嵌入式系统结构与设计课程实验

实验报告 11

CPU Design (Fixing Control Hazard* and Data Hazard)

邱迪聪 / 11331262

2014/1/22

Content

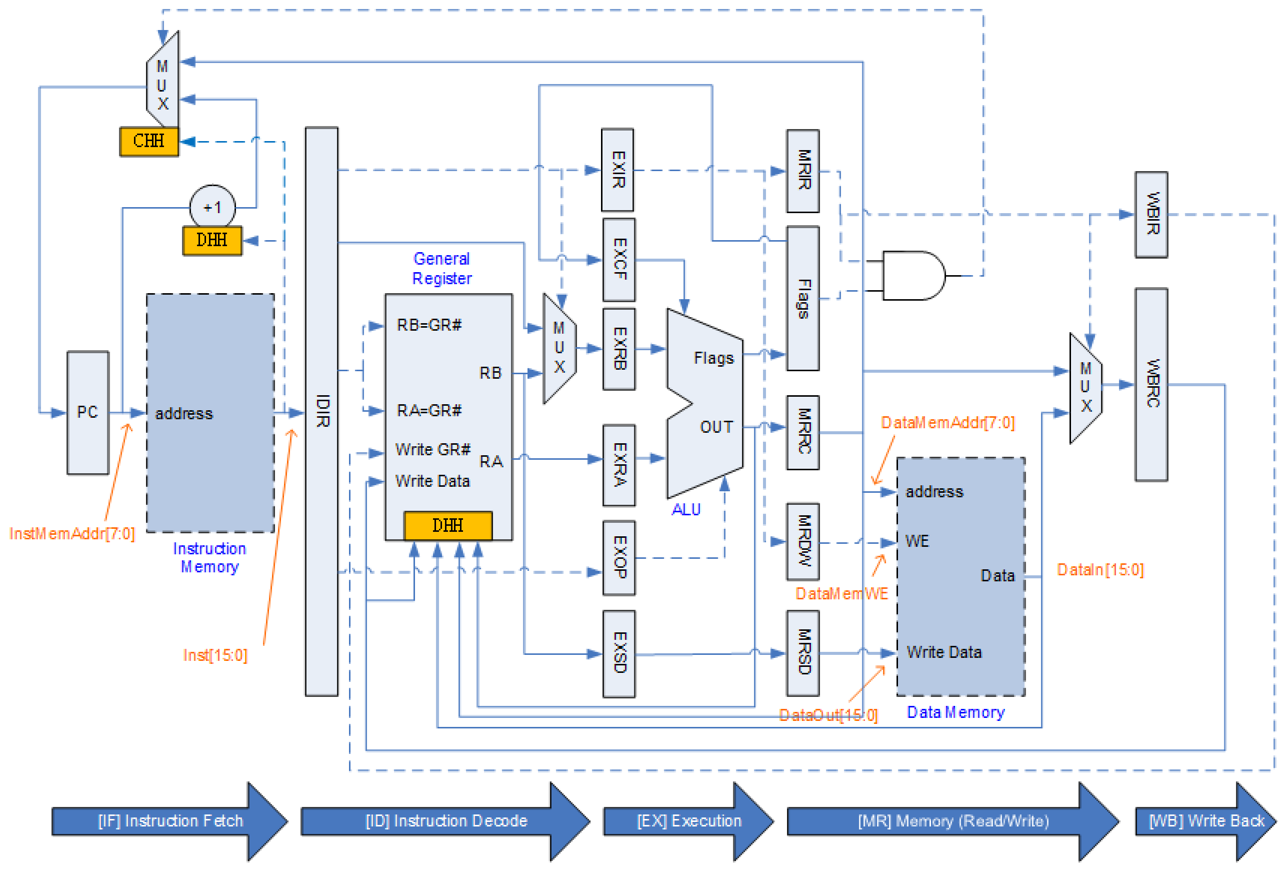
1. Target

- Design a CPU with control hazard* and data hazard fixed
- Further understand and use timing/area/power report to optimize the design

2. Arithmetic and Logic Unit

2.1 Design Overview

The latest design with hazard handling differs by appending two parts, the Data Hazard Handler (DHH) and the Control Hazard Handler (CHH). These two parts and their relations with the other components of the CPU can be seen from the following figure.



Initial version from Dr. Jun Wang.
Specific details improved by David QIU.

The Data Hazard Handler appears in the program counter and the operand-selector in the Instruction Decode stage. Meanwhile, the Control Hazard Handler appears only in the program counter.

Why hazards appear and methods to handle the hazards will be discussed in the following two part.

2.2 Data Hazard Handling

Data hazard happens because General Registers are not updated in time. In other words, data hazard occurs when one CPU instruction (e.g. Inst0), which requires updating a general register (e.g. GR0), is followed by another instruction (e.g. Inst1) that requires the altered register (GR0) as an input. When Inst0 reaches the Instruction Execution stage and is executed, it needs two more clocks for the calculation result to reach the Data Write-Back stage. And the following instruction, Inst1, is executed when GR0 is not updated, thus hazard occurs.

To handle this problem, this design uses data preview and PC stall techniques.

Data Preview: calculation result first comes out at the OUT port of the ALU, goes to MRRC, and finally WBRC. Meanwhile, if the instruction is LOAD, the data will come from data memory and then goes to WBRC. So in the Instruction Decode (ID) stage, the candidate operands RA and RB come from not only General Registers, but also ALUOUT, MRRC, DataMem and WBRC. If the instruction in stages EX, MR or WB updates any GR and the related GR number is the same as RA or RB, the latest matched preview data will replace the original GR data.

PC Stall: if the hazard instruction is LOAD, it is impossible to know the result until MR stage, which means there must be at least one hazard clock is needed. To make the result show up before the following instruction executed, this design stall PC once to make each LOAD instruction duplicated, thus one hazard clock is delayed to provide time for result from data memory appears. In implementation, the design introduces a register named PC Stall State Register, in order to count the duplication of LOAD instruction.

2.3 Control Hazard Handling *

Control hazard happens when branch instructions are executed. A branch instruction entering the CPU needs two clocks to reach EX stage and another clock to execute. So after execution of a branch instruction, three address-continuing instructions are loaded into the CPU before the instruction of branch address loaded. And if any of the three unexpected instructions alters the General Registers or is another branch instruction, the program logic will be different.

The handling method focus on how to avoid the three unexpected instructions affecting the system, and a simple idea is to replace those three instructions with NOP. So a 2-bit register is introduced to be a state machine that counts how many instructions are loaded into the CPU after one branch instruction. If a branch instruction is detected in IF stage, the state machine will enter the counting for three times. During the counting, PC will be held the same and the instruction loaded into the CPU will be replaced as NOP.

2.4 Simulation

Since any functional verification requires multiple lines of code, pasting the whole simulation here will be a mess, so the simulation of a classic case is selected as an example that shows Data Hazard and Control Hazard are both fixed.

The testing code is as followed.

```
DataIn <= 16'h9; // DataMem[0x0000] == 0x0009
Inst <= {`LOAD,  `AR0, `BR0, 4'h0}; #10; CLK <= 1; #10; CLK <= 0; // 0x0000 (D-Hazard)
Inst <= {`LDIL,  `AR1, 4'h0, 4'h2}; #10; CLK <= 1; #10; CLK <= 0; // 0x0001 (D-Hazard)
Inst <= {`ADD,   `AR5, `BR0, `BR1}; #10; CLK <= 1; #10; CLK <= 0; // 0x0002
Inst <= {`JUMP,  `AR0, 4'h2, 4'h0}; #10; CLK <= 1; #10; CLK <= 0; // 0x0003 (C-Hazard)
Inst <= {`SUB,   `AR0, `BR0, `BR1}; #10; CLK <= 1; #10; CLK <= 0; // 0x0004 (Unexpected
Inst <= {`SUB,   `AR0, `BR0, `BR1}; #10; CLK <= 1; #10; CLK <= 0; // 0x0005 Inst loaded
Inst <= {`SUB,   `AR0, `BR0, `BR1}; #10; CLK <= 1; #10; CLK <= 0; // 0x0006 into CPU)

Inst <= {`ADD,   `AR0, `BR0, `BR1}; #10; CLK <= 1; #10; CLK <= 0; // 0x0020 (Branched)
Inst <= {`STORE, `AR5, `BR2, 4'h0}; #10; CLK <= 1; #10; CLK <= 0; // 0x0021
Inst <= {`STORE, `AR0, `BR2, 4'h1}; #10; CLK <= 1; #10; CLK <= 0; // 0x0022
Inst <= {`NOP,   3'h0, 4'h0, 4'h0}; #10; CLK <= 1; #10; CLK <= 0; // 0x0023
Inst <= {`NOP,   3'h0, 4'h0, 4'h0}; #10; CLK <= 1; #10; CLK <= 0; // 0x0024
Inst <= {`NOP,   3'h0, 4'h0, 4'h0}; #10; CLK <= 1; #10; CLK <= 0; // 0x0025
Inst <= {`NOP,   3'h0, 4'h0, 4'h0}; #10; CLK <= 1; #10; CLK <= 0; // 0x0026
Inst <= {`NOP,   3'h0, 4'h0, 4'h0}; #10; CLK <= 1; #10; CLK <= 0; // 0x0027
Inst <= {`HALT,  3'h0, 4'h0, 4'h0}; #10; CLK <= 1; #10; CLK <= 0; // 0x0028
```

In this simulation, two data hazard types, the LOAD and calculation hazard, are both introduced, respectively at the first and the second lines. The forth line is a control hazard that branches to 0x0020 of instruction memory address, followed by three Subtraction instructions as unexpected instructions.

If data hazard is fixed, the operands should be 0x0009 and 0x0002 for the ADD instruction in the third line, whose result, hopefully 0x000b, will be outputted at the first STORE instruction of address 0x0021. And if the control hazard is fixed, the three SUB instructions should not be executed and only the ADD instruction at address 0x0020 is executed, whose result is also 0x000b hopefully and outputted at the second STORE instruction.

The simulation result is as followed.

```
CLK:InstMemAddr:Inst:DataMemAddr:DataIn:DataMemWE:DataOut
0   :00         :1000:00         :0009   :0        :0000
1   :01         :1000:xx         :0009   :0        :0000
0   :01         :b902:xx         :0009   :0        :0000
1   :02         :b902:00         :0009   :0        :0000
0   :02         :8501:00         :0009   :0        :0000
1   :03         :8501:00         :0009   :0        :0000
```

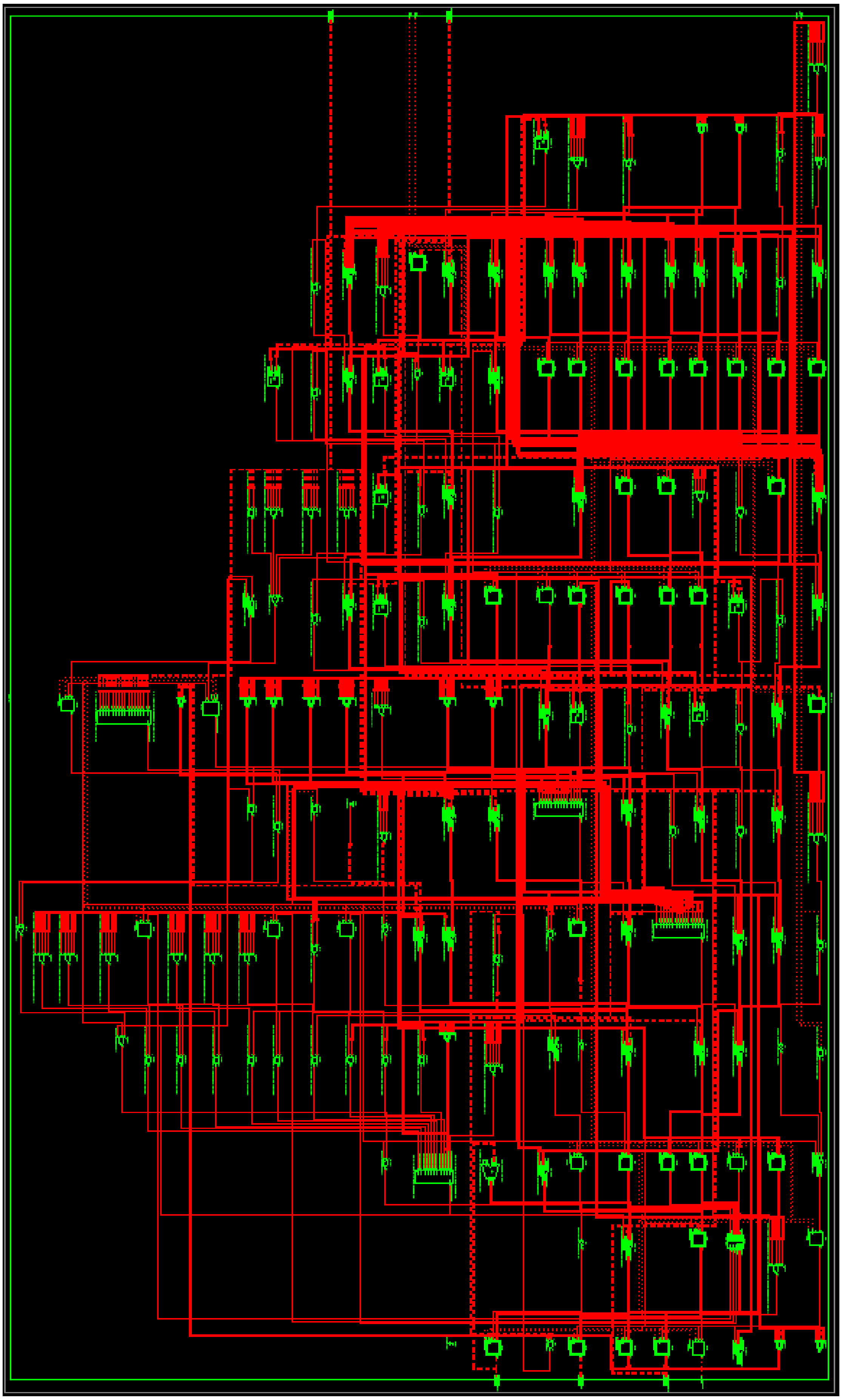
```
0   :03          :3020:00          :0009   :0          :0000
1   :04          :3020:02          :0009   :0          :0000
0   :04          :9801:02          :0009   :0          :0000
1   :04          :9801:0b          :0009   :0          :0000
0   :04          :9801:0b          :0009   :0          :0000
1   :04          :9801:20          :0009   :0          :0000
0   :04          :9801:20          :0009   :0          :0000
1   :20          :9801:12          :0009   :0          :0000
0   :20          :8001:12          :0009   :0          :0000
1   :21          :8001:12          :0009   :0          :0000
0   :21          :1d20:12          :0009   :0          :0000
1   :22          :1d20:12          :0009   :0          :0000
0   :22          :1821:12          :0009   :0          :0000
1   :23          :1821:0b          :0009   :0          :0000
0   :23          :0000:0b          :0009   :0          :0000
1   :24          :0000:00          :0009   :1          :000b   // Fist result (Correct!)
0   :24          :0000:00          :0009   :1          :000b
1   :25          :0000:01          :0009   :1          :000b   // Second result (Correct!)
0   :25          :0000:01          :0009   :1          :000b
1   :26          :0000:16          :0009   :0          :0000
0   :26          :0000:16          :0009   :0          :0000
1   :27          :0000:16          :0009   :0          :0000
0   :27          :0000:16          :0009   :0          :0000
1   :28          :0000:16          :0009   :0          :0000
0   :28          :0800:16          :0009   :0          :0000
1   :29          :0800:16          :0009   :0          :0000
0   :29          :0800:16          :0009   :0          :0000
```

The correct simulation result shows that data and control hazards are fixed in the design.

2.5 Synthesization and RTL diagram

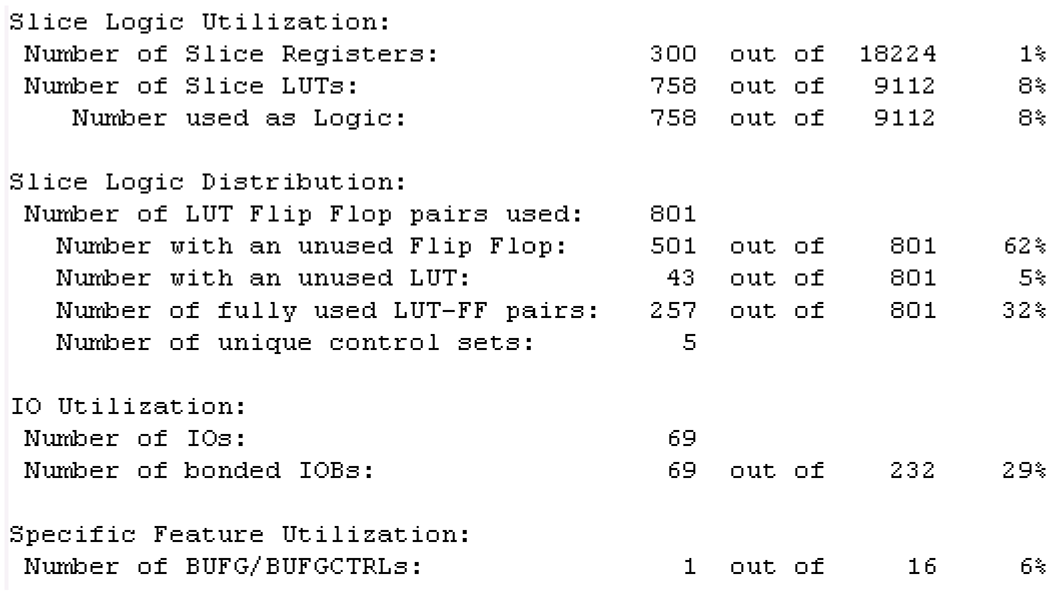
After Synthesization RTL diagram of the design is generated.

The RTL diagram of the core module, the `DCPU` module, generated from Synthesization is as below.



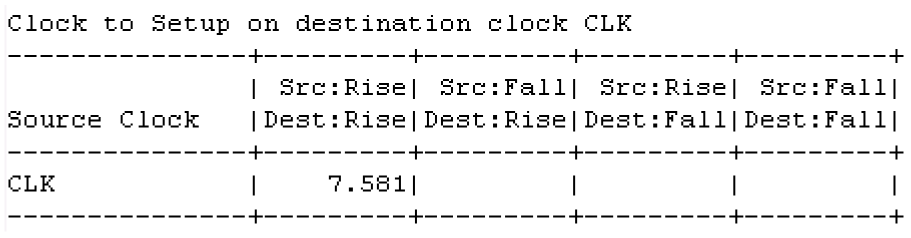
2.6 Design Summary

Device utilization summary:



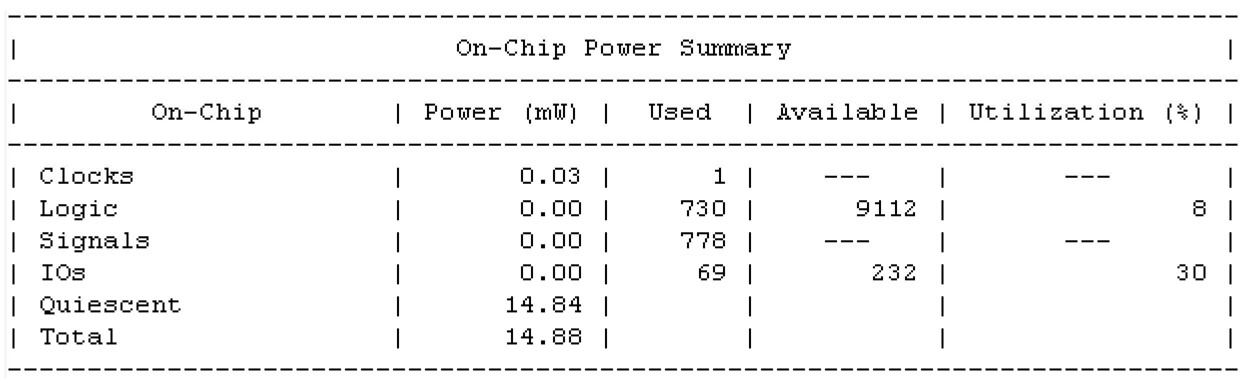
```
Slice Logic Utilization:
 Number of Slice Registers:            300  out of  18224     1%
 Number of Slice LUTs:                 758  out of   9112     8%
    Number used as Logic:              758  out of   9112     8%

Slice Logic Distribution:
 Number of LUT Flip Flop pairs used:   801
   Number with an unused Flip Flop:    501  out of    801    62%
   Number with an unused LUT:           43  out of    801     5%
   Number of fully used LUT-FF pairs:  257  out of    801    32%
   Number of unique control sets:        5

IO Utilization:
 Number of IOs:                         69
 Number of bonded IOBs:                 69  out of    232    29%

Specific Feature Utilization:
 Number of BUFG/BUFGCTRLs:               1  out of     16     6%
```

Timing Report:



```
Clock to Setup on destination clock CLK
---------------+---------+---------+---------+---------+
               | Src:Rise| Src:Fall| Src:Rise| Src:Fall|
Source Clock   |Dest:Rise|Dest:Rise|Dest:Fall|Dest:Fall|
---------------+---------+---------+---------+---------+
CLK            |    7.581|         |         |         |
---------------+---------+---------+---------+---------+
```

Power Report:



```
-----------------------------------------------------------------------------
|                          On-Chip Power Summary                            |
-----------------------------------------------------------------------------
|        On-Chip        | Power (mW) |  Used  | Available | Utilization (%) |
-----------------------------------------------------------------------------
| Clocks                |       0.03 |      1 |    ---    |       ---       |
| Logic                 |       0.00 |    730 |      9112 |               8 |
| Signals               |       0.00 |    778 |    ---    |       ---       |
| IOs                   |       0.00 |     69 |       232 |              30 |
| Quiescent             |      14.84 |        |           |                 |
| Total                 |      14.88 |        |           |                 |
-----------------------------------------------------------------------------
```

Afterthought

This task finally comes to an end. This semester showed me a lot of fantasy, and I've also learnt a lot of chip designing techniques.

Initially, the idea of optimize CPU and ALU design by coupling the two parts is just simple. But as time goes on, I discovered such task is not as easy as one might ever think about. The CPU instruction and ALU operation code needs carefully distributing, otherwise either instruction field will be insufficient or this optimization idea actually add up the complexity of the design.

Fortunately, I finally make it successfully.

And the second part of this design is ridding the hazard. The task requirement is data hazard elimination, but I design to eliminate the control hazard as well. Of course, the beginning is a tough experience; especially I am doing all this so lonely at accommodation when most of my classmates have been home warmly. But things went easier when I thought a lot about it.

And finally, I make it again.

As I mentioned, I thus had to wait for the final assignment of our CPU design, before I could start to build my ALU. What's more, I consider this project an important one, so I pushed it as a currently not visible open source project. And once I finish this design and submit it, I will open the public access to it. So I make it open today, and all the details including designing process record and history can be found in the project home page.

The project's website is: http://www.osysu.org/davidqiu/D-CPU

David Qiu (邱迪聪)

2014.1.22

Appendix 1: Attachment List

1. DCPU

    \ DCPU.v

    \ DCPU_test01.v

    \ ALU.v

D-CPU Design Specifications

    \ D-CPU Design Specifications.xlsx

D-CPU Operation Field Definition

    \ D-CPU Operation Field Definition.png

    \ D-CPU Operation Field Definition.vsdx

D-CPU Pipeline Architecture

    \ D-CPU Pipeline Architecture.png

    \ D-CPU Pipeline Architecture.vsdx

X.v files are primary code files; X_test#.v files are test files.