CPU Roadmap: v1 → v5

Emulator-first CPU + Assembler + OS/Toolchain

Contents

1 Version Comparison (v1–v5)

This table compares what each CPU version adds on top of the previous version. All versions keep: 64-bit GPRs, 16 registers, 64KB byte-addressed memory, little-endian encoding, and fixed 8-byte instructions.

From v3 onward, addresses 0x0000..0x00FF are reserved for the vector table, and reset begins at PC=0x0100.

Category	v1	v2	v3	v4	v5
Primary goal	Minimum viable CPU to run programs in emulator; future-compatible contract	Add stack + CALL/RET + CMP for multi-function programs	Add syscalls/traps + MMIO console + timer + IEEE-754 float ops	Add base+offset addressing + wider loads/stores + compiler-friendly integer ops	Add indirect jumps/calls + protection primitives for advanced runtimes
Registers / width	16 GPRs, 64-bit	Same	Same	Same	Same
Special regs	PC, FLAGS(Z)	PC, SP, FP, FLAGS(Z)	PC, SP, FP, EPC, FLAGS(ZNVC), ICOUNT, TIMER_-PERIOD	Same as v3	Same as v4 + CAUSE, BADADDR + FLAGS(I,K)
Flags	Z only	Z only	ZNVC (integer ALU)	ZNVC (includes bitwise/shift ops)	ZNVC + I (interrupt enable) + K (kernel mode)
Instruction encoding	8 bytes: [opc,rd,ra,rb,imm32]	Same	Same	Same	Same
Memory ops	LOAD8_ABS / STORE8_ABS	Same	Same (plus MMIO semantics)	LOAD/STORE 8/16/32/64 ABS and BO (base+offset) + alignment enforcement	Same + protection checks (user/kernel region)
Addressing modes	Immediate + absolute imm16 + PC-relative (rel)	Same	Same	Adds base+offset: [Ra + imm32]	Same
Stack model	Optional / minimal	Present. PUSH8/POP8 (byte). CALL/RET push/pop 8-byte return addresses; alignment enforced	Same	Same + base+offset makes frames practical	Same
Control flow	Direct jumps/branches (ABS/REL), Z-based conditional	Same + CALL/RET	Same + SYSCALL/IRET via vectors	Same	Adds indirect: JMP_R, JZ_R, JNZ_R, CALL_R (targets in reg)

Category	v1	v2	v3	v4	v5
Indirect control flow	No	No	No	No	Yes (v5-only), with target validation + trap on bad target
MMIO / console	No	No	Yes: MMIO region 0xFE00–0xFFFF; console out/status/in	Yes (byte-width MMIO; wider MMIO disallowed)	Yes, kernel-only; user uses syscall
Syscalls / traps	No	No	Yes: SYSCALL (syscall# in R0), EPC + IRET; vector table	Same	Same + TRAP + richer trap causes
Interrupts / timer	No	No	Yes: deterministic timer interrupt	Same	Same + maskable via I
Floating point	No	No	Yes: FADD/FSUB/FMUL, IEEE-754 in GPRs	Same	Same
Protection model	None	None	None	None	Minimal user/kernel split; kernel-only system/MMIO; traps on violations
Exception handling	Halt on fault + diagnostics	Same	Adds traps (syscall/interrupt) using EPC + vectors	Same	Adds CAUSE/BADADDR; traps for protection + bad targets; halt only on fatal faults
Toolchain readiness	Assembler + emulator baseline	Multi-function programs	OS-like services and I/O	Compiler-friendly mem-ory/addressing	Function pointers/jump tables + isolation primitives

2 How To Make: CPU v1

1. Overview

CPU v1 is the minimum viable CPU needed to unlock the rest of the project (assembler first, then higher layers later). It is an emulator-first design: deterministic, test-driven, and intentionally small.

v1 Goals (frozen).

- Execute simple programs with arithmetic, byte memory access, and control flow.
- Provide a stable binary encoding target for the assembler.
- Keep the design forward-compatible with v2–v5 upgrades without breaking v1 binaries.

Out of scope in v1 (reserved for later versions).

- Stack operations (PUSH/POP), function calls (CALL/RET) (v2+).
- Syscalls/traps/interrupts, OS interface, MMIO (v3+).
- Floating point (IEEE-754) (v3+).
- Wider memory operations beyond byte (LOAD16/32/64, etc.) (v4+).

2. Frozen CPU Contract

This section is the contract that must not change for v1 binaries to remain valid.

Machine widths.

- **Register width:** 64-bit unsigned storage (two's complement interpretation for signed immediates/offsets where specified).
- **Register count:** 16 general-purpose registers `R0..R15`.
- **Address space:** 64KB RAM, byte-addressed, valid addresses `0x0000..0xFFFF`.

Special registers (present in v1).

- **PC** (Program Counter): 16-bit effective address into RAM; stored as a 64-bit value but must always remain within `0x0000..0xFFFF`.
- **FLAGS:** only the **Z** flag (Zero) exists in v1.
- **SP/FP:** *not used* in v1 and not required for correctness; for forward-compatibility, the emulator should still store an `SP` field initialized at reset (see Section 3), but no v1 instruction may read/write it.

Endianness and alignment.

- **Endianness:** Little-endian for multi-byte immediates in instruction encoding.
- **Forced alignment rule (defined now for future widths):** any memory access of width W bytes must satisfy `addr mod W == 0`. In v1, only $W = 1$ is used, so alignment never faults, but the rule is frozen.

Fault policy (v1). On any fault, the CPU **halts immediately** and produces a diagnostic record containing (at minimum): fault code, `PC`, opcode byte, decoded fields `rd/ra/rb/imm32`, and (if applicable) the offending memory address.

Determinism. Given identical initial CPU state and identical memory image, execution must produce identical final state and identical fault/trace results.

3. CPU State Definition

A minimal, explicit state model.

State fields.

- `R[16]`: array of 16 unsigned 64-bit registers.
- `PC`: unsigned 64-bit storage, but treated as a 16-bit effective address in the range `0x0000..0xFFFF`.
- `FLAGS.Z`: 1-bit boolean.
- `SP`: unsigned 64-bit storage, initialized for future use (v2+). Not used by v1 instructions.
- `MEM[65536]`: byte array representing RAM.
- `HALTED`: boolean.
- `HALT_REASON`: enum {`NORMAL`, `FAULT`}.
- `FAULT_INFO`: struct holding diagnostic fields when `HALT_REASON==FAULT`.

Reset convention.

- `PC = 0x0000`
- `FLAGS.Z = 0`
- `R[i] = 0` for all i
- `SP = 0xFFFF` (reserved for future versions; v1 never uses it)
- `HALTED = 0`

CPU state layout diagram.

```
+-------------------------------------------------------------+
| CPU STATE                                                   |
+----------------------+--------------------------------------+
| R0..R15 (16 x u64)   | General-purpose registers            |
| PC (u64; eff u16)    | Program counter (0x0000..0xFFFF)     |
| FLAGS.Z (1 bit)      | Zero flag                            |
| SP (u64)             | Reserved for v2+ (unused in v1)      |
| HALTED / REASON      | Execution status                      |
| FAULT_INFO           | Populated only on fault               |
+----------------------+--------------------------------------+
| MEM[65536] (bytes)   | Address space 0x0000..0xFFFF         |
+-------------------------------------------------------------+
```

4. Instruction Encoding (Binary Spec)

CPU v1 uses a fixed-length 8-byte encoding to keep decoding trivial and stable.

Byte layout (little-endian immediate). Each instruction is exactly 8 bytes:

```
           [ b0=opcode | b1=rd | b2=ra | b3=rb | b4..b7=imm32 (LE) ]

byte index:   0      1     2     3        4  5  6  7
            +------+-----+-----+-----+----------------+
meaning:     | OPC  | rd  | ra  | rb  |   imm32 (LE)   |
            +------+-----+-----+-----+----------------+
```

Field types.

- `opcode`: unsigned 8-bit.
- `rd, ra, rb`: unsigned 8-bit register indices. In v1, valid indices are `0..15`. Any other value is a fault.
- `imm32`: signed 32-bit two's-complement immediate, sign-extended to 64-bit when used as an integer.

Address derivation rules.

- **Absolute address (ABS):** `addr16 = imm32 & 0xFFFF`. Effective address is `addr16`.
- **PC-relative (REL):** `target := PC + imm32` (signed). If `target` is not in `0..0xFFFF`, or if `target+7 > 0xFFFF`, fault `PC_OOB`. If `target mod 8 != 0`, fault `MISALIGNED`. Otherwise `PC := target`.

PC increment rule. If an instruction does not explicitly modify `PC`, then `nextPC := PC + 8`. If `nextPC+7 > 0xFFFF`, fault `PC_OOB`. (`nextPC mod 8 == 0` always holds if `PC` was valid.) Otherwise `PC := nextPC`.

5. Fetch–Decode–Execute Loop

High-level pseudocode.

```
while not HALTED:
  # 1) Fetch
  if PC < 0 or PC+7 > 0xFFFF: fault(PC_OOB)
  if PC % 8 != 0: fault(MISALIGNED)
  instr[0..7] = MEM[PC .. PC+7]

  # 2) Decode
  opcode = instr[0]
  rd = instr[1]; ra = instr[2]; rb = instr[3]
  imm32 = sign_extend_le_u32(instr[4..7])  # into s64

  # 3) Execute
  execute(opcode, rd, ra, rb, imm32)

  # 4) If execute did not set PC explicitly:
  if pc_was_not_modified:
    PC = PC + 8
```

Decode invariants.

- Any register index used by the opcode must be in `0..15`.
- Any field declared *must-be-zero* for a specific opcode (see Section 6) must be zero; otherwise fault `ILLEGAL_ENCODING`.

6. ISA Specification (Instruction Truth Table)

Opcode map (v1).

Opcode	Mnemonic	Operands	Fields Used	Flags
0x00	HALT	–	opcode only	none
0x01	MOV_RI	rd, imm32	rd, imm32	none
0x02	MOV_RR	rd, ra	rd, ra	none
0x10	ADD	rd, ra, rb	rd, ra, rb	Z set
0x11	SUB	rd, ra, rb	rd, ra, rb	Z set
0x20	LOAD8_ABS	rd, [imm16]	rd, imm32	none
0x21	STORE8_ABS	[imm16], ra	ra, imm32	none
0x30	JMP_ABS	imm16	imm32	none
0x31	JMP_REL	imm32	imm32	none
0x32	JZ_ABS	imm16	imm32	reads Z
0x33	JZ_REL	imm32	imm32	reads Z

Reserved/Unused opcode policy (forward compatibility). Any opcode not listed above is **reserved** and must fault as `ILLEGAL_OPCODE` in v1.

Range	Reserved For (planned)
0x40–0x4F	Stack + CALL/RET (v2+)
0x50–0x5F	Syscalls / traps / interrupts (v3+)
0x60–0x6F	Wider loads/stores + base+offset addressing (v4+)
0x70–0x7F	Floating-point ops (v3+)
0x80–0xFF	Future expansion

Instruction Semantics (per instruction)

Common faults referenced below.

- `REG_OOB`: any required register field not in `0..15`.
- `MEM_OOB`: memory read/write address outside `0..65535`.
- `MISALIGNED`: alignment rule violated (e.g., `PC mod 8 != 0`; in v1 RAM accesses are 1 byte so only `PC` alignment matters).
- `PC_OOB`: `PC` fetch or next-`PC` out of range.
- `ILLEGAL_OPCODE`: opcode not defined in v1.
- `ILLEGAL_ENCODING`: must-be-zero fields violated.

HALT (0x00).

- **Operands/fields:** `rd=0, ra=0, rb=0, imm32=0` must hold or fault `ILLEGAL_ENCODING`.
- **Semantics:** `HALTED := 1; HALT_REASON := NORMAL`.
- **PC update:** none (execution stops).

- **Z flag:** unchanged.
- **Example:** `HALT`
  Bytes: `00 00 00 00 00 00 00 00`.

MOV_RI (0x01): `rd←signext(imm32)`.

- **Fields used:** `rd`, `imm32`. Must satisfy `ra=0, rb=0` or fault `ILLEGAL_ENCODING`.
- **Semantics:** `R[rd] := sign_extend_64(imm32)`.
- **PC update:** `PC := PC + 8`.
- **Z flag:** unchanged.
- **Faults:** `REG_OOB`.
- **Worked example:** `MOV_RI R1, 5`
  Bytes: `01 01 00 00 05 00 00 00`
  Effect: `R1=5`.

MOV_RR (0x02): `rd←ra`.

- **Fields used:** `rd`, `ra`. Must satisfy `rb=0, imm32=0` or fault `ILLEGAL_ENCODING`.
- **Semantics:** `R[rd] := R[ra]`.
- **PC update:** `PC := PC + 8`.
- **Z flag:** unchanged.
- **Faults:** `REG_OOB`.
- **Worked example:** If `R2=0xAA`, `MOV_RR R3, R2`
  Bytes: `02 03 02 00 00 00 00 00`
  Effect: `R3=0xAA`.

ADD (0x10): `rd←ra + rb`.

- **Fields used:** `rd`, `ra`, `rb`. Must satisfy `imm32=0` or fault `ILLEGAL_ENCODING`.
- **Semantics:** `tmp := (R[ra] + R[rb]) mod 2^64; R[rd] := tmp`.
- **Z flag:** `Z := (R[rd] == 0)`.
- **PC update:** `PC := PC + 8`.
- **Faults:** `REG_OOB`.
- **Worked example:** If `R1=2`, `R2=3`, `ADD R0,R1,R2`
  Bytes: `10 00 01 02 00 00 00 00`
  Effect: `R0=5`, `Z=0`.

SUB (0x11): `rd←ra - rb`.

- **Fields used:** `rd`, `ra`, `rb`. Must satisfy `imm32=0` or fault `ILLEGAL_ENCODING`.
- **Semantics:** `tmp := (R[ra] - R[rb]) mod 2^64; R[rd] := tmp`.
- **Z flag:** `Z := (R[rd] == 0)`.
- **PC update:** `PC := PC + 8`.
- **Faults:** `REG_OOB`.
- **Worked example:** If `R1=3`, `R2=3`, `SUB R0,R1,R2`
  Bytes: `11 00 01 02 00 00 00 00`
  Effect: `R0=0`, `Z=1`.

LOAD8_ABS (0x20): `rd←MEM[imm16]`.

- **Fields used:** `rd`, `imm32`. Must satisfy `ra=0, rb=0` or fault `ILLEGAL_ENCODING`.
- **Semantics:**
  1. `addr := imm32 & 0xFFFF`
  2. Read byte `b := MEM[addr]`
  3. `R[rd] := zero_extend_64(b)`
- **Z flag:** unchanged.
- **PC update:** `PC := PC + 8`.
- **Faults:** `REG_OOB`, `MEM_OOB` (if addr not in `0..65535`; in practice addr is 16-bit so only possible if implementation mis-checks).
- **Worked example:** If `MEM[0x0100]=0x41`, `LOAD8_ABS R1,[0x0100]`
  Bytes: `20 01 00 00 00 01 00 00`
  Effect: `R1=0x41`.

STORE8_ABS (0x21): `MEM[imm16]←ra&0xFF`.

- **Fields used:** `ra`, `imm32`. Must satisfy `rd=0, rb=0` or fault `ILLEGAL_ENCODING`.
- **Semantics:**
  1. `addr := imm32 & 0xFFFF`
  2. `MEM[addr] := (R[ra] & 0xFF)`
- **Z flag:** unchanged.
- **PC update:** `PC := PC + 8`.
- **Faults:** `REG_OOB`, `MEM_OOB`.
- **Worked example:** If `R1=0x48`, `STORE8_ABS [0x0200],R1`
  Bytes: `21 00 01 00 00 02 00 00`
  Effect: `MEM[0x0200]=0x48`.

JMP_ABS (0x30): PC←imm16.

- **Fields used:** imm32. Must satisfy rd=0, ra=0, rb=0 or fault ILLEGAL_ENCODING.
- **Semantics:** target := imm32 & 0xFFFF. If target mod 8 != 0, fault MISALIGNED. If target+7 > 0xFFFF, fault PC_OOB. Otherwise PC := target.
- **Z flag:** unchanged.
- **Faults:** PC_OOB (if target+7 > 0xFFFF), MISALIGNED, ILLEGAL_ENCODING.
- **Worked example:** JMP_ABS 0x0040
  Bytes: 30 00 00 00 40 00 00 00
  Effect: PC=0x0040.

JMP_REL (0x31): PC←PC + imm32.

- **Fields used:** imm32. Must satisfy rd=0, ra=0, rb=0 or fault ILLEGAL_ENCODING.
- **Semantics:** target := PC + imm32 (signed). If target is not in 0..0xFFFF, or if target+7 > 0xFFFF, fault PC_OOB. If target mod 8 != 0, fault MISALIGNED. Otherwise PC := target.
- **Z flag:** unchanged.
- **Faults:** PC_OOB, MISALIGNED, ILLEGAL_ENCODING.
- **Worked example:** If PC=0x0010, JMP_REL +16
  Bytes: 31 00 00 00 10 00 00 00
  Effect: PC=0x0020.

JZ_ABS (0x32): if Z then PC←imm16 else PC+=8.

- **Fields used:** imm32. Must satisfy rd=0, ra=0, rb=0 or fault ILLEGAL_ENCODING.
- **Semantics:** if Z==1 then PC := (imm32 & 0xFFFF) else PC := PC + 8.
- **Z flag:** unchanged (read-only).
- **Faults:** PC_OOB, ILLEGAL_ENCODING.
- **Worked example:** If Z=1, JZ_ABS 0x0080
  Bytes: 32 00 00 00 80 00 00 00
  Effect: PC=0x0080.

JZ_REL (0x33): if Z then PC←PC + imm32 else PC+=8.

- **Fields used:** imm32. Must satisfy rd=0, ra=0, rb=0 or fault ILLEGAL_ENCODING.
- **Semantics:** if Z==1 then PC := PC + imm32 (signed, bounds check) else PC := PC + 8.
- **Z flag:** unchanged (read-only).
- **Faults:** PC_OOB, ILLEGAL_ENCODING.
- **Worked example:** If Z=1 and PC=0x0100, JZ_REL -8
  Bytes: 33 00 00 00 F8 FF FF FF
  Effect: PC=0x00F8.

7. Memory Model, Endianness, and Alignment

RAM model.

- RAM is a byte array `MEM[65536]`.
- Valid addresses are `0..65535`.
- v1 provides only **byte** memory operations (LOAD8/STORE8).

Endianness.

- Instruction `imm32` is stored little-endian in bytes `b4..b7`.
- v1 does not load/store multi-byte integers from RAM; the chosen endianness is frozen for future versions.

Alignment (forced).

- **Instruction fetch:** `PC mod 8 == 0` must hold, else fault `MISALIGNED`.
- **Memory access:** for any access width W bytes, `addr mod W == 0` must hold, else fault `MISALIGNED`. In v1, $W = 1$ so RAM accesses never trigger this; it is still implemented now to keep behavior stable in v4+.

8. Control Flow Rules

Addressing styles supported in v1.

- **Absolute** jumps use `imm16 = imm32 & 0xFFFF`.
- **PC-relative** jumps add signed `imm32` to the current `PC`.

PC correctness requirements.

- `PC mod 8 == 0` must hold (each instruction begins on an 8-byte boundary).
- Fetch requires `PC+7 <= 0xFFFF`.
- Any branch target must satisfy `PC mod 8 == 0` and `PC+7 <= 0xFFFF`.

No indirect control flow. v1 does not allow `JMP Rk` or computed targets. Any opcode attempting indirect jumps is reserved and must fault.

9. Fault Model and Diagnostics

Fault codes (minimum set).

- `ILLEGAL_OPCODE`
- `ILLEGAL_ENCODING`
- `REG_OOB`
- `MEM_OOB`
- `PC_OOB`
- `MISALIGNED` (defined; unused by v1 instructions)

Diagnostic record format (recommended). On fault, populate:

- `fault_code`
- `pc` (value at time of fault)
- `opcode, rd, ra, rb`
- `imm32` (signed) and `imm32_hex`
- `addr` (if a memory fault occurred; else omit or set to `-1`)
- `message` (short human-readable string)

Fault example (invalid opcode). If memory at `PC` begins with opcode `0xFF`:

- fault: `ILLEGAL_OPCODE`
- include: `pc`, `opcode=0xFF`, decoded fields, imm32
- CPU halts with `HALT_REASON=FAULT`

10. Step-by-Step Implementation Plan

1. **Define state structs/classes.** Implement fields in Section 3 exactly. Add helpers for reading/writing registers and memory with bounds checks.
2. **Implement reset().** Ensure reset sets `PC=0`, `Z=0`, regs zeroed, `SP=0xFFFF`.
3. **Implement fetch().** Read 8 bytes from `MEM[PC..PC+7]` with `PC_OOB` checks.
4. **Implement decode().** Extract `opcode, rd, ra, rb, imm32`. Ensure little-endian imm decoding and sign-extension.
5. **Implement execute() dispatch.** Use a switch/match on opcode. For each instruction:
   - validate required registers in range,
   - validate must-be-zero fields,
   - perform semantics,
   - update `PC` per the rule.
6. **Implement fault().** Centralize halting + diagnostic population.
7. **Add a minimal tracer (optional but recommended).** For each step, record `PC`, decoded instruction, and changed registers. Keep it deterministic.
8. **Write tests as you add each opcode.** Do not implement multiple opcodes before testing at least one.

11. Testing Plan (Unit + Integration + Fault Tests)

Unit tests (per instruction). For each instruction, create a test vector:

- initial registers/memory/PC/Z
- instruction bytes at `PC`
- expected registers/memory/PC/Z after exactly one step

Integration tests (mini-programs). Load a full program image into memory, step until HALT, then assert final state.

Fault tests. At minimum:

- illegal opcode (`0xFF`)
- illegal encoding (e.g., `HALT` with nonzero fields)
- PC out of bounds (set `PC=0xFFFE` and attempt fetch)

Example test vector table (subset).

Name	Instr Bytes (hex)	Expected
mov_ri_r1_5	`01 01 00 00 05 00 00 00`	`R1=5, PC+=8`
add_sets_z	`10 00 01 02 00 00 00 00`	`R0=R1+R2, Z=(R0==0)`
halt_ok	`00 00 00 00 00 00 00 00`	`HALTED=1`

12. Upgrade Notes (Compatibility with v2..v5)

What v1 intentionally freezes.

- 8-byte fixed instruction size and byte-field layout.
- Little-endian immediate encoding.
- 16-register naming and indices (0–15).
- Halt-on-fault behavior and minimum diagnostics.

What later versions add without breaking v1.

- v2: introduce stack/call opcodes in reserved ranges (0x40–0x4F).
- v3: introduce traps/interrupts and optional MMIO mappings (0x50–0x5F).
- v4: add wider memory ops and additional addressing modes (0x60–0x6F).
- v3+/v5: add IEEE-754 floating ops in reserved range (0x70–0x7F) and optional protection/VM.

Compatibility rule. All v1 opcodes must keep their semantics forever. New features must use reserved opcodes or previously must-be-zero field interpretations that do not change v1 decoding.

13. Deliverables Checklist

By the end of CPU v1, you should have:

- `cpu_state.*`: state definition + reset.
- `decoder.*`: fetch + decode of 8-byte instructions.
- `executor.*`: opcode dispatch + semantics.
- `faults.*`: fault codes + diagnostic formatting.
- `tests_cpu_v1.*`: unit tests + integration tests + fault tests.

- `isa_v1.md` (or `isa_v1.tex`): opcode map and semantics reference (can be generated from the same source).
- `programs/v1/`: sample binaries and assembly sources for demos.

14. Appendix (Examples)

A. "Hello-style" demo (writes bytes to memory)

This demo writes ASCII `'H' 'I' '\0'` into RAM at `0x0200..0x0202` and halts.

Assembly (conceptual).

```
; Write "HI\0" into memory starting at 0x0200
MOV_RI   R1, 0x48          ; 'H'
STORE8_ABS [0x0200], R1
MOV_RI   R1, 0x49          ; 'I'
STORE8_ABS [0x0201], R1
MOV_RI   R1, 0x00
STORE8_ABS [0x0202], R1
HALT
```

Machine bytes (each line is 8 bytes).

```
01 01 00 00 48 00 00 00    ; MOV_RI R1, 0x48
21 00 01 00 00 02 00 00    ; STORE8_ABS [0x0200], R1
01 01 00 00 49 00 00 00    ; MOV_RI R1, 0x49
21 00 01 00 01 02 00 00    ; STORE8_ABS [0x0201], R1
01 01 00 00 00 00 00 00    ; MOV_RI R1, 0
21 00 01 00 02 02 00 00    ; STORE8_ABS [0x0202], R1
00 00 00 00 00 00 00 00    ; HALT
```

Expected final state (key observations).

- `MEM[0x0200]=0x48`, `MEM[0x0201]=0x49`, `MEM[0x0202]=0x00`
- `HALTED=1`, `HALT_REASON=NORMAL`

B. Control-flow demo (loop + conditional using Z)

This demo decrements a counter from 3 down to 0 and then halts.

Assembly (conceptual).

```
MOV_RI R1, 3        ; counter
MOV_RI R2, 1        ; decrement
loop:
SUB    R1, R1, R2   ; sets Z when R1 becomes 0
JZ_ABS done
JMP_ABS loop
done:
HALT
```

Machine bytes (assume loop label at 0x0010 and done at 0x0030).

```
01 01 00 00 03 00 00 00    ; 0x0000: MOV_RI R1,3
01 02 00 00 01 00 00 00    ; 0x0008: MOV_RI R2,1
11 01 01 02 00 00 00 00    ; 0x0010: SUB R1,R1,R2
32 00 00 00 30 00 00 00    ; 0x0018: JZ_ABS 0x0030
30 00 00 00 10 00 00 00    ; 0x0020: JMP_ABS 0x0010
00 00 00 00 00 00 00 00    ; 0x0030: HALT
```

Expected behavior. After three iterations, `R1==0` and `Z==1`, so `JZ_ABS` branches to `done` and halts.

C. Fault demo (illegal encoding)

Place a HALT instruction with a nonzero field and confirm `ILLEGAL_ENCODING`.

Faulting bytes.

```
00 01 00 00 00 00 00 00    ; HALT but rd=1 (must be 0) => fault
```

Expected diagnostic fields (minimum).

- `fault_code = ILLEGAL_ENCODING`
- `pc = address of the faulting instruction`
- `opcode = 0x00`, `rd=1`, `ra=0`, `rb=0`
- `imm32 = 0`

3 How To Make: CPU v2

1. Overview

CPU v2 extends CPU v1 by adding a **real stack** and **function calls** so we can run multi-function programs and prepare for the kernel/syscall interface in v3. CPU v2 remains emulator-first: deterministic, test-driven, and forward-compatible with v3–v5.

> **Compatibility note.** v2 adds `SP/FP` and 8-byte CALL/RET stack traffic, but does not change any v1 instruction encodings. Existing v1 binaries still run unchanged. The new 8-byte stack alignment rule applies only to v2 stack/call operations (`PUSH8/POP8/CALL_ABS/RET`).

v2 Goals (frozen).

- Add a downward-growing stack with deterministic PUSH/POP semantics.
- Add CALL/RET to support structured programs and recursion.
- Add CMP to enable branching decisions without clobbering registers.

Still out of scope in v2 (reserved).

- Syscalls/traps/interrupts/timer, privilege modes, MMIO behavior (v3+).
- Floating point (IEEE-754) (v3+).
- Wider RAM loads/stores beyond byte (v4+).
- Indirect/computed jumps (`JMP Rk`, `CALL Rk`) (v5 only).

2. Frozen CPU Contract

CPU v2 inherits all v1 frozen properties and adds a stack + call discipline.

Machine widths.

- **Register width:** 64-bit.
- **Register count:** 16 general-purpose registers `R0..R15`.
- **Address space:** 64KB RAM, byte-addressed, valid addresses `0x0000..0xFFFF`.

Special registers (present in v2).

- **PC:** program counter. Stored as 64-bit but must remain within `0x0000..0xFFFF` and always point to an instruction boundary (8-byte aligned).
- **SP:** stack pointer. Downward-growing stack; rules in Section 7. For data-movement convenience, `SP` may be referenced by certain instructions via a special register encoding (Section 4).
- **FP:** frame pointer. Optional but useful for stable stack frames. `FP` may be referenced by certain instructions via a special register encoding (Section 4).
- **FLAGS:** only **Z** (Zero) exists in v2.

Endianness and alignment.

- **Endianness:** Little-endian for multi-byte immediates in instruction encoding and for stack push/pop of 8-byte return addresses.
- **Forced alignment rule (frozen):** any memory access of width W bytes must satisfy `addr mod W == 0`. In v2, stack writes/reads of width 8 must satisfy 8-byte alignment; misalignment is a fault.

Fault policy. On any fault, the CPU **halts immediately** and emits a diagnostic record containing at minimum: fault code, `PC`, opcode byte, decoded fields `rd/ra/rb/imm32`, and (if applicable) the offending memory address and width.

Determinism. Given the same initial state and memory image, execution produces identical results and identical fault/trace output.

3. CPU State Definition

CPU v2 extends the v1 state with active stack and a frame pointer.

State fields.

- `R[16]`: array of 16 unsigned 64-bit GPRs.
- `PC`: u64 storage, effective address in `0..65535`.
- `SP`: u64 storage, effective address in `0..65535`.
- `FP`: u64 storage, effective address in `0..65535`.
- `FLAGS.Z`: 1-bit boolean.
- `MEM[65536]`: byte array RAM.
- `HALTED`, `HALT_REASON`, `FAULT_INFO`: as in v1.

Reset convention.

- `PC = 0x0000`
- `FLAGS.Z = 0`
- `R[i] = 0` for all i
- `SP = 0xFDFF` (top-of-stack below the reserved region `0xFE00..0xFFFF`)
- `FP = 0xFDFF` (recommended initial FP equals SP)
- `HALTED = 0`

CPU state layout diagram.

```
+-------------------------------------------------------------+
| CPU STATE (v2)                                               |
+----------------------+--------------------------------------+
| R0..R15 (16 x u64)   | General-purpose registers            |
| PC (u64; eff u16)    | Program counter (0x0000..0xFFFF)     |
| SP (u64; eff u16)    | Stack pointer (downward-growing)     |
| FP (u64; eff u16)    | Frame pointer (recommended for frames)|
| FLAGS.Z (1 bit)      | Zero flag                            |
| HALTED / REASON      | Execution status                     |
| FAULT_INFO           | Populated only on fault              |
+----------------------+--------------------------------------+
| MEM[65536] (bytes)   | RAM, 0x0000..0xFFFF                  |
| 0xFE00..0xFFFF       | Reserved for v3+ MMIO/system use     |
+-------------------------------------------------------------+
```

4. Instruction Encoding (Binary Spec)

CPU v2 keeps the v1 fixed-length instruction format for compatibility and simplicity.

Byte layout (little-endian imm32). Each instruction is exactly 8 bytes:

```
          [ b0=opcode | b1=rd | b2=ra | b3=rb | b4..b7=imm32 (LE) ]

byte index:   0      1     2     3        4  5  6  7
            +------+-----+-----+-----+----------------+
meaning:     | OPC  | rd  | ra  | rb  |   imm32 (LE)   |
            +------+-----+-----+-----+----------------+
```

Field types.

- `opcode`: u8.
- `rd, ra, rb`: register indices u8. In general, `0..15` select `R0..R15`. For a limited set of opcodes, `0x10` encodes `SP` and `0x11` encodes `FP` (see "Special register encoding" below). Any other value is `REG_OOB`.
- `imm32`: signed 32-bit two's-complement immediate, sign-extended to 64-bit when used.

Special register encoding (v2). To keep the encoding stable while exposing `SP` and `FP` for simple prologues/epilogues, v2 defines two additional register selector values:

- `0x10` selects `SP`.
- `0x11` selects `FP`.

Only instructions that explicitly say they allow these selectors may use them; all other opcodes treat `rd/ra/rb` values outside `0..15` as `REG_OOB`. When an instruction writes `SP` or `FP`, the written value must be in `0..0xFFFF`; otherwise `MEM_OOB`. (Mispositioning `SP` is allowed, but may cause `MISALIGNED` on later 8-byte stack operations.)

Absolute and relative addressing.

- **Absolute (ABS):** `addr16 = imm32 & 0xFFFF`.
- **PC-relative (REL):** `target := PC + imm32` (signed). If `target` is not in `0..0xFFFF`, or if `target+7 > 0xFFFF`, fault `PC_OOB`. If `target mod 8 != 0`, fault `MISALIGNED`. Otherwise `PC := target`.

PC default increment. If the instruction does not explicitly modify `PC`, then `nextPC := PC + 8`. If `nextPC+7 > 0xFFFF`, fault `PC_OOB`. Otherwise `PC := nextPC`.

5. Fetch–Decode–Execute Loop

The v2 loop is identical to v1, with additional opcodes.

High-level pseudocode.

```
while not HALTED:
  # Fetch
  if PC+7 > 0xFFFF: fault(PC_OOB)
  if PC % 8 != 0: fault(MISALIGNED)
  instr[0..7] = MEM[PC .. PC+7]

  # Decode
  opcode = instr[0]
  rd = instr[1]; ra = instr[2]; rb = instr[3]
  imm32 = sign_extend_le_u32(instr[4..7])

  # Execute
  execute(opcode, rd, ra, rb, imm32)

  # Default PC update
  if pc_was_not_modified:
    nextPC = PC + 8
    if nextPC+7 > 0xFFFF: fault(PC_OOB)
    PC = nextPC
```

Decode invariants.

- Any register index referenced by the opcode must be valid for that opcode. By default only `0..15` (`R0..R15`) are allowed; certain data-movement opcodes additionally permit `0x10` (`SP`) and `0x11` (`FP`) as stated in their semantics. Otherwise `REG_OOB`.
- Any must-be-zero fields for an opcode must be zero; otherwise `ILLEGAL_ENCODING`.

6. ISA Specification (Instruction Truth Table)

Opcode map (v2). CPU v2 includes all v1 instructions plus stack/call/CMP additions. Indirect jumps remain reserved until v5.

Opcode	Mnemonic	Operands	Fields Used	Z
0x00	HALT	–	opcode only	unchanged
0x01	MOV_RI	rd, imm32	rd, imm32	unchanged
0x02	MOV_RR	rd, ra	rd, ra	unchanged
0x10	ADD	rd, ra, rb	rd, ra, rb	set
0x11	SUB	rd, ra, rb	rd, ra, rb	set
0x12	CMP	ra, rb	ra, rb	set
0x20	LOAD8_ABS	rd, [imm16]	rd, imm32	unchanged
0x21	STORE8_ABS	[imm16], ra	ra, imm32	unchanged
0x30	JMP_ABS	imm16	imm32	unchanged
0x31	JMP_REL	imm32	imm32	unchanged
0x32	JZ_ABS	imm16	imm32	read
0x33	JZ_REL	imm32	imm32	read
0x40	PUSH8	ra	ra	unchanged
0x41	POP8	rd	rd	unchanged
0x42	CALL_ABS	imm16	imm32	unchanged
0x43	RET	–	opcode only	unchanged

Reserved/Unused opcode policy (forward compatibility).

Range	Reserved For (planned)
0x44–0x4F	Extended stack/frame helpers (optional v2+/v4+)
0x50–0x5F	Syscalls / traps / interrupts / timer (v3+)
0x60–0x6F	Wider loads/stores + base+offset addressing (v4+)
0x70–0x7F	Floating-point ops (v3+)
0x80–0xFE	Future expansion
0xFF	Never valid in v1/v2 (recommended illegal)

Important: later assignment. In this roadmap, v5 assigns `0x44` to `CALL_R` (indirect call). Treat the `0x44-0x4F` range as *reserved until explicitly assigned by a later version.* v2 code must still treat all of `0x44-0x4F` as illegal opcodes.

Instruction Semantics (per instruction)

Shared rules and faults.

- `REG_OOB`: required register field not valid for the opcode (default: not in `0..15`; some opcodes additionally allow `0x10`=`SP`, `0x11`=`FP`).
- `MEM_OOB`: memory address outside `0..65535`.
- `PC_OOB`: `PC` fetch/target outside range or not enough bytes to fetch.
- `MISALIGNED`: alignment rule violated (e.g. `PC mod 8 != 0`, or 8-byte stack alignment).
- `ILLEGAL_OPCODE`: undefined opcode.
- `ILLEGAL_ENCODING`: must-be-zero field violation.

HALT (0x00). Must satisfy `rd=ra=rb=0, imm32=0`; else `ILLEGAL_ENCODING`. Sets `HALTED=1, HALT_REASON=NORMAL`. PC not updated. Z unchanged. Example bytes: `00 00 00 00 00 00 00 00`.

MOV_RI (0x01): move immediate into `rd`.

- **Operands/fields:** uses `rd, imm32`. Must satisfy `ra=rb=0` or `ILLEGAL_ENCODING`.
- **Semantics:** let `val := sign_extend_64(imm32)`.

  - If `rd` is `0..15`: `R[rd] := val`.
  - If `rd==0x10`: write `SP := val`; require `0 <= SP <= 0xFFFF` else `MEM_OOB`.
  - If `rd==0x11`: write `FP := val`; require `0 <= FP <= 0xFFFF` else `MEM_OOB`.
  - Otherwise: `REG_OOB`.
- **PC update:** apply the default PC increment rule.
- **Z flag:** unchanged.
- **Faults:** `REG_OOB`, `MEM_OOB`, `ILLEGAL_ENCODING`, `PC_OOB`.
- **Example:** `MOV_RI R1, 5`
  Bytes: `01 01 00 00 05 00 00 00` ⇒ `R1=5`.

MOV_RR (0x02): move register into `rd`.

- **Operands/fields:** uses `rd, ra`. Must satisfy `rb=0, imm32=0` or `ILLEGAL_ENCODING`.
- **Source value:**
  - If `ra` is `0..15`: `val := R[ra]`.
  - If `ra==0x10`: `val := SP`.
  - If `ra==0x11`: `val := FP`.
  - Otherwise: `REG_OOB`.
- **Destination write:**
  - If `rd` is `0..15`: `R[rd] := val`.
  - If `rd==0x10`: `SP := val`; require `0 <= SP <= 0xFFFF` else `MEM_OOB`.
  - If `rd==0x11`: `FP := val`; require `0 <= FP <= 0xFFFF` else `MEM_OOB`.
  - Otherwise: `REG_OOB`.
- **PC update:** apply the default PC increment rule.
- **Z flag:** unchanged.
- **Faults:** `REG_OOB`, `MEM_OOB`, `ILLEGAL_ENCODING`, `PC_OOB`.
- **Example:** `02 03 02 00 00 00 00 00` ⇒ `R3=R2`.

ADD (0x10): `rd` ← `ra + rb`. Must satisfy `imm32=0`. PC+=8. Z set to `(R[rd]==0)`. Example: `10 00 01 02 00 00 00 00`.

SUB (0x11): `rd` ← `ra - rb`. Must satisfy `imm32=0`. PC+=8. Z set to `(R[rd]==0)`. Example: `11 00 01 02 00 00 00 00`.

CMP (0x12): compare ra vs rb and set Z.

- **Operands/fields:** uses ra, rb. Must satisfy rd=0, imm32=0 or ILLEGAL_ENCODING.
- **Semantics:**
  1. tmp := (R[ra] - R[rb]) mod 2^64
  2. Z := (tmp == 0)
  3. No registers are modified.
- **PC update:** PC := PC + 8.
- **Faults:** REG_OOB, ILLEGAL_ENCODING.
- **Worked example:** If R1=7, R2=7, CMP R1,R2
  Bytes: 12 00 01 02 00 00 00 00 ⇒ Z=1.

LOAD8_ABS (0x20): rd ← MEM[imm16]. Must satisfy ra=rb=0. Reads addr=imm32&0xFFFF. Zero-extends byte. PC+=8. Z unchanged. Example: 20 01 00 00 00 01 00 00.

STORE8_ABS (0x21): MEM[imm16] ← (ra & 0xFF). Must satisfy rd=rb=0. Writes byte. PC+=8. Z unchanged. Example: 21 00 01 00 00 02 00 00.

JMP_ABS (0x30): PC ← imm16.

- **Operands/fields:** uses imm32. Must satisfy rd=ra=rb=0 or ILLEGAL_ENCODING.
- **Semantics:** target := imm32 & 0xFFFF. If target mod 8 != 0, fault MISALIGNED. If target+7 > 0xFFFF, fault PC_OOB. Otherwise PC := target.
- **Z flag:** unchanged.
- **Faults:** MISALIGNED, PC_OOB, ILLEGAL_ENCODING.

JMP_REL (0x31): PC ← PC + imm32.

- **Operands/fields:** uses imm32. Must satisfy rd=ra=rb=0 or ILLEGAL_ENCODING.
- **Semantics:** target := PC + imm32 (signed). If target is not in 0..0xFFFF, or if target+7 > 0xFFFF, fault PC_OOB. If target mod 8 != 0, fault MISALIGNED. Otherwise PC := target.
- **Z flag:** unchanged.
- **Faults:** MISALIGNED, PC_OOB, ILLEGAL_ENCODING.

JZ_ABS (0x32): if Z then PC ← imm16 else PC ← PC+8.

- **Operands/fields:** uses imm32. Must satisfy rd=ra=rb=0 or ILLEGAL_ENCODING.
- **Semantics:**
  - If Z==1: target := imm32 & 0xFFFF. If target mod 8 != 0, fault MISALIGNED. If target+7 > 0xFFFF, fault PC_OOB. Otherwise PC := target.
  - If Z==0: apply the default PC increment rule (i.e. PC := PC+8 with bounds check).
- **Z flag:** read-only.
- **Faults:** MISALIGNED, PC_OOB, ILLEGAL_ENCODING.

JZ_REL (0x33): if Z then PC ← PC + imm32 else PC ← PC+8.

- **Operands/fields:** uses imm32. Must satisfy rd=ra=rb=0 or ILLEGAL_ENCODING.
- **Semantics:**
  - If Z==1: target := PC + imm32 (signed). If target is not in 0..0xFFFF, or if target+7 > 0xFFFF, fault PC_OOB. If target mod 8 != 0, fault MISALIGNED. Otherwise PC := target.
  - If Z==0: apply the default PC increment rule (i.e. PC := PC+8 with bounds check).
- **Z flag:** read-only.
- **Faults:** MISALIGNED, PC_OOB, ILLEGAL_ENCODING.

PUSH8 (0x40): push the low byte of ra onto the stack (post-decrement).

- **Operands/fields:** uses ra. Must satisfy rd=0, rb=0, imm32=0 or ILLEGAL_ENCODING.
- **Semantics (post-decrement push):**
  1. addr := SP
  2. Bounds check: 0 <= addr <= 0xFFFF; else MEM_OOB
  3. MEM[addr] := (R[ra] & 0xFF)
  4. SP := SP - 1 (signed underflow is a fault; enforce SP stays within 0..65535)
- **PC update:** PC := PC + 8.
- **Z flag:** unchanged.
- **Faults:** REG_OOB, MEM_OOB, PC_OOB.
- **Worked example:** If SP=0xFDFF, R1=0xAB, PUSH8 R1
  Bytes: 40 00 01 00 00 00 00 00
  Effect: MEM[0xFDFF]=0xAB, SP=0xFDFE.

POP8 (0x41): pop one byte from the stack into rd (pre-increment pop).

- **Operands/fields:** uses rd. Must satisfy ra=0, rb=0, imm32=0 or ILLEGAL_ENCODING.
- **Semantics (inverse of PUSH8):**
  1. SP := SP + 1 (overflow beyond 0xFFFF is fault)
  2. addr := SP
  3. R[rd] := zero_extend_64(MEM[addr])
- **PC update:** PC := PC + 8.
- **Z flag:** unchanged.
- **Faults:** REG_OOB, MEM_OOB, PC_OOB.
- **Worked example:** If SP=0xFDFE, MEM[0xFDFF]=0x41, POP8 R2
  Bytes: 41 02 00 00 00 00 00 00
  Effect: SP=0xFDFF, R2=0x41.

CALL_ABS (0x42): call absolute target, push 8-byte return address.

- **Operands/fields:** uses `imm32` as absolute target `imm16=imm32&0xFFFF`. Must satisfy `rd=ra=rb=0` or `ILLEGAL_ENCODING`.
- **Return address size:** 8 bytes, little-endian, pushed onto the stack.
- **Semantics:**
  1. **Return address computation:** require `PC+15 <= 0xFFFF` (so that `return_pc=PC+8` is a fetchable instruction start); else `PC_OOB`. Then `return_pc := PC + 8`.
  2. **Alignment check:** the 8-byte return-address block occupies `SP-7..SP`. Let `base := SP-7`. Require `base mod 8 == 0`; else `MISALIGNED`.
  3. **Bounds check:** require `0 <= base` and `base+7 <= 0xFFFF`; else `MEM_OOB`.
  4. **Push 8 bytes at addresses `SP, SP-1, ..., SP-7`** using post-decrement semantics:
     - For `i=0..7`: `MEM[SP] := byte_i(return_pc); SP := SP - 1`

     where `byte_i` is the i-th little-endian byte.
  5. `target := imm32 & 0xFFFF`
  6. `PC := target`
- **Z flag:** unchanged.
- **Faults:** `PC_OOB`, `MEM_OOB`, `MISALIGNED`, `ILLEGAL_ENCODING`.
- **Worked example:** If `PC=0x0100`, `SP=0xFDFF` then `CALL_ABS 0x0200` succeeds (the 8-byte block base `SP-7=0xFDF8` is aligned).
  If `SP` is not positioned so that `SP-7` is 8-byte aligned (e.g. `SP=0xFDFA`), the call faults `MISALIGNED`.

RET (0x43): return by popping 8-byte address into PC.

- **Operands/fields:** must satisfy `rd=ra=rb=0, imm32=0` or `ILLEGAL_ENCODING`.
- **Semantics:**
  1. **Alignment check:** the 8-byte return-address block to pop is at `base := SP+1`. Require `base mod 8 == 0`; else `MISALIGNED`.
  2. **Bounds check:** require `0 <= base` and `base+7 <= 0xFFFF`; else `MEM_OOB`.
  3. Pop 8 bytes little-endian using inverse of post-decrement push:
     - For `i=0..7`: `SP := SP + 1; b[i] := MEM[SP]`
  4. Reconstruct `new_pc` from `b[0..7]` as little-endian.
  5. Bounds check: `0 <= new_pc <= 0xFFFF` and `new_pc+7 <= 0xFFFF`; else `PC_OOB`.
  6. `PC := new_pc`
- **Z flag:** unchanged.
- **Faults:** `MEM_OOB`, `MISALIGNED`, `PC_OOB`, `ILLEGAL_ENCODING`.
- **Worked example:** If the stack top encodes return address `0x0040`, RET sets `PC=0x0040`.

7. Memory Model, Endianness, and Alignment

RAM model.

- RAM is `MEM[65536]` bytes.
- Valid addresses are `0..65535`. Any access outside is `MEM_OOB`.
- v2 adds stack access patterns, including 8-byte return addresses.

Reserved region. Addresses `0xFE00..0xFFFF` are reserved for v3+ system/MMIO use. v2 does not assign meaning to this region, but the reset stack location avoids it.

Endianness.

- **Instruction immediates:** little-endian imm32 in bytes 4–7.
- **Return addresses on stack:** 8-byte little-endian encoding.

Alignment (forced).

- Width-1 accesses (byte) have no alignment restrictions.
- Width-8 stack accesses (CALL/RET) require 8-byte alignment of the 8-byte block base address as specified in Section 6; otherwise `MISALIGNED`.

8. Control Flow Rules

Supported control flow.

- Absolute jump/call targets: `imm16 = imm32 & 0xFFFF`.
- Relative jumps: `PC := PC + imm32` (signed), bounds checked.
- Conditional branches depend only on `FLAGS.Z`.

No indirect control flow until v5. Any opcode that would interpret a register as a target address is reserved and must fault as `ILLEGAL_OPCODE` in v2.

Instruction boundary rule. All control-flow targets must be 8-byte aligned and fetchable: require `target mod 8 == 0` and `target+7 <= 0xFFFF`. If alignment fails, fault `MISALIGNED`. If fetchability fails, fault `PC_OOB`.

9. Fault Model and Diagnostics

Fault codes (v2 minimum).

- `ILLEGAL_OPCODE`, `ILLEGAL_ENCODING`
- `REG_OOB`
- `MEM_OOB`
- `PC_OOB`
- `MISALIGNED`

Diagnostics: additional recommended fields for v2. In addition to v1 fields, include:

- `sp`, `fp` at time of fault
- `mem_width` (1 or 8)
- `access_type` (`READ`/`WRITE`)

Fault demo (stack misalignment on CALL). If `SP` is positioned so that the 8-byte return-address block base is not aligned (e.g. `SP=0xFDFA`, so `SP-7` is not 8-byte aligned), then `CALL_ABS` faults `MISALIGNED`. Expected: `fault_code=MISALIGNED`, include `pc`, `sp`, and opcode `0x42`.

10. Step-by-Step Implementation Plan

1. **Upgrade state:** add `SP` and `FP` fields; use reset `SP=FP=0xFDFF` (chosen so that an 8-byte CALL push block base `SP-7` starts aligned).
2. **Add new opcodes:** implement `CMP`, `PUSH8`, `POP8`, `CALL_ABS`, `RET`.
3. **Centralize memory helpers:** implement `read8(addr)` and `write8(addr,val)` with bounds checks.
4. **Implement stack helpers:**
   - `push8(byte)` and `pop8()` for PUSH8/POP8.
   - `push64(u64)` and `pop64()->u64` for CALL/RET (little-endian).
5. **Add alignment enforcement:** in `push64/pop64`, enforce 8-byte alignment rules.
6. **Implement recommended prologue/epilogue pattern (optional but testable):**
   - Prologue: `MOV_RR FP, SP`
   - Epilogue: `MOV_RR SP, FP`

   (More structured frame saves may be added in v4.)
7. **Update tests:** expand unit tests for new instructions; add call/return integration programs (including recursion).

11. Testing Plan (Unit + Integration + Fault Tests)

Unit tests.

- **CMP:** equal sets Z=1, not equal sets Z=0, registers unchanged.
- **PUSH8/POP8:** verify exact SP movement and memory writes/reads.
- **CALL/RET:** verify pushed bytes match little-endian return address and PC updates correctly.

Integration tests (mini-programs).

- A multi-function program: `main` calls `add2`, returns result in a register, stores to RAM.
- A recursion program (small depth) to verify correct stack behavior.

Fault tests.

- **MISALIGNED CALL:** with a mispositioned `SP` (e.g. `SP=0xFDFA`), `CALL_ABS` faults `MISALIGNED`.
- **Stack overflow/underflow:** repeated PUSH8 beyond 0x0000 should fault `MEM_OOB`.
- **RET with empty stack:** SP increment exceeds `0xFFFF` should fault `MEM_OOB`.

Example test vector table (subset).

Name	Instr Bytes (hex)	Expected
cmp_eq_sets_z	`12 00 01 02 00 00 00 00`	`Z=1 if R1==R2`
push8_r1	`40 00 01 00 00 00 00 00`	`MEM[SP]=R1&FF; SP-`
pop8_r2	`41 02 00 00 00 00 00 00`	`SP++; R2=MEM[SP]`

12. Upgrade Notes (Compatibility with v3..v5)

What v2 freezes.

- Stack grows downward; PUSH8 is post-decrement; POP8 is pre-increment.
- CALL/RET push/pop return addresses as 8-byte little-endian values.
- CMP sets Z without modifying registers.

What changes later without breaking v2 binaries.

- v3: add `INT`/`IRET` and optional MMIO semantics for the reserved region `0xFE00..0xFFFF`.
- v4: add wider loads/stores and base+offset addressing; optionally add structured frame helpers.
- v5: add indirect/computed jumps/calls and advanced protection/VM features.

13. Deliverables Checklist

By the end of CPU v2, you should have:

- Updated `cpu_state.*`: includes `SP` and `FP`.
- Updated `executor.*`: implements `CMP`, `PUSH8`, `POP8`, `CALL_ABS`, `RET`.
- `stack.*`: reusable `push8/pop8` and `push64/pop64` helpers.
- `tests_cpu_v2.*`: new unit tests + integration tests (multi-function + recursion) + fault tests.
- `programs/v2/`: demo binaries and assembly sources.

14. Appendix (Examples)

A. "Hello-style" demo (write string bytes to RAM using a subroutine)

This v2 demo writes `"OK\0"` into RAM at `0x0200..0x0202` by calling a subroutine that writes one byte.

Assembly (conceptual).

```
; main
MOV_RI  R10, 0x0200      ; destination pointer in RAM
MOV_RI  R1,  0x4F        ; 'O'
CALL_ABS putc
MOV_RI  R1,  0x4B        ; 'K'
CALL_ABS putc
MOV_RI  R1,  0x00        ; '\0'
CALL_ABS putc
HALT

; putc: store byte in R1 to [R10], then increment R10
putc:
STORE8_ABS [0x0000], R1  ; placeholder (see note below)
RET
```

Important note (v2 limitation). CPU v2 still only supports `STORE8_ABS` (absolute store). Therefore, true pointer-based stores (e.g. `STORE8 [R10]`) are not available until v4 (base+offset addressing). For v2 demonstrations, we keep the "subroutine" shape to validate CALL/RET, but memory locations are fixed per call. A realistic `putc` with `[R10]` becomes possible in v4.

Machine bytes (CALL/RET focused snippet). Assume `putc` is at `0x0080`. Then:

```
42 00 00 00 80 00 00 00   ; CALL_ABS 0x0080
43 00 00 00 00 00 00 00   ; RET
```

B. Control-flow demo (loop + conditional with CMP + JZ)

Count down from 3 to 0 in a loop and call a dummy function each iteration.

Assembly (conceptual).

```
MOV_RI R1, 3
MOV_RI R2, 0
loop:
CMP    R1, R2        ; Z=1 when R1==0
JZ_ABS done
CALL_ABS tick
MOV_RI R3, 1
SUB    R1, R1, R3
JMP_ABS loop
done:
HALT

tick:
RET
```

C. Fault demo (MISALIGNED on CALL with mispositioned SP)

With a mispositioned `SP` (e.g. `SP=0xFDFA`), the instruction:

```
42 00 00 00 00 01 00 00    ; CALL_ABS 0x0100
```

must fault `MISALIGNED` because the 8-byte return-address block base `SP-7` is not 8-byte aligned.

Expected diagnostic (minimum fields).

- `fault_code = MISALIGNED`
- `pc = address of CALL`
- `opcode = 0x42`, `imm32 = 0x00000100`
- `sp = 0xFDFA`, `mem_width = 8`, `access_type = WRITE`

4 How To Make: CPU v3

1. Overview

CPU v3 adds an OS-facing interface on top of v2: **syscalls/traps**, a minimal **MMIO console**, a deterministic **timer interrupt**, and **IEEE-754 floating point ops** using the existing 64-bit register file. The design remains emulator-first and test-driven, and stays compatible with future upgrades (v4: richer addressing + wider memory ops; v5: indirect control flow + advanced protection).

> **Compatibility note.** v3 is the first version with a vector table at `0x0000..0x00FF` and (by convention) reset at `PC=0x0100`. v1/v2 binaries remain valid machine code, but should be linked/loaded at `0x0100` (or jumped to from a small bootstrap) to avoid clobbering the vector table. v3 also expands `FLAGS` from `Z` to `ZNVC`: the meaning of `Z` is unchanged; `N/C/V` are newly defined and may be ignored by old code.

v3 Goals (frozen).

- Add syscalls/traps to enable kernel-like services.
- Add MMIO console for basic I/O without needing a full OS yet.
- Add deterministic timer interrupts for preemption-like testing.
- Add IEEE-754 double-precision floating ops in GPRs.

Non-minimal note. CPU v3 is *not* minimal; v1 is the minimal project requirement. v3 focuses on OS/toolchain readiness.

2. Frozen CPU Contract

CPU v3 inherits v1 and v2 contracts and adds a trap/interrupt subsystem and MMIO behavior.

Machine widths.

- **Register width:** 64-bit.
- **Register count:** 16 GPRs `R0..R15`.
- **Address space:** 64KB RAM, byte-addressed, `0x0000..0xFFFF`.

Special registers (v3). In v3 we keep the existing special registers and add a minimal trap return register:

- **PC**, **SP**, **FP** as in v2.
- **FLAGS:** extended in v3 to **ZNVC** (Zero, Negative, Carry, Overflow).
- **EPC:** Exception Program Counter (stores return address on trap/interrupt).

Endianness and alignment.

- **Endianness:** Little-endian for instruction immediates and multi-byte stack values.
- **Forced alignment:** any access width W must satisfy `addr mod W == 0`.
- v3 still uses byte RAM operations for general loads/stores; stack return addresses are 8-byte and therefore require 8-byte alignment for push/pop (v2 rule remains).

Fault policy. Any fault halts the CPU with diagnostics. Traps/interrupts are *not faults*; they transfer control to a handler.

Determinism. Timer interrupts must be deterministic: they occur after an instruction-count threshold, not wall-clock time.

3. CPU State Definition

State fields (v3).

- `R[16]`: GPRs (u64).
- `PC`, `SP`, `FP`: u64 storage with effective 16-bit address semantics.
- `FLAGS`: bits `Z`, `N`, `C`, `V`.
- `EPC`: u64 storage (effective 16-bit on return).
- `MEM[65536]`: RAM byte array.
- `HALTED`, `HALT_REASON`, `FAULT_INFO`.
- `ICOUNT`: monotonic instruction counter (u64), increments by 1 per executed instruction.
- `TIMER_PERIOD`: constant (e.g. 256), in instructions.
- `PENDING_TIMER`: boolean (optional convenience).

Reset convention.

- `PC = 0x0100` (v3 entrypoint; `0x0000..0x00FF` reserved for the vector table)
- `SP = 0xFDFF`, `FP = 0xFDFF`
- `EPC = 0`
- `FLAGS.Z/N/C/V = 0`
- `R[i] = 0`
- `ICOUNT = 0`
- `TIMER_PERIOD = 256` (frozen default; can be configured in emulator for tests)

CPU state layout diagram.

```
+-------------------------------------------------------------+
| CPU STATE (v3)                                              |
+----------------------+--------------------------------------+
| R0..R15 (16 x u64)   | General-purpose registers            |
| PC (u64; eff u16)    | Program counter                      |
| SP (u64; eff u16)    | Stack pointer (downward)             |
| FP (u64; eff u16)    | Frame pointer                        |
| EPC (u64; eff u16)   | Exception PC (return target for IRET)  |
| FLAGS (Z N C V)      | Arithmetic/compare flags              |
| ICOUNT (u64)         | Instruction counter                   |
| TIMER_PERIOD         | Deterministic timer interrupt interval  |
```

```
| HALTED/FAULT_INFO    | Halt state and diagnostics                |
+----------------------+-------------------------------------------+
| MEM[65536]           | RAM                                       |
| 0x0000..0x00FF       | Trap/interrupt vector table (v3)          |
| 0x0100..0xFDFF       | General RAM / program + data              |
| 0xFE00..0xFFFF       | MMIO console region (v3)                  |
+-----------------------------------------------------------------+
```

4. Instruction Encoding (Binary Spec)

v3 keeps the v1/v2 fixed-length instruction format.

Byte layout. Each instruction is 8 bytes:

```
[ opcode | rd | ra | rb | imm32 (little-endian) ]

byte index:   0      1     2     3        4  5  6  7
            +------+-----+-----+-----+----------------+
meaning:     | OPC  | rd  | ra  | rb  |   imm32 (LE)   |
            +------+-----+-----+-----+----------------+
```

Immediate behavior. `imm32` is signed two's-complement, sign-extended to 64-bit when used.

Address derivation.

- Absolute: `addr16 = imm32 & 0xFFFF`.
- PC-relative: `target := PC + imm32` (signed). If `target` is out of `0x0000..0xFFFF` or `target+7 > 0xFFFF`, fault `PC_OOB`. If `target mod 8 != 0`, fault `MISALIGNED`. Otherwise `PC := target`.

Indirect jumps. Indirect control-flow remains **not allowed** until v5.

5. Fetch–Decode–Execute Loop

v3 adds trap/interrupt dispatch around the standard fetch/decode/execute loop.

High-level pseudocode.

```
while not HALTED:

  # Deterministic timer
  if (ICOUNT != 0) and (ICOUNT % TIMER_PERIOD == 0):
      raise_interrupt(vector=0x01)   # timer interrupt

  # Fetch
  if PC+7 > 0xFFFF: fault(PC_OOB)
  if PC % 8 != 0: fault(MISALIGNED)
  instr[0..7] = MEM[PC .. PC+7]

  # Decode
  opcode = instr[0]
  rd = instr[1]; ra = instr[2]; rb = instr[3]
```

```
imm32 = sign_extend_le_u32(instr[4..7])

# Execute (may modify PC)
execute(opcode, rd, ra, rb, imm32)

# Default PC update
if pc_was_not_modified:
   nextPC = PC + 8
   if nextPC+7 > 0xFFFF: fault(PC_OOB)
   PC = nextPC

ICOUNT = ICOUNT + 1
```

Trap/interrupt transfer rule. Raising a trap/interrupt transfers control as follows:

- Compute `return_pc` as the address of the *next* instruction to execute (for a synchronous trap like `SYSCALL`, this is typically `PC+8`; for an interrupt raised between instructions, this is the current `PC`).
- Set `EPC := return_pc`.
- Set `PC := handler_addr(vector)`.

Handlers are addressed via a fixed vector table (Section 8).

6. ISA Specification (Instruction Truth Table)

CPU v3 includes all v2 instructions plus:

- Syscall/trap: `SYSCALL` and `IRET`
- Float ops: `FADD`, `FSUB`, `FMUL`

Opcode map (v3 additions).

Opcode	Mnemonic	Operands	Fields Used	Flags
0x52	SYSCALL	– (syscall# in R0)	opcode only	unchanged
0x53	IRET	–	opcode only	unchanged
0x70	FADD	rd, ra, rb	rd, ra, rb	unchanged
0x71	FSUB	rd, ra, rb	rd, ra, rb	unchanged
0x72	FMUL	rd, ra, rb	rd, ra, rb	unchanged

Reserved/Unused opcode policy.

Range	Meaning
0x00–0x4F	v1/v2 core + stack/call (defined in previous pages)
0x50–0x5F	Trap/interrupt/syscall space (v3)
0x60–0x6F	Reserved for v4 memory/addressing expansion
0x70–0x7F	Floating-point ops (v3)
0x80–0xFE	Future expansion
0xFF	Recommended illegal (always `ILLEGAL_OPCODE`)

Instruction Semantics (v3 instructions)

Note: v3 expands `FLAGS` from `Z` to `ZNVC`. The definition of `Z` is unchanged; `N/C/V` are newly defined. Unless explicitly stated, an instruction leaves unchanged any flags it does not mention.

Flag conventions (ZNVC). For integer ops in v3:

- **Z** = 1 iff result == 0
- **N** = MSB of result (bit 63)
- **C** = carry out for ADD; for SUB, **C=1 means no borrow** (standard convention)
- **V** = signed overflow (two's complement)

In v3, **integer** `ADD`, `SUB`, and `CMP` update ZNVC. Other v2 instructions keep prior behavior (e.g. MOV unchanged; LOAD/STORE unchanged). Float ops do not affect flags.

SYSCALL (0x52): enter syscall handler (syscall# in R0).

- **Fields:** must satisfy `rd=ra=rb=0, imm32=0` or `ILLEGAL_ENCODING`.
- **Semantics:**
  1. `return_pc := PC + 8`
  2. `EPC := return_pc`
  3. `PC := handler_addr(vector=0x00)` (syscall vector)
  4. (Syscall number is available to the handler in `R0`. Arguments may be placed in `R1..R5`, return value in `R0`.)
- **PC update:** explicit (set to handler).
- **Flags:** unchanged.
- **Faults:** `PC_OOB` if handler address not fetchable; `MISALIGNED` if handler address not 8-byte aligned; `ILLEGAL_ENCODING`.
- **Worked example:** If `R0=1` ("write char"), `SYSCALL` transfers to vector 0x00 and sets `EPC=PC+8`.
  Bytes: `52 00 00 00 00 00 00 00`.

IRET (0x53): return from trap/interrupt using EPC.

- **Fields:** must satisfy `rd=ra=rb=0, imm32=0` or `ILLEGAL_ENCODING`.
- **Semantics:**
  1. `new_pc := EPC`
  2. Bounds check: `0 <= new_pc <= 0xFFFF` and `new_pc+7 <= 0xFFFF`; else `PC_OOB`.
  3. Alignment check: if `new_pc mod 8 != 0`, fault `MISALIGNED`.
  4. `PC := new_pc`
- **Flags:** unchanged.
- **Faults:** `PC_OOB`, `MISALIGNED`, `ILLEGAL_ENCODING`.
- **Worked example:** After `SYSCALL`, handler ends with `IRET` to resume at `EPC`.
  Bytes: `53 00 00 00 00 00 00 00`.

FADD (0x70): IEEE-754 double add.

- **Fields:** uses `rd, ra, rb`. Must satisfy `imm32=0` or `ILLEGAL_ENCODING`.
- **Semantics:**
  1. Interpret `R[ra]` and `R[rb]` bitwise as IEEE-754 binary64 values a, b.
  2. Compute $c := a + b$ using IEEE-754 semantics (round-to-nearest ties-to-even).
  3. Store bitwise representation of c into `R[rd]`.
- **PC update:** `PC := PC + 8`.
- **Flags:** unchanged (float ops do not touch ZNVC in v3).
- **Faults:** `REG_OOB`, `ILLEGAL_ENCODING`.
- **Worked example:** If `R1` encodes 1.5 and `R2` encodes 2.0, then `FADD R0,R1,R2` makes `R0` encode 3.5.

FSUB (0x71): IEEE-754 double subtract. Same as FADD but $c := a - b$. Flags unchanged. Example opcode bytes: `71 rd ra rb 00 00 00 00`.

FMUL (0x72): IEEE-754 double multiply. Same as FADD but $c := a \times b$. Flags unchanged. Example opcode bytes: `72 rd ra rb 00 00 00 00`.

7. Memory Model, Endianness, and Alignment

RAM and MMIO. Memory is byte-addressed. In v3, a top region is **MMIO** (memory-mapped I/O), meaning accesses have device semantics.

MMIO region (v3).

- MMIO address range: `0xFE00..0xFFFF`.
- General rule: `LOAD8_ABS` and `STORE8_ABS` to MMIO addresses are allowed and may trigger device behavior.

Console device map (minimal).

- `0xFE00` **CONSOLE_OUT** (write-only): writing a byte prints it to the host console.
- `0xFE01` **CONSOLE_STATUS** (read-only): returns `1` if input is available else `0`.
- `0xFE02` **CONSOLE_IN** (read-only): returns next input byte if available, else `0`.

Alignment.

- Byte loads/stores have no alignment restriction.
- Stack and return-address operations remain 8-byte aligned (v2 rule).

8. Control Flow Rules

Vector table and handler addresses. v3 reserves the first 256 bytes of memory for a fixed trap/interrupt vector table:

- Vector entry size: 2 bytes (u16), little-endian.
- Table base: `0x0000`.
- Table size: `0x0100` bytes (128 vectors, `0x00..0x7F`).
- Reserved region: `0x0000..0x00FF`. Program code/data should start at `0x0100`; therefore v3 resets with `PC=0x0100`.
- Handler address for vector v: read u16 from `MEM[0x0000 + 2*v .. 0x0001 + 2*v]`.

A handler address must be 8-byte aligned and fetchable (`addr mod 8 == 0` and `addr+7 <= 0xFFFF`); otherwise raising the trap/interrupt faults (`MISALIGNED` or `PC_OOB`).

Vectors used in v3.

- **0x00** = syscall handler
- **0x01** = timer interrupt handler

Timer interrupt trigger (deterministic). Every `TIMER_PERIOD` instructions, an interrupt is raised with vector 0x01. The interrupt sets:

- `EPC := PC` (return to the instruction that would execute next)
- `PC := handler_addr(0x01)`

No indirect jumps until v5. Computed targets remain prohibited.

9. Fault Model and Diagnostics

Fault vs trap/interrupt.

- **Fault:** halts the CPU (illegal opcode, OOB access, misalignment, PC OOB).
- **Trap/interrupt:** transfers control to a handler and can return via `IRET`.

Fault codes (v3 minimum).

- `ILLEGAL_OPCODE`, `ILLEGAL_ENCODING`
- `REG_OOB`
- `MEM_OOB`
- `PC_OOB`
- `MISALIGNED`

Diagnostic fields (recommended).

- v2 fields plus `epc`, `flags` (ZNVC), `icount`
- if MMIO is involved: `mmio_addr`, `mmio_op` (READ/WRITE)

Fault demo (invalid opcode in user code). If opcode `0xFF` is fetched at `PC`:

- fault: `ILLEGAL_OPCODE`
- include: `pc, opcode=0xFF, rd/ra/rb, imm32, flags, epc, icount`
- CPU halts immediately (no trap).

10. Step-by-Step Implementation Plan

1. **Extend FLAGS:** upgrade integer ALU ops (ADD/SUB/CMP) to compute ZNVC. Keep other ops unchanged.
2. **Add EPC to CPU state:** update reset and diagnostics to include it.
3. **Implement vector table:** write helper `handler_addr(v)` that reads a u16 from RAM at `0x0000 + 2*v` and validates the handler is 8-byte aligned and fetchable (`addr+7 <= 0xFFFF`).
4. **Implement SYSCALL/IRET:** SYSCALL sets `EPC=PC+8` and jumps to vector 0x00 handler. IRET sets `PC=EPC`.
5. **Implement timer interrupt:** add `ICOUNT` and trigger every `TIMER_PERIOD` instructions; on trigger, set `EPC=PC` and jump to vector 0x01 handler.
6. **Implement MMIO console:** intercept LOAD8/STORE8 at `0xFE00..0xFFFF` and implement the minimal device map (Section 7).
7. **Implement float ops:** add FADD/FSUB/FMUL. Use host language IEEE-754 double to compute, but preserve bitwise round-trip into u64 registers.
8. **Expand tests:** add syscall integration, timer interrupt integration, MMIO tests, and float correctness tests.

11. Testing Plan (Unit + Integration + Fault Tests)

Unit tests: flags (ZNVC). For ADD/SUB/CMP, test:

- Z: result == 0
- N: sign bit set (bit 63)
- C: carry/no-borrow conventions
- V: signed overflow (e.g. `0x7FFF..  + 1`)

Unit tests: SYSCALL/IRET.

- SYSCALL sets `EPC=PC+8` and transfers to handler.
- IRET restores `PC=EPC` and resumes user code.

Unit tests: MMIO.

- STORE8 to `0xFE00` emits exactly one byte to host console (capture stdout).
- LOAD8 from `0xFE01` returns status.

Unit tests: float ops.

- FADD/FSUB/FMUL on known bit patterns yields correct IEEE-754 results.
- Float ops do not change flags.

Integration tests.

- "Hello" via MMIO: write bytes to `0xFE00` to print a string.
- Syscall print: user sets `R0=syscall#`, `R1=char`, executes `SYSCALL`; handler writes to MMIO and `IRET`.
- Timer preemption demo: user loop runs; timer handler increments a counter in RAM; returns with `IRET`.

Fault tests.

- illegal opcode `0xFF` halts with `ILLEGAL_OPCODE`
- PC out of bounds on IRET (set `EPC` to `0xFFFC` and IRET should fault `PC_OOB`)

Example test vector table (subset).

Name	Instr Bytes (hex)	Expected
syscall_enters_handler	`52 00 00 00 00 00 00 00`	`EPC=PC+8; PC=vec[0]`
iret_returns	`53 00 00 00 00 00 00 00`	`PC=EPC`
fadd_basic	`70 00 01 02 00 00 00 00`	`R0=double(R1)+double(R2)`

12. Upgrade Notes (Compatibility with v4..v5)

v4 upgrades (planned).

- Add base+offset addressing and wider memory ops (LOAD16/32/64, STORE16/32/64).
- Potentially add structured frame helpers; keep v2/v3 CALL/RET semantics stable.

v5 upgrades (planned).

- Add indirect/computed control flow.
- Add advanced protection/virtual memory or atomic operations (optional).

Compatibility rule. All v1–v3 opcode semantics must remain stable. New features must use reserved opcode ranges or clearly new opcodes.

13. Deliverables Checklist

By the end of CPU v3, you should have:

- Updated `cpu_state.*`: adds `EPC`, `FLAGS(ZNVC)`, `ICOUNT`, `TIMER_PERIOD`.
- Updated `executor.*`: implements `SYSCALL`, `IRET`, `FADD/FSUB/FMUL`.
- `vector.*`: vector table helper `handler_addr(v)`.
- `mmio.*`: console MMIO implementation and tests.
- `tests_cpu_v3.*`: flags tests, syscall/iret tests, timer tests, float tests.
- `programs/v3/`: demo binaries (MMIO hello, syscall hello, timer demo).

14. Appendix (Examples)

A. “Hello-style” demo (MMIO console output)

This program prints `"HI\n"` by writing bytes to `0xFE00`.

Assembly (conceptual).

```
MOV_RI R1, 0x48        ; 'H'
STORE8_ABS [0xFE00], R1
MOV_RI R1, 0x49        ; 'I'
STORE8_ABS [0xFE00], R1
MOV_RI R1, 0x0A        ; '\n'
STORE8_ABS [0xFE00], R1
HALT
```

Machine bytes (each line is 8 bytes).

```
01 01 00 00 48 00 00 00    ; MOV_RI R1,'H'
21 00 01 00 00 FE 00 00    ; STORE8_ABS [0xFE00],R1
01 01 00 00 49 00 00 00    ; MOV_RI R1,'I'
21 00 01 00 00 FE 00 00    ; STORE8_ABS [0xFE00],R1
01 01 00 00 0A 00 00 00    ; MOV_RI R1,'\n'
21 00 01 00 00 FE 00 00    ; STORE8_ABS [0xFE00],R1
00 00 00 00 00 00 00 00    ; HALT
```

B. Syscall demo (SYSCALL handler writes to console and returns)

Syscall ABI (v3 default):

- `R0` = syscall number
- `R1` = argument (e.g. char)
- return value in `R0`

User code (conceptual).

```
MOV_RI R0, 1           ; syscall 1 = putchar
MOV_RI R1, 0x41        ; 'A'
SYSCALL
HALT
```

Handler (vector 0x00) (conceptual).

```
; at handler address from vector table
; if R0==1: write R1 to CONSOLE_OUT
STORE8_ABS [0xFE00], R1
IRET
```

C. Timer interrupt demo (handler increments a RAM counter)

Vector 0x01 handler increments `MEM[0x0300]` every time it runs.

Handler (conceptual).

```
LOAD8_ABS R1, [0x0300]
MOV_RI    R2, 1
ADD       R1, R1, R2
STORE8_ABS [0x0300], R1
IRET
```

User loop (conceptual).

```
MOV_RI R3, 0
loop:
ADD R3, R3, R2      ; keep CPU busy
JMP_ABS loop
```

D. Fault demo (IRET with invalid EPC)

If the handler sets `EPC=0xFFFC` then `IRET` must fault `PC_OOB` because `EPC+7 > 0xFFFF`.

- Expected: `fault_code=PC_OOB`, include `epc=0xFFFC`, `opcode=0x53`.

5 How To Make: CPU v4

1. Overview

CPU v4 is the "toolchain-friendly" upgrade: it adds **wider integer memory operations** (16/32/64-bit) and **base+offset addressing** to support stack frames, arrays, and compiler-generated code. It also expands the core integer ISA with common bitwise and shift operations. All v1–v3 binaries remain valid.

> **Compatibility note.** v4 keeps the v3 MMIO map `0xFE00..0xFFFF` and its byte-wide device semantics, but explicitly forbids *wide* MMIO (16/32/64-bit) to keep devices simple: any wide MMIO load/store is a fault (`ILLEGAL_ENCODING`). In v5, this width rule still applies; v5 additionally restricts MMIO to kernel mode.

v4 Goals (frozen).

- Add **LOAD/STORE** widths 16/32/64 in addition to byte.
- Add **base+offset addressing** for all supported widths.
- Add **bitwise** and **shift** instructions needed by compilers.
- Keep MMIO/syscalls/timer/float from v3 unchanged and compatible.

Non-minimal note. CPU v4 is not minimal; it exists to make compilers and larger programs practical.

2. Frozen CPU Contract

CPU v4 inherits all v1–v3 contracts and adds new opcodes only.

Machine widths.

- **Register width:** 64-bit.
- **Register count:** 16 GPRs `R0..R15`.
- **Memory:** 64KB, byte-addressed, `0x0000..0xFFFF`.

Special registers.

- **PC, SP, FP** as in v2/v3.
- **FLAGS: ZNVC** (carried from v3).
- **EPC** (exception PC), **ICOUNT**, **TIMER_PERIOD** (from v3).

Endianness and alignment.

- **Little-endian** for multi-byte values in RAM and instruction immediates.
- **Forced alignment:** any access width W bytes requires `addr mod W == 0`; otherwise fault `MISALIGNED`.
- v4 introduces $W \in \{2, 4, 8\}$ loads/stores; alignment is now actively enforced.

Fault policy. Any fault halts with diagnostics. Traps/interrupts remain transfers (not faults).

Determinism. Timer interrupt remains deterministic (instruction-count based).

3. CPU State Definition

No new architectural state is required beyond v3; v4 adds capabilities via new opcodes.

State fields (summary).

- `R[16]`, `PC`, `SP`, `FP`
- `FLAGS(ZNVC)`, `EPC`
- `MEM[65536]`
- `ICOUNT`, `TIMER_PERIOD`
- `HALTED/FAULT_INFO`

CPU state layout diagram.

```
+-------------------------------------------------------------+
| CPU STATE (v4)                                              |
+----------------------+--------------------------------------+
| R0..R15 (16 x u64)   | General-purpose registers            |
| PC / SP / FP         | Control + stack/frame pointers       |
| FLAGS (Z N C V)      | Integer flags (ZNVC)                  |
| EPC                  | Trap/interrupt return address         |
| ICOUNT / PERIOD      | Deterministic timer interrupt         |
| HALTED/FAULT_INFO    | Fault halting and diagnostics         |
+----------------------+--------------------------------------+
| MEM[65536]           | RAM (LE multi-byte)                   |
| 0xFE00..0xFFFF       | MMIO console region (v3+)             |
+-------------------------------------------------------------+
```

4. Instruction Encoding (Binary Spec)

v4 retains the fixed-length 8-byte encoding.

Byte layout (unchanged).

```
              [ opcode | rd | ra | rb | imm32 (little-endian) ]

byte index:   0      1     2     3        4  5  6  7
            +------+-----+-----+-----+----------------+
meaning:     | OPC  | rd  | ra  | rb  |   imm32 (LE)   |
            +------+-----+-----+-----+----------------+
```

Immediate behavior. `imm32` is signed and sign-extended to 64-bit when used as an integer. For absolute addresses, `addr16 = imm32 & 0xFFFF`. For base+offset, effective address is `addr = (R[ra] + imm32)` with bounds checks.

Indirect jumps. Indirect/computed control flow remains prohibited until v5.

5. Fetch–Decode–Execute Loop

Unchanged from v3 (including deterministic timer interrupts), except the execute dispatcher recognizes new opcodes.

6. ISA Specification (Instruction Truth Table)

CPU v4 includes all v1–v3 instructions plus the v4 additions below.

v4 opcode map (new in v4).

Opcode	Mnemonic	Operands	Fields Used	Flags
0x13	AND	rd, ra, rb	rd, ra, rb	ZNVC set
0x14	OR	rd, ra, rb	rd, ra, rb	ZNVC set
0x15	XOR	rd, ra, rb	rd, ra, rb	ZNVC set
0x16	SHL	rd, ra, rb (shift amt)	rd, ra, rb	ZNVC set
0x17	SHR	rd, ra, rb (shift amt)	rd, ra, rb	ZNVC set
0x60	LOAD16_ABS	rd, [imm16]	rd, imm32	unchanged
0x61	STORE16_ABS	[imm16], ra	ra, imm32	unchanged
0x62	LOAD32_ABS	rd, [imm16]	rd, imm32	unchanged
0x63	STORE32_ABS	[imm16], ra	ra, imm32	unchanged
0x64	LOAD64_ABS	rd, [imm16]	rd, imm32	unchanged
0x65	STORE64_ABS	[imm16], ra	ra, imm32	unchanged
0x66	LOAD8_BO	rd, [ra+imm32]	rd, ra, imm32	unchanged
0x67	STORE8_BO	[ra+imm32], rb	ra, rb, imm32	unchanged
0x68	LOAD16_BO	rd, [ra+imm32]	rd, ra, imm32	unchanged
0x69	STORE16_BO	[ra+imm32], rb	ra, rb, imm32	unchanged
0x6A	LOAD32_BO	rd, [ra+imm32]	rd, ra, imm32	unchanged
0x6B	STORE32_BO	[ra+imm32], rb	ra, rb, imm32	unchanged
0x6C	LOAD64_BO	rd, [ra+imm32]	rd, ra, imm32	unchanged
0x6D	STORE64_BO	[ra+imm32], rb	ra, rb, imm32	unchanged

Reserved/Unused opcode policy.

Range	Policy
0x00–0x5F	Defined by v1–v3 (HALT, MOV, ALU, stack/call, traps, float, etc.)
0x6E–0x6F	Reserved (future addressing helpers / extensions)
0x70–0x7F	Float ops defined in v3 (FADD/FSUB/FMUL), more reserved
0x80–0xFE	Reserved for future expansion
0xFF	Always illegal (`ILLEGAL_OPCODE`)

Instruction Semantics (v4 additions)

Note: All v1–v3 instructions keep their prior semantics. This section specifies only the new v4 instructions.

Flag update rule (v4).

- Integer ops `ADD, SUB, CMP` (v3) update **ZNVC**.
- New integer ops `AND, OR, XOR, SHL, SHR` (v4) update **ZNVC** as specified below.
- Memory ops and control-flow ops do not modify flags unless explicitly stated (unchanged).
- Float ops do not modify flags (unchanged).

AND (0x13): `rd` ← `ra & rb`.

- **Fields:** `rd,ra,rb` used; require `imm32=0` else `ILLEGAL_ENCODING`.
- **Semantics:** `R[rd] := R[ra] & R[rb]`.
- **Flags (ZNVC):** `Z` set if result==0; `N` = bit63(result); `C=0`; `V=0`.
- **PC:** `PC := PC + 8`.
- **Faults:** `REG_OOB`, `ILLEGAL_ENCODING`.
- **Example:** If `R1=0xF0`, `R2=0x0F`, `AND R0,R1,R2` ⇒ `R0=0`, `Z=1`.
  Bytes: `13 00 01 02 00 00 00 00`.

OR (0x14): `rd` ← `ra | rb`. Same encoding rules as AND; flags: Z/N set from result, C=0, V=0.
Example bytes: `14 rd ra rb 00 00 00 00`.

XOR (0x15): `rd` ← `ra ^ rb`. Same encoding rules as AND; flags: Z/N set from result, C=0, V=0.
Example bytes: `15 rd ra rb 00 00 00 00`.

SHL (0x16): logical left shift.

- **Fields:** `rd,ra,rb` used; require `imm32=0`.
- **Semantics:**
  1. `sh := R[rb] & 63`
  2. `res := (R[ra] « sh) mod 2^64`
  3. `R[rd] := res`
- **Flags (ZNVC):** `Z` set if `res==0`; `N` = bit63(`res`); `V=0`; `C := 0` if `sh==0` else `((R[ra] » (64 - sh)) & 1)`.
- **PC:** `PC := PC + 8`.
- **Faults:** `REG_OOB`, `ILLEGAL_ENCODING`.
- **Example:** `SHL R0,R1,R2` with `R1=1`, `R2=3` ⇒ `R0=8`.
  Bytes: `16 00 01 02 00 00 00 00`.

SHR (0x17): logical right shift.

- **Fields:** `rd,ra,rb` used; require `imm32=0`.
- **Semantics:**
  1. `sh := R[rb] & 63`
  2. `res := (R[ra] » sh)` (logical shift, zero-fill)
  3. `R[rd] := res`
- **Flags (ZNVC):** `Z` set if `res==0`; `N` = bit63(`res`); `V=0`; `C := 0` if `sh==0` else `((R[ra] » (sh - 1)) & 1)`.
- **PC:** `PC := PC + 8`.
- **Faults:** `REG_OOB`, `ILLEGAL_ENCODING`.

LOADW_ABS / STOREW_ABS: multi-byte absolute loads/stores (W=16/32/64). These are LOAD16/32/64_ABS and STORE16/32/64_ABS.

- **Encoding:**
  - LOADW_ABS: uses rd and imm32; require ra=0, rb=0.
  - STOREW_ABS: uses ra (source) and imm32; require rd=0, rb=0.
- **Semantics (LOADW):**
  1. addr := imm32 & 0xFFFF
  2. **MMIO gating (v4 frozen):** if addr is in 0xFE00..0xFFFF, fault ILLEGAL_ENCODING (wide MMIO is illegal in v4).
  3. Alignment check: addr mod (W/8) == 0 else MISALIGNED.
  4. Bounds check: addr + (W/8) - 1 <= 0xFFFF else MEM_OOB.
  5. Read W bits from RAM in little-endian order and zero-extend to 64: R[rd] := zeroext(load_le_W(addr)).
- **Semantics (STOREW):**
  1. addr := imm32 & 0xFFFF
  2. **MMIO gating (v4 frozen):** if addr is in 0xFE00..0xFFFF, fault ILLEGAL_ENCODING (wide MMIO is illegal in v4).
  3. Alignment and bounds checks as above.
  4. Store the low W bits of R[ra] into RAM in little-endian order.
- **Flags:** unchanged.
- **PC:** PC := PC + 8.
- **Faults:** REG_OOB, ILLEGAL_ENCODING, MISALIGNED, MEM_OOB.
- **Example (LOAD64_ABS):** LOAD64_ABS R1,[0x0400]
  Bytes: 64 01 00 00 00 04 00 00.

LOADW_BO / STOREW_BO: base+offset addressing (W=8/16/32/64). These are LOAD8/16/32/64_BO and STORE8/16/32/64_BO.

- **Encoding:**
  - LOADW_BO: uses rd (dest), ra (base), imm32 (offset); require rb=0.
  - STOREW_BO: uses ra (base), rb (src), imm32 (offset); require rd=0.
- **Effective address:** addr := (R[ra] + imm32) with signed addition in 64-bit then bounds check into 0..0xFFFF.
- **Alignment and bounds:** enforce addr mod (W/8) == 0 and addr + (W/8) - 1 <= 0xFFFF.
- **MMIO rule:** for W=8, if addr is in 0xFE00..0xFFFF, apply MMIO semantics from v3. For W>8, MMIO accesses fault ILLEGAL_ENCODING in v4 (frozen to keep devices simple).
- **Flags:** unchanged.
- **PC:** PC := PC + 8.

- **Faults:** REG_OOB, ILLEGAL_ENCODING, MISALIGNED, MEM_OOB.
- **Example (STORE8_BO):** STORE8_BO [R10+0], R1
  Bytes: 67 00 0A 01 00 00 00 00.

7. Memory Model, Endianness, and Alignment

Little-endian multi-byte RAM. Multi-byte values are stored in RAM little-endian: lowest-address byte is least significant.

Alignment is now active.

- LOAD16/STORE16 require 2-byte alignment.
- LOAD32/STORE32 require 4-byte alignment.
- LOAD64/STORE64 require 8-byte alignment.
- Misalignment is a MISALIGNED fault (halts).

MMIO constraints.

- MMIO region remains 0xFE00..0xFFFF.
- Only byte-width MMIO is supported in v4; wider MMIO is illegal (fault).

8. Control Flow Rules

Control flow remains identical to v3:

- Absolute and PC-relative branches remain.
- Syscalls/traps and timer interrupts transfer via vector table and return via IRET.
- Indirect/computed jumps are still disallowed until v5.

9. Fault Model and Diagnostics

New faults emphasized in v4. Misalignment and bounds failures are common now:

- MISALIGNED for multi-byte loads/stores when addr mod W != 0.
- MEM_OOB for addr + bytes - 1 > 0xFFFF.
- ILLEGAL_ENCODING for invalid field constraints (e.g. wrong must-be-zero fields or illegal wide MMIO).

Diagnostic fields (recommended additions).

- mem_width in bytes (1/2/4/8)
- eff_addr computed effective address (for BO)
- mmio_addr and mmio_op when applicable

10. Step-by-Step Implementation Plan

1. **Memory helpers:** implement `read_le16/32/64` and `write_le16/32/64` over the byte array.
2. **Alignment checks:** implement a single helper `check_align(addr, width)` that faults `MISALIGNED`.
3. **Bounds checks:** implement `check_bounds(addr, width)` that faults `MEM_OOB`.
4. **Add ABS width opcodes:** LOAD16/32/64_ABS and STORE16/32/64_ABS.
5. **Add BO addressing:** LOAD/STORE {8,16,32,64}_BO with effective-address computation.
6. **MMIO gating:** allow only byte-width MMIO; fault on wider MMIO.
7. **Add bitwise/shift ops:** AND/OR/XOR/SHL/SHR and their ZNVC behavior.
8. **Update tests:** add misalignment tests, endian tests, pointer-walking integration demos.

11. Testing Plan (Unit + Integration + Fault Tests)

Unit tests: endianness. Write a known 64-bit pattern to RAM with STORE64 and confirm bytes match little-endian order; then LOAD64 returns the same value.

Unit tests: misalignment. Attempt `LOAD64_ABS` from `0x0401` and confirm `MISALIGNED`.

Unit tests: BO addressing. Set `R10=0x0200`, load/store with offsets ± values, confirm effective address computation is signed and bounds-checked.

Integration tests.

- Pointer-walk string print using `LOAD8_BO` and MMIO output.
- Simple stack-frame style access using `FP` as base (even before full ABI enforcement).

Fault tests.

- Wide MMIO access (e.g. `STORE16_ABS [0xFE00],R1`) must fault `ILLEGAL_ENCODING`.
- Out-of-bounds multi-byte access (e.g. `LOAD32_ABS [0xFFFF]`) must fault `MEM_OOB`.

Example test vector table (subset).

Name	Instr Bytes (hex)	Expected
load64_abs_ok	`64 01 00 00 00 04 00 00`	`R1=MEM64LE[0x0400]`
store8_bo_ok	`67 00 0A 01 00 00 00 00`	`MEM[R10+0]=R1&FF`
load64_misalign	`64 01 00 00 01 04 00 00`	`MISALIGNED fault`

12. Upgrade Notes (Compatibility with v5)

v5 upgrades (planned).

- Add indirect/computed jumps/calls (e.g. `JMP_R`, `CALL_R`).
- Optionally add protection/VM or atomics (if desired), without breaking v4 binaries.

Compatibility rule. All v1–v4 opcode semantics remain stable. New functionality must use new opcodes (reserved space) or explicitly new rules that do not change decoding of existing opcodes.

13. Deliverables Checklist

By the end of CPU v4, you should have:

- Updated `mem_access.*`: `read/write` helpers for 16/32/64 LE with bounds+alignment checks.
- Updated `executor.*`: new ABS width ops and BO ops; bitwise/shift ops.
- Updated `tests_cpu_v4.*`: endian tests, misalignment tests, BO tests, integration demos.
- `programs/v4/`: string-print demo using pointer walking; stack-frame-ish demo using FP as base.

14. Appendix (Examples)

A. "Hello-style" demo (pointer-walk string and print via MMIO)

This v4 demo stores `"HELLO\n\0"` in RAM and prints it by walking a pointer using base+offset loads.

Assembly (conceptual).

```
; Data at 0x0200: "HELLO\n\0"
MOV_RI   R10, 0x0200      ; ptr
loop:
LOAD8_BO R1, [R10+0]      ; ch = *ptr
CMP      R1, R0           ; assume R0==0 at reset; Z=1 if ch==0
JZ_ABS   done
STORE8_ABS [0xFE00], R1   ; console out
MOV_RI   R2, 1
ADD      R10, R10, R2     ; ptr++
JMP_ABS  loop
done:
HALT
```

Machine bytes (illustrative snippet; labels require assembly/linking).

```
66 01 0A 00 00 00 00 00   ; LOAD8_BO R1,[R10+0]
12 00 01 00 00 00 00 00   ; CMP R1,R0
32 00 00 00 ?? ?? 00 00   ; JZ_ABS done (imm16 filled by assembler)
21 00 01 00 00 FE 00 00   ; STORE8_ABS [0xFE00],R1
```

B. Control-flow demo (sum 64-bit array using LOAD64_BO)

Sum four 64-bit integers stored contiguously in RAM at `0x0400` and store the result at `0x0420`.

Assembly (conceptual).

```
MOV_RI   R10, 0x0400      ; base
MOV_RI   R11, 0           ; sum
MOV_RI   R12, 0           ; i
MOV_RI   R13, 4           ; n
MOV_RI   R14, 8           ; stride
loop:
CMP      R12, R13
JZ_ABS   done
; addr = base + i*8  (v4 has no MUL yet by default; unroll or compute via adds)
LOAD64_BO R1, [R10+0]      ; simplest: assume unrolled in real code
ADD      R11, R11, R1
ADD      R10, R10, R14     ; base += 8
MOV_RI   R2, 1
ADD      R12, R12, R2
JMP_ABS  loop
done:
STORE64_ABS [0x0420], R11
HALT
```

C. Fault demo (misaligned 64-bit load)

Attempting to load a 64-bit value from a non-8-aligned address must fault `MISALIGNED`.

Faulting instruction.

```
LOAD64_ABS R1, [0x0401]
```

Faulting bytes.

```
64 01 00 00 01 04 00 00
```

Expected diagnostic (minimum).

- `fault_code = MISALIGNED`
- `pc = address of faulting instruction`
- `opcode = 0x64`, `rd=1`, `imm32 = 0x00000401`
- `mem_width = 8`, `eff_addr = 0x0401`

6 How To Make: CPU v5

1. Overview

CPU v5 is the "conceptual maximum" upgrade over v4: it adds **indirect/computed control flow** (jumps and calls via registers) and a small set of **protection primitives** that are still emulator-friendly and deterministic. The design preserves all v1–v4 binaries and keeps the fixed 8-byte instruction format.

> **Compatibility note (what changed, what did not).** v5 keeps all v1–v4 encodings and behavior for existing opcodes. New behavior appears only when you use v5-only instructions (`JMP_R/JZ_R/JNZ_R/CALL_R/SETI/SETK/TRAP`) or when running in user mode (`K=0`).
>
> - **Halt vs trap split:** v1–v4 faults (including legacy `MISALIGNED` for `CALL_ABS/RET`) still *halt*. v5 introduces new *traps* for protection violations and indirect-target/stack validation so a protected runtime can recover or report cleanly.
> - **MMIO rule stack-up:** the v4 rule "wide MMIO is illegal" still applies in v5 (even in kernel mode). v5 adds a *privilege* rule: user-mode code cannot access `0xFE00..0xFFFF` directly (trap to vector `0x03`).

v5 Goals (frozen).

- Add indirect jumps/calls so compiled code can use function pointers, jump tables, and dynamic dispatch.
- Add a minimal, deterministic protection model suitable for advanced runtime support.
- Keep v3 syscalls/MMIO/timer/floats and v4 memory/addressing intact.

Scope note. v5 is intentionally conceptual: the goal is the *highest upgrade* that still fits the v4 base (64KB RAM, fixed 8-byte encoding, emulator-first determinism). This document freezes a minimal-but-powerful set of protection primitives rather than a full VM/MMU.

2. Frozen CPU Contract

CPU v5 inherits all contracts from v1–v4 and adds: indirect control flow + protection.

Machine widths.

- **Register width:** 64-bit.
- **Register count:** 16 GPRs `R0..R15`.
- **Memory size:** 64KB, byte-addressed, `0x0000..0xFFFF`.

Special registers (v5). We keep all v3/v4 special registers and add a small protection set:

- **PC, SP, FP** as before.
- **FLAGS: ZNVC** plus two additional control flags:
  - **I** (Interrupt Enable): when 0, timer interrupts are masked (syscalls still work).
  - **K** (Kernel Mode): 1 when executing in kernel mode, 0 in user mode.
- **EPC:** exception return address (v3+).
- **CAUSE:** reason code for last trap/interrupt/fault transfer (not for halting faults).
- **BADADDR:** holds faulting address for trap-type violations (e.g. protection trap).
- **ICOUNT, TIMER_PERIOD** as before.

Endianness and alignment.

- **Little-endian** everywhere (instruction immediates, RAM multi-byte values, stack return addresses).
- **Forced alignment** remains: access width W requires `addr mod W == 0`.
- Indirect control-flow targets must be 8-byte aligned and fetchable (`PC+7<=0xFFFF`).

Fault vs trap (v5).

- **Fault (halt):** illegal opcode/encoding, register OOB, fatal PC OOB, etc.
- **Protection violations and indirect-target violations:** are **traps** (transfer), not halting faults.

3. CPU State Definition

v5 extends the v4 state with minimal protection state.

State fields (v5).

- `R[16]` (u64), `PC, SP, FP` (u64 effective u16).
- `FLAGS: Z,N,C,V,I,K`.
- `EPC`, `CAUSE`, `BADADDR`.
- `MEM[65536]`.
- `ICOUNT`, `TIMER_PERIOD`.
- `HALTED`, `HALT_REASON`, `FAULT_INFO`.

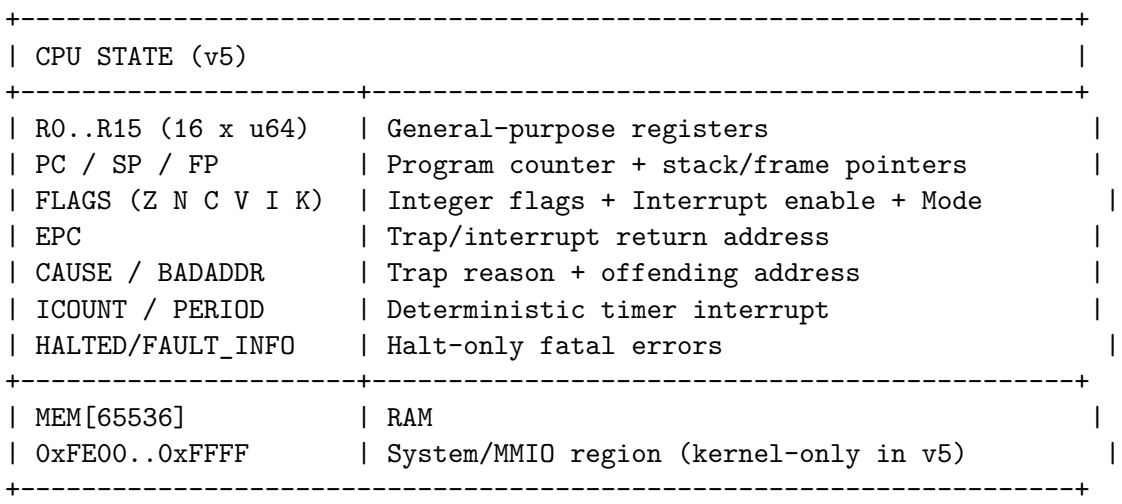
Protection model (minimal, frozen). v5 introduces a user/kernel split without a full MMU:

- **K flag:** `K=1` means kernel mode; `K=0` means user mode.
- **Vector table region (kernel-only):** `0x0000..0x00FF` is reserved for the vector table. Any user-mode *fetch or data* access into this region triggers a **trap** (vector 0x03).
- **User memory window:** user code may access only `0x0100..0xFDFF`. Any user access to `0xFE00..0xFFFF` triggers a **trap** (vector 0x03).
- **MMIO region:** `0xFE00..0xFFFF` remains MMIO/system as in v3; kernel may access it.

Reset convention.

- `PC=0x0100`, `SP=FP=0xFDFF`, `EPC=0`.
- `FLAGS.ZNVC=0`, `FLAGS.I=1`, `FLAGS.K=1` (start in kernel mode so a runtime can initialize vectors, then drop to user with `SETK 0`).
- `CAUSE=0`, `BADADDR=0`.
- `R[i]=0`, `ICOUNT=0`, `TIMER_PERIOD=256`.

CPU state layout diagram.



4. Instruction Encoding (Binary Spec)

v5 keeps the fixed 8-byte encoding:

```
[ opcode | rd | ra | rb | imm32 (little-endian) ]

byte index:   0      1     2     3        4  5  6  7
            +------+-----+-----+-----+----------------+
meaning:     | OPC  | rd  | ra  | rb  |   imm32 (LE)   |
            +------+-----+-----+-----+----------------+
```

Indirect control-flow operand convention (frozen). Indirect targets are taken from `ra`:

- `JMP_R ra`: target in `R[ra]`.
- `CALL_R ra`: target in `R[ra]`.
- `JZ_R ra`, `JNZ_R ra`: target in `R[ra]`.

All other fields must be zero (`rd=0, rb=0, imm32=0`) unless explicitly stated.

Immediate behavior. `imm32` semantics unchanged (signed, sign-extended). Indirect ops require `imm32=0`.

5. Fetch–Decode–Execute Loop

v5 extends the v3 loop by adding:

- mode-aware memory checks (user/kernel region restrictions),
- trap transfers for protection and indirect-target violations,
- interrupt masking via `FLAGS.I`.

High-level pseudocode (delta from v3).

```
while not HALTED:

  # Timer interrupt (deterministic) if enabled
  if FLAGS.I == 1:
    if (ICOUNT != 0) and (ICOUNT % TIMER_PERIOD == 0):
        trap(vector=0x01, cause=TIMER)

  # Fetch (fetch is subject to protection: user cannot execute from vector table or MMIO reg
  if PC+7 > 0xFFFF: fault(PC_OOB)
  if (FLAGS.K == 0) and ((PC < 0x0100) or (PC >= 0xFE00)):
      trap(vector=0x03, cause=PROT_EXEC, badaddr=PC)
  instr = MEM[PC..PC+7]

  decode fields...

  execute(...):
    - may fault (halt) or trap (transfer) or proceed normally

  if pc_not_modified: PC += 8

  ICOUNT += 1
```

6. ISA Specification (Instruction Truth Table)

CPU v5 includes all v1–v4 instructions plus new indirect control-flow and protection-control instructions.

v5 opcode map (new in v5).

Opcode	Mnemonic	Operands	Fields Used	Flags
0x34	JMP_R	ra	ra	unchanged
0x35	JZ_R	ra	ra	read Z
0x36	JNZ_R	ra	ra	read Z
0x44	CALL_R	ra	ra	unchanged
0x54	SETI	imm1 (0/1)	imm32	sets I
0x55	SETK	imm1 (0/1)	imm32	sets K
0x56	TRAP	imm8 vector	imm32	unchanged

Reserved/Unused opcode policy.

Range	Policy
0x00–0x7F	Defined by v1–v4 plus v5 additions above
0x80–0xEF	Reserved for optional atomics/protection extensions
0xF0–0xFE	Reserved for experimental/debug ops
0xFF	Always illegal (`ILLEGAL_OPCODE`)

Instruction Semantics (v5 additions)

Trap transfer primitive (used by multiple rules). A **trap** (non-halting transfer) is defined as:

1. `EPC := return_pc` (typically `PC+8` for synchronous traps; for timer, `PC`)

2. `CAUSE := cause_code`
3. `BADADDR := offending_addr` (0 if not applicable)
4. `FLAGS.K := 1` (enter kernel mode)
5. `PC := handler_addr(vector)` (vector table as in v3)

Trap handler returns with `IRET`, which sets `PC=EPC`. (Mode and I flag are not automatically restored; software controls them via `SETK/SETI`.)

JMP_R (0x34): indirect jump.

- **Fields:** uses `ra`. Require `rd=0, rb=0, imm32=0` else `ILLEGAL_ENCODING`.
- **Semantics:**
  1. `target := R[ra] & 0xFFFF`
  2. Validate: `target mod 8 == 0` and `target+7 <= 0xFFFF`; else **trap** vector 0x02 with `CAUSE=BAD_TARGET`, `BADADDR=target`.
  3. If in user mode (`K=0`) and `target >= 0xFE00`, trap vector 0x03 with `CAUSE=PROT_EXEC`, `BADADDR=target`.
  4. Otherwise `PC := target`.
- **PC update:** explicit.
- **Flags:** unchanged.
- **Faults:** `REG_OOB`, `ILLEGAL_ENCODING` (trap for target issues).
- **Worked example:** If `R5=0x0080`, `JMP_R R5` sets `PC=0x0080`.
  Bytes: `34 00 05 00 00 00 00 00`.

JZ_R (0x35): conditional indirect jump if Z==1.

- **Fields:** `ra` used; require others zero.
- **Semantics:** if `Z==1` then behave like `JMP_R`; else `PC := PC + 8`.
- **Flags:** unchanged (read-only Z).
- **Example:** `35 00 ra 00 00 00 00 00`.

JNZ_R (0x36): conditional indirect jump if Z==0. Same as JZ_R but taken when `Z==0`.

CALL_R (0x44): indirect call with 8-byte return address.

- **Fields:** uses `ra` as target reg; require `rd=0, rb=0, imm32=0`.
- **Semantics:**
  1. `return_pc := PC + 8`
  2. Enforce 8-byte stack alignment for return push (same as v2 CALL_ABS/RET): if `(SP-7) mod 8 != 0` then **trap** vector 0x02 with `CAUSE=BAD_STACK`, `BADADDR=SP`.

3. Push 8 bytes of `return_pc` little-endian at `SP,SP-1,...,SP-7` using post-decrement semantics.
4. `target := R[ra] & 0xFFFF`; validate like JMP_R; on violation trap.
5. `PC := target`.

- **Design note (consistency):** legacy `CALL_ABS/RET` keep their v2 halting `MISALIGNED` behavior for backward compatibility. The indirect family (including `CALL_R`) uses a **trap** for `BAD_STACK` so protected runtimes can recover, report, or terminate user code cleanly.
- **Flags:** unchanged.
- **Worked example:** If `R4=0x0100` and stack aligned, `CALL_R R4` pushes return and sets `PC=0x0100`.
  Bytes: `44 00 04 00 00 00 00 00`.

SETI (0x54): set interrupt enable flag I.

- **Fields:** uses `imm32`. Require `rd=ra=rb=0` else `ILLEGAL_ENCODING`.
- **Semantics:** `FLAGS.I := (imm32 & 1)`.
- **PC:** `PC := PC + 8`.
- **Flags:** sets only I; ZNVC unchanged.
- **Example:** `SETI 0` bytes: `54 00 00 00 00 00 00 00`.

SETK (0x55): set kernel mode flag K (privileged).

- **Fields:** uses `imm32`. Require `rd=ra=rb=0`.
- **Privilege rule (frozen):** if `FLAGS.K==0` (user mode) and `imm32&1==1`, then **trap** vector 0x03 with `CAUSE=PROT_PRIV`, `BADADDR=PC`. User code cannot directly enter kernel.
- **Semantics (kernel allowed):** `FLAGS.K := (imm32 & 1)`.
- **PC:** `PC := PC + 8` (if not trapped).
- **Example:** Kernel exits to user: `SETK 0`.

TRAP (0x56): software trap to a vector (debug/service).

- **Fields:** uses `imm32` low byte as vector. Require `rd=ra=rb=0`.
- **Semantics:** synchronous trap with `vector := imm32 & 0xFF`, `EPC := PC+8`, `CAUSE := SWTRAP`, `BADADDR := 0`, `K:=1`, `PC := handler_addr(vector)`.
- **Example:** `TRAP 0x10` bytes: `56 00 00 00 10 00 00 00`.

7. Memory Model, Endianness, and Alignment

Same as v4 with protection enforcement.

- Multi-byte RAM is little-endian.
- Alignment rules enforced for 2/4/8 byte accesses.
- MMIO region `0xFE00..0xFFFF` exists; in v5 it is **kernel-only**.
- **Wide MMIO is still illegal:** as in v4, only byte-width MMIO is supported. Any 16/32/64-bit MMIO load/store is a halting fault (`ILLEGAL_ENCODING`).

Protection checks (frozen). For any memory access (ABS or BO):

- If `K==0` and (`addr < 0x0100` or `addr >= 0xFE00`), then **trap** vector 0x03 with `CAUSE=PROT_MEM`, `BADADDR=addr`.
- Otherwise apply bounds + alignment checks as in v4.

8. Control Flow Rules

Vector table (carried from v3). Vector table base `0x0000`, entry size 2 bytes (u16 LE), handler address from `MEM[2*v..2*v+1]`. The region `0x0000..0x00FF` is reserved for vectors and is **kernel-only** in v5: any user-mode fetch or data access into this region traps (vector 0x03). Normal user programs should be linked to start at `0x0100` (the reset `PC`).

Vectors (v5).

- 0x00 syscall (v3)
- 0x01 timer interrupt (v3) (masked by I flag)
- 0x02 bad target / bad stack / indirect control-flow violations (v5)
- 0x03 protection violations (v5)

Recommended CAUSE values (symbols). The ISA does not require numeric cause encodings; for implementation and debugging, it helps to standardize symbolic cause names.

Category	CAUSE (suggested)
Syscall / software	`SYSCALL`, `SWTRAP`
Timer	`TIMER`
Indirect violations	`BAD_TARGET`, `BAD_STACK`
Protection	`PROT_MEM`, `PROT_EXEC`, `PROT_PRIV`

Timer interrupts.

- Trigger: every `TIMER_PERIOD` instructions when `I==1`.
- Transfer: `EPC := PC`, `CAUSE := TIMER`, `BADADDR := 0`, `K:=1`, `PC := handler(0x01)`.

Indirect control flow (new rule).

- Targets must be 8-byte aligned and fetchable.
- Invalid target or forbidden region (for example, user attempting to execute from `0x0000..0x00FF` or `0xFE00..0xFFFF`) triggers a **trap**, not a halt.

9. Fault Model and Diagnostics

Two categories of exceptional events.

- **Halting faults:** `ILLEGAL_OPCODE`, `ILLEGAL_ENCODING`, `REG_OOB`, fatal `PC_OOB`, fatal `MEM_OOB`, `MISALIGNED`.
- **Traps (non-halting transfers):** `PROT_*`, `BAD_TARGET`, `BAD_STACK`, `SWTRAP`, `TIMER`, `SYSCALL`.

Trap diagnostics (recommended). When a trap occurs, record:

- `EPC, CAUSE, BADADDR, FLAGS.K/I`
- `vector` used and computed handler address

Fault demo (illegal encoding). `JMP_R` requires non-`ra` fields zero. The bytes below must halt with `ILLEGAL_ENCODING` because `imm32 != 0`:

```
34 00 05 00 01 00 00 00
```

10. Step-by-Step Implementation Plan

1. **Add new state regs:** implement `CAUSE` and `BADADDR`; extend FLAGS with `I` and `K`.
2. **Implement trap() primitive:** centralize `EPC/CAUSE/BADADDR/K` updates and vector table jump.
3. **Add protection checks:** enforce user vs kernel address windows for instruction fetch and data accesses.
4. **Implement indirect control flow:** `JMP_R, JZ_R, JNZ_R, CALL_R` with target validation and trap-on-bad-target.
5. **Implement SETI/SETK/TRAP:** include privilege restriction for SETK (user cannot enter kernel).
6. **Update timer interrupt logic:** mask with I flag, set CAUSE appropriately.
7. **Test harness:** extend tracing to log traps (vector, cause, badaddr, epc).

11. Testing Plan (Unit + Integration + Fault Tests)

Unit tests: indirect targets.

- **Valid aligned target:** JMP_R sets PC to target.
- **Misaligned target:** trap vector 0x02 with `CAUSE=BAD_TARGET`, `BADADDR=target`.
- **OOB target:** trap vector 0x02.

Unit tests: protection.

- User STORE8 to `0xFE00` traps vector 0x03 with `CAUSE=PROT_MEM`.
- User JMP_R into `0xFE00` traps vector 0x03 with `CAUSE=PROT_EXEC`.
- Kernel handler can write to MMIO and return to user with `SETK 0; IRET`.

Integration tests.

- **Hello via syscall:** user calls `SYSCALL`; kernel handler writes to MMIO and returns.
- **Function pointer call:** user loads address of function into reg and uses `CALL_R`.
- **Jump table:** user computes target address and uses `JMP_R` (still no v5-wide pointers beyond 64KB).

Fault tests.

- Illegal encoding for indirect ops (nonzero forbidden fields) halts.
- Illegal opcode `0xFF` halts.

Example test vector table (subset).

Name	Instr Bytes (hex)	Expected
jmp_r_ok	`34 00 05 00 00 00 00 00`	`PC=R5 (validated)`
jmp_r_badalig	`34 00 05 00 00 00 00 00`	`trap if (R5 mod 8)!=0`
seti_off	`54 00 00 00 00 00 00 00`	`I=0`
trap_10	`56 00 00 00 10 00 00 00`	`EPC=PC+8; PC=vec[0x10]`

12. Upgrade Notes (Compatibility with v5)

v5 is the top version in this roadmap; there is no v6. The remaining evolution is implementation and tooling (assembler/linker, higher-level languages, OS services), not ISA-breaking changes.

13. Deliverables Checklist

By the end of CPU v5, you should have:

- Updated `cpu_state.*`: adds `CAUSE`, `BADADDR`, FLAGS `I/K`.
- Updated `trap.*`: unified trap transfer implementation.
- Updated `executor.*`: implements `JMP_R, JZ_R, JNZ_R, CALL_R, SETI, SETK, TRAP`.
- Updated `protection.*`: user/kernel memory rules enforced for fetch and data access.
- `tests_cpu_v5.*`: trap tests, protection tests, indirect call/jump tests.
- `programs/v5/`: demos (function pointers, syscall hello, jump table).

14. Appendix (Examples)

A. "Hello-style" demo (user SYSCALL to kernel console)

User program requests kernel to print a character via syscall #1.

User code (conceptual).

```
MOV_RI R0, 1          ; syscall 1 = putchar
MOV_RI R1, 0x48       ; 'H'
SYSCALL
MOV_RI R1, 0x49       ; 'I'
SYSCALL
MOV_RI R1, 0x0A       ; '\n'
SYSCALL
HALT
```

Kernel syscall handler (vector 0x00) (conceptual).

```
; assumes K==1 on entry
; if R0==1: write R1 to MMIO console out
STORE8_ABS [0xFE00], R1
SETK 0
IRET
```

B. Control-flow demo (function pointer CALL_R)

This demonstrates dynamic dispatch: a function pointer is stored in a register and called indirectly.

Assembly (conceptual).

```
MOV_RI  R4, funcA_addr
CALL_R  R4
HALT

funcA:
MOV_RI  R1, 0x41      ; 'A'
; request kernel putchar (syscall 1)
MOV_RI  R0, 1
SYSCALL
RET
```

Machine bytes (snippet for CALL_R and JMP_R style).

```
44 00 04 00 00 00 00 00   ; CALL_R R4
34 00 04 00 00 00 00 00   ; JMP_R  R4
```

C. Fault/Trap demo (user attempts MMIO write directly)

In v5, user-mode code cannot write MMIO directly; it must use syscalls.

Faulting user instruction (causes trap, not halt).

```
STORE8_ABS [0xFE00], R1
```

Expected trap outcome.

- Trap vector: 0x03 (protection)
- `CAUSE=PROT_MEM`
- `BADADDR=0xFE00`
- `EPC = user PC + 8`
- `K=1` on entry to handler

Example protection handler (vector 0x03) (conceptual).

```
; kernel mode: handle/terminate/log
; simplest: halt by executing HALT (kernel may choose policy)
HALT
```