2-3 1) nop: addi x0, x0, 0

2) ret: jalr x0, 0(x1)

3) call offset: auipc x1, offset[31:12] + offset[11].

jalr x1, offset[11:0](x1)

4) mv rd, rs: addi rd, rs, 0

5) rdcycle rd: csrrs rd, cycle, x0

6) sext.w rd, rs: addiw rd, rs, 0

2-7: 1) slti t3, t2, x0

slt t4, t0, t1

第一条指令判断 t_2 是否为正，第二条指令判断 t_0 与 t_1 的大小，当结果 t_3、t_4 不相等时，判断出现 overflow。

2) 都是无符号数时，只需要判断和是否小于其中某一个数。

add t0, t1, t2

bltu t0, t1, overflow.

3) x86 和 Intel 都使用 OF 标志位。当 ADD 指令导致了溢出时，OF 标志位会被设置为 1，否则设置为 0。然后程序通过判断 OF 标志位的值来决定是否进行异常处理。

2-8. 1)

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,rs1,rs2	x	0	2^L-1	x	-1	x

其中 L 为指令集的位数。

整型除法中除数为 0 并不会引起异常。

原因：如果使用异常，则这将是标准中唯一的算术异常，同时与软语言实现者需要与异常处理程序进行交互。另外，如果不用异常，只需要在每个除法指令处添加一条分支指令，这样一条通常不会执行的处理指令几乎不会增加运行的开销；而如上表所示的结果也是简单除法器实现得到的默认结果；相反，若添加了异常处理标准，往往会增加硬件设计的复杂性。

2) NV：无效操作。 DZ：除数为 0。 OF：溢出(上溢)

UF：溢出(下溢) NX：不精确。

fflags 被置位不会使处理器处于系统调用。

3) 对于 x86与ARM处理器，在除法操作时若除数为0，则会触发一个除法错误的异常。处理器停止当前指令的执行并跳转到异常处理程序。

2-12. 1) 机器模式。2) 机器模式。3) 机器模式 4) 管理员模式。5) 用户模式。

2-13.

```
        .text
        .globl  vecMul
        .type  vecMul, @function
vecMul:
        add  t3, X0, X0   # Set t3 to store the value of i
        addi  t4, X0, 100  # Maximum of i.
Loop:
        bge  t3, t4, endLoop #if i ≥ 100, end the loop.
        slli  t5, t3, 2.   # i * 4.
        add  t6, t5, t0   # t6 is the address of A[i].
        add  t7, t5, t1   # t7 is the address of B[i]
        lw   t7, 0(t7)   # Store the value of B[i] into t7.
        mul  t7, t7, t2   # store B[i]*C into t7.
```

```
        sw   t7, 0(t6)   # A[i] = B[i]*C
        addi t3, t3, 1    # i++
        j    Loop
    endLoop:
        lw   a0, t0       # return A[0]
```

2-14.
```
    bge  a1, a0, else   # if a<=b, go to else
    add  a2, a0, a1     # c = a+b
else:
    sub  a2, a0, a1     # c = a-b.
```

2-15.
```
sw   t0, 0(t0)   # p[0] = p
addi t1, x0, 3   # a = 3
sw   t1, 4(t0)   # p[1] = a
slli t2, t1, 2   # t2 = a*4
add  t2, t0, t2  # t2 = t2+p : t2 is the address of p[a].
sw   t1, 0(t2)   # p[a] = a.
```

2-16.
```
    .text
    .globl swap
    .type swap, @function
swap:
    lw   t2, 0(t0)   # t2 is tmp.
    lw   t3, 0(t1)   # t3 is *b
    sw   t3, 0(t0)   # *a = *b
    lw   t2, 0(t1)   # *b = tmp
    ret
```

2-17
```
    addi a0,x0,0
    addi a1,x0,1
    addi a2,x0,30
loop: beq a0,a2,done
    slli a1,a1,1
    addi a0,a0,1
    j loop
done: # exit code
```

⇒ a0=0, a1=1, a2=30

if (a0==a2) goto done.

a1 = a1/2

a0++

这段代码让a1左移，直到a0与a2相等.

因此实现的功能是：$a_1 *= 2^{30}$.

或 a_1 左移30位.