C P U 大 作 业

漆耘含

无 63

2016011058

一、 阅读处理器的结构图，理解其中控制信号的含义

1. 试回答以下问题

a. 由 RegDst 信号控制的多路选择器，输入 2 对应常数 31。这里的 31 代表什么？在执行哪些指令时需要 RegDst 信号为 2？为什么？

答：RegDst 控制的是决定是哪一个寄存器被写入，31 对应的是$ra，因为$ra 的寄存器编号是 31。在执行 jal，jalr 指令的时候，因为需要保存当前指令地址，所以需要把值保存在$ra 中，故 RegDst 赋值为 2。

b. 由 ALUSrc1 信号控制的多路选择器，输入 1 对应的指令[10-6]是什么？在执行哪些指令时需要 ALUSrc1 信号为 1，为什么？

答：根据指令格式可以看出，指令[10-6]代表的是 R 型指令中的偏移量（shamt[4:0]）。sll，srl，sra 移位指令需要偏移量的输入，因此 ALUSrc1 应该为 1，让偏移量输出。

c. 由 MemtoReg 信号控制的多路选择器，输入 2 对应的是什么？在执行哪些指令时需要 MemtoReg 信号为 2？为什么？

答：MemtoReg 输入为 2 的时候，输出是 PC+4（即下一条指令的地址），然后把这条地址写回寄存器堆中。这种操作应该是 jal，jalr 的时候，需要把地址保存在$ra 中，所以 MemtoReg 输入为 2。

d. 图中的处理器结构并没有 Jump 控制信号，取而代之的是 PCSrc 信号。PCSrc 信号控制的多路选择器，输入 2 对应的是什么？在执行哪些指令时需要 PCSrc 信号为 2？为什么？

答：PCSrc 信号为 2 的时候，输入的是从寄存器堆 1 对应的输出，即地址是从寄存器中读取的，应该是在执行 jr 或者 jalr。因为 jr 和 jalr 是跳转指令，但跳转的地址需要从寄存器中读出。

e. **为什么需要 ExtOp 控制信号？什么情况下 ExtOp 信号为 1？什么情况下 ExtOp 信号为 0？**

答：ExtOp 决定立即数的扩展方式是符号扩展还是无符号扩展。ExtOp 为 1 的时候，是对立即数做符号扩展，比如指令 lw，sw，addi，addiu，slti，beq 等；ExtOp 为 0 的时候，需要对输入立即数做零扩展的指令，如 andi，sltiu。

f. **若想再多实现一条指令 nop（空指令），指令格式全为 0，需要如何修改处理器结构？**

答：需要设置 Branch 和 PCSrc 都为 0，保证顺序指令，同时不能让寄存器和存储器中的值改变，所以 RegWrite 和 MenWrite 都设置为 0.

2. **根据对各控制信号功能的理解，填写如下真值表。（0,1,2，x）**

	PcSrc[1:0]	Branch	RegWrite	RegDst[1:0]	MemRead	MemWrite	MemtoReg[1:0]	ALUSrc1	ALUSrc2	ExtOp	LuOp
lw	0	0	1	0	1	0	1	0	1	1	0
sw	0	0	0	X	0	1	X	0	1	1	0
lui	0	0	1	0	0	0	0	0	1	X	1
add	0	0	1	1	0	0	0	0	0	X	X
addu	0	0	1	1	0	0	0	0	0	X	X
sub	0	0	1	1	0	0	0	0	0	X	X

subu	0	0	1	1	0	0	0	0	0	X	X
addi	0	0	1	0	0	0	0	0	1	1	0
addiu	0	0	1	0	0	0	0	0	1	1	0
and	0	0	1	1	0	0	0	0	0	X	X
or	0	0	1	1	0	0	0	0	0	X	X
xor	0	0	1	1	0	0	0	0	0	X	X
nor	0	0	1	1	0	0	0	0	0	X	X
andi	0	0	1	0	0	0	0	0	1	0	0
sll	0	0	1	1	0	0	0	1	0	X	X
srl	0	0	1	1	0	0	0	1	0	X	X
sra	0	0	1	1	0	0	0	1	0	X	X
slt	0	0	1	1	0	0	0	0	0	X	X
sltu	0	0	1	1	0	0	0	0	0	X	X
slti	0	0	1	0	0	0	0	0	1	1	0
sltiu	0	0	1	0	0	0	0	0	1	1	0
beq	0	1	0	X	0	0	X	0	0	1	0
j	1	X	0	X	0	0	X	X	X	X	X
jal	1	X	1	2	0	0	2	X	X	X	X
jr	2	X	0	X	0	0	X	X	X	X	X
jalr	2	X	1	1	0	0	2	X	X	X	X

二、 完成控制器

1. CPU.v 实现了处理器的整体结构。阅读 CPU.v，理解其实现方式
2. Control.v 是控制器模块的代码。完成 Control.v。

代码（control.v）：

```
module Control(OpCode, Funct,
    PCSrc, Branch, RegWrite, RegDst,
    MemRead, MemWrite, MemtoReg,
    ALUSrc1, ALUSrc2, ExtOp, LuOp, ALUOp);
    input [5:0] OpCode;
    input [5:0] Funct;
    output [1:0] PCSrc;
    output Branch;
    output RegWrite;
    output [1:0] RegDst;
    output MemRead;
    output MemWrite;
    output [1:0] MemtoReg;
    output ALUSrc1;
    output ALUSrc2;
    output ExtOp;
    output LuOp;
    output [3:0] ALUOp;

    // Your code below
    assign PCSrc[1:0]=(OpCode==6'h02||OpCode==6'h03)?2'b01:

(OpCode==0&&(Funct==6'h08||Funct==6'h09))?2'b10:2'b00;
```

```
assign Branch=(OpCode==6'h04)?1'b1:1'b0;

assign
RegWrite=(OpCode==6'h2b||OpCode==6'h04||OpCode==6'h02||(OpCode
==0&&(Funct==6'h08)))?1'b0:1'b1;

assign RegDst[1:0]=(OpCode==6'h03)?2'b10:

(OpCode==6'h23||OpCode==6'h0f||OpCode==6'h08||OpCode==6'h09||O
pCode==6'h0c||OpCode==6'h0a||OpCode==6'h0b)?2'b00:2'b01;

assign MemRead=(OpCode==6'h23)?1'b1:1'b0;

assign MemWrite=(OpCode==6'h2b)?1'b1:1'b0;

assign
MemtoReg[1:0]=(OpCode==6'h03||(OpCode==0&&Funct==6'h09))?2'b10
:(OpCode==6'h23)?2'b01:2'b00;

assign
ALUSrc1=(OpCode==0&&(Funct==0||Funct==6'h03||Funct==6'h02))?1'
b1:1'b0;

assign
ALUSrc2=(OpCode==6'h23||OpCode==6'h2b||OpCode==6'h0f||OpCode==
6'h08||OpCode==6'h09||OpCode==6'h0c||OpCode==6'h0a||OpCode==6'
h0b)?1'b1:1'b0;

assign
```

```
ExtOp=(OpCode==6'h23||OpCode==6'h2b||OpCode==6'h08||OpCode==6'
h09||OpCode==6'h0a||OpCode==6'h0b||OpCode==6'h04)?1'b1:1'b0;

        assign LuOp=(OpCode==6'h0f)?1'b1:1'b0;



        // Your code above

        assign ALUOp[2:0] =
            (OpCode == 6'h00)? 3'b010:
            (OpCode == 6'h04)? 3'b001:
            (OpCode == 6'h0c)? 3'b100:
            (OpCode == 6'h0a || OpCode == 6'h0b)? 3'b101:
            3'b000;

        assign ALUOp[3] = OpCode[0];

endmodule
```

3. 阅读 InstructionMemory.v，根据注释理解指令存储器中的程序

根据指令存储器中的程序：

```
MIPS Assembly
 0          addi $a0, $zero, 12345
 1          addiu $a1, $zero, -11215
 2          sll $a2, $a1, 16
 3          sra $a3, $a2, 16
 4          beq $a3, $a1, L1
 5          lui $a0, -11111
    L1:
 6          add $t0, $a2, $a0
 7          sra $t1, $t0, 8
 8          addi $t2, $zero, -12345
 9          slt $v0, $a0, $t2
10          sltu $v1, $a0, $t2
    Loop:
11          j Loop
```

a. 这段程序执行足够长时间后会发生什么？

长时间执行之后，程序会在第 11 行陷入死循环。

b. 此时，寄存器$a0-$a3，$t0-$t2，$v0-$v1 中的值应是多少？

$a0	0x00003039
$a1	0xffffd431
$a2	0xd4310000
$a3	0xffffd431
$t0	0xd4313039
$t1	0xffd43130
$t2	0xffffcfc7
$v0	0x00000000
$v1	0x00000001

计算过程：

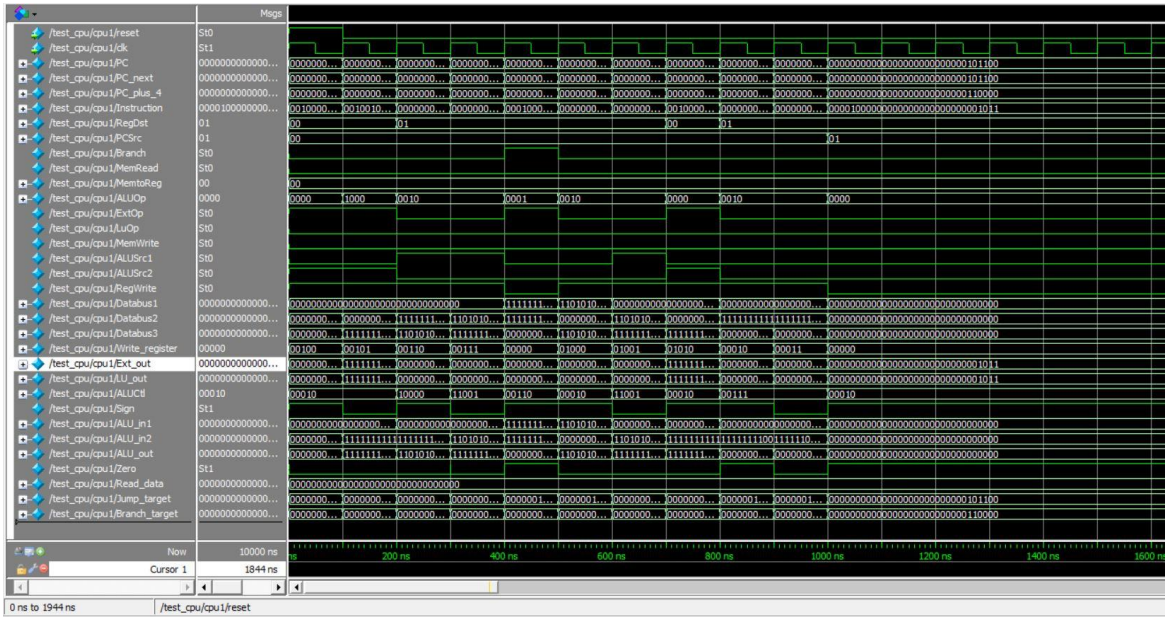
把 12345 和-11215 分别加载到$a0 和$a1，$a1 左移 16 位并储存在$a2 中，$a2 右移 16 位（有符号）并储存在$a3 中，如果$a3=$a1，则跳转 L1。在$t0 中储存$a2 和$a0 的和，$t0 右移 8 位并存储在$t1 中，在$t2 中-12345，如果$a0<$t2（有符号比较），则$v0=1，如果$a0<$t2（无符号比较），则$v1=1。之后陷入死循环。

c. 如果已知某一时刻寄存器中存放数 0xffffcfc7，能否判断出它是有符号数还是无符号数？为什么？

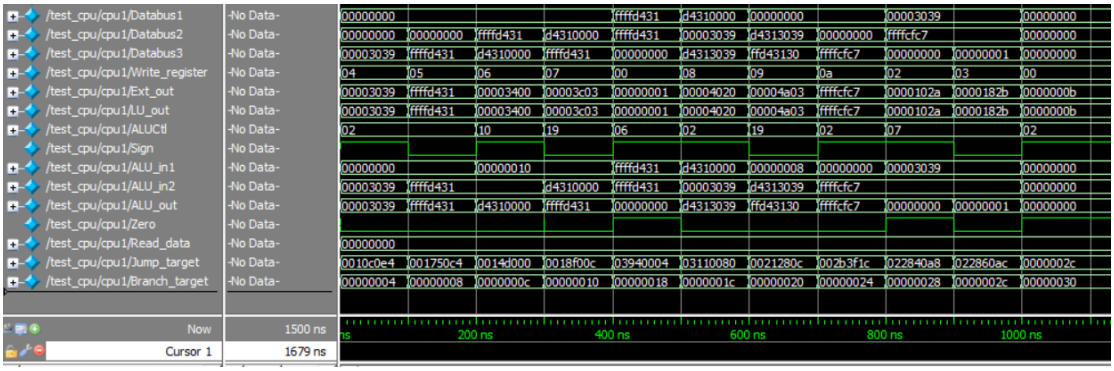
无法判断，因为由程序结果来看，不同的指令运算结果可能相同，所以不能判断是否带有符号

4. 使用 ModelSim 等仿真软件进行仿真。仿真顶层模块为 test_cpu，这是一个 testbench，用于向 cpu 提供复位和时钟。观察仿真结果中各寄存器和控制信号的变化。回答以下问题：

仿真（整体）：



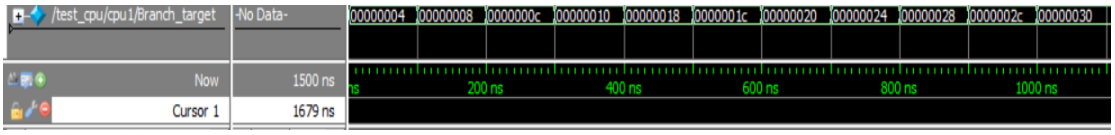
对输出结果进行仿真：



下面由仿真结果来进行回答问题

a. **Pc 如何变化**

Pc 正常的+4，但是在 Branch=1 的时候，pc 由 0x10 变到了 0x18，对应的是 beq 语句



b. **Branch 信号在何时为 1？它引起了 PC 怎样的变化？**

Branch 在第 400ns-500ns 的时候为 1，使 pc 由 0x10 变到了 0x18。

c. **100-200ns 之间，PC 是多少？对应的指令是哪一条？此时$a1 的值是多少？200-300ns 器件$a1 是多少，为什么？下一条指令立即使用到了$a1 的值，会出现错误吗？为什么？**

时间	$a1	PC
100-200ns	0x00000000	0x00000004
200-300ns	0xffffd431	

在 100-200ns 的时候，PC：0x00000004 对应的是 addiu 指令。

在 200-300ns 的时候，$a1=0xffffd431 是因为已经执行完 addiu 指令，因此值会改变。

因为在 addiu 指令中已经完成对$a1 的写入，因此下一条指令是可以用$a1 的，这和流水线是有区别的。

d. 运行足够长时间后(如 1100ns)，各寄存器不再变化，寄存器$a0-$a3，$t0-$t2，$v0-$v1 中的值是多少，与你预期是否一致？

上面已经分析过了，根据输出波形来看也是吻合的：

$a0	0x00003039
$a1	0xffffd431
$a2	0xd4310000
$a3	0xffffd431
$t0	0xd4313039
$t1	0xffd43130
$t2	0xffffcfc7
$v0	0x00000000
$v1	0x00000001

三、 执行汇编程序

阅读并理解下面这段汇编程序：

MIPS Assembly

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

1. **如果第一行的 3 是任意正整数 n，这段程序能实现什么功能？Loop，sum，L1 各有什么作用？为每一句代码添加注释。**

   a. 如果是任意正整数 n，则可实现的功能是：$\sum_{k=1}^{n} k$，即求和功能。Loop 的功能室在程序返回之后，进入死循环，即锁死；sum 的作用是在每次循环之前进行压栈、判断；L1 的作用是进行相加的功能（在每次函数返回之后）。

b. 添加了注释的代码：

```
1         addi $a0, $zero, 3              #给a0赋值为3
2         jal sum                         #跳到sum
3   Loop:
4         beq $zero, $zero, Loop          #在函数返回之后，进入死循环，锁死
5   sum:
6         addi $sp, $sp, -8               #$sp是栈地址，-8相当于压栈
7         sw $ra, 4($sp)                  #ra先入栈
8         sw $a0, 0($sp)                  #a0后入栈
9         slti $t0, $a0, 1                #如果a0<1,则t0=1
10        beq $t0, $zero, L1              #如果t0=0，则跳转到L1
11        xor $v0, $zero, $zero           #使v0=1
12        addi $sp, $sp, 8                #出栈操作
13        jr $ra                          #返回上一级
14  L1:
15        addi $a0, $a0, -1               #a0减一
16        jal sum                         #调用sum
17        lw $a0, 0($sp)                  #出栈并赋值给a0
18        lw $ra, 4($sp)                  #出栈并赋值给ra
19        addi $sp, $sp, 8                #出栈
20        add $v0,$a0,$v0                 #累加
21        jr $ra                          #返回上一级
```

2. 将这段代码翻译成机器码

0x20040003
0x0c000003
0x1000ffff
0x1000ffff
0x23bdfff8
0xafbf0004
0x28880001
0x11000003
0x00001026

0x23bd0008
0x2084ffff
0x0c000003
0x8fa40000
0x8fbf0004
0x23bd0008
0x00821020
0x03e00008

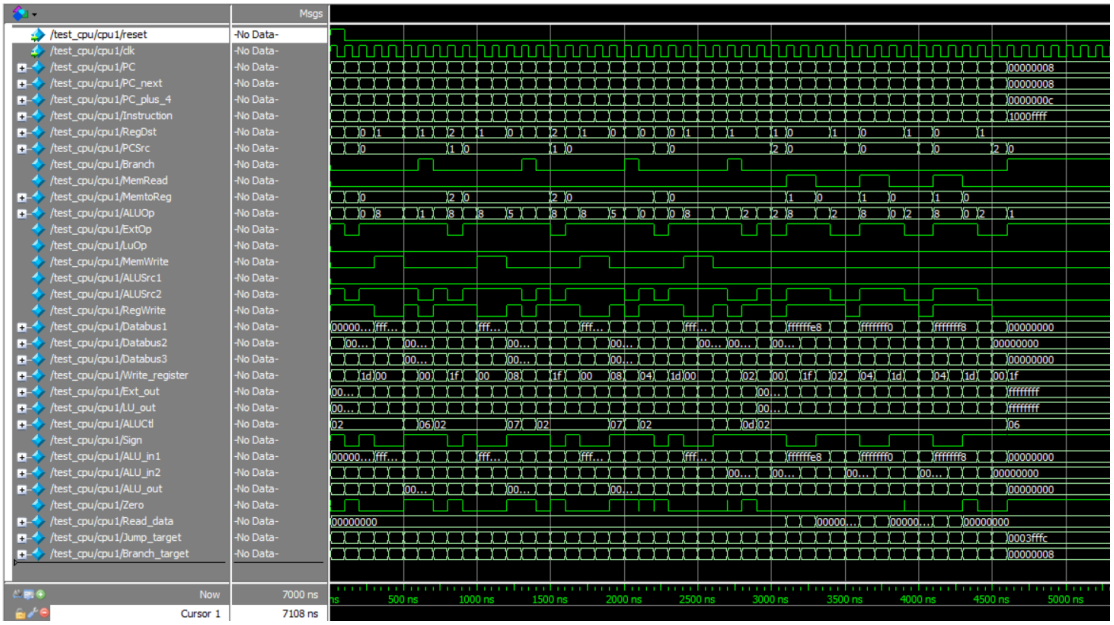
a. 对于 beq 和 jal 中的 Loop，sum，L1,其中 Loop 和 L1 翻译为当前位子和目标位子的距离，sum 翻译为 26 位的二进制数。

b. 立即数-1：16'hffff（二补码）

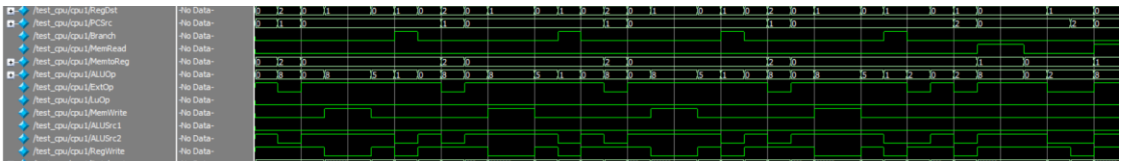
立即数-8：1111111111111000（二补码）

3. **修改 InstructionMemory.v，使 CPU 运行上面的这段程序。注意 case 语句的输入是地址的[9-2]比特。仿真观察各控制信号和寄存器的变化。**

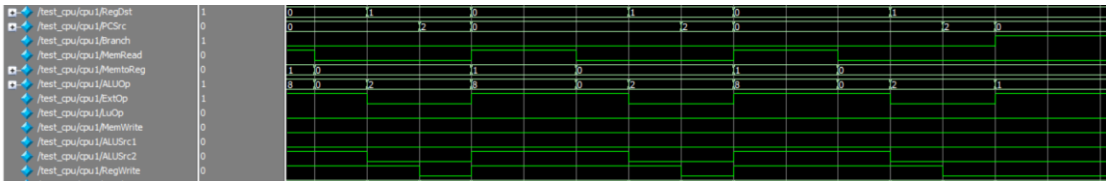
仿真（0-5000ns）：



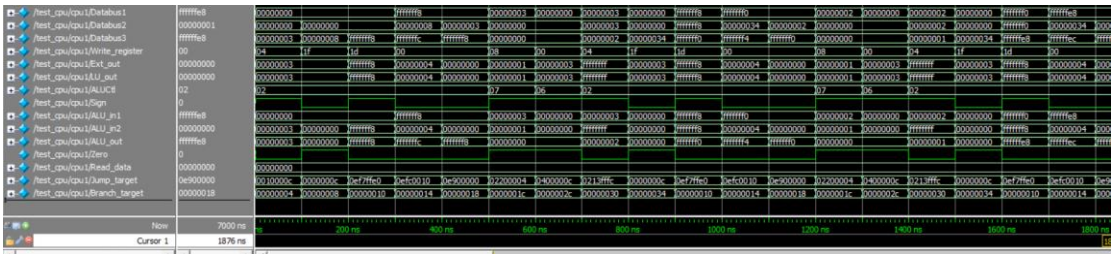
控制信号（0-3500ns）：



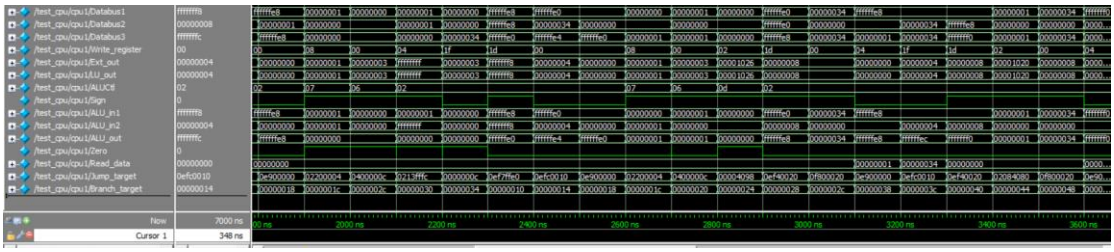
控制信号（3500-5000ns）：



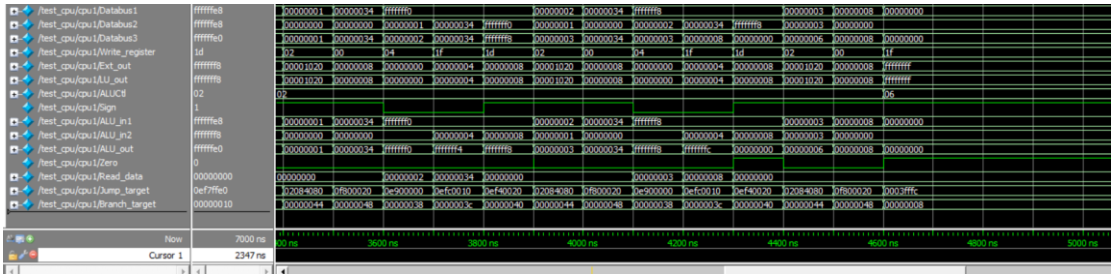
寄存器（0-1800ns）：



寄存器（1800-3600ns）：



寄存器（3600-5000ns）：



a. 运行时间足够长，寄存器$a0，$v0 的值是多少？

$a0	$v0
3	6

因为 6=3+2+1

b. 观察、描述并解释 PC，$a0，$v0，$sp，$ra 如何变化

PC：

0x00
0x04
0x0c（jal sum）
…
0x1c
0x2c（beq）
0x30
0x0c（jal sum）
…（重复直到 a0=0）
0x1c
0x28
0x34（jr $ra）
…（逐步+4）
0x44
0x34（jr $ra）
…（重复操作）
0x08
…（永远停留在 0x08）

$a0：a0 先从 3 减小到 0，再依次返回增加到 3

3
2
1
0
1
2
3

$v0：在返回过程中逐步增加

1
3
6

$sp：每次压栈，地址-8（四次），之后反悔的时候出栈，地址+8（四次）

$ra：最开始是 0x00，之后为 0x08，0x34，交替变换，最后变化为 0x08.

附修改后的 InstructionMemory.v：

```
module InstructionMemory(Address, Instruction);
    input [31:0] Address;
    output reg [31:0] Instruction;

    always @(*)
        case (Address[9:2])
            // addi $a0, $zero, 3
            8'd0:    Instruction <= {6'h08, 5'd0 , 5'd4 , 16'h03};
            // jal sum
            8'd1:    Instruction <= {6'h03, 26'h03};
            //Loop:
            // beq $zero, $zero, Loop
            8'd2:    Instruction <= {6'h04, 5'd0 , 5'd0 ,
16'b1111111111111111};
            //sum:
            // addi $sp, $sp, -8
            8'd3:    Instruction <= {6'h08, 5'd29 ,
5'd29 ,16'b1111111111111000};
            // sw $ra, 4($sp)
            8'd4:    Instruction <= {6'h2b, 5'd29 , 5'd31 , 16'h04};
            // sw $a0, 0($sp)
            8'd5:    Instruction <= {6'h2b, 5'd29 , 5'd4  , 16'h0};
            // slti $t0, $a0, 1
            8'd6:    Instruction <= {6'h0a, 5'd4 , 5'd8 , 16'h01};
            // beq $t0, $zero, L1
            8'd7:    Instruction <= {6'h04, 5'd8 , 5'd0 ,
16'b0000000000000011};
```

```
            // xor $v0, $zero, $zero
            8'd8:    Instruction <= {6'h00, 5'd0 , 5'd0, 5'd2, 5'd0,
6'h26};
            // addi $sp, $sp, 8
            8'd9:    Instruction <= {6'h08, 5'd29 , 5'd29 , 16'h0008};
            // jr $ra
            8'd10:   Instruction <= {6'h00, 5'd31 , 15'h0 , 6'h08};
            // L1:
            // addi $a0, $a0, -1
            8'd11:   Instruction <= {6'h08, 5'd4, 5'd4, 16'hffff};
            //jal sum
            8'd12:   Instruction <= {6'h03, 26'h3};
            //lw $a0, 0($sp)
            8'd13:   Instruction <= {6'h23, 5'd29, 5'd4, 16'h0};
            //lw $ra, 4($sp)
            8'd14:   Instruction <= {6'h23, 5'd29, 5'd31, 16'h04};
            //addi $sp, $sp, 8
            8'd15:   Instruction <= {6'h08, 5'd29, 5'd29, 16'h08};
            //add $v0,$a0,$v0
            8'd16:   Instruction <= {6'h0, 5'd4, 5'd2,
5'd2,5'h0,6'h20};
            //jr $ra
            8'd17:   Instruction <= {6'h0, 5'd31 , 15'h0 , 6'h08};

            default: Instruction <= 32'h00000000;
        endcase

endmodule
```