CSE-COA-LAB-005

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-整体调试

1.2 实验目的

完成单周期的MIPS处理器

1.3 实验范围

本次试验将覆盖以下范围:

1. ISE的使用
2. Xilinx Spartan 3E实验板的使用
3. 使用VerilogHDL进行逻辑设计
4. 仿真测试、下载验证

2 实验内容

2.1 MIPS处理器整体实现概述

在单周期MIPS处理器的整体实现中，我们需要建立一个top模块，将之前实现的主控制模块、算术运算控制模块、算术运算模块、寄存器模块、内存模块和符号扩展模块实例化，并在top模块中建立指令内存模块，用连线将这些模块连接起来，并上板验证.

2.2 单周期MIPS处理器的端口

单周期MIPS处理器的输入端口是: CLK, RESET. 此外，处理器还要从文件中读取相应的指令，这可以通过Verilog内置的语句来完成.

2.3 单周期MIPS处理器的代码实现

与前几个lab相比，实验5的工作就要稍微复杂一些了，原因是在top模块中出现了许多的线网类型的变量. 这些变量将之前我们已经完成的各个模块连接到一起，这就需要我们比较仔细地完成这项工作，才能最终得到正确的结果.

2.3.1 引入之前完成的模块

我们将之前在lab3和lab4中完成的各个模块的Verilog文件复制到当前工程的文件夹内，并把它们添加到工程当中. 复制这些文件的时候务必要小心，如果没有把这些文件复制到新的文件目录下的话，会导致原来项目的文件也被改变了，这将对之前完成的project造成污染. 需要注意的是，在把这些文件添加到工程当中时需要作出适当的删改，比如说为寄存器模块和内存模块添加reset信号等. 当我们完成这些之后就可以正式开始top模块的编程了.

2.3.2 添加指令内存模块

与内存模块相似，我们可以完成指令内存模块instMem. 注意到这里可以使用一个$readmemb的指令来从文件中读取指令的二进制形式. 此外，在标准的MIPS实现中，PC的值代表的是byte address，因此在实现指令内存模块时应当以byte作为单元.

2.3.3 添加RESET信号

在计算机CPU的设计中，RESET信号是十分重要的. 一般当CPU出现难以预料的错误的时候就需要用RESET信号来复位，就像电脑需要重新启动一样. 比较有名的芯片Intel 8086芯片就有reset管脚. 我实现的reset信号是根据电平，在时钟的上升沿触发的. 当reset信号使能的时候，指令内存里的内容会被重新加载，寄存器模块里的各个寄存器会被置为0.

2.3.4 top模块的连线

在连线时，采用较为一致的线路命名方案对于理清思路是十分有帮助的. 我采取的命名方案是把所有的线路用大写字母和下划线连接起来，事实证明这种命名方案还是带来了一些便捷性. 之后我们在实例化的过程中将这些wire类型的变量与端口相连，就成功地把各个模块连接在了一起. 在这里，我们不应该忽视wire类型的数据宽度，而一开始我忽视了一些wire类型的宽度导致这些变量只有1位. 事实上我们应该严格保证连线与端口的位宽相同.

2.3.5 PC地址跳转的实现

PC地址的跳转的实现主要依赖于multiplex的实现，而multiplex的实现在`Verilog`中十分简单，通过一个三目运算符即可实现，如`assign c = predicate?a:b;`. 在实现PC的跳转时，第一步在`beq`和`PC+4`中选取合适的值；第二步，在第一部得到的结果与`jump`地址中选取合适的值. 这样我们就得到了正确的跳转地址.

2.3.6 top模块中的时序逻辑

单周期CPU的时序逻辑并不复杂. 我实现的时序逻辑是在时钟上升沿时改变PC的值. 当PC改变时，重新加载下一条指令的值. 至此，单周期CPU的代码实现就全部完成了. 具体的top模块的代码见附录1.

2.4 单周期`MIPS`处理器的行为级仿真

在进行单周期`MIPS`处理器的行为级仿真的过程中，很重要的一点是要编写较为详尽的testbench，要涵盖寄存器读写、内存读写、算术运算、符号扩展以及指令跳转等情况. 我所编写的testbench如下所示:

```
10001100
00000001
00000000
00000001    //lw $1, 1($0)
10001100
00000010
00000000
00000010    //lw $2, 2($0)
10101100
00000010
00000000
00000000    //sw $2, 0($0)
00000000
00100010
00011000
00100000    //add $3, $1, $2
00000000
01100001
00100000
00100010    //sub $4, $3, $1
00000000
```

```
22      01000001
23      00011000
24      00100101    // or $3, $1, $2
25      00000000
26      01000001
27      00011000
28      00100100    // and $3, $1, $2
29      00001000
30      00000000
31      00000000
32      00000000    // j 0
```

通过ISim仿真得到的结果如下所示：

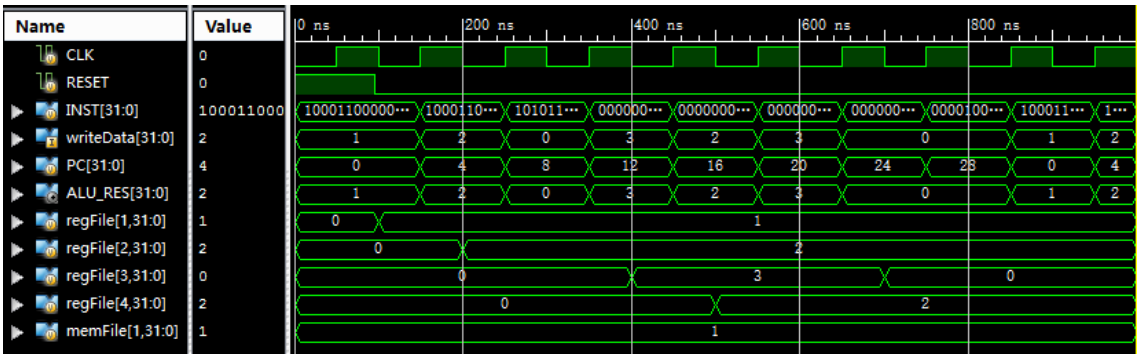

图 1: 单周期MIPS处理器的测试波形图

从仿真波形图上可以看出，单周期MIPS处理器的各个指令均运行正常，寄存器、内存等模块的读写也正常工作. 来自外部的reset信号也能起到置位的作用. 因此，仿真阶段的各项工作都已经顺利完成.

2.5 单周期MIPS处理器的上板验证

因为实验5的仿真部分耗费了我很多时间，因此我上板的时间就比较地紧张，于是我决定用数电实验的板子来完成实验5和实验6的上板. 幸运的是，数电的板子同样可以通过ISE开发套装来完成二进制文件的生成. 当然，在小板子上上班验证同样会带来一些问题，我之后也会详细说明的.

在上板的时候，首先需要完成的就是一个分频器的实现，因为数电的板子它的时钟频率也是100MHz的，因此我编写的分频器是26位的，这样大约1秒就能执行一个指令，是比较合适的. 在我设计的单周期MIPS处理器的上板验证中，我用两个led代表当前PC的值，然后分别两个led表示`regFile[1]`，用两个led表示`regFile[2]`，用两个led表示`regFile[3]`. 然后用一个拨动开关表示`switch`. 具体的管脚定义见附录2.

在上板之前，有些困难是之前所无法估计的. 因为我之前写Verilog代码的时候不可避免地依旧带着C语言的痕迹，因此当在ISE里面进行到implement design的环节中的时候，就会因为ISE无法实

现Verilog代码的要求而导致的一些bug. 这些错误有些是牵一发而全身的，可能某一个bug虽然显示错误在一个地方，但是实际上错误的根源在另一个地方，在调试这种错误的时候就需要我们格外的细致和耐心. 此外，由于我使用的是小板子，有时候会报出某些器件数量不足而无法成功生成二进制文件的错误，对于这种错误，一种可行的措施是减少内存单元的内存大小，可以有效地解决这种状况.

此外，还有一个困扰我的问题就是在上板的时候`initail`块是没有作用的，一开始我一直以为是我的程序出现了问题，但是在我反复调试后发现还是应该用过外部的reset信号来实现指令内存的加载和寄存器堆的清零.

为了方便上板时对各个模块参数的调试，我将指令内存模块与top模块分离开来了，修改后的top模块代码见附录3. 在下载二进制文件到板子上时，我没有采用ISE自带的iMpact工具，而是使用了Adept工具，相比较而言，Adept在使用上更加简便快捷.

3 心得体会

实验5是我第一次实现较为完整的CPU，很多困难也是之前完全没有预料到的. 总体而言，收获有以下几点：

1. 掌握了从底向上的工程构建方法，我们在之前的lab3和lab4中已经完成了要构建一个单周期CPU所必须的一些基本单元. 在lab5完成一个简单的单周期CPU的过程中中，我们从底层的算术运算、内存、寄存器等模块开始，通过top模块中的线网把这些现有的底层的模块组合在一起，形成了一个能够正常工作的CPU.

2. 体会到了在计算机硬件设计中同步性的重要性，许多由于读写不同步而带来的问题都需要我们仔细思考来解决. 在比较复杂的时序逻辑当中，采用时钟上升沿和下降沿来同步往往是比较简洁和方便的.

3. 学习了硬件调试的基本方法，通过把led的端口连接到我们需要调试的变量上面，就能得到有关的信息. 在现实中，硬件往往是复杂的，我们从led上得到的信息比我们之前通过ISim仿真得到结果更加贴合实际，也更需要我们认真的调试.

4. 体会到了实践的重要意义: 虽然ISim的仿真功能十分强大，但是如果要验证我们的结果在实际硬件上的运行情况的话，还是必须要下载到FPGA开发板上去验证，仿真的结果忽略了一些硬件上的限制，有时候太过于理想，反而可能让我们过于乐观地估计结果.

5. 要善于使用一些辅助的工具来提高工作的效率. 上个学期的计算机组成的project1中，我们学习使用了SPIM汇编器来编写MIPS汇编程序，完成了Fibonacci数列的计算. 在这里，我们也可以使用SPIM来将MIPS指令转换成二进制的机器码. 更重要的是，我们可以利用SPIM的单步调试功能来查看每个时间段各个寄存器里的数据，来与我们用ISim仿真出来的结果比对. 事实证明我的结果运行正确.

6. 要善于利用软件自身的documentation和Xilinx forum上的信息. 大多数情况下，我们碰到的问题之前都有人遇到过了，我们在Xilinx论坛上可以很轻松地查询到以前的记录. 虽然这些问题都是用英文叙述的，但是都叙述地比较清楚，还是和容易看懂的.

7. 在编写Verilog代码的时候要注意一些细节问题. 比如说在写testbench指令的时候，我想当然地认为可以在指令的后面添加像C语言的注释，但是事实证明这样没法通过`readmemb`指令来读取文件中的内容. 经过长时间的排查之后才发现不能在指令内存文件中添加注释，这个教训也告诉我在很多情况下要按照规则进行编程，而不能想当然地做出决定.

8. 对于指令中nop的设计，不能简单地把他当做全是0的指令，因为在我们设计的MIPS指令集中，当opCode域为000000时，指令将会被译码为一个R-type指令，可能会导致意想不到的后果. 因此，我们可以把指令设计为例如`add $1, $1, $0`的形式，使得我们的nop指令运行正常.

总而言之，lab5进一步加深了我单周期MIPS处理器的理解，对于计算机的冯诺依曼体系结构有更加深刻的理解. 对于lab5程序的调试，也大大提高了我对Verilog语言的理解，让我在调试程序的时候更加耐心细致. 在上板时碰到的种种问题也为我之后在硬件上的调试工作提供了宝贵的经验.

4 致谢

感谢老师在实验上的仔细指导. 也感谢我的同学在我们实验陷入停滞的时候的互相鼓励.

附录1

top模块的Verilog代码如下所示:

```
module Top(
    CLK,
    RESET
    );
    input CLK;
    input RESET;

    reg [31:0] INST;
    reg [7:0] instMem[0:127];
    reg [31:0] PC;

    initial
    begin
        $readmemb("./src/my_inst_mem.txt", instMem);
    end

    // Ctr
    wire REG_DST;
    wire JUMP;
    wire BRANCH;
    wire MEM_READ;
    wire MEM_TO_REG;
    wire [1:0] ALU_OP;
    wire MEM_WRITE;
    wire ALU_SRC;
    wire REG_WRITE;
    Ctr mainCtr(
        .opCode(INST[31:26]),
        .regDst(REG_DST),
        .jump(JUMP),
        .branch(BRANCH),
        .memRead(MEM_READ),
        .memToReg(MEM_TO_REG),
        .aluOp(ALU_OP),
        .memWrite(MEM_WRITE),
        .aluSrc(ALU_SRC),
        .regWrite(REG_WRITE)
    );

    // register
    wire [4:0] WRITE_REG;
    wire [31:0] WRITE_REG_DATA;
    wire [31:0] READ_DATA_1;
    wire [31:0] READ_DATA_2;
    assign WRITE_REG = REG_DST ? INST[15:11] : INST[20:16];
    register mainRegister(
        .regWrite(REG_WRITE),
        .readReg1(INST[25:21]),
        .readReg2(INST[20:16]),
        .writeReg(WRITE_REG),
        .writeData(WRITE_REG_DATA),
        .readData1(READ_DATA_1),
        .readData2(READ_DATA_2),
        .clock_in(CLK),
        .reset(RESET)
    );
```

```
57
58      // sign extension
59      wire [31:0] SIGNEXT_OUT;
60      signext mainSignext(
61          .inst(INST[15:0]),
62          .signextOut(SIGNEXT_OUT)
63      );
64
65      // aluCtr
66      wire [3:0] ALU_CTR;
67      aluCtr mainAluCtr(
68          .funct(INST[5:0]),
69          .aluOp(ALU_OP),
70          .aluCtr(ALU_CTR)
71      );
72
73      // Alu
74      wire [31:0] ALU_INPUT_2;
75      wire ZERO;
76      wire [31:0] ALU_RES;
77      assign ALU_INPUT_2 = ALU_SRC ? SIGNEXT_OUT : READ_DATA_2;
78      Alu mainAlu(
79          .input1(READ_DATA_1),
80          .input2(ALU_INPUT_2),
81          .aluCtr(ALU_CTR),
82          .zero(ZERO),
83          .aluRes(ALU_RES)
84      );
85
86      // memory
87      wire [31:0] READ_MEM_DATA;
88      data_memory mainDataMem(
89          .clock_in(CLK),
90          .address(ALU_RES),
91          .readData(READ_MEM_DATA),
92          .writeData(READ_DATA_2),
93          .memRead(MEM_READ),
94          .memWrite(MEM_WRITE)
95      );
96
97      // register data_in
98      assign WRITE_REG_DATA = MEM_TO_REG ? READ_MEM_DATA : ALU_RES;
99
100     initial begin
101         PC = 0;
102     end
103
104     wire [31:0] PC_PLUS_FOUR;
105     assign PC_PLUS_FOUR = PC + 4;
106     wire [31:0] JUMP_ADDR;
107     assign JUMP_ADDR = {PC_PLUS_FOUR[31:28], INST[25:0]<<2};
108     wire [31:0] BEQ_ADDR;
109     assign BEQ_ADDR = PC_PLUS_FOUR + SIGNEXT_OUT<<2;
110     wire [31:0] ACTUAL_BEQ_ADDR;
111     assign ACTUAL_BEQ_ADDR = (BRANCH&&ZERO) ? BEQ_ADDR : PC_PLUS_FOUR;
112     wire [31:0] NEXT_PC;
113     assign NEXT_PC = (JUMP) ? JUMP_ADDR : ACTUAL_BEQ_ADDR;
114
115     always @ (posedge CLK) begin
116     if (RESET)
```

```
117         PC = 0;
118     else
119         PC = NEXT_PC;
120     end
121
122     always @ (PC) begin
123     INST = {instMem[PC],instMem[PC+1],instMem[PC+2],instMem[PC+3]};
124     end
125 endmodule
```

附录2

仿真的管脚定义ucf代码如下所示:

```
1  NET "CLK" LOC = B8;
2  NET "RESET" LOC = P11;
3  NET "led[0]" LOC = M5;
4  NET "led[1]" LOC = M11;
5  NET "led[2]" LOC = P7;
6  NET "led[3]" LOC = P6;
7  NET "led[4]" LOC = N5;
8  NET "led[5]" LOC = N4;
9  NET "led[6]" LOC = P4;
10 NET "led[7]" LOC = G1;
```

附录3

修改后的top模块的Verilog代码如下所示(仅包含了关键代码):

```
1  module timeDiv(
2      clk_in,
3      clk_out
4      );
5
6      input clk_in;
7      output reg clk_out;
8      reg [25:0] buffer;
9
10     always @ (posedge clk_in) begin
11         buffer <= buffer + 1;
12         clk_out <= buffer[25];
13     end
14 endmodule
15
16 module Top(
17     CLK,
18     RESET,
19      led
20     );
21     input CLK;
22     input RESET;
23     output wire [7:0] led;
24     wire topClk;
25
26     timeDiv mainTimeDiv(
```

```
27          .clk_in(CLK),
28          .clk_out(topClk)
29      );
30
31      reg [31:0] INST;
32      reg [7:0] instMem[0:40];
33      reg [31:0] PC;
34
35      assign led[6] = PC[3];
36      assign led[7] = PC[4];
37
38      // register
39      register mainRegister(
40          .regWrite(REG_WRITE),
41          .readReg1(INST[25:21]),
42          .readReg2(INST[20:16]),
43          .writeReg(WRITE_REG),
44          .writeData(WRITE_REG_DATA),
45          .readData1(READ_DATA_1),
46          .readData2(READ_DATA_2),
47          .clock_in(topClk),
48          .led(led[5:0]),
49          .reset(RESET)
50      );
51
52      always @ (posedge topClk) begin
53      if (RESET) begin
54      PC = 0;
55      instMem[0] = 8'b00001000;
56      instMem[1] = 8'b00000000;
57      instMem[2] = 8'b00000000;
58      instMem[3] = 8'b00000001; // if (regFile[3]==regFile[0]) goto PC=0
59      instMem[4] = 8'b10001100;
60      instMem[5] = 8'b00000001;
61      instMem[6] = 8'b00000000;
62      instMem[7] = 8'b00000001; // regFile[1]=memFile[1]=1;
63      instMem[8] = 8'b10001100;
64      instMem[9] = 8'b00000010;
65      instMem[10] = 8'b00000000;
66      instMem[11] = 8'b00000010; // regFile[2]=memFile[2]=2;
67      instMem[12] = 8'b00000000;
68      instMem[13] = 8'b00100010;
69      instMem[14] = 8'b00011000;
70      instMem[15] = 8'b00100000; // regFile[3]=regFile[1]+regFile[2]
71      instMem[16] = 8'b00000000;
72      instMem[17] = 8'b01000001;
73      instMem[18] = 8'b00011000;
74      instMem[19] = 8'b00100010; // regFile[3]=regFile[1]-regFile[2]
75      instMem[20] = 8'b00000000;
76      instMem[21] = 8'b01000001;
77      instMem[22] = 8'b00011000;
78      instMem[23] = 8'b00100101; // regFile[3]=regFile[1]|regFile[2]
79      instMem[24] = 8'b00000000;
80      instMem[25] = 8'b01000001;
81      instMem[26] = 8'b00011000;
82      instMem[27] = 8'b00100100; // regFile[3]=regFile[1]&regFile[2]
83      instMem[28] = 8'b10101100;
84      instMem[29] = 8'b00000010;
85      instMem[30] = 8'b00000000;
86      instMem[31] = 8'b00000000; // regFile[3]=memFile[0]
```

```
87      end
88      else PC = NEXT_PC;
89      end
90
91      always @ (PC) begin
92      INST = {instMem[PC],instMem[PC+1],instMem[PC+2],instMem[PC+3]};
93      end
94  endmodule
```