מערכות חומרה / תוכנה
ותכנות שפת סף

Hardware/Software Systems and Assembly Programming

סמסטר ב' — תשע"ה

חומרה בתוכנית הלימודים בהדסה

- **מערכות ספרתיות**
  - שיטות פורמליות ואבני היסוד של חומרה
- **מערכות חומרה / תוכנה ותכנות שפת סף**
  - כיצד מחשב PC מאורגן ופועל
- **מערכות הפעלה**
  - שירותי תיווך בין חומרת המחשב והמשתמש
- **ארכיטקטורות של מחשבים**
  - ביצועים במחשבים ושיטות מודרניות בתכנון מחשבים
- **תקשורת**
  - חיבור בין מחשבים בתוכנה ובחומרה

כיצד לימוד ארכיטקטורה עוזר למתכנת?

הבנת פעולות החומרה בעיבוד תוכנה עוזרת למתכנת/ת:

1. לכתוב תוכניות מהירות וקוד יעיל יותר
2. להבין את הפעולות של מערכות הפעלה
3. להבין את הפעולות והשיקולים של המהדרים
4. לכתוב hardware drivers וקוד שמיועד למערכות משובצות (כמו משחקים למכשירים הניידים)
5. למצוא מקום טוב בתחום ה- high-tech ולכתוב קוד שמשתלב עם חומרה

דוגמא למסקנה למתכנת מארכיטקטורה

C code for calculating 10!

```
main()
{
     int i,j = 10;
     i = factorial(j);
}

int factorial(n)
     int n;
{
     if (n == 0)
          return 1;
     else
          return  n * factorial(n-1);
}
```

Compiled code runs in 250 nanoseconds on 4 GHz Pentium 4

Efficient assembly language program runs 30% faster

Speed-up important to user

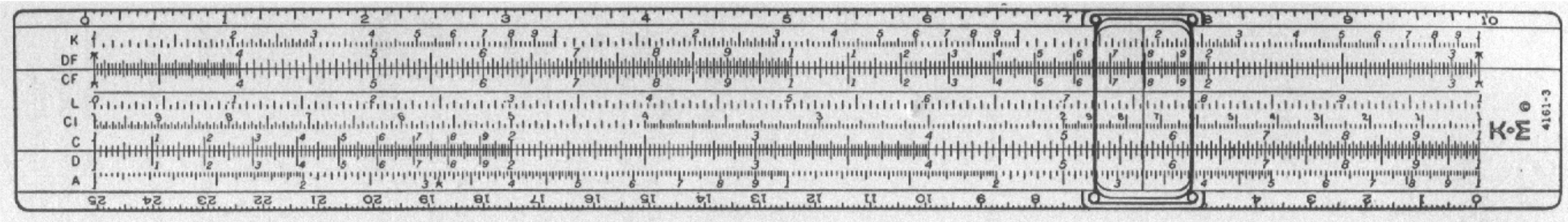
Understanding why important to programmer

Basic Computer Organization

Western Computing Before 1940

Mechanical device for performing arithmetic

- Mechanical Adding Machine
  - Add, Subtract, Multiply, Divide for business records
  - Based on work by Charles Babbage (1821)
- Slide Rule
  - Multiply, Divide, Logarithm, Trigonometry
  - Based on work by Napier, Gunter, Oughtred (1614)





Slide Rule Arithmetic

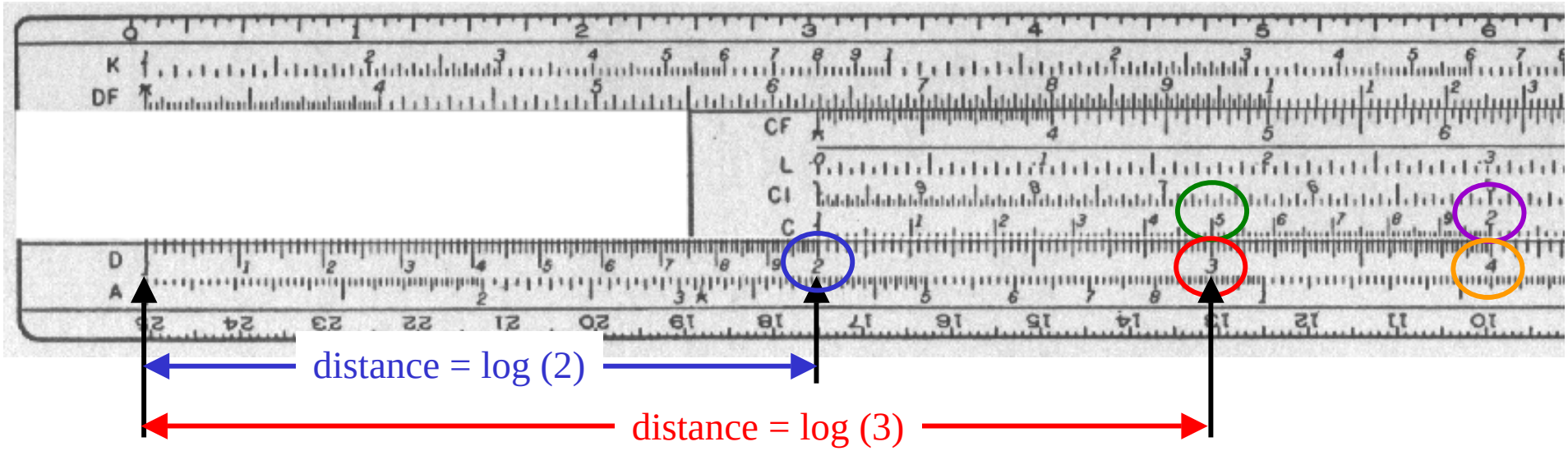
Multiply / Divide

Logarithmic scales C and D

$$\log(xy) = \log(x) + \log(y)$$

Examples: $2 \times 15 = 30$

$2 \times 2 = 4$



Features of Slide Rule Calculating

Analog computation

Based on lengths of scales

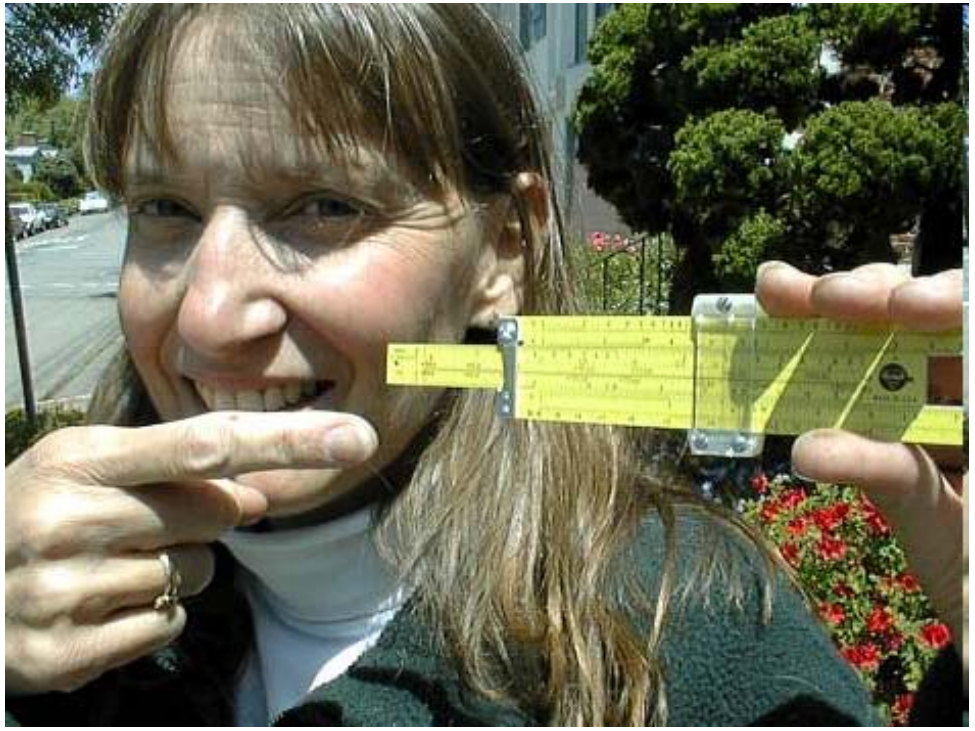
User is operating system

Keeps track of data

Performs each operation

Keeps track of steps

Reads off result



http://arch.ced.berkeley.edu/kap/1998_images/Background/slide%20rule.JPG

End of Slide Rule Calculating

Before 1975

Slide rule was standard calculating instrument for scientists

In 1970

US astronaut flying to moon finished complex calculation by slide rule faster than NASA mainframe in Houston

In mid-1970s

Microprocessor-based calculators and personal computers introduced

In 1980

Slide rules no longer manufactured

Von Neumann's "Computing Instrument"

John von Neumann and others specified (1947) features for an electronic digital computer:

Digital computation in Arithmetic/Logic Unit (ALU)

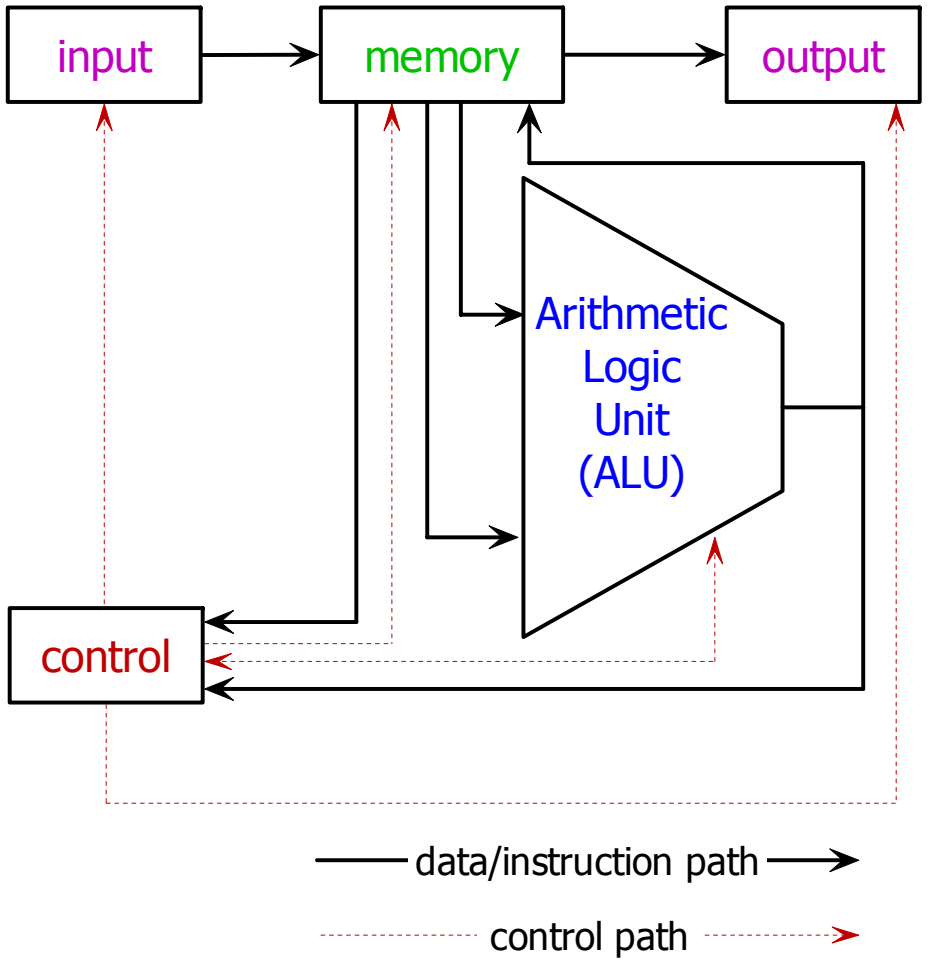
Internal storage of data

Programmable via standard set of instructions

Internal storage of program

Input/Output

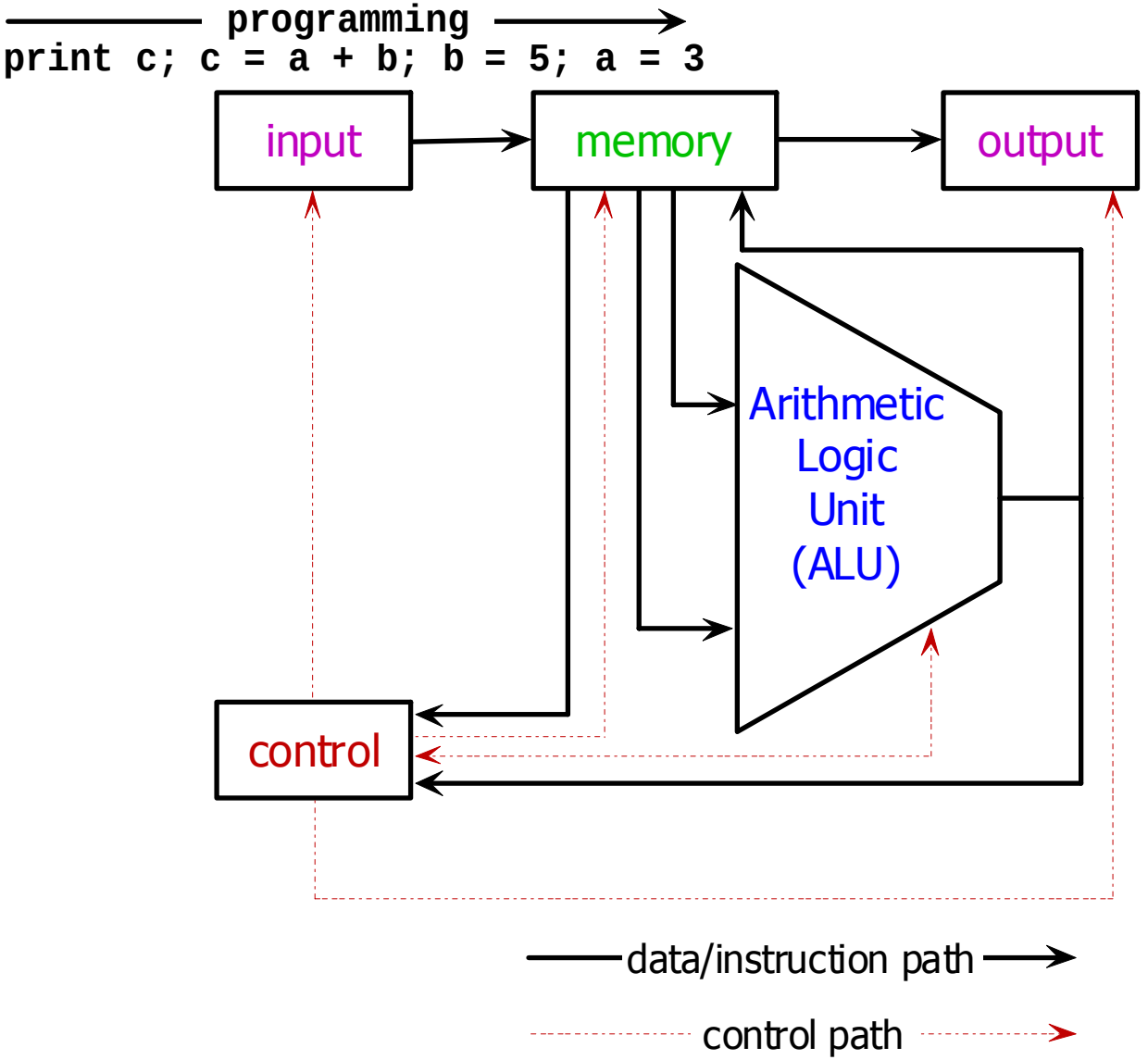
Automatic sequencing of instruction execution



Von Neumann Example — 1

Programming:

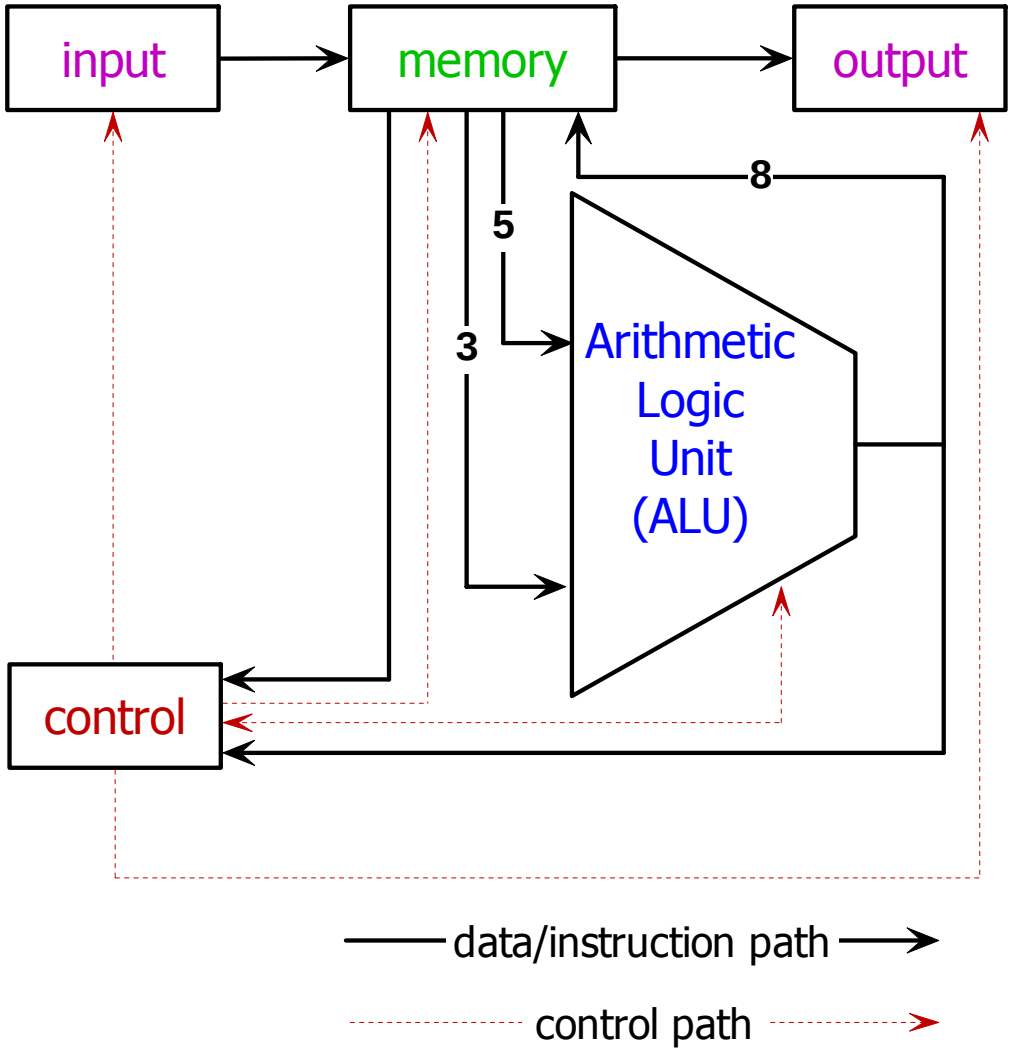
```
a = 3
b = 5
c = a + b
print c
```



Von Neumann Example — 2

Automatic Execution:

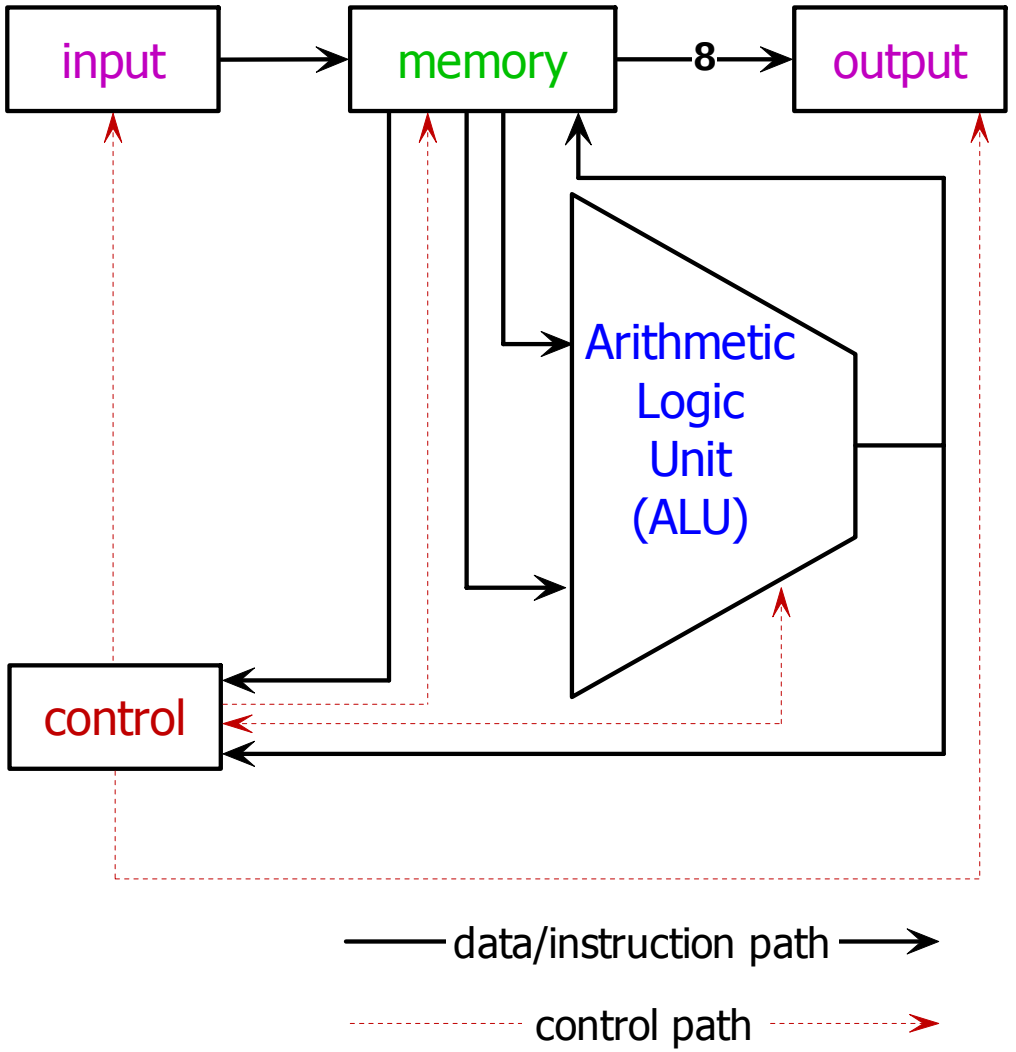
```
ALU ← 3
ALU ← 5
ADD
Memory ← 8
```



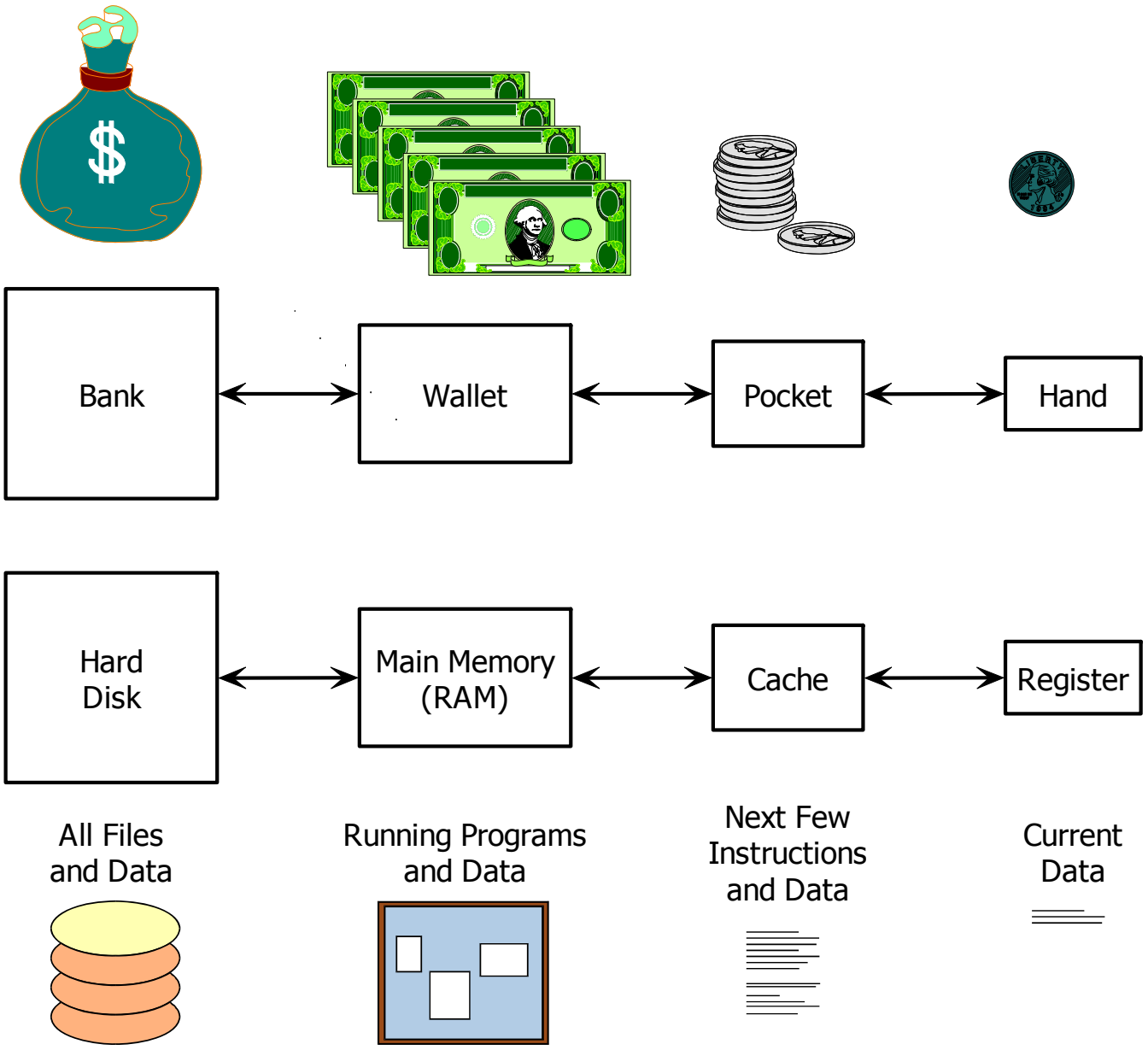
Von Neumann Example — 3

Output:

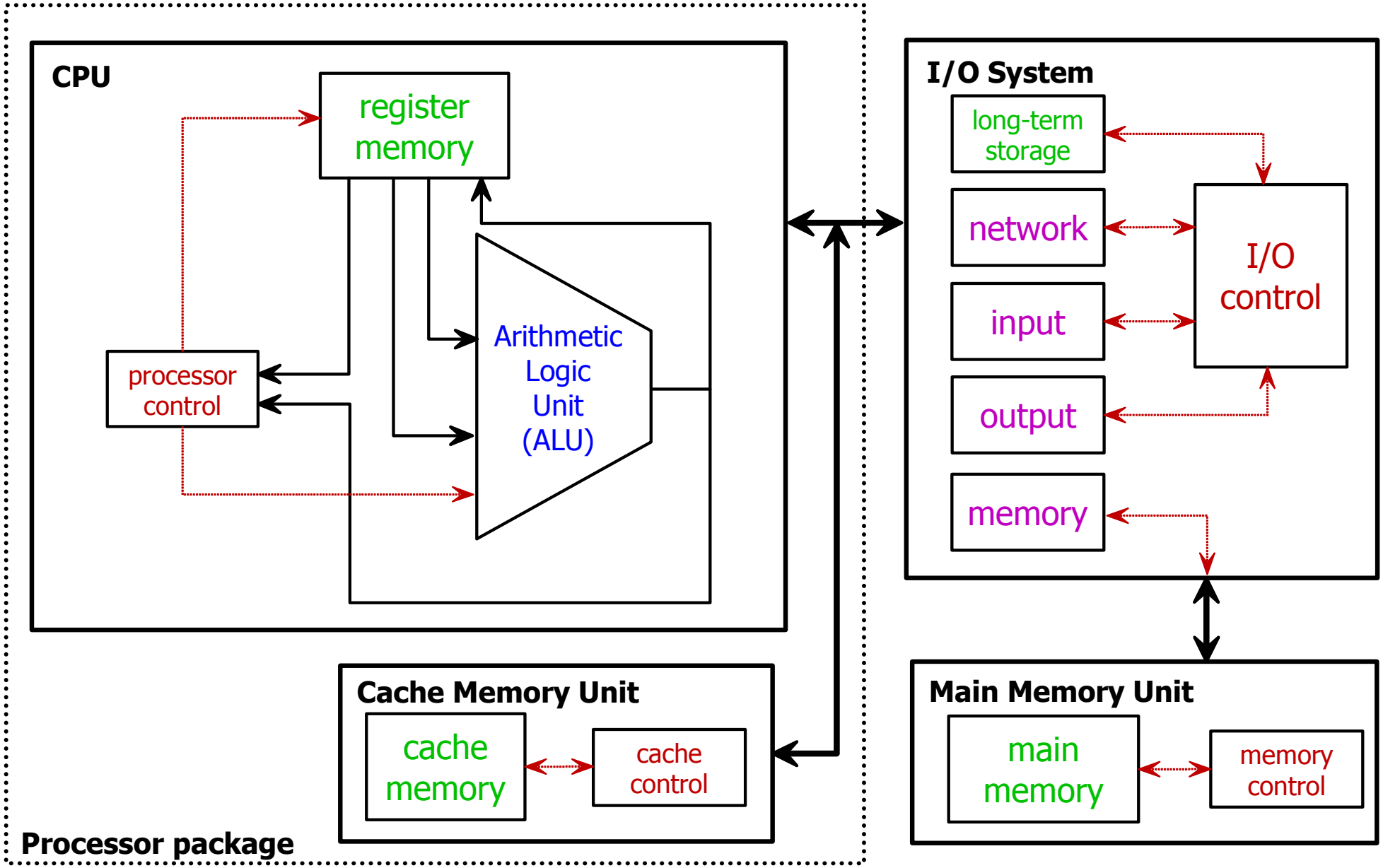
```
Output ← c = 8
```



Memory Hierarchy



Organizing the Pieces — 1



Organizing the Pieces — 2

Physical components of von Neumann machine organized into 3 groups:

Microprocessor – Central Processing Unit (CPU)

- Arithmetic/Logic Unit (ALU)
- Registers – Small and fast memory elements
- Control Unit – Instruction decoding/execution
- Internal cache (optional)

Main Memory Unit (MMU)

- Stores data and program
- Large Random Access Memory (RAM)
- External cache (optional)

Input/Output System (IOS)

- Controls peripheral equipment
- Interface with the user
- Interface with long-term data storage

Basic Types of Digital Systems

Combinational Logic Gates

Output = Boolean function of Inputs

`AND, OR, NOT, XOR,` coders, decoders

Sequential Logic Gates

Output = Boolean function of Inputs and previous Outputs

Registers, Flip-Flops, Shift Registers, Counters, State Machines

Bill Gates

Controls all other digital systems

Physical Components of a Digital Computer

Arithmetic/Logic Unit (ALU)

- Add, Subtract, Multiply, Divide, AND, OR, XOR, Shift, Compare
- Combinations of above
- Built from combinational and sequential logic

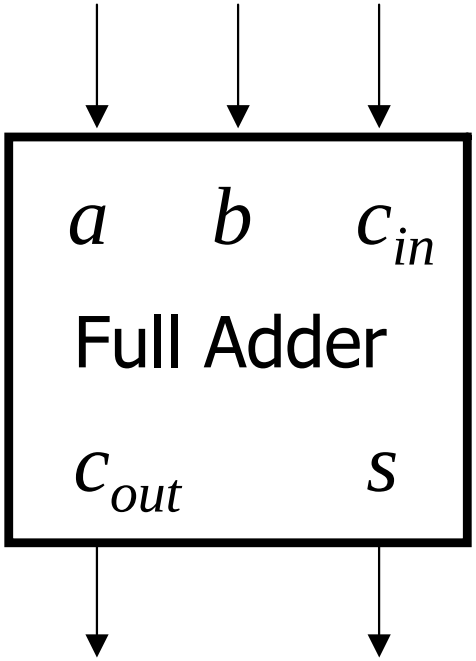
Memory

- Main Memory
  - Random Access Memory (RAM)
  - Internal Registers
  - Read Only Memory (ROM)
- Built from sequential logic devices

Input/Output and Control

- Built from
  - Combinational logic (coders and decoders)
  - Sequential logic (state machines)

ALU Example: Full Adder



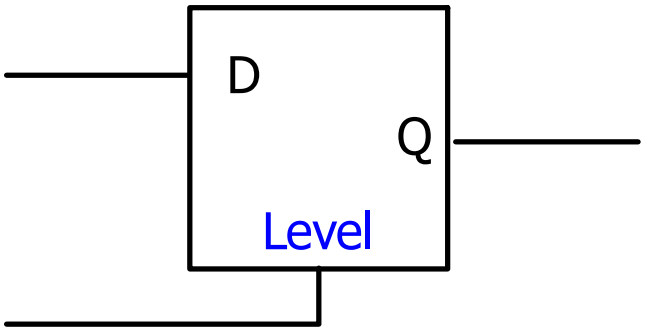
c_{out}	a	b	c_{in}	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$c_{out} = a\ b + a\ c_{in} + b\ c_{in}$$

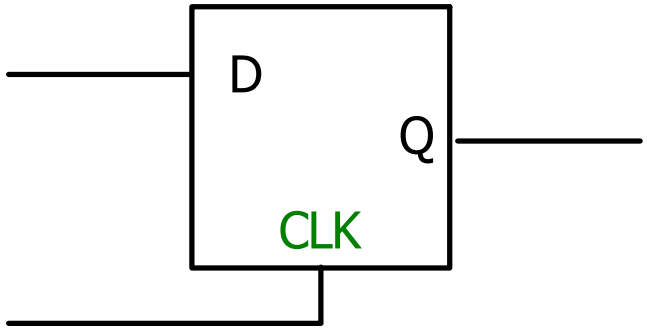
$$s = a \oplus b \oplus c_{in}$$

Memory Example: Latching Register

Level-Sensitive Latch



Edge-Sensitive Latch



Level-Sensitive Latch

When Level = 1, Q = D (a change in D causes a change in Q)

When Level = 0, Q = stored value (changes in D do not effect Q)

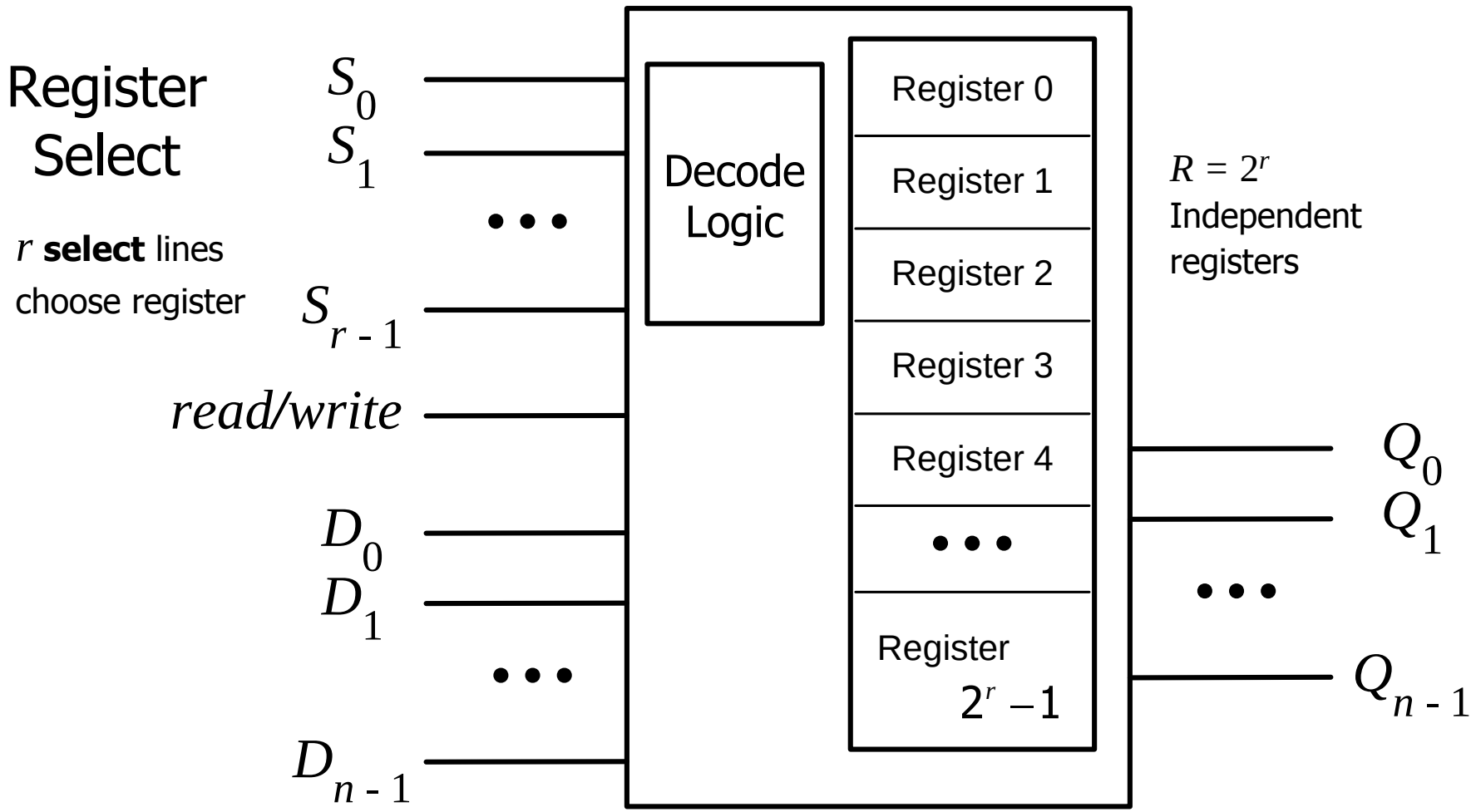
Edge-Sensitive Latch

When CLK makes 0-to-1 transition, latch stores D

Q = stored value (changes in D do not effect Q until next CLK transition)

Registers and Random Access Memory (RAM) built from latches

General Registers



r **select** lines choose register

Register holds *n*-bit number

Read state **— read *n* bits on Q_i**

Write state **— write *n* bits on D_i**

Data Reference — Finding Stored Data

Long term storage

Complex organization depends on hardware design

Usually requires **OPERATING SYSTEM** support

Main memory (RAM)

Random access based on a **PHYSICAL ADDRESS**

Every byte is stored at an address

Cache

Same as main memory

Registers

Random access based on **REGISTER NAMES** in CPU

Information stored in a named register

Data stored is width of standard integer

Physical Memory Organization

Physical memory (main memory — RAM)

- Real device (circuit) to store data
- Stores one **byte** at each physical address

Physical Address = A_{N-1} A_{N-2} … A_1 A_0

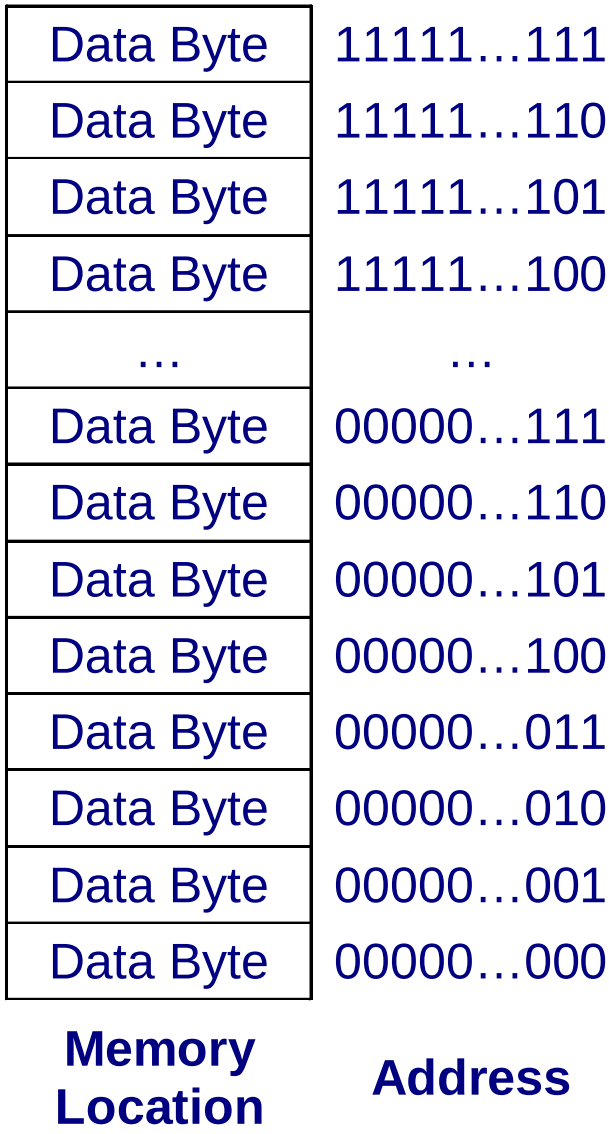
- N-bit **address space**
- 2^N addresses from 0 to (2^N-1)

Flat address CPU

- Programs use physical addresses

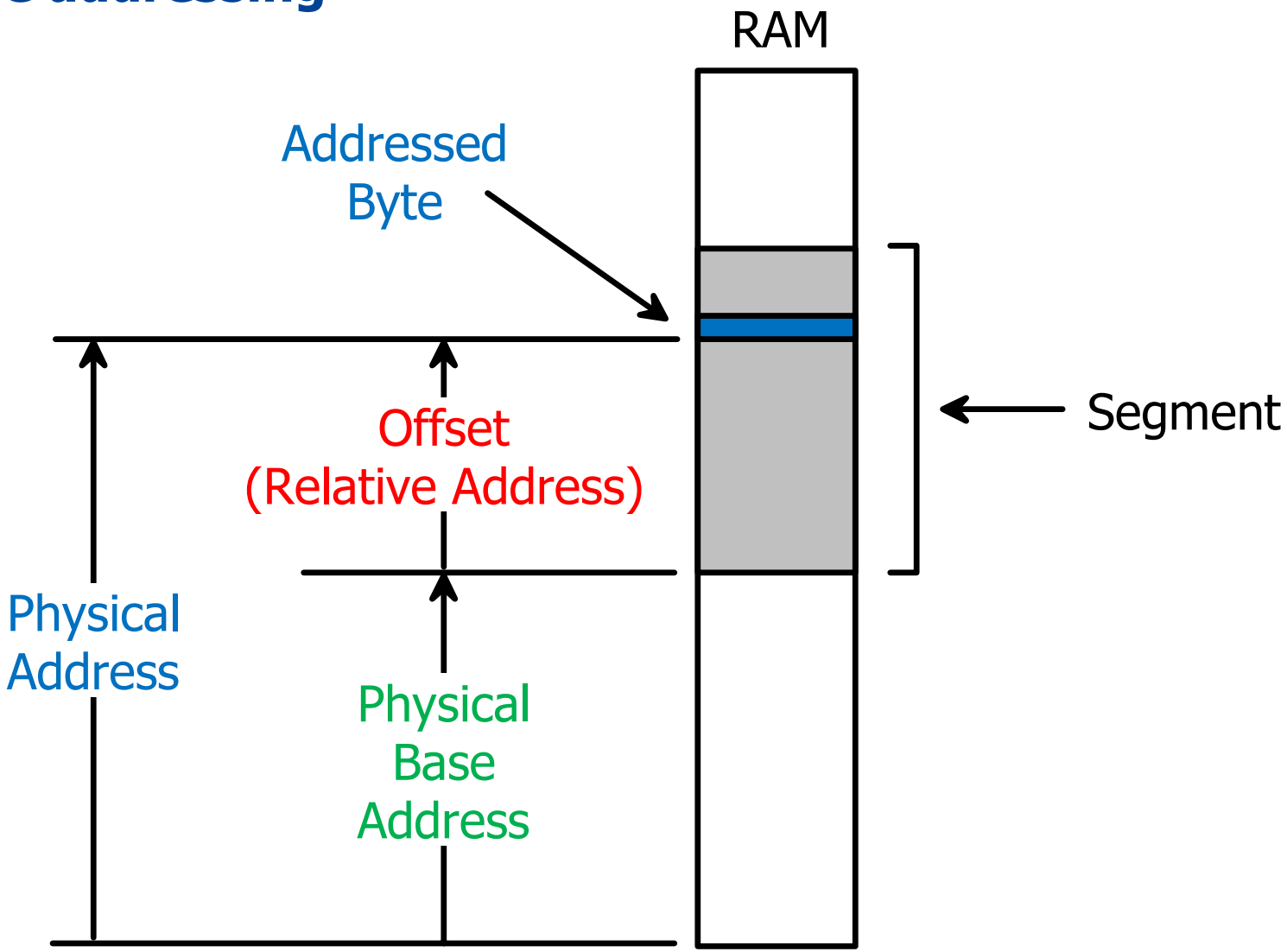
Logical address CPU

- Programs use some other address format
- CPU converts logical address to physical address before RAM access



Segmented Memory Addressing — 1

Relative addressing



Segmented Memory Addressing — 2

Segment

Block of data bytes in memory

Physical base address — physical address of first byte in segment

Offset

Relative address

Location of byte relative to start of segment

Physical address = Physical base address + Offset

Programs use logical address for memory reference

Logical address written as `SEGMENT:OFFSET`

`SEGMENT` and `OFFSET` stored in separate registers

CPU converts **logical address** to **physical address** for RAM access

Register Naming

Registers are part of CPU design

Features determined by CPU designers

General Purpose (GP) registers

Hold data for instructions

Width of data is width of standard integer

Usually similar to one another

Referenced by names or numbers

Standard names: **`R0, R1, … , R15`**

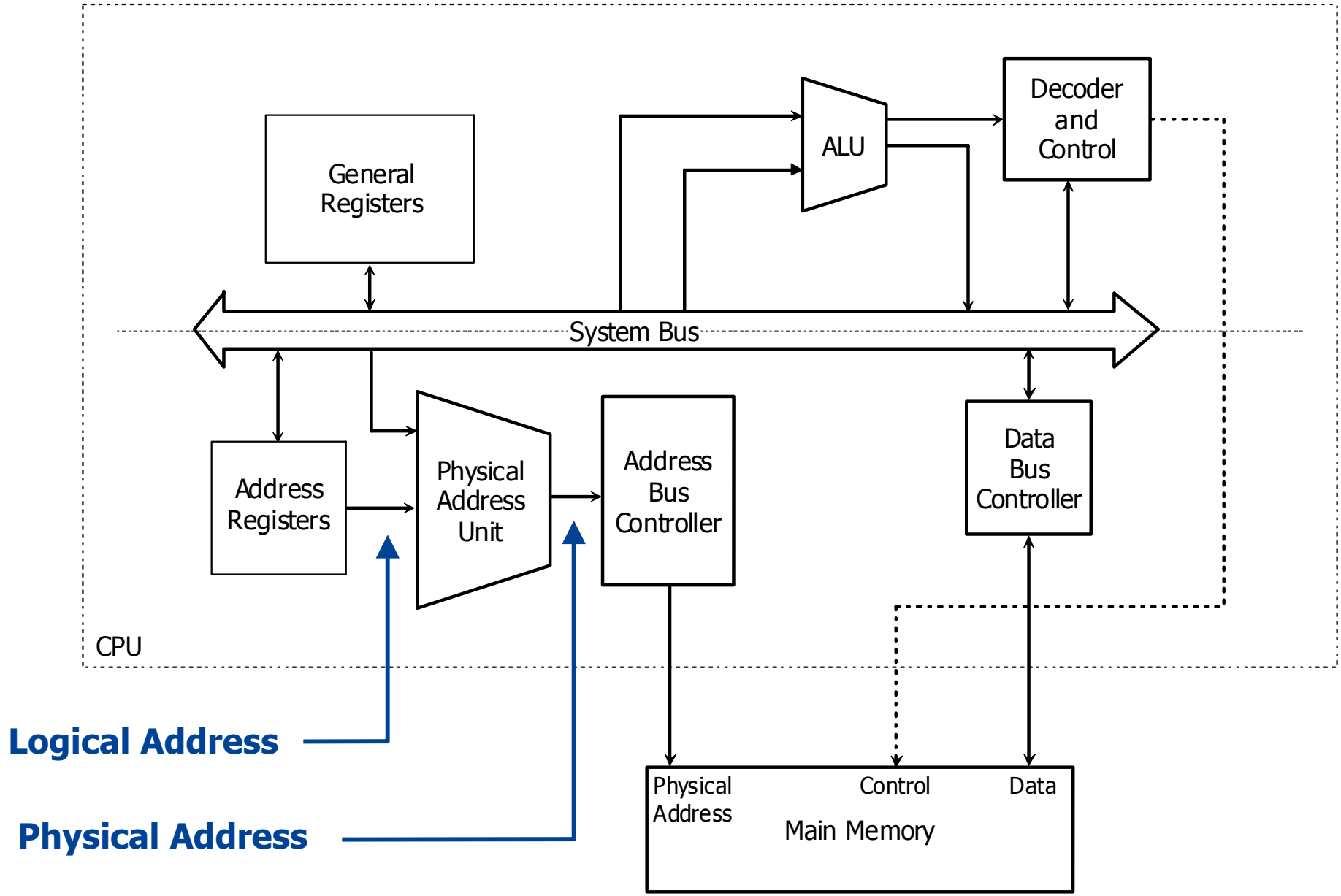
Intel names: **`EAX, EBX, ECX, EDX,`**
`ESI, EDI, EBP, ESP, EIP`

Special Purpose registers

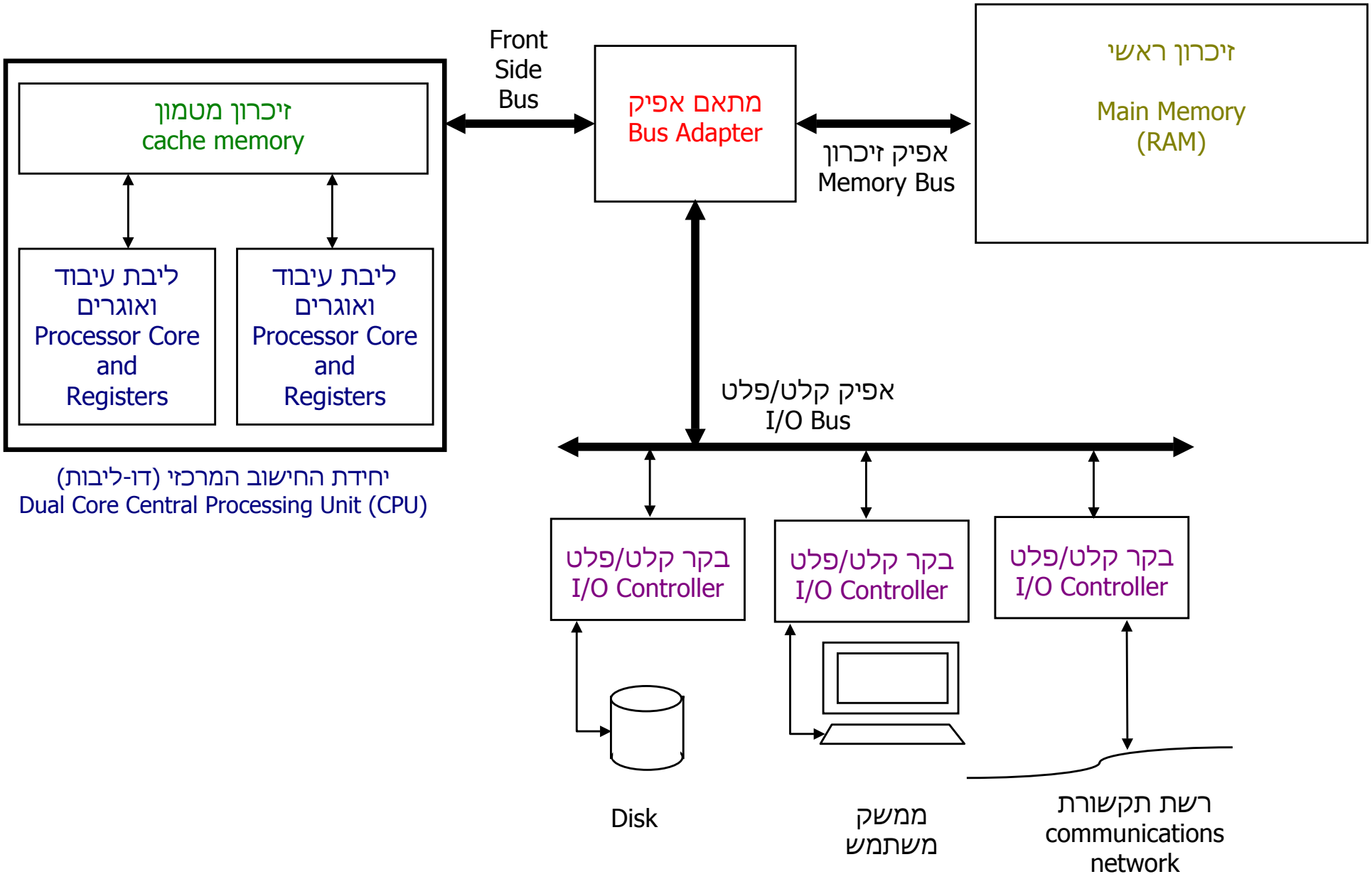
Machine status registers

Operating system registers

Example — 8086 Microprocessor



Simplified PC Organization — 1



Computer Organization — 2

Central Processing Unit (CPU)

- Microprocessor core(s)
- Includes ALU, internal registers, control

Cache Memory

- A copy of a small part of Main Memory
- Allows fast access to important memory data

Main Memory

- Random Access Memory
- Stores data and program

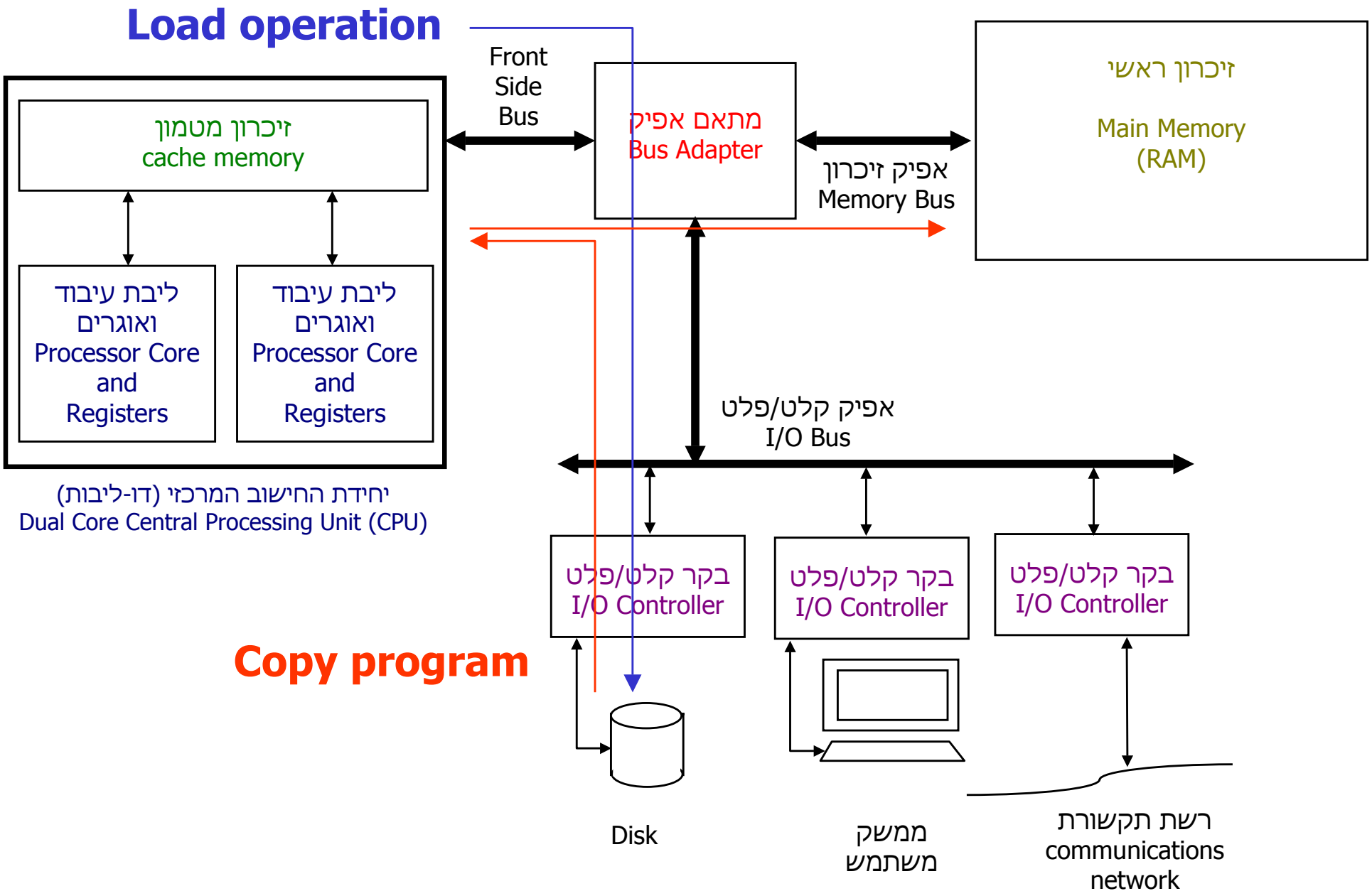
Bus Adapter

- Connects CPU to Memory Bus and I/O Bus

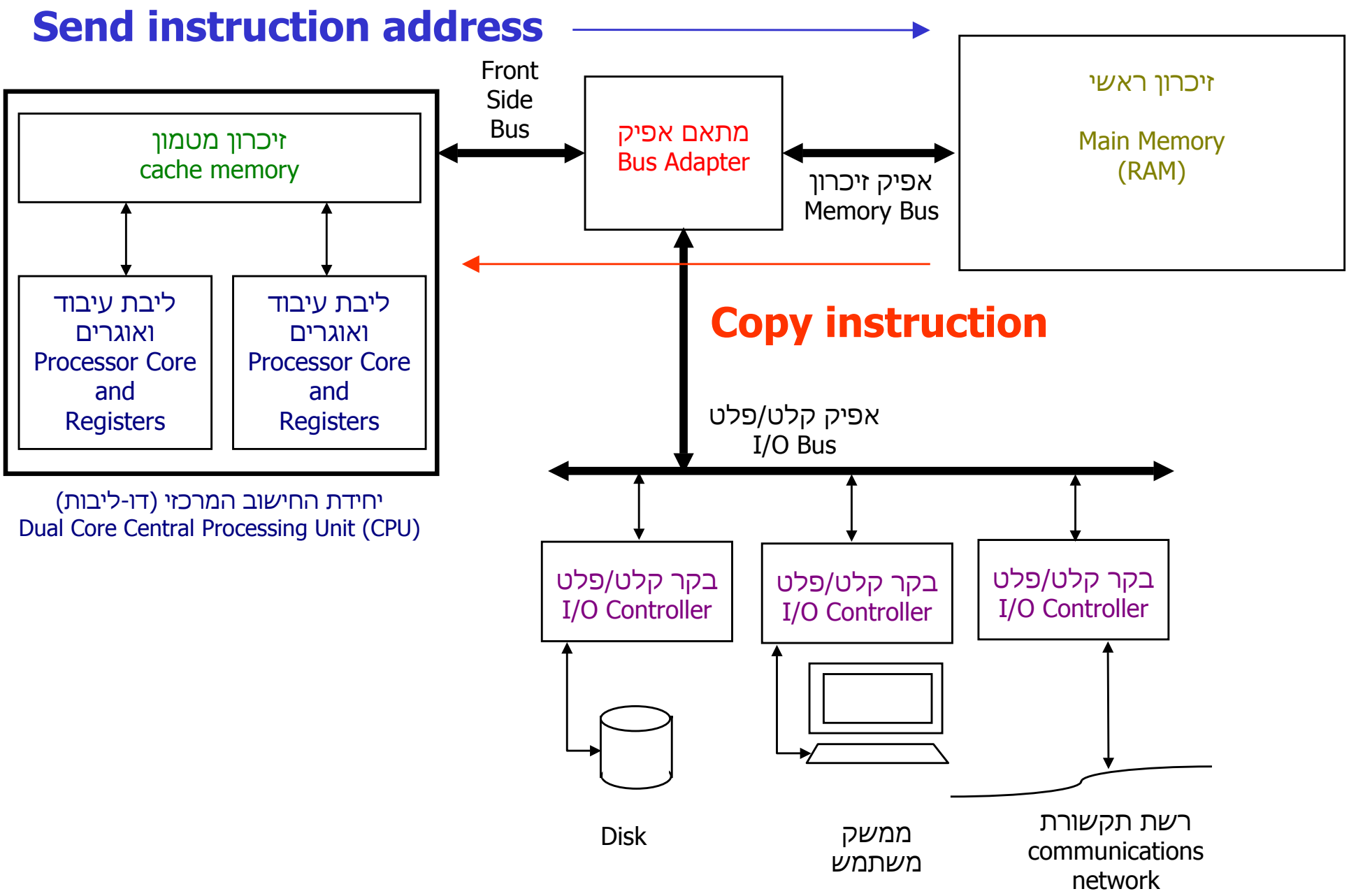
I/O Controller

- Converts I/O bus data to peripheral device instructions

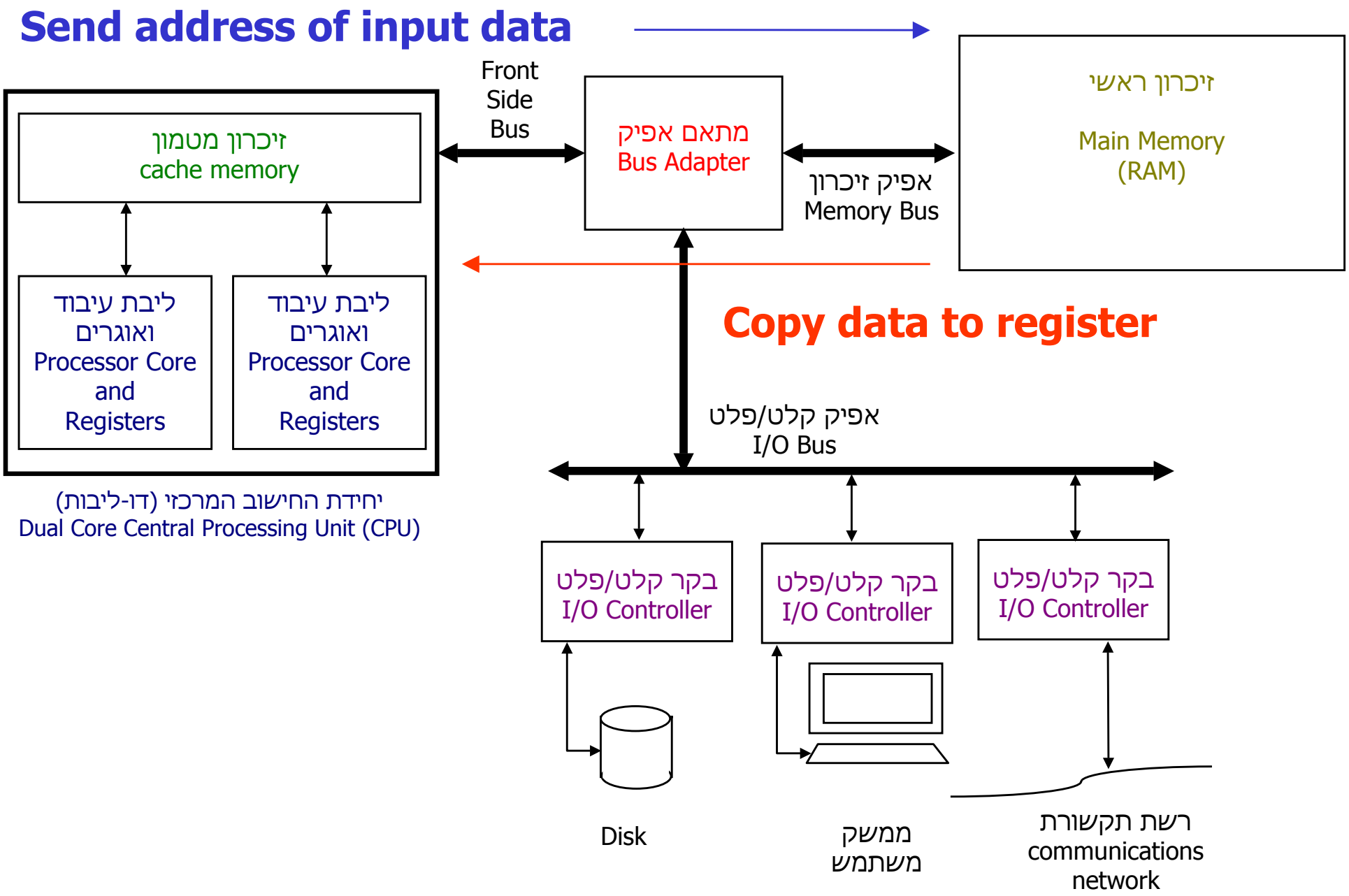
Program Load



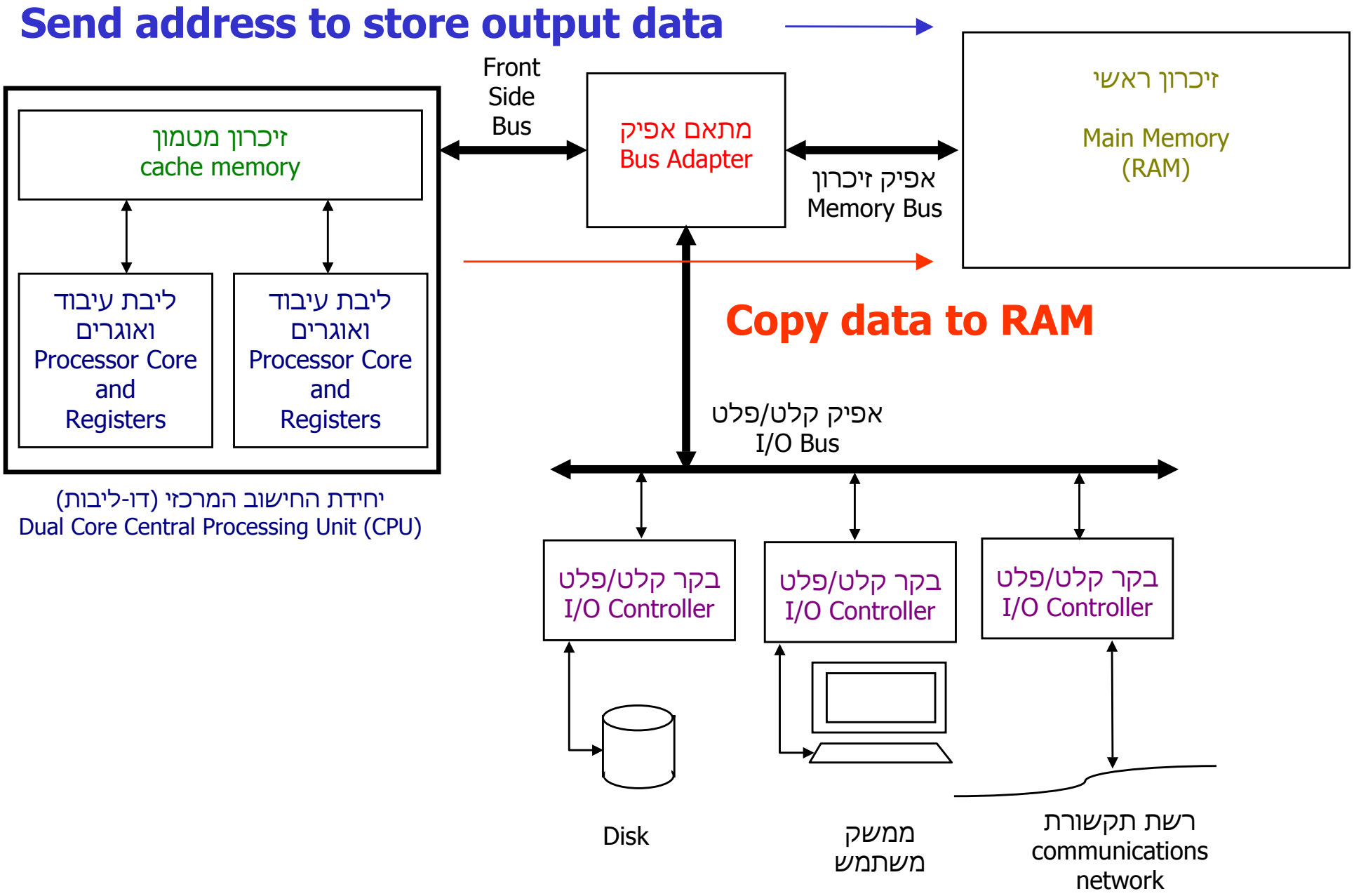
Instruction Fetch



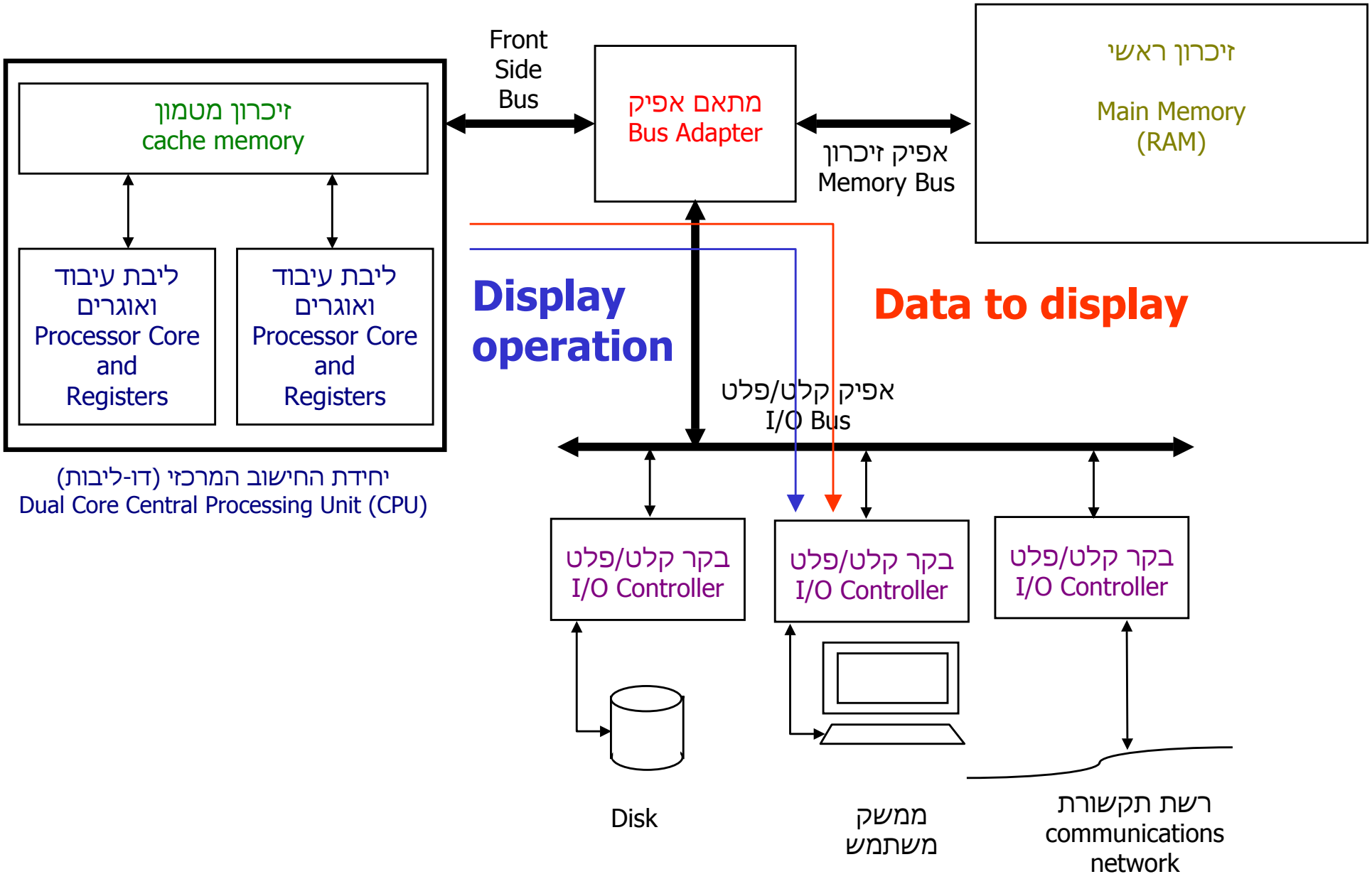
Load Data


Send address of input data
Front Side Bus
מתאם אפיק
Bus Adapter
אפיק זיכרון
Memory Bus
זיכרון ראשי
Main Memory (RAM)
זיכרון מטמון
cache memory
ליבת עיבוד ואוגרים
Processor Core and Registers
ליבת עיבוד ואוגרים
Processor Core and Registers
יחידת החישוב המרכזי (דו-ליבות)
Dual Core Central Processing Unit (CPU)
Copy data to register
אפיק קלט/פלט
I/O Bus
בקר קלט/פלט
I/O Controller
בקר קלט/פלט
I/O Controller
בקר קלט/פלט
I/O Controller
Disk
ממשק משתמש
רשת תקשורת
communications network

Store Result



Display Result



Machine Language Instructions

Instruction

Describes one **OPERATION** and its **OPERANDS**

Operation

Action performed on data

Operand

SOURCE — input data for operation

DESTINATION — output result of operation

Operand description

ADDRESSING MODE

Location of data in register, memory, or instruction

DATA TYPE

Integer, Long, Floating Point, Decimal, String, Constant, etc.

Instruction Set Architecture

General machine instruction is data structure:



Typical machine instruction:

```
ADD destination, source_1, source_2
destination ← source_1 + source_2
```

Language is range of data structure **INSTRUCTION** for:

Operation ∈ {legal **actions**}

Operand ∈ {legal **Addressing Modes**}

Typical Operations

Data transfer

Load (r ← m), store (m ← r), move (r/m ← r/m), convert data types

Arithmetic/Logical (ALU)

Integer arithmetic (+ − × ÷ compare shift) and logical (AND, OR, NOR, XOR)

Decimal

Integer arithmetic on decimal numbers

Floating point (FPU)

Floating point arithmetic (+ − × ÷ sqrt trig exp …)

String

String move, string compare, string search

Control

Conditional and unconditional branch, call/return, trap

Operating System

System calls, memory management instructions

Multimedia

Pixel operations, compression/decompression operations

Addressing Modes for Operands

Immediate

Constant = IMM = numerical value coded into instruction

Register operands

register name = fast storage location in CPU

REGS[register name] = value stored in register

REGS[EBX] = value stored in register **EBX = 00001230**

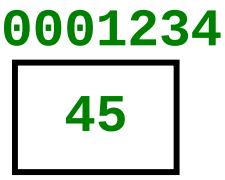


Memory operands

address = slow storage location in main memory (RAM)

MEM[address] = value stored in memory = ***(address)**

MEM[00001234] = value stored at address **00001234 = 45**



Effective Address (EA) — pointer arithmetic

REGS[EBX] ← &(variable) = 00001230

MEM[REGS[EBX]+4] = *(&(variable)+4)

= *(REGS[EBX]+4)

= *(1230+4) = *(1234) = 45

Variables, Pointers, Memory, Registers

Variable allocation

```
int a
```

```
&a = address_a
```

Allocation assigns pointer to some address in RAM

Address depends on program structure and OS

Assignment

```
a = 1
```

```
[address_a] = *(address_a) ← 1
```

Stores value **1** into RAM at address allocated to variable **a**

Software Hierarchies

Instruction Layers	**High-Level Language**
	Machine Language

Programming Layers	**Application Programs**
	Operating System
	Machine Language

Machine Language

Machine Language = processor instruction set

Elementary operations for **CPU**

CPU reads instructions as **binary codes**

Assembly Language

User-friendly form of machine language

Each binary code is represented in words

`INC EBX` ≡ `01000011` ≡ `43` ≡ `REGS[EBX]` ← `REGS[EBX]+1`

Instruction Set Architecture (ISA)

Considerations for choosing a particular set of instructions

"Family" of processors using same instruction set

Pentium, P-II, P-III, P-4, Centrino, Core, i5, i7, Atom

High-Level Language

Powerful instructions and syntax

- Control blocks
- Function calls
- Data structures
- Objects

High-level program compiled to machine code

- 1 line of high-level code compiled to many lines of machine code
- Compiler efficiency as important as good coding

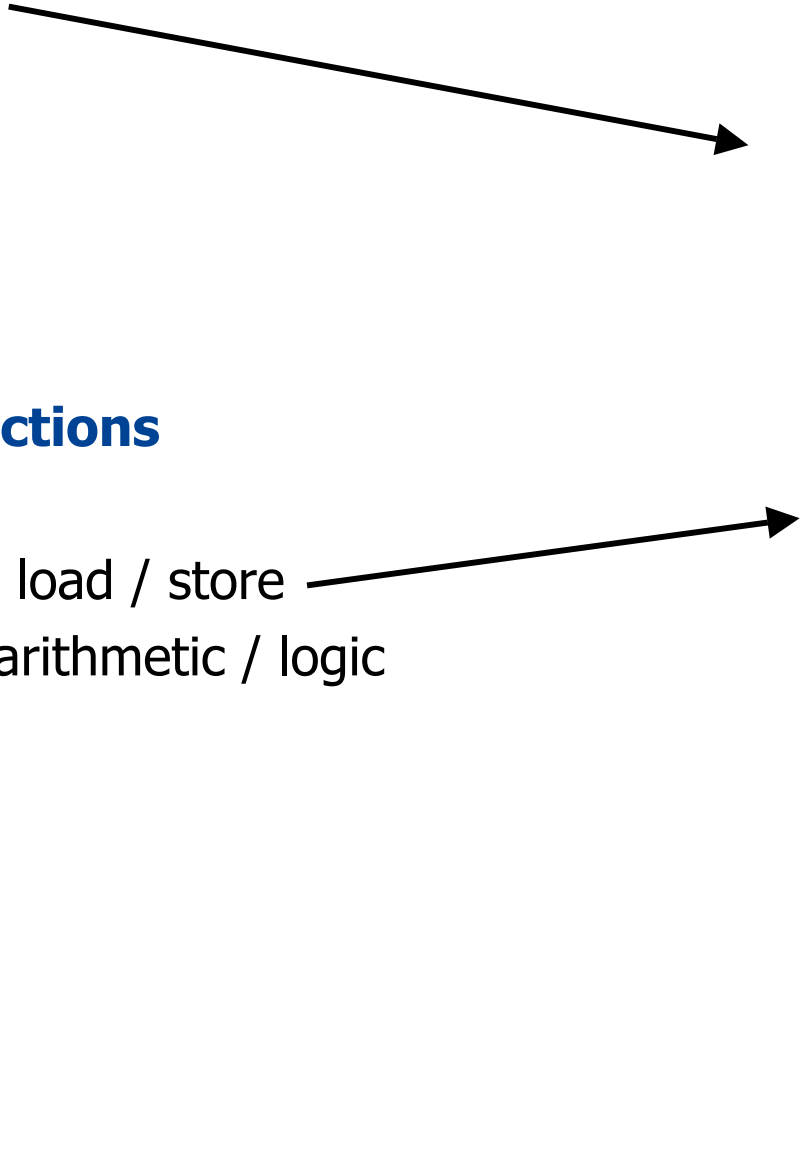
Program Loaded into Memory

Program data

- Constants
- Variables
- Tables
- Structures

Procedures / functions

- Control blocks
- Data access — load / store
- Calculation — arithmetic / logic

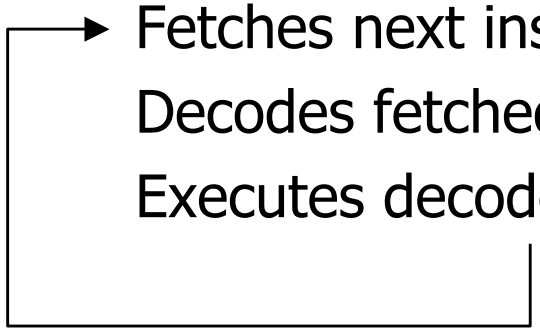


Running Machine Language Program

Program list

Instruction 1
Instruction 2
Instruction 3
Instruction 4
Instruction 5
Instruction 6
…

CPU



Instructions in RAM

Instruction 5	byte	A+11
	byte	A+10
Instruction 4	byte	A+9
	byte	A+8
Instruction 3	byte	A+7
Instruction 2	byte	A+6
	byte	A+5
	byte	A+4
Instruction 1	byte	A+3
	byte	A+2
	byte	A+1
	byte	A

Address

Programming Layers

Machine Language

- CPU can only run machine language code
- CPU switches between application code and operating system code

Operating System

- Stores, loads, and runs executables
- Organizes hardware resources and permits sharing
- Organizes user access
- OS code
  - Mostly compiled from high-level code
  - Parts written in machine language

Application Programs

- Individual task-oriented user programs

Assembler Programs

Programmer codes

Labels, mnemonics, operands, and comments

Assembler does all routine work

Removes comments and mnemonics

Provides HEX code translations

Handles line addresses

Resolves internal references

Output called Object Code

May contain **UNRESOLVED EXTERNAL REFERENCES**

Branches to labels in external modules

Linker produces final executable code

Resolves external references

Types of General Purpose Computer

General Purpose Computer

- Based on von Neumann architecture
- Operating System and I/O System permit users to write, store and run Application Programs

Workstation

- Optimized for single user running input-intensive software for graphical presentation

File Server

- Optimized for multiple users running output-intensive networking software

Mainframe

- Optimized for multiple users running input/output-intensive networking software (storage and transaction)
- High availability, reliability, scaling, security
- May support multiple OS

Enhancements for High Power Computers

	$1000 PC	$10,000 Workstation
CPU Architecture	4 core (Pentium type) 64-bit CISC processor	Fast (expensive) chip technology 64-bit RISC multiprocessor system
Input/Output System	Single PCI I/O controller integrated with main board	Multiple I/O controllers for multiple simultaneous data flows High speed I/O channels
Operating System	Windows/Linux	High efficiency specialized versions of Unix-inspired OS
Specialized Hardware	Single graphic processor Memory management on CPU and PCI	Multiple graphics processors Memory management processors Communications processors Special processors for Java, encryption, error detection

CISC (Complex Instruction Set Computer) — old style (pre-1990) machine language

RISC (Reduced Instruction Set Computer) — new style (post-1990) machine language

PCI (Peripheral Component Interface) — I/O controller on most PC systems