Computer Organization & Design

Hardware/Software interface

楼学庆

Lou Xueqing

玉泉校区曹光彪东楼507室

Website:10.214.47.99/index.php
Email: xqlou@zju.edu.cn
hzlou@163.com

计算机学院教案：9

- 授课专业：计算机2006 班级 1-2 人数 57
- 授课日期：2008年 3月 25日第 五 周 TUE 第1、2节
- 授课地点：2220
- 授课内容：第三章：数据的计算机表示(1)
- 课时目标：
  - 掌握：正整数的数据表示，二进制、十六进制
  - 熟悉：原码、补码、移码、反码
  - 了解：数码运算
- 教学活动
  - 课　　型：理论√　实验　见习　其它
  - 教学方式：讲授√　讨论　指导　示教　其它
  - 教学资源：多媒体√ 模型　标本　实物　示教　其它
- 教学过程

二进制、十进制、十六进制	20
原码、补码、移码、反码	50
补码加法设计	20

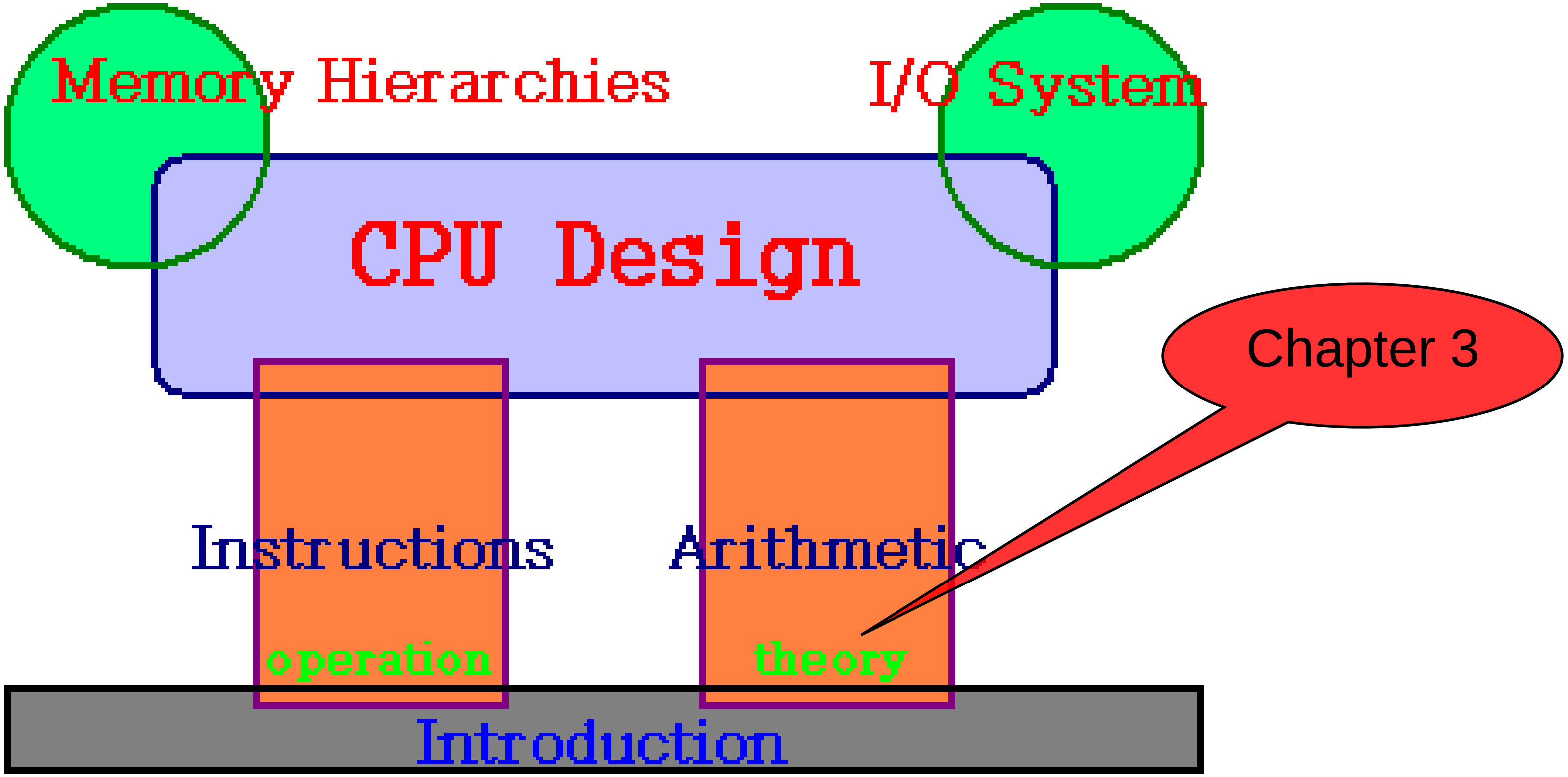
Unit 5:

Arithmetic for Computer

Chapter 3

Chapter 2

- 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……

$$\left.\begin{array}{l}\text{Character}\\ \text{String}\\ \text{Integer}\\ \text{Float}\end{array}\right\} Code = f(Data)$$

$$\left.\begin{array}{l}\text{……}\\ \text{image, music……}\end{array}\right\} \text{Software}$$

Signed number in 2's complement

- Since:

$$Code = f(Data)$$
$$Data = g(Code)$$

- X_{code}, Y_{code} → $(X$ ¿ $Y)_{code}$
  - ☞ We should: f(g(X_{code}) ¿ g(Y_{code}))
  - ☞ Eq. X=sin(x), Y=sin(y) → 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)

      $= X(1-Y*Y)^{½} + Y(1-X*X)^{½}$

Integer: Biased notation(移码)

- $X=M+x$， $Y=M+y$

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

- $(x+y)_{code}=M+(x+y)$
  - ☞ 1. $=M + (X-M) + (Y-M)$
  - ☞ 2. $=X+Y-M$

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

$$X_{移} = M + X$$

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

Binary Representation

- The binary number

$$\xrightarrow{}\ 01011000\ 00010101\ 00101110\ 11100111\ \xleftarrow{}$$

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...2^n-1
- 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: 1's Complement 2's Complement

Sign Magnitude	1's Complement	2's Complement
000 = +0	000 = +0	000 = +0
001 = +1	001 = +1	001 = +1
010 = +2	010 = +2	010 = +2
011 = +3	011 = +3	011 = +3
100 = -0	100 = -3	100 = -4
101 = -1	101 = -2	101 = -3
110 = -2	110 = -1	110 = -2
111 = -3	111 = -0	111 = -1

- Issues: balance, number of zeros, ease of operations
- Which one is best? Why?

BCD code

- Varity
- R:base

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

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

$$x_{移} = 2^{N-1} + x$$

BCD code

- Varity

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?

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

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

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

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

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

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).

$x + x' = -1$
$x' + 1 = -x$ … hence, can compute the negative of a number by
$-x = x' + 1$ 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

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     0xfffff83f
          .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 <= 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

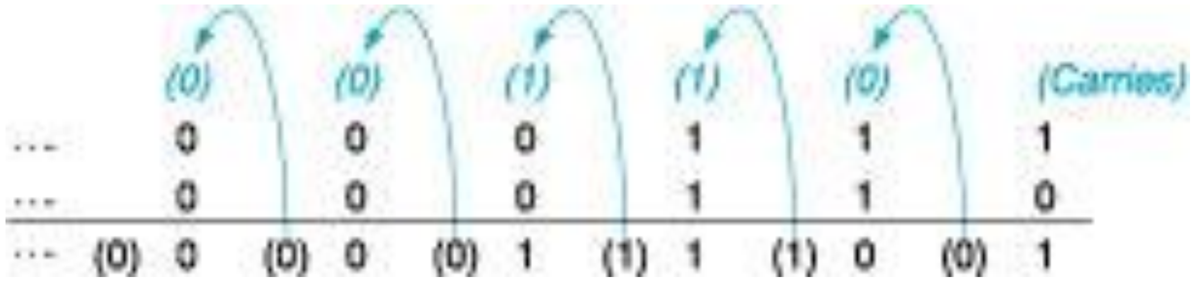
Alternative Representations

- 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

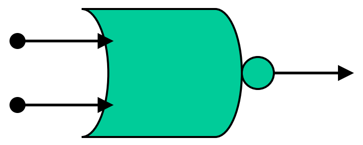
Addition and Subtraction

- 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
  - ☞ S = a 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

计算机学院教案：10

- 授课专业：计算机2006 班级 1-2 人数 57
- 授课日期：2008年 3月 28日第 五 周 TUE 第1、2节
- 授课地点：2220
- 授课内容：第三章：数据的计算机表示(2)
- 课时目标：
  - ☞ 掌握：字符的数据表示
  - ☞ 熟悉：ASCII，汉字内码、字模码、输入码
  - ☞ 了解：汉字显示
- 教学活动
  - ☞ 课　　型：理论√　实验　见习　其它
  - ☞ 教学方式：讲授√　讨论　指导　示教　其它
  - ☞ 教学资源：多媒体√ 模型　标本　实物　示教　其它
- 教学过程

☞ ASCII	20
☞ 汉字内码、字模码、输入码	40
☞ 汉字显示程序	30

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

Addition & Subtraction

- 证明：$$\boxed{X_{补} = \left(2^{N} + X\right) \quad \mathrm{mod} \quad 2^{N}}$$

$$X_{补} + Y_{补} = \left(2^{N} + X + 2^{N} + Y\right) \mathrm{mod} \left(2^{N}\right)$$

$$= (X + Y)_{补}$$

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

$$X_{补} - Y_{补} = \left(2^{N} + X - 2^{N} - Y\right) \mathrm{mod} \left(2^{N}\right)$$

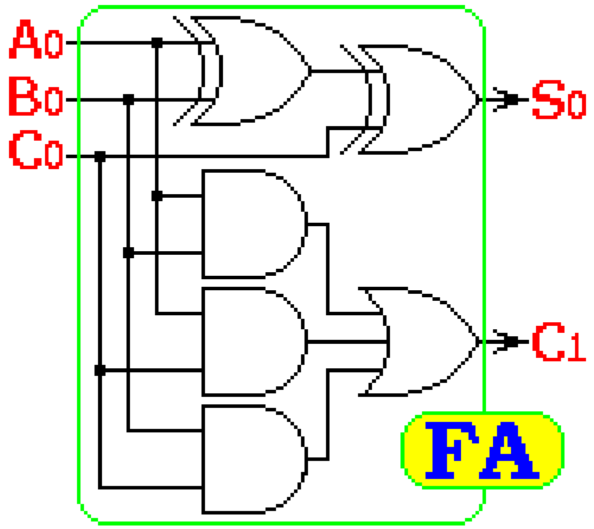
$$= (X - Y)_{补}$$

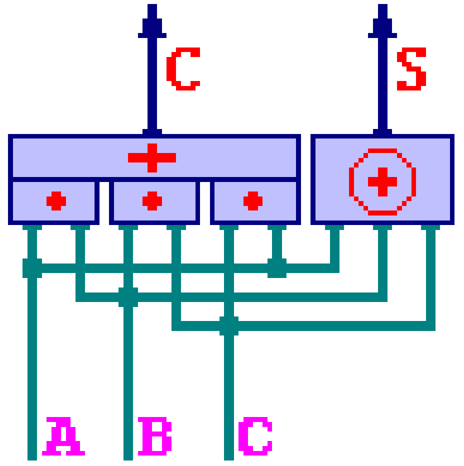
$$\boxed{\therefore (X - Y)_{补} = X_{补} - Y_{补}}$$

Add

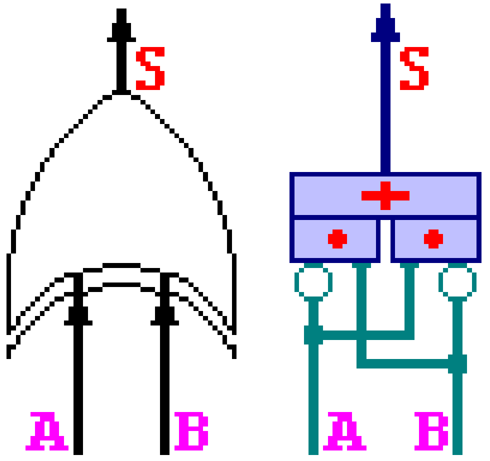
$$C_{i+1} = A_i B_i + B_i C_i + C_i A_i$$

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



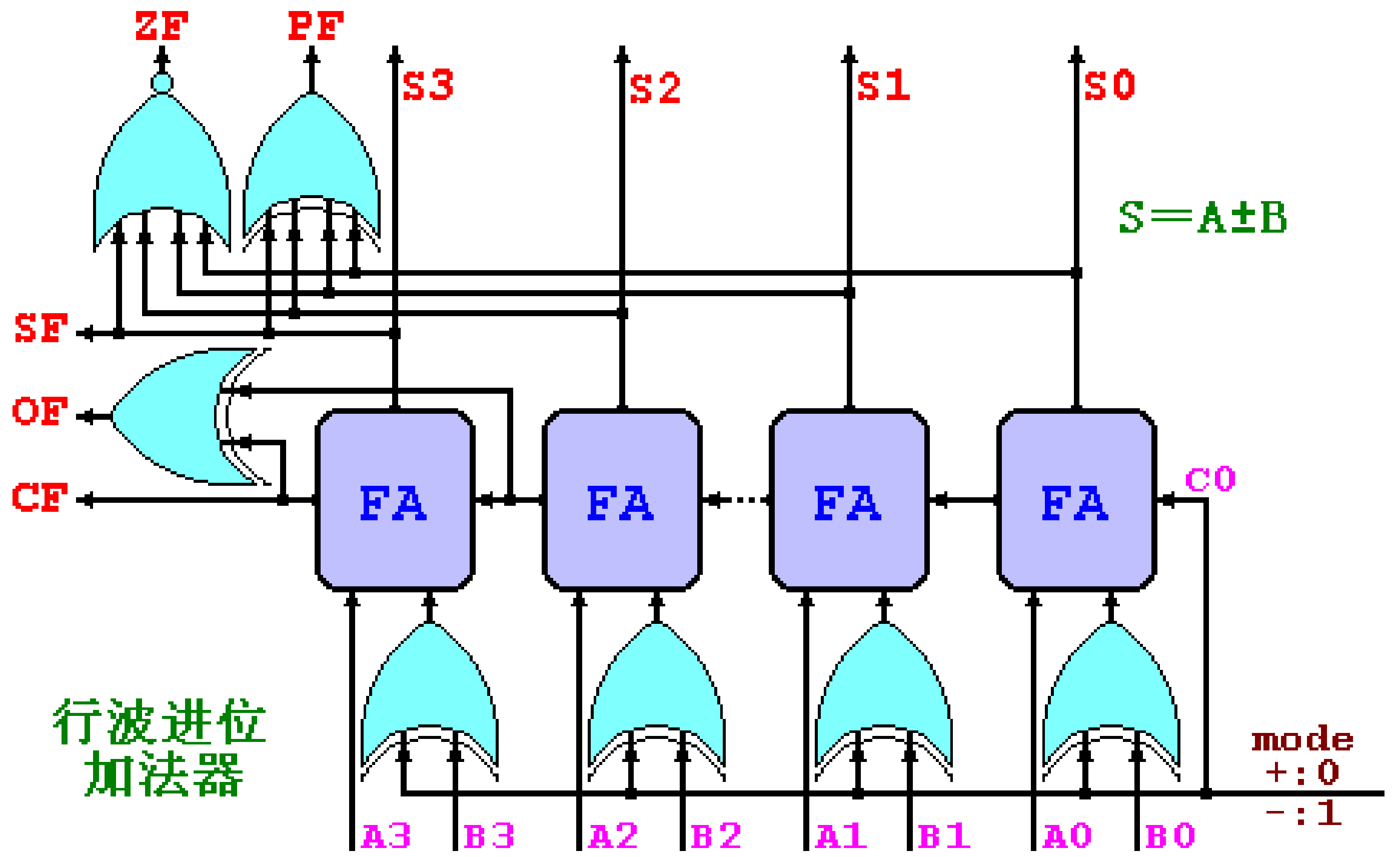


一位全加器



一位半加器

Full Adder


ZF
PF
S3
S2
S1
S0
S=A±B
SF
OF
CF
FA
FA
FA
FA
C0
行波进位
加法器
mode
+:0
-:1
A3
B3
A2
B2
A1
B1
A0
B0

Purpose

- Code: 0000…00 ~ 1111…11
- Data: real data:

$$\left.\begin{array}{l}\text{☞Character}\\ \text{☞String}\\ \text{☞Integer}\\ \text{☞Float}\end{array}\right\} Code = f(Data)$$

$$\left.\begin{array}{l}\text{☞……}\\ \text{☞image, music……}\end{array}\right\} \text{Software}$$

Signed number in 2's complement

- Since:

$$Code = f(Data)$$
$$Data = g(Code)$$

- X_{code}, Y_{code} → $(X$ ¿ $Y)_{code}$
  - ☞ We should: $f(g(X_{code})$ ¿ $g(Y_{code}))$
  - ☞ Eq. X=sin(x), Y=sin(y) → 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)
      $= X(1-Y*Y)^{½} + Y(1-X*X)^{½}$

Integer: Biased notation(移码)

- X=M+x， Y=M+y

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

- $(x+y)_{code}=M+(x+y)$
  - ☞ 1. =M + (X-M) + (Y-M)
  - ☞ 2. =X+Y-M

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

$$X_{移} = M + X$$

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 atoi(char*);`
  - `char* itoa(int);`
  - `int iadd(int, int);`
  - `int isub(int, int);`
  - `int imul(int, int);`
  - `int idiv(int, int);`
  - `int imod(int, int);`
- 进位、溢出
- 扩展
- 大小比较

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);
- 溢出
- 扩展
- 四舍五入
- 大小比较

- 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
  - ☞ 。。。

Binary Multiplication & Division

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

Multiplication Example

```
Multiplicand              1000
Multiplier             x   1001
                       ---------------
                            1000
                          0000
                        0000
                       1000
                      ----------------
```

Product 1001000_{ten}

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

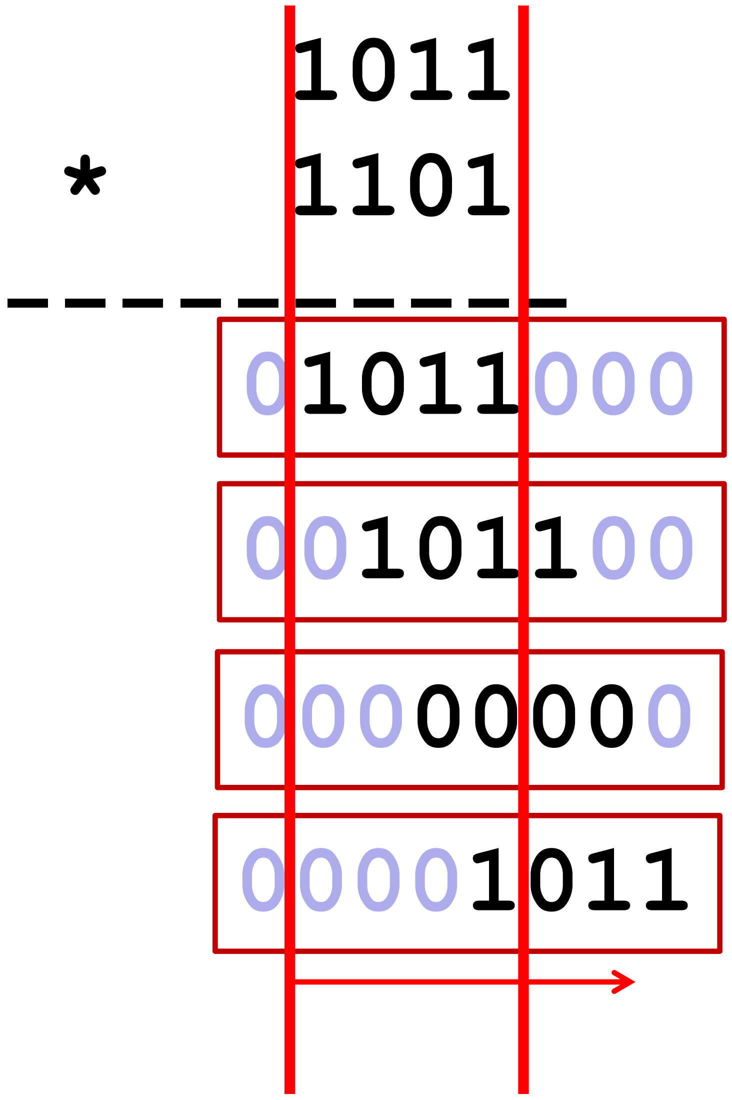
MUL

- MUL

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

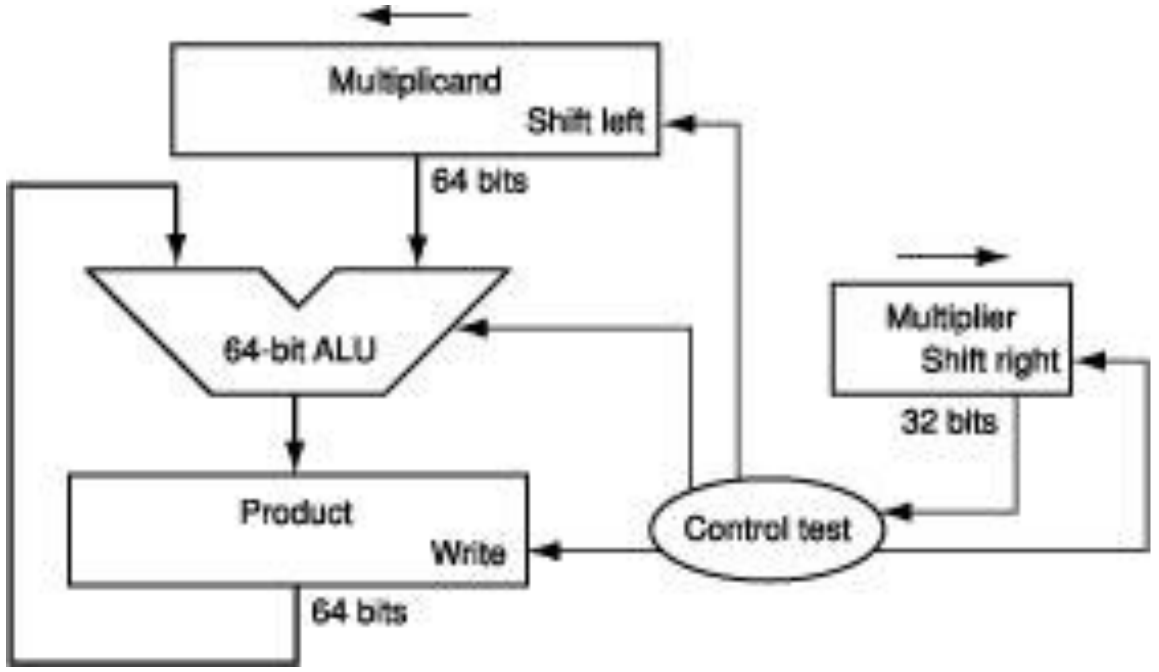
MUL

- MUL



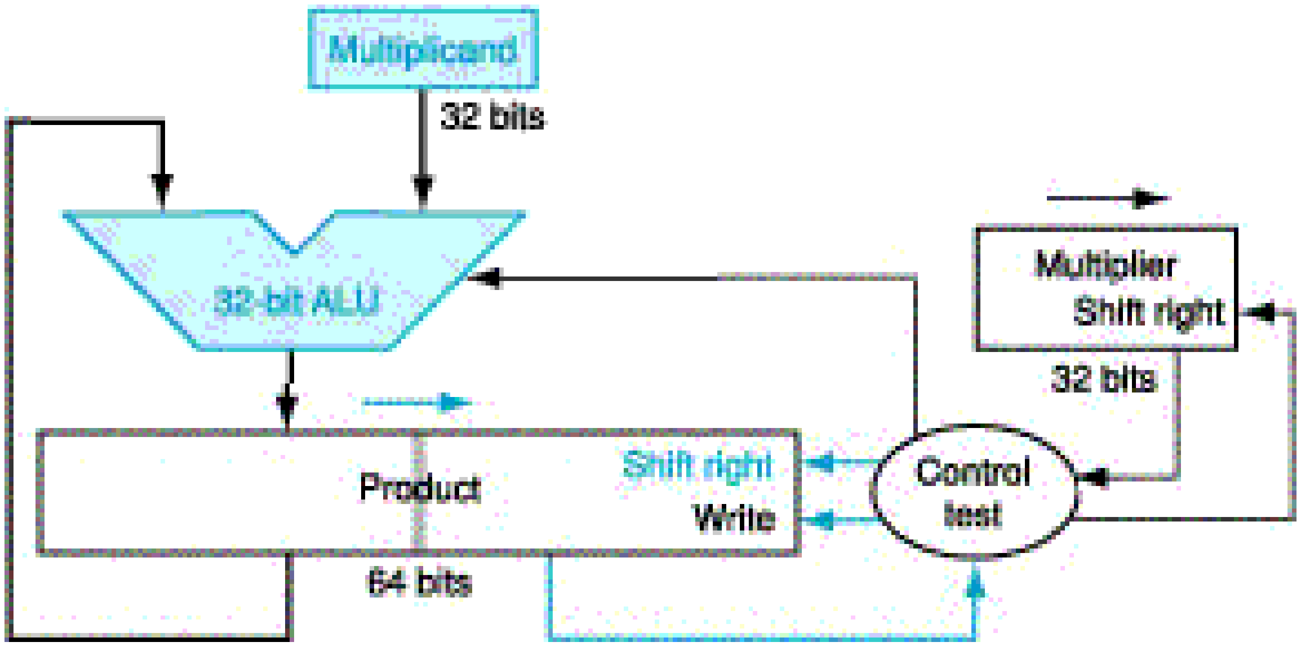
```
        1011
*       1101
----------
    01011000
    00101100
    00000000
    00001011
```

HW Algorithm 1



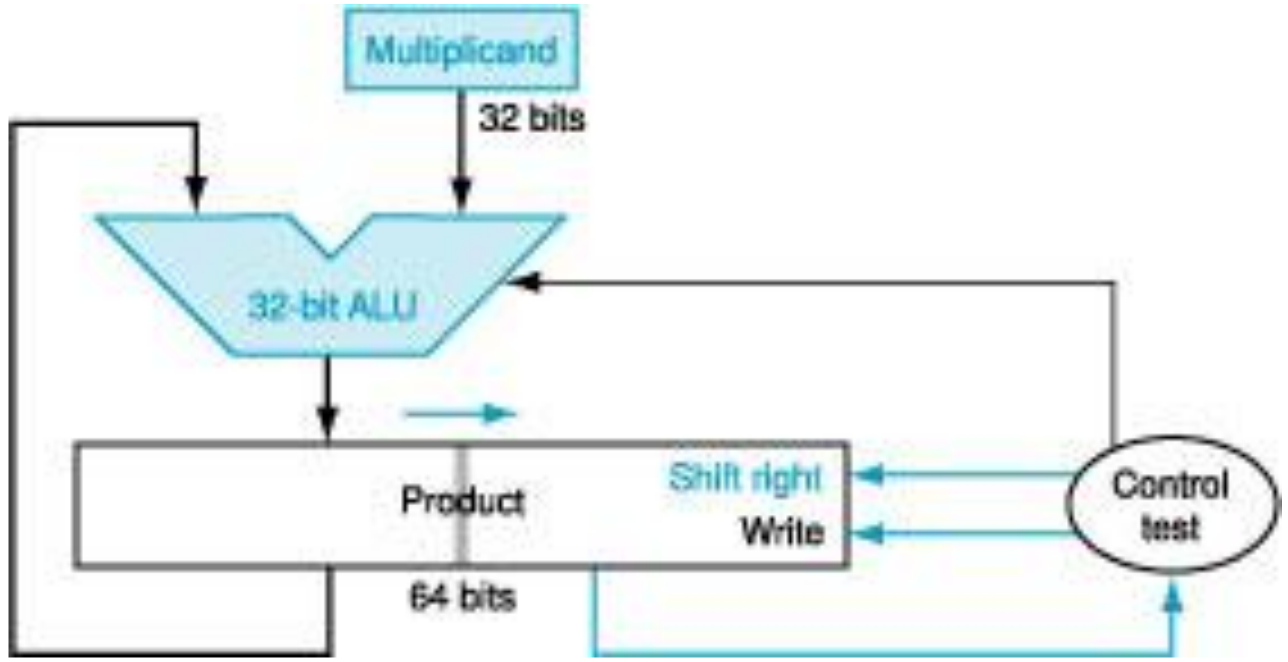
In every step
- multiplicand is shifted
- next bit of multiplier is examined (also a shifting step)
- if this bit is 1, shifted multiplicand is added to the product

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} → $(X*Y)_{code}$
  - ☞ We should: $f(g(X_{code})*g(Y_{code}))$

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

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

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

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

$$+\ldots\ldots$$

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

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

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

$\{-1, 1\}$

Signed number in 2's complement

- For 32-bit: $X_{code} = a_{31}a_{30}a_{29}\ldots a_1 a_0$ {a:0/1}
- The value $X = -a_{31}2^{31}+a_{30}2^{30}+a_{29}2^{29}+\ldots+a_12^1+a_02^0$
- $(X*Y)_{code} = X*Y_{code}$

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

$=(-2a_{31}2^{30}+ a_{30}2^{30}+ a_{29}2^{30}- 2a_{29}2^{28}+ a_{28}2^{28}+ a_{27}2^{28}-$
$...-2a_32^2+ a_22^2+ a_12^2-2a_12^0+ a_02^0+ 0*2^0)*Y_{code}$

$=\{(-2a_{31}+a_{30}+a_{29})2^{30}$

$+(-2a_{29}+a_{28}+a_{27})2^{28}$

$+......$

$+(-2a_3+a_2+a_1)2^2$

$+(-2a_1+a_0+0)2^0\}*Y_{code}$

{-2, 1, 1}

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

1-bit Booth's Algorithm

例：a=1101　b=1011　求：a*b

解：

$a_{补}=01101$

$b_{补}=01011$

$-b_{补}=10101$

```
        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
```

{-1, 1}

2-bit Booth's Algorithm

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

例：A＝1011，B＝－1101　　求：A×B

```
解：    A补=001011            000000  001011 0
        B补=110011        -b  001101
       -B补=001101            001101  001011
       2B补=100110        ->  000011  010010 1
      -2B补=011010        -b  001101
                              010000  010010
                          ->  000100  000100 1
                          +b  110011
                              110111  000100
                          ->  111101  110001
```

{-2, 1, 1}

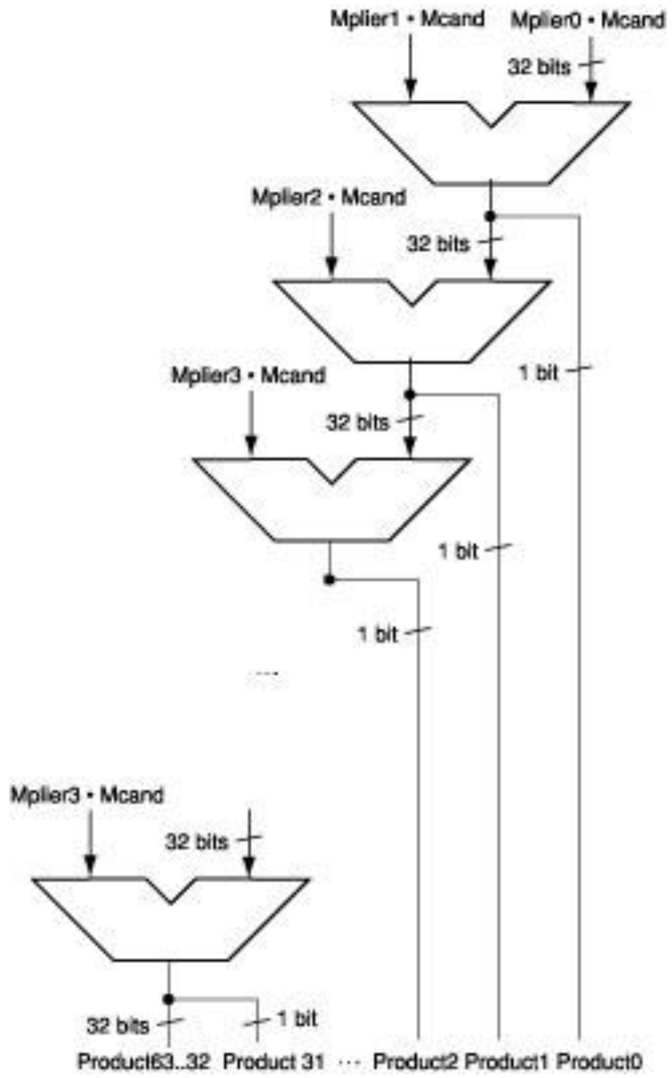
2-bit Booth's Algorithm

MIPS Instructions

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

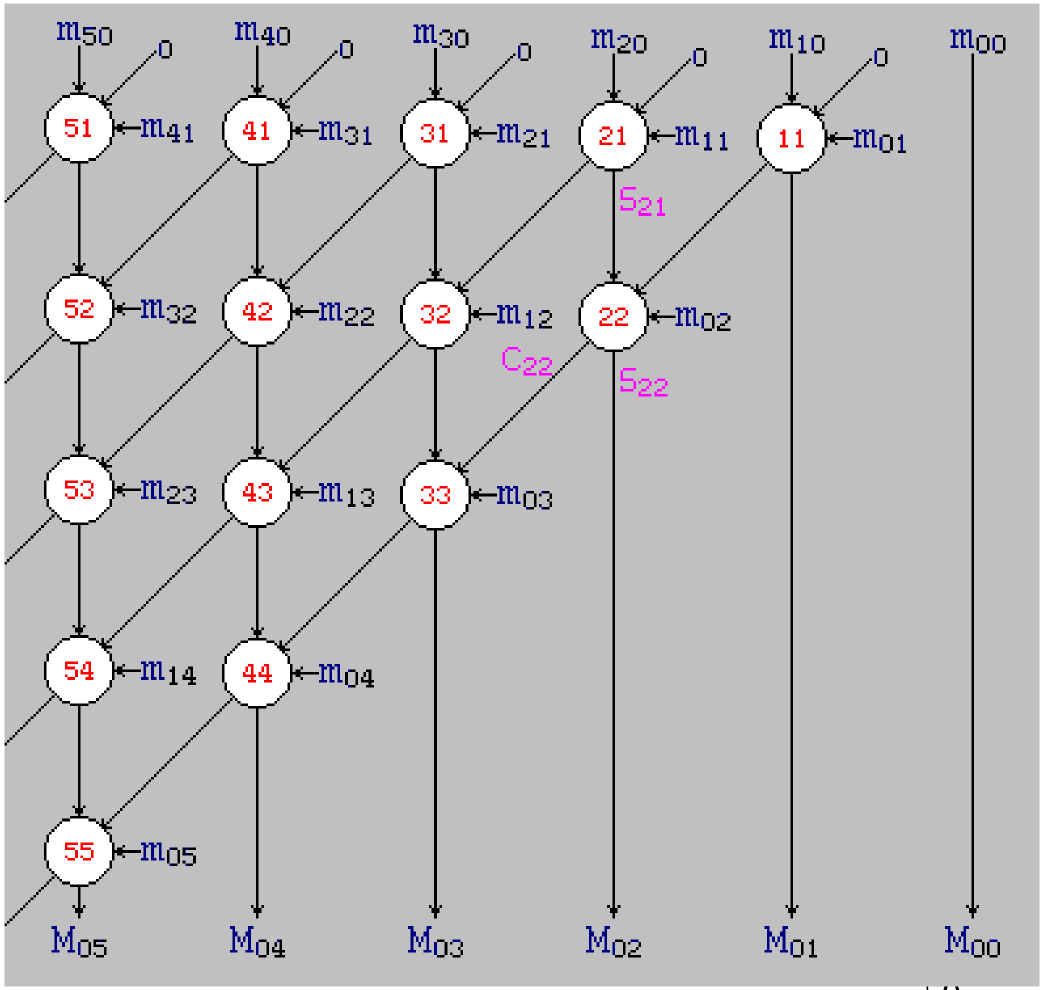
- This algorithm can quickly set up most inputs – it then has to wait for the result of each add to propagate down – faster because no clock is involved

  -- Note: high transistor cost

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



Division

$$\begin{array}{llrl}
 & & \overline{\qquad 1001_{ten}} & \text{Quotient} \\
\text{Divisor} & 1000_{ten} & |\quad 1001010_{ten} & \text{Dividend} \\
 & & \underline{-1000} & \\
 & & 10 & \\
 & & 101 & \\
 & & 1010 & \\
 & & \underline{-1000} & \\
 & & 10_{ten} & \text{Remainder}
\end{array}$$

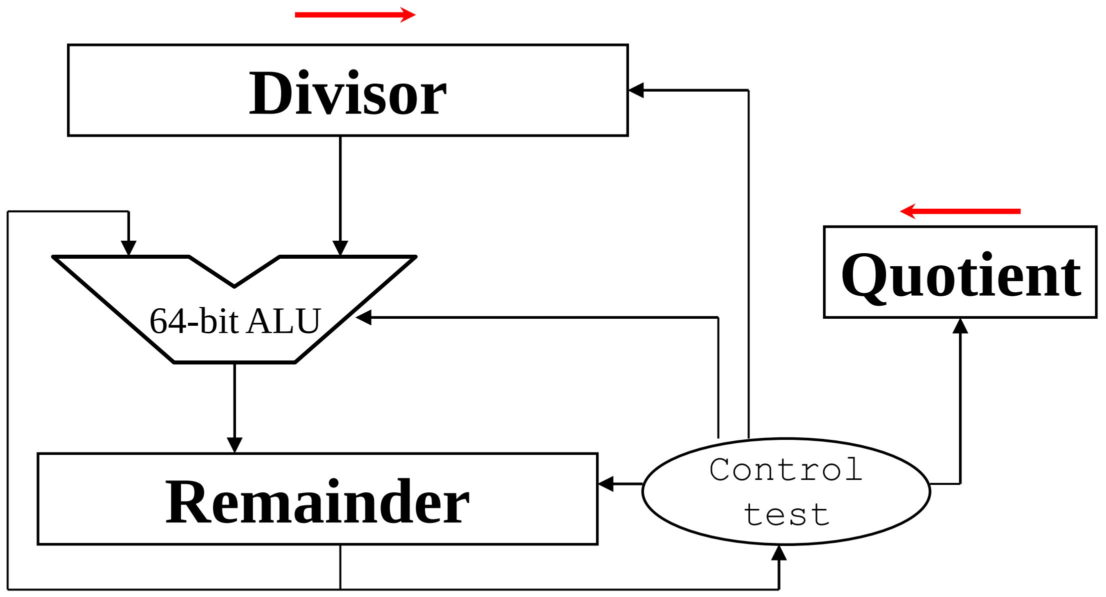
At every step,

- 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:

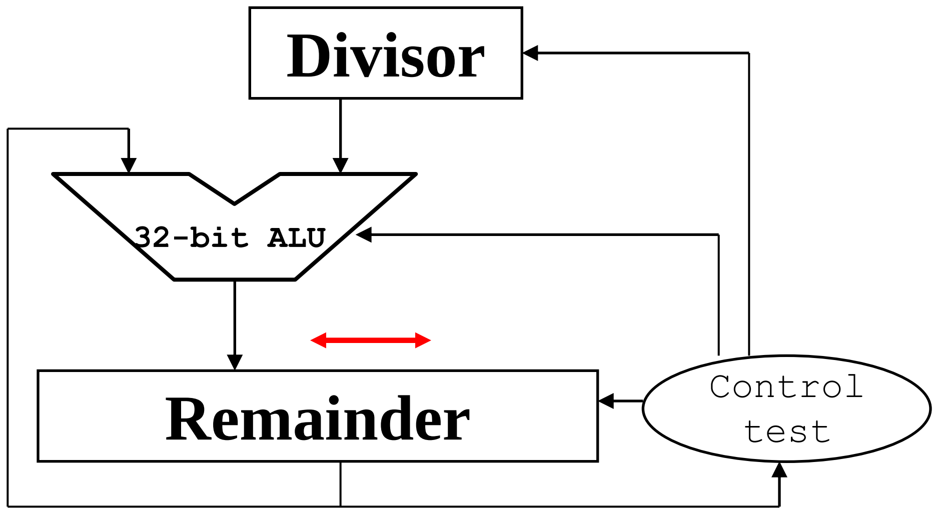
  Dividend=Quotient * Divisor + Remainder



Division

- Division:

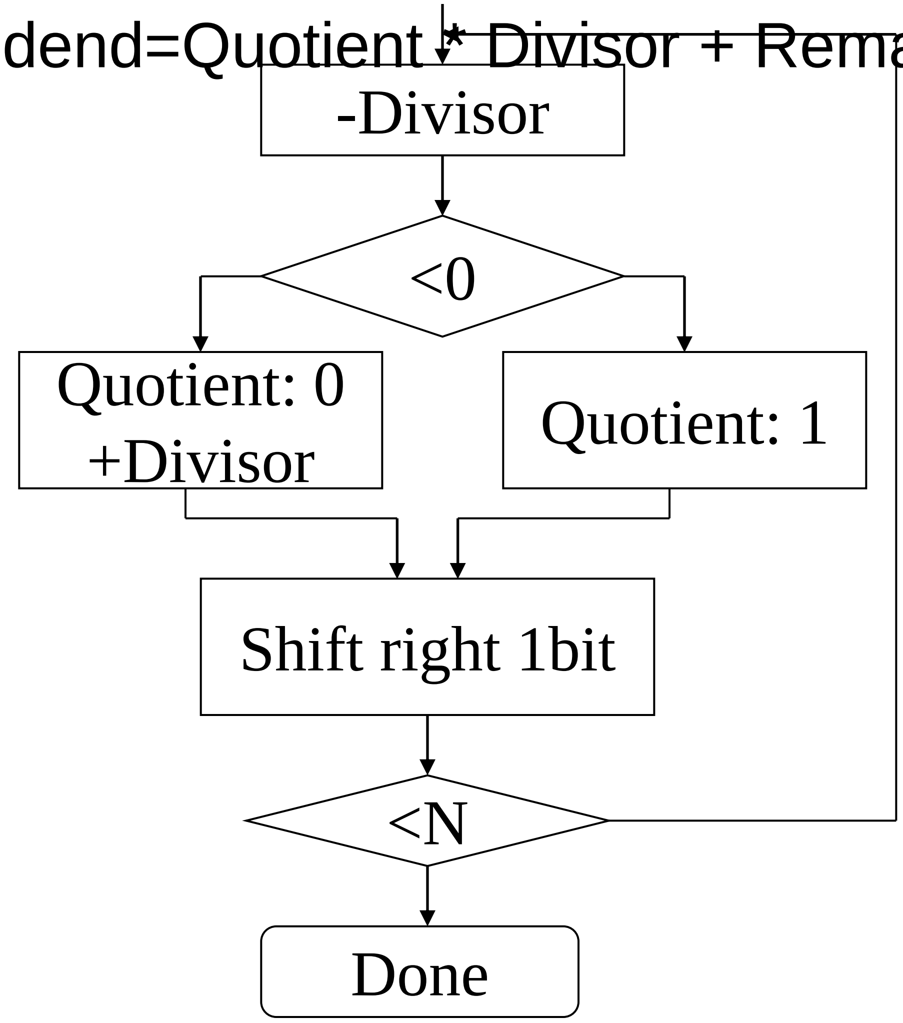
  Dividend=Quotient * Divisor + Remainder



Division

- Division:

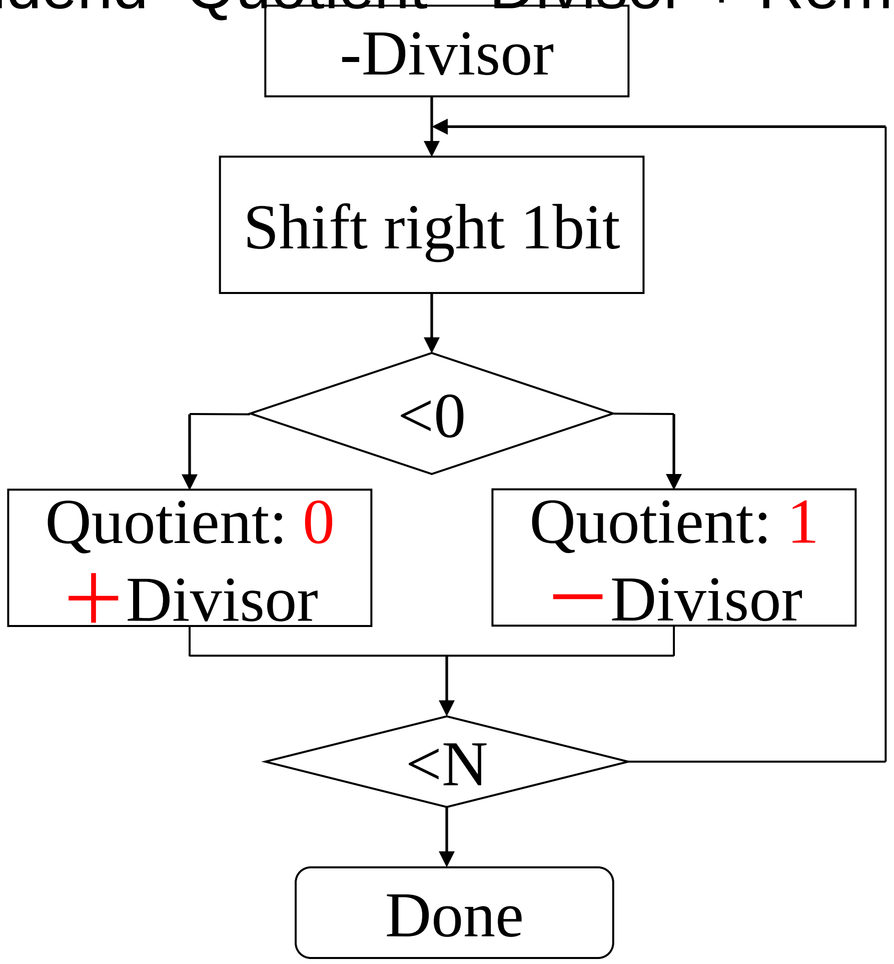
  Dividend=Quotient * Divisor + Remainder

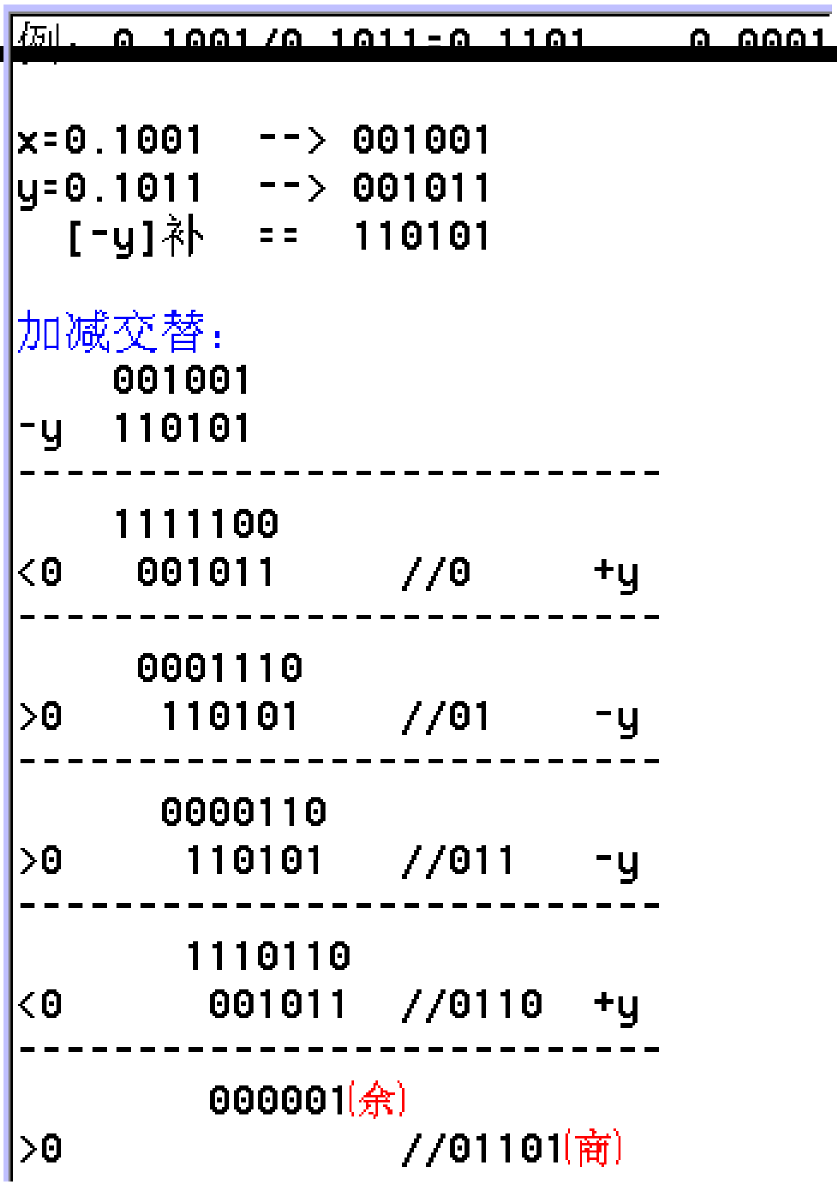


Division

- Division:

  Dividend=Quotient * Divisor + Remainder





```
例: 0.1001/0.1011=0.1101      0.0001

x=0.1001   --> 001001
y=0.1011   --> 001011
  [-y]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

```
                                      1001ten        Quotient
Divisor      1000ten      |      1001010ten          Dividend

   0001001010        0001001010      0000001010    0000001010
100000000000 →    0001000000→    0000100000→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_{ten} ($0000\ 0111_{two}$) by 2_{ten} (0010_{two})

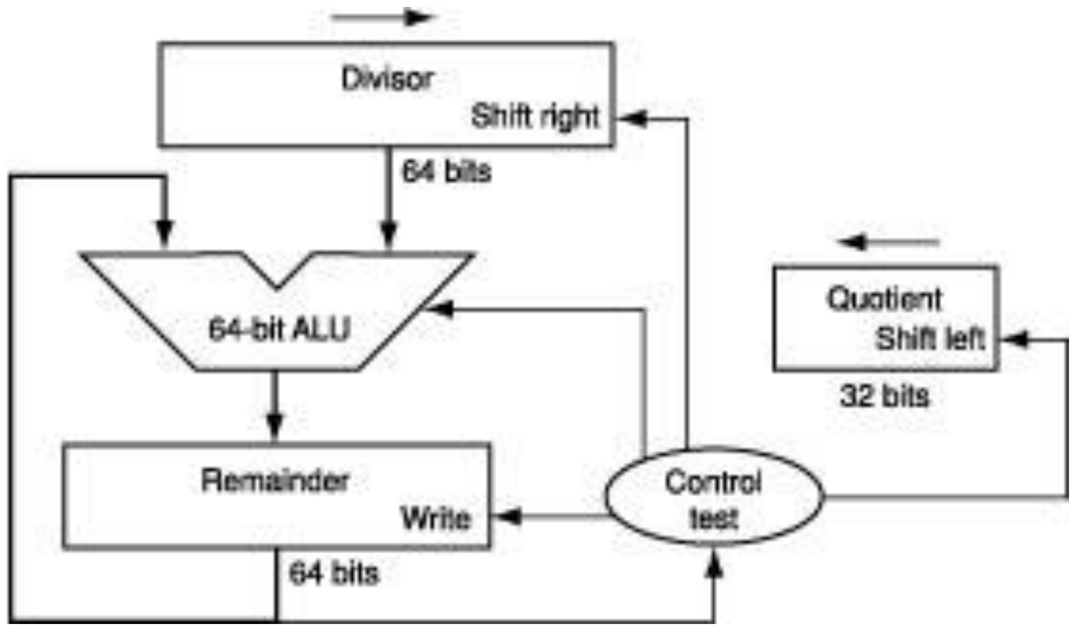
Iter	Step	Quot	Divisor	Remainder
0	Initial values			
1				
2				
3				
4				
5				

Divide Example

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

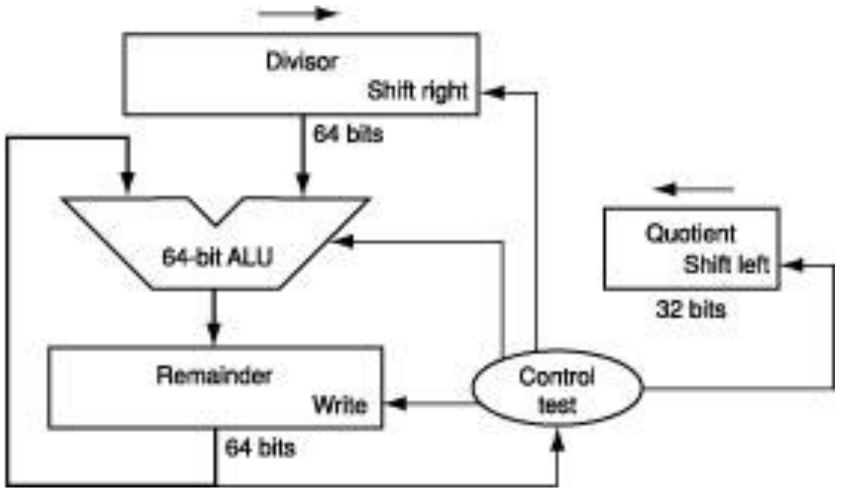
Iter	Step	Quot	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	Rem = Rem – Div	0000	0010 0000	1110 0111
	Rem < 0 $\Rightarrow$ +Div, shift 0 into Q	0000	0010 0000	0000 0111
	Shift Div right	0000	0001 0000	0000 0111
2	Same steps as 1	0000	0001 0000	1111 0111
		0000	0001 0000	0000 0111
		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 >= 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

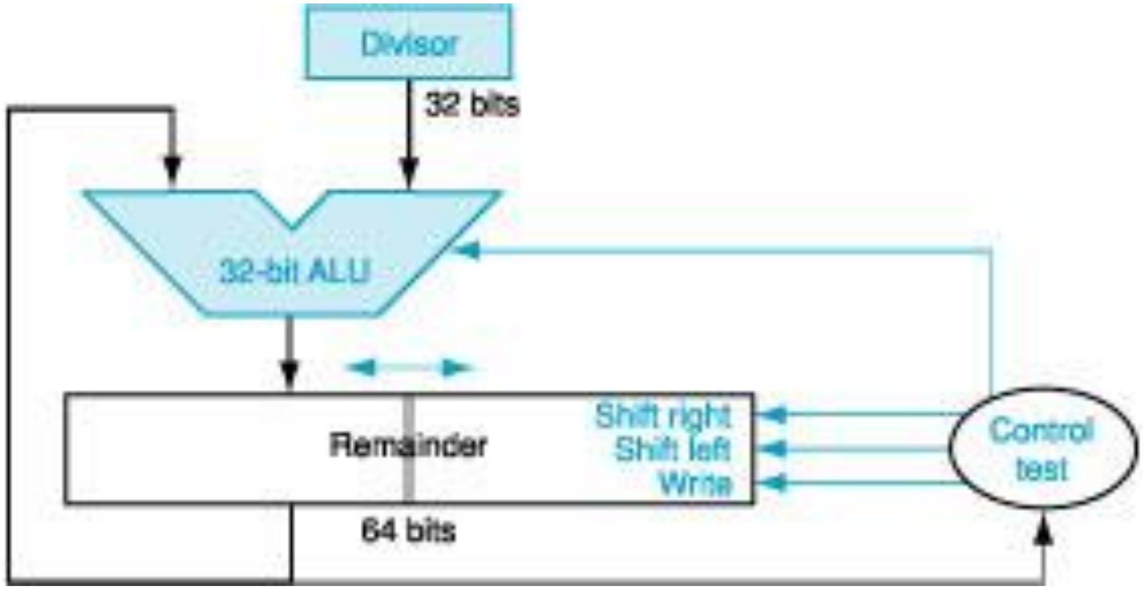
Hardware for Division



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:

  Dividend = Quotient x Divisor + 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:
  Dividend = Quotient x Divisor + 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

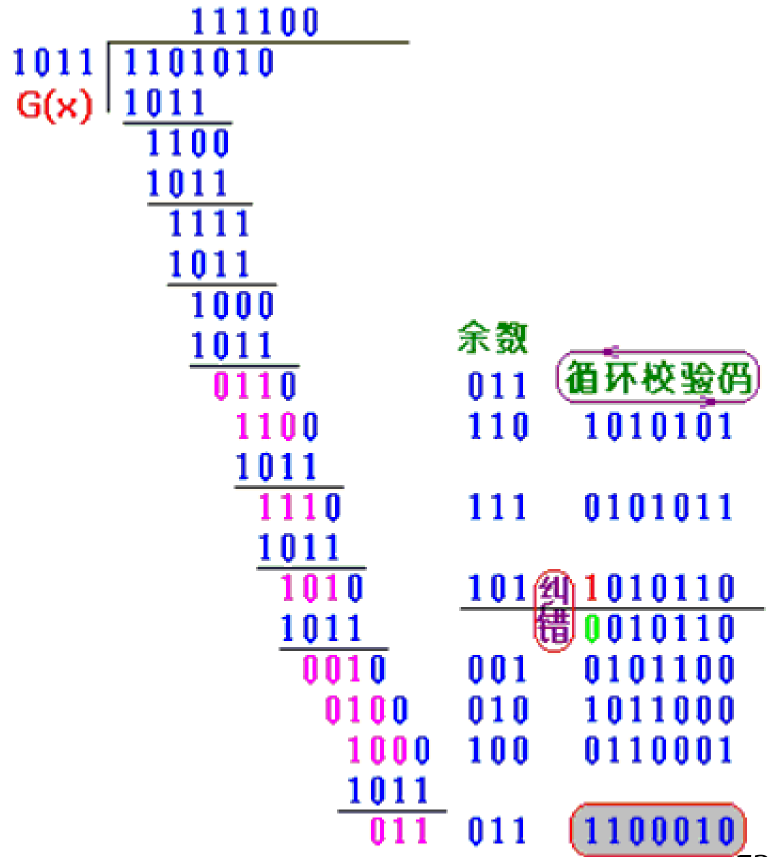
CRC

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：
  1011
  - G(x)=X³+X+1　　　即：1011。
  - 扩展M：　1100 000
  - 模2除G：1100000 / 1011＝1110……010
  - CRC=1100 010。

校验与纠错:

设有位错：1101010



```
            111100
1011 | 1101010
G(x) | 1011
        1100
        1011
         1111
         1011
          1000
          1011
           0110
            1100
            1011
             1110
             1011
              1010
              1011
               0010
                0100
                 1000
                 1011
                  011
```

余数	循环校验码
011	
110	1010101
111	0101011
101	纠错 1010110
	0010110
001	0101100
010	1011000
100	0110001
011	1100010

Floating Point

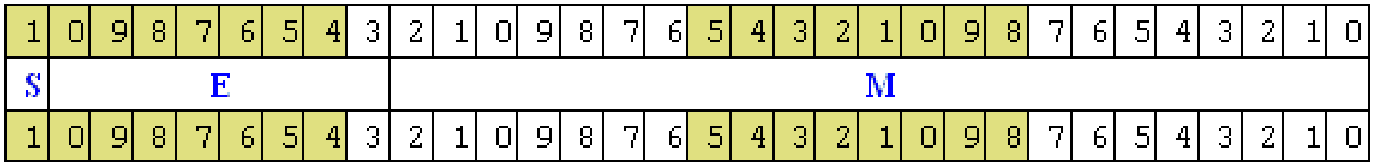
- Today's topics:
  - Division
  - IEEE 754 representations
  - FP arithmetic
- Reminder: assignment 4 will be posted later today

Floating Point (a brief look)

- 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 ´ 10^9
- Representation:
  - sign, exponent, significand: $(–1)^{sign}$ ´ significand ´ $2^{exponent}$
  - more bits for significand gives more accuracy
  - more bits for exponent increases range
- IEEE 754 floating point standard:
  - single precision: 8 bit exponent, 23 bit significand
  - double precision: 11 bit exponent, 52 bit significand

Floating Point (a brief look)

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



	Single	Double
S:	1	1
E:	8	11
M:	23	52

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: $\mathbf{(-1)^{sign} \times (1+significand) \times 2^{exponent - bias}}$

- Example:
  - ☞ decimal: $-.75 = -3/4 = -3/2^2$
  - ☞ binary: $-.11 = -1.1 \times 2^{-1}$
  - ☞ floating point: exponent = 126 = 01111110
  - ☞ IEEE single precision:
    10111111010000000000000000000000

- k

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

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

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

$$E = e + 127$$

E	M	X
0	0	0
0	≠0	X
0<E<255	any	X
255	0	infinity
255	any	not a number

Floating Point Complexities

- Operations are somewhat more complicated (see text)
- 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
- Implementing the standard can be tricky
- Not using the standard can be even worse
  - ☞ see text for description of 80x86 and Pentium bug!

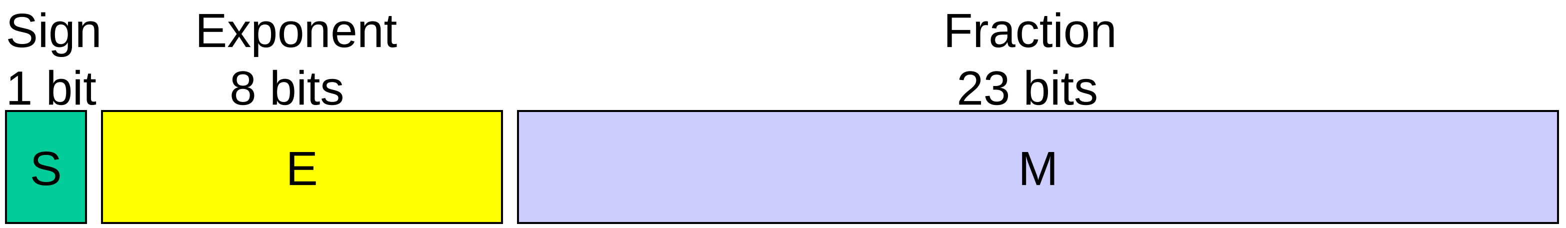
Floating Point

- Normalized scientific notation: single non-zero digit to the left of the decimal (binary) point – example: 3.5×10^{9}

- $1.010001 \times 2^{-5}_{two} = (1 + 0 \times 2^{-1} + 1 \times 2^{-2} + \ldots + 1 \times 2^{-6}) \times 2^{-5}_{ten}$

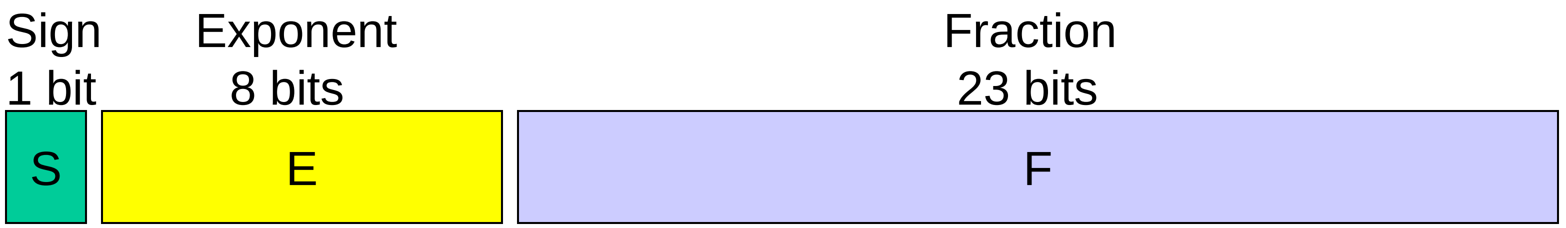
- A standard notation enables easy exchange of data between machines and simplifies hardware algorithms – the IEEE 754 standard defines how floating point numbers are represented

Sign and Magnitude Representation



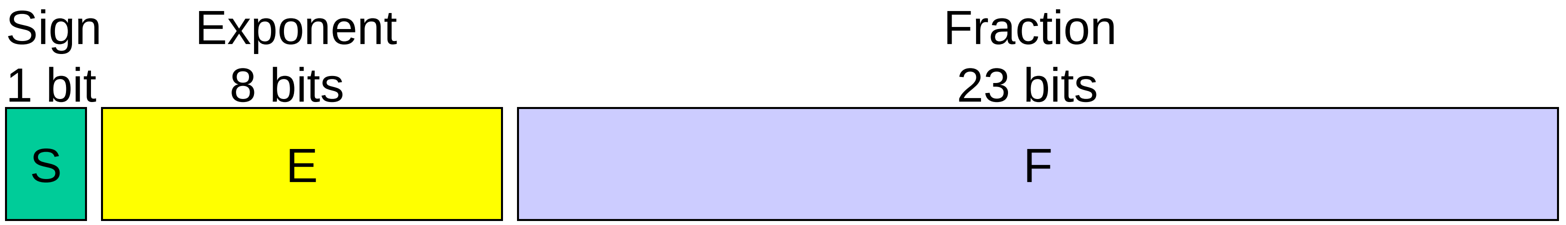
- More exponent bits ➔ wider range of numbers (not necessarily more numbers – recall there are infinite real numbers)
- More fraction bits ➔ higher precision
- Register value = $(-1)^S \times 1.M \times 2^{E-127}$
- Since we are only representing normalized numbers, we are guaranteed that the number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

Sign and Magnitude Representation

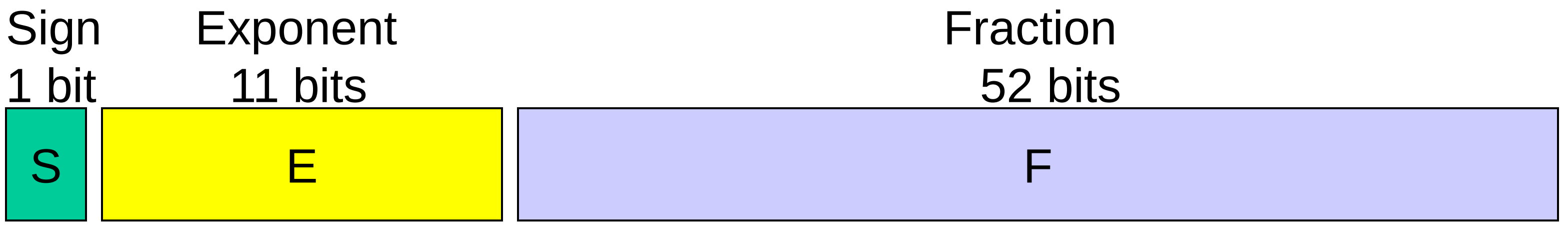


- Largest number that can be represented:
- Smallest number that can be represented:

Sign and Magnitude Representation



- Largest number that can be represented: 2.0 x 2^{128} = 2.0 x 10^{38}
- Smallest number that can be represented: 2.0 x 2^{-128} = 2.0 x 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:
  Largest: Smallest:



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

- Represent -0.75_{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)

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

Convert to the larger exponent:

$9.999 \times 10^{1} \quad + \quad 0.016 \times 10^{1}$

Add

10.015×10^{1}

Normalize

1.0015×10^{2}

Check for overflow/underflow

Round

1.002×10^{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

10.015×10^1

Normalize

1.0015×10^2

Check for overflow/underflow

Round

1.002×10^2

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$_{ten}$ 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_{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)

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

Convert to the larger exponent:

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

Add

10.015×10^1

Normalize

1.0015×10^2

Check for overflow/underflow

Round

1.002×10^2

Re-normalize

FP Addition

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

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

Convert to the larger exponent:

$9.999 \times 10^{1} \quad + \quad 0.016 \times 10^{1}$

Add

10.015×10^{1}

Normalize

1.0015×10^{2}

Check for overflow/underflow

Round

1.002×10^{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

Performance$_X$ = 1 / 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 / 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 Nth 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

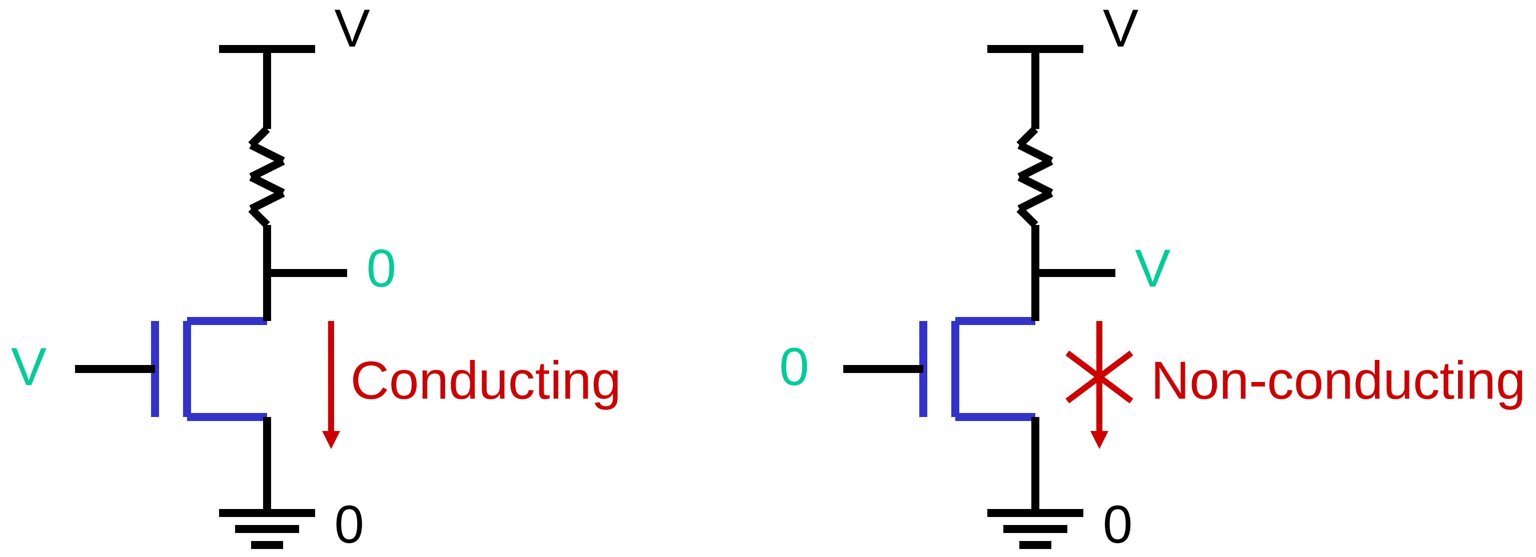
- Architecture design is very bottleneck-driven – make the common case fast, do not waste resources on a component that has little impact on overall performance/power
- Amdahl’s Law: performance improvements through an enhancement is limited by the fraction of time the enhancement comes into play
- Example: 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

A	B	C	E

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 → X is true if at least one of A or B is true
  - AND : symbol **.** , X = A **.** B → X is true if both A and B are true
  - NOT : symbol $\bar{\ }$, X = $\bar{A}$ → X is the inverted value of A

Boolean Algebra Rules

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

DeMorgan's Laws

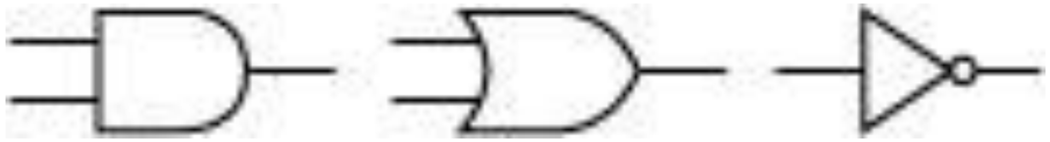
- $\overline{A + B} = \overline{A} \cdot \overline{B}$

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

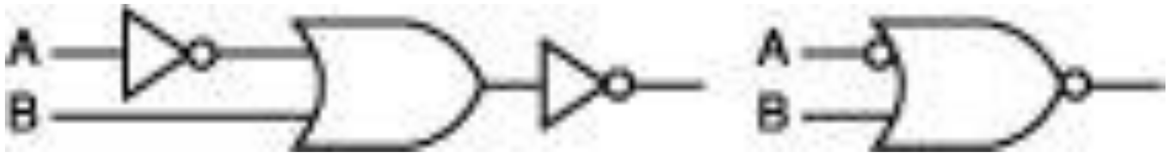
- Confirm that these are indeed true

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:
  $E = ((A\ .\ B) + (B\ .\ C) + (A\ .\ C))\ .\ \overline{(A\ .\ B\ .\ C)}$

  Identify the three cases where it is true:
  $E = (A\ .\ B\ .\ \overline{C}) + (A\ .\ C\ .\ \overline{B}) + (C\ .\ B\ .\ \overline{A})$

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
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

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

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

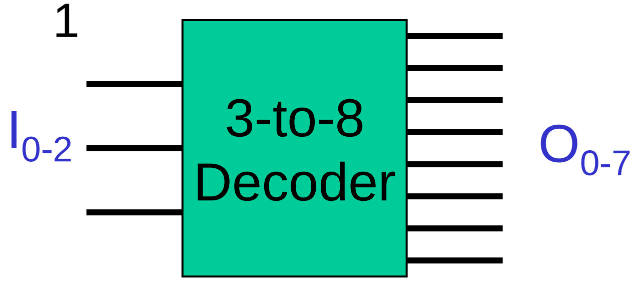
NAND and NOR

- NAND : NOT of AND : A nand B = $\overline{A . B}$
- NOR : NOT of OR : A 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

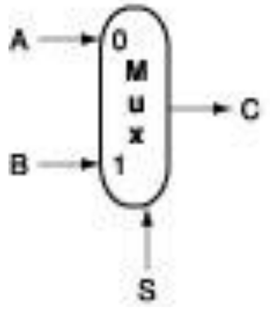
I_0	I_1	I_2	O_0	O_1	O_2	O_3	O_4	O_5	O_6	O_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

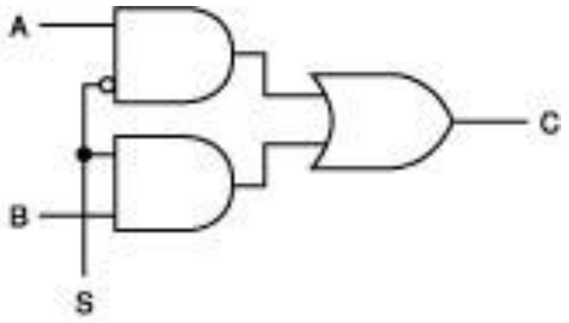


Common Logic Blocks – Multiplexor

- 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

$$(A \,.\, B \,.\, \overline{C}) + (A \,.\, C \,.\, \overline{B}) + (C \,.\, B \,.\, \overline{A})$$

- 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

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

Equations:

$$\begin{aligned} Sum = {} & Cin \,.\, \overline{A} \,.\, \overline{B} \;+ \\ & B \,.\, \overline{Cin} \,.\, \overline{A} \;+ \\ & A \,.\, \overline{Cin} \,.\, \overline{B} \;+ \\ & A \,.\, B \,.\, Cin \end{aligned}$$

$$\begin{aligned} Cout = {} & A \,.\, B \,.\, Cin \;+ \\ & A \,.\, B \,.\, \overline{Cin} \;+ \\ & A \,.\, Cin \,.\, \overline{B} \;+ \\ & B \,.\, Cin \,.\, \overline{A} \\ = {} & A \,.\, B \;+ \\ & A \,.\, Cin \;+ \\ & B \,.\, Cin \end{aligned}$$

Carry Out Logic

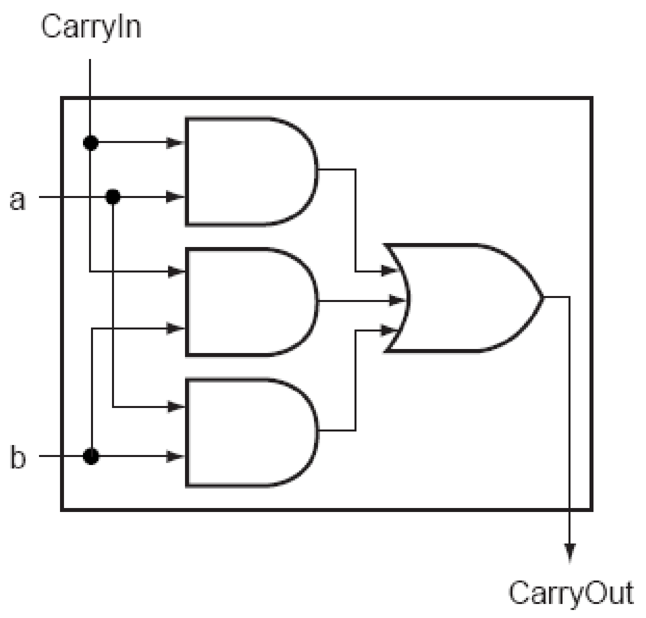


FIGURE B.5.5 Adder hardware for the carry out signal. The rest of the adder hardware is the logic for the Sum output given in the equation on page B-28.

Equations:

$$\begin{aligned} Sum = {} & Cin \,.\, \overline{A} \,.\, \overline{B} + \\ & B \,.\, \overline{Cin} \,.\, \overline{A} + \\ & A \,.\, \overline{Cin} \,.\, \overline{B} + \\ & A \,.\, B \,.\, Cin \end{aligned}$$

$$\begin{aligned} Cout = {} & A \,.\, B \,.\, Cin + \\ & A \,.\, B \,.\, \overline{Cin} + \\ & A \,.\, Cin \,.\, \overline{B} + \\ & B \,.\, Cin \,.\, \overline{A} \\ = {} & A \,.\, B + \\ & A \,.\, Cin + \\ & B \,.\, Cin \end{aligned}$$

1-Bit ALU with Add, Or, And

- Multiplexor selects between Add, Or, And operations

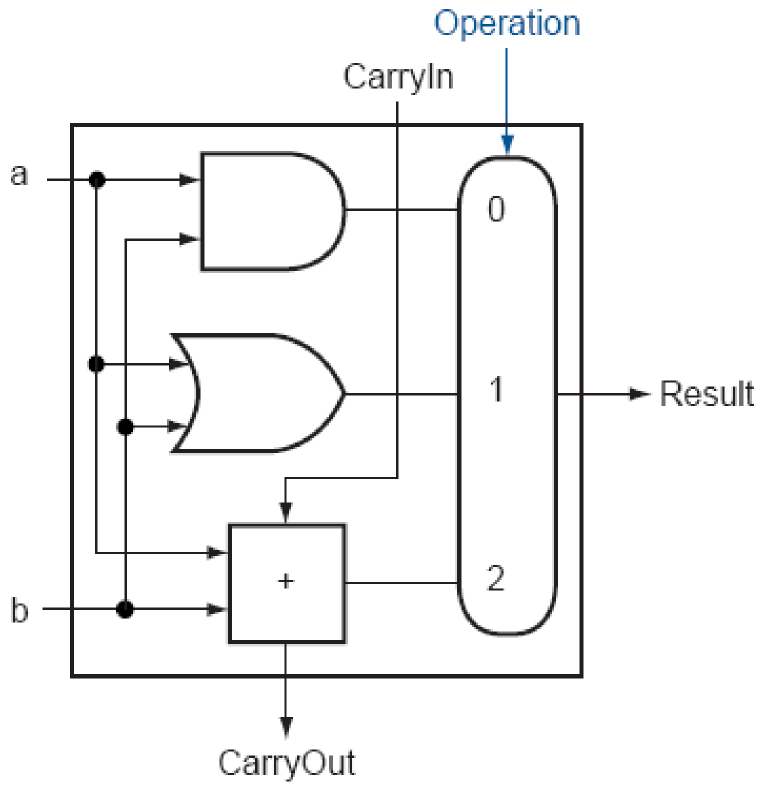


FIGURE B.5.6 A 1-bit ALU that performs AND, OR, and addition (see Figure B.5.5).

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

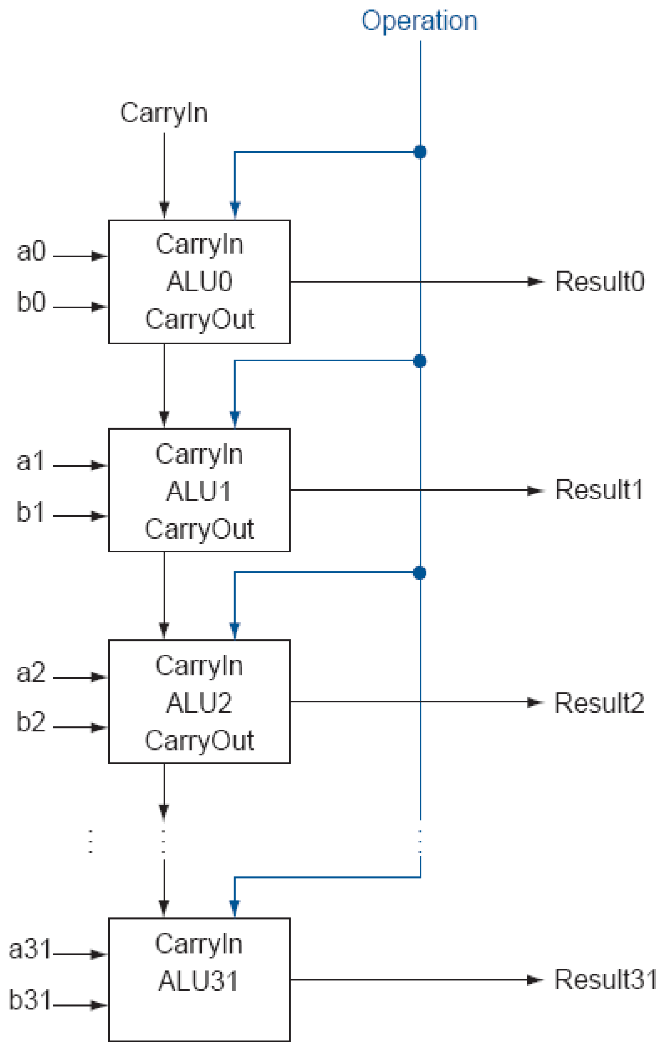


FIGURE B.5.7 A 32-bit ALU constructed from 32 1-bit ALUs. CarryOut of the less significant bit is connected to the CarryIn of the more significant bit. This organization is called ripple carry.

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

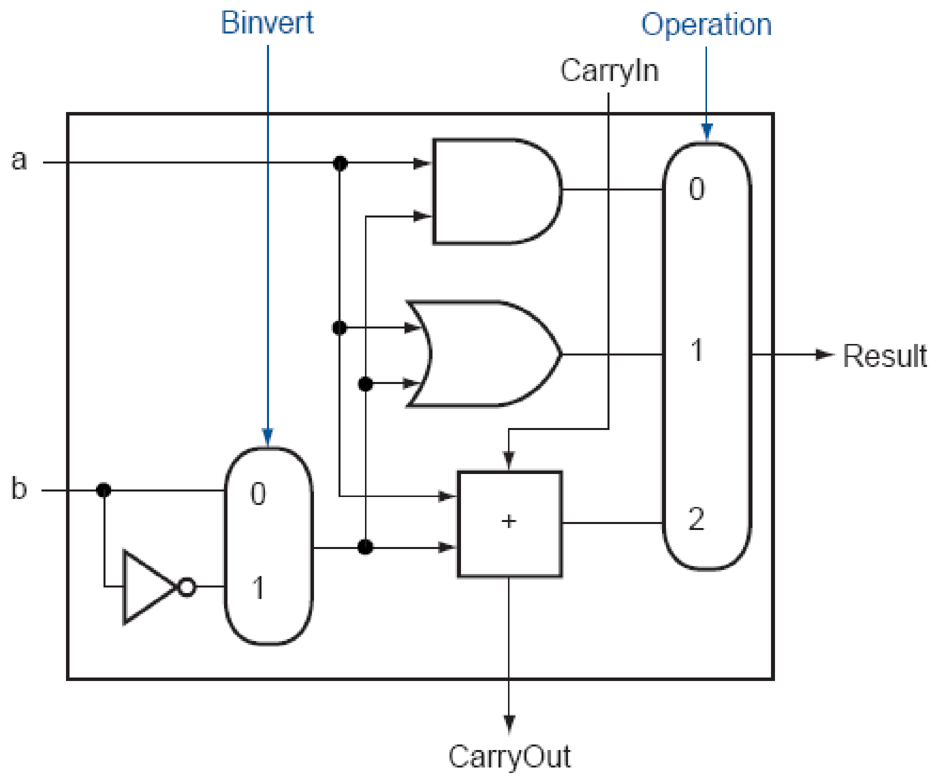


FIGURE B.5.8 A 1-bit ALU that performs AND, OR, and addition on a and b or a and $\overline{\text{b}}$. By selecting $\overline{\text{b}}$ (Binvert = 1) and setting CarryIn to 1 in the least significant bit of the ALU, we get two's complement subtraction of b from a instead of addition of b to a.

Incorporating NOR

Incorporating NOR

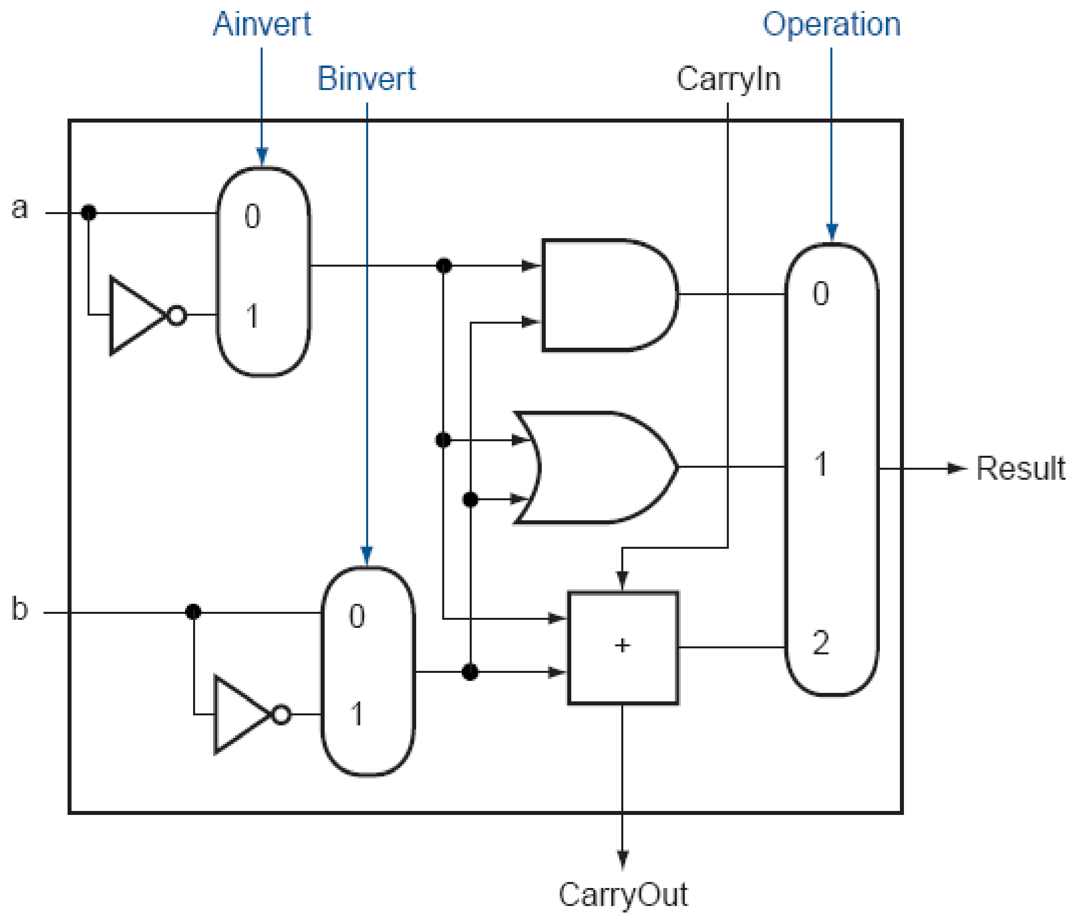


FIGURE B.5.9 A 1-bit ALU that performs AND, OR, and addition on a and b or $\overline{a}$ and $\overline{b}$. By selecting $\overline{a}$ (Ainvert = 1) and $\overline{b}$ (Binvert = 1), we get a NOR b instead of a AND b.

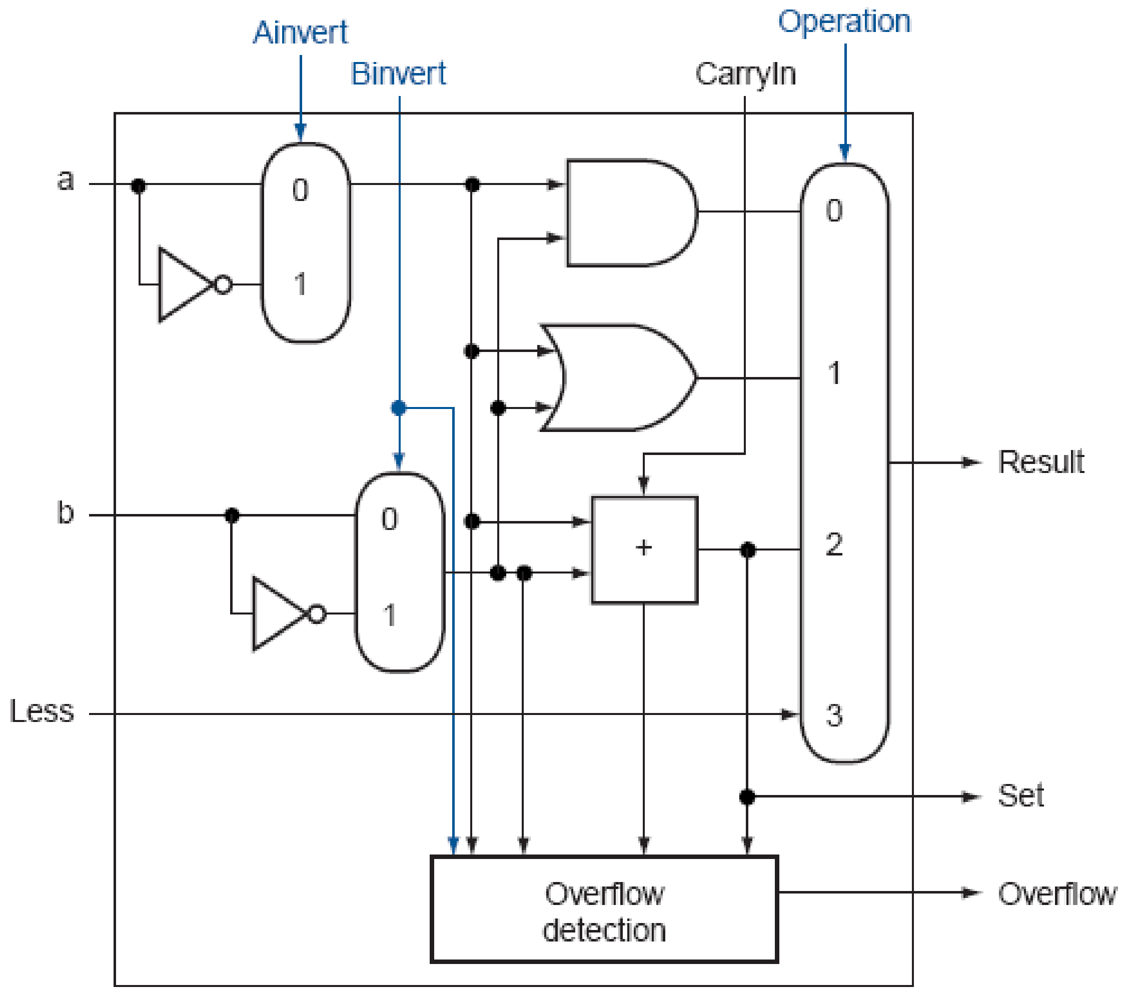
Incorporating slt

Incorporating slt

- Perform a – b and check the sign

- New signal (Less) that is zero for ALU boxes 1-31

- The 31st box has a unit to detect overflow and sign – the sign bit serves as the Less signal for the 0th box



Incorporating beq

- Perform a – b and confirm that the result is all zero’s

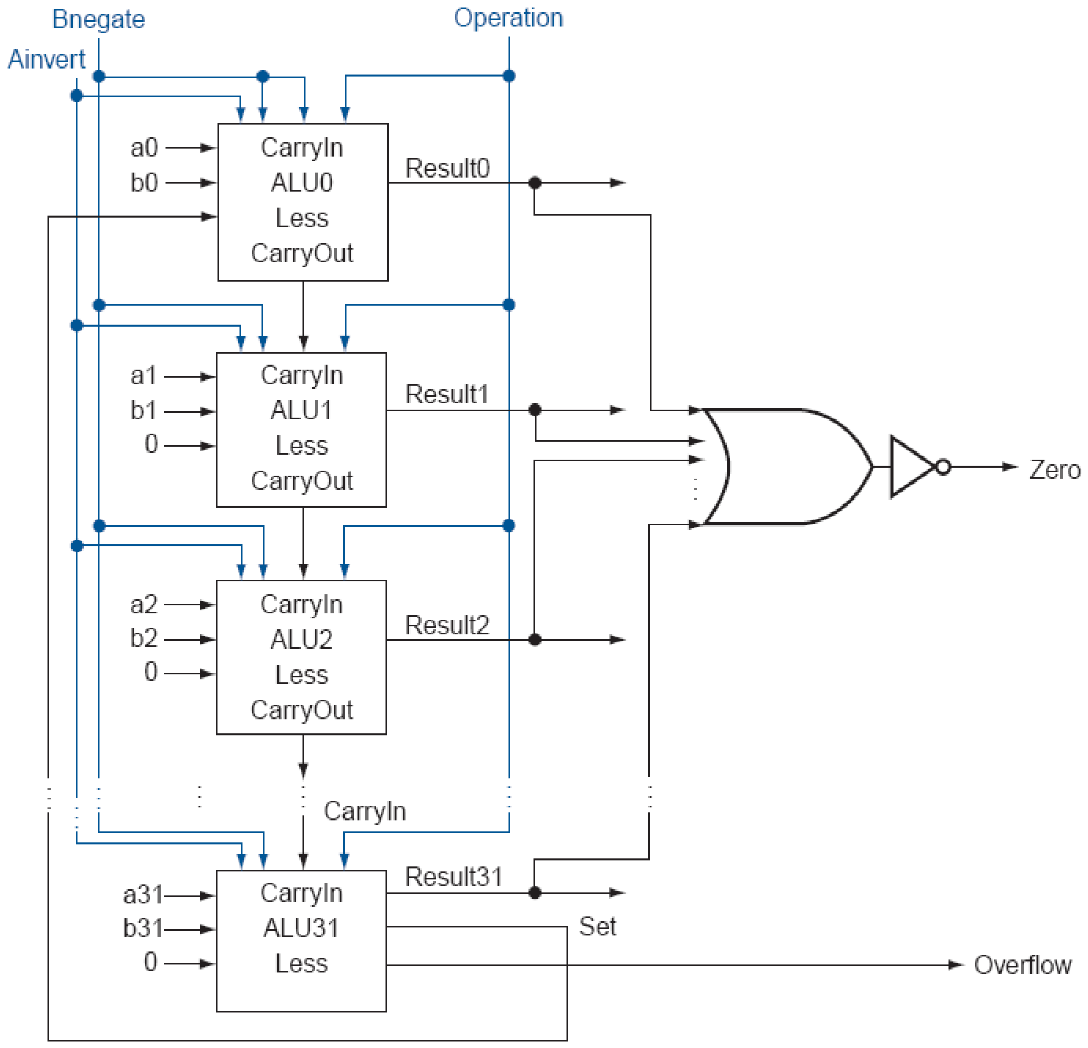


FIGURE B.5.12 The final 32-bit ALU. This adds a Zero detector to Figure B.5.11.

Control Lines

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

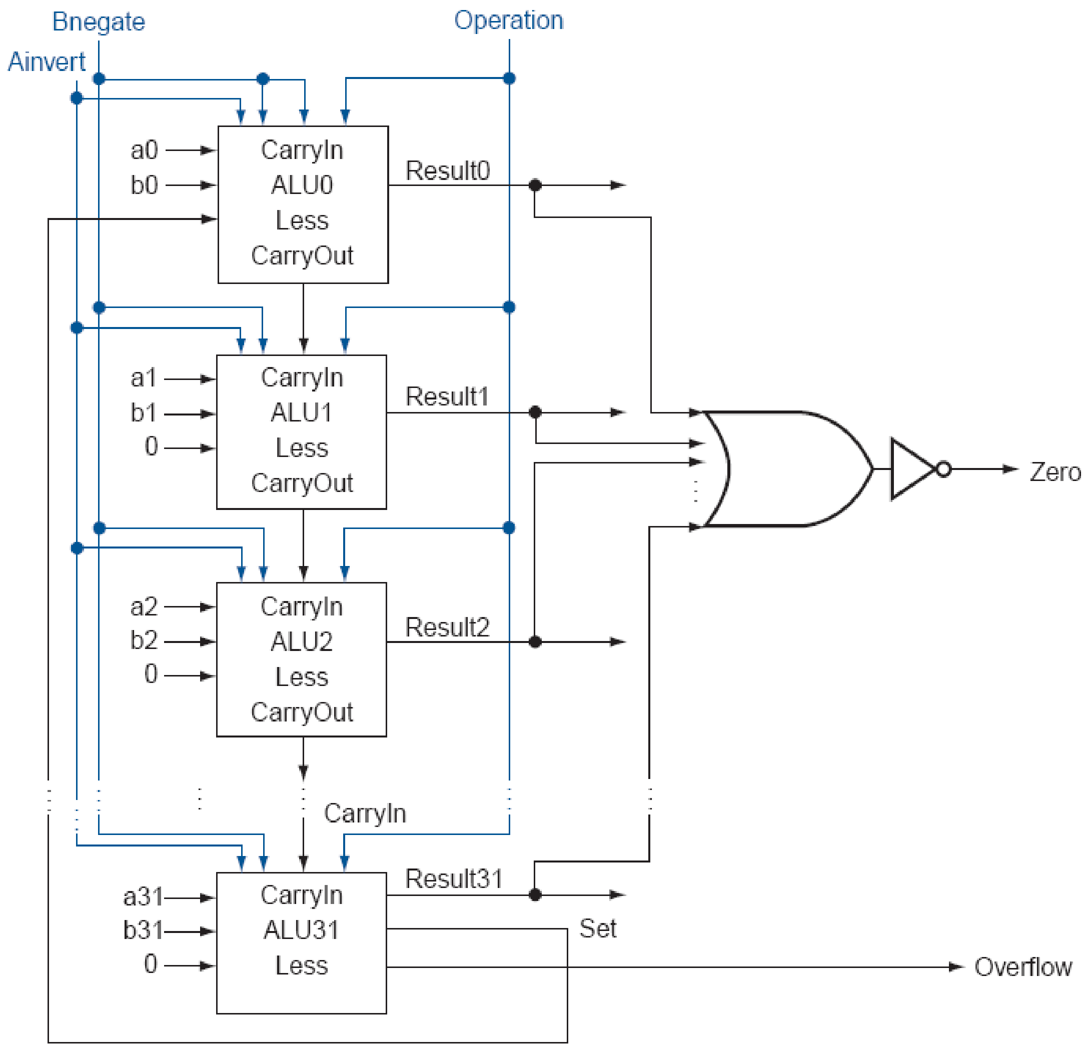
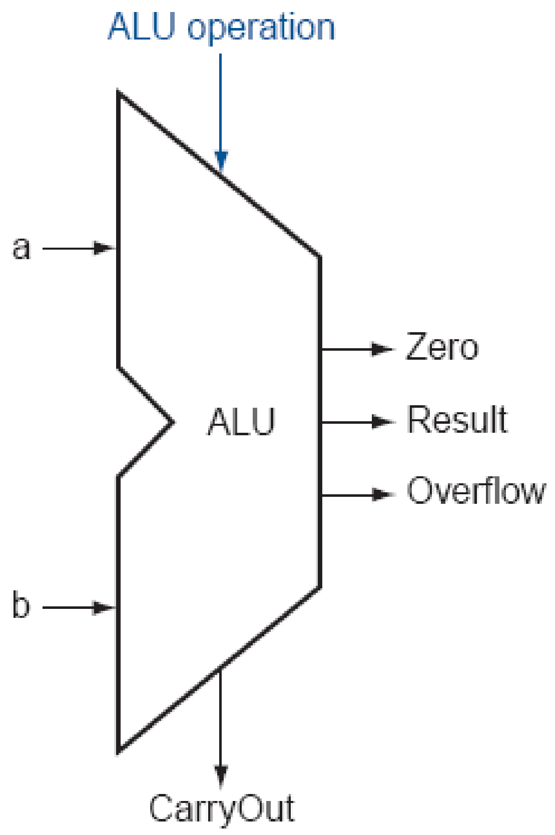


FIGURE B.5.12 The final 32-bit ALU. This adds a Zero detector to Figure B.5.11.

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

CarryIn1 = b0.CarryIn0 + a0.CarryIn0 + a0.b0
CarryIn2 = b1.CarryIn1 + a1.CarryIn1 + a1.b1
= b1.b0.c0 + b1.a0.c0 + b1.a0.b0 +
a1.b0.c0 + a1.a0.c0 + a1.a0.b0 + a1.b1
…
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:

$C_{i+1} = a_i.b_i + a_i.C_i + b_i.C_i$

$= (a_i.b_i) + (a_i + b_i).C_i$

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

$c1 = g0 + p0.c0$

$c2 = g1 + p1.c1$

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

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

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

Either,

a carry was just generated, or

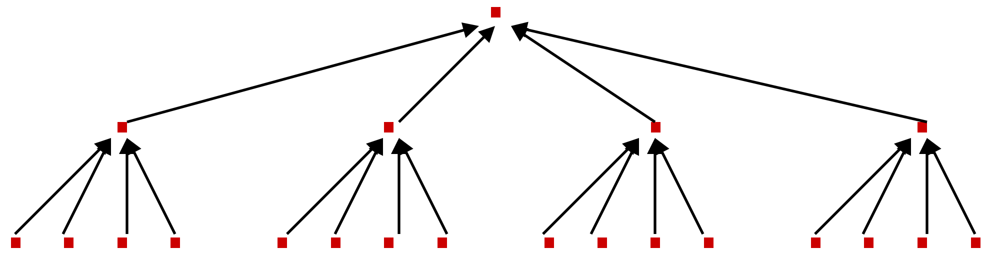
a carry was generated in the last step and was propagated, or

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 32nd 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
	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?

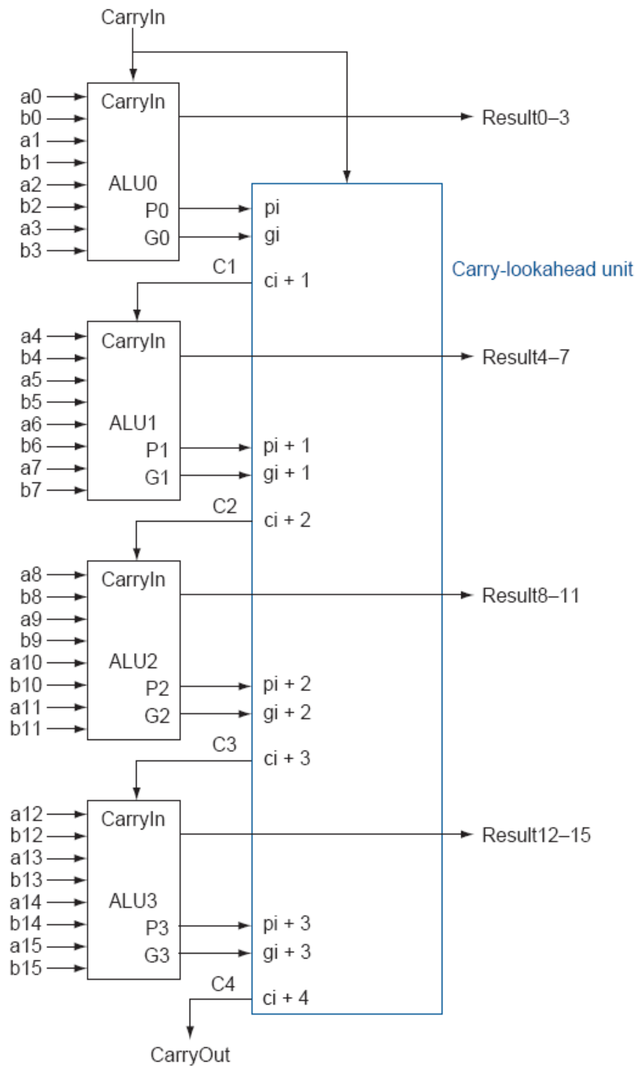


FIGURE B.6.3 Four 4-bit ALUs using carry lookahead to form a 16-bit adder. Note that the carries come from the carry-lookahead unit, not from the 4-bit ALUs.

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

CarryIn1 = b0.CarryIn0 + a0.CarryIn0 + a0.b0
CarryIn2 = b1.CarryIn1 + a1.CarryIn1 + a1.b1
= b1.b0.c0 + b1.a0.c0 + b1.a0.b0 +
a1.b0.c0 + a1.a0.c0 + a1.a0.b0 + a1.b1
…
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:

$c_{i+1} = a_i.b_i + a_i.c_i + b_i.c_i$

$= (a_i.b_i) + (a_i + b_i).c_i$

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
	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?

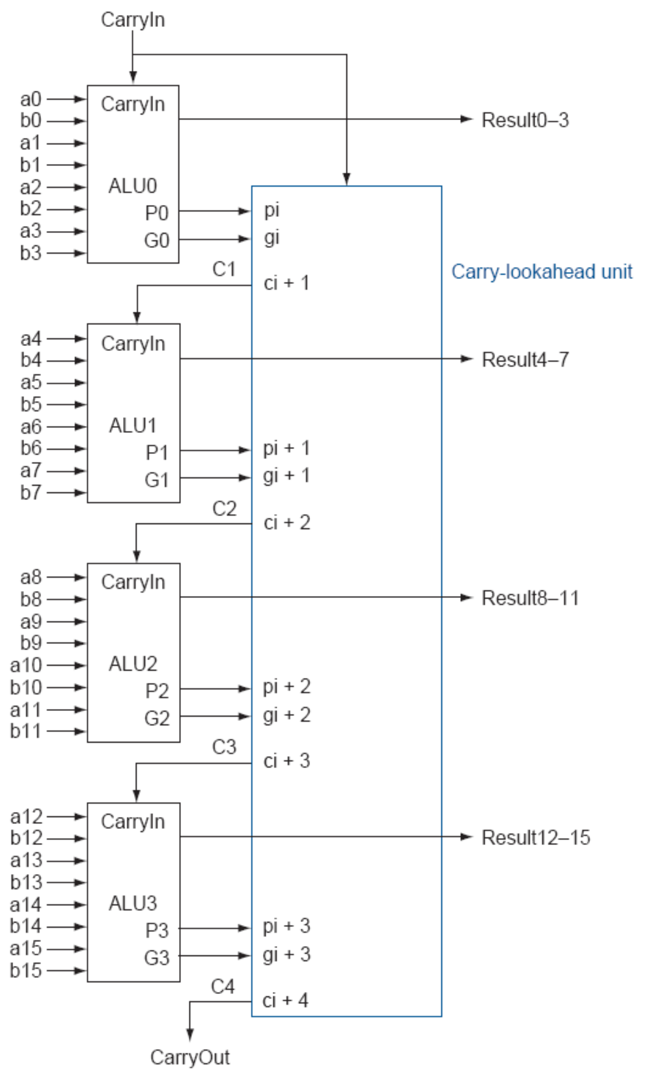


FIGURE B.6.3 Four 4-bit ALUs using carry lookahead to form a 16-bit adder. Note that the carries come from the carry-lookahead unit, not from the 4-bit ALUs.