处理器大作业第二部分

无 08 李煜彤 2020010841

目录（点击跳转）

1 MIPS 单周期 CPU 实现

1.a 控制器模块设计

真值表：

Instruction	PCSrc[1:0]	Branch	RegWrite	RegDst[1:0]	MemRead	MemWrite	MemtoReg[1:0]	ALUSrc1	ALUSrc2	ExtOp	LuOp
lw	00	0	1	01	1	0	01	0	1	1	0
sw	00	0	0	x	0	1	x	0	1	1	0
lui	00	0	1	01	0	0	00	0	1	x	1
add	00	0	1	00	0	0	00	0	0	x	x
addu	00	0	1	00	0	0	00	0	0	x	x
sub	00	0	1	00	0	0	00	0	0	x	x
subu	00	0	1	00	0	0	00	0	0	x	x
addi	00	0	1	01	0	0	00	0	1	1	0
addiu	00	0	1	01	0	0	00	0	1	1	0
and	00	0	1	00	0	0	00	0	0	x	x
or	00	0	1	00	0	0	00	0	0	x	x
xor	00	0	1	00	0	0	00	0	0	x	x
nor	00	0	1	00	0	0	00	0	0	x	x
andi	00	0	1	01	0	0	00	0	1	0	0
sll	00	0	1	00	0	0	00	1	0	x	x
srl	00	0	1	00	0	0	00	1	0	x	x
sra	00	0	1	00	0	0	00	1	0	x	x
slt	00	0	1	00	0	0	00	0	0	x	x
sltu	00	0	1	00	0	0	00	0	0	x	x
slti	00	0	1	01	0	0	00	0	1	1	0
sltiu	00	0	1	01	0	0	00	0	1	1	0
beq	00	1	0	x	0	0	x	0	0	1	0
j	10	0	0	x	0	0	x	x	x	x	x
jal	10	0	1	10	0	0	10	x	x	x	x
jr	11	0	0	x	0	0	x	x	x	x	x
jalr	11	0	1	00	0	0	10	x	x	x	x

说明：

PCSrc：PC Source，取指令过程控制。00-顺序执行或 branch，PC = PC + 4；10-对于 j 和 jal，会跳到 target；11-对于 jr 和 jalr，会跳到$rs。（原本想为 beq 单独设置一个 PCSrc 的值，但是发现如果这样 branch 没有用武之地）

RegWrite：是否写入寄存器。jal 和 jalr 需要，因为要将某个寄存器写入当前 PC 位置。

RegDst：写入的寄存器。00-对于 R 型指令和 jalr，需写入 Rd；01-对于 I 型指令，要写入 Rt；10-对于 jal 需写入$31。若 RegWrite 为 0，则 RegDst 为 x。

MemRead：是否读内存。只有 lw 需要。

MemWrite：是否写入内存。只有 sw 需要。

MemtoReg[1:0]：判断写回寄存器的来源。00-将 ALU 计算的结果写回寄存器；01-将内存的结果写回寄存器，只有 MemRead 为 1 时，MemtoReg 才有可能是 01；10-将 PC+4 写入寄存器。若 RegWrite 为 0，则 MemtoReg 为 x。（注：为什么 lui 也是 alu 的结果呢？因为 lui 是先进行了扩展后与$0 相加）

ALUSrc1：ALU 第一个操作数的来源。0-rs 寄存器；1-shamt，因此只有 sll、srl 和 sra 的 ALUSrc1 是 1。对于跳转指令，由于无需经过 ALU，因此为 x。

ALUSrc2：ALU 第二个操作数的来源。0-rt 寄存器；1-立即数。

ExtOp：立即数扩展方式。0-0 扩展，只有 andi 是 0 扩展；1-符号扩展。对于非 I 型指令，为 x。Lui 由于不需要扩展后的高位，因此也是 x。

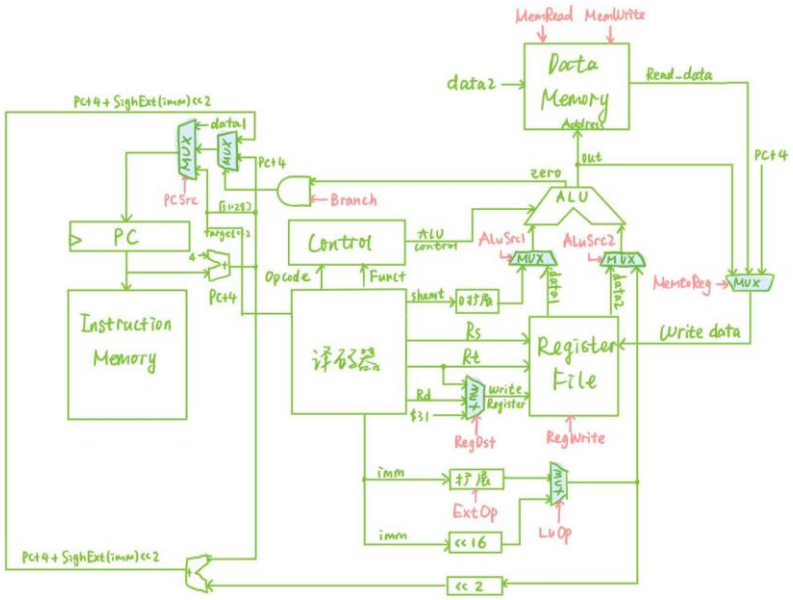
LuOp：判断是否是 lui 指令。对于非立即数的指令，LuOp 可以为 x。

1.b 数据通路设计

在 CPU 顶层设计中，预计将实例化六个模块，它们的功能和接口如下：

名称	说明	输入	输出
ALU	算数逻辑单元	ALUCtrl, Sign, in1, in2	out，zero
ALUControl	ALU 的功能控制	OpCode, Funct	ALUCtrl, Sign
Control	控制单元	OpCode, Funct	控制信号
DataMemory	内存	Address, Write_data, MemRead, MemWrite	Read_data
InstructionMemory	指令内存	Address	Instruction
RegisterFile	寄存器堆	RegWrite，Read_register1, Read_register2, Write_register, Write_data	Read_data1, Read_data2

根据控制信号，设计数据通路如下：



其中的多路选择器共有 7 个，在代码中主要通过三元运算符？：来体现。它们具体为：

- 用于选择扩展后的立即数还是左移 16 位后的数据。主要用于区分 lui 和其它的 I 型指令。选择信号为 LuOp
- 用于选择写入寄存器堆的寄存器是 Rt、Rd 还是$31。选择信号为 RegDst。
- 用于选择 ALU 的第一个操作数为寄存器读出的 data1 还是扩展后的 shamt。选择信号为 ALUSrc1。
- 用于选择 ALU 的第二个操作数为 data2 还是经过挑选的立即数（LuOp 控制的选择器的结果）。选择信号为 ALUSrc2。
- 用于选择写回寄存器堆的数据为内存读取的数据、ALU 计算的结果还是 PC+4。选择信号为 MemtoReg。
- 用于选择 PC+4 还是 PC+4+offset。主要用于区分顺序执行还是 beq 的 branch。选择信号为 zero&&Branch。
- 用于选择写入 PC 的值为上一个选择器的结果、data1 还是 target。主要用于区分跳转指令、跳转的 R 型指令还是其它指令。选择信号为 PCSrc。

1.c 汇编程序分析-1

这段程序运行足够长的时间后，将不断循环最后一条语句。运行过程中，由于 beq，跳过第 5 条指令。

MIPS Assembly 1	
0	addi $a0, $zero, 12123
1	addiu $a1, $zero, -12345
2	sll $a2, $a1, 16
3	sra $a3, $a2, 16
4	beq $a3, $a1, L1
5	lui $a0, 22222
	L1:
6	add $t0, $a2, $a0
7	sra $t1, $t0, 8
8	addi $t2, $zero, -12123
9	slt $v0, $a0, $t2
10	sltu $v1, $a0, $t2
	Loop:
11	j Loop

最后的运行结果：

$a0 0x0000_2f5b

$a1 0xffff_cfc7

$a2 0xcfc7_0000

$a3 0xffff_cfc7

$t0 0xcfc7_2f5b

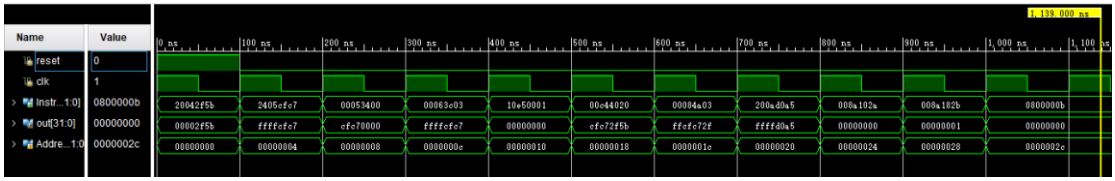
$t1 0xffcf_c72f

$t2 0xffff_d0a5

$v0 0x0000_0000

$v1 0x0000_0001

添加关键信号至波形图，仿真结果如下：



可以看到 Address 为指令地址，跳过了 0x00000014，并在执行完最后一条指令后一直不变，与预期相同；out 也为相应的计算结果。通过查看寄存器堆的值，亦与此前计算结果相同。

Debug:

由于在做大作业第一部分的时候不太清楚 sll、srl 和 sra 的工作原理，因此把 shamt 当作了 ALU 的第二个操作数；而在做这部分作业时，我直接将之前的代码复制了过来，导致了结果的不正确。通过一步一步追查错误的结果进行了修正。

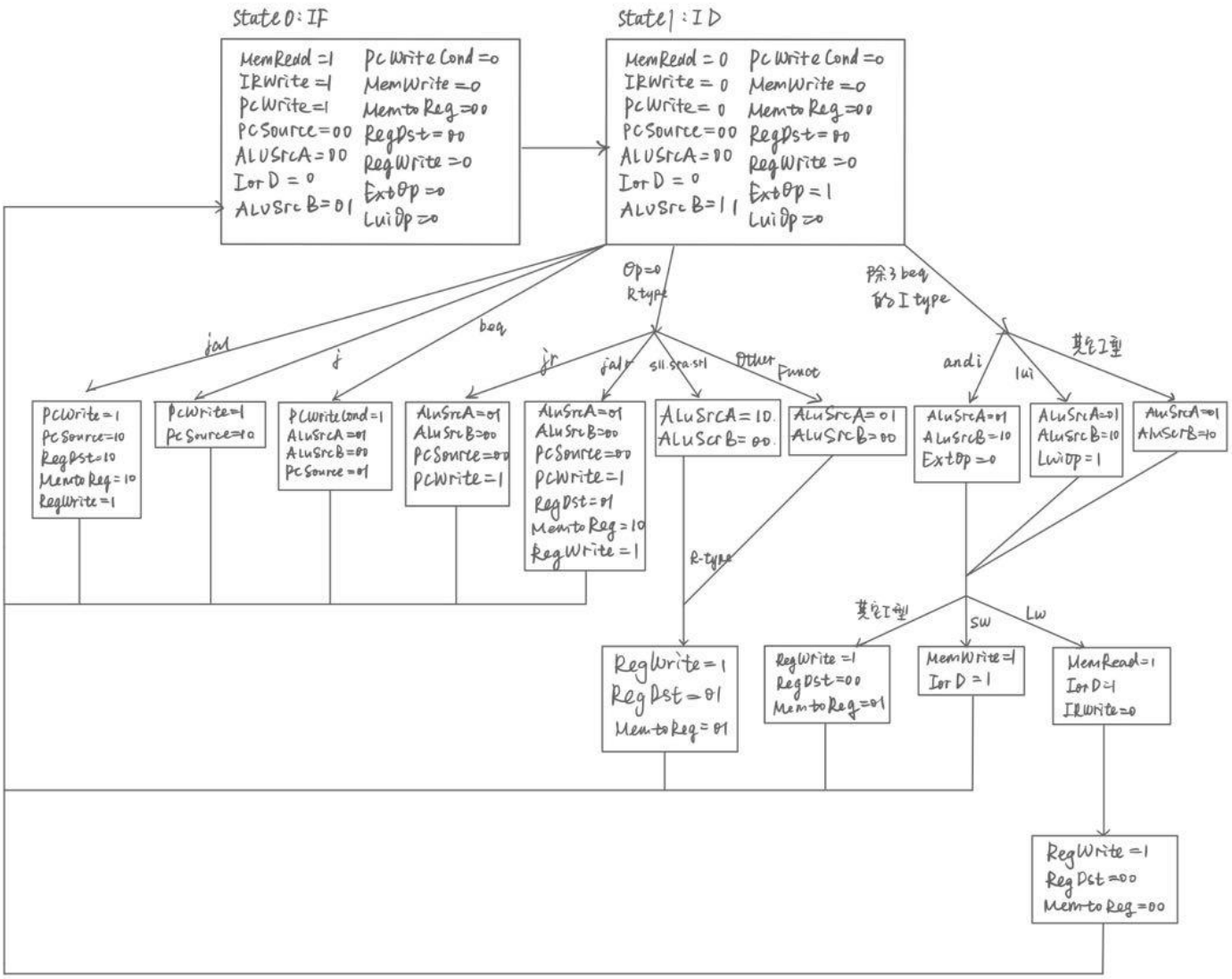
2 MIPS 多周期 CPU 实现

2.a 多周期状态机控制器

注：

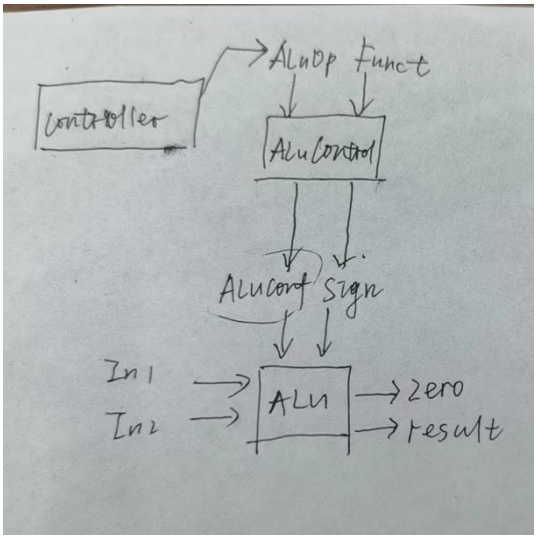
ALUSrcA：00-PC，01-Rs，10-Shamt

ALUSrcB：00-Rt，01-4，10-ImmExt，11-ImmExt<<2



2.b 多周期 CPU 的 ALU 控制逻辑与功能实现

根据其输入输出端口，得到 controller、AluControl 和 ALU 的关系：



可知 controller 的输出 ALUOp 主要用于区分非 R 型指令；Funct 和 ALUOp 作为 ALUControl 的输入，共同觉得 ALU 此时做什么操作。因此，结合上述分析和课上给出的简单的多周期处理器状态转移图做如下设计：

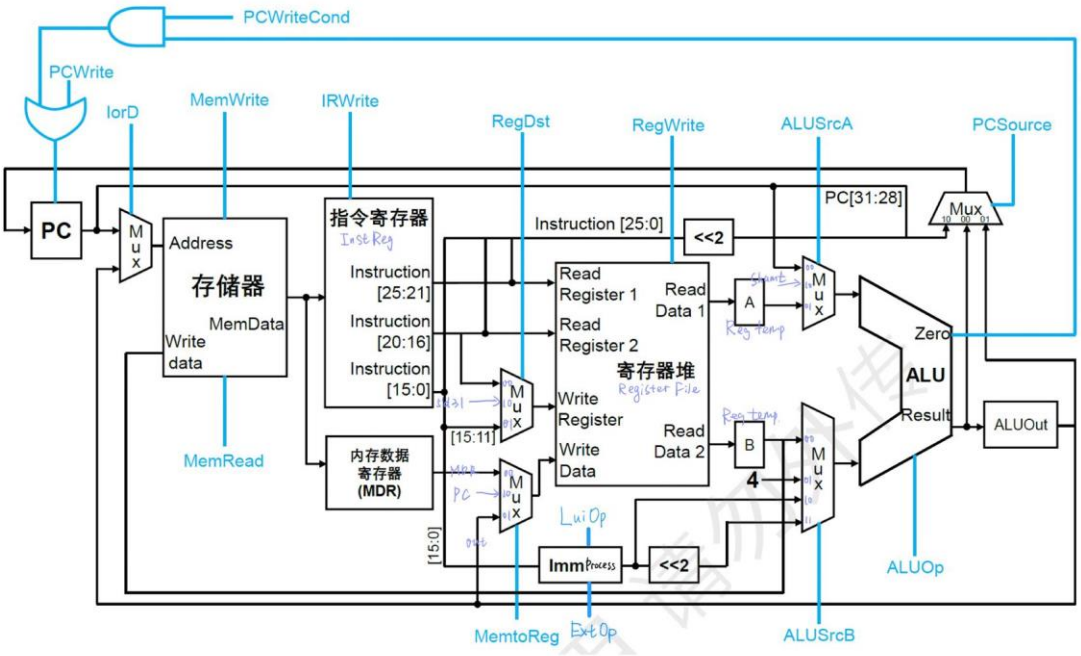
0000	加（包括 IF、ID 阶段）
0001	减（beq 等）
0010	R 型指令（Opcode 为 0）
0011	与（用于 andi）
0100	slti
0101	sltiu

ALUControl 的设计同单周期类似，只不过由 Opcode 的输入变为了 ALUOp，其余设计不变。

ALU 的设计也不用进行更改（除了变量名？）。

2.c 数据通路设计

根据控制信号，根据教材中的电路图，设计（修改）数据通路如下：



其中的多路选择器共有 6 个，在代码中主要通过三元运算符？：来体现。它们具体为：

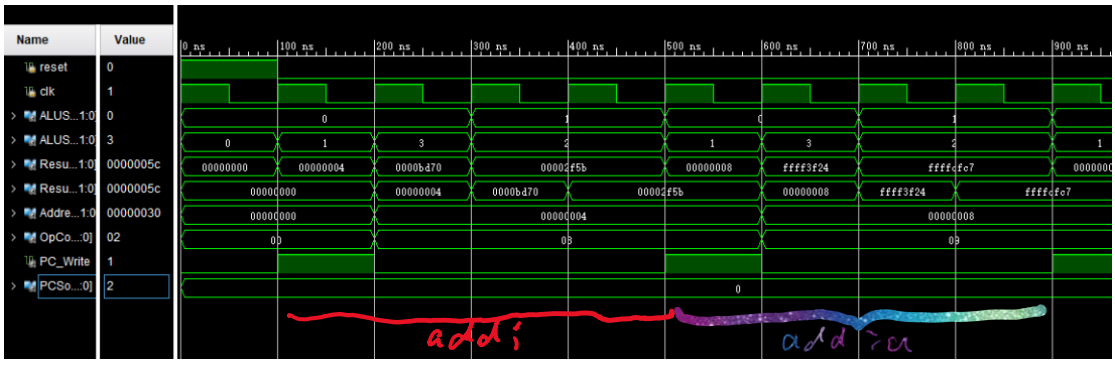
- 用于选择写入寄存器堆的寄存器是 Rt、Rd 还是$31。选择信号为 RegDst。
- 用于选择 ALU 的第一个操作数为寄存器读出的 data1、PC 还是扩展后的 shamt。选择信号为 ALUSrc1。
- 用于选择 ALU 的第二个操作数为 data2、4、ImmExt 还是 ImmExt<<2 经过挑选的立即数。选择信号为 ALUSrc2。
- 用于选择写回寄存器堆的数据为内存读取的数据、ALU 计算的结果还是 PC+4。选择信号为 MemtoReg。
- 用于选择写入 PC 的值为 ALU 的计算结果、ALU 寄存器的结果还是 target。主要用于区分跳转指令、跳转的 R 型指令还是其它指令。选择信号为 PCSrc。
- 用于选择存储器的地址为指令还是数据。选择信号为 IorD。

除了寄存器堆的寄存器，还有如下寄存器：

- PC 寄存器，暂存 PC 的值。
- 指令寄存器，暂存指令的值。
- MDR，暂存从 memory 中读取的数据。
- 寄存器堆后的数据寄存器，暂存从寄存器堆的数据。
- ALUOut 寄存器，暂存 ALU 的计算结果。

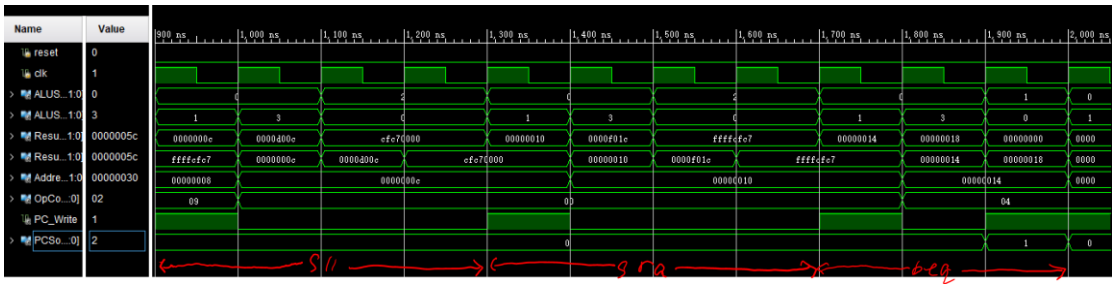
2.d 功能验证（汇编程序分析-1）

进行仿真，结果如下：



100~500ns：指令 0，addi。最后的结果为 0x00002f5b，与此前预期结果相同。

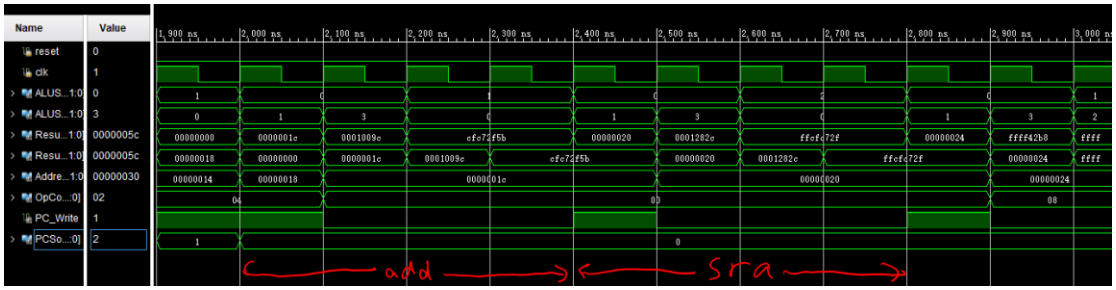
500~500ns：指令 1，addiu。最后的结果为 0xffffcfc7，与此前预期结果相同。



900~1300ns：指令 2，sll。最后的结果为 0xcfc70000，与此前预期结果相同。

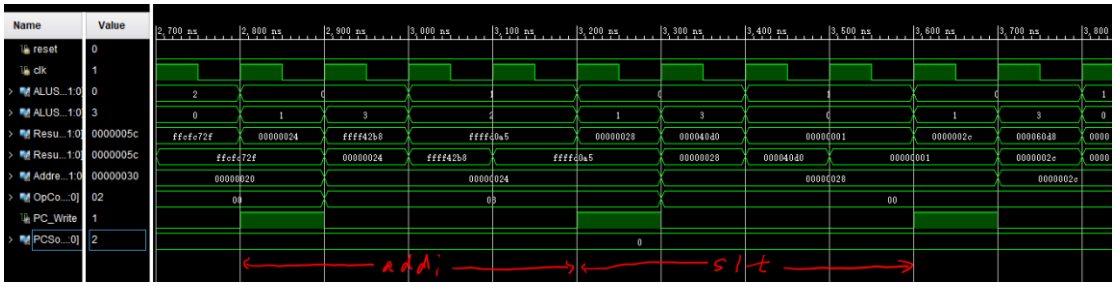
1300~1700ns：指令 3，sra。最后的结果为 0xffffcfc7，与此前预期结果相同。

1700~2000ns：指令 4，beq。最后的结果为 0x00000000，与此前预期结果相同。



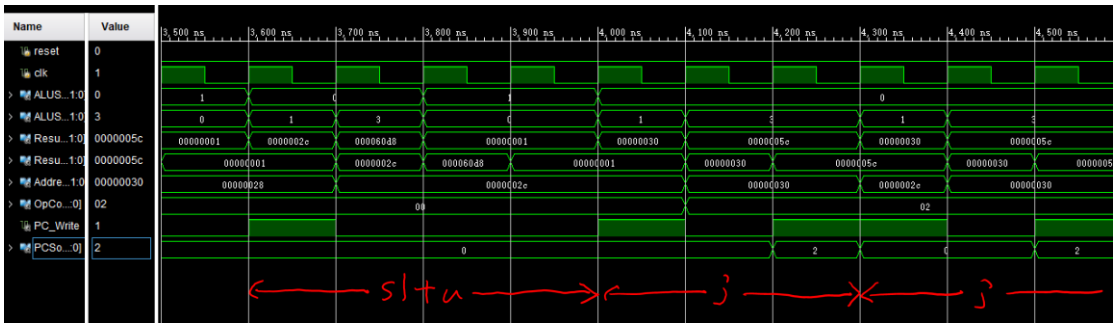
2000~2400ns：指令 6，add。最后的结果为 0xcfc72f5b，与此前预期结果相同。

2400~2800ns：指令 7，sra。最后的结果为 0xffcfc72f，与此前预期结果相同。



2800~3200ns：指令 8，addi。最后的结果为 0xffffd0a5，与此前预期结果相同。

3200~3600ns：指令 9，slt。最后的结果为 0x00000000，与此前预期结果相同。



3600~4000ns：指令 10，sltu。最后的结果为 0x00000001，与此前预期结果相同。

4000~4300ns 及之后：指令 11，j。后续的 OpCode 一直为 02 不变，符合预期。

[10][31:0]	ffffd0a5	Array
[9][31:0]	ffcfc72f	Array
[8][31:0]	cfc72f5b	Array
[7][31:0]	ffffcfc7	Array
[6][31:0]	cfc70000	Array
[5][31:0]	ffffcfc7	Array
[4][31:0]	00002f5b	Array
[3][31:0]	00000001	Array
[2][31:0]	00000001	Array
[1][31:0]	00000000	Array

检查运行一段时间后寄存器的值，亦与计算结果（以及单周期运行结果）相同，可以基本验证程序的正确性。

Debug：

在进行 j 的跳转地址计算以及立即数计算时，一开始仿照单周期进行，直接把单周期的“instruction”替换成了从存储器中读取的值，导致立即数没有进行正确的扩展、j 的跳转地址不正确的问题。修改为从 InstReg 出来后的值，问题得到了解决。

3 MIPS 单周期和多单周期 CPU 的性能对比

3.a 汇编程序分析-2

i)

如果第 0 行的 5 是任意正整数 n，则在程序执行足够长的时间后，可以完成计算 n*(n+1) 的功能，并将结果保存在$v0 中。

Loop：将程序停留在这一行并持续执行，起到了结束程序的作用。

sum：整体实现一个累加的过程，也即从 n、n-1 一直加到 1，在逆序加回来。

L1：作为 sum 的其中一个过程，主要实现了加的功能。

注释：

```
addi $a0, $zero, 5          // 将$a0 赋值为 5
xor $v0, $zero, $zero       // 将$v0 赋值为 0
jal sum                     // 跳转至 sum 并记录当前指令地址
Loop:
beq $zero, $zero, Loop      // 一直循环在这一步
sum:
addi $sp, $sp, -8           // $sp 相当于一个数据的指针；后移两位
sw $ra, 4($sp)              // 将$ra 的值存入$sp 的前一个位置
sw $a0, 0($sp)              // 将$a0 的值存在$sp 的当前位置
slti $t0, $a0, 1            // 若$a0>=1，$t0 为 0。若$a0 的值达到 0，则$t0 为 1
beq $t0, $zero, L1          // 若$a0>=1，$t0 为 0，跳转到 L1。若$a0 的值达到 0，则$t0 为
1，不跳转
addi $sp, $sp, 8            // $sp 前移两位
jr $ra                      // 跳转到$ra 所在位置
L1:
add $v0, $a0, $v0           // $v0 为$a0 的值的累加
addi $a0, $a0, -1           // $a0 --
jal sum                     // 跳转至 sum 并记录当前指令地址
lw $a0, 0($sp)              // 将$sp 的值存入$a0
lw $ra, 4($sp)              // 将$sp 的上一个值存入$ra
addi $sp, $sp, 8            // $sp 前移两位
add $v0, $a0, $v0           // $v0 为$a0 的值的累加
jr $ra                      // 跳转到$ra 所在位置
```

ii)

机器码：

0x20040005
0x00001026
0x0c100004
0x1000ffff
0x23bdfff8
0xafbf0004
0xafa40000
0x28880001
0x11000002
0x23bd0008
0x03e00008
0x00821020
0x2084ffff
0x0c100004
0x8fa40000
0x8fbf0004
0x23bd0008
0x00821020
0x03e00008

以上为根据 MARS 的结果。

这无疑是一种取巧的办法，但这种取巧的正确性有赖于：

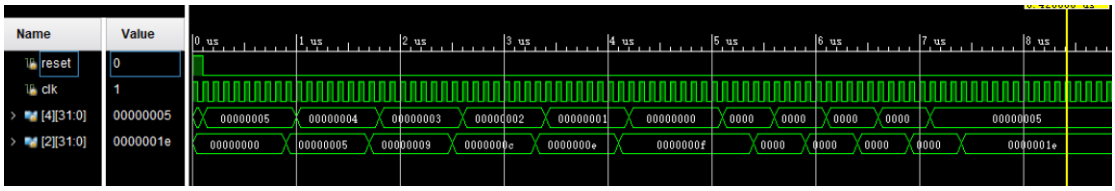
1. MIPS 的统一
2. 存储空间不大，因此不需要考虑较高位地址

iii)

将修改后的机器码放到 InstructionMemory2.v、InstAndDataMemory2.v 中。

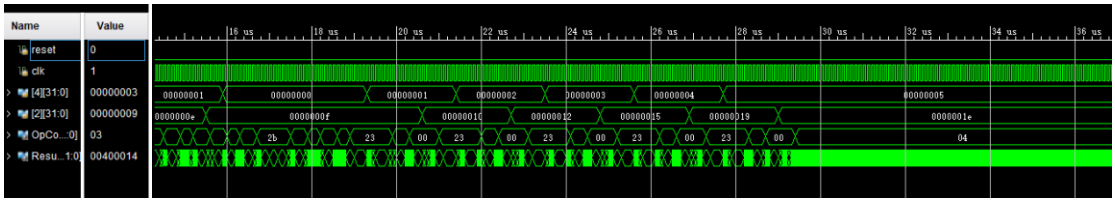
$a0 为 4 号寄存器，$v0 为 2 号寄存器。在窗口添加对应的寄存器，结果如下：

单周期：



经过足够长的时间后，$a0 的值变为 5，$v0 的值变为 30。与预期相符。

多周期：



经过足够长的时间后，$a0 的值变为 5，$v0 的值变为 30。与预期相符。

Debug:

在多周期的仿真时，发现 jr 跳不到正确的地方。添加 ALU 计算的结果发现，计算结果为 0，而不是应当跳转的地址。检查 ALUControl，发现忘记写了 jr 和 jalr 的相加。添加相应的代码后，问题得到了解决。

3.b 资源与性能对比

发现单周期进行综合分析之后，netlist 为 empty。其它同学提供了增加输出端口（PC）的办法，试了一下，可以；为了进行对比，在多周期中也添加了相应的信号。在数逻实验四的提示中，提到为了防止优化过度，可以将 vo、ao、sp、ra 加入输出端口。因此根据此修改了代码。

静态时序分析：

单周期：

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	85.975 ns	Worst Hold Slack (WHS):	0.323 ns	Worst Pulse Width Slack (WPWS):	49.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	18400	Total Number of Endpoints:	18400	Total Number of Endpoints:	9217

多周期：

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	91.396 ns	Worst Hold Slack (WHS):	0.134 ns	Worst Pulse Width Slack (WPWS):	49.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	18770	Total Number of Endpoints:	18770	Total Number of Endpoints:	9503

占用资源情况：

单周期：

Resource	Utilization	Available	Utilization %
LUT	3928	20800	18.88
FF	9230	41600	22.19
IO	162	106	152.83

多周期：

Resource	Utilization	Available	Utilization %
LUT	4079	20800	19.61
FF	9503	41600	22.84
IO	162	106	152.83

可以看到，多周期的延时较长，占用的资源也比较多。但是理论上学到，多周期由于资源的复用，可以节约资源。究其原因，我认为可能是由于这次作业中涉及的指令数都比较少、相对比较简单，因此单周期、多周期的资源占用情况几乎相同，无法体现出多周期的资源优势。

单周期：

可能达到的最高频率：

$$\frac{1}{85.975ns + 0.323ns} \approx 11.59GHz$$

单次计算所需要的最低延时：

最短时钟周期为：

$$85.975 + 0.323 = 86.298ns$$

执行该指令至少需要 75 个周期。则单次计算需要的最低延时：

$$86.298ns \times 76 = 6{,}558.648ns$$

多周期

可能达到的最高频率：

$$\frac{1}{91.396ns + 0.134ns} \approx 10.93GHz$$

单次计算所需要的最低延时：

最短时钟周期为：

$$91.013ns + 0.139ns = 91.530ns$$

执行该指令至少需要 296 个周期。则单次计算需要的最低延时：

$$91.530ns \times 296 = 27092.880ns$$

实验小结

这次作业带给我的感受就是“具体”，一旦具体思考、写代码，就会发现许多问题与细节，从而对处理器的控制信号、数据通路、计算过程等等有了更深刻的理解。