

# **Modulo 1: Introduzione al Calcolo Automatico e Fondamenti di Informatica**

Francesco Sisini<sup>1</sup>, Lisa Galvani<sup>2</sup>, and Annalisa Pazzi<sup>3</sup>

<sup>1</sup>Docente esperto di informatica e matematica applicate alle STEM

<sup>2</sup>Tutor corsi STEM

<sup>3</sup>Consulenza artistica e creativa

# Capitolo 1

## Teoria degli automi

### Introduzione storica al concetto di calcolo automatico

L'idea di calcolo automatico affonda le sue radici nel XVII secolo con il filosofo e matematico **Gottfried Wilhelm Leibniz**, che concepì la possibilità di una “*macchina di ragionamento*” capace di eseguire calcoli logici e matematici in modo meccanico. Questi concetti furono formalizzati nel corso dei secoli successivi, culminando con l'introduzione del **calcolo simbolico** e degli **algoritmi** come strumenti fondamentali per automatizzare operazioni ripetitive.

Con l'avvento del XX secolo, **Alan Turing** diede un contributo decisivo, introducendo la **Macchina di Turing**, un modello astratto che rappresenta qualsiasi calcolo eseguibile tramite un algoritmo. Questo concetto segnò la nascita della *teoria del calcolo* e gettò le basi per la creazione dei moderni computer.

Parallelamente, l'interesse per la formalizzazione dei linguaggi, attraverso la grammatica e la logica, portò allo sviluppo della **teoria dei linguaggi formali**. Autori come **Noam Chomsky** classificarono i linguaggi in gerarchie, legando il concetto di linguaggio a quello di automi.

Infine, grazie ai lavori pionieristici di **Claude Shannon**, nacque la **teoria dell'informazione**, che definì formalmente il concetto di *informazione*, compressione dei dati e trasmissione efficiente dei messaggi, elementi alla base delle tecnologie digitali moderne.

Dalle intuizioni di Leibniz alle odierni applicazioni dell'informatica, il calcolo automatico, il linguaggio e la teoria dell'informazione hanno trasformato radicalmente la nostra capacità di risolvere problemi e comunicare, rappresentando il fondamento del mondo digitale contemporaneo.

### Concetto di Automa e Classificazione secondo Chomsky

#### Cos'è un Automa?

Un **automa** è un modello matematico che rappresenta un sistema capace di eseguire calcoli o riconoscere linguaggi tramite una serie di stati e transizioni. Esso è un concetto fondamentale in informatica teorica e automazione.

Gli automi possono essere rappresentati attraverso:

- Stati (finito o infinito).
- Simboli di ingresso (alfabeto).
- Transizioni tra stati.

Un automa riceve un input (sequenza di simboli) e si muove tra stati seguendo regole definite. Il comportamento è deterministico (determinato) o non deterministico (scelte multiple).

## Classificazione degli Automi secondo Chomsky

La classificazione di Chomsky distingue quattro tipi principali di automi, ciascuno capace di riconoscere linguaggi diversi:

1. **Automi a Stati Finiti (Finite State Automata - FSA)**: Riconoscono linguaggi regolari. Non hanno memoria esterna e si limitano a muoversi tra stati finiti.
2. **Automi a Pila (Pushdown Automata - PDA)**: Riconoscono linguaggi liberi dal contesto. Dispongono di una memoria a pila.
3. **Macchine di Turing**: Riconoscono linguaggi ricorsivamente enumerabili. Dispongono di un nastro infinito che funge da memoria.
4. **Automi Lineari Limitati (Linear Bounded Automata - LBA)**: Variante della Macchina di Turing con memoria limitata.

## Esempio: Automa a Stati Finiti

Consideriamo un **Automa a Stati Finiti Deterministico (DFA)** che riconosce stringhe composte da simboli  $\{0, 1\}$  e che terminano con il simbolo "1".

### Definizione Formale

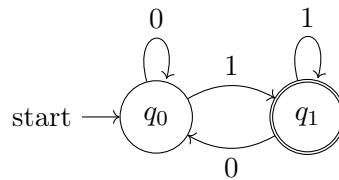
L'automa è definito dalla quintupola  $(Q, \Sigma, \delta, q_0, F)$  dove:

- $Q$ : Insieme degli stati,  $Q = \{q_0, q_1\}$ .
- $\Sigma$ : Alfabeto,  $\Sigma = \{0, 1\}$ .
- $\delta$ : Funzione di transizione.
- $q_0$ : Stato iniziale.
- $F$ : Stato finale,  $F = \{q_1\}$ .

### Tabella di Transizione

Stato	Input	Nuovo Stato
$q_0$	0	$q_0$
$q_0$	1	$q_1$
$q_1$	0	$q_0$
$q_1$	1	$q_1$

### Schema Grafico dell'Automa



### Esempio di Esecuzione

Se l'input è la stringa 1011:

- Stato iniziale:  $q_0$ .
- Legge 1: passa a  $q_1$ .
- Legge 0: torna a  $q_0$ .
- Legge 1: va a  $q_1$ .
- Legge 1: rimane a  $q_1$ .
- Stato finale:  $q_1$  (accettazione).

La stringa viene accettata perché termina in "1".

## Lezione 1: Macchina di Turing e il Concetto di Calcolo Automatico

### Obiettivi

- Comprendere il concetto di algoritmo e computabilità.
- Conoscere la figura di Alan Turing e il funzionamento della Macchina di Turing.
- Sviluppare algoritmi semplici manualmente.

### La Macchina di Turing

La Macchina di Turing è un modello astratto di calcolatore ideato da Alan Turing per rappresentare il processo di calcolo automatico. Consiste in:

- Un nastro infinito diviso in celle, che rappresenta la memoria.
- Una testina di lettura/scrittura che si sposta sul nastro.
- Uno stato interno che indica il "programma" attualmente eseguito.
- Una tabella di transizione che definisce le azioni della macchina in base allo stato e al simbolo letto.

## Esempio: Somma di Numeri Interi (0-10)

Supponiamo di sommare due numeri interi compresi tra 0 e 10, rappresentati in formato unario:

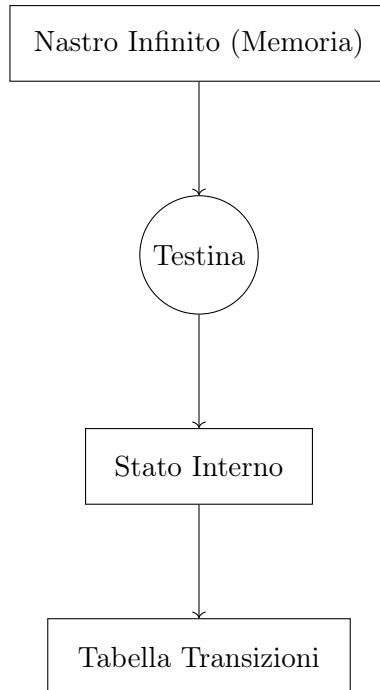
- 2 è rappresentato come 11, 3 come 111.
- La macchina sposta i simboli "1" del primo gruppo accanto al secondo gruppo.

**Tabella delle Transizioni per la Somma Aritmetica**

Stato	Simbolo Letto	Scrivi	Movimento	Nuovo Stato
$q_0$	1	—	Destra	$q_1$
$q_1$	—	1	Destra	$q_1$
$q_1$	1	1	Destra	$q_1$
$q_1$	—	—	Sinistra	$q_2$
$q_2$	1	1	Sinistra	$q_2$
$q_2$	—	—	Stop	Halt

**Risultato Finale:** Il nastro conterrà i due gruppi di "1" concatenati, rappresentando la somma unaria.

## Schema della Macchina di Turing



## Lezione 2: Architettura Base del Computer

### Obiettivi

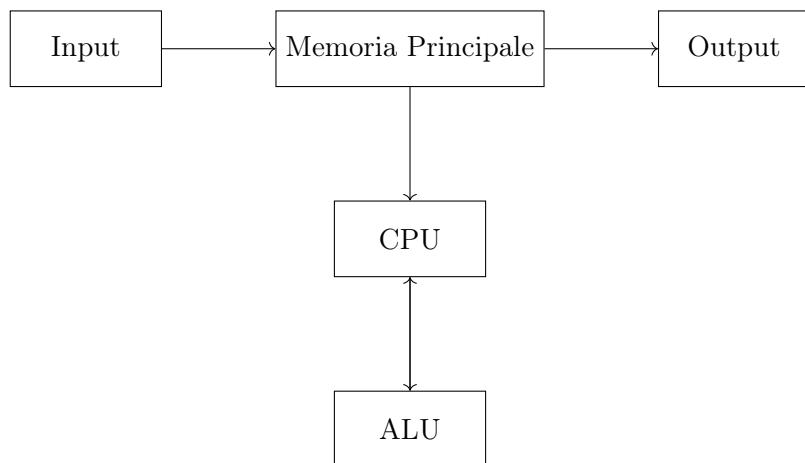
- Comprendere la struttura interna di un computer.
- Conoscere i componenti principali: ALU, CPU, registri e memoria.
- Simulare il funzionamento della CPU con un esempio pratico.

### Registri Principali

I registri sono celle di memoria velocissime interne alla CPU. Ecco i registri principali:

- **Registro Istruzione (IR)**: Contiene l'istruzione attualmente in esecuzione.
- **Program Counter (PC)**: Tiene traccia dell'indirizzo della prossima istruzione da eseguire.
- **Accumulator (ACC)**: Usato per operazioni aritmetiche e logiche.
- **Registro Dati (DR)**: Contiene i dati temporanei utilizzati durante l'elaborazione.

### Schema dell'Architettura della CPU



## Esempio: Automa a Stati Finiti per un Ascensore Semplice

Un automa a stati finiti per un ascensore semplice si basa su un modello deterministico (DFA) che gestisce solo l'apertura e la chiusura delle porte e il movimento tra i piani. Non ha memoria dei piani visitati in precedenza.

### Definizione Formale

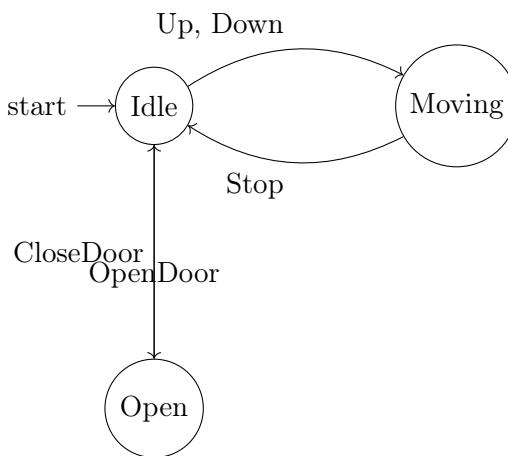
L'automa può essere definito dalla quintupola  $(Q, \Sigma, \delta, q_0, F)$  dove:

- $Q = \{\text{Idle}, \text{Moving}, \text{Open}\}$ : Stati possibili.
- $\Sigma = \{\text{Up}, \text{Down}, \text{OpenDoor}, \text{CloseDoor}\}$ : Input possibili.
- $\delta$ : Funzione di transizione definita da una tabella (vedi sotto).
- $q_0 = \text{Idle}$ : Stato iniziale.
- $F = \{\text{Idle}\}$ : Stato finale (opzionale, quando l'ascensore è inattivo).

### Tabella di Transizione

Stato Attuale	Input	Nuovo Stato
Idle	Up	Moving
Idle	Down	Moving
Idle	OpenDoor	Open
Moving	Stop	Idle
Open	CloseDoor	Idle

### Schema Grafico dell'Automa



## Esempio: Automa a Pila per un Ascensore con Memoria del Piano

Un automa a pila (PDA) consente di tenere traccia dei piani visitati utilizzando una pila per memorizzare le sequenze di piani. Questo modello è utile per ascensori che ricordano i piani richiesti e visitati in ordine.

### Definizione Formale

L'automa è definito da  $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$  dove:

- $Q = \{\text{Idle}, \text{Moving}, \text{Open}\}$ : Stati possibili.
- $\Sigma = \{\text{Up}, \text{Down}, \text{OpenDoor}, \text{CloseDoor}, \text{Request}(n)\}$ : Input possibili.
- $\Gamma = \{1, 2, 3, \dots, Z_0\}$ : Alfabeto della pila (piani richiesti).
- $\delta$ : Funzione di transizione (vedi sotto).
- $q_0 = \text{Idle}$ : Stato iniziale.
- $Z_0$ : Simbolo iniziale della pila.
- $F = \{\text{Idle}\}$ : Stato finale (opzionale, quando l'ascensore è inattivo).

### Funzione di Transizione

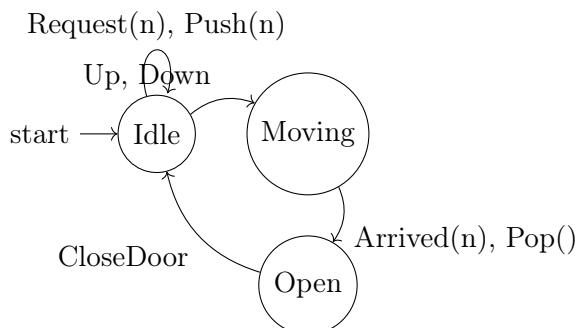
Le transizioni aggiungono piani alla pila e li rimuovono quando vengono visitati.

$$\begin{aligned} \delta(q, \text{Request}(n), Z) &= (q, \text{Push}(n)) \\ \delta(\text{Moving}, \text{Arrived}(n), n) &= (\text{Open}, \text{Pop}()) \end{aligned}$$

### Tabella di Transizione

Stato Attuale	Input	Simbolo Pila	Nuovo Stato / Azione
Idle	Request(n)	$Z_0$	Idle, Push(n)
Idle	Up, Down	n	Moving
Moving	Arrived(n)	n	Open, Pop()
Open	CloseDoor	-	Idle

### Schema Grafico dell'Automa



## Spiegazione del Funzionamento dell'Automa a Pila

L'automa a pila per l'ascensore con memoria del piano utilizza una struttura dati a pila per ricordare i piani richiesti. Ogni volta che un piano viene richiesto, il numero del piano viene aggiunto alla pila. Quando l'ascensore raggiunge un piano, il numero del piano viene rimosso dalla pila. Questo meccanismo consente all'ascensore di tenere traccia dell'ordine dei piani da visitare, in base alle richieste degli utenti.

### Dettaglio del Funzionamento

1. **Richiesta di un Piano:** Quando un utente richiede di andare a un piano (`Request(n)`), l'automa passa allo stato `Idle`, mantenendo il piano richiesto nella pila tramite l'operazione di `Push(n)`.
2. **Movimento:** Una volta che l'ascensore inizia a muoversi verso un piano (Up o Down), l'automa entra nello stato `Moving`.
3. **Arrivo al Piano:** Quando l'ascensore raggiunge un piano specifico (`Arrived(n)`), verifica che il piano corrisponda al valore in cima alla pila. Se corrisponde, il numero viene rimosso dalla pila tramite l'operazione `Pop()`, e l'automa passa allo stato `Open`.
4. **Apertura e Chiusura delle Porte:** Lo stato `Open` indica che le porte dell'ascensore sono aperte. Dopo che le porte vengono chiuse (`CloseDoor`), l'automa torna allo stato `Idle`.
5. **Verifica della Pila:** L'automa controlla se la pila è vuota. Se lo è, l'ascensore rimane nello stato `Idle`. In caso contrario, prosegue verso il prossimo piano in cima alla pila.

### Esempio di Esecuzione

Consideriamo un esempio pratico:

- Un utente richiede i piani 3 e 5 in quest'ordine (`Request(3)` e `Request(5)`).
- La pila inizialmente contiene il simbolo iniziale  $Z_0$ . Dopo le richieste, la pila sarà:  $[Z_0, 5, 3]$ .
- L'ascensore si muove (Up) e raggiunge il piano 3 (`Arrived(3)`). Il piano 3 viene rimosso dalla pila tramite `Pop()`, lasciando la pila con  $[Z_0, 5]$ .
- Dopo aver aperto e chiuso le porte, l'ascensore riparte verso il piano 5. Al suo arrivo (`Arrived(5)`), il piano 5 viene rimosso dalla pila, che ritorna allo stato iniziale  $[Z_0]$ .
- L'automa ritorna allo stato `Idle`, pronto per nuove richieste.

### Vantaggi del Modello a Pila

Questo modello consente di gestire in modo efficiente richieste multiple, garantendo che l'ascensore visiti i piani in ordine di richiesta. Inoltre, utilizza la pila per ricordare i piani ancora da visitare, evitando comportamenti incoerenti o disordinati.

# Approfondimenti sul Funzionamento dell'Automa a Pila

## Come l'Automa Riconosce di Essere Giunto a un Piano

L'automa sa di essere giunto a un determinato piano grazie a un segnale esterno, come un sensore installato in ciascun piano dell'edificio. Quando l'ascensore si ferma in corrispondenza del piano, il sensore invia un segnale all'automa, indicando il numero del piano. Questo segnale viene interpretato come l'input `Arrived(n)`.

Ad esempio:

- Se l'ascensore si ferma al 3° piano, il sensore invia l'input `Arrived(3)`.
- L'automa confronta questo input con il simbolo in cima alla pila. Se il simbolo in cima è "3", l'automa conferma di essere giunto al piano richiesto e procede a rimuovere "3" dalla pila tramite l'operazione di `Pop()`.

Questo meccanismo garantisce che l'automa non rimuova piani dalla pila in modo casuale, ma solo quando corrispondono al piano effettivamente raggiunto.

## Esempio: Raggiungere un Piano Diverso dall'Ordine di Richiesta

Consideriamo un esempio in cui l'ascensore riceve due richieste: prima per il 5° piano (`Request(5)`) e poi per il 3° piano (`Request(3)`). La pila viene gestita in ordine LIFO (Last In, First Out), quindi la richiesta per il piano 3 viene posizionata sopra quella per il piano 5.

### Passaggi Eseguiti dall'Automa

- 1. Stato Iniziale:** La pila contiene  $[Z_0, 5, 3]$ . L'automa si trova nello stato `Idle`.
- 2. Movimento verso il 3° piano:**
  - L'ascensore inizia a muoversi (`Up`) verso il piano più vicino.
  - Quando arriva al 3° piano, il sensore invia l'input `Arrived(3)`.
  - L'automa verifica che il simbolo in cima alla pila sia "3". Poiché c'è corrispondenza, rimuove "3" dalla pila (`Pop()`), lasciando la pila con  $[Z_0, 5]$ .
- 3. Apertura e Chiusura delle Porte al 3° piano:**
  - L'automa passa allo stato `Open`, indicando che le porte sono aperte.
  - Dopo che le porte si richiudono (`CloseDoor`), l'automa ritorna allo stato `Idle`.
- 4. Movimento verso il 5° piano:**
  - L'ascensore riparte verso il 5° piano.
  - Quando arriva al 5° piano, il sensore invia l'input `Arrived(5)`.
  - L'automa verifica che il simbolo in cima alla pila sia "5". Poiché c'è corrispondenza, rimuove "5" dalla pila (`Pop()`), lasciando la pila con  $[Z_0]$ .

### 5. Apertura e Chiusura delle Porte al 5° piano:

- L'automa passa allo stato **Open**, indicando che le porte sono aperte.
- Dopo che le porte si richiudono (**CloseDoor**), l'automa ritorna allo stato **Idle**.

### Osservazioni Importanti

- L'automa non si preoccupa di eseguire le richieste in ordine di inserimento nella pila. Segue invece l'ordine LIFO, servendo prima il piano più vicino (in cima alla pila).
- Questo comportamento potrebbe essere subottimale in alcuni contesti reali, ma riflette accuratamente il modello di funzionamento di una pila (Last In, First Out).

### Vantaggi e Limitazioni del Modello

- **Vantaggi:** La pila semplifica la gestione delle richieste e permette un controllo chiaro e deterministico dei piani visitati.
- **Limitazioni:** L'ordine LIFO potrebbe non essere ottimale in contesti con molte richieste, poiché i piani richiesti per primi potrebbero essere serviti per ultimi.

# Capitolo 2

## Un Gioco Collaborativo con Arduino

### 2.1 Introduzione

Lo scopo di questo capitolo è presentare un semplice gioco basato su **automazione a stati finiti**, realizzato con **Arduino** e due pulsanti. Il gioco nasce come esempio di **gioco cooperativo** nell'ambito della **teoria dei giochi** di John von Neumann.

Il sistema prevede quattro stati principali, associati a quattro LED di colori diversi (Rosso R, Verde G, Blu B, Bianco W). A ogni pressione di un pulsante, il sistema cambia stato secondo una tabella di transizione concordata. L'obiettivo globale è produrre la sequenza

$$W \rightarrow G \rightarrow R \rightarrow G \rightarrow R \rightarrow G \rightarrow B \rightarrow W,$$

sequenza che, come vedremo, non può essere realizzata da un singolo giocatore, ma richiede la collaborazione di entrambi.

### 2.2 Inquadramento nella teoria dei giochi

La *teoria dei giochi* di von Neumann analizza le decisioni strategiche di agenti (giocatori) che operano in un sistema di regole ben definito. Nei *giochi competitivi*, ogni giocatore massimizza il proprio payoff individuale. Tuttavia, esistono anche *giochi cooperativi* o collaborativi, dove due (o più) giocatori devono *cooperare* per raggiungere un obiettivo comune.

Nel nostro caso:

- I giocatori sono due *persone* (due pulsanti).
- Non hanno interessi in conflitto: **entrambi** vogliono ottenere la sequenza desiderata.
- Di conseguenza, non siamo in presenza di una competizione a somma zero, bensì di un gioco *collaborativo*.

Poiché la sequenza da produrre include alternanze di mosse (pressioni) da parte del “giocatore 1” e del “giocatore 2”, un solo giocatore non può completare l’intero percorso. Serve necessariamente l’azione dell’altro.

### 2.3 Modello come automa a stati finiti

Formalmente, possiamo descrivere il sistema come un **automa a stati finiti**:

- $\Sigma = \{P_1, P_2\}$  è l'insieme degli *ingressi* (pulsante 1 e pulsante 2).
- $Q = \{R, G, B, W\}$  è l'insieme degli *stati* (Rosso, Verde, Blu, Bianco).
- $\delta: Q \times \Sigma \rightarrow Q$  è la funzione di transizione. Ad esempio:

$$\delta(R, P_1) = G, \quad \delta(R, P_2) = B, \quad \dots$$

secondo la tabella di gioco concordata.

- $q_0 = R$  è lo *stato iniziale*.

Premendo i pulsanti (ingressi), i giocatori causano transizioni nello spazio degli stati. L'obiettivo di squadra è passare per una certa sequenza di stati (corrispondente ai colori dei LED) che *non* è realizzabile da un singolo giocatore.

## 2.4 Realizzazione con Arduino

### 2.4.1 Descrizione hardware

La realizzazione hardware prevede:

- Una scheda Arduino (ad es. Arduino UNO).
- Due pulsanti di input (BUTTON1 e BUTTON2), ciascuno con una resistenza di *pull-down* da 10 k $\Omega$ .
- Quattro LED (RED\_LED, GREEN\_LED, BLUE\_LED, WHITE\_LED) per indicare lo stato corrente.

Il pulsante 1 è collegato al pin digitale 2, il pulsante 2 al pin 3. I LED sono collegati a 9, 10, 11, 12 rispettivamente (o a discrezione del progettista).

### 2.4.2 Schema dei collegamenti

In modo semplificato:

- BUTTON1: un capo a +5V, l'altro al pin 2, e in parallelo una resistenza da 10 k $\Omega$  tra il pin 2 e GND.
- BUTTON2: un capo a +5V, l'altro al pin 3, e in parallelo una resistenza da 10 k $\Omega$  tra il pin 3 e GND.
- RED\_LED: pin 9 → Resistenza (220  $\Omega$ ) → Anodo LED, catodo a GND.
- GREEN\_LED, BLUE\_LED, WHITE\_LED similmente, sui pin 10, 11, 12.

### 2.4.3 Codice di esempio

Il seguente listato ?? implementa la logica di cambiamento di stato in base al pulsante premuto. Ogni volta che il giocatore 1 (pulsante su pin 2) preme, passiamo a un “prossimo” colore; se invece preme il giocatore 2 (pulsante su pin 3), la transizione segue la colonna dedicata a player 2.

```
1  ****
2  * Gioco collaborativo con 4 LED e 2 pulsanti
3  * Basato su transizioni di stato distinte per P1 e P2
4  ****
5 // Pin LED
6 const int RED_LED = 9;
7 const int GREEN_LED = 10;
8 const int BLUE_LED = 11;
9 const int WHITE_LED = 12;
10
11 // Pin pulsanti (pull-down esterna)
12 const int BUTTON1 = 2;
13 const int BUTTON2 = 3;
14
15 // Stato iniziale
16 char currentState = 'R'; // Rosso
17
18 void setup() {
19     Serial.begin(9600);
20     pinMode(RED_LED, OUTPUT);
21     pinMode(GREEN_LED, OUTPUT);
22     pinMode(BLUE_LED, OUTPUT);
23     pinMode(WHITE_LED, OUTPUT);
24
25     pinMode(BUTTON1, INPUT);
26     pinMode(BUTTON2, INPUT);
27
28     // Spegne tutti i LED
29     digitalWrite(RED_LED, LOW);
30     digitalWrite(GREEN_LED, LOW);
31     digitalWrite(BLUE_LED, LOW);
32     digitalWrite(WHITE_LED, LOW);
33
34     // Accende il LED iniziale
35     updateLED(currentState);
36     Serial.println("Gioco avviato. Stato iniziale: R (Rosso).");
37 }
38
39 void loop() {
40     // Lettura dei pulsanti
41     int stateButton1 = digitalRead(BUTTON1);
42     int stateButton2 = digitalRead(BUTTON2);
43
44     // Se pulsante 1 premuto
45     if (stateButton1 == HIGH) {
46         nextState(1);
47         delay(200); // per evitare ripetizioni veloci
48     }
49 }
```

```
50 // Se pulsante 2 premuto
51 if (stateButton2 == HIGH) {
52     nextState(2);
53     delay(200);
54 }
55 }

56 // nextState: cambia stato a seconda del giocatore
57 void nextState(int player) {
58     Serial.print("Stato attuale: ");
59     Serial.print(currentState);
60     Serial.print(" | Giocatore: ");
61     Serial.println(player);

62     if (currentState == 'R') {
63         currentState = (player == 1) ? 'G' : 'B';
64     }
65     else if (currentState == 'G') {
66         currentState = (player == 1) ? 'B' : 'R';
67     }
68     else if (currentState == 'B') {
69         currentState = 'W'; // per entrambi i giocatori
70     }
71     else if (currentState == 'W') {
72         currentState = (player == 1) ? 'R' : 'G';
73     }
74 }

75 Serial.print("Nuovo stato: ");
76 Serial.println(currentState);

77 updateLED(currentState);
78 }

79 // updateLED: spegne tutti e accende solo quello del colore
80 // corrente
81 void updateLED(char state) {
82     digitalWrite(RED_LED, LOW);
83     digitalWrite(GREEN_LED, LOW);
84     digitalWrite(BLUE_LED, LOW);
85     digitalWrite(WHITE_LED, LOW);

86     switch(state) {
87         case 'R': digitalWrite(RED_LED, HIGH); break;
88         case 'G': digitalWrite(GREEN_LED, HIGH); break;
89         case 'B': digitalWrite(BLUE_LED, HIGH); break;
90         case 'W': digitalWrite(WHITE_LED, HIGH); break;
91     }
92 }
93 }
```

---

Listing 2.1: Codice Arduino

## 2.5 Proposta di ingegnerizzazione

Come esercitazione di laboratorio, si può proporre agli studenti di:

1. **Progettare il circuito stampato (PCB)**: partendo dallo schema elettrico (Arduino, due pulsanti, quattro LED), realizzare uno *shield* o un PCB dedicato.
2. **Integrare un display o indicatori aggiuntivi**: mostrare la sequenza già realizzata e quella ancora necessaria per il completamento del gioco.
3. **Introdurre una temporizzazione o penalità**: ad esempio, se un giocatore preme troppo presto o troppo tardi, inserire meccanismi di *timeout* e notifiche sonore.
4. **Analisi matematica di cooperazione**: quantificare il numero di *mosse* necessarie per completare la sequenza e dimostrare che un giocatore da solo non può arrivare alla soluzione finale.

## 2.6 Conclusioni

Abbiamo descritto un semplice *automa a stati finiti* con quattro stati (R, G, B, W) e due ingressi (i pulsanti) che, premuti con opportune transizioni, possono realizzare una sequenza obiettivo che **richiede** la cooperazione di entrambi i giocatori. La natura *collaborativa* del gioco lo inserisce pienamente fra i giochi cooperativi della teoria di von Neumann, evidenziando come in alcune situazioni i giocatori non siano *concorrenti*, bensì *alleati* verso un unico scopo.

Questo semplice prototipo può essere esteso e migliorato, offrendo numerose occasioni di riflessione e *apprendimento pratico* sull'elettronica di base, sulla programmazione embedded e sulle implicazioni teoriche di cooperazione tra agenti.

# Capitolo 3

## Aritmetica e Logica Binaria

<b>Indice dei Contenuti</b>		
1.	Rappresentazione binaria .....	17
2.	Operazioni logiche (true e false) .....	19
3.	Aritmetica binaria (somma, sottrazione, moltiplicazione, divisione) .....	20
4.	Rappresentazione algebrica della logica (Algebra di Boole) .....	21
5.	Aritmetica binaria con porte logiche .....	21
6.	Aritmetica binaria implementata come un automa a pila (semplice) .....	22
7.	Aritmetica decimale implementata con un automa a pila (solo decenni) .....	23

### 3.1 Rappresentazione binaria

La **rappresentazione binaria** di un numero intero non negativo  $n$  è data da una sequenza di bit  $(b_k b_{k-1} \dots b_1 b_0)$  tale che:

$$n = \sum_{i=0}^k b_i 2^i, \quad \text{dove ciascun } b_i \in \{0, 1\}.$$

Esempi:

$$13_{10} = 1101_2,$$

$$5_{10} = 101_2,$$

$$0_{10} = 0_2,$$

$$1_{10} = 1_2.$$

#### Tabella di alcuni numeri in binario/decimale

Decimale	Binario	Decimale	Binario
0	0	8	1000
1	1	9	1001
2	10	10	1010
3	11	11	1011
4	100	12	1100
5	101	13	1101
6	110	14	1110
7	111	15	1111

Questa tabella estende i primi 16 numeri, evidenziando come ogni valore decimale corrisponda a univoca sequenza di bit.

#### Esercizi di Conversione

- Converti i seguenti numeri decimali in binario: 18, 20, 31, 32, 40, 50, 64, 100.
- Converti da binario a decimale i seguenti numeri:  $10101_2, 100000_2, 111111_2, 1100100_2$ .
- Spiega (in almeno 5-6 righe) come eseguire manualmente la conversione da decimale a binario, usando il metodo delle divisioni successive per 2.

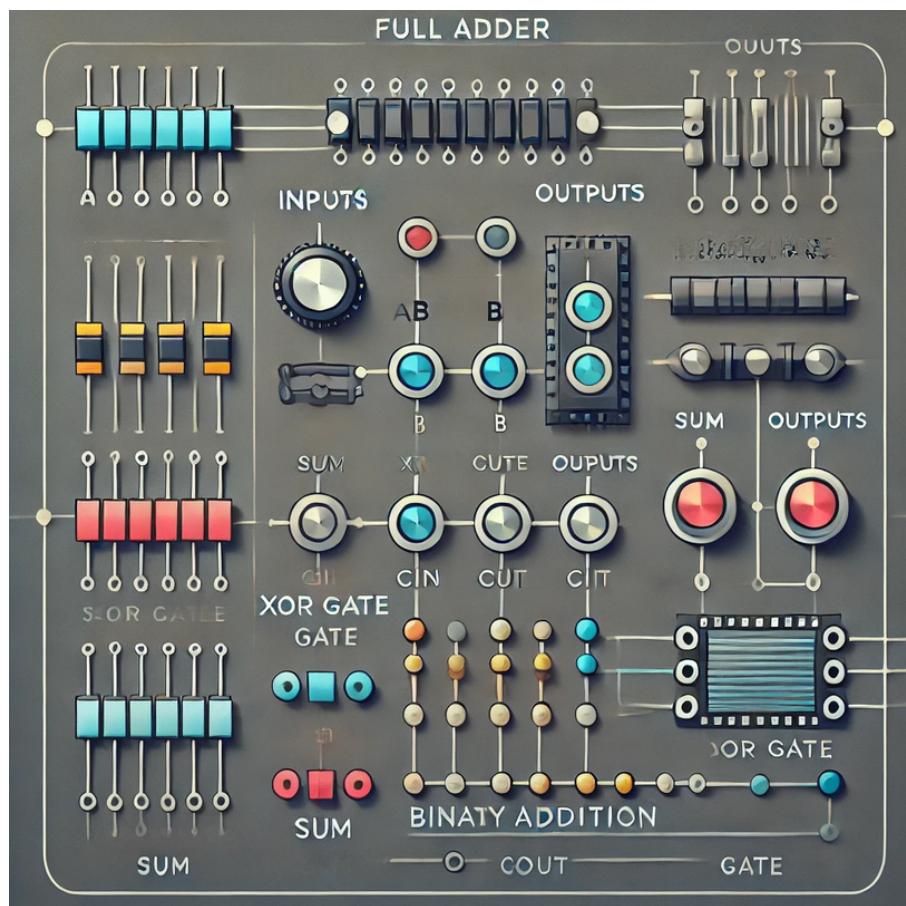


Figura 3.1: Diagramma del full adder binario: illustra in modo fantasioso le componenti principali, incluse le porte logiche XOR, AND e OR, con ingressi e uscite etichettate.

## 3.2 Operazioni logiche (true e false)

La **logica booleana** ammette due valori: TRUE (vero) e FALSE (falso). Le *operazioni fondamentali* sono:

$$\wedge \text{ (AND)}, \quad \vee \text{ (OR)}, \quad \neg \text{ (NOT)}.$$

Le relative tabelle di verità sono:

<b>AND (<math>\wedge</math>)</b>			<b>OR (<math>\vee</math>)</b>			<b>NOT (<math>\neg</math>)</b>	
<b>A</b>	<b>B</b>	<b><math>A \wedge B</math></b>	<b>A</b>	<b>B</b>	<b><math>A \vee B</math></b>	<b>A</b>	<b><math>\neg A</math></b>
F	F	F	F	F	F	F	T
F	T	F	F	T	T	T	F
T	F	F	T	F	T	F	T
T	T	T	T	T	T		

### Osservazioni

- L'operazione  $\wedge$  è “vera” solo se *entrambe* le variabili d'ingresso sono vere.
- $\vee$  è vera se almeno un operando è vero.
- $\neg$  inverte (NOT) il valore logico.

### Esercizi di Logica

1. Verifica  $(A \vee B) \wedge (A \vee \neg B)$  per tutti i possibili valori di  $A, B$ .
2. Dimostra che  $\neg(\neg A) = A$  con la tabella di verità.
3. Spiega che differenza c'è tra “AND esclusivo” ( $\oplus$ ) e “AND” ( $\wedge$ ).

### 3.3 Aritmetica binaria: somma, sottrazione, moltiplicazione, divisione

La **base 2** ( $\{0, 1\}$ ) richiede di gestire i riporti (o prestiti) in base 2.

#### 3.3.1 Somma binaria

$$0 + 0 = 0, \quad 0 + 1 = 1, \quad 1 + 0 = 1, \quad 1 + 1 = 0 \quad (\text{con riporto} = 1).$$

Esempio:  $(1001)_2 + (0110)_2 = (1111)_2$ .

#### 3.3.2 Tabella di somma binaria (2 bit + carry)

$x$	$y$	$c_{in}$	$s$	$c_{out}$
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Questa tabella mostra tutte le combinazioni di due bit  $x, y$  e un carry in ingresso  $c_{in}$ , e restituisce un bit di somma  $s$  e un carry in uscita  $c_{out}$ .

#### 3.3.3 Sottrazione binaria

Analogamente, se facciamo  $(x - y)$ , dobbiamo gestire il *borrow* (prestito). Esempio:  $(1000)_2 - (1)_2 = (0111)_2$ .

#### 3.3.4 Moltiplicazione e divisione binaria

Come in base 10, ma tutti i calcoli parziali usano bit.

$$(101)_2 \times (11)_2 = (1111)_2, \quad (1100)_2 \div (10)_2 = (110)_2, \dots$$

### Esercizi Aritmetici

1. Somma in binario:  $(1011)_2 + (0101)_2, (1111)_2 + (0001)_2$ .
2. Sottrai:  $(10000)_2 - (0111)_2$ .
3. Moltiplica:  $(101)_2 \times (101)_2$ .
4. Dividi:  $(1110)_2 \div (10)_2$ .
5. Riporta in decimale i risultati e controlla la correttezza.

### 3.4 Rappresentazione algebrica della logica (Algebra di Boole)

L'**Algebra di Boole** utilizza i valori  $\{0, 1\}$  con le operazioni  $\wedge$  (AND),  $\vee$  (OR),  $\neg$  (NOT). Le leggi di De Morgan affermano:

$$\neg(A \wedge B) = (\neg A) \vee (\neg B), \quad \neg(A \vee B) = (\neg A) \wedge (\neg B).$$

Altre leggi fondamentali:

$$A \wedge 1 = A, \quad A \wedge 0 = 0, \quad A \vee 1 = 1, \quad A \vee 0 = A, \quad (A \wedge B) \wedge C = A \wedge (B \wedge C), \dots$$

#### Tabella di Verità per De Morgan

$A$	$B$	$\neg(A \wedge B)$	$\neg A \vee \neg B$	Coincidenti?
0	0	1	1	Sì
0	1	1	1	Sì
1	0	1	1	Sì
1	1	0	0	Sì

Si vede che  $\neg(A \wedge B)$  e  $(\neg A) \vee (\neg B)$  danno gli stessi risultati.

#### Esercizi di Algebra Booleana

1. Semplifica  $A \wedge (A \vee B)$  usando le leggi di assorbimento.
2. Riscrivi  $\neg(A \vee (B \wedge C))$  usando De Morgan.
3. Dimostra che  $(A \wedge B) \wedge C = A \wedge (B \wedge C)$  (associatività).

### 3.5 Aritmetica binaria con porte logiche

Le operazioni binarie si implementano con **porte logiche**. Esempio: un *full adder* somma due bit  $x, y$  più un carry in ingresso  $c_{in}$ , restituendo la somma  $s$  e il carry in uscita  $c_{out}$ .

$x$	$y$	$c_{in}$	$s$ (somma)	$c_{out}$ (riporto)
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Con *porte XOR, AND, OR* si realizza questo comportamento:

$$\begin{cases} s = (x \oplus y) \oplus c_{in}, \\ c_{out} = (x \wedge y) \vee ((x \oplus y) \wedge c_{in}). \end{cases}$$

## Esempio di somma di 4 bit

Usando 4 full adders in cascata (uno per ciascuna posizione), si ottiene la somma di due numeri binari di 4 bit. Il carry in uscita di un full adder diventa il carry in ingresso di quello sulla cifra più significativa successiva.

## Esercizi sulle porte logiche

1. Disegna la porta XOR (o EX-OR) e spiega la differenza tra XOR e OR.
2. Calcola manualmente  $(1011)_2 + (0101)_2$  usando la logica dei carry in ingresso e in uscita.
3. Progetta un *half adder* (somma di 2 bit senza carry in ingresso) e scrivi la relativa tabella di verità.

## 3.6 Aritmetica binaria implementata come un automa a pila (semplice)

Mostriamo come un *automa a pila* (PDA) riconosca correttamente  $x + y = z$  in base 2, assumendo che le cifre siano lette “rovesciate” (cifra meno significativa per prima).

### Fasi principali

1. **Caricamento  $x$ :** leggiamo i bit di  $x$  e li empiliamo come simboli (ad es. X\_0 o X\_1).
2. **Lettura  $y$  e calcolo parziale:** ogni volta che leggiamo un bit di  $y$ , pop di un bit di  $x$  e somma binaria + carry. In cima alla pila spingiamo la cifra di somma (S\_0 o S\_1), mentre il carry (0 o 1) viene memorizzato nello *stato* dell'automa.
3. **Verifica di  $z$ :** confrontiamo i bit di  $z$  letti con la somma parziale presente in cima alla pila.
4. Accettiamo se, alla fine, la pila è vuota e il carry=0.

### Esempio: Somma $101 + 11 = 1000$

In “forma rovesciata” diventa  $101 + 11 = 0001$  (da destra a sinistra!). L'automa:

- Carica 101 in pila (X\_1, X\_0, X\_1).
- Legge 11, bit a bit, somma e produce la “somma parziale” in pila.
- Verifica che 0001 corrisponda al risultato.

In decimale,  $5 + 3 = 8$ .

## Esercizi

1. Scrivi la forma rovesciata di  $100 + 11 = 111$  e verifica i passaggi.
2. Descrivi almeno due stati diversi dell'automa: uno per la lettura di  $x$ , uno per la lettura di  $y$ .
3. Spiega perché un automa a stati finiti *senza pila* non è sufficiente a riconoscere correttamente tutte le somme binarie.

## 3.7 Aritmetica decimale implementata con un automa a pila (solo decenni)

Analogamente, **in base 10** (decimale) possiamo costruire un automa a pila che riconosca espressioni come  $x + y = z$ . Naturalmente, la gestione del *carry* va da 0 a 1 (per due cifre 0–9 la somma può arrivare a 18), ma il principio è lo stesso.

## Schema

1. Caricamento di  $x$  (cifre 0–9) in pila come X\_digit.
2. Lettura di  $y$  e somma delle cifre: pop di X\_digit e, con lo stato che memorizza carry 0 o 1, calcoliamo la cifra di somma (S\_digit) e lo spingiamo in pila.
3. Infine, verifichiamo  $z$  confrontando le cifre (ritte in forma rovesciata) con quelle in cima alla pila.
4. Accettiamo se la pila torna vuota e il carry finale è 0.

## Esempio

$$12 + 5 = 17 \Rightarrow \text{rovesciato: } 21 + 5 = 71.$$

In cifre rovesciate:

- Carichiamo 2 e 1 in pila (come X\_2 e X\_1).
- Leggiamo “5” per  $y$  e sommiamo X\_2 (in cima) con 5 più carry=0, ecc.
- Verifichiamo 7 e 1 nel risultato.

## Esercizi

1. Rovescia  $123 + 45 = 168$  e spiega come funziona il carry (0 o 1).
2. Disegna una piccola porzione di tabella di transizione, per lo stato “lettura di  $y$ , carry=1” e cima della pila = X\_7.
3. Argomenta (in 5 righe) se servono stati diversi per carry=0 e carry=1.

## Conclusioni Generali

Abbiamo visto:

- La **rappresentazione binaria** e come essa sia la base dell'informatica.
- Le **operazioni logiche**  $\wedge, \vee, \neg$  e l'algebra di Boole, fondamentali per l'elettronica digitale.
- L'**aritmetica binaria** (con esempi di somma, sottrazione, moltiplicazione e divisione).
- L'uso delle **porte logiche** per realizzare fisicamente tali operazioni (full adder, half adder, etc.).
- Come un **automa a pila** possa riconoscere " $x + y = z$ " sia in base 2 sia in base 10, *rovesciando* le cifre e memorizzandone alcune in pila.

**Ulteriori spunti:**

- Estendere il concetto all'algebra booleana di più variabili (circuiti con più ingressi).
- Analizzare la costruzione di una *piccola CPU* con sommatore binario a 8 bit.
- Verificare se per la *moltiplicazione in base 10* si può costruire un automa a pila simile (spoiler: si può, ma è più complesso).

# Capitolo 4

## Algoritmo di Somma e Architettura a Stati Finiti

### 4.1 Introduzione

In questo capitolo descriveremo in dettaglio l'algoritmo di somma mostrato nel diagramma di flusso e nell'architettura schematica, soffermandoci su un esempio in **binario**. L'obiettivo è comprendere come i registri (quali RC, ACC, AR, RI, ecc.) interagiscano con l'Unità Aritmetico-Logica (ALU) e con la memoria dati per svolgere l'operazione di addizione di due operandi, bit per bit, gestendo correttamente il riporto (**carry**).

Verranno analizzate le singole fasi dell'algoritmo, mettendo in luce:

- Il ruolo di ciascun registro;
- L'inizializzazione dei valori di **carry** e degli indici;
- Il ciclo di lettura dei bit, la somma e l'immagazzinamento del risultato parziale;
- Il controllo dell'indice e la gestione del **carry** finale.

Inoltre, metteremo in relazione questo processo con alcuni concetti fondamentali di teoria degli automi, come l'automa a stati finiti (FSM), l'automa a pila (PDA) e la Macchina di Turing (MdT). Tali riferimenti sono utili a comprendere la natura sequenziale e a stati dell'algoritmo, nonché l'idea di come si possa estendere a meccanismi di calcolo più potenti (ad esempio, se volessimo gestire un numero di bit potenzialmente molto grande).

### 4.2 Descrizione dell'Architettura

Lo schema a blocchi dell'architettura proposta mostra diversi registri e un'ALU (Arithmetic Logic Unit). I componenti principali sono:

- **AR (Address Register)**: Registro che contiene l'indirizzo di memoria corrente da cui leggere o in cui scrivere.
- **RI (Registro Indice)**: Registro che tiene traccia dell'indice corrente, utile per iterare sui bit dei due operandi.
- **RC (Registro Conteggio)**: Registro che contiene il numero di bit totali da sommare (o il numero di cicli da eseguire).

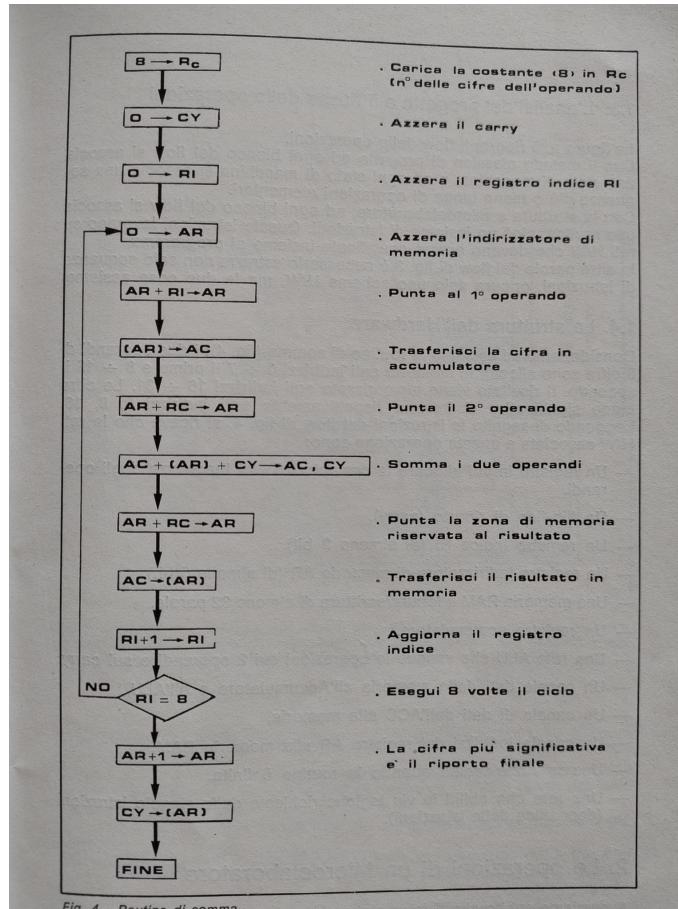


Figura 4.1: Diagramma di flusso dell'algoritmo di somma.

- **ACC (Accumulator):** Registro accumulatore in cui si depositano i dati temporanei e i risultati intermedi di calcolo.
- **CY (Carry):** Flip-flop o registro dedicato a immagazzinare il riporto (*carry*) risultante dall'operazione di somma.
- **ALU:** L'unità aritmetico-logica che, in base ai segnali di controllo, esegue l'operazione di addizione (o altre operazioni aritmetico-logiche).

La memoria dati è organizzata in celle, ciascuna delle quali può contenere un singolo bit dell'operando (o più bit, a seconda dell'implementazione). Per semplicità, supponiamo che ogni cella di memoria contenga **un singolo bit**.

Nell'architettura, vediamo come l'ALU abbia diversi ingressi provenienti da multiplexer (MUXA e MUXB), i quali selezionano quale registro o bus deve essere collegato all'ingresso dell'ALU. L'uscita dell'ALU, oltre a fornire il risultato dell'operazione, aggiorna il **carry** (CY).

### 4.3 Il Diagramma di Flusso: Fasi Principali

Il diagramma di flusso rappresenta l'algoritmo di somma passo per passo (Figura 4.1). Di seguito, riportiamo una descrizione testuale dettagliata dei blocchi, collegandoli all'architettura mostrata in Figura 4.2.

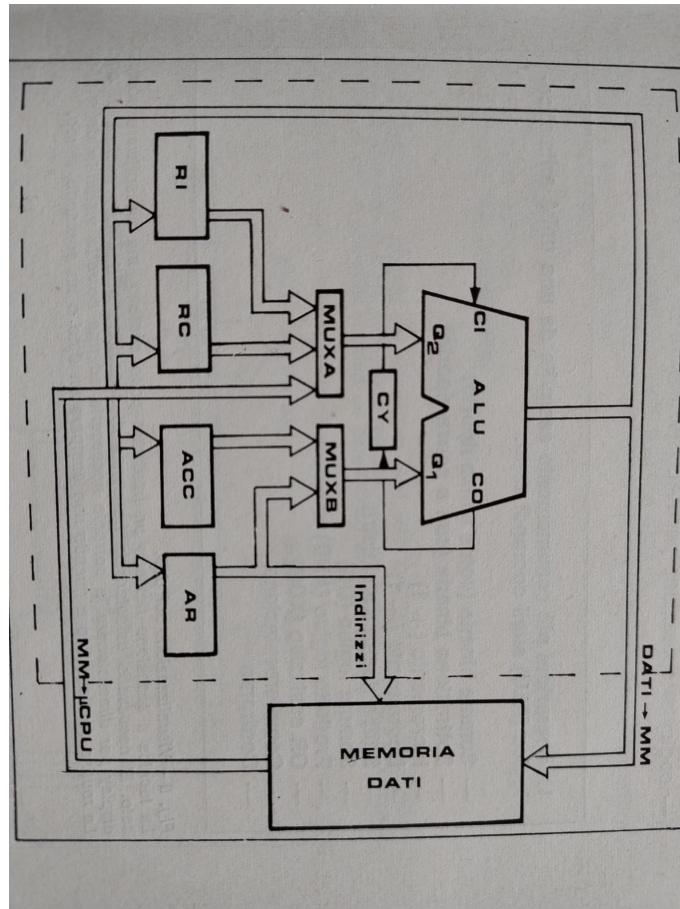


Figura 4.2: Schema dell'architettura con i registri e l'ALU.

### 4.3.1 Caricamento e Inizializzazione

1. **Carica la costante S in RC:** la costante S rappresenta il numero di bit dei due operandi (supponendo siano lunghi uguali). In RC memorizziamo tale valore, che useremo per sapere quante iterazioni fare.
2. **Azzera il carry:** poniamo a zero il registro CY, in quanto inizialmente non abbiamo nessun riporto in ingresso.
3. **Azzera il registro RI:** RI viene impostato a 0 per indicare che non abbiamo ancora elaborato alcun bit.
4. **Punta AR al 1º operando:** il registro AR (Address Register) viene impostato all'indirizzo di memoria dove inizia il primo operando.

### 4.3.2 Lettura del Primo Operando

1. **Trasferisci il bit in ACC:** leggiamo dalla memoria (all'indirizzo puntato da AR) il bit corrente del primo operando e lo carichiamo nell'accumulatore ACC.

### 4.3.3 Lettura del Secondo Operando

1. **Punta AR al 2º operando:** aggiorniamo AR in modo che punti all'indirizzo di memoria dove è memorizzato il bit corrispondente del secondo operando.
2. **Somma in ALU con CY:** l'ALU somma il contenuto di ACC, il bit del secondo operando (letto in memoria) e il **carry CY**. Il risultato (bit di somma) viene scritto in ACC e CY viene aggiornato se c'è un riporto (cioè se la somma dei bit è 2 o 3 in decimale, corrispondente a 10 o 11 in binario).

### 4.3.4 Memorizzazione del Risultato Parziale

1. **Punta AR alla zona di memoria per il risultato:** impostiamo AR sull'indirizzo di memoria dove vogliamo salvare il bit di risultato.
2. **Trasferisci il risultato in memoria:** il contenuto di ACC (bit di risultato) viene scritto nella cella di memoria puntata da AR.

### 4.3.5 Aggiornamento Indice e Verifica di Fine

1. **Incrementa RI:** il registro indice RI viene incrementato di 1, perché abbiamo elaborato un bit.
2. **Confronta RI con RC:** se RI è uguale a RC, significa che abbiamo terminato di sommare tutti i bit previsti. Altrimenti, se  $RI < RC$ , ripetiamo il ciclo di somma sul bit successivo.

### 4.3.6 Gestione del carry Finale

Se, dopo aver elaborato l'ultimo bit, CY risulta ancora a 1, significa che c'è un riporto finale. In tal caso, si deve:

- Allocare una cella di memoria aggiuntiva per questo bit di riporto;
- Scrivere il valore di CY in tale cella, in modo da ottenere la corretta somma finale (che avrà un bit in più).

## 4.4 Analisi dell'Algoritmo come Automa a Stati Finiti

L'algoritmo di somma può essere visto come un **automa a stati finiti** (FSM, Finite State Machine), dove ciascun blocco del diagramma di flusso corrisponde a uno *stato* o a una *transizione* tra stati.

### 4.4.1 Stati e Transizioni

- **Stato di inizializzazione:** l'automa si trova nello stato in cui RC, RI e CY vengono inizializzati. Al termine di questa fase, si passa allo stato successivo.
- **Stato di lettura primo operando:** si legge il bit dal primo operando, si carica in ACC.

- **Stato di lettura secondo operando e somma:** si legge il bit dal secondo operando e lo si somma con ACC e CY.
- **Stato di memorizzazione:** si scrive il risultato in memoria.
- **Stato di incremento e verifica:** si incrementa RI, si verifica se RI = RC.

Ciascuno stato ha delle *condizioni di uscita* che determinano se passare allo stato successivo o terminare (in caso di RI = RC e gestione dell'eventuale **carry** finale).

#### 4.4.2 Osservazioni

Questo modello di calcolo è sequenziale e *limitato* dal numero di stati previsti. Un FSM puro, infatti, ha un numero finito di stati e non dispone di memoria ausiliaria arbitraria. Qui, la memoria per la gestione dei dati è esterna (la RAM), ma la sequenza di controllo dell'algoritmo si può comunque mappare su un insieme finito di stati.

### 4.5 Collegamento con Automi a Pila e Macchina di Turing

#### 4.5.1 Automa a Pila (PDA)

Un **automa a pila** è un automa a stati finiti dotato di una memoria a pila. Se consideriamo la *pila* come un meccanismo per gestire il **carry** o per immagazzinare i bit dei risultati intermedi, potremmo *estendere* l'algoritmo in modo da supportare, ad esempio, la somma di numeri rappresentati in modo più complesso (o con struttura ricorsiva). Tuttavia, per l'algoritmo di somma di base, la memoria a pila non è strettamente necessaria: la gestione del **carry** avviene tramite un singolo bit dedicato e la memorizzazione dei bit avviene in RAM.

#### 4.5.2 Macchina di Turing (MdT)

Una **Macchina di Turing** è un modello di calcolo più potente, che può simulare qualsiasi algoritmo. L'algoritmo di somma di due numeri può essere realizzato facilmente da una MdT, la quale sposterebbe la testina su un nastro di memoria, leggendo e scrivendo bit, eventualmente tenendo traccia del **carry** in uno stato interno.

In un certo senso, l'architettura di un calcolatore reale (con registri, ALU, memoria, bus, ecc.) può essere vista come un'implementazione pratica di una Macchina di Turing limitata dalla dimensione finita della memoria. Il nostro algoritmo di somma è un *caso particolare* di operazione aritmetica, ma dimostra bene la natura sequenziale, a stati e basata sulla memoria, tipica di ogni calcolatore.

### 4.6 Dettagli Implementativi

#### 4.6.1 Modalità di Indirizzamento

Nell'architettura mostrata, il registro AR punta di volta in volta a:

1. Celle del primo operando;

2. Celle del secondo operando;
3. Celle del risultato.

Ogni volta che passiamo dal primo al secondo operando, cambiamo il valore di **AR**. Analogamente, quando vogliamo scrivere il risultato, impostiamo **AR** sull'area di memoria destinata al risultato.

#### 4.6.2 Ruolo del Registro Indice RI

**RI** funge da contatore del numero di bit elaborati. In molte architetture reali, **RI** potrebbe essere incrementato automaticamente dopo ogni accesso in memoria (*auto-increment*), oppure potremmo avere istruzioni dedicate per incrementare **RI**. In ogni caso, **RI** viene confrontato con **RC** (che rappresenta il numero totale di bit) per stabilire il *punto di uscita* dal ciclo di somma.

#### 4.6.3 Gestione del carry

Il **carry** (**CY**) è un singolo bit che può assumere valore 0 o 1. Nelle architetture binarie, ciò accade quando la somma dei bit in ingresso supera la capacità di 1 bit. In particolare:

- $0 + 0 + \text{carry}(0) = 0$  (nessun riporto)
- $1 + 0 + \text{carry}(0) = 1$  (nessun riporto)
- $1 + 1 + \text{carry}(0) = 0$  (con riporto = 1)
- $1 + 1 + \text{carry}(1) = 1$  (con riporto = 1)

Se stiamo lavorando in binario, questa logica è integrata nell'**ALU**. Il **carry** aggiornato viene poi conservato per l'iterazione successiva.

### 4.7 Esempio di Funzionamento in Binario

Supponiamo di voler sommare i numeri (in binario) 1100 (che corrisponde a 12 in decimale) e 1011 (che corrisponde a 11 in decimale). Entrambi hanno 4 bit, quindi **S** = 4. L'ordine di elaborazione dei bit, nel caso classico, parte dal *meno significativo* (a destra) verso il *più significativo* (a sinistra). Pertanto, i bit saranno gestiti nel seguente ordine:

Bit Index	3 (MSB)	2	1	0 (LSB)
1100 (A)	1	1	0	0
1011 (B)	1	0	1	1

Vediamo passo passo:

#### Bit LSB (Indice 0)

1.  $\text{RI} \leftarrow 0$ ,  $\text{CY} \leftarrow 0$ .
2. Leggiamo bit 0 di A: 0. Carichiamo in **ACC**.
3. Leggiamo bit 0 di B: 1. Sommiamo **ACC** (0), B (1) e **CY** (0) = 1, **CY** = 0.
4. Scriviamo 1 in memoria come bit 0 del risultato.
5.  $\text{RI} \leftarrow 1$ .  $\text{RI} < \text{RC}$  ( $1 < 4$ )  $\rightarrow$  continuiamo.

## Bit di indice 1

1. Leggiamo bit 1 di A: 0.  $\text{ACC} = 0$ .
2. Leggiamo bit 1 di B: 1. Sommiamo  $\text{ACC}$  (0), B (1) e  $\text{CY}$  (0) = 1,  $\text{CY} = 0$ .
3. Scriviamo 1 in memoria come bit 1 del risultato.
4.  $\text{RI} \leftarrow 2$ .  $\text{RI} < \text{RC}$  ( $2 < 4$ )  $\rightarrow$  continuiamo.

## Bit di indice 2

1. Leggiamo bit 2 di A: 1.  $\text{ACC} = 1$ .
2. Leggiamo bit 2 di B: 0. Sommiamo  $\text{ACC}$  (1), B (0) e  $\text{CY}$  (0) = 1,  $\text{CY} = 0$ .
3. Scriviamo 1 in memoria come bit 2 del risultato.
4.  $\text{RI} \leftarrow 3$ .  $\text{RI} < \text{RC}$  ( $3 < 4$ )  $\rightarrow$  continuiamo.

## Bit di indice 3 (MSB)

1. Leggiamo bit 3 di A: 1.  $\text{ACC} = 1$ .
2. Leggiamo bit 3 di B: 1. Sommiamo  $\text{ACC}$  (1), B (1) e  $\text{CY}$  (0).  
La somma binaria di  $1 + 1 = 10$  (in binario), quindi il risultato nel  $\text{ACC}$  sarà 0, e  $\text{CY} = 1$ .
3. Scriviamo 0 in memoria come bit 3 del risultato.
4.  $\text{RI} \leftarrow 4$ . Ora  $\text{RI} = \text{RC}$  ( $4 = 4$ ).

A questo punto, abbiamo  $\text{CY} = 1$ , il che indica un riporto *finale*. Poiché  $\text{RI} = \text{RC}$ , controlliamo se  $\text{CY} = 1$ . Se sì, scriviamo questo bit in una nuova cella di memoria (bit di indice 4 del risultato).

## Bit di riporto finale

1. Scriviamo  $\text{CY}$  (1) nella cella di memoria successiva (indice 4).

Ricostruendo il risultato dal bit meno significativo a quello più significativo, otteniamo:

$$\text{Risultato} = \underbrace{1}_{\text{Carry finale}} \underbrace{0}_{\text{bit 3}} \underbrace{1}_{\text{bit 2}} \underbrace{1}_{\text{bit 1}} \underbrace{1}_{\text{bit 0}} = 10111_2$$

che in decimale corrisponde a 23 (infatti  $12 + 11 = 23$ ).

## 4.8 Conclusioni

L'algoritmo di somma descritto è un tipico esempio di come, all'interno di un calcolatore, si possa implementare un'operazione aritmetica di base in **binario**. L'uso di registri specifici (**AR**, **RI**, **RC**, **ACC**, **CY**) e la gestione sequenziale dell'accesso alla memoria mostrano la natura a *stati finiti* del controllo del processore, mentre la memoria dati fornisce lo spazio per operandi e risultati.

Collegando il tutto alla teoria degli automi, notiamo che:

- Come FSM, l'algoritmo è descritto da un numero finito di stati e transizioni (inizializzazione, lettura, somma, scrittura, incremento indice, ecc.).
- Non è strettamente necessario un automa a pila (PDA) per realizzare questa operazione, sebbene una pila possa essere introdotta per estensioni più complesse (ad esempio, per gestire forme di notazione più articolate).
- Qualsiasi architettura di calcolatore, nella sua interezza, può essere considerata una Macchina di Turing finita (a causa dei limiti di memoria fisica), ma *equivalente* a una MdT *ideale* dal punto di vista computazionale, almeno finché non si esauriscono le risorse di memoria.

L'implementazione in linguaggio macchina (o microcodice) di questo algoritmo segue fedelmente il diagramma di flusso. Ciascuna istruzione **move**, **add**, **store** corrisponde a uno dei blocchi indicati. Il **carry** viene gestito da segnali di controllo interni all'ALU. Infine, l'architettura può essere generalizzata per altre operazioni (sottrazione, moltiplicazione, divisione) adottando schemi simili e integrando i necessari segnali di controllo nell'ALU.

# Capitolo 5

## Collaborazione Creativi e Tecnici: Un Progetto Interdisciplinare

### 5.1 Introduzione al Progetto

L'obiettivo di questo progetto è stimolare la collaborazione tra studenti con competenze creative e tecniche, unendo il mondo artistico a quello dell'ingegneria. Il punto di partenza è un semplice gioco implementato con Arduino, che utilizza pulsanti e LED per creare una sequenza di stati basati su un automa a stati finiti.

Il progetto si evolve trasformando questo prototipo in un'esperienza arricchita, dove ogni squadra è chiamata a contribuire in base alle proprie capacità:

- Il **Team Creativo** immagina come rendere il gioco più accessibile, originale e coinvolgente, progettando controller, decorazioni e contesti narrativi.
- Il **Team Tecnico** traduce queste idee in soluzioni pratiche, realizzando i controller e migliorando il sistema Arduino per implementare le modifiche richieste.

La sfida è creare un gioco che sia divertente, esteticamente accattivante e tecnicamente funzionale, favorendo al contempo la comunicazione e la collaborazione tra i due team.

### 5.2 Guida per il Team Creativo

I creativi hanno il compito di trasformare il progetto da un semplice prototipo tecnico a un'esperienza visiva e interattiva unica. Per fare ciò, possono seguire questi passaggi:

#### 5.2.1 Brainstorming delle Idee

Iniziate immaginando come migliorare il gioco:

- **Controller originale:** Pensate a un design che i giocatori possano utilizzare facilmente, come:
  - Controller da tenere in mano con una forma ergonomica e colori vivaci.
  - Pedali per controllare il gioco con i piedi.
  - Oggetti interattivi che si attivano toccandoli o muovendoli.

- **Elementi visivi:** Create decorazioni che rappresentino la storia del gioco, come un'ambientazione spaziale o un cruscotto di comando futuristico.
- **Esperienza sociale:** Progettate il gioco in modo che i giocatori interagiscano fisicamente o socialmente tra loro.

### 5.2.2 Creazione di Schizzi e Prototipi

Dopo aver definito le idee, rappresentatele visivamente:

- **Disegni:** Usate carta, matite colorate o software come Canva o TinkerCAD per creare rappresentazioni delle vostre idee.
- **Prototipi fisici:** Costruite modelli con materiali semplici come carta, cartone, carta pesta o plastica riciclata.
- **Moodboard:** Realizzate un collage di immagini che trasmetta l'estetica e l'atmosfera del progetto.

### 5.2.3 Presentazione delle Idee

Organizzate una presentazione per condividere le vostre idee con il Team Tecnico. Assicuratevi di includere:

- Schizzi o prototipi.
- Una spiegazione chiara delle funzionalità e dell'estetica proposte.
- Eventuali sfide tecniche che immaginate e come potrebbero essere affrontate.



Figura 5.1: Un esempio di creatività nel design del gioco.

## 5.3 Guida per il Team Tecnico

I tecnici hanno il compito di implementare le idee del Team Creativo utilizzando soluzioni pratiche ed efficaci. Questo richiede la conoscenza di alcuni componenti chiave e la capacità di integrarli nel sistema Arduino.

### 5.3.1 Componenti Principali

- **Sensori:** Dispositivi che convertono un fenomeno fisico (pressione, movimento, tocco) in segnali elettronici. Esempi:
  - **Sensore di pressione (FSR):** Rileva la forza esercitata su una superficie, ideale per pedali.
  - **Sensore touch capacitivo:** Rileva il tocco su una superficie conduttriva, utile per controller tattili.
- **Accelerometri:** Dispositivi che misurano l'accelerazione o il movimento. Possono essere usati per controlli basati sul movimento del corpo (ad esempio, inclinazione o rotazione).
- **LED RGB:** LED che possono emettere una combinazione di colori. Utilizzati per creare effetti visivi accattivanti.
- **Pulsanti arcade:** Pulsanti robusti e colorati, ideali per controller manuali.

### 5.3.2 Realizzazione Tecnica

- **Prototipazione su breadboard:** Utilizzate una breadboard per testare i componenti prima di realizzare un circuito definitivo.
- **Saldatura e cablaggio:** Dopo i test, assemblate i componenti su un circuito stampato o un supporto stabile.
- **Programmazione:** Scrivete il codice Arduino per gestire i nuovi input e output, implementando le modifiche richieste.

### 5.3.3 Debug e Ottimizzazione

Assicuratevi che il sistema funzioni correttamente:

- Testate ogni componente individualmente prima di integrarli.
- Monitorate il comportamento del sistema utilizzando la comunicazione seriale di Arduino.
- Collaborate con il Team Creativo per apportare modifiche in base al feedback.

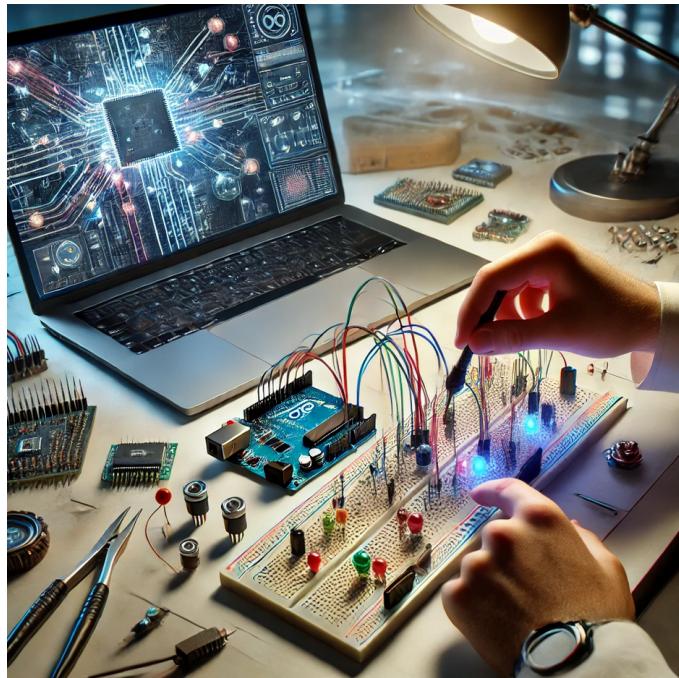


Figura 5.2: Un esempio di lavoro tecnico per implementare le idee del progetto.

### 5.3.4 Presentazione del Progetto

Documentate il processo tecnico e preparate una presentazione che mostri:

- I componenti scelti e come sono stati integrati.
- Le sfide incontrate e come sono state risolte.
- Il risultato finale e come soddisfa i requisiti del Team Creativo.

## 5.4 Progettazione e Creazione del Package

### 5.4.1 Design del Package

Per racchiudere e proteggere il sistema, il package deve essere progettato considerando sia l'estetica che la funzionalità. Ecco alcune linee guida:

- **Materiali:** Scegliete materiali leggeri ma resistenti come plastica riciclata, legno o acrilico.
- **Forma:** Il design deve essere ergonomico e funzionale, consentendo ai giocatori di interagire comodamente con il sistema.
- **Decorazioni:** Collaborate con il Team Creativo per integrare decorazioni che riflettono il tema del gioco.

### 5.4.2 Processo di Costruzione

- **Prototipazione:** Costruite un modello in scala con materiali economici come cartone o foamboard per testare il design.

- **Taglio e Assemblaggio:** Utilizzate strumenti come taglierine laser o stampanti 3D per realizzare il package definitivo.
- **Integrazione dei Componenti:** Alloggiate saldamente i componenti elettronici nel package, assicurandovi che siano facilmente accessibili per manutenzione o aggiornamenti.

### 5.4.3 Test del Package

- Verificate che il package protegga adeguatamente i componenti elettronici.
- Assicuratevi che il package sia comodo da usare per i giocatori.
- Fate test di durabilità per garantire che il package resista a un uso prolungato.

## 5.5 Componenti del Gioco e il loro Funzionamento

In questa sezione descriviamo i principali componenti elettronici utilizzati per il gioco e approfondiamo i concetti fisici fondamentali che ne regolano il funzionamento.

### 5.5.1 Legge di Ohm e Legge di Joule

Prima di analizzare i componenti, è essenziale comprendere due leggi fondamentali dell'elettricità:

1. **Legge di Ohm:** Descrive la relazione tra tensione ( $V$ ), corrente ( $I$ ) e resistenza ( $R$ ):

$$V = I \cdot R$$

2. **Legge di Joule:** Descrive la potenza dissipata sotto forma di calore in un resistore:

$$P = I^2 \cdot R$$

dove  $P$  è la potenza (in watt),  $I$  la corrente (in ampere) e  $R$  la resistenza (in ohm).

Queste leggi sono fondamentali per dimensionare correttamente le resistenze e garantire che i LED non vengano danneggiati da una corrente eccessiva.

### 5.5.2 LED e Semiconduttori

**Principio di Funzionamento:** I LED (*Light Emitting Diodes*) sono dispositivi semiconduttori che emettono luce quando attraversati da corrente elettrica in polarizzazione diretta.

#### Cosa sono i semiconduttori?

- I semiconduttori (es. silicio e arsenico di gallio) hanno una conducibilità intermedia tra conduttori e isolanti.
- La loro conducibilità può essere modificata aggiungendo impurità (*drogaggio*), creando zone di tipo  $p$  (ricche di lacune) e di tipo  $n$  (ricche di elettroni).

**Emissione di luce senza effetto Joule:** Quando un elettrone si ricombina con una lacuna in una giunzione *p-n*, rilascia energia sotto forma di un fotone (luce). Questa energia è determinata dalla banda proibita ( $E_g$ ) del materiale semiconduttore:

$$E = h \cdot f$$

dove:

- $E$  è l'energia del fotone.
- $h$  è la costante di Planck ( $6.63 \cdot 10^{-34} \text{ J}\cdot\text{s}$ ).
- $f$  è la frequenza della luce emessa.

A differenza delle resistenze, l'emissione di luce nei LED non avviene tramite l'effetto Joule, ma tramite transizioni elettroniche, rendendoli efficienti dal punto di vista energetico.

---

### 5.5.3 Altri Componenti e Collegamenti

#### Resistenze

**Calcolo della resistenza per un LED:** Per evitare sovraccorrenti nei LED, è necessario calcolare il valore della resistenza:

$$R = \frac{V_{\text{alimentazione}} - V_{\text{LED}}}{I_{\text{LED}}}$$

dove:

- $V_{\text{alimentazione}}$  è la tensione fornita da Arduino (es. 5V).
  - $V_{\text{LED}}$  è la caduta di tensione tipica del LED (es. 2V per un LED rosso).
  - $I_{\text{LED}}$  è la corrente desiderata (tipicamente 20mA).
- 

#### Bottoni e Interruttori

**Funzionamento:** I bottoni momentanei chiudono il circuito solo quando premuti. Sono utilizzati come input digitali per Arduino.

**Problema dei segnali fluttuanti:** Senza una resistenza di pull-up o pull-down, il segnale del bottone può essere instabile, generando falsi input.

**Soluzione:** Usa una resistenza di pull-up (connessa a  $V_{cc}$ ) o pull-down (connessa a  $GND$ ) per stabilizzare il segnale:

$$R_{\text{pull-up/pull-down}} \approx 10 \text{ k}\Omega$$


---

#### Arduino Uno

##### Specifiche principali:

- Microcontrollore ATmega328.
- Tensione operativa: 5V.
- Ingressi digitali e analogici per il controllo di sensori e attuatori.

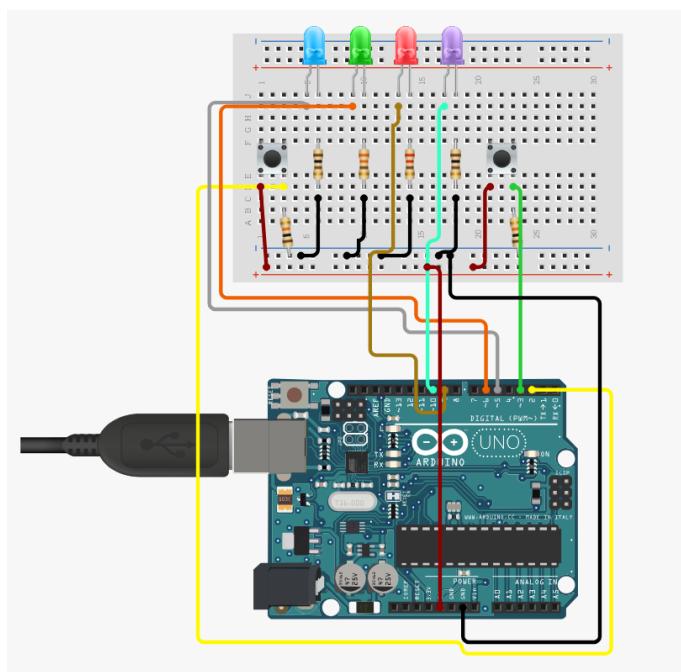


Figura 5.3: Diagramma dei collegamenti

# Capitolo 6

## Introduzione a Python e Simulazione del Gioco dei 4 Colori

### 6.1 Introduzione a Python

Python è un linguaggio di programmazione ad alto livello, ampiamente utilizzato per la sua sintassi semplice e leggibilità. È ideale per imparare i concetti base della programmazione e per creare applicazioni pratiche.

#### 6.1.1 La Sintassi Base di Python

In Python, il codice si organizza in blocchi identificati dall'indentazione. Ecco alcune strutture base:

##### Stampa e Input

Listing 6.1: Esempio di stampa e input

```
print("Ciao , mondo!") # Stampa un messaggio a schermo
nome = input("Come ti chiami? ") # Chiede un input all'utente
print("Benvenuto , " + nome + " !")
```

##### Condizioni e Cicli

Listing 6.2: Esempio di condizioni e cicli

```
# Esempio di condizione
numero = int(input("Inserisci un numero: "))
if numero > 0:
    print("Il numero è positivo .")
else:
    print("Il numero è zero o negativo .")

# Esempio di ciclo
for i in range(5): # Ripete per i valori da 0 a 4
    print("Iterazione: ", i)
```

## Funzioni

Listing 6.3: Esempio di funzione

```
# Una funzione che calcola il quadrato di un numero
def quadrato(x):
    return x * x

risultato = quadrato(4)
print("Il quadrato di 4 è:", risultato)
```

## 6.2 Python e gli Automi

Un programma in Python può essere considerato equivalente a un automa perché entrambi seguono regole precise di transizione da uno stato all'altro basandosi sugli input ricevuti.

### 6.2.1 Esempio di Automa in Python

Un automa a stati finiti per il gioco dei 4 colori può essere rappresentato in Python come segue:

Listing 6.4: Esempio di automa a stati finiti in Python

```
# Definizione degli stati
stati = ["R", "G", "B", "W"]
stato_corrente = "R"

# Funzione per la transizione
def transizione(giocatore):
    global stato_corrente
    if stato_corrente == "R":
        stato_corrente = "G" if giocatore == 1 else "B"
    elif stato_corrente == "G":
        stato_corrente = "B" if giocatore == 1 else "R"
    elif stato_corrente == "B":
        stato_corrente = "W"
    elif stato_corrente == "W":
        stato_corrente = "R" if giocatore == 1 else "G"

# Simulazione
print("Stato iniziale:", stato_corrente)
transizione(1) # Giocatore 1 preme il tasto
print("Stato dopo Giocatore 1:", stato_corrente)
transizione(2) # Giocatore 2 preme il tasto
print("Stato dopo Giocatore 2:", stato_corrente)
```

## 6.3 Simulazione del Gioco dei 4 Colori

Portiamo il nostro gioco completo in Python. La sequenza di gioco è la seguente:

1. Il sistema genera una sequenza casuale di transizioni.
2. Mostra la sequenza ai giocatori.
3. I giocatori ripetono la sequenza premendo i tasti corretti.
4. Il sistema verifica se la sequenza dei giocatori è corretta.

### 6.3.1 Codice Completo in Python

Listing 6.5: Codice Python del gioco dei 4 colori

```

import random

# Sequenze di colori per ogni giocatore
sequenza_p1 = ["W", "R", "G", "B"]
sequenza_p2 = ["W", "G", "R", "B"]

# Stati del gioco
stati = ["R", "G", "B", "W"]

# Inizializzazione
stato_corrente = "W"
sequenza_azioni = [] # Sequenza generata dal sistema (P1, P2, ...)
sequenza_colorata = [] # Sequenza di colori da mostrare ai giocatori
input_giocatori = [] # Sequenza degli input dei giocatori
N = 5 # Lunghezza della sequenza

# Funzione per determinare il prossimo stato in base al giocatore
def transizione(stato, giocatore):
    if giocatore == 1:
        indice = sequenza_p1.index(stato)
        return sequenza_p1[(indice + 1) % len(sequenza_p1)]
    elif giocatore == 2:
        indice = sequenza_p2.index(stato)
        return sequenza_p2[(indice + 1) % len(sequenza_p2)]


# Genera una sequenza valida di azioni
print("Generazione della sequenza...")
for _ in range(N):
    giocatore = random.choice([1, 2])
    sequenza_azioni.append(giocatore)
    stato_corrente = transizione(stato_corrente, giocatore)
    sequenza_colorata.append(stato_corrente)

# Mostra la sequenza di colori ai giocatori
print("\nSequenza di colori mostrata:")
print("→".join(sequenza_colorata))

# Reset dello stato per l'input dei giocatori
stato_corrente = "W"

```

```
# Input dei giocatori
print("\nInizia il gioco! Premi 1 (Player 1) o 2 (Player 2) per replicare la s
for i in range(N):
    giocatore = int(input(f"Input giocatore {i+1}/{N}: "))
    input_giocatori.append(giocatore)
    stato_corrente = transizione(stato_corrente, giocatore)
    if stato_corrente != sequenza_colorata[i]:
        print("Game Over! Sequenza errata.")
        break
else:
    print("Congratulazioni! Avete riprodotto la sequenza correttamente.")
```

## 6.4 Conclusione

Questa introduzione a Python mostra come il linguaggio possa essere utilizzato per simulare automi e implementare il gioco dei 4 colori. Gli studenti possono facilmente espandere il programma aggiungendo nuove funzionalità o modificando le regole del gioco.

# Capitolo 7

## Prototipazione con cartapesta

### 7.1 Introduzione

Nel corso del progetto, gli studenti hanno sviluppato tre diverse versioni di una console di gioco interattiva, combinando competenze artistiche e tecniche. La progettazione ha coinvolto sia l'aspetto estetico e di usabilità (curato dagli studenti dell'indirizzo artistico) sia la realizzazione elettronica e programmabile con Arduino (a cura degli studenti dell'indirizzo tecnico).

### 7.2 Prototipi sviluppati

#### 7.2.1 Console classica

Questo prototipo è il più vicino ai videogiochi tradizionali, con due joystick e pulsanti colorati. Gli studenti hanno progettato e realizzato il case in cartapesta, garantendo un design ergonomico per un'interazione naturale con i controlli.



Figura 7.1: Prototipo della console classica, con pulsanti e joystick separati.

### 7.2.2 Nuvola con bottoni integrati

Questa versione presenta un design più astratto e giocoso, con i pulsanti incorporati direttamente nella superficie della console. Gli studenti dell'indirizzo artistico hanno creato il case in cartapesta, mentre il team tecnico ha integrato i circuiti e i pulsanti per la corretta interazione.



Figura 7.2: Prototipo "nuvola", con pulsanti integrati nel design della console.

### 7.2.3 Cocomero

L'ultima versione della console richiama la forma di un cocomero, combinando elementi visivi accattivanti con la funzionalità interattiva. Gli studenti hanno lavorato sulla colorazione e sulla disposizione dei pulsanti, creando un design coinvolgente.



Figura 7.3: Prototipo "cocomero", un mix tra estetica e funzionalità di gioco.

### 7.3 Processo di realizzazione

Il lavoro si è articolato in diverse fasi:

- **Progettazione:** gli studenti hanno ideato il design e la disposizione dei pulsanti.
- **Costruzione in cartapesta:** sono stati realizzati i case con materiali modellabili e decorati a mano.
- **Integrazione hardware:** i circuiti con Arduino sono stati collegati ai pulsanti e ai LED, testando la funzionalità di gioco.
- **Programmazione:** il software su Arduino gestisce gli input degli utenti e la sequenza del gioco.

### 7.4 Conclusioni

Il progetto ha consentito agli studenti di esplorare la sinergia tra design e tecnologia, sviluppando competenze pratiche nella costruzione di prototipi e nella programmazione. L'approccio multidisciplinare ha favorito la collaborazione tra studenti con background differenti, valorizzando sia la creatività che la risoluzione di problemi tecnici.

# Capitolo 8

## Realizzazione

### 8.1 Introduzione

Nel corso del progetto, gli studenti hanno sviluppato tre diverse versioni di una console di gioco interattiva, combinando competenze artistiche e tecniche. La progettazione ha coinvolto sia l'aspetto estetico e di usabilità (curato dagli studenti dell'indirizzo artistico) sia la realizzazione elettronica e programmabile con Arduino (a cura degli studenti dell'indirizzo tecnico).

### 8.2 Prototipi sviluppati

#### 8.2.1 Console classica

Questo prototipo è il più vicino ai videogiochi tradizionali, con due joystick e pulsanti colorati. Gli studenti hanno progettato e realizzato il case in cartapesta, garantendo un design ergonomico per un'interazione naturale con i controlli.

#### 8.2.2 Nuvola con buttoni integrati

Questa versione presenta un design più astratto e giocoso, con i pulsanti incorporati direttamente nella superficie della console. Gli studenti dell'indirizzo artistico hanno creato il case in cartapesta, mentre il team tecnico ha integrato i circuiti e i pulsanti per la corretta interazione.

#### 8.2.3 Cocomero

L'ultima versione della console richiama la forma di un cocomero, combinando elementi visivi accattivanti con la funzionalità interattiva. Gli studenti hanno lavorato sulla colorazione e sulla disposizione dei pulsanti, creando un design coinvolgente.

### 8.3 Processo di realizzazione

Il lavoro si è articolato in diverse fasi:

- **Progettazione:** gli studenti hanno ideato il design e la disposizione dei pulsanti.

- **Costruzione in cartapesta:** sono stati realizzati i case con materiali modellabili e decorati a mano.
- **Integrazione hardware:** i circuiti con Arduino sono stati collegati ai pulsanti e ai LED, testando la funzionalità di gioco.
- **Programmazione:** il software su Arduino gestisce gli input degli utenti e la sequenza del gioco.

## 8.4 Conclusioni

Il progetto ha consentito agli studenti di esplorare la sinergia tra design e tecnologia, sviluppando competenze pratiche nella costruzione di prototipi e nella programmazione. L'approccio multidisciplinare ha favorito la collaborazione tra studenti con background differenti, valorizzando sia la creatività che la risoluzione di problemi tecnici.

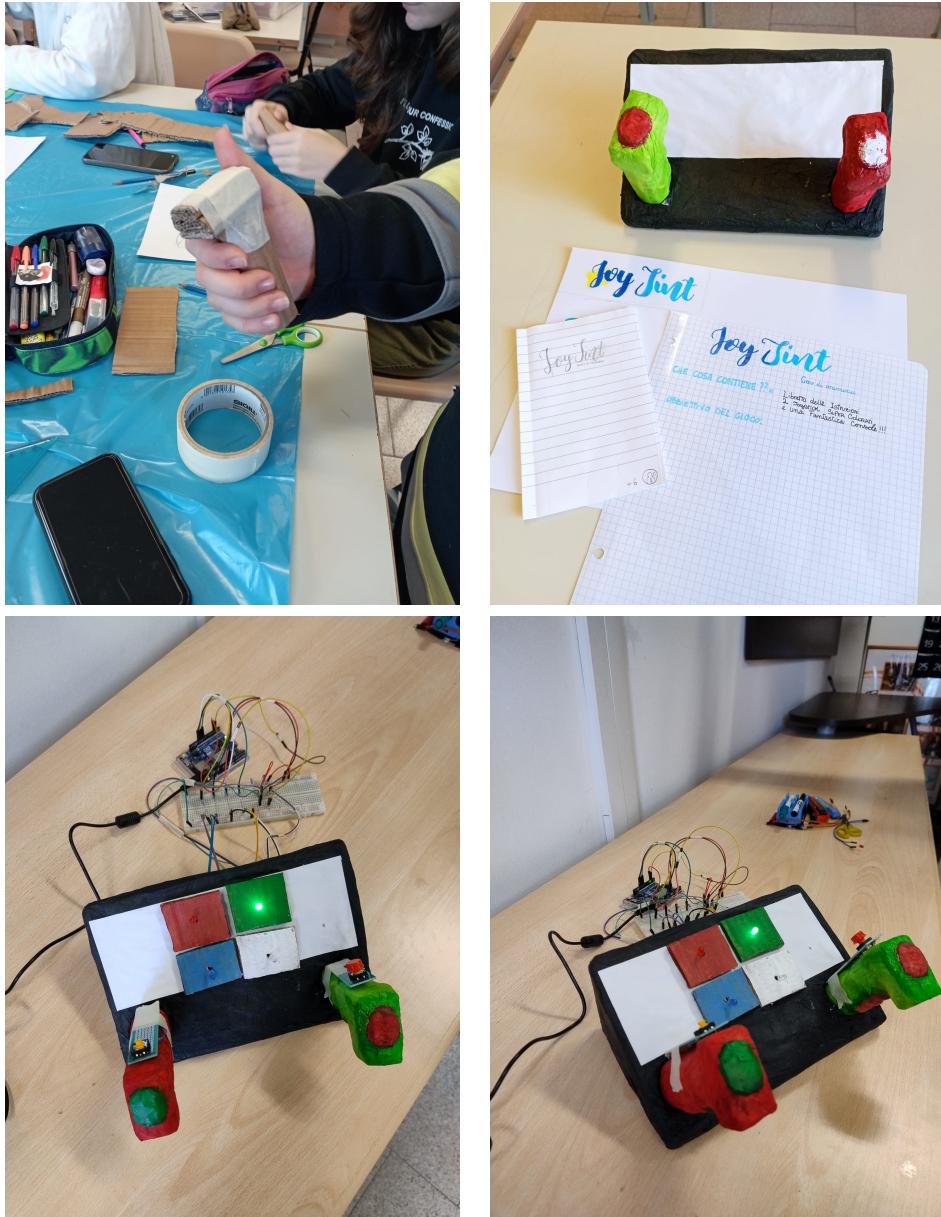


Figura 8.1: Prototipo della console classica, con pulsanti e joystick separati.



Figura 8.2: Prototipo "nuvola", con pulsanti integrati nel design della console.



Figura 8.3: Prototipo "cocomero", un mix tra estetica e funzionalità di gioco.