

15

SIDE-EFFECTS

In caso si

- voglie usare un registro
nella chiamata e chiamata
 - Si usino funzioni comandate
che andranno a sottoscrivere ra
 - Si vogliono usare più parcellari
di quanti disponibili come
registri

Allora si salva tutto sulla memoria.

Side effect – parametri numerosi (3)

- Problema:
 - che cosa succede se i parametri e le variabili di una procedura superano il numero di registri disponibili?
 - Soluzione:

STACK > (Def)

In particolare questi dati vengono salvati nello stack, ovvero un'area di memoria dedicata nella forma LIFO. Ovvero di Pillo.

Nello stack viene usato il registro `xc`
detto "sp" stack pointer che punterà
all'ultimo elemento allocato.

Essendo una pila, effettua le operazioni di

- push : aggiunge al 'fondo' delle pile
 - pop : toglie dal 'fondo' delle pile

Nell'esempio di prima:

int somma(int x, int y) {	SOMMA: addi sp,sp,-16	
int rst;		Stack 64 bit
rst = x + y + 2;	x > 10 sd x5,0(sp)	addr addr-4
return rst;	x > x11 sd x20,8(sp)	val x20 addr-8
}	rst > x20 addi x5,x10,x11	val x5 addr-12
....	r > x6 addi x20,x5,2	addr-16
f=f6!	addi x10,x20,0 ld x5,(sp)	addr-20
risultato=somma(f,g);	ld x20,(sp) addi sp,sp,16	addr-24
....	jal SOMMA jal x0,(x)	
PUSH: salvataggio del valore di x5 e x20	...	
	...	
	addi x6,de,1	
	...	
	jal SOMMA	

Nell'esempio di prima

```

int somma(int x, int y) {
    int rst;
    x = x + y;
    return rst;
}

int f(int x) {
    if (x <= 0)
        f = -1;
    else
        f = 1;
    return f;
}

int risultato=somma(f,x,y);
...
```

POP: ripristino del valore
di x5 e x20

SOMMA:	add sp,_esp
sd x0,0(sp)	sd x0,0(sp)
add x0,x20	add x0,x20
add x0,x20	add x0,x20
addi x1,x20	addi x1,x20
ld x0,0(sp)	ld x0,0(sp)
addi sp,_esp	addi sp,_esp
jlr x0,0(x1)	jlr x0,0(x1)
...	...
addi x6,x6,1	addi x6,x6,1
jsi SOMMA	jsi SOMMA

Side effect – sovrascrittura dei registri (1)

```

int somma(int x, int y) {           SOMMA: add    x5,x0,x1
    int rst;                      x  →  x10
    rst = x + y + 2;              y  →  x11
    return rst;                   rst → x20
}                                     f  →  x6
                                         ↗
                                         ...
                                         addi   x6,x6,1
                                         ...
                                         jal   SOMMA

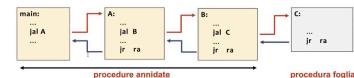
...
f=f+1;
risultato=somma(f,g)
...

```

- **Problema:** che cosa succede se il registro x5 contiene un valore usato dalla procedura chiamante?

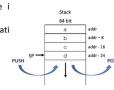
Side effect – procedure annidate (2)

- Problema
 - Nel caso di procedure annidate, il return address (`x1`) viene sovrascritto
 - Soluzione:



Lo stac

- In quale area di memoria è possibile salvare i registri per evitare la perdita del valore?
 - In memoria viene definita una struttura dati dinamica, lo stack
 - Accesso: Last In First Out (LIFO)
 - Operazioni:
 - PUSH: aggiunge un elemento sin rima allo stack*
 - POP: rimuove un elemento dalla cima dello stack
 - La cima dello stack è identificata dallo Stack Pointer (SP)

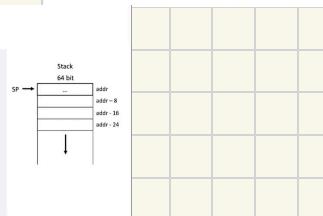


- PUS

- Decrement SP
 - Scribe in M[SP]

• PC

- Legge da M[SP] `ld x20, 0(sp)`
 - Incrementa SP `addi sp, sp, 8`



REGISTRI > FRAME POINTER (x8)

È un puntatore che punta al primo spazio allocato dal record di attivazione corrente.

Viene usato come punto di riferimento per l'utilizzo di variabili dentro un record in quanto immutabile a differenza dello stack pointer.

Record 1
= 2
= 3

REGISTRI > CONVENZIONI

L'uso di un registro piuttosto che un altro fa varicare le convenzioni sui salvataggi in memoria:

- Se si usano $\text{a}0, \dots, \text{a}7$ e $\text{t}3, \dots, \text{t}6$, allora:
 - Il chiamante è tenuto (se serve) a salvare il contenuto se gli serve.
 - Il chiamato **NON È TENUTO** a salvare niente.
- Se si usano $\text{ra}(x_1), \text{sp}(x_2), \text{gp}(x_3), \text{tp}(x_4), \text{fp}(x_5), x_6, [s_1, \dots, s_{11}]$
 - Il chiamante **NON** è tenuto a gestire la memoria.
 - Il chiamato È tenuto a salvare i dati (se servono).

Questo per ragioni di performance riguardo l'uso della memoria.

FASI DI ESECUZIONE ➤

1. PREPARAZIONE ALL'ESECUZIONE [CHIAMANTE]

- Se è necessario, si salvano i valori dei registri
- Si preparano gli argomenti salvandoli nei vari registri dei parametri

2. CHIAMATA (INVOCAZIONE PROCEDURA): Tramite jal [CHIAMATO]

3. PROLOGO NEL CHIAMATO [CHIAMATO]

- Se si usano i registri ra, fp, sp, s1...s8, si dovranno salvare
- Inizializzazione fp al nuovo stack di allocazione.
- Se è ammesso e se ne vuole chiamare un'altra vengono le regole del diamante.

4. ESECUZIONE CHIAMATO

5. EPILOGO CHIAMATO

- Salvare i valori di ritorno da α_0, α_1
- Ripristinare i registri in carico al chiamante (ra, sp, fp)
- Decrementare sp e fp.

6. RITORNO AL CHIAMANTE: tramite jr ra

7. POST-CHIAMATA

- Si prende i valori di ritorno da α_0, α_1 (salvati dal chiamato)
- Si prende i dati che aveva scritto in memoria

L7 •

201 > [Lipo U]

Lo Ioi o Local Upper Immediate impone i bit nel range **[31, 12]** il valore specificato:

SYNTAX >

Ioi x_{dest} IMMEDIATE

- con
- x_{dest} il registro di destinazione
 - IMMEDIATE il valore da impostare

I 32 bit rimanenti avranno valore 0 o 1 a seconda del segnale

ESEMPIO : "Voglio aggiungere il valore 0x12345678 in xs"

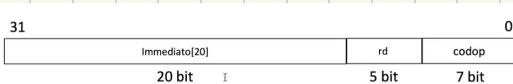
(1) uso lui

$lui \quad xs \quad 0x12345 \leftarrow$ 20 bit, perfetti per lui

(2) uso ori

$ori \quad xs, xs, 0x678 \leftarrow$ I bit rimanenti li riempio con una maschera di bit.

ISTRUZIONI > "U"



DEFINIZIONI >

• **LINGUAGGIO ASSEMBLATIVO**

Linguaggio con il quale si sostituisce il codice in binario con istruzioni mnemoniche

• **ASSEMBLACORE**

Elemento che converte il linguaggio assemblativo in programma esegibile

ASSEMBLYNG >

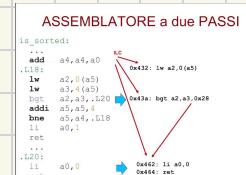
In fase di assemblamento, l'assemblatore effettua 2 giri delle istruzioni:

- (1) Legge le istruzioni asseguando un ILC (Instruction Location Counter) per enumerare le istruzioni.

- Si salva i valori in una symbol table di supporto.
- Risolve le pseudo-istruzioni.

- (2) Ri-legge le istruzioni in linguaggio assemblativo e

- Le converte in linguaggio macchina
- Risolve le etichette usando come offset l'ILC salvato nella symbol table.



LINKING E LOADING >

Sono le fasi finali che si occupano di combinare insieme più file oggetto (es: link di printf). In particolare:

- **LINKER**

Fonde gli oggetti usando uno spazio unico di indirizzamento fornendo ad ogni oggetto:

- Range
- Indice di inizio

Modulo	Lunghezza	Indirizzo partenza
A	400	100
B	600	500
C	500	1100
D	300	1600

Determinando le precedenze, ri-ordina gli elementi

- **LOADER**

Involto durante l'esecuzione dell'eseguibile :

- Allocare memoria per il programma
- Inizializza i registri
- Chiama il main

RIVOCAZIONE DINAMICA >

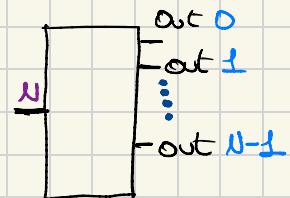
Quando si fa riferimento a librerie esterne, è possibile fare un:

- **COLLEGAMENTO STATICO**: Le librerie sono compilate col codice
- **= DINAMICO**: ~~=~~ sono referenziate con degli indirizzi (DLC)

C. COMBINATORI > DECODER

Circuito con

- 1 ingresso da N bit
- 2^N uscite che vengono selezionate con una funzione booleana



C. COMBINATORI > MULTIPLEXER

È un circuito con 2 ingressi, 1 selettore e 1 uscita in cui l'ingresso da passare nell'uscita verrà selezionato usando i bit del selettore.

Più facile avendo così

S	A	B	Out
0	0	0	0 → è tutto spento
0	0	1	0 → A è spento
0	1	0	1 → A è acceso
0	1	1	1 → —
1	0	0	0] Guarda B
1	0	1	1
1	1	0	0
1	1	1	1

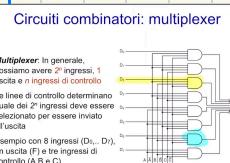
A è B sono binari di dei bit

$$\begin{aligned} \bar{S} \cdot A \cdot \bar{B} + \bar{S} \cdot A \cdot B + S \cdot \bar{A} \cdot B \\ S \cdot A \cdot B \end{aligned}$$

Raccogliendo

$$= \bar{S} \cdot A (\bar{B} + B) + S \cdot A \cdot B$$

$$= \bar{S} \cdot A + S \cdot A$$



LQ •

ALU

È composta da una serie di circuiti quali:

- INPUTS:**

- **SELETTORE DI OPERAZIONE**

Usato come input per selezionare l'operazione da eseguire nel multiplexer dell'ALU

- **A** : Il primo operatore

- **B** : Il secondo

- **CARRY-IN** : Il carry in ingresso all'ALU per la somma.

- **A-INVERT** : Inverte] Utili per -de-morgan

- **B-INVERT** : =] Sottrazioni

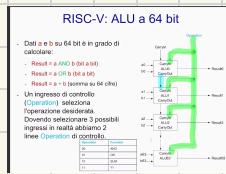
- **LESS** : Bit in ingresso per controllare che $a < b$ lo
vai impostato a 1 solo per il bit più significativo
Per gli altri 0.

- OPERAZIONI** : Variano a seconda dei valori sul selettore di operazione. Sono (somma, AND, OR, NOT...)

- OUTPUTS:**

- **RESULT** : È il risultato dell'op.

- **CARRY OUT** : Il riporto in uscita. Usato per mettere in parallelo più ALU - Divenireà carry-in



WB: Il carry-in rimane nello stesso ALU per le sottrazioni (serve a negare che si fa) (invertendo i bit + somma 1)

- **SET** :

Sarà il valore di ritorno dell'ultimo ALU per i bit meno significativi.

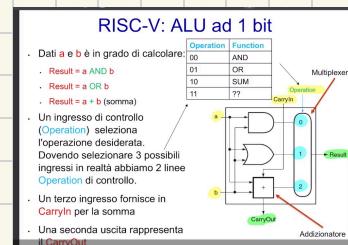
Funge da input per il less della 1° ALU

- **OVERFLOW DETECTION**

$C_{-W} \oplus C_{-OUT}$

Si ha quando il carry in e il carry out del Bit più significativo sono discordi.

WB: $B_{-INVERT} \in C_{-W} \oplus C_{-OUT}$ A 1 per sottrazioni (0 regole)



ALU > A GRANDI LINEE

A grandi linee l'ALU si rappresenta con 4 bit

A	B	operation
0000	0000	AND
0001	0001	OR
0010	0010	ADD
0110	0110	SUB
0111	0111	LESS
1100	1100	NOR

- Zero
- Result
- Overflow

[1] [2] [3] [4]

- [4-3] Selettore di operazione
 $[00 \rightarrow \text{AND}, 01 \rightarrow \text{OR}, 10 \rightarrow \text{SUB}, 11 \rightarrow \text{LESS}]$

- [2] B-negate

Imposta $C-in=1$ e $B\text{-negate}=1$

- [3] A-invert

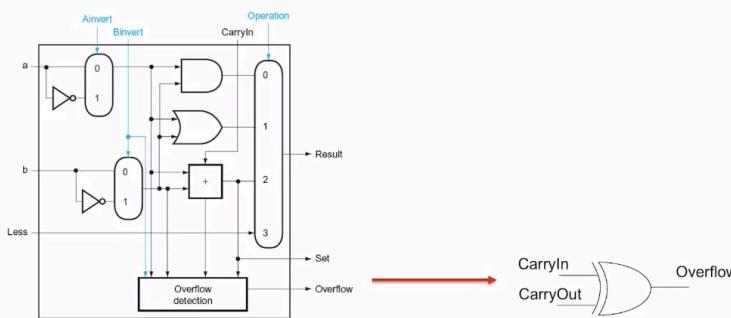
CIRCUITI SEQUENZIALI >

Sono circuiti con collegamenti retro-attivi. Vengono anche chiamati Latch.

CIRCUITI SEQUENZIALI > LATCH-SR

Tipo di circuito che memorizza 1 bit. In particolare ha 2 inputs:

- | | |
|------------|-------------------------------|
| 00 → HOLD | Restituisce il bit in memoria |
| 01 → RESET | Imposta 0 in memoria. |
| 10 → SET | Imposta 1 in memoria. |



PROBLEMI COL LATCH-SR \rightarrow CLOCK

Siccome la combinazione 11 è instabile nei circuiti sequenziali di Tipo Latch-SR, bisogna evitare che accada.

Ciò si può fare con un circuito di clock, ovvero un circuito che emette segnali a onde quadre:



$\cancel{1}$: Le sezioni orizz. si chiamano livelli

I fronti possono essere

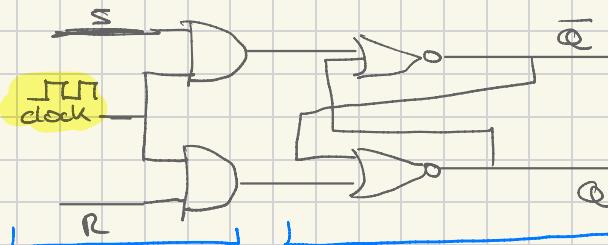
- Di salita ↑
- Di discesa ↓

$\cancel{1}$: Le sezioni verticali si chiamano fronti.

Così facendo, si sincronizza l'esecuzione col clock per decidere di salvare (accendere) ai dati solo quando essi sono stabili

LATCH SR \rightarrow SINCRONIZZATO

Incorporando il comportamento del clock all'interno del latch avremmo:



Parte standard del Latch.

N.B.: Questa soluzione
non risolve $S=R=1$
ma consente di
eseguire le operazioni
SET e RESET in situazione
stabile

Viene inserita la parte iniziale con il confronto con il clock

- CLOCK = 0 [liv. basso $\square \underline{\square}$]

S ed R verranno sempre 0 Hold

- CLOCK = 1 [liv. basso $\square \underline{\square}$]

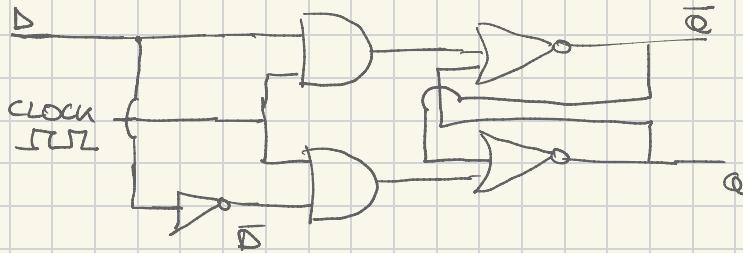
S ed R si comportano come gli input normali

L10 •

2

LATCH \Rightarrow SINCRONIZZATO

Per impedire la "combinazione anomala" 11 nel latch
si sono sostituite le 2 variabili con 1 sola (D):



Così facendo la comb
11 diventa impossibile

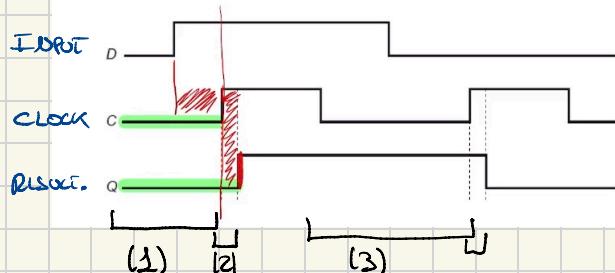
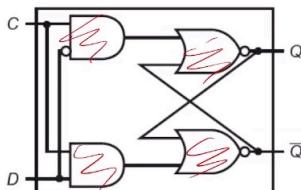
I casi possibili diventano

CLOCK=0 \rightarrow Hold 00

CLOCK=1

- $\rightarrow D=1 \rightarrow$ Set 10
- $\rightarrow D=0 \rightarrow$ Re-set 01

A livello di analisi, il comportamento nel Tempo diventa:



(1) Anche se $D=1$
dopo un po', il
circuito NON è ancora
stabile per cui rimane
in Hold (0)

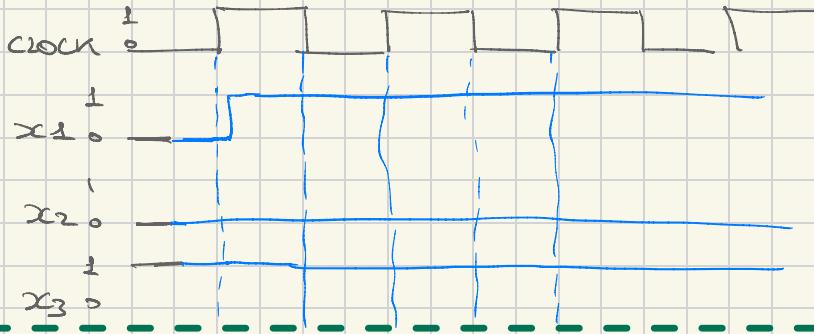
(2) Dopo il passaggio al
livello $C=1$, c'è comunque
un offset per alterare
il risultato.

(3) Anche qui se $D=0$ dopo un
po' ma $C=0$, Q rimane
in Hold

LATCH D > SINCRONIZZATO

ES: "Data la seguente istruzione, descriverla nel tempo"

ADD x_1, x_2, x_3

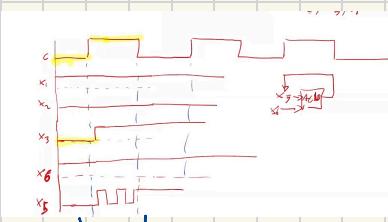


LATCH-D > PROBLEMI / DIFFICI

Il problema principale di questo approccio è che

- $\text{clock} = 1$ I segnali dovrebbero rimanere sempre gli stessi per evitare casi di oscillazione
- $\text{clock} = 0$ Si può "lavorare" sui dati solo qui:

ES: ADD x_5, x_6, x_7



Detto "Problema della Trasparenza" del Latch

Ip Sommandolo a se stesso, quando $\text{clock} = 1$ il risultato oscilla

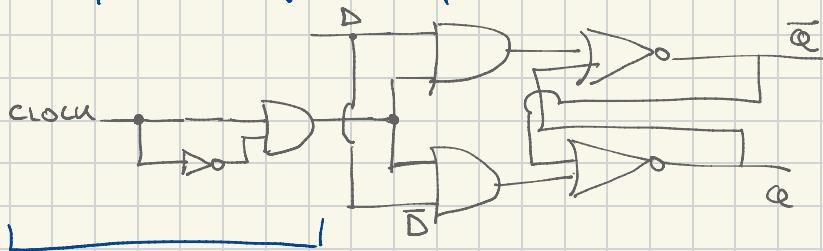
FLIP FLOP D >

Per risolvere il Problema della Trasparenza e ottimizzare il Tempo con cui si eseguono le operazioni si effettua la commutazione nei fronti al posto dei livelli



! commutazione di Livello (come è fino a ora)
/ = Fronte (come si vuole)

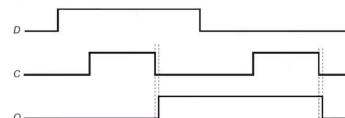
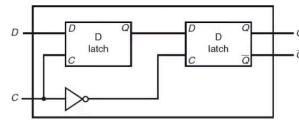
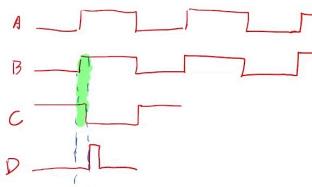
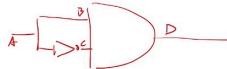
Per implementare questo comportamento nel latch D sincronizzato:



Condizione Teoricamente sempre falsa.

Ma che, in realtà, per un piccolo ritardo causato dalla porta NOT, per $CLOCK=1$ il segnale sarà 1

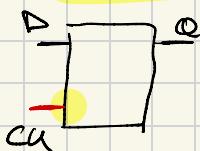
Effettivamente, si simula il fronte in salita minimizzando il livello



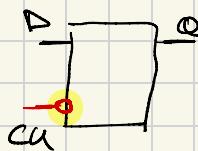
Implementazione per il Fronte di discesa.

FLIP FLOP / LATCH SR > SIMBOLOGIA

- LATCH D

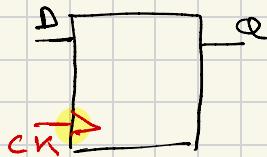


Tipo D sync
con commutazione
su livello alto

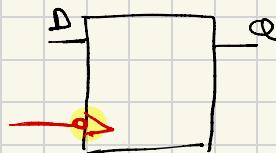


Tipo D sync attivato
a livello basso (0)

- FLIP FLOP D



Flip flop D
con
attivazione
a fronte di
salita



Flip flop D
con
attivazione
a fronte di
Discesa.

BUFFER NON INVERTERI / THREE STATE

È un tipo di circuito che può

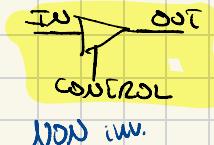
- CHIUDERE IL CIRCUITO:

Passando quindi 0/1 a seconda
dell'input iniziale

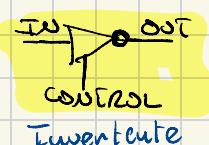
IN ————— OUT

- APRIRE IL CIRCUITO

IN → OUT



NON inv.



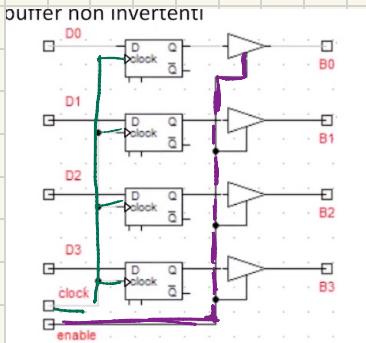
Invertente

REGISTRI >

Ogni registro viene composto da un'unione di flip-flops.

D:

ES: Registro da 4 bit



- Il clock è propagato sugli altri flip flops del registro

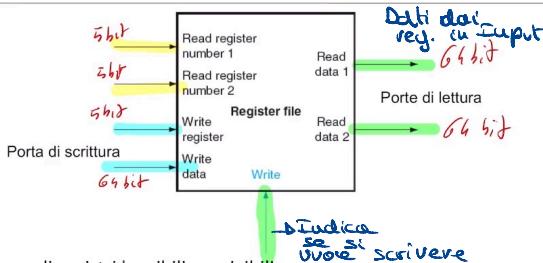
- Per attivare / dis-attivare il circuito, si utilizzano i buffer non invertenti

REGISTRI > BLOCCO DI REGISTRI

È un tipo di registro che, ad istruzione data consente di specificare all' ALU quali sono le porte in scrittura e quali sono le porte in lettura.

ADD $x_1, x_2 \rightarrow R$

Blocco di registry (register file)



INPUT:

- Read Register 1:

L'-esimo primo registro da cui leggere i dati (0,31)

- Read register 2:

L'-esimo secondo registro da cui leggere i dati (0,31)

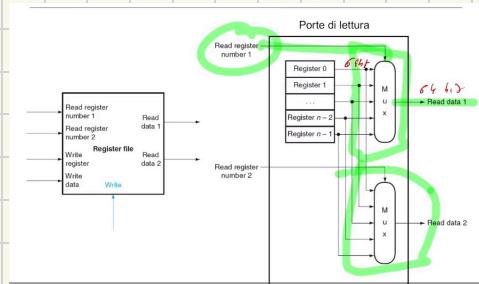
- Write register

L'-esimo registro su cui scrivere

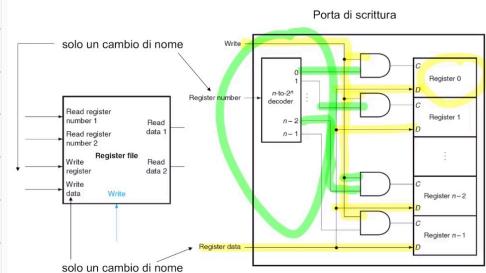
- Write data

I dati da scrivere

REGISTRI ➤ BLOCCO DI REGISTRI: IMPLEMENTAZIONE



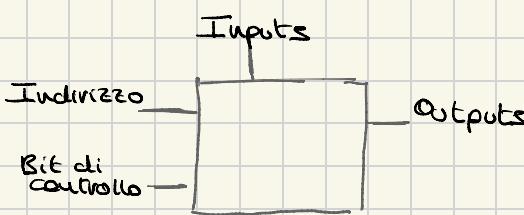
L' n -esimo registro scelto nel blocco si seleziona con un multiplexer.



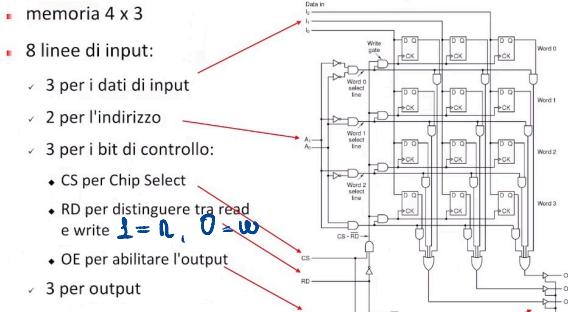
Se e dove andare a scrivere si decide con un decoder.

MEMORIA ➤

La memoria ha una struttura simile ai registri:



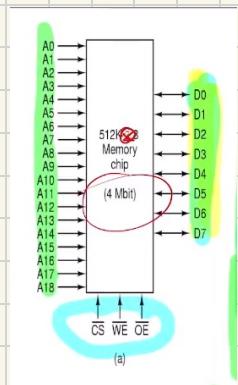
ES: Memoria $4 \times 3 = 4$ word $\times 3$ inputs



le word hanno una priorità in output

CASO D'USO: LETTURA MEMORIA

(1) "Data la seguente memoria, calcolarne le capienza"



L1) Determino il tipo di circuito:

Non ha le porte \overline{RAS} e \overline{CAS} per cui è normale

(2) Calcolo il num di porte

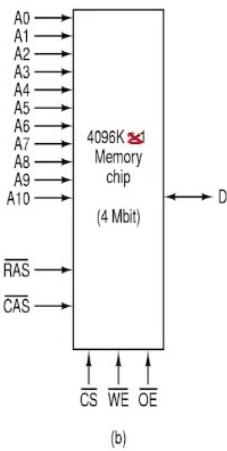
$$1 \text{9 porta} \Rightarrow 2^{10}$$

(3) Calcolo i dati: 8 per def.

(4) Trovo il resi:

$$\begin{aligned} 2^{10} \cdot 8 &= 2^9 \cdot 2^3 = 2^{22} & [\text{Mbit} = 2^10] \\ &= 2^{10} \cdot 2^10 \cdot 2^2 = 4 \text{ Mbit} \end{aligned}$$

(2) Con matrice:



L1) Determino il tipo di circuito:

Ha \overline{RAS} V \overline{CAS} . È di tipo matriciale

(2) Calcolo num di porte

$$2^{10} \cdot 2^1 \cdot 1 = 2^{21} = 4 \text{ Mbit}$$

\uparrow \uparrow
rows cols \rightarrow Dim dato

RAM >

Random Access Memory
È un tipo di memoria che può essere creata con più modi:

- SRAM: Usa i Flip Flop. È molto veloce ma molto costosa
- DRAM: Usa i Transistors. È più lenta ma costa meno.

ROM

Read Only Memory. È un tipo di memoria che si suppone debba essere scritta 1 volta.

In realtà si può scrivere tante volte ma a veloci volte

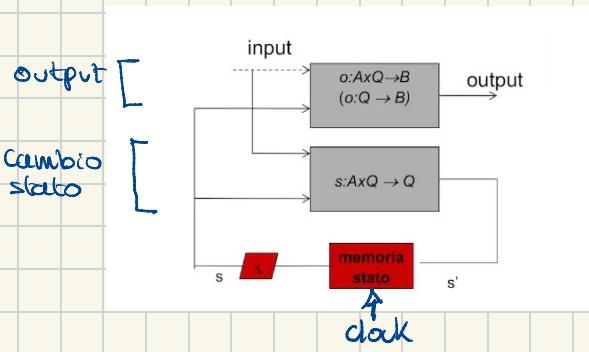
MACHLINE A STATE FWITE ➤

I ripassare da FDI! I Solo macchine!

MACHLINE A STATE FWITE ➤ REALIZZAZIONE

Le macchine a stati finiti possono essere realizzate tramite dei circuiti combinatori.

Per tenere memoria di stato corrente e output, si utilizzano i flip flop per ogni bit:



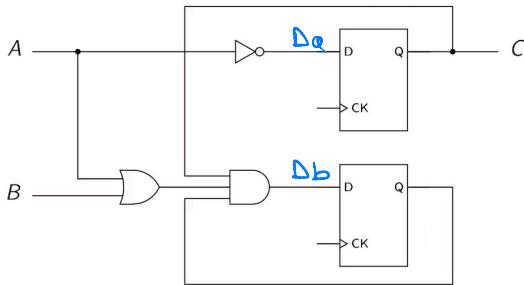
Anche in questo caso per sincronizzare si aggiunge un clock.

CASO D'USO: ANALISI DI UN CIRCUITO SEQUENZIALE

Data una rete sequenziale sincrona (circuito) si ricavano: Eq. booleane, Tabelle di Verità, Diagrammi di stato.

"Analizzare il seguente circuito sequenziale"

Analisi di Reti Sequentziali Sincrone: Esempio



(1) Analisi del circuito

Ingressi: 2 (A,B) $\rightarrow \{00, 01, 10, 11\}$

Uscite : 1 (C) $\rightarrow \{0,1\}$ \rightarrow output dello stato

(2) Ricavo l'eq. booleana

- Definisco degli inputs per il Flip Flop
 - Δa : Input al 1° flip flop
 - Δb : $=$ $=$ 2° $=$ $=$
 - O_a : Output del 1° flip flop
 - O_b : $=$ $=$ 2° flip flop

• Ricavo output

$$\bullet C = O_a$$

• Ricavo Input

$$\bullet \Delta a = \bar{A}$$

$$\bullet \Delta b = C \cdot (A+B) \cdot O_b \Rightarrow \Delta b = O_a \cdot (A+B) \cdot O_b$$

CASO D'USO: ANALISI DI UN CIRCUITO SEQUENZIALE

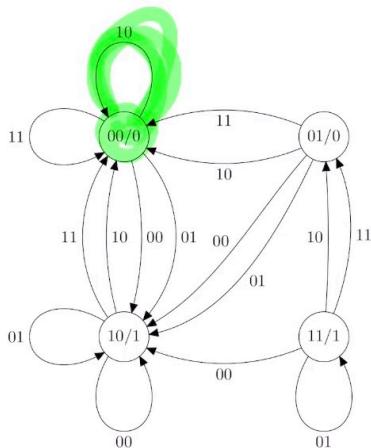
(2) Ricavo Tabella di verità

Tabella di Stato

Stato presente		Ingressi		Stato futuro		Uscite
E	F	A	B	D_E	D_F	C
0	0	0	0	1	0	0
0	0	0	1	1	0	0
0	0	1	0	0	0	0
0	0	1	1	0	0	0
0	1	0	0	1	0	0
0	1	0	1	1	0	0
0	1	1	0	0	0	0
0	1	1	1	0	0	0
1	0	0	0	1	0	1
1	0	0	1	1	0	1
1	0	1	0	0	0	1
1	0	1	1	0	0	1
1	1	0	0	1	0	1
1	1	0	1	1	1	1
1	1	1	0	0	1	1
1	1	1	1	0	1	1

O_a, O_b
Output
del flip
flop

D_a, D_b \rightarrow uscite
Input del
flip flop



Dato l'eq. booleana
creando la tavola
di verità si possono
ricavare tutti i valori.

(3) Macchine a stati finiti

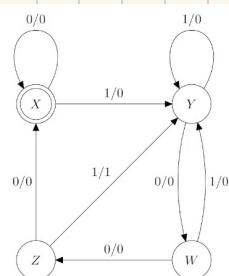
SINTESI >

Da Testo libero verso Diagramma Stati \rightarrow
 Tabella stati \rightarrow f. bool \rightarrow circuito

ES

- ▶ Vogliamo costruire una rete sequenziale che riconosca la presenza di una certa sequenza di bit fissata, anche se inclusa in una sequenza più lunga.
- ▶ La sequenza di bit che vogliamo riconoscere è 1001.
- ▶ La rete dovrà avere un ingresso A e un'uscita B e dovrà riconoscere la sequenza di bit 1001 applicata all'ingresso A .
- ▶ Più precisamente, l'uscita dovrà valere 1 se e solo se:
 - ▶ In corrispondenza ai 3 tre precedenti fronti di salita del clock, i valori letti in A erano, rispettivamente, 1, 0 e 0.
 - ▶ Il valore attuale dell'ingresso A è 1.

(1) Ricavo diagramma degli stati



Da cui si assegnano dei Bit per

- S iniziale: 4 stati $2^2 = 2$ bit
- S futuro: --
- Input: 1 bit
- Output: 1 bit

(2) Ricavo Tabella Stati

Tabella di Stato Esplicita

Stato presente		Ingressi	Stato futuro		Uscite
E	F	A	D _E	D _F	B
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	1	0	0
0	1	1	0	1	0
1	0	0	1	1	0
1	0	1	0	1	0
1	1	0	0	0	0
1	1	1	0	1	1

Ricordarsi di dividere in S input e stati output

← EFA

Da cui leg. booleana con Forme Norm. digitaliva per uscite