RISC-V Processor Core

Reece Wayt
Department of Electrical & Computer Engineering
Portland State University
Portland, OR, USA
reecwayt@pdx.edu

Maithreyi Venkatesan
Department of Electrical & Computer Engineering
Portland State University
Portland, OR, USA
maithven@pdx.edu

Philip Nevins
Department of Electrical & Computer Engineering
Portland State University
Portland, OR, USA
pnevins@pdx.edu

Raghavendra Davarapalli
Department of Electrical & Computer Engineering
Portland State University
Portland, OR,USA
ragha@pdx.edu

***Abstract*—This paper presents the implementation of a RISC-V processor core, focusing on the fundamental pipeline stages required for a modern processor design. The implementation was modeled and simulated using SystemVerilog and QuestaSim simulator tools. We've incorporated both structural and behavioral-level modeling approaches. We selected the RISC-V architecture for its open-source instruction set architecture (ISA), and its growing popularity in academic and industrial applications. While the current implementation is an unpipelined version, the design was constructed with future pipelining in mind; in that, with the addition of pipeline register and control logic, it would support pipelining. This project focuses on thorough unit testing and verification of the instruction cycle which takes 4-5 cycles in our unpipelined version. We did not focus on the synthesis of our design.**

Keywords—Architecture, SystemVerilog, Behavioral/Structural Modeling, RISC-V, ISA, RTL

I. Introduction

Reduced Instruction Set Computers (RISC) have been around since the early 1980s and were originally invented at UC Berkeley by David Patterson, Carlo Sequin, and associates. The RISC Project set out to assuage the trend towards increasingly complex computer instruction sets at the time [1]. They formalized this by defining a set of fixed-length instructions (32-bit) designed to support high-level languages. Not only this, but instructions were able to execute in one machine cycle, and load/store operations came from instruction memory accesses, the rest of the operations happen in register sets. Today we have the RISC-V which has maintained over 30 years of development since RISC-I but provides more efficiency plus extensions for 64-bit instructions and float point units for both 32-bit and 64-bit instructions [2].

This project focuses on the most recent RV32I Base Integer Instruction Set [3]. RV32I can be summarized by the following characteristics:

- Contains 40 unique instructions (see Appendix A for full instruction set)
- Sufficient ISA for compiler target and OS support
- 32x-register (x0-x32) each 32-bits wide; 0x is hardwired to zero
- x1 and x2 are subroutine return address pointer, and stack pointer respectively
- 6 instruction formats dependent on opcode (see Fig 1)
    - R-type: Register to register arithmetic instructions (ADD, SUB, AND, OR)
    - I-type: Immediate arithmetic and load instructions (ADDI, LW, JALR)
    - S-type: Store instructions that use base register and immediate offset (SW, SH, SB)
    - U-type: Long immediate instructions using upper 20 bits (LUI, AUIPC)
    - B-type: Conditional branch instructions (BEQ, BNE, BLT)
    - J-type: Unconditional jump instructions (JAL)

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

Fig 1: RISC-V Instruction Types

II. Architecture Overview

Implementation

The RISC-V core we've implemented closely follows the description found in [2], appendix C of that text. This processor core implements a five-stage pipeline consisting of Instruction Fetch, Instruction Decode, Execute, Memory Access, and Write Back stages; therefore, at most a single instruction will take 5 clock cycles. In this section we will talk in detail about the pipeline stage implementations, which was split up per team member. Below is a block diagram of the stages and each of the major modules you'll find in our HDL design.

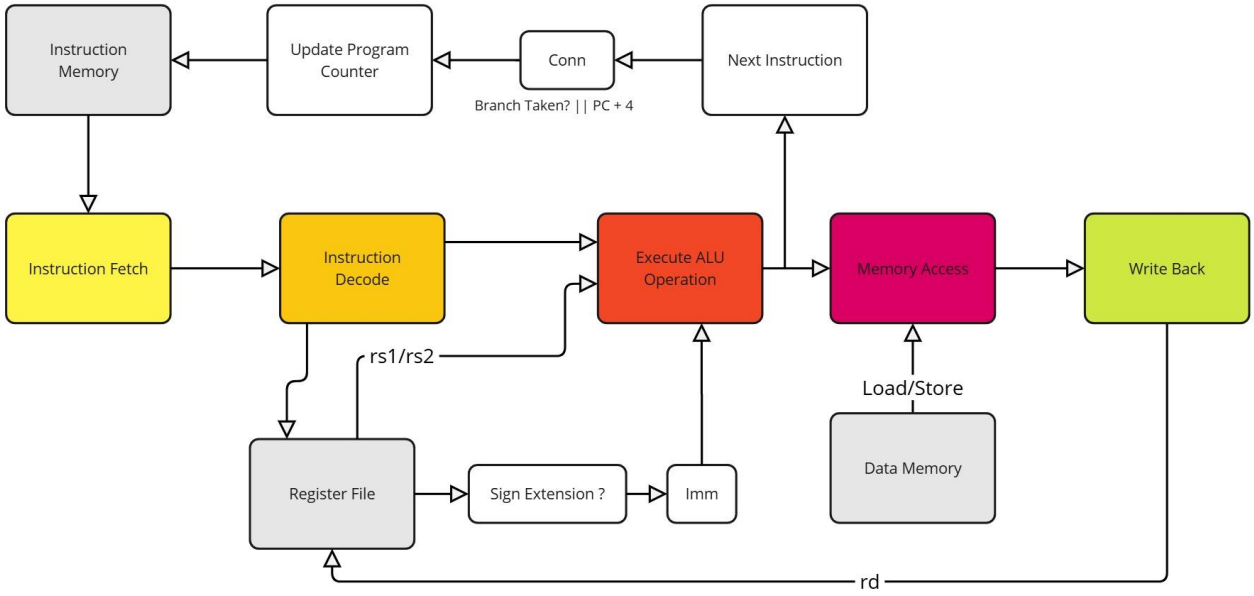


Fig 2: Pipeline Block Diagram

Pipeline Control

For inter-stage communication, we implemented a simple handshaking protocol using valid and ready signals between adjacent pipeline stages, rather than using a more complex finite state machine (FSM) approach. Each stage asserts a valid signal when it has completed its operation and has data ready for the next stage. The receiving stage asserts a ready signal when it is available to accept new data. This handshaking mechanism would not be optimal for performance in a pipeline processor, but was a straightforward and reliable way to manage the pipeline flow and

ensure proper data transfer between stages. Performance was not a concern for us in this project, but more work could be done to make this more efficient.

Pipeline Stages

Instruction Fetch

Internal to the instruction fetch stage is an instruction register. At the start of an instruction cycle, the fetch unit sends out the PC and fetches the instruction from the instruction memory (i.e. instruction cache). The fetched instruction then gets loaded into the instruction register, which gets passed to the decode stage. It then increments the PC + 4 and stores it in the Next Program Counter (NPC). The actual PC on the next instruction is determined by the output of the ALU operation in the case of branching or conditional operations.

Instruction Decode

The decode stage breaks down the fetched instruction into its constituent parts. It extracts the opcode, function codes, source register addresses (rs1/rs2), destination register address (rd), and immediate values if present. The register file is accessed during this stage to read the values of the source registers specified by rs1 and rs2. The values obtained by rs1, and rs2 are piped directly into the ALU unit for operating on. For instructions requiring immediate values, the Sign Extension unit determines whether sign extension is needed based on the instruction type.

Execute/ALU Stage

The execute stage contains the Arithmetic Logic Unit (ALU) which performs the actual computation or operation specified by the instruction. The ALU receives its operands either from the register file (rs1/rs2) or from the immediate value (Imm) generated in the decode stage. Operations include arithmetic calculations (add, subtract), logical operations (AND, OR, XOR), shifts, and comparisons for branch instructions. For branch instructions, the ALU computes the target address and determines whether the branch condition is met. The result is then passed to the Conn unit which decides between the branch target and PC+4 for the next instruction address.

Memory Access

During the Memory Access stage, load and store operations are handled through the Data Memory interface (i.e. Data Cache). For load instructions, the ALU result from the execute stage is used as the memory address to fetch data from memory. For store instructions, both the ALU result (address) and the data to be stored (from rs2) are used. The Load/Store unit controls the memory operation and handles the data transfer. For non-memory instructions, this stage simply passes the ALU result through to the next stage. The Data Memory unit is isolated from the Instruction Memory to simplify the memory management of the pipeline, and prevent memory faults or hazards.

Write Back

The Write Back stage is the final stage of the pipeline where results are written back to the register file. The write-back data can come from either the ALU result (for arithmetic/logical operations) or from the data memory (for load operations). The destination register (rd) specified in the instruction receives the result.

III. Design And Testing Methodology

Designing a RISC-V processor from the ground up is a complex and iterative task. Very rarely in industry will you be building a design from scratch, so our group took a top-down approach in creating our processor core. This methodology involved starting with a high-level system view, like that shown in Fig 2, and breaking it down into small manageable units. This decomposition of units led to our testing strategy.

Unit Testing Approach

Our testing framework centered around independent verification of each SystemVerilog module before integration to our top-level design. We drew inspiration from the VUnit Framework's philosophy by "testing early and often" [4]. Each functional unit - from the ALU to the register file- underwent individual testing early on in our development cycle. While we didn't use an official unit testing framework, we did utilize the key principles of:

1. Early bug detection through module-level testing
2. Independent module bring-up and development
3. Systematic integration testing following unit verification

To support this methodology, we developed a modular build system that allows both isolated unit testing and full system builds. This type of setup was designed so that a particular bug could be reduced to a single module. Once this module is identified, an isolated unit test could be run to investigate further. In the subsequent sections, we will cover, in greater detail, the steps we took to verify each pipeline stage.

IV. Testing and Verification

Instruction Fetch:

The Instruction fetch unit was systematically verified through targeted scenarios ensuring correctness and readability. It has reset behaviour that validates proper initialization of Program Counter, Next Program Counter(NPC) and instructions upon reset. The testbench consists of three interfaces that are SystemVerilog constructs and these interfaces facilitate signal exchange between the module and its connected components ensuring modular and maintainable testing. The testbench operates with the clock signal toggling every 5 time units generated using an initial block.

When reset signal is deasserted, the module initializes its output including the PC, NPC and fetched instruction to default values. The testbench validates the instruction-fetching mechanism by enabling read enable signal, setting a conditional program counter(condpc) and writing a predefined instruction to memory.

```
// Test vectors
      // Test 1: Reset check
      $display("------------Test 1: Reset Check------------");
        #10 rst_n = 0;
      display();
        #10 rst_n = 1;
        display();

        // Test 2: Simple instruction fetch
        $display("------------Test 2: Simple Instruction Fetch------------");
        fetch_if.read_enable = 1;
        fetch_if.condpc = 32'h0;
        fetch_if.write_instruction = 32'h00000093;
        display();


        // Test 3: Next instruction
        $display("------------Test 3: Next Instruction------------");
        fetch_if.condpc = 32'h4;
        fetch_if.write_instruction = 32'h00500113;  // addi x2, x0, 5
        display();
        #10;
```

Fig 3: Fetch Test Cases

The initial block in the testbench provides a structured sequence of test scenarios to validate the instruction fetch module's functionality. A clock signal is generated to synchronize operations and initial conditions are set with reset enabled, instructions read disabled and counters initialized. The first test verifies the reset functionality by toggling the reset signal and checking if the module initialized correctly. Subsequent tests simulate instruction fetch operations at different memory locations by enabling reads, updating the program counter and writing specific instructions, including arithmetic operations. The behaviour is further tested by disabling reads to confirm that previously fetched values are retained and re-enabling reads to fetch new instructions. Throughout the simulation, a display task monitors and prints key signal states to verify functionality, concluding with a message marking the test completion.

```
# ------------Test 1: Reset Check------------
# Time=50 rst_n=0 read_enable=0 pc=00000000 npc=00000004 instruction=00000013
# Time=100 rst_n=1 read_enable=0 pc=00000000 npc=00000004 instruction=00000013
# ------------Test 2: Simple Instruction Fetch------------
# Time=140 rst_n=1 read_enable=1 pc=00000000 npc=00000004 instruction=00000093
# ------------Test 3: Next Instruction------------
# Time=180 rst_n=1 read_enable=1 pc=00000004 npc=00000008 instruction=00500113
# ------------Test 4: Disable Read------------
# Time=230 rst_n=1 read_enable=0 pc=00000004 npc=00000008 instruction=00500113
# ------------Test 5: Re-enable Read------------
# Time=280 rst_n=1 read_enable=1 pc=00000008 npc=0000000c instruction=00310233
# ----------------------------------------Tests completed----------------------------------------
```

Fig 4: Fetch Test Output

The expected result is the test result that confirms the correct behaviour of the fetch module. During the reset test, the module initializes the PC to 00000000 and NPC to 00000004 and loads the default instruction that is 00000013. When read enable is 1, the module fetches instructions based on the program counter(00000093 at PC=00000000 and 00500113 at PC=00000004). When read_enable halts instruction updates, retaining the current instruction(00500113). Re-enabling it allows fetching a new instruction (00310233 at PC=00000008) with appropriate PC and NPC updates. These tests validate that the module initializes correctly, fetches instructions accurately, halts updates when required and updates the PC and NPC as expected.

Instruction Decode:

The decode unit was tested with a set of vectors, which can be thought of as individual instructions. In that sense, the vectors acted as a driver for the test bench where each entry in the vector would be input into the decode unit and then was tested to ensure the output was the expected segmented instruction and contained the correct register data. Below is an excerpt of one of our test vectors:

```
// Test vectors
      class DecodeTests;
      static test_instruction_t test_vectors[] = '{
            // R-type: ADD x1, x2, x3
            '{
            test_name: "R-type ADD",
            instruction: 32'h003100b3,  // ADD x1, x2, x3
            expected_pc: 32'h1000,
            expected_opcode: OPCODE_REG_REG,
            expected_rd: 5'h1,
            expected_rs1: 5'h2,
            expected_rs2: 5'h3,
            expected_funct3: 3'h0,
            expected_funct7: 7'h00,
            expected_imm: 32'h0,
            reg_a_value: 32'h5,
            reg_b_value: 32'h3
            },

            // additional vectors ...
```

Fig 5: Decode Test Vector

Therefore, it was trivial to add additional tests and scale this to additional cases. Above is an example of an ADD instruction, where the decode stage is responsible for partitioning the instruction (0x0031_00B3) into its constituent parts as should in Fig 1. The expected output values were initially generated using Claude AI assistant [5] and verified through manual review to ensure correctness. These expected values were ultimately used to verify that they matched the actual output of the decode stage. The complete results from this test are presented below.

```
# Starting Decode Stage Tests
# Test R-type ADD PASSED
# Test I-type ADDI PASSED
# Test S-type SW PASSED
# Test B-type BEQ PASSED
# Test U-type LUI PASSED
# Test J-type JAL PASSED
#
# Test Summary:
# Tests run: 6
# Errors: 0
```

As a specific example, consider the ADD instruction test vector (0x0031_00B3) shown in the waveform below. The decode unit successfully partitioned this instruction into its constituent fields, with register specifiers rs1 and rs2 correctly identified as x1 and x2, containing values 5 and 3 respectively. The waveform demonstrates proper opcode extraction, confirming the R-type instruction. Lastly, see Appendix B, which shows this decoded instruction propagated through the pipeline, resulting in the ALU computing the expected sum of 8.

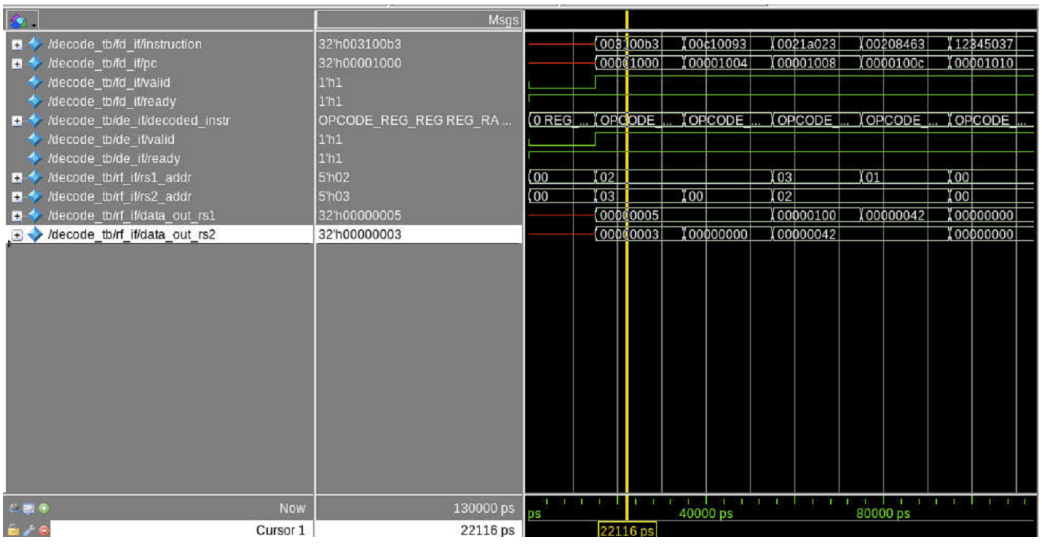


Fig 7: Decode Stage Waveform

Execute/ ALU Stage:

The ALU unit was tested using pre-made instructions with expected outcomes that target each of the ALU commands within the RV32I instruction set. Here an an excerpt from the testbench:

```
//TESTING REG_REG

    // Test 1: ADD operation
    de_if.decoded_instr.opcode = OPCODE_REG_REG;
    de_if.decoded_instr.funct3 = F3_ADD_SUB;
    de_if.decoded_instr.funct7 = F7_ADD_SRL; // Use F7_ADD_SRL for ADD
    de_if.decoded_instr.reg_A = 32'h0000_0005;
    de_if.decoded_instr.reg_B = 32'h0000_0003;
    de_if.valid = 1'b1;

    #10;
    de_if.valid = 1'b0;

    // Check results
    #10;
    if (em_if.alu_result !== 32'h0000_0008)
      $display("Test 1 (ADD) failed: result = %h", em_if.alu_result);
    else
      $display("Test 1 (ADD) passed");
```

Fig 8: ALU Test Set

The ALU module was tested with a comprehensive suite of test cases targeting key arithmetic, logical, and special operations to validate the functionality of the Arithmetic Logic Unit. The test cases are as follows:

Register-to-Register Operations

1. **ADD Operation (Test 1):** Adds two positive numbers (5 + 3). Validates that the ALU correctly performs addition, with the expected result of 8.
2. **SUB Operation (Test 2):** Subtracts 3 from 8. Confirms the ALU handles subtraction correctly, expecting the result 5.
3. **OR Operation (Test 3):** Performs a logical OR on 0xA and 0x5. Verifies that bitwise OR functionality outputs 0xF.
4. **AND Operation (Test 4):** Performs a logical AND on 0xA and 0x5. Ensures the result is 0x0, confirming proper implementation.
5. **XOR Operation (Test 5):** Applies XOR to 0xA and 0x5. Confirms correct bitwise XOR functionality with the result 0xF.
6. **SLT Operation (Signed, Test 6):** Compares -11 and 5, expecting the result 1. Validates signed comparison.
7. **SLT Operation (Signed Reverse, Test 7):** Compares 5 and -11, expecting the result 0. Verifies reverse signed comparison.
8. **SLTU Operation (Unsigned, Test 8):** Compares 3 and 5. Ensures the ALU handles unsigned less-than comparison, expecting 1.
9. **SLTU Operation (Reverse Unsigned, Test 9):** Compares unsigned values 0xFFFF_FFFF and 1, expecting 0. Tests larger-than cases.

Register-to-Immediate Operations

10. **ADDI Operation (Test 10):** Adds an immediate value (5) to 0. Validates addition with immediate values, expecting 5.
11. **ORI Operation (Test 11):** Performs logical OR with an immediate (0xA OR 0x5). Ensures correctness with the result 0xF.
12. **ANDI Operation (Test 12):** Applies logical AND with an immediate value (0xFFF). Verifies the result is 0xFFF.
13. **XORI Operation (Test 13):** Applies XOR with an immediate (0xA XOR 0x6). Ensures the result is 0xC.
14. **SLLI Operation (Test 14):** Performs a left shift on 0xF by 2 bits, expecting 0x3C. Tests immediate-based shift-left.
15. **SRLI Operation (Test 15):** Shifts 0xF right logically by 2 bits, expecting 0x3. Verifies proper logical shift-right.
16. **SRAI Operation (Test 16):** Performs an arithmetic right shift on -16 (0xFFFF_FFF0) by 2 bits, expecting 0xFFFF_FFFC.

Special Cases

17. **ADD with Overflow (Test 17):** Tests addition overflow by adding 0x7FFF_FFFF and 1, expecting 0x8000_0000. Confirms overflow behavior.
18. **SUB with Underflow (Test 18):** Tests subtraction underflow by subtracting 1 from 0x8000_0000. Expects 0x7FFF_FFFF.
19. **AND with Zero (Test 19):** Evaluates logical AND between all bits set and all bits cleared. Expects 0x0.
20. **Load Effective Address (Test 20):** Computes load effective address with base 0x1000 and offset 0x10, expecting 0x1010.
21. **Store Effective Address (Test 21):** Computes store effective address with base 0x2000 and offset -16, expecting 0x1FF0.

Additional Register-to-Immediate Tests

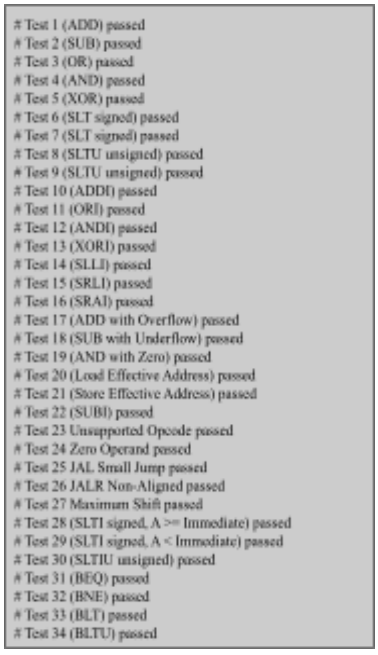
22. **SUBI Operation (Test 22):** Performs subtraction with an immediate, subtracting 2 from 5. Expects 3.
23. **Unsupported Opcode (Test 23):** Tests an invalid opcode (0xFF). Ensures the ALU outputs 0 for unsupported operations.
24. **Zero Operand (Test 24):** Tests addition with one operand as zero (0 + 5). Verifies the result is 5.

Jump and Branch Tests

25. **JAL (Test 25):** Computes a jump address with base PC 0x1000 and offset 4. Expects the result 0x1004.
26. **JALR Misaligned (Test 26):** Tests jump-and-link register with a misaligned base (0x1001) and offset 4. Expects 0x1004.
27. **Maximum Shift (Test 27):** Tests boundary condition with a maximum shift on 0x8000_0000, expecting 0x1.

Advanced Tests

28. **SLTI (Test 28):** Tests signed comparison of a register (5) against a negative immediate (-11). Expects 0 since 5 >= -11.
29. **SLTI with Less Than (Test 29):** Compares -11 against 5 using SLTI. Ensures the result is 1 since -11 < 5.
30. **SLTIU (Test 30):** Tests unsigned comparison of 3 and 5 with an immediate. Verifies the result 1.
31. **BEQ (Test 31):** Verifies branch-if-equal logic by comparing 1 and 1. Ensures `zero` flag is set (1).
32. **BNE (Test 32):** Tests branch-if-not-equal with 0xA and 0xB. Confirms `zero` flag is set (1).
33. **BLT (Signed, Test 33):** Evaluates signed branch-if-less-than for -11 and 5. Ensures the `zero` flag is set (1).
34. **BLTU (Unsigned, Test 34):** Tests unsigned branch-if-less-than for 3 and 5. Expects the `zero` flag to be 1.



Memory Access:

The Memory Access unit handles critical load/store operations and conditional Program Counter (PC) updates.

```
// Test scenarios
      // Test 1: Reset
       $display("Test 1: Reset Check");
       #10 rst_n = 0;
       display();
       #10 rst_n = 1;

       // Test 2: Write to memory
       $display("Test 2: Memory Write");
       #10;
       mem_if.WE = 1;
       mem_if.RE = 0;
       e_m_if.alu_result = 32'h4;  // Memory address
       mem_if.REG_B = 32'hABCD1234;  // Data to write
       display();
       #10;
       mem_if.WE = 0;


            // more test scenarios..
```

Fig 9: Memory Access Unit Test Cases

The Memory Access unit was tested with a set of seven test cases each targeting a specific aspect of the Memory Access unit and validates the functionality of the Memory Access module. The seven test cases are as follows:

1. **Reset Operation (Test 1):** This test initializes the module, ensuring all components are correctly reset to their default state. By toggling the reset signal, the test verifies that the Memory Access module can be safely initialized, resetting all internal states and preparing the module for subsequent operations.
2. **Memory Write Operation (Test 2):** The second test validates the module's write functionality by storing a specific data value (0xABCD1234) at a designated memory address (0x4). It sets the Write Enable(in case of Store operations) signal and provides both the target address and data to be written, confirming the module's ability to perform basic memory write operations.
3. **Memory Read Operation (Test 3)**: Immediately following the write operation, this test checks the read(Load) functionality. By setting the Read Enable(in case of Load operation) signal and using the same address from the previous write, the test verifies that the data can be correctly read back from memory, ensuring the write-read cycle operates as expected.
4. **Alternative Memory Write (Test 4)**: This test demonstrates the module's ability to write to a different memory location. Writing a new data value (0x87654321) to a different address (0x8) confirms the module's flexibility in handling multiple write operations to various memory locations.
5. **Alternative Memory Read (Test 5)**: Corresponding to the previous write, this test reads from the new memory address, verifying that the different write operation was successful and that the module can retrieve data from multiple distinct memory locations.
6. **Conditional Branch (Test 6)**: This test focuses on the conditional Program Counter (PC) update for branch instructions. By setting the zero flag and providing a branch target address, the test checks the module's ability to modify the PC based on condition flags, simulating a branch taken scenario.
7. **Jump and Link (JAL) (Test 7)**: The final test examines the PC update mechanism for Jump and Link (JAL) instructions. This scenario tests the module's ability to handle unconditional jump instructions, verifying that the PC can be updated to a specific target address regardless of condition flags.

Finally, the output of the test was manually reviewed with the expected results.

```
// Test results


Test 1: Reset Check
Time=50 rst_n=0 WE=0 RE=0 Addr=00000000 Data=00000000 ReadData=00000000 LMD=00000000 condpc=00000000

Test 2: Memory Write
Time=110 rst_n=1 WE=1 RE=0 Addr=00000004 Data=abcd1234 ReadData=00000000 LMD=00000000 condpc=00000000

Test 3: Memory Read
Time=170 rst_n=1 WE=0 RE=1 Addr=00000004 Data=abcd1234 ReadData=abcd1234 LMD=abcd1234 condpc=00000000

Test 4: Write Different Location
Time=220 rst_n=1 WE=1 RE=0 Addr=00000008 Data=87654321 ReadData=abcd1234 LMD=abcd1234 condpc=00000000

Test 5: Read Different Location
Time=270 rst_n=1 WE=0 RE=1 Addr=00000008 Data=87654321 ReadData=87654321 LMD=87654321 condpc=00000000

Test 6: Conditional PC Test
Time=320 rst_n=1 WE=0 RE=1 Addr=00000100 Data=87654321 ReadData=00000000 LMD=87654321 condpc=00000100 //Zero flag set
Time=370 rst_n=1 WE=0 RE=1 Addr=00000100 Data=87654321 ReadData=00000000 LMD=87654321 condpc=00000200 //Zero flag not set

Test 7: JAL PC Test
Time=420 rst_n=1 WE=0 RE=1 Addr=00000300 Data=87654321 ReadData=00000000 LMD=00000200 condpc=00000300
Time=470 rst_n=1 WE=0 RE=1 Addr=00000300 Data=87654321 ReadData=00000000 LMD=00000200 condpc=00000300

Tests completed
```

Fig 10: Memory Access Test results

Memory Writeback:

The Writeback unit is the last stage in the RISC-V pipeline, the testbench is comprised of four comprehensive test cases that validate it's functionality:

1. **ALU Result Writeback (Test Case 1):** This test verifies the writeback module's main functionality of writing ALU executed results to the register file. By enabling the write enable signal to the writeback unit and specifying a destination register (x1 in this case), the test checks that the ALU result is correctly propagated to the register file when processing standard register to register operations.
2. **Load Memory Data Writeback (Test Case 2)**: Focusing on memory load instructions, this test case confirms the module's ability to write loaded memory data into the register file. By changing the operation code to a load instruction and targeting a different destination register (x2), the test ensures that memory loaded values are correctly written back for the appropriate instruction.
3. **Write Protection for x0 Register (Test Case 3)**: A critical feature of the RISC-V architecture is that the x0 register will always be zero. This test explicitly attempts to write a non-zero value to x0, verifying that the module prevents any modifications to this hardwired zero register, in spite of the input value.
4. **Write Enable Control (Test Case 4)**: The final test examines the writeback mechanism by disabling the write enable signal. This test confirms that no register writes occur when the write enable signal is low, ensuring proper control over register file modifications.

```
// Test scenarios
      // Test case 1: ALU result writeback
       rf_write_if.write_en = 1;
       mw_if.decoded_instr.rd = register_name_t'(5'd1);
       mw_if.opcode = OPCODE_REG_REG;
       #10;

       // Check if ALU result is written to rd_data
       if (rf_write_if.rd_data !== mw_if.alu_result) begin
           $display("Test case 1 failed: Expected rd_data = %h, got %h",
                    mw_if.alu_result, rf_write_if.rd_data);
       end else begin
           $display("Test case 1 passed: rd_data = %h", rf_write_if.rd_data);
       end

       // Test case 2: Load memory data writeback
       rf_write_if.write_en = 1;
       mw_if.decoded_instr.rd = register_name_t'(5'd2);
       mw_if.opcode = OPCODE_LOAD;
       #10;

       // Check if load data is written to rd_data
       if (rf_write_if.rd_data !== mw_if.LMD) begin
           $display("Test case 2 failed: Expected rd_data = %h, got %h",
                    mw_if.LMD, rf_write_if.rd_data);
       end else begin
           $display("Test case 2 passed: rd_data = %h", rf_write_if.rd_data);
       end



            // more test scenarios..
```

Fig 11: Writeback Unit Test Cases

By precomputing the expected results, the test cases were manually reviewed for verification. This approach allows for immediate detection and diagnosis of any potential issues in the writeback stage's implementation, providing confidence in the module's correct behavior within the larger processor design.

```
Test case 1 passed: rd_data = 12345678
Test case 2 passed: rd_data = 87654321
Test case 3 passed: x0 remains zero
Test case 4 passed: No write when write_en is low
```

Fig 12: Writeback Unit Test Results

Top Module:

The top level module simply instantiates all of the individual units, since we have utilized interfaces to connect and handshake among our individual stages in the pipeline, the port connections for the top module was essentially connecting the interfaces together after each module instantiation.

Our testbench for the top module uses the modular interface signals to drive the control signals and instruction to the Design Under Test (DUT). A structured test data type encapsulates the necessary fields for testing, including the instruction, expected opcode, operand registers, immediate values, and expected results. The testbench iterates through

an array of test vectors, each representing a specific RISC-V instruction type such as R-type, I-type, Load, Store, Jump, or Branch.

```
// Test Data

  typedef struct {
     string test_name;
     logic [31:0] instruction;
     logic [31:0] pc;
     opcode_t expected_opcode;
     logic [4:0] expected_rd;
     logic [4:0] expected_rs1;
     logic [4:0] expected_rs2;
     logic [31:0] reg_a_value;
     logic [31:0] reg_b_value;
     logic [31:0] expected_result;
     logic register_write;
  } test_case_t;
```

Fig 13: Test Case Structure Data type

During each test, the processor is stimulated with the test vector inputs, including instructions and register values, while control signals like memory read and write, enable proper execution. The outputs, such as decoded opcodes, ALU results, and memory data, are compared against expected values, and mismatches are logged as errors. Specific validation is performed for each instruction type—for example, R-type instructions verify ALU operations, Load/Store instructions validate memory access, and Branch/Jump instructions confirm PC updates. The testbench provides a summary of the test results, indicating the total number of tests executed, errors encountered, and the pass/fail status of each case. While comprehensive in validating multiple instruction types and pipeline stages, the testbench currently supports a limited subset of RISC-V instructions and assumes ideal memory behavior without hazards.

```
// Test results
# Starting RISC-V Pipeline Tests...
# Running Test: R-type ADD
# Test R-type ADD PASSED
# Running Test: I-type ADDI
# Test I-type ADDI PASSED
# Running Test: Store-type
# Test Store-type PASSED
# Running Test: Load-type
# Test Load-type PASSED
# Running Test: Jump-type
# Test Jump-type PASSED
# Running Test: Branch-type
# Test Branch-type PASSED
#
# Test Summary:
# Tests run: 6
# Errors: 0
```

Fig 14: Top Module Results

The modular, interface-based design makes the testbench scalable and efficient for verification.

V. Future Work

Our work has demonstrated the successful implementation and basic validation of a single-cycle RISC-V processor core. However, comprehensive verification of a processor pipeline involves an extensive combination of possible inputs and outcomes that warrant further exploration. Future work should focus on expanding test coverage to verify all data paths and handle corner cases systematically.

Furthermore, our current implementation is non-pipelined, requiring 5 clock cycles per instruction. A key enhancement would be implementing pipelining to achieve an ideal throughput of one instruction per clock cycle. This improvement would require several additional design units:

- Branch prediction to minimize control hazard penalties
- Hazard detection to identify and manage data dependencies
- Data forwarding to efficiently handle pipeline hazards

These additions would significantly improve the performance of our processor core and throughput; not to mention most modern processors utilize pipelining in their designs.

VI. Conclusion

In this project, we've successfully implemented functional units for each stage of the classic RISC-V pipeline - Fetch, Decode, Execute, Memory Access, and Memory Writeback. We developed a comprehensive testing framework and unit testing build system using Makefile. Our RISC-V core implementation includes all essential components such as an ALU, register file, instruction cache, data cache, and memory interfaces.

Throughout this project, we gained valuable experience in computer architecture and digital design. One of the major challenges was managing the complexity and scaling the design to achieve a functional prototype within our limited timeframe. Critical to achieving our goals was the use of systematic unit testing, which allowed us to design and verify each pipeline stage in isolation. This approach proved highly effective, as we encountered minimal issues during top-level integration, and verification of the complete system required surprisingly little time and effort.

The design and implementation of a RISC-V processor is a complex undertaking, and as discussed earlier, the verification process should be exhaustive to validate all possible operations. While our design successfully models the RISC-V ISA, there remains work to be done on the verification front. Additionally, future design work should focus on implementing a pipelined version of our current architecture to improve performance and throughput.

Acknowledgment

The authors acknowledge the assistance and use of Claude (Anthropic) in generating general test vectors and helping in writing this paper (particularly with formatting and grammar checking) [5].

References

[1] - Patterson, David & Sequin, Carlo. (1998). RISC I: a reduced instruction set VLSI computer. Proceedings of the 8th Annual Symposium on Computer Architecture. 216-230. 10.1145/285930.285981.

[2] - J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Cambridge, MA: Morgan Kaufmann Publishers, 2019.

[3] - "The RISC-V Instruction Set Manual Volume I: Unprivileged ISA Version 20240411," RISC-V International, May 2024. [Online]. Available: https://riscv.org/specifications/

[4] - https://github.com/VUnit/vunit

[5] - Anthropic. (2024). Claude [Large language model]. Retrieved from anthropic.com

Appendixes

Appendix A: RISC-V RV32I Full Instruction Set

Instruction Name	Type	Opcode	funct3	funct7	Immediate	Registers	Description
ADD	R	110011	0	0	N/A	rs1, rs2, rd	Add two registers.
SUB	R	110011	0	100000	N/A	rs1, rs2, rd	Subtract two registers.
AND	R	110011	111	0	N/A	rs1, rs2, rd	Bitwise AND.
OR	R	110011	110	0	N/A	rs1, rs2, rd	Bitwise OR.
XOR	R	110011	100	0	N/A	rs1, rs2, rd	Bitwise XOR.
SLT	R	110011	10	0	N/A	rs1, rs2, rd	Set if less than (signed).
SLTU	R	110011	11	0	N/A	rs1, rs2, rd	Set if less than (unsigned).
SLL	R	110011	1	0	N/A	rs1, rs2, rd	Shift left logical.
SRL	R	110011	101	0	N/A	rs1, rs2, rd	Shift right logical.
SRA	R	110011	101	100000	N/A	rs1, rs2, rd	Shift right arithmetic.
ADDI	I	10011	0	N/A	imm[11:0]	rs1, rd	Add immediate.
ANDI	I	10011	111	N/A	imm[11:0]	rs1, rd	Bitwise AND immediate.
ORI	I	10011	110	N/A	imm[11:0]	rs1, rd	Bitwise OR immediate.
XORI	I	10011	100	N/A	imm[11:0]	rs1, rd	Bitwise XOR immediate.
SLTI	I	10011	10	N/A	imm[11:0]	rs1, rd	Set less than immediate (signed).
SLTIU	I	10011	11	N/A	imm[11:0]	rs1, rd	Set less than immediate (unsigned).
SLLI	I	10011	1	0	shamt[4:0]	rs1, rd	Shift left logical immediate.
SRLI	I	10011	101	0	shamt[4:0]	rs1, rd	Shift right logical immediate.
SRAI	I	10011	101	100000	shamt[4:0]	rs1, rd	Shift right arithmetic immediate.
LB	I	11	0	N/A	imm[11:0]	rs1, rd	Load byte.
LH	I	11	1	N/A	imm[11:0]	rs1, rd	Load halfword.
LW	I	11	10	N/A	imm[11:0]	rs1, rd	Load word.
SB	S	100011	0	N/A	imm[11:5], imm[4:0]	rs1, rs2	Store byte.
SH	S	100011	1	N/A	imm[11:5], imm[4:0]	rs1, rs2	Store halfword.
SW	S	100011	10	N/A	imm[11:5], imm[4:0]	rs1, rs2	Store word.
BEQ	B	1100011	0	N/A	`imm[12	10:5	4:1
BNE	B	1100011	1	N/A	`imm[12	10:5	4:1
BLT	B	1100011	100	N/A	`imm[12	10:5	4:1
BGE	B	1100011	101	N/A	`imm[12	10:5	4:1
BLTU	B	1100011	110	N/A	`imm[12	10:5	4:1

BGEU	B	1100011	111	N/A	`imm[12	10:5	4:1
JAL	J	1101111	N/A	N/A	`imm[20	10:1	11
JALR	I	1100111	0	N/A	imm[11:0]	rs1, rd	Jump and link register.
LUI	U	110111	N/A	N/A	imm[31:12]	rd	Load upper immediate.
AUIPC	U	10111	N/A	N/A	imm[31:12]	rd	Add upper immediate to PC.

RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
fm | pred | succ		rs1	000	rd	0001111	FENCE
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK

Appendix B: Top-level waveform