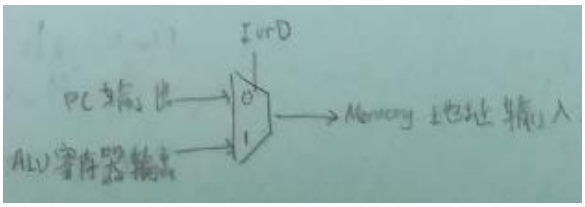
多周期 CPU 大作业

林宝诚 2019080030 无 98

1. 多路选择器

- 储存器地址输入

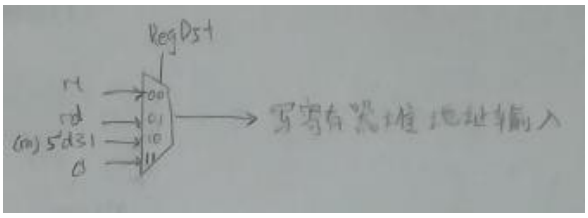
```
//mux for memory address input
always@(*)
begin
case (i_IorD_s)
1'b0    :mem_inputaddr<=PC_output;
1'b1    :mem_inputaddr<=ALUOut_Reg;
endcase
end
```



功能：由于指令和数据都存放在同一个储存器，因此需要 MUX 来选择储存器的输入地址，地址可能来自 PC（指令地址）或者来自 ALU 的输出（lw，sw 用到的数据地址）

- 寄存器堆的写入地址输入

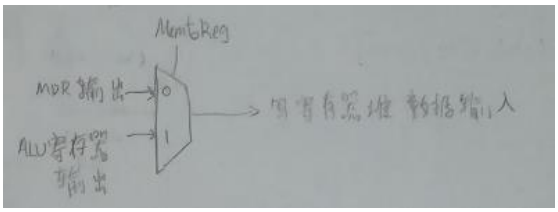
```
//mux for register file address input
always@(*)
begin
case (i_RegDst_s)
    2'b00   : mux_regwrite_addr<= rt;
    2'b01   : mux_regwrite_addr<= rd;
    2'b10   : mux_regwrite_addr<= 5'd31;        //$ra
    default : mux_regwrite_addr<=5'd0 ;         //dont care
    endcase
end
```



功能：由于寄存器堆的写入地址有多种可能性，因此需要 MUX 对地址进行选择。其中 rd 是 R 型指令的写入地址；rt 是 I 型指令的写入地址；常数 31 为 ra 的地址，用于 jal 指令；最后的常数 0 在这里并没有实际意义，仅用来填上 2 比特的最后一个可能性。

- 寄存器堆的写入数据输入

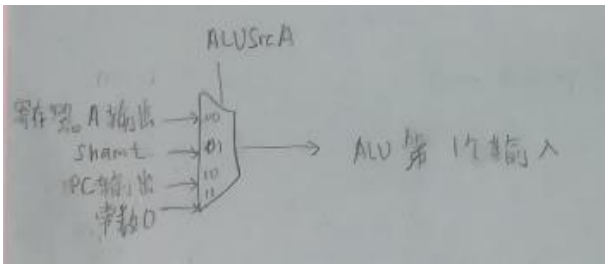
```
//mux for register file data input
always@(*)
begin
case (i_MemtoReg_s)
1'b0    : mux_regwrite_data<= MDR_output;       // for lw
1'b1    : mux_regwrite_data<= ALUOut_Reg;
endcase
end
```



功能：由于寄存器堆的写入数据可能来自不同数据通路，因此需要 MUX 对数据进行选择。其中，MDR 输出用于 lw 指令，ALU 寄存器数据则用于其他需要写数据进寄存器堆的指令。

- ALU 第一个输入

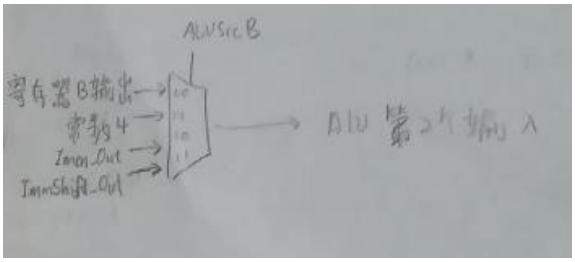
```
//mux for ALU two inputs
always@(*)
begin
case (i_ALUSrcA_s)
2'b00    : mux_ALU_inputA<=reg_a;
2'b01    : mux_ALU_inputA<={28'd0,shamt};
2'b10    : mux_ALU_inputA<=PC_output;
2'b11    : mux_ALU_inputA<=32'd0;

endcase
```



功能：ALU 输入有多种可能源头，因此需要 MUX 进行选择。其中，寄存器 A 输出是地址为 Rs 的寄存器堆输出，用于 R 型指令和一些 I 型指令；shamt 为 sll, srl, sra 指令中的位移量（0 扩展，为了满足 32bits）；PC 输出用于 PC+4 和其他需要计算跳转地址的指令；常数 0 用于 lui 指令，此举只是为了是 lui 需要的计算结果出现在 ALU 输出，否则寄存器堆的数据写入端需要增加多一个端口写入 lui 的结果（lui 的结果于 ImmProcess 产生）

- ALU 第二个输入

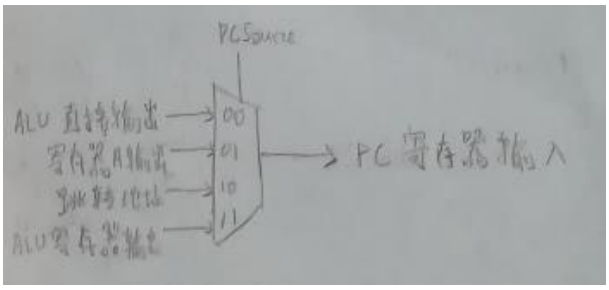
```
case (i_ALUSrcB_s)
2'b00    : mux_ALU_inputB<=reg_b;
2'b01    : mux_ALU_inputB<=32'd4;
2'b10    : mux_ALU_inputB<=Imm_out;
2'b11    : mux_ALU_inputB<=ImmShift_out;
endcase
```



功能：同样地，ALU 第二个输入也需要 MUX 进行选择。其中，寄存器 B 输出是地址为 Rt 的寄存器堆输出；常数 4 用于 PC+4；Imm_out 为 Immediate 的 32bits 扩展，可为 0 扩展，符号位扩展和 lui 的后 16bits 加 0（具体实现在 ImmProcess 中）；ImmShift_out 为 Immediate 的 32bits 扩展乘以 4，用于 beq。

- PC 寄存器输入

```
//mux for pc input
always@(*)
begin
case (i_PCSource_s)
2'b00 :mux_PC_input<=ALUOut;      //for PC+4
2'b01 :mux_PC_input<=reg_a;        // R[rs]
2'b10 :mux_PC_input<={PC_output[31:28],rs,rt,rd,shamt,funct,2'b00};
2'b11: mux_PC_input<=ALUOut_Reg;
endcase
end
```



功能：由于有一些跳转指令，因此 PC 寄存器输入有多种不同来源，因此需要 MUX 进行选择。其中，ALU 直接输出用于 PC+4；寄存器 A 输出用于 jr 和 jalr 两条指令；跳转地址用于 j 和 jal 两条指令；ALU 寄存器输出用于 beq 指令，因为 beq 的跳转地址需要用 ALU 计算。

寄存器

- PC 寄存器（PC），输出端口：PC_output

功能：用于储存指令的执行顺序。

- 寄存器 A &寄存器 B（RegTemp），输出端口：reg_a，reg_b

功能：用于储存寄存器堆读取的两个数据。

- MDR （RegTemp），输出端口：MDR_output

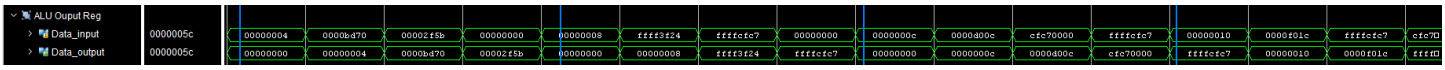
功能：用于储存储存器的输出，在 lw 会用到。

- ALU 寄存器（RegTemp），输出端口：ALUOut_Reg

功能：用于储存 ALU 的计算结果。

问题：为什么 ALU 需要寄存器？

答：见如下仿真图，



**Data_input 为 ALU 的直接计算结果，即 ALU 寄存器输入；Data_output 为 ALU 寄存器输出

注意到 ALU 寄存器输出比 ALU 直接输出慢一个周期，因此 ALU 运算和寄存器写入若发生在同一个周期（如 PC<=PC+4），需要使用 ALU 的直接计算结果；另一方面，若寄存器写入发生在 ALU 运算的下一个周期，为了保证写入结果的正确性，则需要使用 ALU 寄存器输出。

同理，MDR 是需要的因为 lw 的储存器数据读取（stage4 阶段）发生在寄存器堆写入（stage5 阶段）的上一个周期；作为 ALU 的输入，寄存器 A &寄存器 B 也是需要的因为寄存器堆读取（ID 阶段）发生在 ALU 运算（EX 阶段）的上一个周期。

2. 控制信号

2.1）

控制信号	0	1
PCWrite	不允许写 PC	允许写 PC
PCWriteCond	非 beq 指令	Beq 指令写入 PC
MemWrite	不可写入储存器	可写入储存器
MemRead	不可读取储存器	可读取储存器
IRWrite	不可写入指令寄存器	可写入指令寄存器
RegWrite	不可写入寄存器堆	可写入寄存器堆
ExtOp	对 Immediate 进行“0”扩展	对 Immediate 进行符号位扩展
LuiOp	非 lui 指令	Lui 指令处理 Immediate

Mux 控制信号	0	1
IorD	PC 作为储存器地址输入	ALU 结果（Reg）作为储存器地址输入
MemtoReg	MDR 作为寄存器堆数据输入	ALU 结果（Reg）作为寄存器堆数据输入

Mux 控制信号	00	01	10	11
RegDst	Rt 作为寄存器堆地址输入	Rd 作为寄存器堆地址输入	常数 31（ra）作为寄存器堆地址输入	常数 0 作为寄存器堆地址输入
ALUSrcA	暂时寄存器 A 作为 ALU 第一个输入	shamt 作为 ALU 第一个输入	PC 作为 ALU 第一个输入	常数 0 作为 ALU 第一个输入
ALUSrcB	暂时寄存器 B 作为 ALU 第二个输入	常数 4 作为 ALU 第二个输入	ImmExtOut 作为 ALU 第二个输入	ImmExtShift 作为 ALU 第二个输入
PCSource	ALU 结果作为 PC 寄存器输入	跳转地址作为 PC 寄存器输入	暂时寄存器 A 作为 PC 寄存器输入	ALU 结果（Reg）作为 PC 寄存器输入

2.2）定义指令类型：

- R 型指令- 1： add, addu, aub, subu, and, or, xor, nor, slt, sltu
- R 型指令- 2：sll, srl, sra
- I 型指令- 1： addi, addiu, sltiu, slti

*注：若未写明，ExtOp 默认为 1，其他所有控制信号默认为 0

定义状态：

状态 0：IF
PCWrite=1
MemRead=1
IRWrite=1
IorD=0
ALUSrcA=10
ALUSrcB=01
PCSource=00

状态 1-1：ID (所有除了 jal,jalr 指令)
ALUSrcA=10
ALUSrcB=11

状态 1-2：ID (jal 指令)
RegWrite=1
RegDst=10
MemtoReg=1

状态 1-3：ID (jalr 指令)
RegWrite=1
RegDst=01
MemtoReg=1

状态 2-1：EX (R-1 指令)
ALUSrcA=00
ALUSrcB=00

状态 2-2：EX (R-2 指令)
ALUSrcA=01
ALUSrcB=00

状态 2-3：EX (I-1, lw, sw 指令)
ALUSrcA=00
ALUSrcB=10

状态 2-4：EX (j,jal 指令)
PCWrite=1
PCSource=10

状态 2-5：EX (lui 指令)
ALUSrcA=11
ALUSrcB=10
LuiOp=1

状态 2-6：EX (jr, jalr 指令)
PCWrite=1
PCSource=11

状态 2-7：EX (andi 指令)
ALUSrcA=00
ALUSrcB=10
ExtOp=0

状态 2-8：EX (beq 指令)
PCWriteCondition=1
PCSource=01

状态 3-1：WB_and_Mem (I-1, andi. lui 指令)
RegWrite=1
RegDst=00
MemtoReg=1

状态 3-2：WB_and_Mem (R-1,R-2 指令)
RegWrite=1
RegDst=01
MemtoReg=1

状态 3-3：WB_and_Mem (lw 指令)
IorD=1
MemRead=1

状态 3-4：WB_and_Mem (sw 指令)
IorD=1
MemWrite=1

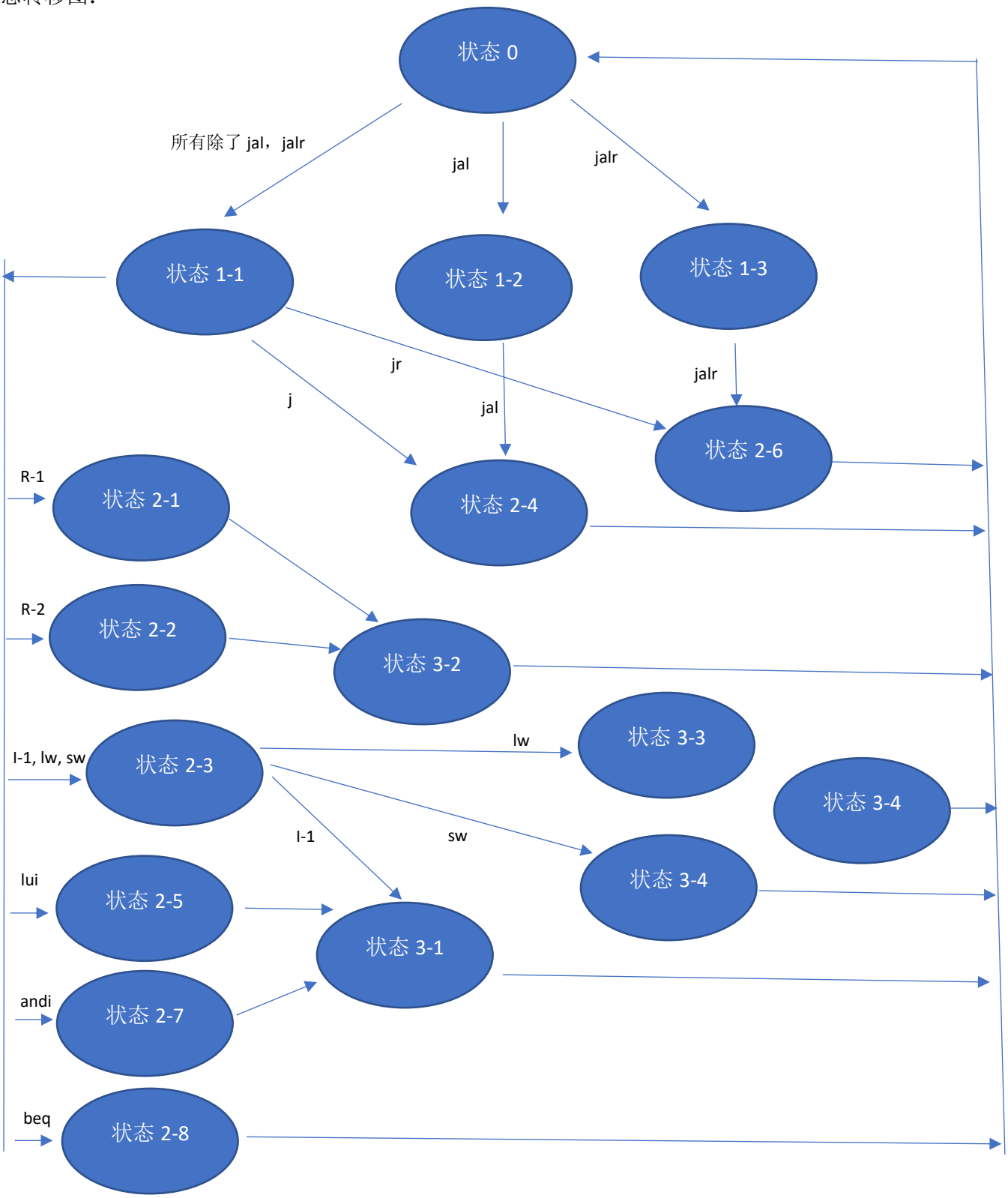
状态 4：Mem_lw (lw 指令)

RegWrite=1

RegDst=00

MemtoReg=1

状态转移图：



3. ALU 功能扩展

3.1）setsub - R 型指令

Instruction	OpCode[5:0]	Rs[4:0]	Rt[4:0]	Rd[4:0]	Shamt[4:0]	Funct[5:0]
setsub	0	rs	rt	rd	0	0x2f

3.2）集合的差 A-B 定义为 A 中不包含在 B 内的元素，其真值真值表如下：

S_{iA}（A 集合第 i 个元素）	S_{iB}（B 集合第 i 个元素）	$S_{i(A-B)}$（A-B 集合第 i 个元素）
0	0	0
0	1	0
1	0	1
1	1	0

所以 $S_{A-B} = S_A\overline{S_B}$

3.3）

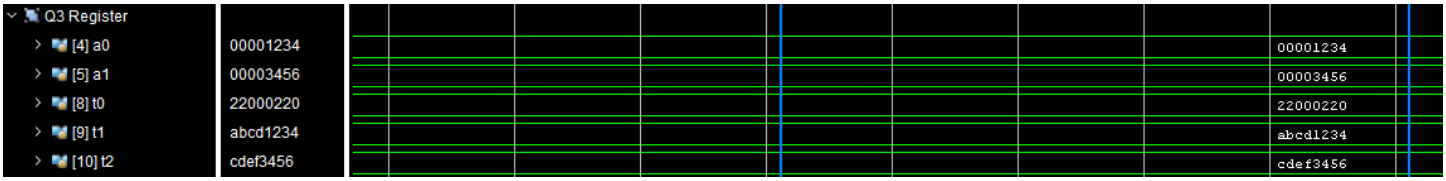
假设 A: 0xABCD1234， B: 0xCDEF3456

指令序列

0	lui $t1, 0xABCD
1	lui $t2, 0xCDEF
2	addi $a0, 0x1234
3	addi $a1, 0x3456
4	add $t1, $t1, $a0
5	add $t2, $t2, $a1
6	setsub $t0, $t1, $t2
7	Loop: j Loop

```
RAM_data[8'd0] <= {6'h0f, 5'd0 , 5'd9 , 16'habcd};          //lui
RAM_data[8'd1] <= {6'h0f, 5'd0 , 5'd10 , 16'hcdef};         //lui
RAM_data[8'd2] <= {6'h08, 5'd0 , 5'd4 , 16'h1234};          //addi
RAM_data[8'd3] <= {6'h08, 5'd0 , 5'd5 , 16'h3456};          //addi
RAM_data[8'd4] <= {6'h00, 5'd9, 5'd4 , 5'd9 , 5'd0, 6'h20};   //add
RAM_data[8'd5] <= {6'h00, 5'd10, 5'd5 , 5'd10 , 5'd0, 6'h20}; //add
RAM_data[8'd6] <= {6'h00, 5'd9, 5'd10 , 5'd8, 5'd0, 6'h2f};   //setsub
RAM_data[8'd7] <= {6'h02, 26'd7};                           //loop
```

Instruction	OpCode[5:0]	Rs[4:0]	Rt[4:0]	Rd[4:0]	Shamt[4:0]	Funct[5:0]
setsub	0	0x09	0x0a	0x08	0	0x2f



最后结果：0x2200_0220

4. 汇编程序分析-1

MIPS Assembly	
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

4.1）计算过程 & 最终结果：

0. $a0 = 12123

1. $a1 = -12345

2. $a2 = -12345×2^{16}

   $= -809041920$

3. $a3 = -12345$ (sra 是 sign extension)

4. $a3 == $a1

6. $t0 = -809041920 + 12123

   $= -809029797$

7. $t1 = \left\lfloor \frac{-809029797}{2^8} \right\rfloor$

   $= -3160273$

8. $t2 = -12123$

9. $a0 > $t2 , $v0 = 0

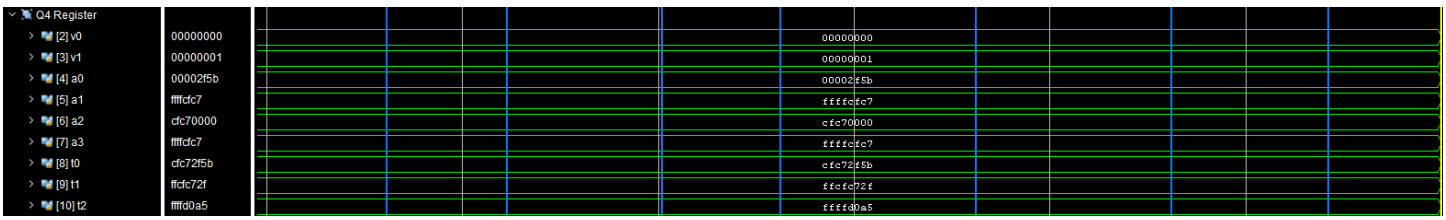
10. 如果 unsigned , -12123 为 53413

    $\therefore$ $a0 < $t2 , $v1 = 1

最终值：

$a0 = 12123	0x00002f5b	$t0 = -809029797	0xcfc72f5b	$v0 = 0	
$a1 = -12345	0xffffcfc7	$t1 = -3160273	0xffcfc72f	$v1 = 1	
$a2 = -809041920	0xcfc70000	$t2 = -12123	0xffffd0a5		
$a3 = -12345	0xffffcfc7				

4.2）仿真截图



5. 汇编程序分析-2

5.1）

MIPS Assembly	
0	addi $a0, $zero, 5
1	xor $v0, $zero, $zero
2	jal sum
	Loop:
3	beq $zero, $zero, Loop
	sum:
4	addi $sp, $sp, -8
5	sw $ra, 4($sp)
6	sw $a0, 0($sp)
7	slti $t0, $a0, 1
8	beq $t0, $zero, L1
9	addi $sp, $sp, 8
10	jr $ra
	L1:
11	add $v0, $a0, $v0
12	addi $a0, $a0, -1
13	jal sum
14	lw $a0, 0($sp)
15	lw $ra, 4($sp)
16	addi $sp, $sp, 8
17	add $v0, $a0, $v0
18	jr $ra

0：令 $a0=5

1：令 $v0=0

2. 跳转至 sum，并将返回地址存入 $ra

4：将 stack pointer 向下移动两位

5. 将返回地址存入 stack

6. 将 $a0 存入 stack

7. 比较 $a0 与 1，若 $a0<1，$t0=1

8. 判断 $t0 是否为 0，若是，跳转至 L1

9. 恢复 stak

10. 跳转至返回地址

11. $v0 = $a0 + $v0 （第1次累加）

12. 将 $a0 减去 1

13. 跳转至 sum，并将返回地址存在 $ra

14. 从 stack 读取 $a0

15. 从 stack 读取 $ra

16. 恢复 stack

17. $v0 = $a0 + $v0 （第2次累加）

18. 跳转至返回地址

程序功能：这段程序能实现进行两次1累加至 n 的功能，即输出结果为 n(n+1)

Loop 功能：执行3-次 sum 后，通过程序再次执行 sum，使得结果出错

Sum：将必要的数据存入 stack，以在递归结束后使用，此外也用以判断 $a0 是否已经减少至0

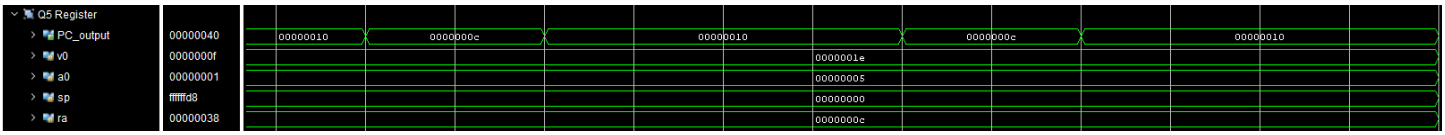
L1：进行两次的累加

5.2）

Instruction	OpCode[5:0]	Rs[4:0]	Rt[4:0]	Rd[4:0]	Shamt[4:0]	Funct[5:0]
Addi	0x08	0	4	0x05		
Xor	0	0	0	2	0	0x26
jal	0x03	0x0004				
beq	0x04	0	0	0xffff		
addi	0x08	29	29	-8 (0xfff8)		
sw	0x2b	29	31	4		
sw	0x2b	29	4	0x0004		
slti	0x0a	4	8	0x01		
beq	0x04	8	0	0x0002		
addi	0x08	29	29	0x0008		
jr	0	31	0			0x08
add	0	2	4	2	0	0x20
addi	0x08	4	4	0xffff		
jal	0x03	0x0004				
lw	0x23	29	4	0		
lw	0x23	29	31	0x0004		
Addi	0x08	29	29	0x0008		
add	0	4	2	2	0	0x20
jr	0	31	0			0x08

```
RAM_data[8'd0] <= {6'h08, 5'd0 , 5'd4 , 16'h0005};    //addi
RAM_data[8'd1] <= {6'h00, 5'd0 , 5'd0 , 5'd2 , 5'd0, 6'h26};   //xor
RAM_data[8'd2] <= {6'h03, 26'd4};    //jal
RAM_data[8'd3] <= {6'h04, 5'd0 , 5'd0 , 16'hffff};    //beq
RAM_data[8'd4] <= {6'h08, 5'd29, 5'd29 , 16'hfff8}; //addi
RAM_data[8'd5] <= {6'h2b, 5'd29, 5'd31 , 16'h0004}; //sw
RAM_data[8'd6] <= {6'h2b, 5'd29, 5'd4 , 16'h0000}; //sw

RAM_data[8'd7] <= {6'h0a, 5'd4, 5'd8 , 16'h0001}; //slti
RAM_data[8'd8] <= {6'h04, 5'd8, 5'd0 , 16'h0002}; //beq
RAM_data[8'd9] <= {6'h08, 5'd29, 5'd29 , 16'h0008}; //addi
RAM_data[8'd10] <= {6'h00, 5'd31, 15'd0 , 6'h08}; //jr
RAM_data[8'd11] <= {6'h00, 5'd2, 5'd4, 5'd2, 5'd0, 6'h20}; //add
RAM_data[8'd12] <= {6'h08, 5'd4, 5'd4 , 16'hffff}; //addi
RAM_data[8'd13] <= {6'h03, 26'd4}; //jal
RAM_data[8'd14] <= {6'h23, 5'd29, 5'd4 , 16'h0000}; //lw
RAM_data[8'd15] <= {6'h23, 5'd29, 5'd31 , 16'h0004}; //lw
RAM_data[8'd16] <= {6'h08, 5'd29, 5'd29 , 16'h0008}; //addi
RAM_data[8'd17] <= {6'h00, 5'd4, 5'd2 , 5'd2, 5'd0, 6'h20}; //add
RAM_data[8'd18] <= {6'h00, 5'd31, 15'd0 , 6'h08}; //jr
```

5.3）仿真结果如下图：



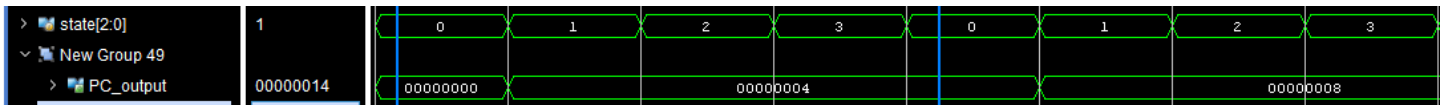
$a0=5，$v0=30（0x0000_001e)，与我预期的结果相符合

5.4）

PC：

- 代码中在 IF 阶段（状态 0）会完成 PC<=PC+4，因此在状态 0 转移至状态 1，PC 值会+4（所有指令都会执行这步）。

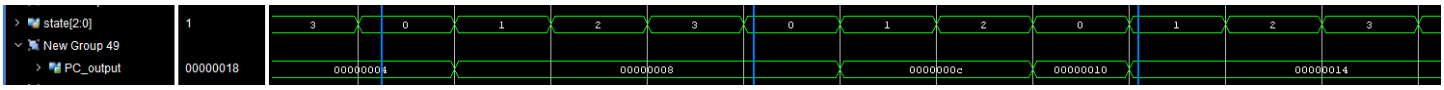
验证 IF 阶段的 PC+4



观察上方仿真图（PC 0x0000_0000 和 PC 0x0000_0004 分别代表程序首两个指令 addi 和 xor），显然能够验证 PC<=PC+4 有正常运行。

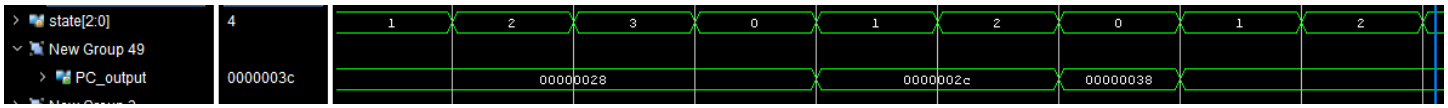
- 另外，若涉及 j，jal，jr，jalr 指令，代码中在 EX（状态 2）阶段完成 PC 的更新，因此在状态 2 转移至状态 0，PC 值会更新为跳转地址。

验证 jal 更新



观察上方仿真图（PC 0x0000_0008 和 PC 0x0000_000c 分别代表程序中的指令 2 和 3：jal 和 beq），可以看到 jal 指令在执行时，先在 IF 阶段完成 PC<=PC+4，之后再 EX 阶段完成跳转地址的更新（PC 更新为：0x0000_0010，对应指令 4，sum 中的 addi），jal 指令有正常执行。

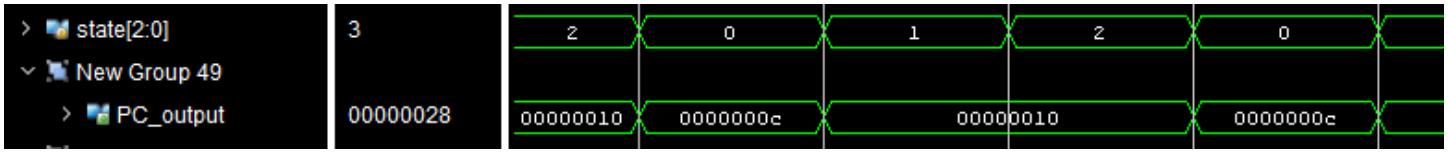
验证 jr 更新



观察上方仿真图（PC 0x0000_0028，PC 0x0000_002c 和 PC 0x0000_0038 分别代表程序中的指令 10，11 和 14：jr，add，lw），同样可以看到 jr 指令在执行时，先在 IF 阶段完成 PC<=PC+4，之后再 EX 阶段完成跳转地址的更新（PC 更新为：0x0000_0038，对应指令 14，L1 中的第一个 lw），因此根据程序的思路，jr 指令有正常执行。

- 再者，若涉及 beq 指令，则代码中在 ID 阶段（状态 1）完成跳转地址计算，并在 EX（状态 2）阶段完成判断和 PC 更新（如果条件成立），因此在状态 2 转移至状态 0，PC 值会更新为跳转地址。

验证 beq 更新



观察上方仿真图（PC 0x0000_000c 和 PC 0x0000_0010 分别代表程序中的指令 3 和 4：beq，addi），可以看到 beq 指令在执行时，先在 IF 阶段完成 PC<=PC+4，之后再 EX 阶段完成跳转地址的更新（若判断成功，PC 更新为：0x0000_000c，对应指令 3，beq，这里是个无限循环），根据图中结果得以验证 beq 指令有正常执行。

总结：

PC 值会先进行 PC<=PC+4，之后若遇到需要进行地址跳转的指令（在程序中为 jal，jr 和 beq）就会进行相应的跳转（上述已详细解释）。根据程序思路，程序会先运行指令 0，1，2，接着跳转入 sum，若$a0 不为 0，则跳转至 L1，然后在 L1 中又跳转回 sum，周而复始直到$a0 为 0，接着在 L1 跳转回相应的返回地址以完成累加操作。

$a0：

变化：n，n-1，n-2 …, 0, 1, …, n

根据仿真，$a0 会逐渐减小至 0，接着再逐渐增加至其原本的数，与程序思路相符。这是因为$a0 每次循环的值会先储存在栈中（指令 6，sw），然后指令 12 的 addi 会将$a0 减去 1，$a0 变成 0 后，指令 14 会将栈中的$a0 取出并写入 a0（根据栈先入后出的特性，最后出的就是$a0 原本的数值）

$v0：

变化：n, n+n-1, n+n-1+n-2, …, n(n+1)/2, n(n+1)/2+1, … n(n+1)

根据仿真，$v0 会呈现上述变化，与程序思路相符。这是因为指令 11 中的 add 会在$a0 从 n 逐渐变小至 0 的过程中进行累加工作；另外，指令 17 的 add 会在$a0 从 0 增加回 n 的过程中进行累加工作。

$sp：

变化 1：‘-8’，如 0x0000_0000 -> 0xffff_fff8

变化 2：‘+8’，如 0xffff_fff0 -> 0xffff_fff8

根据仿真，$sp 的变化为+8 或-8，与程序思路相符。在 sum 中的指令 4，会进行-8 操作，以腾出栈空间储存$ra 和$a0 两个数据（栈指针是往低地址方向“增加”）。另外，在指令 9 和指令 17 会恢复栈（读出已储存且需要的数据后），这时就会进行+8 操作。

$ra：

变化：0x0000_0000 -> 0x0000_000c ->0x0000_0038 …. ->0x0000_0038 -> 0x0000_000c

根据仿真，此程序$ra 的值只会有 3 种可能性，分别为 0x0000_0000，0x0000_000c 和 0x0000_0038，对应初始地址，指令 2 的返回地址（即指令 3，beq）和指令 13 的返回地址（即指令 14，lw），与程序思路相符。首先程序完成指令 2 后会将下一条指令（0x0000_000c），储存在 ra。之后进行递归时，每次跳转所储存的$ra 会被压入栈中（指令 5 的 sw），因为之后的程序中，可能会用到 ra（指令 13）。递归执行过程中，程序会逐步将栈中的 ra 取出，并返回相应地址（0x0000_0038），最后当程序要结束时，最先压入栈的 0x0000_000c 会被取出，程序陷入一个无限循环（代表执行结束）。