Miniproject 4 Report

Nividh Singh, Ahan Trivedi, Ishan Porwal, Ertug Umsur

April 22, 2025

Table of Contents

Link to GitHub for this project:
https://github.com/Ahan-Trivedi2/Computer_Architecture_MP4/tree/main

Processor Architecture Overview

In Miniproject 4, we designed and implemented a 32-bit RV32I RISC-V processor. The processor followed a Von Neumann architecture, meaning both instructions and data were stored in the same memory space. It was built as an unpipelined, multicycle processor using a finite state machine (FSM). Instructions progress through five distinct states in our processor. The first, MEMORY_PULL, initiates the memory read by using the provided program counter address. In the following FETCH stage, the fetched instruction is latched into the instruction register. The EXECUTE stage performs computation, control flow decisions, and memory address calculations for both store and load operations. For load instructions, a dedicated SECOND_EXECUTE stage follows, where the data retrieved from memory is written back to the destination register. Finally, PC_UPDATE advances the program counter to the next instruction, completing the cycle before it restarts. This expanded FSM allowed for clearer separation of control between memory and execution stages, particularly for load instructions requiring multiple cycles. The processor supports all major instruction types in the base RV32I set, excluding the following instructions: ecall, ebreak, csrrw, csrrs, csrrc, csrrwi, csrrsi, and csrrci. To verify correctness, we ran a suite of compiled RV32I RISC-V machine code instructions using a testbench and checked the corresponding register or memory location to ensure the correct value was stored. We also used GTKWave to visualize the processor's internal data flow and confirm that instructions moved through the FSM and datapath as expected. These tests confirmed that our implementation correctly handled the supported instruction set across a variety of scenarios. Screenshots verifying test results are included in the Simulation Result Analysis section of this report.

Memory Integration and Operation Through Instruction Register (MEMORY_PULL, FETCH)

Our processor utilizes a single 8 kB memory module to store both instructions and data. This unified memory is accessed through the same interface for both instruction fetches and data read/write operations. Before simulation, we preload the memory with compiled RV32I RISC-V machine code using an INIT_FILE parameter, which loads a hexadecimal file containing instructions.

During the MEMORY_PULL stage of the FSM, the Program Counter (PC) provides an address to the memory module to prepare for the instruction fetch. The following FETCH stage reads this instruction from memory and latches it into the Instruction Register (IR), holding it steady across multiple cycles as the processor moves through the EXECUTE and SECOND_EXECUTE stages. This ensures that the instruction remains available throughout the multicycle operation, even as memory may be used for other purposes like data access.

The memory is byte-addressable, allowing access to bytes, half-words (16 bits), and full words (32 bits), in compliance with the RISC-V specification. For load and store instructions, the ALU computes the effective address using register values, and the control unit activates the appropriate memory signals. The memory_funct3 signal is set to 3'b010 during instruction fetches and is dynamically assigned to the instruction's funct3 value during memory accesses.

Control Unit, Decoding, Execution, and PC Update (EXECUTE, PC_UPDATE)

The control unit serves as the brain of the processor, managing instruction execution through a five-stage finite state machine. At each stage, it generates the necessary control signals to coordinate the behavior of all major datapath components.

As noted in the previous section, instructions are fetched from memory during the MEMORY_PULL stage and are latched into the instruction register during FETCH stage. The latched instruction is then passed into the instruction_decode module, which extracts the key fields: opcode, rd, rs1, rs2, funct3, and funct7. These decoded outputs are used by the control unit to determine the instruction type and generate the appropriate control signals for the remainder of the cycle. If the instruction includes an immediate value, such as addi, lw, sw, beq, jal, etc., the instruction is also passed to the immediate_generator module, which extracts and sign-extends the correct immediate field based on the instruction format.

For arithmetic and logic instructions, the control unit enables the ALU and sets its alu_control mode accordingly (addition, subtraction, bit shifts, or logical operations). Registers that our ALU stores values into are defined by our register_file module. For memory instructions, the ALU computes the effective address, often by adding a register value and an immediate, and the control unit activates either a memory read or write based on the instruction's funct3 value. The control unit also sets the memory_funct3 field to indicate the width of the access (byte, halfword, or word). Store (S-type) instructions are completed in the EXECUTE stage, where data from the source register (rs2) is written directly to memory.

Load (I-type) instructions span two stages. In EXECUTE, the effective memory address is calculated. The processor then transitions to SECOND_EXECUTE, where the fetched memory data is written back to the destination register. The control unit enables register_write_en and ensures the correct data width is written, handling sign extension or zero extension as specified by funct3.

Branching is handled through coordination between the control unit and a dedicated branch_logic module. During the EXECUTE stage, the control unit checks the branch_taken signal, generated by comparing rs1 and rs2 register values according to the funct3 field. If the branch condition is met, the control unit sets pc_control to select the branch target and transitions through the remaining stages. If the branch is not taken, the processor proceeds normally.

Unconditional jumps such as JAL and JALR are also handled during EXECUTE. For JAL, the PC is updated to a PC-relative offset, and for JALR, it jumps to a register-relative target (rs1 + immediate). In both cases, the return address (PC + 4) is written to the destination register, and the processor continues through the remaining stages.

The final PC_UPDATE stage concludes every instruction cycle. If the PC was not modified by a branch or jump, the control unit enables the program counter to increment by 4, moving to the next sequential instruction in memory. This stage is necessary for instructions like ALU operations, loads, and stores, which do not modify the PC directly during EXECUTE.

Testing and Verification Methodology

We used a dedicated testbench to verify that the processor was operating as expected. This consisted of two main components: test files in text file format and the SystemVerilog testbench. The test files include the mandatory 2,048 lines of 32-bit hex values required for inputting into the memory module, with the first lines being the hexadecimal instructions that should be executed by the processor and the rest being zeros. The test files are designed so that they together contain representative examples of each type of RV32I instruction, which allows us to verify entire instruction set coverage. This provided a structured approach to validating processor behavior by comparing expected and actual outcomes for chosen instruction sequences.

The desired test file is input as a parameter into the testbench within the INIT_FILE parameter in the memory module. The testbench is then simulated using Icarus Verilog to compile the testbench into an executable, and the Icarus Verilog Runtime Engine to run the simulation. The testbench operates on a 12 MHz clock and performs a reset before execution begins. Verification is performed by analyzing the final states of the register file at the conclusion of the simulation, validating correct data manipulation and program flow control. Memory operations are also verified by writing data into memory (via store instructions) and reading it back into registers (via load instructions), confirming end-to-end correctness.

Additionally, GTKWave is used to trace internal processor signals, observing the movement through the five FSM stages. This ensures correct timing for instruction fetch, execution, and memory interactions. For load instructions, SECOND_EXECUTE is specifically monitored to confirm correct data capture and register writes. The Simulation Result Analysis section presents expected and actual register file values along with assembly and machine code breakdowns for each test.

Simulation Result Analysis

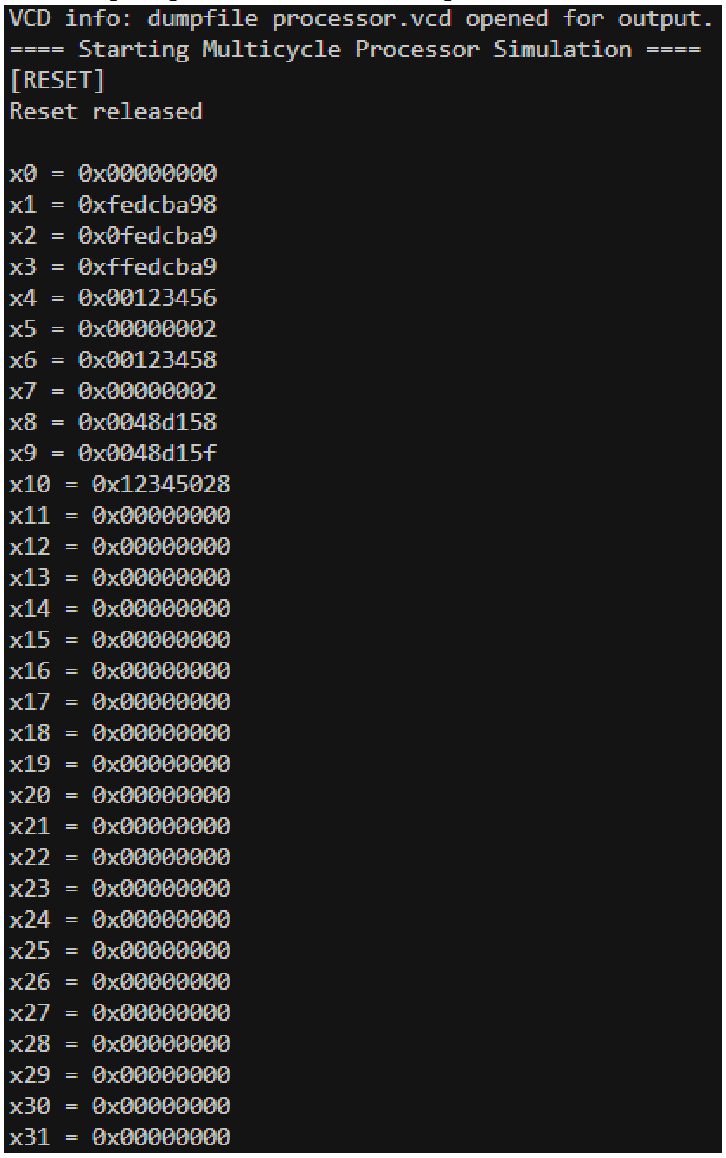
First Test File (test_1.txt):

```
fedcc0b7     ← lui x1, -4660
a9808093     ← addi x1, x1, -1384
0040d113     ← srli x2, x1, 4
4040d193     ← srai x3, x1, 4
fff1c213     ← xori x4, x3, -1
00200293     ← addi x5, x0, 2
00428333     ← add x6, x5, x4
404303b3     ← sub x7, x6, x4
00521433     ← sll x8, x4, x5
00746493     ← ori x9, x8, 7
12345517     ← auipc x10, 74565
```

Register	Expected Register Value
x0	0x00000000
x1	0xfedcba98
x2	0x0fedcba9
x3	0xffedcba9
x4	0x00123456
x5	0x00000002
x6	0x00123458
x7	0x00000002
x8	0x0048d158
x9	0x0048d15f
x10	0x12345028
x11	0x00000000
x12	0x00000000
x13	0x00000000
x14	0x00000000
x15	0x00000000
x16	0x00000000
x17	0x00000000
x18	0x00000000

x19	0x00000000
x20	0x00000000
x21	0x00000000
x22	0x00000000
x23	0x00000000
x24	0x00000000
x25	0x00000000
x26	0x00000000
x27	0x00000000
x28	0x00000000
x29	0x00000000
x30	0x00000000
x31	0x00000000

Resulting Register values from Register File:



```
VCD info: dumpfile processor.vcd opened for output.
==== Starting Multicycle Processor Simulation ====
[RESET]
Reset released

x0 = 0x00000000
x1 = 0xfedcba98
x2 = 0x0fedcba9
x3 = 0xffedcba9
x4 = 0x00123456
x5 = 0x00000002
x6 = 0x00123458
x7 = 0x00000002
x8 = 0x0048d158
x9 = 0x0048d15f
x10 = 0x12345028
x11 = 0x00000000
x12 = 0x00000000
x13 = 0x00000000
x14 = 0x00000000
x15 = 0x00000000
x16 = 0x00000000
x17 = 0x00000000
x18 = 0x00000000
x19 = 0x00000000
x20 = 0x00000000
x21 = 0x00000000
x22 = 0x00000000
x23 = 0x00000000
x24 = 0x00000000
x25 = 0x00000000
x26 = 0x00000000
x27 = 0x00000000
x28 = 0x00000000
x29 = 0x00000000
x30 = 0x00000000
x31 = 0x00000000
```

Analysis: The final testbench register output accurately matches with the expected values based on the given instruction set. Hence, this allows us to believe that the processor is working as expected.

<u>Second Test File (test_2.txt):</u>

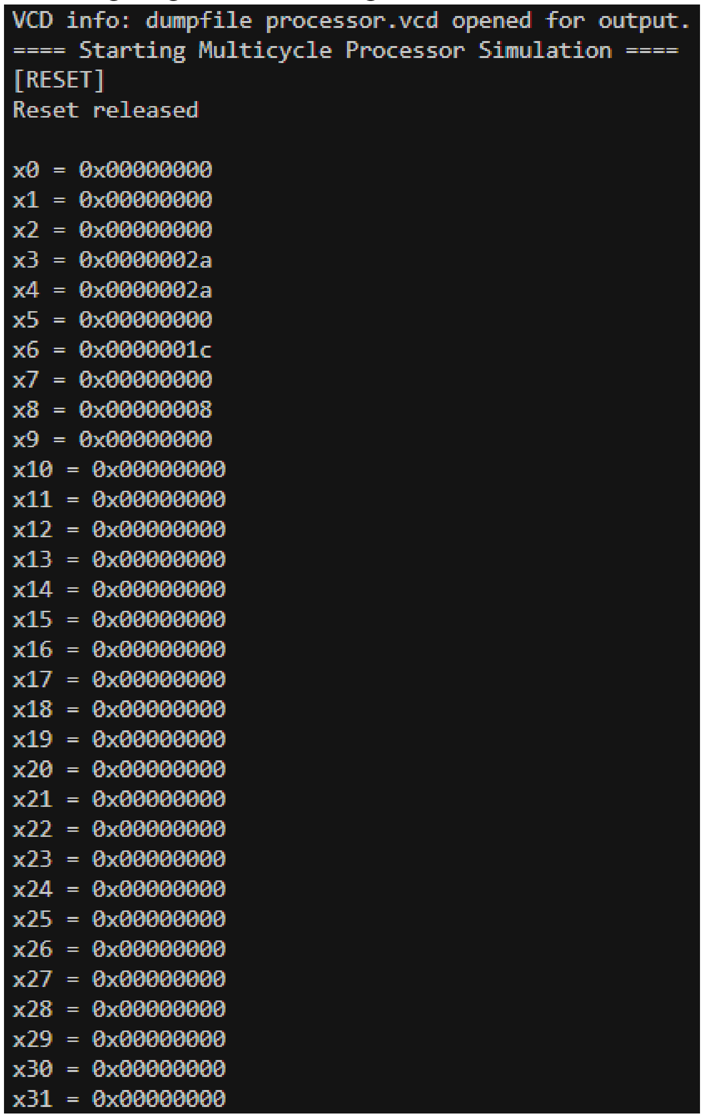
```
00000137     ← lui x2, 0
02A00193     ← addi x3, x0, 42
00312023     ← sw x3, 0(x2)
00012203     ← lw x4, 0(x2)
00320463     ← beq x4, x3, 8
00100293     ← addi x5, x0, 1
0080036f     ← jal x6, 8
00700393     ← addi x7, x0, 7
```

00800413 ← addi x8, x0, 8

Register	Expected Register Value
x0	0x00000000
x1	0x00000000
x2	0x00000000
x3	0x0000002A
x4	0x0000002A
x5	0x00000000
x6	0x0000001C
x7	0x00000000
x8	0x00000008
x9	0x00000000
x10	0x00000000
x11	0x00000000
x12	0x00000000
x13	0x00000000
x14	0x00000000
x15	0x00000000
x16	0x00000000
x17	0x00000000
x18	0x00000000
x19	0x00000000
x20	0x00000000
x21	0x00000000
x22	0x00000000
x23	0x00000000
x24	0x00000000
x25	0x00000000
x26	0x00000000
x27	0x00000000

x28	0x00000000
x29	0x00000000
x30	0x00000000
x31	0x00000000

Resulting Registers from Register File:



```
VCD info: dumpfile processor.vcd opened for output.
==== Starting Multicycle Processor Simulation ====
[RESET]
Reset released

x0 = 0x00000000
x1 = 0x00000000
x2 = 0x00000000
x3 = 0x0000002a
x4 = 0x0000002a
x5 = 0x00000000
x6 = 0x0000001c
x7 = 0x00000000
x8 = 0x00000008
x9 = 0x00000000
x10 = 0x00000000
x11 = 0x00000000
x12 = 0x00000000
x13 = 0x00000000
x14 = 0x00000000
x15 = 0x00000000
x16 = 0x00000000
x17 = 0x00000000
x18 = 0x00000000
x19 = 0x00000000
x20 = 0x00000000
x21 = 0x00000000
x22 = 0x00000000
x23 = 0x00000000
x24 = 0x00000000
x25 = 0x00000000
x26 = 0x00000000
x27 = 0x00000000
x28 = 0x00000000
x29 = 0x00000000
x30 = 0x00000000
x31 = 0x00000000
```

Analysis: The final testbench register output accurately matches with the expected values based on the given instruction set. Hence, this allows us to believe that the processor is working as expected.

<u>Third Test File (test_3.txt):</u>

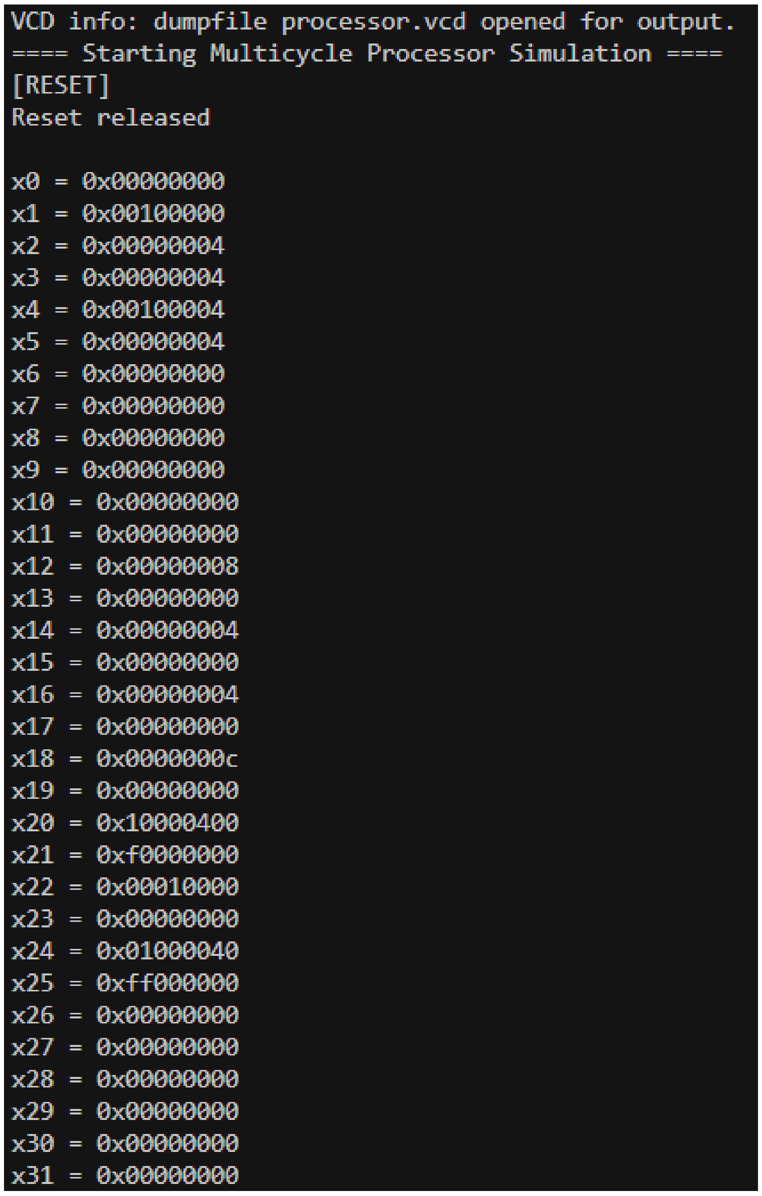
001000b7	← lui x1, 256
00000117	← auipc x2, 0

```
00212023      ← sw x2, 0(x2)
00402183      ← lw x3, 4(x0)
00208233      ← add x4, x1, x2
00410023      ← sb x4, 0(x2)
00400283      ← lb x5, 4(x0)
00510633      ← add x12, x2, x5
40260733      ← sub x14, x12, x2
00264933      ← xor x18, x12, x2
00266833      ← or x16, x12, x2
00227833      ← and x16, x4, x2
00221a33      ← sll x20, x4, x2
00225b33      ← srl x22, x4, x2
002a1a33      ← sll x20, x20, x2
402a5c33      ← sra x24, x20, x2
f0000ab7      ← lui x21, -65536
402adcb3      ← sra x25, x21, x2
```

Register	Expected Register Value
x0	0x00000000
x1	0x00100000
x2	0x00000004
x3	0x00000004
x4	0x00100004
x5	0x00000004
x6	0x00000000
x7	0x00000000
x8	0x00000000
x9	0x00000000
x10	0x00000000
x11	0x00000000
x12	0x00000008
x13	0x00000000
x14	0x00000004
x15	0x00000000
x16	0x00000004

x17	0x00000000
x18	0x0000000C
x19	0x00000000
x20	0x10000400
x21	0xF0000000
x22	0x00010000
x23	0x00000000
x24	0x01000040
x25	0xFF000000
x26	0x00000000
x27	0x00000000
x28	0x00000000
x29	0x00000000
x30	0x00000000
x31	0x00000000

Resulting Registers from Register File:



```
VCD info: dumpfile processor.vcd opened for output.
==== Starting Multicycle Processor Simulation ====
[RESET]
Reset released

x0 = 0x00000000
x1 = 0x00100000
x2 = 0x00000004
x3 = 0x00000004
x4 = 0x00100004
x5 = 0x00000004
x6 = 0x00000000
x7 = 0x00000000
x8 = 0x00000000
x9 = 0x00000000
x10 = 0x00000000
x11 = 0x00000000
x12 = 0x00000008
x13 = 0x00000000
x14 = 0x00000004
x15 = 0x00000000
x16 = 0x00000004
x17 = 0x00000000
x18 = 0x0000000c
x19 = 0x00000000
x20 = 0x10000400
x21 = 0xf0000000
x22 = 0x00010000
x23 = 0x00000000
x24 = 0x01000040
x25 = 0xff000000
x26 = 0x00000000
x27 = 0x00000000
x28 = 0x00000000
x29 = 0x00000000
x30 = 0x00000000
x31 = 0x00000000
```

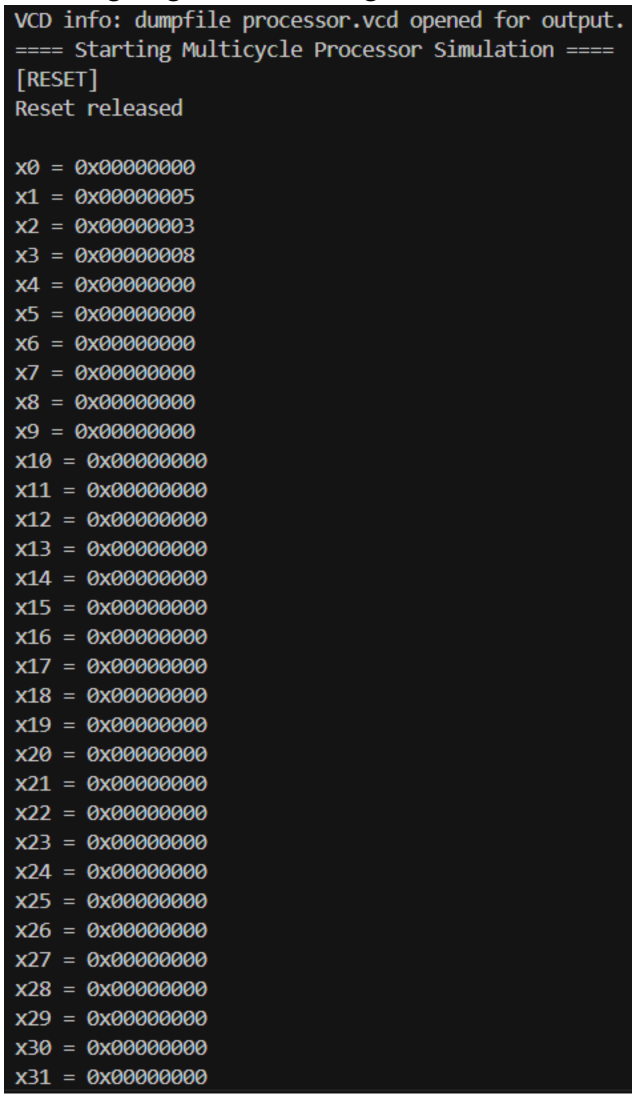
Analysis: The final testbench register output accurately matches with the expected values based on the given instruction set. Hence, this allows us to believe that the processor is working as expected. Also, with this specific test, we tested whether data was correctly being stored within the memory module by writing a known value from a register into memory using a store word instruction and then retrieving that value back into a different register using a load word instruction. This is confirmed by observing that x2 (the original value stored) and x3 (the value loaded back from memory) both hold 0x00000004 in the final output. The fact that the loaded value matched the stored value confirms correct functionality of both the memory write and memory read paths, as well as proper address calculation within the memory module.

Fourth Test File (datapath_test.txt):

```
00500093    ← addi x1, x0, 5
00300113    ← addi x2, x0, 3
002081b3    ← add x3, x1, x2
```

Register	Expected Register Value
x0	0x00000000
x1	0x00000005
x2	0x00000003
x3	0x00000008

Resulting Registers from Register Files

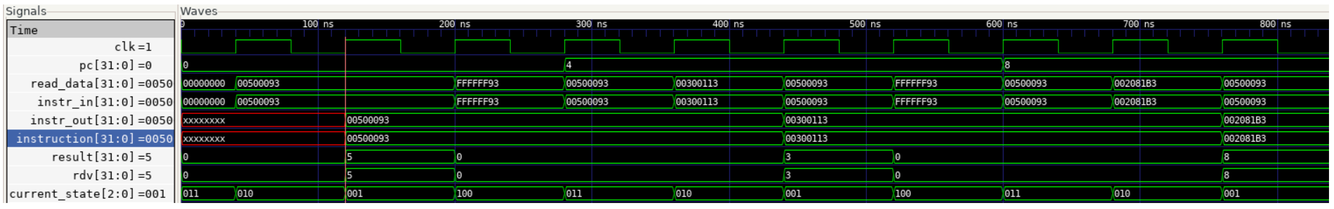


```
VCD info: dumpfile processor.vcd opened for output.
==== Starting Multicycle Processor Simulation ====
[RESET]
Reset released

x0 = 0x00000000
x1 = 0x00000005
x2 = 0x00000003
x3 = 0x00000008
x4 = 0x00000000
x5 = 0x00000000
x6 = 0x00000000
x7 = 0x00000000
x8 = 0x00000000
x9 = 0x00000000
x10 = 0x00000000
x11 = 0x00000000
x12 = 0x00000000
x13 = 0x00000000
x14 = 0x00000000
x15 = 0x00000000
x16 = 0x00000000
x17 = 0x00000000
x18 = 0x00000000
x19 = 0x00000000
x20 = 0x00000000
x21 = 0x00000000
x22 = 0x00000000
x23 = 0x00000000
x24 = 0x00000000
x25 = 0x00000000
x26 = 0x00000000
x27 = 0x00000000
x28 = 0x00000000
x29 = 0x00000000
x30 = 0x00000000
x31 = 0x00000000
```

Analysis: The final testbench register output accurately matches with the expected values based on the given instruction set. Hence, this allows us to believe that the processor is working as expected.

Data Path Verification from GTKWave



Analysis: The GTKWave screenshot illustrates the correct data flow through our processor as it executes instructions. At the first rising clock edge (far left of the screenshot), the processor enters the MEMORY_PULL stage (011), initiating a memory read for the instruction located at the program counter. Since the memory module is clocked, the instruction takes a full cycle to become available on read_data. At the second rising edge, the FSM transitions to the FETCH stage (010), and the fetched instruction (0x00500093, corresponding to addi x1, x0, 5) appears on instr_in[31:0]. This confirms that the instruction was successfully read from memory and latched into the instruction register. By the third rising edge, the processor enters the EXECUTE stage (001). The latched instruction is now available on instruction[31:0], and the ALU computes the result of the operation. The result of the addition (5) appears on both result[31:0] and rdv[31:0], indicating that the value was correctly computed and is ready to be written back to register x1.At the fourth rising edge, the processor enters the PC_UPDATE stage (100), updating the program counter to fetch the next instruction. This FSM cycle repeats for the remaining instructions. By the end of the sequence, the waveform confirms that the expected value (8) is correctly stored in the destination register, validating both the instruction execution and register write-back paths.

These four test instruction sets and results comprehensively cover all six instruction types in the RV32I ISA, and confirm that our data flow path is working correctly.

Design Justifications

We decided to have the processor organized into distinct, modular components. This approach allowed for easier debugging, readability, and allowed for isolated testing and verification when necessary.

We implemented the control unit using a five-stage finite state machine (FSM) consisting of MEMORY_PULL, FETCH, EXECUTE, SECOND_EXECUTE, and PC_UPDATE stages. This breakdown allowed us to isolate control logic responsibilities by stage and handle memory latency cleanly, especially for clocked memory modules with single-cycle read delays.

We decided to have a dedicated module for instruction decoding to separate instructions based on their opcodes while extracting the relevant bits and ensuring only relevant fields are used for each instruction, preventing unintended behavior, and removing complexity from having these tedious decoding tasks in the control unit.

We decided to have a module specifically for immediate generation that centralizes the logic for extracting and sign-extending immediate values from instructions. This allowed us to prevent redundant code and made it easier to debug issues related to computing immediate values.

We decided to create an individual branch logic module since it was a distinct form of logic. It didn't belong in the ALU and made for a cleaner control unit design, and having it separated made it much easier to analyze in GTKWave debugging.

We decided to group related control signals into compact, multibit control variables such as pc_control, ir_control, and alu_control rather than assigning a separate wire or Boolean signal to each function. This simplified the FSM control logic, allowed for better readability, and cleaner top-level wiring.

Conclusion

Our multicycle RV32I processor was able to successfully execute a wide range of integer instructions, including arithmetic, logic, memory access, and control-flow operations. We tested its functionality by checking the register-file final register values and using GTKWave to step through the execution cycle-by-cycle to confirm correct data flow. The processor correctly handled all six instruction types in the base RV32I set, with the exception of the instructions we were allowed to leave out. Looking ahead, we aim to build off this by either getting our processor working on hardware or adding multiply and divide operations into our ALU.