Lab05　简单的类 MIPS 单周期处理器的实现 - 整体调试

w1049

目录

1 实验目的

1. 理解简单的类 MIPS 单周期处理器的工作原理（即几类基本指令执行时所需的数据通路和与之对应的控制线路及其各功能部件间的互联定义、逻辑选择关系）
2. 完成简单的类 MIPS 单周期处理器
   (a) 9 条 MIPS 指令（`lw`, `sw`, `beq`, `add`, `sub`, `and`, `or`, `slt`, `j`）的实现与调试
   (b) 拓展至 16 条指令（增加 `addi`, `andi`, `ori`, `sll`, `srl`, `jal`, `jr`）的设计与实现
3. 仿真测试
4. 上板验证

2 原理分析

2.1 总体框架

总体来看，我先按照图 1编写了一个简单的单周期处理器，然后再在此基础上增加其他指令的支持。报告的剩余部分也是从简单到复杂，逐渐扩展的。处理器有两个主要部分，一是数据通路，二是控制信号。完善了数据通路各部件后，在顶层模块中实例化其他模块，再加入控制信号，就可以完成整个处理器的设计。

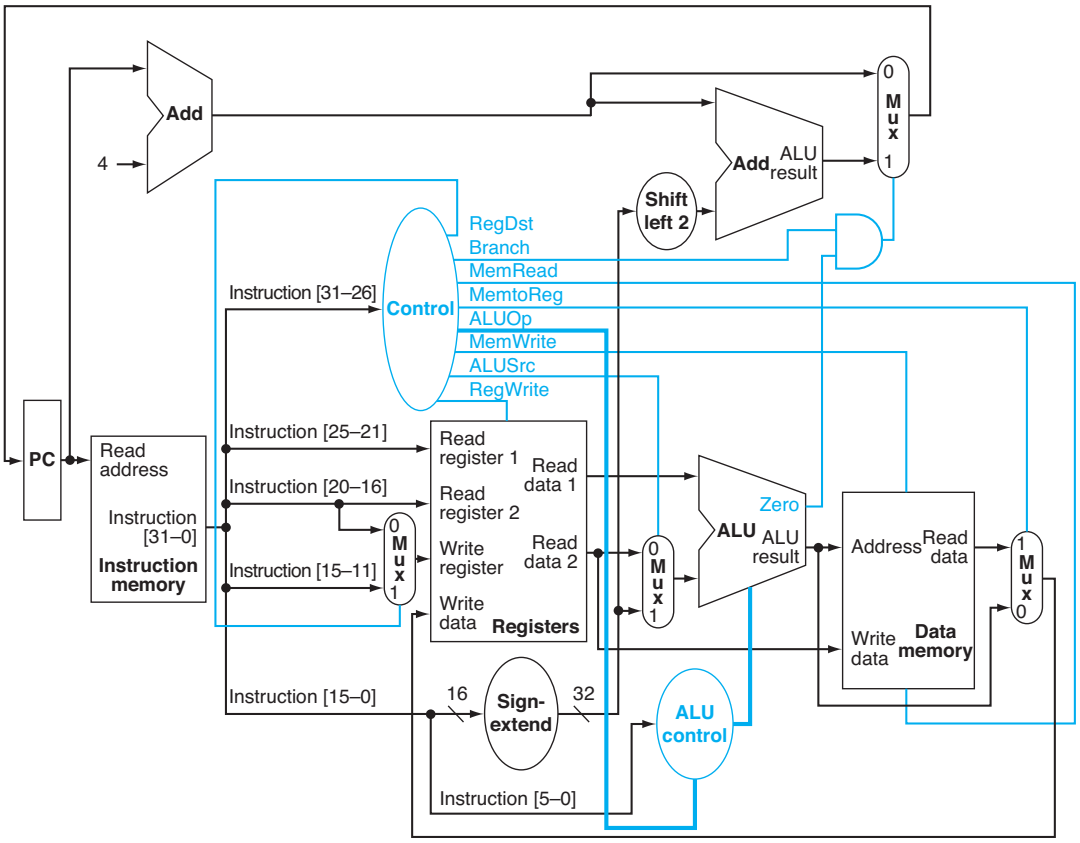

图 1: 能处理跳转等几类 MIPS 指令的简单的单周期处理器电路设计图

2.2 数据通路

数据通路的主要部件已经在前几次实验中完成，还有两个没有实现：

- 指令内存：指令内存与数据内存的功能一致，但是本次设计的处理器不支持更改指令内容，指令内存是只读的；
- 程序计数器（PC）：程序计数器实际上是个寄存器，每到时钟上升沿就会读取输入，改变输出。

为了将各部分连接起来，还需要做很多修改和调整，我会在功能实现中详细阐述。

3 功能实现

3.1 数据通路与控制

首先打通数据通路。把数据内存的写入功能删除就得到了指令内存；对于 PC，我使用了一个新模块，如下：

```
module PC(
    input Clk,
    input reset,
    input [31:0] in,
    output reg [31:0] out
    );

    always @(posedge Clk) begin
        if (!reset) out <= in;
        else out <= 0;
    end

endmodule
```

这个模块就是简单的触发器，在时钟上升沿将输入搬到输出。

如此，数据通路的部件已经齐全，先把它们连接起来，也就是在顶层模块中实例化各模块。我简单地把每个模块的定义（包括 input、output 的一串参数）复制到顶层模块中，然后使用同名变量作为输入输出。比如，ALU 的定义是：

```
module ALU(
    input [31:0] input1,
    input [31:0] input2,
    input [3:0] aluCtr,
    output reg zero,
    output reg [31:0] aluRes
    );
```

在 Top.v 中，我首先复制粘贴，并利用正则表达式修改成：

```
    ALU alu (
        .input1(input1),
        .input2(input2),
        .aluCtr(aluCtr),
        .zero(zero),
        .aluRes(aluRes)
        );
```

完成实例化后，这么多的“未知变量”如何连接呢？我的方案是首先把所有变量都定义出来，然后修改输入变量。也即，我需要做的是，找出每个模块的输入应该是什么。可能有两种修改方式，一种是使用 assign 为输入变量赋值；一种是直接把输入改成对应的变量。为了减少变量数目，也是为了方便，我对直接连接的线采用改变实参的方式，对于需要用 Mux 处理的线使用 assign。

仍然拿 ALU 作例子，从图 1 中我们知道，ALU 有两个数据输入，一个直接来自于寄存器组，一个来自于寄存器组和符号扩展二选一。因此，ALU 的代码修改为：

```
ALU alu (
  .input1(readData1), // 第一个输入肯定是寄存器组的输出
  .input2(input2),    // 第二个输入需要选择
  .aluCtr(aluCtr),    // 输出，不管
  .zero(zero),        // 输出，不管
  .aluRes(aluRes)     // 输出，不管
  );
// 第二个输入的选择，三元表达式即可，不用显式写 Mux
assign input2 = aluSrc ? imm : readData2;
```

除了 ALU，其他模块的修改也是类似的，这里不再赘述。唯一值得注意的是 PC，它的输入比较复杂：

```
PC u0 (
    .Clk(Clk),
    .reset(reset),
    .in(newpc),
    .out(pc)
);
assign pcplus4 = pc + 4;
assign newpc = jump ? {pcplus4[31:28], inst[25:0], 2'b0}
                    : (branch && zero ? (imm << 2) + pcplus4 : pcplus4);
```

控制模块也是模块，按照上文做法写好输入即可。

3.2 9 条指令

连接完成后，CPU 还不能直接处理 9 条指令，需要对各部件做出一定修改。

3.2.1 复位

为了支持 `reset`，要给寄存器组加上 `reset` 输入。我采用了同步复位，即只有在时钟边沿才会检测是否复位。这种设计有些时候会出现问题，因为有的部件在上升沿复位，有的在下降沿复位，而且无法预判实际的复位信号会跨几个周期，在哪里停下。比如，第一条指令是修改寄存器内容，而复位信号在后半周期结束，那么寄存器不会被修改（下降沿时还是复位状态），但 PC 会向后走一步（上升沿时复位早已结束），导致第一条指令被跳过。简单的规定第一条指令是 `nop` 可以规避这个问题。不过当第一条指令不是 `nop` 时，我也很少遇到错误。

在实现时，`if reset` 的判断要写在原本的 `always` 块中，否则会引发错误（Vivado 可能报告为警告，不影响仿真，但会影响上板）。代码如下：

```
always @(negedge Clk) begin
    if (reset) for (i = 0; i < 32; i = i + 1)
        regFile[i] <= 0;
    else if (regWrite) regFile[writeReg] <= writeData;
end
```

3.2.2 寻址问题

在 MIPS 中，指令按字节寻址；但在我们的实现中，指令内存、数据内存是按字（4 个字节，32 位）寻址的。为了规范，我们应该修改寻址方式。好消息是，如果只考虑 4 字节数据，并且所有地址都是对齐到 4 的倍数的，那么只需要稍作修改，数组下标除以 4 即可：

```
    assign readData = memRead && !memWrite ? memFile[address >> 2] : 0;
    always @(negedge Clk) begin
        if (memWrite)
            memFile[address >> 2] <= writeData;
```

当然，也可以不修改存储器，而是重新解释指令中的地址。我不推荐这样与标准背道而驰的行为，如果这样做了，不能使用汇编器来帮助调试，而需要自己编写指令——因为主流的汇编器是按字节寻址的。

3.2.3 数据初始化

在仿真测试时，数据可以在 `initial` 块中初始化，这样会让 `reset` 成为摆设，但是方便调试。使用以下代码即可把 `lab05.sim/sim_1/behav/xsim/Data` 文件中的数据按十六进制（h）读入到内存中：

```
    initial begin
        $readmemh("Data", memFile);
    end
```

3.2.4 测试

以上步骤完成后，CPU 应当已经可以支持 9 条指令了。可以用以下简单的代码来测试：

```
    j a
    nop
    nop
    nop
a:
    lw $1, 4($0)
    lw $2, 8($0)
    add $3, $1, $2
    sub $4, $1, $2
```

```
00001000000000000000000000000100
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
10001100000000010000000000000100
10001100000000010000000000001000
00000000001000100001100000100000
00000000001000100010000000100010
```

3.3 16 条指令

接下来，将指令数目拓展到 16 条。一共有 7 条待添加的指令，都需要不同程度地修改控制信号，甚至修改部件功能。

3.3.1 addi, andi, ori

符号扩展 这三条都是 I 型指令，但 `addi` 的立即数是有符号扩展，而 `andi`、`ori` 的立即数是零扩展。因此，需要修改符号扩展模块，增加一个控制信号 `extOp`，使其可以根据指令类型选择扩展方式。修改后的代码如下：

```
module Signext(
    input extOp,
    input [15:0] inst,
    output [31:0] data
    );
    assign data = extOp ? {{16{inst[15]}}, inst} : {{16{1'b0}}, inst};
endmodule
```

extOp 由 opCode 决定，故加入到 Ctr 模块中。

ALU 控制 I 型指令并不由 funct 区分，而是直接由 opCode 决定，所以要在 Ctr 模块中判断；但是我们希望 ALU 只由 ALUCtr 控制，所以为 aluOp 增加一位，变为 3 位控制信号。Ctr 模块根据 opCode 决定 aluOp，再让 ALUCtr 模块根据 aluOp（和 funct）决定 ALU 的操作。

3.3.2 sll, srl

ALU 为 ALU 添加移位功能，注意逻辑右移在 Verilog 中是用 >>> 实现的，还要注意两个操作数的顺序：

```
    // 在 ALU 的 case 中
    4'b0011: aluRes = input2 << input1;  // sll
    4'b0100: aluRes = input2 >>> input1; // srl
```

ALU 控制 ALU 的第一个输入也需要选择了，增加一个控制信号 ALUSrc1 选择 input1，为 1 是指令中的 shamt，为 0 是寄存器中的 readData1。

3.3.3 jal

将 regDst 改为两位，若 regDst[1] == 1，则把 PC+4 写入 31 号寄存器。这需要修改 Ctr 模块，并修改数据通路，以选择 writeReg 和 writeData。

3.3.4 jr

将 jump 改为两位，若 jump[1] == 1，则把 readData1 写入 PC。jr 是特殊的 R 型指令，需要修改 Ctr 模块，传入 funct；还需要修改顶层模块中关于 newpc 的选择。

4 功能仿真

使用助教提供的乘法进行测试，仿真波形图见图 2。该程序用于计算乘法，当内存数据是 9,17 时，波形图中可以看到 117，即 9*17 的结果。

```
    lw $1, 0($3)
    lw $2, 4($3)
label3:
    srl $4, $2, 1
    sll $5, $4, 1
    beq $5, $2, label1
    add $8, $8, $1
label1:
    srl $2, $2, 1
    sll $1, $1,1
    beq $2, $3, label2
    j label3
label2:
    sw $8, 8($3)
```

```
10001100011000010000000000000000
10001100011000010000000000000100
00000000000000010001000000100010
00000000000000010000101000001000000
00010000101000010000000000000001
00000001000000001010000000100000
00000000000000010000100000100010
00000000000000010000100001000000
00010000010000110000000000000001
00001000000000000000000000000010
10101100011010000000000000001000
```

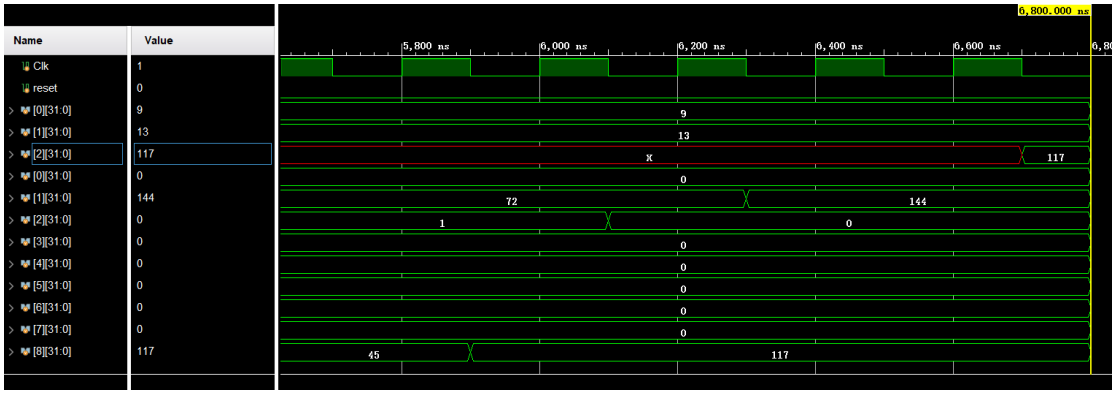

图 2: 仿真波形

5 上板验证

5.1 修改代码

为了能够上板验证，需要像前两次实验一样增加一部分代码。以 Lab02 的数码管、开关控制为基础，配置好约束文件，再从 Lab01 找到 reset 键的约束。将 display 的 IP 核加入项目中，在顶层模块中配置显示模块、时钟等。

板上 reset 键是低电平有效，时钟也和仿真不同，可以直接使用 `assign` 来定义：

```
    wire [3:0] aa;
    wire [3:0] bb;
    wire [7:0] sum;
    display DISPLAY (
        .clk(Clk_25M),
        .rst(1'b0),
        .en(8'b01010011),
        .data({4'b0, aa, 4'b0, bb, 8'b0, sum}),
        .dot(8'b0),
        .led(~{aa, bb, sum}),
        .led_clk(led_clk),
        .led_en(led_en),
        .led_do(led_do),
        .seg_clk(seg_clk),
        .seg_en(seg_en),
        .seg_do(seg_do)
    );

    wire Clk;
    wire reset;
    assign Clk = Clk_25M;   // 分频后的时钟
    assign reset = !_reset; // _reset 低电平有效
```

这里，aa,bb,sum 是直接连接在数据内存上的，可以通过数码管观察。

除此之外，上板不能使用 initial 块，所以一切初始化都要写在 always 块的 if reset 中。我将指令内存预先写好，数据内存则从开关中读取。所以还要给一些模块添加 reset 输入。

5.2 为测试程序加上无限循环

为了防止 PC 无限制增加导致问题，可以在汇编代码的最后加上一条无限循环的指令：

```
end:
    j end
```

这样可以让程序停在最后一行，不会继续执行。

5.3 验证结果

经过综合、实现后把程序下载到板子上，使用 8 个开关确定两个乘数，再按下 reset 键将数据写入内存、从头开始运行。

由于未知的原因，我从内存中获取的数字一直不稳定，有限的时间内我也没有得出解决方法。于是我直接从寄存器中读取所需数字并显示，达到了相同的效果，见图 3。

图 3: 上板验证，9H * dH = 75H

6 总结与反思

本次实验我由简单到复杂，一步一步实现了支持 16 条指令的单周期处理器。期间为了扩展功能，对各个部件和控制信号进行了修改，这要求我不能对代码一知半解，而是要对每一条语句的功能有清晰的认识。

上板验证时，我最后也没能发现为什么内存中的数据不稳定，只好采取读取寄存器的方法。如果时间充裕，或许可以考虑将更多内存数据显示在数码管上，观察其变化；或者使用专门的调试工具。从仿真来看，`sw` 指令的实现应该没有问题，我猜问题可能出在对内存的操作上。

从后面的实验来看，这里对控制信号的有些修改是不合理的，比如将一位的信号改成两位，其实不如使用两个一位信号。这些问题在后面的实验中得到了优化。