# 计算机体系结构实验课程第一次实验报告

实验名称		多周期 CPU		班级	李雨森班
学生姓名	姚知言	学号	2211290	指导老师	董前琨
实验地点	实验楼	& A306	实验时间	2024年	9月9日

## 1、 实验目的

在单周期 CPU 实验完成的提前下,理解多周期的概念。

熟悉并掌握多周期 CPU 的原理和设计。

进一步提升运用 verilog 语言进行电路设计的能力。

为后续实现流水线 cpu 的课程设计打下基础

# 2、 实验内容说明

复习单周期 CPU 的实验内容,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;

依据自己设计中实现的指令,编写一段不少于 **40** 行的汇编程序,要求包含所有实现的指令;

认真学习多周期的概念,了解流水线的概念,明白划分为多周期的意义;

认真学习 CPU 各模块的功能,确认模块的划分。设计本次实验的方案,画出实验方案的设计框图;

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;

确认多周期 CPU 的设计框图的正确性;

编写 verilog 代码,将表 8.2 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;

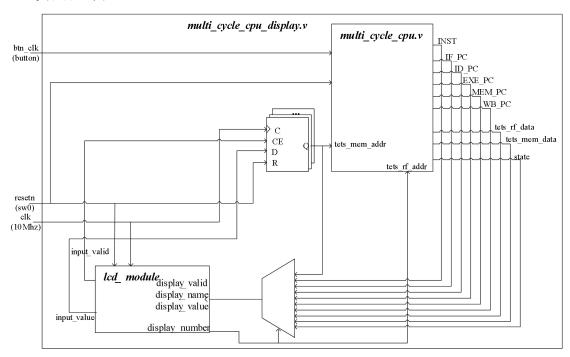
对该模块进行仿真,得出正确的波形,截图作为实验报告结果一项的材料;

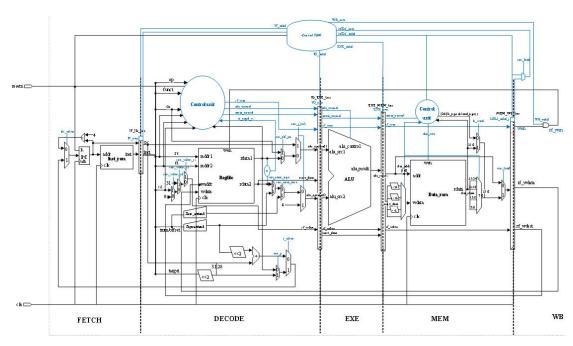
完成调用多周期 CPU 的外围模块的设计,并编写代码;

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。

实验结束后,需按照规定的格式完成实验报告的撰写。

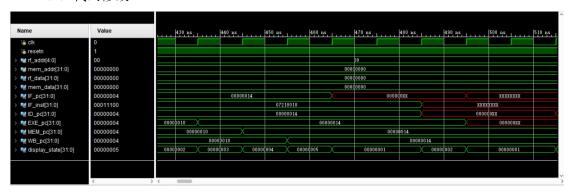
## 3、 实验原理图





## 4、 实验步骤

(1) 代码修改



通过会仿真结果的观察,很容易发现在执行第 0x14 的语句后,取指发生了一些问题,也就是在执行 bgez \$25,#16 跳转后,CPU 应当从译码阶段返回取指阶段。

观察源代码,可以发现其他四条总线都通过触发器保存在了寄存器中,想到可能是总线的保存存在问题,于是想到设置一个jbr\_bus\_r,通过jbr\_bus\_r <= jbr\_bus 维护。

重新仿真,此时全部仿真都变绿了,但是发现在 0x14 之后就变成了 0x18,并没有进行跳转,意识到触发器的保存会慢一个周期,因此需要一个办法等待指令到达后再在 fetch 中读取。

此时,注意到原本的图中 fetch 阶段是经过两个时钟周期的,想到可以借鉴这个设计,将 decode 也改为两个周期,并将触发器的存储改在 ID\_valid 阶段(ID\_over 比 ID\_valid 慢一个时钟周期),这样就可以完成触发器的更新了。

```
//decode.v
always @(posedge clk)
begin
ID_over <= ID_valid;
end
```

(此时还需要为 decode.v 增加一个输入参数 clk,并在 multi\_cycle\_cpu 中同步进行模块调用的修改,否则将会缺少时钟周期的参数报错)

```
//multi_cycle_cpu.v
always @(posedge clk)
if(ID_valid)
begin
    jbr_bus_r <= jbr_bus;
end</pre>
```

此时仍然不能实现预期结果,观察下图 pc,jbr\_target 结果可以发现,jbr\_target 看起来没什么问题,但是 pc 却没有得到更新,意识到更新 pc 的时间可能存在问题。

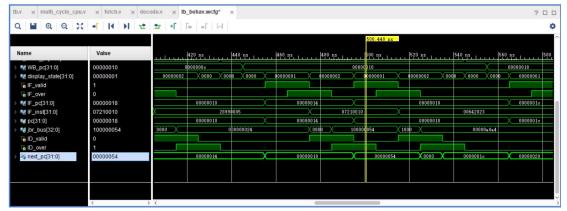
	<del> </del>		<b>4</b>     (a)   (b)	PI								
								510, 560 ns				
lame	Value	42	0 ns	440 ns	460 ns	480 ns	500 ns		520 ns	540 ns	560 ns	580 ns
mem_data[31:0]	00000000						00000000					
→ IF_pc[31:0]	00000018		00000010	X	00000014	X_			00000018		X 000	001c
IF_inst[31:0]	07210010			28990005	X_	07210010		$\rightarrow$		00642823		ac05
ID_pc[31:0]	00000014			00000010	X	00000014		$\supset \subset$		00000018		0000
EXE_pc[31:0]	00000014	000	X	00000	010	X	(	0000014	$\longrightarrow$		00000018	
MEM_pc[31:0]	00000010	00	00000c X			00000010				X	00000018	
₩B_pc[31:0]	00000010		0000000c	X		0000	010			X	00000018	
display_state[31:0]	00000001	000	X 0000 X 000	0 0000	0000001	00000002	00000001	$\rightarrow$	00000002 X	0000 × 0000 × 0	000 × 00000001	X 0000
<pre>jbr_bus_r[32:0]</pre>	100000054	000	X	000000024		X 0000 X	100000	054	(1000 X		00000a0a4	
I jbr_taken	1											
M next_pc[31:0]	00000054		00000014	X	00000018	X	00000	354	X 0000 X	0000001c	( 000	0020
<pre>jbr_target[31:0]</pre>	00000054	000	X	00000024		X 0000 X	00000	354	X 0000 X		0000a0a4	
₩ pc[31:0]	00000018		00000010	X	00000014	X			00000018		X 000	001c

```
//fetch.v
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
```

从中可以发现,设置合理的 next\_fetch 是解决问题的关键。

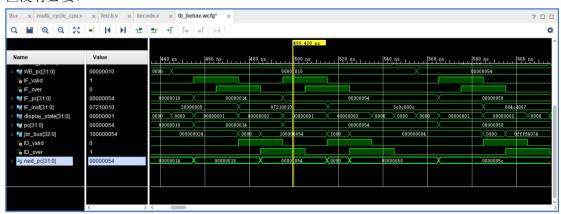
起初设想过,将 next\_fetch 去掉,直接在 IF 阶段开始的时候(第一次失败分析原因后,还为 jbr 总线增加了 resetn 初始值)读取指令,但是引发了更严重的问题(主要是对于经过5个阶段后跳转的指令出现了问题)。

后放弃该设想,转而调整 next 的判定条件,然而无论是将 ID\_over 调整为 ID\_valid 还是保持不变,都无法在规定时间上完成更新。



从图中可以看到,虽然 next\_pc 能及时完成更新,但始终慢上一个始终周期,这是 0x18 已经进入取指阶段了,并不合理。

最终想到,如果不使用寄存器而使用 wire 完成更新,就不存在这个问题了,于是删除 jbr\_bus\_r, 回调传入 fetch 的参数。(可能另一种解决方案是把 decode 改为 3 周期保障更新,但没有必要)



由此,next\_pc 可以和 ID\_over 在同一时间周期完成更新,保证了 PC 在取指之前的顺利 更新,完成修改。

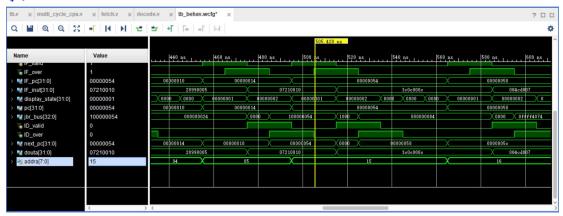
修改的全部代码如下:

```
//decode.v
`timescale 1ns / 1ps
module decode(
                                 // 译码级
   input
                      clk,
                     ID valid,
                               // 译码级有效信号
   input
   input
             [63:0] IF_ID_bus_r, // IF->ID 总线
             [ 31:0] rs_value,
                             // 第一源操作数值
   input
             [ 31:0] rt_value,
                             // 第二源操作数值
   input
             [ 4:0] rs,
                              // 第一源操作数地址
   output
             [ 4:0] rt,
                              // 第二源操作数地址
   output
                              // 跳转总线
   output
             [ 32:0] jbr_bus,
                     jbr_not_link,// 指令为跳转分支指令,且非 link 类指令
   output
   output
              reg
                     ID_over,
                                // ID 模块执行完成
//增加一拍时延,保证数据传输
             [149:0] ID EXE bus, //ID->EXE 总线
   output
```

```
//展示 PC
    output
              [31:0] ID pc
);
//-----{ID 执行完成}begin
    //为保证数据传输,需要 ID 是两周期的
    always @(posedge clk)
    begin
        ID_over <= ID_valid;
    end
//----{ID 执行完成}end
endmodule
(2) 指令 ROM 中指令执行过程分析
为区别原有注释,个人分析标蓝处理。
//multi_cycle_cpu.v
    inst_rom inst_rom_module(
                                  // 指令存储器
                               ), // I, 1, 时钟
        .clka
                  (clk
        .addra
                  (inst_addr[9:2]), // I, 8,指令地址
                                   // O, 32,指令
        .douta
                  (inst
                               )
    );
```

这里是指令 ROM 在上层模块 multi\_cycle\_cpu 中的调用,可以看出指令 ROM 读入的共有两个参数,clk 时钟不必多说,addra 读取的值 inst\_addr[9:2]指令地址,输出 douta 为 inst 指令。

到这里,其实已经较为明确了,由于一条指令的长度是 4 个字节,读取 inst\_addr[9:2] 即去掉后两位,对应的位置刚好是设计的.coe 文件的行数,对应行数的指令返回输出就可以了。



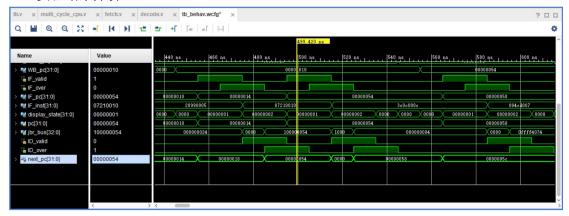
对应 display\_state 来说,我们可以发现每一次 addra 的更新时间都是 IF 阶段开始的时刻,和 PC 的更新时间是同步的,具体 PC 的更新时间也就是上一部分中提到了 next\_fetch 的位置,在 IF 的前一个时钟周期开始,刚好在 IF 开始的时候完成更新,而对应的 douta 的更新时间是每次 IF 结束 ID 开始的时候。

也正是因为 rom 取指需要有一拍的时钟周期时延,取指阶段才需要 2 拍的时间。//fetch.v

```
// 取指级
   module fetch(
                               // 时钟
       input
                      clk,
                               // 复位信号, 低电平有效
       input
                      resetn,
                      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
       input
                [31:0] inst,
                             // inst_rom 取出的指令
                             // 跳转总线
                [32:0] jbr bus,
       input
                [31:0] inst_addr, // 发往 inst_rom 的取指地址
       output
                         IF over, // IF 模块执行完成
       output
                [63:0] IF_ID_bus, // IF->ID 总线
       output
       //展示 PC 和取出的指令
       output
                [31:0] IF_pc,
       output
                [31:0] IF_inst
   );
   简单观察以下 fetch 的输入输出,fetch 的输出 inst_addr 对应的就是发往 rom 的取指地
址,而最后输出的 IF PC 和它也没有任何差别,从赋值语句中就可以看出来。
   //fetch.v
   assign inst_addr = pc;
   assign IF_pc =pc;
   assign IF inst = inst;
   而 fetch 阶段最终返回的 IF_inst, 也就是指令 ROM 的返回结果。
   总的来说, fetch 把 pc 参数带给指令 ROM, ROM 去掉后两位后存储文件.coe 中找到对
```

### 5、 实验结果分析

应的指令, 最终再给到取指阶段打包。



在 0x14 执行 2 周期后顺利跳转到 0x54。 0x14 的指令是 bgez \$25,#16,此时\$25 的值是 1,大于等于 0,向后跳转 16 个,刚好是 0x54,完成了修改。

. ■ 0 0 %	<b>■</b>	<u> </u>	al [H]							
	1								10, 981, 950 ns	
ame	Value	10,840 ns	10,860 ns	10,880 ns	10,900 ns	10, 920 ns	10,940 ns	10, 960 ns	1), 980 ns	11,000 ns
₩ WB_pc[31:0]	00000060		00000058	X		00000050		X 000	0 1060	
I IF_valid	1									
<b>%</b> [31][31:0]	00000064			0	0000054			$\longrightarrow$ X	00000064	
I IF_over	0									
<pre>IF_pc[31:0]</pre>	00000018	X	0000005c		X	0000	060	$\sim$	00000018	
IF_inst[31:0]	0360f809	004cd007		003ad804		$\rightarrow$	0360			
display_state[31:0]	00000001	00000001	00000002 000	0 0000 000	00000001	00000002	X 0000 X 00	00 0000 000	0000001	
pc[31:0]	00000018	X	00000050		X		060	-	00000018	
<b>  </b> jbr_bus[32:0]	100000018	0ffff4074	Offf	Offff606c		X Offf X		100000018		
In ID_valid	0									
In ID_over	0									
<pre>mext_pc[31:0]</pre>	00000018	X	00000060		0000	d064 X		00000018		

程序在 11000ns 中已经运行多轮,并没有发生崩溃。而且图中 0x60 的指令应该为 jalr \$27, 实现效果是跳转到寄存器\$27 的数值(18H),完成了该操作(说明之前的修改并没有影响 5 周期跳转的实现)。同时查看图中高亮部分,可以发现\$31 确实被修改为 0x64。

# 6、 总结感想

在这次实验中,我完成了多周期 CPU 的理解,并进行了 Bug 的修复以及 rom 指令的介绍等。虽然在 bug 的修复上走了一些弯路,然后发现最终实际需要修改的代码量如此之少,但是在此过程中,我也更加对多周期 CPU 加深了一些理解。完成实验后我也曾设想过是否可以仅对这一种特殊指令增加时延,但考虑到后期将要实现流水线 CPU,无论如何都是要迁就最长时间的阶段的,似乎纠结于这个问题的意义并不是很大。

无论如何,这次实验让我深刻熟悉了多周期 CPU 的架构,也为后续实验打下良好的基础。

## 注意: 1、班级用任课老师姓名表示。

2、实验报告提交的文件名为"学号\_姓名\_组成原理第一次实验.pdf",注意要导出成 pdf 文件。