1.加减乘除

判断有符号数溢出

Detect Overflow for Signed Addition

- Ideas:（t0=t1+t2，3 signs: s1、s2、s0）
  - 1.The different signs (s1<>s2), means no overflow;
  - 2.The same signs (s1=s2), means possible overflow:

    If s0=s1, no overflow;

    If s0<>s1, overflow and exception.

```
addu $t0, $t1, $t2 # $t0 = sum, but don't trap
xor  $t3, $t1, $t2 # Check if signs differ
slt  $t3, $t3, $zero # $t3 = 1 if signs differ
bne  $t3, $zero, No_overflow # $t1, $t2 signs ≠,
                             # so no overflow
xor $t3, $t0, $t1 # signs =; sign of sum match too?
                  # $t3 negative if sum sign different
slt $t3, $t3, $zero # $t3 = 1 if sum sign different
bne $t3, $zero, Overflow # All 3 signs ≠; goto overflow
```

如果 s1 和 s2 异号肯定不溢出，如果同号则比较 s0 与 s1 的符号，不同就溢出

比较符号可以用异或，然后与 0 进行比较

判断无符号数溢出

Detect Overflow for Unsigned Addition

- Ideas:（t0=t1+t2）
  - if t1+t2>2^{32}-1, (t2>2^{32}-1-t1),overflow and exception

    2^{32}-1-t1 is ones complement / inversion of t1.

```
addu $t0, $t1, $t2      # $t0 = sum
nor $t3, $t1, $zero     # $t3 = NOT $t1
                        # (2's comp - 1: 2^32 - $t1 - 1)
sltu $t3, $t3, $t2      # (2^32 - $t1 - 1) < $t2
                        # ⇒ 2^32 - 1 < $t1 + $t2
bne $t3,$zero,Overflow # if(2^32-1<$t1+$t2) goto overflow
```

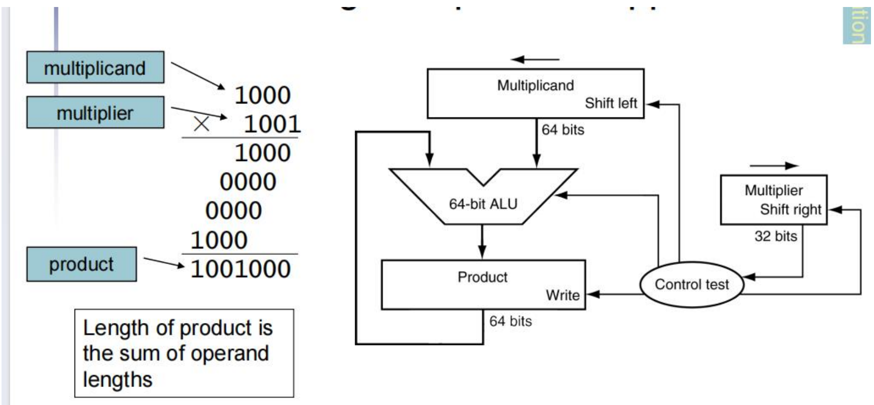
当 t1+t2>2 的 32 次方-1 就溢出，所以将 t2 与 t1 的补码进行比较，大于则溢出

```
nor $t3, $t1, $zero     # $t3 = NOT $t1
                        # (2's comp - 1: 2^32 - $t1 - 1)
```

由于没有直接取非的操作，于是用或非 0 来取非

乘法

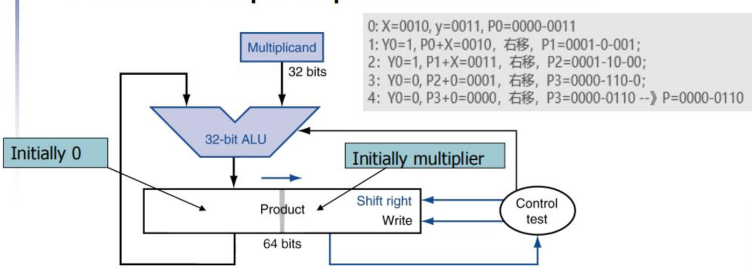
最初设计



对应的运算规则：因为全是二进制，那么每次取乘数的 LSB（只能是 0 或 1），那么被乘数也只能是 0 或者本身，变化的地方在于乘数的每个位对应一个权值。因此我们每次将被乘数左移一位（乘 2，体现权值的变化），乘数右移一位，方便取到想要的 LSB，不断地加到 product，就通过加法实现了乘法

优化设计 **乘数和积共用一个硬件**

- Perform steps in parallel: add/shift



一开始乘数放到 Product 的低 32 位，高 32 位为 0，接下来还是取乘数的 LSB，判断是 0 还是 1，被乘数加到 product 的高 32 位，然后每加一次 product 右移一位（有两个作用，一是改变乘数的 LSB，第二是在最后移动了 32 位，得到正确的结果）

乘法指令 HI 就是 high，LO 就是 low 乘法的结果存在这两个寄存器

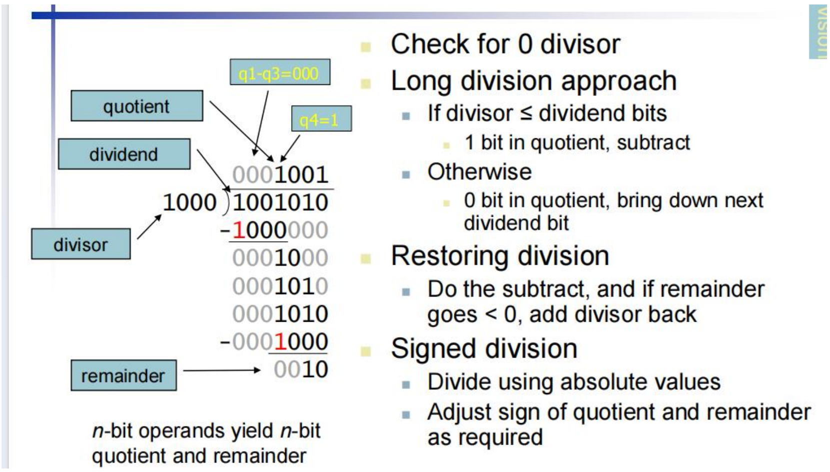
Two 32-bit registers for product

- HI: most-significant 32 bits
- LO: least-significant 32-bits

Instructions

- `mult rs, rt / multu rs, rt`
  - 64-bit product in HI/LO
- `mfhi rd / mflo rd`
  - Move from HI/LO to rd
  - Can test HI value to see if product overflows 32 bits
- `mul rd, rs, rt` (pseudo instuction)
  - Least-significant 32 bits of product –> rd

除法



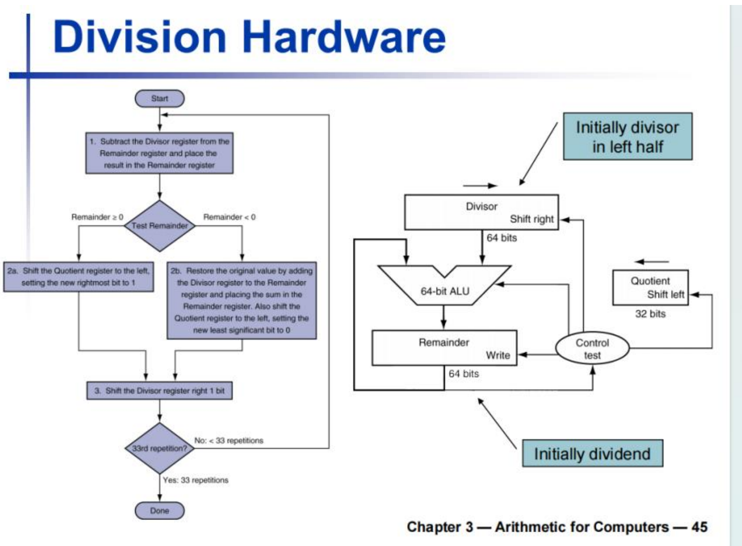
1.检查除数是否为 0

2.除法的本质是减法，比较当前的被除数与除数，如果被除数大于除数，则被除数减去除数，商上 1，否则移动到被除数下一位，上 0

3.如果余数小于 0，则加上除数

4.做除法时都用绝对值，后面记得补符号

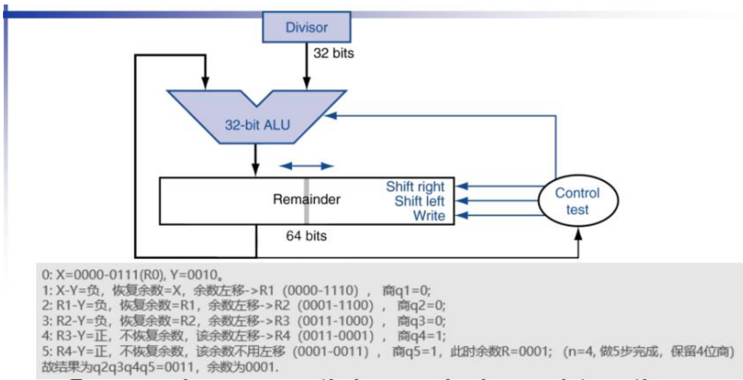
最初设计



Divisor 的左半边初始化为除数（相当于除数左移了 32 位，目的是为了从被除数最高位与其进行比较），Remainder 被初始化位被除数，Quotient 一开始就是 0

每次用被除数减去 divisor，Quotient 左移一位，如果 remainder 里面最高位为 1（意味着小于 0）则让 remainder 加上 divisor 进行复原，商上 0，否则商上 1，这一过程结束后 divisor 右移一位（为了与被除数次高位进行比较）

优化设计 **余数和商共用一个硬件**

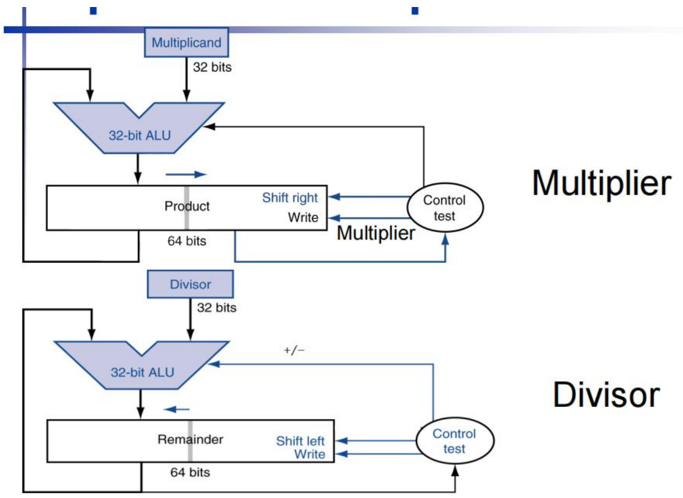


Remainder 初始化为被除数，因为我们要移动的是 remainder 而不移动 divisor,所以每移动一次，remainder 的低一位就没有用了，商一定是 32 位，所以将商存放在 remainder 低 32 位。左移被除数的思想也是让高 32 位与除数进行比较，也就是一直比较 remainder 的高 32 位与除数，这和最初设计的思想一样，只不过我们实现的方式不同

除法指令

- Use HI/LO registers for result
  - HI: 32-bit remainder
  - LO: 32-bit quotient
- Instructions
  - `div rs, rt` / `divu rs, rt`
  - No overflow or divide-by-0 checking
    - Software must perform checks if required
  - Use `mfhi`, `mflo` to access result

优化设计的乘法与除法硬件



可以发现特点就是不移位的是要一直保持“相同”的数，比如**被乘数**，因为每次加到 Product 的高 32 位，被乘数一直不变，所以它不进行移位。又比如**除数**，因为每次 remainder 高 32 位都是要减去除数，所以除数不进行移位

最初设计是根据乘法或者除法的最基本原理设计的，所以没有设计对硬件使用的优化，就是缺什么我就补什么硬件，优化后将硬件的使用效率变高

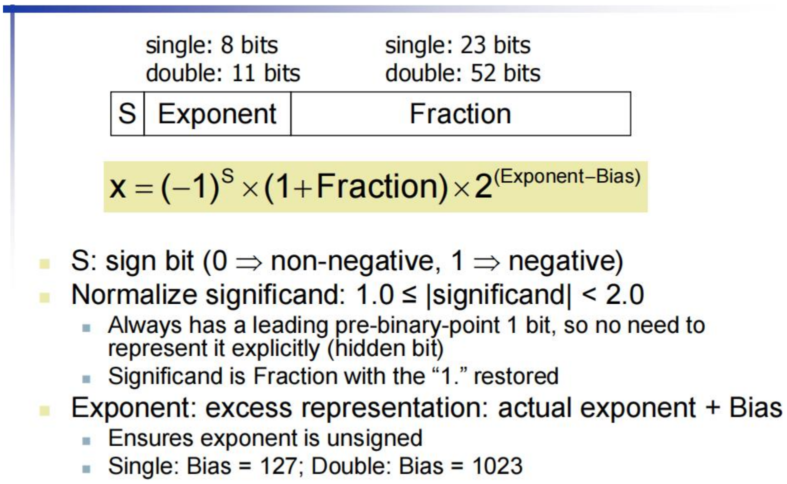
记忆这个也不需要死记硬背，牢牢抓住乘法与除法的本质，乘法的本质是加法，除法的本质是减法，进行联想即可

考试中硬件这一章的重点一般是浮点数，这些硬件不是重点

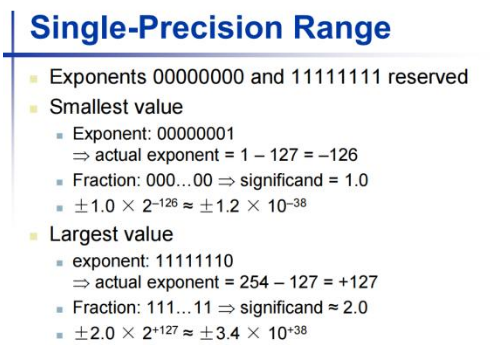
2.浮点数

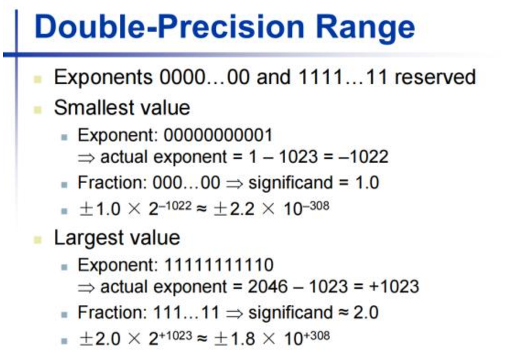
浮点数格式

符号位 指数(exponent) 尾数(fraction) 偏阶(bias)单精度浮点数偏阶为 127，双精度浮点数了解即可，考试不太好考



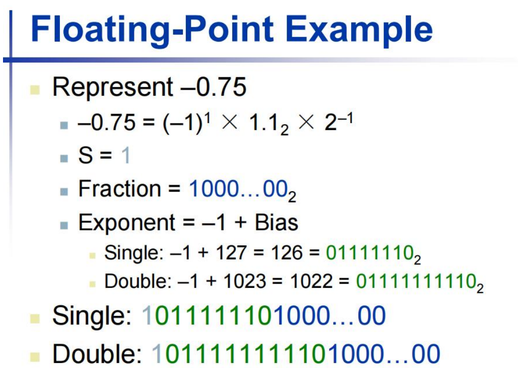
范围

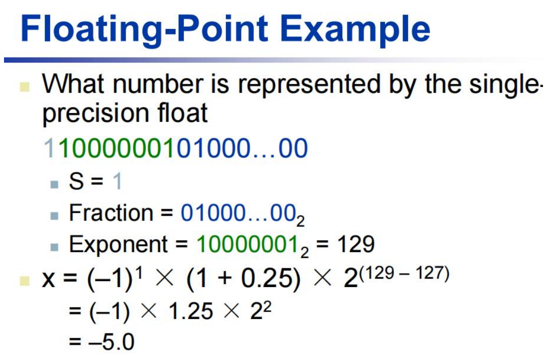




00000000 和 11111111 不用，因为有特殊用途

例子





将一个十进制数表示为 IEEE754 浮点数

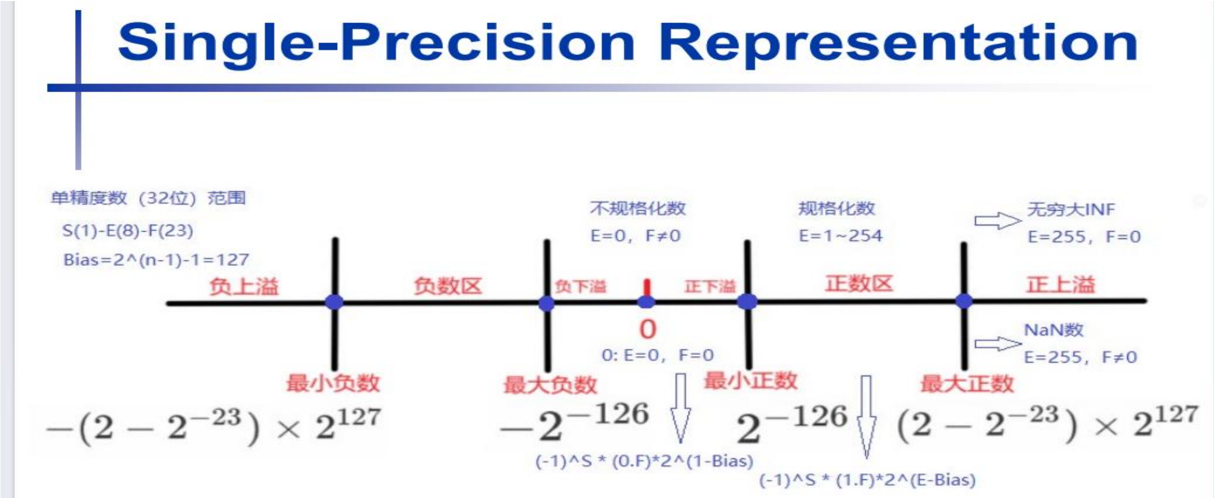
1.将科学表示法表示的数转换为一般十进制表示 2.十进制数转换为二进制表示（分为整数部分和小数部分，小数部分记得是不断乘 2） 3.移位将二进制数表示为

$$x = (-1)^S \times (1+\text{Fraction}) \times 2^{(\text{Exponent}-\text{Bias})}$$

这种形式 IEEE754

上下溢出

00000000 和 11111111 的特殊用途



浮点数的加法

- Now consider a 4-digit binary example
  - $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ (0.5 + –0.4375)
- 1. Align binary points
  - Shift number with smaller exponent
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$
- 2. Add significands
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2{-1} = 0.001_2 \times 2^{-1}$
- 3. Normalize result & check for over/underflow
  - $1.000_2 \times 2^{-4}$, with no over/underflow
- 4. Round and renormalize if necessary
  - $1.000_2 \times 2^{-4}$ (no change) = 0.0625

1.对齐，将指数小的数与指数更大的数对齐

2.将有效数相加

3.标准化并且检查有无溢出

4.进行舍入（如果精度够，其实有些可以不用舍入，直接保留即可）

浮点数的乘法（考的也很少）

- Now consider a 4-digit binary example
  - $1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$ (0.5 × –0.4375)
- 1. Add exponents
  - Unbiased: –1 + –2 = –3
  - Biased: (–1 + 127) + (–2 + 127) = –3 + 254 – 127 = –3 + 127
- 2. Multiply significands
  - $1.000_2 \times 1.110_2 = 1.110_2 \Rightarrow 1.110_2 \times 2^{-3}$
- 3. Normalize result & check for over/underflow
  - $1.110_2 \times 2^{-3}$ (no change) with no over/underflow
- 4. Round and renormalize if necessary
  - $1.110_2 \times 2^{-3}$ (no change)
- 5. Determine sign: +ve × –ve ⇒ –ve
  - $-1.110_2 \times 2^{-3} = -0.21875$

1.指数相加

2.有效数相乘

3.检查有无溢出并且标准化

4.进行舍入

5.确定符号

浮点数的指令操作（看看就行，不可能考）

硬件 coprocessor 1（协处理器 1）

- Separate FP registers
  - 32 single-precision: $f0, $f1, … $f31
  - Paired for double-precision: $f0/$f1, $f2/$f3, …
    - Release 2 of MIPs ISA supports 32 × 64-bit FP reg's
- FP instructions operate only on FP registers
  - Programs generally don't do integer ops on FP data, or vice versa
  - More registers with minimal code-size impact
- FP load and store instructions
  - `lwc1`, `ldc1`, `swc1`, `sdc1`
    - e.g., `ldc1 $f8, 32($sp)`

double 类型的就用两个 32 位寄存器，fp 类型的寄存器只做浮点数的运算，lw 和 ld 后面要加上 c1 表示使用协处理器

- Single-precision arithmetic
  - `add.s`, `sub.s`, `mul.s`, div.s
    - e.g., `add.s $f0, $f1, $f6`
- Double-precision arithmetic
  - `add.d`, `sub.d`, `mul.d`, `div.d`
    - e.g., `mul.d $f4, $f4, $f6`
    （`$f4/$f5*$f6/$f7→$f4/$f5` ）
- Single- and double-precision comparison
  - `c.xx.s`, `c.xx.d` (*xx* is eq, neq, `lt`, `le`, gt，ge)
  - Sets or clears FP condition-code bit
    - e.g. `c.lt.s $f3, $f4`
- Branch on FP condition code true or false
  - `bc1t`, `bc1f`
    - e.g., `bc1t TargetLabel`（`if` cond=1 then branch）

s 表示 single，d 表示 double bc1t 的 t 表示 true

浮点数精确度

- 保护位：在浮点数中间计算中，在右边多保留的两位中的首位；用于提高舍入精度。
- 舍入位：在浮点数中间计算中，在右边多保留的两位中的第二位；使浮点中间结果满足浮点格式，得到最接近的数。

舍入听起来很简单，但它需要硬件支持在计算中产生更多的有效位。在前面的例子中，我们对中间结果占有多少位未做介绍，但很明显的是，如果每个中间结果都截短成准确的位数，就没法做舍入了。IEEE 754 因此在中间计算中，右边总是多保留两位，分别称为**保护位**（guard）和**舍入位**（round）。我们用一个十进制的例子来说明它们的作用。

有非 0 的数据粘贴位就为 1

粘贴位：同保护位和舍入位一样用于舍入的位，当舍入位右边有非零的数据时将其置 1。

中间结果，然后执行舍入那样。为了支持这个目标并向最靠近的偶数舍入，IEEE 754 标准在保护位和舍入位之后还有一位粘贴位（sticky bit）；当舍入位右边的数非零时将它置 1。粘贴位可以让计算机在舍入时，能够区分 $0.50\cdots00_{10}$ 和 $0.50\cdots01_{10}$。

粘贴位可能被置 1，例如，在加法中，当较小数右移时就可能这样。假设在前面的例子里我们将 $5.01_{10}\times10^{-1}$ 和 $2.34_{10}\times10^{2}$ 相加。即使有保护位和舍入位，我们将 0.0050 和 2.34 相加，得到 2.3450。粘贴位会被置 1，因为右边是非零的。假设没有粘贴位来记住是否有 1 被移走，我们会假设这个数等于 2.345000…00，然后向最靠近的偶数舍入得到 2.34。使用粘贴位记住这个数是大于 2.345000…00 的，我们舍入后会得到 2.35。

g 保护位，r 舍入位,s 粘贴位

进位的规则是看 grs 三位有没有过半

当刚好是一半的时候

时间里使用向上舍入，另一半时间里使用向下舍入。IEEE 754 处理这种中间情况的方法是如果最后一位是奇数，就加 1；如果是偶数，则截去。这种方法总是在中间情况下将最低位设为 0，正如舍入模式的名称。这种模式是用得最多的，而且是唯一被 Java 支持的模式。

奇进 1，偶舍去

几个例题

1.饱和计算

精解 饱和（saturating）操作是通用微处理器中一个不常出现的特性。饱和意味着当计算结果溢出时，结果被设置为最大的正数或者最小的负数，而不像二进制补码运算那样采用取模操作来获得结果。饱和操作一般更适合多媒体操作。例如，当不断旋转收音机音量的旋钮时，起初声音逐渐增大，但如果大到一定值后声音突然变小，那么这样的收音机设计是不合理的。然而，对一台有饱和操作的收音机，当向最大值方向旋转音量旋钮到一定程度后，即使再旋转，音量也只会停在最大值上。标准指令集上媒体扩展通常提供饱和算法。

3.11 [10] <3.2>假定 151 和 214 是无符号 8 位十进制数。使用饱和算术计算 151＋214。结果必须使用十进制。

3.11 8位无符号数范围：-256-+255，饱和结果为255

2.用 IEEE754 表示浮点数（重点）

标准。

3.23 [10] <3.5>写出十进制数 63.25 的二进制表达。设采用 IEEE 754 单精度格式。

3.23

63.25=111111.01 * 2^0

=1.1111101 *2^5=1.1111101 *$2^{132-127}$

Sign=0，exp= 127+5=132

Final bit pattern: 0　1000 0100　1111 1010 0000 0000 0000 000

=0x427D0000

3.其他浮点数表示形式（信息题，要理解题目）

这里余-16 不好理解，我们主要想知道 bias 是多少，可以类比单精度浮点数 单精度浮点数指数有 8 位，bias 为 127（2 的 7 次方-1），因此该 bias 为（2 的 4 次方-1,也就是 15）

3.27 [20] <3.5>IEEE 754-2008 包含一种“半精度”格式，只有 16 位宽。最左边仍为符号位，指数有 5 位宽且以余 –16（excess –16）的形式存储，尾数有 10 位宽。具有隐含 1。写出 -1.5625×10^{-1} 的这种格式的二进制位模式。同 IEEE 754 标准的单精度比较，评估这个 16 位位模式的范围和精确度。

3.27 补充

$-1.5625*10^{-1}=-1.01*2^{-3}$

Exponent=-3+15=12, fraction=-.0100000000

结果: 1 01100 0100000000

4.浮点数精确度

3.29 [20] <3.5>手算 2.6125×10^1 和 4.150390625×10^{-1} 的和，设 A 和 B 以练习题 3.27 中提到的 16 位半精度格式存储。假设有 1 位保护位、1 位舍入位和 1 位粘贴位，并采用向最靠近的偶数舍入的模式。给出所有步骤。

3.29 *向最近的偶数舍入：当GR位=10且S位=0时（正好一半），最低有效位如果为偶数，则舍；最低有效位如果为奇数，则入。当GR位=10且S位=1时（超过一半），则入。

$2.6125\times10^1 + 4.150390625\times10^{-1}$

$2.6125\times10^1 = 26.125 = 11010.001 = 1.1010001000\times2^4$

$4.150390625\times10^{-1} = .4150390625 = .011010100111 = 1.1010100111\times2^{-2}$

Shift binary point 6 to the left to align exponents,

```
1.1010001000 00
0.0000011010 10 0111 (Guard 5 1, Round 5 0,
Sticky 5 1)
-------------------
1.1010100010 10
```

In this case the extra bit (G,R,S) is more than half of the least significant bit (0).

Thus, the value is rounded up.

$1.1010100011\times2^4 = 11010.100011\times2^0 = 26.546875 = 2.6546875\times10^1$