FU05 Computer Architecture

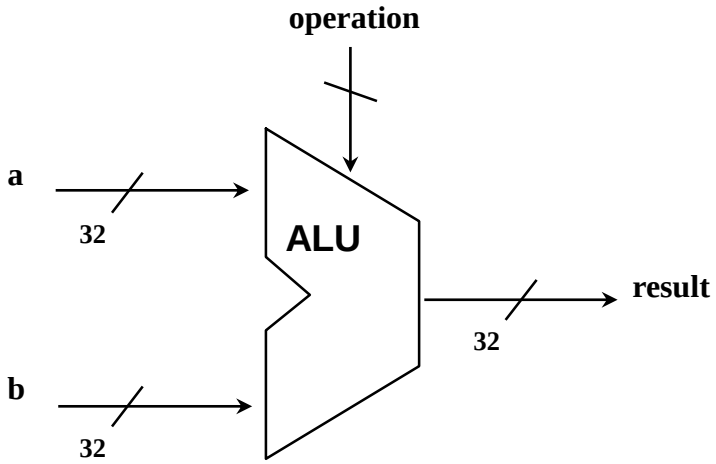
6. Arithmetic 2 (演算回路2)

Ben Abdallah Abderazek
E-mail: benab@u-aizu.ac.jp

Lectures of this course are adopted from the slides of the Book ‘’Computer Organization and Design - The Hardware/Software Interface, 5th Ed’’ by David. A. Patterson & John L. Hennessy

Review: Arithmetic

- So far, we have seen:
  - Performance (seconds, cycles, instructions)
  - Instruction Set Architecture
  - Assembly Language and Machine Language
- What's next:
  - Implementing the Architecture

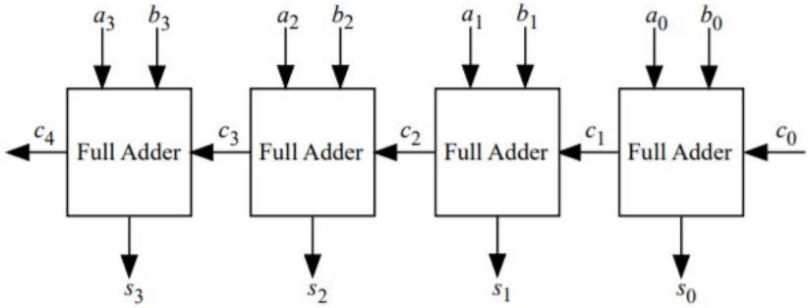


Contents

- Faster Addition
- Multiplication
- Division
- Floating Points (next lecture)

Ripple Carry Adder (RCA)

- Is a 32-bit ALU as fast as a 1-bit ALU?
- Is there more than one way to do addition?
  - Two extremes: ripple carry and sum-of-products
  - How many logic layers do we need for these two extremes?



4-bit full adder

- The sum of position 1 cannot complete until it receives the carry in (c1) from the sum in position 0.
- In this way, the carry ''ripples'' through the circuit from right to left.
- This configuration is known as **Ripple Carry Adder (RCA)**

Can you see the ripple? How could you get rid of it?

$$c_1 = b_0c_0 + a_0c_0 + a_0b_0$$

$$c_2 = b_1c_1 + a_1c_1 + a_1b_1 \qquad c_2 =$$

$$c_3 = b_2c_2 + a_2c_2 + a_2b_2 \qquad c_3 =$$

$$c_4 = b_3c_3 + a_3c_3 + a_3b_3 \qquad c_4 =$$

… …

Carry-lookahead Adder (CLA) - 1

- A carry look ahead adder contains circuitry that determines whether the previous adder stages produce a carry. This circuitry produces the ‘’carry’’ in for each stage without having to wait for the carry to ripple through the prior stage.

- We want to create look ahead circuits that are only dependent of the system inputs as opposed to the intermediate carry out signals. This will eliminate the ripple delay.

Carry-lookahead Adder (CLA) - 2

Cin	a	b	Cout
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	1

generate

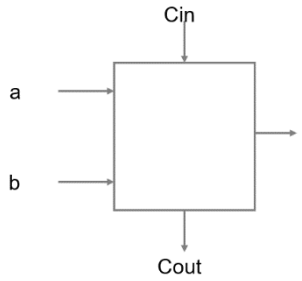
propagate

For the input codes where Cin = 0, the full adder ‘’generates” a new carry when a=1, b=1. This can be described with the expression : $g_i = a_i b_i$

For the input codes where Cin = 1, the full adder ‘’propagates” the incoming carry when either a=1 or b=1. This behavior can be described with the expression : $p_i = a_i + b_i$

The entire expression for the carry out can be written as:

Cout = g + p.Cin
Cout = a.b + (a+b). Cin



$g_i = a_i bi$
$p_i = a_i + b_i$

Carry-lookahead Adder (CLA)- 3

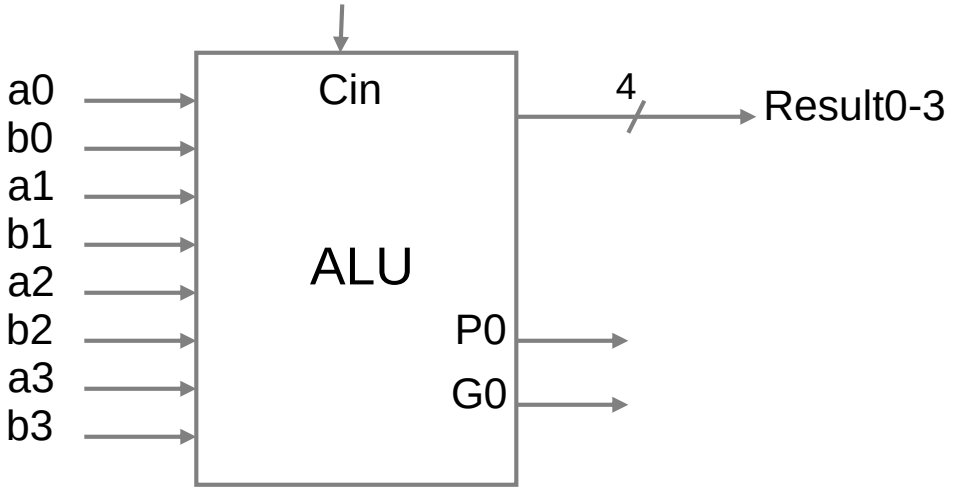
- Did we get rid of the ripple?

$C_{i+1} = g_i + p_i c_i$

$g_i = a_i b_i$
$p_i = a_i + b_i$

$c_1 = g_0 + p_0 c_0$
$c_2 = g_1 + p_1 c_1$ $\quad c_2 = g_1 + p_1(g_0 + p_0 c_0)$
$c_3 = g_2 + p_2 c_2$ $\quad c_3 = g2 + p2c2 = g2+p2(g1+p1g0+p1p0c0)$
$c_4 = g_3 + p_3 c_3$ $\quad c_4 = g3 + p3c3 = g3 + p3(g2+p2g1+p2p1g0+p2p1p0c0)$
$= g3 + p3g2 + p3p2g1 + p3p2p1g0 + p3p2p1p0c0$

- Feasible ?



$P0 = p_0.p_1.p_2.p_3$
$G0 = g_3+(p_3.g_2)+(p_3.p_2.g_1)+(p_3.p_2.p_1.g_0)$

16-bit adder configuration with 4-bit adder

4bit 加算器による16bit 加算器構成

$P_0=p_3p_2p_1p_0$

$P_1=p_7p_6p_5p_4$

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

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

$G_0 = g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0$

$G_1 = g_7 + p_7g_6 + p_7p_6g_5 + p_7p_6p_5g_4$

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

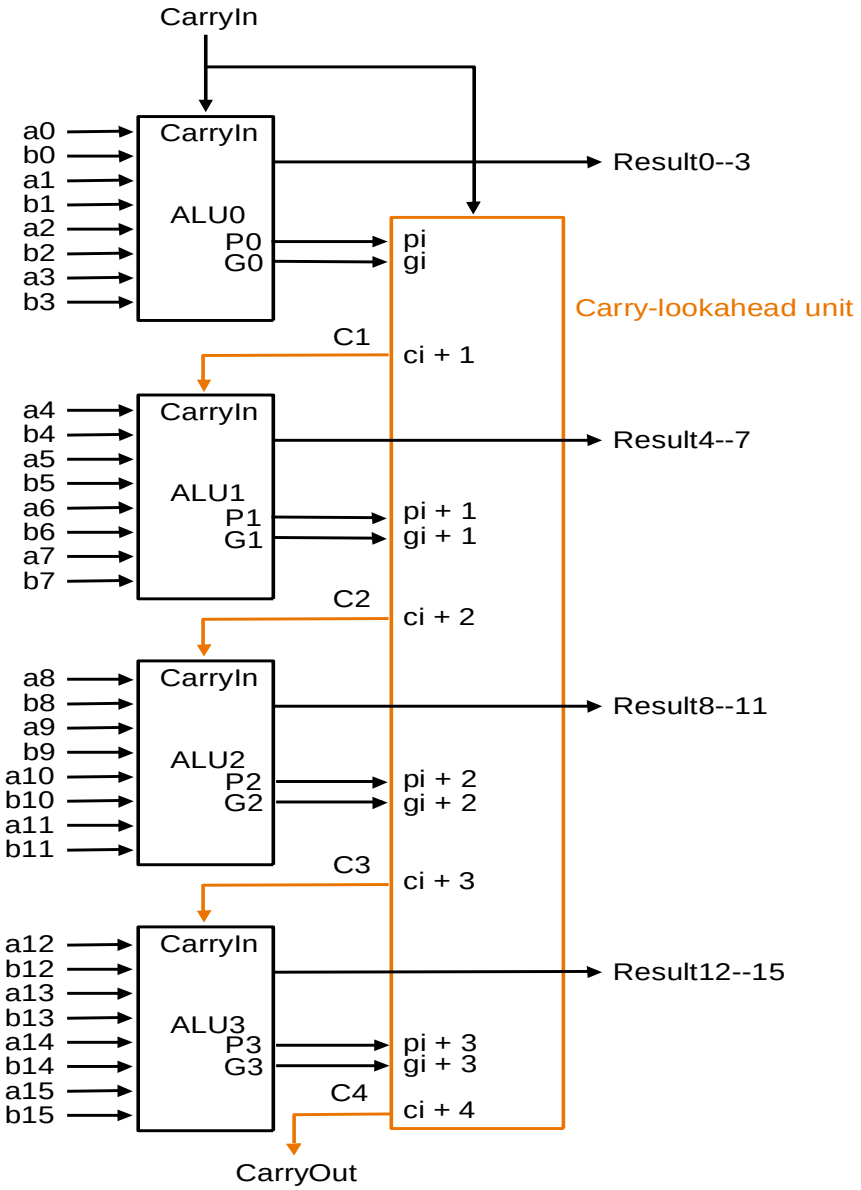
$G_3 = g_{15} + p_{15}g_{14} + p_{15}p_{14}g_{13} + p_{15}p_{14}p_{13}g_{12}$

$C_1 = G_0 + P_0c_0$

$C_2 = G_1 + P_1G_0 + P_1P_0c_0$

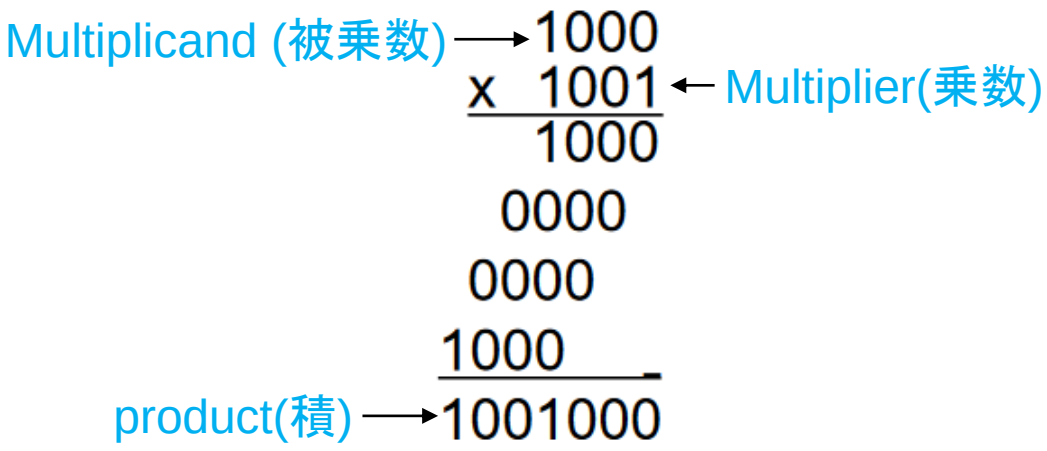
$C_3 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0c_0$

$C_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0c_0$



Multiplication (1)

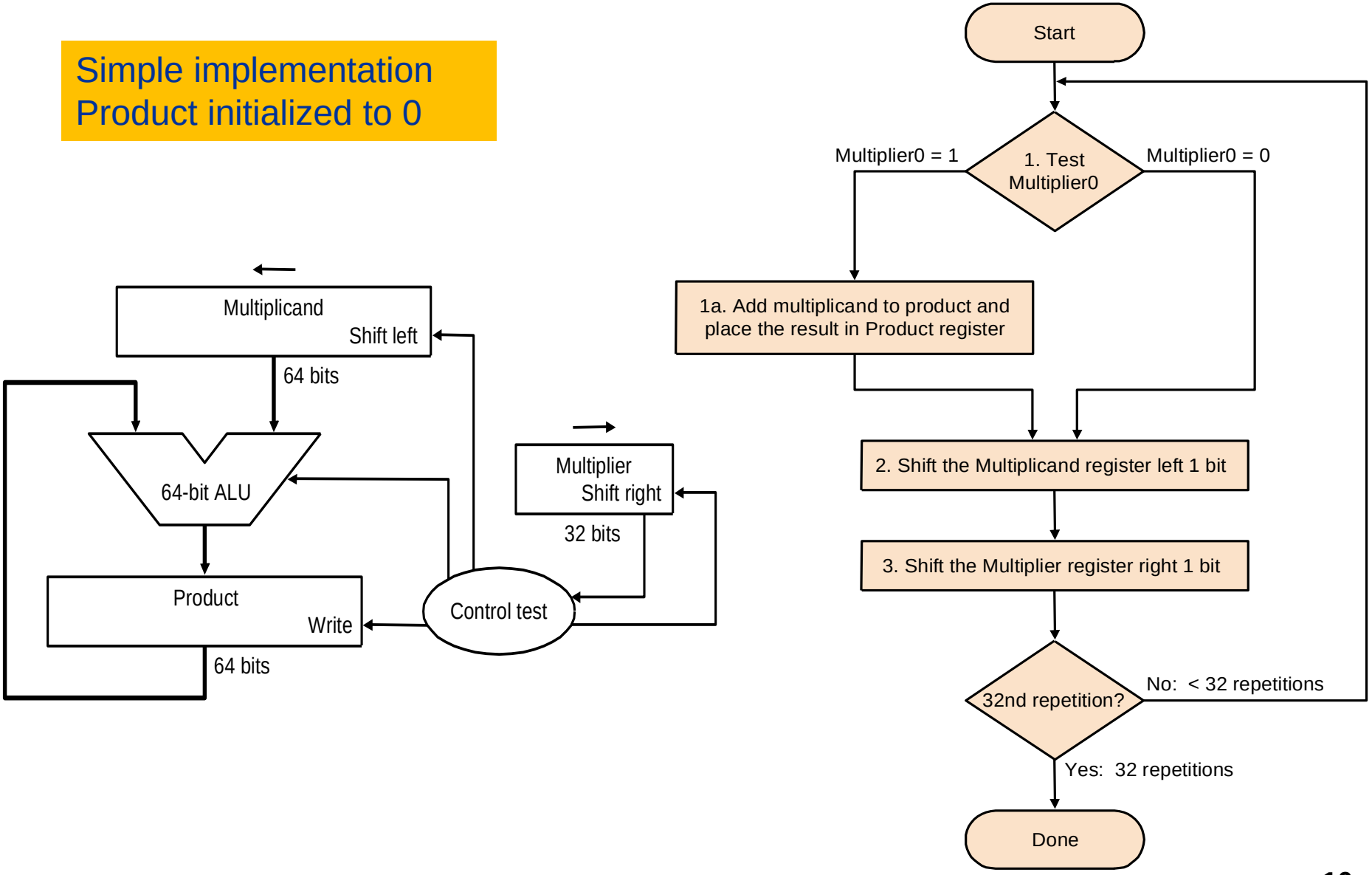
- More complicated than addition
  - accomplished via shifting and addition
- More time and more area
- Let's look at 3 versions based on gradeschool algorithm



- Negative numbers: convert and multiply
  - there are better techniques, we won't look at them now

Multiplication: Implementation Version 1

Simple implementation
Product initialized to 0



Multiplication: Implementation Version 1

Example

Using 4-bit numbers, multiply 2_{ten} 3_{ten}, or 0010_{two} 0011_{two}.

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

FIGURE 3.6 Multiply example using algorithm in Figure 3.4. The bit examined to determine the next step is circled in color.

Multiplication: Improved version

- Product = HI and LO registers, HI =0
- Product is shifted right
- Rediced 32-bit Multiplicand & Adder

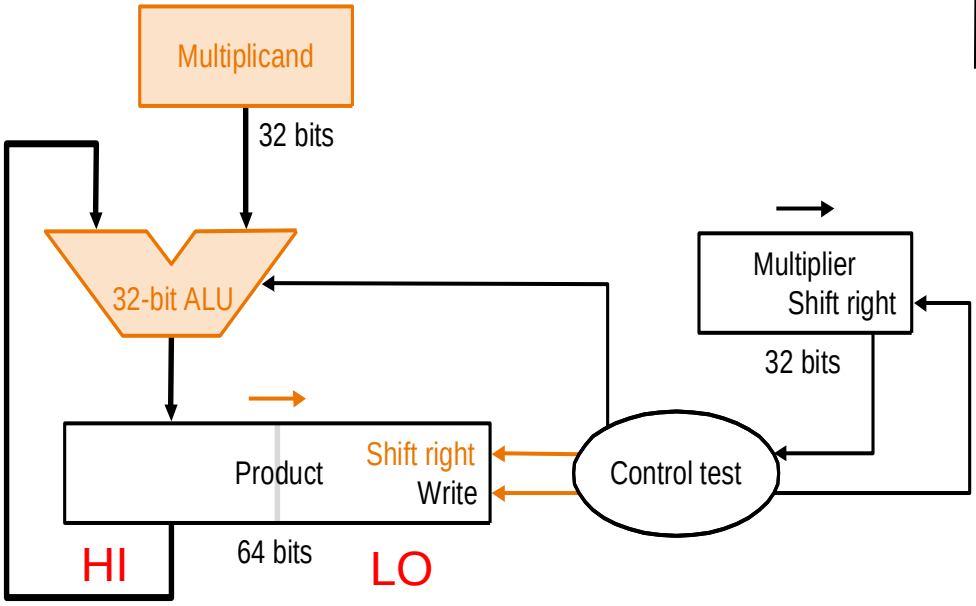
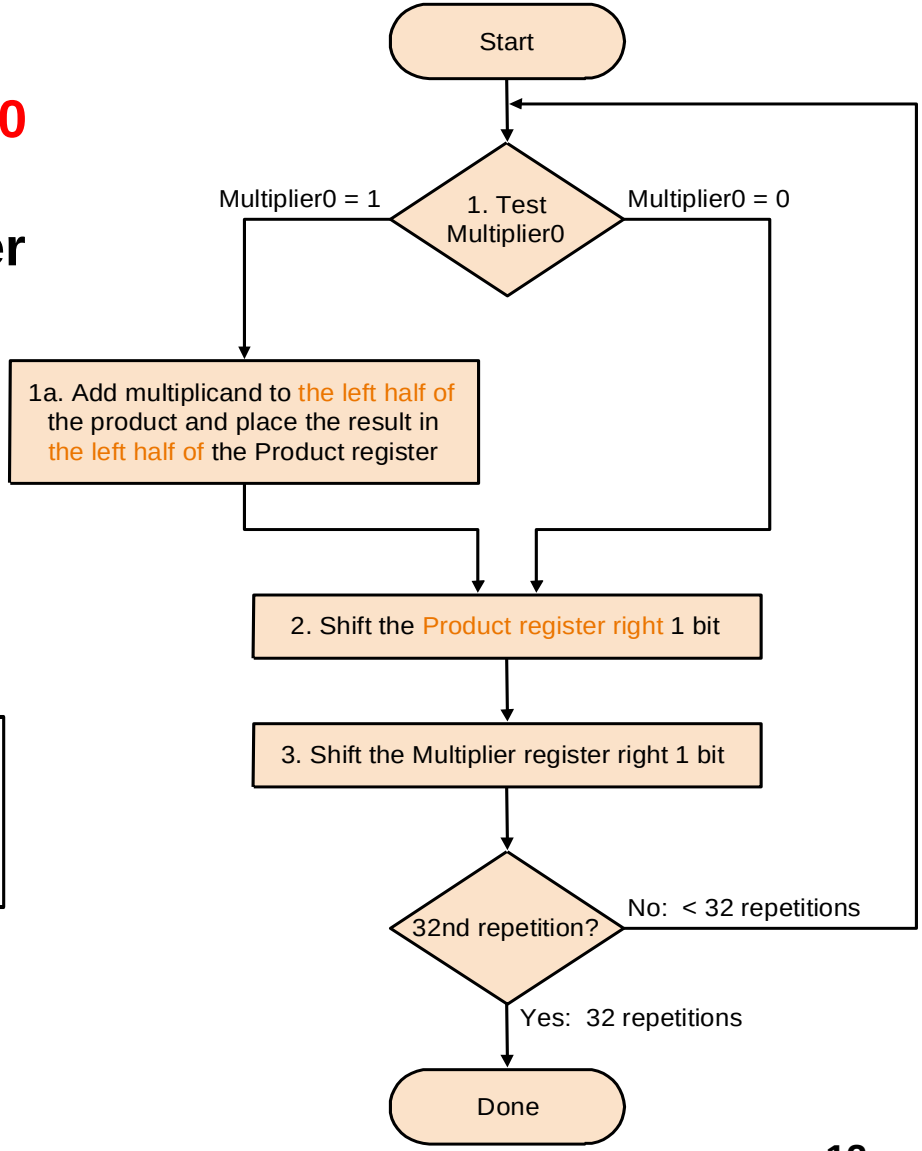


Fig. 3.5



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

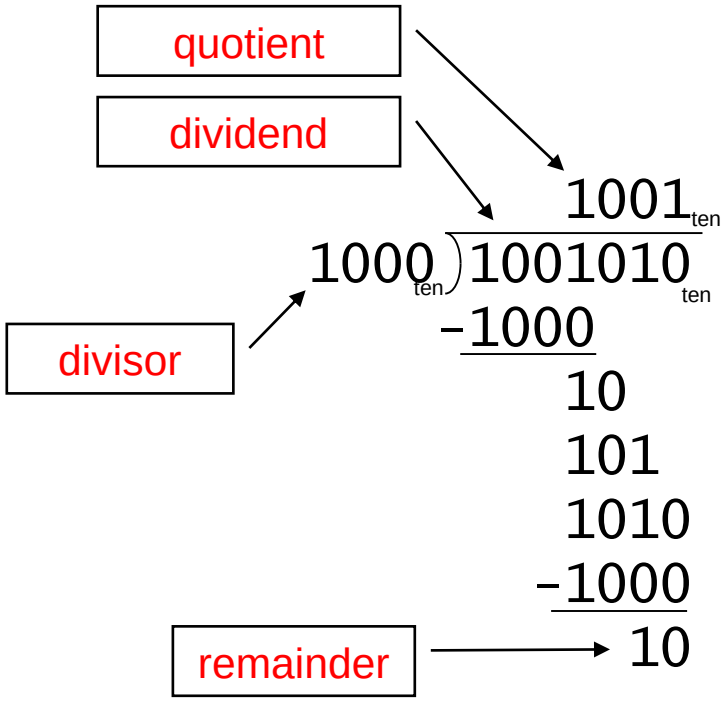
Division (除算)

- Dividend (被除数）
- Divisor(除数）
- Quotient(商）
- Remainder(剩余）

Dividend **=** Quotient **x** Divisor **+** Remainder

- Division is similar to multiplication: repeated subtract

Division: Paper & Pencil



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

- Observation:
- Three registers
  - Dividend: 64 bits
  - Divisor: 64 bits, shift right
  - Quotient: 32 bits, shift left
  - Remainder: ?

Dividend = Quotient x Divisor + Remainder

Division Hardware

- Takes 33 steps for 32-bit Quotient & Rem.

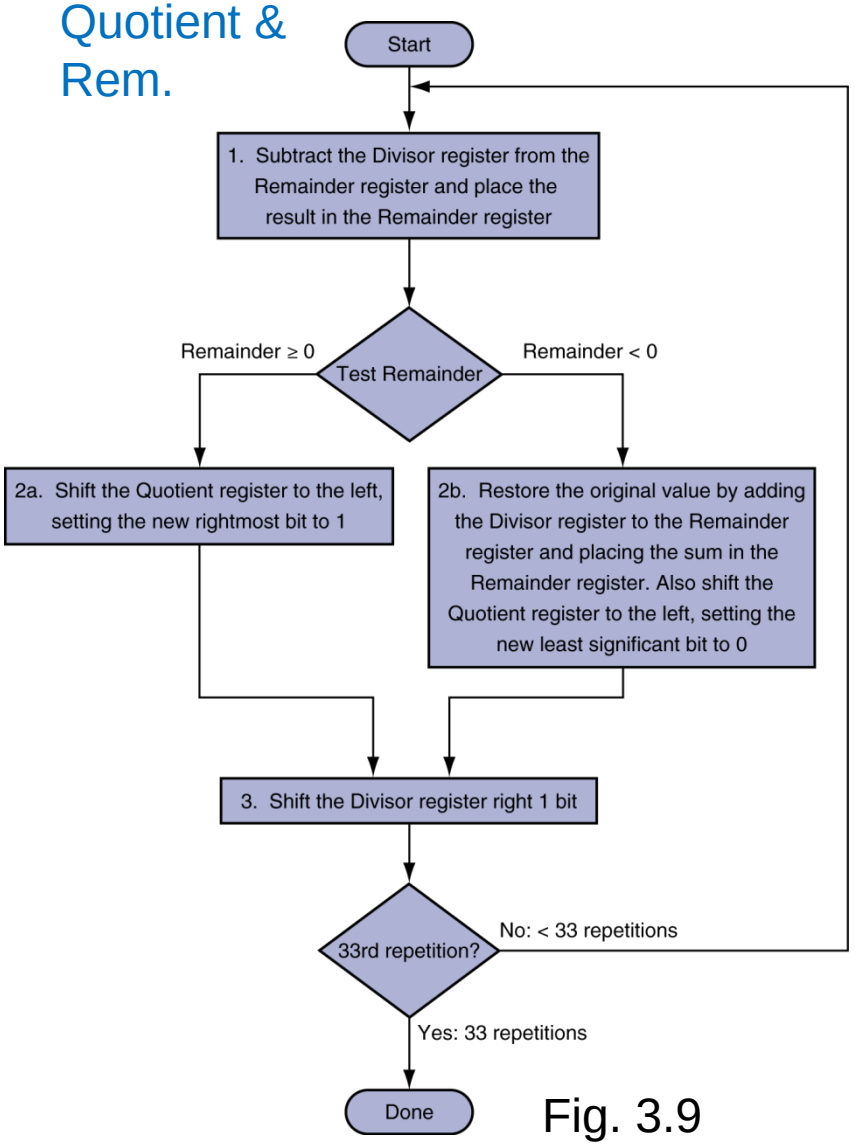


Fig. 3.9

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

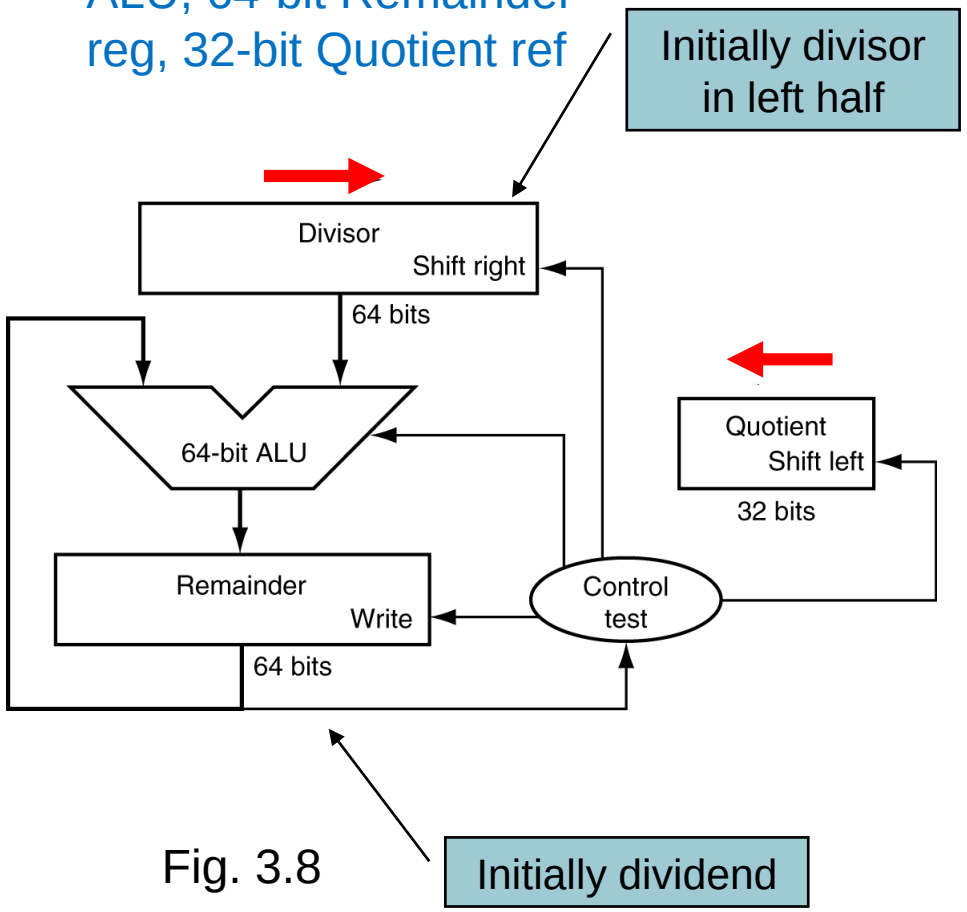


Fig. 3.8

Division- Example

Using a 4-bit version of the algorithm, let's try dividing 7ten by 2ten, or 0000 0111two by 0010two

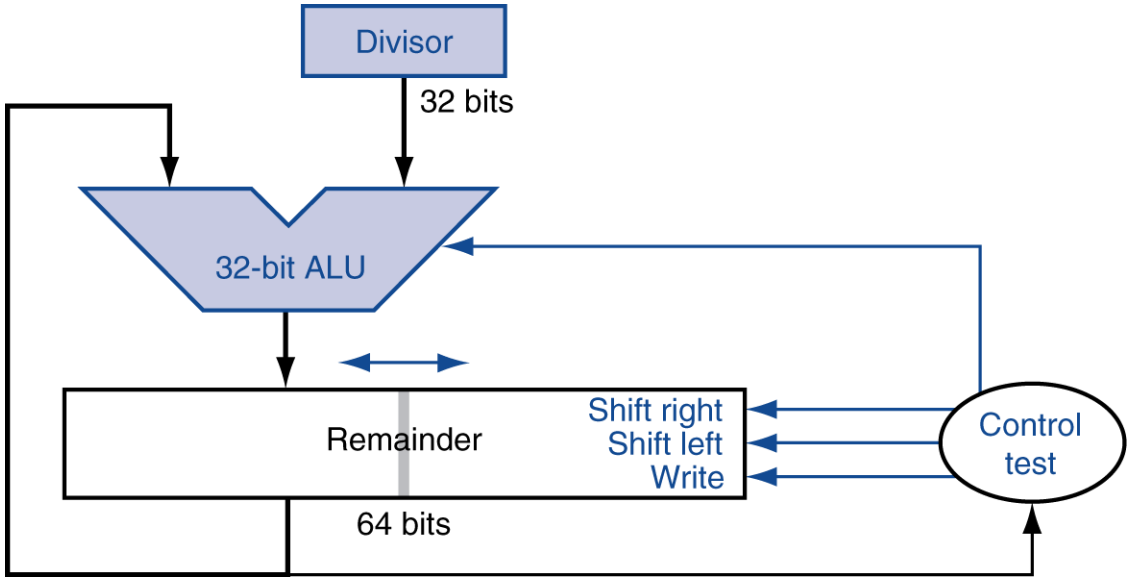
1. The 4-bit Divisor is put in the left most four bits
2. The 4-bit Dividend is put in the right most four bit of the remainder

Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem – Div	0000	0010 0000	(1)110 0111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem – Div	0000	0001 0000	(1)111 0111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0001 0000	0000 0111
	3: Shift Div right	0000	0000 1000	0000 0111
3	1: Rem = Rem – Div	0000	0000 1000	(1)111 1111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0000 1000	0000 0111
	3: Shift Div right	0000	0000 0100	0000 0111
4	1: Rem = Rem – Div	0000	0000 0100	(0)000 0011
	2a: Rem ≥ 0 ⟹ sll Q, Q0 = 1	0001	0000 0100	0000 0011
	3: Shift Div right	0001	0000 0010	0000 0011
5	1: Rem = Rem – Div	0001	0000 0010	(0)000 0001
	2a: Rem ≥ 0 ⟹ sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

FIGURE 3.10 Division example using the algorithm in Figure 3.9. The bit examined to determine the next step is circled in color.

Optimized Divider

- One cycle per partial-remainder subtraction
- Looks a lot like a multiplier!
  - Same hardware can be used for both



(Fig. 3.11)

- The Divisor register, ALU, and quotient register are all 32 bits wide, with only the Remainder register left at 64 bits.
- Compared to Figure 3.8, the ALU and Divisor registers are halved and the remainder is shifted left .
- This version also combines the Quotient register with the right half of the Remainder register.

Faster Division

- Can’t use parallel hardware as in multiplier
  - Subtraction is conditional on sign of remainder
- Faster dividers (e.g. SRT devision) generate multiple quotient bits per step
  - Still require multiple steps

MIPS Division

- Use HI/LO registers for result
  - HI: 32-bit remainder (HI = remainder of (rs ÷ rt)
  - LO: 32-bit quotient (LO = quotient of (rs ÷ rt))
- 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
    - for example, mfhi $a0

Example

Instruction	*Example*	*Meaning*	*Comments*
add	add S1,S2,S3	S1 = S2 + S3	3 operands; exception possible
subtract	sub S1,S2,S3	S1 = S2 – S3	3 operands; exception possible
add immediate	addi S1,S2,100	S1 = S2 + 100	+ constant; exception possible
add unsigned	addu S1,S2,S3	S1 = S2 + S3	3 operands; no exceptions
subtract unsigned	subu S1,S2,S3	S1 = S2 – S3	3 operands; no exceptions
add imm. unsign.	addiu S1,S2,100		S1 = S2 + 100; + constant; no exceptions
multiply	mult S2,S3	Hi, Lo = S2 x S3	64-bit signed product
multiply unsigned	multuS2,S3	Hi, Lo = S2 x S3	64-bit unsigned product
divide	div S2,S3	Lo = S2 ÷ S3,	Lo = quotient, Hi = remainder Hi = S2 mod S3
divide unsigned remainder	divu S2,S3	Lo = S2 ÷ S3,	Unsigned quotient & Hi = S2 mod S3
Move from Hi	mfhi S1	S1 = Hi	Used to get copy of Hi
Move from Lo	mflo S1	S1 = Lo	Used to get copy of Lo

Quiz

1. This problem covers 4-bit binary multiplication. Fill in the table for the Product, Multplier and Multiplicand for each step. You need to provide the DESCRIPTION of the step being performed (shift left, shift right, add, no add). The value of M (Multiplicand) is 1011, Q (Multiplier) is initially 1010.

Product	Multiplicand	Multiplier	Description	Step
0000 0000	0000 1011	1010	Initial Values	Step 0
				Step 1
				Step 2
				Step 3
				Step 4
				Step 5
				Step 6
				Step 7
				Step 8
				Step 9
				Step 10
				Step 11
				Step 12
				Step 13
				Step 14
				Step 15

Solution

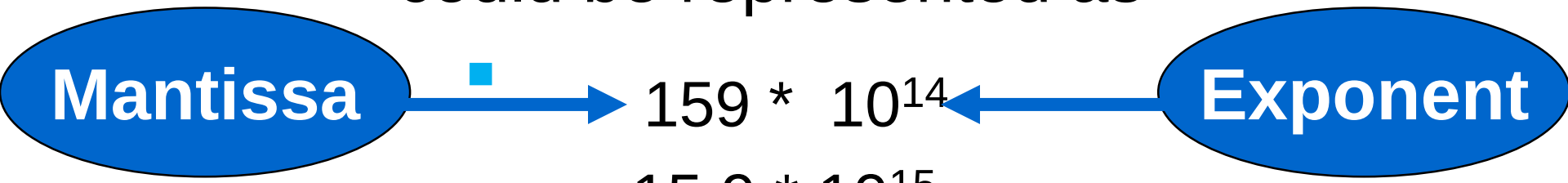
1. This problem covers 4-bit binary multiplication. Fill in the table for the Product, Multplier and Multiplicand for each step. You need to provide the DESCRIPTION of the step being performed (shift left, shift right, add, no add). The value of M (Multiplicand) is 1011, Q (Multiplier) is initially 1010.

Product	Multiplicand	Multiplier	Description	Step
0000 0000	0000 1011	1010	Initial Values	Step 0
0000 0000	0000 1011	1010	0 => No add	Step 1
0000 0000	0001 0110	1010	Shift left M	Step 2
0000 0000	0001 0110	0101	Shit right Q	Step 3
0001 0110	0001 0110	0101	1 => Add M to Product	Step 4
0001 0110	0010 1100	0101	Shift left M	Step 5
0001 0110	0010 1100	0010	Shift right Q	Step 6
0001 0110	0010 1100	0010	0 => No add	Step 7
0001 0110	0101 1000	0010	Shift left M	Step 8
0001 0110	0101 1000	0001	Shift right Q	Step 9
0110 1110	0101 1000	0001	1 => Add M to Product	Step 10
0110 1110	1011 0000	0001	Shift left M	Step 11
0110 1110	1011 0000	0000	Shift right Q	Step 12
0110 1110	1011 0000	0000	0 => No add	Step 13
0110 1110	0110 0000	0000	Shift left M	Step 14
0110 1110	0110 0000	0000	Shift Right Q	Step 15

Floating Point

159000000000000

could be represented as



$159 * 10^{14}$

$15.9 * 10^{15}$

$1.59 * 10^{16}$

A calculator might display 159 E14

Binary

The value of real binary numbers…

Scientific	2^2	2^1	2^0
Fractions			
Decimal	4	2	1
	1	**0**	**1**

Binary Fractions

The value of real binary numbers…

Scientific	2^2	2^1	2^0	.	2^{-1}	2^{-2}	2^{-3}
Fractions				.	½	¼	⅛
Decimal	4	2	1	.	.5	.25	.125

1 0 1 . 1 0 1

Binary Fractions

The value of real binary numbers…

Scientific	2^2	2^1	2^0	.	2^{-1}	2^{-2}	2^{-3}
Fractions				.	½	¼	⅛
Decimal	4	2	1	.	.5	.25	.125

1 0 1 . 1 0 1

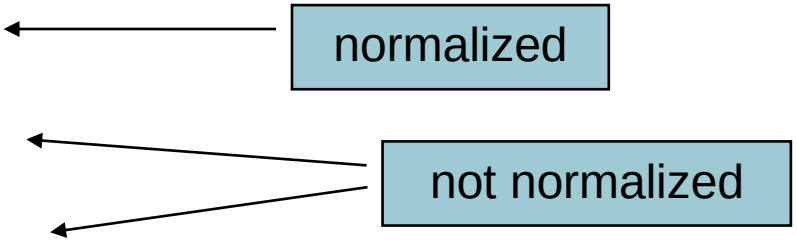
101.101 = 4+1+1/2+1/8

= 4+1+.5+.125= 5.625

= 5 ⅝

Floating Point

- Representation for non-integral numbers
  - Including very small and very large numbers
- Like scientific notation
  - -2.34×10^{56}
  - $+0.002 \times 10^{-4}$
  - $+987.02 \times 10^{9}$

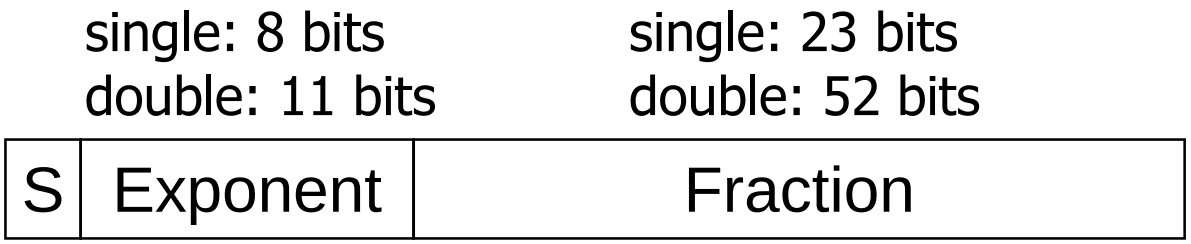


- In binary
  - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$
- Types `float` and `double` in C

Floating Point Standard

- Defined by IEEE Std 754-1985
- Developed in response to divergence of representations
  - Portability issues for scientific code
- Now almost universally adopted
- Two representations
  - Single precision (32-bit)
  - Double precision (64-bit)

IEEE Floating-Point Format



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

- S: sign bit (0 ⇒ non-negative, 1 ⇒ negative)
- Normalize significand: 1.0 ≤ |significand| < 2.0
  - Always has a leading pre-binary-point 1 bit, so no need to represent it explicitly (hidden bit)
  - Significand is Fraction with the “1.” restored
- Exponent: excess representation: actual exponent + Bias
  - Ensures exponent is unsigned
  - Single: Bias = 127; Double: Bias = 1203

Floating-Point Example

- Represent –0.75
  - $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
  - S = 1
  - Fraction = $1000\ldots00_2$
  - Exponent = –1 + Bias
    - Single: –1 + 127 = 126 = 01111110_2
    - Double: –1 + 1023 = 1022 = 01111111110_2
- Single: 10111111101000…00
- Double: 10111111111101000…00

Floating-Point Example

- What number is represented by the single-precision float

  1100000010100 0…00

  - S = 1
  - Fraction = $01000\ldots00_2$
  - Fxponent = $10000001_2 = 129$
- $x = (-1)^1 \times (1 + 01_2) \times 2^{(129 - 127)}$

  $= (-1) \times 1.25 \times 2^2$

  $= -5.0$

Exercise

Ex. This problem covers floating-point IEEE format.

(a) List four floating-point operations that cause NaN to be created?

(b) Assuming single precision IEEE 754 format, what decimal number is represent by this word:

1 01111101 00100000000000000000000

(Hint: remember to use the biased form of the exponent.)

Exercise Solution

Ex. This problem covers floating-point IEEE format.

(a) List four floating-point operations that cause NaN to be created?

Four operations that cause Nan to be created are as follows:

(1) Divide 0 by 0

(2) Multiply 0 by infinity

(3) Any floating point operation involving Nan

(4) Adding infinity to negative infinity

(b) Assuming single precision IEEE 754 format, what decimal number is represent by this word:

1 01111101 00100000000000000000000

(Hint: remember to use the biased form of the exponent.)

The decimal number

= (+1)* (2^(125-127))*(1.001)2

= (+1)*(0.25)*(0.125)

= 0.03125