

## 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. As we

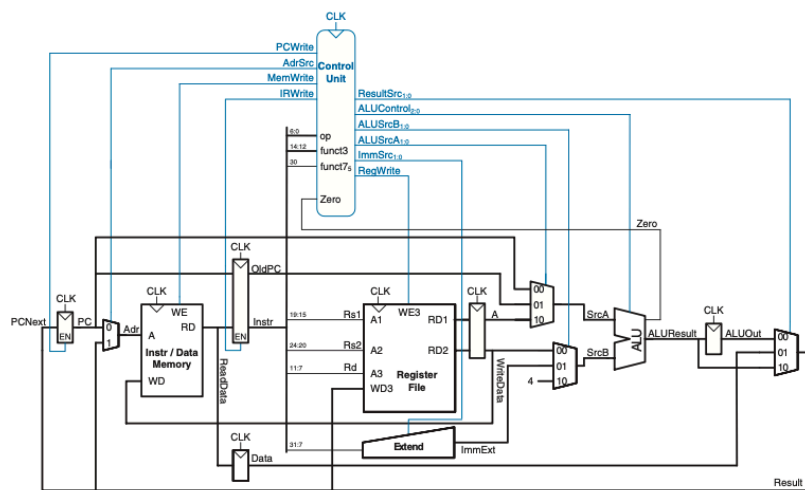


Figure 7.27 Complete multicycle processor

went on, we did then modified our implementation from this original base in order to accommodate our provided memory module as well as implement additional instructions.

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 5 clock cycles for the load instruction. 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 mainly 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 with a clock and reset signal and contains a "riscv" module as well as the provided memory module. We chose to add a

reset signal in order to allow our processor to settle for a few clock cycles upon initialization (incorporated into the final test\_top.py file described later on); we found this fixed some errors with executing the stages of the first instruction. 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 16 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 states for a few instructions not described in the textbook, such as auipc and lui, as well as a few states like a specific branch target write back stage (to accomodate what we found to be a needed extra cycle for our beq implementation) and a error stage.

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, as well as the ALU Decoder and branch Decoder.

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. We also expanded our Result multiplexer to include input from the output of the Immediate Generator, in order to implement our lui instruction.

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.
WARNING: ./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 simulation.*

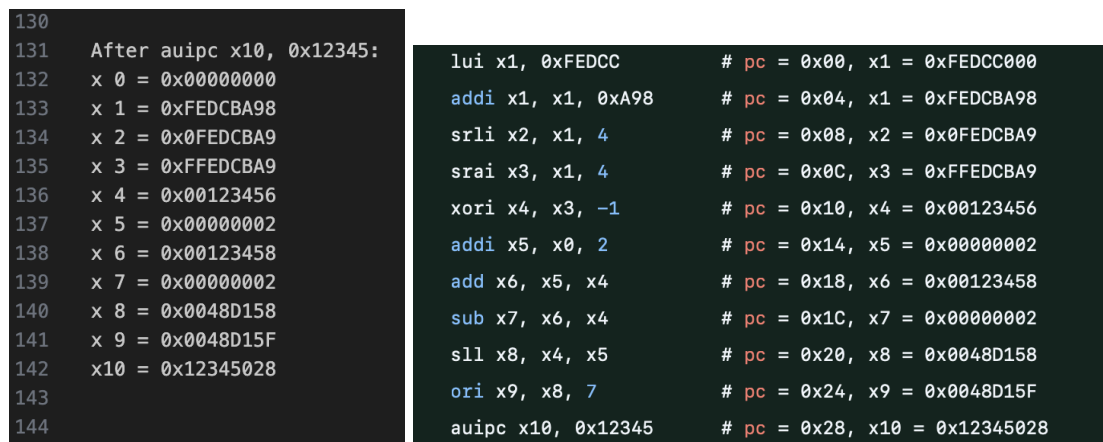
```

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: 00000000
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

```



We started with running the provided instruction sequence. Our results from running the provided instruction are in output\_Brad\_program\_test.txt, which we obtained by running an earlier version of test\_top.py. The final register values are shown here in Figure 4, which we verified the correct passing of the test by comparing the .txt to the commented provided expected values:



130			
131	After auipc x10, 0x12345:	lui x1, 0xFEDCC	# pc = 0x00, x1 = 0xFEDCC000
132	x 0 = 0x00000000	addi x1, x1, 0xA98	# pc = 0x04, x1 = 0xFEDCBA98
133	x 1 = 0xFEDCBA98	srlr x2, x1, 4	# pc = 0x08, x2 = 0x0FEDCBA9
134	x 2 = 0x0FEDCBA9	srai x3, x1, 4	# pc = 0x0C, x3 = 0xFFEDCBA9
135	x 3 = 0xFFEDCBA9	xori x4, x3, -1	# pc = 0x10, x4 = 0x00123456
136	x 4 = 0x00123456	addi x5, x0, 2	# pc = 0x14, x5 = 0x00000002
137	x 5 = 0x00000002	add x6, x5, x4	# pc = 0x18, x6 = 0x00123458
138	x 6 = 0x00123458	sub x7, x6, x4	# pc = 0x1C, x7 = 0x00000002
139	x 7 = 0x00000002	sll x8, x4, x5	# pc = 0x20, x8 = 0x0048D158
140	x 8 = 0x0048D158	ori x9, x8, 7	# pc = 0x24, x9 = 0x0048D15F
141	x 9 = 0x0048D15F	auipc x10, 0x12345	# pc = 0x28, x10 = 0x12345028
142	x10 = 0x12345028		
143			
144			

Fig 4: These two screenshots show the final register values in output\_Brad\_program\_test.txt from running the provided starter program (left) through test\_top.py, corresponding correctly to the provided expected values of the registers (right).

From here, we modified our testing to include representatives of the base instruction classes that weren't covered in the provided test program, namely beq for B-type instruction, sw, lw, and jal for J-type instruction. We used the same format of running test\_top.py and comparing the output of the registers, dumped into "register\_dump.txt" for this final run, with the expected values of the register based on the instructions. Our successful results are shown here.

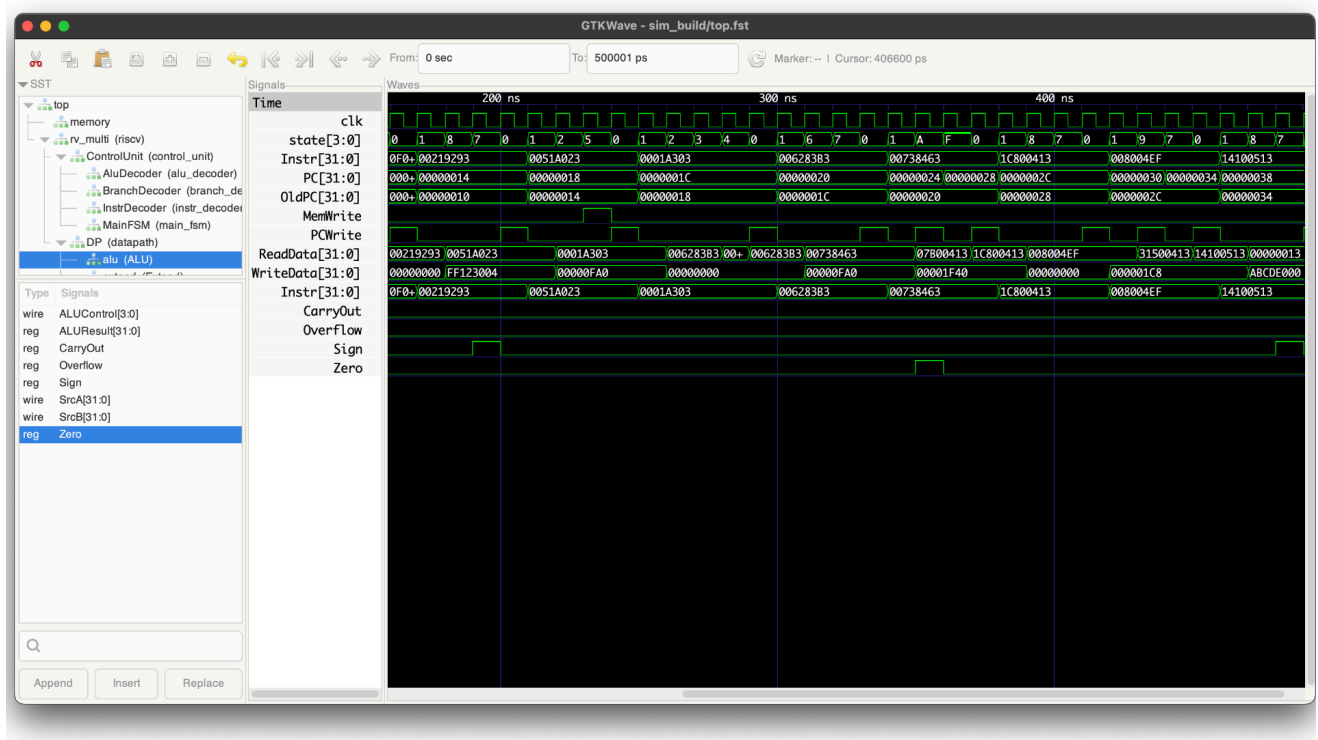


Fig 5: This depicts a sample of various relevant waveforms in Gtkwave from successfully running our processor in our final sequence. For example, we can see the sw and lw being run successfully in the state sequence of “0 1 5 0” for sw, and “0 1 2 3 4 0” for lw, and what data is being written and read (among other things) during their execution.



```

480.00ns INFO cocotb.top readdata: 0x13
480.00ns INFO cocotb memwrite: 0
480.00ns INFO cocotb writedata: 0xABCDE000
480.00ns INFO cocotb.top adr: 0x38
-----
490.00ns INFO cocotb PC: 00000038, oldPC: 00000034, Instr: 14100513
490.00ns INFO cocotb state: 7
490.00ns INFO cocotb next: 0
490.00ns INFO cocotb opcode: 0010011
-----
490.00ns INFO cocotb.top readdata: 0x13
490.00ns INFO cocotb memwrite: 0
490.00ns INFO cocotb writedata: 0xABCDE000
490.00ns INFO cocotb.top adr: 0x38
-----
500.00ns INFO cocotb PC: 00000038, oldPC: 00000034, Instr: 14100513
500.00ns INFO cocotb state: 0
500.00ns INFO cocotb next: 1
500.00ns INFO cocotb opcode: 0010011
-----
500.00ns INFO cocotb.top readdata: 0x13
500.00ns INFO cocotb memwrite: 0
500.00ns INFO cocotb writedata: 0xABCDE000
500.00ns INFO cocotb.top adr: 0x38
500.00ns INFO cocotb.top Register dump saved to register_dump.txt
500.00ns INFO cocotb.top ALL REGISTERS CORRECT
500.00ns INFO cocotb.top dump_registers_test passed
500.00ns INFO cocotb.regression *****
** TEST STATUS SIM TIME (ns) REAL TIME (s) RATIO (ns/s) **
*****
** test_top.dump_registers_test PASS 500.00 0.03 1560.24 **
*****
** TESTS=1 PASS=1 FAIL=0 SKIP=0 500.00 0.08 5979.22 **
*****

# Expected comments for each PC step
instruction_log = [
    "lui x1, 0xABCDE", # pc = 0x00, x1 = 0xABCDE000
    "auipc x2, 0xFF123", # pc = 0x04, x2 = 0xFF123004
    "addi x3, x0, 1000", # pc = 0x08, x3 = 0x000003E8
    "ori x4, x3, 0xF0", # pc = 0x0C, x4 = 0x000003F8
    "slli x5, x3, 2", # pc = 0x10, x5 = 0x00000FA0
    "sw x5, 0(x3)", # pc = 0x14, Mem[0x3E8] = 0x00000FA0
    "lw x6, 0(x3)", # pc = 0x18, x6 = 0x00000FA0
    "add x7, x5, x6", # pc = 0x1C, x7 = 0x00001F40
    "beq x7, x7, SKIP", # pc = 0x20, branch taken
    # "addi x8, x0, 123", # pc = 0x24, skipped
    "SKIP: addi x8, x0, 456", # pc = 0x28, x8 = 0x000001C8
    "jal x9, END", # pc = 0x2C, x9 = 0x00000030
    # "addi x8, x0, 789", # pc = 0x30, skipped
    "END: addi x10, x0, 321", # pc = 0x34, x10 = 0x00000141
]

```

After END: addi x10, x0, 321:

```

x 0 = 0x00000000
x 1 = 0xABCDE000
x 2 = 0xFF123004
x 3 = 0x000003E8
x 4 = 0x000003F8
x 5 = 0x00000FA0
x 6 = 0x00000FA0
x 7 = 0x00001F40
x 8 = 0x000001C8
x 9 = 0x00000030
x10 = 0x00000141

```

Fig 7: These screenshots depict the output of our final test. The top image shows the successful terminal output of our test passing and some of the value messages we used to debug along the way, which are stored in “terminal\_output.txt” for reference. The bottom two screenshots show the comparison between our actual final register output in “register\_dump.txt” (left) and the correctly corresponding expected values and instructions specified towards the top of “test\_top.py” (right).

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.