

3. (1) 空操作 `addi x0, x0, 0` .  
 (2) `jalu x0, 0(x1)` .  
 (3) 调用函数 `anipc ra, (offset)`  
`jalu 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  $t_0 < t_1$  时跳到 overflow 处, 这说明了加法溢出

(3) 在 x86 中, `add` 与 `adc` 都会影响 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` 与 `ADCS` 也会影响 CF 标志位, VF 标志位与 CCR, 加法结果不适合目标寄存器时, 溢出标志位被设为 1, 用以下指令: `ADD dst, src1, src2`  
`BVS overflow`

在 MIPS 中, 没有专用指令来读取 CF 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`  
`sli t3, t1, 0` .  
`sli t4, t2, 0` .  
`xor t5, t3, t4` .  
`sli t6, to, 0` .  
`xor t7, t3, t6` .  
`and t8, t5, t7` .  
`beqz 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正确可靠性, 同时也能够帮助程序员及时发现代码中错误并修正。

(2) **FFLAG[0]**: 无效操作标志位, 操作数非数值时置位.

**FFLAGS[1]**: 非规格化数据标志位, 操作数太小而无法表示为规格化数时置位.

**FFLAGS[2]**: 除以0标志位, 当除数为0时置位.

**FFLAGS[3]**: 溢出标志位, 操作结果超过浮点数值范围时置位.

**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, 当操作结果无法准确表示为浮点数时置位.

当fflags被置位时, 不会导致处理器陷入系统调用, 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. 除法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 `fddiv` 或 `fldivp` 指令进行浮点数除法, 当除数为0时, **fpn** 控制字中的相关标志位会被置位, 并且 **fpn** 会把结果设置为特定值, 如 **Inf** 或 **NaN**.

**ARM**: 当使用 `sdiv` 或 `udiv` 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) 用户模式或管理级模式

(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)`

`muli t4, t4, t5`

`sw t4, 0(t0).`

`addi t3, t3, 1.`

`addi t0, t0, 4`

`addi t1, t1, 4.`

`j loop.`

`end-loop:`

`lw t0, 0(t0).`

`...`

14. lw a0, 0(sp).  
 lw a1, 4(sp).  
 addi a2, zero, 0.  
 blt a0, a1, else.  
 add a2, a0, a1.  
 j end-if.  
 else:  
 snb a2, a0, a1  
 end-if:  
 sw a2, 8(sp).  
 jr ra.

15. li to, 16.  
 li t7, 9.  
 call malloc.  
 mv to, a0.  
 mv t1, to.  
 sw t1, 0(to).  
 li t1, 3.  
 addi t2, to, 4  
 sw t1, 0(t2)  
 sw t1, 0(t0).  
 slli t3, t1, 2  
 add t3, t3, to  
 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. 这段RISC-V汇编代码实现了一个循环，循环执行。  
 以下操作直到“a0”的值大于“a2”：  
 ① 左移“a1”值1位  
 ② 将“a0”值加1  
 ③ 跳转到循环开始(loop标签处)。  
 最后直到a0值达到30为止，结束执行。