Chapter 3
Arithmetic for computers

Computer Architecture and Organization

School of CSEE

Arithmetic for Computers

- Operations on integers
  - Addition and subtraction
  - Multiplication and division
  - Dealing with overflow
- Floating-point real numbers
  - Representation and operations

1. Addition and Subtraction
2. Multiplication
3. Division
4. Floating point number arithmetic

Integer Addition

- **Example: 7 + 6**

		(0)		(0)		(1)		(1)		(0)			(Carries)
. . .		0		0		0		1		1		1	
. . .		0		0		0		1		1		0	
. . .	(0)	0	(0)	0	(0)	1	(1)	1	(1)	0	(0)	1	

- Overflow if result out of range
  - Adding +ve and –ve operands, no overflow
  - Adding two +ve operands
    - Overflow if result sign is 1
  - Adding two –ve operands
    - Overflow if result sign is 0

Integer Subtraction

- Add negation of second operand
- Example: 7 – 6 = 7 + (–6)

+7:	0000 0000 … 0000 0111
–6:	1111 1111 … 1111 1010
+1:	0000 0000 … 0000 0001

- Overflow if result out of range
  - Subtracting two +ve or two –ve operands, no overflow
  - Subtracting +ve from –ve operand
    - Overflow if result sign is 0
  - Subtracting –ve from +ve operand
    - Overflow if result sign is 1

Detecting Overflow

- 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

Dealing with Overflow

- **Some languages (e.g., C) ignore overflow**
  - **MIPS addu (add unsigned), addiu, subu (subtract unsigned) instructions**

- **Other languages (e.g., Ada, Fortran) require raising an exception**
  - **On overflow, invoke exception handler**

Exercise

1. **Build the logic circuit that checks overflow.**

1. Addition and Subtraction
2. **Multiplication**
3. Division
4. Floating point number arithmetic

Introduction

- In many cases ALU does not have capability of multiplying two positive integers.
- In this section we will find out how to multiply two integers with just ‘add’ and ‘shift’ operation.
- Also multiplying negative integers will be discussed.

Multiplication

- **We can build multiplier as discussed in logic design course.**

Ex) Multiplying 3-bit numbers : a2 a1 a0 X b2 b1 b0

construct truth table

a2	a1	a0	b2	b1	b0	c5	c4	c3	c2	c1	c0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	1	0	0	0	0	0	0
. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .
0	1	1	0	0	0	0	0	0	0	0	0
0	1	1	0	0	1	0	0	0	0	1	1
. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .
1	1	1	1	1	0	1	0	1	0	1	0
1	1	1	1	1	1	1	1	0	0	0	1

Multiplication

- **We can multiply two numbers with adder and shift registers.**
- **Let's think about the multiplication algorithm in grade school.**

Ex)　　312 X 203

```
    312
  X 203
  -----
    936
   000
  624
  -----
  63336
```

- **If we multiply binary numbers, only 'add' and 'shift' operation is necessary**

Multiplication

- **More complicated than addition**
  - **accomplished via shifting and addition**
- **Multiplying two binary numbers**

```
      0010    (multiplicand)
    x 1001    (multiplier)
    ------
      0010
     0000
    0000
   0010
  -------
  0010010     (product)
```

- **Check multiplier bit**

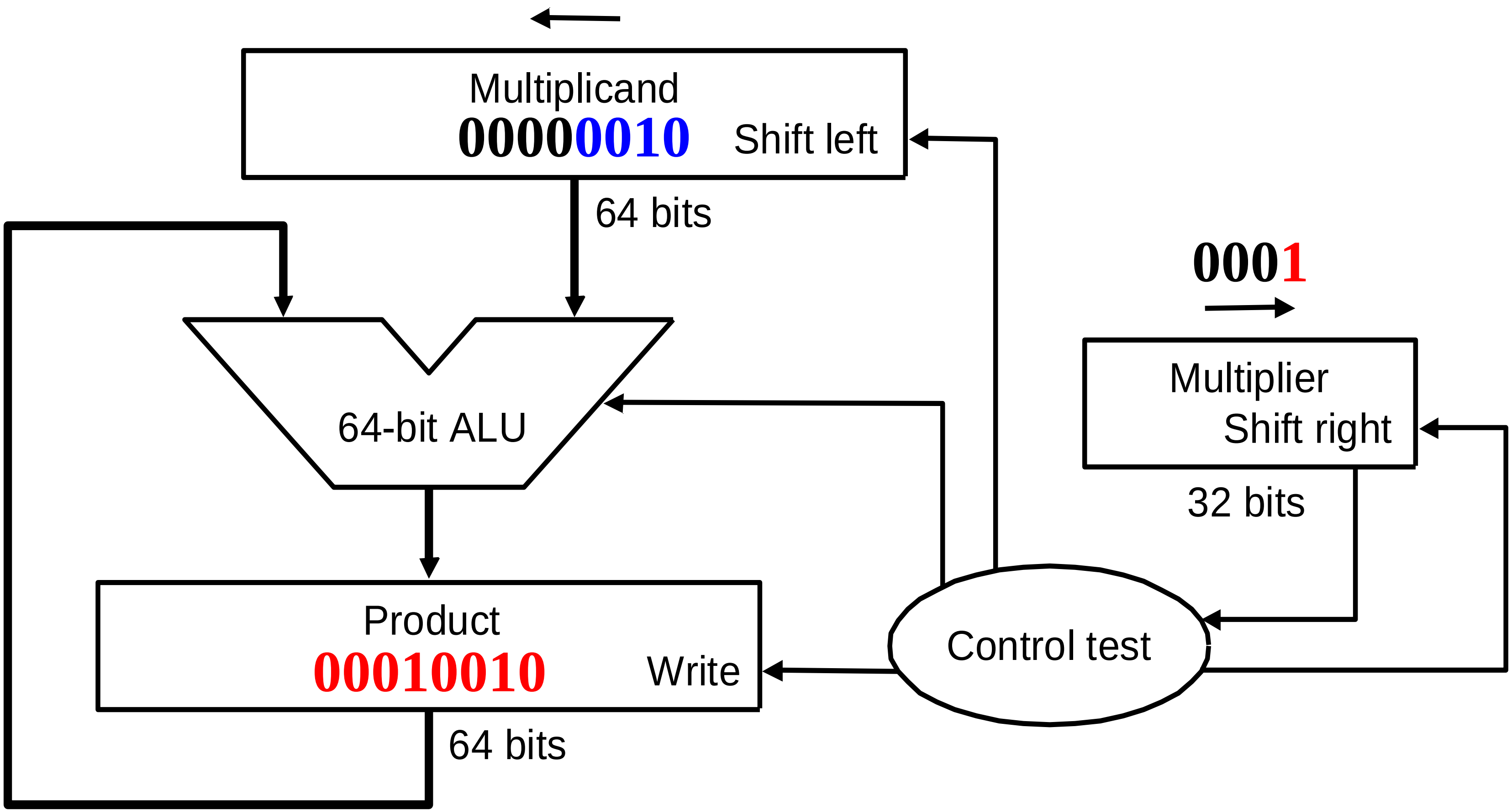
  if 0, don't add, shift

  if 1, add multiplicand and shift

Multiplication

$$
\begin{array}{rl}
0010 & \text{(multiplicand)} \\
\times\ 1001 & \text{(multiplier)} \\
\hline
0000\boxed{0010} & \\
000\boxed{0000}0 & \\
00\boxed{0000}00 & \\
0\boxed{0010}000 & \\
\hline
00010010 & \text{(product)}
\end{array}
$$

Multiplication: Implementation



Multiplication: Implementation

- **Version 1**

  Multiplicand : Shift Left

  Product : doesn't move

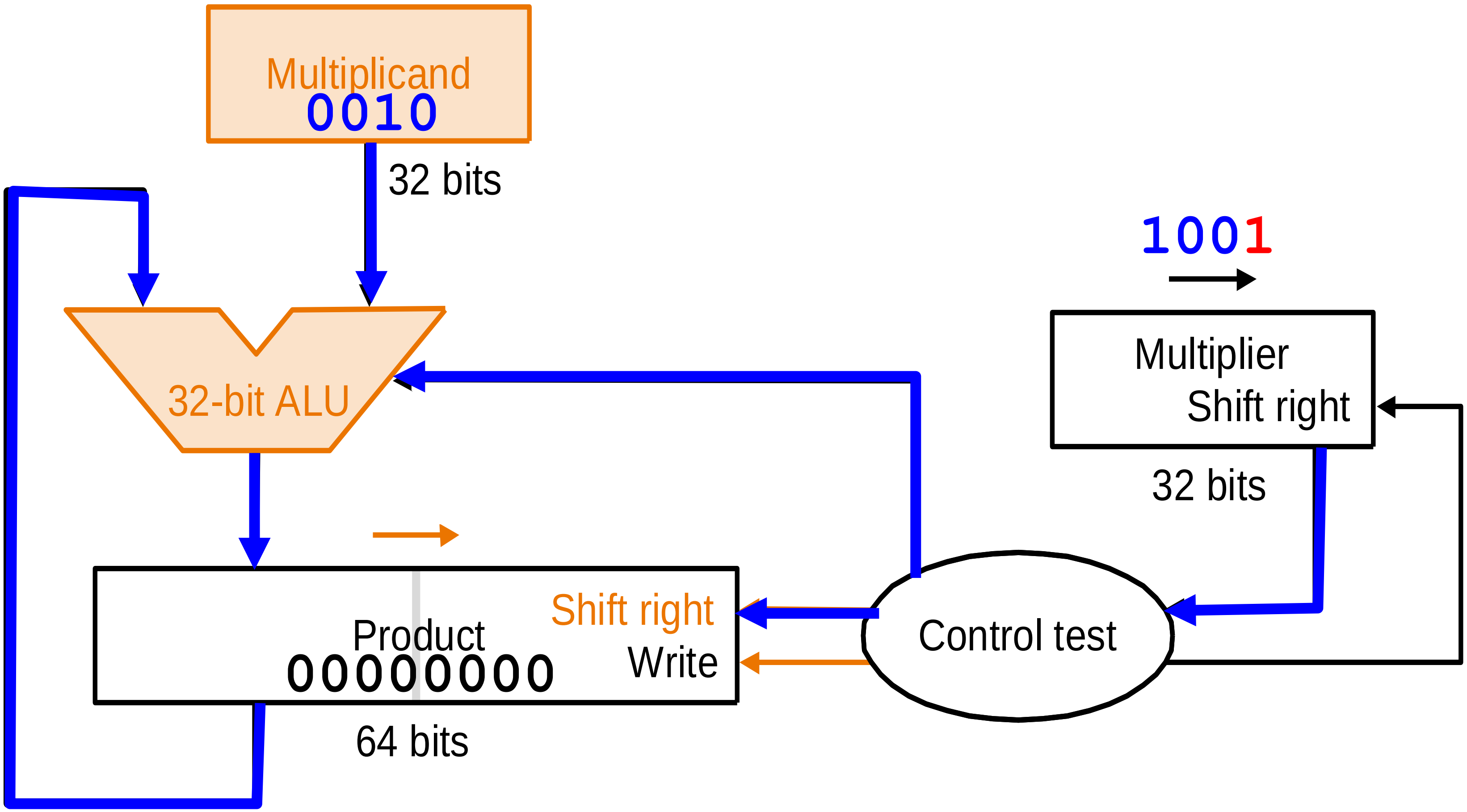
- **Version 2**

  Multiplicand : doesn't move

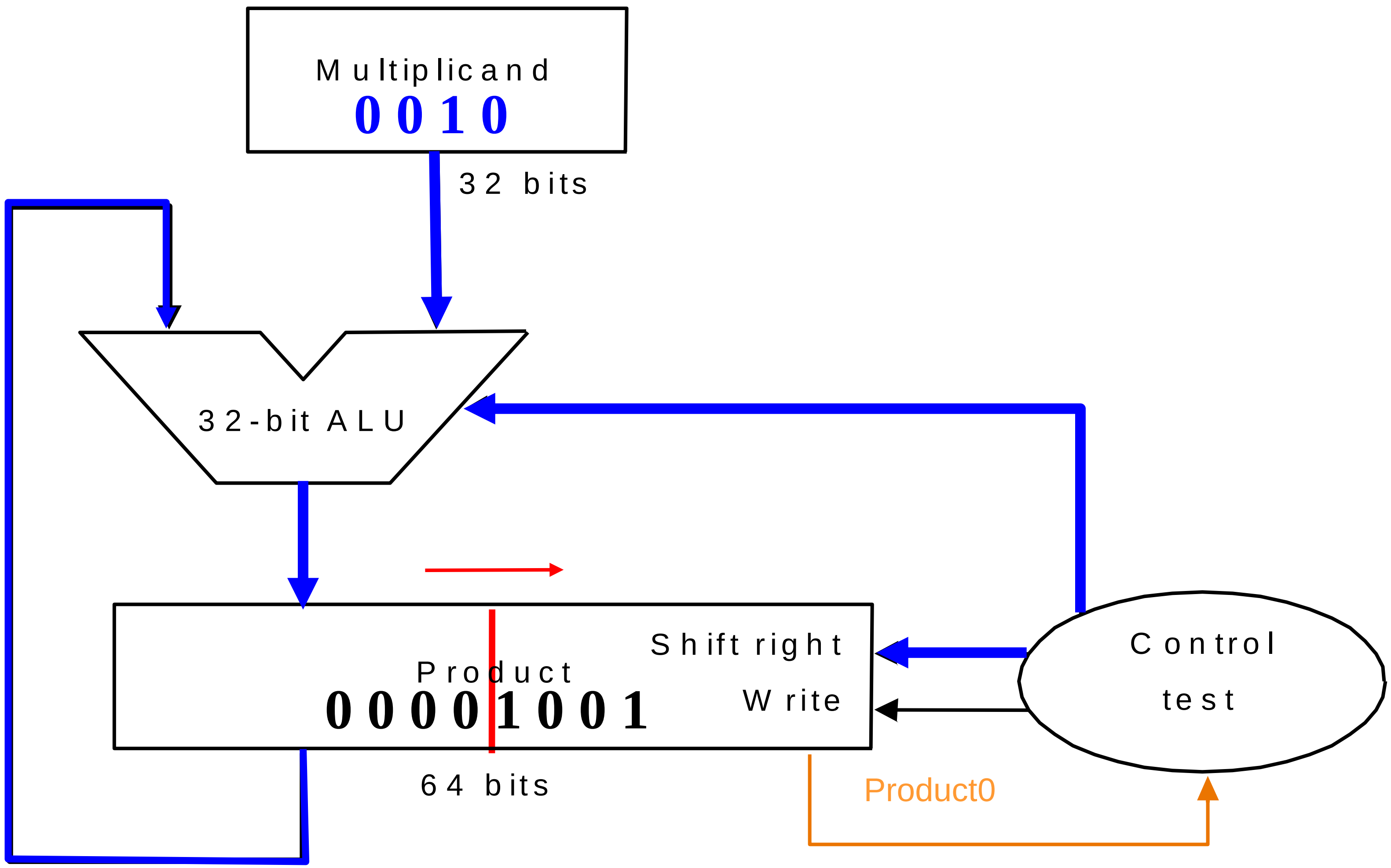
  Product : Shift Right

➔ **We can save spaces with version 2 implementation.**

Second Version



Version 3 : Multiplier is saved at Product register.



Final Version

Multiplicand 0 0 1 0

Product	0	0	0	0	1	0	0	1	
									Add
	0	0	1	0	1	0	0	1	
									Shift
	0	0	0	1	0	1	0	0	
									Shift
	0	0	0	0	1	0	1	0	
									Shift
	0	0	0	0	0	1	0	1	
									Add
	0	0	1	0	0	1	0	1	
									Shift
	0	0	0	1	0	0	1	0	

Exercise

- 10101 X 01110 = 21 X 14

	0 0 0 0 0 0 1 1 1 0	shift
	0 0 0 0 0 0 0 1 1 1	add
+	1 0 1 0 1	
	1 0 1 0 1 0 0 1 1 1	shift
	0 1 0 0 1 0 0 1 1 0	= 294

Multiplication of negative numbers

- What about the multiplication of negative number(s)?
  1) Negate negative number(s).
  2) Perform unsigned multiplication
  3) If multiplier and multiplicand have different sign,
     negate the result.

Or use Booth algorithm

Exercise

Perform (-7) X 3 with add-and-shift method. Assume multiplicand and multiplier are 4-bit number.

Summary

- In this section, we have discussed multiplying two integers.
- We can multiply two integers without dedicated multiplying unit.
- We can multiply negative integers by considering their sign bit only.

1. Addition and Subtraction
2. Multiplication
3. **Division**
4. Floating point number arithmetic

Introduction

- **In this section, we will learn how to divide integers with same MIPS ALU which does not have division capability.**
- **The division algorithm is based on the division method in elementary school.**
- **Also dividing negative integers will be discussed.**

Division

- **We can build divider using truth table**
- **Or we can do division with adder (subtracter) and shifter registers.**
- **Grade school algorithm example**

```
                  0001001        Quotient
                 _________
Divisor  1000   | 1001010        Dividend
                -1000
                 ---------
                     10
                     101
                     1010
                    -1000
                    ------
                       10        Remainder
```

Division

- **Algorithm**

```
If partial remainder ≥ divisor then
    quotient bit = 1;
    remainder = remainder – divisor;
else
    quotient bit = 0;
shift down next dividend bit
```

Division Hardware

- **Same Hardware as Multiplication:**
  just need ALU to add or subtract

- **Hi and Lo registers in MIPS combine to act as 64-bit register for multiplication and division.**

  - register Hi : stores Remainder

  - register Lo : stores Quotient

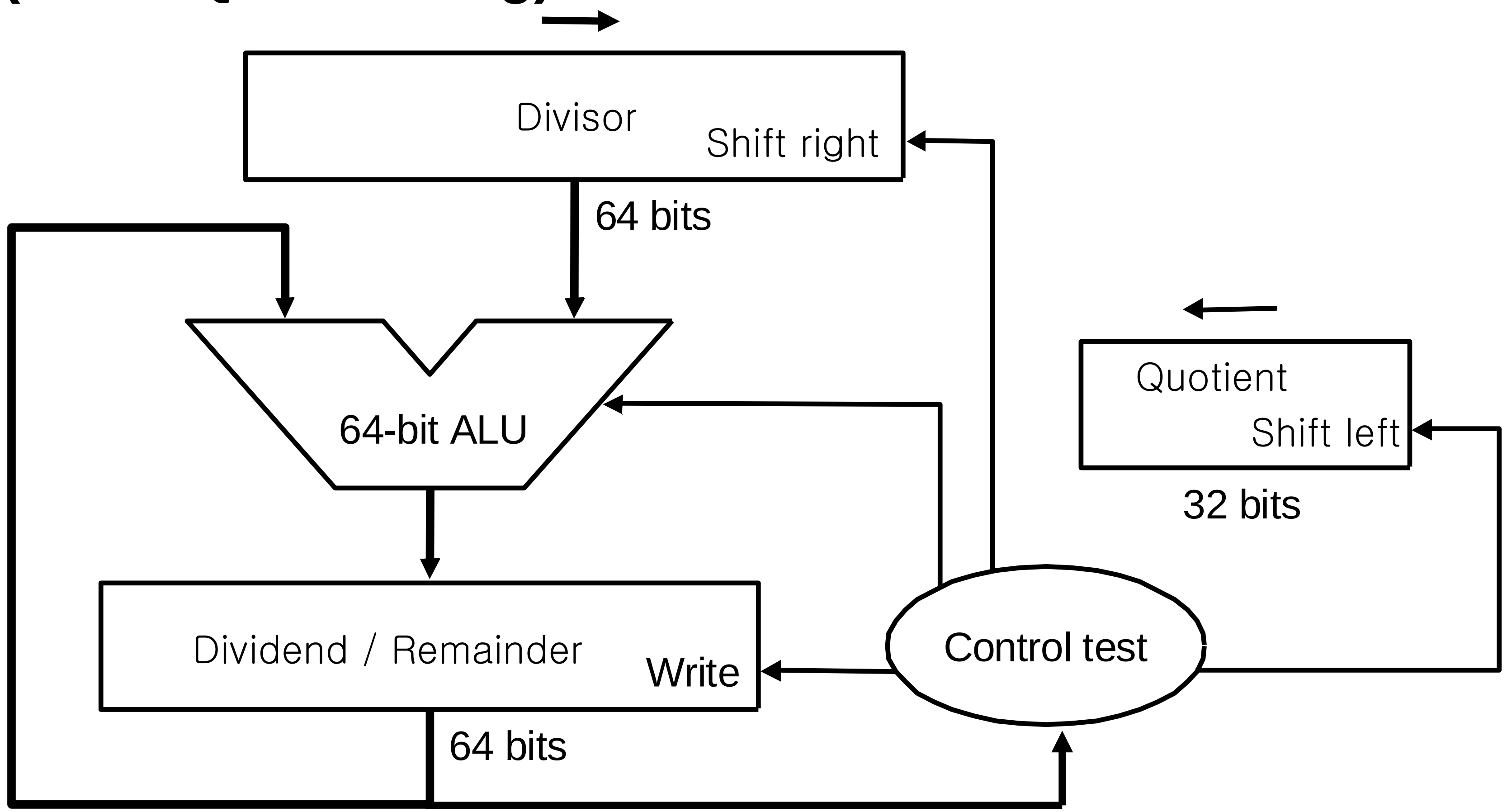
  - storing results at registers : mfhi $t0, mflo $t1

  *** For multiplication, Hi stores higher 32bits of result**

  and Lo stores lower 32 bits of result.

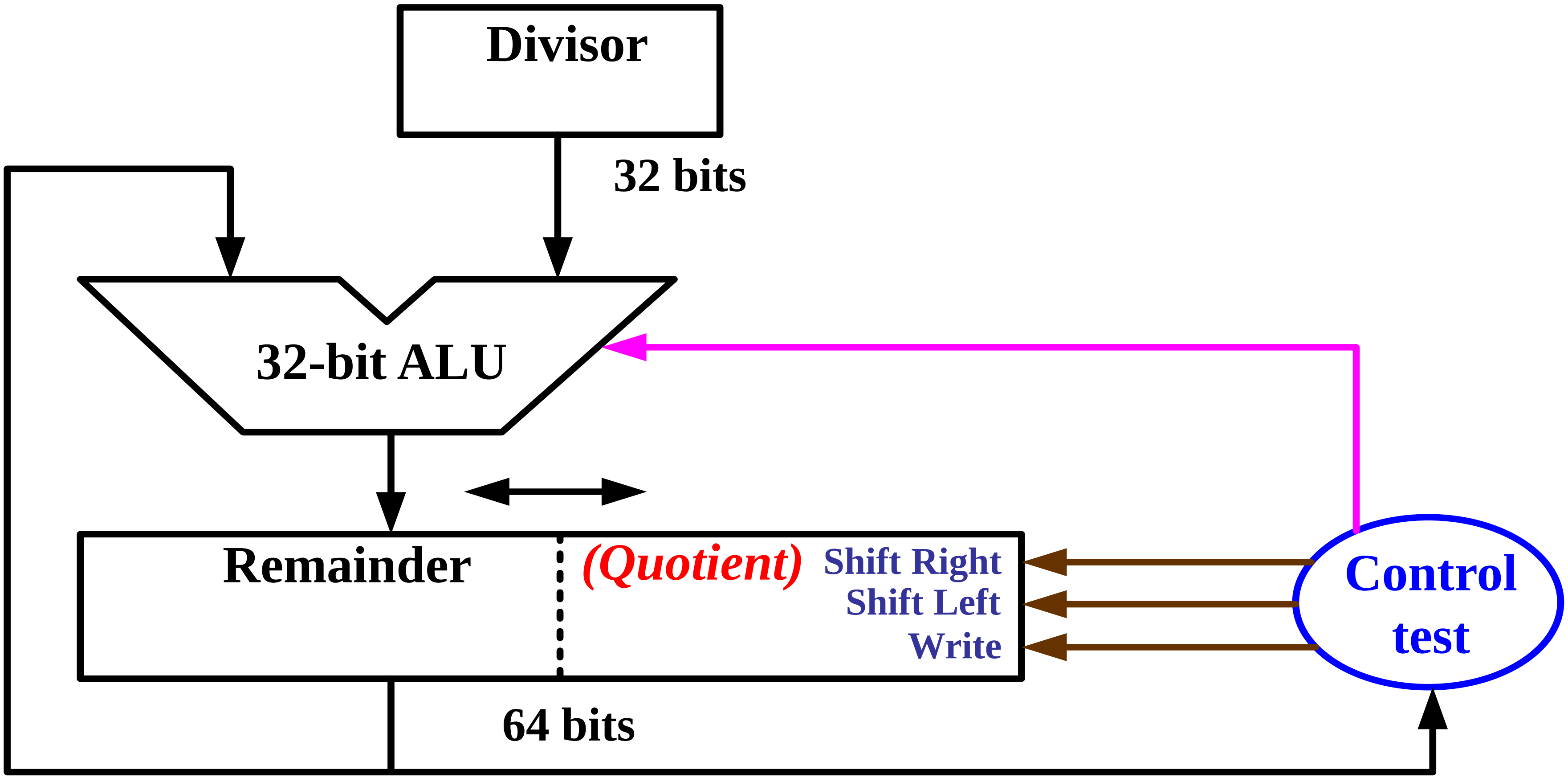
Division Hardware

- **64-bit Divisor reg, 64 -bit ALU, 64-bit Remainder reg, (32-bit Quotient reg)**



Division Hardware – improved version

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



Example

Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem – Div	0000	0010 0000	1110 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	1111 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	1111 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	0000 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	0000 0001
	2a: Rem ≥ 0 ⟹ sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

FIGURE 3.12 Division example using the algorithm in Figure 3.11. The bit examined to determine the next step is circled in color.

Example : 7 / 2

0 0 0 0 0 1 1 1	SLL
0 0 0 0 1 1 1 0	SUB
- 0 0 1 0	
1 1 1 0 1 1 1 0	Put 0
+ 0 0 1 0	Restore
0 0 0 0 1 1 1 0	SLL
0 0 0 1 1 1 0 0	SUB
- 0 0 1 0	
1 1 1 1 1 1 0 0	Put 0
+ 0 0 1 0	Restore
0 0 0 1 1 1 0 0	SLL
0 0 1 1 1 0 0 0	SUB
- 0 0 1 0	
0 0 0 1 1 0 0 1	Put 1
0 0 1 1 0 0 1 0	SLL
- 0 0 1 0	SUB
0 0 0 1 0 0 1 0	Put 1
0 0 0 1 0 0 1 1	

Remainder : 0001

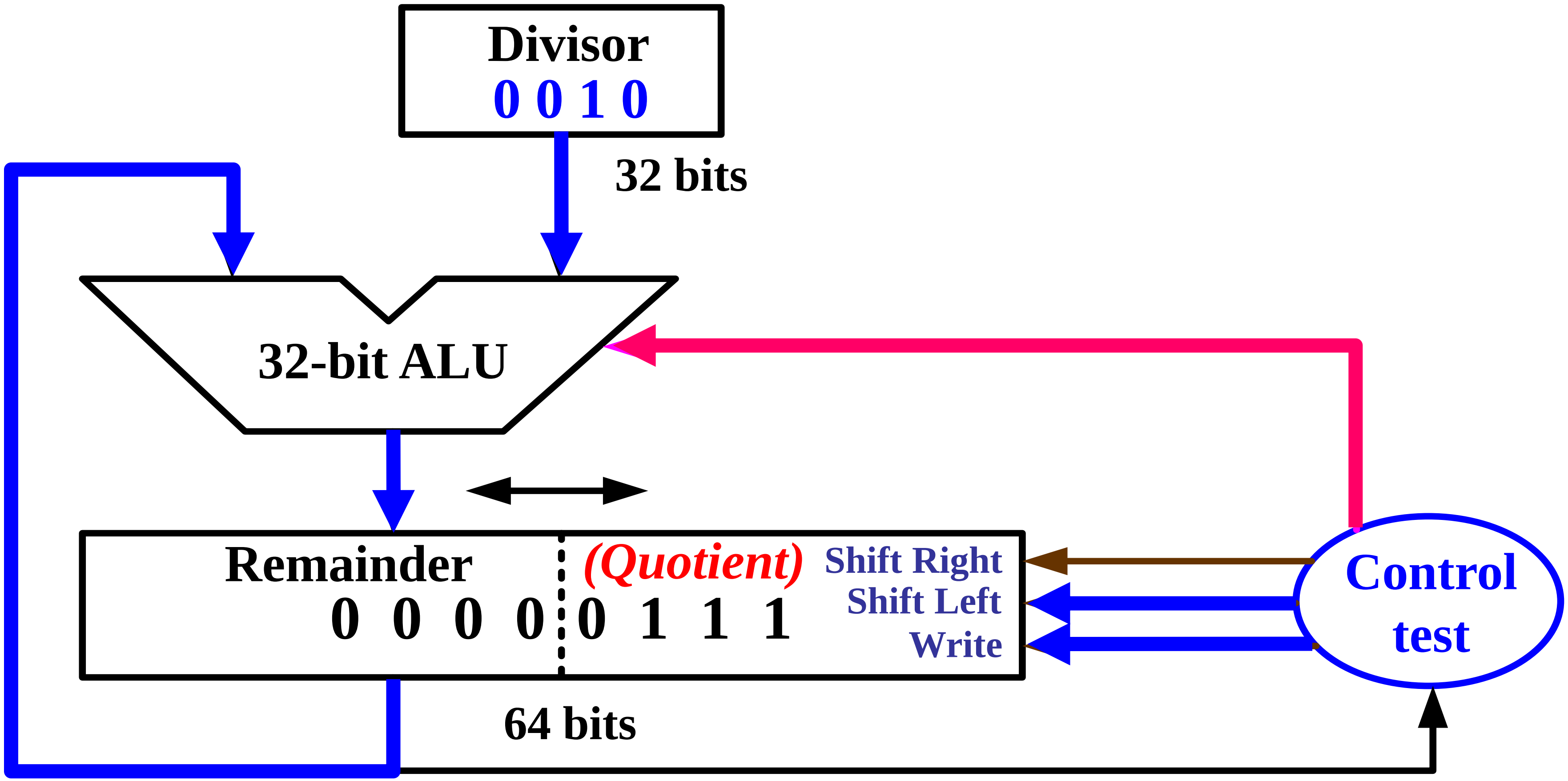
Quotient : 0011

Note :

Above implementation is a little different from the one in the textbook

Division Hardware – improved version

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



Division of signed number

Example)

					Q	R
+7	/	2	=		3	1
-7	/	2	=		-3	-1
+7	/	-2	=		-3	1
-7	/	-2	=		3	-1

- The magnitude of quotient and remainder depends on the magnitude of the dividend and divisor.
- The sign of quotient and remainder depends on the sign of dividend and divisor.

Exercise

Perform (-7) / 3 with MIPS ALU.

Summary

- In this section, we have discussed dividing two integers.
- We can divide two integers without dedicated dividing unit.
- We can divide negative integers by considering their sign bit only.

  - The magnitude of quotient and remainder depends on the magnitude of the dividend and divisor.

  - The sign of quotient and remainder depends on the sign of dividend and divisor.

1. Addition and Subtraction
2. Multiplication
3. Division
4. **Floating point number arithmetic**

Introduction

- In this section, we will learn how to represent floating point number and its arithmetic.
- IEEE (Institute of Electrical and Electronics Engineers) has its own standard form of representing floating point numbers.

Floating Point Number

- **We need a way to represent**
  - **numbers with fractions, e.g., 3.1416**
  - **very small numbers, e.g., .000000001**
  - **very large numbers, e.g., 3.15576 X 10^9**

➔ solution : floating point number representation

Normalized Scientific Notation

- **Scientific notation**

  coefficient X (base number)$^{\text{exponent}}$

  coefficient : a single digit to the left of the decimal point

  ex) 7.15576×10^4, 0.314×10^1

- **Normalized scientific notation**

  $1 \leq$ coefficient < 10

  ex) 7.15576×10^4, 3.14×10^0

  - For binary number,

  $1.yyyy\ldots \times 2^z$

Normalized Scientific Notation

- **Representation: Normalized scientific notation**
  - **sign, exponent, significand :**

    $(-1)^{sign} \times significand \times 2^{exponent}$
  - **more bits for significand gives more accuracy**
  - **more bits for exponent increases range**

- **IEEE 754 floating point standard:**
  - **single precision (32bits): 8 bit exponent, 23 bit fraction**

1	8	23

  - **double precision (64bits): 11 bit exponent, 52 bit fraction**

IEEE 754 floating-point standard

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

- **Exponent is "biased" to make sorting easier**
  - **bias of 127 for single precision and 1023 for double precision**
  - **all 0s : smallest exponent**

    exponent value = 0-127 = -127
  - **all 1s : largest exponent**

    exponent value = 255-127 = 128

- **Summary:** $(-1)^{\text{sign}}\ X\ (1+\text{faction})\ X\ 2^{\text{exponent} - \text{bias}}$

S	Exponent	Fraction

IEEE 754 floating-point standard

- **Example:**
  - **decimal: $-0.75 = -3/4 = -3/2^2$**
  - **binary: $-.11 = -1.1 \times 2^{-1}$**
  - **floating point: exponent = $-1 + 127 = 126 =$ 01111110**
  - **IEEE single precision:**

$$\mathbf{1}\ \underbrace{\mathbf{01111110}}_{126}\ \underbrace{\mathbf{10000000000000000000000}}_{0.1}$$

$$(-1)^{1} \times 1.1 \times 2^{126-127} = -1.1 \times 2^{-1}$$

Single-Precision Range

- **Exponents 00000000 and 11111111 reserved**
- **Smallest value**
  - **Exponent: 00000001**
    $\Rightarrow$ actual exponent = 1 – 127 = –126
  - **Fraction: 000…00 $\Rightarrow$ significand = 1.0**
  - **$\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$**
- **Largest value**
  - **exponent: 11111110**
    $\Rightarrow$ actual exponent = 254 – 127 = +127
  - **Fraction: 111…11 $\Rightarrow$ significand ≈ 2.0**
  - **$\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$**

Double-Precision Range

- Exponents 0000…00 and 1111…11 reserved
- Smallest value
  - Exponent: 00000000001
    $\Rightarrow$ actual exponent = 1 – 1023 = –1022
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- Largest value
  - Exponent: 11111111110
    $\Rightarrow$ actual exponent = 2046 – 1023 = +1023
  - Fraction: 111…11 $\Rightarrow$ significand ≈ 2.0
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

Floating Point Complexities

- **If digits are all zero… ➔ It represents 0, not 1.0 x 2^{-127}**

- **In addition to overflow there is "underflow"**

  0.5 x 2^{-129} ??

Floating-Point Precision

- **Relative precision**
  - **all fraction bits are significant**
  - **Single: approx 2^{-23}**
    - Equivalent to $23 \times \log_{10}2 \approx 23 \times 0.3 \approx 6$ decimal digits of precision
  - **Double: approx 2^{-52}**
    - Equivalent to $52 \times \log_{10}2 \approx 52 \times 0.3 \approx 16$ decimal digits of precision

Floating-Point Example

- **Represent –3.3125**

1 1 ... 0 10101 ... 0

1 8 23

Floating-Point Example

- **What number is represented by the single-precision float?**

11000000101000…00

−5

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

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

- Smaller than normalized numbers
- Denormalized # with fraction = 000...0

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

Two representations of 0.0!

Infinities and NaNs

- **Exponent = 111...1, Fraction = 000...0**
  - **±Infinity**
- **Exponent = 111...1, Fraction ≠ 000...0**
  - **Not-a-Number (NaN)**
  - **Indicates illegal or undefined result**
    - e.g., 0.0 / 0.0

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)

Basic Addition Algorithm

- **Operations are somewhat more complicated than addition of integer.**

Addition of 9.999×10^{1} to 1.610×10^{-1}

Step 1 : Align the decimal point (denormalize smaller number)

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

Step 2 : Add significand part

10.015×10^{1}

Step 3 : Normalize result & check for over/underflow

1.0015×10^{2}

*** IEEE 754 keeps two extra bits, guard bit and round bit.**

Step 4 : Round and renormalize if necessary

1.002×10^{2}

Guard bit and Round bit

$2.56 \times 10^0 + 2.34 \times 10^2$

1) **Without them**

2) **With them**

Floating-Point Addition

- Now consider a 4-digit binary example
  - $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ (0.5 + –0.4375)
- 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

Exercise

$1.0110 \times 2^3 + 1.1000 \times 2^2$

Exercise

$1.0110 \times 2^3 + 1.1001 \times 2^2$

Basic Multiplication Algorithm

1. Compute exponents

- Exp = Exp(X) + Exp(Y) – bias

2. Multiply significands

- Sig = Sig(X) x Sig(Y)

3. Normalize the product

- Shift Sig right until leading bit is 1; incrementing Exp.
- Check for overflow in Exp
- Round
- Repeat step 3 if not still normalized

4. Set sign

- positive if two number have same sign; negative otherwise

Multiplication example

$(1.0110 \times 2^3) \times (1.1100 \times 2^2)$

= 0 10000010 0110000… x 0 10000001 110000…

1. Compute exponents

- Exp = Exp(X) + Exp(Y) – bias

  = 10000010 + 10000001 – 01111111 = 10000100

2. Multiply significands

- Sig = Sig(X) x Sig(Y) = 1.011000… x 1.110000… = 10.01101000…

3. Normalize the product

- Shift Sig right until leading bit is 1; incrementing Exp

  $10.01101000\ldots \times 2^{10000100} = 1.00110100\ldots \times 2^{10000101}$

4. Set sign

- Sign = 0

Result : 0 10000101 00110100… = $1.001101 \times 2^{133-127}$

Exercise

1. Represent -12.75 with IEEE 754 floating-point single precision standard representation.

2. Add 1.01010×2^4 to 1.00111×2^6

   Do we need guard / round bit? Assume significand is 5-bits long.

3. Multiply 1.010×2^4 to 1.001×2^6

Summary

- We have learned IEEE 754 standard form of representing floating point numbers.
- We need three fields – sign bit, fraction, and exponent – to represent floating point numbers.
- We use ‘bias’ representation for exponent field.
- We have to shift and align significand when adding two floating point numbers.
- Checking overflow and having precise result is also important, so we keep ‘guard’ bit and ‘round’ bit.