



1506  
UNIVERSITÀ  
DEGLI STUDI  
DI URBINO  
CARLO BO

DISPEA  
DIPARTIMENTO DI  
SCIENZE PURE E  
APPLICATE

---

# SOMMA DI QUADRATI

$$A^2 + B^2$$

---

PROGETTO D'ESAME DI RETI LOGICHE  
Anno Accademico 2024/2025

**Selvetti Simona**  
MATRICOLA: 336466

# Contents

<b>1 Specifica</b>	<b>3</b>
1.1 Scopo del progetto . . . . .	3
1.2 Specifica funzionale . . . . .	3
1.3 Specifica parametrica . . . . .	4
<b>2 Impostazione del progetto a livello RT</b>	<b>4</b>
2.1 Data Flow Graph . . . . .	5
2.2 Risorse . . . . .	5
<b>3 Progetto delle risorse a livello gate</b>	<b>6</b>
3.1 Porte logiche elementari . . . . .	7
3.1.1 myINV . . . . .	7
3.1.2 myNAND . . . . .	8
3.1.3 myNOR . . . . .	9
3.2 Porte logiche composte . . . . .	10
3.2.1 myAND . . . . .	10
3.2.2 myOR . . . . .	10
3.2.3 myEXOR . . . . .	11
3.3 Macro Aritmetiche . . . . .	12
3.3.1 myHA . . . . .	12
3.3.2 myFA . . . . .	13
3.3.3 MUL4x4 . . . . .	14
3.3.4 SQ4bit . . . . .	15
3.3.5 RCA8bit . . . . .	17
3.4 Registri . . . . .	18
3.4.1 LatchSR . . . . .	18
3.4.2 FFDls . . . . .	19
3.4.3 FFDDet . . . . .	20

3.4.4	REG4bit . . . . .	21
3.4.5	REG8bit . . . . .	22
3.5	Multiplexer . . . . .	23
3.5.1	MUX . . . . .	23
3.5.2	MUX4bit . . . . .	24
<b>4</b>	<b>Data Path</b>	<b>25</b>
4.1	Architettura A: Moltiplicatore generico . . . . .	25
4.2	Architettura B: Quadratore ottimizzato . . . . .	26
<b>5</b>	<b>Control Unit</b>	<b>27</b>
5.1	Specifica . . . . .	27
5.2	Implementazione della macchina di Moore . . . . .	28
5.2.1	Scheda Tecnica: myCU . . . . .	29
<b>6</b>	<b>Simulazione e analisi del progetto</b>	<b>30</b>
6.1	Verifica funzionale . . . . .	30
6.1.1	Prima simulazione . . . . .	30
6.1.2	Seconda simulazione . . . . .	31
6.1.3	Terza simulazione . . . . .	31
6.2	Valutazione di prestazioni e complessità . . . . .	32
6.2.1	Area . . . . .	32
6.2.2	Tempo di propagazione . . . . .	32
6.2.3	Tempo di contaminazione . . . . .	33
6.3	Conclusioni . . . . .	33

# 1 Specifica

## 1.1 Scopo del progetto

Lo scopo del progetto è la realizzazione di un circuito sequenziale in grado di calcolare la somma dei quadrati di due numeri interi positivi. Il progetto è stato sviluppato mediante il simulatore circuitale *TkGate 2.1* esplorando due differenti approcci architetturali con l'obiettivo di confrontarne le prestazioni:

- **Architettura A (Generica):** Utilizza un moltiplicatore generico  $4 \times 4$  bit progettato per eseguire il prodotto  $A \times B$ .
- **Architettura B (Specializzata):** Sostituisce il moltiplicatore generico con un modulo aritmetico dedicato, specializzato per la funzione  $A^2$ .

## 1.2 Specifica funzionale

L'espressione aritmetica che descrive il comportamento del sistema è:

$$f(A, B) = A^2 + B^2$$

- **Dominio (input):** I valori  $A$  e  $B$  sono numeri interi senza segno rappresentati su **4 bit** quindi:

$$D_{in} = [0, 2^4 - 1] = [0, 15]$$

- **Codominio (output):** Il risultato sarà sicuramente un numero intero positivo quindi rappresentabile senza segno. Calcoliamo i bit necessari:
  - (i) Il valore massimo dell'uscita si ottiene per  $A = 15$  e  $B = 15$
  - (ii) Il valore massimo del quadrato è  $15^2 = 225$ . Sappiamo che il quadrato di un numero a  $N$  bit richiede al massimo  $2N$ , quindi, nel nostro caso, 8 bit per ogni operando intermedio ( $2^8 - 1 = 255$ )
  - (iii) Il valore massimo del risultato finale è  $225 + 225 = 450$ .
  - (iv) Poiché  $2^8 = 256 < 450 < 512 = 2^9$ , sono necessari **9 bit** per rappresentare il risultato finale senza overflow.

Quindi:

$$D_{out} = [0, 2^9 - 1] = [0, 511]$$

### 1.3 Specifica parametrica

Poiché il sistema lavora con input a 4 bit, prenderemo in considerazione solo valori rappresentabili entro questo limite, scartando qualsiasi codifica che richieda una dimensione maggiore.

L'obiettivo principale di ottimizzazione è stato trovare il giusto compromesso tra la complessità del circuito (quanta Area occupa) e la latenza (quanto tempo impiega). Per ottenere questo bilanciamento, ci avvaliamo di due strategie fondamentali:

- **Resource Sharing:** L'idea è quella di "riciclare" le risorse hardware. Invece di duplicare i componenti per ogni operazione matematica, utilizziamo lo stesso blocco funzionale in momenti diversi dell'esecuzione. In questo modo possiamo ridurre notevolmente l'area occupata in cambio di un leggero aumento dei cicli di clock necessari.
- **Costruzione a livello micro (Gate Level):** Evitiamo di utilizzare macro funzionali predefinite se queste risultano sovradimensionate. Preferiamo costruire i componenti partendo dal livello logico più basso al fine di inserire solo le risorse strettamente necessarie.

## 2 Impostazione del progetto a livello RT

In questa fase della progettazione definiamo l'architettura del sistema a livello *Register Transfer* (RT). Traduciamo, quindi, la specifica funzionale in una struttura hardware concreta definendo come i dati si trasferiscono tra i registri e come vengono elaborati dalle unità funzionali.

## 2.1 Data Flow Graph

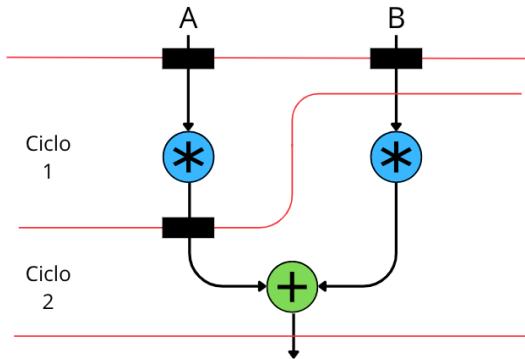


Figure 1: Data Flow Graph

Come illustrato nel grafico, abbiamo suddiviso l'esecuzione in **2 cicli di clock**:

- **Ciclo 1:** Il sistema seleziona il primo operando ( $A$ ), ne calcola il quadrato tramite la macro aritmetica principale (Moltiplicatore o Quadratore, a seconda dell'architettura) e memorizza il risultato parziale in un registro di appoggio.
- **Ciclo 2:** Il sistema seleziona il secondo operando ( $B$ ), ne calcola il quadrato utilizzando la medesima macro aritmetica (risorsa condivisa) e somma questo nuovo valore con il l'operando intermedio memorizzato nel ciclo precedente, ottenendo il risultato finale.

Questa scelta progettuale comporta l'impossibilità di sfruttare il *pipelining*, ma ci permette di dimezzare l'area dedicata al calcolo dei quadrati, utilizzando una singola unità funzionale invece di due in parallelo.

## 2.2 Risorse

Per implementare il Data Path descritto sopra e supportare entrambe le architetture, abbiamo identificato e sviluppato le seguenti risorse hardware:

- **Registri**
  - 2 Registri a 4 bit (REG4bit) per stabilizzare gli ingressi  $A$  e  $B$ .

- 1 Registro a **8 bit** (**REG8bit**) per memorizzare il risultato parziale ( $A^2$ ) tra il primo e il secondo ciclo di clock.

- **Multiplexer**

- 1 Multiplexer a **4 bit** (**MUX4bit**) posto a monte della risorsa condivisa. La sua funzione è selezionare quale operando ( $A$  o  $B$ ) inviare all’unità di calcolo in base al segnale di controllo fornito dalla Control Unit.

- **Macro Aritmetiche**

- 1 Moltiplicatore generico che prende due input a 4 bit (**MUL4x4**), utilizzato nell’*Archittettura A*.
- 1 Quadratore ottimizzato che prende un solo input a4 bit (**SQ4bit**), utilizzato nell’*Archittettura B*.
- 1 Addizionatore RCA a **8 bit** (**RCA8bit**) con output a 9 bit per eseguire la somma finale  $A^2 + B^2$ .

- **Control Unit**

- 1 Unità di Controllo (**myCU**) implementata come macchina a stati finiti (FSM) per generare i segnali di selezione del multiplexer e di abilitazione dei registri, sincronizzando le fasi del Resource Sharing.

### 3 Progetto delle risorse a livello gate

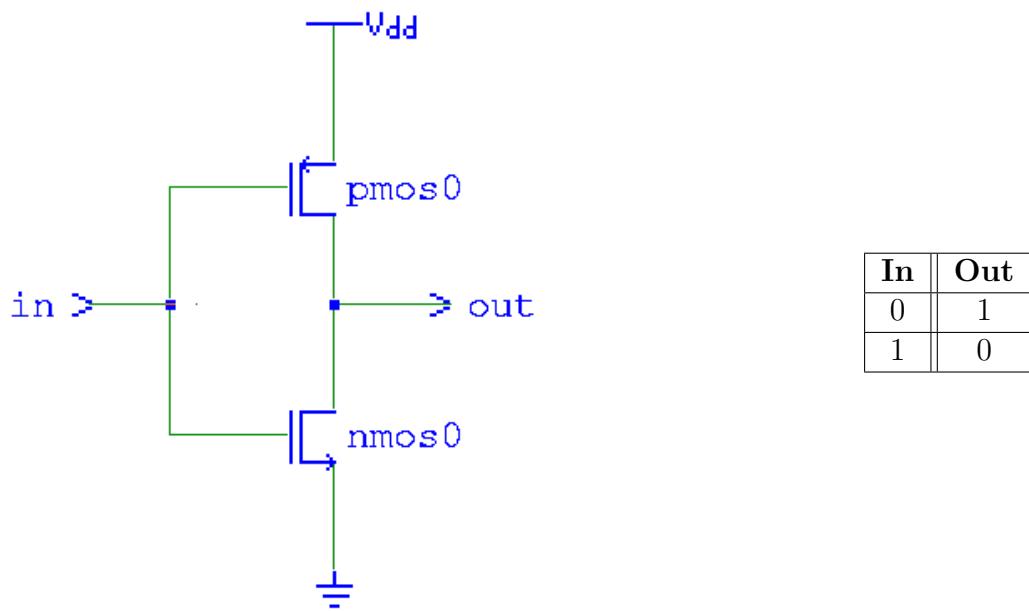
In questa sezione descriviamo l’implementazione di tutte le risorse hardware necessarie per realizzare il Data Path e la Control Unit definiti in precedenza. Abbiamo adottato una metodologia di progettazione *bottom-up*: partendo dalla tecnologia FCMOS (Fully Complementary Metal-Oxide Semiconductor), abbiamo prima realizzato le porte logiche elementari basate su transistor. Successivamente, utilizzando queste come mattoni fondamentali, abbiamo costruito le porte composte e infine le macro-funzioni complesse.

Questo approccio gerarchico ci permette di calcolare con precisione le metriche di Area, Tempo di Propagazione ( $T_p$ ) e Tempo di Contaminazione ( $T_c$ ) per l’intero sistema, sommando i contributi dei sottocomponenti.

### 3.1 Porte logiche elementari

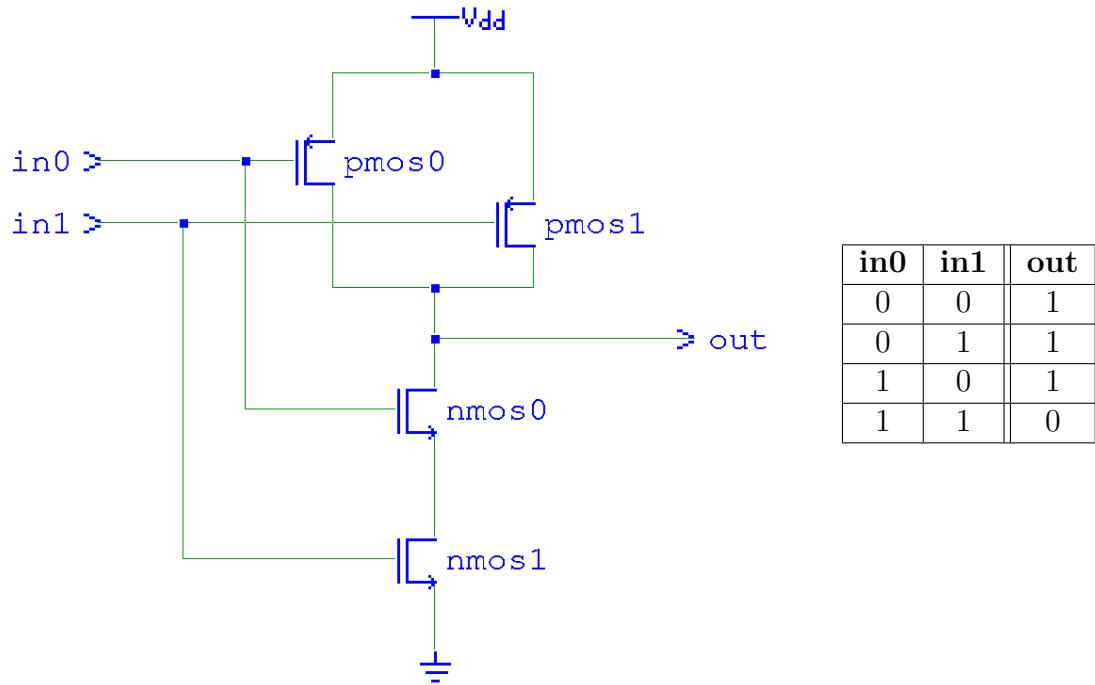
Definiamo come porte logiche elementari quei componenti composti solo da transistor. Siccome le consideriamo come "unità di base" avranno Ate  $T_p$  e  $T_c$  pari a 1.

#### 3.1.1 myINV



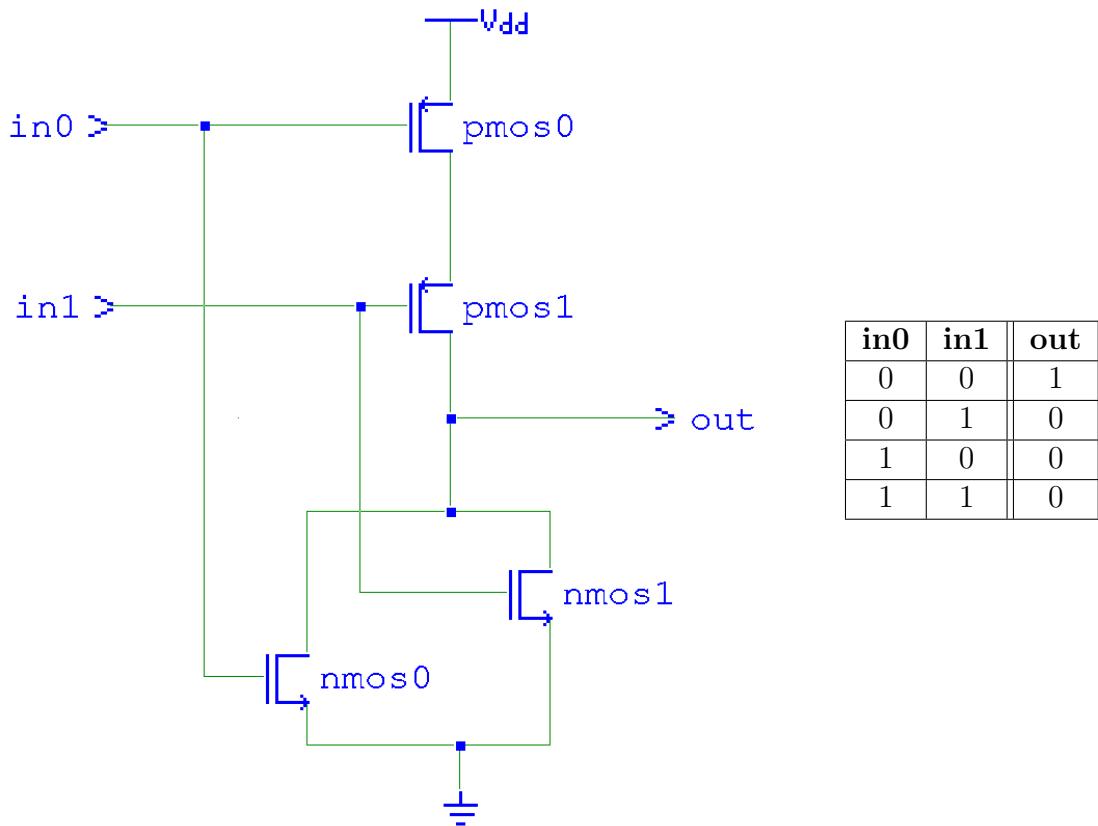
<b>Descrizione</b>	Implementa l'operatore logico <b>NOT</b> Inverte il segnale in ingresso.
<b>Composizione</b>	$1 \times PMOS$ $1 \times NMOS$
<b>Area (A)</b>	<b>1</b>
$T_p$	<b>1</b>
$T_c$	<b>1</b>

### 3.1.2 myNAND



<b>Descrizione</b>	Implementa la negazione dell'operatore logico <b>AND</b> Restituisce 0 solo quando entrambi gli input sono 1
<b>Composizione</b>	$2 \times PMOS$ collegati in parallelo $2 \times NMOS$ collegati in serie
<b>Area (A)</b>	1
$T_p$	1
$T_c$	1

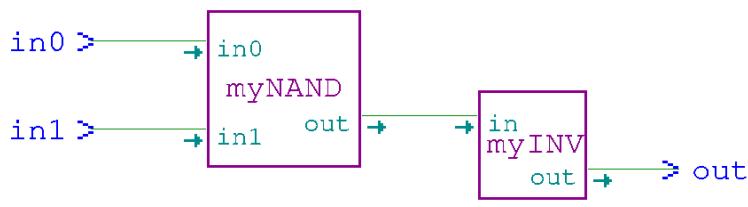
### 3.1.3 myNOR



<b>Descrizione</b>	Implementa la negazione dell'operatore logico <b>OR</b> Restituisce 1 solo quando entrambi gli input sono 0
<b>Composizione</b>	2 × <i>PMOS</i> collegati in serie 2 × <i>NMOS</i> collegati in parallelo
<b>Area (A)</b>	1
<b>T<sub>p</sub></b>	1
<b>T<sub>c</sub></b>	1

## 3.2 Porte logiche composte

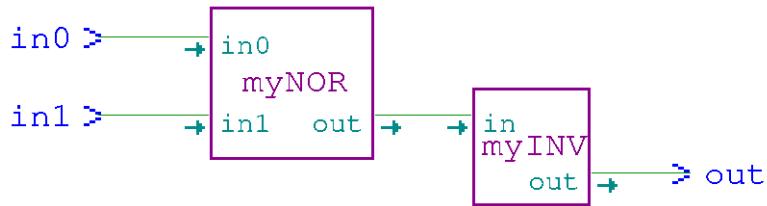
### 3.2.1 myAND



in0	in1	out
0	0	0
0	1	0
1	0	0
1	1	1

Descrizione	Implementa l'operatore logico AND
Composizione	$1 \times myNAND$ $1 \times myINV$
Area (A)	2
$T_p$	2
$T_c$	2

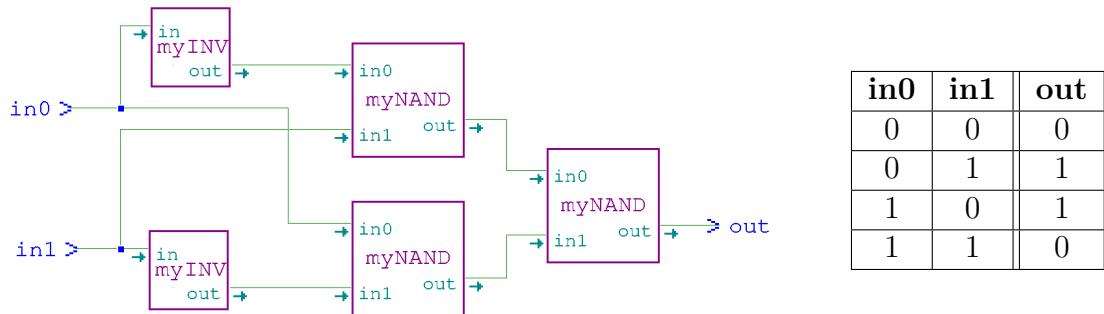
### 3.2.2 myOR



in0	in1	out
0	0	0
0	1	1
1	0	1
1	1	1

<b>Descrizione</b>	Implementa l'operatore logico <b>OR</b>
<b>Composizione</b>	$1 \times myNAND$ $1 \times myINV$
<b>Area (A)</b>	<b>2</b>
$T_p$	<b>2</b>
$T_c$	<b>2</b>

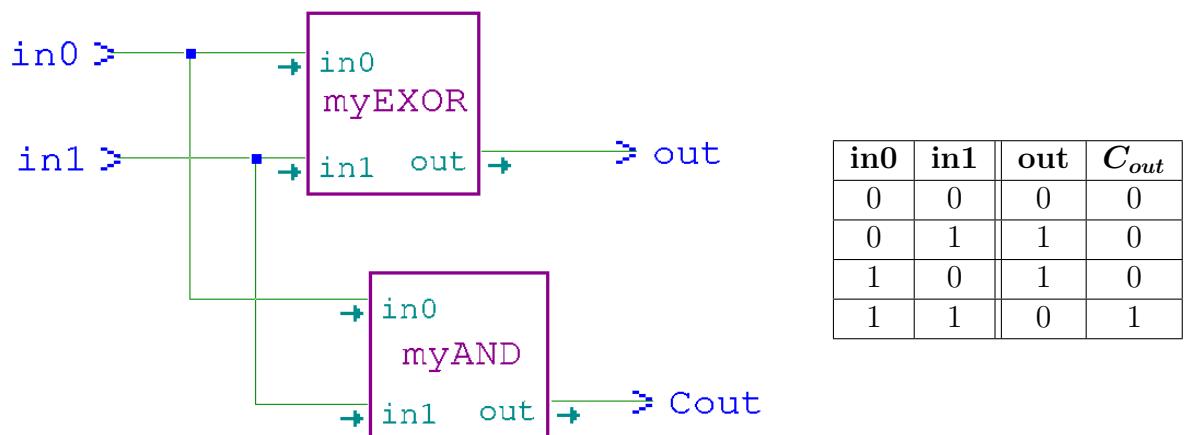
### 3.2.3 myEXOR



<b>Descrizione</b>	Implementa la funzione logica $A \oplus B$ , implementata fisicamente come $(\overline{A} \cdot \overline{B}) \cdot (\overline{\overline{A} \cdot B})$
<b>Composizione</b>	$3 \times myNAND$ $2 \times myINV$
<b>Area (A)</b>	<b>5</b>
$T_p$	<b>3</b>
$T_c$	<b>2</b>

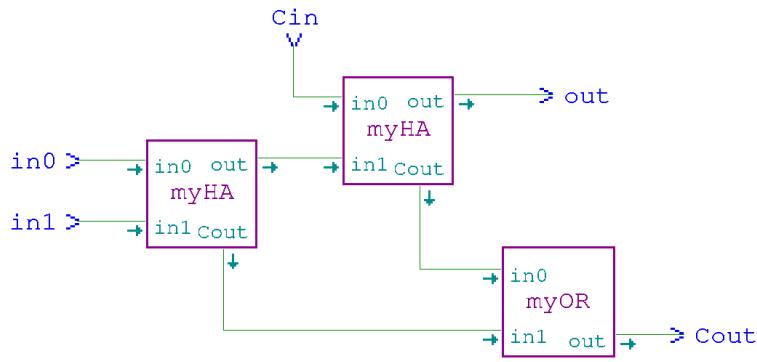
### 3.3 Macro Aritmetiche

#### 3.3.1 myHA



<b>Descrizione</b>	Somma due bit producendo un risultato e un eventuale riporto ma non gestisce un riporto in ingresso
<b>Composizione</b>	$1 \times myAND$ $1 \times myEXOR$
<b>Area (A)</b>	7
$T_p$	3
$T_c$	2

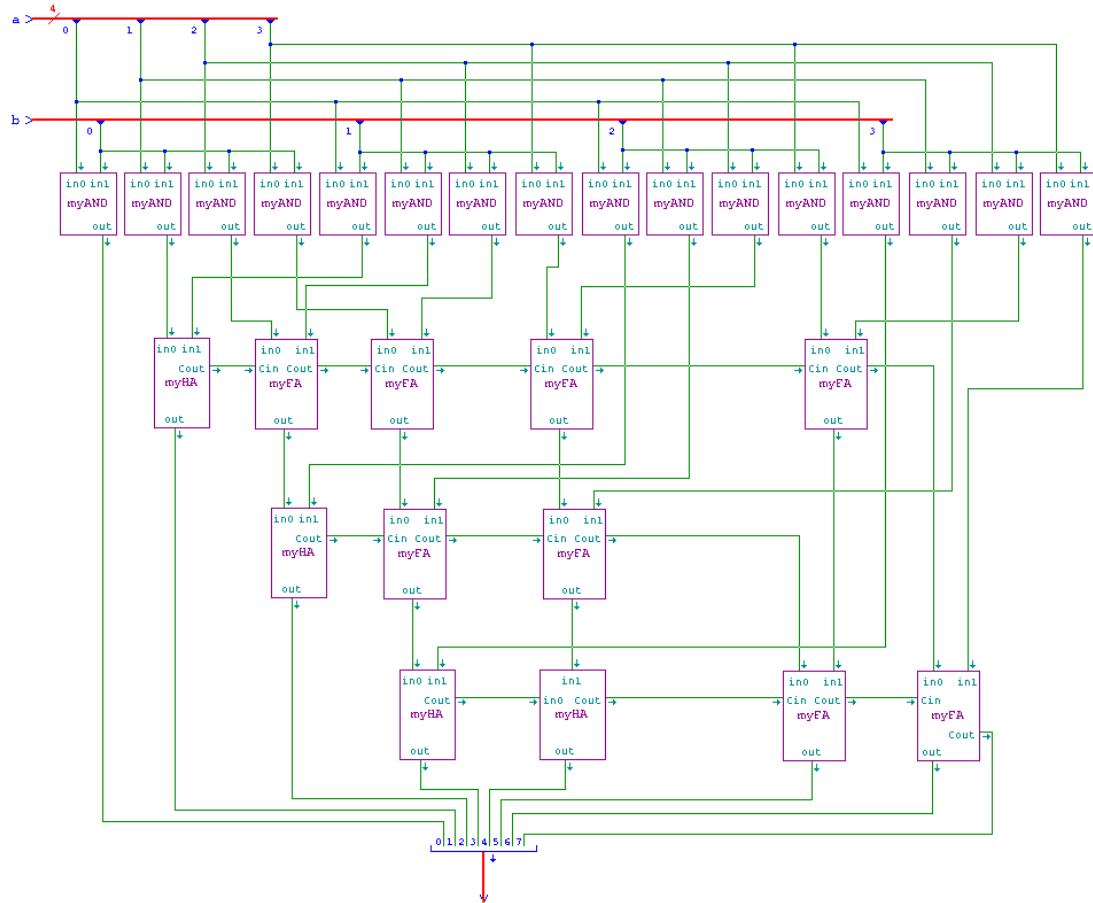
### 3.3.2 myFA



$in_0$	$in_1$	$C_{in}$	$out$	$C_{out}$
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

<b>Descrizione</b>	Somma 3 bit (due operandi e un riporto in ingresso) producendo un risultato e un eventuale riporto
<b>Composizione</b>	$2 \times myHA$ $1 \times myEXOR$
<b>Area (A)</b>	<b>16</b>
$T_p$	<b>8</b>
$T_c$	<b>4</b>

### 3.3.3 MUL4x4



<b>Descrizione</b>	Permette di moltiplicare tra loro due operandi, ciascuno composto da 4 bit. Sarà composto quindi da porte AND per generare i prodotti parziali e da una rete di Half Adder e Full Adder per sommarli
<b>Composizione</b>	$16 \times \text{myAND}$ $4 \times \text{myHA}$ $8 \times \text{myFA}$
<b>Area (A)</b>	<b>188</b>
<b><math>T_p</math></b>	<b>47</b>
<b><math>T_c</math></b>	<b>2</b>

### 3.3.4 SQ4bit

Con questa macro aritmetica vogliamo implementare la funzione

$$y = a^2$$

e invece di calcolare "ciecamente" tutti i prodotti parziali come abbiamo fatto nel generico moltiplicatore, abbiamo sfruttato due proprietà dell'algebra booleana:

- **Idempotenza:**  $x \cdot x = x$
- **Commutatività:**  $a \cdot b = b \cdot a$

Abbiamo osservato che:

$a_3$	$a_2$	$a_1$	$a_0$	$\times$
$a_3$	$a_2$	$a_1$	$a_0$	$=$
	$a_3a_0$	$a_2a_0$	$a_1a_0$	$a_0a_0$
	$a_3a_1$	$a_2a_1$	$a_1a_1$	$a_0a_1$
$a_3a_2$	$a_2a_2$	$a_1a_2$	$a_0a_2$	/ / +
$a_3a_3$	$a_2a_3$	$a_1a_3$	$a_0a_3$	/ / =
$a_3$	$2a_2a_3$	$a_2 + 2a_1a_3$	$2a_1a_2 + 2a_0a_3$	$a_1 + a_0a_2$
				$2a_0a_1$
				$a_0$

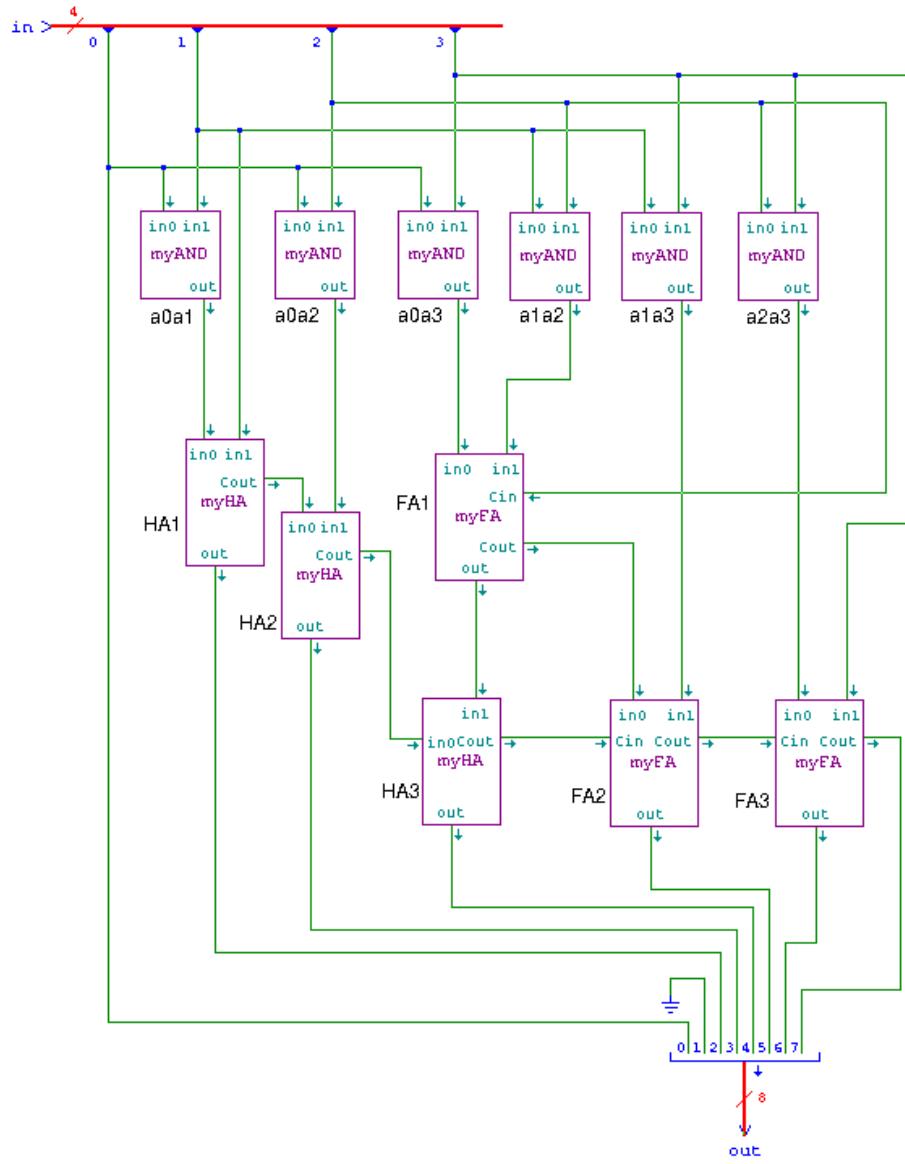
In binario, quando facciamo la moltiplicazione per 2 otteniamo l'operazione di **scorrimento a sinistra**, che a livello pratico possiamo tradurre con

$$2x = 0 \text{ riporto di } x$$

Quindi otteniamo che:

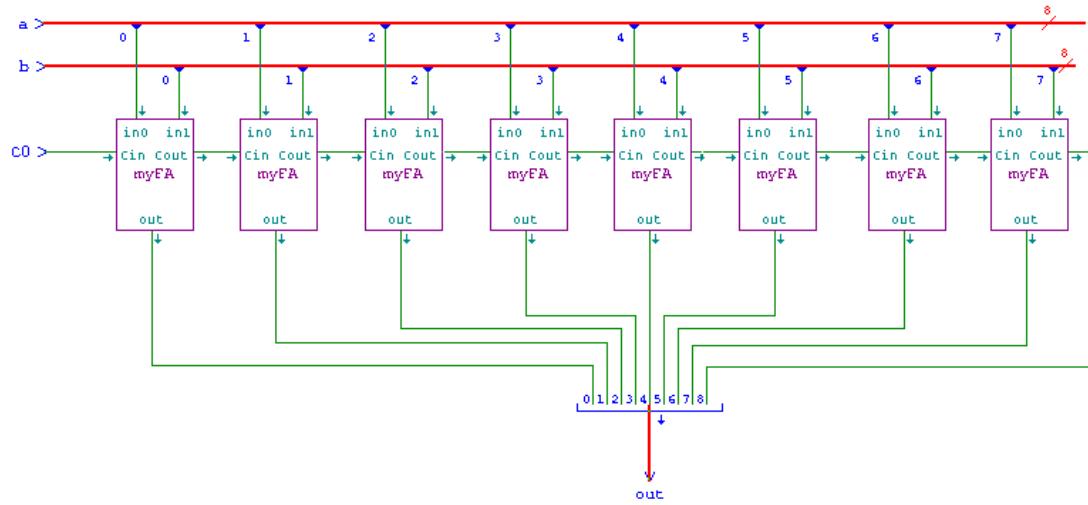
$$\begin{aligned} y_0 &= a_0 \\ y_1 &= 0 \\ y_2 &= a_1 + a_0a_1 \\ y_3 &= a_0a_2 \\ y_4 &= a_2 + a_1a_2 + a_0a_3 \\ y_5 &= a_1a_3 \\ y_6 &= a_2a_3 + a_3 \\ y_7 &= C_{out} \end{aligned}$$

A livello di circuito otteremo quindi il seguente risultato:



<b>Composizione</b>	$6 \times myAND$ $3 \times myHA$ $3 \times myFA$
<b>Area (A)</b>	<b>58</b>
<b><math>T_p</math></b>	<b>16</b>
<b><math>T_c</math></b>	<b>1</b>

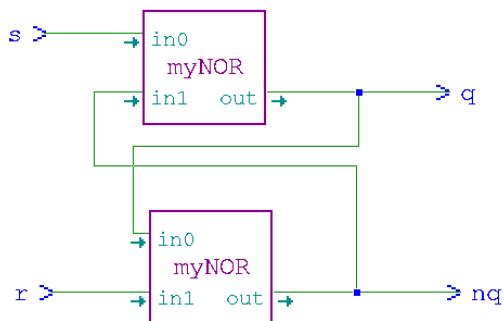
### 3.3.5 RCA8bit



<b>Descrizione</b>	Permette di sommare due operandi da 8 bit e un eventuale riporto in ingresso. Nello specifico del nostro progetto alle classiche due uscite (risultato da 8bit + riporto 1bit) abbiamo preferito un'unica uscita a 9 bit che rappresenta il risultato finale di tutto il circuito
<b>Composizione</b>	$8 \times myFA$
<b>Area (A)</b>	<b>128</b>
<b><math>T_p</math></b>	<b>8</b>
<b><math>T_c</math></b>	<b>4</b>

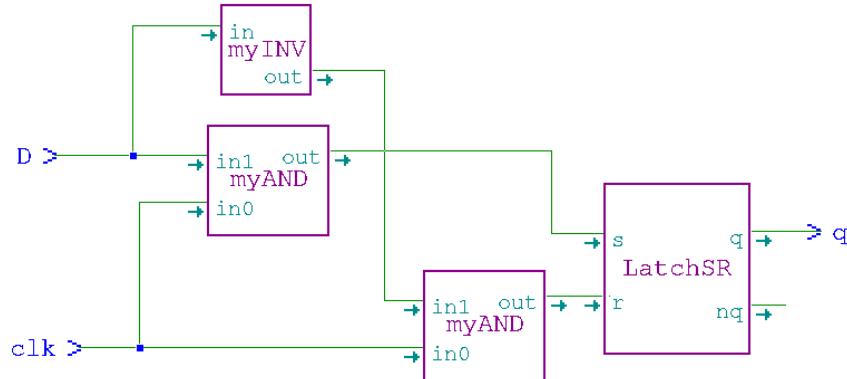
## 3.4 Registri

### 3.4.1 LatchSR



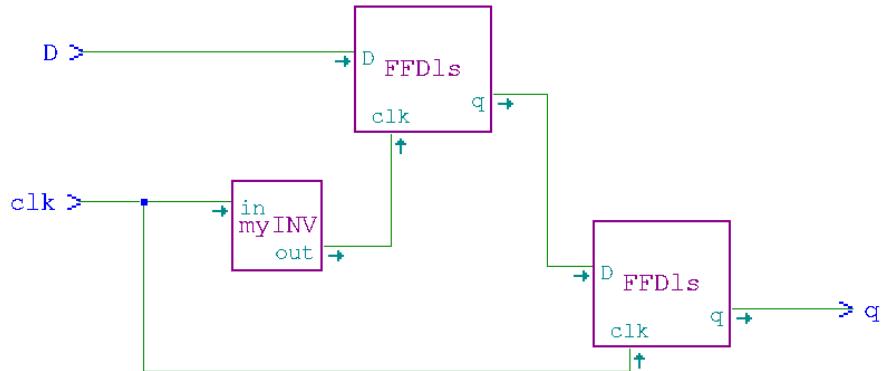
<b>Descrizione</b>	Il <b>Latch Set-Reset</b> è la cella di memoria più semplice. È un circuito bistabile creato con due porte NOR retroazionate. Le uscite sono una l'opposto dell'altra. Possiamo riassumere il suo funzionamento così:
	<ul style="list-style-type: none"> <li>• <math>s=1, r=0</math>: <b>set</b>, ovvero l'uscita <math>q</math> varrà 1</li> <li>• <math>s=0, r=1</math>: <b>reset</b>, ovvero l'uscita <math>q</math> varrà 0</li> <li>• <math>s=0, r=0</math>: <b>hold</b>, ovvero l'uscita dipende dal valore dei risultati precedenti e in particolare si manterrà lo stato del circuito.</li> <li>• <math>s=1, r=1</math>: <b>stato non valido!</b></li> </ul>
<b>Composizione</b>	$2 \times myNOR$
<b>Area (A)</b>	<b>2</b>
<b><math>T_p</math></b>	<b>1</b>
<b><math>T_c</math></b>	<b>1</b>

### 3.4.2 FFDls



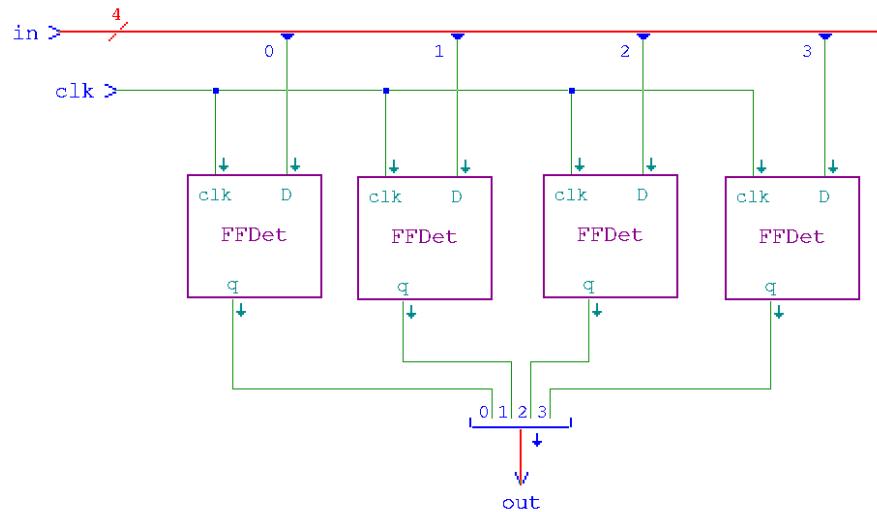
<b>Descrizione</b>	Il <b>Flip-Flop D Level Sensitive</b> risolve il problema dello stato non gestito che aveva il LachtSR utilizzando un unico input $D$ e grazie all'input $clk$ cambia stato solo quando il segnale di clock è alto
<b>Composizione</b>	$2 \times myAND$ $1 \times myINV$ $1 \times LatchSR$
<b>Area (A)</b>	<b>7</b>
<b><math>T_p</math></b>	<b>4</b>
<b><math>T_c</math></b>	<b>3</b>

### 3.4.3 FFDet



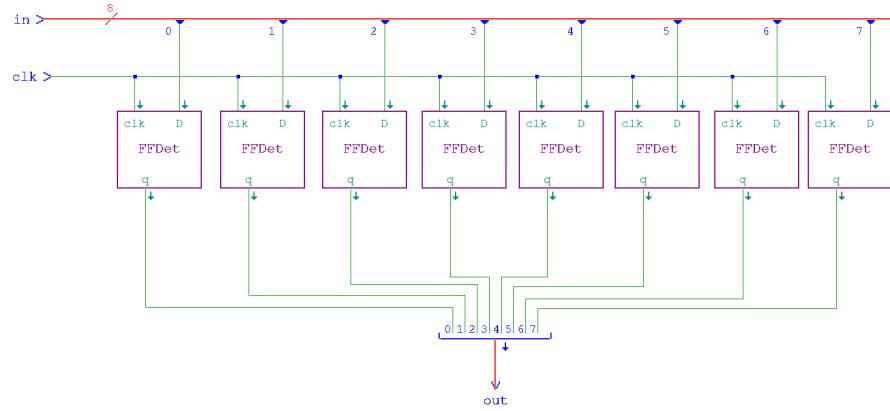
<b>Descrizione</b>	Il <b>Flip-Flop D Edge Triggered</b> è il componente standard per i registri. Può controllare i cambiamenti di valore anche quando il segnale di <b>clock</b> vale 0.
<b>Composizione</b>	$2 \times FFDls$ $1 \times myINV$
<b>Area (A)</b>	<b>15</b>
<b>T<sub>p</sub></b>	<b>9</b>
<b>T<sub>c</sub></b>	<b>3</b>

### 3.4.4 REG4bit



<b>Descrizione</b>	Servono per memorizzare una valore binario di massimo 4bit.
<b>Composizione</b>	$4 \times FFDet$
<b>Area (A)</b>	<b>60</b>
<b><math>T_p</math></b>	<b>9</b>
<b><math>T_c</math></b>	<b>3</b>

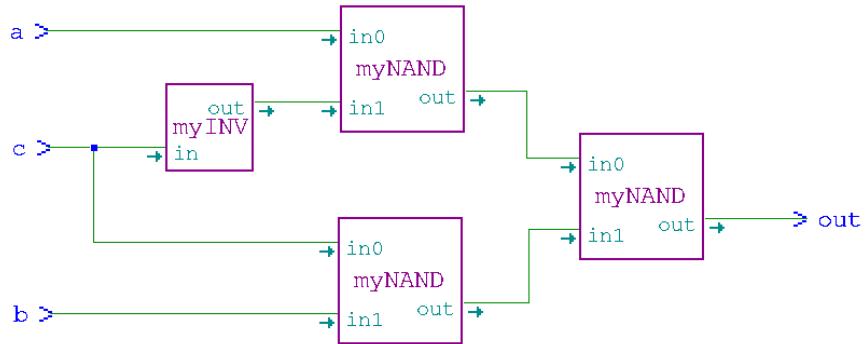
### 3.4.5 REG8bit



<b>Descrizione</b>	Servono per memorizzare una valore binario di massimo 8bit.
<b>Composizione</b>	$8 \times FFDet$
<b>Area (A)</b>	<b>120</b>
<b><math>T_p</math></b>	<b>9</b>
<b><math>T_c</math></b>	<b>3</b>

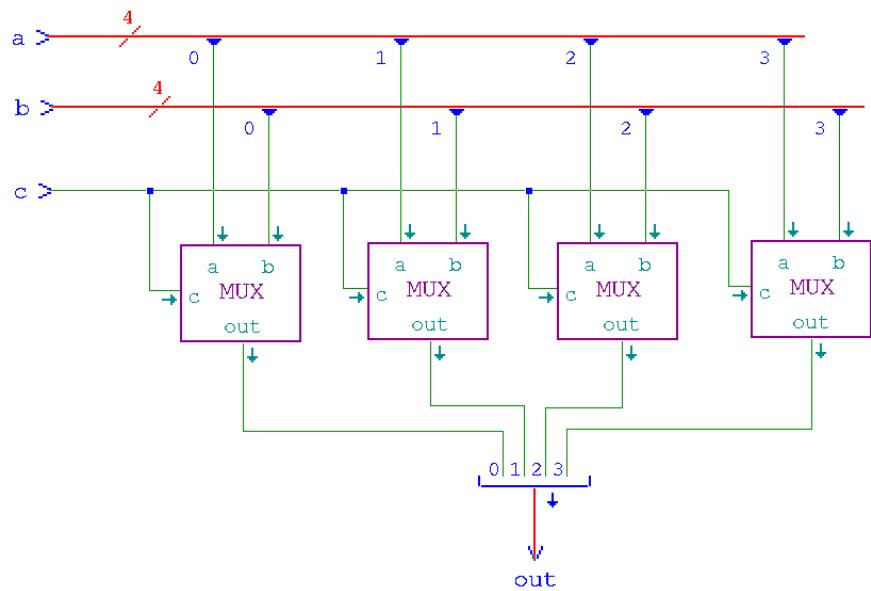
## 3.5 Multiplexer

### 3.5.1 MUX



<b>Descrizione</b>	Il Multiplexer ha due ingressi dati ( <i>a</i> e <i>b</i> ), un'uscita e un ingresso di controllo ( <i>c</i> ). A seconda del valore di <i>c</i> restituisce <i>a</i> oppure <i>b</i>
<b>Composizione</b>	$1 \times myINV$ $3 \times myNAND$
<b>Area (A)</b>	4
$T_p$	3
$T_c$	2

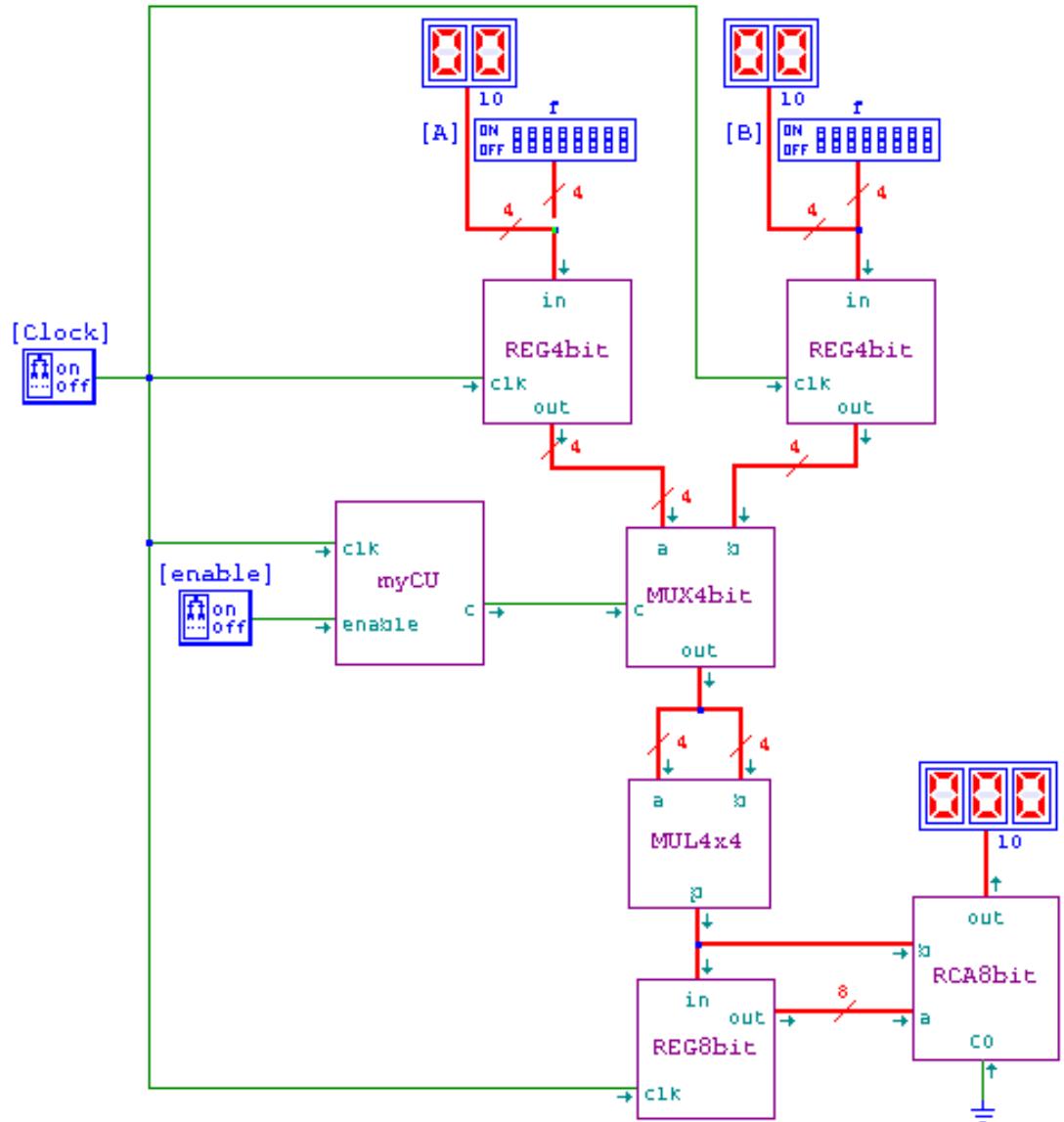
### 3.5.2 MUX4bit



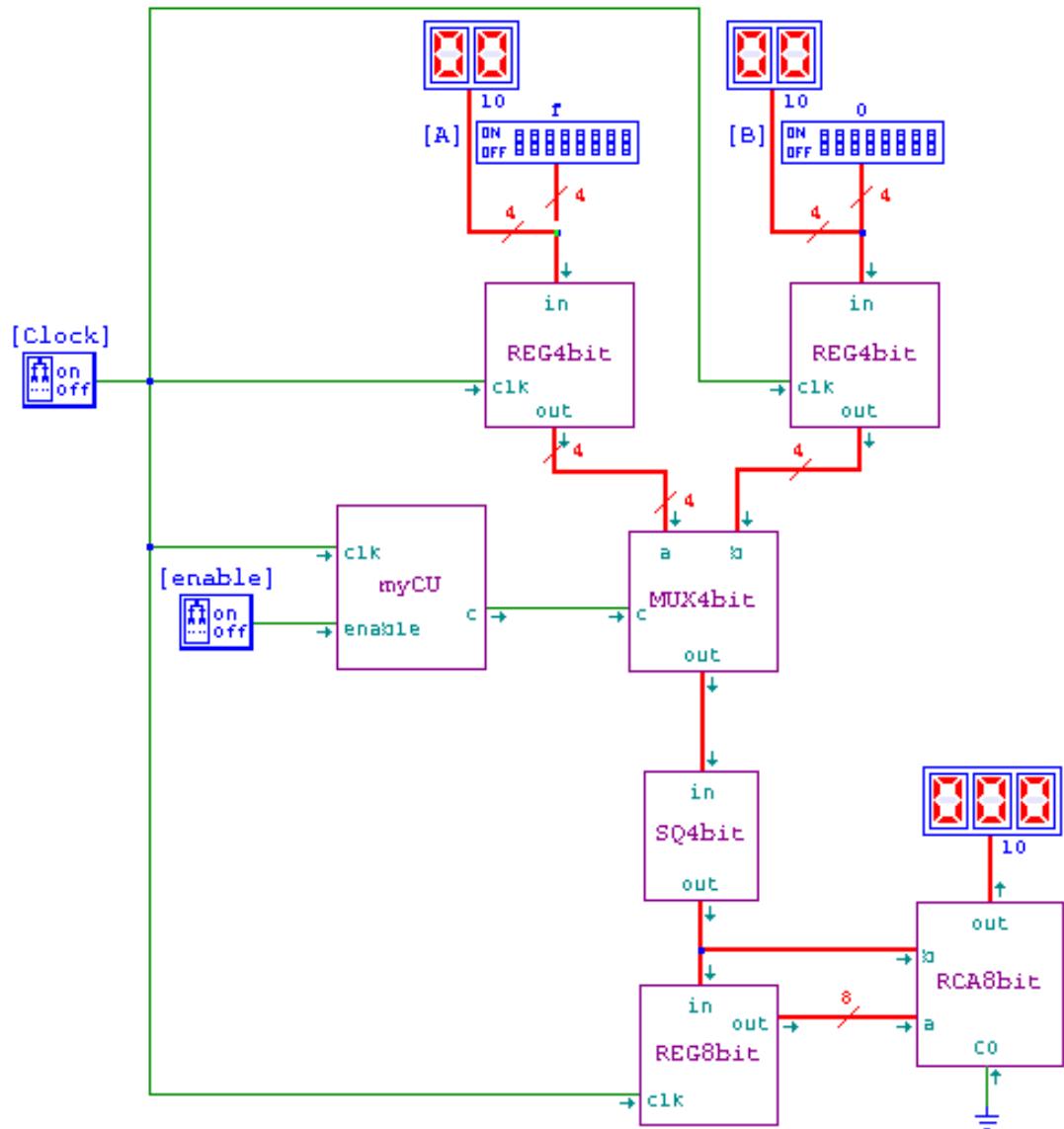
<b>Composizione</b>	$4 \times MUX$
<b>Area (A)</b>	16
$T_p$	3
$T_c$	2

## 4 Data Path

### 4.1 Architettura A: Moltiplicatore generico



## 4.2 Architettura B: Quadratore ottimizzato



## 5 Control Unit

### 5.1 Specifica

La Control Unit (CU) è il componente sequenziale incaricato di gestire il flusso dei dati all'interno del circuito. Nel nostro progetto, avendo adottato la tecnica del *Resource Sharing* per l'elevamento al quadrato, è necessario coordinare l'uso del moltiplicatore (o quadratore) e dell'addizionatore in due istanti temporali distinti.

Lo scopo della CU è generare un segnale di controllo ( $c$ ) che piloti i Multiplexer, alternando la selezione degli ingressi ( $A$  e  $B$ ) ad ogni ciclo di clock. La Control Unit è caratterizzata dai seguenti segnali:

- **Input:**

- $clk$ : Segnale di sincronismo globale.
- $enable$ : Segnale di attivazione. Se basso (0), resetta la macchina; se alto (1), avvia l'alternanza degli stati.

- **Output:**

- $c$ : Segnale di controllo a 1 bit inviato ai Multiplexer.

Abbiamo modellato la CU come una **Macchina di Moore a 2 stati finiti**, in cui l'uscita dipende esclusivamente dallo stato corrente. La logica è ciclica: lo stato futuro ( $S_{next}$ ) è l'inverso dello stato attuale ( $S$ ).

- **Stato 0** ( $S = 0 \rightarrow c = 0$ ): Viene selezionato il primo operando ( $A$ ).
- **Stato 1** ( $S = 1 \rightarrow c = 1$ ): Viene selezionato il secondo operando ( $B$ ).

Di seguito riportiamo la tabella di verità che descrive la transizione di stato e la funzione d'uscita:

Enable	Stato Attuale ( $S$ )	Stato Futuro ( $S_{next}$ )	Output ( $c$ )
0	X	0	0
1	0	1	0
1	1	0	1

Table 1: Tabella delle transizioni e delle uscite della Control Unit.

## 5.2 Implementazione della macchina di Moore

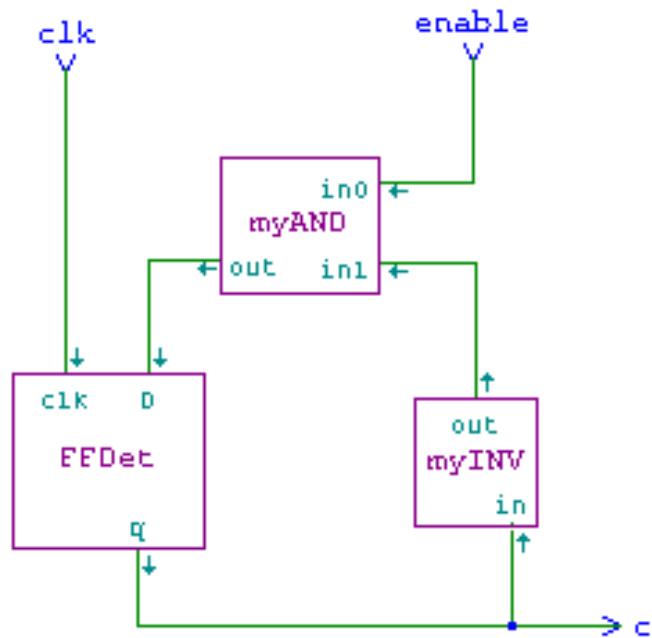
Per implementare fisicamente la macchina descritta sopra, abbiamo realizzato un circuito sequenziale utilizzando un Flip-Flop per memorizzare lo stato e una rete combinatoria per calcolare lo stato successivo.

Nello specifico, il circuito `myCU` è composto da:

1. **Flip-Flop D Edge Triggered (FFDet):** Memorizza lo stato corrente ( $S$ ). Ad ogni fronte di salita del clock, aggiorna il suo valore con quello presente all'ingresso  $D$ . L'uscita  $Y$  del Flip-Flop corrisponde direttamente al segnale di controllo  $c$ .
2. **Inverter (myINV):** Preleva l'uscita corrente del Flip-Flop e la inverte. Questo realizza la logica di oscillazione ( $S_{next} = \bar{S}$ ).
3. **Porta AND (myAND):** Gestisce il segnale di *enable*. Prende in ingresso il segnale *enable* esterno e l'uscita dell'Inverter.
  - Se *enable* è 0, l'uscita dell'AND è forzata a 0 (Reset/Inizializzazione).
  - Se *enable* è 1, l'uscita dell'AND segue l'inverter, permettendo il ciclo continuo  $0 \rightarrow 1 \rightarrow 0$ .

L'uscita della porta AND è collegata all'ingresso  $D$  del Flip-Flop, chiudendo l'anello di retroazione.

### 5.2.1 Scheda Tecnica: myCU



<b>Descrizione</b>	Macchina a stati finiti (Moore) a 2 stati per la gestione del multiplexing temporale. Include logica di reset sincrono tramite porta AND.
<b>Composizione</b>	$4 \times MUX$
<b>Area (A)</b>	18
<b><math>T_p</math></b>	9
<b><math>T_c</math></b>	3

# 6 Simulazione e analisi del progetto

## 6.1 Verifica funzionale

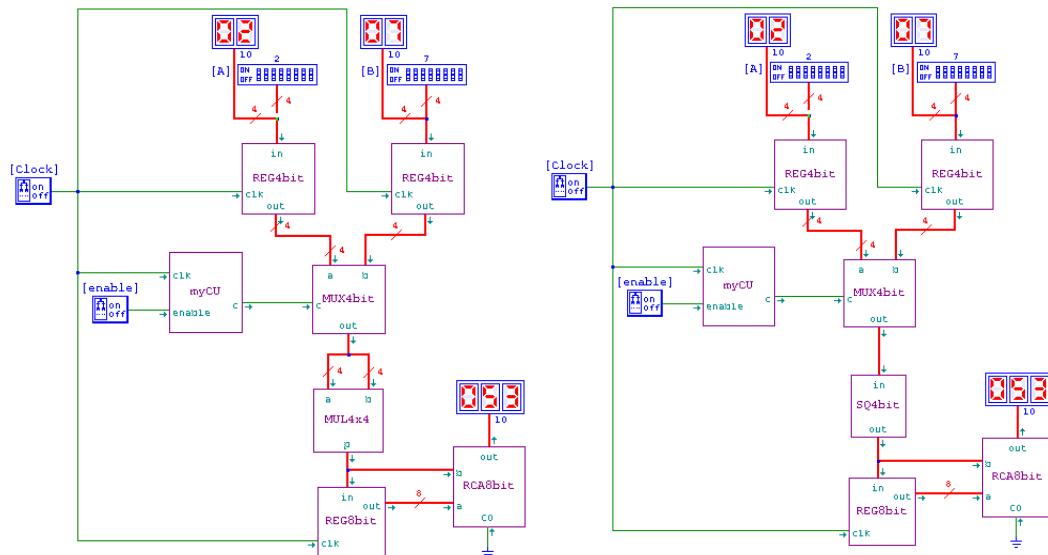
La correttezza funzionale di entrambe le architetture è stata verificata tramite simulazione nel software TkGate. Il processo di verifica segue la scansione temporale imposta dalla Control Unit:

1. **Reset:** Attivazione del segnale *enable*.
2. **Ciclo 1:** Selezione dell'operando *A*. Verifica che il registro intermedio contenga il valore  $A^2$  al fronte di salita del clock.
3. **Ciclo 2:** Selezione dell'operando *B*. Verifica che l'uscita finale dell'addizionatore mostri il risultato  $A^2 + B^2$ .

Entrambe le architetture (con **MUL4x4** e con **SQ4bit**) hanno prodotto i medesimi risultato corretto, validando l'equivalenza funzionale dei due approcci.

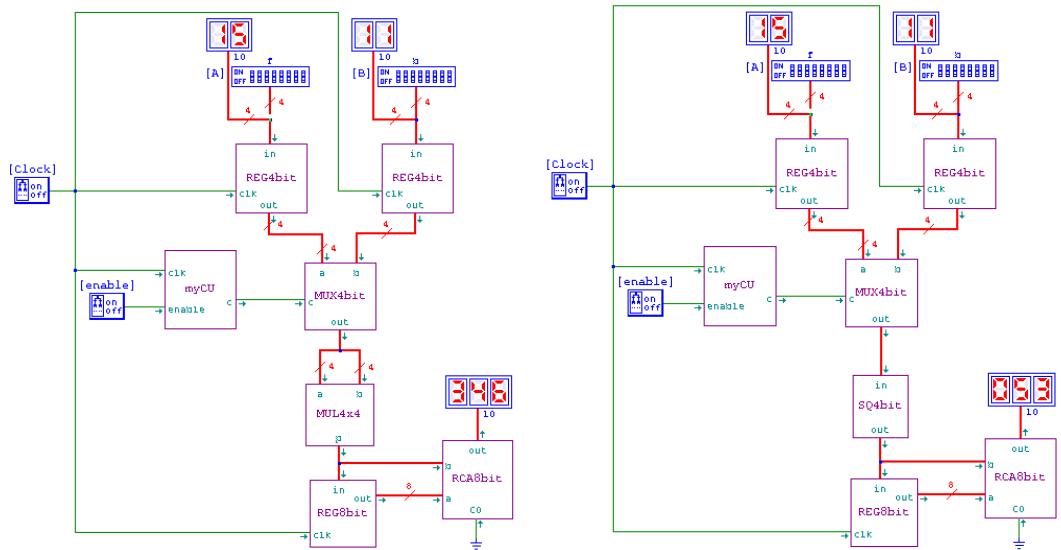
### 6.1.1 Prima simulazione

$$A = 2, B = 7, A^2 + B^2 = 53$$



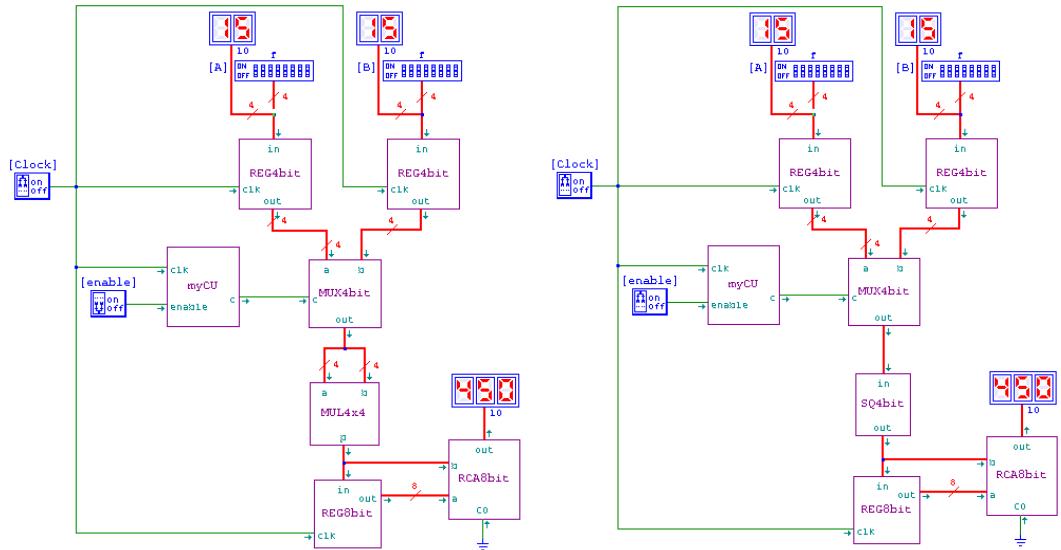
### 6.1.2 Seconda simulazione

$$A = 15, B = 11, A^2 + B^2 = 346$$



### 6.1.3 Terza simulazione

$$A = 15, B = 15, A^2 + B^2 = 450$$



## 6.2 Valutazione di prestazioni e complessità

In questa sezione confrontiamo le due architetture basandoci sulle metriche dei singoli componenti calcolate nel Capitolo 3 e le metriche del CU calcolate nel Capitolo 5.

### 6.2.1 Area

Architettura A			Architettura B		
REG4bit	$2 \times 60$	120	REG4bit	$2 \times 60$	120
REG8bit	$1 \times 120$	120	REG8bit	$1 \times 120$	120
MUX4bit	$1 \times 16$	16	MUX4bit	$1 \times 16$	16
MUL4x4	$1 \times 188$	188	SQ4bit	$1 \times 81$	81
RCA8bit	$1 \times 128$	128	RCA8bit	$1 \times 128$	128
myCU	$1 \times 18$	18	myCU	$1 \times 18$	18
<b>Totale Area</b>	<b>590</b>		<b>Totale Area</b>	<b>483</b>	

### 6.2.2 Tempo di propagazione

Architettura A			Architettura B		
REG4bit	$2 \times 9$	18	REG4bit	$2 \times 9$	18
REG8bit	$1 \times 9$	9	REG8bit	$1 \times 9$	9
MUX4bit	$1 \times 3$	3	MUX4bit	$1 \times 3$	3
MUL4x4	$1 \times 47$	47	SQ4bit	$1 \times 16$	16
RCA8bit	$1 \times 8$	8	RCA8bit	$1 \times 8$	8
myCU	$1 \times 9$	9	myCU	$1 \times 9$	9
<b>Totale <math>T_p</math></b>	<b>104</b>		<b>Totale <math>T_p</math></b>	<b>63</b>	

### 6.2.3 Tempo di contaminazione

Architettura A			Architettura B		
REG4bit	$2 \times 3$	6	REG4bit	$2 \times 3$	6
REG8bit	$1 \times 3$	3	REG8bit	$1 \times 3$	3
MUX4bit	$1 \times 2$	2	MUX4bit	$1 \times 2$	2
MUL4x4	$1 \times 2$	2	SQ4bit	$1 \times 1$	1
RCA8bit	$1 \times 4$	4	RCA8bit	$1 \times 4$	4
myCU	$1 \times 3$	3	myCU	$1 \times 3$	3
<b>Totale <math>T_c</math></b>		<b>20</b>	<b>Totale <math>T_c</math></b>		<b>19</b>

## 6.3 Conclusioni

Il cuore del confronto risiede nella differenza tra il Moltiplicatore Generico e il Quadratore Specifico.

L'approccio specifico (Architettura B) ha permesso di risparmiare **107 unità di area**, che corrispondono a una riduzione del **18%** sull'intero circuito e del **57%** sulla sola componente combinatoria di calcolo. Anche in termini di prestazioni temporali ( $T_p$ ), l'Architettura B risulta vantaggiosa. Il percorso critico nel moltiplicatore generico attraversa l'intera matrice a cascata ( $T_p = 47$ ), mentre nel quadratore, avendo ridotto drasticamente il numero di stadi di addizione (da 8 FA a soli 3 FA in serie nel percorso peggiore), il tempo di propagazione interno è nettamente inferiore ( $T_p = 16$ ).

Possiamo quindi concludere che, per l'applicazione specifica del calcolo di  $A^2 + B^2$ , la combinazione di **Resource Sharing** (per i registri e l'addizionatore) e di **Ottimizzazione Logica Specifica** (modulo SQ4bit) rappresenta la scelta progettuale vincente, minimizzando l'area senza compromettere la funzionalità .