3.

(1) nop伪指令 ⇔ addi x0, x0, 0 空操作

(2) ret伪指令 ⇔ jalr x0, x1, 0 从子过程返回

(3) call offset伪指令 ⇔ auipc x6, offset[31:12]
jalr x1, x6, offset[11:0] 调用远处的子程序
其中auipc用于构建PC的相对地址(20位)与jalr(12位)组合形成32位PC相对地址调用

(4) mv rd, rs伪指令 ⇔ addi rd, rs, 0 复制寄存器

(5) rdcycle rd 伪指令 ⇔
读取cycle CSR低XLEN位，这个计数值默认硬[illegible]件线程从过去的[illegible]到开始执行此指令时钟周期计数值

(6) sext.w rd rs 伪指令 ⇔ addiw rd, rs, x0 符号扩展字

7.(1)

~~add t0, t1, t2~~
~~sub t3, t0, t1~~
~~addi t4, t3, 0~~
~~bne t3, t4, overflow~~

7.(1)
```
add  t0, t1, t2
slti t3, t2, 0
slt  t4, t0, t1
bne  t3, t4, overflow
```
$t_2<0 \Rightarrow t_3=1$
$t_0<t_1 \Rightarrow t_4=1$
若 $t_2>0$ 则为加溢出时有 $t_0<t_1$
若 $t_2<0$ 则为溢出时有 $t_0>t_1$

(2) ~~add t0, t1, t2~~
~~bge t1, t0, overflow~~
~~bgez t0, overflow~~

7.(2)
```
addu t0, t1, t2
bltu t0, t1, overflow
```

这里如果不溢出，则 $x+y \geq x$ ， $x+y \geq y$
若溢出，则相加得到的数为 $x+y-2^n$ ~~[illegible]~~ $x+y-2^n<x$
$x+y-2^n<y$
只需 bltu t0, t1, overflow 一条指令即可。

(3) ARM体系架构中，通过CPSR的状态寄存器反映当前指令的溢出状态。CPSR是条件码寄存器，在执行加法时，如果发生了溢出则CPSR中的标志位被设置为1，可以通过CPSR的标志位是否为1判断执行是否发生了溢出。

X86通过使用进位标志位(CF)和溢出标志位(OF)来检测是否溢出，如果加法溢出，则OF标志位被设置为1，通过这个判断是否溢出。

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 | $2^{XLEN}-1$ | X | -1 | X |

~~整型除法中除数为0不会引起RISC-V抛出异常~~

整型除法中除数为0会引起RISC-V抛出异常，这样做是为了防止系统受到损伤。

(2) NV：非法操作

DZ：除以0

OF：上溢

UF：下溢

NX：不精确。

~~除数为零~~ fflags被置位，会使处理器陷入系统调用，因为此时frm被设置为非法值，后续如使用了动态舍入模式加浮点操作都会触发引起一个非法指令的陷。

(3) ~~ARM在遇到除数为0的情况时~~

ARM和x86都有特定的标志来反映除数为0，会抛出异常

12.

(1) S (2) M (3) M (4) S (5) U

13.

```
        add   a3, x0, x0     # i=0, a3=i
        addi  a4, x0, 100    # a4=100
        mv    a0, t0         # A
        mv    a1, t1         # B
        mv    a2, t2         # C
loop:   bge   a3, a4, exit
        slli  a5, a3, 2      # i*4
        add   a5, a5, a0     # &(A+i)
        add   a6, a5, a1     # &(B+i)
        lw    a7, 0(a6)      # *(B+i)
        mul   a8, a7, a2     # a8=B[i]*C
        sw    a8, 0(a5)      # 乘积A[i]值存到&(A+i)
        addi  a3, a3, 1      # ++i
        j     loop
exit:   lw    a0, 0(a0)      # A[0]放入a0寄存器返回.
        ret
```

14.

```
        bge   a1, a0, part1  # a≤b 跳转.
        add   a2, a0, a1     # c=a+b
part1:  sub   a2, a0, a1     # c=a-b.
```

15.
```
add  a1, x0, x0    # i=0
slli a2, a1, 2     # i*4
add  a2, a2, t0    # &(P+i) 这里是&P[0]
sw   t0, 0(a2)     # P[0]=P
addi a1, a1, 1     # i=1
slli a2, a1, 2     # i*4
add  a2, a2, t0    # &(P+i) 这里是&P[1]
addi t1, x0, 3     # a=3
sw   t1, 0(a2)     # P[1]=a
sll  a2, t1, 2     # a*4
add  a2, a2, t0    # &(P+i) 这里是&P[a]
sw   t1, 0(a2)     # P[a]=a
```

16.
```
lw   a0, 0(t0)     # *a
lw   a1, 0(t1)     # *b
add  a2, x0, a0    # tmp=*a
add  a0, x0, a1    # *a=*b
add  a1, x0, a2    # *b=tmp
ret
```

8 17. 其对应的C代码为：
```
for(i=0; i<30; i++)
    a1 = a1*2;
```
其功能为 计算 $2^{30}$，并将其存放到 $a_1$ 寄存器中.