THE DESIGN DOCUMENT FOR CSEE 4840 PROJECT RISCY

Shuai Zhang (sz3034)

Spring 2022

Contents

Introduction

RISC-V is an open-source instruction set architecture developed by UC-Berkley and have been picking up popularity and community over the years. Apart from the open-source instruction encoding, there are also development tools such as C compiler, simulator etc. available. This document will be about the specification of my own home-brew RISC-V core (project name: RISCY), The goal for this project is to load it into the FPGA and making the DE-1 development board behave like an Arduino. Specifically, this project will:

- HW side: the DE-1 development board shall be loaded with RISCY, which reads pre-compiled instructions, perform read / write to memory locations (including MMIO), and interact with the board's peripheral such as GPIO pins, 7-Segs, switches, UART command lines. Etc.
- SW side: I will use the HPS module to communicate with RISCY, and develop respective software including:
    - Compiler proxy: pass the C / C++ code to my cloud server for compilation and fetch back compiled assembly and binary. The compilation could not be done on HPS because the toolchain doesn't support 32-bit arm
    - Bootloader: write the binary file into RISCY memory space
    - Debugger: handling debugger instructions (ECALL), read memory content, alter memory content, and resume RISCY

Sidenote1: project is called RISCY is because I morphed RISC-V's V to Y, so it makes easier for me to pronounce. (And because I'm almost certain it will consume me hundreds of hours building this project and it's a *ris-ky* decision)

Sidenote2: I updated the project objective from my proposal from 'just a debugger' to 'anything I want it to be', I noticed that since my RISCY core and the HPS shares memory space, I can even use my RISCY core as a co-processor for the HPS!

Sidenote 3: GitHub link:

System Block Design

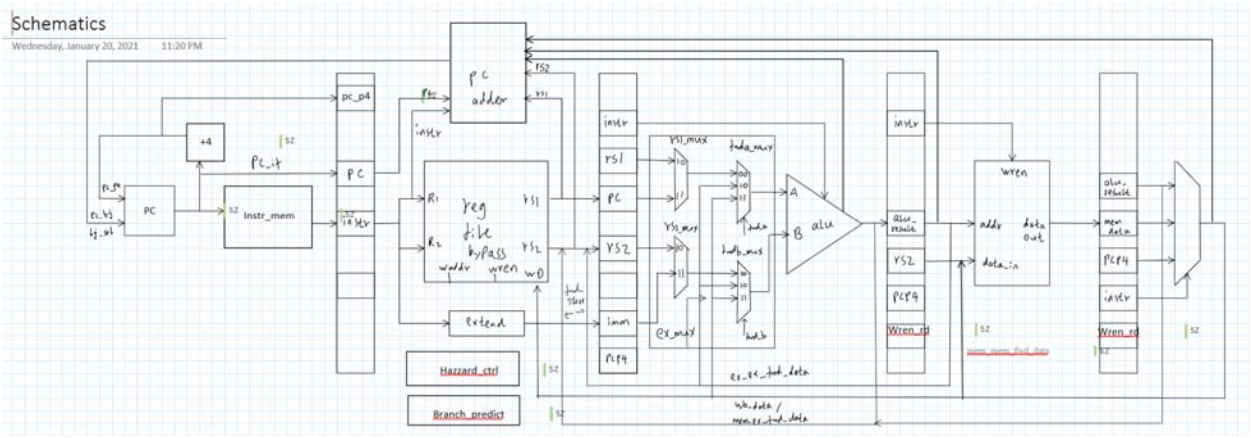

Fig 1, the sketch of RISCY core architecture

Fig 1 shows the very basic architecture of the RISCY core, it's a basic classic 5-stage pipeline design

Core input: clock, reset, enable

Core output: instruction fetch bus, data read/write bus (self-defined simple ready-valid handshake bus)

Note that this sketch was created a long time ago, and it does not exactly match what the core is now, and data path are not all labeled. Its only purpose is to give a sense of design principles.

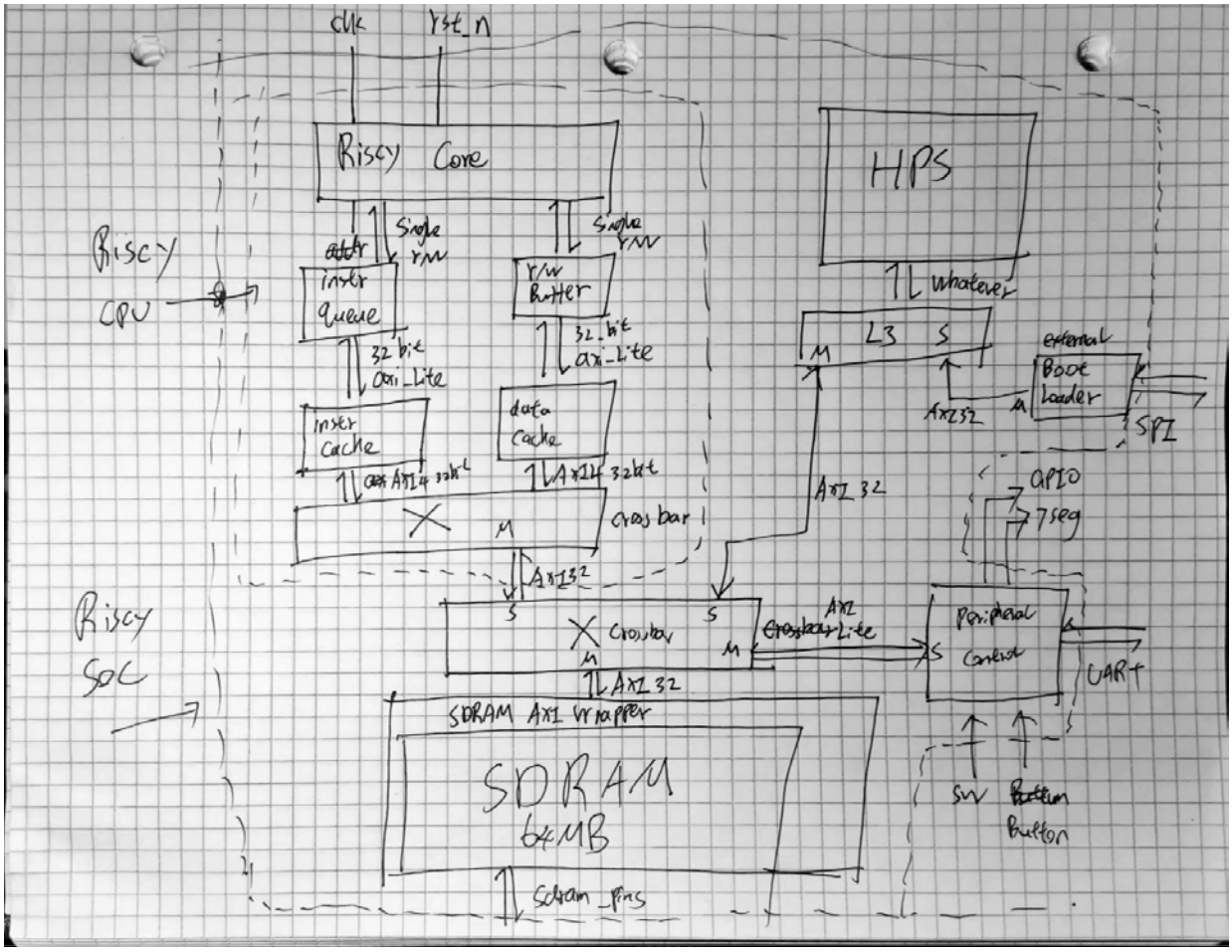

Fig 2: the sketch of the overall SoC system

Fig 2 shows the overall SoC structure of the project. The Riscy Core's instruction bus and data bus are merged into one single AXI bus. This bus is then merged with the HPS's AXI bus into a center crossbar interconnect. The interconnect is then connected to an AXI-wrapped SDRAM controller and peripheral controller.

Both the HPS and the external bootloader can bootstrap the RISCY system via the HPS H2F AXI bus.

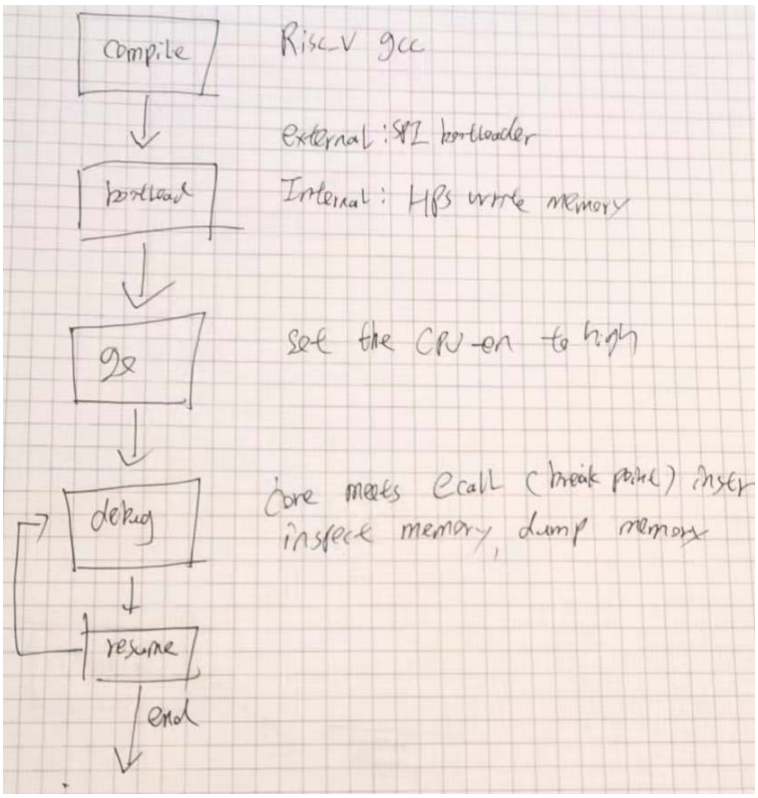

Fig 3, Software workflow

Fig 3 shows the software front workflow for a typical debugging process. This is a very coarse draft and will add more details once I finish the hardware system.

Algorithms

Frankly there's not a whole lot of fancy algorithms involved in this project. Everything is retrospective and old-fashioned. Instead, I'll just list the key specs of this project

1. CPU Core:
    a. In-Order Classical 5 stage pipeline
    b. Support RV32IMA instruction set
        i. 32-bit word size, small endian, byte addressable
        ii. I: base integer operation ISA
        iii. M: integer multiplication / division operation ISA
        iv. A: atomic instruction set
    c. Full forwarding to solve hazard
        i. Side notes 1: this is a BAD idea because it hurts timing a LOT. Should've gone with plain stalling…
        ii. Side notes 2: the forwarding / stalling logic is so complicated that I once though I should've gone with out-of-order core design…
    d. Parameterizable instruction fetch buffer
    e. Parameterizable memory read/write buffer
    f. Parameterizable system cache

        i. Can be either unified or separate I / D cache
        ii. 2-Way set associative cache, parameterizable depth
        iii. Write-back policy, LRU victim eviction policy
    g. Hardware Multiplication / Division support (can be generated by either pure combinational or using the Altera DSP)
    h. Atomic instruction support
    i. Unified system bus, can be either AXI4 or AXI-Lite
        i. AXIL to AXI and AXI to AXIL bridge available
    j. Pure System Verilog, platform independent, no priority IP
    k. Formally verified (better than nothing)

2. SoC System:
    a. 512Mb on-board SDRAM
        i. Controller with AXI wrapper
    b. Unified bus everywhere
        i. All buses are either AXI or AXI-Lite, made bus translation overhead much smaller.
    c. Support most on-board peripheral:
        i. All GPIO
        ii. UART for serial I/O
        iii. SW and push button
        iv. 7-seg for system state indicator
        v. SPI for bootloader
        vi. (hopefully) simple VGA output API

Resource Budgets

Only for RISCY core, not the whole SoC system

Logic: around 4K logic element at 50MHz timing requirement, 3K if 40MHz

Memory block: 0 (set all buffer depth to 0), infinite if I just ramp up the numbers

DSP block: 23 if support mult/div instruction, 0 if not

The Hardware/Software Interface

The RISC-V ISA includes 4096 32-bit control status registers (CSR), the registers are memory mapped and controlled by a stand-alone controller, and I will choose some of them to be my debugging interface.

The CSR will include: (note, the RW privileges are from the RISCY core's perspective)

1. 0x100: The "GO" register (R)
    a. the processor will pull on it after reset or after ECALL/EBREAK instructions
    b. PC resume if bit 0 set, otherwise pulse
    c. Bit clear once PC pulls on it when set
2. 0x101: the 'FSM_ERR' register (stand-alone W)
    a. Each bit corresponds to one FSM in the system (TBD)
    b. Bit set if corresponding FSM goes into bad state
3. 0x102: the ASSERT_ERR register (stand-alone W)

    a. Each bit corresponds to one assertion error latch (TBD)
    b. Bit set if corresponding assertion fails
4. 0x103: the EBREAK register (W)
    a. Bit 0 set indicates the RISCY core run into a EBREAK instruction.
    b. Indicates RISCY is in debug state, PC freezes.
    c. HPS can read this bit and can safely read / write into RISCY memory space
5. 0x103: the ECALL register (W)
    a. Bit 0 set indicates the RISCY core run into a ECALL instruction.
    b. Indicates RISCY requests supervisor handler
    c. More of a placeholder, beyond the scope of this project
6. 0x104: the STOP registers (W)
    a. Bit set indicates the core run into a HALT instruction
    b. Indicates the program either entered a SW bad state or ended execution
7. 0x105: the GPIO_0_RW_CTRL register (W)
    a. Bit set if output, bit clear if input (Inout not supported)
8. 0x106: the GPIO_0_OUT register (W)
    a. Write to GPIO 0, output 1 if set
9. 0x107: the GPIO_0_IN register (R)
    a. Mapped from GPIO, read 1 if set
10. 0x108: the GPIO_0_mask register (W)
    a. Mask GPIO IO data

Let's not go so far ahead of ourselves. I will extend the peripheral control registers once I finish building the system.

At this point what really matters is the CSR 0x100 – 0x104, which are vital for the bootloader / debugger to function. After the core starts running I will add support for more peripheral.