A Fable 5.1 / Opus 5.5 experiment, run for one month, to implement different instruction set architectures. ARM, SPARC and RISC-V were generated almost instantly. X86 was a bit harder. The rest of the month was spent on continuous debugging boot of Windows 98. Hammering out the difficult cases took Fable running in agent mode for days.

Getting started

Clone and start the menu:

```
git clone https://github.com/eiselekd/l8hdl_pub.git
cd l8hdl_pub/cpu
make
```

`make` asks a few questions (below), checks the programs your machine needs and says how to install what is missing, downloads what has to be downloaded, programs the board and starts the console. `make help` lists the single steps.

Verilog source

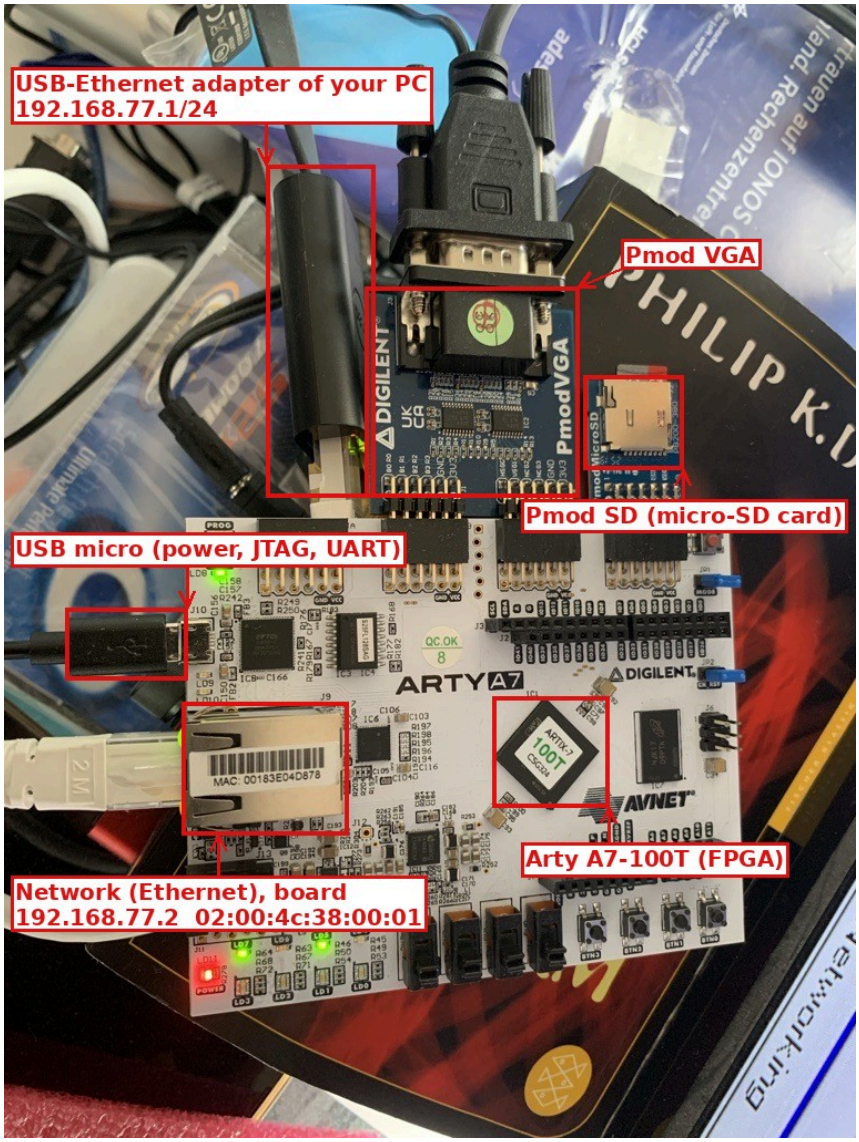
The Verilog of every system is in this archive: generated from the high-level description, one file per system, without comments. Each `top.v` is the whole system as Verilator simulates it -- the processor, its caches, its memory and its peripherals -- and `tb.cpp` beside it is the C++ testbench that drives it.

system	Verilog	top module	testbench
x86, Linux	verilator/x86/top.v (`verilator/x86/top.v`)	`x86_pc_linux_top`	tb.cpp (`verilator/x86/tb.cpp`)
x86, Windows 98	verilator/win98/top.v (`verilator/win98/top.v`)	`x86_pc_win98_top_hb`	tb.cpp (`verilator/win98/tb.cpp`)
SPARC V8, Linux	verilator/sparc/top.v (`verilator/sparc/top.v`)	`top`	tb.cpp (`verilator/sparc/tb.cpp`)
ARMv8-A (A32), Linux	verilator/arm/top.v (`verilator/arm/top.v`)	`top`	tb.cpp (`verilator/arm/tb.cpp`)
Cortex-M3, FreeRTOS	verilator/m3/top.v (`verilator/m3/top.v`)	`top`	tb.cpp (`verilator/m3/tb.cpp`)
serv (RISC-V), FreeRTOS	verilator/serv/top.v (`verilator/serv/top.v`)	`top`	tb.cpp (`verilator/serv/tb.cpp`)

CPUs

Processors written in a high-level hardware description language, running real operating systems on a **Digilent Arty A7-100T** FPGA board -- or simulated on your PC with **Verilator**:

- a microcoded **32-bit x86** booting **Linux 6.12** and **Windows 98**, with dial-up **internet over a modem** (PPP),
- a **SPARC V8** and an **ARMv8-A** (A32 state), both booting **Linux 6.12**,
- two small cores running **FreeRTOS**: **serv** (a bit-serial RISC-V, RV32I) and an **ARMv7-M** (Cortex-M3 shape).



What you see on the board:

Arty A7-100T	the FPGA board (Xilinx Artix-7 100T) that carries the processor and its peripherals
Pmod SD	a Digilent Pmod MicroSD with a micro-SD card: the Windows 98 hard disk
Pmod VGA	a Digilent Pmod VGA: the PC's screen on a real monitor (optional -- the screen is also shown on your PC)
USB micro	the board's USB port: power, programming (JTAG) and the serial console

Network (Ethernet)	the board's Ethernet port: the console, the debugger and the disk link to your PC. Address 192.168.77.2, MAC 02:00:4c:38:00:01
USB-Ethernet adapter	the adapter on your PC (top left, beside the Pmod VGA); its cable runs under the board to the board's Ethernet port. Address 192.168.77.1/24

Choosing what to run

The menu asks one question after the other; each answer can also be given on the command line.

question	choices	on the command line
1. Where should the system run?	`arty` -- the Arty A7-100T board, or `verilator` -- simulated on this PC	`TARGET=arty` / `TARGET=verilator`
2. Which processor?	`x86`, `arm`, `sparc`, `serv`, `m3`	`CPU=...`
3. Which system on the x86? (x86 only)	`linux` -- Linux 6.12, or `win98` -- Windows 98	`OS=linux` / `OS=win98`
4. A microSD Pmod on the Arty's JD connector? (Windows 98 on the Arty)	`yes` -- the disk is the card, `no` -- the disk is a file on this PC, served over Ethernet	`SD=yes` / `SD=no`
5. Write the Windows 98 image onto the card first? (with SD)	`yes` -- a first run, ~15 min, `sync` -- bring the card back to the image, ~3 min, `no` -- the card as it is	`WRITE=yes` / `WRITE=sync` / `WRITE=no`
6. Networking through SLIRP on this PC? (Windows 98 on the Arty)	`slirp` -- the dial-up provider on the PC's modem port, `no`	`NET=slirp`

Every combination, started directly:

what	command
Windows 98 on the Arty, disk on the SD card, with dial-up internet	`make TARGET=arty CPU=x86 OS=win98 SD=yes WRITE=sync NET=slirp`
Windows 98 on the Arty, disk on this PC (no SD Pmod; an older bitstream: dial-up is reliable only with the SD card)	`make TARGET=arty CPU=x86 OS=win98 SD=no`
Linux on the x86	`make TARGET=arty CPU=x86 OS=linux`
Linux on the ARMv8-A	`make TARGET=arty CPU=arm`
Linux on the SPARC V8	`make TARGET=arty CPU=sparc`
FreeRTOS on serv (RISC-V)	`make TARGET=arty CPU=serv`
FreeRTOS on the Cortex-M3	`make TARGET=arty CPU=m3`
any of them simulated	the same with `TARGET=verilator` (for the x86: `OS=linux` or `OS=win98`)

Windows 98 and the internet

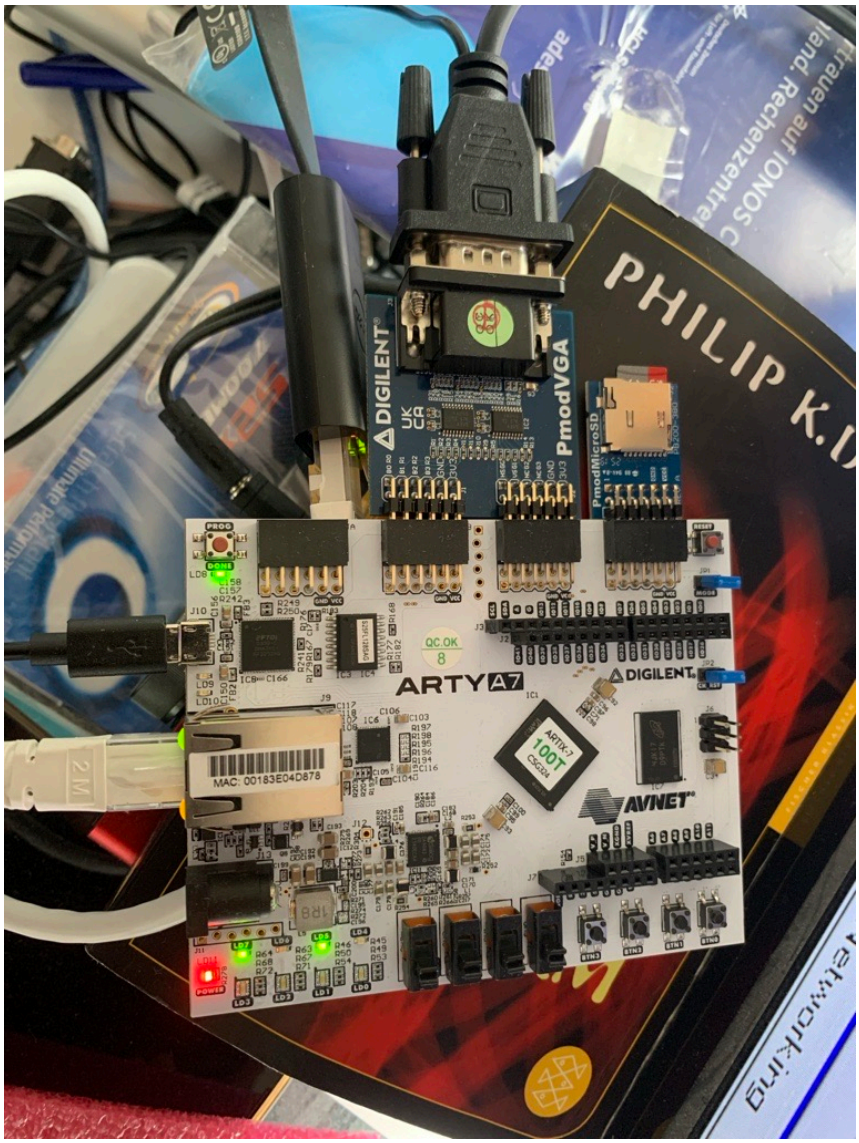
The Windows 98 disk image is copy.sh's (taken from `images/` when present, downloaded otherwise) and patched on your machine (see the note below); it comes up with a modem on COM2 and a Dial-Up Networking connection already set up.

> **TIP -- OPEN THINGS WITH ONE CLICK AND ENTER, NOT A DOUBLE CLICK.** On the Arty a double click is often not recognised. Click an icon **once** to select it, then press **Enter** to open it.

Start it with `NET=slirp`, then in Windows:

1. select **My Computer** (one click) and press **Enter**; then the same with **Dial-Up Networking**, then with **My Connection** -- or Start, Run: `rundll32 rnaui.dll,RnaDial My Connection`;
2. press **Connect** (or Enter; any user name and password); after about 15 seconds "Connection Established" appears;
3. start **Internet Explorer** (its start page is http://l8hdl.se), or try `ping www.example.com` in an MS-DOS Prompt.

The modem, the PPP peer and the network behind it run on your PC, in user space (libslirp: no root rights). To end, use Start, Shut Down and wait for the message below before you switch off or reprogram the board -- Windows hangs the modem up on its own.



> **NOTE -- THE WINDOWS 98 IMAGE IS PATCHED ON YOUR MACHINE.** The disk image is copy.sh's (the image of copy.sh's v86 demo), with its SeaBIOS and VGA BIOS. The archive normally carries the three in `images/`, as downloaded; only when one is missing (as in a git checkout) is it downloaded from copy.sh -- or, when copy.sh fails, from the same files at l8hdl.de, l8hdl.com, l8hdl.se in that order (`images/win98/`, `images/bios/`; `MIRRORS="…"` sets another list). All three are checked by sha256, whichever source they came from. Then a copy of the disk image is **modified on this machine**: this archive's "no wizard" patch (6 910 sectors) installs the driver for the system's PCI VGA, so Windows 98 starts without the Add New Hardware Wizard. A second patch (5 255 sectors) on top of that copy adds a serial port COM2, the "Standard 33600 bps Modem" on it, a Dial-Up Networking connection "My Connection" and Internet Explorer's start page set to http://l8hdl.se. The original stays as it was, `images/windows98.img`; the patched copies are `images/windows98_vga.img` and `images/windows98_net.img` (the one the board and the Verilator run use).

What it needs

- **On the Arty:** an Arty A7-100T on USB (programming) and on ethernet (the console and the debugger): the host needs the address 192.168.77.1/24 on that interface — `make check` says what to plug in and which command to run. For Windows 98 a Digilent Pmod MicroSD on **JD** with a microSD card (the

image is written to it) — or none (`SD=no`): then the disk is the image file on this machine, served to the board over ethernet (`KEEP=1` keeps what Windows writes for the next run; without it every boot starts from the image as downloaded). A Pmod VGA on JB/JC is optional — the screen is shown on this machine in any case.
- **On this machine:** `python3` and the C / C++ toolchain (Debian's `build-essential`: `cc`, `c++`, `make`, the C library's headers -- the console's screen, the OCR and the host halves are built on first use); for Windows 98 also `curl` (or `wget`) and `mtools`; for the Verilator target `verilator` (5.0 or newer). `make` checks and prints one `sudo apt install …` line for Debian / Ubuntu; on another distribution install the same programs by its own means. The programmer (openFPGALoader) comes with the archive, and its source too, should the binary not run here.

The IP setup

Two networks are involved: the cable between your PC and the board, and -- only for Windows 98's internet -- the dial-up line inside the board's COM2.

network	who	address
the Ethernet cable (192.168.77.0/24)	your PC's USB-Ethernet adapter	**192.168.77.1/24** (set by you)
	the Arty board	**192.168.77.2**, MAC 02:00:4c:38:00:01 (both fixed in the bitstream; the MAC on the board's sticker is not used)
the dial-up (PPP over COM2, slirp on your PC; 10.0.2.0/24)	Windows 98	10.0.2.15 (given by the provider when it dials)
	the gateway (your PC, through slirp)	10.0.2.2
	the name server (your PC's resolver, through slirp)	10.0.2.3

On Linux, once (the adapter's name from `ip link show`, e.g. `enx00e04c…`):

```
sudo ip addr add 192.168.77.1/24 dev <name>
sudo ip link set <name> up
```

`make check` tests the link and prints these lines if the address is missing. No routing, NAT or root rights are needed for Windows 98's internet: slirp runs as your user and opens ordinary sockets on your PC.

The consoles

- **Linux:** the system's serial console, in this terminal.
- **Windows 98:** the screen in this terminal — text as text, graphics as ASCII art; **Ctrl-O** shows the screen's text read back by OCR, **Ctrl-G** opens the real picture in the browser. Keys and the mouse go to the machine (from the terminal, or from the browser tab: click the picture to capture the mouse, Esc

releases it). Open icons by selecting them with one click and pressing **Enter** -- a double click is often not recognised. **Ctrl-D** opens a SoftICE-style debugger over the lower half of the screen, in the terminal and in the browser tab alike (in the browser also the **Debugger** button in the status bar or the **Pause** key, since Firefox keeps Ctrl-D for itself): the processor halts, a window shows its registers, the stack, the code at EIP, and a command line with SoftICE's commands (`?` lists them: R D E U T P G BPX BPM BPR BL BC STACK, and PCH WT PORT SREG for what the hardware records -- the pc history, the bus write trace, the port trace). Ctrl-D again, or `X`, resumes. `U` disassembles with `objdump` (binutils) when it is installed, and shows bytes otherwise.
- **Ctrl-]** leaves the console; on the Arty the system keeps running.

The co-simulated parts: screen, keyboard, mouse, modem, disk

The Windows 98 PC has no keyboard, mouse or network connector of its own on the Arty. Those devices are split in two: the **PC side** is ordinary hardware in the FPGA (the 8042 keyboard controller, the VGA, the 16550 on COM2, the IDE disk), and the **host side** runs on your PC and talks to it over the Ethernet cable through one register block, `vga_host` at 0xC002_0000 on the board's debugger bus. Under Verilator the same register block is served by the model on a local TCP socket, so the host programs are the same for the board and the simulation; only the transport differs.

```
your PC                                  the Arty
console.py (terminal) --+                          +-- VGA planes + palette
webview.py (browser)  --+  Ethernet                +-- 8042: keys, PS/2 mouse
dialup.py  (modem)    --+-(UDP link)--> vga_host -+
blk_system (disk)     --+             0xC002_0000 +-- COM2 queues + modem lines
                                                  +-- blk_proxy (top_hb.bit)
under Verilator: simlink.py, a local TCP socket to the model
```

part	board side	host side	what it does
Screen	`vga_host` reads the four VGA planes (512 words a request) and the palette out of the block RAM in the vertical blank only, so the picture and the CPU are never disturbed	`tools/console.py` (terminal), `tools/webview.py` (browser, Ctrl-G), drawn by `cosim/vga/vga_screen.h` through `vga_ffi.cpp`	text mode as text, mode 13h / mode X and the 640x480 16-colour desktop as pixels; about 0.9 frames a second
Text on the screen	--	`cosim/vga/ocr/` (`vga_ocr.h`)	reads the text of a graphics screen back by matching the Windows bitmap font pixel for pixel (Ctrl-O): menus, dialogs, icon labels
Keyboard	a write to `SCAN` pushes a byte into the stream the 8042 takes, as if from a PS/2 keyboard	console.py from the terminal's keys; webview.py from the browser's `KeyboardEvent.code`	set-1 scan codes, press and release

part	board side	host side	what it does
Mouse	a write to `MOUSE` pushes a byte into the 8042's auxiliary stream	console.py from the terminal's mouse reports (xterm SGR); webview.py from the browser tab (click the picture to capture the pointer, Esc releases it)	3-byte PS/2 packets
Modem / internet	COM2's 16550: its bytes out through a ring, the host's bytes in, the modem lines (`VH_COM2` / `VH_C2N` / `VH_C2L`)	`tools/dialup.py` with `cosim/eth/`: `modem.h` (a Hayes modem: AT commands), `ppp_peer.h` (the provider's PPP), `eth_slirp.h` (libslirp)	Windows' Standard 33600 bps modem driver talks to it; any number dialed connects; see "Windows 98 and the internet"
Disk without an SD Pmod	`blk_proxy` in place of the SD controller behind the IDE (`top_hb.bit`)	`cosim/hostblk/blk_system.cpp` through the bridge (`cosim/bridge/`)	the sectors come from `images/windows98_net.img` on your PC; writes kept in memory (or `KEEP=1`)

The C++ host halves are compiled on first use and cached (`cosim/cdll.py` for the ones loaded into the Python tools; `cosim/bridge/host.sh` for the disk): a C++ compiler is all they need, and the dial-up provider libslirp (`libslirp-dev`). The same headers serve the Verilator testbench directly: `vga_door.h` reads the model's planes, and `vga_host_door.h` answers the board's register block from the model, so `console.py` and `webview.py` attach to a simulated PC exactly as to the board.

The FreeRTOS systems: serv and the Cortex-M3

```
make TARGET=arty CPU=serv          # or CPU=m3
make TARGET=verilator CPU=serv     # or CPU=m3
```

Both run the same application, **the DSP ring**: four FreeRTOS tasks pass packets round a ring -- a generator makes two tones, one station transforms them (a 64-point FFT), one filters the higher tone away, one transforms back, and the generator checks that the lower tone alone came back. Equal-priority stations share the processor by time slicing, the generator preempts them, semaphores and a mutex hold it together. It never stops: every packet prints a line at once,

```
dsp ring on freertos
packet 0 ok     sum 1b743c3c  ok 1 wrong 0  tick 107
packet 1 ok     sum be4b396a  ok 2 wrong 0  tick 173
...
stations' background work: fft 1494 filter 4598 ifft 0
```

-- the packet's number and verdict, a checksum of the samples that came back (it repeats every eight packets: the amplitude is the packet number's low three bits, so a checksum out of that cycle is a wrong sample), the running counts and the FreeRTOS tick (100 a second on the board). Every sixteen packets a line says how much background work the stations did between packets. Both systems print the same checksums.

- **On the Arty** only the **USB cable** is needed (no ethernet): the program is inside the bitstream (64 KB of block RAM), the processor starts the moment the board is programmed, and the console is the board's

serial line (115200 8N1, the FTDI's second port: `/dev/ttyUSB1`, or `TTY=/dev/ttyUSBn`; your user must be allowed to open it, usually the `dialout` group). The console is already listening while the board is programmed. **Ctrl-]** leaves; the board keeps running (`python3 tools/serial_console.py /dev/ttyUSB1` attaches again). Both cores run at 25 MHz. Measured on the board (2026-10-10): the Cortex-M3 prints a packet every 3 ticks (some 33 lines a second), serv (bit-serial) one every ~65 ticks (about 1.5 lines a second); both print the checksums the Verilator runs print.

- **Under Verilator** the emitted Verilog of the same design -- the 16550's line decoded by the testbench as the board's own path does -- is built on first use (a minute) and runs until **Ctrl-C**: the M3 model some 600 000 clocks a second (about a packet a second), the serv model some 3 million (the first packet after about ten seconds, then one every five). `LIMIT=clocks` ends a run.
- `systems/serv/dspring_forever.bin` and `systems/m3/dspring_forever.bin` are the programs the RAM holds.

Performance

All numbers below were measured, on the Arty A7-100T or under Verilator on the build host (20 cores), unless marked as derived. The Arty's memory controller runs at 81.25 MHz; the processors run in their own clock domain, a divided copy of it, so the clock of each system is what its timing closure allowed.

system	processor clock	instruction throughput	on the board
x86, Windows 98	20.3 MHz (81.25 / 4)	~5.25 clocks an instruction on kernel / VMM code, so ~3.9 M instructions a second (derived) -- about a 386 at 25 MHz	desktop **148 s** from the boot's start (SD card) (191 s with a screenshot every 20 s); the VGA screen at **~0.9 frames a second** in the browser / console view
x86, Linux	16.25 MHz (81.25 / 5)	the same core: ~5.25 clocks an instruction, ~3.1 M instructions a second (derived)	root prompt 63 s after `run-linux`
SPARC V8, Linux	81.25 MHz	MMU under the caches, 39.68 BogoMIPS	login prompt 17 s after `run-linux`, the image upload included
ARM (A32), Linux	16.25 MHz (81.25 / 5)	2.00 BogoMIPS (timer-based delay loop: says nothing about the core)	shell after 48 s of kernel time, 2 min 14 s wall clock (80 s of it the 17 MB image upload)
Cortex-M3, FreeRTOS	25 MHz	--	the DSP ring: a packet every 30 ms (3 ticks of 10 ms), no wrong packet in 2000+
serv (RISC-V), FreeRTOS	25 MHz	bit-serial: some tens of clocks an instruction	the DSP ring: a packet every ~0.65 s (~65 ticks), first after 1.07 s

Where the x86's clocks go. The microcoded core was measured on the Linux kernel's boot (3.8 M instructions, the front end's state recorded each clock): 15.8 clocks an instruction at the start, then 8.5 (the fetch window kept across instructions), 6.0 (the decoder runs one instruction ahead), 5.25 (the decoder answers from a partial window). What remains is mostly the microcode routine itself: `mov r, [m]` is three micro-operations, `push` / `pop` three, `call` five. Hot loops of simple instructions do better -- the simulation's micro-benchmark (clocks an instruction, caches warm):

instruction	clocks
`mov r, r` / `test`	1.25 (1.0 in a run)
`mov r, [m]`	3.7
`jz` taken	2.25
`call` + `ret` (the pair)	23
`div` (32-bit)	50
`rep movsd`, a dword	6.8

Windows 98's boot, improvement by improvement (desktop time on the board): 1320 s on the first working core, 810 s after the fetch / decode / cache work, 840 s with the SD card on its native 4-bit bus (the disk was no longer the limit: busy ~11 s of the boot), 191 s and finally 148 s with the processor at 20.3 MHz and the branch work (no hardware wizard any more). A profile of the 148 s boot is flat: the VMM and its drivers take 56 %, there is no idle loop to remove.

The screen. The VGA frame is read out of the board over Ethernet plane by plane and redrawn on the host: about 0.9 frames a second. Enough to work with the mouse and the keyboard, not for video.

Under Verilator (the emitted Verilog, this host):

model	clocks a second	boot
x86 Windows 98	~42 000	~3·10^9 clocks to the desktop: **on the order of 20 hours**
x86 Linux	--	257.0 M clocks to the `~ #` prompt, 98 min
SPARC Linux	--	289.5 M clocks, 37 min
ARM Linux	--	411.8 M clocks, 41 min
Cortex-M3 FreeRTOS	~600 000	a ring packet about every 1.2 s
serv FreeRTOS	~3 000 000	first packet after ~9 s, then one every ~5 s

References: what each processor is checked against

Every core was built against a working reference -- a "golden sample" that runs the same binary -- and a mismatch against it is a bug in the core until shown otherwise.

processor	golden reference	how it is compared
x86 (microcoded, 486-class)	**v86** (copy.sh's JavaScript PC) for Windows 98; **qemu-system-i386** for the test programs and Linux; the Intel SDM for the instruction semantics	Windows 98: the same disk images run in v86 (port traces, the dial-up's host side, what a driver's routine returns); every test program (`pc_check.sh`) and the random instruction streams (torture) also built as a qemu flavour and their result words / program-counter streams compared; a suspect DLL routine's exact bytes run on both

processor	golden reference	how it is compared
SPARC V8	**qemu-system-sparc -M leon3_generic**	the same PROM + kernel boot there to a root shell; the same test programs' output compared
ARM (A32, ARMv8-A AArch32)	**qemu-system-arm -M virt -cpu max**	test programs whose result words are qemu's; a random instruction stream with registers and flags folded into a trace word after each instruction, qemu's trace beside the core's
Cortex-M3 (ARMv7-M)	**qemu-system-arm -M mps2-an385**	instruction for instruction: qemu's program-counter stream (`-d exec`, one instruction a block) beside the core's, then all sixteen registers per instruction (`compare_regs.sh`)
serv (RISC-V RV32I)	**olofk/serv** (Olof Kindgren's SERV, Verilog)	using it as spec; runs its Zephyr and hello programs

The rule behind it: when a working reference exists, copy it line by line first and restructure only once it runs. Debugging a full system against a reference that agrees instruction by instruction found many of the bugs in `doc/history.md`.

Inside

directory	what
`scripts/`	the questions, the checks, the images, the runs
`systems/`	per processor: the Arty bitstream and the Linux images (serv, m3: the bitstream carries the program); `win98/`: its bitstream, its boot stub and the "no wizard" patch
`tools/`	the console, the board's debugger protocol, the card programmer, openFPGALoader
`cosim/`	the C++ host halves of the co-simulated peripherals (the VGA screen and its OCR, …)
`images/`	copy.sh's Windows 98 image and its two BIOSes (downloaded on the first run when missing), and the patched copies made here
`doc/`	the pictures above, this page as a PDF, the x86's history

How it was built

`doc/history.md` is the record of the x86 processor's way from its first instruction to Windows 98 on the board: every bug that stood in the way, how it was found and what fixed it -- read the history (`doc/history.md`).

SoC diagrams

What each bitstream holds, from the board tops and SoC files named under each picture. The boxes inside the domain frame run in the processor's clock; the host link, the DDR3 controller and (for Windows 98) the VGA and the SD card controller run in the memory controller's 81.25 MHz.

x86, Windows 98 (`systems/win98/top_sd.bit`)

```
PC clock domain: 20.3 MHz (81.25 MHz / 4)

  microcoded x86, 486-class
  x87: small sequential FPU
  I-cache 8 KB (2 x 256 x 16 B)
  D-cache 8 KB, write buffer 8

            |  memory
            v

  AHB: pcmap (reset alias,
    VGA ROM window, A20)
  -> multilayer -> async fifos

  I/O ports                                IRQ
  8259 x 2 (master 0x20, slave 0xA0)
  8254 PIT 0x40-0x43, port 0x61            0
  8042 kbd/mouse 0x60 / 0x64               1, 12
  CMOS / RTC 0x70-0x71                     8
  16550A COM1 0x3F8                        4
  16550A COM2 0x2F8 (the modem)            3
  IDE 0x1F0-0x1F7, 0x3F6                   14
  DMA, floppy stub, port 0x92 A20
  PCI config 0xCF8: 440FX, PIIX3 ISA,
    PIIX3 IDE, VGA 1234:1111 (ROM BAR)
  VGA 0x3C0-0x3DA, debug port 0x402

                            |  async fifos (PC <-> 81.25 MHz)
                            v

81.25 MHz domain

  DDR3 256 MB (MIG)
  PC physical 0 = DDR 0

  VGA: planes 256 KB BRAM
  640x480, Pmod VGA JB/JC
  vga_host 0xC002_0000

  SD host, 4-bit bus
  Pmod MicroSD on JD
  IDE -> blk -> sd_host

                            |
                            v

Ethernet (DP83848 MII), 192.168.77.2, UDP:
0x8E92 debugger bus master: DDR at 0x4000_0000, control 0xC000_0000,
       sd_prog 0xC001_0000, vga_host 0xC002_0000 (screen, keys, mouse, COM2),
       traces 0xC003_0000..0xC005_0000
0x8E94 console: COM1 bytes
```

`x86/microcodev0/win98/soc/arty/x86_pc_win98_arty.l8`, `win98/soc/x86_pc_win98.l8`; `top_hb.bit` is the same with the SD host replaced by `blk_proxy` (the disk on your PC).

Pipeline of the microcoded x86:

```
FRONT  fetch window
  4 words (16 B), kept across instructions,
  refilled while a routine runs;
  I-cache 8 KB, streaming read, hit under fill;
  fetch TLB; a store into code empties it

  |
  v

D1  pre-decode
  prefixes, operand / address sizes, ModRM,
  immediates; the LENGTH in the request's clock;
  direct target (jmp / call rel); interrupt
  line and faults sampled

branch prediction (front)
bimodal: 512 x 2-bit
target buffer: 64
return stack: 16

x87 (FPU = 2): a sequential
unit on the M stage's
port space; busy polled

measured: 5.25 clocks an
instruction (Linux kernel
```

```
   |                                                  boot); mov r, r: 1.25 (sim.)
   v
┌──────────────────────────────────────────────────┐
│ D2  decode                                       │
│   decode tree -> MacroOp (entry, registers, imm); │
│   entries: interrupt (inta), #PF, exception,     │
│   TLB-miss walk; decodes AHEAD while a routine   │
│   runs (2-entry queue), taken at the retire      │
└──────────────────────────────────────────────────┘
   |
   v
┌──────────────────────────────────────────────────┐
│ R  microcode ROM                                 │
│   8192 x 69 bit block RAM; sequencer: next,      │
│   jmp, cjp, call/ret (4 deep), loop, map         │
└──────────────────────────────────────────────────┘
   |
   v
┌──────────────────────────────────────────────────┐
│ I  issue                                         │
│   temporaries renamed (8 a routine, 4 frames);   │
│   hazards: pending load, flags / counter,        │
│   list register, CSR                             │
└──────────────────────────────────────────────────┘
   |
   v
┌──────────────────────────────────────────────────┐
│ X  execute                                       │
│   register read, barrel shifter, ALU, ADD3       │
│   (an address in one uop); multiplier (2 ranks), │
│   divider (a bit a clock); data TLB 16 entries;  │
│   D-cache issue; no forwarding                   │
└──────────────────────────────────────────────────┘
   |
   v
┌──────────────────────────────────────────────────┐
│ M  memory                                        │
│   D-cache 8 KB, write buffer 8; load aligned;    │
│   I/O ports; a missing translation is killed     │
│   here, the epoch flips, the pc decodes again    │
└──────────────────────────────────────────────────┘
   |
   v
┌──────────────────────────────────────────────────┐
│ RETIRE                                           │
│   one-entry commit (RD RS1 RDH PC NPC FLAGS)     │
│   at the routine's last uop; rep strings keep    │
│   the elements done (resume)                     │
└──────────────────────────────────────────────────┘
```

A macro-instruction is decoded into a microcode routine; its micro-operations flow R I X M. `cpux86.l8` (front, decoder), `cpu.l8` (the micro-op pipe), `microcodex86.c` (the ROM).

x86, Linux (`systems/x86/top.bit`)

```
┌──────────────────────────────────────────────────────────────────────────────┐
│ PC clock domain: 16.25 MHz (81.25 MHz / 5)                                   │
│                                                                              │
│ ┌──────────────────────────────────────────┐  ┌──────────────────────────┐  │
│ │ microcoded x86 (486-class), no x87       │  │ I/O ports                IRQ │  │
```

```
│ │ I-cache 8 KB, D-cache 8 KB (2 x 256 x 16 B) │   │ 8259 0x20-0x21                  │ │
│ └──────────────────────────────────────────────┘   │ 8254 PIT 0x40-0x43        0     │ │
│                │  memory                           │ 16550 0x3F8-0x3FF         4     │ │
│                v                                   │ cpuid / rdtsc (dev_ident)       │ │
│ ┌──────────────────────────────┐                   │ other ports read all ones       │ │
│ │ AHB multilayer -> ahb_slave │                   └──────────────────────────────────┘ │
│ │ -> async fifos               │                                                        │
│ └──────────────────────────────┘                                                        │
└─────────────────────────────────────────────────────────────────────────────────────────┘
                              │
                              v
┌──────────────────────────┐  ┌──────────────────────────────────────┐
│ DDR3 256 MB (MIG)        │  │ Ethernet, 192.168.77.2, UDP:         │
│ PC physical 0 = DDR 0    │  │ 0x8E92 debugger: DDR at 0x4000_0000, │
└──────────────────────────┘  │        control 0xC000_0000           │
                              │ 0x8E94 console: the 16550's bytes    │
                              └──────────────────────────────────────┘
```

`x86/microcodev0/soc/arty/x86_pc_arty.l8`, `soc/x86_pc.l8`.

The pipeline is the microcoded core shown under Windows 98, built without the x87 (`FPU = 0`) and with its own generics (`soc/x86_pc.l8`).

SPARC V8, Linux (`systems/sparc/top.bit`)

```
┌──────────────────────────────────────────────────────────────────────────────────────────┐
│ one clock domain: 81.25 MHz (the memory controller's)                                    │
│                                                                                          │
│ ┌──────────────────────────────────────────────────────┐                                │
│ │ SPARC V8 integer unit, 7 stages (after LEON3's iu3) │                                │
│ │ 8 register windows, multiplier, divider, no FPU     │                                │
│ │ SRMMU: 16-entry TLB, hardware table walk            │                                │
│ │ I-cache 8 KB, D-cache 8 KB (2 x 256 x 16 B), VIPT,  │                                │
│ │ write buffer 8; cacheable: 0x4000_0000 only         │                                │
│ └──────────────────────────────────────────────────────┘                                │
│                   │  128-bit cache port; uncached: AHB multilayer                        │
│                   v                                                                      │
│ ┌──────────────────────────────────────┐  ┌──────────────────────────────────────────┐ │
│ │ DDR3 256 MB (MIG) at 0x4000_0000    │  │ AHB -> APB bridge 0x8000_0000         IRQ │ │
│ │ PROM 16 KB loaded at 0x4000_0000,   │  │ APBUART           0x8000_0100        2    │ │
│ │ kernel 0x4000 behind it             │  │ IRQMP (to the core's IRL) 0x8000_0200    │ │
│ └──────────────────────────────────────┘  │ GPTIMER, 2 timers 0x8000_0300        8, 9 │ │
│                                           │ plug & play ROM   0x800F_F000             │ │
│                                           └──────────────────────────────────────────┘ │
└──────────────────────────────────────────────────────────────────────────────────────────┘
                              │
                              v
┌──────────────────────────────────────────────────────────────────────────────┐
│ Ethernet, 192.168.77.2, UDP:                                                 │
│ 0x8E92 debugger: DDR at 0x4000_0000, control 0xC000_0000,                    │
│        sparcv8_dbgif 0xC001_0000 (halt, step, registers, 4 breakpoints)      │
│ 0x8E94 console: the APBUART's bytes                                          │
└──────────────────────────────────────────────────────────────────────────────┘
```

`sparc/sparcv8/soc/arty/sparcv8_linux_arty.l8`, `soc/sparcv8_soc_fast.l8`, `soc/sparcv8_soc_real.l8`.

Pipeline of the SPARC V8:

```
F  fetch
  I-cache 8 KB (VIPT), 128-bit line port;
  next pc: trap vector, jump from E, branch
  from D, pc + 4

   |
   v

D  decode
  decode tree, register file read;
  Bicc / call resolved HERE: the delay slot
  hides it (annul for b,a); windows: save /
  restore move CWP, WIM traps

   |
   v

A  operands
  newest of E / M / X / W or the file;
  load-use: one stall (ldlock)

   |
   v

E  execute
  ALU, shifts, address; jmpl / rett target
  (2 bubbles); D-cache access starts;
  multiplier stage 1

   |
   v

M  memory
  D-cache answers, load aligned, store out;
  multiplier: 4 partial products;
  MMU refusal captured

   |
   v

X  exception / commit
  register file, PSR, %y, WIM, TBR;
  precise traps: epoch flips, %l1 / %l2,
  fetch from TBR

   |
   v

W
  shadow stage: the last forwarding source
```

```
divider: sequential,
a bit a clock (~34),
holds the pipe; /0 traps
```

```
SRMMU: last-page register
+ 16-entry TLB, hardware
table walk on the narrow
AHB port
```

```
any cache miss holds
every stage (hold)
```

Seven stages after grlib's LEON3 integer unit; `sparc/sparcv8/sparcv8.l8`.

ARM A32, Linux (`systems/arm/top.bit`)

```
core clock domain: 16.25 MHz (81.25 MHz / 5)
```

```
┌──────────────────────────────────────────────────────────────────────────────┐
│ ┌──────────────────────────────────────┐   ┌──────────────────────────────────┐ │
│ │ ARMv8-A AArch32 (Cortex-A32 shape)   │   │ devices            IRQ           │ │
│ │ 7 stages F D A E M X W; no VFP       │   │ 16550 0x1000_0000  GIC 33        │ │
│ │ ARMv7 MMU, 16-entry TLB              │   │ GIC   0x1002_0000                │ │
│ │ I-cache 8 KB, D-cache 8 KB           │   │   CPU 0x1002_1000                │ │
│ │ generic timer (CP15 c14) 1 MHz       │   │ timers (core)      GIC 30, 27    │ │
│ └──────────────────────────────────────┘   └──────────────────────────────────┘ │
│              │  memory                                                          │
│              v                                                                  │
│ ┌──────────────────────────────┐                                                │
│ │ AHB multilayer -> ahb_slave  │                                                │
│ │ -> async fifos               │                                                │
│ └──────────────────────────────┘                                                │
└──────────────────────────────────────────────────────────────────────────────┘
                         │
                         v
┌──────────────────────────────────┐   ┌──────────────────────────────────────┐
│ DDR3 (MIG), physical 0 = DDR 0   │   │ Ethernet, 192.168.77.2, UDP:         │
│ stub at 0, Image 0x0020_8000,    │   │ 0x8E92 debugger: DDR at 0x4000_0000, │
│ DTB 0x01F0_0000                  │   │        control 0xC000_0000,          │
└──────────────────────────────────┘   │        a32_dbgif 0xC001_0000         │
                                       │ 0x8E94 console: the 16550's bytes    │
                                       └──────────────────────────────────────┘
```

`arm/cortex-a32-armv8-a/soc/arty/a32_arty.l8`, `soc/a32_soc.l8`.

Pipeline of the ARM A32:

```
┌──────────────────────────────────────────────┐   ┌──────────────────────────┐
│ F  fetch                                     │   │ divider: sequential,     │
│   I-cache 8 KB (VIPT); next pc: resume,      │   │ a bit a clock, holds     │
│   redirect from X, bx from E, branch from D  │   │ the pipe; /0 gives 0     │
└──────────────────────────────────────────────┘   └──────────────────────────┘
    │
    v
┌──────────────────────────────────────────────┐   ┌──────────────────────────┐
│ D  decode                                    │   │ ARMv7 MMU: 16-entry      │
│   decode tree, 3 read ports; b / bl resolved │   │ TLB, hardware walk on    │
│   HERE (1 bubble, no delay slot);            │   │ its own port             │
│   ldm / stm / long multiply re-issue the word │   └──────────────────────────┘
└──────────────────────────────────────────────┘
    │                                            ┌──────────────────────────┐
    v                                            │ banked r13 / r14 per     │
┌──────────────────────────────────────────────┐   │ mode; CPSR / SPSR       │
│ A  operands                                  │   └──────────────────────────┘
│   newest of E / M / X / W, written-back base, │
│   or the file; r15 reads pc + 8;             │
│   load-use: one stall                        │
└──────────────────────────────────────────────┘
    │
    v
┌──────────────────────────────────────────────┐
│ E  execute                                   │
│   condition check (a failing one becomes a   │
│   nop), shifter, ALU, 32x32 multiply;        │
│   address, D-cache access starts; bx here    │
└──────────────────────────────────────────────┘
    │
    v
┌──────────────────────────────────────────────┐
│ M  memory                                    │
```

```
  D-cache answers; interrupts attach here

   |
   v
X  exception / commit
  load aligned; rd + base (2 write ports);
  aborts, swi, undef, irq; a write to r15
  redirects (epoch flips)

   |
   v
W
  shadow stage: the last forwarding source
```

`arm/cortex-a32-armv8-a/armv8a.l8`: F and D, then the held pipe A E M X W.

serv (RISC-V), FreeRTOS (`systems/serv/top.bit`)

```
slow domain: 25 MHz (100 MHz / 4)

 serv: RV32I + Zicsr, bit-serial          AHB matrix (via serv_ahb)          IRQ
 32 steps an instruction, a bit a clock   RAM 64 KB   0x0000_0000 (the program)
 machine timer interrupt only             16550       0x4000_0000
                                          timer       0x8000_0000   machine timer

              |  the 16550's line, 97 656 bit/s
              v

 line_rx / line_tx: the bits back to bytes

                              |
                              v
board UART 115200 8N1 -> the Arty's USB: the console
only the USB cable: the program is the RAM's init
```

`riscv/serv/freertos/arty/serv_arty_dsp.l8`, `riscv/serv/freertos/serv_rtos_soc.l8`, `riscv/serv/serv.l8`.

Pipeline of the serv:

```
fetch  (1 + bus)                                   branch / jal / jalr:
  read the word at pc; an interrupt goes to trap   prep2 (1) + pass2 (32):
                                                   the target, a 2nd pass

   |
   v
load  (1)                                          shift: 1 + shamt clocks
  a = rs1 / pc, b = rs2 / imm (LUT register file)

   |
```

```
   v                                                    load / store: one bus
                                                        access after pass1
pass1  (32)
  a and b turn once, LSB first, through the
  one-bit ALU into r; pc + 4 into p4; flags

                                                        csr (1), trap (1)
   |
   v
wb  (1)
  rd <- r / p4 / flag; pc <- p4 or the target
```

Not a pipeline: one instruction at a time, a bit a clock -- about 35 clocks for an ALU instruction, 68 for a branch or jump ("some forty" on average). `riscv/serv/serv.l8`.

Cortex-M3 (ARMv7-M), FreeRTOS (`systems/m3/top.bit`)

```
slow domain: 25 MHz (100 MHz / 4)

 ARMv7-M, 3 stages (F D X)      AHB multilayer                      IRQ
 no caches; NVIC (16 lines),    RAM 64 KB   0x0000_0000 (the program)
 SysTick, SCB 0xE000_E000       16550       0x4000_0000              ext. 0
                                SysTick: the FreeRTOS tick

          |  the 16550's line, 97 656 bit/s
          v

 line_rx / line_tx: the bits back to bytes

                          |
                          v
board UART 115200 8N1 -> the Arty's USB: the console
only the USB cable: the program is the RAM's init
```

`arm/cortex-m3-armv7-m/soc/arty/m3_arty_dsp.l8`, `soc/m3_rom.l8`, `soc/m3_soc.l8`, `armv7m.l8`.

Pipeline of the Cortex-M3:

```
F  fetch                                          exception sequencer:
  32-bit word, no cache; a 32-bit Thumb           stacks 8 words, reads
  instruction across words: one extra clock       the vector, unstacks
                                                  on EXC_RETURN
   |
   v
D  decode                                         divider: a bit a
  m3_decode; the IT block's state advances        clock (32 + 1)
   |
   v
                                                  list walker: ldm / stm /
X  execute + memory + write-back                  push / pop, a register
```

```
│   register read (no forwarding: hazards hold),  │   │ an access                    │
│   shifter, ALU, multiply;                        │   └──────────────────────────────┘
│   a branch resolves here (~2 bubbles);           │
│   an access: three clocks, X waits               │
└──────────────────────────────────────────────────┘
```

`arm/cortex-m3-armv7-m/armv7m.l8`: three stages, one instruction each; the long parts sit below the pipe.

Contact

eiselekd at gmail.com