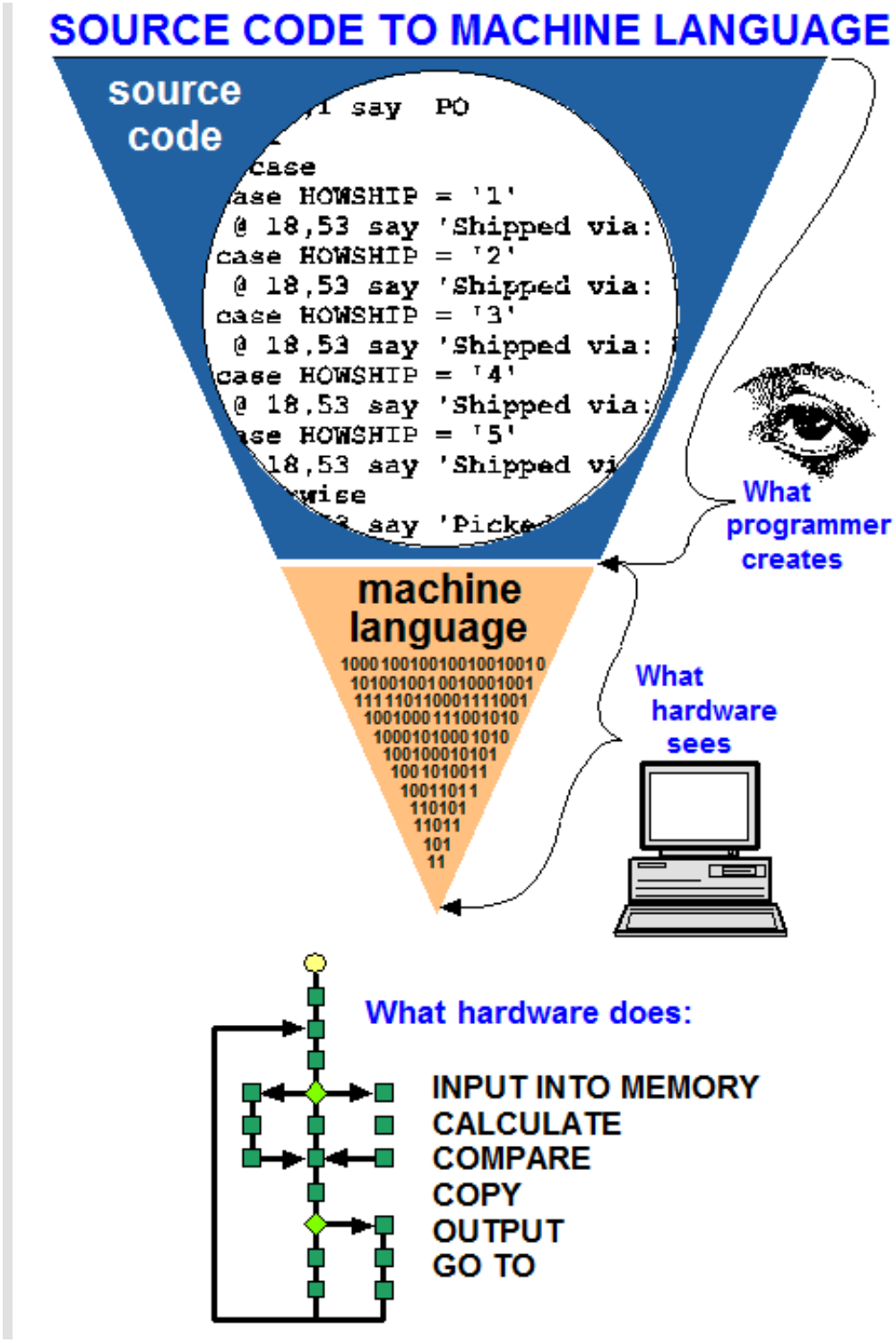
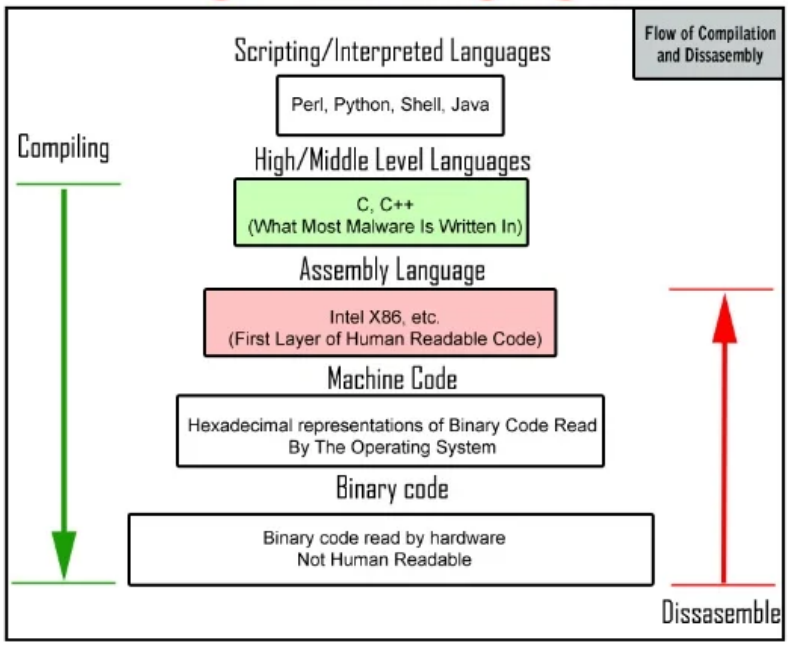
# Computer Organization and Architecture

## Programming Language



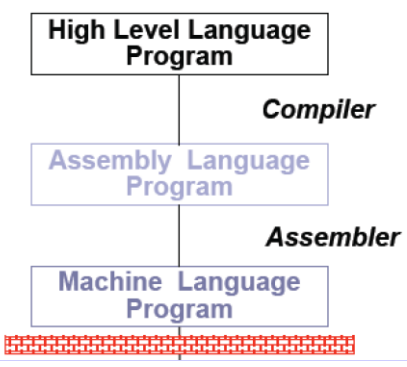



## High Vs. Low Level Languages

| | | |
|---|---|---|
| Human languages | E.g., English, French, Spanish, Chinese, German, Arabic etc. | High level |
| High level programming language | E.g., Python, Java, C++<br>for (i = 1; i <= 10; i++) | |
| Low level programming language | Assembly<br>MOV #10, R0 | |
| Machine language | Binary<br>10100000 1010 00 | |
|  | Computer hardware | Low level |

James Tam

## Levels of Representation



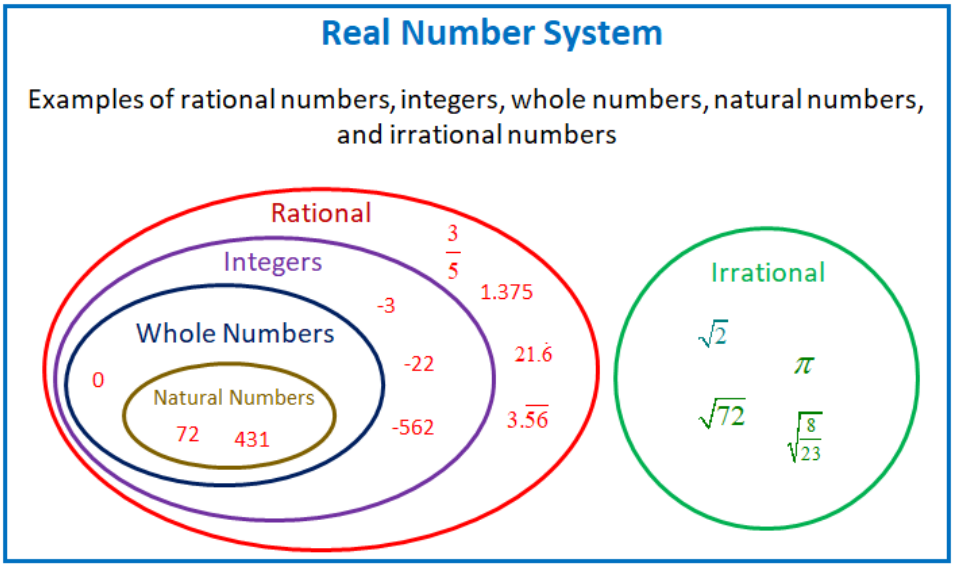

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

- mov BL, 02h
- mov AX,0A56h
- mul BL

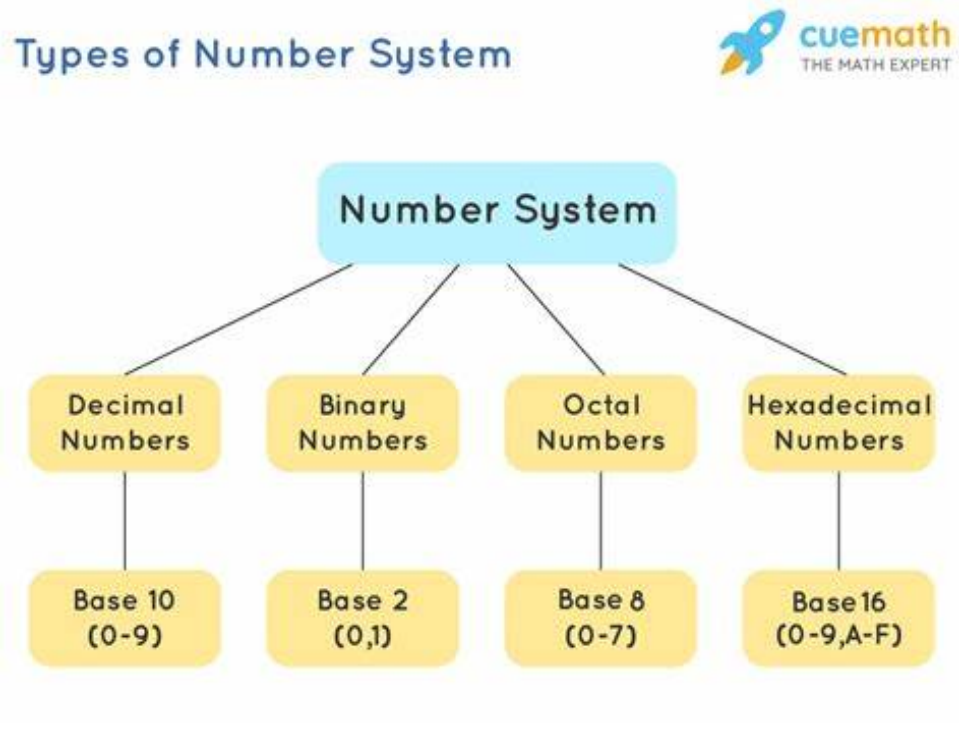
```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

# Number System

## Real Number System




## Number System Conversion



# Number Systems Conversion Chart

Binary

| Place | $2^{11}$ | $2^{10}$ | $2^{9}$ | $2^{8}$ | $2^{7}$ | $2^{6}$ | $2^{5}$ | $2^{4}$ | $2^{3}$ | $2^{2}$ | $2^{1}$ | $2^{0}$ | $2^{-1}$ | $2^{-2}$ | $2^{-3}$ | $2^{-4}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Weight | 2048 | 1024 | 512 | 256 | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 | 0.5 | 0.25 | 0.125 | 0.0625 |

## Binary, Hex, and Octal Conversions

| Binary | Octal | Hexadecimal | Decimal |
|---|---|---|---|
| 0000 | 0 | 0 | 0 |
| 0001 | 1 | 1 | 1 |
| 0010 | 2 | 2 | 2 |
| 0011 | 3 | 3 | 3 |
| 0100 | 4 | 4 | 4 |
| 0101 | 5 | 5 | 5 |
| 0110 | 6 | 6 | 6 |
| 0111 | 7 | 7 | 7 |
| 1000 | 10 | 8 | 8 |
| 1001 | 11 | 9 | 9 |
| 1010 | 12 | A | 10 |
| 1011 | 13 | B | 11 |
| 1100 | 14 | C | 12 |
| 1101 | 15 | D | 13 |
| 1110 | 16 | E | 14 |
| 1111 | 17 | F | 15 |

## Methodology

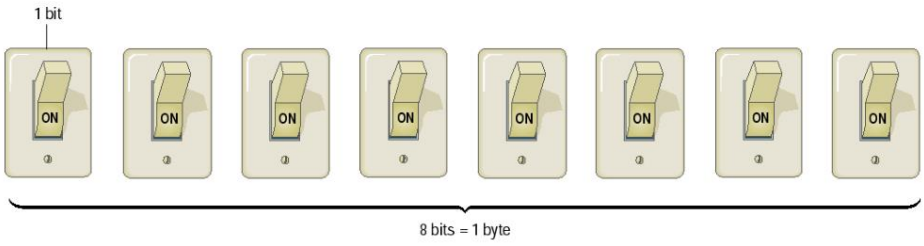
1. Convert from decimal to binary
   Divide the decimal by the largest binary weight it is divisible by and place a "1" in that column. Then select the next largest weight, if it is divisible put a "1" in that column otherwise place a "0" in the column. Continue until all the columns have either a "1" or "0" resulting in a binary expression.
2. Convert from decimal to hex.
   Convert to binary first, then group the binary in groups of 4 beginning on the right working to the left. For each group determine the hex value based on the table to the left.
3. Convert to octal
   Convert to binary first, then group the binary in groups of 3 beginning on the right working to the left. For each group determine the octal value based on the table to the left.

# Binary Number System

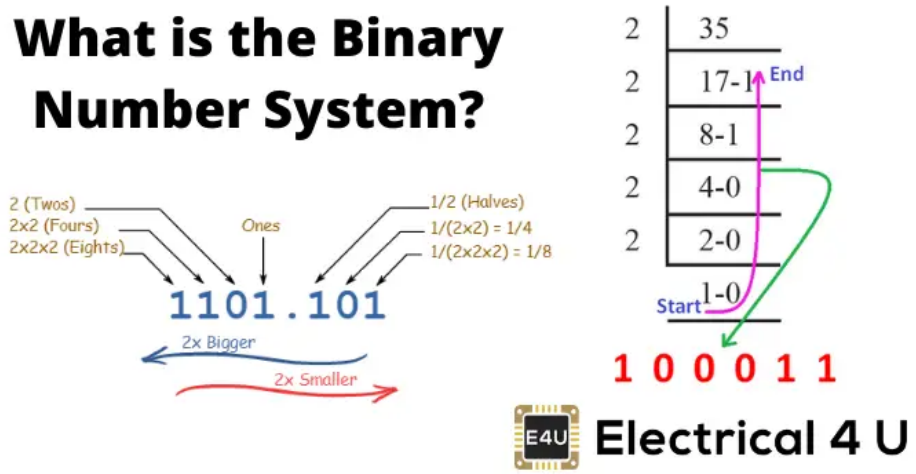
## Number System

Bits and Bytes

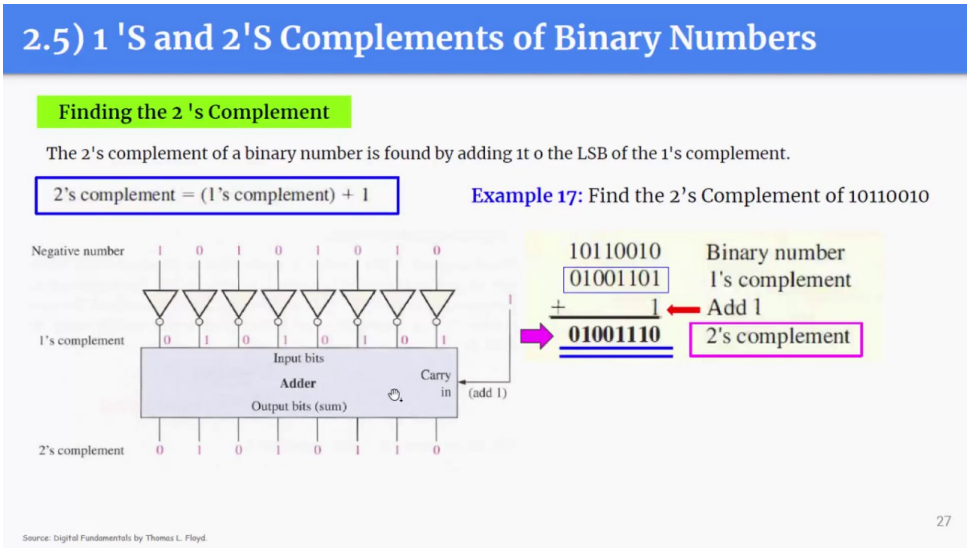
- A single unit of data is called a bit, having a value of 1 or 0.
- Computers work with collections of bits, grouping them to represent larger pieces of data, such as letters of the alphabet.
- Eight bits make up one byte. A byte is the amount of memory needed to store one alphanumeric character.
- With one byte, the computer can represent one of 256 different symbols or characters.

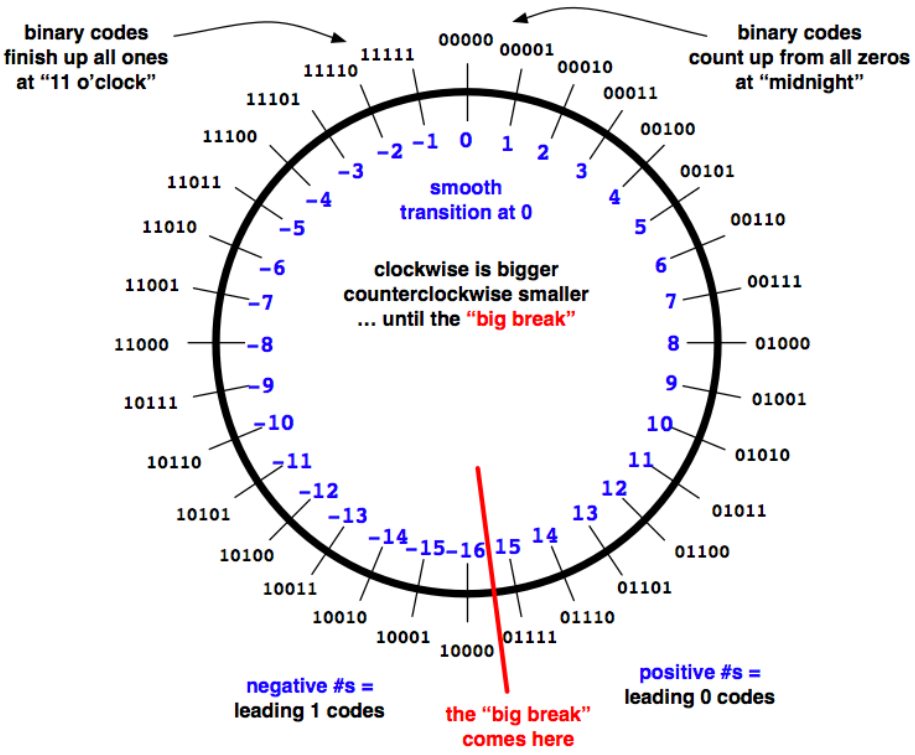



1 1 1 1 1 1 1 1


What is the Binary Number System?
2 (Twos)
2x2 (Fours)
2x2x2 (Eights)
Ones
1/2 (Halves)
1/(2x2) = 1/4
1/(2x2x2) = 1/8
1101.101
2x Bigger
2x Smaller
2 35
2 17-1
2 8-1
2 4-0
2 2-0
2 1-0
End
Start
1 0 0 0 1 1
E4U
Electrical 4 U

# Complement


2.5) 1 'S and 2'S Complements of Binary Numbers
Finding the 2 's Complement
The 2's complement of a binary number is found by adding 1t o the LSB of the 1's complement.
2's complement = (1's complement) + 1
Example 17: Find the 2's Complement of 10110010
Negative number
1's complement
2's complement
Input bits
Adder
Output bits (sum)
Carry in
(add 1)
10110010 Binary number
01001101 1's complement
+ 1 Add 1
01001110 2's complement
Source: Digital Fundamentals by Thomas L. Floyd.
27


binary codes finish up all ones at "11 o'clock"
binary codes count up from all zeros at "midnight"
smooth transition at 0
clockwise is bigger
counterclockwise smaller
... until the "big break"
negative #s = leading 1 codes
positive #s = leading 0 codes
the "big break" comes here

# Complements of numbers

## (r-1 )'s Complement

- Given a number $N$ in base $r$ having $n$ digits,
- the (r- 1)'s complement of $N$ is defined as

$$(r^n - 1) - N$$

- For decimal numbers the base or $r = 10$ and $r- 1= 9$,
- so the 9's complement of $N$ is $(10^n-1)-N$
- $99999\ldots\ldots - N$

| 9 | 9 | 9 | 9 | 9 |
|---|---|---|---|---|
| Digit n | Digit n-1 | Next digit | Next digit | First digit |

-

## Complements

♣ There are two types of complements for each base-$r$ system: the radix complement and diminished radix complement.

⟹ the $r$'s complement and the second as the $(r-1)$'s complement.

**Diminished Radix Complement**

Given a number $N$ in case $r$ having $n$ digits, the $(r-1)$ 's complement of $N$ is defined as $(r^n - 1) - N$. For decimal numbers, $r = 10$ and $r - 1 = 9$, so the 9's complement of $N$ is $(10^n - 1) - N$.

**Example:**

The 9's complement of 546700 is $999999 - 546700 = 453299$.
The 9's complement of 012398 is $999999 - 012398 = 987601$.

♣ For binary numbers, $r = 2$ and $r - 1 = 1$, so the 1's complement of $N$ is $(2^n - 1) - N$.

**Example:**

The 1's complement of 1011000 is 0100111
The 1's complement of 0101101 is 1010010

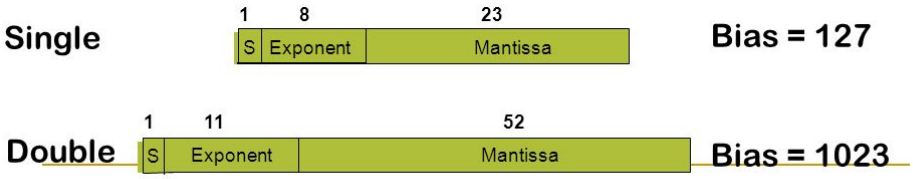
## Precision floating-point

# Floating point numbers– the IEEE standard

- IEEE Standard 754
  - Most (but not all) computer manufactures use IEEE-754 format
  - Number represented:

$$(-1)^S * (1.M)*2^{(E - Bias)}$$

2 main formats: single and double




# Floating-Point Numbers

- Examples of floating-point numbers in base 10 …
  - $5.341\times10^{3}$ , $0.05341\times10^{5}$ , $-2.013\times10^{-1}$ , $-201.3\times10^{-3}$ (decimal point)
- Examples of floating-point numbers in base 2 …
  - $1.00101\times2^{23}$ , $0.0100101\times2^{25}$ , $-1.101101\times2^{-3}$ , $-1101.101\times2^{-6}$ (binary point)
  - Exponents are kept in decimal for clarity
  - The binary number $(1101.101)_2 = 2^3+2^2+2^0+2^{-1}+2^{-3} = 13.625$
- Floating-point numbers should be normalized
  - Exactly one non-zero digit should appear before the point
    - In a decimal number, this digit can be from 1 to 9
    - In a binary number, this digit should be 1
  - Normalized FP Numbers: $5.341\times10^{3}$ and $-1.101101\times2^{-3}$
  - NOT Normalized: $0.05341\times10^{5}$ and $-1101.101\times2^{-6}$

*Floating Point* *ICS 233 – KFUPM* *© Muhamed Mudawar slide 4*

## Normalized Floating Point Numbers

- For a normalized floating point number ($S$, $E$, $F$)


- Significand is equal to $(1.F)_2 = (1.f_1f_2f_3f_4\ldots)_2$
  - IEEE 754 assumes hidden 1. (not stored) for normalized numbers
  - Significand is 1 bit longer than fraction
- Value of a Normalized Floating Point Number is

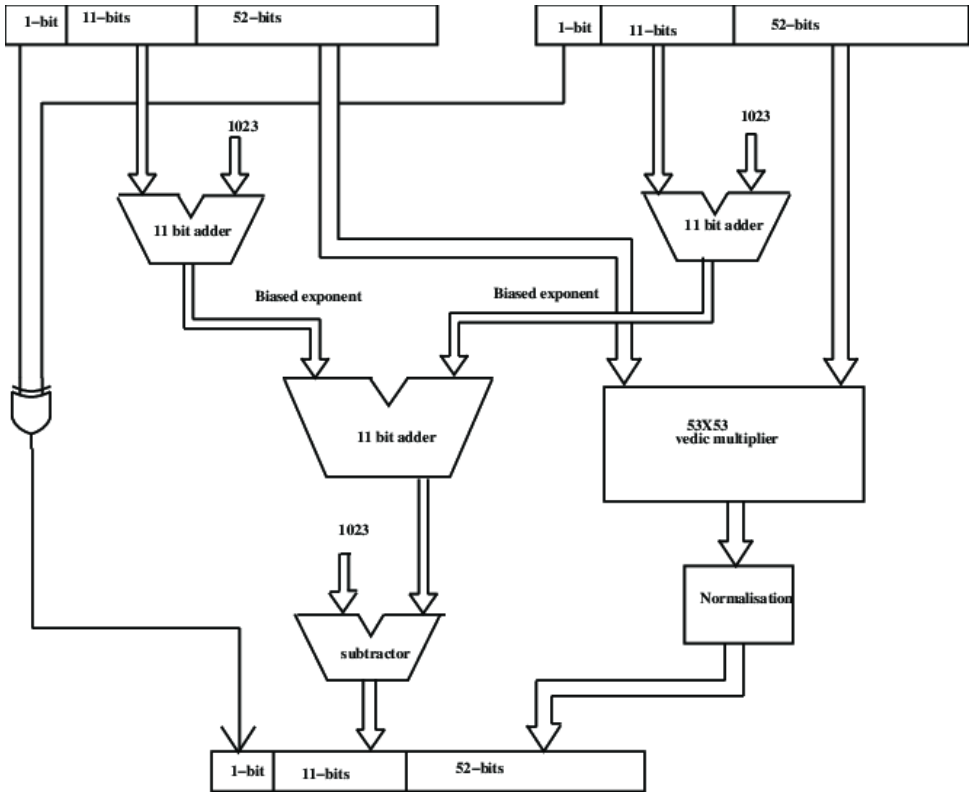
$$(-1)^S \times (1.F)_2 \times 2^{val(E)}$$

$$(-1)^S \times (1.f_1f_2f_3f_4\ldots)_2 \times 2^{val(E)}$$

$$(-1)^S \times (1 + f_1\times2^{-1} + f_2\times2^{-2} + f_3\times2^{-3} + f_4\times2^{-4}\ldots)_2 \times 2^{val(E)}$$

$(-1)^S$ is 1 when $S$ is 0 (positive), and –1 when $S$ is 1 (negative)

*Floating Point* *ICS 233 – KFUPM* *© Muhamed Mudawar slide 8*




## ASCII

### ASCII control characters

| | | |
|---|---|---|
| 00 | NULL | (Null character) |
| 01 | SOH | (Start of Header) |
| 02 | STX | (Start of Text) |
| 03 | ETX | (End of Text) |
| 04 | EOT | (End of Trans.) |
| 05 | ENQ | (Enquiry) |
| 06 | ACK | (Acknowledgement) |
| 07 | BEL | (Bell) |
| 08 | BS | (Backspace) |
| 09 | HT | (Horizontal Tab) |
| 10 | LF | (Line feed) |
| 11 | VT | (Vertical Tab) |
| 12 | FF | (Form feed) |
| 13 | CR | (Carriage return) |
| 14 | SO | (Shift Out) |
| 15 | SI | (Shift In) |
| 16 | DLE | (Data link escape) |
| 17 | DC1 | (Device control 1) |
| 18 | DC2 | (Device control 2) |
| 19 | DC3 | (Device control 3) |
| 20 | DC4 | (Device control 4) |
| 21 | NAK | (Negative acknowl.) |
| 22 | SYN | (Synchronous idle) |
| 23 | ETB | (End of trans. block) |
| 24 | CAN | (Cancel) |
| 25 | EM | (End of medium) |
| 26 | SUB | (Substitute) |
| 27 | ESC | (Escape) |
| 28 | FS | (File separator) |
| 29 | GS | (Group separator) |
| 30 | RS | (Record separator) |
| 31 | US | (Unit separator) |
| 127 | DEL | (Delete) |

### ASCII printable characters

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

### Extended ASCII characters

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 128 | Ç | 160 | á | 192 | └ | 224 | Ó |
| 129 | ü | 161 | í | 193 | ┴ | 225 | ß |
| 130 | é | 162 | ó | 194 | ┬ | 226 | Ô |
| 131 | â | 163 | ú | 195 | ├ | 227 | Ò |
| 132 | ä | 164 | ñ | 196 | ─ | 228 | õ |
| 133 | à | 165 | Ñ | 197 | ┼ | 229 | Õ |
| 134 | å | 166 | ª | 198 | ã | 230 | µ |
| 135 | ç | 167 | º | 199 | Ã | 231 | þ |
| 136 | ê | 168 | ¿ | 200 | ╚ | 232 | Þ |
| 137 | ë | 169 | ® | 201 | ╔ | 233 | Ú |
| 138 | è | 170 | ¬ | 202 | ╩ | 234 | Û |
| 139 | ï | 171 | ½ | 203 | ╦ | 235 | Ù |
| 140 | î | 172 | ¼ | 204 | ╠ | 236 | ý |
| 141 | ì | 173 | ¡ | 205 | ═ | 237 | Ý |
| 142 | Ä | 174 | « | 206 | ╬ | 238 | ¯ |
| 143 | Å | 175 | » | 207 | ¤ | 239 | ´ |
| 144 | É | 176 | ░ | 208 | ð | 240 | ≡ |
| 145 | æ | 177 | ▒ | 209 | Ð | 241 | ± |
| 146 | Æ | 178 | ▓ | 210 | Ê | 242 | ‗ |
| 147 | ô | 179 | │ | 211 | Ë | 243 | ¾ |
| 148 | ö | 180 | ┤ | 212 | È | 244 | ¶ |
| 149 | ò | 181 | Á | 213 | ı | 245 | § |
| 150 | û | 182 | Â | 214 | Í | 246 | ÷ |
| 151 | ù | 183 | À | 215 | Î | 247 | ¸ |
| 152 | ÿ | 184 | © | 216 | Ï | 248 | ° |
| 153 | Ö | 185 | ╣ | 217 | ┘ | 249 | ¨ |
| 154 | Ü | 186 | ║ | 218 | ┌ | 250 | · |
| 155 | ø | 187 | ╗ | 219 | █ | 251 | ¹ |
| 156 | £ | 188 | ╝ | 220 | ▄ | 252 | ³ |
| 157 | Ø | 189 | ¢ | 221 | ¦ | 253 | ² |
| 158 | × | 190 | ¥ | 222 | Ì | 254 | ■ |
| 159 | ƒ | 191 | ┐ | 223 | ▀ | 255 | nbsp |

Representing Text

## ASCII Example

**ASCII Code Chart**

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | NUL | SOH | STX | ETX | EOT | ENQ | ACK | BEL | BS | HT | LF | VT | FF | CR | SO | SI |
| 1 | DLE | DC1 | DC2 | DC3 | DC4 | NAK | SYN | ETB | CAN | EM | SUB | ESC | FS | GS | RS | US |
| 2 | | ! | " | # | $ | % | & | ' | ( | ) | * | + | , | - | . | / |
| 3 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | : | ; | < | = | > | ? |
| 4 | @ | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
| 5 | P | Q | R | S | T | U | V | W | X | Y | Z | [ | \ | ] | ^ | _ |
| 6 | ` | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o |
| 7 | p | q | r | s | t | u | v | w | x | y | z | { | \| | } | ~ | DEL |

| | | | | | |
|---|---|---|---|---|---|
| 01001000 | 01100101 | 01101100 | 01101100 | 01101111 | 00101110 |
| H | e | l | l | o | . |

National Taiwan University OpenCourseWare 臺大開放式課程

Instructor: Tian-Li Yu Data Storage 17 / 1

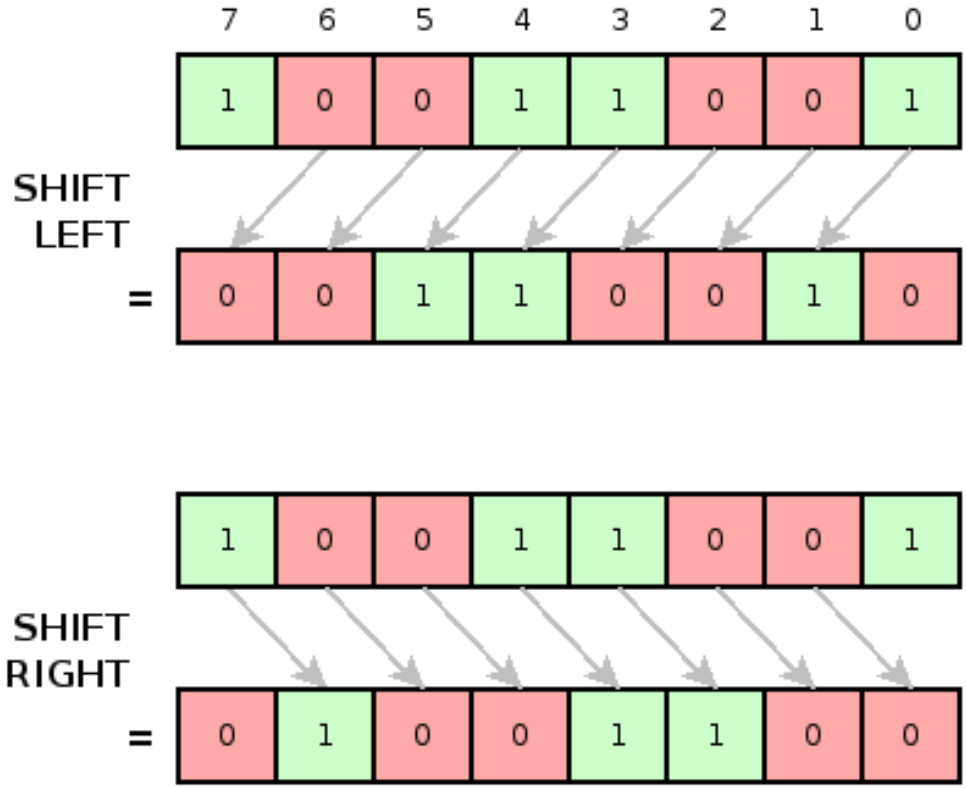
# Bitwise Operation

## Bitwise Operators

int a = 10, b = 2 for all examples below

| Operator | Meaning | Example | Result |
|---|---|---|---|
| ~ | Bitwise unary NOT | ~a | -11 |
| & | Bitwise AND | a&b | 2 |
| \| | Bitwise OR | a\|b | 10 |
| ^ | Bitwise Ex-OR | a^b | 8 |
| >> | Shift right | a>>1 | 5 |
| >>> | Shift right zero fill | a>>>1 | 5 |
| << | Shift left | a<<1 | 20 |
| &= | Bitwise AND assignment | a &= b | 2 |
| \|= | Bitwise OR assignment | a \|= b | 10 |
| ^= | Bitwise Ex-OR assignment | a ^= b | 8 |
| >>= | Shift right assignment | a >>= 1 | 5 |
| >>>= | Shift right zero fill assignment | a >>>=1 | 5 |
| <<= | Shift left assignment | a <<= 1 | 20 |

Startertutorials.com




## Machine Language

### Machine Language

- Instructions, like registers and words of data, are also 32 bits long
  - Example: add $t0, $s1, $s2
  - registers have numbers, $t0=8, $s1=17, $s2=18
- Instruction Format:

| 000000 | 10001 | 10010 | 01000 | 00000 | 100000 |
|---|---|---|---|---|---|
| op | rs | rt | rd | shamt | funct |

- **Can you guess what the field names stand for?**

op = Basic operation of the instruction: opcode
rs = The first register source operand
rt = The second register source operand
rd = The register destination operand
shamt = shift amount
funct = function code

25

## Assembly Language

## Assembly Language

- Tied to the specifics of the underlying machine
- Commands and names to make the code readable and writeable by humans
- Hand-coded assembly code may be more efficient
- E.g., IA32 from Intel

```
      movl   $0, %ecx
.loop:
      cmpl   $1, %edx
      jle    .endloop
      addl   $1, %ecx
      movl   %edx, %eax
      andl   $1, %eax
      je     .else
      movl   %edx, %eax
      addl   %eax, %edx
      addl   %eax, %edx
      addl   $1, %edx
      jmp    .endif
.else:
      sarl   $1, %edx
.endif:
      jmp    .loop
.endloop:
```

## Reading IA32 Assembly Language

- Assembler directives: starting with a period (“.”)
  - E.g., “.section .text” to start the text section of memory
  - E.g., “.loop” for the address of an instruction
- Referring to a register: percent size (“%”)
  - E.g., “%ecx” or “%eip”
- Referring to a constant: dollar sign (“$”)
  - E.g., “$1” for the number 1
- Storing result: typically in the second argument
  - E.g. “addl $1, %ecx” increments register ECX
  - E.g., “movl %edx, %eax” moves EDX to EAX
- Comment: pound sign (“#”)
  - E.g., “# Purpose: Convert lower to upper case”

28

# High level language

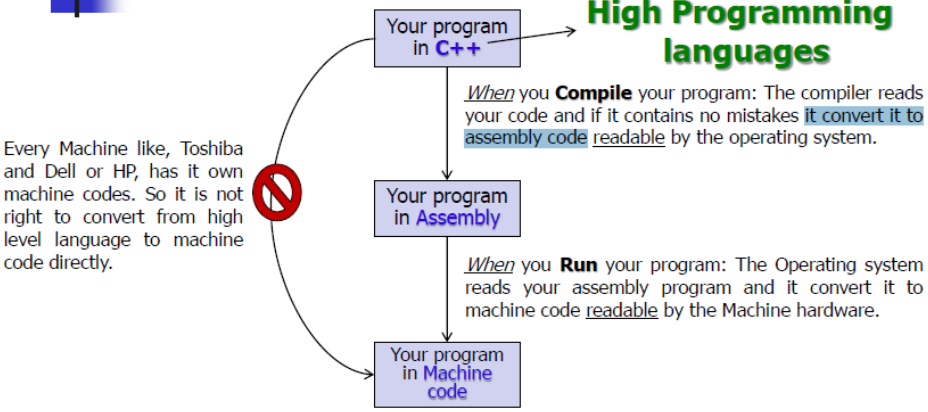
- As stated earlier, a program written in any programming language is a set of logically related instructions. These instructions have two parts, as shown in the following figure:
- The two parts of a programming language instruction are:
  - ,*Operation code (opcode): This part instructs a computer about the operation to be performed.*
  - ,*Operand: This part instructs the computer about the location of the data on which the operation specified by the opcode is to be performed.*

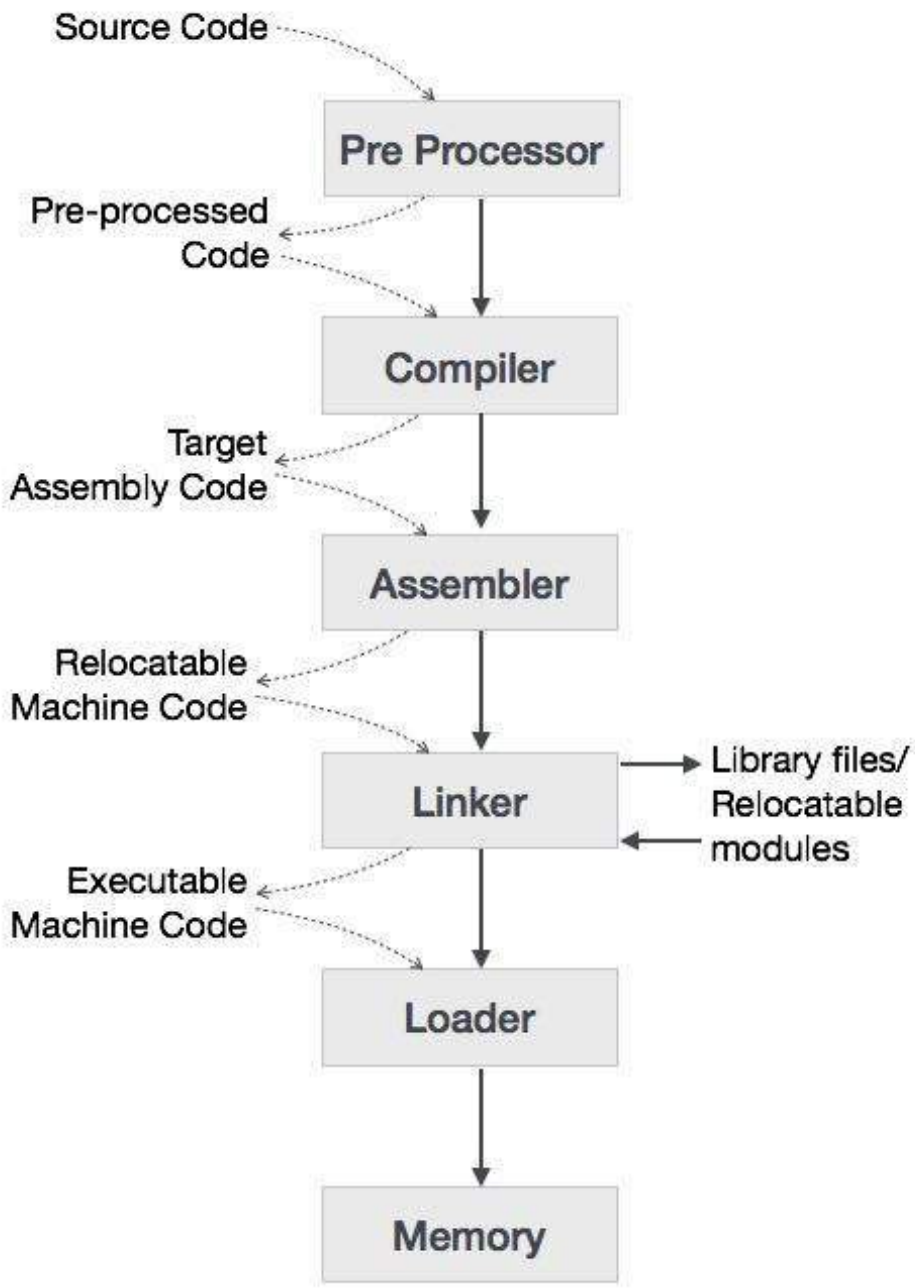

- For example, in the instruction Add A and B, Add is the opcode and A and B are operands

NIIT Hai Phong Software Park 11

## Compiler and Assember

# Compiling and running your program





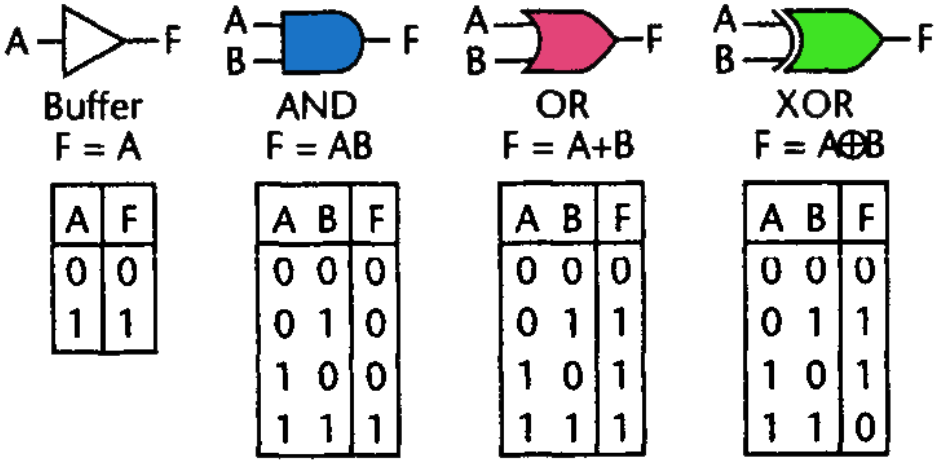

## COMPILER VS LINKER VS LOADER

| COMPILER | LINKER | LOADER |
|---|---|---|
| A software that transforms computer code written in one programming language into another programming language | A computer utility program that takes one or more object files generated by a compiler and combines them into a single executable file | A part of an operating system that is responsible for loading programs to memory |
| Transforms the source code into object code | Combines multiple object code and links them with libraries | Prepares the executable file for running |

Visit www.pediaa.com

# Logic Design

## Logic Gates

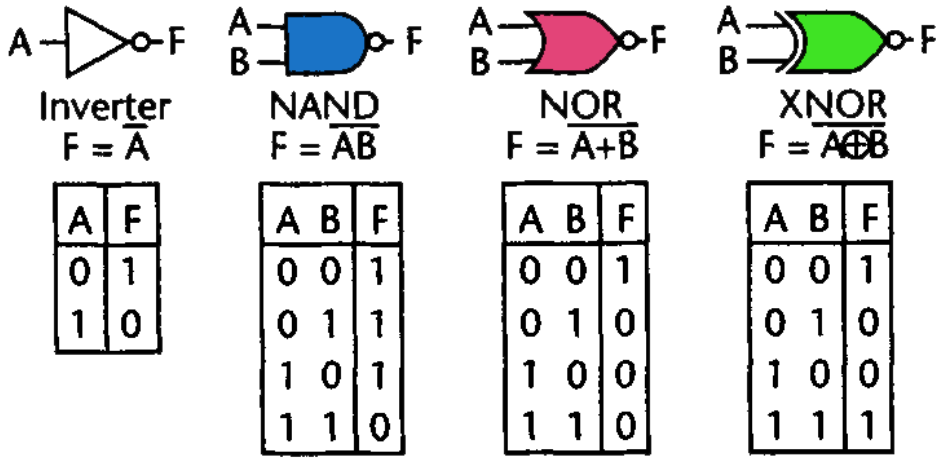



| A | F |
|---|---|
| 0 | 0 |
| 1 | 1 |

| A | B | F |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

| A | B | F |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

| A | B | F |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

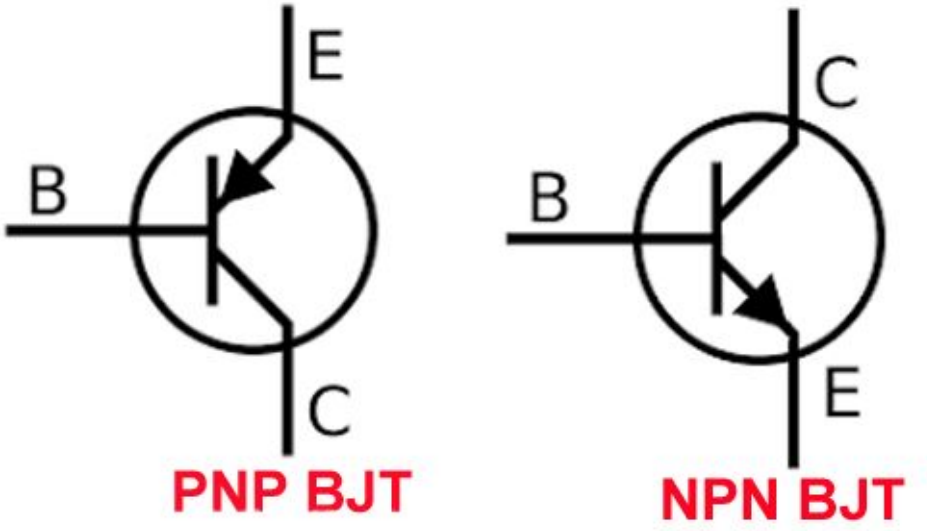



| A | F |
|---|---|
| 0 | 1 |
| 1 | 0 |

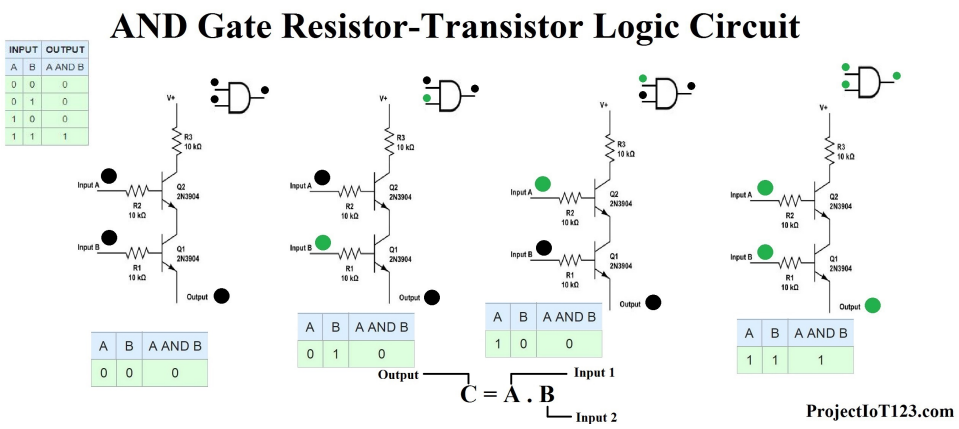
| A | B | F |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

| A | B | F |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

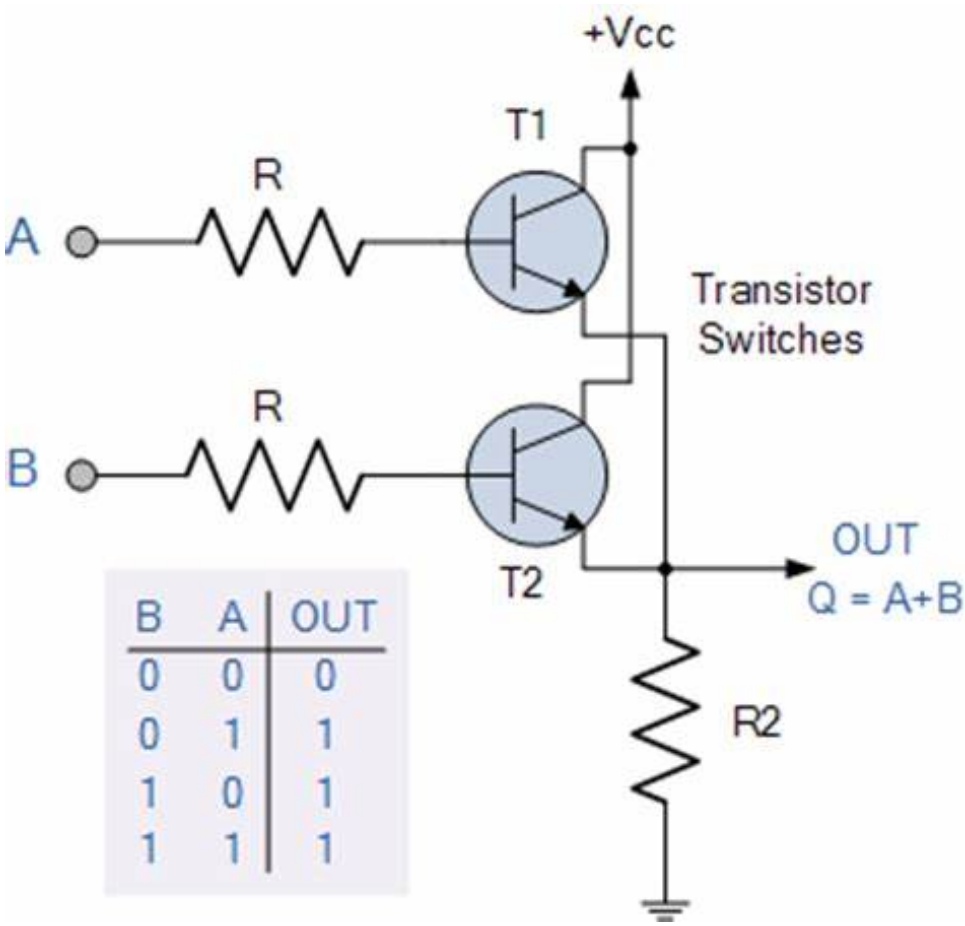
| A | B | F |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |



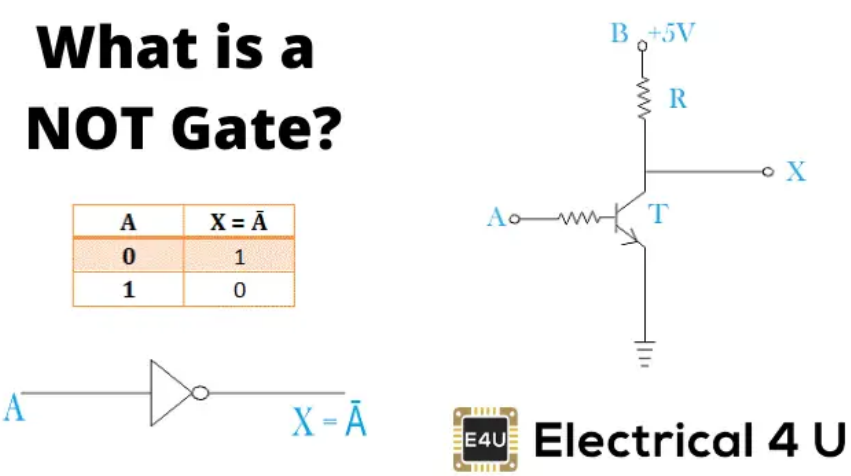

## AND Gate



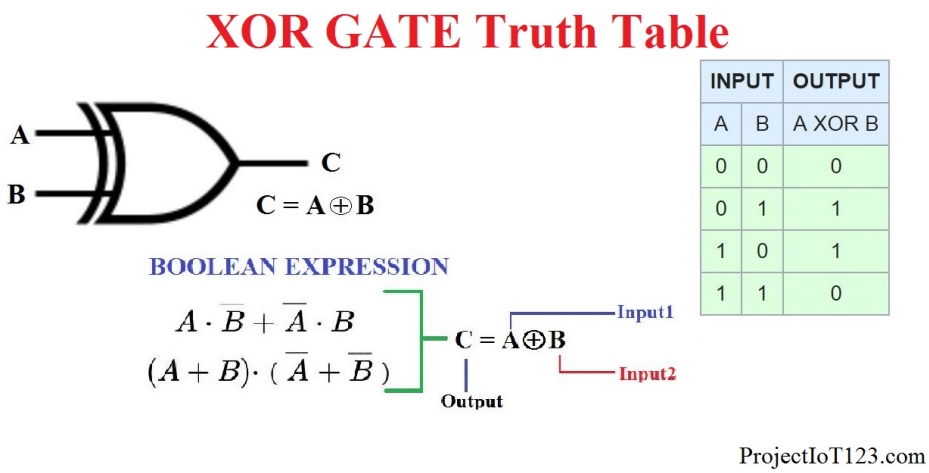

## OR Gate

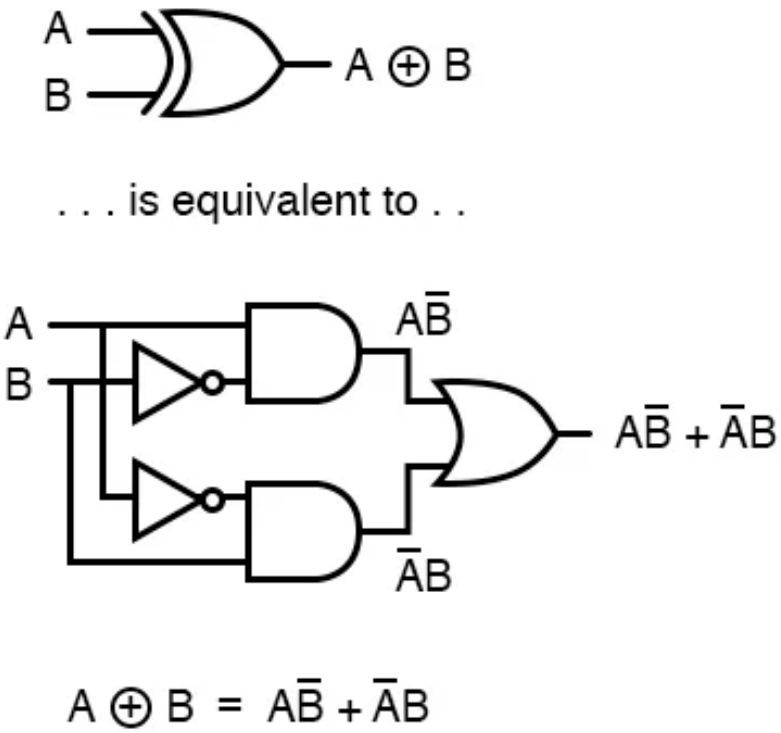


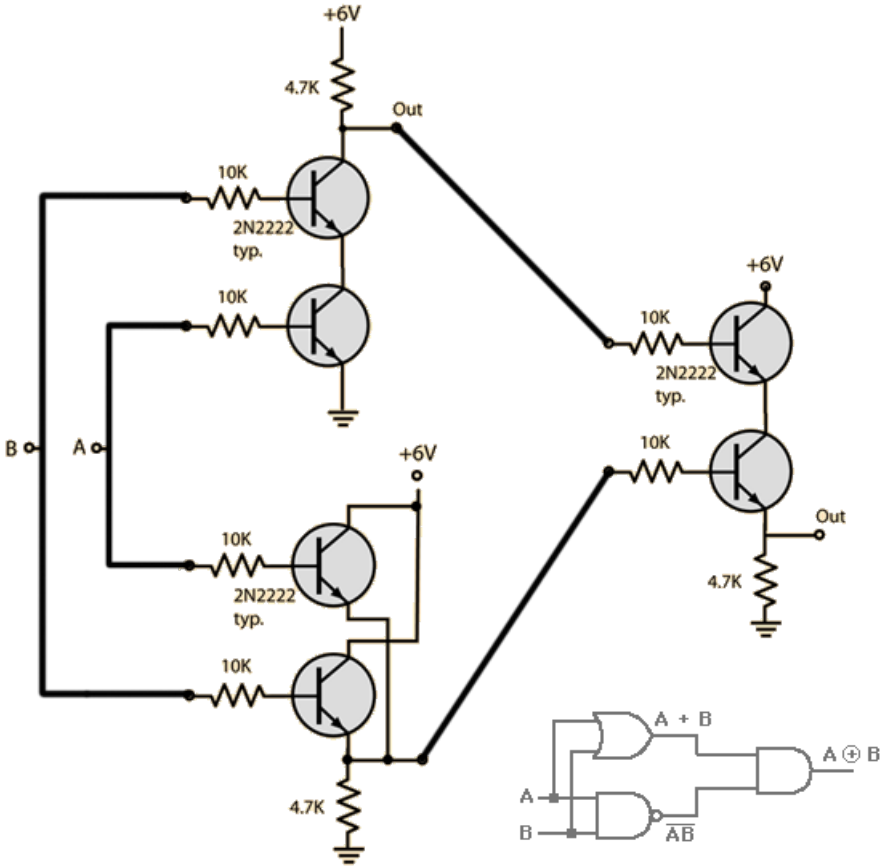

## NOT Gate

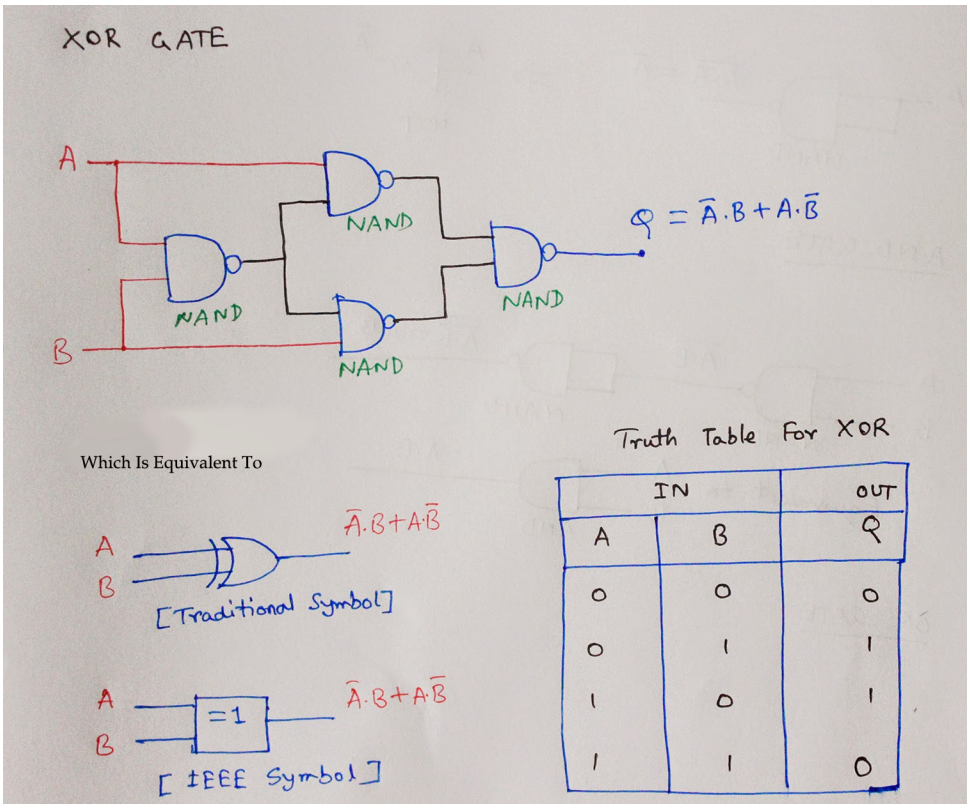

What is a
NOT Gate?
A
X = Ā
0
1
1
0
B +5V
R
X
A
T
A
X = Ā
E4U
Electrical 4 U

## XOR Gate


XOR GATE Truth Table
INPUT
OUTPUT
A
B
A XOR B
0 0 0
0 1 1
1 0 1
1 1 0
A
B
C
C = A⊕B
BOOLEAN EXPRESSION
A · B̄ + Ā · B
(A + B) · ( Ā + B̄ )
C = A⊕B
Input1
Input2
Output
ProjectIoT123.com

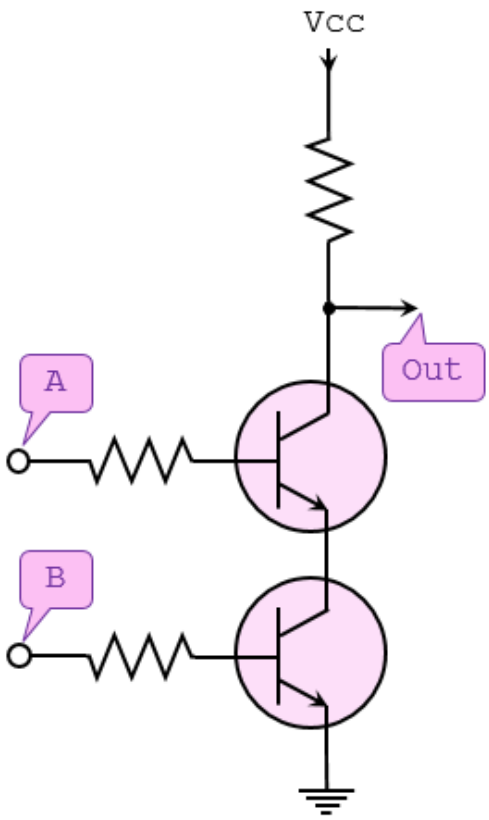

A
B
A ⊕ B
. . . is equivalent to . .
A
B
AB̄
ĀB
AB̄ + ĀB
A ⊕ B = AB̄ + ĀB


+6V
4.7K
Out
10K
2N2222
typ.
10K
B
A
+6V
10K
2N2222
typ.
10K
Out
4.7K
+6V
10K
2N2222
typ.
10K
4.7K
A + B
A ⊕ B
A
B
AB

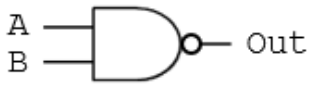

XOR GATE
A
B
NAND
NAND
NAND
NAND
Q = Ā·B + A·B̄
Which Is Equivalent To
A
B
Ā·B+A·B̄
[Traditional Symbol]
A
B
=1
Ā·B+A·B̄
[ IEEE Symbol ]
Truth Table For XOR
IN
OUT
A
B
Q
0 0 0
0 1 1
1 0 1
1 1 0

# NAND Gate

# NAND Gate



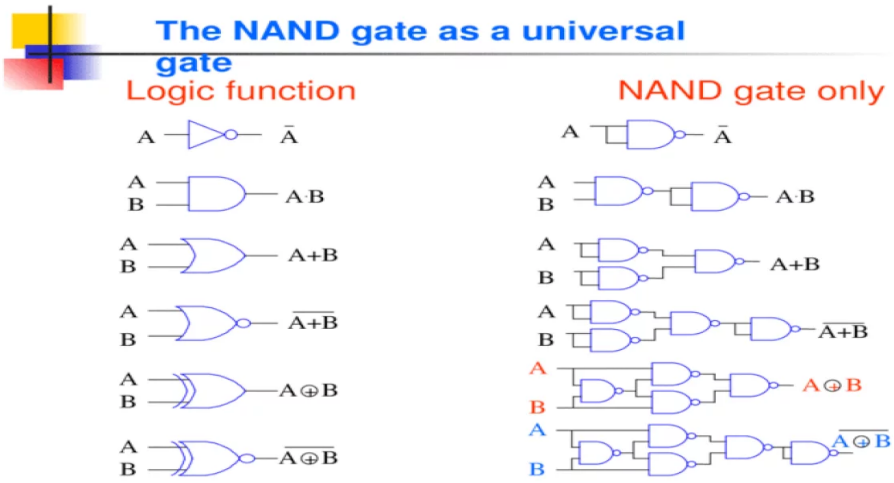

Symbol:



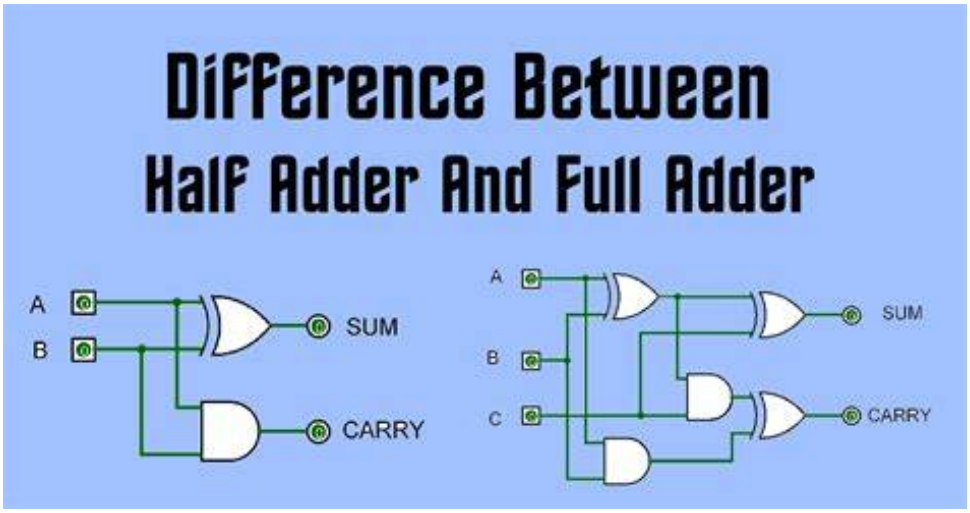

Truth table: NAND Gate

| A | B | Out |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

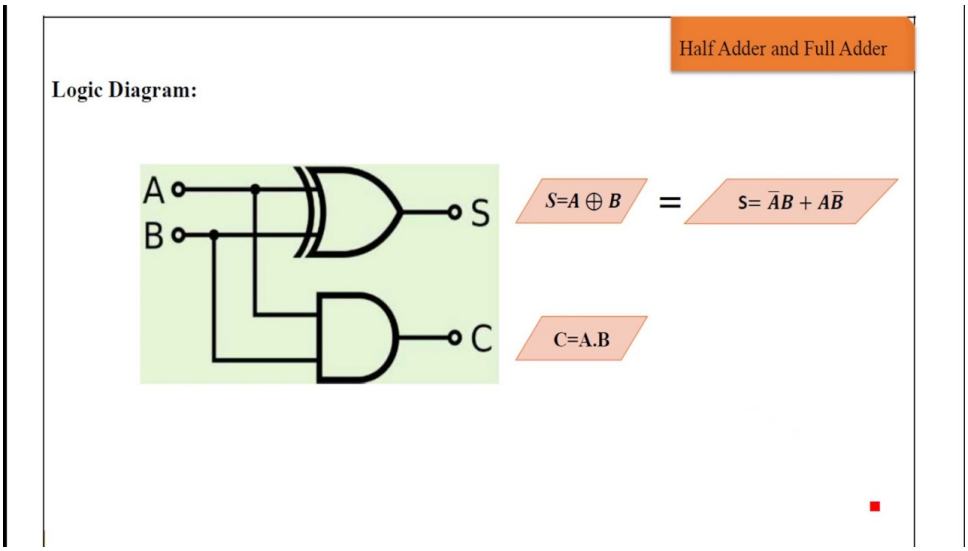
## The NAND gate as a universal gate




## Half-Adder vs Full-Adder




## Half-Adder



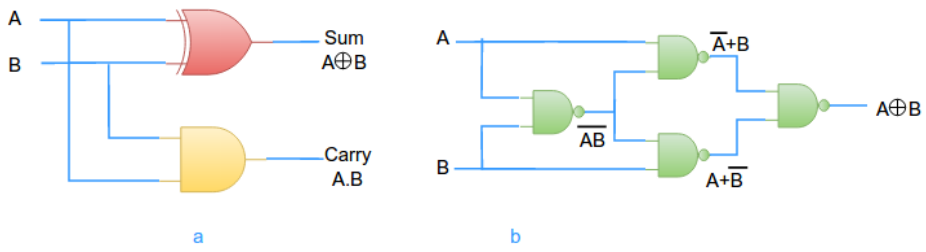
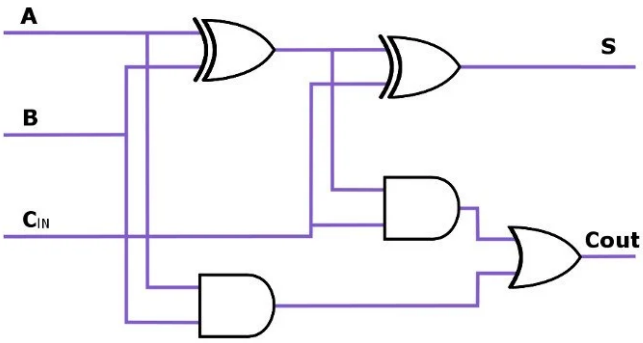
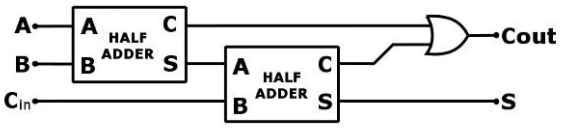



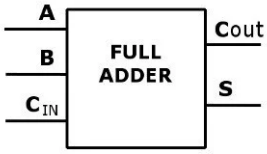

Figure-1 : a)Half Adder b)XOR implementation using NAND gates

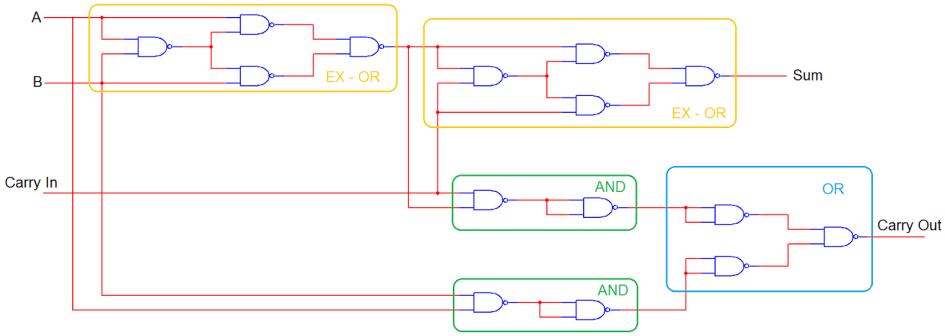
## Full Adder



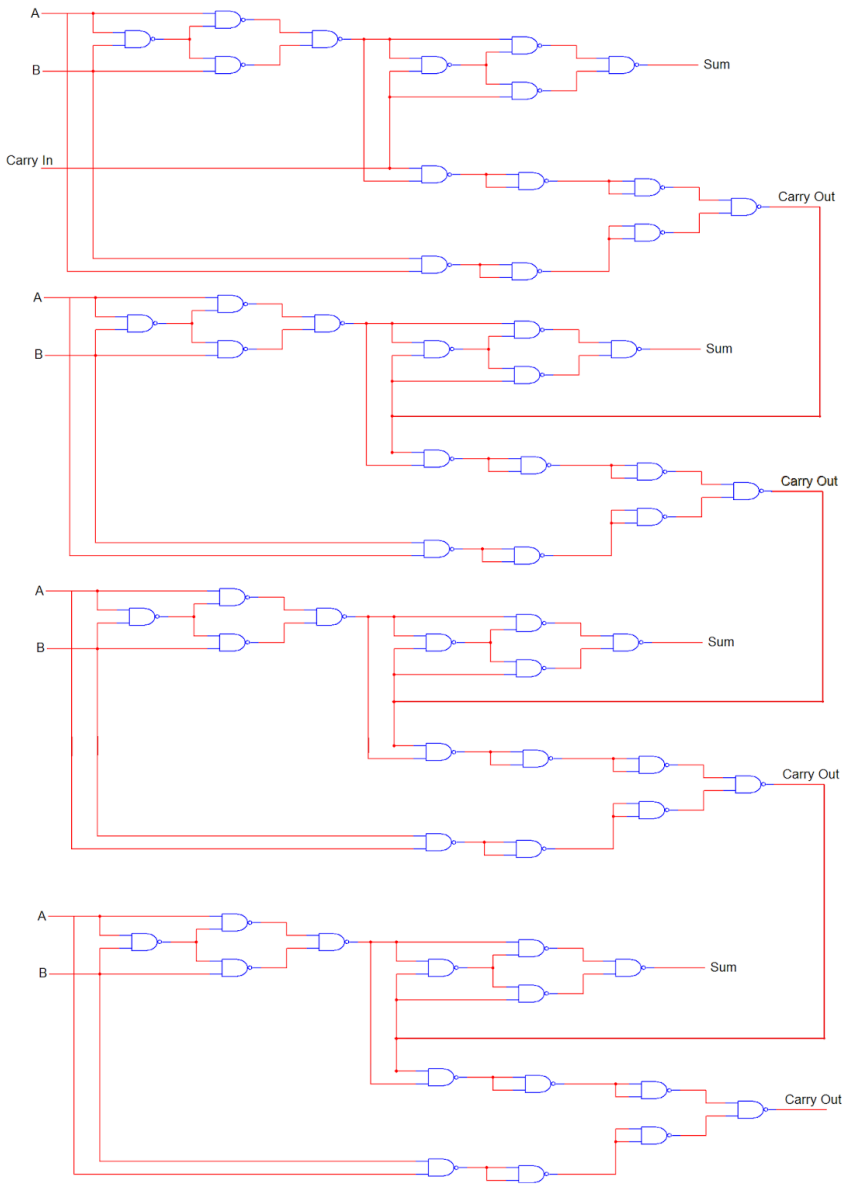

| Input | | | Output | |
|---|---|---|---|---|
| A | B | Cin | Sum | Carry |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |




# 4 Bit Full Adder




# Substract

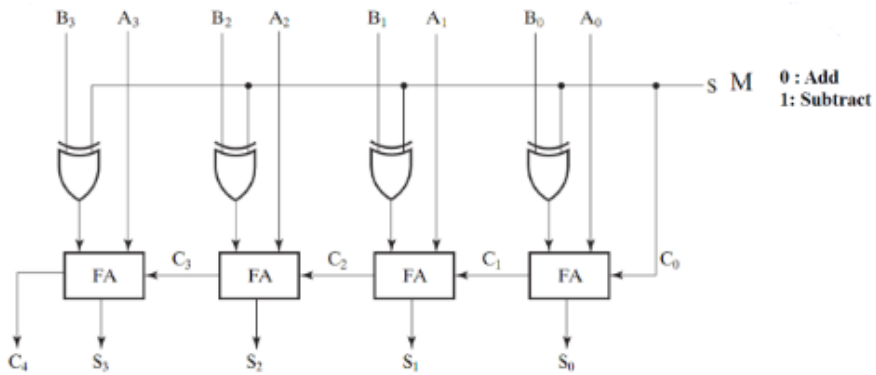
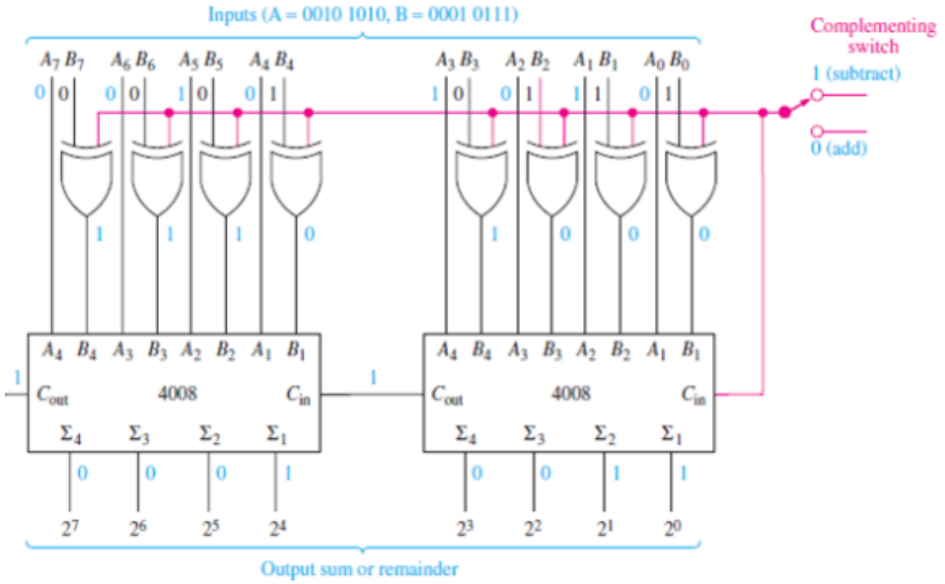
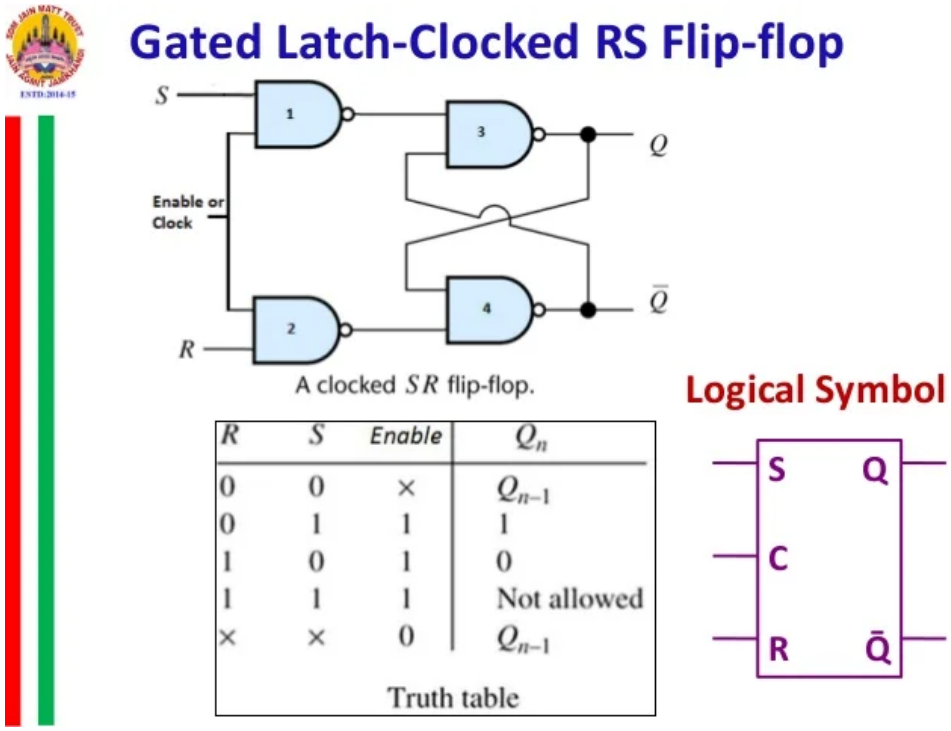



**Fig. 1** Modular design of 4-bit adder/subtractor

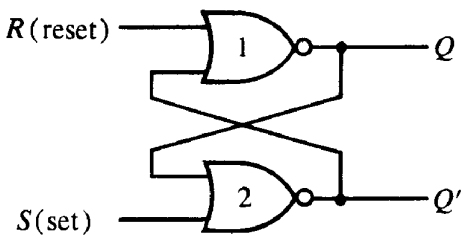



# Circuits as Memory

## SR Filp Folp



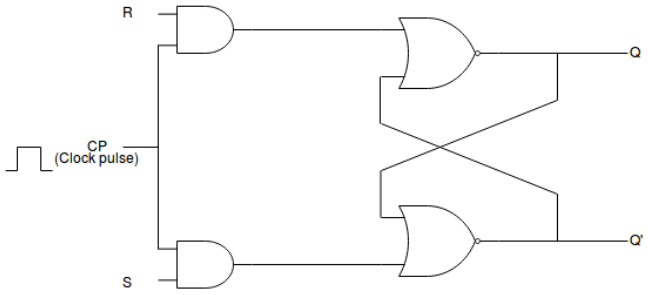

| R | S | Enable | $Q_n$ |
|---|---|---|---|
| 0 | 0 | × | $Q_{n-1}$ |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | Not allowed |
| × | × | 0 | $Q_{n-1}$ |




(a) Logic diagram

| S | R | Q | Q′ | |
|---|---|---|---|---|
| 1 | 0 | 1 | 0 | |
| 0 | 0 | 1 | 0 | (after $S = 1, R = 0$) |
| 0 | 1 | 0 | 1 | |
| 0 | 0 | 0 | 1 | (after $S = 0, R = 1$) |
| 1 | 1 | 0 | 0 | |

(b) Truth table




a) Logic diagram

fig: Clocked SR flip flop

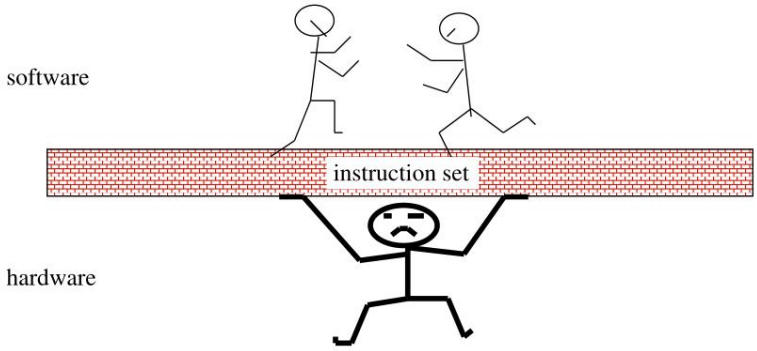
| Q | S | R | Q(t+1) |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | Intermediate |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | Intermediate |

b) Truth table

---

# Instruction Set Architecture - Interface of S/W and H/w

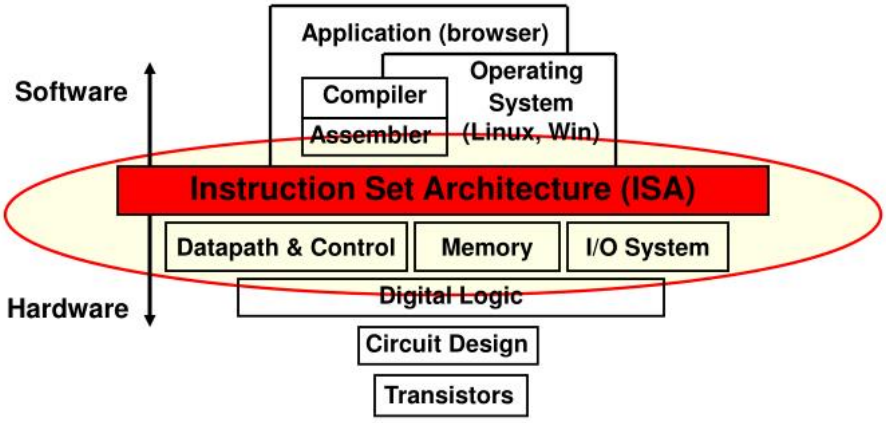
## Instruction Set Architecture: Critical Interface




- Properties of a good abstraction
  - Lasts through many generations (portability)
  - Used in many different ways (generality)
  - Provides convenient functionality to higher levels
  - Permits an efficient implementation at lower levels

*Computer Architecture- 8*

# The Instruction Set Architecture




ISA

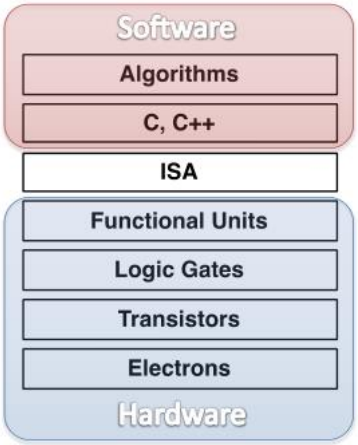
## Instruction Set Architecture

- The computer ISA defines all the *programmer-visible* components and operations of the computer
  - Memory organization
    - address space -- how may locations can be addressed?
    - addressibility -- how many bits per location?
  - Register set
    - how many? what size? how are they used?
  - Instruction set
    - opcodes
    - data types
    - addressing modes
- ISA provides all information needed for someone that wants to write a program in machine language (or translate from a high-level language to machine language).

BYU CS 224 ISA 5

# Instruction Set Architecture (ISA)

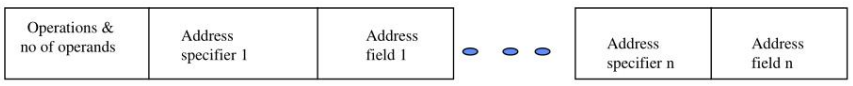
- ISA is the interface provided by the hardware to the software
  - Defines the available:
    - instructions
    - registers
    - addressing modes
    - memory architecture
    - interrupt and exception handling
    - external I/O
  - Syntax defined by assembly language
    - Symbolic representation of the machine instructions
  - Examples: x86, ARM, HCS12



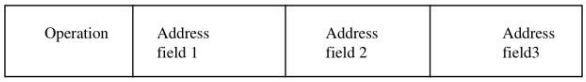




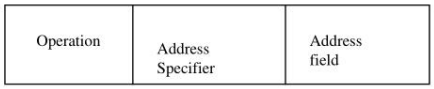

## Three Examples of Instruction Set Encoding

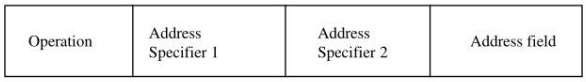



**Variable: VAX (1-53 bytes)**




**Fixed: DLX, MIPS, PowerPC, SPARC**





**Hybrid : IBM 360/370, Intel 80x86**

**EECC551 - Shaaban**

#20 Lec # 2 Fall 2000 9-12-2000

## CICS vs RICS

# CISC vs. RISC

**Complex Instruction Set Computer**

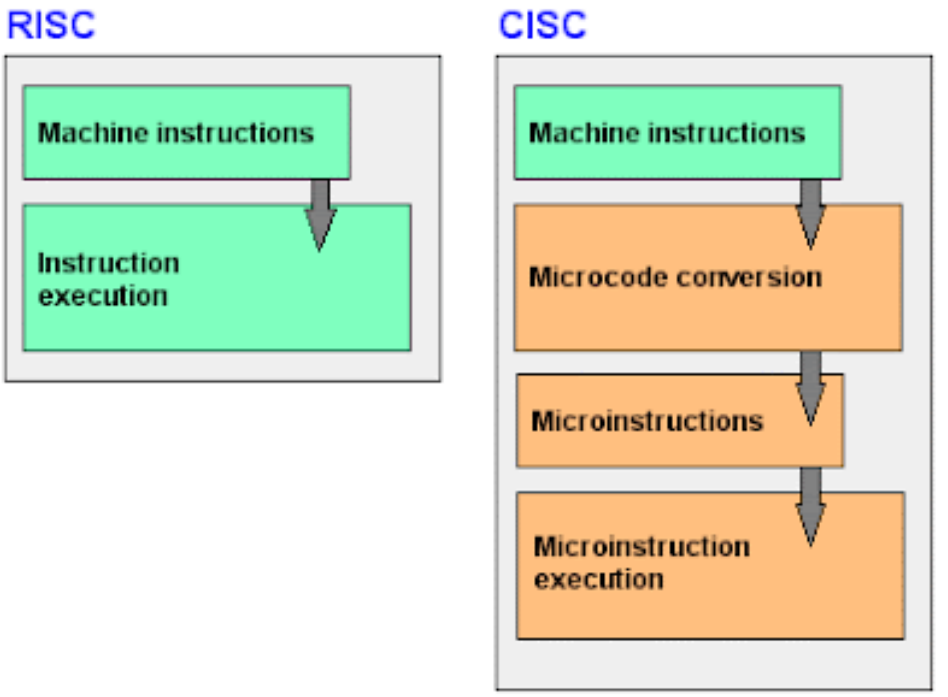
- Many instructions
  - e.g., 75-100
- Many instructions are macro-like
  - Simplifies programming
- Most microcontrollers are based on CISC concept
  - e.g., PDP-11, VAX, Motorola 68k
  - PIC is an exception

**Reduced Instruction Set Computers**

- Few instructions
  - e.g., 30-40
- Smaller chip, smaller pin count, & very low-power consumption
  - Simple but fast instructions
- Harvard architecture, instruction pipelining
- Industry trend for microprocessor design
  - e.g., Intel Pentium, PIC

24

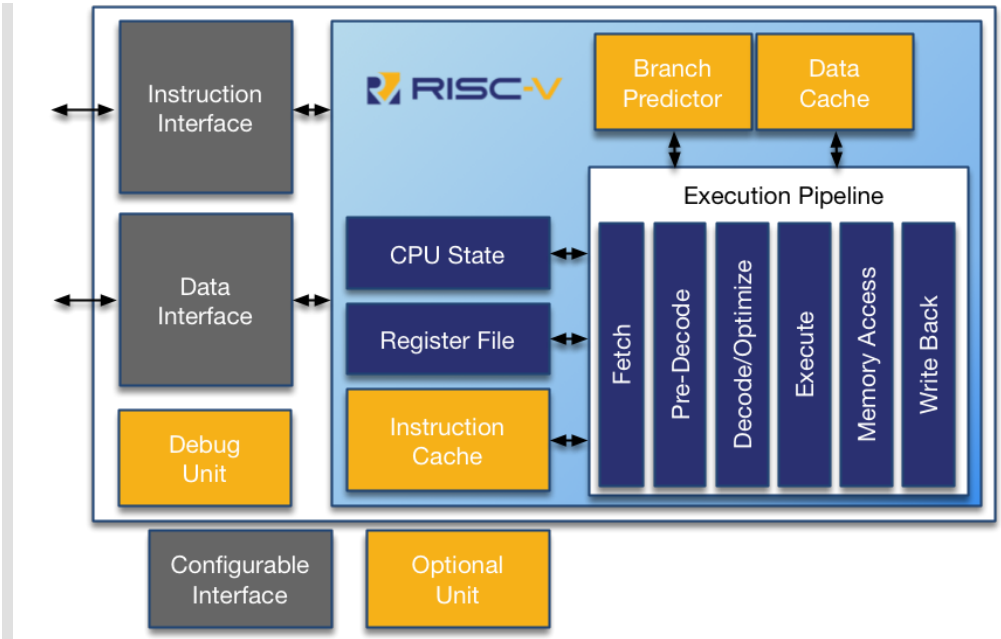
From Computer Desktop Encyclopedia
© 1998 The Computer Language Co. Inc.



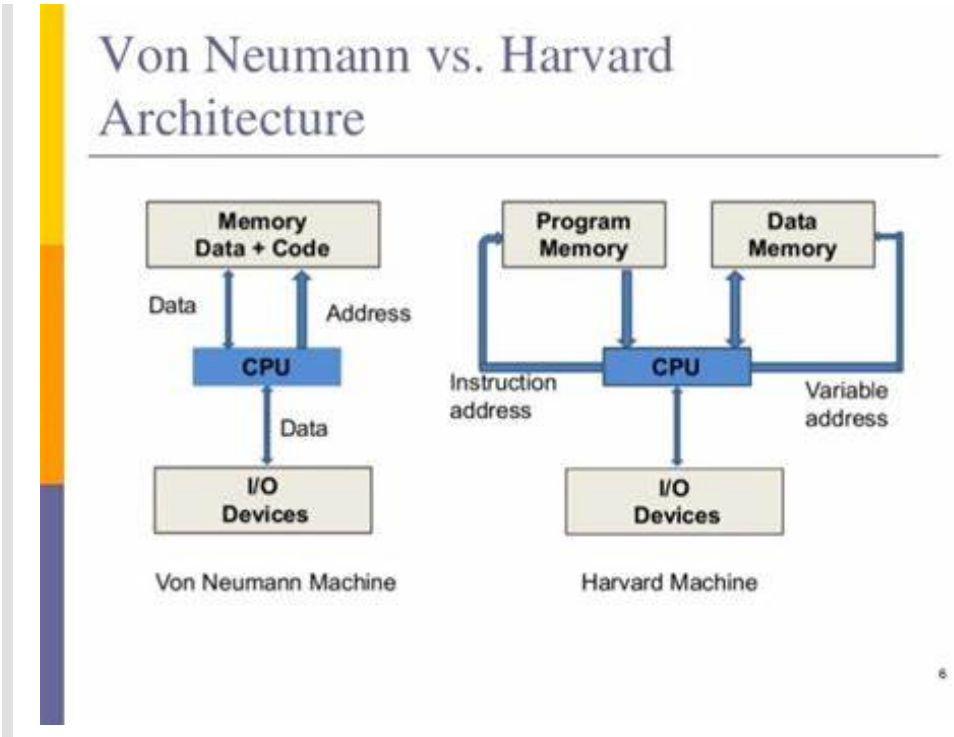

**32-bit RISC-V Instruction Formats**

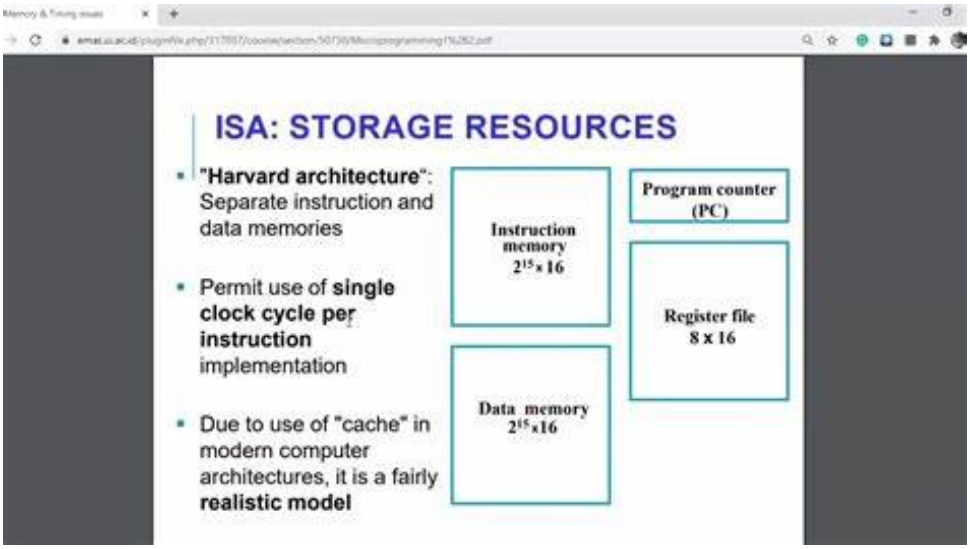
| Instruction Formats | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Register/register | funct7 | | | | | | | rs2 | | | | | rs1 | | | | | funct3 | | | rd | | | | | opcode | | | | | | |
| Immediate | imm[11:0] | | | | | | | | | | | | rs1 | | | | | funct3 | | | rd | | | | | opcode | | | | | | |
| Upper Immediate | imm[31:12] | | | | | | | | | | | | | | | | | | | | rd | | | | | opcode | | | | | | |
| Store | imm[11:5] | | | | | | | rs2 | | | | | rs1 | | | | | funct3 | | | imm[4:0] | | | | | opcode | | | | | | |
| Branch | [12] | imm[10:5] | | | | | | rs2 | | | | | rs1 | | | | | funct3 | | | imm[4:1] | | | | [11] | opcode | | | | | | |
| Jump | [20] | imm[10:1] | | | | | | | | | | [11] | imm[19:12] | | | | | | | | rd | | | | | opcode | | | | | | |

- **opcode (7 bit):** partially specifies which of the 6 types of *instruction formats*
- **funct7 + funct3 (10 bit):** combined with **opcode**, these two fields describe what operation to perform
- **rs1 (5 bit):** specifies register containing first operand
- **rs2 (5 bit):** specifies second register operand
- **rd (5 bit):**: Destination register specifies register which will receive result of computation

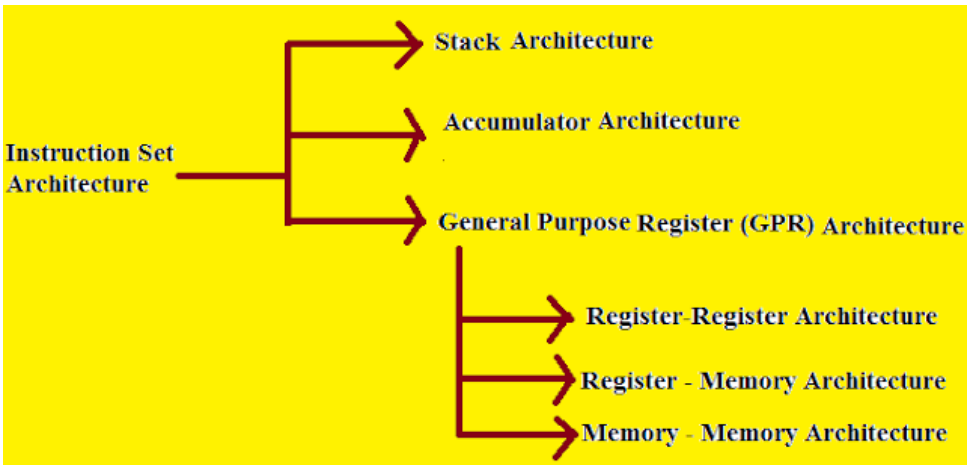


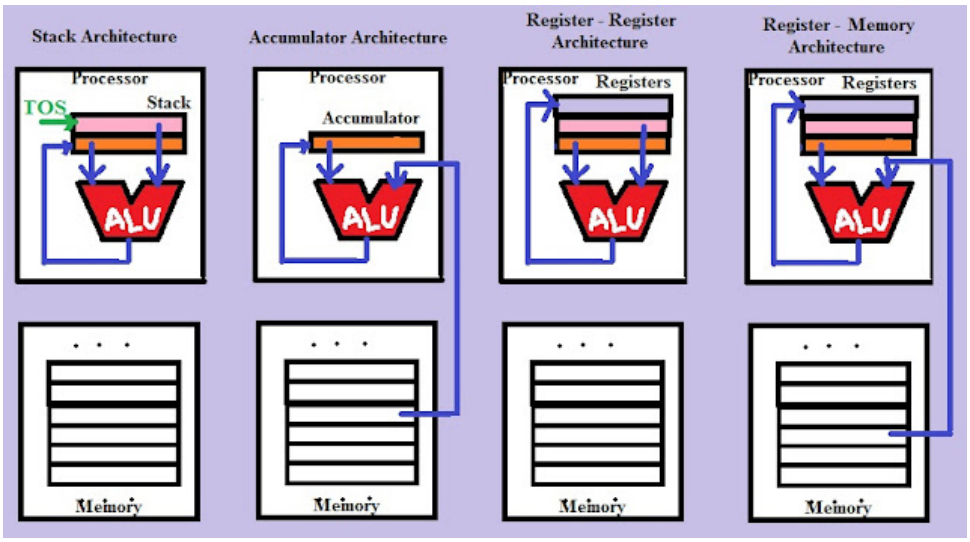

# Memory Architecture






## Classification of ISA (or) Types of ISA




## ISA - MIPS Instrucstion Foramt and Addressing Model

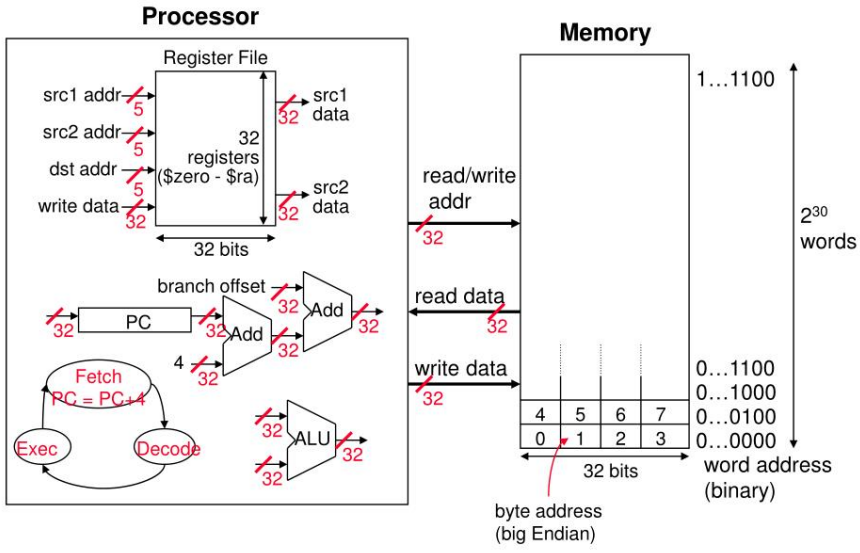
### MIPS Design Priciple

# MIPS Design Paradigms

- Simplicity favors regularity
  - all instructions single size
  - three register operands in arithmetic instr.
  - keep register fields in the same place
- Smaller is faster
  - 32 registers
- Make good compromises
  - large addresses and constants versus unique instruction length
- Make the common case fast
  - PC-relative addressing for conditional branches

# MIPS Organization

## MIPS Organization So Far




34

# MIPS Register File

# The MIPS ISA – Register File

- When writing assembly, these registers can be referenced by their address (number) or name
- General purpose and special purpose registers

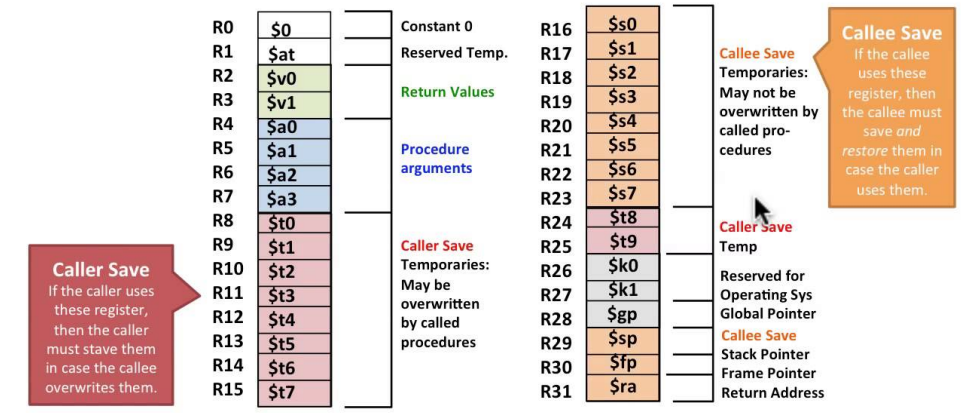
| # | Name | Purpose |
|---|---|---|
| $0 | $zero | Constant zero |
| $1 | $at | Reserved for assembler |
| $2 | $v0 | Function return value |
| $3 | $v1 | |
| $4 | $a0 | Function parameter |
| $5 | $a1 | |
| $6 | $a2 | |
| $7 | $a3 | |
| $8 | $t0 | Temporary – Caller-saved |
| $9 | $t1 | |
| $10 | $t2 | |
| $11 | $t3 | |
| $12 | $t4 | |
| $13 | $t5 | |
| $14 | $t6 | |
| $15 | $t7 | |

| # | Name | Purpose |
|---|---|---|
| $16 | $s0 | Temporary – Callee-saved |
| $17 | $s1 | |
| $18 | $s2 | |
| $19 | $s3 | |
| $20 | $s4 | |
| $21 | $s5 | |
| $22 | $s6 | |
| $23 | $s7 | |
| $24 | $t8 | Temporary – Caller-saved |
| $25 | $t9 | |
| $26 | $k0 | Reserved for OS |
| $27 | $k1 | |
| $28 | $gp | Global pointer |
| $29 | $sp | Stack pointer |
| $30 | $fp | Frame pointer |
| $31 | $ra | Function return address |

17

## Who saves what?

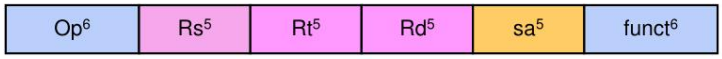
53




# Instruction Format

## Instruction Formats

- All instructions are 32-bit wide. Three instruction formats:
- Register (R-Type)
  - Register-to-register instructions
  - Op: operation code specifies the format of the instruction




- Immediate (I-Type)
  - 16-bit immediate constant is part in the instruction

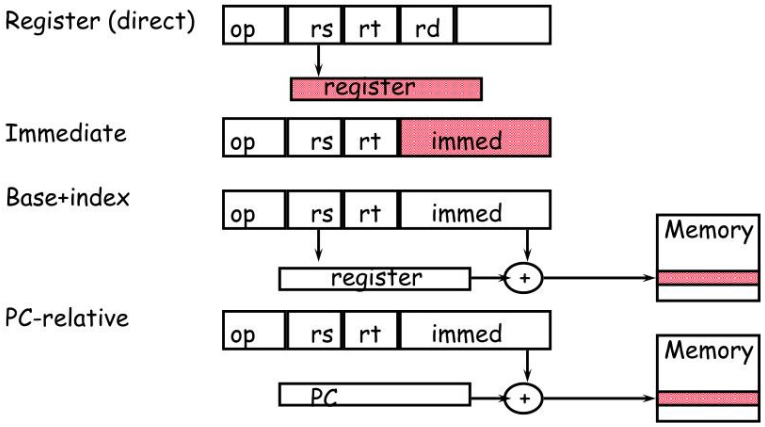

- Jump (J-Type)
  - Used by jump instructions


*Instruction Set Architecture* *ICS 233 – Computer Architecture and Assembly Language – KFUPM* *© Muhamed Mudawar slide 9*

# Addressing Model

## Example: MIPS Instruction Formats and Addressing Modes

- All instructions 32 bits wide




ECE 361 3-45

## 5.6 Addressing Modes

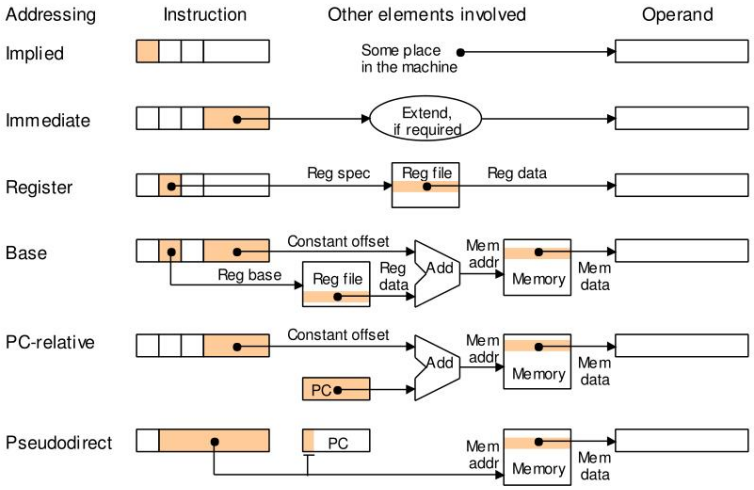



Figure 5.11 Schematic representation of addressing modes in MiniMIPS.

Computer Architecture, Instruction-Set Architecture Slide 22

## Instruction Categories

MIPS: ISA

| Category | Instruction | Op Code | Example | Meaning |
|---|---|---|---|---|
| Arithmetic (R & I format) | Add | 0 and 32 | add $s1, $s2, $s3 | $s1 = $s2 + $s3 |
| | Subtract | 0 and 34 | sub $s1, $s2, $s3 | $s1 = $s2 - $s3 |
| | add immediate | 8 | addi $s1, $s2, 6 | $s1 = $s2 + 6 |
| | or immediate | 13 | ori $s1, $s2, 6 | $s1 = $s2 v 6 |
| Logical (R & I format) | And | 0 and 36 | and $s1, $s2, $s3 | $s1 = $s2 & $s3 |
| | Or | 0 and 37 | or $s1, $s2, $s3 | $s1 = $s2 \| $s3 |
| | Nor | 0 and 39 | nor $s1, $s2, $s3 | $s1 = ~($s2 \| $s3) |
| | And immediate | 12 | andi $s1, $s2, 100 | $s1 = $s2 & 100 |
| | Or immediate | 13 | ori $s1, $s2, 100 | $s1 = $s2 \| 100 |
| | Shift left logical | 0 and 0 | sll $s1, $s2, 10 | $s1 = $s2 << 10 |
| | Shift right logical | 0 and 2 | srl $s1, $s2, 10 | $s1 = $s2 >> 10 |
| Data Transfer (I format) | load word | 35 | lw $s1, 24($s2) | $s1 = Memory[$s2+24] |
| | store word | 43 | sw $s1, 24($s2) | Memory[$s2+24] = $s1 |
| | load byte | 32 | lb $s1, 25($s2) | $s1 = Memory[$s2+25] |
| | store byte | 40 | sb $s1, 25($s2) | Memory[$s2+25] = $s1 |
| | load upper imm | 15 | lui $s1, 6 | $s1 = 6 * $2^{16}$ |
| Cond. Branch (I & R format) | br on equal | 4 | beq $s1, $s2, L | if ($s1==$s2) go to L |
| | br on not equal | 5 | bne $s1, $s2, L | if ($s1 !=$s2) go to L |
| | set on less than | 0 and 42 | slt $s1, $s2, $s3 | if ($s2<$s3) $s1=1 else $s1=0 |
| | set on less than immediate | 10 | slti $s1, $s2, 6 | if ($s2<6) $s1=1 else $s1=0 |
| Uncond. Jump (J & R format) | jump | 2 | j 2500 | go to 10000 |
| | jump register | 0 and 8 | jr $t1 | go to $t1 |
| | jump and link | 3 | jal 2500 | go to 10000; $ra=PC+4 |

40

# MIPS ISA as an Example

- Instruction categories:
  - Load/Store
  - Computational
  - Jump and Branch
  - Floating Point
  - Memory Management
  - Special

Registers

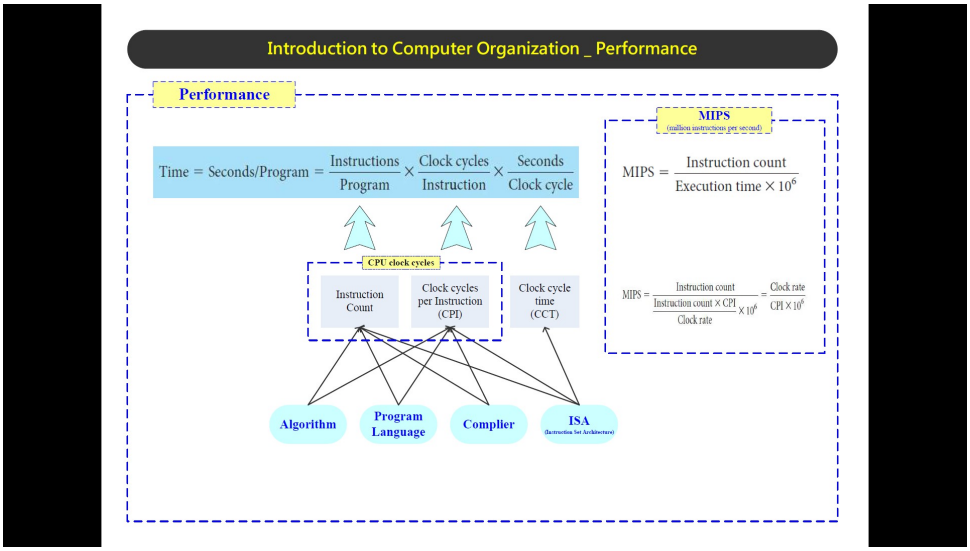
| $r0 - $r31 |
|---|
| PC |
| HI |
| LO |

3 Instruction Formats: all 32 bits wide

| OP | $rs | $rt | $rd | sa | funct |
|---|---|---|---|---|---|
| OP | $rs | $rt | immediate | | |
| OP | jump target | | | | |

National Tsing Hua University
COMPUTER ARCHITECTURE 6

# ISA - Performance



## Performance Summary

- Measurements at different level
- Combined to yield execution time in seconds per program

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

- Performance depends on
  - Algorithm
  - Programming language
  - Compiler
  - Instruction set architecture



## CPU Execution Time: The CPU Equation

- **A program is comprised of a number of instructions, I**
  - **Measured in: instructions/program**
- **The average instruction takes a number of cycles per instruction (CPI) to be completed.**
  - **Measured in: cycles/instruction, CPI**
- **CPU has a fixed clock cycle time C = 1/clock rate**
  - **Measured in: seconds/cycle**
- **CPU execution time is the product of the above three parameters as follows:**

$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

$$T = I \times CPI \times C$$

# Speed Up Equation for Pipelining

$$CPI_{pipelined} = \text{Ideal CPI} + \text{Average Stall cycles per Inst}$$

$$\text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline depth}}{\text{Ideal CPI} + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{unpipelined}}{\text{Cycle Time}_{pipelined}}$$

For simple RISC pipeline, CPI = 1:

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{unpipelined}}{\text{Cycle Time}_{pipelined}}$$

CS211 41

*Example*: *Speed-up Challenge*

課程:計算機組織 / 老師:朱宗賢

- Suppose you want to achieve a speed-up 90 times faster with 100 processors. What percentage of the original computation can be sequential?

**Amdahl's law concept (1967):** The speedup of a program using multiple processors in parallel computing is limited by the time needed for the sequential fraction of the program.

- $T_{new} = T_{parallelizable}/100 + T_{sequential}$
- $\text{Speedup} = \frac{1}{(1-F_{parallelizable}) + F_{parallelizable}/100} = 90$
- Solving: $F_{parallelizable} = 0.999$
- Need sequential part to be 0.1% of original time

1. (*2 marks*) A particular program has an execution time of 12 seconds on a particular system that has a clock rate of 2.5 GHz. 8 billion (i.e., 8 x $10^9$) machine language instructions are executed. What is the CPI?

2. (*10 marks*) Suppose that the following clock cycles per instruction, and frequencies of usage by a particular program, have been determined for the four instruction types of a particular system with a clock cycle time of 0.5 nanoseconds.

| Instruction type | Clock cycles per instruction | Frequency |
|---|---|---|
| *A* | 3 | 10% |
| *B* | 1 | 50% |
| *C* | 2 | 20% |
| *D* | 4 | 20% |

(a) What MIPS rating is achieved for this program on this system?

(b) Suppose that the CPU execution time is 0.5 seconds. What must be the number of instructions executed?

(c) Suppose that through use of a compiler optimization, it is possible to **halve** the number of executed type *B* instructions, with the numbers of all other types of instructions remaining the same.

(i) Would the MIPS rating increase, or would it decrease, when using this optimization?

(ii) By what percentage would the MIPS rating change?

(iii) By what percentage would the CPU execution time decrease?

3. Calculate the CPU time to execute a program containing one million machine instructions, when the average CPI of the CPU is 2.5 and CPU clock frequency is 2GHz. [15pts]

4. Let us overclock the machine given in problem 3 by increasing the CPU clock frequency from 2GHz to 2.5GHz. Calculate the CPU time of the overclocked machine running the same program containing one million instructions. How much faster (i.e., performance ratio) is the overclocked machine than the original machine? [15pts]

(*8 marks*) Suppose that the following clock cycles per instruction, and frequencies of usage by a particular program, have been determined for the four instruction types of a system with a clock rate of 2 GHz.

| Instruction type | Clock cycles per instruction | Frequency |
|---|---|---|
| *A* | 3 | 20% |
| *B* | 1 | 40% |
| *C* | 2 | 30% |
| *D* | 4 | 10% |

(a) What MIPS rating is achieved is achieved for this program on this system?

(b) Suppose that 500 x $10^6$ instructions are executed in total. What is the CPU execution time?

(c) Suppose it was possible to replace particular pairs of type *C* instructions with single type *A* instructions. Would the MIPS rating increase, or decrease, with this change?

(d) If it were possible to replace 2/3 of the type *C* instructions in the manner described in part (c), by what percentage would CPU execution time decrease?

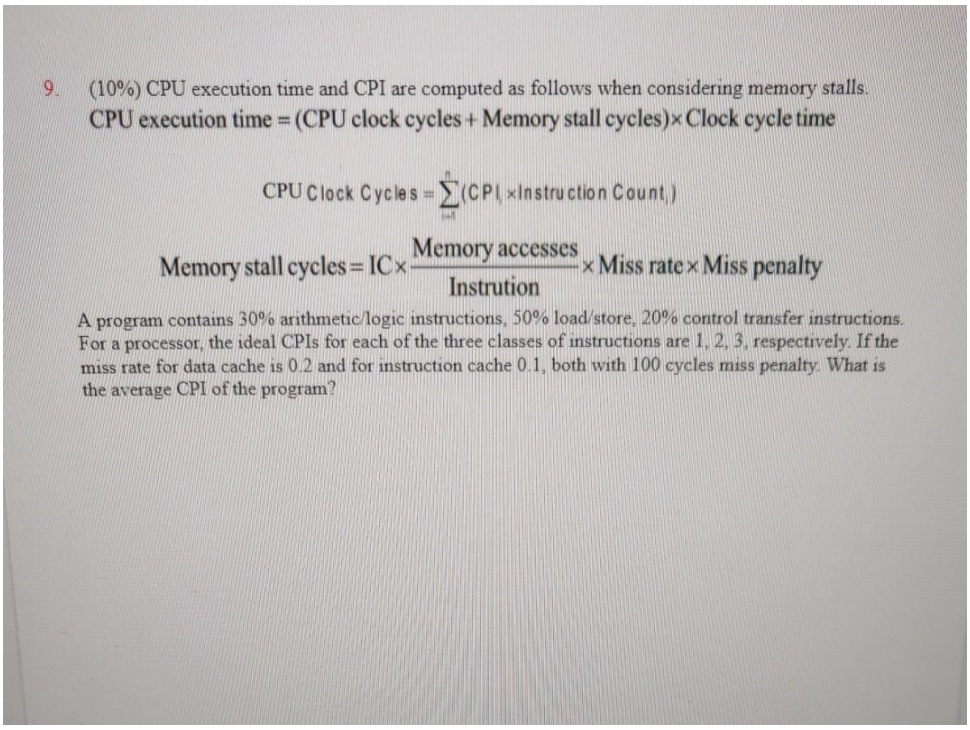
Complete the following questions on CPU performance and show all of your work. For longer decimal results, round to three decimal places.

1. Suppose you wish to run a program P with $5 \times 10^9$ instructions on a 3 GHz machine with a CPI of 2.3. What is the expected CPU time?
2. Suppose you have the following instruction mix. What is the CPI?

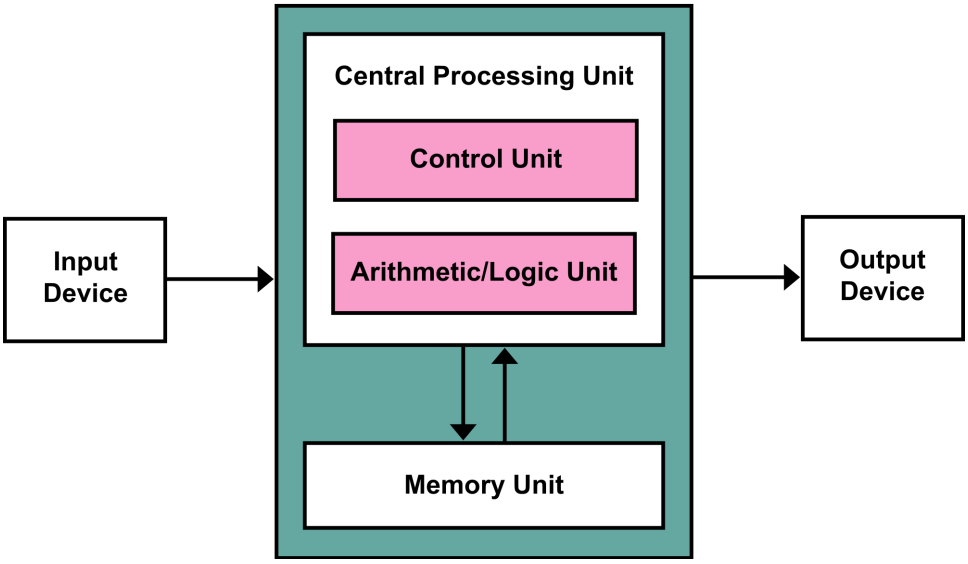
| Instruction Type | Frequency | Cycles |
|---|---|---|
| Memory | 22 | 2 |
| ALU | 40 | 1 |
| Branch | 3 | 2 |
| Jump | 8 | 1 |

3. Consider two different processors P1 and P2 executing the same instruction set with the clock rates and CPIs given in the following table.
    1. If the processors each execute a program in 15 seconds, find the number of cycles and the number of instructions for each processor.
    2. We are trying to reduce the program execution time by 20%, but this leads to an increase of 25% in the CPI. What clock rate should we have to get this time reduction for each processor?

| Processor | Clock Rate | CPI |
|---|---|---|
| P1 | 4 GHz | 2 |
| P2 | 4.5 GHz | 1.5 |

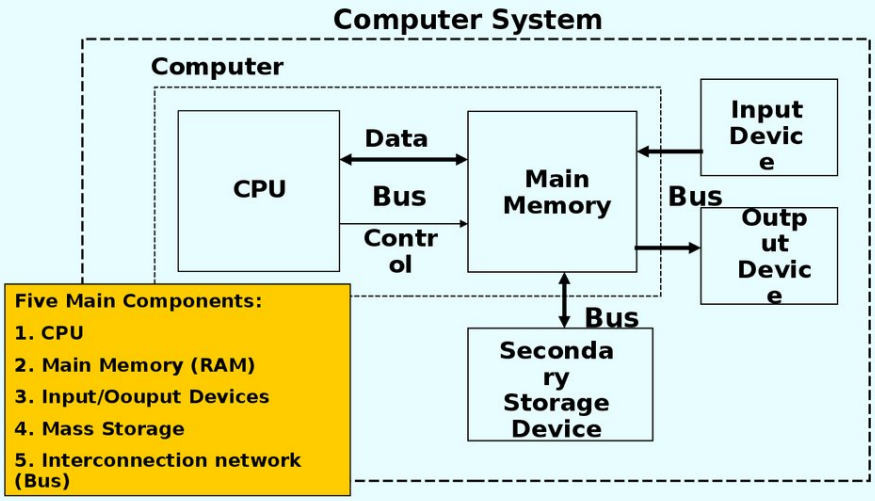


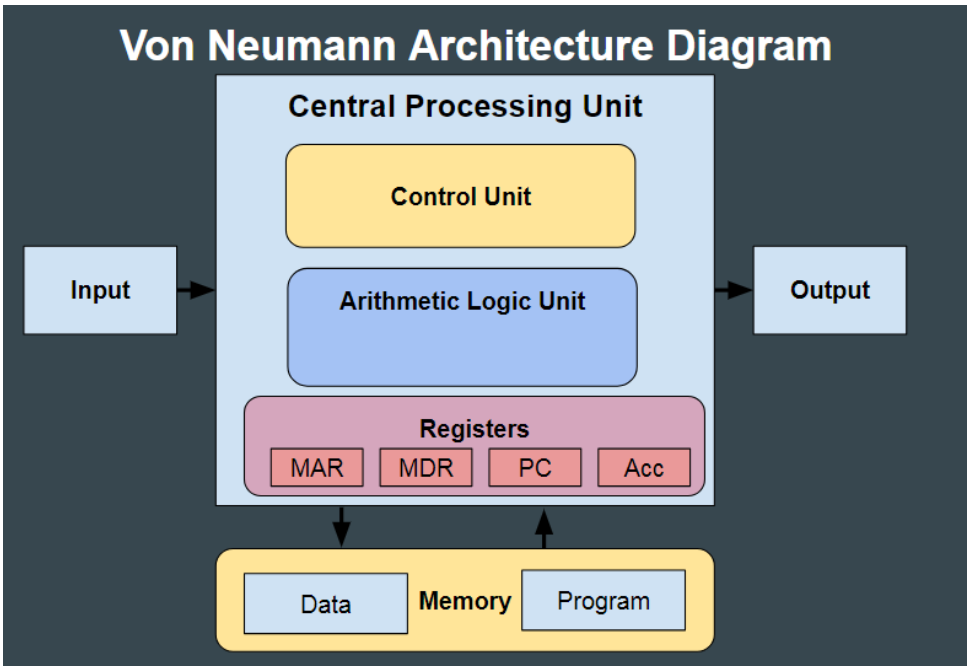

# Processor - Von Neumann Architecture

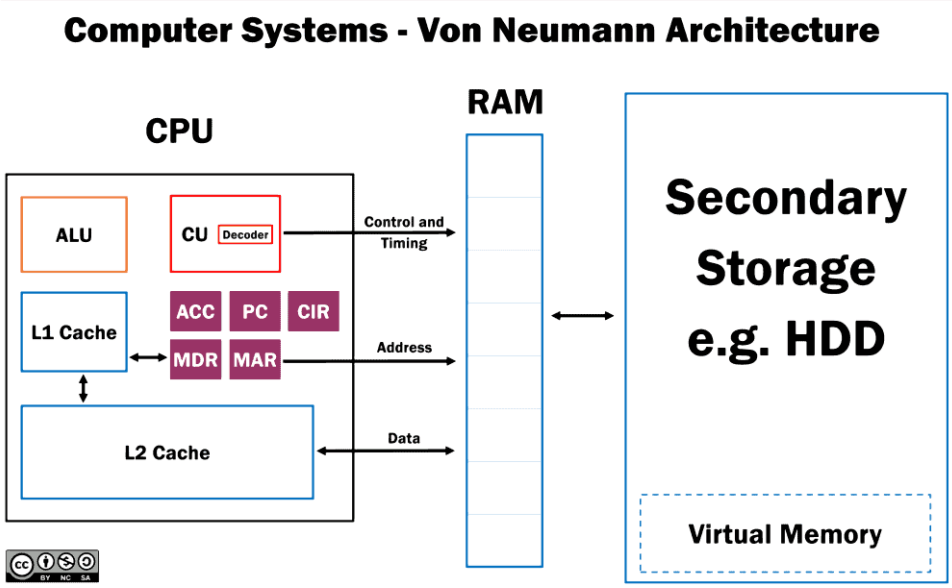


# von Neumann Architecture

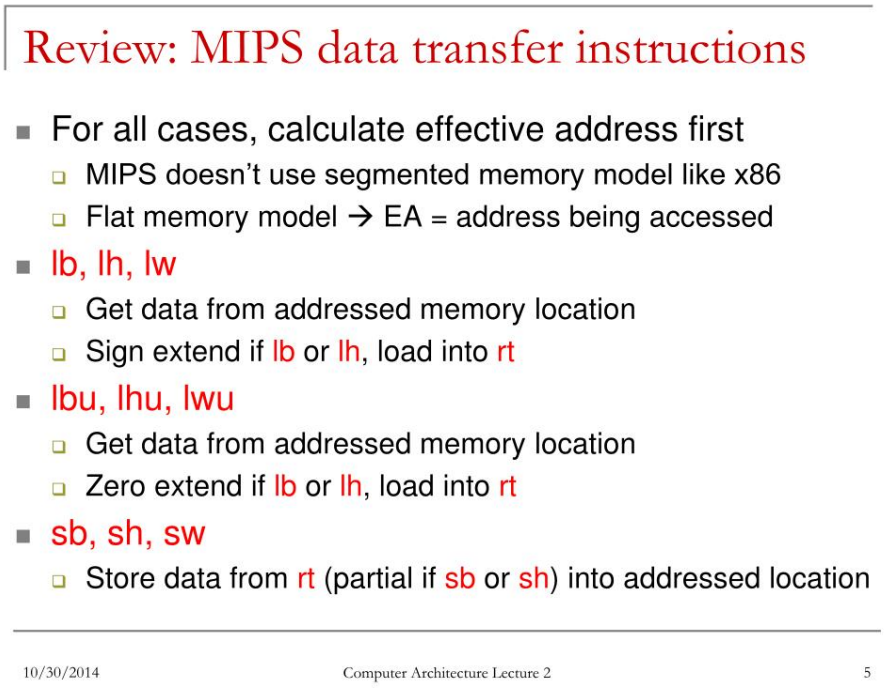
- A more complete view of the computer *system* architecture that integrates interaction (human or otherwise) consists of:





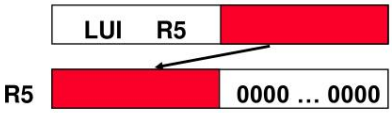

# Processer - Datapath



# MIPS Data Transfer Instructions

| Instruction | Comment |
|---|---|
| SW R3, 500(R4) | Store word |
| SH R3, 502(R2) | Store half |
| SB R2, 41(R3) | Store byte |
| LW R1, 30(R2) | Load word |
| LH R1, 40(R3) | Load half word |
| LHU R1, 40(R3) | Load half word unsigned |
| LB R1, 40(R3) | Load byte |
| LBU R1, 40(R3) | Load byte unsigned |
| LUI R1, 40 | Load Upper Immediate (16 bits shifted left by 16) |

**Why do we need LUI?**

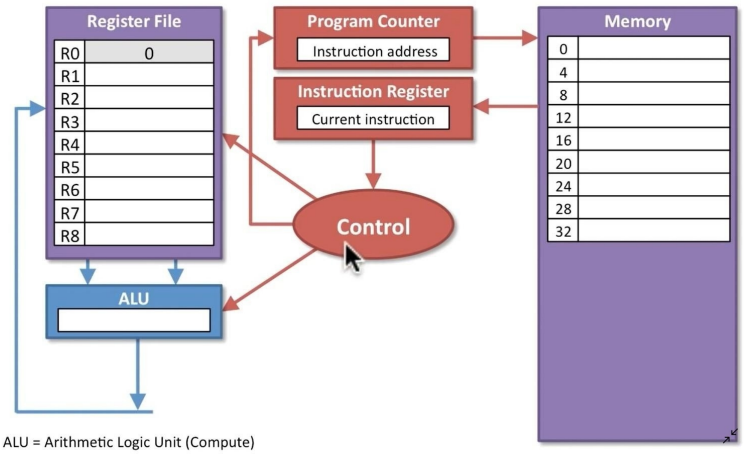



## Data operations in detail

1. Data Operations

26

1. Program Counter holds the instruction address.
2. Instructions are *fetched* from memory into the Instruction Register.
3. Control logic *decodes* the instruction and tells the ALU and Register File what to do.
4. ALU *executes* the instruction and results flow back to the Register File.
5. The Control logic *updates* the Program Counter for the next instruction.



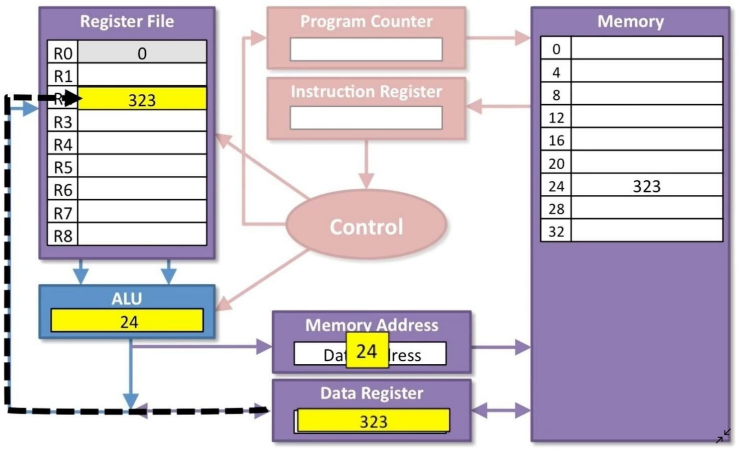

ALU = Arithmetic Logic Unit (Compute)

## Data transfers in detail

2. Data Transfers

32

1. ALU *generates* address
2. Address goes to the Memory Address Register
3. Results to/from memory are stored in the Memory Data Register
4. Data from memory can now be stored back into the Register File or to memory can be written.

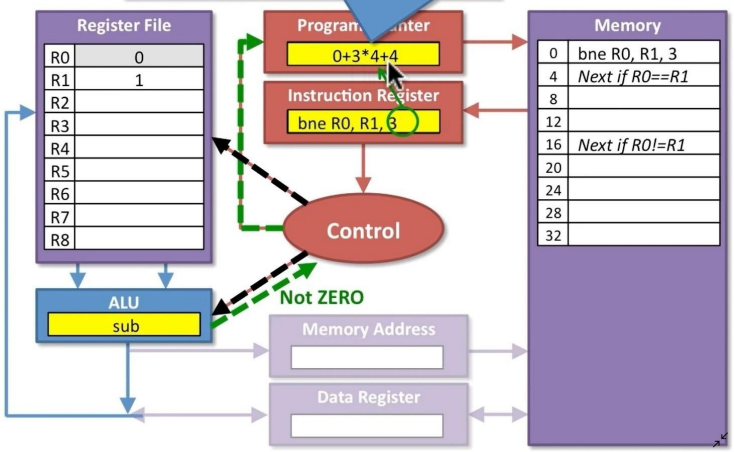


## Sequencing in detail



The label constant is in instruction **words**, so it needs to be multiplied by 4 to convert to **byte address**.

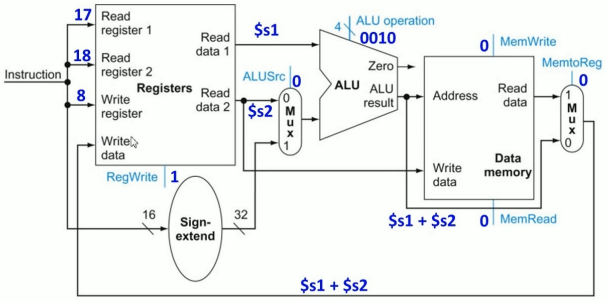
1. ALU *compares* registers
2. Result tells the Control whether to branch
3. If the branch is *taken*, then the Control **adds** a constant from the instruction to the Program Counter
4. The Control always adds 4 to the Program Counter




## *Example: Building a datapath for R − type and memory instructions*

```
add $t0, $s1, $s2
```

| R-type | 0 | $s1 (17) | $s2 (18) | $ t0(8) | Shamt (0) | Funct (32) |
|---|---|---|---|---|---|---|
| | 31:26 | 25:21 | 20:16 | 15:11 | 10:6 | 5:0 |




| Name | Number |
|---|---|
| $t0 - $t7 | 8-15 |
| $s0 - $s7 | 16-23 |

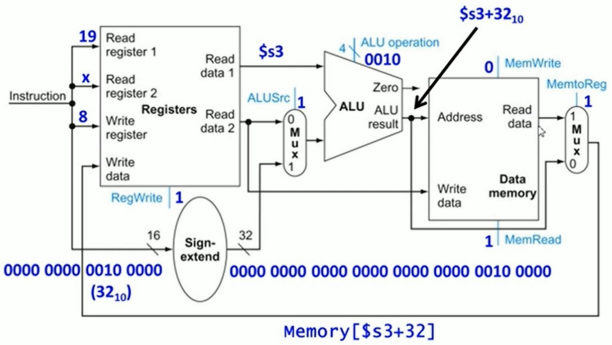
| ALU control | Function |
|---|---|
| 0000 | AND |
| 0001 | OR |
| 0010 | add |
| 0110 | subtract |
| 0111 | set-on-less-than |
| 1100 | NOR |

課程:計算機組織 / 老師:朱宗賢

## *Example: Building a datapath for R − type and memory instructions*

```
lw  $t0, 32($s3)
```

| Load | 35 | $s3 (19) | $t0 (8) | Address (32) |
|---|---|---|---|---|
| | 31:26 | 25:21 | 20:16 | 15:0 |




| Name | Number |
|---|---|
| $t0 - $t7 | 8-15 |
| $s0 - $s7 | 16-23 |

| ALU control | Function |
|---|---|
| 0000 | AND |
| 0001 | OR |
| 0010 | add |
| 0110 | subtract |
| 0111 | set-on-less-than |
| 1100 | NOR |

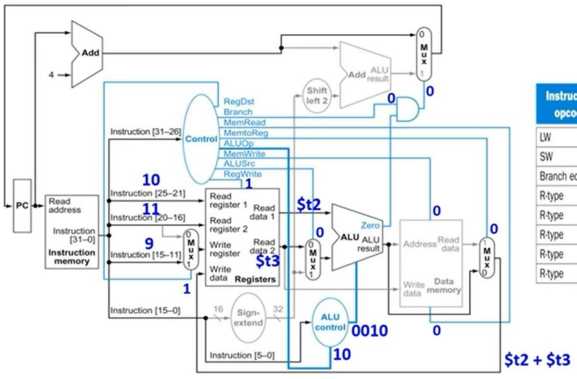
課程:計算機組織 / 老師:朱宗賢

## *The datapath in operation for an R – type instruction*

課程:計算機組織 / 老師:朱宗賢

```
add $t1, $t2, $t3  # $t1 = $t2 + $t3
```

| R-type | 0 | $t2 (10) | $t3 (11) | $ t1 (9) | Shamt (0) | Funct (32) |
|---|---|---|---|---|---|---|
| | 31:26 | 25:21 | 20:16 | 15:11 | 10:6 | 5:0 |




- R-type instructions are used when all the data values used by the instruction are located in registers. Ex: add, sub, or and sll
- Instruction fetch -> register read -> ALU operation -> register write

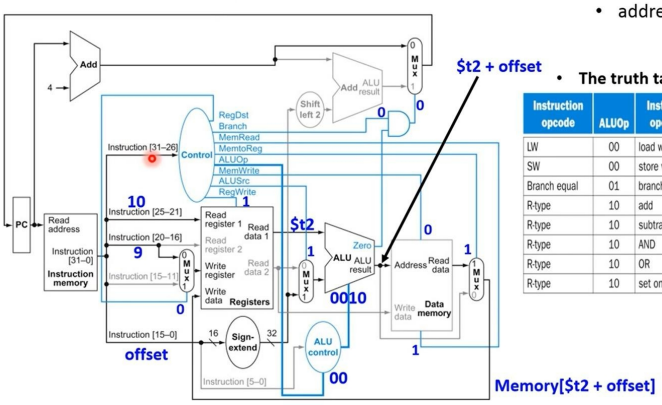
| Instruction opcode | ALUOp | Instruction operation | Funct field | Desired ALU action | ALU control input |
|---|---|---|---|---|---|
| LW | 00 | load word | XXXXXX | add | 0010 |
| SW | 00 | store word | XXXXXX | add | 0010 |
| Branch equal | 01 | branch equal | XXXXXX | subtract | 0110 |
| R-type | 10 | add | 100000 | add | 0010 |
| R-type | 10 | subtract | 100010 | subtract | 0110 |
| R-type | 10 | AND | 100100 | AND | 0000 |
| R-type | 10 | OR | 100101 | OR | 0001 |
| R-type | 10 | set on less than | 101010 | set on less than | 0111 |

## *The datapath in operation for a load instruction*

課程:計算機組織 / 老師:朱宗賢

```
lw $t1, offset($t2)  # $t1 = Memory[$t2 + offset]
```

| 0x23 | $t2 (10) | $t1 (9) | offset |
|---|---|---|---|
| 31:26 | 25:21 | 20:16 | 15:0 |




- instruction fetch -> register read -> ALU operation -> data access -> register write
- address line, control line, data line
- **The truth table for the 4 ALU control bits**

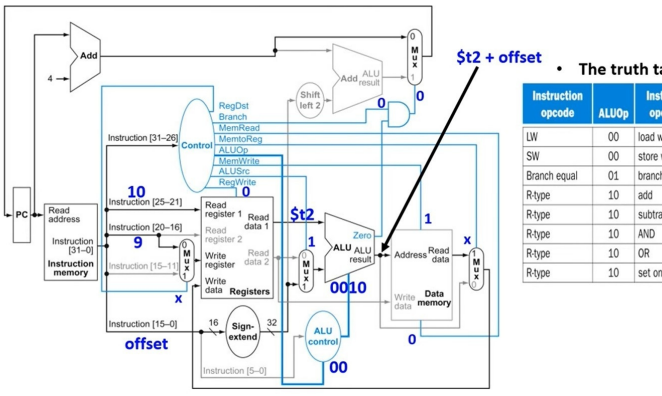
| Instruction opcode | ALUOp | Instruction operation | Funct field | Desired ALU action | ALU control input |
|---|---|---|---|---|---|
| LW | 00 | load word | XXXXXX | add | 0010 |
| SW | 00 | store word | XXXXXX | add | 0010 |
| Branch equal | 01 | branch equal | XXXXXX | subtract | 0110 |
| R-type | 10 | add | 100000 | add | 0010 |
| R-type | 10 | subtract | 100010 | subtract | 0110 |
| R-type | 10 | AND | 100100 | AND | 0000 |
| R-type | 10 | OR | 100101 | OR | 0001 |
| R-type | 10 | set on less than | 101010 | set on less than | 0111 |

## *The datapath in operation for a store instruction*

課程:計算機組織 / 老師:朱宗賢

```
sw $t1, offset($t2)  # Memory[$t2 + offset] = $t1
```

| 0x2b | $t2 (10) | $t1 (9) | offset |
|---|---|---|---|
| 31:26 | 25:21 | 20:16 | 15:0 |




- instruction fetch -> register read -> ALU operation -> data access
- **The truth table for the 4 ALU control bits**

| Instruction opcode | ALUOp | Instruction operation | Funct field | Desired ALU action | ALU control input |
|---|---|---|---|---|---|
| LW | 00 | load word | XXXXXX | add | 0010 |
| SW | 00 | store word | XXXXXX | add | 0010 |
| Branch equal | 01 | branch equal | XXXXXX | subtract | 0110 |
| R-type | 10 | add | 100000 | add | 0010 |
| R-type | 10 | subtract | 100010 | subtract | 0110 |
| R-type | 10 | AND | 100100 | AND | 0000 |
| R-type | 10 | OR | 100101 | OR | 0001 |
| R-type | 10 | set on less than | 101010 | set on less than | 0111 |

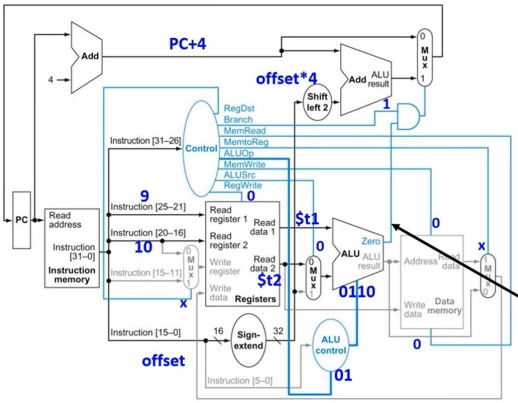
*The datapath in operation for a branch – on – equal instruction*

課程:計算機組織 / 老師:朱宗賢

```
beq $t1, $t2, offset  # if ($t1==$t2) goto PC+4+offset*4
```

- instruction fetch -> register read -> ALU operation -> decide which adder result to store in to the PC

| 4 | $t1 (9) | $t2 (10) | offset |
|---|---|---|---|
| 31:26 | 25:21 | 20:16 | 15:0 |




- **The truth table for the 4 ALU control bits**

| Instruction opcode | ALUOp | Instruction operation | Funct field | Desired ALU action | ALU control input |
|---|---|---|---|---|---|
| LW | 00 | load word | XXXXXX | add | 0010 |
| SW | 00 | store word | XXXXXX | add | 0010 |
| Branch equal | 01 | branch equal | XXXXXX | subtract | 0110 |
| R-type | 10 | add | 100000 | add | 0010 |
| R-type | 10 | subtract | 100010 | subtract | 0110 |
| R-type | 10 | AND | 100100 | AND | 0000 |
| R-type | 10 | OR | 100101 | OR | 0001 |
| R-type | 10 | set on less than | 101010 | set on less than | 0111 |

**The ALU's Zero output is set if ($t1-$t2) is 0.**

*Example: Implementing Jumps*

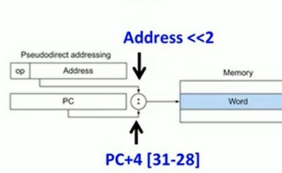
課程:計算機組織 / 老師:朱宗賢

```
j  10000  # go to location 10000h (word address)
```

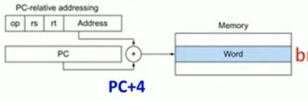
| J-type | 2 | 10000 |
|---|---|---|
| | 31:26 | 25:0 |

**PC+4 [31-28]** | **40000**

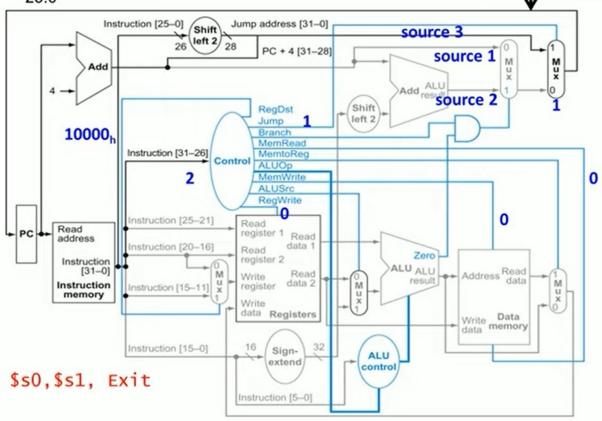
31:28 27:0




- **The MIPS jump instruction only replaces the lower 28bit of PC+4**
- **The sources of new PC:**
  1. **Incremented PC (PC+4)**
  2. **The branch target PC**
  3. **The jump target PC**




bne $s0,$s1, Exit




*Showing Branch Offset in Machine Language*

國立雲林科技大學資訊工程系 朱宗賢老師

| bne | op | rs | rt | address |
|---|---|---|---|---|
| | 6 bits | 5 bits | 5 bits | 16 bits |

**PC-relative addressing**

- Target address = PC+ 4 + address × 4
- Range = $2^{16}$ words = $2^{18}$ bytes = 256 Kb
- PC+ 4 + ($2^{15}$ -1) × 4 ~ PC+ 4 – ($2^{15}$) × 4

| j | op | address |
|---|---|---|
| | 6 bits | 26 bits |

**Pseudo-direct Addressing**

- Target address = $PC_{31...28}$ : (address × 4)
- Range = $2^{26}$ words = $2^{28}$ bytes = 256Mb
- (PC & 0xF0000000) | 0x00000000
  ~ (PC & 0xF0000000) | 0x0FFFFFFC

Assume Loop at location 80000

```
Loop: sll  $t1, $s3, 2
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1
      j    Loop
Exit: …
```

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 80000 | 0 | 0 | 19 | 9 | 4 | 0 |
| 80004 | 0 | 9 | 22 | 9 | 0 | 32 |
| 80008 | 35 | 9 | 8 | 0 | | |
| 80012 | 5 | 8 | 21 | 2 | | |
| 80016 | 8 | 19 | 19 | 1 | | |
| 80020 | 2 | 20000 | | | | |
| 80024 | | | | | | |

- Note: **jr $ra** #jump to the address stored in $ra, range = 4G bytes
- 80024 = PC + 4 + address × 4
  = 80012 + 4 + 2 × 4
- 80000 = $PC_{31...28}$ : (address × 4)
  = $80020_{31...28}$ :(address × 4)
  = 0 : 2000 × 4

# Processor - Pipeline

# Pipelining Lessons

- Pipelining doesn't help latency (execution time) of single task, it helps throughput of entire workload
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number of pipe stages
- Time to "fill" pipeline and time to "drain" it reduces speedup

- Pipeline rate limited by slowest pipeline stage
- Unbalanced lengths of pipe stages also reduces speedup

## The Fetch-Execute Cycle

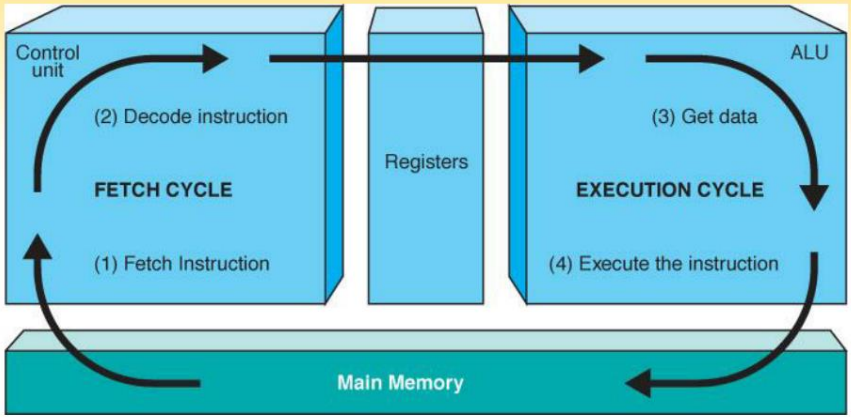
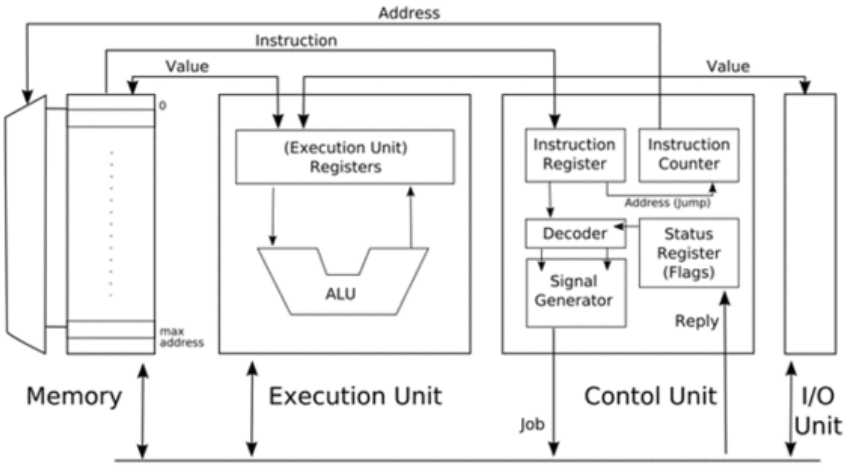
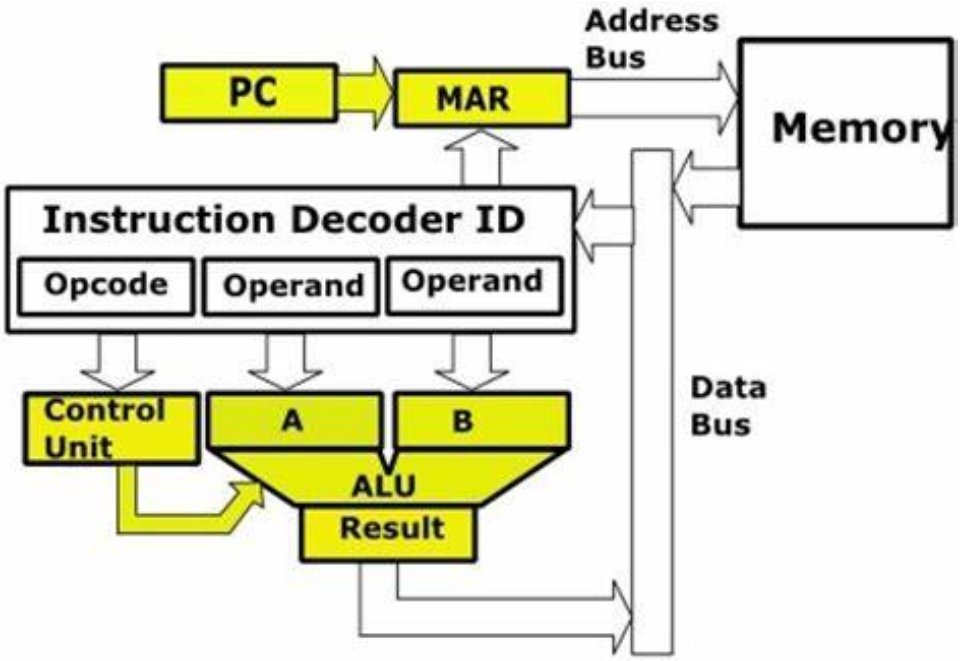


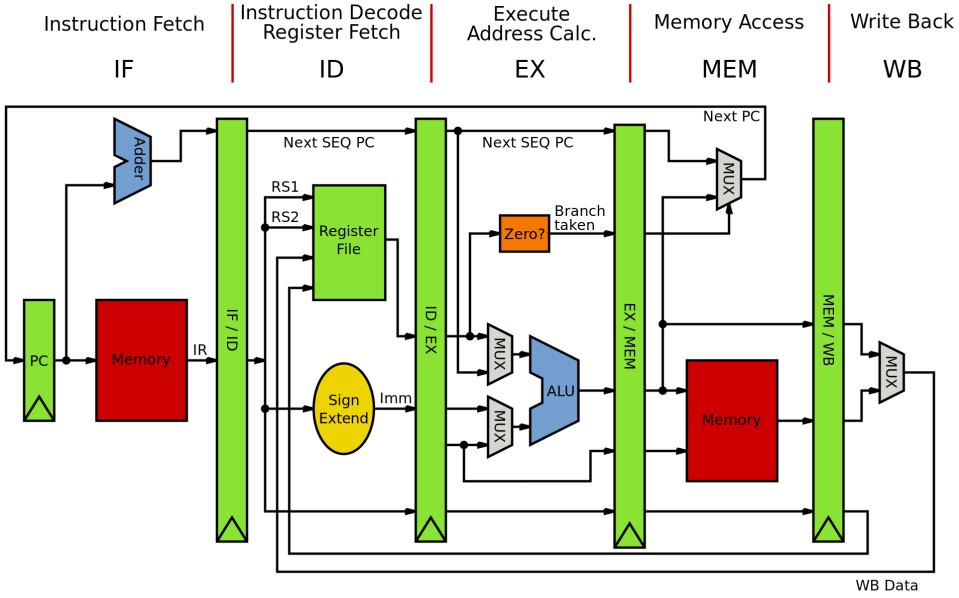

Figure 5.3 The Fetch-Execute Cycle

17





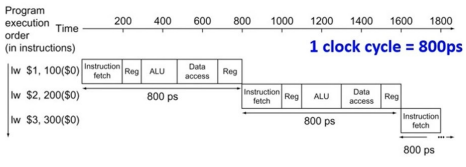


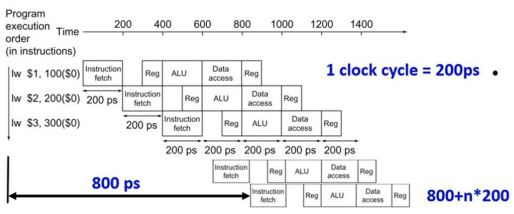




*Example: Single – Cycle versus Pipelined Performance*

課程:計算機組織 / 老師:朱宗賢

| Instruction class | Instruction fetch | Register read | ALU operation | Data access | Register write | Total time |
|---|---|---|---|---|---|---|
| Load word (lw) | 200 ps | 100 ps | 200 ps | 200 ps | 100 ps | 800 ps |
| Store word (sw) | 200 ps | 100 ps | 200 ps | 200 ps | | 700 ps |
| R-format (add, sub, AND, OR, slt) | 200 ps | 100 ps | 200 ps | | 100 ps | 600 ps |
| Branch (beq) | 200 ps | 100 ps | 200 ps | | | 500 ps |



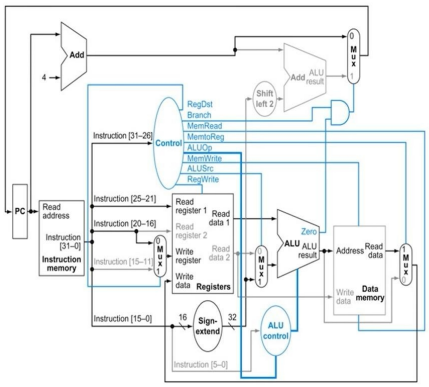




- Single-cycle: each instruction takes one clock cycle
- Piplelined: each stage takes one clock cycle

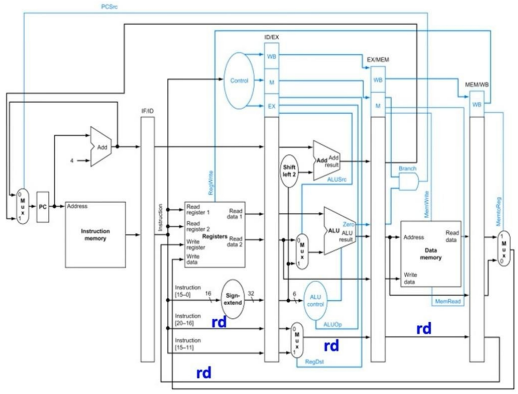
- 3 load instructions
  - The total execution time in nonpiplelined design is **2400 ps**
  - The total execution time in piplelined design is **1400 ps**
- 1,000,003 load instructions
  - The total execution time in nonpiplelined design is **800 ps x 1,000,003 = 800,002,400 ps**
  - The total execution time in piplelined design is **800 ps + 200 ps x 1,000,003 = 200,001,400 ps** **800,002,400 / 200,001,400 = ~ 4**
- IF Register read = 200ps, Register write = 200ps
  - The total execution time in nonpiplelined design is **1000 ps x 1,000,003 = 1,000,003,000 ps**
  - The total execution time in piplelined design is **800 ps + 200 ps x 1,000,003 = 200,001,400 ps** **1,000,003,000 / 200,001,400 = ~ 5**

*Control signals of the pipelined datapath*

課程:計算機組織 / 老師:朱宗賢



***Non-pipelined data path & control***



***Pipelined data path & control***

- Using pipeline registers to propagate results and control signals forward

# Pipeline Hazard

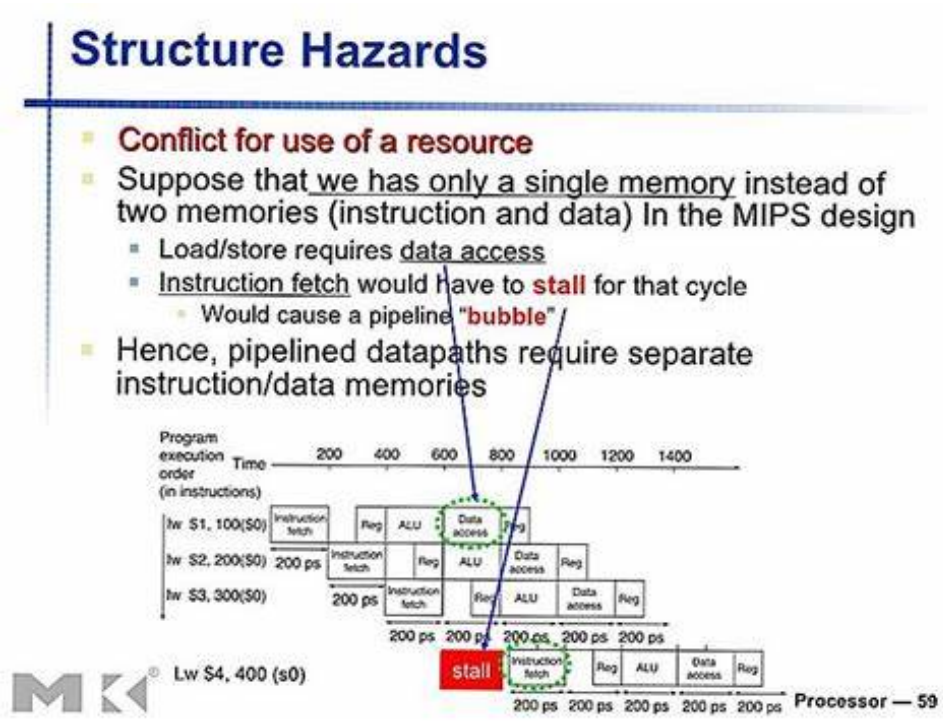
## Pipeline Hazards (1)

- ***Pipeline Hazards*** are situations that prevent the next instruction in the instruction stream from executing in its designated clock cycle
- Hazards reduce the performance from the ideal speedup gained by pipelining
- Three types of hazards
  - ***Structural hazards***
    - Arise from resource conflicts when the hardware can't support all possible combinations of overlapping instructions
  - ***Data hazards***
    - Arise when an instruction depends on the results of a previous instruction in a way that is exposed by overlapping of instruction in pipeline
  - ***Control hazards***
    - Arise from the pipelining of branches and other instructions that change the PC (Program Counter)

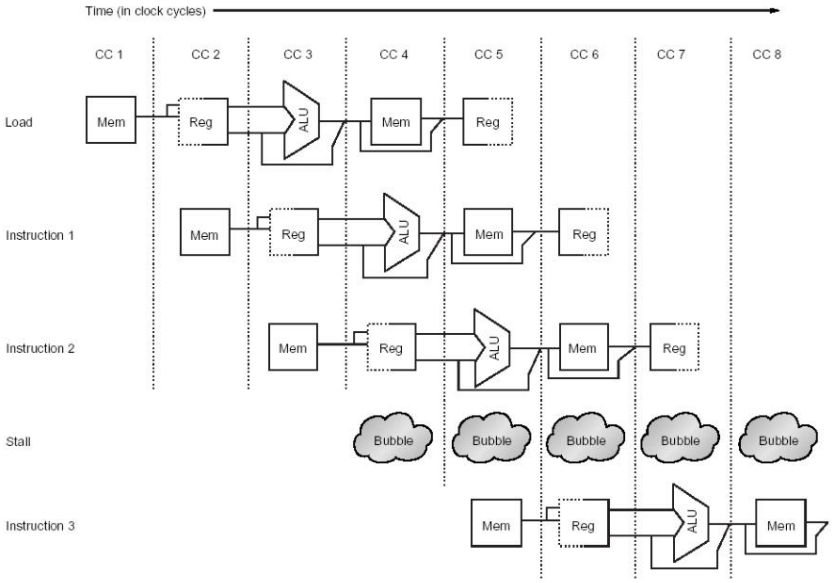
# Pipeline Hazards (2)

- Hazards in pipeline can make the pipeline to ***stall***
- Eliminating a hazard often requires that some instructions in the pipeline to be allowed to proceed while others are delayed
  - When an instruction is stalled, instructions issued *latter* than the stalled instruction are stopped, while the ones issued *earlier* must continue
- No new instructions are fetched during the stall

## Solution for Pipeline Hazards



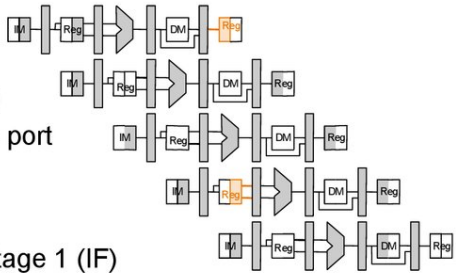
# Structural Hazards (3)




- Stall cycle added (commonly called pipeline *bubble*)

## Structural Hazard

- **Different instructions using the same resource at the same time**
- **Register File**:
  - Accessed in 2 stages:
    - Read during stage 2 (ID)
    - Write during stage 5 (WB)
  - Solution: 2 read ports, 1 write port
- **Memory**
  - Accessed in 2 stages:
    - Instruction Fetch during stage 1 (IF)
    - Data read/write during stage 4 (MEM)
  - Solution: separate instruction cache and data cache
- **Each functional unit can only be used once per instruction**
- **Each functional unit must be used at the same stage for all instructions**

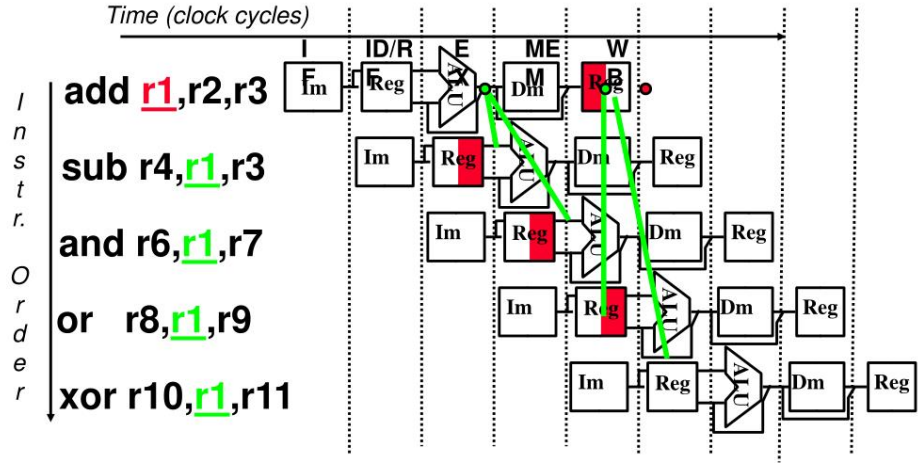



7 Computer Structure 2015 – Pipeline

# Pipline - Data Hazard

## Data Hazard Solution:

- "Forward" result from one stage to another




- "or" OK if define read/write properly

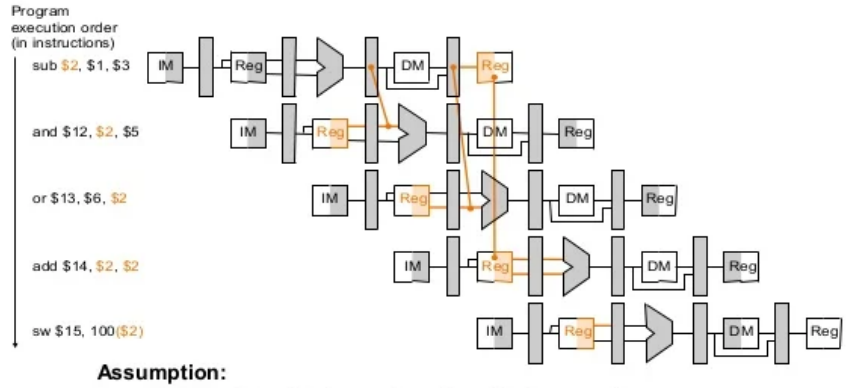


## Data Hazard Solution: Forwarding

- Key idea: connect data internally before it's stored

Time (in clock cycles)

| | CC 1 | CC 2 | CC 3 | CC 4 | CC 5 | CC 6 | CC 7 | CC 8 | CC 9 |
|---|---|---|---|---|---|---|---|---|---|
| Value of register $2 : | 10 | 10 | 10 | 10 | 10/–20 | –20 | –20 | –20 | –20 |
| Value of EX/MEM : | X | X | X | –20 | X | X | X | X | X |
| Value of MEM/WB : | X | X | X | X | –20 | X | X | X | X |



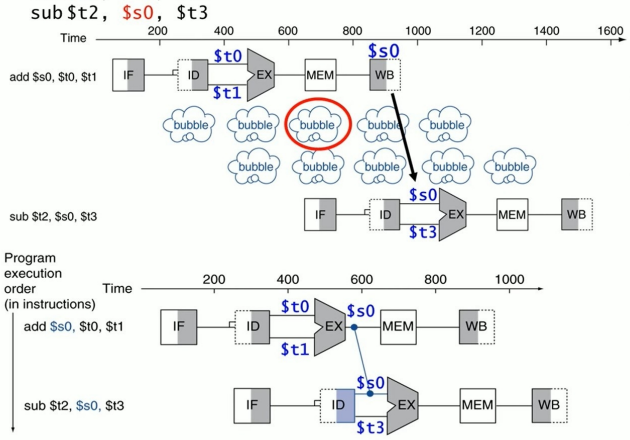

Assumption:

- The register file forwards values that are read and written during the same cycle.

## *Example: Forwarding with Two instructions*

課程:計算機組織 / 老師:朱宗賢

```
add $s0, $t0, $t1
sub $t2, $s0, $t3
```







***Forwarding (or bypassig)***: retrieving the missing data element from internal buffer rather than waiting for it to arrive from registers or memory.

## *Example: Reordering Code to Avoid Pipeline Stalls*

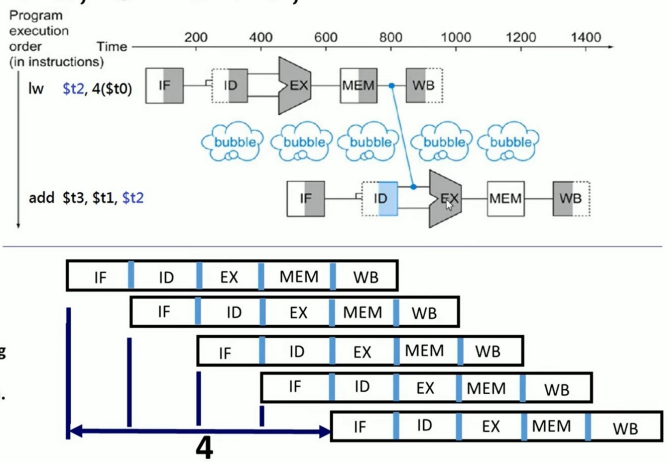
課程:計算機組織 / 老師:朱宗賢

C code for A = B + E; C = B + F;

```
lw  $t1, 0($t0)
lw  $t2, 4($t0)
add $t3, $t1, $t2
sw  $t3, 12($t0)
lw  $t4, 8($t0)
add $t5, $t1, $t4
sw  $t5, 16($t0)
```

13 cycles = 4 + 1x7 + 2

**We need a stall even with forwarding when an R-format instruction following a load tries to use the data.**




## *Example: Reordering Code to Avoid Pipeline Stalls*

課程:計算機組織 / 老師:朱宗賢

stall

stall

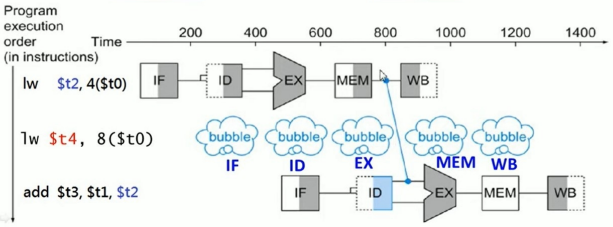
```
lw  $t1, 0($t0)
lw  $t2, 4($t0)
add $t3, $t1, $t2
sw  $t3, 12($t0)
lw  $t4, 8($t0)
add $t5, $t1, $t4
sw  $t5, 16($t0)
```

13 cycles

```
lw  $t1, 0($t0)
lw  $t2, 4($t0)
lw  $t4, 8($t0)
add $t3, $t1, $t2
sw  $t3, 12($t0)
add $t5, $t1, $t4
sw  $t5, 16($t0)
```

11 cycles

1. Instruction "lw $t4, 8($t0)" does not change the input of "add $t3, $t1, $t2" and "sw $t3, 12($t0)".
2. Instructions "add $t3, $t1, $t2" and "sw $t3, 12($t0)" do not change the input of "lw $t4, 8($t0)".

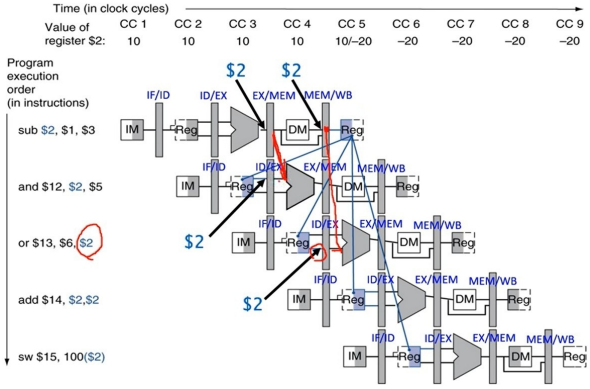


## *Dependence Detection*

課程:計算機組織 / 老師:朱宗賢

```
sub $2, $1,$3
and $12,$2,$5
or  $13,$6,$2
add $14,$2,$2
sw  $15,100($2)
```

1a, 2b




- Non-independent program
  - Detect hazards
  - Forward proper data
- Data hazards when
  - 1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
  - 1b. EX/MEM.RegisterRd = ID/EX.RegisterRt
  - 2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
  - 2b. MEM/WB.RegisterRd = ID/EX.RegisterRt
- But only if forwarding instruction will write to a register!
  - EX/MEM.RegWrite=1, MEM/WB.RegWrite=1
- And only if Rd for that instruction is not $zero
  - EX/MEM.RegisterRd ≠ 0,
  - MEM/WB.RegisterRd ≠ 0
- R0: Reserved, read as zero, ignore writes by software.

```
sll $0,$1,3
and $12,$0,$5
or  $13,$6,$0
add $14,$0,$0
sw  $15,100($0)
```

Independent program

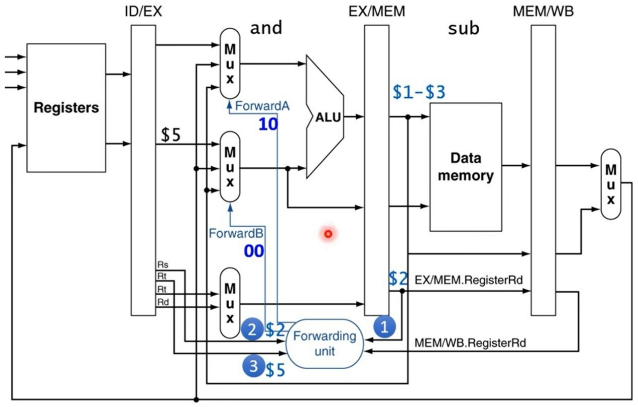
## *EX Data Hazard – conditions & control signals*

課程:計算機組織 / 老師:朱宗賢

- Conditions & control signals

(1) (2)
if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRs)) ForwardA = 10
if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRt)) ForwardB = 10

(3)




b. With forwarding — Clock cycle 4

```
sub $2, $1,$3
and $12,$2,$5
or  $13,$6,$2
add $14,$2,$2
sw  $15,100($2)
```

EX data hazard
ForwardA = 10

| Mux control | Source |
|---|---|
| ForwardA = 00 | ID/EX |
| ForwardA = 10 | EX/MEM |
| ForwardA = 01 | MEM/WB |
| ForwardB = 00 | ID/EX |
| ForwardB = 10 | EX/MEM |
| ForwardB = 01 | MEM/WB |

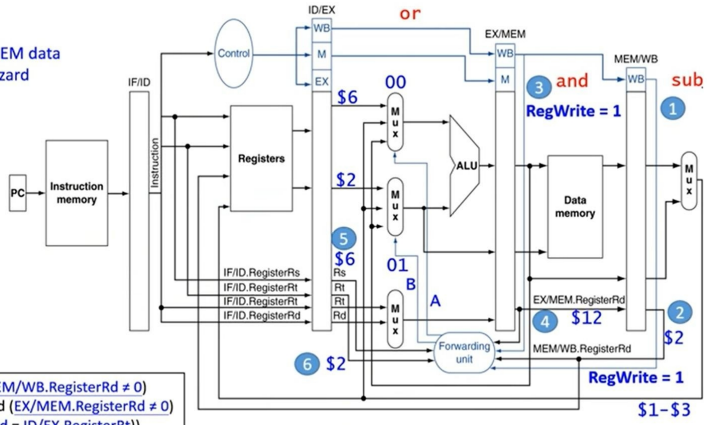
## *MEM Data Hazard – conditions & signals*

課程:計算機組織 / 老師:朱宗賢

```
sub $2, $1,$3
and $12,$2,$5
or  $13,$6,$2
add $14,$2,$2
sw  $15,100($2)
```

MEM data hazard

| Mux control | Source |
|---|---|
| ForwardA = 00 | ID/EX |
| ForwardA = 10 | EX/MEM |
| ForwardA = 01 | MEM/WB |
| ForwardB = 00 | ID/EX |
| ForwardB = 10 | EX/MEM |
| ForwardB = 01 | MEM/WB |




2. if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
   and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
   and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
   and (MEM/WB.RegisterRd = ID/EX.RegisterRt))

*MEM data hazard happens in clock cycle 5!*

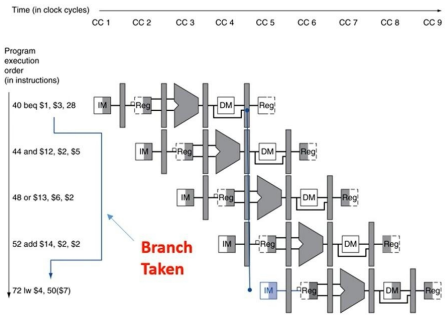
The image you are requesting does not exist or is no longer available.

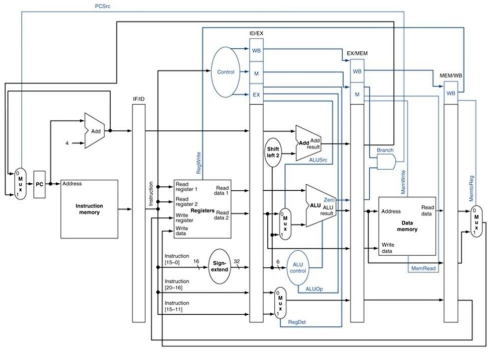
imgur.com

# Pipeline - Control Hazard

*Control Hazards – Performance Impact and Solutions Overview* 課程:計算機組織 / 老師:朱宗賢

- Control hazards: attempt to make branching decisions before branch condition is evaluated
  (1)Taken: PC = PC + 4 + Offset, (2)Not Taken: PC = PC + 4
- The decision about whether to branch doesn't occur until the MEM stage
- Braches have a penalty of 3 clock cycles!



## Summary - Control Hazard Solutions

- **Stall** - stop fetching instr. until result is available
  - Significant performance penalty
  - Hardware required to stall
- **Predict** - assume an outcome and continue fetching (undo if prediction is wrong)
  - Performance penalty only when guess wrong
  - Hardware required to "squash" instructions
- **Delayed branch** - specify in architecture that following instruction is always executed
  - Compiler re-orders instructions into delay slot
  - Insert "NOP" (no-op) operations when can't use (~50%)
  - This is how original MIPS worked

CSCE430/830 Pipeline Hazards

## Control Hazard Review

**The nub of the problem:**

- In what pipeline stage does the processor fetch the next instruction?
- If that instruction is a conditional branch, when does the processor know whether the conditional branch is taken (execute code at the target address) or not taken (execute the sequential code)?
- What is the difference in cycles between them?

**The cost of stalling until you know whether to branch**

- number of cycles in between * branch frequency = the contribution to CPI due to branches

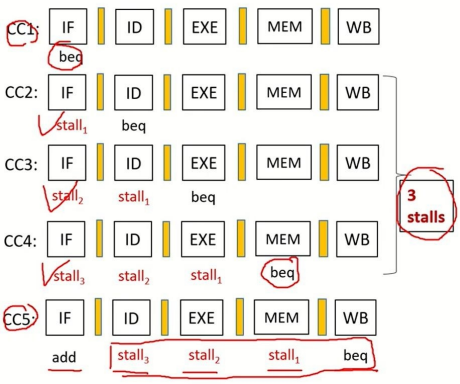
**Predict the branch outcome to avoid stalling**

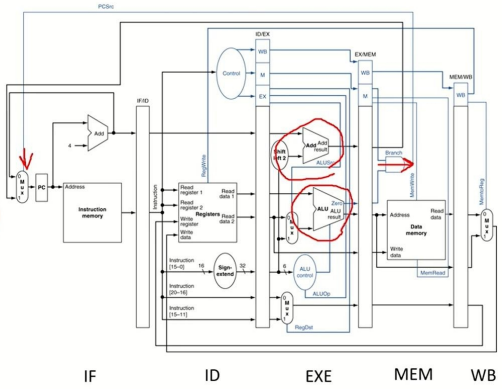
Spring 2003 CSE P548 1

## *Control Hazards – Reducing the Delay of Branches*

國立雲林科技大學資工系 朱宗賢老師「計算機組織」課程

- Control hazard: the flow of instruction addresses is not what the pipeline expect



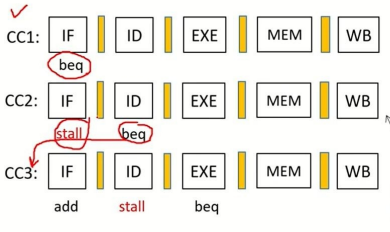



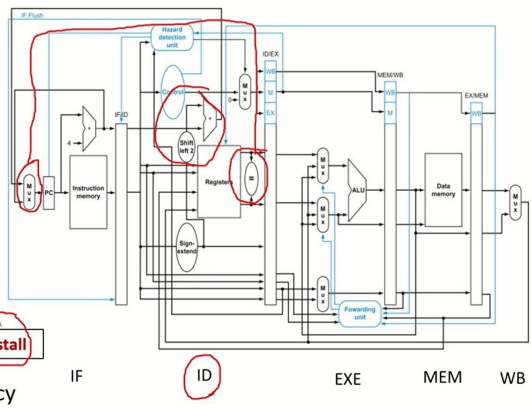

- Ideal CPI = 1, with 30% branch frequency
  -> Actual CPI = 1 + 0.3*3 = 1.9

## *Control Hazards – Reducing the Delay of Branches*

國立雲林科技大學資工系 朱宗賢老師「計算機組織」課程

- Move hardware to determine outcome from EXE to ID stage:
  (1) Target address adder
  (2) Register comparator
- One clock-cycle stall on branches.

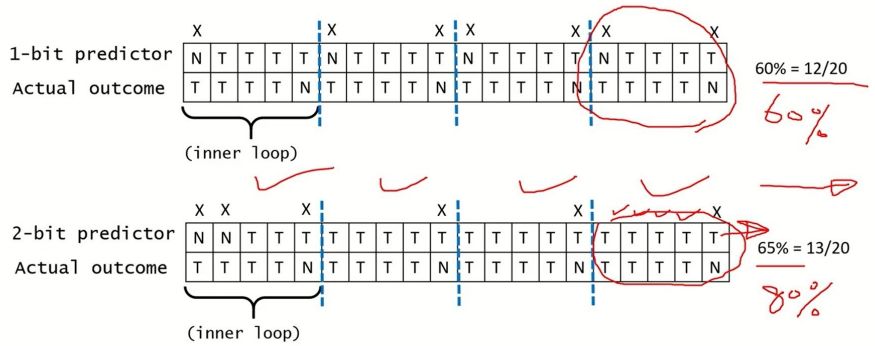



- Ideal CPI = 1, with 30% branch frequency
  -> Actual CPI = 1 + 0.3*1 = 1.3

*Control Hazard Solution – Dynamic Branch Prediction*

國立雲林科技大學資工系 朱宗賢老師「計算機組織」課程

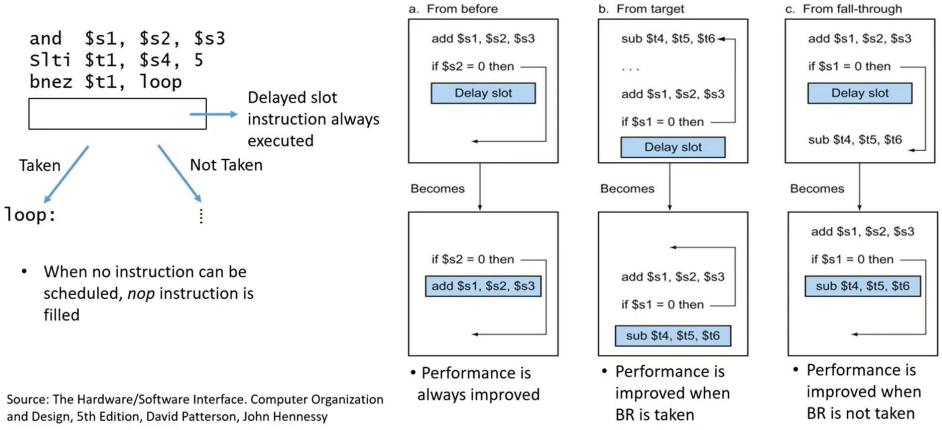



- 1-bit predictor: we change our mind if we mispredict once about the branch
- 2-bit predictor: we change our mind if we mispredict twice about the branch
  - Works well for loop-intensive programs

*Control Hazard Solution – Scheduling branch-delay slot*

國立雲林科技大學資工系 朱宗賢老師「計算機組織」課程

- Branch delay slot: the slot directly after a delayed branch instruction, which in the MIPS architecture is filled by an instruction that does not affect the branch



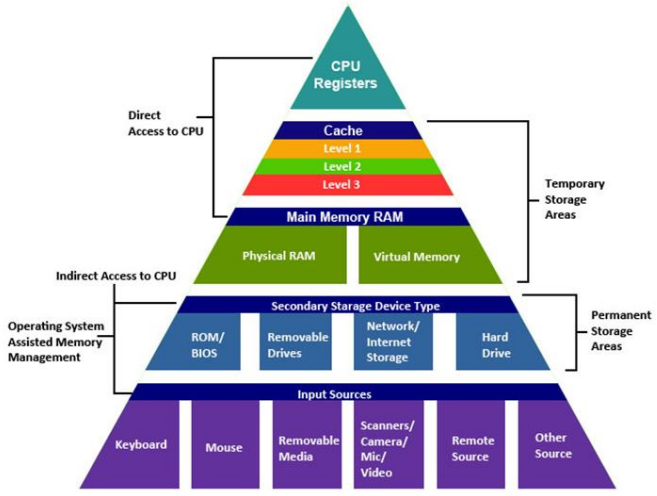

- When no instruction can be scheduled, *nop* instruction is filled

- Performance is always improved
- Performance is improved when BR is taken
- Performance is improved when BR is not taken

Source: The Hardware/Software Interface. Computer Organization and Design, 5th Edition, David Patterson, John Hennessy

# Mermory Hierarchy

## The Memory Hierarchy

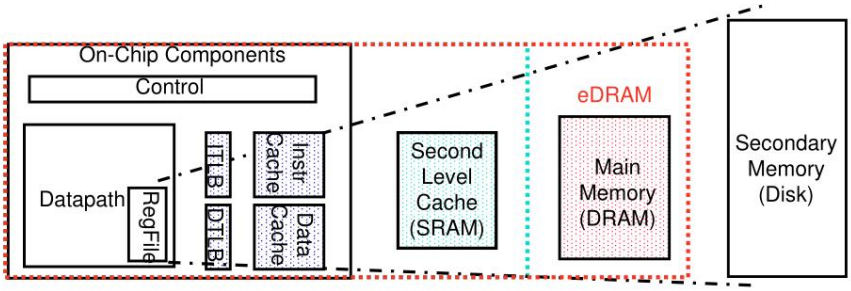



http://cse1.net/recaps/4-memory.html

2

## A Typical Memory Hierarchy

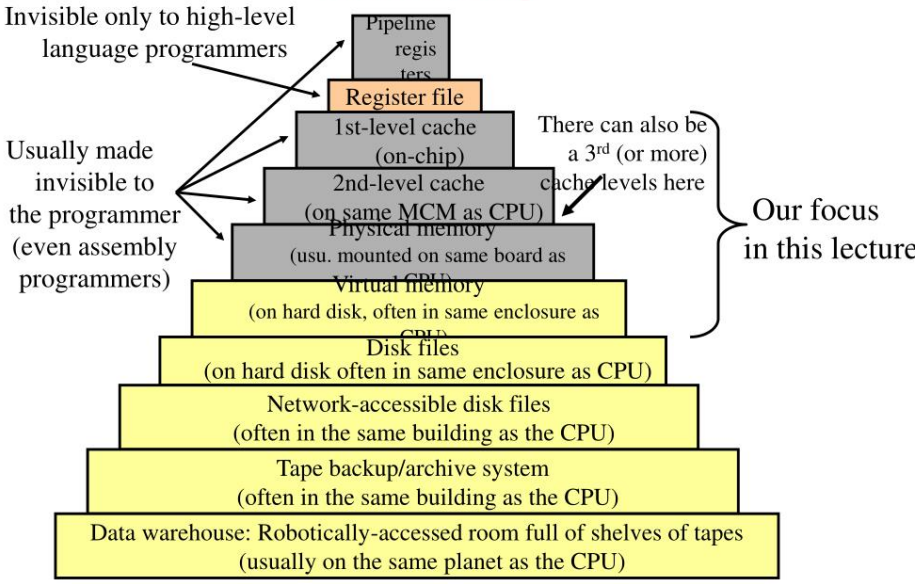
- By taking advantage of the principle of locality
  - Can present the user with as much memory as is available in the cheapest technology
  - at the speed offered by the fastest technology



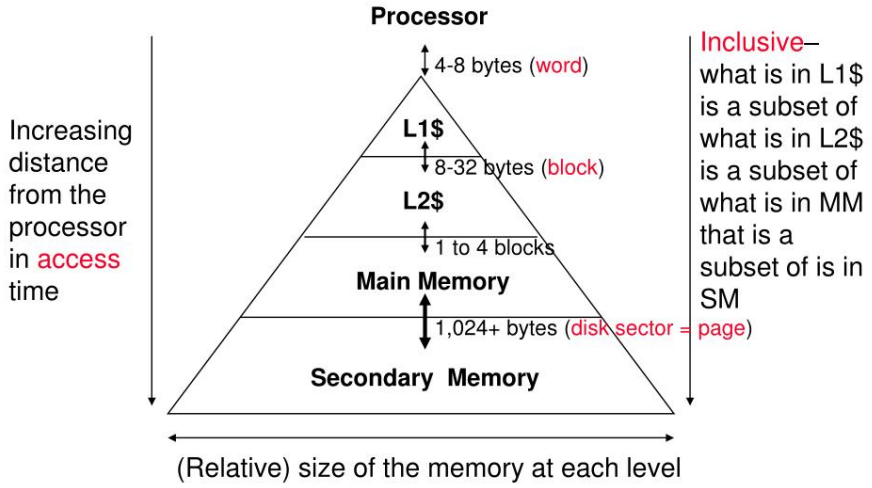

| | | | | | |
|---|---|---|---|---|---|
| **Speed (%cycles):** | ½'s | 1's | 10's | 100's | 1,000's |
| **Size (bytes):** | 100's | K's | 10K's | M's | G's to T's |
| **Cost:** | highest | | | | lowest |

CEG3420 L13.9 Qiang Xu CUHK, Spring 2011

## Many Levels in Memory Hierarchy



## Characteristics of the Memory Hierarchy

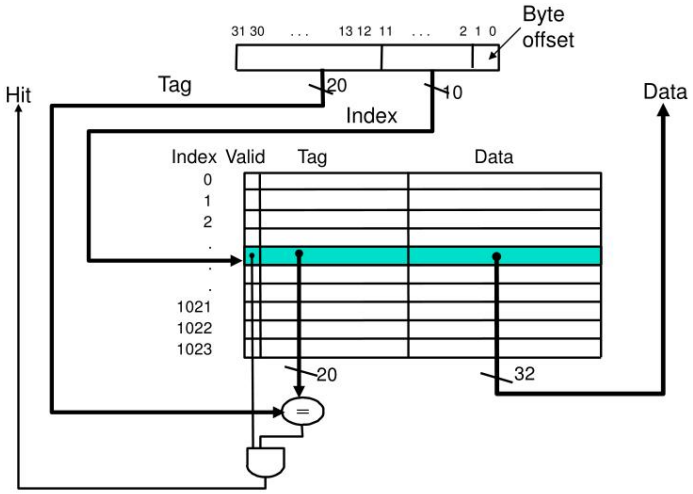



CPE432 Chapter 5A.7 Dr. W. Abu-Sufah, UJ

# Cache

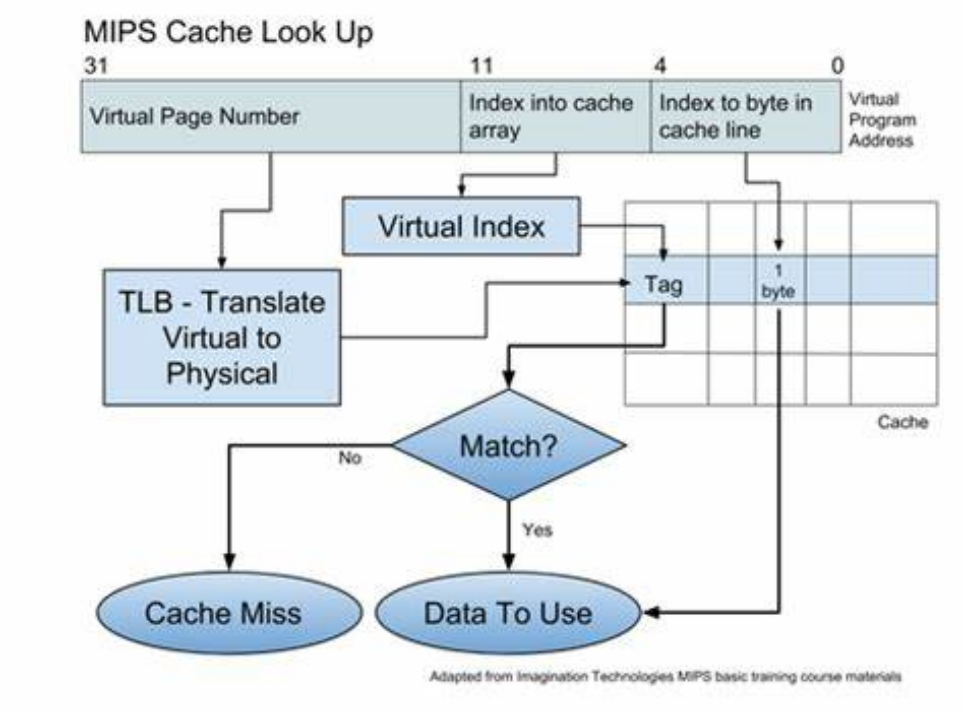
## MIPS Direct Mapped Cache Example

- One word/block, cache size = 1K words



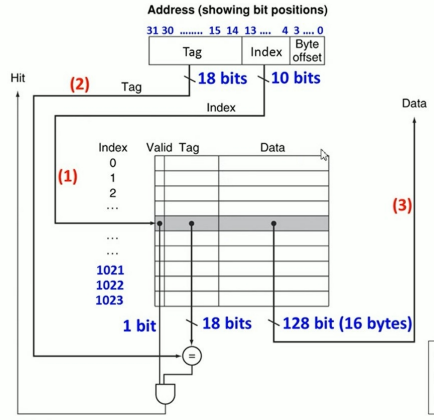

*What kind of locality are we taking advantage of?*

CEG3420 L13.34 Qiang Xu CUHK, Spring 2011




*Example*: *Bits in a Cache* 課程:計算機組織 / 老師:朱宗賢

A direct-mapped cache with 16 KiB of data and 4-word blocks, assuming a 32-bit address.
(unit : 1 KiB (kibibyte) = 1024 byte, 1 Kibit (Kibibit) = 1024 bit)

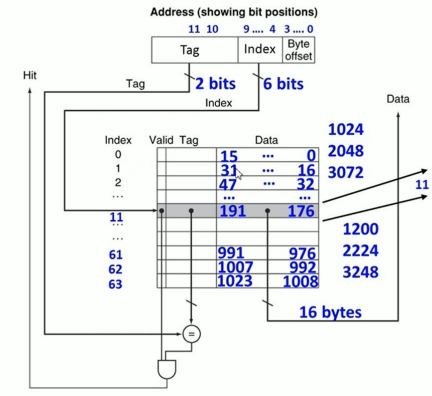



(1) Total size = f (Valid, Tag, Data, Index) = the size of (Valid + Tag + Data) * Index
(2) The size of the Tag field = The size of address – the size of Index field - the size of Byte offset field
(3) 1 bock = 4 words = 16 bytes
The size of the Byte offset field **= 4 bits**
(4) Data **= 16 KiB = 16 * 1024 bytes = 16384 bytes**
Index **= 16384 bytes / (4 * 4 bytes) = 1024**
The size of the Index field **= 10 bits**
(5) The size of the Tag field **= 32 – 4 – 10 = 18 bits**
The total number of bits in a direct-mapped cache
**= 1024 * (1+ 18 + 128) = 1024*147 = 147 Kibits**
(6) The ratio of total cache to data
**= 147 Kibits/ 16 KiB = 18.375 KiB / 16 KiB = 1.15**

Q1: 1-word block ? Byte offset = 2 bits, Index = 12 bits, Tag: 18 bits
Q2: 32 KiB? Byte offset = 4 bits, Index = 11 bits, Tag: 17 bits
Q2: 64-bit address? Byte offset = 4 bits, Index = 10 bits, Tag: 50 bits

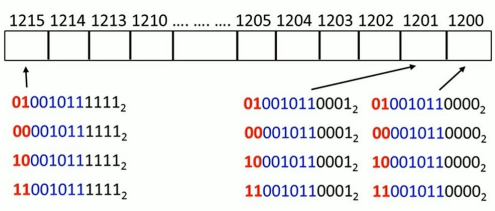
*Example*: *Mapping a Address to a Multiword Cache Block* 課程:計算機組織 / 老師:朱宗賢

Consider a direct-mapped cache with 64 blocks and a block size of 16 bytes. To what block number does byte address 1200 map?




Block number = 1200/16 = 75
75 mod 64 = 11

12 bit address
Address: $1200_{10}$ = 010010110000$_2$, Index: $001011_2 = 11_{10}$




# Cache Performance

*Example: Calculating Cache Performance*

課程:計算機組織 / 老師:朱宗賢

- Given
  - I-cache miss rate = 2%, D-cache miss rate = 4%, Miss penalty = 100 cycles
  - Load & stores are 36% of instructions (ex: lw, sw)
  - Base CPI (ideal cache) = 2
- Miss cycles per instruction
  - I-cache: 0.02 × 100 = 2 cycles, D-cache: 0.36 × 0.04 × 100 = 1.44 cycles
- Actual CPI
  - Base CPI + I-cache miss cycles + D-cache miss cycles = 2 + 2 + 1.44 = 5.44 cycles
- The amount of execution time spent on memory stalls
  - 3.44 / 5.44 = 63%
- Ideal CPU is 5.44/2 =2.72 times faster

Q: Base CPI = 1, Actual CPI = 4.44, memory stalls = 3.44 /4.44 = 77%, Ideal CPU is 4.44/1 = 4.44 times faster

# Cache Performance Equations

- **Memory stalls per program (blocking cache):**

$$MemoryStallCycle = IC \times (\frac{MemoryAccesses}{Instruction}) \times MissRate \times MissPenalty$$

$$MemoryStallCycle = IC \times (\frac{Misses}{Instruction}) \times MissPenalty$$

- **CPU time formula:**

$$CPU\ Time = IC \times (CPI_{Execu} + \frac{Memory\ Stall\ Cycle}{Instruction}) \times Cycle\ Time$$

- **More cache performance will be given later!**

46

# Cache Performance

- CPI $_{\text{contributed by cache}}$ = $CPI_c$
  = miss rate * number of cycles to handle the miss
- Another important metric
  Average memory access time = cache hit time * hit rate
  + Miss penalty * (1 - hit rate)

*Example: Calculating Average Memory Access Time*

課程:計算機組織 / 老師:朱宗賢

- Average memory access time (AMAT)
  AMAT = Hit time + Miss rate × Miss penalty
- Given:
  CPU with 1ns clock, hit time = 1 cycle, miss penalty = 20 cycles, miss rate = 5%
  AMAT = 1 + 0.05 × 20 = 2ns (2 cycles per instruction)
- Design strategy
  - Reducing cache misses by more flexible placement of blocks
  - Reducing the miss penalty using multilevel caches

# Improving Cache Performance

- **Miss Rate Reduction Techniques:**
  - Increased cache capacity
  - Higher associativity
  - Hardware prefetching of instructions and data
  - Compiler-controlled prefetching
  - Larger block size
  - Victim caches
  - Pseudo-associative Caches
  - Compiler optimizations
- **Cache Miss Penalty Reduction Techniques:**
  - Giving priority to read misses over writes
  - Early restart and critical word first
  - Second-level cache ($L_2$)
  - Sub-block placement
  - Non-blocking caches
- **Cache Hit Time Reduction Techniques:**
  - Small and simple caches
  - Avoiding address translation during cache indexing
  - Pipelining writes for fast write hits

EECC551 - Shaaban

#7 Lec # 10 Winter2000 1-23-2000

## Qualitative Cache Performance Model

### Miss Types

- Compulsory ("Cold Start") Misses
  - First access to line not in cache
- Capacity Misses
  - Active portion of memory exceeds cache size
- Conflict Misses
  - Active portion of address space fits in cache, but too many lines map to same cache entry
  - Direct mapped and set associative placement only
- Coherence Misses
  - Block invalidated by multiprocessor cache coherence mechanism

### Hit Types

- Temporal locality hit
  - Accessing same word that previously accessed
- Spatial locality hit
  - Accessing word spatially near previously accessed word



# Cache Hit and Cache Miss

## Types of Cache Misses: *The Three C's*

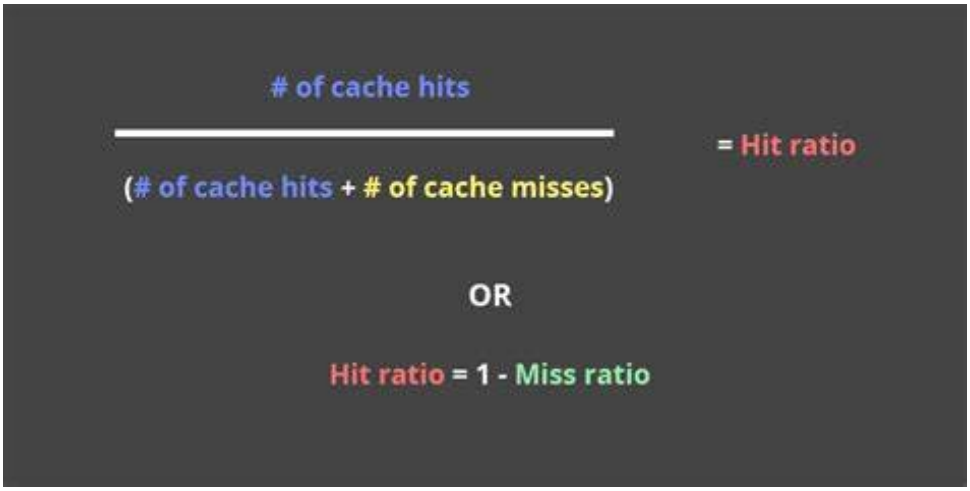
1 **Compulsory:** On the first access to a block; the block must be brought into the cache; also called cold start misses, or first reference misses.

2 **Capacity:** Occur because blocks are being discarded from cache because cache cannot contain all blocks needed for program execution (program working set is much larger than cache capacity).

3 **Conflict:** In the case of set associative or direct mapped block placement strategies, conflict misses occur when several blocks are mapped to the same set or block frame; also called collision misses or interference misses.

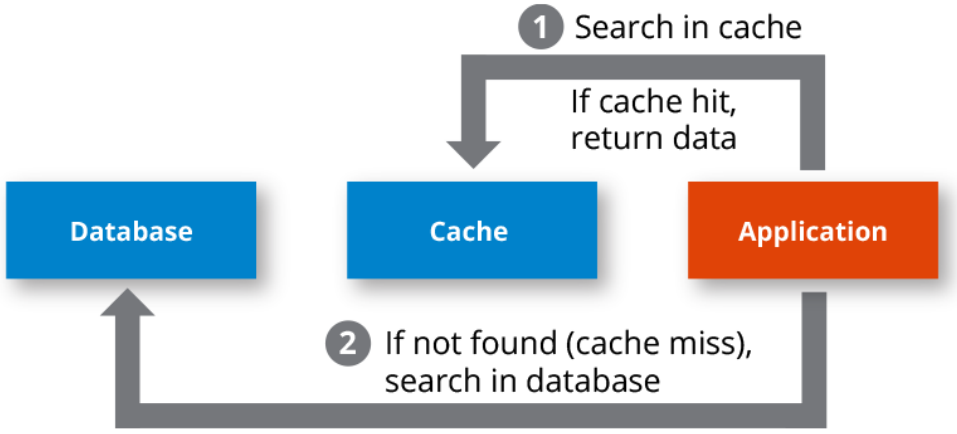
EECC551 - Shaaban

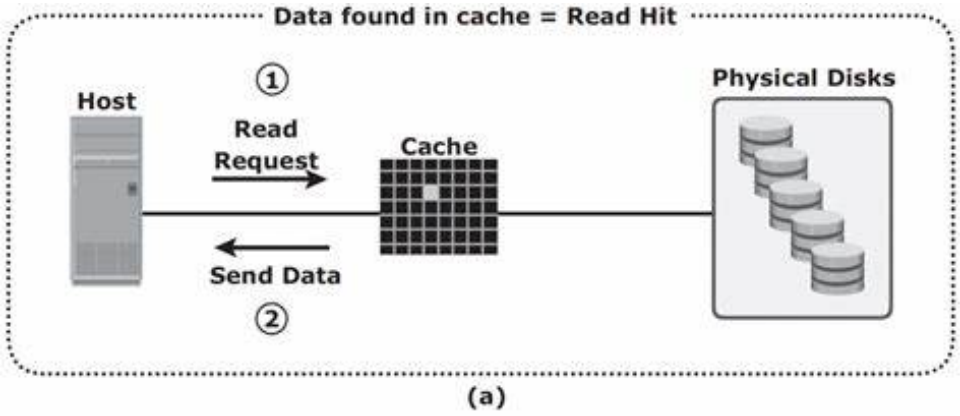
#1 Lec # 8 Winter 2001 1-30-2002

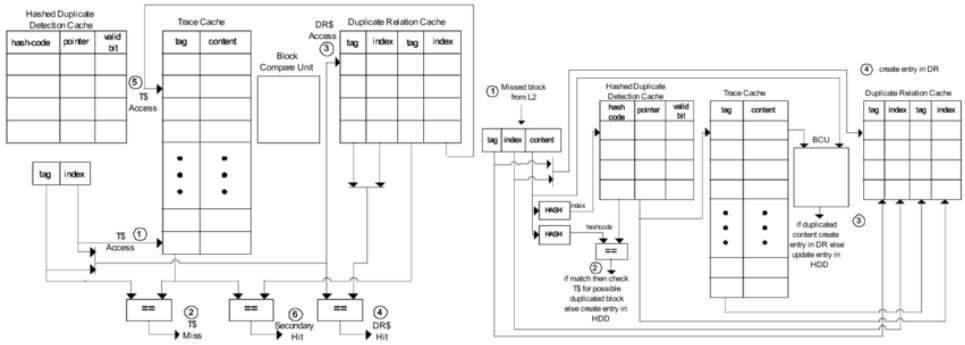


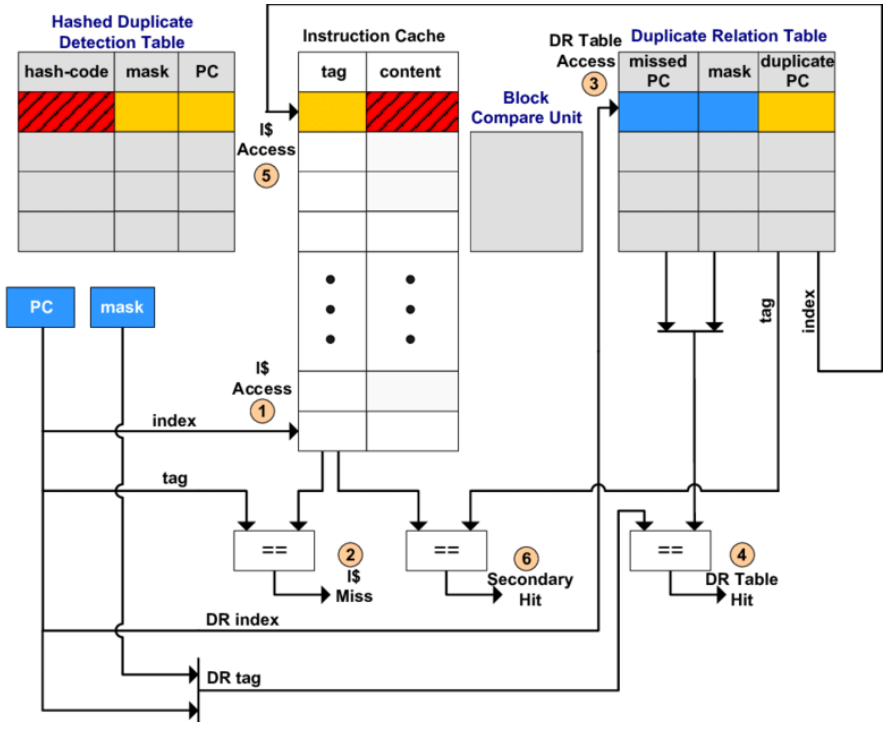

$$\text{Avg mem access time} = \text{Hit time}_{L1} + \text{Miss rate}_{L1} \times \text{Miss penalty}_{L1}$$
$$\text{Miss penalty}_{L1} = \text{Hit time}_{L2} + \text{Miss rate}_{L2} \times \text{Miss penalty}_{L2}$$

## Handle Cache Miss




Data found in cache = Read Hit
Host
Read Request
Cache
Physical Disks
Send Data
(a)
Data not found in cache = Read Miss
Host
Read Request
Cache
Read Request
Physical Disks
Send Data
Read from the Disk




Hashed Duplicate Detection Table
hash-code
mask
PC
Instruction Cache
tag
content
I$ Access
DR Table Access
Duplicate Relation Table
missed PC
mask
duplicate PC
Block Compare Unit
PC
mask
index
tag
I$ Miss
Secondary Hit
DR Table Hit
DR index
DR tag

# Associative Cache

*Example: Misses and Associativity in Caches*

課程:計算機組織 / 老師:朱宗賢

Given a 4-block cache and block access sequence (0, 8, 0, 6, 8), find the number of misses for please compare direct mapped, 2-way set associative, fully associative cache organization

**direct mapped**

Ans: **5** misses

| Block address | Cache index | Hit/miss | Cache content after access | | | |
|---|---|---|---|---|---|---|
| | | | 0 | 1 | 2 | 3 |
| 0 | 0 | miss | **Mem[0]** | | | |
| 8 | 0 | miss | **Mem[8]** | | | |
| 0 | 0 | miss | **Mem[0]** | | | |
| 6 | 2 | miss | **Mem[0]** | | **Mem[6]** | |
| 8 | 0 | miss | **Mem[8]** | | **Mem[6]** | |

**(Block number) mod (Number of blocks in the cache)**

**2-way set associative**

Ans: **4** misses

| Block address | Cache index | Hit/miss | Cache content after access | | | |
|---|---|---|---|---|---|---|
| | | | Set 0 | | Set 1 | |
| 0 | 0 | miss | **Mem[0]** | | | |
| 8 | 0 | miss | **Mem[0]** | **Mem[8]** | | |
| 0 | 0 | hit | **Mem[0]** | **Mem[8]** | | |
| 6 | 0 | miss | **Mem[0]** | **Mem[6]** | | |
| 8 | 0 | miss | **Mem[8]** | **Mem[6]** | | |

**Step1: (Block number) mod (Number of sets in the cache)**
**Step2: Search all tags of the set**

LRU (least recently used) Policy

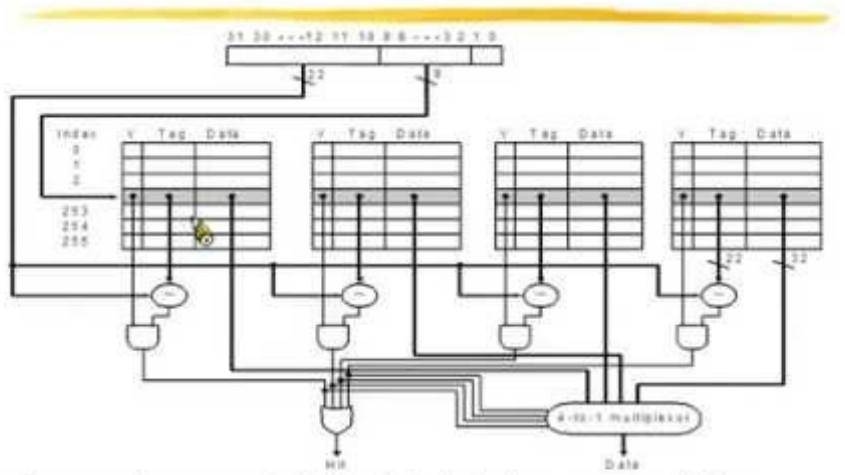
**fully associative**

Ans: **3** misses

| Block address | | Hit/miss | Cache content after access | | | |
|---|---|---|---|---|---|---|
| 0 | | miss | **Mem[0]** | | | |
| 8 | | miss | **Mem[0]** | **Mem[8]** | | |
| 0 | | hit | **Mem[0]** | **Mem[8]** | | |
| 6 | | miss | **Mem[0]** | **Mem[8]** | **Mem[6]** | |
| 8 | | hit | **Mem[0]** | **Mem[8]** | **Mem[6]** | |

Q1: 4 blocks -> 8 blocks
Q2: 4 blocks -> 16 blocks

## A 4-Way Set-Associative Cache

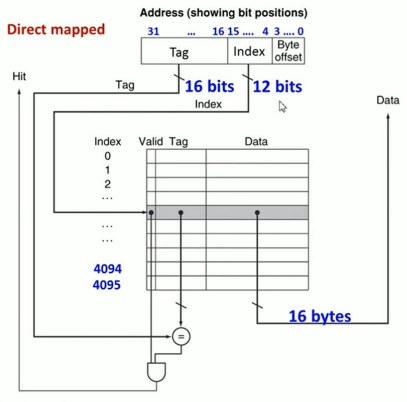


- Increasing associativity shrinks index, expands tag

Memory 19 Computer Architecture

課程:計算機組織 / 老師:朱宗賢

Q: Assuming a cache of 4096 blocks, a 4-word block size, and a 32-bit address, find the total number of set and the total number of tag bits for caches that are direct mapped, two-way and four way set associative and fully associative.




- Direct mapped
  **Size of the byte offset field = 4 bits**
  **Size of the index field = 12 bits**
  **Size of the tag field = 32-12-4 = 16 bits**
  **The total number of tag bits = 16 x 4096 = 65536 bits**
- Two-way set associative
  **Size of the byte offset field = 4 bits**
  **Size of the index field = 11 bits**
  **Size of the tag field = 32-11-4 = 17 bits**
  **The total number of tag bits = 17 x 4096 = 69632 bits**
- Four-way set associative
  **Size of the byte offset field = 4 bits**
  **Size of the index field = 10 bits**
  **Size of the tag field = 32-10-4 = 18 bits**
  **The total number of tag bits = 18 x 4096 = 73728 bits**
- Fully associative
  **Size of the byte offset field = 4 bits**
  **Size of the index field = 0 bits**
  **Size of the tag field = 32- 0- 4 = 28 bits**
  **The total number of tag bits = 28 x 4096 = 114688 bits**

# Block Size Tradeoff

## Miss Rate vs. Block Size

| Block size | Cache size | | | | |
|---|---|---|---|---|---|
| | 1K | 4K | 16K | 64K | 256K |
| 16 | 15.05% | 8.57% | 3.94% | 2.04% | 1.09% |
| 32 | 13.34% | 7.24% | 2.87% | 1.35% | 0.70% |
| 64 | 13.76% | 7.00% | 2.64% | 1.06% | 0.51% |
| 128 | 16.64% | 7.78% | 2.77% | 1.02% | 0.49% |
| 256 | 22.01% | 9.51% | 3.29% | 1.15% | 0.49% |

**FIGURE 5.12 Actual miss rate versus block size for five different-sized caches in Figure 5.11.** Note that for a 1-KB cache, 64-byte, 128-byte, and 256-byte blocks have a higher miss rate than 32-byte blocks. In this example, the cache would have to be 256 KB in order for a 256-byte block to decrease misses.

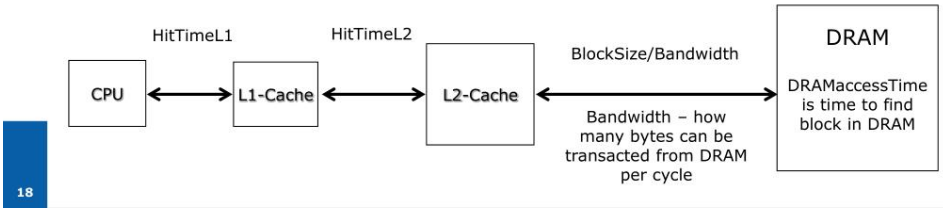
2/15/99 CS520S99 Memory C. Edward Chow Page 34

## Block Size Tradeoffs

- Larger block sizes...
  - Take advantage of spatial locality
  - Incur larger miss penalty since it takes longer to transfer the block into the cache
  - Can increase the average hit time and miss rate
- Average Access Time (AMAT) = HitTime + MissPenalty*MR

## 2-level Cache Performance Equations

- L1 AMAT = HitTimeL1 + MissRateL1 * MissPenaltyL1
  - MissLatencyL1 is low, so optimize HitTimeL1
- MissPenaltyL1 = HitTimeL2 + MissRateL2 * MissPenaltyL2
  - MissLatencyL2 is high, so optimize MissRateL2
  - MissPenaltyL2 = DRAMaccessTime + (BlockSize/Bandwidth)
  - If DRAM time high or bandwidth high, use larger block size
- L2 miss rate:
  - Global: L2 misses / total CPU references
  - Local: L2 misses / CPU references that miss in L1
  - The equation above assumes local miss rate

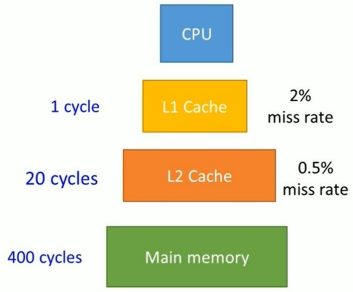



18

*Example: Performance of Multilevel Caches*

課程:計算機組織 / 老師:朱宗賢

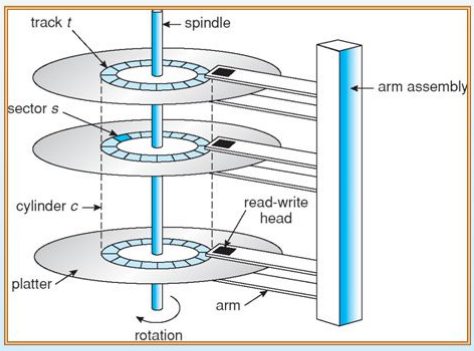
- Given
  CPU base CPI = 1, clock rate = 4GHz, Miss rate/instruction = 2%, Main memory access time = 100ns
- With just primary cache
  Miss penalty = 100ns/0.25ns = 400 cycles, Effective CPI = 1 + 0.02 × 400 = 9
- Now add L-2 cache
  Access time = 5ns, Global miss rate to main memory = 0.5%
- Primary miss with L-2 hit
  Penalty = 5ns/0.25ns = 20 cycles
- Primary miss with L-2 miss
- CPI = 1 + 0.02 × 20 + 0.005 × 400 = 1 + 0.4 + 2 = 3.4
- Performance ratio = 9/3.4 = 2.6
- Multilevel Cache Considerations
  - Primary cache - focus on minimal hit time
  - L-2 cache - focus on low miss rate
- Miss rate requirement of L-2 cache
  - 1+0.02x20 + Rx400 < 9 ----> R < 1.9%




# Storage Management

## Disk Structure

## Disk Structure



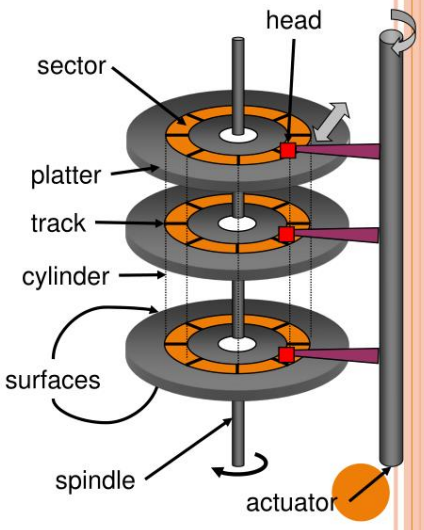

Moving-head disk mechanism

- Disk drives are addressed as large 1-dimensional arrays of **logical blocks**
  - The logical block is the smallest unit of transfer, usually 512 bytes
- The array of logical blocks is mapped into the **sectors** of the disk sequentially
  - Sector 0 is the first sector of the first **track** on the outermost **cylinder**
  - Mapping proceeds in order through that track, then the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost



## DISK DRIVE STRUCTURE

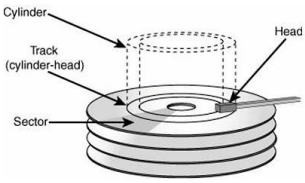
- Data stored on surfaces
  - Up to two surfaces per platter
  - One or more platters per disk
- Data in concentric tracks
  - Tracks broken into sectors
    - 256B-1KB per sector
  - Cylinder: corresponding tracks on all surfaces
- Data read and written by heads
  - Actuator moves heads
  - Heads move in unison




# Disk Access Time

## Operating System

# Disk Structure




- Track
- Sector
- Cylinder
- Spindle
- Read/ Write Head (Arm assembly)
- Seek Time
- Rotational Latency
- RPM
- Transfer Time

Presented by :Parul Vaghamshi

*Disk Access Time*

課程:計算機組織 / 老師:朱宗賢

**Question** : Given 512B sector, 15,000rpm, 4ms average seek time, 100MB/s transfer rate, 0.2ms controller overhead, idle disk, please calculate average read time.

**Access to a sector involves**

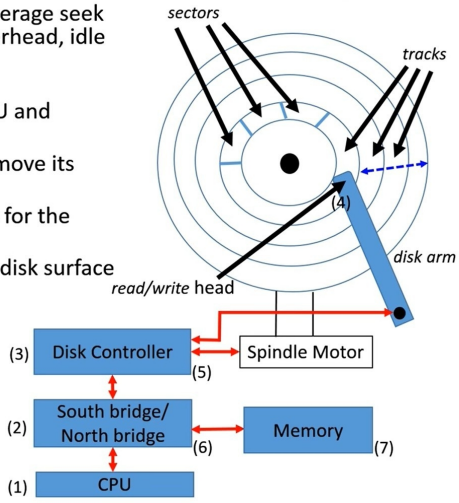
- Controller overhead: receiving orders from CPU and interpreting them
- Seek time: the time it takes for a disk drive to move its head(s) from one track to another
- Rotational latency: the amount of time it takes for the desired sector of a disk
- Data transfer rate: moving data from between disk surface and the host system

**Average read time**

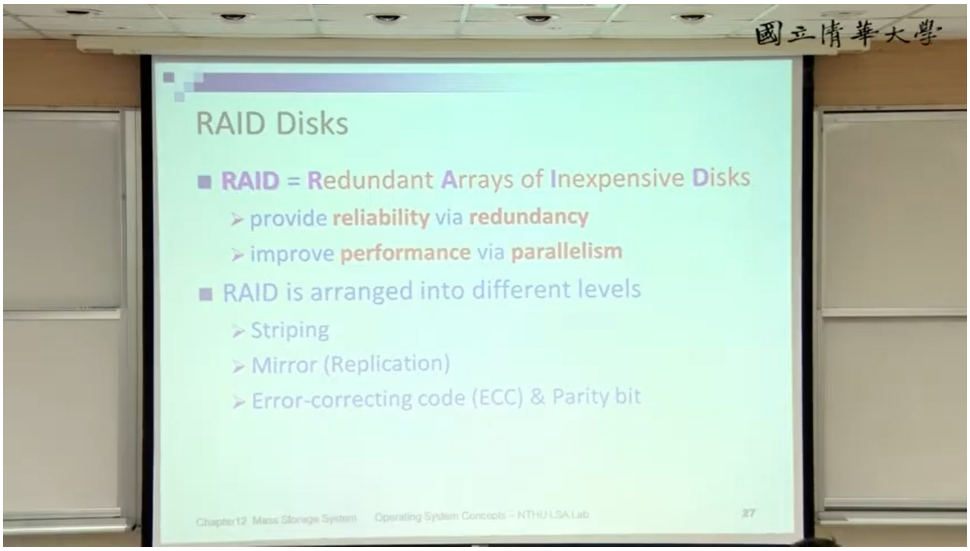
4ms seek time
+ ½ *(60/15,000) = 2ms rotational latency
+ 512 / 100MB/s = 0.005ms transfer time
+ 0.2ms controller delay = 6.2ms

**Operating System Design Policy**

- File Systems/ Contiguous allocation




# Redundant Array of Inexpensive Disk



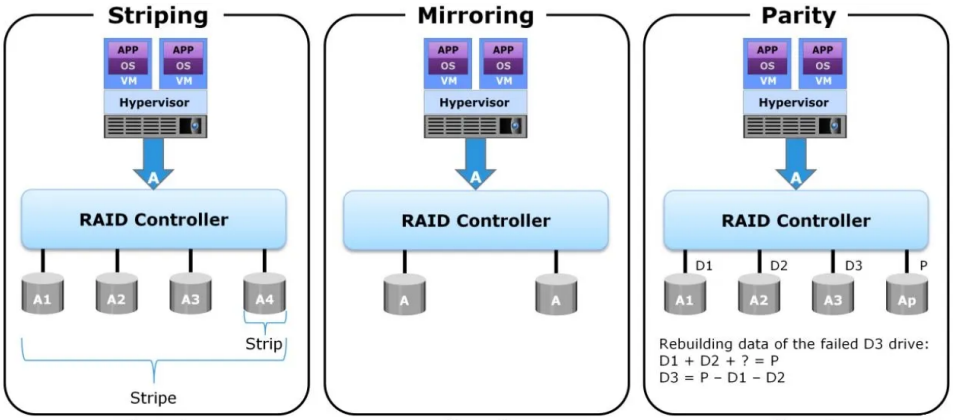
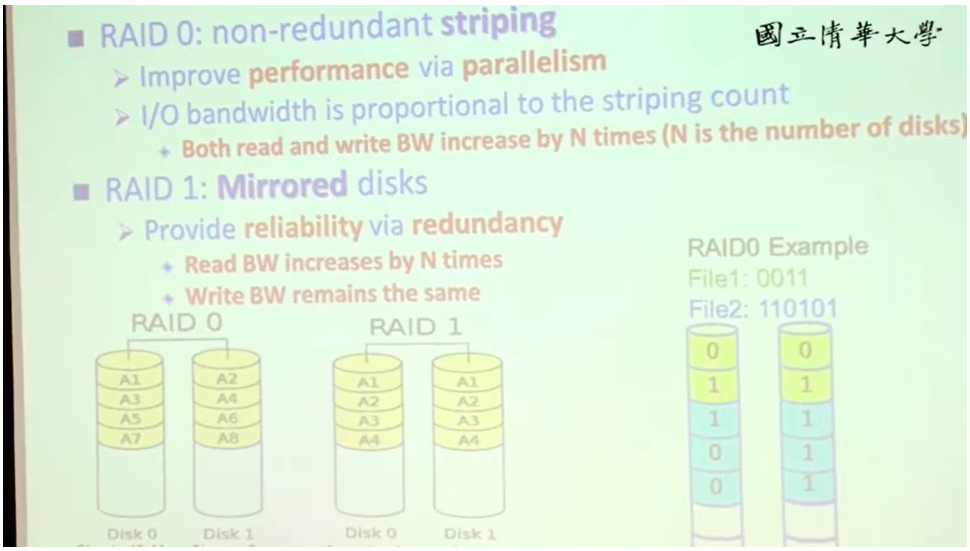
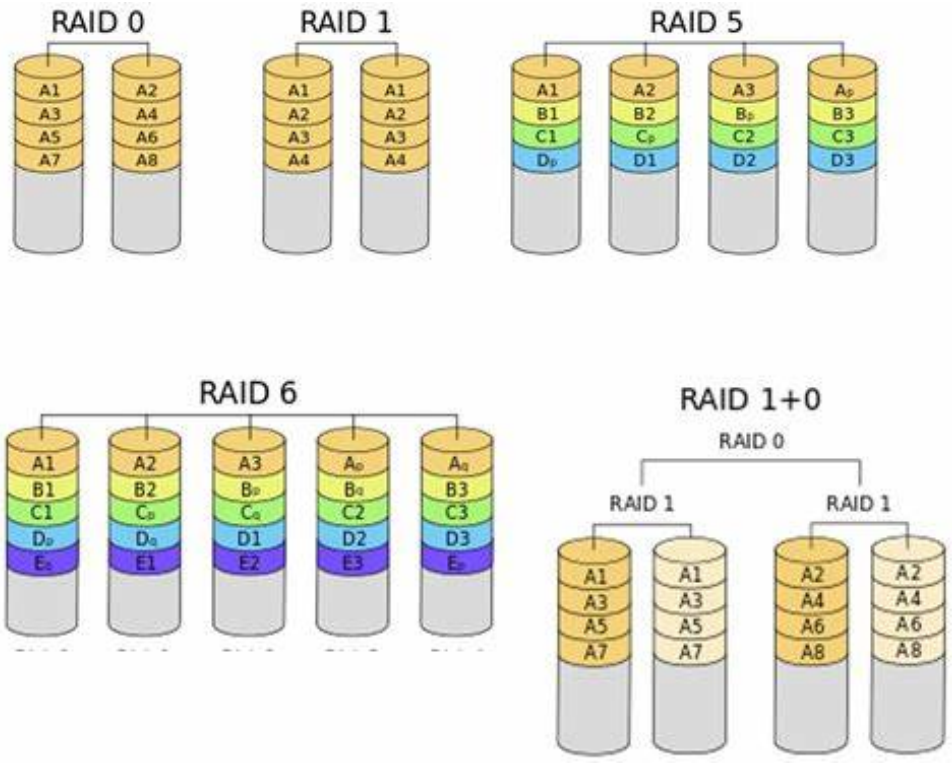
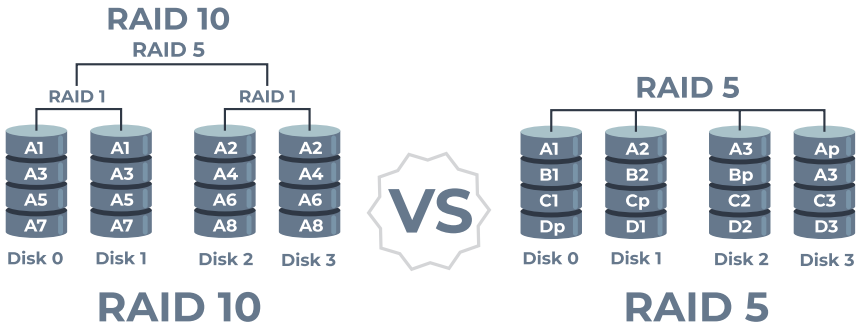



Figure 1 Figure 2 Figure 3



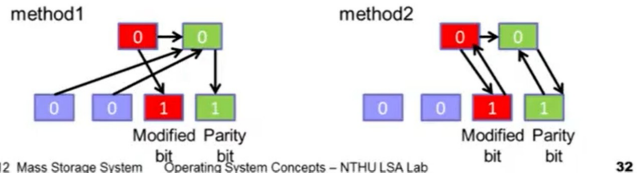






## RAID 5: Distributed Parity

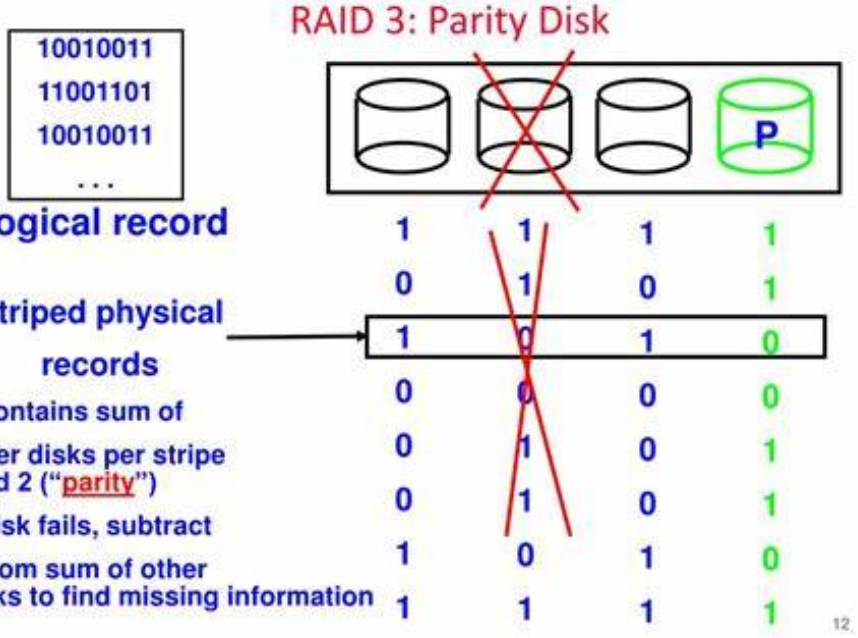
國立清華大學

- Read BW increases by N times, because all four disks can serve a read request
- Write BW
  - Method1: (1)read out all unmodified (N-2) data bits. (2) re-compute parity bit. (3) write both modified bit and parity bit to disks.
    → write BW = N / ((N-2)+2) = 1 → remains the same
  - Method2: (1)only read the parity bit and modified bit. (2) re-compute parity bit by the difference. (3) write both modified bit and parity bit.
    → write BW = N / (2+2) = N/4 times faster




Chapter12 Mass Storage System Operating System Concepts – NTHU LSA Lab 32

## Redundant Array of Inexpensive Disks
### RAID 3: Parity Disk

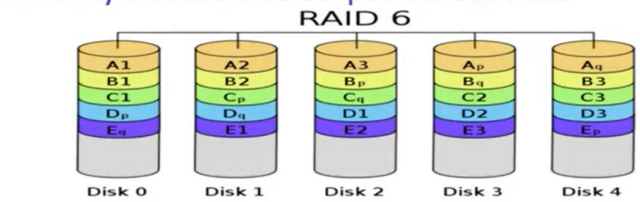



12

國立清華大學

## RAID 6: P+Q Dual Parity Redundancy

- Like RAID 5, but stores extra redundant information to guard against **multiple disk failure**
- Use ECE code (i.e. Error Correction Code) instead of single parity bit
- Parity bits are also striped across disks

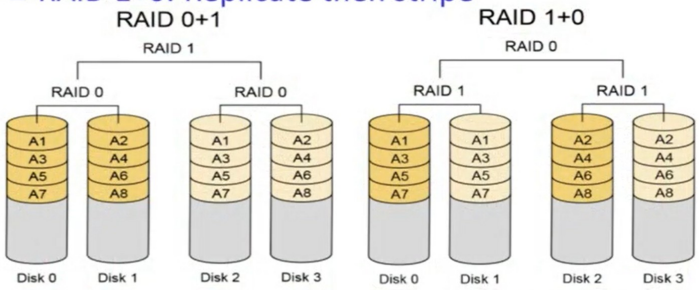





國立清華大學

## Hybrid RAID

- RAID 0+1: Stripe then replicate
- RAID 1+0: Replicate then stripe



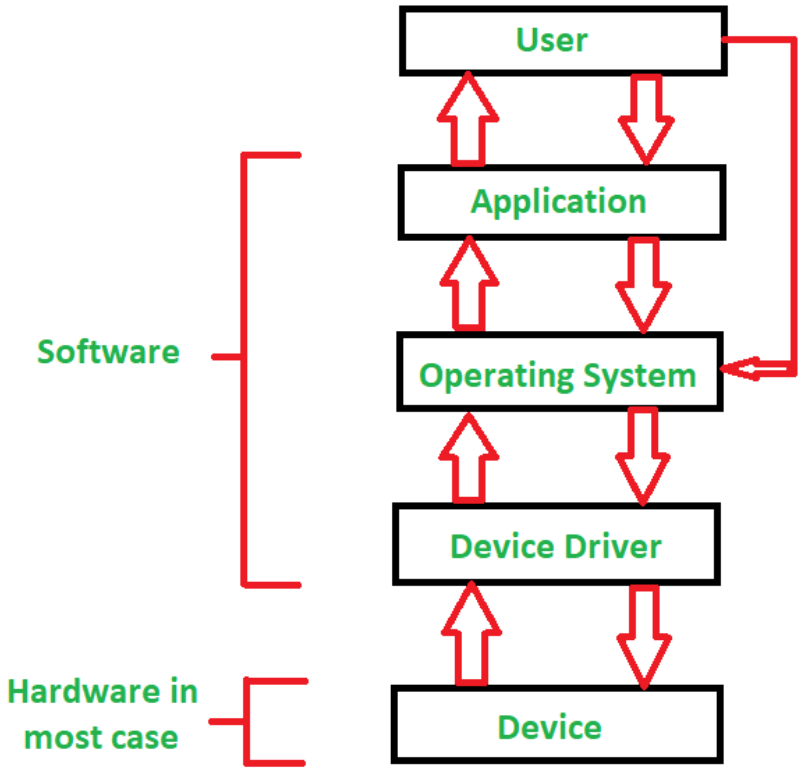

*First level often control by a controller. Therefore, RAID 10 has better fault tolerance than RAID 01 when multiple disk fails
http://www.thegeekstuff.com/2011/10/raid10-vs-raid01/

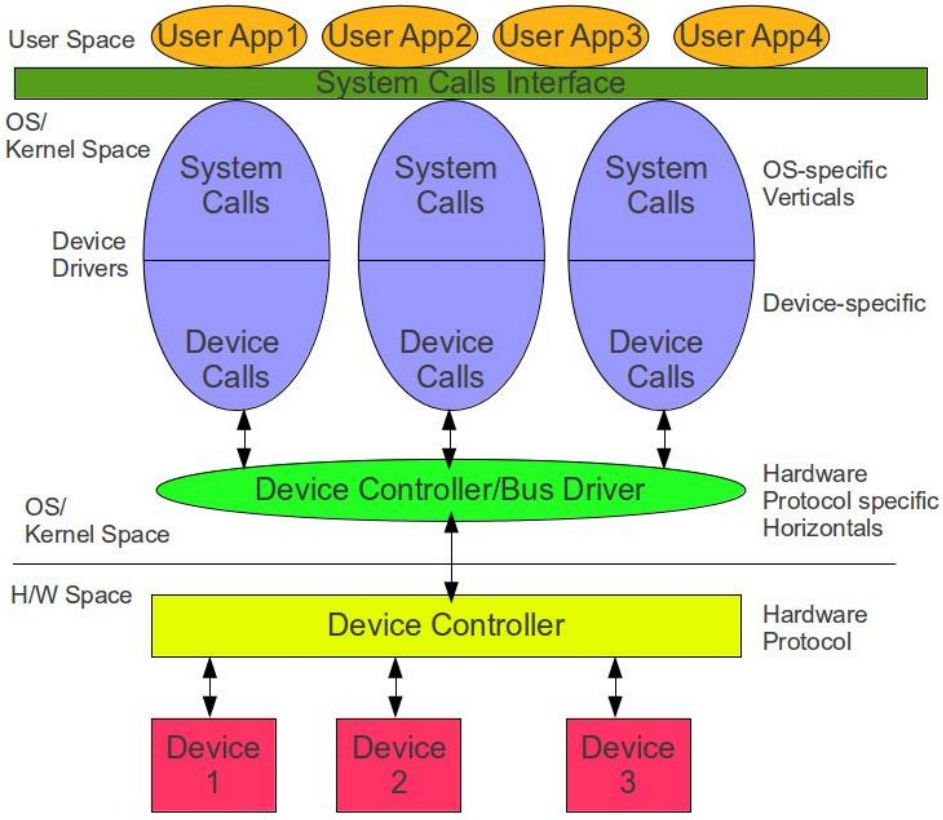


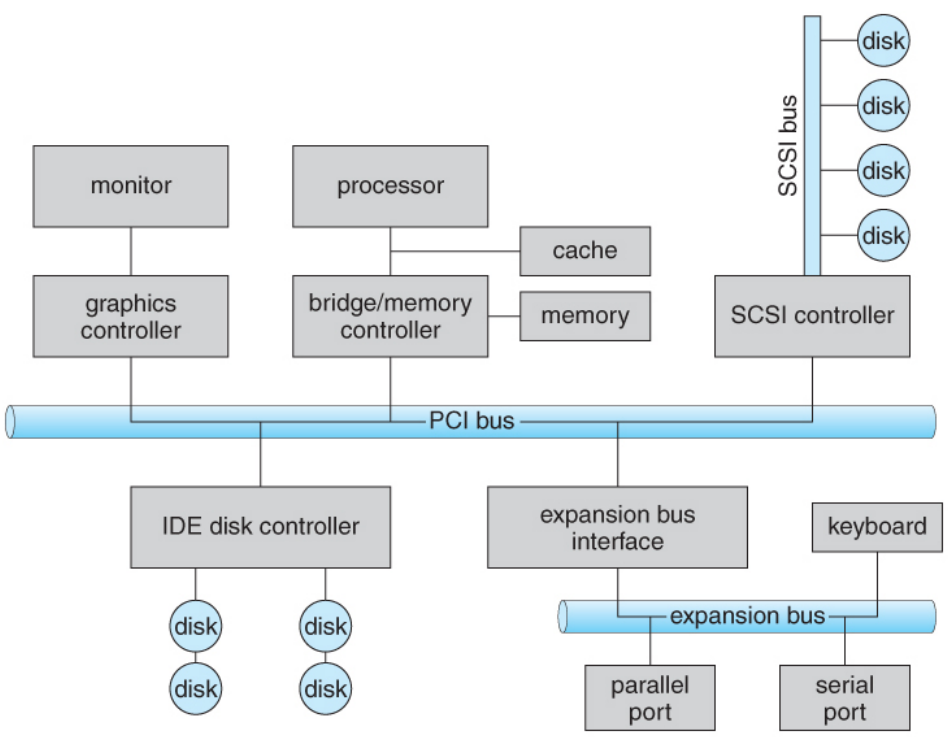
# I/O System Mangement

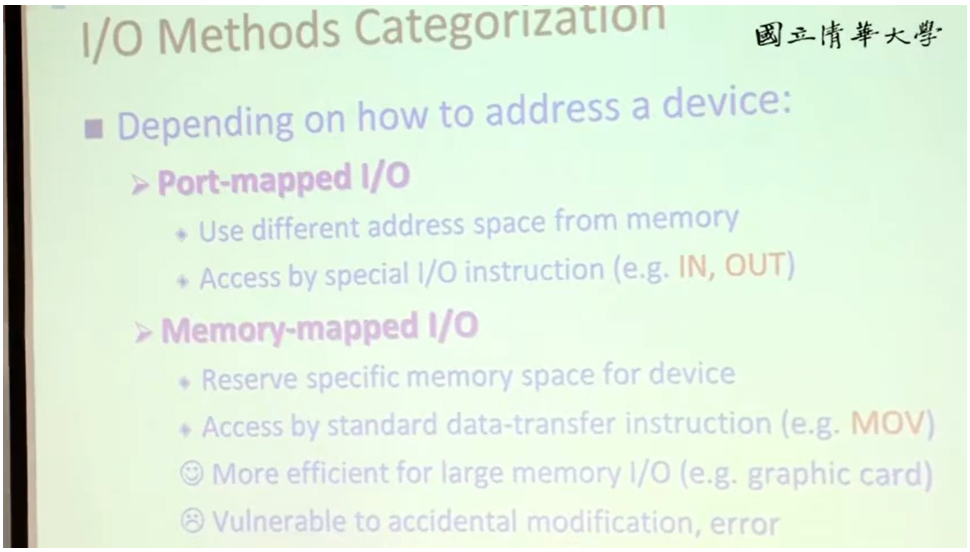
## I/O Hardware

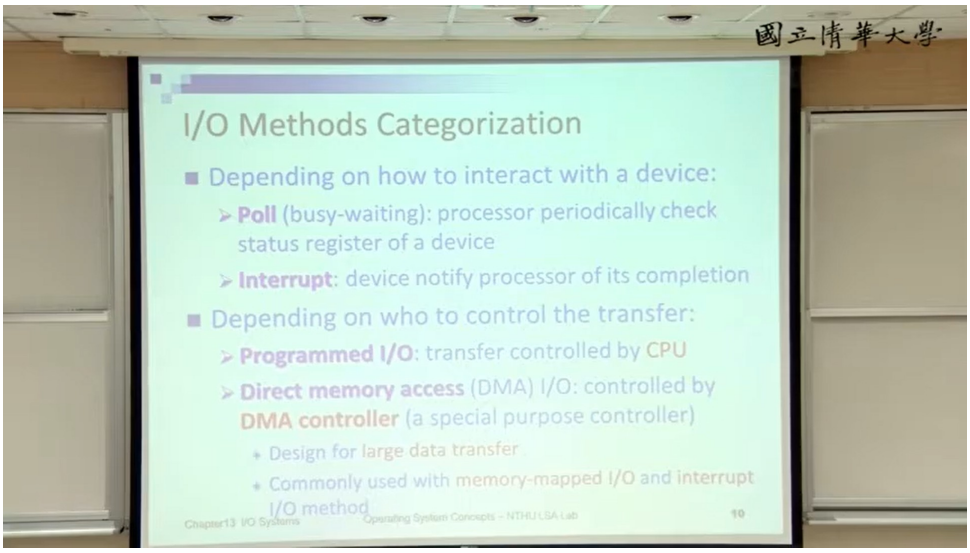
- Computers support a wide variety of I/O devices, but common concepts apply to all:
  - **Port**: connection point for a device
  - **Bus**: set of wires that connects to many devices, with a protocol specifying how information can be transmitted over the wires
  - **Controller**: a chip (or part of a chip) that operates a port, a bus, or a device
- How can the processor communicate with a device?
  - Special instructions allow the processor to transfer data and commands to registers on the device controller
  - The device controller may support **memory-mapped I/O**:
    - The same address bus is used for both memory and device access.
    - The same CPU instructions can access memory locations or devices.


User
Application
Operating System
Device Driver
Device
Software
Hardware in most case


User Space
User App1
User App2
User App3
User App4
System Calls Interface
OS/ Kernel Space
System Calls
System Calls
System Calls
OS-specific Verticals
Device Drivers
Device Calls
Device Calls
Device Calls
Device-specific
Device Controller/Bus Driver
Hardware Protocol specific Horizontals
OS/ Kernel Space
H/W Space
Device Controller
Hardware Protocol
Device 1
Device 2
Device 3



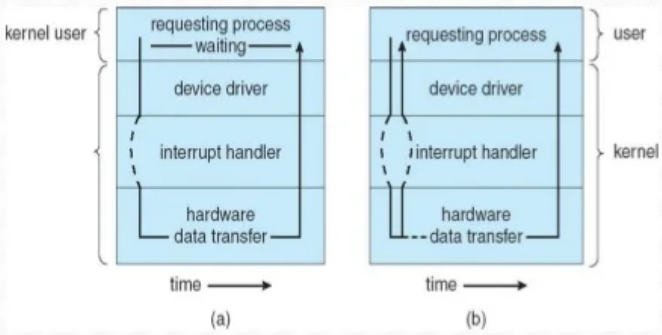

# IO System Performance

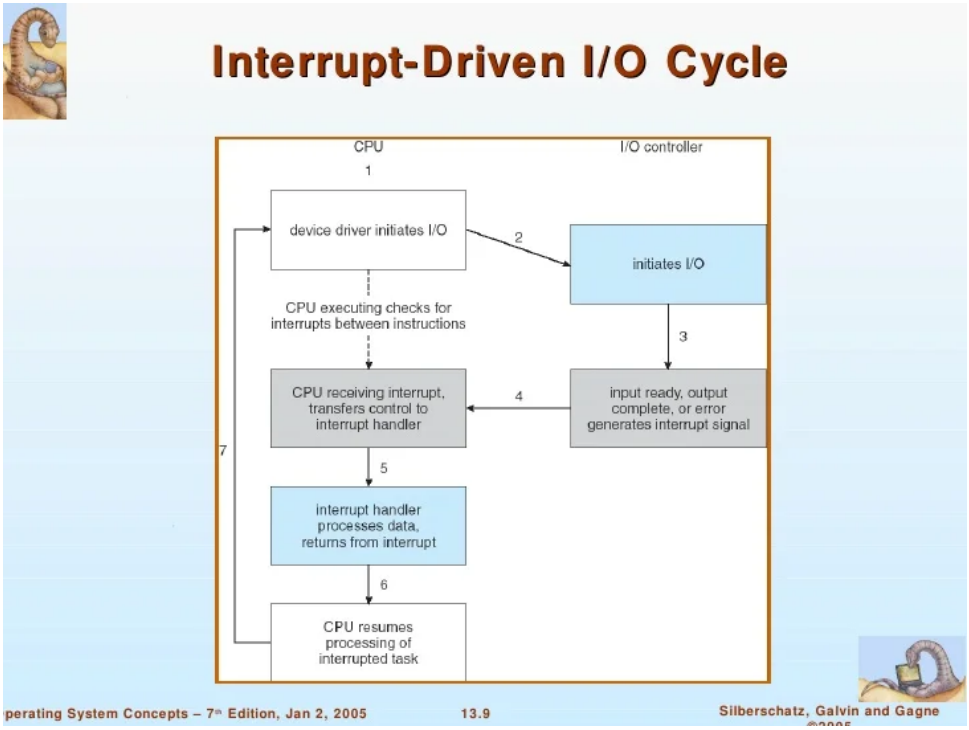
## Common Concepts in I/O Hardware

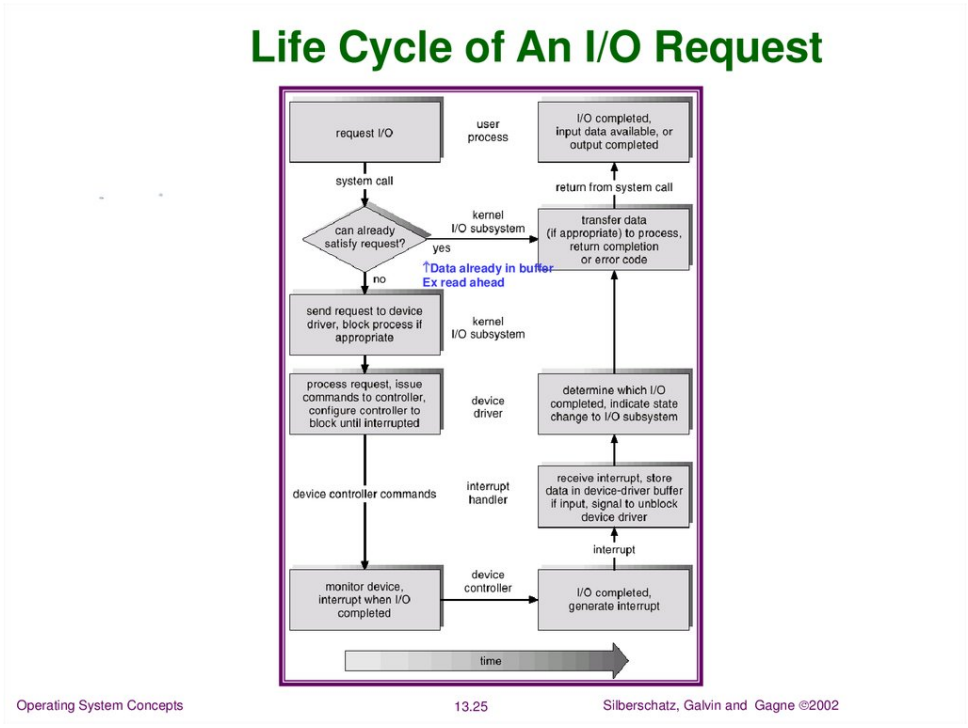
- Common concepts: signals from I/O devices interface with computer.
- Port: connection point for device
- Bus: set of wires and a protocol that specifies a set of messages that can be sent on the wires.
- Controller: a collection of electronics that can operate a port, a bus, or a device.
  - Sometimes integrated and sometimes separate circuit board (host adapter)
  - Contains processor, microcode, private memory, bus controller, etc



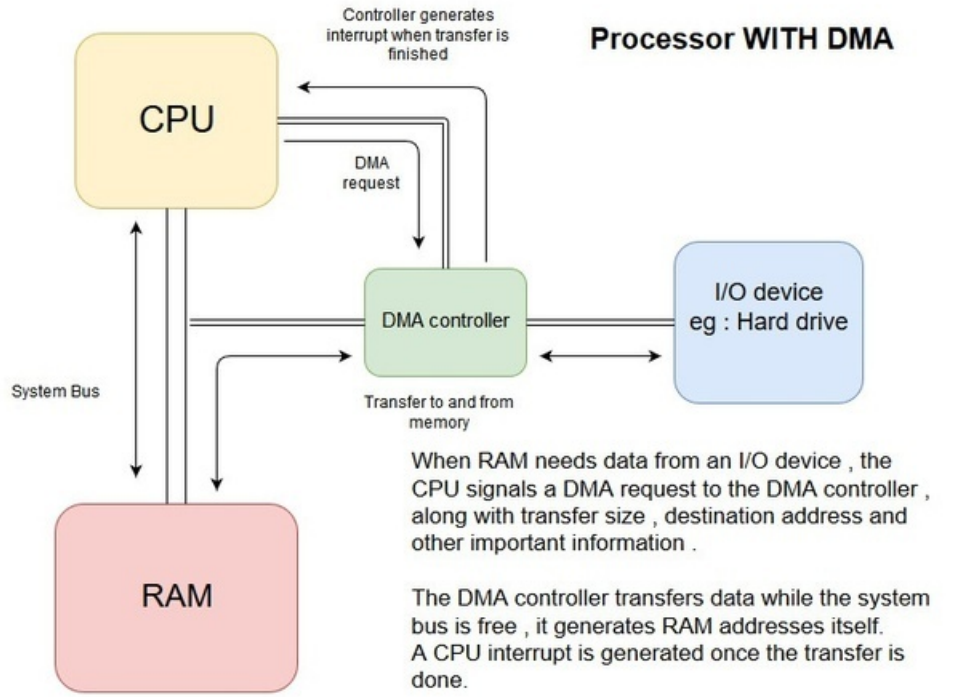
## Two I/O Methods

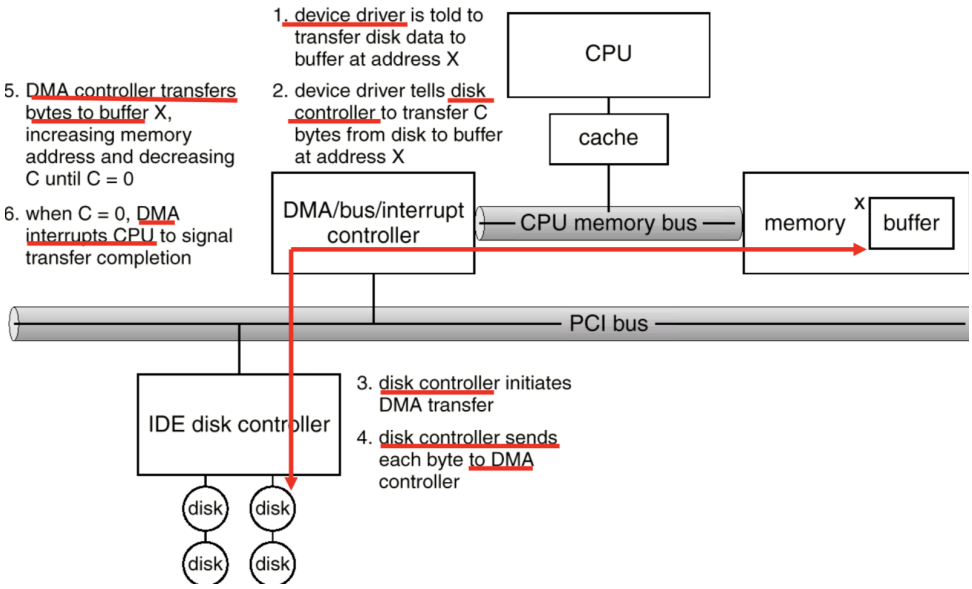


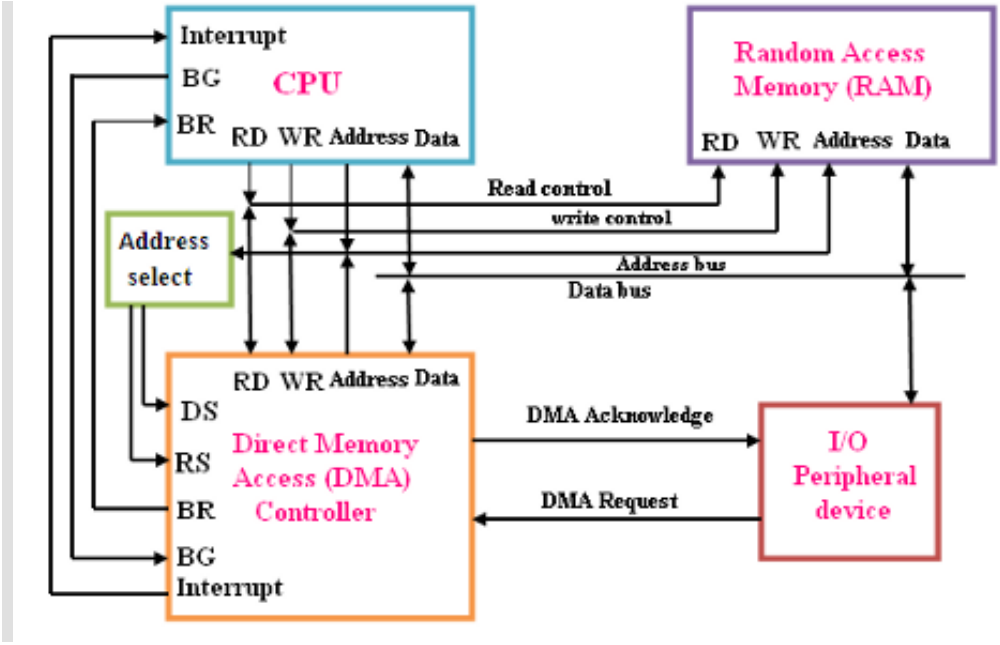



# DMA(Direct Memroy Access)


Processor WITH DMA
Controller generates interrupt when transfer is finished
CPU
DMA request
DMA controller
I/O device eg : Hard drive
System Bus
Transfer to and from memory
RAM
When RAM needs data from an I/O device , the CPU signals a DMA request to the DMA controller , along with transfer size , destination address and other important information .
The DMA controller transfers data while the system bus is free , it generates RAM addresses itself.
A CPU interrupt is generated once the transfer is done.
Result : CPU is free to do other things !


1. device driver is told to transfer disk data to buffer at address X
2. device driver tells disk controller to transfer C bytes from disk to buffer at address X
3. disk controller initiates DMA transfer
4. disk controller sends each byte to DMA controller
5. DMA controller transfers bytes to buffer X, increasing memory address and decreasing C until C = 0
6. when C = 0, DMA interrupts CPU to signal transfer completion
CPU
cache
DMA/bus/interrupt controller
CPU memory bus
memory
x
buffer
PCI bus
IDE disk controller
disk
disk
disk
disk


Interrupt
BG
CPU
BR
RD WR Address Data
Random Access Memory (RAM)
RD WR Address Data
Read control
write control
Address select
Address bus
Data bus
RD WR Address Data
DS
Direct Memory Access (DMA) Controller
RS
BR
BG
Interrupt
DMA Acknowledge
DMA Request
I/O Peripheral device

-- Memo End --