第五次作业：

3. 1) nop： addi x0、x0、0，　无操作 no operation.

2) ret： jalr x0、x1、0　子程序返回指令 Return from subroutine.

3) call offset： auipc x6，offset[31:12]　目标为4KB-4GB空间的远程函数 call far-away subroutine

jalr x1，x6，offset[11:0]

4) mv rd，rs： addi rd，rs，0　数据传送指令 copy register

5) rdcycle rd： csrrs rd，cycle，x0.　周期数读取指令 read cycle counter

6) sext.w rd，rs： addiw rd，rs，0　符号位扩展指令 sign extend word

7. 1). t0和t1都是有符号数时：(t2<0)&&(t0>=t1)||(t2>=0)&&(t0<t1)

```
addi  t0、t1、t2.
slti  t3、t2、0
slti  t4、t0、t1
bne   t3、t4、overflow
```

slti 有符号立即数比较小于置位指令.

slti rd、rs1、imm12

if (rs1 < sign_extend(imm12))

rd ← 1

else

rd ← 0

2) add t0，t1，t2.

bltu t0，t1，overflow

bltu 无符号小于分支指令

bltu rs1、rs2、label

if (rs1 < rs2)

next pc = current pc + sign_extend(imm12<<1)

3). ARM架构中，加法溢出检测通过 Condition Code 寄存器的标志位来实现的。V标志位表示操作结果是否产生了溢出，C标志位表示是否产生了进位或借位。

x86架构中加法通过进位标志位(Carry Flag)来检测，并且可以用JC(Jump if Carry)指令来检测进位标志符是否被设置。

5. 8. 1). Op = DIVU时: 0xffffffffffffffff 无符号除法指令

Op = REMU时: x 无符号取余指令

Op = DIV时 : 0xffffffffffffffff 有符号除法指令

Op = REM时 : x 有符号取余指令

6. 2) NV: Invalid Operation

DZ: Divide by Zero

OF : Overflow

UF: Underflow

NX: Inexact

处理器不会陷入系统调用.

3). x86架构通过异常处理机制来实现对除数0处理. "溢出异常" CPU将控制权转移到一个预定义的中断处理程序中、保存当前CPU各寄存器的值和下条指令的地址、然后执行"溢出异常处理程序、之后恢复.

ARM中会产生"零除异常"(Divide by Zero Exception)然后将处理器的控制权转给操作系统的异常处理程序.

11. 1) jal ra、0x88 偏移量寻址.

2) jalr x0, ra、0 寄存器间接寻址

3) addi a0、a1、4 立即数寻址.

4). mul a0、a1、a2 寄存器直接寻址

5) ld a4, 16(sp). 偏移量寻址.

12. 1) Linux Kernel → Supervisor mode

2) BootROM → Machine mode. 系统启动时最早执行的代码

3) BootLoader → Supervisor mode

4) USB Driver → Supervisor mode

5) Vim → User mode

13.
```
# Assume t0 holds pointer to A, t1 holds pointer to B.
    add  a3, x0, x0   # a3 = i
    addi a4, x0, 100
    lw   a7, 0(t2)    # a7 = C
Loop: bge  a3, a4, end
    sll  a5, a3, 2    # i*4
    add  a6, t0, a5   # a6 holds the pointer to (A+i)
    add  a5, t1, a5
    lw   a5, 0(a5)    # *(B+i)
    mul  a5, a5, a7   # B[i]*C
    sw   a5, 0(a6)
    addi a3, a3, 1    # i++
    j    Loop
end: lw  a0, 0(t0)    # A[0]
```

14.
```
    mv   a4, a0       # a4 = a
    mv   a5, a1       # a5 = b
    blt  a5, a4, part1   (有符号小于分支指令)
    sub  a5, x0, a5   # a5 = -b
    add  a2, a4, a5   # c = a-b
part1:
    add  a2, a4, a5   # c = a+b
```

16.
```
    lw   a3, 0(t0)    # a3 = *a
    lw   a4, 0(t1)    # a4 = *b
    add  a5, x0, x0   # a5 = tmp = 0
    mv   a5, a3       # tmp = *a
    mv   a4, a3       # *a = *b
    mv   a4, a5       # *b = tmp
    lw   a3, 0(t0)
    lw   a4, 0(t1)
```

15.
```
    sw   t0, 0(t0)    # p[0] = p
    addi t1, x0, 3    # a = 3
    addi a3, t0, 4    # a3存p[1]地址
    sw   t1, 0(a3)    # p[1] = a
    addi a4, x0, 4    # a4 = 4
    mul  a5, t1, a4   # a5 = 4a
    add  a3, t0, a5   # a3存p[a]地址
    sw   t1, 0(a3)    # p[a] = a
```

17.

通过比较a0和a2寄存器中值的大小，实现2的乘方运算。