Faculty of Engineering & Technology Electrical & Computer Engineering Department

ENCS3310 Advanced Digital Design

Course Project Report

Prepared by:

Diaa Badaha 1210478

Instructor: Dr. Elias Khalil

Section: 3

Date: 21/1/2023

Abstract

This report presents the development process of a simplified microprocessor, depending on the design and implementation of an Arithmetic Logic Unit (ALU) and its associated register file. The ALU supports a variety of arithmetic and logical operations on 32-bit inputs, interfacing seamlessly with the register file. To ascertain the integrity and functionality of the design, a comprehensive testbench was constructed. A testbench was developed to validate the design, demonstrating the system's capability to execute predefined machine code instructions efficiently. This project exemplifies the practical use of digital design fundamentals via Verilog, highlighting the importance of component synchronization in microprocessors.

Table of Contents

Table of figures

List of Tables

Design and Implementation

The main ideas of the microprocessor design are focused on efficiency, scalability, and commitment to the project specifications.

ALU

The initial step was the construction of an Arithmetic Logic Unit (ALU), designed to carry out operations as determined by an input opcode. This step was crucial, as the ALU is the cornerstone for processing instructions. The customization process was particularly tailored to the unique identifier provided by the last digit of my student ID, '8', which dictated the exclusive set of operations the ALU was to support. To do this, I developed the ALU module using case statements within the Verilog code, which determined the specific operation and corresponding output based on the given opcode. This approach ensured that the ALU's functionality was directly aligned with the designated opcode inputs. The following table shows my opcodes.

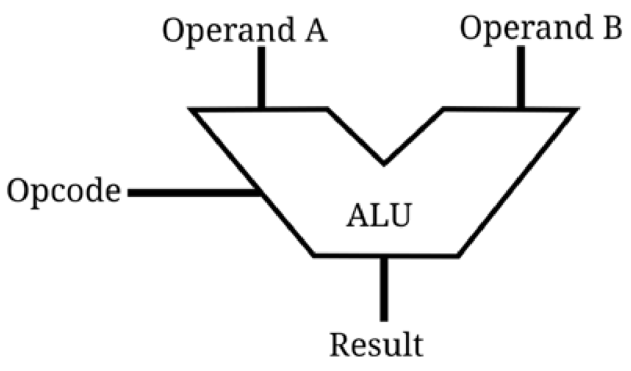


Figure 1: Designed ALU

Opcode	1	6	13	8	7	4	11	15	3	5	2
Operation	Add	Sub	Abs	Negative	Max	Min	Avg	Not	OR	AND	XOR

Table 1: Opcodes Based on Last Digit '8'

Register File

Following the ALU, the next step involved developing a register file associated with the ALU. The register file was to be pre-loaded with a set of values, as specified by the project. Here, the second-from-last digit of my student ID, '7', specified the 32 distinct values to populate the register file. To implement this, I defined an array of 32 registers, each being 32 bits in width. These values were then entered in an initial block to set up the register file at the beginning of the simulation.

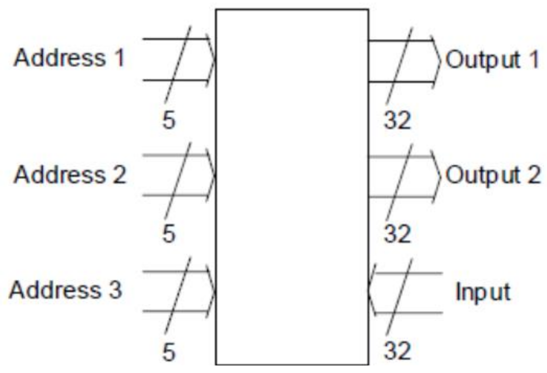


Figure 2: Designed Register File

The following tables detail these assigned values. The output of each register was assigned based on specific addresses. Moreover, inputs to the registers were set to update at every positive clock edge, depending on the state of a 'valid_opcode' bit, which ensured that only valid instructions would be executed, preserving the integrity of the microprocessor's operations.

Index	0	1	2	3	4	5	6	7
Value	0	15034	8854	170	7226	4480	8928	1044

Table 2: Register Values Based on Second-From-Last Digit '7' - 1

Index	8	9	10	11	12	13	14	15
Value	6706	258	7354	3264	14740	6532	10436	11900

Table 3: Register Values Based on Second-From-Last Digit '7' – 2

Index	16	17	18	19	20	21	22	23
Value	14694	8830	8712	4532	9084	13838	9084	13838

Table 4: Register Values Based on Second-From-Last Digit '7' - 3

Index	24	25	26	27	28	29	30	31
Value	10018	1280	5814	670	8832	15186	4512	0

Table 5: Register Values Based on Second-From-Last Digit '7' – 4

Microprocessor Core

Subsequently, I focused on constructing the core of the microprocessor, which integrates the ALU with the register file. A critical aspect of this process was to implement a mechanism for validating the input opcode to ensure its validity before execution. This design was started by distributing the instruction bits as shown in table 6. To ensure the validity of the opcode, I employed an approach where an array containing all possible opcodes was defined. During operation, a for loop would iterate through this array to determine if the input opcode matched any of the valid opcodes listed. If a match was found, the opcode was considered valid; otherwise, it was classified as invalid. With the validation operation in place, I then proceeded to instantiate the previously developed ALU and register file modules, completing the architecture of the microprocessor core.

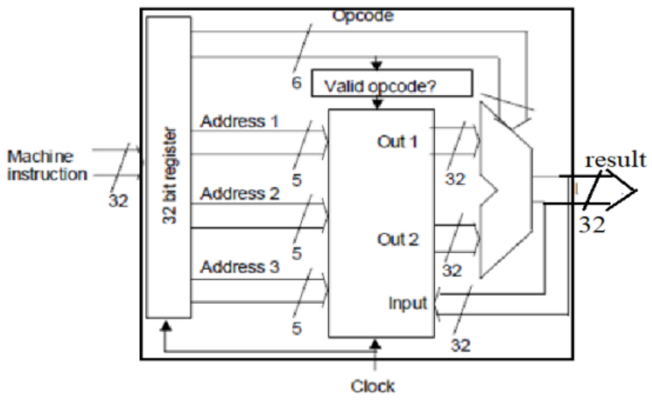


Figure 3: Designed Microprocessor Core

Bits of instruction [32:0]	[5:0]	[10:6]	[15:11]	[20:16]	[31:21]
Distribution	opcode	Address 1	Address 2	Address 3	Unused

Table 6: Register Bits Distribution

Synchronizing to Rising Clock Edge

On each rising clock edge, the microprocessor core performs a sequence of operations that constitute a single cycle of instruction execution. The opcode and register addresses are first extracted from the incoming 32-bit instruction, setting the stage for the operation. The opcode is then checked against a predefined array of valid opcodes; if a match is found, the 'valid_opcode' flag is set to true, so the instruction can proceed.

If the opcode is validated , the register file module reads the data from two specified registers corresponding to 'addr1' and 'addr2' and also writes new data into the register at 'addr3' with the input from the ALU's previous computation.

Meanwhile, the ALU is instantiated, awaiting inputs from the register file. It performs the computation based on the opcode as addition, subtraction, or any other defined operation and the results are ready for use in the next cycle. This process ensures that at every clock pulse, the microprocessor core is actively decoding, validating, and executing instructions in a synchronized manner.

Testing and Validation

ALU Test

Since the ALU plays a critical role within the microprocessor core, I had constructed a testbench in order to validate its performance. To do this, I initialized two inputs: a with a value of 5 and b with a value of 3, and set the initial opcode to 0. I then employed a repeat(15) loop, incrementing the opcode by 1 in each iteration, to verify each opcode's functionality by observing the output ‘result’.

As in Figure 4, Appendix 1, The results from the ALU tests confirmed that arithmetic and logic operations were performed accurately. This ensured that every operation executed by the ALU produced the correct result which calculated manually.

Register File Test

As the ALU, the register file is an essential component of the microprocessor, so a testbench was built to confirm its reliability. This verification involved executing six distinct read and write operations, then checking the success of each transaction was implemented. Additionally, the test conditions accounted for scenarios with invalid opcodes to ensure the register file responded correctly by prohibiting operations under such circumstances.

The results showed that the read and write operations were found to be reliable, with the register values reflecting the data from the test input. The valid_opcode control mechanism effectively enabled the write operation only when the opcode was recognized as valid, and disabled it when the opcode was recognized as invalid providing an additional layer of error checking. Results are shown in Figure 5, Appendix 1.

Top Module Test

The mp_top module stands as the most important part of this project, representing the entire microprocessor core that is under construction. Consequently, the most complicated and comprehensive testbench was developed specially for this module. This testing is critical, as it includes the entire array of functionalities and interactions within the microprocessor's architecture. It serves to validate the related operation of the ALU and register file, ensuring that the processor performs as intended when executing instructions.

For the creation of this testbench, I initiated by defining an array of 12 instructions: one for each of the 11 valid opcodes and an additional one to test the system's response to an invalid opcode. These instructions were carefully chosen to cover all possible opcodes. Alongside this, I prepared an array of 12 expected outcomes to compare them with the outputs from the mp_top module.

Initially, I manually computed the expected results for each instruction and assigned them directly to the array of expected outcomes. However, I found out that this method tended to be impractical for testing. Instead of this, I defined a memory within the testbench that mirrored the contents of the register file. Using custom functions such as absolute_value, min_value, and max_value, along with operators like '+', '-', '~', '&', '|', and '^', I automated the calculation of expected results. This approach improved the practicality of the testbench, providing the flexibility to adjust memory values.

With the instruction and expected results arrays are filled, I implemented a for loop to process and validate each instruction output against its expected result. The values were then compared, with a 'pass' being printed for matches and a 'fail' for mismatches. A 'fail_flag', initialized to 0, was assigned to one upon each test failure to check if the whole test had passed or failed.

The test concluded with a final evaluation based on the 'fail_flag' value. A zero value indicated that all cases had passed, indicating to a successful test. Conversely, if that flag was one then one or more cases had failed, indicating to a failed test result. The end of the testbench process was marked by printing out the overall outcome, clearly telling the test's success or failure.

As shown in Figure 7, Appendix 1, the outcomes of this test clearly demonstrate that the testbench has been constructed properly, as they display the results from the mp_top module and the expected outcomes. This allows the tester to visually confirm the accuracy of the results. Moreover, the results indicate whether each test case has passed or failed within its given instruction, providing an immediate and clear verification of the system's performance.

A Fail Case

To make a fail case, I swapped the instructions for addition and subtraction while maintaining the original expected values, leading to a predictable failure of those two test cases. Figure 7 and Figure 8 in Appendix 1 show the instructions both before and after the swap, as well as the resulting outputs that reflect the unsuccessful test outcomes

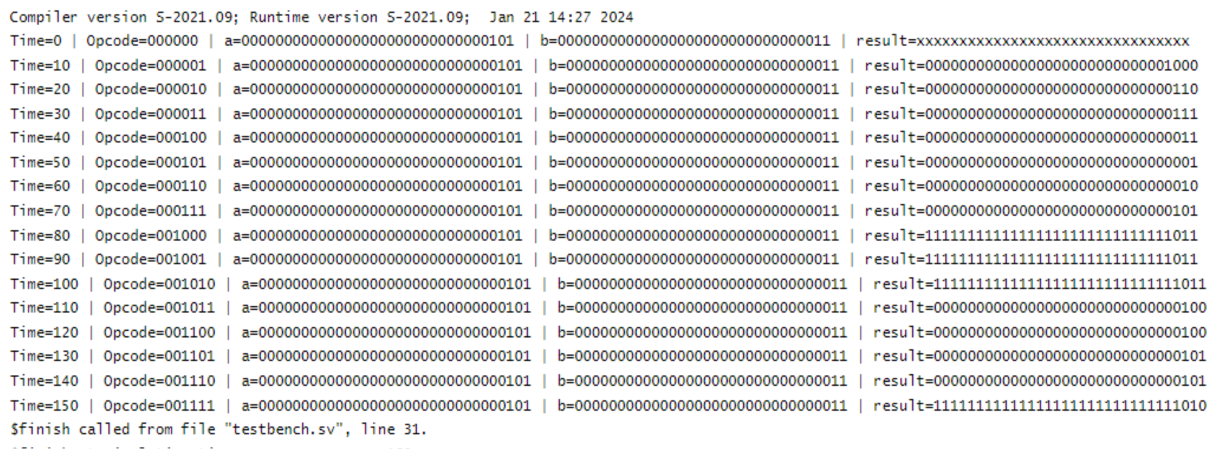
The results in Figure 9, Appendix 1, indicate that as expected, the first two test cases have failed due to the swapping of addition and subtraction instructions. This mismatch between the actual results and the expected values led to the failure of the entire test, as denoted by the final note "At least one case failed to pass the test successfully!".

Conclusion

In conclusion, we had arrived to a successful design, implementation, and verification of a custom microprocessor core, including an ALU and register file, based on specific operational codes. The comprehensive testing confirmed the functionality and reliability of each component, with all test cases corresponding to expected results. The testing has verified the microprocessor's integrity for executing machine-level code in a controlled simulation, marking the project's objectives as successfully achieved.

Appendix 1

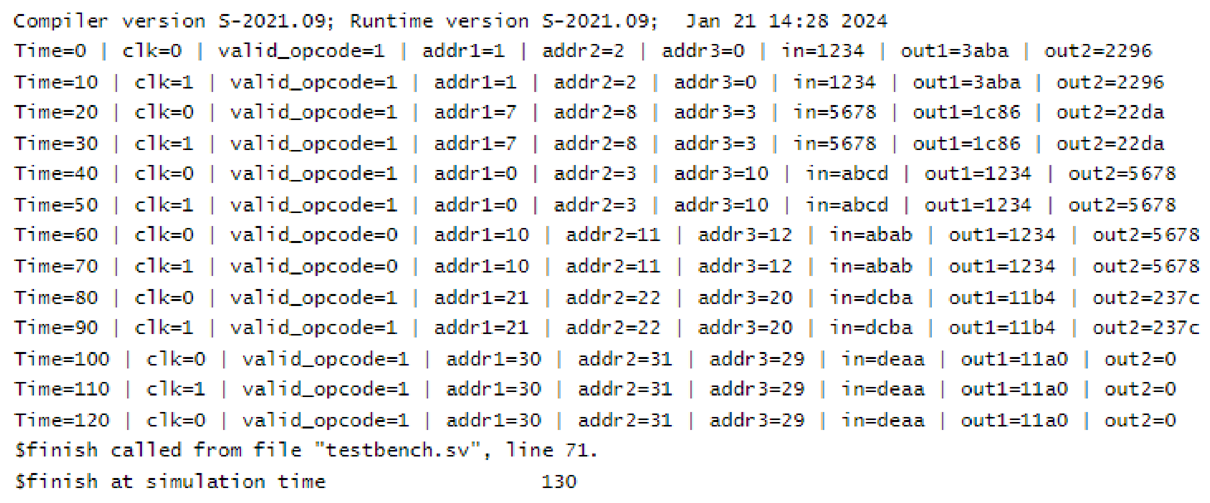
ALU Test Results



```
Compiler version S-2021.09; Runtime version S-2021.09;  Jan 21 14:27 2024
Time=0 | Opcode=000000 | a=00000000000000000000000000000101 | b=00000000000000000000000000000011 | result=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Time=10 | Opcode=000001 | a=00000000000000000000000000000101 | b=00000000000000000000000000000011 | result=00000000000000000000000000001000
Time=20 | Opcode=000010 | a=00000000000000000000000000000101 | b=00000000000000000000000000000011 | result=00000000000000000000000000000110
Time=30 | Opcode=000011 | a=00000000000000000000000000000101 | b=00000000000000000000000000000011 | result=00000000000000000000000000000111
Time=40 | Opcode=000100 | a=00000000000000000000000000000101 | b=00000000000000000000000000000011 | result=00000000000000000000000000000011
Time=50 | Opcode=000101 | a=00000000000000000000000000000101 | b=00000000000000000000000000000011 | result=00000000000000000000000000000001
Time=60 | Opcode=000110 | a=00000000000000000000000000000101 | b=00000000000000000000000000000011 | result=00000000000000000000000000000010
Time=70 | Opcode=000111 | a=00000000000000000000000000000101 | b=00000000000000000000000000000011 | result=00000000000000000000000000000101
Time=80 | Opcode=001000 | a=00000000000000000000000000000101 | b=00000000000000000000000000000011 | result=11111111111111111111111111111011
Time=90 | Opcode=001001 | a=00000000000000000000000000000101 | b=00000000000000000000000000000011 | result=11111111111111111111111111111011
Time=100 | Opcode=001010 | a=00000000000000000000000000000101 | b=00000000000000000000000000000011 | result=11111111111111111111111111111011
Time=110 | Opcode=001011 | a=00000000000000000000000000000101 | b=00000000000000000000000000000011 | result=00000000000000000000000000000100
Time=120 | Opcode=001100 | a=00000000000000000000000000000101 | b=00000000000000000000000000000011 | result=00000000000000000000000000000100
Time=130 | Opcode=001101 | a=00000000000000000000000000000101 | b=00000000000000000000000000000011 | result=00000000000000000000000000000101
Time=140 | Opcode=001110 | a=00000000000000000000000000000101 | b=00000000000000000000000000000011 | result=00000000000000000000000000000101
Time=150 | Opcode=001111 | a=00000000000000000000000000000101 | b=00000000000000000000000000000011 | result=11111111111111111111111111111010
$finish called from file "testbench.sv", line 31.
```

Figure 4: ALU Test Results

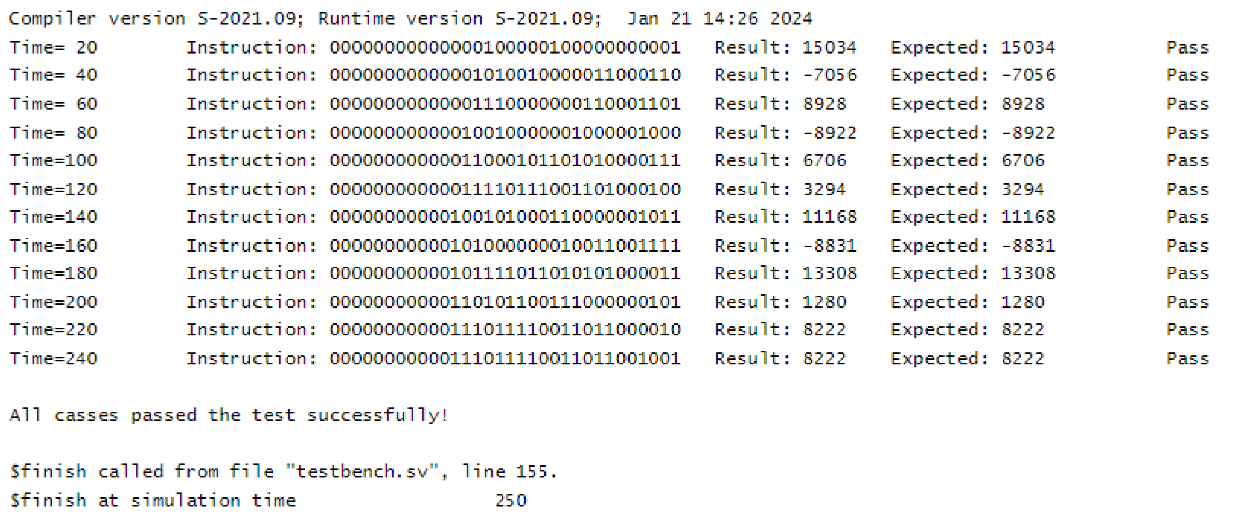
Register File Test Results



```
Compiler version S-2021.09; Runtime version S-2021.09;  Jan 21 14:28 2024
Time=0 | clk=0 | valid_opcode=1 | addr1=1 | addr2=2 | addr3=0 | in=1234 | out1=3aba | out2=2296
Time=10 | clk=1 | valid_opcode=1 | addr1=1 | addr2=2 | addr3=0 | in=1234 | out1=3aba | out2=2296
Time=20 | clk=0 | valid_opcode=1 | addr1=7 | addr2=8 | addr3=3 | in=5678 | out1=1c86 | out2=22da
Time=30 | clk=1 | valid_opcode=1 | addr1=7 | addr2=8 | addr3=3 | in=5678 | out1=1c86 | out2=22da
Time=40 | clk=0 | valid_opcode=1 | addr1=0 | addr2=3 | addr3=10 | in=abcd | out1=1234 | out2=5678
Time=50 | clk=1 | valid_opcode=1 | addr1=0 | addr2=3 | addr3=10 | in=abcd | out1=1234 | out2=5678
Time=60 | clk=0 | valid_opcode=0 | addr1=10 | addr2=11 | addr3=12 | in=abab | out1=1234 | out2=5678
Time=70 | clk=1 | valid_opcode=0 | addr1=10 | addr2=11 | addr3=12 | in=abab | out1=1234 | out2=5678
Time=80 | clk=0 | valid_opcode=1 | addr1=21 | addr2=22 | addr3=20 | in=dcba | out1=11b4 | out2=237c
Time=90 | clk=1 | valid_opcode=1 | addr1=21 | addr2=22 | addr3=20 | in=dcba | out1=11b4 | out2=237c
Time=100 | clk=0 | valid_opcode=1 | addr1=30 | addr2=31 | addr3=29 | in=deaa | out1=11a0 | out2=0
Time=110 | clk=1 | valid_opcode=1 | addr1=30 | addr2=31 | addr3=29 | in=deaa | out1=11a0 | out2=0
Time=120 | clk=0 | valid_opcode=1 | addr1=30 | addr2=31 | addr3=29 | in=deaa | out1=11a0 | out2=0
$finish called from file "testbench.sv", line 71.
$finish at simulation time                  130
```

Figure 5: Register File Test Results

Microprocessor Core Results

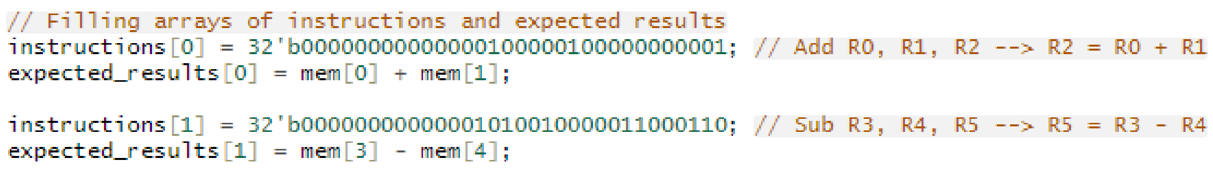


```
Compiler version S-2021.09; Runtime version S-2021.09;  Jan 21 14:26 2024
Time= 20        Instruction: 00000000000000010000010000000001   Result: 15034   Expected: 15034         Pass
Time= 40        Instruction: 00000000000001010010000011000110   Result: -7056   Expected: -7056         Pass
Time= 60        Instruction: 00000000000001110000000110001101   Result: 8928    Expected: 8928          Pass
Time= 80        Instruction: 00000000000010010000001000001000   Result: -8922   Expected: -8922         Pass
Time=100        Instruction: 00000000000011000101101010000111   Result: 6706    Expected: 6706          Pass
Time=120        Instruction: 00000000000011110111001101000100   Result: 3294    Expected: 3294          Pass
Time=140        Instruction: 00000000000100101000110000001011   Result: 11168   Expected: 11168         Pass
Time=160        Instruction: 00000000000101000000010011001111   Result: -8831   Expected: -8831         Pass
Time=180        Instruction: 00000000000101111011010101000011   Result: 13308   Expected: 13308         Pass
Time=200        Instruction: 00000000000110101100111000000101   Result: 1280    Expected: 1280          Pass
Time=220        Instruction: 00000000000111011110011011000010   Result: 8222    Expected: 8222          Pass
Time=240        Instruction: 00000000000111011110011011001001   Result: 8222    Expected: 8222          Pass

All casses passed the test successfully!

$finish called from file "testbench.sv", line 155.
$finish at simulation time                250
```

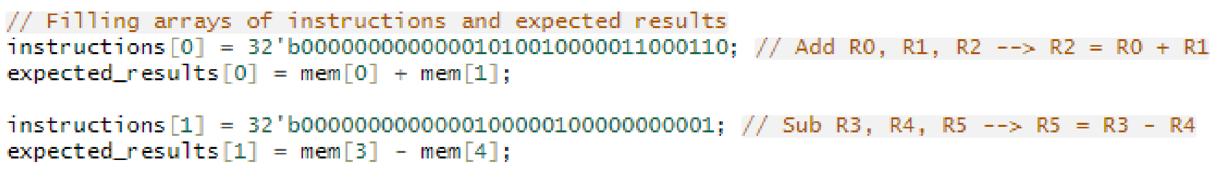
Figure 6: Microprocessor Core Results – 1



```
// Filling arrays of instructions and expected results
instructions[0] = 32'b00000000000000010000010000000001; // Add R0, R1, R2 --> R2 = R0 + R1
expected_results[0] = mem[0] + mem[1];

instructions[1] = 32'b00000000000001010010000011000110; // Sub R3, R4, R5 --> R5 = R3 - R4
expected_results[1] = mem[3] - mem[4];
```

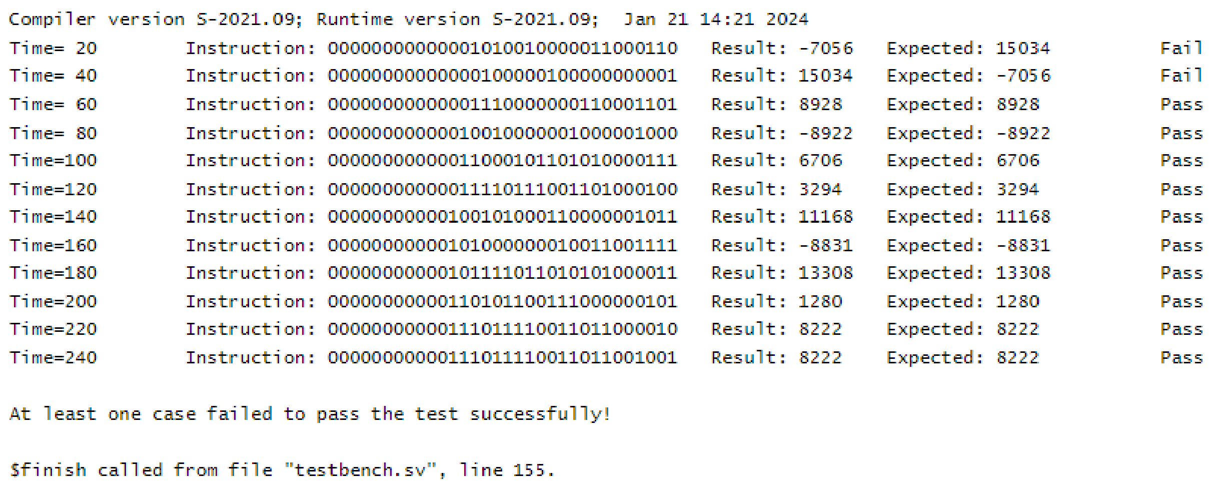
Figure 7: Instructions before The Swap



```
// Filling arrays of instructions and expected results
instructions[0] = 32'b00000000000001010010000011000110; // Add R0, R1, R2 --> R2 = R0 + R1
expected_results[0] = mem[0] + mem[1];

instructions[1] = 32'b00000000000000010000010000000001; // Sub R3, R4, R5 --> R5 = R3 - R4
expected_results[1] = mem[3] - mem[4];
```

Figure 8: Instructions after The Swap



```
Compiler version S-2021.09; Runtime version S-2021.09;  Jan 21 14:21 2024
Time= 20        Instruction: 00000000000001010010000011000110   Result: -7056   Expected: 15034         Fail
Time= 40        Instruction: 00000000000000010000010000000001   Result: 15034   Expected: -7056         Fail
Time= 60        Instruction: 00000000000001110000000110001101   Result: 8928    Expected: 8928          Pass
Time= 80        Instruction: 00000000000010010000001000001000   Result: -8922   Expected: -8922         Pass
Time=100        Instruction: 00000000000011000101101010000111   Result: 6706    Expected: 6706          Pass
Time=120        Instruction: 00000000000011110111001101000100   Result: 3294    Expected: 3294          Pass
Time=140        Instruction: 00000000000100101000110000001011   Result: 11168   Expected: 11168         Pass
Time=160        Instruction: 00000000000101000000010011001111   Result: -8831   Expected: -8831         Pass
Time=180        Instruction: 00000000000101111011010101000011   Result: 13308   Expected: 13308         Pass
Time=200        Instruction: 00000000000110101100111000000101   Result: 1280    Expected: 1280          Pass
Time=220        Instruction: 00000000000111011110011011000010   Result: 8222    Expected: 8222          Pass
Time=240        Instruction: 00000000000111011110011011001001   Result: 8222    Expected: 8222          Pass

At least one case failed to pass the test successfully!

$finish called from file "testbench.sv", line 155.
```

Figure 6: Fail Case Output