P32W Instruction Set

The user ISA, the system architecture and every instruction

P32W — Ken Rother's 32-bit TTL computer

Generated 2026-11-13 from the repository at 3cacc22 2026-11-13

Contents

P32W ISA v0 `docs/isa-v0.md`

1. Machine state (v1)
2. Instruction formats
3. Opcode map
4. Reset state (v1)
5. Memory map (emulator v1, physical, no MMU)
6. Exceptions (v1)
7. Review outcome (2026-10-11)

P32W System Architecture — Stage A: Traps and Privilege `docs/isa-system.md`

1. Modes
2. Control registers
3. Exception causes
4. New instructions (opcodes 0x70–0x7F)
5. Trap entry and return
6. Protection in stage A
7. Emulator v2 behaviour
8. Software conventions (kernel ABI, not ISA)
9. Review outcome (2026-10-11)

B1. Interrupt lines
B2. When an interrupt is taken
B3. `WAIT` (opcode 0x75, privileged, format N)
B4. Timer device (IRQ0)
B5. Emulator v3 behaviour
B6. Kernel (software, not ISA)
B7. Review outcome (2026-10-11)

C1. Design in one paragraph
C2. Address spaces
C3. Pages and the TLB
C4. New control registers and instruction
C5. Exceptions (codes 8–11)
C6. Emulator v4 behaviour
C7. Kernel (software, not ISA)
C8. Review outcome (2026-10-11)

P32W instruction reference `docs/instruction-reference.md`

Control registers (MFCR / MTCR)
Exception causes (CAUSE)

ISA additions: measured candidates `docs/isa-measurements.md`

How it was measured
Results: cycles saved, % of all cycles
What the measurements say besides
The outcome (D59)

P32W ISA v0

docs/isa-v0.md

Status: ACCEPTED (2026-10-11, D8). Implements decisions D5–D10 in decisions.md. Scope: what the instruction-level reference emulator v1 (D7) needs. Privileged instructions, the MMU, interrupts and exceptions-as-traps come in a later revision.

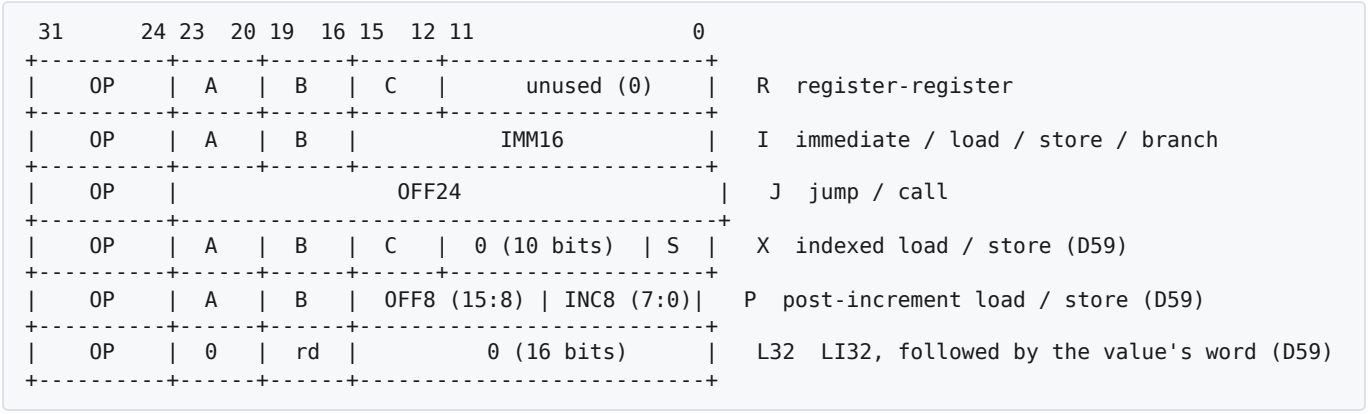
1. Machine state (v1)

State	Notes
r0	Always 0; writes discarded (D4)
r1–r14	General purpose, 32 bits
r15	Stack pointer (D6). Banked: supervisor SP / user SP. v1 runs in supervisor mode only, so only the supervisor SP is used
PC	32 bits, always a multiple of 4

No flags register (D5). Memory is byte-addressed and little-endian (D4).

2. Instruction formats

Every instruction is 32 bits, word-aligned (D3). The fields never move:



- **OP** — 8-bit opcode. The microcode will be addressed by this field (D3).
- **A** — destination register (rd); for stores, the data register; for branches, the first compared register.
- **B** — first source register (rs1); base register for loads/stores; second compared register for branches; target register for JR/CALLR.
- **C** — second source register (rs2), R format only.
- **IMM16** — 16-bit immediate, sign- or zero-extended per instruction.
- **OFF24** — 24-bit signed word offset.
- Unused bits must be 0. The emulator reports a nonzero unused field as an illegal instruction, keeping those bits free for later revisions.

Because fields sit on 4-bit boundaries, a hex dump reads directly: `ADD r1,r2,r3` = `0x10123000`, `ADDI r1,r0,42` = `0x2010002A`.

Branch and jump targets are relative to the *next* instruction: `target = PC + 4 + offset × 4`.

- Conditional branches (IMM16): ±128 KiB. (D5 estimated ±512 KiB for a 6-bit opcode; an 8-bit opcode trades that for simpler decode.)
- JMP/CALL (OFF24): ±32 MiB.
- JR/CALLR reach anywhere.

3. Opcode map

`sext` = sign-extend, `zext` = zero-extend, `mem32[x]` = word at byte address x. All arithmetic is modulo 2^{32}. `rd` = A, `rs1` = B, `rs2` = C.

0x00–0x0F — Special

OP	Mnemonic	Fmt	Operation
0x00	—	—	**Illegal** (so executing zeroed memory stops at once)
0x01	`HALT`	—	Stop. Emulator exit status = r1 & 0xFF. (Privileged in later revisions)
0xFF	—	—	**Illegal** (so executing erased EEPROM stops at once)

0x10–0x1F — Register-register ALU (R)

OP	Mnemonic	Operation
0x10	`ADD rd,rs1,rs2`	rd = rs1 + rs2
0x11	`SUB rd,rs1,rs2`	rd = rs1 − rs2
0x12	`AND rd,rs1,rs2`	rd = rs1 & rs2
0x13	`OR rd,rs1,rs2`	rd = rs1 | rs2
0x14	`XOR rd,rs1,rs2`	rd = rs1 ^ rs2
0x15	`SLL rd,rs1,rs2`	rd = rs1 << (rs2 & 31)
0x16	`SRL rd,rs1,rs2`	rd = rs1 >> (rs2 & 31), zero fill
0x17	`SRA rd,rs1,rs2`	rd = rs1 >> (rs2 & 31), sign fill
0x18	`SLT rd,rs1,rs2`	rd = (signed rs1 < signed rs2) ? 1 : 0
0x19	`SLTU rd,rs1,rs2`	rd = (unsigned rs1 < unsigned rs2) ? 1 : 0
0x1A	`NOR rd,rs1,rs2`	rd = ~(rs1 | rs2)

0x20–0x2F — Immediate ALU (I)

The low nibble matches the register form where one exists.

OP	Mnemonic	Operation
0x20	`ADDI rd,rs1,imm`	rd = rs1 + sext(imm)
0x22	`ANDI rd,rs1,imm`	rd = rs1 & zext(imm)
0x23	`ORI rd,rs1,imm`	rd = rs1 | zext(imm)
0x24	`XORI rd,rs1,imm`	rd = rs1 ^ zext(imm)
0x25	`SLLI rd,rs1,sh`	rd = rs1 << sh (imm bits 4:0; bits 15:5 must be 0)
0x26	`SRLI rd,rs1,sh`	logical right shift

OP	Mnemonic	Operation
0x27	`SRAI rd,rs1,sh`	arithmetic right shift
0x28	`SLTI rd,rs1,imm`	rd = (signed rs1 < signed sext(imm)) ? 1 : 0
0x29	`SLTIU rd,rs1,imm`	rd = (unsigned rs1 < unsigned sext(imm)) ? 1 : 0
0x2F	`LUI rd,imm`	rd = imm << 16 (B must be 0)

There is no SUBI (use ADDI with a negative immediate). A full 32-bit constant is `LUI rd,hi` + `ORI rd,rd,lo`; the assembler provides `LI rd,value`, which emits one instruction when the value fits and two otherwise.

0x30–0x3F — Multiply / divide (R)

Implemented by the FPGA ALU in hardware (D21); the emulator models them directly.

OP	Mnemonic	Operation
0x30	`MUL rd,rs1,rs2`	rd = low 32 bits of rs1 × rs2
0x31	`MULH rd,rs1,rs2`	rd = high 32 bits, signed × signed
0x32	`MULHU rd,rs1,rs2`	rd = high 32 bits, unsigned × unsigned
0x34	`DIV rd,rs1,rs2`	signed quotient, rounded toward zero
0x35	`DIVU rd,rs1,rs2`	unsigned quotient
0x36	`REM rd,rs1,rs2`	signed remainder (sign of rs1)
0x37	`REMU rd,rs1,rs2`	unsigned remainder

Divide never traps:

- Division by zero: quotient = 0xFFFFFFFF, remainder = rs1.
- Signed overflow ($-2^{31} \div -1$): quotient = -2^{31}, remainder = 0.

These are the RISC-V results, chosen so that software checks the cases it cares about and the hardware needs no divide trap.

0x40–0x4F — Loads and stores (I)

Address = rs1 + sext(imm). Misaligned halfword/word access → exception (D4).

OP	Mnemonic	Operation
0x40	`LB rd,imm(rs1)`	rd = sext(mem8)
0x41	`LH rd,imm(rs1)`	rd = sext(mem16)
0x42	`LW rd,imm(rs1)`	rd = mem32
0x44	`LBU rd,imm(rs1)`	rd = zext(mem8)
0x45	`LHU rd,imm(rs1)`	rd = zext(mem16)
0x48	`SB ra,imm(rs1)`	mem8 = ra & 0xFF
0x49	`SH ra,imm(rs1)`	mem16 = ra & 0xFFFF
0x4A	`SW ra,imm(rs1)`	mem32 = ra
0x4C	`SWAP rd,imm(rs1)`	t = mem32; mem32 = rd; rd = t — atomic (D51)

(`ra` = the A field = the data register.)

`SWAP` (D51) exchanges a register with an aligned word in memory as one indivisible operation: the read and the write are one bus tenure under `-LOCK` (bus.md), so no DMA master (or, later, another processor) can come between them. It is the primitive for locks: `SWAP` a 1 into the lock word; a 0 back means the lock was free and is now yours. Faults are taken as for a load and then a store of the same word (MISALIGNED, TLBMISS_R, then PROTECT if the page is read-only): the memory and `rd` change only when both accesses succeed.

0x80–0x9F — Indexed and post-increment loads and stores (D59)

Added for the three-bus datapath with its address adder (D59): a load or store's address goes straight into MAR, so these take no longer than a plain load. The low opcode bits mean what they do in 0x40–0x4F (size, sign, store), in every group.

Indexed (X): address = rs1 + rs2 × scale, scale 1, 2, 4 or 8 (`S` = log2 scale). Written `LWX rd, rs1, rs2, 4` (`, 1` may be left out).

OP	Mnemonic	Operation
0x80	`LBX rd,rs1,rs2,s`	rd = sext(mem8)
0x81	`LHX rd,rs1,rs2,s`	rd = sext(mem16)
0x82	`LWX rd,rs1,rs2,s`	rd = mem32
0x84	`LBUX rd,rs1,rs2,s`	rd = zext(mem8)
0x85	`LHUX rd,rs1,rs2,s`	rd = zext(mem16)
0x88	`SBX ra,rs1,rs2,s`	mem8 = ra & 0xFF
0x89	`SHX ra,rs1,rs2,s`	mem16 = ra & 0xFFFF
0x8A	`SWX ra,rs1,rs2,s`	mem32 = ra

Post-increment (P): the access at rs1 + sext(OFF8), then rs1 += sext(INC8) (both −128…127). Written `LWP rd, off(rs1), inc` (`(rs1)` for offset 0). rs1 is written **after** the access has succeeded, and last: a fault leaves rd and rs1 unchanged (the instruction can be restarted), and with rd = rs1 a load's value is overwritten by the increment.

OP	Mnemonic	Operation
0x90	`LBP rd,off(rs1),inc`	rd = sext(mem8); rs1 += inc
0x91	`LHP rd,off(rs1),inc`	rd = sext(mem16); rs1 += inc
0x92	`LWP rd,off(rs1),inc`	rd = mem32; rs1 += inc
0x94	`LBUP rd,off(rs1),inc`	rd = zext(mem8); rs1 += inc
0x95	`LHUP rd,off(rs1),inc`	rd = zext(mem16); rs1 += inc
0x98	`SBP ra,off(rs1),inc`	mem8 = ra & 0xFF; rs1 += inc
0x99	`SHP ra,off(rs1),inc`	mem16 = ra & 0xFFFF; rs1 += inc
0x9A	`SWP ra,off(rs1),inc`	mem32 = ra; rs1 += inc

`LI32 rd, value` (0x2E, two words): rd = the word that follows the instruction, read as instructions are (execute permission); the next instruction is the one after it. rd is in field **B** (A and the low 16 bits are 0), which keeps the CPU card's decode table at eight must-be-zero classes. The assembler's `LA`, and `LI` of a value that needs 32 bits, are `LI32` (the same 8 bytes as `LUI` + `ORI`, in 3 cycles instead of 5).

0x50–0x5F — Conditional branches (I)

`target = PC + 4 + sext(imm) × 4`. Compares A with B (D5).

OP	Mnemonic	Taken if
0x50	`BEQ ra,rb,label`	ra == rb
0x51	`BNE ra,rb,label`	ra != rb
0x54	`BLT ra,rb,label`	signed ra < rb
0x55	`BGE ra,rb,label`	signed ra ≥ rb
0x56	`BLTU ra,rb,label`	unsigned ra < rb
0x57	`BGEU ra,rb,label`	unsigned ra ≥ rb

Assembler aliases: `BGT a,b` = `BLT b,a`; `BLE a,b` = `BGE b,a` (and the unsigned forms); `BEQZ a` = `BEQ a,r0`; `BNEZ a` = `BNE a,r0`.

0x60–0x6F — Jumps, calls, return

The stack is full-descending, SP = r15 (D6). Calls push the return address (D6). SP and PC are committed only after the memory access succeeds, so a faulted CALL or RET can be restarted.

OP	Mnemonic	Fmt	Operation
0x60	`JMP label`	J	PC = PC + 4 + sext(off) × 4
0x61	`CALL label`	J	mem32[SP−4] = PC + 4; SP −= 4; PC = target
0x62	`JR rb`	I (B only)	PC = rb (must be a multiple of 4)
0x63	`CALLR rb`	I (B only)	mem32[SP−4] = PC + 4; SP −= 4; PC = rb
0x64	`RET`	—	PC = mem32[SP]; SP += 4

Assembler aliases:

Alias	Expands to
`NOP`	`ADD r0,r0,r0`
`MOV rd,rs`	`ADD rd,rs,r0`
`NEG rd,rs`	`SUB rd,r0,rs`
`NOT rd,rs`	`NOR rd,rs,r0`
`PUSH rs`	`ADDI sp,sp,-4` + `SW rs,0(sp)`
`POP rd`	`LW rd,0(sp)` + `ADDI sp,sp,4`

0x70–0x7F — System (reserved in v0)

Reserved for `SYSCALL`, return-from-exception, status/MMU register access and user-SP access (D1, D6). In v1 they decode as illegal.

All other opcodes are illegal.

4. Reset state (v1)

- PC = 0x0000_0000.
- Supervisor mode.
- r1–r15 = 0 in the emulator (undefined in hardware). Startup code must set SP.

5. Memory map (emulator v1, physical, no MMU)

Range	Size	Use
0x0000_0000 – 0x000F_FFFF	1 MiB	ROM (program image loaded here; writes are an error)
0x0010_0000 – 0x03FF_FFFF	up to 63 MiB	RAM: one to four 16 MiB memory cards in windows 0–3 (D52); the ROM hides the first MiB of window 0
0xFE00_0000 – 0xFE07_FFFF	64 KiB a card	video memory (D52, D53)
0xFFFF_8000 – 0xFFFF_FFFF	32 KiB	I/O page: eight 4 KiB device windows (D52)
everything else	—	unmapped → exception

The I/O page is the top 32 KiB, so `LB rd,-256(r0)` reaches `0xFFFF_FF00` with no extra instructions: a sign-extended 16-bit offset from r0 covers 0xFFFF_8000–0xFFFF_FFFF, echoing the P8X $FF00 I/O page.

I/O page (D50, D52): each device card answers in a 4 KiB window chosen by its jumpers, its registers in the window's top 256 bytes; the defaults put the I/O card in window 7 and the disk card in window 6. Byte registers answer byte accesses only; the timer, word accesses only; anything else gets no `-ACK` → BUSERROR. Full register descriptions: the programmer's guide (programmers-guide.md).

Address	Device
0xFFFF_FF00–07	UART 0, the console (TL16C550 registers 0–7): data at FF00, line status at FF05 (bit 0 = receive data ready, bit 5 = transmit ready)
0xFFFF_FF08–0F	UART 1 (TL16C550)
0xFFFF_FF10/14/18	timer TCOUNT / TRELOAD / TCTRL (isa-system.md B4)
0xFFFF_FF20	input port (8 bits)
0xFFFF_FF21	output port (8 bits, reads back)
0xFFFF_FF30–3F	UARTs 2 and 3 (TL16C550)
0xFFFF_FF40/41	real-time clock (D23)
0xFFFF_FF50/51, FF54/55	USB ports 0 and 1 (CH376S, D43)
0xFFFF_FF52/53	network coprocessor (D45)
0xFFFF_EF00–07	the disk card's port 0, CompactFlash, True-IDE task file (16-bit data register, D26); ports 1–3 follow, 16 bytes each

6. Exceptions (v1)

The emulator stops and prints the PC, the offending instruction and all registers on:

- illegal instruction
- misaligned access, or a JR/CALLR target that is not a multiple of 4
- unmapped address
- write to ROM

7. Review outcome (2026-10-11)

All four review questions were accepted as proposed:

1. 8-bit opcode; conditional branches reach ±128 KiB.
2. Branch and jump offsets are relative to PC + 4 (the next instruction).
3. Memory map as in §5.
4. `NOR` added at 0x1A; `NOT rd,rs` = `NOR rd,rs,r0`.

P32W System Architecture — Stage A: Traps and Privilege

docs/isa-system.md

Status: stage A ACCEPTED (2026-10-11, D10); stage B ACCEPTED (2026-10-11, D11); stage C ACCEPTED (2026-10-11, D12). Extends isa-v0.md for stage A of D9. Stage B (timer interrupt) and stage C (MMU) will add to this document. Everything here is designed so interrupts and page faults use the same mechanism later.

1. Modes

Mode	Can do
Supervisor (S = 1)	everything
User (S = 0)	ordinary instructions only; privileged instructions and the I/O page trap

Reset starts in supervisor mode with interrupts disabled. The only ways from user to supervisor mode are traps (exceptions, SYSCALL, and later interrupts); the only way back is `RTE`.

r15 is banked (D6): in supervisor mode r15 is the **supervisor SP (SSP)**, in user mode the **user SP (USP)**. The switch happens automatically with the mode. Supervisor code reads and writes the user SP through control register USP.

2. Control registers

Accessed only with the privileged `MFCR` / `MTCR` instructions (§4).

#	Name	Contents
0	`STATUS`	bit 0 **S** (supervisor), bit 1 **IE** (interrupt enable, used from stage B), bit 2 **PS** (previous S), bit 3 **PIE** (previous IE). Other bits read 0, ignore writes
1	`EPC`	exception PC — address of the instruction that trapped
2	`CAUSE`	exception code (§3); bit 31 = 1 for interrupts (stage B)
3	`BADADDR`	faulting address for misaligned, bus-error, upper-half/I/O and page-fault exceptions; **0** for exceptions without an address (illegal, privileged instruction, SYSCALL, BREAK) and for interrupts (D15)
4	`VBR`	vector base: address of the single exception handler; must be a multiple of 4; reset value 0
5	`USP`	the user-mode r15, readable/writable from supervisor mode
6–15	—	reserved (timer in stage B, MMU in stage C); access traps as illegal

3. Exception causes

Code	Name	Raised by	BADADDR
1	`ILLEGAL`	undefined opcode or nonzero must-be-zero bits	—
2	`PRIVILEGE`	privileged instruction, or I/O-page access, in user mode	address (I/O case)
3	`MISALIGNED`	misaligned load/store, or JR/CALLR/RET to an address not a multiple of 4	the address
4	`BUSERROR`	access to an unmapped physical address, write to ROM	the address
5	`SYSCALL`	`SYSCALL` instruction	—
6	`BREAK`	`BREAK` instruction (debugger breakpoint)	—

Code	Name	Raised by	BADADDR
8–11	—	reserved: page faults (stage C)	
bit 31 set	—	reserved: interrupts (stage B)	

Divide by zero never traps (isa-v0.md).

4. New instructions (opcodes 0x70–0x7F)

New format **CR**: `OP | A = general register | B = control-register number | bits 15:0 = 0`.

OP	Mnemonic	Fmt	Privileged	Operation
0x70	`SYSCALL`	N	no	trap with CAUSE = SYSCALL
0x71	`RTE`	N	yes	return from exception (§5)
0x72	`MFCR rd, cr`	CR	yes	rd = control register cr
0x73	`MTCR cr, rs`	CR	yes	control register cr = rs
0x74	`BREAK`	N	no	trap with CAUSE = BREAK

The assembler accepts control-register names: `STATUS`, `EPC`, `CAUSE`, `BADADDR`, `VBR`, `USP`.

`HALT` **becomes privileged** (traps with PRIVILEGE in user mode). Opcodes 0x75–0x7F stay reserved (e.g. `WAIT` for stage B, TLB/page-table control for stage C).

5. Trap entry and return

Trap entry (any exception; later also interrupts), done by the hardware:

1. `EPC` ← address of the trapping instruction
2. `CAUSE` ← code; `BADADDR` ← faulting address (if any)
3. `PS` ← `S`, `PIE` ← `IE`; then `S` ← 1, `IE` ← 0
4. `PC` ← `VBR`

Trap entry **writes no memory** and changes no general register except that r15 now selects the supervisor SP. So it cannot itself fault, and a user program with a bad SP cannot crash trap entry. The handler saves whatever registers it uses.

The trapping instruction has **no effect**: no register, SP, PC or memory change (the D6 commit-after-access rule). So after a fault, `RTE` re-executes it — this is what page faults in stage C rely on.

`RTE`: `PC` ← `EPC`; `S` ← `PS`; `IE` ← `PIE`. The r15 bank follows the restored mode. `PS` and `PIE` are left unchanged.

`MTCR STATUS` takes effect at once: writing S = 0 switches to user mode (and the user r15) for the next instruction.

`VBR` **and** `EPC` **read back exactly as written;** the PC holds only bits 31:2, so trap entry and `RTE` ignore their low 2 bits (D15).

SYSCALL and BREAK also leave `EPC` pointing at themselves; the handler adds 4 to `EPC` before `RTE` to continue after them. (One rule for every exception keeps the microcode simple.)

Nesting: a trap overwrites `EPC`, `CAUSE`, `BADADDR` and the previous-mode bits. A handler that may trap again (or, in stage B, re-enable interrupts) must first save `EPC` and `STATUS` on the supervisor stack.

6. Protection in stage A

Without an MMU, user programs can still read and write any RAM — real memory protection arrives with stage C. Stage A enforces what it can cheaply:

- privileged instructions trap in user mode;
- **the I/O page traps in user mode** (one address comparison: A31–A15 all ones and S = 0), so user programs must reach devices through SYSCALL.

7. Emulator v2 behaviour

- Exceptions are taken as traps (§5) instead of stopping the emulator.
- **While `VBR` = 0 (no handler installed), an exception still stops the emulator with the register dump**, as in v1, so bare-metal test programs keep working. (Hardware would jump to address 0.)
- `HALT` in supervisor mode stops the emulator as before.
- Trace output shows the mode (S/U) and reports each trap.

8. Software conventions (kernel ABI, not ISA)

Proposed for the first kernel, recorded here for reference:

- `SYSCALL` number in r1, arguments in r2–r5, result in r1.
- First calls: `exit(status)`, `putc(ch)`, `getc()`, `write(buf, len)`.

9. Review outcome (2026-10-11)

All five review points were accepted as proposed (D10): trap entry writes no memory; one handler entry point at VBR that dispatches on CAUSE; EPC is the address of the trapping instruction for every exception; user-mode I/O-page access traps in stage A; VBR = 0 stops the emulator on an exception.

Stage B: Interrupts and the Timer

Status: ACCEPTED (2026-10-11, D11). Adds what preemptive multitasking needs: hardware interrupts, a programmable timer, and a way to idle. Interrupts reuse the stage A trap mechanism unchanged.

B1. Interrupt lines

- **8 interrupt request lines**, IRQ0–IRQ7 (backplane signals; each device's line is set by a jumper, and devices may share one). Level-triggered: a line stays asserted until the device is acknowledged.
- **IRQ0 = timer** (B4). IRQ1–IRQ7 reserved (e.g. console receive, disk). The default jumpers (D50, D52): 1 disk, 2 UARTs 0 and 2, 3 UARTs 1 and 3, 4 PS/2, 5 real-time clock (D23), 6 USB, 7 network.
- Fixed priority: the lowest-numbered pending, enabled line is taken first.

New control registers:

#	Name	Contents
6	`IPEND`	read-only: bit n = IRQn currently asserted. Writes ignored
7	`IMASK`	bit n = 1 enables IRQn. Reset value 0

8–15 stay reserved (MMU in stage C). `IMASK` ignores bits 8–31.

B2. When an interrupt is taken

Between instructions: before fetching the next instruction, if `STATUS.IE` = 1 and `(IPEND & IMASK)` ≠ 0, the machine takes an interrupt instead of fetching. Trap entry is exactly stage A §5, with:

- `EPC` ← address of the next instruction, **not yet executed** — so `RTE` executes it, and "EPC = the instruction that was interrupted" holds for traps and interrupts alike;
- `CAUSE` ← `0x8000_0000 | v << 8 | n` for IRQn, where *v* is the **vector** the interrupting card gave in the acknowledge cycle (D57): its window number × 16 + its asking device's interrupt status bit, or `0xFF` when no card answered;
- `IE` ← 0 (as for every trap), so interrupts do not nest unless the handler saves `EPC`/`STATUS` and sets `IE` again.

An instruction is never interrupted half-way.

Interrupt shadow (D20, 2026-10-13): the check after an instruction uses `IE` and `IMASK` as they were *before* that instruction if it was `MTCR` or `RTE` — so after enabling interrupts, or returning from a handler, one more instruction always runs before a pending interrupt is taken (as in the hardware, which decides in the instruction's last microcycle, before its writes take effect at the closing clock edge). After `RTE` this guarantees the interrupted program makes progress. Trap and interrupt entry never take an interrupt at their end.

B3. `WAIT` (opcode 0x75, privileged, format N)

Stops fetching instructions until some line in `IPEND & IMASK` is asserted, then continues with the next instruction (and, if `IE` = 1, takes the interrupt first). The kernel's idle loop uses it instead of spinning.

B4. Timer device (IRQ0)

A memory-mapped device in the I/O page (on the I/O card in hardware):

Address	Name	Access	Meaning
`0xFFFF_FF10`	`TCOUNT`	read/write	current count; counts down by 1 per tick
`0xFFFF_FF14`	`TRELOAD`	read/write	period: loaded into `TCOUNT` when it reaches 0
`0xFFFF_FF18`	`TCTRL`	read/write	bit 0 = enable (counting); bit 1 = pending (IRQ0 asserted) — write 1 to bit 1 to acknowledge

When enabled, `TCOUNT` decrements each tick; on reaching 0 it sets pending (asserting IRQ0) and reloads from `TRELOAD`. The handler acknowledges by writing `TCTRL` with bit 1 set (and bit 0 set to keep counting).

Tick: in hardware, a fixed divided clock (rate chosen with the clock, open decision 8). **In the emulator, one tick per instruction executed**, so interrupt timing is exactly repeatable and tests can check exact output. During `WAIT` the emulator advances the timer straight to its next expiry.

B5. Emulator v3 behaviour

- Interrupts as B2; `IPEND` bit 0 mirrors the timer's pending flag.
- `WAIT` with no possible wake-up (`IPEND & IMASK` = 0 and the timer disabled or masked) stops the emulator with a message instead of hanging.
- `-t` trace reports each interrupt.

B6. Kernel (software, not ISA)

The stage B kernel test: a few tasks in one address space, each in user mode with its own stack. The timer interrupt saves the running task's registers, `EPC` and user SP into its task record, picks the next task round-robin, restores that task's state and `RTE`s into it. Tasks never yield on their own, so their interleaved output proves preemption.

B7. Review outcome (2026-10-11)

All five review points were accepted as proposed (D11): interrupts only between instructions (`EPC` = next instruction, `CAUSE` = bit 31 | line); 8 level-triggered lines with fixed priority, `IMASK` plus `STATUS.IE`; the timer as a memory-mapped device; one emulator tick per instruction; a privileged `WAIT`.

Stage C: Virtual Memory (MMU)

Status: ACCEPTED (2026-10-11, D12). Completes D1 and D9: each process gets its own address space, and cannot reach another process's memory, the kernel, or the devices.

C1. Design in one paragraph

User-mode addresses are **virtual** and go through a small **TLB** (translation lookaside buffer) that maps 4 KiB virtual pages to physical pages. The TLB is **managed by software**: when an access finds no matching entry, the CPU raises a *TLB miss* exception and the kernel loads the right entry from its own page tables, then `RTE` re-executes the access (stage A already guarantees this is safe). Supervisor mode is **untranslated** — the kernel uses physical addresses directly — so the miss handler itself can never miss. This is the MIPS approach: the least hardware, with the page-table format left entirely to software.

Why not a hardware page-table walker? A walker means extra microcode and datapath to read page tables on every miss, and fixes the page-table format in hardware. On a microcoded TTL machine the software refill costs roughly 15–20 instructions per miss, and the hardware stays small: an SRAM, a comparator and a few gates.

C2. Address spaces

Mode	Addresses	Translation
User	`0x0000_0000`–`0x7FFF_FFFF` (2 GiB)	through the TLB
User	`0x8000_0000` and above	**trap** (PRIVILEGE, BADADDR = address) — this includes the stage A I/O-page rule
Supervisor	everything	**none**: virtual = physical

The backplane address bus always carries **physical** addresses (settles the D1 question). The physical address is 32 bits; this machine populates 1 MiB of ROM and up to 64 MiB of RAM, in 16 MiB memory cards (D52).

C3. Pages and the TLB

- **Page size 4 KiB.** A virtual address = virtual page number **VPN** (bits 31:12) + offset (bits 11:0).
- **Split instruction / data TLB (D13):** an **I-TLB** for instruction fetches and a **D-TLB** for loads and stores, each **64 entries, direct-mapped**: entry index = VPN bits 5:0. In hardware both halves are one 128-entry SRAM whose top address line selects fetch or data, plus one tag comparator — one lookup per access.
- **Why split:** one instruction may need two translations, its fetch and its data access. In a single direct-mapped TLB, when both pages fall in the same slot each refill evicts the other and the instruction can never complete (livelock; found in emulator testing, 2026-10-11). With the split, every instruction needs at most one entry in each half, so it always completes.
- **Address-space ID (ASID), 8 bits.** Each entry is tagged with the ASID of its process, and matches only when it equals the current `ASID` register. A context switch just changes `ASID`; the TLB needs no flush (until ASIDs are reused).

Entry format (as seen through `TLBHI` / `TLBLO`):

```
TLBHI:  VPN [31:12]                     | 0 [11:9] | I [8] | ASID [7:0]
TLBLO:  PFN [31:12] (physical page)     | 0 [11:3] | X [2] | W [1] | V [0]
```

- **V** valid · **W** writable · **X** executable. Read is allowed whenever V = 1. (X is checked by the I-TLB, W by the D-TLB.)
- **I** (in `TLBHI` only) selects the half: 1 = I-TLB, 0 = D-TLB.
- No hardware accessed/dirty bits: the kernel can track dirty pages by mapping them read-only first and catching the protection fault.

C4. New control registers and instruction

#	Name	Contents
8	`ASID`	current address-space ID (bits 7:0)
9	`TLBHI`	entry tag for `TLBW`; **set by the hardware on every TLB miss or protection fault** to the faulting VPN, I = 1 for a fetch (0 for a load/store), and the current ASID
10	`TLBLO`	entry data for `TLBW`

11–15 stay reserved.

OP	Mnemonic	Privileged	Operation
0x76	`TLBW`	yes	write `TLBHI` / `TLBLO` into the entry at index `TLBHI` bits 17:12 (VPN bits 5:0) of the I-TLB if `TLBHI.I` = 1, else of the D-TLB

Because the hardware pre-loads `TLBHI` on a miss (including the I bit), the refill handler only looks up the page-table entry, writes `TLBLO`, executes `TLBW` and `RTE` — the entry lands in the half that missed. To invalidate the whole TLB, write all 128 entries (64 with I = 1, 64 with I = 0) with V = 0. TLB contents are undefined at reset, so the kernel invalidates it before first entering user mode. For a BUSERROR in user mode, BADADDR is the *physical* address.

C5. Exceptions (codes 8–11)

Code	Name	Raised when	BADADDR	TLBHI
8	`TLBMISS_X`	instruction fetch finds no valid matching entry	virtual address	set
9	`TLBMISS_R`	load finds no valid matching entry	virtual address	set
10	`TLBMISS_W`	store finds no valid matching entry	virtual address	set
11	`PROTECT`	store to an entry with W = 0, or fetch from X = 0	virtual address	set

A matching entry with V = 0 counts as a miss. As for every exception, the faulting instruction has no effect and `EPC` points at it.

Order of checks for a user-mode access: alignment (MISALIGNED) → upper half (PRIVILEGE) → TLB (miss / PROTECT) → physical bus (BUSERROR).

C6. Emulator v4 behaviour

- Models the split 64 + 64-entry direct-mapped TLB exactly, so a kernel that works in the emulator works on the hardware.

- Supervisor mode is untranslated (all existing tests keep working unchanged; they never enter user mode with translation, except the stage A/B kernels, which are updated to map their user pages).
- `-t` trace reports TLB misses and writes.

C7. Kernel (software, not ISA)

Proposed for the stage C kernel test:

- **Page tables:** per process, kernel-defined (first version: one linear table covering the first 16 MiB of user space, entries in `TLBLO` format, so a refill is a single indexed load).
- **Processes:** each has its own ASID, page table, stack and data pages. Program code stays in ROM, mapped read-only + executable into every process; data and stack pages are private RAM frames — two processes use the **same virtual addresses for different physical memory**.
- **System calls translate user pointers** through the caller's page table and check permissions (closing the stage A `write()` hole).
- **Tests:** two processes at the same virtual addresses keep separate data; a process that touches an unmapped page, writes a read-only page, or reaches above 2 GiB is killed while the others continue; the TLB is shared between processes without flushing (ASIDs).

C8. Review outcome (2026-10-11)

All six review points were accepted as proposed (D12): software-managed TLB; translation in user mode only; 4 KiB pages with a 64-entry direct-mapped, ASID-tagged TLB; user space is the lower 2 GiB; V/W/X permission bits only; physical addresses on the backplane.

Amended by D13 (2026-10-11): the TLB is split into a 64-entry I-TLB and a 64-entry D-TLB (see C3), after emulator testing showed that a single direct-mapped TLB can livelock.

P32W instruction reference

docs/instruction-reference.md

Generated by `generators/gen_isaref.py` *from* `microcode/p32isa.py` *— do not edit.* Encodings: isa-v0.md; system instructions: isa-system.md; how to use them: programmers-guide.md.

Word: `OP[31:24] A[23:20] B[19:16] C[15:12] IMM16[15:0]` (or `OFF24[23:0]`). **P** = privileged (traps PRIVILEGE in user mode). Branch and jump offsets count words from PC + 4.

Op	Mnemonic	Operands	Format	P	Meaning
`01`	**HALT**	``	N	P	stop the machine (emulators: exit status = r1 & 0xFF)
`10`	**ADD**	`rd, rs1, rs2`	R		rd = rs1 + rs2
`11`	**SUB**	`rd, rs1, rs2`	R		rd = rs1 − rs2
`12`	**AND**	`rd, rs1, rs2`	R		rd = rs1 & rs2
`13`	**OR**	`rd, rs1, rs2`	R		rd = rs1 | rs2
`14`	**XOR**	`rd, rs1, rs2`	R		rd = rs1 ^ rs2
`15`	**SLL**	`rd, rs1, rs2`	R		rd = rs1 << (rs2 & 31)
`16`	**SRL**	`rd, rs1, rs2`	R		rd = rs1 >> (rs2 & 31), zero fill
`17`	**SRA**	`rd, rs1, rs2`	R		rd = rs1 >> (rs2 & 31), sign fill
`18`	**SLT**	`rd, rs1, rs2`	R		rd = (rs1 < rs2) signed ? 1 : 0
`19`	**SLTU**	`rd, rs1, rs2`	R		rd = (rs1 < rs2) unsigned ? 1 : 0
`1A`	**NOR**	`rd, rs1, rs2`	R		rd = ~(rs1 | rs2)
`20`	**ADDI**	`rd, rs1, simm16`	I		rd = rs1 + sext(imm)
`22`	**ANDI**	`rd, rs1, uimm16`	IU		rd = rs1 & zext(imm)
`23`	**ORI**	`rd, rs1, uimm16`	IU		rd = rs1 | zext(imm)
`24`	**XORI**	`rd, rs1, uimm16`	IU		rd = rs1 ^ zext(imm)
`25`	**SLLI**	`rd, rs1, shamt`	SH		rd = rs1 << shamt
`26`	**SRLI**	`rd, rs1, shamt`	SH		rd = rs1 >> shamt, zero fill
`27`	**SRAI**	`rd, rs1, shamt`	SH		rd = rs1 >> shamt, sign fill
`28`	**SLTI**	`rd, rs1, simm16`	I		rd = (rs1 < sext(imm)) signed
`29`	**SLTIU**	`rd, rs1, simm16`	I		rd = (rs1 < sext(imm)) unsigned
`2E`	**LI32**	`rd, value32`	L32		rd = the next word (two words; rd in field B)
`2F`	**LUI**	`rd, uimm16`	LUI		rd = imm << 16
`30`	**MUL**	`rd, rs1, rs2`	R		rd = low 32 bits of rs1 × rs2
`31`	**MULH**	`rd, rs1, rs2`	R		rd = high 32 bits, signed × signed
`32`	**MULHU**	`rd, rs1, rs2`	R		rd = high 32 bits, unsigned × unsigned
`34`	**DIV**	`rd, rs1, rs2`	R		rd = rs1 / rs2 signed, toward zero (÷0 → −1; 0x80000000/−1 → 0x80000000)
`35`	**DIVU**	`rd, rs1, rs2`	R		rd = rs1 / rs2 unsigned (÷0 → 0xFFFFFFFF)
`36`	**REM**	`rd, rs1, rs2`	R		rd = rs1 % rs2 signed, sign of rs1 (÷0 → rs1; 0x80000000 % −1 → 0)
`37`	**REMU**	`rd, rs1, rs2`	R		rd = rs1 % rs2 unsigned (÷0 → rs1)
`40`	**LB**	`rd, simm16(rs1)`	LD		rd = sext(byte at rs1 + imm)
`41`	**LH**	`rd, simm16(rs1)`	LD		rd = sext(halfword), aligned

Op	Mnemonic	Operands	Format	P	Meaning
`42`	**LW**	`rd, simm16(rs1)`	LD		rd = word, aligned
`44`	**LBU**	`rd, simm16(rs1)`	LD		rd = zext(byte)
`45`	**LHU**	`rd, simm16(rs1)`	LD		rd = zext(halfword), aligned
`48`	**SB**	`rs, simm16(rb)`	ST		byte at rb + imm = rs
`49`	**SH**	`rs, simm16(rb)`	ST		halfword = rs, aligned
`4A`	**SW**	`rs, simm16(rb)`	ST		word = rs, aligned
`4C`	**SWAP**	`rd, simm16(rs1)`	LD		exchange rd with the aligned word at rs1 + imm, atomically (D51)
`50`	**BEQ**	`ra, rb, label`	BR		if ra == rb: PC = PC + 4 + 4 × off
`51`	**BNE**	`ra, rb, label`	BR		if ra != rb
`54`	**BLT**	`ra, rb, label`	BR		if ra < rb signed
`55`	**BGE**	`ra, rb, label`	BR		if ra ≥ rb signed
`56`	**BLTU**	`ra, rb, label`	BR		if ra < rb unsigned
`57`	**BGEU**	`ra, rb, label`	BR		if ra ≥ rb unsigned
`60`	**JMP**	`label`	J		PC = PC + 4 + 4 × off24
`61`	**CALL**	`label`	J		push PC + 4; PC = PC + 4 + 4 × off24
`62`	**JR**	`rb`	JR		PC = rb (a multiple of 4)
`63`	**CALLR**	`rb`	JR		push PC + 4; PC = rb
`64`	**RET**	``	N		PC = pop
`70`	**SYSCALL**	``	N		trap SYSCALL (EPC = this instruction)
`71`	**RTE**	``	N	P	PC = EPC; S = PS; IE = PIE
`72`	**MFCR**	`rd, cr / cr, rs`	CR	P	rd = control register cr
`73`	**MTCR**	`rd, cr / cr, rs`	CR	P	control register cr = rs
`74`	**BREAK**	``	N		trap BREAK
`75`	**WAIT**	``	N	P	idle until (IPEND & IMASK) ≠ 0
`76`	**TLBW**	``	N	P	write TLBHI / TLBLO into the TLB entry TLBHI selects
`80`	**LBX**	`rd, rs1, rs2[, scale]`	X		rd = sext(byte at rs1 + rs2 × scale); scale 1, 2, 4 or 8
`81`	**LHX**	`rd, rs1, rs2[, scale]`	X		rd = sext(halfword at rs1 + rs2 × scale)
`82`	**LWX**	`rd, rs1, rs2[, scale]`	X		rd = word at rs1 + rs2 × scale
`84`	**LBUX**	`rd, rs1, rs2[, scale]`	X		rd = zext(byte at rs1 + rs2 × scale)
`85`	**LHUX**	`rd, rs1, rs2[, scale]`	X		rd = zext(halfword at rs1 + rs2 × scale)
`88`	**SBX**	`rd, rs1, rs2[, scale]`	X		byte at rs1 + rs2 × scale = rd
`89`	**SHX**	`rd, rs1, rs2[, scale]`	X		halfword at rs1 + rs2 × scale = rd
`8A`	**SWX**	`rd, rs1, rs2[, scale]`	X		word at rs1 + rs2 × scale = rd
`90`	**LBP**	`rd, off8(rs1), inc8`	P		rd = sext(byte at rs1 + off8); then rs1 += inc8
`91`	**LHP**	`rd, off8(rs1), inc8`	P		rd = sext(halfword at rs1 + off8); rs1 += inc8
`92`	**LWP**	`rd, off8(rs1), inc8`	P		rd = word at rs1 + off8; rs1 += inc8 (rs1 written last)
`94`	**LBUP**	`rd, off8(rs1), inc8`	P		rd = zext(byte at rs1 + off8); rs1 += inc8
`95`	**LHUP**	`rd, off8(rs1), inc8`	P		rd = zext(halfword at rs1 + off8); rs1 += inc8
`98`	**SBP**	`rd, off8(rs1), inc8`	P		byte at rs1 + off8 = rd; rs1 += inc8
`99`	**SHP**	`rd, off8(rs1), inc8`	P		halfword at rs1 + off8 = rd; rs1 += inc8
`9A`	**SWP**	`rd, off8(rs1), inc8`	P		word at rs1 + off8 = rd; rs1 += inc8

Control registers (MFCR / MTCR)

#	Name
0	`STATUS`
1	`EPC`
2	`CAUSE`
3	`BADADDR`
4	`VBR`
5	`USP`
6	`IPEND`
7	`IMASK`
8	`ASID`
9	`TLBHI`
10	`TLBLO`

Exception causes (CAUSE)

Code	Name
1	ILLEGAL
2	PRIVILEGE
3	MISALIGNED
4	BUSERROR
5	SYSCALL
6	BREAK
8	TLBMISS_X
9	TLBMISS_R
10	TLBMISS_W
11	PROTECT
0x8000_0000 | v << 8 | n	interrupt on IRQn, v = the card's vector (D57)

ISA additions: measured candidates

docs/isa-measurements.md

Status: MEASURED (2026-10-25). Ken chose the address adder, post-increment, scaled indexed and `LI32` (D59): built, and measured again below ("The outcome").

How it was measured

`tools/isamix.sh SESSION` runs a recorded session on `p32uemu` (the real control store, the three-bus datapath, D56) from a fresh image and streams its instruction trace (`p32uemu -I`: per instruction its PC, IR and microcycles) into `tools/isamix.c`. For each candidate the analyzer finds the instruction sequences a new instruction would replace, in the dynamic trace, and adds up the **cycles the replaced instructions actually took**, less what the new instruction would add. A register a fused instruction would no longer write must be **dead** afterwards (the trace writes it before reading it), so only sequences the compiler could really fuse count. The savings are estimates: the new instructions' cycle counts are assumed (below), and the compiler would have to emit them.

Workload	Session	Instructions	CPI	Kernel
Shell, files, kernel heavy	session1	25.5 M	3.58	90%
BASIC, `make basic` (the C compiler on P32W)	session12	599 M	3.58	17%
MicroPython	session15	17.2 M	3.62	84%
SQL	session11	43.6 M	3.61	68%

(Cycles without `WAIT`, the idle loop.)

Results: cycles saved, % of all cycles

Candidate	Assumed cost	shell	compile	Python	SQL
Post-increment `LW rd, off(rs)+k`: the access, then `rs += k`	one cycle more than the load / store (the second register write)	1.0	**2.8**	1.1	1.8
Scaled indexed `LW rd, [rs + rt << k]` (k = 0–3)	as a load / store; the ALU adds the shifted index (a new ALU function)	**3.5**	1.5	**2.0**	1.7
Unscaled indexed `LW rd, [rs + rt]`	as a load / store	0.2	0.7	0.3	0.3
Leaf calls with a link register (`JAL` 3 cycles, `JR lr` 4, instead of `CALL` 7, `RET` 6)	a register given up as the link	1.0	2.3	1.0	1.7
Compare with a constant and branch `BEQ ra, simm8, label`	as a branch; the label's reach shrinks to ±2 K words (a 12-bit offset)	1.0	1.9	0.8	0.9
LDM (epilogue: load registers, 2 cycles each)	a register-list sequencer on the CPU card	1.0	1.6	1.0	1.2
STM (prologue: store registers, 2 cycles each)	as LDM	0.8	1.3	0.8	1.0
LI32: `LUI` + `ORI` as one two-word instruction	microcode only (the second word read like a fetch)	1.0	1.3	1.0	1.3
`ADDSL rd = rs + (rt << k)` (an ALU function alone)	—	0.0	0.0	0.1	0.0
(compiler only) `LA` + `LW 0` → `LUI` + `LW lo`	none	0.3	0.6	0.4	0.4
(compiler only) `ADDI` folded into the offset	none	0.0	0.0	0.0	0.0

Calls: 62–71% of all calls are to **leaf** functions (no call inside).

Together (they overlap little): **about 10% fewer cycles on the compile, 8–11% on the others.** No single addition is worth more than 3.5%.

What the measurements say besides

- **Loads and stores take 42–54% of all cycles** (4 cycles each: the fetch, the address into the ALU, the ALU's result into MAR, the access). An **address adder** that puts `rs + offset` into MAR in the cycle the operands are read would make every load and store 3 cycles: **11–13% fewer cycles** on every workload, more than all the ISA additions together, with no ISA or compiler change. It is a datapath question for the CPU and MMU / data cards (a 32-bit adder on B and A into MAR, its carry chain inside the cycle at 5 MHz), to measure next.
- **Disk transfers by the CPU**: in the shell session `LHU` / `SH` (the CompactFlash's 16-bit data register into a buffer) are 13% of all instructions. A **DMA disk card** (already planned, for swap) takes those away entirely.
- `ADDI` is the most frequent instruction (18–22%): loop counters and pointer steps; post-increment absorbs part of it.

The outcome (D59)

Built: the address adder (every load and store 3 cycles), indexed loads and stores (`LWX` …, scale 1–8), post-increment (`LWP` …) and `LI32`, used by both compilers (`p[i]`, `*p++`) and by the assemblers' `LA` / `LI`. The same workloads, before and after (`tools/isamix.sh`; cycles without `WAIT`):

Workload	Instructions	Cycles before	Cycles after	Saved	CPI
shell (session1)	25.5 M → 23.7 M	91.6 M	75.9 M	**17.1%**	3.58 → 3.20
compile (session12)	599 M → 554 M	2,147 M	1,799 M	**16.2%**	3.58 → 3.25
Python (session15)	17.2 M → 17.1 M	62.2 M	54.1 M	**12.9%**	3.62 → 3.18
SQL (session11)	43.6 M → 40.3 M	157.4 M	129.6 M	**17.7%**	3.61 → 3.22
boot, root login, `halt`	7.49 M → 7.05 M	26.74 M	22.43 M	**16.1%**	3.57 → 3.18

1.19× faster at the same clock, a little more than the estimate (the address adder's 11–13% and the additions' few percent): `LI32` replaces every `LA` (2.5% of the compile's instructions), and fewer instructions also mean fewer fetches. Python gains least (12.9%), and its instruction count barely changed: the compilers found few `p[i]` and `*p++` there.