3.
(1)nop 等效于 addi x0,x0,0
(2)ret 等效于 jalr x0,0
(3)call offset 等效于 jal offset+PC
(4)mv rd,rs 等效于 addi rd,rs,0
(5)rdcycle rd 等效于 rdstime rd
(6)sext.w rd 等效于 addiw rd,rs,0

7.
(1)
SLTI t3，t2，0　　#思路：分小于零和大于零两种情况比较 t1+t2 和 t1 的大小
SLT t4，t0，t1
(2)
ADD x0，x1，x2
BLTU x0，x1，overflow
(3)

在 x86 架构中，检测加法溢出使用符号位（最高位）和进位标志位（Carry Flag）组合确定。

在执行加法指令时，符号位的处理方式如下：对于有符号整数加法，x86 将源操作数和目的操作数的符号位视为第一位值，在两个操作数中执行标准的二进制加法。对于无符号整数加法，x86 不将操作数视为有符号或无符号，而是执行标准的二进制加法。加法结果的符号位反映了两个操作数之和的符号。如果两个操作数的符号不同，加法结果的符号位将被置为 0，反之，为 1。如果发生了溢出（最高位进位），符号位将不再反映正确的结果。

x86 的 Carry Flag 用于检测进位或借位情况。如果向无符号整数的最高位进位，或从有符号整数的最高位置借用，则 Carry Flag 将被设置为 1，否则为 0。

在执行加法指令后，可以通过检查符号位和 Carry Flag 的值来确定操作是否溢出：如果符号位和进位标志都为 1，则发生了有符号加法溢出。如果进位标志为 1，但符号位为 0，则发生了无符号加法溢出。

为了检测整数加法溢出，x86 架构使得 Carry Flag 和 Overflow Flag 位在 EFLAGS 寄存器中面向所有代码可使用。这两个标志都提供了有关进行算术运算时表现的额外信息。

8 阅读 RISC-V 规范以了解 RISC-V 对除数为 0 的除法指令的处理方法，回答以下问题。
（1）对整型除法，填写下表。整型除法中除数为 0 是否会引起 RISC-V 抛出异常？试分析为什么采取这样的设计。

指令	rs1	rs2	Op=DIVU 时 rd 值	Op=REMU 时 rd 值	Op=DIV 时 rd 值	Op=REM 时 rd 值
Op rd,rsl,rs2	x	0	0xfff...f 即最大的 64 位 UINT 型整数	求余结果为被除数	0xfff...f 即-1	求余结果为被除数

整型除法中除数为 0 会引起 RISC-V 抛出异常：是“整型除数溢出异常”，采取这样设计的原因是：①避免在除法运算时出现不确定的结果。当除数为 0 时，无法得到一个确定的商（返回值 rd），这会可能引发不可预料的错误或导致执行异常的代码。②为了避免这种情况发生，RISC-V 采取了这样的设计，即当除数为 0 时，抛出异常以提示程序员检查他们的代码。

（2）对浮点除法，除数为 0 将会引起 fcsr 控制寄存器中的相关标志位被置位。下图给出了 fcsr 的构成，请说明 fflags 的各位分别代表什么含义。fflags 被置位是否会使得处理器陷入系统调用？

fflags 字段是浮点运算中的异常标志——NV：非规范的操作（Invalid operation）；DZ：除以零（Divide-by-zero）；OF：上溢（Overflow）；UF：下溢（Underflow）；NX：不精确（Inexact）。

（3）调研其他指令集架构（如 x86、ARM 等）是如何处理除数为 0 的。

在 x86 和 ARM 等指令集架构中，除法指令对于除数为 0 都会产生异常并触发异常处理程序的执行以防在程序执行过程中出现不确定的结果，同时也保证了程序的正确性和可靠性。如下：

x86 架构：x86 架构中的除法指令被称为 idiv 指令，其将 EDX 和 EAX 寄存器中的 64 位有符号整数值看作一个被除数，再将一个 32 位有符号整数值看作除数，执行除法运算并将结果放入 EAX 寄存器中。如果除数为 0，则会产生一个“除数为零异常”并触发异常处理程序的执行。

ARM 架构：ARM 架构下的除法指令被称为 SDIV（有符号整数除法指令）或 UDIV（无符号整数除法指令），都采用 RD 指令将结果存入指定寄存器中。如果除数为 0，则会产生一个“数据终止异常”并导致程序终止执行，ARM 处理器会跳转至异常向量中存储的指令进行异常处理。

12.

(1)位于 M-mode 和 S-mode

(2)位于 M-mode

(3)位于 S-mode 和 U-mode

(4)位于 S-mode

(5)位于 U-mode

13.

```
vecMul(int*, int*, int):
        addi    sp,sp,-64
        sd      s0,56(sp)
        addi    s0,sp,64
        sd      t0,-40(s0)
        sd      t1,-48(s0)
        mv      t5,t2
        sw      t5,-52(s0)
        sw      0,-20(s0)       #建立堆栈保存 i 的值
.L3:
        lw      t5,-20(s0)
        sext.w  t4,t5
        li      t5,100
        beq     t4,t5,.L2       #完成对于 i<100 的判断（循环条件）
        lw      t5,-20(s0)      #t5=i
        slli    t5,t5,2         #t5=4*i
        ld      t4,-48(s0)      #t4 的值为 B 的地址
        add     t5,t4,t5        # t5 相当于数组元素地址，每次下标加 1，地址要加 4（int 型变
量占有四个字节）
        lw      t4,0(t5)        #t4=B[i]
        lw      t5,-20(s0)      #t5=i
        slli    t5,t5,2         #t5=4*i
        ld      t3,-40(s0)      #t3 的值为 A 的地址
        add     t5,t3,t5        # t5 相当于数组元素地址，每次下标加 1，地址要加 4
        lw      t3,-52(s0)      #t3=C
        mulw    t4,t3,t4        #t4=B[i]*C
        sw      t4,0(t5)        #将 B[i]*C 存放到地址为 t5 也就是 A[i]的堆栈中
        lw      t5,-20(s0)      #t5=i
        addiw   t5,t5,1
        sw      t5,-20(s0)      #i=i+1
        j       .L3
.L2:
        ld      t5,-40(s0)      #t5 为 A 数组的起始地址
        lw      t5,0(t5)        #t5 入栈 A[0]
        mv      t0,t5           #t0=A[0]
        ld      s0,56(sp)
        addi    sp,sp,64
        jr      ra
```

14.

```
        mv      a3,a0
        mv      a4,a1
        mv      a5,a2
        sw      a3,-20(s0)   #a0 空间
        sw      a4,-24(s0)   #a1 空间
        sw      a5,-28(s0)   #a2 空间
        lw      a5,-20(s0)
        mv      a4,a5
        lw      a5,-24(s0)
        bge     a5,a4,.L2    #对 a 和 b 进行大小比较，如果 a<=b，跳转到 L2
        lw      a5,-20(s0)
        mv      a4,a5
        lw      a5,-24(s0)
        addw    a5,a4,a5
        sw      a5,-28(s0)   #c=a+b
        j       .L3
.L2:
        lw      a5,-20(s0)
        mv      a4,a5
        lw      a5,-24(s0)
        subw    a5,a4,a5
        sw      a5,-28(s0)   #c=a-b
.L3:
        li      a5,0
        mv      a0,a5
        ld      s0,24(sp)
        addi    sp,sp,32
        jr      ra
```

15.

```
addi    sp,sp,-48
sd      s0,40(sp)
addi    s0,sp,48
sd      t0,-40(s0)
sd      t1,-48(s0)
lw      t5,-40(s0)    #t5=p
sw      t5,0(t5)      #*p=p[0]=p
li      t5,3          #t5=3
sd      t5,-48(s0)    #a=3
lw      t5,-40(s0)    #t5=p
addi    t5,t5,4       #t5=p+4(整型数组元素（int 型变量）的地址为四位字节，32 位变量)
li      t4,3          #a=3
sw      t4,0(t5)      #p[1]=3
addi    t5,t5+8       #t5=p+12
sw      t4,0(t5)      #p[3]=3
```

16.

```
addi    sp,sp,-48     #总体
sd      s0,40(sp)
addi    s0,sp,48
sd      t0,-40(s0)    #存放 a 指针地址
sd      t1,-48(s0)    #存放 b 指针地址
lw      t5,0(t0)      #t5=*a
sw      t5,-20(s0)          #为 temp 创造堆栈空间，同时将*a 的值存入为 temp 创造的堆栈空间
lw      t4,0(t1)      #t4=*b
ld      t5,-40(s0)    #t5=a
sw      t4,0(t5)      #*a=*b
lw      t4,-20(s0)    #t4=temp
sw      t4,0(a1)      #*b=temp
nop
ld      s0,40(sp)
```

```
addi     sp,sp,48
jr       ra                  #存放 a 指针地址
sd       t1,-48(s0)          #存放 b 指针地址
lw       t5,0(t0)            #t5=*a
sw       t5,-20(s0)          #temp 入栈，同时将*a 的值存入为 temp 创造的堆栈空间
lw       t4,0(t1)            #t4=*b
ld       t5,-40(s0)          #t5=a
sw       t4,0(t5)            #*a=t4=*b
```

17.

由上到下每行代码的意思依次为：

1. a0 寄存器中的值为 x0 寄存器的值
2. a1 寄存器中的值为 x0 寄存器的值+1
3. a1 寄存器中的值为 x0 寄存器的值+30
4. 如果 a0 寄存器中的值和 a2 寄存器中的值相等，则跳转到最后一条语句
5. a1 寄存器中的值向左移动一位，也就是说乘 2
6. a0 寄存器中的值+1
7. 跳转到第 4 条语句
8. 退出程序

该代码实现的功能是让 a0 寄存器中的值等于 x0 寄存器中的值+30，同时让 a1 寄存器中的值等于（x0 寄存器中的值+1)的 2^30 倍