体系结构实验课程第_二_次实报告

实验名称	**静态 5 级流水线 CPU 实现**			班级	李雨森
学生姓名	李娅琦	学号	2213603	指导老师	董前琨
实验地点	A304		实验时间	星期一 12：00-13：30	

一. 实验目的

1. 在多周期 CPU 实验完成的提前下，深入理解 CPU 流水线的概念。
2. 熟悉并掌握流水线 CPU 的原理和设计。
3. 最终检验运用 verilog 语言进行电路设计的能力。
4. 通过亲自设计实现静态 5 级流水线 CPU,加深对计算机组成原理和体系结构理 论知识的理解。
5. 培养对 CPU 设计的兴趣，加深对 CPU 现有架构的理解和深思。

二. 实验要求

1. 做好预习：
1） 复习好上一次多周期 CPU 的实验，归纳常用的 MIPS 指令，确定自己准备实现的 MIPS 指令，对其进行分析，完成表 9.1 的填写；
2） 认真学习流水线的概念，明白流水线的意义和架构，理解数据相关、控制相关和结构相关，尤其需要注意分支跳转指令及其延迟槽指令的处理；
3） 依据自己设计中实现的指令，编写一段不少于 50 行的汇编程序，要求包含所有实现的指令，完成表 9.2 的填写。要求标注出指令间存在的相关，指出 CPU 可能存在的阻塞；
4） 在多周期 CPU 实验的设计框图的基础上，完善课程设计的设计框图，即补充完善图 9.2；
5） 如果对 FPGA 板了解的话，可确定设计中与 FPGA 板上交互的接口，画出包含外围模块的整体设计框图，即补充完善图 9.3。
2. 实验实施：
1） 确认流水 CPU 的设计框图的正确性；
2） 编写 verilog 代码，将表 9.2 中自己编写的汇编程序翻译为二进制，以 coe 文件的方式初始化到指令 ROM 中；
3） 对该模块进行仿真，得出正确的波形，截图作为实验报告结果一项的材料，在仿真时需要将生成指令 ROM 时产生的.mif 文件拷贝到工程目录下，才能仿真成功；
4） 完成调用流水 CPU 的外围模块的设计，并编写代码；
5） 对代码进行综合布局布线下载到实验箱里 FPGA 板上，进行上板验证。
3. 实验检查：
1） 完成上板验证后，让指导老师或助教进行检查，进行现场演示。先解读表 9.2 中自己编写的汇编程序，然后采用手动输入时钟，每个周期查看 CPU 状态，按照检查人员的要求进行演示，检查指令运行结果的正确性，可对演示结

果进行拍照作为实验报告结果一项的材料。

4. 实验报告的撰写：

1） 实验结束后，需按照规定的格式完成实验报告的撰写。

三．实验原理图

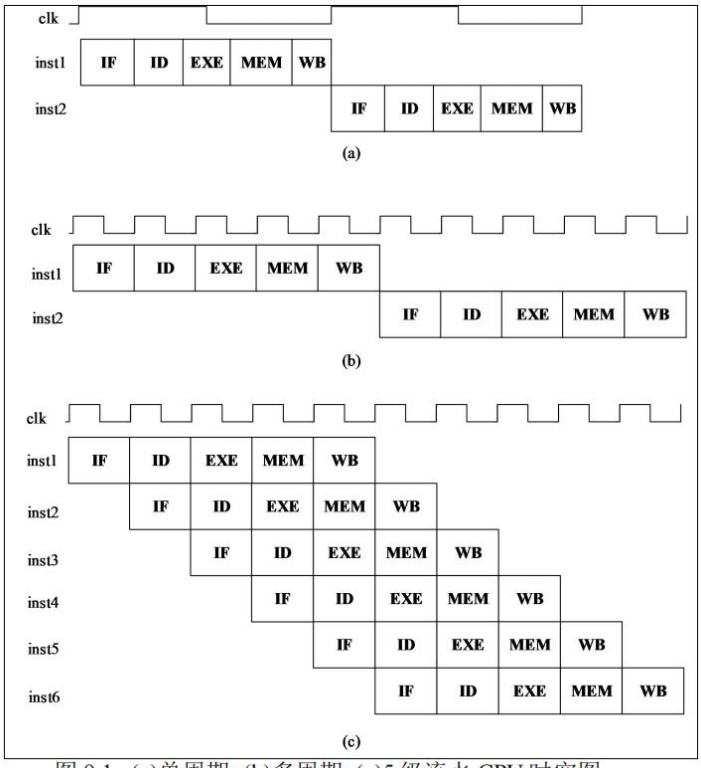


图 9.1 (a)单周期; (b)多周期; (c)5 级流水 CPU 时空图

四．代码修改

1.跳转指令。进行仿真实验时，发现跳转指令无法实现。同实验一将 IF 阶段再延迟一拍，确保跳转逻辑正确执行。

代码修改如下(fetch.v)：

```
`timescale 1ns / 1ps
//************************************************************************
*******
//    > 文件名: fetch.v
//    > 描述   :五级流水 CPU 的取指模块
//    > 作者   : LOONGSON
//    > 日期   : 2016-04-14
//************************************************************************
*******
`define STARTADDR 32'H00000034     // 程序起始地址为 34H
module fetch(                                  // 取指级
      input                clk,           // 时钟
      input                resetn,       // 复位信号，低电平有效
      input                IF_valid,    // 取指级有效信号
```

```
    input                   next_fetch,// 取下一条指令，用来锁存 PC 值
    input       [31:0] inst,          // inst_rom 取出的指令
    input       [32:0] jbr_bus,     // 跳转总线
    output      [31:0] inst_addr, // 发往 inst_rom 的取指地址
    output reg          IF_over,     // IF 模块执行完成
    output      [63:0] IF_ID_bus, // IF->ID 总线

    //5 级流水新增接口
    input       [32:0] exc_bus,     // Exception pc 总线

    //展示 PC 和取出的指令
    output      [31:0] IF_pc,
    output      [31:0] IF_inst
);
//-----{程序计数器 PC}begin
    wire [31:0] next_pc;
    wire [31:0] seq_pc;
    reg  [31:0] pc;
    //跳转 pc
    wire        jbr_taken;
    wire [31:0] jbr_target;
    assign {jbr_taken, jbr_target} = jbr_bus;   // 跳转总线传是否跳转和目标地
址
    //Exception PC
    wire        exc_valid;
    wire [31:0] exc_pc;
    assign {exc_valid,exc_pc} = exc_bus;
    //pc+4
    assign seq_pc[31:2]     = pc[31:2] + 1'b1;   // 下一指令地址：PC=PC+4
    assign seq_pc[1:0]      = pc[1:0];
    // 新指令：若有 Exception,则 PC 为 Exceptio 入口地址
    //         若指令跳转，则 PC 为跳转地址；否则为 pc+4
    assign next_pc = exc_valid ? exc_pc :
                     jbr_taken ? jbr_target : seq_pc;
    always @(posedge clk)       // PC 程序计数器
    begin
        if (!resetn)
        begin
            pc <= `STARTADDR; // 复位，取程序起始地址
        end
        else if (next_fetch)
        begin
            pc <= next_pc;      // 不复位，取新指令
        end
```

```
    end
//-----{程序计数器 PC}end
//-----{发往 inst_rom 的取指地址}begin
    assign inst_addr = pc;
//-----{发往 inst_rom 的取指地址}end
//-----{IF 执行完成}begin
    //由于指令 rom 为同步读写的,
    //取数据时，有一拍延时
    //即发地址的下一拍时钟才能得到对应的指令
    //故取指模块需要两拍时间
    //故每次 PC 刷新，IF_over 都要置 0
    //然后将 IF_valid 锁存一拍即是 IF_over 信号
    reg IF_over_d;
    always @(posedge clk)
    begin
        if (!resetn || next_fetch)
        begin
            IF_over <= 1'b0;
            IF_over_d <= 1'b0;
        end
        else
        begin
            IF_over_d <= IF_valid;
            IF_over <= IF_over_d;
        end
    end
//      always @(posedge clk)
//begin
//      if (!resetn || next_fetch)
//      begin
//          IF_over <= 1'b0;
//      end
//      else
//      begin
//          IF_over <= IF_valid;
//      end
//end
    //如果指令 rom 为异步读的，则 IF_valid 即是 IF_over 信号，
    //即取指一拍完成
//-----{IF 执行完成}end
//-----{IF->ID 总线}begin
    assign IF_ID_bus = {pc, inst};   // 取指级有效时，锁存 PC 和指令
//-----{IF->ID 总线}end
//-----{展示 IF 模块的 PC 值和指令}begin
```

```
    assign IF_pc      = pc;
    assign IF_inst = inst;
//-----{展示 IF 模块的 PC 值和指令}end
endmodule
```

2. **分析指令执行过程中的 RAW 和 WAR**。查看 ID 阶段代码可以发现：

```
    wire rs_wait;
    wire rt_wait;
    assign rs_wait = ~inst_no_rs & (rs!=5'd0)
                        & ( (rs==EXE_wdest) | (rs==MEM_wdest) |
(rs==WB_wdest) );
    assign rt_wait = ~inst_no_rt & (rt!=5'd0)
                        & ( (rt==EXE_wdest) | (rt==MEM_wdest) |
(rt==WB_wdest) );
    assign ID_over = ID_valid & ~rs_wait & ~rt_wait & (~inst_jbr | IF_over);
```

即通过 rs_wait 和 rt_wait 的计算逻辑确保了如果当前指令需要读取的寄存器（rs 或 rt）与后续阶段（执行、访存、写回）将要写入的目标寄存器相同，那么当前指令需要等待，ID_over 确保了 ID 阶段不会在需要的数据还未写入之前就完成，即正确解决了 RAW 问题。

对于 WAR 问题即在一个指令要读取寄存器值，另一个指令写入了同一个寄存器，通常是通过寄存器重命名或在指令调度时避免这样的依赖关系来解决的。

3. **优化。**在分析现有指令的执行过程中，发现现有 CPU 存在的不足，因此基于实验指导手册第十章中的优化部分提出以下想法：

①前递技术：由于 RAW 问题会造成堵塞和等待问题，因此可尝试实现前递技术直接从执行阶段将结果传递给后续需要的指令，避免等待写回阶段。目前感觉难度中等。

②分支预测：不使用延迟槽技术考虑实现转移预测技术，如静态分支预测或动态分支预测，以提升流水线效率。但是目前个人感觉实现会有些难度。

③预取机制：可以考虑对流水线设计方案作修改，使得取指级和访存级的 load 不需要多等一拍。可以引入数据和指令的预取机制，提前加载下一条指令或数据。但感觉这个更困难。

④cache 缓存：但是目前 MEM 阶段只用了一个周期，所以对关于增加 cache 缓存以提高效率存在疑问。

下面针对前递技术进行了初步修改：

主要分为以下几种类型（其中后两个可以合并）：

- EXE 级到 ID 级的前递
- MEM 级到 ID 级的前递
- WB 级到 ID 级的前递

基于此，个人认为可实现性较大，可能需要实现的步骤有：

a. 检测数据相关:在 ID 阶段检查是否存在后续指令需要前面的指令的结果

b. 选择前递路径:如果后续指令需要的操作数尚未写回，通过选择适当的前递路径将结果直接传递给需要它的指令，而不是等待写回

目前已经进行的修改有（decode.v）：

```
    //删除了 rs_wait 以及 rt_wait
    input         [153:0] EXE_MEM_bus_r,
    input         [117:0] MEM_WB_bus_r,
//-----{ID 执行完成}begin
    //由于是流水的，存在数据相关
      //判断前递类型
      wire rs_forward_EXE = (rs == EXE_wdest);
      wire rt_forward_EXE = (rt == EXE_wdest);
      wire rs_forward_MEM_WB = (rs == MEM_wdest||WB_wdest);
      wire rt_forward_MEM_WB = (rt == MEM_wdest||WB_wdest);
       wire [31:0] exe_result;
       assign {exe_result} = EXE_MEM_bus_r;
       wire [31:0] mem_result;
       assign {mem_result} = MEM_WB_bus_r;
       // rs 的前递选择逻辑
       assign rs_value = rs_forward_EXE ? exe_result :
                         rs_forward_MEM_WB ? mem_result :
                         rs_value; //  如果没有前递，则使用原本的寄存器值
       // rt 的前递选择逻辑
       assign rt_value =rt_forward_EXE ? exe_result :
                         rt_forward_MEM_WB ? mem_result :
                         rt_value; //  如果没有前递，则使用原本的寄存器值
```

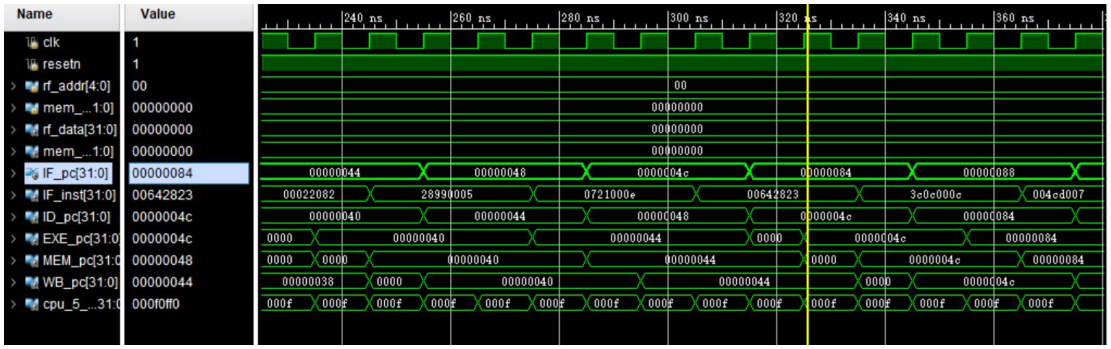
（pipline_cpu.v）：

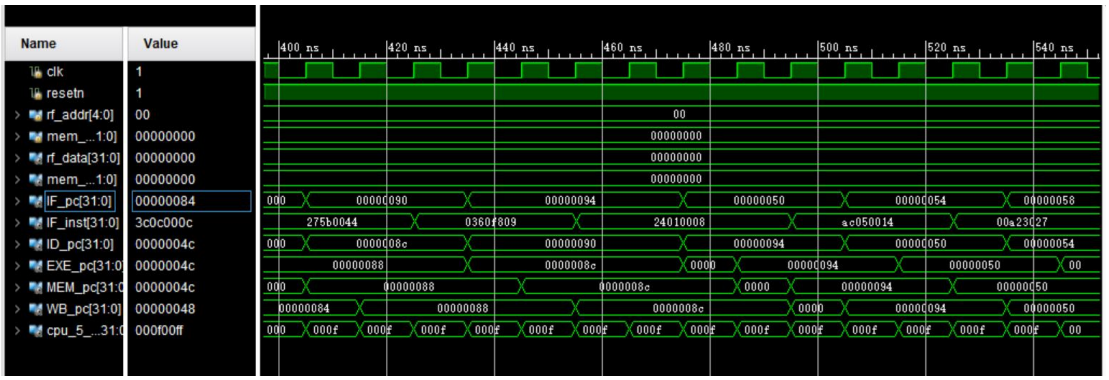
```
    decode ID_module(                    //  译码级
            .ID_valid     (ID_valid     ),   // I, 1
            .IF_ID_bus_r(IF_ID_bus_r),   // I, 64
            . EXE_MEM_bus_r( EXE_MEM_bus_r),
            . MEM_WB_bus_r(MEM_WB_bus_r),
            ......
     );
```

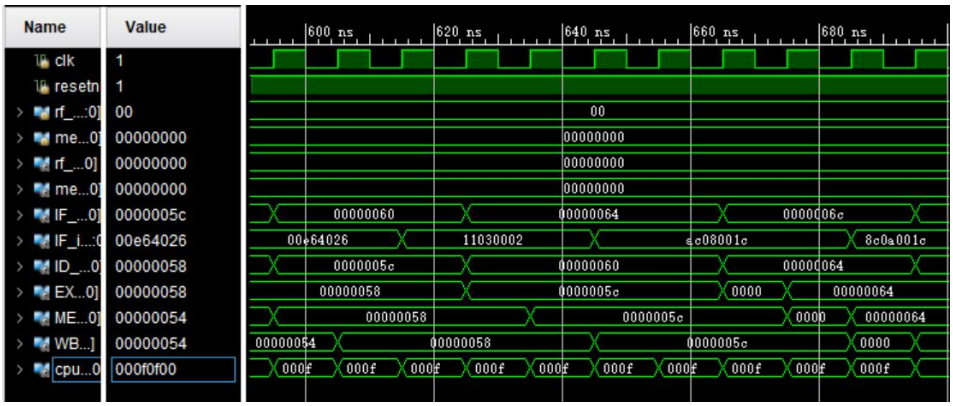
但是可能有些方面的问题，因此优化尚未实现。

五．实验结果分析

1.跳转与分支指令







结果如上图，因为在 ID 阶段跳转和分支指令的实现时，实现了延迟槽：

```
wire [31:0] bd_pc; // 延迟槽指令 PC 值
```

因此，后续指令在跳转指令执行之前继续运行，可在图中看出。

48H	bgez $25,#14	跳转到 84H	0721000E	0000_0111_0010_0001_0000_0000_0000_1110
4CH	subu $5, $3,$4	[$5] = 0000_000DH	00642823	0000_0000_0110_0100_0010_1000_0010_0011
90H	jalr $27	跳转到 50H, [$31] = 0000_0098H	0360F809	0000_0011_0110_0000_1111_1000_0000_1001
94H	addiu $1, $0,#8	[$1] = 0000_0008H	24010008	0010_0100_0000_0001_0000_0000_0000_1000
60H	beq $8, $3,#2	跳转到 6CH	11030002	0001_0001_0000_0011_0000_0000_0000_0010
64H	sw $8, #28($0)	Mem[0000_001CH] = 0000_0011H	AC08001C	1010_1100_0000_1000_0000_0000_0001_1100

结合上面的指令，可以看到正确跳转至 84H，50H 和 6CH 正确执行。

2.WAR 与 RAW

在 pipline_cpu 部分添加下面代码以监测寄存器值的实时变化：

```
// 在 WB 阶段监控寄存器的写入
always @(posedge clk) begin
    if (WB_valid && rf_wen) begin
        // 当写回阶段有效且寄存器写入使能时，更新监控数组
        monitor_registers[rf_wdest] <= rf_wdata;
        $display($time, "ns: Register RF[%d] updated with value %h by instruction at PC %h",
                                    rf_wdest, rf_wdata, WB_pc);
    end
end
// 打印寄存器值的变化
initial begin
    $monitor($time, " RF[%d] <= %h", rf_wdest, rf_wdata);
end
```

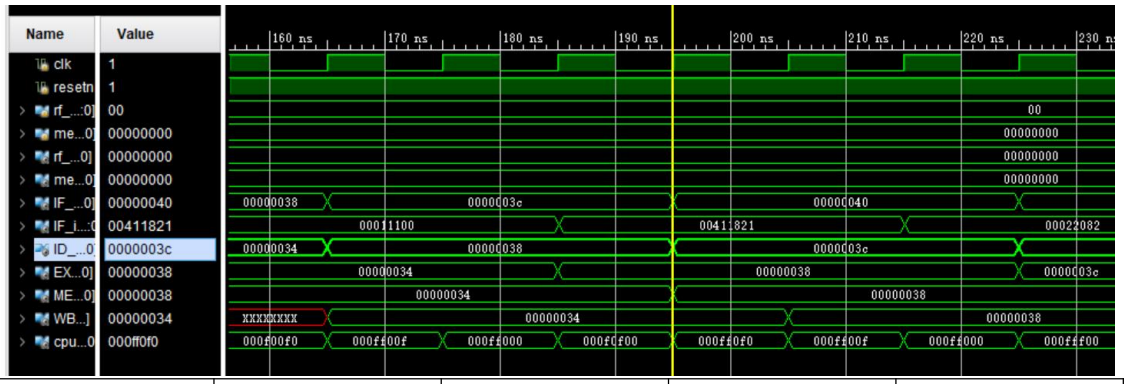
寄存器值变化部分输出如下（指令参考给出的五级流水线 CPU 测试程序）：

```
                    0 RF[ x] <= xxxxxxxx
                  165 RF[ 1] <= 00000001
175ns: Register RF[ 1] updated with value 00000001 by instruction at PC 00000034
                  205 RF[ 2] <= 00000010
215ns: Register RF[ 2] updated with value 00000010 by instruction at PC 00000038
                  245 RF[ 3] <= 00000011
255ns: Register RF[ 3] updated with value 00000011 by instruction at PC 0000003c
                  255 RF[ 4] <= 00000004
265ns: Register RF[ 4] updated with value 00000004 by instruction at PC 00000040
                  295 RF[25] <= 00000001
305ns: Register RF[25] updated with value 00000001 by instruction at PC 00000044
                  335 RF[ 0] <= 00000000
                  345 RF[ 5] <= 0000000d
355ns: Register RF[ 5] updated with value 0000000d by instruction at PC 0000004c
                  375 RF[12] <= 000c0000
385ns: Register RF[12] updated with value 000c0000 by instruction at PC 00000084
                  415 RF[26] <= 0000000c
425ns: Register RF[26] updated with value 0000000c by instruction at PC 00000088
                  455 RF[27] <= 00000050
465ns: Register RF[27] updated with value 00000050 by instruction at PC 0000008c
                  495 RF[31] <= 00000098
505ns: Register RF[31] updated with value 00000098 by instruction at PC 00000090
                  505 RF[ 1] <= 00000008
515ns: Register RF[ 1] updated with value 00000008 by instruction at PC 00000094
                  535 RF[ 0] <= 00000014
                  565 RF[ 6] <= fffffffe2
575ns: Register RF[ 6] updated with value fffffffe2 by instruction at PC 00000054
                  605 RF[ 7] <= ffffffff3
.....
```

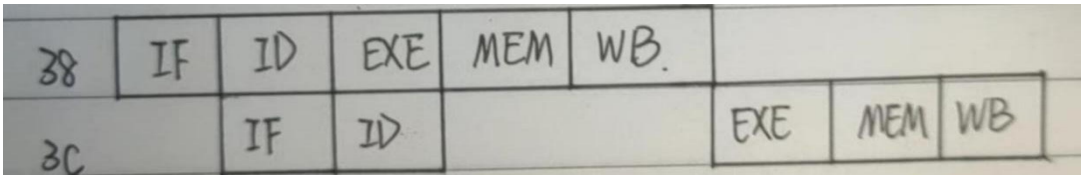
简单分析如下：

38H	sll $2, $1,#4	[$2] = 0000_0010H	00011100	0000_0000_0000_0001_0001_0001_0000_0000
3CH	addu $3, $2,$1	[$3] = 0000_0011H	00411821	0000_0000_0100_0001_0001_1000_0010_0001

PC=38H 和 3CH 两条指令会存在关于[$2]的 RAW 问题，但是 3 号寄存器结果最终为 11H,即前面指令已经把 2 号寄存器的 10H 写入，结果正确，具体通过分析仿真图与 cpu_5_valid 的值可以分析得到如下执行过程：



IF	ID	EXE	MEM	WB
3c	38			
3c		38		
40	3c		38	
40	3c			38
40	3c			
44	40	3c		



因此问题已经被解决。

六．总结感想

通过此次实验，更加深入理解了有关五级流水线中存在的一些问题，包括分支指令，数据冒险等以及如何解决这些问题，包括阻塞等，将理论应用于实践，尝试了前递技术的实现等，同时思考现有的五级流水线 cpu 可有的优化方向，为后面进一步改进五级流水线打下基础。