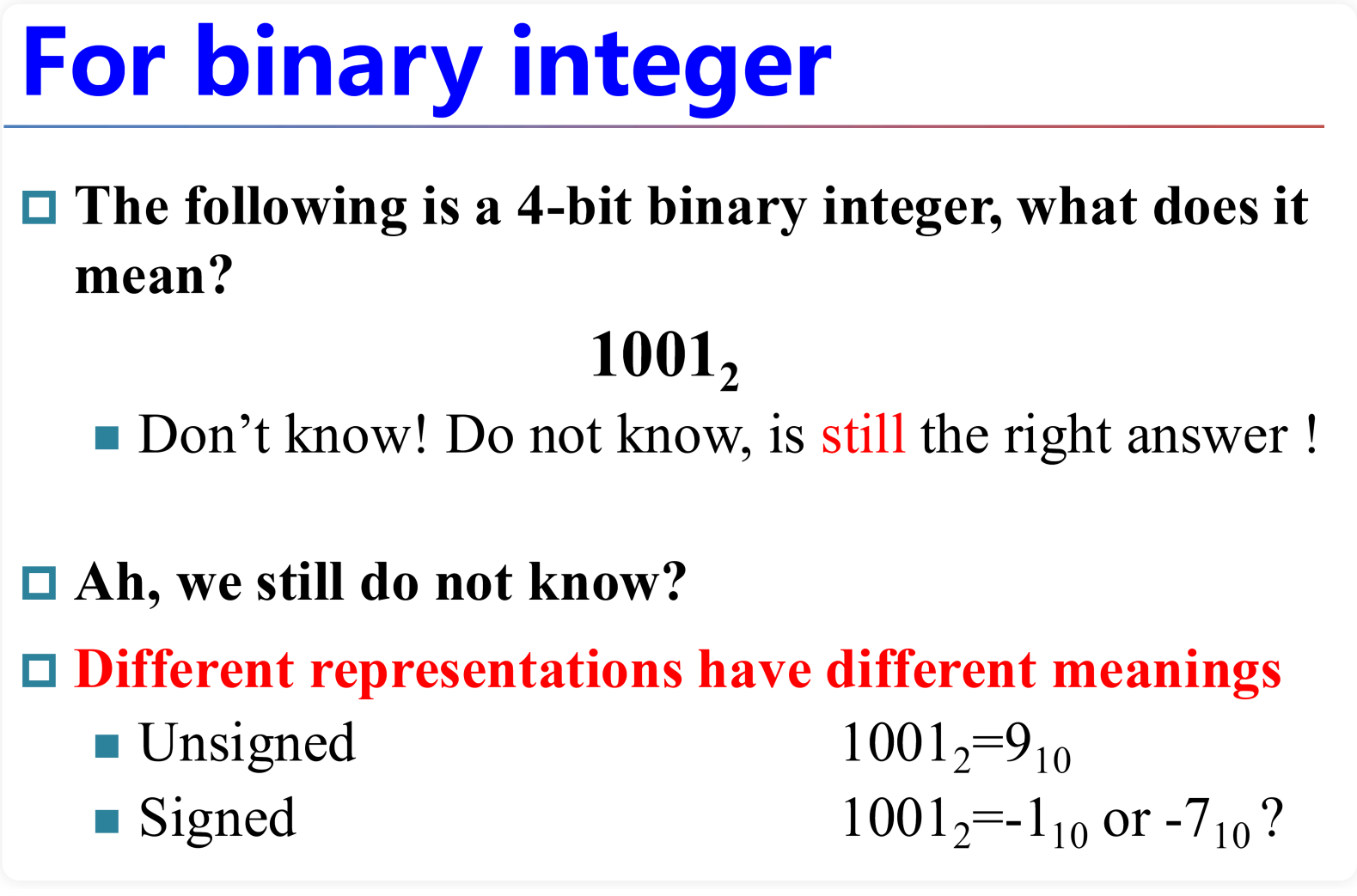
# CO 03 Arithmetic for Computer

## 1 Introduction

- 约定 word 是 32bits
- double word 是 64bits
- program counter `PC + 4`，指令都是 32 位的

## 2 Signed and Unsigned Numbers




- unsigned
- signed
  - sign magnitude
  - 2's complement
- *移码*，例如加上 -128 的 bias 来表示负数..

## 3 Addition, subtranction and ALU

- 加法需要注意 carry
- 减就是加上补码

## 3.1 Overflow

- 什么时候发生
  - 负数相加得正数
  - 正数相加得负数
- **设计逻辑电路可以进行溢出判断**
- 如何处理?
  - 可以 ignore，或者在编译的时候避免
  - OS 来处理
    - 将 PC 设置成 OS 处理溢出的指令地址
    - 可以修正然后 return
    - 也可以 exit 并报错
- ！发生的时候寄存器写入失效

## 3.2 ALU

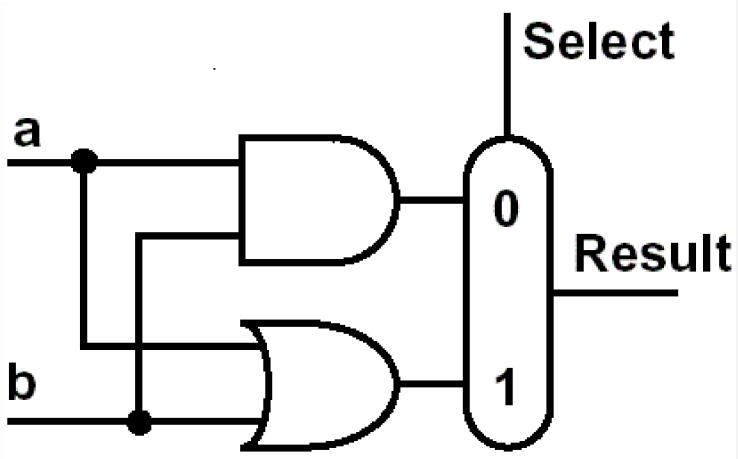
- ALU 支持的操作没有定义，一个加法器也可以叫做 ALU
  - 模块化设计，不同指令需要的部分是不一样的
  - MUX 输出选择

## Two methods constitute the ALU

- Modular design (e.g. add extensions to support add)
- Sharable logic with “select”

## Step by step:

- build a single bit ALU
- and expand it to the desired width



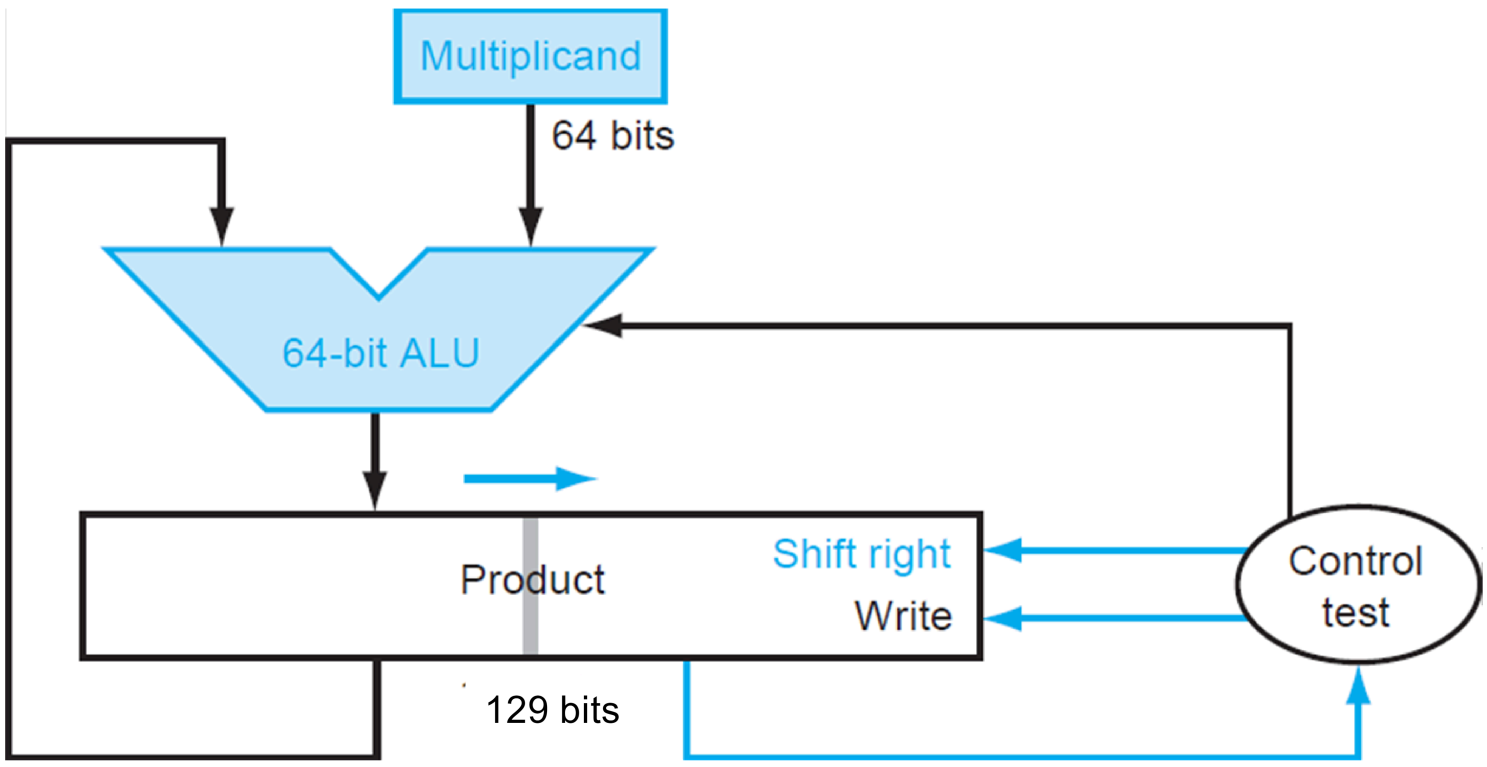

## First function:

- logic AND and OR

> **Tip**
>
> - 设计 1-bit 并扩展
> - 设计不同功能并选择

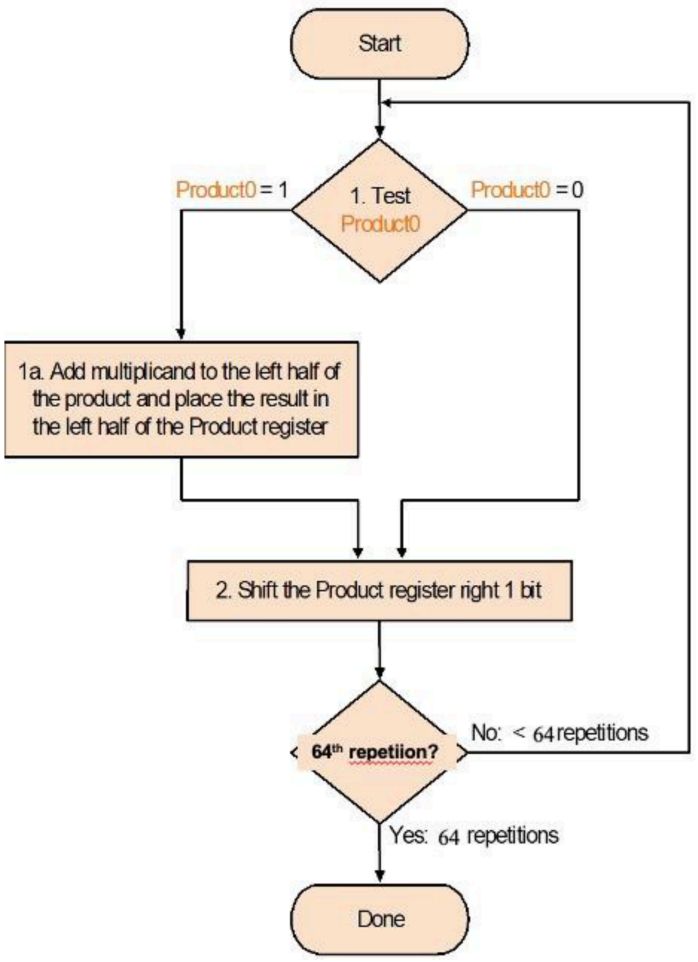
# 4 Multiplication

## Multiplier V3 Logic Diagram




- multiplier 先保存在 `product[63:0]`
- 如果 `product[0] == 1` ，那么 `product[127:64]` 加上 multiplicand，然后右移
- 129 bit 是为了保证过程中加法可能算出的最高位
- 一共需要移动 64 次，这是因为原本 `product[63:0]` 是 multiplier

- Set product register to '0'
- Load lower bits of product register with multiplier
- Test least significant bit of product register




## 4.1 Booth's Algorithm

Tip >

- Assumes: $Z=y\times 10111100$

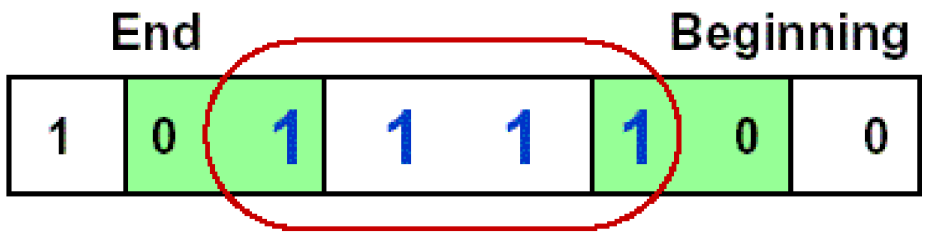
$$Z=y(10000000+111100+100-100)$$
$$=y(1\times 2^7+1000000-100)$$
$$=y(1\times 2^7+1\times 2^6-2^2)$$
$$=y(1\times 2^7+1\times 2^6+0\times 2^5+0\times 2^4+0\times 2^3+0\times 2^2+0\times 2^1+0\times 2^0-1\times 2^2)$$
$$=y(1\times 2^7+1\times 2^6+0\times 2^5+0\times 2^4+0\times 2^3+0\times 2^2-1\times 2^2+0\times 2^1+0\times 2^0)$$
$$=y\times 2^7+y\times 1\times 2^6+0\times 2^5+0\times 2^4+0\times 2^3+0\times 2^2-y\times 2^2+0\times 2^1+0\times 2^0)$$

add — Only shift — sub — Only shift

1 01 11 1 00

# Booth's Algorithm

- **Idea: If you have a sequence of '1's**
  - subtract at first '1' in multiplier
  - shift for the sequence of '1's
  - add where prior step had last '1'



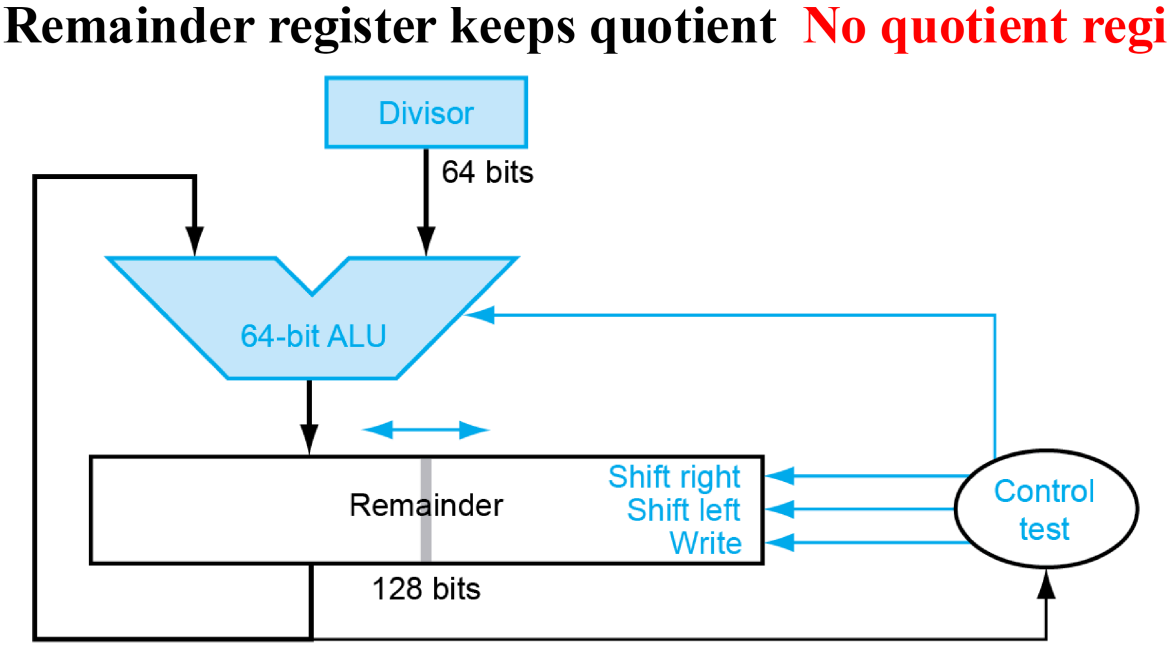

- **Result:**
  - Possibly less additions and more shifts
  - Faster, if shifts are faster than additions

- 将连续的 `1111` 变成 `10000 - 1`

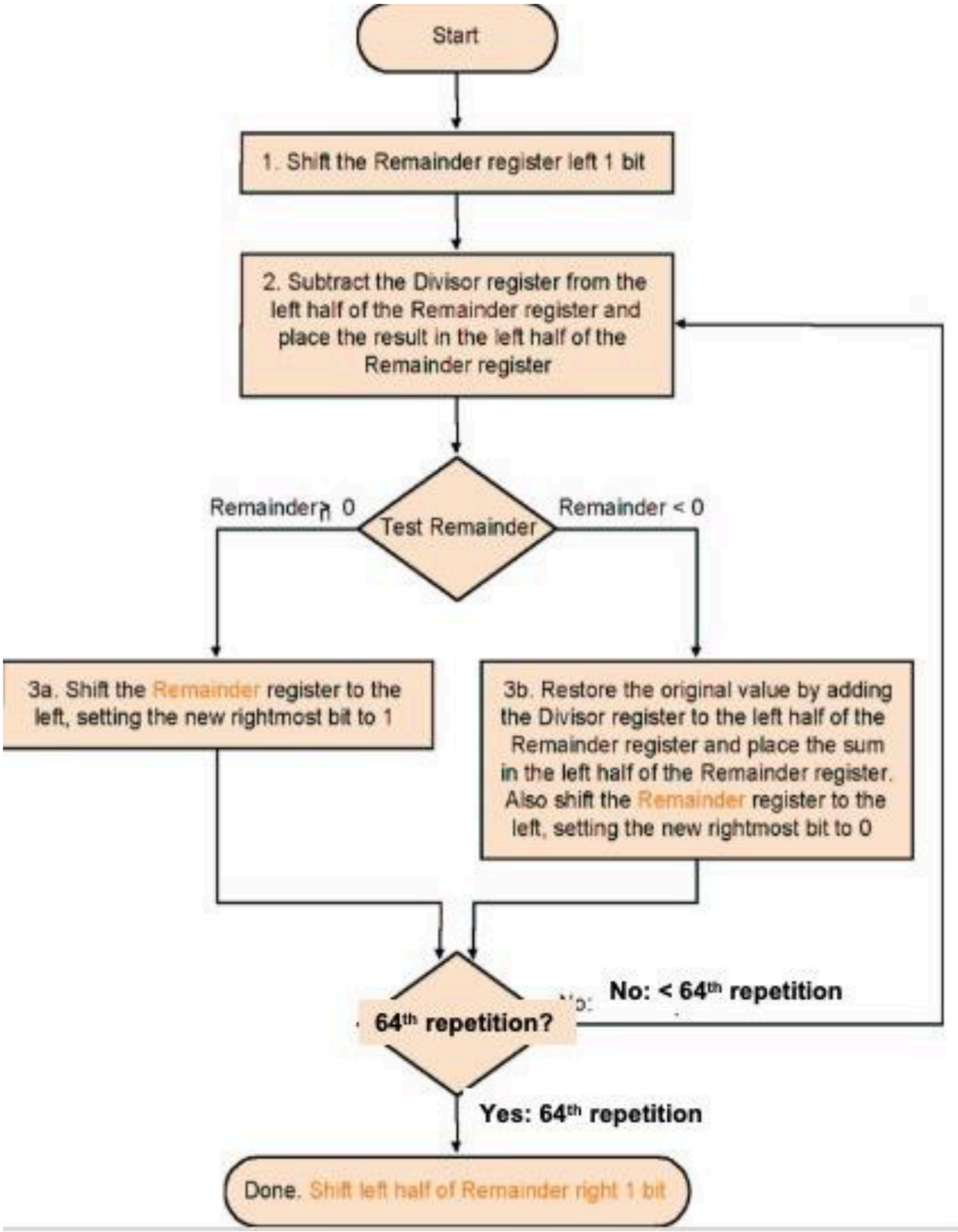
## 5 Division

# Modified Division

- **Reduction of Divisor and ALU width by half**
- **Shifting of the remainder**
- **Saving 1 iteration**
- **Remainder register keeps quotient No quotient register required**




- **One cycle per partial-remainder subtraction**
- **Looks a lot like a multiplier!**
  - Same hardware can be used for both




- Dividend 初始加载到 `remainder[63:0]`，并左移一位
- 将 `remainder[127:64]` 减去 divisor，结果保存在 `remainder[127:64]`
  - 如果结果大于 0，那么 `remainder` 左移，并使 `remainder[0] = 1`
  - 如果结果小于 0，那么将 `remainder[127:63]` 加上 divisor 复原，`remainder` 左移，`remainder[0] = 0`
  - 上述执行 64 次
- 最终，因为多左移了一次，需要 `remainder` 右移一位，`remainder[127:64]` 是余数，`remainder[63:0]` 是商

**⑦ 为什么同时需要 shift left/right**

- dividend 初始放在寄存器的右侧，肯定需要左移
- 最后的 remainder 在左半边，总的左移次数是 65 次，所以要右移一位

# Example 7/2 for Division V3

- Well known numbers: 0000 0111/0010

| iteration | step | Divisor | Remainder |
|---|---|---|---|
| 0 | Initial Values | 0010 | 0000 0111 |
| | Shift Rem left 1 | 0010 | **0000 1110** |
| 1 | 1.Rem=Rem-Div | 0010 | 1110 1110 |
| | 2b: Rem<0 →+Div,sll R,$R_0$=0 | 0010 | **0001 1100** |
| 2 | 1.Rem=Rem-Div | 0010 | 1111 1100 |
| | 2b: Rem<0 →+Div,sll R,$R_0$=0 | 0010 | **0011 1000** |
| 3 | 1.Rem=Rem-Div | 0010 | 0001 1000 |
| | 2a: Rem>0 →sll R,$R_0$=1 | 0010 | **0011 0001** |
| 4 | 1.Rem=Rem-Div | 0010 | 0001 0001 |
| | 2a: Rem>0 →sll R,$R_0$=1 | 0010 | **0010 0011** |
| | Shift left half of Rem right 1 | | **0001 0011** |

## 5.1 Signed Division

- 余数的符号与被除数的符号一致
- 其他的部分用 unsigned 就好
- ！除以 0，引发 overflow

# 6 Floating Point Numbers

- `sign`， 1 表示负数
- `exp`，具有 bias，fp32 是 -127，fp64 是 -1023
- `frac`，舍掉最左侧的 1

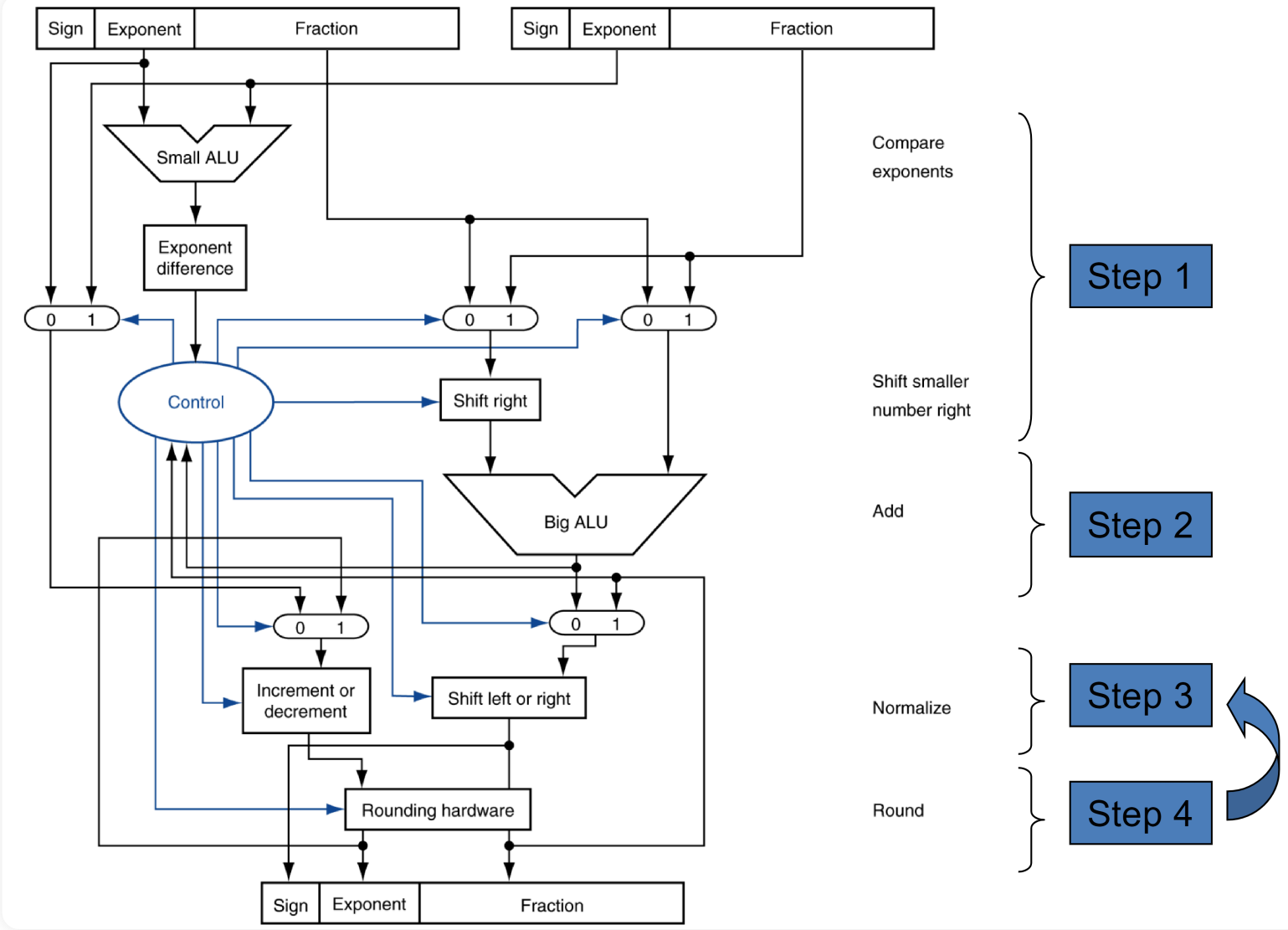
# Single-Precision Range

- **Exponents 00000000 and 11111111 reserved**
- **Smallest value**
  - Exponent: 00000001
    ⇒ actual exponent = 1 – 127 = –126
  - Fraction: 000…00 ⇒ significand = 1.0
  - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- **Largest value**
  - exponent: 11111110
    ⇒ actual exponent = 254 – 127 = +127
  - Fraction: 111…11 ⇒ significand ≈ 2.0
  - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

- ! 注意保留了一些 exp 的取值，以表示溢出

# Infinities and NaNs

- **Exponent = 111...1, Fraction = 000...0**
  - ±Infinity
  - Can be used in subsequent calculations, avoiding need for overflow check
- **Exponent = 111...1, Fraction ≠ 000...0**
  - Not-a-Number (NaN)
  - Indicates illegal or undefined result
    - e.g., 0.0 / 0.0




## 6.2 Rounding

- methods
  - always round up (to $+\infty$)
  - always round down (to $-\infty$)
  - truncate 直接截断
  - round to nearest even
    - 四舍五入
    - 0.5 要看前面 bit 的奇偶
- `guard (frac[-1]) | round (frac[-2]) | sticky (frac[-3])`
  - sticky 表示 round 后面是否还有 1，如果有 1 那么 sticky 就是 1
  - round to nearest even 精度是 0.5ulp

**Rounding to nearest even (Keep LSB to 0 when extra bits are 100)**

| | | |
|---|---|---|
| 0101010100 \| 011 -> | 0101010100 | (+0) |
| 0101010101 \| 011 -> | 0101010101 | (+0) |
| 0101010100 \| 100 -> | 0101010100 | (+0, keep LSB to 0) |
| 0101010101 \| 100 -> | 0101010110 | (+1, keep LSB to 0) |
| 0101010100 \| 101 -> | 0101010101 | (+1) |
| 0101010101 \| 101 -> | 0101010110 | (+1) |