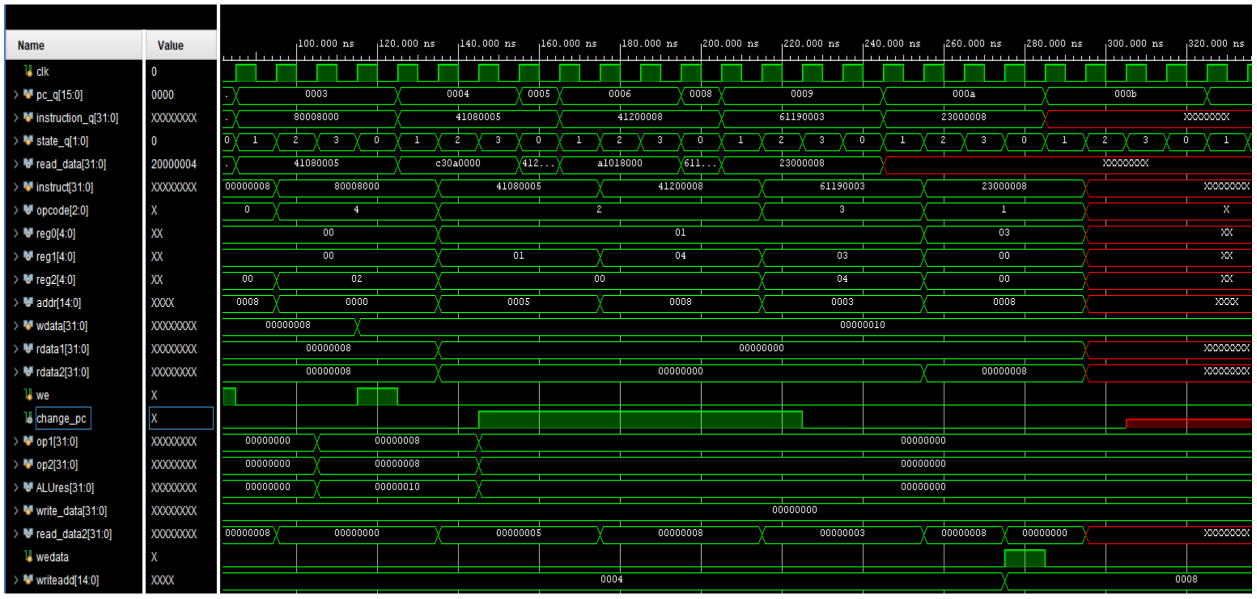


Name	Value
ReadZero[4:0]	11
ReadOne[4:0]	12
write[4:0]	12
EnableWrite	0
DataWrite[31:0]	0000029e
Read_Data_One[31:0]	000001a4
Read_Data_Two[31:0]	0000029e

The RegisterFile module is essentially a 2-dimensional register array with two read ports, and one write port. It takes two 5-bit read addresses (read_address_0 and read_address_1), a 5-bit write address (write_address_0), a write enable signal (write_en), and a 32-bit write data input (write_data). The module provides two 32-bit read data outputs (read_data_0 and read_data_1). The module contains a 256-element array of 32-bit registers (ram), which is initialized to zero in the initial begin...end block. The read_data_0 and read_data_1 output are assigned the contents of the registers at the addresses specified by read_address_0 and read_address_1, respectively. If the write_en signal is high, the write_data is written to the register at the address specified by write_address_0. The RegisterTB module is a testbench for the RegisterFile module. It instantiates the RegisterFile module and provides it with test inputs for different operations. The initial begin...end block in the testbench defines a sequence of operations to be performed on the RegisterFile. The ReadZero, ReadOne, write, EnableWrite, and DataWrite signals are assigned different values at different times, simulating the operation of the RegisterFile under different conditions. The #10 after each assignment statement introduces a delay of 10-time units before the next statement is executed. This allows enough time for the RegisterFile to process each operation and for the outputs to stabilize. The outputs of the RegisterFile (Read_Data_One and Read_Data_Two) can be observed to verify that it is functioning correctly. By comparing these outputs with the expected results for each set of inputs, we can verify that the RegisterFile is functioning correctly.

CPU Waveform:





The simulation shows a pattern in the clock signal, with a flip every 5 nanoseconds and a new positive edge every 10 nanoseconds, which marks the start of a new CPU state. Looking into the InstructionMemory Verilog source file, a set of instructions were added to test all 8 instructions on the CPU.

Each state in the simulation shows a step-by-step stage, where data moves to the next registers or modules, helping the CPU to do its operations smoothly. In the first state, the CPU starts by getting instructions from the InstructionMemory file. Then, the CPU starts to decode the instructions, moving them into a register that holds inputs for the decoder module. This helps with decoding the instruction and figuring out what the CPU needs to do next. At the same time, data is pulled from the register file, ready to be used for ALU operations if needed by the instructions.

Moving to the next state in stage 2, the CPU does ALU operations, moving the results of register file data into registers that are inputs for the ALU. Along with these inputs, the ALU also sends out a variable to change the program counter in cases where branching instructions are involved, making sure the PC counter matches with the address of the instruction.

The last stage of the CPU includes the memory stage, where instructions like lw/sw start to fetch or store data between the data memory and register addresses. Also, this stage looks after writing the results from the ALU stage into the destination registers of the register file or changing the PC counter to adjust for any changes from the execution stage.

Challenges:

While working on this project of implementing a Multi-Cycle Processor in Verilog, I encountered several challenges. Here's how I tackled them:

1. Understanding the Instruction Set Architecture (ISA): The ISA was a bit complex with different types of instructions such as memory, control, and arithmetic instructions. I spent time studying the ISA, understanding how each instruction works, and how they are represented in binary form. This helped me to correctly implement the Decoder module which is responsible for decoding the instructions.
2. Designing the CPU Components: Designing the CPU components like Memory, Register File, Decoder, and ALU was another challenge. Each component had its own functionality, and it was crucial to get them right for the CPU to work correctly. I started by writing a high-level design of each component, detailing what inputs it takes, what outputs it produces, and how it processes the inputs to produce the outputs. This helped me to have a clear picture of each component before starting the implementation.
3. Implementing the CPU Execution Stages: Implementing the CPU execution stages was a bit tricky as it involved coordinating between different CPU components. I had to ensure that the data flows smoothly from one stage to the next and that the CPU state is updated correctly after each stage. To overcome this challenge, I used a state machine to control the flow of execution. This allowed me to clearly define what happens in each state and how the CPU transitions from one state to the next.
4. Testing the CPU: Testing the CPU was another major challenge. It was important to test not only each CPU component individually but also the entire CPU as a whole. I wrote a testbench for each CPU component and another testbench for the entire CPU. I made sure to test all possible instructions and edge cases to ensure that the CPU works correctly under all conditions.
5. Debugging: Debugging the CPU was quite challenging due to the complexity of the design. To make debugging easier, I used a systematic approach where I first tested each CPU component individually before testing the entire CPU. This allowed me to isolate any issues to a specific component and made it easier to find and fix bugs.

By tackling these challenges head-on and persisting through the difficulties, I was able to successfully implement the Multi-Cycle Processor in Verilog. This project not only helped me to improve my Verilog skills but also gave me a deeper understanding of how a CPU works.

In conclusion, the project of implementing a Multi-Cycle Processor in Verilog was a challenging yet rewarding experience. It provided a deep understanding of the inner workings of a CPU and the role of each component in executing instructions. Despite the challenges faced during the design, implementation, and testing phases, the systematic approach adopted for this project ensured each component and the CPU as a whole functioned as expected. This project not only honed my Verilog skills but also provided valuable insights into the practical aspects of CPU design and operation. The successful completion of this project has bolstered my confidence in taking on more complex design and implementation tasks in the future. It was a significant step forward in my journey of exploring and understanding computer architecture.