Communal Pocket Universe

A Simple CPU Implementing Tomasulo's Algorithm with Reorder Buffer

邵俊儒[*] 陈昕昀[†]

ACM Honored Class
Zhiyuan College
Shanghai Jiao Tong University

Abstract

In the first section, we would describe in outline the design of our CPU. In the second section, we would introduce each unit or module and its utility respectively. As structural and memory hazards happen frequently at runtime, the third section gives a solution we use to resolve problems that might occur, which is mainly based on schedule arrangement. To avoid annoying conditions in the original MIPS instruction set, we defined our own MIPS-like ISA. We will talk about it in the fourth section. We would talk something about design philosophy in the fifth section. Collaboration is the most enjoyable part in this project, we present collaboration division in the sixth section. In the end, we would like to thank people who generously gave us warm help.

[*]Student ID:5130309028.
[†]Student ID:5130309066.

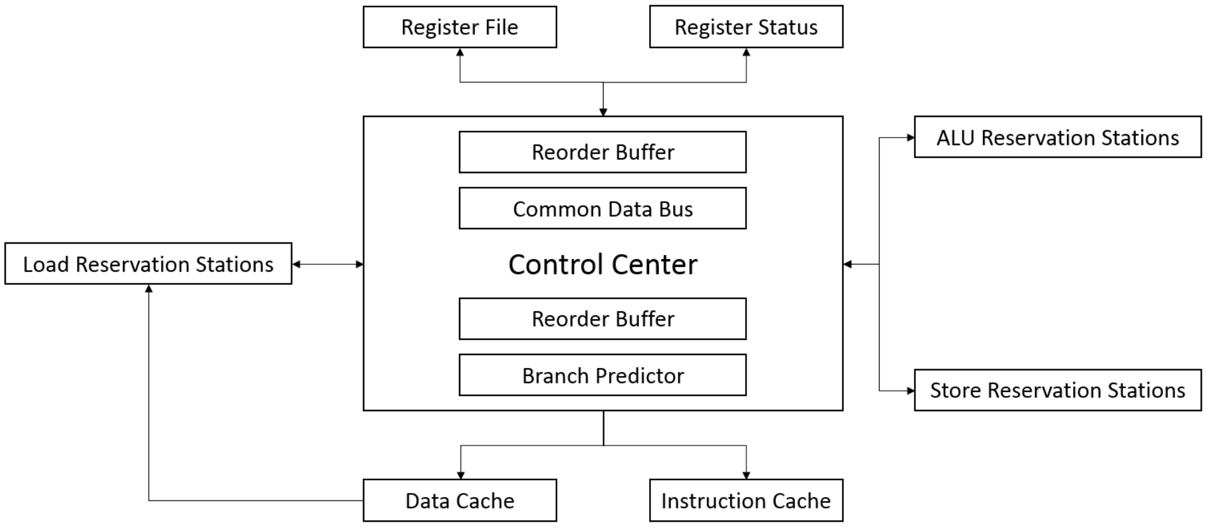
Contents

1 Introduction

This is a report for the final project of course MS108, computer system I. We implement an out-of-order execution algorithm with reorder buffer, a direct mapping instruction cache and a direct mapping data cache. In addition, we design an ISA on our own.

Our verilog program needs iverilog >= 0.10.0, Ubuntu >= 14.04, JRE: Oracle JRE >= 1.8.

The figure below shows the internal architecture of our CPU.



As we all know, a control center is indispensable for a CPU. In our implementation of Tomasulo's Algorithm, the control center is integrated in the module reorder buffer. The reorder buffer communicates with all other modules. More specifically:

The reorder buffer has 4 sets of wires connected with ALU reservation station, issue bus to send instruction to a certain reservation station, common data bus, and a wire back indicating the reservation station is busy, a set of wires back to notify the control center to modify reorder buffer.

The reorder buffer also has 4 sets of wires connected with store reservation station, same as described above.

The reorder buffer also has 4 sets of wires connected with load reservation station, same as described above. However, there is another set of wires connecting data cache and load reservation stations, letting the reservation stations read data from data cache.

The reorder buffer also connects with register file, register status table, data cache and instruction cache, which allows it read or write the corresponding module.

2 Unit Design and Utility

2.1 Parameters

Global parameters are defined in “Utility.v” , including size of cache, size of reorder buffer, number of function units, opcode of instructions, and delay of memory access.

2.2 Register File

Our register file provides two input wires for querying the value of a given register, and a single wire for writing a register. All reading and writing operations are responded immediately.

Hazards may exist when reading and writing the same register at the same time, which could be avoided using slickly designed schedule arrangement.

2.3 Register Status Table

Typical Tomasulo’s Algorithm uses register status table to record which function unit are being used to calculate each register. However, with a reorder buffer, we modify this algorithm using this table to record which cell in reorder buffer the register is waiting for.

Two queries can be done simultaneously using two separate input wires, but only one writing wire is provided. Like register file, these services are done at any time and possible hazards could be resolved with a good arrangement.

2.4 ALU Reservation Station

A certain ALU reservation station could only do a certain kind of ALU operation, like adding or multiplying two numbers.

The ALU reservation station generally has two functions: fetch issue and do calculation.

In function fetch issue, the reservation station fetchs the instruction in the issue stage.

In function do calculation, the reservation station checks whether operands are ready; if so, the reservation station does corresponding calculation, and then sends a signal to reorder buffer controller for writing the result to common data bus.

2.5 Load Reservation Station

Like ALU Reservation Station, a load reservation station has two similar functions. A load reservation station is essentially an add reservation station, because we should calculate the offset in instructions like “lw $t0, 3($sp)”.

The difference is that each load reservation station has a permission, in other words, a set of wires connected to the data cache for reading data.

In function fetch issue, the reservation station fetchs the instruction in the issue stage.

In function do calculation, the reservation station checks whether the source address is ready; if so, the reservation station does add operation with the source address and offset, reads out data needed from cache, and then sends a signal to reorder buffer controller for writing the result to common data bus.

2.6 Store Reservation Station

Also like reservation station, a store reservation station has two similar functions. A store reservation station is inherently an add reservation station as well.

The difference is that store reservation station has three operands, for instructions like "sw $a, b($c)", $a, $c are two registers we are awaiting, and b is also an operand. Unlike load reservation station, any store reservation station has no access to either data cache or data memory. Any writing to data memory could only occur in commit stage.

In function fetch issue, the reservation station fetchs the instruction in the issue stage.

In function do calculation, the reservation station checks whether the target address is ready; if so, the reservation station does add operation with target address and offset, reads out data to be written in register file and common data bus, and then sends a signal to reorder buffer controller for writing the result back to cache.

2.7 Instruction Cache

Our instruction cache uses a set of input wires to read a given address and outputs the corresponding instruction after some time of delay. It does not support memory writing, because instructions could not be modified at runtime.

Thus, the cache only has one major function: memory read. In this function, the cache checks whether a cache miss occurs, waits for delay and returns the instruction.

2.8 Data Cache

Our data cache uses three set of input wires in order to read three addresses at the same time, and outputs the corresponding data after some time of delay. It also provides a set of wires for writing. Because the delay time is uncertain, we use wires readSuccess and writeSuccess to indicate whether the operation is done.

Thus, the cache has two major functions: memory read and memory write.

In function memory read, the cache checks whether a cache miss occurs, waits for delay and returns the corresponding data.

In function memory write, the cache checks whether a cache miss occurs, waits for delay and writes the corresponding address.

2.9 Reorder Buffer

Reorder buffer is the most complicated module in our CPU. For convenience and lower expense of wires, three stages out of four are executed in this module. Reorder buffer includes the following functions:

The function issue: Does the same thing as the issue stage in typical Tomasulo's Algorithm.

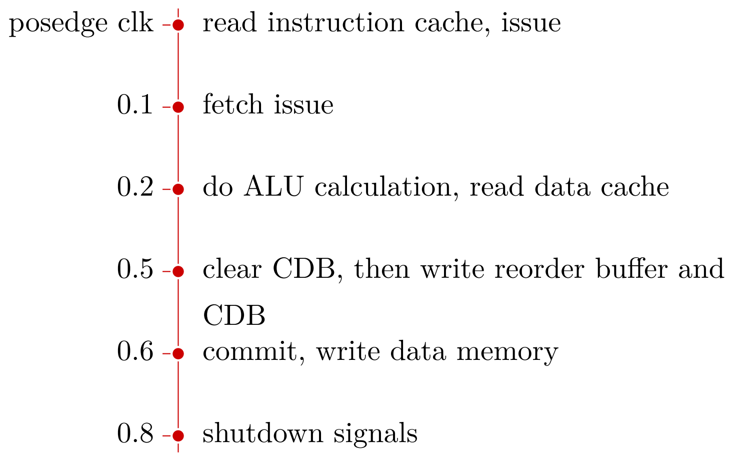
The function write buffer: Does the same thing as write back stage in typical Tomasulo's Algorithm.

The function commit: Does the same thing as commit stage in typical Tomasulo's Algorithm.

The function fetch instruction: Fetches the instruction from instruction cache.

3 Schedule Arrangement

Modules depend on each other, which means that hazards are easy to occur, such as reading and writing one register at the same time. Therefore, a robust schedule is keenly needed. The picture below is the timeline we use which resolves all scheduling hazards.


posedge clk
read instruction cache, issue
0.1
fetch issue
0.2
do ALU calculation, read data cache
0.5
clear CDB, then write reorder buffer and CDB
0.6
commit, write data memory
0.8
shutdown signals

4 Self-designed Instruction Set Architecture

The architecture we designed is a kind of MIPS-like ISA. As we all know, MIPS instructions are defined in three types, R-type, I-type and J-type. Our instructions are defined in two types: R-type and I-type. The two tables below describes instructions we support.

Instruction	Type	Opcode[31 : 26]	rd[25 : 21], rs[20 : 16], rt[15 : 11]
add	R	000000	\$rd = \$rs + \$rt
sub	R	000010	\$rd = \$rs - \$rt
mul	R	000100	\$rd = \$rs * \$rt
shl	R	001000	\$rd = \$rs >> \$rt
shr	R	001001	\$rd = \$rs << \$rt

Instruction	Type	Opcode[31 : 26]	rd[25 : 21], rs[20 : 16], immediate[15 : 0]
addi	I	000001	\$rd = \$rs + immediate
subi	I	000011	\$rd = \$rs - immediate
lw	I	000101	load memory address (\$rs + immediate) to \$rd
sw	I	000110	store \$rd to memory address (\$rs + immediate)
bge	I	000111	if \$rd >= \$rs, then PC = PC + immediate

There is another instruction halt(00101000000000000000000000000000) used for terminating the program.

5 Design Philosophy

5.1 From C++ to Verilog

As we all know, Verilog is a funny language, which supports nearly none feature of a high-level language. For this reason, coding in Verilog is a great challenge for normal programmers. In our philosophy, a simple restricted C++ program could be translated to Verilog.

In one sentence, an input of a method could be viewed as an input wire, and an outcome of a method could be considered as an output wire.

Take the C++ program below as an example:

```cpp
class Example {
        unsigned int a;
        unsigned int f() {
                return a;
        }
        void g(unsigned int c) {
                a = c;
        }
};
```

This can be translated to the Verilog module below:

```
module Example(fEnable, fReturn, gEnable, gC);
        input wire fEnable;
        output reg[31: 0] fReturn;
        input wire gEnable;
        input wire[31: 0] gC;

        reg[31: 0] a;

        always @ (posedge fEnable)
        begin: functionF
                fReturn = a;
        end

        always @ (posedge gEnable)
        begin: functionG
                a = gC;
        end

endmodule
```

5.2 How to Arrange a Schedule

Schedule arrangement can be abstracted into a typical graph scheduling problem.

If two functions cannot be operate at the same time, we add an edge in the interference graph between the two nodes representing these two functions. Although graph coloring is NP-Complete problem, in CPU design, the problem size is small enough to calculate even by human.

To avoid too much interference, in our CPU, we abide a philosophy that one module should be used by other modules as least as possible.

6 Collaboration Division

Task	Author
code	邵俊儒 & 陈昕昀
debug	陈昕昀
testbench	陈昕昀
assembler	邵俊儒
report	邵俊儒 & 陈昕昀

7 Acknowlegements

We first have to thank Prof. Xiaoyao Liang for bringing us such a wonderful course.

We also have to thank our TA, Ran Ye, for his hard work and warm devotion.

References

[1] Xiaoyao Liang. Slides of Computer System I.

[2] John L. Hennessy, David A. Patterson. Computer Architecture: A Quantitative Approach. 5th edition. Morgan Kaufmann Publisher.

[2] David A. Patterson, John L. Hennessy. Computer Organization and Design: The Hardware / Software Interface. 5th edition. Morgan Kaufmann Publisher.

[3] Deepak Kumar Tala. Verilog Tutorial. http://www.asic-world.com/verilog/veritut.html.