Mini Project 4 Report

Xavier Nishikawa, Jack Wei, Hong Yi Zhang
Github: https://github.com/We1chJ/iceBlinkPico/tree/xmp4

Design Overview

This report details the design of an unpipelined, multicycle 32-bit RISC-V integer microprocessor featuring a Harvard architecture. The processor separates instruction and data memories physically while mapping them to a unified 32-bit address space. We utilized the supplied memory module, which implements 8 kB of physical memory (4 kB of data memory from 0x0000 to 0x0FFF and 4 kB of instruction memory from 0x1000 to 0x1FFF) and handles the memory-mapped I/O logic.

The design supports interaction with hardware peripherals accessible through the data memory ports at high addresses. These include 8-bit PWM generators for the user/RGB LEDs and two hardware timers (milliseconds/microseconds). While the PWM and timer logic resides within the provided memory module, our processor's datapath and control unit were specifically designed to support the standard load (lw) and store (sw) instructions required to interface with these peripherals. The processor implements the base RV32I instruction set (excluding ecall, ebreak, csrrw, csrrs, csrrc, csrrwi, csrrsi, and csrrci) and is capable of executing complex assembly programs. We verified the correct operation of our microprocessor using Icarus Verilog for simulation and GTKWave for waveform analysis.

Circuit Architecture

The design of our RV32I processor was built off of Harris & Harris's textbook. Due to the project requirement of adapting to the given memory modules, modifications are made to the circuit diagrams and the FSM states shown below.

As shown in Figure 1 below, while we kept most of the designs from the textbook, like Figure 2, we removed some of the signals out of the Control Unit, such as AdrSrc and its 3-to-1 mux, because the memory module provided takes in both instruction address and memory address at the same time.

Similarly, as shown in Figure 3, the corresponding FSM state flowcharts in the Control Unit are also modified to perform both instructions included from the book and additional instructions we added like BLT, LUI, AUIPC, etc.

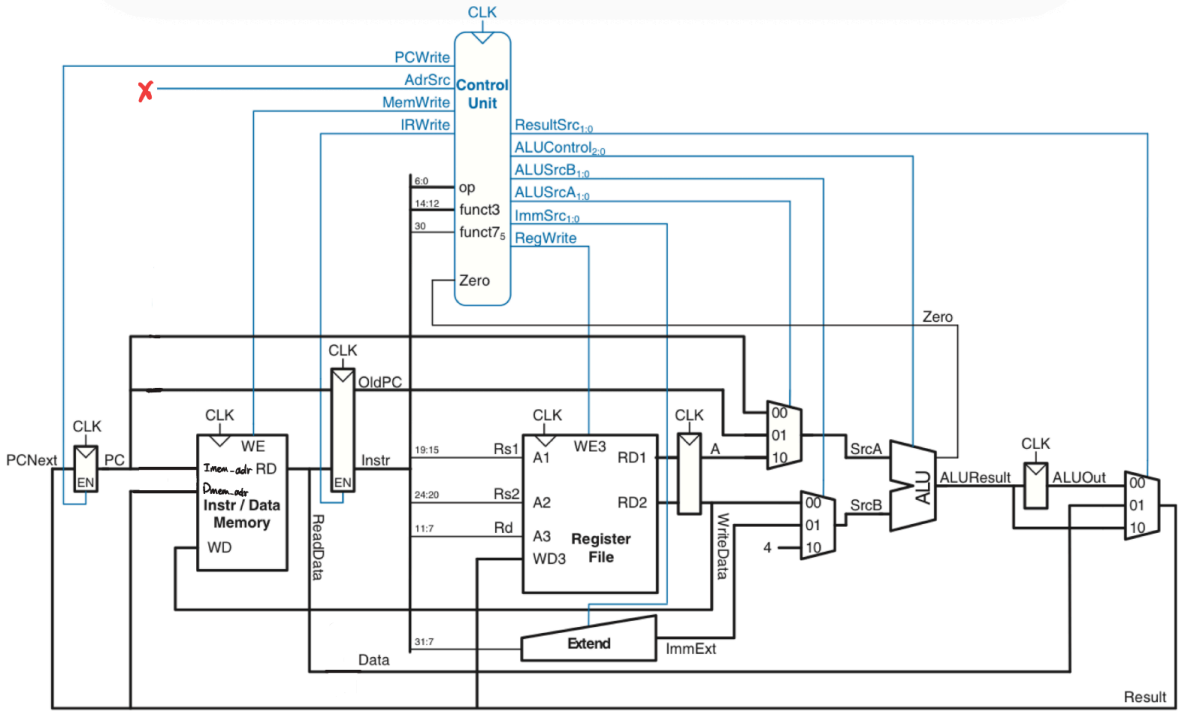

Figure 1: Modified Complete RISC-V Multicycle Logic Circuit

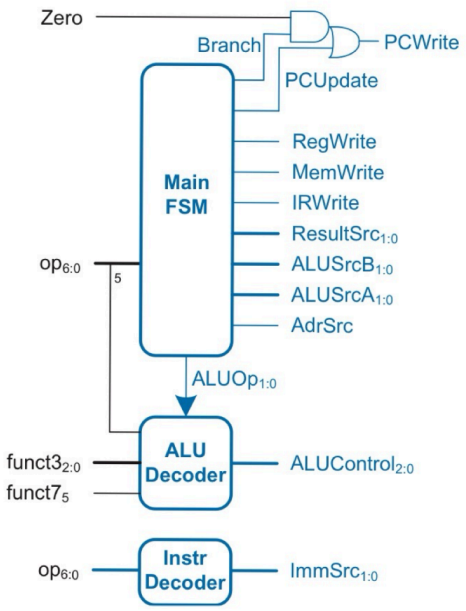

Figure 2: Internal View of Signal Computation in Control Unit

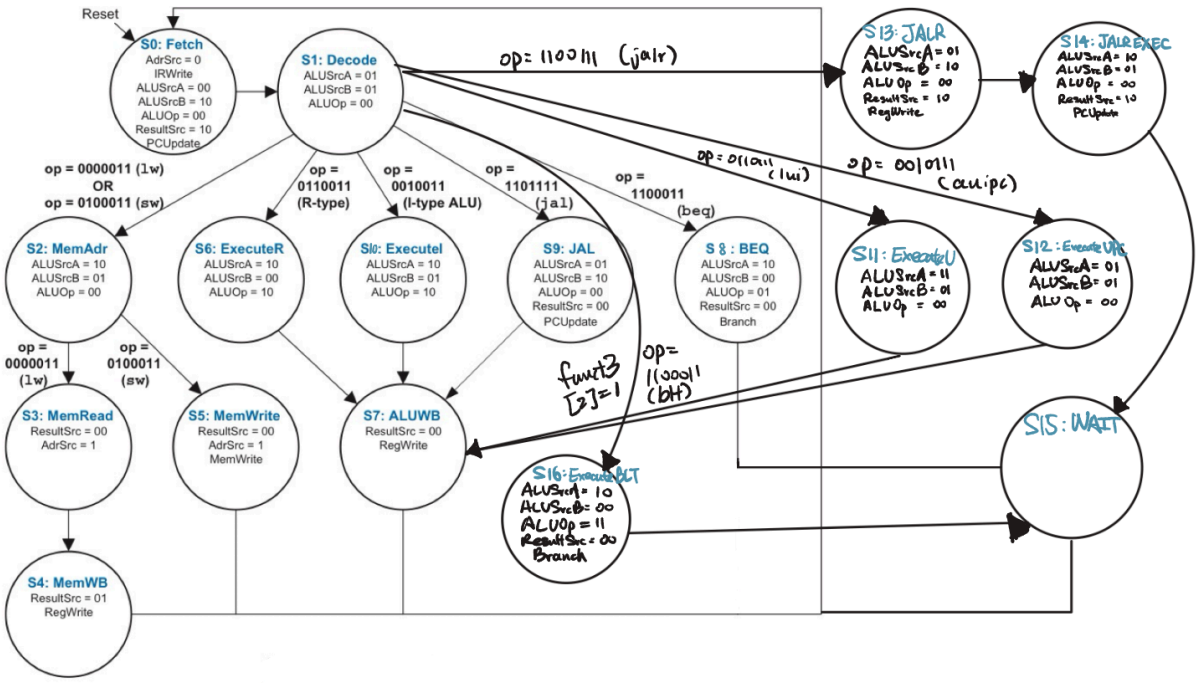

Figure 3: Modified FSM States Flowchart

The design is modularized into four main SystemVerilog components: the top module, the ControlUnit, the datapath components (ALU, Register_File, PC), and the provided memory module.

The top module acts as the wrapper that interconnects all system components. It instantiates the Control Unit and datapath modules, explicitly wiring them together to form the complete processor. For example, it connects the Program Counter's output to the instruction memory address input and routes the memory's data output back to the instruction register. It also houses the critical multiplexers (Mux_ALUSrcA, Mux3_to_1 for ALUSrcB, and ResultSrc) which steer data between the ALU, Register File, and Memory based on the control signals active during each instruction phase.

The Control Unit serves as the processor's finite state machine (FSM). It transitions through a standard multi-stage cycle for every instruction: FETCH, DECODE, etc. It handles complex control flow by implementing specific states for branches and jumps (e.g., BEQ, BLT, JALR) and includes wait states for any instructions that use an offset like JAL or BEQ to ensure proper timing for PC updates. The Control Unit generates all control signals (e.g., RegWrite, MemWrite, ALUControl) required to drive the datapath for each state.

The Datapath handles the actual data processing and storage:

ALU: Performs all arithmetic (ADD, SUB) and logical (AND, OR, XOR, comparator) operations. It also calculates effective addresses for memory access and generates the Zero flag used by the Control Unit to resolve conditional branches.

Register File: Contains the 32 general-purpose RISC-V registers. It supports dual asynchronous reads and a single synchronous write, allowing the processor to fetch two operands and store a result within the execution cycle.

Program Counter (PC): Maintains the current instruction address, updating synchronously based on sequential execution (+4) or branch/jump target calculations provided by the ALU.

The processor supports the conditional branch instructions BEQ, BLT, and BLTU. The BEQ waveform demonstrates equality comparison via subtraction, while BLT demonstrates signed comparison using the SLT operation.

Simulation Results

```
// Define state variable values
localparam [4:0] FETCH      = 5'd0;
localparam [4:0] DECODE     = 5'd1;
localparam [4:0] MEM_ADR    = 5'd2;
localparam [4:0] MEM_READ   = 5'd3;
localparam [4:0] MEM_WB     = 5'd4;
localparam [4:0] MEM_WRITE  = 5'd5;
localparam [4:0] EXECUTE_R  = 5'd6;
localparam [4:0] ALU_WB     = 5'd7;
localparam [4:0] BEQ        = 5'd8;
localparam [4:0] JAL        = 5'd9;
localparam [4:0] EXECUTE_I   = 5'd10;
localparam [4:0] EXECUTE_U   = 5'd11;
localparam [4:0] EXECUTE_UPC = 5'd12;
localparam [4:0] JALR        = 5'd13;
localparam [4:0] JALR_EXEC   = 5'd14;
localparam [4:0] WAIT        = 5'd15;
localparam [4:0] EXECUTE_BLT = 5'd16;
```

Figure 4: Encoding of the control unit's multicycle states. Each localparam assigns a unique 5-bit value to a named state (such as FETCH, DECODE, memory access, ALU execute, branch, and jump), which the control logic uses to step through instruction execution.

I-Type Instruction Execution (Arithmetic Immediate):

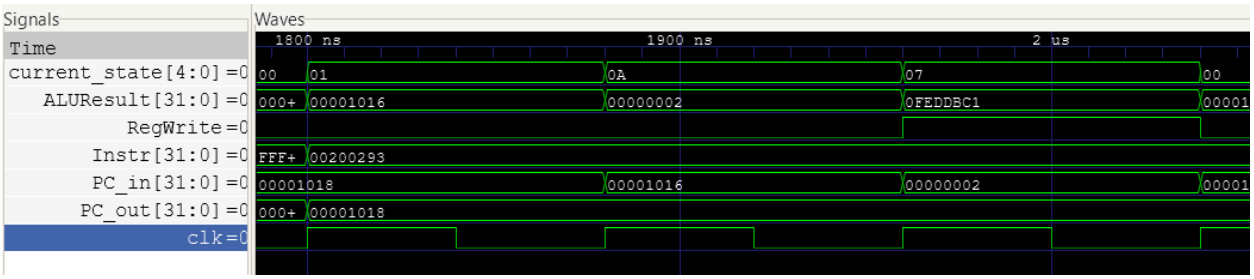

Figure 5: Simulation waveform detailing the execution of an addi instruction. The current_state signal cycles through Fetch (00), Decode (01), Execute Immediate (0A), and Writeback (07). The ALUResult is computed during the Execute phase, and RegWrite is asserted high precisely during the Writeback state (07) to store the result. The PC_out signal (showing 0x1018) remains stable throughout the instruction's execution before advancing to the next address.

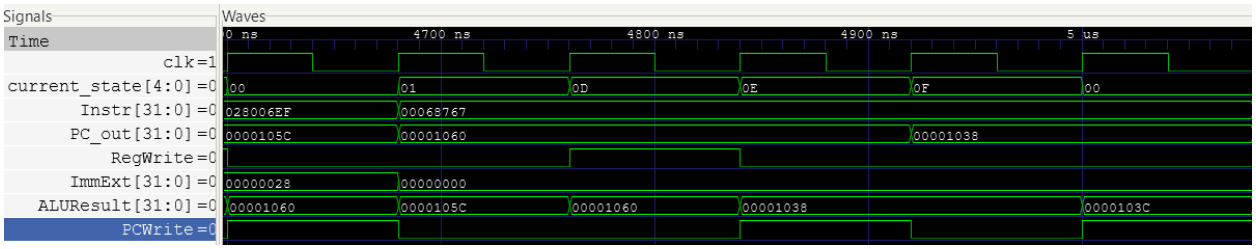

Figure 6: JALR Instruction Execution. The processor executes jalr x14, 0(x13), transitioning through JALR (0D) and JALR_EXEC (0E) states. During JALR, RegWrite asserts to store the return address (0x1060 = PC+4) in x14. During JALR_EXEC, the ALU computes the jump target (x13 + 0 = 0x1038) and PCWrite asserts. The PC updates to 0x1038 after the WAIT state, demonstrating the register-indirect jump behavior distinct from JAL's PC-relative jump.

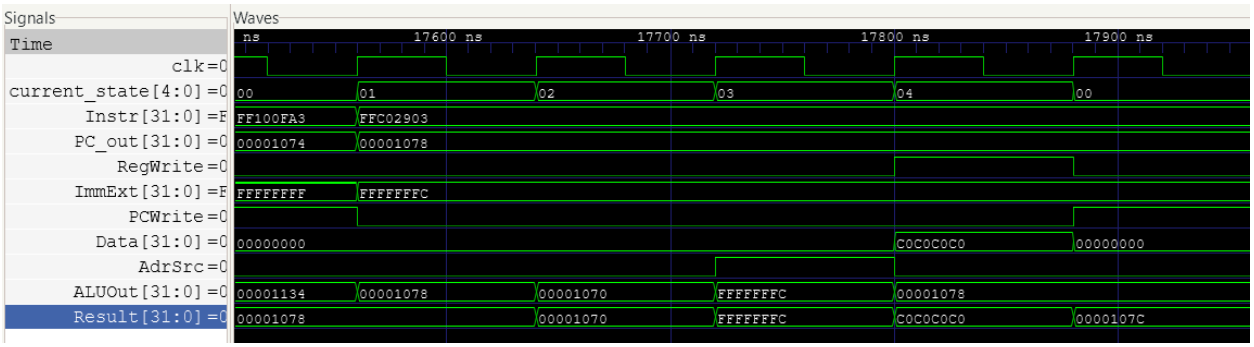

Figure 7: Load Word (LW) Instruction Execution. The processor executes lw x18, -4(x0), demonstrating the 5-state memory access cycle: FETCH (00), DECODE (01), MEM_ADR (02), MEM_READ (03), and MEM_WB (04). During MEM_ADR, the ALU computes the effective address (0xFFFFFFFC). AdrSrc asserts during MEM_READ to route this address to memory. The loaded data (0xC0C0C0C0) appears on the Data bus and is written to x18 when RegWrite asserts during MEM_WB.

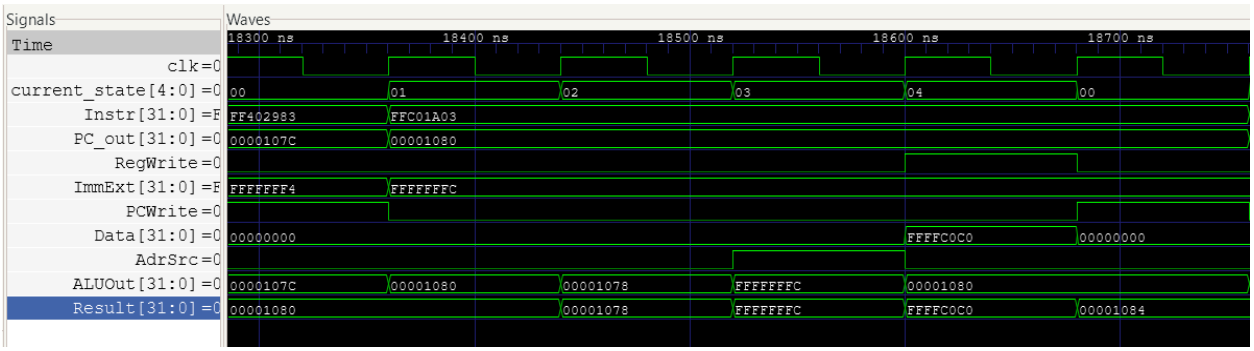

Figure 8: Load Half (LH) Instruction Execution. The processor executes lh x20, -4(x0), following the same 5-state memory access cycle as LW. The key distinction is in the loaded data: while the memory contains 0xC0C0C0C0, the LH instruction loads only the lower 16 bits (0xC0C0) and sign-extends it to 32 bits, resulting in 0xFFFFC0C0. This demonstrates the memory module's handling of sub-word loads based on the funct3 field (001 for LH). RegWrite asserts during MEM_WB (state 04) to store the sign-extended value in x20.

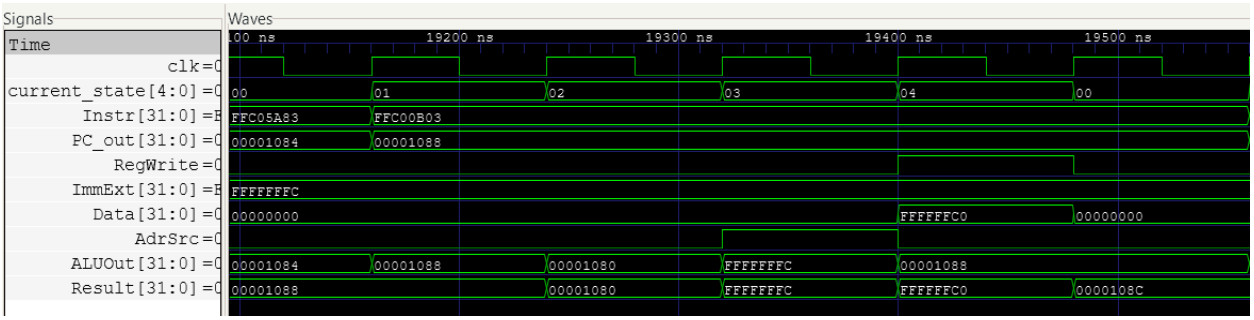

Figure 9: Load Byte (LB) Instruction Execution. The processor executes lb x22, -4(x0), loading a single byte (0xC0) from memory and sign-extending it to 32 bits (0xFFFFFFC0). Since bit 7 of the byte is 1, the upper 24 bits are filled with 1s. This demonstrates the most aggressive sign-extension among load instructions.

S-Type Instruction Execution (Memory Store):

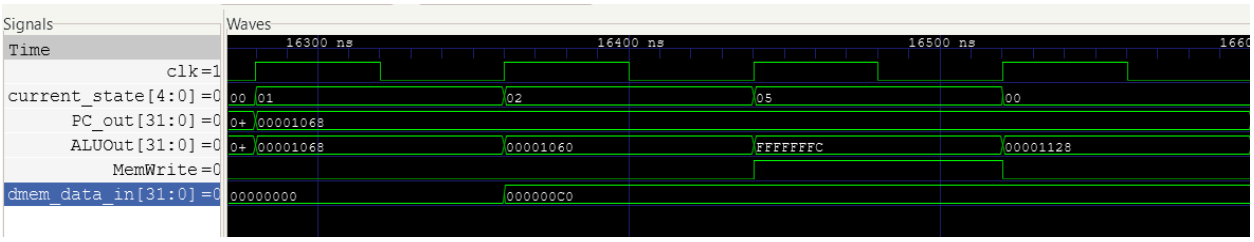

Figure 10: Simulation waveform demonstrating a Store Word (sw) instruction targeting the memory-mapped LED peripheral. The current_state signal transitions through Fetch (00), Decode (01), Address Calculation (02), and Memory Write (05). During the final state (05), the MemWrite signal is asserted while ALUOut drives the peripheral address 0xFFFFFFFC and

dmem_data_in provides the data value (0xC0), successfully performing the write operation to the LED controller.

B-Type Instruction Execution (Conditional Branch):

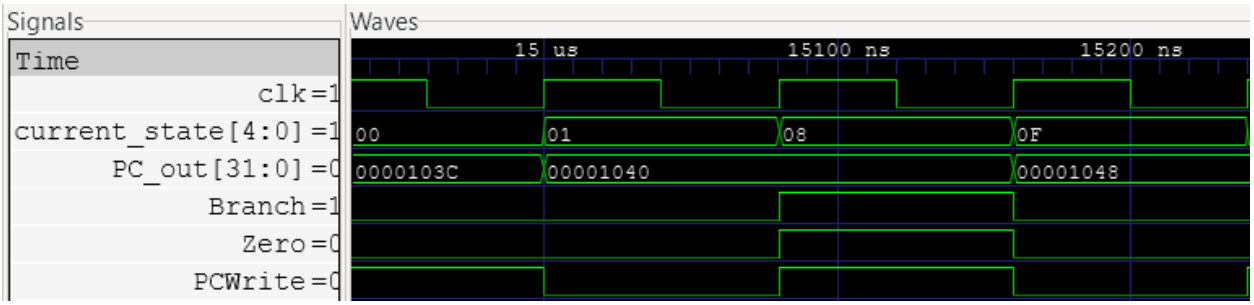

Figure 11: Simulation waveform demonstrating a taken conditional branch (BEQ). The processor transitions through Fetch (00), Decode (01), and the Branch state (08). The ALU evaluates the comparison, asserting Zero, while the Control Unit asserts Branch. The combination of these signals triggers PCWrite, updating the PC_out to the target address 0x1048 (skipping the sequential address 0x1044) during the Wait state (0F).

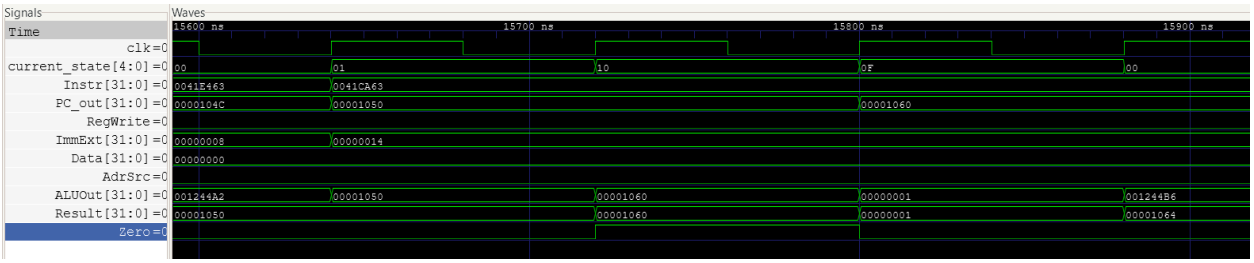

Figure 12: Branch Less Than (BLT) Instruction Execution. The processor executes BLT x3, x4, 20, transitioning through EXECUTE_BLT (state 10) where the ALU performs a signed comparison using SLT. Since x3 (0xFFEDCBA9, negative) is less than x4 (0x00123456, positive), the Zero flag asserts high, triggering PCWrite. The PC updates from 0x1050 to the branch target 0x1060 (PC + 20) during the WAIT state, demonstrating a taken conditional branch with signed comparison.

R-Type Instruction Execution (Register-Register Arithmetic)

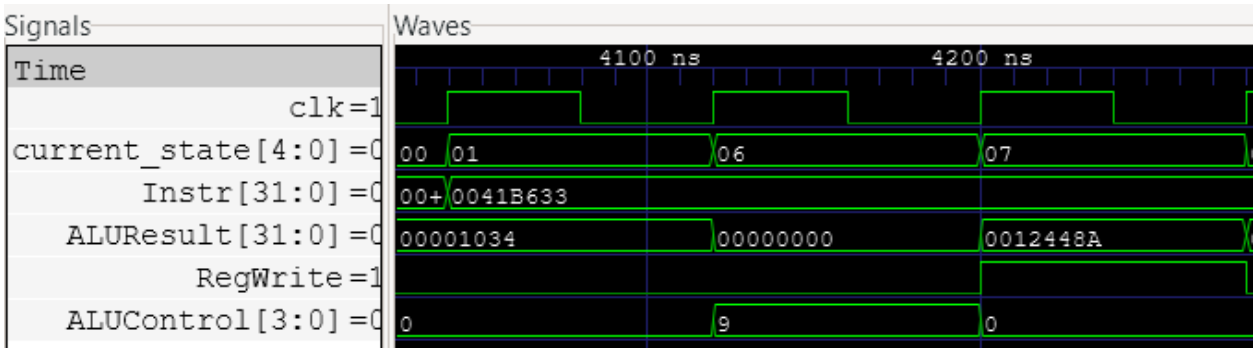

Figure 13: Simulation waveform for an R-Type instruction (SUB). The processor transitions to the EXECUTE_R state (06), where the ALU performs the subtraction between two register operands. The resulting value is then written back to the destination register during the Writeback state (07), indicated by the RegWrite signal asserting high.

U-Type Instruction Execution (Upper Immediate):

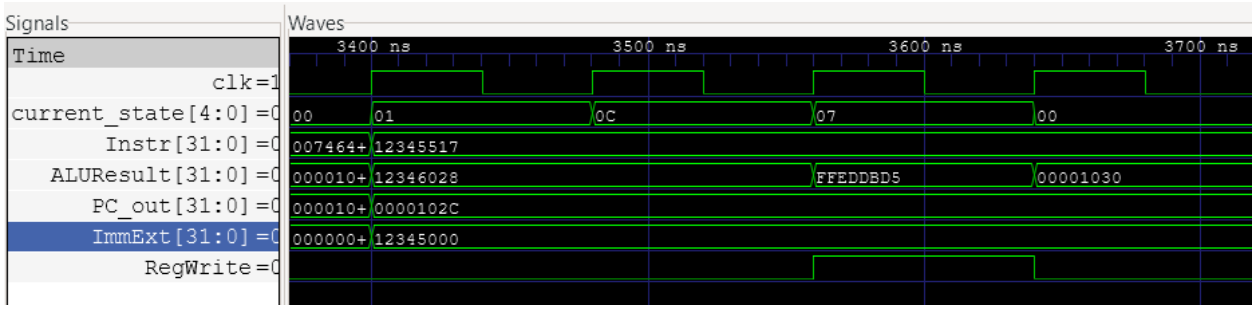

Figure 14: Simulation waveform for a U-Type instruction (AUIPC). The FSM transitions to the EXECUTE_UPC state (0C), where the ALU adds the sign-extended upper immediate value (ImmExt) to the current Program Counter (PC_out). The resulting calculated address (ALUResult) is then written to the destination register during the Writeback state (07), as indicated by the RegWrite signal.

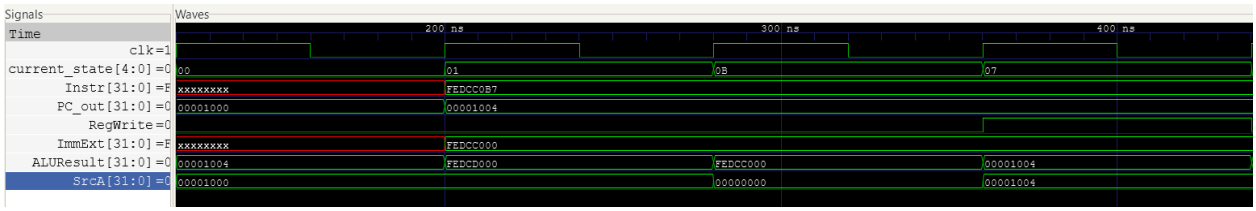

Figure 15: LUI Instruction Execution. The processor executes lui x1, 0xFEDCC, transitioning through FETCH (00), DECODE (01), EXECUTE_U (0B), and ALU_WB (07). During EXECUTE_U, SrcA is set to zero while the sign-extended immediate (0xFEDCC000) is

supplied via SrcB. The ALU computes 0 + immediate = 0xFEDCC000, which is written to register x1 when RegWrite asserts during the writeback state.

J-Type Instruction Execution (Jump and Link):

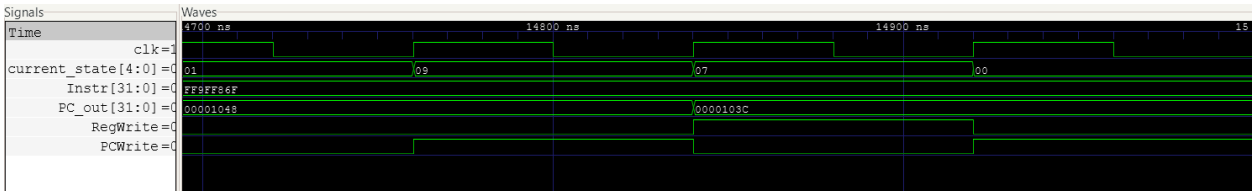

Figure 16: Simulation waveform for a J-Type instruction (JAL). The processor enters the JAL state (09), asserting PCWrite to update the Program Counter to the target address (visible as PC_out jumping from 0x1048 to 0x103C). In the subsequent Writeback state (07), RegWrite is asserted to store the return address (PC + 4) into the register file.

In addition to the waveforms shown, our test program successfully executes shift instructions (srli, srai, sll), logical operations (xori, ori, add), comparison operations (slt, sltu), store byte (sb), and unsigned loads (lhu, lbu), all of which follow the same state sequences as their respective instruction classes.

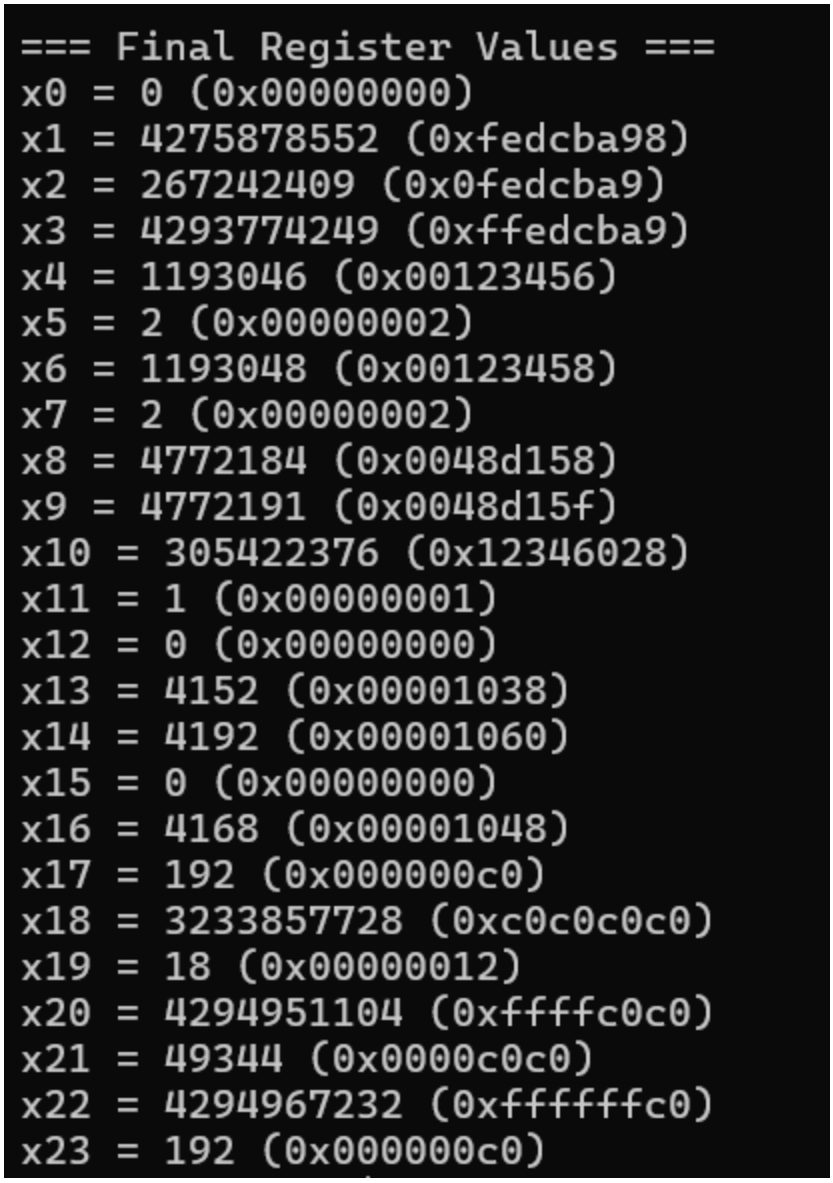

Figure 17: Final values of all registers printed by the testbench after program execution. Comparing these results to the expected register values annotated in the provided rv32i_test.s file shows that every register matches, confirming correct execution of our instruction set and datapath.

The expected register values in rv32i_test.s assume a base PC of 0x00000000, but our instruction memory starts at address 0x00001000; as a result, any PC-relative values (e.g., from auipc, jal, and jalr) appear exactly 0x1000 higher than the comments, even though they are functionally correct for our memory map.