Oscar Bao, Mia Chevere, Ashley Yang

MP4 Report

This report outlines the design and operation of an unpipelined, multicycle, 32-bit RISC-V integer microprocessor based on a von Neumann architecture. It also presents simulation results that demonstrate the processor's correct functionality. Throughout the project, we emphasized a modular design approach.

The design process began with a whiteboard sketch depicting the digital circuit structure of the processor. We chose to follow the textbook's multicycle, unpipelined design very closely for as many instructions described in the book there were – we figured it was the most comprehensive resource we had to go off of, and having a complete block diagram like Figure 7.27 allowed us to start work in a more modular fashion as well as keep on the same page. This visual framework helped us define the datapaths required for each instruction and identify the essential hardware components.

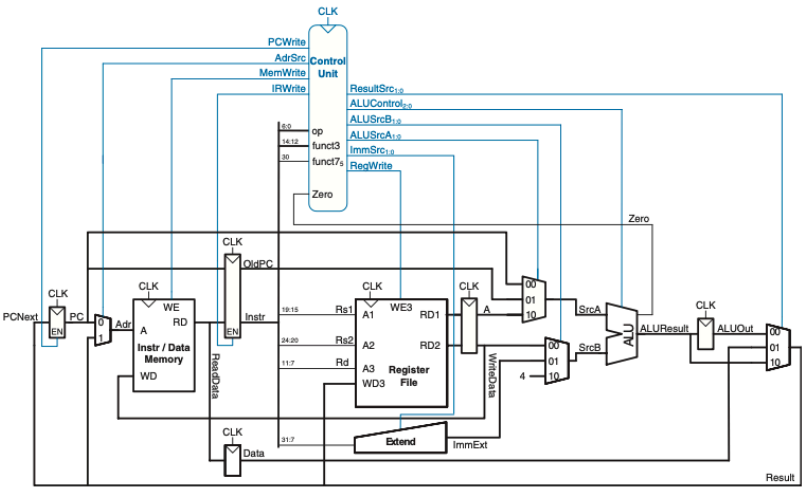

Figure 7.27 Complete multicycle processor

From there, we broke the system down into smaller, familiar digital blocks, such as arithmetic logic units and logic gates. We also made the choice to implement each block as separate .sv files in our organization, including the non-architectural registers and the muxes, in order to again keep consistency and a modular workflow. Each component was created with clear boundaries and standard IO parameters that followed the textbook's naming conventions,

enabling our processor to be developed, tested, and modified independently. This structure not only improved collaboration within the team, allowing async work, but also allowed us to directly map discrete hardware elements from our circuit diagram into corresponding modules in the final implementation.

Our processor takes 4 clock cycles for each instruction, and 3 clock cycles for lui and branch instructions. Each clock cycle is set to 100 MHz, so the timing of instruction execution is based on a 10 ns clock period. Though we at one point thought about putting the state switches on both the positive and negative clock edges in order to take even less clock cycles, we ended up choosing to synchronize all our sequential logic modules on the positive rising edge of the clock to reduce complexity and thereby make it easier to modify our processor if needed.

Our organization of the processor takes inspiration from the textbook's example System Verilog implementation of a single cycle RISC-V processor in Section 7.6.1; though ours is multicycle, we thought their structure made a lot of sense, particular with its separate memory module handling at the top module as our provided memory module is the "odd one out" in being the one we didn't write. So, this structure would allow us to test our written modules, such as the control unit and datapath, separately and together before interfacing with the memory module.

We start with a top level module, which drives the whole processor and contains a "riscv" module as well as the provided memory module. The riscv module handles the main interaction between the Control Unit and Datapath modules, as well as splits Instr from the datapath into its appropriate op, funct3, and funct7b5 bit ranges to go into the control unit. We use the Control Unit module to generate the control signals that guide the flow of data through the processor. It

functions as a finite state machine (FSM), managing the overall stages of the instruction cycle: fetch, decode, execute, and write-back. In practice, though we can conceptually group these stages as fetch, decode, execute, and write-back, we implemented our FSM with 13 states in order to handle the various execution processes of the different instructions; for example, Execute state signals from R-type instructions are different than from I-type instructions, since I-type instructions use immediates so the ALUSrcB comes from the Immediate Generator, or Extend Unit, rather than the Register File. Implementing specific states for various types also importantly allowed us to add 2 states for a few instructions not described in the textbook, which were auipc and lui.

Other components in the Control Unit are the Instruction Decoder, which interprets binary instructions retrieved from memory, decoding opcodes and operands and generating the necessary control outputs using combinational logic.

Datapath defines how data moves between components, directed by these control signals. The Register File provides temporary storage for operands and results, while the Memory (though not organized filewise with the datapath) holds both program instructions and runtime data. The ALU handles both arithmetic and logical operations using combinational logic, with control inputs determining which operation to perform. Our ALU expands from the 5 operations given in the textbook (ADD, SUB, AND, OR, and SLT) to additionally accommodate shifts (SRA, SRL, SLL), XOR, and SLTU (SLT but unsigned). An additional small but important difference was axing the non-architectural register Data that was part of the original textbook design, since our provided memory module is clocked on reads and so the non-architectural register actually fell behind in our processor.

Instruction execution follows a conventional multicycle flow. The Program Counter (PC) begins the process by pointing to the memory address of the next instruction to fetch. Once the instruction is retrieved, it is decoded, and the Control Unit generates the signals needed to drive the rest of the processor. During the execution stage, the ALU performs the specified operation using inputs from the Register File. The result is then written back to either a register or memory. Finally, the PC is updated, and the cycle repeats with the next instruction.

To verify the processor's functionality, we developed a series of testbenches using Cocotb. These simulations allowed us to not only validate the overall behavior of the processor but also isolate and debug specific components under controlled conditions. We created dedicated tests for key modules such as the ALU, the datapath, and the finite state machine within the Control Unit. These unit tests ensured that each module performed correctly in isolation. From there, we moved into implementing a full-system testbench to simulate the processor as a whole, running sequences of instructions and checking outputs at each stage of execution. We started with attempting to validate instruction fetching and correct outputs of the first two instructions provided in rv32i_test.txt, through outputting various relevant values, asserting that the instructions fetched correctly, and that the x1 register contained the correct values. The three following images show this initial scrap testing:

```
  -.--ns INFO     gpi                                ../gpi/GpiCommon.cpp:102  in gpi_print_registered_impl
   0.00ns INFO     cocotb                             Running on Icarus Verilog version 13.0 (devel)
   0.00ns INFO     cocotb                             Seeding Python random module with 1745364294
   0.00ns INFO     cocotb                             Initialized cocotb v2.0.0.dev0+61910544 from /Application
10544-py3.11-darwin-aarch64.egg/cocotb
   0.00ns INFO     cocotb                             Running tests
   0.00ns INFO     cocotb.regression                  running test_final.test_instruction_output (1/3)
                                                          Test the instruction output of the processor.
NING: ./memory.sv:76: $readmemh(rv32i_test.txt): Too many words in the file for the requested range [0:2047].
[0] = fedcc0b7
  20.00ns INFO     cocotb                             PC: 00000000, Instr: 00000000
  30.00ns INFO     cocotb                             PC: 00000004, Instr: fedcc0b7
  40.00ns INFO     cocotb                             PC: 00000004, Instr: fedcc0b7
  50.00ns INFO     cocotb                             PC: 00000004, Instr: fedcc0b7
  60.00ns INFO     cocotb                             PC: 00000008, Instr: a9808093
  70.00ns INFO     cocotb                             PC: 00000008, Instr: a9808093
  80.00ns INFO     cocotb                             PC: 00000008, Instr: a9808093
  90.00ns INFO     cocotb                             PC: 00000008, Instr: a9808093
 100.00ns INFO     cocotb                             PC: 0000000c, Instr: 0040d113
 110.00ns INFO     cocotb                             PC: 0000000c, Instr: 0040d113
 120.00ns INFO     cocotb                             PC: 0000000c, Instr: 0040d113
 130.00ns INFO     cocotb                             PC: 0000000c, Instr: 0040d113
 140.00ns INFO     cocotb                             PC: 00000010, Instr: 4040d193
 150.00ns INFO     cocotb                             PC: 00000010, Instr: 4040d193
 160.00ns INFO     cocotb                             PC: 00000010, Instr: 4040d193
 170.00ns INFO     cocotb                             PC: 00000010, Instr: 4040d193
 180.00ns INFO     cocotb                             PC: 00000014, Instr: fff1c213
 190.00ns INFO     cocotb                             PC: 00000014, Instr: fff1c213
 200.00ns INFO     cocotb                             PC: 00000014, Instr: fff1c213
 210.00ns INFO     cocotb                             PC: 00000014, Instr: fff1c213
 210.00ns INFO     cocotb.regression                  test_final.test_instruction_output passed
```

Fig 1: This screenshot shows a successful run of the test_instruction_output Cocotb test. The test verified the processor's ability to correctly fetch the provided sequence of instructions from memory. The Program Counter and instruction values are logged at each clock cycle to confirm proper sequencing and instruction decoding. The green passed at the bottom confirms the processor behaved as expected throughout the simution

```
 210.00ns INFO     cocotb.regression                  running test_final.test_first_instruction_fetch (2/3)
                                                          Check that the first instruction is correctly fetched and loaded.
 240.00ns INFO     cocotb.top                         state: 0000
 240.00ns INFO     cocotb.top                         next: 0001
 240.00ns INFO     cocotb.top                         instr: 0x00000000
 240.00ns INFO     cocotb.top                         opcode: 0000000
 240.00ns INFO     cocotb.top                         funct3: 000
 250.00ns INFO     cocotb.top                         state: 0001
 250.00ns INFO     cocotb.top                         next: 1100
 250.00ns INFO     cocotb.top                         instr: 0xfedcc0b7
 250.00ns INFO     cocotb.top                         opcode: 0110111
 250.00ns INFO     cocotb.top                         funct3: 100
 260.00ns INFO     cocotb.top                         state: 1100
 260.00ns INFO     cocotb.top                         next: 0000
 260.00ns INFO     cocotb.top                         instr: 0xfedcc0b7
 260.00ns INFO     cocotb.top                         opcode: 0110111
 260.00ns INFO     cocotb.top                         funct3: 100
 270.00ns INFO     cocotb.top                         state: 0000
 270.00ns INFO     cocotb.top                         next: 0001
 270.00ns INFO     cocotb.top                         instr: 0xfedcc0b7
 270.00ns INFO     cocotb.top                         opcode: 0110111
 270.00ns INFO     cocotb.top                         funct3: 100
 270.00ns INFO     cocotb.top                         Instruction fetched correctly: 0xfedcc0b7
 270.00ns INFO     cocotb.regression                  test_final.test_first_instruction_fetch passed
```

Fig 2: This screenshot shows the output of the test_first_instruction_fetch test case, verifying that the processor correctly fetches and loads the first instruction (0xfedcc0b7, corresponding to a lui instruction). The FSM, decoded instruction fields, and raw instruction values are logged across multiple cycles, confirming proper control flow and memory interaction. This test also passed.

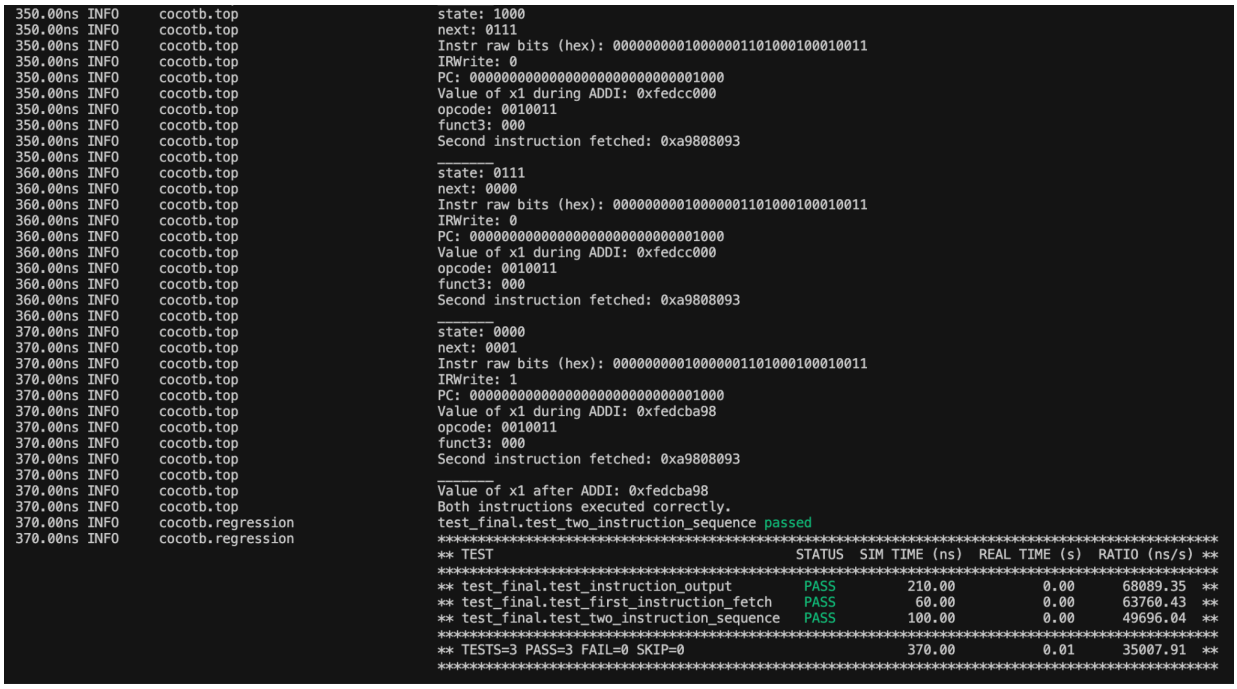

Fig 3: *This screenshot captures the successful execution of all three Cocotb testbenches. Specifically, it shows detailed debug output from the test_two_instruction_sequence, verifying correct instruction fetch and execution of a lui followed by an addi, along with an accurate write to the register x1. The FSM states, instruction values, and register contents are also logged. The bottom summary confirms all tests passed, with zero failures.*

To then comprehensively test our processor, we ran it through a sequence of instructions that contained at least one representative instruction from each class of RV32I instructions. We use the registers_debug logical array from the Register File to output the registers after each of our instructions executed into a separate .txt file, so that we were able to verify the actual results of our processor with the expected. Our results from running the provided instruction are in output_Brad_program_test.txt, where the final register values are shown here in Figure 4:

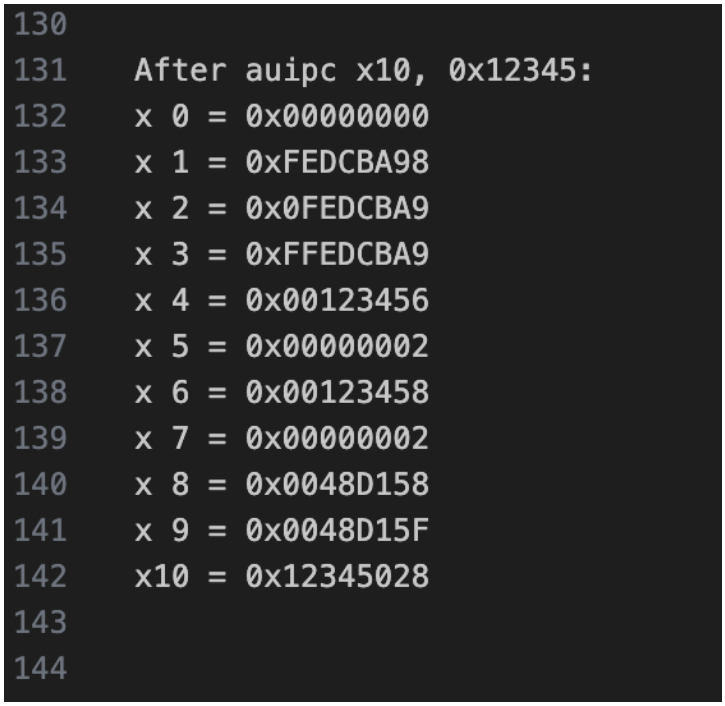

Fig 4: *This screenshot shows the successful running of the provided starter program.*

We expanded our testing to include missing representatives of the instruction classes, namely beq for B-type instruction, sw, low, and jal. The results are shown here:

Overall, our processor was successful. Our modular approach not only helped us implement our processor but also understand the components that went into it. By decomposing the design into distinct hardware components we better learned how each element functions individually as well as how they interact within the larger architecture. Organizing the processor into separate modules also allowed us to clearly map a multicycle design into hardware, while maintaining clean, testable code. Although our final implementation may not be the most efficient in terms of speed or clock cycles, efficiency was not our primary goal. Instead, we prioritized comprehension and correctness, and in that regard, we succeeded.