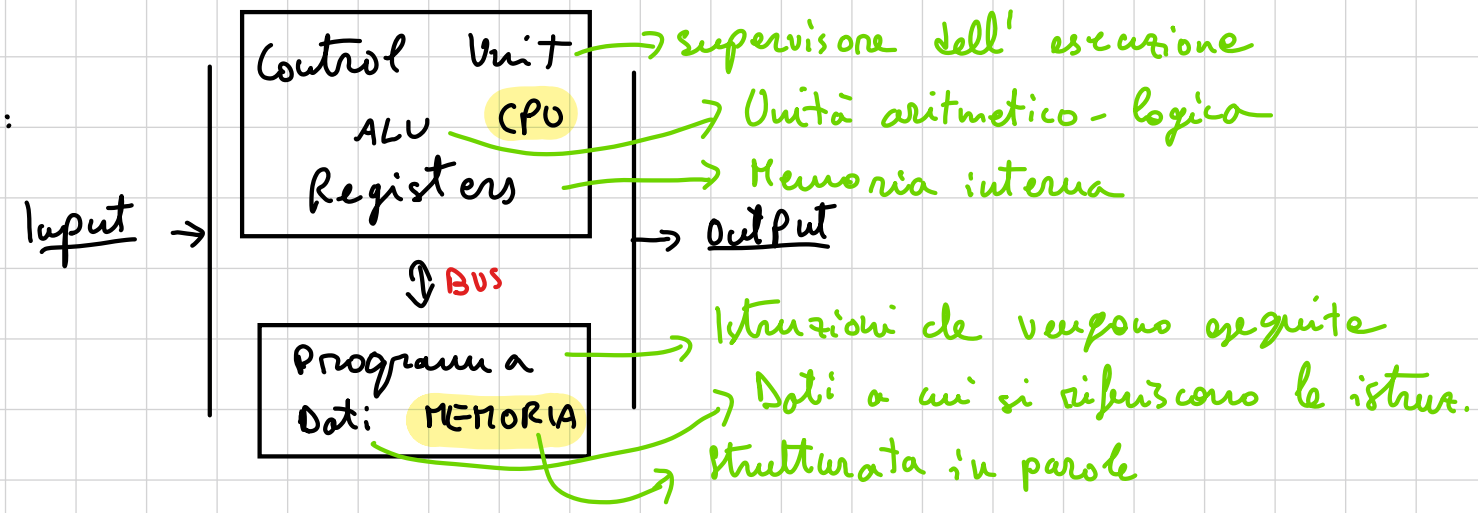
APPUNTI ARCHI I

NB: appunti presi sulle slide di Basilico. Gli argomenti sono gli stessi, ma le modalità possono differire tra Borghese e Basilico.

(parte su circuiti FW è di Borghese)

Von Neumann:



Fetch: Istruzione caricata in Instruction register (IR)
Decode: La CU riconosce l'istruzione e predispone ALU
Execute: La ALU svolge le operazioni
Memory: Accesso alla memoria se previsto
Write back: I registri vengono aggiornati

IS - Instruction Set

Insieme delle istruzioni che la macchina sa eseguire
↳ in linguaggio macchina
È il primo livello di astrazione sull' hw (ISA, Instruction set Architecture)
Ce ne sono di 2 tipi:
- RISC (Reduced): istruz. semplici e regolari (chiedono poche risorse) ARM, Mips
- CISC (Complex): istruz. complesse (.. .. più .. ..) Intel x86_64

Elaborazione digitale

Un elaboratore deve rappresentare l'info. su cui lavora tramite valori di tensione elettrica. I segnali sono in analogia con il fenomeno che rappresentano (segnale ANALOGICO / CONTINUO)
L'elaborazione più importante è quella numerica
Si usano 2 simboli convenzionali: 0 e 1. I transistor sono in grado di mantenere dentro uno dei 2 simboli
Ci sono vari sistemi di enumerazione: (base + notazione)
regole con cui si calcola il valore in base ai simboli
↳ insieme di simboli che possiamo usare per la rappresentazione

- $B_{10} = \{0,1,2,3,4,5,6,7,8,9\}$
- $B_8 = \{0,1,2,3,4,5,6,7\}$
- $B_{16} = \{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F\}$ 9F3A, 0x9F3A, 9F3A hex
- $B_2 = \{0,1\}$

Si possono scrivere B^h cifre

Una notazione fondamentale è quella posizionale

$$\sum_{i=0}^{n-1} val(d_i)\, B^i$$

Esempio: 1984_{10}, $n=4$ $\quad 4\cdot 10^0 + 8\cdot 10^1 + 9\cdot 10^2 + 1\cdot 10^3$

(3 2 1 0)

Es. 10100

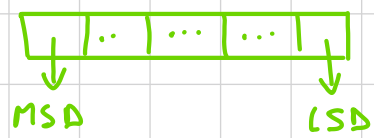
$B_{10} = 1\cdot 10^2 + 1\cdot 10^4 = 10100$; succ: 10101, prec: 10099

$B_8 = 1\cdot 8^2 + 1\cdot 8^4 = 4160$; succ: 4161, prec: 4159

$B_2 = 1\cdot 2^2 + 1\cdot 2^4 = 20$; succ: 21, prec: 19

$B_b \to B_{10}$: Posizionale

$B_{10} \to B_b$: Dato $(N)_{10}$:
1) Dividere N per b
2) Il resto diventa 1ª LSD → la cifra + a dx
3) Se $q=0$ → finito
4) Se $q \neq 0$ → torna a 1



Es.

$(113)_{10} \to B_2, B_8, B_{16}$

.) 113 % 2 = 1
56 % 2 = 0
28 % 2 = 0
14 % 2 = 0
7 % 2 = 1
3 % 2 = 1
1 % 2 = 0
1
⇒ 0110001

.) 113 % 8 = 1
14 % 8 = 6
1 % 8 = 1
0
⇒ 161

.) 113 % 16 = 1
7 % 16 = 7
0
⇒ 71

$B_2 \to B_{16}$; $B_{16} \to B_2$

Caso particolare di quello generale

$16 = 2^4$ ⇒ la i-esima cifra di B^m corrisponde all'i-esimo gruppo di m cifre di B

Es. $(A8FB)_{16} \to B_2$

A = 10 = 1010
8 = 1000
F = 1111
B = 1011

$\Rightarrow (A8FB)_{16} = (1010\ 1000\ 1111\ 1011)_2$

Un numero binario su n bit può essere interpretato come un punto di uno spazio n-dimensionale

DEF. la **Distanza di Hamming** è il numero di posizioni in cui un bit ha un valore diverso tra due numeri binari n_1 e n_2

Es. Distanza di Hamming tra 10**1**1**0**1**0** e 1001 0111 : (4)

CODICE DI GREY

Nel codice di Grey le distanze di Hamming tra un numero binario n e il suo successivo n+1 sono sempre 1. I codici con questa proprietà sono detti a distanza unitaria

OPERAZIONI ARITMETICHE

Si usano riporti e prestiti come con la base 10.

$b_1 = 0011$

$b_2 = 0101$

Somma

```
  1 1 1
  0 0 1 1 +
  0 1 0 1 =
-----------
  1 0 0 0
```

Es. 0101 + 1011 su 4 bit

```
  1 1 1
  0 1 0 1 +
  1 0 1 1 =
-----------
1 0 0 0 0
```

↳ OVERFLOW (n.° non rappresentabile su n bit, servono n+1 bit ALMENO)

MODULO E SEGNO

Per rappresentare i numeri interi con il segno ci sono due modi. Il primo è modulo e segno:

- MSB indica il segno (+ = 0, − = 1)
- n-1 bit indicano il modulo

Esempio:

10011 → N = 19, m.s. = −3

N.B. Per l'elaboratore è inefficiente:

- RIDONDANZA (0 ha 2 codifiche)
- COMPLESSITÀ

COMPLEMENTO A 2 (ANCHE NEL C2 IL 1° BIT INDICA INDIRETTAMENTE IL SEGNO)

Il metodo usato oggi è il C2

Se $N \geq 0$ lo codifico come quello N su $n-1$ bit e pongo $msb = 0$

Se $N < 0$ lo codifico come il naturale $2^n - |N|$ su N bit

Es. $n = 4$ bit

$N = 5 \rightarrow$ su 3 bit 0101

$N = -5 \rightarrow 2^4 - 5 = 16 - 5 = 11 = 1011$

$N = 8 \rightarrow$ overflow su 3 bit $\rightarrow$ servono 5 bit

$N = -1 \rightarrow 2^4 - 1 = 15 = 1111$

$N = -8 \rightarrow 2^4 - 8 = 8$ su 4 bit $= 1000$

$N = -11 \rightarrow 2^4 - 11 = 5$ su 4 bit $= 0101$

→ è 5 o -11? Chi sta fuori dall'intervallo di rappresentabilità $[-2^{n-1}; 2^{n-1} - 1]$ su n bit non può essere rappr. con C2

Metodo alternativo:

1) Verifico rappresentabilità
2) Converto N in binario
3) C1 (inverto tutti i bit)
4) Sommo 1 in binario

1) Controllo intervallo
2) Trasformo in binario
3) Inverto tutti i bit fino all'ultimo 1 (escluso) e lascio uguale il resto

Es. -6 su 3 bit

$n = 3 \Rightarrow [-4; 3] \Rightarrow$ non si può fare

Es. -6 su 4 bit

$n = 4 \Rightarrow [-8; 7]$

$6 = 0110 \Rightarrow 1001$

$$\begin{array}{r} {\scriptstyle 1\ \ \ } \\ 1001 + \\ 0001 = \\ \hline 1010 \end{array}$$

Es. $4 - 5$ $n = 4$ (si fa $4 + (-5)$)

$4 = 0100$

$5 = 0101 \rightarrow 1011$

$$\Rightarrow \begin{array}{r} 0100 + \\ 1011 = \\ \hline 1111 \end{array}$$

Es. $7 - 1$ $n = 4$

$7 = 0111$

$1 = 0001 \rightarrow 1111$

$$\begin{array}{r} {\scriptstyle 1\ 1\ 1\ 1} \\ 0111 + \\ 1111 = \\ \hline (1)0110 \end{array}$$

↳ si ignora l'ultimo riporto

Il risultato di una somma in C2 deve stare nell'intervallo di rappresentabilità sennò OVERFLOW (succede solo con numeri dello stesso segno)

CONTROLLARE GLI ULTIMI 2 RIPORTI → SE DIVERSI C'È OF

NUMERI $\mathbb{R}$

Non li usiamo per contare ma per misurare, inoltre non sono enumerabili

Come si rappresentano? Introduco la virgola e associo alle posizioni alla sua destra indici negativi

Esempio: $(11.001)_2 = 1\cdot 2^1 + 1\cdot 2^0 + 0\cdot 2^{-1} + 0\cdot 2^{-2} + 1\cdot 2^{-3}$

$B_{10} \to B_2$ Dato $(I.F)_{10}$ da conv. in B_2:
1) I si converte in bin IN
2) Moltiplicare .F per 2
3) La parte intera del risultato diventa la 1^a MSD
4) Tornare a 2 considerando la parte frar. al posto di .F

Es. $(4.4375)_{10} \to B_2$

$4 = 100$

$.4375 \cdot 2 = .875$

$.875 \cdot 2 = 1.75$

$.75 \cdot 2 = 1.5$

$.5 \cdot 2 = 1$

$\Rightarrow (100.0111)_2$

TRONCAMENTO: genero cifre fino a quando esaurisco i bit
(arrotondamento verso lo 0: eccesso +, difetto -)

ARROTONDAMENTO: come troncamento ma scelgo come arrotondare

Es. $(101.11011010100)_2 \to 101.111$ eccesso $(101.110 + 000.001)$

$(0.00101001101010100)_2 \to 0.00101$ difetto / troncamento

VIRGOLA FISSA & MOBILE

•) FISSA

$n = n_I + n_F$

Assegno un n.° di bit n_I alla parte intera; $n - n_I = n_F$

Es. 11.001 $n=5$ $n_I=2$ $n_F=3$

La virgola non cambia mai posizione

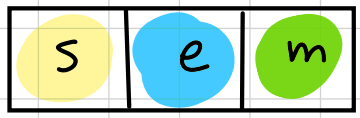
Il n.° max rappr. è $2^{n_I} - 1 + \sim 1 \cong 2^{n_I}$ $(11.111)_2 = (3,875)_{10} \cong 4$

Il n.° + vicino allo 0 è 2^{-n_F} → PRECISIONE

↳ differenza di valore tra 2 rappr. successive

•) MOBILE

La virgola cambia posizione. Si usano dei bit per indicarla



→ mantissa frac. su n_m bit in virgola fissa

$(-1)^s \times m \times 2^e$ → notazione scientifica

$1 + n_e + n_m = n$

↓ segno: 1 bit, 0 +, 1 −

↓ esponente intero su n_e bit

↳ fa muovere la virgola!

Quella in virgola mobile è la forma normalizzata (se parte intera di m ha una sola S.D. ⇒ 1 (...))

Es. $1101 \cdot 2^{-1} \;?\; 10.11 \cdot 2^1$

$1.101 \cdot (2^{-1} \cdot 2^3) \quad 1.011 \cdot (2^1 \cdot 2^1)$

$1.101 \cdot 2^2 \;>\; 1.011 \cdot 2^2$

Con la notazione in virgola mobile, a seconda della posizione della virgola, possiamo rappresentare o numeri molto grandi o molto più piccoli, in quanto la precisione è variabile

STANDARD IEEE 754

L'IEEE-SA Standard n.° 754 for Floating-Point Arithmetic regola la rappr. in virgola mobile nei moderni calcolatori

- Formato a precisione singola (FLOAT 32) → s=1, e=8, m=23
- .. .. .. doppia (DOUBLE 64)

i) Se 0<E<255 (binario naturale) ⇒ 32 bit codificano un normalizzato

- •) s è il segno
- •) m è mantissa $(1.m)_2$
- •) E è esponente a cui è stato sommato 127 ⇒ e = E−127 ↳ BIAS

} ⇒ $(-1)^s \cdot 1.m \cdot 2^e$

Es. 3,25 in IEEE 754 s.

$(3)_{10} = (11)_2$

$0.25 \cdot 2 = 0.5 = 0$ ↓ ⇒ .01 ⇒ $(11.01)_2 = 1.101 \cdot 2^1$

$.5 \cdot 2 = 1$

⇒ s=0, E = 1+127 = 128 = $(10000000)_2$, m = 1010000... ⇒ 01000000010100(..)

è positivo — fino a 32 bit

Es. -5.828125

$(5)_{10} = (101)_2$

$0.828125 \cdot 2 = 1.656250 = 1$

$.656250 \cdot 2 = 1.312500 = 1$

$.312500 \cdot 2 = 0.625000 = 0$ $\Rightarrow 0.110101$

$.625 \cdot 2 = 1.3 = 1$

$.3 \cdot 2 = 0.6 = 0$

$.6 \cdot 2 = 1.2 = 1$

arrivo allo stesso n.° di cifre decimali del B_{10}

$\Rightarrow 101.110101 = 1.01110101 \cdot 2^2$

$\Rightarrow s = 1$

$e = 2 + 127 = 129 \rightarrow (129)_{10} = (10000001)_2$

$m = .01110101$

129 % 2 =	1
64 % 2 =	0
32 % 2 =	0
16	0
8	0
4	0
2	0
1	1
0	

$\Rightarrow$ 1 100|0000|1011|1010|1000|0000|0000|0000

C 0 B A 8 0 0 0

$\Rightarrow$ 0xC0BA8000

Es. -30.375

$30 \% 2 = 0$
$15 \% 2 = 1$
$7 \% 2 = 1$
$3 \% 2 = 1$
$1 \% 2 = 1$
0

$(30)_{10} = (11110)_2$

$\Rightarrow 11110.010 = 1.1110010 \cdot 2^4$

$.375 \cdot 2 = 0.75 = 0$
$.75 \cdot 2 = 1.5 = 1$
$0.5 \cdot 2 = 0.9 = 0$

$(0.375)_{10} = (010)_2$

$e = 4 + 127 = 131$

131 % 2 =	1
65 % 2 =	1
32 % 2 =	0
16	0
8	0
4	0
2	0
1	1
0	

$\Rightarrow (131)_{10} = (10000011)_2$

$\Rightarrow$ 1 1000|0011|1110|0100|0000|0000|0000|0000

$\Rightarrow$ 0xC1F20000

ii) Se $E=0$ e $m \neq 0$ $\Rightarrow$ sub-normalizzato

-) s = segno
-) m = frac. $(0.m)_2$
-) $e = -126$

$\Rightarrow (-1)^s \cdot 0.m \cdot 2^{-126}$

- $0 < E < 255$ ($e \in [-126, 127]$)
  ↳ normalizzato
- $m = 0$, $E = 0$
  ↳ ± 0 (a seconda di s)
- $m = 0$, $E = 255$
  ↳ $\pm \infty$ (.. ..)
- $m \neq 0$, $E = 255$
  ↳ NaN
- $m \neq 0$, $E = 0$
  ↳ subnormalizzato

Es. n.° normalizzato + vicino a 0?

$v_0 = 1.000(\dots) \cdot 2^{-126} \cong 2^{-126} \cong 1.17 \cdot 10^{-38}$

Il successivo?

$v_1 = 1.000(\dots)001 \cdot 2^{-126} \cong (1 + 2^{-23}) \cdot 2^{-126} = 2^{-126} + 2^{-149} \cong 1.17 \cdot 10^{-38} + 1.4 \cdot 10^{-45}$

n.° de-normalizzato + vicino a 0?

$\delta_0 = 0.000(\dots)1 \cdot 2^{-126} = 2^{-149} \cong 1.4 \cdot 10^{-45}$

Il successivo?

$\delta_1 = 0.00(\dots)010 \cdot 2^{-126} = 2 \cdot \delta_0 = 2^{-148} \cong 2{,}8 \cdot 10^{-45}$

Le operazioni in virgola mobile richiedono hw + complesso e quindi di solito si usa un'unità dedicata negli elaboratori (F.P.U.). Il n.° di operazioni in virgola mobile che una CPU può svolgere nel tempo è una metrica della sua potenza di calcolo (FLOPS, floating-point operations per second)

ELABORAZIONI LOGICHE

* cambiando NAND con NOR cambia anche AND con OR

Le elaborazioni logiche sono operazioni matematiche definite da un'algebra detta booleana.

↳ algebra che comprende simboli e valori binari su cui effettuiamo oper. logiche

$B = \{0, 1\}$

$a \in B$

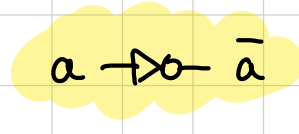
$f: B^{(n)} \to B$, $f(a_1, a_2, .., a_n) \in B$ (n → n.° di variabili)

NOT ($\bar{a}$, $\neg a$, NOT(a))

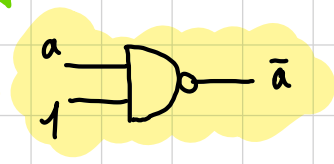
Negazione logica di una espressione bool

Ha precedenza su AND e OR

a	$\bar{a}$
1	0
0	1



implementazione con solo porte NAND

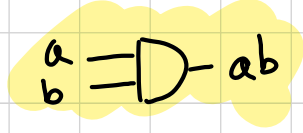


AND (ab, $a \wedge b$, a AND b)

Congiunzione logica tra due espressioni bool

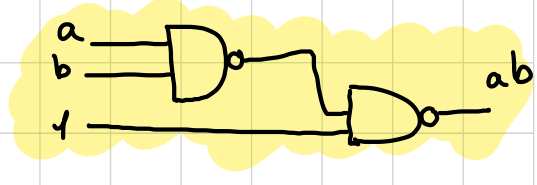
Prodotto logico, min

Ha precedenza su OR



a	b	ab
1	1	1
1	0	0
0	1	0
0	0	0

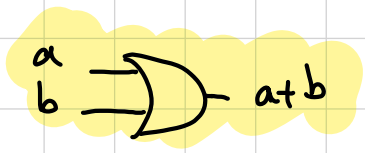
implementazione con solo porte NAND



OR $(a+b,\ a \vee b,\ a \text{ OR } b)$

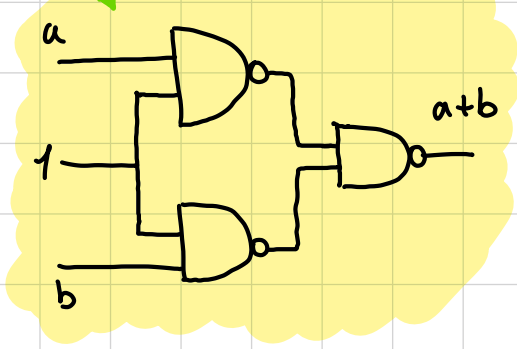
Disgiunzione logica tra due espressioni bool

Somma logica, max



a	b	a+b
1	1	1
1	0	1
0	1	1
0	0	0

implementazione con solo porte NAND



Esempio: $f(a,b,c) = a + \bar{b}\,c$

PRINCIPIO DI DUALITÀ: Se un'uguaglianza booleana è valida $\Rightarrow$ lo è anche la sua duale

La duale si ottiene:

1) Scambiando AND $\Leftrightarrow$ OR
1) Scambiando $1 \Leftrightarrow 0$

Esempio: $a + \bar{a} = 1 \rightarrow a\bar{a} = 0$

$(a\bar{b}) + 1 = 1 \rightarrow (a + \bar{b})\,0 = 0$

PROPRIETÀ

- Identità ($1a = a$; $0 + a = a$)
- Elem. nullo ($0a = 0$; $1 + a = 1$)
- Idempotenza ($aa = a$; $a + a = a$)
- Inverso ($a\bar{a} = 0$; $a + \bar{a} = 1$)
- Commutativa ($ab = ba$; $a + b = b + a$)
- Associativa ($(ab)c = a(bc)$; $(a+b)+c = a+(b+c)$)
- Distributiva ($a(b+c) = ab + ac$; $a + bc = (a+b)(a+c)$)
- Assorbimento I ($a(a+b) = a$; $a + ab = a$)
- Assorbimento II ($a(\bar{a}+b) = ab$; $a + \bar{a}b = a+b$)
- De Morgan ($\overline{ab} = \bar{a} + \bar{b}$; $\overline{a+b} = \bar{a}\bar{b}$)

Es.

$\overline{\bar{a}+c} + \bar{c} + b\left(\overline{c + c\bar{b}}\right) =$

$= \bar{a}\bar{c} + \bar{c} + b\left(\overline{c+c\bar{b}}\right) =$ De Morgan

$= \bar{c} + b\left(\overline{c+c\bar{b}}\right) =$ Ass. I

$= \bar{c} + b\bar{c}\left(\overline{1+\bar{b}}\right) =$ Distributiva

$= \bar{c} + b\bar{c} =$ El. nullo e id.

$= \bar{c}$ Ass. I

Es. $\overline{\overline{(\bar{a}\bar{b}+\bar{c})}\,\overline{\bar{a}\bar{b}}} + \overline{\bar{a}bc} =$

$= \overline{\bar{a}\bar{b}}\,\bar{c} + \overline{\bar{a}\bar{b}}\,c =$ ↘ Ass II

$= \overline{\bar{a}\bar{b}}\,(\bar{c}+c) =$ ↘ Elem. nulla

$= \overline{\bar{a}\bar{b}} =$ ↘ Inverso

$= \bar{\bar{a}} + \bar{b} =$ ↘ De Morgan

$= a + \bar{b}$

Es.

$\bar{A}(A+B) + \bar{C} + BC =$

$= \bar{A}A + \bar{A}B + \bar{C} + BC =$

$= 0 + \bar{A}B + \bar{C} + B =$

$= (\bar{A}+1)B + \bar{C} =$

$= 1B + \bar{C} = B + \bar{C}$

Es.

$\overline{A + A\bar{B} + CD} =$

$= \overline{A(1+\bar{B}) + CD} =$

$= \overline{A + CD} =$

$= \bar{A}(\overline{C+D}) =$

Es.

$\overline{(\overline{A+B})C} =$

$= \overline{(\bar{A}\bar{B})C} =$

$= \bar{\bar{A}}\bar{\bar{B}} + \bar{C} =$

$= AB + \bar{C}$

Es.

$\bar{B}\bar{C} + AB\bar{C} =$

$= \bar{C}(\bar{B} + AB) =$

$= \bar{C}(\bar{B} + A) =$ Ass. II

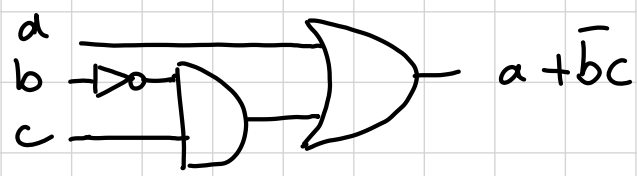
$= \bar{B}\bar{C} + A\bar{C}$

CIRCUITI COMBINATORI

Un circuito combinatorio combina gli input per ottenere un output

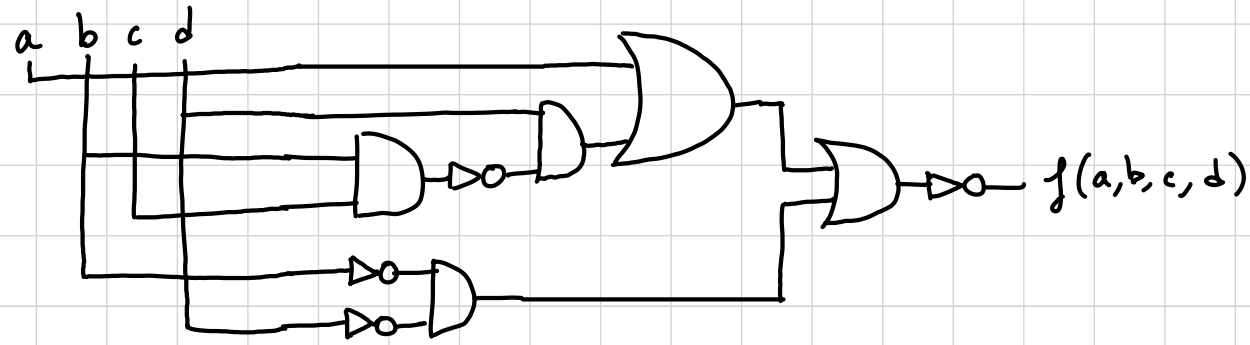
Esempio:

$f(a,b,c) = a + \bar{b}c$ →



1) L'elaborazione procede solo sx → dx
2) A parità di input l'uscita è sempre la stessa

Es. $f(a,b,c,d) = \overline{a + \overline{bc}d + \bar{b}\bar{d}}$



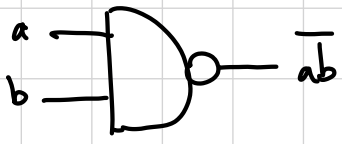
Data una funzione logica $f: B^n \to B$:

1) TT (truth table): 2^n righe, n+1 colonne (1)
2) Espress. bool.: descrizione formale manipolabile con Algebra Booleana (infinite)
3) C.C.: implementazione hw della funzione (infiniti)
   (circuito combinatorio)

OPERATORI COMPOSTI

NAND

È definita porta universale perché può implementare NOT, AND, OR

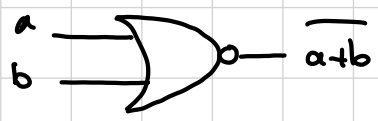


a	b	$\overline{ab}$
1	1	0
1	0	1
0	1	1
0	0	1

- $\bar{a} = \overline{1a}$
- $ab = \overline{\overline{ab}} = \overline{\overline{ab}1}$
- $a+b = \overline{\bar{a}\bar{b}} = \overline{\overline{1a}\,\overline{1b}}$

NOR

È definita porta universale perché può implementare NOT, AND, OR

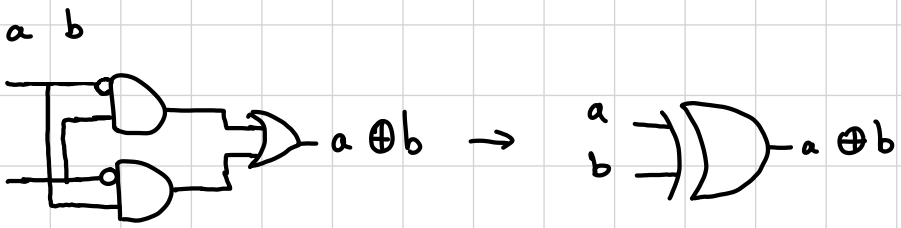


a	b	$\overline{a+b}$
1	1	1
1	0	0
0	1	0
0	0	0

- $\bar{a} = \overline{0+a}$
- $a+b = \overline{\overline{a+b}} = \overline{0 + \overline{a+b}}$
- $ab = \overline{\bar{a}+\bar{b}} = \overline{\overline{0+a} + \overline{0+b}}$

XOR ($\oplus$)

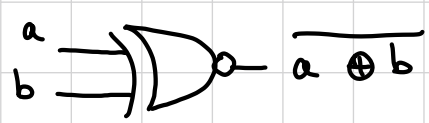
"eXclusive OR"



a	b	$a \oplus b$
1	1	0
1	0	1
0	1	1
0	0	0

- se bit diversi $\Rightarrow$ 1
- o 1
- più avanti

XNOR



a	b	$\overline{a \oplus b}$
1	1	1
1	0	0
0	1	0
0	0	1

- se bit uguali $\Rightarrow$ 1

PRIMA FORMA CANONICA

Esempio:

a	b	c	y
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

$y = 1$ se:

- $a=0, b=0, c=0 \Rightarrow \bar{a}=1, \bar{b}=1, \bar{c}=1 \Rightarrow \bar{a}\bar{b}\bar{c}$
- $a=0, b=1, c=0 \Rightarrow \bar{a}=1, \bar{b}=0, \bar{c}=1 \Rightarrow \bar{a}b\bar{c}$
- $a=1, b=0, c=1 \Rightarrow \bar{a}=0, \bar{b}=1, \bar{c}=1 \Rightarrow a\bar{b}c$
- $a=1, b=1, c=1 \Rightarrow \bar{a}=0, \bar{b}=0, \bar{c}=0 \Rightarrow abc$

$\Rightarrow y = \dots\dots\dots$

Viene detta 1^a F.C.

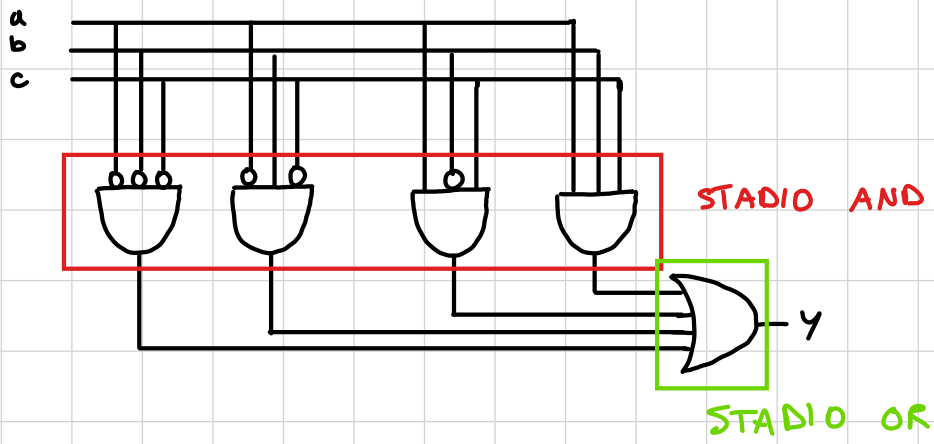
MINTERMINI'

1) Definiti come AND Tra Tutti gli input. Ogni variabile compare 1 volta
2) 1 mint. per ogni config. input in cui $y=1$
3) Se in config. input var $= 0$, in mint. sarà negata, semnó naturale
4) $y = \sum_{i=1}^{n} m_i$, $n =$ n.° mint. (OR)

$\Rightarrow I^a$ F.C. = **SOP** (Sum of products)

Riassumendo:

1) Identificare i mint. di y
2) Combinarli con OR
3) Ogni mint. → AND a n input → le porte a n ingressi si ottengono collegando in serie n-1 porte a 2 ingressi



4) Se un termine della SOP non ha Tutte le variabili => IMPLICANTE (sintetizza dei mint.)

SECONDA FORMA CANONICA

Esempio:

a	b	c	y
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

$\rightarrow a=0, b=0, c=1 \Rightarrow \bar{a}=1, \bar{b}=1, c=1 \Rightarrow \bar{a}\,\bar{b}\,c$

$\rightarrow a=0, b=1, c=1 \Rightarrow \bar{a}=1, b=1, \bar{c}=1 \Rightarrow \bar{a}\,b\,\bar{c}$

$\rightarrow a=1, b=0, c=0 \Rightarrow a=1, \bar{b}=1, \bar{c}=1 \Rightarrow a\,\bar{b}\,\bar{c}$

$\rightarrow a=1, b=1, c=0 \Rightarrow a=1, b=1, \bar{c}=1 \Rightarrow a\,b\,\bar{c}$

$\Rightarrow y = \overline{(\dots)}$

Viene detta II^a F.C.

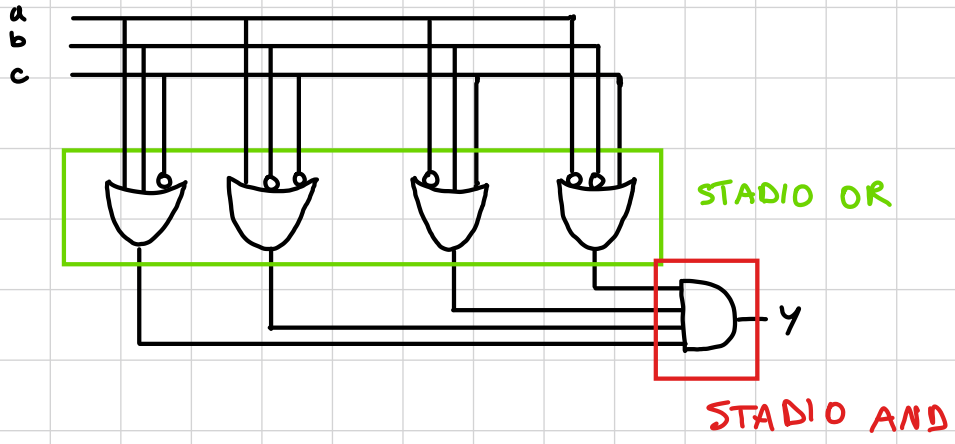
MAXTERMINI:

1) Definiti come OR tra tutti gli input. Ogni variabile compare 1 volta
2) 1 maxt. per ogni config. input in cui $y=0$
3) Se in config. input var = 1, in maxt. sarà negata, se no naturale
4) $y = \prod_{i=1}^{n} M_i$, n = n.° mint. n = n.° maxT (AND)

$\Rightarrow II^a$ F.C. = POS (Product of Sums)

Riassumendo:

1) Identificare i maxt di y
2) Combinarli con AND
3) Ogni maxT → OR a n input → le porte a n ingressi si ottengono collegando in serie n-1 porte a 2 ingressi

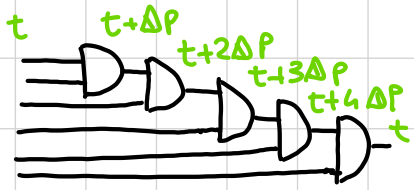


VALUTARE COSTI E PRESTAZIONI

Da TT a C.C. → SOP e POS + veloci
↳ hanno però limiti fisici

PARAGONATION DELAY: ∀ porta logica se al $t=t_i$ gli input cambiano, l'uscita non commuta subito, ma dopo $t+\Delta p$

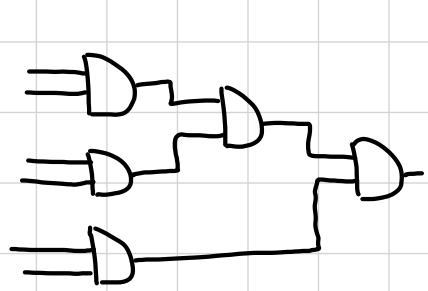
FAN-OUT LIMITATO: il n.° di ingressi a cui posso collegare un uscita è limitato ⇒ + ingressi → + Δt di commutazione.



→ Δp = ritardo

→ Camm. Crit. = n-1 (n ingressi)
è di una porta a n ingressi

Lunghezza max in un C.C. è **CAMMINO CRITICO**
↳ metrica per calcolare le performance del C.C.



→ Camm. Crit. $\log_2 n$

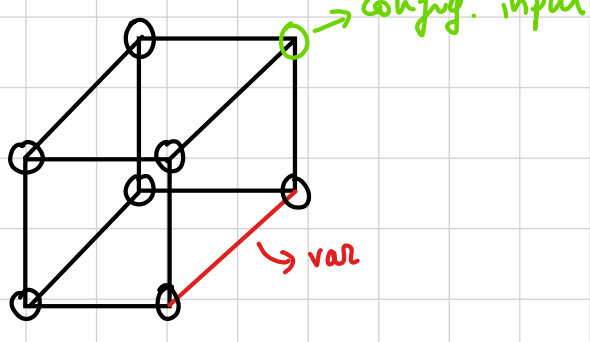
↳ Area occupata, n.° porte, ΔE dissipata, facilità interpretazione

MAPPE DI KARNAUGH

Per semplificare un C.C. → agire su espressione bool
↳ Metodo meccanico/grafico per semplificare espressioni bool basato su implicanti
↳ ESPRESSIONE MINIMA GARANTITA!

In Y con n var ogni config. input è rappresentata da stringa di n-bit

a	b	c	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1



Dati 2 vertici mint. ⇒ il lato che li collega permette di trovare un implicante semplificando

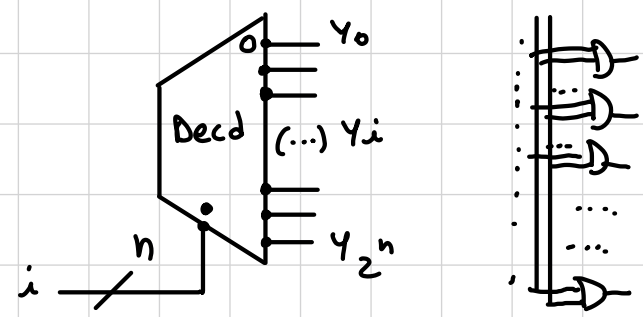
Per usare le mappe serve una rappr. grafica ma un SW può lavorare nello spazio delle adiacenze in n dimensioni

1) In ogni direzione usiamo codifica di Grey
2) Scriviamo il valore corrispondente
3) Formiamo rettangoli (anche sovrapposti, anche wraparound) che racchiudano gli 1 (A max, A % 2 == 0)
4) Per ogni rettangolo scrivo l'implicante associato includendo var che non cambiano di valore quando percorro ∀ direz.
5) Metto OR

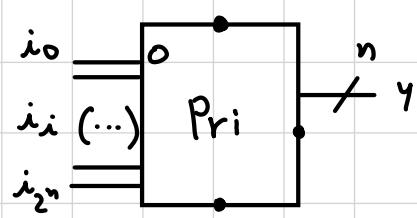
BLOCCHI FUNZIONALI

Alcuni circuiti si possono modularizzare perché formati da sotto-circuiti notevoli

1) **Decoder** Ca.Cr. = n-1 complessità = $2^n(n-1)$
   n input, 2^n output
   ↳ n bit $\in [0, 2^n-1]$ =
   Se input = i ⇒ i-esima uscita = 1
   Converte un valore nel suo codice One-hot
   (101 = 00000100)
   Ogni uscita è una funzione logica con un solo mint.
   TT come una sorta di matrice diagonale

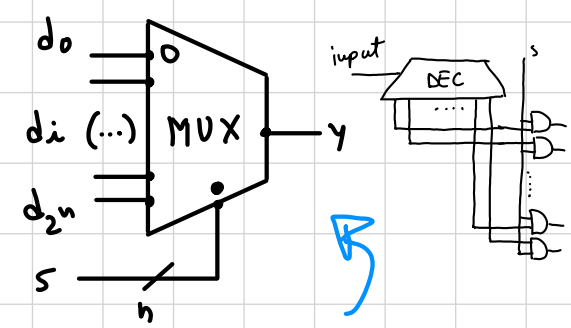


2) **Encoder** CaCr e Co dipendono da y
   2^n input, n output
   Solo un input deve essere 1



3) **Multiplexer**
   $2^n + n$ input, 1 uscita
   ↳ segnale di controllo/selezione (un valore bin. che dice quale linea dati passa in y)
   linee dati (1 bit)

Esempio. se s=01 (1 su 2 bit) ⇒ $y = d_1$
se s=11 (3 su 2 bit) ⇒ $y = d_3$



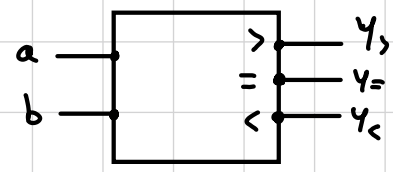
Decoder per il segno + and per gli ingressi e poi un or complessivo

= con xnor

> e < con and (2 inp. e 1 neg.)

4) Comparatore

2 input su n bit, 3 output (•) ">" $a > b \to y_>$
Solo 1 uscita = 1 •) "=" $a == b \to y_=$
•) "<" $a < b \to y_<$)



→ per "=" si usa XNOR

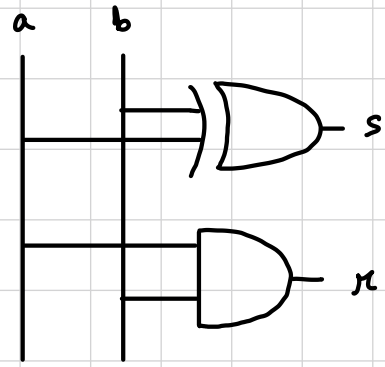
Un comparatore da n bit si può ottenere usando due comparatori da $\frac{n}{2}$ bit

5) HALF ADDER

Somma binaria di 2 numeri su 1 bit

a	b	s	r
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

↓ XOR ↓ AND



CaCr = 1 ; Co = 2

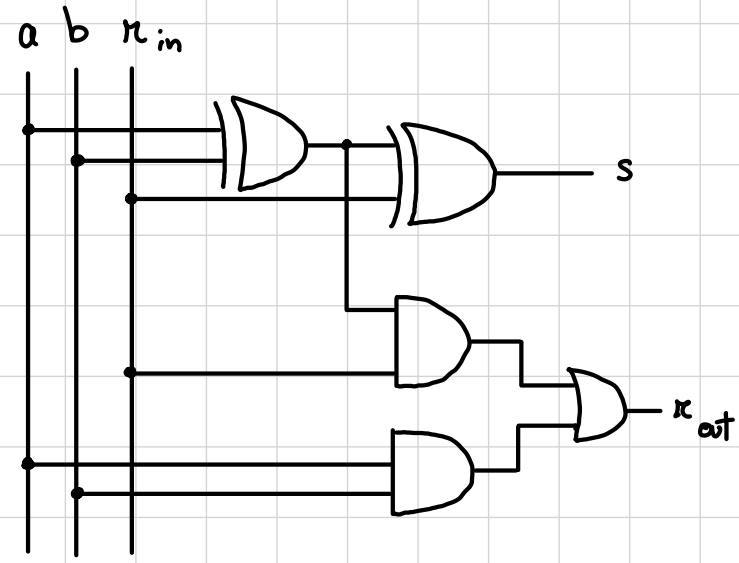
6) FULL ADDER

È un HA con un riporto aggiuntivo ai 2 bit da sommare

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

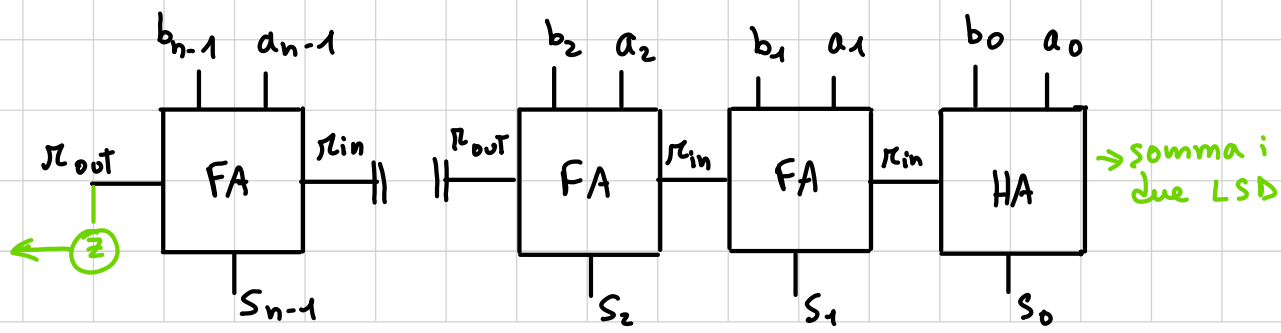
$s = \bar{a}b\bar{r}_{in} + a\bar{b}\bar{r}_{in} + \bar{a}\bar{b}r_{in} + abr_{in} = r_{in} \oplus (a \oplus b) = r_{in} \oplus a \oplus b$

$r_{out} = ab\bar{r}_{in} + \bar{a}br_{in} + a\bar{b}r_{in} + abr_{in} = ab + r_{in}(a+b)$



CaCr = 3 ; Co = 5

Per sommare numeri $a = a_{n-1}\, a_{n-2} \dots a_1 a_0$ e $b = b_{n-1}\, b_{n-2} \dots b_1 b_0$ su n bit si possono collegare in serie diversi moduli (gli adder non lavorano in parallelo)



$a_{n-1}\, a_{n-2} \dots a_1 a_0 +$
$b_{n-1}\, b_{n-2} \dots b_1 b_0 =$
$s_{n-1}\, s_{n-2} \dots s_1 s_0$

CaCr HA = 1
CaCr FA = 3
} => CaCr = 3n

Ca(r alto => lentezza → a causa della propagazione dei riporti
=> scrivo un'espressione diretta per ogni riporto con un'implementazione di soli FA

dipende da r_i

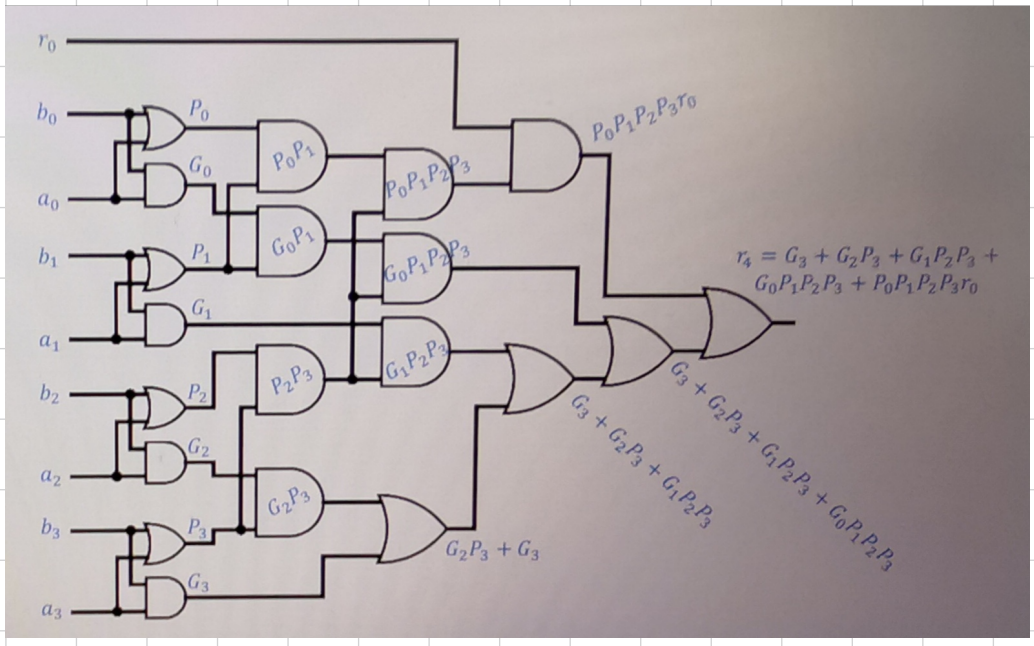
In FA_i $\Rightarrow r_{i+1} = a_i b_i + r_i (a_i + b_i)$

$a_i b_i$ → G_i = termine di generazione

$a_i + b_i$ → P_i = termine di propagazione

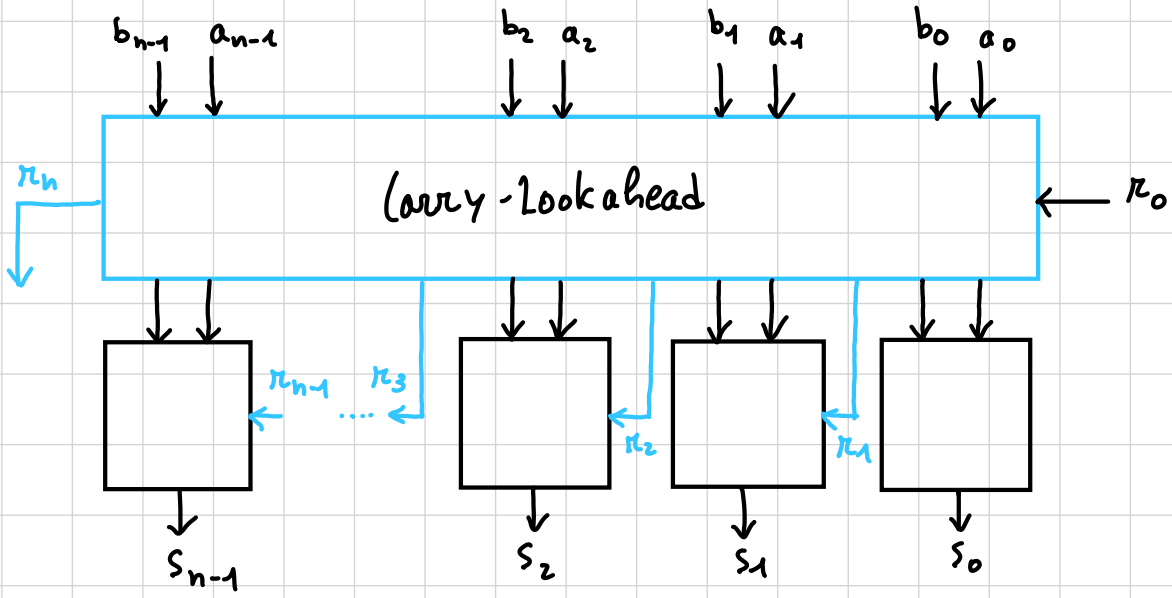
$$\Rightarrow r_1 = a_0 b_0 + r_0 (a_0 + b_0) = G_0 + r_0 P_0$$
$$r_2 = a_1 b_1 + r_1 (a_1 + b_1) = G_1 + (G_0 + r_0 P_0) P_1 = G_1 + G_0 P_1 + r_0 P_0 P_1$$
$$r_3 = a_2 + b_2 + r_2 (a_2 + b_2) = G_2 + (G_1 + G_0 P_1 + r_0 P_0 P_1) P_2 = G_2 + G_1 P_2 + G_0 P_1 P_2 + r_0 P_0 P_1 P_2$$
$$\dots$$
$$r_n = G_{n-1} + P_{n-1} G_{n-2} + P_{n-1} P_{n-2} G_{n-3} + \dots + P_{n-1} P_{n-2} \dots P_0 r_0$$

Esempio
$n = 4$

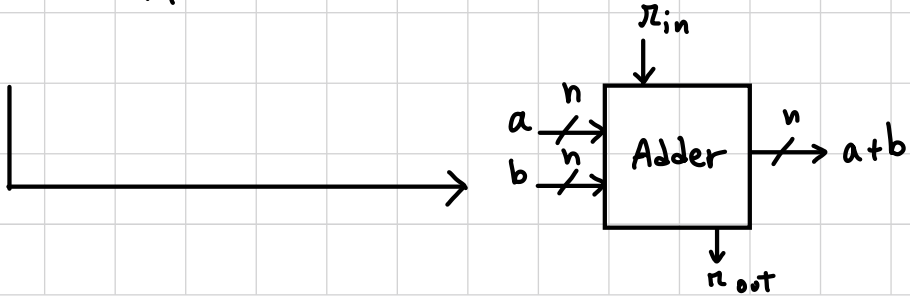


Ca Cr = 6
La metà di quello con la propagazione del riporto con 4 FA

=>

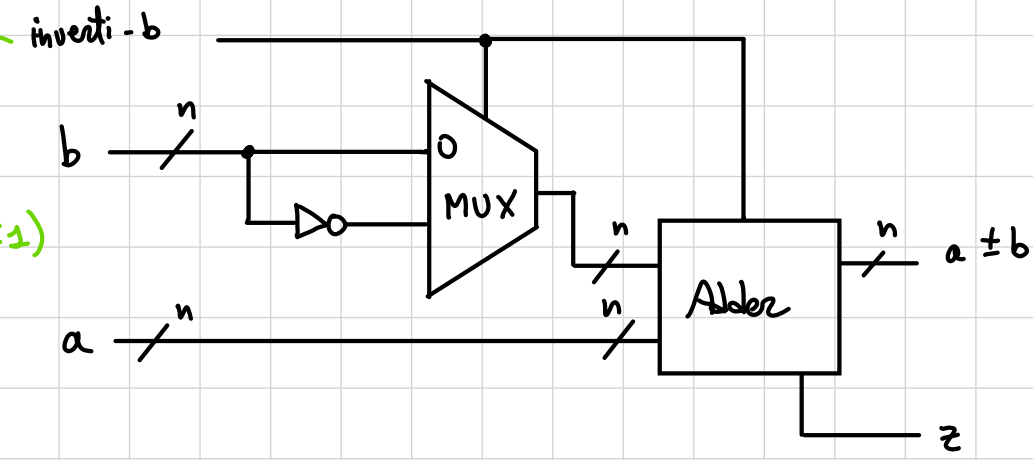


Così tutti i FA lavorano in parallelo e l'unità di look-ahead calcola in anticipo i riporti



SOTTRAZIONE

l'operazione $a-b$ si intende come $a - (2_b \Rightarrow a - \bar{b} + 1$



Bit di controllo

se i-b = 1:

•) Adder riceve $\bar{b}$ (c_1)

•) $c_{in} = 1$

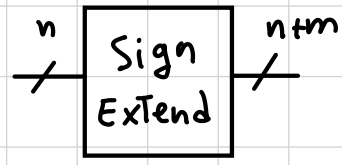
•) z va scartato

↳ => Adder: $a + \bar{b} + 1 = a + (\bar{b} + 1) = a - b$

ESTENSIONE DI SEGNO

Ho un segnale di n bit che do in input a un circuito che riceve n+m bit

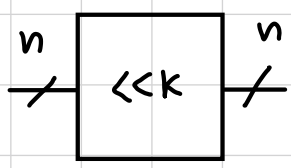
=> replico a sx della MSB la MSB stessa => se il n.° originale era un N o un intero in C2 non si altera il valore



SHIFT

Traslare gli n bit verso sx o dx di k posizioni ($sx << k$; $dx >> k$)

Se non si tolgono 1 e se si rappresenta un N è come moltiplicare per $2^{\pm k}$



Esempio

MOLTIPLICATORE

1) Calcolare prodotti parziali (AND a coppie di bit in posizione shiftata)

2) Somma bit a bit (FA)

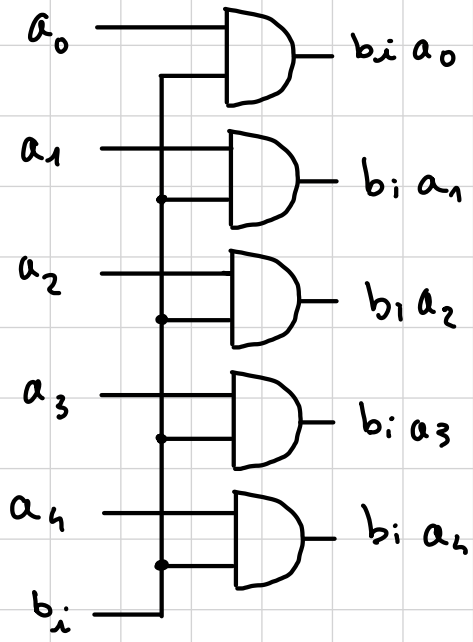
Es. 10111 x
101 =

```
   111
  10111
 00000-
10111--
-------
1110011
```

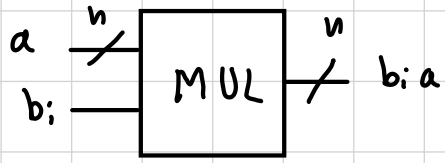
2 n.° su n bit → 1 n.° su 2n bit

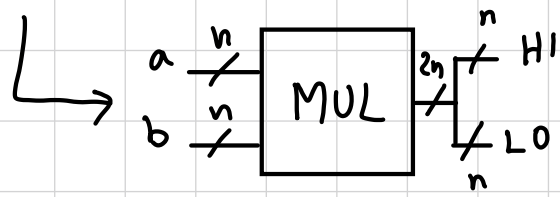
↳ HI : n bit parte alta

LO : n bit parte bassa

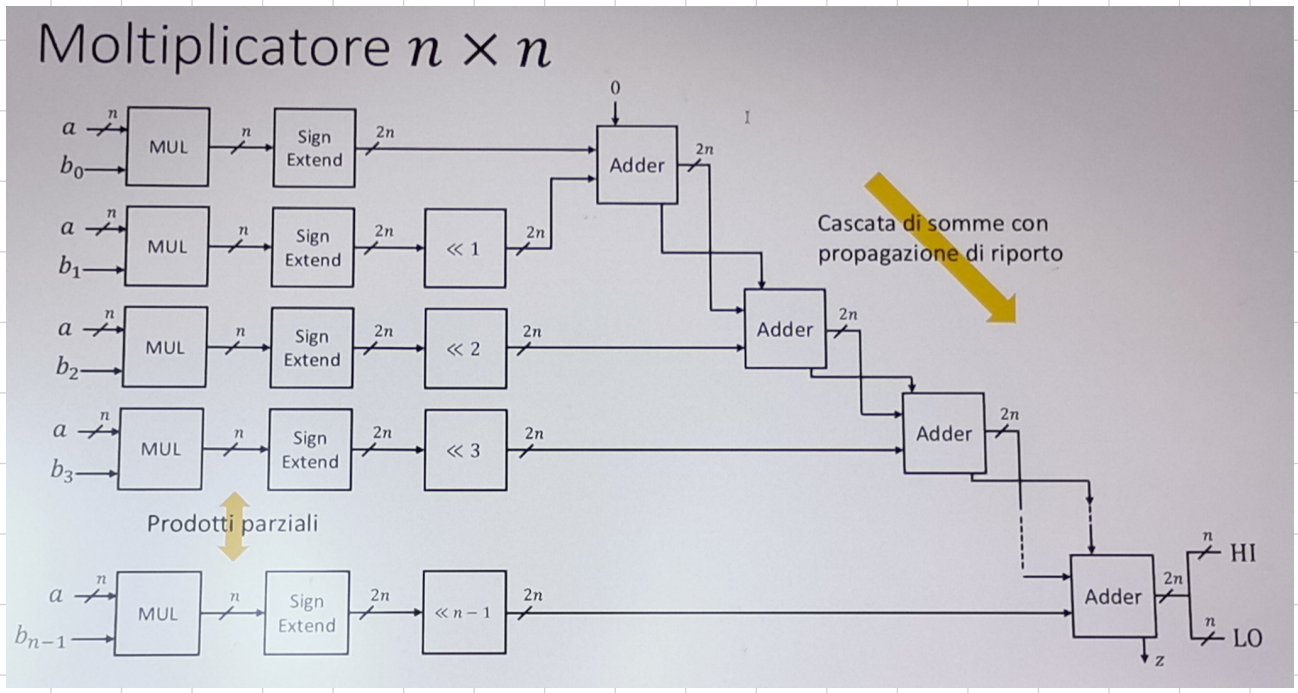


Moltiplicatore 1×n



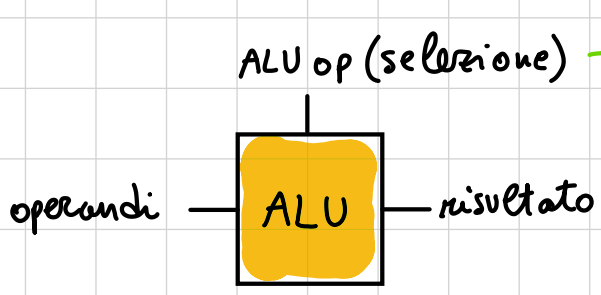


Moltiplicatore n×n



ALU - UNITÀ ARITMETICO-LOGICA → fase di EXECUTE

È la parte della CPU che si occupa di operazioni matematiche e logiche
È un C.C. multifunzione



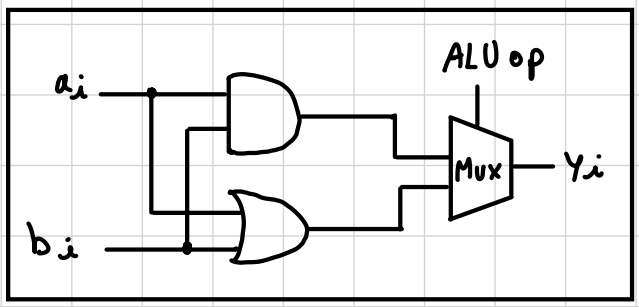
→ identifica la funzione

2 modi per usare ALU:

1) **Modulare** (Per fare ALU da n bit compongo con n ALU da 1 bit
2) **Parallela** (Presi gli operandi ALU calcola tutto quello che riesce, poi un MUX fa uscire)
   ↳ eseguite in hw da un sotto-C.C. dedicato

AND & OR

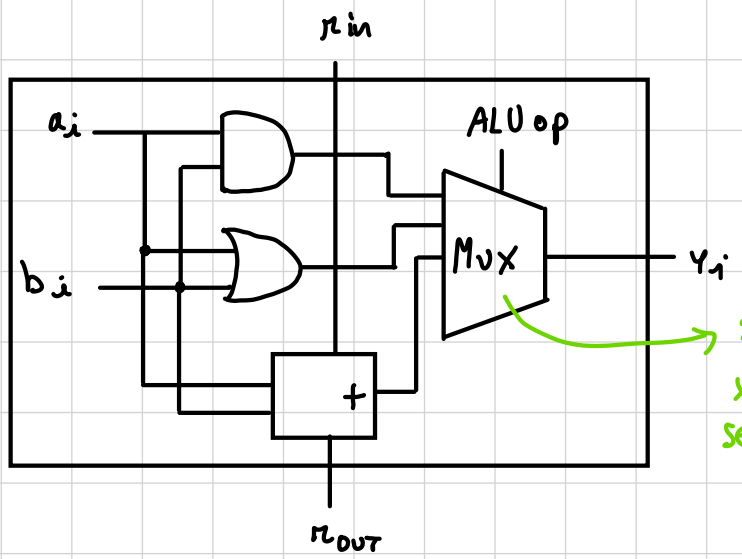
ALU da 1 bit che esegue AND e OR su 1 bit



ALU op	f calcolata
0	a_i AND b_i
1	a_i OR b_i

Calcola sempre sia AND sia OR
Un MUX fa uscire una delle due in base a ALU op

ADD

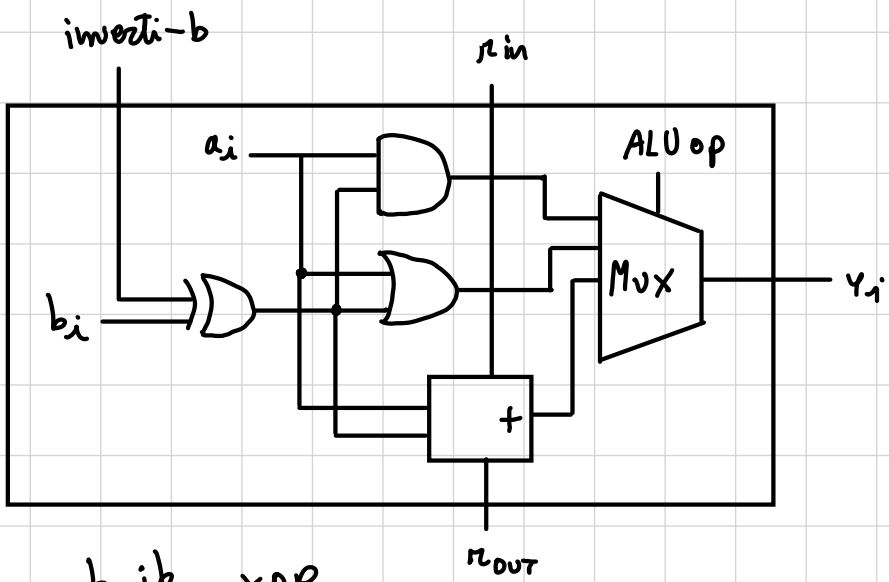


2 bit xkè deve selezionare 1 su 3

Moduli a 1 bit per costruire ALU a n bit

ALU op	f calcolata
00	a_i AND b_i
01	a_i OR b_i
10	$a_i + b_i$
11	/

SUB



b	ib	XOR
0	0	0
0	1	1
1	0	1
1	1	0

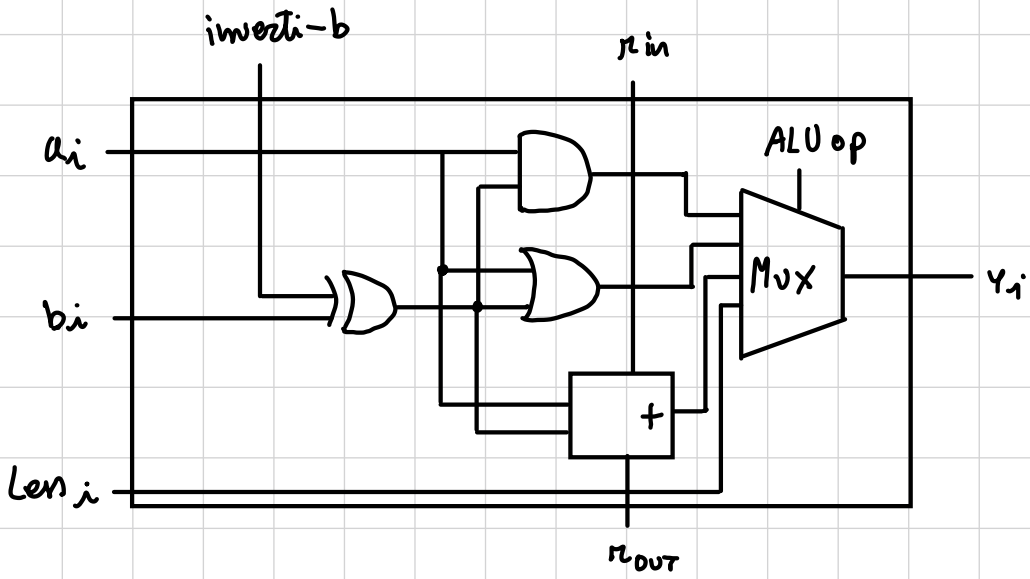
Aggiungendo il bit di controllo "inverti-b" e settandolo, insieme a r_{in}, uguale a 1 posso fare la sottrazione

inverti-b	ALU op	f calcolata
0	00	a_i AND b_i
0	01	a_i OR b_i
0	10	$a_i + b_i$
1	10	$a_i - b_i$

SLT - comparazione (set less than)

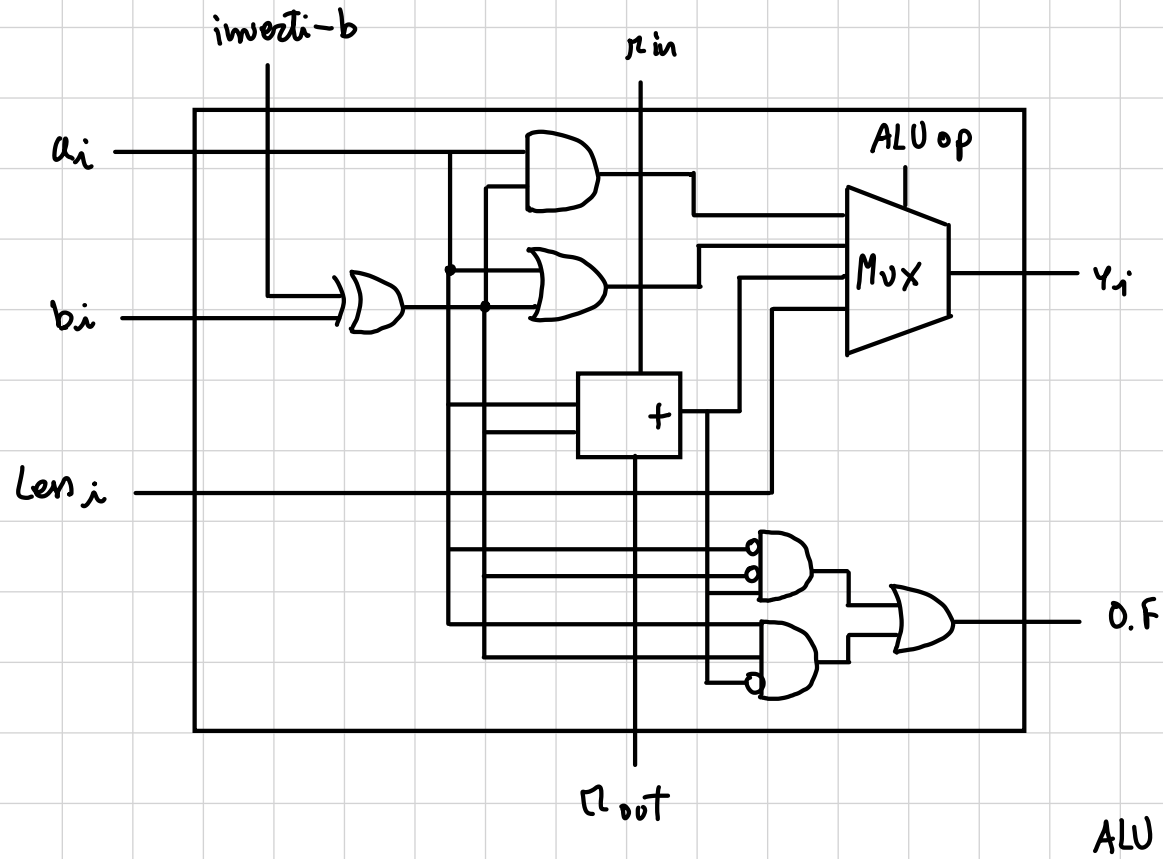
Si può ottenere il confronto di uguaglianza calcolando $a-b$ e vedendo se il risultato è 0

Aggiungiamo $SLT(a,b) = \begin{cases} 1, & a < b \\ 0, & a \geq b \end{cases}$



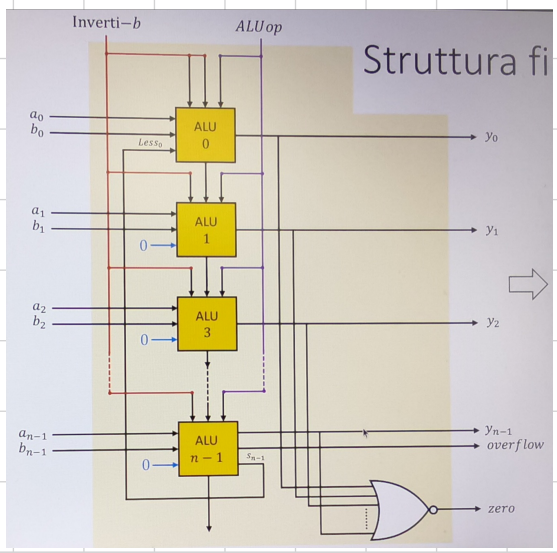
In 32 bit colleghiamo l'output dell' Adder dell' ultima ALU con $Less_i$ input della prima ALU

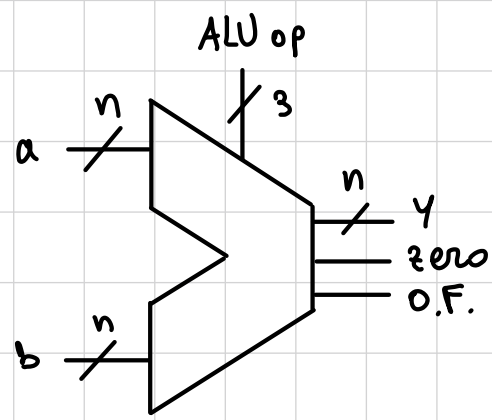
1) $Less_1, Less_2, \dots, Less_{n-1} = 0$
2) $Inverti\text{-}b = 1$
3) $Less_0 = S_{n-1}$ (bit segno di a-b)
4) ALU op = 11



Per verificare la presenza di O.F. implemento il circuito facendo prendere in input i bit di segno di a, b e s (stanno tutti nella ALU_{n-1})
↳ b è $b_{n-1} \oplus inverti\text{-}b$

⇒





ALU op	f calcolata
000	AND
001	OR
010	SUM
110	SUB
111	SLT

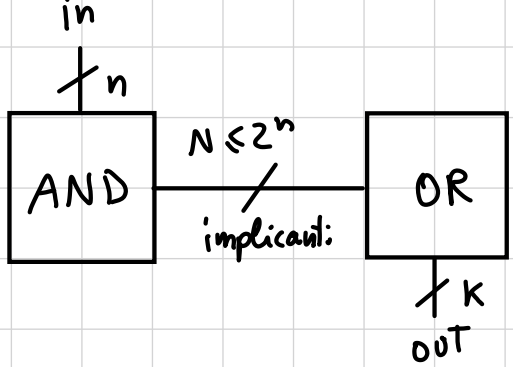
Esempio:

$N_1 = \langle s_1, E_1, m_1 \rangle$
$N_2 = \langle s_2, E_2, m_2 \rangle$
} per fare $N_1 + N_2$ serve ALU-FP (floating point)

$99.5 + 0.85$? $\rightarrow 9.95 \cdot 10^1 + 8.5 \cdot 10^{-1} \rightarrow 9.95 \cdot 10^1 + 0.085 \cdot 10^1 \rightarrow 9.95 + 0.085 =$
$= 10.035 \rightarrow 1.0035 \cdot 10^2$

PLA → Programmable Logic Array
Dispositivo logico per implementare hw di c.c.
È un dispositivo generico su cui si può forgiare la funzione che si vuole
↳ si può ottenere un circuito sop che rappresenta la nostra funzione

Si basa su due stadi: AND e OR
1) calcola N implicanti: (max 2^n, n = input)
2) codifica K uscite
→ parametri dimensionali decisi durante fabbricazione



Per programmare un PLA brucio alcuni collegamenti e ne lascio intatti altri

1) Opportunamente dimensionati, n.° implicanti limitato
2) Con $N \leq 2^n$ implicanti → non si può rappresentare qualsiasi funzione logica
3) Semplificazione necessaria

ROM → Read-Only Memory ↳ Decoder
Come PLA ma l'array AND è hardwired → per generare tutti i mint possibili
L'input è l'indirizzo di una cella di memoria in cui ci sono i K bit delle uscite
L'array OR è programmabile (EEPROM → è riscrivibile)

1) n input → 2^n celle di memoria (sempre)
2) Tutti i mint. ⇒ qualsiasi funzione logica
3) Svantaggio: crescita esponenziale di dimensione con n

CIRCUITI SEQUENZIALI

I circuiti sequenziali sono come i combinatori, con una caratteristica in più: una memoria interna (STATO)

↳ codifica la sequenza di input che ha visto

⇒ output dipende da $x(t)$ e da stato

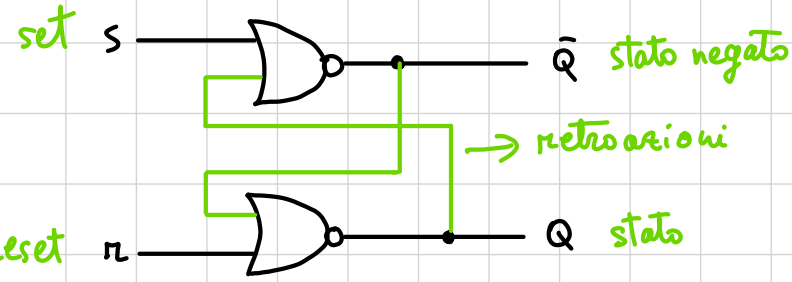
C.C. : $y \leftarrow f(x)$ → a parità di $x \Rightarrow y =$

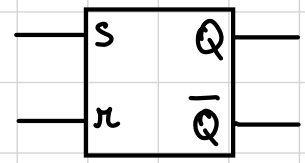
C.S. : $\langle y, s_{next} \rangle \leftarrow f(x, s)$ → a parità di $x \Rightarrow y \neq$ se $s_1 \neq s_2$ → $s \geq 2$

Nei C.S. c'è anche un'altra differenza: l'elaborazione può procedere anche da dx verso sx (RETROAZIONE)

BISTABILE SR (o anche LATCH)

C.S. da cui alla base di tutti





1) $s=1 \rightarrow Q=1$, $\bar{Q}=0$
2) $s=0 \rightarrow Q$ rimane 1 → è lo stato
1) $r=1 \rightarrow Q=0$, $\bar{Q}=1$
2) $r=0 \rightarrow Q$ rimane 0 → è lo stato
1) $s=r=1 \rightarrow Q=\bar{Q}=0$
2) $s=r=0 \rightarrow ?$ → c'è Δt ⇒ •) se $r \rightarrow 0$ prima di $s \rightarrow$ reset $\rightarrow Q=0$, $\bar{Q}=1$
•) se $s \rightarrow 0$ prima di $r \rightarrow$ set $\rightarrow Q=1$, $\bar{Q}=0$

} evitare di farlo perché imprevedibile

s	r	Q	Q_{next}
0	0	0	0
0	1	0	0
1	0	0	1
1	1	0	X → (x→1)
0	0	1	1
0	1	1	0
1	0	1	1
1	1	1	x → (x→1)

Serve un formalismo più generale: Tabella delle transizioni

$$\Rightarrow Q_{next} = T(s, r, Q) = s\bar{r}\bar{Q} + sr\bar{Q} + \bar{s}\bar{r}Q + s\bar{r}Q + srQ =$$
$$= s\bar{r}\bar{Q} + sr\bar{Q} + \bar{s}\bar{r}Q + s\bar{r}Q + s\bar{r}Q + srQ =$$
$$= s(\bar{r}\bar{Q} + r\bar{Q} + \bar{r}Q + rQ) + \bar{r}Q(s + \bar{s}) =$$
$$= s + \bar{r}Q$$

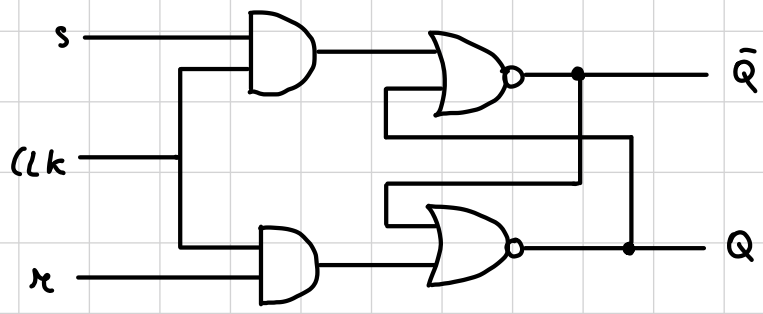
I C.S. servono a:
1) conservare risultati di alcune operazioni
2) affrontare problema di CdC in C.C molto complessi
SOL: Dividere il C.C. in sotto-C.C. e sincronizzarli tra loro

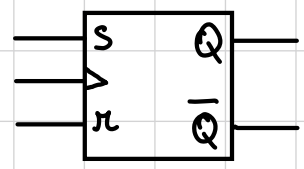
ARCHITETTURE SINCRONE

Siccome un C.C. complesso ha un CdC elevato, prima che le uscite si stabilizzino deve passare molto tempo ⇒ si segmenta il C.C. in sotto-C.C. più semplici collegati in serie tra loro tramite dei C.S.
I C.S. salvano i risultati intermedi del C.C. i e li rendono disponibili per il C.C. $i+1$
Per farlo, però, devono sincronizzarsi con il segnale di CLOCK
Il periodo T del ciclo di clock va dimensionato in modo che le commutazioni avvengano stabilmente
→ $f = \frac{1}{T}$ = frequenza di clock
→ dice ai C.S. quando ricevere un dato e quando trasmetterlo

Nelle architetture sincrone le variazioni di stato avvengono quando sono verificate determinate condizioni sul clock

- LIVELLI : •) stati cambiano ⇒ clock = Alto / basso
  •) stati stabili ⇒ clock = Basso / alto
- FRONTI : •) stati cambiano ⇒ solo quando cambia il clock





Se CLK=0 → non cambia
Se CLK=1 → cambia

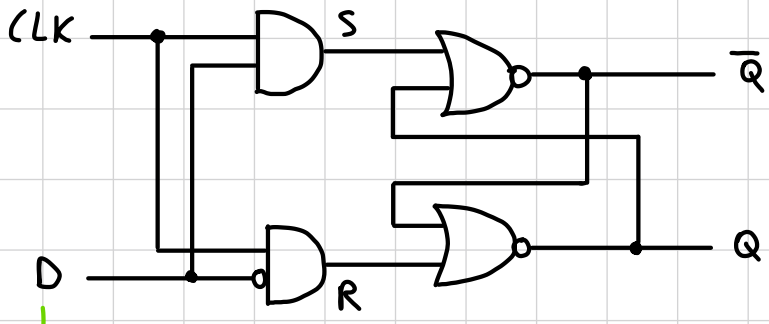
$T(S, R, q, CLK)$ →

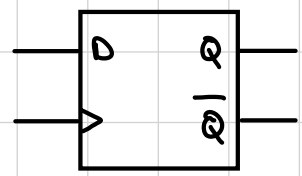
CLKq \ SR	00	01	11	10
00	0	0	0	0
01	1	1	1	1
10	1	0	1	1
11	0	0	1	1

→ $\overline{CLK}\,Q + CLK\,S + CLK\,Q\,\bar{R} =$
$= \overline{CLK}\,Q + CLK(S + \bar{R}\,Q)$

se CLK basso si conserva Q
se CLK → come bistabile SR asyn

LATCH D COM: 4 C.C. = 3





↳ Siccome usiamo o π o S li raggruppiamo in D → DATO

$T(D, Q, CLK) =$

	0	1
00	0	0
01	1	1
10	0	1
11	0	1

D = 1 ⇒ set

D = 0 ⇒ reset (anche qui sensibilità al livello)

$= \overline{CLK}\,Q + CLK\,D$

se CLK basso → conserva Q

se CLK alto → D scritto

FLIP FLOP

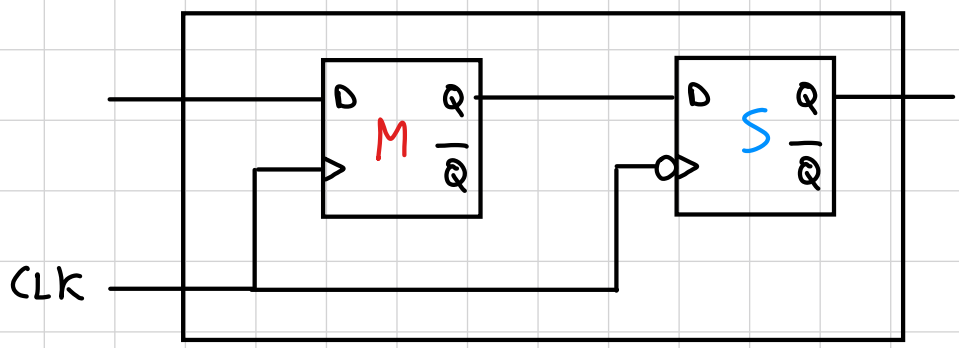
Si può implementare il latch D nell'architettura sincrona, ma ciò un problema

- se CLK=0 tutti i latch mantengono lo stato corrente
- se CLK=1 tutti i latch sono sensibili a cambiare

} ⇒ perdiamo la separazione tra stadi ⇒ CaCr alto.

Questo problema si risolve con i FLIP FLOP, combinando 2 latch D

↳ combinati in master-slave ↳ riceve $\overline{CLK}$



C.C. = 6

COM = 8

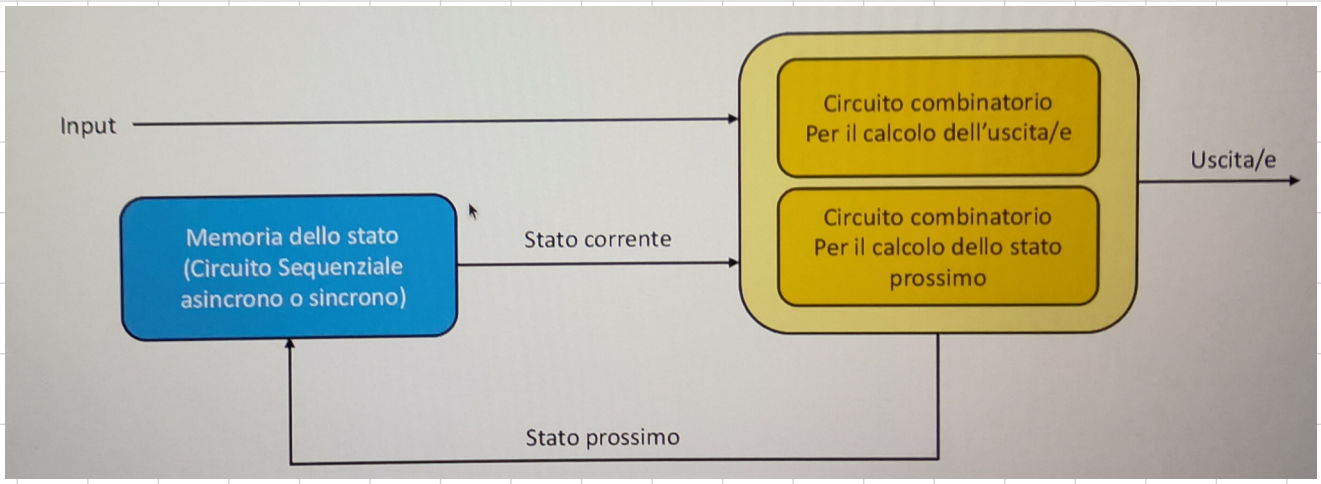
•) CLK=1 ⇒ FLIP
- Master aperto → input viene scritto
- Slave isolato → mette in out lo s corrente

•) CLK=0 → FLOP
- Master isolato → mette in out lo s corrente
- Slave aperto → in un $\Delta t \cong 0$ s_M viene copiato in Slave che lo pone in out

Il FLIP FLOP è sensibile ai fronti → l'uscita commuta in modo istantaneo sul fronte di discesa e resta stabile per tutto il ciclo

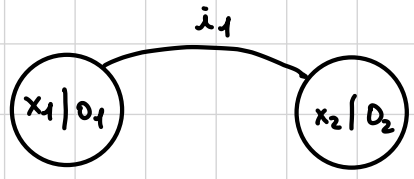
MACCHINE A STATI FINITI

Per costruire C.S con più di 2 stati possibili servono 2 cose:
- più di un bit di memoria (se n stati → $\log_2 n$ bit) → usando n FLIPFLOP costruisco un registro da n bit
- un formalismo più complesso



$X = \{x_1, x_2, \dots, x_n\}$ insieme finito di stati
$x^i \in X$ stato iniziale
$I = \{i_1, i_2, \dots, i_n\}$ insieme di input che può ricevere
$O = \{o_1, o_2, \dots, o_n\}$ insieme di output che può generare
$T : X \times I \to X$ funzione di transizione o stato prossimo
$g()$ funzione di uscita
↳ $g: X \to O$ Moore
$g: X \times I \to O$ Mealy

Per rappresentare graficamente una FSM si usa un grafo orientato:



$g(x_1) = o_1$, $g(x_2) = o_2$
$T(x_1, i_1) = x_2$

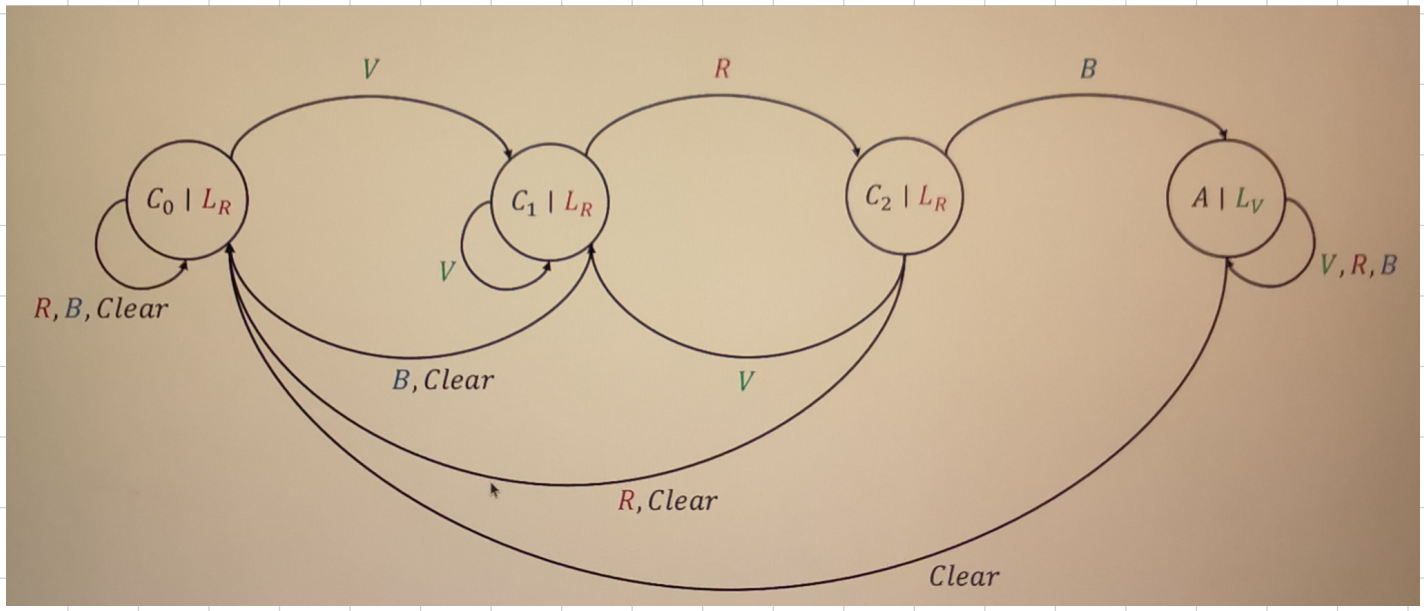
ESEMPIO:



$X = \{C_0, C_1, C_2, A\}$
C_1 chiusa e so i primi i simboli
↳ aperta

$C_0 \in X$
$I = \{V, R, B, Clear\}$
$O = \{L_R, L_V\}$

$g(C_i) = L_R \; \forall i \in \{0, 1, 2\}$
$g(A) = L_V$



	V	R	B	Clear	Out
C_0	C_1	C_0	C_0	C_0	L_R
C_1	C_1	C_2	C_0	C_0	L_R
C_2	C_1	C_0	A	C_0	L_R
A	A	A	A	C_0	L_R

↳ indicano lo stato prossimo

⇒ per costruire un circuito di una FSM serve:
- sapere da quanti Flip Flop è costituita la memoria
- sapere com'è fatto il circuito per il calcolo di out
- sapere com'è fatto il circuito per il calcolo di stato prox.

① codificare ogni stato in binario e memorizzarlo
n bit → n FF
② codificare ogni input e output in binario (non serve memoria, non rappresentano lo stato)
③ sintetizzare espressione bool di ogni bit di out e stato prox come $f(x)$ logiche indipendenti
↳ costruire TT usando bit invece di nomi astratti, costruire SOP, per sicurezza faccio anche mappe di Karnaugh (oppure le uso subito $x\bar{x}$ + veloci)
④ sintetizzare i circuiti combinatori
⑤ composizione del circuito finale

CPU A SINGOLO CICLO

↳ 1 istruzione per ciclo di clock
↳ stringa binaria che codifica le info necessarie per un'operazione
La CPU decodifica la stringa ed esegue l'elaborazione

L'esecuzione di un'istruzione si svolge in 5 fasi:
- FETCH: istruzione prelevata da memoria e caricata in IR
- DECODE: la U.C. riconosce istruzione e predispone ALU per esecuzione
- EXECUTE: ALU svolge operazioni
- MEMORY: se previsto si fa accesso a memoria
- WRITE BACK: Registri aggiornati

Il circuito della CPU è composto da due macro-elementi fondamentali:
- DATA PATH: percorso delle istruzioni nella CPU
- LOGICA DI CONTROLLO: manovra i percorsi delle istruzioni e controlla i sottocomponenti della CPU a seconda delle istruzioni

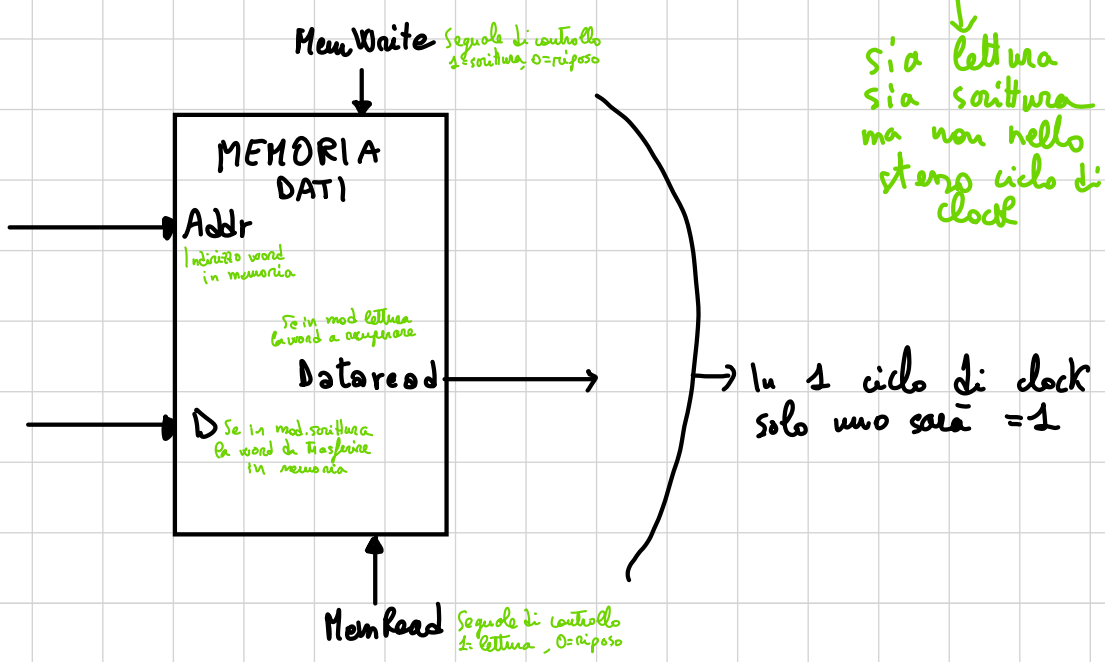
Aritmetico-logiche
accesso memoria dati
controllo di flusso
} quello che la CPU può fare (CODIFICATE su 32 BIT)
↳ organizzati in campi

Formato R (Register) = ALU istruzioni su registri
Formato I (Immediate) = ALU istruzioni su costanti e salto condizionato
Formato J (Jump) = salto non condizionato

MIPS è BIG ENDIAN

MEMORIA

Modulo a cui possiamo accedere tramite un indirizzo
2 sezioni.
- MEMORIA ISTRUZIONI: ogni word è istruzione per CPU → solo in lettura ✗ leggere PC
  ↓
  L'istruzione successiva sta nel Program Counter (PC)
- MEMORIA DATI: ogni word è un dato
  ↓
  sia lettura sia scrittura ma non nello stesso ciclo di clock



→ In 1 ciclo di clock solo uno sarà =1

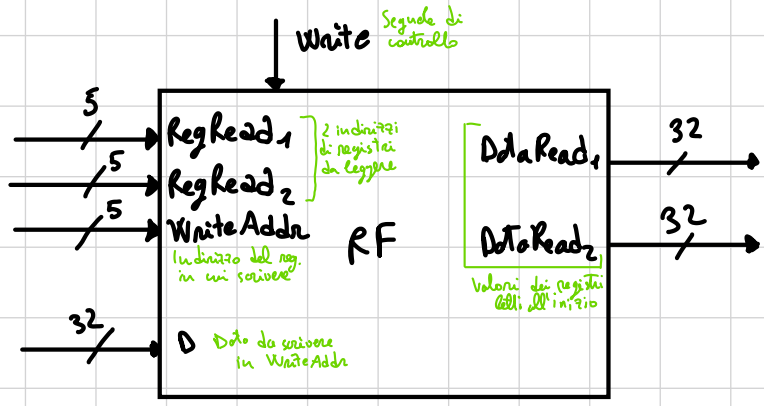
RAM: At di accesso a parola di memoria è fisso, indipendentemente dall'indirizzo
↓
Random access memory

REGISTER FILE

↳ memoria interna ad accesso rapido → formata da REGISTRI
mantiene in memoria i dati su cui opera più frequentemente

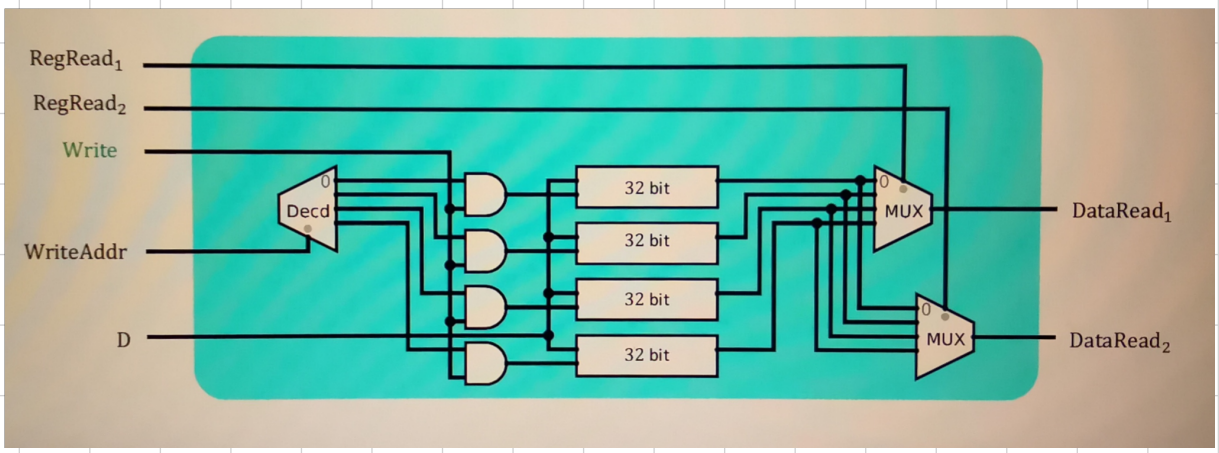
In MIPS un registro = parola in memoria (32 bit)
↳ il RF contiene 32 registri da 32 bit → per indirizzare servono 5 bit

Il RF può essere usato sia in lettura sia in scrittura, e può leggere 2 registri contemporaneamente



↱ bastano 2 bit x indirizzare

ESEMPIO: RF, 4 registri, 32 bit



SPLITTER

Per separare un segnale su più bit in sottogruppi
Si usa per estrarre i campi dai 32 bit di un'istruzione

FORMATO R:

codice identificativo

$$OPCODE + r_1 + r_2 + r_d + shamt + funct$$
$$6 \quad 5 \quad 5 \quad 5 \quad 5 \quad 6$$

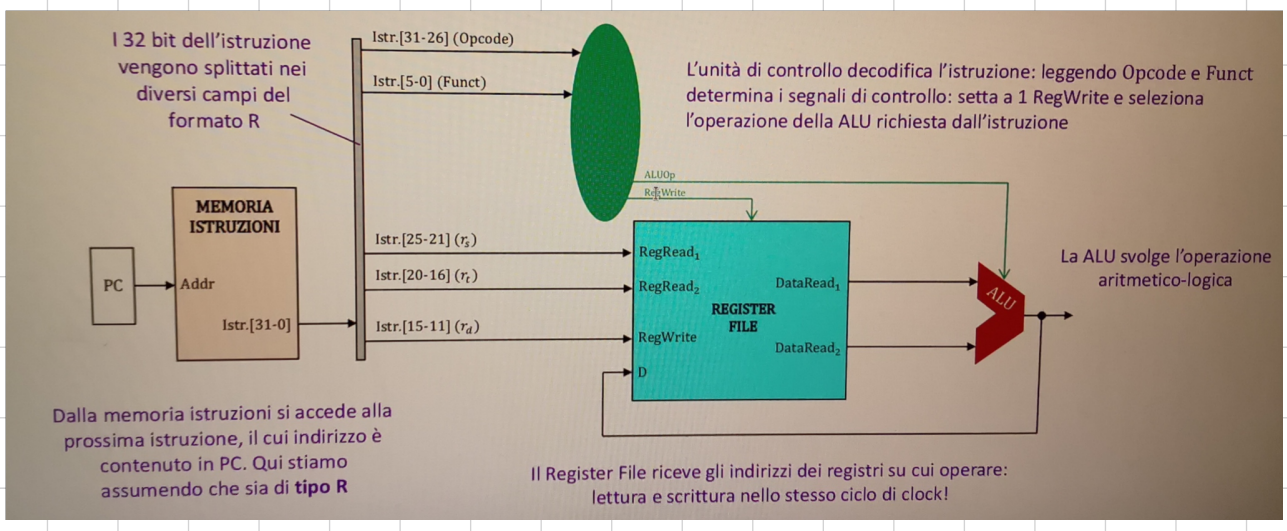
x shift

funzione da svolgere

add 100000
sub 100010
and 100100
or 100101
slt 101010

$OPCODE_R = 000000$

DATA PATH$_R$:



FORMATO I: OPCODE + r_1 + r_2 + OFFSET

6 5 5 16

$OPCODE_{LW} = 100011$

LW: - r_2 è r_d

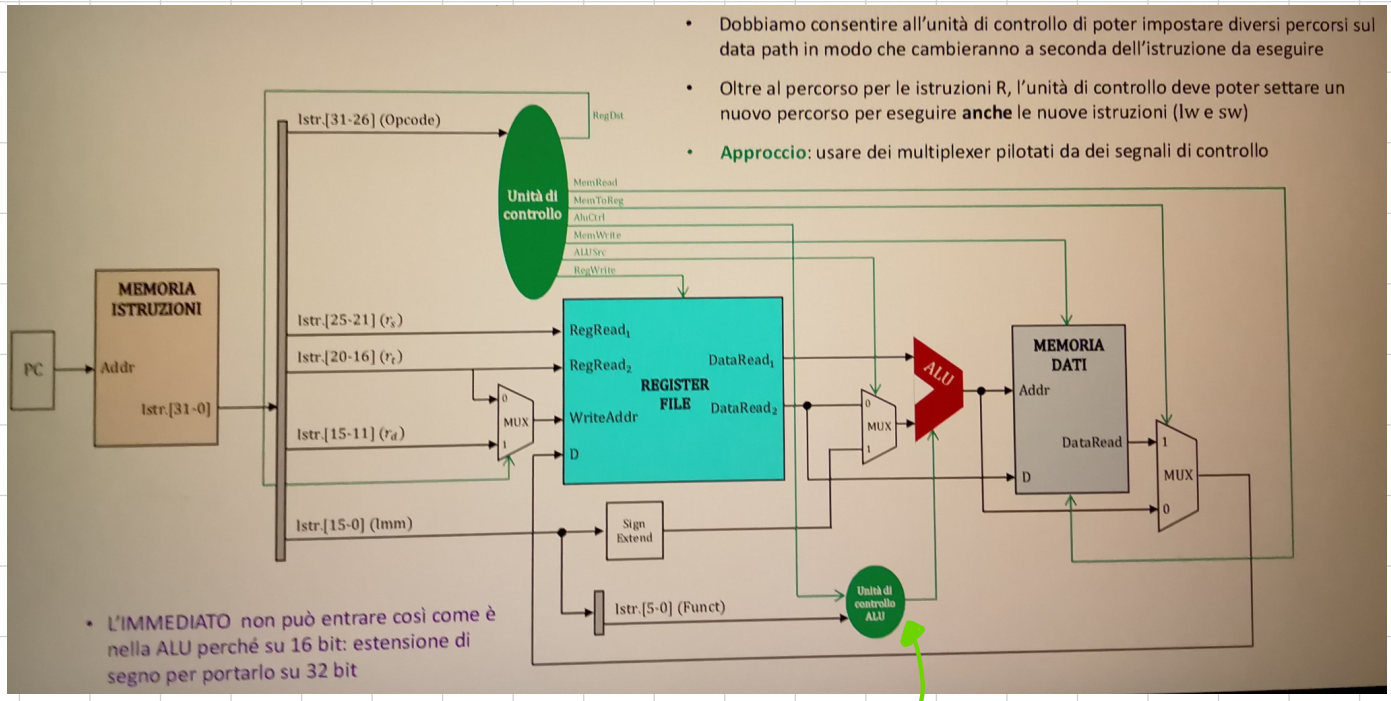
- indirizzo a cui accedere = base address (+) OFFSET

in LW è r_1 → ALU

$OPCODE_{SW} = 101011$

SW: - scrive in RF non memoria

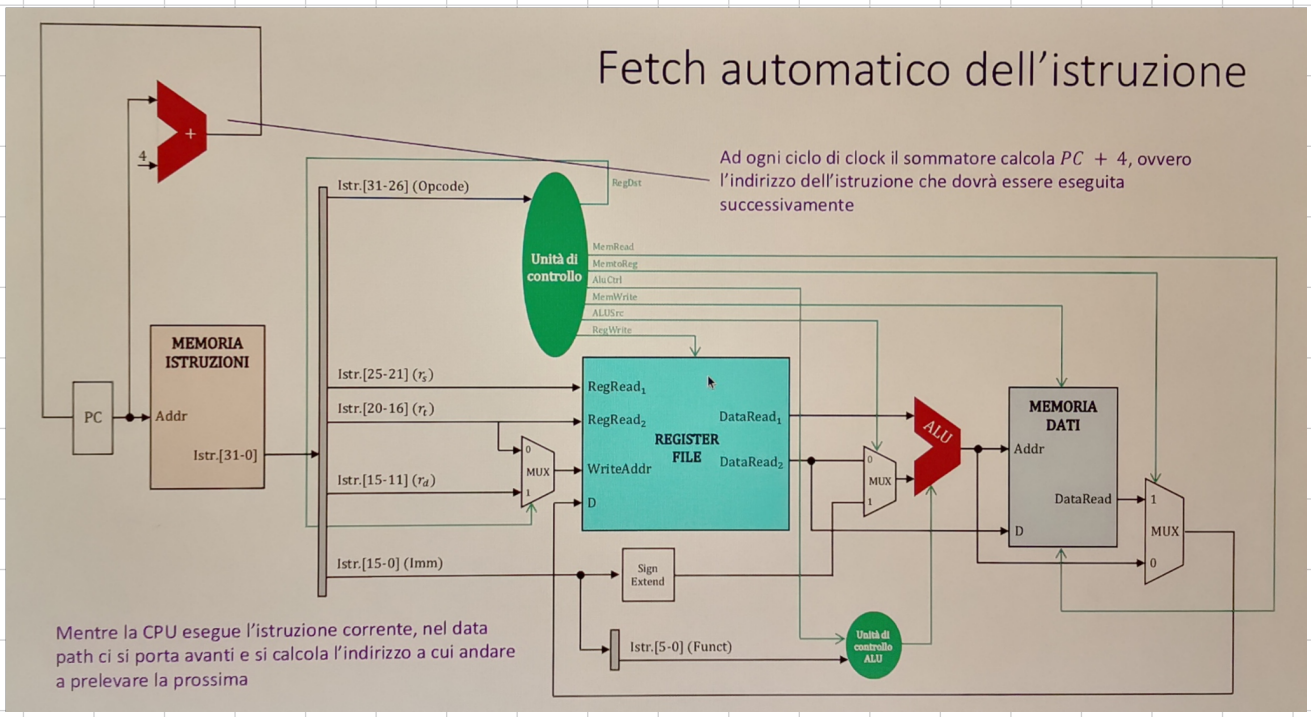
DATA PATH$_I$:



abbiamo separato CU_{ALU} da CU

DATA PATH$_J$:

Fetch auto.



Controllare il flusso significa decidere quale sarà la prossima istruzione e si effettua avendo un indirizzo diverso da PC+4 nel .PC

Si fa in 2 modi:

- salto incondizionato (si sovrascrive sempre PC)
- salto condizionato (se condizione è T => si sovrascrive PC)

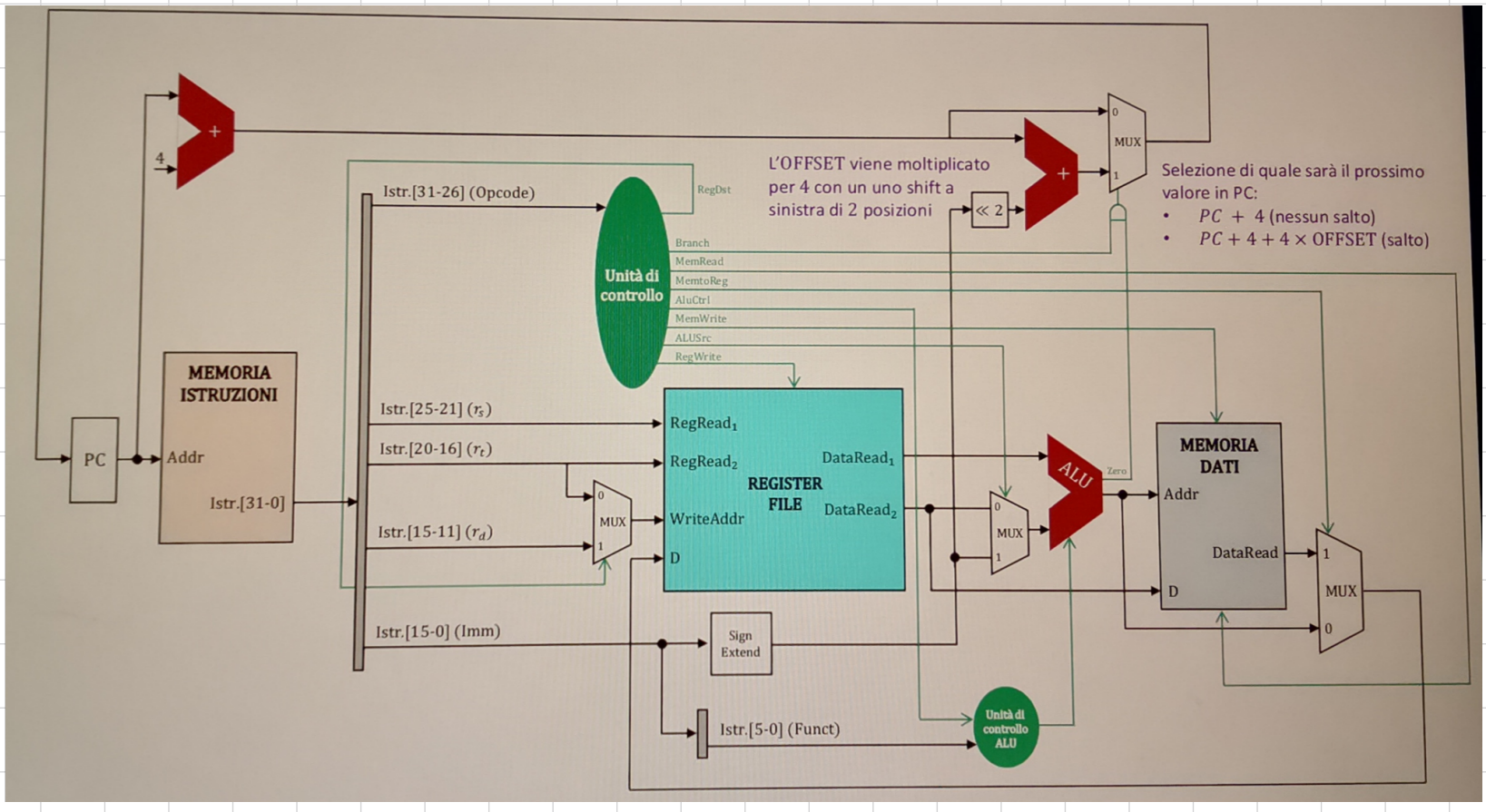
S.C.: beq · beq + r_1 + r_2 + OFFSET
6 5 5 16

OPCODE = 000100

↘

Si guarda l'uguaglianza ⇒ con ALU fai sottrazione e controlli il bit di 0
Se cond. F ⇒ niente
Se cond. T → jump
↳ Modalità di indirizzamento PC-Relative
↳ I 16 bit dell'immediato sono come OFFSET in n.° di istruzioni
(PC + 4) + 4 × OFFSET
↳ $(PC+4)+4\times 2^{-15}$; $(PC+4)+4\times(2^{15}-1)$

DATA PATH'



FORMATO J: OPCODE + PSEUDO-INDIRIZZO
6 26

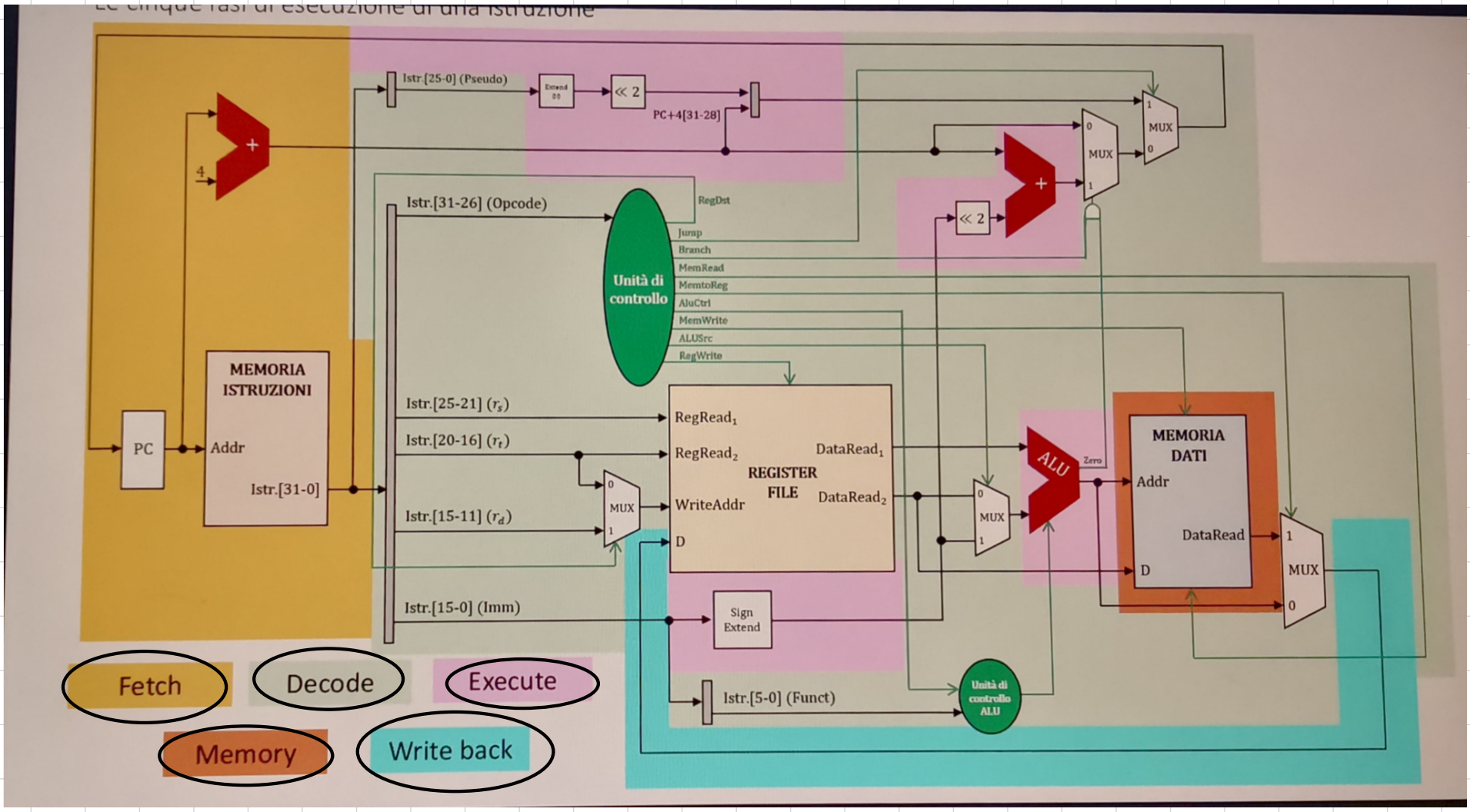
↳ Scrivi in PC un nuovo indirizzo ottenuto applicando modalità di indirizzamento pseudo-diretta

- aggiungi due zeri a dx di 26 bit
- aggiungi a sx i 4 bit MSB di PC

} 32 bit da sommare scrivere in PC

$OPCODE_J$ = 000010

DATA PATH finale:



- **RegWrite**: attiva scrittura in RF (D scritto in WriteAddr)
- **MemRead**: attiva lettura in memoria (in DataRead si mette il valore di word che è in Addr)
- **MemWrite**: attiva scrittura in memoria (D scritto in word di memoria che è in Addr)
- **ALUSrc**: se=0 → il 2° operando della ALU viene da RF → out DataRead$_2$
  se=1 → il 2° operando della ALU è l'estensione su 32 bit del campo IMMEDIATE
- **MemtoReg**: se=0 → D da scrivere in RF viene da OUT di ALU
  se=1 → D da scrivere in RF viene da OUT di DataRead di memoria
- **RegDst**: se=0 → indirizzo di destinazione è estratto da r_2
  se=1 → indirizzo di destinazione è estratto da r_d
- **Branch AND Zero**: se=0 → PC=PC+4 (se jump ≠ 1)
  se=1 → PC=PC+4+4×OFFSET (32)
- **Jump**: se=0 → PC=PC+4 (se beq ≠ 1)
  se=1 → PC= MSB_{PC} + PSEUDO-INDIRIZZO + 00
  (MSB_{PC} → 4)

UNITÀ DI CONTROLLO ALU

ALUCtrl: comunica a CU_{ALU} quale istruzione CPU sta facendo (ALU's, access memory, branch)

ALUOp: configura ALU per operazione giusta

(RTL: ALU's (R) → funct ; LW || SW → somma ; beq → differenza

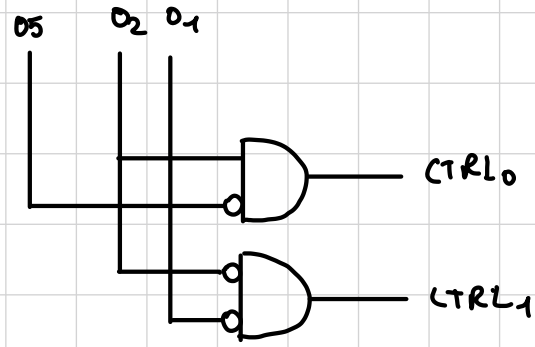
OP:

Si sceglie codifica per ALUCtrl che dipende da OPCODE

OPCode	CTRL
000000	→ 10 → ALU's
100100	→ 00 → memory
101100	→ 00 → memory
000100	→ 01 → Branch

	O_5	O_4	O_3	O_2	O_1	O_0	$Ctrl_1$	$Ctrl_0$
R	0	0	0	0	0	0	1	0
LW	1	0	0	1	0	0	0	0
SW	1	0	1	1	0	0	0	0
beq	0	0	0	1	0	0	0	1
J	0	0	0	0	1	0	×	×

$ALUCtrl_1 = \bar{O}_5 O_2 \quad ALUCtrl_0 = \bar{O}_2 \bar{O}_1$



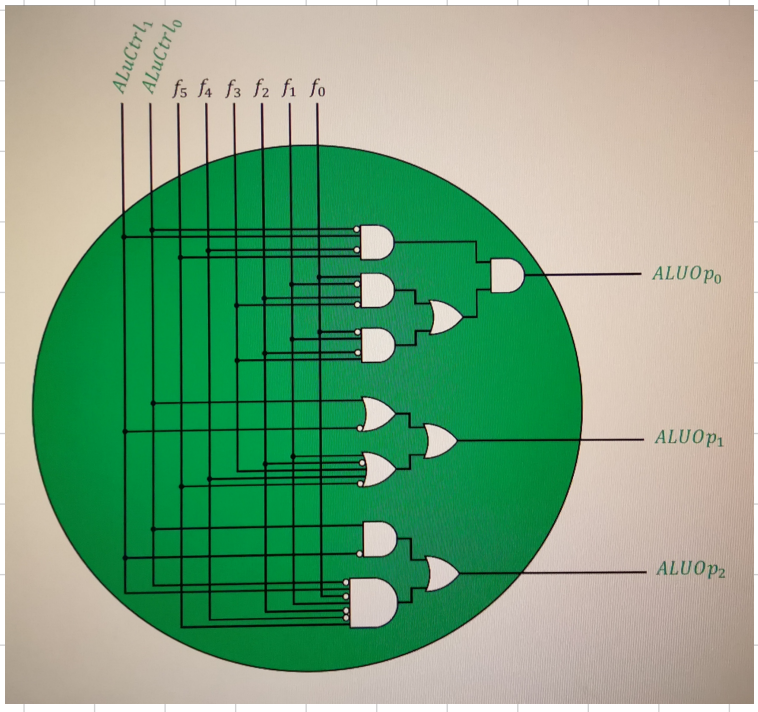
	AluCtrl		funct						ALUOp		
	$AluCtrl_1$	$AluCtrl_0$	f_5	f_4	f_3	f_2	f_1	f_0	$ALUOp_2$	$ALUOp_1$	$ALUOp_0$
lw	0	0	x	x	x	x	x	x	0	1	0
sw	0	0	x	x	x	x	x	x	0	1	0
beq	0	1	x	x	x	x	x	x	1	1	0
add	1	0	1	0	0	0	0	0	0	1	0
sub	1	0	1	0	0	0	1	0	1	1	0
and	1	0	1	0	0	1	0	0	0	0	0
or	1	0	1	0	0	1	0	1	0	0	1
slt	1	0	1	0	1	0	1	0	1	1	1

$ALUCtrl_i = C_i$

$ALUOp_0 = C_1 \bar{C}_0 f_5 \bar{f}_4 (\bar{f}_3 f_2 \bar{f}_1 f_0 + f_3 \bar{f}_2 f_1 \bar{f}_0)$

$ALUOp_1 = (\bar{C}_1 + C_0) + (\bar{f}_5 + f_4 + f_3 + \bar{f}_2 + f_1)$

$ALUOp_2 = \bar{C}_1 C_0 + C_1 \bar{C}_0 (f_5 \bar{f}_4 f_2 \bar{f}_1 \bar{f}_0)$



UNITÀ DI CONTROLLO PRINCIPALE

	O_5	O_4	O_3	O_2	O_1	O_0	RegWr	RegDst	ALUSrc	MemRead	MemWr	M to R	Branch	Jump
R	0	0	0	0	0	0	1	1	0	0	0	0	0	0
LW	1	0	0	1	0	0	1	0	1	1	0	1	0	0
SW	1	0	1	1	0	0	0	x	1	0	1	x	0	0
beq	0	0	0	1	0	0	0	x	0	0	0	x	1	0
j	0	0	0	0	1	0	0	x	x	0	0	x	x	1

$RegWrite = \overline{O_2}\,\overline{O_1} + O_5\,\overline{O_3}$

$RegDst = \overline{O_2}\,\overline{O_1}$

$ALUSrc = O_5$

$MemRead = O_5\,\overline{O_3}$

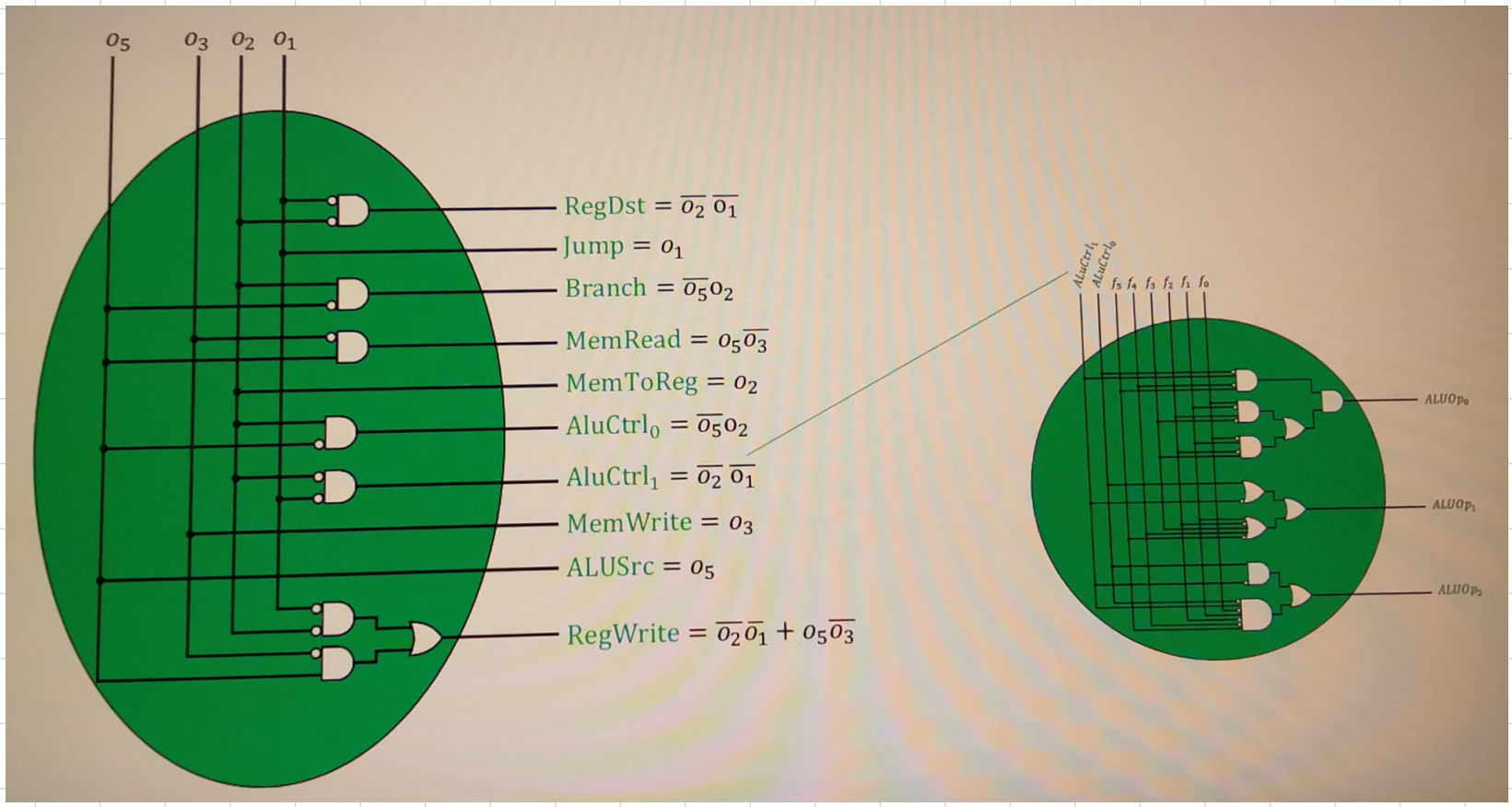
$MemWrite = O_3$

$MemtoReg = O_2$

$Branch = \overline{O_5}\,O_2$

$Jump = O_1$

CU finale:



PARTE SUI CIRCUITI FIRMWARE PRESA DA BORGHESE

ALGORITMO MOLTIPLICAZIONE BINARIA + FIRMWARE

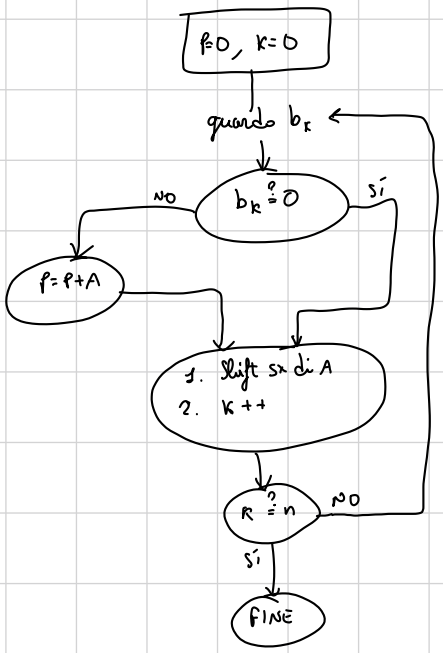
A = moltiplicando (4 bit ma ne uso 8)

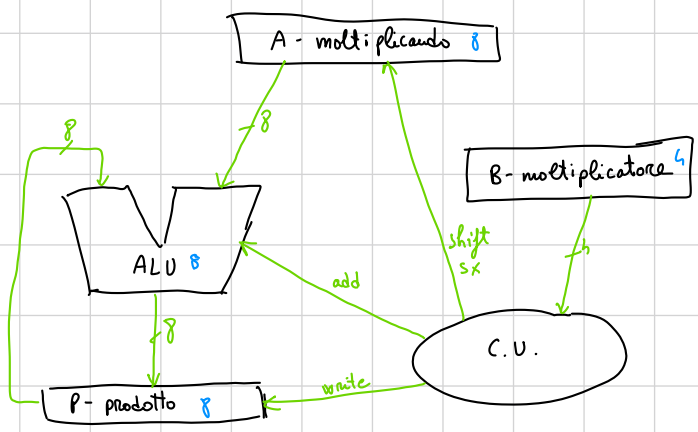
B = moltiplicatore (4 bit)

P = prodotto

K = variabile di iterazione

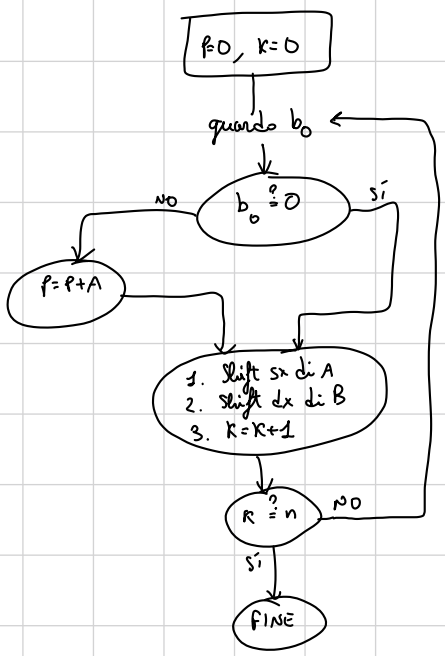
n = numero di bit

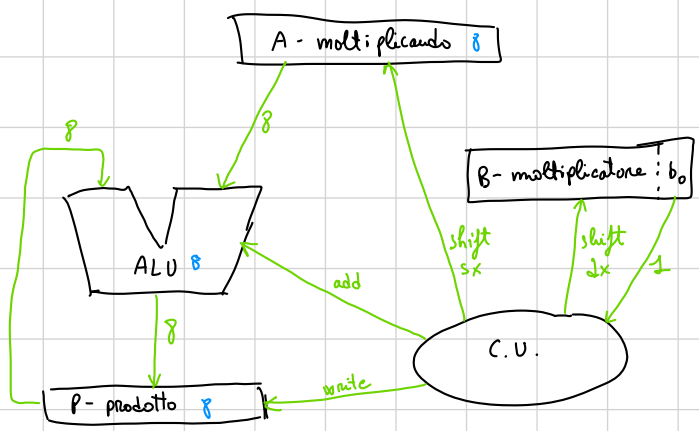




Come funziona?

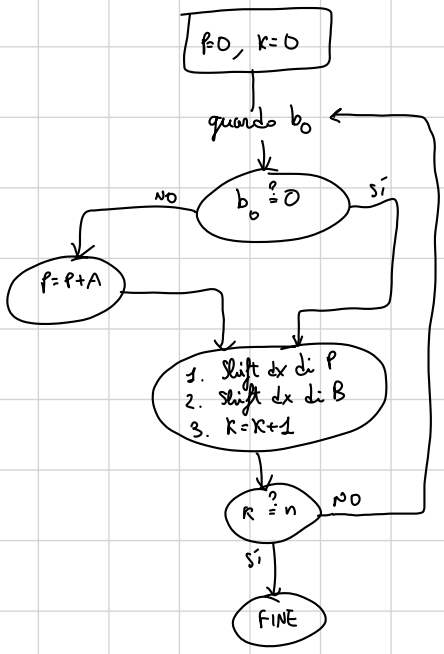
Ad ogni iterazione mando nella C.U. tutti i bit di B, e poi con un MUX la C.U. guarda solo la LSD. Siccome a noi serve solo un bit di B possiamo migliorare il circuito: ad ogni iterazione mandiamo b_0 nella C.U. e poi shiftiamo a dx B, così ad ogni iterazione guardiamo sempre e solo la LSD.

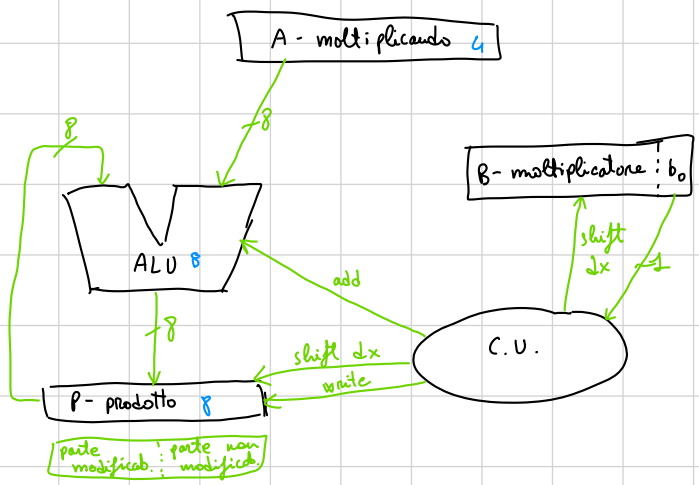




Come si può migliorare?

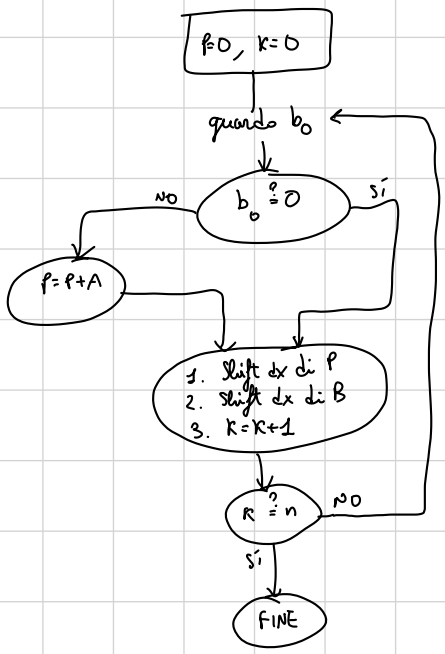
Ad ogni iterazione usiamo solo metà dei bit di A, quindi, invece di shiftare A a sx rispetto a P, possiamo shiftare P a dx rispetto ad A.

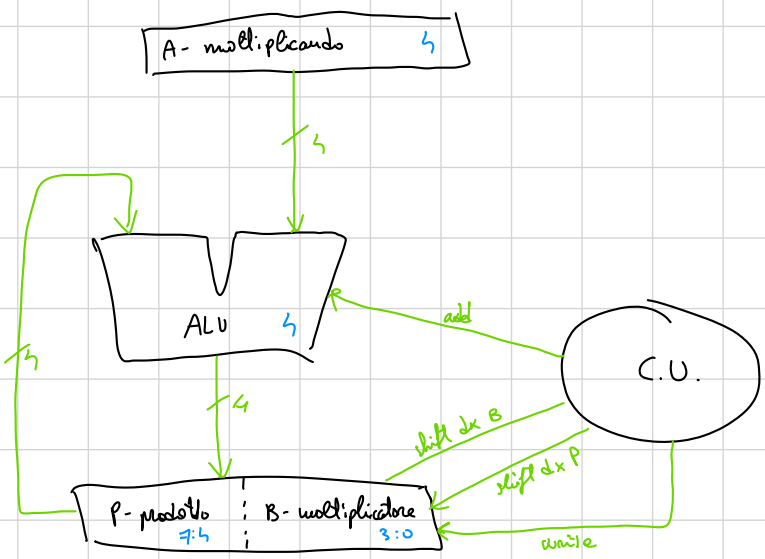




Come si può migliorare?

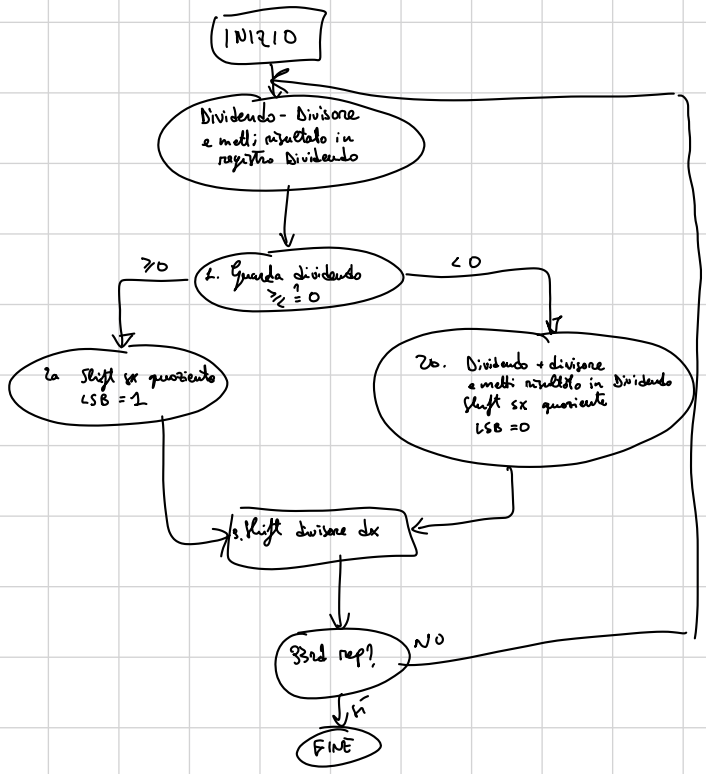
bit prodotto corrente + bit moltiplicatore rimane costante (4+4, 5+3, 6+2, 7+1, 8+0) ⇒ ad ogni iterazione il prodotto si allunga e il moltiplicatore si restringe

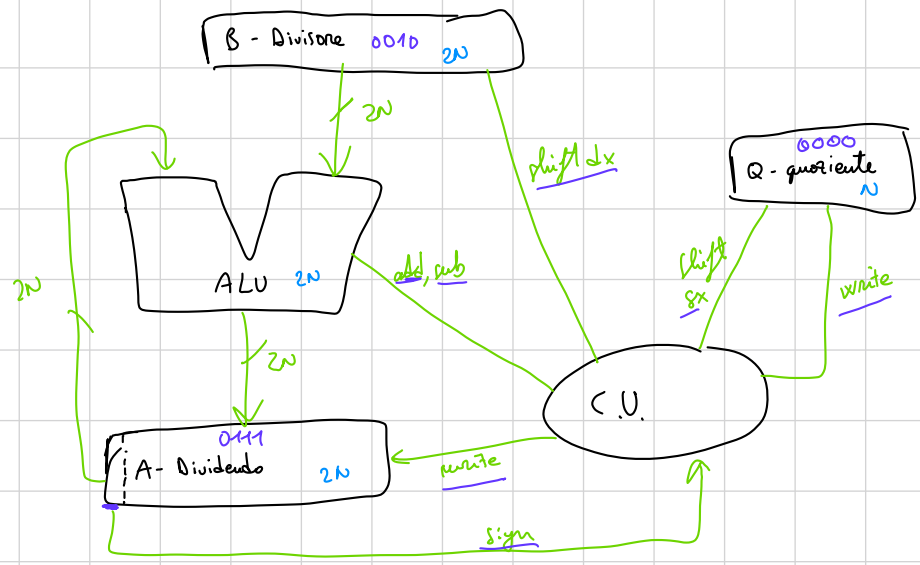




La complessità di questo circuito sommatore è 68 + U.C., maggiore di quella del moltiplicatore HW (65), però con questo circuito posso fare anche la divisione

ALGORITMO DIVISIONE BINARIA + FIRMWARE



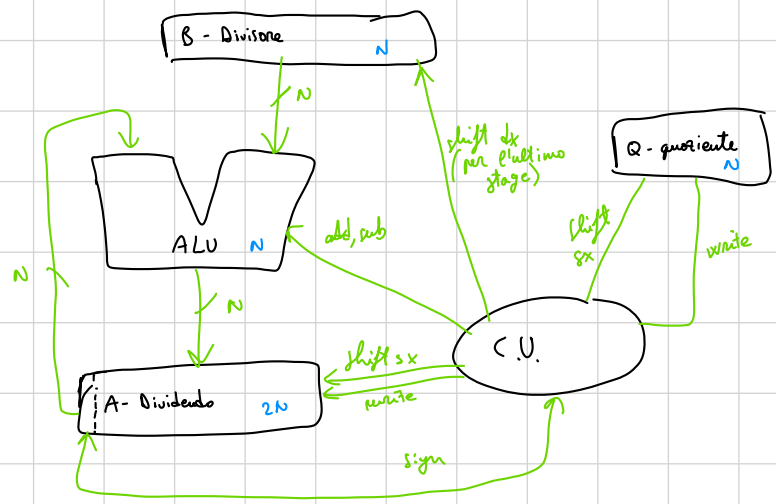


I passaggi sono: • Dividendo - Divisore, se MSB = 1 ⇒ Restore Dividendo, shift sx quoziente $LSB_{quoziente} = 0$, shift dx divisore

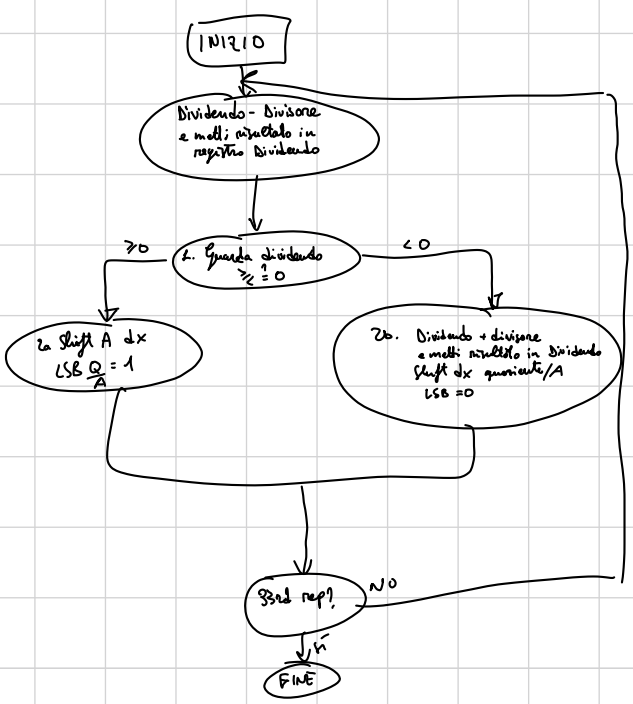
• Dividendo - Divisore, se MSB = 0 ⇒ shift sx quoziente $LSB_{quoziente} = 1$, shift dx divisore

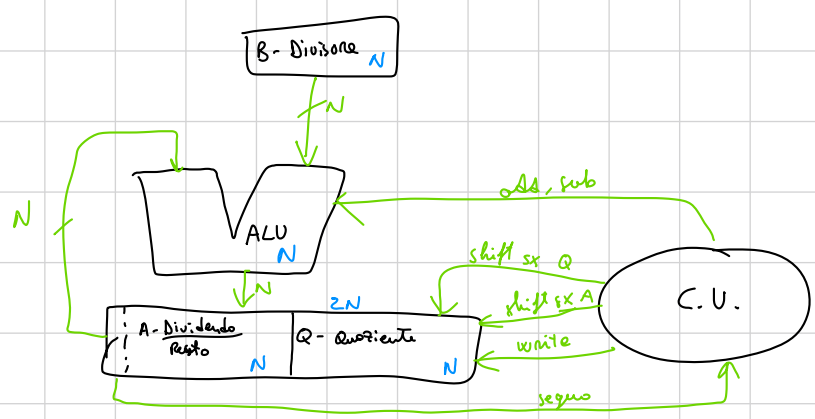
(per i bit da usare si prende il n° + alto e si raddoppia)

Come si può ottimizzare? Invece di shiftare il divisore a dx possiamo shiftare il dividendo/rest. a dx e ottenere lo stesso risultato.

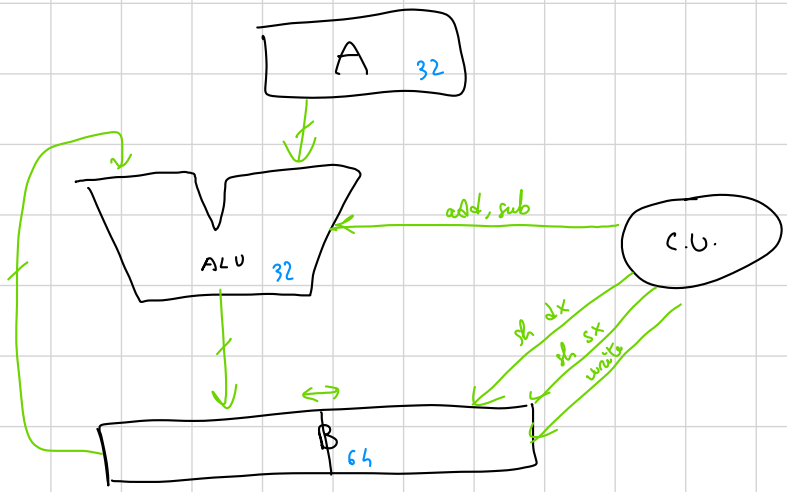


Siccome il dividendo si sposta da dx a sx, e così anche il quoziente, entrambi 1 bit alla volta, si può ottimizzare





UNIAMOCI...



A = moltiplicando / divisore

B = resto = 0 | Dividendo / Prodotto / moltiplicatore

- mult. : XOR bit di segno A e B → se A XOR B == 1 → P è negativo
- div. : XOR bit di segno A e B → se A XOR B == 1 → Q è negativo

↳ resto? $b \cdot q = a - r$ ⟶

$a>0,\ b>0 \rightarrow q>0,\ r\geq 0$
$a>0,\ b<0 \rightarrow q<0,\ r\geq 0$
$a<0,\ b>0 \rightarrow q<0,\ r\leq 0$
$a<0,\ b<0 \rightarrow q>0,\ r\leq 0$

dividendo e resto sempre concordi

SOMMA VIRGOLA MOBILE

