



Aphrodite

*Security
Properties of
RISC-V*

Juni L DeYoung
CS/DS Tea
Willamette University
2022-11-10 11:30 PST



01



Table of contents

01

Overview

Computer architecture,
instruction sets, and
emulation

02

Emulating RISC-V

Data generation is a
complicated profession

03

Aphrodite

Design choices and the
engineering process

04

The REU Experience

What is it like doing CS
research over the summer?



01

Overview

03

1.1

Goals

What are we doing and why are we doing it?

**What are
security-relevant
properties of
computer hardware?**

Research Process

Collect

1. Model processor in software
2. Record register transfers

Analyze

- 3 . Mine traces for properties
4. Check properties against common weaknesses

Report

- 5 . Security properties found!



QEMU



fedora



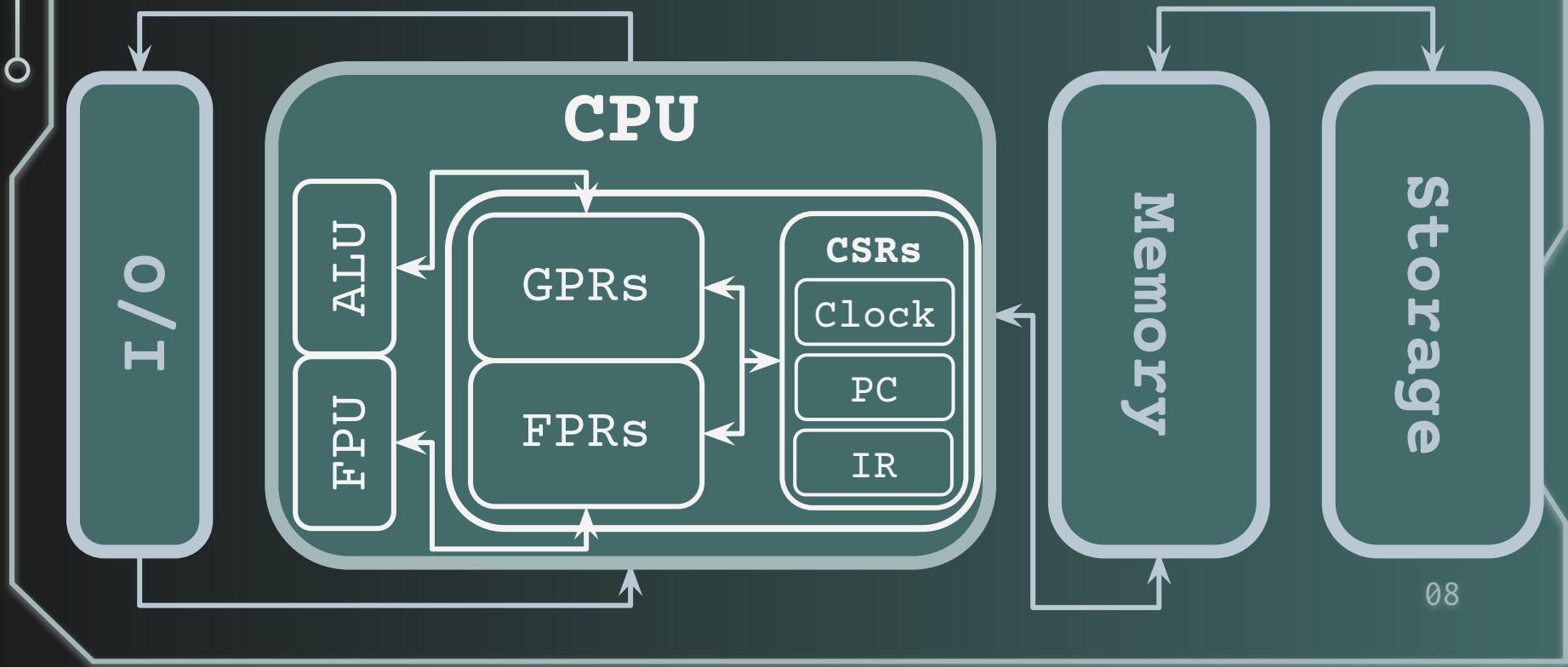
06

1.2

Background

What exactly are we studying here?

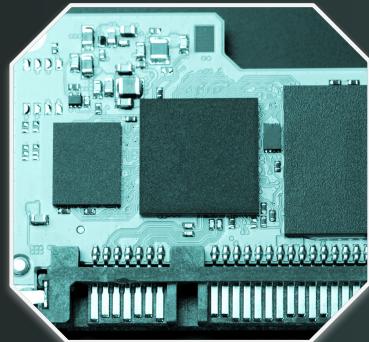
Computer Anatomy



Virtualizing Hardware

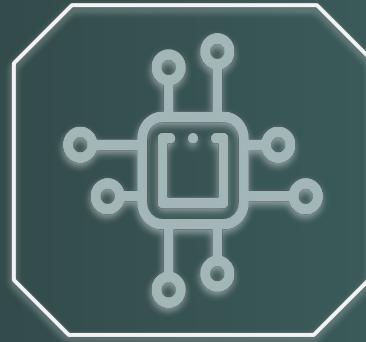
Simulation

- Recreates a processor at register transfer level (RTL)
 - Modeling the actual configuration of wires and transistors in software



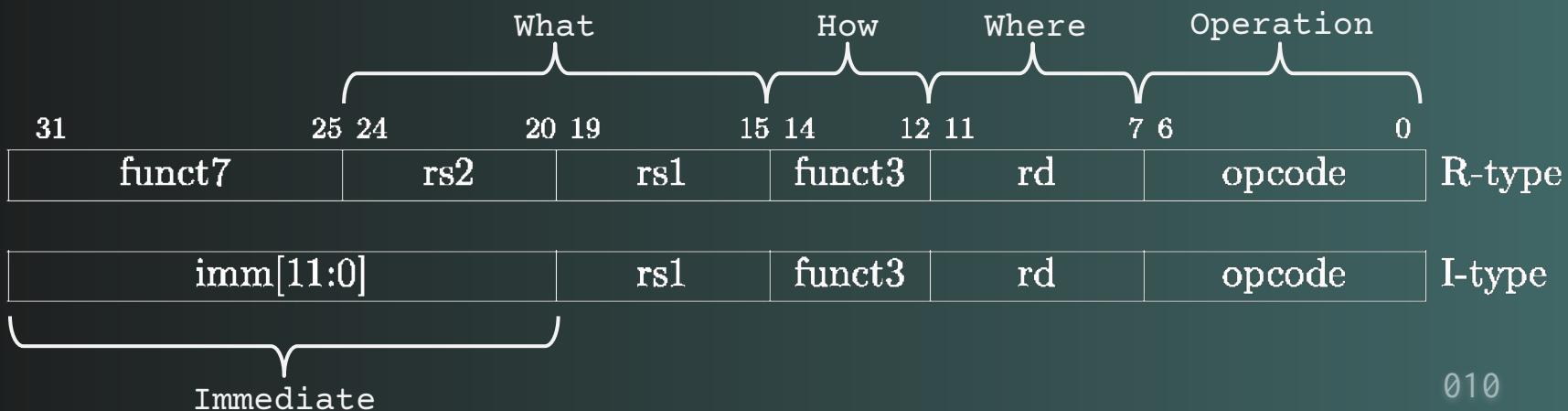
Emulation

- Recreates an instruction-set architecture (ISA)
 - Doesn't replicate specific hardware idiosyncrasies, only its instruction set



Instructions

- Contained in memory
 - Addresses correspond to values in the program counter
- Control information flow through the processor
 - Performing operations (arithmetic, load/store, navigation)



ISA Paradigms

RISC

- One operation per instruction
- “Load-Store” architecture
- More difficult to write programs in assembly
- ARM



CISC

- “Microcoding”
- Instructions execute multiple operations at once
- Smaller programs
- Fewer main memory accesses
- x86



Why Study RISC?

- CISC processors are proprietary trade secrets
- RISC architectures are easier to study
 - Fixed-length instructions
 - One instruction -> one operation
- RISC-V is an open-source design
 - Funded by Intel and AMD





02

RISC-V

Emulation is the highest form of
flattery

013

The RISC-V Spec

- Highly customizable to different configurations
- Designed for academic study **and** hardware implementation
- 32- and 64-bit variants

General Purpose Registers x0-x31

- x0 is fixed to value 0
- x1-x31 are read as booleans or (un)signed 2's complement integers

Floating-point registers f0-f31

- Correspond to IEEE standard for floating-point

Control and Status Registers

- 4096 CSRs, mostly used by the privileged architecture
 - Some use in unprivileged code, mostly as counters and timers
 - Exceptions, interrupts, traps, control transfer

Configuring Qemu

1. [Download Qemu](#)
2. [Build RISC-V emulator](#)
 - a. `$ sudo apt install qemu-system-misc`

This includes the `qemu-system-riscv64` and `riscv32` commands, which allows Qemu to boot executable files with the RISC-V virt emulator. It also includes several additional emulators.



The RISC-V Toolchain

In:

```
$ git clone https://github.com/riscv/riscv-gnu-toolchain --recursive  
$ sudo apt-get install autoconf automake autotools-dev curl python3 [...]  
$ ./configure --prefix=/opt/riscv --enable-multilib  
$ sudo make linux
```

[A few hours pass]

Out:

```
[...]  
gcc: error: unrecognized argument in option '-mcmode=medany'  
gcc: note: valid arguments to '-mcmode=' are: 32 kernel large medium small  
make: *** [Makefile:319: file.o] Error 1
```

“Hello World”

```
.global _start      Initialize the program at “_start” label

_start:

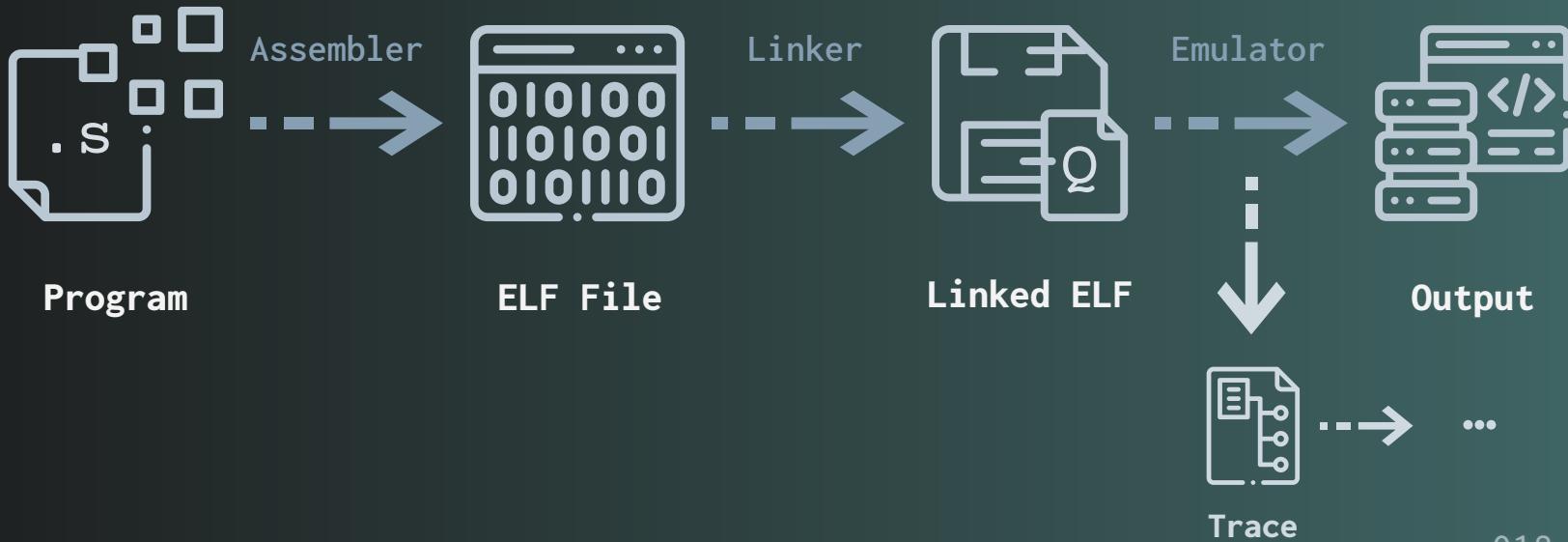
    lui t0, 0x10000    Load address of serial port into register t0

    andi t1, t1, 0      Zero out t1
    addi t1, t1, 72     Add ord("H") = 72 to t1
    sw t1, 0(t0)        Send value of t1 == 'H' to location addressed by t0 (UART0)

    [...]
                    The previous three lines are repeated for 'e', 'l', 'l', 'o'
                    and finally LF (line feed, aka '\n')

finish:
    beq t1, t1, finish  Jump to label finish if t1==t1
```

Bare-Metal Programs on RISC-V



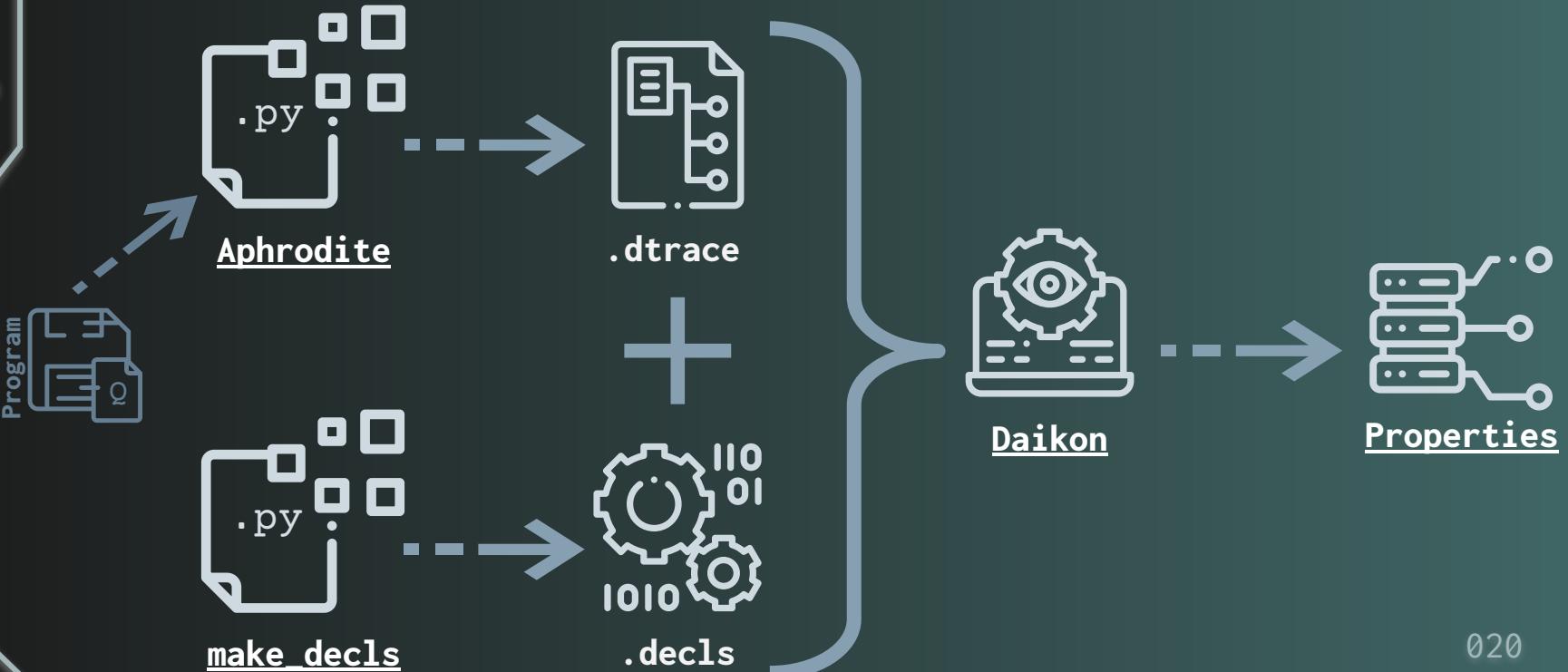
Booting Fedora

After downloading [the Fedora prebuilt images](#), decompress and boot according to the [Qemu documentation](#).

```
fedora-riscv login: root
Password:
Last failed login: Mon Jul 11 19:17:36 EDT 2022 on ttys0
There were 3 failed login attempts since the last successful login.
[root@fedora-riscv ~]# ls
anaconda-ks.cfg
[root@fedora-riscv ~]# mkdir jldey
[root@fedora-riscv ~]# cd jldey
[root@fedora-riscv jldey]# ls
[root@fedora-riscv jldey]# echo "Hello World!"
Hello World!
[root@fedora-riscv jldey]# echo $PATH
/root/.local/bin:/root/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:
[root@fedora-riscv jldey]# echo $PATH > path.txt
[root@fedora-riscv jldey]# ls
path.txt
[root@fedora-riscv jldey]# cat path.txt
/root/.local/bin:/root/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:
```

Above: a sample session in the Fedora emulation

Data Mining





03

Aphrodite

Now how do we *do* all that?

Gathering Register Values

The Trick: collecting register values without changing any of them.

Tools:

- riscv-probe (CSRs)
- Qemu debugging tools (GPRs)
 - Logging (-d cpu)
 - Qemu built in “trace”
 - Monitor -> info registers
 - GDB (GNU debugger)
 - Multiarch
 - riscv64_gdb
 - Qemu source (fprintf hacking)
 - **Qemu wrapper to inject commands to monitor and write output to file**
 - subprocess library
 - pexpect library

qscript (pseudocode)

1. Start QEMU with a linked ELF as input
 - start the VM paused (`-S`)
 - `'-monitor stdio` so program can write commands to monitor
2. Ping monitor every so often (specify as commandline option?)
 - build a simple character driver to use instead of `stdio`?
 - write this output to a trace file
 - QEMU “single-step” mode (take the first N cycles)
3. Terminate VM
 - send ‘quit’ command (or simply ‘q’) to monitor
 - quit condition?
 - timeout
 - > fixed time?
 - > based on last output change (pc?)
 - user-specified?
 - > if the program is reading/writing to the monitor console, does that mean the user can issue a ‘quit’?
 - > does the user ping the script, or the monitor?

Trace formats

qtrace

```
i\x1b[K\x1b[Din\x1b[K[\...]  
pc      0000000000001000\r  
mhartid 0000000000000000\r  
[...]  
x0/zero 0000000000000000  
x1/ra   0000000000000000  
x2/sp   0000000000000000  
x3/gp   0000000000000000\r  
[...]  
f28/ft8 0000000000000000  
f29/ft9 0000000000000000  
f30/ft10 0000000000000000  
f31/ft11 0000000000000000\r  
[...]
```

.dtrace

```
.tick():::ENTER  
this_invocation_nonce  
1  
pc  
4096  
1  
mhartid  
0  
1  
[...]  
f31/ft11  
0  
1
```

Parsing qtrace to dtrace

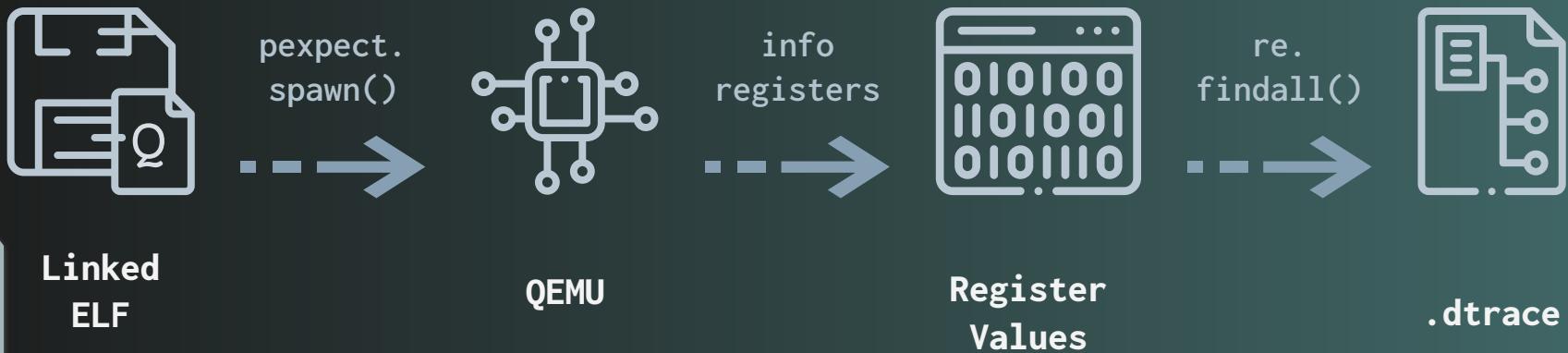
1. Parse register values into a list
 - a. QEMU logs don't parse each timestep neatly (is this a reason not to use them?)
 - b. Monitor output (qtraces) can parse each timestep
 - i. Is there a potential for duplicate data?
 - ii. I can parse to Daikon format at runtime and only write to file once
 - iii. qtraces contain FPR values.
2. Add list generated in (1.) to a 2D list of all timesteps
 - a. Get rid of any empty sublists (or completely ignore identical data)
3. Parse this 2D list into Daikon .dtrace format and write to file

Conclusion: pexpect monitor traces are a better solution than QEMU native debugging.

Parsing qtrace to dtrace

```
99      # grab `info registers` output
100     out = qemu.before
101     #print(out)
102
103     # find all register name/value pairs on current line
104     # returns empty list if no register values found,
105     # i.e. the output was not a string of register/value pairs
106     vals = re.findall(r"[a-zA-Z0-9/]+\s+[0-9a-f]{16}|\w+\s+[0-9a-f]x[0-9a-f]",out)
```

Aphrodite.py





04

The REU Experience

Faff around. Find out. Get paid.

Whiteboard Notes (overall checklist)

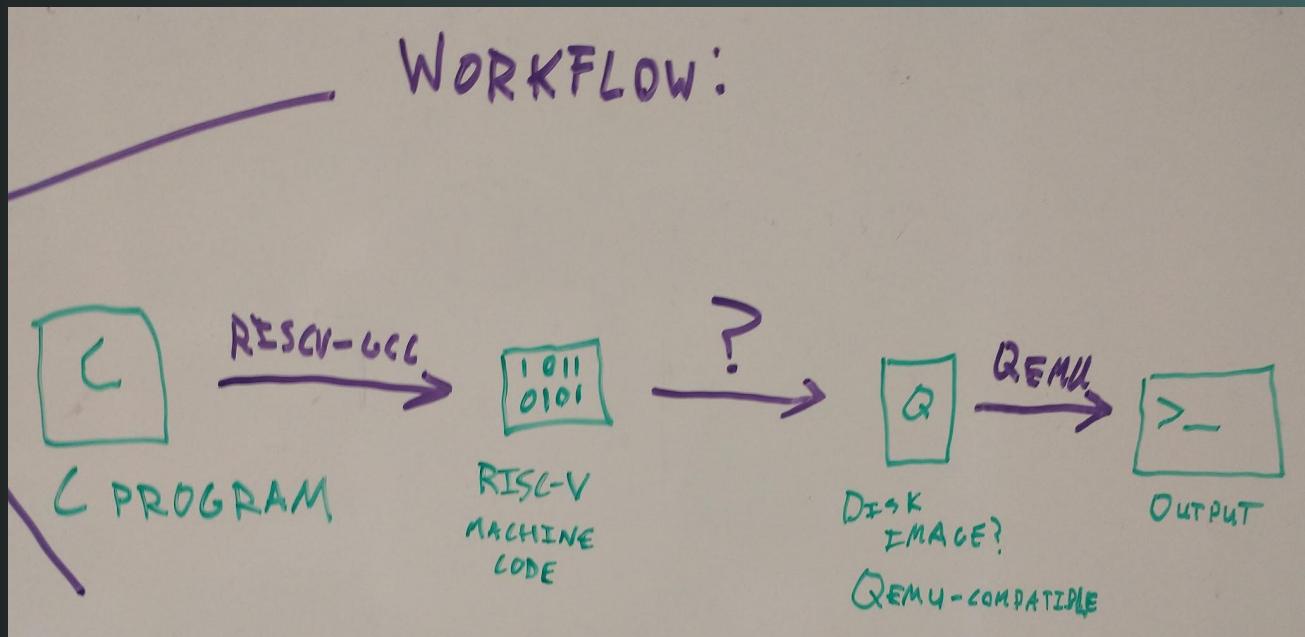
- BOOT LINUX ON RISCV-VIRT EMULATION
- COMPILE + RUN A BARE-METAL C PROGRAM (OR ASM!)
 - GET A COMPILER WORKING
 - CLONE
 - BUILD → MULTILIB SUPPORT (RV32 & RV64)
 - TGST (COMPILE!)
 - RUN THE PROGRAM
- GET TRACES OF RISCV EMULATION
 - PRINT REGISTER VALUES THRU QEMU (-D)
 - WRAPPER SCRIPT W/ PEXPECT
 - DTRACE → DTRACE
 - DTRACE → DECLS?
- OVERALL SCRIPT

- BOOT LINUX ON SIFTIVE (W/ LOGGING)
 - ARE THERE "EXTRA" CSRs NOT ON VIRT?
 - NOT IN THE DEBUGGING LOG
 - MONITOR?

1. PARSE REG. VALS INTO LIST (PER Timestep)
 - LOGS DON'T PARSE TIMESTEPS NEATLY
 - DTRACES DO.
 - ← POTENTIAL FOR DUPLICATE DATA?
 - WHY NOT TO USE THEM?
2. JOIN TIMESTEPS INTO 2D LIST
 - GET RID OF EMPTY SUBLISTS
 - BUT,
RUNTIME PARING TO DTRACE + FPR ACCESS
3. 2D LIST → DTRACE

PEXPECT MONITOR TRACES ARE BETTER.

Flowchart Draft (slide 18)



Whiteboard Notes (slide 25)

1. PARSE REG. VALS INTO LIST (PER Timestep)

- LOGS DON'T PARSE TIMESTEPS NEATLY

- QTRACES DO. ← POTENTIAL FOR
DUPLICATE DATA?

2. JOIN TIMESTEPS INTO 2D LIST
→ GET RID OF EMPTY SUBLISTS

WHY NOT
TO USE THEM?

3. 2D LIST → DTRACE

BUT,
RUNTIME
PARSING TO DTRACE
+
FPR ACCESS

P EXPECT MONITOR TRACES ARE BETTER.



Questions?

jldeyoung@willamette.edu
willamette.edu/~jldeyoung
github.com/wu-jldeyoung

CREDITS: This presentation template was created by **Slidesgo**, and includes icons by **Flaticon**, and infographics & images by **Freepik**

Please keep this slide for attribution