Linux Booting Feasibility Report

`bitbangers`

March 3, 2023

Disclaimer: This document is a somewhat rushed summary of features needed for booting Linux, and is closer to notes jotted down while figuring out what to do than an actual specification or report, which is upcoming.

1 Overview

The goal of this document is to discuss what RISC-V 32-bit ISA extensions need to be implemented to boot MMU Linux with at least `bash` as PID1. For the rest of this document, we implicitly assume a in-order uniprocessor CPU (exactly one RISC-V *hart*[1]). This choice may at a later time be revisited (but probably not this semester…) This choice makes some of the memory ordering constraints and atomics much easier to implement.

2 Booting Linux

2.1 No MMU

Linux itself has a `nommu` build that can be booted from the following minimal specification:

[1] A *hart* is a hardware thread.

Feature	Implementation
RV32I	Mostly implemented, we'll need to take a look at `FENCE` (memory barriers, not too bad since no multiprocessor/OOO), `ECALL`, `EBREAK`.
RV32M	Multiplication/division/modulo, should be easy to implement. These instructions, however, will be quite slow. Not sure if Verilog `*`, `/`, `%` are synthesizable.
RV32A	Atomics (LL/SC and AMO), should be easy to implement with our uniprocessor assumption.
Zifencei	Memory barrier for instruction prefetch caches. Again, hopefully no OOO/multiprocessor will make this "trivial", at worst a cache flush.
Zicsr	Instructions to read-modify-write a single CSR register out of the 4096. The instructions are simple, but *which* actual CSRs need to be present for Linux booting? The list for `nommu` is simply a few M-mode registers, simple counters, and a timer: `[mstatus, cyclel, cycleh, timerl, timerh, timermatchl, timermatchh, mscratch, mtvec, mie, mip, mepc, mtval, mcause]`
PLIC	Basic memory-mapped interrupt controller. Sends interrupts via a MMIO-CSR based mechanism. To implement, need to understand this better, this isn't a part of the privileged spec but seems to be "common" RISC-V knowledge. Probably only need timer interrupts for `nommu`.

The first goal is to get `nommu` Linux booting as a checkpoint — this would mean we have a small, embedded RISC-V core booting Linux.

2.2 With MMU

The key idea of an MMU for RISC-V is adding virtual memory, a supervisor and user mode, and a bunch of instructions/CSRs that go with the additional modes. The advantage of building a `nommu` Linux first is that we can add a CPU feature, flip the corresponding Linux configuration flag, and (hopefully) still have a working system.[2] The key question is: how much should we implement? The following table summarizes what's necessary for a "classical" UNIX-y operating system on RISC-V:

[2] Obviously, things won't be so clean in the real world, but one can dream…

Feature	Implementation
S-mode	This is the big one. The main extra instruction here is SFENCE.VMA, which flushes the TLB (MMU cache). There are a bunch of extra CSRs, relating to trap vectors, trap cause, virtual memory, counters etc.
sv32	This is the 32-bit virtual memory implementation with 4KiB pages and 2-level page table radix tree. Requires a hardware page-table walker. Protection bits (RWX, user access) and bookkeeping bits (A, D) in each PTE. There is support for separate address spaces (ASIDs), which complicates TLB implementation and SFENCE.VMA.
Traps	Traps deserve a second mention since they're more complex than in `nommu`. Now, we need support for trapping, then directing the program counter to an instruction from the trap vector table, and setting CSRs with cause data. This working correctly is vital for Linux to correctly handle page faults, load faults, store faults, etc. We'll only support hardware interrupts from the PLIC (no soft interrupts that come from other harts.)

The primary added complexity is of course the MMU itself, consisting of a TLB, a hardware page table walker, and logic to figure out if the translation violates memory protection etc. There are also more subtle issues with speculative loading of the TLB and VIPT indexing into our memory cache that are mentioned by the RISC-V spec that'll have to be looked into in greater detail. We'll also have to look into implementing the U-mode, although this shouldn't be hard once we enforce certain access policies on the CSRs and memory addresses, since that implicitly defines a low-privilege mode.

3 Anticipated Difficulties

3.1 Synthesis/FPGA

At its current state, we know our CPU synthesizes and passes timing, but don't know if the final bitstream, once flashed to an FPGA will actually have expected behavior. If we are to boot Linux on an FPGA, this should be done sooner rather than later, since the processor is going to get significantly more complicated, and ensuring the base RV32I processor works on the FPGA is important.

3.2 Making Linux Images

Ideally, we'd be able to use `buildroot`. This is nice because it makes a `qemu` binary that can boot the image it builds. It also builds a `busybox` binary that is loaded as the first user process. The difficulty here is likely going to be figuring out the kernel options, but we have access to the configuration file for `no-mmu` Linux, so editing that with `gconfig` may just work.

4 Resources

- Linux `nommu`: `https://github.com/cnlohr/mini-rv32ima`
- CLINT: `https://chromitem-soc.readthedocs.io/en/latest/clint.html`
- Privileged instruction set slides: `https://riscv.org/wp-content/uploads/2018/05/riscv-privileged-BCN.v7-2.pdf`