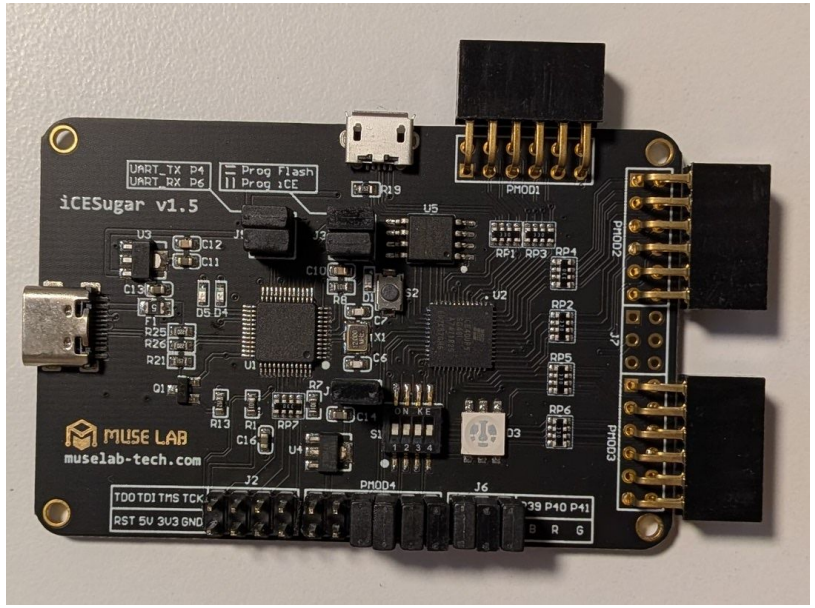
Building a virtio-serial FPGA device

Stefan Hajnoczi
stefanha@redhat.com

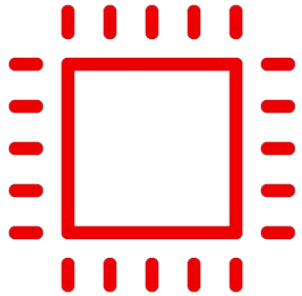
Red Hat

What we'll discuss today



- Background on VIRTIO hardware devices
- The FPGA development process
- Implementing the virtio-mmio transport
- UART transceiver implementation
- Options for more advanced devices & ASICs

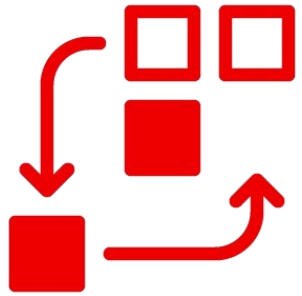
Why hardware VIRTIO devices?



Offload & accelerate



Isolation for security and management



Reuse existing drivers

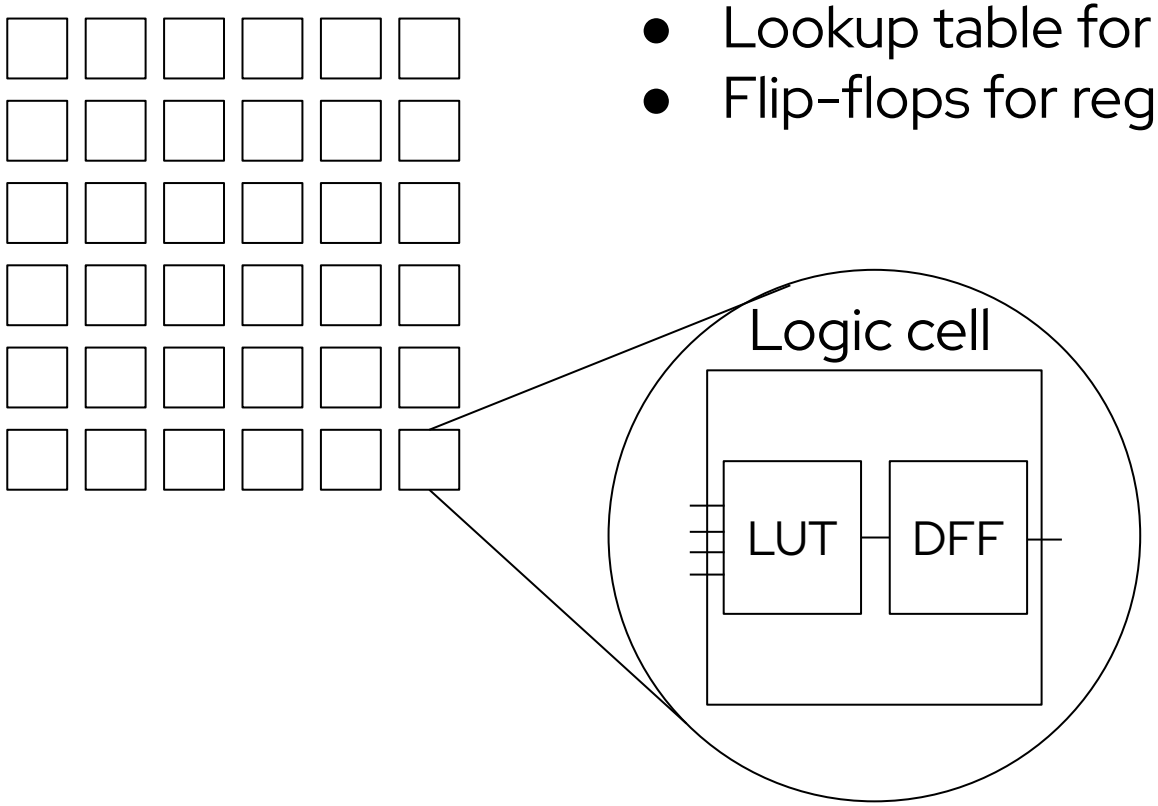
Software vs hardware VIRTIO device implementation

- Early VIRTIO versions were not implementable in hardware
  - But they were close: already realized as PCI or MMIO devices
- **VIRTIO gained features for hardware implementation**
  - VIRTIO_F_ACCESS_PLATFORM (IOMMU)
  - VIRTIO_F_ORDER_PLATFORM (DMA memory barriers)
  - VIRTIO_F_NOTIFICATION_DATA (reduces DMA xfers)
- Today a number of hardware implementations exist, largely DPUs for servers



FPGA overview

Grid of logic cells



Stateful computation via logic cells consisting of:

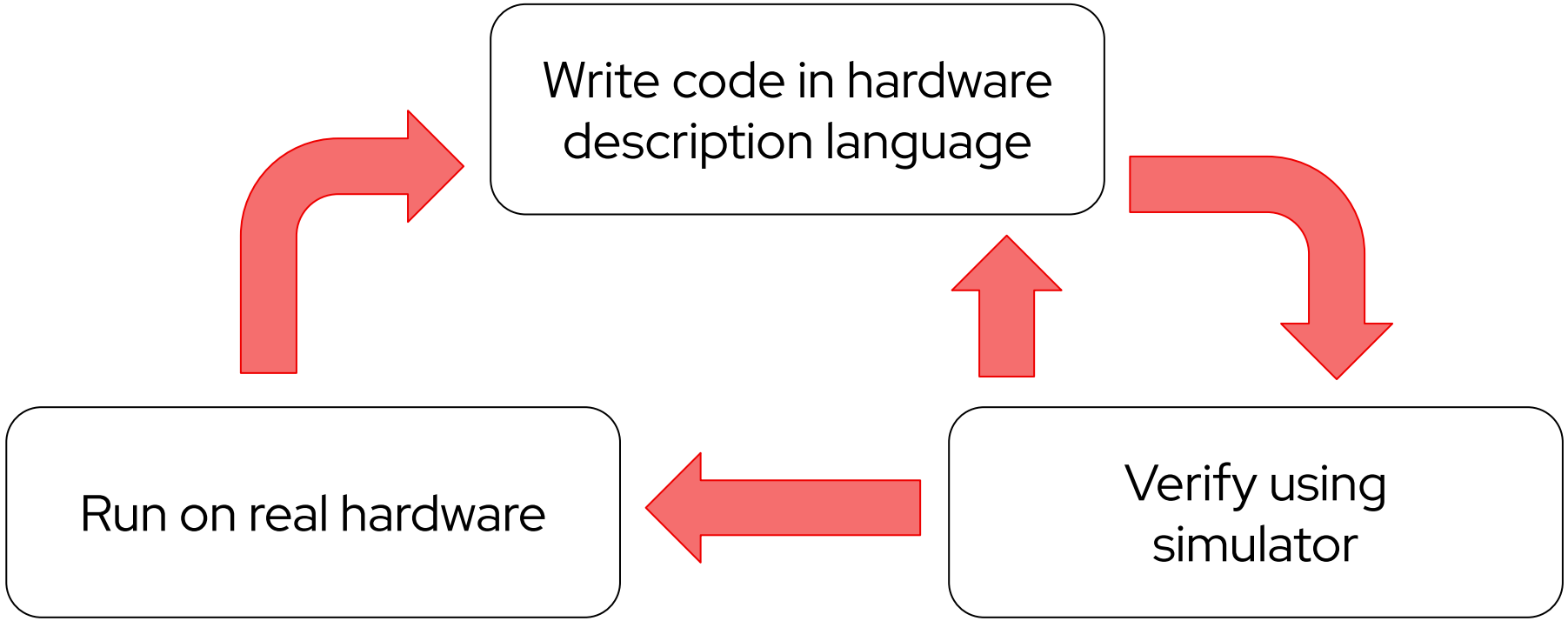
- Lookup table for digital logic (computation)
- Flip-flops for registers (state)

Programmable and can be loaded from EEPROM

Augmented with clocks, accelerators, and I/O “hard blocks” for interfacing with the outside world



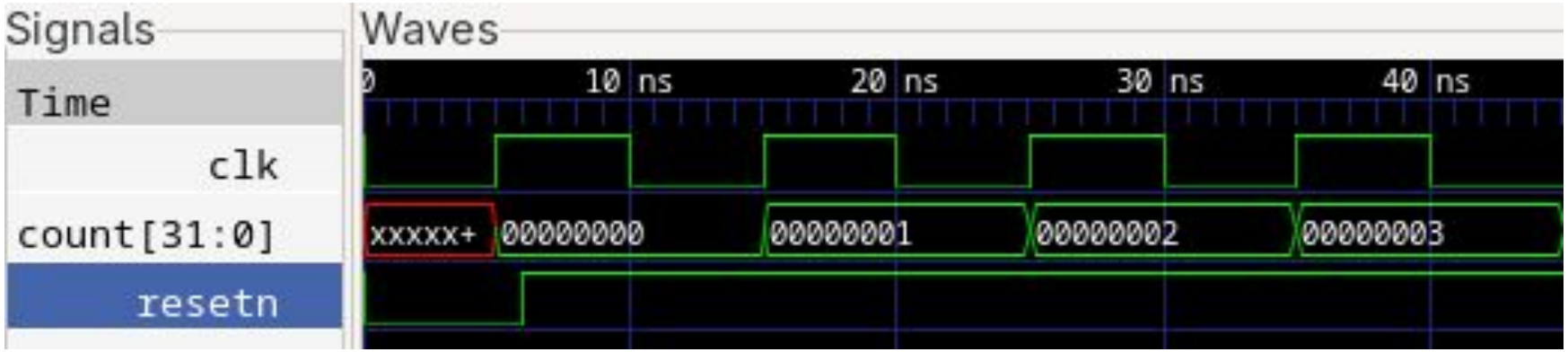
FPGA development process



Example Verilog module

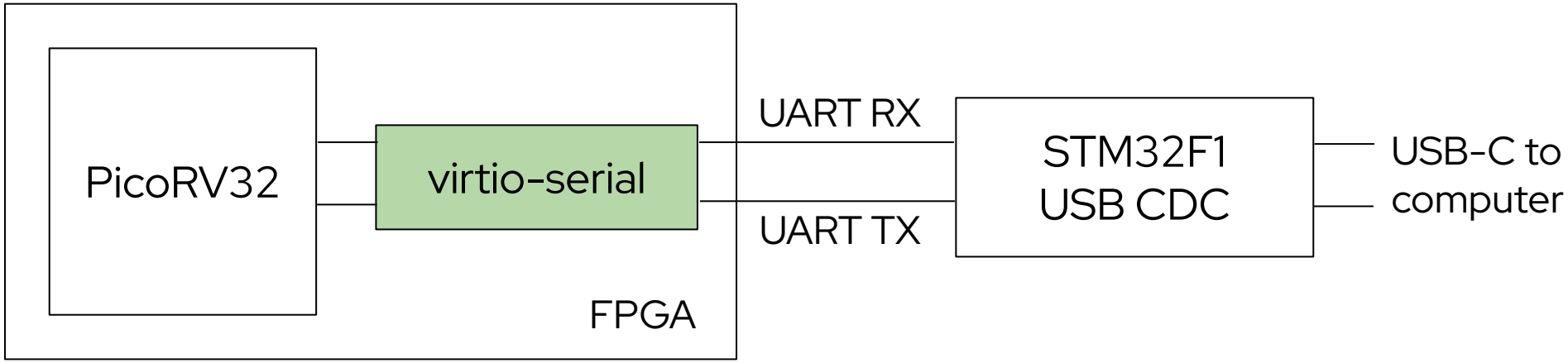
```
module counter (input clk, resetn, output reg [31:0] count);

    always @(posedge clk) begin
        if (!resetn)
            count <= 0;
        else
            count <= count + 1;
    end
endmodule
```





A virtio-serial device for PicoRV32 RISC-V core



iCESugar dev board with iCE50UP5K FPGA:

- **Open source toolchain** (Yosys, nextpnr, Icarus Verilog)
- Fits PicoRV32 open source **RISC-V soft core**
- PicoRV32 bus is **extensible with your own MMIO devices**
- STM32F1 USB CDC serial bridge <-> FPGA UART pins



PicoRV32 bus

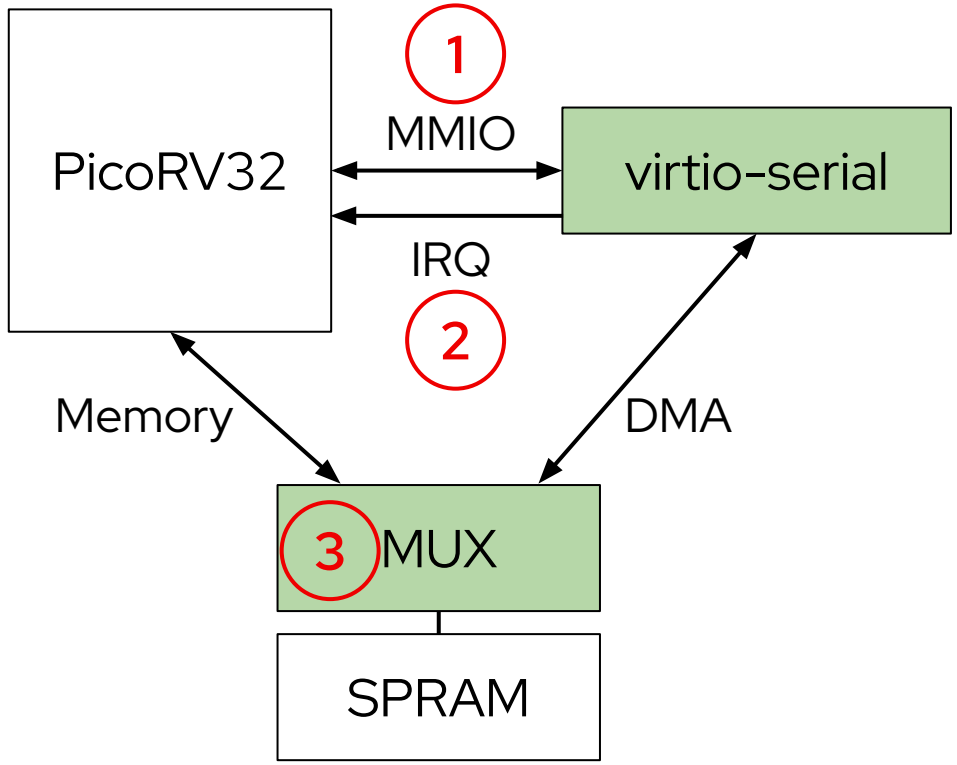
```
output          mem_valid  // request a memory transfer
output          mem_instr  // hint that CPU is fetching an instruction
input           mem_ready  // reply to memory transfer

output [31:0] mem_addr   // address
output [31:0] mem_wdata  // data being written
output [ 3:0] mem_wstrb  // 0000 - read
                         // 0001 - write 1 byte
                         // 0011 - write 2 bytes
                         // 0111 - write 3 bytes
                         // 1111 - write 4 bytes

input  [31:0] mem_rdata  // data being read
```



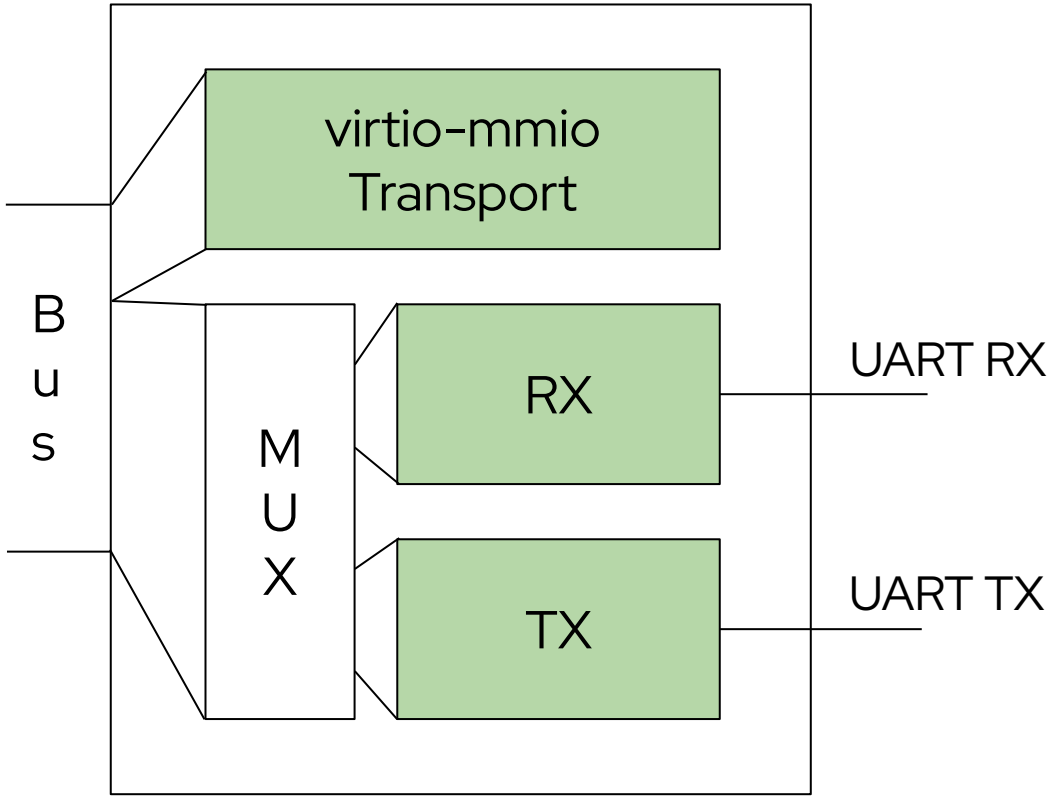

Integrating an MMIO device with the CPU



1) Read/write MMIO **registers**
   - Decode address, handle register, and signal ready
2) **Interrupts** to signal completion
   - Cut out to reduce logic cell footprint, VIRTIO supports polling
3) **DMA** transfers to access RAM (vrings and buffers)
   - SPRAM multiplexer arbitrates CPU and device memory transfers



Device design



- Virtio-mmio is the control plane
- RX/TX are the I/O plane
- RX/TX access virtqueues via DMA, so share the memory bus via a multiplexer (arbiter)



virtio-mmio transport

```
case (iomem_addr)
`REG_DEVICE_FEATURES: begin
    ready <= 1;
    rdata <= device_features_sel == 1 ?
             1 /* VIRTIO_F_VERSION_1 */
             …;
end
`REG_DEVICE_FEATURES_SEL: begin
    ready <= 1;
    if (iomem_wstrb == 4'hf) begin
        device_features_sel <= iomem_wdata;
    end
end
…
```

Straightforward register handler code

Implements VIRTIO MMIO Transport

Configures RX/TX units so they can process virtqueues



Reader and writer interfaces

Reader:

```
input phase,           <- toggle to initiate read
output [31:0] data,    <- bytes read
output [2:0] data_len,<- number of bytes (1-4)
output ready,          <- ready for next phase?
```

Writer:

```
input phase,           <- toggle to initiate write
input [31:0] data,     <- bytes to write
input [2:0] data_len,  <- number of bytes (1-4)
output ready,          <- ready for next phase?
```

RX/TX units are made up of:

- A reader
- A writer

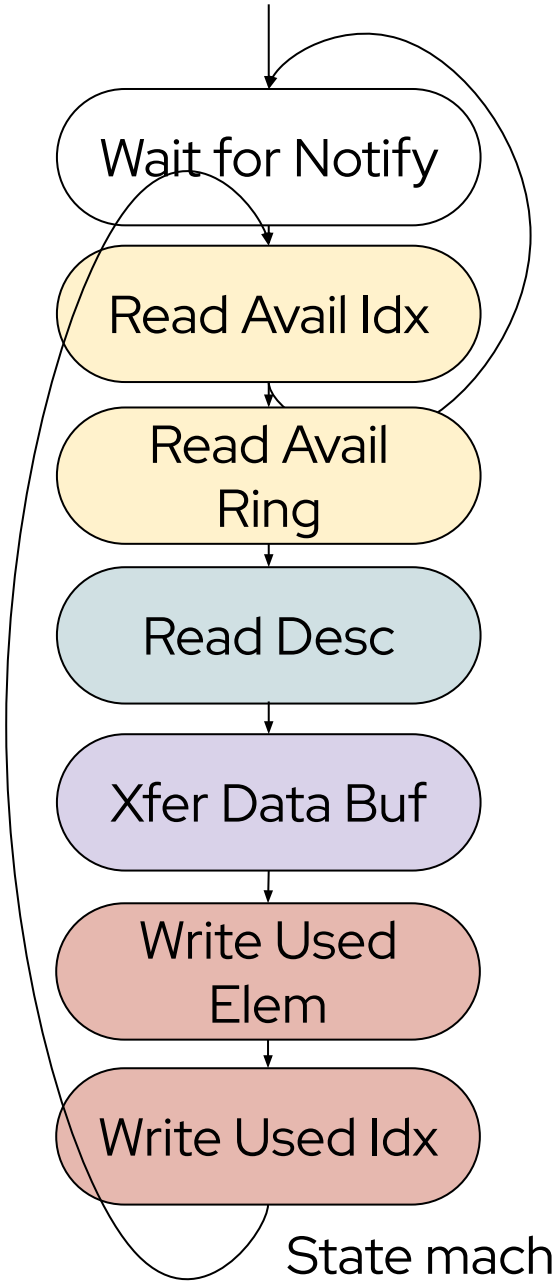
Data is streamed from reader to writer

RX: uart_reader + virtqueue_writer
TX: virtqueue_reader + uart_writer

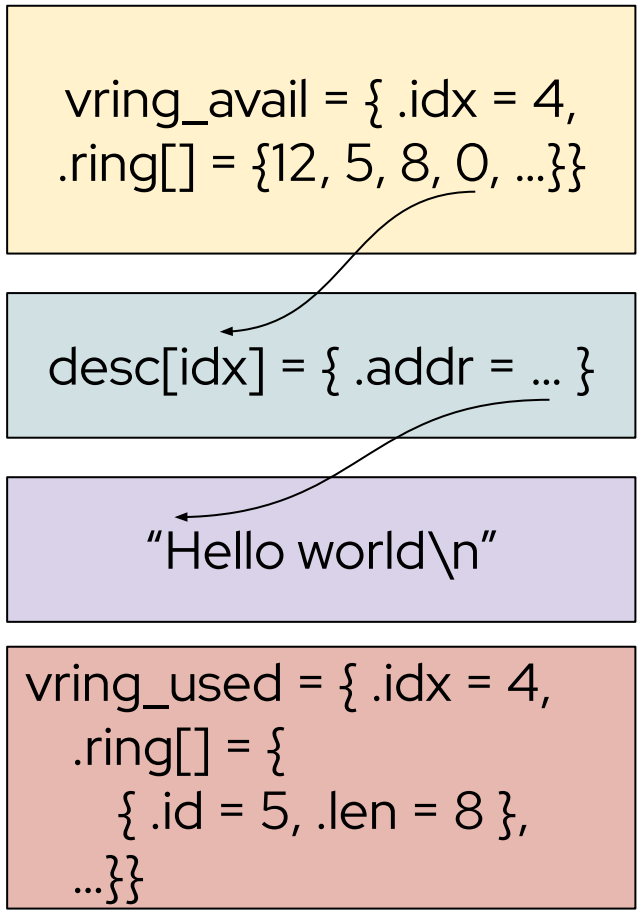
Loopback also possible for testing



Virtqueue processing



State machine



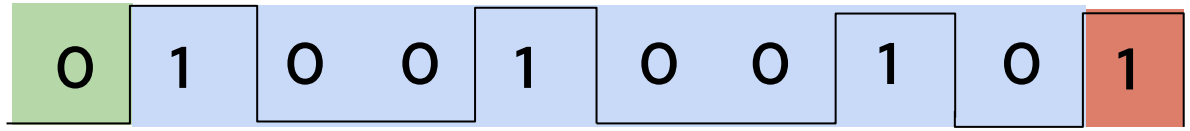
vring data structure

State machine that issues DMA transfers and parses vring.

Triggered by Available Buffer Notification.



UART transceiver



8N1 encoding (8 data bits, no parity bit, 1 stop bit)

- **Start bit (0)** indicates start of a byte
- **Stop bit (1)** indicates end of a byte and stays 1 during idle
- Framing bits allow asynchronous receiver to synchronize from just the data
- Transmitter and receiver must be configured with the same baud rate



Writing the PicoRV32 firmware

Simple firmware for PicoRV32 using existing Rust crates:

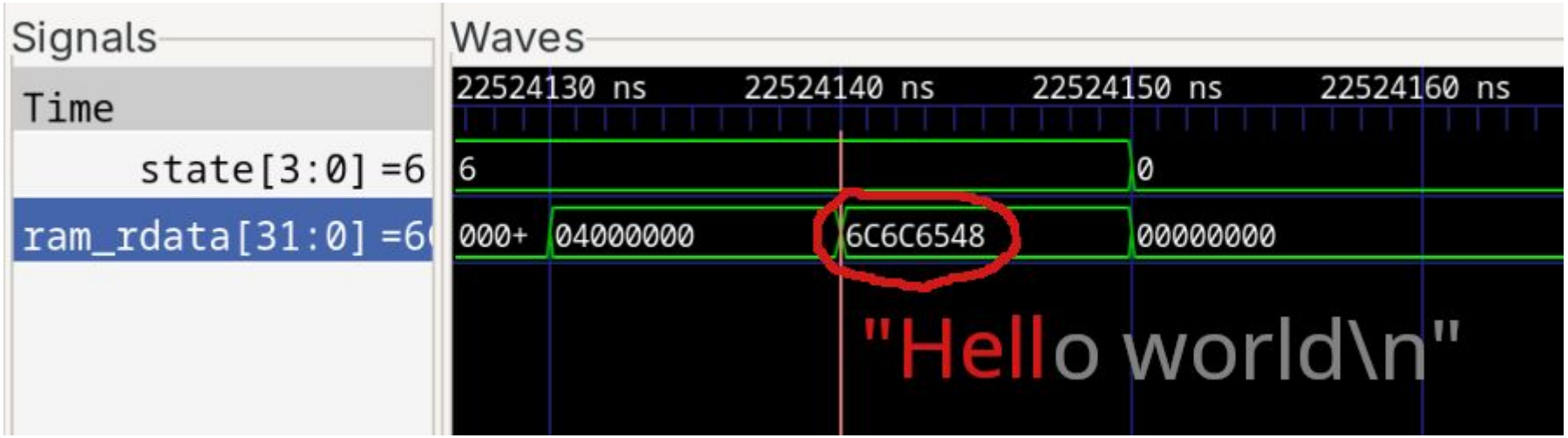
- riscv-rt
- embedded-alloc
- virtio-drivers (has a poll-mode virtio-serial driver!)

Firmware is mostly glue code and echoes characters from RX to TX

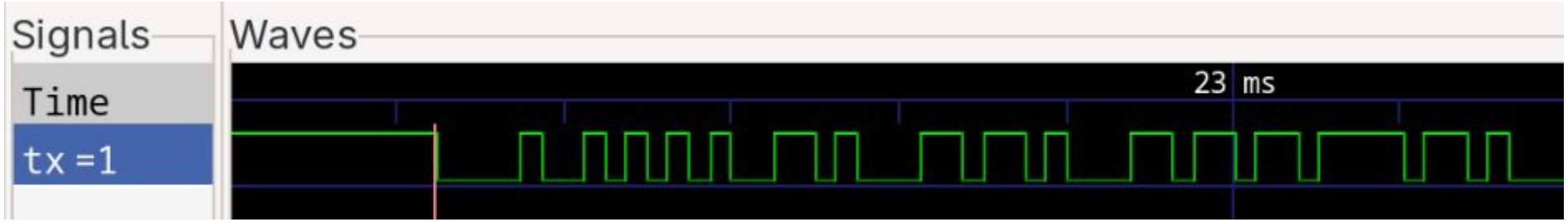
Minor tweaks necessary to make riscv-rt and PicoRV32 work together (missing some RISC-V registers and FENCE instructions)



Transmitting “Hello world”



Virtqueue_reader DMA reads data buffer



UART TX transmission



Alternatives to explore

- Run parts of device in software, use FPGA for bus interfacing or fast path
- Linux PCI Endpoint Framework (https://www.kernel.org/doc/html/latest/PCI/endpoint/index.html) - software implementation, PCI device interface
- ASIC - Tiny Tapeout (https://tinytapeout.com/) shuttle program
- DPU - See dpio-af KVM Forum presentation



Where to find out more

Source code



https://gitlab.com/stefanha/virtio-serial-fpga

Blog series



https://blog.vmsplice.net/2026/01/building-virtio-serial-fpga-device-part.html



Thank you

Red Hat is the world's leading provider of enterprise open source software solutions. Award-winning support, training, and consulting services make Red Hat a trusted adviser to the Fortune 500.


linkedin.com/company/red-hat


youtube.com/@redhat


facebook.com/redhat


x.com/RedHat

Red Hat