SECR2033
Computer Organization and Architecture

Lecture slides prepared by "Computer Organization and Architecture", 9/e, by William Stallings, 2013.

Module 2

Data Representation in Computer Systems

Objectives:

- To understand the fundamentals of **numerical data** representation and manipulation in digital computers.
- To master the skill of converting between various **radix systems**.
- To understand how **errors** can occur in computations because of overflow and truncation.
- To understand the fundamental concepts of **floating-point** representation.

Module 2
Data Representation in Computer Systems

Module 2

Data Representation in Computer Systems

2.1 Introduction

2.2 Fixed-Number (Integer) Re

2.3 Fixed-Number (Integer) Ari

2.4 Floating-Points Representa

2.5 Floating-Points Arithmetic

2.6 Summary

- The Arithmetic and Logic Unit

2.1 Introduction

2

Numbers are represented by binary bits:

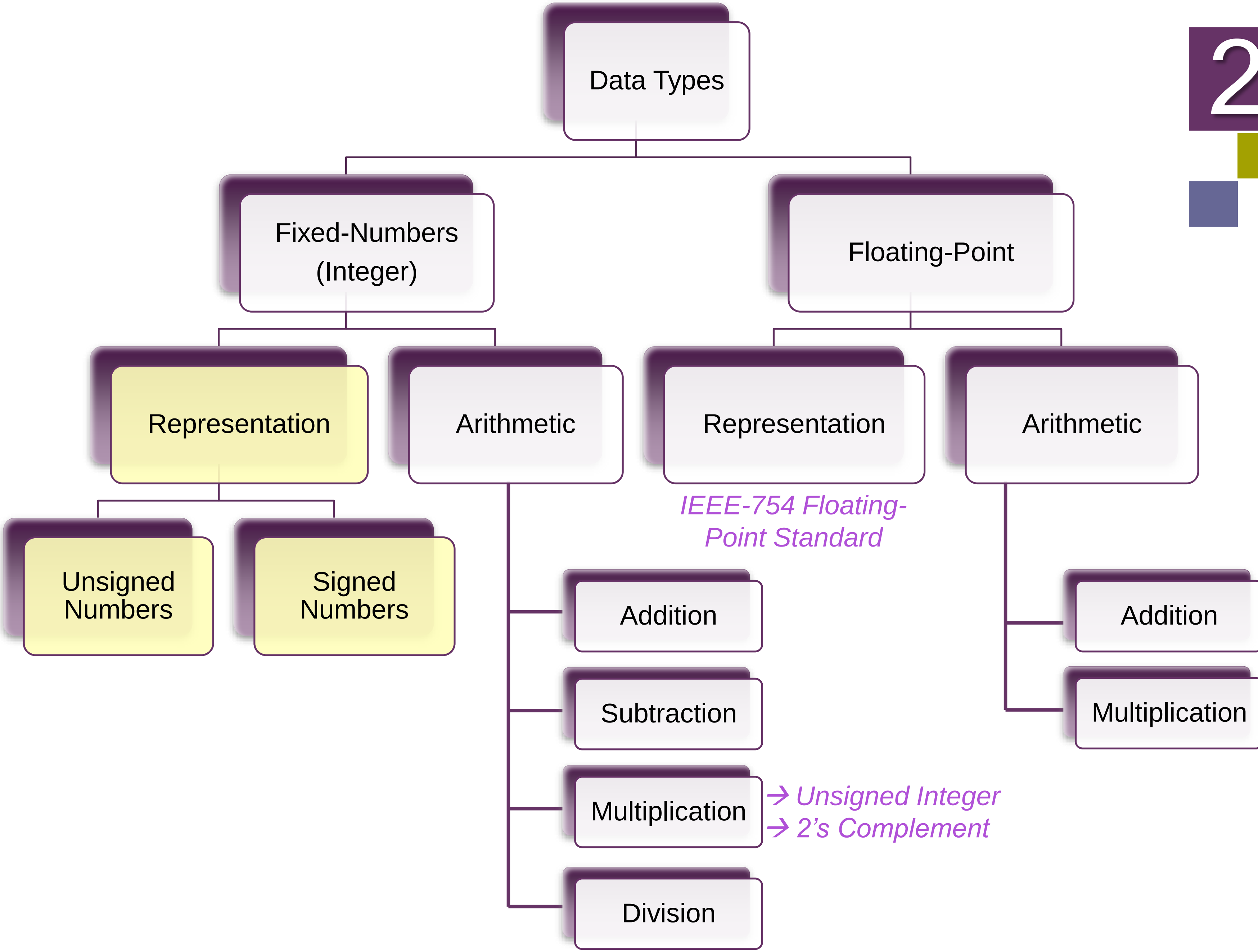
- *How are negative numbers represented?*
- *What is the largest number that can be represented in a computer world?*
- *What happens if an operation creates a number bigger than can be represented?*
- *What about fractions and real numbers?*
- *A mystery: How does hardware really multiply or divide numbers?*

- We begin our examination of the processor with an overview of the arithmetic and logic unit (ALU) → *computer arithmetic*.
- Computer arithmetic is commonly performed on two very different types of numbers: integer and floating point.

- In both cases, the representation chosen is a crucial design issue and is treated first, followed by a discussion of arithmetic operations.

C Basic Data Types	32-bit CPU		64-bit CPU	
	Size (bytes)	Range	Size (bytes)	Range
char	1	-128 to 127	1	-128 to 127
short	2	-32,768 to 32,767	2	-32,768 to 32,767
int	4	-2,147,483,648 to 2,147,483,647	4	-2,147,483,648 to 2,147,483,647
long	4	-2,147,483,648 to 2,147,483,647	8	- 9,223,372,036,854,775,808-9,223,372,036,854,775,807
long long	8	9,223,372,036,854,775,808-9,223,372,036,854,775,807	8	9,223,372,036,854,775,808-9,223,372,036,854,775,807
float	4	3.4E +/- 38	4	3.4E +/- 38
double	8	1.7E +/- 308	8	1.7E +/- 308

William Stallings (2013). *Computer Organization and Architecture: Designing for Performance* (9th Edition). United States: Pearson Education Limited, p.342.



William Stallings (2013). *Computer Organization and Architecture: Designing for Performance* (9th Edition). United States: Pearson Education Limited, p.341.
Linda Null and Julia Lobur (2003). *The Essentials of Computer Organization and Architecture.* United States: Jones and Bartlett Publishers. p.37.

The Arithmetic and Logic Unit

- The ALU is that part of the computer that actually performs arithmetic and logical operations on data.

- All electronic components in the computer are based on the use of simple digital logic devices that can store binary digits and perform simple Boolean logic operations.

- ❑ Data are presented to the ALU in registers,
- ❑ and the results of an operation are stored in registers.

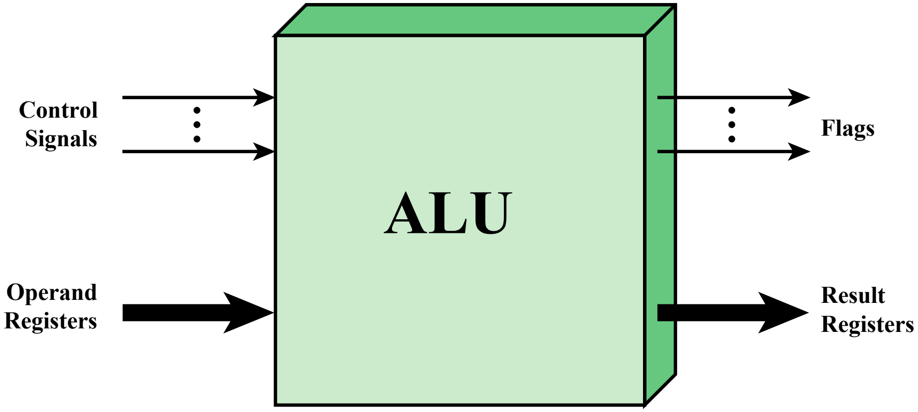


Figure: ALU inputs and outputs.

Module 2

Data Representation in Computer Systems

2.1 Introduction

2.2 Fixed-Number (Integer) Representation

- Overview
- Unsigned Numbers
- Signed Numbers

2.3 Fixed-Number (Integer) Arithn

2.4 Floating-Points Representatio

2.5 Floating-Points Arithmetic

2.6 Summary

RECAP

2

Overview

(Binary Number)

- Numbers are kept in computer hardware as a series of high and low electronic signals.
- They are considered base 2 numbers (Binary)
- All information is composed of binary digits or *bits*.
- In any number base, the value of i th digit d is

$$d \times Base^{i}$$

where i start at 0 and increases from right to left.

RECAP

2

Why is a kilobyte 1024 bytes and not 1000?
This 8-bit unit called a **byte**. Because every memory unit is based on powers of 2, a **kilobyte** is defined **not** as a thousand (as in other conventional measurements), but as 2^10 **bytes** = **1024 bytes**. **1024** is close enough to a thousand to earn the kilo tag.

Binary Equivalents

- 1 *Nybble* (or *nibble*) = 4 *bits*
- 1 *Byte* = 2 *nybbles* = 8 *bits*
- 1 *Kilobyte* (*KB*) = 1024 *Bytes*
- 1 *Megabyte* (*MB*) = 1024 *Kilobytes* = 1,048,576 *Bytes*
- 1 *Gigabyte* (*GB*) = 1024 *Megabytes* = 1,073,741,824 *Bytes*

RECAP

2

Overview

(Sign Extension)

- Extending a number representation to a larger number of bits.
- **Example**: 2 in 8 bit binary to 16 bit binary.

00000010 → 00000000 00000010

- In signed numbers, it is important to extend the sign bit to *preserve* the number (+ve or -ve)
- **Example**: (–2) in 8 bit binary to 16 bit binary.

11111110 → 11111111 11111110

Sign bit — *Sign bit extended* — *Sign bit*

RECAP

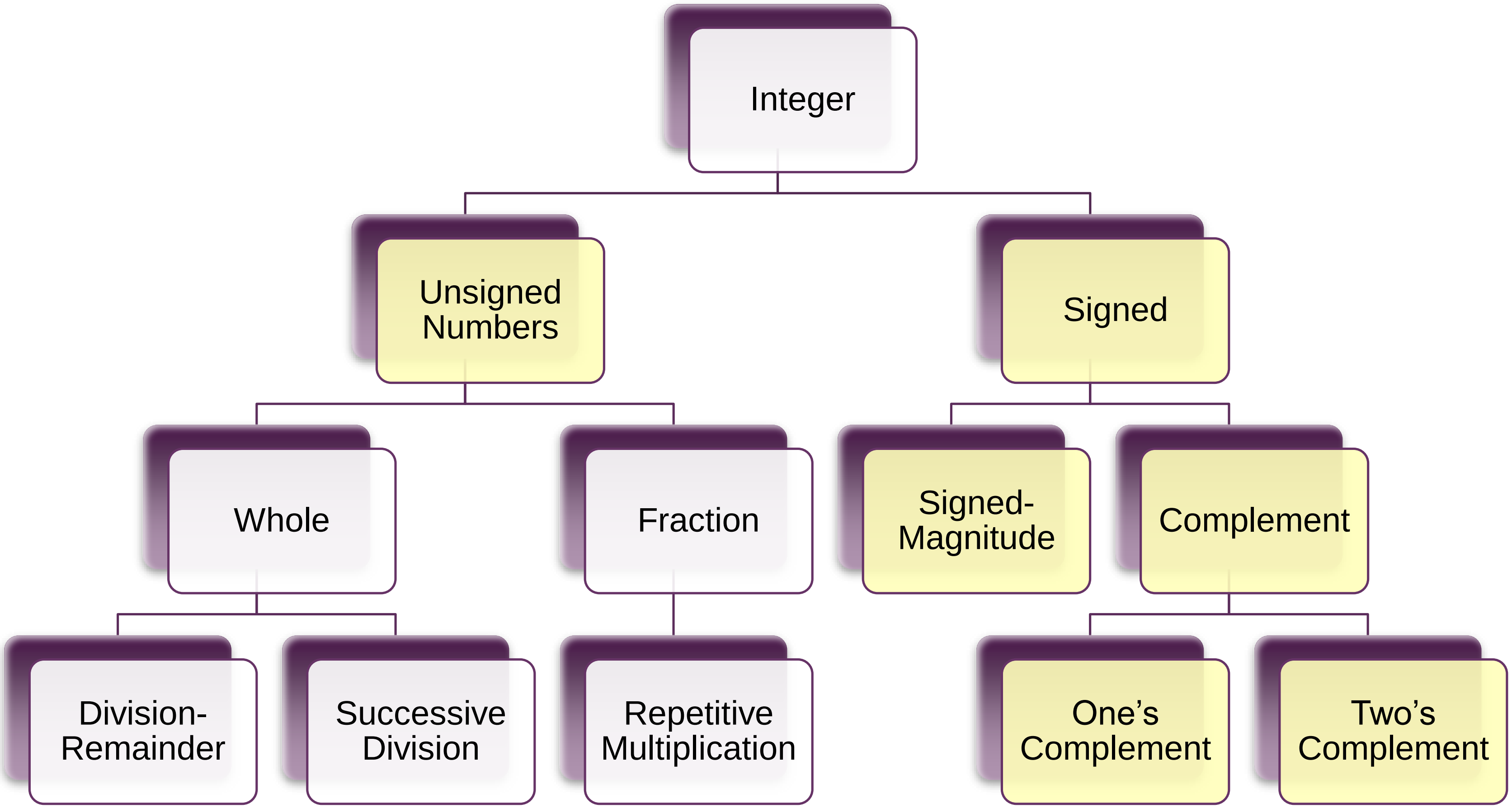


Figure: Types of numbers.

RECAP

2

Unsigned Numbers

MIPS (Millions Instruction Per Second)
LSB (Least Significant Bit)
MSB (Most Significant Bit)

- Since MIPS word is 32 bits long:
  - LSB → bit 0
  - MSB → bit 31

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1

(32 bits wide)

Figure: The numbering of bits in MIPS word for 1011_2 (11_{10})

- Range Number can represent : 0 to $2^{32} - 1$

  $0 - 4{,}294{,}967{,}295_{10}$

Signed Numbers

(a) Signed-Magnitude

- The conversions we have so far presented have involved only positive numbers.
- To represent negative values, computer systems allocate the high-order bit to indicate the sign of a value.
  - The high-order bit is the leftmost bit in a byte. It is also called the Most Significant Bit (MSB).
- The remaining bits contain the value of the number.

RECAP

2

Example 1:

Add 79_{10} to 35_{10} using signed-magnitude arithmetic in 8-bit binary.

				1	1	1	1			⇐ carries
0		1	0	0	1	1	1	1		(79)
0	+	0	1	0	0	0	1	1		+ (35)
0		1	1	1	0	0	1	0		(114)

7 bits

0 represents positive

RECAP

2

- The sum of two positive numbers, which is positive.
- *Overflow* (and thus an erroneous result) in signed numbers occurs when the sign of the result is incorrect.
- The *sign bit* is used only for the sign, so we can't "carry into" it, otherwise the result will be truncated as the MSB bit overflows, giving an incorrect sum.

- If the *overflow* bit is not discard, it would carry into the sign, causing the more outrageous result of the sum of two positive numbers being negative.

Linda Null and Julia Lobur (2003). *The Essentials of Computer Organization and Architecture.* United States: Jones and Bartlett Publishers. p.45-46.

Example 2:

Add 01001111_2 to 01100011_2 using signed-magnitude arithmetic in 8-bit binary.

Last carry	1 ←	1 1 1 1	⇐ carries
overflows and is	0	1 0 0 1 1 1 1	(79)
discarded.	0 +	1 1 0 0 0 1 1	+ (99)
	0	0 1 1 0 0 1 0	(50)

Error result !

Answer should be 178_{10}

Linda Null and Julia Lobur (2003). *The Essentials of Computer Organization and Architecture.* United States: Jones and Bartlett Publishers. p.46.

Example 3:

Add (–19_{10}) to 13_{10} using signed-magnitude arithmetic in 8-bit binary.

- The first number (*augend*) is negative (sign bit 1)
- The second number (*addend*) is positive (sign bit 0)
- Since larger magnitude is *augend*, then subtract to *addend*.

				0	1	2			⇐ borrows	
1		0	0	~~1~~	~~0~~	~~0~~	1	1		(−19)
0	−	0	0	0	1	1	0	1		+ (13)
1		0	0	0	0	1	1	0		(−6)

The sign of result will be same as sign of larger magnitude.

R

Video on binary subtraction with examples

https://www.youtube.com/watch?v=h_fY-zSiMtY

https://www.youtube.com/watch?v=Pr-j2fmcpxc

Ex

Add (-19_{10}) to 13_{10} using signed-magnitude arithmetic in 8-bit binary.

- The first number (*augend*) is negative (sign bit 1)
- The second number (*addend*) is positive (sign bit 0)
- Since larger magnitude is *augend*, then subtract to *addend*.

				0	1	2			⇐ borrows
1		0	0	~~1~~	~~0~~	~~0~~	1	1	(−19)
0	−	0	0	0	1	1	0	1	+ (13)
1		0	0	0	0	1	1	0	(−6)

The sign of result will be same as sign of larger magnitude.

Linda Null and Julia Lobur (2003). *The Essentials of Computer Organization and Architecture.* United States: Jones and Bartlett Publishers. p.48.

RECAP

2

- In signed magnitude, the sign bit is used only for the sign.

Problem 1:

If there any carry emitting from the last bit, the result will be truncated as the last bit overflow, giving an incorrect sum.

Problem 2:

Complicated to define the larger magnitude, to subtract negative number, borrow from the *minuend*.

Solution: Need a simpler method for representing signed numbers → complement systems.

Signed Numbers

(b) One's Complement

- In complement systems, negative values are represented by some difference between a number and its base.
- In *diminished radix complement* systems, a negative value is given by the difference between the absolute value of a number and one less than its base.
- In the binary system, this gives us one's complement.
  - It amounts to little more than flipping the bits of a binary number.

Example 4:

Using one's complement 8-bit binary arithmetic, find the sum of 9_{10} and (-23_{10}).

The last carry is zero so we are done.	0 ←	0 0 0 0 1 0 0 1	(9)
	+	1 1 1 0 1 0 0 0	+ (–23)
		1 1 1 1 0 0 0 1	-14_{10}

00010111 (-23)
11101000 (1s)

One's complement for -23
Flip the binary bit from 00010111 to 11101000

Example 5:

Using one's complement 8-bit binary arithmetic, find the sum of (-9_{10}) and 23_{10}.

Still remember how to convert -9 into 8 bit binary using 1s?

00001001 (-9)

11110110 (1s)

	1 ←	1	1	1		1	1			⇐ carries
		0	0	0	1	0	1	1	1	(23)
The last	+	1	1	1	1	0	1	1	0	+(–9)
carry is added		0	0	0	0	1	1	0	1	
								+	1	
to the sum.		0	0	0	0	1	1	1	0	14_{10}

With one's complement addition, the carry bit is "carried around" and added to the sum.

Linda Null and Julia Lobur (2003). *The Essentials of Computer Organization and Architecture.* United States: Jones and Bartlett Publishers. p.51.

RECAP

2

Signed Numbers

(c) Two's Complement

- Two's complement is an example of a *radix complement*.
- The radix complement is often more intuitive than the *diminished radix complement*.
- The two's complement is nothing more than one's complement incremented by 1.
- To find the two's complement of a binary number, simply flip bits and add 1.

RECAP

Express bit binary to 2s
1) Flip the bit
2) Add 1

Example 6:

Express 23_{10}, (-23_{10}), and (-9_{10}) in 8-bit binary two's complement form.

$$23_{10} = +(00010111_2) = 00010111_2$$
$$-23_{10} = -(00010111_2) = 11101000_2 + 1 = 11101001_2$$
$$-9_{10} = -(00001001_2) = 11110110_2 + 1 = 11110111_2$$

Example 7:

Add 9_{10} to (-23_{10}) using 8-bit binary two's complement arithmetic.

```
   0 0 0 0 1 0 0 1       (9)
 + 1 1 1 0 1 0 0 1    +(-23)
   ---------------    ------
   1 1 1 1 0 0 1 0    -14_10
```

00010111 (-23)
11101001 (2s)

Example 8:

Find the sum of 23_{10} and $(–9_{10})$ in binary using two’s complement arithmetic .

	1←	1	1	1		1	1	1		⇐ carries
Discard		0	0	0	1	0	1	1	1	(23)
carry.	+	1	1	1	1	0	1	1	1	+ (–9)
		0	0	0	0	1	1	1	0	14_{10}

With two’s complement addition, the carry bit is discarded.

Aside: Detecting Overflow

2

- Notice that the discarded carry in Example 8 did not cause an erroneous result.
- An overflow occurs if:
  - two positive numbers are added and the result is negative, or
  - two negative numbers are added and the result is positive.
- It is not possible to have overflow when using *two's complement notation* if a positive and a negative number are being added together.

- A simple rule for detecting an overflow condition using two’s complement arithmetic (Assume 8-bit binary):

If the carry into the *sign bit* <u>equals</u> the carry out of the bit, no overflow has occurred.

Carry out of the bit

Carry into the sign bit

	1 ←	1	1	1		1	1	1		⇐ carries
Discard		0	0	0	1	0	1	1	1	(23)
carry.	+	1	1	1	1	0	1	1	1	+ (–9)
		0	0	0	0	1	1	1	0	14_{10}

If the carry into the *sign bit* is <u>different</u> from the carry out of the sign bit, overflow (and thus an error) has occurred.

	0 ←	1	1	1	1					⇐ carries
Discard last		0	1	1	1	1	1	1	0	(126)
carry.	+	0	0	0	0	1	0	0	0	+(8)
		1	0	0	0	0	1	1	0	(–122???)

Review Questions - Activity 1

2.1 Represent the following decimal numbers in both binary sign/magnitude and two's complement using 16-bits binary system:

(a) +512 (b) –29

2.2 Represent the following two's complement values in decimal:

(a) 1101011_2 (b) 0101101_2

2.3 Convert the following 8-bit two's complement value to 16-bits binary system:

(a) 11001100_2 (b) 00101110_2

Review Questions - Activity 1

2.4 Assume numbers are represented in 8-bits two's complement representation. Show the calculation of the following:

(a) 6 + 13

(b) –6 + 13

(c) 6 – 13

(d) –6 – 13

2.5 Find the following differences using twos complement arithmetic:

(a) $111000_2 - 110011_2$

(b) $11001100_2 - 101110_2$

(c) 111100001111 – 110011110011

Module 2

Data Representation in Computer Systems

2.1 Introduction

2.2 Fixed-Number (Integer) Representation

2.3 Fixed-Number (Integer) Arithmetic

- Negation
- Addition
- Subtraction
- Multiplication
- Division

2.4 Floating-Points Representation

2.5 Floating-Points Arithmetic

2.6 Summary

RECAP

Still remember 2s complement?

2

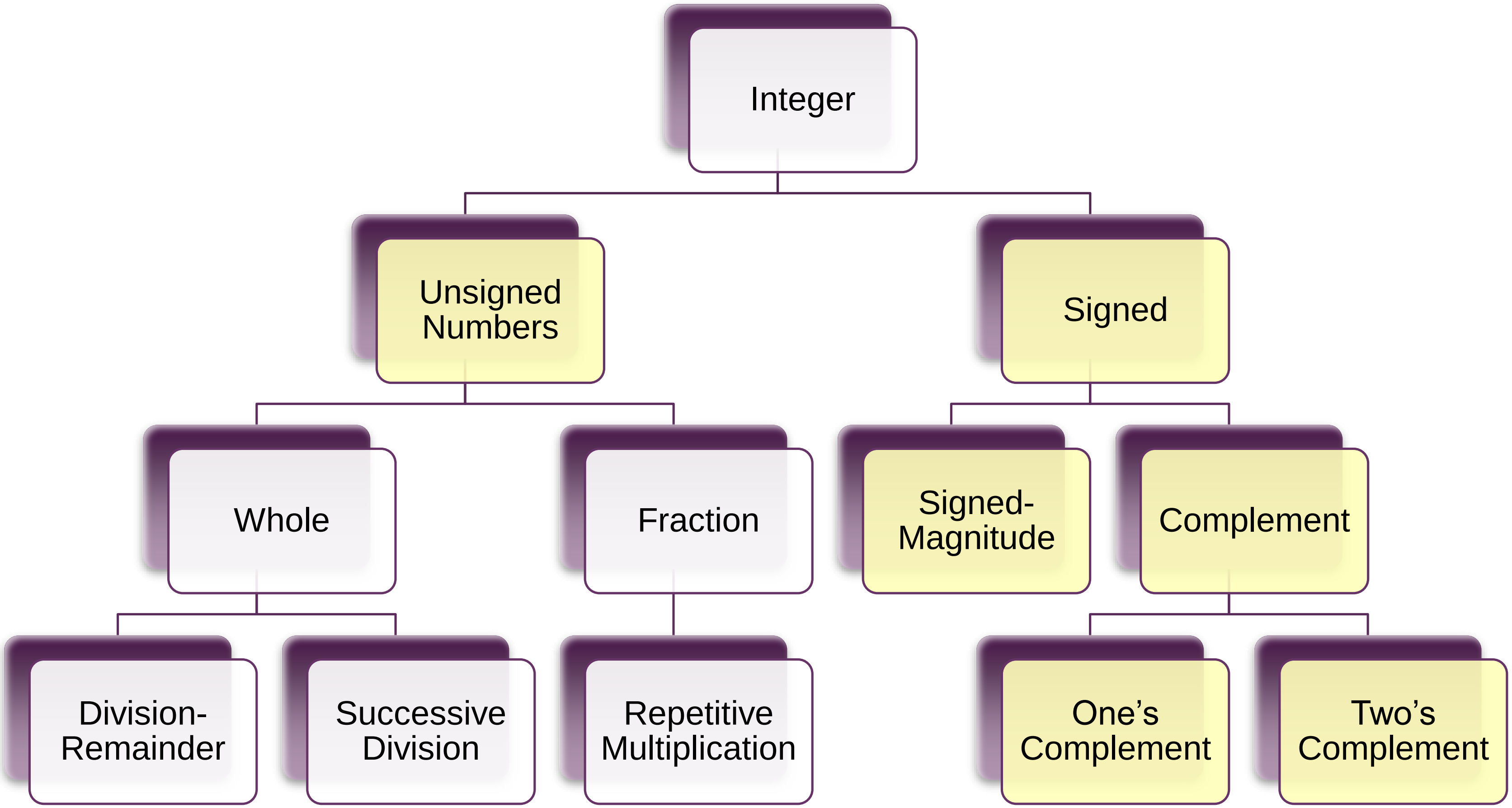


Figure: Types of numbers.

2.3 Fixed-Number Arithmetic

2

- This section examines common arithmetic functions on numbers in two’s complement representation.

Negation

- In two’s complement notation, the *negation* of an integer can be formed with the following rules:

1) Take the Boolean complement of each bit of the integer (including the sign bit) → Set each 1 to 0 and each 0 to 1.

2) Treating the result as an unsigned binary integer, add 1.

William Stallings (2013). *Computer Organization and Architecture: Designing for Performance* (9th Edition). United States: Pearson Education Limited, p.349.

- Those two-step process is referred to as the two’s complement operation, or the taking of the two’s complement of an integer.

$$
\begin{array}{rcl}
+18 & = & 00010010 \text{ (twos complement)} \\
\text{bitwise complement} & = & 11101101 \\
 & & \underline{+ \qquad\quad 1} \\
 & & 11101110 = -18
\end{array}
$$

$$
\begin{array}{rcl}
-18 & = & 11101110 \text{ (twos complement)} \\
\text{bitwise complement} & = & 00010001 \\
 & & \underline{+ \qquad\quad 1} \\
 & & 00010010 = +18
\end{array}
$$

2

(a) Addition

- Digits are added bit by bit from right to left, with carries passed to the next digit to the left.

RECAP

Rules of Binary Addition

- 0 + 0 = 0 carry = 0
- 0 + 1 = 1 carry = 0
- 1 + 0 = 1 carry = 0
- 1 + 1 = 0 carry = 1

2

Example 9:

Addition of numbers in two's complement representation.

0111 (-7)
1001 (2s)

Discard !

1001 = −7 +0101 = 5 1110 = −2 (a) (−7) + (+5)	1100 = −4 +0100 = 4 10000 = 0 (b) (−4) + (+4)
0011 = 3 +0100 = 4 0111 = 7 (c) (+3) + (+4)	1100 = −4 +1111 = −1 11011 = −5 (d) (−4) + (−1)

Discard !

(e) and (f) show examples of overflow that can occur whether or not there is a carry.

0101 = 5 +0100 = 4 1001 = Overflow (e) (+5) + (+4)	1001 = −7 +1010 = −6 10011 = Overflow (f) (−7) + (−6)

Example 9:

Addition of numbers in two's complement representation.

0111 (-7)
1001 (2s)

1001 = −7 +0101 = 5 1110 = −2 (a) (−7) + (+5)	1100 = −4 +0100 = 4 10000 = 0 (b) (−4) + (+4)
0011 = 3 +0100 = 4 0111 = 7 (c) (+3) + (+4)	1100 = −4 +1111 = −1 11011 = −5 (d) (−4) + (−1)

Discard ! (leading 1 in (b))

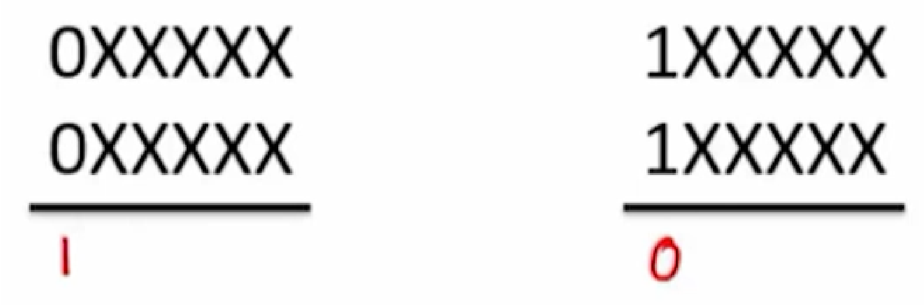
Discard ! (leading 1 in (d))

All successful addition operations

Rule:

1. If the result is +, we get a + number in 2s = unsigned integer form like in case b) & c)
2. If the result is -, we get a – number in 2s like in case a) & d)

William Stallings (2013). *Computer Organization and Architecture: Designing for Performance* (9th Edition). United States: Pearson Education Limited, p.351.



Overflow

(a) Addition

- On any addition, **overflow** happens when
  - If two numbers are added, and they are both positive or both negative, then overflow occurs if and only if the result has the opposite sign.
  - e) - result but 5 and 4 are +; f) + result but 7 and 6 are -
- When overflow occurs, the ALU must signal this fact so that no attempt is made to use the result

Overflow happens in e) because 9 > 2^3-1=7 i.e. result > largest number in 4 bit 2s system

(e)		(f)	
0101	= 5	1001	= −7
+0100	= 4	+1010	= −6
1001	= Overflow	10011	= Overflow
(e) (+5) + (+4)		(f) (−7) + (−6)	

(b) Subtraction

- Subtraction is easily handled with the following rule:

- **SUBTRACTION RULE:** To subtract one number (*subtrahend*) from another (*minuend*), take the two's complement (*negation*) of the *subtrahend* and add it to the *minuend*.

M (Minuend)
S (Subtrahend)

$$M - S = M + (-S)$$

(–S) → 2's Complement (negation)

Example 10:

Subtraction of numbers in two's complement representation $(M - S)$.

M (Minuend)
S (Subtrahend)

2

$$M - S = M + (-S)$$
$$= 2 + (-7)$$

0111 (-7)
1001 (2s)

```
   0010 =  2
  +1001 = –7
   1011 = –5

(a) M = 2 = 0010
    S = 7 = 0111
   –S =     1001
```

```
   0101 =  5
  +1110 = –2
  10011 =  3

(b) M = 5 = 0101
    S = 2 = 0010
   –S =     1110
```

```
   1011 = –5
  +1110 = –2
  11001 = –7

(c) M = –5 = 1011
    S =  2 = 0010
   –S =      1110
```

```
   0101 = 5
  +0010 = 2
   0111 = 7

(d) M =  5 = 0101
    S = –2 = 1110
   –S =      0010
```

Remember how overflow happens? Slide 39

```
   0111 = 7
  +0111 = 7
   1110 = Overflow

(e) M =  7 = 0111
    S = –7 = 1001
   –S =      0111
```

```
   1010 = –6
  +1100 = –4
  10110 = Overflow

(f) M = –6 = 1010
    S =  4 = 0100
   –S =      1100
```

Overflow !

William Stallings (2013). *Computer Organization and Architecture: Designing for Performance* (9th Edition). United States: Pearson Education Limited, p.351.

0111 (-7)
1001 (2s)

$$M - S = M + (-S)$$
$$= 2 + (-7)$$

M (Minuend)
S (Subtrahend)

2

Example 10:

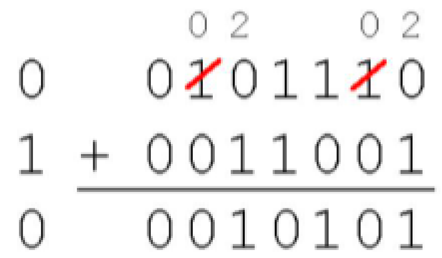
Check you answer e.g. in case a)
Convert 1011 (-5 in 2s) into 4bit binary

1)Flip the bit
1011-> 0100
2)Add 1;
0100+1=0101
Which is 5 in 4bit binary
So, 0101 = 5
And 1011 is -5 in 2s

```
     0010 =  2                 0101 =  5
    +1001 = –7                +1110 = –2
     1011 = –5                10011 =  3

(a) M = 2 = 0010          (b) M = 5 = 0101
    S = 7 = 0111              S = 2 = 0010
   –S =     1001             –S =     1110

     1011 = –5                 0101 = 5
    +1110 = –2                +0010 = 2
    11001 = –7                 0111 = 7

(c) M = –5 = 1011         (d) M =  5 = 0101
    S =  2 = 0010             S = –2 = 1110
   –S =      1110            –S =      0010

     0111 = 7                  1010 = –6
    +0111 = 7                 +1100 = –4
     1110 = Overflow          10110 = Overflow

(e) M =  7 = 0111         (f) M = –6 = 1010
    S = –7 = 1001             S =  4 = 0100
   –S =      0111            –S =      1100
```

Overflow !

- Subtraction can be done directly, e.g.



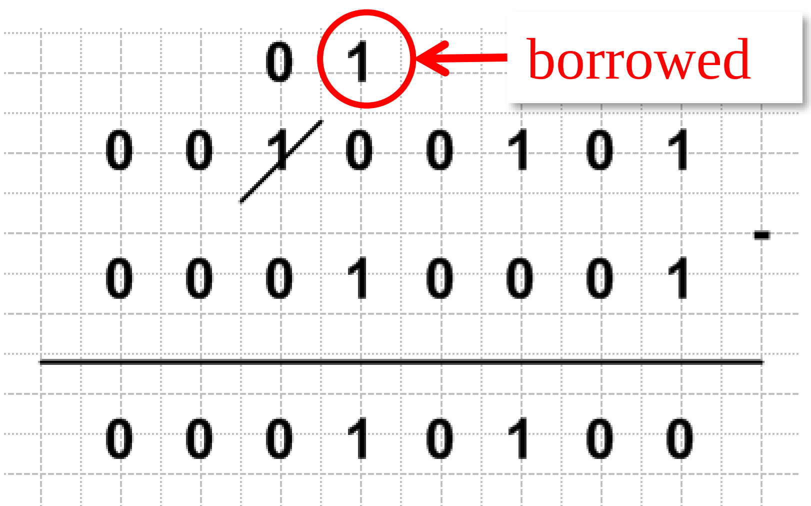
- Rules of binary subtraction:

> 0 - 0 = 0
> 0 - 1 = 1, *and borrow 1 from the next more significant bit*
> 1 - 0 = 1
> 1 - 1 = 0

Example 11:

$37_{10} - 17_{10}$

(Assume 8 bit binary)



Activity 2

2

Exercise 2.1:

Perform the subtraction in two's complement representation for $37_{10} - 17_{10}$ in 8 bit and 16 bit binary system.

(c) Multiplication

- Multiplication is a complex operation, whether performed in hardware or software.
- The simpler problem of multiplying using unsigned integers, and the most common techniques for multiplication of numbers is two's complement representation.
- Multiplication of binary numbers must always use 0 and 1.

Rules of Binary Multiplication:

0 x 0 = 0
0 x 1 = 0
1 x 0 = 0
1 x 1 = 1
and no carry or borrow bit.

(c) Multiplication

Unsigned Integer

- Consist of operands called *multiplicand* and *multiplier*, and final result as *product.*

```
      Multiplicand
x     Multiplier
      ____________
      Product
      ____________
```

William Stallings (2013). *Computer Organization and Architecture: Designing for Performance* (9th Edition). United States: Pearson Education Limited, p.353.

Example 12:

Multiply the unsigned binary numbers of 1011_2 by 1101_2.

```
      1011₂        → (11) Multiplicand
    x 1101₂        → (13) Multiplier
      1011   ┐
     0000    │
    1011     ├     Partial products
   1011      ┘
   10001111₂       → (143) Product
```

If we ignore the sign bits, the length of multiplication of an *n*-bit *multiplicand* and an *m*-bit *multiplier* is a *product* that is (*n*+*m*) bit long.

1	0	0	0	1	1	1	1
128	0	0	0	8	4	2	1

128+8+4+2+1=143

(c) Multiplication

Signed Integer: Two's Complement

- We have seen that addition and subtraction can be performed on numbers in two's complement notation by treating them as unsigned integers (positive numbers).
- **Example**:

```
  1001
+ 0011
  1100
```

If these numbers are considered to be unsigned integers, then we are adding 9 (1001_2) plus 3 (0011_2) to get 12 (1100_2)

As two's complement integers, we are adding –7 (1001_2) to 3 (0011_2) to get –4 (1100_2).

- Unfortunately, this simple scheme will not work for multiplication.

Example 13:

To see this, consider again **Example 12**.

Multiply the two's complement binary numbers of 1011_2 by 1101_2.

```
      1011₂     → (–5) Multiplicand
  x   1101₂     → (–3) Multiplier
      1011   ┐
     0000    │
    1011     ├  Partial products
   1011      ┘
  10001111₂     → (–113) Product
```

2

- This example demonstrates that straightforward multiplication will not work if both the multiplicand and/or multiplier are negative.

Regular multiplication clearly yields incorrect result !

- **Solution** : Multiplication algorithm.

  This algorithm has the benefit of speeding up the multiplication process, relative to a more straightforward approach.

A Multiplication Algorithm and Hardware

- Multiplication must cope with overflow because we frequently want a 32-bit *product* as the result of multiplying two 32-bit numbers.

- In the next slides, assume that we are multiplying only positive number (unsigned) with the 1st version of highly optimized multiplication hardware.

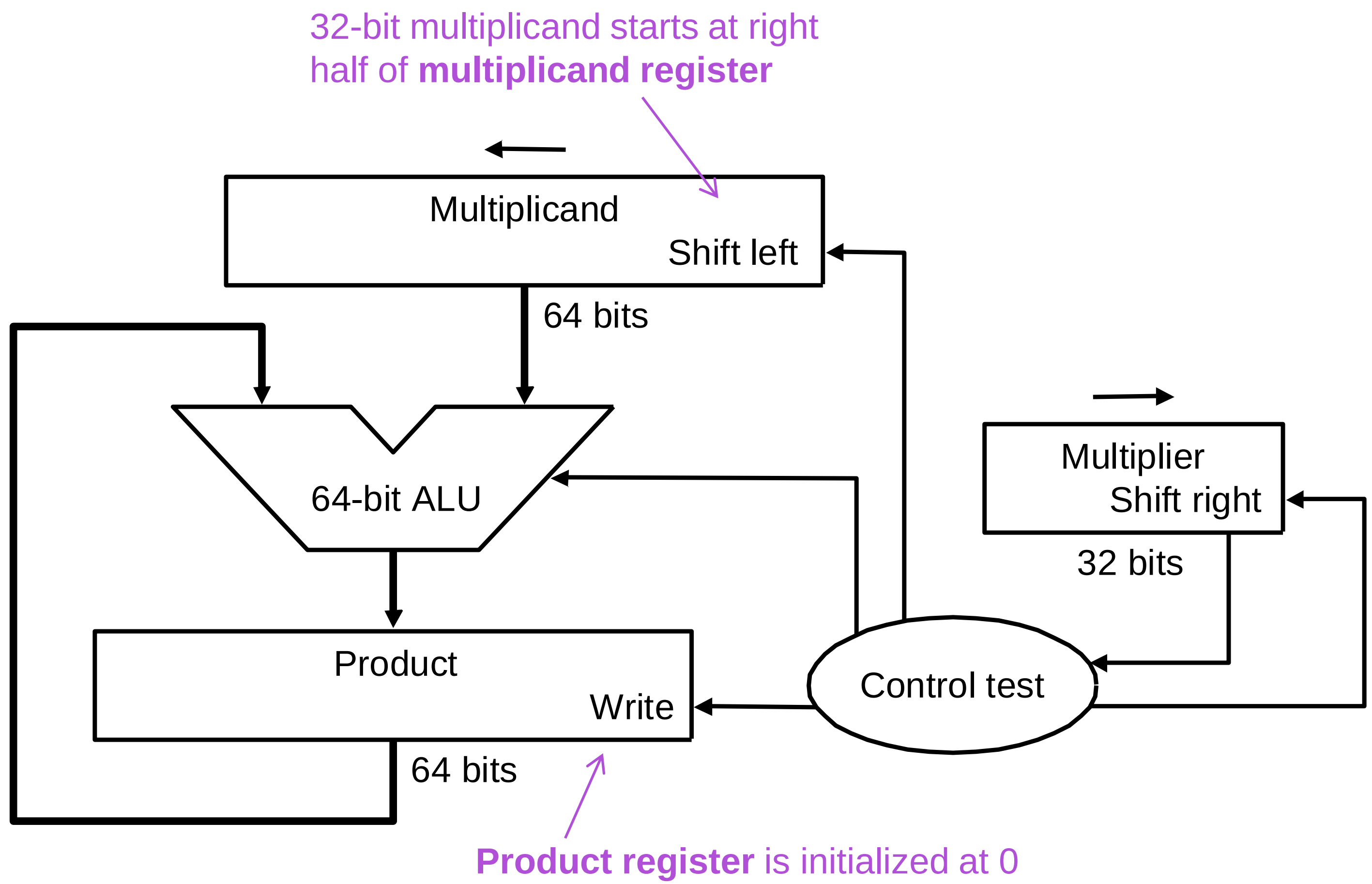


Figure: First version of Multiplication Hardware.

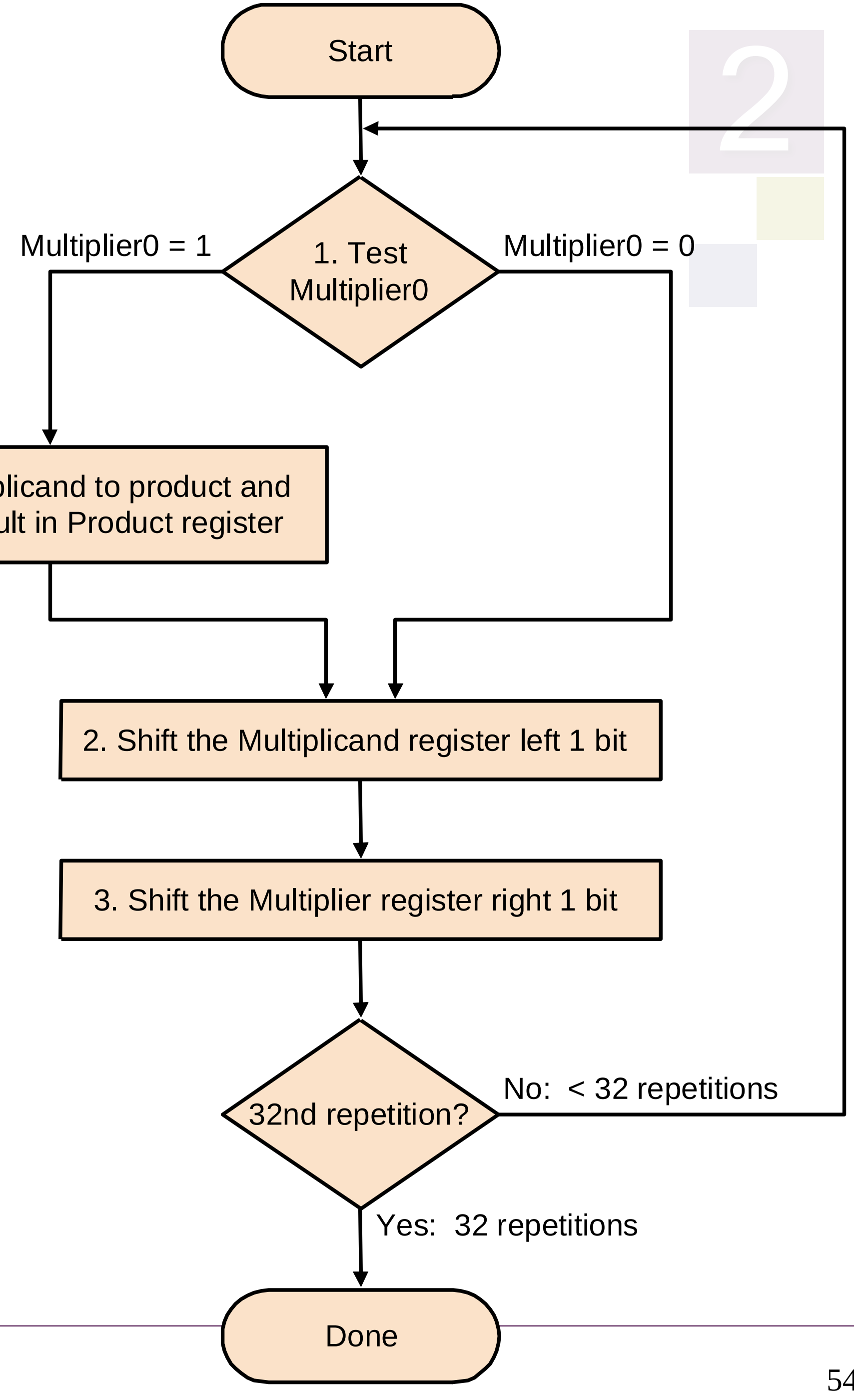


Figure:
The Multiplication Algorithm using the Hardware.

Patterson, D.A. and Hennessy, J.L. (2014). *Computer Organization and Design: The Hardware/Software Interface* (5th Edition). United States: Elsevier, p.185.

Example 14:

Using 4-bit numbers, multiply of $2_{10} \times 3_{10}$.

2 x 3 = ?

$0010_2 \times 0011_2$

Product (P)

Multiplier (MP)

Multiplicand (MC)

<u>Steps</u>:

1 – Test multiplier (0 or 1)

If 1 then 1a: P = P + MC

If 0 then no operation

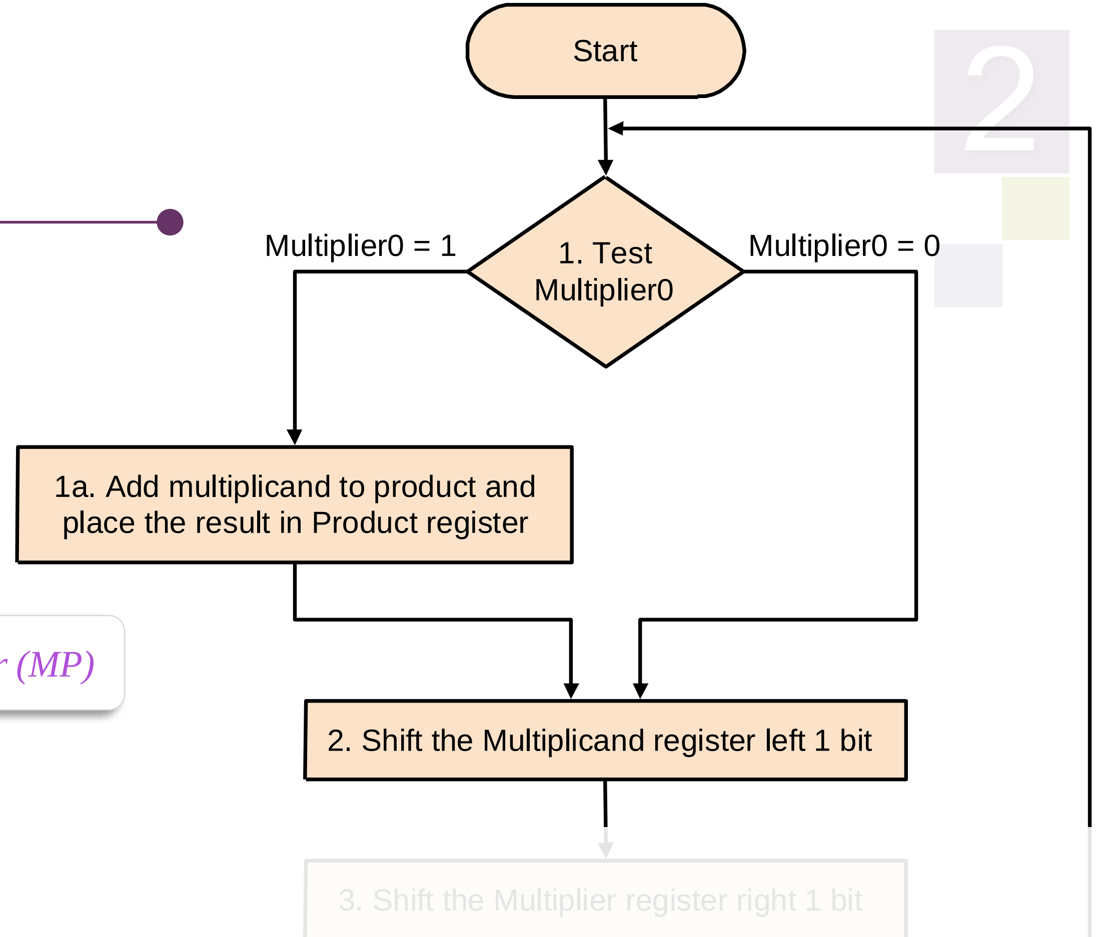
2 – Shift MC left

3 – Shift MP right

All bits done?

If still <max bit, repeat

If = max bit, stop



Max bit = Number of iteration.

→ Based on 4-bits number system used.

$2_{10} \times 3_{10} = \_\_\_\_\_\__{10}$

Iteration	Step	Multiplier (MP)	Multiplicand (MC)	Product (P)
0	Initial value	0011	0000 0010	0000 0000
1	1a: P = P + MC	0011		0000 0010
	2: Shift MC left		0000 0100	
	3: Shift MP right	0001		
2	1a: P = P + MC	0001		0000 0110
	2: Shift MC left		0000 1000	
	3: Shift MP right	0000		
3	1: No Operation	0000		
	2: Shift MC left		0001 0000	
	3: Shift MP right	0000		
4	1: No Operation	0000		
	2: Shift MC left		0010 0000	
	3: Shift MP right	0000		

If iter = max bit, stop

Answer:
0000 0110
$= 6_{10}$

Exercise 2.2:

In 4-bit binary arithmetic, find the multiplication of 5_{10} with 4_{10} using the 1^{st} version of highly optimized multiplication hardware.

Solution 2.2:

Iteration	Step	Multiplier (MP)	Multiplicand (MC)	Product (P)
0	Initial value			
1				
	2: Shift MC left			
	3: Shift MP right			
2				
	2: Shift MC left			
	3: Shift MP right			
3				
	2: Shift MC left			
	3: Shift MP right			
4				
	2: Shift MC left			
	3: Shift MP right			

Aside:

- The *multiplier* (MP) must always in positive number.
- Do an *additive inverse* to the *multiplicand* (MC) and the MP.

$$Multiplicand \times (-Multiplier) = (-Multicand) \times Multiplier$$
$$MC \times (-MP) = (-MC) \times MP$$

- **Examples**:

$$7 \times (-5) = (-7) \times 5$$
$$(-7) \times (-5) = 7 \times 5$$

Linda Null and Julia Lobur (2003). *The Essentials of Computer Organization and Architecture.* United States: Jones and Bartlett Publishers. p.53.

Example 15:

Using a 4-bit binary arithmetic, multiply 2_{10} with (-3_{10}) using the 1st version of highly optimized multiplication hardware.

Solution:

→ $2 \times (-3)$

- Do an *additive inverse* to the multiplicand (MC) and the MP:

$$(-2) \times 3$$

- Perform the multiplication as usual.

$(-2_{10}) \times 3_{10} = \underline{\hspace{3cm}}_{10}$

Iteration	Step	Multiplier (MP)	Multiplicand (MC)	Product (P)
0	Initial value			
1	2: Shift MC left			
	3: Shift MP right			
2	2: Shift MC left			
	3: Shift MP right			
3	2: Shift MC left			
	3: Shift MP right			
4	2: Shift MC left			
	3: Shift MP right			

$(-2_{10}) \times 3_{10} =$ __________$_{10}$

Iteration	Step	Multiplier (MP)	Multiplicand (MC)	Product (P)
0	Initial value	0011	1111 1110	0000 0000
1	1a: P = P + MC	0011		1111 1110
	2: Shift MC left		1111 1100	
	3: Shift MP right	0001		
2	1a: P = P + MC	0001		1 1111 1010
	2: Shift MC left		1111 1000	
	3: Shift MP right	0000		
3	1: No Operation	0000		
	2: Shift MC left		1111 0000	
	3: Shift MP right	0000		
4	1: No Operation	0000		
	2: Shift MC left		1110 0000	
	3: Shift MP right	0000		

discard

Check your answer:
1111 1010 (2s)

1) Flip the bit
1111 1010 -> 0000 0101
2) Add 1
0000 0101 + 1 =
0000 0110
$= 6_{10}$

1111 1010 is -6 in 2s

Activity 3

Exercise 2.3:

In 6-bit binary arithmetic, find the multiplication of 21_{10} with 14_{10} using the 1^{st} version of highly optimized multiplication hardware.

Solution 2.3:

Iteration	Step	Multiplier (MP)	Multiplicand (MC)	Product (P)
0	Initial value	001110	0000 0001 0101	0000 0000 0000
1				
	2: Shift MC left			
	3: Shift MP right			
2				
	2: Shift MC left			
	3: Shift MP right			
3				
	2: Shift MC left			
	3: Shift MP right			
4				
	2: Shift MC left			
	3: Shift MP right			

Iteration	Step	Multiplier (MP)	Multiplicand (MC)	Product (P)
5				
	2: Shift MC left			
	3: Shift MP right			
6				
	2: Shift MC left			
	3: Shift MP right			

If iter = max bit, stop

Activity 4

2

Exercise 2.4:

In 6-bit binary arithmetic, find the multiplication of 21_{10} with (-14_{10}) by using:

a) the two's complement binary numbers. Proof that it yields incorrect result.

b) the 1st version of highly optimized multiplication hardware.

Solution 2.4 (a):

The two's complement binary numbers.

Solution 2.4 (b):

The 1st version of highly optimized multiplication hardware.

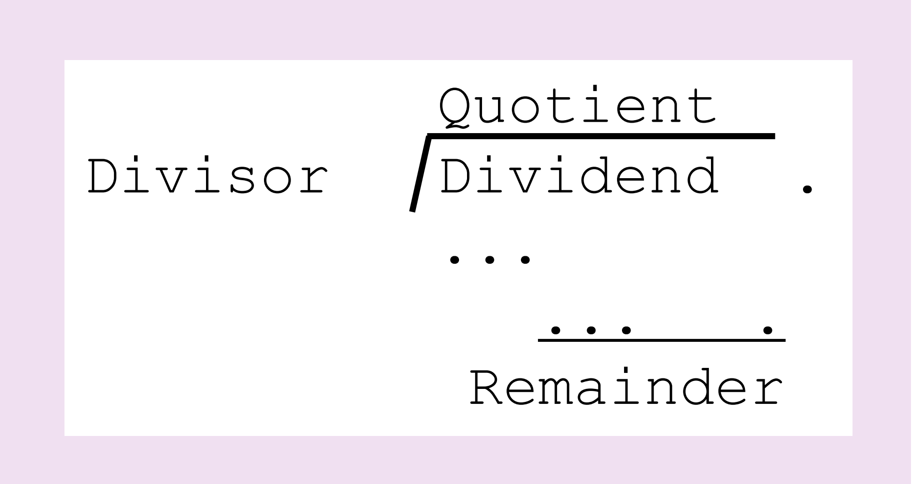
Iteration	Step	Multiplier (MP)	Multiplicand (MC)	Product (P)
0	Initial value	001110	0000 0001 0101	0000 0000 0000
1				
	2: Shift MC left			
	3: Shift MP right			
2				
	2: Shift MC left			
	3: Shift MP right			
3				
	2: Shift MC left			
	3: Shift MP right			
4				
	2: Shift MC left			
	3: Shift MP right			

Iteration	Step	Multiplier (MP)	Multiplicand (MC)	Product (P)
5				
	2: Shift MC left			
	3: Shift MP right			
6				
	2: Shift MC left			
	3: Shift MP right			

If iter = max bit, stop

(d) Division

- More complex than multiplication but is based on the same general principles.
- An operation that is even less frequent and even more quickly.
- It even offers the opportunity to perform a mathematically invalid operations in dividing by 0.
- Two operands called *dividend* and *divisor*, the result as *quotient* with secondary result called *remainder*.



- Another way to express the relationship between the components:

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

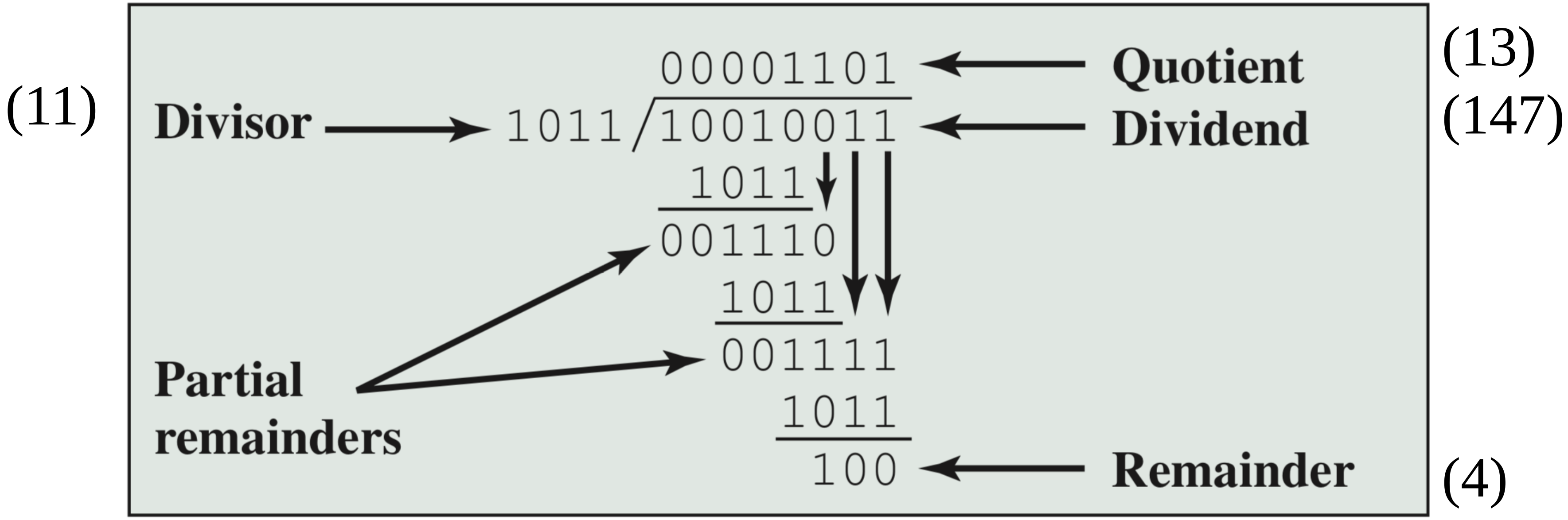
  where the *remainder* is smaller than the *divisor*.

- Infrequently, programs use the divide instruction just to get the *remainder*, ignoring the *quotient*.

(d) Division

Unsigned Integer

- The following figure shows an example of the long division of unsigned binary integers of 147_{10} divided by 11_{10}.



(d) Division

Signed Integer: Two's Complement

- Make both *dividend* and *divisor* positive and perform division.
- Make the sign of the *remainder* match to the *dividend*, no matter what the signs of the *divisor* and *quotient*.
- The rules:

Divisor

Dividend

$$+7 \div +2: Quotient = +3, Remainder = +1$$
$$+7 \div -2: Quotient = -3, Remainder = +1$$
$$-7 \div +2: Quotient = -3, Remainder = -1$$
$$-7 \div -2: Quotient = +3, Remainder = -1$$

- *Negate* the *quotient* if *dividend* and *divisor* were of opposite signs.

A Division Algorithm and Hardware

2

- Binary division is restricted to 0 or 1, thereby simplifying binary division.

- In the next slides, assume that both the *dividend* and *divisor* are positive number; Hence the *quotient* and *remainder* are non-negative.

- Since iteration of the algorithm needs to move the *divisor* to the right one digit, we start the *divisor* placed in the left half of the 64-bit Divisor Register.

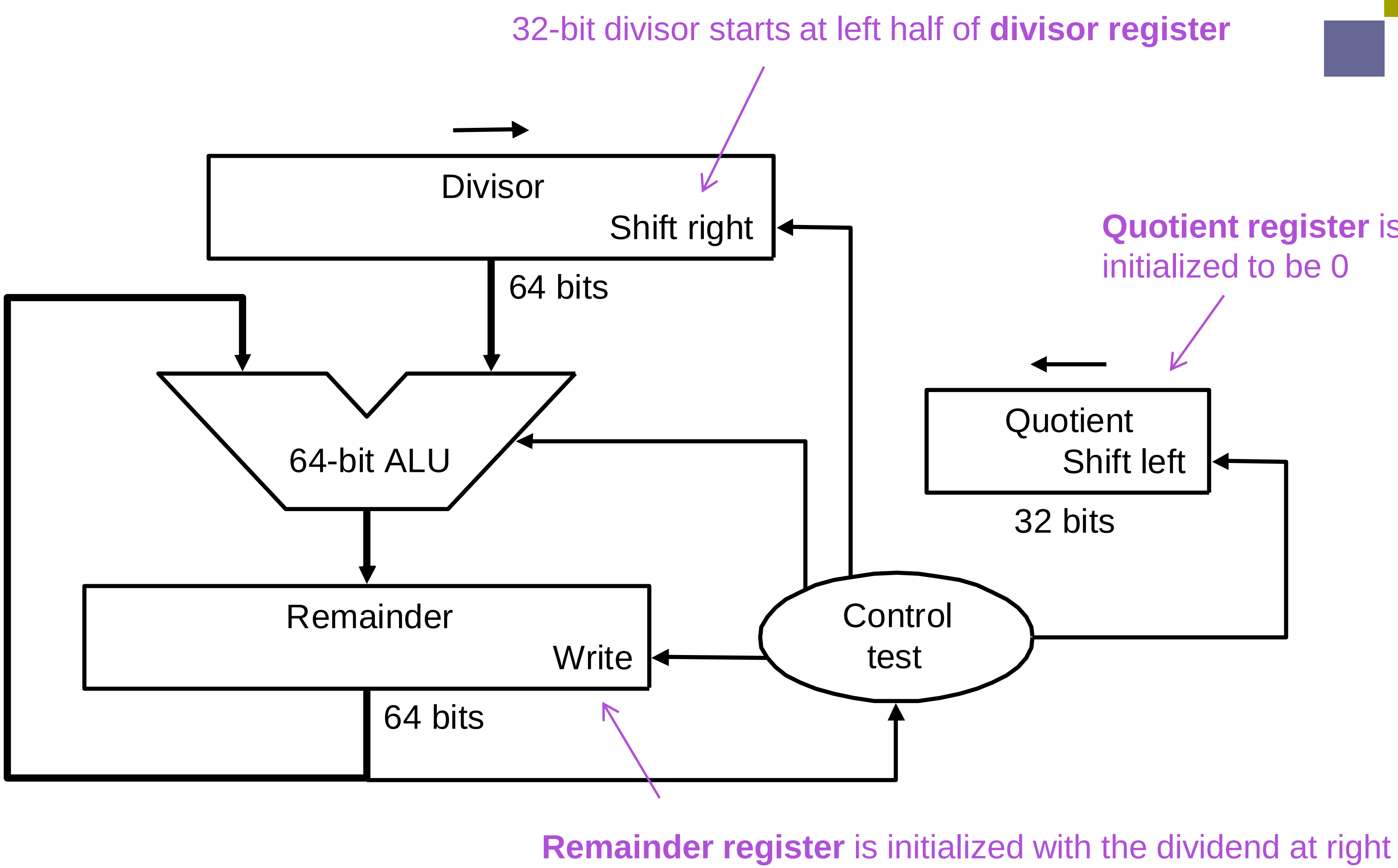


Figure: First version of Division Hardware.

Patterson, D.A. and Hennessy, J.L. (2014). *Computer Organization and Design: The Hardware/Software Interface (5th* Edition). United States: Elsevier, p.190.

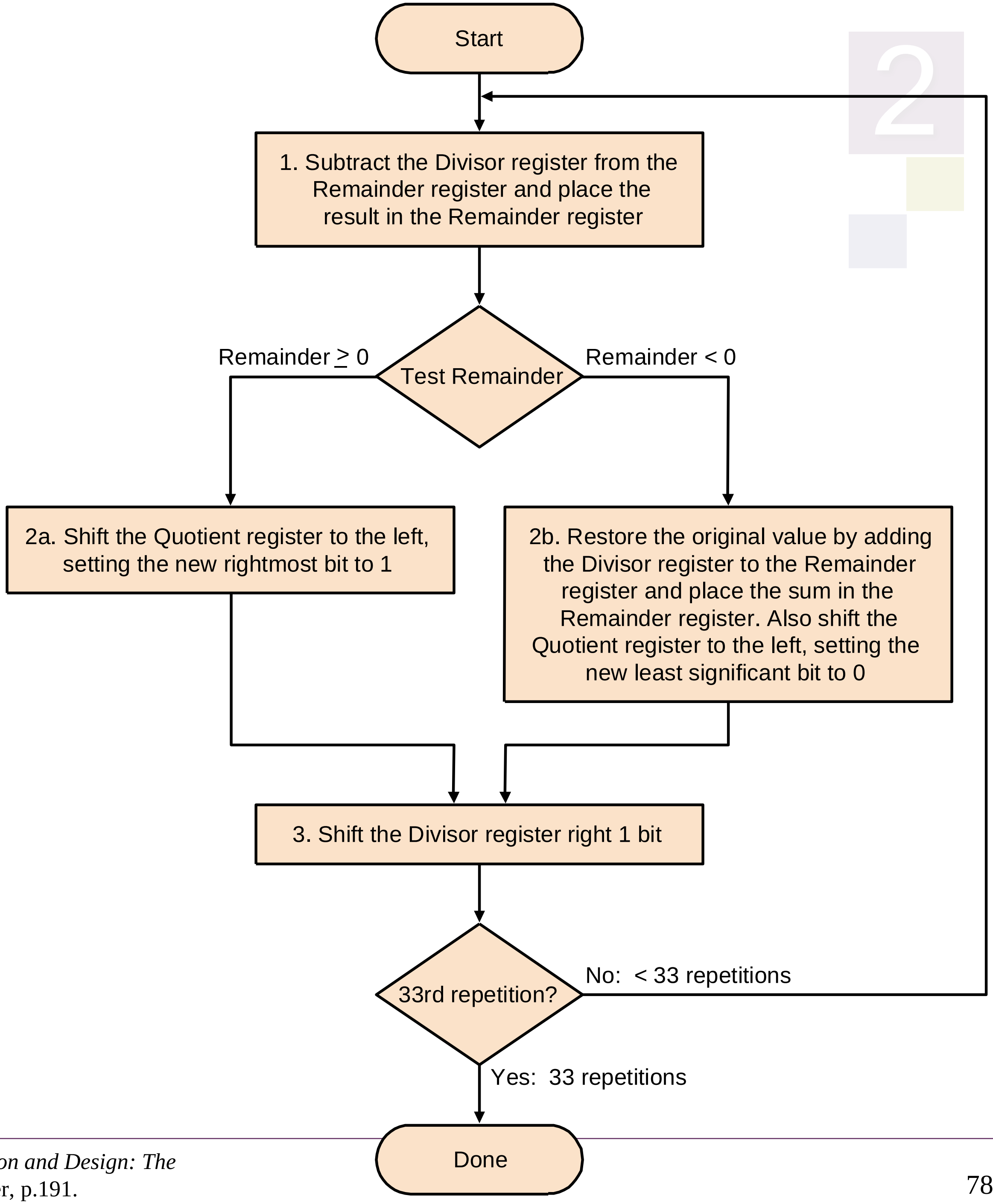


Figure:
The Division Algorithm using the Hardware.

Patterson, D.A. and Hennessy, J.L. (2014). *Computer Organization and Design: The Hardware/Software Interface* (5th Edition). United States: Elsevier, p.191.

Example 16:

Using 4-bit numbers, divide 7_{10} by 2_{10} .

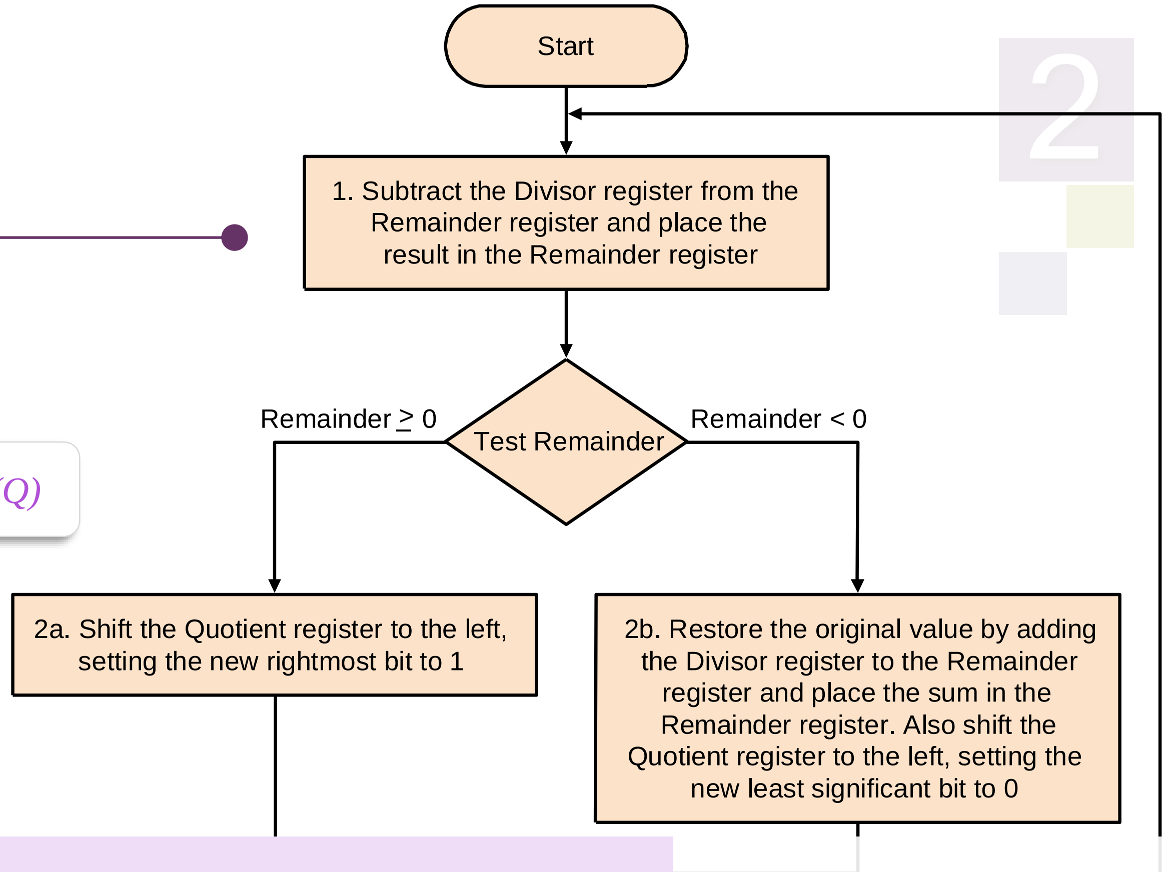
7 / 2 = ?

Quotient (Q)

0111_2 / 0010_2

Dividend (DD)

Divisor (D)



Steps:

1 – Remainder (R) = R – D

2 – Test new R

2a - If >=0 then Shift left Q (add 1 at LSB)

2b - If <0 then R = D + R, Shift left Q (add 0 at LSB)

3 – Shift D right

All bits done?

If still <(max bit + 1), repeat

If = (max bit + 1), stop

Max bit + 1 = Number of iteration.

→ Based on 4-bits number system used

Dividend (DD)

$7_{10} / 2_{10} =$ _____

Divisor (D)

$0111_2 / 0010_2$

Divisor start at left half of divisor register

2

Iteration	Steps	Quotient (Q)	Divisor (D)	Remainder (R)
0	Initial value	0000	0010 0000	0000 0111
1	1 : R = R – D			1110 0111
	3 : D = Shift right			
2	1 : R = R – D			
	3 : D = Shift right			
3	1 : R = R – D			
	3 : D = Shift right			

Remainder register is initialized with the dividend at right

$$R = R - D$$
$$= R + (-D)$$

```
  0000 0111    (7)
+ 1110 0000    (2's for D)
  1110 0111
```

7 / 2 = ____ $0111_2 / 0010_2$

Divisor start at left half of divisor register !

Iteration	Steps	Quotient (Q)	Divisor (D)	Remainder (R)
0	Initial value	0000	0010 0000	0000 0111
1	1 : R = R – D			1110 0111
	2b: R < 0; R = D+R			0000 0111
	Q : Shift Left (+0)	0000		
	3 : D = Shift right		0001 0000	
2	1 : R = R – D			1111 0111
	2b: R < 0; R = D+R			0000 0111
	Q : Shift Left (+0)	0000		
	3 : D = Shift right		0000 1000	
3	1 : R = R – D			1111 1111
	2b: R < 0; R = D+R			0000 0111
	Q : Shift Left (+0)	0000		
	3 : D = Shift right		0000 0100	

R=D+R
0010 0000 + 1110 0111
= 1 0000 0111

R = R –D = R+(-D)
0000 0111 + 1110 1111 (2s)
= 1111 0111

R=D+R
0001 0000 + 1111 0111
= 1 0000 0111

R = R –D = R+(-D)
0000 0111 + 1111 1000 (2s)
= 1111 1111

R=D+R
0000 1000 + 1111 111
= 1 0000 0111

Try to complete the table for the remaining iterations:

R = 0000 0111_2; Q = 0000_2; D = 0000 0100_2

Iteration	Steps	Quotient (Q)	Divisor (D)	Remainder (R)
4	1 : R = R – D			0000 0011
	2a: No Operation			
	Q : Shift Left (+1)	0001		
	3 : D = Shift right		0000 0010	
5	1 : R = R – D			0000 0001
	2a: No Operation			
	Q : Shift Left (+1)	0011		
	3 : D = Shift right		0000 0001	

R = R –D = R+(-D)
0000 0111 + 1111 1100 (2s)
= 1 0000 0011

R = R –D = R+(-D)
0000 0011 + 1111 1110 (2s)
= 1 0000 0001

If iter = (max bit + 1), stop

Answer: 7 / 2 = **3** remainder **1**

Exercise 2.5:

Using a 4-bit binary arithmetic, find the division of (-7_{10}) by 2_{10} with the 1st version of highly optimized division hardware.

We solve this by following the rules below - repeating the same steps as the division of 7 by 2 1)Take the absolute value of 7 and 2 and perform division. 2)Then change remainder sign as below. 3)Then the quotient will be negated at the end because -7 and 2 have opposite sign

- Make both *dividend* and *divisor* positive and perform division. 1
- Make the sign of the *remainder* match to the *dividend*, no matter what the signs of the *divisor* and *quotient*. 2
- The rules:

Divisor

Dividend

$$+7 \div +2: Quotient = +3, Remainder = +1$$
$$+7 \div -2: Quotient = -3, Remainder = +1$$
$$-7 \div +2: Quotient = -3, Remainder = -1$$
$$-7 \div -2: Quotient = +3, Remainder = -1$$

- *Negate* the *quotient* if *dividend* and *divisor* were of opposite signs. 3

Dividend (DD)

$-7_{10} / 2_{10} =$ ____

Divisor (D)

$0111_2 / 0010_2$

$1001 \quad 0010_2$

Divisor start at left half of divisor register

Iteration	Steps	Quotient (Q)	Divisor (D)	Remainder (R)
0	Initial value	0000	0010 0000	0000 0111
1	1 : R = R – D			1110 0111
				..
	3 : D = Shift right			
2	1 : R = R – D			
	3 : D = Shift right			
3	1 : R = R – D			
	3 : D = Shift right			

Remainder register is initialized with the dividend at right

Make both *dividend* and *divisor* positive and perform division

Negate the *quotient* if *dividend* and *divisor* were of opposite signs
3 becomes -3

Make the sign of the *remainder* match to the *dividend*, no matter what the signs of the *divisor* and *quotient*
1 becomes -1

Try to complete the table for the remaining iterations:

$R = 0000\ 0111_2$; $Q = 0000_2$; $D = 0000\ 0100_2$

R = R –D = R+(-D)
0000 0111 + 1111 1100 (2s)
= 1 0000 0011

Iteration	Steps	Quotient (Q)	Divisor (D)	Remainder (R)
4	1 : R = R – D			0000 0011
	2a: No Operation			
	Q : Shift Left (+1)	0001		
	3 : D = Shift right		0000 0010	
5	1 : R = R – D			0000 0001
	2a: No Operation			
	Q : Shift Left (+1)	0011		
	3 : D = Shift right		0000 0001	

R = R –D = R+(-D)
0000 0011 + 1111 1110 (2s)
= 1 0000 0001

If iter = (max bit + 1), stop

Answer: 7 / 2 = **3** remainder **1**

Dividend (DD)

$-7_{10} / 2_{10} =$ ______

Divisor (D)

Activity 5

Exercise 2.6:

Using a 4-bit binary arithmetic, find the division of the following numbers with the 1st version of highly optimized division hardware.

a) 6_{10} by 3_{10}

b) 6_{10} by (-3_{10})

c) (-12_{10}) by 5_{10}

Conclusion

- Unsigned integer vs signed integer
- The only arithmetic operation that a computer system does is **Addition**
  - Addition
  - Subtraction - addition with signed integers (negative numbers)
  - Multiplication - repetitive addition of product to multiplicand
  - Division - repetitive subtraction of dividend with divisor