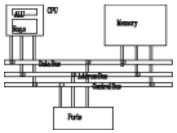
# *Computer Organization and Assembly Languages*

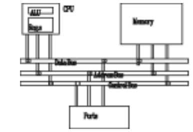
*with slides by Kip Irvine*

# Prerequisites



- Programming experience with some high-level language such C, C ++,Java …

# Books



## Textbook

*Assembly Language for Intel-Based Computers*, 4th, 5th, 6th Edition, Kip Irvine

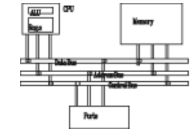
## Reference

*The Art of Assembly Language*, Randy Hyde

# Grading (subject to change)

- Assignments/Quizzes (20%)
- Class participation (10%)
- Midterm exam (30%)
- Final Exam (40%)

# Why learning assembly?

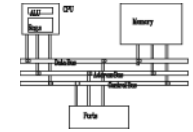


- It is required.
- It is foundation for computer architecture and compilers.
- At times, you do need to write assembly code.

*"I really don't think that you can write a book for serious computer programmers unless you are able to discuss low-level details."*

Donald Knuth

# Why programming in assembly?



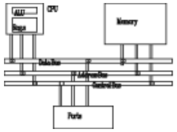
- It is all about lack of smart compilers

- Faster code, compiler is not good enough
- Smaller code , compiler is not good enough, e.g. mobile devices, embedded devices, also Smaller code → better cache performance → faster code
- Unusual architecture , there isn’t even a compiler or compiler quality is bad, eg GPU, DSP chips, even MMX.

# Syllabus (topics we might cover)

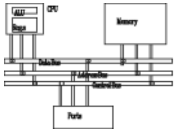
- IA-32 Processor Architecture
- Assembly Language Fundamentals
- Data Transfers, Addressing, and Arithmetic
- Procedures
- Conditional Processing
- Integer Arithmetic
- Advanced Procedures
- Strings and Arrays
- Structures and Macros

# What you will learn



- Basic principle of computer architecture
- IA-32 modes and memory management
- Assembly basics
- How high-level language is translated to assembly
- How to communicate with OS

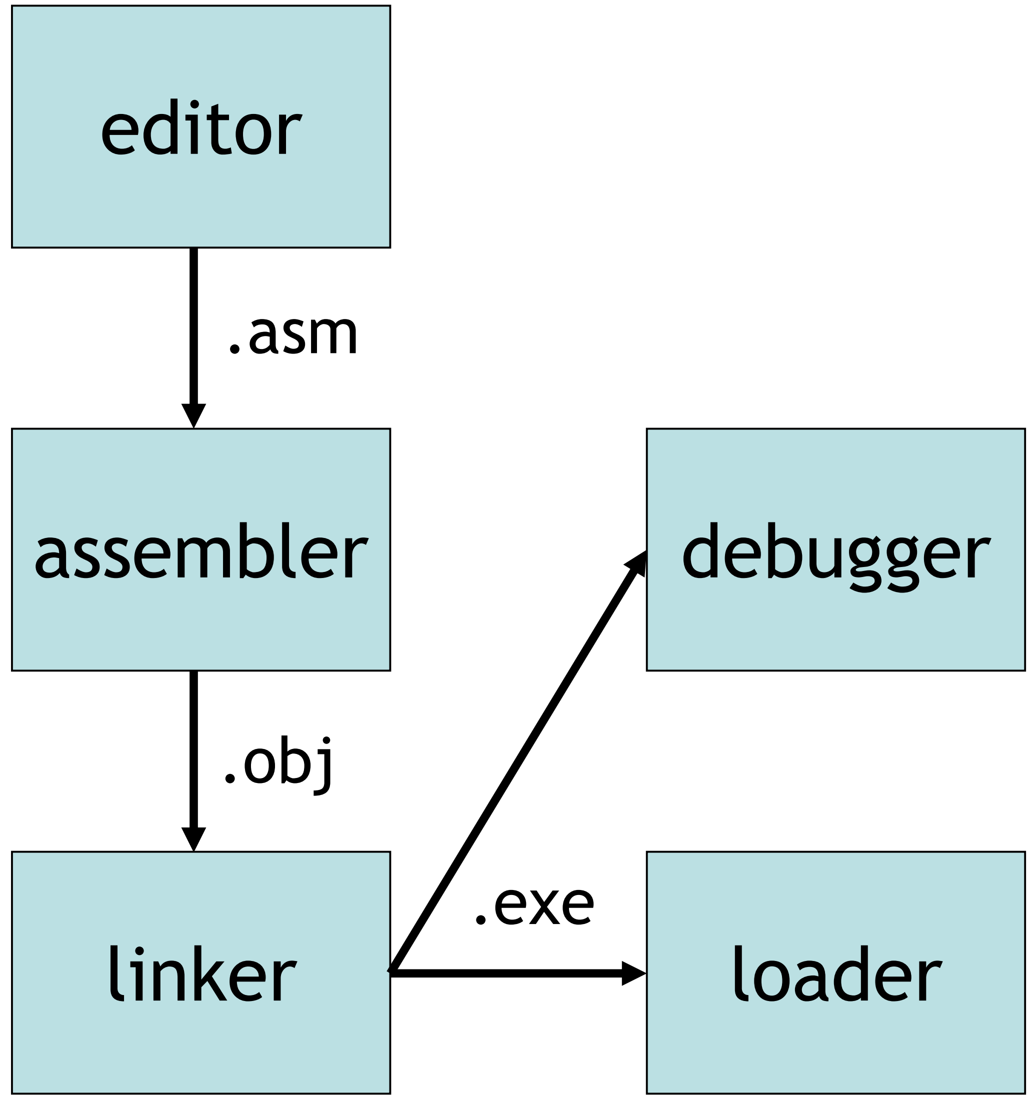
# Chapter.1 Overview



- Virtual Machine Concept
- Data Representation
- Boolean Operations

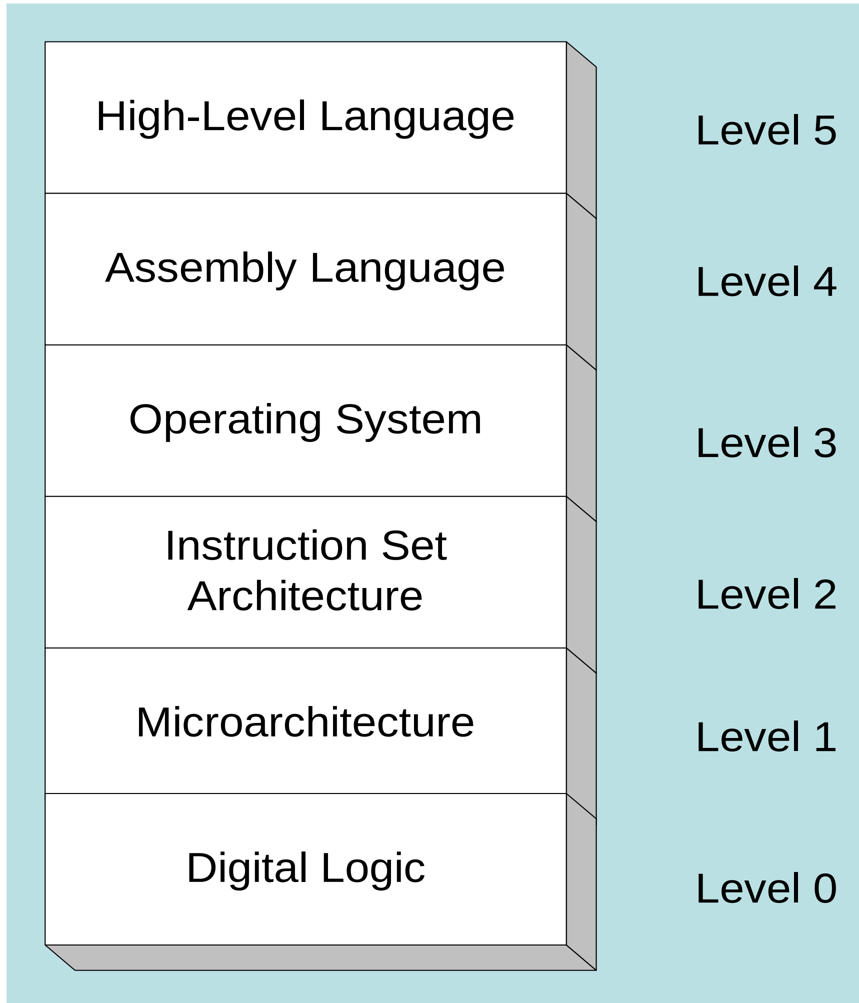
# Assembly programming

```
mov eax, Y
add eax, 4
mov ebx, 3
imul ebx
mov X, eax
```

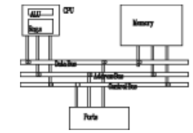


# Virtual machines

## Abstractions for computers



# High-Level Language



- Level 5
- Application-oriented languages
- Programs compile into assembly language (Level 4)

```
X:=(Y+4)*3
```

# Assembly Language

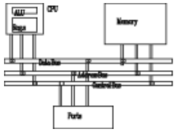
- Level 4
- Instruction mnemonics that have a one-to-one correspondence to machine language
- Calls functions written at the operating system level (Level 3)
- Programs are translated into machine language (Level 2)

```
mov eax, Y
add eax, 4
mov ebx, 3
imul ebx
mov X, eax
```

# Operating System

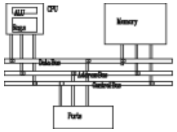
- Level 3
- Provides services
- Programs translated and run at the instruction set architecture level (Level 2)

# Instruction Set Architecture



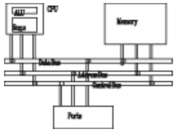
- Level 2
- Also known as conventional machine language
- Executed by Level 1 program (microarchitecture, Level 1)

# Microarchitecture



- Level 1
- Interprets conventional machine instructions (Level 2)
- Executed by digital hardware (Level 0)

# Digital Logic



- Level 0
- CPU, constructed from digital logic gates
- System bus
- Memory

# Data representation

- Computer is a construction of digital circuits with two states: *on* and *off*
- You need to have the ability to translate between different representations to examine the content of the machine
- Common number systems: binary, octal, decimal and hexadecimal

# Binary numbers

- Digits are 1 and 0
  (a binary digit is called a bit)
  1 = true
  0 = false
- MSB –most significant bit
- LSB –least significant bit
- Bit numbering:

| MSB | | LSB |
|---|---|---|
| | 1 0 1 1 0 0 1 0 1 0 0 1 1 1 0 0 | |
| 15 | | 0 |

- A bit string could have different interpretations

# Unsigned binary integers

- Each digit (bit) is either 1 or 0
- Each bit represents a power of 2:

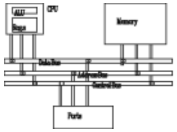
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|
| $2^7$ | $2^6$ | $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ |

Every binary number is a sum of powers of 2

**Table 1-3** Binary Bit Position Values.

| $2^n$ | Decimal Value | $2^n$ | Decimal Value |
|---|---|---|---|
| $2^0$ | 1 | $2^8$ | 256 |
| $2^1$ | 2 | $2^9$ | 512 |
| $2^2$ | 4 | $2^{10}$ | 1024 |
| $2^3$ | 8 | $2^{11}$ | 2048 |
| $2^4$ | 16 | $2^{12}$ | 4096 |
| $2^5$ | 32 | $2^{13}$ | 8192 |
| $2^6$ | 64 | $2^{14}$ | 16384 |
| $2^7$ | 128 | $2^{15}$ | 32768 |

# Translating Binary to Decimal



Weighted positional notation shows how to calculate the decimal value of each binary bit:

$$\boldsymbol{dec = (D_{n-1} \times 2^{n-1}) + (D_{n-2} \times 2^{n-2}) + \ldots + (D_1 \times 2^1) + (D_0 \times 2^0)}$$

D = binary digit

binary 00001001 = decimal 9:

$(1 \times 2^3) + (1 \times 2^0) = 9$

# Translating Unsigned Decimal to Binary

- Repeatedly divide the decimal integer by 2. Each remainder is a binary digit in the translated value:

| Division | Quotient | Remainder |
|---|---|---|
| 37 / 2 | 18 | 1 |
| 18 / 2 | 9 | 0 |
| 9 / 2 | 4 | 1 |
| 4 / 2 | 2 | 0 |
| 2 / 2 | 1 | 0 |
| 1 / 2 | 0 | 1 |

37 = 100101

# Binary addition

- Starting with the LSB, add each pair of digits, include the carry if present.

carry: 1

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | (4) |
| + | | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | (7) |
| | | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | (11) |
| bit position: | | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |

# Integer storage sizes

Standard sizes:

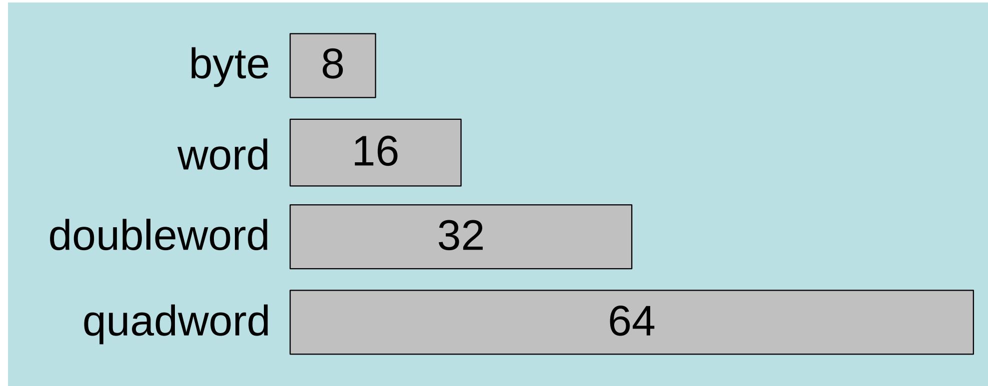



**Table 1-4** Ranges of Unsigned Integers.

| Storage Type | Range (low–high) | Powers of 2 |
|---|---|---|
| Unsigned byte | 0 to 255 | 0 to $(2^8 - 1)$ |
| Unsigned word | 0 to 65,535 | 0 to $(2^{16} - 1)$ |
| Unsigned doubleword | 0 to 4,294,967,295 | 0 to $(2^{32} - 1)$ |
| Unsigned quadword | 0 to 18,446,744,073,709,551,615 | 0 to $(2^{64} - 1)$ |

Practice: What is the largest unsigned integer that may be stored in 20 bits?

# Large measurements

- Kilobyte (KB), $2^{10}$ bytes
- Megabyte (MB), $2^{20}$ bytes
- Gigabyte (GB), $2^{30}$ bytes
- Terabyte (TB), $2^{40}$ bytes
- Petabyte
- Exabyte
- Zettabyte
- Yottabyte

# Hexadecimal integers

All values in memory are stored in binary. Because long binary numbers are hard to read, we use hexadecimal representation.

**Table 1-5** Binary, Decimal, and Hexadecimal Equivalents.

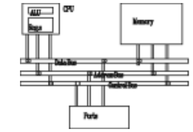
| Binary | Decimal | Hexadecimal | Binary | Decimal | Hexadecimal |
|---|---|---|---|---|---|
| 0000 | 0 | 0 | 1000 | 8 | 8 |
| 0001 | 1 | 1 | 1001 | 9 | 9 |
| 0010 | 2 | 2 | 1010 | 10 | A |
| 0011 | 3 | 3 | 1011 | 11 | B |
| 0100 | 4 | 4 | 1100 | 12 | C |
| 0101 | 5 | 5 | 1101 | 13 | D |
| 0110 | 6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

# Translating binary to hexadecimal

- Each hexadecimal digit corresponds to 4 binary bits.
- Example: Translate the binary integer 000101101010011110010100 to hexadecimal:

| | | | | | |
|---|---|---|---|---|---|
| 1 | 6 | A | 7 | 9 | 4 |
| 0001 | 0110 | 1010 | 0111 | 1001 | 0100 |

# Converting hexadecimal to decimal



- Multiply each digit by its corresponding power of 16:

$$dec = (D_3 \times 16^3) + (D_2 \times 16^2) + (D_1 \times 16^1) + (D_0 \times 16^0)$$

- Hex 1234 equals $(1 \times 16^3) + (2 \times 16^2) + (3 \times 16^1) + (4 \times 16^0)$, or decimal 4,660.

- Hex 3BA4 equals $(3 \times 16^3) + (11 * 16^2) + (10 \times 16^1) + (4 \times 16^0)$, or decimal 15,268.

# Powers of 16

Used when calculating hexadecimal values up to 8 digits long:

| $16^n$ | Decimal Value | $16^n$ | Decimal Value |
|---|---|---|---|
| $16^0$ | 1 | $16^4$ | 65,536 |
| $16^1$ | 16 | $16^5$ | 1,048,576 |
| $16^2$ | 256 | $16^6$ | 16,777,216 |
| $16^3$ | 4096 | $16^7$ | 268,435,456 |

# Converting decimal to hexadecimal

| Division | Quotient | Remainder |
|---|---|---|
| 422 / 16 | 26 | 6 |
| 26 / 16 | 1 | A |
| 1 / 16 | 0 | 1 |

decimal 422 = 1A6 hexadecimal

# Hexadecimal addition

Divide the sum of two digits by the number base (16). The quotient becomes the carry value, and the remainder is the sum digit.

| | | | |
|---|---|---|---|
| | | **1** | **1** |
| 36 | 28 | 28 | 6A |
| 42 | 45 | 58 | 4B |
| 78 | 6D | 80 | B5 |

Important skill: Programmers frequently add and subtract the addresses of variables and instructions.

# Hexadecimal subtraction

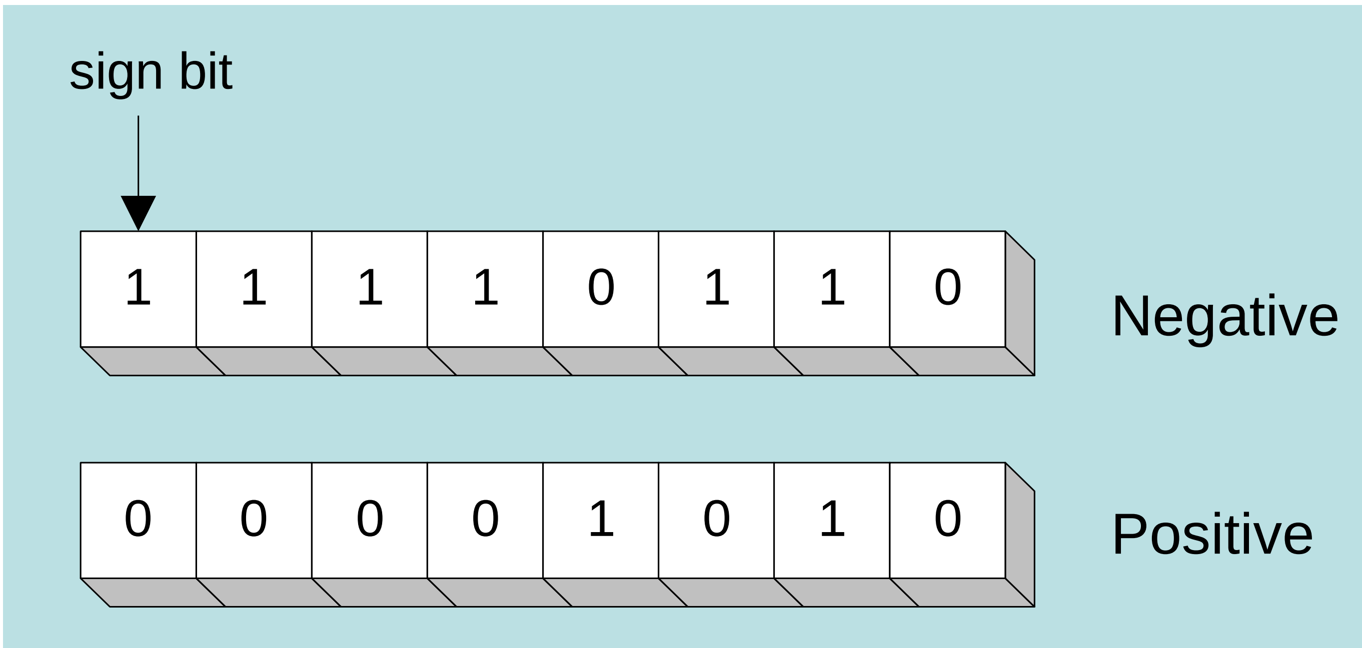
When a borrow is required from the digit to the left, add 10h to the current digit's value:

$$\begin{array}{rr} & \mathbf{-1} \\ \text{C6} & \text{75} \\ \text{A2} & \text{47} \\ \hline \text{24} & \text{2E} \end{array}$$

Practice: The address of **var1** is 00400020. The address of the next variable after var1 is 0040006A. How many bytes are used by var1?

# Signed integers

The highest bit indicates the sign. 1 = negative, 0 = positive




If the highest digit of a hexadecmal integer is > 7, the value is negative. Examples: 8A, C5, A2, 9D

# Two's complement notation

Steps:

- Complement (reverse) each bit
- Add 1

| | |
|---|---|
| Starting value | 00000001 |
| Step 1: reverse the bits | 11111110 |
| Step 2: add 1 to the value from Step 1 | 11111110<br>+00000001 |
| Sum: two's complement representation | 11111111 |

Note that 00000001 + 11111111 = 00000000

# Binary subtraction

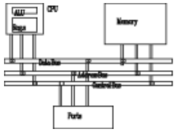
- When subtracting A – B, convert B to its two's complement
- Add A to (–B)

$$\begin{array}{r} 1\,1\,0\,0 \\ -\;0\,0\,1\,1 \\ \hline \end{array} \longrightarrow \begin{array}{r} 1\,1\,0\,0 \\ 1\,1\,0\,1 \\ \hline 1\,0\,0\,1 \end{array}$$

Advantages for 2's complement:

- No two 0's
- Sign bit
- Remove the need for separate circuits for add and sub

# Ranges of signed integers



The highest bit is reserved for the sign. This limits the range:

| Storage Type | Range (low–high) | Powers of 2 |
|---|---|---|
| Signed byte | −128 to +127 | $-2^7$ to $(2^7 - 1)$ |
| Signed word | −32,768 to +32,767 | $-2^{15}$ to $(2^{15} - 1)$ |
| Signed doubleword | −2,147,483,648 to 2,147,483,647 | $-2^{31}$ to $(2^{31} - 1)$ |
| Signed quadword | −9,223,372,036,854,775,808 to +9,223,372,036,854,775,807 | $-2^{63}$ to $(2^{63} - 1)$ |

# Character

- Character sets
  - Standard ASCII(0 – 127)
  - Extended ASCII (0 – 255)
  - ANSI (0 – 255)
  - Unicode  (0 – 65,535)
- Null-terminated String
  - Array of characters followed by a *null byte*
- Using the ASCII table
  - back inside cover of book

# Boolean algebra

- Boolean expressions created from:
  - NOT, AND, OR

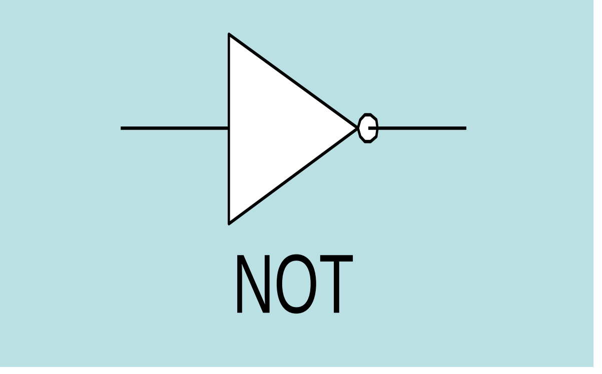
| Expression | Description |
|---|---|
| $\neg X$ | NOT X |
| $X \wedge Y$ | X AND Y |
| $X \vee Y$ | X OR Y |
| $\neg X \vee Y$ | ( NOT X ) OR Y |
| $\neg(X \wedge Y)$ | NOT ( X AND Y ) |
| $X \wedge \neg Y$ | X AND ( NOT Y ) |

# NOT

- Inverts (reverses) a boolean value
- Truth table for Boolean NOT operator:

| **X** | **¬X** |
|---|---|
| F | T |
| T | F |

Digital gate diagram for NOT:

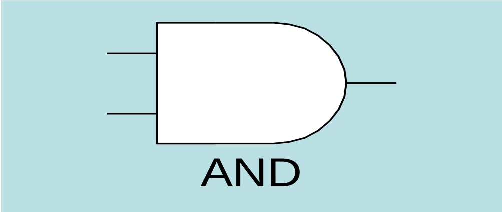


# AND

- Truth if both are true
- Truth table for Boolean AND operator:

| X | Y | X ∧ Y |
|---|---|---|
| F | F | F |
| F | T | F |
| T | F | F |
| T | T | T |

Digital gate diagram for AND:

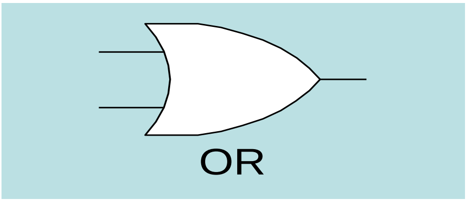


# OR

- True if either is true
- Truth table for Boolean OR operator:

| X | Y | X ∨ Y |
|---|---|---|
| F | F | F |
| F | T | T |
| T | F | T |
| T | T | T |

Digital gate diagram for OR:



# Operator precedence

- NOT > AND > OR
- Examples showing the order of operations:

| Expression | Order of Operations |
|---|---|
| $\neg X \vee Y$ | NOT, then OR |
| $\neg(X \vee Y)$ | OR, then NOT |
| $X \vee (Y \wedge Z)$ | AND, then OR |

- Use parentheses to avoid ambiguity

# Truth Tables (1 of 3)

- A Boolean function has one or more Boolean inputs, and returns a single Boolean output.
- A truth table shows all the inputs and outputs of a Boolean function

Example: $\neg X \vee Y$

| $X$ | $\neg X$ | $Y$ | $\neg X \vee Y$ |
|---|---|---|---|
| F | T | F | T |
| F | T | T | T |
| T | F | F | F |
| T | F | T | T |

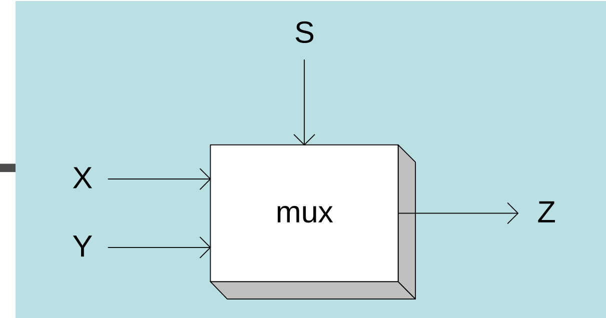
# Truth Tables (2 of 3)

- Example: $X \wedge \neg Y$

| $X$ | $Y$ | $\neg\mathbf{Y}$ | $\mathbf{X} \wedge \neg\mathbf{Y}$ |
|---|---|---|---|
| F | F | T | F |
| F | T | F | F |
| T | F | T | T |
| T | T | F | F |

# Truth Tables (3 of 3)

- Example: $(Y \wedge S) \vee (X \wedge \neg S)$




Two-input multiplexer

| X | Y | S | $Y \wedge S$ | $\neg S$ | $X \wedge \neg S$ | $(Y \wedge S) \vee (X \wedge \neg S)$ |
|---|---|---|---|---|---|---|
| F | F | F | F | T | F | F |
| F | T | F | F | T | F | F |
| T | F | F | F | T | T | T |
| T | T | F | F | T | T | T |
| F | F | T | F | F | F | F |
| F | T | T | T | F | F | T |
| T | F | T | F | F | F | F |
| T | T | T | T | F | F | T |

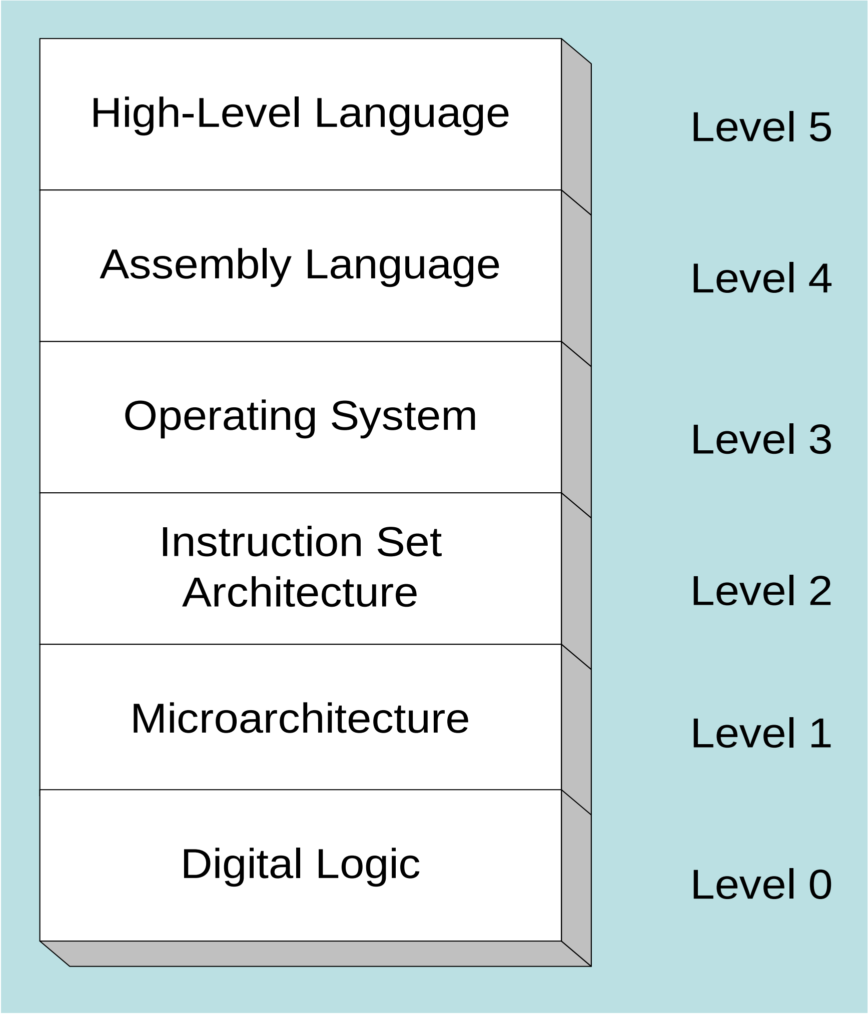
# IA-32 Architecture

***Computer Organization and Assembly Languages***

*with slides by Kip Irvine and Keith Van Rhein*

# Virtual machines

## Abstractions for computers

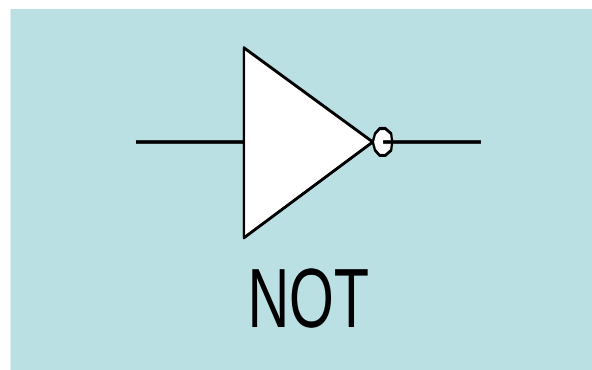


# NOT

- Inverts (reverses) a boolean value
- Truth table for Boolean NOT operator:

| X | ¬X |
|---|---|
| F | T |
| T | F |

Digital gate diagram for NOT:

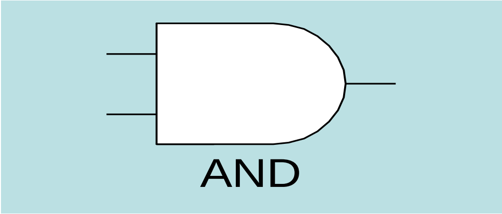


# AND

- Truth if both are true
- Truth table for Boolean AND operator:

| X | Y | X ∧ Y |
|---|---|---|
| F | F | F |
| F | T | F |
| T | F | F |
| T | T | T |

Digital gate diagram for AND:

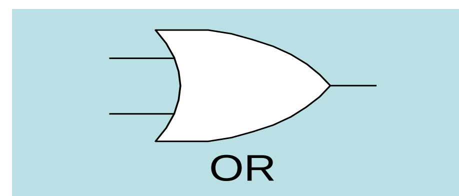


# OR

- True if either is true
- Truth table for Boolean OR operator:

| X | Y | $X \vee Y$ |
|---|---|---|
| F | F | F |
| F | T | T |
| T | F | T |
| T | T | T |

Digital gate diagram for OR:



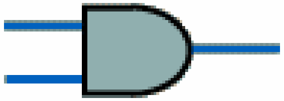
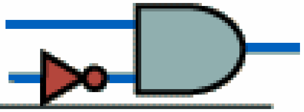
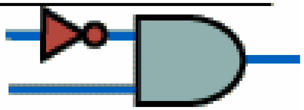
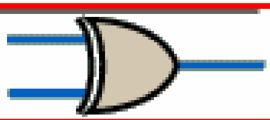
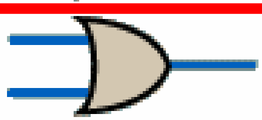
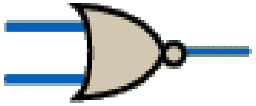
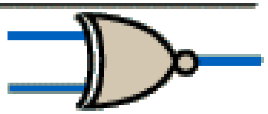
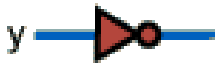
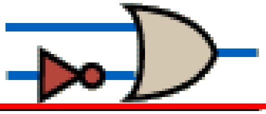
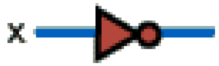
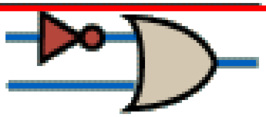
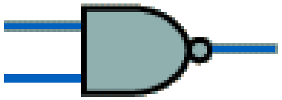
# Truth tables

- A Boolean function has one or more Boolean inputs, and returns a single Boolean output.
- A truth table shows all the inputs and outputs of a Boolean function

Example: $\neg X \vee Y$

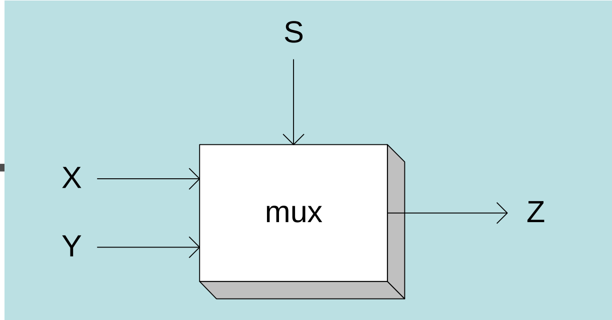
| **X** | **$\neg$X** | **Y** | **$\neg$X $\vee$ Y** |
|---|---|---|---|
| F | T | F | T |
| F | T | T | T |
| T | F | F | F |
| T | F | T | T |

# All possible 2-input Boolean functions

| | | |
|---|---|---|
| 0 0 0 0 | 0 |  |
| 0 0 0 1 | AND |  |
| 0 0 1 0 | xy' |  |
| 0 0 1 1 | x |  |
| 0 1 0 0 | x'y |  |
| 0 1 0 1 | y |  |
| 0 1 1 0 | XOR |  |
| 0 1 1 1 | OR |  |
| 1 0 0 0 | NOR |  |
| 1 0 0 1 | XNOR |  |
| 1 0 1 0 | y' |  |
| 1 0 1 1 | x + y' |  |
| 1 1 0 0 | x' |  |
| 1 1 0 1 | x'+y |  |
| 1 1 1 0 | NAND |  |
| 1 1 1 1 | 1 |  |

# Truth tables

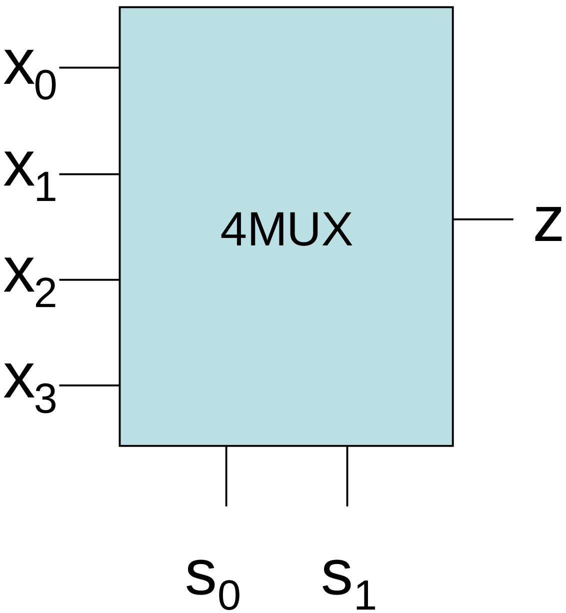
- Example: $(Y \wedge S) \vee (X \wedge \neg S)$

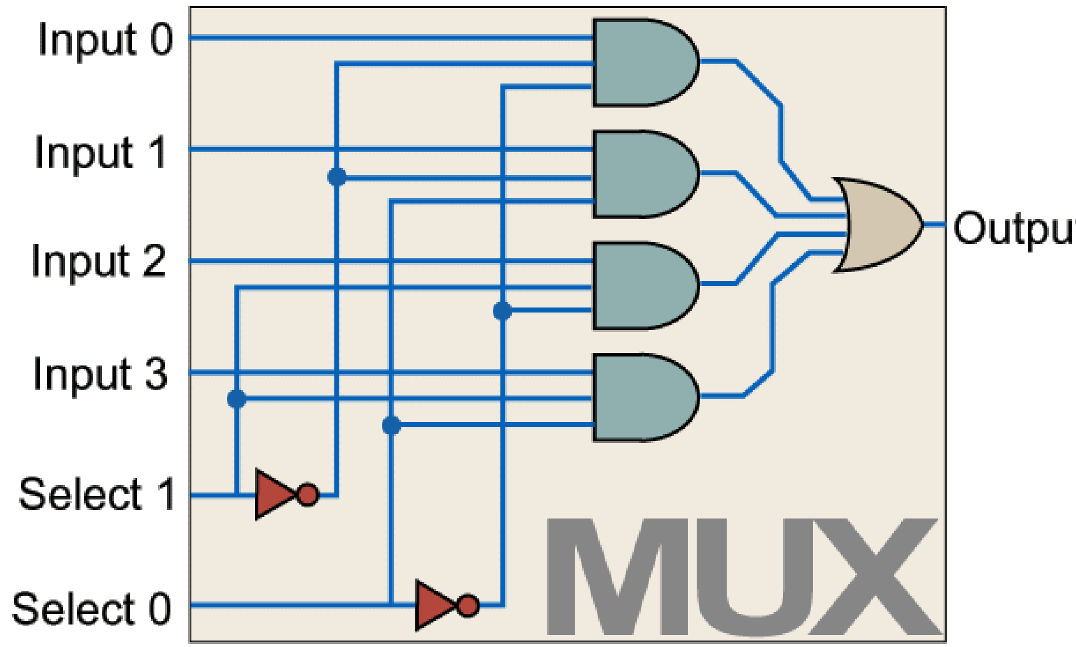



Two-input multiplexer

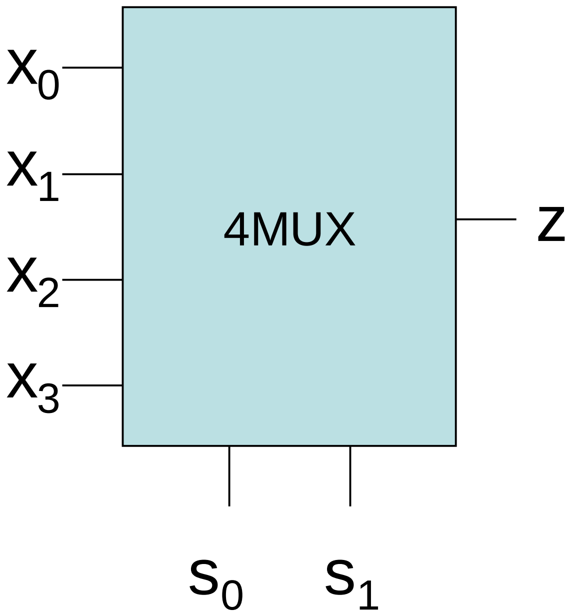
| X | Y | S | $Y \wedge S$ | $\neg S$ | $X \wedge \neg S$ | $(Y \wedge S) \vee (X \wedge \neg S)$ |
|---|---|---|---|---|---|---|
| F | F | F | F | T | F | F |
| F | T | F | F | T | F | F |
| T | F | F | F | T | T | T |
| T | T | F | F | T | T | T |
| F | F | T | F | F | F | F |
| F | T | T | T | F | F | T |
| T | F | T | F | F | F | F |
| T | T | T | T | F | F | T |

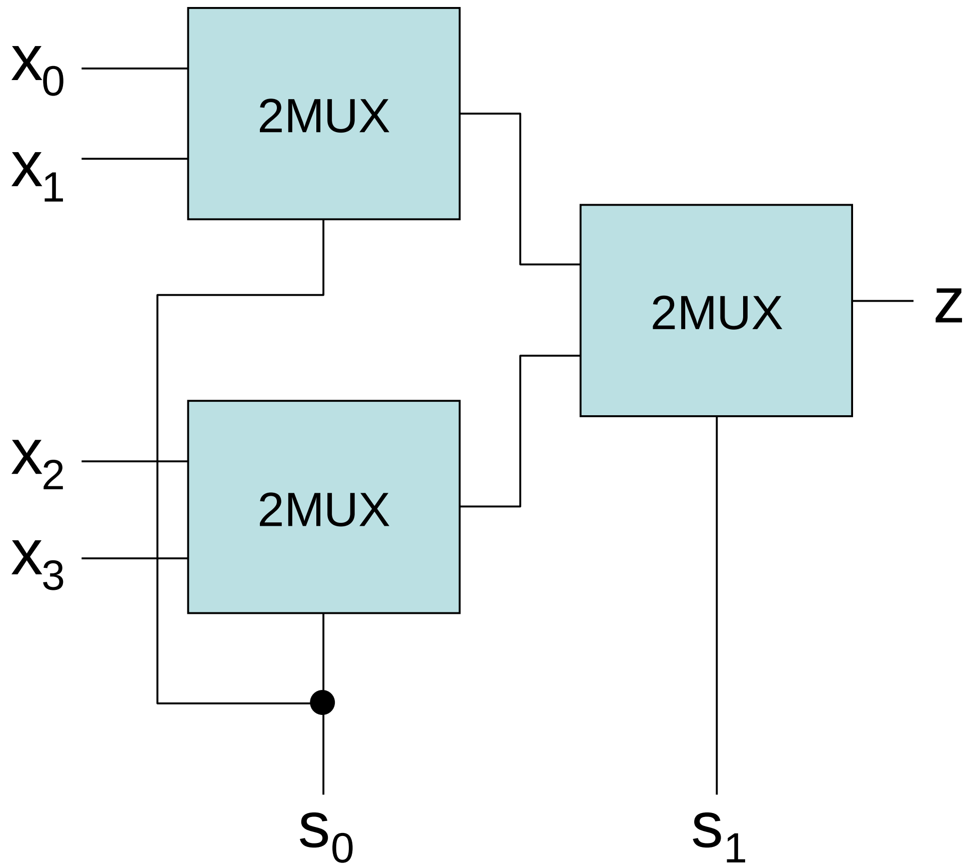
# 4-multiplexer



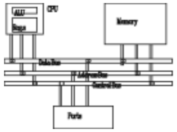



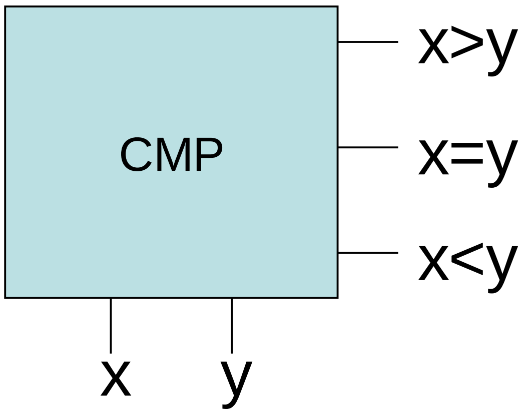
# 4-multiplexer



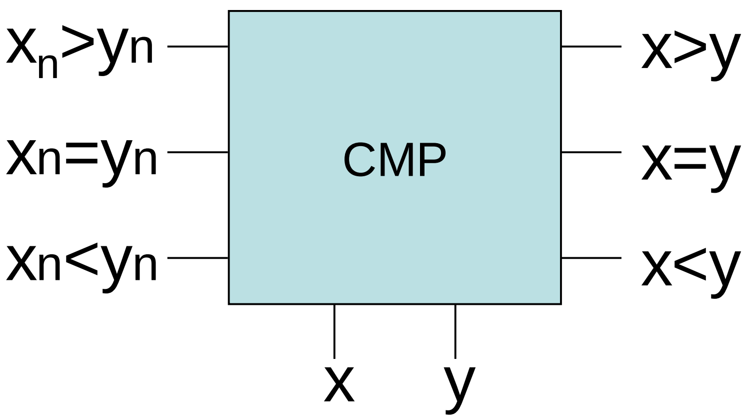
# Comparator

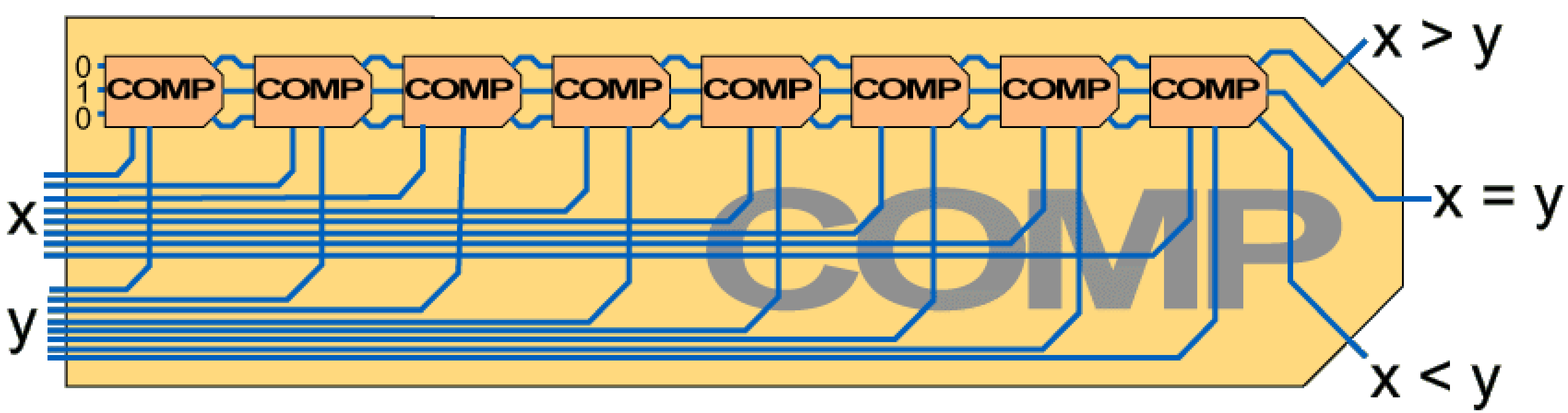




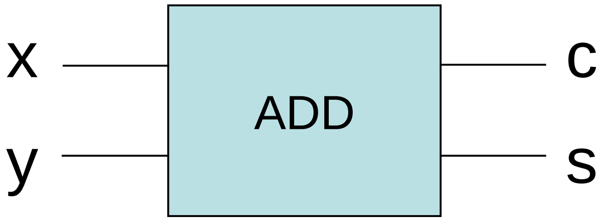

| x | y | x>y | x=y | x<y |
|---|---|---|---|---|
| | | | | |
| | | | | |
| | | | | |
| | | | | |

# 8-bit comparator

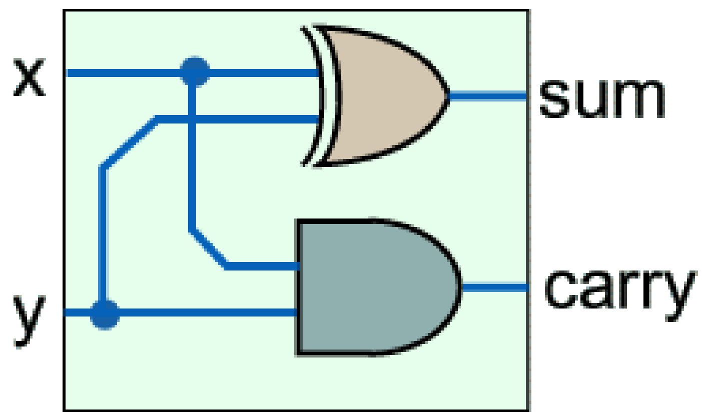


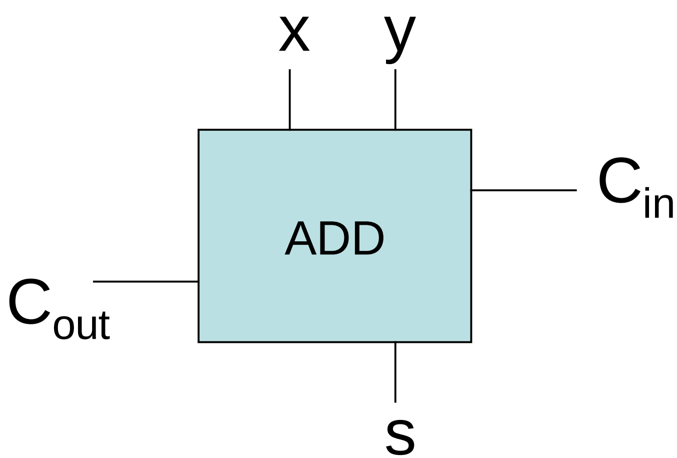
# 1-bit half adder

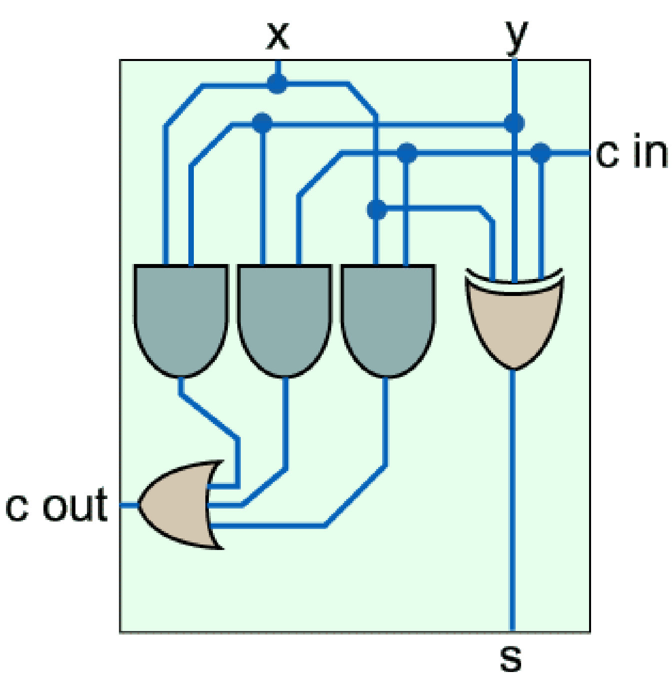



| x | y | s | c |
|---|---|---|---|
| | | | |
| | | | |
| | | | |
| | | | |



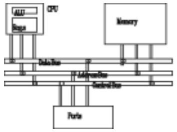
# 1-bit full adder

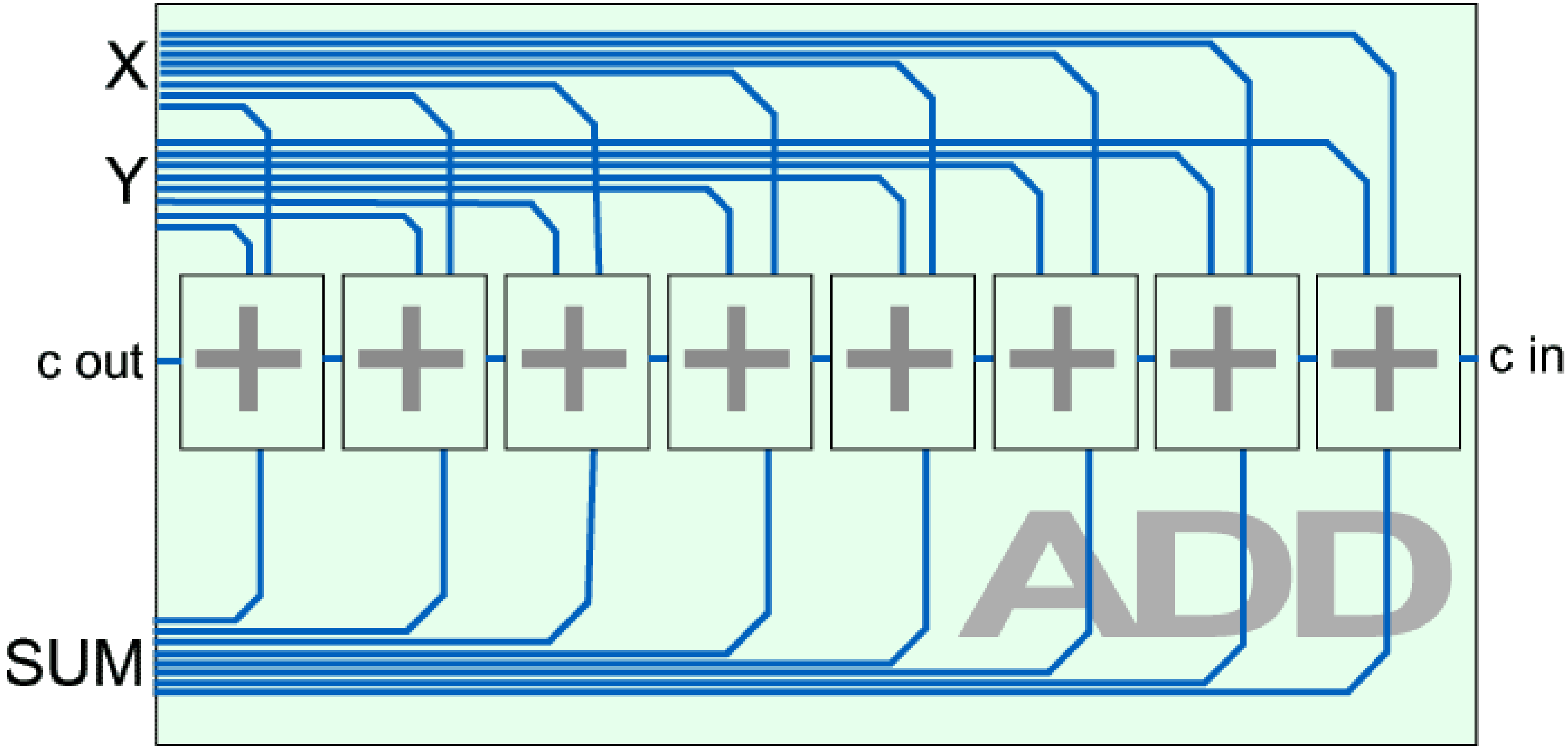


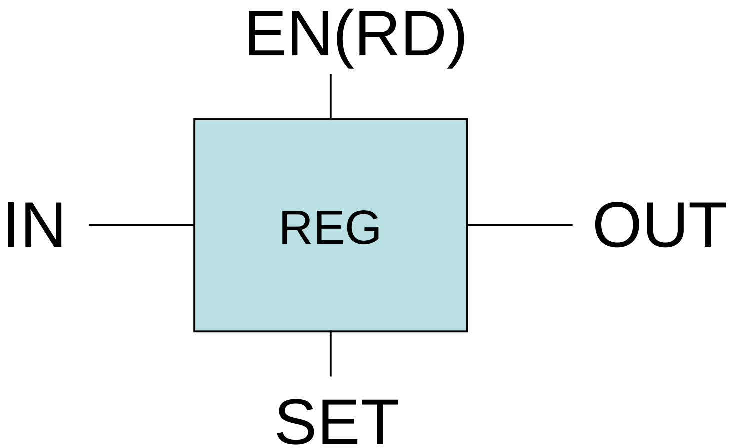

| x | y | $C_{in}$ | $C_{out}$ | s |
|---|---|---|---|---|
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |

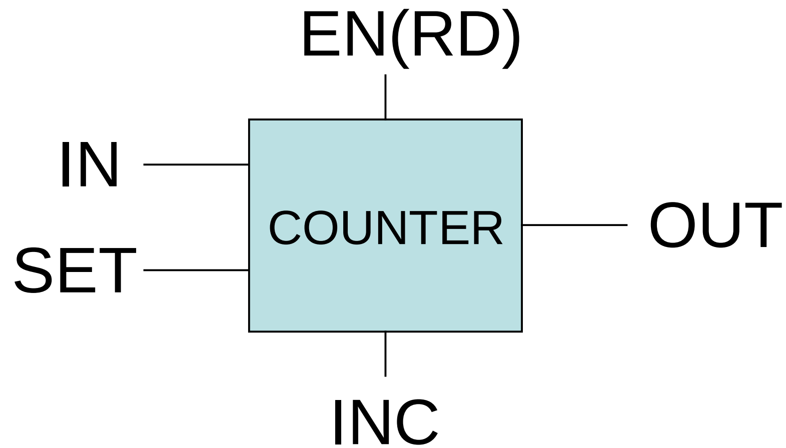
# 8-bit adder

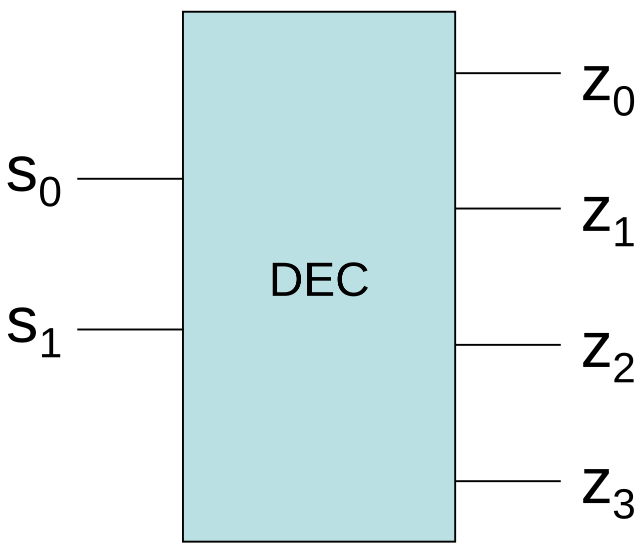




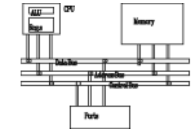
# Registers and counters

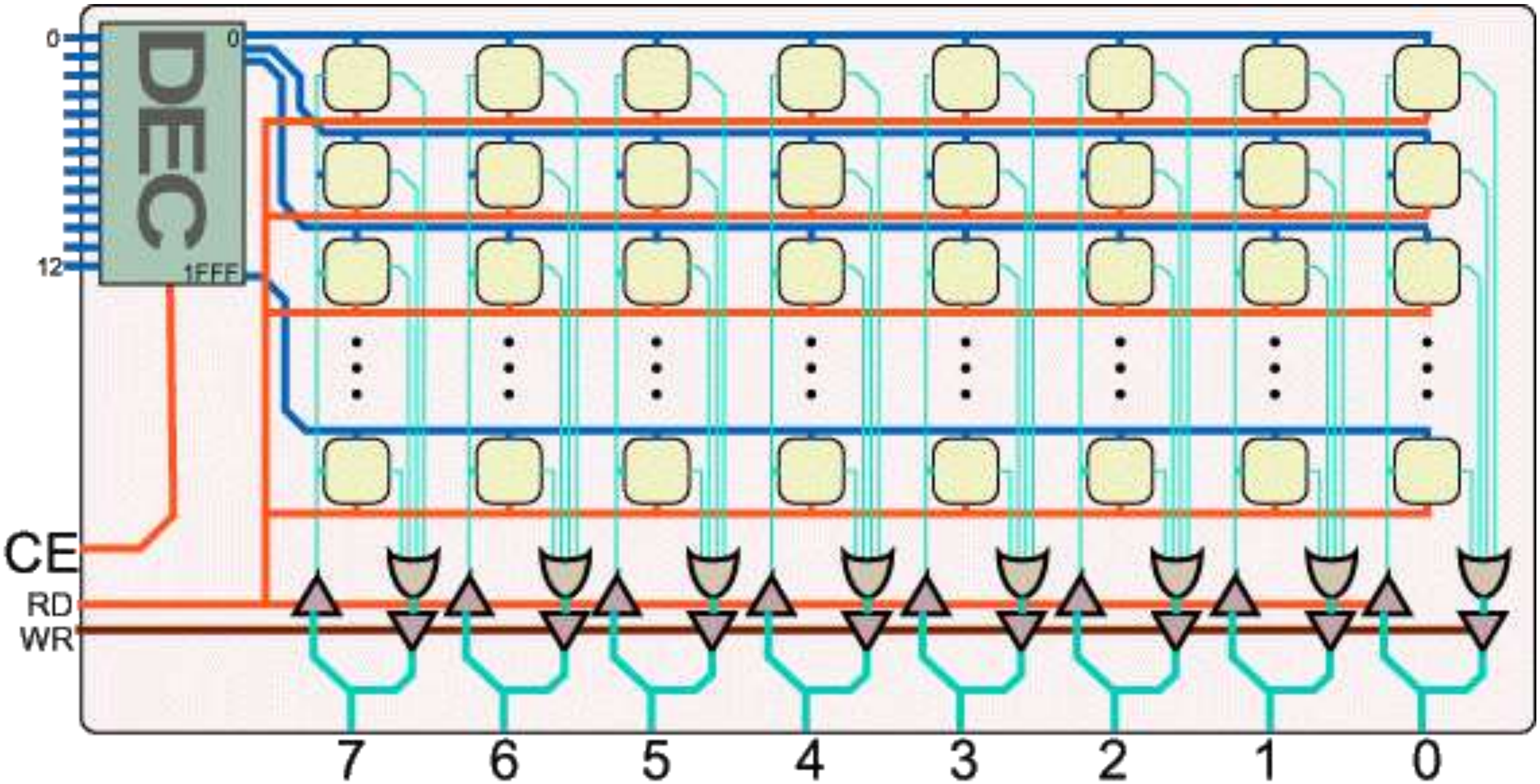


# Memory



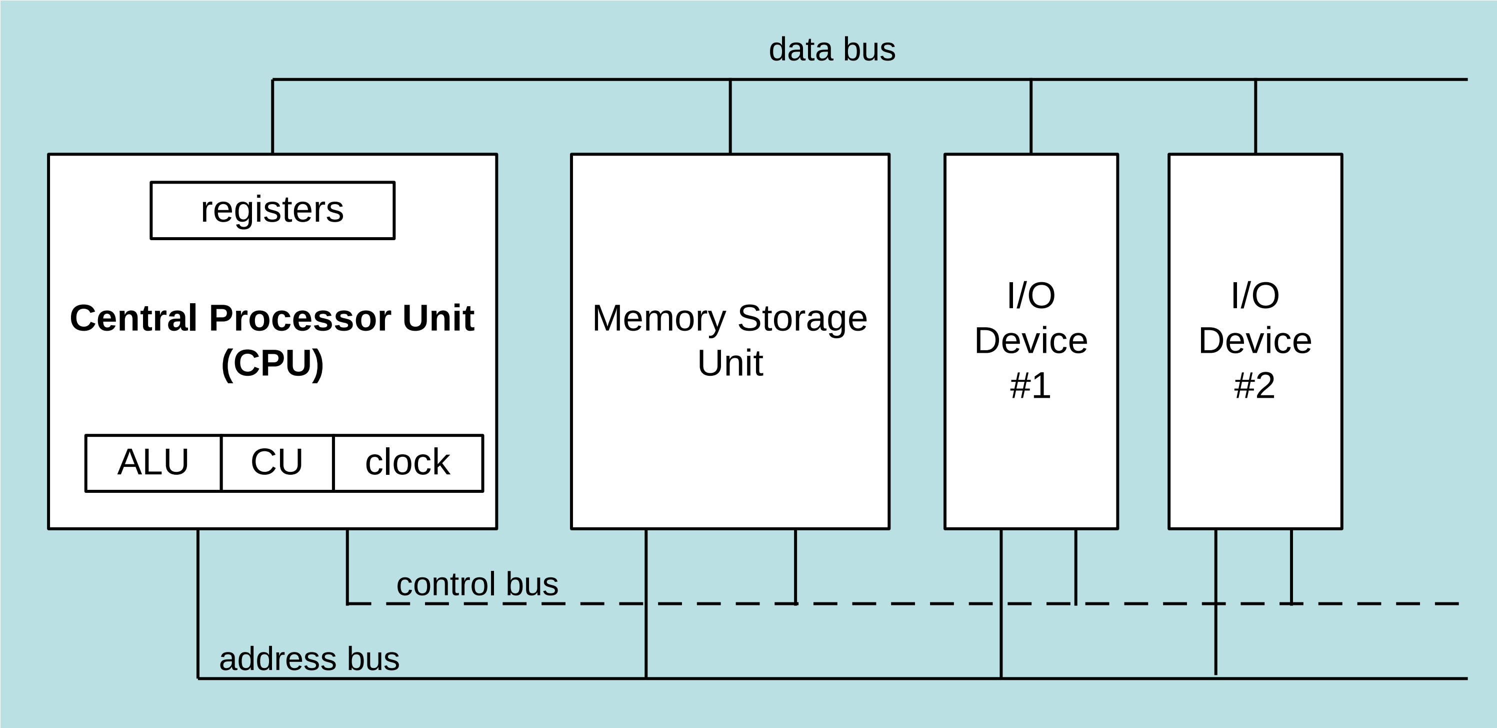



8K 8-bit memory

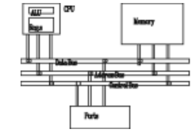
# Microcomputer concept

# Basic microcomputer design

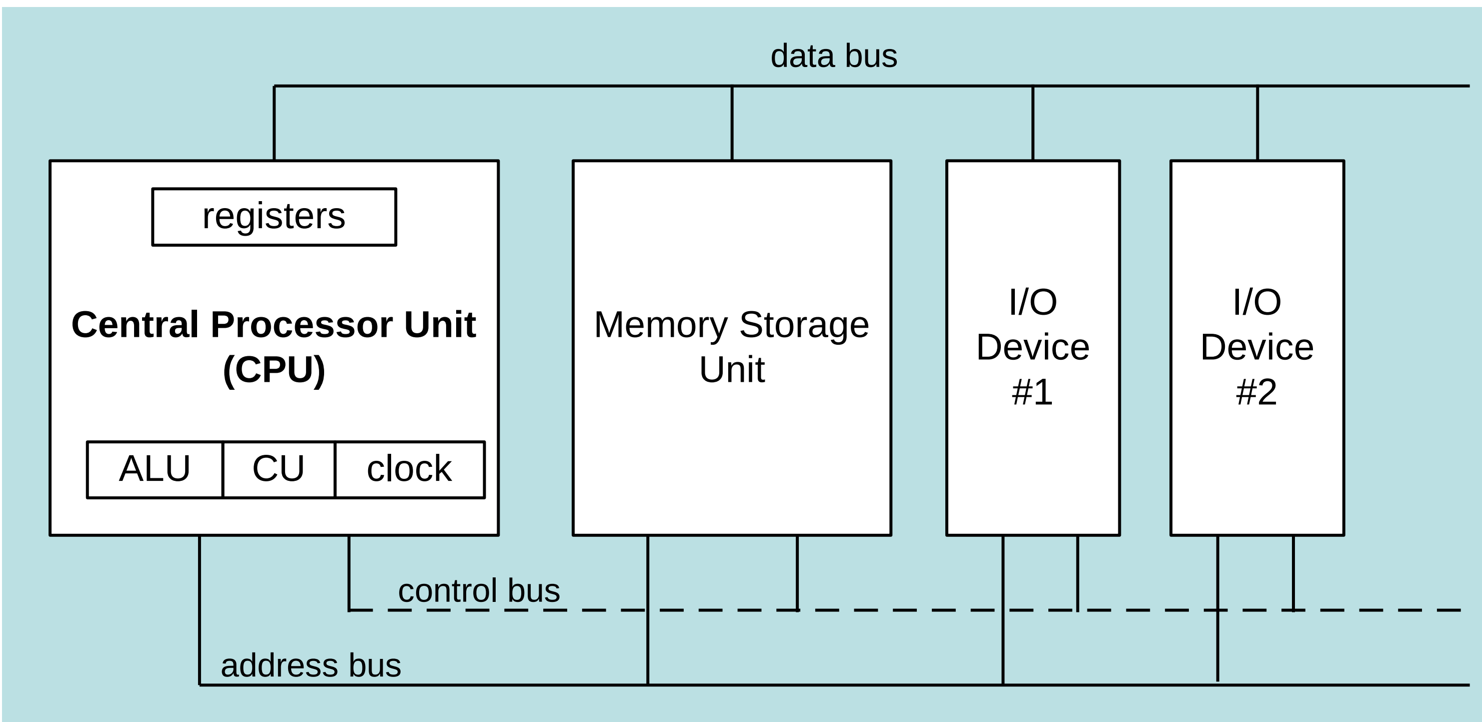
- clock synchronizes CPU operations
- control unit (CU) coordinates sequence of execution steps
- ALU performs arithmetic and logic operations



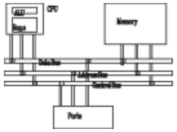
# Basic microcomputer design



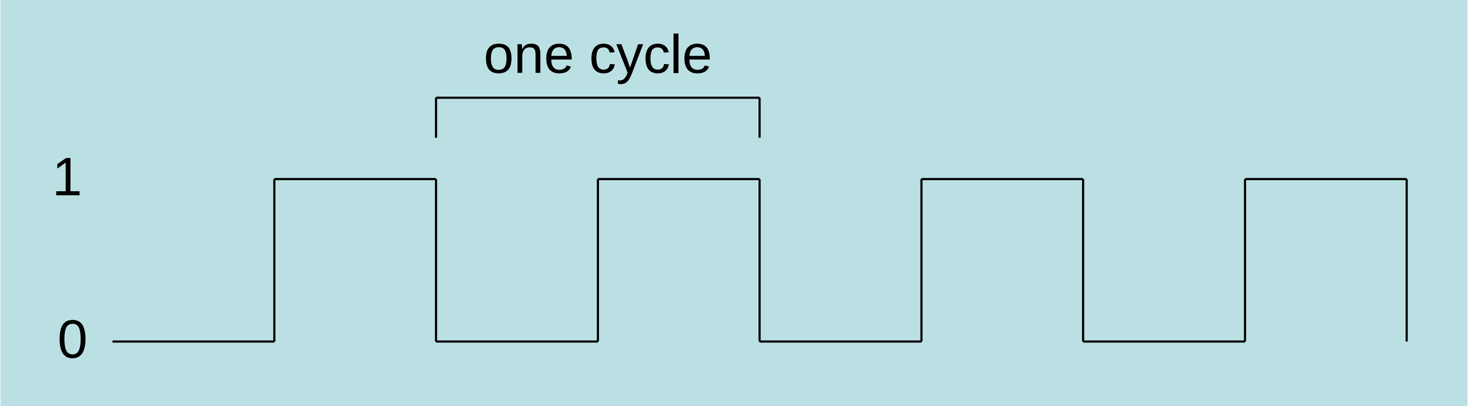
- The memory storage unit holds instructions and data for a running program
- A bus is a group of wires that transfer data from one part to another (data, address, control)



# Clock

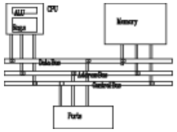


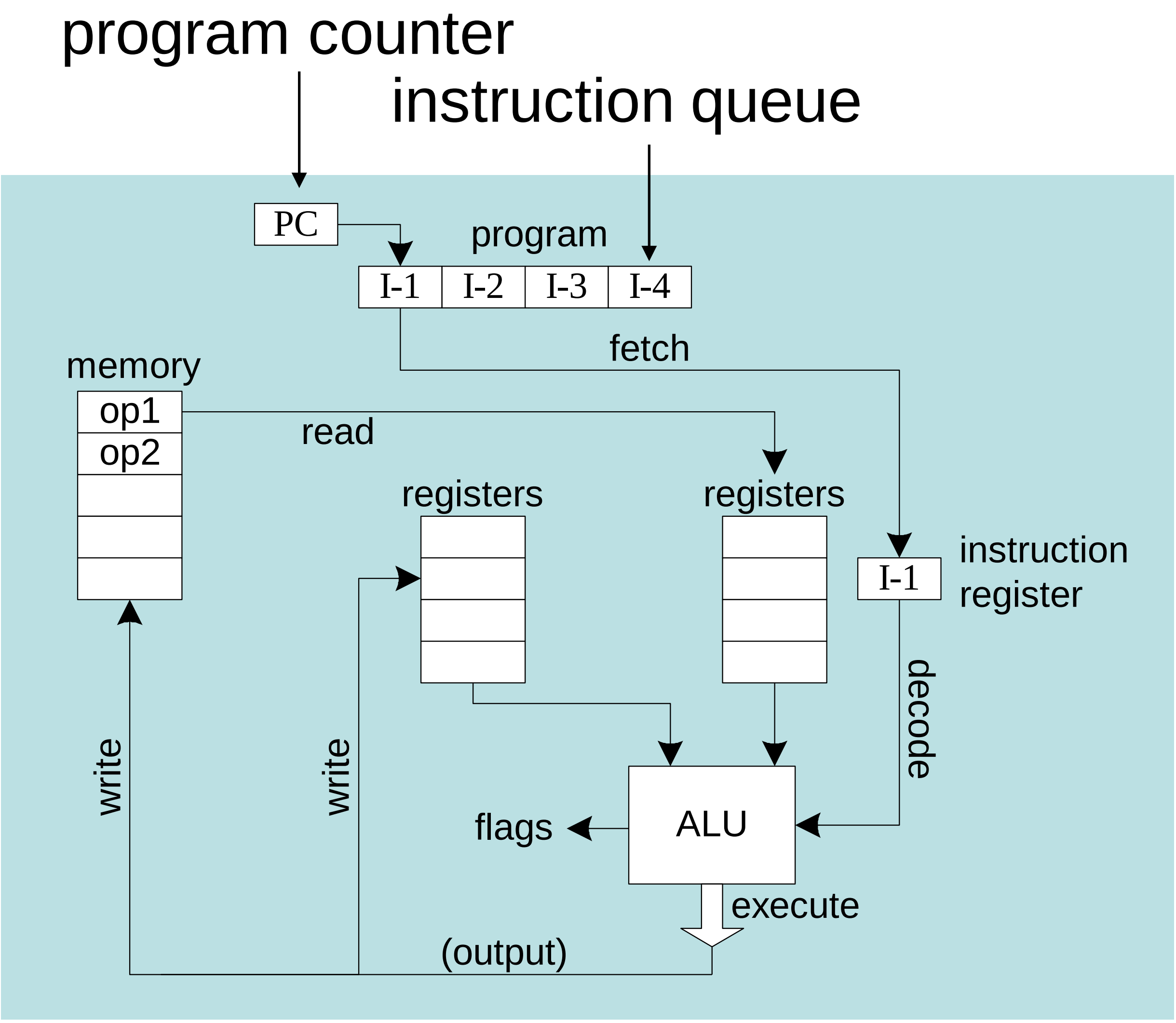
- synchronizes all CPU and BUS operations
- machine (clock) cycle measures time of a single operation
- clock is used to trigger events




- Basic unit of time, 1GHz→clock cycle=1ns
- A instruction could take multiple cycles to complete, e.g. multiply in 8088 takes 50 cycles

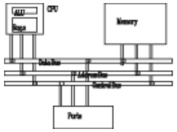
# Instruction execution cycle

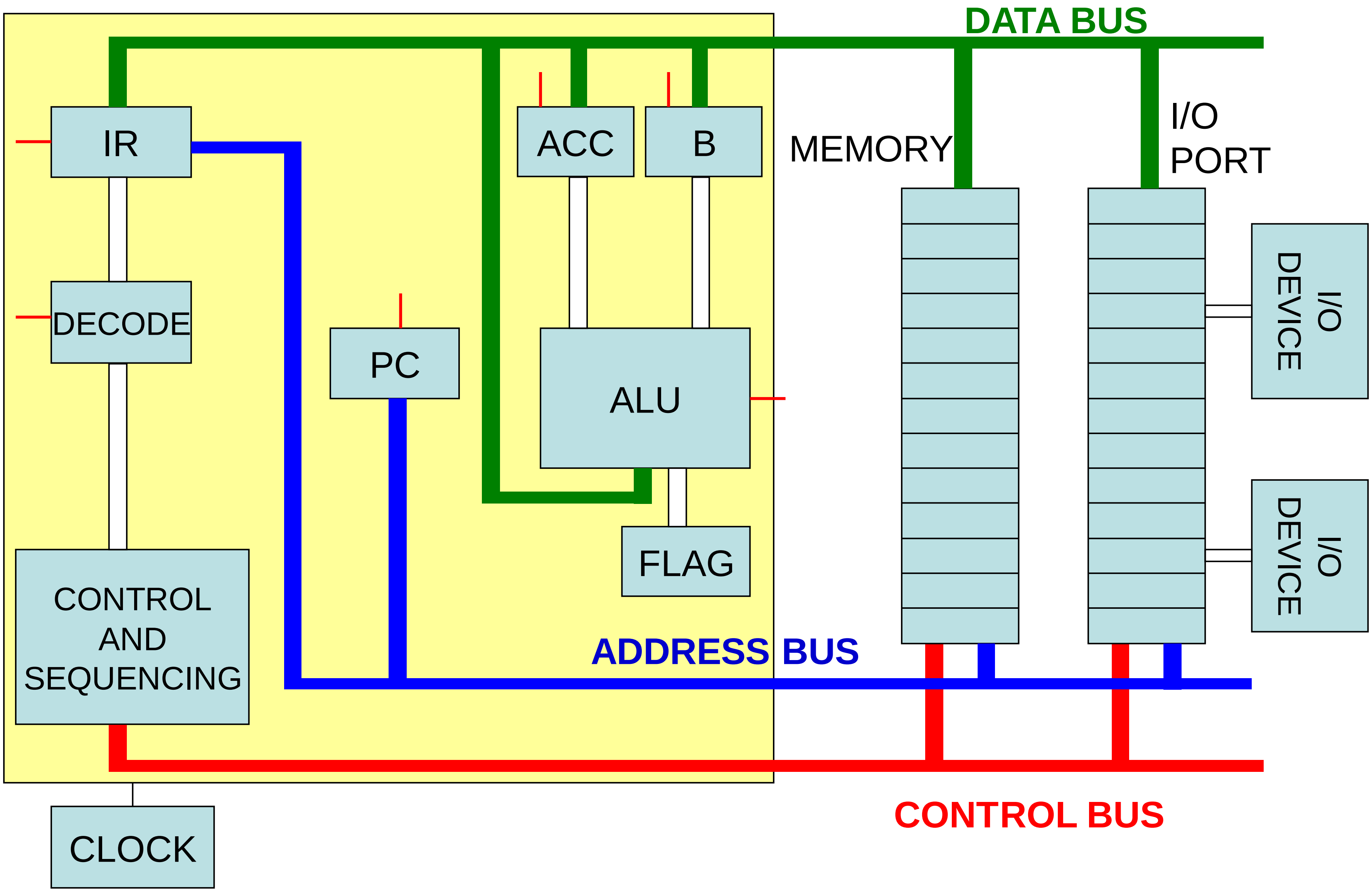




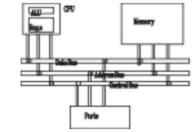

- **Fetch**
- **Decode**
- Fetch operands
- **Execute**
- Store output

# A simple microcomputer





# Instruction set



| OPCODE | MNEMONIC | OPCODE | MNEMONIC |
|---|---|---|---|
| 0 | NOP | A | CMP |
| 1 | LDA | B | JG |
| 2 | STA | C | JE |
| 3 | ADD | D | JL |
| 4 | SUB | | |
| 5 | IN | | |
| 6 | OUT | | |
| 7 | JMP | | |
| 8 | JN | | |
| 9 | HLT | | |

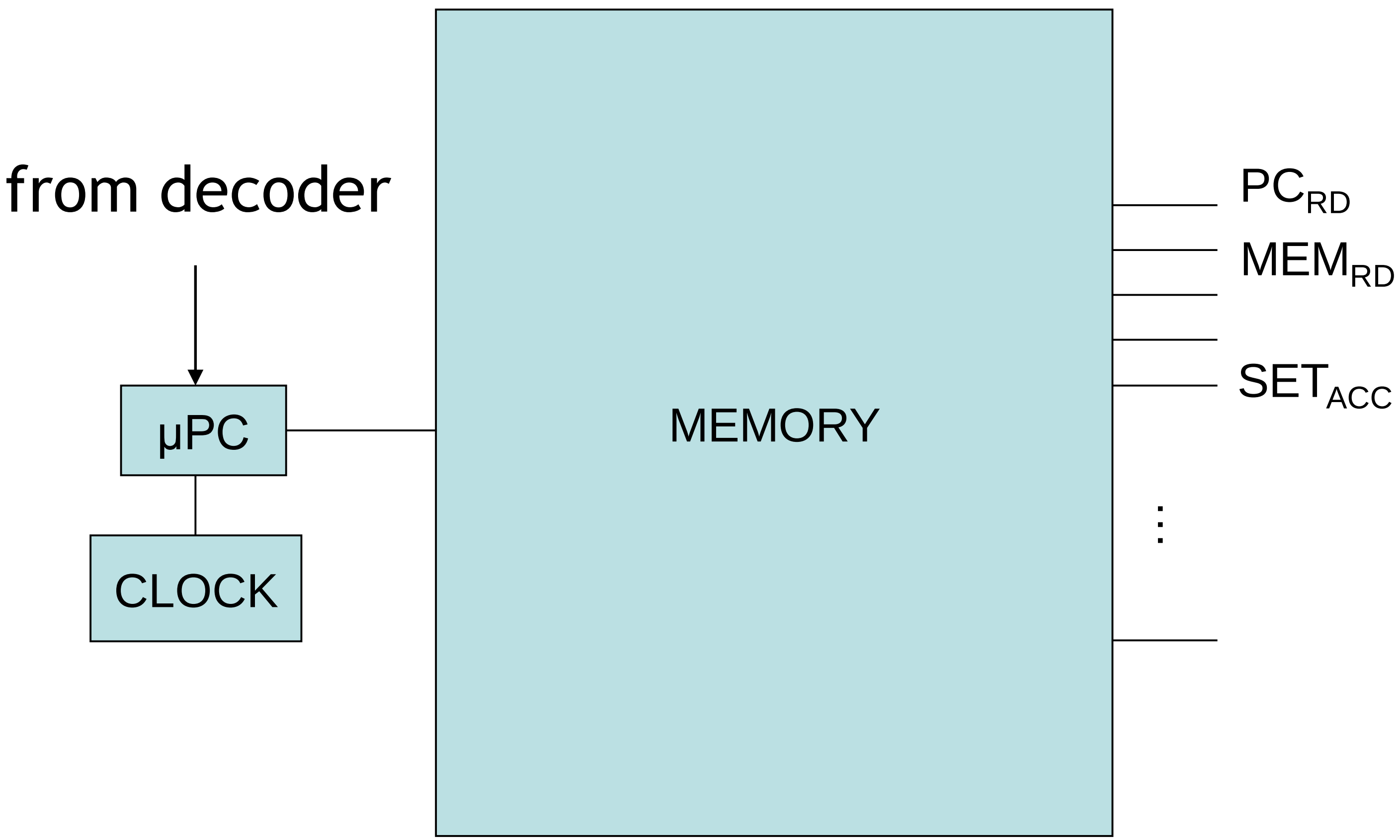
| OPCODE | OPERAND |
|---|---|
| 4 | 12 |

# Control bus

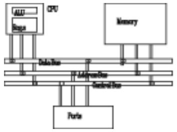
- A series of control signals to control all components such as registers and ALU
- Control signal for load ACC:

  $SET_{ACC}$=1, others=0

# Control and sequencing unit

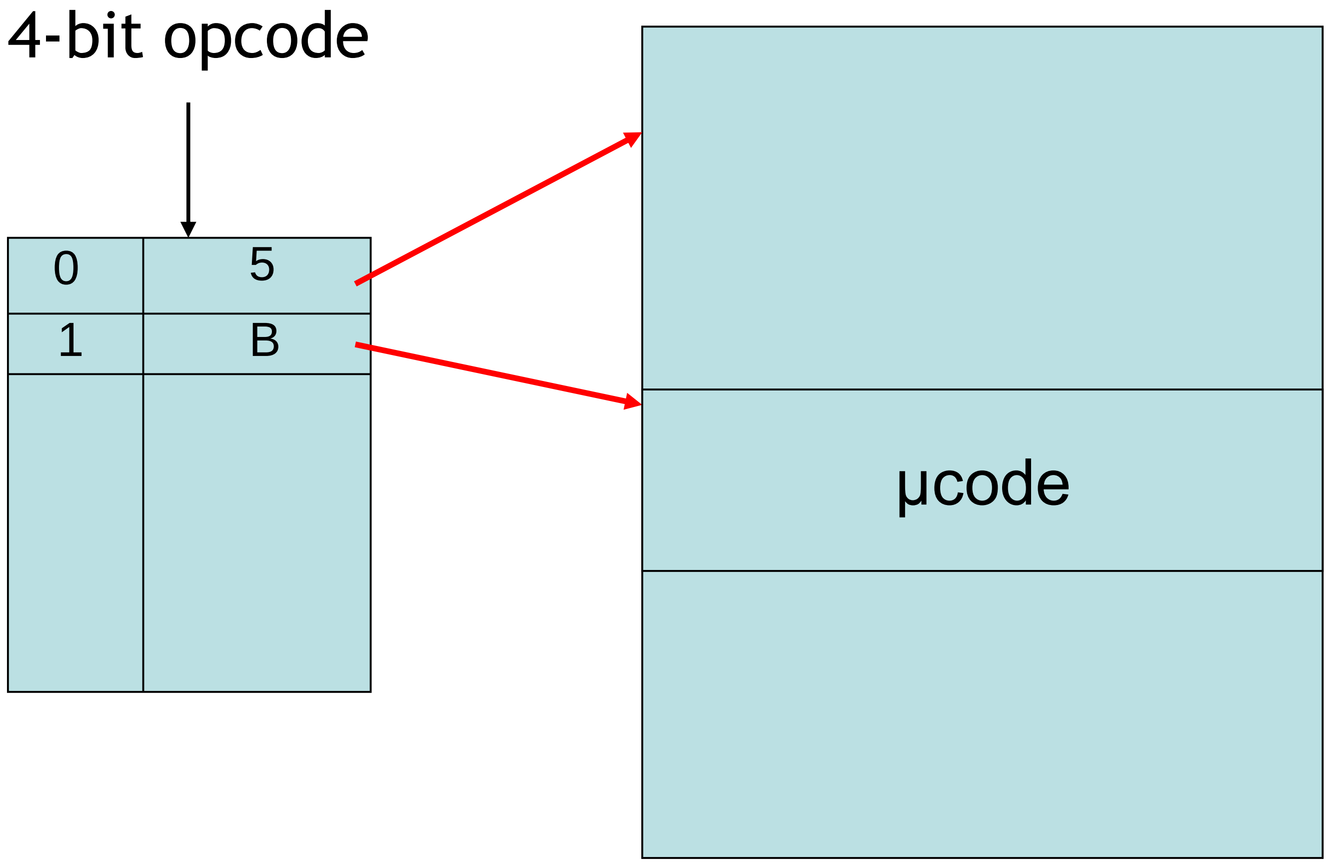


# Control and sequencing unit



| | | $PC_{RD}$ | $MEM_{RD}$ | $MEM_{WT}$ | $IR_{SET}$ | …. |
|---|---|---|---|---|---|---|
| fetch | 0000 | 1 | 0 | 0 | 0 | 0…. |
| | 0001 | 0 | 1 | 0 | 0 | |
| | 0002 | 0 | 0 | 0 | 1 | |
| decode | 0003 | 4-bit IR RD | | | | |
| | 0004 | DECODER RD, μPC SET | | | | |
| exec | 0005 | | | | | |
| fetch decode | | | | | | |
| | 000B | | | | | |

# Decoder



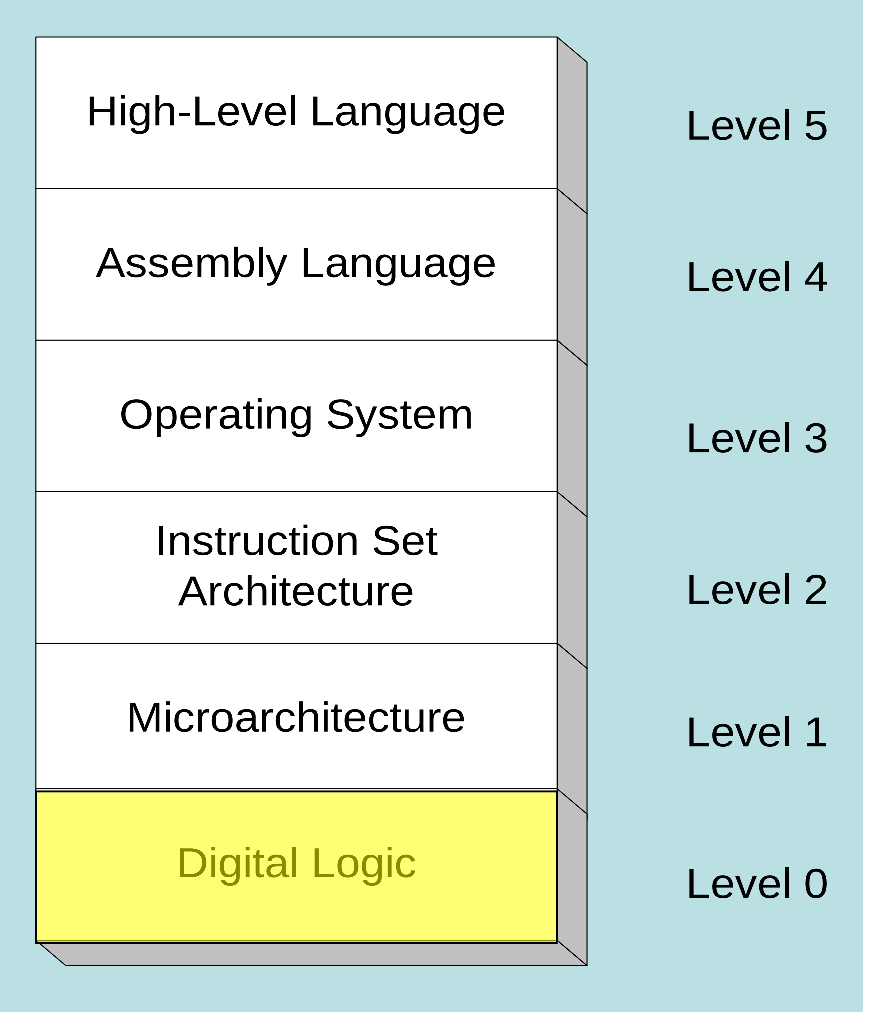
# IA-32 Architecture

***Computer Organization and Assembly Languages***

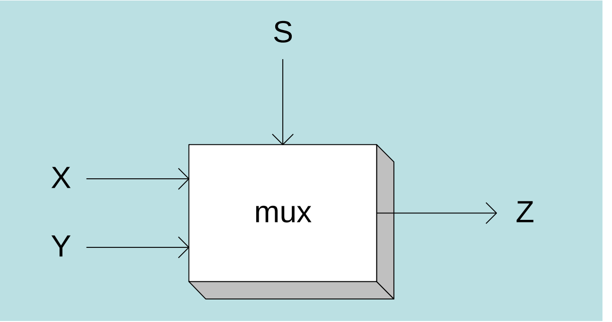
*with slides by Kip Irvine and Keith Van Rhein*

# Virtual machines

## Abstractions for computers



# Truth tables

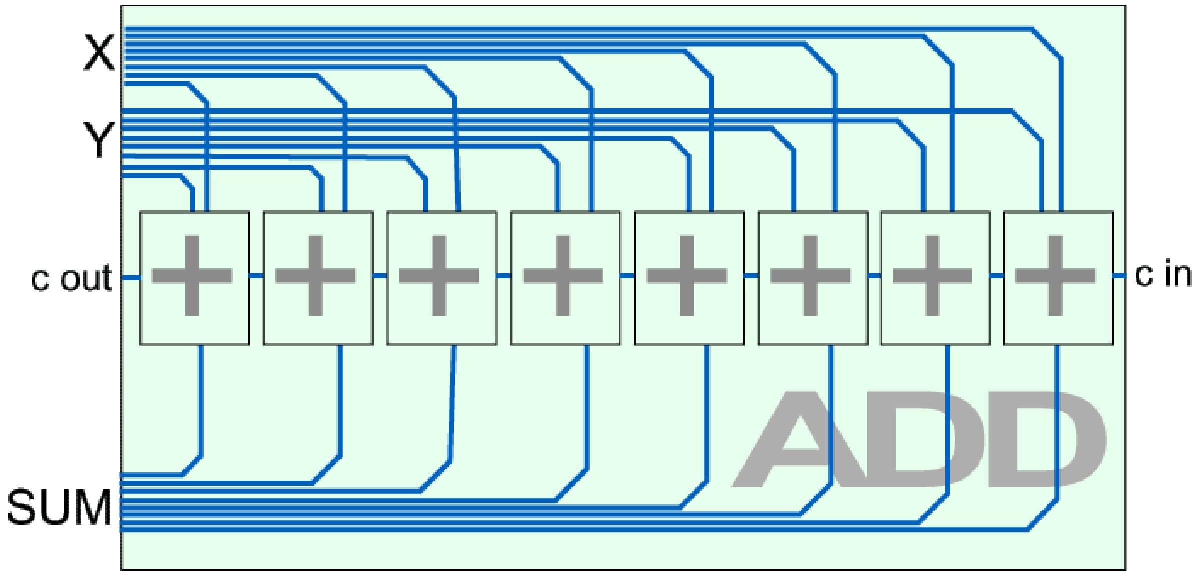


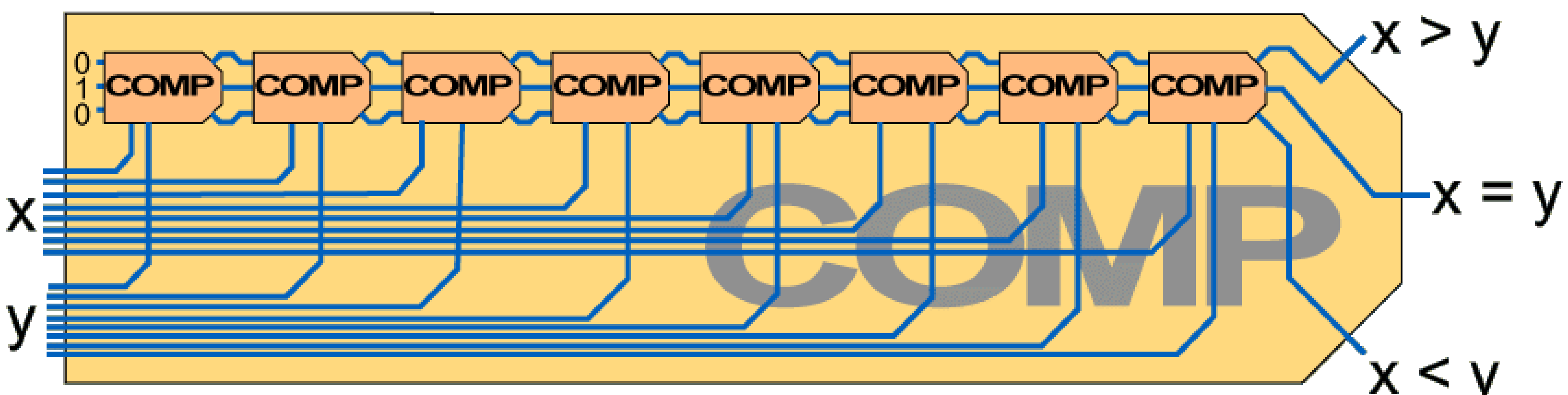

Two-input multiplexer

- Example: $(Y \wedge S) \vee (X \wedge \neg S)$

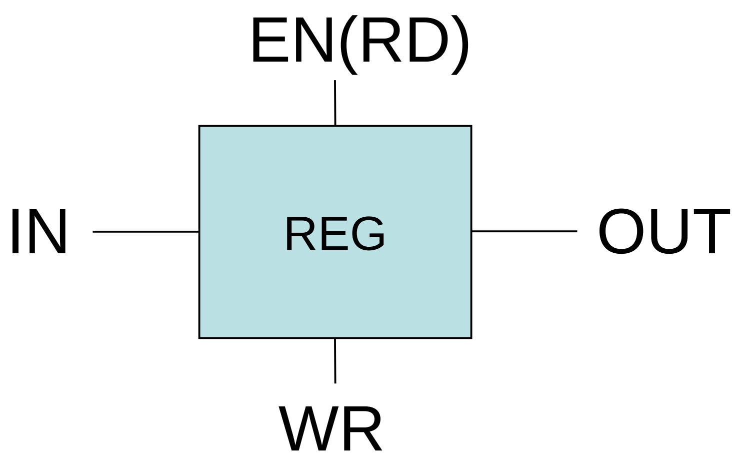
| X | Y | S | $(Y \wedge S) \vee (X \wedge \neg S)$ |
|---|---|---|---|
| F | F | F | F |
| F | T | F | F |
| T | F | F | T |
| T | T | F | T |
| F | F | T | F |
| F | T | T | T |
| T | F | T | F |
| T | T | T | T |

# Combinational logic

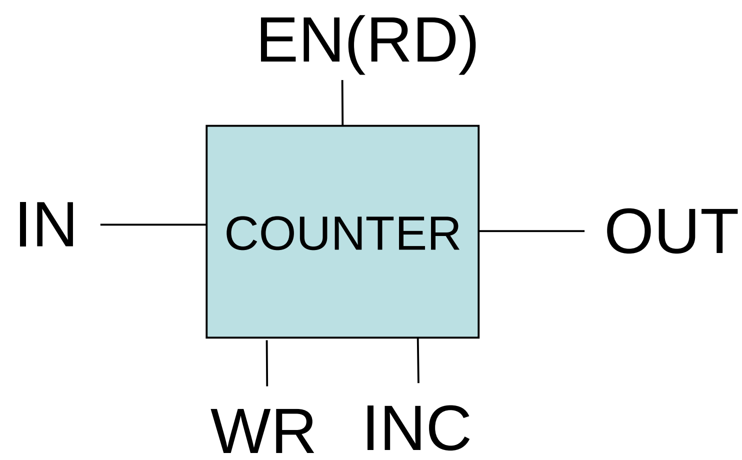


# Sequential logic

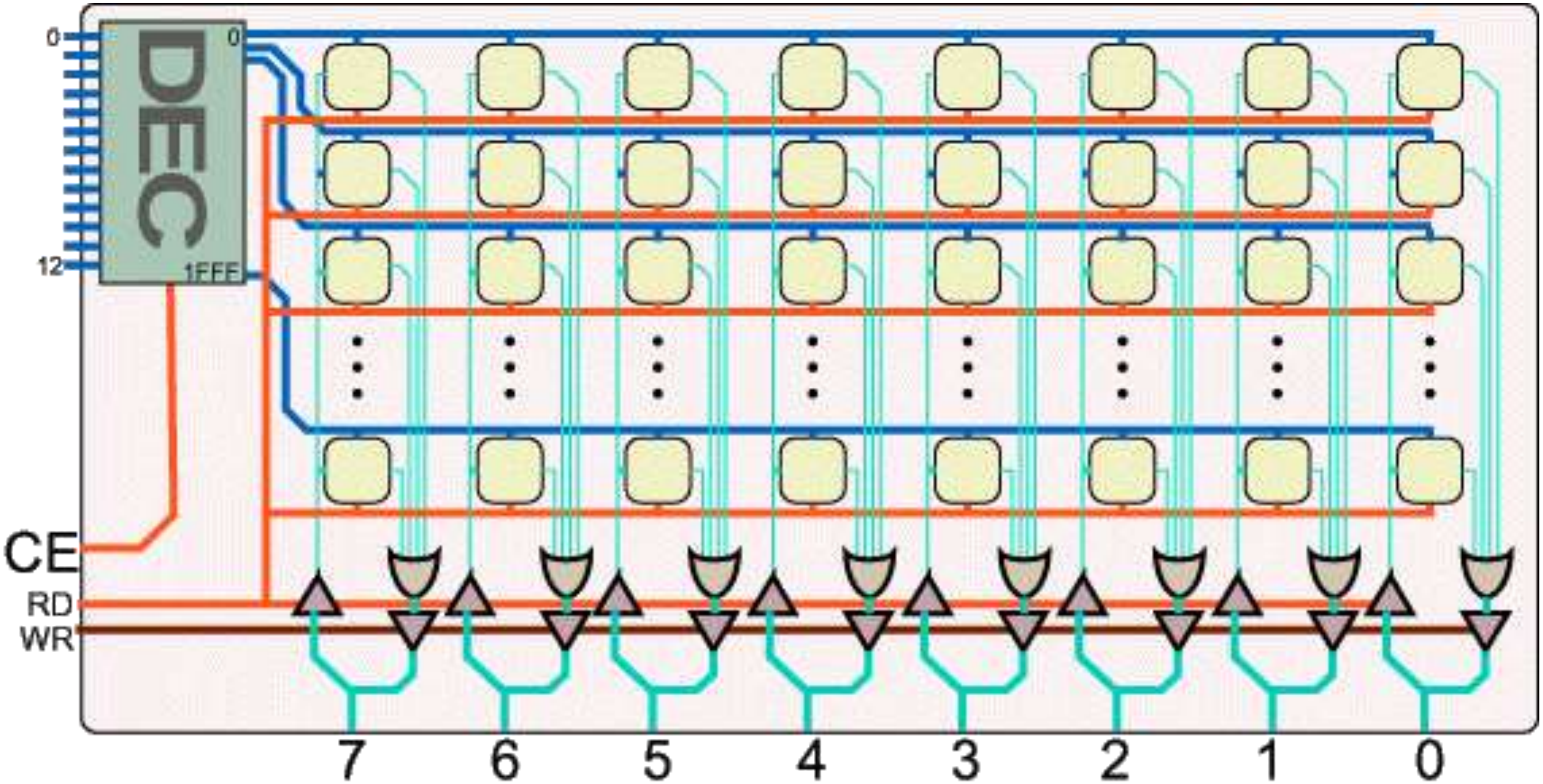



register




counter

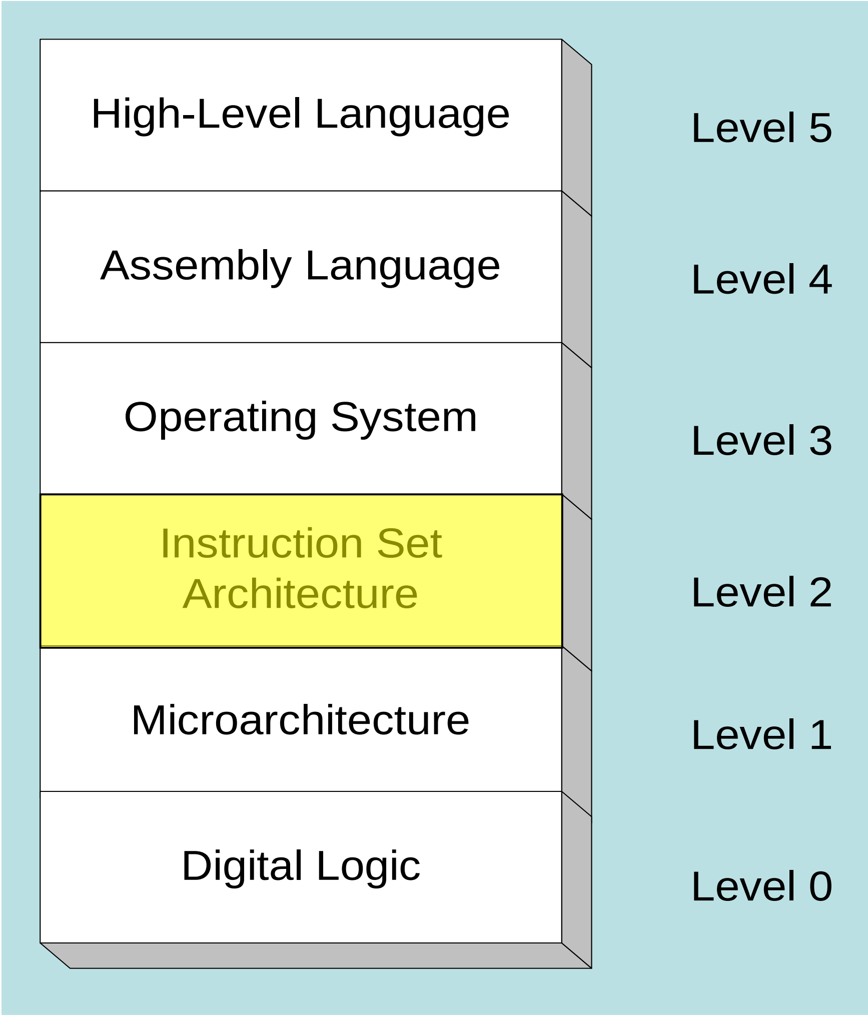
# Memory



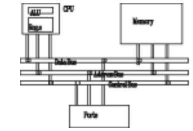

8K 8-bit memory

# Virtual machines

## Abstractions for computers



# Instruction set

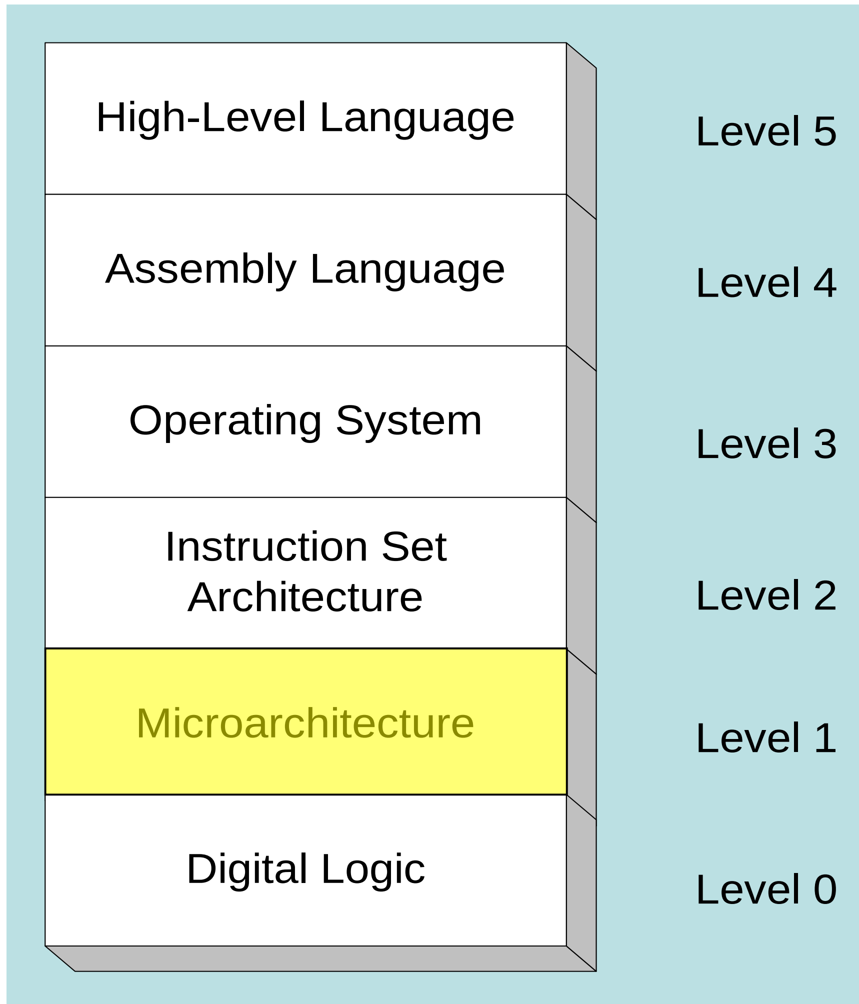


| OPCODE | MNEMONIC | OPCODE | MNEMONIC |
|---|---|---|---|
| 0 | NOP | A | CMP addr |
| 1 | LDA addr | B | JG addr |
| 2 | STA addr | C | JE addr |
| 3 | ADD addr | D | JL addr |
| 4 | SUB addr | | |
| 5 | IN port | | |
| 6 | OUT port | | |
| 7 | JMP addr | | |
| 8 | JN addr | | |
| 9 | HLT | | |

| OPCODE | OPERAND |
|---|---|
| 4 | 12 |

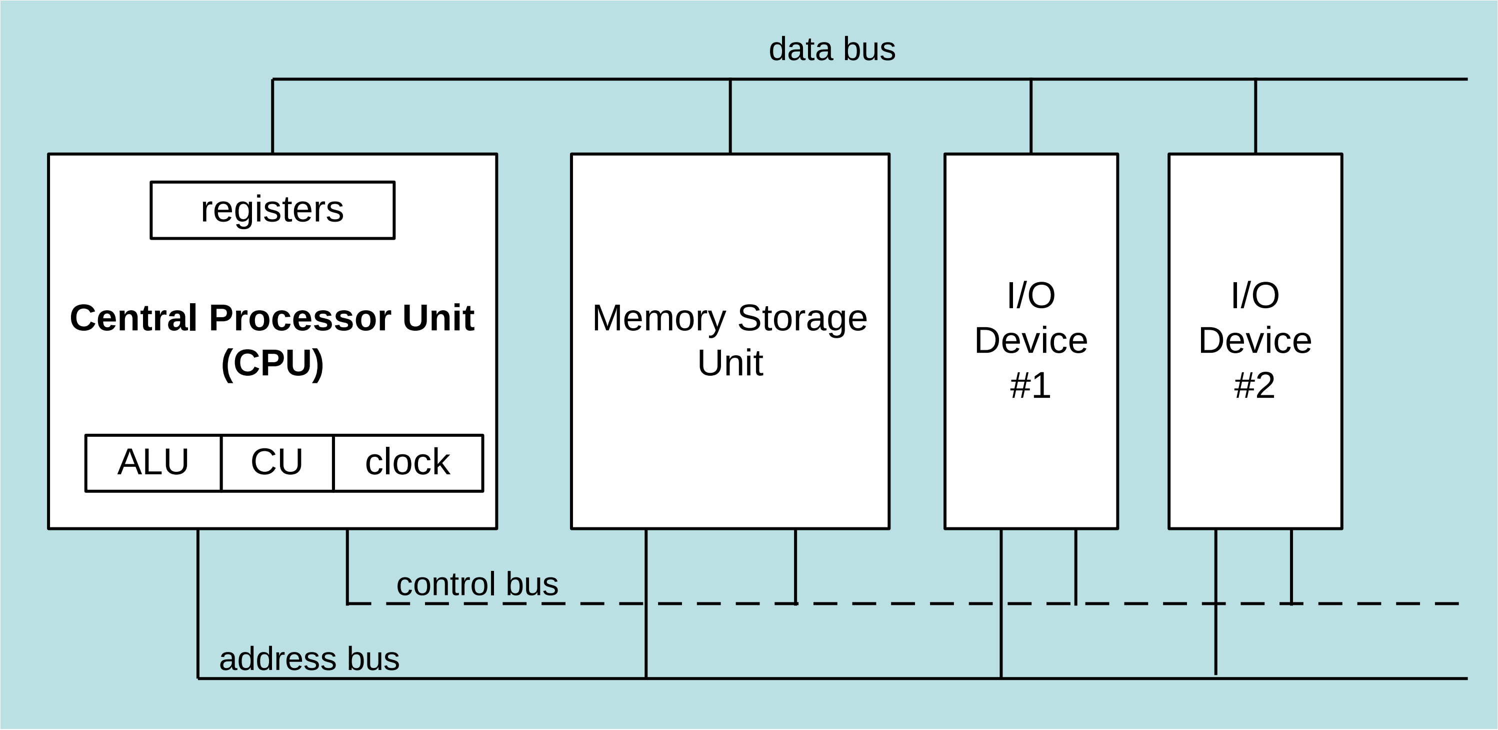
# Virtual machines

## Abstractions for computers



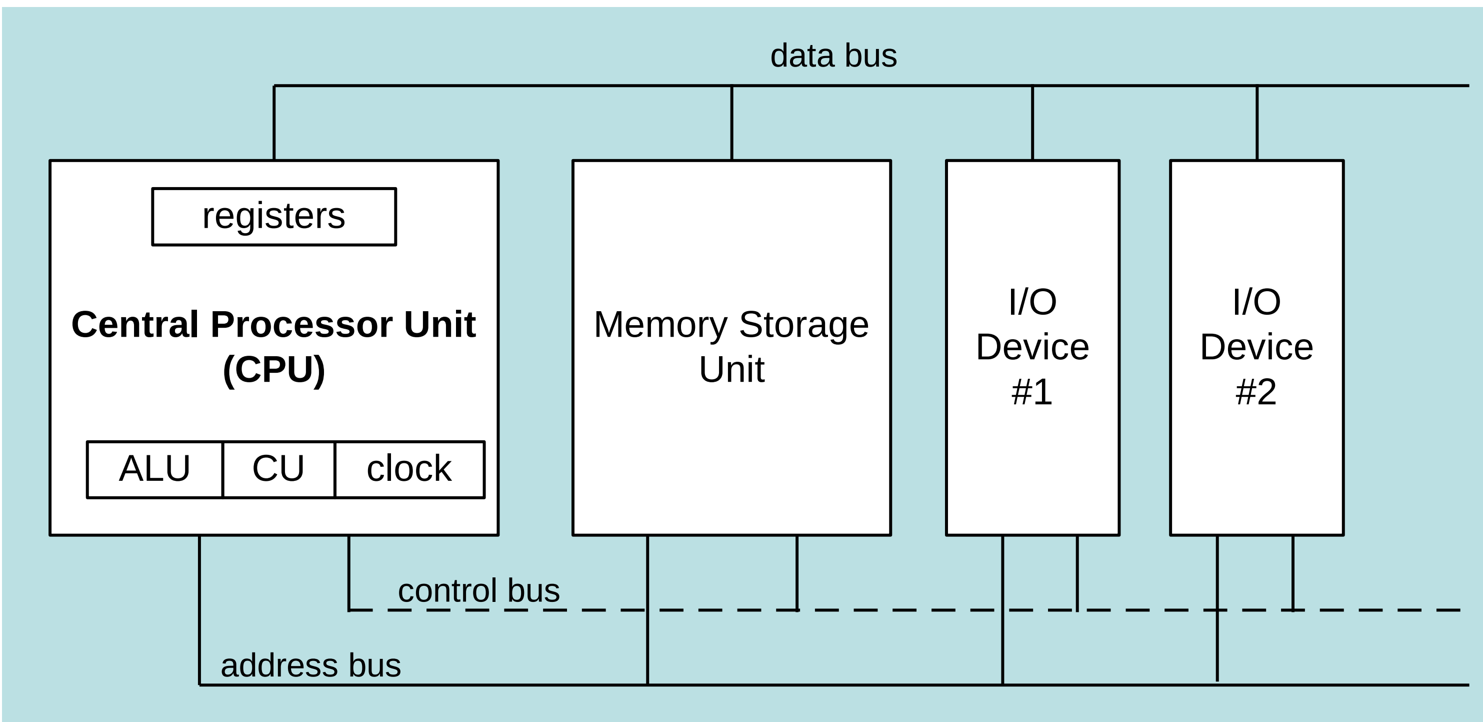
# Basic microcomputer design

- clock synchronizes CPU operations
- control unit (CU) coordinates sequence of execution steps
- ALU performs arithmetic and logic operations

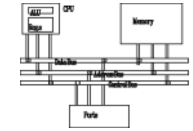


# Basic microcomputer design

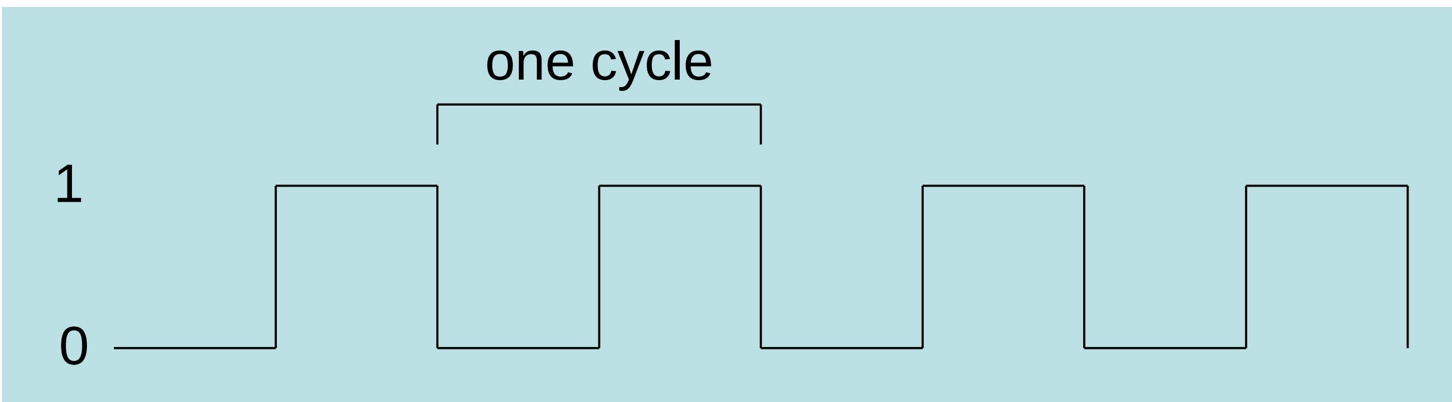
- The memory storage unit holds instructions and data for a running program
- A bus is a group of wires that transfer data from one part to another (data, address, control)



# Clock

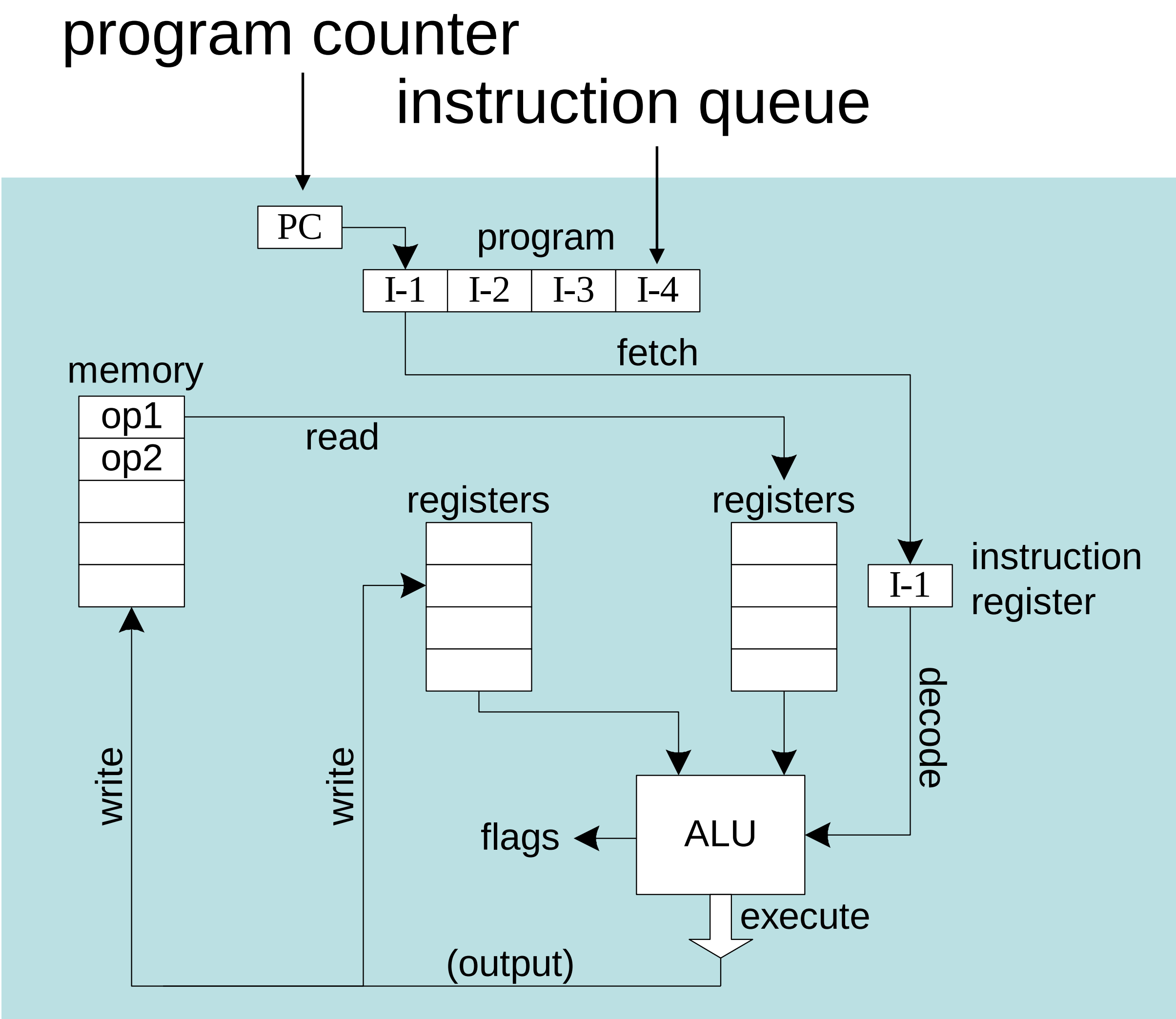


- synchronizes all CPU and BUS operations
- machine (clock) cycle measures time of a single operation
- clock is used to trigger events



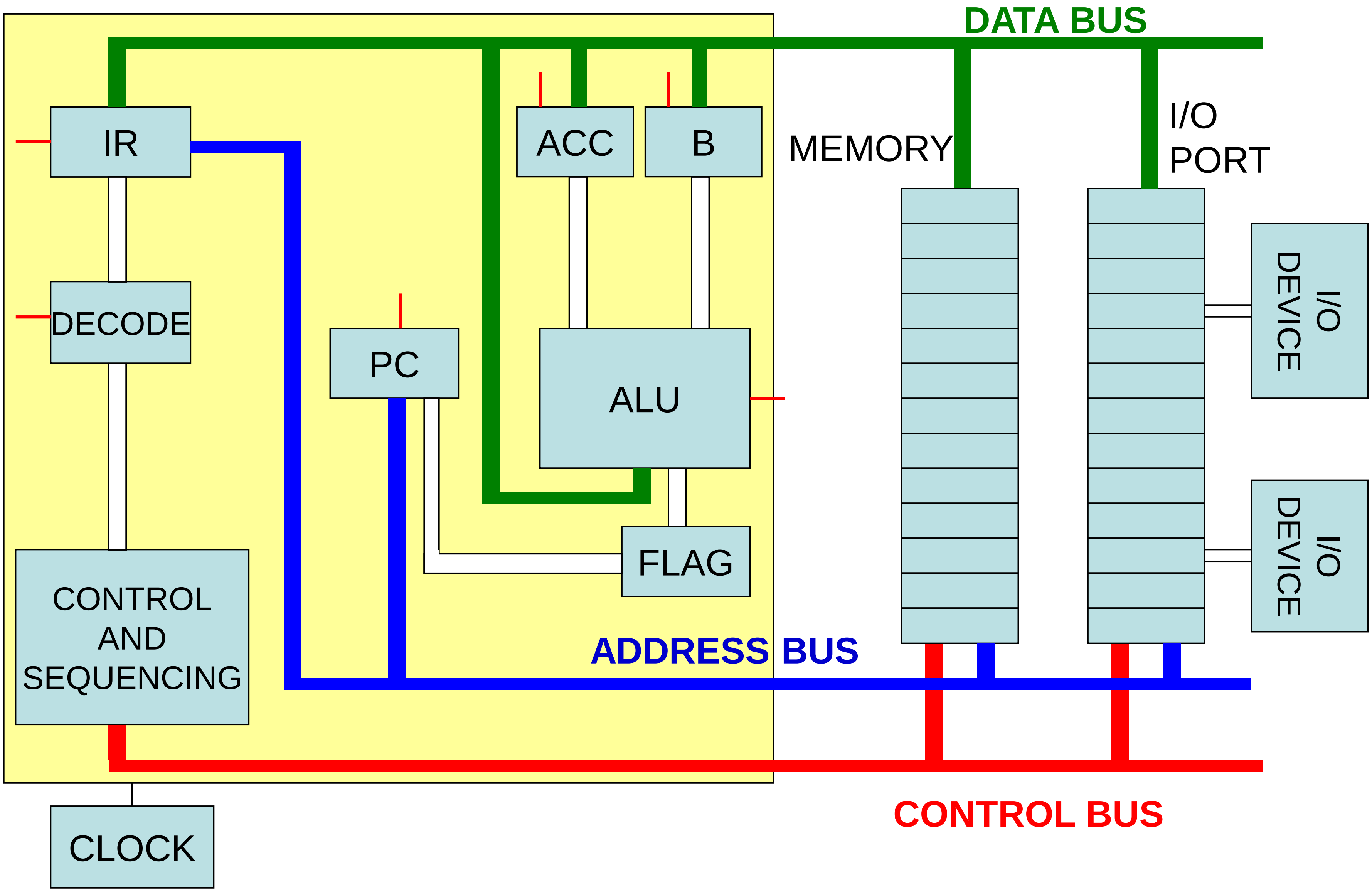

- Basic unit of time, 1GHz→clock cycle=1ns
- A instruction could take multiple cycles to complete, e.g. multiply in 8088 takes 50 cycles

# Instruction execution cycle

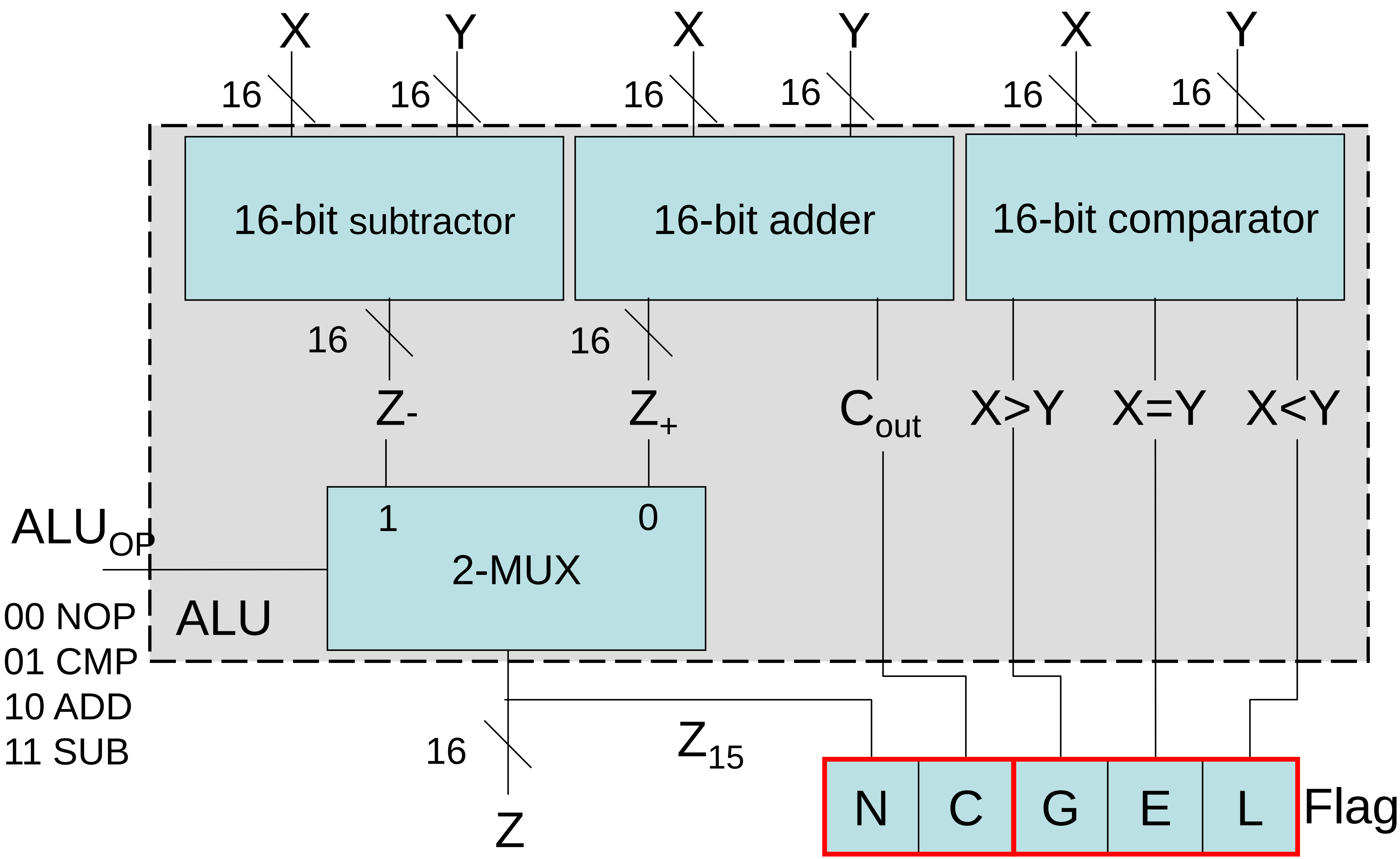



- **Fetch**
- **Decode**
- Fetch operands
- **Execute**
- Store output

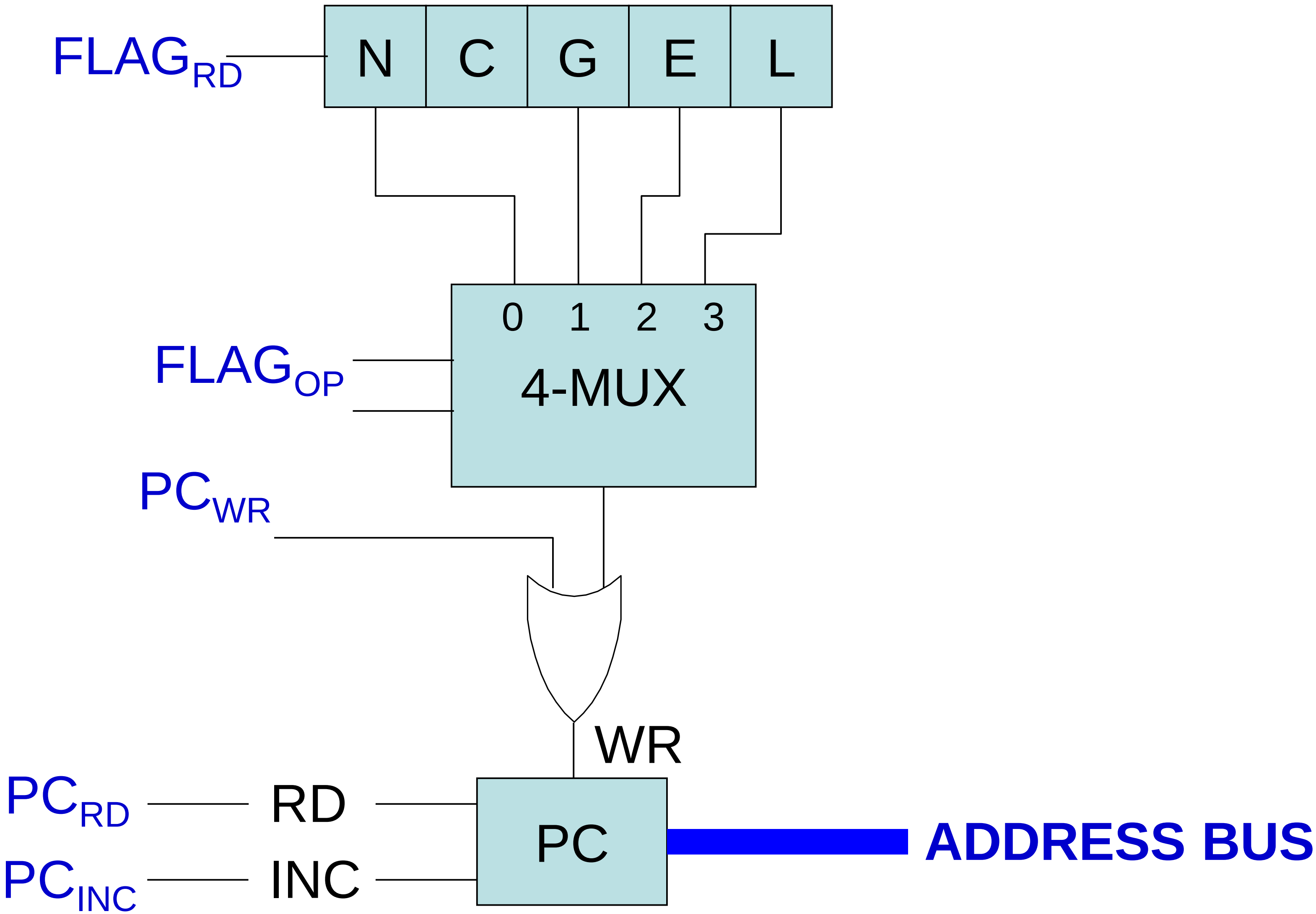
# A simple microcomputer



# ALU and Flag

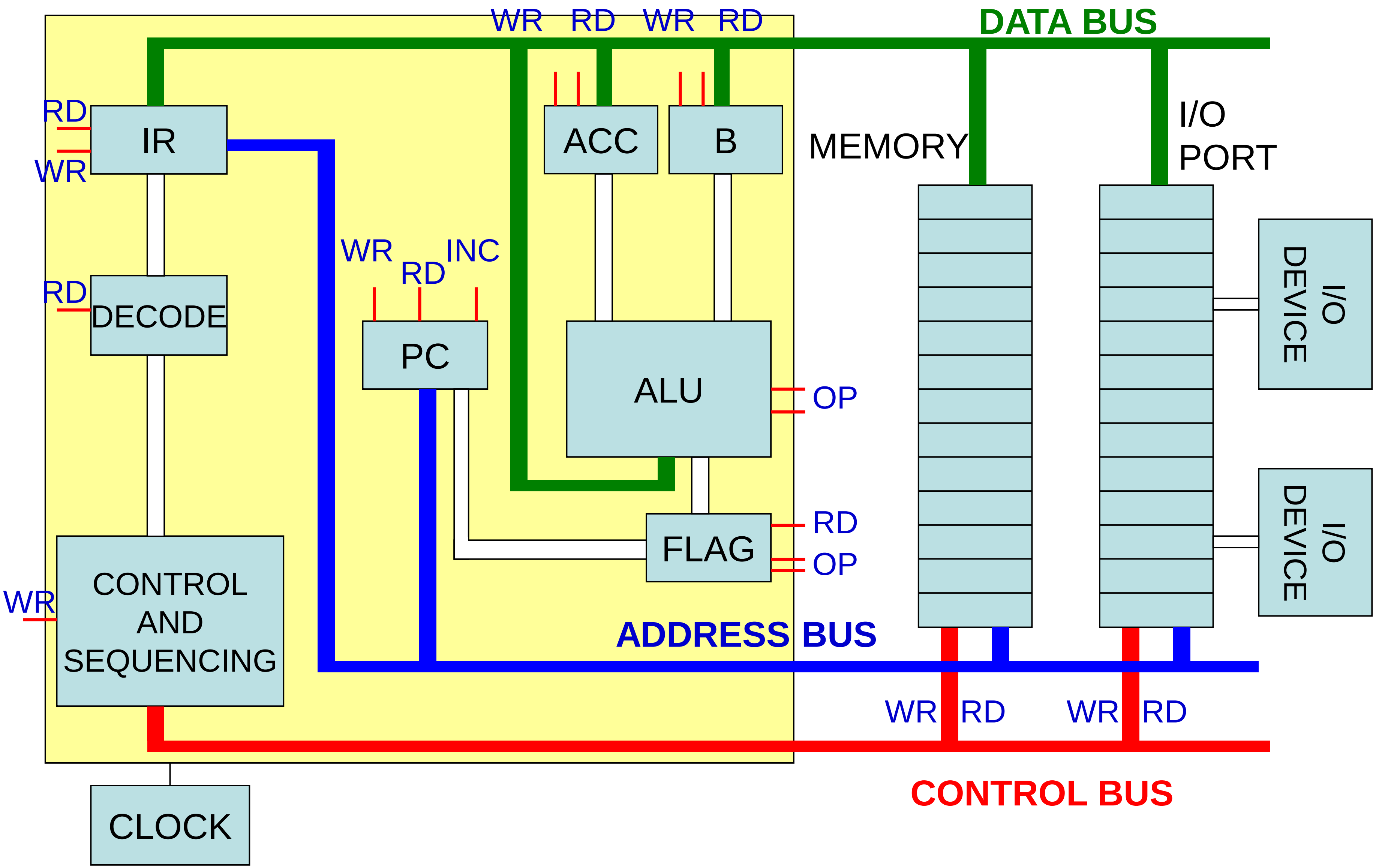

X
Y
X
Y
X
Y
16
16
16
16
16
16
16-bit subtractor
16-bit adder
16-bit comparator
16
16
Z-
Z+
Cout
X>Y
X=Y
X<Y
1
0
2-MUX
ALUOP
00 NOP
01 CMP
10 ADD
11 SUB
ALU
16
Z15
Z
N
C
G
E
L
Flag

# Flags

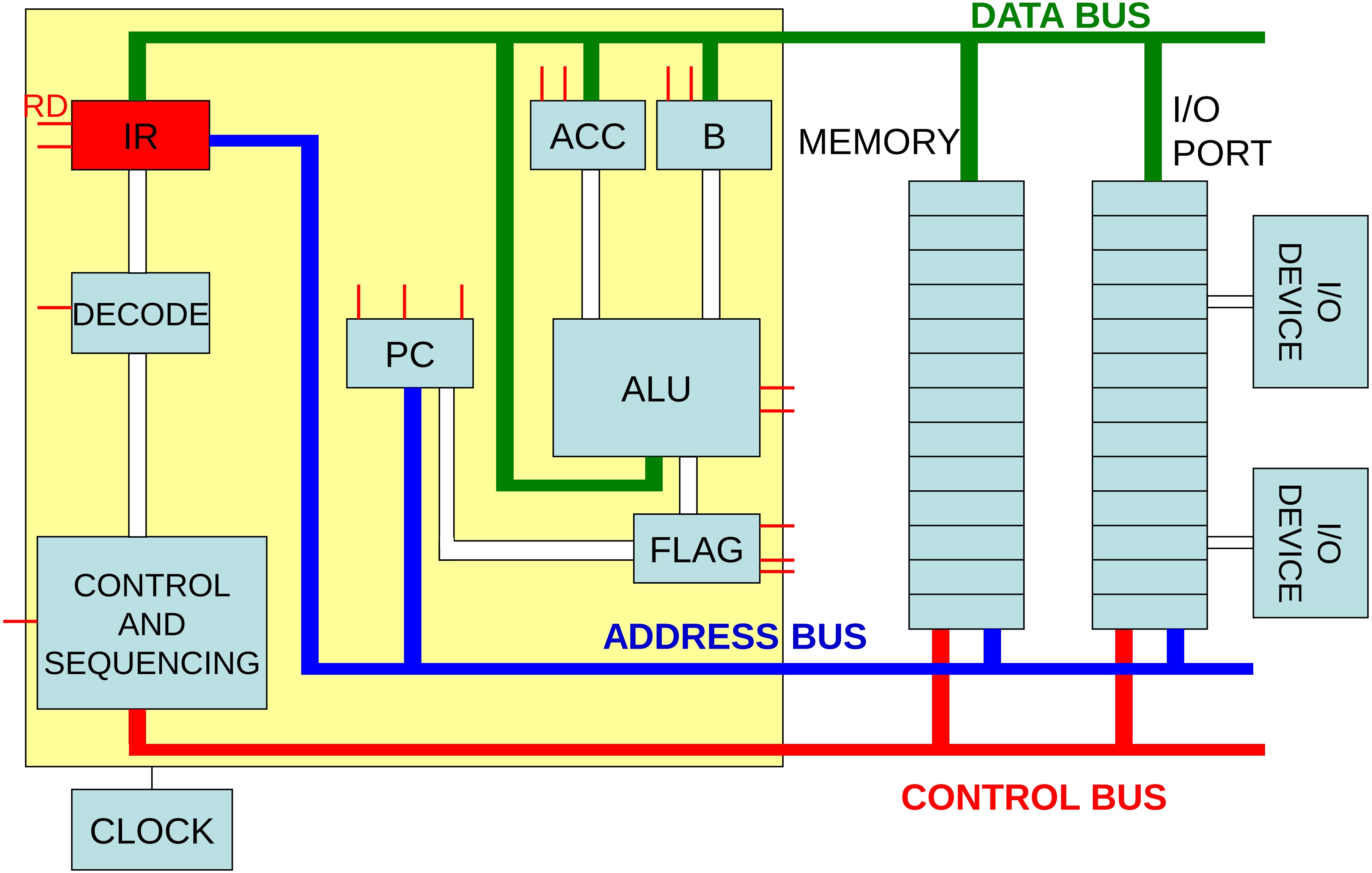


FLAG_RD
N
C
G
E
L
0
1
2
3
4-MUX
FLAG_OP
PC_WR
WR
PC_RD
RD
PC_INC
INC
PC
ADDRESS BUS

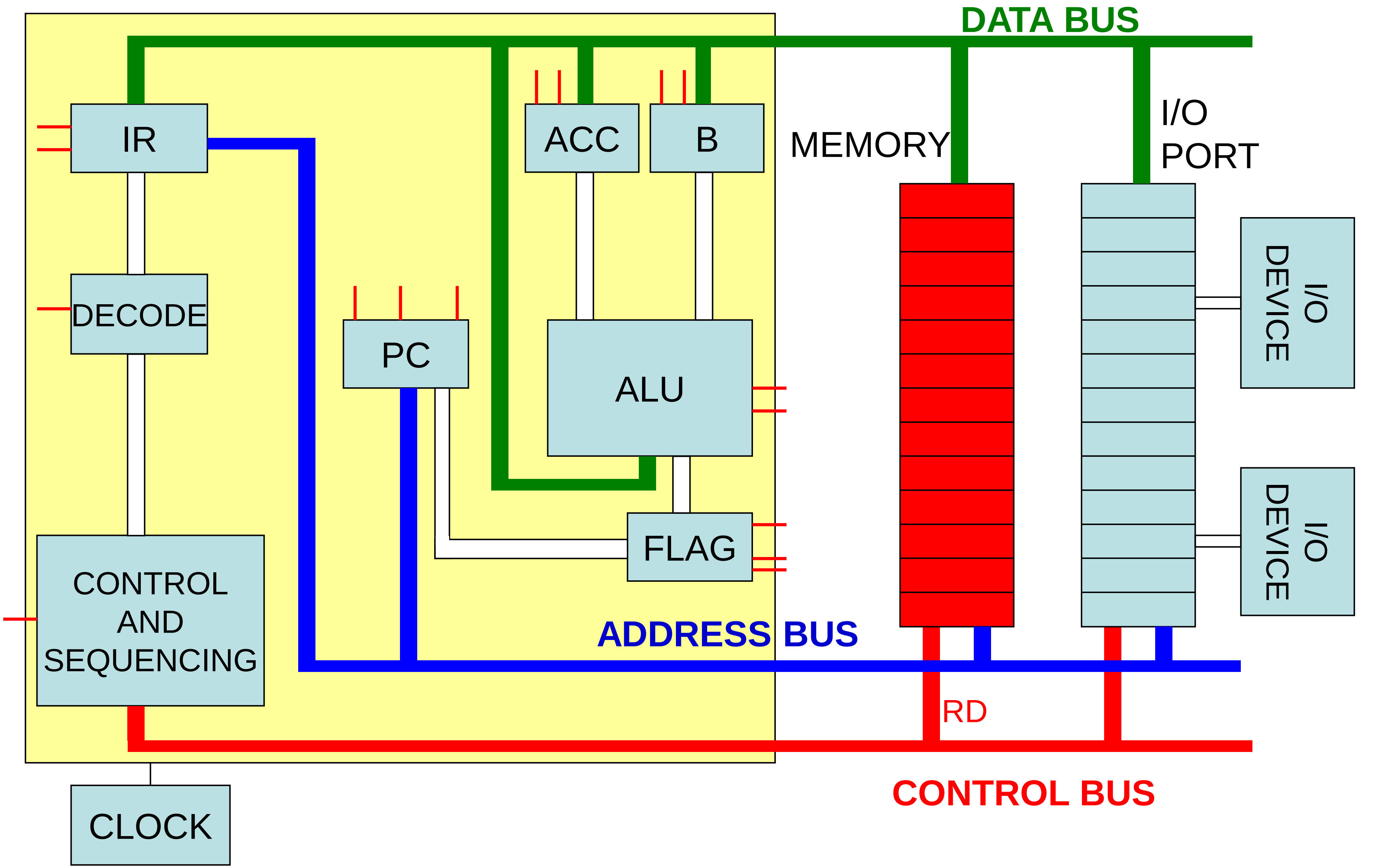
# Control signals (20 in total)



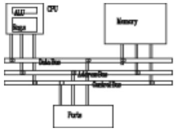
# LDA (execution cycle 1): $IR_{RD}$

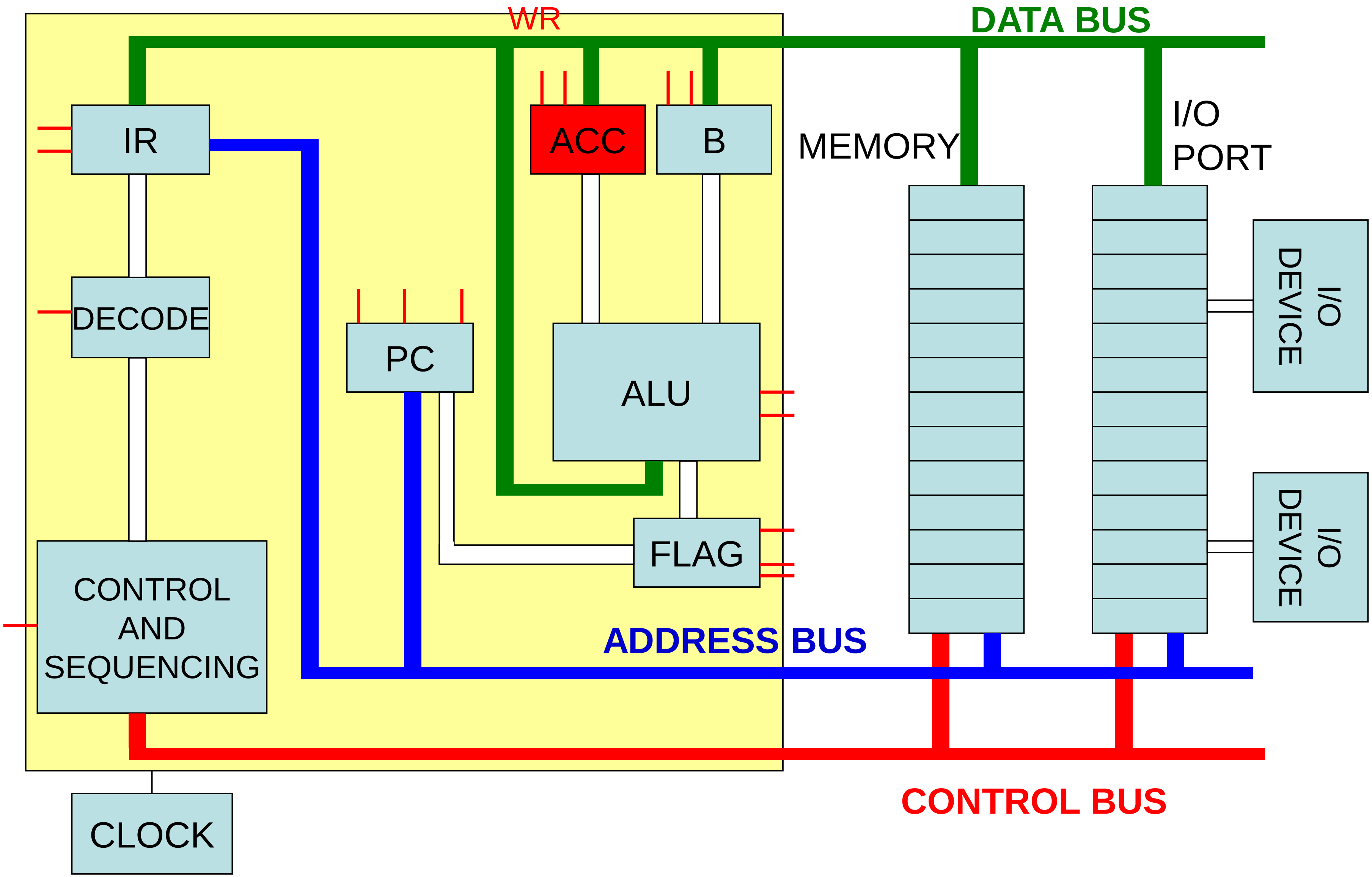


# LDA (execution cycle 2): $MEM_{RD}$

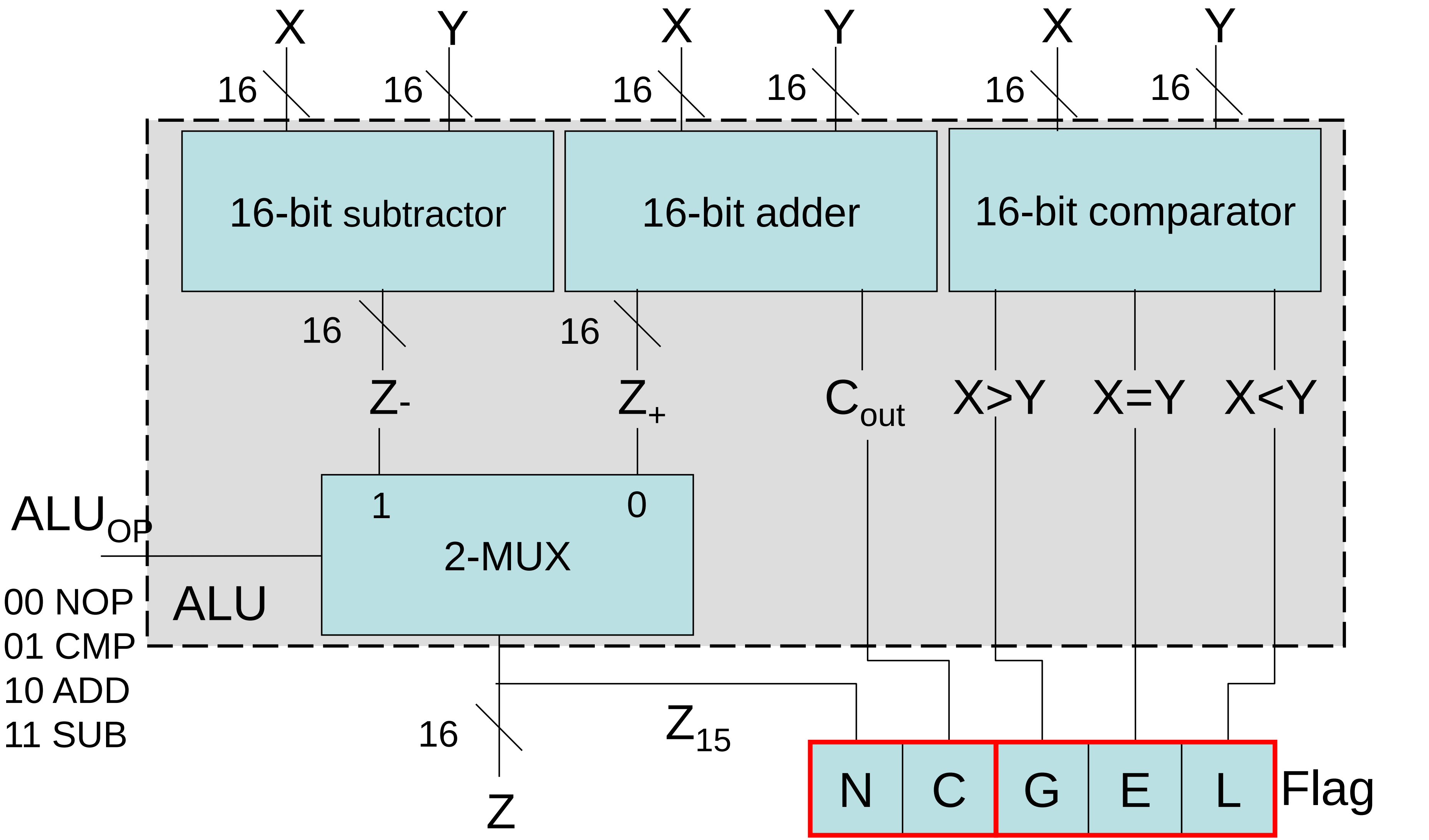


# LDA (execution cycle 3): $ACC_{WR}$

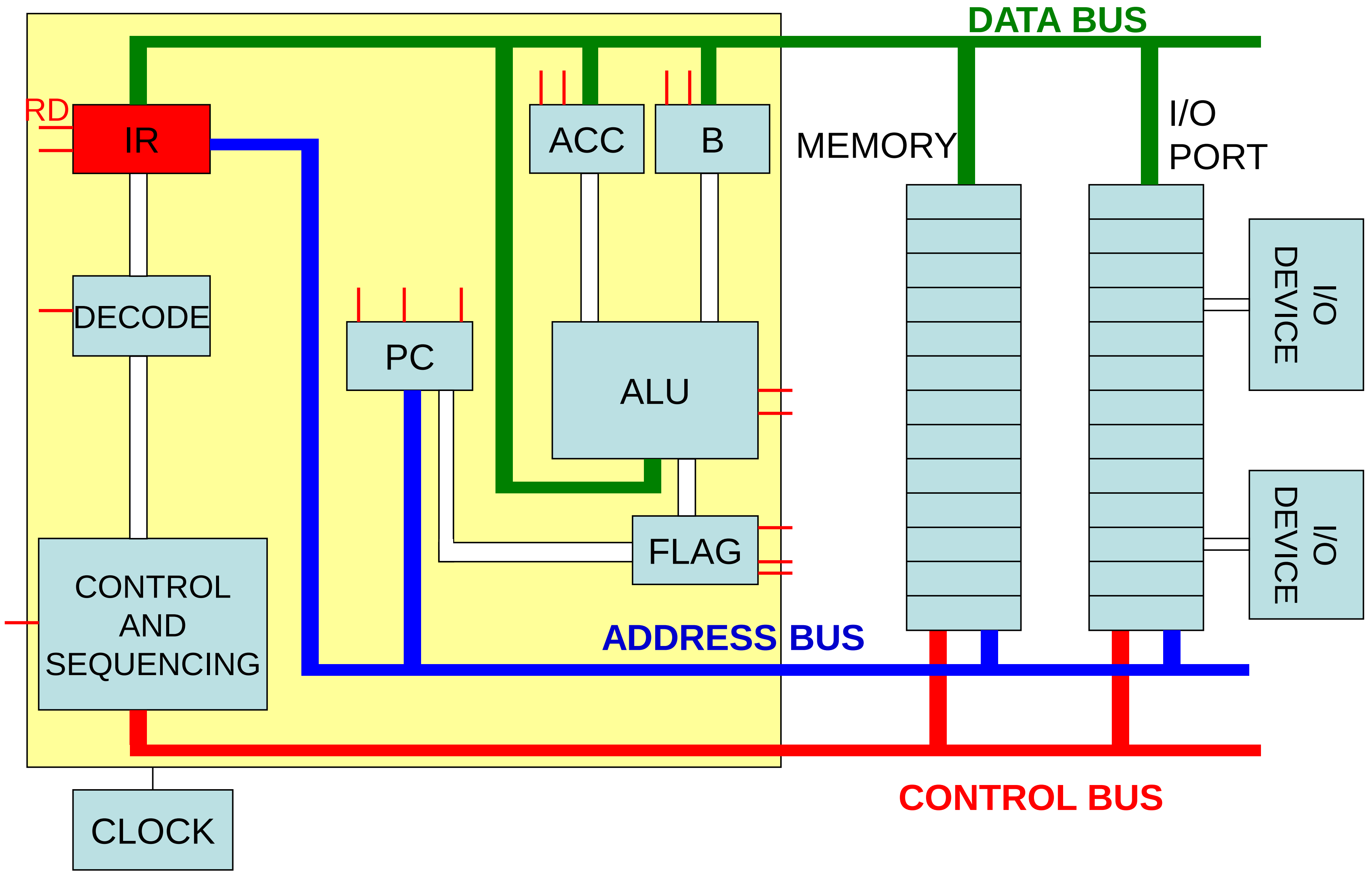




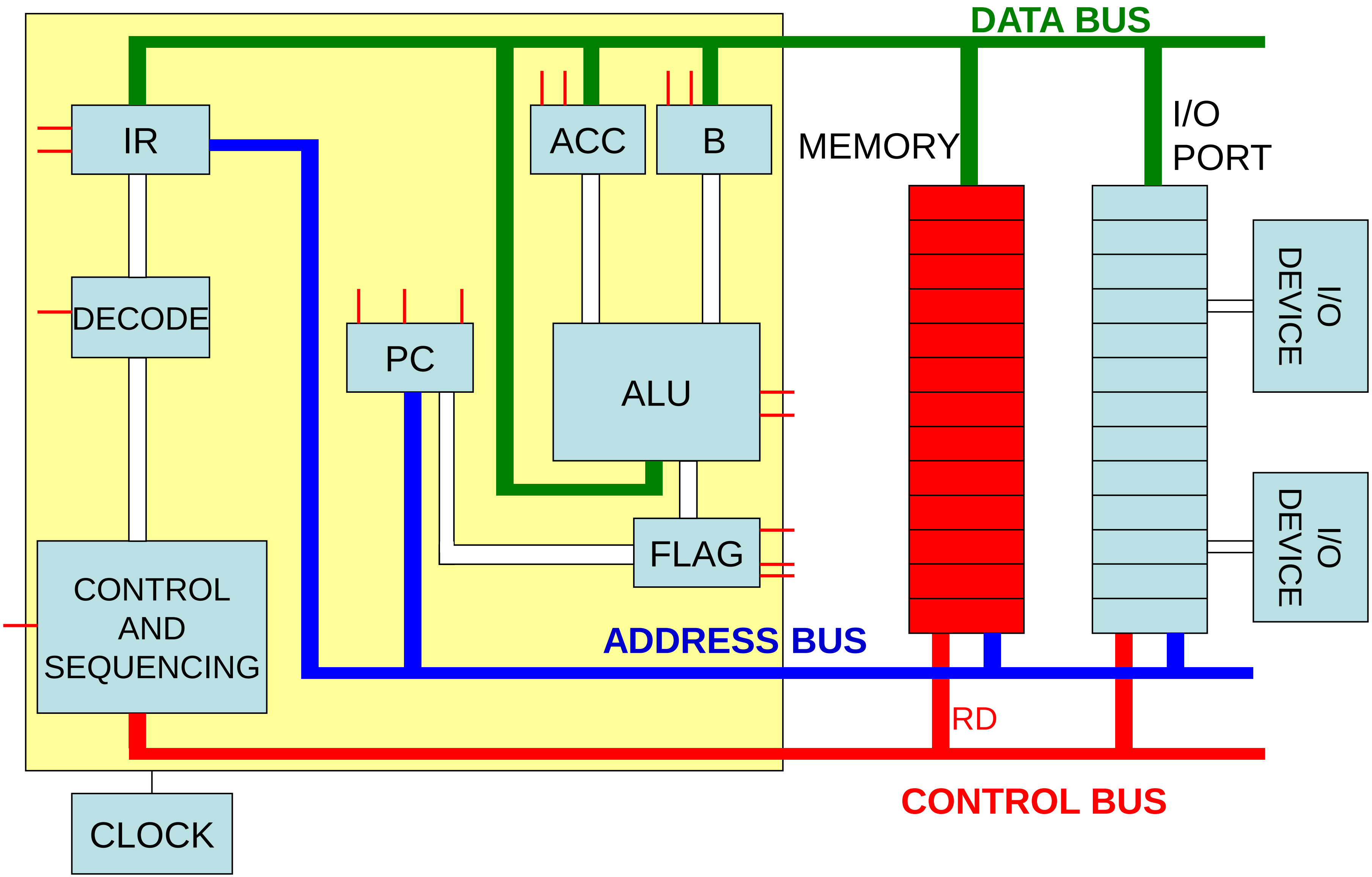
# ALU and Flag


X
Y
X
Y
X
Y
16
16
16
16
16
16
16-bit subtractor
16-bit adder
16-bit comparator
16
16
Z-
Z+
Cout
X>Y
X=Y
X<Y
ALUOP
1
0
2-MUX
ALU
00 NOP
01 CMP
10 ADD
11 SUB
16
Z15
Z
N
C
G
E
L
Flag

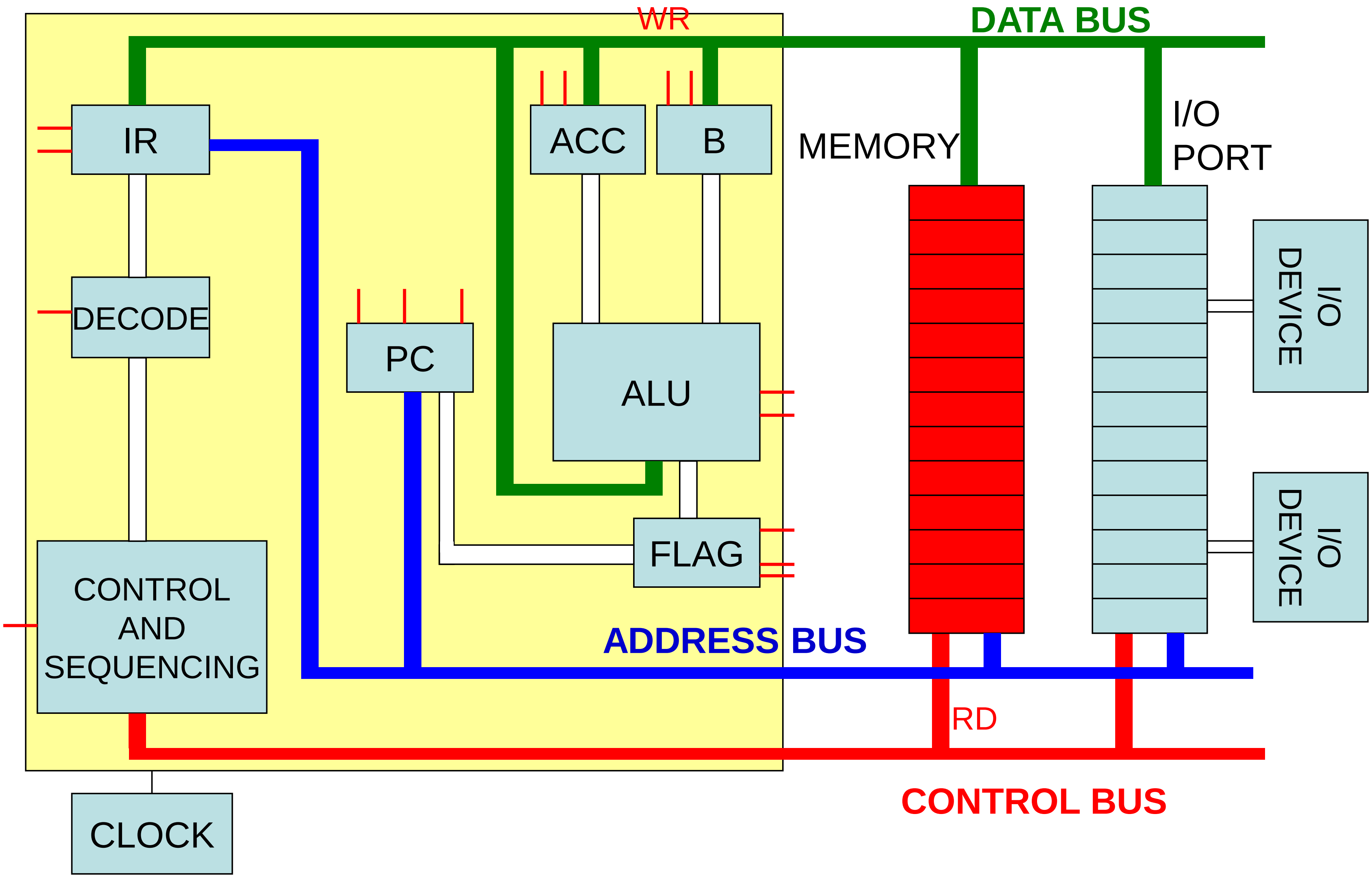
# ADD (execution cycle 1): $IR_{RD}$



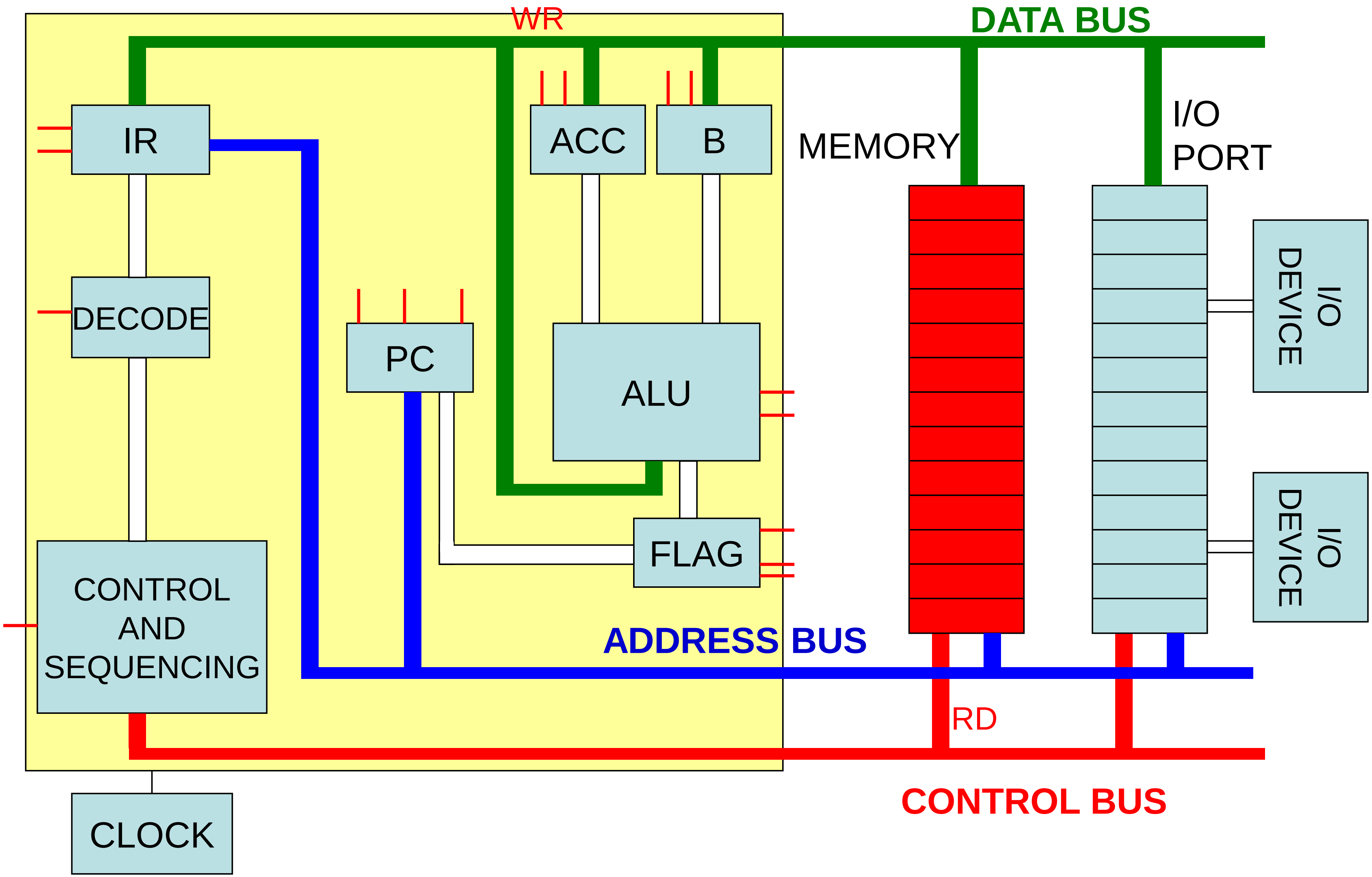
# ADD (execution cycle 2): $MEM_{RD}$



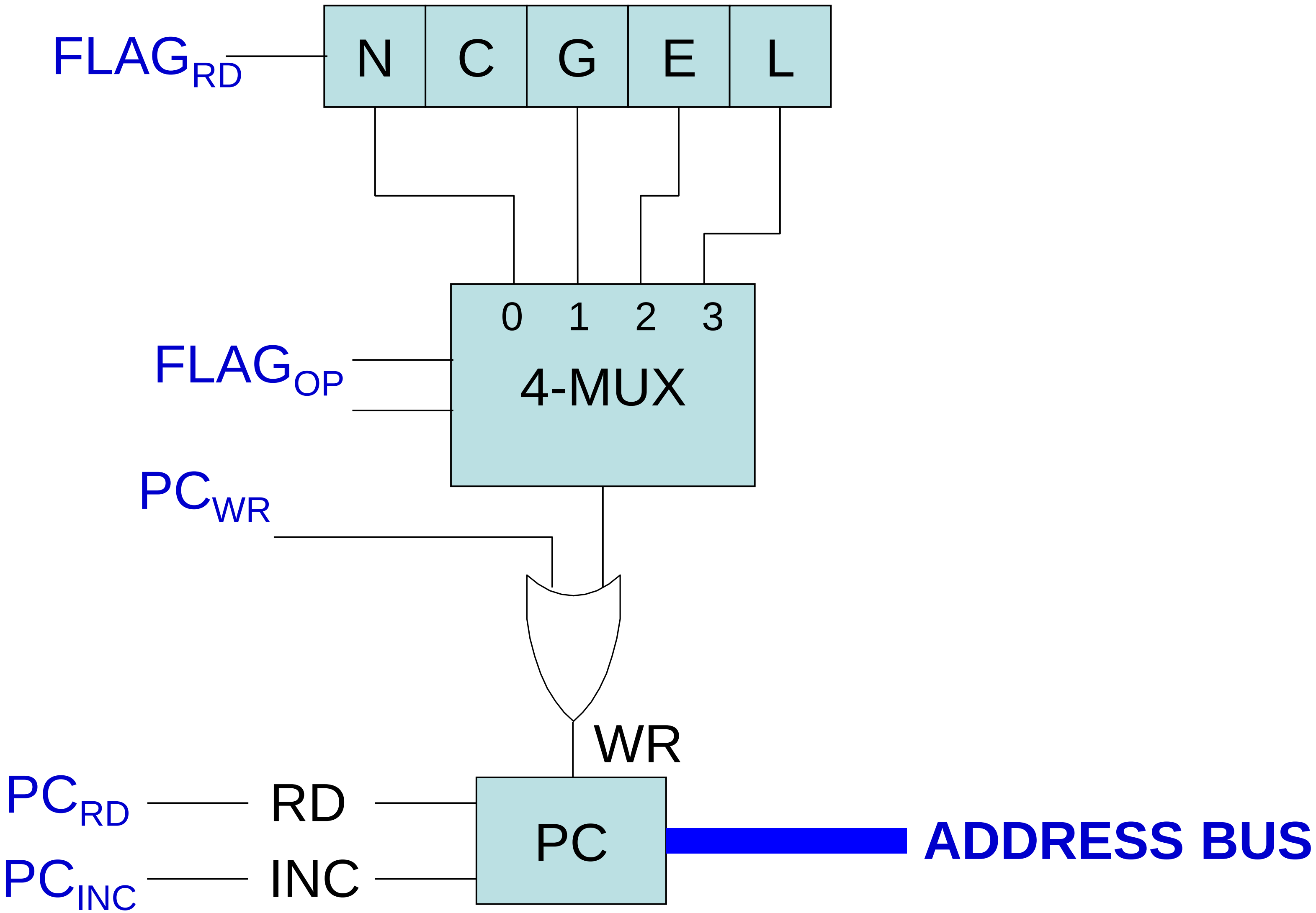
# ADD (execution cycle 3): $B_{WR}$



# ADD (execution cycle 4): $ALU_{10}$,$ACC_{WR}$

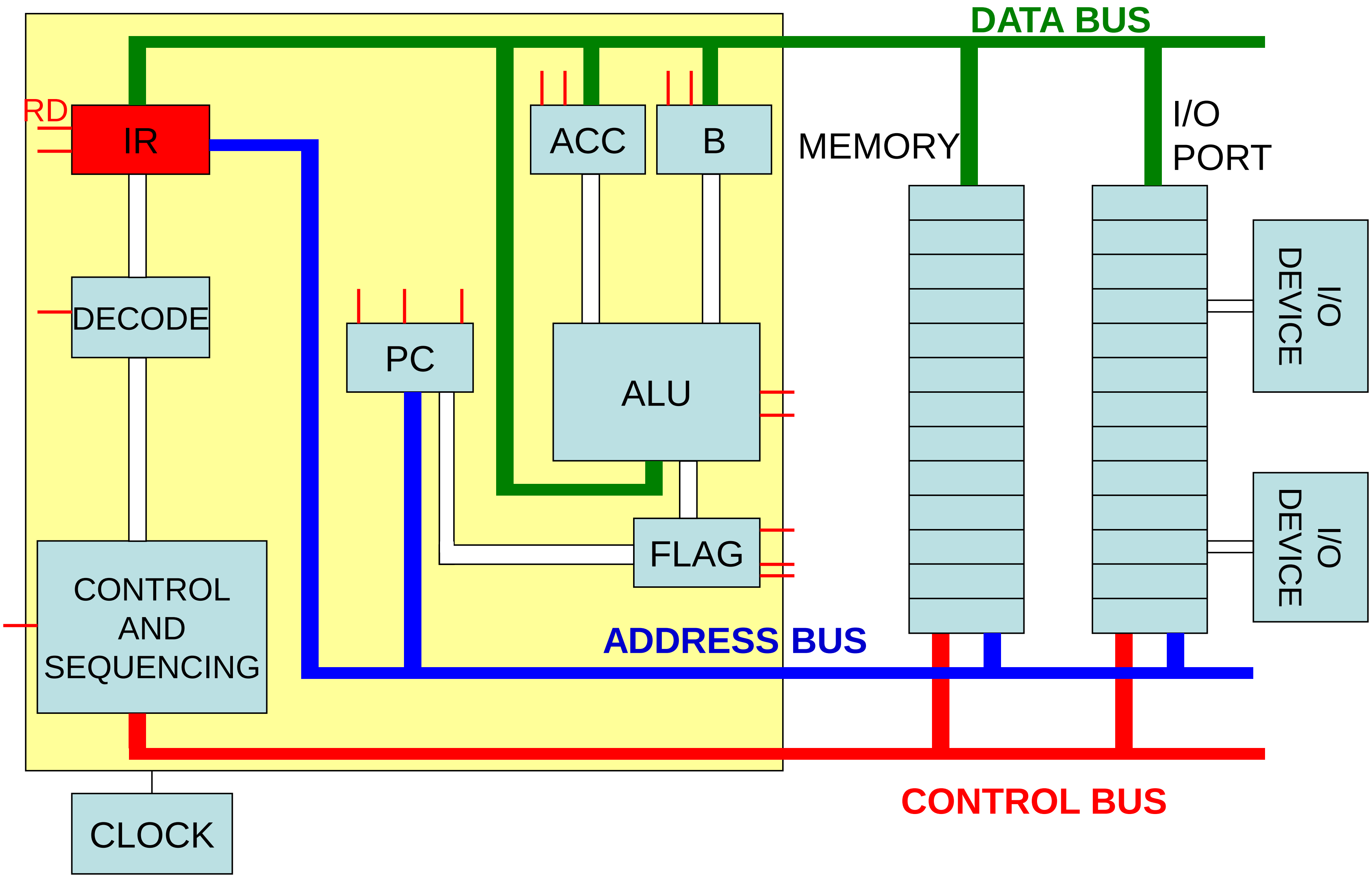


# Flags

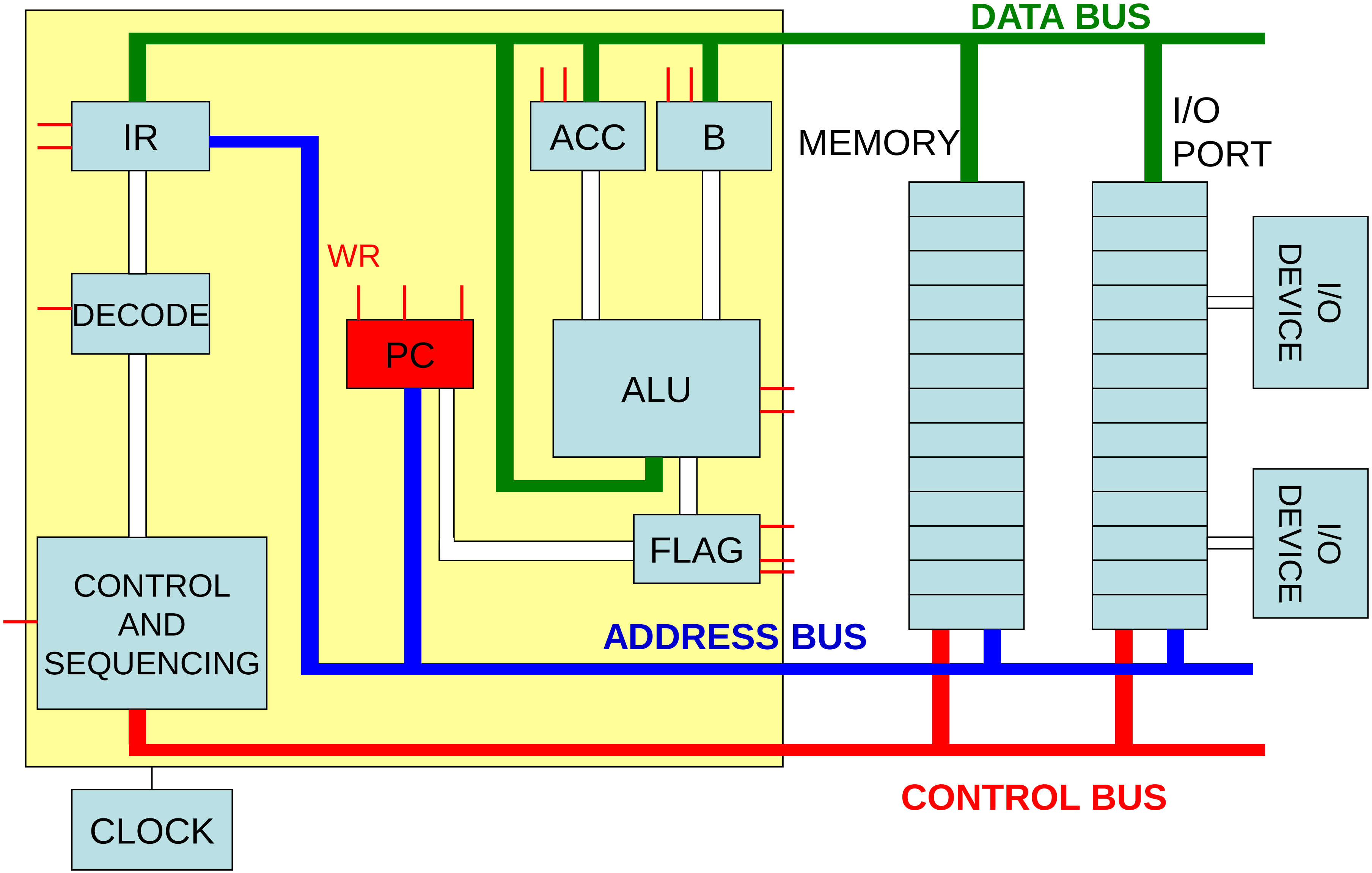


$FLAG_{RD}$
N
C
G
E
L
0
1
2
3
4-MUX
$FLAG_{OP}$
$PC_{WR}$
WR
$PC_{RD}$
RD
$PC_{INC}$
INC
PC
ADDRESS BUS

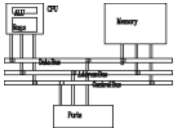
# JMP (execution cycle 1): $IR_{RD}$

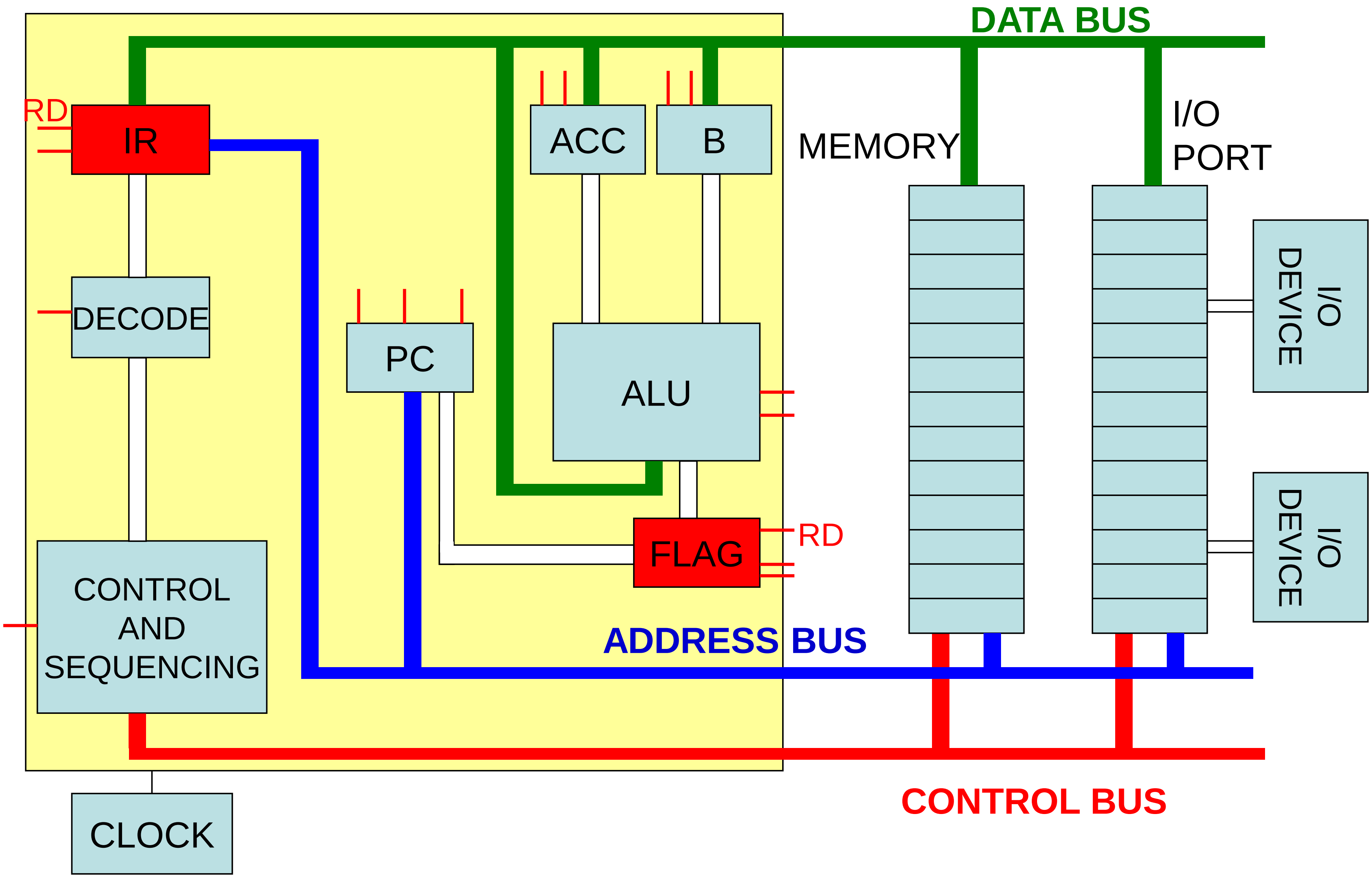


# JMP (execution cycle 2): $PC_{WR}$

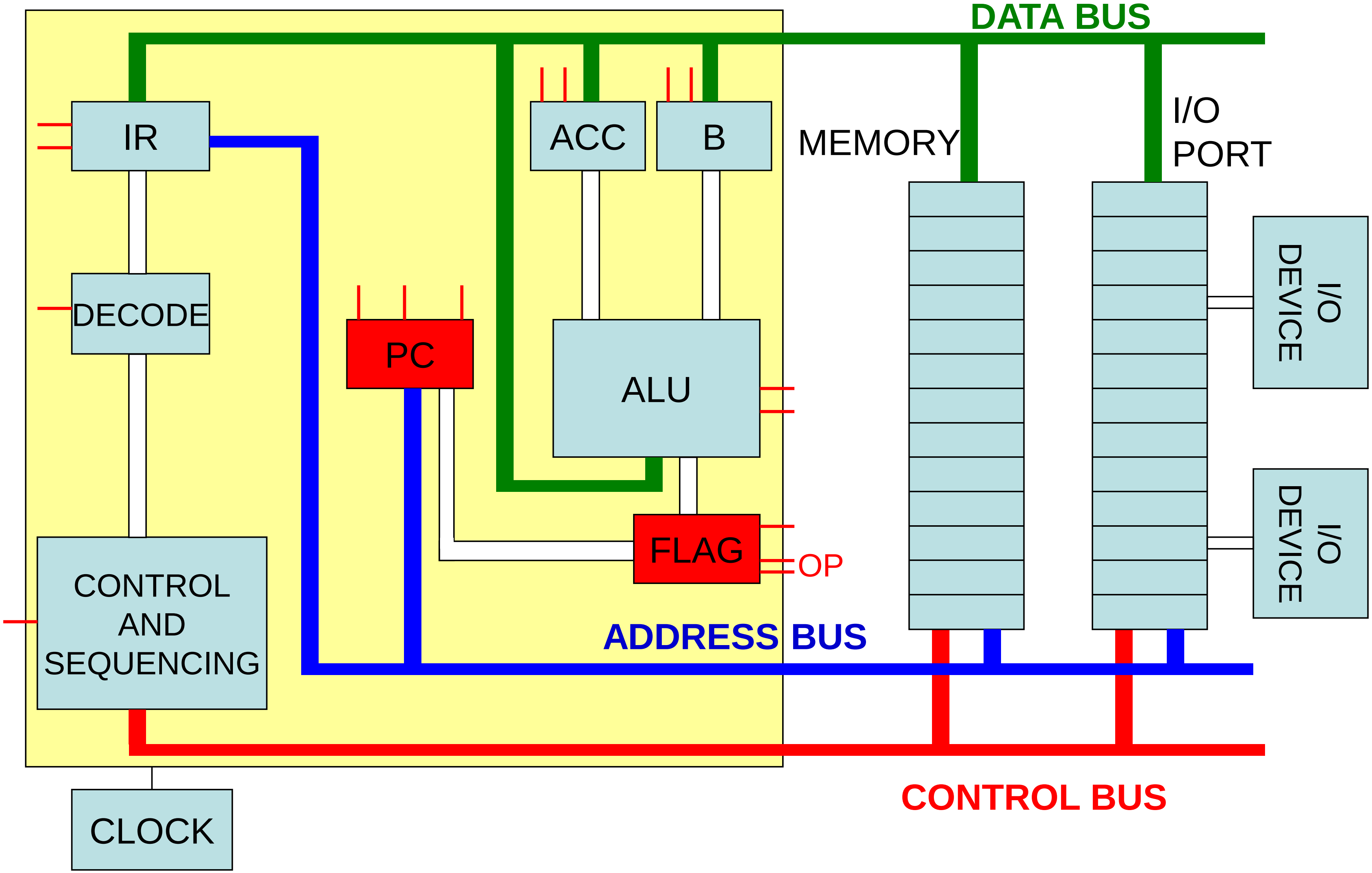


# JG (execution cycle 1): $IR_{RD}$,$FLAG_{RD}$

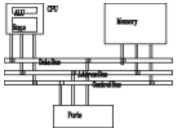




# JG (execution cycle 2): $FLAG_{01}$



# Microcode sequence



**`LDA 510`**

$PC_{RD}$
$MEM_{RD}$
$IR_{WR}$ $PC_{INC}$

$IR_{RD}$
$DECODER_{RD}$
$\mu PC_{WR}$

$IR_{RD}$
$MEM_{RD}$
$ACC_{WR}$

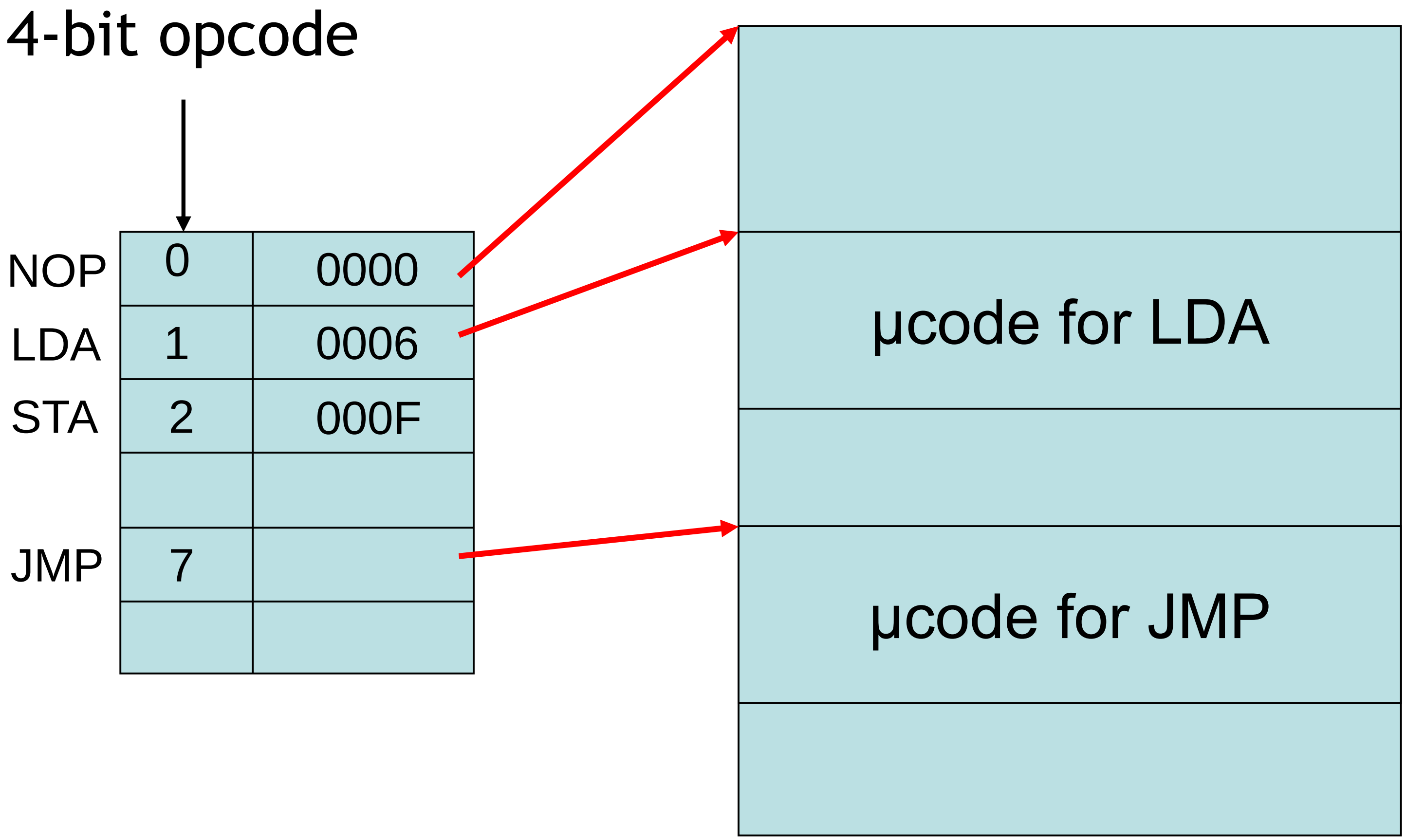
**`JMP 10`**

$PC_{RD}$
$MEM_{RD}$
$IR_{WR}$ $PC_{INC}$

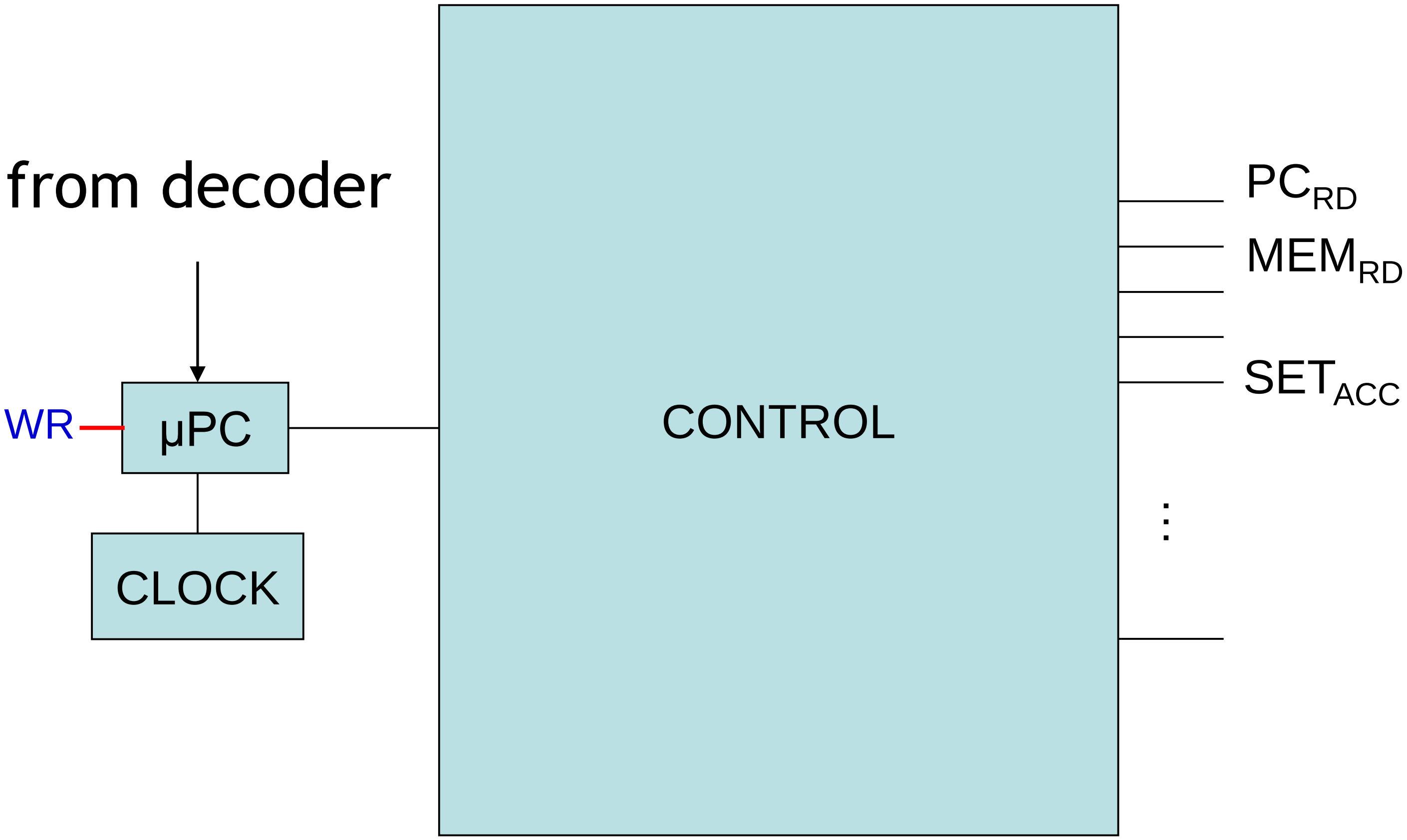
$IR_{RD}$
$DECODER_{RD}$
$\mu PC_{WR}$

$IR_{RD}$
$PC_{WR}$

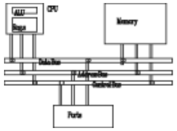
# Decoder


4-bit opcode
NOP
0
0000
LDA
1
0006
STA
2
000F
JMP
7
µcode for LDA
µcode for JMP

# Control and sequencing unit



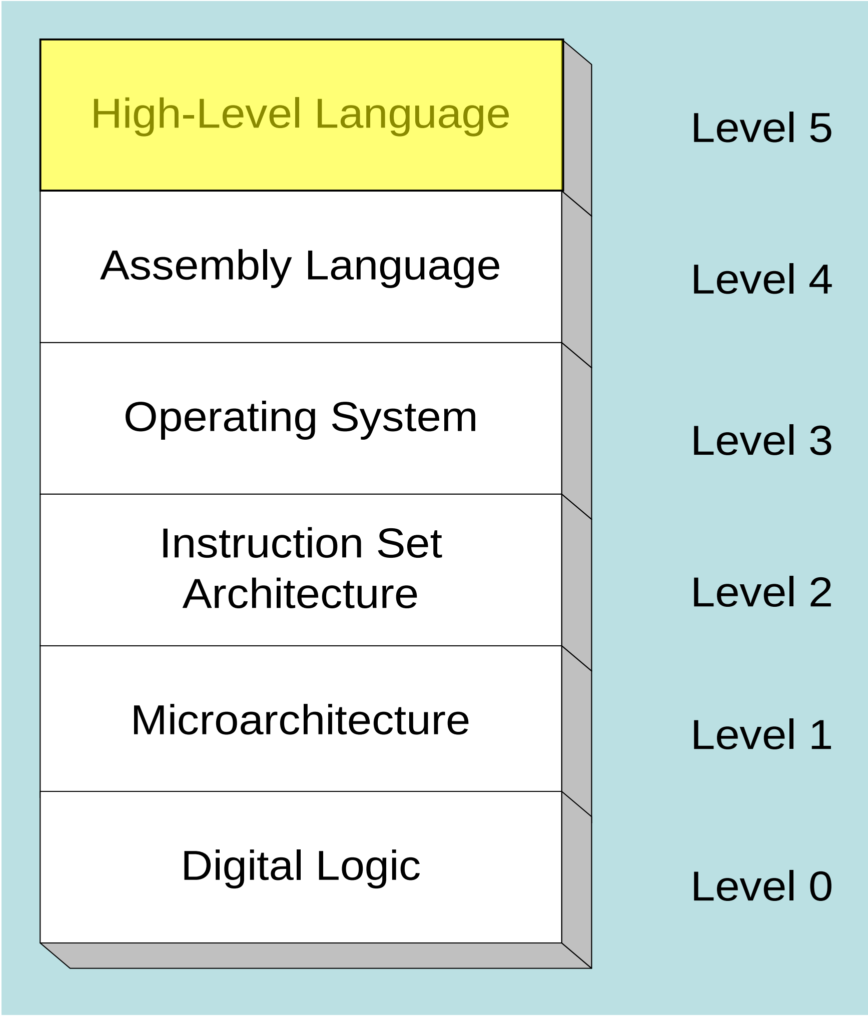
# Control and sequencing unit



| | | $PC_{RD}$ | $MEM_{RD}$ | $MEM_{WR}$ | $IR_{WR}$ | $PC_{INC}$ | …. |
|---|---|---|---|---|---|---|---|
| NOP fetch | 0000 | 1 | 0 | 0 | 0 | 0 | 0…. |
| | 0001 | 0 | 1 | 0 | 0 | 0 | |
| | 0002 | 0 | 0 | 0 | 1 | 1 | |
| decode | 0003 | $IR_{RD}$ | | | | | |
| | 0004 | $DECODER_{RD}$ | | | | | |
| | 0005 | $\mu PC_{WR}$ | | | | | |
| LDA fetch decode | | | | | | | |
| exec | 0006 | $IR_{RD}$ | | | | | |
| | 0007 | $MEM_{RD}$ | | | | | |
| | 0008 | $ACC_{WR}$ | | | | | |
| | 000F | | | | | | |

# Virtual machines

## Abstractions for computers



# X=min of X,Y,Z

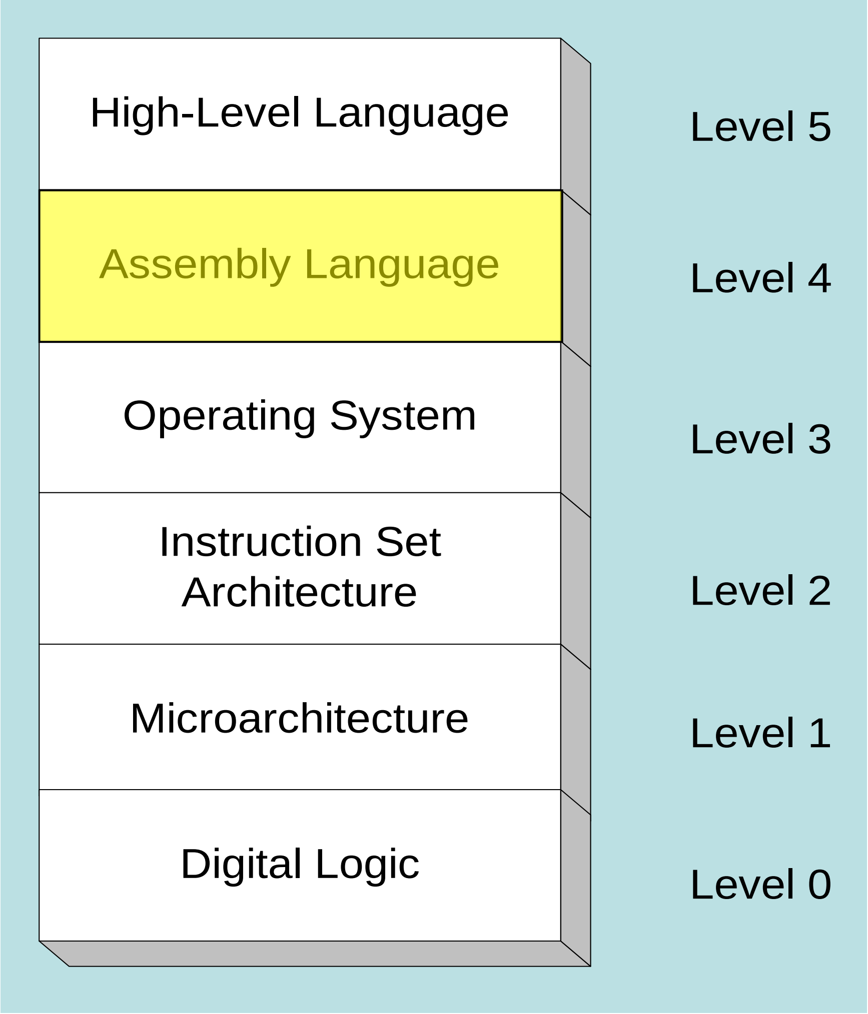
```
int X=7; Y=2; Z=9;
if (X>Y) then
   if (Y>Z) then
      X=Z;
   else
      X=Y;
   end
else
   if (X<Z) then
      X=Z;
   end          else?
end
```

compiler

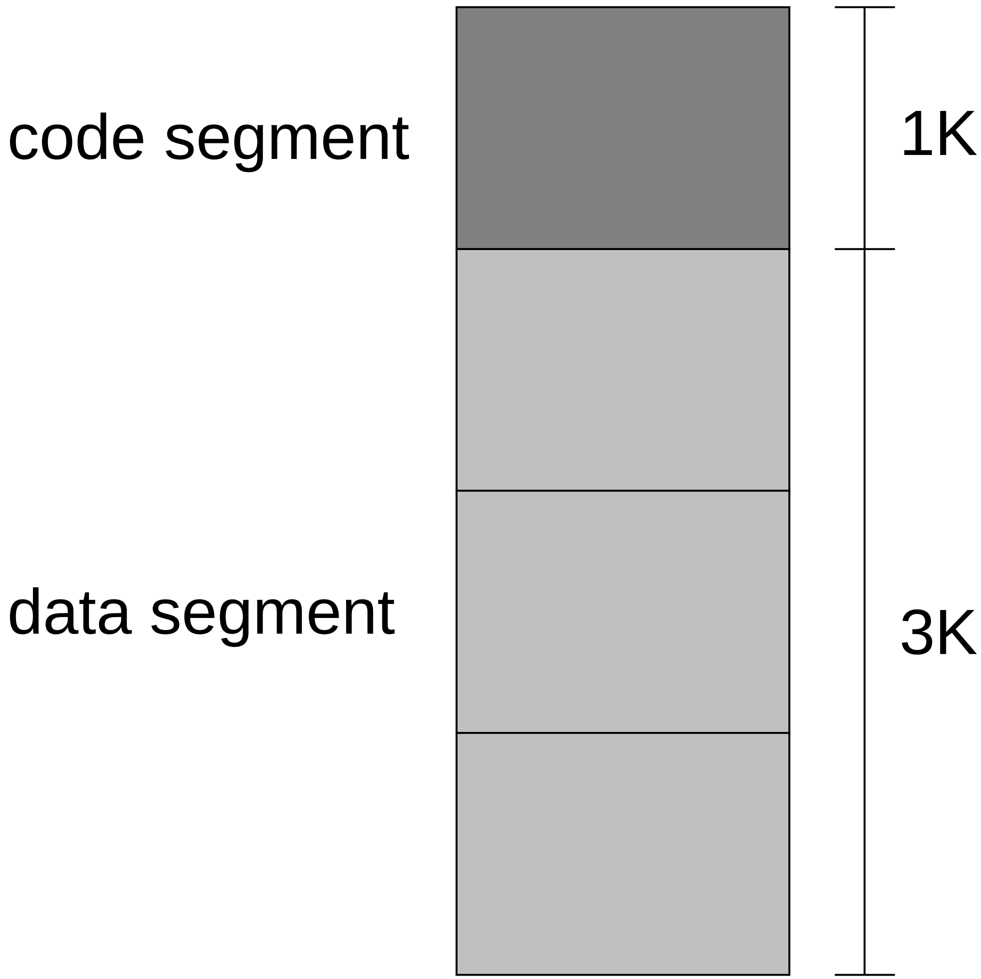
```
.DATA
X       007
Y       002
Z       009
.CODE
        LDA     X
        CMP     Y
        JG      L1
        CMP     Z
        JL      L0
        JMP     END
L0      LDA     Z
        STA     X
L1      LDA     Y
        CMP     Z
        JG      L2
        STA     X
        JMP     END
L2      LDA     Z
        STA     X
END     HLT
```

# Virtual machines

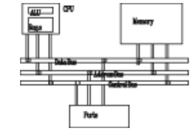
## Abstractions for computers



# Memory layout



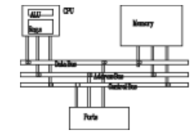
# X=min of X,Y,Z



```
.DATA
X       007
Y       002
Z       009
.CODE
        LDA    X
        CMP    Y
        JL     L1
        LDA    Y
L1      CMP    Z
        JL     L2
        LDA    Z
L2      STA    X
        HLT
```

```
.DATA
X       007
Y       002
Z       009
.CODE
        LDA    Y
        CMP    Z
        JL     L1
        LDA    Z
L1      CMP    X
        JG     END
        STA    X
END     HLT
```

# X=min of X,Y,Z

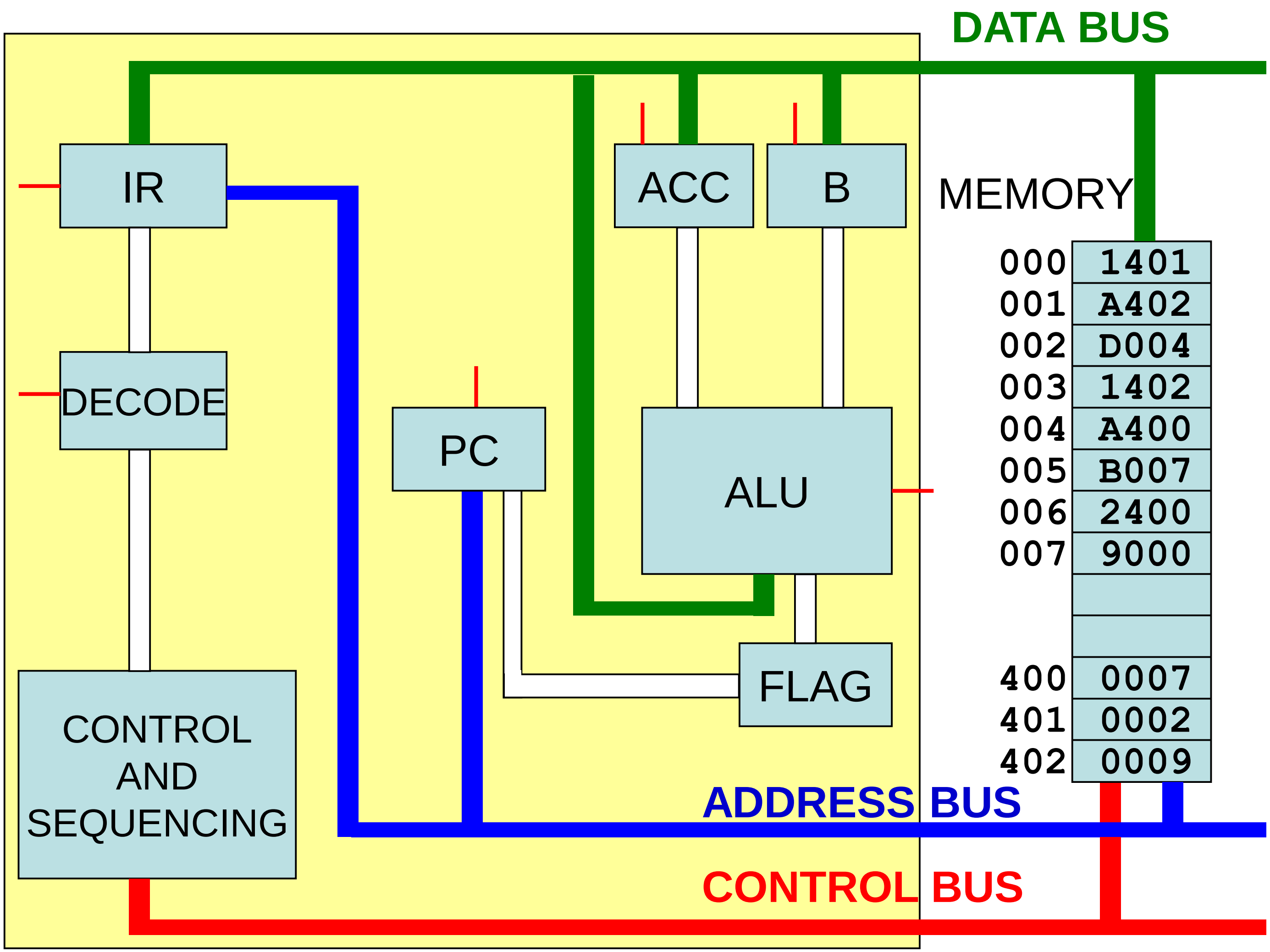


| | | | |
|---|---|---|---|
| .DATA | | | |
| X | 007 | | |
| Y | 002 | | |
| Z | 009 | | |
| .CODE | | | |

| # | Label | Op | Operand | Machine code |
|---|---|---|---|---|
| 0 | | LDA | Y | 1401 |
| 1 | | CMP | Z | A402 |
| 2 | | JL | L1 | D004 |
| 3 | | LDA | Z | 1402 |
| 4 | L1 | CMP | X | A400 |
| 5 | | JG | END | B007 |
| 6 | | STA | X | 2400 |
| 7 | END | HLT | | 9000 |

| | |
|---|---|
| X | 400 |
| Y | 401 |
| Z | 402 |
| | |

| | |
|---|---|
| L1 | 4 |
| END | 7 |
| | |
| | |




| | |
|---|---|
| **LDA** | **401** |
| **CMP** | **402** |
| **JL** | **004** |
| **LDA** | **402** |
| **CMP** | **400** |
| **JG** | **007** |
| **STA** | **400** |
| **HLT** | |

# Advanced architecture

# Multi-stage pipeline

- Pipelining makes it possible for processor to execute instructions in parallel
- Instruction execution divided into discrete stages

Example of a non-pipelined processor. For example, 80386. Many wasted cycles.

Stages

| Cycles | S1 | S2 | S3 | S4 | S5 | S6 |
|---|---|---|---|---|---|---|
| 1 | I-1 | | | | | |
| 2 | | I-1 | | | | |
| 3 | | | I-1 | | | |
| 4 | | | | I-1 | | |
| 5 | | | | | I-1 | |
| 6 | | | | | | I-1 |
| 7 | I-2 | | | | | |
| 8 | | I-2 | | | | |
| 9 | | | I-2 | | | |
| 10 | | | | I-2 | | |
| 11 | | | | | I-2 | |
| 12 | | | | | | I-2 |

# Pipelined execution

- More efficient use of cycles, greater throughput of instructions: (80486 started to use pipelining)

Stages

| Cycles | S1 | S2 | S3 | S4 | S5 | S6 |
|---|---|---|---|---|---|---|
| 1 | I-1 | | | | | |
| 2 | I-2 | I-1 | | | | |
| 3 | | I-2 | I-1 | | | |
| 4 | | | I-2 | I-1 | | |
| 5 | | | | I-2 | I-1 | |
| 6 | | | | | I-2 | I-1 |
| 7 | | | | | | I-2 |

For *k* stages and *n* instructions, the number of required cycles is:

$$k + (n - 1)$$

compared to k*n

# Wasted cycles (pipelined)

- When one of the stages requires two or more clock cycles, clock cycles are again wasted.

Stages

| Cycles | S1 | S2 | S3 | S4 (exe) | S5 | S6 |
|---|---|---|---|---|---|---|
| 1 | I-1 | | | | | |
| 2 | I-2 | I-1 | | | | |
| 3 | I-3 | I-2 | I-1 | | | |
| 4 | | I-3 | I-2 | I-1 | | |
| 5 | | | I-3 | I-1 | | |
| 6 | | | | I-2 | I-1 | |
| 7 | | | | I-2 | | I-1 |
| 8 | | | | I-3 | I-2 | |
| 9 | | | | I-3 | | I-2 |
| 10 | | | | | I-3 | |
| 11 | | | | | | I-3 |

For $k$ stages and $n$ instructions, the number of required cycles is:

$$k + (2n - 1)$$

# Superscalar

A superscalar processor has multiple execution pipelines. In the following, note that Stage S4 has left and right pipelines (u and v).

Stages

| Cycles | | | | S4 | | | |
|---|---|---|---|---|---|---|---|
| | S1 | S2 | S3 | u | v | S5 | S6 |
| 1 | I-1 | | | | | | |
| 2 | I-2 | I-1 | | | | | |
| 3 | I-3 | I-2 | I-1 | | | | |
| 4 | I-4 | I-3 | I-2 | I-1 | | | |
| 5 | | I-4 | I-3 | I-1 | I-2 | | |
| 6 | | | I-4 | I-3 | I-2 | I-1 | |
| 7 | | | | I-3 | I-4 | I-2 | I-1 |
| 8 | | | | | I-4 | I-3 | I-2 |
| 9 | | | | | | I-4 | I-3 |
| 10 | | | | | | | I-4 |

For *k* states and *n* instructions, the number of required cycles is:

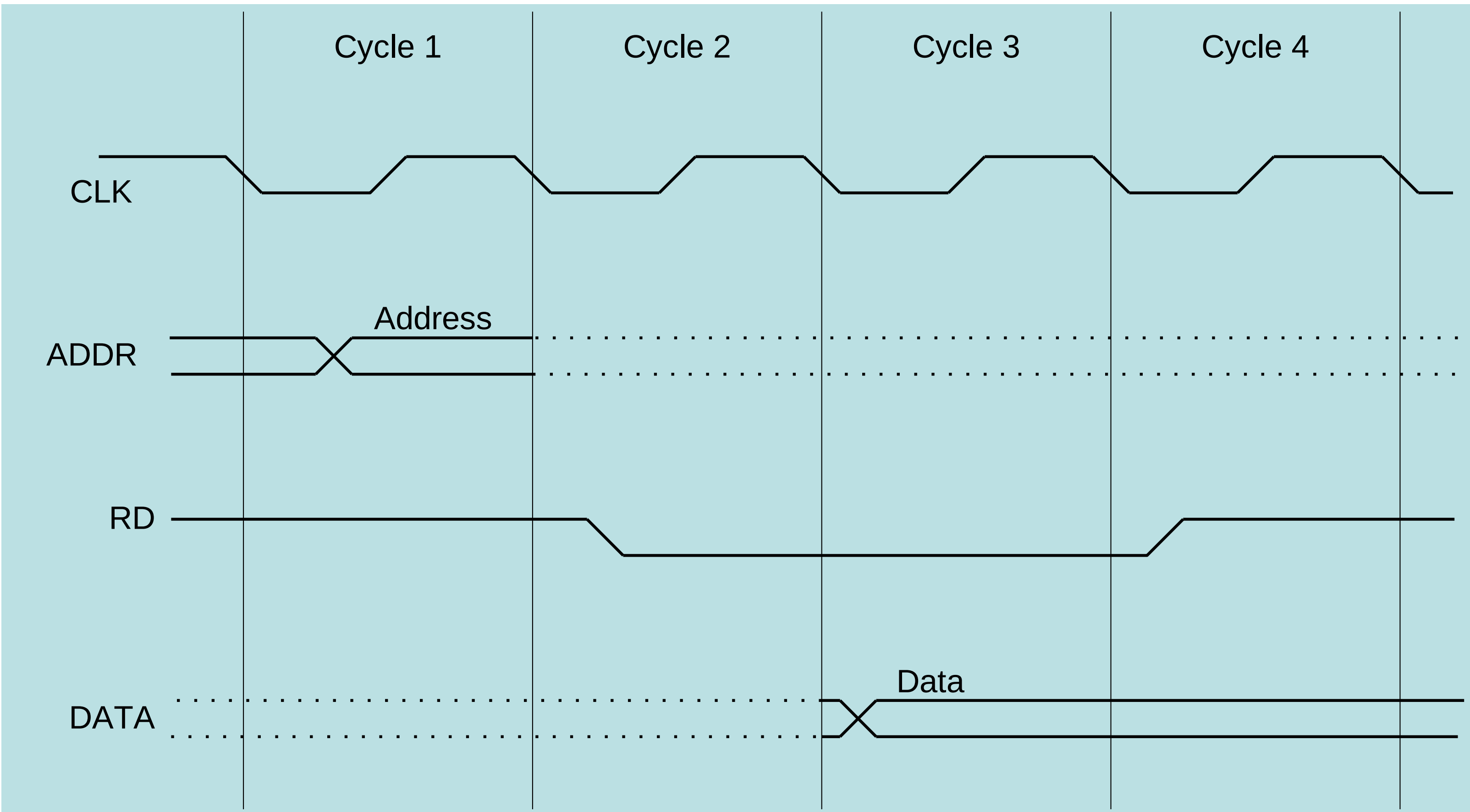
$$k + n$$

Pentium: 2 pipelines

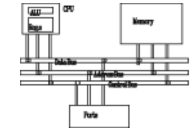
Pentium Pro: 3

# Reading from memory

- Multiple machine cycles are required when reading from memory, because it responds much more slowly than the CPU. The four steps are:

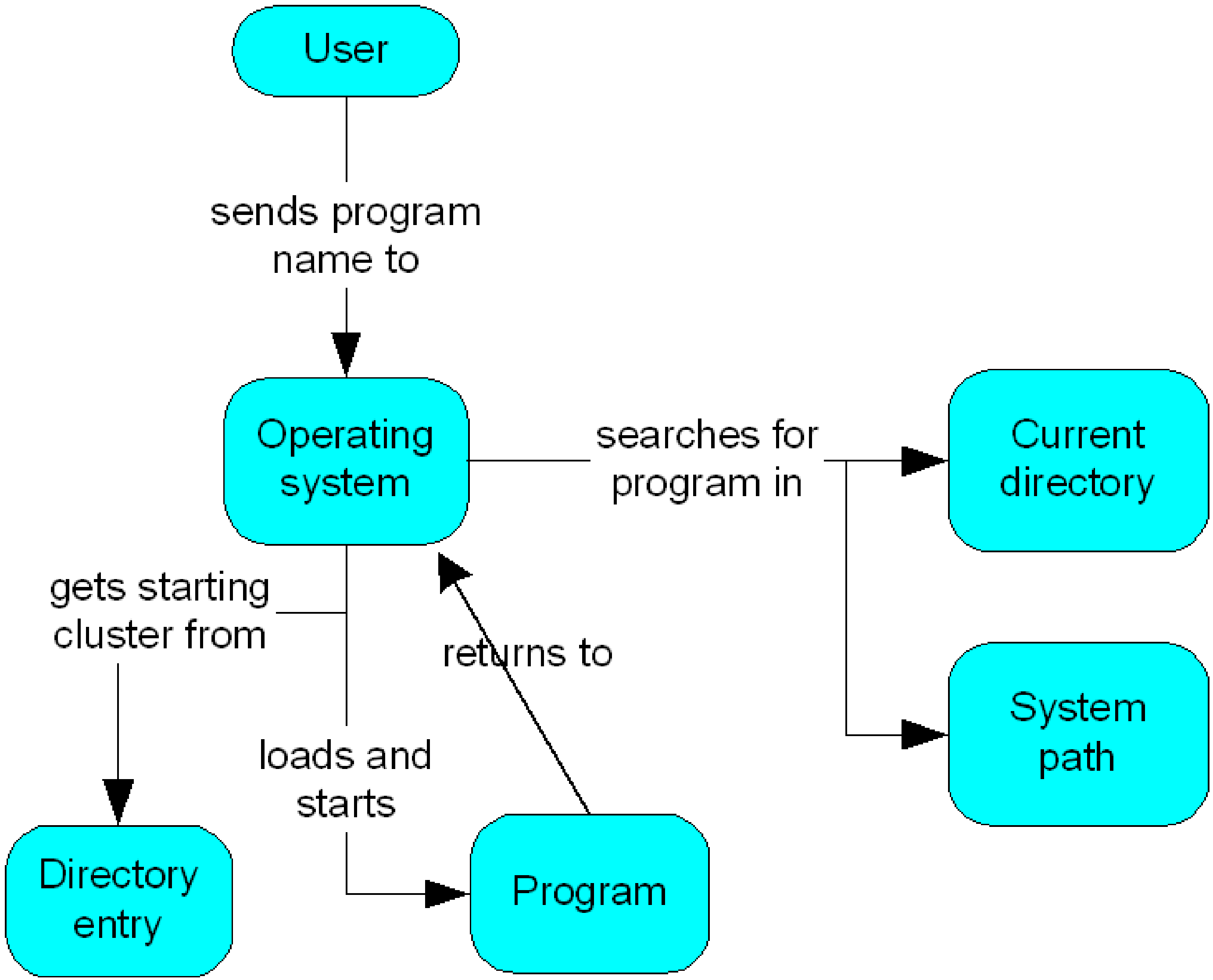


# Cache memory

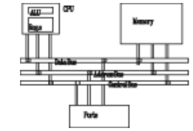


- High-speed expensive static RAM both inside and outside the CPU.
  - Level-1 cache: inside the CPU
  - Level-2 cache: outside the CPU
- Cache hit: when data to be read is already in cache memory
- Cache miss: when data to be read is not in cache memory. When? compulsory, capacity and conflict.
- Cache design: cache size, n-way, block size, replacement policy

# How a program runs



# Multitasking



- OS can run multiple programs at the same time.
- Multiple threads of execution within the same program.
- Scheduler utility assigns a given amount of CPU time to each running program.
- Rapid switching of tasks
  - gives illusion that all programs are running at once
  - the processor must support task switching
  - scheduling policy, round-robin, priority

# IA-32 Architecture

# IA-32 architecture

- From 386 to the latest 32-bit processor, P4
- From programmer's point of view, IA-32 has not changed substantially except the introduction of a set of high-performance instructions

# Modes of operation

- Protected mode
  - native mode (Windows, Linux), full features, separate memory

- Virtual-8086 mode
  - hybrid of Protected
  - each program has its own 8086 computer

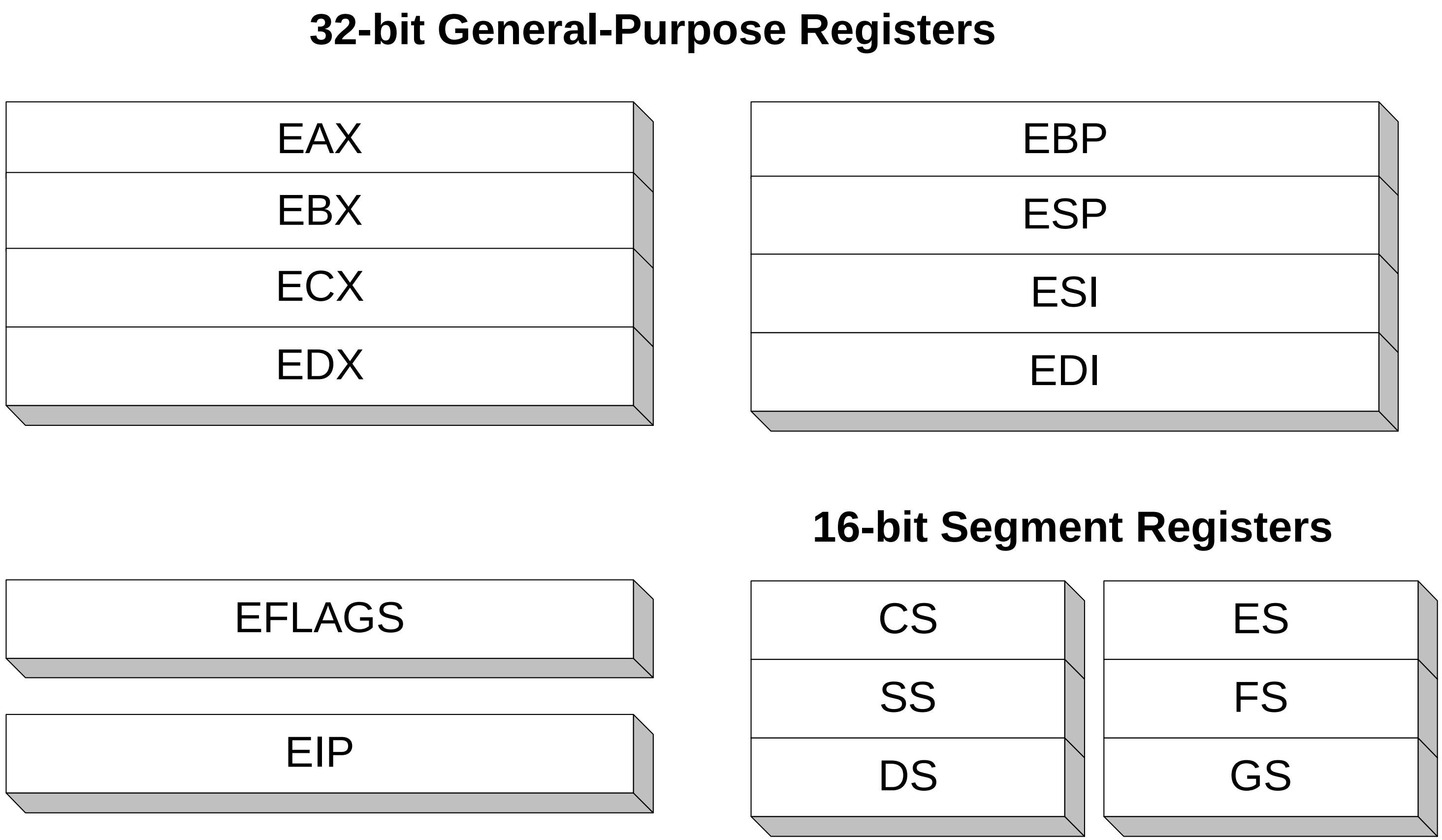
- Real-address mode
  - native MS-DOS
- System management mode
  - power management, system security, diagnostics

# Addressable memory

- Protected mode
  - 4 GB
  - 32-bit address
- Real-address and Virtual-8086 modes
  - 1 MB space
  - 20-bit address

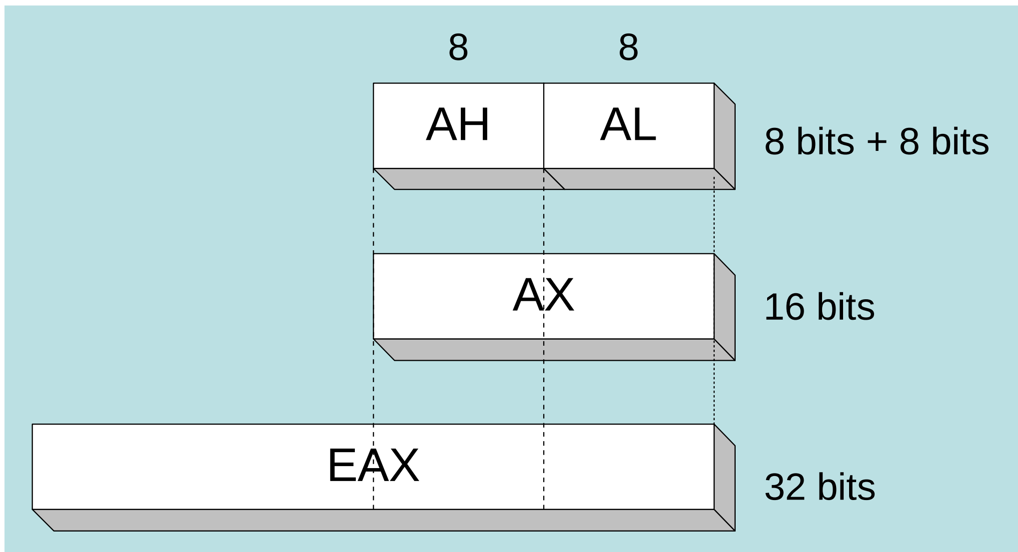
# General-purpose registers

Named storage locations inside the CPU, optimized for speed.



# Accessing parts of registers

- Use 8-bit name, 16-bit name, or 32-bit name
- Applies to EAX, EBX, ECX, and EDX



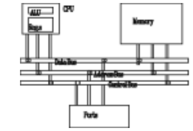

| 32-bit | 16-bit | 8-bit (high) | 8-bit (low) |
|---|---|---|---|
| EAX | AX | AH | AL |
| EBX | BX | BH | BL |
| ECX | CX | CH | CL |
| EDX | DX | DH | DL |

# Index and base registers

- Some registers have only a 16-bit name for their lower half. The 16-bit registers are usually used only in real-address mode.

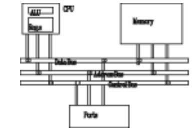
| 32-bit | 16-bit |
|---|---|
| ESI | SI |
| EDI | DI |
| EBP | BP |
| ESP | SP |

# Some specialized register uses (1 of 2)



- General-Purpose
  - EAX – accumulator (automatically used by division and multiplication)
  - ECX – loop counter
  - ESP – stack pointer (should never be used for arithmetic or data transfer)
  - ESI, EDI – index registers (used for high-speed memory transfer instructions)
  - EBP – extended frame pointer (stack)

# Some specialized register uses (2 of 2)



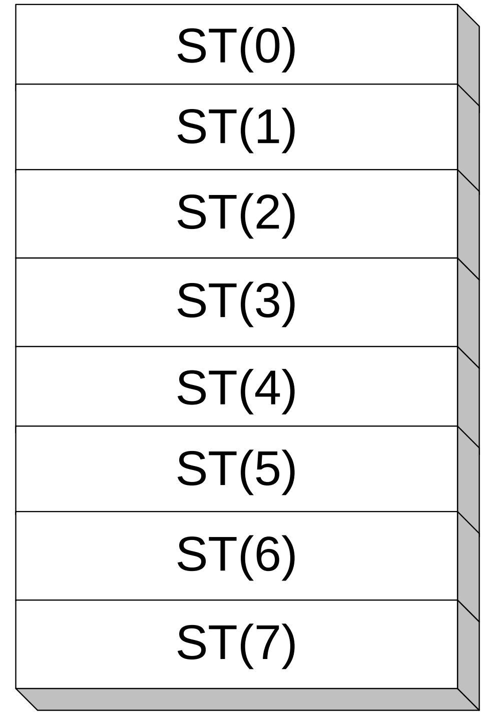
- Segment
  - CS – code segment
  - DS – data segment
  - SS – stack segment
  - ES, FS, GS - additional segments
- EIP – instruction pointer
- EFLAGS
  - status and control flags
  - each flag is a single binary bit (*set* or *clear*)

# Status flags

- Carry
  - unsigned arithmetic out of range
- Overflow
  - signed arithmetic out of range
- Sign
  - result is negative
- Zero
  - result is zero
- Auxiliary Carry
  - carry from bit 3 to bit 4
- Parity
  - sum of 1 bits is an even number

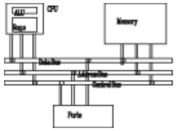
# Floating-point, MMX, XMM registers

- Eight 80-bit floating-point data registers
  - ST(0), ST(1), . . . , ST(7)
  - arranged in a stack
  - used for all floating-point arithmetic
- Eight 64-bit MMX registers
- Eight 128-bit XMM registers for single-instruction multiple-data (SIMD) operations



# IA-32 Memory Management

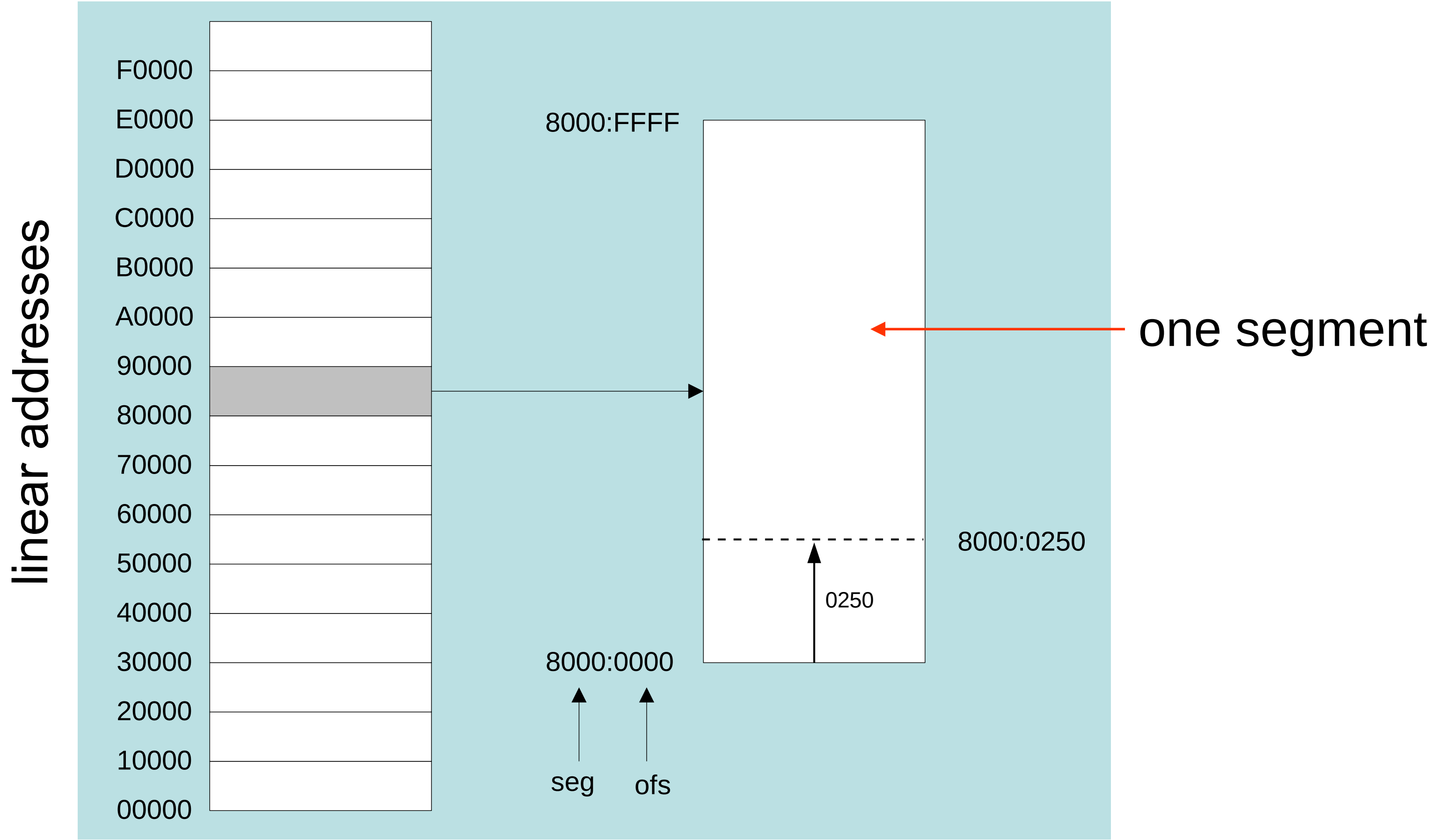
# Real-address mode



- 1 MB RAM maximum addressable (20-bit address)
- Application programs can access any area of memory
- Single tasking
- Supported by MS-DOS operating system

# Segmented memory

Segmented memory addressing: absolute (linear) address is a combination of a 16-bit segment value added to a 16-bit offset



# Calculating linear addresses

- Given a segment address, multiply it by 16 (add a hexadecimal zero), and add it to the offset
- Example: convert 08F1:0100 to a linear address

| | |
|---|---|
| **Adjusted Segment value:** | **0 8 F 1 0** |
| **Add the offset:** | **0 1 0 0** |
| **Linear address:** | **0 9 0 1 0** |

- A typical program has three segments: code, data and stack. Segment registers CS, DS and SS are used to store them separately.

# Example

What linear address corresponds to the segment/offset address 028F:0030?

028F0 + 0030 = 02920

Always use hexadecimal notation for addresses.

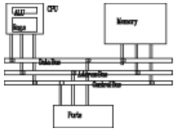
# Example

What segment addresses correspond to the linear address 28F30h?

Many different segment-offset addresses can produce the linear address 28F30h. For example:

28F0:0030, 28F3:0000, 28B0:0430, . . .

# Protected mode (1 of 2)



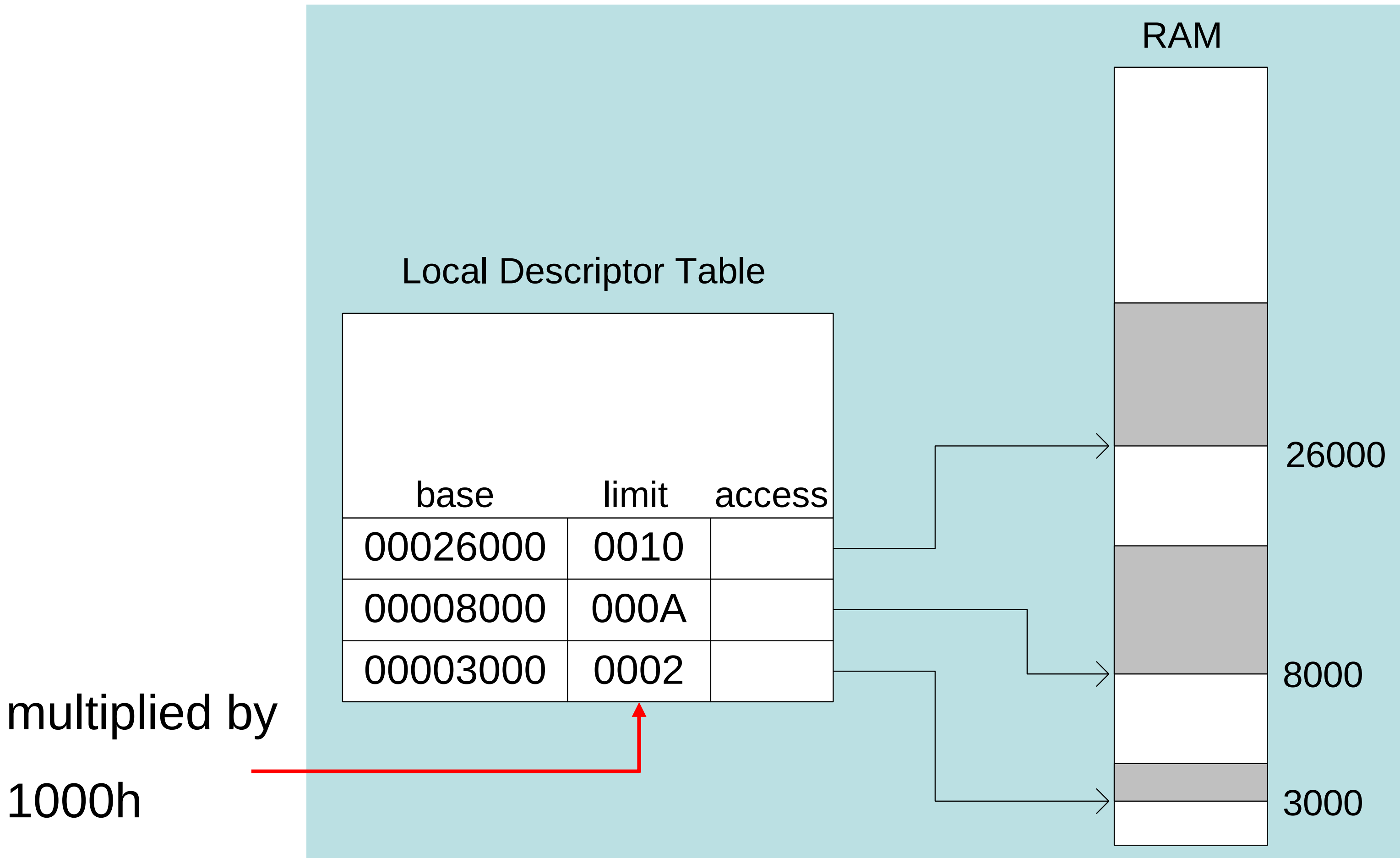

- 4 GB addressable RAM (32-bit address)
  - (00000000 to FFFFFFFFh)
- Each program assigned a memory partition which is protected from other programs
- Designed for multitasking
- Supported by Linux & MS-Windows

# Protected mode (2 of 2)

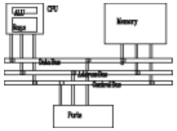
- Segment descriptor tables
- Program structure
  - code, data, and stack areas
  - CS, DS, SS segment descriptors
  - global descriptor table (GDT)
- MASM Programs use the Microsoft flat memory model

# Multi-segment model

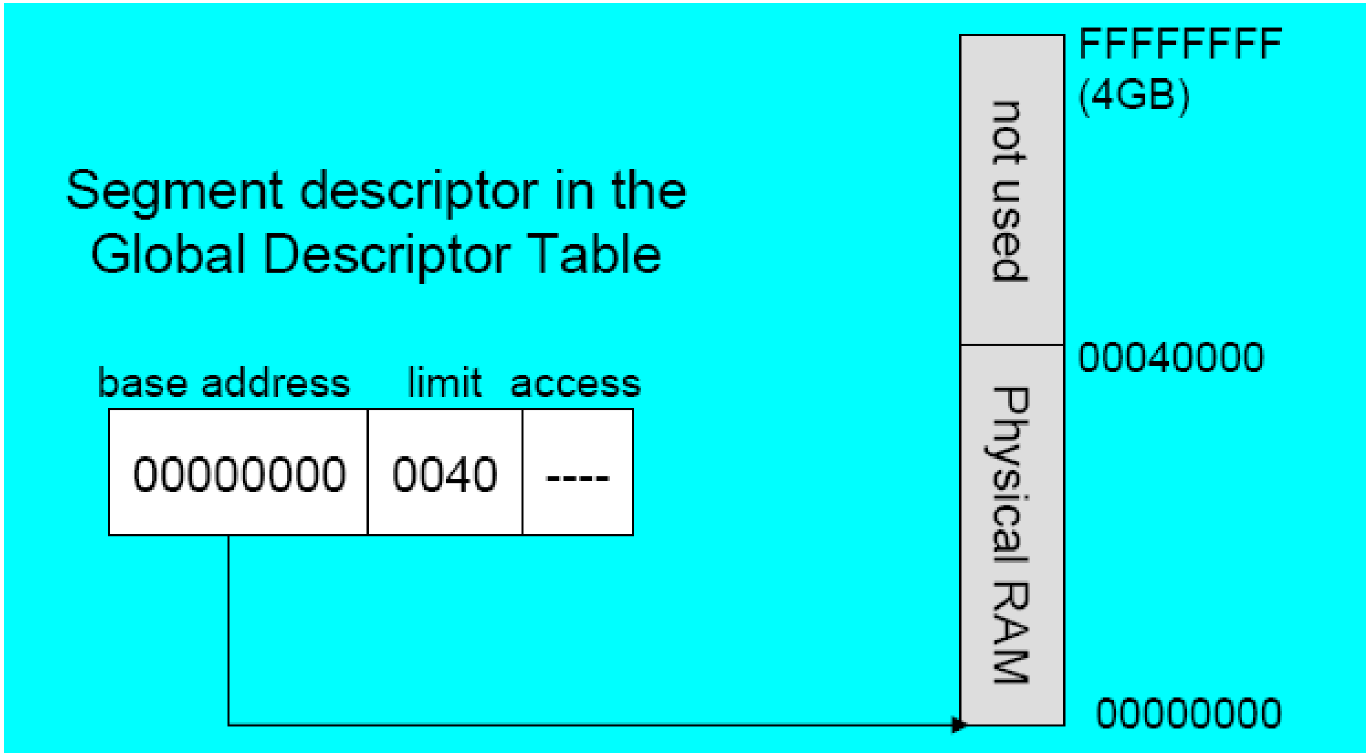
- Each program has a local descriptor table (LDT)
  - holds descriptor for each segment used by the program



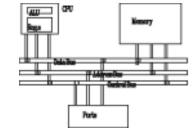
# Flat segmentation model



- All segments are mpped to the entire 32-bit physical address space, at least two, one for data and one for code
- global descriptor table (GDT)



# Paging

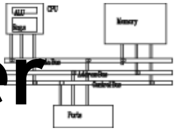


- Virtual memory uses disk as part of the memory, thus allowing sum of all programs can be larger than physical memory
- Divides each segment into 4096-byte blocks called pages

- Page fault (supported directly by the CPU) – issued by CPU when a page must be loaded from disk
- Virtual memory manager (VMM) – OS utility that manages the loading and unloading of pages

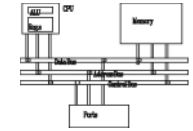
# Components of an IA-32 microcomputer

# Components of an IA-32 Microcomputer



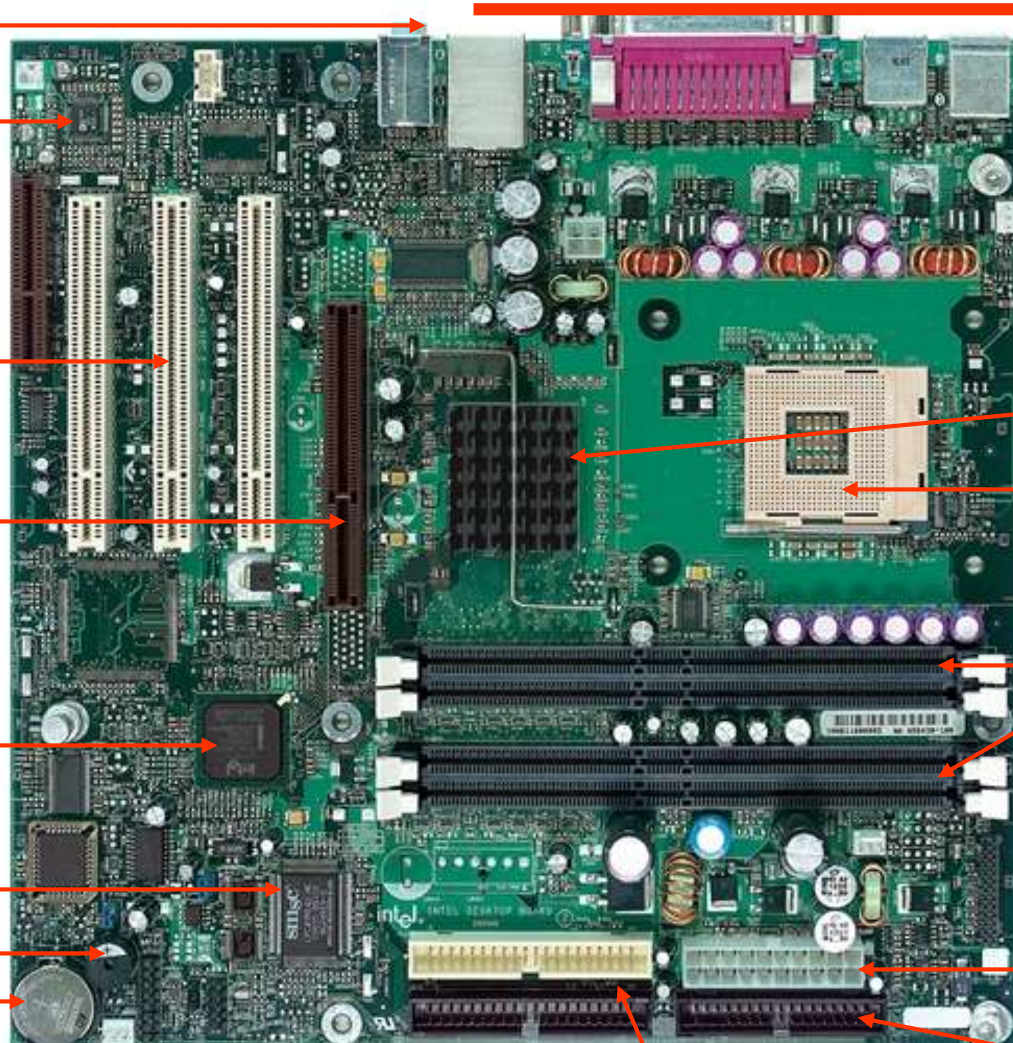

- Motherboard
- Video output
- Memory
- Input-output ports

# Motherboard



- CPU socket
- External cache memory slots
- Main memory slots
- BIOS chips
- Sound synthesizer chip (optional)
- Video controller chip (optional)
- IDE, parallel, serial, USB, video, keyboard, joystick, network, and mouse connectors
- PCI bus connectors (expansion cards)

# Intel D850MD motherboard

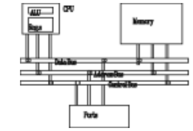



Source: Intel® Desktop Board D850MD/D850MV Technical Product Specification

# Video Output

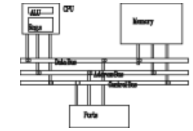
- Video controller
  - on motherboard, or on expansion card
  - AGP (accelerated graphics port)
- Video memory (VRAM)
- Video CRT Display
  - uses raster scanning
  - horizontal retrace
  - vertical retrace
- Direct digital LCD monitors
  - no raster scanning required

# Memory



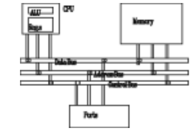
- ROM
  - read-only memory
- EPROM
  - erasable programmable read-only memory
- Dynamic RAM (DRAM)
  - inexpensive; must be refreshed constantly
- Static RAM (SRAM)
  - expensive; used for cache memory; no refresh required
- Video RAM (VRAM)
  - dual ported; optimized for constant video refresh
- CMOS RAM
  - refreshed by a battery
  - system setup information

# Input-output ports



- USB (universal serial bus)
  - intelligent high-speed connection to devices
  - up to 12 megabits/second
  - USB hub connects multiple devices
  - *enumeration*: computer queries devices
  - supports *hot* connections
- Parallel
  - short cable, high speed
  - common for printers
  - bidirectional, parallel data transfer
  - Intel 8255 controller chip

# Input-output ports (cont)



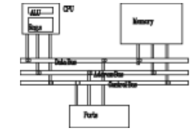
- Serial
  - RS-232 serial port
  - one bit at a time
  - used for long cables and modems
  - 16550 UART (universal asynchronous receiver transmitter)
  - programmable in assembly language

# Intel microprocessor history

# Early Intel microprocessors

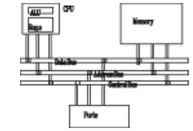
- Intel 8080
  - 64K addressable RAM
  - 8-bit registers
  - CP/M operating system
  - 5,6,8,10 MHz
  - 29K transistros
- Intel 8086/8088 (1978)
  - IBM-PC used 8088
  - 1 MB addressable RAM
  - 16-bit registers
  - 16-bit data bus (8-bit for 8088)
  - separate floating-point unit (8087)
  - used in low-cost microcontrollers now

# The IBM-AT



- Intel 80286 (1982)
  - 16 MB addressable RAM
  - Protected memory
  - several times faster than 8086
  - introduced IDE bus architecture
  - 80287 floating point unit
  - Up to 20MHz
  - 134K transistors

# Intel IA-32 Family

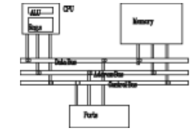


- Intel386 (1985)
  - 4 GB addressable RAM
  - 32-bit registers
  - paging (virtual memory)
  - Up to 33MHz
- Intel486 (1989)
  - instruction pipelining
  - Integrated FPU
  - 8K cache
- Pentium (1993)
  - Superscalar (two parallel pipelines)

# Intel P6 Family

- Pentium Pro (1995)
  - advanced optimization techniques in microcode
  - More pipeline stages
  - On-board L2 cache
- Pentium II (1997)
  - MMX (multimedia) instruction set
  - Up to 450MHz
- Pentium III (1999)
  - SIMD (streaming extensions) instructions (SSE)
  - Up to 1+GHz
- Pentium 4 (2000)
  - NetBurst micro-architecture, tuned for multimedia
  - 3.8+GHz
- Pentium D (Dual core)

# CISC and RISC



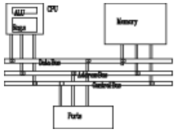
- CISC – complex instruction set
  - large instruction set
  - high-level operations (simpler for compiler?)
  - requires microcode interpreter (could take a long time)
  - examples: Intel 80x86 family
- RISC – reduced instruction set
  - small instruction set
  - simple, atomic instructions
  - directly executed by hardware very quickly
  - easier to incorporate advanced architecture design
  - examples:
    - ARM (Advanced RISC Machines)
    - DEC Alpha (now Compaq)

# Assembly Fundamentals

## ***Computer Organization and Assembly Languages***

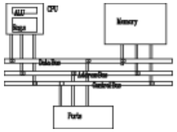
*with slides by Kip Irvine*

# Announcements



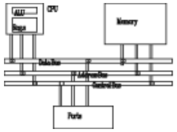
- Homework#1 assigned, due on 10/27
- Next week's class (10/20) will be taught by TAs
- Midterm examination will be held on the week of 11/10

# Chapter Overview



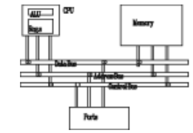
- Basic Elements of Assembly Language
- Example: Adding and Subtracting Integers
- Assembling, Linking, and Running Programs
- Defining Data
- Symbolic Constants

# Basic elements of assembly language



- Integer constants
- Integer expressions
- Character and string constants
- Reserved words and identifiers
- Directives and instructions
- Labels
- Mnemonics and Operands
- Comments
- Examples

# Integer constants



- [{+|-}] *digits* [*radix*]
- Optional leading + or – sign
- binary, decimal, hexadecimal, or octal digits
- Common radix characters:
  - **`h`** – hexadecimal
  - **`d`** – decimal (default)
  - **`b`** – binary
  - **`r`** – encoded real
  - **`o`** – octal

Examples: **`30d, 6Ah, 42, 42o, 1101b`**

Hexadecimal beginning with letter: **`0A5h`**

# Integer expressions

- Operators and precedence levels:

| Operator | Name | Precedence Level |
|---|---|---|
| ( ) | parentheses | 1 |
| +, - | unary plus, minus | 2 |
| *, / | multiply, divide | 3 |
| MOD | modulus | 3 |
| +, - | add, subtract | 4 |

- Examples:

| Expression | Value |
|---|---|
| 16 / 5 | 3 |
| -(3 + 4) * (6 - 1) | -35 |
| -3 + 4 * 6 - 1 | 20 |
| 25 mod 3 | 1 |

# Real number constants (encoded reals)

- Fixed point v.s. floating point

| 1 | 8 | 23 |
|---|---|---|
| S | E | M |

$$\pm 1.bbbb \times 2^{(E-127)}$$

- Example **`3F800000r=+1.0,37.75=42170000r`**

- double

| 1 | 11 | 52 |
|---|---|---|
| S | E | M |

# Real number constants (decimal reals)

- [*sign*]*integer*.[*integer*][*exponent*]

  sign $\longrightarrow$ {+|-}

  exponent $\longrightarrow$ E[{+|-}]integer

- Examples:

```
2.
+3.0
-44.2E+05
26.E5
```

# Character and string constants

- Enclose character in single or double quotes
  - `'A', "x"`
  - ASCII character = 1 byte
- Enclose strings in single or double quotes
  - `"ABC"`
  - `'xyz'`
  - Each character occupies a single byte
- Embedded quotes:
  - `‘Say "Goodnight," Gracie’`
  - `"This isn't a test"`

# Reserved words and identifiers

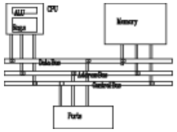
- Reserved words (Appendix D) cannot be used as identifiers
  - Instruction mnemonics, directives, type attributes, operators, predefined symbols
- Identifiers
  - 1-247 characters, including digits
  - case insensitive (by default)
  - first character must be a letter, `_`, `@`, or `$`
  - examples:

```
var1        Count      $first
_main       MAX        open_file
@@myfile    xVal       _12345
```

# Directives

- Commands that are recognized and acted upon by the assembler
  - Part of assembler's syntax but not part of the Intel instruction set
  - Used to declare code, data areas, select memory model, declare procedures, etc.
  - case insensitive
- Different assemblers have different directives
  - NASM != MASM, for example
- Examples: `.data` `.code` `PROC`

# Instructions



- Assembled into machine code by assembler
- Executed at runtime by the CPU
- Member of the Intel IA-32 instruction set
- Four parts
  - Label (optional)
  - Mnemonic (required)
  - Operand (usually required)
  - Comment (optional)

| | | | |
|---|---|---|---|
| Label: | Mnemonic | Operand(s) | ;Comment |

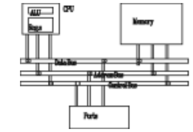
# Labels

- Act as place markers
  - marks the address (offset) of code and data
- Easier to memorize and more flexible

  **`mov ax, [0020]`** → **`mov ax, val`**
- Follow identifier rules
- Data label
  - must be unique
  - example: **`myArray  BYTE  10`**
- Code label
  - target of jump and loop instructions
  - example:

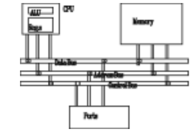
```
L1: mov ax, bx
    ...
    jmp L1
```

# Mnemonics and operands



- Instruction mnemonics
  - "reminder"
  - examples: **`MOV, ADD, SUB, MUL, INC, DEC`**
- Operands
  - constant (immediate value), **`96`**
  - constant expression, **`2+4`**
  - Register, **`eax`**
  - memory (data label), **`count`**
- Number of operands: 0 to 3
  - **`stc               ; set Carry flag`**
  - **`inc ax            ; add 1 to ax`**
  - **`mov count, bx     ; move BX to count`**

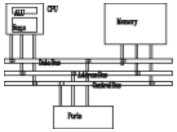
# Comments



- Comments are good!
  - explain the program's purpose
  - tricky coding techniques
  - application-specific explanations
- Single-line comments
  - begin with semicolon (;)
- block comments
  - begin with COMMENT directive and a programmer-chosen character and end with the same programmer-chosen character

```
COMMENT !
   This is a comment
   and this line is also a comment
!
```

# Example: adding/subtracting integers



directive marks comment

```
TITLE Add and Subtract                 (AddSub.asm)
```

comment

```
; This program adds and subtracts 32-bit integers.
```

```
INCLUDE Irvine32.inc
```

copy definitions from Irvine32.inc

```
.code
```

code segment. 3 segments: code, data, stack

```
main PROC
```

beginning of a procedure

```
    mov eax,10000h          ; EAX = 10000h
    add eax,40000h          ; EAX = 50000h
    sub eax,20000h          ; EAX = 30000h
    call DumpRegs           ; display registers
```

source

destination

```
    exit
main ENDP
```

defined in Irvine32.inc to end a program

```
END main
```

mark the last line and startup procedure

# Example output

Program output, showing registers and flags:

```
EAX=00030000  EBX=7FFDF000  ECX=00000101  EDX=FFFFFFFF
ESI=00000000  EDI=00000000  EBP=0012FFF0  ESP=0012FFC4
EIP=00401024  EFL=00000206  CF=0  SF=0  ZF=0  OF=0
```

# Suggested coding standards (1 of 2)

- Some approaches to capitalization
  - capitalize nothing
  - capitalize everything
  - capitalize all reserved words, including instruction mnemonics and register names
  - capitalize only directives and operators (used by the book)
- Other suggestions
  - descriptive identifier names
  - spaces surrounding arithmetic operators
  - blank lines between procedures

# Suggested coding standards (2 of 2)

- Indentation and spacing
  - code and data labels – no indentation
  - executable instructions – indent 4-5 spaces
  - comments: begin at column 40-45, aligned vertically
  - 1-3 spaces between instruction and its operands
    - ex: `mov  ax,bx`
  - 1-2 blank lines between procedures

# Alternative version of AddSub

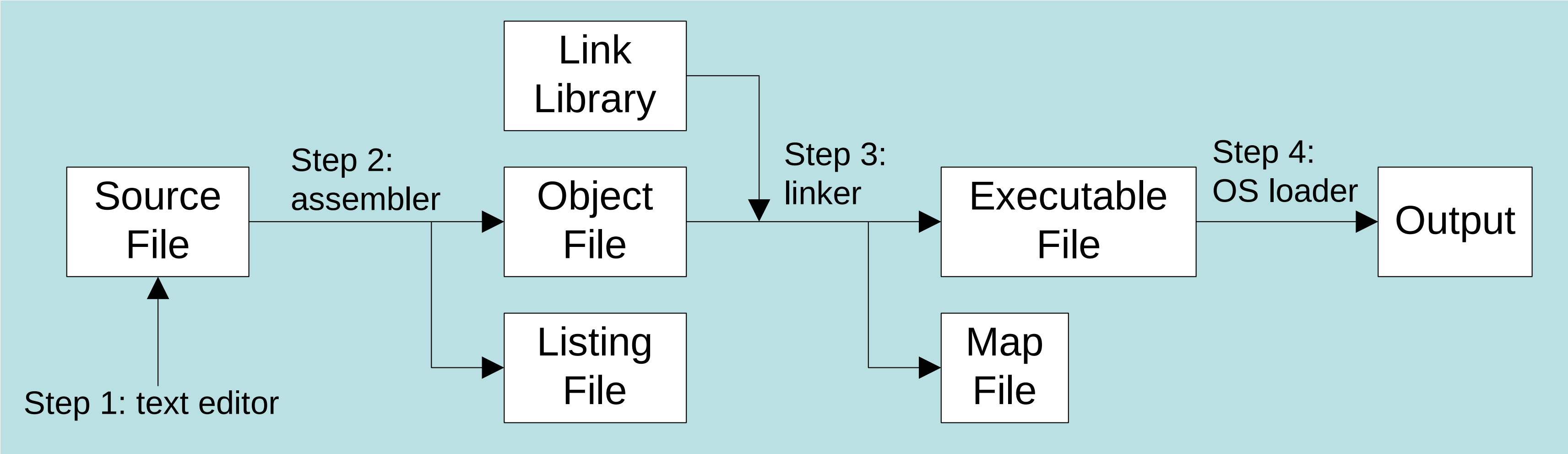
```
TITLE Add and Subtract                    (AddSubAlt.asm)

; This program adds and subtracts 32-bit integers.
.386
.MODEL flat,stdcall
.STACK 4096

ExitProcess PROTO, dwExitCode:DWORD
DumpRegs PROTO

.code
main PROC
    mov eax,10000h               ; EAX = 10000h
    add eax,40000h               ; EAX = 50000h
    sub eax,20000h               ; EAX = 30000h
    call DumpRegs
    INVOKE ExitProcess,0
main ENDP
END main
```

# Program template

```
TITLE Program Template            (Template.asm)

; Program Description:
; Author:
; Creation Date:
; Revisions:
; Date:              Modified by:

INCLUDE Irvine32.inc
.data
    ; (insert variables here)
.code
main PROC
    ; (insert executable instructions here)
    exit
main ENDP
    ; (insert additional procedures here)
END main
```

# Assemble-link execute cycle

- The following diagram describes the steps from creating a source program through executing the compiled program.
- If the source code is modified, Steps 2 through 4 must be repeated.



# make32.bat

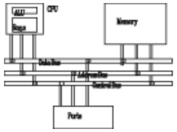
- Called a batch file
- Run it to assemble and link programs
- Contains a command that executes ML.EXE (the Microsoft Assembler)
- Contains a command that executes LINK32.EXE (the 32-bit Microsoft Linker)
- Command-Line syntax:

  **make32** ***progName***

  (*progName* includes the .asm extension)

(use make16.bat to assemble and link Real-mode programs)

# Listing file



- Use it to see how your program is compiled
- Contains
  - source code
  - addresses
  - object code (machine language)
  - segment names
  - symbols (variables, procedures, and constants)
- Example: addSub.lst

# Defining data

- Intrinsic data types
- Data Definition Statement
- Defining BYTE and SBYTE Data
- Defining WORD and SWORD Data
- Defining DWORD and SDWORD Data
- Defining QWORD Data
- Defining TBYTE Data
- Defining Real Number Data
- Little Endian Order
- Adding Variables to the AddSub Program
- Declaring Uninitialized Data

# Intrinsic data types (1 of 2)

- **`BYTE`**, **`SBYTE`**
  - 8-bit unsigned integer; 8-bit signed integer
- **`WORD`**, **`SWORD`**
  - 16-bit unsigned & signed integer
- **`DWORD`**, **`SDWORD`**
  - 32-bit unsigned & signed integer
- **`QWORD`**
  - 64-bit integer
- **`TBYTE`**
  - 80-bit integer

# Intrinsic data types (2 of 2)

- **`REAL4`**
  - 4-byte IEEE short real
- **`REAL8`**
  - 8-byte IEEE long real
- **`REAL10`**
  - 10-byte IEEE extended real

# Data definition statement

- A data definition statement sets aside storage in memory for a variable.
- May optionally assign a name (label) to the data
- Syntax:

  [*name*] *directive initializer* [,*initializer*] . . .

  At least one initializer is required, can be `?`
- All initializers become binary data in memory

# Defining BYTE and SBYTE Data

Each of the following defines a single byte of storage:

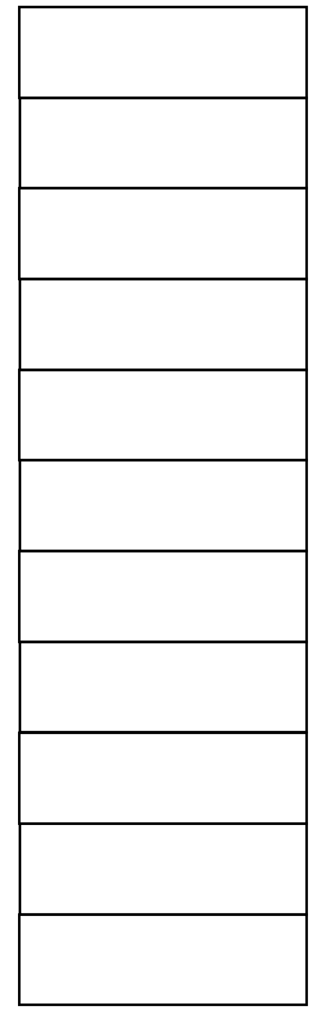
```
value1 BYTE 'A'          ; character constant
value2 BYTE 0            ; smallest unsigned byte
value3 BYTE 255          ; largest unsigned byte
value4 SBYTE -128        ; smallest signed byte
value5 SBYTE +127        ; largest signed byte
value6 BYTE ?            ; uninitialized byte
```

A variable name is a data label that implies an offset (an address).

# Defining multiple bytes

Examples that use multiple initializers:

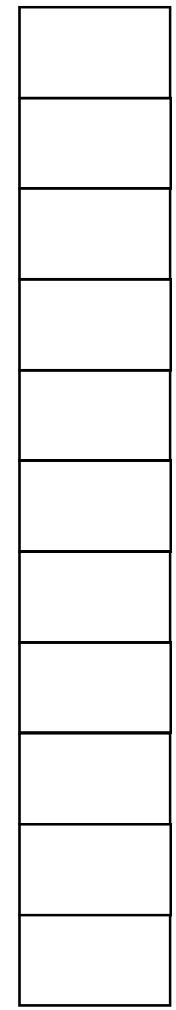
```
list1 BYTE 10,20,30,40
list2 BYTE 10,20,30,40
      BYTE 50,60,70,80
      BYTE 81,82,83,84
list3 BYTE ?,32,41h,00100010b
list4 BYTE 0Ah,20h,'A',22h
```



# Defining strings (1 of 2)

- A string is implemented as an array of characters
  - For convenience, it is usually enclosed in quotation marks
  - It usually has a null byte at the end
- Examples:

```
str1 BYTE "Enter your name",0
str2 BYTE 'Error: halting program',0
str3 BYTE 'A','E','I','O','U'
greeting1 BYTE "Welcome to the Encryption Demo program "
          BYTE "created by Kip Irvine.",0
greeting2 \
    BYTE "Welcome to the Encryption Demo program "
    BYTE "created by Kip Irvine.",0
```



# Defining strings (2 of 2)

- End-of-line character sequence:
  - 0Dh = carriage return
  - 0Ah = line feed

```
str1 BYTE "Enter your name:    ",0Dh,0Ah
     BYTE "Enter your address: ",0

newLine BYTE 0Dh,0Ah,0
```

Idea: Define all strings used by your program in the same area of the data segment.

# Using the DUP operator

- Use **DUP** to allocate (create space for) an array or string.
- Counter and argument must be constants or constant expressions

```
var1 BYTE 20 DUP(0)             ; 20 bytes, all equal to zero
var2 BYTE 20 DUP(?)             ; 20 bytes, uninitialized
var3 BYTE 4 DUP("STACK")        ; 20 bytes: "STACKSTACKSTACKSTACK"
var4 BYTE 10,3 DUP(0),20
```

# Defining WORD and SWORD data

- Define storage for 16-bit integers
  - or double characters
  - single value or multiple values

```
word1 WORD   65535          ; largest unsigned value
word2 SWORD –32768          ; smallest signed value
word3 WORD   ?              ; uninitialized, unsigned
word4 WORD "AB"             ; double characters
myList WORD 1,2,3,4,5       ; array of words
array WORD 5 DUP(?)         ; uninitialized array
```

# Defining DWORD and SDWORD data

Storage definitions for signed and unsigned 32-bit integers:

```
val1 DWORD 12345678h            ; unsigned
val2 SDWORD -2147483648         ; signed
val3 DWORD 20 DUP(?)            ; unsigned array
val4 SDWORD -3,-2,-1,0,1        ; signed array
```

# Defining `QWORD`, `TBYTE`, Real Data

Storage definitions for quadwords, tenbyte values, and real numbers:

```
quad1 QWORD  1234567812345678h
val1  TBYTE  1000000000123456789Ah
rVal1 REAL4  -2.1
rVal2 REAL8  3.2E-260
rVal3 REAL10 4.6E+4096
ShortArray REAL4 20 DUP(0.0)
```

# Little Endian order

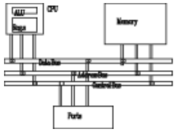
- All data types larger than a byte store their individual bytes in reverse order. The least significant byte occurs at the first (lowest) memory address.

- Example:

```
val1 DWORD 12345678h
```

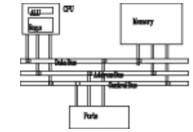
| | |
|---|---|
| 0000: | 78 |
| 0001: | 56 |
| 0002: | 34 |
| 0003: | 12 |

# Adding variables to AddSub



```
TITLE Add and Subtract, Version 2              (AddSub2.asm)
; This program adds and subtracts 32-bit unsigned
; integers and stores the sum in a variable.
INCLUDE Irvine32.inc
.data
val1 DWORD 10000h
val2 DWORD 40000h
val3 DWORD 20000h
finalVal DWORD ?
.code
main PROC
    mov eax,val1                   ; start with 10000h
    add eax,val2                   ; add 40000h
    sub eax,val3                   ; subtract 20000h
    mov finalVal,eax               ; store the result (30000h)
    call DumpRegs                  ; display the registers
    exit
main ENDP
END main
```

# Declaring unitialized data



- Use the **`.data?`** directive to declare an unintialized data segment:

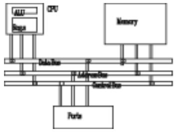
```
.data?
```

- Within the segment, declare variables with "?" initializers:

Advantage: the program's EXE file size is reduced.

```
.data
smallArray DWORD 10 DUP(0)
.data?
bigArray   DWORD 5000 DUP(?)
```

# Mixing code and data

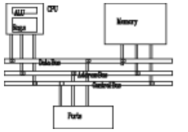


```
.code
mov eax, ebx
.data
temp DWORD ?
.code
mov temp, eax
```

# Symbolic constants

- Equal-Sign Directive
- Calculating the Sizes of Arrays and Strings
- EQU Directive
- TEXTEQU Directive

# Equal-sign directive

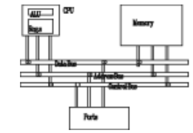


- *name* = *expression*
  - expression is a 32-bit integer (expression or constant)
  - may be redefined
  - *name* is called a symbolic constant
- good programming style to use symbols
  - Easier to modify
  - Easier to understand, **`ESC_key`**
  - **`Array DWORD COUNT DUP(0)`**
  - **`COUNT=5`**

```
 Mov al, COUNT
 COUNT=10
 Mov al, COUNT
```

```
COUNT = 500
.
mov al,COUNT
```

# Calculating the size of a byte array



- current location counter: $
  - subtract address of list
  - difference is the number of bytes

```
list BYTE 10,20,30,40
ListSize = 4
```

```
list BYTE 10,20,30,40
ListSize = ($ - list)
```

```
list BYTE 10,20,30,40
Var2 BYTE 20 DUP(?)
ListSize = ($ - list)
```

```
myString BYTE “This is a long string.”
myString_len = ($ - myString)
```

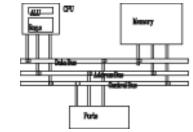
# Calculating the size of a word array

- current location counter: $
  - subtract address of list
  - difference is the number of bytes
  - divide by 2 (the size of a word)

```
list WORD 1000h,2000h,3000h,4000h
ListSize = ($ - list) / 2
```

```
list DWORD 1,2,3,4
ListSize = ($ - list) / 4
```

# EQU directive



- name EQU expression

  name EQU symbol

  name EQU <text>
- Define a symbol as either an integer or text expression.
- Can be useful for non-integer constant
- Cannot be redefined

# EQU directive

```
PI EQU <3.1416>
pressKey EQU <"Press any key to continue...",0>
.data
prompt BYTE pressKey
```

```
Matrix1 EQU 10*10
matrix1 EQU <10*10>
.data
M1 WORD matrix1              ; M1 WORD 100
M2 WORD matrix2              ; M2 WORD 10*10
```

# TEXTEQU directive

- name TEXTEQU <text>
  name TEXTEQU textmacro
  name TEXTEQU %constExpr
- Define a symbol as either an integer or text expression.
- Called a text macro
- Can be redefined

```
continueMsg TEXTEQU <"Do you wish to continue (Y/N)?">
rowSize = 5
.data
prompt1 BYTE continueMsg
count TEXTEQU %(rowSize * 2)       ; evaluates the expression
move TEXTEQU <mov>
setupAL TEXTEQU <move al,count>
.code
setupAL                            ; generates: "mov al,10"
```

# Chapter recap

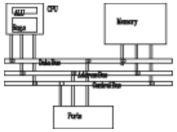
- Basic Elements of Assembly Language
- Example: Adding and Subtracting Integers
- Assembling, Linking, and Running Programs
- Defining Data
- Symbolic Constants

# Data Transfer, Addressing and Arithmetic

***Computer Organization and Assembly Languages***

*with slides by Kip Irvine*

# Chapter overview

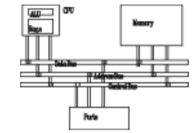


- Data Transfer Instructions
- Addition and Subtraction
- Data-Related Operators and Directives
- Indirect Addressing
- JMP and LOOP Instructions

# Data transfer instructions

- Operand Types
- Instruction Operand Notation
- Direct Memory Operands
- MOV Instruction
- Zero & Sign Extension
- XCHG Instruction
- Direct-Offset Instructions

# Operand types



- Three basic types of operands:
  - Immediate – a constant integer (8, 16, or 32 bits)
    - value is encoded within the instruction
  - Register – the name of a register
    - register name is converted to a number and encoded within the instruction
  - Memory – reference to a location in memory
    - memory address is encoded within the instruction, or a register holds the address of a memory location

# Instruction operand notation

| Operand | Description |
|---|---|
| *r8* | 8-bit general-purpose register: AH, AL, BH, BL, CH, CL, DH, DL |
| *r16* | 16-bit general-purpose register: AX, BX, CX, DX, SI, DI, SP, BP |
| *r32* | 32-bit general-purpose register: EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP |
| *reg* | any general-purpose register |
| *sreg* | 16-bit segment register: CS, DS, SS, ES, FS, GS |
| *imm* | 8-, 16-, or 32-bit immediate value |
| *imm8* | 8-bit immediate byte value |
| *imm16* | 16-bit immediate word value |
| *imm32* | 32-bit immediate doubleword value |
| *r/m8* | 8-bit operand which can be an 8-bit general register or memory byte |
| *r/m16* | 16-bit operand which can be a 16-bit general register or memory word |
| *r/m32* | 32-bit operand which can be a 32-bit general register or memory doubleword |
| *mem* | an 8-, 16-, or 32-bit memory operand |

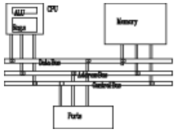
# Direct memory operands

- A direct memory operand is a named reference to storage in memory
- The named reference (label) is automatically dereferenced by the assembler

```
.data
var1 BYTE 10h,
.code
mov al,var1                  ; AL = 10h
mov al,[var1]                ; AL = 10h
```

alternate format

# MOV instruction



- Move from source to destination. Syntax:

  **MOV** *destination,source*

- Source and destination have the same size
- No more than one memory operand permitted
- CS, EIP, and IP cannot be the destination
- No immediate to segment moves

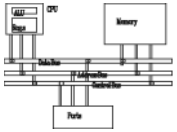
```
.data
count BYTE 100
wVal  WORD 2
.code
   mov bl,count
   mov ax,wVal
   mov count,al

   mov al,wVal                  ; error
   mov ax,count                 ; error
   mov eax,count                ; error
```

# Your turn . . .

Explain why each of the following **MOV** statements are invalid:

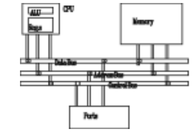
```
.data
bVal   BYTE   100
bVal2  BYTE   ?
wVal   WORD   2
dVal   DWORD  5
.code
    mov ds,45                    ; a.
    mov esi,wVal                 ; b.
    mov eip,dVal                 ; c.
    mov 25,bVal                  ; d.
    mov bVal2,bVal               ; e.
```

# Memory to memory



```
.data
var1 WORD ?
var2 WORD ?
.code
mov ax, var1
mov var2, ax
```

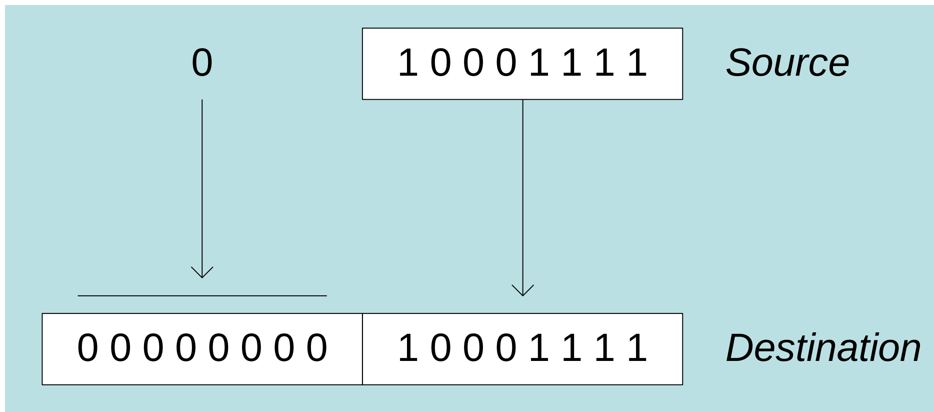
# Copy smaller to larger



```
.data
count WORD 1
.code
mov ecx, 0
mov cx, count

.data
signedVal SWORD -16 ; FFF0h
.code
mov ecx, 0              ; mov ecx, 0FFFFFFFFh
mov cx, signedVal
```

# Zero extension

When you copy a smaller value into a larger destination, the **MOVZX** instruction fills (extends) the upper half of the destination with zeros.

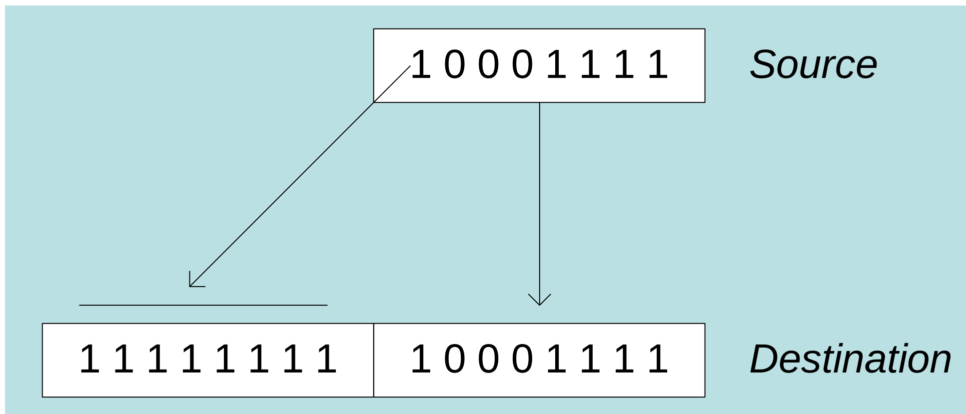



```
mov bl,10001111b
movzx ax,bl                ; zero-extension
```

The destination must be a register.

# Sign extension

The **MOVSX** instruction fills the upper half of the destination with a copy of the source operand's sign bit.




```
mov bl,10001111b
movsx ax,bl                        ; sign extension
```

The destination must be a register.

# MOVZX MOVSX

From a smaller location to a larger one

```
mov bx, 0A69Bh
movzx eax, bx           ; EAX=0000A69Bh
movzx edx, bl           ; EDX=0000009Bh
movzx cx, bl            ; EAX=009Bh

mov bx, 0A69Bh
movsx eax, bx           ; EAX=FFFFA69Bh
movsx edx, bl           ; EDX=FFFFFF9Bh
movsx cx, bl            ; EAX=FF9Bh
```

# LAHF SAHF

```
.data
saveflags BYTE ?
.code
lahf
mov saveflags, ah
...
mov ah, saveflags
sahf
```

# XCHG Instruction

**XCHG** exchanges the values of two operands. At least one operand must be a register. No immediate operands are permitted.

```
.data
var1 WORD 1000h
var2 WORD 2000h
.code
xchg ax,bx              ; exchange 16-bit regs
xchg ah,al              ; exchange 8-bit regs
xchg var1,bx            ; exchange mem, reg
xchg eax,ebx            ; exchange 32-bit regs

xchg var1,var2          ; error: two memory operands
```

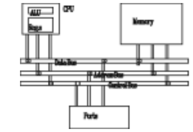
# Direct-offset operands

A constant offset is added to a data label to produce an effective address (EA). The address is dereferenced to get the value inside its memory location. (no range checking)

```
.data
arrayB BYTE 10h,20h,30h,40h
.code
mov al,arrayB+1                 ; AL = 20h
mov al,[arrayB+1]               ; alternative notation
```

Q: Why doesn't arrayB+1 produce 11h?

# Direct-offset operands (cont)



A constant offset is added to a data label to produce an effective address (EA). The address is dereferenced to get the value inside its memory location.

```
.data
arrayW  WORD 1000h,2000h,3000h
arrayD  DWORD 1,2,3,4
.code
mov ax,[arrayW+2]             ; AX = 2000h
mov ax,[arrayW+4]             ; AX = 3000h
mov eax,[arrayD+4]            ; EAX = 00000002h
```

```
; Will the following statements assemble and run?
mov ax,[arrayW-2]          ; ??
mov eax,[arrayD+16]        ; ??
```

# Your turn. . .

Write a program that rearranges the values of three doubleword values in the following array as: 3, 1, 2.

```
.data
arrayD DWORD 1,2,3
```

- Step1: copy the first value into EAX and exchange it with the value in the second position.

```
mov eax,arrayD
xchg eax,[arrayD+4]
```

- Step 2: Exchange EAX with the third array value and copy the value in EAX to the first array position.

```
xchg eax,[arrayD+8]
mov  arrayD,eax
```

# Evaluate this . . .

- We want to write a program that adds the following three bytes:

```
.data
myBytes BYTE 80h,66h,0A5h
```

- What is your evaluation of the following code?

```
mov al,myBytes
add al,[myBytes+1]
add al,[myBytes+2]
```

- What is your evaluation of the following code?

```
mov ax,myBytes
add ax,[myBytes+1]
add ax,[myBytes+2]
```

- Any other possibilities?

# Evaluate this . . . (cont)

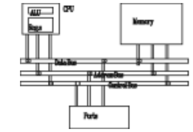
```
.data
myBytes BYTE 80h,66h,0A5h
```

- How about the following code. Is anything missing?

```
movzx ax,myBytes
mov   bl,[myBytes+1]
add   ax,bx
mov   bl,[myBytes+2]
add   ax,bx                    ; AX = sum
```

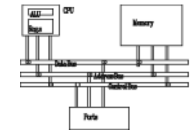
Yes: Move zero to BX before the MOVZX instruction.

# Addition and Subtraction



- INC and DEC Instructions
- ADD and SUB Instructions
- NEG Instruction
- Implementing Arithmetic Expressions
- Flags Affected by Arithmetic
  - Zero
  - Sign
  - Carry
  - Overflow

# INC and DEC Instructions

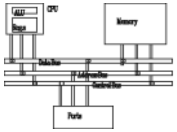


- Add 1, subtract 1 from destination operand
  - operand may be register or memory
- INC *destination*
  - Logic: *destination* ← *destination* + 1
- DEC *destination*
  - Logic: *destination* ← *destination* – 1

# INC and DEC Examples

```
.data
myWord  WORD 1000h
myDword DWORD 10000000h
.code
   inc myWord                    ; 1001h
   dec myWord                    ; 1000h
   inc myDword                   ; 10000001h

   mov ax,00FFh
   inc ax                        ; AX = 0100h
   mov ax,00FFh
   inc al                        ; AX = 0000h
```

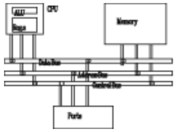
# Your turn...



Show the value of the destination operand after each of the following instructions executes:

```
.data
myByte BYTE 0FFh, 0
.code
    mov al,myByte               ; AL = FFh
    mov ah,[myByte+1]           ; AH = 00h
    dec ah                      ; AH = FFh
    inc al                      ; AL = 00h
    dec ax                      ; AX = FEFF
```

# ADD and SUB Instructions



- ADD destination, source
  - Logic: *destination* ← *destination* + source
- SUB destination, source
  - Logic: *destination* ← *destination* – source
- Same operand rules as for the MOV instruction

# ADD and SUB Examples

```
.data
var1 DWORD 10000h
var2 DWORD 20000h
.code                          ; ---EAX---
   mov eax,var1                ; 00010000h
   add eax,var2                ; 00030000h
   add ax,0FFFFh               ; 0003FFFFh
   add eax,1                   ; 00040000h
   sub ax,1                    ; 0004FFFFh
```

# NEG (negate) Instruction

Reverses the sign of an operand. Operand can be a register or memory operand.

```
.data
valB BYTE -1
valW WORD +32767
.code
   mov al,valB              ; AL = -1
   neg al                   ; AL = +1
   neg valW                 ; valW = -32767
```

Suppose AX contains –32,768 and we apply NEG to it. Will the result be valid?

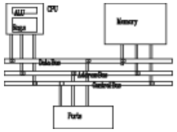
# Implementing Arithmetic Expressions

HLL compilers translate mathematical expressions into assembly language. You can do it also. For example:

**Rval = –Xval + (Yval – Zval)**

```
Rval DWORD ?
Xval DWORD 26
Yval DWORD 30
Zval DWORD 40
.code
    mov eax,Xval
    neg eax                         ; EAX = -26
    mov ebx,Yval
    sub ebx,Zval                    ; EBX = -10
    add eax,ebx
    mov Rval,eax                    ; -36
```

# Your turn...



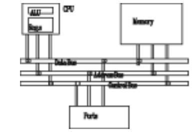
Translate the following expression into assembly language. Do not permit Xval, Yval, or Zval to be modified:

```
Rval = Xval - (-Yval + Zval)
```

Assume that all values are signed doublewords.

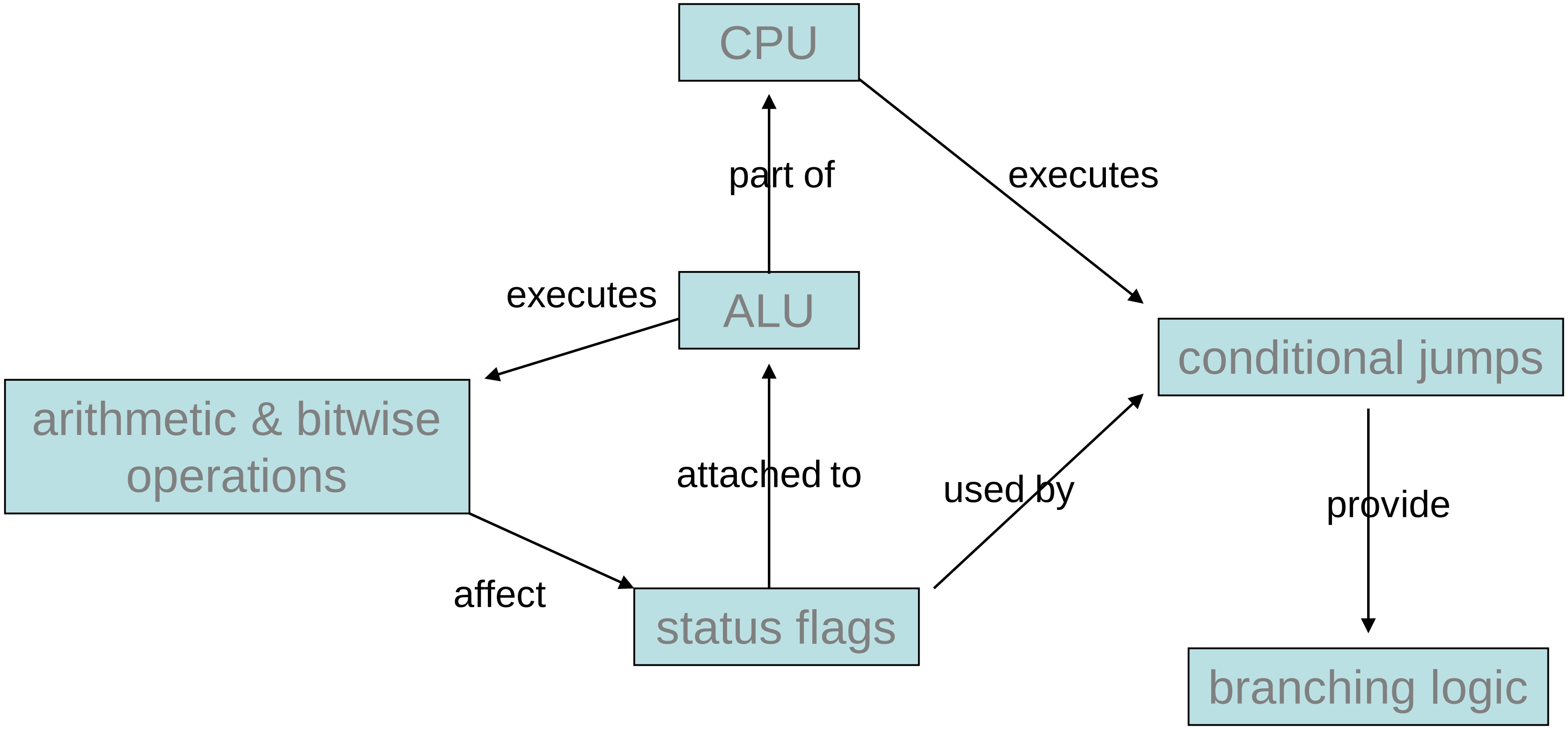
```
mov ebx,Yval
neg ebx
add ebx,Zval
mov eax,Xval
sub ebx
mov Rval,eax
```

# Flags Affected by Arithmetic



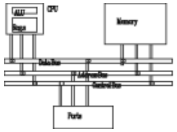
- The ALU has a number of status flags that reflect the outcome of arithmetic (and bitwise) operations
  - based on the contents of the destination operand
- Essential flags:
  - Zero flag – destination equals zero
  - Sign flag – destination is negative
  - Carry flag – unsigned value out of range
  - Overflow flag – signed value out of range
- The MOV instruction never affects the flags.

# Concept Map




You can use diagrams such as these to express the relationships between assembly language concepts.

# Zero Flag (ZF)



Whenever the destination operand equals Zero, the Zero flag is set.

```
mov cx,1
sub cx,1                    ; CX = 0, ZF = 1
mov ax,0FFFFh
inc ax                      ; AX = 0, ZF = 1
inc ax                      ; AX = 1, ZF = 0
```

A flag is set when it equals 1.

A flag is clear when it equals 0.

# Sign Flag (SF)

The Sign flag is set when the destination operand is negative. The flag is clear when the destination is positive.

```
mov cx,0
sub cx,1                                ; CX = -1, SF = 1
add cx,2                                ; CX = 1, SF = 0
```

The sign flag is a copy of the destination's highest bit:

```
mov al,0
sub al,1                    ; AL = 11111111b, SF = 1
add al,2                    ; AL = 00000001b, SF = 0
```

# Carry Flag (CF)

The Carry flag is set when the result of an operation generates an unsigned value that is out of range (too big or too small for the destination operand).

```
mov al,0FFh
add al,1                        ; CF = 1, AL = 00

; Try to go below zero:

mov al,0
sub al,1                        ; CF = 1, AL = FF
```

In the second example, we tried to generate a negative value. Unsigned values cannot be negative, so the Carry flag signaled an error condition.

# Your turn . . .

For each of the following marked entries, show the values of the destination operand and the Sign, Zero, and Carry flags:

```
mov ax,00FFh
add ax,1                ; AX=0100h  SF=0 ZF=0 CF=0
sub ax,1                ; AX=00FFh  SF=0 ZF=0 CF=0
add al,1                ; AL=00h    SF=0 ZF=1 CF=1
mov bh,6Ch
add bh,95h              ; BH=01h    SF=0 ZF=0 CF=1

mov al,2
sub al,3                ; AL=FFh    SF=1 ZF=0 CF=1
```

# Overflow Flag (OF)

The Overflow flag is set when the signed result of an operation is invalid or out of range.

```
; Example 1
mov al,+127
add al,1                        ; OF = 1,   AL = ??

; Example 2
mov al,7Fh                      ; OF = 1,   AL = 80h
add al,1
```

The two examples are identical at the binary level because 7Fh equals +127. To determine the value of the destination operand, it is often easier to calculate in hexadecimal.

# A Rule of Thumb

- When adding two integers, remember that the Overflow flag is only set when . . .
  - Two positive operands are added and their sum is negative
  - Two negative operands are added and their sum is positive

```
What will be the values of the Overflow flag?
    mov al,80h
    add al,92h                  ; OF =

    mov al,-2
    add al,+127                 ; OF =
```

# Your turn . . .

What will be the values of the Carry and Overflow flags after each operation?

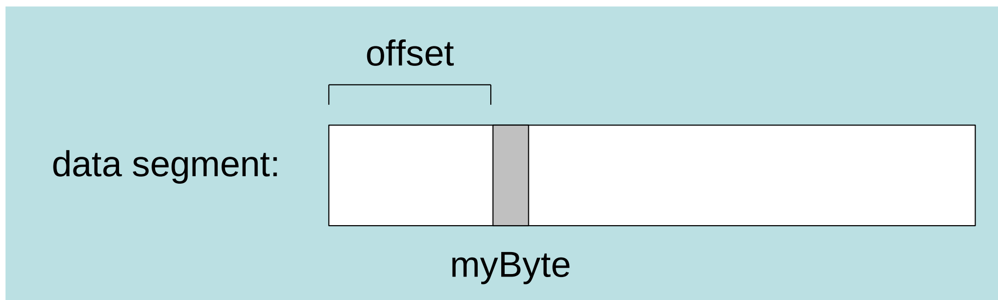
```
mov al,-128
neg al                    ; CF = 0   OF = 1

mov ax,8000h
add ax,2                  ; CF = 0   OF = 0

mov ax,0
sub ax,2                  ; CF = 1   OF = 0

mov al,-5
sub al,+125               ; CF = 0   OF = 1
```

# Data-Related Operators and Directives

- OFFSET Operator
- PTR Operator
- TYPE Operator
- LENGTHOF Operator
- SIZEOF Operator
- LABEL Directive

# OFFSET Operator

- OFFSET returns the distance in bytes, of a label from the beginning of its enclosing segment
  - Protected mode: 32 bits
  - Real mode: 16 bits




The Protected-mode programs we write only have a single segment (we use the flat memory model).

# OFFSET Examples

Let's assume that the data segment begins at 00404000h:

```
.data
bVal BYTE ?
wVal WORD ?
dVal DWORD ?
dVal2 DWORD ?

.code
mov esi,OFFSET bVal        ; ESI = 00404000
mov esi,OFFSET wVal        ; ESI = 00404001
mov esi,OFFSET dVal        ; ESI = 00404003
mov esi,OFFSET dVal2       ; ESI = 00404007
```

# Relating to C/C++

The value returned by OFFSET is a pointer. Compare the following code written for both C++ and assembly language:

```
; C++ version:
char array[1000];
char * p = &array;
```

```
.data
array BYTE 1000 DUP(?)
.code
mov  esi,OFFSET myArray          ; ESI is p
```

# PTR Operator

Overrides the default type of a label (variable). Provides the flexibility to access part of a variable.

```
.data
myDouble DWORD 12345678h
.code
mov ax,myDouble                    ; error – why?

mov ax,WORD PTR myDouble           ; loads 5678h

mov WORD PTR myDouble,4321h        ; saves 4321h
```

To understand how this works, we need to know about little endian ordering of data in memory.

# Little Endian Order

- Little endian order refers to the way Intel stores integers in memory.
- Multi-byte integers are stored in reverse order, with the least significant byte stored at the lowest address
- For example, the doubleword 12345678h would be as:

| byte | offset |
|---|---|
| 78 | 0000 |
| 56 | 0001 |
| 34 | 0002 |
| 12 | 0003 |

When integers are loaded from memory into registers, the bytes are automatically re-reversed into their correct positions.

# PTR Operator Examples

```
.data
myDouble DWORD 12345678h
```

| doubleword | word | byte | offset | |
|---|---|---|---|---|
| 12345678 | 5678 | 78 | 0000 | myDouble |
| | | 56 | 0001 | myDouble + 1 |
| | 1234 | 34 | 0002 | myDouble + 2 |
| | | 12 | 0003 | myDouble + 3 |

```
mov al,BYTE PTR myDouble          ; AL = 78h
mov al,BYTE PTR [myDouble+1]      ; AL = 56h
mov al,BYTE PTR [myDouble+2]      ; AL = 34h
mov ax,WORD PTR [myDouble]        ; AX = 5678h
mov ax,WORD PTR [myDouble+2]      ; AX = 1234h
```

# PTR Operator (cont)

PTR can also be used to combine elements of a smaller data type and move them into a larger operand. The CPU will automatically reverse the bytes.

```
.data
myBytes BYTE 12h,34h,56h,78h

.code
mov ax,WORD PTR [myBytes]          ; AX = 3412h
mov ax,WORD PTR [myBytes+2]        ; AX = 5634h
mov eax,DWORD PTR myBytes          ; EAX = 78563412h
```

# Your turn . . .

Write down the value of each destination operand:

```
.data
varB BYTE 65h,31h,02h,05h
varW WORD 6543h,1202h
varD DWORD 12345678h

.code
mov ax,WORD PTR [varB+2]          ; a. 0502h
mov bl,BYTE PTR varD              ; b. 78h
mov bl,BYTE PTR [varW+2]          ; c. 02h
mov ax,WORD PTR [varD+2]          ; d. 1234h
mov eax,DWORD PTR varW            ; e. 12026543h
```

# TYPE Operator

The TYPE operator returns the size, in bytes, of a single element of a data declaration.

```
.data
var1 BYTE ?
var2 WORD ?
var3 DWORD ?
var4 QWORD ?

.code
mov eax,TYPE var1          ; 1
mov eax,TYPE var2          ; 2
mov eax,TYPE var3          ; 4
mov eax,TYPE var4          ; 8
```

# LENGTHOF Operator

The LENGTHOF operator counts the number of elements in a single data declaration.

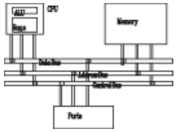
```
.data                                          LENGTHOF
byte1  BYTE 10,20,30                           ; 3
array1 WORD 30 DUP(?),0,0                      ; 32
array2 WORD 5 DUP(3 DUP(?))                    ; 15
array3 DWORD 1,2,3,4                           ; 4
digitStr BYTE "12345678",0                     ; 9

.code
mov ecx,LENGTHOF array1                        ; 32
```

# SIZEOF Operator

The SIZEOF operator returns a value that is equivalent to multiplying LENGTHOF by TYPE.

```
.data                                             SIZEOF
byte1  BYTE 10,20,30                              ; 3
array1 WORD 30 DUP(?),0,0                         ; 64
array2 WORD 5 DUP(3 DUP(?))                       ; 30
array3 DWORD 1,2,3,4                              ; 16
digitStr BYTE "12345678",0                        ; 9

.code
mov ecx,SIZEOF array1                             ; 64
```

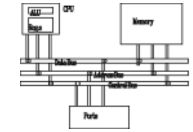
# Spanning Multiple Lines (1 of 2)



A data declaration spans multiple lines if each line (except the last) ends with a comma. The LENGTHOF and SIZEOF operators include all lines belonging to the declaration:

```
.data
array WORD 10,20,
    30,40,
    50,60

.code
mov eax,LENGTHOF array              ; 6
mov ebx,SIZEOF array                ; 12
```

# Spanning Multiple Lines (2 of 2)



In the following example, array identifies only the first WORD declaration. Compare the values returned by LENGTHOF and SIZEOF here to those in the previous slide:

```
.data
array  WORD 10,20
       WORD 30,40
       WORD 50,60

.code
mov eax,LENGTHOF array          ; 2
mov ebx,SIZEOF array            ; 4
```

# LABEL Directive

- Assigns an alternate label name and type to an existing storage location
- LABEL does not allocate any storage of its own
- Removes the need for the PTR operator

```
.data
dwList   LABEL DWORD
wordList LABEL WORD
intList  BYTE 00h,10h,00h,20h
.code
mov eax,dwList              ; 20001000h
mov cx,wordList             ; 1000h
mov dl,intList              ; 00h
```

# Indirect Addressing

- Indirect Operands
- Array Sum Example
- Indexed Operands
- Pointers

# Indirect Operands (1 of 2)

An indirect operand holds the address of a variable, usually an array or string. It can be dereferenced (just like a pointer).

```
.data
val1 BYTE 10h,20h,30h
.code
mov esi,OFFSET val1
mov al,[esi]                    ; dereference ESI (AL = 10h)

inc esi
mov al,[esi]                    ; AL = 20h

inc esi
mov al,[esi]                    ; AL = 30h
```

# Indirect Operands (2 of 2)

Use PTR when the size of a memory operand is ambiguous.

```
.data
myCount WORD 0

.code
mov esi,OFFSET myCount
inc [esi]                    ; error: ambiguous
inc WORD PTR [esi]           ; ok
```

Should PTR be used here?

```
add [esi],20
```

# Array Sum Example

Indirect operands are ideal for traversing an array. Note that the register in brackets must be incremented by a value that matches the array type.

```
.data
arrayW WORD 1000h,2000h,3000h
.code
    mov esi,OFFSET arrayW
    mov ax,[esi]
    add esi,2                   ; or: add esi,TYPE arrayW
    add ax,[esi]
    add esi,2                   ; increment ESI by 2
    add ax,[esi]                ; AX = sum of the array
```

ToDo: Modify this example for an array of doublewords.

# Indexed Operands

An indexed operand adds a constant to a register to generate an effective address. There are two notational forms:

**[*label* + *reg*]**                **_label_[*reg*]**

```
.data
arrayW WORD 1000h,2000h,3000h
.code
   mov esi,0
   mov ax,[arrayW + esi]           ; AX = 1000h
   mov ax,arrayW[esi]              ; alternate format
   add esi,2
   add ax,[arrayW + esi]
   etc.
```

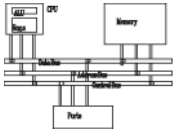
ToDo: Modify this example for an array of doublewords.

# Pointers

You can declare a pointer variable that contains the offset of another variable.

```
.data
arrayW WORD 1000h,2000h,3000h
ptrW DWORD arrayW
.code
    mov esi,ptrW
    mov ax,[esi]              ; AX = 1000h
```

# JMP and LOOP Instructions



- JMP Instruction
- LOOP Instruction
- LOOP Example
- Summing an Integer Array
- Copying a String

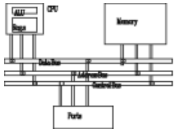
# JMP Instruction

- JMP is an unconditional jump to a label that is usually within the  same procedure.
- Syntax: JMP *target*
- Logic: EIP $\leftarrow$ *target*
- Example:

```
top:
     .
     .
     jmp top
```

A jump outside the current procedure must be to a special type of label called a global label (see Section 5.5.2.3 for details).

# LOOP Instruction



- The LOOP instruction creates a counting loop
- Syntax: LOOP *target*
- Logic:
  - ECX $\leftarrow$ ECX – 1
  - if ECX > 0, jump to *target*
- Implementation:
  - The assembler calculates the distance, in bytes, between the current location and the offset of the target label. It is called the relative offset.
  - The relative offset is added to EIP.

# LOOP Example

The following loop calculates the sum of the integers 5 + 4 + 3 +2 + 1:

```
offset      machine code                  source code
00000000    66 B8 0000                    mov   ax,0
00000004    B9 00000005                   mov   ecx,5

00000009    66 03 C1                  L1: add   ax,cx
0000000C    E2 FB                         loop L1
0000000E
```

When LOOP is assembled, the current location = 0000000E. Looking at the LOOP machine code, we see that –5 (FBh) is added to the current location, causing a jump to location 00000009:

$$00000009 \leftarrow 0000000E + FB$$

# Your turn . . .

If the relative offset is encoded in a single byte,

(a) what is the largest possible backward jump?

(b) what is the largest possible forward jump?

(a) $-128$

(b) $+127$

# Your turn . . .

What will be the final value of AX?

10

```
    mov ax,6
    mov ecx,4
L1:
    inc ax
    loop L1
```

How many times will the loop execute?

4,294,967,296

```
    mov ecx,0
X2:
    inc ax
    loop X2
```

# Nested Loop

If you need to code a loop within a loop, you must save the outer loop counter's ECX value. In the following example, the outer loop executes 100 times, and the inner loop 20 times.

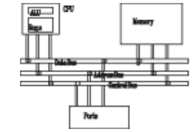
```
.data
count DWORD ?
.code
   mov ecx,100            ; set outer loop count
L1:
   mov count,ecx          ; save outer loop count
   mov ecx,20             ; set inner loop count
L2: .
   .
   loop L2                ; repeat the inner loop
   mov ecx,count          ; restore outer loop count
   loop L1                ; repeat the outer loop
```

# Summing an Integer Array

The following code calculates the sum of an array of 16-bit integers.

```
.data
intarray WORD 100h,200h,300h,400h
.code
    mov edi,OFFSET intarray        ; address of intarray
    mov ecx,LENGTHOF intarray      ; loop counter
    mov ax,0                       ; zero the accumulator
L1:
    add ax,[edi]                   ; add an integer
    add edi,TYPE intarray          ; point to next integer
    loop L1                        ; repeat until ECX = 0
```

# Your turn . . .



What changes would you make to the program on the previous slide if you were summing a doubleword array?

# Copying a String

The following code copies a string from source to target.

```
.data
source  BYTE  "This is the source string",0
target  BYTE  SIZEOF source DUP(0),0

.code
   mov  esi,0                     ; index register
   mov  ecx,SIZEOF source         ; loop counter
L1:
   mov  al,source[esi]            ; get char from source
   mov  target[esi],al            ; store it in the target
   inc  esi                       ; move to next character
   loop L1                        ; repeat for entire string
```

**good use of SIZEOF**

# Your turn . . .

Rewrite the program shown in the previous slide, using indirect addressing rather than indexed addressing.