DOOM-V: Running DOOM on a RISC-V Core

Doomed

Jacob Farrell, Lawrence Su, Yu Xia, Daniel Zhao

Project Proposal Document

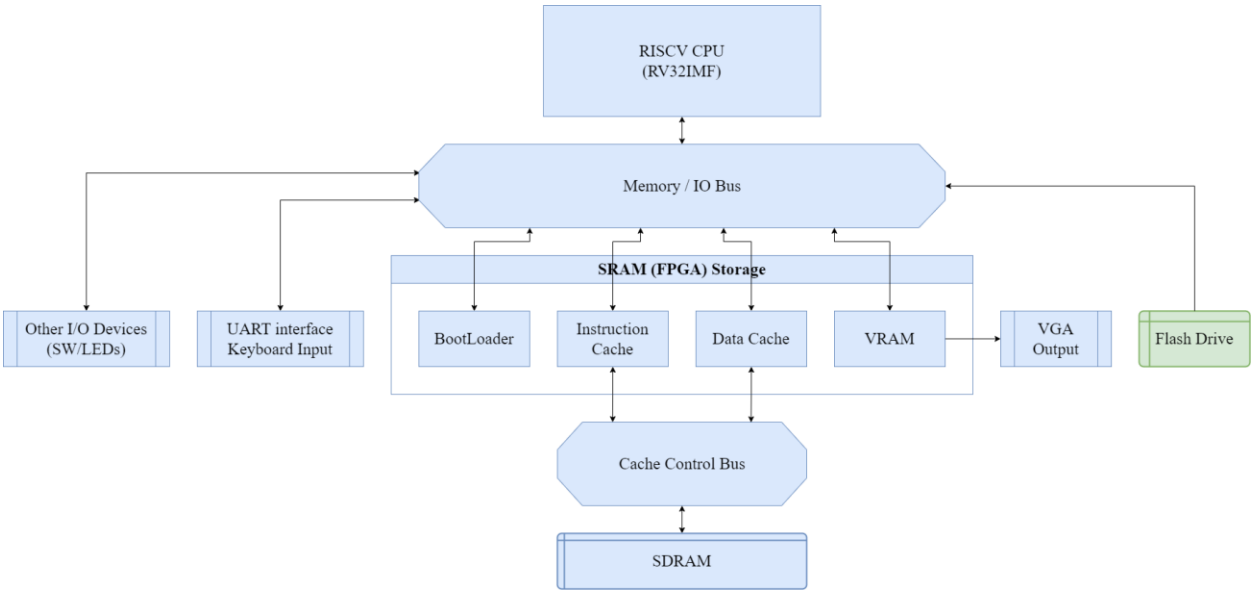
University of Wisconsin-Madison

Table of Contents

1. Introduction

DOOM-V aims to run a faithful implementation of the 1993 game DOOM on an FPGA implementing a RISC-V core. This project is inspired by posts on the internet that have demonstrated running DOOM on obscure and strange hardware, mostly for the sake of humor. We would like to run DOOM on our own “obscure and strange” self-made hardware not just for the humor but because of the technically interesting challenges it poses. These challenges include making a fast processor, utilizing a memory hierarchy that includes caches, and implementing a couple hardware-mapped peripherals.

2. Hardware Block Diagram



The project would use a RISCV CPU using the RV32IMF specification for most of its computing operations. The processor would access through its main memory interface and a range of I/O devices through a bus, which would contain the dedicated instruction and data caches. The code for the DOOM program would be placed within the external Flash Drive, which would be loaded onto the SDRAM on startup and reset by the BootLoader.

2.1. RISCV CPU

The CPU is the centralized processor for the project. While some information regarding the ISA has been discussed later in Section 3, additional information regarding the microarchitecture of the processor will be discussed here. The CPU would contain a 5-state pipelining and would have a dedicated Floating-Point Unit and Multiplier for the computation of Integer multiplication and Floating-Point operations. The address space for the CPU would be a 32-bit address space, and a primitive memory map with different devices is shown below:

- `0x0000_0000 – 0x2000_0000`: BootLoader Code
- `0x2000_0000 – 0x6000_0000`: SDRAM (64M)
- `0x6000_0000 – 0x7000_0000`: VRAM
- `0x7000_0000 – 0x7FFF_FFFF`: I/O Mapped Memory

2.2. SW/LEDs block

The Switch and the LEDs would be controlled through the I/O Mapped Memory by two address locations. By the time of writing this document, address 0x7000_C000 is being used to write the status of the LEDs (2 bytes, where the lowest 10 bits were used for the output of LEDs), and address

0x7000_C002 is being used to read the status of the SW (2 bytes, where the lowest 10 bits were used for the input of SWs).

2.3. UART block

The UART driver would be a generalized driver that supports UART communication between a keyboard and the FPGA, while parts of the I/O Mapped Memory would be used to control the transmission of data. The current idea of implementation is based on the SPART module in the previous Minilab, which maps address 0x7000_C004 to the buffer of reading from keyboard (1 byte), 0x7000_C005 to the status register of the UART communication (1 byte), and 0x7000_C006 to the baud rate control of the register.

2.4. VGA/VRAM block

VRAM is a flat map and can be written to directly from the processor. There are no control signals for the hardware VGA driver. Whatever is written to VRAM will be displayed on the next iteration of the VGA driver. This may lead to screen-tearing so double-buffering may be investigated as well for the VGA block.

2.5. Flash + SPI

The Flash Drive would be an external drive that contains the code and assets of the program DOOM. The data would be loaded by BootLoader on startup. The flash chip will be interfaced over SPI.

2.6. BootLoader block

BootLoader is the code being executed on reset, which does some start up preperations. More details will be discussed later in Section 4 on the supporting software.

2.7. Caches and SDRAM

The processor uses dedicated Instruction and Data Caches and a unified main memory in the SDRAM. A dedicated cache control bus would be used to ensure that the requests for main memory access would be arranged in a timely manner and avoid any collision conditions.

3. Processor

DOOM-V will use a complete implementation of the RV32IMF ISA.

3.1. ISA Summary

We will be using the 32-bit variant of RISC-V with the I (integer), M (multiplication and division), and F (32-bit floating point) extensions. This ISA implementation can be concisely referred to as RV32IMF. The full specification can be found at https://riscv.org/technical/specifications/.

3.2. Condition Codes

RISC-V does not support condition flags.

3.3. Addressing Modes

RISC-V uses only register + offset addressing for loads and stores.

3.4. Immediate

RISC-V has I, S, B, U, and J instruction formats that encode immediates. Further details can be found in the specification.

3.5. Harvard vs Von Neumann

We will implement a Von Neumann architecture, having a singular unified memory for instructions and data. We will include separate instruction and data caches.

3.6. Special Features

Our implementation will implement the M (multiplication and divide) and F (32-bit floating point) extensions on top of the base integer implementation.

Additionally, the processor will utilize branch prediction and separate data and instruction caches for improved performance.

3.7. Co-processors

Our processor does not implement any co-processors but will implement a floating point functional unit.

3.8. Other architectural features

There are no other notable architectural features.

4. Software Blocks

4.1. Assembler/Compiler

We will be using the riscv32-unknown-elf GCC toolchain.

4.2. Simulator

Can do function tests in modelsim. Test suite: https://github.com/riscv-software-src/riscv-tests

The first part of the tests is functional tests under modelsim. A verilog testbench run the self-checking test suite (https://github.com/riscv-software-src/riscv-tests) on our implementation.

The second part is to compare the trace against SPIKE (RISC-V foundation's simulator). Ideally, we need to make the simulator to output every write/read operation to registers and memory.

4.3. Application

https://github.com/cnlohr/embeddedDOOM

We will run 3D video game DOOM (1993) on our RISCV implementation, with graphics output. We will compile and link the game from a copy of the source linked above. Of course, the source will be modified to fit with our hardware and ISA (it is currently designed to run on 32-bit x86 Linux). The game utilizes floating point unit and multiplier. The graphics are rendered directly into VRAM. UART interface is used to control the game from a host computer.

On reset, PC is set at the start of Bootloader. The Bootloader code either copies the entire Flash to a specific address located in DRAM and jumps to the target code or accepts commands from the host computer through UART interface.

5. Division of Labor

Name	Labor
Jacob Farrell	SPI controller, bootloader, porting DOOM code
Lawrence Su	Project management, integer MUL and DIV, FPU
Yu Xia	Implement pipelined RV32IMF processor
Daniel Zhao	DRAM, caches