Project Report of RISCV-CPU

Yi Gu

January 4, 2020

1 Summary

This project is a RISC-V CPU with five-stage pipeline, implemented in Verilog HDL.

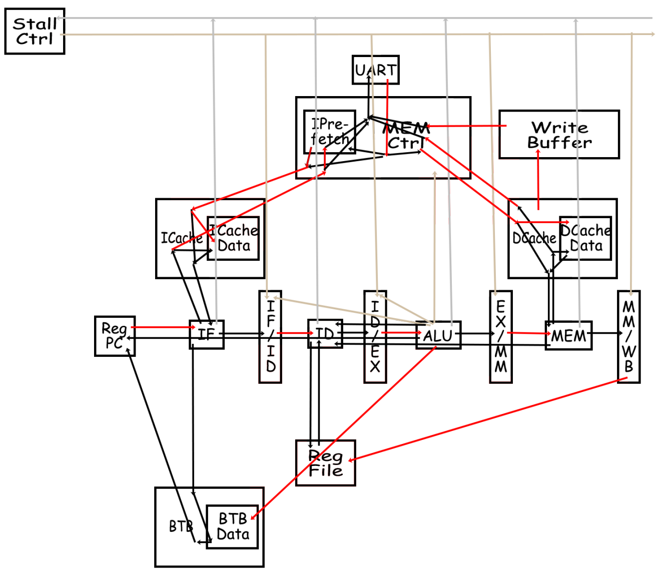
The CPU can be simulated by Icarus Verilog or Xilinx xsim, and can be implemented on Basys3 FPGA, using Xilinx vivado.

The repository of this project may be found at: `https://github.com/wu-qing-157/RISCV-CPU`.

2 Design

2.1 Project Overview

This project has a standard 5-stage pipeline with some additional features, and the overview structure is shown in the below figure:



Note: Red lines represent data flow controlled by clock. Lines with light color represent control flow.

2.2 Features

Main features implemented in this project are listed as below, with some of them described in later part of this section.

Feature	Description
ISA	RV32I subset of RISC-V
Pipelining	Standard 5 stages
Data Forwarding	Complete forwarding paths
Cache	Direct mapping Instruction Cache & Direct mapping Data Cache
Write Buffer	1-block 4-byte Write Buffer
Branch Prediction	2-bit Branch Target Buffer
IF Prefetch	Prefetch during spare time of memory port

Details of some features are listed below:

Instruction Cache A direct mapping I-Cache with size of 256×4 bytes is implemented. If cache hits, the data will be returned in the same cycle, while, otherwise, fetching of the instruction will be started from the next cycle. The slow behavior of cache-hit check and memory port is to blame for the additional cycle. The latency caused by the additional cycle is eliminated with *IF Prefetch* described below.

Data Cache A direct mapping D-Cache with size of 128×4 bytes is implemented. The write policy is a mixture of write back and write through, and in detail, 4-byte store insructions use write back and other store instructions use write through. It is so arranged because of the large proportion of 4-byte store instructions, and the 4-byte block size due to limitation of units on FPGA. Similar as I-Cache, cache-hit data is returned in the same cycle, while cache-misses take an addtional cycle.

Write Buffer A naive single-block 4-byte write buffer is added for D-Cache. Data in write buffer will be flushed when memory port is not busy.

IF Prefetch After an instruction is fetched, the next instruction will be pre-fetched. The prioity of the pre-fetching is lower than load needed by D-Cache, but higher than write buffer. Note that in this implementation, I-Cache-miss and D-Cache-miss takes 3 unused cycles, the pre-fetching will also take advantage of these 3 cycles, with exception described in next section. Note that *IF Prefetch* will be interrupted by wrong branch predition (prefetching is meaningless), or branches predicted taken (cache-hit rate of the target instruction is very high).

2.3 Performance

The implemented CPU can be run on FPGA with a frequency of 230 MHz.

It can pass all tests on FPGA listed in the project repository. Note that it has some problems during simulation due to some issues in the given *hci.v*.

Some detailed performance quota are tested based on testcases in the project repository.

Testname	Instructions	Cycles	CPI	FPGA Time (sec)
bulgarian	872042	955046	1.095	-
pi	-	-	-	0.469
magic	661360	723539	1.094	0.031
queens	521577	585617	1.123	0.563
love	-	-	-	183.3

3 Thinkings

Issues in given files In the given files, there does exist lots of issues disturbing my finishing this project. When testing with simulation, some bytes written to IO port will be displayed twice or even four times. In order to check who is to blame, I need to check the huge wave grapth. Reading from IO port will cause last reading from memory port invalid, and the reading address one cycle after reading from IO port must have 1 on its 17-th bit to keep the IO reading valid. These two issues disturb me a lot when debugging my project on FPGA. I think as a course project, the above 3 issues are supposed not happen, and we students should at least be informed. However, the assignment repository and TAs give me a bad feel. Addtionally, due to the issues mentioned above, the *IF Prefetch* will be interrupted by a IO reading, and will be restarted after next IF reading.

The motivation for *IF Prefetch* and *Write Buffer* The memory port on FPGA is completely different from RAM in modern computer, where the motivation for *IF Prefetch* and *Write Buffer* comes from. *Write Buffer* dalays unimportant write behavior to gaps between reading, and *IF Prefetch* makes full use of the actually unused cycles. Based on simulation tests, it's better to grant *IF Prefetch* a higher priority than *Write Buffer*.

The block size of Cache Due to the limitation of units on FPGA, the way of memory access, and the proportion of 4-byte load instructions, as well as the difficulties in implementation, 4-byte block size is chosen for both I-Cache and D-Cache.

Mapping method of Cache and BTB The performance on FPGA is not so good for me to choose fully-associated or set-associated. Actually, even implemented as direct mapping, I-Cache-hit and D-Cache-hit are the bottleneck of the performance.

Total size of Cache and BTB Currently I have 256 blocks for I-Cache, 128 blocks for D-Cache, and 32 blocks for BTB. As tested, the CPI performance can only increase by less than 0.5% if I double any of it, so these 3 figures are chosen.

4 Acknowledgement

Special thanks would go to Yuan Yao and Zonghan Yang who discuss with me some features implemented in this project, and Haichen Dong who informed me of the issues in given files. Also, I would like to express my gratitude to TAs for providing codes for memory and IO ports (although with some issued). I also referenced the book 自己动手写 *CPU* and Zhou Fan's project at the very beginning of my own project.

Reference

- 雷思磊. 自己动手写 *CPU*, 电子工业出版社, 2014
- Zhou Fan. RISC-V-CPU. `https://github.com/Evensgn/RISC-V-CPU`