CSE-COA-LAB-003

计算机组成与系统实验

马叶晟[1] [2]

5140209064

2016 年 4 月 6 日

[1] email: kimi.ysma@gmail.com

[2] 上海交通大学，计算机科学与工程系.

1 实验介绍

1.1 实验名称

简单的类MIPS单周期处理器的实现：控制单元与运算单元.

1.2 实验目的

理解CPU和ALU的原理.

1.3 实验范围

本次实验将覆盖以下范围：

1. ISE 13.4的使用
2. 使用VerilogHDL进行逻辑设计
3. 编辑UCF
4. CPU和ALU的实现

2 MIPS处理器概述

2.1 MIPS处理器简介

MIPS是精简指令集(RISC)的一种代表，它是80年代最热门的RISC芯片，也是第一种商用的RISC芯片. MIPS的意思是"无内部互锁流水级的微处理器"(Microprocessor without interlocked piped stages), MIPS 同样也可以被解释为million instructions per second, 也代表了MIPS是一种运行十分快的指令集. 其机制是尽量利用软件办法避免流水线中的数据相关问题. 它最早是在80年代初期, 由斯坦福大学Hennessy教授领导的研究小组制出来.

2.2 MIPS指令结构

MIPS是一种RISC架构的指令集，所有的指令都是32位长的指令. 所有的指令被分为三种格式:

- R-type: opcode, rs, rt, rd, shamt, funct
- I-type: op, rs, rt, immediate
- J-type: op, address

MIPS会根据不同的指令执行相应的操作，对于各种类型的指令，译码的过程如下：

- **R-type**: OpCode传入Ctr模块，译码产生各种控制信号. Rs, rt, rd为相应的读写寄存器. Funct域译码生成运算器控制信号ALUCtr.

- **I-type**: I-type的opCode的译码与R-type类似，而immediate域通过符号扩展成为运算数.
- **J-type**: address与PC的低四位连接而成，生成一个32位的地址用于跳转.

3 实验内容

3.1 主控制单元模块Ctr的实现

3.1.1 主控制单元概述

由于MIPS采用了RISC架构，因此这给我们主控制单元的实现带来了很大的方便. MIPS的规整的指令集，使得我们能够轻松地通过opCode译码得到一系列的控制信号，这些控制信号使得CPU其余各个模块可以正常运行.

3.1.2 主控制单元模块的端口

主控制单元的输入端口是: opCode.
主控制单元的输出端口是:RegDst, Branch, MemRead, MemToReg, ALUOp, MemWrite, ALUSrc, RegWrite.

3.1.3 主控制单元的译码

对于简化的MIPS指令集, 其译码表如下所示:

input or output	signal name	R-type	lw	sw	beq
inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemToReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	0	1	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp2	0	0	0	1

表 1: Decoding Table for Simplified MIPS ISA

特别地，对于Jump指令而言，其opCode值为000010. R型指令包括add，sub，and，or，slt. I型指令包括lw，swm beq. J型指令包括j. 根据这个译码表即可写出相应的Verilog代码, 注意always块和default的使用. 具体代码见附录一, 主控制单元模块的实现比较简单, 就不赘述了.

3.2 ALU控制单元模块ALUCtr的实现

3.2.1 ALUCtr模块概述

算术运算控制单元ALUCtr通过主控制模块的ALUOp控制信号判断指令的类型, 通过指令后六位的fucnt字段区分R型指令.

3.2.2 算术运算单元的端口

主控制单元的输入端口是: ALUOp, funct.
主控制单元的输出端口是:ALUCtr.

3.2.3 算术运算控制模块的译码

通过ALUOp和funct来得到ALUCtr信号的译码表如下所示:

opCode	ALUOp	funct	ALUop	ALUCtr
lw	00	XXXXXX	add	0010
sw	00	XXXXXX	add	0010
beq	01	XXXXXX	sub	0110
R-type	10	100000	add	0010
R-type	10	100010	sub	0110
R-type	10	100100	and	0000
R-type	10	100101	or	0001
R-type	10	101010	slt	0111

表 2: Decoding Table for ALU Control

注意到，在ALUCtr的译码中，出现了若干个未定项，对于这些未定项不能再使用传统的case语句, 而应该使用包含未定项的语句casex语句. 由于这里实现的MIPS指令集并不是完整的，因此事实上很多指令上的设计存在浪费.

ALUCtr的代码实现在附录二中.

3.3 算术运算单元模块ALU的实现

3.3.1 ALU模块概述

算术运算单元ALU根据之前完成的ALUCtr的控制信号，对输入信号input1和input2做相应的算术

运算，并得出结果ALURes

3.3.2 算术运算单元ALU的端口

算术运算单元的输入端口是: input1, input2, ALUCtr. 算术运算单元的输出端口是: zero, ALURes.

3.3.3 算术运算单元模块的译码

算术运算单元ALU的功能很简单，通过传入的控制信号ALUCtr对两个输入的数据进行处理，并输出算术运算结果以及结果是否为0(为跳转和条件跳转设计). 算术运算单元的译码表如下所示:

ALU control lines	Function
0000	AND
0001	OR
0110	add
0111	set on less than
1100	NOR

表 3: Decoding Table for ALU

ALU模块的实现也比较简单, 主要的组成部分就是一个always块. 具体的代码实现见附录3.

3.4 实验仿真

3.4.1 主控制单元模块Ctr的仿真

在Ctr模块的仿真中，我使用的测试样例如下所示:

```
1     #100 opCode = 6'b000010;  // jump
2     #100 opCode = 6'b000000;  // R type
3     #100 opCode = 6'b100011;  // lw
4     #100 opCode = 6'b101011;  // sw
5     #100 opCode = 6'b000100;  // beq
```

实验仿真的波形图如下所示：

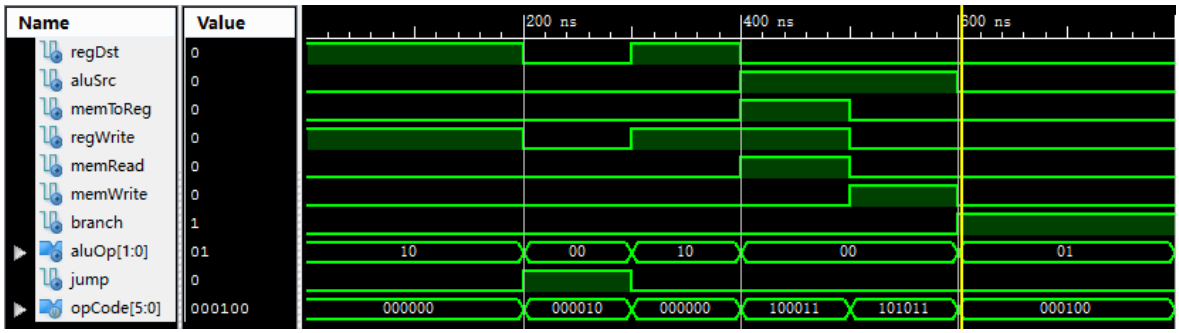

图 1: Ctr模块测试波形图

3.4.2 算术运算控制单元ALUCtr的仿真

在ALUCtr模块的仿真中，我使用的测试样例如下所示:

```
1     #100 {aluOp, funct} = 8'b00xxxxxx;
2     #100 {aluOp, funct} = 8'bx1xxxxxx;
3     #100 {aluOp, funct} = 8'b1xxx0000;
4     #100 {aluOp, funct} = 8'b1xxx0010;
5     #100 {aluOp, funct} = 8'b1xxx0100;
6     #100 {aluOp, funct} = 8'b1xxx0101;
7     #100 {aluOp, funct} = 8'b1xxx1010;
```

实验仿真的波形图如下所示：

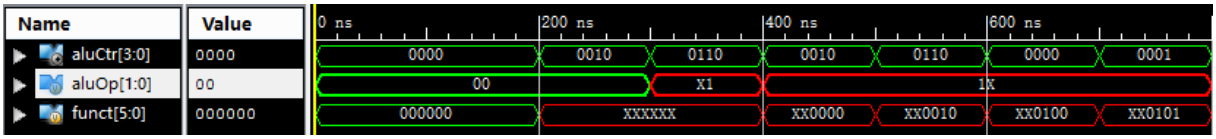

图 2: ALUCtr模块测试波形图

3.4.3 算术运算单元ALU的仿真

在ALU模块的仿真中，我使用的测试样例如下所示:

```
1     #100 aluCtr = 4'b0000;  // and
2          input1 = 32'b11111111111111110000000000000000;
3          input2 = 32'b00000000000000001111111111111111;
4
5     #100 aluCtr = 4'b0001;  // or
6          input1 = 32'b10000000000000000000000000000000;
7          input2 = 32'h00000000000000000000000000000000;
8
9     #100 aluCtr = 4'b0010;  // add
10         input1 = 1;
11         input2 = 7;
12
13    #100 aluCtr = 4'b0110;  // sub
14         input1 = 32;
```

```
15          input2 = 16;
16
17      #100 aluCtr = 4'b0110;
18          input1 = 1;
19          input2 = 1;
20
21      #100 aluCtr = 4'b0111;  // set on less than
22          input1 = 512;
23          input2 = 511;
24
25      #100 aluCtr = 4'b0111;
26          input1 = 0;
27          input2 = 4;
28
29      #100 aluCtr = 4'b1100;  // NOR
30          input1 = 0;
31          input2 = 1;
```

实验仿真的波形图如下所示：

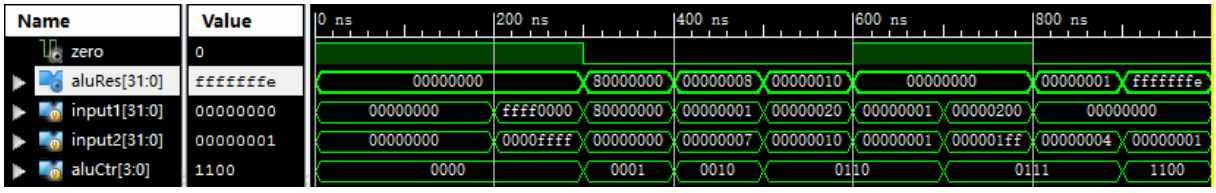

图 3: ALU模块测试波形图

4 实验心得体会

这次实验相比于之前的实验稍微复杂了一些，也是我们之后要完成的MIPS处理器中的一部分. 在主控制单元模块Ctr、算术运算控制单元ALUCtr和算术运算单元ALU的实现中，我们做的工作总体来说还是比较机械的，即简单地实现了相应的输入输出端口的设置和相关的译码工作. 然而，虽然这些工作都很简单，在其中我们学习到的module的实现和实例化方法、if,case等语句都为我们之后实现完整的CPU打下了基础. 特别是VerilogHDL特有的always块等语法结构，对于我们这些只学过C,C++等高级语言却从未接触过硬件语言的学生来说，理解上还是存在一些困难. 在理解了这些硬件编程语言特有的语法后，我们之后从事较为复杂的多周期流水线CPU的设计时就能轻松很多了.

附录1

主控制模块的Verilog代码实现:

```
module Ctr(
    opCode,
    regDst,
    aluSrc,
    memToReg,
    regWrite,
    memRead,
    memWrite,
    branch,
    aluOp,
    jump
    );

    input [5:0] opCode;
    output regDst;
    output aluSrc;
    output memToReg;
    output regWrite;
    output memRead;
    output memWrite;
    output branch;
    output [1:0] aluOp;
    output jump;
    reg regDst;
    reg aluSrc;
    reg memToReg;
    reg regWrite;
    reg memRead;
    reg memWrite;
    reg branch;
    reg [1:0] aluOp;
    reg jump;

    always @(opCode)
    begin
    case (opCode)
        6'b000010:  // jump
        begin
            regDst = 0;
            aluSrc = 0;
            memToReg = 0;
            regWrite = 0;
            memRead = 0;
            memWrite = 0;
            branch = 0;
            aluOp = 2'b00;
            jump = 1;
        end
        6'b000000:  // R type
        begin
            regDst = 1;
            aluSrc = 0;
            memToReg = 0;
            regWrite = 1;
            memRead = 0;
            memWrite = 0;
            branch = 0;
```

```
58              aluOp = 2'b10;
59              jump = 0;
60          end
61          6'b100011:  // lw
62          begin
63              regDst = 0;
64              aluSrc = 1;
65              memToReg = 1;
66              regWrite = 1;
67              memRead = 1;
68              memWrite = 0;
69              branch = 0;
70              aluOp = 2'b00;
71              jump = 0;
72          end
73          6'b101011:  // sw
74          begin
75              regDst = 0;
76              aluSrc = 1;
77              memToReg = 0;
78              regWrite = 0;
79              memRead = 0;
80              memWrite = 1;
81              branch = 0;
82              aluOp = 2'b00;
83              jump = 0;
84          end
85          6'b000100:  // beq
86          begin
87              regDst = 0;
88              aluSrc = 0;
89              memToReg = 0;
90              regWrite = 0;
91              memRead = 0;
92              memWrite = 0;
93              branch = 1;
94              aluOp = 2'b01;
95              jump = 0;
96          end
97          default:
98          begin
99              regDst = 0;
100             aluSrc = 0;
101             memToReg = 0;
102             regWrite = 0;
103             memRead = 0;
104             memWrite = 0;
105             branch = 0;
106             aluOp = 2'b00;
107             jump = 0;
108         end
109     endcase
110     end
111 endmodule
```

附录2

算术运算单元控制模块的Verilog代码实现：

```
module aluCtr(
    aluOp,
    funct,
    aluCtr
    );

    input [1:0] aluOp;
    input [5:0] funct;
    output [3:0] aluCtr;

    reg [3:0] aluCtr;

    always @ (aluOp or funct)
    casex ({aluOp, funct})
        8'b00xxxxxx: aluCtr = 4'b0010;
        8'bx1xxxxxx: aluCtr = 4'b0110;
        8'b1xxx0000: aluCtr = 4'b0010;
        8'b1xxx0010: aluCtr = 4'b0110;
        8'b1xxx0100: aluCtr = 4'b0000;
        8'b1xxx0101: aluCtr = 4'b0001;
        8'b1xxx1010: aluCtr = 4'b0111;
        default: aluCtr = 4'b0000;
    endcase
endmodule
```

附录3

算术运算单元模块的Verilog代码实现：

```
module Alu(
    input1,
    input2,
    aluCtr,
    zero,
    aluRes
    );

    input [31:0] input1;
    input [31:0] input2;
    input [3:0] aluCtr;
    output zero;
    output [31:0] aluRes;

    reg zero;
     reg [31:0] aluRes;

     always @ (input1 or input2 or aluCtr)
     begin
     case (aluCtr)
     4'b0000:  // and
         aluRes = input1 & input2;
     4'b0001:  // or
         aluRes = input1 | input2;
     4'b0010:  // add
         aluRes = input1 + input2;
     4'b0110:  // sub
         aluRes = input1 - input2;
```

```
29      4'b0111:  // set on less than
30           begin
31           aluRes = input1 - input2;
32          if (aluRes[31] == 0) aluRes = 0;
33           else aluRes = 1;
34          end
35      4'b1100:  // NOR
36          aluRes = ~(input1 | input2);
37      default:
38          aluRes = 0;
39      endcase
40      if (aluRes == 0) zero = 1;
41      else zero = 0;
42      end
43  endmodule
```