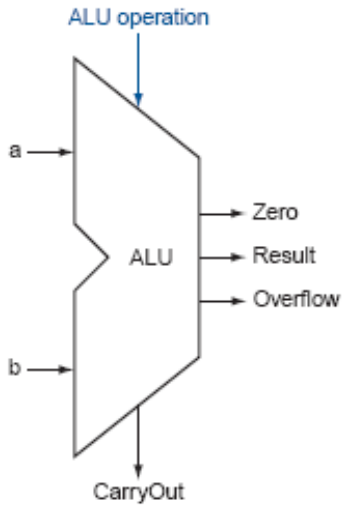
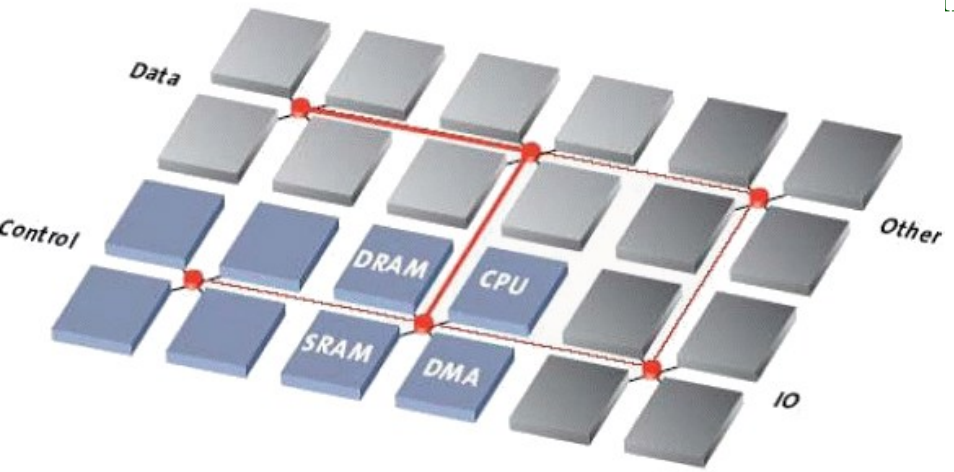
Chapter 3

Arithmetic for Computers





Kuei-Chung Chang

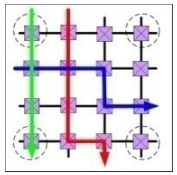
Email: changkc@fcu.edu.tw

Ext. 3753, Office 211

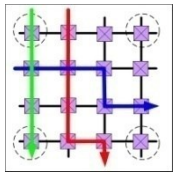
nation Engineering and Computer Science

Feng Chia Univ.

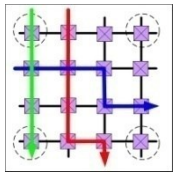
Spring 2019



Outline

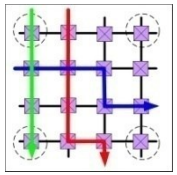


Outline



Introduction

- Computer words are composed of bits; thus, words can be represented as binary numbers.
  - Decimal or Binary?
  - ***Fractions and real numbers?***
  - What happens if an operation creates a number bigger than can be represented [***overflow/underflow***]?
  - How does hardware really ***multiply*** or divide numbers?



Addition & Subtraction

- Just like in grade school (carry/borrow 1s)

```
  0111        0111        0110
+ 0110      - 0110      - 0101
```

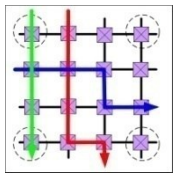
- Two's complement operations
  - ***subtraction using addition of negative numbers***

```
  0111          0111
+ 1010    ⇐   - 0110
```

- ***Overflow*** (result too large for finite computer word):
  - e.g., adding two n-bit numbers does not yield an n-bit number

```
  0111
+ 0001
------
 1000
```

note that overflow term is somewhat misleading, it does not mean a carry "overflowed"

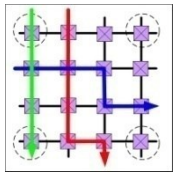


Detecting Overflow

- No overflow when adding a positive and a negative number
- No overflow when signs are the same for subtraction
- ***Overflow occurs when the value affects the sign:***
  - overflow when adding two positives yields a negative
  - or, adding two negatives gives a positive
  - or, subtract a negative from a positive and get a negative
  - or, subtract a positive from a negative and get a positive
- Detecting Overflow

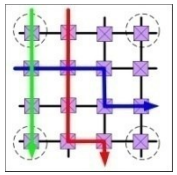
	A	B	Result
A+B	+	+	-
A+B	-	-	+
A-B	+	-	-
A-B	-	+	+

Yield overflow



Effects of Overflow

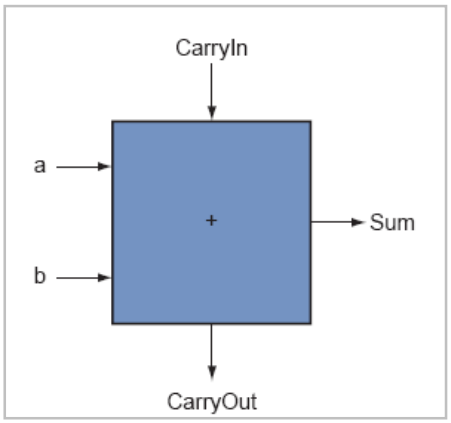
- *An exception (interrupt) occurs*
  - Control jumps to predefined address for exception
  - Interrupted address is saved for possible resumption
- Don't always want to detect overflow (unsigned)
  - — ARM instructions:
    - `BVS(branch if overflow set)`
    - `BVC(branch if overflow clear)`
    - ***`Saturation`*** `arithmetic instructions`
      - » *It means that when a calculation overflows, the result is set to the largest positive number or most negative number, rather than a modulo calculation.*

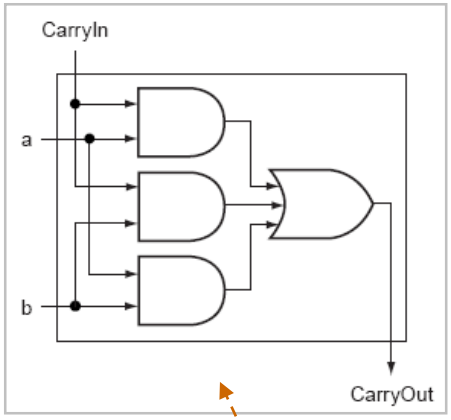


ALU (Arithmetic Logic Unit) – Appendix C

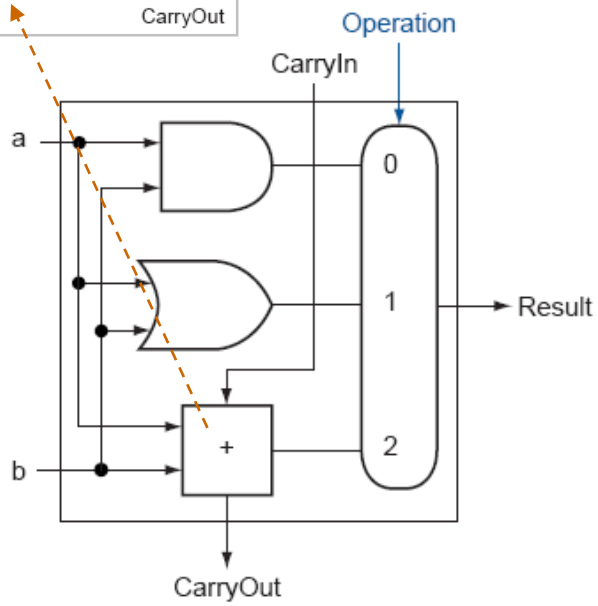
- 1-bit Adder

$$c_{out} = a\ b + a\ c_{in} + b\ c_{in}$$
$$sum = a \text{ xor } b \text{ xor } c_{in}$$



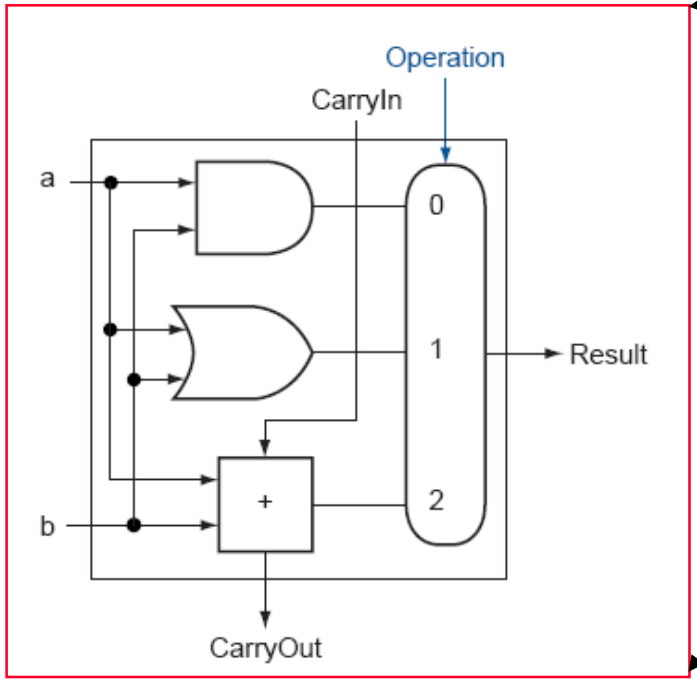


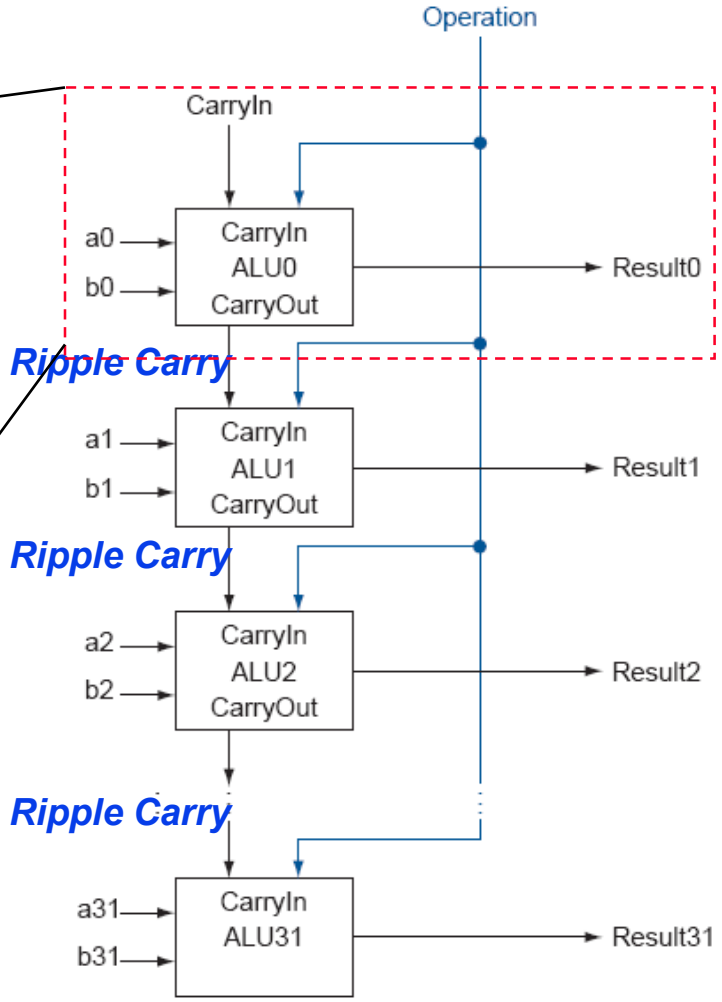
- 1-bit **ALU** that *performs AND, OR, +*



32-bit ALU

1-bit ALU that performs **AND, OR, +**



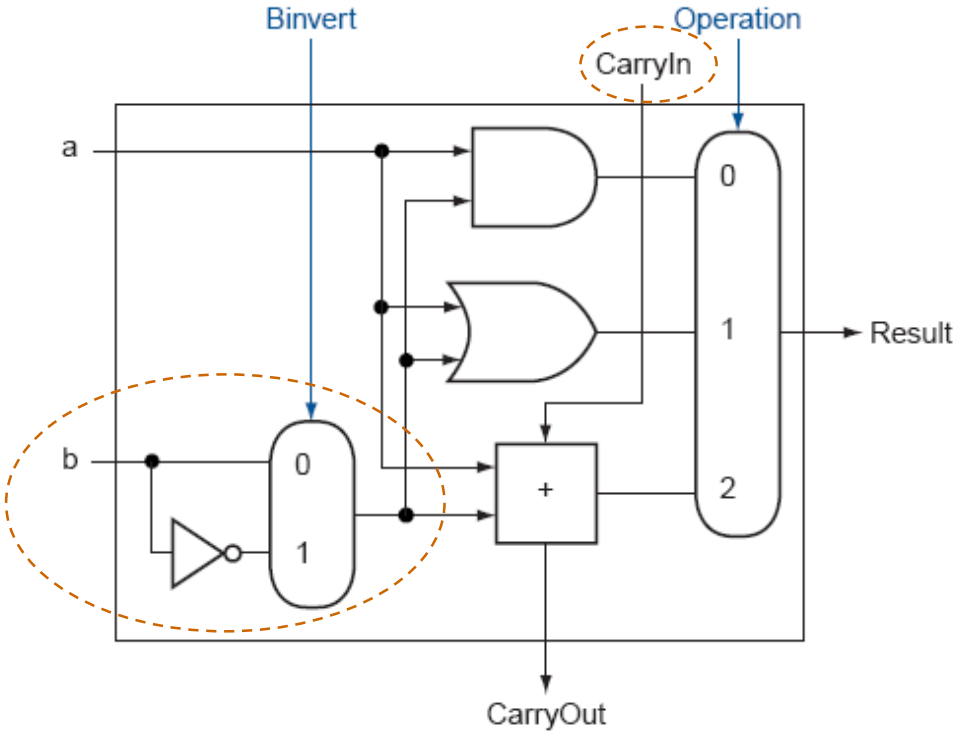


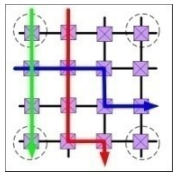
Subtraction - Supporting 2's complement

1-bit ALU that performs AND, OR, +, "-"

Supporting 2's complement (a+(**-b**))

1. *-b = b'* *(0→1, 1→0)*
2. *CarryIn = 1* *(+1)*





Outline

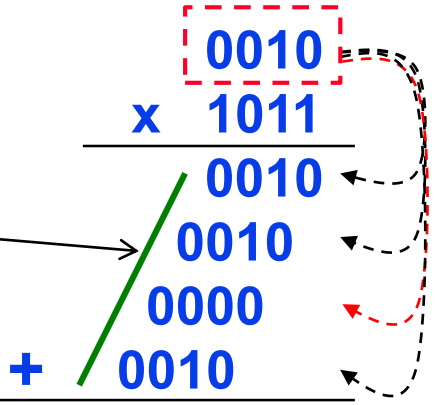
Multiplication

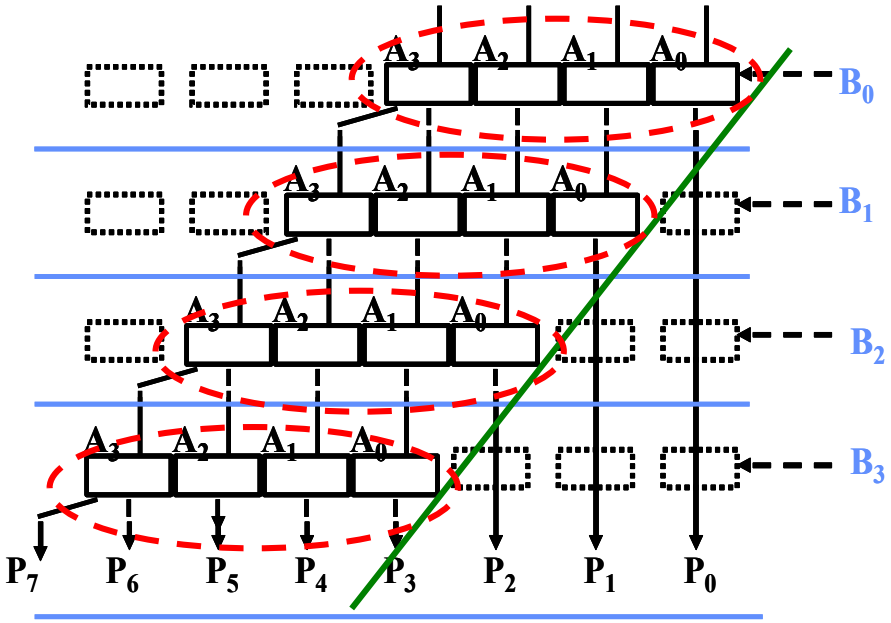
- More complicated than addition
  - accomplished via **shifting** and **addition**
- *More time and more H.W. area*
- Let's look at 2 versions based on a gradeschool algorithm

```
    0010   (multiplicand)
  x 1011   (multiplier)
```

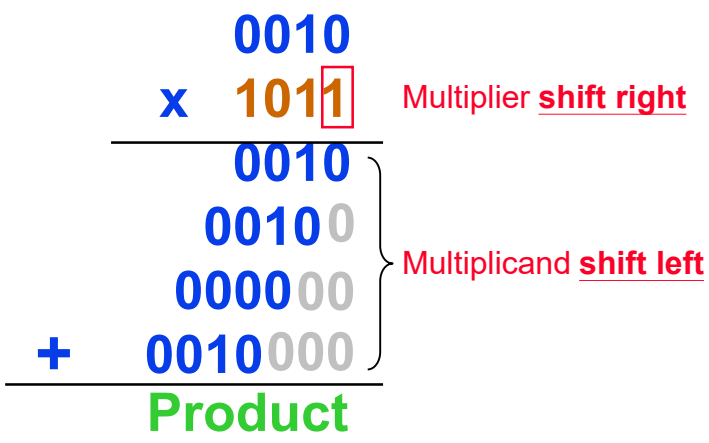
- Negative numbers:

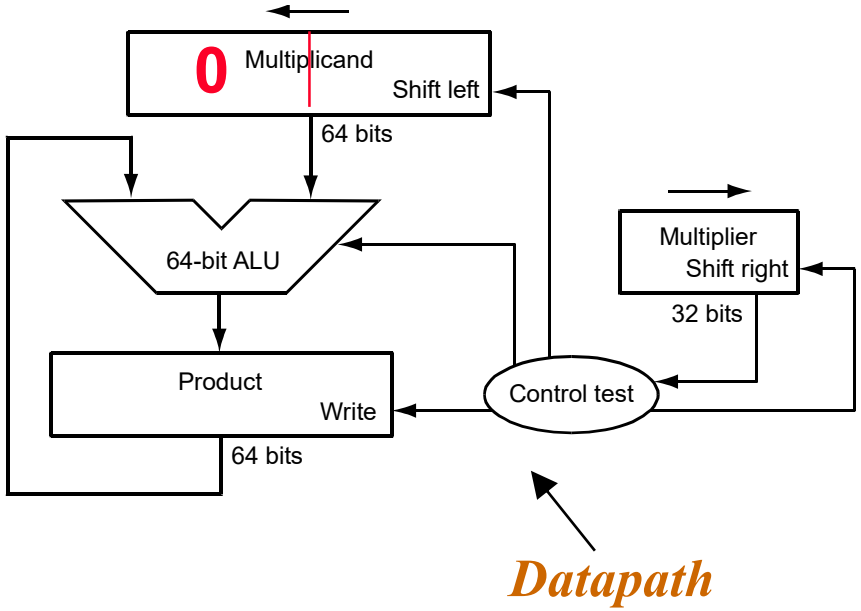
  convert and multiply

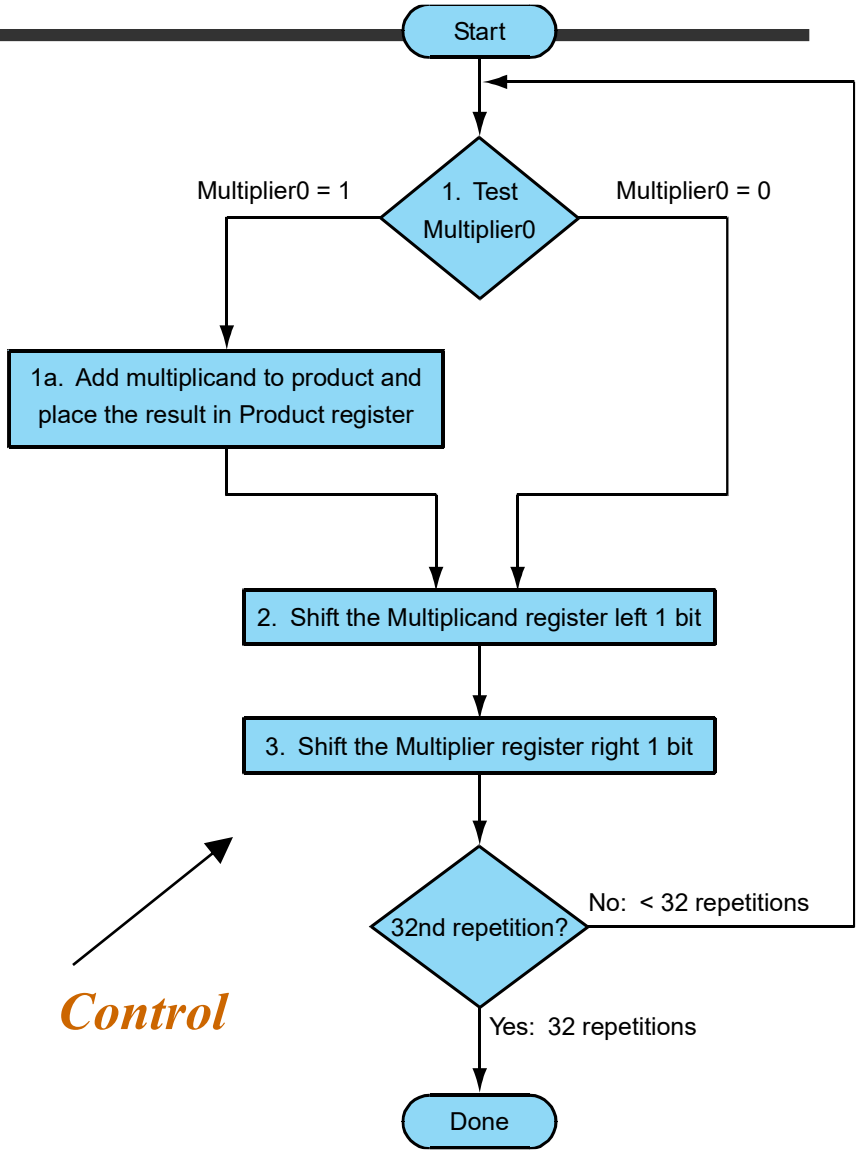




Multiplication: 1st sequential version

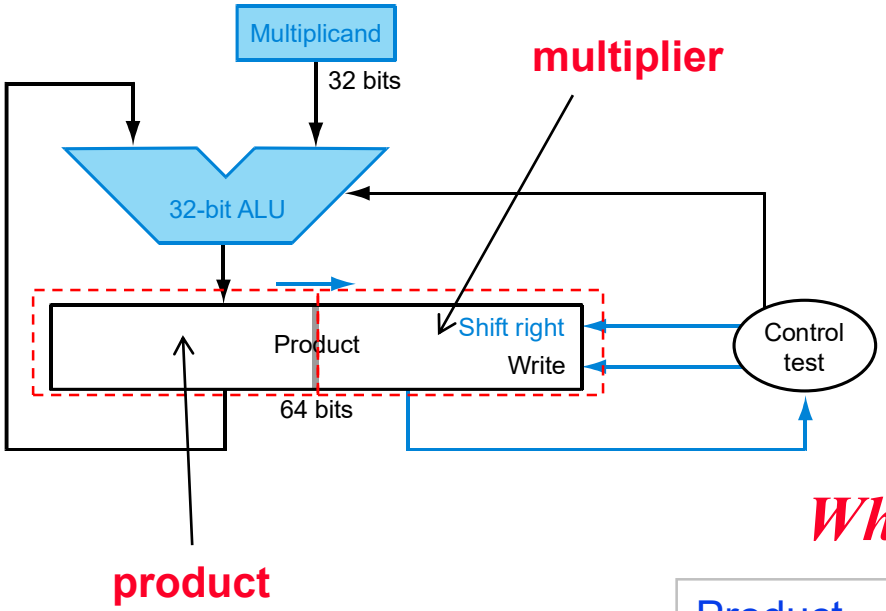


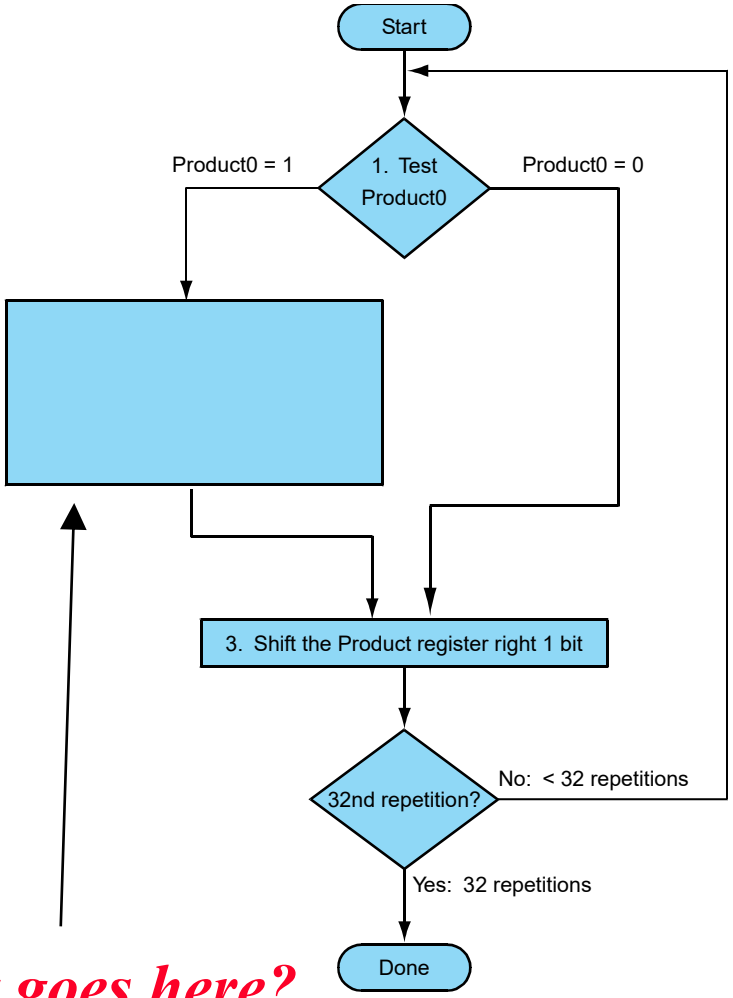




Refined Version – parallel operations

- Combine right part of the ***product*** with ***multiplier***
- ***Multiplier*** starts in right half of ***product***

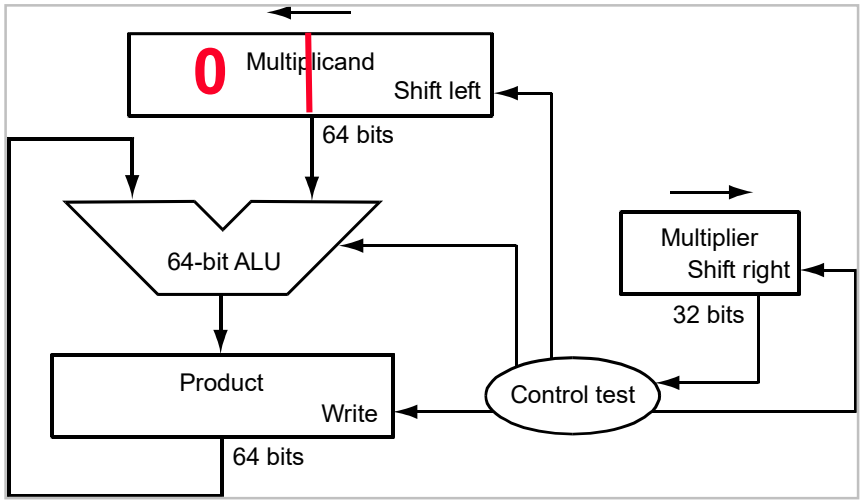


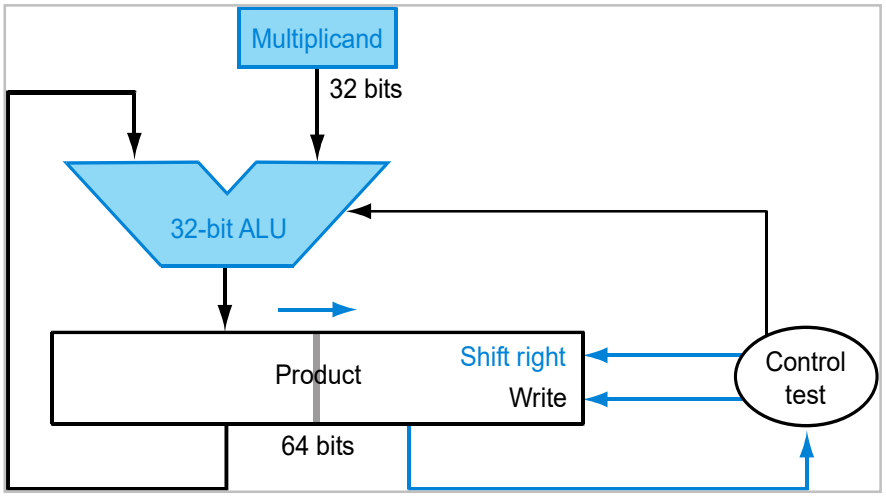


What goes here?

$Product_{left} = Product_{left} + Multiplicand$

Why can be improved?

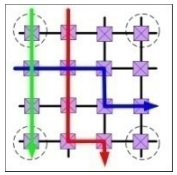




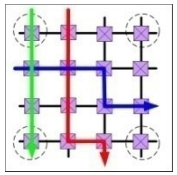
improved

- Performing the operations in parallel:

  the multiplier and multiplicand (both are in product register) are shifted while the multiplicand is added to the product if the multiplier bit is a 1



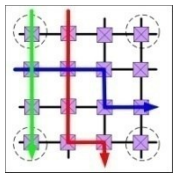
Outline



Divide: idea?

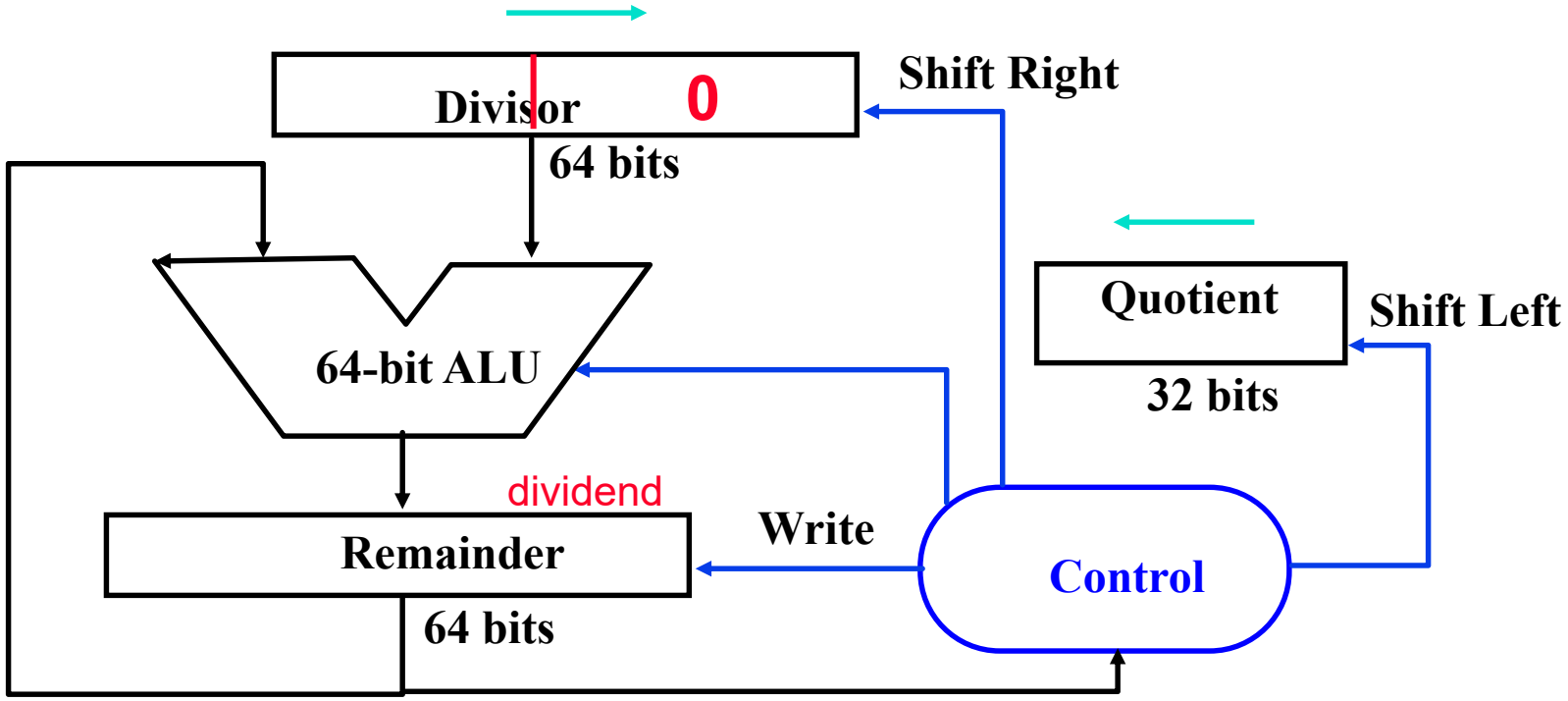
```
                1001        Quotient
           ------------
Divisor 1000 | 1001010      Dividend
             -1000
             ------
                 10
                101
               1010
              -1000
              -----
                 10         Remainder (or Modulo result)
```

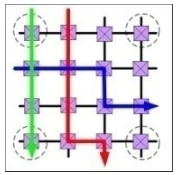
- See how big a number can be subtracted, creating quotient bit on each step
  - Binary => 1 * divisor or 0 * divisor
- Dividend = Quotient x Divisor + Remainder
  => | Dividend | = | Quotient | + | Divisor |
- => | Quotient | = | Dividend | - | Divisor |
- 3 versions of divide, successive refinement



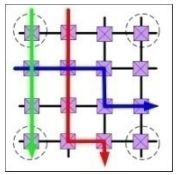
DIVIDE HARDWARE Version 1

- 64-bit Divisor reg, 64-bit ALU, 64-bit Remainder reg, 32-bit Quotient reg



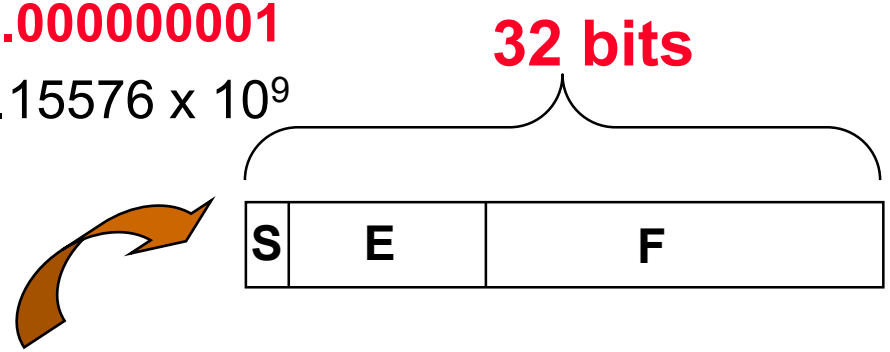


Outline



Floating Point (a brief look)

- We need a way to represent
  - numbers with **fractions**, e.g., 3**.1416**
  - very **small** numbers, e.g., **0.000000001**
  - very **large** numbers, e.g., 3.15576 x 10^9
- ***Representation***:
  - sign, exponent, fraction:



3.1416 ⇒ $(-1)^{sign} \times (1+fraction) \times 2^{exponent}$

  - more bits for fraction gives more *accuracy*
  - more bits for exponent increases *range*

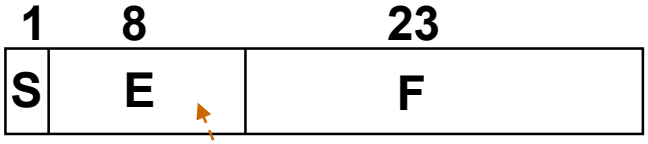
Trade-off

- IEEE 754 floating point standard:
  - Single precision: 8 bits exponent, 23 bits fraction
  - Double precision: 11 bits exponent, 52 bits fraction

IEEE 754 floating-point standard

- Leading "1" bit of ***significand*** is implicit

  $(-1)^{sign} \times (1 + fraction) \times 2^{exponent}$



- Exponent is "***biased***" to make ***sorting easier***
  - exponent all 0s is smallest, exponent all 1s is largest
  - bias of **127** for single precision and **1023** for double precision
  - summary: $(-1)^{sign} \times (1+fraction) \times 2^{exponent - bias}$

Exp_new = Exp_old + 127

Why ?

Numbers with bigger exponents look larger than numbers with smaller exponents.

If we use 2's complement for negative exponents, a negative exponent will look like a big number.

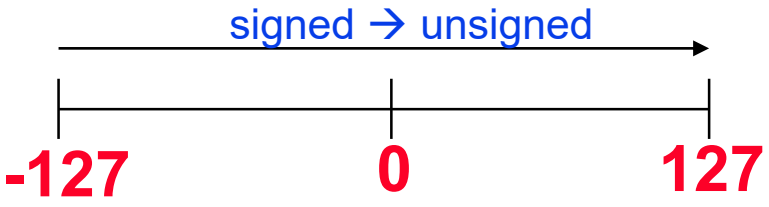
$1.0_2 \times 2^2$
Exp = 00000010

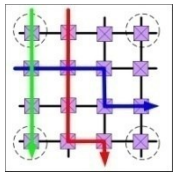
$1.0_2 \times 2^1$
Exp = 00000001

$1.0_2 \times 2^{-1}$
Exp = 11111111

How ?

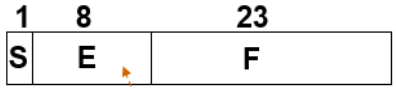
$(-1)^S \times (1+Fraction) \times 2^{(Exp - Bias)}$





IEEE 754 floating-point standard

- ***Example*** for biased notation – (p. 189)
  - decimal: -0.75 = - (½ + ¼) = **$-3/4_{10}$**
  - binary: $= -0.11_2 = -1.1 \times 2^{-1}$ (Normalized)
  - $(-1)^{sign} \times (1 + fraction) \times 2^{exponent}$ $= -1.1 \times 2^{(126 - 127)}$
  - floating point: *exponent = 126 = 01111110*
  - IEEE single precision:



    - 10111111000000000000000000000

Exp+127

$1.0_2 \times 2^1$
Exp = 00000001

$1.0_2 \times 2^{-1}$
Exp = 11111111

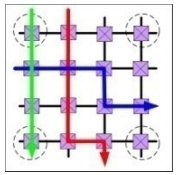
After Bias

$1.0_2 \times 2^1$
Exp = 10000000

$1.0_2 \times 2^{-1}$
Exp = 01111110

Exp= 128

Exp= 126



IEEE 754 floating-point standard

- Example for biased notation – (p. 190)
  - IEEE single precision:
    - 1**10000001**01000000000000000000000

8 bit
(注意多一個0)

10-base Number = $(-1)^{1}$ x $(1+0.\mathbf{01}_{2})$ x $2^{(\mathbf{129}-\mathbf{127})}$

= -1 x 1.25 x 4

= -5.0

Basic Addition (+) Algorithm

Example (p. 191):

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

(assume 4 decimal digitals of the Sig. and 2 decimal digitals of the Exp.)

Step 1 : *match exponents*

$1.61010 \times 10^{-1} = 0.01610_{10} \times 10^{1}$

Step 2 : *add significands*

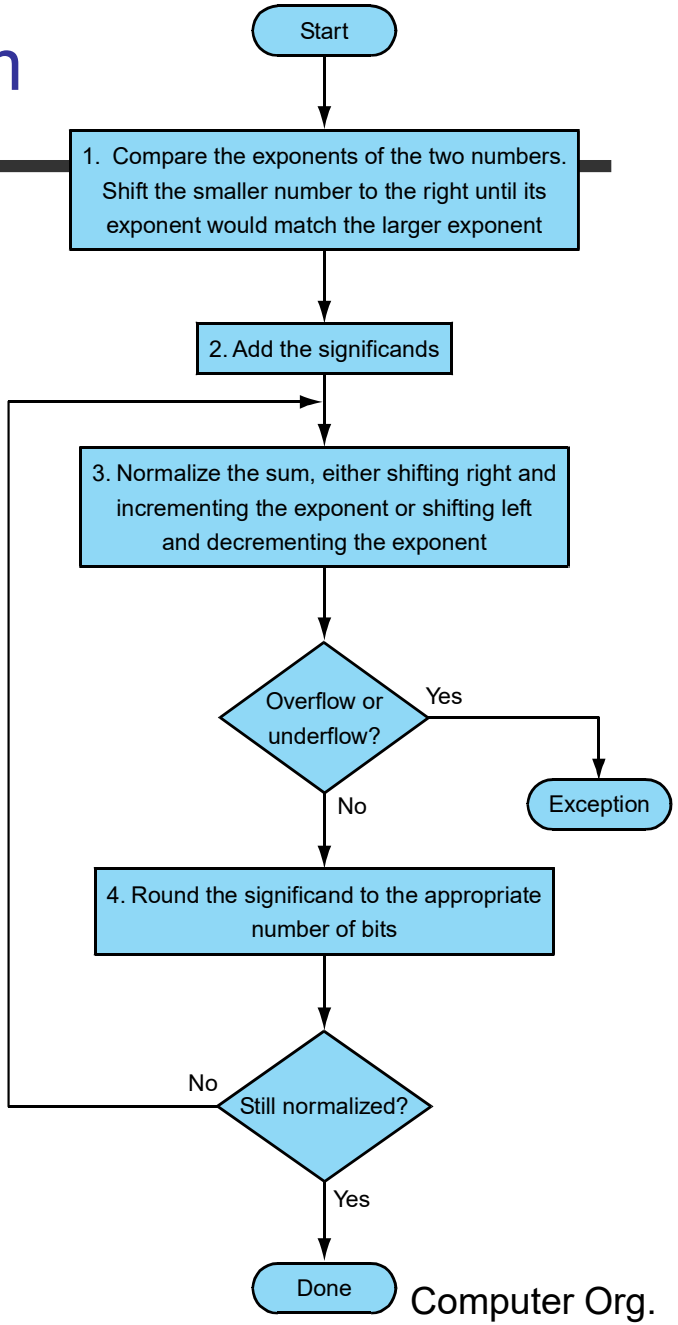
$9.999_{10} + 0.016_{10} = 10.015_{10}$

Step 3 : *normalize*

$10.015_{10} \times 10^{1} = 1.0015_{10} \times 10^{2}$

Step 4 : *round*

$1.0015_{10} \times 10^{2} = 1.002_{10} \times 10^{2}$



Multiplication

Example (p. 194):

$1.110_{10} \times 10^{10} \times 9.200_{10} \times 10^{-5}$

(assume 4 decimal digitals of the Sig. and 2 decimal digitals of the Exp.)

Step 1 : *add exponents*

$10 + (-5) = 5$ (bias : $5 + 127$)

Step 2 : *significands multiplication*

$1.110_{10} \times 9.200_{10} = 10.212_{10} \times 10^5$

Step 3 : *normalize*

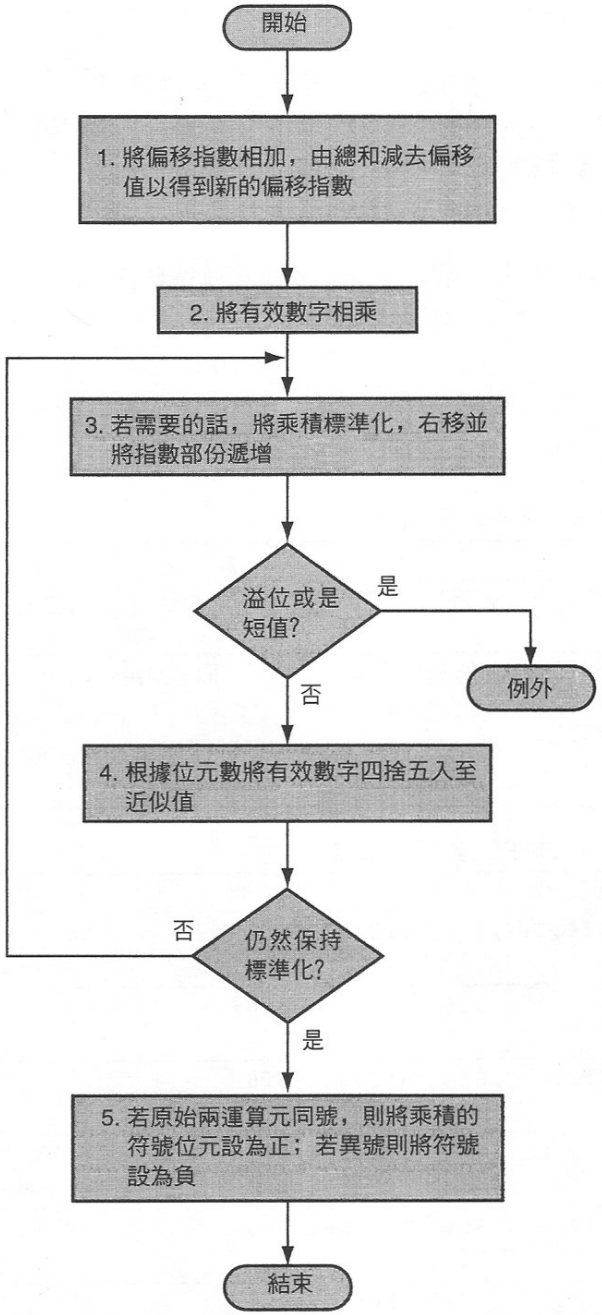
$10.212_{10} \times 10^5 = 1.0212_{10} \times 10^6$

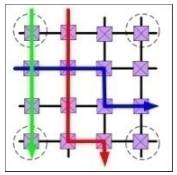
Step 4 : *round*

$1.0212_{10} \times 10^6 = 1.021_{10} \times 10^6$

Step 5 : *decide sign bit*

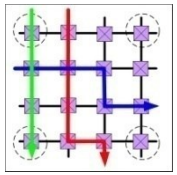
$+1.021_{10} \times 10^6$





Floating Point Complexities

- Operations are somewhat more complicated (see text)
- In addition to overflow we can have "*underflow*"
  - a calculation produces a non-zero result that is less than the smallest non-zero quantity
- *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!



Guard and Round

$2.56_{10} \times 10^0 + 2.34_{10} \times 10^2 = ?$

$2.56_{10} \times 10^0 = 0.0256_{10} \times 10^2$

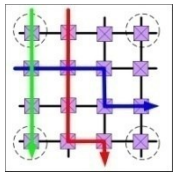
0.0256 → ***Truncated to 0.02 without guard and round digitals***

\+ 2.3400

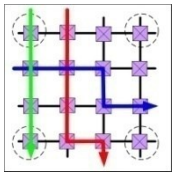
2.3656 56 > 50 ➔ round up

2.37×10^2

2.36×10^2 (without guard and round digits)

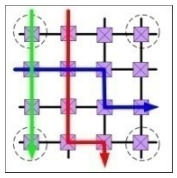


Outline



Parallelism and Computer Arithmetic: Associativity

- Programs have typically been written first to run sequentially before being rewritten to run concurrently.
- If you were to add a million numbers together, you would get the same results whether you used 1 processor or 1000 processors.
- *Floating-point addition is not associativity.*
  - floating-point numbers are approximations of real numbers
  - computer arithmetic has limited precision



Parallelism and Computer Arithmetic: Associativity

- ***Example*** : *(p. 217)*

X + (Y + Z) = (X + Y) + Z

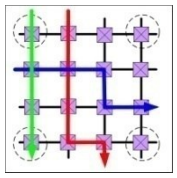
-1.5 x 10^{38}

However, …

		(x+y)+z	x+(y+z)
x	**-1.50E+38**		-1.50E+38
y	**1.50E+38**	0.00E+00	
z	**1.0**	1.0	1.50E+38
		1.00E+00	**0.00E+00**

Y is so much larger than Z that Y + Z is still Y.

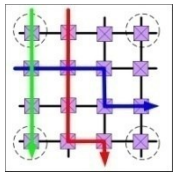
- Parallel programs may interleave operations in unexpected orders
  - Assumptions of associativity may fail
- **Need to validate parallel programs under varying degrees of parallelism**



Java Example 1

```
public class FloatCommutativeLaw {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        float total = 0f;

        // commutative law 應用在小數點加法不一定適用，以下兩段程式碼應該總數一樣~~但是結果卻不一樣
        for (int i = 1 ; i <= 100 ; i++) {
            total += 1f/(float)i;
        }

        System.out.println(total);

        total = 0f;
        for (int i = 100 ; i >= 1 ; i--) {
            total += 1f/(float)i;
        }

        System.out.println(total);

    }

}
```

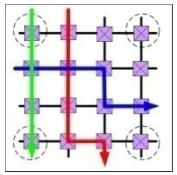
```
5.187378
5.187377
```



Java Example 2

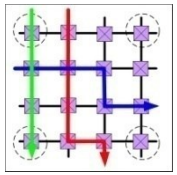
```
Total=2.14748368E8
```

```
float total = 214748368f;

for (long i=0 ; i < 10000000 ; i++)
    total = total + 0.5f;

System.out.println("Total=" + total);
```



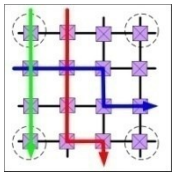
Pentium Bug

- ***Pentium FP Divider*** uses algorithm to generate multiple bits per steps
  - FPU uses most significant bits (MSB) of divisor & dividend/remainder to ***guess*** next 2 bits of quotient
  - *Guess is taken from lookup table*: -2, -1,0,+1,+2 (if a previous guess gets too large a reminder, the partial remainder is adjusted in a subsequent pass)
  - Guess is multiplied by divisor and subtracted from remainder to generate a new remainder
  - Called SRT division after 3 people who came up with idea
- Pentium table uses 7 bits of remainder + 4 bits of divisor = 2^{11} entries
- ***5 entries of divisors omitted***: 1.0001, 1.0100, 1.0111, 1.1010, 1.1101 from PLA (*fix is just add 5 entries back into PLA****: cost $200,000***)
- Since indexed also by divisor/remainder bits, *sometimes bug doesn't show even with dangerous divisor value*



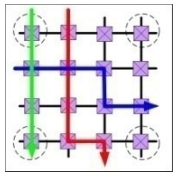
Pentium bug appearance

- First 11 bits to right of decimal point always correct: bits 12 to 52 where bug can occur (*4th to 15th decimal digits*)
- FP divisors near integers 3, 9, 15, 21, 27 are dangerous ones:
  - $3.0 > d \geq 3.0 - 36 \times 2^{-22}$, $9.0 > d \geq 9.0 - 36 \times 2^{-20}$
  - $15.0 > d \geq 15.0 - 36 \times 2^{-20}$, $21.0 > d \geq 21.0 - 36 \times 2^{-19}$
- 0.333333 x 9 could be problem
- In Microsoft Excel, try (4,195,835 / 3,145,727) * 3,145,727
  - = 4,195,835 => not a Pentium with bug
  - = 4,195,**579** => ***Pentium with bug***
    (assuming Excel doesn't already have SW bug patch)
  - Rarely noticed since error in 5th significant digit
  - Success of IEEE standard made discovery possible:
    all computers should get same answer



Pentium Bug Time line

- June 1994: Intel discovers bug in Pentium: takes months to make change, reverify, put into production: plans good chips in January 1995, 4 to 5 million Pentiums produced with bug
- Scientist suspects errors and posts on Internet in September 1994
- Nov. 22 Intel Press release: "Can make errors in 9th digit ... Most engineers and financial analysts need only 4 of 5 digits. Theoretical mathematician should be concerned. ... So far only heard from one."
- **Intel claims** happens once in 27,000 years for typical spread sheet:
  - (1000 divides/day) x (error rate) assuming numbers random
- Dec 12: IBM claims happens once per 24 days
  - 5000 divides/second x 15 minutes = 4,200,000 divides/day
  - Intel said it regards IBM's decision to halt shipments of its Pentium processor-based systems as unwarranted.
- Dec 21 : Intel will exchange the current version for an updated version, for any owner who requests it, free of charge anytime during the life of their computer. This recall cost Intel **$500 million**



Pentium jokes

- **Q**: What's another name for the "Intel Inside" sticker they put on Pentiums?

  A: ***Warning label***.

- **Q**: Have you heard the new name Intel has chosen for the Pentium?

  A: ***the Intel Inacura***.

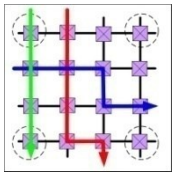
- **Q:** What algorithm did Intel use in the Pentium's floating point divider?

  A: "***Life is like a box of chocolates***." *(Source: F. Gump of Intel)*

Many jokes and new slogans for Pentium ...., and it imply that

How much cheaper would it have been to fix the bug in July 1994?

What was the cost to repair the damage to Intel's reputation?



Pentium conclusion: Dec. 21, 1994 $500M write-off

"To owners of Pentium processor-based computers and the PC community:

We at Intel wish to sincerely apologize for our handling of the recently publicized Pentium processor flaw.

The Intel Inside symbol means that your computer has a microprocessor second to none in quality and performance. Thousands of Intel employees work very hard to ensure that this is true. But no microprocessor is ever perfect.

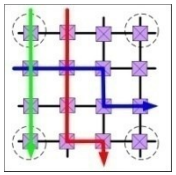
What Intel continues to believe is technically an extremely minor problem has taken on a life of its own. Although Intel firmly stands behind the quality of the current version of the Pentium processor, we recognize that many users have concerns.

We want to resolve these concerns.

Intel will exchange the current version of the Pentium processor for an updated version, in which this floating-point divide flaw is corrected, for any owner who requests it, free of charge anytime during the life of their computer. Just call 1-800-628-8686."

Sincerely,

Andrew S. Grove	Craig R. Barrett	Gordon E. Moore
President /CEO	Executive Vice President &COO	Chairman of the Board



Summary

- Computer arithmetic is constrained **by *limited precision***
- Bit patterns have no inherent meaning but standards do exist
  - two's complement
  - IEEE 754 floating point
- *Computer instructions determine "**meaning**" of the bit patterns*
- Performance and accuracy are important so there are many complexities in real machines
- Algorithm choice is important and may lead to hardware optimizations for both space and time (e.g., multiplication)