

3. 7. 8. 12. 13. 14. 15. 16. 17

3. (1) nop    addi x0, x0, 0

(2) ret    jalr x0, x1, 0     $\Rightarrow$      $x_1, ra$  返回地址  
 $PC \leftarrow PC + Regs[x_1]$

(3) call offset    远程序调用了程序

auipc x1, offset [31:12]     $\bar{r}$

jalr x1, x1, offset [11:0]

(4) mv rd, rs    addi rd, rs, 0

(5) rdcycle rd    crrs rd, cycle, x0

(6) sext.w rd, rs    addiw rd, rs, 0

7 (1) add to, t1, t2

sub t3, to, t1

addi t2, t2, 0

bge t3, t4, overflow

(2) ~~addi t3, zero, 4294967295~~

bgt zero, to, overflow

(3) x86 通过 ~~寄存器~~ 标志位来测: 当有符号数相加时

超过了有符号数所表示的范围, 标志位置1

ARM 架构中的 ALU (算术逻辑单元) 也有一个溢出标志位, 用于

检测加减法的溢出, 有符号数相加或相减时,  $-V$  置1

8.	$Op = DIVU$	$Op = REMU$	$Op = DIV$	$Op = REM$
	$0x\text{FFFFFFFFFFFFFF}$	$\times$	$0x\text{FFFFFFFFFFFFFF}$	$\times$

(1) 会引起异常，CPU会抛出异常，中断当前正在执行的程序

意义 1. 避免错误的计算结果

2. 增强代码的可靠性，将此视为异常，可以帮助程序员在开发和调试过程中更容易地发现与排除错误

3. 确保程序的可移植性，可确保代码在不同的处理器和操作系统上运行

(2) 无效操作<sup>发生</sup>，非法操作，此位被置1

行不交

DZ 除以0：当除数为0，此位被置1

溢出：结果的绝对值大，无法被浮点数表示，置1

下溢，太小，置1

不精确 结果不能准确地用浮点数表示，置1

不会直接陷入，但一些情况下需要由操作系统的异常处理程序处理

(3) x86: 除数为0会引发“divide error”异常，即除0异常，这是一种致命异常，当发生这种异常时，CPU会将异常号压入堆栈中，并跳转到异常处理程序中。在异常处理程序中，可通过读取状态寄存器中的标志位来确定发生异常的原因，并采取相应的处理措施。

ARM: 除数0会引发“division by zero”异常，即未定义指令异常，当发生这种异常时，CPU会将异常号压入堆栈中，并跳转到异常处理程序中。在异常处理程序中，可以通过读取协处理器中的状态寄存器来确定发生异常的原因。

- 12.
- (1) Linux Kernel S
  - (2) BootROM M
  - (3) BootLoader M
  - (4) USB Driver S
  - (5) vim U

vecMul:

13

```

addi sp, sp, -32
add x11, x0, x0    # i=0
sd ra, 24(sp)
addi x12, x0, 100  # x12=100
sd s0, 16(sp)
addi s0, sp, 32

```

Loop:

```

bge x11, x12, exit
sll x13, x11, 2    # i * 4
add x15, x13, t0   # &A + i
add x14, x13, t1   # &B + i
lw x15, 0(x15)     # *CA + i)
lw x14, 0(x14)

```

```
mul x15, x14, t2
```

```
addi x11, x11, 1
```

j Loop

```
sw x15, 0(x13)
```

```
exit: ld ra, 24(sp)
```

```
ld s0, 16(sp)
```

```
addi sp, sp, 32
```

```
ret.
```

14.

Part: 1

bgt a0, a1, Part 2

sub a2, a0, a1

Part: 2 j exit

add a2, a0, a1

j exit

exit:

15.

```

addi t2, t0, 0 # t2 = t1 = P
lw t2, 0(t2) # t2 = *P
addi t2, zero, t0 # P[0] = P
sw t2, 0(t0) # 存储地址 t0
addi t3, t0, 4 # t3 指向 P[1]
lw t3, 0(t3) # t3 = *P[1]
addi t3, zero, t1 # *t3 即 P[1] = a
addi t4, t0, 8, sw t3, 0(t3)
sll t4, t1, 2 # a * 4
addi t5, t0, t4 # t5 指向 P[2]
lw t5, 0(t5) # t5 = *P[2]
addi t5, zero, t1 # P[2] = a
sw t5, 0(t5)

```

swap

16.

```

lw t2, 0(t0) # t2 = *a
lw t3, 0(t1) # t3 = *b
addi t4, t2, 0 # t4 = tmp = *a
addi t2, t3, 0 # *a = *b
addi t3, t4, 0 # *b = tmp
sw t2, 0(t0)
sw t3, 0(t1)
ret

```