嵌入式系统结构与设计课程实验

实验报告 10

CPU Design (with hazard)

邱迪聪 / 11331262

2014/1/19

Content

1. Target

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

2. Arithmetic and Logic Unit

2.1 Designing

The most important step is to design the available instruction list for CPU. By following the instruction list, actual CPU implementation is much easier.

As mentioned in the last report, since the available operations of ALU are directly affected what the CPU can execute, and the optimization of the ALU is related to the CPU requirements, I have optimized the association interfaces between CPU and ALU. The detailed optimization method will be discussed in the next part.

The available CPU instructions are as followed.

(See "Appendix 2: CPU Instruction List" or the attachment "D-CPU Instruction List.png")

The CPU provides different kinds of instructions: Data Transfer Instructions, Control Instructions, Arithmetic Instructions, Logic Instructions and Shift Instructions.

A pipeline design of CPU departs it into 5 stages: Instruction Fetch, Instruction Decode, Execution, Memory Read/Write, and Data Write-Back. Full CPU pipeline architecture can be seen below.

(See Appendix 3: D-CPU Pipeline Architecture)

2.2 Improvement *

The interface is usually a MUX (multiplexer) between an ALU and a CPU, which is an optional optimization.

In my design, the CPU instructions have deliberatively designed overlap with the ALU operations. These overlap happens in the higher coding area (10000 ~ 11111). This area is mainly the arithmetic and logic parts, where the ALU's multiple functions work.

So just by pulling all the arithmetic and logic functions to a higher coding area, it is possible just to drop the CPU's instruction prefix "1" and leave the remaining as the ALU's operation code. After doing this, the multiplexer is much simplified merely to deal with about half of the translation work compared to the original design.

The overlapped CPU instruction encoding and ALU operation encoding is as followed.

		Inst[15:11]			
Category	Mnemonic	Opcode	CALL		
	BS	01111	ALU_ADDC		
Arithmetic	ADD	10000	ALU_ADDC	0101	
	ADDI	10001	ALU_ADDC	0101	
	ADDC	10010	ALU_ADDC	0010	
	SUB	10011	ALU_SUBB	0101	10XXX
	SUBI	10100	ALU_SUBB	0101	
	SUBB	10101	ALU_SUBB	0101	
	INC	10110	ALU_INC	0110	
	CMP	10111	ALU_CMP	0111	
Control	BE	11000	ALU_ADDC	1000	
Logic	XOR	11001	ALU_XOR	1001	
	AND	11010	ALU_AND	1010	
	OR	11011	ALU_OR	1011	11XXX
Shift	SLL	11100	ALU_SLL	1100	
	SLA	11101	ALU_SLA	1101	
	SRL	11110	ALU_SRL	1110	
	SRA	11111	ALU_SRA	1111	

2.3 Simulation

Since hazard exists, any functional verification requires multiple lines of code. To paste the whole simulation here will be a mess, so the simulation of “Addition” function is selected as an example that shows the CPU works.

The testing code is as followed.

```
Inst <= {`LDIL,   `AR0, 4'h0, 4'h1}; #10; CLK <= 1; #10; CLK <= 0; // Load 01 to R0
Inst <= {`LDIL,   `AR1, 4'h0, 4'h2}; #10; CLK <= 1; #10; CLK <= 0; // Load 02 to R1
Inst <= {`NOP,    3'h0, 4'h0, 4'h0}; #10; CLK <= 1; #10; CLK <= 0; // Hazard
Inst <= {`NOP,    3'h0, 4'h0, 4'h0}; #10; CLK <= 1; #10; CLK <= 0;
Inst <= {`NOP,    3'h0, 4'h0, 4'h0}; #10; CLK <= 1; #10; CLK <= 0;

Inst <= {`ADD,    `AR0, `BR0, `BR1}; #10; CLK <= 1; #10; CLK <= 0; // R0 <- R0 + R1
Inst <= {`NOP,    3'h0, 4'h0, 4'h0}; #10; CLK <= 1; #10; CLK <= 0; // Hazard
Inst <= {`NOP,    3'h0, 4'h0, 4'h0}; #10; CLK <= 1; #10; CLK <= 0;
Inst <= {`NOP,    3'h0, 4'h0, 4'h0}; #10; CLK <= 1; #10; CLK <= 0;

Inst <= {`STORE,  `AR0, `BR2, 4'h0}; #10; CLK <= 1; #10; CLK <= 0; // Output R0
Inst <= {`NOP,    3'h0, 4'h0, 4'h0}; #10; CLK <= 1; #10; CLK <= 0;
Inst <= {`NOP,    3'h0, 4'h0, 4'h0}; #10; CLK <= 1; #10; CLK <= 0;
Inst <= {`NOP,    3'h0, 4'h0, 4'h0}; #10; CLK <= 1; #10; CLK <= 0;
Inst <= {`NOP,    3'h0, 4'h0, 4'h0}; #10; CLK <= 1; #10; CLK <= 0;
Inst <= {`NOP,    3'h0, 4'h0, 4'h0}; #10; CLK <= 1; #10; CLK <= 0;
Inst <= {`HALT,   3'h0, 4'h0, 4'h0}; #10; CLK <= 1; #10; CLK <= 0;
```

The simulation result is as followed.

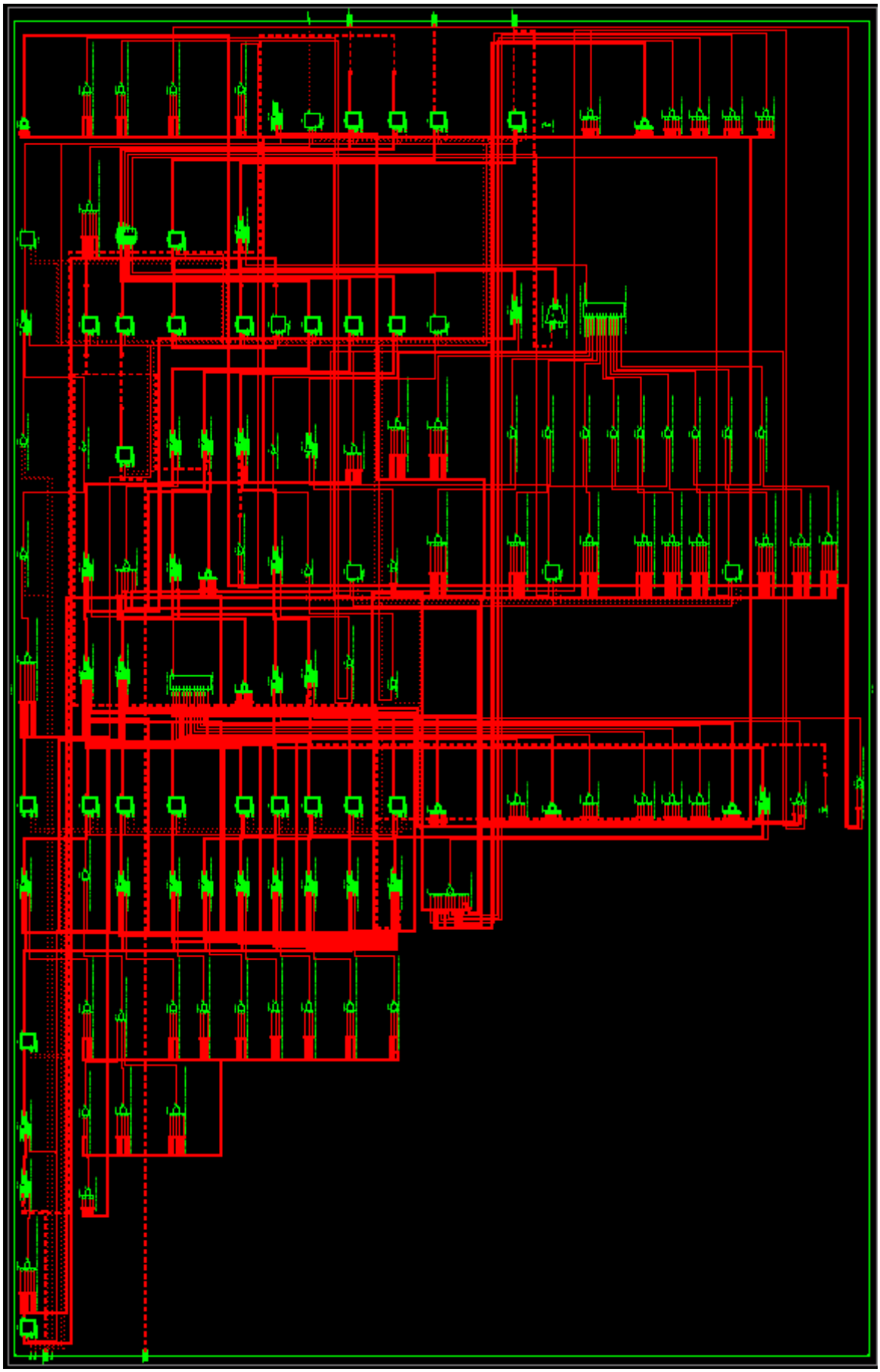
```
CLK:InstMemAddr:Inst:DataMemAddr:DataIn:DataMemWE:DataOut
0  :01         :2001:xx         :0000  :0        :0000   // LDIL R0 0 1
1  :02         :2001:00         :0000  :0        :0000
0  :02         :2102:00         :0000  :0        :0000   // LDIL R1 0 2
1  :03         :2102:00         :0000  :0        :0000
0  :03         :0000:00         :0000  :0        :0000   // NOP
1  :04         :0000:01         :0000  :0        :0000
0  :04         :0000:01         :0000  :0        :0000   // NOP
1  :05         :0000:02         :0000  :0        :0000
0  :05         :0000:02         :0000  :0        :0000   // NOP
1  :06         :0000:00         :0000  :0        :0000
0  :06         :8001:00         :0000  :0        :0000   // ADD R0 R0 R1
1  :07         :8001:00         :0000  :0        :0000
0  :07         :0000:00         :0000  :0        :0000   // NOP
1  :08         :0000:00         :0000  :0        :0000
0  :08         :0000:00         :0000  :0        :0000   // NOP
1  :09         :0000:03         :0000  :0        :0000
0  :09         :0000:03         :0000  :0        :0000   // NOP
1  :0a         :0000:00         :0000  :0        :0000
0  :0a         :1820:00         :0000  :0        :0000   // STORE R0 R2 0
1  :0b         :1820:00         :0000  :0        :0000   // (MemAddr = R2+0 = 0)
0  :0b         :0000:00         :0000  :0        :0000   // NOP
1  :0c         :0000:00         :0000  :0        :0000
0  :0c         :0000:00         :0000  :0        :0000   // NOP
1  :0d         :0000:00         :0000  :1        :0003   // Result: R0 = 3 (Correct)
0  :0d         :0000:00         :0000  :1        :0003   // NOP
1  :0e         :0000:00         :0000  :0        :0000
0  :0e         :0000:00         :0000  :0        :0000   // NOP
1  :0f         :0000:00         :0000  :0        :0000
0  :0f         :0000:00         :0000  :0        :0000   // NOP
1  :10         :0000:00         :0000  :0        :0000
0  :10         :0800:00         :0000  :0        :0000   // HALT
1  :11         :0800:00         :0000  :0        :0000
0  :11         :0800:00         :0000  :0        :0000
```

The simulation result shows that addition function is correctly designed.

2.4 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.



2.5 Design Summary

Device utilization summary:

```
Slice Logic Utilization:
 Number of Slice Registers:              290  out of  18224      1%
 Number of Slice LUTs:                   600  out of   9112      6%
    Number used as Logic:                600  out of   9112      6%

Slice Logic Distribution:
 Number of LUT Flip Flop pairs used:     655
   Number with an unused Flip Flop:      365  out of    655     55%
   Number with an unused LUT:             55  out of    655      8%
   Number of fully used LUT-FF pairs:    235  out of    655     35%
   Number of unique control sets:          4

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:

```
Cross Clock Domains 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            |    7.934|         |         |         |
---------------+---------+---------+---------+---------+
```

Power Report:

```
------------------------------------------------------------------------------------
|                             On-Chip Power Summary                                |
------------------------------------------------------------------------------------
|        On-Chip        | Power (mW) |  Used  | Available | Utilization (%) |
------------------------------------------------------------------------------------
| Clocks                |       0.04 |      1 |    ---    |       ---       |
| Logic                 |       0.00 |    594 |      9112 |               7 |
| Signals               |       0.00 |    629 |    ---    |       ---       |
| IOs                   |       0.00 |     69 |       232 |              30 |
| Quiescent             |      14.84 |        |           |                 |
| Total                 |      14.88 |        |           |                 |
------------------------------------------------------------------------------------
```

Afterthought

The design of CPU is quite an art. It combines theories, techniques and knowledge. And yet the worse news is CPU gets these things more complex because it is a system that can hardly be divided into subparts. That means the CPU is exact a highly coupling module, one has to think hard to optimize the design until the final product comes out.

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 so 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.

To build an experimental, or says example, ALU is easy with the guidance from our teacher. However, what I want to build is an ALU fit and optimized for the later CPU design, which means these two designs is highly coupling but quite optimized that they seem to be exactly one piece.

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.

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

I wanted to optimize this design as far as possible, but the payment is time. I am so sorry that I have to delay this until now, after the final exams. But the outcome is not so bad, because merely all my initial expectations finally come true in my design.

David Qiu (邱迪聪)

2014.1.19

Appendix 1: Attachment List

1. DCPU

    \ DCPU.v

    \ DCPU_test01.v

    \ ALU.v

D-CPU Instruction List.png

D-CPU Pipeline Architecture.png

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

Appendix 2: CPU Instruction List

Central Processer Unit Instruction List

Category	Mnemonic	Inst[15:11] Opcode	Inst[10:8] Operand1 (3 bits)	Inst[7:4] Operand2 (4 bits)	Inst[3:0] Operand3 (4 bits)	Full Name	Operation
General	NOP	00000	/	/	/	NO OPERATION	no operation
	HALT	00001	/	/	/	HALT	halt
Data Transfer	LOAD	00010	r1	r2	val3	LOAD	gr[r1] <- m[r2+val3]
	STORE	00011	r1	r2	val3	STORE	m[r2+val3] <- r1
	LDIL	00100	r1	val2	val3	LOAD IMMEDIATE LOW	r1 <- {val2, val3}
	LDIH	00101	r1	val2	val3	LOAD IMMEDIATE HIGH	r1 <- {val2, val3, 0000_0000} (lower 8'b0 can be given with ADDI)
Control	JUMP	00110	/	val2	val3	JUMP	jump to {val2, val3}
	JMPR	00111	r1	val2	val3	JUMP REGISTER	jump to r1+{val2, val3}
	BZ	01000	r1	val2	val3	BRANCH ZERO	if ZF=1 branch to r1+{val2, val3}
	BNZ	01001	r1	val2	val3	BRANCH NOT ZERO	if ZF=0 branch to r1+{val2, val3}
	BN	01010	r1	val2	val3	BRANCH NEGATIVE	if NF=1 branch to r1+{val2, val3}
	BNN	01011	r1	val2	val3	BRANCH NOT NEGATIVE	if NF=0 branch to r1+{val2, val3}
	BC	01100	r1	val2	val3	BRANCH CARRY	if CF=1 branch to r1+{val2, val3}
	BNC	01101	r1	val2	val3	BRANCH NOT CARRY	if CF=0 branch to r1+{val2, val3}
	BB	01110	r1	val2	val3	BRANCH BIGGER	if ZF=0 & NF=0 branch to r1+{val2, val3}
	BS	01111	r1	val2	val3	BRANCH SMALLER	if ZF=0 & NF=1 branch to r1+{val2, val3}
Arithmetic	ADD	10000	r1	r2	r3	ADD	r1 <- r2+r3
	ADDI	10001	r1	val2	val3	ADD IMMEDIATE	r1 <- r1+{val2, val3}
	ADDC	10010	r1	r2	r3	ADD CARRY	r1 <- r2+r3+CF
	SUB	10011	r1	r2	r3	SUBTRACT	r1 <- r2-r3
	SUBI	10100	r1	val2	val3	SUBTRACT IMMEDIATE	r1 <- r1-{val2, val3}
	SUBB	10101	r1	r2	r3	SUBTRACT BORROW	r1 <- r2-r3-CF
	INC	10110	r1	r2	/	INCREASE	r1 <- r2 + 1
	CMP	10111	/	r2	r3	COMPARE	r2-r3; set CF,ZF and NF
Control	BE	11000	r1	val2	val3	BRANCH EQUAL	if ZF=1 branch to r1+{val2, val3}
Logic	XOR	11001	r1	r2	r3	EXCLUSIVE OR	r1 <- r2 XOR r3
	AND	11010	r1	r2	r3	AND	r1 <- r2 AND r3
	OR	11011	r1	r2	r3	OR	r1 <- r2 OR r3
Shift	SLL	11100	r1	r2	val3	SHIFT LEFT LOGICAL	r1 <- r2 shift left logical (val3 bit shift)
	SLA	11101	r1	r2	val3	SHIFT LEFT ARITHMETICAL	r1 <- r2 shift left arithmetical (val3 bit shift)
	SRL	11110	r1	r2	val3	SHIFT RIGHT LOGICAL	r1 <- r2 shift right logical (val3 bit shift)
	SRA	11111	r1	r2	val3	SHIFT RIGHT ARITHETICAL	r1 <- r2 shift right arithmetical (val3 bit shift)

Appendix 3: D-CPU Pipeline Architecture