CPU with Five-Stage MIPS Pipeline

Zhijian LIU

September 12, 2016

Course:	Computer System
Instructor:	Professor Alei Liang
Teaching Assistant:	Shusheng He

Contents

1 Introduction

This report is a detailed summary on the final project of the "Computer System" course. In this project, I implemented a toy CPU with five-stage MIPS pipeline.
The project has the following features:

- It supports almost all of the MIPS standard integer instructions (except those related to the coprocessors).
- It supports the full bypassing in order to mitigate the data hazards.
- It has a well-organised structure and supports the automatic testing.

The project is written in Verilog HDL and is available under the open-source MIT license at `https://github.com/zhijian-liu/mips-cpu`.

2 Main Modules

2.1 Stage Modules

In the five-stage MIPS pipeline, there are obviously five separate pipeline stages:

- stage IF (instruction fetch)
- stage ID (instruction decode)
- stage EX (execution)
- stage MEM (memory access)
- stage WB (write back)

In my design philosophy, each of them should be implemented as a single module.

2.1.1 Instruction Fetch

The main function of this module is to maintain the program counter and send it to the instruction ROM in order to fetch the current instruction.

Interface	Type	Usage
clock	in	clock signal
reset	in	reset signal
stall	in	stall signal
register_pc_read_data	out	register PC
register_pc_write_enable	in	
register_pc_write_data	in	

Table 1: the interfaces of the stage IF

This module is implemented in `src/cpu/stage/stage_if.v`.

2.1.2 Instruction Decode

The main function of this module is to decode the instruction and extract the operator, category, operands and destination from it.

Interface	Type	Usage
reset	in	reset signal
instruction_i	in	instruction
instruction_o	out	
operator	out	operator
category	out	category of the operator
operand_a	out	operands
operand_b	out	
register_read_enable_a	out	read port A of the register file
register_read_address_a	out	
register_read_data_a	in	
register_read_enable_b	out	read port B of the register file
register_read_address_b	out	
register_read_data_b	in	
register_write_enable	out	write port of the register file
register_write_address	out	
register_write_data	out	
register_pc_read_data	in	register PC
register_pc_write_enable	out	
register_pc_write_data	out	
ex_operator	in	data forwarding from the stage EX
ex_register_write_enable	in	
ex_register_write_address	in	
ex_register_write_data	in	
mem_register_write_enable	in	data forwarding from the stage MEM
mem_register_write_address	in	
mem_register_write_data	in	
stall_request	out	whether or not is to request stall

Table 2: the interfaces of the stage ID

In order to mitigate the data hazards[1], some interfaces are designed to fetch the latest data forwarding[2] from the stage EX and the stage MEM.
After implementing the full bypassing, there is only one kind of data hazard: a load instruction immediately followed by an instruction which has data dependency on it. In this case, we have to enable the "stall_request" signal in order to stall the pipeline for one cycle.
This module is implemented in `src/cpu/stage/stage_id.v`.

[1] https://en.wikipedia.org/wiki/Hazard_(computer_architecture)#Data_hazards
[2] https://en.wikipedia.org/wiki/Operand_forwarding

2.1.3 Execution

The main function of this module is to calculate the operation result according to the operator and operands received from the stage ID.

Interface	Type	Usage
reset	in	reset signal
instruction_i	in	instruction
instruction_o	out	
operator_i	in	operator
operator_o	out	
category	in	sub-category of the operator
operand_a_i	in	operands
operand_a_o	out	
operand_b_i	in	
operand_b_o	out	
register_write_enable_i	in	write port of the register file
register_write_enable_o	out	
register_write_address_i	in	
register_write_address_o	out	
register_write_data_i	in	
register_write_data_o	out	
register_hi_write_enable	out	register HI
register_hi_write_data	out	
register_lo_write_enable	out	register LO
register_lo_write_data	out	
mem_register_hi_write_enable	in	data forwarding from the stage MEM
mem_register_hi_write_data	in	
mem_register_lo_write_enable	in	
mem_register_lo_write_data	in	
wb_register_hi_read_data	in	data forwarding from the stage WB
wb_register_hi_write_enable	in	
wb_register_hi_write_data	in	
wb_register_lo_read_data	in	
wb_register_lo_write_enable	in	
wb_register_lo_write_data	in	
stall_request	out	whether or not is to request stall

Table 3: the interfaces of the stage EX

In order to mitigate the data hazards, some interfaces are designed to fetch the latest data forwarding from the stage MEM and the stage WB.
Note that the "stall_request" signal is actually redundant because in the functional simulation, all kinds of calculation can be finished in a single cycle.
This module is implemented in `src/cpu/stage/stage_ex.v`.

2.1.4 Memory Access

The main function of this module is to handle the memory access instruction: calculate the effective memory address[3] and send it to the data RAM in order to execute the load or store operation.

Interface	Type	Usage
reset	in	reset signal
instruction	in	instruction
operator	in	operator
operand_a	in	operands
operand_b	in	
memory_read_enable	out	read port of the data RAM
memory_read_address	out	
memory_read_data	in	
memory_write_enable	out	write port of the data RAM
memory_write_address	out	
memory_write_select	out	
memory_write_data	out	
register_write_enable_i	in	write port of the register file
register_write_enable_o	out	
register_write_address_i	in	
register_write_address_o	out	
register_write_data_i	in	
register_write_data_o	out	
register_hi_write_enable_i	in	register HI
register_hi_write_enable_o	out	
register_hi_write_data_i	in	
register_hi_write_data_o	out	
register_lo_write_enable_i	in	register LO
register_lo_write_enable_o	out	
register_lo_write_data_i	in	
register_lo_write_data_o	out	

Table 4: the interfaces of the stage MEM

This module is implemented in `src/cpu/stage/stage_mem.v`.

[3] https://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Mips/addr.html

2.1.5 Write Back

The main function of this module is to maintain the register HI and register LO.

Interface	Type	Usage
clock	in	clock signal
reset	in	reset signal
register_hi_read_data	out	register HI
register_hi_write_enable	in	
register_hi_write_data	in	
register_lo_read_data	out	register LO
register_lo_write_enable	in	
register_lo_write_data	in	

Table 5: the interfaces of the stage WB

This module is implemented in `src/cpu/stage/stage_wb.v`.

2.2 Latch Modules

In the five-stage MIPS pipeline, there are totally four pipeline latches which act as the buffers between the neighbouring pipeline stages.
These modules are implemented in:

- `src/cpu/latch/latch_if_id.v`
- `src/cpu/latch/latch_id_ex.v`
- `src/cpu/latch/latch_ex_mem.v`
- `src/cpu/latch/latch_mem_wb.v`

Note that all of the pipeline latches propagate the data on the rising edge of the clock signal.

2.3 Storage Modules

In my implementation, all of the storage modules write the data on the falling edge of the clock signal. Otherwise, some additional codes will be needed to support the bypassing inside the storage modules.

2.3.1 Instruction ROM

Because the instructions cannot be modified at runtime, the instruction ROM has only one major function: memory read. It uses a set of input wires to read the program counter and outputs the corresponding instruction.
This module is implemented in `src/rom.v`.

2.3.2 Data RAM

Unlike the instruction ROM, the data RAM has two major functions: memory read and memory write. It uses a set of input wires to read the address and outputs the corresponding data. At the meantime, it also provides a set of wires for writing.
This module is implemented in `src/ram.v`.

2.3.3 Register File

The register file provides two sets of input wires for querying the value of a given register, and a set of wires for writing a register.
This module is implemented in `src/cpu/register.v`.

2.4 Control Module

The main function of the control module is to receive the stall request from the pipeline stages and send the stall signal to the pipeline latches.
This module is implemented in `src/cpu/control.v`.

3 Test Summary

3.1 Test Case

In the `test` folder, there are ten test cases in total:

- `test-logic` from Lei (2014): check the logical instructions.
- `test-shift` from Lei (2014): check the shift instructions.
- `test-move` from Lei (2014): check the move instructions.
- `test-arithmetic` from Lei (2014): check the arithmetic instructions.
- `test-jump`: check the jump instructions.
- `test-branch`: check the branch instructions.
- `test-memory` from Lei (2014): check the memory instructions.
- `test-stall` from Lei (2014): check the stall detection.
- `test-forwarding` from Lei (2014): check the data forwarding.
- `test-yamin` from Li (2011): overall check.

In each test case folder, there are totally three files:

- `asm.s`: the testing MIPS assembly code.
- `test.v`: the automatic testing code written in Verilog HDL.
- `Makefile`: the testing script.

A detailed guide to running the automatic testing is written in `README.md`.

3.2 Test Result

After running the automatic testing, you will see the following in your terminal.

```
cd test-stall      && make
==> start testing stall
cd ../ && python assemble.py -s test-stall/asm.s -o test-stall/rom.txt
iverilog -c ../filelist.txt -g2009 -o test.vvp
vvp test.vvp
VCD info: dumpfile test.vcd opened for output.
VCD warning: array word test.sopc.cpu.register.storage[1] will conflict with an escaped identifier.
WARNING: ./test.v:19: $readmemh(rom.txt): Not enough words in the file for the requested range [0:1024].
==> test passed.
rm -rf rom.txt test.vvp

cd test-forwarding && make
==> start testing forwarding
cd ../ && python assemble.py -s test-forwarding/asm.s -o test-forwarding/rom.txt
test-forwarding/asm.s: Assembler messages:
test-forwarding/asm.s: Warning: end of file in comment; newline inserted
iverilog -c ../filelist.txt -g2009 -o test.vvp
vvp test.vvp
VCD info: dumpfile test.vcd opened for output.
VCD warning: array word test.sopc.cpu.register.storage[5] will conflict with an escaped identifier.
WARNING: ./test.v:19: $readmemh(rom.txt): Not enough words in the file for the requested range [0:1024].
==> test passed.
rm -rf rom.txt test.vvp

==> all tests passed.
```

Figure 1: the automatic testing result in the terminal

At the meantime, a waveform simulation file (`dump.vcd`) will be generated in each test case folder. Here are some waveforms captured on Scansion[4].

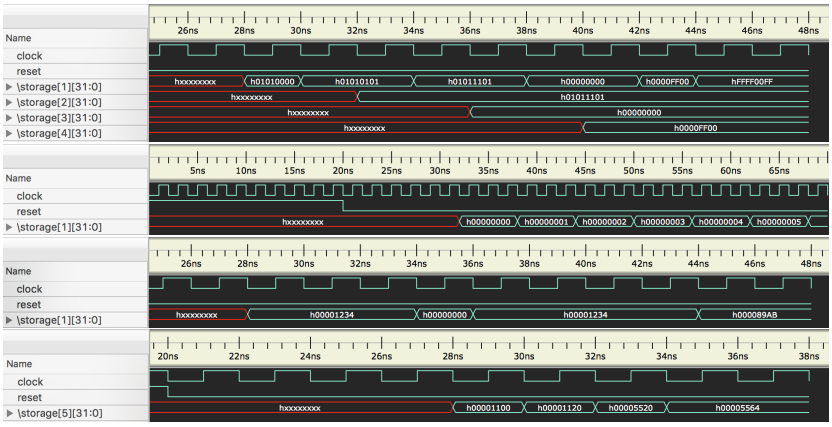

Figure 2: the waveform simulation of some test cases

[4] http://www.logicpoet.com/scansion/

4 Discussion

From this project, I got acknowledged quite a lot:

- A new programming language, Verilog HDL.

  Verilog is a hardware description language which supports nearly none features of a high-level language. I took Palnitkar (2003) as my reference book.

- A better understanding of computer architecture.

  I went over most materials in the textbooks: Hennessy and Patterson (2011) and Patterson and Hennessy (2013) and gained a better understanding of them.

Due to time constraints, it would be impossible to support all of the amazing features. As far as I know, this project has multiple aspects that can be further enhanced and refined:

- Instruction cache and data cache.

  There is no memory delay by now because all of the tests are functional simulation. Nevertheless, instruction cache and data cache are indispensable if we want to run this CPU on the FPGA.

- Static and dynamic branch prediction.

  Control hazards harm the performance badly while the static and dynamic branch prediction can mitigate the control hazards to a large extent.

  In my current implementation, one delay slot is supported to help alleviate this problem. Unfortunately, this solution is not always practicable since in some cases, the compiler cannot find any codes to fill in the delay slot.

- Out-of-order execution with Tomasulo's algorithm[5].

  The major innovations of Tomasulo's algorithm include register renaming in hardware, reservation stations for all execution units, and a common data bus on which computed values broadcast to all reservation stations. These developments allow for improved parallel execution of instructions.

- Exception and interruption.

  To run a modern operating system, the CPU should be able to handle the internal exceptions and external interruptions. In order to support these features, the coprocessor should be implemented.

I plan to support most of these features in the coming "Computer System" course.

5 Acknowledgements

I would like to thank Professor Alei Liang for bringing me such a wonderful course, Shusheng He for his hard work and warm devotion, and Yurong You, Zihao Ye and Lequn Chen for valuable discussions.

[5] https://en.wikipedia.org/wiki/Tomasulo_algorithm

References

Hennessy, J. L. and Patterson, D. A. (2011). *Computer architecture: a quantitative approach*. Elsevier.

Lei, S. (2014). *CPU: do it yourself*. Publishing House of Electronics Industry.

Li, Y. (2011). *Computer Principles and Design in Verilog HDL*. Tsinghua University Press.

Palnitkar, S. (2003). *Verilog HDL: a guide to digital design and synthesis*. Prentice Hall Professional.

Patterson, D. A. and Hennessy, J. L. (2013). *Computer organization and design: the hardware/software interface*. Newnes.