

CHAPTER

1

INTRODUCTION



fit@hcmus

KHOA CÔNG NGHỆ THÔNG TIN
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

REVIEW

- Computer Generations
- Classes of Computers
- Terminology: wafer, chip, chipset
- 8 great ideas in Computer architecture

GENERATION OF DIGITAL COMPUTER

Generation	Time	Technology
1	1940 – 1956	Vacuum tubes
2	1956 – 1963	Transistors
3	1964 – 1971	Integrated Circuits
4	1971 – now	Microprocessors
5	Under Development	Parallel Processing/ Artificial intelligence

CLASS OF COMPUTERS

- Personal Computers
- Server Computers
- Super Computers
- Embedded Computers

Personal Computers

- General-purpose variety of software
- Subject to cost/performance tradeoff



Server Computers

- Network-based
- High capacity performance, reliability
- Range from small server to building size



Supper Computers



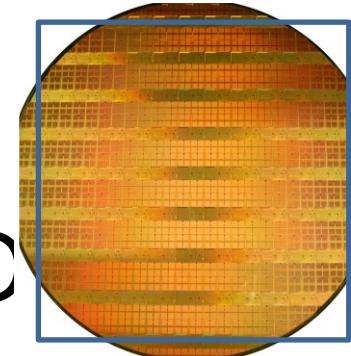
- High-end scientific and engineering calculations
- Highest capacity but represent a small fraction of the overall computer market

Embedded Computers

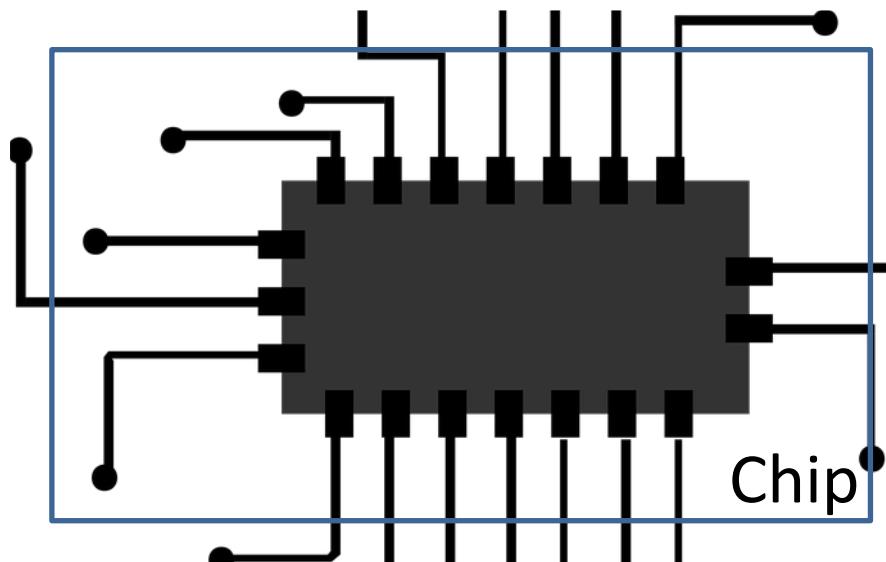
- Hidden components from the system
- Stringent power/performance/cost constraints
- Only work on a specific task



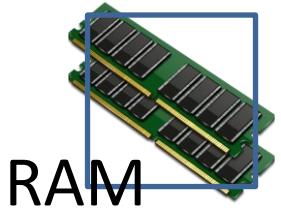
TERMINOLOGY



Wafer



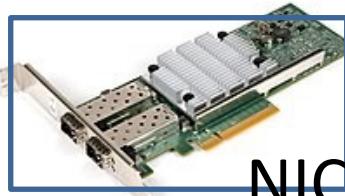
Chip



RAM



CPU

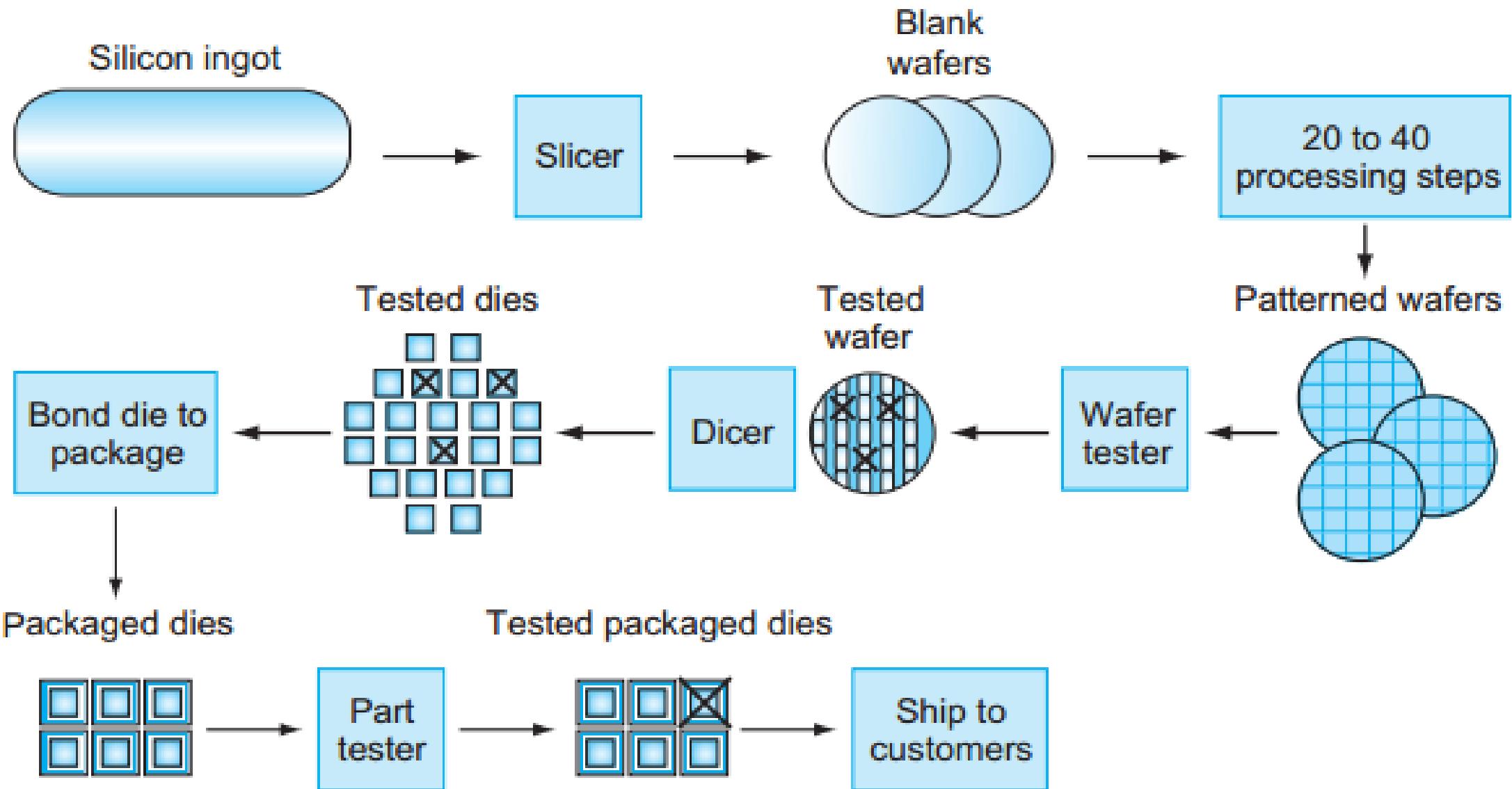


NIC



GPU

The chip manufacturing process



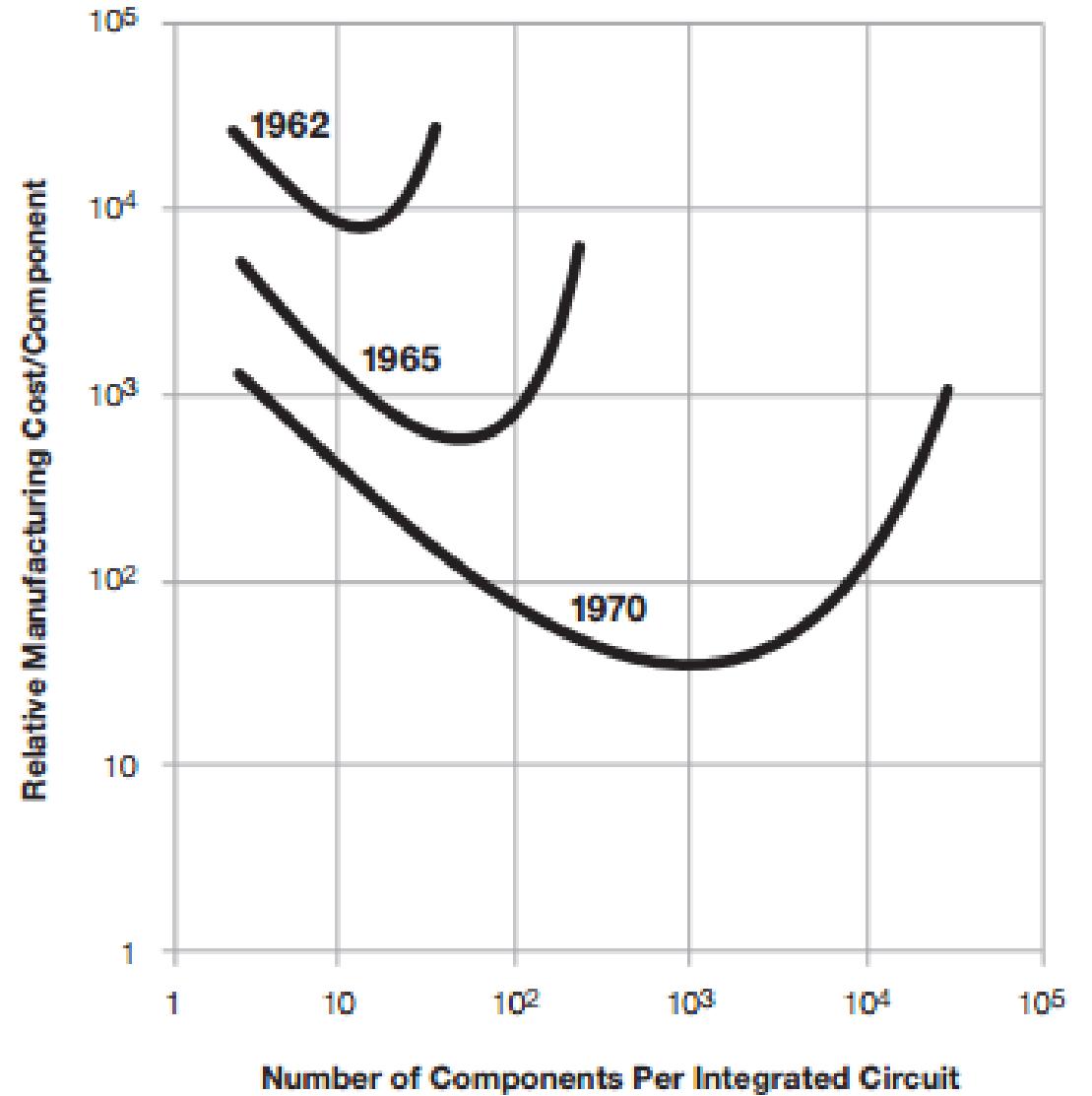
8 great ideas in Computer Architecture

- Design of Moore's law**
- Use abstraction to simplify design**
- Make a common case fast
- Performance via: Parallelism
- Performance via: Pipelining
- Performance via: Prediction
- Hierarchy of Memory**
- Dependability via Redundancy



Moore's Law

"The number of transistors incorporated in a chip will approximately double every 24 months."—Gordon Moore, Intel co-founder



- 01_Timeline.pdf
- 02_Hardware.pdf
- Patterson and Hennessy, **Computer Organization and Design: The Hardware / Software Interface (5th edition)**, Chapter 1





CHAPTER

1

INTRODUCTION



fit@hcmus

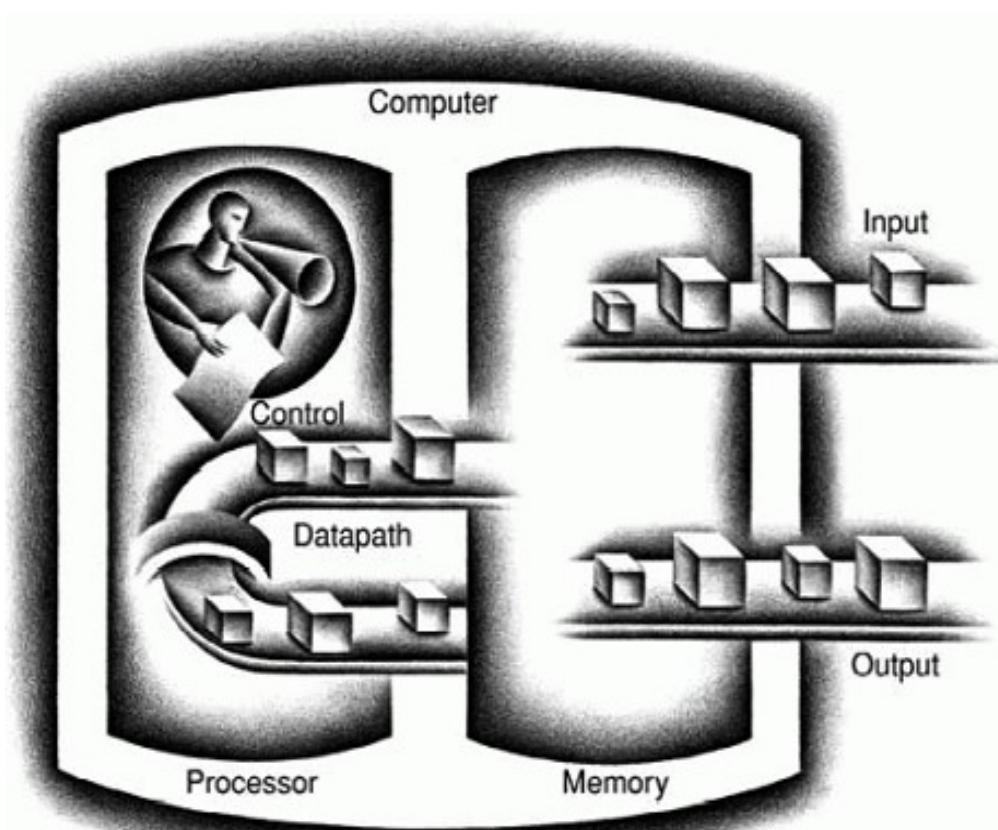
KHOA CÔNG NGHỆ THÔNG TIN
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

What will you learn?

- Basic components of a privacy computer (hardware interface)
- Inside a Processor
- Level of program code
- Abstraction layer (hardware & software)



Components of a Computer

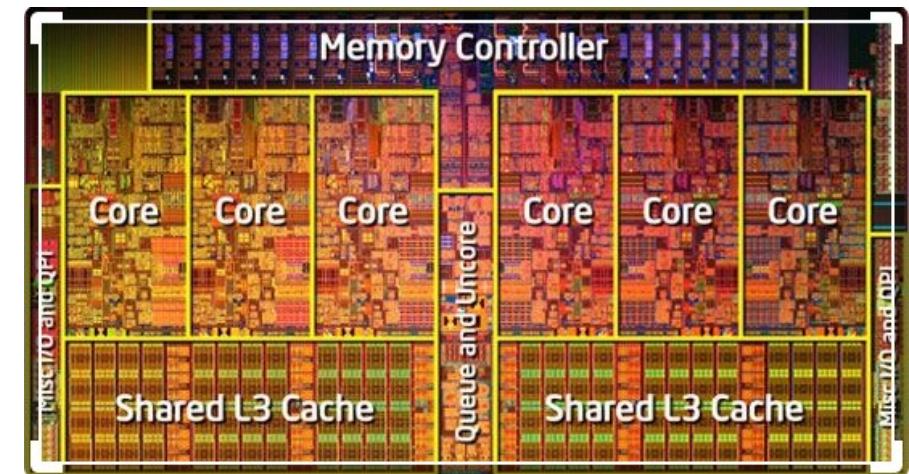


Same components of all kinds of computer

1. Input (mouse, keyboard, ...)
2. Output (display, printer, ...)
3. Memory:
Main memory (DRAM), cache (SRAM), secondary (disk, CD, DVD, ...)
4. Datapath } *Processor*
5. Control } *(CPU)*

Inside the Processor

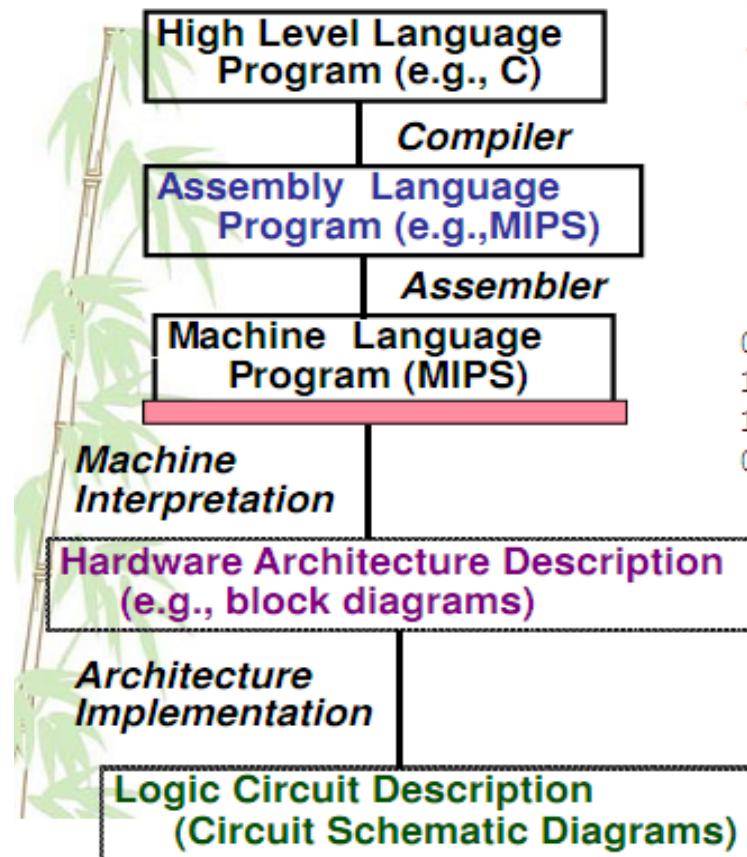
- Central Processing Unit - CPU
 - Datapath: performs operations on data
 - Control: sequences data-path, memory,
 - Cache memory:
 - Small fast SRAM memory for immediate access to data



<https://www.pinterest.com/pin/381117187187475687/visual-search/?x=10&y=10&w=530&h=274&cropSource=6>

The Inside of Intel Core i9 six core Processor

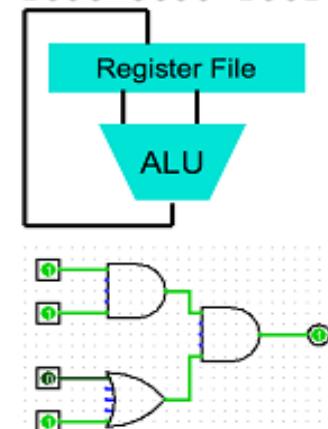
Level of Program Code



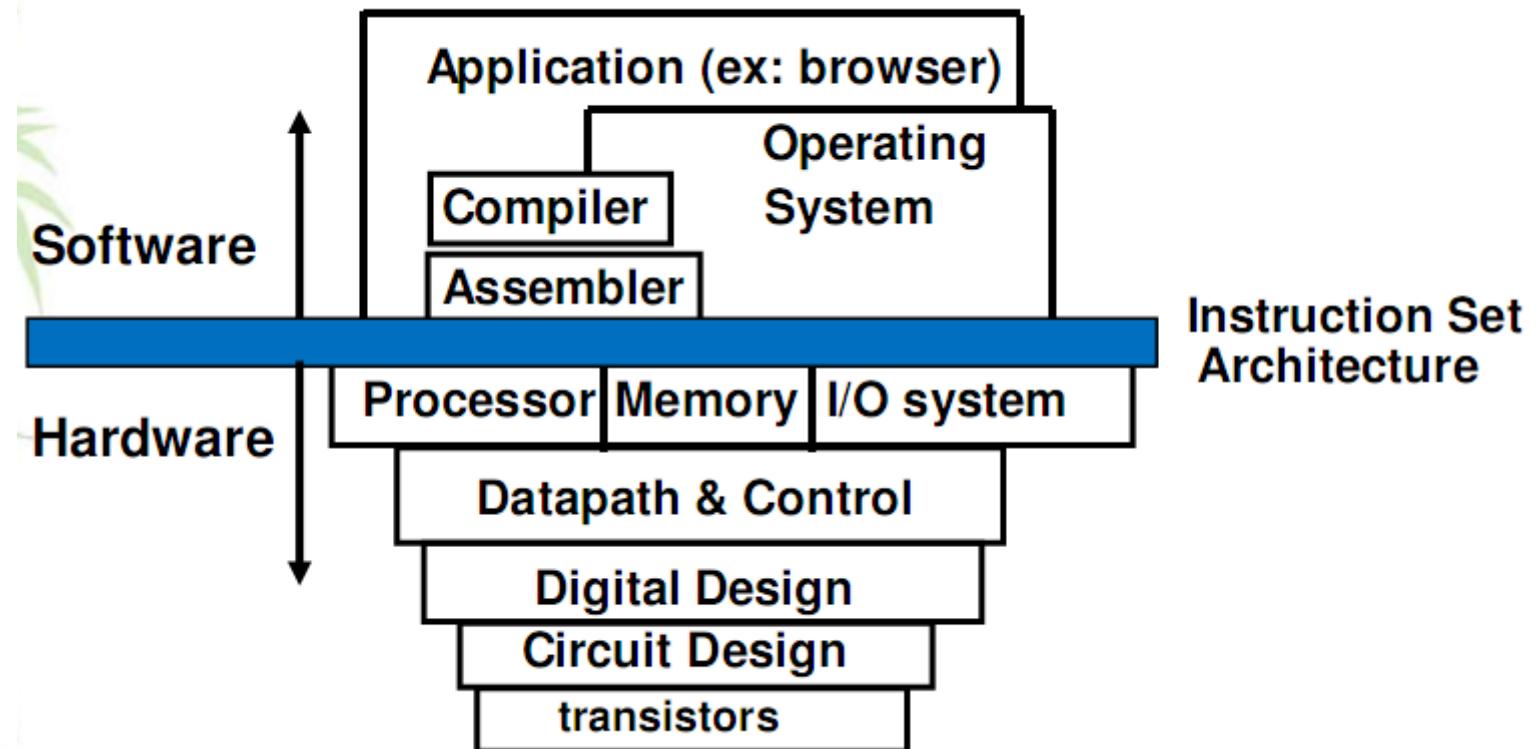
`temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;`

`lw $t0, 0($2)
lw $t1, 4($2)
sw $t1, 0($2)
sw $t0, 4($2)`

0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111



Abstraction layer



CHAPTER

1

INTRODUCTION



fit@hcmus

KHOA CÔNG NGHỆ THÔNG TIN
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

OUTLINE

- Common systems of number:
convert from a system of
number to the other (2,10,16)
- Integer representation &
comparison
- Operation with integer
- Dealing with overflow
- Character (ASCII, Unicode)
- Data format in C program
- Heterogeneous data
structures
- Data alignment



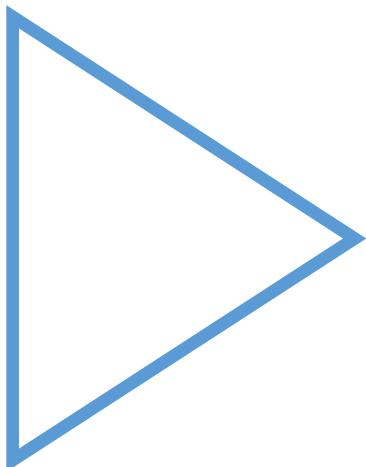
Number systems

- An unsigned integer with n digits in q-base system is represented in general formular:

$$x_{n-1} \dots x_1 x_0 = x_{n-1} \cdot q^{n-1} + \dots + x_1 \cdot q^1 + x_0 \cdot q^0$$

- The base system commonly represented in the computer is base 2

Number systems



Watch these videos:

- Introduction to number system and library
- Hexadecimal number system
- Convert from decimal to binary
- Converting larger number from decimal to binary
- Converting from decimal to hexadecimal representation
- Converting directly from binary to hexadecimal

Unsigned Integers representation

- Represent positive values such as length, weight, ASCII code, color code, ...
- The values of an unsigned integer is the magnitude of its underlying binary pattern
- Ex: Suppose that $n = 8$, the binary representation
10000001

Absolute value is **10000001** -> 129 (decimal)

Hence, the integer is **129** (decimal)

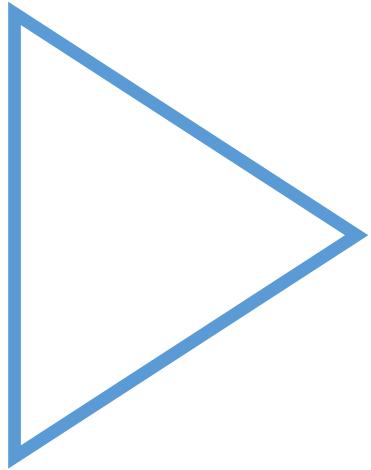
Unsigned Integers representation

- The n-bits binary pattern can represent the values from **0 to $2^{n-1} - 1$**

n	Minimum value	Maximum Value
8	0	$2^7 - 1 = 255$
16	0	$2^{16} - 1 = 65535$
32	0	$2^{32} - 1 = 4294967295$
64	0	$2^{64} - 1 = 18446744073709551615$



Unsigned Integers representation



Read the explanation in this link and watch these videos:

- The binary number system
- Converting the decimal numbers to binary (remind)
- Pattern in binary numbers

Take a practice

Signed Integers representation

Signed integers can represent zero, positive integers, as well as negative integers

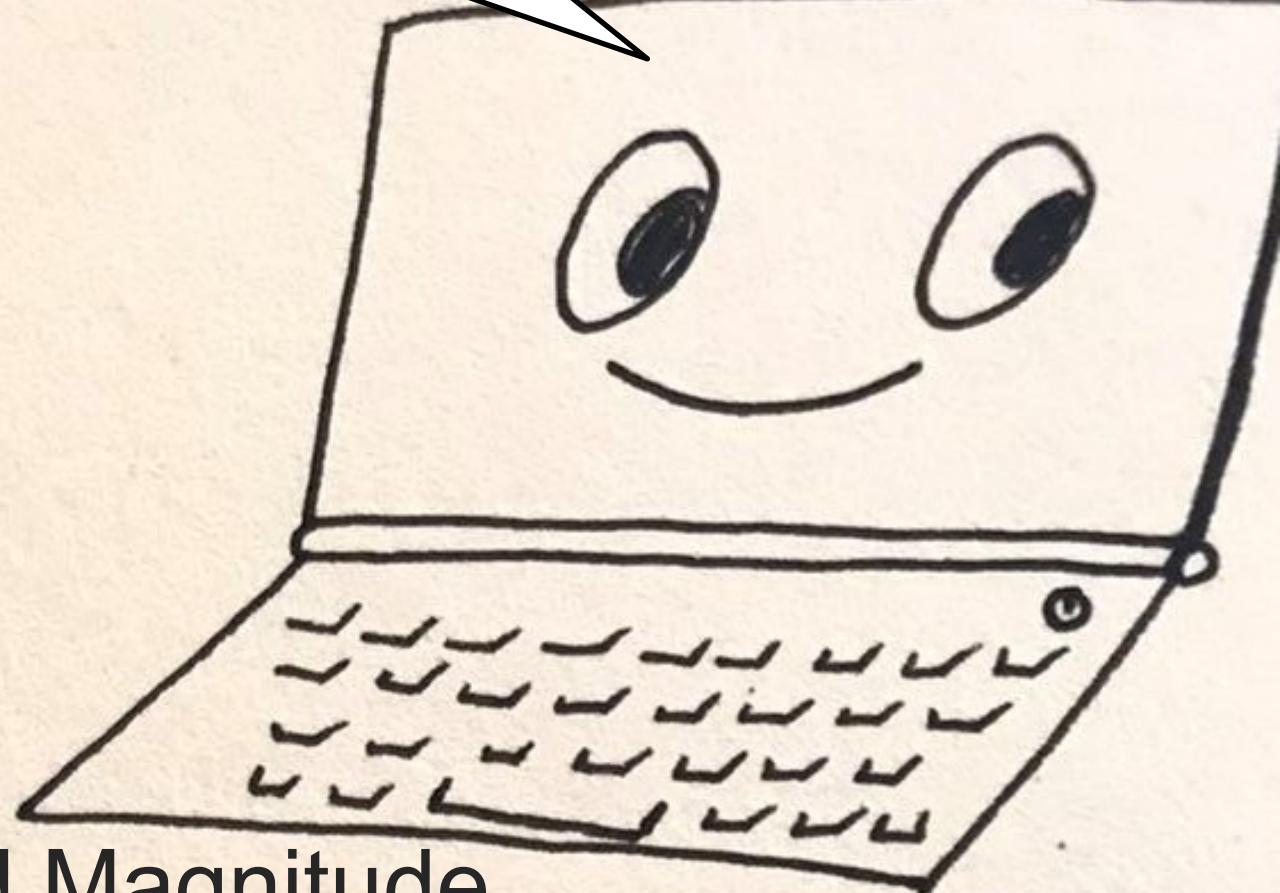
Four representation schemes are available for signed integers:

- Sign-Magnitude representation
- 1's Complement representation
- 2's Complement representation
- Bias (k-excess)

Signed numbers in the computer system are represented in the 2's Complement scheme



10000101



Signed Magnitude

Sign-Magnitude representation

- The most-significant bit (MSB) is the sign bit:
 - 0 -> positive integer
 - 1 -> negative integer
- The remaining $n-1$ bits represents the magnitude (absolute value) of the integer (n is the length of bit pattern)



Sign-Magnitude representation

- Ex: Suppose that $n = 8$, the binary representation
10000001

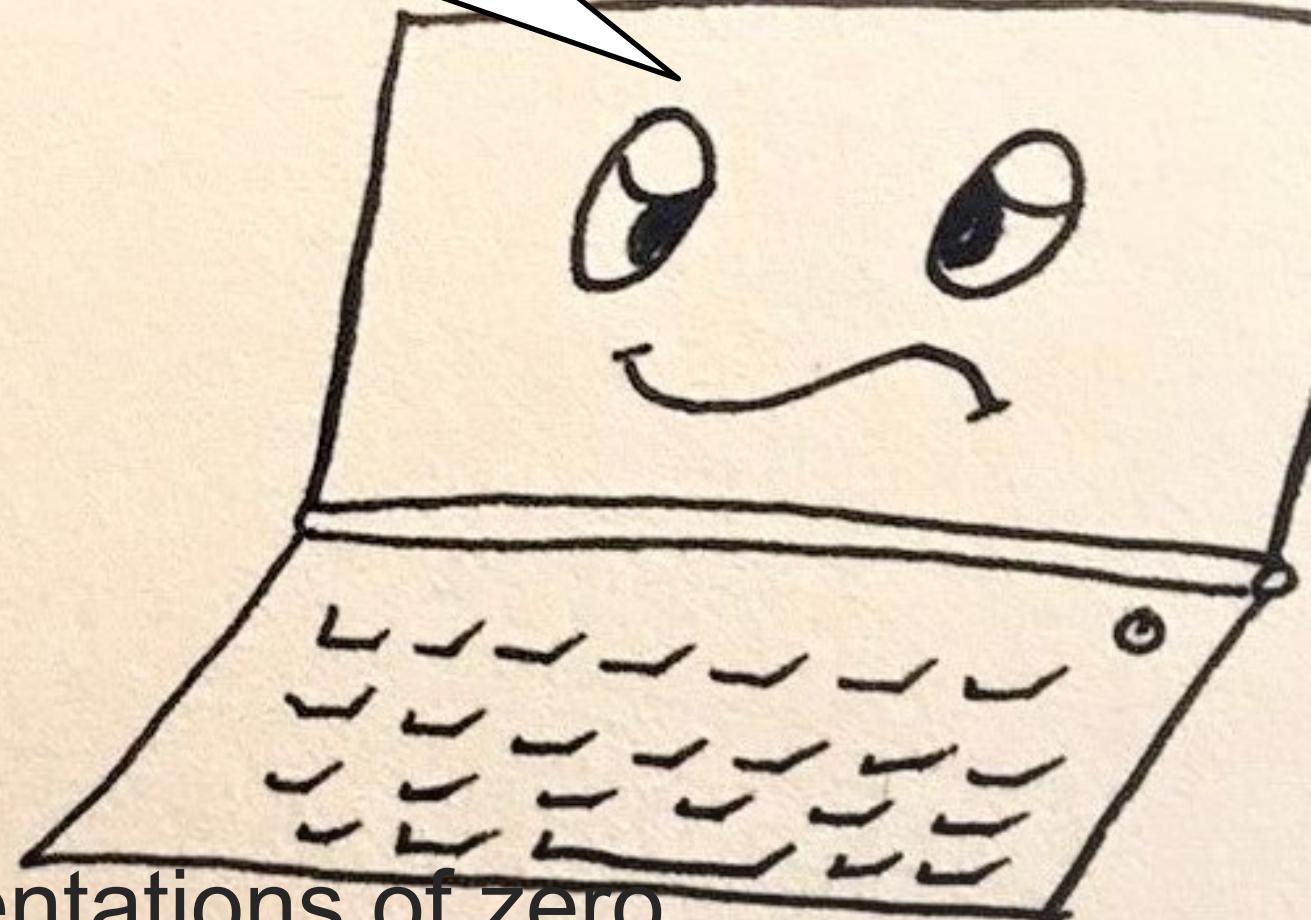
Sign bit is **1** -> negative number

Absolute value is **0000001**-> 1 (decimal)

Hence, the integer is -1 (decimal)



00000000 ?
10000000 ?



2 representations of zero

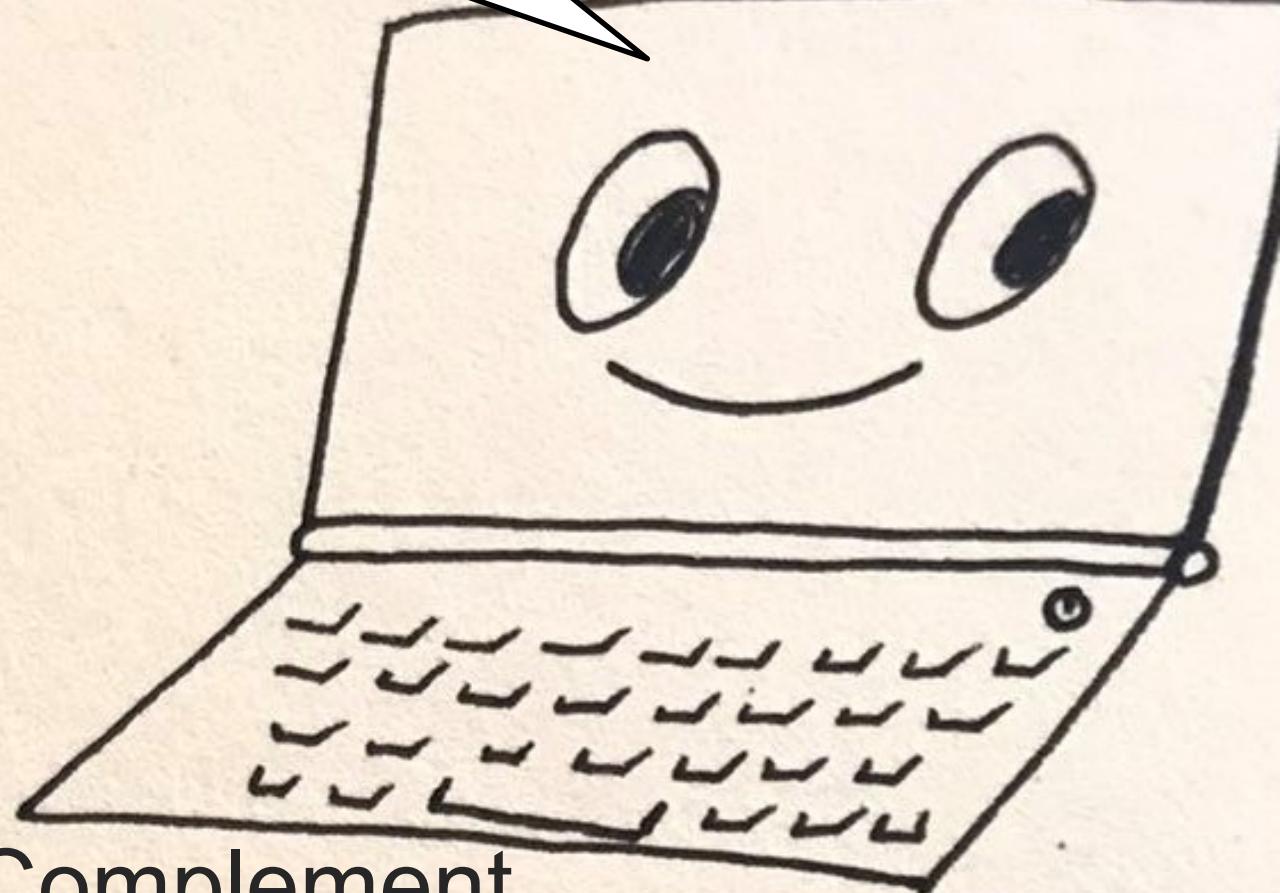
Sign-Magnitude representation

- The n-bits binary pattern can represent the values from $(-2^{n-1})+1$ to $2^n - 1 - 1$

Ex: Suppose that $n = 8$, the range of values is **-127 to 127**

- Positive number and negative number differ MSB value(sign bit), the absolute value are the same
- There are two representations for the number zero, which could lead to inefficiency and confusion.

11111010



One's Complement

One's Complement representation

- The most-significant bit (MSB) is the sign bit:
 - 0 -> positive integer
 - 1 -> negative integer
- The remaining $n-1$ bits represents the magnitude of the integer (n is the length of bit pattern) as follow:
 - positive integers**: the absolute value of the integer is equal to the magnitude of the $(n-1)$ -bit binary pattern
 - negative integers**: the absolute value of the integer is equal to the magnitude of the *complement (inverse)* of the $(n-1)$ -bit binary pattern

One's Complement representation

Ex: Suppose that $n = 8$, the binary representation **10000001**

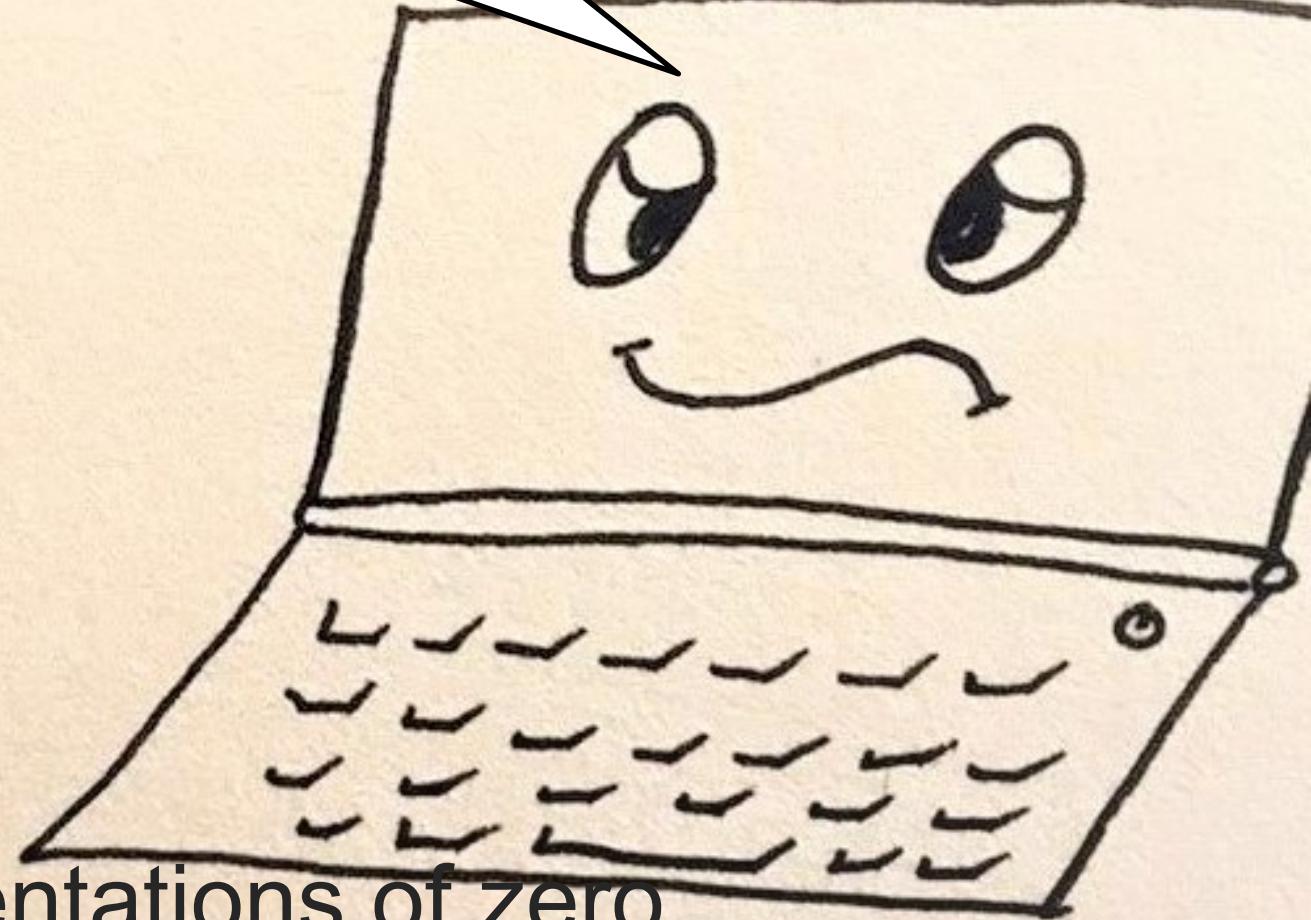
Sign bit is **1** -> negative number

Absolute value is the complement of **0000001** -> **1111110** ->
126 (decimal)

Hence, the integer is **-126** (decimal)



00000000 ?
11111111 ?



2 representations of zero

One's Complement representation

- The n-bits binary pattern can represent the values from $(-2^{n-1})+1$ to $2^{n-1} - 1$

Ex: Suppose that $n = 8$, the range of values is -127 to 127

- There are two representations for the number zero, which could lead to inefficiency and confusion.
- The positive integers and negative integers need to be processed separately



One's Complement representation

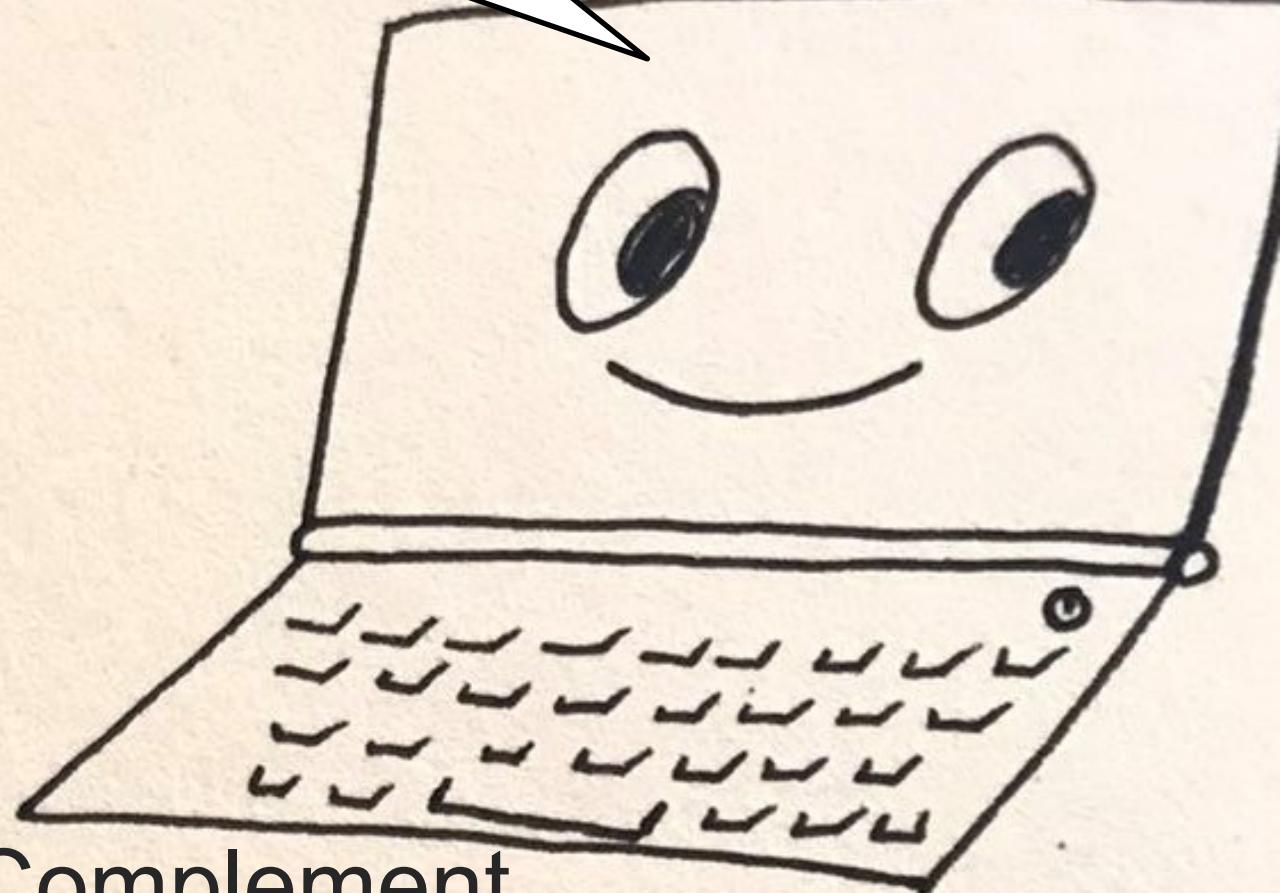
- The n-bits binary pattern can represent the values from $(-2^{n-1})+1$ to 2^{n-1}

Ex: Suppose that $n = 8$, the range of values is -127 to 127

- There are two representations for the number zero, which could lead to inefficiency and confusion.
- The positive integers and negative integers need to be processed separately



11111011



Two's Complement

Two's Complement representation

- The most-significant bit (MSB) is the sign bit:
 - 0 -> positive integer
 - 1 -> negative integer
- The remaining $n-1$ bits represents the magnitude of the integer (n is the length of bit pattern) as follow:
 - positive integers**: the absolute value of the integer is equal to the magnitude of the $(n-1)$ -bit binary pattern
 - negative integers**: the absolute value of the integer is equal to the magnitude of the *complement (inverse)* of the $(n-1)$ -bit binary pattern *plus one*

Two's Complement representation

Ex: Suppose that $n = 8$, the binary representation **10000001**

Sign bit is **1** -> negative number

Absolute value is the complement of **0000001 + 1**

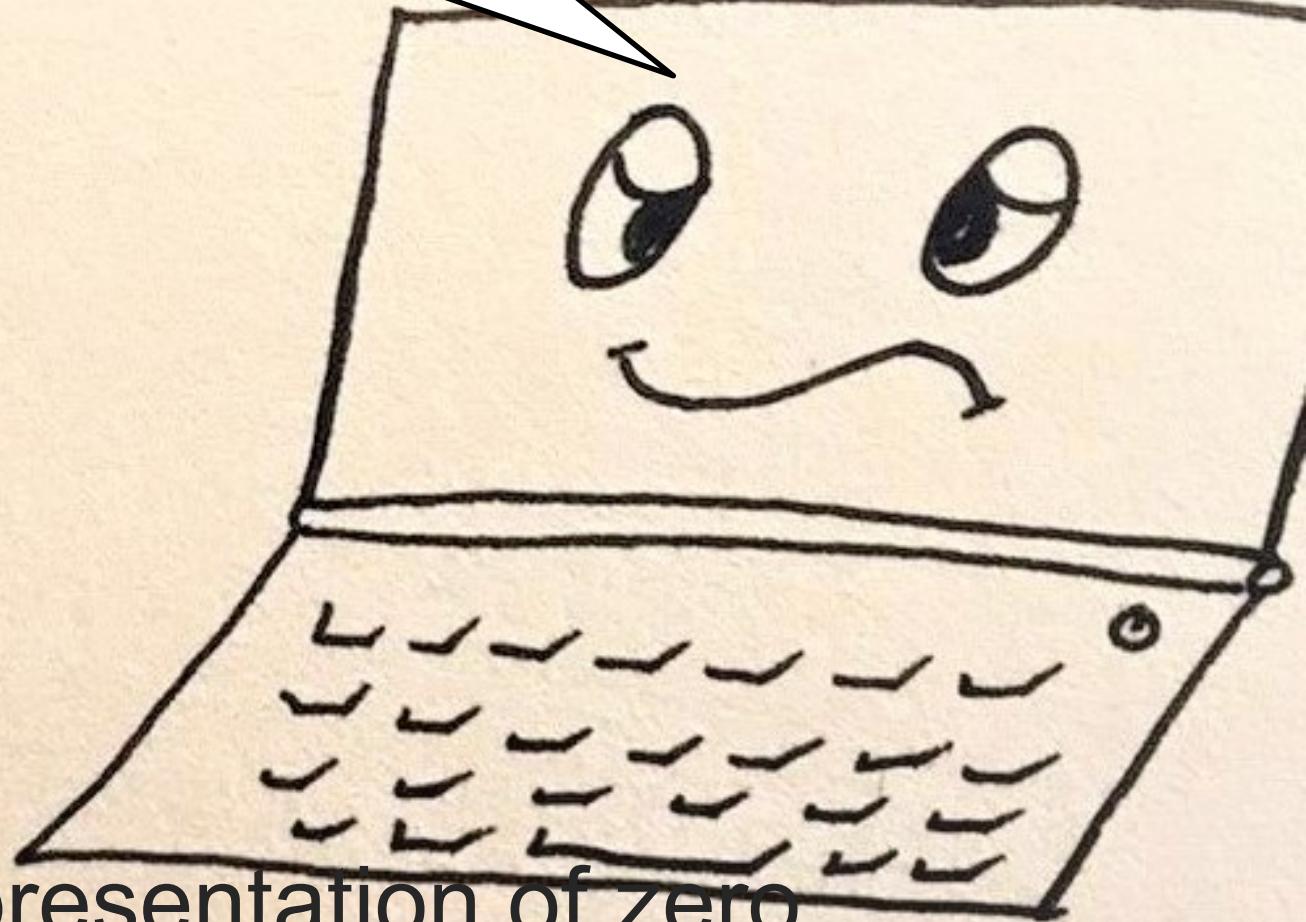
$$\rightarrow \textcolor{blue}{1111110} + 1$$

$$\rightarrow 127 \text{ (decimal)}$$

Hence, the integer is **-127** (decimal)



00000000
Any question?



Only one representation of zero

Two's Complement representation

- The n-bits binary pattern can represent the values from -2^{n-1} to $(2^{n-1})-1$

Ex: Suppose that $n = 8$, the range of values is -128 to 127

- There is one representations for the number zero
- The two's complement number of N is the negative form of N

Ex: How to represent the -5 (decimal) in binary:

The bit binary pattern of 5 is 00000101

The two's complement number of 5 is 11111010 plus 1



11111011

Two's Complement representation

Ex: Suppose that $n = 8$, the binary representation **10000001**

Sign bit is **1** -> negative number

Absolute value is ***the complement*** of **0000001 + 1**

$$\rightarrow \textcolor{blue}{1111110} + 1$$

$$\rightarrow \textcolor{blue}{1111111}$$

$$\rightarrow 127 \text{ (decimal)}$$

Hence, the integer is **-127** (decimal)



Bias
(k-excess) ????

Biased (K-excess)

- Choose K (a positive integer) to allows operations on the biased numbers to be the same as for unsigned integers, but represents both positive and negative values
- The remaining n bits represents the biased value (n is the length of bit pattern)
- The biased value is equal to the magnitude of the n-bits binary pattern
- The absolute value is equal to the bias value subtract/add to K
 - positive integers:** $N - K$
 - negative integers:** $N + K$



Biased (K-excess)

Ex: Suppose that $n = 8$, $K = 128$, the binary representation of N is **10000001** Biased value is 129 decimal (greater than K)

-> N is positive integer

-> Absolute value is $N - K = 129 - 128 = 1$
(decimal)

Hence, the integer is 1(decimal)



Biased (K-excess)

- The n-bits binary pattern can represent the values from -
 2^{n-1} to $2^{n-1} + 1$

Ex: Suppose that $n = 8$, $K = 128$, the range of values is -
128 to 127

- There is one representations for the number zero:
10000000
- Biased representations are now primarily used for the exponent of *floating-point numbers*



Biased (K-excess)

Ex: Suppose that $n = 8$, $K = 128$, how to represent a number in binary

Positive number: $N = 25$ (decimal)

$$N + K = 25 + 128 = 153$$

The bit binary pattern is $\rightarrow 10011001$

Negative number: $N = -25$ (decimal)

$$N + K = -25 + 128 = 103$$

The bit binary pattern is $\rightarrow 01100111$



Integer Operations

□ Logical operations

AND, OR, XOR, NOT

SHL, SHR, SAR

□ Arithmetic operation

Add/Subtract

Multiply

Division



Logical Operations

AND	0	1
0	0	0
1	0	1

OR	0	1
0	0	1
1	1	1

XOR	0	1
0	0	1
1	1	0

NOT	0	1
	1	0

$$\begin{array}{r} \text{AND} \\ \begin{array}{r} 11010011 \\ 00001111 \\ \hline 00000011 \end{array} \end{array}$$

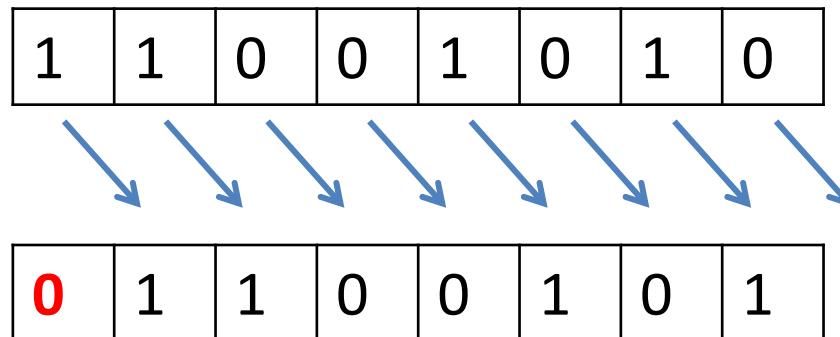
$$\begin{array}{r} \text{OR} \\ \begin{array}{r} 00000011 \\ 01100000 \\ \hline 01100011 \end{array} \end{array}$$

$$\begin{array}{r} \text{XOR} \\ \begin{array}{r} 01100011 \\ 01100011 \\ \hline 00000000 \end{array} \end{array}$$

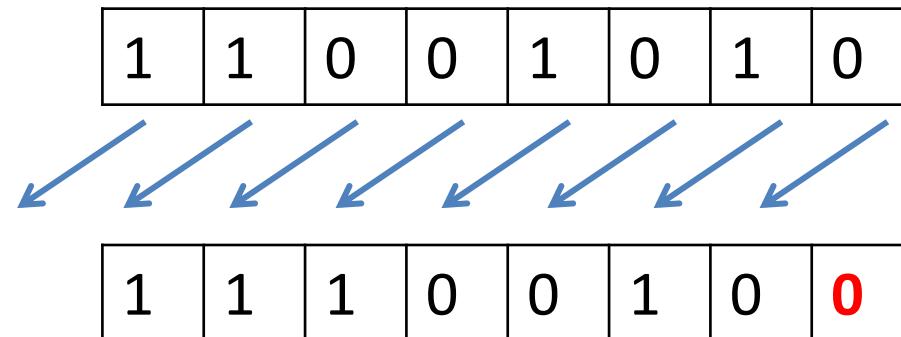
$$\begin{array}{r} \text{NOT} \\ \begin{array}{r} 11010011 \\ = 00101100 \end{array} \end{array}$$

Shift Operations

A logical shift moves bits to the left/ right and places a 0's form in the vacated bit on either end. The bits “fall off” will be discarded.



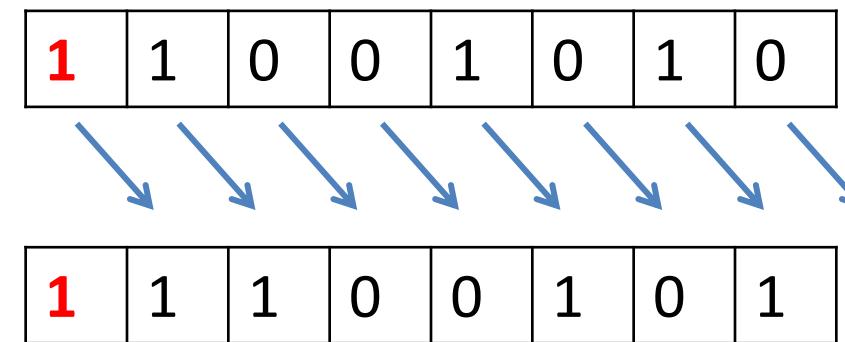
Shift Right Logical
(SHR)



Shift Left Logical (SHL)

Shift Operations

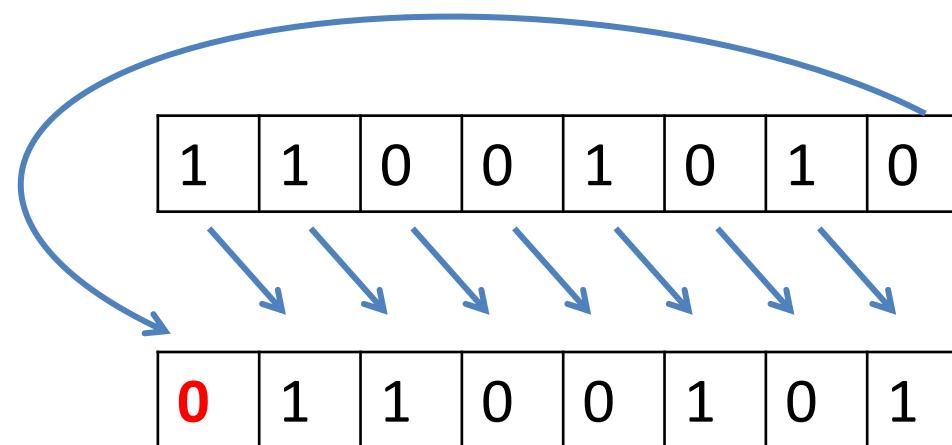
An arithmetic shift right preserves the sign bit



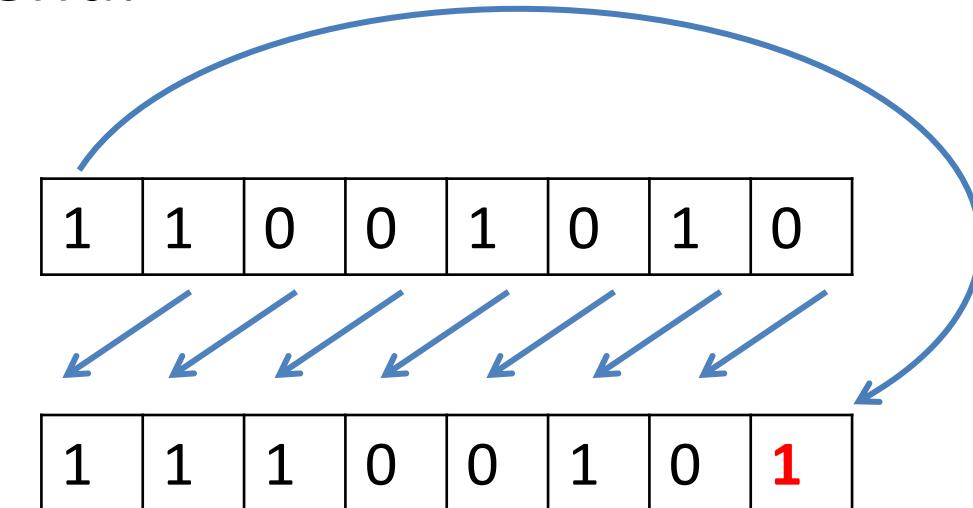
Shift Right Arithmetic
(SAR)

Shift Operations

A circular shift (rotate) places the bit shifted out of one end into the vacated position on the other end.



Rotate Right (ROR)



Rotate left (ROL)

Advanced

$x \text{ SHL } y = x . 2^y$

$x \text{ SAR } y = x / 2^y$

AND uses to switch off a bit(AND with 0 = 0)

OR uses to switch on a bit (OR with 1 = 1)

XOR, NOT uses to reverse a bit (bit i XOR with 1 = NOT(i))

$x \text{ AND } 0 = 0$

$x \text{ XOR } x = 0$

Advanced

Suppose that x is an integer:

- Get the value of bit i : $(x \text{ SHR } i) \text{ AND } 1$
- Set the value 1's of bit i : $(1 \text{ SHL } i) \text{ OR } x$
- Set the value 0's of bit i : $\text{NOT}(1 \text{ SHL } i) \text{ AND } x$
- Reverse bit i : $(1 \text{ SHL } i) \text{ XOR } x$



Integer Operations

□ Logical operations

AND, OR, XOR, NOT

SHL, SHR, SAR

□ Arithmetic operation

Add/Subtract

Multiply

Division

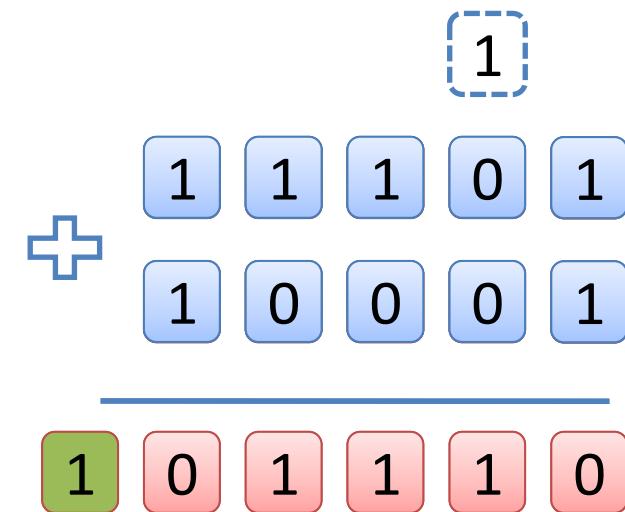


Add Operation

Rule:

$+$	0	1
0	0	1
1	1	10

Ex:



$$\begin{array}{r} 1001 = -7 \\ +0101 = 5 \\ \hline 1110 = -2 \end{array}$$

(a) $(-7) + (+5)$

$$\begin{array}{r} 1100 = -4 \\ +0100 = 4 \\ \hline 10000 = 0 \end{array}$$

(b) $(-4) + (+4)$

$$\begin{array}{r} 0011 = 3 \\ +0100 = 4 \\ \hline 0111 = 7 \end{array}$$

(c) $(+3) + (+4)$

$$\begin{array}{r} 1100 = -4 \\ +1111 = -1 \\ \hline 11011 = -5 \end{array}$$

(d) $(-4) + (-1)$

$$\begin{array}{r} 0101 = 5 \\ +0100 = 4 \\ \hline 1001 = \text{Overflow} \end{array}$$

(e) $(+5) + (+4)$

$$\begin{array}{r} 1001 = -7 \\ +1010 = -6 \\ \hline 10011 = \text{Overflow} \end{array}$$

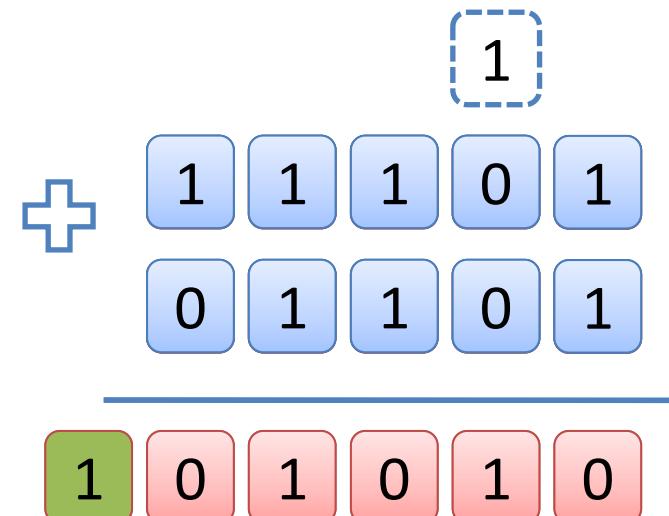
(f) $(-7) + (-6)$

Subtract Operation

Rule:

$$A - B = A + (-B) = A + (\text{the 2's complement of } B)$$

Ex: $11101 - 10011 = 11101 + 01101$



$$\begin{array}{r} 0010 = 2 \\ +1001 = -7 \\ \hline 1011 = -5 \end{array}$$

(a) $M = 2 = 0010$
 $S = 7 = 0111$
 $-S = 1001$

$$\begin{array}{r} 0101 = 5 \\ +1110 = -2 \\ \hline 10011 = 3 \end{array}$$

(b) $M = 5 = 0101$
 $S = 2 = 0010$
 $-S = 1110$

$$\begin{array}{r} 1011 = -5 \\ +1110 = -2 \\ \hline 11001 = -7 \end{array}$$

(c) $M = -5 = 1011$
 $S = 2 = 0010$
 $-S = 1110$

$$\begin{array}{r} 0101 = 5 \\ +0010 = 2 \\ \hline 0111 = 7 \end{array}$$

(d) $M = 5 = 0101$
 $S = -2 = 1110$
 $-S = 0010$

$$\begin{array}{r} 0111 = 7 \\ +0111 = 7 \\ \hline 1110 = \text{Overflow} \end{array}$$

(e) $M = 7 = 0111$
 $S = -7 = 1001$
 $-S = 0111$

$$\begin{array}{r} 1010 = -6 \\ +1100 = -4 \\ \hline 10110 = \text{Overflow} \end{array}$$

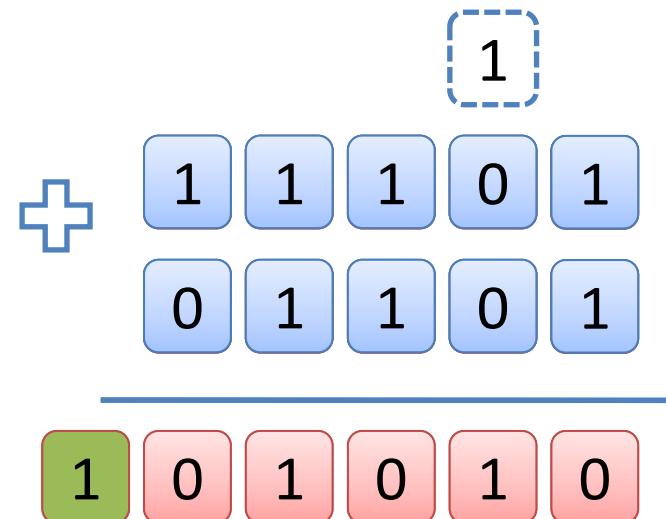
(f) $M = -6 = 1010$
 $S = 4 = 0100$
 $-S = 1100$

Subtract Operation

Rule:

$$A - B = A + (-B) = A + (\text{the 2's complement of } B)$$

Ex: $11101 - 10011 = 11101 + 01101$



Relational of integer and two's complement addition.

When $x + y < -2^{w-1}$,
there is a *negative* overflow

When $x + y > 2^{w-1} + 1$,
there is a *positive* overflow

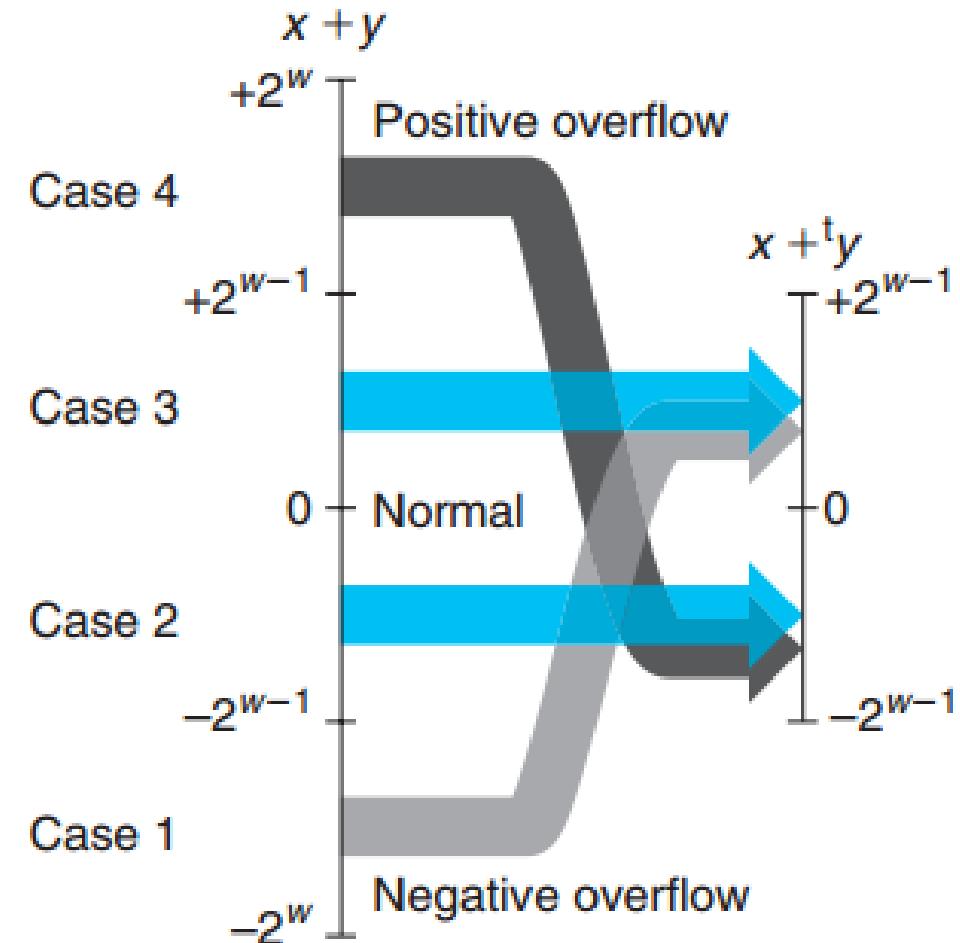


Photo by Chap2, Prentice.Hall.Computer.Systems.A.Programmers.Perspective.2nd.2011

Multiply Operation

Rule:

X	0	1
0	0	0
1	0	1

Ex:

$$\begin{array}{r} \times \\ \begin{array}{ccccc} 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 \end{array} \\ \hline \begin{array}{ccccc} 0 & 0 & 0 & 0 & 0 \end{array} \end{array}$$

$$\begin{array}{r} + \\ \begin{array}{ccccc} 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \end{array} \\ \hline \begin{array}{cccccc} 1 & 1 & 0 & 0 & 1 & 1 & 0 \end{array} \end{array}$$

Multiply Operation

$$\begin{array}{r} \textcolor{red}{\overbrace{}^{1011}} \\ \times \textcolor{red}{\overbrace{}^{1101}} \\ \hline 1011 \\ 0000 \\ 1011 \\ \hline 1011 \\ \hline 10001111 \end{array} \quad = 11 \quad = 13 \quad = 143$$

$$\begin{array}{r} 1011 \\ \times 1101 \\ \hline 00000000 \\ + 1011 \\ \hline 00001011 \\ + 0000 \\ \hline 00001011 \\ + 1011 \\ \hline 00110111 \\ + 1011 \\ \hline 10001111 \end{array}$$

Multiply Algorithm

- Suppose that Q's binary pattern has n-bits length, $M \times Q$
M: multiplicand
Q: multiplier
- Variable definition :
 - C (1 bit): carry bit
 - A (n bit): a part of the result ($[C, A, Q]$: product)
 - $[C, A]$ (n + 1 bit) ; $[C, A, Q]$ (2n + 1 bit): considered as compound registers



Multiply Algorithm

Initialize: $[C, A] = 0; k = n$

While $(k > 0)$

{

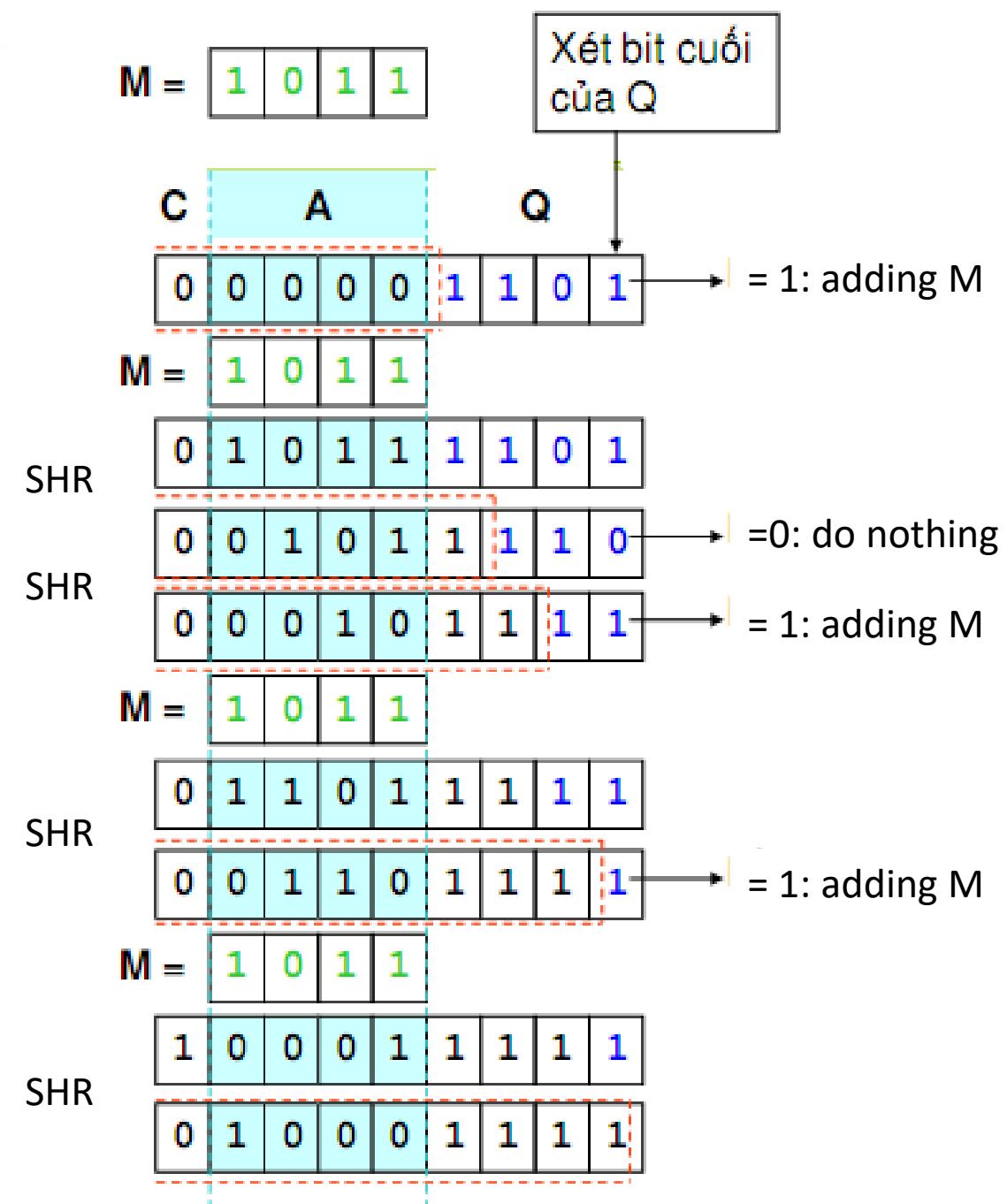
If LSB bit of Q equal to 1 then
 $(A + M) \rightarrow [C, A]$

SHR $[C, A, Q]$ 1 bit

$k = k - 1$

}

Return $[C, A, Q]$



Booth's Multiplication Algorithm

- A multiplication algorithm that multiplies two signed binary numbers in 2's complement notation
- Suppose that Q's binary pattern has n-bits length, $M \times Q$

M: multiplicand

Q: multiplier

- Variable definition :
 - A (n bit): a part of the result
 - $[A, Q]$: product
- Q_0 (1 bit): the LSB bit of Q
- $[Q, Q_{-1}]$ (n + 1 bit)

```
Initialize: A = 0; k = n; Q-1 = 0
#Add 1-bit Q-1 in the end of Q
While (k > 0)
{
    If Q0Q-1
    {
        = 10 then A - M -> A
        = 01 thi A + M -> A
        = 00, 11 then A -> A
    # Ignore any overflow
    }
    SAR[A, Q, Q-1] 1 bit
    k = k - 1
}
Return [A, Q]
```

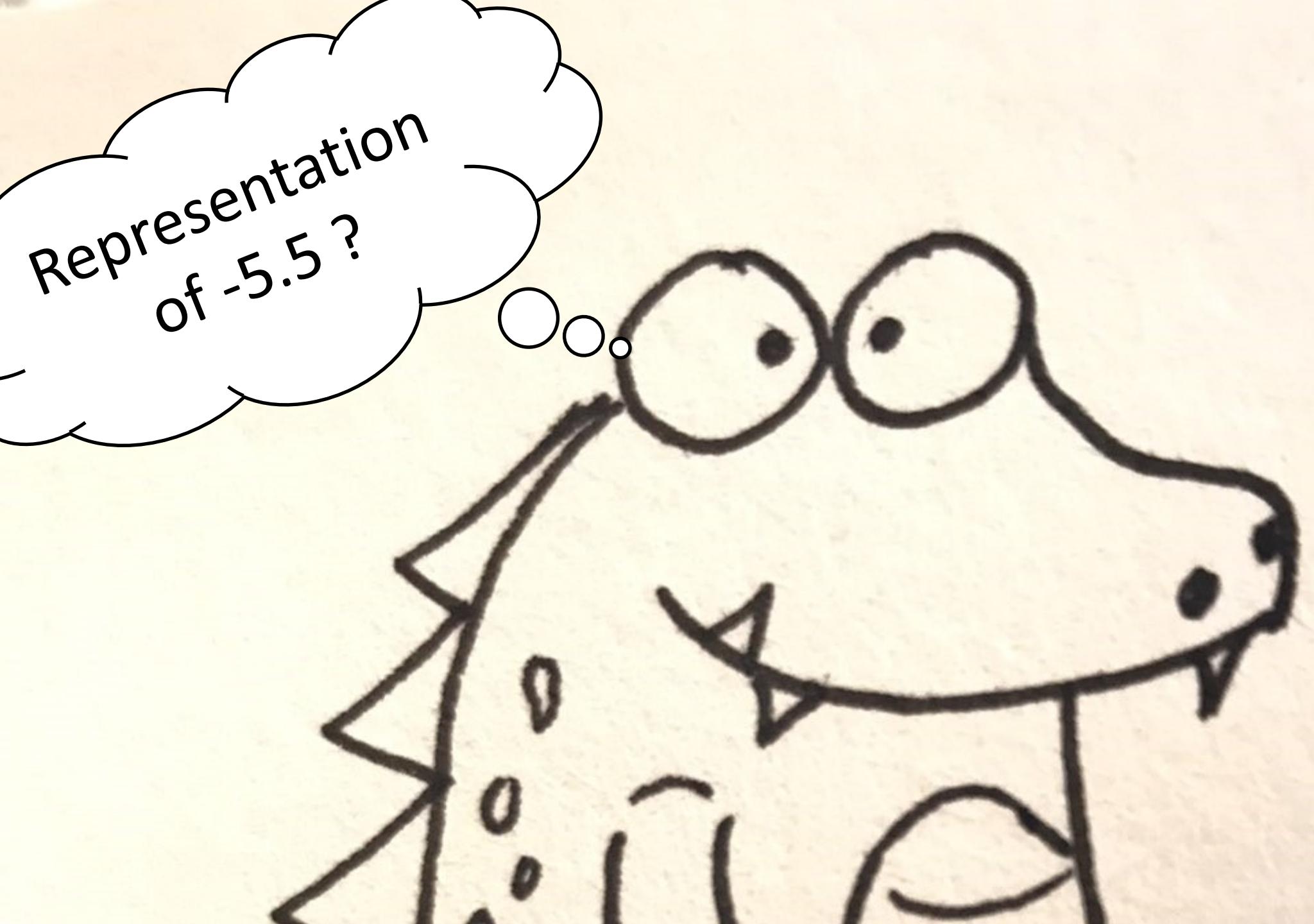
Ex: Suppose that n = 4, M = 7, Q =

-3		A	Q	Q ₋₁	M
	Initialize	0000	1101	0	0111
k = 4	A = A-M	1001	1101	0	0111
	SAR	1100	1110	1	0111
k = 3	A = A+M	0011	1110	1	0111
	SAR	0001	1111	0	0111
k = 2	A = A-M	1010	1111	0	0111
	SAR	1101	0111	1	0111
k = 1	SAR	1110	1011	1	0111

A	Q	M = 0011	A	Q	M = 1101
0000	0111	Initial value	0000	0111	Initial value
0000	1110	Shift	0000	1110	Shift
1101		Subtract	1101		Add
0000	1110	Restore	0000	1110	Restore
0001	1100	Shift	0001	1100	Shift
1110		Subtract	1110		Add
0001	1100	Restore	0001	1100	Restore
0011	1000	Shift	0011	1000	Shift
0000		Subtract	0000		Add
0000	1001	Set $Q_0 = 1$	0000	1001	Set $Q_0 = 1$
0001	0010	Shift	0001	0010	Shift
1110		Subtract	1110		Add
0001	0010	Restore	0001	0010	Restore

(a) (7)/(3)

(b) (7)/(-3)



Representation
of -5.5 ?

Fixed point numbers

Represent 123.375_{10} in 2-base system ?

Idea: Represent the integer part and fraction part separately

Integer part: use 8-bits for representation. Range of values is [0, 255] (decimal)

$$123_{10} = 64 + 32 + 16 + 8 + 2 + 1 = 0111\ 1011_2$$

Fraction part: use 8-bits for representation.

$$0.375 = 0.25 + 0.125 = 2^{-2} + 2^{-3} = 0110\ 0000_2$$

Signed fixed point	Signed bit	Integer (8-bits)	Fraction (8-bits)
0		0111 1011	0110 0000

Formular: $x_{n-1}x_{n-2}\dots x_0.x_{-1}x_{-2}\dots x_{-m} = x_{n-1} \cdot 2^{n-1} + x_{n-2} \cdot 2^{n-2} \dots + x_0 \cdot 2^0 + x_{-1} \cdot 2^{-1} + x_{-2} \cdot 2^{-2} + \dots + x_{-m} \cdot 2^{-m}$



Fixed point numbers

- Suppose that $n = 8\text{-bits}$

Largest integer value can be represented: 255

Smallest fraction value can be represented: $2^{-8} \sim 10^{-3} = 0.001$

- **Problem:** limited range of values can be represented, it does not allow enough numbers and accuracy
- **Solution:** *Floating point Number*



Floating point number

- Express in the follow notation: $F \times 2^E$ (with: F: fraction, E: Exponent, radix of 2)
- IEEE-754 standard: modern computer use to represent floating point number in form: $V = (-1)^S \times F \times 2^E$



- **Sign:** 1: Negative, 0: Positive
- **Exponent:** saved in n-bits pattern. Represented in K-excess form with
 - Single precision: K = 127 ($2^{n-1} - 1 = 2^{8-1} - 1$)*
 - Double precision: K = 1023 ($2^{n-1} - 1 = 2^{11-1} - 1$)*
- **Significand (Fraction):** the remaining bits after dot sign

IEEE-754 standard

Single precision (32-bits)



Double precision (64-bits)



Ex: Represent $X = -5.25$ in single precision scheme

Step 1: Convert X to binary system

$$X = -5.25_{10} = -101.01_2$$

Step 2: Normalize X in this form $\pm 1.F \times 2^E$

$$X = -5.25 = -101.01 = -1.0101 \times 2^2$$

Step 3: Represent X in floating point

Signed bit = **1** (Negative number)

Exponent= represent E in K-excess form (with K = 127)

$$\rightarrow \text{Exponent} = E + 127 = 2 + 127 = 129_{10} = \textcolor{red}{1000\ 0001}_2$$

Significant (Fraction)= **0101 0000 0000 0000 0000 000** (plus 19 times of 0's)

\rightarrow Result: **1 1000 0001 0101 0000 0000 0000 000**

Discussion

1. Why is the exponent stored in K-excess form?
2. Why do we choose K=127(in single precision scheme) instead K=128 (original biased value in 8-bits pattern)
3. How can we represent the zero value in floating point number?



Categories of floating-point values

□ Normalized number



□ Denormalized number



□ Infinity



□ Not a number (NaN)

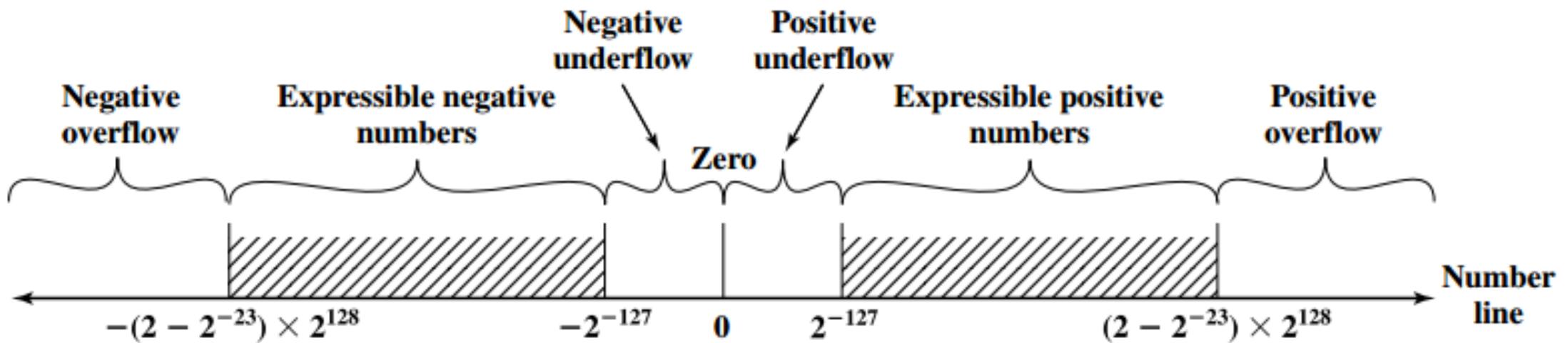


Nonnegative floating-point numbers

Description	exp	frac	Single precision		Double precision	
			Value	Decimal	Value	Decimal
Zero	00 ··· 00	0 ··· 00	0	0.0	0	0.0
Smallest denorm.	00 ··· 00	0 ··· 01	$2^{-23} \times 2^{-126}$	1.4×10^{-45}	$2^{-52} \times 2^{-1022}$	4.9×10^{-324}
Largest denorm.	00 ··· 00	1 ··· 11	$(1 - \epsilon) \times 2^{-126}$	1.2×10^{-38}	$(1 - \epsilon) \times 2^{-1022}$	2.2×10^{-308}
Smallest norm.	00 ··· 01	0 ··· 00	1×2^{-126}	1.2×10^{-38}	1×2^{-1022}	2.2×10^{-308}
One	01 ··· 11	0 ··· 00	1×2^0	1.0	1×2^0	1.0
Largest norm.	11 ··· 10	1 ··· 11	$(2 - \epsilon) \times 2^{127}$	3.4×10^{38}	$(2 - \epsilon) \times 2^{1023}$	1.8×10^{308}

Chap2, Prentice.Hall.Computer.Systems.A.Programmers.Perspective.2nd.2011, Figure 2.35

Distribution of Single-precision floating-point numbers



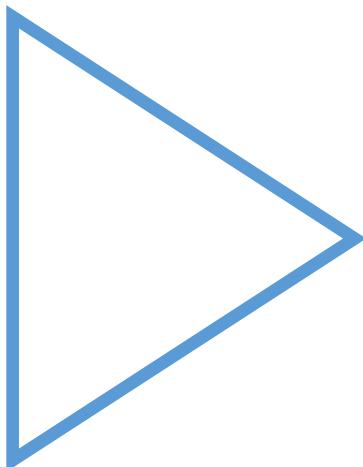
Chap9, Computer Organization and Architecture: Design for performance, 8th edition Figure 9.19

Number limits, Overflow and Roundoff

Watch this video:

Number limits, Overflow and roundoff

Take a practice bellow the video



Store text in binary

Read this document and take a practice

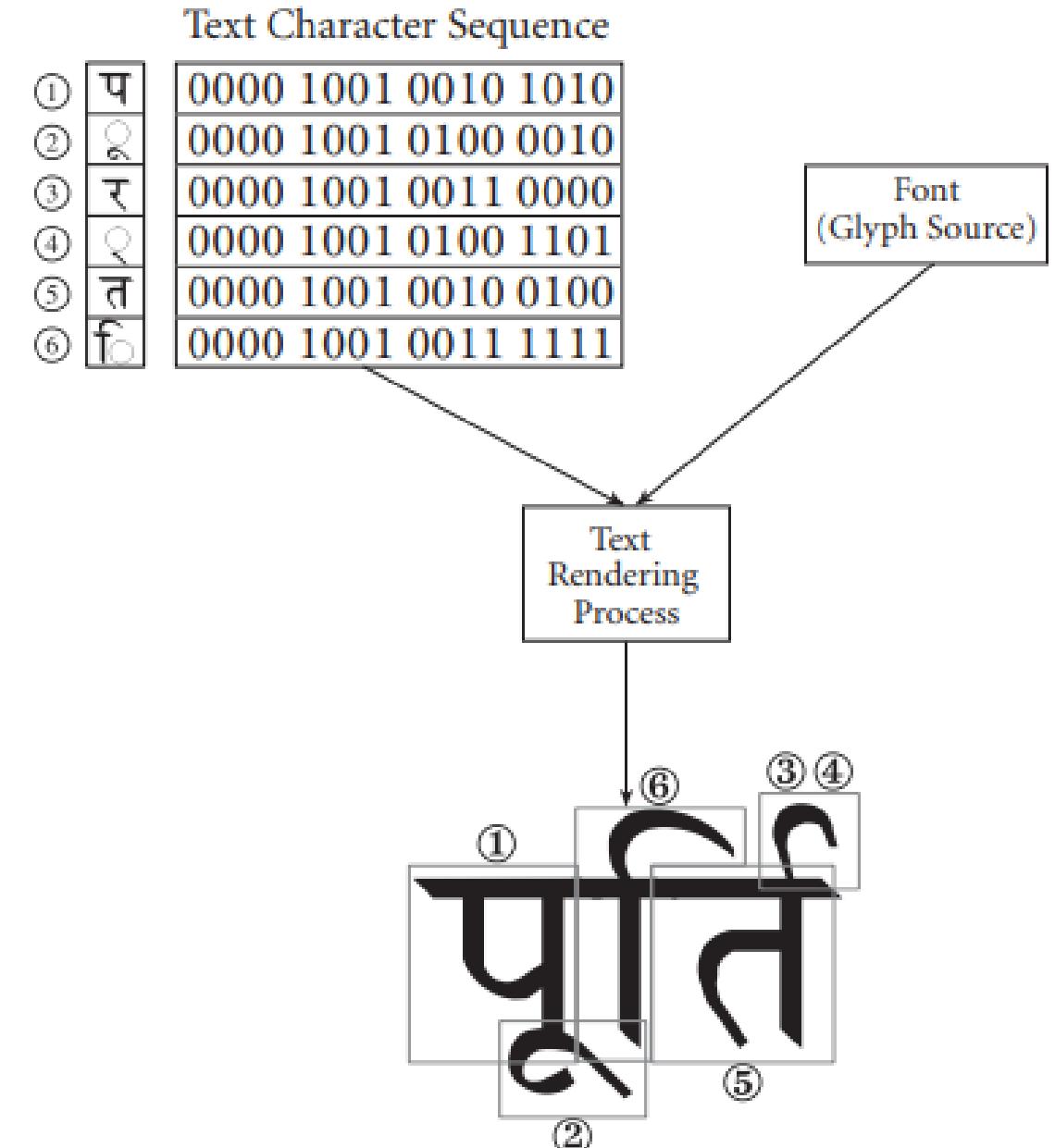


ASCII representation of character

ASCII value	Character	ASCII value	Character	ASCII value	Character	ASCII value	Character	ASCII value	Character	ASCII value	Character	ASCII value	Character
32	space	48	0	64	@	80	P	96	~	112	p		
33	!	49	1	65	A	81	Q	97	a	113	q		
34	"	50	2	66	B	82	R	98	b	114	r		
35	#	51	3	67	C	83	S	99	c	115	s		
36	\$	52	4	68	D	84	T	100	d	116	t		
37	%	53	5	69	E	85	U	101	e	117	u		
38	&	54	6	70	F	86	V	102	f	118	v		
39	'	55	7	71	G	87	W	103	g	119	w		
40	(56	8	72	H	88	X	104	h	120	x		
41)	57	9	73	I	89	Y	105	i	121	y		
42	*	58	:	74	J	90	Z	106	j	122	z		
43	+	59	:	75	K	91	[107	k	123	{		
44	,	60	<	76	L	92	\	108	l	124			
45	-	61	=	77	M	93]	109	m	125	}		
46	.	62	>	78	N	94	^	110	n	126	~		
47	/	63	?	79	O	95	_	111	o	127	DEL		

Unicode Standard

- Unicode version 4.0 has more than 160 “blocks,” which is their name for a collection of symbols. Each block is a multiple of 16
- A 16-bit encoding, called UTF-16, is the default.
- A variable-length encoding, called UTF-8, keeps the ASCII subset as eight bits and uses 16 or 32 bits for the other characters.
- UTF-32 uses 32 bits per character



Unicode Character Code to Rendered Glyphs

Data Format in C Programs

C declaration	Intel data type	Assembly code suffix	Size (bytes)
char	Byte	b	1
short	Word	w	2
int	Double word	l	4
long int	Double word	l	4
long long int	—	—	4
char *	Double word	1	4
float	Single precision	s	4
double	Double precision	l	8
long double	Extended precision	t	10/12

Figure3.1 -
Prentice.Hall.Computer.Systems.A.Programmers.Perspective.2nd.2011

Size of C data type in IA32

Heterogeneous Data

- C provides two mechanisms for creating data types by
 - Combining objects of different types: *structures*, declared using the keyword ***struct***
 - Aggregate multiple objects into a single unit: *unions*, declared using the keyword ***union***
- Read more information in Prentice Hall, 2011, *Computer Systems A Programmers Perspective 2nd*, Section 3.9, page 241



Data Alignment

- Alignment restrictions simplify the design of the hardware forming the interface between the processor and the memory system
- The compiler may need to add padding to the end of the structure so that each element in an array of structures will satisfy its alignment requirement



Data Alignment

Ex: consider the following structure declaration

```
struct S1 {  
    int i;  
    char c;  
    int j;  
};
```

Size of struct S1 without alignment

	Offset	0	4	5	9
Contents	i	c	j		

Size of struct S1 with 4-bytes alignment

	Offset	0	4	5	8	12
Contents	i	c			j	

- 04_Floating-point.pdf
- Willian Stalling, **Computer Organization and Architecture: Design for performance**, 8th edition, *Chapter 9*
- Patterson and Hennessy, **Computer Organization and Design: The Hardware / Software Interface**, 5th edition, *Chapter 3*
- Prentice Hall, **Computer Systems A Programmers Perspective** 2nd, 2011, *Chapter 2*



CHAPTER

2

PROCESSOR



fit@hcmus

KHOA CÔNG NGHỆ THÔNG TIN
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

REMIND

- Inside a CPU
- Abstraction layer

What will you learn?

- How programs are translated into the machine language
- How hardware executes a program
- How CPU process an instruction
- Measuring execution time
- Uniprocessor vs Multiprocessor

Instruction

- The sequence bit that contains the request that the processor must make.
- An instruction consists of 2 part:
 - Opcode:** the operation ALU must take
 - Operand:** objects affected by the action contained in the code

Instruction Set Architecture (ISA)

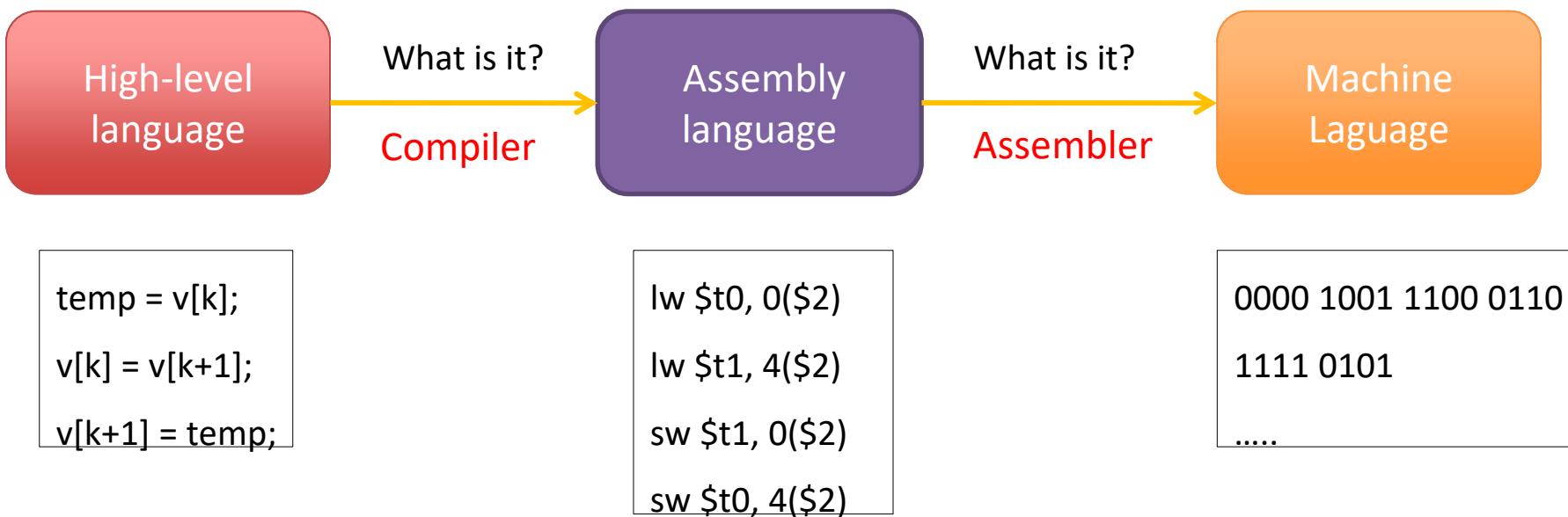
- The format and behavior of a machine-level program is defined by the instruction set architecture
- Different computers have different instruction sets but with many aspects in common
- Commonly ISA:

- MIPS: used in embedded system
- ARM: A64, A32, T32
- Power-PC
- IA-16: 16-bits processor (Intel 8086, 80186, 80286)
- IA-32: 32-bits processor (Intel 80368 – i386, 80486 – i486, Pentium II, Pentium III ...)
- IA-64: 64-bits processor (Intel x86-64 - Pentium D...)

ISA design: CISC & RISC

- Complete Instruction Set Computer (CISC): includes many instructions, from simple to complex
 - Reduced Instruction Set Computer (RISC): consists of only simple instructions
- *Which one is better?*

Discussion



Assembly Language

- A symbolic representation of machine code, clearer than in machine code
- Each assembly instruction represents exactly one machine instruction
- Ex: Save the value 5 decimal in the register \$4

Machine language: 00110100 0000100 00000000 00000101

Assembly: ori \$4, \$0, 5

opcode	dest reg	src reg
immediate		

Assembly Language

- Since each processor has its own register structure and instruction set when setting the assembly, it must be clear which processor is set, or the family of the processor.
- Ex:
- Assembly for MIPS
- Assembly for the line of Intel 8086 processors

Compiler

- A program that translates **high-level language statements** into **assembly language statements**
- Belong to:
 - The system hardware architecture below which it is running
 - The high-level language which it compiles
- Ex:
 - Compiler for C <> Compiler cho Java
 - Compiler for “C on Windows” <> “C on Linux”

Assembler

- A program that translates **a symbolic version of instructions** into the **machine code**
- A single processor (1 set of definitions) can have multiple assemblers from different vendors running on different operating systems.
- Ex: list of assembler for x86 architecture
A86, GAS, TASM, MASM, NASM
- The Assembly program depends on the assembler it uses

Discussions

□ Who will compile the compiler? (It's also a program)

→ Assembler

□ How the hardware execute a program?

→ Loader & Linker

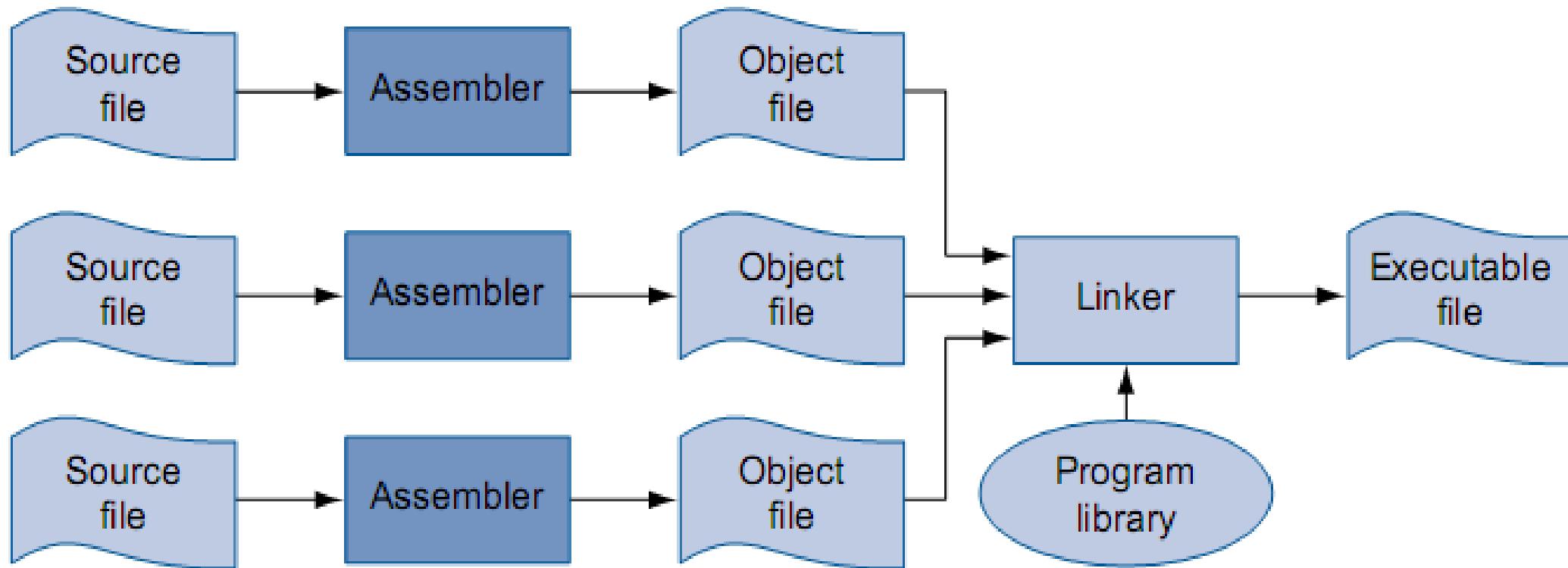
Linker

- A systems program that combines independently assembled machine language programs (object file) and resolves all undefined labels into an executable file.

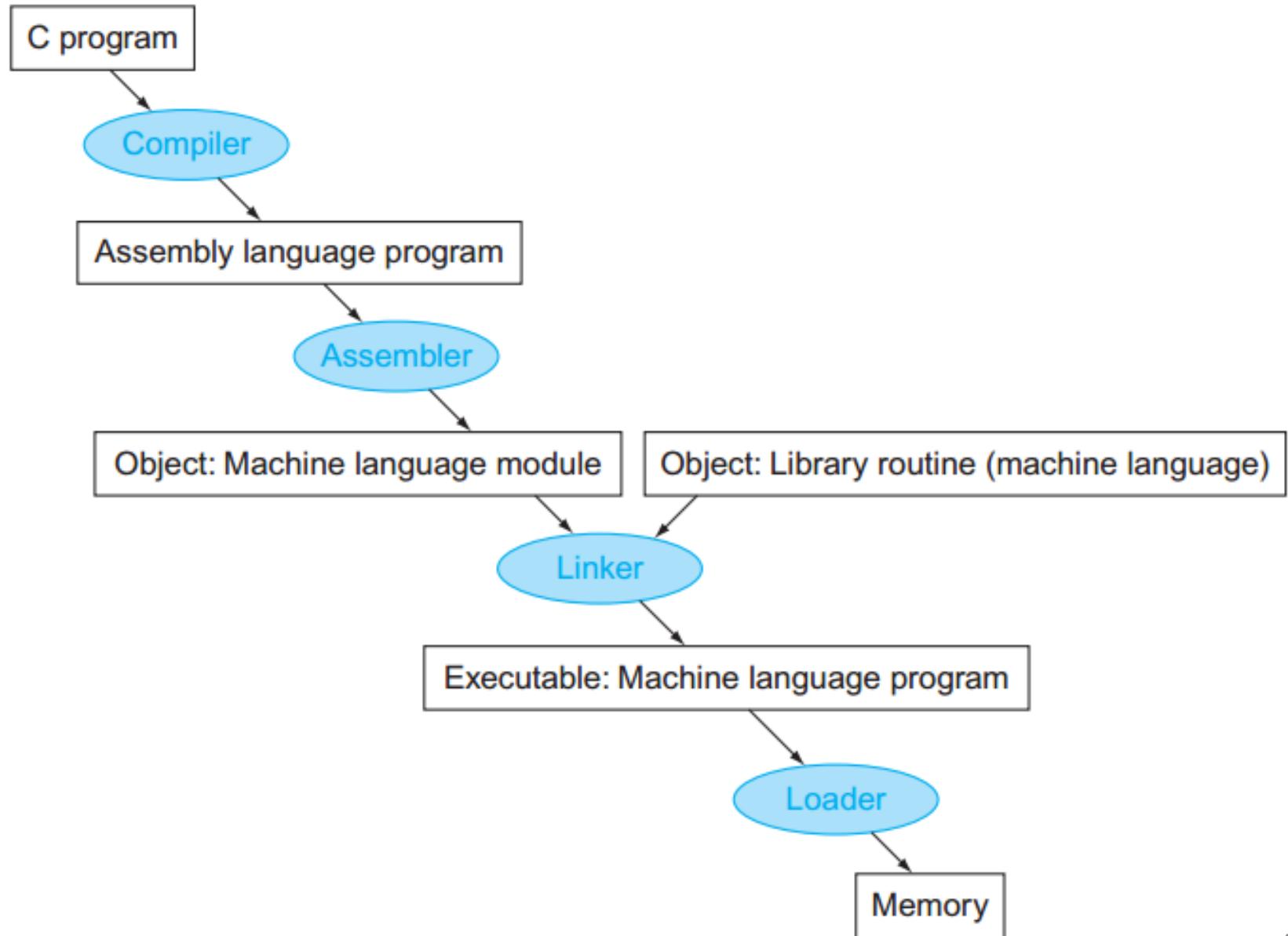
Loader

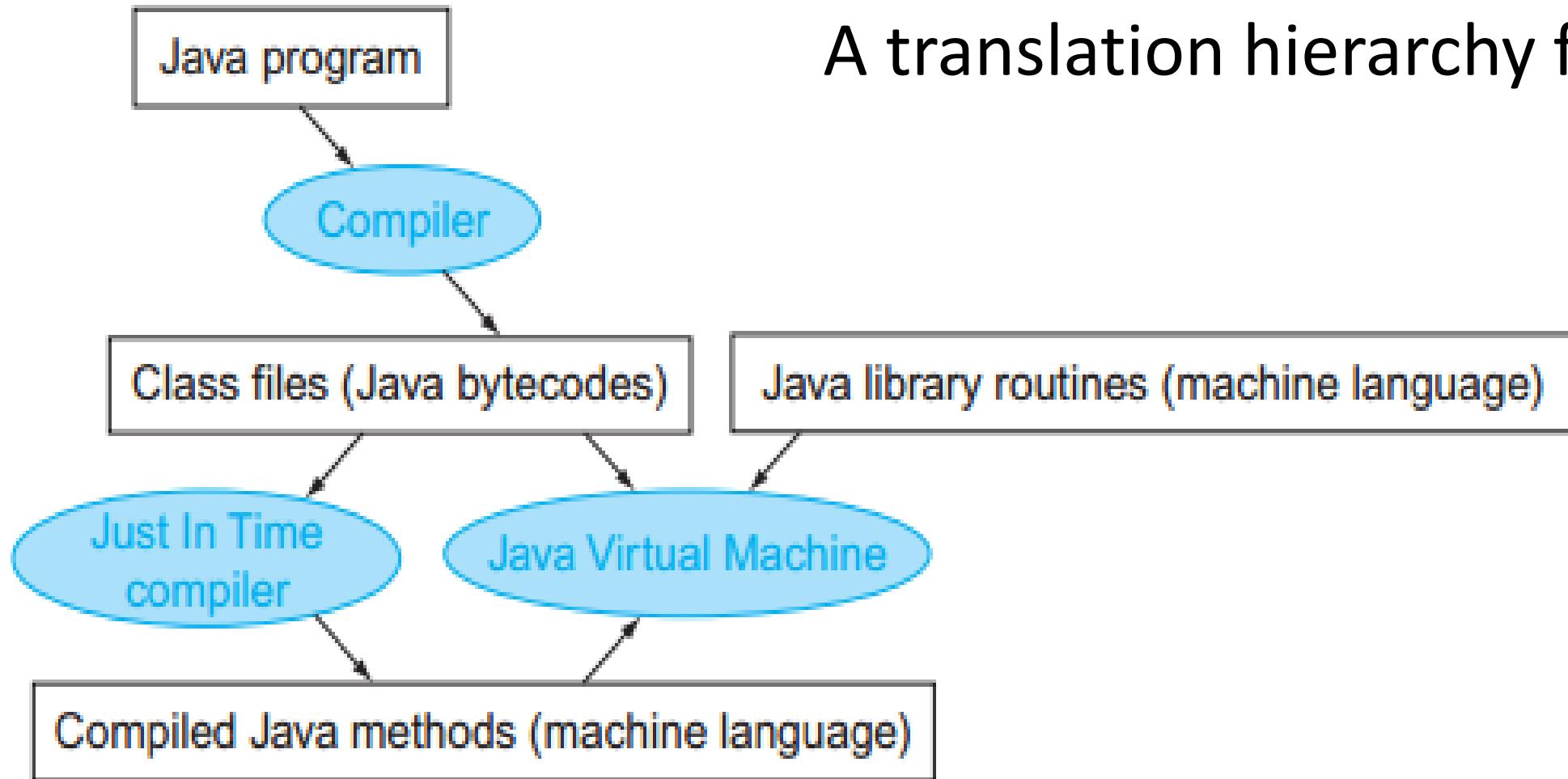
- A systems program that places an object program in main memory so that it is ready to execute.

Create executable file

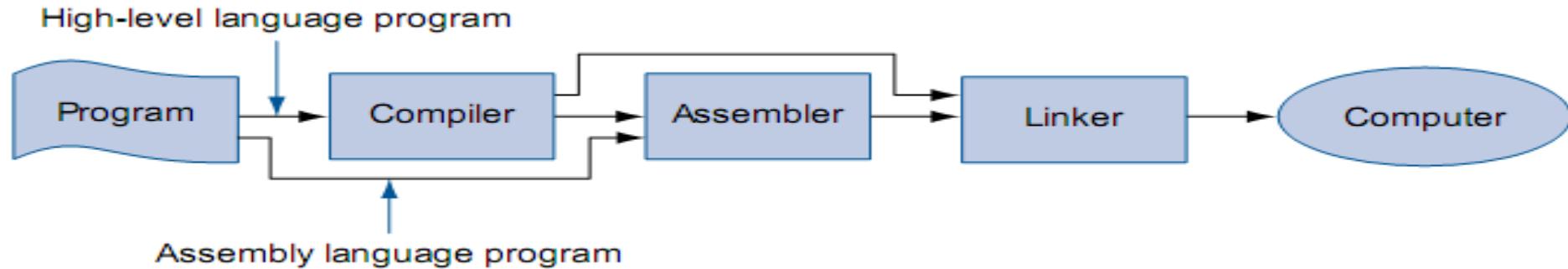


A translation hierarchy for C



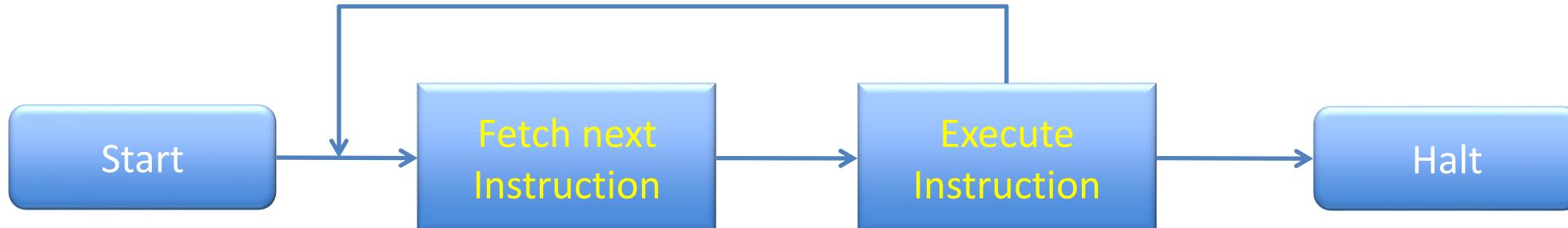


Realistic Model



- Compiler and assembler can be skipped in the certain cases
- In fact, there are several compilers that can create executables on a variety of underlying architecture platforms (cross-platform compiler)
- Ex: Compiler for Java, Cygwin, Code::Block Studio

Instruction processing

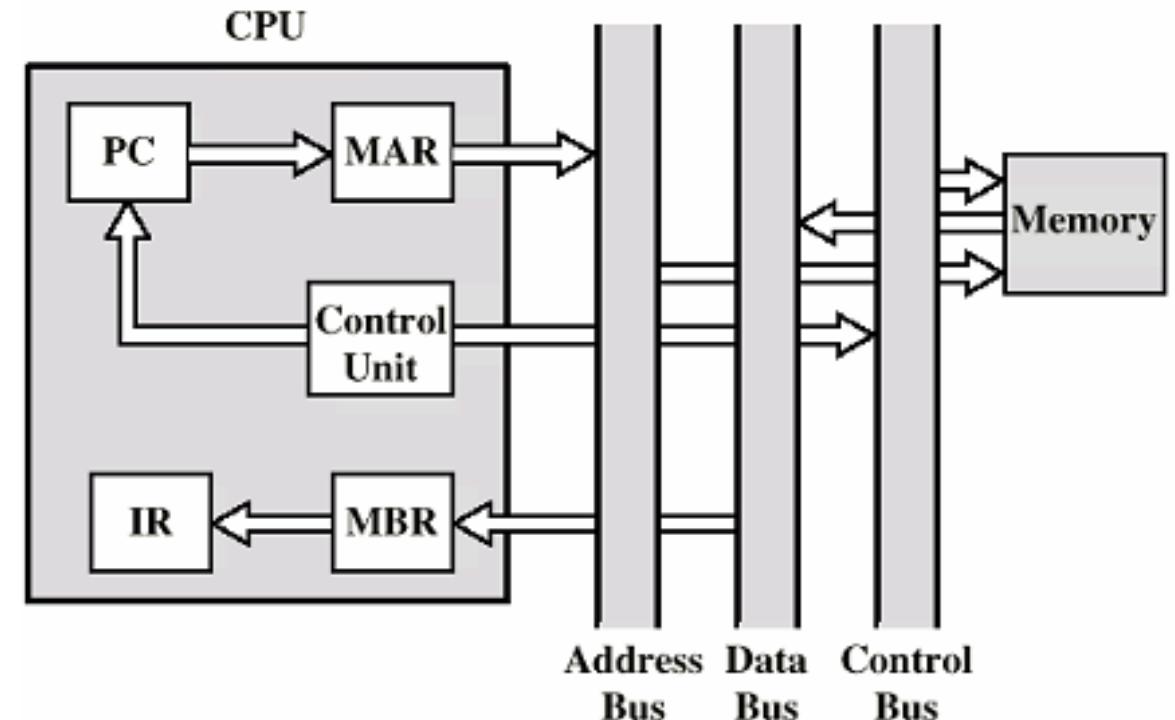


Instruction Cycle: consists of 2 phases

- Fetch cycle: Transfer data from memory to registers
- Execute cycle: Decode the instruction and execute the requirements of it

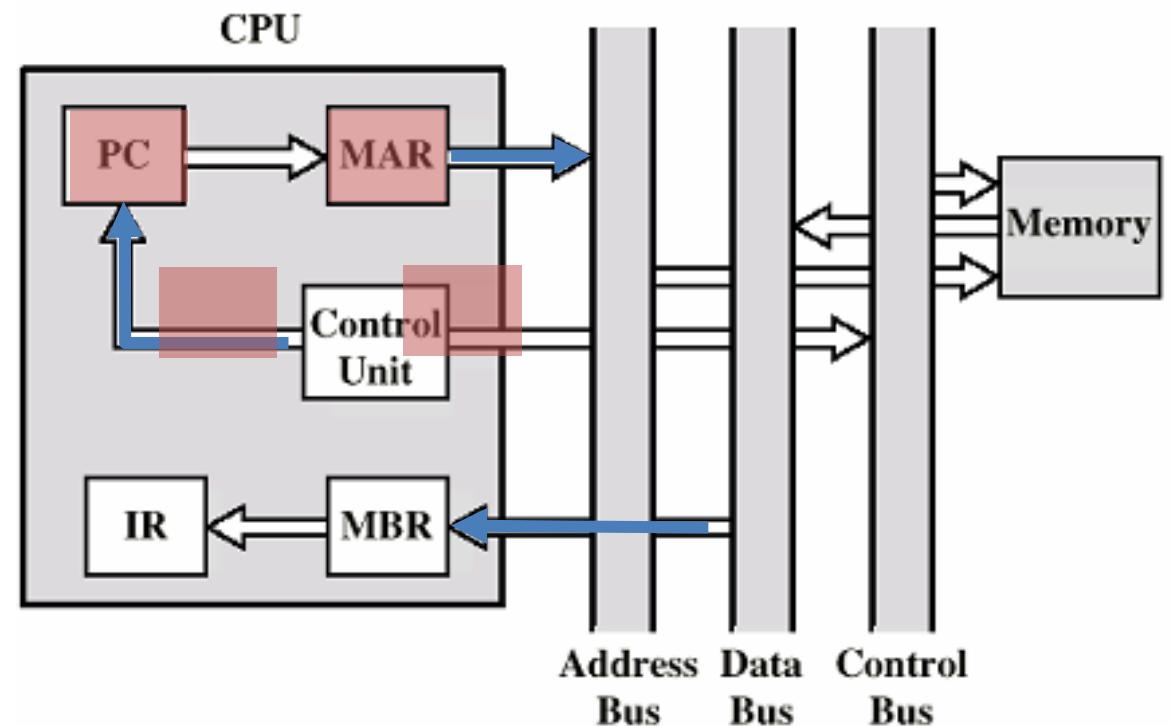
Fetch cycle

- PC (Program Counter)
 - Store the next instruction's address
- MAR (Memory Address Register)
 - Store the address of a location in memory (output to address bus)
- MBR (Memory Buffer Register)
 - A word of data to be written to memory or the word most recently read (output to data bus)
- IR (Instruction Register)
 - Contain the most recently fetched instruction

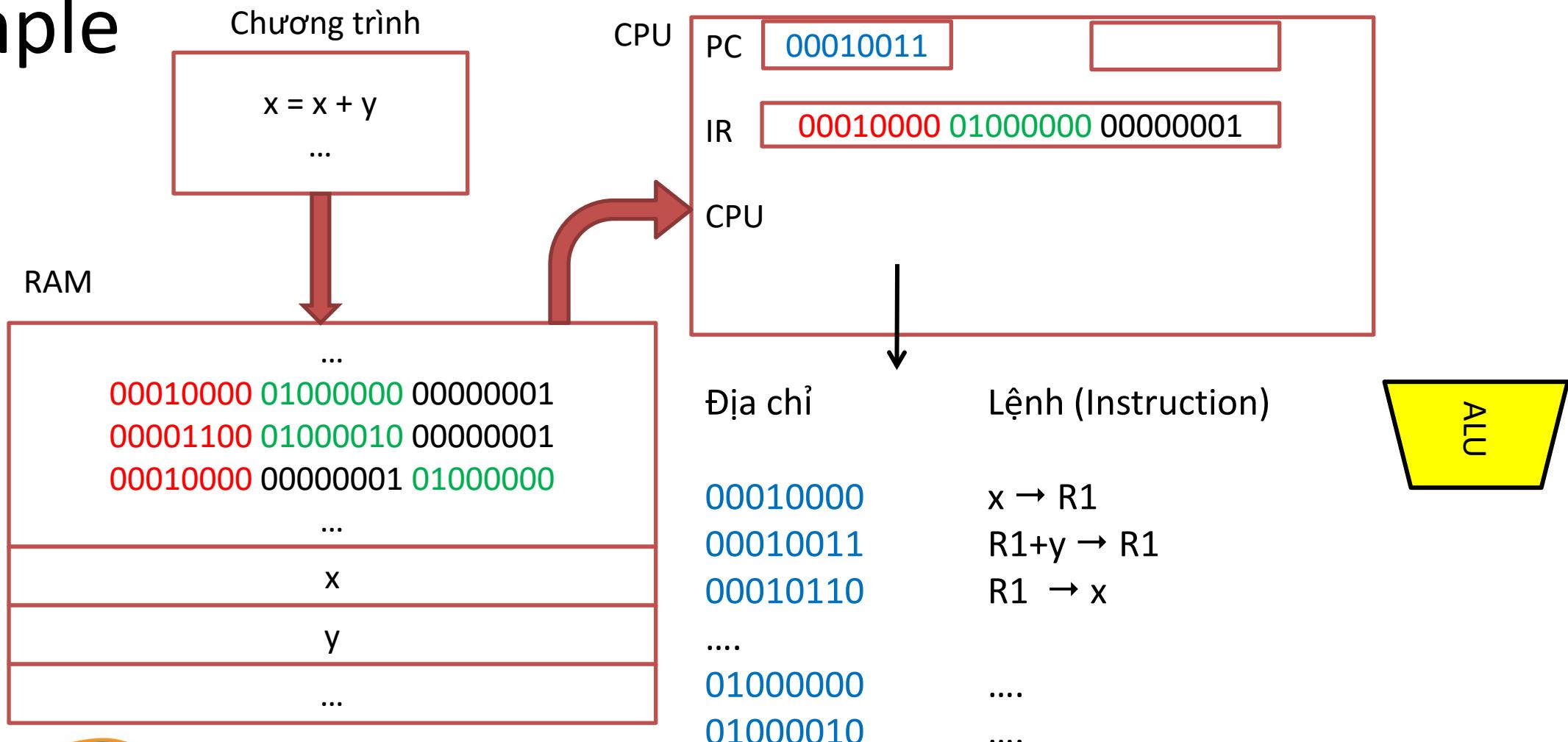


Fetch cycle

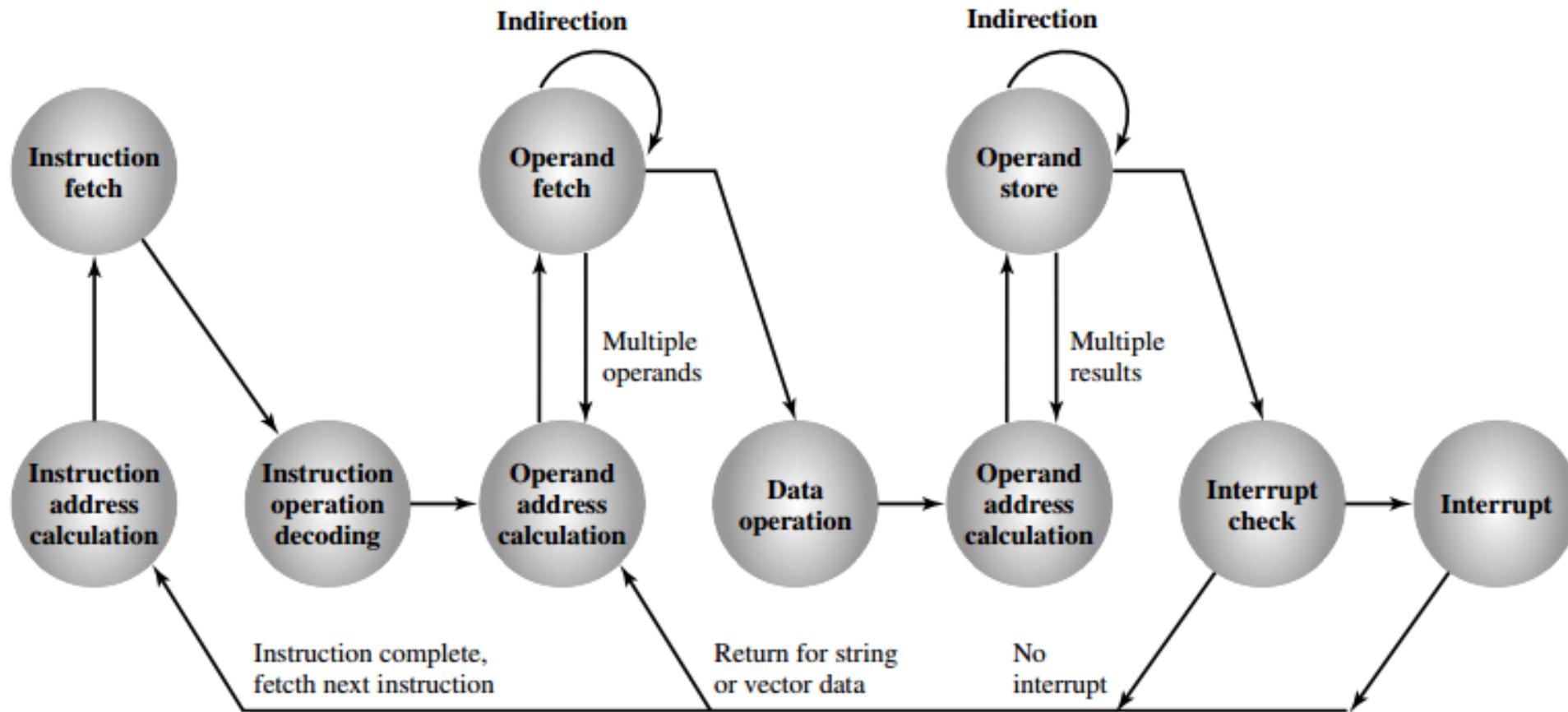
- The control unit move the instruction which has address in PC regs to IR
→ Default update PC reg:
 $\text{PC} += \text{size of the fetched instruction}$
- The fetched instruction is loaded into an IR, where the opcode and operand are analyzed
- Data are exchanged with memory using the MAR and MBR



Example



Execute cycle



Chap3, William Stalling, *Computer Organization and Architecture*, 8th ed,
2014 Figure 3.6

Measuring execution time

Elapsed time

- Total response time, including all aspects: processing, i/o, idle time, OS overhead

CPU time

- Time spent processing a given task
- Comprise user CPU time and system CPU time
- Different program are affected differently by CPU and system performance

Clock Cycles

Instead of reporting execution time in seconds, we often use *cycles*. In modern computers hardware events progress cycle by cycle: in other words, each event, e.g., multiplication, addition, etc., is a sequence of cycles

$$\frac{\text{seconds}}{\text{program}} = \frac{\text{cycles}}{\text{program}} \times \frac{\text{seconds}}{\text{cycle}}$$

cycle time = time between ticks = seconds per cycle
clock rate (frequency) = cycles per second (1 Hz. = 1 cycle/sec, 1 MHz. = 10^6 cycles/sec)

CPU Time

$$\text{CPU time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{CPU clock cycles}}{\text{Instruction}} \times \frac{\text{seconds}}{\text{CPU clock cycles}}$$

Performance depends on:

- Algorithm
- Programming language
- Compiler
- ISA

Uniprocessor vs Multiprocessor

Constrained by:

Power

Instruction-level parallelism

Memory latency

Multicore microprocessors (>1 processor/chip)

Requires explicitly parallel programming

- Compare with instruction level parallelism
 - Hardware executes multiple instructions at once
 - Hidden from the programmer
- Hard to do
 - Programming for performance
 - Load balancing
 - Optimizing communication and synchronization



CHAPTER

3

MIPS – 32 BITS



fit@hcmus

KHOA CÔNG NGHỆ THÔNG TIN
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

REMIND

- ISA
- RISC vs CISC



PREREQUITES

- Install MARS software already

What will you learn?

- Basic of a MIPS program
- Registers
- Memory Organization
- Operations
- System calls
- Procedures



MIPS Instruction Set

- Large share of embedded core market
 - Applications in consumer electronics, network/storage equipment, cameras, printers, ...
- Typical of many modern ISAs

MIPS Principles

According to the RISC architecture with 4 principles:

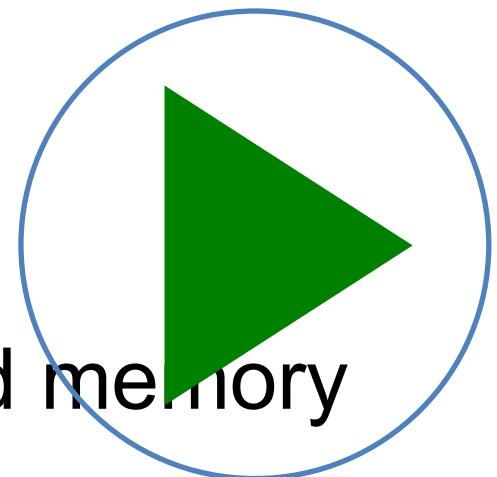
- Simpler is more stable
- Smaller is faster
- Increase processing speed for frequent cases
- Design requires good compromise

MIPS Assembly Program

```
.data      # data label declaration
label1: <data type> <initial value>
label2: <data type> <initial value>
...
.text      # instructions follow this directive
.globl    #global text label, can access from another file
.globl    main      # This is the required global text label of
the program
...
main:      # indicated start of code
...
#end of program
```

How to use MARS to code an assembly program

- Introduce how to use MARS tool
- Create a basic assembly program
- Observe the changing in the registers set and memory



Policy of use conventions for registers

Name	Register Number	Usage
\$zero	0	The constant value 0
\$v0	2	Result from callee
\$v1	3	Return to caller
\$a0	4	Argument to callee
\$a1-\$a3	5-7	From caller: caller saves
\$t0-\$t7	8-15	Temporaries: callee can clobber
\$s0-\$s7	16-23	Saved: callee can clobber
\$t8-\$t9	24-25	More temporaries (conditions)
\$gp	28	Global pointer
\$sp	29	Stack pointer
\$fp	30	Frame pointer
\$ra	31	Return address (caller saves)
\$at	1	Reserved for assembler
\$k0-\$k1	26-27	Reserved for the OS kernel

Memory Organization

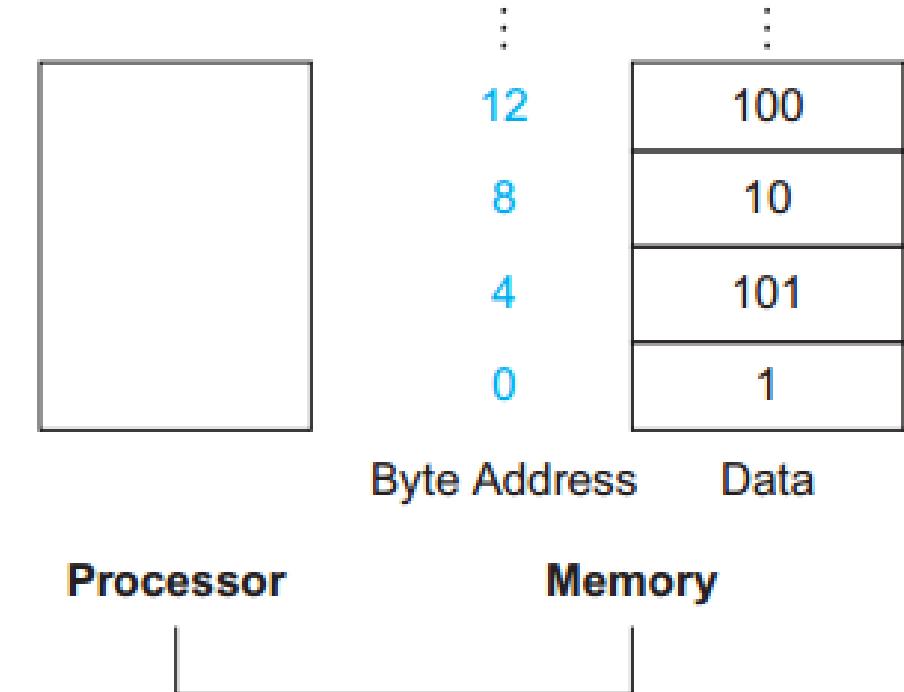
- Address, index
- Byte addressing
- Little/ Big Endian byte order
- MIPS addressing mode

Address/ Index

- The **address** is a value used to specify **the location of a specific data element** within a memory array
- The address of a word matches the address of one of the 4 bytes within the word and addresses of sequential words differ by 4

Byte addressing

- **Byte addressing** is the index points to a byte of memory
- MIPS uses byte addressing, with word addresses are multiples of 4
- To access a word in memory, the instruction must supply the memory address



Chap2, A. Patterson and J. L. Hennessy, Computer Organization and Design: The Hardware/Software Interface, 5th ed, 2014 Figure 2.3

Actual MIPS memory addresses and contents of memory for those words

Little/ Big Endian byte order

Bytes in a word can be numbered in two ways:

- Byte 0 at the leftmost (most significant) to the rightmost (least significant), called *big-endian* (*MIPS uses big-endian byte order*)
- The leftmost (most significant) to byte 0 at the rightmost (least significant), called *little-endian*

Big-endian

Byte 0	Byte 1	Byte 2	Byte 3
00	00	07	E4

Bit 31 Bit 0

Little-endian

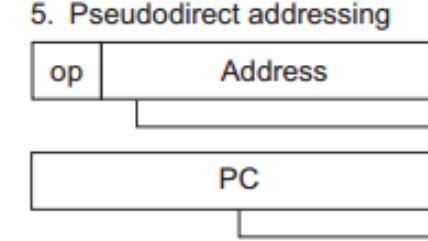
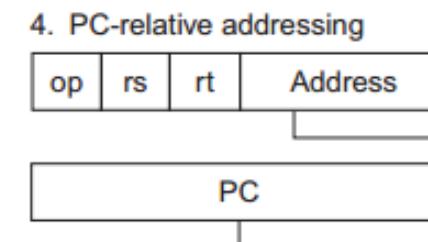
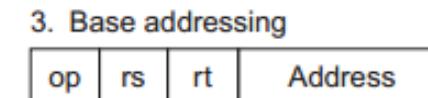
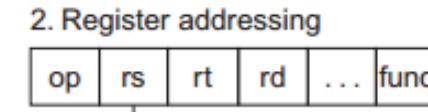
Byte 3	Byte 2	Byte 1	Byte 0
E4	07	00	00

Bit 31 Bit 0

MIPS addressing mode

MISPS supports the following addressing mode:

- Immediate addressing
- Register addressing
- Memory addressing
 - Base addressing
 - PC-relative addressing
 - Pseudo-direct addressing



MIPS Operations

- Arithmetic
- Logical
- Data transfer
- Branch (Unconditional/ conditional)

Arithmetic operations

Syntax (**R-format**): op rd, rs, rt

op: operation code

rd: destination register number

rs: the first source register number

rt: the second source register number/ an immediate

Arithmetic operations

C code:

$$A = B + C$$

MIPS code

```
add $s0, $s1, $s2
```

Compiler will
associate the
variables to the
registers

Discussions

1. How to know if an operation compiled from C is a signed operation or not?
→ Compiler
2. Can an operand be used as both a source and a target?
→ Yes
3. Can constant data specify in an instruction?
→ Yes (use I-format)

addi \$s0, \$s1, 3 (addi = add immediate)
addi \$s0, \$s1, -3 (do not have subi)

Example: compute multiple operands

C code:

$$f = (g + h) - (i + j)$$

MIPS code: $f \rightarrow \$s0$, $h \rightarrow \$s1$, $i \rightarrow \$s2$, $j \rightarrow \$s3$

```
add    $t0, $s1, $s2      # temp1 = g + h
add    $t1, $s3, $s4      # temp2 = i + j
sub    $s0, $t0, $t1      # f = temp1 - temp2
```

Example: assignment/ move between registers

C code:

```
a = b
```

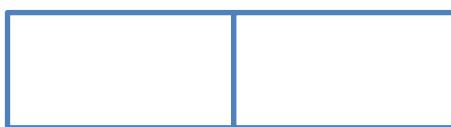
MIPS code:

```
add    $t1, $t0, $zero # a → $t1, b → $t0  
#$zero contains the constant value 0
```

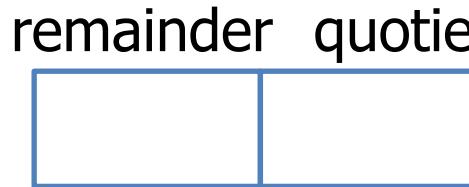
Arithmetic operations: Multiply & Division



rs rt



rs rt



remainder quotient HI LO

- Two 32-bit registers for product
HI: most-significant 32 bits
LO: least-significant 32-bits
- Use HI/LO registers for result
 - HI: 32-bit remainder
 - LO: 32-bit quotient
- Use **mfhi**, **mflo** instructions to access the result

Arithmetic operations: Multiplication

Syntax 1: mult rs, rt / multu rs, rt

Get the result:

```
mfhi rd      #test HI value to see if product overflow 32 bits
```

```
mflo rd
```

Syntax 2: mul rd, rs, rt

```
# Least significant 32 bits of product → rd
```

Arithmetic operations: Division

Syntax : div rs, rt / divu rs, rt

Get the result:

mfhi rd #HI contains 32 bits of remainder

mflo rd #LO contains 32 bits of quotient

No overflow or divide by 0 checking (software's job)

Arithmetic operations: Floating-point numbers

- Coprocessor1: the adjunct processor that extends the ISA
- Separate FP (Floating-point) registers:
 - Single-precision: \$f0 - \$31 (32 registers)
 - Use the paired registers to save double precision floating-point number
- FP instructions only operate on FP registers



Example: floating-point operands

Single-precision:

add.s \$f0, \$f1, \$f2

Double-precision:

add.d \$f0, \$f1, \$f2

Logical Operations

Instructions for bitwise manipulation

Operation	C	Java	MIPS
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bitwise AND	&	&	and, andi
Bitwise OR			or, ori
Bitwise NOT	~	~	nor

Useful for extracting and inserting groups of bits in a word

Logical Operations: Shift operations

Syntax (**R-format**): op rd, rs, shamt

op: operation code

rd: destination register number

rs: the source register number

shamt: How many positions to shift (< 32 bits)

Logical Operations: Shift operations

Shift left logical: shift left and fill with 0 bits

sll \$s0, \$s1, 2 # \$s1 shl 2 → \$s0

Shift right logical: shift right and fill with 0 bits

srl \$s0, \$s1, 2 # \$s1 shr 2 → \$s0

Shift right arithmetic: shift right and fill with bits which has a value equal to the MSB bit

sra \$s0, \$s1, 2 # \$s1 sar 2 → \$s0

Logical Operations: Logical math operations

Syntax (**R-format**): op rd, rs, rt

op: operation code

rd: destination register number

rs: the first source register number

rt: the second source register/ immediate

Logical Operations: Logical math operations

MIPS does not support instructions for NOT, XOR, NAND

$a \rightarrow \$s1$, $b \rightarrow \$s2$

and \$s0, \$s1, \$s2 # a AND b

ori \$s0, \$s1, 2 # a OR i (a constant
number)

nor \$t0, \$t1, \$zero #NOT A = A NOR 0

Data transfer

- RAM access only allow load/ store instructions
- Load word has destination first, store has destination last

Data transfer operations

Syntax (**I-format**): op rt, (constant/address) rs

op: operation code

rs: base address

rt: destination/ source register number

Constant: $-2^{15} \rightarrow 2^{15} - 1$

Address: offset added to base address in rs (offset is always a multiple of 4)

Data transfer operations

```
lw    rt, offset(rs)  # load a word at the  
address           $s0 + 12 from the memory  
  
sw    rt, offset(rs)  # store a word to the  
memory at          the location $s0 + 12
```

Example

- Suppose that A is an array of 100 words with the starting address (base address) contained in register \$s0.
- The value of the variable g are stored in registers \$s1 and \$s2 respectively

C code: $g = A[2];$

MIPS code:

lw \$t0, 8(\$s0) #load A[2] to the register

\$t0

add \$s1, \$t0,\$zero #save the value of A[2]

to g

Example

C code:

A[12] = h + A[8];

MIPS code:

```
lw $t0, 32($s0) #load A[8] to the register  
$t0  
add $t0, $s2,$t0 #temporary register $t0=h +  
A[8]  
sw $t0, 48 ($s0) #store h + A[8] back to  
A[12]
```

Discussions

- Load a byte **x₇ z₆ z₅ z₄ z₃ z₂ z₁ z₀** from memory to a register in CPU.
(Useful for ASCII)

lb \$t0, g

What is the value of this register (x is the MSB of the byte) ?

→ Sing-extended: **xxxx xxxx xxxx xxxx xxxx xxxx x₇ z₆ z₅ z₄ z₃ z₂ z₁ z₀**

- If you want the remaining bits from the right to have a zero value (unsigned number)

→ Load byte unsigned: **lbu \$t0, g**

Discussions

Load/ store a half of word (useful for Unicode)

→ Load half: `lh $t0, g` #load $\frac{1}{2}$ word – the 2 right most bytes

→ Store half: `sh $t0, g` #store $\frac{1}{2}$ word – the 2 right most bytes

Branch operations:

- Two kinds of branch instructions:
 - Conditional
 - Unconditional
- MIPS branch destination address = $PC + (4 * \text{offset})$
 - Target address = $PC + \text{offset} \times 4$
 - PC already plus 4 bytes by this time

Branch operations: Conditional branch

Syntax (**I-format**): oprs , rt , target address

rs: the first source register number

rt: the second source register number

Target Address: the address of the next instruction

```
beq    rs, rt, label      #if (opr1 == opr2) goto  
label
```

```
bne    rs, rt, label      #if (opr1 != opr2) goto  
label
```

Example

C Code:

```
if (a==b)  
    i = 1;  
  
else  
    i = 0;
```

MIPS code:

```
#a → $t0, b → $t1, i → $t2  
beq    $t0, $t1, Label  
addi   $t2, $zero, 0  
Label  
addi   $t2, $zero, 1
```

Branch operations: Unconditional branch

Syntax (**J-format**): op Target Address

rs: the first source register number

rt: the second source register number

Target Address: the address of the next instruction

j label #goto label

Example

C Code:

```
Do {  
    Task 1;  
} while(a != 0)
```

Task 2

MIPS code:

#a → \$t0

Loop:

Task 1

bne \$t0, \$zero, Loop

j Task 2

Discussion

- Bigger/ smaller comparison ?
 - MIPS support **slt** instruction to make the solutions for not equal comparison

Syntax: **slt rd, rs, rt**

Explain: **if (rs < rt)**
 rd = 1
 else
 rd = 0

Example

a < b

```
slt      $t0, $s0, $s1          # if (a < b) then $t0 = 1
bne      $t0, $0, Label         # if (a < b) then goto Label
Another Task                      # else then do something
```

Label

a > b

```
slt      $t0, $s1, $s0          # if (b < a) then $t0 = 1
bne      $t0, $0, Label         # if (b < a) then goto Label
Another Task                      # else then do something
Label
```

Another Task

else then do something

Example

a ≥ b

```
slt      $t0, $s0, $s1          # if (a < b) then $t0 = 1
beq      $t0, $0, Label        # if (a ≥ b) then goto Label
Another Task
# else then do something
```

[Label](#)

a ≤ b

```
slt      $t0, $s1, $s0          # if (b < a) then $t0 = 1
beq      $t0, $0, Label        # if (b ≥ a) then goto Label
Another Task
# else then do something
```

[Label](#)

Discussion

Compare with constant?

→ MIPS support **slti** instruction to make the solutions for not equal comparison

Syntax: **slti rd, rs, constant**

Explain: **if (rs < constant)**

rd = 1

else

rd = 0

Example: Switch...case in C

C code:

```
switch (k) {  
    case 0: f = i + j; break;  
    case 1: f = g + h; break;  
    case 2: f = g - h; break;  
}
```

C code: Convert to if...else

```
if (k == 0)  
    f = i + j;  
else if (k == 1)  
    f = g + h;  
else if (k == 2)  
    f = g - h;
```

f	g	h	i	j	k
\$s0	\$s1	\$s2	\$s3	\$s4	\$s5

Example: Switch..case in MIPS

```
bne $s5, $0, L1          # if (k != 0) then goto L1
Add $s0, $s3, $s4          # else (k == 0) then f = i + j
J Exit                     # end of case → Exit (break)

L1:
addi $t0, $s5, -1          # $t0 = k - 1
bne $t0, $0, L2          # if (k != 1) then goto L2
add $s0, $s1, $s2          # else (k == 1) then f = g+ h
J Exit                     # end of case → Exit (break)

L2:
addi $t0, $s5, -2          # $t0 = k - 2
bne $t0, $0, Exit          # if (k != 2) then goto Exit
sub $s0, $s1, $s2          # else (k == 2) then f = g - h

Exit:
```

System calls

- Through the system call (**syscall**) instruction to request a service from a small set of the operating system services
- System call code → **\$v0**
- Arguments → **\$a0 - \$a3** (**\$f12** for floating-point)
- Return value → **\$v0** (**\$f0** for floating-point)

System calls

Service	System call code	Arguments	Result
print_int	1	\$a0 = integer	
print_float	2	\$f12 = float	
print_double	3	\$f12 = double	
print_string	4	\$a0 = string	
read_int	5		integer (in \$v0)
read_float	6		float (in \$f0)
read_double	7		double (in \$f0)
read_string	8	\$a0 = buffer, \$a1 = length	
sbrk	9	\$a0 = amount	address (in \$v0)
exit	10		
print_char	11	\$a0 = char	
read_char	12		char (in \$v0)
open	13	\$a0 = filename (string), \$a1 = flags, \$a2 = mode	file descriptor (in \$a0)
read	14	\$a0 = file descriptor, \$a1 = buffer, \$a2 = length	num chars read (in \$a0)
write	15	\$a0 = file descriptor, \$a1 = buffer, \$a2 = length	num chars written (in \$a0)
close	16	\$a0 = file descriptor	
exit2	17	\$a0 = result	

Example

```
addi    $v0, $0, 4    # $v0 = 0 + 4 = 4
                      # print string syscall

la     $a0, str        # $a0 = address(str)

syscall                  # execute the system call
```



CHAPTER

4

BASIC MIPS IMPLEMENT



fit@hcmus

KHOA CÔNG NGHỆ THÔNG TIN
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

PREREQUITES

- Install Procsim software already

REMIND

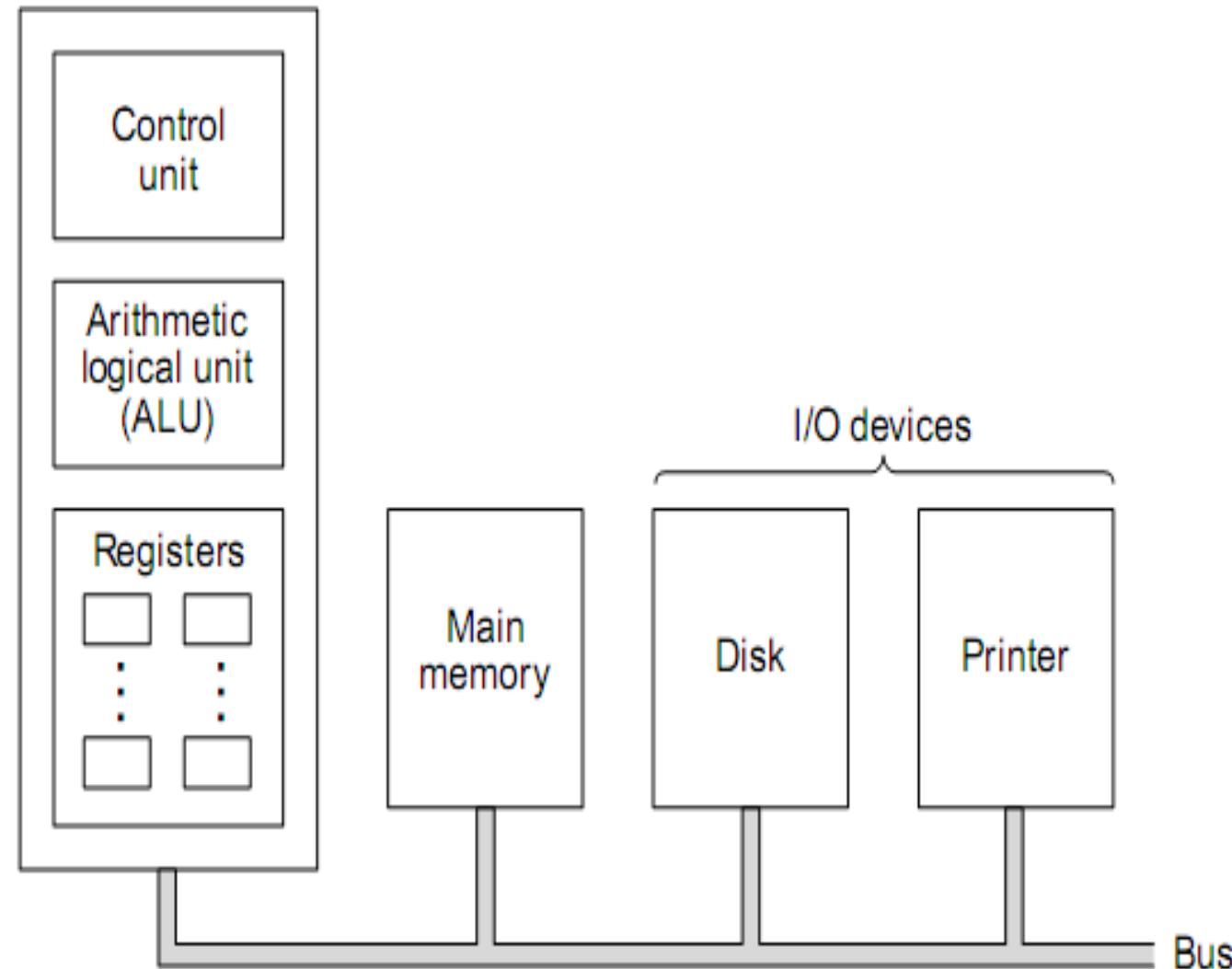
- Inside a CPU
- MIPS-32 bits operations



REMIND

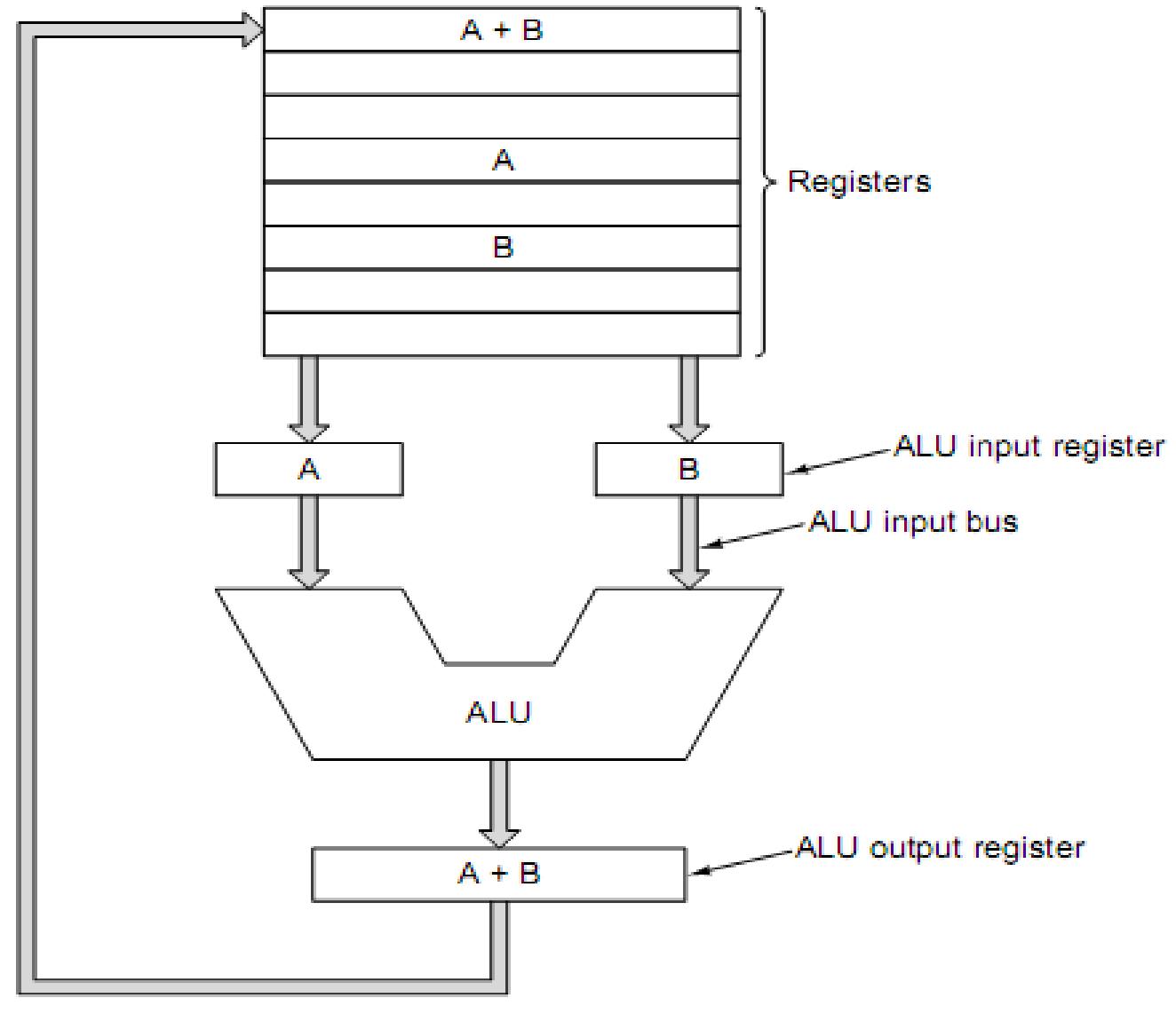
- Control Unit
- Datapath
- ALU (Arithmetic Logical Unit)
- Registers

Central processing unit (CPU)



REMIND

- ALU read operand from register or load from memory
- ALU executes the operation which the CU directed
- Store the result in memory or write back to the register



The data path of a typical Von Neumann machine

What will you learn?

- Processor Implementation Styles
 - Abstract view of MIPS subset
 - Building Datapath
 - ALU Control
 - Control Unit
- 
- Design Main Control Unit
 - Multi-cycle approach
 - Pipelined approach
(additional)

Processor Implementation Styles

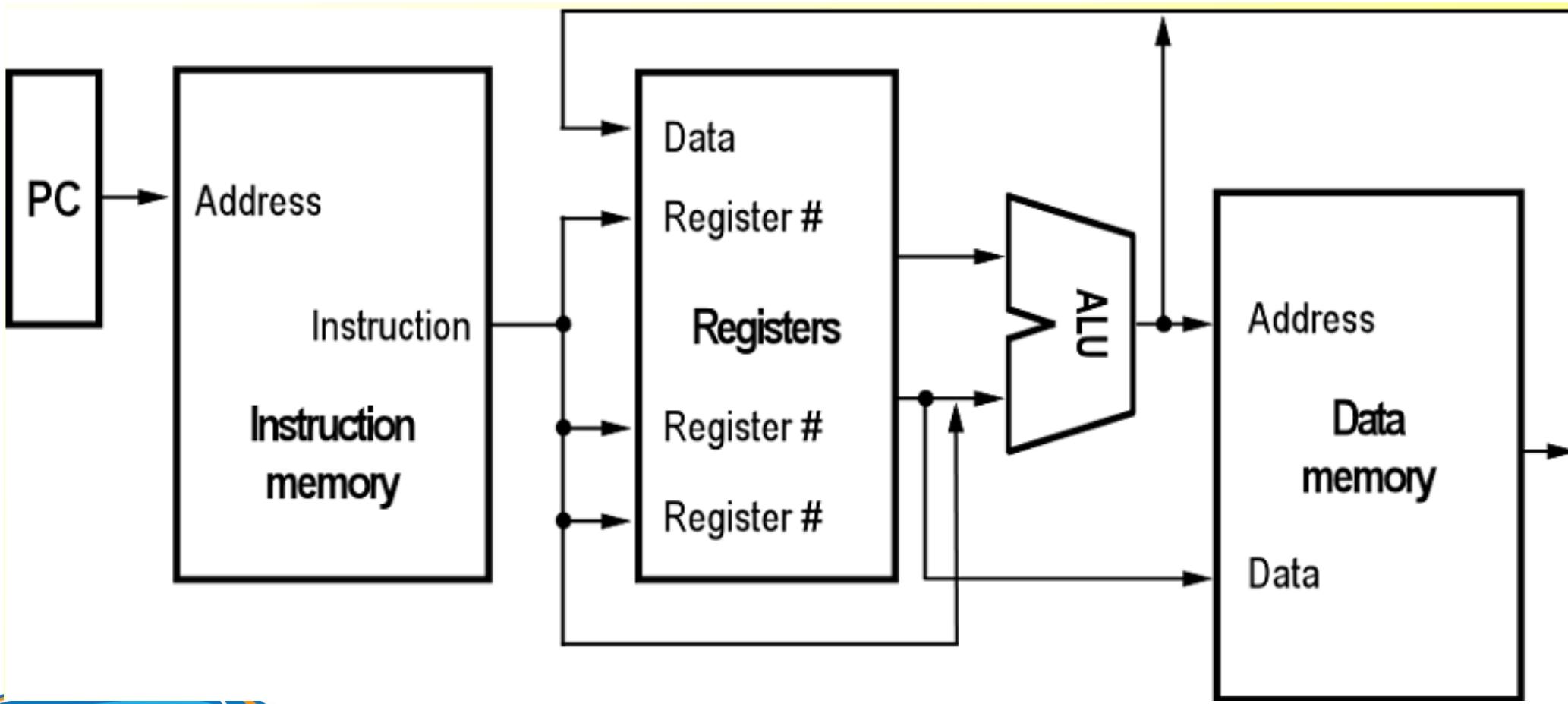
- Single Cycle
- Multi-cycle
- Pipelining

Single Cycle approach

- Execute each instruction in 1 clock cycle
- The clock cycle must have the same length for every instruction
- Therefore, the clock cycle equal to the longest possible path in the processor
- Inefficient



Abstract view of MIPS subset



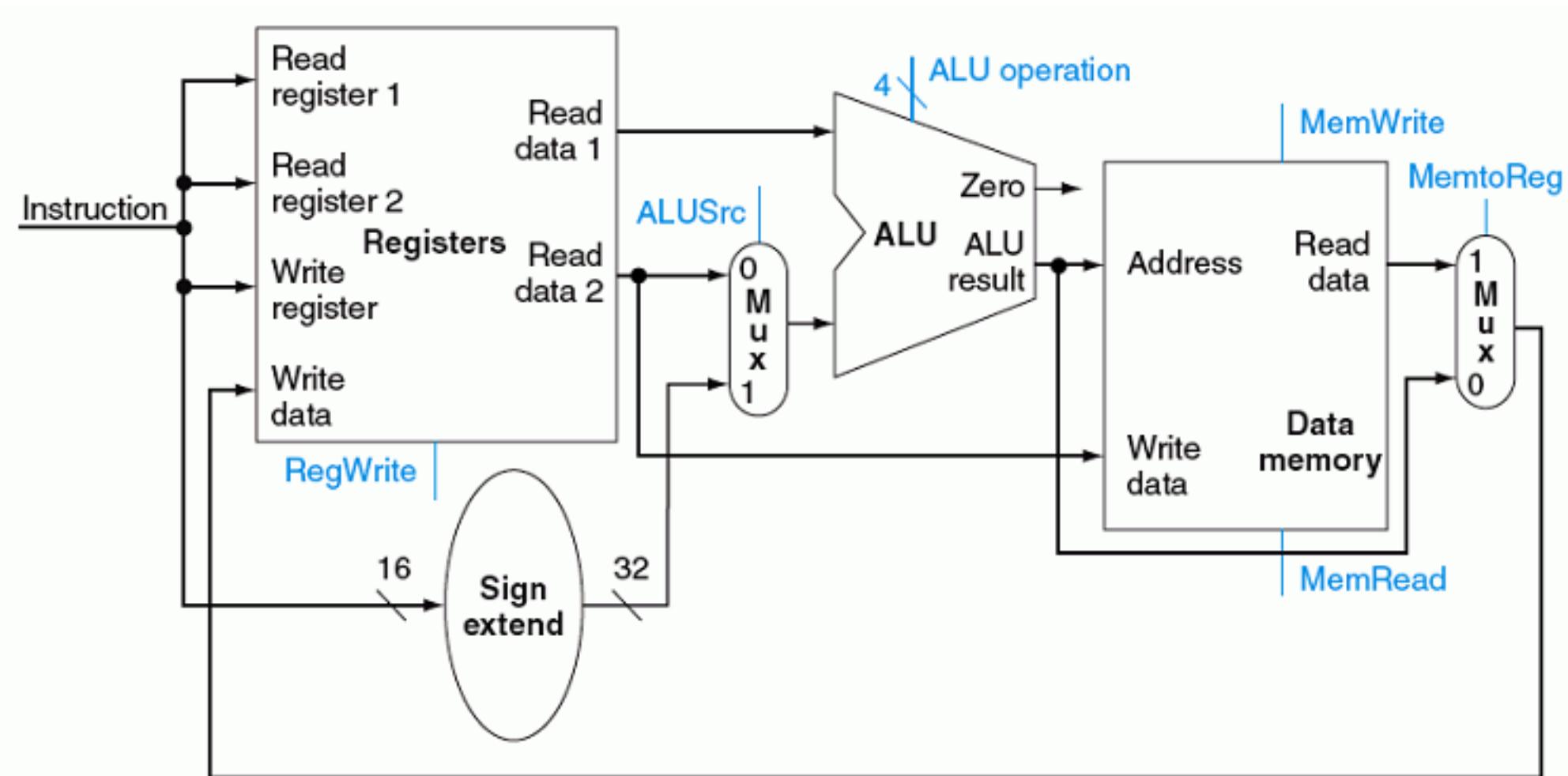
MIPS Instructions Format

Name	Fields						Comments
Filed size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instruction 32 bits
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	Address/ immediate			Transfer, branch, immediate format
J-format	op	Target address				Jump instruction format	

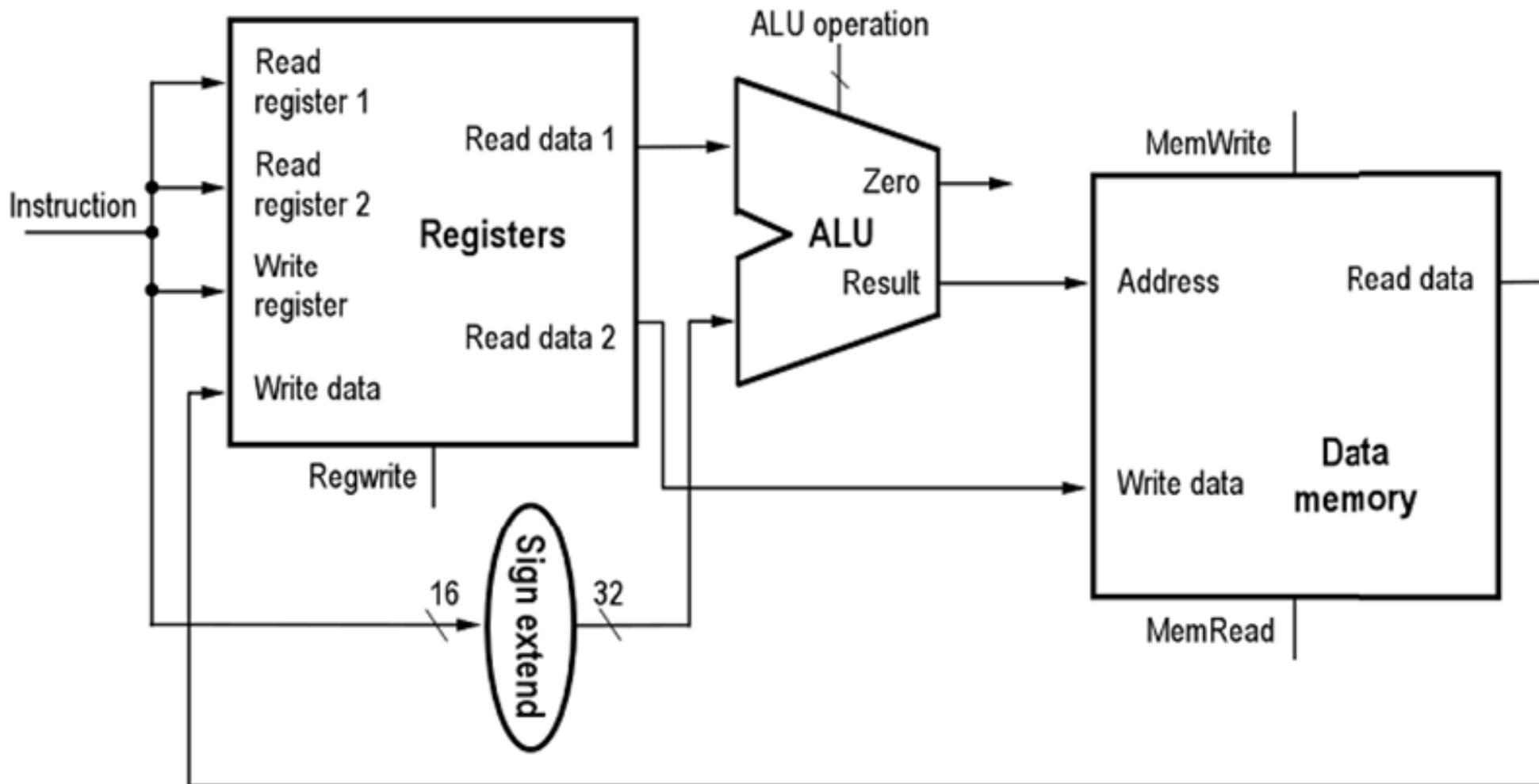
Building Datapath

- A unit used to operate or hold data in the processor
- In the MIPS implementation, the Datapath includes the instruction and data memories, the register files, the ALU, and the adders

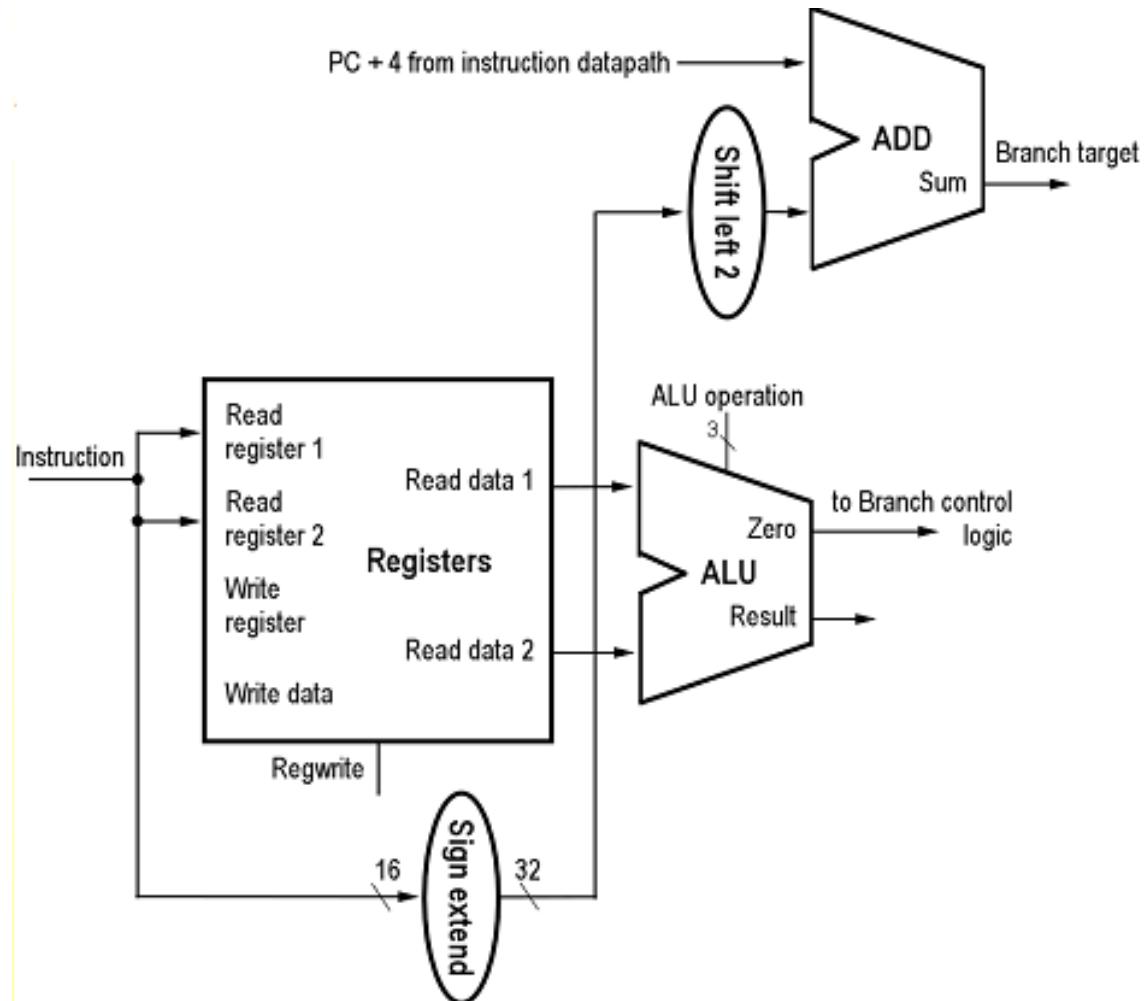
Building Datapath: R-format



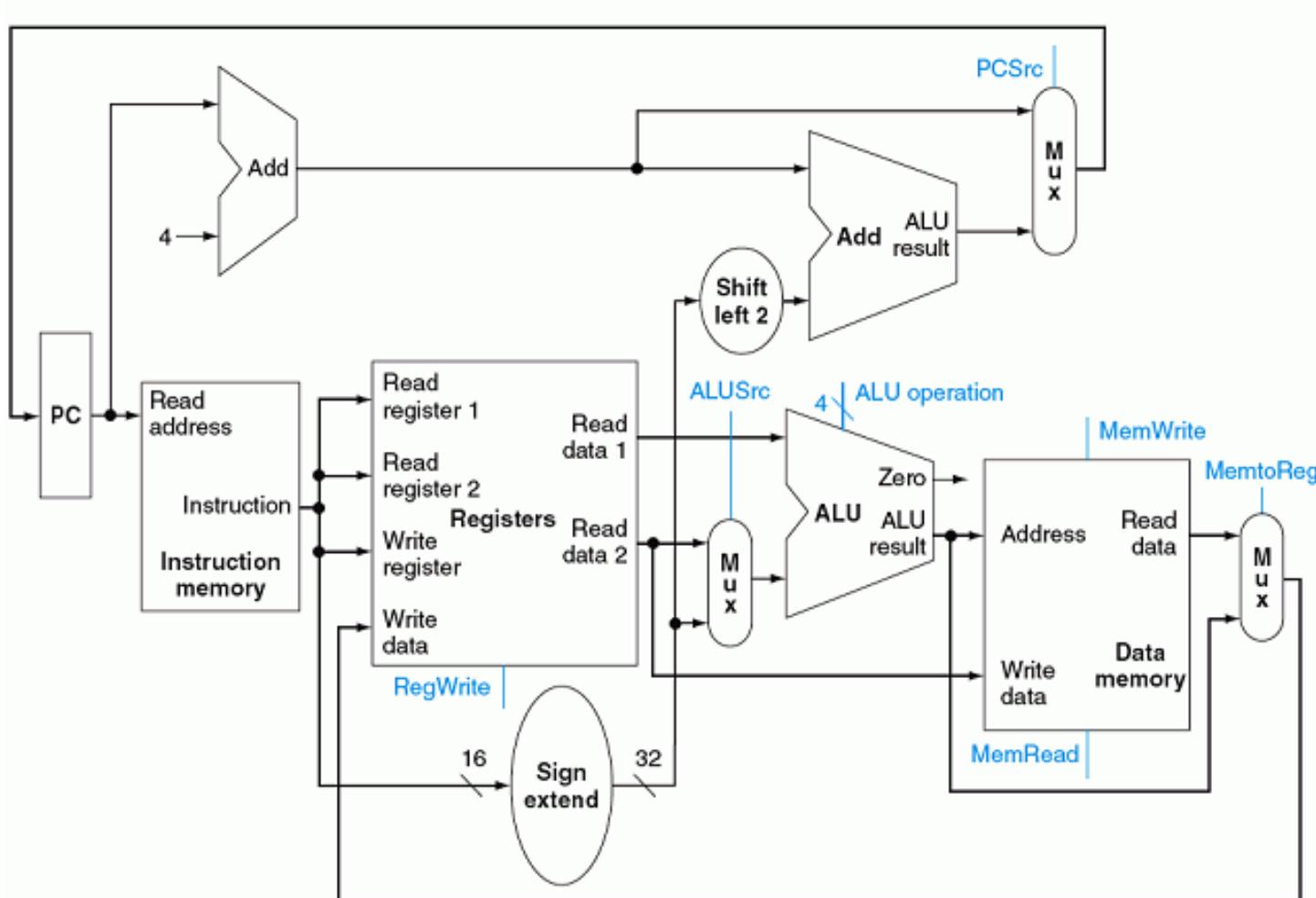
Building Datapath: I-format: lw, sw



Building Datapath: I,J-format: beq, j

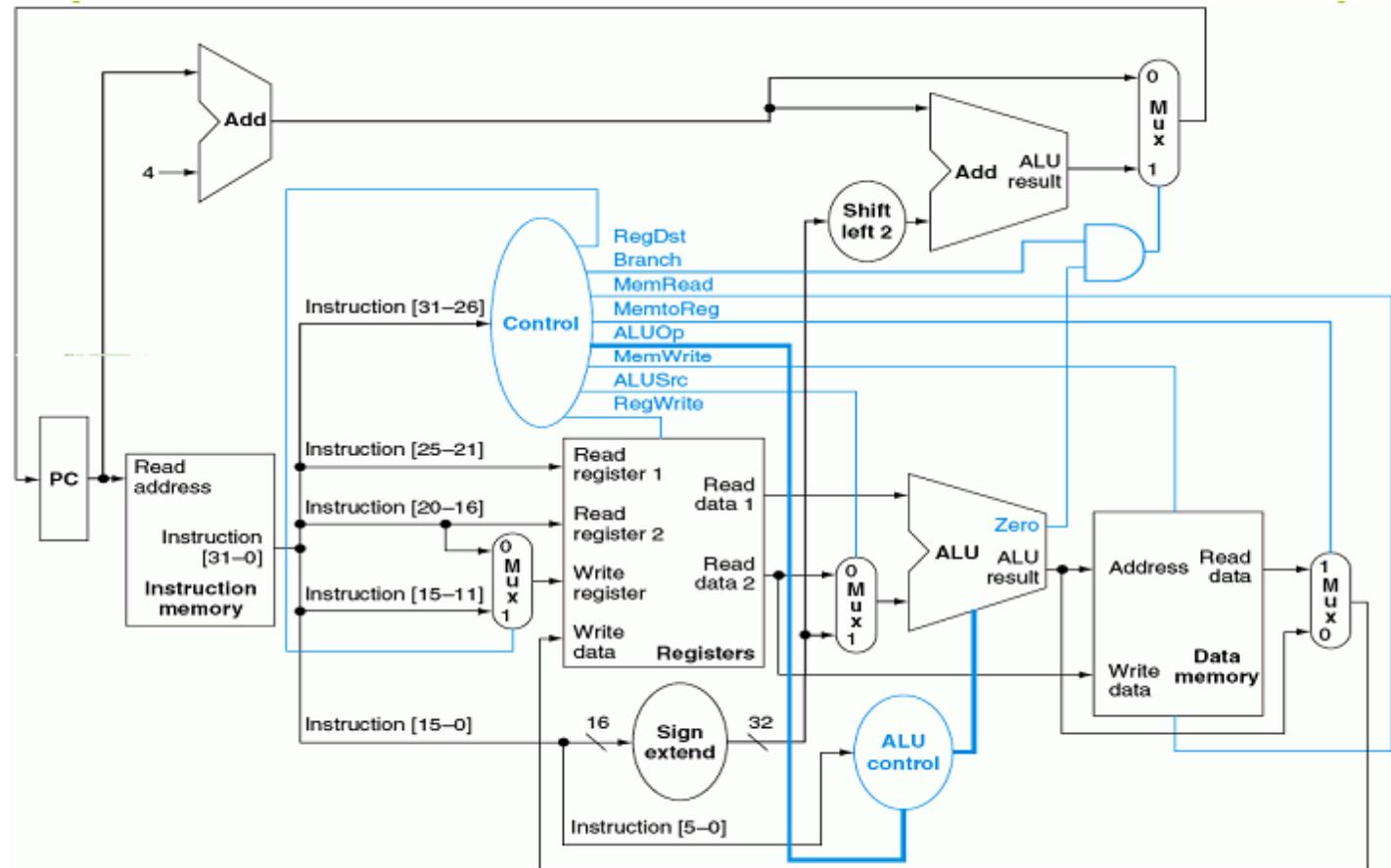


Building Datapath: R,I,J-format



Control Unit

ALU receives the directives from the control unit (ALU Control)



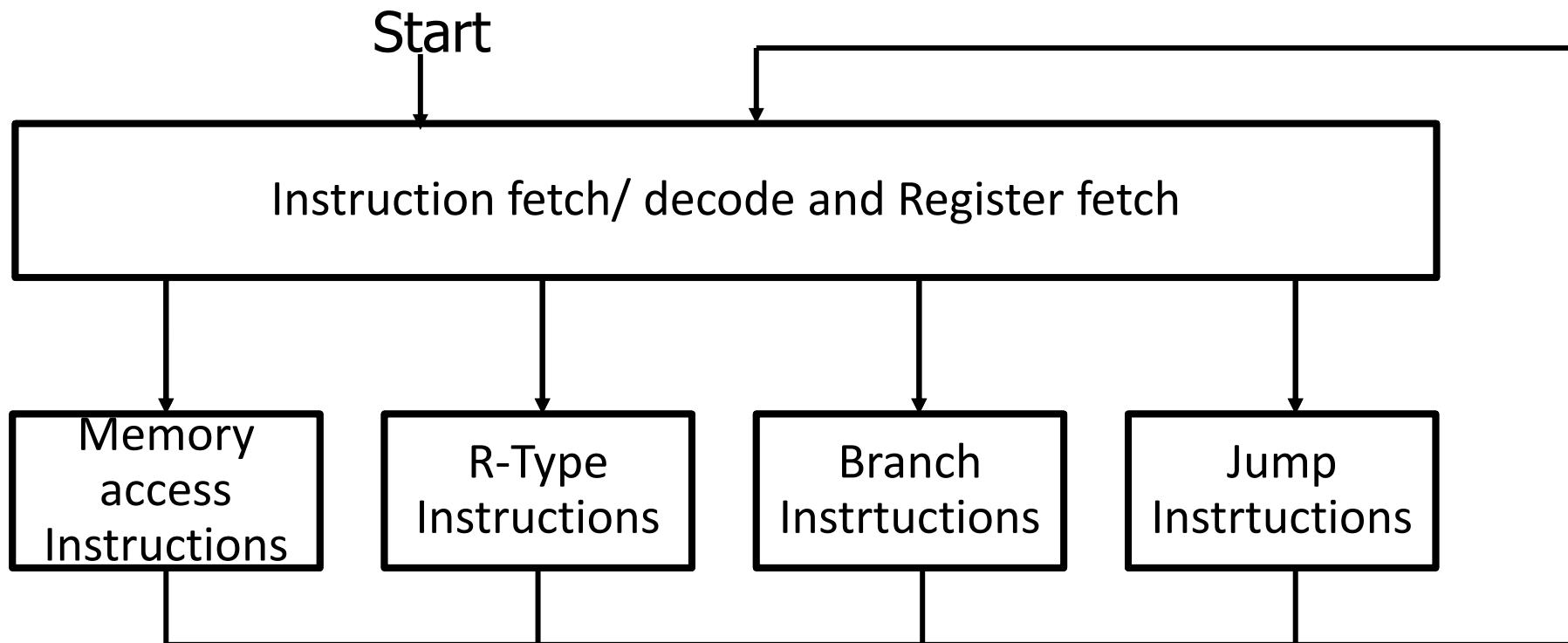
Control Unit: Control Signal

Signal Name	Effect when deasserted	Effect when asserted
RegDst	The register destination number for the Write-register comes from the rt field (bits 20-16)	The register destination number for the Write register comes from the rd field (bits 15-11)
RegWrite	None	
ALUSrc	The second ALU operand comes from the second register file output (Read data 2)	The second ALU operand is the sign-extended, lower 16 bits of the instruction
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4	The PC is replaced by the output of the adder that compute the branch target
MemRead	None	Data memory contents designated by the address input are put on the first Read-data output
MemWrite	None	Data memory contents designated by the address input are replaced by the value of the Write-data input
MemtoReg	The value fed to the register Write data input comes from the ALU	The value fed to the register Write-data input comes from the data memory

Multi-cycle Approach

- Break up the instruction cycle to multiple step (perform 1 step/clock cycle)
 - Allow different instructions to be executed in different number of cycles
 - Reuse expensive hardware on multiple cycles (ALU, memory)
- Only one instruction can be executed at the same time

Multi-cycle Approach: Execute FSM

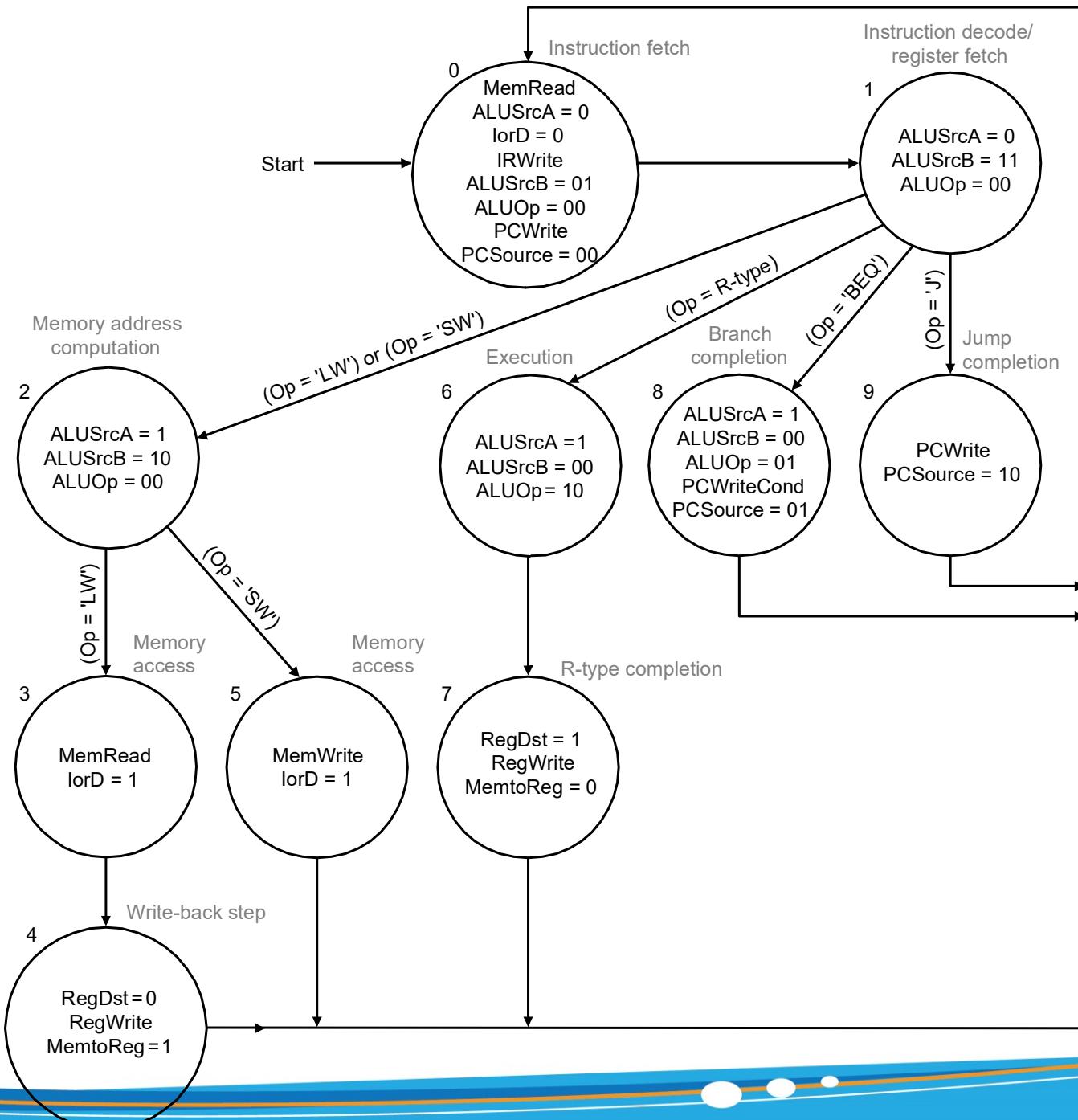


Multi-cycle: Complete Controller FSM

Execute

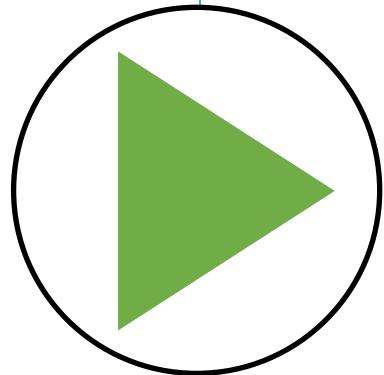
Memory access

Write Back



SIMULATE MIPS R2000 PROCESSOR

- Please watch the video besides



Pipelining: (modern approach)

- Multiple instructions are overlapped in execution
- Fixed the number of clock cycles/ instruction
- Advantage
 - The cycle time of the processor is reduced.
 - It increases the throughput of the system
 - It makes the system reliable.
- Disadvantage
 - The design of pipelined processor is complex and costly to manufacture.
 - The instruction latency is more





CHAPTER

5

X86 INSTRUCTION SET



fit@hcmus

KHOA CÔNG NGHỆ THÔNG TIN
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

REMIND

- CISC
- MIPS-32 bits operations

PREREQUITES

- Take a view on tutorial video
- Install NASM already



What will you learn?

- Inside a CPU Intel 8080/8086
- Registers
- Data addressing modes
- Operations
- Interrupt (I/O calling)
- Instruction Encoding
- Procedure

X86 Architecture

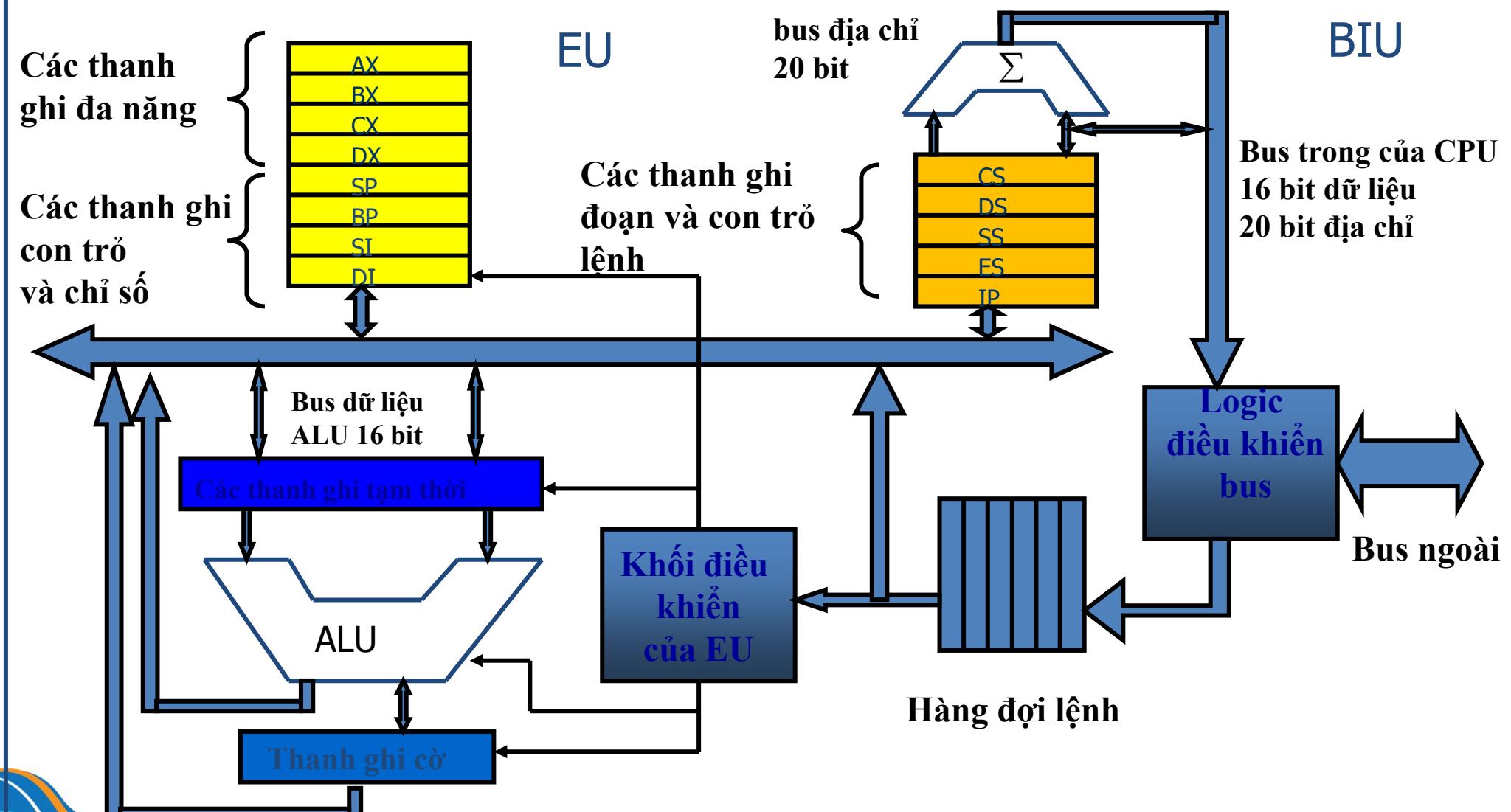
- Complexity
 - instructions from 1 to 17 bytes long
 - one operand *must* act as both a source and destination
 - one operand *may* come from memory
 - several complex addressing modes
- Saving grace:
 - the most frequently used instructions are not too difficult to build
 - compilers avoid the portions of the architecture that are slow



The Intel x86 ISA

- 1978: The Intel 8086 is announced (16 bit architecture)
- 1980: The 8087 floating point coprocessor is added
- 1982: The 80286 increases address space to 24 bits, +instructions
- 1985: The 80386 extends to 32 bits, new addressing modes
- 1989-1995: The 80486, Pentium, Pentium Pro add a few instructions (mostly designed for higher performance)
- 1997: MMX is added
- I486: pipelined, on chip cache
- 2017: Core i9 (x86-64), instruction set extensions SSE4.1, SSE4.2, AVX2

Inside a 8086- CPU



Register Files

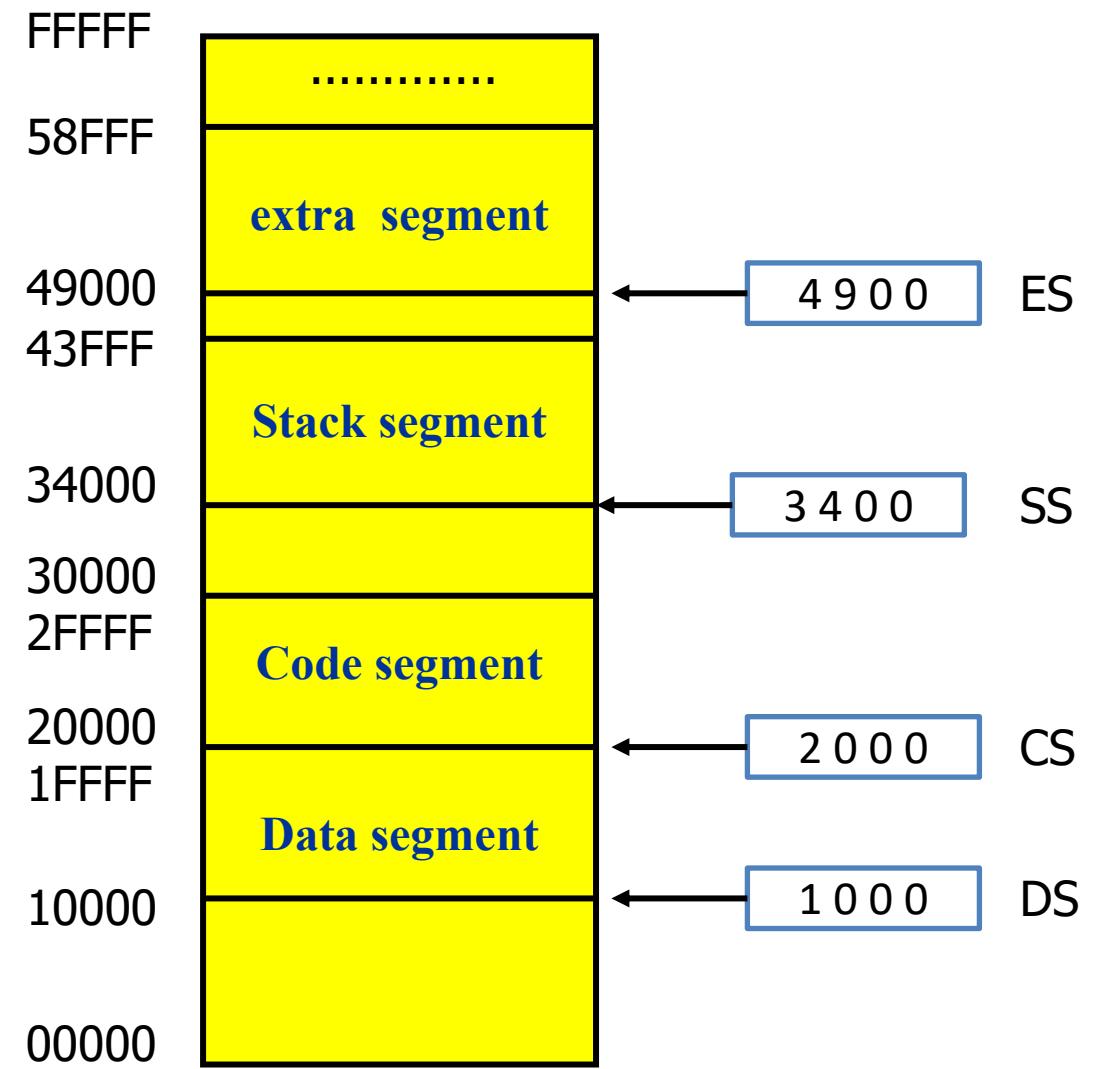
- *H register holds the 8-bit high,*L holds the 8-bit low of *X register
- AX (accumulator): hold the result of operations
- BX (base): base address
- CX (count): the number of times (Loop). CL saves the number of bits in shift / rotate bit operations
- DX (data): combine with AX to hold the result of multiplication/ division. DX also holds the address of in/output port (IN/OUT)

		8-bit high	8-bit low
General purpose registers	AX	AH	AL
	BX	BH	BL
	CX	CH	CL
	DX	DH	DL
	SI		
	DI		
Pointer	SP		
	BP		

- 8088/8086 đến 80286 : 16 bits
- 80386 trở lên: 32 bits EAX, EBX, ECX, EDX
- Y86-64: 64 bits RAX, RBX, RCX, RDX

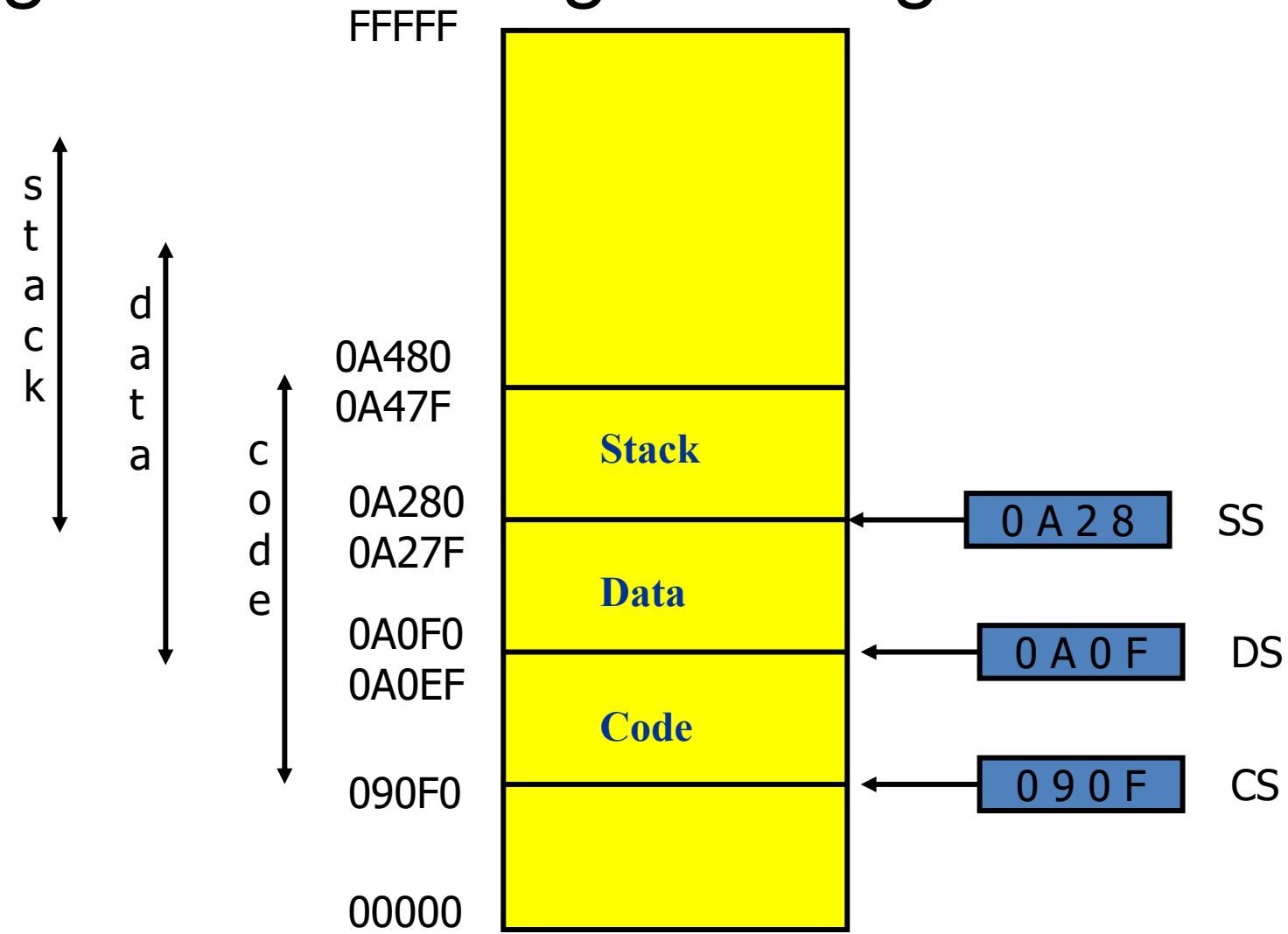
Register Files- Segment registers

- 8088/8086 đến 80286 : 16 bits
- 80386 trở lên: 32 bits EAX, EBX, ECX, EDX
- Y86-64: 64 bits RAX, RBX, RCX, RDX



Register Files- Segment registers

- Nested segments: there is an overlap between the data segment, code segment and stack segment



Pointer register and Indexed

Segment	Offset	Meaning
CS	IP	Instruction address
SS	SP hoặc BP	Stack address
DS	BX, DI, SI, số 8 bit hoặc số 16 bit	Data segment address
ES	DI	Destination string address

Flag register



6 bits are used to be status flags:

- C/CF (carry flag): CF=1
- P/PF (parity flag): PF=1 (0) when the number of 1's bit in the result is even (odd)
- A/AF (auxiliary carry flag): extended carry flag
- Z/ZF (zero flag): ZF=1 when the result is 0
- S/SF (Sign flag): SF=1 when the result is less than 0
- O/OF (Overflow flag): overflow detected in signed number computation

Flag register



3 bits are used to be control flags:

- T/TF (trap flag): used for on chip debugging, TF=1 CPU will work in a single step mode. Generate an interrupt after each instruction
- I/IF (Interrupt enable flag): I = 1, CPU will recognize the interrupts from peripherals. For I = 0, the interrupts will be ignored
- D/DF (direction flag: D=1 the string will be accessed from higher memory address to lower memory address, and if D = 0, it will do the reverse

Data Addressing Mode

- Immediate
- Direct
- Indirect
- Register Direct
- Register Indirect
- Relative
- Indexed

Data Addressing Mode

Mode	Description	Register restrictions	MIPS equivalent
Register Indirect	Address is in a register.	Not ESP or EBP	<code>lw \$s0,0(\$s1)</code>
Based mode with 8- or 32-bit displacement	Address is contents of base register plus displacement.	Not ESP	<code>lw \$s0,100(\$s1) #<= 16-bit # displacement</code>
Base plus scaled index	The address is $\text{Base} + (2^{\text{Scale}} \times \text{Index})$ where Scale has the value 0, 1, 2, or 3.	Base: any GPR Index: not ESP	<code>mul \$t0,\$s2,4 add \$t0,\$t0,\$s1 lw \$s0,0(\$t0)</code>
Base plus scaled Index with 8- or 32-bit displacement	The address is $\text{Base} + (2^{\text{Scale}} \times \text{Index}) + \text{displacement}$ where Scale has the value 0, 1, 2, or 3.	Base: any GPR Index: not ESP	<code>mul \$t0,\$s2,4 add \$t0,\$t0,\$s1 lw \$s0,100(\$t0) #<=16-bit # displacement</code>

Data Addressing Mode

- Assume the following are stored as an indicated memory address and register
- Fill in the following table showing the value for indicated operands:

Address	Value	Register	Value
0x100	0xFF	%rax	0x100
0x104	0xAB	%rcx	0x1
0x108	0x13	%rdx	0x3
0x10C	0x11		

Operand	Value
%rax	_____
0x104	_____
\$0x108	_____
(%rax)	_____
4(%rax)	_____
9(%rax, %rdx)	_____
260(%rcx, %rdx)	_____
0xFC(,%rcx,4)	_____
(%rax,%rdx,4)	_____

Data Addressing Mode

- Assume the following are stored as an indicated memory address and register
- Fill in the following table showing the value for indicated operands: (Solutions)

Address	Value	Register	Value
0x100	0xFF	%rax	0x100
0x104	0xAB	%rcx	0x1
0x108	0x13	%rdx	0x3
0x10C	0x11		

Operand	Value	Comment
%rax	0x100	Register
0x104	0xAB	Absolute address
\$0x108	0x108	Immediate
(%rax)	0xFF	Address 0x100
4(%rax)	0xAB	Address 0x104
9(%rax, %rdx)	0x11	Address 0x10C
260(%rcx, %rdx)	0x13	Address 0x108
0xFC(,%rcx,4)	0xFF	Address 0x100
(%rax,%rdx,4)	0x11	Address 0x10C

OPERATIONS

- Data movement instructions
- Arithmetic and Logic instructions
- Control flow
- String instructions (include string move & compare)



Data movement instructions

- MOV: The mov instruction copies the data item referred to by its second operand into the location referred to by its first operand

Syntax:

```
mov <reg>,<reg>
mov <reg>,<mem>
mov <mem>,<reg>
mov <reg>,<const>
mov <mem>,<const>
```

Example:

copy the value in %bx into %ax
`mov %AX, %BX`
store the value 5 into the byte at location var
`mov byte ptr[var], 5`

Data movement instructions

- PUSH: places its operand onto the top of the hardware supported stack in memory.

Syntax

```
push <reg32>
push <mem>
push <con32>
```

Example:

Push %eax on the stack
push %EAX
push the 4 bytes at address var onto the stack
push [var]

Data movement instructions

- POP: removes the 4-byte data element from the top of the hardware-supported stack into the specified operand.

Syntax

pop <reg32>

pop <mem>

Example:

pop the top element of the stack into EDI

pop %EDI

pop the top element of the stack into memory at the four bytes starting at location EBX

push [%EBX]

Data movement instructions

□ LEA: Load effective address.

Syntax

`lea <reg32> <mem>`

Example:

*the quantity EBX+4*ESI is placed in EDI*

`lea edi, [ebx+4*esi]`

the value in var is placed in EAX

`lea eax, [var]`

the value val is placed in EAX

`lea eax, [val]`

Arithmetic and Logic instructions

- ADD: adds together its two operands, storing the result in its first operand.

Syntax

add <reg>, <reg>
add <reg>, <mem>
add <mem>, <reg>
add <reg>, <con>
add <mem>, <con>

Example:

EAX \leftarrow *EAX* + 5
add %eax, 5
*add 5 to the single byte stored at
memory address var*
add BYTE PTR[var], 5

Arithmetic and Logic instructions

- SUB: adds together its two operands, storing the result in its first operand.

Syntax

sub <reg>, <reg>
sub <reg>, <mem>
sub <mem>, <reg>
sub <reg>, <con>
sub <mem>, <con>

Example:

$AL \leftarrow AL - AH$
sub %AL, %AH
subtract 5 from the value stored at %EAX
sub %EAX, 5



Arithmetic and Logic instructions

- INC/DEC: increments/ decrements the contents of its operand by one.

Syntax

inc <reg>
inc <mem>
dec <reg>
dec <mem>

Example:

add one to the 32-bit integer stored at location var

inc DWORD PTR [var]

subtract 1 from the contents of %EAX

dec %EAX

Arithmetic and Logic instructions

- iMUL: two basic formats: two-operand (first two syntax listings above) and three-operand (last two syntax listings above)

Syntax

```
imul <reg32>,<reg32>
imul <reg32>,<mem>
imul <reg32>,<reg32>,<con>
imul <reg32>,<mem>,<con>
```

Example:

multiply the contents of EAX by the 32-bit contents of the memory location var. Store the result in EAX

```
imul %EAX, [var]
%ESI ← %EDI * 25
imul %ESI, %EDI, 25
```

Arithmetic and Logic instructions

- iDIV: divides the contents of the 64 bit integer EDX:EAX by the specified operand value. The quotient result of the division is stored into EAX, while the remainder is placed in EDX

Syntax

```
idiv <reg32>
idiv <mem>
```

Example:

divide the contents of EDX:EAX by the contents of EBX

idiv %BX

divide the contents of EDX:EAX by the 32-bit value stored at memory location var.

idiv DWORD PTR [var]

Arithmetic and Logic instructions

□ CMP: Compare the values of the two specified operands, setting the condition codes in the machine status word appropriately (based on flag register)

- Đích = nguồn : CF=0 ZF=1
- Đích > nguồn : CF=0 ZF=0
- Đích < nguồn : CF=1 ZF=0

Syntax

```
cmp <reg>,<reg>
cmp <reg>,<mem>
cmp <mem>,<reg>
cmp <reg>,<con>
```

Example:

If the 4 bytes stored at location var are equal to the 4-byte integer constant 3, jump to the location labeled loop

```
cmp DWORD PTR [var], 3
jeq loop
```

Arithmetic and Logic instructions

- AND, OR, XOR: Bitwise logical and, or and exclusive or.
Placing the result in the first operand location

Syntax

opcode <reg>, <reg>

opcode <reg>, <mem>

opcode <mem>, <reg>

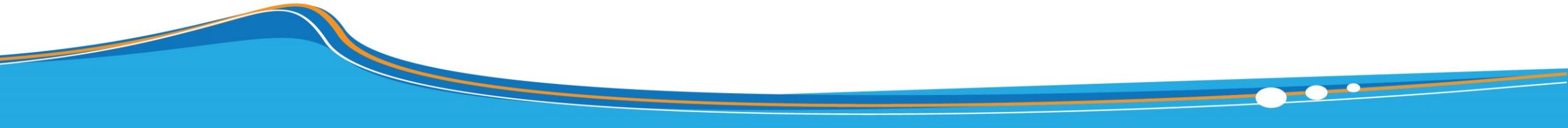
opcode <reg>, <con>

opcode <mem>, <con>

Example:

clear all but the last 4 bits of EAX
and %EAX, 0fH

set the contents of EDX to zero
xor %EDX, %EDX



Arithmetic and Logic instructions

- SHL, SHR: shift the bits in their first operand's contents left and right, padding the resulting empty bit positions with zeros

Syntax

opcode <reg>, <con8>
opcode <mem>, <con8>
opcode <reg>, <cl>
opcode <mem>, <cl>

Example:

Multiply the value of EAX by 2 (if the most significant bit is 0)
shl %EAX, 1
Store in EBX the floor of result of dividing the value of EBX by 2^n where n is the value in CL
shr %EBX, %CL

Control flow instructions

- JMP: transfers program control flow to the instruction at the memory location indicated by the operand
- 3 types of JMP instruction: JUMP SHORT(short jump), JUMP NEAR (near jump), JUMP FAR (far jump)

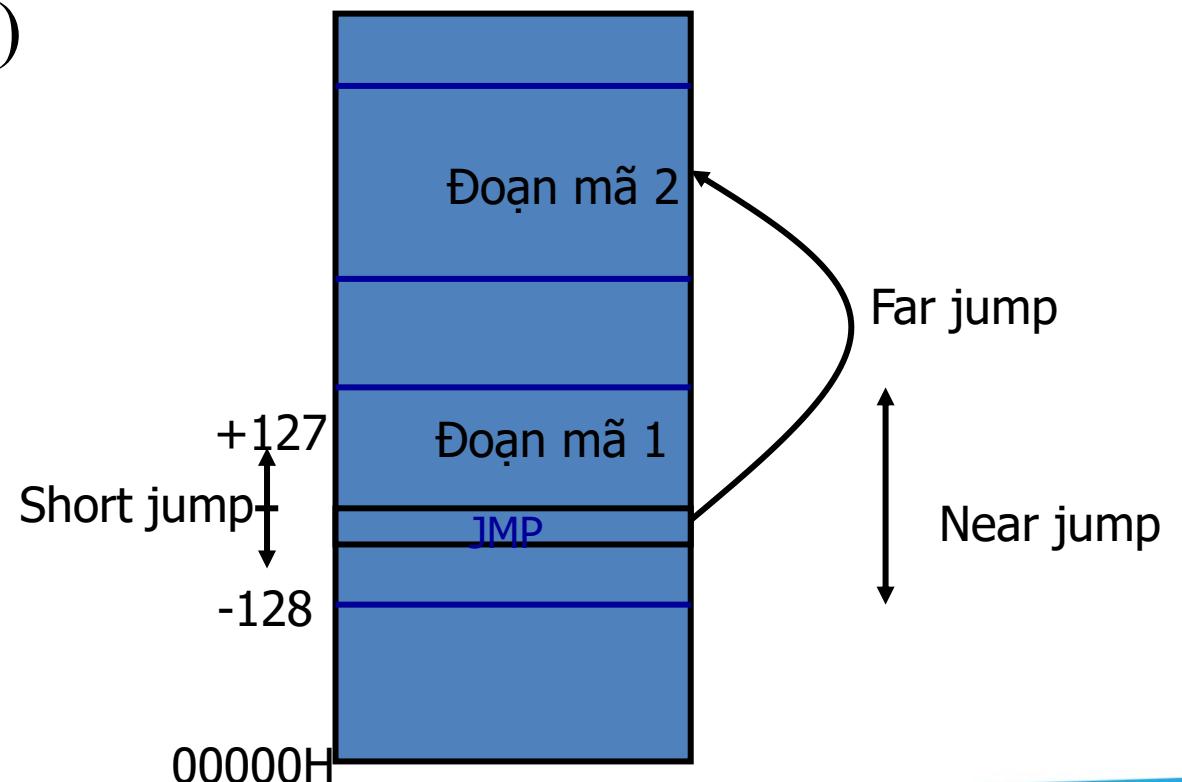
Syntax

```
jmp <Label>
```

Example:

Jump to the instruction
named "BEGIN"

```
Jmp BEGIN
```



Control flow instructions

- JE, JNE, JZ, JG, JGE, JL, JLE: Conditional jump
- Based on the status of a set of condition codes that are stored in a special register called the machine status word

Syntax

opcode <Label>

Example:

Jump to the instruction named “DONE”
if the condition satisfies

```
cmp %EAX, %EBX  
jle done
```



Control flow instructions

- LOOP, LOOPE/LOOPZ, LOOPNE/LOOPNZ: is a combination instruction of DEC CX and JNZ

Syntax

<Label:>

Task

Loop <Label>

Example:

Repeat when CX != 0. Decrements CX
after each loop

XOR AL, AL

MOV CX, 16

myloop:

INC AL

LOOP myloop

String instructions

- MOVS, MOVSB, MOVSW: copy from the string source (located in data segment) to destination (located in extra segment) by increment ESI and EDI; may be repeated

Example:

```
move a string of length 4 bytes from source to destination
MOV SI, SRC
MOV DI, DST
MOV CX, 04H
CLD; Clear the direction flag
REP MOVSB
```

Instruction Encoding

Variable length encoding

- Postfix bytes specify addressing mode
- Prefix bytes modify operation
Operand length,
repetition, locking, ...

a. JE EIP + displacement

4	4	8
JE	Condition	Displacement

b. CALL

8	32
CALL	Offset

c. MOV EBX, [EDI + 45]

6	1	1	8	8
MOV	d	w	r/m Postbyte	Displacement

d. PUSH ESI

5	3
PUSH	Reg

e. ADD EAX, #6765

4	3	1	32
ADD	Reg	w	Immediate

f. TEST EDX, #42

7	1	8	32
TEST	w	Postbyte	Immediate

Interrupt I/O

- Generated by the software
- 2 types:
 - Use IN, OUT command to swap with out of external devices
 - Use DOS and BIOS interrupt server subroutines (independent on system)



Interrupt I/O

- Interrupt 21h of DOS

Code (AH)	Meaning	Arguments	Result
1	Read a character		AL
2	Print a character	DL = ascii code of the character	
9	Print string with null character in the end	DX = address of the string	
4CH	End of .exe program		



CHAPTER

7

MEMORY HIERARCHY



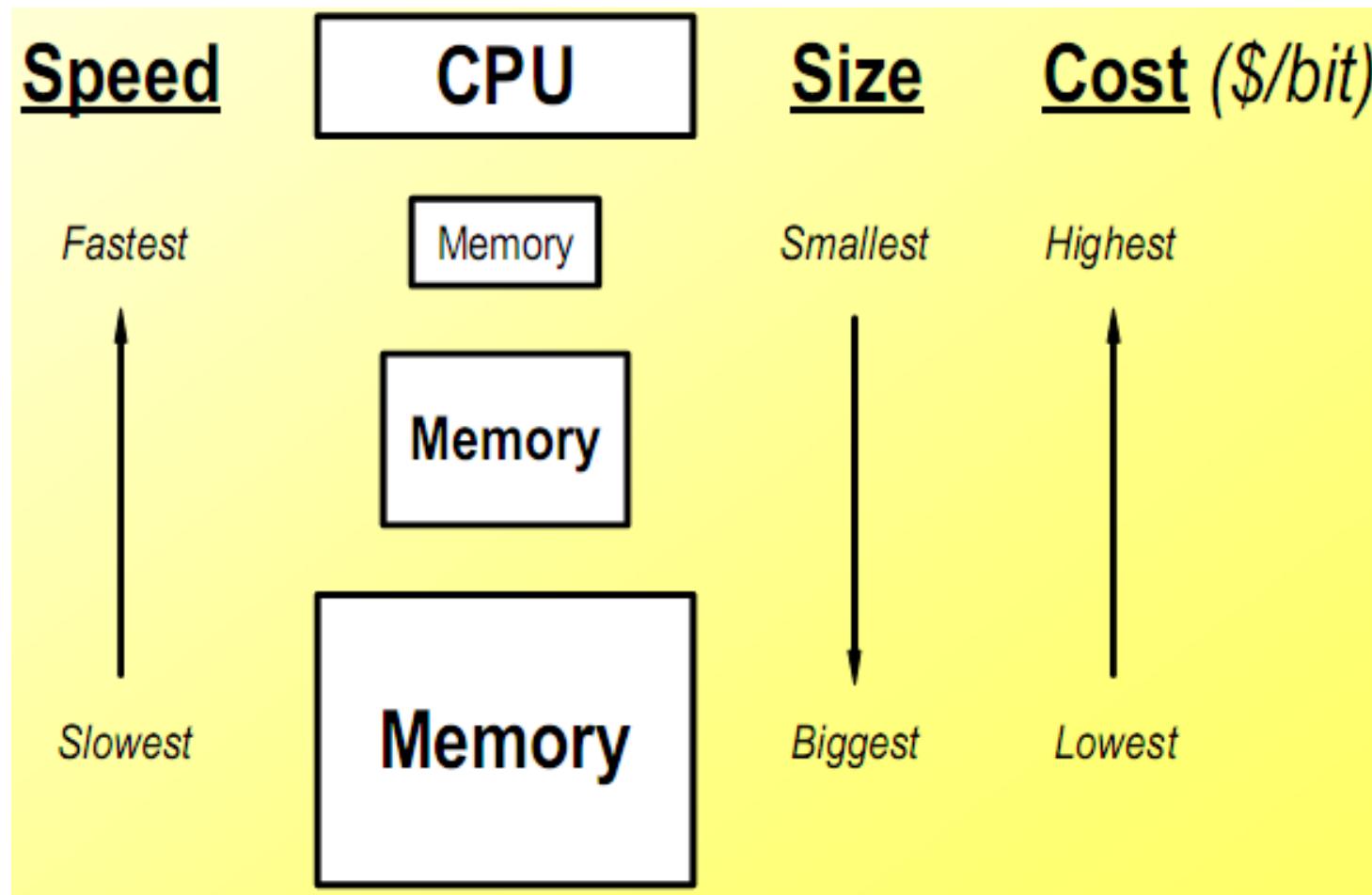
fit@hcmus

KHOA CÔNG NGHỆ THÔNG TIN
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

What will you learn?

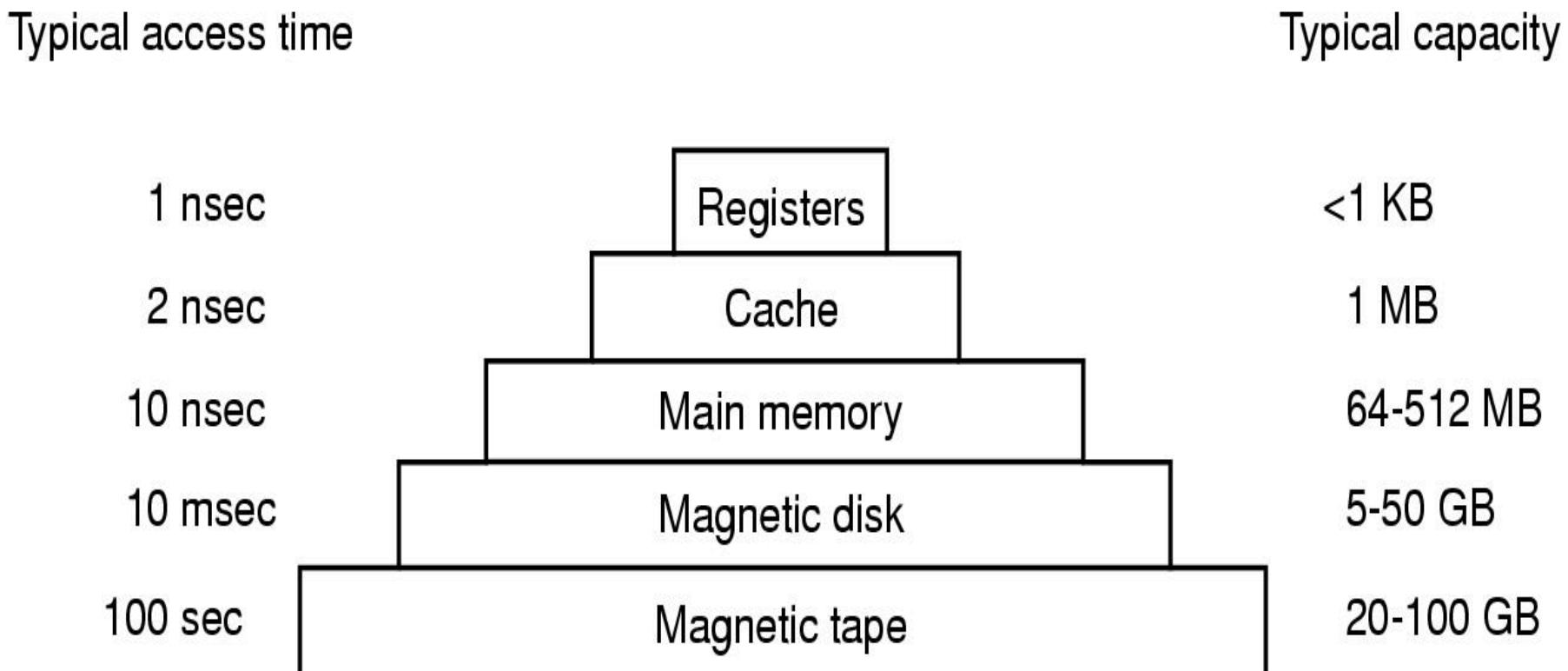
- Memory hierarchy levels
 - Memory Technology
 - Principle of locality
 - Cache Memory
 - Cache addressing scheme
- 
- Replacement Policy
 - Write Policy
 - Parameter influent to Cache performance
 - Interactions with advanced CPUs & Software
 - Writing cache-friendly code

Memory Hierarchy Levels



Memory Hierarchy Levels

Example:



Memory Technology

□ Access types:

- Serial/Sequential
- Direct
- Random

□ Physical types:

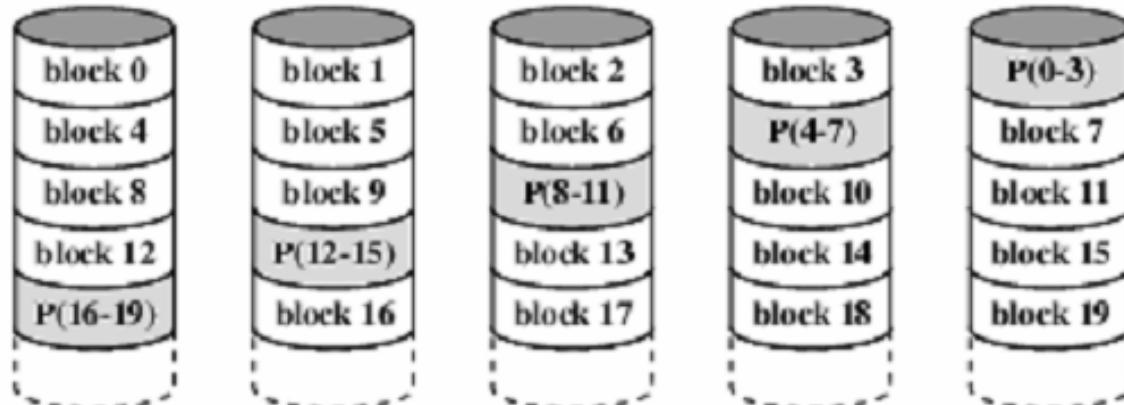
- Transistor (cache, register, RAM, ROM)
- Magnetic disk (HDD, FDD)
- Flash (CD, DVD, SD, SSD)



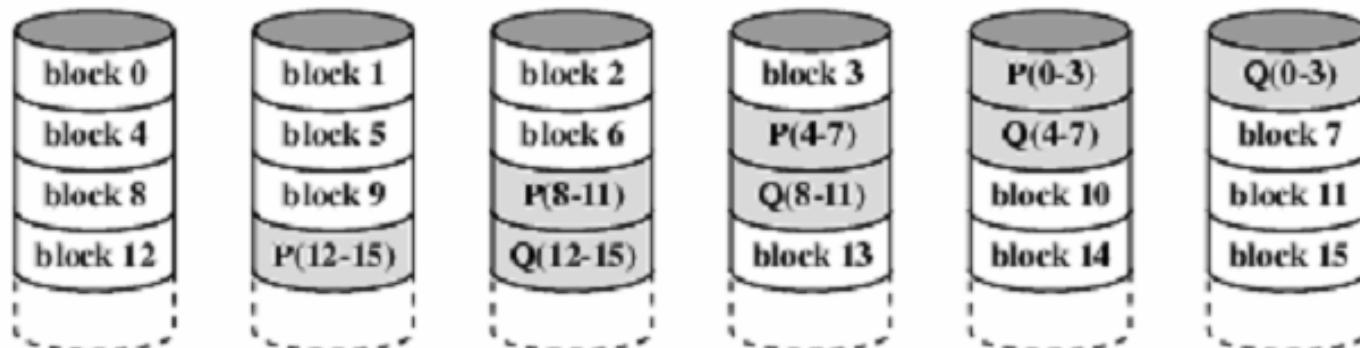
Mass storage device: RAID

- Redundant Array of Inexpensive (Independent) Disks
- A data storage technology that combines multiple physical disk drive components into one or more logical units
 - Storing data in distributed physical disk
 - Using parity bits/ check byte to check data errors
- The common RAID system: RAID 0 → RAID 6

Mass storage device: RAID



(f) RAID 5 (block-level distributed parity)

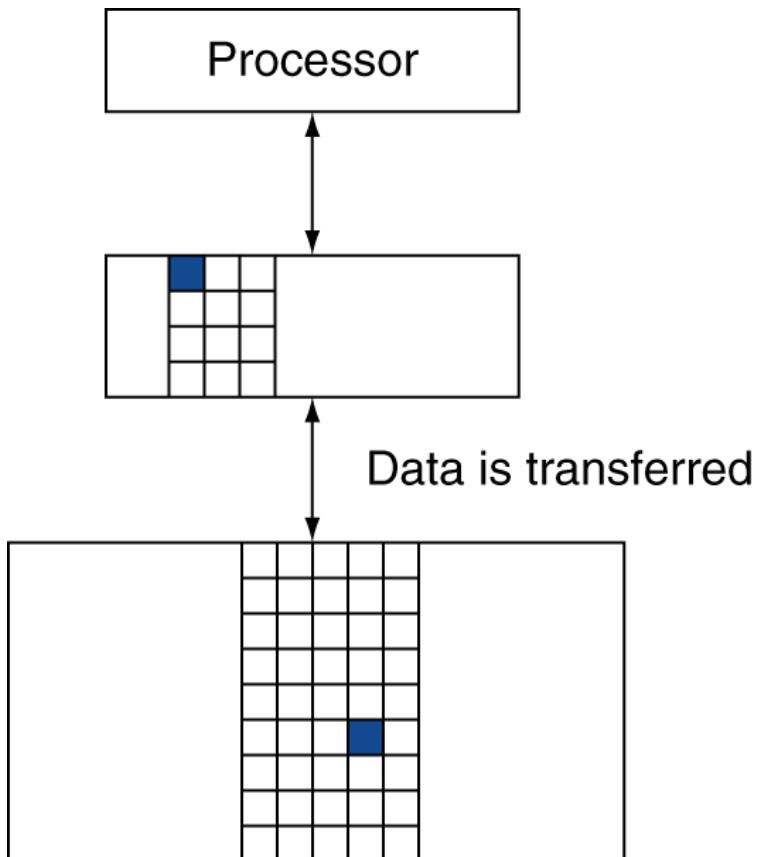


(g) RAID 6 (dual redundancy)

Principle of Locality

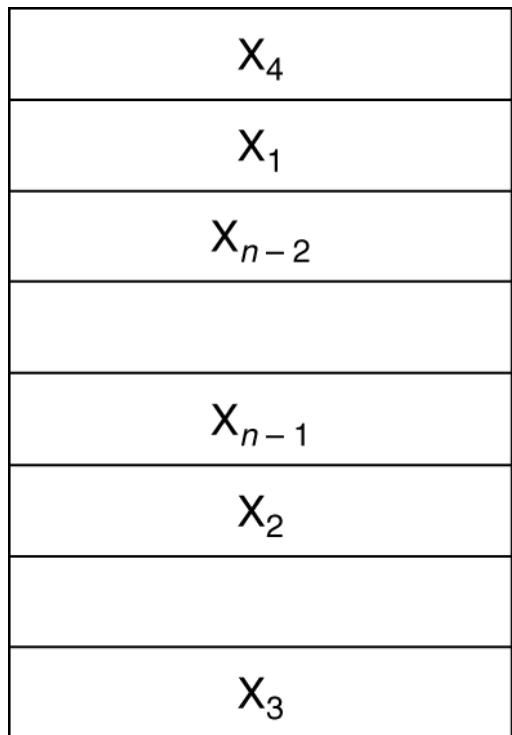
- Programs access a small portion of their address space at any time
- Temporal locality
 - Items accessed recently are likely to be accessed again soon
 - e.g., instructions in a loop, induction variables
- Spatial locality
 - Items near those accessed recently are likely to be accessed soon
 - E.g., sequential instruction access, array data

Hit and Miss

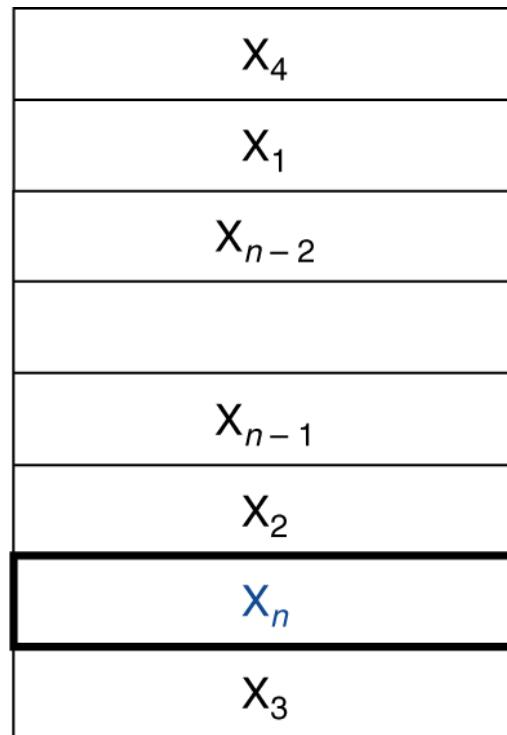


- If accessed data is present in upper level
 - **Hit:** access satisfied by upper level
 - Hit ratio = $\frac{hits}{accessed}$
- If accessed data is absent
 - **Miss:** block copied from lower level
 - Miss ratio = $\frac{misses}{accessed} = 1 - hit\ ratio$

Cache Memory



a. Before the reference to X_n



b. After the reference to X_n

- Using SRAM technology which has higher access speed than main memory (using DRAM technology)
- The level of the memory hierarchy closest to the CPU
- A partial copy of the main memory
- Given access $X_1, X_2, \dots, X_{n-1}, X_n$

Reference to X_n causes miss so it is fetched from memory



Discussion

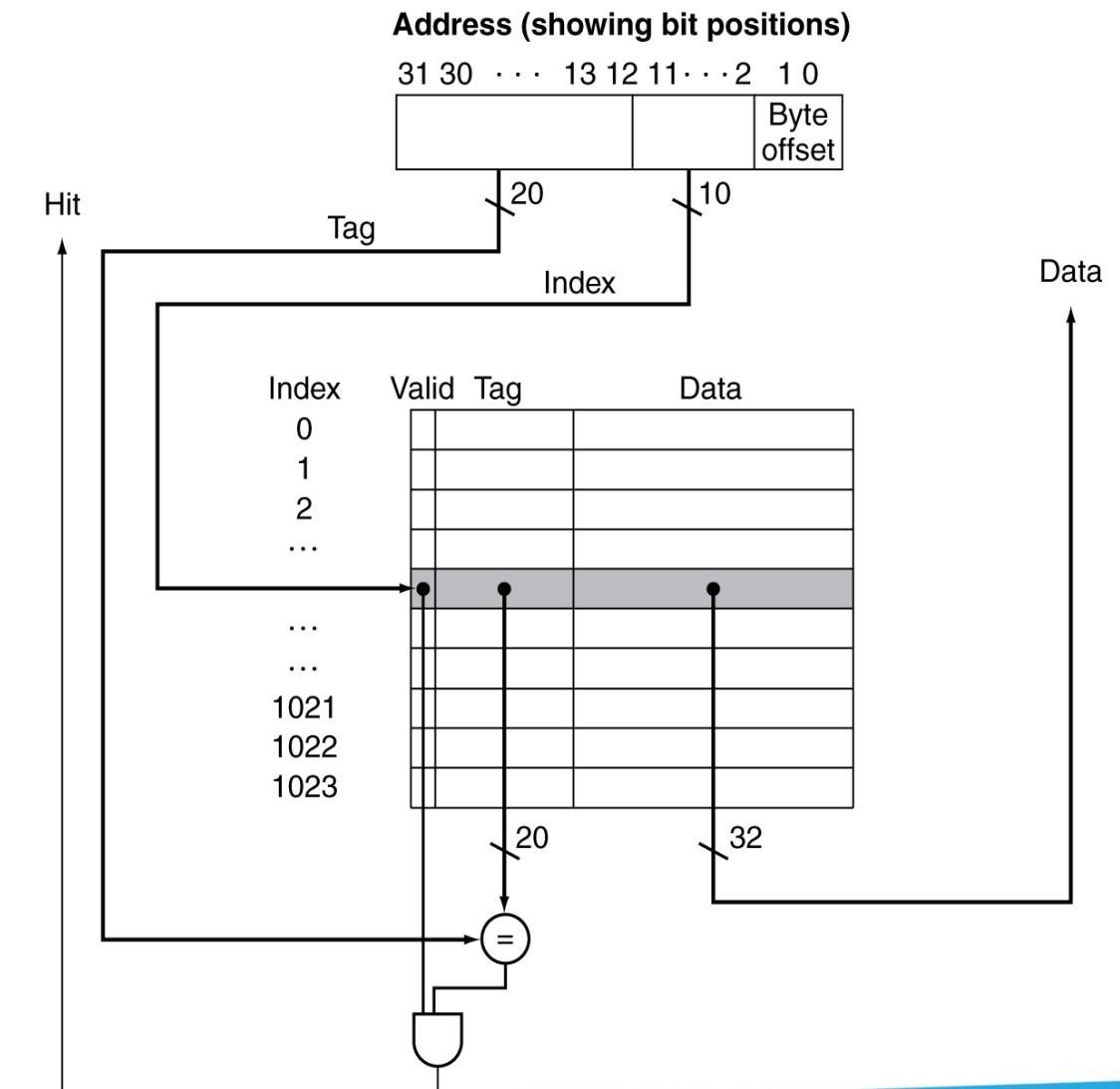
- When you need to access a memory cell, how do you know if that cell is already in the cache? If so, how to identify that place?

- Which memory cells will be selected for loading to cache? When does the selection happen?

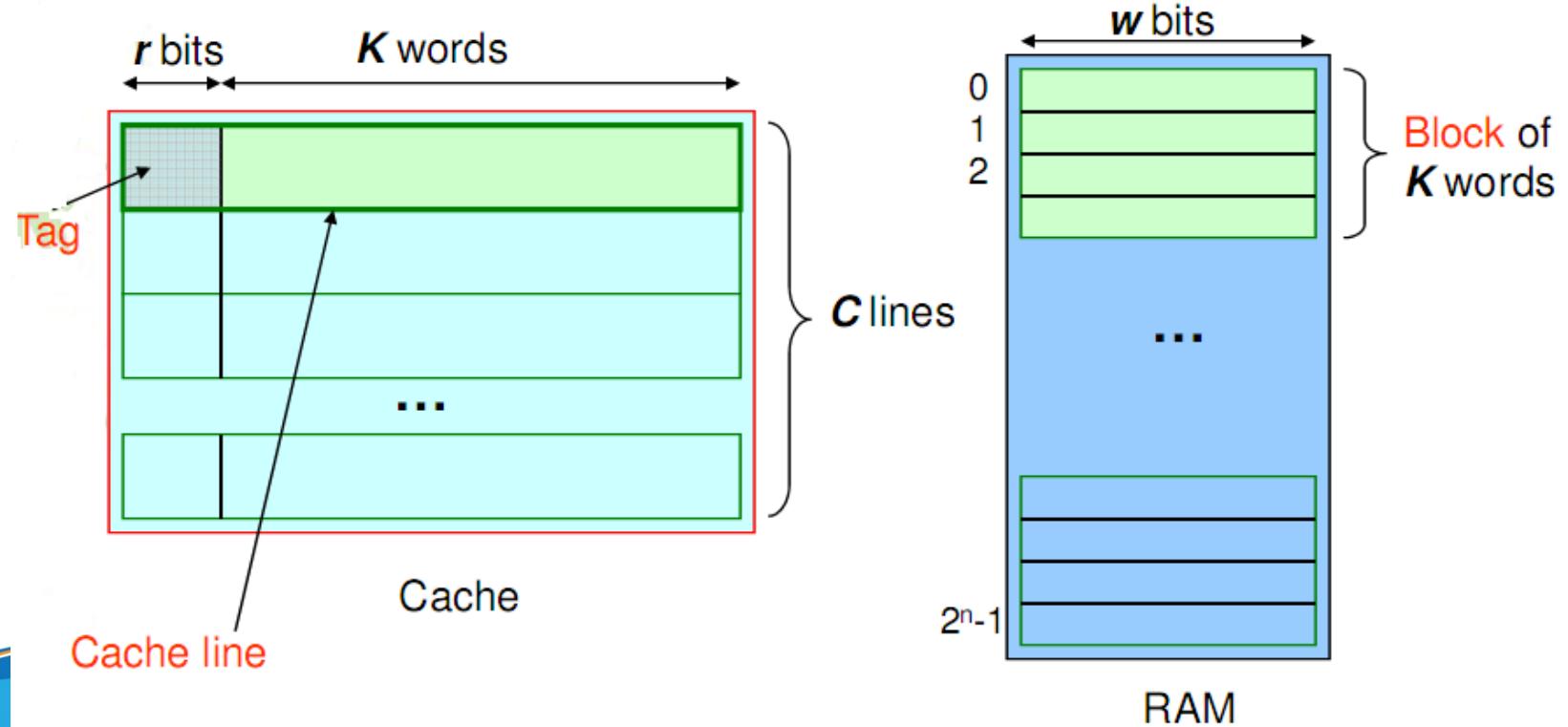
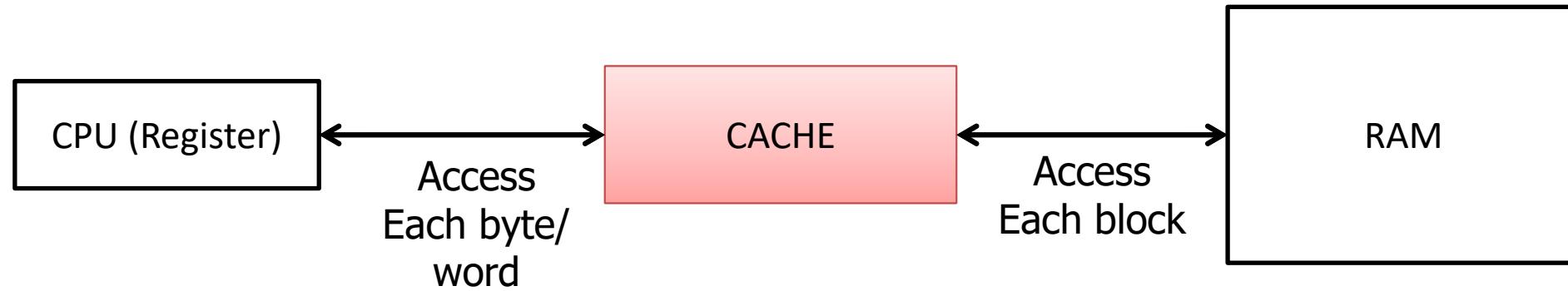


Cache Memory: Structure

Index	Valid	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	Y	11	Mem[11010]
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		



Cache Access Method



Cache Access Method

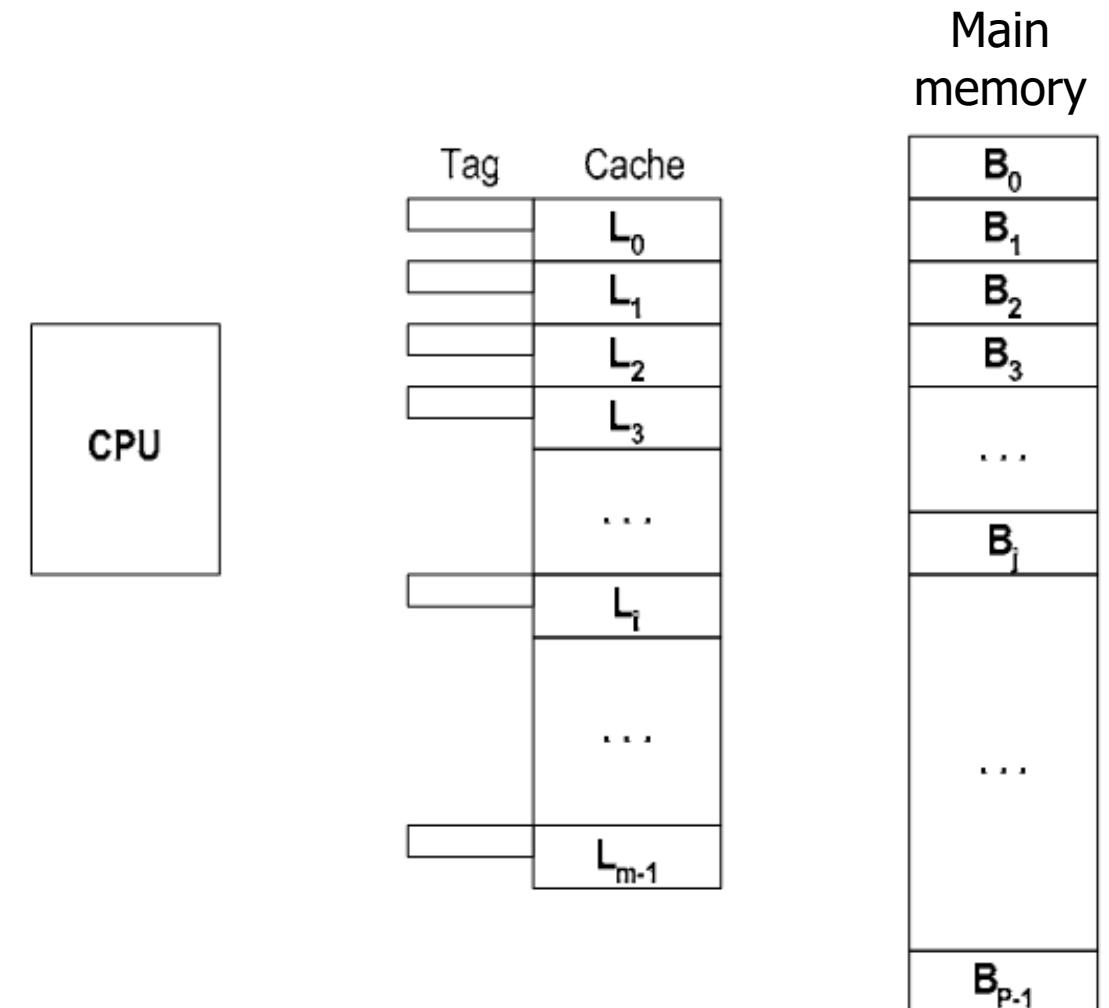
- Main memory has 2^n bytes of memory, numbered $0 \rightarrow 2^n - 1$
- The main memory and the cache are divided into equal-sized blocks
 - 1 block of main memory = 1 line in cache**
- Some main memory blocks are loaded into lines of cache
- **Tag** content shows which block of main memory is currently stored on that line (*not the serial number of that line in the Cache*)

Cache Addressing Schemes

- Direct mapping
- Associative mapping (Full associative mapping)
- Set associative mapping

Direct Mapping

- Each block of main memory can only be loaded into 1 line of cache
- $B_j \rightarrow L_{j \bmod m}$
- m : the number of lines in cache

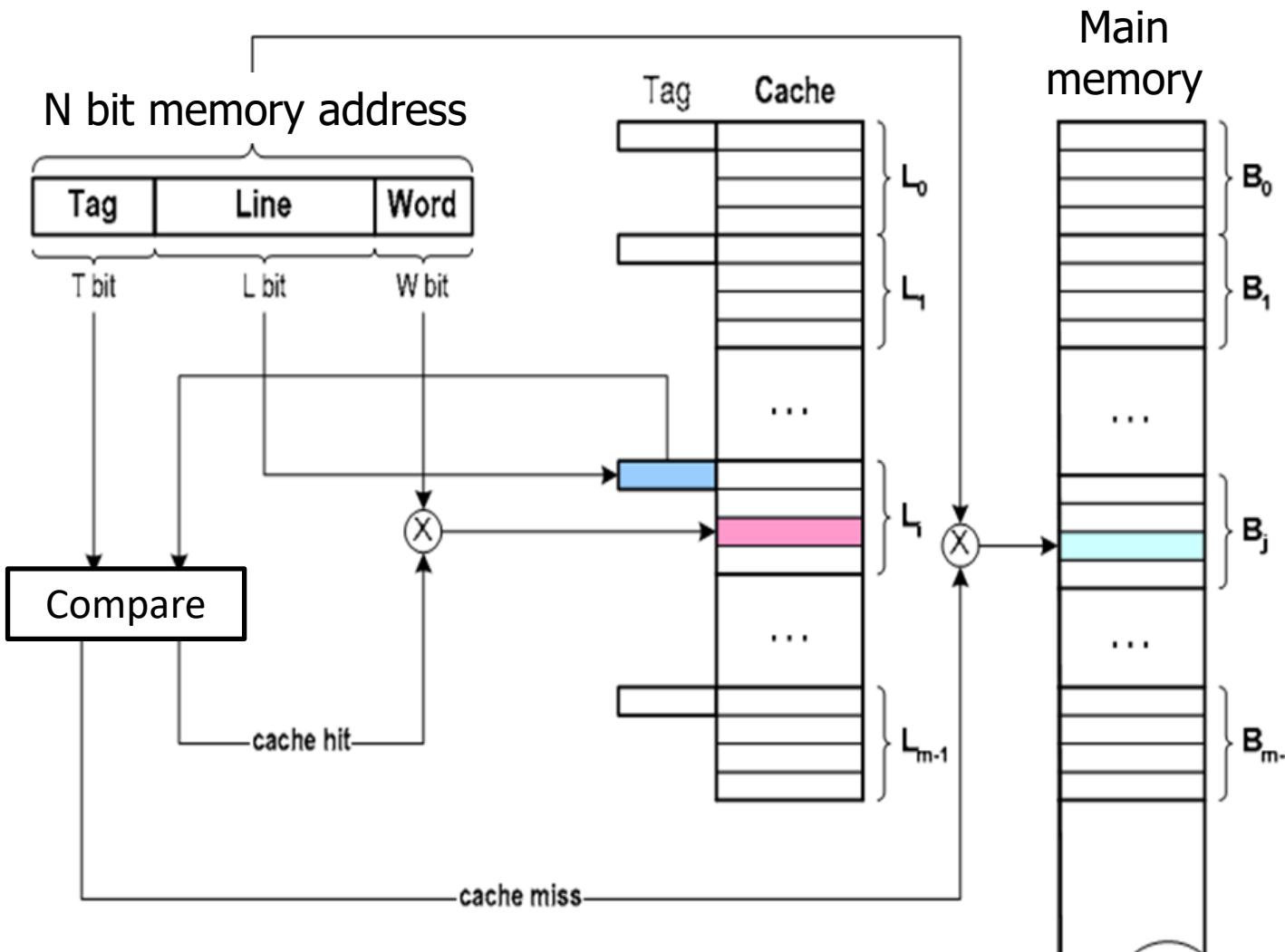


Direct Mapping

Tag	Line	Word
-----	------	------

Each X address in main memory consists of N bits divided into three fields:

- Word (W-bits): Defines the word address in 1 block
- Line (L-bits): Defines the line address in cache
- Tag (T-bits): the block address



Direct Mapping: Example

Ex1: Suppose a direct-mapped cache has 32-bytes blocks, The size of the cache and main memory are respectively are 256KB, 4GB. The machine has 32-bits addresses

- How many address bits are used for the byte offset (Word)?
- How many address bits are used for the index (Line)?
- How many address bits are used for the tag?

Direct Mapping: Example

Solution:

- The size of main memory = 4GB = 2^{32} bytes $\rightarrow N = 32$ bits
- The size of cache = 256KB = 2^{18} bytes \rightarrow using 18 bits to address each memory word in cache
- The size of 1 line in cache = 32 = 2^5 bytes $\rightarrow W = 5$ bits (using 5 bits to address each memory word in a line of cache)
 \rightarrow *The number of lines in cache* = $2^{18}/2^5 = 2^{13}$ Lines $\rightarrow L = 13$ bits
- Tag = $N - L - W = 32 - 13 - 5 = 14$ bits

Tag 14 bits	Line 13 bits	Word 5 bits
----------------	-----------------	----------------

Direct Mapping: Example

Ex2: Reuse the assumptions in example 1. Suppose that we have M^{th} Block (27 bits, value from 0 to $2^{27} - 1$) that want to store in the cache. Where will it be stored in cache?

Solution:

$$\text{The number of blocks in main memory} = 2^{32} / 2^5 = 2^{27}$$

→ Using 27 bit to address 1 block in the main memory

The place of block M in cache :

Line: $L = M \% \text{ the number of lines in cache} = M \% 2^{13}$

Tag: $T = M / \text{the number of lines in cache} = M / 2^{13}$



Direct Mapping

- Advantage:

Simple comparison

- Disadvantage:

Low probability of cache hit

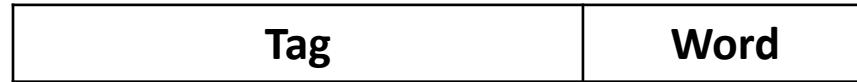
Ex: Suppose that accessing the memory word (cell) X at Block 0 and cell Y at Block 2^L (2^L is the number of lines in cache) at the same time?

→ **Conflict**, both of these cells will be saved in Line 0

$$(0 \% 2^L = 2^L \% 2^L = 0)$$

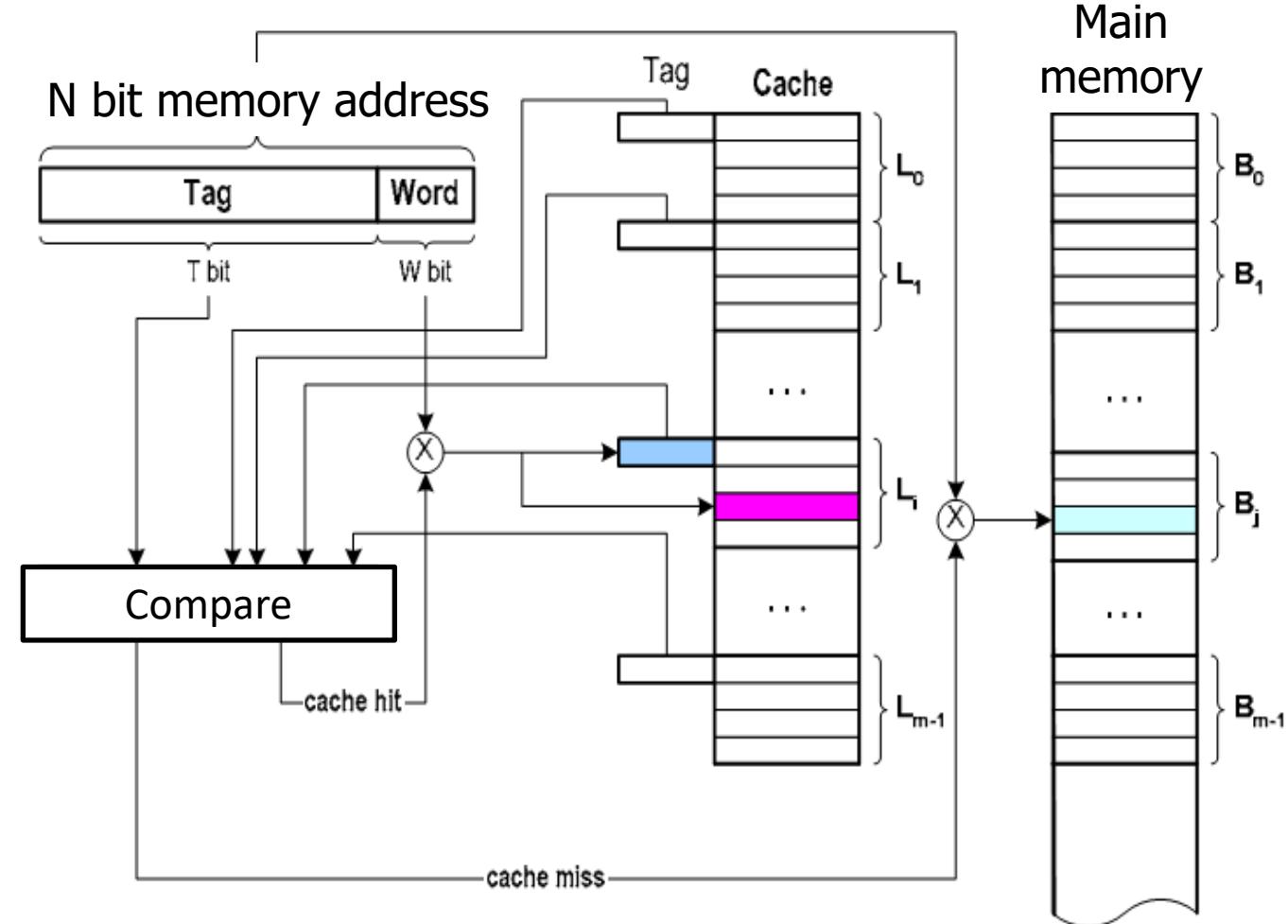


Associative Mapping



Each X address in main memory consists of N bits divided into two fields:

- Word (W-bits): Defines the word address in 1 block
- Tag (T-bits): the block address (which block of main memory is placed in this line)



Direct Mapping: Example

Ex1: Suppose an **associative-mapped cache** has 32-bytes blocks, The size of the cache and main memory are respectively are 256KB, 4GB. The machine has 32-bits addresses

- How many address bits are used for the byte offset (Word)?
- How many address bits are used for the tag?

Associative Mapping: Example

Solution:

- The size of main memory = $4\text{GB} = 2^{32}$ bytes $\rightarrow N = 32$ bits
- The size of 1 line in cache = $32 = 2^5$ bytes $\rightarrow W = 5$ bits (using 5 bits to address each memory word in a line of cache)
- Tag = $N - W = 32 - 5 = 27$ bits

Tag 27 bits	Word 5 bits
------------------------------	------------------------------

Direct Mapping: Example

Ex2: Reuse the assumptions in example 1. Suppose that we have M^{th} Block (27 bits, value from 0 to $2^{27} - 1$) that want to store in the cache. Where will it be stored in cache?

Solution:

$$\text{The number of blocks in main memory} = 2^{32} / 2^5 = 2^{27}$$

→ Using 27 bit to address 1 block in the main memory

The position of the M block in the cache is any line containing the tag equal to M

Tag: $T = M$

Associative Mapping

- Advantage:

- High probability of cache hit

- Disadvantage:

- Complex comparison

To find out which Line contains the content of 1 Block, we need to detect and compare in turn with the Tag of all Lines of the Cache

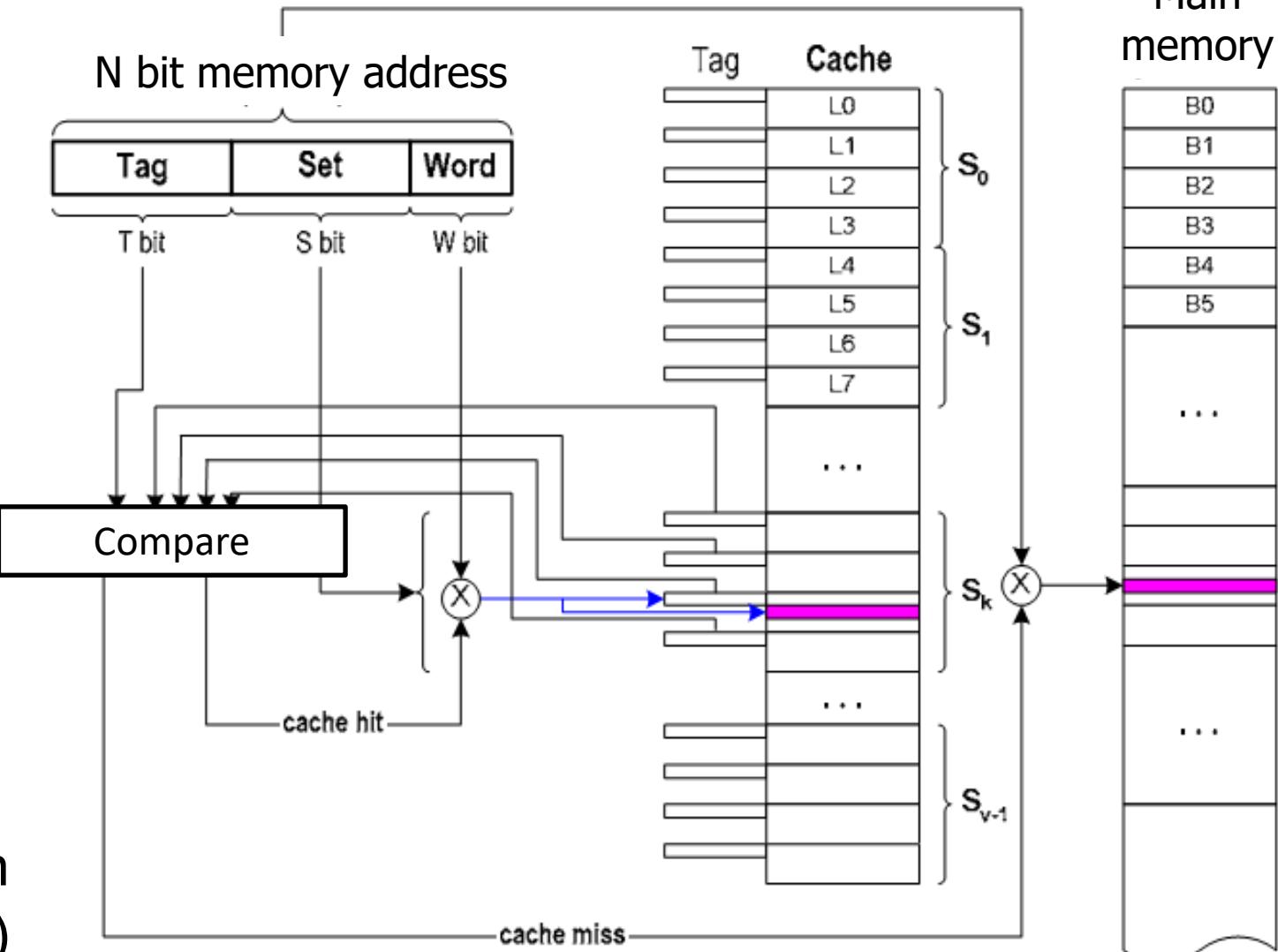
→ Takes a long time to compare

Set Associative Mapping

Tag	Set	Word
-----	-----	------

Each X address in main memory consists of N bits divided into three fields:

- Word (W-bits): Defines the word address in 1 block
- Set (S-bits): Defines the set address in cache, each set contains multiple lines
- Tag (T-bits): the block address (which block of main memory is placed in this line)



Set Associative Mapping: Example

Ex1: Suppose an 4-ways associative-mapped cache has 32-bytes blocks, The size of the cache and main memory are respectively are 256KB, 4GB. The machine has 32-bits addresses

- How many address bits are used for the byte offset (Word)?
- How many address bits are used for the set?
- How many address bits are used for the tag?

Set Associative Mapping: Example

Solution:

- The size of main memory = 4GB = 2^{32} bytes $\rightarrow N = 32$ bits
- The size of cache = 256KB = 2^{18} bytes \rightarrow using 18 bits to address each memory word in cache
- The size of 1 line in cache = 32 = 2^5 bytes $\rightarrow W = 5$ bits (using 5 bits to address each memory word in a line of cache)
- $The\ number\ of\ lines\ in\ cache\ = 2^{18}/2^5 = 2^{13}\ Lines$ \rightarrow using 13 bits to address each line in cache
- The number of lines in each set is 4 (4-ways) = 2^2 lines
- $The\ number\ of\ sets\ in\ cache\ = 2^{13}/2^2 = 2^{11}\ Sets$ $\rightarrow S = 11$ bits (using 11 bits to address each set in cache)
- Tag = $N - S - W = 32 - 11 - 5 = 16$ bits

Tag 16 bits	Set 11 bits	Word 5 bits
------------------------	------------------------	------------------------

Set Associative Mapping

- Advantage:

- High probability of cache hit

- Reduce time to compare

- Disadvantage:

- Complex to implement → the higher cost

Replacement Algorithms

- Random
 - FIFO (First In First Out)
 - LFU (Least Frequently Used)
 - LRU (Least Recently Used)
- The optimal algorithm is LRU



Replacement Policy

- Direct mapped: no choice
- Set associative
 - Prefer non-valid entry, if there is one
 - Otherwise, choose among entries in the set
 - Using LRU algorithm is simple for 2-ways, difficult to manage 4-ways associative-mapped cache
 - Using random algorithm gives approximately the same performance as LRU for high associativity

Discussion

- If a line is changed in the cache, when will the RAM rewrite operation be performed?
- If multiple processors share RAM, each one has its own cache, which block will be updated on RAM?



Write Policy

- Write through
 - Update both upper and lower levels when the data has changed
 - Simplifies replacement, but may require write buffer
- Write back
 - Update upper level only
 - Update lower level when block is replaced
 - Need to keep more state
- Virtual Memory
 - Only write-back is feasible, given disk write latency

Write Policy

- **Bus watching with Write through:**
 - removes the line when it is modified in another cache
- **Hardware transparency**
 - automatically update other caches when the line is changed by one cache
- **Noncacheable shared memory**
 - shared memory will not be put into cache

Multiple level cache

- Level cache:
 - Primary cache: focus on minimal hit time
 - L1: block size smaller than L2 block size
 - L2: focus on low miss rate to avoid main memory access
 - L3,...
- The low-level caches can be on-chip, while the high-level caches are usually off-chip and accessed via external bus or dedicated bus
- The cache can be used for both data and instruction or for each type



Parameter influent to Cache performance

□ Block size:

- Too small: decrease spatial locality
- Too large: the number of blocks in the cache is small, the time to move the block to the cache is long (miss penalty)

□ Cache size:

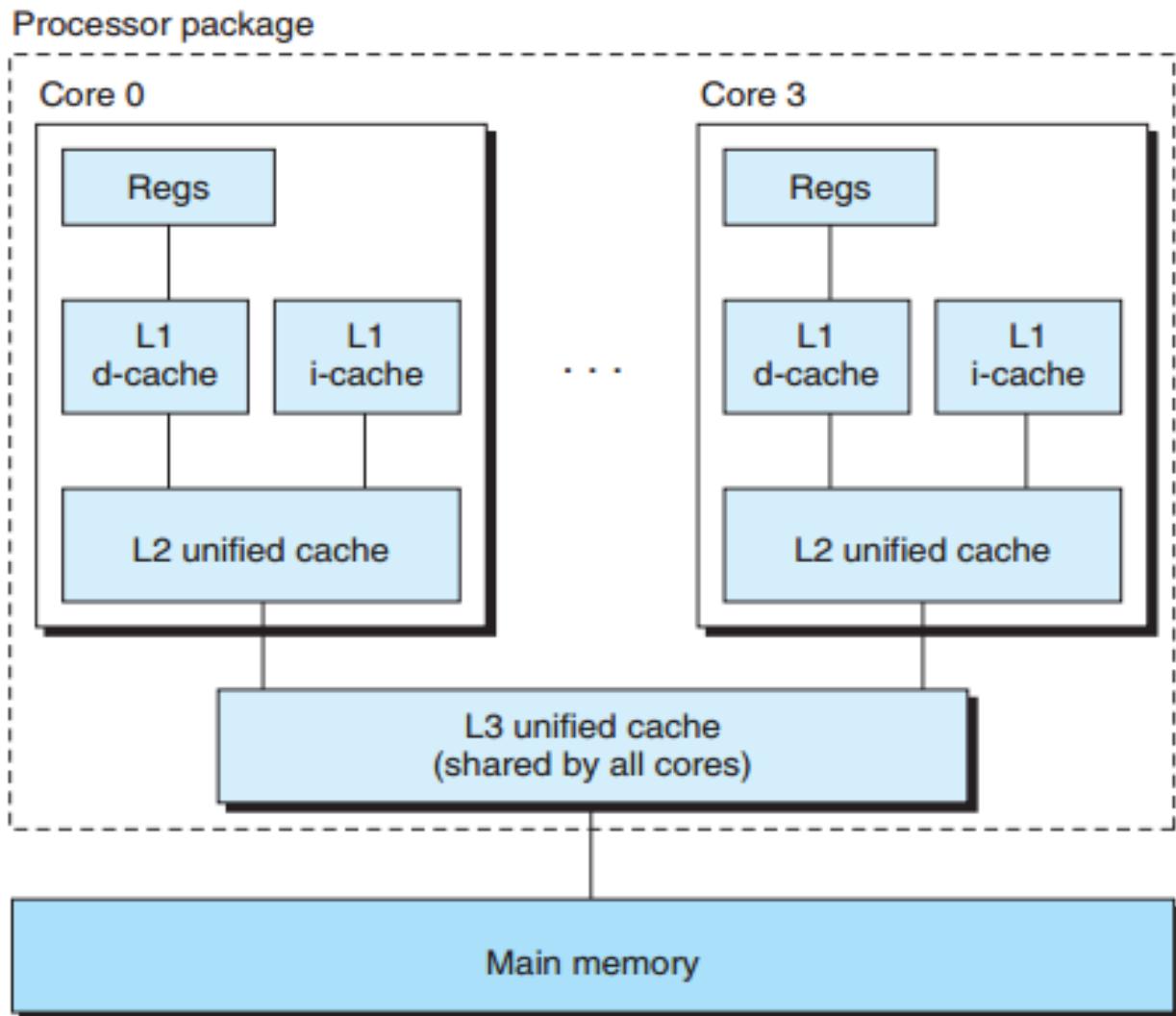
- Too small: The volume that can be stored in the buffer is too small, increasing the rate of cache miss
- Large too: The ratio of the actual memory needed to the cache memory will be lower, meaning overhead will be high, cache access speed will decrease.



Intel core i7 hierarchy

Cache type	Access time (cycles)	Cache size (<i>C</i>)	Assoc. (<i>E</i>)	Block size (<i>B</i>)	Sets (<i>S</i>)
L1 i-cache	4	32 KB	8	64 B	64
L1 d-cache	4	32 KB	8	64 B	64
L2 unified cache	10	256 KB	8	64 B	512
L3 unified cache	40-75	8 MB	16	64 B	8,192

Characteristic of the Intel core i7 cache hierarchy



Interactions with advanced CPUs & Software

- With advanced CPUs:
 - Out-of-order CPUs can execute during cache miss
 - Effect of miss depends on program data flow
- With software:
 - Algorithm behavior
 - Compiler optimization for memory access



Writing cache-friendly code

- Make the common case go fast:
 - The core functions of the program
 - Especially the loops inside functions
- Minimize the number of cache misses in each inner loop:
 - The total number of loads and stores, loops with higher hit rates will run faster
 - Maximize the temporal locality in your programs by using a data object as often as possible once it has been read from memory
 - Maximize the spatial locality in your programs by reading data objects sequentially, in the order they are stored in memory



Writing cache-friendly code

□ Example: Compare the following two code snippets

```
int sumarr(int a[M] [N])
{
    int i, j, sum =0;
    for (i=0; i<M; i++)
        for(j=0; j<N; j++)
            sum += a[i] [j];
    return sum;
}
```

```
int sumarr(int a[M] [N])
{
    int i, j, sum =0;
    for (j=0; j<N; j++)
        for(i=0; i<M; i++)
            sum += a[i] [j];
    return sum;
}
```

a[i] [j]	j = 0	j = 1	j = 2	j = 3	j = 4	j = 5	j = 6	j = 7
i = 0	1 [m]	2 [h]	3 [h]	4 [h]	5 [m]	6 [h]	7 [h]	8 [h]
i = 1	9 [m]	10 [h]	11 [h]	12 [h]	13 [m]	14 [h]	15 [h]	16 [h]
i = 2	17 [m]	18 [h]	19 [h]	20 [h]	21 [m]	22 [h]	23 [h]	24 [h]
i = 3	25 [m]	26 [h]	27 [h]	28 [h]	29 [m]	30 [h]	31 [h]	32 [h]

a[i] [j]	j = 0	j = 1	j = 2	j = 3	j = 4	j = 5	j = 6	j = 7
i = 0	1 [m]	5 [m]	9 [m]	13 [m]	17 [m]	21 [m]	25 [m]	29 [m]
i = 1	2 [m]	6 [m]	10 [m]	14 [m]	18 [m]	22 [m]	26 [m]	30 [m]
i = 2	3 [m]	7 [m]	11 [m]	15 [m]	19 [m]	23 [m]	27 [m]	31 [m]
i = 3	4 [m]	8 [m]	12 [m]	16 [m]	20 [m]	24 [m]	28 [m]	32 [m]



CHAPTER

8

LOGICAL CIRCUIT DESIGN



fit@hcmus

KHOA CÔNG NGHỆ THÔNG TIN
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

REMIND

- Boolean algebra

Read chapter 11 Computer Organization and Architecture
10th edition, William Stallings



PREREQUITES

- Install Logisims tool already

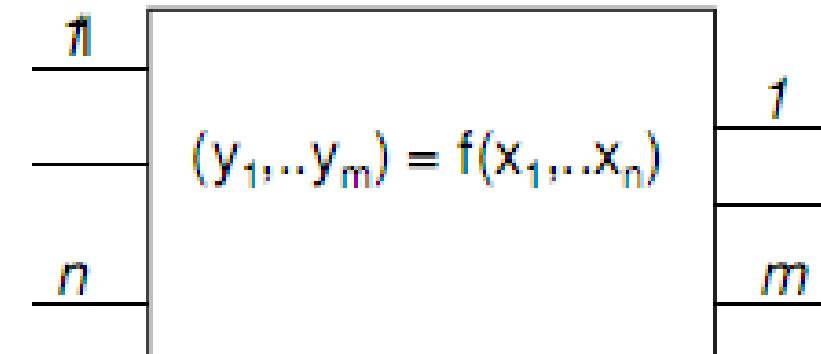
What will you learn?

- Combinational circuit
 - How to design a combinational circuit?
 - Application of combinational circuit
- 
- Basic ALU design
 - Sequential circuit
 - Application of sequential circuit



Combinational circuit

- A combinational circuit is an interconnected set of gates whose *output at any time is a function only of the input at that time*
- Consists of n binary input, m binary output
- Can be defined in three ways:
 - True table
 - Graphic symbol
 - Boolean equation



Implement of Boolean functions

- Sum of Product: The **SOP** form expresses that the output is 1 if **any of the input combinations that produce 1 is true**

$$f = u_1 + u_2 + \dots + u_n \quad \text{With } u_j = x_1 \cdot x_2 \dots x_i$$

- Product of Sum: The **POS** form expresses that the output is 1 if **all the input combinations that produce 1 is true**

$$f = u_1 \cdot u_2 \cdot \dots \cdot u_n \quad \text{With } u_j = x_1 + x_2 + \dots + x_i$$

Sum of Product (SOP)

x	y	z	f
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

$$f = \bar{x}\bar{y}z + \bar{x}y\bar{z} + x\bar{y}z$$

$\bar{x}\bar{y}z$

$\bar{x}y\bar{z}$

$x\bar{y}z$

Product of Sum (POS)

x	y	z	f = \bar{g}	g
0	0	0	1	0
0	0	1	1	0
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	0

$$g = \bar{x} \cdot y \cdot \bar{z} + x \bar{y} \bar{z}$$

$$f = \bar{g} = (x + \bar{y} + z)(\bar{x} + y + z)$$

Simplify the Boolean function

- Using simplification methods:

- Algebraic Simplification
 - Karnaugh map

Simplify the Boolean function

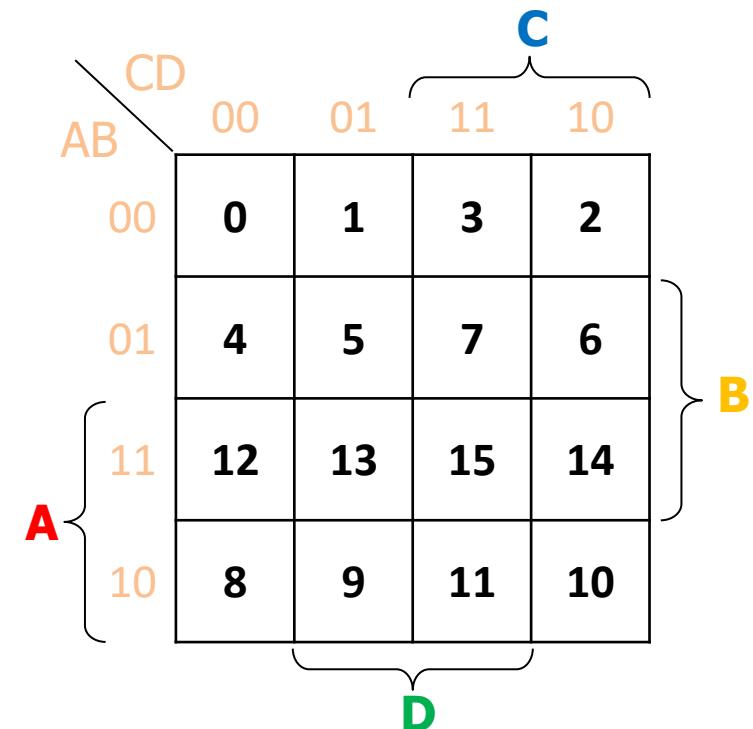
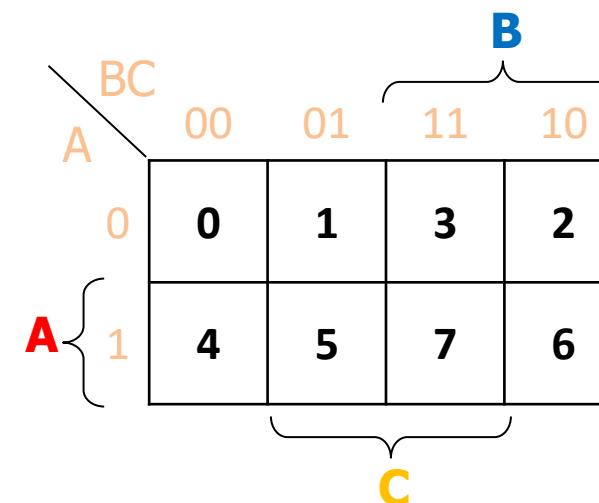
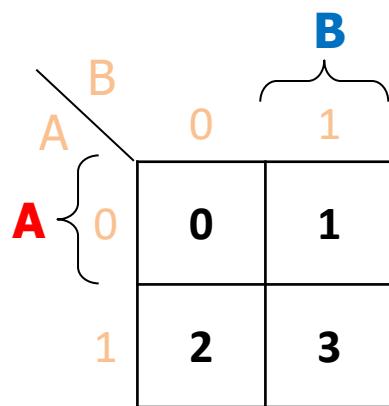
Refer to Basic Identities of Boolean Algebra

Basic Postulates		
$A \cdot B = B \cdot A$	$A + B = B + A$	Commutative Laws
$A \cdot (B + C) = (A \cdot B) + (A \cdot C)$	$A + (B \cdot C) = (A + B) \cdot (A + C)$	Distributive Laws
$1 \cdot A = A$	$0 + A = A$	Identity Elements
$A \cdot \overline{A} = 0$	$A + \overline{A} = 1$	Inverse Elements
Other Identities		
$0 \cdot A = 0$	$1 + A = 1$	
$A \cdot A = A$	$A + A = A$	
$A \cdot (B \cdot C) = (A \cdot B) \cdot C$	$A + (B + C) = (A + B) + C$	Associative Laws
$\overline{A \cdot B} = \overline{A} + \overline{B}$	$\overline{A + B} = \overline{A} \cdot \overline{B}$	DeMorgan's Theorem



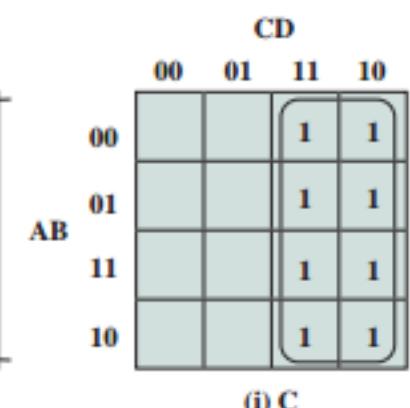
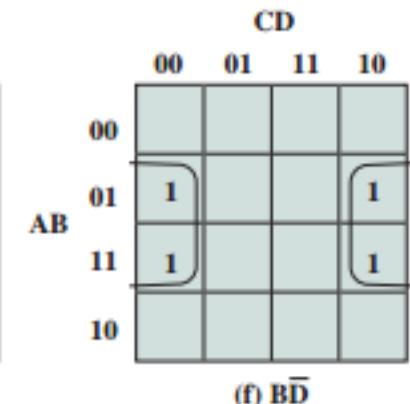
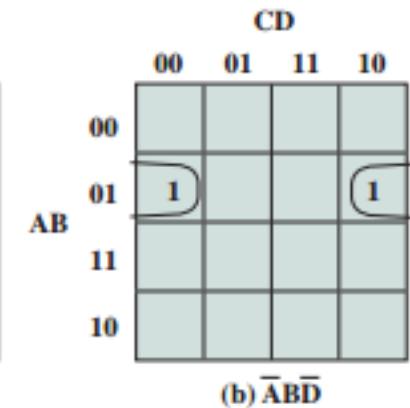
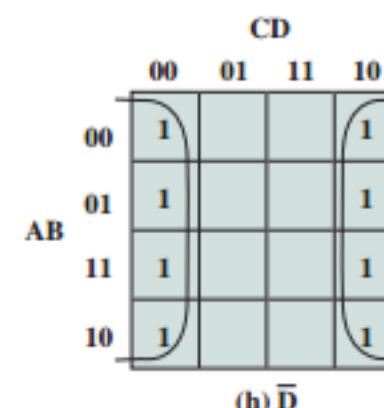
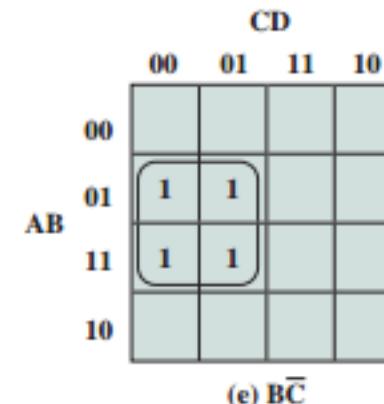
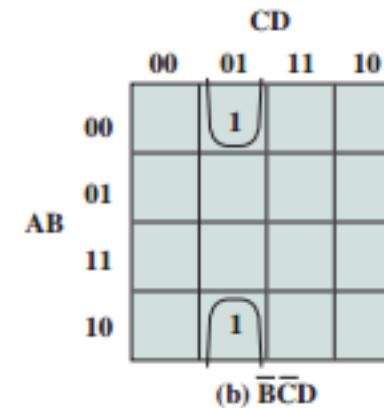
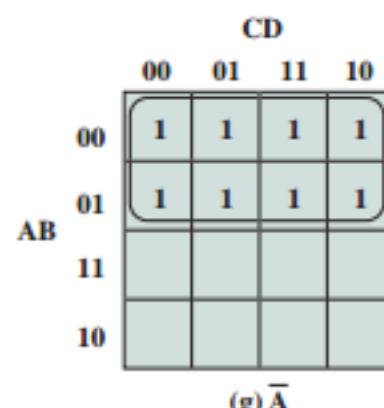
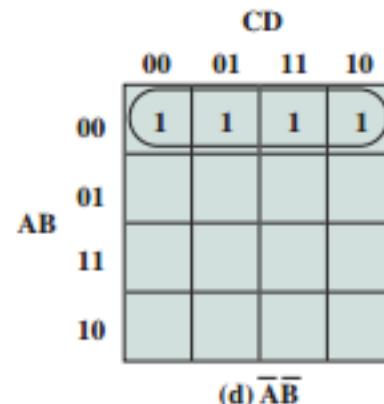
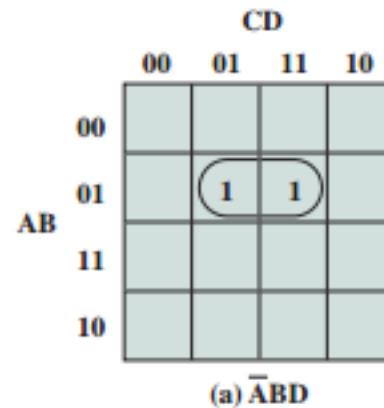
Simplify the Boolean function

□ Basic types of Karnaugh map



Simplify the Boolean function

- Any two squares that are adjacent differ in only one of the variables
- The top square of a column is adjacent to the bottom square, and the leftmost square of a row is adjacent to the rightmost square
- 4 cells located in 4 corners of the map are also considered adjacent cells



Design a combinational circuit

3 steps:

- Step 1: Construct the true table
- Step 2: Identify the Boolean function
- Step 3: Draw the logical circuit and test

Design a combinational circuit

Example:

Design a 3-input, 1-output combination circuit, so that the logic value of the output is the majority of the inputs.

Design a combinational circuit

x	y	z	f
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

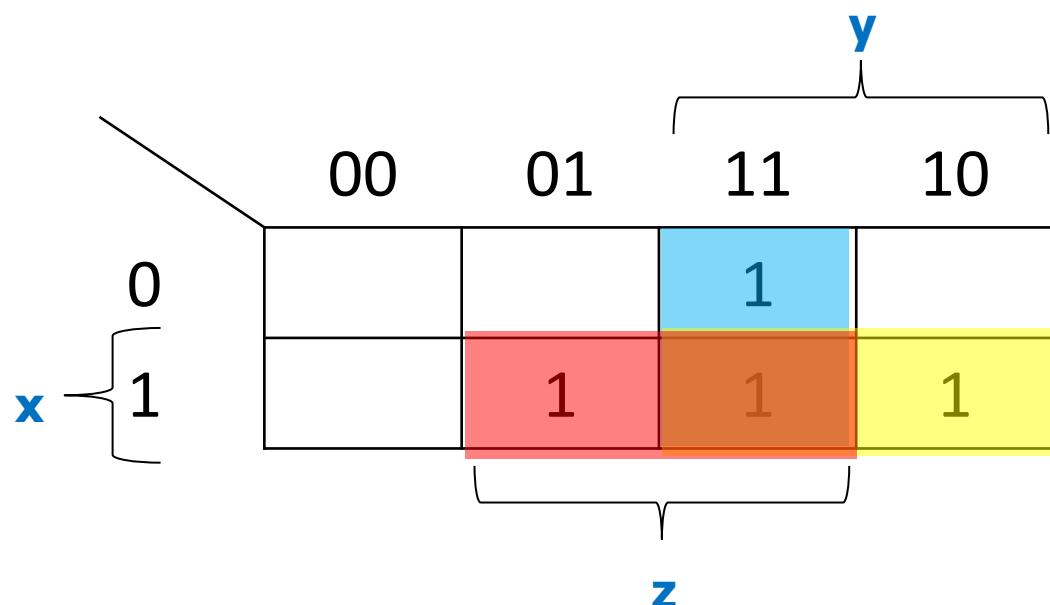
Step 1: The true table

$$f(x, y, z) = \sum(3, 5, 6, 7)$$

Design a combinational circuit

$$f(x, y, z) = \sum(3, 5, 6, 7)$$

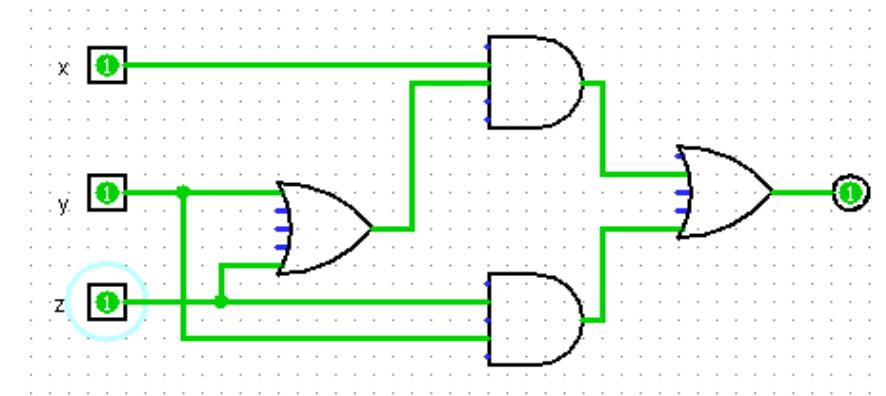
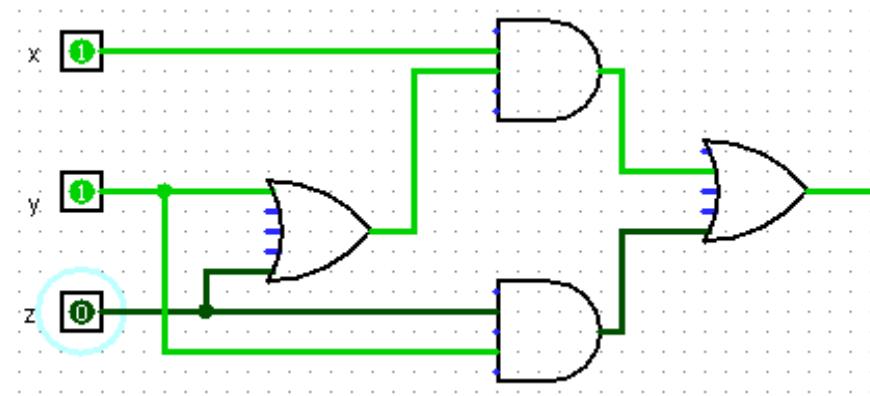
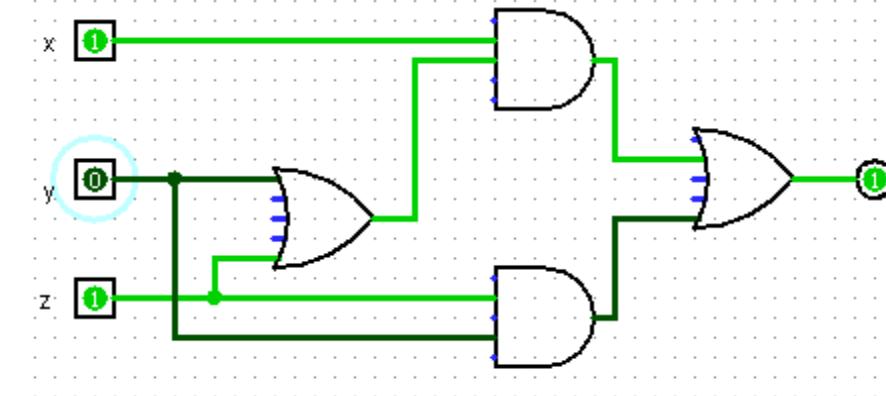
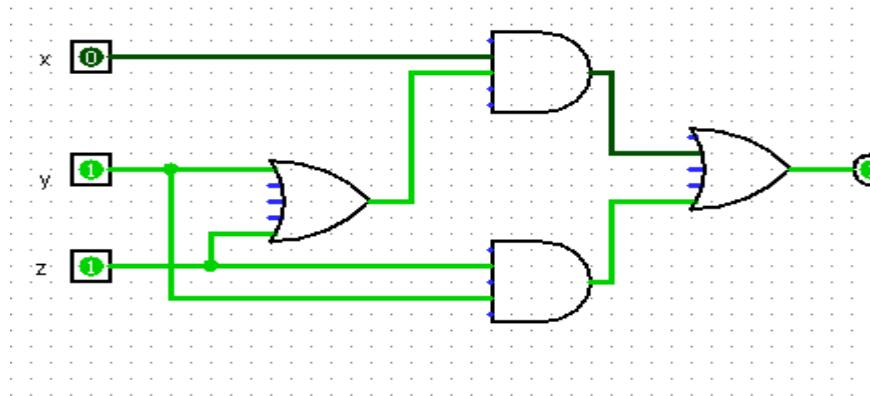
Step 2: Boolean Algebra function



$$f = xz + xy + yz = x \cdot (y + z) + yz$$

Design a combinational circuit

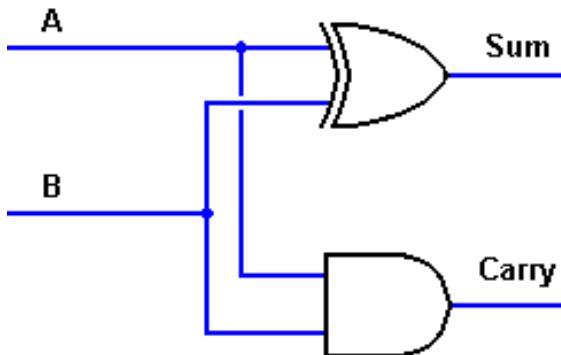
□ Step 3: Drawing the logic circuit and testing



Application of combinational circuit

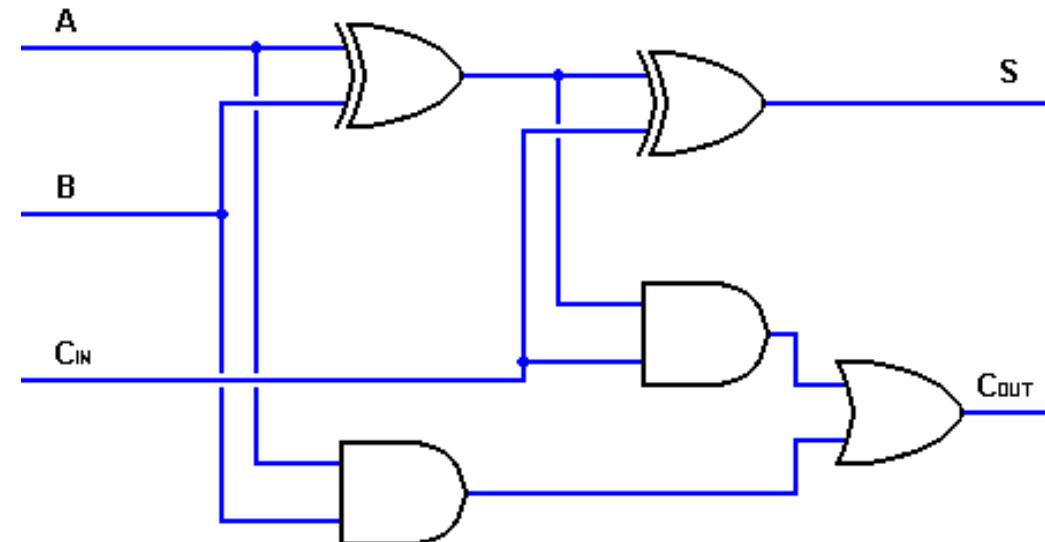
- Adder/ Subtractor
- Encode/ Decode
- Multiplexer/ Demultiplexer
- ALU
- ...

1-bit Adder



$$S = F(A, B) = \Sigma(1, 2)$$

$$C_{\text{carr}} = F(A, B) = \Sigma(0, 3)$$

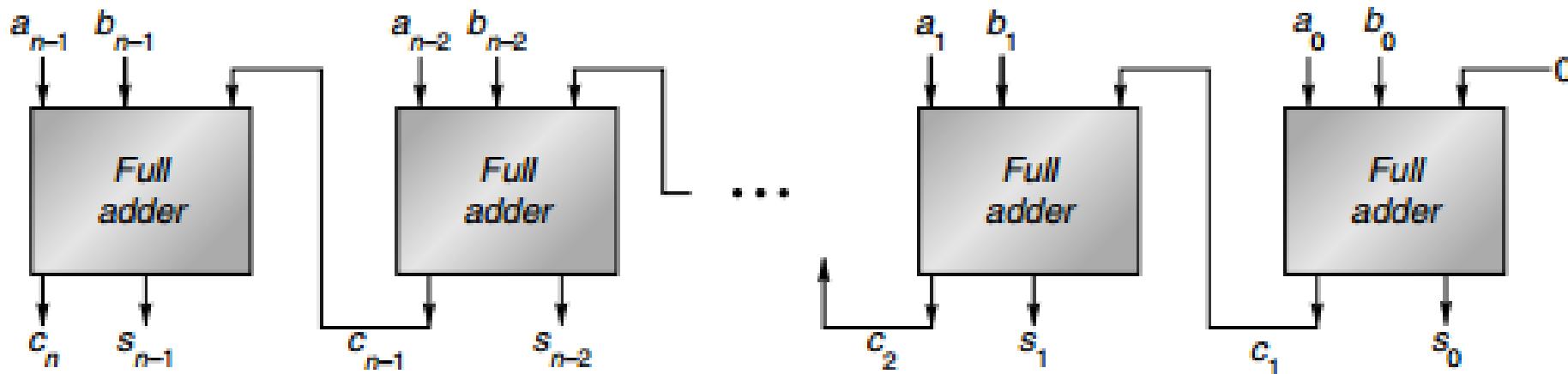


$$S = F(A, B, C_{\text{in}}) = \Sigma(1, 2, 4, 7)$$

$$C_{\text{carr}} = F(A, B, C_{\text{in}}) = \Sigma(3, 5, 6, 7)$$

n-bit Full Adder

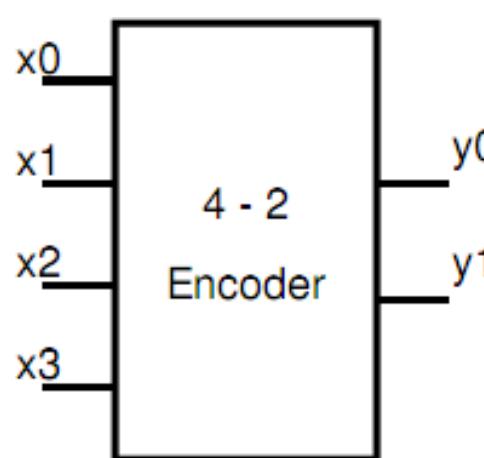
- The carry-out of one full adder is connected to the carry-in of the adder for the next most-significant bit
- The carries ripple from the least-significant bit (on the right) to the most-significant bit (on the left)



The ripple-carry adder, consists of n-bit full adders

Encoder

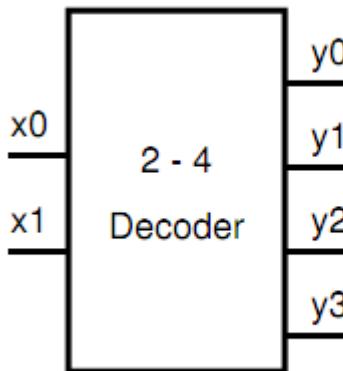
- Consists of 2^n input, n output
- Only one of input has the 1's value at the time
- If the k^{th} input has 1's value, the output will perform a value equal to k



x_0	x_1	x_2	x_3	y_1	y_0
1	0	0	0	0	0
0	1	0	0	0	1
0	0	1	0	1	0
0	0	0	1	1	1

Decoder

- Consists of n input, 2^n output
- Only one of which is asserted at any time
- If the inputs form a binary pattern with the value k, then the output = 1 is the k^{th} output



x_1	x_0	y_0	y_1	y_2	y_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

$$y_0 = \overline{x_1} \cdot \overline{x_0}$$

$$y_1 = \overline{x_1} \cdot x_0$$

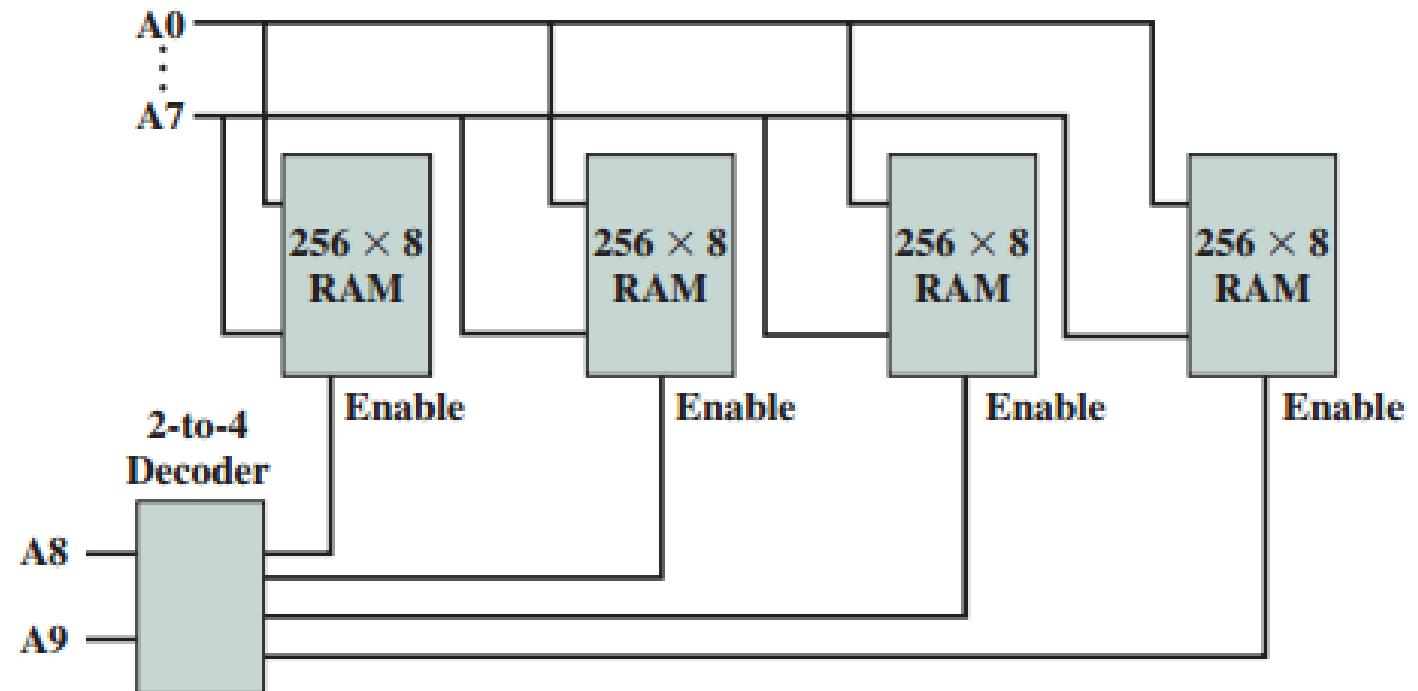
$$y_2 = x_1 \cdot \overline{x_0}$$

$$y_3 = x_1 \cdot x_0$$

Decoder

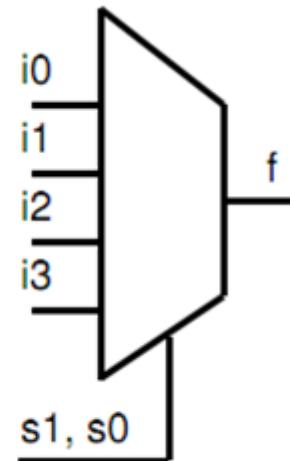
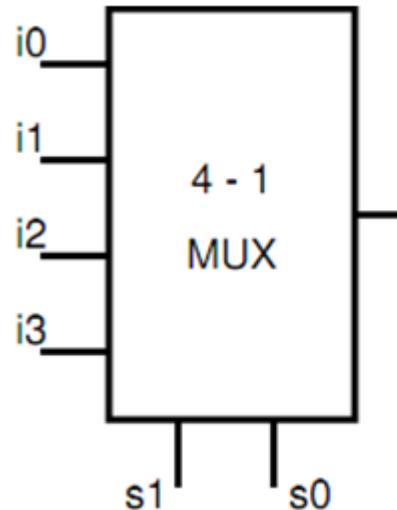
Ex: Construct a 1K-byte memory using four 256 * 8-bit RAM chips. A single unified address space, which can be broken down as follows:

Address	Chip
0000–00FF	0
0100–01FF	1
0200–02FF	2
0300–03FF	3

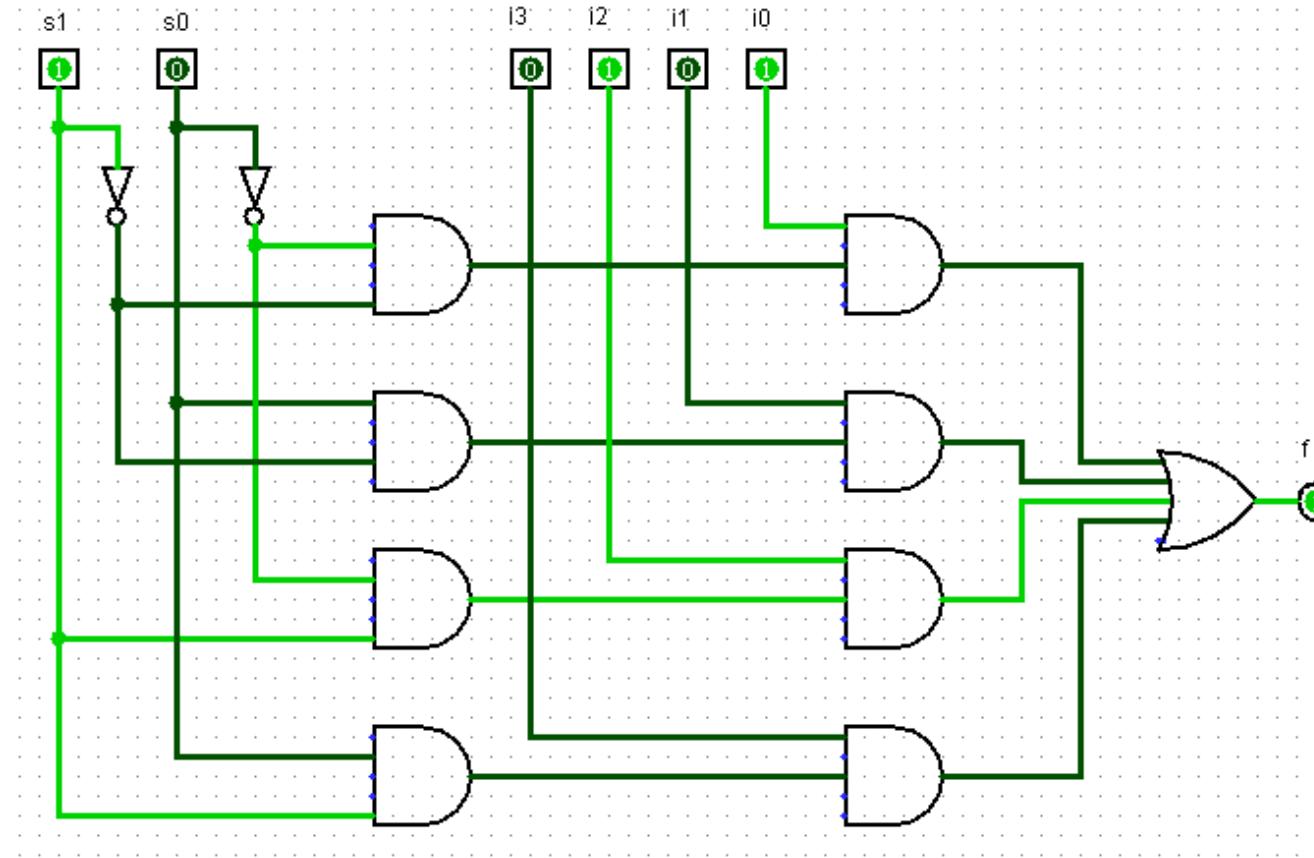


Multiplexer

- Selects one of the 2^{n-1} inputs to be only **one output** based on the 2^n control input
- Ex: MUX 4-1 has 4 inputs, 1 output, and 2 control input



s_1	s_0	f
0	0	i_0
0	1	i_1
1	0	i_2
1	1	i_3

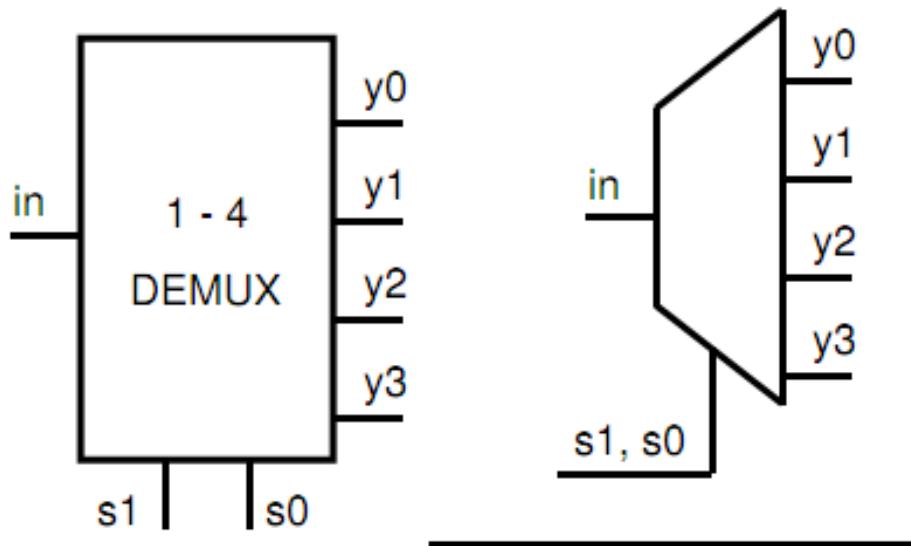


Multiplexer

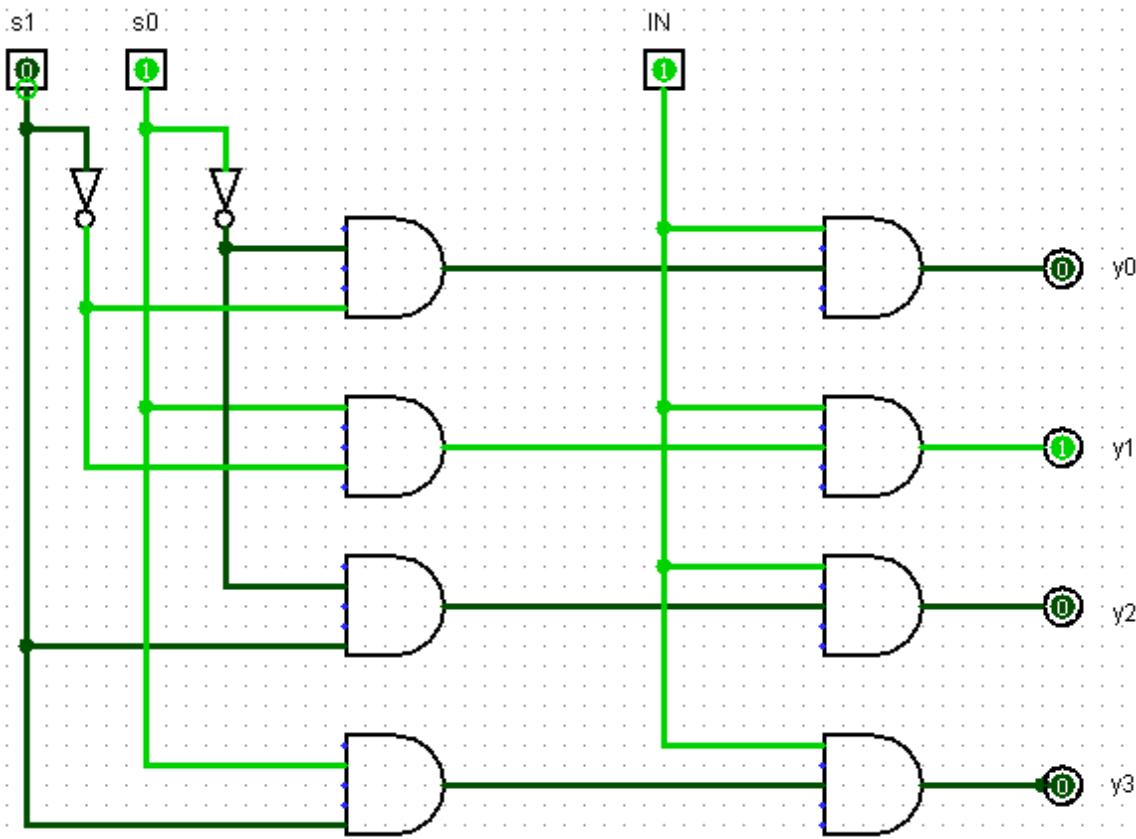
The logic circuit of MUX 4-1

Demultiplexer

- Selects one of the 2^{n-1} output from only **one input** based on the 2^n control input
- Ex: DEMUX 1-4 has 1 input, 4 output and 2 control input



s1	s0	y0	y1	y2	y3
0	0	in	0	0	0
0	1	0	in	0	0
1	0	0	0	in	0
1	1	0	0	0	in

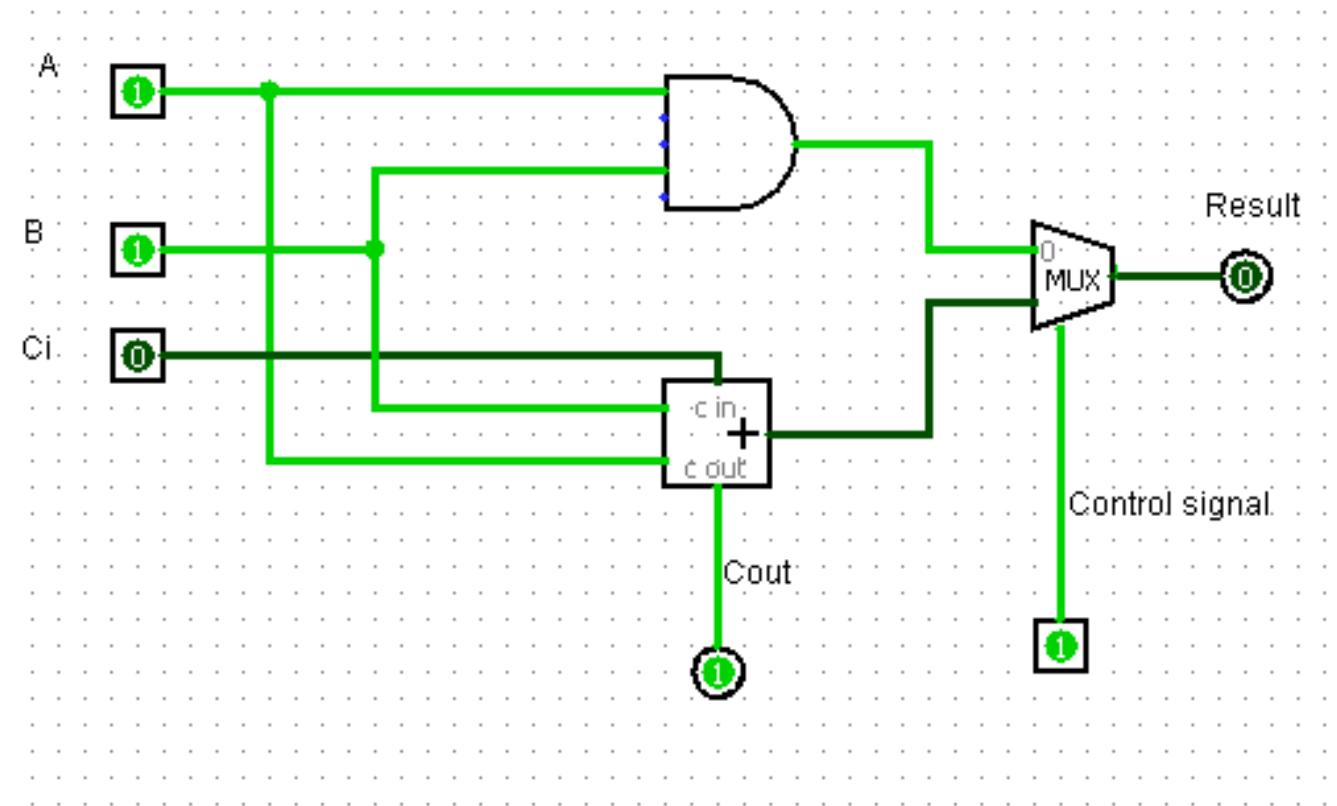


Demultiplexer

The logic circuit of DEMUX 1-4

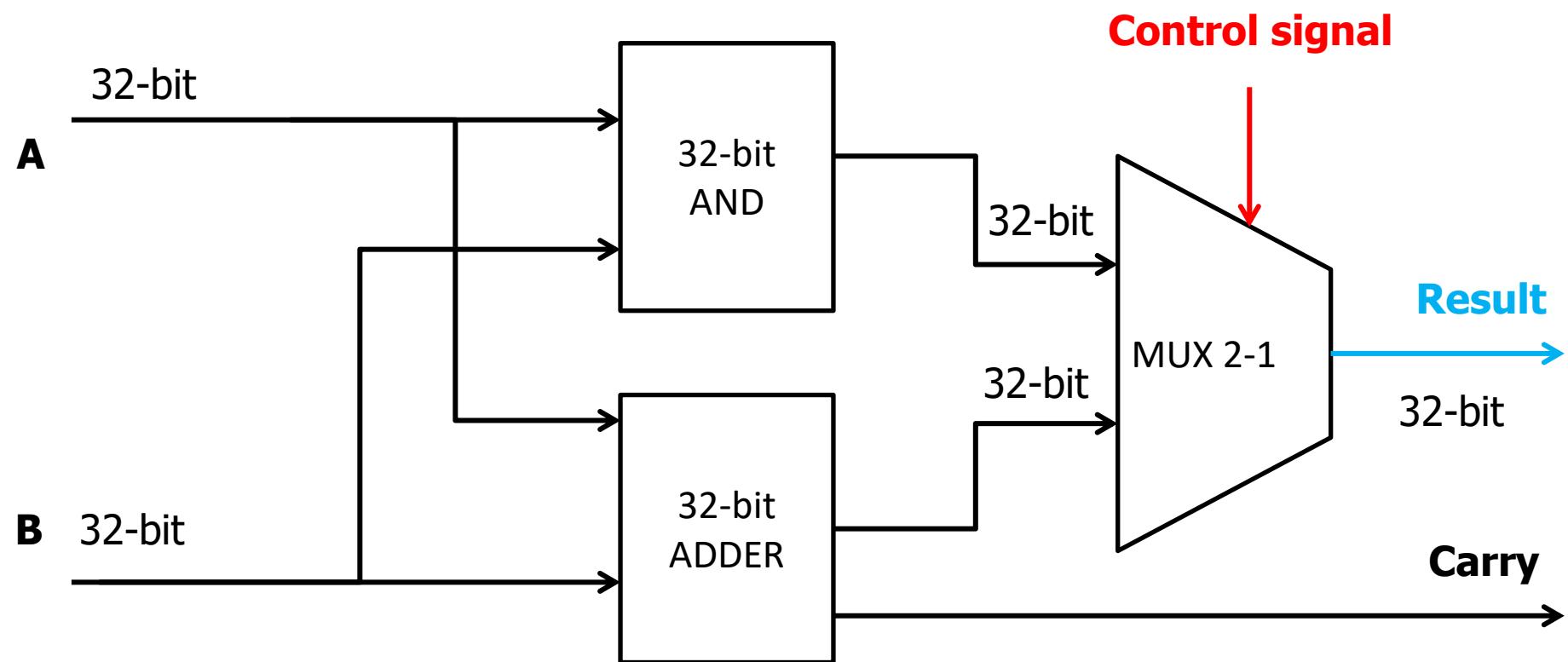
ALU

- The 1-bit ALU with 2 functions: and, add



ALU

- The 32-bits ALU with 2 functions: and, add

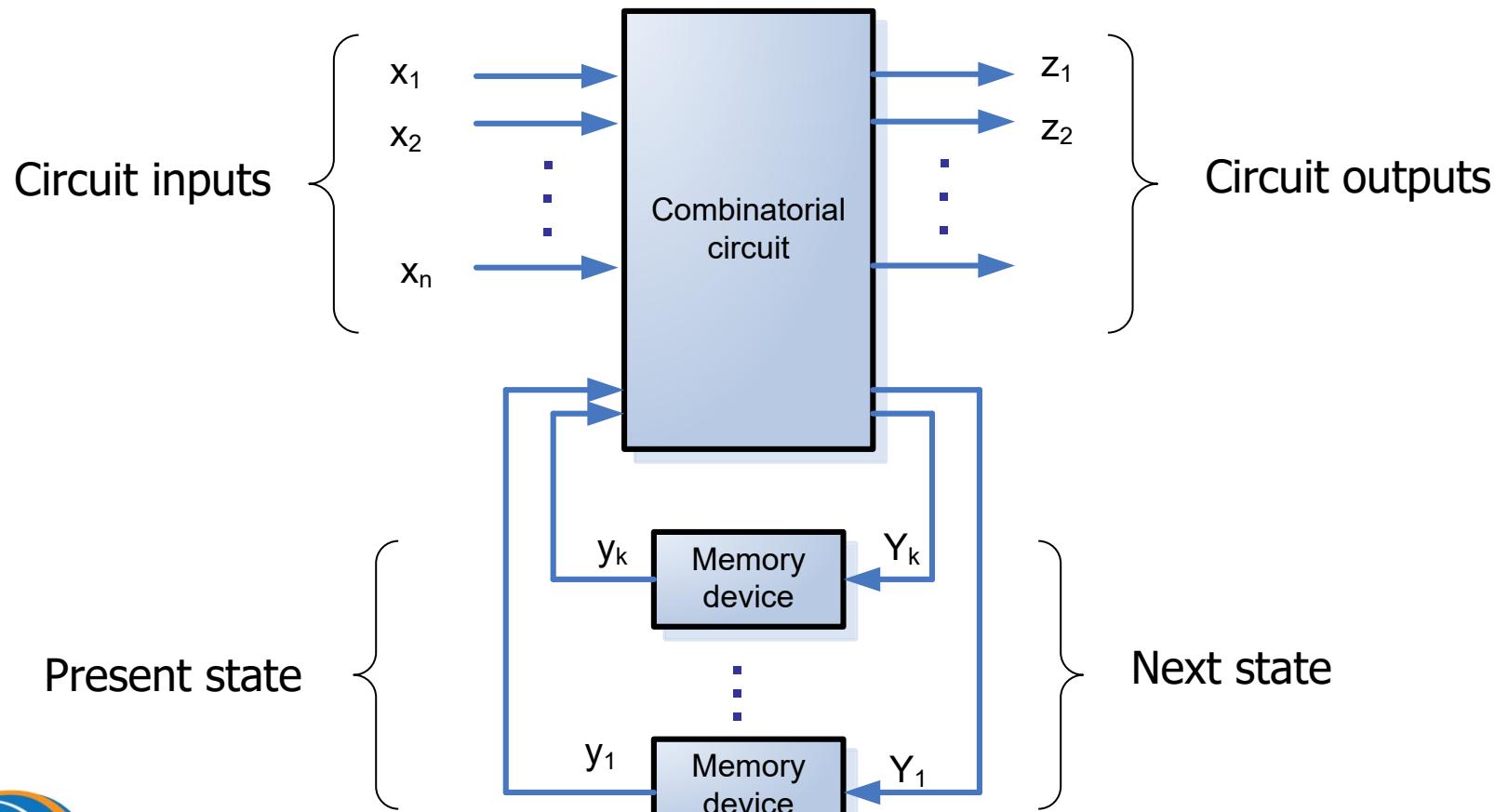


Sequential circuit

- A sequential circuit is an interconnected set of gates and the clock pulse
- The current output of a sequential circuit depends not only on the current input, but also on the history of inputs
- Sequential circuit is capable of "remembering past states"

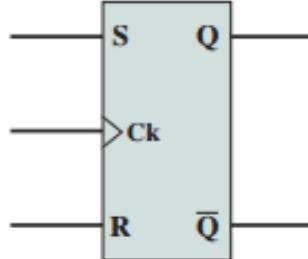
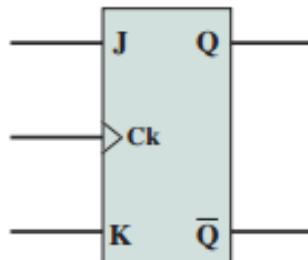
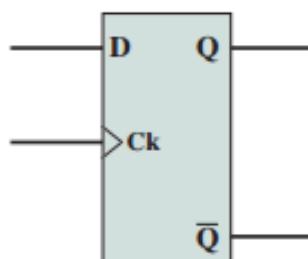


Sequential circuit



Sequential circuit

- Basic sequential circuit is flip-flops
- Some types of flip-flops circuit is shown in the next table

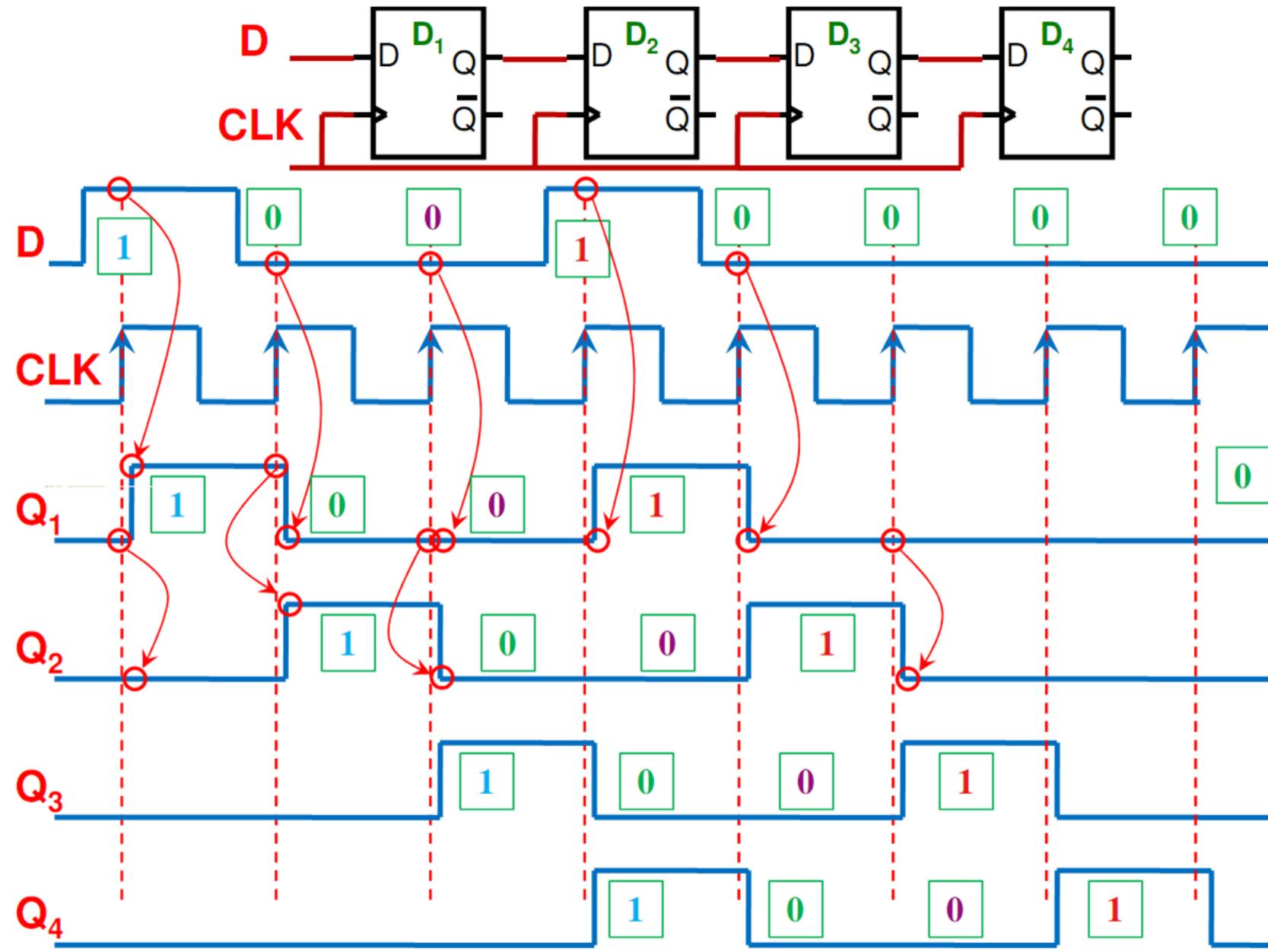
Name	Graphical Symbol	Truth Table															
S-R		<table border="1"><thead><tr><th>S</th><th>R</th><th>Q_{n+1}</th></tr></thead><tbody><tr><td>0</td><td>0</td><td>Q_n</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>-</td></tr></tbody></table>	S	R	Q_{n+1}	0	0	Q_n	0	1	0	1	0	1	1	1	-
S	R	Q_{n+1}															
0	0	Q_n															
0	1	0															
1	0	1															
1	1	-															
J-K		<table border="1"><thead><tr><th>J</th><th>K</th><th>Q_{n+1}</th></tr></thead><tbody><tr><td>0</td><td>0</td><td>Q_n</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>$\overline{Q_n}$</td></tr></tbody></table>	J	K	Q_{n+1}	0	0	Q_n	0	1	0	1	0	1	1	1	$\overline{Q_n}$
J	K	Q_{n+1}															
0	0	Q_n															
0	1	0															
1	0	1															
1	1	$\overline{Q_n}$															
D		<table border="1"><thead><tr><th>D</th><th>Q_{n+1}</th></tr></thead><tbody><tr><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td></tr></tbody></table>	D	Q_{n+1}	0	0	1	1									
D	Q_{n+1}																
0	0																
1	1																

Application of sequential circuit

- Register
- Counter
- ...

Register

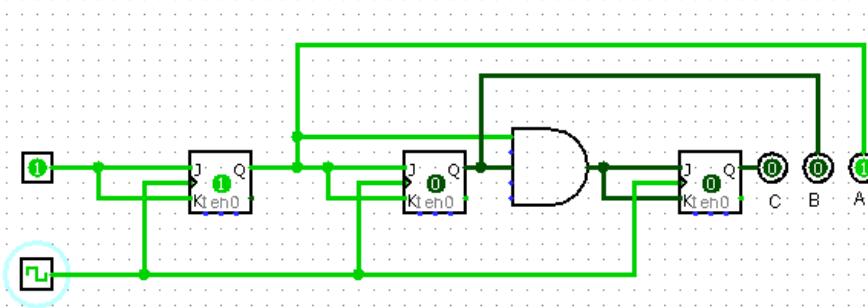
4-bit shift register



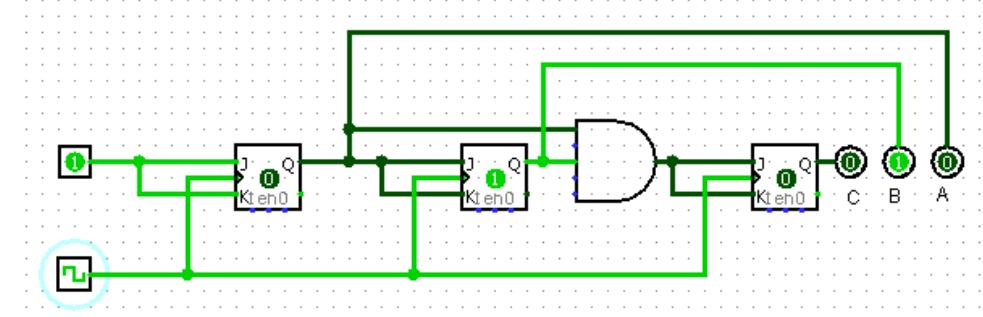
Counter

□ A synchronous counter

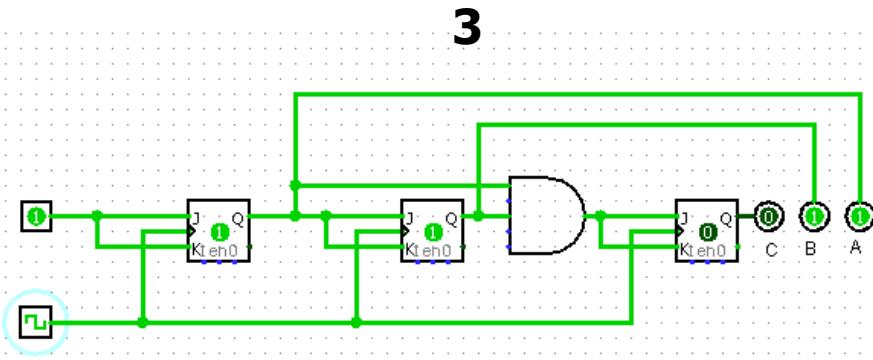
1



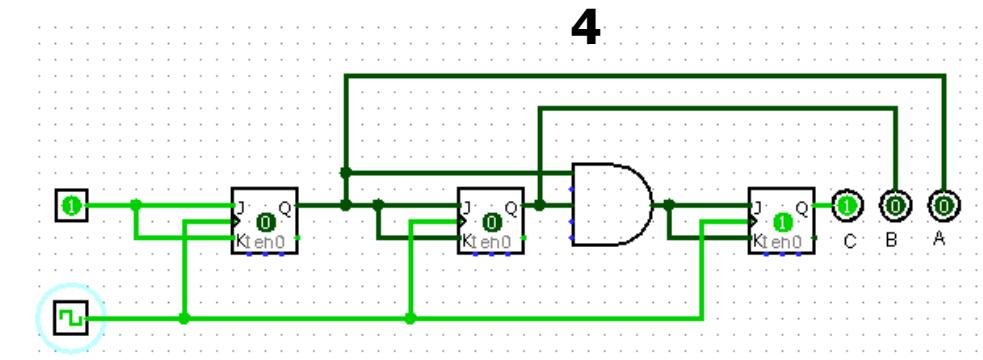
2



3



4



CHAPTER

9

I/O SYSTEM



fit@hcmus

KHOA CÔNG NGHỆ THÔNG TIN
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

What will you learn?

- I/O devices
 - I/O System Characteristics
 - I/O Modules
 - I/O Register Mapping
 - I/O Data transfer
-
- I/O Command
 - Life cycle of an I/O request
 - I/O Bus
 - Typical x86 PC I/O System

I/O devices

- Can be typify by:
 - Behavior: input, output, storage
 - Partner: human / machine
 - Data rate: bytes/sec, transfer/sec

□ Character & Block devices

The device interface gives the illusion that devices support the same API – character stream and block access

application/user:	<i>read character from device</i>	naming, protection, read, write
operating system:	<i>character & block API</i>	hardware specific
hardware:	<i>keyboard, mouse, etc.</i>	PIO, interrupt handling, or DMA

I/O Modules

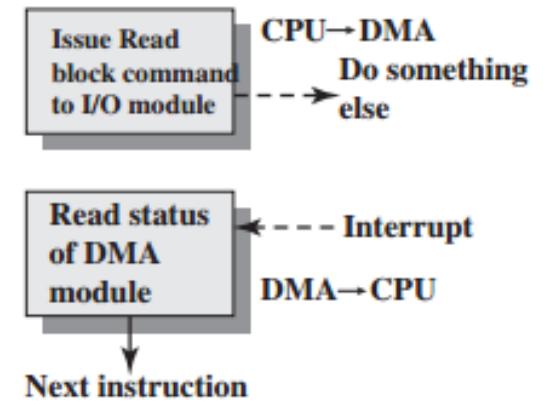
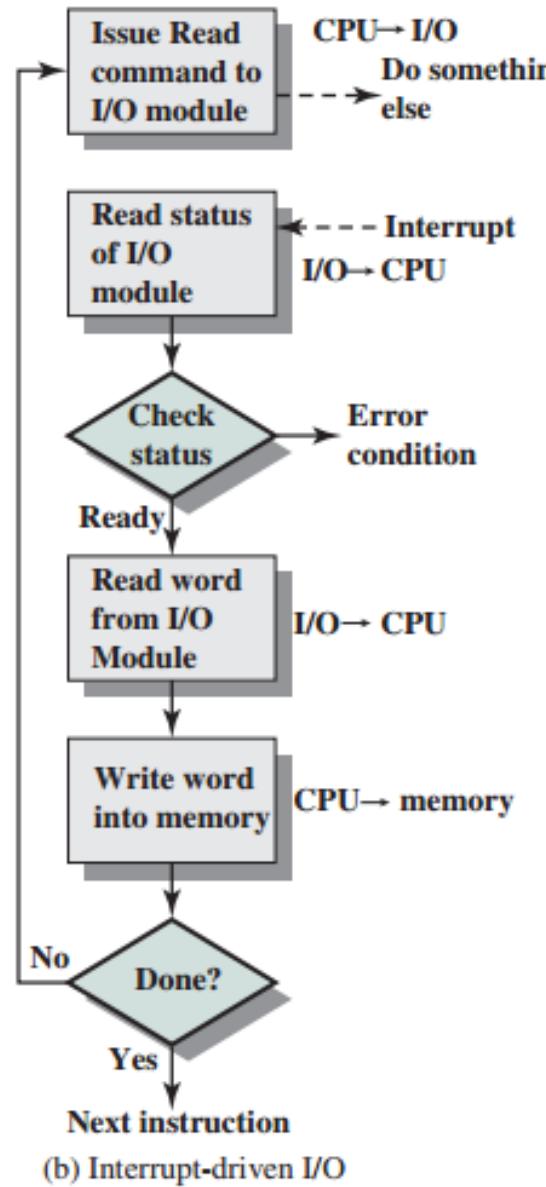
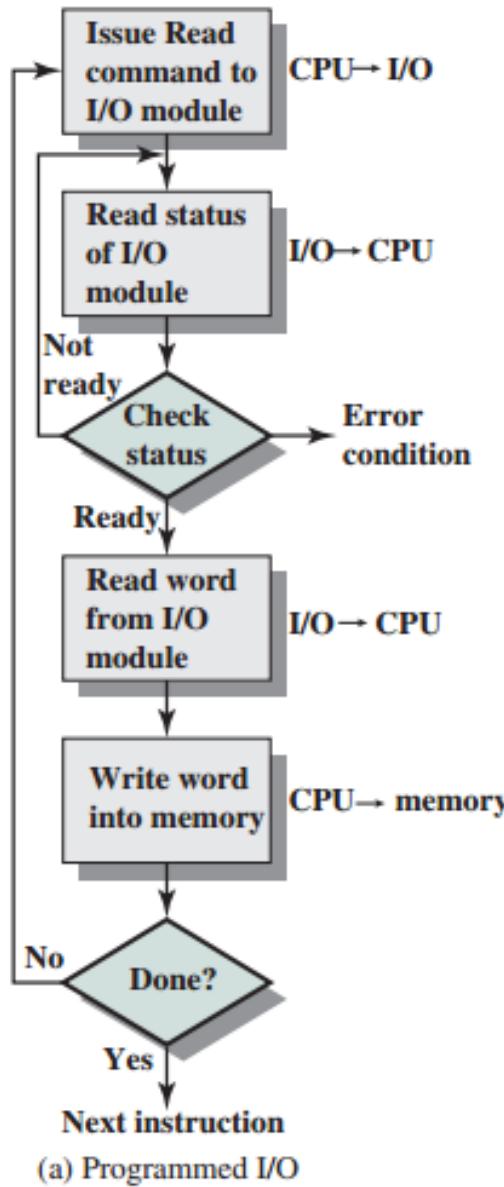
- Interface to the processor and memory via the system bus or control switch
- Interface to one or more peripheral devices

I/O Register Mapping

- Memory-mapped I/O
 - Registers are addressed in same space as memory
 - Address decoder distinguishes between them
 - OS uses address translation mechanism to make them only accessible to kernel
- Isolated I/O
 - Separate instructions to access I/O registers
 - Can only be executed in kernel mode

I/O Data Transfer

- Data transfer between CPU and I/O devices can be handled in generally three types of modes:
 - Programed I/O
 - Interrupt Driven I/O
 - Direct Memory Access

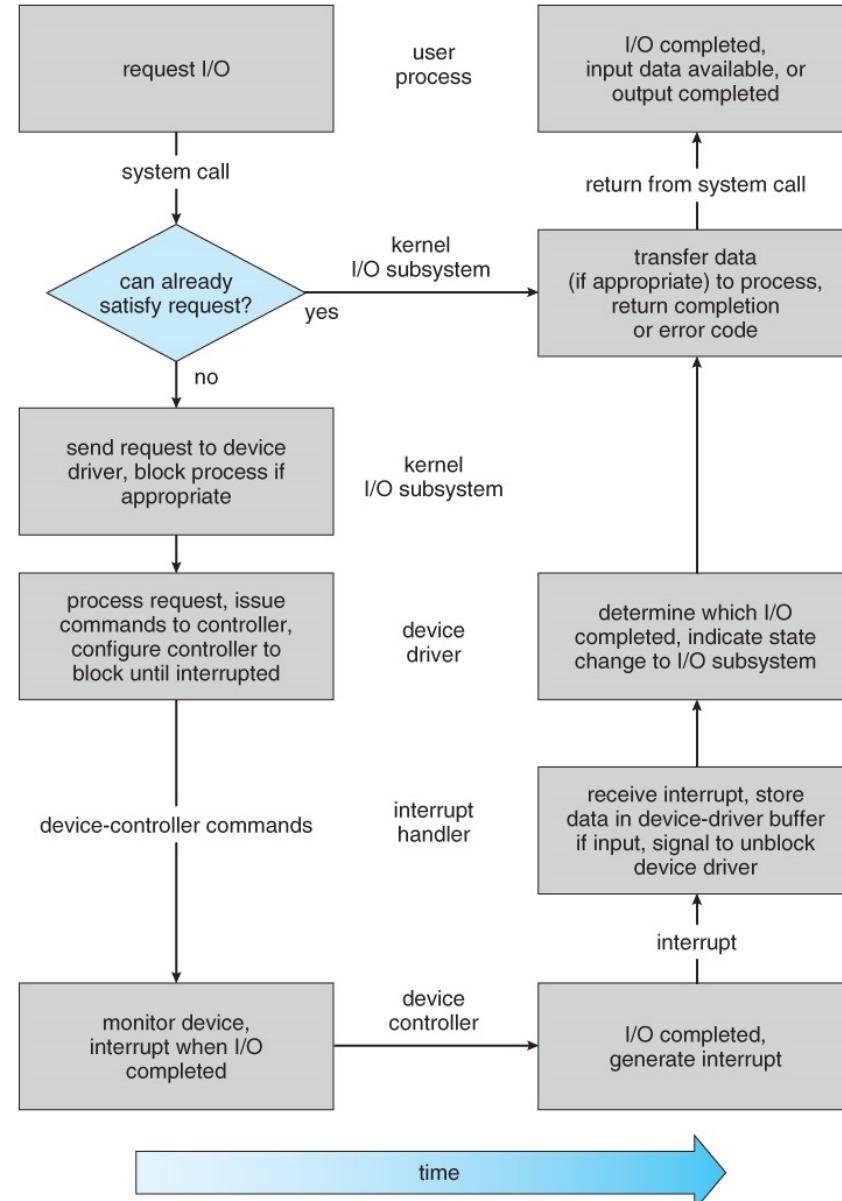


I/O Commands

- I/O devices are managed by I/O Controller hardware
- The processor issues an address, specifying I/O module and device, and an I/O command. The commands are:
 - Control
 - Test
 - Read
 - Write

Life cycle of an I/O request

- ❑ Users request data using file names, which must ultimately be mapped to specific blocks of data from a specific device managed by a specific device driver.
- ❑ UNIX uses a *mount table* to map filename prefixes (e.g. /usr) to specific mounted devices
- ❑ UNIX uses special *device files*, usually located in /dev, to represent and access physical devices directly



I/O Bus Types

- Processor-Memory buses
 - Short, high speed
 - Design is matched to memory organization
- I/O buses
 - Longer, allowing multiple connections
 - Connect to processor-memory bus through a bridge

I/O Bus Example

	Firewire	USB 2.0	PCI Express	Serial ATA	Serial Attached SCSI
Intended use	External	External	Internal	Internal	External
Devices per channel	63	127	1	1	4
Data width	4	2	2/lane	4	4
Peak bandwidth	50MB/s or 100MB/s	0.2MB/s, 1.5MB/s, or 60MB/s	250MB/s/lane 1x, 2x, 4x, 8x, 16x, 32x	300MB/s	300MB/s
Hot pluggable	Yes	Yes	Depends	Yes	Yes
Max length	4.5m	5m	0.5m	1m	8m
Standard	IEEE 1394	USB Implementers Forum	PCI-SIG	SATA-IO	INCITS TC T10

Typical X86 PC I/O System

