2-3

(1) nop → addi x0, x0, 0

(2) ret → jalr x0, 0(x1)

(3) call offset → auipc x6, offset[31:12]
jalr x1, x6, offset[11:0]

(4) mv rd, rs → addi rd, rs, 0

(5) rdcyde rd → csrrs rd, cycle, x0

(6) sext.w rd, rs → addiw rd, rs, 0

2-7

(1) addi t0, t1, t2
sub t3, t0, t1
mv t4, t2
bne t3, t4, overflow

(2) 对于无符号数加法：
addu t0, t1, t2
bltu t0, t1, overflow

(3). ARM指令集通过 CPSR 状态寄存器表示溢出：
无符号数：加法溢出时(进位)将 C 寄存器置1，减法溢出时(借位)将 C 寄存器置0。
有符号数：有符号数加减法中，若发生溢出，将V寄存器置1
x86 通过 CF 和 OF 寄存器来检测溢出：
无符号数：无符号数加减法出现溢出时，将CF寄存器置1
有符号数：有符号数加减法出现溢出时，将OF寄存器置1

2-8

(1)

指令	rs1	rs2	DIVU	REMU	DIV	REM
Op rd,rs1,rs2	x	0	2^L-1	x	-1	x

除数为0时不会抛出异常，而是产生某个特殊的默认值，同时设置共享状态寄存器的状态位，因为这样可以大幅简化处理流水线的硬件实现。

(2) fflags各位的含义：

NV : 无效操作

DZ : 除以零

OF : 溢出

UF : 下溢

NX : 不精确

fflags被置位不会使处理器陷入系统调用。

(3) x86架构：cpu通过8位的中断类型码，通过中断向量表找到对应的中断处理程序的入口地址，随即控制权交由操作系统内核进行故障处理。

ARM架构：同样会抛出一个异常，交由异常处理程序处理。

2-12.

(1) Linux Kernel　　M模式

(2) Boot ROM　　M模式

(3) Boot Loader　　S模式

(4) USB Driver　　S模式

(5) vim　　U模式

2-13.

```
start :  addi   sp, sp, -48
         sw     s0, 44(sp)
         addi   s0, sp, 48
         sw     t0, -36(s0)   # A
         sw     t1, -40(s0)   # B
         sw     t2, -44(s0)   # C
part1 :  sw     zero, -20(s0) # i
         j      part4
part2 :  lw     a5, -20(s0)   # a5 : i
         slli   a5, a5, 2     #左移2位，即给i乘4，因为数组中一个int占4字节
         lw     a4, -40(s0)   # a4 : B
         add    a4, a4, a5
         lw     a6, 0(a4)     # a6 : B[i]
         lw     a3, -36(s0)   # a3 : A
         add    a3, a3, a5
         lw     a7, -44(s0)   # a7 : C
```

```
        mul   a6, a6, a7    # B[i] * C
        sw    a6, 0(a3)     # 存入A[i]
part3:  lw    a5, -20(s0)
        addi  a5, a5, 1     # i++
        sw    a5, -20(s0)
part4:  lw    a5, -20(s0)
        li    a4, 99
        ble   a5, a4, part2 # i<=99 开循环
.end:   lw    a5, -36(s0)
        lw    a5, 0(a5)     # return A[0]
        mv    a0, a5
        lw    s0, 44(sp)
        addi  sp, sp, 48
        jr    ra
```

2.14

```
start:  addi  sp, sp, -32
        sw    s0, 24(sp)
        addi  s0, sp, 32
        sw    a0, -20(s0)   # a
        sw    a1, -24(s0)   # b
        sw    a2, -28(s0)   # c
part1:  lw    a5, -20(s0)
        lw    a4, -24(s0)
```

```
        ble     a5, a4, part2   # a<=b 时跳转到 else
        add     a3, a5, a4
        sw      a3, -28(s0)
        j       end
part2:  sub     a3, a5, a4
        sw      a3, -28(s0)
end:    lw      s0, 24(sp)
        addi    sp, sp, 32
        jr      ra
```

2-15.

```
start:  addi    sp, sp, -32
        sw      s0, 24(sp)
        addi    s0, sp, 32
        sw      t0, -20(s0)     # p
        sw      t1, -24(s0)     # a
part1:  lw      a5, -20(s0)     # a5: p
        addi    a4, a5, 0       # a4: &p[0]
        sw      a5, 0(a4)       # p[0] = p
part2:  lw      a5, -24(s0)     # a5: a
        li      a4, 3
        sw      a4, -24(s0)     # a = 3
part3:  lw      a5, -20(s0)     # a5: p
        lw      a4, -24(s0)     # a4: a
```

```
        addi    a3, a5, 4       # a3:&p[1]
        sw      a4, 0(a3)       # p[1]=a
part4:  lw      a5, -20(s0)     # a5: p
        lw      a4, -24(s0)     # a4: a
        slli    a6, a4, 2       # a6: a×4
        add     a7, a5, a6      # a7: &p[a]
        sw      a4, 0(a7)       # p[a]=a
end:    lw      s0, 24(sp)
        addi    sp, sp, 32
        jr      ra
```

2-16

```
start   addi    sp, sp, -32
        sw      s0, 24(sp)
        addi    s0, sp, 32
        sw      t0, -20(s0)     # a
        sw      t1, -24(s0)     # b
part1:  lw      a5, -20(s0)
        lw      a4, 0(a5)       # a4: tmp = *a
        sw      a4, -28(s0)
part2:  lw      a5, -24(s0)     # a5: b
        lw      a4, 0(a5)       # a4: *b
        lw      a3, -20(s0)     # a3: a
        sw      a4, 0(a3)       # *a = *b
```

```
part3:  lw   a5, -24(s0)   # a5 : b
        lw   a4, -28(s0)   # a4 : tmp
        sw   a4, 0(a5)     # *b = tmp
end:    lw   s0, 24(sp)
        addi sp, sp, 32
        jr   ra            # return
```

2-17

代码功能：让 a1 的值乘以 2^{30}，即循环30次 a1 = a1 × 2

转换为 C 语言：

```
int a0 = 0
int a1 = 1
int a2 = 30
for ( ; a0 < a2 ; a0++ )
    a1 = a1 * 2
```