P32W Emulators and Tests

p32emu, p32uemu, the gate-level simulations, benchmarks

P32W — Ken Rother's 32-bit TTL computer

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

Contents

The P32W emulators: user guide and theory of operation `docs/emulators.md`

1. Which emulator, when
2. Building
3. Running: the command line
4. Interactive use
5. P32W Unix in scripts and on the second emulator
6. Disk images
7. Bare-metal programs
8. Debugging
9. Tests
10. Scripted sessions
11. The shared devices (`p32io.h`)
12. `p32emu`: the instruction-level reference
13. `p32uemu`: the microcode-level model
14. Keeping the two honest
15. Limits and differences from the hardware

emulator/ `emulator/README.md`

Build and run

Behaviour worth knowing

Tests (`test/run_tests.sh`, run by `make test`)

Hand-written assembly against C (first measurement, 2026-10-14) `docs/bench.md`

Where the gap comes from

Done after the measurement (2026-10-14)

What this suggests

The P32W emulators: user guide and theory of operation

docs/emulators.md

P32W has two emulators, both in `emulator/`:

	`p32emu`	`p32uemu`
Source	`emulator/p32emu.c` (D7)	`emulator/p32uemu.c` (D14)
Level	**instruction**: decodes each instruction and performs its effect in C	**microcycle**: runs the real control store, one microword per cycle, over a model of the CPU card and the MMU / data card
Knows the instruction set?	yes — it *is* the definition (the golden model)	no — every instruction's behaviour comes from the microcode
Counts	instructions	instructions **and** microcycles
Speed on an M3 Pro (measured below)	~125 M instructions/s on plain code	~70 M microcycles/s ≈ 22 M instructions/s

Both include the **same devices**, `emulator/p32io.h` (D50): the I/O card (four UARTs, the timer, the input and output ports, the real-time clock (D23), two USB ports (D43) and the network coprocessor (D45)), the disk card (two CompactFlash sockets and two IDE channels) and the video card (its HD63484 chip, modelled in `emulator/p32acrtc.h`, and its PS/2 section). The memory cards (`-M`) are modelled in each emulator. The two programs run the same images and take the same device options, and `make test` checks that they agree at every instruction.

This document has two parts. **Part 1** covers using the emulators: building them, the options, interactive use, disks, bare-metal programs, debugging and scripted tests. **Part 2** explains how each emulator works inside and how the two are kept in agreement. Everything here was checked against the code (`p32emu.c`, `p32uemu.c`, `p32io.h` and `test/run_tests.sh` as of 2026-10-24). The examples were run on an Apple M3 Pro, on scratch copies of the images, on 2026-10-14, and the `p32uemu` runs and the measurements on 2026-10-24; instruction and cycle counts in them can differ slightly from today's builds.

Part 1 — User guide

1. Which emulator, when

Use `p32emu` by default. It is the reference: it is fast, it has an instruction trace (`-t`), and when it reports an error it shows the instruction that caused it. Use it for running P32W Unix, for developing software, and for tests.

Use `p32uemu` when the question is about the hardware. It runs the 53-bit microwords that will go into the seven control-store flash chips. It reports how many microcycles a program takes (at 3 MHz that is the real machine's run time, D17). It can model the I/O card's timer cycle for cycle (`-C`), bus wait states (`-W`) and a DMA master competing for the bus (`-D`), and it shows whether the microcode — not the C code — implements the ISA. A program that works on `p32emu` but not on `p32uemu` points to a microcode bug, or to a gap in the model of the CPU card or the MMU / data card.

Measured speed

All runs used the same images on an Apple M3 Pro, with both emulators built `-O2` by Apple clang 21. The C loop and the boot were each run two or three times and varied by under 3%; session1 ran once.

Workload	Instructions	`p32emu`	`p32uemu`	Microcycles	`p32uemu` slower by
Stand-alone C loop (`--sa`, an array sum, no I/O)	107,666,722	0.87 s = **124 M instr/s**	4.80 s = **68 M cycles/s** (22.4 M instr/s)	328,143,482 (3.05 per instruction)	**5.5×**
Boot P32W Unix, log in as root, `halt` (`-w` script)	7,490,575	0.33 s (23 M instr/s)	0.65 s (41 M cycles/s, 11.5 M instr/s)	26,738,361 (3.57 per instruction)	2.0×
`software/tests/session1` (boot, log in, 30 shell commands, `halt`; `-w`)	25,586,345	1.30 s (20 M instr/s)	2.60 s (35 M cycles/s, 9.8 M instr/s)	91,699,408 (3.58 per instruction)	2.0×

On plain computation `p32uemu` is about **5.5 times slower** than `p32emu`. Under P32W Unix the gap is only 2 times, and the reason is in `p32emu`: with interrupts enabled, both emulators ask the I/O card for pending lines once per instruction (`io_ipend()`). Under `-w` that call runs the prompt matcher on every UART poll. A `sample` profile of `p32emu` running `session1` shows `prompted()`, `memcmp` and `strlen` as the top entries. So scripted P32W Unix sessions measure the I/O model more than the CPU model.

For comparison, the real machine runs a 3 MHz microcycle (D17). At that rate the boot-and-halt run's 26.7 M microcycles take **about 9 s**, session1 **about 31 s**, and the C loop **about 109 s**.

2. Building

```
make -C emulator            # both emulators (and the generated files they need)
```

The `emulator/Makefile` does three things:

Target	Made by	From
`p32opc.h` — opcodes, formats, must-be-zero masks, the privileged set, control-register numbers and names, exception causes and names	`generators/gen_p32opc.py`	`microcode/p32isa.py` (the ISA's single source of truth)

Target	Made by	From
`p32uc.h` — the microword's width (`UC_BITS`) and field layout (`UF_*`), the codes of every field (`ASRC_*`, `BSRC_*`, `RSRC_*`, `RDST_*`, `RSEL_*`, `ALU_*`, `IMM_*`, `MEM_*`, `SEQ_*`, `COND_*`, `MODE_*`), internal-sequence numbers, the physical register-file addresses (`PR_*`), the control-register decode table	`microcode/p32ucode.py --header`	`p32isa.py` + the generator
`microcode/p32ucode.bin` — the control store, 16,384 words × 8 bytes (little-endian; 53 bits used). The same run writes `u0.bin`–`u6.bin` (one 16 KiB image per flash chip, byte n of each word) and `dec_op.bin` / `dec_cr.bin` (the CPU card's decode flash)	`microcode/p32ucode.py -o ../microcode`	the same
`p32emu`	`cc -O2 -std=c99 p32emu.c`	`p32emu.c`, `p32opc.h`, `p32io.h`
`p32uemu`	`cc -O2 -std=c99 p32uemu.c`	`p32uemu.c`, `p32opc.h`, `p32uc.h`, `p32io.h`

Never edit `p32opc.h` or `p32uc.h`. Change `microcode/p32isa.py` (the ISA) or `microcode/p32ucode.py` (the microcode) and rebuild. The microcode generator refuses to write images that break its static checks (§13.5). `python3 microcode/p32ucode.py --stats` prints the cycle count of every instruction.

`make -C emulator test` runs the whole regression suite (§9). `make -C emulator clean` removes the emulators, the generated headers, the microcode images and the test outputs.

3. Running: the command line

```
p32emu  [-t] [-m] [-k FILE] [-M N] [-s FILE] [-l N] [device options] image.bin
p32uemu [-u UCODE] [-s FILE] [-l N] [-c | -C] [-G [-q FILE]] [-W N] [-D ADDR:PERIOD] [-M N] [device options] image.bin
```

`image.bin` is a **boot ROM image**. It is loaded at physical address 0 (1 MiB at most) and the CPU starts there in supervisor mode, with interrupts off and `VBR` = 0. Options and the image can appear in any order. Numbers for `-l`, `-p` and `-R` are read with C's base-0 rules: `0x1F` is hex, `010` is **octal**, and anything else is decimal.

Options of both emulators

Option	Meaning
`-s FILE`	Write the **architectural state at the start of every instruction** to FILE, one line each, in the same format for both emulators (§8.2). This is the basis of the cross-check.
`-l N`	Stop after N instructions with a register dump and status 255. **Default: 100,000,000**. When the console is an interactive terminal (stdin is a tty and `-w` is not given) the default is **no limit**, so a session runs until `halt` or ^]. `-l 0` means no limit. Instructions are counted; trap and interrupt entry are not.
`-M N`	**Memory cards** fitted: 1 (default) to 4, 16 MiB each, in the memory windows 0 … N−1 (D52). RAM runs from 1 MiB up to N × 16 MiB; an address above it gets no acknowledge, a BUSERROR (the kernel's memory probe relies on this). `-M 16,0,6`: each window's MiB instead — a card with only some of its 2 MiB banks fitted (D60), 0 for no card; window 0 needs one. An address in a bank that is not fitted is a BUSERROR, so memory may have holes.
`-d IMG`	The disk card's first CompactFlash socket (port 0): a raw image of 512-byte sectors, **opened read/write and written in place** (§6). The file must exist; its size sets the card's capacity (whole sectors). Without `-d` the socket is empty.
`-d1 IMG`	The second CompactFlash socket (port 1).
`-d2 IMG`, `-d2s IMG`	IDE channel A (port 2): the master and the slave drive.
`-d3 IMG`, `-d3s IMG`	IDE channel B (port 3): the master and the slave drive.
`-a OUT`	UART 1's output goes to the file OUT. `-a2`, `-a3`: the same for UARTs 2 and 3.
`-A IN`	UART 1's input comes from the file IN. `-A2`, `-A3`: UARTs 2 and 3.

Option	Meaning
`-T PORT`	UART 1 is a TCP server on `localhost:PORT` (loopback only, one connection at a time): a second terminal (§4.3). `-T2`, `-T3`: UARTs 2 and 3.
`-p N`	The value the input port (`FFFF_FF20`) reads (8 bits; default 0).
`-U IMG`	A USB drive in the I/O card's first USB port (CH376S, D43), holding IMG (read/write). `-Uout N` pulls it out after N sectors have moved.
`-U1 IMG`	A USB drive in the second USB port.
`-N`	The network coprocessor (D45), carried out with the Mac's sockets (10.0.2.2 is the Mac; net-mailbox.md). `-Nport N` moves listening ports below 1024 up by N (default 8000); `-Nca FILE` names the certificates TLS trusts. Without `-N` the coprocessor is present but not running.
`-V FILE`	**Fits the video card** and writes its screen at exit: the picture's 640 × 480 dots to `FILE.pbm` (once the chip is started), and to FILE the console's 25 rows of 80 characters, read from the dots: each 8 × 16 cell on lines 40–439 is matched against the font's glyphs, plain, bold, underlined or inverse, with the cursor or without (`?` where nothing matches).
`-K FILE`	**Fits the video card** and has its PS/2 keyboard type FILE's text. The keys go as set-2 scan codes into the model of the card's PS/2 section, which turns them into ASCII as the ATmega328P's firmware does. What is typed waits until the section's queue (63 bytes) has room, so a long file loses nothing.
`-Kr`	**Fits the video card** with its PS/2 section **ready from the reset**. Without it the section's ATmega328P starts as the real one does: about two seconds (its bootloader, then the keyboard's and the mouse's resets) in which nothing is FULL and a command written is dropped or waits (D109). For a test that is compared with the gate-level machine, whose ATmega is always ready.
`-Km B0,B1,B2`	**Fits the video card**; its PS/2 mouse sends this 3-byte packet (hex), after the keys. Without `-V`, `-K`, `-Km` or `-dbg` there is no video card: its window does not answer.
`-w`	Scripted input: each input line is held back until the output ends with a prompt (§10.1). This also **turns off raw/interactive mode**.
`-P STR`	Adds a prompt for `-w`. The first `-P` replaces the default list. At most 12 prompts.
`-R SECS`	The real-time clock starts at SECS (seconds since 1970, UTC) and then advances with *emulated* time. Without it, the clock starts at the host's current time (§4.6).

`p32emu` only

Option	Meaning
`-t`	Trace every instruction to stderr (address, mode S/U, the instruction word and its disassembly), every trap and interrupt, every `TLBW`, and every write to the output port. It also flushes UART output after every byte, so program output and trace lines appear in order.
`-m`	TLB statistics at exit, on stderr: for the I-TLB and the D-TLB, the user-mode accesses, the misses (cold: the slot was empty; conflict: another page held it), the miss rate, protection faults, `TLBW`s and valid entries replaced. Then, for the same accesses, the misses of four "what if" TLBs that assume every miss is refilled: 128 entries direct-mapped, 64 entries 2-way, 128 entries 2-way (LRU), and 64 entries indexed by `VPN ^ VPN >> 6`. In scripts: `EMUFLAGS=-m software/tests/run_session.sh NAME` prints the report after the transcript.
`-k FILE`	A kernel profile at exit, to FILE: how many instructions ran at each supervisor-mode address in the kernel's MiB (`0010_0000`–`001F_FFFF`), plus totals for user mode and for other supervisor code (the ROM). `tools/kprof.py` totals it by function.

`p32uemu` only

Option	Meaning
`-u FILE`	The control-store image. **Default `../microcode/p32ucode.bin`, relative to the current directory**, which is right only when you run from `emulator/`. From the repository root, pass `-u microcode/p32ucode.bin`.
`-c`	The timer ticks once per **microcycle**, instead of once per instruction.
`-C`	The I/O card's timer **exactly as the hardware does it**: a 1 MHz tick every third 3 MHz microcycle (with `-F`, every fourth 4 MHz one: the CPU card's clock jumper, D74), reload one cycle after reaching 0, and `WAIT` holds its microcycle instead of skipping time (§13.6); and the disk card's **two wait states** on every port access (D66), the video card's three on every access to its chip and one on a PS/2 command written (D104, D106). The gate-level simulation of the device cards compares against this. If `-C` is given, `-c` has no effect.
`-G`	**The measuring gate** (implies `-C`; §4.7, D85): while bit 7 of the output port is up it counts the machine's cycles, and reports when the bit falls.
`-q FILE`	With `-G`, **a profile** of what runs while the gate is up, written at exit: the cycles at each address, in user mode and in the kernel (§4.7, D88, D92).

Option	Meaning
`-W N`	**N wait states** on every bus cycle (a fetch, load or store): the bus's `-WAIT` (D50). The sequencer holds the microcycle, so they show only in the cycle count and in timer ticks (with `-c` or `-C`).
`-D ADDR:PERIOD`	**A DMA master** on the bus, to test the bus lock (D54): every PERIOD microcycles it asks for the bus; the arbiter grants it at the next microcycle boundary unless `-LOCK` is held, and it adds 1 to the RAM word at physical ADDR (a read and a write: two bus clocks, which a CPU bus cycle in that microcycle waits for). At `HALT` it reports how many it added and the word's value (§13.5). `-Dnolock ADDR:PERIOD`: the arbiter ignores `-LOCK`, so a `SWAP` can lose an addition.

`p32uemu` has no `-t`. Both emulators reject an unknown option, or a second image name, by printing the usage message and exiting with status 255.

Exit status and the final line

How the run ends	stderr	Exit status
`HALT`	`HALT at PC, r1=XXXXXXXX (N instructions[, M idle ticks])` (p32emu) / `(N instructions, M cycles)` (p32uemu)	`r1 & 0xFF`
An exception or interrupt while `VBR` = 0 (no handler)	`EXCEPTION: …` and a register dump	255
The instruction limit	`STOP: instruction limit …` and a dump	255
`WAIT` that can never wake up	`STOP: WAIT with no possible wake-up` and a dump	255
^] typed on the console (interactive)	`[p32emu: ^] -- stopped]`	0
Usage error, missing or oversized image, a file that cannot be opened, `-T` cannot bind, a control store of the wrong size	a message	255
`p32uemu` only: a microword with an undefined field code	`microcode: undefined …` and a dump	255

`HALT` is privileged, so a user-mode program cannot stop the emulator; it gets a PRIVILEGE trap instead. In `p32emu`, "idle ticks" are timer ticks that `WAIT` skipped (§12.6), and they are printed only when there were any. `p32uemu`'s HALT line does not report idle ticks.

```
$ ./p32emu hello.bin; echo "status=$?"
P32W lives! instruction-level reference emulator
HALT at 00000014, r1=00000000 (254 instructions)
status=0
$ ./p32uemu -u p32ucode.bin hello.bin; echo "status=$?"
P32W lives! instruction-level reference emulator
HALT at 00000014, r1=00000000 (254 instructions, 868 cycles)
status=0
```

4. Interactive use

4.1 The console

UART 0 is the console, connected to the emulator's stdin and stdout. When stdin is a terminal and `-w` is not given, the emulator starts in **interactive mode** and prints this to stderr:

```
[p32emu: interactive; ^] stops the emulator]
```

(`p32uemu` prints the same text, with "p32emu" in it, because the message comes from the shared `p32io.h`.) In interactive mode:

- The Mac terminal is put in **raw mode**: no echo, no line editing, no signals, no CR→NL translation, no XON/XOFF. Every key goes straight to UART 0, so ^C, ^Z, the arrow keys and `vi` reach P32W, and P32W does its own echo. Return arrives as CR.
- stdout is unbuffered.
- **^] (Ctrl-]) stops the emulator** at once, with status 0. The terminal is restored on exit. ^] is checked only on the console; on the TCP terminal it is passed to P32W.
- There is **no instruction limit** unless you give `-l`.
- Idle time is real time (§4.6).

The terminal is read on every 256th poll of the UART (a few hundred instructions at most), so the delay is not noticeable.

4.2 The boot monitor and booting P32W Unix

The ROM image you normally run is the **boot monitor**, `software/monitor/monitor.asm`. It sets `VBR` to its own exception handler, sets UART 0 to 9600 baud (the emulator ignores the divisor), and prompts with `> `. Commands, with numbers in hex:

Command	Effect
`d ADDR [LEN]`	dump memory as words (16 bytes by default)
`m ADDR VALUE`	store a word
`g ADDR`	call ADDR; the program may `RET` to the monitor, which then shows r1
`l LBA COUNT ADDR`	load COUNT sectors from the CompactFlash card to ADDR
`b`	boot from the card: sector 0 is a header (`"P32B"`, sector count, load address, entry point); sectors 1.. are loaded and the entry point is called
`?`	help
`q`	`HALT` with r1 = 0 (the emulator exits with status 0)

If a command causes an exception (a bad address, for example), the monitor prints CAUSE, EPC and BADADDR and goes back to the prompt. Its line buffer and stack are at `0x001F_FC00`, the top of the first memory card's first 2 MiB bank (every machine has it, D60).

To boot P32W Unix (see also unix.md):

```
make -C emulator
make -C software disk                      # software/build/disk.img
python3 assembler/p32asm.py software/monitor/monitor.asm -o monitor.bin
emulator/p32emu -d software/build/disk.img monitor.bin
```

Type `b` at `>`. The kernel loads and the system goes multi-user. Log in on the console as `root`, `ken` or `guest` (no password); the passwords are listed in unix.md, "Logging in".

The same session, scripted (not interactive, so the transcript can be shown; the image is a scratch copy):

```
$ printf '?\nl 0 1 100000\nd 100000 10\nq\n' | ./p32emu -d m.img monitor.bin | tr -d '\r'
HALT at 000000BC, r1=00000000 (5406 instructions)

P32W monitor -- ? for help
> ?
d ADDR [LEN]      dump memory
m ADDR VALUE      store a word
g ADDR            call a program
l LBA COUNT ADDR  load sectors from the disk
b                 boot from the disk
q                 halt
> l 0 1 100000
loaded 00000001 sectors at 00100000
> d 100000 10
00100000: 42323350 00000083 00100000 00100000
> q
```

The dump shows the boot header: the magic `"P32B"` (`0x42323350`), `0x83` kernel sectors, load address `0x0010_0000`, entry `0x0010_0000`. The HALT line comes first only because stdout went through `tr` while stderr went straight to the terminal.

Halting cleanly. As root, use `halt`, `reboot` or `shutdown [-r]`. These write every buffer to the disk, mark the filesystem clean, and then execute `HALT`, so the emulator exits with status 0. `reboot` and `shutdown -r` restart into the monitor instead, and you type `b` again. **^] is pulling the plug.** The disk image keeps whatever the kernel had already written, the superblock's *clean* flag stays 0, and the next boot reports that the filesystem was not unmounted cleanly. For example, after a boot that was stopped with ^] at the `login:` prompt:

```
superblock word 6 ("clean"), read with a few lines of Python at sector 2048:
disk.img  (the original)        clean=1
a.img     (a run ended by halt) clean=1
t.img     (a run ended by ^])   clean=0
```

4.3 The second terminal (`-T`)

`-T PORT` makes UART 1 a TCP server on `localhost:PORT`. The emulator prints `[UART 1: nc localhost PORT]` and then `[UART 1: connected]` / `[UART 1: disconnected]` as clients come and go. Only one client can be connected at a time. When it hangs up, the port waits for the next one. P32W Unix runs a `getty` on `/dev/tty1`, so this gives a second user:

```
emulator/p32emu -T 2323 -d software/build/disk.img monitor.bin
python3 tools/p32term.py           # in another window; default port 2323
```

- `tools/p32term.py [port]` puts the Mac terminal in raw mode and sends every key at once, as the console does, so `vi`, `top`'s `q`, `more` and ^C all work. **^]** leaves p32term; the emulator keeps running. If the emulator is not running yet, p32term retries every 2 s.
- `nc localhost 2323` also works, but the Mac's line discipline holds each line until Return, and ^C ends `nc` instead of reaching P32W.
- **Output sent while nobody is connected is lost.** It is not buffered. If you connect after the system has booted, the `login:` prompt has already gone. Press Return and `getty` prints it again:

  ```
  tty1 before Return: b''
  tty1 after Return:  b'\r\n\r\nP32W Unix (tty1)\r\n\r\nlogin: '
  ```

Alternatively, start `p32term.py` first, since it waits for the emulator.

- The TCP terminal is also the path for **Kermit file transfer**: C-Kermit's `set host /raw-socket localhost PORT`, or `tools/pykermit.py --tcp PORT send|get|dir|finish` (unix.md, "Moving files with Kermit").

A two-user run, driven through a pseudo-terminal (console) and a socket (tty1), with ken logged in on tty1 and root on the console:

```
# who
root     tty0     Oct 14 17:19
ken      tty1     Oct 14 17:19
# sleep 2; date
Wed Oct 14 17:19:25 2026
# halt

P32W Unix: halted
HALT at 0010025C, r1=00000000 (13441216 instructions, 17115522 idle ticks)
```

`sleep 2` took 2.6 s of wall-clock time: idle time was real time (§4.6). The 17 million idle ticks are the emulated microseconds the CPU spent in `WAIT`.

4.4 Files as the second UART (`-A`, `-a`)

Without `-T`, UART 1 can read from a file (`-A IN`) and write to one (`-a OUT`). A file input is "typed" as fast as P32W reads it (or, with `-w`, a line per prompt, using UART 1's own output to detect the prompts). At end of file, UART 1 simply never has data again. The two-user session test uses `-A tests/session2.tty1 -a tty1.out`.

4.5 What happens to the disk image

`-d IMG` opens the image read/write. **Every sector the program writes goes to the file at once** (`fwrite` and `fflush` per sector), so the image always holds what the emulated card holds. There is no copy-on-write and no undo. Keep a copy if you need the old state.

Never point two emulators at the same image, and never change an image with `p32fs.py` while an emulator is using it. Neither the emulators nor `p32fs.py` lock the file. The kernel caches blocks and the superblock in memory, so the two writers would overwrite each other and corrupt the filesystem. The test scripts always build or copy their own image (`run_session.sh` builds one in `$TMPDIR`; `make -C software testdisk` makes `build/test.img`). Use a copy in the same way for experiments. `software/build/disk.img` is the image Ken boots.

4.6 Time: the timer, idle waiting and the real-time clock

The timer counts one tick per instruction (§12.5). P32W Unix programs it for 50 Hz: `TRELOAD` = 20,000 ticks (`kernel/param.h`: `HZ` 50, `TICK` 20000). So in the emulators one instruction counts as one microsecond of emulated time. The real machine runs P32W Unix at about 0.94 MIPS (3.18 microcycles per instruction at 3 MHz, §13.2), so emulated time runs a little fast.

Idle. When the kernel has nothing to run it executes `WAIT`. The emulator handles this in one of two ways:

- **Interactive** (the console is a terminal, or `-T` is in use): it sleeps in `select()` on the terminal (and the TCP socket) for as many microseconds as the timer has left, and wakes early if a key arrives. Idle emulated time therefore passes at **real speed**, and `sleep 10` takes ten seconds.
- **Not interactive** (piped or scripted input, no `-T`): it **jumps straight to the timer's expiry**. A `sleep 60` in a scripted session costs almost no host time. This is what keeps the tests fast and repeatable.

`p32uemu -C` does neither: it holds the `WAIT` microcycle until a line comes up, simulating every cycle at full host speed.

The real-time clock (D23, a DS12C887 model at `FFFF_FF40/41`):

- By default it reads **the host's clock (UTC) at the moment P32W first reads it**, and from then on advances with *emulated* time: instructions plus idle ticks, one microsecond each. During interactive idle that matches real time. During heavy computation the emulator runs faster than 1 MIPS, so P32W's clock gains on the Mac's. P32W Unix reads the clock only at boot and counts its own ticks after that, so the system time behaves the same way.
- `-R SECS` starts it at a fixed time (`-R 1791936000` = 2026-10-14 00:00 UTC, as all the tests use). Together with non-interactive input, this makes every run exactly repeatable.
- Setting the clock (`date 2610141230` as root) changes it for the rest of this run only. Nothing is saved, and the 114 bytes of battery-backed RAM are also lost when the emulator exits.
- Under `p32uemu -C` the clock does not advance at all (§15).

4.7 Measuring a run: `p32uemu -G` and `cycles`

To find how long something takes on the real machine, boot under `p32uemu -G` and run it through `cycles` (D85). Never point this at `disk.img` while you need what is on it: a build writes to the disk. Use a copy.

```
$ cp software/build/disk.img m.img
$ emulator/p32uemu -u microcode/p32ucode.bin -G -d m.img monitor.bin
...
# cycles make
```

`cycles command` raises bit 7 of the output port, runs the command, lowers the bit, and prints the command's real, user and system time. For several steps use `cycles sh -c '...'`, or `cycles start`, the steps and `cycles stop`. With `-G` the emulator counts while the bit is up, and when it falls it prints on its standard error:

```
[cycles #1: 32,107,476 cycles, 0:00:10.7 at 3 MHz (10.702 s)
 9,772,072 instructions, 0.91 MIPS busy; idle 0 cycles (0.0%)
 disk 152 sectors read, 0 written, 18 commands; console 6,143 bytes]
```

`MIPS busy` leaves the idle cycles out. A count still open when the machine stops is reported then. The counter is 64 bits: 32 would wrap every 23.9 minutes at 3 MHz, and 64 last 195,000 years. Under `-G` the emulator does about 30 million cycles a second on P32W Unix on an M3 Pro, some 10 times the real machine's speed.

What it covers. The processor's work exactly: the count is `p32uemu -C`'s, which the gate-level simulations match cycle for cycle, wait states included. It counts the whole machine, not one process, so anything else running is in the total. Devices take no time (§15): to the count add the drive's access time for each disk command, and the console's bytes at its baud rate.

Where the time goes. Add `-q FILE` and the emulator also keeps the cycles at each address while the gate is up, in user mode (D88) and in the kernel (D92). `tools/uprof.py FILE MAP` totals them by function, the static ones too, from the program's link map (`p32ld -M`) and the objects it names, run from where the map's paths start:

```
$ cd software
$ python3 ../assembler/p32ld.py build/crt0.o build/cc/*.o build/libc.a -M cc.map -o /tmp/cc.x
$ python3 ../tools/uprof.py prof cc.map
```

Addresses are a process's, so profile one program at a time: `cycles cc -S x.c` is `cc` alone, but `cycles cc -c x.c` runs `as` too, and the two would add up at the same addresses.

The kernel has one set of addresses, so `-k` totals a whole job, every process in it, in cycles (`p32emu -k` and `kprof.py` count instructions, from boot):

```
$ cd kernel
$ python3 ../tools/uprof.py prof unix.map -k
```

It is for the emulator only, not a hardware feature. `cycles` tells the emulator when to count by writing the output port, so on the real machine it would only set that port and clear it. There, use `time`.

5. P32W Unix in scripts and on the second emulator

Everything in §4 works the same with `p32uemu` (add `-u microcode/p32ucode.bin` when running from the repository root). It boots P32W Unix, runs the shell, and writes the same disk image as `p32emu` does. Two scratch copies of the image, booted with the same script and the same `-R`, came out **byte-identical** after `halt`:

```
$ ./p32emu  -w -R 1791936000 -l 0 -d a.img monitor.bin < boot.in > a.out
HALT at 00100270, r1=00000000 (7490575 instructions)
$ ./p32uemu -u p32ucode.bin -w -R 1791936000 -l 0 -d b.img monitor.bin < boot.in > b.out
HALT at 00100270, r1=00000000 (7490575 instructions, 26738361 cycles)
$ cmp a.out b.out && cmp a.img b.img && echo identical
identical
```

(`boot.in` is `b`, `root`, `root`, `halt`, one per line.)

6. Disk images

6.1 Building one

```
make -C software disk        # -> software/build/disk.img (512 MB, D40), from software/root.manifest
```

This runs `tools/p32fs.py build IMAGE 512 root.manifest --kernel ../kernel/unix.bin --inodes 16384`. `p32fs.py` lays out the disk as follows:

Sectors	Contents
0	boot header: `"P32B"`, kernel sector count, load address, entry, first filesystem sector
1 … 2047	the kernel image (1 MiB boot area)
2048 …	the filesystem: 1 KiB blocks, superblock, inodes (64 bytes), free-block bitmap, data

`P32FS_TIME=SECS` fixes the timestamps for reproducible images (the session tests use it). `p32fs.py mkfs IMAGE SIZE_MB [--inodes N] [--kernel FILE]` makes an empty filesystem.

6.2 Looking at and changing an image

```
p32fs.py ls IMAGE [PATH]          p32fs.py cat IMAGE PATH
p32fs.py put IMAGE HOSTFILE PATH [--mode OCT] [--uid N] [--gid N]
p32fs.py mkdir IMAGE PATH [--mode OCT] [--uid N] [--gid N]
p32fs.py mknod IMAGE PATH c|b MAJOR MINOR [--mode OCT]
p32fs.py rm IMAGE PATH            p32fs.py ln IMAGE OLD NEW
p32fs.py chmod IMAGE OCT PATH     p32fs.py chown IMAGE UID GID PATH
p32fs.py fsck IMAGE
```

On a scratch copy (ken is uid 100, gid 10):

```
$ p32fs.py cat fs.img /etc/motd
Welcome to P32W Unix.
$ p32fs.py put fs.img note.txt /usr/ken/note.txt --uid 100 --gid 10
$ p32fs.py ls fs.img /usr/ken
drwxr-xr-x  2  100   10      192 .
drwxr-xr-x 10    0    0      320 ..
-rw-r--r--  1  100   10      235 hello.c
-rw-------  1  100   10      279 .sh_history
-rwxr-xr-x  1  100   10    37332 a.out
-rw-r--r--  1  100   10       20 note.txt
$ p32fs.py fsck fs.img
fs.img: 540 inodes, 8822 data blocks, clean
$ p32fs.py rm fs.img /usr/ken/note.txt
$ p32fs.py fsck fs.img
fs.img: 539 inodes, 8821 data blocks, clean
```

`fsck` checks the bitmap against the blocks in use, link counts, sizes, `.` and `..`, and that every block is owned once. Its "clean" means *no problems found*. It does not report the superblock's unmounted-cleanly flag; the kernel checks that flag at boot.

6.3 Updating files on a system you are running

There are two safe ways:

1. **Stop, edit, boot.** `halt` P32W Unix (so the image is written out and clean), run `p32fs.py put …` on the image, then boot it again. Do not edit the image while the emulator is still running.
2. **Copy the file in while the system runs**, over the second terminal with Kermit (`-T PORT`, then `kermit -x` on P32W and C-Kermit or `tools/pykermit.py --tcp PORT send FILE` on the Mac). The kernel writes the file, so the image stays consistent.

7. Bare-metal programs

Anything that is not P32W Unix runs as a **ROM image**: a flat little-endian binary placed at physical address 0 and started there in supervisor mode.

7.1 In assembly

```
python3 assembler/p32asm.py prog.asm -o prog.bin [-l prog.lst]
emulator/p32emu prog.bin
```

When the program starts: PC = 0, supervisor mode (untranslated addresses), interrupts off, `VBR` = 0, all registers 0, RAM all zeros (on the real machine RAM will hold garbage at power-on, so do not rely on this). A minimal program, `emulator/test/hello.asm`:

```
CON     = 0xFFFFFF00            ; UART 0 data register
        li    sp, 0x00500000    ; top of RAM; full-descending stack (D6)
        la    r10, msg
        call  puts
        li    r1, 0
        halt                    ; exit status = r1 & 0xFF
puts:   lbu   r11, 0(r10)
        beqz  r11, puts_done
        sb    r11, CON(r0)
        addi  r10, r10, 1
        jmp   puts
puts_done:
        ret
msg:    .asciz "P32W lives! instruction-level reference emulator\n"
```

In the emulators the UART needs no setup (there is no baud rate) and its transmitter is always ready. For portable code, set up the UART and poll the line-status register as the programmer's guide shows (`software/lib/p32io.inc` has the routines). The program's `HALT` sets the exit status, which makes self-checking tests simple: `isa_test.asm` halts with the number of the failing case.

7.2 In C

```
python3 cc/p32cc.py --sa prog.c -o prog.bin
emulator/p32emu prog.bin < /dev/null
```

`--sa` links `software/sa/crt0_sa.s`. The C runtime copies `.data` to RAM, clears `.bss`, sets the stack at the top of RAM, and installs a small trap handler that serves `exit`, `write`, `read` and `sbrk` through the real `SYSCALL` instruction. `main`'s result becomes the exit status:

```
$ cat sq.c
#include <stdio.h>
int main(void)
{
    int i;
    for (i = 1; i <= 5; i++)
        printf("%d squared is %d\n", i, i * i);
    return 42;
}
$ python3 cc/p32cc.py --sa sq.c -o sq.bin
$ ./p32emu sq.bin </dev/null; echo "status=$?"
1 squared is 1
2 squared is 4
3 squared is 9
4 squared is 16
5 squared is 25
HALT at 000000DC, r1=0000002A (19977 instructions)
status=42
$ ./p32uemu -u p32ucode.bin sq.bin </dev/null; echo "status=$?"
1 squared is 1
...
5 squared is 25
HALT at 000000DC, r1=0000002A (19977 instructions, 71312 cycles)
status=42
```

7.3 The memory map and the I/O page

The emulators follow the address map of D52 (architecture.md §4):

Physical address	What	Notes
`0000_0000` – `000F_FFFF`	ROM, 1 MiB (the image), on the memory card in window 0	a write is BUSERROR
`0010_0000` up to `N × 16 MiB`	RAM: `-M N` memory cards of 16 MiB in windows 0 … N−1 (1 to 4, default 1), or `-M S0,S1,…` MiB per window (D60)	
above it, to `FDFF_FFFF`	nothing	any access is BUSERROR (an empty memory window too)
`FE05_0000` – `FE05_FFFF`	nothing: the video card (revision 2, D103) has no memory on the bus	BUSERROR
`FFFF_8000` – `FFFF_FFFF`	the device windows: eight of 4 KiB, window n at `FFFF_8000 + n × 1000`, a card's registers in the window's top 256 bytes. The I/O card is in window 7, the disk card in window 6, the video card in window 5	only the registers below answer; anything else is BUSERROR

I/O address	Device	Access	IRQ
`FFFF_FF00` – `07`	UART 0 (TL16C550 registers 0–7): the console, stdin/stdout	bytes	2
`FFFF_FF08` – `0F`	UART 1: `-A` / `-a` files or `-T` TCP	bytes	3
`FFFF_FF30` – `37`	UART 2: `-A2` / `-a2` or `-T2`	bytes	2 (shared with UART 0)
`FFFF_FF38` – `3F`	UART 3: `-A3` / `-a3` or `-T3`	bytes	3 (shared with UART 1)
`FFFF_FF10` `TCOUNT`, `FF14` `TRELOAD`, `FF18` `TCTRL` (bit 0 enable, bit 1 pending; write 1 to bit 1 to acknowledge)	timer	aligned words only	0
`FFFF_FF20`	input port (reads `-p`; writes ignored)	byte	
`FFFF_FF21`	output port (reads back; `-t` traces writes)	byte	
`FFFF_FF40` / `FF41`	real-time clock: register number (write) / register (read/write)	bytes	(5, not generated)
`FFFF_FF50` / `FF51`	USB port 0 (CH376S): data / command and status (`-U`)	bytes	6
`FFFF_FF52` / `FF53`	the network coprocessor: data / status and control (`-N`)	bytes	7
`FFFF_FF54` / `FF55`	USB port 1, as `FF50` / `FF51` (`-U1`)	bytes	6
`FFFF_EF00 + 16 p`	the disk card's port p (0–3): +0–7 the task file (+0 data, +7 status / command), +8–15 the control block (+14 alternate status). Port 0 is CF socket 0 (`-d`), port 1 CF socket 1 (`-d1`), ports 2 and 3 the IDE channels A and B, master and slave (`-d2 -d2s`, `-d3 -d3s`; the DEV bit of +6 picks the drive). A **halfword** access to +0 is the 16-bit data register (D26). A port with nothing on it reads `00` in every register, as the card's pull-downs make it (D66)	bytes; halfword at +0	1 (every port)
`FFFF_DF00` / `DF04`	the video card's chip, an HD63484: its status and address register; the register addressed or the FIFOs (programmer's guide §6.6; only with the card fitted)	halfwords	4
`FFFF_DF08` / `DF0C`	the video card's status and PS/2 command; the PS/2 byte	bytes	4
`FFFF_xFF0` / `xFF1` / `xFF4` in windows 7, 6 and 5	card ID (1 I/O, 2 disk, 3 video), revision (bytes), interrupt status (word, a bit per device, 1 = asking)	as listed	

The rule is the card's: byte registers answer byte accesses only, the timer answers aligned word accesses only, and anything else gets no acknowledge, which the MMU / data card turns into BUSERROR.
`emulator/test/io_test.asm` exercises the I/O card's devices and CF socket 0 (`card_test.asm` the card registers and UARTs 2 and 3, `video_test.asm` the video card):

```
$ ./p32emu -p 0x5A -a aux.out -d io.img io_test.bin </dev/null; echo "status=$?"
io ok
HALT at 0000025C, r1=00000000 (10498 instructions)
status=0
$ cat aux.out
aux port
```

7.4 Going to the hardware

The same ROM image is what the memory card's boot flash holds: `python3 assembler/romsplit.py IMAGE.bin [PREFIX]` splits it into the four byte-lane files, one per flash chip.

8. Debugging

8.1 The instruction trace (`p32emu -t`)

```
$ ./p32emu -t hello.bin 2>&1 >/dev/null | head -8
00000000 S 2FF00050  LUI r15,0x50
00000004 S 2FA00000  LUI r10,0x0
00000008 S 23AA0030  ORI r10,r10,0x30
0000000C S 61000002  CALL 00000018
00000018 S 44BA0000  LBU r11,0(r10)
0000001C S 50B00003  BEQ r11,r0,0000002C
00000020 S 48B0FF00  SB r11,-256(r0)
00000024 S 20AA0001  ADDI r10,r10,1
```

Each line shows the address, the mode (S or U), the instruction word and the disassembly. Pseudo-instructions appear as what they expand to (`li` → `LUI`/`ORI`, `beqz` → `BEQ … r0`), and branch and jump targets are shown as absolute addresses. Events appear as indented lines:

```
  TRAP TLBMISS_X (8) addr=00008014 -> 00000168        (vm_demo)
  TLBW I[8] VPN=00008 ASID=1 -> PFN=00008 V-X          (vm_demo: valid, executable)
  TLBW D[0] VPN=00400 ASID=1 -> PFN=00200 VW-          (vm_demo: valid, writable)
  INTERRUPT IRQ0 at 000000F4 -> 000001F8               (irq_test)
  OUTPUT PORT <- 3C                                    (io_test)
```

The TRAP line gives the cause name and number, BADADDR, and the handler address (`VBR`). The INTERRUPT line gives the line, the `EPC` (the next instruction) and the handler. A trap line follows the instruction that faulted, except for a fault **on the fetch** itself: then there is no instruction line, only the TRAP line, because the instruction was never read. This happens for example after an `RTE` to a user page that is not in the TLB yet.

The trace can be large: P32W Unix runs about 9 million instructions to boot. Combine `-t` with `-l` or a small program, and filter it with `grep`.

8.2 State traces (`-s FILE`, both emulators)

One line per instruction, written **before** the instruction runs:

```
PC  MODE  r1 … r14  r15  o=OTHER-r15  st=STATUS  epc=EPC  cause=CAUSE  bad=BADADDR
00000004 S 00000000 … 00000000 00500000 o=00000000 st=1 epc=00000000 cause=00000000 bad=00000000
```

`r15` is the current mode's stack pointer and `o=` the other bank (the USP in supervisor mode, the SSP in user mode). The trace does **not** include r0, `VBR`, `IMASK`, `ASID`, `TLBHI`/`TLBLO`, the TLB, memory or devices. Differences in those show up a few instructions later, when they affect registers.

The two emulators write identical traces for any program they agree on, so `cmp` finds the first instruction at which they disagree. The first 300,000 instructions of a Unix boot (a 61 MB trace each):

```
$ printf 'b\n' | ./p32emu  -l 300000 -R 1791936000 -s r.trace -d a.img monitor.bin >/dev/null
$ printf 'b\n' | ./p32uemu -u p32ucode.bin -l 300000 -R 1791936000 -s u.trace -d b.img monitor.bin >/dev/null
$ cmp r.trace u.trace && echo "300000 states identical"
300000 states identical
```

And a deliberate disagreement: `-c` changes the timer, so the interrupt test diverges as soon as timing matters:

```
$ ./p32emu -s r.trace irq_test.bin; ./p32uemu -c -u p32ucode.bin -s u.trace irq_test.bin
$ cmp r.trace u.trace
r.trace u.trace differ: char 4087, line 21
```

Line 21 is the 21st instruction. `diff r.trace u.trace | head` shows both states there, and `p32emu -t -l 21` shows what led up to it.

8.3 The stop dump

When an emulator stops on an unhandled exception, the instruction limit or a hopeless `WAIT`, it prints the machine's state. The two dumps differ because the two machines differ:

```
$ ./p32emu mis.bin            (li r2, 0x00100002 ; lw r3, 0(r2))
EXCEPTION: misaligned load at address 00100002
  pc=00000008  ins=42320000  LW r3,0(r2)   (3 instructions)
  mode=S  status=1  epc=00000000  cause=00000000  badaddr=00000000  vbr=00000000  usp=00000000
  imask=00  timer: count=0 reload=0 enable=0 pending=0  asid=0 tlbhi=00000000 tlblo=00000000
  r0 =00000000  r1 =00000000  r2 =00100002  r3 =00000000
  ...
$ ./p32uemu -u p32ucode.bin mis.bin
EXCEPTION: cause 00000003 at address 00100002 (no handler, VBR = 0)
  pc=0000000C ir=42320000 uaddr=1:01:0:0 mode=S status=1  (3 instructions, 11 cycles)
  rf0 =00000000  rf1 =00000000  rf2 =00100002  rf3 =00000000
  ...
  rf28=00000000  rf29=00000000  rf30=00000000  rf31=00000000
```

- `p32emu` shows the **faulting instruction**: its address, word and disassembly, the control registers, the timer and MMU registers, and r0–r15. For a limit stop, `pc`/`ins` are the *last instruction executed*.
- `p32uemu` shows the hardware state: `pc` is the PC register, which has already advanced past the fetch (`0000000C` here, for the instruction at `00000008`); `ir` is the instruction register; `uaddr` is the microaddress `SEQ:OPCODE:STEP:COND` (here `1:01:0:0` = the TRAP internal sequence, step 0); and the dump lists all **32 physical register-file words**: rf15 = USP, rf16–19 = EPC/CAUSE/BADADDR/VBR, rf20–27 = the microcode scratch registers T0–T7, rf28 = the r0 write sink, rf31 = SSP. The cause is a number (3 = MISALIGNED, as in `p32isa.py`).

8.4 The instruction limit

The default limit of 100 M instructions stops a program that has gone into a loop. Raise it for long scripted runs: the session tests use `-l 4000000000`, and the native-compiler tests use up to `2000000000000`.

```
$ ./p32emu -l 1000 primes.bin >/dev/null
STOP: instruction limit (1000) reached
  pc=0000001C  ins=20220001  ADDI r2,r2,1   (1000 instructions)
  ...
$ ./p32uemu -u p32ucode.bin -l 1000 primes.bin >/dev/null
STOP: instruction limit reached
  pc=00000020 ir=20220001 uaddr=0:20:0:0 mode=S status=1  (1000 instructions, 3497 cycles)
```

Both emulators stop after exactly N instructions, at the same point.

8.5 Debugging programs on P32W itself

Inside P32W Unix, `db program [args]` is a machine-level debugger with breakpoints, single-stepping, `bt`, registers, memory and disassembly (unix.md, "Debugging"). For user programs this is usually more convenient than an emulator trace, which mixes in the kernel. To see what the kernel does, use `p32emu -t` with a limit, or look up addresses in `kernel/unix.map`.

9. Tests

`make -C emulator test` (`emulator/test/run_tests.sh`) covers the whole project. The parts that involve the emulators are:

- **Programs on p32emu** (`emulator/test/*.asm`): exact encodings, `hello`, `isa_test` (every instruction), `echo`, `fib` and `primes` (compared with Python references), `trap_test`, the stage-A kernel programs, `irq_test`, `wait_nowake`, `mt_demo`, `vm_test`, `vm_demo`, and the `exc_*` programs (each must stop with status 255 and the right `EXCEPTION:` message).
- **The devices on both emulators**, instruction by instruction: `io_test` (UARTs, ports, the CF disk; identical disk images afterwards), `card_test` (the card ID registers, UARTs 2 and 3, the shared interrupt levels) and `video_test` (the video card: its chip's registers and FIFOs, the font, characters, a line, interrupts, the keyboard and the mouse; identical `-V` screen dumps). `acrtc_test.c` checks the chip's model by itself. `vidcard_test` is the video card's short program for the gate-level machine: `generators/sim_sys.py --video` against `p32uemu -C`, the same trace and cycles, both interrupts taken (D107).
- **The bus lock** (D54): `swap_dma` runs `SWAP` on `p32uemu` against the DMA master (`-D`) at several periods and must lose no addition; with `-Dnolock` it must lose some, so the test can tell.
- **The cross-check** (§14.1): 20 programs run on both emulators. Exit status, console output and the `-s` trace must be identical. `lib_test` and the scripted monitor session also compare disk images.
- **P32W Unix sessions** (§10.2), on p32emu, with session1 also on p32uemu.

10. Scripted sessions

10.1 `-w` and prompts

An interrupt-driven program such as P32W Unix reads input when it is ready. `-w` makes piped input arrive the way a person would type it, after the program has asked for it:

- Input is delivered **one line at a time**. A line is released only when the output **ends with a prompt**. After a line is released, the output must show a prompt again before the next one.
- A prompt counts only **at the start of a line** (just after a `\n`), unless it is written with a leading `*`, which allows it anywhere.

- The defaults are `# `, `$ `, `> `, `login: `, `Password: `, `Ok\r\n` (BASIC) and `*? ` (BASIC's `INPUT`, anywhere in a line).
- `-P STR` adds a prompt; the first `-P` **replaces** the defaults (up to 8 in all). Example from `db_test.sh`: `-w -P '# ' -P 'db> ' -P 'login: ' -P 'Password: ' -P '> '`.
- `-w` applies to UART 0 and, with `-A`, to UART 1, each watching its own output.
- `-w` turns off interactive mode (no raw terminal). The default instruction limit of 100 M therefore applies: pass `-l`.
- When the input runs out, nothing more is typed. End every script with `halt`. Otherwise the system keeps idling (skipping time) until the instruction limit stops it with status 255.

10.2 The session tests

`software/tests/run_session.sh NAME [uemu]` builds a fresh image with fixed timestamps in `$TMPDIR`, assembles the monitor, and then runs:

```
p32emu -w -R 1791936000 -l 4000000000 -d $T/os.img [-A NAME.tty1 -a tty1.out] monitor.bin < NAME.in
```

It removes the lines that depend on binary sizes or speed (kernel sector count, free memory, seconds in `date`/`uptime`/`time`), prints the transcript (and tty1's, if there is one), and runs `p32fs.py fsck` on the image the session leaves behind. `make test` compares the output with `NAME.expect`. `session1` also runs with `uemu`, on `p32uemu` and the real control store. Other tests use the same pattern: `toolchain_test.sh`, `cc_test.sh`, the tar and zip tests, `db_test.sh` (with `-P`), and `kermit_test.py` / `vi_test.py` (which use the TCP terminal, `-T`).

Part 2 — Theory of operation

11. The shared devices (`p32io.h`)

Both emulators `#include "p32io.h"`. It holds the only copy of every device (D50): the I/O card, the disk card and the video card, so the two cannot disagree about devices. The CPU side calls six functions:

Function	Use
`io_read(pa, size, &v)` / `io_write(pa, size, v)`	an access to a device window, the value right-aligned. They return 1 if nothing answers (no `-ACK`), which the caller turns into BUSERROR.
`io_ipend()`	the IRQ lines now asserted, as `IPEND` shows them
`io_tick()`	one timer tick (per instruction, or per microcycle with `-c`)
`io_wait(imask)`	`WAIT` with nothing pending: idle until something can wake the CPU, or report that nothing can
`io_hwedge()`	`p32uemu -C`: the I/O card's clock edge

The video card's chip is a model of its own, `p32acrtc.h` (from YACC1-D's, moved to the chip's 16-bit bus), which `p32io.h` includes: the registers, the two FIFOs, the commands a driver needs, the chip's busy time. Its time is the machine's: the timer's microseconds, or the microcycles with `p32uemu -C`, where each access to the chip takes three wait states and a PS/2 command written to the card takes one (D106). With `-C` the CPU's interrupt logic has the chip's IRQ as the microcycle began: a request that a register write causes is seen in the next microcycle, as on the card, where the write lands in the middle of the cycle's last clock, the moment the CPU card samples the lines (D107). The PS/2 section's firmware has the same time base: its delays are microseconds, counted in the timer's (an instruction each) or, with `-C`, in microcycles at 3 or 4 MHz, and at gate level the same model decides in which clock the stand-in moves the ATmega's pins, so the card's own FULL and BUSY flip-flops and `p32uemu -C` agree clock for clock (D109). `io_wait` knows that the chip finishes its work by itself and may then interrupt, and that the firmware loads its next byte by itself, as it knows the timer. The option parsing for the device options (`io_option`: `-d -d1 -d2 -d2s -d3 -d3s -a -A -T` and their UART 2 and 3 forms, `-p -w -P -R -U -U1 -Uout -N -Nport -Nca -V -K -Km`) and the terminal setup (`io_init`) are shared as well.

Devices respond instantly. There is no baud-rate delay, no disk seek, and no BSY phase. Software must depend only on status bits, as on the hardware. Concretely:

- **UARTs 0–3** (TL16C550 subset): RBR/THR, IER, IIR/FCR, LCR (DLAB), MCR, LSR, MSR, SCR, DLL/DLM. The divisor and FCR are stored and otherwise ignored. A byte written to THR is sent at once, so LSR always shows THRE and TEMT. LSR.DR shows whether input is available, read ahead one byte. MSR reads `B0` (DCD, DSR, CTS). Interrupts: receive-data-available while IER bit 0 is set and input is waiting; THRE after a byte is sent or when IER bit 1 is newly enabled, cleared by reading IIR (16550 rules). There is no FIFO model and no line errors.
- **Timer:** `TCOUNT`, `TRELOAD`, `TCTRL` (§12.5, §13.6).
- **The disk card's drives** (True-IDE task file, four ports with a master and a slave each; the DEV bit selects the drive, and register writes 1–6 go to both drives as on a shared cable): READ SECTOR(S) 20h/21h and WRITE SECTOR(S) 30h/31h (LBA mode only, count 0 = 256), IDENTIFY ECh (model "P32W EMULATED CF CARD" on the CF sockets, "P32W EMULATED IDE DISK" on the IDE channels), SET FEATURES EFh (01h = 8-bit, 81h = 16-bit transfers), the power/cache/parameter no-ops (E0 E1 E5 E7 91). Anything else aborts with ABRT. Data moves through a 512-byte buffer: byte reads and writes at the port's +0, or halfwords there (D26). INTRQ is set when a sector is ready or written and cleared by reading Status; every port's INTRQ drives IRQ1, and the disk card's interrupt status shows which. Reading past the end of an image gives IDNF. A drive that is not fitted reads Status `00`.

- **Ports:** `-p` value in, latch out.
- **RTC:** MC146818 registers 0, 2, 4, 6–9, `0x32` (century), A–D, and RAM `0x0E`–`0x7F`. Binary or BCD and 12/24-hour follow register B. SET (B bit 7) freezes the time for setting and commits it when cleared. UIP is never set and register C reads 0 (no alarm or periodic interrupts).
- **USB** (two CH376S in sector mode, D43; usb-design.md): a drive whose contents are an image file (`-U`, `-U1`), or no drive.
- **The network coprocessor** (D45): the two byte registers and the request / reply protocol of net-mailbox.md, carried out with the Mac's own sockets; TLS goes through `tools/p32tls.py`.
- **The video card** (only with `-V`, `-K`, `-Km` or `-dbg`), revision 2 (D103, D104): the HD63484 through its two addresses, and the PS/2 section as its ATmega328P's firmware shows it (ASCII from the keyboard, the mouse's packets whole, the nine commands), **with the firmware's time** (D109; docs/video-card-acrtc.md §4.6): two seconds from a reset (`-Kr`: none), FULL clear some 50 µs after each byte read, BUSY some 30 µs after a command and through a command that talks to a device, a command written over one not yet taken lost, the flush leaving the latch as it is. No memory on the bus. It draws no picture; `-V` writes the screen at exit. What the chip's model lacks (circles, arcs, PAINT, character screens, the cursors) is listed at the head of `p32acrtc.h`: such a command sets the chip's error bit and is named on stderr.
- **Card registers:** each card answers at `+FF0` of its window its card ID, at `+FF1` its revision (the I/O card 0, the disk card 1, the video card 2) and at `+FF4` its interrupt status.
- **IRQ lines asserted:** 0 timer, 1 disk, 2 UART 0 or 2, 3 UART 1 or 3, 4 the video card (its PS/2 section, its chip), 6 USB, 7 network. IRQ5 (RTC) is defined but never asserted.

12. `p32emu`: the instruction-level reference

12.1 Machine state

```
uint32_t r[16];          // r15 = the CURRENT mode's SP
uint32_t sp_other;       // the inactive r15 bank (USP in S mode, SSP in U mode)
uint32_t pc, status;     // status: S=1 IE=2 PS=4 PIE=8; reset = S
uint32_t epc, cause, badaddr, vbr, imask, asid, tlbhi, tlblo;
struct { uint32_t hi, lo; } tlb[2][64];   // [0] D-TLB, [1] I-TLB
uint8_t *rom /* 1 MiB */, *ram /* from 1 MiB to -M N x 16 MiB */;
```

The r15 bank (D6) is implemented by **swapping values**: `set_mode()` exchanges `r[15]` and `sp_other` whenever S changes. `MFCR USP` (which is privileged, so it is always used in S mode) reads `sp_other`.

12.2 The loop

`run()` repeats the following for every instruction:

1. **Limit:** if the count has reached `-l`, stop.
2. **Interrupt check** (between instructions, B2): if `IE` and `io_ipend() & IMASK`, enter the interrupt (§12.4). After an `MTCR` or an `RTE`, the check uses the `IE` and `IMASK` from *before* that instruction (the interrupt shadow, D20).
3. **State trace** (`-s`), then count the instruction and `io_tick()`.
4. **Fetch** through `load_acc(pc, 4, ACC_FETCH)`, which translates the address and checks the bus.
5. **Decode** with the tables in `p32opc.h`: `p32_fmt[op] == F_ILLEGAL` or any set must-be-zero bit (`p32_zmask[op]`) → ILLEGAL; `p32_priv[op]` in user mode → PRIVILEGE. Field extraction: A = bits 23:20, B = 19:16, C = 15:12, 16-bit immediate (sign- or zero-extended), 24-bit jump offset.

6. **Execute** in a `switch` on the opcode. The C code is the definition of each instruction, for example DIV by zero gives `0xFFFFFFFF`, and branches compare A with B (D5).
7. **Commit:** `pc = next`, `r[0] = 0`.

`p32opc.h` is generated from `p32isa.py`, so the opcode numbers, formats and the illegal/privileged decisions are the same ones the assembler and the microcode generator use.

12.3 Exceptions: precise by construction

`trap(code, addr, what)` is called from any check. It sets `EPC = cur_pc` (the faulting instruction), `CAUSE`, and `BADADDR = addr`. Then it updates `STATUS` (PS ← S, PIE ← IE, S ← 1 with the r15 swap, IE ← 0), sets `pc = VBR & ~3`, and `longjmp`s back to the top of the loop, abandoning the instruction. This is precise because **every check happens before any state is written**. A load writes `rd` only after the access succeeds. `CALL` stores the return address and then updates `SP`. `RET` loads the return address, checks its alignment, and only then pops. So `RTE` can simply run the instruction again (D10).

The check order for an access is (C5): alignment → user access to the upper 2 GiB (PRIVILEGE) → TLB miss → protection → physical bus (ROM write, unmapped address, an I/O register that does not answer or answers another size). `BADADDR` is the virtual address for the first four and the **physical** address for a bus error. Traps raised by the control logic (ILLEGAL, PRIVILEGE for an opcode, SYSCALL, BREAK) have `BADADDR` = 0. **No memory is written by trap entry** (D10).

While `VBR` = 0 no handler is installed, so an exception or interrupt stops the emulator with the dump (§8.3) and status 255. This is how bare test programs report a fault.

12.4 Privilege and interrupts

User mode can run neither the privileged opcodes (`HALT`, `RTE`, `MFCR`, `MTCR`, `WAIT` and `TLBW`, the set `PRIVILEGED` in `p32isa.py`) nor access `0x8000_0000` and above. `RTE` returns to `EPC & ~3` and restores S ← PS, IE ← PIE.

`interrupt()` picks the **lowest** asserted, enabled line (fixed priority) and enters the trap with `CAUSE = 0x8000_0000 | v << 8 | n` (*v* the vector `io_iack(n)` gives: the acknowledge cycle, D57), `BADADDR = 0` and `EPC` = the next, unexecuted instruction. Entering clears `IE`, so the next check cannot interrupt the handler's first instruction. The **interrupt shadow** (D20) copies the hardware: the sequencer decides "interrupt" in an instruction's last microcycle, while that cycle's own write to `STATUS` or `IMASK` lands only at its closing edge. `p32emu` therefore saves `IE` and `IMASK` before each instruction (`sh_ie`, `sh_im`) and uses them for the check after an `MTCR` or an `RTE`. As a result one more instruction always runs after `RTE` before a pending interrupt is taken, which guarantees progress.

12.5 The timer: one tick per instruction (D11)

`io_tick()` runs once per instruction, before the fetch: if enabled, `TCOUNT` counts down; at 0 it sets pending (IRQ0) and reloads from `TRELOAD`. The hardware timer runs at 1 MHz from the clock divider, so the real interrupt point depends on the exact cycle counts of the instructions. Tying the tick to the instruction count instead makes **interrupt timing exactly repeatable and independent of the microarchitecture**. A program sees the same interrupt at the same instruction every run. That is what lets `irq_test` check "the interrupt arrives at exactly this instruction". It also means the instruction-level model never needs cycle counts, which keeps it the simple, golden reference (D11 point 4). The cost is that `p32emu`'s timing is not the hardware's (§15). The RTC's emulated time (`rtc_us`) advances with the same ticks.

12.6 `WAIT` and idle skipping

`WAIT` (privileged) calls `io_wait(IMASK)`:

- If an enabled line is already up, it returns at once, and the interrupt is taken at the next check if `IE` is set.
- If the timer is off or IRQ0 is masked, **nothing can ever wake the CPU**, because the timer is the only device that changes by itself. The emulator stops with `STOP: WAIT with no possible wake-up` and status 255 (`wait_nowake.asm` tests this). This also applies when only a UART interrupt is enabled: the model never considers input that might arrive later.
- Otherwise, if interactive (a terminal console or `-T`), it `select()`s for up to `TCOUNT` microseconds. If input arrives first, the timer advances by the time that actually passed and the UART is polled at once. Otherwise the timer expires.
- Otherwise it **jumps to the expiry**: the remaining `TCOUNT` is added to "idle ticks" and to the RTC's emulated time, pending is set, and `TCOUNT` reloads. That costs no host time.

Either way `WAIT` itself is one instruction. The skipped ticks are not instructions.

12.7 The TLB (stage C, D12/D13)

There are two direct-mapped tables of 64 entries: the **I-TLB** for fetches and the **D-TLB** for loads and stores. Each entry is `hi = VPN<<12 | ASID` and `lo = PFN<<12 | X W V`. Translation applies to **user mode only**; supervisor mode is untranslated. For a user access to address `va` below 2 GiB:

```
index = (va >> 12) & 63                 side = fetch ? I : D
hit   = lo.V && hi.VPN == va.VPN && hi.ASID == ASID
miss  -> TLBMISS_X / _R / _W, TLBHI = va.VPN | I-bit(0x100) | ASID
store to W=0 or fetch from X=0 -> PROTECT, TLBHI set the same way
pa    = lo.PFN | (va & 0xFFF)
```

Refill is done by software: the kernel's miss handler finds the faulting page in `TLBHI`, writes `TLBLO`, and runs `TLBW`, which writes entry `[TLBHI.I][TLBHI.VPN & 63]`. Then `RTE` retries the access. There is no hardware page-table walker and no global bit. ASIDs are 8 bits, so the kernel flushes by writing invalid entries. The writable bits are `TLBHI & 0xFFFFF1FF` and `TLBLO & 0xFFFFF007`.

12.8 Memory and the I/O page

`load_acc` and `store` check alignment, translate, and then either call `io_read`/`io_write` (physical ≥ `0xFFFF_8000`) or `map()` the address into the ROM or the RAM of the fitted memory cards (`-M`), little-endian, `size` bytes at a time. A ROM write or an address in none of them is BUSERROR.

13. `p32uemu`: the microcode-level model

13.1 What it models

`p32uemu` is the **three-bus datapath** of microarch.md, clocked one microcycle at a time. Bus **A** carries the first operand (register-file read port A, or the PC), bus **B** the second (read port B, or the immediate unit), and bus **R** the result, which goes to one destination a cycle. It models the two cards that make up the processor, the **CPU card** and the **MMU / data card** (architecture.md §3), joined by the local link on J3 (bus.md), and the rest of the machine as the bus sees it:

Card, part	Model state
CPU card: sequencer, control store, instruction register and decode	microaddress `{useq, uop, ustep, ucond}`, `IR`, `STATUS`, `IMASK`, the cause latch, the fault flags (FFAULT: the fault was in the fetch; MMUF: the trap came from the MMU / data card), the decode trap, the CR decode table (`uc_crkind`, `uc_craddr` from `p32uc.h`), the immediate unit

Card, part	Model state
CPU card: register file and PC	`rf[32]`, the 32-word register SRAM, with two read ports (A, B) and one write port (from R); `pc` (bits 31:2: loads mask the low bits, `PCINC` adds 4)
CPU card: MDR	`mdr`, the store data, loaded from B (`MDRB`) or from R, and put on R replicated (`MDRS`)
CPU card: the ALU module	latches `alua`, `alub` and the function `aluop`, loaded together by `ALUGO`; the result on R (`RSRC = ALU`) from the next cycle until the next `ALUGO`; compare outputs `EQ`, `LT`, `LTU` of the latched operands
MMU / data card: address side	`MAR`, `ASID`, `TLBHI`, `TLBLO`, the split TLB, the access checks, the BADADDR latch (`mmu_badaddr`, updated by MMU faults only, D17)
MMU / data card: data side	load steering: in a `LOAD` cycle the addressed byte or halfword, brought down to bit 0 and extended, onto R
MMU / data card: the bus master interface	wait states (`-W`), the bus lock (`-LOCK`, held from a locked `LOADW` to its `STOREW`, D54) and, with `-D`, the arbiter and a DMA master
memory cards	ROM and RAM arrays (`-M`)
I/O, disk and video cards	`p32io.h`

13.2 The control store and the microword

`p32ucode.bin` holds 16,384 words, one per microaddress:

```
microaddress = SEQ (bit 13) | OPCODE (12:5) | STEP (4:1) | COND (bit 0)
```

`SEQ` = 0 runs the instruction in `IR` (`OPCODE` = `IR[31:24]`). `SEQ` = 1 runs an internal sequence: 0 RESET, 1 TRAP, 2 INTR. `STEP` counts cycles within a sequence (at most 16). Every step is a **pair** of words: `COND` picks the twin, using the condition named by the *previous* word and evaluated on the state *after* that word's clock edge (the P8X scheme). The 56-bit microword (`p32uc.h`'s `UF_*` macros; seven flash chips, `u0.bin`– `u6.bin`) has these fields, from bit 0 up:

Field	Bits	Values
`ASRC`	2	bus A's source: NONE, REG (read port A), PC
`BSRC`	2	bus B's source: NONE, REG (read port B), IMM (the immediate unit)
`RSRC`	4	bus R's source: NONE, ALU, MEM (a read cycle's data, steered for `LOAD`), IMM, MAR, MDRS (MDR replicated, the store data), CAUSE (the cause latch), BADADDR (the MMU / data card's latch), CRHW (a hardware CR), AGU (the address adder: A + the `AOFF` offset, D59)
`RDST`	3	R's destination: NONE, REG (the write port), PC, MAR, PCMAR (PC and MAR together), IR, MDR, CRHW
`MDRB`	1	MDR ← B at the edge
`MARPC`	1	MAR ← PC at the edge (the next fetch's address)
`PCINC`	1	PC += 4 at the edge
`ALUGO`	1	latch A, B and `ALU` into the ALU module at the edge
`ALU`	5	ADD SUB AND OR XOR NOR SLL SRL SRA SLT SLTU MUL MULH MULHU DIV DIVU REM REMU PASSA
`RSA`, `RSB`, `RSW`	3 + 3 + 3	the register address of port A, port B and the write port: IR field A, B or C (with the r15 bank rule), CONST, CRMAP (the CR table), SP (15 with the bank rule)
`RCONST`	5	the physical register address for any port that names CONST (r0 0, EPC 16, CAUSE 17, BADADDR 18, VBR 19, T0–T7 20–27)
`IMM`	3	SEXT16, ZEXT16, LUI, BR (sext16·4), J (sext24·4), FOUR, ZERO, INC8 (IR 7:0 sign-extended, D59); on B or on R
`MEM`	3	NONE, FETCH, LOAD, STORE (size from IR), LOADW, STOREW (word), CHECKW (alignment check only), IACK (the interrupt acknowledge cycle, D57)
`LOCK`	1	on a `LOADW`: LOCKREQ on the local link, so `-LOCK` is held from this read to the end of the following `STOREW` (D54; `SWAP`, D51). The generator rejects it on any other word, and a locked read without its `STOREW`
`SEQ`	2	NEXT (step + 1), END (instruction done: interrupt or fetch), TRAP (raise `RAISE`)
`COND`	3	ZERO, EQ, LT, LTU (of the ALU's latched operands), CRILL (reserved CR), CRHW (hardware CR), FFAULT, MMUF

Field	Bits	Values
`MODE`	3	NONE, TRAPENTER, RTE, TLBW, HALT, WAIT
`RAISE`	4	the cause for `SEQ = TRAP` (SYSCALL, BREAK, ILLEGAL)
`AOFF`	2	the address adder's offset: SEXT16 (IR 15:0), OFF8 (IR 15:8), INDEX (B << IR 1:0) (D59)
`MARINC`	1	MAR += 4 at the edge (`LI32`, D59)

Step 0 of every opcode's block is the shared fetch. Every instruction starts with MAR = PC, and its last word sets MAR to the next PC (`MARPC`, or `RDST = PCMAR`), so the fetch needs no address cycle of its own. For example `ADD rd, rs1, rs2`:

Step	Microword
0	`RSRC=MEM MEM=FETCH RDST=IR PCINC` (MAR already holds the PC; the opcode now selects the block)
1	`ASRC=REG(RSA=B) BSRC=REG(RSB=C) ALUGO ALU=ADD`
2	`RSRC=ALU RDST=REG(RSW=A) MARPC SEQ=END`

The ALU is pipelined: step 1 latches both operands and the function, and step 2 puts the result on R while the PC goes to MAR for the next fetch.

Cycle counts, including the fetch (`p32ucode.py --stats`): ALU and immediate operations 3, `LUI` 2, `LI32` 3, loads and stores 3 (indexed stores and post-increment 4; the address adder, D59), `SWAP` 5, branches 3 (not taken) or 4 (taken), `JMP` 3, `JR` 4, `CALL` 7, `CALLR` 10, `RET` 6, `RTE` 3, `MFCR`/`MTCR` 3–5, `HALT`/`SYSCALL`/`BREAK`/`WAIT`/`TLBW` 2. An illegal or privileged-in-user-mode opcode costs one suppressed cycle after the fetch (§13.3, step 4). Trap entry takes 6 cycles and interrupt entry 6. Measured: P32W Unix booted, root logged in, `halt` (7,049,876 instructions) takes 22,427,463 cycles, **3.18 per instruction**: 0.94 MIPS at 3 MHz. For comparison, before the address adder and the D59 instructions the same run took 26,738,361 cycles (7,490,575 instructions, 3.57 per instruction), and a one-bus datapath 47,030,356 (6.28).

13.3 One microcycle (`cycle_body`)

1. **Instruction boundary** (`SEQ` = 0, step 0): check the limit, write the state trace, count the instruction, and `io_tick()` (default timing).
2. Record `IE` and `IMASK` **as the cycle begins**, for the interrupt decision.
3. Read the microword at the microaddress.
4. **Decode trap** (D18): in step 1, the first after the fetch, the CPU card's decode table flags an illegal opcode (format, must-be-zero bits) or a privileged one in user mode. The cycle does nothing and the TRAP sequence follows.
5. **Buses A and B**, from the state as the cycle began. `ASRC` puts a register-file word (address from `RSA`) or the PC on A; `BSRC` puts a register-file word (address from `RSB`) or the immediate unit on B.
6. **Bus R: exactly one source.** `RSRC` selects one value: the ALU's result from the operands and function an earlier `ALUGO` latched, a **memory read cycle** (a fetch or `LOADW` gives the whole word; a `LOAD` gives the addressed byte or halfword brought down and extended), the immediate unit, MAR, the MDR replicated (`MDRS`: a byte or halfword on all four lanes, by the size from `IR`), the cause latch, the BADADDR latch, or a hardware control register. If `MEM` names a cycle but `RSRC` is not MEM (a store, or a `CHECKW`), the memory cycle runs with R as the store data.
7. **A fault aborts the cycle** (§13.5).
8. **The clock edge.** `RDST` latches R into one destination: a register-file word (address from `RSW`), `PC` (low two bits dropped), `MAR`, `PC` and `MAR` together, `IR`, `MDR`, or a hardware CR. At the same edge `MDRB` loads MDR from B, `MARPC` loads MAR from the PC, `ALUGO` latches A, B and the `ALU` function, and `PCINC` adds 4 to the PC. Then the

`MODE` action: TRAPENTER (PS ← S, PIE ← IE, S ← 1, IE ← 0), RTE (S ← PS, IE ← PIE), TLBW (write the entry `TLBHI` selects), HALT (print and exit), WAIT (under `-C`, with nothing pending, the microcycle is held and runs again).

9. **Next microaddress.** NEXT: step + 1, `ucond` = the condition on the post-edge state, so EQ, LT and LTU compare the operands this word latched; leaving step 0, `uop` is loaded from `IR[31:24]`. END: if the start-of-cycle `IE` is set and a line in the start-of-cycle `IMASK` is pending — **and this is not a TRAPENTER cycle** (D20) — go to INTR with the cause `0x8000_0000 | n`. Otherwise go to step 0, the next fetch, with MAR already at its address. TRAP: go to the TRAP sequence with `RAISE` as the cause.

This model has the same restrictions as the hardware's datapath. Each bus has one driver a cycle; R has one destination, beside which only MDR (from B), MAR (from the PC), the ALU's latches and the PC's increment change at the edge; the register file reads two words and writes one. Adding two registers therefore takes two cycles after the fetch: both operands into the ALU, then the result back to the register file. The generator checks every word: R with a destination has a source, a read's data is on R, a store's data is `MDRS`, MAR and MDR are not loaded twice, the PC is not both loaded and incremented, and the immediate unit is not on B and R at once.

13.4 Registers, the r15 bank and r0

The register file is 32 physical words:

Physical	Contents
0	r0 (written 0 once, by the RESET sequence)
1–14	r1–r14
15	r15 in user mode = **USP**
16–19	EPC, CAUSE, BADADDR, VBR
20–27	T0–T7, microcode scratch
28	**write sink**
31	r15 in supervisor mode = **SSP**

The three ports each form their own address (`RSA`, `RSB`, `RSW`) and share the word's one `RCONST`. The **bank rule** is applied to the address: an IR-field or `SP` reference to register 15 means physical 31 when S = 1. No value is copied on a mode change (compare `p32emu`'s swap). **r0** has no special storage. When a write's address comes from an IR field equal to 0, the CPU card redirects it to physical 28, so physical 0 keeps the 0 that RESET wrote. `MFCR`/`MTCR` reach EPC, CAUSE, BADADDR, VBR and USP through the CR decode table as ordinary SRAM words (`RSA` or `RSW` = CRMAP). `STATUS`, `IPEND`, `IMASK`, `ASID`, `TLBHI` and `TLBLO` are hardware registers (`RSRC` / `RDST` = CRHW). A reserved CR number makes the `CRILL` condition true, and the microcode raises ILLEGAL.

13.5 Memory cycles, byte lanes and faults

`mem_cycle()` models the MMU / data card and the memory cards. It checks alignment (`MAR` against the size from `IR`, or 4 for the W kinds), then translation (the same TLB rules as §12.7, with `TLBHI` set on a miss or protection fault), then the physical bus. Reads return the **whole aligned word** with all four lanes driven, as a memory card does; for a `LOAD` the MMU / data card's steering brings the addressed byte or halfword down and extends it onto R in the same cycle. Stores write only the lanes the **byte enables** select, from the replicated data on R (`MDRS`). For the I/O page the model shifts the value to and from the lane, as the MMU / data card's steering does.

A fault (`mem_cycle` returns 1):

- **Suppresses the whole cycle**: no R destination, no `MDRB`, `MARPC`, `ALUGO` or `PCINC`, no `MODE`. This is the CPU card's abort gating (D17).
- Latches the cause and `mmu_badaddr` = the faulting address (virtual for the MMU checks, physical for BUSERROR, as the hardware's PA lines give, D50), sets MMUF, and sets FFAULT if the fault is in the fetch cycle (step 0).
- Jumps to the **TRAP sequence**, which is ordinary microcode:

Step	Microword
0	`CAUSE ← cause latch`, condition FFAULT
1	`A = PC`, `B = 4` (fault after the fetch: PC already incremented) or `B = 0` (fault in the fetch), `ALU ← SUB`, condition MMUF
2	`BADADDR ← 0` (raised by the CPU card) or `← BADADDR latch` (MMU fault)
3	`EPC ← ALU`
4	`A = VBR`, `ALU ← PASSA`
5	`PC, MAR ← ALU`, `MODE=TRAPENTER`, END

INTR is similar, in 6 cycles: `MAR ← cause latch` (`0x8000_0000 | n`) with `A = PC`, `ALU ← PASSA`; `EPC ← ALU` (the next instruction); the acknowledge cycle, `CAUSE ← MEM` (`MEM = IACK`: MAR with bits 15:8 = the vector, D57); `BADADDR ← 0`; `A = VBR`, `ALU ← PASSA`; then `PC, MAR ← ALU` with TRAPENTER. Both end with MAR at the handler, ready for its fetch. Entering either sequence with `VBR` = 0 stops the emulator (as `p32emu` does).

The generator's **commit-after-access** check (D14) makes this precise. On every path through every instruction, no programmer-visible state (an architectural register, PC, a store, STATUS, the TLB) is written before the instruction's last cycle that can fault. Earlier results go to T0–T7 or wait in the ALU. For example `CALL` computes `SP - 4` in the ALU, stores the return address there, and the ALU still holds `SP - 4` when the next step writes it to SP; `SWAP` keeps the word it read in T0 until its store has succeeded. A faulting instruction therefore leaves nothing behind, and `RTE` re-runs it, exactly as in `p32emu`. The generator also checks that step 1 never starts a memory cycle, because a decode trap must be able to suppress that step (D18), and that every END leaves MAR at the next PC.

Wait states, the lock and DMA (`cycle()`). A microcycle whose `MEM` field names a bus cycle (a fetch, load or store) is held for `-W N` more clocks, as the MMU / data card holds the CPU card through `STALL` while a slave asserts `-WAIT` (D50). Only the cycle count and, with `-c` or `-C`, the timer see the difference. A locked `LOADW` (the `LOCK` bit) sets the bus lock, and its `STOREW`, or a fault, clears it (D54). With `-D`, the DMA master asks for the bus every PERIOD microcycles; the arbiter grants it at the next microcycle boundary unless the lock is held (`-Dnolock`: in any case). Its read and write of the word come first, and a CPU bus cycle in that microcycle waits two clocks for them. This is how `swap_dma` shows that `SWAP` loses nothing to a DMA master under `-LOCK` (§9).

13.6 Timer timing: default, `-c` and `-C`

Mode	Tick	Matches	Use
default	once per instruction, at the instruction boundary (not in trap or interrupt entry), like `p32emu`	`p32emu`, instruction by instruction	the cross-check; running software
`-c`	once per microcycle, including trap and interrupt entry	nothing exactly: a quick approximation of hardware timing	experiments

Mode	Tick	Matches	Use
`-C`	`io_hwedge()` at the end of every microcycle, with the I/O card's logic: a tick every **third** cycle while enabled (1 MHz from the 3 MHz microcycle), the enable taking effect one edge late, a `TCOUNT` write in that cycle winning over the count, and at 0 (enabled) the next edge reloading and setting pending, so `TCOUNT` reads 0 for one cycle. `WAIT` with nothing pending **holds its microcycle** (the sequencer's hold) instead of skipping	the I/O card's timer logic, cycle for cycle	checking hardware timing

With `-c` or `-C`, interrupts arrive at different instructions than in `p32emu`, so self-checking programs that test exact interrupt timing fail on purpose. `irq_test` halts with r1 = 3 under `-c` and r1 = 8 under `-C`:

```
$ ./p32emu irq_test.bin                    HALT at 000001F0, r1=00000000 (169 instructions, 98 idle ticks)
$ ./p32uemu irq_test.bin                   HALT at 000001F0, r1=00000000 (169 instructions, 582 cycles)
$ ./p32uemu -c irq_test.bin                HALT at 000001F4, r1=00000003 (23 instructions, 77 cycles)
$ ./p32uemu -C irq_test.bin                HALT at 000001F4, r1=00000008 (92 instructions, 319 cycles)
```

(`-u p32ucode.bin` omitted.) Programs that depend only on correct behaviour, not exact timing, pass in every mode: `mt_demo` exits 0 under all three, after 8,215, 104,978 and 9,553 instructions.

13.7 What `p32uemu` deliberately does not know

- **Instruction semantics.** There is no `ADD`, branch or `CALL` in the C code. It only knows how to move a value from a source to a destination, how the ALU computes each function, the immediate formats, and the memory card rules. What `ADD` means is defined by the sequence of microwords. A wrong microword gives a wrong result, which the cross-check catches.
- What it does take from the ISA tables is **what the hardware also has in tables**: the decode trap uses `p32_fmt`, `p32_zmask` and `p32_priv` (the CPU card's decode flash, `dec_op.bin`, holds the same data); load and store size and signedness come from the opcode (the same flash's SIZE/SIGNED bits); the CR table comes from `p32uc.h`. It does not read the `dec_*.bin` images themselves.
- The ALU is behavioural: its result is on R from the cycle after `ALUGO`, as the module's pipeline gives it. The ALU module's FPGA is verified against the same functions separately (§14.3).

14. Keeping the two honest

14.1 The state-trace cross-check

For every one of 20 test programs (`hello`, `isa_test`, `echo`, `fib`, `primes`, `trap_test`, `kernel_hello`, `kernel_fault`, `irq_test`, `wait_nowake`, `mt_demo`, `vm_test`, `vm_demo`, and the seven `exc_*`), `run_tests.sh` runs both emulators with the same input and requires the following to be identical:

1. the **exit status**,
2. the **console output**, and
3. the `-s` **state trace, line by line**: PC, mode, r1–r15, the other r15, STATUS, EPC, CAUSE and BADADDR at the start of every instruction.

If they differ, the test reports the first differing line. `io_test`, `lib_test` and the scripted monitor session also require identical **disk images** afterwards. `session1` runs the whole P32W Unix session on `p32uemu` against the same expected transcript. Both emulators include the same `p32io.h` and tick the timer at the same points, so any difference comes from the CPU: either the microcode or the C model of a card.

The trace runs at instruction boundaries, so the comparison is exact at the architectural level but does not cover the cycle count.

14.2 Shared sources

`p32opc.h` and `p32uc.h` are generated from `p32isa.py` and `p32ucode.py`, and the assembler imports `p32isa.py` directly. The emulators, the assembler, the microcode and the decode flash therefore cannot disagree about an encoding. `make test` also checks that the generated instruction reference (`docs/instruction-reference.md`) and the bus definition (`docs/bus.md`, from `generators/p32bus.py`, checked against the microcode) are up to date.

14.3 The other checks against the emulators' functions

The FPGA ALU's Verilog is run against the emulator's ALU functions, all 19, with the results sampled when the bus needs them (skipped without `iverilog`). Simulation of the CPU card at gate level, against `p32uemu`'s microcycle counts, comes with the CPU card's netlist.

15. Limits and differences from the hardware

Timing

- `p32emu` has no cycles. Its timer ticks per instruction (≈ 1 MIPS of emulated time; the hardware runs ≈ 0.94 MIPS under P32W Unix), so interrupts land at different instructions than on the hardware.
- `p32uemu`'s **microcycle counts** are those of the microcode, but only `-C` makes the timer cycle-exact. The run time on the real machine is cycles ÷ 3 MHz. Wait states appear only with `-W`, the same number on every bus cycle; flash timing and the ALU's internal timing are not modelled, and every other microcycle is one clock.
- Devices take no time: no baud rate, no disk access time, no BSY (`-G` counts disk commands and console bytes, for adding them on, §4.7). Software that works only because a device is fast will fail on the hardware. Software that follows the status bits will work.
- Interactive idle runs in real time; non-interactive idle skips ahead. Under `-C` idle spins at full host speed.

Not modelled

- IRQ5 (RTC) is never asserted, and the front panel raises no interrupt. The RTC's alarm, periodic and update interrupts, UIP, and register C flags are absent.
- UART FIFOs, line errors, break, modem-control effects; THR is always empty.
- The disk card's drives: only the commands in §11 and LBA addressing; the alternate status reads Status without touching INTRQ, and the device-control register is not modelled. The disk card's DMA engine is not modelled.
- The video card draws no picture (`-V` writes the screen at exit; the debugger's page shows it live). The mouse moves only by `-Km`. Its chip's model stops short of the chip (the head of `p32acrtc.h`).
- The front panel, the debug card (its host link and logic analyzer; `-D` stands in for a DMA master), in-place programming of the boot flash; reset other than power-on (software restarts by jumping to the monitor).
- Power-on contents: RAM and the RTC's RAM start as zeros, the clock as host time or `-R`.
- `p32uemu -C`: the RTC does not advance, because only `io_tick` and `io_wait` move emulated time, and `-C` uses neither.
- `WAIT` with only a UART interrupt enabled (timer off) stops the emulator instead of waiting for input.

Sources: `emulator/p32emu.c`, `emulator/p32uemu.c`, `emulator/p32io.h`, `emulator/Makefile`, `emulator/test/run_tests.sh`, `microcode/p32ucode.py`, `microcode/p32isa.py`, `tools/p32fs.py`, `tools/p32term.py`, `software/monitor/monitor.asm`, `software/tests/run_session.sh`*; decisions D6, D7, D10, D11, D13, D14, D17, D18, D20, D23, D26, D43, D45, D50, D52, D54.*

emulator/

emulator/README.md

The two P32W emulators. The full user guide and theory of operation is ../docs/emulators.md; this page is a summary.

`p32emu.c` — the **instruction-level reference emulator** (D7). It decodes each instruction and performs its effect directly: ISA v0 with stage A traps and privilege, stage B interrupts and timer, and stage C virtual memory (../docs/isa-system.md). It is the golden model that the microcode-level emulator, the FPGA ALU, the gate-level model and the hardware are checked against, and the fast one for running P32W Unix: 23–163 million instructions a second on an M3 Pro (emulators.md §1). Emulated time runs at 1 MHz: one timer tick per instruction.

`p32uemu.c` — the **microcode-level emulator** (D14). It runs the real control-store image from `../microcode/p32ucode.py` one microcycle at a time over a card-level model (register SRAM with the r15 bank rule, PC counter, ALU latches, MAR/MDR with byte lanes, split TLB, sequencer). It knows no instruction semantics. `make test` runs programs on both and requires identical state at every instruction (`-s` traces), identical output and identical exit status. It boots P32W Unix too, 2 to 8 times more slowly.

`p32io.h` — **the devices**, one model included by both emulators (D50, D52): the I/O card (four TL16C550 UARTs, the timer, the input and output ports, the DS12C887 real-time clock (D23), two USB ports and the network coprocessor), the disk card (two CompactFlash sockets and two IDE channels, True-IDE task files, 16-bit data transfers, D26) and the video card (D53; its HD63484 chip in `p32acrtc.h`, D103), with the cards' bus rules (byte registers answer byte accesses only, the timer word accesses only; anything else is BUSERROR).

`p32opc.h` is generated — never edit it; edit `microcode/p32isa.py`.

Build and run

```
make            # generates p32opc.h, the microcode image and p32uc.h; builds both emulators
make test       # the whole project's tests (see below)
./p32emu  [-t] [-m] [-s FILE] [-l N] [I/O options] image.bin
./p32uemu [-u ../microcode/p32ucode.bin] [-s FILE] [-l N] [-c] [I/O options] image.bin
```

Both emulators:

- `-l N` — stop after N instructions (default 100000000; no limit when the console is a terminal, so an interactive session runs until ^] or `halt`; 0 = no limit). A stop prints `STOP: instruction limit (N) reached` on stderr.
- `-s FILE` — write the architectural state at every instruction start (the same format from both).

`p32emu` only: `-t` traces every instruction (address, mode S/U, word, disassembly) and every trap to stderr; `-m` prints TLB statistics at exit, with the miss counts of larger and 2-way "what if" TLBs (emulators.md §3); `-U IMG` plugs a USB drive holding IMG into the I/O card's first CH376S (D43; `FF50`/`FF51`, IRQ6), `-U1 IMG` into the second (`FF54`/`FF55`), and `-Uout N` pulls it out after N sectors have moved; `-M N` fits N memory cards (1, the default, to 4; 16 MiB each, in windows 0 to N − 1, D52): above them RAM is a BUSERROR, so the kernel's memory probe is tested as the hardware will test it — in `p32uemu` too; `-k FILE` writes a kernel profile at exit, the instructions run at each kernel address and in user mode, which `tools/kprof.py FILE` totals by function (from `kernel/unix.map`; static functions count under the one before them).

Both emulators: `-N` starts the I/O card's network coprocessor (D45; `FF52`/`FF53`, IRQ7): the mailbox protocol of docs/net-mailbox.md, carried out with the Mac's own sockets, so P32W reaches the real network. 10.0.2.2 is the Mac; P32W's listening ports below 1024 appear on the Mac 8000 higher (`-Nport N` changes that); TLS goes through `tools/p32tls.py`, trusting the Mac's certificates or `-Nca FILE`. `test/net_peer.py` runs `test/net_model.c` against servers of its own.

The long tests on the hardware models: `sh test/hwtests.sh [MICROPYTHON_TESTS_DIR]` runs every recorded P32W Unix session on `p32uemu`, the full set of programs at gate level, and, given the release's `tests` directory, MicroPython's 645 tests on `p32uemu` against the host reference build (`tools/mptests.py`). It takes about 45 minutes (2026-10-17: 24 of 24 passed); `make test` runs the quick part of the same checks.

`p32uemu` only: `-u FILE` the control-store image; `-c` ticks the timer once per microcycle, as the hardware will, instead of once per instruction (the default, which matches p32emu).

The I/O options (both):

Option	
`-d IMG`	a CompactFlash disk image (raw 512-byte sectors, read and written) in the disk card's first CF socket
`-d1 IMG`; `-d2`, `-d2s`; `-d3`, `-d3s`	the second CF socket; IDE A's master and slave; IDE B's
`-a OUT`, `-A IN`	UART 1's output and input as files (UART 0 is stdin/stdout)
`-T PORT`	UART 1 on a TCP port of localhost: a second terminal (`tools/p32term.py PORT`, or `nc localhost PORT`)
`-a2 -A2 -T2`, `-a3 -A3 -T3`	the same for UARTs 2 and 3
`-V FILE`	the video card's screen at exit: its 640 × 480 dots to FILE.pbm, the console's 25 rows of text (read from the dots) to FILE
`-K FILE`	the PS/2 keyboard types FILE's text
`-Km B0,B1,B2`	the PS/2 mouse sends this packet (hex)
`-p N`	the value the input port reads (default 0)
`-w`	for scripts: feed the console's input a line at a time, each after a prompt (`# $ > login: Password:` and the others in p32io.h)
`-P STR`	a prompt for `-w` (the first `-P` replaces the defaults)
`-R SECS`	the real-time clock starts at SECS (seconds since 1970, UTC) and follows emulated time; without it, it starts at the host's time

Exit status: `r1 & 0xFF` on `HALT`; 255 on an unhandled exception, the instruction limit, or a usage or load error.

Behaviour worth knowing

- **Memory map** (D52): 1 MiB ROM at 0, RAM from `0x0010_0000` on 1 to 4 memory cards of 16 MiB (`-M`), the device registers in the top 32 KiB of the address space (the video card, revision 2, has no memory on the bus: D103). User and supervisor modes with a banked r15; exceptions trap through `VBR`; interrupts are taken between instructions; a software-managed split TLB (64 + 64 entries, direct-mapped, ASID-tagged, D12/D13) translates every user-mode access, and supervisor mode is untranslated.
- **Time:** the timer ticks once per instruction, so interrupt timing is exactly repeatable (D11). `WAIT` skips straight to the timer's expiry, and the HALT line reports the skipped ticks as "idle ticks". The real-time clock follows emulated time

(one microsecond per tick).

- **No handler installed:** while `VBR` = 0 an exception or interrupt stops the emulator with a register dump (D10). A `WAIT` that can never wake up stops it too.

Tests (`test/run_tests.sh`, run by `make test`)

The script is the whole project's regression suite. It runs about 130 checks in these groups (numbered as in the script):

§	What
1–5	the ISA: exact encodings, every instruction (self-checking), console I/O, Fibonacci and primes against Python references, stage A traps, stage B interrupts and a pre-emptive kernel, stage C virtual memory and protected processes, the I/O card and its library, the boot monitor, exceptions stopping the emulator
6	the hardware models: the microcode emulator against the reference on every program, the FPGA ALU's Verilog, the instruction reference up to date
7	the bus definition (../docs/bus.md): consistent with the microcode, every radial line two-ended, the document up to date
12–15	the toolchain on the host: assembler errors, relocatable objects, the linker and archives, p32cc's test programs against the host's C compiler, the disk-image tool and its fsck
16	P32W Unix in p32emu: recorded sessions (`software/tests/session*.in` and `.expect`) covering boot, the shell, users, signals, history, the manual, make, BASIC, SQL, administration; the native toolchain (`cc_test.sh` : the same assembly as p32cc for every C file, and self-hosting byte for byte); the system's own tests on P32W (`/usr/tests`)
17	the archivers (tar, gzip, zip) against the host's

The sessions boot a private image (`software/tests/run_session.sh`), never `software/build/disk.img` .

Hand-written assembly against C (first measurement, 2026-10-14)

docs/bench.md

The backlog item "Measure hand-written assembly against C" asks how much speed assembly buys over p32cc. This is its first round: the inner loops that the kernel, libc and commands spend their time in. Each loop is written twice, in C and in hand-written assembly with the same ABI, and timed in P32W microcycles on the microcode-level emulator (`p32uemu`).

Rerun the measurement with `software/bench/run.sh`, which prints the table below. The files:

- `software/bench/bench.c` holds the C routines and the driver; `-DBENCH=n` picks the routine.
- `software/bench/bench_asm.s` holds the assembly versions.

Each figure has the setup-only baseline (BENCH=0) subtracted. All three builds compute the same results: the exit statuses match.

routine	stands for	C cycles	C, no regvars	asm cycles	C / asm	no-regvars / asm	C / asm at first
strlen (999 B × 20)	libc string functions	404 181	824 235	312 781	1.29	2.63	1.29
wcopy (4 KiB in words × 20)	`memcpy`, `copyin` / `copyout`	864 801	1 622 455	406 861	2.12	3.98	3.08
bytesum (4 KiB × 10)	`wc`, `cksum` loops	1 436 291	2 951 835	626 191	2.29	4.71	3.20
crc32 (4 KiB × 4)	gzip/zip, table lookups	1 099 001	1 934 555	934 633	1.17	2.06	1.42
isort (200 ints)	`sort`, compare-and-move	771 431	1 493 896	372 069	2.07	4.01	2.07
count (4 KiB × 10)	`grep`'s byte matching	1 560 031	2 790 975	1 035 981	1.50	2.69	2.05
hcopy (512 B in halfwords × 40)	the CF sector transfer	439 341	817 995	206 981	2.12	3.95	3.06

The last column is the C compiler before the three fixes below (loop headers, constant offsets, `*p++`).

Where the gap comes from

Register variables (M7 step 1) gain the most. Turning them off makes these loops 1.5–2× slower again.

A fix made during this measurement. p32cc counted a loop's condition and increment as being outside the loop when it chose which variables to keep in registers. So the counter in `while (n--)` stayed in memory. The fix is in both compilers, `cc/cparse.py` and `software/cc/parse.c`, which must agree byte for byte. It made wcopy, bytesum, crc32, count and hcopy 10–18% faster. Any code with a `while (n--)` loop gains, including the kernel and the commands.

What is left is mostly two things:

1. **Post-increment through a temporary.** `s += *p++` compiles to:

   mov r1, r12 addi r12, r12, 1 lbu r1, 0(r1)

   A person writes `lbu r1, 0(r12)` and then `addi r12, r12, 1`. That is one instruction more per `*p++` and per `n--` test. Counting instructions, bytesum's C loop is 7 instructions per byte, against 4 for a plain assembly loop.

2. **Unrolling.** The assembly versions unroll by 4 where it comes naturally (copies, sums, strlen). That takes the loop's overhead from 4 instructions per byte to 2.5. A compiler without an optimiser will not do this; libc can.

What a compiler can't do here:

- The loop of crc32 (1.26×) is almost all real work, so the compiler is close.
- strlen is close too (1.29×): p32cc already walks the pointer in a register.

Done after the measurement (2026-10-14)

- `memcpy` **and** `memset` **in assembly** (`software/libc/mem.s`). libc and the kernel both use it; the kernel's Makefile assembles the same file.
  - Each one aligns, then moves 16 bytes per loop, then words, then the tail. memcpy falls back to bytes when source and destination are differently aligned.
  - 4 KiB timings: memcpy went from 62 800 to 22 500 cycles (2.8×); memset from 45 400 to 14 000 (3.2×).
  - `cc/tests/mem.c` checks every alignment and lengths 0–40 against a byte loop.
  - Every `copyin`/`copyout`, `read`/`write` through the buffer cache and struct copy uses them.
- **Constant offsets folded into loads and stores** (both compilers):
  - `p[2]` and `*(p + 3)` become `lw r1, 8(rP)` instead of an `addi` then a load.
  - A constant address within ±32 KiB of 0 (the I/O page at the top of the address space) becomes an offset from r0: `sh r1, -4352(r0)` for the CF data register (`0xFFFF_EF00`), instead of `li` plus `sh`.
  - So the kernel's CF sector loop now takes 2 instructions per halfword instead of 4, and every device register access is one instruction.
  - (This doesn't change the table: its loops walk pointers with `*p++`.)
- `*p++` **on a register pointer** (both compilers): the load or store goes through p, then `addi` moves it, with no temporary in between.
  - This takes one instruction out of every pointer-walking loop: 7–20% on the loops above.
  - The suite's check that the C compiler matches p32cc caught a missing case (the store side) before this went in.
  - The remaining compiler gap is the `n--` test, which goes through a temporary, plus unrolling.

What this suggests

- **libc and the kernel:** a few routines are worth writing in assembly, at 2.6–3× faster each (memcpy and memset are done; see above):
  - `memcpy`, `memset` and `copyin`/`copyout`, as unrolled word copies;
  - the CF transfer loop (`cf.c`), unrolled halfword moves;
  - `strlen`.

  These are small, stable routines, called everywhere.
- **The compiler:** `*p++` is done. `while (n--)` still copies n to a temporary to test it; a branch on n, then the decrement, would save one more instruction per loop. Any change has to go into both compilers (cc_test compares them).
- **Commands as a whole** (wc, grep, sort and a boot-plus-login figure) and **the kernel's trap path** are the next round of this measurement. Their time is spread across code that assembly would not change, so the ratio for a whole

program will be well under these inner-loop figures.