Department of Electrical Engineering, Feng-Chia University

# Chapter 3
# Arithmetic for Computers
# (Part 2)

王振傑 (Chen-Chieh Wang)
ccwang@mail.ee.ncku.edu.tw

Department of Electrical Engineering, Feng-Chia University

## Outline

3.3 Multiplication
3.4 Division
3.5 Floating Point
3.9 Concluding Remarks

*Text Book : P220*

# Multiplication

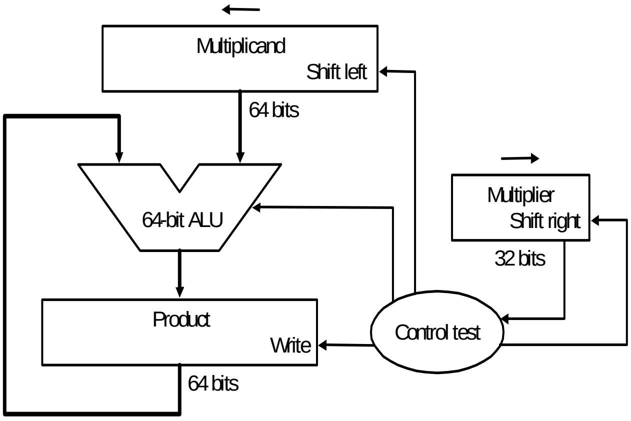
- More complicated than addition
  - accomplished via shifting and addition
- More time and more area
- Let's look at 3 versions based on a gradeschool algorithm

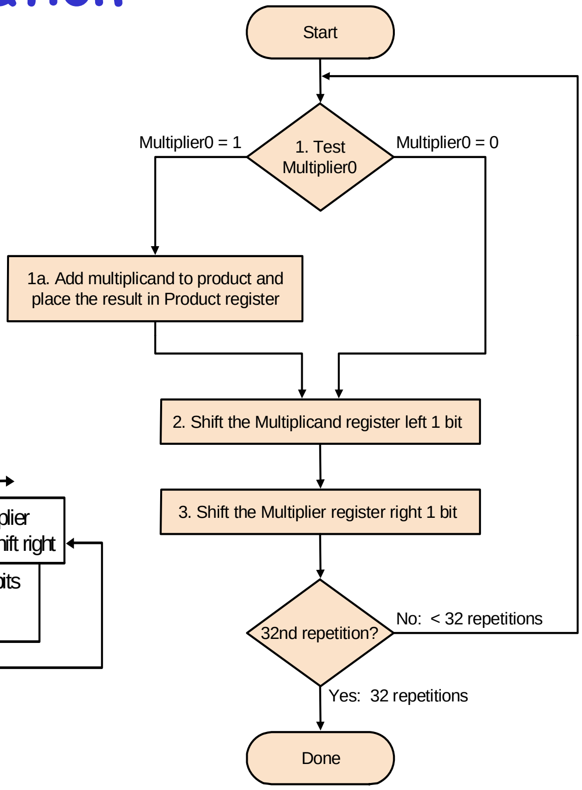
```
      0010   (multiplicand)
 ___x_1011   (multiplier)
```

- See if the right most bit of the multiplier is 1.
  - Shift 1 bit right for the multiplier
  - Note the position of placement - Multiplicand is shifted left.

*Text Book : P221 ~ P222*

# Multiplication: First Version Implementation




Start
Multiplier0 = 1
1. Test Multiplier0
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done

Department of Electrical Engineering, Feng-Chia University

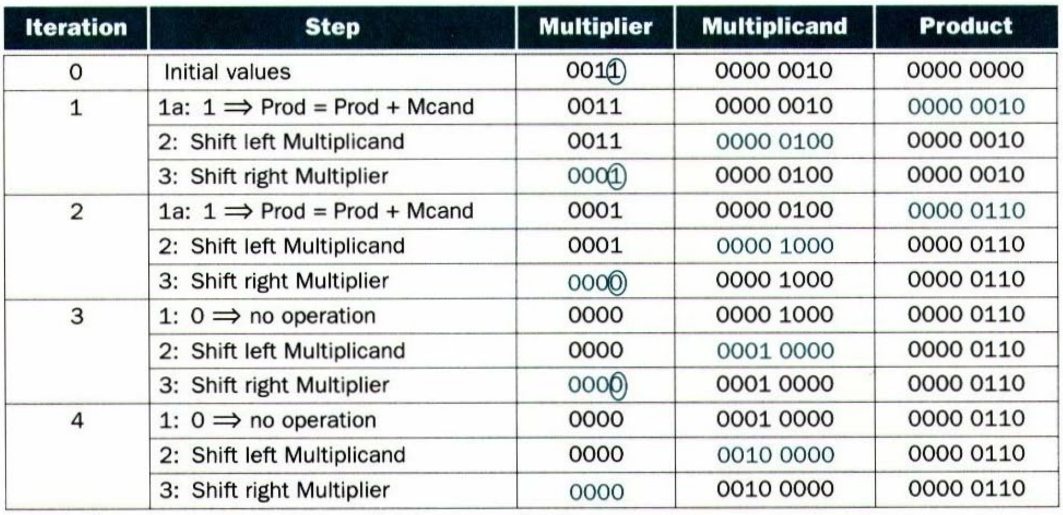
*Text Book : P224*

**EXAMPLE**

## A Multiply Algorithm

Using 4-bit numbers to save space, multiply $2_{ten}$ x $3_{ten}$, or $0010_{two}$ x $0011_{two}$.

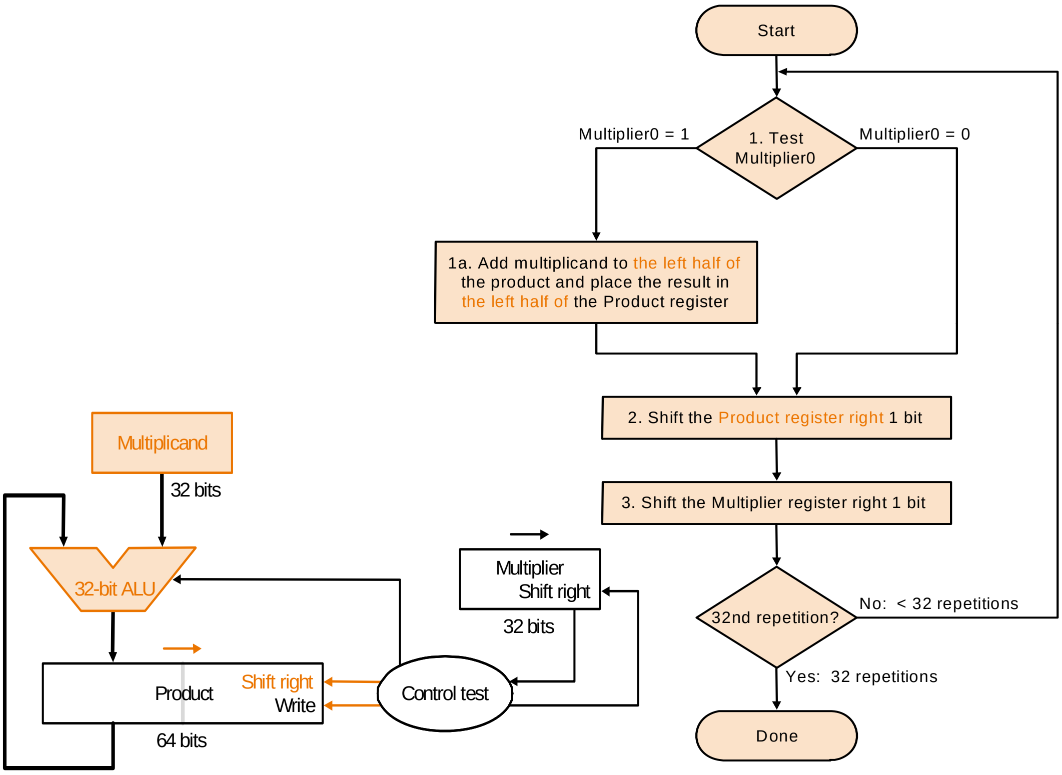
**ANSWER**



| Iteration | Step | Multiplier | Multiplicand | Product |
|---|---|---|---|---|
| 0 | Initial values | 0011 | 0000 0010 | 0000 0000 |
| 1 | 1a: 1 ⇒ Prod = Prod + Mcand | 0011 | 0000 0010 | 0000 0010 |
| | 2: Shift left Multiplicand | 0011 | 0000 0100 | 0000 0010 |
| | 3: Shift right Multiplier | 0001 | 0000 0100 | 0000 0010 |
| 2 | 1a: 1 ⇒ Prod = Prod + Mcand | 0001 | 0000 0100 | 0000 0110 |
| | 2: Shift left Multiplicand | 0001 | 0000 1000 | 0000 0110 |
| | 3: Shift right Multiplier | 0000 | 0000 1000 | 0000 0110 |
| 3 | 1: 0 ⇒ no operation | 0000 | 0000 1000 | 0000 0110 |
| | 2: Shift left Multiplicand | 0000 | 0001 0000 | 0000 0110 |
| | 3: Shift right Multiplier | 0000 | 0001 0000 | 0000 0110 |
| 4 | 1: 0 ⇒ no operation | 0000 | 0001 0000 | 0000 0110 |
| | 2: Shift left Multiplicand | 0000 | 0010 0000 | 0000 0110 |
| | 3: Shift right Multiplier | 0000 | 0010 0000 | 0000 0110 |

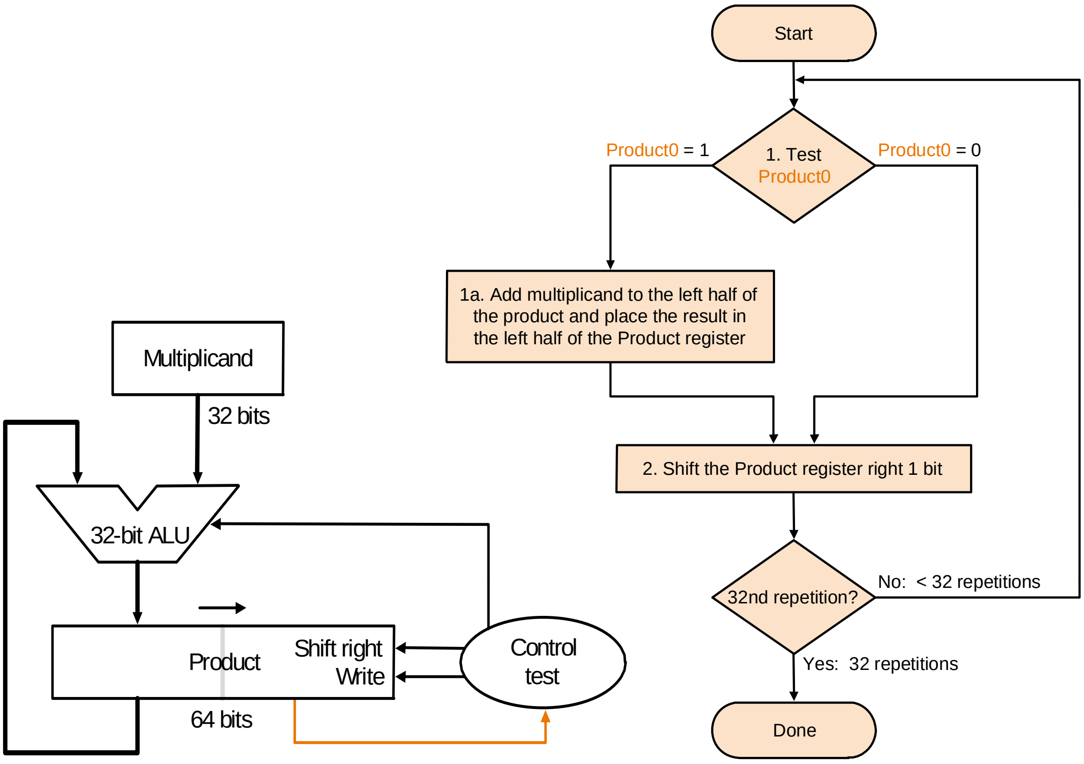
Department of Electrical Engineering, Feng-Chia University

# Multiplication: Second Version



*Text Book : P223*

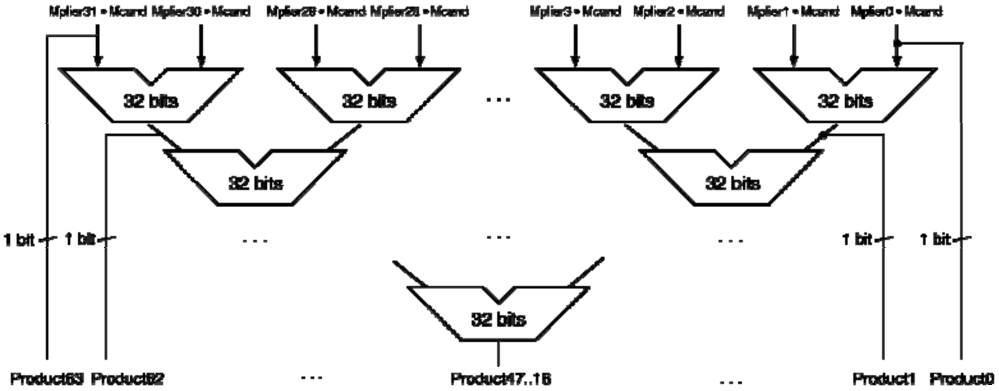
# Multiplication: Final Version



*Text Book : P225*

# Faster Multiplier

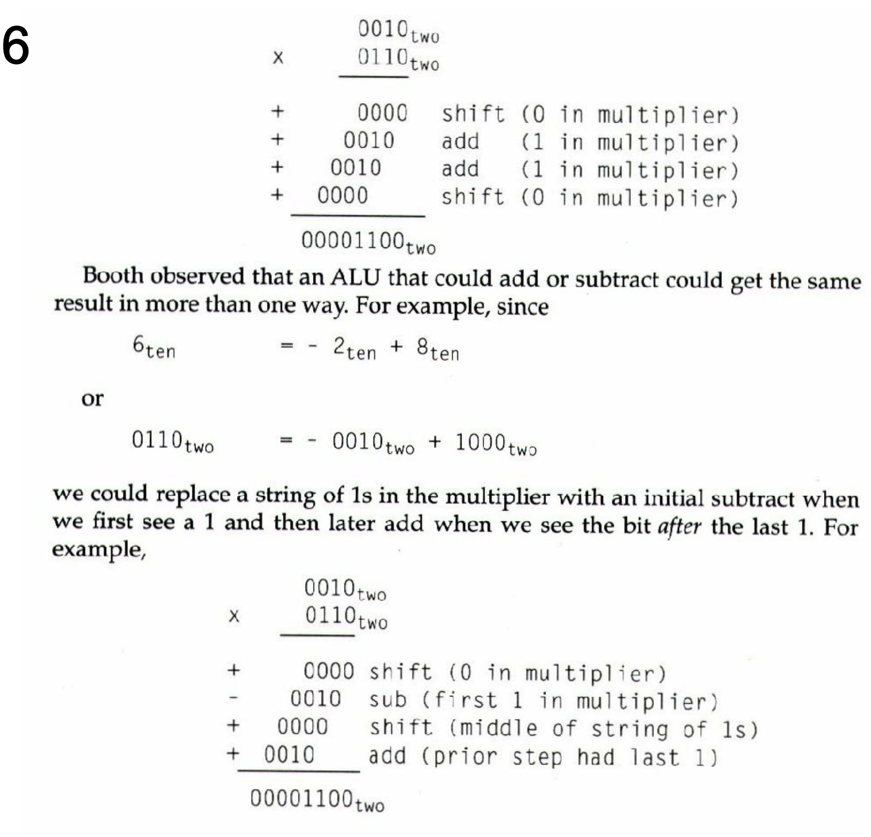
- Uses multiple adders
  - Cost/performance tradeoff



- Can be pipelined
  - Several multiplication performed in parallel

# Signed Multiplication : Booth's Algorithm

- 2 x 6



```
             0010two
x            0110two
          ---------
+            0000    shift (0 in multiplier)
+           0010     add   (1 in multiplier)
+          0010      add   (1 in multiplier)
+         0000       shift (0 in multiplier)
         ----------
         00001100two
```

Booth observed that an ALU that could add or subtract could get the same result in more than one way. For example, since

$6_{ten} = -\ 2_{ten} + 8_{ten}$

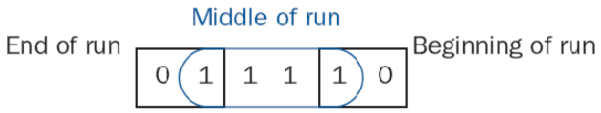
or

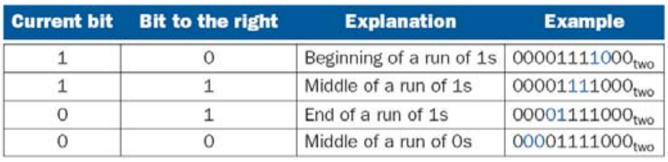
$0110_{two} = -\ 0010_{two} + 1000_{two}$

we could replace a string of 1s in the multiplier with an initial subtract when we first see a 1 and then later add when we see the bit *after* the last 1. For example,

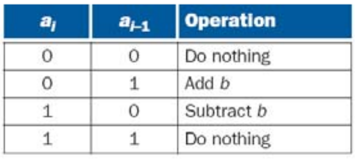
```
             0010two
x            0110two
          ---------
+            0000 shift (0 in multiplier)
-           0010  sub (first 1 in multiplier)
+          0000   shift (middle of string of 1s)
+         0010    add (prior step had last 1)
         ----------
         00001100two
```

# Booth's Algorithm

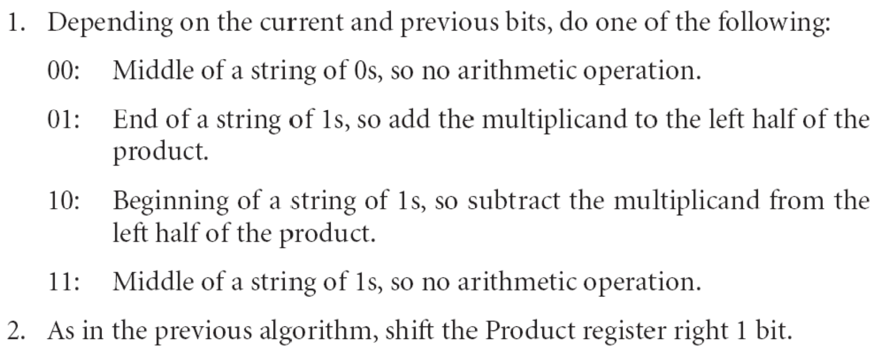




| Current bit | Bit to the right | Explanation | Example |
|---|---|---|---|
| 1 | 0 | Beginning of a run of 1s | $00001111000_{two}$ |
| 1 | 1 | Middle of a run of 1s | $00001111000_{two}$ |
| 0 | 1 | End of a run of 1s | $00001111000_{two}$ |
| 0 | 0 | Middle of a run of 0s | $00001111000_{two}$ |



| $a_i$ | $a_{i-1}$ | Operation |
|---|---|---|
| 0 | 0 | Do nothing |
| 0 | 1 | Add *b* |
| 1 | 0 | Subtract *b* |
| 1 | 1 | Do nothing |



1. Depending on the current and previous bits, do one of the following:

   00: Middle of a string of 0s, so no arithmetic operation.

   01: End of a string of 1s, so add the multiplicand to the left half of the product.

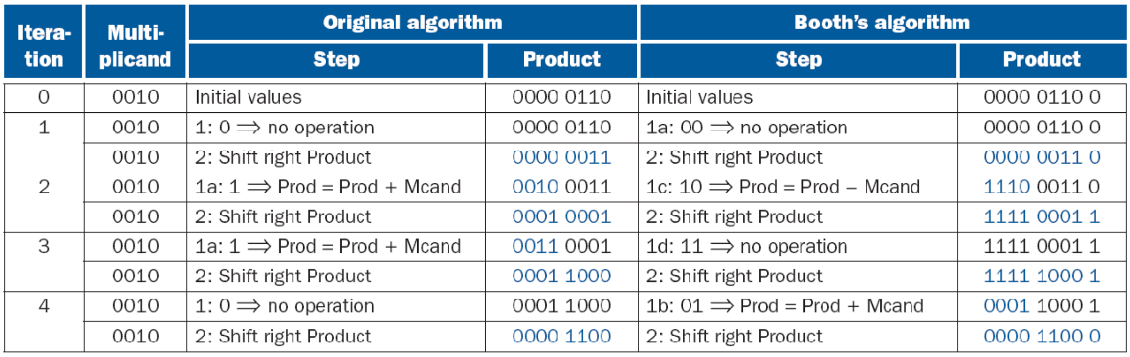
   10: Beginning of a string of 1s, so subtract the multiplicand from the left half of the product.

   11: Middle of a string of 1s, so no arithmetic operation.

2. As in the previous algorithm, shift the Product register right 1 bit.

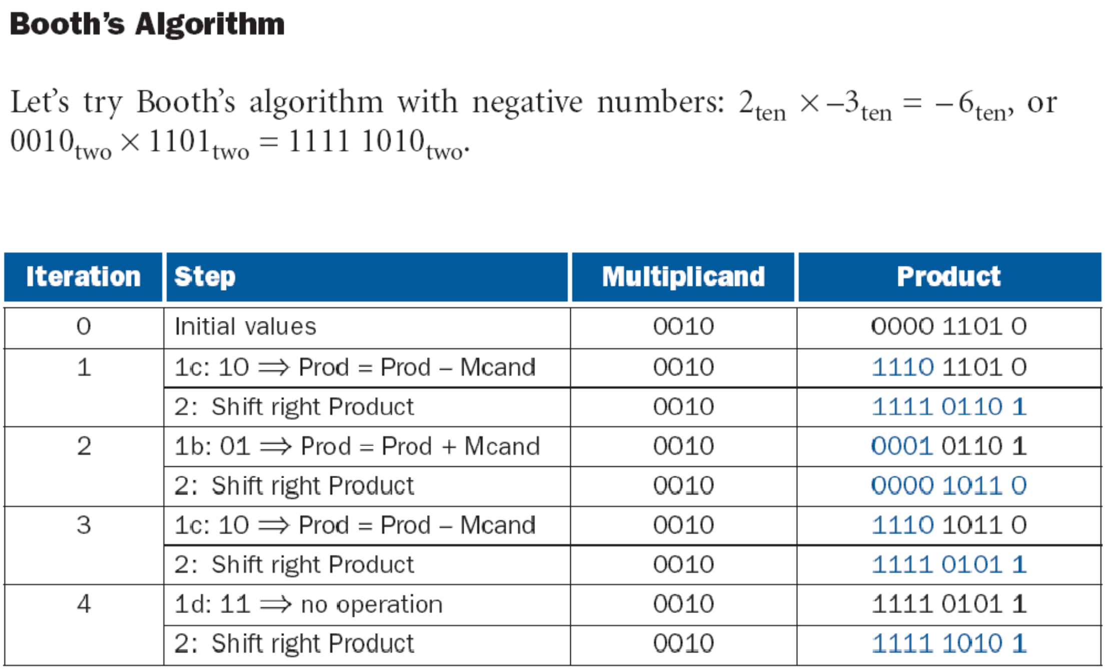
# Booth's Algorithm Example

- 2 x 6



| Itera-tion | Multi-plicand | Original algorithm | | Booth's algorithm | |
|---|---|---|---|---|---|
| | | Step | Product | Step | Product |
| 0 | 0010 | Initial values | 0000 0110 | Initial values | 0000 0110 0 |
| 1 | 0010 | 1: 0 ⇒ no operation | 0000 0110 | 1a: 00 ⇒ no operation | 0000 0110 0 |
| | 0010 | 2: Shift right Product | 0000 0011 | 2: Shift right Product | 0000 0011 0 |
| 2 | 0010 | 1a: 1 ⇒ Prod = Prod + Mcand | 0010 0011 | 1c: 10 ⇒ Prod = Prod – Mcand | 1110 0011 0 |
| | 0010 | 2: Shift right Product | 0001 0001 | 2: Shift right Product | 1111 0001 1 |
| 3 | 0010 | 1a: 1 ⇒ Prod = Prod + Mcand | 0011 0001 | 1d: 11 ⇒ no operation | 1111 0001 1 |
| | 0010 | 2: Shift right Product | 0001 1000 | 2: Shift right Product | 1111 1000 1 |
| 4 | 0010 | 1: 0 ⇒ no operation | 0001 1000 | 1b: 01 ⇒ Prod = Prod + Mcand | 0001 1000 1 |
| | 0010 | 2: Shift right Product | 0000 1100 | 2: Shift right Product | 0000 1100 0 |

# Booth's Algorithm Example



### Booth's Algorithm

Let's try Booth's algorithm with negative numbers: $2_{\text{ten}} \times -3_{\text{ten}} = -6_{\text{ten}}$, or $0010_{\text{two}} \times 1101_{\text{two}} = 1111\ 1010_{\text{two}}$.

| Iteration | Step | Multiplicand | Product |
|---|---|---|---|
| 0 | Initial values | 0010 | 0000 1101 0 |
| 1 | 1c: 10 ⇒ Prod = Prod – Mcand | 0010 | 1110 1101 0 |
| | 2: Shift right Product | 0010 | 1111 0110 1 |
| 2 | 1b: 01 ⇒ Prod = Prod + Mcand | 0010 | 0001 0110 1 |
| | 2: Shift right Product | 0010 | 0000 1011 0 |
| 3 | 1c: 10 ⇒ Prod = Prod – Mcand | 0010 | 1110 1011 0 |
| | 2: Shift right Product | 0010 | 1111 0101 1 |
| 4 | 1d: 11 ⇒ no operation | 0010 | 1111 0101 1 |
| | 2: Shift right Product | 0010 | 1111 1010 1 |

# MIPS Multiplication

- Two 32-bit registers for product
  - HI: most-significant 32 bits
  - LO: least-significant 32-bits
- Instructions
  - `mult rs, rt` / `multu rs, rt`
    - 64-bit product in HI/LO
  - `mfhi rd` / `mflo rd`
    - Move from HI/LO to rd
    - Can test HI value to see if product overflows 32 bits
  - `mul rd, rs, rt`
    - Least-significant 32 bits of product –> rd

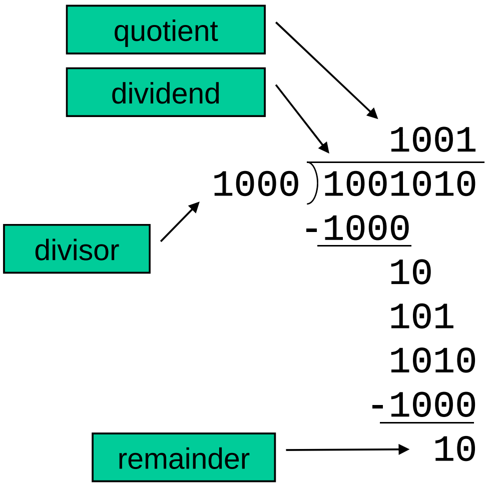
# Outline

3.3 Multiplication
3.4 Division
3.5 Floating Point
3.9 Concluding Remarks

Department of Electrical Engineering, Feng-Chia University

*Text Book : P226 ~ P227*

# Division



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

*n*-bit operands yield *n*-bit quotient and remainder

- Check for 0 divisor
- Long division approach
  - If divisor ≤ dividend bits
    - 1 bit in quotient, subtract
  - Otherwise
    - 0 bit in quotient, bring down next dividend bit
- Restoring division
  - Do the subtract, and if remainder goes < 0, add divisor back
- Signed division
  - Divide using absolute values
  - Adjust sign of quotient and remainder as required

Dividend = Quotient × Divisor + Remainder

Department of Electrical Engineering, Feng-Chia University

# MIPS Division

- Use HI/LO registers for result
  - HI: 32-bit remainder
  - LO: 32-bit quotient
- Instructions
  - `div rs, rt` / `divu rs, rt`
  - No overflow or divide-by-0 checking
    - Software must perform checks if required
  - Use `mfhi`, `mflo` to access result

# Outline

3.3 Multiplication

3.4 Division

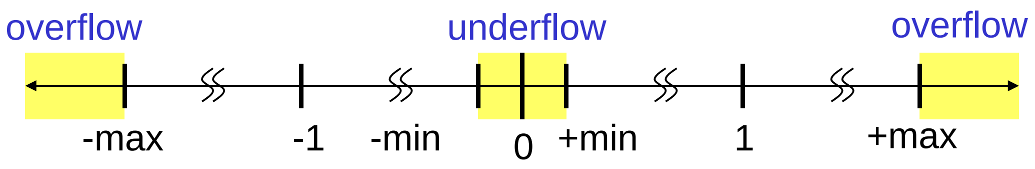
3.5 Floating Point

3.9 Concluding Remarks

*Text Book : P232*

# Floating Point (a brief look)

- We need a way to represent
  - numbers with fractions, e.g., 3.14159265… (pi)
  - very small numbers, e.g., 0.000000001 = 1.0 x 10 $^{-9}$
  - very large numbers, e.g., 3,155,760,000 = 3.15576 x 10 $^{9}$
- Representation:
  - sign, exponent, significand: $(-1)^{\text{sign}}$ x significand x $2^{\text{exponent}}$
  - more bits for significand gives more accuracy
  - more bits for exponent increases range



*Text Book : P233 ~ P234*

# IEEE 754 floating-point standard

- **Single precision** : 8 bit exponent, 23 bit significand

| S | Exponent | significand |
|---|---|---|
| 1 bit | 8 bits | 23 bits |

- **Double precision** : 11 bit exponent, 52 bit significand

| S | Exponent | significand ~ |
|---|---|---|
| 1 bit | 11 bits | 20 bits |

| ~ significand |
|---|
| 32 bits |

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

*Text Book : P236 ~ P237*

# IEEE 754 floating-point standard

- Example:
  - decimal: - 0.75 = - ( ½ + ¼ )
  - binary: $-0.11 = -1.1 \times 2^{-1}$
  - floating point: exponent = 126 = 01111110 (after add 127)
  - IEEE single precision:

| 1 | 01111110 | 10000000000000000000000 |
|---|---|---|
| 1 bit | 8 bits | 23 bits |

Department of Electrical Engineering, Feng-Chia University

*Text Book : P235*

# IEEE 754 Encoding

| 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) |

| S | Exponent | significand |
|---|---|---|
| 1 bit | 8 bits | 23 bits |

Department of Electrical Engineering, Feng-Chia University

*Text Book : P235*

# Denormalized Numbers

- Exponent = 000...0 ⇒ hidden bit is 0

$$x = (-1)^S \times (0 + \text{Fraction}) \times 2^{-\text{Bias}}$$

- Smaller than normal numbers
  - allow for gradual underflow, with diminishing precision
- Denormal with fraction = 000...0

$$x = (-1)^S \times (0 + 0) \times 2^{-\text{Bias}} = \pm 0.0$$

Two representations of 0.0!

*Text Book : P235*

# Infinities and NaNs

- Exponent = 111...1, Fraction = 000...0
  - ±Infinity
  - Can be used in subsequent calculations, avoiding need for overflow check
- Exponent = 111...1, Fraction ≠ 000...0
  - Not-a-Number (NaN)
  - Indicates illegal or undefined result
    - e.g., 0.0 / 0.0
  - Can be used in subsequent calculations

# IEEE 754 floating-point (32-bit)

- Maximum

| 0 | 11111110 | 11111111111111111111111 |
|---|---|---|
| 1 bit | 8 bits | 23 bits |

= 1.11111111111111111111111 x 2 $^{(+127)}$<br>
= (2- 0.00000000000000000000001) x 2$^{(+127)}$<br>
= 2 x 2$^{(+127)}$ - 2$^{(-23)}$ x 2$^{(+127)}$<br>
= 2$^{(+128)}$ - 2$^{(+104)}$

- Minimum

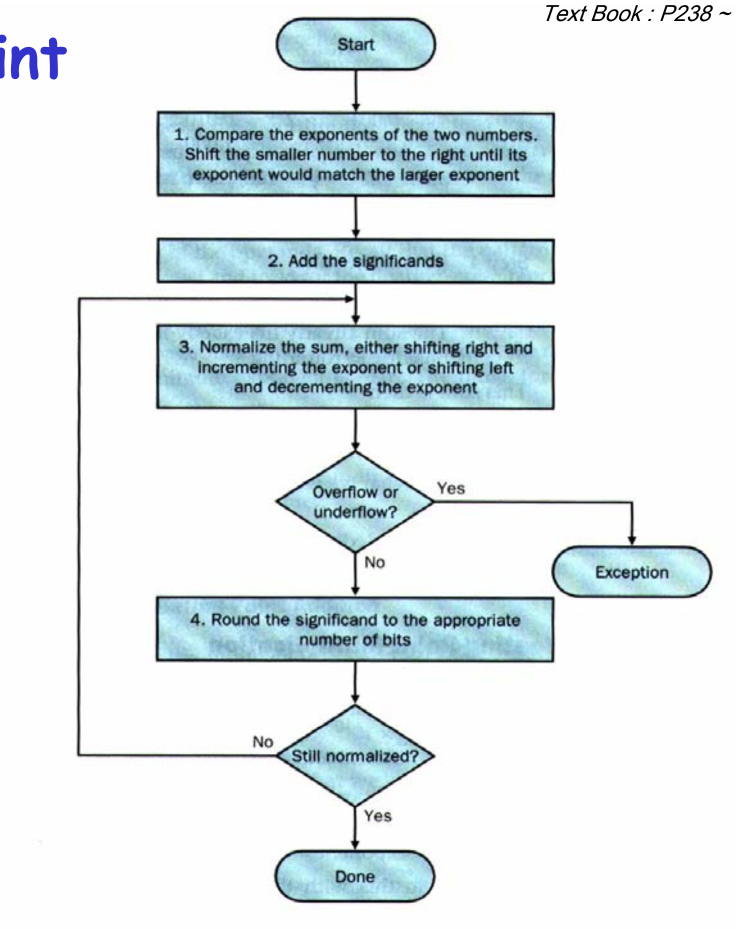
| 0 | 00000000 | 00000000000000000000001 |
|---|---|---|
| 1 bit | 8 bits | 23 bits |

= 0.00000000000000000000001 x 2 $^{(-126)}$<br>
= 2$^{(-23)}$ x 2$^{(-126)}$<br>
= 2$^{(-149)}$

*Text Book : P238 ~ P241*

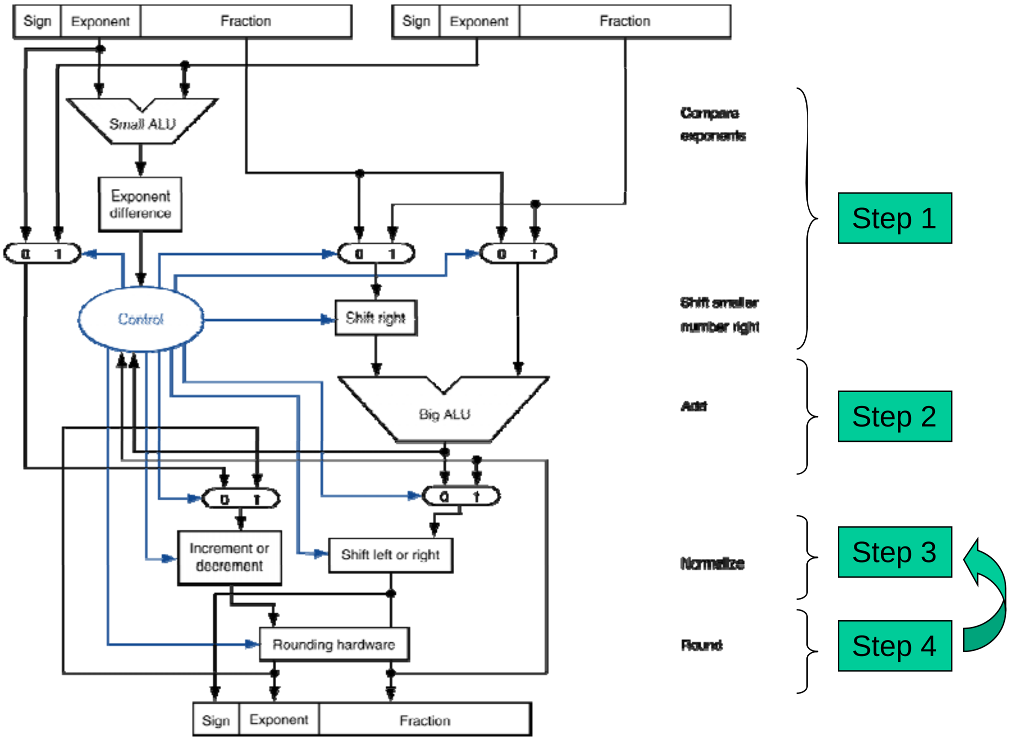
# Floating-point Addition

- guard and round ?


Start
1. Compare the exponents of the two numbers. Shift the smaller number to the right until its exponent would match the larger exponent
2. Add the significands
3. Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to the appropriate number of bits
No
Still normalized?
Yes
Done

*Text Book : P243*

# FP Adder Hardware



Step 1

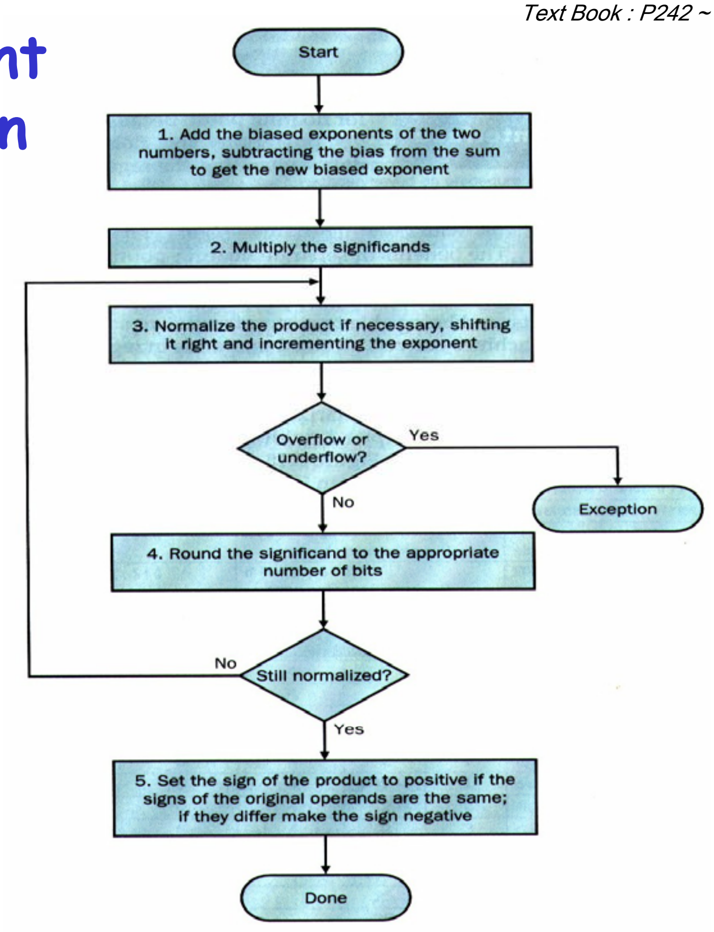
Step 2

Step 3

Step 4

# Floating-point Multiplication

Text Book : P242 ~ P246


Start
1. Add the biased exponents of the two numbers, subtracting the bias from the sum to get the new biased exponent
2. Multiply the significands
3. Normalize the product if necessary, shifting it right and incrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to the appropriate number of bits
Still normalized?
No
Yes
5. Set the sign of the product to positive if the signs of the original operands are the same; if they differ make the sign negative
Done

# FP Instructions in MIPS

- FP hardware is coprocessor 1
  - Adjunct processor that extends the ISA
- Separate FP registers
  - 32 single-precision: $f0, $f1, … $f31
  - Paired for double-precision: $f0/$f1, $f2/$f3, …
    - Release 2 of MIPs ISA supports 32 × 64-bit FP reg’s
- FP instructions operate only on FP registers
  - Programs generally don’t do integer ops on FP data, or vice versa
  - More registers with minimal code-size impact
- FP load and store instructions
  - `lwc1`, `ldc1`, `swc1`, `sdc1`
    - e.g., `ldc1 $f8, 32($sp)`

# FP Instructions in MIPS

- Single-precision arithmetic
  - `add.s`, `sub.s`, `mul.s`, div.s
    - e.g., `add.s $f0, $f1, $f6`
- Double-precision arithmetic
  - `add.d`, `sub.d`, `mul.d`, `div.d`
    - e.g., `mul.d $f4, $f4, $f6`
- Single- and double-precision comparison
  - `c.`*`xx`*`.s`, `c.`*`xx`*`.d` (*xx* is `eq`, `lt`, `le`, …)
  - Sets or clears FP condition-code bit
    - e.g. `c.lt.s $f3, $f4`
- Branch on FP condition code true or false
  - `bc1t`, `bc1f`
    - e.g., `bc1t TargetLabel`

# 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!

# Outline

3.3 Multiplication

3.4 Division

3.5 Floating Point

3.9 Concluding Remarks

# 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)

## In More Depth: Booth's Algorithm

A more elegant approach to multiplying signed numbers than above is called *Booth's algorithm.* It starts with the observation that with the ability to both add and subtract there are multiple ways to compute a product. Suppose we want to multiply $2_{ten}$ by $6_{ten}$, or $0010_{two}$ by $0110_{two}$:

```
         0010_two
x        0110_two
        ---------
+        0000 shift (0 in multiplier)
+       0010  add   (1 in multiplier)
+      0010   add   (1 in multiplier)
+     0000    shift (0 in multiplier)
     ----
      00001100_two
```

Booth observed that an ALU that could add or subtract could get the same result in more than one way. For example, since

$$6_{ten} = -\ 2_{ten} + 8_{ten}$$

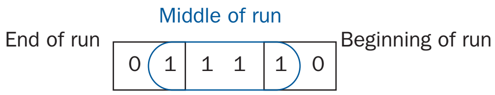
or

$$0110_{two} = -\ 0010_{two} + 1000_{two}$$

we could replace a string of 1s in the multiplier with an initial subtract when we first see a 1 and then later add when we see the bit *after* the last 1. For example,

```
      0010_two
x     0110_two
     ---------
+     0000  shift (0 in multiplier)
-    0010   sub (first 1 in multiplier)
+ 0000      shift (middle of string of 1s)
+0010       add (prior step had last 1)
 ----
00001100_two
```

Booth invented this approach in a quest for speed because in machines of his era shifting was faster than addition. Indeed, for some patterns his algorithm would be faster; it's our good fortune that it handles signed numbers as well, and we'll prove this later. The key to Booth's insight is in his classifying groups of bits into the beginning, the middle, or the end of a run of 1s:



Of course, a string of 0s already avoids arithmetic, so we can leave these alone.

If we are limited to looking at just 2 bits, we can then try to match the situation in the preceding drawing, according to the value of these 2 bits:

If we are limited to looking at just 2 bits, we can then try to match the situation in the preceding drawing, according to the value of these 2 bits:

| Current bit | Bit to the right | Explanation | Example |
|---|---|---|---|
| 1 | 0 | Beginning of a run of 1s | $00001111000_{two}$ |
| 1 | 1 | Middle of a run of 1s | $00001111000_{two}$ |
| 0 | 1 | End of a run of 1s | $00001111000_{two}$ |
| 0 | 0 | Middle of a run of 0s | $00001111000_{two}$ |

Booth's algorithm changes the first step of the algorithm—looking at 1 bit of the multiplier and then deciding whether to add the multiplicand—to looking at 2 bits of the multiplier. The new first step, then, has four cases, depending on the values of the 2 bits. Let's assume that the pair of bits examined consists of the current bit and the bit to the right—which was the current bit in the previous step. The second step is still to shift the product right. The new algorithm is then the following:

1. Depending on the current and previous bits, do one of the following:

   00: Middle of a string of 0s, so no arithmetic operation.

   01: End of a string of 1s, so add the multiplicand to the left half of the product.

   10: Beginning of a string of 1s, so subtract the multiplicand from the left half of the product.

   11: Middle of a string of 1s, so no arithmetic operation.

2. As in the previous algorithm, shift the Product register right 1 bit.

Now we are ready to begin the operation, shown in Figure 3.11.2. It starts with a 0 for the mythical bit to the right of the rightmost bit for the first stage. Figure 3.11.2 compares the two algorithms, with Booth's on the right. Note that Booth's operation is now identified according to the values in the 2 bits. By the fourth step, the two algorithms have the same values in the Product register.

The one other requirement is that shifting the product right must preserve the sign of the intermediate result, since we are dealing with signed numbers. The solution is to extend the sign when the product is shifted to the right. Thus, step 2 of the second iteration turns $1110\ 0011\ 0_{two}$ into $1111\ 0001\ 1_{two}$ instead of $0111\ 0001\ 1_{two}$. This shift is called an *arithmetic right shift* to differentiate it from a logical right shift.

| Iteration | Multiplicand | Original algorithm | | Booth's algorithm | |
|---|---|---|---|---|---|
| | | Step | Product | Step | Product |
| 0 | 0010 | Initial values | 0000 0110 | Initial values | 0000 0110 0 |
| 1 | 0010 | 1: 0 ⇒ no operation | 0000 0110 | 1a: 00 ⇒ no operation | 0000 0110 0 |
| | 0010 | 2: Shift right Product | 0000 0011 | 2: Shift right Product | 0000 0011 0 |
| 2 | 0010 | 1a: 1 ⇒ Prod = Prod + Mcand | 0010 0011 | 1c: 10 ⇒ Prod = Prod – Mcand | 1110 0011 0 |
| | 0010 | 2: Shift right Product | 0001 0001 | 2: Shift right Product | 1111 0001 1 |
| 3 | 0010 | 1a: 1 ⇒ Prod = Prod + Mcand | 0011 0001 | 1d: 11 ⇒ no operation | 1111 0001 1 |
| | 0010 | 2: Shift right Product | 0001 1000 | 2: Shift right Product | 1111 1000 1 |
| 4 | 0010 | 1: 0 ⇒ no operation | 0001 1000 | 1b: 01 ⇒ Prod = Prod + Mcand | 0001 1000 1 |
| | 0010 | 2: Shift right Product | 0000 1100 | 2: Shift right Product | 0000 1100 0 |

**FIGURE 3.11.2 Comparing algorithm in Booth's algorithm for positive numbers.** The bit(s) examined to determine the next step is circled in color.

### Booth's Algorithm

EXAMPLE

Let's try Booth's algorithm with negative numbers: $2_{ten} \times -3_{ten} = -6_{ten}$, or $0010_{two} \times 1101_{two} = 1111\ 1010_{two}$.

ANSWER

Figure 3.11.3 shows the steps.

| Iteration | Step | Multiplicand | Product |
|---|---|---|---|
| 0 | Initial values | 0010 | 0000 1101 0 |
| 1 | 1c: 10 ⟹ Prod = Prod – Mcand | 0010 | 1110 1101 0 |
| | 2: Shift right Product | 0010 | 1111 0110 1 |
| 2 | 1b: 01 ⟹ Prod = Prod + Mcand | 0010 | 0001 0110 1 |
| | 2: Shift right Product | 0010 | 0000 1011 0 |
| 3 | 1c: 10 ⟹ Prod = Prod – Mcand | 0010 | 1110 1011 0 |
| | 2: Shift right Product | 0010 | 1111 0101 1 |
| 4 | 1d: 11 ⟹ no operation | 0010 | 1111 0101 1 |
| | 2: Shift right Product | 0010 | 1111 1010 1 |

**FIGURE 3.11.3 Booth's algorithm with negative multiplier example.** The bits examined to determine the next step are circled in color.

Our example multiplies one bit at a time, but it is possible to generalize Booth's algorithm to generate multiple bits for faster multiplies (see Exercise 3.50)

Now that we have seen Booth's algorithm work, we are ready to see *why* it works for two's complement signed integers. Let $a$ be the multiplier and $b$ be the multiplicand and we'll use $a_i$ to refer to bit $i$ of $a$. Recasting Booth's algorithm in terms of the bit values of the multiplier yields this table:

| $a_i$ | $a_{i-1}$ | Operation |
|---|---|---|
| 0 | 0 | Do nothing |
| 0 | 1 | Add *b* |
| 1 | 0 | Subtract *b* |
| 1 | 1 | Do nothing |

Instead of representing Booth's algorithm in tabular form, we can represent it as the expression

$$(a_{i-1} - a_i)$$

where the value of the expression means the following actions:

$0$: do nothing
$+1$: add $b$
$-1$: subtract $b$

Since we know that shifting of the multiplicand left with respect to the Product register can be considered multiplying by a power of 2, Booth's algorithm can be written as the sum

$$\begin{array}{llllll}
 & & (a_{-1} & - a_0) & \times b & \times 2^0 \\
 & + & (a_0 & - a_1) & \times b & \times 2^1 \\
 & + & (a_1 & - a_2) & \times b & \times 2^2 \\
\dots & & \dots & & & \\
 & + & (a_{29} & - a_{30}) & \times b & \times 2^{30} \\
 & + & (a_{30} & - a_{31}) & \times b & \times 2^{31}
\end{array}$$

We can simplify this sum by noting that

$$-a_i \times 2^i + a_i \times 2^{i+1} = (-a_i + 2a_i) \times 2^i = (2a_i - a_i) \times 2^i = a_i \times 2^i$$

recalling that $a_{-1} = 0$ and by factoring out $b$ from each term:

$$b \times ((a_{31} \times -2^{31}) + (a_{30} \times 2^{30}) + (a_{29} \times 2^{29}) + \dots + (a_1 \times 2^1) + (a_0 \times 2^0))$$

The long formula in parentheses to the right of the first multiply operation is simply the two's complement representation of *a* (see page 163). Thus, the sum is further simplified to

$$b \times a$$

Hence, Booth's algorithm does in fact perform two's complement multiplication of *a* and *b*.

**3.23** [30] <§3.6> The original reason for Booth's algorithm was to reduce the number of operations by avoiding operations when there were strings of 0s and 1s. Revise the algorithm on page IMD 3.11-2 to look at 3 bits at a time and compute the product 2 bits at a time. Fill in the following table to determine the 2-bit Booth encoding:

| Current bits | | Previous bit | Operation | Reason |
|---|---|---|---|---|
| ai+1 | ai | ai–1 | | |
| 0 | 0 | 0 | | |
| 0 | 0 | 1 | | |
| 0 | 1 | 0 | | |
| 0 | 1 | 1 | | |
| 1 | 0 | 0 | | |
| 1 | 0 | 1 | | |
| 1 | 1 | 0 | | |
| 1 | 1 | 1 | | |

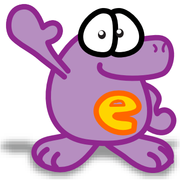
Assume that you have both the multiplicand and 2 × multiplicand already in registers. Explain the reason for the operation on each line, and show a 6-bit example that runs faster using this algorithm. (Hint: Try dividing to conquer; see what the operations would be in each of the eight cases in the table using a 2-bit Booth algorithm, and then optimize the pair of operations.)

# Floating-Point Numbers!

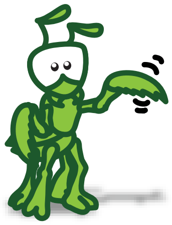
An IEEE 754 floating point number consists of three parts:

the Sign,

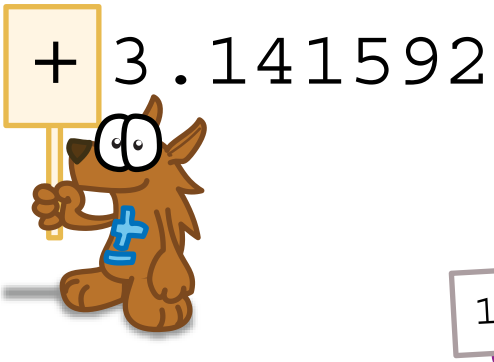
the Exponent,



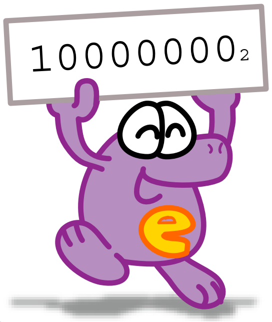
and the Mantissa.



(Also known as the Significand)

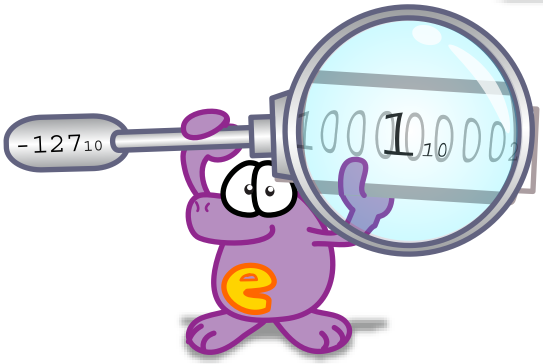


The Sign, as its name suggests, determines the sign of the number.

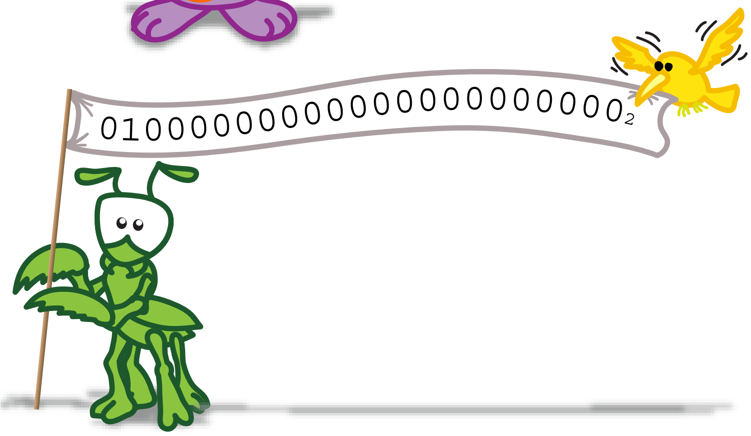


The Exponent plays a vital role in determining how big (or small) the number is. However, it's encoded so that unsigned comparison can be used to check floating-point numbers.

To see the true magnitude of the Exponent, you'd need to subtract the Bias, a special number determined by the length of the Exponent.

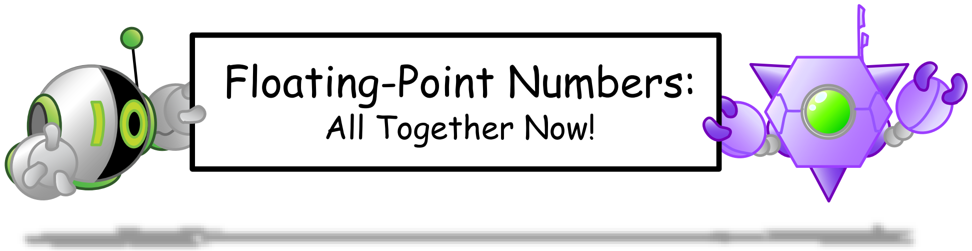


And last but not least, the Mantissa holds the significant digits of the floating point number.

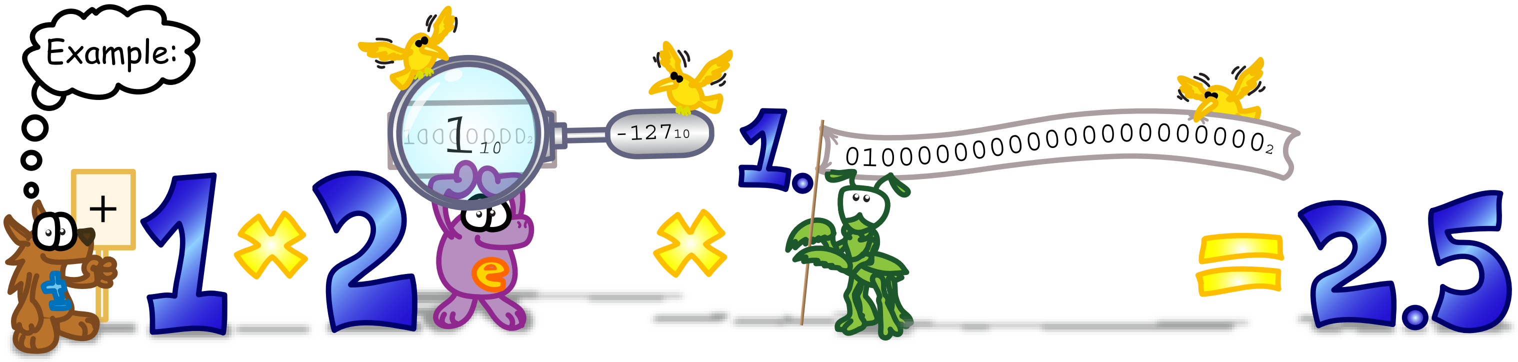


Ketrina Yim

# Floating-Point Numbers:
## All Together Now!

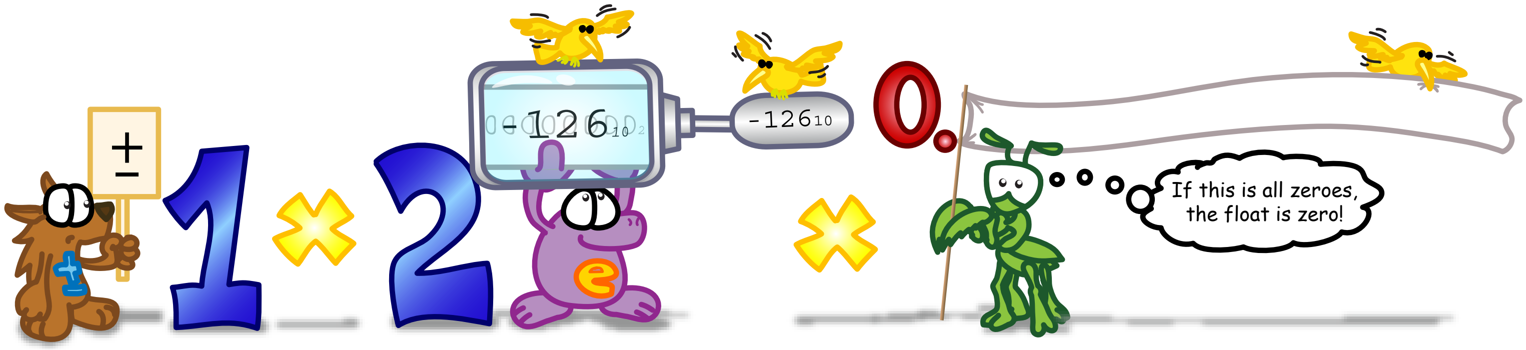


Once all the parts of the floating-point number are obtained, converting it to decimal is just a matter of applying the following formula:

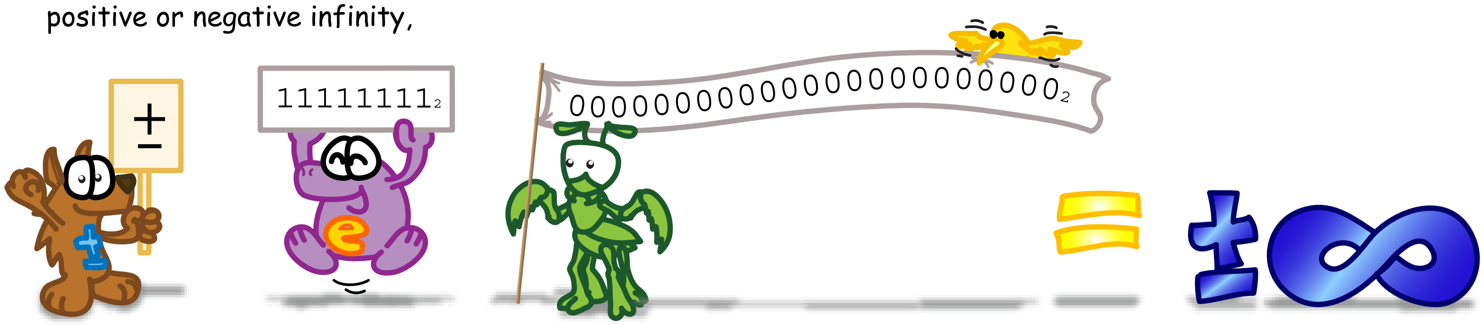


Notice that the Mantissa actually represents a fraction, instead of an integer!
In addition to representing real numbers, the IEEE 754 representation can also indicate...

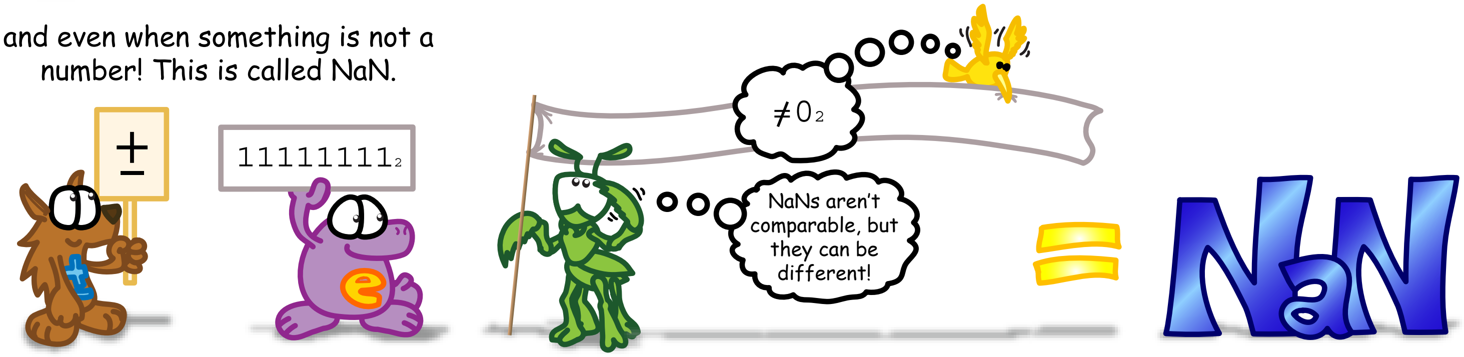
the set of numbers known as denormalized numbers (including zero),



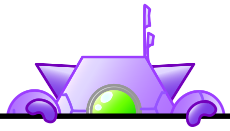
positive or negative infinity,



and even when something is not a number! This is called NaN.

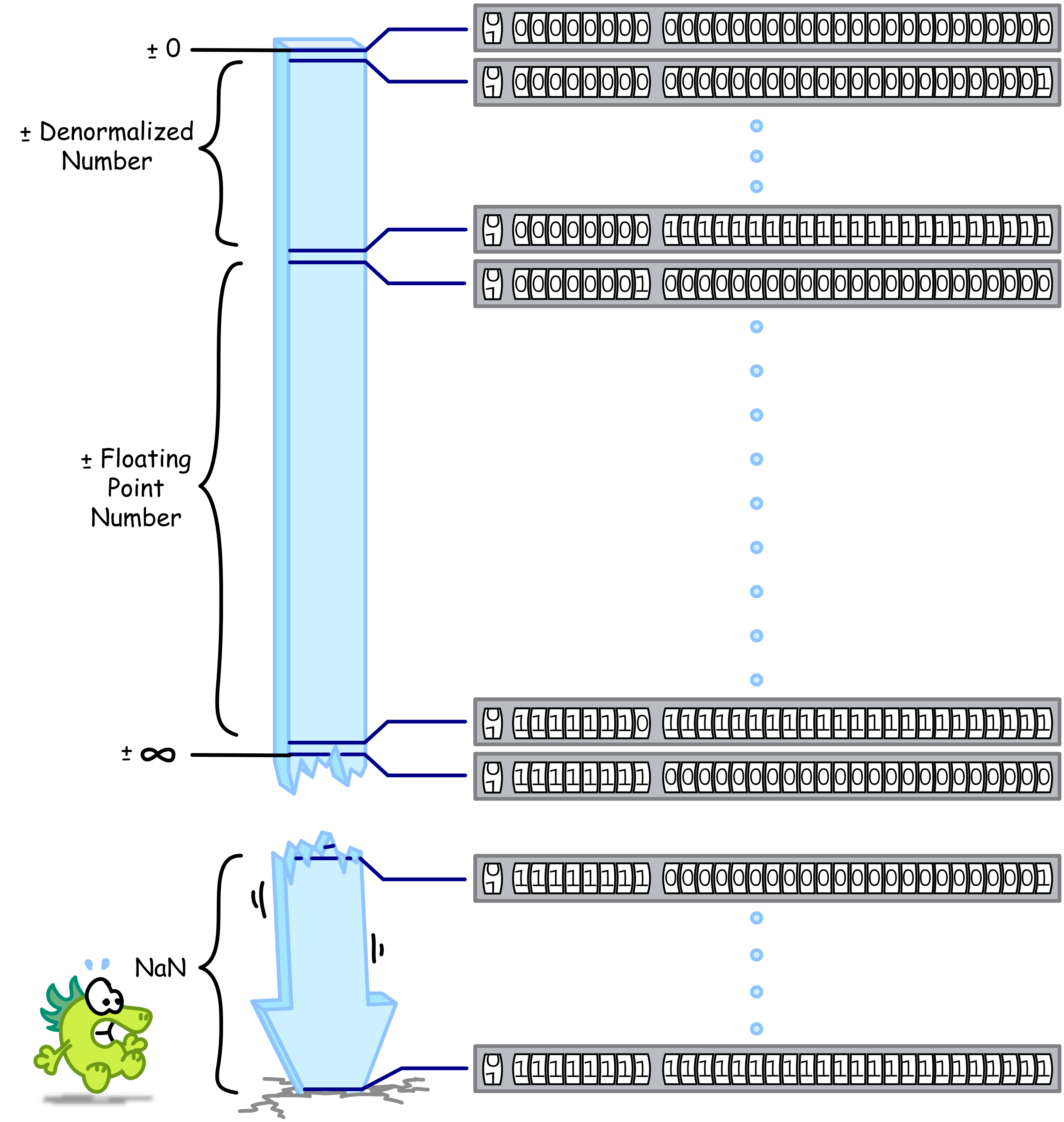


Ketrina Yim



# Floating-Point Numbers:
## The Great Number Line

Due to the format of the IEEE-754 standard, the floating-point numbers can be plotted on a number line. In fact, the floating-point numbers are arranged so that they can be incremented like a binary odometer!



Ketrina Yim