《计算机系统结构》课程直播

2020. 3.5

请将ZOOM名称改为“姓名”；

听不到声音请及时调试声音设备；签到将在课间休息进行

本次讲课内容

1. 补码及其加减运算
2. 无符号整数
3. 移位运算

Why binary？

为什么二进制?

数制

- 十进制数（decimal system）
  - 采用十个计数符号
  - 计数规则
    - 逢十进一
  - 一个n位的十进制数$x_0\ x_1 \ldots x_{n-2}\ x_{n-1}$代表的数值为：

$$x_0 * 10^{n-1} + x_1 * 10^{n-2} + \ldots + x_{n-2} * 10^1 + x_{n-1} * 10^0$$

二进制 Binary Representation

- The binary number

01011000 00010101 00101110 11100111

Most significant bit   Least significant bit

represents the quantity

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

- 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

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

Why binary？

- 最少物理设备

(a) 数位长度与基的关系

给定任意进位制R下的一个数 $N=(d_n d_{n-1} \ldots\ldots d_0)_R$

j是R进位制下的数位长度， $j\,(\min) = \log_R N$

(b)设备量与基数的关系 $D=R\log_R N$

$D' = R'\log_R N + R(\log_R N)' = (\ln N * \ln R - \ln N)/(\ln R)^2$

所以 $\ln R=1$ $R=e=2.718$ 时所用设备量最少。

- 最简单物理实现：二态比三态好实现

计算机常用的其他进位制

- 计算机内部运算以二进制为基础，而外部表示时除了二进制、十进制，还常用到：
- 八进制
  - octal，缩写O，采用0，1，2，3，4，5，6，7八个数字
  - 一位八进制数转换成三位二进制数，例如：$001010110_2=126_8$
- 十六进制 （Hexadecimal：便于阅读和转换）
  - 用数字0到9和字母A到F（或a~f），其中:A~F表示10~15，
  - 一位十六进制数转换成四位二进制数，例如： $0010101102 =056_{16}$
  - C语言、C++、Shell、Python、Java语言及其他相近的语言使用字首“0x”，例如“0x5A3”。
  - Intel的汇编语言中用字尾“h”或“H”来标识16进制的数，例如“0A3Ch”、“5A3H”
  - 现代计算机使用16 - 32位，或者64位，进一步划分为多位字节表达，所以十六进制更方便和常用.

Why two's-complement？

为什么使用补码?

补码表示法: 2’s Complement

- 方法1
  - 正数：直接取其原来的二进制码（加符号位0）
  - 负数：对其二进制码按位取反之后再在最低位加1

    例：$[010101]_{补}$= 00010101

    $[-010101]_{补}$= 11101010+1 = 11101011

- 方法2
  - 正数：直接取其原来的二进制码
  - 负数：从二进制码的最低位开始，对遇到的0和第一个1取其原来的二进制编码，从第一个1以后开始直到最高位均取其相反编码。

    例：$[101010]_{补}$= 00101010

    $[-101010]_{补}$= 11010110

2's Complement

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}$
…
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 2^{31}-1$

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -2^{31}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -(2^{31} - 1)$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -(2^{31} - 2)$
…
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -2$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -1$

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

$x_{31}\ -2^{31} + x_{30}\ 2^{30} + x_{29}\ 2^{29} + \ldots + x_1\ 2^1 + x_0\ 2^0$

Why the name 2's Complement ?

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}$
…
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 2^{31}-1$

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -2^{31}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -(2^{31} – 1)$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -(2^{31} – 2)$
…
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -2$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -1$

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

$x + x' = -1 \quad \rightarrow \quad x' + 1 = -x \quad \rightarrow \quad -x = x' + 1$

… can compute the negative of a number by 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

补码的特点

- 零是唯一的，没有正零和负零的区别
- 负数比正数多一个
- 补码是以2^{n+1} 为模的计量系统
- $[X + Y]_{补} = [X]_{补} + [Y]_{补}$
- 减法可以转换为加法
  - $[X-Y]_{补} = [X]_{补} + [-Y]_{补}$
- 符号位可以直接参与运算
- 计算机中广泛采用补码表达有符号整数。

补码加减法

- **补码加法**
  - 根据补码加法公式，补码可以直接相加。

$$[x]_{补} + [y]_{补} = [x+y]_{补} \qquad (\mathrm{mod}\ 2)$$

- **补码减法**
  - 根据补码减法公式，补码可以直接相减。

$$[x-y]_{补} = [x]_{补} - [y]_{补} = [x]_{补} + [-y]_{补} \qquad (\mathrm{mod}\ 2)$$

Example

- Compute 5-6

```
 5:  0000 0000 0000 0000 0000 0000 0000 0101
 6:  0000 0000 0000 0000 0000 0000 0000 0110
-6:  1111 1111 1111 1111 1111 1111 1111 1010
```

溢出的概念

- 溢出(Overflow) ：运算结果超出了数据表示范围

例如： short i = 23456;

short j = 23456; //short 最大值32767

short k = i + j; //此时k 为-18624

注意：数据取模时的丢弃 vs. 溢出

- 数据运算中最高位的进位被丢弃并不一定是溢出
- 例如

  设x= -0110，即-6_{10}； y= -0101，即-5_{10}。

  则$[x]_{补}$=11010，$[y]_{补}$=11011。

  $[x+y]_{补}$=10101 (mod 2^5)，即-11_{10}

  运算结果正确，没有发生溢出

溢出概念及其检测方法之三

- 将运算数的符号位设置为00 （正数）或 11（负数）
- 如果结果的符号位不是00或11，而是01或10则溢出

	0	0	1	1	1	7
+	0	0	0	1	1	3
	0	1	0	1	0	– 6

	1	1	1	0	0	–4
+	1	1	0	1	1	– 5
	1	0	1	1	1	7

避免数据的溢出的方法

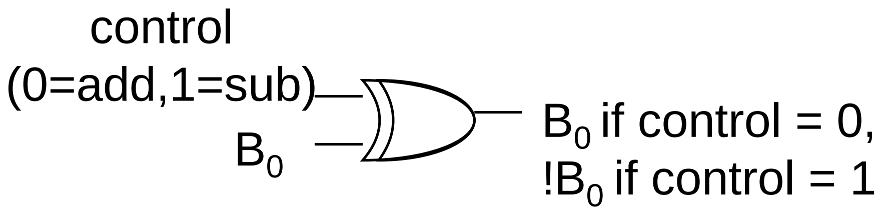
- 增加数据的表示位数
- 例如数据6
  - 在8位的计算机中表示为00000110
  - 在16位计算机中表示为0000000000000110
- 例如用补码表示-2时
  - 在8位计算机中是1111 1110
  - 在16位计算机中是1111 1111 1111 1110
- 带符号扩展

32位加/减法器

$[A+B]_{补} = [A]_{补} + [B]_{补}$

$[A-B]_{补} = [A]_{补} + [-B]_{补}$

已知$[B]_{补}$，求$[-B]_{补}$： 取反， 最低位加1



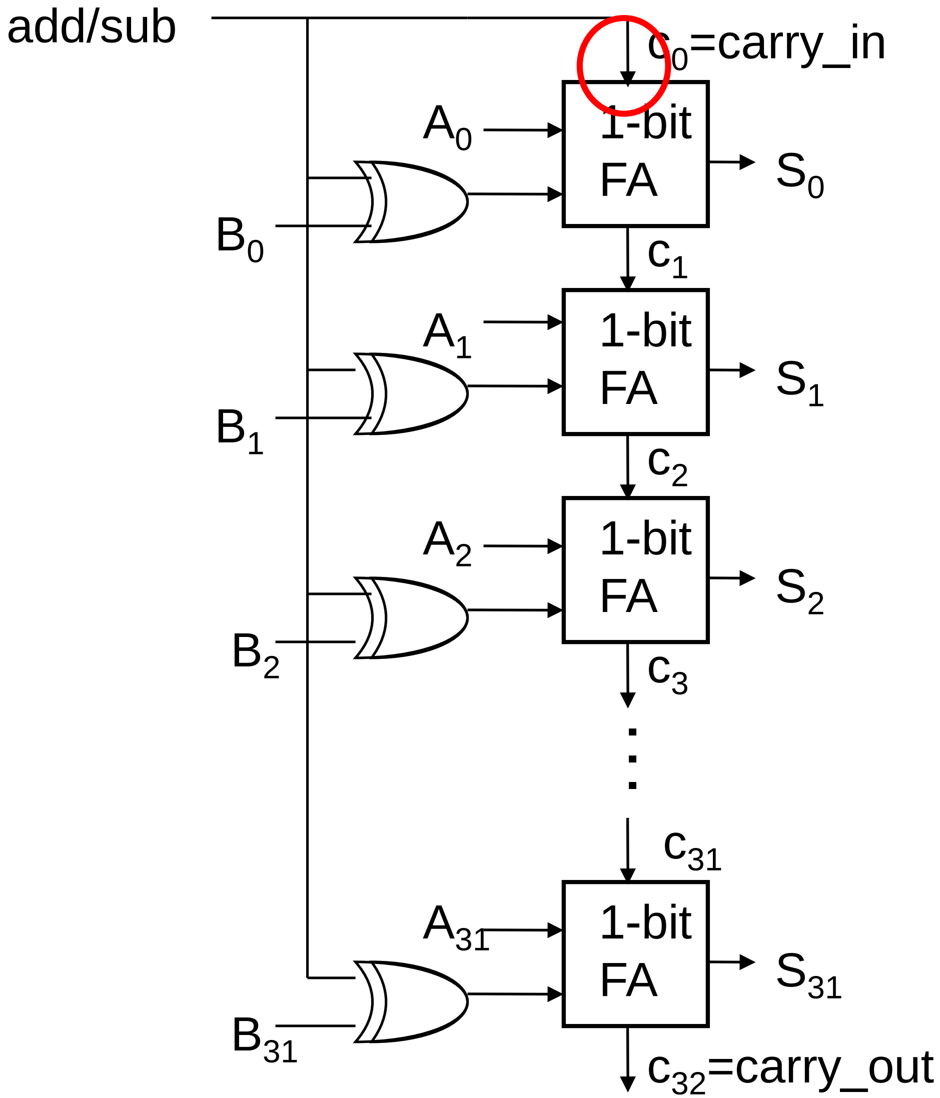
A 0111 → 0111

B − 0110 → + 1001

0001 +

1

1 0001



加速生成进位：Carry Lookahead Adder (CLA)

- 进位信号

$$c_1 = y_0c_0 + x_0c_0 + x_0y_0$$

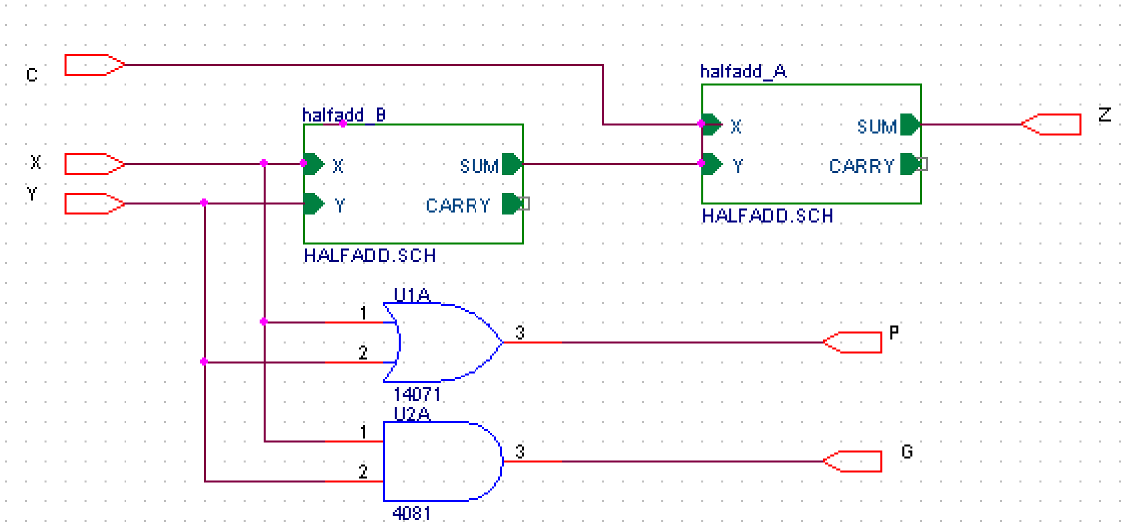
$$c_2 = y_1c_1 + x_1c_1 + x_1y_1$$

$$= x_1x_0y_0 + x_1x_0c_0 + x_1y_0c_0 + y_1x_0y_0 + y_1y_0c_0 + y_1x_0c_0 + x_1y_1$$

- 简化的进位信号
  - 由每个全加器输出

$$g_i = x_iy_i$$

$$p_i = x_i+y_i$$



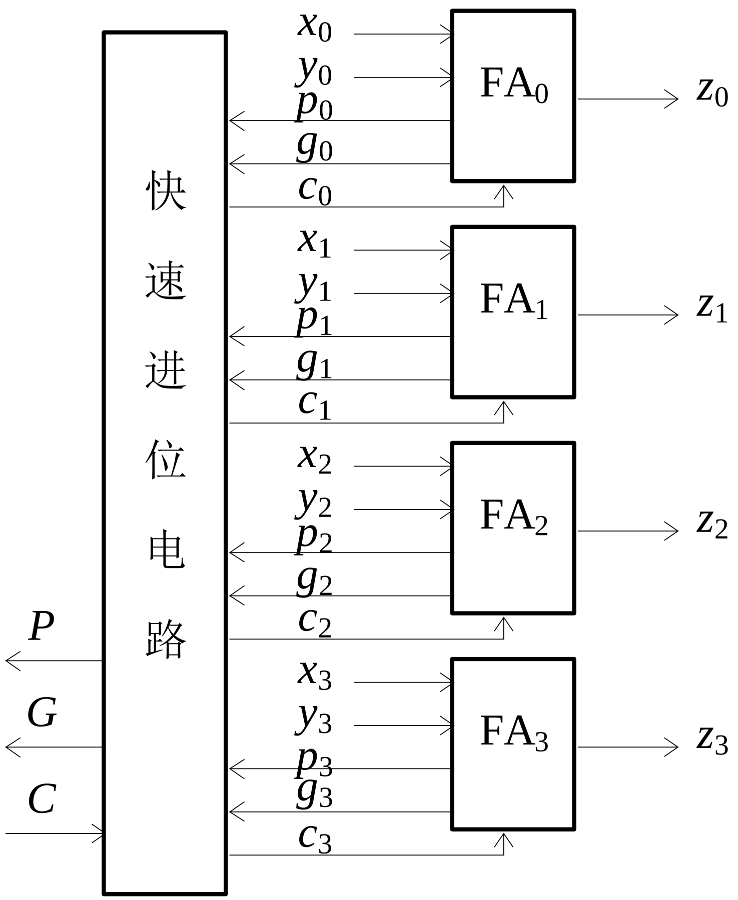
加速生成进位信号

$$c_1 = g_0 + p_0 c_0$$

$$c_2 = g_1 + p_1 c_1$$
$$= g_1 + p_1 g_0 + p_1 p_0 c_0$$

$$c_3 = g_2 + p_2 c_2$$
$$= g_2 + p_2 g_1 + p_2 p_1 g_0$$
$$+ p_2 p_1 p_0 c_0$$

$$c_4 = g_3 + p_3 g_2 + p_3 p_2 g_1$$
$$+ p_3 p_2 p_1 g_0$$
$$+ p_3 p_2 p_1 p_0 c_0$$



加速生成进位信号

$P_0 = p_3p_2p_1p_0$

$P_1 = p_7p_6p_5p_4$

$P_2 = p_{11}p_{10}p_9p_8$

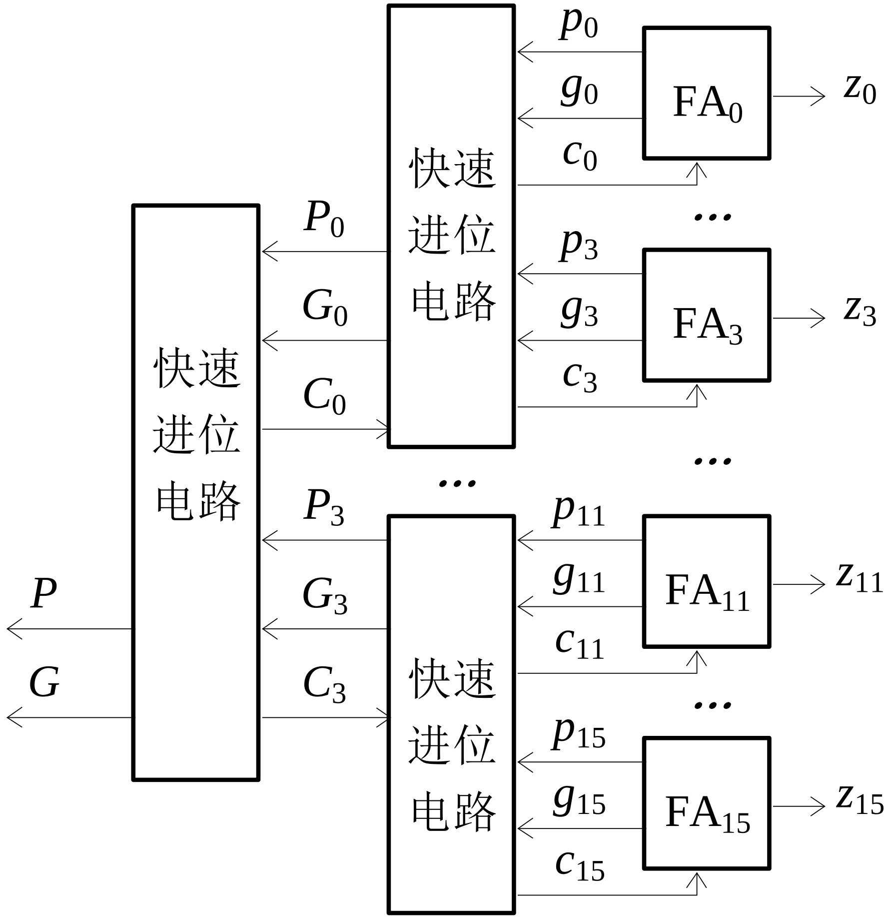
$P_3 = p_{15}p_{14}p_{13}p_{12}$

$G_0 = g_3 + p_3g_2$
$\quad + p_3p_2g_1 + p_3p_2p_1g_0$

$G_1 = g_7 + p_7g_6$
$\quad + p_7p_6g_5 + p_7p_6p_5g_4$

$G_2 = g_{11} + p_{11}g_{10}$
$\quad + p_{11}p_{10}g_9 + p_{11}p_{10}p_9g_8$

$G_3 = g_{15} + p_{15}p_{14}$
$+ p_{15}p_{14}g_{13} + p_{15}p_{14}p_{13}g_{12}$



加速生成进位信号

$$C_1 = G_0 + P_0 c_0$$

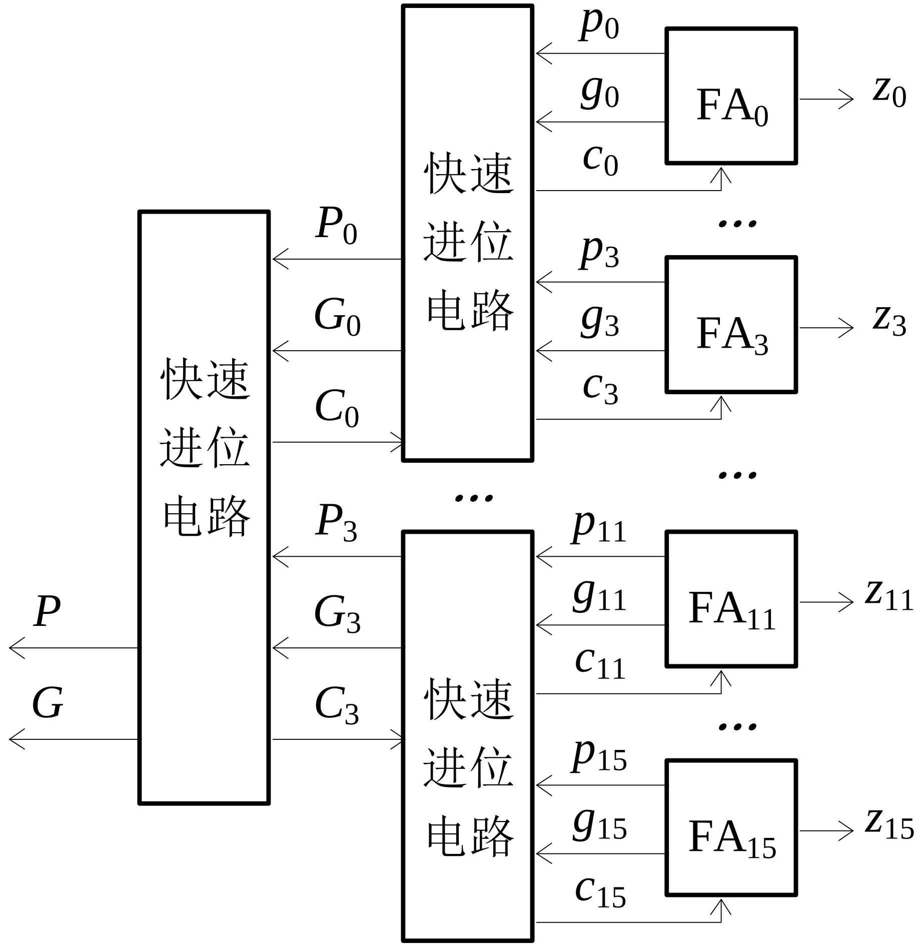
$$C_2 = G_1 + P_1 G_0 + P_1 P_0 c_0$$

$$C_3 = G_2 + P_2 G_1 + P_2 P_1 G_0$$

$$+ P_2 P_1 P_0 c_0$$

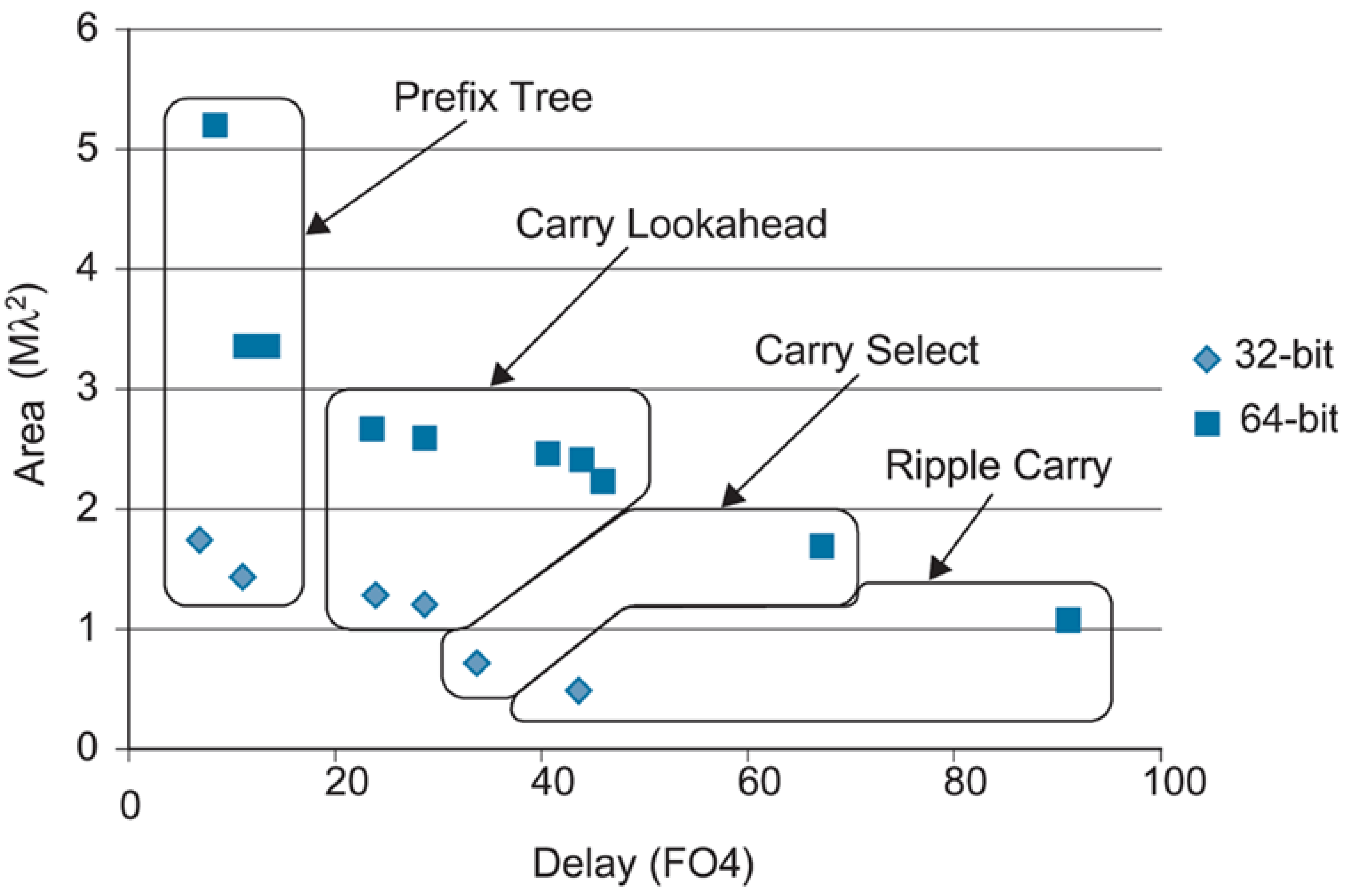
$$C_4 = G_3 + P_3 G_2 + P_3 P_2 G_1$$

$$+ P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 c_0$$



Adders In Real Processors

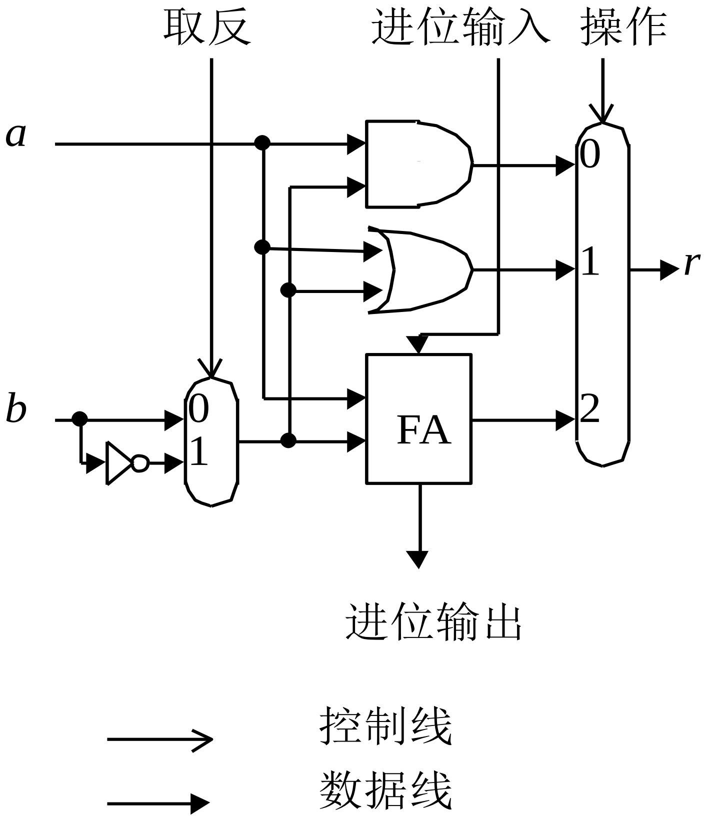
- Real processors super-optimize their adders
  - Ten or so different versions of CLA
  - Highly optimized versions of carry-select
  - Other gate techniques: carry-skip, conditional-sum
  - Sub-gate (transistor) techniques: Manchester carry chain
  - Combinations of different techniques
    - Alpha 21264 used CLA+CSeA+RippleCA
    - Used at different levels

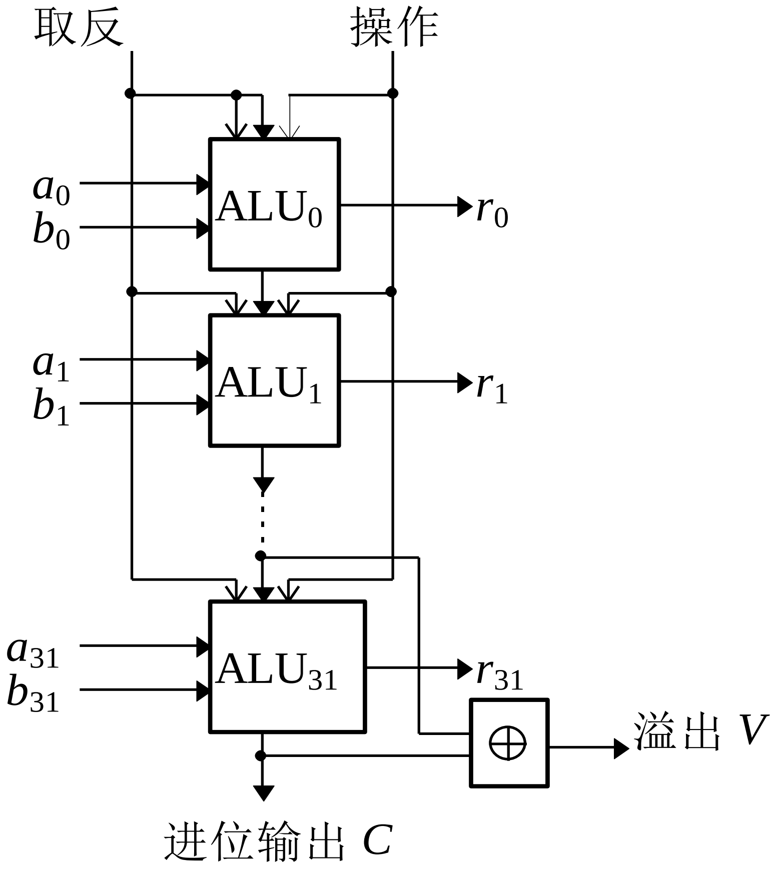


Area vs. delay of synthesized adders

定点运算器的组成结构

逻辑电路





Why unsigned?

为什么引入无符号数表示?

MIPS Instructions

Consider a comparison instruction:

slt $t0, $t1, $zero,

$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
```

举例:带符号位的扩展 Sign Extension

```
short int x =  15213;
int       ix = (int) x;
short int y = -15213;
int       iy = (int) y;
```

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
ix	15213	00 00 3B 6D	00000000 00000000 00111011 01101101
y	-15213	C4 93	11000100 10010011
iy	-15213	FF FF C4 93	11111111 11111111 11000100 10010011

- C语言：　自动完成符号位扩展

Unsigned Extension：“0”扩展

- 假设编译器规定int和short类型长度分别为32位和16位，若有下列C语言语句:
- unsigned short x=65530; //： 0X FFFA

  int y=x;
- 得到的y的机器数是（ ）
- A. 0000 7FFA H B. 0000 FFFA H
- C. FFFF 7FFA H D. FFFF FFFA H

已知 f(n)= = 11…1 B，计算f(n)的C语言函数f 如下：

```
int f ( unsigned n)
{       int sum = 1 , power = 1 ;
        for( unsigned i= 0； i<= n - 1 ； i + + )
        {       power *= 2;
                sum + = power；
        }
        return sum;
}
```

当n = 0时，f 会出现死循环，为什么？

After executing the following code, which of the variables are equal to 0 ?

```
unsigned int a = 0xffffffff;
    unsigned int b = 1;
    unsigned int c = a + b;
    unsigned long d = a + b;
    unsigned long e = (unsigned long)a + b;
```

(Assume ints are 32 bits wide and longs are 64 bits wide.)

(a) None of them **(b) c** **(c) c and d** **(d) c, d, and e**

移位运算

通过左移 实现乘 2的幂 运算

- **u << k** 等价于 $u\times 2^k$
- 对无符号数、带符号数 均有效
- 例如：
  - **u << 3 == u * 8**
  - **(u << 5) – (u << 3) == u * 24**
  - 移位比乘法计算速度快
  - 编译器能自动产生优化代码

编译器对（常量）乘法的优化

```
long mul12(long x)
{
  return x*12;
}
```

C 函数

编译后的指令:

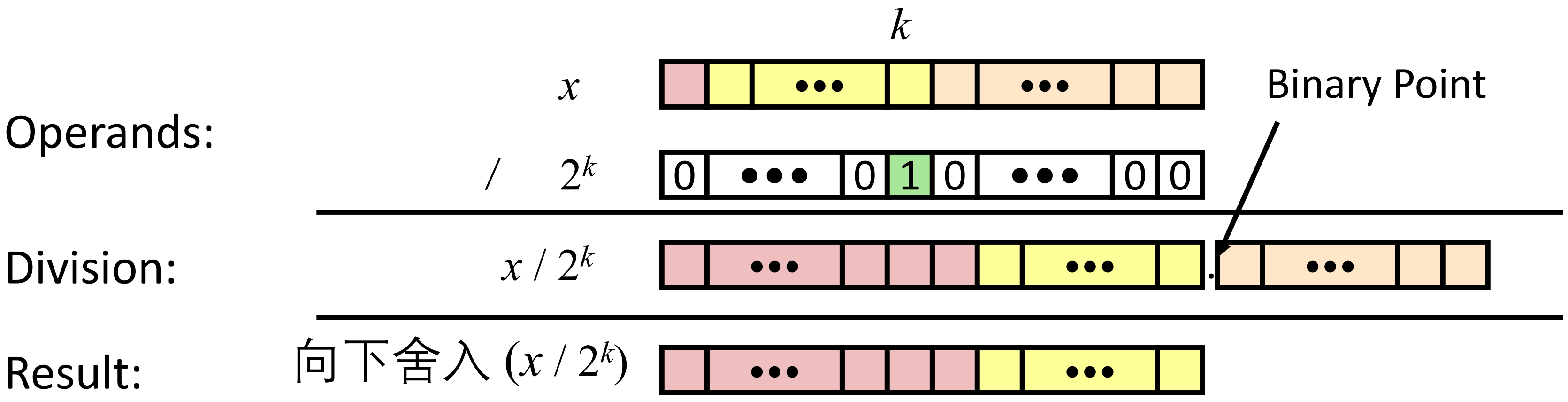
```
  leaq (%rax,%rax,2), %rax
  salq $2, %rax
```

解释

```
  t <- x+x*2
  return t << 2;
```

带符号数的除法：右移

- `x >> k` gives $\lfloor x \;/\; 2^k \rfloor$
- 算术右移
- 当 `u < 0`，舍入方向错误

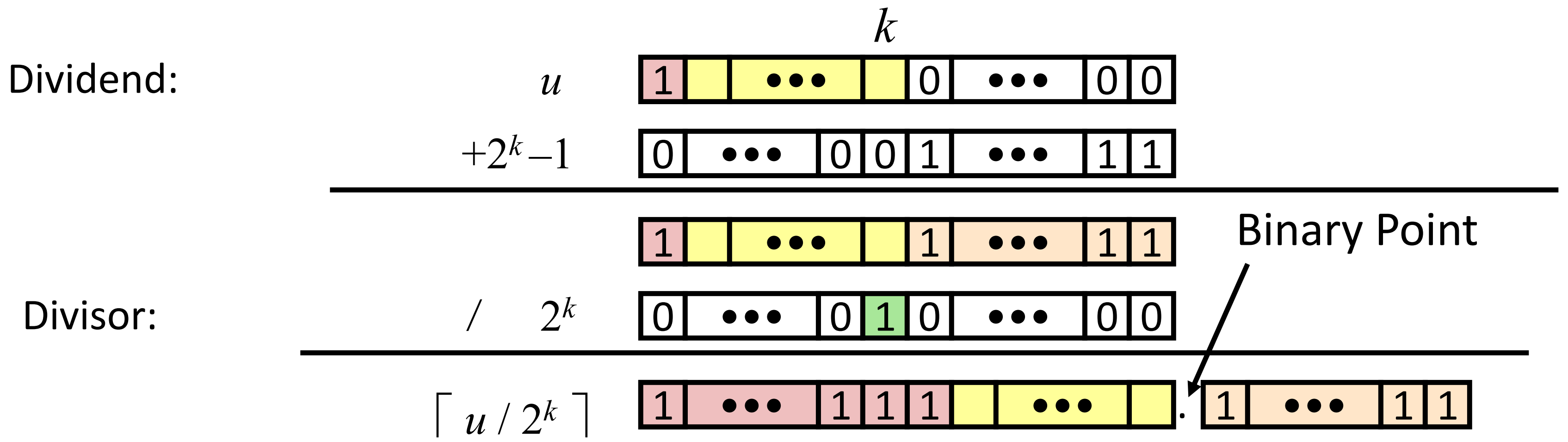


	Division	Computed	Hex	Binary
`y`	-15213	-15213	`C4 93`	`11000100 10010011`
`y >> 1`	-7606.5	-7607	`E2 49`	`11100010 01001001`
`y >> 4`	-950.8125	-951	`FC 49`	`11111100 01001001`
`y >> 8`	-59.4257813	-60	`FF C4`	`11111111 11000100`

带符号数的除法：修正+ 右移

- 负数除以 Power of 2
  - Want $\lceil$ **x / 2**$^k\rceil$ (向零方向舍入)
  - Compute as $\lfloor$ **(x+2**^{k}**−1) / 2**$^k\rfloor$
    - In C: `(x + (1<<k)-1) >> k`

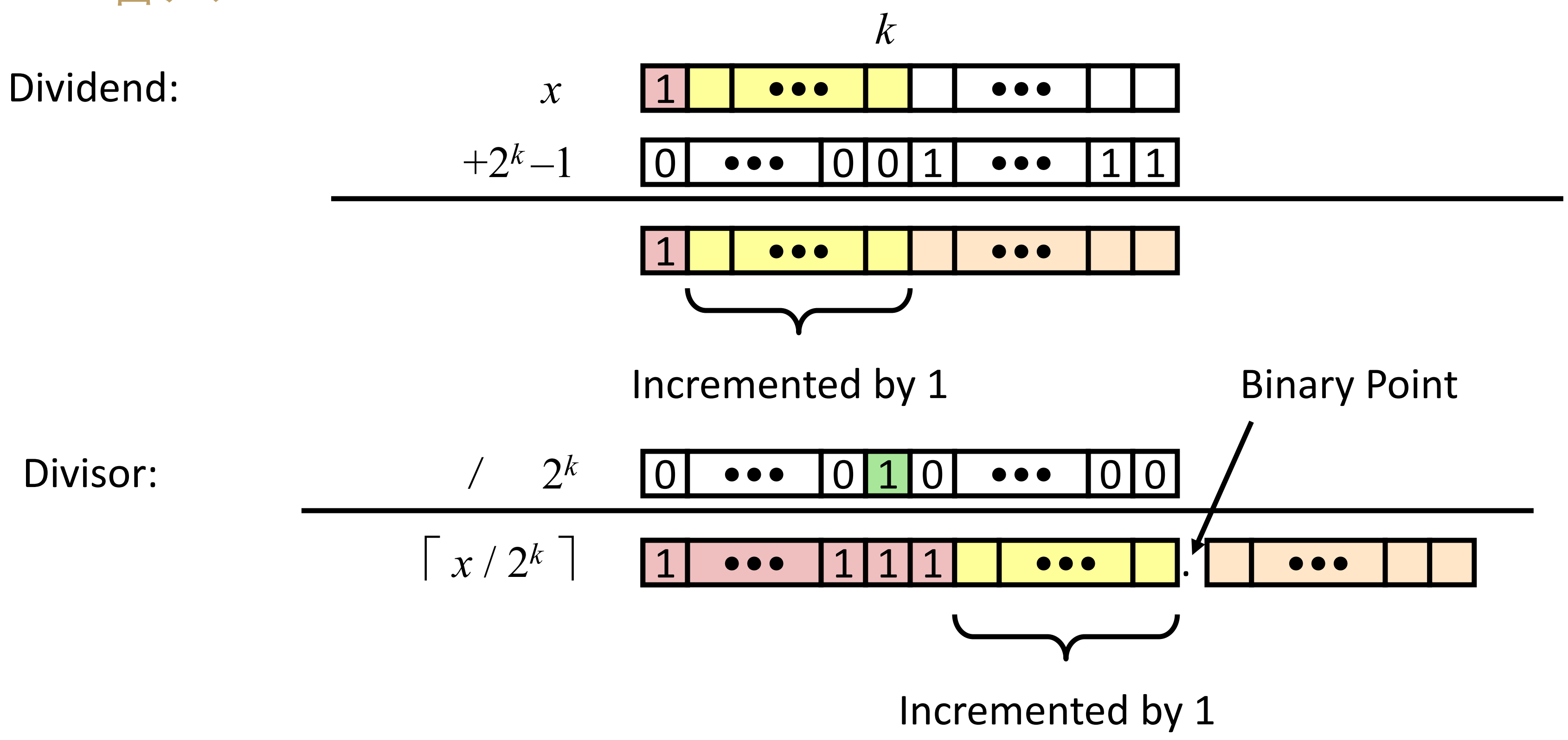
Case 1: 没有舍入



修正量没有起作用

带符号数的除法：修正+ 右移

Case 2: 舍入



修正量为结果加1

编译器对（2^K）除法的优化

C Function

```
long idiv8(long x)
{
   return x/8;
}
```

Compiled Arithmetic Operations

```
   testq %rax, %rax
   js    L4
L3:
   sarq $3, %rax
   ret
L4:
   addq $7, %rax
   jmp   L3
```

Explanation

```
  if x < 0
    x += 7;
  # Arithmetic shift
  return x >> 3;
```

无符号整数变量ux和uy的声明和初始化如下：

```
unsigned ux=x;
unsigned uy=y;
```

若sizeof(int)=4，则对于任意int型变量x和y，判断以下表达式哪些为永真?

- x*y==ux*uy
- (x*x)>=0
- x/4+y/8==(x>>2)+(y>>3)
- x*4+y*8==(x<<2)+(y<<3)

下一节

- 下周二 16： 00
- 浮点数的表示和运算
- 习题为主
- 请做好准备

再见