Simple Processor in VHDL

Final Project
ECE 443

Authors:
Brice Zimmerman
UIN: 01172015

Emily Ly
UIN: 01190089

Old Dominion University
Department of Electrical and Computer Engineering
Project Due: December 4, 2023

Honor Code:
"I [we] pledge to support the Honor System of Old Dominion University. I [we] will refrain from any form of academic dishonesty or deception, such as cheating or plagiarism. I [we] are aware that as a member of the academic community it is my responsibility to turn in all suspected violations of the Honor Code. I [we] will report to a hearing if summoned."

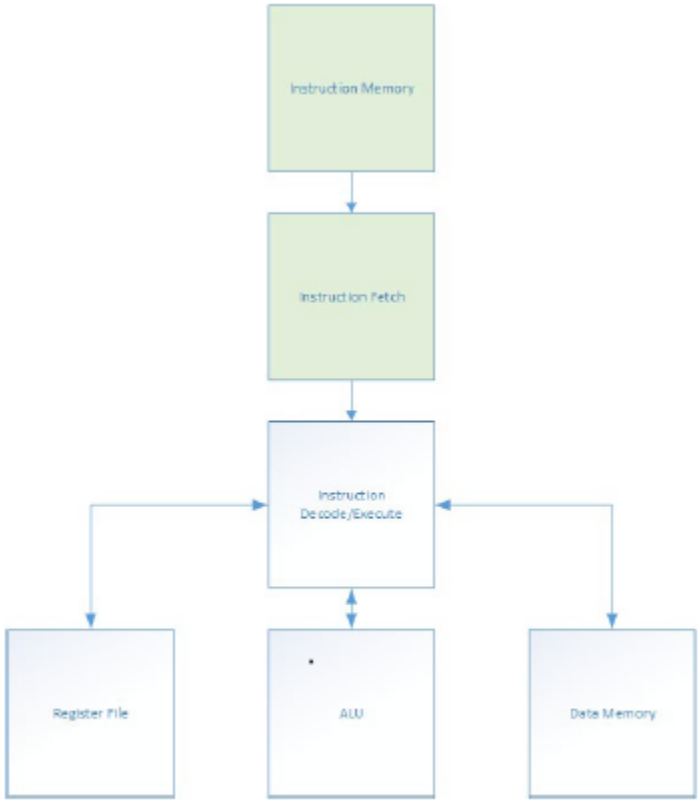
Table of Contents

1. Introduction

The task for this assignment was to create a simple processor that implements the previously designed Arithmetic Logic Unit (ALU) using VHSIC Hardware Description Language (VHDL) in the Aldec environment. The simple processor must support different instructions to include signed addition, signed multiplication, passthrough A and B, signed subtraction, load immediate, and store and load halfwords. It must also be able to use register (R-type) and immediate (I-Type) formats, support 256 addresses by 16 bits, and have eight registers. The processor must then be able to properly execute a set of instructions that are compiled into hexadecimal.

2. Background

Creating a simple processor involves understanding ALU operations, instruction sets, memory structure, and the construction of processor components. A high level overview of the processor is shown below.



High level overview of the processor

To explain each part separately, the ALU is the heart of the processor. It executes arithmetic operations such as addition, multiplication, and subtraction as well as logical operations like AND and OR.

The instruction sets encompass encoding operations into binary form, zeros and ones, and operation codes (OpCodes) which dictate each operation's specific code. There are two

formats, R-Type and I-Type instructions where R-type performs operations specified in A and B portions and stored in C portions from the ALU. I-Type instructions specify immediate values and will use ldi to load immediate instructions specified by D and also will store and load halfwords into the memory address. Formats for R-Type and I-Type are shown below.

R-Type

unused	opcode	c	a	b
1 bit	3 bits	4 bits	4 bits	4 bits

I-Type

unused	opcode	d	value
1 bit	3 bits	4 bits	8 bits

A summary chart for the different instruction opcodes is below.

S0	S1	S2	Instruction Format	Operation	Mnemonic
0	0	0	R	Signed Addition	add
0	0	1	R	Signed Multiplication	mult
0	1	0	R	Passthrough A	pa
0	1	1	R	Passthrough B	pb
1	0	0	R	Signed Subtraction	sub
1	0	1	I	Load Immediate	ldi
1	1	0	I	Store halfword	sh
1	1	1	I	Load halfword	lh

Next in the processor components, the memory structure defines how data is stored and accessed. It defines addressing mechanisms, word sizes (16 bits in our case), and the register structure which will be eight registers each 16 bits wide.

Constructing the processor components involves implementing distinct VHDL workspaces for the ALU, instruction decoder and executer, register file, data memory interface, and control unit. Each of these components is instrumental in executing instructions accurately and efficiently.

Finally, as a processor is used to process information, a specific program must be executed. This program is as follows and should output the below hexadecimal values.

```
ldi $r0, 10
ldi $r1, 5
ldi $r2, 0
ldi $r3, 0
ldi $r4, 0
ldi $r5, 0
ldi $r6, 0
ldi $r7, 0
add $r2, $r0, $r1
mult $r3, $r0, $r1
sub $r4, $r0, $r1
sh $r3, 0x0B
sh $r4, 0x0A
lh $r6, 0x0A
lh $r7, 0x0B
```

The following is the program compiled into hexadecimal.

```
500A
5105
5200
5300
5400
5500
5600
5700
0201
1301
4401
630B
640A
760A
770B
```

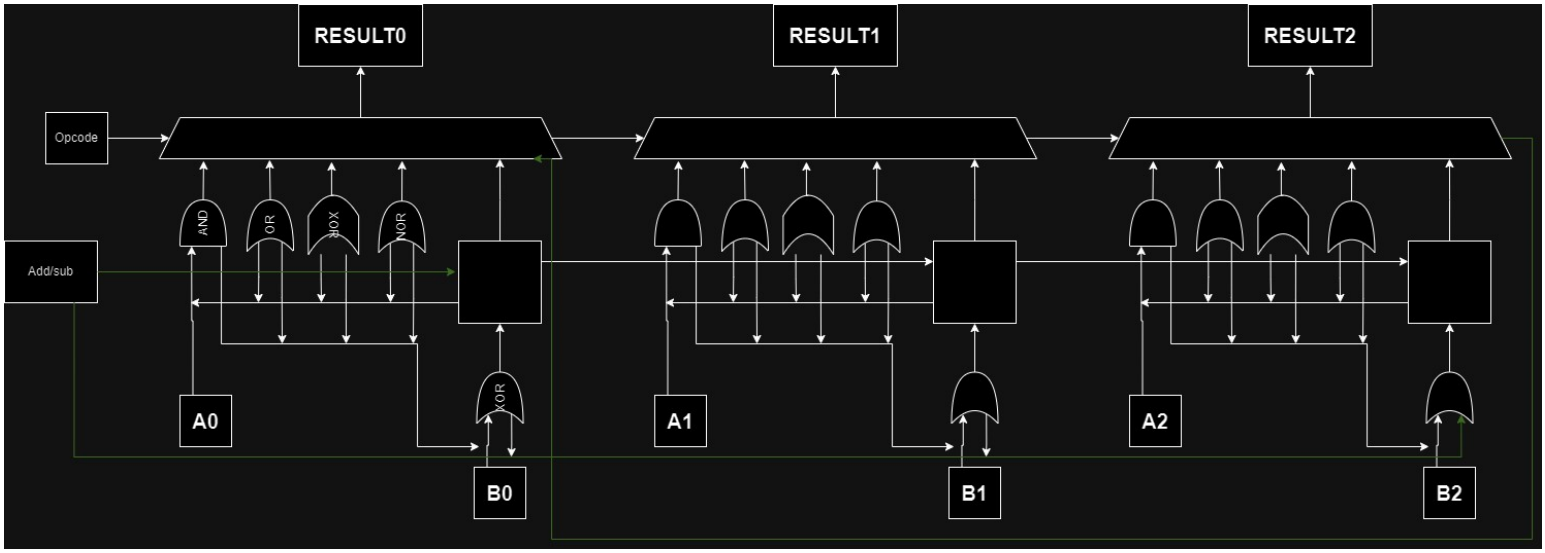
3. Preliminary

Workspaces were created separately to ensure proper working order before combining them all into the processor. Each will be described below.

3.1. ALU

An arithmetic logic unit (ALU) is a fundamental component of many digital logic devices. The ALU we implemented was a combination of a half adder, full adder, 16-bit adder, 16-bit subtractor, 16-bit multiplier, and 16-bit multiplexer. It took two 16-bit arguments and produced a 16-bit output that were all signed. The

instructions were selected using three signal lines and outputted a three bit status output. The overall logic is shown in the diagram below.



Specifics of each component of the ALU are described below.

3.1.1 Half Adder

A half adder in VHDL is a digital logic component that takes two input bits and outputs a 1-bit sum and the carryout. A half adder is called "half" because it can only handle one bit at a time and doesn't consider any carry from previous additions. A half adder is a building block and was used to help create the full adder. The VHDL code defines three entities for basic logic gates, AND gate, XOR gate, and a Half Adder. For XOR if the bits are the same, the result is 0. If the bits are different, the result is 1. For AND if the bits are the same, the result is 1. If the bits are different, the result is 0. The and_gate entity represents a simple AND gate, the xor_gate entity represents an XOR gate, and the half adder entity uses the previously defined AND and XOR components. The components are instantiated and connected to the proper signals.

3.1.2 Full Adder

The full adder adds three 1-bit inputs (A, B, Cin) and outputs two 1-bit sums (Sum, Cout). The full adder we implemented avoids needing a separate component declaration by directly instantiating the half adder entity that was defined in the external half_adder component. It connects the carry outs with an OR gate and maps the final Sum and Cout ports.

3.1.3 16-bit Adder

A 16-bit adder takes two 16-bit binary inputs, adds them together, and then outputs a 16-bit binary result. It works by putting together multiple full adders to handle one bit of the input numbers. The full adders cascade to accommodate the sixteen bits. This means that the carry-out from each full adder is passed as the

carry-in to the next one, allowing for the propagation of carry through all the bits. The outputs of each full adder produce the 16-bit sum, while the final carry-out represents any overflow from the addition, indicating whether the result exceeds the 16-bit capacity. The 16-bit multiplexer we implemented takes a collection of full adders for each bit to make it a parallel binary adder.

3.1.4 16-bit Subtractor

A 16-bit Subtractor calculates the difference between two input numbers and borrows a signal to indicate if a borrow had occurred during the subtraction. In our implementation of the 16-bit Subtractor, the architecture begins by generating a bitwise complement of input B creating the signal "not_B_sig". It iterates through each bit of B and the NOT operator. The code instantiates two 16-bit adder components where the first instance adds A to the complement of B, and the second adds A to the result of the first. The results are then stored. The borrow signal is determined by examining the most significant bit (MSB) of the previous result from the adder out signal.

3.1.5 16-bit Multiplier

A 16-bit Multiplier multiplies two 16-bit binary numbers to produce a 32-bit result. We implemented this operation by using "Result <= A*B;" since the program can directly multiply the two and then store it in a temp signal. We also implemented overflow detection if the result cannot be represented within 32 bits. Furthermore, the 16-bit multiplier was written using the behavioral model.

3.1.6 16-bit Multiplexer

A 16-bit multiplexer is designed to select one out of sixteen input data lines and route it to the output based on a set of control signals. The output of the multiplexer will be the data value from the selected input. The 16-bit width indicates that each data input and the output can handle 16 individual binary digits, or bits. In our implementation of the multiplexer, it has eight 16-bit input ports from A0 to A7 and a 3-bit control signal to determine which input is routed to the 16-bit output. The 16-bit multiplexer architecture uses a case statement to select the input value. Based on the case statement, such as "001", the multiplexer performs addition. Other operation selection cases are "001" multiplication, "101" passthrough A, "011" passthrough B, and "100" subtraction. The other three cases are undefined. The last case statement is "when other" to ensure that if there's an option that doesn't have a selection, the result will be zeros.

3.2 Register File

The register file holds the processor's registers and performs read and write operations. This is implemented as an 8 register file with 16-bit widths. It has inputs for the clock, reset, register write enable, write destination register, and write data. It reads from two specific registers, and the outputs are read from those two registers. The architecture uses a process sensitive to clock and reset to implement the register storage and writing. On reset, all registers are cleared to 0.

On the rising clock edge, if the write enable is high, the write data is written to the register specified by the write destination.

3.3 Data Memory

The data memory stores data that is used for load and store instructions. This is implemented using an internal array of 256, 16-bit values to implement RAM storage. It has inputs for clock, memory address, write data, write enable, and read enable. The output is the read data. On clock edges, if write enable is high, it writes the input write data to the RAM location specified by the address bus. For reads, it outputs the data at the given address when read enable is high, otherwise it outputs all 0s.

3.4 Instruction Fetch

The instruction fetch fetches instructions. It works using a program counter register that increments by 2 each clock cycle to step through instruction memory addresses. It then uses combinational logic to generate the next program counter (PC) value. It then instantiates instruction memory to fetch the instruction based on the current PC. This modularizes the first stage of fetch and frees up the processor to focus on the decode and execute states. Registered outputs to external logic are driven on the rising clock edge.

3.5 Control Unit

Generates the necessary control signals that control the execution of instructions. It interprets the opcode of an instruction and produces outputs that regulate components in the processor.

3.6 Instruction Memory

The instruction memory stores all the instructions. It provides the instructions based on the program counter (PC). The PC keeps track of the address of the next instruction.

3.7 Instruction Decode Execute

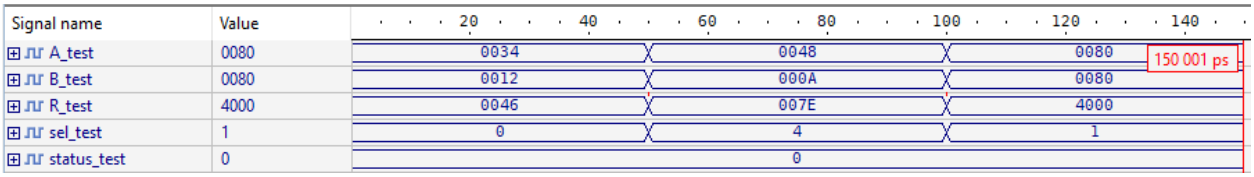
The instruction decode execute component uses the instruction memory, control unit, register file, ALU, data memory, and PC. The PC fetches the instruction address from the instruction memory. The instruction is then retrieved. Next the control unit decodes the instruction opcode that generates control signals based upon instruction type. Registers are read based on the instruction's register address. The ALU operations and memory operations are executed based on the instruction type and control signals. Data is accessed from or written to the data memory for load and store instructions. Finally, the results are written back to the registers and updated.

4. Project

4.1. ALU

The testbench simulates three operations (addition, subtraction, and multiplication) on the ALU and observes the resulting outputs and status signals. Test cases are executed sequentially, and after each operation, the testbench waits for 50ns before moving to the next test case. The report statements within the monitor process display the calculated result (R_test) and the ALU status (status_test) after a certain simulation time. Test case 1 added 52 (A_test) and 18 (B_test) using selection “000” (sel_test) to get the result of 70 (R_test). This is correctly shown in the waveform below in hex values in the first 50ns. The waveform from 50ns to 100ns, tests case 2 which used the subtractor to subtract 10 from 72. Finally from 100 to 150ns tests case 3 which selected the multiplier to multiply 80 and 80. This is shown in the figures below.

```
52      -- Test case 1: A + B
53      A_test <= "0000000000110100"; -- 52
54      B_test <= "0000000000010010"; -- 18
55      sel_test <= "000"; -- Select adder
56      wait for 50 ns;
57
58      -- Test case 2: A - B
59      A_test <= "0000000001001000"; -- 72
60      B_test <= "0000000000001010"; -- 10
61      sel_test <= "100"; -- Select subtractor
62      wait for 50 ns;
63
64      -- Test case 3: A * B
65      A_test <= "0000000010000000"; -- 80
66      B_test <= "0000000010000000"; -- 80
67      sel_test <= "001"; -- Select multiplier
68      -- Result = 0x4000
69      wait for 50 ns;
```



Test cases for each component of the ALU to include the half adder, full adder, 16-bit adder, 16-bit subtractor, 16-bit multiplier, and 16-bit multiplexer are shown in the appendix.

4.2 Register File

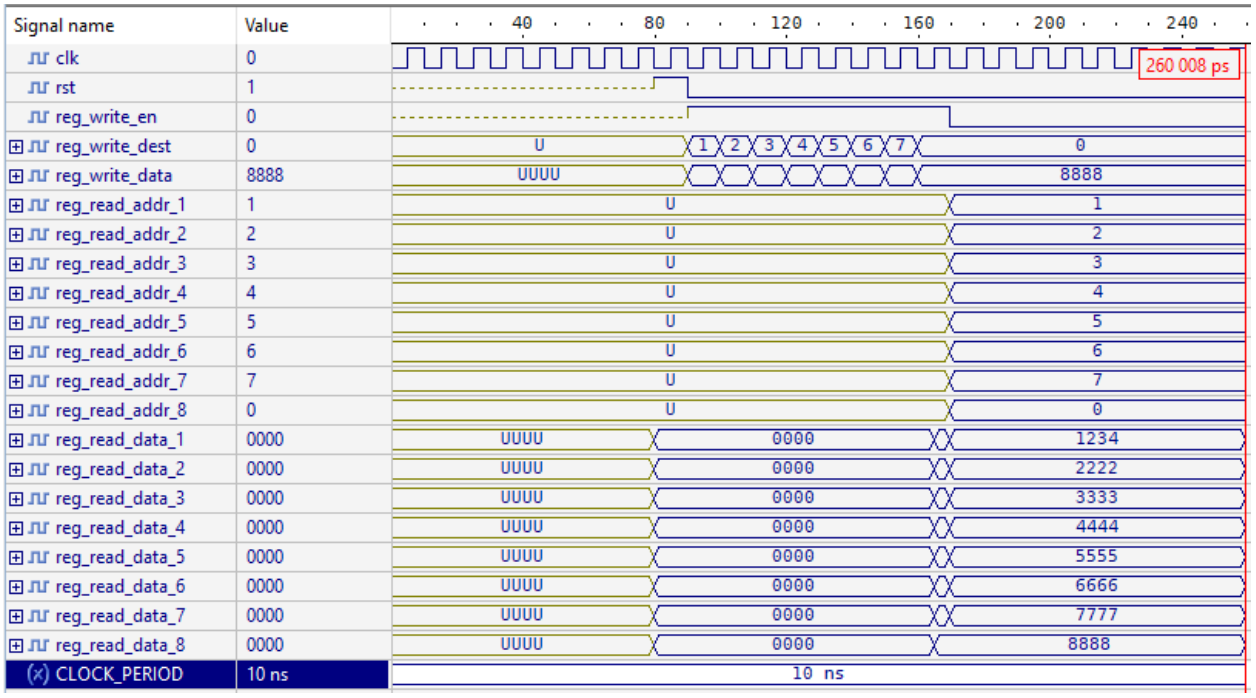
The testbench first declares the component with all of its ports, including the eight read address and read data registers. It then declares signals to connect to each of the in/out ports of the registerfile for stimulus and monitoring. An instance of the registerfile component is mapped to these signals under the unit under test "UUT".

The testbench includes a clock generation process that provides a 10ns period clock for 1200ns total to provide enough cycles to fully test writing then reading all the registers. The stimulus process first resets the registers then begins writing unique 16-bit values to each of the 8 registers sequentially, toggling the write enable for each. It then reads back from each register, storing the output into the corresponding read data signals. Assertions are included after the read to check that each register retained the unique value that was written to it, ensuring no unintended overwriting occurred between registers. If any register contains incorrect data, an error is reported. A portion of the testbench is shown below.

The subsequent waveform in the simulation would exhibit a clock signal alternating between '0' and '1' with a period of 10 ns. A reset signal (rst) briefly goes high ('1') before becoming low ('0'). Following the reset, a write operation occurs where reg_write_en becomes '1', writing x"1234" to register 1 ("001"). Subsequently, read operations are initiated for registers 1 and 2 ("001" and "010" respectively). This iterates through all 8 registers with x“2222” written to register 2 (“010”), x “3333” written to register 3 (“011), and so on. During the simulation, an assertion checks if the data read from register 1 (reg_read_data_1) matches the expected value (x"1234"), ensuring the correctness of the register file's read functionality. No assertions were triggered to indicate that observed read back data was as is expected. This is shown in the figures below.

```
process
begin
    wait for 80 ns; -- Allow for initializations

    -- Test Write Operation
    rst <= '1';
    wait for CLOCK_PERIOD;
    rst <= '0';

    reg_write_en <= '1';
    reg_write_dest <= "001";    -- Write to register 1
    reg_write_data <= x"1234";  -- Data to be written
    wait for CLOCK_PERIOD;
    reg_write_en <= '0';

    reg_write_en <= '1';
    reg_write_dest <= "010"; -- Write to register 2
    reg_write_data <= x"2222";
    wait for CLOCK_PERIOD;
    reg_write_en <= '0';
```

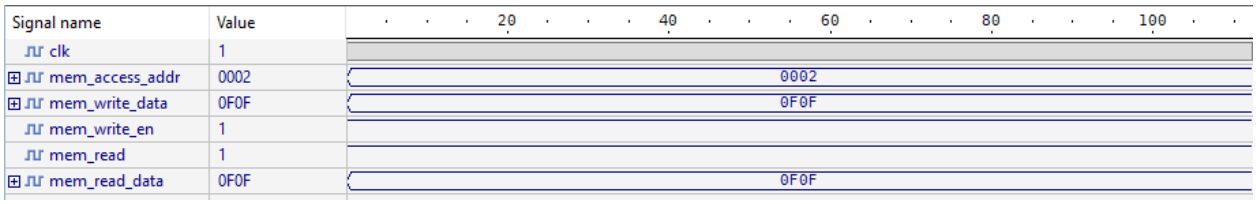
```
-- Test Read Operation
reg_read_addr_1 <= "001";   -- Read from register 1
reg_read_addr_2 <= "010";   -- Read from register 2
reg_read_addr_3 <= "011";   -- Read from register 3
reg_read_addr_4 <= "100";   -- Read from register 4
reg_read_addr_5 <= "101";   -- Read from register 5
reg_read_addr_6 <= "110";   -- Read from register 6
reg_read_addr_7 <= "111";   -- Read from register 7
reg_read_addr_8 <= "000";   -- Read from register 8

wait for CLOCK_PERIOD;

-- Check the read data
assert reg_read_data_1 = x"1234" report "Error: Incorrect
assert reg_read_data_2 = x"2222" report "Error: Incorrect
assert reg_read_data_3 = x"3333" report "Error: Incorrect
assert reg_read_data_4 = x"4444" report "Error: Incorrect
assert reg_read_data_5 = x"5555" report "Error: Incorrect
assert reg_read_data_6 = x"6666" report "Error: Incorrect
assert reg_read_data_7 = x"7777" report "Error: Incorrect
assert reg_read_data_8 = x"8888" report "Error: Incorrect
```



4.3 Data Memory

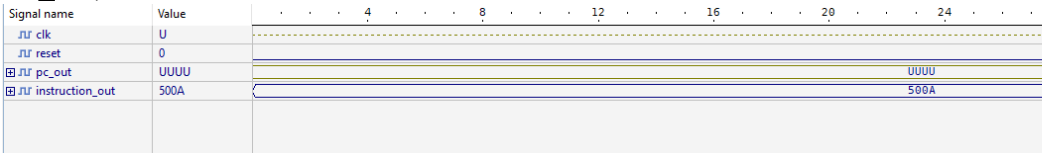
The testbench instantiates the RAM module and includes two processes: one generates a clock signal with a 10 ns period, while the other initiates memory operations such as writing and reading. It first writes a random value to address 1, reads it back to verify, writes 0x0F0F (3855 in decimal) to address 2, and reads that. This is shown in the figures below.

```
 -- Testing the read and write
process
begin
    -- initializing the memory address 1 with random data
    mem_write_en <= '1';                    -- enabling writing
    mem_access_addr <= "0000000000000001"; -- selecting address 1
    mem_write_data <= "1111000011110000"; -- writing data to address 1 (61680 in decimal)
    wait for 10 ns;

    -- reading from memory address 1
    mem_read <= '1';                        -- enabling reading
    mem_access_addr <= "0000000000000001"; -- reading from address 1
    wait for 10 ns;

    -- writing to address 2
    mem_read <= '0';                       -- disabiling reading
    mem_write_en <= '1';                   -- enabling writing
    mem_access_addr <= "0000000000000010"; -- selecting address 2
    mem_write_data <= "0000111100001111"; -- writing random data to address 2 (3855 in decimal)
    wait for 10 ns;

    -- reading from address 2
    mem_read <= '1';                        -- enabling reading
    mem_access_addr <= "0000000000000010"; -- reading from address 2
    wait for 10 ns;
```



4.4 Instruction Fetch

The instruction fetch testbench is designed to verify the functionality of the instruction fetch entity. The testbench instantiates the instruction fetch entity and provides a clk and reset stimulus. The clock process generates a clock signal with a period of 10 ns and the reset process resets the signal after 10 ns each time. The observed signals (pc_out and instruction_out) are monitored for correctness and simulated in a wave file.

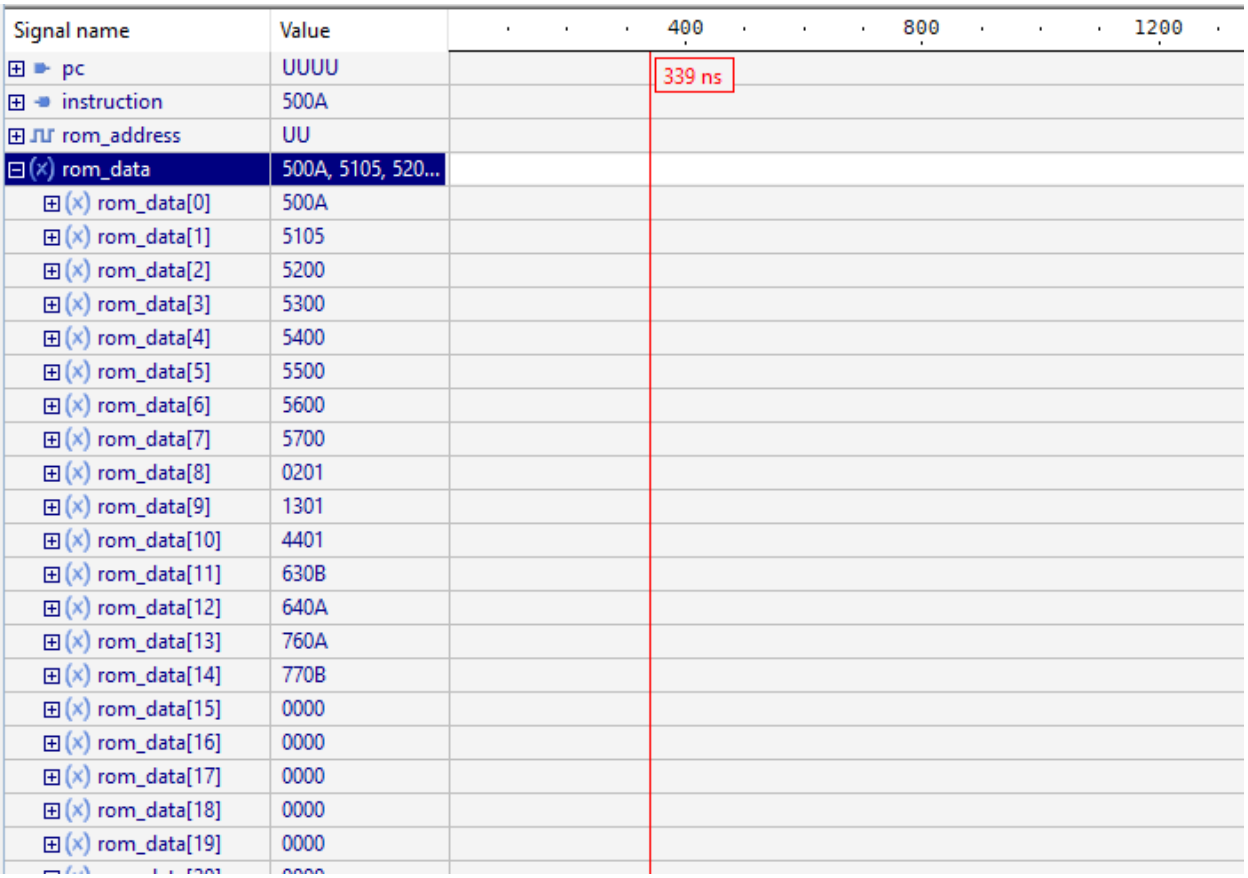


4.5 Control Unit

No separate testbench was created.

4.6 Instruction Memory

No testbench was created for the instruction memory, however, we can still simulate the loading of the memory into the memory array. Below you can see the simulated results where all of the instructions have been loaded into memory and are ready to be sifted through. You can see the instruction signal is also set on the first instruction correctly, “500A”.



4.7 Instruction Decode Execute

A test bench for the instruction decode and execute was created. Because this was the script that tied the whole processor together we unfortunately don't have any significant simulation results, but a testbench was created in the case that we finished designing the processor and fixing the bugs.

To verify the functionality of the Processor entity the testbench initialized a clock, reset, alu_result, and pc_out. A memory array contained the set of hex instructions for the processor to execute. The clock process generates a periodic

clock signal with a specified period.

```
-- Memory to hold instructions
type memory_array is array(0 to 15) of std_logic_vector(15 downto 0);
signal memory: memory_array := (
    X"500A", -- ldi #r0, 10
    X"5105", -- ldi #r1, 5
    X"5200", -- ldi $r2, 0
    X"5300", -- ldi #r3, 0
    X"5400", -- ldi $r4, 0
    X"5500", -- ldi $r5, 0
    X"5600", -- ldi $r6, 0
    X"5700", -- ldi $r7, 0
    X"0201", -- add $r2, $r0, $r1
    X"1301", -- mult $r3, $r0, $r1
    X"4401", -- sub $r4, $r0, $r1
    X"630B", -- sh $r3, 0x0B
    X"640A", -- sh $r4, 0x0A
    X"760A", -- lh $r6, 0x0A
    X"770B"  -- lh $r7, 0x0B
);
```

The testbench has a monitor process that simulates the execution of instructions stored in the memory. For each instruction the behavior is simulated, and the result is compared to the expected result based on the predefined instructions that were given to us. Checks are made for any mismatches between the results and the expected results.

5. Results

The processor program all compiles. It should take all the components together and feed them through the processor to implement a specific set of instructions. However, the testbench was unable to be correctly implemented. The logic of the testbench is designed to simulate a processor's behavior by executing a sequence of instructions stored in the memory array and validating the resulting output (alu_result). The testbench starts by initializing signals such as the clock (clk), reset (reset), alu_result, and pc_out. It then instantiates the Processor entity, mapping its ports accordingly.

The testbench operates through several processes: a clock generation process, a reset application process, and a monitor process. The clock process generates a clock signal with a specific period, while the reset process initializes and deserts the reset signal after a predefined duration.

The monitor process is the core of this testbench. It waits for the reset phase to complete and then iterates through the instructions in the memory array. For each instruction, it waits for a clock edge, updates the program counter (pc_out), simulates the processor's behavior based on the instruction, and compares the resulting alu_result with the expected value from the memory array. If any mismatches occur, it raises an assertion error, indicating a discrepancy between the expected and actual output.

```
-- Monitor process to verify output after each instruction
monitor_process: process
begin
    wait for 2 * clk_period; -- Wait for reset and initializations
    for i in memory'range loop
        wait until rising_edge(clk);
        pc_out <= std_logic_vector(to_signed(i, pc_out'length));
        wait for clk_period; -- Allow one clock cycle for execution

        -- Simulate the processor's behavior for each instruction
        case i is
            when 0 =>
                -- Execute ldi #r0, 10
                alu_result <= to_signed(10, alu_result'length); -- Assuming '10' as the immediate value
            when 1 =>
                -- Execute ldi #r1, 5
                alu_result <= to_signed(5, alu_result'length); -- Assuming '5' as the immediate value
                -- Add cases for other instructions
            when others =>
        null;
end case;

-- Verify the result
assert alu_result = signed(to_unsigned(conv_integer(memory(i)), alu_result'length))
    report "Mismatch at instruction " & integer'image(i)
    severity error;
    end loop;
    wait;
end process;
```

6. Conclusion

In conclusion, our project involved implementing a simple processor. We were able to successfully design a half adder, full adder, 16-bit adder, 16-bit subtractor, 16-bit multiplier and 16-bit multiplexer and incorporate that into an ALU.

In regards to the ALU implementation, it would take 50ns each to add 1000 to 2000 and also multiply 128 by 128. This is due to the 50ns wait instruction between each operation. This was added to ensure that each instruction had enough time to process before moving to the next instruction. To calculate the operations per second, it would take a total of 100ns to run both the addition and multiplication operations. To determine the operations per second under these conditions, we utilize the formula: Operations per second equals the reciprocal of the total time for the two operations multiplied by the number of operations achievable in one second. By plugging in the values, the calculation yields approximately 20 million operations per second. Thus, with this time constraint for the two distinct operations, the ALU demonstrates an approximate capability of executing 20 million operations within a single second. We could optimize the ALU by pipelining the operations instead of operating sequentially as we have in our code. Currently it waits 50ns before initiating the next operation. If we used pipelining, the ALU could start the next operation before the previous one finishes. For example, the addition operation could start in parallel with the multiplication operation. It would reduce the overall time to perform all operations.

Moreover, we were also able to successfully design the Register, Data Memory, and Instruction Fetch, and Instruction Memory components for the processor. These components were thoroughly tested in separate workspaces to confirm that they properly worked.

The testing of each component, as demonstrated in the test benches, displayed the correctness of our designs and implementations. The ALU, Register, Data Memory, Instruction Fetch, and Instruction Memory all performed as expected.

While a specific testbench for the processor with a timing diagram couldn't be provided as the simple processor was not functioning well enough for us to complete those tests. The performance of our simple processor is promising as various tests displayed accurate results of the components that we designed. We should have created separate workspaces to test the components the control unit. Furthermore we could have split the instruction "decode execute" into decode, execute, and processor instead of combining all three into one. This would have assisted in streamlining the code and debugging purposes.

Overall this project has been a valuable learning experience in digital circuit design and VHDL programming. It took creating an ALU a step further and realized the process into going into a processor. With further optimization, our processor design would have the potential to become more efficient and capable.

Appendices

Source code has been zipped into a folder and a README file has been included in the folder.

Each of the components for the ALU were previously tested and are shown below:

A. Half Adder

A test bench was created and verified that our half adder was working correctly. Below are the half adder test cases.

In the first test case, A was set to 0, B was set to 0 and therefore the Sum and Cout should also be 0. If you look at figure 1, the first 10ns are all at 0.

In the second test case, A was set to 1, B was set to 0, and therefore the Sum should be 1 and the Cout should be 0 as shown below. Keep in mind that Sum was mapped to the XOR gate, and Cout was mapped to the AND gate.

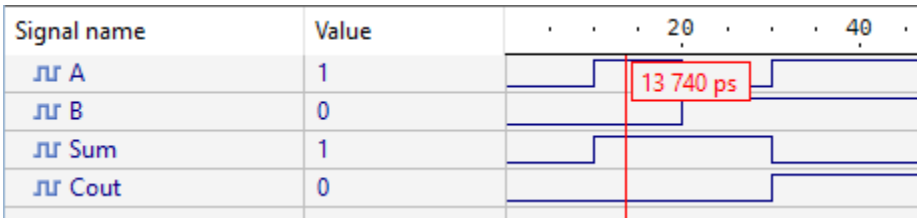


Figure 1: Half Adder test bench test case 2

In the third test case, A was set to 1, B was set to 0, and therefore the Sum should be 1 and the Cout should be 0 as shown below.

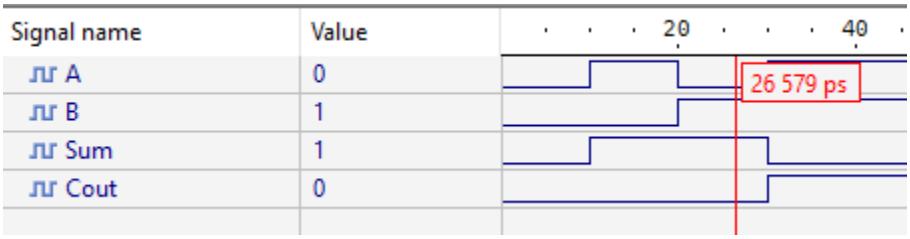


Figure 2: Half Adder test bench test case 3

In the fourth test case, A and B were set to 1, and therefore the Sum should be 0 and the Cout should be 1 as shown below

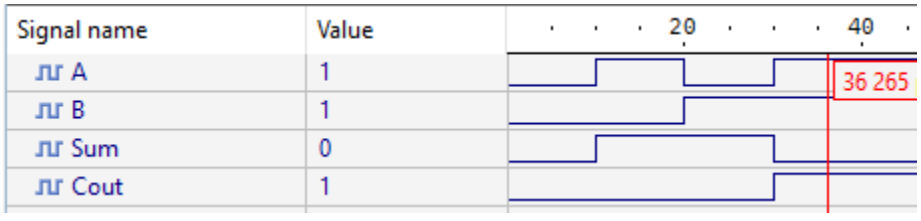


Figure 3: Half Adder test bench test case 4

B. Full Adder

A test bench was created and verified that our half adder was working correctly. Eight different cases were tested. For the first case, A, C, and Cin were all set to 0 so both Sum and Cout would be zero. For test case four, A was set to 0, B was set to 1, and Cin was set to 1. This gives that the Sum would be 0 since A XOR B XOR C is 0 XOR 1 XOR 1. This logic follows for Cout as shown in Figure 4 as well as the other test cases.

```
-- Test Case 4: A=0, B=1, Cin=1
A <= '0';        -- plug in the values
B <= '1';
Cin <= '1';
-- Sum: 0 (0 XOR 1 XOR 1)
-- Cout: 1 ((0 AND 1) OR (1 AND (0 XOR 1)))
wait for 10 ns;  -- this is needed for simulation
```

Figure 4: Full Adder sample test bench case 4 code

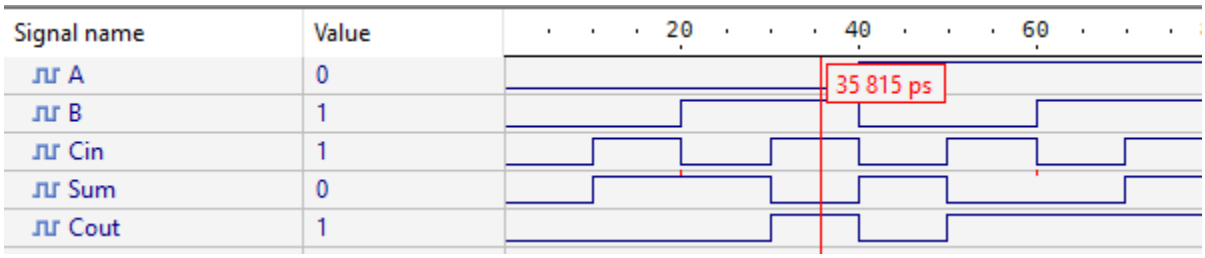


Figure 5: Full Adder test bench test case 4

C. 16-bit Adder

A test bench was created and verified that our half adder was working correctly. Two different cases were tested. For the first case, A was set as 2 and B as -2. Therefore the Sum = A+B = 2-2 = 0. As shown in figure 7, for the first 10ns, A is set as 0002 which is 2 and B as FFFE which is -2 with the result as 0. For the second test cast, we set A as 3 instead of 2. This resulted in a sum of 1 as shown in the timing diagram.

```
-- Test #1
A <= "0000000000000010";  --  2
B <= "1111111111111110";  -- -2
-- Sum = 0
wait for 10 ns;

-- Test #2
A <= "0000000000000011";  --  3
B <= "1111111111111110";  -- -2
-- Sum = 1
wait for 10 ns;
```

Figure 6: 16-bit Adder Test case code

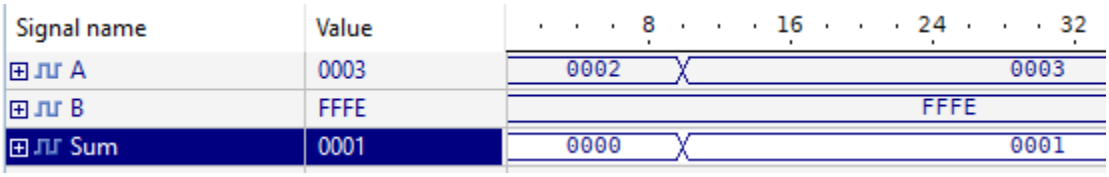


Figure 7: 16-bit Adder test bench test case

D. 16-bit Subtractor

A test bench was created and verified that our half adder was working correctly. For the below test case in figure 8, we set A as 0xFFFE and B as 0x000A in binary. The result was 0xFF4.

```
-- Test case 1
A_tb <= "1111111111111110"; -- 0xFFFE or -2
B_tb <= "0000000000001010"; -- 0x000A or 10
-- Correct output: 0xFFF4
wait for 10 ns;
```

Figure 8: 16-bit Subtractor Test case code

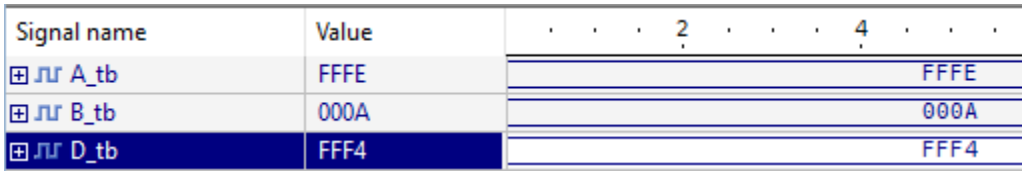


Figure 9: 16-bit Subtractor testbench test case

E. 16-bit Multiplier

A test bench was created and verified that our 16-bit multiplier was working correctly. Two different cases were tested. For the first case, A and B were set as 2 in binary. Therefore the Result = A*B = 2*2 = 4. The second case A and B were set as 3 and 9 subsequently which results as an output of 27. As shown in figure 11, for the first 10ns, A and B are set to 0002 which results as 4. For the second test cast, you can see the correct result of 0000001B which is 27 in hexadecimal.

```
-- Test Case 1
A <= "0000000000000010"; -- 2
B <= "0000000000000010"; -- 2
-- Result = 4
wait for 10 ns;

        -- Test Case 2
A <= "0000000000000011"; -- 3
B <= "0000000000001001"; -- 9
-- Result = 27 = 0000001B
wait for 10 ns;
```

Figure 10: 16-bit Multiplier Test case code

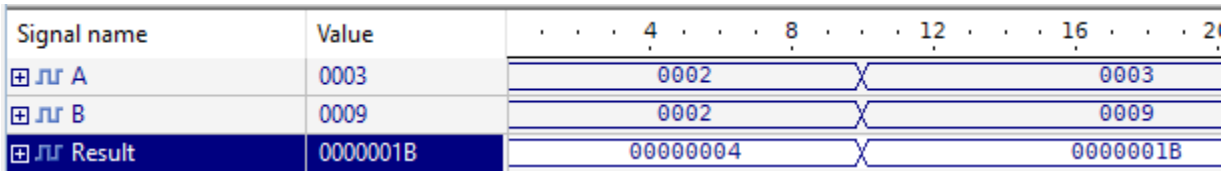


Figure 11: 16-bit Subtractor testbench test case

F. 16-bit Multiplexer

A test bench was created and verified that our 16-bit multiplexer was working correctly. Two different cases were tested. Keep in mind that even though the comments in figure 10 say the selection is for multiplication or passthrough B, these have not been setup yet. They are what will be implemented in the ALU. Currently as the code is written, A0 through A7, are arbitrary numbers to show that the multiplexer chooses that selection. As shown in figure 11, this is indeed true. In the first 10ns, selection "sel" 1 results in AAAA which is what corresponds to A1. In the second test case and next 10 ns, "sel" 3 results in 0000 which corresponds to A4.

```
-- Arbitrarily chose example values
A0 <= "0101010101010101";
A1 <= "1010101010101010";
A2 <= "1111111111111111";
A3 <= "0000000000000000";
A4 <= "1100110011001100";
A5 <= "0011001100110011";
A6 <= "0101010101010101";
A7 <= "1010101010101010";

-- Test Case 1
sel <= "001";   -- Setting selection to what will be multiplication
wait for 10 ns; -- result should show A1

-- Test Case 2
sel <= "011";   -- Setting selection to what will be Passthrough B
wait for 10 ns; -- result should show A3
```

Figure 12: 16-bit Multiplexer Test case code

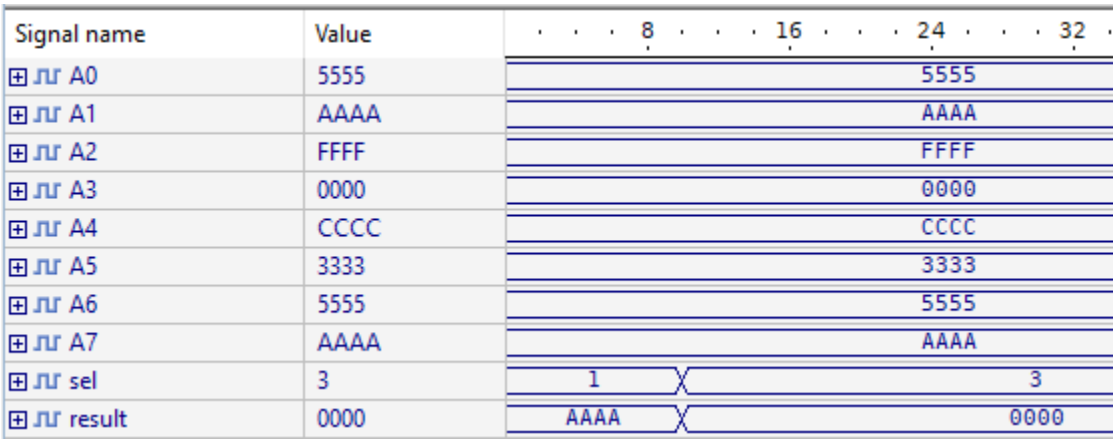


Figure 13: 16-bit Subtractor testbench test case