



非常天空

# Computer Organization & Design

*Hardware/Software interface*

楼学庆

浙江大学计算机学院

<http://10.214.200.99/>

[Email:hzlou@163.com](mailto:hzlou@163.com)



玉泉校区曹光彪东楼507室



20:03:30

浙江大学计算机学院

# 联系方式

- 网站:
  - <http://10.214.200.99>
- 邮箱:
  - [hzlou@163.com](mailto:hzlou@163.com) (不收作业)



# 九·一八事变76周年

- 抗日战争的开始
  - 从1931年9月18日，日本关东军完成了向中国军队进攻，向中国百姓动武的最后准备，悍然发动了震惊中外的“九一八”事变。





# Computer Organization & Design

## 第03章: Arithmetic for Computer

**楼学庆**

<http://10.214.200.99/>

[Email:hzlou@163.com](mailto:hzlou@163.com)



玉泉校区曹光彪东楼507室

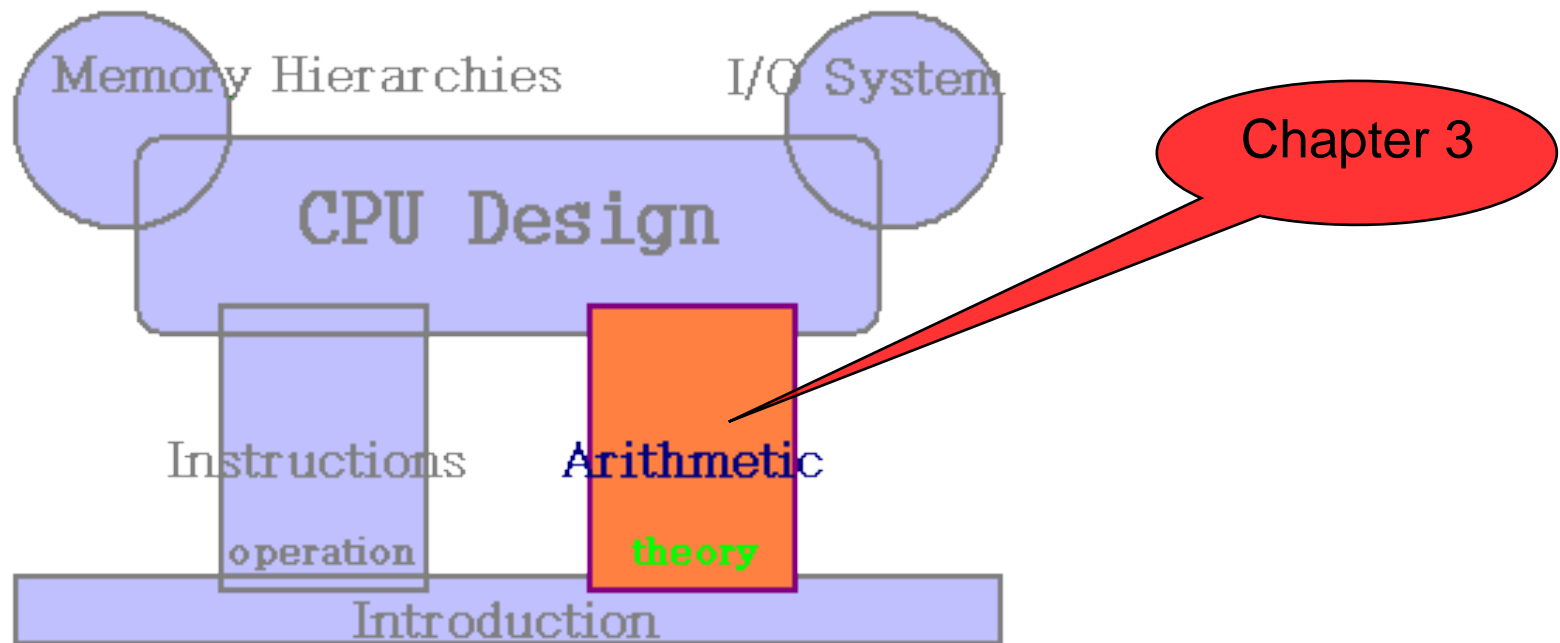


20:03:30

浙江大学计算机学院

# Chapter 3

## ■ Topics: Arithmetic for Computer





# Computer Arithmetic

---

- Today's topics:
  - Chapter 2 wrap-up
  - Numerical representations
  - Addition and subtraction



# Purpose

- Code: that can read/write for computer, it's a combination of 0 and 1.
- Data: real data, like: character, string, integer, float,.....image, music.....
- The principle for choosing  $f()$  and  $g()$ :
  - Can do
  - Easy to operate

$$Code = f(Data)$$

$$Data = g(Code)$$

# Purpose

■ Code: 0000...00 ~ 1111...11

■ Data: real data:

□ Character

□ String

□ Integer

□ Float

□ .....

□ image, music.....

$$Code = f(Data)$$

Software



# Signed number in 2's complement

- Since:  $Code = f(Data)$   
 $Data = g(Code)$

- $X_{code}, Y_{code} \rightarrow (X \text{ ? } Y)_{code}$

- We should:  $f(g(X_{code}) \text{ ? } g(Y_{code}))$

- Eq.  $X = \sin(x), Y = \sin(y) \rightarrow \sin(x+y) = ?$

- $\sin(x+y) \neq X+Y$

- 1.  $x = \arcsin(X), y = \arcsin(Y)$

$$\sin(x+y) = \sin(x = \arcsin(X) + \arcsin(Y))$$

- 2.  $\sin(x+y) = \sin(x) \cos(y) + \sin(y) \cos(x)$

$$= X(1-Y*Y)^{\frac{1}{2}} + Y(1-X*X)^{\frac{1}{2}}$$

# Biased notation (移码)

- $X=M+x, Y=M+y$

$$\rightarrow (x+y)_{\text{code}}=M+(x+y)$$

- $(x+y)_{\text{code}}=M+(x+y)$

- 1.  $=M + (X-M) + (Y-M)$

- 2.  $=X+Y-M$

- $M: 2^{N-1}: -2^{N-1} \sim +2^{N-1}-1$

$$X_{\text{移}} = M + X$$

# Biased notation (移码)

- $X = x + 2^{N-1}$ 
  - $x < 0$ :  $X = 2^{N-1} - |x| = 8000 - |x| = 7FFH - |x|$
  - $x \geq 0$ :  $X = 2^{N-1} + |x| = x + 8000$
- $x = X - 2^{N-1}$ 
  - $X < 2^{N-1}$ :  $x = -(2^{N-1} - |x|) = -(7FFH - |x|)$
  - $X \geq 2^{N-1}$ :  $x = X - 2^{N-1} = x \& 7FFF$
- $M$ :  $2^{N-1} : -2^{N-1} \sim +2^{N-1} - 1$

$$X_{\text{移}} = M + X$$

# Sign Magnitude (原码)

$$X_{\text{原}} = \begin{cases} X \dots\dots\dots X & \geq 0 \\ |X| + 2^{N-1} \dots\dots\dots X & < 0 \end{cases}$$

# Sign Magnitude (原码)

■  $X = (x + 2^N) \bmod 2^N$

□  $x \leq 0: X = 2^{N-1} + |x| = 8000 + |x| = 8000 \mid |x|$

□  $x \geq 0: X = x$

■  $x = X - 2^N$

□  $X < 2^{N-1}: x = +X$

□  $X \geq 2^{N-1}: x = -(2^{N-1} - X) = -(7FFF \& X)$

$$X_{\text{原}} = \begin{cases} X \dots\dots\dots X \geq 0 \\ |X| + 2^{N-1} \dots\dots\dots X < 0 \end{cases}$$

# 2's Complement (补码)

$$x_{\text{补}} = (2^N + x) \bmod 2^N$$

$$x_{\text{补}} = \begin{cases} x \dots x \geq 0 \\ 2^N + x = 2^N - |x| \dots x < 0 \end{cases}$$

当  $x < 0$  (16-bit,  $N=16$ ):

$$\begin{aligned} X &= 2^N - |x| = 8000 - |x| \\ &= ((2^N - 1) - |x|) + 1 = \text{FFFF} - |x| + 1 \\ &= ((2^N - 1) + 1) - |x| = \text{FFFG} - |x| \end{aligned}$$

# 2's Complement (补码)

- $X = (x + 2^N) \bmod 2^N$ 
  - $x < 0$ :  $X = 2^N - |x| = 10000 - |x| = \text{FFFG} - |x|$
  - $x \geq 0$ :  $X = x$
- $x = X - 2^N$ 
  - $X < 2^{N-1}$ :  $x = X$
  - $X \geq 2^{N-1}$ :  $x = -(2^N - X) = -(\text{FFFG} - X)$

$$x_{\text{补}} = (2^N + x) \bmod 2^N$$

# Binary Representation

- The binary number

01011000 00010101 00101110 11100111

Most significant bit ←      → Least significant bit

represents the quantity

$$0 \times 2^{31} + 1 \times 2^{30} + 0 \times 2^{29} + \dots + 1 \times 2^0$$

- A 32-bit word can represent  $2^{32}$  numbers between 0 and  $2^{32}-1$

... this is known as the unsigned representation as  
we're assuming that numbers are always positive



# Numbers

- Bits are just bits (no inherent meaning)
  - – conventions define relationship between bits and numbers
- Binary numbers (base 2)
  - 0000 0001 0010 0011 0100 0101 0110 0111  
1000 1001...  
decimal:  $0 \dots 2^n - 1$

# Numbers

- It gets more complicated:
  - numbers are finite (overflow)
  - fractions and real numbers
  - negative numbers
  - e.g., no MIPS subi instruction; addi can add a negative number)
- How do we represent negative numbers?
  - i.e., which bit patterns will represent which numbers?

# Possible Representations



- Sign Magnitude (原码)
- 1's Complement (反码)
- 2's Complement (补码)

000 = +0  
001 = +1  
010 = +2  
011 = +3  
100 = -0  
101 = -1  
110 = -2  
111 = -3

000 = +0  
001 = +1  
010 = +2  
011 = +3  
100 = -3  
101 = -2  
110 = -1  
111 = -0

000 = +0  
001 = +1  
010 = +2  
011 = +3  
100 = -4  
101 = -3  
110 = -2  
111 = -1

- Issues: balance, number of zeros, ease of operations

■ Which one is best? Why?



# BCD code

- Varsity
- R:base

$$a_i R^i \dots i = -m, \dots, 0, \dots, +n$$

$$x_{\text{补}} = (2^N + x) \bmod 2^N$$

$$x_{\text{移}} = 2^{N-1} + x$$

# BCD code

## ■ Varsity

Decimal	Binary	Hex	Octal	2421	5211	Gary	余3
0	0000	0	000	0000	0000	0000	0011
1	0001	1	001	0001	0001	0001	0100
2	0010	2	002	0010	0011	0011	0101
3	0011	3	003	0011	0101	0010	0110
4	0100	4	004	0100	0111	0110	0111
5	0101	5	005	1011	1000	1110	1000
6	0110	6	006	1100	1010	1010	1001
7	0111	7	007	1101	1100	1000	1010
8	1000	8	010	1110	1110	1100	1011
9	1001	9	011	1111	1111	0100	1100
10	1010	A	012				
11	1011	B	013				
12	1100	C	014				
13	1101	D	015				
14	1110	E	016				
15	1111	F	017				

# ASCII Vs. Binary

---

- Does it make more sense to represent a decimal number in ASCII?
- Hardware to implement arithmetic would be difficult
- What are the storage needs? How many bits does it take to represent the decimal number 1,000,000,000 in ASCII and in binary?



# Negative Numbers

32 bits can only represent  $2^{32}$  numbers – if we wish to also represent negative numbers, we can represent  $2^{31}$  positive numbers (incl zero) and  $2^{31}$  negative numbers

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{\text{two}} = 0_{\text{ten}}$$

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{\text{two}} = 1_{\text{ten}}$$

...

$$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{\text{two}} = 2^{31}-1$$

$$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{\text{two}} = -2^{31}$$

$$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{\text{two}} = -(2^{31} - 1)$$

$$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{\text{two}} = -(2^{31} - 2)$$

...

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{\text{two}} = -2$$

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{\text{two}} = -1$$



# 2's Complement

0000 0000 0000 0000 0000 0000 0000 0000<sub>two</sub> = 0<sub>ten</sub>

0000 0000 0000 0000 0000 0000 0000 0001<sub>two</sub> = 1<sub>ten</sub>

...

0111 1111 1111 1111 1111 1111 1111 1111<sub>two</sub> =  $2^{31}-1$

1000 0000 0000 0000 0000 0000 0000 0000<sub>two</sub> =  $-2^{31}$

1000 0000 0000 0000 0000 0000 0000 0001<sub>two</sub> =  $-(2^{31} - 1)$

1000 0000 0000 0000 0000 0000 0000 0010<sub>two</sub> =  $-(2^{31} - 2)$

...

1111 1111 1111 1111 1111 1111 1111 1110<sub>two</sub> = -2

1111 1111 1111 1111 1111 1111 1111 1111<sub>two</sub> = -1

Why is this representation favorable?

Consider the sum of 1 and -2 .... we get -1

Consider the sum of 2 and -1 .... we get +1

This format can directly undergo addition without any conversions!

Each number represents the quantity

$$x_{31} \cdot 2^{31} + x_{30} \cdot 2^{30} + x_{29} \cdot 2^{29} + \dots + x_1 \cdot 2^1 + x_0 \cdot 2^0$$



# 2's Complement

```

0000 0000 0000 0000 0000 0000 0000 0000two = 0ten
0000 0000 0000 0000 0000 0000 0000 0001two = 1ten
...
0111 1111 1111 1111 1111 1111 1111 1111two = 231-1

1000 0000 0000 0000 0000 0000 0000 0000two = -231
1000 0000 0000 0000 0000 0000 0000 0001two = -(231 - 1)
1000 0000 0000 0000 0000 0000 0000 0010two = -(231 - 2)
...
1111 1111 1111 1111 1111 1111 1111 1110two = -2
1111 1111 1111 1111 1111 1111 1111 1111two = -1

```

Note that the sum of a number  $x$  and its inverted representation  $x'$  always equals a string of 1s (-1).

$$x + x' = -1$$

$$x' + 1 = -x \quad \dots \text{hence, can compute the negative of a number by}$$

$$-x = x' + 1 \quad \text{inverting all bits and adding 1}$$

Similarly, the sum of  $x$  and  $-x$  gives us all zeroes, with a carry of 1

In reality,  $x + (-x) = 2^n$  ... hence the name 2's complement

# Example

---

- Compute the 32-bit 2's complement representations for the following decimal numbers:  
5, -5, -6

# Example

- Compute the 32-bit 2's complement representations for the following decimal numbers:

5, -5, -6

5: 0000 0000 0000 0000 0000 0000 0000 0101

-5: 1111 1111 1111 1111 1111 1111 1111 1011

-6: 1111 1111 1111 1111 1111 1111 1111 1010

Given -5, verify that negating and adding 1 yields the number 5

# Signed / Unsigned

- The hardware recognizes two formats:

unsigned (corresponding to the C declaration `unsigned int`)

-- all numbers are positive, a 1 in the most significant bit just means it is a really large number

signed (C declaration is `signed int` or just `int`)

-- numbers can be +/- , a 1 in the MSB means the number is negative

This distinction enables us to represent twice as many numbers when we're sure that we don't need negatives

# 输入与转换

## ■ 键盘输入



-123

## ■ 字符转数字



"-123"

*atoi()*

原码: 807B

补码: FF85

移码: 7F85

*atof()*

单精度浮点:  
0xC2F60000

# ASCII Vs. Binary

---

- Does it make more sense to represent a decimal number in ASCII?
- Hardware to implement arithmetic would be difficult
- What are the storage needs? How many bits does it take to represent the decimal number 1,000,000,000 in ASCII and in binary?

In binary: 30 bits ( $2^{30} > 1$  billion)

In ASCII: 10 characters, 8 bits per char = 80 bits

# Chinese character

■ hzk

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0	
0								X									01,00
1									X						X		00,82
2		X	X	X	X	X	X	X	X	X	X	X	X	X	X		7F,FE
3		X													X		40,02
4	X														X		80,02
5				X	X	X	X	X	X	X	X	X	X				1F,F8
6												X					00,10
7									X	X	X						00,E0
8								X									01,00
9		X	X	X	X	X	X	X	X	X	X	X	X	X			7F,FC
A									X								00,80
B									X								00,80
C						X			X								04,80
D							X		X								02,80
E								X	X								01,80
F																	00,00

# gb2312

- `char s[10]=" 习 ";`
- `Q = s[0]-0xA1;`
- `W = s[1]-0xA1;`
- `ofs = (Q*94+W)*16;`

#16 short/hz

16x16点阵汉字:

○○○○○●○○○●○○○○○○○○	→ 0480
○○○○○●●●●●○○○○○○○○	→ 0EA0
○●●●●○○○○●○○●○○○○○	→ 7890
○○○○○●○○○○●○○●○○○○○	→ 0890
○○○○○●○○○○●○○○○●○○○	→ 0884
●●●●●●●●●●●●●●●●○○	→ FFFE
○○○○○●○○○○●○○○○○○○○	→ 0880
○○○○○●○○○○●○○●○○○○○	→ 0890
○○○○○●○○●●○○●○○○○○	→ 0A90
○○○○○●●○○○○●○○○○○○○	→ 0C60
○○○○●●○○○○●○○○○○○○	→ 1840
○●●○●○○○○●○○●○○○○○	→ 68A0
○○○○○●○○●○○●○○○○○	→ 0920
○○○○○●○○●○○●○○●○○○	→ 0A14
○○●○●○○○○○○○○●○○●○○	→ 2814
○○○○●○○○○○○○○●●○○○	→ 100C



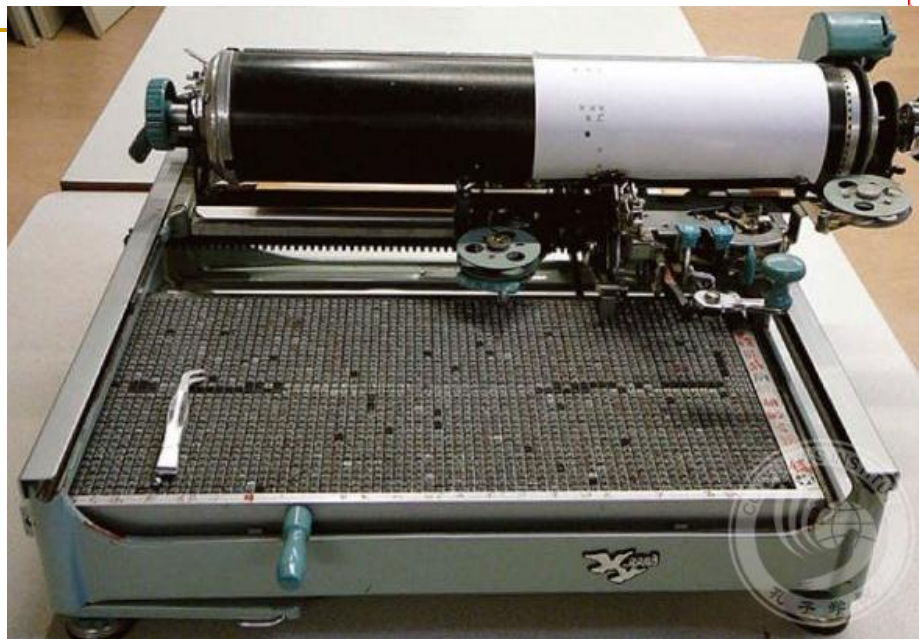
# 汉字处理

- 英文打字机
- 汉字绝望时期
  - 英文打字机



# 汉字处理

- 铅字打字机
- 汉字罗马化



## 《施氏食狮史》

石室诗士施氏，嗜狮，誓食十狮。施氏时  
时适市视狮。十时，适十狮适市。是时，适施  
氏适市。施氏视是十狮，恃矢势，使是十狮逝  
世。氏拾是十狮尸，适石室。石室湿，氏使侍  
拭石室。石室拭，施氏始试食是十狮尸。食时，  
始识是十狮尸，实十石狮尸。试释是事。

# 中文计算机

## ■ 问题:

- 什么是中文计算机?
- 对比中国, 为什么印度软件业相对其它产业更为发达?
- 乱码:

### ■ “锕斤拷”

- unicode编码转成GBK, 并不是所有的字都能够用unicode表示, 因此导致一些unicode无法比较的字默认填充一些字符, 一般反复填充的 `\xef\xbf\xbd` (UTF-8), 当使用GBK进行解析的时候, 按照汉字占两个字节的规则, `0xEFBF`就对应“锕”, `0xBDEF`对应“斤”, `0xBFBD`对应“拷”。
- 学计算机必先学英文
- 变量名、函数名、标点符号
  - 用c语言编写一个函数, 计算硬件系列课程平均成绩。要求程序简单易读、便于修改维护。硬件系列包括以下课程, 每门课成绩在0~100: 计算机组成, 体系结构, 逻辑与计算机, 汇编与接口, 嵌入式系统。



# 中文计算机

## ■ 关键字不是问题

### □ for (来自百度)

- 英 [fə(r); fɔ:(r)] 美 [fər, fɔ:r]
- prep. (表示对象、用途等) 给, 对; 为了; 关于; 代表; 受雇于; 意思是; 支持; 因为; 为得到; 换取; 就.....而言; .....后 (更好、更快乐等); (表示去向) 往; (安排或预定) 在.....时; 对 (某人) 来说 (困难、必需、愉快等); 以.....为价格; (表示一段时间) 计; 表示一系列事件之一
- conj. 因为, 由于
- abbr. 外国 (foreign); 林业 (forestry)

## ■ Windows Api

### □ WindowsApi



# 中文编程

- 中文编程不是降低门槛，是提高效率。

```
double HardwareAverage(int ComputerOrganization, int
ComputerArchitecture,int LogisticsAndComputerDesign,
int AssemblyAndInterface, int EmbeddedSystem){
    return (ComputerOrganization +
ComputerArchitecture + LogisticsAndComputerDesign
+ AssemblyAndInterface + EmbeddedSystem)/5.0;
}
```

```
public static double 硬件课程平均分(int 计组,int 体系,
int 逻辑,int 汇编,int 嵌入式){
    return (计组+体系+逻辑+汇编+嵌入式)/5.0;
}
```



# 什么是中文计算机？

- 什么是中文计算机？
- 变：
  - 中文计算机是相对现在的英文为主的计算机而言的。
    - 改的只是英文部分
      - `for`->循环, `if`->如果, `continue`->继续
      - `BroadcastSystemMessage()` -> 系统消息广播()
    - `Byte` -> `zjie` 作为基础寻址、操作单元
- 不变：
  - `i`、`j`、`k`、`x`、`y`、`z`不是英文，是符号。
  - 数学函数、物理公式、化学元素，数理化怎么写就还怎么写。
    - `sin()` -> `sin()`, `log()` -> `log()`

# ZB2014

## ■ 中文计算机关键：

### 1. 编程语言：

1. 关键字
2. 变量名
3. 函数名
4. 标点符号
5. 注释

### 2. 适应性：

1. 编码标准
2. 硬件结构
3. 操作系统
4. 软件生态

### 3. 积累：经年累月、年复一年、项目复项目、软件。。。。

# ZB2014

## ■ 编码:

### □ ZPC符号编码方法: ZB2014, z码 (浙标、浙码)

- 目前编码方法, 汉字与ASCII分别编码, 规则繁琐, 影响使用尤其在文本模式。
- 尝试结合硬件设计, 软硬件配合实行新的编码方案, 也测试新方案可能会对哪些产生影响。

## ■ 硬件结构&操作系统:

### □ 改变基础寻址、操作单位, 以16位为一字节 (zjie) 取代8位白特 (byte)。

$$1 \text{ zjie (字节)} = 16 \text{ bits (比特)} = 2 \text{ bytes (白特)}$$

- 硬件已经足够强大允许为性能多花一些冗余空间 (实际也不多)。



# ZB2014

- 1、以16位为一字节 (**zjie**) 取代8位白特 (byte)。  
(以下字节均为16位)
- 2、所有字符分“**使用字符**”与“**显示字符**”。
- 3、常规软件只处理使用字符，显示字符由专业软件使用。
- 4、**使用字符**为单字节16位，以15位统一编码，从0~0x7FFF。第16位为0，共32768个字符。其中：
  - 4.1、0~0x1FFF: 共8192个字符。原ASCII码(0~255)，其它字母符号等共5000左右使用，约3192保留以备扩展。  
**000x xxxx xxxx xxxx**
  - 4.2、0x2000~0x7FFFF: 共24576个字符。汉字，目前GBK共收汉字21003个，其余可作扩展保留。  
**001x xxxx xxxx xxxx ~ 011x xxxx xxxx xxxx**

# ZB2014

- 5、**显示字符**为双字节，32位。（一般为阅读软件）

- 第1字节最高两位为**10**,

**10xx xxxx xxxx xxxx**

- 第2字节最高两位为**11**,

**11xx xxxx xxxx xxxx**

共**256M**个字符，足够收录一切现有字符。最高位的设置可保证即使丢失数据造成乱码，也只乱一个字。“显示符号”一般只由专业阅读软件处理。

- 6、暂以**GB2312**码为汉字编码，不考虑区位，只按顺序。
  - 前**256**为**ASCII**
  - 随后为**GB2312**汉字
- 7、建议采用大头 (**Big-Endian**) 方式。

# ZB2014

## ■ 转换

```
#hzk: 国标字符串
for (k=0; k<hzk.length; k++) {
    if ((hzk[k] & 0x80) == 0) {           //ASCII
        mmf.writeByte(0);
        mmf.writeByte(hzk[k]);
    } else {                             //HanZi
        q = (hzk[k++] - 0xA1) & 0xFF;    //Qu
        w = (hzk[k] - 0xA1) & 0xFF;      //Wei
        m = 0x2000 + q * 94 + w;         //ZB码
        mmf.writeShort((short)m);
    }
}
```

# ZB2014

## ■ 判断

```
#xts: 哲标字符串
if((xts[idx]&0xE000)==0){ //ASCII、其它符号
    k=xts[idx]*16;        //字库中位置
} else if(xts[idx]&0x8000)==0){ //汉字字符
    k=(xts[idx]-0x2000+256)*16;
} else {
    /* 显示字符 */
}
```

# MIPS programming

- `sllv`     `$s0, $s1, $s2`
- `?`        `$s0, $s1`

```
srl    $s1, $s1, 1
sll    $t0, $s0, 31
srl    $s0, $s0, 1
or     $s1, $s1, $t0
```

```
        .data
mask    .word    0xffff83f
        .text
start:  lw        $t0, mask
        lw        $s0, shifter
        and       $s0, $s0, $t0
        andi     $s2, $s2, 0x1f
        sll      $s2, $s2, 6
        or       $s0, $s0, $s2
        sw       $s0, shifter

shifter:
        sll      $s0, $s1, 0
```

# MIPS Instructions

---

Consider a comparison instruction:

`slt $t0, $t1, $zero`

and \$t1 contains the 32-bit number 1111 01...01

What gets stored in \$t0?

# MIPS Instructions

---

Consider a comparison instruction:

`slt $t0, $t1, $zero`

and \$t1 contains the 32-bit number 1111 01...01

What gets stored in \$t0?

The result depends on whether \$t1 is a signed or unsigned number – the compiler/programmer must track this and accordingly use either `slt` or `sltu`

`slt $t0, $t1, $zero` stores 1 in \$t0

`sltu $t0, $t1, $zero` stores 0 in \$t0

# The Bounds Check Shortcut

- Suppose we want to check if  $0 \leq x < y$   
and  $x$  and  $y$  are signed numbers (stored in  $\$a1$  and  $\$t2$ )

The following single comparison can check both conditions

`sltu $t0, $a1, $t2`

`beq $t0, $zero, EitherConditionFails`

We know that  $\$t2$  begins with a 0

If  $\$a1$  begins with a 0, `sltu` is effectively checking the second condition

If  $\$a1$  begins with a 1, we want the condition to fail and coincidentally,  
`sltu` is guaranteed to fail in this case



# Sign Extension

- Occasionally, 16-bit signed numbers must be converted into 32-bit signed numbers – for example, when doing an add with an immediate operand
- The conversion is simple: take the most significant bit and use it to fill up the additional bits on the left – known as sign extension

So  $2_{10}$  goes from 0000 0000 0000 0010 to  
0000 0000 0000 0000 0000 0000 0000 0010

and  $-2_{10}$  goes from 1111 1111 1111 1110 to

1111 1111 1111 1111 1111 1111 1111 1110

# Alternative Representations

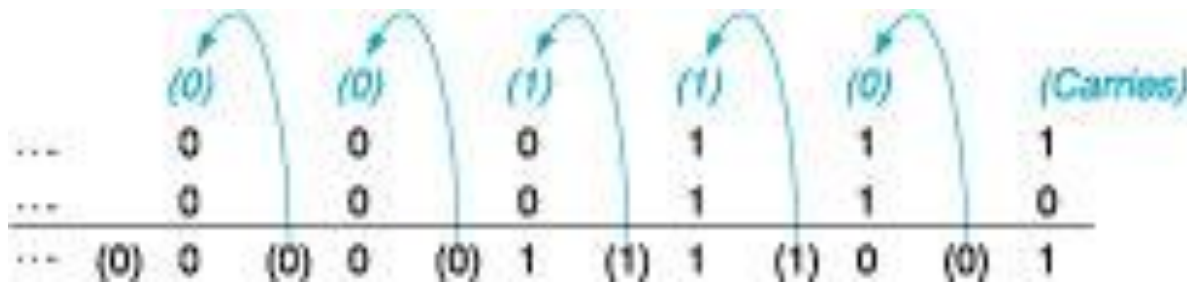
---

- The following two (intuitive) representations were discarded because they required additional conversion steps before arithmetic could be performed on the numbers
  - sign-and-magnitude: the most significant bit represents +/- and the remaining bits express the magnitude
  - one's complement:  $-x$  is represented by inverting all the bits of  $x$

Both representations above suffer from two zeroes

# Addition and Subtraction

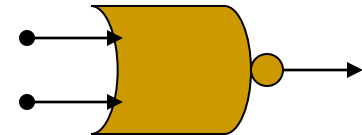
- Addition is similar to decimal arithmetic
- For subtraction, simply add the negative number – hence, subtract  $A-B$  involves negating  $B$ 's bits, adding 1 and  $A$



## ■ NOR

$$\square S = a \text{ NOR } b$$

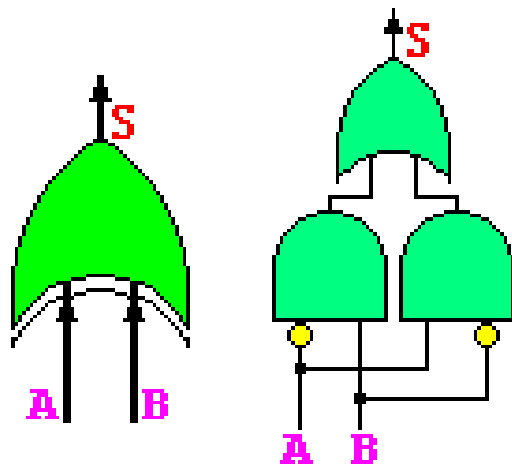
a	b	S
0	0	1
0	1	0
1	0	0
1	1	0



# Overflows

- For an unsigned number, overflow happens when the last carry (1) cannot be accommodated
- For a signed number, overflow happens when the most significant bit is not the same as every bit to its left
  - when the sum of two positive numbers is a negative result
  - when the sum of two negative numbers is a positive result
  - The sum of a positive and negative number will never overflow
- MIPS allows **addu** and **subu** instructions that work with unsigned integers and never flag an overflow – to detect the overflow, other instructions will have to be executed

# Half Adder



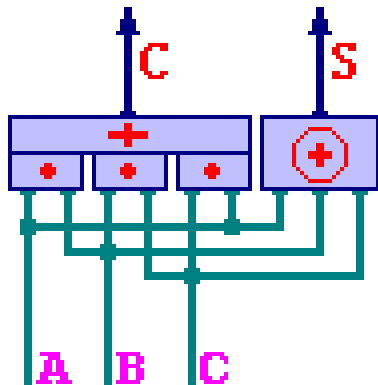
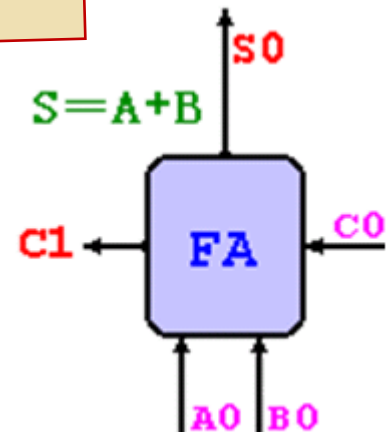
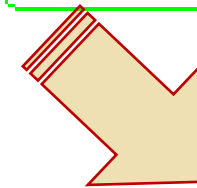
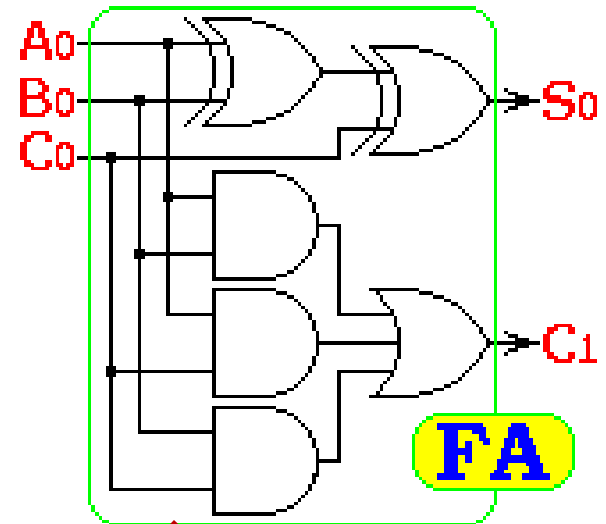
一位半加器

输入		输出	
.A.	.B.	xor	add
0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	0

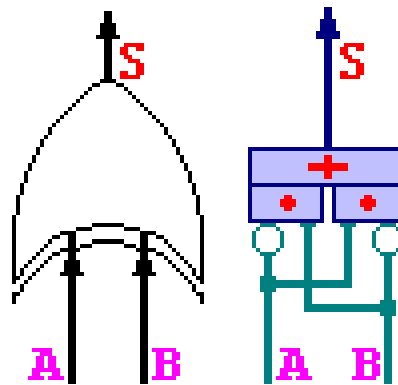
# Add

$$C_{i+1} = A_i B_i + B_i C_i + C_i A_i$$

$$S_i = A_i \oplus B_i \oplus C_i$$



一位全加器



一位半加器

# Ripple-carry Adder

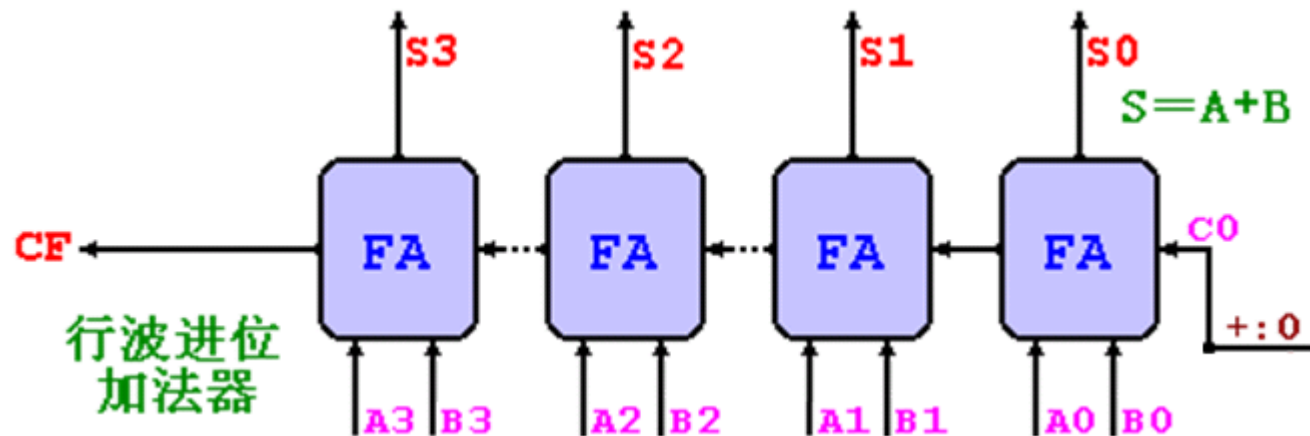
## ■ 4-bit Ripple-carry adder

$$C_1 = A_0B_0 + B_0C_0 + C_0A_0$$

$$C_2 = A_1B_1 + B_1C_1 + C_1A_1$$

$$C_3 = A_2B_2 + B_2C_2 + C_2A_2$$

$$C_4 = A_3B_3 + B_3C_3 + C_3A_3$$





# Adder

## ■ Add

$$C_1 = A_0B_0 + B_0C_0 + C_0A_0$$

$$C_2 = A_1B_1 + B_1C_1 + C_1A_1$$

$$C_3 = A_2B_2 + B_2C_2 + C_2A_2$$

$$C_4 = A_3B_3 + B_3C_3 + C_3A_3$$

## ■ 代入

$$C_1 = A_0B_0 + B_0C_0 + C_0A_0$$

$$C_2 = A_1B_1 + B_1(A_0B_0 + B_0C_0 + C_0A_0)$$

$$+ A_1(A_0B_0 + B_0C_0 + C_0A_0)$$

$$= A_1B_1 + B_1A_0B_0 + B_1B_0C_0 + B_1C_0A_0$$

$$+ A_1A_0B_0 + A_1B_0C_0 + A_1C_0A_0$$

# carry lookahead

$$2^{n+1}-1$$

## ■ Add

$$\begin{aligned}
 C_3 &= A_2B_2 + B_2(A_1B_1 + B_1A_0B_0 + B_1B_0C_0 + B_1C_0A_0 \\
 &\quad + A_1A_0B_0 + A_1B_0C_0 + A_1C_0A_0) \\
 &\quad + A_2(A_1B_1 + B_1A_0B_0 + B_1B_0C_0 + B_1C_0A_0 \\
 &\quad + A_1A_0B_0 + A_1B_0C_0 + A_1C_0A_0) \\
 &= A_2B_2 + B_2A_1B_1 + B_2B_1A_0B_0 + B_2B_1B_0C_0 + B_2B_1C_0A_0 \\
 &\quad + A_1A_0B_2B_0 + A_1B_2B_0C_0 + B_2A_1C_0A_0 \\
 &\quad + A_2A_1B_1 + A_2B_1A_0B_0 + A_2B_1B_0C_0 + A_2B_1C_0A_0 \\
 &\quad + A_2A_1A_0B_0 + A_2A_1B_0C_0 + A_2A_1C_0A_0 \\
 C_4 &= A_3B_3 + B_3C_3 + C_3A_3
 \end{aligned}$$

# Adder

## ■ Add

$$C_1 = A_0B_0 + (A_0+B_0)C_0$$

$$C_2 = A_1B_1 + (A_1+B_1)C_1$$

$$C_3 = A_2B_2 + (A_2+B_2)C_2$$

$$C_4 = A_3B_3 + (A_3+B_3)C_3$$

## ■ p , g

$$p_i = A_iB_i$$

$$g_i = A_i + B_i$$

$$C_1 = p_0 + g_0C_0$$

$$C_2 = p_1 + g_1C_1$$

$$C_3 = p_2 + g_2C_2$$

$$C_4 = p_3 + g_3C_3$$

# Adder

$n+1$

$$p_i = A_i B_i$$
$$g_i = A_i + B_i$$

## ■ Add

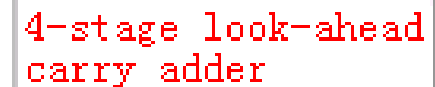
$$C_1 = p_0 + g_0 C_0$$

$$C_2 = p_1 + g_1 C_1 = p_1 + g_1 (p_0 + g_0 C_0)$$
$$= p_1 + g_1 p_0 + g_1 g_0 C_0$$

$$C_3 = p_2 + g_2 (p_1 + g_1 p_0 + g_1 g_0 C_0)$$
$$= p_2 + g_2 p_1 + g_2 g_1 p_0 + g_2 g_1 g_0 C_0$$

$$C_4 = p_3 + g_3 (p_2 + g_2 p_1 + g_2 g_1 p_0 + g_2 g_1 g_0 C_0)$$
$$= p_3 + g_3 p_2 + g_3 g_2 p_1 + g_3 g_2 g_1 p_0 + g_3 g_2 g_1 g_0 C_0$$

$$C_5 = p_4 + g_4 C_4$$

$$\begin{aligned} \mathbf{p}_i &= \mathbf{A}_i \mathbf{B}_i \\ \mathbf{g}_i &= \mathbf{A}_i + \mathbf{B}_i \end{aligned}$$


# Addition & Subtraction

■ 证明:

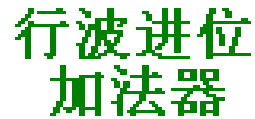
$$X_{\text{补}} = (2^N + X) \bmod 2^N$$

$$\begin{aligned} X_{\text{补}} + Y_{\text{补}} &= (2^N + X + 2^N + Y) \bmod (2^N) \\ &= (X + Y)_{\text{补}} \end{aligned}$$

$$\therefore (X + Y)_{\text{补}} = X_{\text{补}} + Y_{\text{补}}$$

$$\begin{aligned} X_{\text{补}} - Y_{\text{补}} &= (2^N + X - 2^N - Y) \bmod (2^N) \\ &= (X - Y)_{\text{补}} \end{aligned}$$

$$\therefore (X - Y)_{\text{补}} = X_{\text{补}} + (-Y)_{\text{补}}$$



# Purpose

■ Code: 0000...00 ~ 1111...11

■ Data: real data:

□ Character

□ String

□ Integer

□ Float

□ .....

□ image, music.....

$$Code = f(Data)$$

Software



# Signed number

- Since:

$$Code = f(Data)$$

$$Data = g(Code)$$

- $X_{code}, Y_{code} \rightarrow (X \text{ ? } Y)_{code}$

- We should:  $f(g(X_{code}) \text{ ? } g(Y_{code}))$

- Eq.  $X=\sin(x), Y=\sin(y) \rightarrow \sin(x+y)=?$

- $\sin(x+y) \neq X+Y$

- 1.  $x=\arcsin(X), y=\arcsin(Y)$

$$\sin(x+y)=\sin(x=\arcsin(X) + \arcsin(Y))$$

- 2.  $\sin(x+y)=\sin(x)\cos(y)+\sin(y)\cos(x)$

$$= X(1-Y*Y)^{\frac{1}{2}}+ Y(1-X*X)^{\frac{1}{2}}$$

# Biased notation (移码)

- $X=M+x, Y=M+y$

$$\rightarrow (x+y)_{\text{code}}=M+(x+y)$$

- $(x+y)_{\text{code}}=M+(x+y)$

- 1.  $=M + (X-M) + (Y-M)$

- 2.  $=X+Y-M$

- $M: 2^{N-1}: -2^{N-1} \sim +2^{N-1}-1$

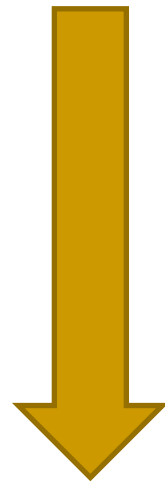
$$X_{\text{移}} = M + X$$

# 整数运算器

## ■ Integer

- 已知: **and**、**or**、**not**、**xor**、移位
- 多种码制: 原码、移码、**补码**、BCD码、.....

- `int atom(char*);`
- `char* mtoa(int);`
- `int madd(int, int);`
- `int msub(int, int);`
- `int mmul(int, int);`
- `int mdiv(int, int);`
- `int mmod(int, int);`



## ■ CarryOut, Overflow

■ Extend: **8->16, 16->32**

■ **compare**

# IEEE754浮点数运算器

## ■ Float

□ 已知：整数运算

□ `float atof(char*) ;`

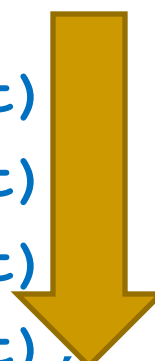
□ `char* ftoa(float) ;`

□ `float fadd(float, float)`

□ `float fsub(float, float)`

□ `float fmul(float, float)`

□ `float fdiv(float, float) ,`



■ Overflow: INF

■ Extend: *int->float->double->float->int*

■ Round:

■ compare

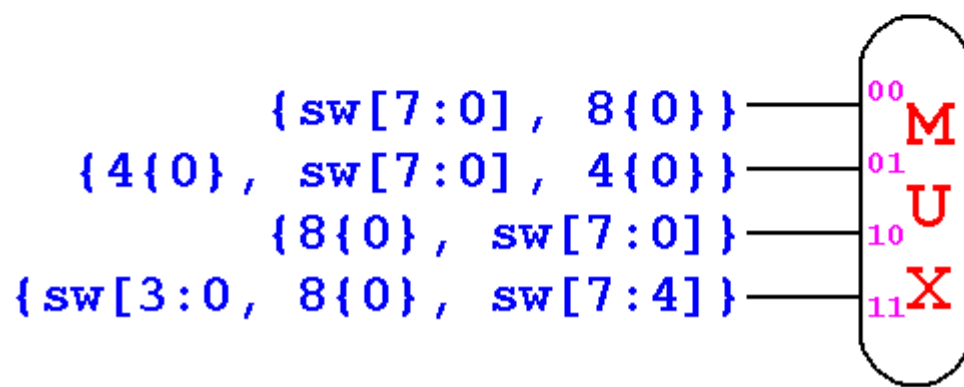
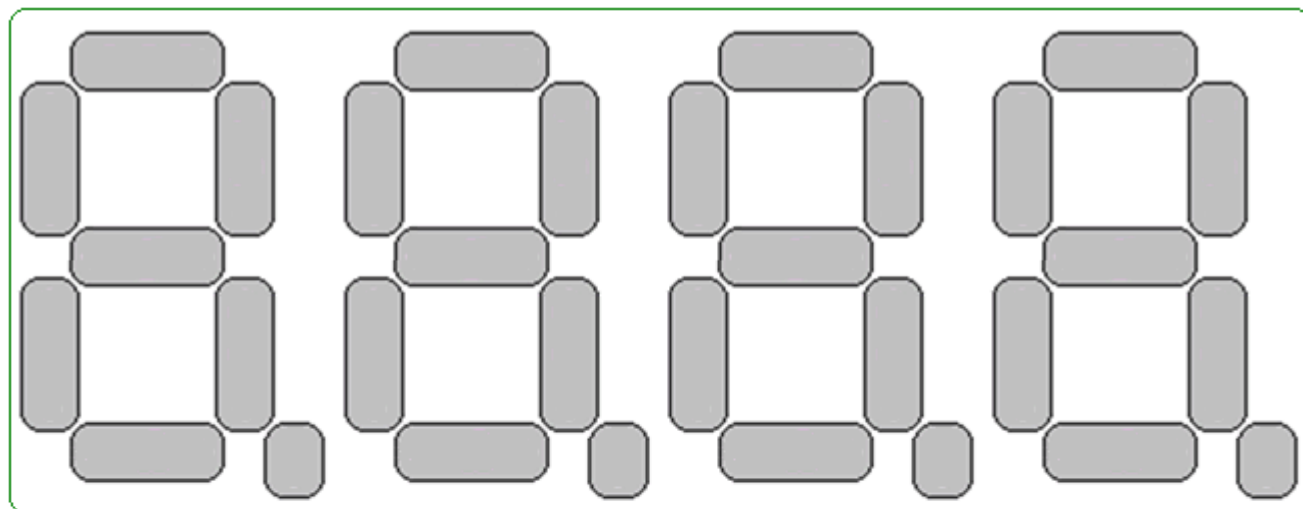
- 表达方式特点
- 算法证明、分析
- 程序实现
- 特例讨论
  - 溢出
  - 扩展
  - 零与无穷大
  - . . .



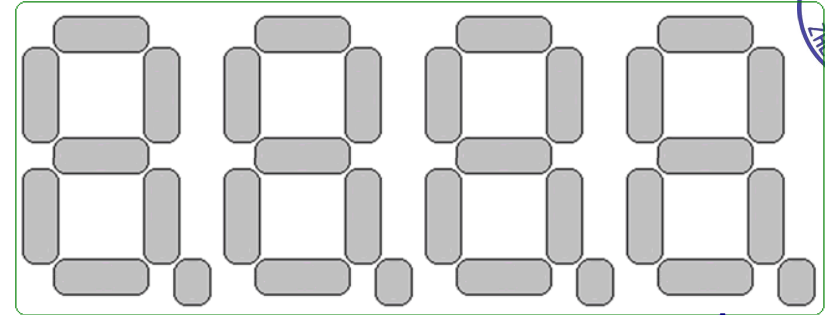
# module MUX16\_4x1

```
module MUX16_4x1(  
    output [15:0] S, input [1:0]  ctr  
    input [15:0] A, input [15:0] B, input [15:0] C, input [15:0] D);  
  
    wire[15:0] M0, M1, M2, M3;  
    and (t0, ~ctr[1], ~ctr[0]),  
        (t1, ~ctr[1],  ctr[0]),  
        (t2,  ctr[1], ~ctr[0]),  
        (t3,  ctr[1],  ctr[0]);  
  
    assign M0 = A & {16{t0}};  
    assign M1 = B & {16{t1}};  
    assign M2 = C & {16{t2}};  
    assign M3 = D & {16{t3}};  
  
    assign S = M0 | M1 | M2 | M3;  
endmodule
```

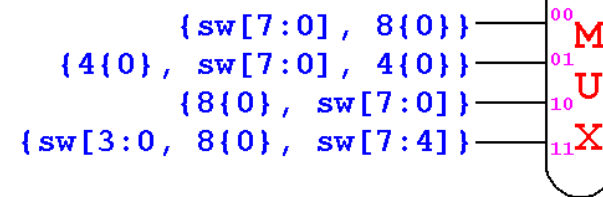




# Display



```
module top(  
    input [7:0] sw,  
    ... ..
```



```
    MUX16_4x1 mx2(numbr, Mode[1:0],  
        {sw[7:0], 8{0}},           //00  
        {4{0}, sw[7:0], 4{0}},     //01  
        {8{0}, sw[7:0]},           //10  
        {sw[3:0], 8{0}, sw[7:4]}   //11  
    );
```

```
    Display dy(clk, Rst, theCnt, numbr, node, segment);  
endmodule
```







# Binary Mul & Div

- Today's topics:
  - Addition/Subtraction
  - Multiplication
  - Division



# Multiplication Example

- Multiplicand
- Multiplier

$$\begin{array}{r} 1000 \\ \times 1001 \\ \hline 1000 \\ 0000 \\ 0000 \\ 1000 \\ \hline 1001000_{\text{ten}} \end{array}$$

- Product
- In every step
  - multiplicand is shifted next bit of multiplier is examined (also a shifting step)
  - if this bit is 1, shifted multiplicand is added to the product

# MUL

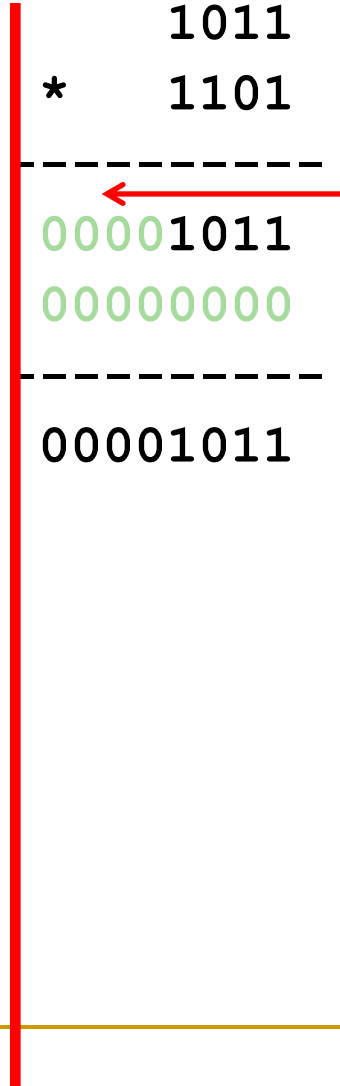
## ■ MUL

```
      1011
    * 1101
    -----
  00001011
  00000000
  00101100
  01011000
  -----
 10001111
```

# MUL

## ■ MUL

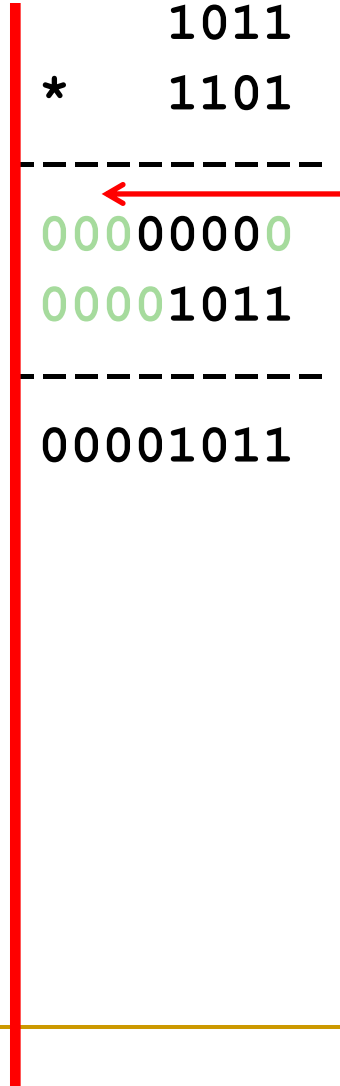
```
      1011
    * 1101
    -----
  00001011
  00000000
  -----
  00001011
```

A diagram illustrating the binary multiplication (MUL) of 1011 and 1101. The numbers are aligned vertically, separated by two vertical red lines. A dashed line separates the multiplicand (1011) and multiplier (1101) from the partial products. The first partial product is 00001011, with a red arrow pointing to its rightmost bit. The second partial product is 00000000. A second dashed line separates the partial products from the final product, 00001011.

# MUL

## ■ MUL

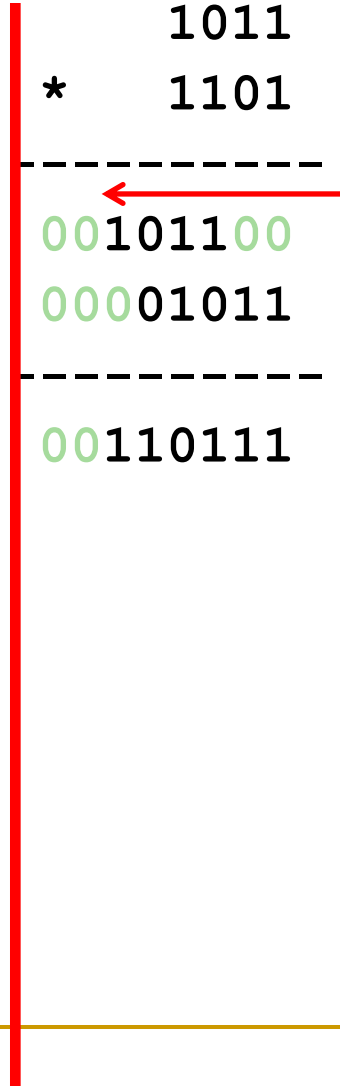
```
      1011
    *  1101
    -----
    00000000
    00001011
    -----
    00001011
```

A diagram illustrating the binary multiplication of 1011 and 1101. The numbers are aligned vertically with a multiplication symbol. A dashed line separates the multiplicand from the multiplier. Below the multiplier, the partial products are shown: 00000000 (multiplied by the least significant bit) and 00001011 (multiplied by the second bit). A red arrow points from the right edge of the second partial product to the left edge of the first partial product, indicating a leftward shift. A final dashed line separates the partial products from the final result, 00001011.

# MUL

## ■ MUL

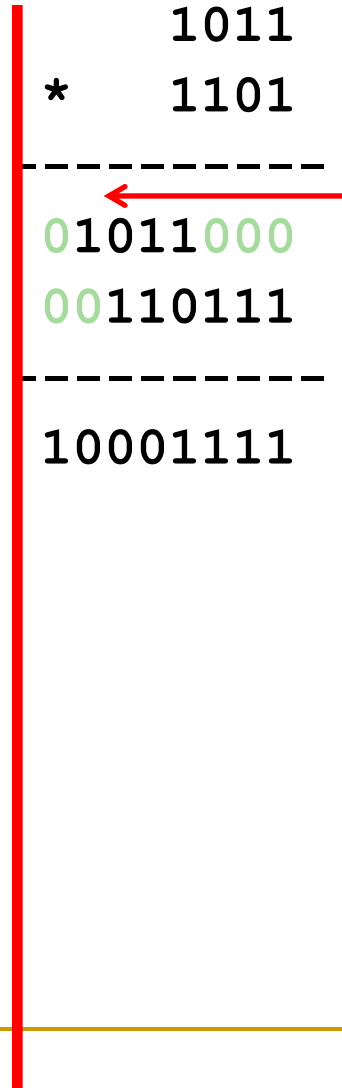
```
      1011
    *  1101
    -----
  00101100
  00001011
  -----
  00110111
```

A diagram illustrating the binary multiplication (MUL) of 1011 and 1101. The numbers are aligned vertically, with the multiplier 1101 on the right. A red arrow points from the rightmost bit of the multiplier (1) to the start of the first partial product (00101100). The partial products are shown in green, and the final result (00110111) is also in green. The diagram is framed by two vertical red lines.

# MUL

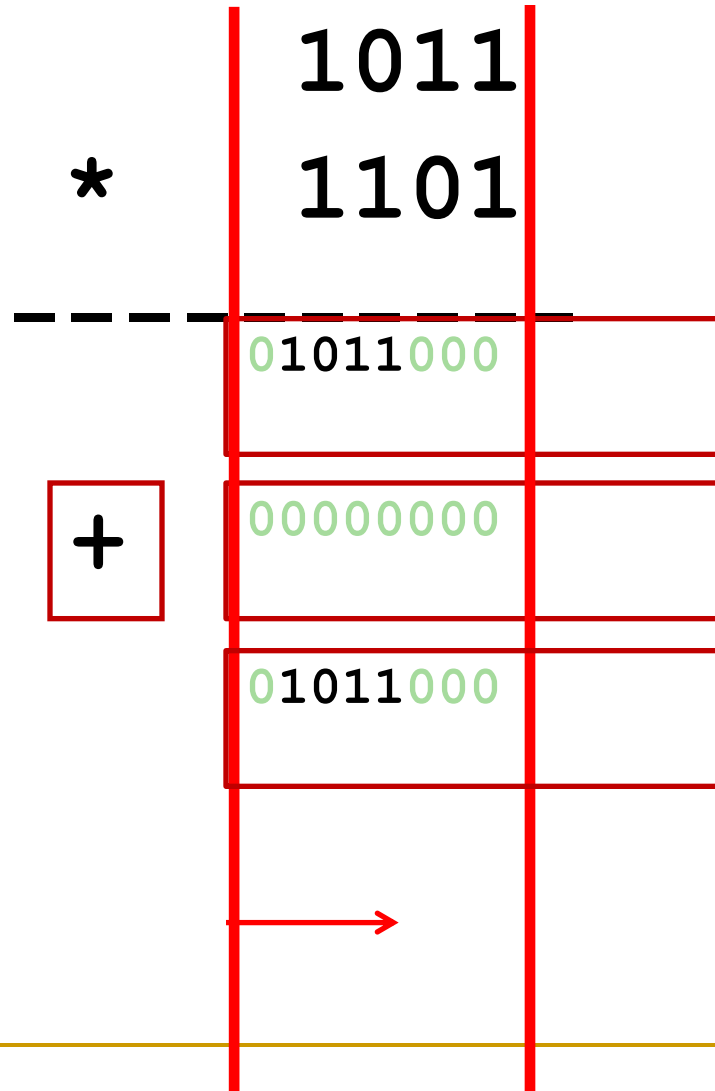
## ■ MUL

```
      1011
    *  1101
    -----
  01011000
  00110111
  -----
  10001111
```

A diagram illustrating the binary multiplication (MUL) of 1011 and 1101. The numbers are aligned vertically with a multiplication symbol (\*). Two horizontal dashed lines separate the multiplicand and multiplier from the partial products, and the partial products from the final product. The first partial product, 01011000, is shown in green, with a red arrow pointing to its rightmost four digits (1000). The second partial product, 00110111, is also in green. The final product, 10001111, is shown in black. Two vertical red lines are positioned on either side of the multiplication process.

# MUL

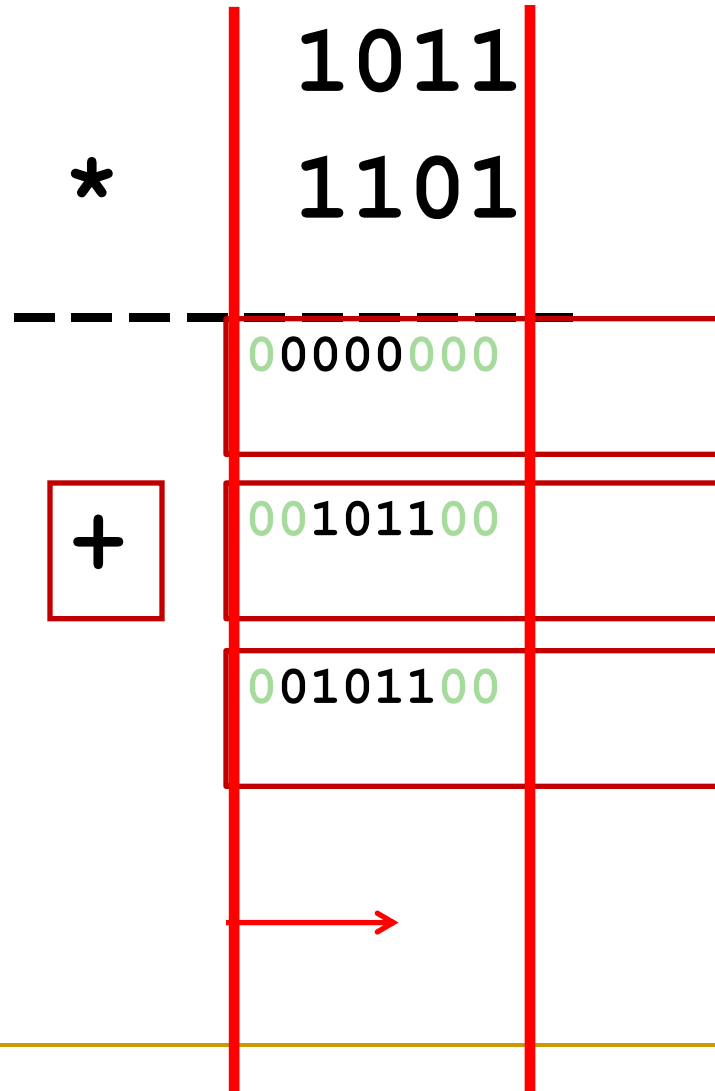
## MUL





# MUL

## MUL



# MUL

## MUL

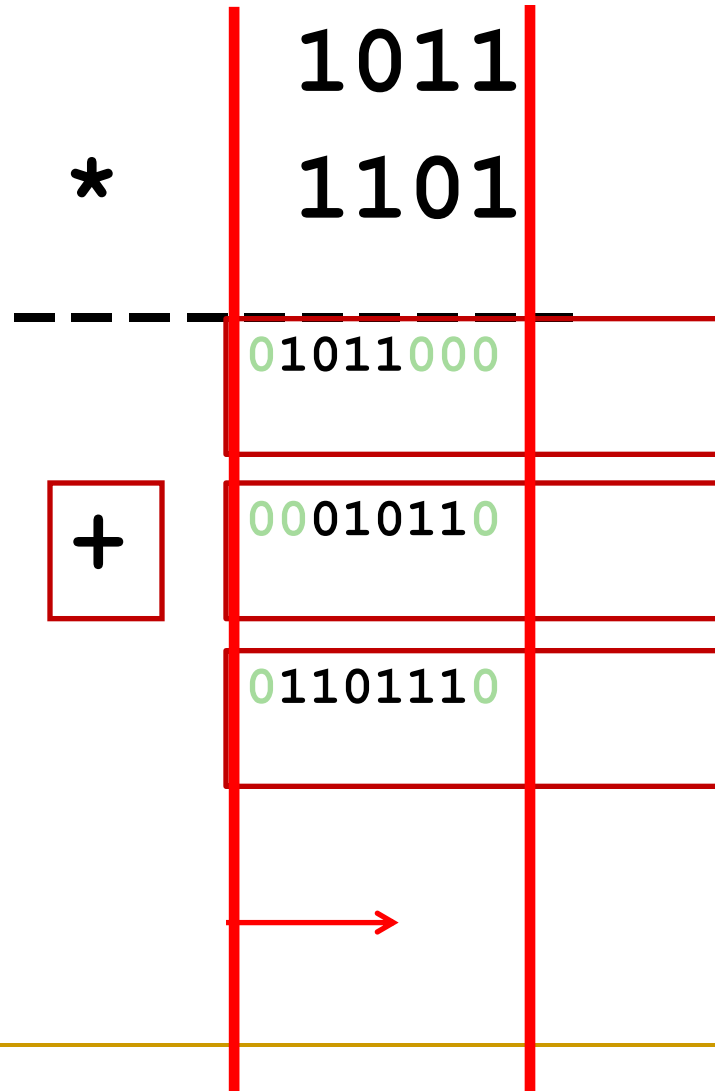
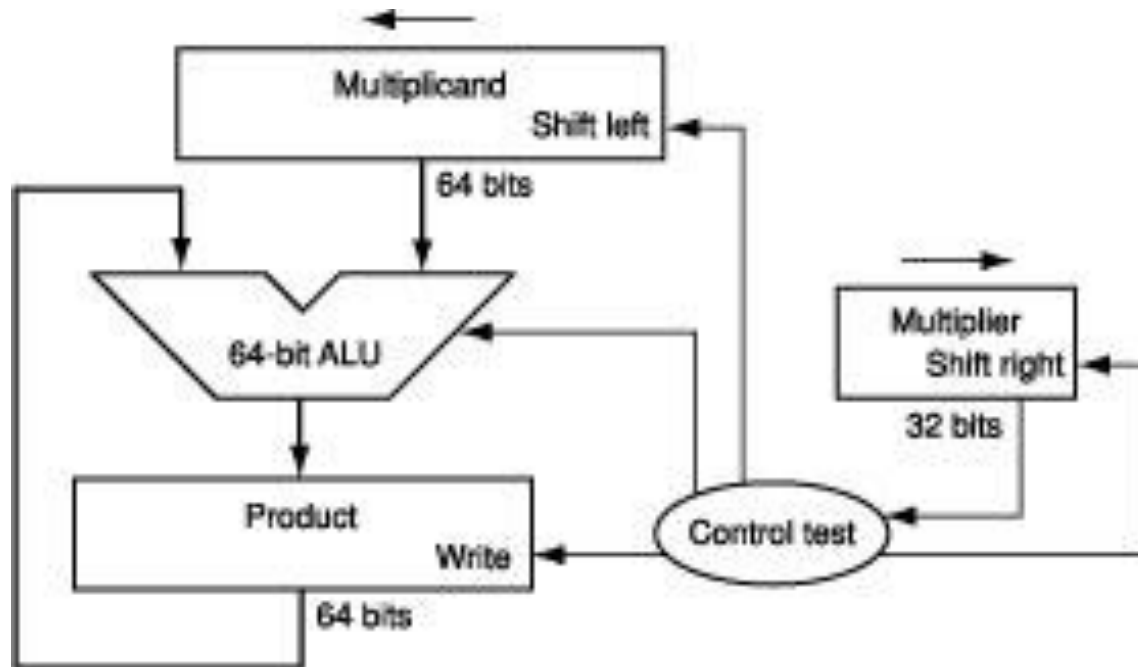


Diagram illustrating a 4-bit ripple-carry adder circuit. The inputs are two 4-bit numbers: 1011 and 1101. The circuit includes a carry-in (0) and a carry-out (1). The sum is 1001111. The diagram uses red lines for the circuit structure and green text for the carry values.

Carry	Input 1	Input 2	Sum
0	1	1	0
1	0	1	0
1	1	0	0
1	1	1	1

Final Sum: 1001111

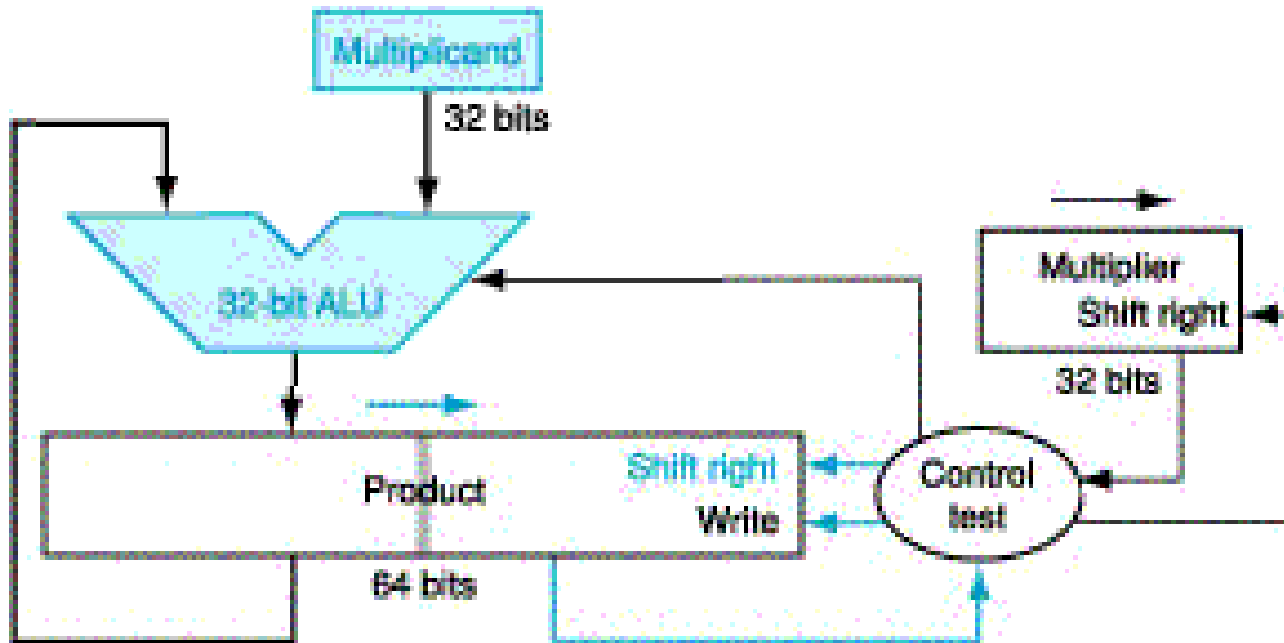
# HW Algorithm 1



In every step

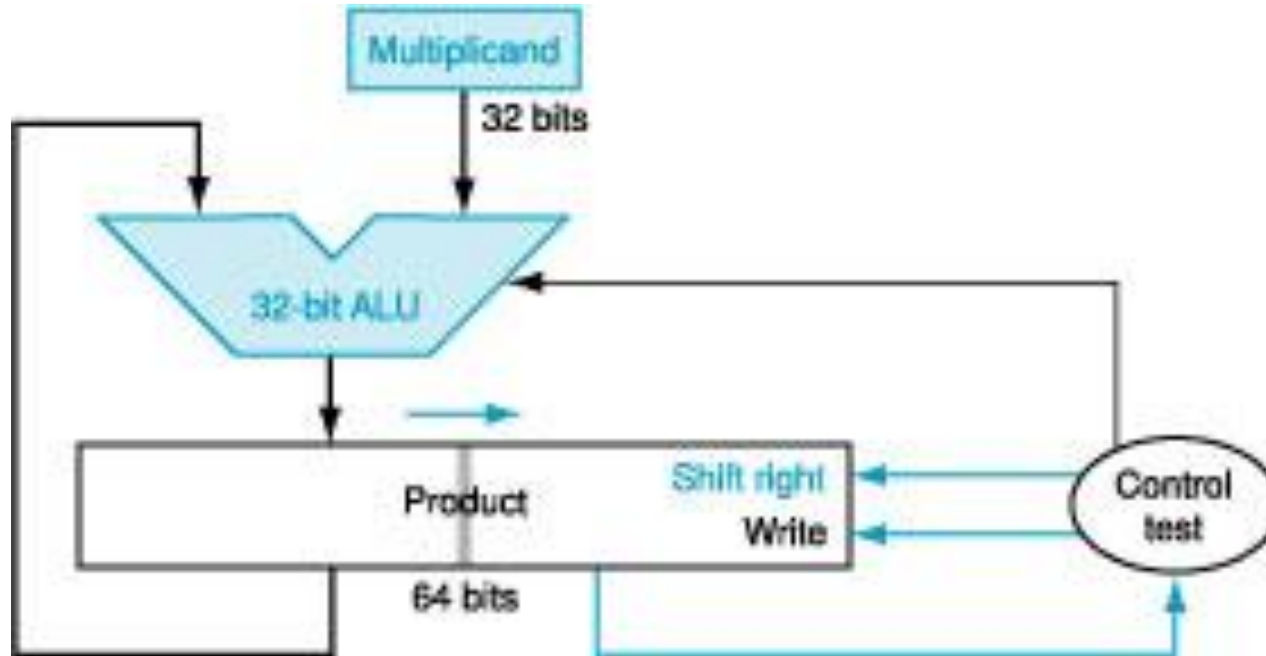
- multiplicand is shifted
- next bit of multiplier is examined (also a shifting step)
- if this bit is 1, shifted multiplicand is added to the product

# HW Algorithm 2



- 32-bit ALU and multiplicand is untouched
- the sum keeps shifting right
- at every step, number of bits in product + multiplier = 64, hence, they share a single 64-bit register

# HW Algorithm 3



- 32-bit ALU and multiplicand is untouched
- the sum keeps shifting right
- at every step, number of bits in product + multiplier = 64, hence, they share a single 64-bit register

# Notes

- The previous algorithm also works for signed numbers (negative numbers in 2's complement form)
- We can also convert negative numbers to positive, multiply the magnitudes, and convert to negative if signs disagree
- The product of two 32-bit numbers can be a 64-bit number
  - -- hence, in MIPS, the product is saved in two 32-bit registers

# Signed number in 2's complement

- Since:  
$$Code = f(Data)$$
$$Data = g(Code)$$

- $X_{code}, Y_{code} \rightarrow (x*y)_{code}$ 
  - We should:  $f(g(X_{code}) * g(Y_{code}))$

- 2's complement:

- $X_{code} = (2^N + X) \bmod 2^N$

- For any number  $a$ :

$$aX_{code} = a2^N + ax \bmod 2^N = (ax)_{code}$$

- So:

$$(X*Y)_{code} = X*Y_{code} \rightarrow g(X_{code}) * Y_{code}$$



# Signed number in 2's complement

- For 32-bit:  $X_{\text{code}} = a_{31}a_{30}a_{29}\dots a_1a_0$   $\{a: 0/1\}$
- The value  $X = -a_{31}2^{31} + a_{30}2^{30} + a_{29}2^{29} + \dots + a_12^1 + a_02^0$
- $(X*Y)_{\text{code}} = X*Y_{\text{code}}$   

$$= (-a_{31}2^{31} + a_{30}2^{30} + a_{29}2^{29} + \dots + a_12^1 + a_02^0) * Y_{\text{code}}$$

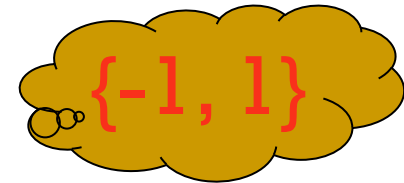
$$= \{ (-a_{31} + a_{30})2^{31}$$

$$+ (-a_{30} + a_{29})2^{30}$$

$$+ \dots$$

$$+ (-a_1 + a_0)2^1$$

$$+ (-a_0 + 0)2^0 \} * Y_{\text{code}}$$



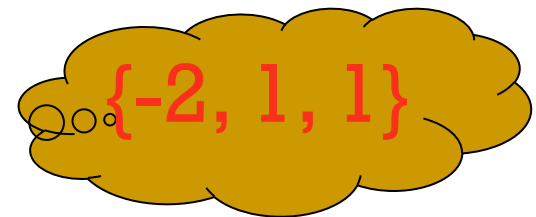
■ Here:  $a_{30}2^{30} = 2a_{30}2^{30} - a_{30}2^{30} = a_{30}2^{31} - a_{30}2^{30}$

# Signed number in 2's complement

■ For 32-bit:  $X_{\text{code}} = a_{31}a_{30}a_{29}\dots a_1a_0$  {a:0/1}

■ The value  $X = -a_{31}2^{31} + a_{30}2^{30} + a_{29}2^{29} + \dots + a_12^1 + a_02^0$

$$\begin{aligned}
 (X*Y)_{\text{code}} &= X*Y_{\text{code}} \\
 &= (-a_{31}2^{31} + a_{30}2^{30} + a_{29}2^{29} + \dots + a_12^1 + a_02^0) * Y_{\text{code}} \\
 &= (-2a_{31}2^{30} + a_{30}2^{30} + a_{29}2^{30} - 2a_{29}2^{28} + a_{28}2^{28} + a_{27}2^{28} - \dots \\
 &\quad - 2a_32^2 + a_22^2 + a_12^2 - 2a_12^0 + a_02^0 + 0*2^0) * Y_{\text{code}} \\
 &= \{ (-2a_{31} + a_{30} + a_{29})2^{30} \\
 &\quad + (-2a_{29} + a_{28} + a_{27})2^{28} \\
 &\quad + \dots \\
 &\quad + (-2a_3 + a_2 + a_1)2^2 \\
 &\quad + (-2a_1 + a_0 + 0)2^0 \} * Y_{\text{code}}
 \end{aligned}$$



■ Here:  $a_{29}2^{29} = 2a_{29}2^{29} - a_{29}2^{29} = a_{29}2^{30} - 2a_{29}2^{28}$

例:  $A = 1011$ ,  $B = -1101$

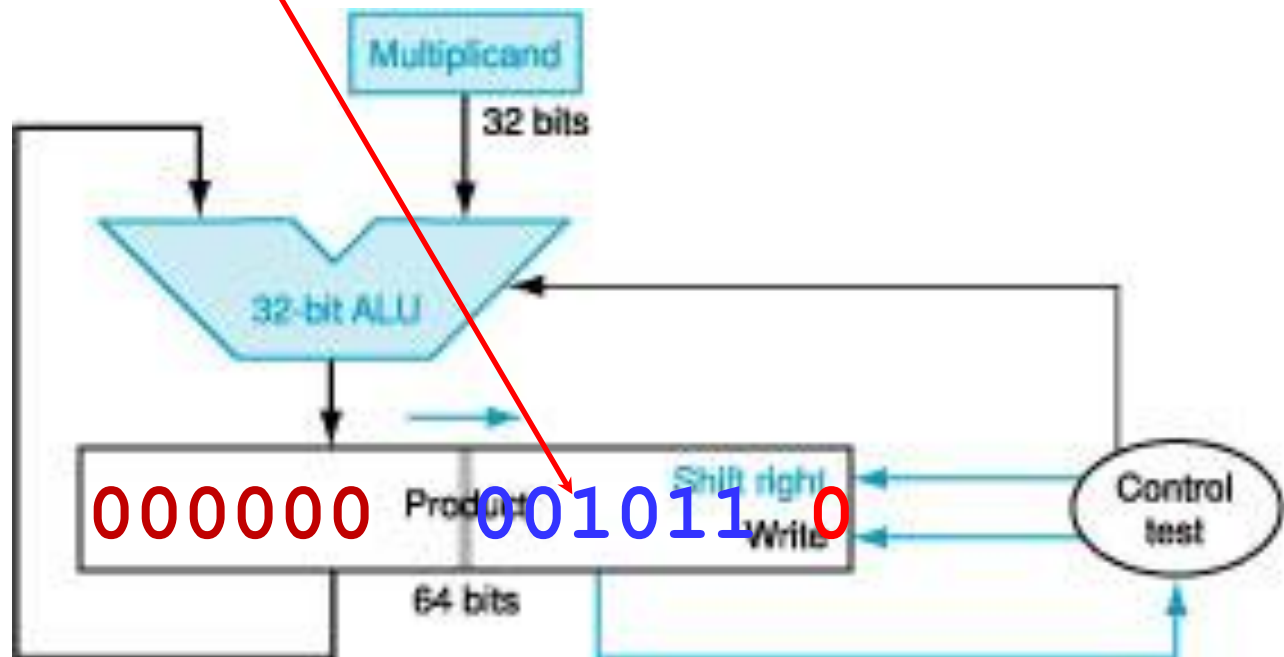
解:  $A_{补} = 001011$

$B_{补} = 110011$

$-B_{补} = 001101$

$2B_{补} = 100110$

$-2B_{补} = 011010$



例:  $A = 1011$ ,  $B = -1101$

解:  $A_{\#} = 001011$

$B_{\#} = 110011$

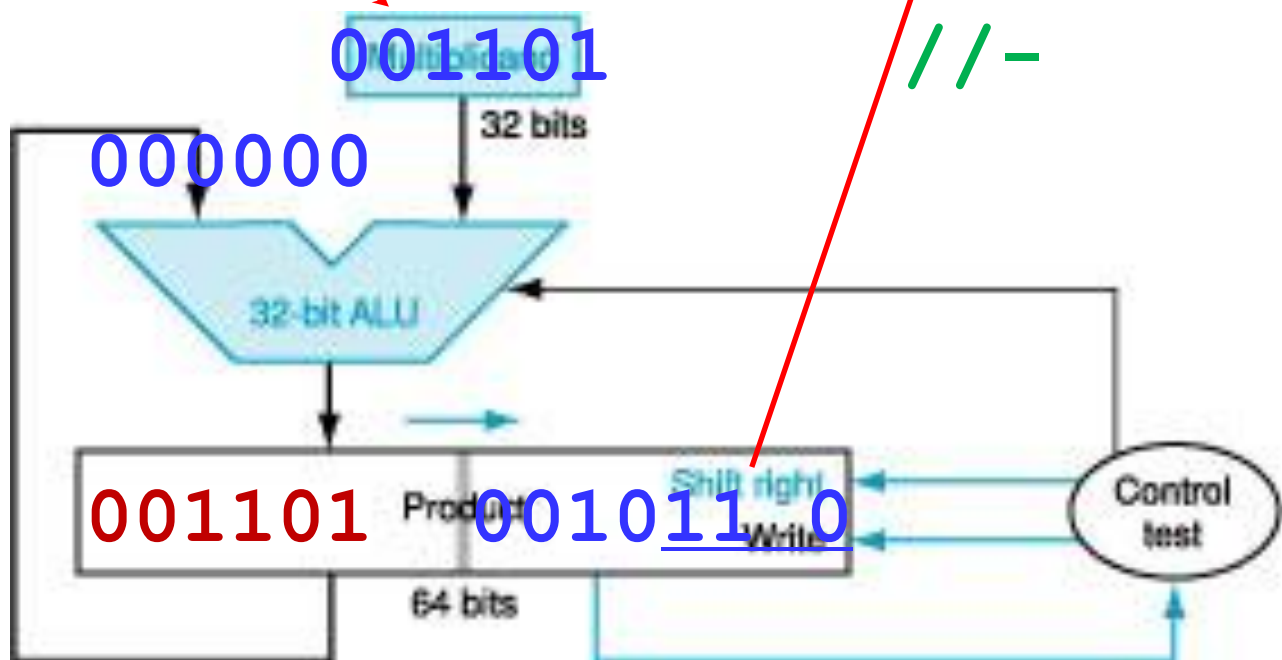
$-B_{\#} = 001101$

$2B_{\#} = 100110$

$-2B_{\#} = 011010$

$\{-2, 1, 1\}$

// -



例:  $A = 1011$ ,  $B = -1101$

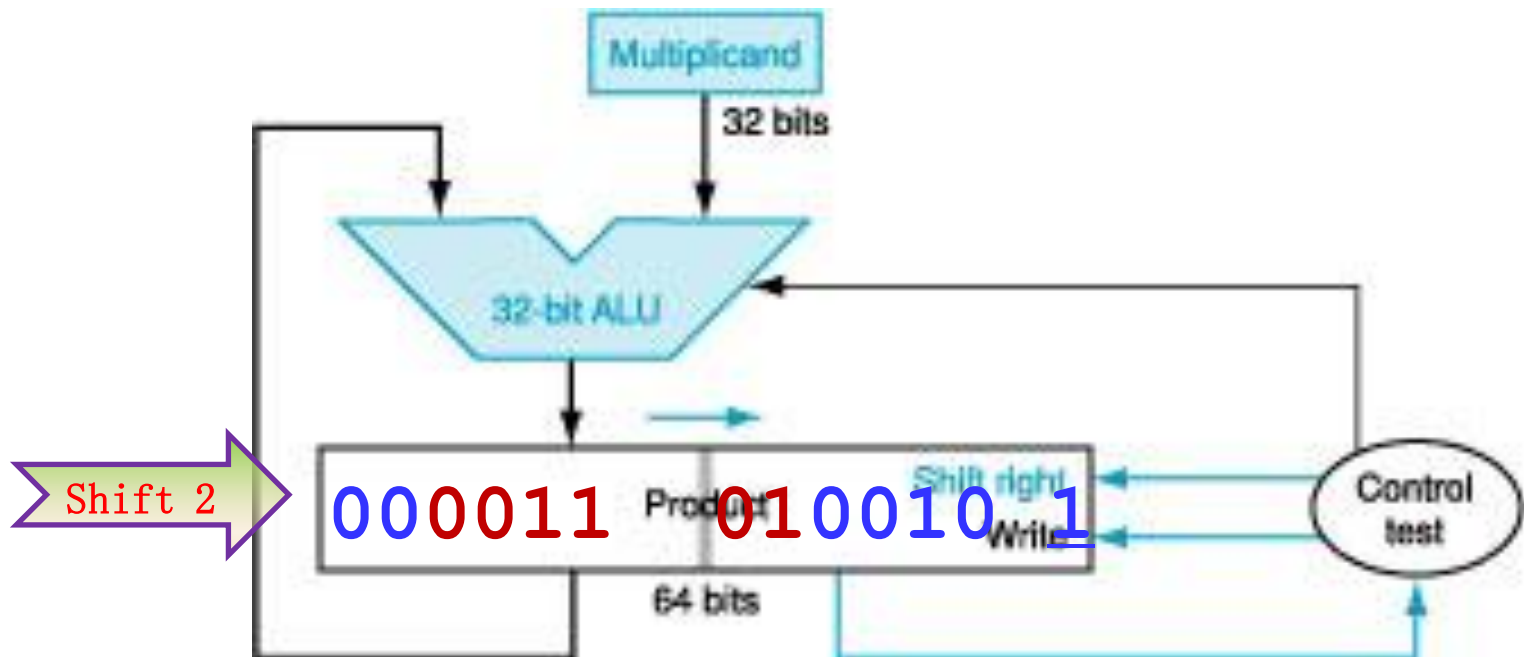
解:  $A_{补} = 001011$

$B_{补} = 110011$

$-B_{补} = 001101$

$2B_{补} = 100110$

$-2B_{补} = 011010$



例:  $A = 1011$ ,  $B = -1101$

解:  $A_{\#} = 001011$

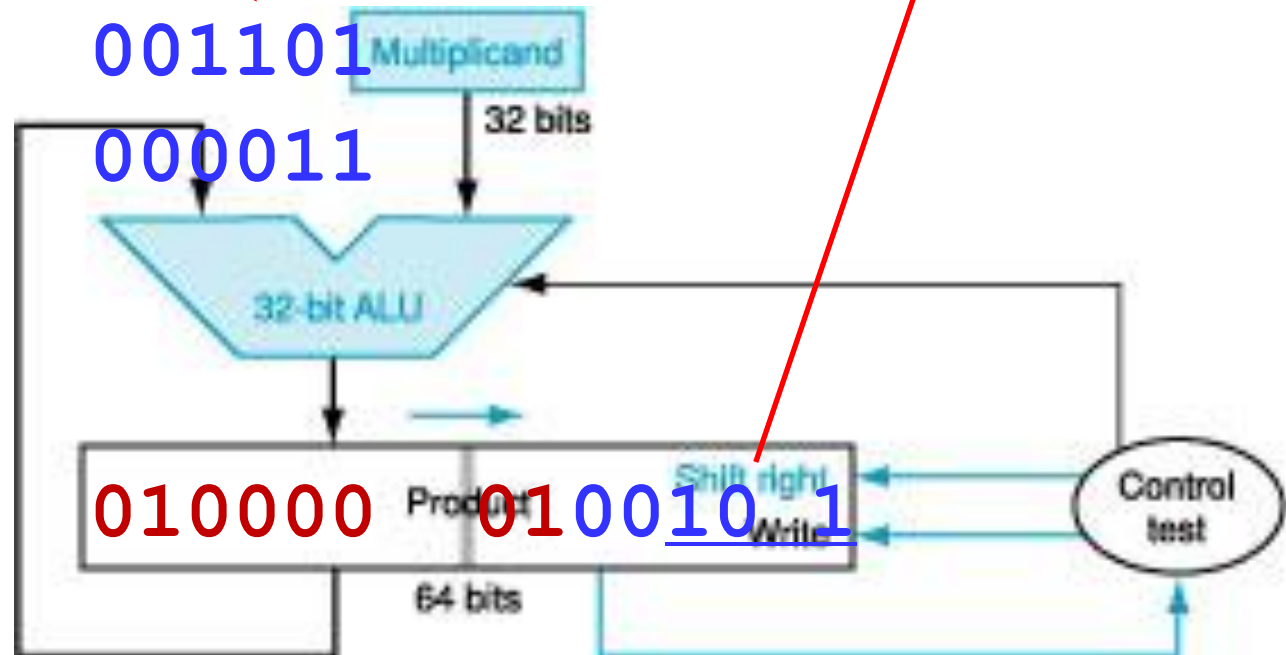
$B_{\#} = 110011$

$-B_{\#} = 001101$

$2B_{\#} = 100110$

$-2B_{\#} = 011010$

$\{-2, 1, 1\}$



例:  $A = 1011$ ,  $B = -1101$

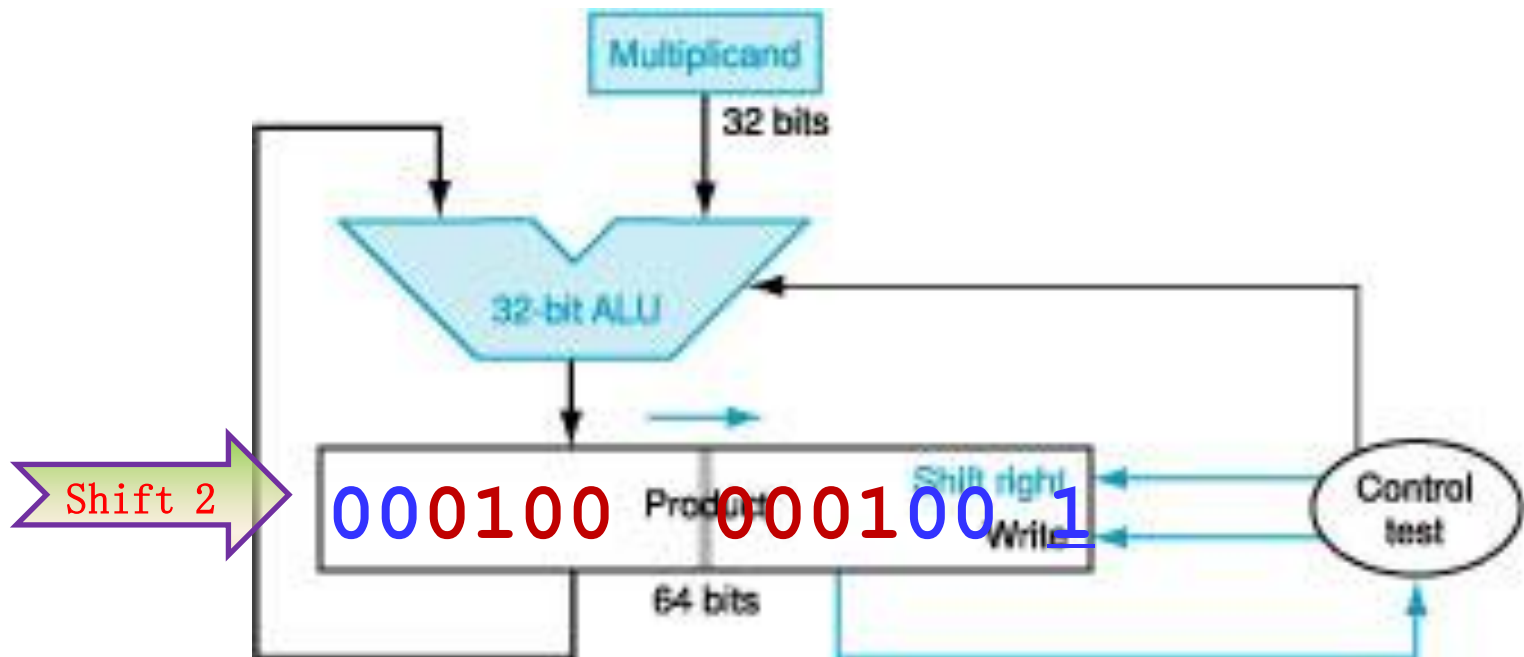
解:  $A_{\#} = 001011$

$B_{\#} = 110011$

$-B_{\#} = 001101$

$2B_{\#} = 100110$

$-2B_{\#} = 011010$



例:  $A = 1011$ ,  $B = -1101$

解:  $A_{\#} = 001011$

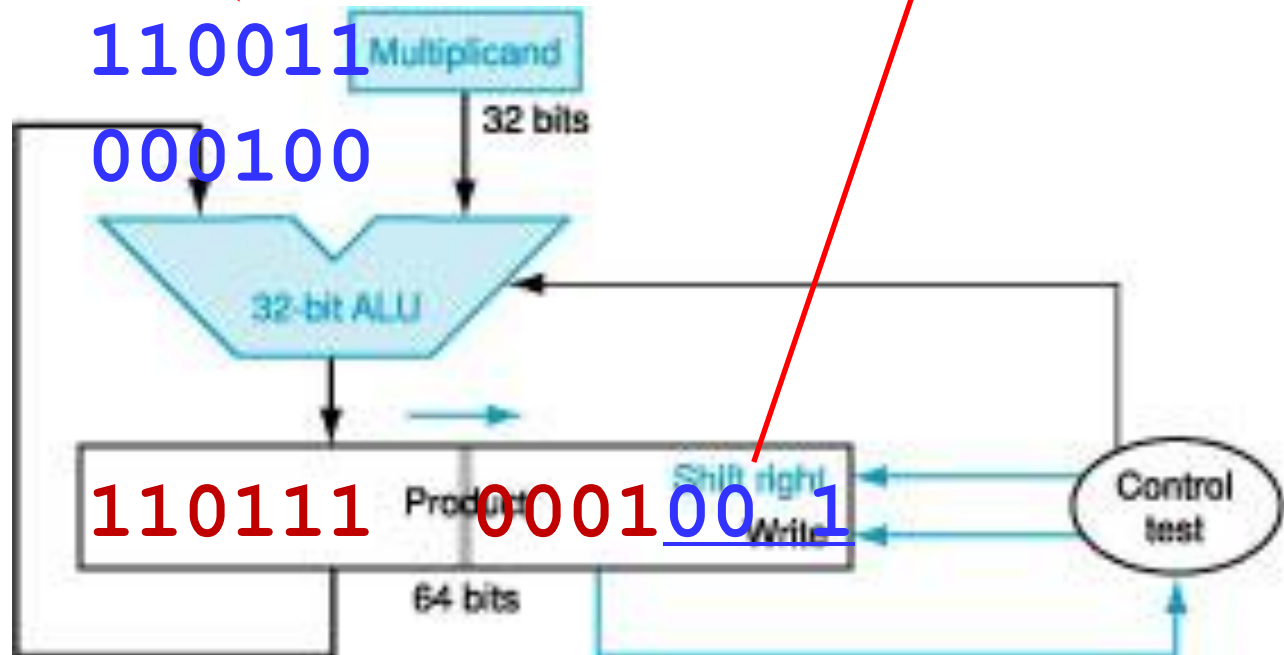
$B_{\#} = 110011$

$-B_{\#} = 001101$

$2B_{\#} = 100110$

$-2B_{\#} = 011010$

$\{-2, 1, 1\}$





例:  $A = 1011$ ,  $B = -1101$

解:  $A_{补} = 001011$

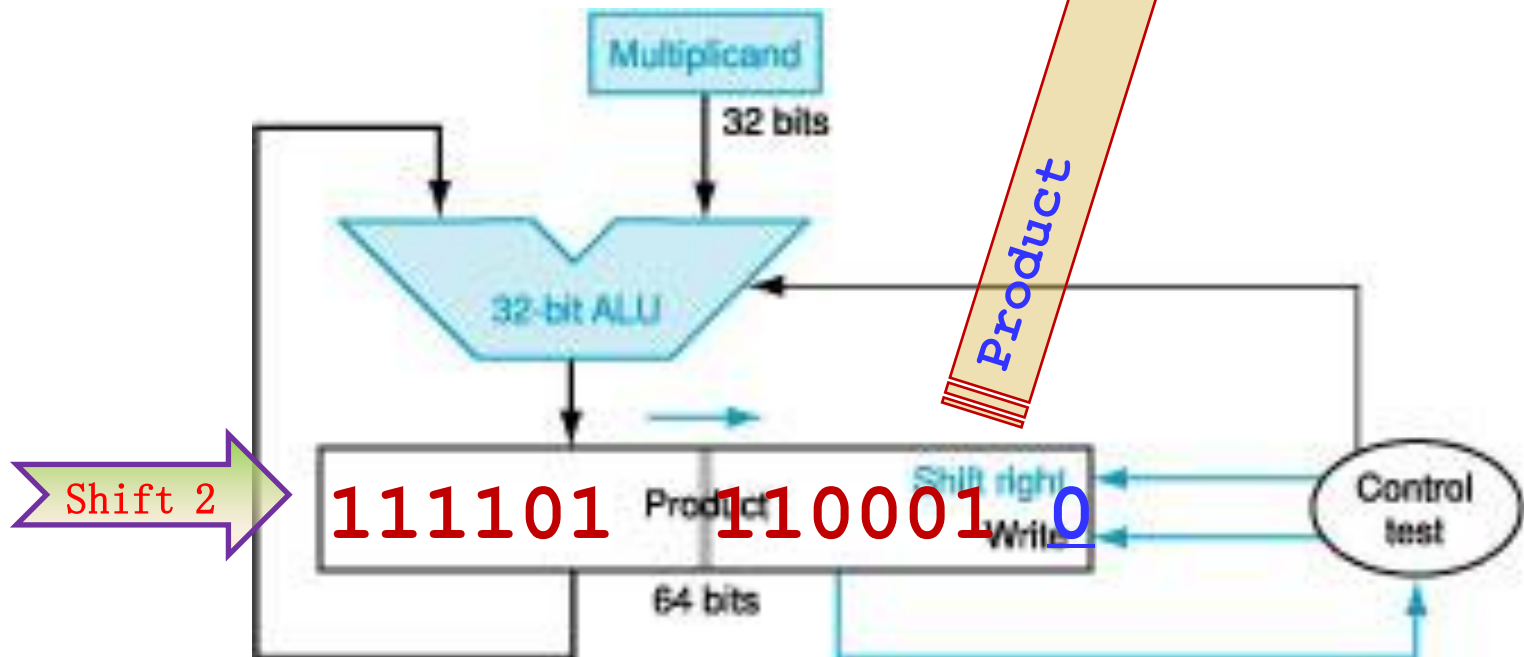
$B_{补} = 110011$

$-B_{补} = 001101$

$2B_{补} = 100110$

$-2B_{补} = 011010$

**Product:**  
**111101110001**



# 1-bit Booth's Algorithm

例:  $a=1101$   $b=1011$  求:  $a*b$

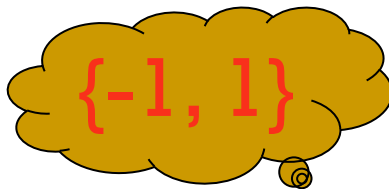
解:

$a_{\#}=01101$

$b_{\#}=01011$

$-b_{\#}=10101$

	00000	01101	0
-b	10101		
	10101	01101	
->	11010	10110	1
+b	01011		
	00101	10110	
->	00010	11011	0
-b	10101		
	10111	11011	
->	11011	11101	1
->	11101	11110	1
+b	01011		
	01000	11110	
->	00100	01111	0



# 2-bit Booth's Algorithm

- For 32-bit:  $X_{\text{code}} = a_{31}a_{30}a_{29}...a_1a_0$  {a: 0/1}
- The value  $X = -a_{31}2^{31} + a_{30}2^{30} + a_{29}2^{29} + ... + a_12^1 + a_02^0$

例:  $A = 1011$ ,  $B = -1101$  求:  $A * B$

解:

$A_{\#} = 001011$	
$B_{\#} = 110011$	000000 001011 0
$-B_{\#} = 001101$	-b 001101
$2B_{\#} = 100110$	001101 001011
$-2B_{\#} = 011010$	-> 000011 010010 1
	-b 001101
	010000 010010
	-> 000100 000100 1
	+b 110011
	110111 000100
	-> 111101 110001

$\{-2, 1, 1\}$

2-bit Booth's  
Algorithm

# MIPS Instructions: MUL

Mult                \$s2, \$s3

- #computes the product and stores it in two "internal" registers that can be referred to as *hi* and *lo*

Mfhi                \$s0

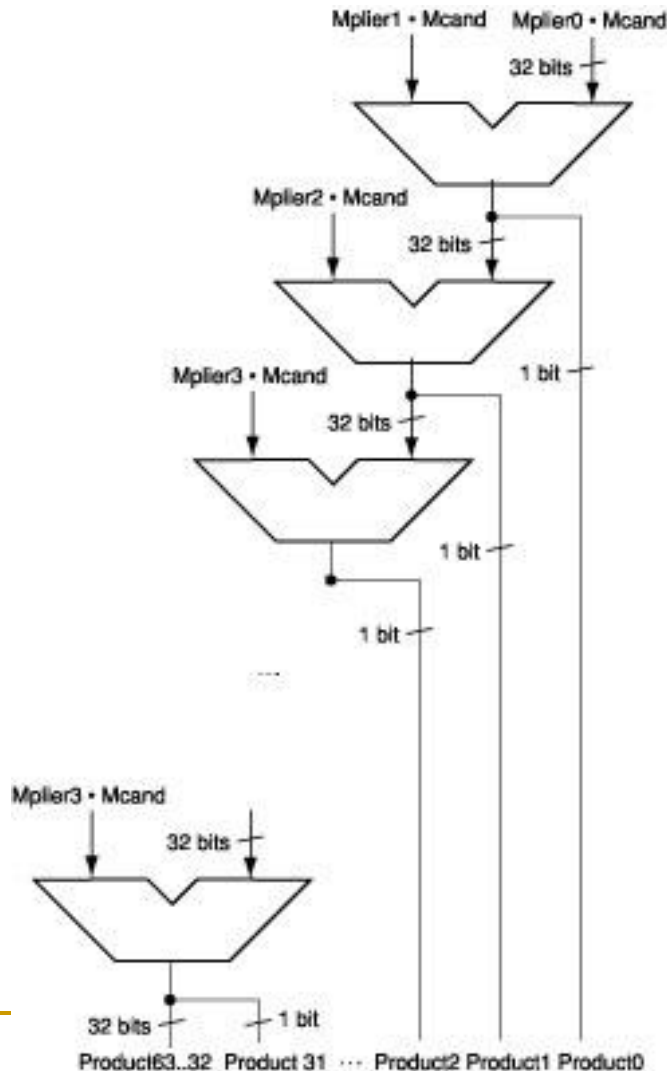
- #moves the value in *hi* into \$s0

Mflo                \$s1

- #moves the value in *lo* into \$s1

- *Similarly for multu*

# Fast Algorithm

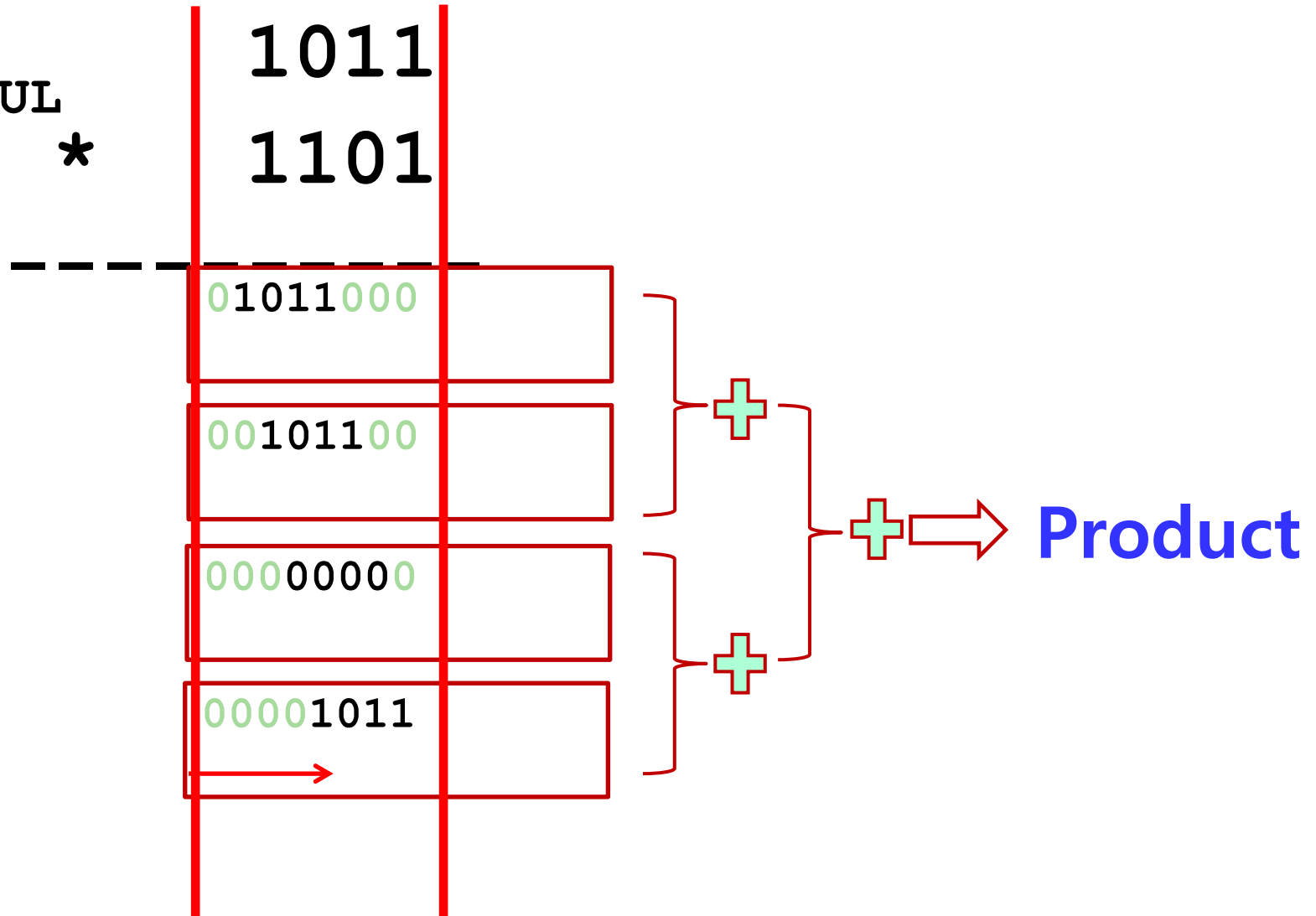


- The previous algorithm requires a clock to ensure that the earlier addition has completed before shifting
- This algorithm can quickly set up most inputs – it then has to wait for the result of each add to propagate down – faster because no clock is involved

-- Note: high transistor cost

# MUL

■ MUL  
\*



# Fast Algorithm

- 设:  $m_{ij} = a_i * b_j$

$a_i$	0	0	1	1
$b_j$	0	1	0	1
$m_{ij}$	0	0	0	1

$a_3 \ a_2 \ a_1 \ a_0$   
 $b_3 \ b_2 \ b_1 \ b_0$

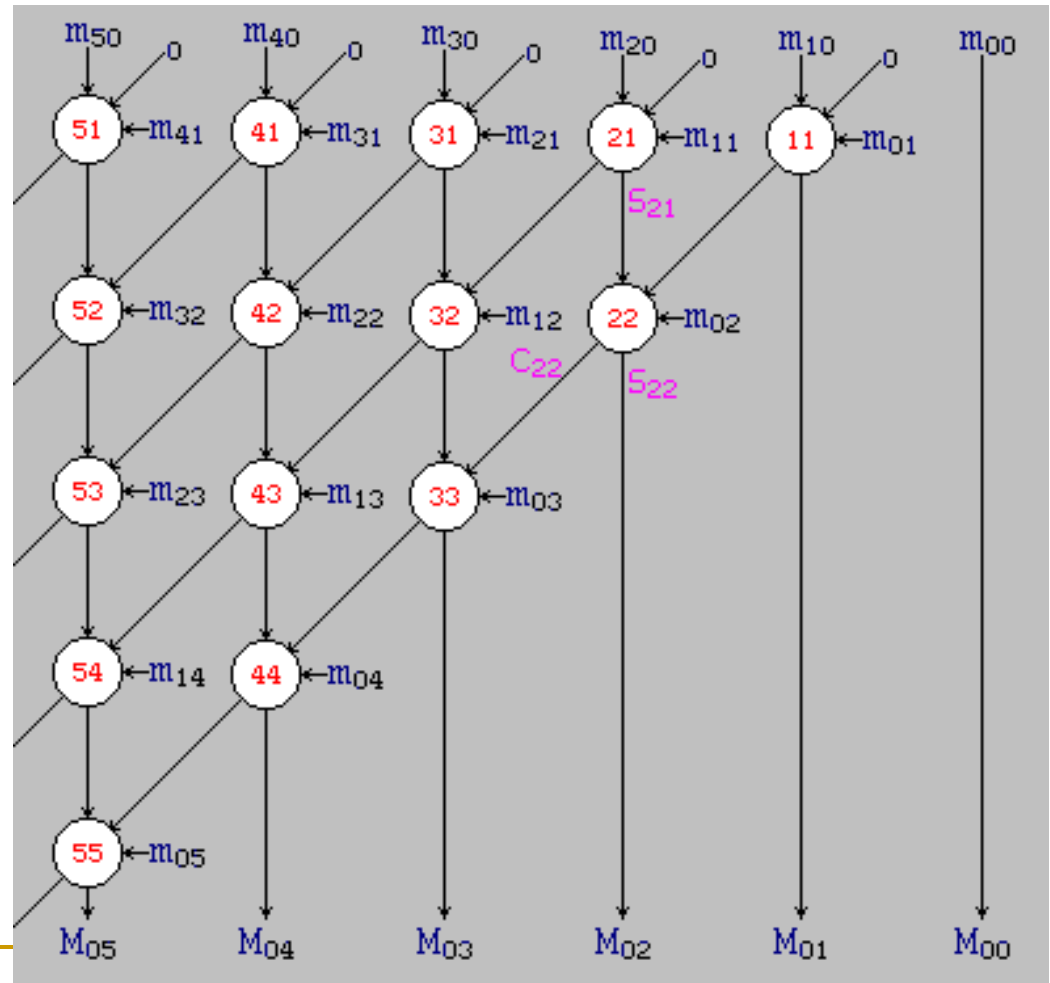
$m_{30} \ m_{20} \ m_{10} \ m_{00}$

$m_{31} \ m_{21} \ m_{11} \ m_{01}$

$m_{32} \ m_{22} \ m_{12} \ m_{02}$

$m_{33} \ m_{23} \ m_{13} \ m_{03}$

$M_{07} \ M_{06} \ M_{05} \ M_{04} \ M_{03} \ M_{02} \ M_{01} \ M_{00}$



设:  $m_{ij} = a_i * b_j$

$a_7 \quad a_6 \quad a_5 \quad a_4 \quad a_3 \quad a_2 \quad a_1 \quad a_0$   
 $b_7 \quad b_6 \quad b_5 \quad b_4 \quad b_3 \quad b_2 \quad b_1 \quad b_0$

$$\begin{array}{cccccccc}
 & & & & m_{70} & m_{60} & m_{50} & m_{40} & m_{30} & m_{20} & m_{10} & m_{00} \\
 & & & m_{71} & m_{61} & m_{51} & m_{41} & m_{31} & m_{21} & m_{11} & m_{01} \\
 & & m_{72} & m_{62} & m_{52} & m_{42} & m_{32} & m_{22} & m_{12} & m_{02} \\
 & m_{73} & m_{63} & m_{53} & m_{43} & m_{33} & m_{23} & m_{13} & m_{03} \\
 & & m_{74} & m_{64} & m_{54} & m_{44} & m_{34} & m_{24} & m_{14} & m_{04} \\
 & & & m_{75} & m_{65} & m_{55} & m_{45} & m_{35} & m_{25} & m_{15} & m_{05} \\
 & & & & m_{76} & m_{66} & m_{56} & m_{46} & m_{36} & m_{26} & m_{16} & m_{06} \\
 + & & & & & m_{77} & m_{67} & m_{57} & m_{47} & m_{37} & m_{27} & m_{17} & m_{07}
 \end{array}$$

$M_{15} \quad M_{14} \quad M_{13} \quad M_{12} \quad M_{11} \quad M_{10} \quad M_{09} \quad M_{08} \quad M_{07} \quad M_{06} \quad M_{05} \quad M_{04} \quad M_{03} \quad M_{02} \quad M_{01} \quad M_{00}$



# Division

			$1001_{\text{ten}}$	Quotient
Divisor	$1000_{\text{ten}}$		$1001010_{\text{ten}}$	Dividend
			<u><math>-1000</math></u>	
			10	
			101	
			1010	
			<u><math>-1000</math></u>	
			$10_{\text{ten}}$	Remainder

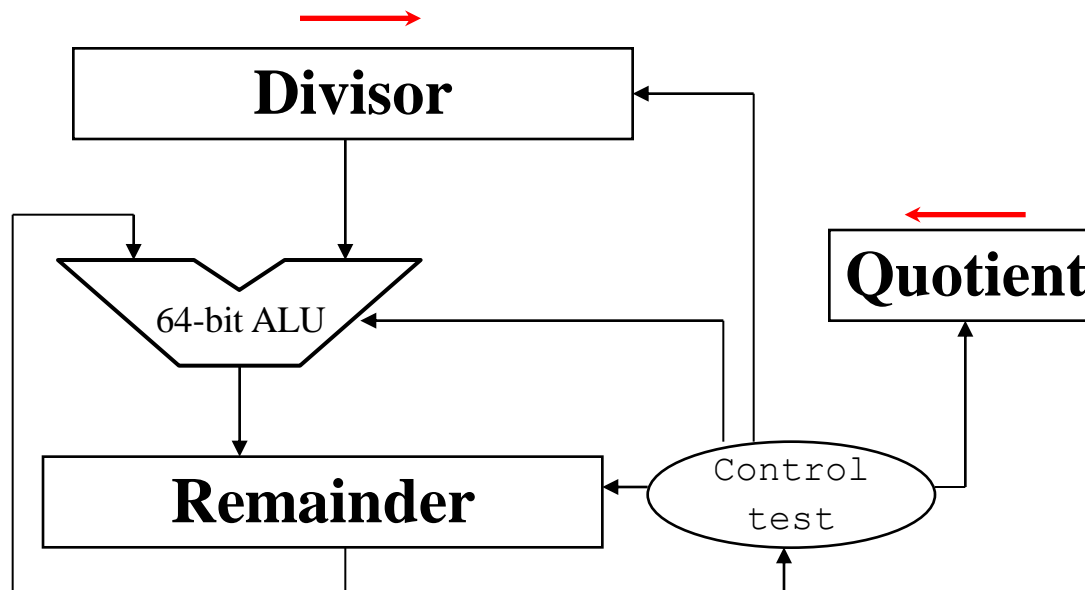
At every step,

- shift divisor right and compare it with current dividend
- if divisor is larger, shift 0 as the next bit of the quotient
- if divisor is smaller, subtract to get new dividend and shift 1 as the next bit of the quotient

# Division

## ■ Division:

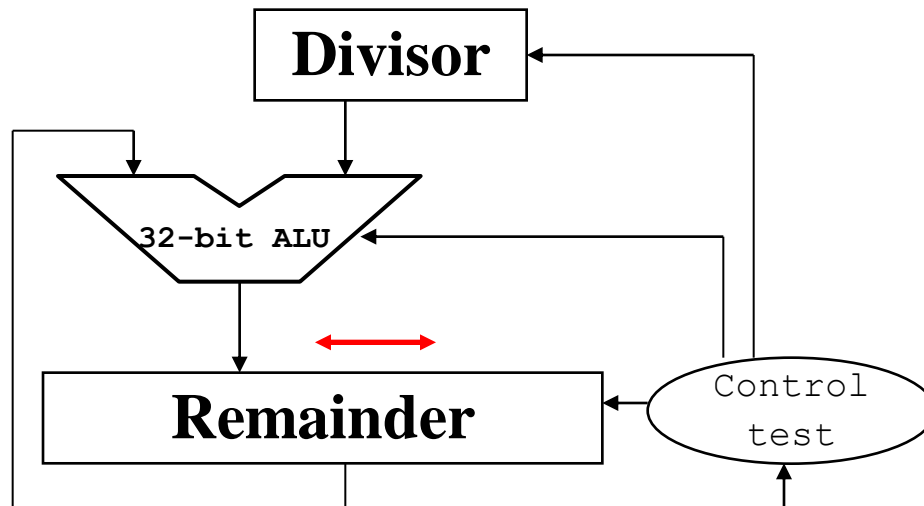
$$\text{Dividend} = \text{Quotient} * \text{Divisor} + \text{Remainder}$$



# Division

## ■ Division:

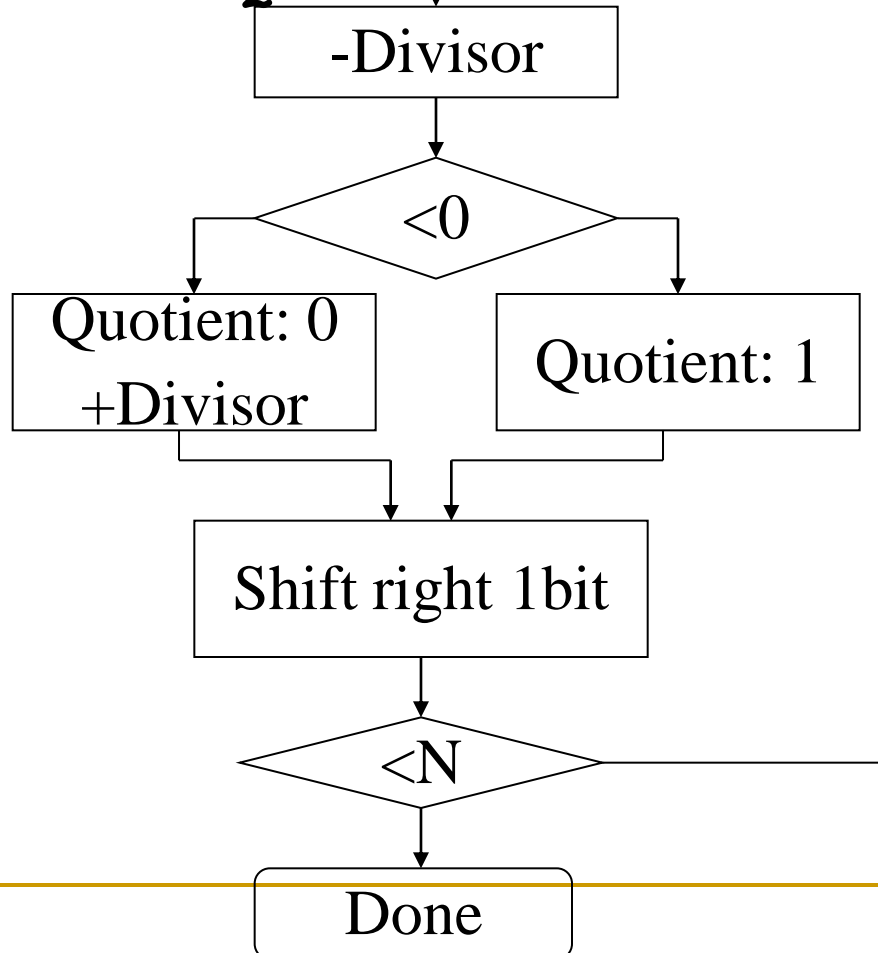
$$\text{Dividend} = \text{Quotient} * \text{Divisor} + \text{Remainder}$$



# Division

## ■ Division:

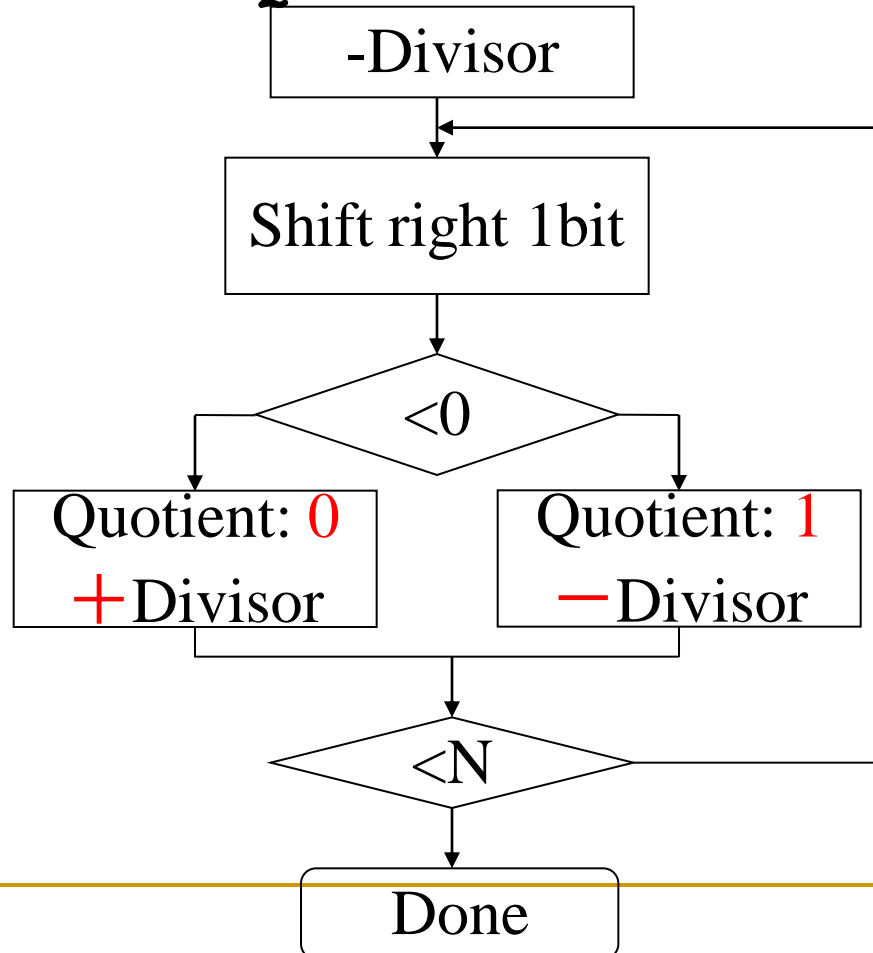
$$\text{Dividend} = \text{Quotient} * \text{Divisor} + \text{Remainder}$$



# Division

## ■ Division:

$$\text{Dividend} = \text{Quotient} * \text{Divisor} + \text{Remainder}$$



例:  $0.1001/0.1011=0.1101\dots0.0001$

$x=0.1001 \rightarrow 001001$

$y=0.1011 \rightarrow 001011$

$[-y]_{\text{补}} = 110101$

加减交替:

	001001		
-y	110101		
-----			
	1111100		
<0	001011	//0	+y
-----			
	0001110		
>0	110101	//01	-y
-----			
	0000110		
>0	110101	//011	-y
-----			
	1110110		
<0	001011	//0110	+y
-----			
	000001(余)		
>0		//01101(商)	

# Division

		$\overline{1001}_{\text{ten}}$	Quotient	
Divisor	$1000_{\text{ten}}$	$  1001010_{\text{ten}}$	Dividend	
	0001001010	0001001010	0000001010	0000001010
	100000000000 $\rightarrow$	0001000000 $\rightarrow$	0000100000 $\rightarrow$	0000001000
Quo: 0		000001	0000010	000001001

At every step,

- shift divisor right and compare it with current dividend
- if divisor is larger, shift 0 as the next bit of the quotient
- if divisor is smaller, subtract to get new dividend and shift 1 as the next bit of the quotient

# Divide Example

- Divide  $7_{\text{ten}}$  ( $0000\ 0111_{\text{two}}$ ) by  $2_{\text{ten}}$  ( $0010_{\text{two}}$ )

Iter	Step	Quot	Divisor	Remainder
0	Initial values			
1				
2				
3				
4				
5				

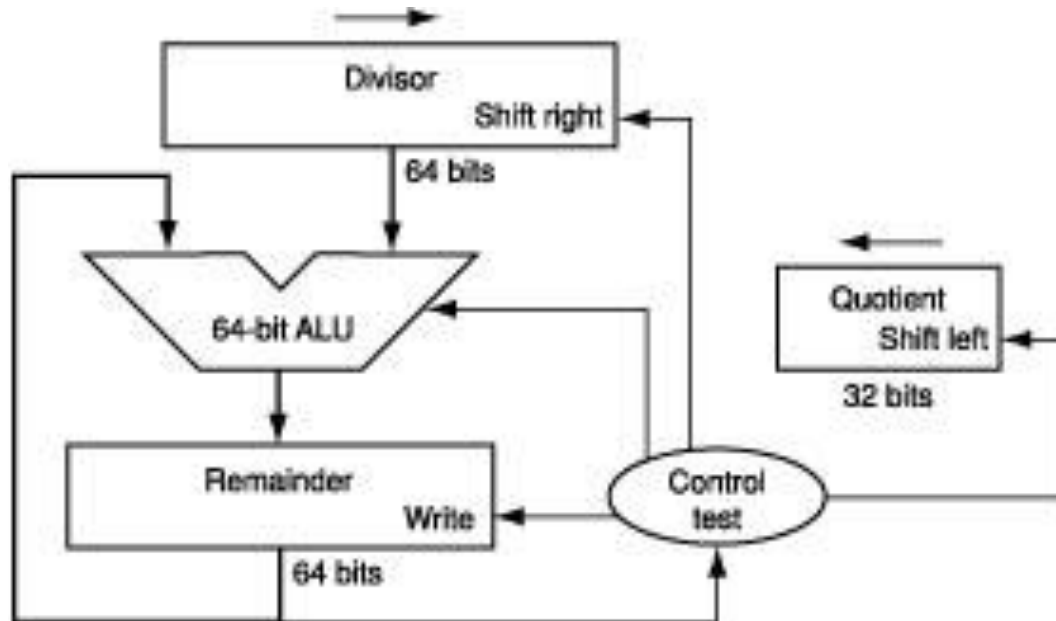


# Divide Example

- Divide  $7_{\text{ten}}$  ( $0000\ 0111_{\text{two}}$ ) by  $2_{\text{ten}}$  ( $0010_{\text{two}}$ )

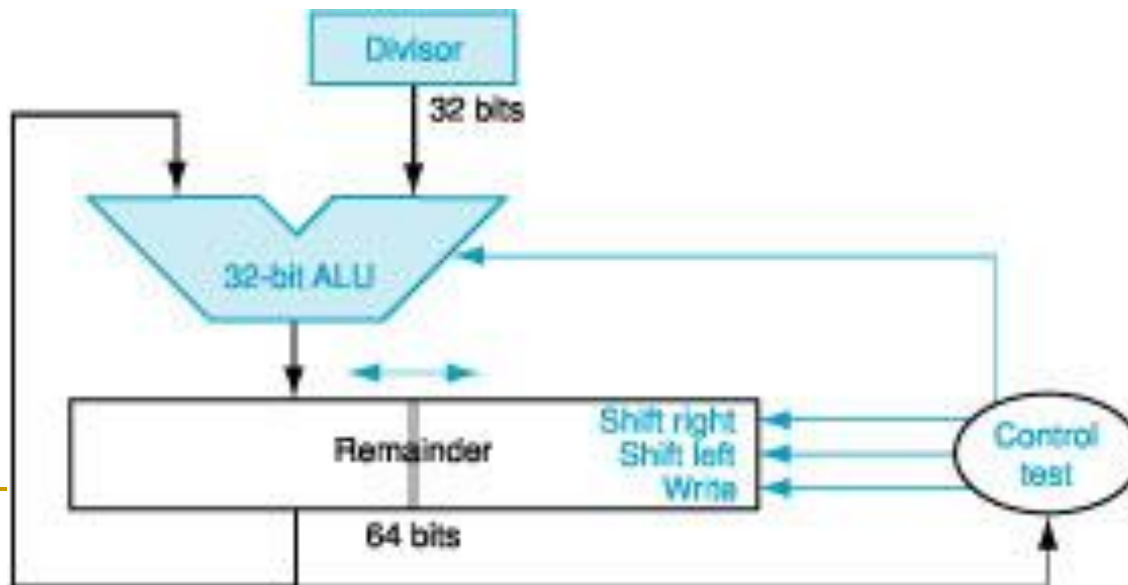
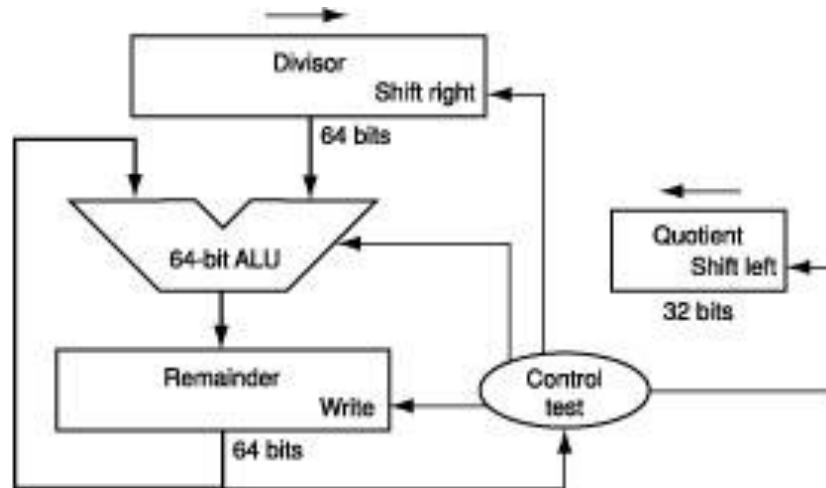
Iter	Step	Quot	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	Rem = Rem – Div	0000	0010 0000	1110 0111
	Rem < 0 $\rightarrow$ +Div, shift 0 into Q	0000	0010 0000	0000 0111
	Shift Div right	0000	0001 0000	0000 0111
2	Same steps as 1	0000	0001 0000	1111 0111
		0000	0001 0000	0000 0111
		0000	0000 1000	0000 0111
3	Same steps as 1	0000	0000 0100	0000 0111
4	Rem = Rem – Div	0000	0000 0100	0000 0011
	Rem $\geq$ 0 $\rightarrow$ shift 1 into Q	0001	0000 0100	0000 0011
	Shift Div right	0001	0000 0010	0000 0011
5	Same steps as 4	0011	0000 0001	0000 0001

# Hardware for Division



A comparison requires a subtract; the sign of the result is examined; if the result is negative, the divisor must be added back

# Efficient Division



# Divisions involving Negatives

- Simplest solution: convert to positive and adjust sign later
- Note that multiple solutions exist for the equation:  
$$\text{Dividend} = \text{Quotient} \times \text{Divisor} + \text{Remainder}$$

+7	div	+2	Quo =	Rem =
-7	div	+2	Quo =	Rem =
+7	div	-2	Quo =	Rem =
-7	div	-2	Quo =	Rem =

# Divisions involving Negatives

- Simplest solution: convert to positive and adjust sign later
- Note that multiple solutions exist for the equation:  
$$\text{Dividend} = \text{Quotient} \times \text{Divisor} + \text{Remainder}$$

+7	div	+2	Quo = +3	Rem = +1
-7	div	+2	Quo = -3	Rem = -1
+7	div	-2	Quo = -3	Rem = +1
-7	div	-2	Quo = +3	Rem = -1

Convention: Dividend and remainder have the same sign  
Quotient is negative if signs disagree  
These rules fulfil the equation above

# CRC

例：将4位有效信息1100编成循环校验码(按(7,4)循环码)：

- 生成多项式：  $G(x)=X^3+X+1$  即：1011。

1011

- 扩展M: 1100 000
- 模2除G:  $1100000 / 1011 = 1110 \dots 010$
- CRC=1100 010。

校验与纠错：

设有位错：1101010

$  \begin{array}{r}  1011 \overline{) 1101010} \\  \underline{1011} \phantom{000} \\  1100 \phantom{00} \\  \underline{1011} \phantom{00} \\  1111 \phantom{00} \\  \underline{1011} \phantom{00} \\  1000 \phantom{00} \\  \underline{1011} \phantom{00} \\  0110 \phantom{00} \\  \phantom{0}1100 \phantom{00} \\  \phantom{0}1011 \phantom{00} \\  \phantom{00}1110 \phantom{00} \\  \phantom{00}1011 \phantom{00} \\  \phantom{000}1010 \phantom{00} \\  \phantom{000}1011 \phantom{00} \\  \phantom{0000}0010 \phantom{00} \\  \phantom{0000}0100 \phantom{00} \\  \phantom{0000}1000 \phantom{00} \\  \phantom{0000}1011 \phantom{00} \\  \phantom{00000}011  \end{array}  $	<p>余数</p> <p>011 循环校验码</p> <p>110 1010101</p> <p>111 0101011</p> <p>101 1010110</p> <p>001 0101100</p> <p>010 1011000</p> <p>100 0110001</p> <p>011 1100010</p>
--	---

# Floating Point

---

- Today's topics:
  - Division
  - IEEE 754 representations
  - FP arithmetic
- Reminder: assignment 4 will be posted later today

# Floating Point

- We need a way to represent numbers with fractions, e.g., 3.1416
  - very small numbers, e.g., .000000001
  - very large numbers, e.g.,  $3.15576 \times 10^9$
- Representation:
  - sign, exponent
  - significand:  $(-1)^{\text{sign}}$
  - more bits for significand gives more accuracy
  - more bits for exponent increases range



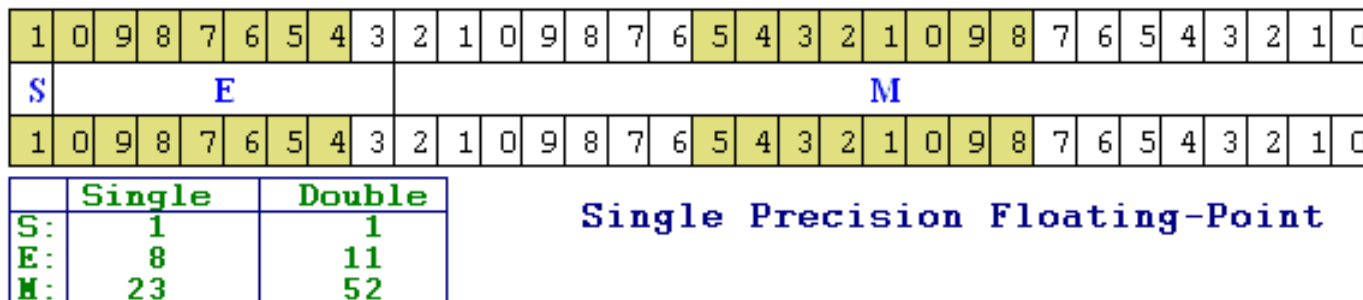


# IEEE 754 Floating Point

- IEEE 754 floating point standard:
  - single precision: 8 bit exponent, 23 bit significand
  - double precision: 11 bit exponent, 52 bit significand

# Floating Point

- IEEE 754 floating point standard:
  - single precision: 8 bit exponent, 23 bit significand
  - double precision: 11 bit exponent, 52 bit significand



Single Precision Floating-Point

# IEEE 754 floating-point standard



- Leading "1" bit of significand is implicit
- Exponent is "biased" to make sorting easier
  - all 0s is smallest exponent all 1s is largest
  - bias of 127 for single precision and 1023 for double precision
  - summary:  $(-1)^{\text{sign}} \cdot (1 + \text{significand}) \cdot 2^{\text{exponent} - \text{bias}}$

# IEEE 754 floating-point standard



## ■ Example:

- decimal:  $-.75 = -3/4 = -3/2^2$
- binary:  $-.11 = -1.1 \times 2^{-1}$
- floating point: exponent = 126 = 01111110
- IEEE single precision:  
10111110100000000000000000000000

■ k

$$x = (-1)^S * 1.M * 2^{E-127}$$

$$\because x = \pm 1.M * 2^e$$

$$S = \begin{cases} 0 & \dots x > 0 \\ 1 & \dots x < 0 \end{cases}$$

$$E = e + 127$$

E	M	X
0	0	0
0	$\neq 0$	X
$0 < E < 255$	any	X
255	0	infinity
255	any	not a number



# Floating Point Complexities

- Operations are somewhat more complicated
- In addition to overflow we can have "underflow"
- Accuracy can be a big problem
  - IEEE 754 keeps two extra bits, guard and round
  - four rounding modes
  - positive divided by zero yields "infinity"
  - zero divide by zero yields "not a number"
  - other complexities



# Floating Point Complexities

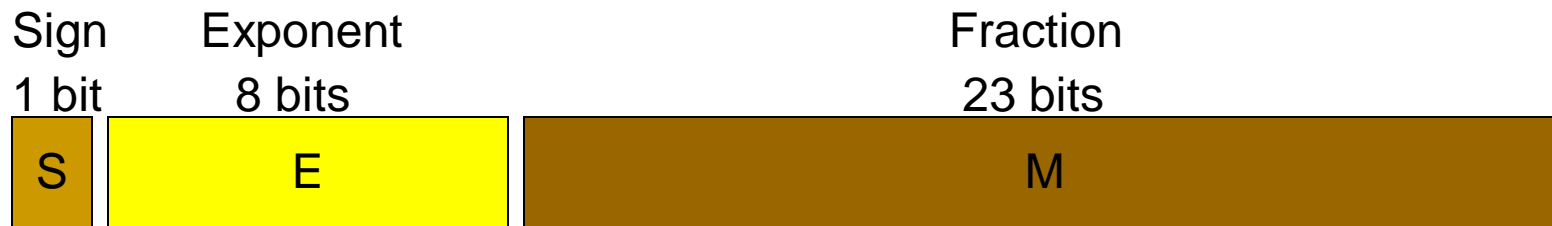
- Implementing the standard can be tricky
- Not using the standard can be even worse
  - see text for description of 80x86 and Pentium bug!

# Floating Point

- Normalized scientific notation: single non-zero digit to the left of the decimal (binary) point – example:  $3.5 \times 10^9$
- $1.010001 \times 2^{-5}_{\text{two}} = (1 + 0 \times 2^{-1} + 1 \times 2^{-2} + \dots + 1 \times 2^{-6}) \times 2^{-5}_{\text{ten}}$
- A standard notation enables easy exchange of data between machines and simplifies hardware algorithms – the IEEE 754 standard defines how floating point numbers are represented



# Sign and Magnitude Representation



- More exponent bits → wider range of numbers (not necessarily more numbers – recall there are infinite real numbers)
- More fraction bits → higher precision
- Register value =  $(-1)^S \times 1.M \times 2^{E-127}$
- Since we are only representing normalized numbers, we are guaranteed that the number is of the form 1.xxxx..  
Hence, in IEEE 754 standard, the 1 is implicit

# Sign and Magnitude Representation

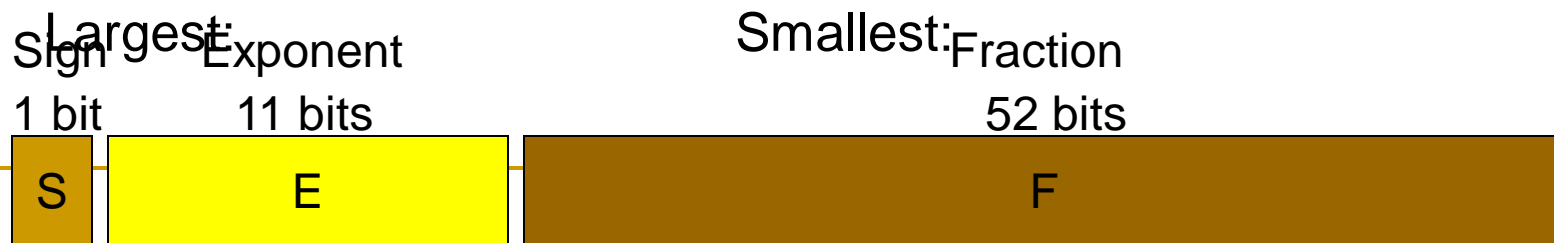


- Largest number that can be represented:
- Smallest number that can be represented:

# Sign and Magnitude Representation



- Largest number that can be represented:  $2.0 \times 2^{128} = 2.0 \times 10^{38}$
- Smallest number that can be represented:  $2.0 \times 2^{-128} = 2.0 \times 10^{-38}$
- Overflow: when representing a number larger than the one above;  
Underflow: when representing a number smaller than the one above
- Double precision format: occupies two 32-bit registers:



# Details

---

- The number “0” has a special code so that the implicit 1 does not get added: the code is all 0s  
(it may seem that this takes up the representation for 1.0, but given how the exponent is represented, we’ll soon see that that’s not the case)
- The largest exponent value (with zero fraction) represents +/- infinity
- The largest exponent value (with non-zero fraction) represents NaN (not a number) – for the result of 0/0 or (infinity minus infinity)

# Exponent Representation

- To simplify sort, sign was placed as the first bit
- For a similar reason, the representation of the exponent is also modified: in order to use integer compares, it would be preferable to have the smallest exponent as 00...0 and the largest exponent as 11...1
- This is the biased notation, where a bias is subtracted from the exponent field to yield the true exponent
- IEEE 754 single-precision uses a bias of 127 (since the exponent must have values between -127 and 128)...double precision uses a bias of 1023

Final representation:  $(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$

# Examples

Final representation:  $(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$

- Represent  $-0.75_{\text{ten}}$  in single and double-precision formats

Single:  $(1 + 8 + 23)$

Double:  $(1 + 11 + 52)$

- What decimal number is represented by the following single-precision number?

1 1000 0001 01000...0000

# Examples

Final representation:  $(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$

- Represent  $-0.75_{\text{ten}}$  in single and double-precision formats

Single: (1 + 8 + 23)

1 0111 1110 1000...000

Double: (1 + 11 + 52)

1 0111 1111 110 1000...000

- What decimal number is represented by the following single-precision number?

1 1000 0001 01000...0000

-5.0

# FP Addition

- Consider the following decimal example (can maintain only 4 decimal digits and 2 exponent digits)

$$9.999 \times 10^1 + 1.610 \times 10^{-1}$$

Convert to the larger exponent:

$$9.999 \times 10^1 + 0.016 \times 10^1$$

Add

$$10.015 \times 10^1$$

Normalize

$$1.0015 \times 10^2$$

Check for overflow/underflow

Round

$$1.002 \times 10^2$$

Re-normalize



# FP Addition

- Consider the following decimal example (can maintain only 4 decimal digits and 2 exponent digits)

$$9.999 \times 10^1 + 1.610 \times 10^{-1}$$

Convert to the larger exponent:

$$9.999 \times 10^1 + 0.016 \times 10^1$$

Add

$$10.015 \times 10^1$$

Normalize

$$1.0015 \times 10^2$$

Check for overflow/underflow

Round

$$1.002 \times 10^2$$

Re-normalize

If we had more fraction bits,  
these errors would be minimized

# FP Multiplication

---

- Similar steps:
  - Compute exponent (careful!)
  - Multiply significands (set the binary point correctly)
  - Normalize
  - Round (potentially re-normalize)
  - Assign sign

# MIPS Instructions

---

- The usual add.s, add.d, sub, mul, div
- Comparison instructions: c.eq.s, c.neq.s, c.lt.s....  
These comparisons set an internal bit in hardware that is then inspected by branch instructions: bc1t, bc1f
- Separate register file \$f0 - \$f31 : a double-precision value is stored in (say) \$f4-\$f5 and is referred to by \$f4
- Load/store instructions (lwc1, swc1) must still use integer registers for address computation

# Code Example

```
float f2c (float fahr)
{
    return ((5.0/9.0) * (fahr – 32.0));
}
```

(argument fahr is stored in \$f12)

```
lwc1  $f16, const5($gp)
lwc1  $f18, const9($gp)
div.s  $f16, $f16, $f18
lwc1  $f18, const32($gp)
sub.s  $f18, $f12, $f18
mul.s  $f0, $f16, $f18
jr     $ra
```

# FP, Performance Metrics

---

- Today's topics:
  - IEEE 754 representations
  - FP arithmetic
  - Evaluating a system
- Reminder: assignment 4 due in a week – start early!

# Examples

Final representation:  $(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$

- Represent  $-0.75_{\text{ten}}$  in single and double-precision formats

Single:  $(1 + 8 + 23)$

Double:  $(1 + 11 + 52)$

- What decimal number is represented by the following single-precision number?

1 1000 0001 01000...0000

# Examples

Final representation:  $(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$

- Represent  $-0.75_{\text{ten}}$  in single and double-precision formats

Single:  $(1 + 8 + 23)$

1 0111 1110 1000...000

Double:  $(1 + 11 + 52)$

1 0111 1111 110 1000...000

- What decimal number is represented by the following single-precision number?

1 1000 0001 01000...0000

-5.0

# FP Addition

- Consider the following decimal example (can maintain only 4 decimal digits and 2 exponent digits)

$$9.999 \times 10^1 + 1.610 \times 10^{-1}$$

Convert to the larger exponent:

$$9.999 \times 10^1 + 0.016 \times 10^1$$

Add

$$10.015 \times 10^1$$

Normalize

$$1.0015 \times 10^2$$

Check for overflow/underflow

Round

$$1.002 \times 10^2$$

Re-normalize



# FP Addition

- Consider the following decimal example (can maintain only 4 decimal digits and 2 exponent digits)

$$9.999 \times 10^1 + 1.610 \times 10^{-1}$$

Convert to the larger exponent:

$$9.999 \times 10^1 + 0.016 \times 10^1$$

Add

$$10.015 \times 10^1$$

Normalize

$$1.0015 \times 10^2$$

Check for overflow/underflow

Round

$$1.002 \times 10^2$$

Re-normalize

If we had more fraction bits,  
these errors would be minimized

# FP Multiplication

---

- Similar steps:
  - Compute exponent (careful!)
  - Multiply significands (set the binary point correctly)
  - Normalize
  - Round (potentially re-normalize)
  - Assign sign

# MIPS Instructions

---

- The usual add.s, add.d, sub, mul, div
- Comparison instructions: c.eq.s, c.neq.s, c.lt.s....  
These comparisons set an internal bit in hardware that is then inspected by branch instructions: bc1t, bc1f
- Separate register file \$f0 - \$f31 : a double-precision value is stored in (say) \$f4-\$f5 and is referred to by \$f4
- Load/store instructions (lwc1, swc1) must still use integer registers for address computation

# Performance Metrics

---

- Possible measures:
  - response time – time elapsed between start and end of a program
  - throughput – amount of work done in a fixed time
- The two measures are usually linked
  - A faster processor will improve both
  - More processors will likely only improve throughput
- What influences performance?

# Execution Time

---

Consider a system  $X$  executing a fixed workload  $W$

$$\text{Performance}_X = 1 / \text{Execution time}_X$$

Execution time = response time = wall clock time

- Note that this includes time to execute the workload as well as time spent by the operating system co-ordinating various events

The UNIX “time” command breaks up the wall clock time as user and system time

# Speedup and Improvement

- System X executes a program in 10 seconds, system Y executes the same program in 15 seconds
- System X is 1.5 times faster than system Y
- The speedup of system X over system Y is 1.5 (the ratio)
- The performance improvement of X over Y is  $1.5 - 1 = 0.5 = 50\%$
- The execution time reduction for the program, compared to Y is  $(15-10) / 15 = 33\%$   
The execution time increase, compared to X is  $(15-10) / 10 = 50\%$



# Performance Equation - I

CPU execution time = CPU clock cycles x Clock cycle time

Clock cycle time =  $1 / \text{Clock speed}$

If a processor has a frequency of 3 GHz, the clock ticks 3 billion times in a second – as we'll soon see, with each clock tick, one or more/less instructions may complete

If a program runs for 10 seconds on a 3 GHz processor, how many clock cycles did it run for?

If a program runs for 2 billion clock cycles on a 1.5 GHz processor, what is the execution time in seconds?



# Performance Equation - II

---

CPU clock cycles = number of instrs x avg clock cycles  
per instruction (CPI)

Substituting in previous equation,

Execution time = clock cycle time x number of instrs x avg CPI

If a 2 GHz processor graduates an instruction every third cycle,  
how many instructions are there in a program that runs for  
10 seconds?



# Factors Influencing Performance

---

Execution time = clock cycle time x number of instrs x avg CPI

- Clock cycle time: manufacturing process (how fast is each transistor), how much work gets done in each pipeline stage (more on this later)
- Number of instrs: the quality of the compiler and the instruction set architecture
- CPI: the nature of each instruction and the quality of the architecture implementation

# Example

---

Execution time = clock cycle time x number of instrs x avg CPI

Which of the following two systems is better?

- A program is converted into 4 billion MIPS instructions by a compiler ; the MIPS processor is implemented such that each instruction completes in an average of 1.5 cycles and the clock speed is 1 GHz
- The same program is converted into 2 billion x86 instructions; the x86 processor is implemented such that each instruction completes in an average of 6 cycles and the clock speed is 1.5 GHz

# Benchmark Suites

- Measuring performance components is difficult for most users: average CPI requires simulation/hardware counters, instruction count requires profiling tools/hardware counters, OS interference is hard to quantify, etc.
- Each vendor announces a SPEC rating for their system
  - a measure of execution time for a fixed collection of programs
  - is a function of a specific CPU, memory system, IO system, operating system, compiler
  - enables easy comparison of different systems

The key is coming up with a collection of relevant programs

# Deriving a Single Performance Number

---

How is the performance of 29 different apps compressed into a single performance number?

- SPEC uses geometric mean (GM) – the execution time of each program is multiplied and the  $N^{\text{th}}$  root is derived
- Another popular metric is arithmetic mean (AM) – the average of each program's execution time
- Weighted arithmetic mean – the execution times of some programs are weighted to balance priorities

# Amdahl's Law

---

- Architecture design is very bottleneck-driven – make the common case fast, do not waste resources on a component that has little impact on overall performance/power
- Amdahl's Law: performance improvements through an enhancement is limited by the fraction of time the enhancement comes into play
- Example: a web server spends 40% of time in the CPU and 60% of time doing I/O – a new processor that is ten times faster results in a 36% reduction in execution time (speedup of 1.56) – Amdahl's Law states that maximum execution time reduction is 40% (max speedup of 1.66)



# Digital Design

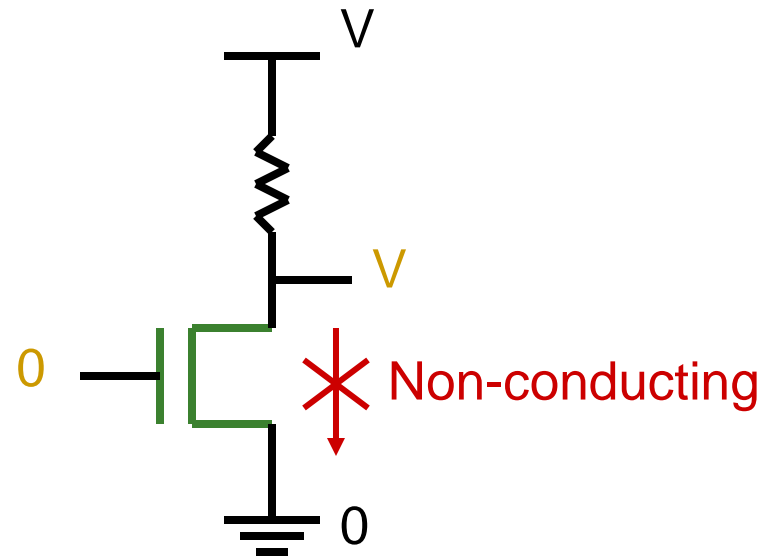
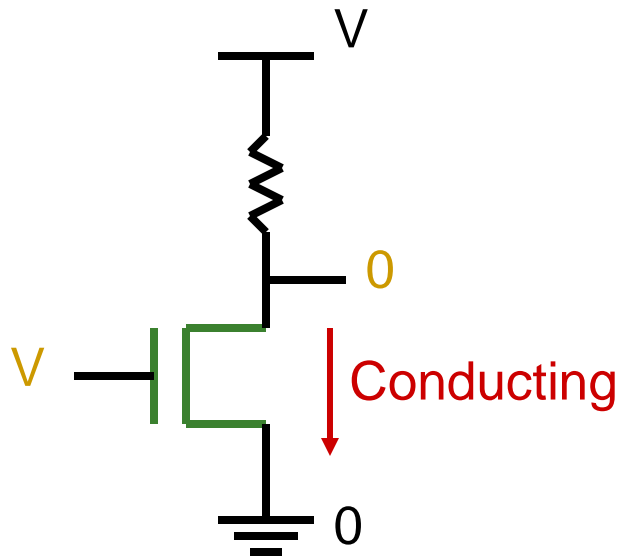
---

- Today's topics:
  - Evaluating a system
  - Intro to boolean functions



# Digital Design Basics

- Two voltage levels – high and low (1 and 0, true and false)  
Hence, the use of binary arithmetic/logic in all computers
- A transistor is a 3-terminal device that acts as a switch



# Logic Blocks

---

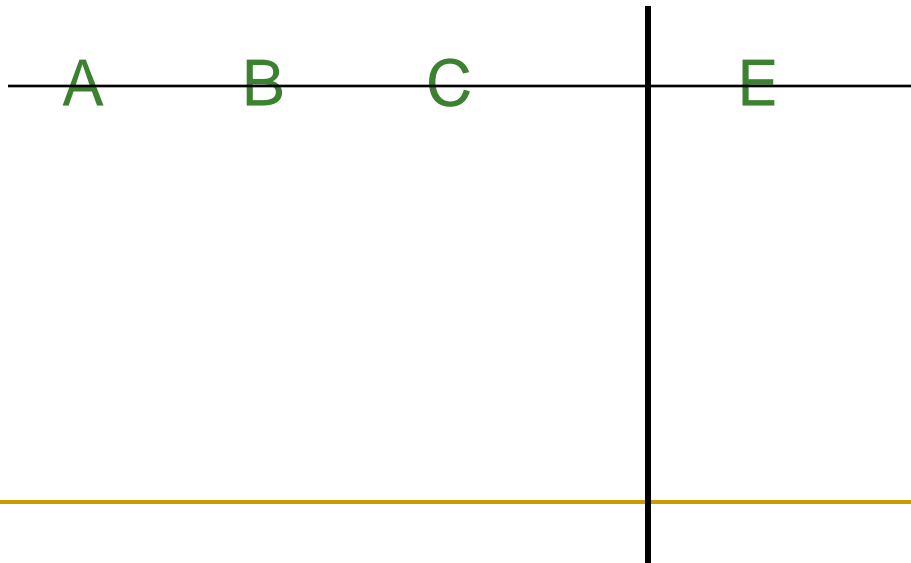
- A logic block has a number of binary inputs and produces a number of binary outputs – the simplest logic block is composed of a few transistors
- A logic block is termed *combinational* if the output is only a function of the inputs
- A logic block is termed *sequential* if the block has some internal memory (state) that also influences the output
- A basic logic block is termed a *gate* (AND, OR, NOT, etc.)

We will only deal with combinational circuits today



# Truth Table

- A truth table defines the outputs of a logic block for each set of inputs
- Consider a block with 3 inputs A, B, C and an output E that is true only if *exactly* 2 inputs are true



# Truth Table

- A truth table defines the outputs of a logic block for each set of inputs
- Consider a block with 3 inputs A, B, C and an output E that is true only if *exactly* 2 inputs are true

A	B	C	E
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Can be compressed by only representing cases that have an output of 1

# Boolean Algebra

- Equations involving two values and three primary operators:
  - OR : symbol  $+$  ,  $X = A + B \Rightarrow X$  is true if at least one of A or B is true
  - AND : symbol  $\cdot$  ,  $X = A \cdot B \Rightarrow X$  is true if both A and B are true
  - NOT : symbol  $\neg$  ,  $X = \neg A \Rightarrow X$  is the inverted value of A

# Boolean Algebra Rules

- Identity law :  $A + 0 = A$  ;  $A \cdot 1 = A$
- Zero and One laws :  $A + 1 = 1$  ;  $A \cdot 0 = 0$
- Inverse laws :  $A \cdot \overline{A} = 0$  ;  $A + \overline{A} = 1$
- Commutative laws :  $A + B = B + A$  ;  $A \cdot B = B \cdot A$
- Associative laws :  $A + (B + C) = (A + B) + C$   
 $A \cdot (B \cdot C) = (A \cdot B) \cdot C$
- Distributive laws :  $A \cdot (B + C) = (A \cdot B) + (A \cdot C)$   
 $A + (B \cdot C) = (A + B) \cdot (A + C)$

# DeMorgan's Laws

---

- $\overline{A + B} = \overline{A} \cdot \overline{B}$

- $\overline{A \cdot B} = \overline{A} + \overline{B}$

- Confirm that these are indeed true

# Pictorial Representations

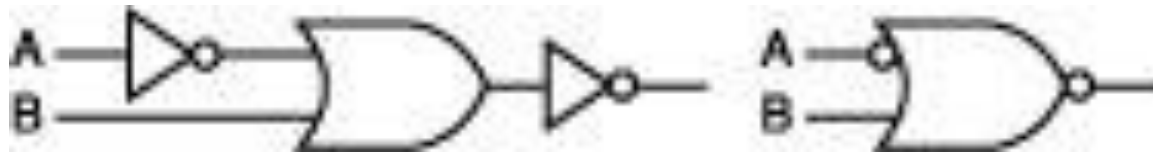
AND

OR

NOT



What logic function is this?





# Boolean Equation

---

- Consider the logic block that has an output  $E$  that is true only if exactly two of the three inputs  $A$ ,  $B$ ,  $C$  are true

# Boolean Equation

- Consider the logic block that has an output E that is true only if exactly two of the three inputs A, B, C are true

Multiple correct equations:

Two must be true, but all three ~~cannot be true~~:

$$E = ((A \cdot B) + (B \cdot C) + (A \cdot C)) \cdot (A \cdot B \cdot C)$$

Identify the ~~three cases~~ where it is true:

$$E = (A \cdot B \cdot C) + (A \cdot C \cdot B) + (C \cdot B \cdot A)$$



# Sum of Products

- Can represent any logic block with the AND, OR, NOT operators
  - Draw the truth table
  - For each true output, represent the corresponding inputs as a product
  - The final equation is a sum of these products

A	B	C	E	
0	0	0	0	
0	0	1	0	$(A \cdot B \cdot \overline{C}) + (A \cdot C \cdot \overline{B}) + (C \cdot B \cdot \overline{A})$
0	1	0	0	
0	1	1	1	• Can also use “product of sums”
1	0	0	0	• Any equation can be implemented
1	0	1	1	with an array of ANDs, followed by
1	1	0	1	an array of ORs
1	1	1	0	

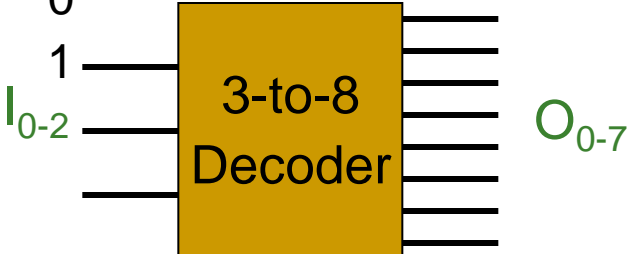
# NAND and NOR

- NAND : NOT of AND :  $A \text{ nand } B = \overline{A \cdot B}$
- NOR : NOT of OR :  $A \text{ nor } B = \overline{A + B}$
- NAND and NOR are *universal gates*, i.e., they can be used to construct any complex logical function

# Common Logic Blocks – Decoder

Takes in  $N$  inputs and activates one of  $2^N$  outputs

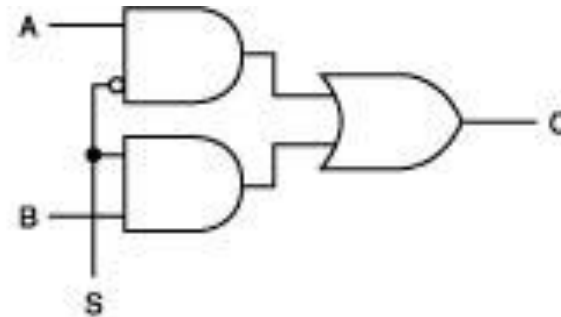
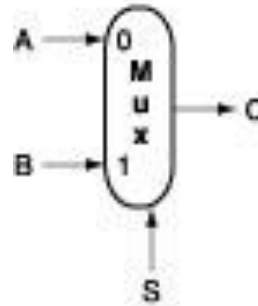
$I_0$	$I_1$	$I_2$								
			$O_0$	$O_1$	$O_2$	$O_3$	$O_4$	$O_5$	$O_6$	$O_7$
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

# Common Logic Blocks – Multiplexor

- Multiplexor or selector: one of N inputs is reflected on the output depending on the value of the  $\log_2 N$  selector bits

2-input mux



# Hardware for Arithmetic

---

- Today's topics:
  - Designing an ALU
  - Carry-lookahead adder

# Sum of Products

- Can represent any logic block with the AND, OR, NOT operators
  - Draw the truth table
  - For each true output, represent the corresponding inputs as a product
  - The final equation is a sum of these products

A	B	C	E
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

$(A \cdot B \cdot \overline{C}) + (A \cdot C \cdot \overline{B}) + (C \cdot B \cdot \overline{A})$

- Can also use “product of sums”
- Any equation can be implemented with an array of ANDs, followed by an array of ORs

# Adder Algorithm

	1	0	0	1
	0	1	0	1
Sum	1	1	1	0
Carry	0	0	0	1

Truth Table for the above operations:

A	B	Cin	Sum	Cout
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

# Adder Algorithm

	1	0	0	1
	0	1	0	1
Sum	1	1	1	0
Carry	0	0	0	1

Equations:

$$\begin{aligned} \text{Sum} &= \text{Cin} \oplus A \oplus B + \\ &\quad B \oplus \text{Cin} \oplus A + \\ &\quad A \oplus \text{Cin} \oplus B + \\ &\quad A \oplus B \oplus \text{Cin} \end{aligned}$$

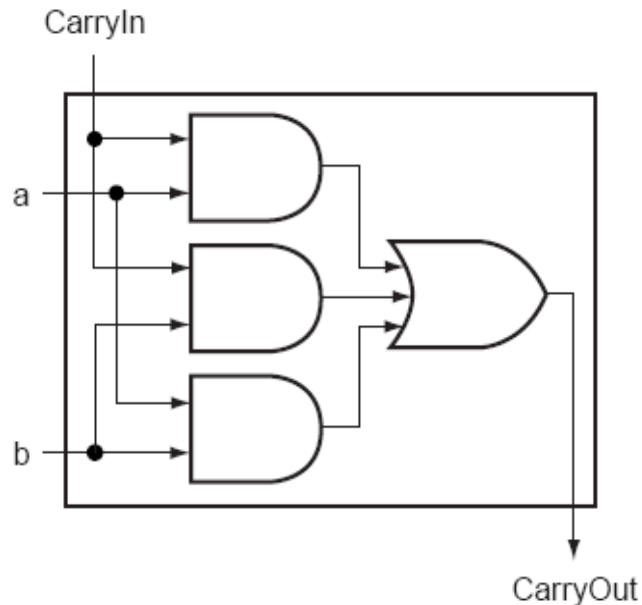
Truth Table for the above operations:

A	B	Cin	Sum	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$\begin{aligned} \text{Cout} &= A \cdot B \cdot \text{Cin} + \\ &\quad A \cdot B \cdot \text{Cin} + \\ &\quad A \cdot \text{Cin} \cdot B + \\ &\quad B \cdot \text{Cin} \cdot A + \\ &= A \cdot B + \\ &\quad A \cdot \text{Cin} + \\ &\quad B \cdot \text{Cin} \end{aligned}$$



# Carry Out Logic



Equations:

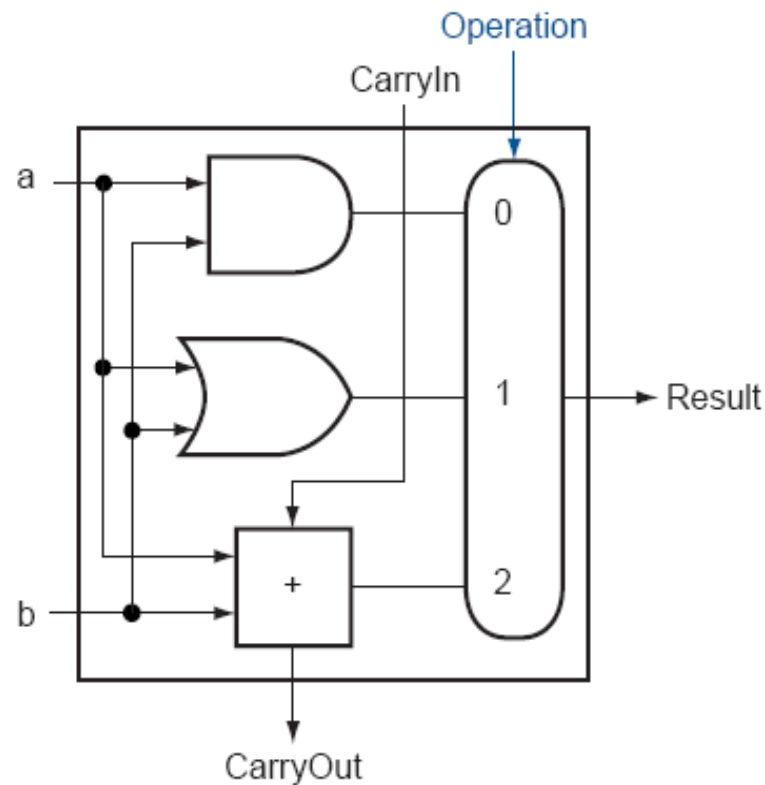
$$\begin{aligned} \text{Sum} = & \text{Cin} \cdot \bar{A} \cdot \bar{B} + \\ & B \cdot \bar{\text{Cin}} \cdot \bar{A} + \\ & A \cdot \text{Cin} \cdot B + \\ & A \cdot B \cdot \text{Cin} \end{aligned}$$

$$\begin{aligned} \text{Cout} = & A \cdot B \cdot \text{Cin} + \\ & A \cdot B \cdot \bar{\text{Cin}} + \\ & A \cdot \text{Cin} \cdot \bar{B} + \\ & B \cdot \text{Cin} \cdot \bar{A} \\ = & A \cdot B + \\ & A \cdot \text{Cin} + \\ & B \cdot \text{Cin} \end{aligned}$$

**FIGURE B.5.5 Adder hardware for the carry out signal.** The rest of the adder hardware is the logic for the Sum output given in the equation on page B-28.

# 1-Bit ALU with Add, Or, And

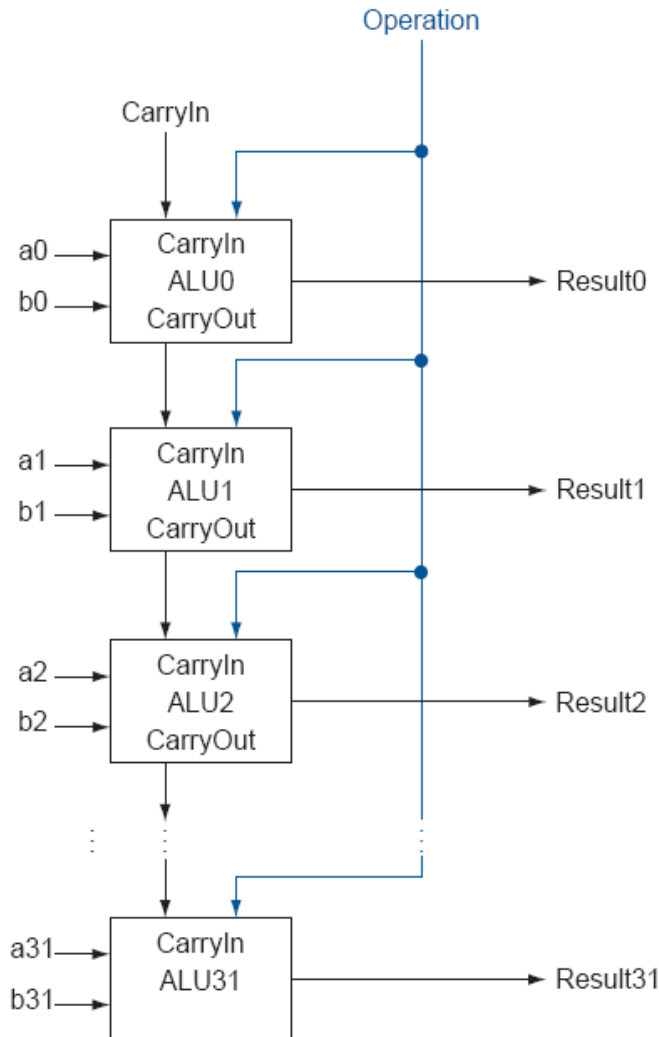
- Multiplexor selects between Add, Or, And operations



**FIGURE B.5.6** A 1-bit ALU that performs AND, OR, and addition (see Figure B.5.5).

# 32-bit Ripple Carry Adder

1-bit ALUs are connected  
“in series” with the  
carry-out of 1 box  
going into the carry-in  
of the next box



**FIGURE B.5.7 A 32-bit ALU constructed from 32 1-bit ALUs.** CarryOut of the less significant bit is connected to the CarryIn of the more significant bit. This organization is called ripple carry.



# Incorporating Subtraction

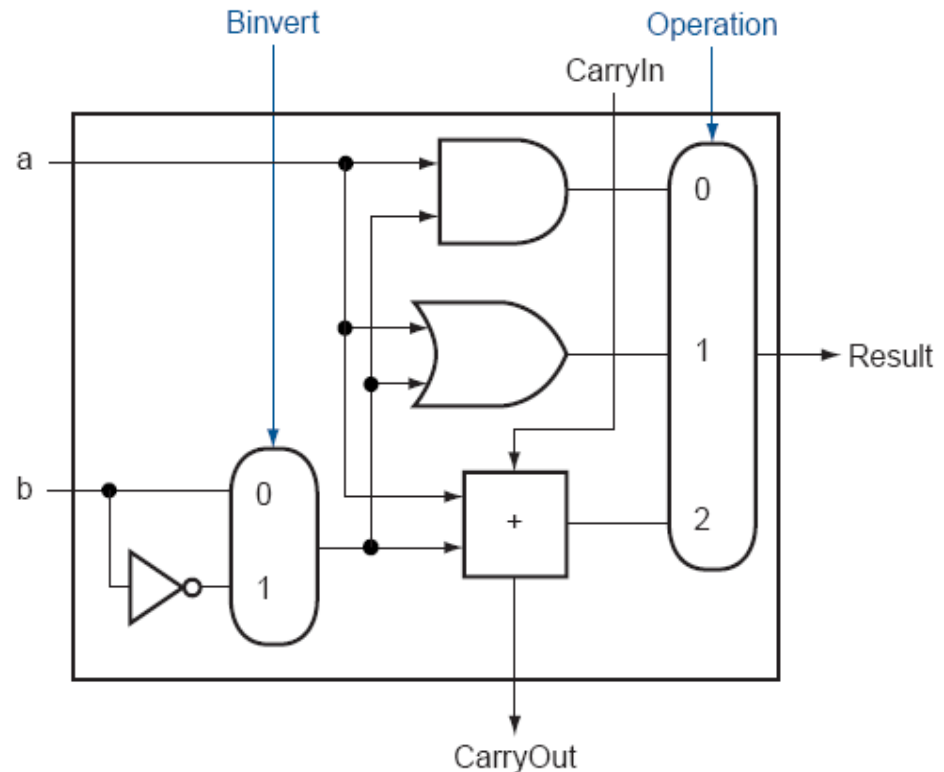
---



# Incorporating Subtraction

Must invert bits of B and add a 1

- Include an inverter
- CarryIn for the first bit is 1
- The CarryIn signal (for the first bit) can be the same as the Binvert signal



**FIGURE B.5.8 A 1-bit ALU that performs AND, OR, and addition on *a* and *b* or *a* and  $\bar{b}$ .** By selecting  $\bar{b}$  (*Binvert* = 1) and setting *CarryIn* to 1 in the least significant bit of the ALU, we get two's complement subtraction of *b* from *a* instead of addition of *b* to *a*.

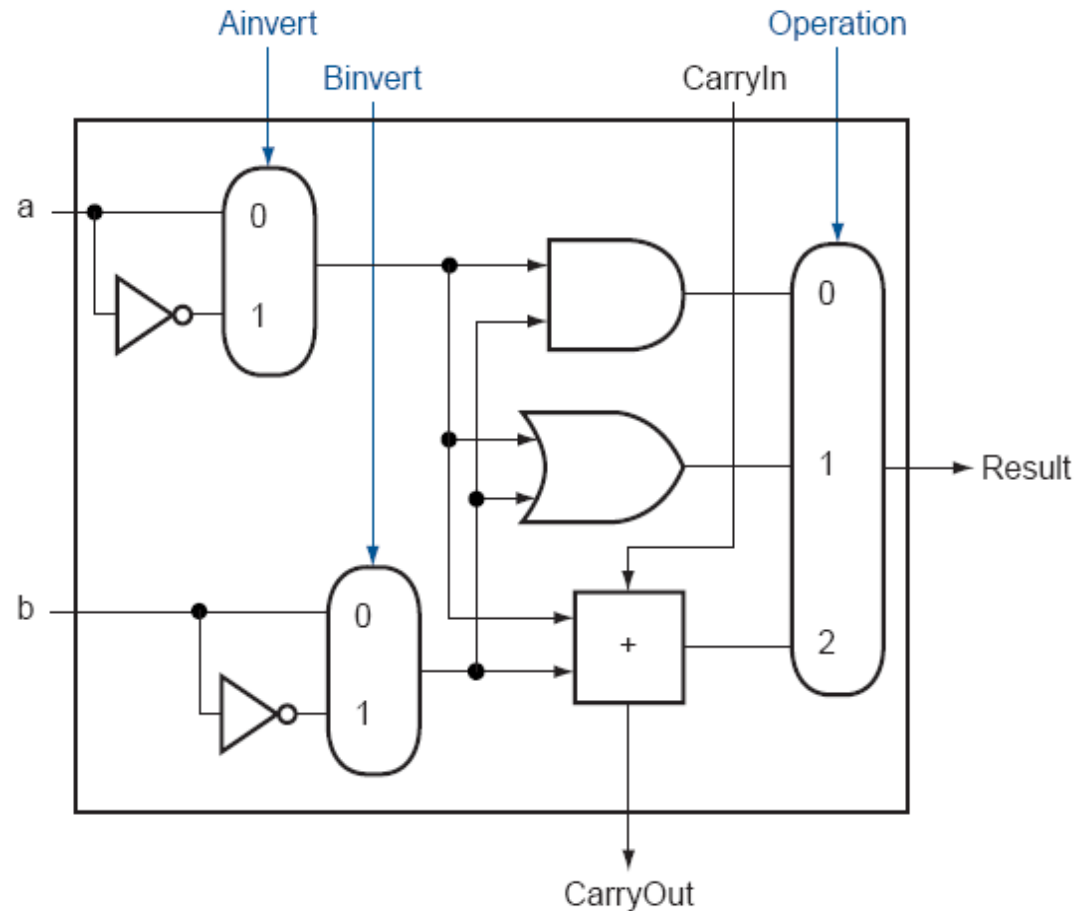


# Incorporating NOR

---



# Incorporating NOR



**FIGURE B.5.9 A 1-bit ALU that performs AND, OR, and addition on  $a$  and  $b$  or  $\bar{a}$  and  $\bar{b}$ .** By selecting  $\bar{a}$  ( $Ainvert = 1$ ) and  $\bar{b}$  ( $Binvert = 1$ ), we get a NOR  $b$  instead of  $a$  AND  $b$ .



# Incorporating slt

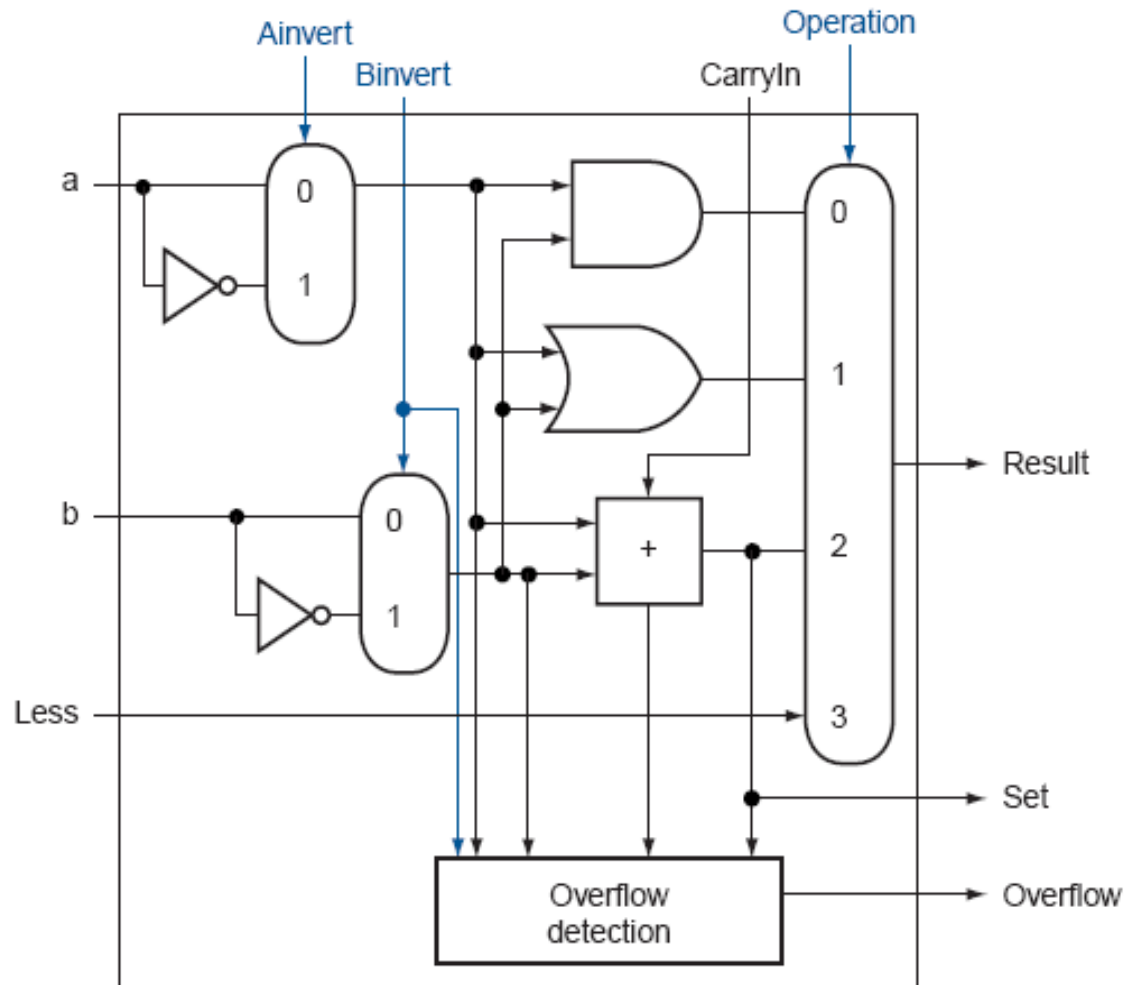
---





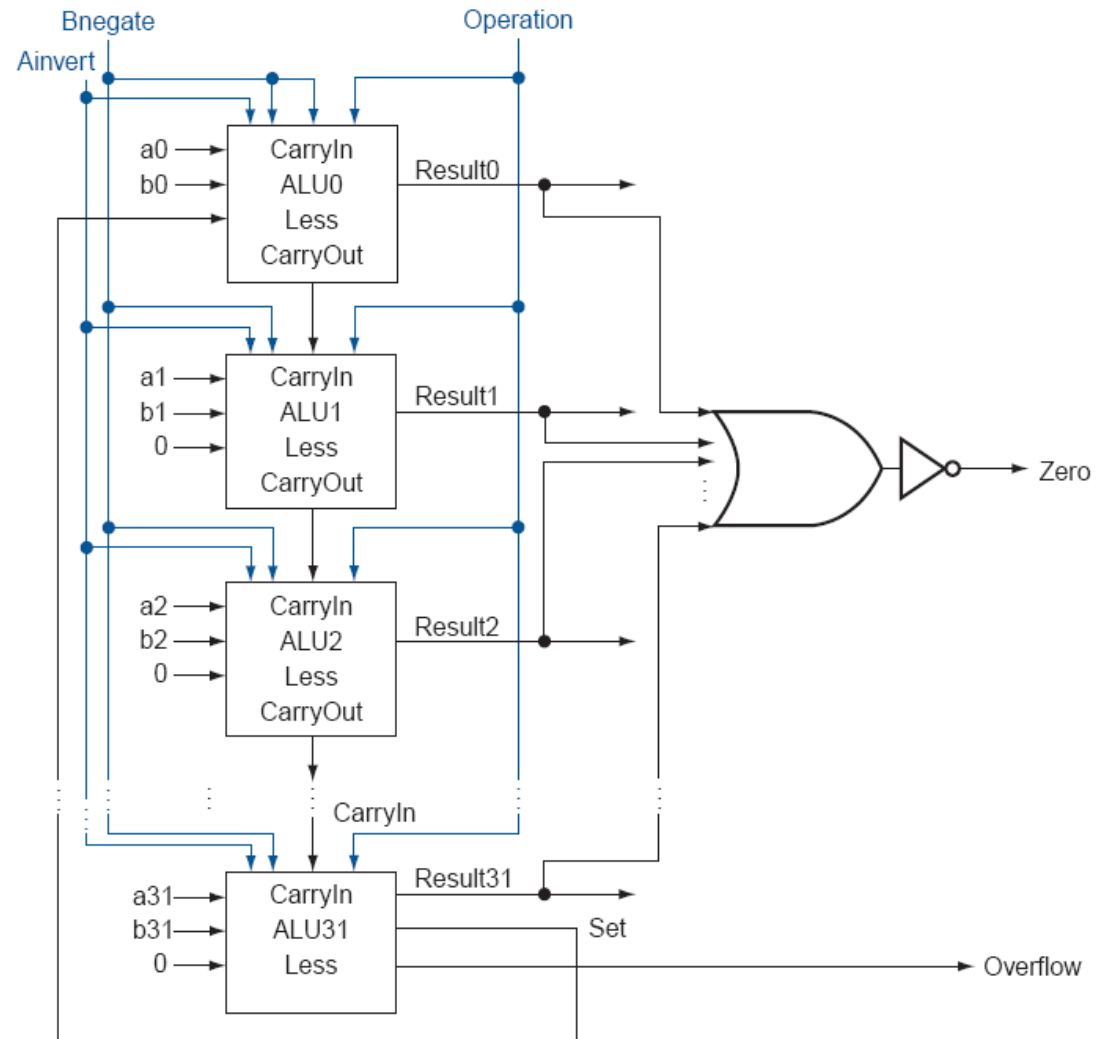
# Incorporating slt

- Perform  $a - b$  and check the sign
- New signal (Less) that is zero for ALU boxes 1-31
- The 31<sup>st</sup> box has a unit to detect overflow and sign – the sign bit serves as the Less signal for the 0<sup>th</sup> box



# Incorporating beq

- Perform  $a - b$  and confirm that the result is all zero's



**FIGURE B.5.12 The final 32-bit ALU.** This adds a Zero detector to Figure B.5.11.

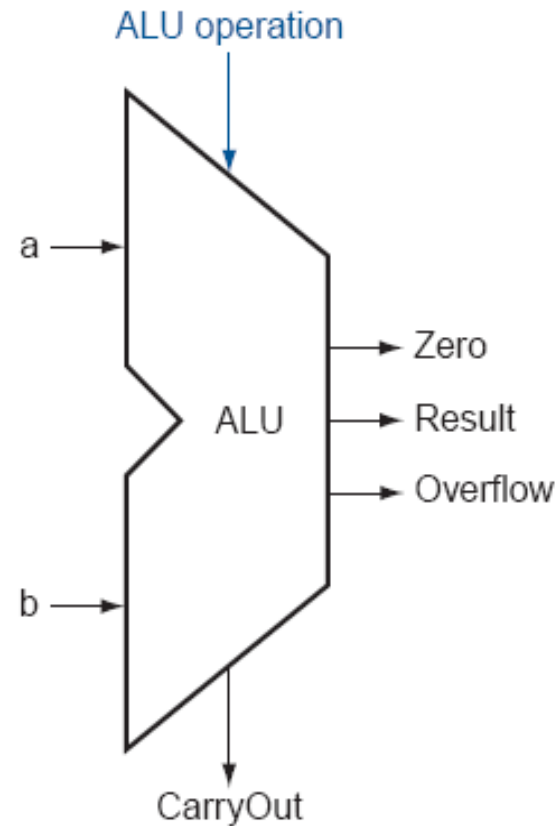
The diagram illustrates a multi-precision ALU architecture. It consists of a chain of ALU units, labeled ALU0, ALU1, ALU2, ..., ALU31. Each ALU unit has three inputs: a data input (a<sub>i</sub>), a second data input (b<sub>i</sub>), and a carry-in (CarryIn). The ALU units are connected in a chain, with the CarryOut of one unit serving as the CarryIn for the next unit. The ALU units are controlled by a common Operation signal and a Bnegate signal. The ALU0 unit also receives an Ainvert signal. The ALU units produce a Result (Result0, Result1, Result2, ..., Result31) and a Set signal. The Set signal is used to generate the Zero and Overflow flags. The Zero flag is generated by a logic circuit that combines the Set signal with the Result signals. The Overflow flag is generated by a logic circuit that combines the Set signal with the Result signals.

山先生印

# Control Lines

What are the values of the control lines and what operations do they correspond to?

	Ai	Bn	Op
AND	0	0	00
OR	0	0	01
Add	0	0	10
Sub	0	1	10
SLT	0	1	11
NOR	1	1	00



# Speed of Ripple Carry

---

- The carry propagates thru every 1-bit box: each 1-bit box sequentially implements AND and OR – total delay is the time to go through 64 gates!
- We've already seen that any logic equation can be expressed as the sum of products – so it should be possible to compute the result by going through only 2 gates!
- Caveat: need many parallel gates and each gate may have a very large number of inputs – it is difficult to efficiently build such large gates, so we'll find a compromise:
  - moderate number of gates
  - moderate number of inputs to each gate
  - moderate number of sequential gates traversed

# Computing CarryOut

$$\text{CarryIn1} = b0.\text{CarryIn0} + a0.\text{CarryIn0} + a0.b0$$

$$\begin{aligned}\text{CarryIn2} &= b1.\text{CarryIn1} + a1.\text{CarryIn1} + a1.b1 \\ &= b1.b0.c0 + b1.a0.c0 + b1.a0.b0 + \\ &\quad a1.b0.c0 + a1.a0.c0 + a1.a0.b0 + a1.b1\end{aligned}$$

...

$\text{CarryIn32}$  = a really large sum of really large products

- Potentially fast implementation as the result is computed by going thru just 2 levels of logic – unfortunately, each gate is enormous and slow

# Generate and Propagate

Equation re-phrased:

$$\begin{aligned} C_{i+1} &= a_i.b_i + a_i.C_i + b_i.C_i \\ &= (a_i.b_i) + (a_i + b_i).C_i \end{aligned}$$

Stated verbally, the current pair of bits will *generate* a carry if they are both 1 and the current pair of bits will *propagate* a carry if either is 1

Generate signal =  $a_i.b_i$

Propagate signal =  $a_i + b_i$

Therefore,  $C_{i+1} = G_i + P_i . C_i$

# Generate and Propagate

$$c1 = g0 + p0.c0$$

$$c2 = g1 + p1.c1$$

$$= g1 + p1.g0 + p1.p0.c0$$

$$c3 = g2 + p2.g1 + p2.p1.g0 + p2.p1.p0.c0$$

$$c4 = g3 + p3.g2 + p3.p2.g1 + p3.p2.p1.g0 + p3.p2.p1.p0.c0$$

Either,

a carry was just generated, or

a carry was generated in the last step and was propagated, or

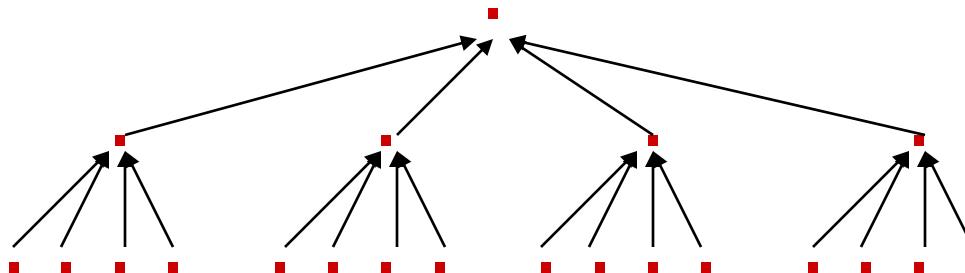
a carry was generated two steps back and was propagated by both the next two stages, or

a carry was generated N steps back and was propagated by every single one of the N next stages



# Divide and Conquer

- The equations on the previous slide are still difficult to implement as logic functions – for the 32<sup>nd</sup> bit, we must AND every single propagate bit to determine what becomes of c0 (among other things)
- Hence, the bits are broken into groups (of 4) and each group computes its group-generate and group-propagate
- For example, to add 32 numbers, you can partition the task as a tree



# P and G for 4-bit Blocks

- Compute P0 and G0 (super-propagate and super-generate) for the first group of 4 bits (and similarly for other groups of 4 bits)

$$P0 = p0.p1.p2.p3$$

$$G0 = g3 + g2.p3 + g1.p2.p3 + g0.p1.p2.p3$$

- Carry out of the first group of 4 bits is

$$C1 = G0 + P0.c0$$

$$C2 = G1 + P1.G0 + P1.P0.c0$$

...

- By having a tree of sub-computations, each AND, OR gate has few inputs and logic signals have to travel through a modest set of gates (equal to the height of the tree)

# Example

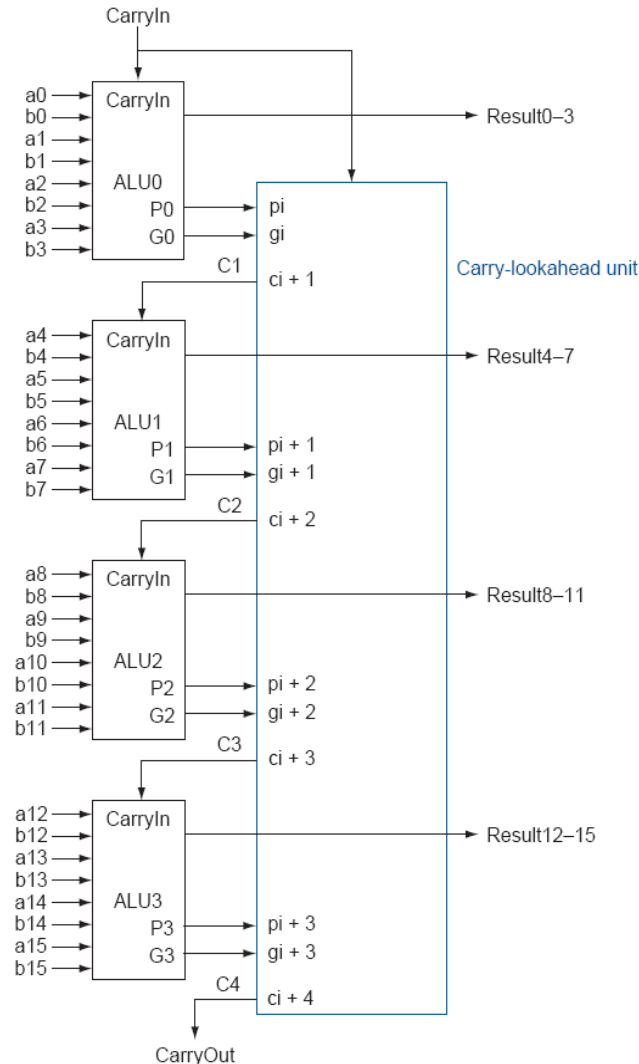
Add	A	0001	1010	0011	0011
and	B	1110	0101	1110	1011
<hr/>					
	g	0000	0000	0010	0011
	p	1111	1111	1111	1011

P	1	1	1	0
G	0	0	1	0

$C4 = 1$

# Carry Look-Ahead Adder

- 16-bit Ripple-carry takes 32 steps
- This design takes how many steps?



**FIGURE B.6.3 Four 4-bit ALUs using carry lookahead to form a 16-bit adder.** Note that the carries come from the carry-lookahead unit, not from the 4-bit ALUs.

# Sequential Circuits

---

- Today's topics:
  - Carry-lookahead adder
  - Clocks and sequential circuits
  - Finite state machines

# Speed of Ripple Carry

---

- The carry propagates thru every 1-bit box: each 1-bit box sequentially implements AND and OR – total delay is the time to go through 64 gates!
- We've already seen that any logic equation can be expressed as the sum of products – so it should be possible to compute the result by going through only 2 gates!
- Caveat: need many parallel gates and each gate may have a very large number of inputs – it is difficult to efficiently build such large gates, so we'll find a compromise:
  - moderate number of gates
  - moderate number of inputs to each gate
  - moderate number of sequential gates traversed

# Computing CarryOut

$$\text{CarryIn1} = b0.\text{CarryIn0} + a0.\text{CarryIn0} + a0.b0$$

$$\begin{aligned}\text{CarryIn2} &= b1.\text{CarryIn1} + a1.\text{CarryIn1} + a1.b1 \\ &= b1.b0.c0 + b1.a0.c0 + b1.a0.b0 + \\ &\quad a1.b0.c0 + a1.a0.c0 + a1.a0.b0 + a1.b1\end{aligned}$$

...

$\text{CarryIn32}$  = a really large sum of really large products

- Potentially fast implementation as the result is computed by going thru just 2 levels of logic – unfortunately, each gate is enormous and slow

# Generate and Propagate

Equation re-phrased:

$$\begin{aligned} c_{i+1} &= a_i.b_i + a_i.c_i + b_i.c_i \\ &= (a_i.b_i) + (a_i + b_i).c_i \end{aligned}$$

Stated verbally, the current pair of bits will *generate* a carry if they are both 1 and the current pair of bits will *propagate* a carry if either is 1

Generate signal =  $a_i.b_i$

Propagate signal =  $a_i + b_i$

Therefore,  $c_{i+1} = g_i + p_i . c_i$



# P and G for 4-bit Blocks

- Compute P0 and G0 (super-propagate and super-generate) for the first group of 4 bits (and similarly for other groups of 4 bits)  
$$P0 = p0.p1.p2.p3$$
$$G0 = g3 + g2.p3 + g1.p2.p3 + g0.p1.p2.p3$$
- Carry out of the first group of 4 bits is  
$$C1 = G0 + P0.c0$$
$$C2 = G1 + P1.G0 + P1.P0.c0$$
  
...
- By having a tree of sub-computations, each AND, OR gate has few inputs and logic signals have to travel through a modest set of gates (equal to the height of the tree)

# Example

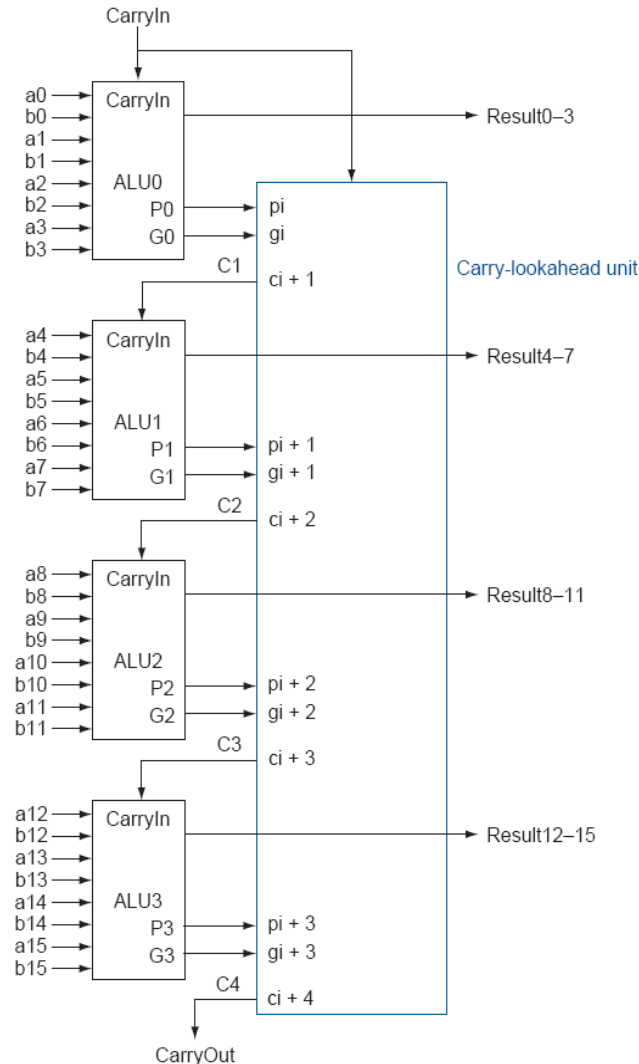
Add	A	0001	1010	0011	0011
and	B	1110	0101	1110	1011
<hr/>					
	g	0000	0000	0010	0011
	p	1111	1111	1111	1011

P	1	1	1	0
G	0	0	1	0

$C4 = 1$

# Carry Look-Ahead Adder

- 16-bit Ripple-carry takes 32 steps
- This design takes how many steps?



**FIGURE B.6.3 Four 4-bit ALUs using carry lookahead to form a 16-bit adder.** Note that the carries come from the carry-lookahead unit, not from the 4-bit ALUs.

# 信息隐藏技术

- 信息隐藏 (Information Hiding) 作为一门新兴的交叉学科, 伴随着信息和网络技术的飞速发展, 在隐蔽通信、数字版权保护等方面起着越来越重要的作用。信息隐藏是将秘密信息隐藏在另一非机密的载体信息中, 通过公共信道进行传递。秘密信息被隐藏后, 攻击者无法判断载体信息中是否隐藏了秘密信息, 也无法从载体信息中提取或去除所隐藏的秘密信息。信息隐藏研究的内容包括信息隐藏算法、数字水印、隐通道技术和匿名通信技术等。



# 信息安全与技术

## 第05章：网络攻击技术

### § 5.1：网络攻击概述



玉泉校区曹光彪东楼507室



20:03:30

浙江大学计算机学院



# 信息安全与技术

## 第05章：网络攻击技术

### §：本章小结



玉泉校区曹光彪东楼507室



20:03:30

浙江大学计算机学院



# 信息安全与技术

## 第05章：网络攻击技术

### §：思考题



玉泉校区曹光彪东楼507室



20:03:30

浙江大学计算机学院





# Thank You!

非常天空



浙江大学计算机学院