ساختار و زبان کامپیوتر

فصل سه

محاسبات کامپیوتری

Computer Structure & Machine Language

Chapter Three

Computer Arithmetic

Copyright Notice

Parts (text & figures of this lecture are adopted from:

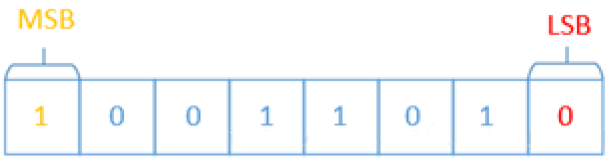
- D. Patterson, J. Henessy, “Computer Organization & Design, The Hardware/Software Interface, MIPS Edition”, 6th Ed., MK Publishing, 2020

Contents

- Weighted Number System
- Signed Number Representation
  - 2’s Compliment/ 1’s Compliment
  - Signed-Magnitude Notation
  - Biased Notation
- Arithmetic Operations
  - Addition/ Subtraction/ Multiplication/ Division
- Real Numbers
  - Fixed Point / Floating Point Representation
  - IEEE 754 Standard

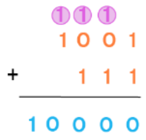
Some Concepts

- LSB and MSB
  - Least Significant Bit (LSB)
  - Most Significant Bit (MSB)



- Signed versus Unsigned
  - Unsigned (Assume all non-negative numbers)
    - Used usually for memory addresses
  - Signed
    - Using sign bit
    - Using two's complement notation

- Carry Out



- Overflow

Number Representation

- Weighted number system
  - Can be represented in any base (radix)
    - Value of i^{th} digit "d_i" = $d_i \times Base^i$
    - $0 \le d_i < Base$

31	30	29	...	i	...	3	2	1	0
d_{31}	d_{30}	d_{29}	...	d_i	...	d_3	d_2	d_1	d_0

Base Examples

Binary

$$(101101)_2 = 1x2^5 + 0x2^4 + 1x2^3 + 1x2^2 + 0x2^1 + 1x2^0$$

Octal

$$(736.4)_8 = 7 \times 8^2 + 3 \times 8^1 + 6 \times 8^0 + 4 \times 8^{-1}$$
$$= 7 \times 64 + 3 \times 8 + 6 \times 1 + 4/8 = (478.5)_{10}$$

Hexadecimal

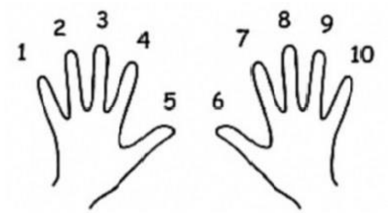
$$(F3)_{16} = F \times 16 + 3 = 15 \times 16 + 3 = (243)_{10}$$

Decimal

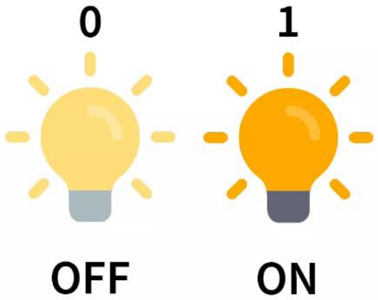
$$(7245)_{10} = 7x10^3 + 2x10^2 + 4x10^1 + 5x10^0$$

Which Base?

- Number Representation
  - Humans prefer base 10, why?



  - Base 2 best works for computers, why?



  - Base 10 inefficient for computers, why?

Base Conversion

- Decimal to Base i Conversion
  - Convert 65_{10} to base 5
  - Convert 19_{10} to base 2
- Binary to Decimal Conversion
  - What is decimal value of this 32-bit number?
    $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1000_{two}$
  - Depends on the notation
    - Signed
    - Unsigned
- Converting between power of 2 radices
  - Convert 11010101000111_2
    - to base 8
    - to base 16

Converting Fractions

- Convert 0.4304_{10} to base 5 = 0.2034

$$\begin{array}{r} .4304 \\ \times \quad 5 \\ \hline 2.1520 \end{array} \qquad \begin{array}{r} .1520 \\ \times \quad 5 \\ \hline 0.7600 \end{array}$$

$$0.7600$$

$$\begin{array}{r} .7600 \\ \times \quad 5 \\ \hline 3.8000 \end{array} \qquad \begin{array}{r} .8000 \\ \times \quad 5 \\ \hline 4.0000 \end{array}$$

- Convert 0.34375_{10} to base 2 = 0.01011

Unsigned Numbers

$$N = (d_{31}*2^{31}) + (d_{30}*2^{30}) + \ldots + (d_1*2^1) + (d_0*2^0)$$

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = 2_{ten}$
... ...
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = 4{,}294{,}967{,}293_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = 4{,}294{,}967{,}294_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 4{,}294{,}967{,}295_{ten}$

Signed Numbers (2's Complement)

$$N = (d_{31}*-2^{31}) + (d_{30}*2^{30}) + \ldots + (d_1*2^1) + (d_0*2^0)$$

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = 2_{ten}$
... ...

$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = 2,147,483,645_{ten}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = 2,147,483,646_{ten}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 2,147,483,647_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -2,147,483,648_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -2,147,483,647_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -2,147,483,646_{ten}$
... ...

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = -3_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -2_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -1_{ten}$

Other Signed Number Notations

- Signed-Magnitude Notation
- Ones’ Complement Notation
- Biased Notation

Signed-Magnitude Notation

- Signed Notation with Sign Flag
- Most positive number
  - 011 … 1
- Most negative number
  - 111 … 1
- There are two zero’s
  - 000 … 0
  - 100 … 0
- Used in floating point representation (Mantissa)

Ones' Complement Notation

- Positive number same as 2's complement
- Negative number:
  - Invert each bit in positive representation
- There are two zero's in ones' complement
  - 000…0
  - 111…1
- Most positive number
  - 0111 … 1
- Most negative number
  - 1000 … 0

Biased Notation (Excess 2^{n-1})

- If n bits used for representation:
  - Add all numbers with 2^{n-1}
- Zero represented by
  - 100 … 0
- Most negative number (-2^{n-1})
  - 000 … 0
- Most positive number ($2^{n-1}-1$)
  - 111 … 1

N	Excess-4	2's Comp
-4	000	100
-3	001	101
-2	010	110
-1	011	111
0	100	000
1	101	001
2	110	010
3	111	011

Biased Notation (Excess $2^{n-1}-1$)

- If n bits used for representation:
  - Add all numbers with $2^{n-1}-1$
- Zero represented by
  - 011 … 1
- Most negative number ($-2^{n-1}+1$)
  - 000 … 0
- Most positive number (2^{n-1})
  - 111 … 1
- Used in floating point representation (exponent)

N	Excess-3	Excess-4
-4	-	000
-3	000	001
-2	001	010
-1	010	011
0	011	100
1	100	101
2	101	110
3	110	011
4	111	-

Signed Number Notations (Summary)

- Unbiased
  - Positive Binary N=+14 0 0001110
  - Signed-Magnitude -N=-14 1 0001110
  - 1s' Complement (2^n-N-1) -N=-14 1 1110001
  - 2's Complement (2^n-N) -N=-14 1 1110010

- Biased (2^{n-1}-1+N)
  - Positive Binary (n=8) N=+14 1 0001101
  - Negative Binary (n=8) -N=-14 0 1110001

Integer Addition/ Subtraction

$$
\begin{array}{r r}
0000000000000000 & \\
0000000001000000 & 64 \\
0000000000101010 & +42 \\
\hline
0000000001101010 & 106
\end{array}
$$

$$
\begin{array}{r r}
1111111111000000 & \\
0000000001000000 & 64 \\
1111111111010110 & -42 \\
\hline
0000000000010110 & 22
\end{array}
$$

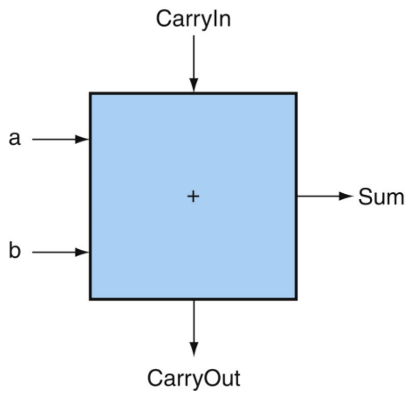
2's complement

Overflow Conditions for Add/Sub

Operation	Operand A	Operand B	Result indicating overflow
$A + B$	≥ 0	≥ 0	< 0
$A + B$	< 0	< 0	≥ 0
$A - B$	≥ 0	< 0	< 0
$A - B$	< 0	≥ 0	≥ 0

- While adding signed numbers, an overflow occurs when
  - Both operands have the same sign,
  - but the result has the opposite sign
- the carry into and out of the MSB differ

One-bit Full Adder



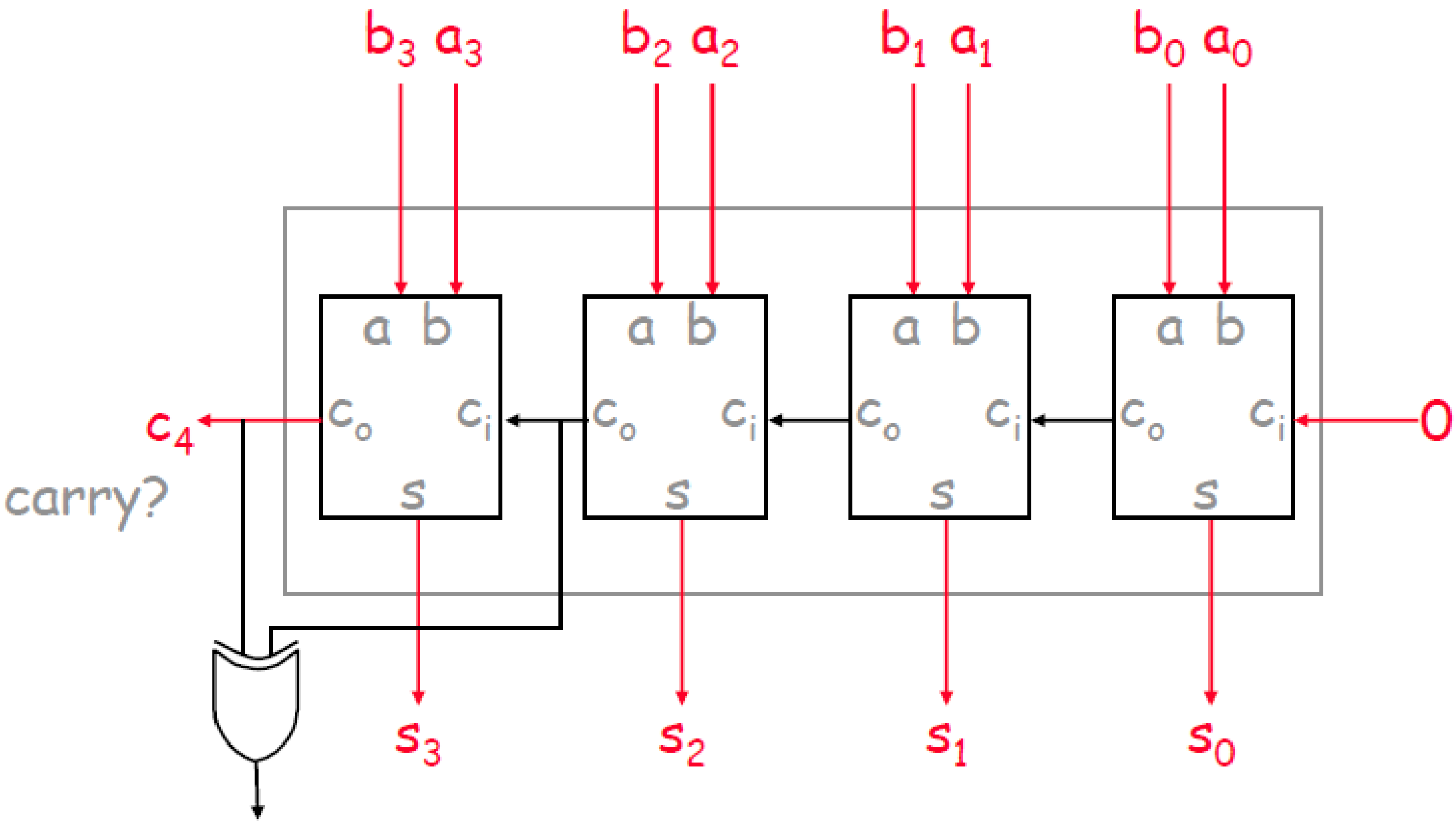
Input and output specification for a 1-bit adder

Inputs			Outputs		Comments
a	b	CarryIn	CarryOut	Sum	
0	0	0	0	0	$0 + 0 + 0 = 00_{two}$
0	0	1	0	1	$0 + 0 + 1 = 01_{two}$
0	1	0	0	1	$0 + 1 + 0 = 01_{two}$
0	1	1	1	0	$0 + 1 + 1 = 10_{two}$
1	0	0	0	1	$1 + 0 + 0 = 01_{two}$
1	0	1	1	0	$1 + 0 + 1 = 10_{two}$
1	1	0	1	0	$1 + 1 + 0 = 10_{two}$
1	1	1	1	1	$1 + 1 + 1 = 11_{two}$

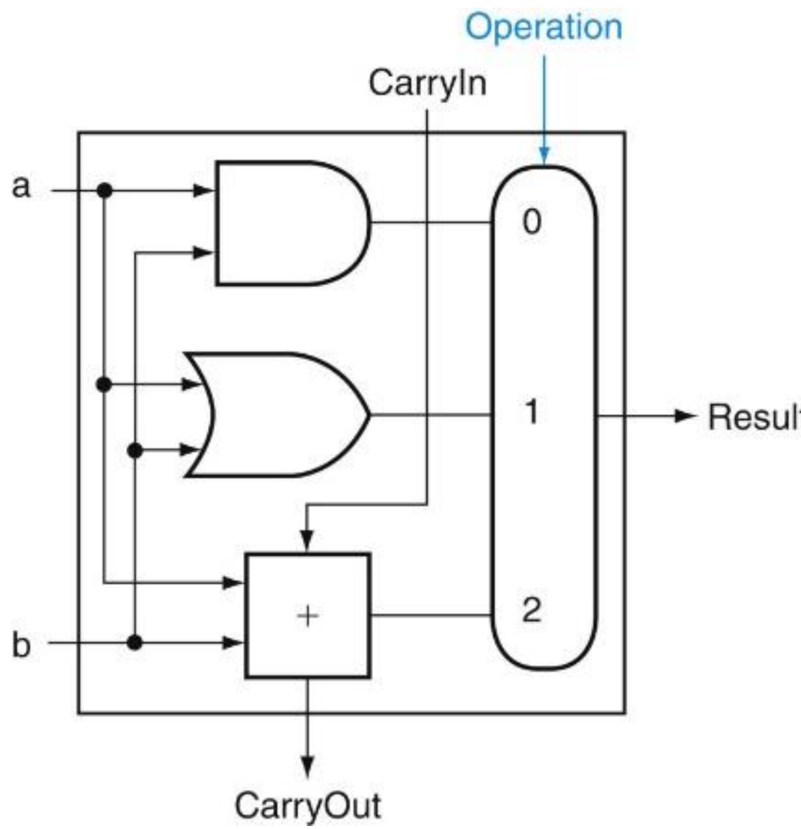
$$Sum = a \oplus b \oplus CarryIn$$

$$CarryOut = a.b + a.CarryIn + b.CarryIn$$

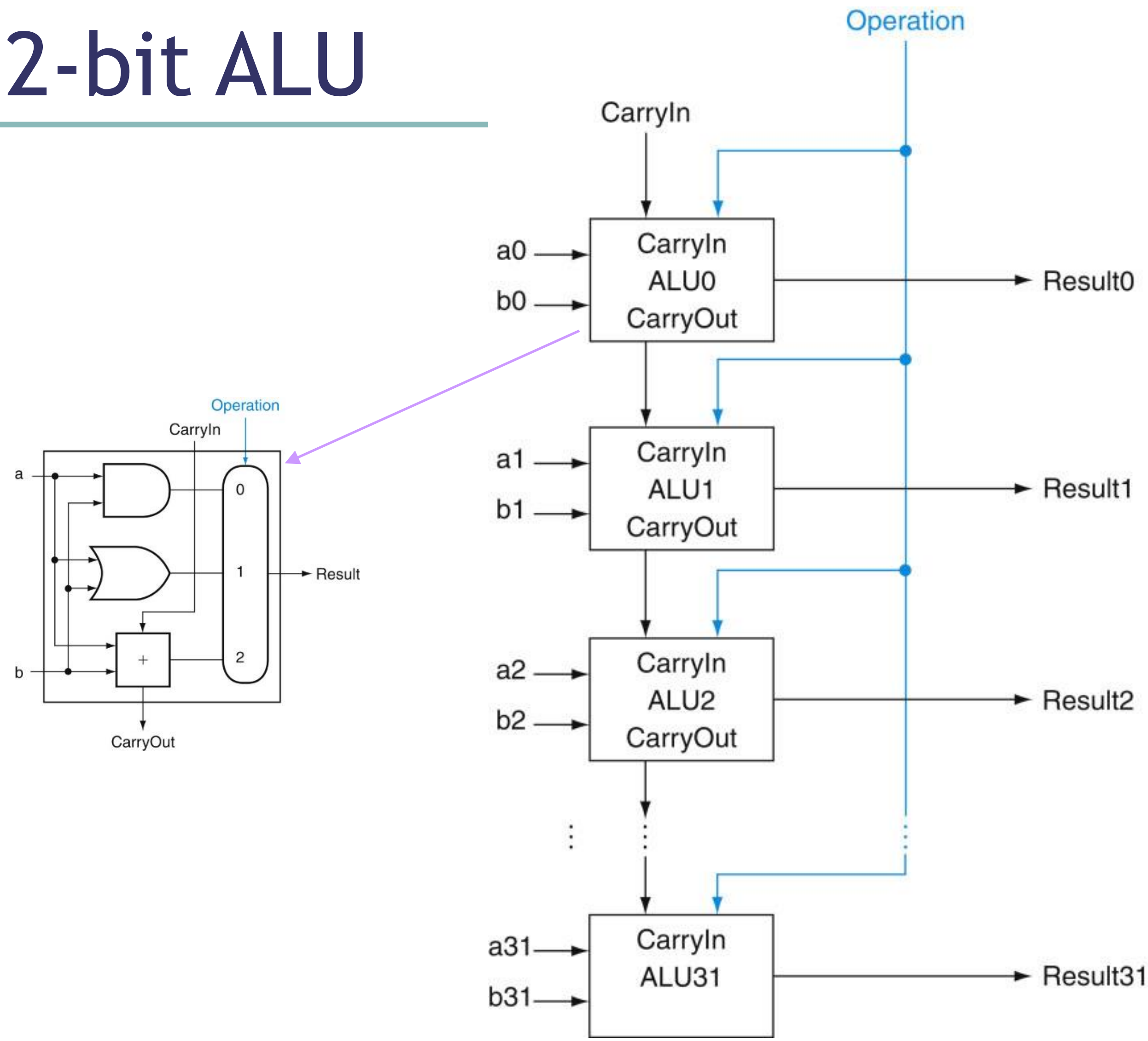
Ripple-Carry Signed Adder



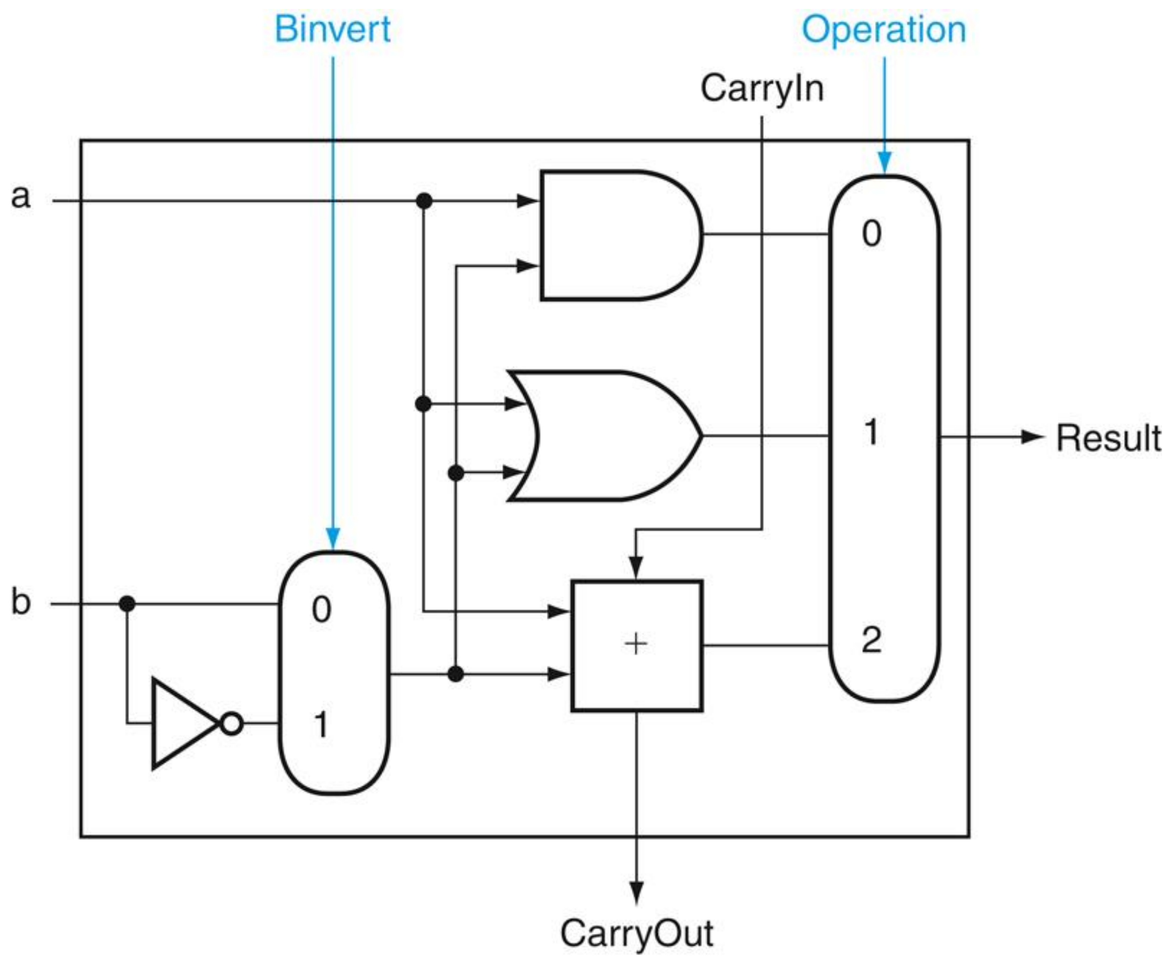
One-bit ALU



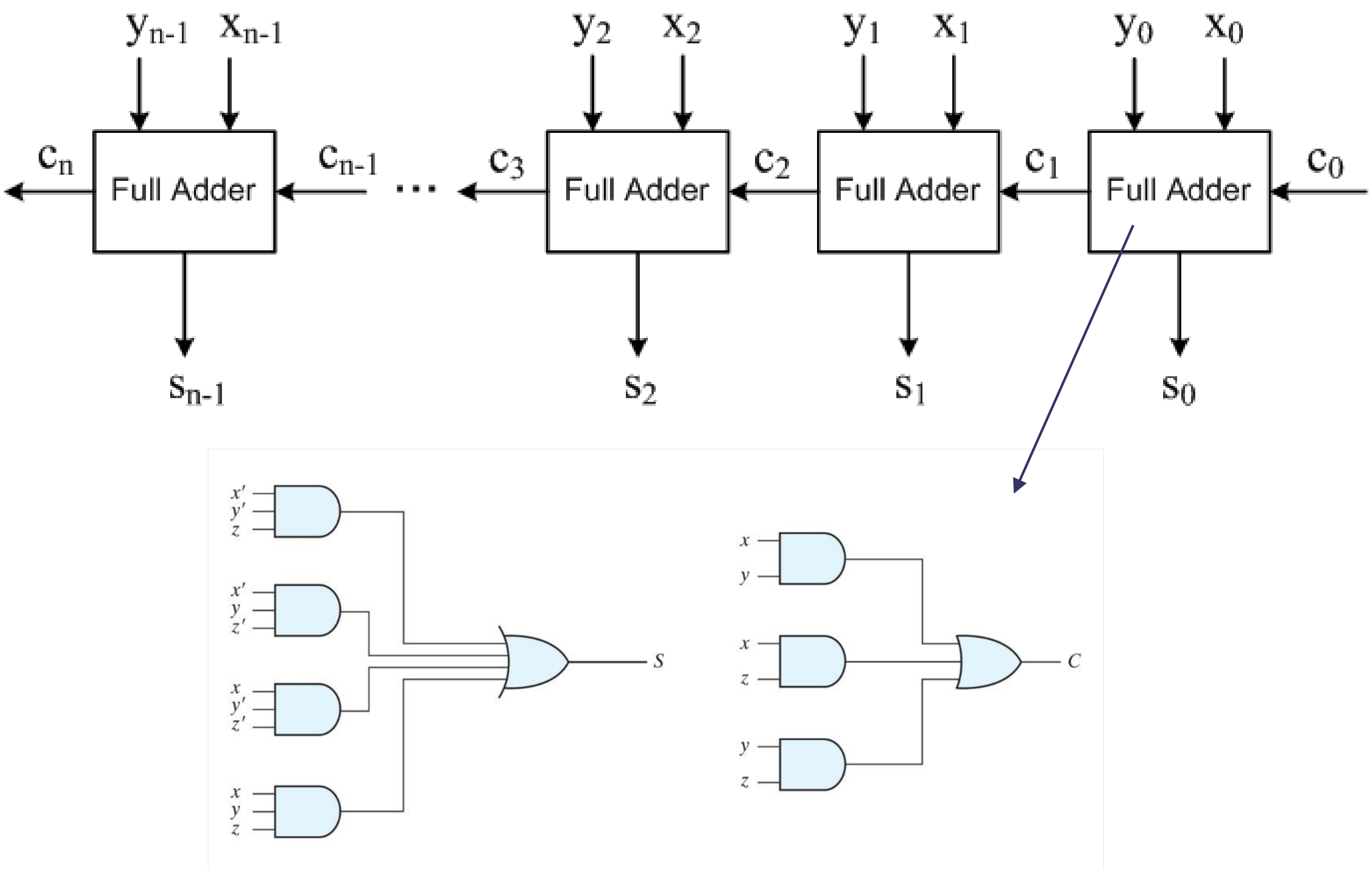
32-bit ALU


Operation
CarryIn
a
b
0
1
2
+
Result
CarryOut
Operation
CarryIn
a0
b0
CarryIn
ALU0
CarryOut
Result0
a1
b1
CarryIn
ALU1
CarryOut
Result1
a2
b2
CarryIn
ALU2
CarryOut
Result2
a31
b31
CarryIn
ALU31
Result31

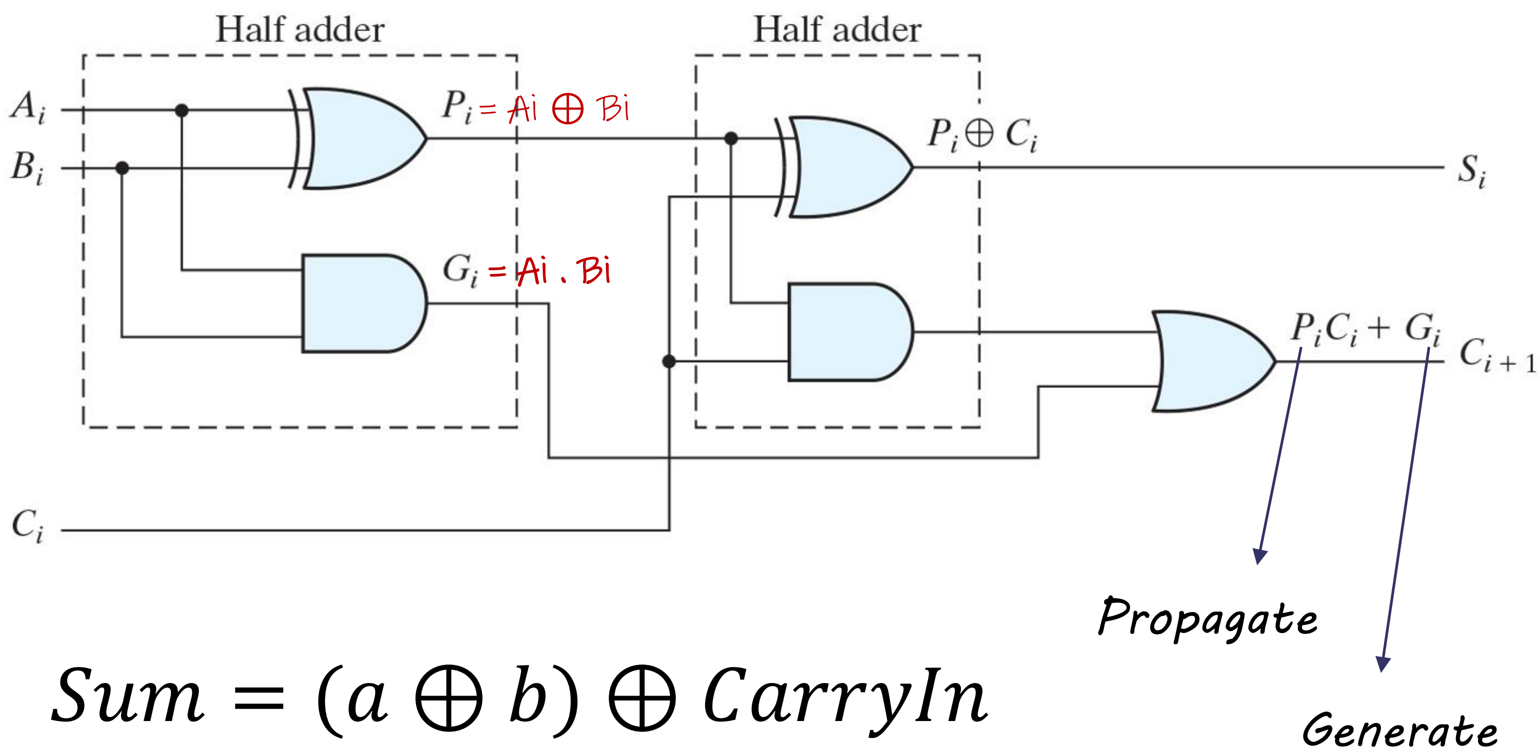
One-bit ALU – more operations



Reminder: Ripple-Carry Adder



Carry Generate / Propagate



$$Sum = (a \oplus b) \oplus CarryIn$$

$$CarryOut = a.b + CarryIn.(a \oplus b)$$

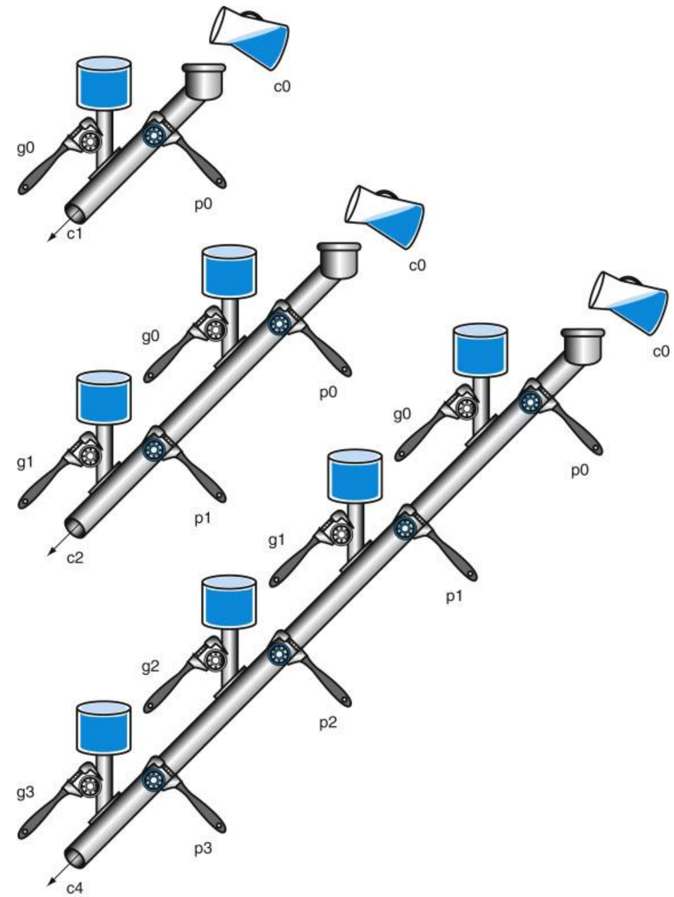
CLA vs. Pipes

$c_1 = g_0 + p_0 . c_0$

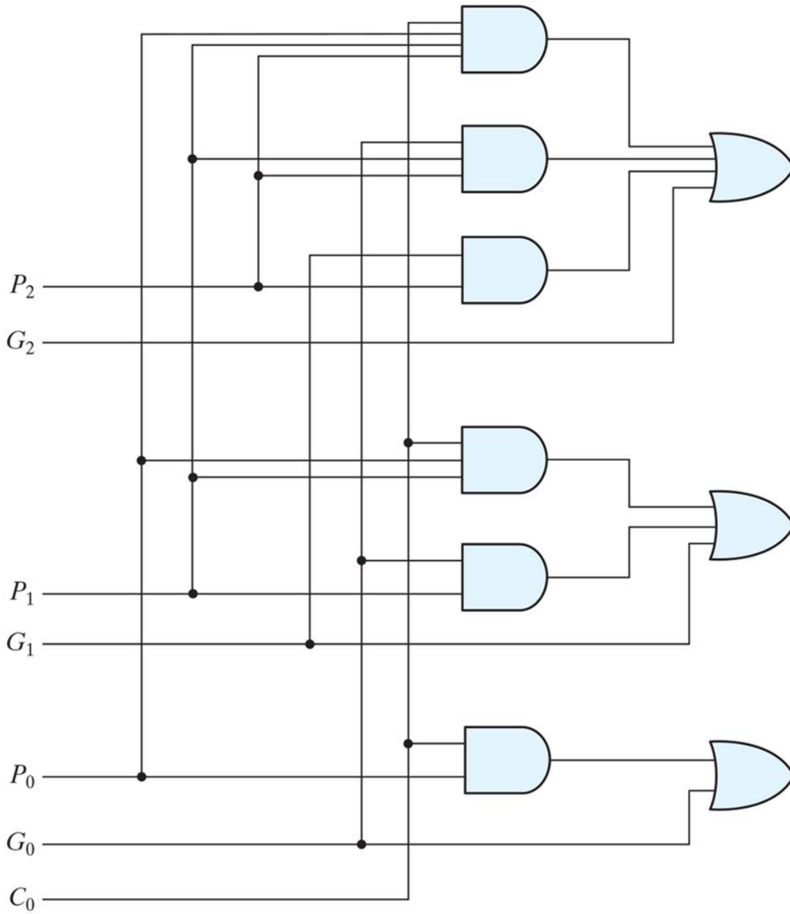
$c_2 = g_1 + p_1 . g_0 + p_1 . p_0 . c_0$

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

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



CLA Generate/ Propagate Circuit

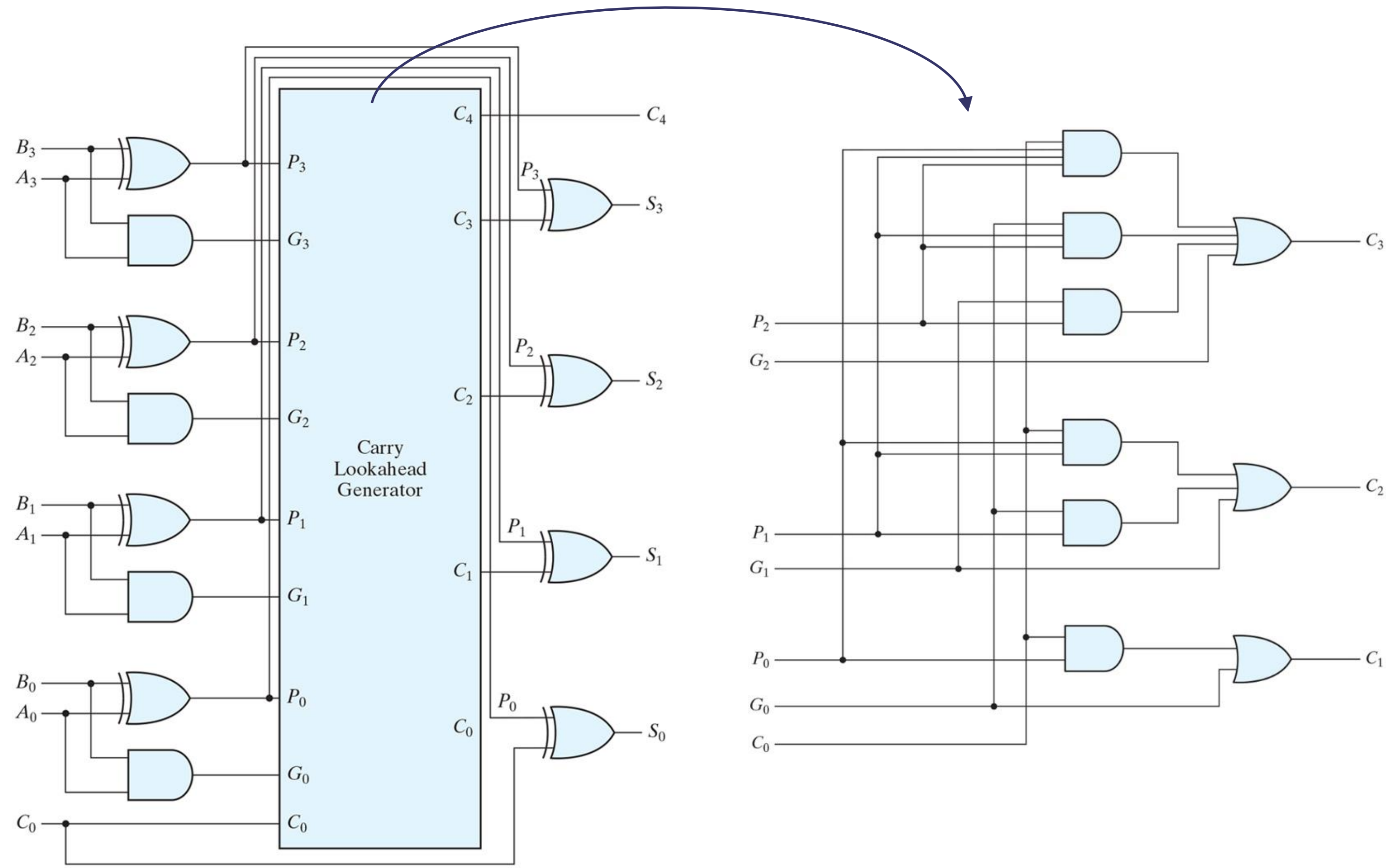


$c_3 = g_2 + p_2 . g_1 + p_2 . p_1 . g_0 + p_2 . p_1 . p_0 . c_0$

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

$c_1 = g_0 + p_0 . c_0$

CLA 4-bit Adder



Multiplication Approach (1st version)

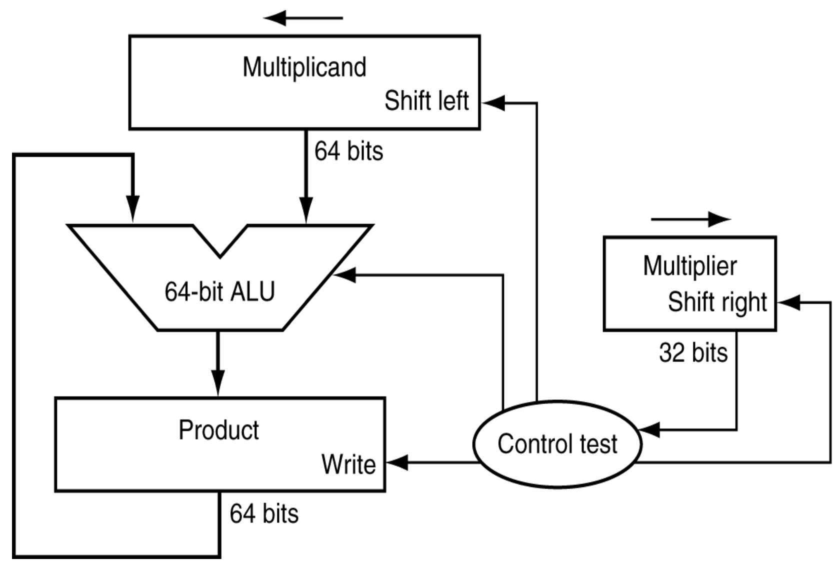
مضروب
(multiplicand)

مضروب‌فیه
(multiplier)

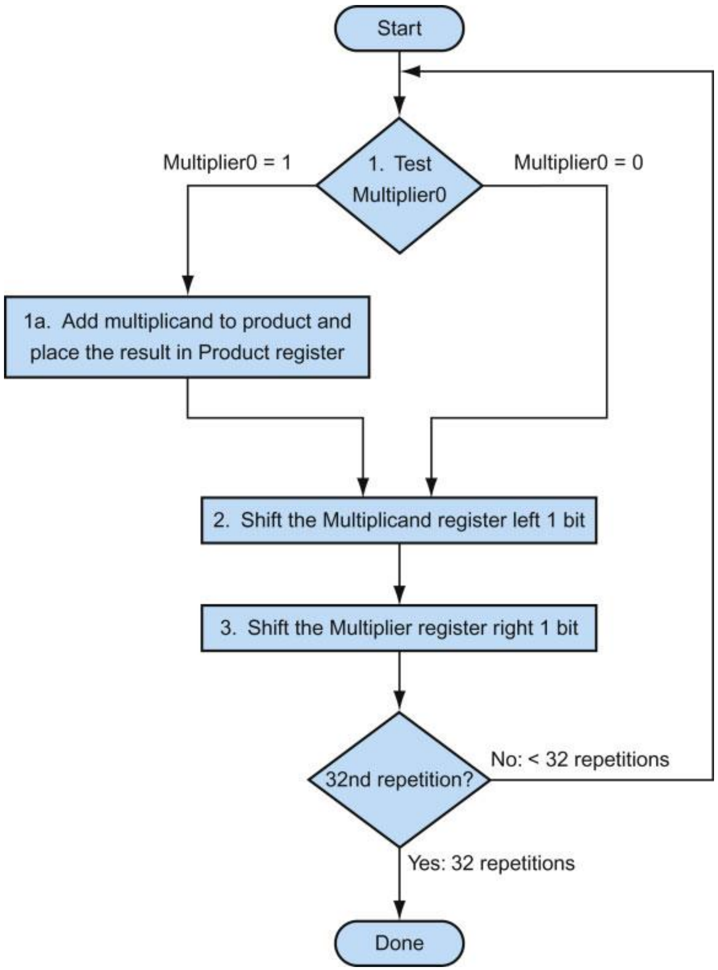
حاصل‌ضرب
(product)

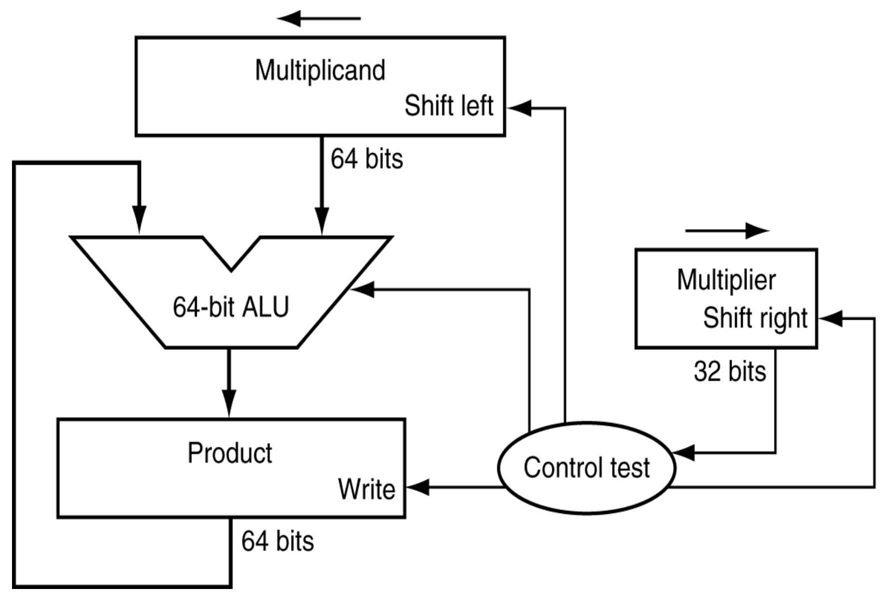
```
       1000
   ×   1001
   --------
       1000 +
      0000
     0000
    1000
   --------
    1001000
```

Length of product is the sum of operand lengths



Multiplication Algorithm (1st version)





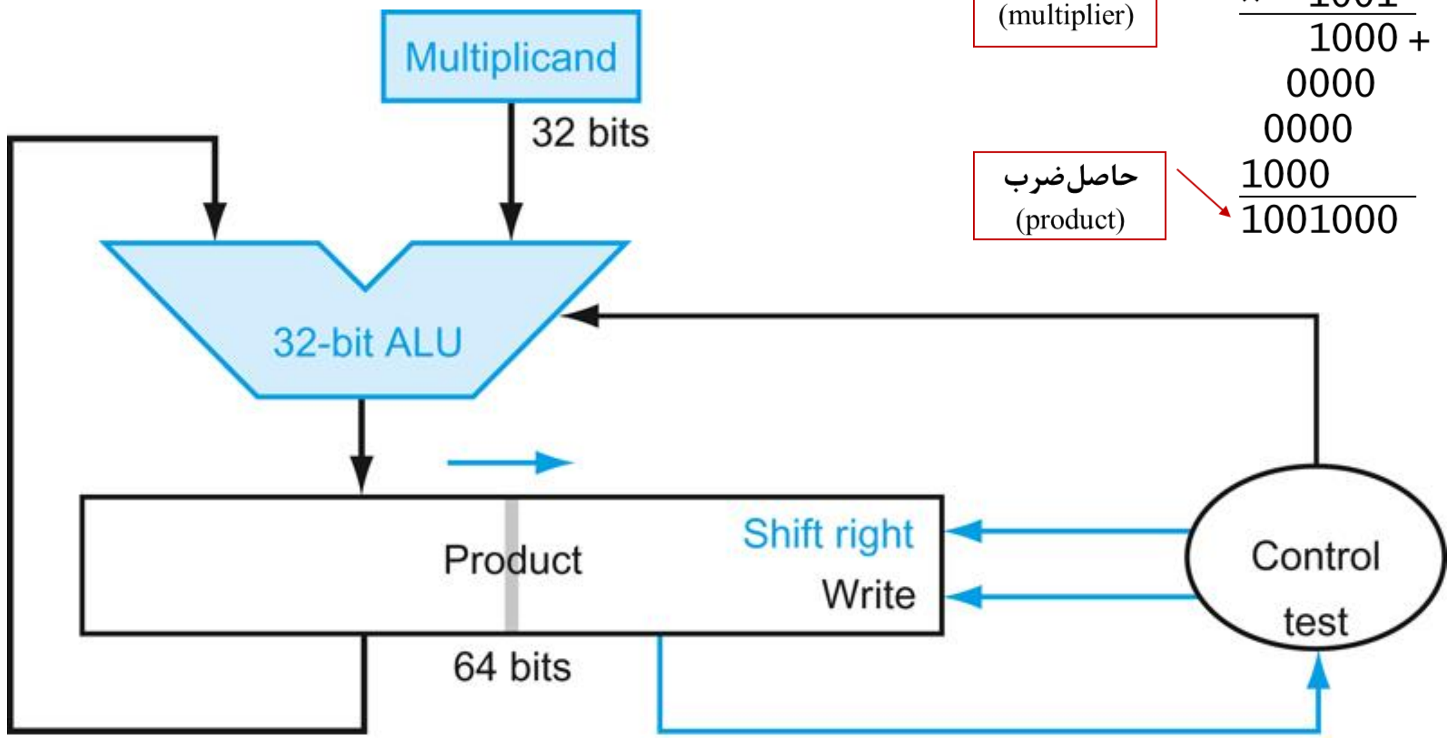
Multiplication (2nd version)

مضروب
(multiplicand)

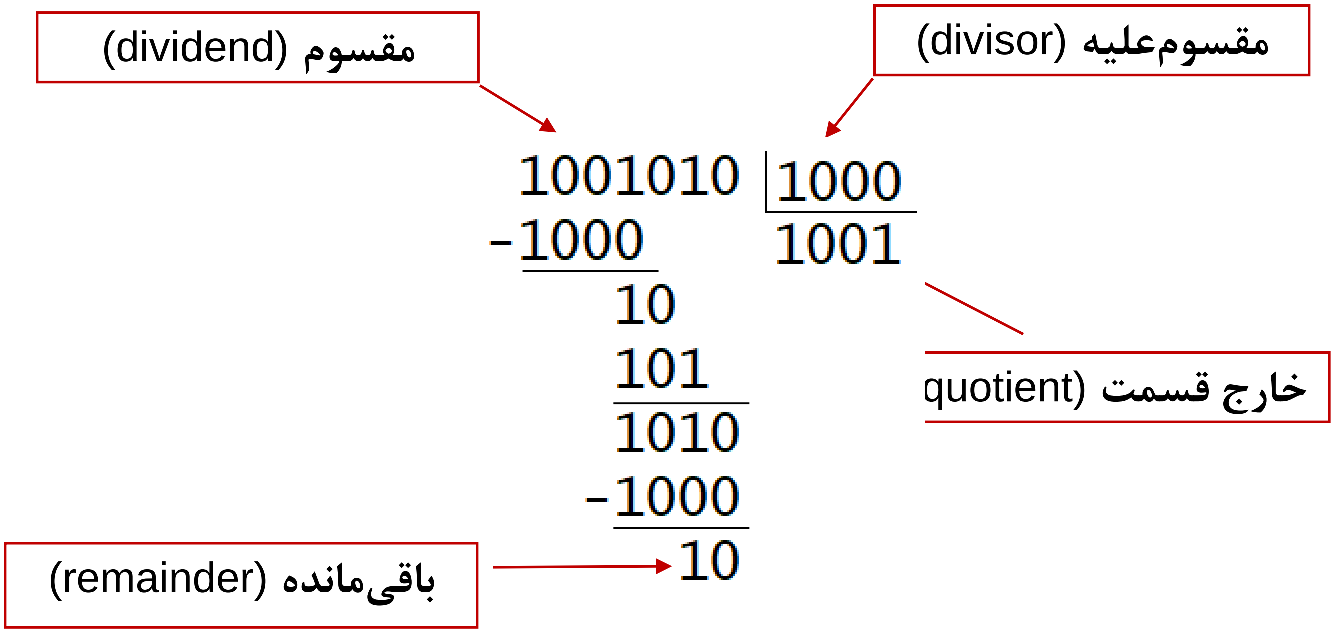
مضروب‌فیه
(multiplier)

حاصل‌ضرب
(product)

```
     1000
×    1001
---------
     1000 +
    0000
   0000
  1000
---------
  1001000
```



Division

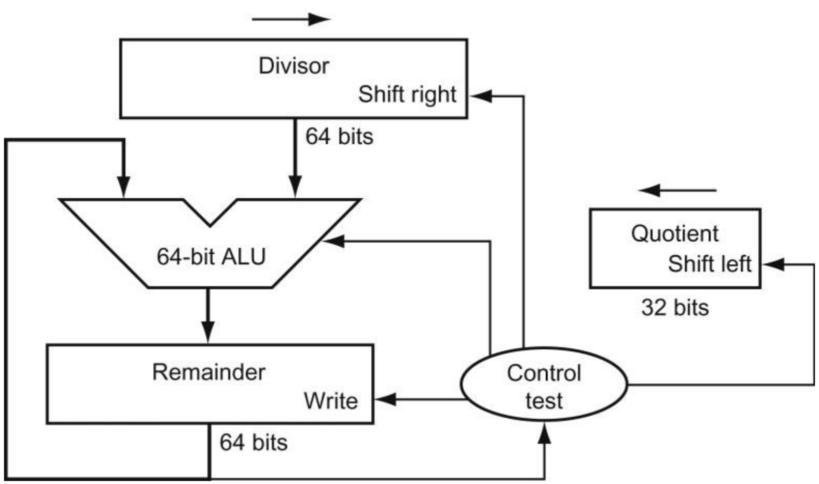


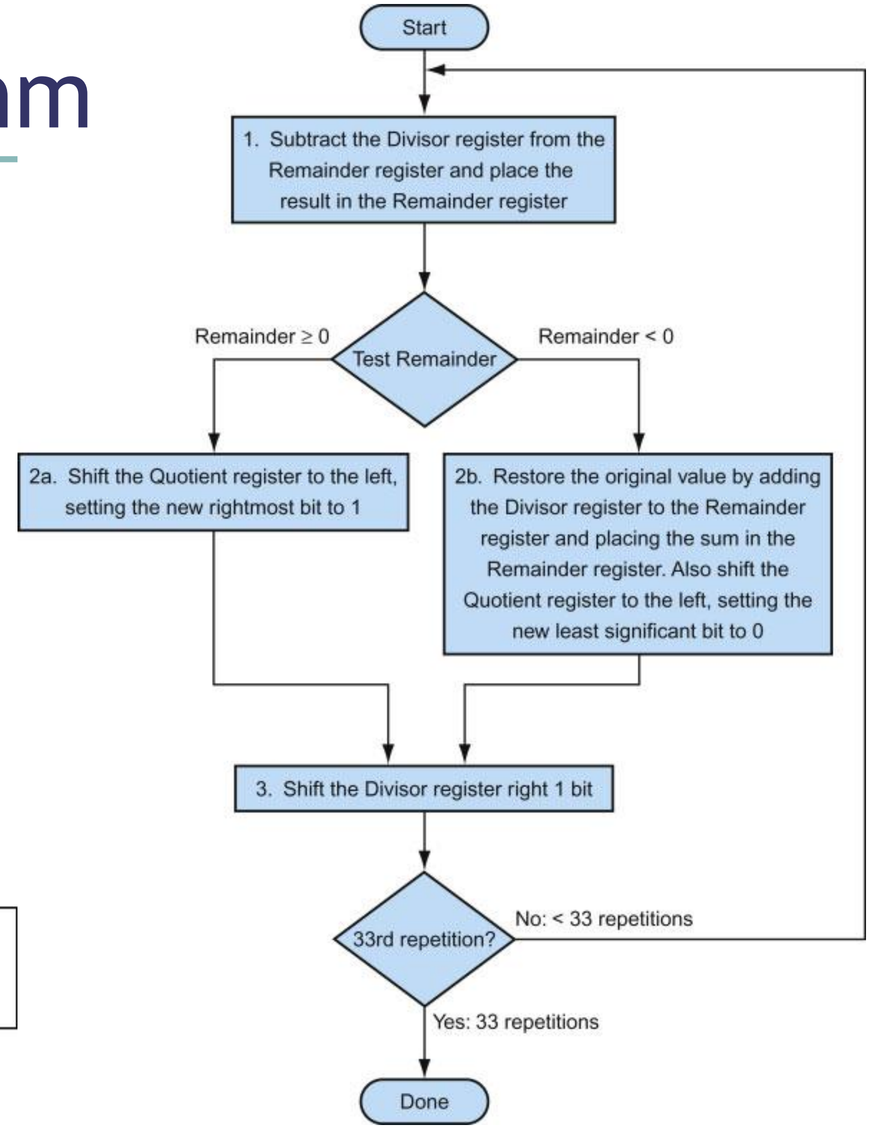
$$Dividend = Quotient \times Divisor + Remainder$$

$$|Remainder| < |Divisor|$$

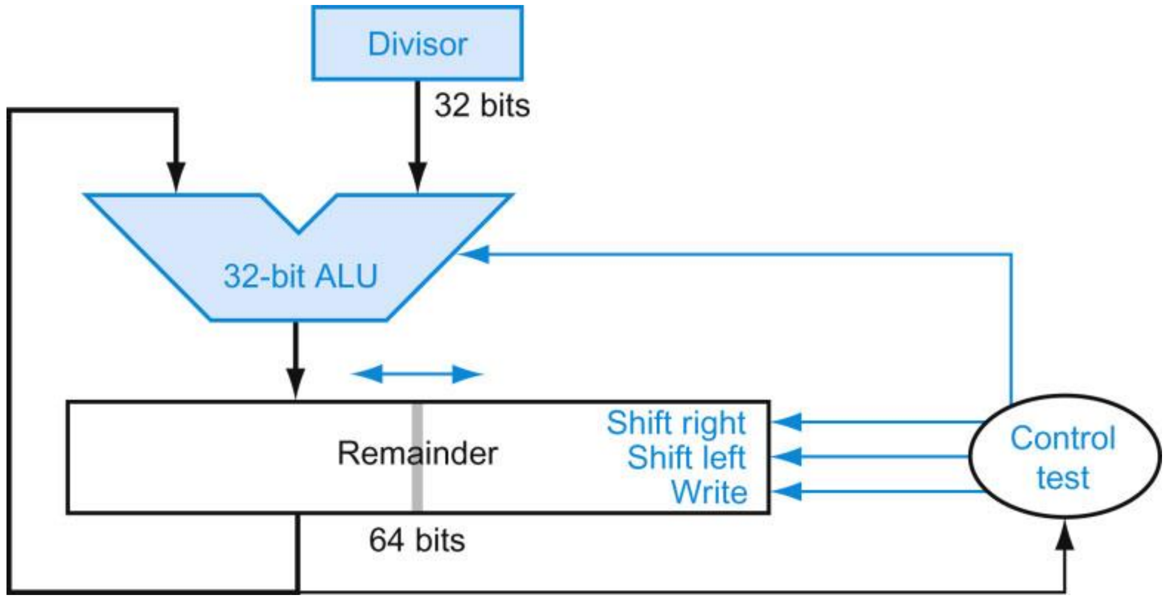
Division Algorithm

```
  1001010 |1000
 -1000     ----
  ----     1001
       10
       101
       ----
       1010
      -1000
       ----
         10
```





Division Algorithm (improved)



The Divisor register, ALU, and Quotient register are all 32 bits wide.

The ALU and Divisor registers are halved and the remainder is shifted left.

The Quotient register is combined with the right half of the Remainder register.

The Remainder register should really be 65 bits to make sure the carry out of the adder is not lost.

Signed Division

- Divide using absolute values, considering:

$$\mathit{Dividend} = \mathit{Quotient} \times \mathit{Divisor} + \mathit{Remainder}$$

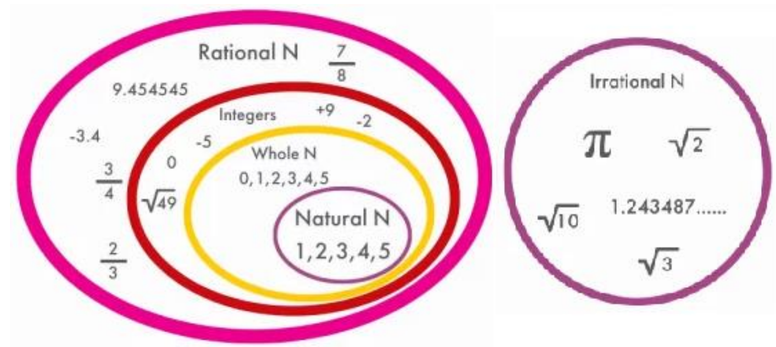
- Adjust sign of quotient and remainder as required
  - no change in the absolute value of quotient
  - the dividend and remainder must have the same signs
  - e.g., divide ±7 by ±2

Fallacy

- Just as a left shift instruction can replace an integer multiply by a power of 2, a right shift is the same as an integer division by a power of 2
  - Only true for unsigned integers
  - Even with arithmetic right shift
  - e.g., try to shift right -5 twice

Real Numbers

- Numbers with Fractions:
  - 3.14159…
  - 2.17
  - 0.0000001
  - 1.25×10^{-12}
  - $1.43 \times 10^{+12}$



- Representation in computers:
  - Fixed point
  - Floating point

Fixed-Point Representation

- A real Example:
  - $d_{23}d_{22}\ldots d_1d_0.f_{-1}f_{-2}f_{-3}f_{-4}f_{-5}f_{-6}f_{-7}f_{-8}$
  - 24-bit: integer bits
  - 8-bit: fraction bits
- Application
  - Used in CPUs with no floating-point unit
    - Embedded microprocessors and microcontrollers
  - Digital Signal Processing (DSP) applications

Fixed-Point Representation (cont.)

- Consider 5-Bit Representation
  - $d_2d_1d_0.f_{-1}f_{-2}$
  - $(d_2\times-2^2) + (d_1\times2^1) + (d_0\times2^0) + (f_{-1}\times2^{-1}) + (f_{-2}\times2^{-2})$
- Largest positive number?
- Smallest positive number?
- Largest magnitude negative number?
- Smallest magnitude negative number?

Fixed-Point Representation (cont.)

- Arithmetic:
  - 011.11 + 011.11 = 111.10
  - 010.10 × 000.10 = 000001.0100
  - 000.01 × 000.01 = 000000.0001
  - 111.10 × 111.10 = ?

out of range (overflow)

out of range (underflow)

Fixed-Point Representation (cont.)

- Arithmetic:
  - 011.11 + 011.11 = 111.10
  - 010.10 × 000.10 = 000001.0100
  - 000.01 × 000.01 = 000000.0001
  - 111.10 × 111.10 = 001010.1001

out of range (overflow)

out of range (underflow)

Both overflow & underflow

Fixed-Point Representation (cont.)

☺ Pros

- Simple hardware
- Fast computation
- Different precisions at different applications?
  - 24bits/8bits , 18bits/12bits, 8bits/24bits

☹ Cons

- Low precision
- Small range

Scientific Notation (Decimal)

mantissa

exponent

$6.02_{10} \times 10^{23}$

decimal point

radix (base)

- Normalized Form:
  - Exactly one non-zero digit to left of decimal point
- Alternatives to representing 0.0000000012:
  - Normalized: 1.2×10^{-9}
  - Not normalized: 0.12×10^{-8}, 12.0×10^{-10}

Normalized Scientific Notation (Binary)

fraction

exponent

$$1.xxxx_{two} \times 2^{yyy}$$

binary point

radix (base)

Floating-Point Notation

- Floating Point Notation Consists of:
  - Fraction (F): 23 bits
  - Exponent (E): 8 bits
  - Sign bit (S)
  - Also called, single precision floating-point
- $N = (-1)^S \times (1+F) \times 2^E$

31	30	...	24	23	22	21	...	1	0
S	Exponent				Fraction				

Floating-Point Notation (cont.)

- Pros (compared to fixed-point)
  - Very Wide Range
  - More precision bits
- Cons (compared to fixed-point)
  - Arithmetic operation more complicated
  - HW more complicated

<table>
<tr><td>31</td><td>30</td><td>…</td><td>24</td><td>23</td><td>22</td><td>21</td><td>…</td><td>1</td><td>0</td></tr>
<tr><td>S</td><td colspan="4">Exponent</td><td colspan="5">Fraction</td></tr>
</table>

Floating-Point Notation (cont.)

- $N = (-1)^S \times (1 + F) \times 2^E$
- Precision versus Range
  - More precision ➔ smaller range?
  - Wider range ➔ less precision?
- True for fixed-point
  - Not necessarily correct for floating point

31	30	…	24	23	22	21	…	1	0
S	Exponent				Fraction				

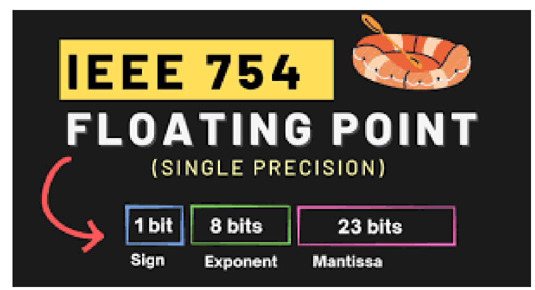
Floating-Point Notation (cont.)

- Overflow:
  - Exponent too large to fit in “Exponent” field
- Underflow:
  - Non-zero fraction so small to represent
  - Negative exponent too large to fit

31	30	…	24	23	22	21	…	1	0
S	Exponent				Fraction				

IEEE 754 Floating Point Standard

- There are many reasonable ways to represent floating-point numbers
- For many years, computer manufacturers used incompatible floating-point formats
- Results from one computer could not directly be interpreted by another computer.
- The Institute of Electrical and Electronics Engineers solved this problem by defining the IEEE 754 floating point standard in 1985 defining floating-point numbers



- This floating-point format is now almost universally used

IEEE 754 – Single Precision

- Signed-magnitude notation for mantissa
- Biased (Excess 2^{n-1}-1) notation for exponent
- E_{min}=00000001
- E_{max}=11111110
- E=00000000 reserved for zero
- E=11111111 reserved for infinity & NaN
- Smallest positive no: 1.17549435 E^{-38}
- Largest positive no: 3.4028235 E^{+38}

31	30	…	24	23	22	21	…	1	0
S	Exponent				Fraction				

$$N = (-1)^S \times (1 + F) \times 2^E$$

IEEE 754 – Double Precision

- Two words long (64 bits)
- Reduced chances of overflow/underflow
- Format
  - Sign bit (S)
  - Fraction (F): 52 bits
  - Exponent (E): 11 bits
- A bias of 1023 in Exponential part

More on IEEE 754 Standard

- Single precision (32 bits)/ Double precision (64 bits)
- Normalized/ Denormalized forms
- Standard definitions for zero, infinity, NaN
- Check: https://www.h-schmidt.net/FloatConverter/IEEE754.html

Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	Nonzero	0	Nonzero	± denormalized number
1–254	Anything	1–2046	Anything	± floating-point number
255	0	2047	0	± infinity
255	Nonzero	2047	Nonzero	NaN (Not a Number)

Denormalized Forms

- An attempt to squeeze every last bit of precision from a floating-point operation
- The smallest positive single precision normalized no:
  - $1.00000000000000000000000 \times 2^{-126}$
- The smallest single precision denormalized no:
  - $0.00000000000000000000001 \times 2^{-126} = 1.0 \times 2^{-149}$

31	30	...	24	23	22	21	...	1	0
S	Exponent				Fraction				

Floating-Point Addition

1. Align binary points
   - Shift number with smaller exponent (why?)
2. Add significands
3. Normalize result & check for over/underflow
4. Round and renormalize if necessary

Example

- Consider 0.5 + (-0.4375)
  - $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$

1. Align binary points (Shift number with smaller exponent)
   - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$
2. Add significands
   - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1} = 0.001_2 \times 2^{-1}$
3. Normalize result & check for over/underflow
   - $1.000_2 \times 2^{-4}$, with no over/underflow
4. Round and renormalize if necessary
   - $1.000_2 \times 2^{-4}$ (no change) $= 0.0625$

Floating-Point Multiplication

1. Add exponents
   - For biased exponents, subtract bias from sum
2. Multiply significands
3. Normalize result & check for over/underflow
4. Round and renormalize if necessary
5. Determine sign of result from signs of operands

Example

- Consider $0.5 \times (-0.4375)$
  - $1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$

1. Add exponents
   - Biased: $(-1 + 127) + (-2 + 127) = -3 + 254 - 127 = -3 + 127$
2. Multiply significands
   - $1.000_2 \times 1.110_2 = 1.1102 \Rightarrow 1.110_2 \times 2^{-3}$
3. Normalize result & check for over/underflow
   - $1.110_2 \times 2^{-3}$, with no over/underflow
4. Round and renormalize if necessary
   - $1.110_2 \times 2^{-3} = 0.21875$
5. Determine sign of result from signs of operands
   - $-1.1102 \times 2^{-3} = -0.21875$

Concluding Remarks

- Bits have no inherent meaning
  - Interpretation depends on the operations applied
- Computer representations of numbers
  - Finite range and precision
  - Need to account for this in programs
- Bounded range and precision
  - Operations can overflow and underflow

Outlines

- Weighted Number System
- Signed Number Representation
- Arithmetic Operations
  - Addition/ Subtraction/ Multiplication/ Division
- Floating Point Representation (IEEE-754 Standard)