Bootstrapping a Libre, Self-Hosting RISC-V Computer

Gabriel L. Somlo, Ph.D.

CERT / SEI
Carnegie Mellon University
Pittsburgh, PA 15213

The Plan

- Build an FPGA based computer from Free / Libre software *and* gateware sources:
  - Linux, BusyBox
  - LiteX, RocketChip
- Using Free / Libre software *and* HDL toolchains:
  - gcc
  - yosys, trellis, nextpnr

You Can Follow Along

- On a pre-configured Fedora VM
  - http://mirror.ini.cmu.edu/litex/litexdemo.f32.ova
    - Built for VMWare (Fusion / Workstation), for convenience
    - Link availability *not* guaranteed beyond April 2021!
  - Login: *user*
  - Password: *tartans*
  - Pre-installed with toolchains, sources

Building the Development VM

- Kickstart a Fedora VM: litexdemo.f32.ks
- Log in as *user* (password: *tartans*), and run:

```
# git clone –recursive https://github.com/riscv/riscv-gnu-toolchain
# pushd riscv-gnu-toolchain; configure –prefix=$HOME/RISCV –enable-multilib
# make newlib linux; popd; rm -rf riscv-gnu-toolchain
# echo ’export PATH=$PATH:$HOME/RISCV/bin’ >> ~/.bashrc

# mkdir ~/LITEX; cd ~/LITEX
# git clone https://github.com/litex-hub/linux -b litex-rebase
# git clone https://github.com/litex-hub/linux-on-litex-rocket
# git clone https://github.com/riscv/riscv-pk
# wget https://raw.githubusercontent.com/enjoy-digital/litex/master/litex_setup.py
# python3 ./litex_setup.py init install –user
# wget https://busybox.net/downloads/busybox-1.31.0.tar.bz2 -O - | tar xfj -
```

Let’s start building!

- We’ll talk *theory* while waiting for build to finish!
- Start with the bitstream (it takes the longest):

```
# cd ~/LITEX
# litex_boards/litex_boards/targets/lambdaconcept_ecpix5.py --build \
      --cpu-type rocket --cpu-variant linuxd --sys-clk-freq 50e6 \
      --with-ethernet --with-sdcard
```

- Next, start building BusyBox:

```
# cp linux-on-litex-rocket/conf/busybox-*.config busybox*/.config
# (cd busybox*; make CROSS_COMPILE=riscv64-unknown-linux-gnu-)
```

Create the *initramfs.cpio* image

```
# mkdir initramfs; pushd initramfs; mkdir -p \
      bin sbin lib etc dev home proc sys tmp mnt nfs root usr/bin usr/sbin usr/lib

# cat > etc/inittab <<- ”EOT”
::sysinit:/bin/busybox mount -t proc proc /proc
::sysinit:/bin/busybox mount -t tmpfs tmpfs /tmp
::sysinit:/bin/busybox mount -t sysfs sysfs /sys
::sysinit:/bin/busybox --install -s
/dev/console::sysinit:-/bin/ash
EOT

# cp ../busybox*/busybox bin/; ln -s bin/busybox ./init

# fakeroot <<- ”EOT”
mknod dev/null c 1 3; mknod dev/zero c 1 5; mknod dev/tty c 5 0
mknod dev/console c 5 1; mknod dev/mmcblk0 b 179 0
mknod dev/mmcblk0p1 b 179 1; mknod dev/mmcblk0p2 b 179 2
find . | cpio -H newc -o > ../initramfs.cpio
EOT
# popd
```

Start building Linux

- Embed *initramfs.cpio* into the kernel image:

```
# cp initramfs.cpio linux/; pushd linux

# make ARCH=riscv CROSS_COMPILE=riscv64-uknown-linux-gnu- \
       litex_rocket_defconfig litex_rocket_initramfs.config

# make ARCH=riscv CROSS_COMPILE=riscv64-unknown-linux-gnu- -j3

# popd
```

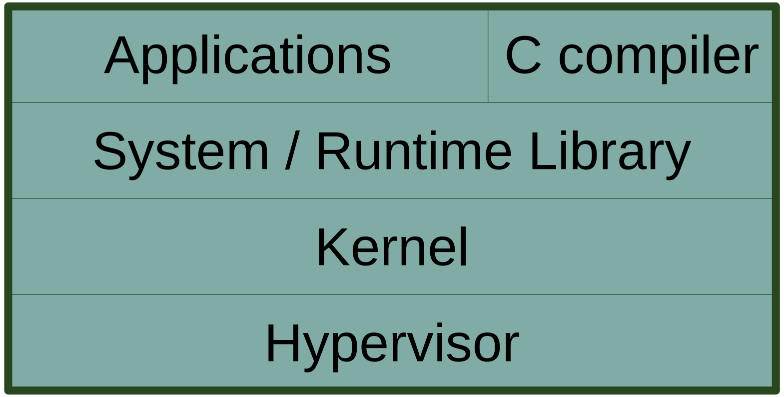
- Let's discuss the *bigger picture* while we wait…

Self-Hosting (Compiler)

C compiler

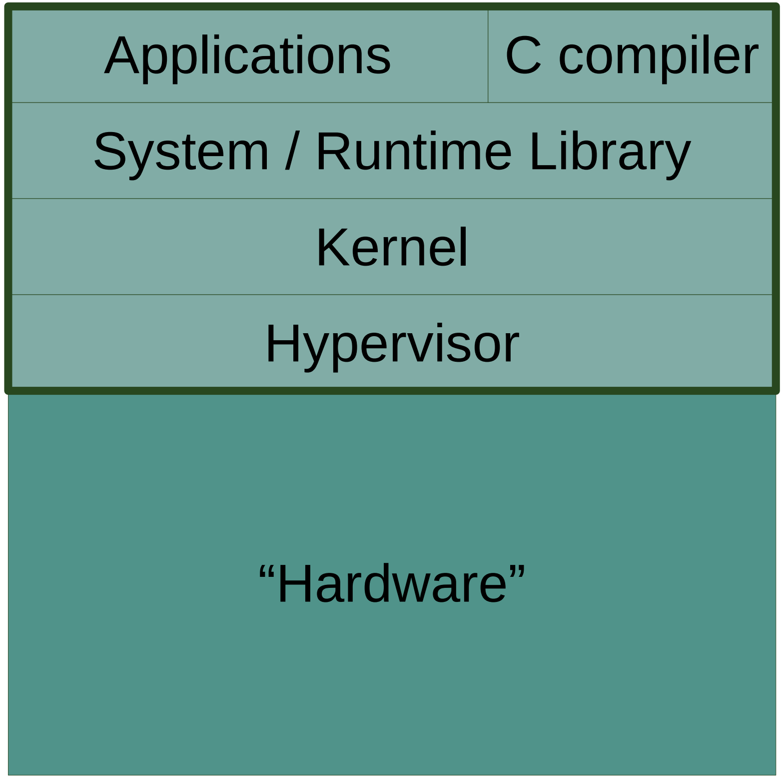
- Written in its own language
- Compiles its own sources
  - Subject to *Trusting Trust* attack (Ken Thompson)
    - Mitigated by *Diverse Double Compilation* (D. A. Wheeler)
- *Bootstrapping*

Self-Hosting Software Stack



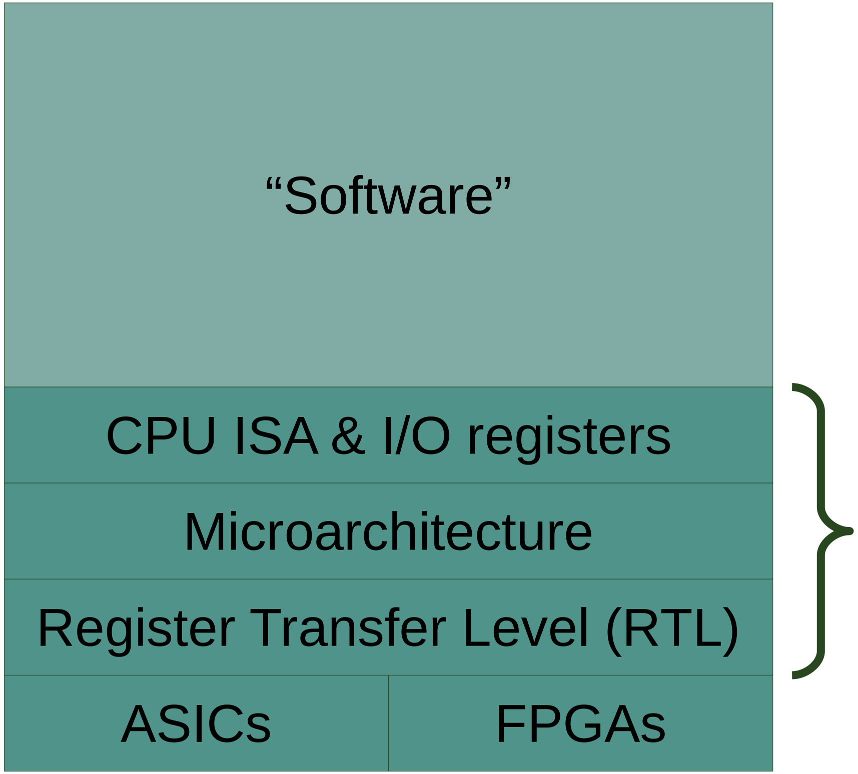
- Self-hosting compiler can build all software needed for its own execution
- Free / Libre sources for all components!

Self-Hosting Software Stack



- Self-hosting compiler can build all software needed for its own execution
- Relies on (deployed on top of) *Hardware*

More Details re. *Hardware*



- *Gateware* (written in HDL)
- *Physical* (i.e., silicon)

Self-Hosting Extended to Gateware

App.	HDL compiler	C compiler
System / Runtime Library		
Kernel		
Hypervisor		
CPU ISA & I/O registers		
Microarchitecture		
Register Transfer Level (RTL)		
ASICs	FPGAs	

- C compiler → *Software*
- HDL compiler → *Gateware*
- Free / Libre sources for all components!
- *Physical* (silicon, ASICs or FPGAs) out of scope!

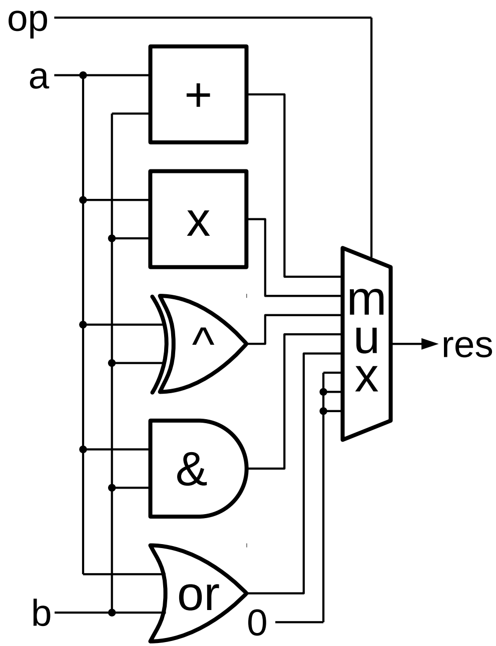
Gateware Compilation Stages

HDL Sources

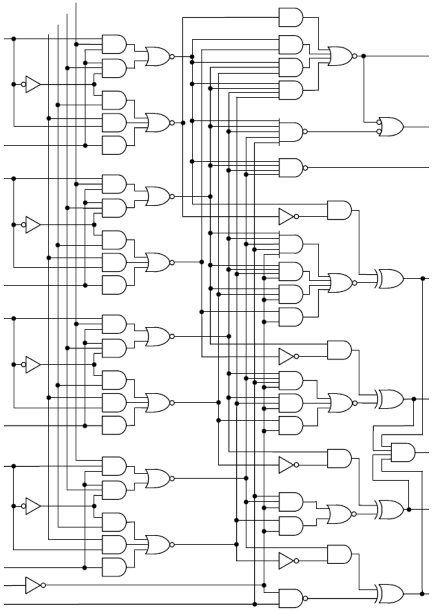
```
module alu_mod (
  // operator:
  input  alu_op_t     op,
  // operands:
  input  logic [31:0] a, b,
  // result:
  output logic [31:0] res);

  always_comb begin
    unique case (op)
      ALU_ADD: res = a + b;
      ALU_MUL: res = a * b;
      ALU_XOR: res = a ^ b;
      ALU_AND: res = a & b;
      ALU_OR : res = a | b;
      default: res = 32'b0;
    endcase
  end
endmodule: alu_mod
```

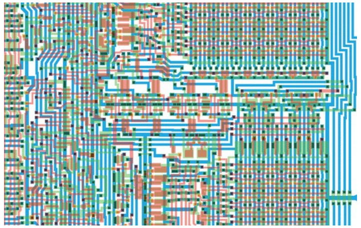
Elaboration



Synthesis,
Optimization

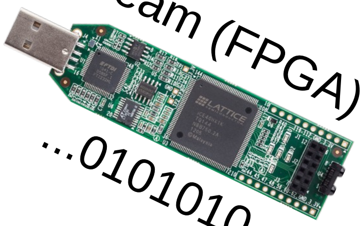


Tech. Mapping,
Place & Route



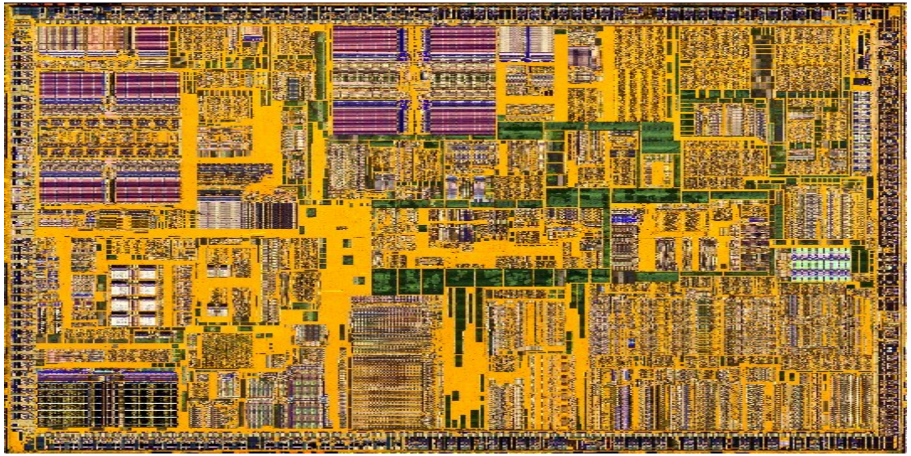
mask (ASIC)

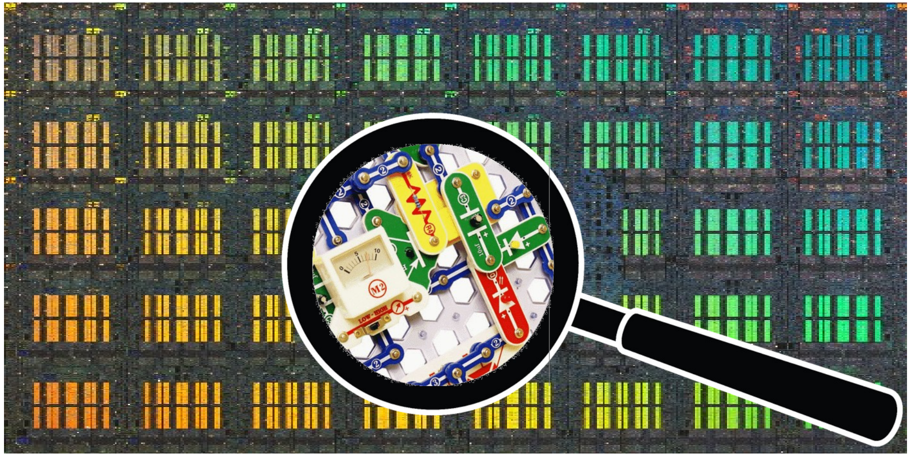
bitstream (FPGA)



...0101010...

ASICs vs. FPGAs





- Application Specific Integrated Circuit
- dedicated, optimized etched silicon
  - photolithographic masks
- *hard IP* cores

- Field Programmable Gate Array
- grid: programmable blocks, interconnect
  - bitstream
- *soft IP* cores

Hardware Attack Surface

- Fabrication (Malicious ASIC Foundry)
  - masks reverse engineered, modified to insert malicious behavior into ASIC
    - privilege escalation CPU backdoor (A2 Trojan)
    - tamper with silicon doping polarity (e.g., to weaken hardware-based crypto)
  - problematic to test / verify after the fact!
  - mitigated by using FPGAs: hard to predict where to add *useful* Trojan silicon!
- Compilation (Malicious HDL Toolchain)
  - generate *malicious* design from *clean* HDL sources
- Design Defects (accidental or intentional HDL bugs)
  - Spectre, Meltdown, etc.

Why *Self-Hosting* ?

- Freedom!   Liberty!   Independence!   :)
  - From *black-box*, and/or *non-Libre* dependencies
- Trust a running *software + gateware stack* to the same extent as its *cumulative sources*
  - Gateware HDL sources
  - Software sources (including C and HDL compilers)

Bootstrap Software+Gateware Stack

- *Host* (x86_64/Linux):
  - Build clean C (cross-)compiler
  - Build clean HDL compiler (for both x86_64 and rv64gc)
  - Cross-compile target (rv64gc) software stack
  - Build gateware (FPGA bitstream) for target system
- *Target* (rv64gc/Linux):
  - Program FPGA board with gateware/bitstream
  - Boot into target software stack
    - Self-hosting from this point forward!
  - Natively rebuild gateware bitstream, software stack, from sources, as needed

LiteX + Rocket SoC Block Diagram

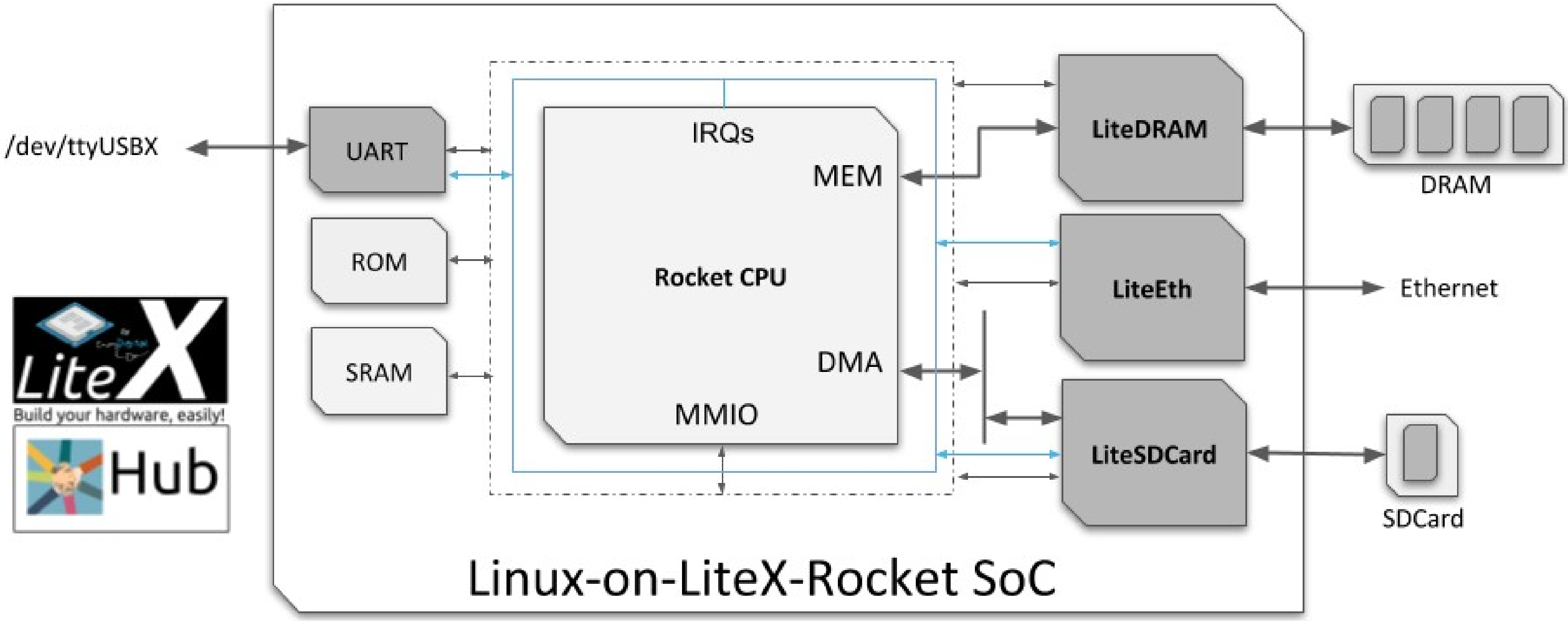


Image credit: Florent Kermarrec

Finish building *boot.bin* image

- Embed kernel into boot.bin (BBL):

```
# mkdir riscv-pk/build; pushd riscv-pk/build

# ../configure host=riscv64-unknown-linux-gnu --enable-logo --with-arch=rv64imac
               --with-payload=../../linux/vmlinux
               --with-dts=../../linux-on-litex-rocket/conf/ecpix5.dts

# make bbl

# riscv64-uknown-linux-gnu-objcopy -O binary bbl ../../boot.bin

# popd
```

- Make *boot.bin* available via TFTP, or copy to 1st MSDOS / FAT16 primary partition of SDCard

Boot Linux on LiteX+Rocket SoC

- Connect board via USB, and start a console:

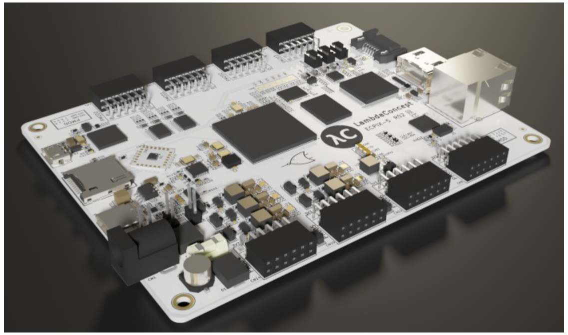
```
# screen dev/ttyUSB1 115200
```

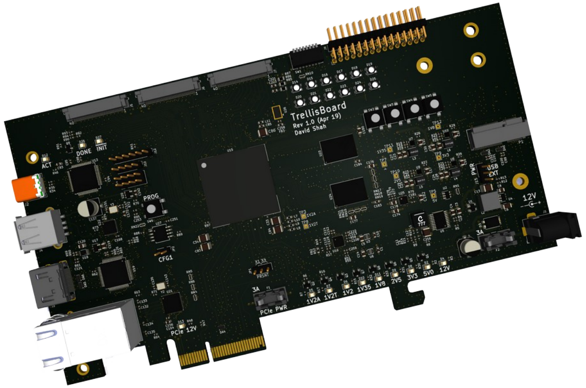
- Program the board with the compiled bitstream:

```
# openocd -f litex-boards/litex_boards/prog/openocd_ecpix5.cfg \
      -c ’transport select jtag; init;
          svf build/lambdaconcept_ecpix5/gateware/lambdaconcept_ecpix5.svf; exit’
```

- LiteX loads *boot.bin*, Linux boots into BusyBox

Try it on your own FPGA board!





ECPIX-5 trellisboard ecp5-5g-versa

Demo, then Q&A

- For up-to-date build steps, see: https://github.com/litex-hub/linux-on-litex-rocket

- Thank You!

Backup Material

Compilers, Trusting Trust, and DDC

- Ken Thompson's self-propagating C compiler hack
  - malicious compiler inserts Trojan during compilation of *victim program*
    - clean sources → malicious binary (incl. *compiler's own sources*!)
    - compiler source hack *no longer needed* beyond 1st iteration!
- David A. Wheeler's mitigation: Diverse Double Compilation (DDC)
  - suspect compiler A: sources S_A, binary B_A
  - trusted compiler T: binary B_T

$S_A \rightarrow B_A \rightarrow X$ $\qquad$ $S_A \rightarrow B_T \rightarrow Y$

  - X and Y are *functionally identical*, but *different binaries*

$S_A \rightarrow X \rightarrow X_1$ $\qquad$ $S_A \rightarrow Y \rightarrow Y_1$

  - X_1 and Y_1 must be *identical binaries* (output of two *functionally identical* compilers)!