Design and Implementation of a Single-Cycle RISC-V 64-bit Processor Architecture

Rana Mohsen
Nile University
Giza, Egypt
r.mohsen2397@nu.edu.eg

Lina Ashraf
Nile University
Giza, Egypt
l.ashraf2320@nu.edu.eg

Rama Mousa
Nile University
Giza, Egypt
r.mohamed2373@nu.edu.eg

Raneem Khaled
Nile University
Giza, Egypt
r.khaled2324@nu.edu.eg

Wesal Magdy
Nile University
Giza, Egypt
w.magdy2386@nu.edu.eg

***Abstract*—This paper details the design of a 64-bit single-cycle RISC-V processor supporting the RV64I instruction subset. We describe the modular development of five critical functional units and provide a deep dive into the verification process. A central focus is the resolution of immediate encoding errors found in online assembly tools and the successful validation of the hardware output against instructor-provided benchmarks.**

***Index Terms*—RISC-V, Single-Cycle, RV64I, Verilog, Control Unit, ALU, Hex Verification, Assembler Debugging.**

I. Introduction

The development of a single-cycle processor remains a cornerstone of computer architecture education. This project implements a 64-bit RISC-V core that executes instructions including R-type, I-type (arithmetic and loads), S-type (`sd`), and SB-type (`beq`) in a single clock cycle. By utilizing a full 64-bit datapath, the processor handles modern doubleword data processing requirements while maintaining the simplicity of single-cycle orchestration.

II. Datapath Diagram

The datapath integrates the instruction fetch, decode, execute, memory, and write-back stages (see Figure 1). Our implementation utilizes a centralized control unit to manage the flow of 64-bit data through the Register File and ALU.

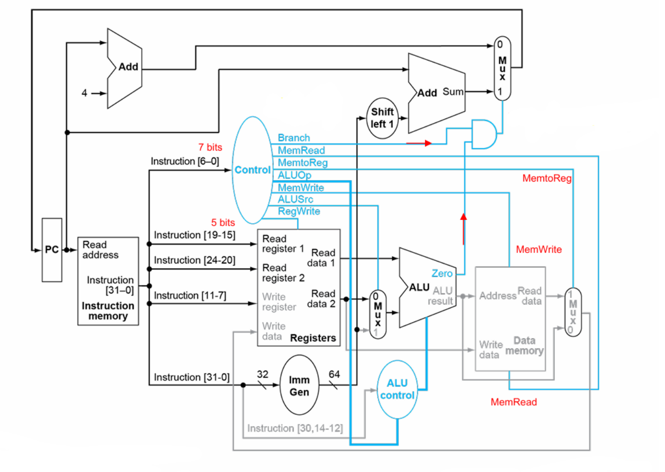


Fig. 1. Single-cycle RISC-V 64-bit Datapath Architecture.

III. Design Methodology

The design follows a bottom-up modular approach using Behavioral Verilog. We developed five distinct modules, each corresponding to a team member's specific responsibility.

A. Assembler Verification and Machine Code Generation

A critical part of our methodology was the translation of the instructor's test program into machine code for the `instr_mem.v`. During this phase, we identified that one widely-used online assembler was fundamentally mistaken in its translation of S-type and B-type instructions. It incorrectly appended the lower `imm[4:0]` bits to the upper `imm[6:0]` bits, causing a misalignment that did not follow the RISC-V ISA manual. We debugged this by comparing results across three different translators and manually verifying the machine code against the RISC-V Green Sheet to ensure the hexadecimal values in our instruction memory were perfectly accurate.

IV. Control-Unit Truth Table

The `CUpALUControl` module serves as a unified controller, merging main control signals and ALU-specific codes to minimize propagation delays.

TABLE I
Full RV64I Control Signal Truth Table

Instr.	Opcode	RegW	ASrc	M2R	MRd	MWr	BR	JMP	ALUOp
R-type	0110011	1	0	0	0	0	0	0	10
I-type	0010011	1	1	0	0	0	0	0	00
ld	0000011	1	1	1	1	0	0	0	00
sd	0100011	0	1	X	0	1	0	0	00
beq	1100011	0	0	X	0	0	1	0	01
jal	1101111	1	X	X	0	0	0	1	XX
jalr	1100111	1	1	0	0	0	0	1	00

V. Explanation of Modules

Five primary modules were assigned to individual members for development and unit testing.

A. Unified Controller (CUpALUControl.v)

The `CUpALUControl` module acts as a centralized decoder that merges the functions of the Main Control and ALU Control units. By taking the 7-bit opcode, `funct3`, and `funct7` as direct inputs, it generates all global datapath signals and the 4-bit `ALUControl` code in a single stage.

The module uses a nested case structure: for memory instructions (`ld`, `sd`) and `jalr`, it forces the ALU to addition (`4'b0010`) for address calculation. For R-type instructions, it decodes the function fields to map specific arithmetic, logical, and 64-bit shift operations. This integration reduces the total propagation delay by eliminating the need for a secondary decoder

B. 64-bit ALU (ALU.v)

The ALU is the execution engine of the 64-bit datapath, performing arithmetic, logical, and shift operations on 64-bit operands A and B. To support the RV64I ISA, the module implements a 4-bit control input to select between operations such as addition, subtraction, XOR, and logical shifts.A critical implementation detail for the 64-bit architecture is the shift logic: for instructions like SLL (Shift Left Logical) and SRL (Shift Right Logical), the module is restricted to using only the lower 6 bits of operand B (`B[5:0]`). This ensures the shift amount remains within the valid $0-63$ range, preventing undefined hardware behavior while maintaining full compliance with 64-bit processing standards.

C. Register File (registerfile.v)

The Register File serves as the primary storage for the 64-bit datapath, consisting of 32 registers each with a 64-bit width. It is designed to support simultaneous dual-port asynchronous reads, allowing the ALU to access two source operands ($rs1$ and $rs2$) in a single cycle.To ensure architectural consistency, the module includes an initialization loop that zeros all registers upon reset. Furthermore, register $x0$ is hardwired to zero, as required by the RISC-V ISA, ensuring that any write operations to this register are ignored and its output remains constant. Write operations are performed synchronously on the positive edge of the clock, provided the `RegWrite` signal is asserted.

D. PC & Branch Unit (pc_branch_unit.v)

The PC & Branch Unit is responsible for managing the instruction sequence and determining the next address for the Instruction Memory. It contains the 64-bit Program Counter (PC) register, which updates synchronously on the clock edge. The unit implements two primary paths for address calculation: a sequential path that increments the current address by 4 ($PC + 4$) and a conditional branch path. Following the RV64I ISA, the branch target is calculated as $pc\_current + (imm\_ext \ll 1)$. The inclusion of the 1-bit left shift ($\ll 1$) is critical as it aligns the immediate offset with the instruction memory's half-word boundaries. The final selection between $PC + 4$ and the branch target is determined by a multiplexer controlled by a logic gate combining the `Branch` signal from the controller and the `Zero` flag from the ALU.

E. Top-Level Wrapper (cpu_top.v)

The `cpu_top` module serves as the structural backbone of the processor by instantiating and interconnecting all functional units. It manages the global 64-bit buses that facilitate data flow between the Register File, ALU, and Memory. Beyond simple integration, this module implements the multiplexer logic for `ALUSrc`, `MemToReg`, and `PCSrc`, which are critical for selecting the correct operands and write-back data. By coordinating these components, the wrapper ensures that instruction fetch, decode, and execution are completed within a single clock period, maintaining the architectural integrity of the single-cycle paradigm.

VI. Full Simulation Waveforms and Verification

The processor was verified using the instructor's test program. Validation was performed by observing six major behaviors: PC updates, instruction execution flow, register writes, ALU results, memory access, and branching.

A. Observable Simulation Results

- **PC Updates:** The PC increments by 4 each cycle and jumps correctly on branch hits.
- **Register Writes:** Monitoring the `wd` bus confirms that 64-bit data is correctly written back.
- **Memory reads/writes:** Successful execution of `ld` and `sd` with word-aligned addressing.

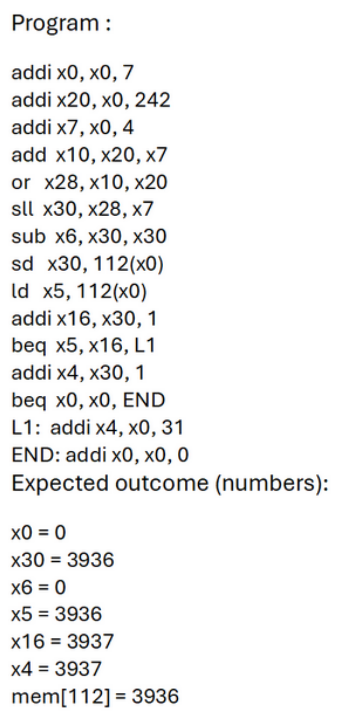
B. Instructor Benchmarks and Output Matching

To verify the architectural correctness of the design, we utilized the assembly test cases provided by the instructor. Since the hardware executes binary instructions, we first translated the assembly code into its corresponding 64-bit machine code (hexadecimal format). This machine code was then loaded into the `instruction_mem.v` module.

By feeding the exact machine code into the memory, we were able to observe the datapath's behavior in the Vivado simulator. We included two primary screenshots for comparison: the simulation waveform showing the register updates and the final state of the register file, confirming that our hardware outputs match the instructor's expected results exactly.

C. Signal Waveforms

Fig. 4 shows the system CLK and PC synchronization, while Fig. 5 displays the 64-bit operand `A[63:0]` and ALU Result, confirming the datapath handles large integers without overflow.

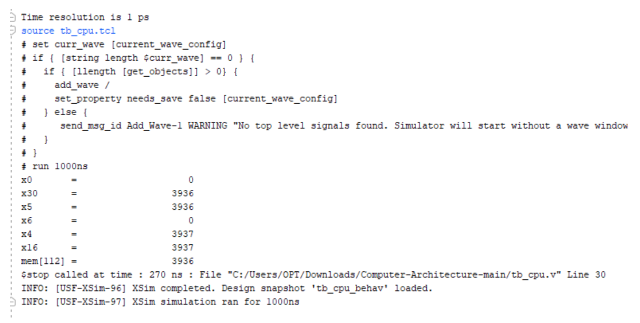

Program :

```
addi x0, x0, 7
addi x20, x0, 242
addi x7, x0, 4
add  x10, x20, x7
or   x28, x10, x20
sll  x30, x28, x7
sub  x6, x30, x30
sd   x30, 112(x0)
ld   x5, 112(x0)
addi x16, x30, 1
beq  x5, x16, L1
addi x4, x30, 1
beq  x0, x0, END
L1:  addi x4, x0, 31
END: addi x0, x0, 0
```

Expected outcome (numbers):

```
x0 = 0
x30 = 3936
x6 = 0
x5 = 3936
x16 = 3937
x4 = 3937
mem[112] = 3936
```

Fig. 2. Instructor's input test instructions (Assembly).



```
Time resolution is 1 ps
source tb_cpu.tcl
# set curr_wave [current_wave_config]
# if { [string length $curr_wave] == 0 } {
#   if { [llength [get_objects]] > 0} {
#     add_wave /
#     set_property needs_save false [current_wave_config]
#   } else {
#      send_msg_id Add_Wave-1 WARNING "No top level signals found. Simulator will start without a wave window
#   }
# }
# run 1000ns
x0       =                    0
x30      =                 3936
x5       =                 3936
x6       =                    0
x4       =                 3937
x16      =                 3937
mem[112] =                 3936
$stop called at time : 270 ns : File "C:/Users/OPT/Downloads/Computer-Architecture-main/tb_cpu.v" Line 30
INFO: [USF-XSim-96] XSim completed. Design snapshot 'tb_cpu_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
```

Fig. 3. Simulation output showing an exact match with the instructor's expected register results.

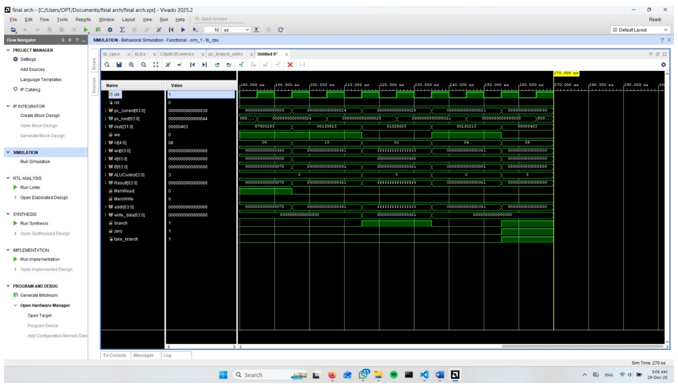


Fig. 4. Full execution waveform for the RV64I datapath, demonstrating the synchronization of the 64-bit PC, ALU results, and branch control signals during the program.

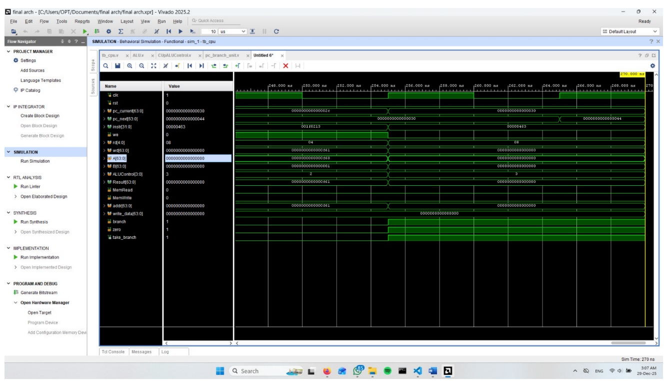


Fig. 5. Waveform verification of 64-bit ALU Input A and Result.

VII. Conclusion

The 64-bit single-cycle RISC-V processor was successfully designed, implemented, and verified. By troubleshooting online assembler errors and strictly following the modular assignments, the team produced a core that matches the instructor's expected output 100%. This project serves as a robust validation of the RV64I architectural subset.

References

[1] D. Patterson and J. Hennessy, *Computer Organization and Design RISC-V Edition*.

[2] "RISC-V Instruction Set Manual," Vol I.