实验三 流水线 MIPS CPU

张海斌*

2019 年 5 月

目 录

1 概要

流水线处理器与单周期处理器在结构上比较类似，但是流水线处理器通过时间上并行的方式提高 CPU 的吞吐率。最简单的流水线处理器的做法是将单周处理器分成取指令（Fetch）、译码（Decode）、执行（Execute）、存储器（Memory）、写回（Writeback）五个阶段，并在每两个阶段之间添加一个状态寄存器用来存储后续阶段运行所需的数据和控制信号。但是这样简单的流水线处理器在实际运行过程中会出现各种错误情况，比如写后读。所以为了让流水线处理器能正确运行，我们还需要增加冒险、预测分支等部份。

2 功能展示

LED[15] 显示 CPU 时钟信号状态。SW[0] 控制状态的清零，SW[1] 控制 CPU 的运行和暂停，SW[3:2] 控制七段数码管数据显示。当 SW[3] 为 1 时，八个七段数码管显示当前从指令内存（imem）中读取的 32 位指令。当 SW[3] 为 0 时，从左向右前两个七段数码管显示当前 PC 值，接着的两个数码管显示写入 PC 使能寄存器的值（只有当有使能信号的时候才会写入 PC），最后四位七段数码管显示内存或寄存器中的值。此时，SW[9:4] 提供需要查看的

*学号 17307130118

寄存器或内存的地址：当 SW[2] 为 0 时，以 SW[8:4] 的内容为寄存器地址，显示相应寄存器中的数据在右侧四个七段数码管上；当 SW[2] 为 1 时，以 SW[9:4] 的内容为内存地址，在相同的七段数码管上显示内存的数据。

3 代码结构

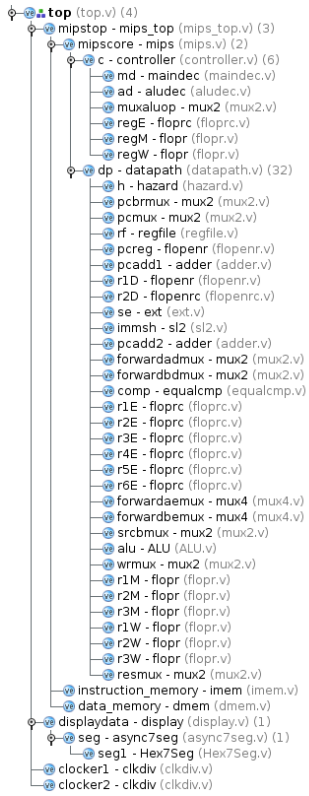

Figure 1: Verilog 模块层次结构图

Figure 1 展现了 Verilog 中各个模块的层次结构。`top` 是上板子的顶层文件，下面有两大模块 `mips_top` `display` 及两个时钟分频器 `clkdiv`。`mips_top` 和 `display`，分别为多周期处理器的顶层模块和负责处理板子显示的模块。在显示模块中包括了处理八个七段数码管连续显示的 `async7seg` 模块，而它下含的 `Hex7Seg` 则是将十六进制数转化为七段数码管上数字或字母显示的信号。`mips_top` 模块下主要有三个模块：`mips` MIPS 处理器核心、`imem` 指令内存和 `dmem` 数据内存。在 MIPS 核心模块下分为两个模块：`controller` 控制器和 `datapath` 数据通路。

4 模块设计

由于 `imem` 与 `dmem` 模块与单周期处理器完全相同，在报告中就不再赘述了。

4.1 `display` 显示模块

在 `display` 模块中，使用 `if` 语句和 `case` 语句来根据控制信号控制七段数码管的显示数据（相应代码见 Listing 1）。七段数码管要显示的数据存储在 `Data[31:0]` 中，并被提供

```
    always @(*)
    begin
        if (Ctl[1] == 0)
        begin
            case(Ctl[0])
                1'b0:
                begin
                    DispReadReg <= Addr[4:0];
                    DispReadMem <= 6'b0;
                    Data[15:0] <= DispRegData;
                end
                1'b1:
                begin
                    DispReadReg <= 5'b0;
                    DispReadMem <= Addr;
                    Data[15:0] <= DispMemData[15:0];
                end
            endcase
            Data[31:16] <= {PC[7:0], PCNext[7:0]};
        end
        else
        begin
            Data <= Instr;
        end
    end
```

Listing 1: `display` 模块数据显示的选择

给 `async7seg` 模块用于数据显示。和之前的显示模块相同地，使用了两个时钟分频模块分别为 CPU 和七段数码管的显示提供时钟信号。

4.2 `controller` 控制器模块

`controller` 模块中，除了和单周期处理器相同的与控制信号生成相关的 `maindec` 模块、`aludec` 模块以及一个二选一多路选择器外，还添加了存储执行 Execute, Memory, Writeback 阶段所需控制信号的三个寄存器。这些寄存器中只有 Execute 阶段对应的寄存器带有 Clear 清除控制信号。通过清除控制信号，可以清除 Execute 阶段寄存器中的数据，从而使这条指令在流向后续阶段时不会改变内存或寄存器的值，也就等价于 `nop` 指令。这个清除控制信号在冲突模块中将会被使用到。另一方面，三个阶段控制信号寄存器中的数据也是像流水线般从前向后传递数据，而且由于每个阶段都使用了部份控制信号，后续阶段的寄存器中控制信号数目逐层递减。相关代码见 Listing 2 。

```
    floprc #(8) regE(CLK, Reset, FlushE,
                {memToRegD, memWriteD, aluSrcD, regDstD, regWriteD, aluCtlD},
                {MemToRegE, memWriteE, ALUSrcE, RegDstE, RegWriteE, ALUCtlE});
    flopr #(3) regM(CLK, Reset,
                {MemToRegE, memWriteE, RegWriteE},
                {MemToRegM, MemWriteM, RegWriteM});
    flopr #(2) regW(CLK, Reset,
                {MemToRegM, RegWriteM},
                {MemToRegW, RegWriteW});
```

Listing 2: `controller`模块阶段控制信号寄存器

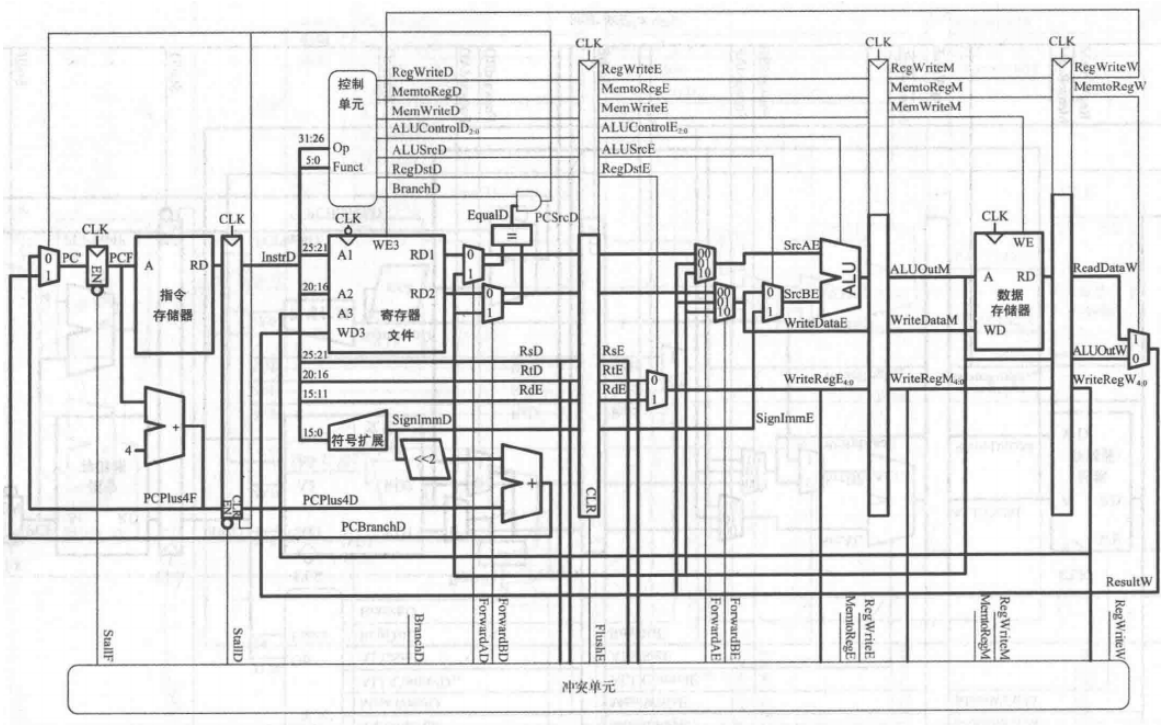

Figure 2: 流水线处理器结构图

4.3 datapath 数据通路模块

在 Figure 2 中完整画出了数据通路模块。相比于单周期处理器，数据通路中多了阶段数据寄存器、hazard 模块以及很多数据转发的线路。详细的冲突解决方法在 4.4 hazard 冲突模块中进行解释。

我所实现的实际电路中，在 PC' 的位置还有一个二选一多路选择器用来选择下一个 PC 是否为 Jump 指令对应的跳转地址。对于 Branch 指令，跳转判断发生在 Decode 阶段。默认预测为跳转不成功，即继续执行后续指令。如果 Branch 指令在 Decode 阶段跳转成功或者是 Jump 指令在 Decode 阶段，此时 PC 地址指向跳转指令的后一条指令处。在下一个时钟周期到来时，如果该 PC 对应的的指令写入 Decode 阶段数据寄存器，则它会被流水线错误地执行下去。所以我们要阻止其写入，即通过 Decode 阶段寄存器的 Clear 同步清除信号来控制错误指令不会写入其中。flushD 控制信号对应于 Clear 信号。

4.4 hazard 冲突模块

hazard 模块主要用来解决流水线执行过程中可能遇到的执行错误的问题。下面就对各种错误情况进行解决。

对于写后读的情况，可以采用数据转发的方式解决，这种方法的好处是可以保证流水线的执行速度。在流水线 CPU 数据通路中，由于 Decode 阶段执行 beq 或 bne 指令需要使用通用寄存器以及 Execute 阶段 ALU 计算也需要通用寄存器中数据，所以这两个阶段读取通用寄存器的位置都需要多路选择器对数据进行选择。相应选择信号控制见 Listing 3和 Listing 4 。

```
1     assign ForwardaD = (rsD != 5'b0 & rsD == WriteRegM & RegWriteM);
2     assign ForwardbD = (rtD != 5'b0 & rtD == WriteRegM & RegWriteM);
```

Listing 3: hazard 模块 Decode 阶段数据转发控制信号

首先我们讨论 Decode 阶段执行 Branch 指令时的转发问题。Writeback 阶段如果有需要写回到寄存器中的数据，那么它会在本周期中的时钟下降沿写入，之后如果寄存器文件恰好读这个寄存器，那就会读出一个正确的结果，这样就不需要将 Writeback 阶段数据转发到 Decode 阶段，但我们仍需处理 Execute 阶段和 Memory 阶段的寄存器写入数据。如果 Memory 阶段执行的指令要写入我们在 Decode 阶段需要读取的寄存器而且 Decode 阶段正在执行 Branch 指令，我们就通过数据转发传输正确的数据到 Decode 阶段。这对 Memory 阶段不需要读取内存时是正确的，但如果是从内存中获取数据写入寄存器中，就无法正常转发了，这就需要通过 stall 解决，即暂停 Fetch 和 Decode 阶段执行，让 Execute 及后续阶段继续执行，同时在下一周期将 Execute 阶段数据和控制寄存器清零。这样在下一时钟周期内，转发的数据会在时钟下降沿写入寄存器文件，从而在 Decode 阶段获得正确的数据。对于 Execute 阶段执行指令要写入 Decode 阶段 Branch 指令读取的寄存器的情况，我们也通过 stall 的方式解决。这样，在下一周期时，问题就变成了转发 Memory 阶段的寄存器数据。stall 控制信号代码见 Listing 5 。

接着我们来讨论转发到 Execute 阶段的问题。因为 Execute 阶段使用的数据是在上一个周期的寄存器数据，且 Writeback 阶段更新的寄存器数据无法改变 Execute 阶段数据寄存器中的数据，所以我们需要通过数据转发将 Writeback 阶段更新的寄存器数据传递到 Execute 阶段。同样，Memory 阶段对应执行的指令对寄存器的数据更新也应当转发到 Execute 阶段。

```
    always @(*)
    begin
        ForwardaE = 2'b00;
        ForwardbE = 2'b00;
        if (rsE != 0)
            if (rsE == WriteRegM & RegWriteM)
                ForwardaE = 2'b10;
            else
                if (rsE == WriteRegW & RegWriteW)
                    ForwardaE = 2'b01;
        if (rtE != 0)
            if (rtE == WriteRegM & RegWriteM)
                ForwardbE = 2'b10;
            else
                if (rtE == WriteRegW & RegWriteW)
                    ForwardbE = 2'b01;
    end
```

Listing 4: `hazard`模块 Execute 阶段数据转发控制信号

```
    assign lwstallD = MemToRegE & (rtE == rsD | rtE == rtD);
    assign branchstallD = BranchD &
            (RegWriteE & (WriteRegE == rsD | WriteRegE == rtD) |
            MemToRegM & (WriteRegM == rsD | WriteRegM == rtD));

    assign StallD = lwstallD | branchstallD;
    assign StallF = StallD;
    assign FlushE = StallD;
```

Listing 5: `hazard`模块 stall 信号

Memory 阶段指令有两种情况：一种是无需从数据内存中获得数据，此时直接转发即可；另一种是`lw`指令，需要从数据内存中获取数据然后写入寄存器，这种情况只能让 Execute 阶段暂停一个周期，等其读出内存数据再进行转发，这也就相当与从 Writeback 阶段转发。为了简化电路，我们对于这种情况，即`lw`指令写入寄存器后，后一条指令又读取同一个寄存器的情况，在后一条指令处于 Decode 阶段的时候让其暂停一个周期，这与让它在 Execute 阶段暂停一个周期的效果是等价的。判断这种情况的发生是 Listing 5 中`lwstallD`控制信号的作用。

5 仿真结果

```
#         Assembly              Description              Address    Machine
main:     addi $2, $0, 5        # initialize $2 = 5      0          20020005
          addi $3, $0, 12       # initialize $3 = 12     4          2003000c
          addi $7, $3, -9       # initialize $7 = 3      8          2067fff7
          or   $4, $7, $2       # $4 = (3 OR 5) = 7      c          00e22025
          and  $5, $3, $4       # $5 = (12 AND 7) = 4    10         00642824
          add  $5, $5, $4       # $5 = 4 + 7 = 11        14         00a42820
          beq  $5, $7, end      # shouldn't be taken     18         10a7000a
          slt  $4, $3, $4       # $4 = 12 < 7 = 0        1c         0064202a
          beq  $4, $0, around   # should be taken        20         10800001
          addi $5, $0, 0        # shouldn't happen       24         20050000
around:   slt  $4, $7, $2       # $4 = 3 < 5 = 1         28         00e2202a
          add  $7, $4, $5       # $7 = 1 + 11 = 12       2c         00853820
          sub  $7, $7, $2       # $7 = 12 - 5 = 7        30         00e23822
          sw   $7, 68($3)       # [80] = 7               34         ac670044
          lw   $2, 80($0)       # $2 = [80] = 7          38         8c020050
          j    end              # should be taken        3c         08000011
          addi $2, $0, 1        # shouldn't happen       40         20020001
end:      sw   $2, 84($0)       # write mem[84] = 7      44         ac020054
```

Figure 3: 测试代码

Figure 3是测试使用的代码。Figure 4和 Figure 5是流水线 CPU 仿真结果。值得注意的是`PC`只是代表当前周期内`PC`寄存器中的数据，也意味着当前周期内 Fetch 阶段从指令内存的`PC`位置读取指令，但在后续阶段中是否真正执行这条指令尚不确定，因为 Decode 阶段数据寄存器可能会受 Clear 信号控制从而被清除掉。

内存的写入与读出即`sw`与`lw`指令经过验证是正确的。

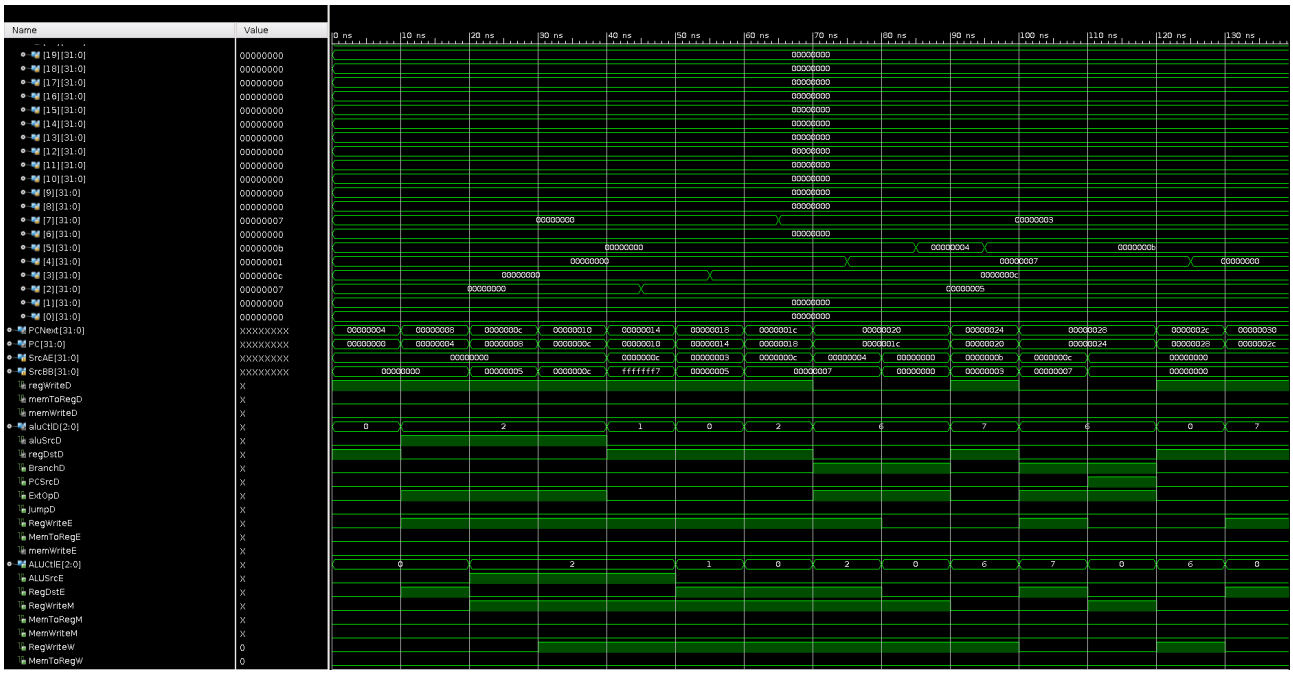

Figure 4: 仿真测试图 1

Figure 5: 仿真测试图 2