

3. (1) 加操作 addi x0, x0, 0 .

(2) jalr x0, 0 (x1) .

(3) 调用函数 njmpc ra, (offset)  
jalr rd, (offset)(ra) .

(4) 复制: addi rd, rs, 0

(5) rdtime to  
mv rd, to .

(6). slli to, rs, 16

srai rd, to, 16 .

7. (1) sub to, to, t1

bne to, t2, overflow

(2) add to, t1, t2

bltn to, t1, overflow .

"bltn"指令会在  $to < t1$  时跳到 overflow 处, 这说明了加法溢出

(3) 在 x86 中, add 及 adc 指令影响 CF 标志位, 其会在加法后设置为 1, 如果出现进位, 否则设为 0, 可以用以下指令: add dst, src1, src2

jno no-overflow .

no-overflow

在 ARM 中, ADD 及 ADCS 也会影响 CF 标志位, VF 标志位与 CCR, 加法结果不适合用寄存器时, 溢出标志位设为 1, 用以下指令 / ADD dst, src1, src2  
BVS overflow

在 MIPS 中, 没有专门指令读取 OF 标志位, 因此需要使用其他指令进行检测。

用以下指令 add to, t1, t2

slti t3, t1, 0 .

slti t4, t2, 0 .

xor t5, t3, t4 .

slti t6, t5, 0 .

xor t7, t3, t6 .

and t8, t5, t7 .

beq t8, 1, overflow .

8. (1) ① 异常 ② rs1 ③ 异常 ④ rs1

对于整型除法, 当除数为 0 时, 会抛出, 称为 Divide-by-Zero 异常。在这种情况下, 指令执行无法得到正确结果, 因为数学上无法定义除以 0。采用这种设计是为了保证指令执行可靠性, 同时也能够帮助程序员及时发现代码中错误并更正。

- (2) FFLAGS[0]: 溢出操作数基址, 操作数非负值时置位。  
 FFLAGS[1]: 非规格化数基址, 操作数太小而无法表示为规格化数时置位  
 FFLAGS[2]: 除以0有溢位, 商除数为0置位  
 FFLAGS[3]: 溢出标志位, 操作结果超过浮点数范围置位。  
 FFLAGS[4]: 下溢标志位, 操作结果太小而无法表示为规格化数时置位。  
 FFLAGS[5]: 不精确标志位, 商操作结果无法准确表示为浮点数时置位。  
 当fflags被置位时, 并不会使处理器陷入特权级别, 处理器会继续执行程序, 但程序员可以通过检查fflags的值来判断是否出现了异常情况
- (3). x86: 当使用 idiv 或 div 指令进行有符号或无符号整数除法时, 如果除数为0, 将会触发一个异常, 称为 Divide Error 异常。该类指令会把异常的类型保存在 EFLAGS 寄存器的 DF 位上, 并将 CPU 控制转移到 IDT 中异常处理程序。如果使用 fdiv 或 fdivp 指令进行浮点数除法, 如果除数为0时, FPU 控制字中的相关标志位会被置位, 并且 FPU 将把结果设置为非数值, 如 Inf 或 NaN。
- ARM: 当使用 sdiv 或 ndiv 指令进行有符号或无符号整数除法时, 如果除数为0, 会触发 Divide by zero 异常。指令会把异常的类型保存在 CPSR 寄存器。

12. (1) 用户模式或管理模式  
 (2) 机器模式  
 (3) 机器模式或监督模式  
 (4) 监督模式  
 (5) 用户模式

13. addi t3, zero, 0

```

loop:
  bge t3, 100, end_loop
  lw t4, 0(t1)
  lw t5, 0(t2)
  mul t4, t4, t5
  sw t4, 0(t0).
  addi t3, t3, 1.
  addi t0, t0, 4
  addi t1, t1, 4.

```

loop.

end\_loop:
 lw t0, 0(t0)

14.   
lw a0, 0(sp).  
lw a1, 4(sp).  
addi a2, zero, 0.  
bit a0, a1, else.  
add a2, a0, a1.  
} end\_if.  
else:  
snb a2, a0, a1  
end\_if:  
sw a2, 8(sp).  
jr ra.

15. li t0, 1b.  
li t7, 9.  
call malloc.  
mv t0, a0.  
mv t1, t0.  
sw t1, 0(t0).  
li t1, 3.  
addi t2, t0, 4  
sw t1, 0(t2)  
sw t1, 0(t0).  
slli t3, t1, 2  
add t3, t3, t0  
sw t1, 0(t3).

16. swap:  
addi sp, sp, -4  
sw ra, 0(sp).  
lw t2, 0(t0).  
sw t2, 0(sp).  
lw t2, 0(t1).  
sw t2, 0(t0).  
lw t2, 0(sp).  
sw t2, 0(t1).  
lw ra, 0(sp).  
addi sp, sp, 4

17. 从RISC-V汇编代码实现了一个循环，循环执行  
以下操作直到“a0”的值等于a2”。  
①  $T_2 = a_1$  “值1往”  
② 将“a0”值加1  
③ 跳转到循环开始 (loop 标号处)  
最后直到 a0 的值达到 30 为止，结束执行。