RISC-V Pipelined CPU

Ouail Slama

Yahia Kilany

Department of computer science and engineering

CSCE 3301: Computer Architecture

Dr. Shrief Salama

Nov 24, 2025

Introduction

RISC-V is a modern open-source instruction set architecture (ISA) that has gained significant attention due to its simplicity, modular design, and extensibility. Originating in academia, RISC-V offers an accessible and learner-friendly platform for processor design and research, making it an ideal choice for educational projects in computer architecture. Its clean and well-documented specifications allow a wide range of hardware implementations, from simple embedded processors to high-performance CPUs.

In this project, we implemented a pipelined version of the RV32I, the base 32-bit integer instruction subset of the RISC-V ISA, along with the RV32C and RV32M extensions, on the Nexys A7 FPGA platform. The pipelined design improves instruction throughput by overlapping the execution of multiple instructions through distinct pipeline stages. Our implementation includes standard instruction support, hazard detection and handling mechanisms such as data forwarding and stalling, and thorough testing to verify functionality.

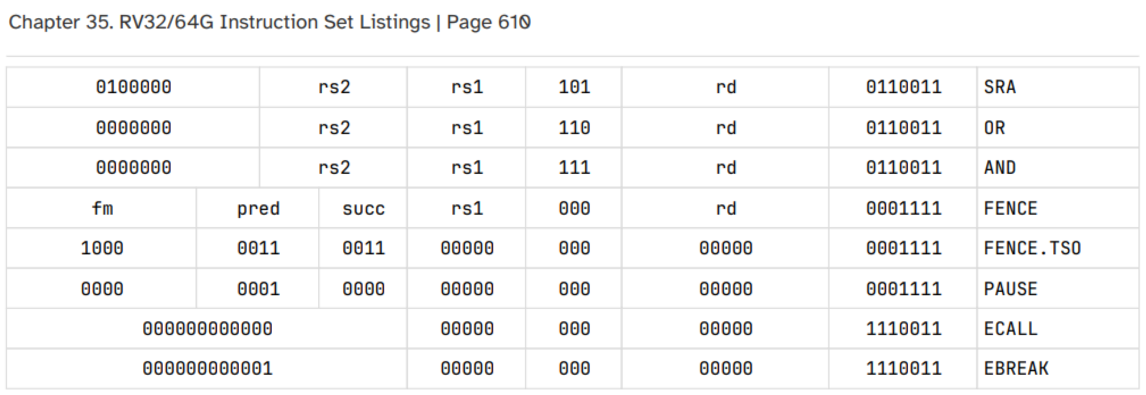
An important constraint in our design is the use of a single, single-ported memory for both instructions and data. This limitation introduces structural hazards since instruction fetch and data memory accesses cannot occur simultaneously in the same clock cycle. To address this, our pipeline issues an instruction every two clock cycles, effectively doubling the CPI compared to an ideal single-cycle-per-instruction pipeline. This design choice influences hazard handling strategies and overall pipeline performance, and it is thoroughly documented within this report.

Project Overview

In this project, we implemented a pipelined version of the RV32I base integer instruction subset of the RISC-V ISA. Our design integrates support for the 42 main user-level instructions defined in the RV32I specification. Additionally, we extended functionality by implementing the multiplication and division instructions from the M extension, as well as support for the compressed instruction set to enhance code density and efficiency.

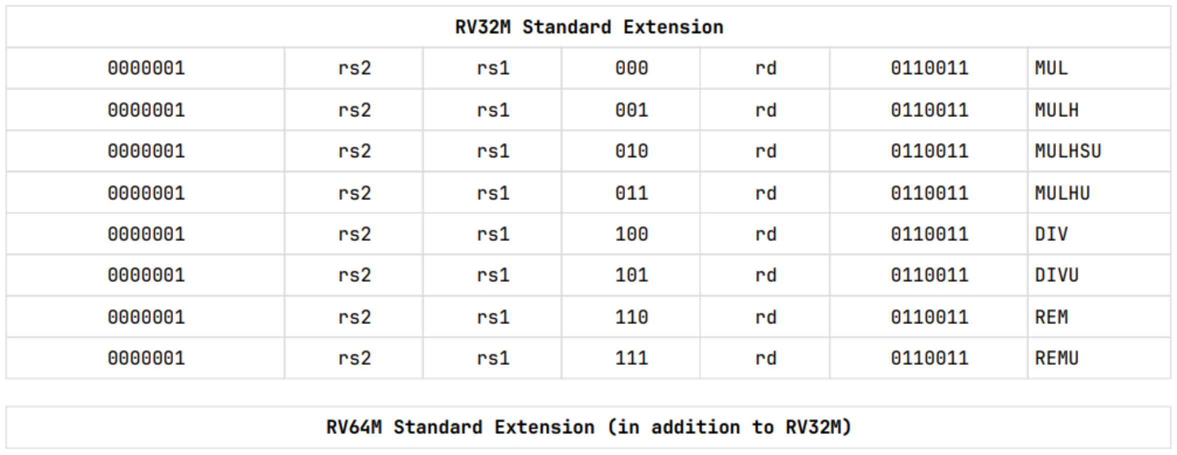
31	27	26	25	24 20	19 15	14 12	11 7	6 0	
funct7				rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]					rs1	funct3	rd	opcode	I-type
imm[11:5]				rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]				rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]							rd	opcode	U-type
imm[20|10:1|11|19:12]							rd	opcode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

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


Chapter 35. RV32/64G Instruction Set Listings | Page 610

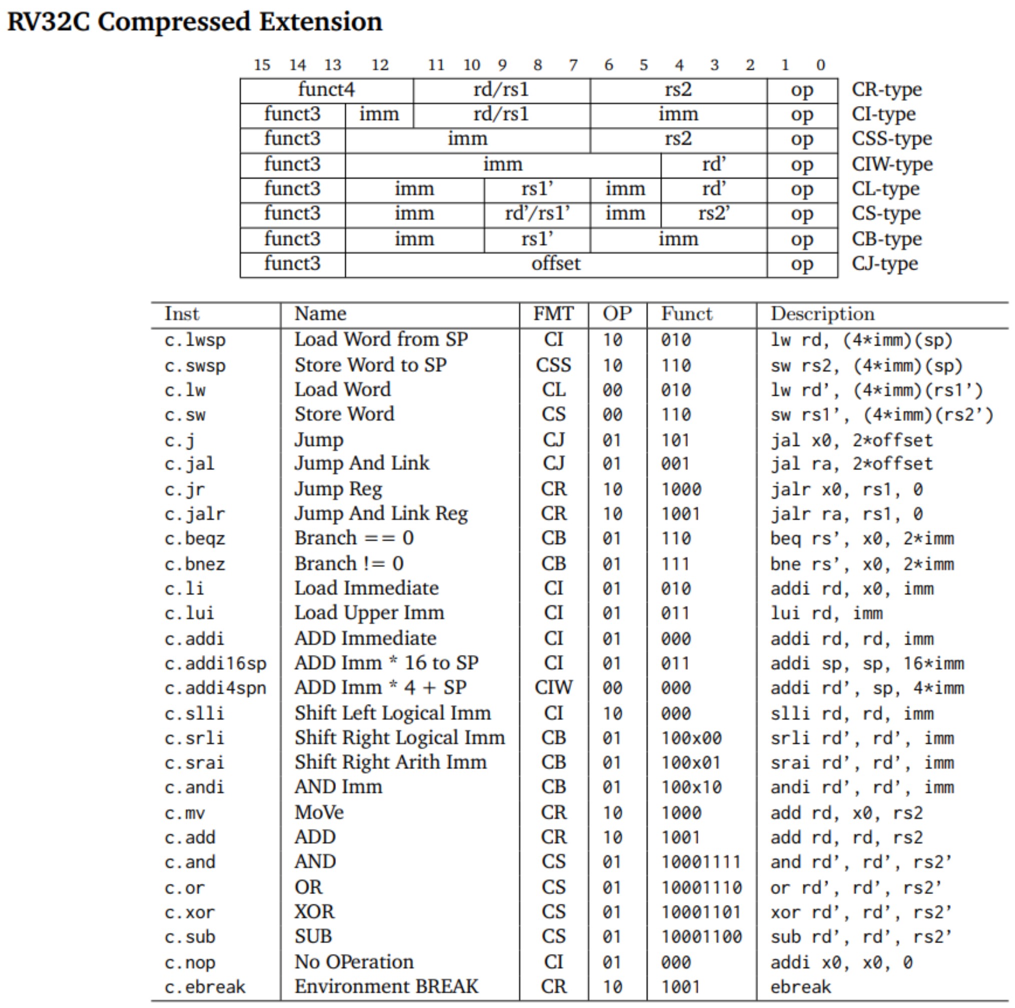
0100000		rs2	rs1	101	rd	0110011	SRA
0000000		rs2	rs1	110	rd	0110011	OR
0000000		rs2	rs1	111	rd	0110011	AND
fm	pred	succ	rs1	000	rd	0001111	FENCE
1000	0011	0011	00000	000	00000	0001111	FENCE.TSO
0000	0001	0000	00000	000	00000	0001111	PAUSE
000000000000			00000	000	00000	1110011	ECALL
000000000001			00000	000	00000	1110011	EBREAK

Figure 1: Encoding of base RV32I instructions


RV32M Standard Extension						
0000001	rs2	rs1	000	rd	0110011	MUL
0000001	rs2	rs1	001	rd	0110011	MULH
0000001	rs2	rs1	010	rd	0110011	MULHSU
0000001	rs2	rs1	011	rd	0110011	MULHU
0000001	rs2	rs1	100	rd	0110011	DIV
0000001	rs2	rs1	101	rd	0110011	DIVU
0000001	rs2	rs1	110	rd	0110011	REM
0000001	rs2	rs1	111	rd	0110011	REMU

RV64M Standard Extension (in addition to RV32M)

Figure 2: Encoding of RVM instructions

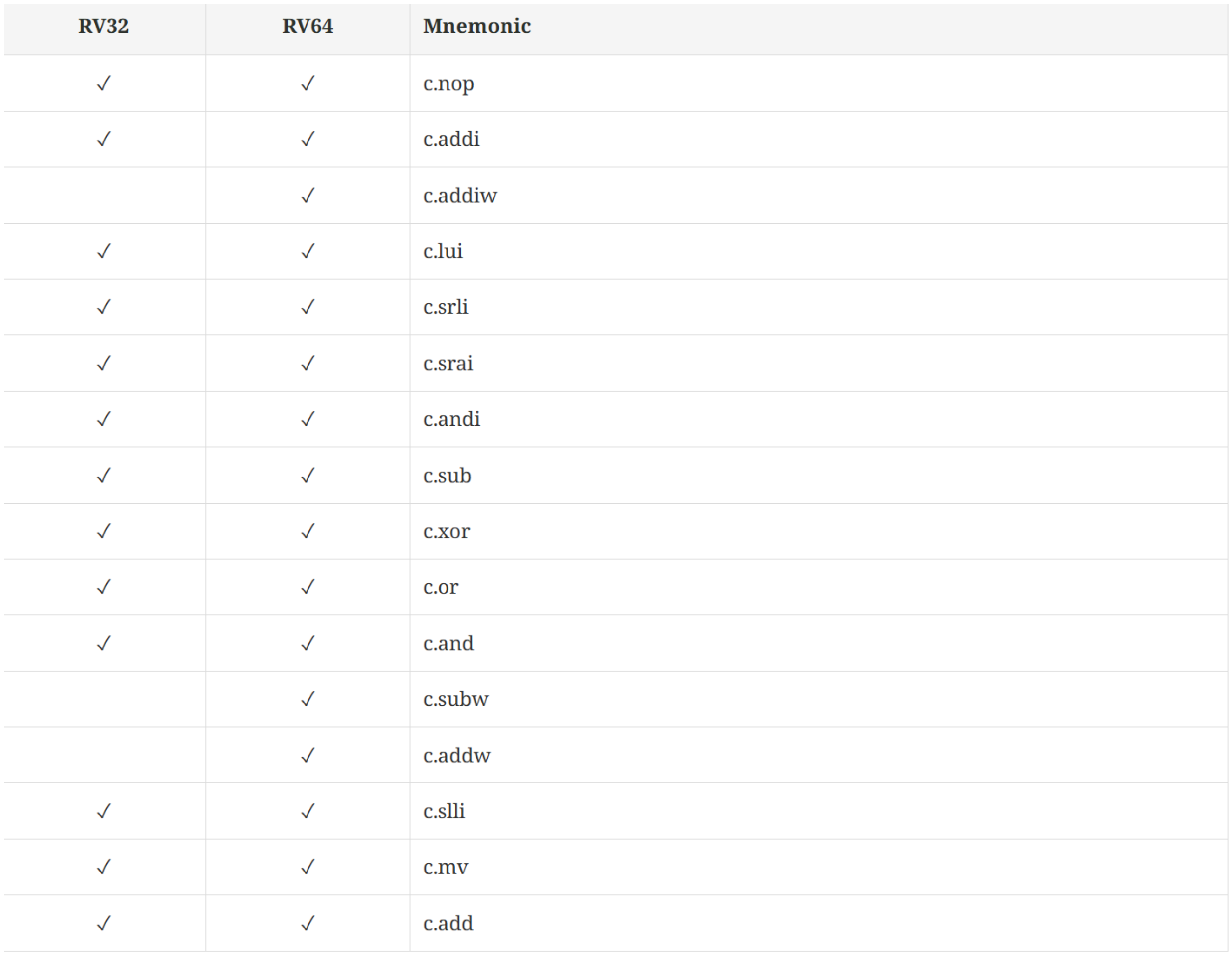


RV32C Compressed Extension

15 14 13	12	11 10	9 8 7	6 5	4 3 2	1 0	
funct4		rd/rs1		rs2		op	CR-type
funct3	imm	rd/rs1		imm		op	CI-type
funct3	imm			rs2		op	CSS-type
funct3	imm				rd’	op	CIW-type
funct3	imm		rs1’	imm	rd’	op	CL-type
funct3	imm		rd’/rs1’	imm	rs2’	op	CS-type
funct3	imm		rs1’	imm		op	CB-type
funct3	offset					op	CJ-type

Inst	Name	FMT	OP	Funct	Description
`c.lwsp`	Load Word from SP	CI	10	010	`lw rd, (4*imm)(sp)`
`c.swsp`	Store Word to SP	CSS	10	110	`sw rs2, (4*imm)(sp)`
`c.lw`	Load Word	CL	00	010	`lw rd’, (4*imm)(rs1’)`
`c.sw`	Store Word	CS	00	110	`sw rs1’, (4*imm)(rs2’)`
`c.j`	Jump	CJ	01	101	`jal x0, 2*offset`
`c.jal`	Jump And Link	CJ	01	001	`jal ra, 2*offset`
`c.jr`	Jump Reg	CR	10	1000	`jalr x0, rs1, 0`
`c.jalr`	Jump And Link Reg	CR	10	1001	`jalr ra, rs1, 0`
`c.beqz`	Branch == 0	CB	01	110	`beq rs’, x0, 2*imm`
`c.bnez`	Branch != 0	CB	01	111	`bne rs’, x0, 2*imm`
`c.li`	Load Immediate	CI	01	010	`addi rd, x0, imm`
`c.lui`	Load Upper Imm	CI	01	011	`lui rd, imm`
`c.addi`	ADD Immediate	CI	01	000	`addi rd, rd, imm`
`c.addi16sp`	ADD Imm * 16 to SP	CI	01	011	`addi sp, sp, 16*imm`
`c.addi4spn`	ADD Imm * 4 + SP	CIW	00	000	`addi rd’, sp, 4*imm`
`c.slli`	Shift Left Logical Imm	CI	10	000	`slli rd, rd, imm`
`c.srli`	Shift Right Logical Imm	CB	01	100x00	`srli rd’, rd’, imm`
`c.srai`	Shift Right Arith Imm	CB	01	100x01	`srai rd’, rd’, imm`
`c.andi`	AND Imm	CB	01	100x10	`andi rd’, rd’, imm`
`c.mv`	MoVe	CR	10	1000	`add rd, x0, rs2`
`c.add`	ADD	CR	10	1001	`add rd, rd, rs2`
`c.and`	AND	CS	01	10001111	`and rd’, rd’, rs2’`
`c.or`	OR	CS	01	10001110	`or rd’, rd’, rs2’`
`c.xor`	XOR	CS	01	10001101	`xor rd’, rd’, rs2’`
`c.sub`	SUB	CS	01	10001100	`sub rd’, rd’, rs2’`
`c.nop`	No OPeration	CI	01	000	`addi x0, x0, 0`
`c.ebreak`	Environment BREAK	CR	10	1001	`ebreak`

Figure 3: Encoding of base RVC instructions



RV32	RV64	Mnemonic
✓	✓	c.nop
✓	✓	c.addi
	✓	c.addiw
✓	✓	c.lui
✓	✓	c.srli
✓	✓	c.srai
✓	✓	c.andi
✓	✓	c.sub
✓	✓	c.xor
✓	✓	c.or
✓	✓	c.and
	✓	c.subw
	✓	c.addw
✓	✓	c.slli
✓	✓	c.mv
✓	✓	c.add

Figure 4: Compressed instruction compatibility table

To handle pipeline hazards effectively, the CPU incorporates a forwarding system that mitigates data hazards by forwarding intermediate data directly between pipeline stages. Furthermore, a branch hazard detection unit was implemented to detect and correctly handle control hazards caused by branch instructions. Together, these features contribute to improving pipeline throughput and correctness under typical program execution scenarios.

Our implementation balances completeness in instruction support and robust hazard management, reflecting both the complexity and practical considerations of a modern pipelined RISC-V CPU design.

Architecture and Design

Our CPU implements a three-stage pipeline with every-other-cycle issuing, designed specifically to operate with a single, unified, byte-addressable memory that serves both instructions and data. This memory constraint is crucial: it means the CPU cannot fetch an instruction and access data in the same clock cycle, so the pipeline scheduling must carefully separate these events.

Memory Organisation

Single-Ported, Byte Addressable Memory:

Both instructions and data are stored in a unified memory block. Addresses refer to bytes—so word accesses require appropriate alignment and address calculation. The pipeline must ensure instruction fetches and data operations never conflict.

Pipeline Stages

Walkthrough:

The pipeline consists of three main stages, each spread over two clock cycles:

- ***Stage 0: Instruction Fetch (C0) and Register Read (C1)***
- ***Stage 1: ALU Operation (C0) and Memory Access (C1)***
- ***Stage 2: Register Write Back (C0)***

1. Instruction Fetch (IF) — Stage 0, C0

- On the first clock cycle of the stage, the CPU fetches the next instruction from the unified memory at the address held in the PC (program counter).
- The fetched instruction and the current PC are stored into the IF/ID pipeline register so they're ready for the next phase.
- This ensures only one memory operation occurs per cycle, avoiding read conflicts.

2. Instruction Decode (ID) — Stage 0, C1

In the second clock cycle, the instruction is decoded:

- The control unit generates the appropriate control signals for downstream stages, such as ALU operation codes, memory read/write enables, register write enables, and multiplexor selectors.
- The register file is accessed: source operands (RS1, RS2) are read out, and the destination register index (RD) is identified.
- The immediate generator produces the immediate value for instructions that require it.
- If the instruction is compressed, the decompression unit first expands it to standard 32 bits.
- All decoded values, control signals, and register reads are latched into the ID/EX pipeline register.

3. Execute (EX) — Stage 1, C0

- The ALU performs arithmetic or logical operations, or computes target addresses for memory and branches:
- Operand selection uses multiplexers to choose, based on the type of instruction and forwarding control, between:
- The register read value, forwarded data (to resolve hazards), or the PC for operand A.
- The register read value, forwarded data, or the immediate value for operand B.
- The ALU produces calculation results and updates flags (zero, carry, overflow, sign) needed for branch decisions.
- The branch target is also calculated, and branch conditions are evaluated.
- Results, control signals, and calculated addresses are stored in the EX/MEM pipeline register.

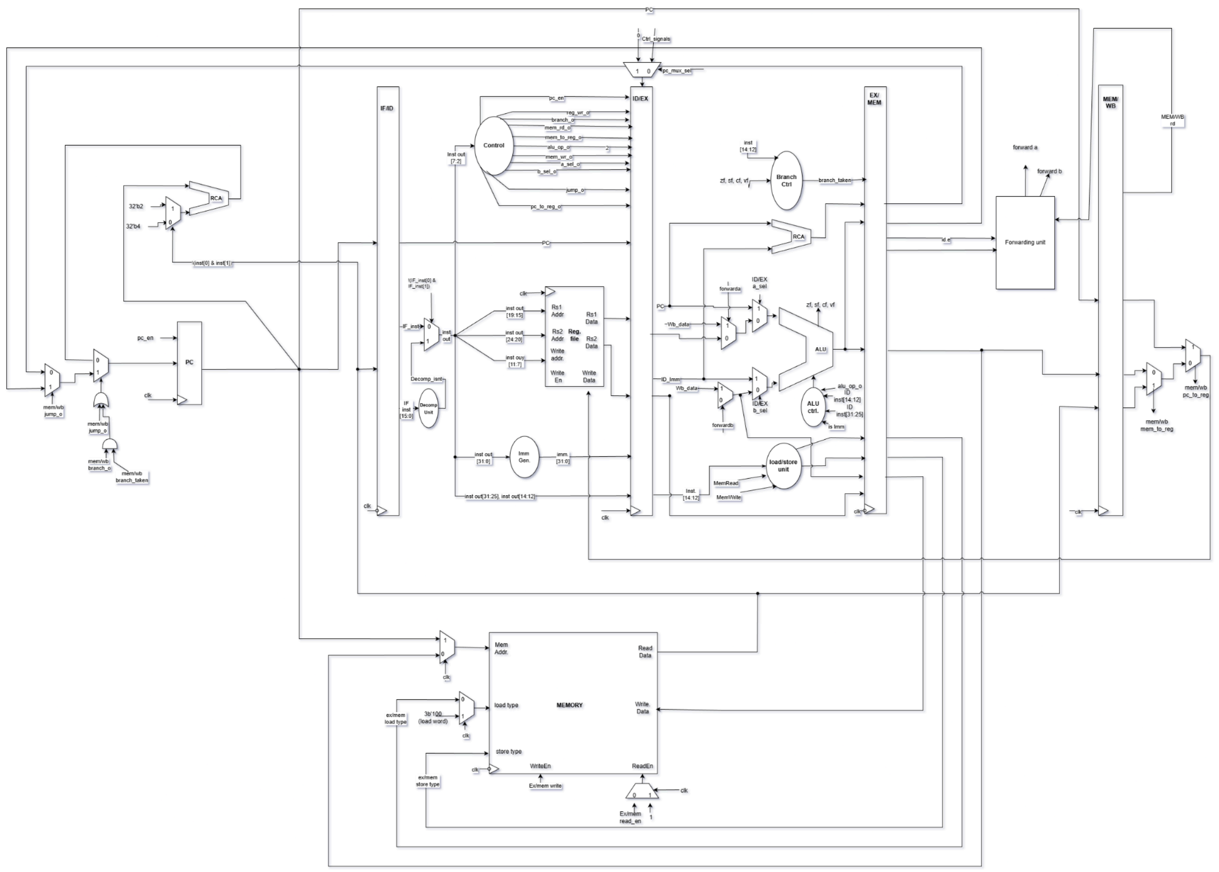
4. Memory Access (MEM) — Stage 1, C1

- Loads and stores are performed:
- For a store, data from the pipeline register is written to the byte-addressable unified memory at the computed address.
- For a load, memory is read at the computed address and the result is passed forward.
- Only one memory access (instruction or data) ever takes place per cycle due to unified memory.
- Loaded data, ALU results, and control signals are stored in the MEM/WB pipeline register.

5. Write Back (WB) — Stage 2, C0

- The result to be written back to the registers is selected:
- This can be the output of the ALU, data loaded from memory, or PC+4 for certain jumps.
- Multiplexers are used to choose among these, guided by the instruction type.
- The selected value is written into the destination register if register writing is enabled.
- This completes the instruction's journey through the pipeline, and a new instruction can enter the fetch stage every other cycle (ensured by the pipeline register/timing design).

Block Diagram:



Hazard handling strategies

Structural hazards

Structural hazards occur when the hardware cannot support all possible combinations of instructions in simultaneous pipeline stages, often due to resource conflicts such as multiple requests to the same memory or ALU at the same time. In our implementation, the main structural hazard arises from the use of a single-port, byte-addressable memory shared for both instruction fetch and data access. To resolve this, we designed the CPU to issue each instruction every other cycle, organising the pipeline so that instruction fetch and data memory operations never overlap in the same cycle. By carefully scheduling the stages and using pipeline registers latching on alternating clock edges, we ensure that only one memory access—either instruction or data—occurs per cycle, completely eliminating structural hazards at the cost of increased CPI.

Data hazards

Data hazards happen when one instruction depends on the result of a previous instruction that has not yet completed its passage through the pipeline (such as read-after-write or "RAW" hazards). In our pipeline, both arithmetic and load-use data hazards are solved primarily through a forwarding mechanism: data computed in the MEM stage is routed directly to where it is needed for subsequent instructions, bypassing the need to wait for it to be written back to the register file. Notably, because our pipeline's timing means there is no overlap between a prior instruction's execution stage and the current instruction's decode stage, forwarding only needs to source from the MEM stage, not EX—greatly simplifying the logic. For load-use hazards, where the value required is not available until after the memory access, our hazard detection logic inserts a stall (bubble) to pause the dependent instruction, ensuring safe and correct operation without data corruption.

Control hazards

also known as branch hazards, arise when the outcome of a branch or jump instruction is not known early in the pipeline, leading to the potential for fetching and partially executing instructions along the wrong control path. In our architecture, the branch outcome is resolved at the MEM stage. At most, one instruction following the branch will have been fetched and decoded (in the ID stage) by the time the branch decision is known. When a branch is taken, we simply flush the ID/EX pipeline register—effectively cancelling the in-flight instruction that would otherwise be invalid—while leaving the other pipeline registers (IF/ID, EX/MEM) untouched. This targeted flush mechanism ensures correct control flow with minimal pipeline disruption, as no further instructions need to be discarded.

Implementation Details

Module-by-Module Overview

1. *cpu* (Top-Level Module)

Coordinates pipeline operation, bringing together all stages, hazard logic, and interface signals. Declares and connects wires such as *pc_out_w, reg_read1_w, alu_out_w, mem_read_data_w*, and all pipeline registers like *if_id_pc_w, id_ex_reg1_w, ex_mem_alu_out_w,* etc. Integrates all submodules for datapath, control, memory, hazard handling, and display output.

2. Pipeline Registers: *register*

Implements all pipeline latches (e.g., *if_id_reg, id_ex_reg, ex_mem_reg, mem_wb_reg*). Each saves bundles of signals—such as *{if_id_pc_w, if_id_inst_w}*—between pipeline stages on alternating clock edges. This avoids single-port memory conflicts and realizes every-other-cycle issuing.

3. Basic Datapath Elements: *rca, nmux*

- *rca* (Ripple-Carry Adder) is used for address calculations, such as PC increment (*following_pc_adder* for *pc_step_w*, *offset_adder* for *pc_branch_w*).
- nmux is a parameterized multiplexer; used for selecting PC increment (*pc_add_mux*), determining operand sources for the ALU (*alu_a_2_mux*, *alu_b_2_mux*), and choosing between write-back data sources (*mem_to_reg_mux*, *pc_to_reg_mux*).

4. Program Counter: *register*

The PC register (*pc*) holds the address of the next instruction (*pc_out_w*) and is updated via logic involving *pc_in_w*, based on sequential, branch, or jump control.

5. Memory: *data_mem*

Implements unified, byte-addressable RAM, accessed through *addr_i*, supporting reads/writes of 8, 16, or 32 bits (using *store_type_i, load_type_i*). Both instruction fetch and data operations go through this module (*mem_read_data_w* supplies IF/ID and load results).

6. Register File: *reg_file*

Implements a 32-register file. It supports two simultaneous read ports and one write port with synchronous write and asynchronous reset functionality.

7. ALU and Control Logic

- *alu_control* converts *alu_op_i, funct3_i / funct7_i,* and isimm_i to an ALU command (*alu_ctrl_w*).
- *alu* receives inputs via carefully selected sources (*alu_in_a_w, alu_in_b_w*) and produces the computation output (*alu_out_w*) and relevant flags (*cf_w, zf_w, vf_w, sf_w*).
- Operand routing is handled by muxes (*alu_a_2_mux, alu_a_1_mux, alu_b_2_mux, alu_b_1_mux*), reflecting forwarding and instruction type.

8. Immediate Generator: *imm_gen*

Given *inst_final_w, imm_gen* outputs *imm_w*, the sign-extended or zero-extended offset/parameter field required for immediate-type instructions.

9. Decompression: *decomp_unit*

Accepts compressed instructions from *if_id_inst_w[15:0]* and outputs 32-bit RISC-V instructions (*decomp_inst_w*). Selected via the *inst_mux* for later stages.

10. Control Unit: *control_unit*

Decodes *opcode_i* from *inst_final_w* to produce all main pipeline control signals (*branch_w, mem_read_w, mem_write_w, alu_op_w, reg_write_w,* etc.), which flow through pipeline registers.

11. Branch Logic: *branch_control*

Combines ALU flags and instruction encoding to compute *take_branch_w*, controlling branch/jump PC updates.

12. Load/Store Type: *load_store_unit*

Generates *load_type_w* and *store_type_w* signals for load and store instructions, ensuring correct access width and sign extension in *data_mem.*

13. Forwarding and Hazard Detection: *forwarding_unit*

Receives *id_ex_rs1_w, id_ex_rs2_w*, destination index from MEM/WB *(mem_wb_rd_w)*, and its write enable (*mem_wb_ctrl_w[0]*). Outputs *forwarda_w, forwardb_w* to drive operand selection muxes for hazard resolution.

14. Top Module (*top*):

The top module is the main entry point for FPGA synthesis and board integration. It wires up all CPU core and peripheral modules, exposing standard board-level I/O such as (*rclk*), reset, LED selection inputs (*ledSel*), and seven-segment display selection (*ssdSel*). Internally, it links the CPU's pipeline and state with user-visible debugging outputs, ensuring seamless operation on the Nexys A7 board.

15. Clock Divider (*clk_divider_2*):

The system clock from the FPGA board (*rclk*) typically runs much faster than what's required for visibility or real-time control. The *clk_divider_2* module reduces this frequency, producing a slower clock (*sclk*) suitable for a button-simulated clock.

16. Seven-Segment Display Driver (*four_digit_seven_segment_driver*):

To visually observe and debug CPU internals, the output from the CPU's *ssd_o* (a 13-bit value encoding select state or register/memory data) is fed to the SSD driver. The module accepts the clock (*clkssd*), formats the data (*num_i*), multiplexes it across anodes (anode), and decodes it to segment outputs (*led_out*). This makes it possible to observe CPU state, PC, register values, or other chosen pipeline data directly on the FPGA's seven-segment display, selectable by switches.

Testing and Verification

Our CPU was verified using a range of focused test files, each designed to cover different architectural features and hazard scenarios. For each, we observed simulation waveforms and register/memory values to confirm correct operation.

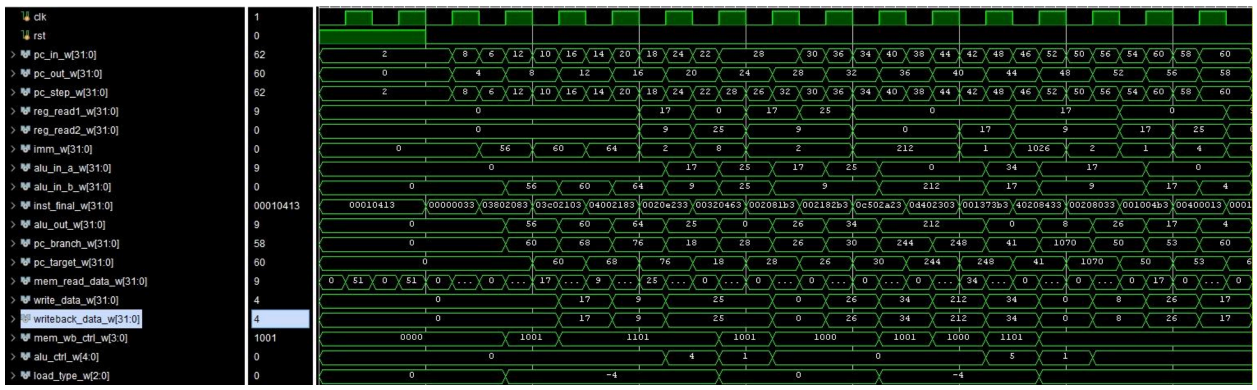
lab.mem

Purpose

A comprehensive standard instruction test based on the lab. Covers some base instructions, including arithmetic, logic, memory, and branch operations. The memory layout and instruction flow were crafted to exercise load-use data hazards and to confirm that the forwarding unit and stall logic behave as intended.

Hazards Tested

- Arithmetic data hazards (back-to-back register dependencies)
- Load-use hazards (instruction immediately following a load)
- Branch hazard (simple taken and not-taken scenarios)



Result

The waveform illustrates the correct execution of the load–use sequence and forwarding logic for this simple program. The initial three lw instructions properly fetch the values at addresses 56, 60, and 64 into x1, x2, and x3, with the memory read data matching the expected values. The subsequent or x4, x1, x2 uses forwarded operands from the preceding loads, demonstrating that the ALU receives the correct values before they reach the register file. The beq x4, x3, 4 also shows proper operand forwarding, evaluating the equality condition using the freshly computed OR result. Because the branch is taken, the next instruction (add x3, x1, x2) is flushed, which is clearly visible in the control signals (mem_wb_ctrl = 1000), where the least significant bit — the register-write enable — is zero, preventing the incorrect writeback of the value 26. After the flush, execution resumes normally: the ADD, SW, and LW instructions that follow all execute with the correct forwarded values, and the waveform confirms that the store writes x5 to memory at offset 12, and that the final LW retrieves this stored value correctly

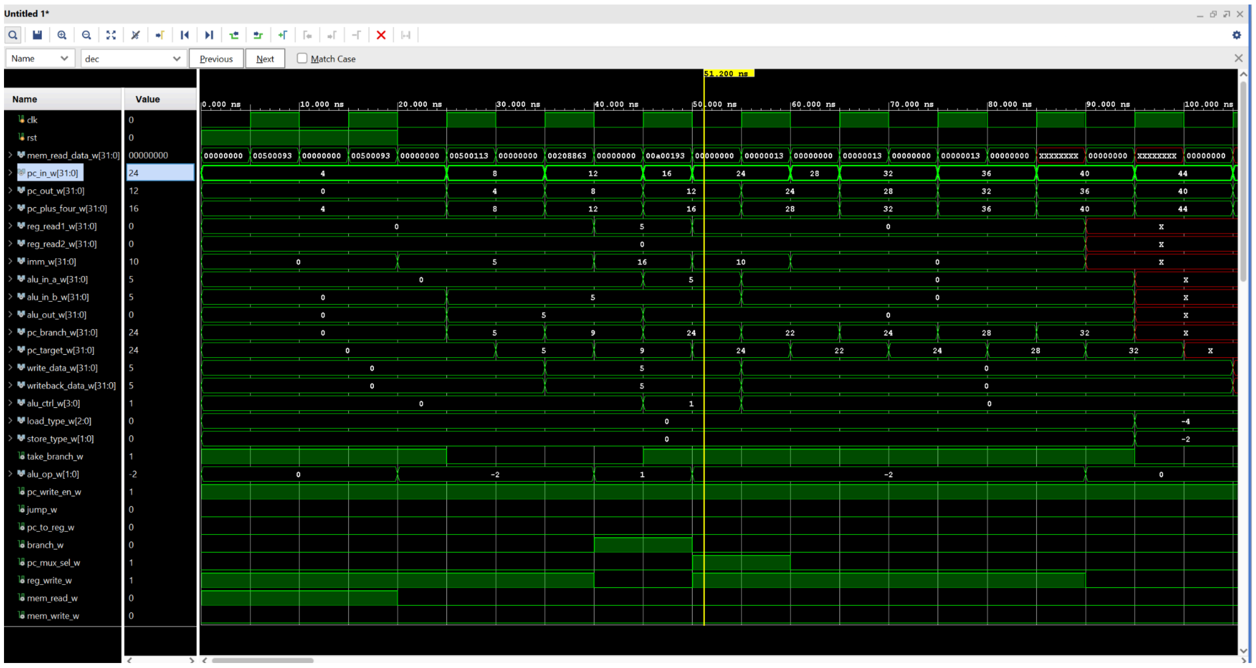
branchflush.hex

Purpose

Specifically tests branch (control) hazards. Contains one branch instruction designed to be taken.

Hazards Tested

Control hazard (branch taken and not-taken)



Name	Value
clk	0
rst	0
mem_read_data_w[31:0]	00000000
pc_in_w[31:0]	24
pc_out_w[31:0]	12
pc_plus_four_w[31:0]	16
reg_read1_w[31:0]	0
reg_read2_w[31:0]	0
imm_w[31:0]	10
alu_in_a_w[31:0]	5
alu_in_b_w[31:0]	5
alu_out_w[31:0]	0
pc_branch_w[31:0]	24
pc_target_w[31:0]	24
write_data_w[31:0]	5
writeback_data_w[31:0]	5
alu_ctrl_w[3:0]	1
load_type_w[2:0]	0
store_type_w[1:0]	0
take_branch_w	1
alu_op_w[1:0]	-2
pc_write_en_w	1
jump_w	0
pc_to_reg_w	0
branch_w	0
pc_mux_sel_w	1
reg_write_w	1
mem_read_w	0
mem_write_w	0

Result:

The branch is resolved at MEM, and the only instruction in the ID/EX register when a branch is taken is flushed. No false flushes or pipeline bubbles are introduced elsewhere.

Video explanation:

https://drive.google.com/file/d/1uqKoR4fMtulk_rBXelfMPA9MLPBznNbd/view?usp=sharing

mextensiontest.hex

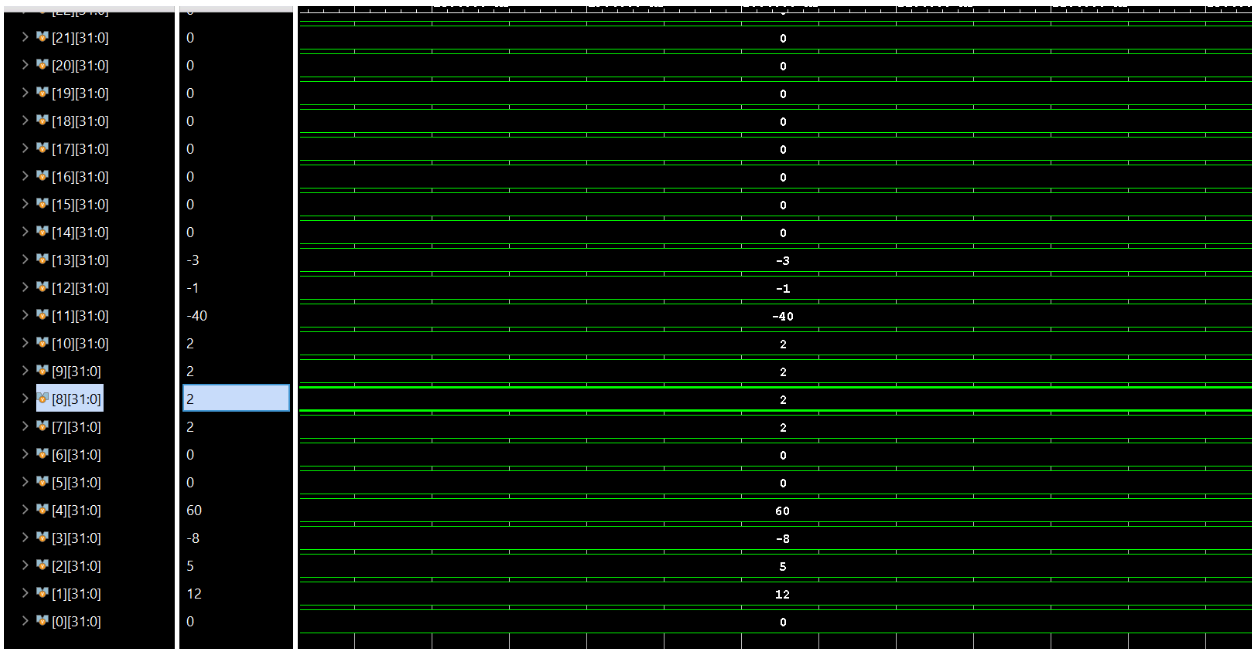
Purpose:

Validates support for the RISC-V “M” extension (integer multiplication/division). Exercises all MUL, MULH, MULHU, MULHSU, DIV, DIVU, REM, and REMU instructions, checking results across a variety of signed and unsigned cases.

Hazards Tested:

Chained multiply/divide instructions for dependency hazards

Data forwarding from MEM for multi-cycle arithmetic



[21][31:0]	0	0
[20][31:0]	0	0
[19][31:0]	0	0
[18][31:0]	0	0
[17][31:0]	0	0
[16][31:0]	0	0
[15][31:0]	0	0
[14][31:0]	0	0
[13][31:0]	-3	-3
[12][31:0]	-1	-1
[11][31:0]	-40	-40
[10][31:0]	2	2
[9][31:0]	2	2
[8][31:0]	2	2
[7][31:0]	2	2
[6][31:0]	0	0
[5][31:0]	0	0
[4][31:0]	60	60
[3][31:0]	-8	-8
[2][31:0]	5	5
[1][31:0]	12	12
[0][31:0]	0	0

Expected Results:

x4 = 60 (0x3C)

x5 = 0 (upper bits of small multiply)

x6 = 0 (upper bits unsigned)

x7 = 2 (12 / 5)

x8 = 2 (12 / 5 unsigned)

x9 = 2 (12 % 5)

x10 = 2 (12 % 5 unsigned)

x11 = -40 (0xFFFFFFD8)

x12 = -1 (0xFFFFFFFF, -8 / 5 rounds toward zero) x13 = -3 (0xFFFFFFFD, remainder)

decomp.mem

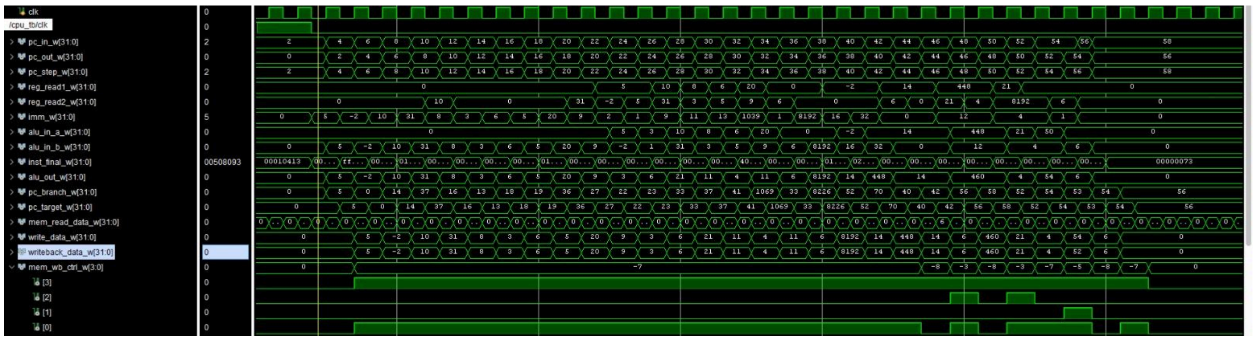
Purpose

Tests basic compressed instruction (RVC) decompression, including control flow and data operations in compressed form. Focus is on correct expansion, operand fetch, and execution.

Hazards Tested

Compressed to standard instruction sequence conversion/order

Data hazards between compressed instructions



Result

The waveform shows the execution of the compressed-instruction test starting with a series of c.addi instructions that initialize registers x1–x15 with the expected immediate values. Each instruction is correctly decompressed, the ALU receives the proper operands, and the writeback bus updates the registers in order. After initialization, the arithmetic operations execute sequentially: c.add, c.slli, c.xor, c.or, c.and, and c.sub, each producing the expected results visible on alu_out_w and then on the writeback stage. This is followed by c.mv and c.lui, which update x3 and x4, and then the stack-related instructions (c.addi16sp, c.addi4spn, c.swsp, c.lwsp) which correctly modify the stack pointer and perform memory transfers—confirmed by activity on the memory data signals and the restored values in x2 and x10. The final ALU operations (c.srli and c.andi) also commit correct results.

When c.j 4 executes, the PC redirects to the jump target and the pipeline flushes the following instruction (c.addi x6, 1), visible as disabled control signals and no writeback for x6. After the jump, c.nop passes through with no effect, and the program cleanly halts at c.ebreak, after which the PC stops advancing and no further register or memory activity occurs. The waveform therefore confirms correct decompression, ALU behavior, memory access, pipeline flushing, and termination.

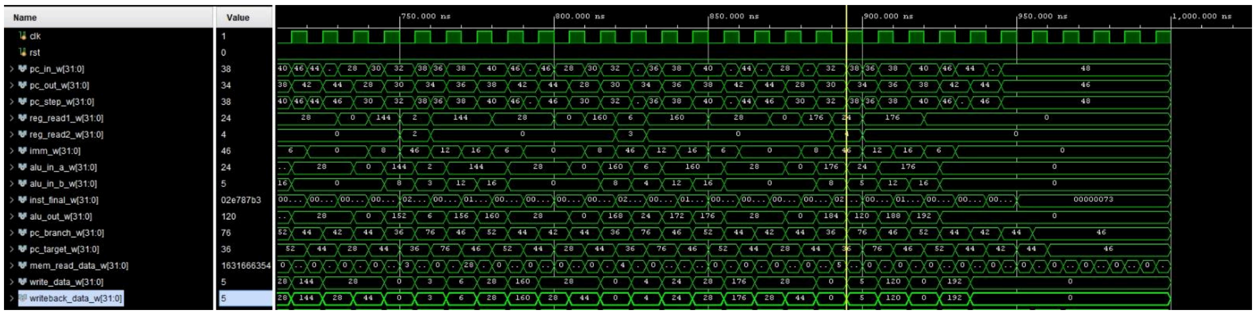
factorial.hex

Purpose

Executes a factorial routine using primarily compressed RVC instructions. Designed to test compressed instruction flow, register reuse, memory accesses within loops, and branch hazard handling in a practical program context.

Hazards Tested

- RVC instruction decompression within loops
- Data and control hazards in a real-world loop/body
- Interaction of compressed loads/stores and jumps



Name	Value
clk	1
rst	0
pc_in_w[31:0]	38
pc_out_w[31:0]	34
pc_step_w[31:0]	38
reg_read1_w[31:0]	24
reg_read2_w[31:0]	4
imm_w[31:0]	46
alu_in_a_w[31:0]	24
alu_in_b_w[31:0]	5
inst_final_w[31:0]	02e787b3
alu_out_w[31:0]	120
pc_branch_w[31:0]	76
pc_target_w[31:0]	36
mem_read_data_w[31:0]	1631666354
write_data_w[31:0]	5
writeback_data_w[31:0]	5

Result:

The waveform shows the execution of the recursive factorial program, which mixes compressed instructions, standard 32-bit RISC-V instructions, and the MUL operation. Execution begins by loading the input value n = 5 into x15 from memory (address 128), followed by initialization of the stack pointer to sp = 192 using c.addi16sp. The base-case registers x14 = 1 and x13 = 1 are set with c.li. The bge instruction correctly detects that n ≥ 1, so the program skips the base-case block and enters the recursive path. The waveform clearly shows the alternating PC increments of +2 for compressed instructions and +4 for full 32-bit instructions, confirming that decompression and fetch alignment work correctly.

During each recursive call, the function allocates stack space with c.addi16sp -16, saves the return address (x1) and the current value of n using c.swsp, decrements x15, and jumps back to the start using c.jal -20. In the waveform, the stack pointer decreases by 16 every call and increases back by 16 in each return, showing correct stack-frame management. The c.jr x1 instruction correctly restores control flow to the caller on every unwind step, and mul receives the proper operands, producing the intermediate and final multiplication results.

At the end of the final return, the waveform shows x15 = 120, confirming that the CPU computed 5! = 120 correctly. The stack pointer is restored back to sp = 192, proving that all frames were properly deallocated. Execution terminates cleanly at the c.ebreak instruction.

Bonus Features

RVC (Compressed Instruction Extension):

The RVC extension allows our processor to execute 16-bit compressed instructions as defined in the RISC-V standard. Compressed instructions decrease code size by roughly 25-30%, resulting in improved cache utilization and reduced memory bandwidth, which is particularly valuable in resource-constrained or embedded environments. In our pipeline, compressed instructions are detected in the fetch stage and expanded to their full 32-bit equivalent using the decomp_unit before decoding, thus integrating seamlessly with the rest of the datapath. Supported instructions include compressed loads/stores, stack pointer operations, register moves, and basic arithmetic, among others.

RVM (Multiply/Divide Extension):

The RVM extension adds hardware support for integer multiplication and division instructions (MUL, MULH, MULHU, MULHSU, DIV, DIVU, REM, and REMU). These instructions accelerate various computational workloads, enabling faster arithmetic beyond simple addition and subtraction. We integrated these into our ALU path and ensure all additional funct3/funct7 encodings are correctly handled by the alu_control module.

Conclusion

Through this project, we built a RISC-V pipelined CPU supporting RV32I alongside RVM (multiply/divide) and RVC (compressed) extensions. Tackling the challenges of single-port, byte-addressable memory, hazard management, and pipeline control required us to think creatively about architecture and timing. By implementing every-other-cycle instruction issue, forwarding logic, and targeted register flushing, we achieved robust instruction support and handled the full range of arithmetic, memory, and branching operations with correctness and efficiency.

Collaboration and regular testing were crucial to integration and debugging. Our experience deepened our understanding of real-world datapath design, modular hardware structure, and ISA extensibility. While our choice of memory architecture raises CPI, the project offers a strong basis for further enhancements—such as multi-port memory, extended instruction sets, or advanced hazard solutions—making it suitable for embedded applications and future optimisation work.

References

https://docs.riscv.org/reference/isa/_attachments/riscv-unprivileged.pdf

https://www.inf.ufpr.br/dagoliveira/ci1210/RV32C_reference_card.pdf

https://riscv.github.io/riscv-isa-manual/snapshot/unprivileged/#_rvc_compressed