单周期处理器

《数字逻辑与处理器》

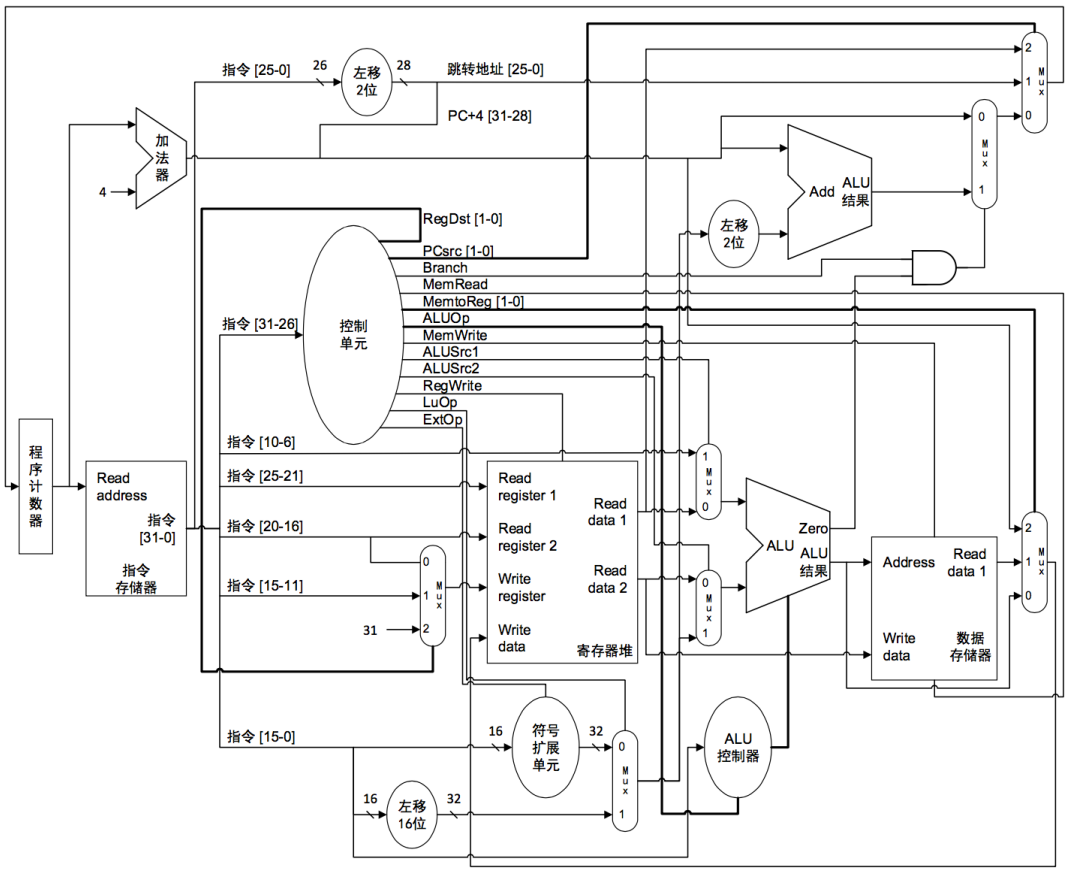
学号：2013011283

班级：无 31

姓名：赵埔塬

日期：2015 年 6 月 18 日

一、 处理器结构



1.阅读处理器结构后对以下问题作出如下解答

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 代表寄存器$ra 的寄存器号，在结构中利用其保存返回地址。为使 RegDst 的信号为 2，需要执行指令 jal,原因在于执行 jal 指令时需要将 PC+4 的值存到 31 号寄存器$ra。

b) 由 ALUSrc1 信号控制的多路选择器，输入 1 对应的指令[10-6]
代表 shamt。在执行指令 sra、srl、sll 时需要 ALUSrc1 信号为 1。这是因为指令[25-21]在这些指令中为 0，所以在寄存器中的数据也为 0，而这些实际是指令[20-16]和指令[10-6]做算术运算得到的。

c) 由 MemtoReg 信号控制的多路选择器，输入 2 对应的是 PC+4。执行指令 jalr 和 jal 时需要 MemtoReg 信号为 2。因为这两个指令需要将 PC+4 的当前值储存到相应寄存器中去。

d) PCSrc 信号控制的多路选择器，输入 2 对应的是指令[25-21]对应寄存器中的值。在执行指令 jalr 和 jr 时需要 PCSrc 信号为 2。这是因为这两个指令要求跳转到寄存器所指地址。

e）当有符号数进行有符号扩展时 ExtOp 信号为 1，对应的无符号数进行无符号扩展时 ExtOp 信号为 0。ExtOp 控制信号，在控制符号扩展时判断是有符号扩展还是无符号扩展。

f）采用指令 shamt、rd、sll、rt，并使得 rd=rt=0，且偏移量 shamt=0 即可使得指令格式为全 0，即可以实现要求。若想实现 nop 指令，
无需修改处理器结构。

2.根据对控制信号的理解，填写真值表

	PCSrc[1:0]	Branch	RegWrite	RegDst[1:0]	MenRead	MenWrite	MentoReg[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	0	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	0	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

2 完成控制器

1.文件夹中的处理器 verilog 文件描述了单周期 CPU 的整体情况。其中，Control.v 为控制器模块的代码

```
1
2  module Control(OpCode, Funct,
3      PCSrc, Branch, RegWrite, RegDst,
4      MemRead, MemWrite, MemtoReg,
5      ALUSrc1, ALUSrc2, ExtOp, LuOp, ALUOp);
6      input [5:0] OpCode;
7      input [5:0] Funct;
8      output [1:0] PCSrc;
9      output Branch;
10     output RegWrite;
11     output [1:0] RegDst;
12     output MemRead;
13     output MemWrite;
14     output [1:0] MemtoReg;
15     output ALUSrc1;
16     output ALUSrc2;
17     output ExtOp;
18     output LuOp;
19     output [3:0] ALUOp;
20
21     // Your code below
22
23         assign RegDst[1:0]=
24         (OpCode==6'h03)? 2'b10:
25         (OpCode==6'h0f)? 2'b01:
26         (OpCode==6'h08)? 2'b01:
27         (OpCode==6'h09)? 2'b01:
28         (OpCode==6'h0c)? 2'b01:
29         (OpCode==6'h0a)? 2'b01:
30         (OpCode==6'h0b)? 2'b01: 2'b00;
31     assign Branch=(OpCode==6'h04)? 1:0;
32         assign PCSrc[1:0]=
33         (OpCode==6'h02)? 2'b01:
34         (OpCode==6'h03)? 2'b01:
35         (OpCode==6'h00 && Funct==6'b08)? 2'b10:
36         (OpCode==6'h00 && Funct==6'b09)? 2'b10:2'b00;
37         assign MemtoReg[1:0]=
38         (OpCode==6'h23)? 2'b01:
39         (OpCode==6'h03)? 2'b10:
```

```
40          (OpCode==6'h00 && Funct==6'b09)? 2'b10:2'b00;
41          assign MemRead=(OpCode==6'h23)? 1:0;
42          assign RegWrite=
43          (OpCode==6'h2b)? 1:
44          (OpCode==6'h04)? 1:
45          (OpCode==6'h02)? 1:
46          (OpCode==6'h00 && Funct==6'b08)? 1:0;
47          assign MemWrite=(OpCode==6'h2b)? 1:0;
48          assign ExtOp=
49          (OpCode==6'h09)? 0:
50          (OpCode==6'h0b)? 0:1;
51      assign ALUSrc1=
52          (OpCode==6'h00 && Funct==6'b00)? 1:
53          (OpCode==6'h00 && Funct==6'b02)? 1:
54          (OpCode==6'h00 && Funct==6'b03)? 1: 0;
55      assign ALUSrc2 =
56          (OpCode==6'h23)? 1:
57          (OpCode==6'h2b)? 1:
58          (OpCode==6'h0f)? 1:
59          (OpCode==6'h08)? 1:
60          (OpCode==6'h09)? 1:
61          (OpCode==6'h0c)? 1:
62          (OpCode==6'h0a)? 1:
63          (OpCode==6'h0b)? 1:0;
64          assign LuOp=(OpCode==6'h0f)? 1:0;
65
66      // Your code above
67
68      assign ALUOp[2:0] =
69          (OpCode == 6'h00)? 3'b010:
70          (OpCode == 6'h04)? 3'b001:
71          (OpCode == 6'h0c)? 3'b100:
72          (OpCode == 6'h0a || OpCode == 6'h0b)? 3'b101:
73          3'b000;
74
75      assign ALUOp[3] = OpCode[0];
76
77  endmodule
```

3.阅读 InstructionMemory.v，根据注释理解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
```

解：首先明确这段程序执行足够长时间后会发生死循环。

此时寄存器中的$a0=0xd4990000

$a1=（54321）$_{10}$=0x0000d431【补码】

$a2=（-734986240）$_{10}$=0xd4310000【补码】

$a3=0xffffd431

1、 指令 addi $a0,$zero,12345 用来计算$a0，之后指令 lui $a0,-11111 将-11111=0xd499 看做无符号数装入$a0 高 16 位，故而得到$a1 的值；

2、 指令 addiu $a1,$zero,-11215 将-11215 当作无符号数与零相加赋给了$a1。-11215 的 16 位补码形式为 1101010000110001，将其当作无符号数进行 32 位扩展后代表 54321 的补码，从而可以得到$a1；

3、 指令 sll $a2,$a1,16 将 54321 的补码向左移动了 16 位后赋予$a2，故而$a2 中的数为（11010100001100010000000000000000）$_2$，得到$a2；

4、 指令 sra $a3,$a2,16 将$a2 算术右移 16 位，高位补符号位，故而得到$a3；

$t0=0xa8ca0000

$t1=0xffa8ca00

$t2=oxffffcfc7

1、 指令 add $t0,$a2,$a0 将$a2=0xd4310000 和$a0=0xd4990000 相加产生溢出结果为 0xa8ca，故而可以得到$t0；

2、 指令 sra $t1,$t0,8 将$t0 算术右移 8 位，高位补符号位，从而得到$t1；

3、 指令 addi $t2, $zero, -12345 将立即数-12345=0xcfc7 赋予$t2，可以得到其值；

$v0=ox00000001

$v1=ox00000001

1、 指令 slt $v0,$a0,$t2 判断$a0 小于$t2，从而得到$v0；

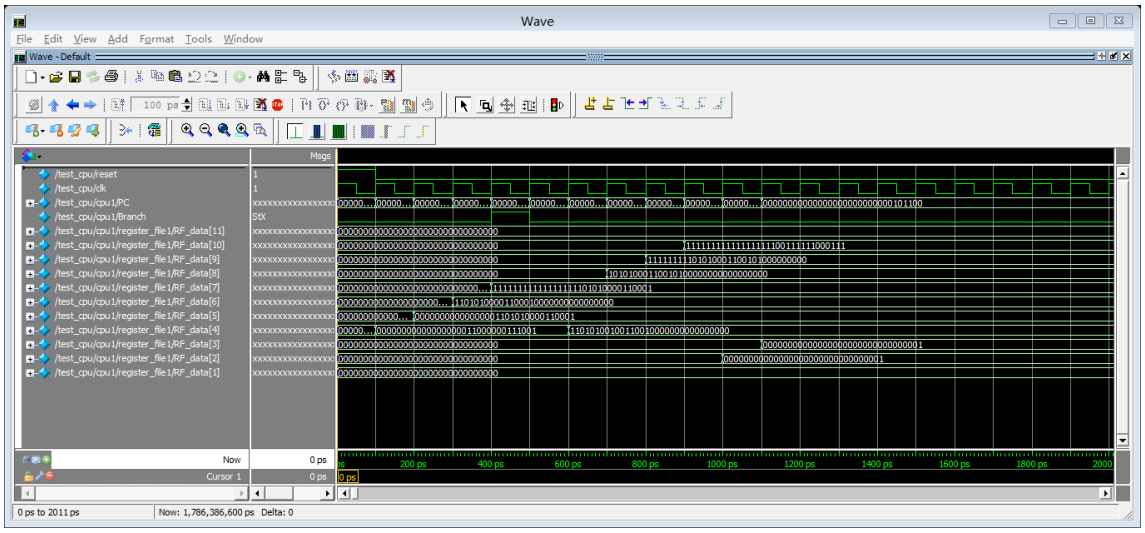
2、 指令 sltu $v1,$a0,$t2 判断无符号数 $a0=0xd4990000 小于$t2=0xffffcfc7，从而得到$v1；

如果已知某一时刻在某寄存器中存放着数0xffffcfc7,不能判断出它是有符号数还是无符号数。这是因为在指令slt $v0, $a0, $t2执行完毕后寄存器$t2=oxffffcfc7，这一表示既可为有符号数也可为无符号数。

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 提供复位和时钟信号。观察仿真结果中各寄存器和控制信号的变化。

仿真结果为：



a）PC 变化情况为：0->4->8->12->16->20->24->28->32->36->40->44。随后保持 44 的状态。

b）400~500ns 期间 Branch 没有改变或者影响 PC 由 16->20 的进程，因为其信号为 1。

c）1、100~200ns 期间，PC=4，对应的指令为 addiu $a1，$zero，-11215 此时$a1 的值是 0x00000000。
2、200~300ns 期间，指令将-11215 当作无符号数与零相加赋给了$a1，所以其值是 0x0000d431，-11215 的 16 位补码形式为 1101010000110001，将其当作无符号数进行 32 位扩展后代表 54321 的补码也即 0x0000d431。在第二条指令运行完毕也即 200ns 以后$a1 的值才有效。
3、即使在下一条指令立即使用到了$a1的值，但不会出现错误。因为该处理器为单周期处理器，每当一条指令执行完毕后才会去执行下一条指令。

d） a0 = 1101_0100_1001_1001_0000_0000_0000_0000

a1 = 0000_0000_0000_0000_1101_0100_0011_0001

a2 = 1101_0100_0011_0001_0000_0000_0000_0000

a3 = 1111_1111_1111_1111_1101_0100_0011_0001

t0 = 1010_1000_1100_1010_0000_0000_0000_0000

t1 = 1111_1111_1010_1000_1100_1010_0000_0000

t2 = 1111_1111_1111_1111_1100_1111_1100_0111

v0 = 0000_0000_0000_0000_0000_0000_0000_0001

v1 = 0000_0000_0000_0000_0000_0000_0000_0001

与预期一致

3 执行汇编程序

1、对汇编代码进行注释

这段程序实现的为对从 1 到 n 之间的整数求和。Loop 使得程序运行结束后进入死循环，sum 将小于等于 n 的正整数进行压栈，L1 将栈中数据取出并进行求和。

```
addi $a0, $zero, 3          #定义参数用于计算不小于3的正整数的和
    jal sum                 #跳转到sum语句
Loop:
    beq $zero, $zero, Loop  #程序运行完毕则进入死循环
sum:
    addi $sp, $sp, -8       #为寄存器开空间
    sw $ra, 4($sp)          #压栈$ra
    sw $a0, 0($sp)          #压栈$a0
    slti $t0, $a0, 1        #$a0小于1，$t0==1
    beq $t0, $zero, L1      #$t0==1，$a0!==1时，跳转到L1语句
    xor $v0, $zero, $zero   #$v0==0，之后清零
    addi $sp, $sp, 8        #恢复堆栈指针
    jr $ra                  #返回调用函数
L1:
    addi $a0, $a0, -1       #$a0==$a0-1
    jal sum                 #跳转到sum语句
    lw $a0, 0($sp)          #恢复$a0
    lw $ra, 4($sp)          #恢复$ra
    addi $sp, $sp, 8        #恢复堆栈指针
    add $v0, $a0, $v0       #将堆栈中各数据相加
    jr $ra                  #返回调用函数
```

2、将这段汇编程序翻译成机器码。

1、Loop 的 PC 地址为 8，sum 的 PC 地址为 12，L1 的 PC 地址为 44，再根据不同指令寻址方式的不同翻译成不同长度的机器码。

2、Jal 指令后有 26 位地址码，寻址时在前面加上当前 PC+4 的高 4 位，即 0000 在最后加两位 0。

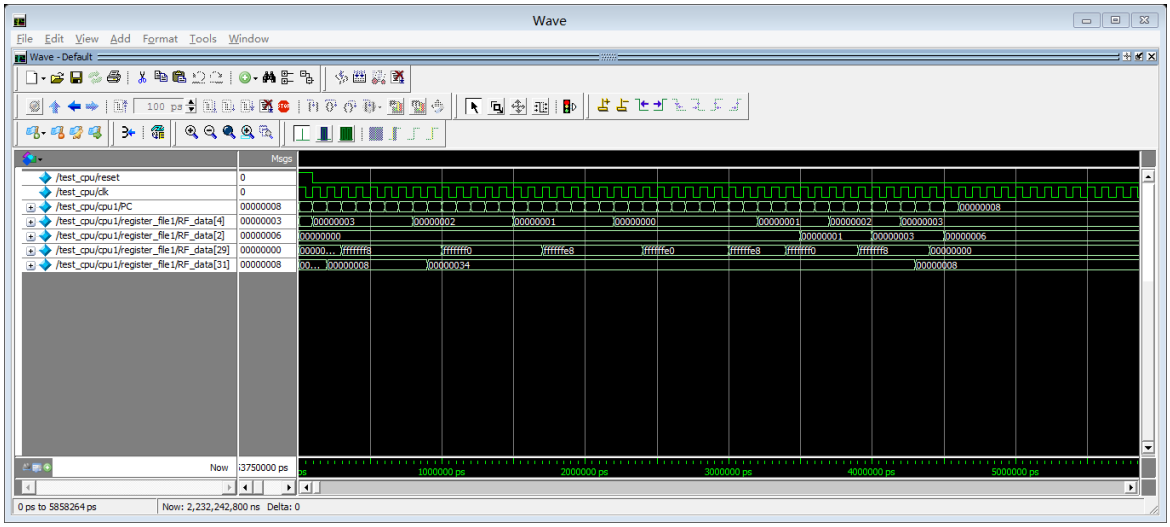
3、Sum 对应的 PC 地址为 0000_0000_0000_0000_0000_0000_0000_1100，从而翻译时的 26 位地址码为 0000_0000_0000_0000_0000_0000_11。

```
001000_00000_00100_0000_0000_0000_0011
## addi $a0,$zero,3
000011_00000_00000_00000_00000_00001_1
## jal sum
000100_00000_00000_1111_1111_1111_1111
## beq $zero,$zero,Loop
001000_11101_11101_1111_1111_1111_1000
## addi $sp,$sp,-8
101011_11101_11111_0000_0000_0000_0100
## sw $ra 4($sp)
101011_11101_00100_0000_0000_0000_0000
## sw $a0,0($sp)
001010_00100_01000_0000_0000_0000_0001
## slti $to,$a0,1
000100_01000_00000_0000_0000_0000_1100
## beq $t0,$zero,L1
000000_00000_00000_00010_00000_ 100110
## xor $v0,$zero,$zero
001000_11101_11101_0000_0000_0000_1000
## addi $sp,$sp,8
000000_11111_00000_00000_00000_001000
## jr $ra

## L1:
001000_00100_00100_1111_1111_1111_1111
## addi $a0,$a0,-1
000011_00000_00000_00000_00000_00001_1
## jal sum
100011_11101_00100_0000_0000_0000_0000
## lw $a0,0($sp)
100011_11101_11111_0000_0000_0000_0100
## lw $ra,4($sp)
001000_11101_11101_0000_0000_0000_1000
## addi $sp,$sp,8
000000_00100_00010_00010_00000_100000
## add $v0,$a0,$v0
000000_11111_00000_00000_00000_001000
## jr $ra
```

3、修改 InstructionMemory.v，使 CPU 运行程序。

修改之后利用 Modelism 进行波形仿真从而进行分析，获得的波形为：



a) 在 5000ns 时，可以观察到$a0=0x00000003、$v0=0x00000006，与预期一致

b) 下面对 PC, $a0, $v0, $sp, $ra 的变化情况进行总结与分析。

i. 0~100ns

开始执行指令 addi $a0, $zero, 3

PC=0x000000000; $a0=0x000000000; $v0=0x00000000; $sp=0x00000000; $ra=0x00000000;

ii. 100~300ns

addi $a0, $zero, 3 执行完毕，保存下一条指令地址到$ra 中，$ra=0x00000008，同时跳转到 PC=0x00000000c。

开始执行指令 addi $sp, $sp, -8

PC=0x00000000c; $a0=0x000000003; $v0=0x00000000; $sp=0x00000000; $ra=0x00000008;

iii. 300~400ns

addi $sp, $sp, -8 执行完毕，结果为$sp=0xfffffff8。

开始执行指令 sw $ra, 4($sp)

PC=0x000000010; $a0=0x000000003; $v0=0x00000000; $sp=0xfffffff8; $ra=0x00000008;

iv. 400~600ns

sw $ra, 4($sp)执行完毕，压栈完毕。

开始执行指令 slti $t0, $a0, 1

PC=0x000000018; $a0=0x000000003; $v0=0x00000000; $sp=0xfffffff8; $ra=0x00000008;

v. 600~800ns

slti $t0, $a0, 1 执行完毕，由于$a0 不小于 1，故而$t0=0。

之后 beq $t0, $zero, L1 执行完毕，使用分支，PC 跳转到 0x00000002c。

开始执行指令 addi $a0, $a0, -1

PC=0x00000002c; $a0=0x000000003; $v0=0x00000000; $sp=0xfffffff8; $ra=0x00000008;

vi. 800~1000ns

addi $a0,$a0,-1 执行完毕，$a0==$a0-1。
之后 jal sum 执行完毕，PC 跳转至 0x00000000c。
开始执行指令 addi $sp,$sp,-8
PC=0x00000000c; $a0=0x000000002; $v0=0x00000000; $sp=0xffffffff8;
$ra=0x00000034;

vii. 1000~1200ns

addi $sp,$sp,-8 执行完毕，$sp=0xffffffff0。
之后 sw $ra,4($sp)执行完毕，$ra 压栈完毕。
开始执行指令 sw $a0,0($sp)
PC=0x000000014; $a0=0x000000002; $v0=0x00000000; $sp=0xffffffff0;
$ra=0x00000034;

viii. 1200~1400ns

sw $a0,0($sp)执行完毕，$a0 压栈完毕。
紧接着 slti $t0,$a0,1 执行完毕；判断$a0 不小于 1，令$t0=0。
开始执行指令 beq $t0, $zero, L1
PC=0x00000001c; $a0=0x000000002; $v0=0x00000000; $sp=0xffffffff0;
$ra=0x00000034;

ix. 1400~1600ns

beq $t0,$zero,L1 执行完毕，进行分支，PC 跳转到 0x00000002c。
之后开始执行 addi $a0,$a0,-1，$a0==$a0-1。
完毕后开始执行指令 jal sum
PC=0x000000030; $a0=0x000000001; $v0=0x00000000; $sp=0xffffffff0;
$ra=0x00000034;

x. 1600~1800ns

addi $sp,$sp,-8 执行完毕，$sp=0xffffffe8。
开始执行指令 sw $ra, 4($sp)
PC=0x000000010; $a0=0x000000001; $v0=0x00000000; $sp=0xffffffe8;
$ra=0x00000034;

xi. 1800~2200ns

sw $ra,4($sp)执行完毕，$ra 压栈完毕，$a0 压栈完毕，$t0=0。
开始执行指令 beq $t0, $zero, L1
PC=0x00000001c; $a0=0x000000001; $v0=0x00000000; $sp=0xffffffe8;
$ra=0x00000034;

xii. 2100~2200ns

beq $t0,$zero,L1 执行完毕，进行分支， PC 跳转到 0x00000002c。
开始执行指令 addi $a0,$a0,-1
PC=0x00000002c; $a0=0x000000001; $v0=0x00000000; $sp=0xffffffe8;
$ra=0x00000034;

xiii. 2200~2300ns

addi $a0,$a0,-1 执行完毕，$a0==$a0-1。
开始执行指令 jal sum
PC=0x000000030; $a0=0x000000000; $v0=0x00000000; $sp=0xffffffe8;
$ra=0x00000034;

xiv. 2300~2500ns

jal sum 执行完毕，PC 跳转至 0x00000000c，$sp=0xfffffffe0。
开始执行指令 sw $ra,4($sp)
PC=0x000000010; $a0=0x000000000; $v0=0x00000000; $sp=0xfffffffe0;
$ra=0x00000034;

xv. 2500~2800ns

sw $ra,4($sp)执行完毕，$ra 压栈完毕，$a0 压栈完毕，$t0=1。
开始执行指令 beq $t0,$zero,L1
PC=0x00000001c; $a0=0x000000000; $v0=0x00000000; $sp=0xfffffffe0;
$ra=0x00000034;

xvi. 2800~3200ns

beq $t0,$zero,L1 执行完毕，开始分支，PC=PC+4。之后 PC 跳转到寄存器$ra 所存地址，即 PC=0x000000034。
开始执行指令 lw $a0,0($sp)
PC=0x000000034; $a0=0x000000000; $v0=0x00000000; $sp=0xfffffffe8;
$ra=0x00000034;

xvii. 3200~3400ns

lw $a0, 0($sp)执行完毕，开始从栈中取出以前所存内容，
开始执行指令 addi $sp,$sp,8
PC=0x00000003c; $a0=0x000000001; $v0=0x00000000; $sp=0xfffffffe8;
$ra=0x00000034;

xviii. 3400~3700ns

addi $sp,$sp,8 执行完毕，$sp 中内容增 8。将$a0 累加到$v0 中，PC 跳转到寄存器$ra 所存地址，PC=0x000000034。
开始执行指令 lw $a0,0($sp)
PC=0x000000034; $a0=0x000000001; $v0=0x00000001; $sp=0xffffffff0;
$ra=0x00000034;

xix. 3700~4100ns

lw $a0,0($sp)执行完毕，从栈中取出以前所存内容， $sp 中内容增 8，将$a0 累加到$v0 中。
开始执行指令 jr $ra
PC=0x000000044; $a0=0x000000002; $v0=0x00000003; $sp=0xffffffff8;
$ra=0x00000034;

xx. 4100~4300ns

jr $ra 执行完毕，PC 跳转到寄存器$ra 所存地址，PC=0x000000034。从栈中取出以前所存内容。
开始执行指令 lw $ra,4($sp)
PC=0x000000038; $a0=0x000000003; $v0=0x00000003; $sp=0xffffffff8;
$ra=0x00000034;

xxi. 4300~4400ns

lw $ra,4($sp)执行完毕，$sp 中内容增 8。
开始执行指令 add $v0,$a0,$v0
PC=0x000000040; $a0=0x000000003; $v0=0x00000003; $sp=0x00000000;
$ra=0x00000008;

xxii. 4500 之后进入死循环

指令 add $v0,$a0,$v0 执行完毕，将$a0 累加到$v0 中，PC 跳转至 0x000000008，循环执行指令 beq $zero, $zero, Loop
PC=0x000000008; $a0=0x000000003; $v0=0x00000006; $sp=0x00000000;
$ra=0x00000008;