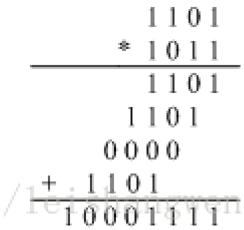
date created: 2022-10-05 21:23
date updated: 2022-10-06 00:06

PART-I

[2-1] 分析串行移位乘法器原理，说明操作数位宽和乘法器时延的关系。

- 串行移位乘法器原理
  - 移位寄存器初始存放被乘数，而后与乘数的第一个bit与运算，之后在移位重复并相加结果，过程如又图。


  - 二进制中与单个bit的与运算等价于相乘，故相当于把$1101 * 1101$拆为了$1101 * 0001 + 1101 * 0000 + 1101 * 0100 + 1101 * 1000$，类别十位数乘法，结果相加即为二者乘积。
- 关系
  - 部分乘积项数目决定了求和运算的次数，直接影响乘法器的速度。
  - 阵列式乘法器延时与位宽成线性关系

[2-3] 分析符号扩展（sign-extended）和有符号数表示方法的关系。

- 符号扩展用于保护有符号数的符号位，会根据有符号数最高位的不同选择使用0还是1进行扩展。
  - 8-bit数值“0000 1010”扩展至16-bit“0000 0000 0000 1010 ”
  - 8-bit数值“1000 1010”扩展至16-bit“1 1 1 1 1 1 1 1 1000 1010”

[2-6] 定点小数可以表示的精度与浮点数的最小精度单位（Unit of Least Precision）有何区别?

- 定点小数小数点的位置固定在最高有效数位之前，符号位之后，所以精度只与位数有关。
- 浮点数的最小精度单位ULP是指相邻的两个浮点数之间的距离，例如0xcccccccc与0xcccccccd间的距离就是对应十进制的ULP，又因为浮点数以指数形式表示，所以ULP不是线性变换的，因此不同数的ULP也会不同，例如用python中的math.ulp输出0.1与0.2的ULP会发现二者是不同的， 且一般比定点小数间距大。

```
>>> math.ulp(0.1)
        1.3877787807814457e-17
>>> math.ulp(0.2)
        2.7755575615628914e-17
>>> 2**(-63)
        1.0842021724855044e-19
```

- 即定点小数的精度不随小数本身变换，而ULP却与要表示的数有关。且ULP一般比定点小数间距大，精度更低。

[2-7] 举例说明什么是二进制浮点数的规格化（Normal）。

- 对定点小数而言，规格化是指把小数点移动到最左侧的数位，并且修改指针进行补偿。例如 1101.011 变成 $1.101011 * 2^3$
- 对浮点数而言规格化是针对浮点数的尾数的，即当尾数M用二进制表示时，应满足 $1/2 <= M < 1$，也就是说对于正数来说，有M = 00.1XXXXXXX 对于负数来说，有M = 11.0XXXXXXX 的形式，则可以判断其为规格化的数。
  - e.g.当得到的运算结果为 X = 0011,11.1001 时，显然不符合规格化的要求，需向左规格化——尾数左移一位，阶码减一， X = 0010,11.0010 ，达到要求。

[2-10] 非规格化（SubNormal）的二进制浮点数在规格化（Normal）二进制浮点数基础上新增的数的表示范围是?

- 以尾码m位，阶码n位的无符号浮点数为例：
  - 非规格化数的取值范围为$2^{-m} \times 2^{-(2^n-1)} < x < (1-2^{-m}) \times 2^{(2^n-1)}$
  - 规格化数的取值范围为$2^{-1} \times 2^{-(2^n-1)} < x < (1-2^{-m}) \times 2^{(2^n-1)}$
- 由此可见二者最大值没有差别，但非规格化数最小值要更接近0.
- 有符号数依0对称即可

[2-11] 各举一个非规格化（SubNormal）的二进制单精度浮点数在规格化（Normal）二进制单精度浮点数的例子，并给出出其十进制数值的结果（可以仅写表达式）。

类型	符号位	阶码	尾码	十进制
非规格化	0	1000 0101	0.00 1100 0000 0000 0000 0000 0000	0.1875x2^5=6
规格化	0	1000 0010	1.11 0000 0000 0000 0000 0000 0000	1.75x2^2 = 6

[2-12] 为什么IEEE-754标准中浮点数的阶码不需要符号位?

- 因为IEEE-754标准中从二进制数换算到浮点数的公式为$(-1)^S \times 2^{E-127} \times (1+F)$所以阶码E取[1, 254]时就可以表示原本的[-126, 127]，省去了符号位。

[2-15] 软件代码来实现超越函数计算有哪些常用的思路?

- 查表法
- 级数近似
  - 把超越函数化为相近的非超越函数
- CORDIC(COordinate Rotation DIgital Computer)
  - 通过移位和加减运算，能递归计算常用函数值
  - J. Walther在1974年用它研究了一种能计算出多种超越函 数的统一算法

[2-19] 分析CORDIC算法计算正弦函数的误差来源，并解释通过什么措施可以减小误差?

- 把γ_i近似为$arctan(2^{-\gamma_i})$时会产生误差

- 可以通过减小γ_i的值来减小误差，代价是增加计算次数