ARCHITETTURA DEGLI ELABORATORI

A.A. 2020-2021

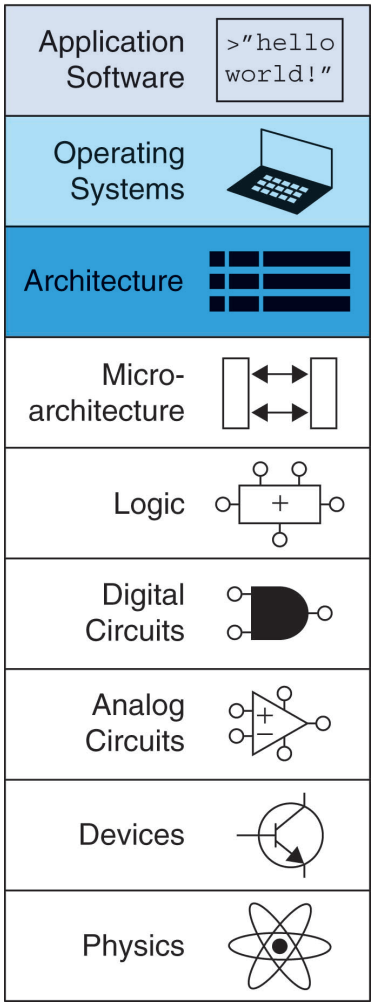
Università di Napoli Federico II
Corso di Laurea in Informatica

Docenti
Proff. Luigi Sauro gruppo 1 (A-G)
Silvia Rossi gruppo 2 (H-Z)

ARCHITETTURA ARM

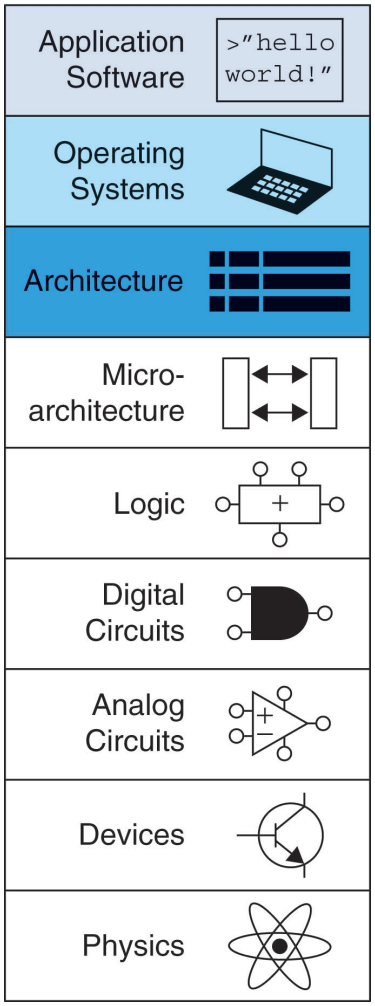
Chapter 6 :: Topics

- **Introduction**
- **Assembly Language**
- **Machine Language**
- **Programming**
- **Addressing Modes**



Introduction

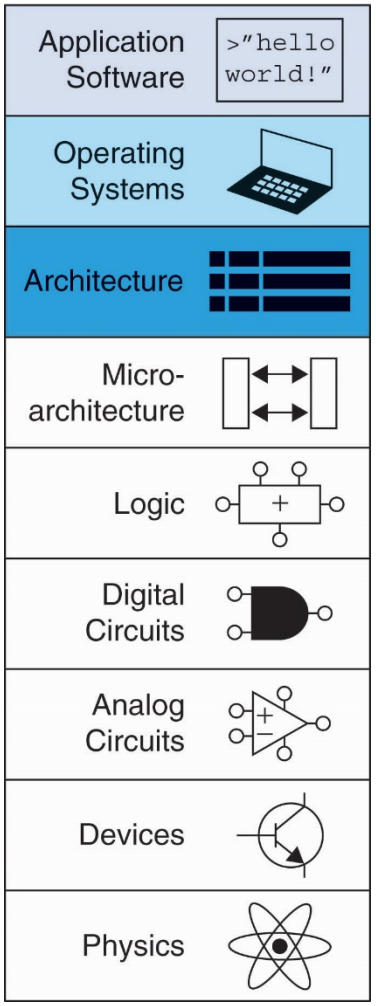
- **Jumping up a few levels of abstraction**
  - **Architecture:** programmer's view of computer
    - Defined by **instructions** & **operand locations**
  - **Microarchitecture:** how to implement an architecture in hardware (covered in Chapter 7)



Introduction

Architettura: descrizione operazionale di computer

- Come un programmatore *vede* a basso livello un computer
- Definisce un **insieme di istruzioni** e di registri che fungono da **operandi** per tali istruzioni



Instruzioni

- Istruzioni ARM
  - **Assembly language:** formato human-readable
  - **Machine language:** formato computer-readable

ARM Architecture

- Developed in the 1980's by Advanced RISC Machines – now called ARM Holdings
- Nearly 10 billion ARM processors sold/year
- Almost all cell phones and tablets have multiple ARM processors
- Over 75% of humans use products with an ARM processor
- Used in servers, cameras, robots, cars, pinball machines, etc.

Architecture Design Principles

Underlying design principles, as articulated by Hennessy and Patterson:

1. **La regolarità favorisce la semplicità**
2. **Rendere veloci le cose frequenti**
3. **Più piccolo è più veloce**
4. **Un buon Progetto richiede buoni compromessi**

Instruction: Addition

C Code

```
a = b + c;
```

ARM Assembly Code

```
ADD a, b, c
```

- **ADD:** mnemonic – indicates operation to perform
- **b, c:** source operands
- **a:** destination operand

Instruction: Subtraction

Similar to addition - only mnemonic changes

C Code	ARM assembly code
`a = b - c;`	`SUB a, b, c`

- **SUB:** mnemonic
- **b, c:** source operands
- **a:** destination operand

Design Principle 1

Regularity supports design simplicity

- Consistent instruction format
- Same number of operands (two sources and one destination)
- Ease of encoding and handling in hardware

Multiple Instructions

More complex code handled by multiple ARM instructions

C Code

```
a = b + c - d;
```

ARM assembly code

```
ADD t, b, c  ; t = b + c
SUB a, t, d  ; a = t - d
```

Design Principle 2

Make the common case fast

- ARM includes only simple, commonly used instructions
- Hardware to decode and execute instructions kept simple, small, and fast
- More complex instructions (that are less common) performed using multiple simple instructions

Design Principle 2

Make the common case fast

- ARM is a **Reduced Instruction Set Computer (RISC)**, with a small number of simple instructions
- Other architectures, such as Intel's x86, are **Complex Instruction Set Computers (CISC)**

Operand Location

Physical location in computer

- Registers
- Constants (also called *immediates*)
- Memory

Operands: Registers

- ARM has 16 registers
- Registers are faster than memory
- Each register is 32 bits
- ARM is called a “32-bit architecture” because it operates on 32-bit data

Design Principle 3

Smaller is Faster

- ARM includes only a small number of registers

ARM Register Set

- ARM ha 16 registri a 32 bit (R0… R15) che sono fisicamente equivalenti fra loro, ma dal punto di vista logico sono usati con scopi specifici

Name	Use
R0	Argument / return value / temporary variable
R1-R3	Argument / temporary variables
R4-R11	Saved variables
R12	Temporary variable
R13 (SP)	Stack Pointer
R14 (LR)	Link Register
R15 (PC)	Program Counter

Operands: Registers

- **Registers:**
  - R before number, all capitals
  - Example: “R0” or “register zero” or “register R0”

Operands: Registers

- **Registers used for specific purposes:**
  - **Saved registers:** R4-R11 hold variables
  - **Temporary registers:** R0-R3 and R12, hold intermediate values
  - Discuss others later

Instructions with Registers

Revisit ADD instruction

C Code	ARM Assembly Code
	`; R0 = a, R1 = b, R2 = c`
`a = b + c`	`ADD R0, R1, R2`

Operands: Constants\Immediates

- Many instructions can use constants or *immediate* operands
- For example: ADD and SUB
- value is *immediate*ly available from instruction

C Code

```
a = a + 4;
b = a - 12;
```

ARM Assembly Code

```
; R0 = a, R1 = b
ADD R0, R0, #4
SUB R1, R0, #12
```

Generazione di costanti

E’ possibile definire costanti con l’istruzione `MOV`:

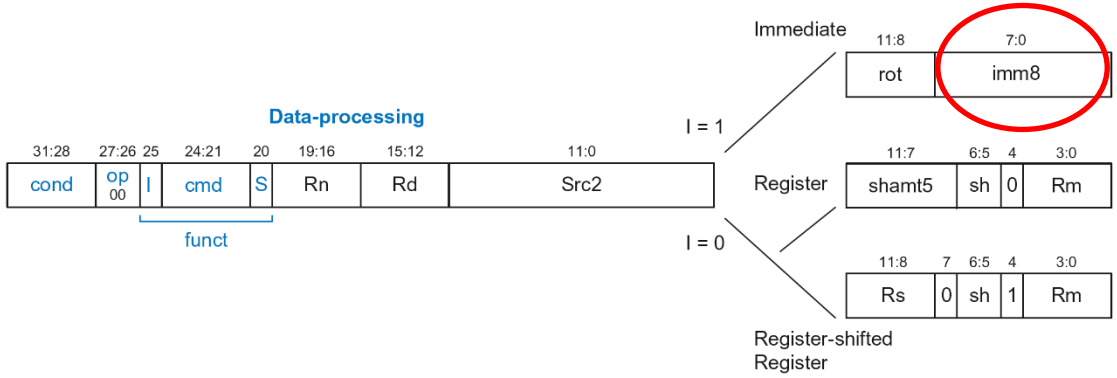
C Code

```
//int: 32-bit signed word
int a = 23;
int b = 0x45;
```

ARM Assembly Code

```
; R0 = a, R1 = b
MOV R0, #23
MOV R1, #0x45
```

Le costanti così generate hanno una precisione massima di < 8 bits



Generazione di costanti

E' possibile definire costanti con l'istruzione `MOV`:

C Code

```
//int: 32-bit signed word
int a = 23;
int b = 0x45;
```

ARM Assembly Code

```
; R0 = a, R1 = b
MOV R0, #23
MOV R1, #0x45
```

Nota: `MOV` può anche essere usato per spostare il contenuto di un registro in un altro registro:

```
MOV R7, R9
```

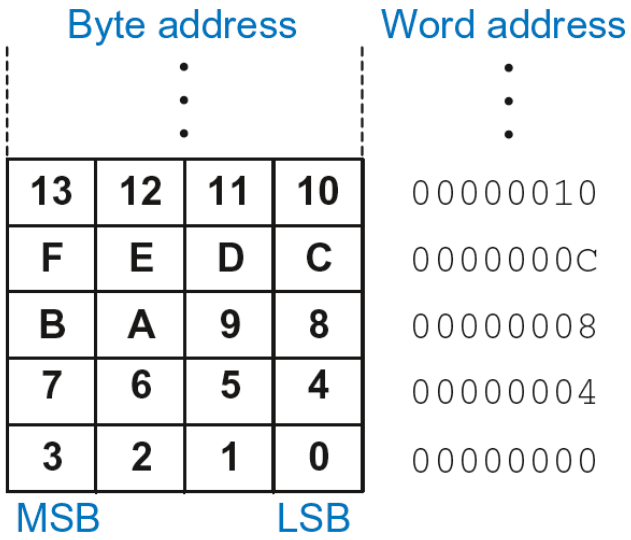
Operands: Memory

- Too much data to fit in only 16 registers
- Store more data in memory
- Memory is large, but slow
- Commonly used variables still kept in registers

- ARM
  - 32 bit per gli indirizzi
  - 32 bit per le parole

Byte-Addressable Memory

- Each data **byte** has unique address
- 32-bit word = 4 bytes, so word address increments by 4



Reading Memory

- Memory read called ***load***
- **Mnemonic:** *load register* (`LDR`)
- **Format:**

```
LDR R0, [R1, #12]
```

Reading Memory

- Memory read called ***load***
- **Mnemonic:** *load register* (`LDR`)
- **Format:**

```
LDR R0, [R1, #12]
```

Address calculation:

- add *base address* (R1) to the *offset* (12)
- address = (R1 + 12)

Result:

- R0 holds the data at memory address (R1 + 12)

Reading Memory

- Memory read called ***load***
- **Mnemonic:** *load register* (`LDR`)
- **Format:**

```
LDR R0, [R1, #12]
```

Address calculation:

- add *base address* (R1) to the *offset* (12)
- address = (R1 + 12)

Result:

- R0 holds the data at memory address (R1 + 12)

Any register may be used as base address

Reading Memory

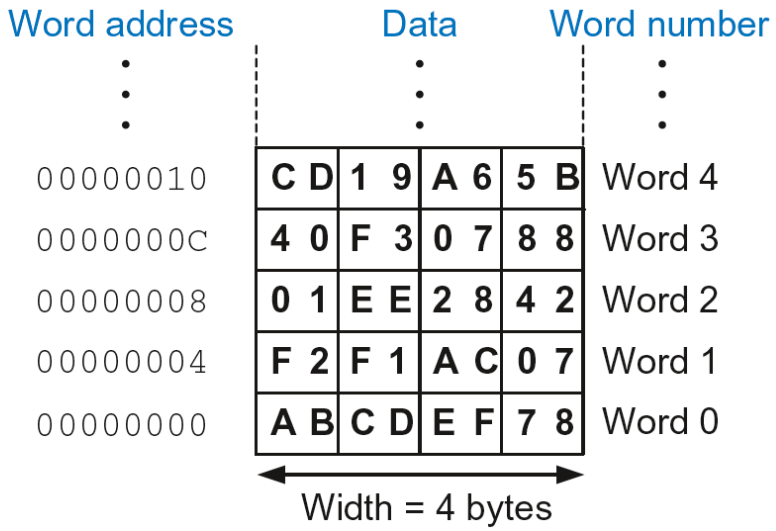
- **Example:** Read a word of data at memory address 8 into R3

Reading Memory

- **Example:** Read a word of data at memory address 8 into R3
  - Address = (R2 + 8) = 8
  - R3 = 0x01EE2842 after load

ARM Assembly Code

```
MOV R2, #0
LDR R3, [R2, #8]
```



Writing Memory

- Memory write are called ***stores***
- **Mnemonic:** *store register* (`STR`)

Writing Memory

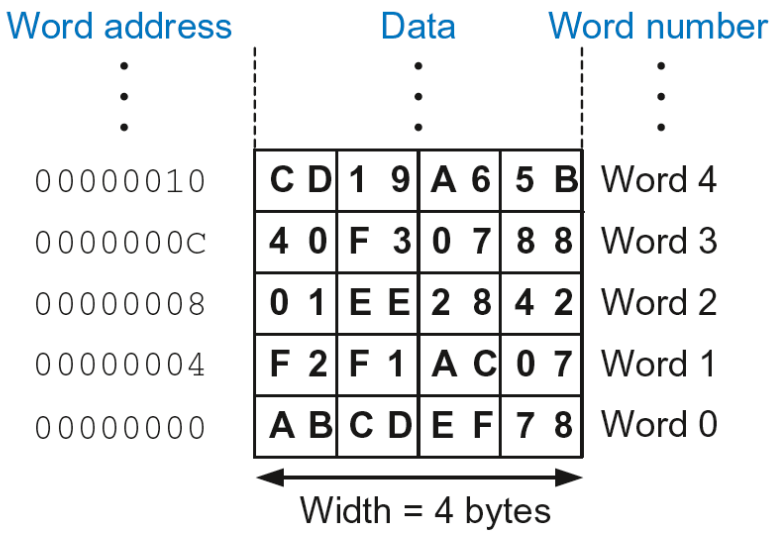
- **Example:** Store the value held in R7 into memory word 21.

Writing Memory

- **Example:** Store the value held in R7 into memory word 21.
- Memory address = 4 x 21 = 84 = 0x54

ARM assembly code

```
MOV R5, #0
STR R7, [R5, #0x54]
```



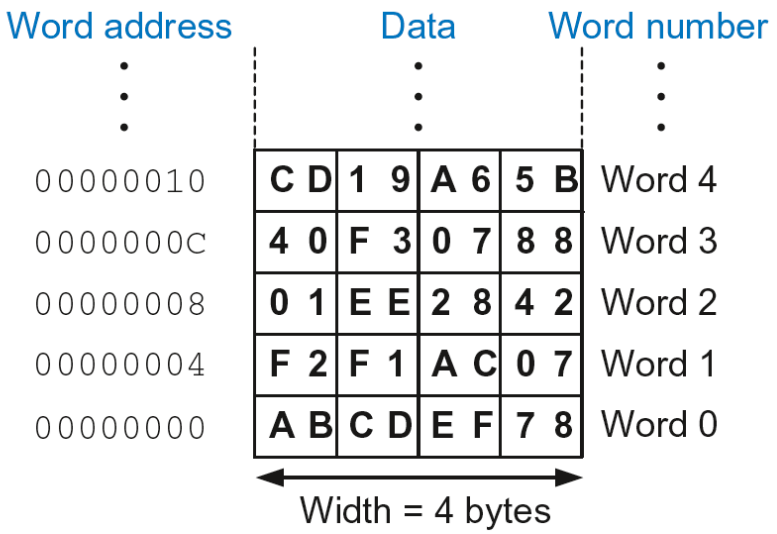
Writing Memory

- **Example:** Store the value held in R7 into memory word 21.
- Memory address = 4 x 21 = 84 = 0x54

ARM assembly code

```
MOV R5, #0
STR R7, [R5, #0x54]
```

The offset can be written in decimal or hexadecimal



Recap: Accessing Memory

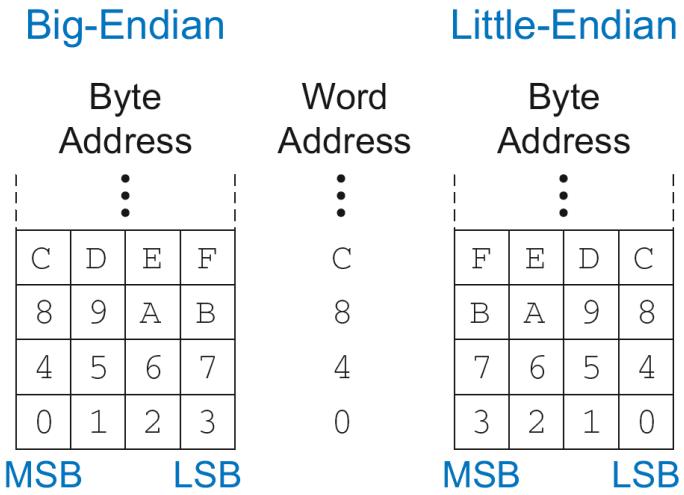
- Address of a memory **word** must be multiplied by 4
- **Examples:**
  - Address of memory word 2 = $2 \times 4 = 8$
  - Address of memory word 10 = $10 \times 4 = 40$

Big-Endian & Little-Endian Memory

- **How to number bytes within a word?**

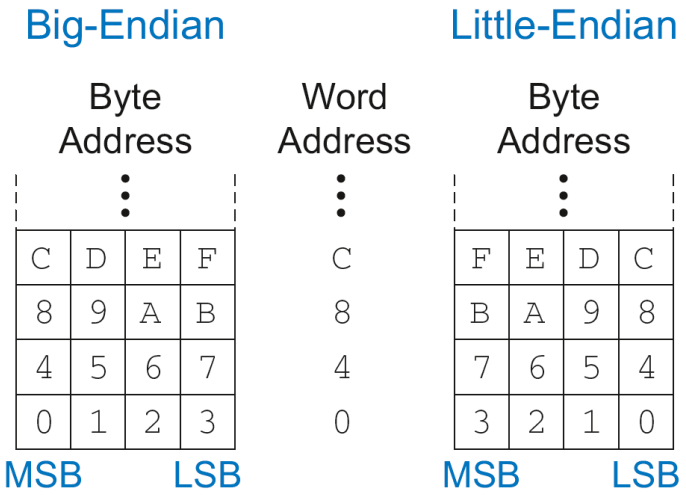
Big-Endian & Little-Endian Memory

- **How to number bytes within a word?**
  - **Little-endian:** byte numbers start at the **little** (least significant) end
  - **Big-endian:** byte numbers start at the **big** (most significant) end



Big-Endian & Little-Endian Memory

- **Jonathan Swift's *Gulliver's Travels*:** the Little-Endians broke their eggs on the little end of the egg and the Big-Endians broke their eggs on the big end
- **It doesn't really matter** which addressing type used – **except** when two systems **share data**



Programming

High-level languages:

- e.g., C, Java, Python
- Written at higher level of abstraction

Ada Lovelace, 1815-1852

- British mathematician
- Wrote the first computer program
- Her program calculated the Bernoulli numbers on Charles Babbage's Analytical Engine
- She was a child of the poet Lord Byron

Programming Building Blocks

- **Data-processing Instructions**
- **Conditional Execution**
- **Branches**
- **High-level Constructs:**
  - if/else statements
  - for loops
  - while loops
  - arrays
  - function calls

Programming Building Blocks

- **Data-processing Instructions**
- **Conditional Execution**
- **Branches**
- **High-level Constructs:**
  - if/else statements
  - for loops
  - while loops
  - arrays
  - function calls

Data-processing Instructions

- Logical operations
- Shifts / rotate
- Multiplication

Logical Instructions

- `AND`
- `ORR` **(OR)**
- `EOR` **(XOR)**
- `BIC` **(Bit Clear)**
  - Operano tutte bit a bit
  - Prima sorgente un registro, seconda registro/immediato

- `MVN` **(MoVe and NOT)**

Logical Instructions: Examples

Source registers

R1	0100 0110	1010 0001	1111 0001	1011 0111
R2	1111 1111	1111 1111	0000 0000	0000 0000

Assembly code

```
AND  R3, R1, R2
ORR  R4, R1, R2
EOR  R5, R1, R2
BIC  R6, R1, R2
MVN  R7, R2
```

Result

R3	0100 0110	1010 0001	0000 0000	0000 0000
R4	1111 1111	1111 1111	1111 0001	1011 0111
R5	1011 1001	0101 1110	1111 0001	1011 0111
R6	0000 0000	0000 0000	1111 0001	1011 0111
R7	0000 0000	0000 0000	1111 1111	1111 1111

Logical Instructions: Uses

- `AND` or `BIC`: useful for **masking** bits

Logical Instructions: Uses

- `AND` or `BIC`: useful for **masking** bits

  Example: Masking all but the least significant byte of a value

  0xF234012F `AND` 0x000000FF = 0x0000002F

  0xF234012F `BIC` 0xFFFFFF00 = 0x0000002F

Logical Instructions: Uses

- `AND` or `BIC`: useful for **masking** bits

  Example: Masking all but the least significant byte of a value

  0xF234012F `AND` 0x000000FF = 0x0000002F

  0xF234012F `BIC` 0xFFFFFF00 = 0x0000002F

- `ORR`: useful for **combining** bit fields

Logical Instructions: Uses

- `AND` or `BIC`: useful for **masking** bits

  Example: Masking all but the least significant byte of a value

  0xF234012F `AND` 0x000000FF = 0x0000002F

  0xF234012F `BIC` 0xFFFFFF00 = 0x0000002F

- `ORR`: useful for **combining** bit fields

  Example: Combine 0xF2340000 with 0x000012BC:

  0xF2340000 `ORR` 0x000012BC = 0xF23412BC

Generating Constants

Generare costanti usando `MOV` e `ORR`:

C Code

```
int a = 0x7EDC8765;
```

ARM Assembly Code

```
; R0 = a
MOV R0, #0x7E000000
ORR R0, R0, #0xDC0000
ORR R0, R0, #0x8700
ORR R0, R0, #0x65
```