

Un moderno **elaboratore** è un  
**Sistema elettronico digitale programmabile**

- Sistema perché costituito da componenti che interagiscono in modo organico tra loro;
- Elettronico digitale perché sfrutta componenti elettr. digitali;
- Programmabile perché il sistema è flessibile e specificato mediante un programma.

I computer parlano in **binario**, un alfabeto costituito da due soli simboli, 0 e 1, corrispondenti agli stati OFF e ON e conosciuti come **bit**, cioè binary digit

Gli elementi costitutivi di un elaboratore sono:

- Dispositivi di input
  - Dispositivi di output
  - Memoria
  - Unità di elaborazione dati
  - Unità di controllo
- } spesso unite a formare il **processore**.

Quest'ultimo è il cuore dell'elaboratore; è fatto da milioni di transistor, viene studiato per astrazione ed è diviso in due tipi di componenti:

- Combinatorie (senza memoria)
- Sequentiali (con memoria) (richiede Algebra Booleana)

Studieremo una versione semplificata del **MIPS**, sia nella sua implementazione a ciclo singolo, in cui le istruzioni hanno la stessa durata, sia in quella con pipeline, più realistica e con sovrapposizioni temporali delle esecuzioni delle istruzioni.

La **memoria** è il luogo dove vengono tenuti i programmi in esecuzione e i dati di cui essi necessitano ed è caratterizzata da capacità (quantità di dati memorizzabili), consumo (potenza media assorbita) e velocità (intervallo tra richiesta ed accesso <sup>al dato</sup>)

Data una sequenza di  $n$  bit, possiamo rappresentare  $2^n$  interi compresi tra  $0$  e  $2^n - 1$ . Si dimostra per induzione che se con  $K$  bit si può rappresentare  $p$  sequenze distinte, allora con  $K+1$  bit si hanno  $2p$ .

$$1 \text{ bit} \rightarrow 2 \text{ sequenze}$$

$$2 \text{ bit} \rightarrow 4 = 2 \times 2 \text{ sequenze}$$

$$3 \text{ bit} \rightarrow 8 = 2 \times 4 \text{ sequenze}$$

:

$$K \text{ bit} \rightarrow 2^K = 2 \times 2^{K-1}$$

Il minimo numero di bit necessari a rappresentare un numero  $N$  in binario è tale che  $N < 2^n$  cioè  $n > \log_2 N$  (l'intero)

Così come la decimale, anche la notazione binaria è posizionale: il peso della cifra cambia a seconda della posizione.

$$101 = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 5$$

$$1101 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 13$$

Questo appena applicato è il metodo di conversione da binario a decimale, cioè moltiplico ogni cifra per l'appropriata potenza di 2 (inizio da destra con 0) e poi sommo. In realtà questo algoritmo è utilizzabile per qualsiasi base  $b$ .)

Per quanto riguarda le conversioni da decimale a binario:

Dato  $N = (281)_{10}$  cerco la più grande potenza di 2

$$181 = 128 + 53 = 2^7 + 53$$

$$53 = 32 + 21 = 2^5 + 21$$

$$21 = 16 + 5 = 2^4 + 5$$

$$5 = 4 + 1 = 2^2 + 1$$

$$1 = 2^0 + 1$$

arrivato a  $2^0$  mi fermo e rincaro:

$$181 = 2^7 + 2^5 + 2^4 + 2^2 + 1$$

ognuna di queste è moltiplicata per 1, mentre le mancanti per 0

$$(281)_{10} = (10110101)_2$$

Esiste però un altro algoritmo: dato  $(52)_{10}$

$$52/2 = 25 \text{ R}1$$

$$25/2 = 12 \text{ R}1$$

$$12/2 = 6 \text{ R}0$$

$$6/2 = 3 \text{ R}0$$

$$3/2 = 1 \text{ R}1$$

$$1/2 = 0 \text{ R}1$$

leggo dal basso verso l'alto: resti

$$(52)_{10} = (110011)_2$$

Un numero  $N$  espresso in una generica base  $b$  è rappresentato da:

$$N = a_{m-1} \cdot b^{m-1} + a_{m-2} \cdot b^{m-2} + \dots + a_1 \cdot b^1 + a_0 \cdot b^0 = \sum_{i=0}^{m-1} a_i \cdot b^i$$

La conversione da binario a decimale funziona perché

$$\text{dato } N = a_{m-1} \cdot 2^{m-1} + a_{m-2} \cdot 2^{m-2} + \dots + a_1 \cdot 2 + a_0$$

$$\text{definiamo } S_{m-1} = a_{m-1}$$

$$\begin{aligned} S_{m-2} &= a_{m-2} + 2 \cdot S_{m-1} \\ &\vdots \end{aligned}$$

$$S_i = a_i + 2S_{i+1}$$

Ese:

$$n=8 \quad (10210101)_2$$

$$S_7 = a_7 = 1 \quad (\text{la più significativa a sin})$$

$$S_6 = a_6 + 2S_7 = 0 + 2 \cdot 1 = 2$$

$\vdots$

$$S_0 = a_0 + 2S_1 = 1 + 1 \cdot 0 = 1 \text{ da cui}$$

$$S_0 = N$$

mentre seguendo il procedimento al contrario si ottiene  
l'algoritmo di conversione da decimale a binario

Le altre basi più importanti ed usate sono l'**ottale**  $\{0, 1, \dots, 7\}$   
e l'**esadecimale**  $\{0, 1, \dots, 9, A, B, C, D, E, F\}$

La conversione da ottale a binario consiste nella semplice conversione di ogni cifra singolarmente:

$(27)_8 = 010 \ 110 \ 111$  e poi si concatenano  $\Rightarrow 010110111$

Allo stesso modo da binario ad ottale si parla raggruppando le cifre a 3 a 3 a partire da destra e convertendole singolarmente.

Analoghi le ~~tre~~ conversioni da binario ad esadecimale:

$$(100101111)_2 = 10 \ 0101 \ 111 = (25F)_{16}$$

e viceversa:

$$AG7_{16} = 1010 \ 0110 \ 0111 = (101001100111)_2$$

I passaggi da ottale ad esa e da esa ad ottale richiedono la conversione in binario come intermedio

$$(107C)_8 = ?_{10}$$

1	0	7	C
↓	↓	↓	↓
001	000	111	110
raggr. per 4			
2	3	E	

$$(107C)_8 = (23E)_{16}$$

$$(1FOC)_{16}$$

$$\begin{array}{cccccc} 0 & 0 & 0 & 1 & 1 & 1 \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ 1 & 8 & 4 & 1 & 4 & \end{array} \quad (1FOC)_{16} = (17414)_8$$

N.B. - In binario i numeri per terminano con uno 0, mentre i dispern con 1

Un numero con la virgola viene rappresentato come:

$$N = a_{m-1} b^{m-1} + a_{m-2} b^{m-2} + \dots + a_0 b^0 + a_{-1} b^{-1} + a_{-2} b^{-2} + \dots + a_{-s} b^{-s} =$$

parte intera  $\sum_{i=0}^{m-1} a_i b^i$       parte frazionaria

Dato un numero in binario:

$$N = 1 \cdot 2^3 + 2 + 1 + 2^{-1} + 2^{-2} = 11,75$$

Per convertire da decimale a binario invece:

$$F = 0,234_{10}$$

$$2 \cdot 0,234 = 0 + 0,468$$

$$2 \cdot 0,468 = 0 + 0,936$$

$$2 \cdot 0,936 = 1 + 0,872$$

$$2 \cdot 0,872 = 1 + 0,744$$

$$2 \cdot 0,744 = 1 + 0,488$$

... si ferma a bit terminati oppure  
quando si raggiunge lo 0

$$= (0,00111\dots)_2$$

Si può anche seguire un altro procedimento per convertire binario.

$$\bar{F} = (0, a_1 a_2 \dots a_s)_2$$

$$F_{-s+1} = a_{-s}/2$$

$$F_{-s+2} = (a_{-s+1} + F_{-s+1})/2$$

...

$$F_{-1} = (a_{-2} + F_{-2})/2$$

$$F_0 = (a_{-1} + F_{-1})/2 = F$$

$$F = (0,1101)_2$$

$$F_{-3} = a_{-4}/2 = 1/2 = 0,5$$

$$F_{-2} = (a_{-3} + F_{-3})/2 = 0,25$$

$$F_{-1} = (a_{-2} + F_{-2})/2 = 0,625$$

$$F_0 = (a_{-1} + F_{-1})/2 = 0,8125_{10}$$

Per effettuare le addizioni in binario si procede bit per bit

$$- 0+0=0$$

$$- 0+1=1$$

$$- 1+0=1$$

$$- 1+1=0 \text{ con resto } 1$$

Caso particolare:

$$\begin{array}{r} 111111 \\ + (63) \\ \hline 1000000 \end{array}$$

Da cui:

$$(m)_2 = 2^{n-1} + 2^{n-2} + \dots + 2^0$$

$$(m)_2 + 1 = (m+1)_2 = 2^n \Rightarrow m = 2^n - 1$$

$$\sum_{i=0}^{n-1} 2^i = 2^n - 1$$

La precedente somma mostra che la somma di due numeri rappresentati con  $n$  bit può essere un numero che necessita di  $n+1$  bit per essere rappresentato e questa condizione è detta **overflow**

Per quanto riguarda la sottrazione:

$$- 1-0=1$$

$$- 1-1=0$$

$$- 0-0=0$$

$$- 0-1=1 \text{ perché viene prestata una cifra}$$

Per codificare informazioni non numeriche è stato creato il **Codice ASCII** (American Standard Code for Information Interchange)

un codice alfabetico a 7 bit, poi esteso ad 8 bit:

- Se i primi 3 bit sono 011, il simbolo è un numero

- A 100 e 101 corrispondono lettere maiuscole

- A 110 e 111 corrispondono lettere minuscole

- A 010 e 000 corrispondono punteggiatura e caratteri speciali

Un metodo per rappresentare il segno di un numero è quello detto **modulo e Segno** in cui il bit più significativo è usato per il Segno: 0 positivo e 1 negativo, mentre i restanti indicano il modulo, fornendo un intervallo che va da  $[2^{m-1} + 1, 2^{m-1} - 1]$ , molto ristretto rispetto al binario puro. Presenta inoltre il problema della doppia rappresentazione dello zero oltre che quello di complicare i calcoli.

Si è quindi deciso di usare la rappresentazione in complemento a 2 in cui il peso del bit più a sx è sempre negativo:

$$N = -2^{m-1} \cdot b_{m-1} + \sum_{i=0}^{m-2} 2^i b_i$$

$$00010010_{c_2} = 2^4 + 2 = 18_{10}$$

$$10010010_{c_2} = -2^7 + 2^4 + 2 = -110_{10}$$

$$\text{Sia } B = (b_{m-1} b_{m-2} \dots b_0)_{c_2}$$

- Se  $B$  è positivo o zero allora  $b_{m-1} = 0$  e il valore di  $b_{m-2} \dots b_0$  è uguale a  $|B|$

- Se  $B$  è negativo allora  $b_{m-1} = 1$  e il valore di  $b_{m-2} \dots b_0$  è uguale a  $2^{m-1} - |B|$ , cioè il complemento a 2<sup>m-1</sup> di  $|B|$

Il C2 presenta un'unica rappresentazione per lo zero, un'aritmetica semplice in cui è facile trovare l'opposto ed un intervallo da  $[-2^{m-1}, 2^{m-1} - 1]$

Per quanto riguarda l'opposto ci sono due algoritmi:

- negazione, cioè si trasformano gli 1 in 0 e viceversa, ed al risultato si somma 1
- partendo da destra, si lasciano invariati tutti i bit fino al primo 1 incluso, poi si invertono i rimanenti

DIM. primo algoritmo:

$$N = -2^{m-1} b_{m-1} + \sum_{i=0}^{m-2} 2^i b_i \text{ da cui}$$

$$\begin{aligned} -N &= 2^{m-1} b_{m-1} - \sum_{i=0}^{m-2} 2^i b_i \quad \text{aggiungo e sottraggo } 2^{m-1} \\ &= -2^{m-1} + 2^{m-1} b_{m-1} + 2^{m-1} - \sum_{i=0}^{m-2} 2^i b_i \end{aligned}$$

$$\sum_{i=0}^{m-2} 2^i = 2^{m-1} - 2$$

$$= -(1 - b_{m-1}) 2^{m-1} + \sum_{i=0}^{m-2} 2^i (2 - b_i) + 1$$

può essere solo 0 oppure 1

$2 - b_i$  è il negato di  $b_i$

Nessun algoritmo è applicabile al minimo rappresentabile in quanto restituirei sempre se stesso; questo è un caso di overflow in quanto il suo opposto è sempre un numero escluso dall'intervallo.

Per estendere il numero di bit da  $m$  con cui è rappresentato un numero basta replicare la cifra più a dx

$$-2_{10} = 1110_2 = -2^3 + 2^2 + 2 = -8 + 6 = -2_{10}$$

$$\begin{aligned} -2_{10} &= 1111\ 1110_2 = -2^7 + (2^6 + 2^5 + 2^4 + 2^3) + 2^2 + 2 \\ &= -2^7 + (2^7 - 2^0) + 2^2 + 2 \\ &= -2^7 + 2^2 + 2 = -2_{10} \end{aligned}$$

Questo perché

$$\sum_{i=h}^k 2^i = \sum_{i=0}^k 2^i - \sum_{i=0}^{h-1} 2^i = (2^{k+1} - 2) - (2^h - 2) = 2^{k+1} - 2^h$$

Ricordando di troncare sempre l'ultimo bit di segno, un modo veloce di controllare accorgersi di overflow è controllare la consistenza dei segni; alternativamente  $e_{m-1} = e_m$  indica risultato corretto

La notazione scientifica prevede una sola cifra a sinistra della virgola e viene pertanto detta **normalizzata**. Per i numeri molto grandi e quelli molto ~~negative~~ piccoli, positivi e negativi, si usa la **rappresentazione in virgola mobile o Floating Point**

Essi sono espressi da una tripla  $\langle s, M, E \rangle$  dove

$s \rightarrow$  indica il segno

$-M$  è la mantissa, che indica la parte frazionaria

$-E$  è l'esponente, rappresentato come intero con segno  
da cui

$$N = (-1)^s \cdot (1+M) \cdot 2^E \quad \leftarrow \text{base non rapp. - è binario}$$

<sup>1</sup>non viene rapp. - unico possibile

Per stabilire il numero di bit da assegnare a qualunque degli elementi è nato lo **Standard IEEE 754** nel 1985.

Il formato a precisione singola usa 32 bit: 1 al segno, 8 all'esponente e 23 alla mantissa:

- da  $(\pm 1)^s$  ricaviamo che  $s=0$  indica i positivi

- l'esponente è in binario puro con 00...00 riservato allo 0 e  $11..11 = 255$  per NaN; i restanti  $[1, +254]$  sono messi in corrispondenza con l'intervallo  $[-126, 127]$  per polarizzazione:

$$E = e \text{ (quello rapp. in bin)} - 127$$

$$-0,25_{10} = 0,01_2 \Rightarrow 1,0_2 \cdot 2^{-2} \quad -2 = e - 127 \Rightarrow e = 125$$

$$s=1 \quad e=0111101_2 \quad M=00\dots 00_2$$

$$\begin{aligned} \text{Data } s=1 & \quad e=00001011_2 & M=010\dots 00_2 \\ & \downarrow 11_{10} & = 0,01_2 = 0,25_{10} \end{aligned}$$

$$N = (\pm 1)^s \cdot (1+M) \cdot 2^{E-s-127} = -1,25_{10} \cdot 2^{-126}$$

Esiste anche un formato a precisione doppia che invece riserva 1 bit per il segno, 11 all'esponente e 52 alla mantissa. Per effettuare la somma di due numeri si effettua, se necessario, un allineamento delle mantisse; cioè il numero con esponente più alto basso raggiunge lo stesso ordine di grandezza del più grande spostando la virgola a sinistra; dopo la somma, si normalizza il risultato e se necessario si arrotonda al numero di bit richiesto.

Un processore è formato dall'ALU e dalla CC e presenta due tipi di componenti: combinatorie, che ~~non usano memoria~~<sup>senza memoria</sup>, e sequenziali, con memoria. Le prime sono anche dette blocchi logici o reti combinatorie.

Un blocco logico riceve diversi input e restituisce diversi output tramite segnali elettrici alti (1) e bassi (0) a realizzarsi in funzioni logiche di  $m$  variabili in  $\{0,1\}$  a valori in  $\{0,1\}$

$$f: \{0,1\}^m \rightarrow \{0,1\}$$

Sono dette anche funzioni di commutazione in quanto l'output dipende dalla combinazione degli input.

Le TAVOLE DI VERITÀ sono Tabelle che elencano i valori di ogni variabile e delle loro combinazioni all'interno di una funzione.

Somma  $(a, b, c_m) = s$

a	b	c <sub>m</sub>	s
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Ripporto  $(a, b, c_m) = cout$

a	b	c <sub>m</sub>	cout
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

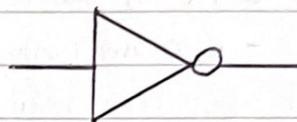
Le principali funzioni booleane sono NOT, AND e OR e vengono dette funzioni logiche in quanto sono realizzate da porte logiche.

NOT ( $x$ ) dà come risultato Vero se la variabile è posta a Falso, mentre è Falso altrettanto; corrisponde alla negazione di  $x$ .

$x$	$\bar{x}$
0	1
1	0

$$\text{da cui } \bar{0} = 1 \\ \bar{1} = 0$$

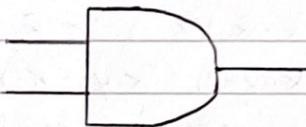
$$\text{NOT}(V) = F \\ \text{NOT}(F) = V$$



AND ( $x, y$ ) dà come risultato Vero se entrambi le variabili sono poste a Vero, Falso altrettanto; corrisponde al prodotto  $x \cdot y$ .

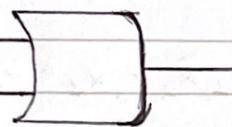
$x$	$y$	$x \cdot y$
0	0	0
0	1	0
1	0	0
1	1	1

$$0 \cdot 0 = 0 \\ 0 \cdot 1 = 0 \\ 1 \cdot 0 = 0 \\ 1 \cdot 1 = 1$$



OR ( $x, y$ ) dà come risultato Vero se almeno una delle due variabili è posta a Vero, Falsa altrettanto; corrisponde alla somma  $x+y$ .

$x$	$y$	$x+y$
0	0	0
0	1	1
1	0	1
1	1	1



Il processo di calcolo della funzione associata alla rete si chiama analisi della rete. Il processo inverso è invece detto sintesi della rete e porta alla creazione di molte reti formate da componenti diversi e con prestazioni diverse. Il processo di minimizzazione consente di ottenere la rete con prestazioni migliori.

Nell'algebra di Boole

le costanti 0 e 1 hanno le seguenti proprietà:

$$-\text{NOT}(0) = 1$$

$$-x \text{ AND } 1 = x$$

$$-x \text{ OR } 0 = x$$

$$x \text{ AND } 0 = 0$$

$$x \text{ OR } 1 = 1$$

vale inoltre la seguente assunzione:

- Ciascuna espressione vale 0 o 1 per ogni assegnazione di valori alle sue variabili

Sono definite le seguenti proprietà:

$$-\bar{\bar{x}} = x$$

involutiva

$$-x \cdot y = y \cdot x, x + y = y + x$$

commutativa

$$-x \cdot x = x, x + x = x$$

idempotenza

$$-x \cdot \bar{x} = 0, x + \bar{x} = 1$$

complemento

$$-x \cdot (y + z) = x \cdot y + x \cdot z$$

distributiva

$$-x \cdot (\bar{x} + y) = x, x + (x \cdot y) = x$$

assorbimento

$$-x \cdot (\bar{x} + y) = x \cdot y, x + (\bar{x} \cdot y) = x + y$$

associativa

$$-\bar{x} \cdot \bar{y} = \bar{x} + \bar{y}, \bar{x} + \bar{y} = \cancel{x \cdot y} \quad \text{de Morgan}$$

Esercizi.

$$E = x_1 \bar{x}_2 x_3 x_4 + x_1 \bar{x}_2 x_3 \bar{x}_4 = x_1 \bar{x}_2 x_3 (x_4 + \bar{x}_4) = x_1 \bar{x}_2 x_3$$

$$E = x_2 x_4 + \bar{x}_2 x_3 \quad \text{da cui}$$

$$\bar{E} = \overline{x_2 x_4 + \bar{x}_2 x_3} = \overline{x_2 x_4} \cdot \overline{\bar{x}_2 x_3} = (\bar{x}_2 + \bar{x}_4) \odot (x_2 + \bar{x}_3)$$

Una funzione logica o di commutazione è una funzione di variabili binarie a valore binario

Altri operatori logici importanti sono XOR, NAND e NOR

XOR. OR esclusivo dà come risultato Vero se una variabile è posta a Vero, ma non entrambe; Falso, altrimenti  $x \cdot \bar{y} + \bar{x} \cdot y = x \oplus y$

$$x \cdot y \quad x \oplus y$$

$$0 \quad 0 \quad 0 \quad \text{XOR}(F, F) = F$$

$$0 \quad 1 \quad 1 \quad \text{XOR}(F, V) = V$$

$$1 \quad 0 \quad 1 \quad \text{XOR}(V, F) = V$$

$$1 \quad 1 \quad 0 \quad \text{XOR}(V, V) = F$$



NAND dà come risultato Falso se entrambi le variabili sono poste a Vero; Vero, altrimenti

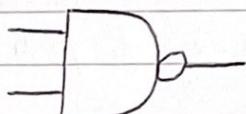
$$x \quad y \quad \overline{x \cdot y}$$

$$0 \quad 0 \quad 1$$

$$0 \quad 1 \quad 0$$

$$1 \quad 0 \quad 0$$

$$1 \quad 1 \quad 0$$



NOR dà come risultato Vero se entrambi le variabili sono poste a Falso; Falso, altrimenti

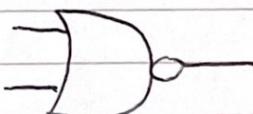
$$x \quad y \quad \overline{x+y}$$

$$0 \quad 0 \quad 1$$

$$0 \quad 1 \quad 0$$

$$1 \quad 0 \quad 0$$

$$1 \quad 1 \quad 0$$



Un' espressione booleana è una combinazione di variabili booleane, dette letterali, ed operatori e denota una funzione logica. Si presenta in forma normale SOP quando è l'OR di AND di letterali ed in particolare viene detta canonica quando tutti i suoi termini sono mintermini, presentano cioè ogni variabile o vera o negata.

Data una funzione logica  $f$ , ne ricaveremo la tavola di verità e con essa costruiamo un'espressione da riportare in forma canonica SOP; da essa si ottiene una rete a due livelli: nel primo troviamo tutte le porte AND e NOT, nel secondo una singola porta OR. Questo tipo di rete esegue molte operazioni in parallelo, dunque è molto più veloce di una sviluppata in orizzontale.

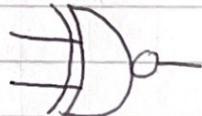
Se la funzione logica è un mintermone, la sua tavola presenta un solo 1; se ne fosse più di uno, basta sommarli.

Le prestazioni di una rete dipendono dalla velocità, che dipende dal massimo numero di porte che un segnale attraversa, e dal costo, che dipende dal numero di porte e linee.

XOR negata o coincidenza dà come risultato Vero se entrambe le variabili sono poste a vero oppure a falso, Falso, altrimenti

$$x \oplus y = \bar{x} \cdot \bar{y} + x \cdot \bar{y}$$

$x$	$y$	$\overline{x \oplus y}$
0	0	1
0	1	0
1	0	0
1	1	1



Un'espressione booleana è in forma normale POS quando è l'AND di OR di letterali ed in particolare viene detta canonica quando ogni suo termine è un maxtermone, cioè una somma di letterali in cui compare ogni variabile.

Se  $f$  è un maxtermone la sua tavola presenta un solo 0; se ne fossero di più basterebbe moltiplicarli ottenendo una rete OR-to-AND

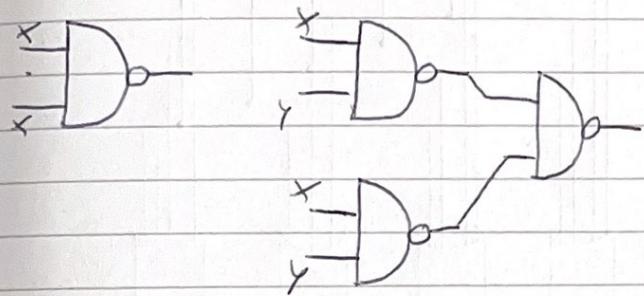
(a 2 liv.)

L'operatore NAND è un operatore logicamente completo in quanto è possibile utilizzarlo per costruire gli operatori NOT, AND e OR (e dunque anche ogni funzione logica)

$$\text{NOT}(x) = \overline{x} = \overline{x \cdot x} \text{ (idempotenza)} = \text{NAND}(x, x)$$

$$\text{AND}(x, y) = x \cdot y = \overline{\overline{x} \cdot \overline{y}} \text{ (involturazione)} = \overline{\overline{x} \cdot \overline{y}} = \text{NAND}(\text{NAND}(x, y), \text{NAND}(x, y))$$

$$\text{OR}(x, y) = x + y = \overline{\overline{x} + \overline{y}} = \overline{\overline{x} \cdot \overline{y} \cdot y} = \overline{\overline{x} \cdot \overline{y}} = \text{NAND}(\text{NAND}(x, x), \text{NAND}(y, y))$$



Lo stesso discorso vale anche per NOR

$$\text{NOT}(x) = \text{NOR}(x, x)$$

$$\text{AND}(x, y) = \text{NOR}(\text{NOR}(x, x), \text{NOR}(y, y))$$

$$\text{OR}(x, y) = \text{NOR}(\text{NOR}(x, y), \text{NOR}(x, y))$$

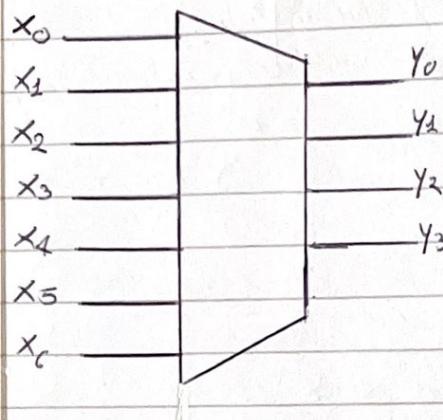
Per entrambi gli operatori vale la proprietà commutativa, ma non vale quella associativa

I moduli combinatori di base sono realizzati come circuiti integrati: realizzati su chip di silicio detti piastre, sono composti da gate e fili e vengono inseriti in un package, il quale è collegato all'esterno tramite pin, detti anche piedini.

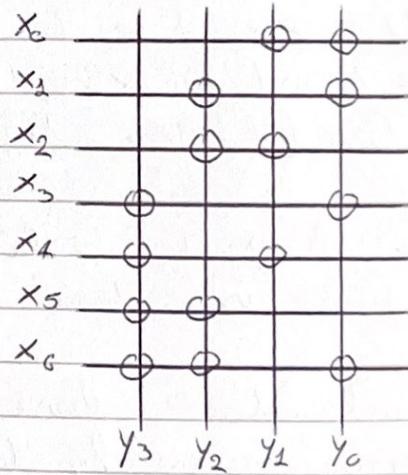
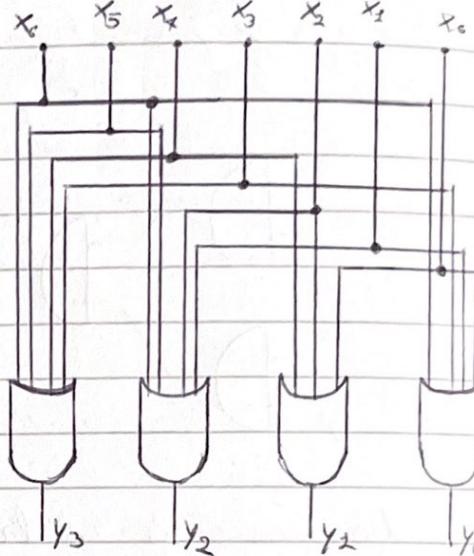
Quelli usati dai calcolatori sono codificatori, decodificatori, multiplexer, demultiplexer, PLA (Programmable Logic Array) e ROM (Read Only Memory)

Codificatore ha  $m$  input ed  $n$  output legati dalla relazione  $2^m \geq n$ .  
 Istante per istante, una sola linea  $i$  è attiva e ad essa valgono  
 associate la sequenza di  $n$  bit corrispondente alla sua rappresentazione  
 binaria.

Supponiamo di avere  $m=7$  input e  $n=4$  linee di uscita



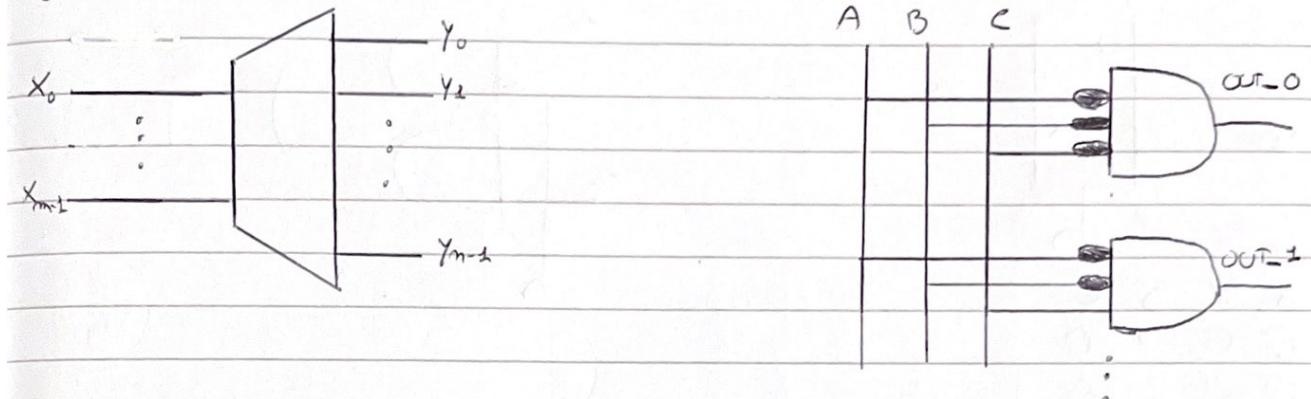
		$y_3$	$y_2$	$y_1$	$y_0$
$x_6$	3	0	0	1	1
$x_5$	5	0	1	0	1
$x_4$	6	0	1	1	0
$x_3$	9	1	0	0	1
$x_2$	10	1	0	1	0
$x_1$	12	1	1	0	0
$x_0$	13	1	1	0	1



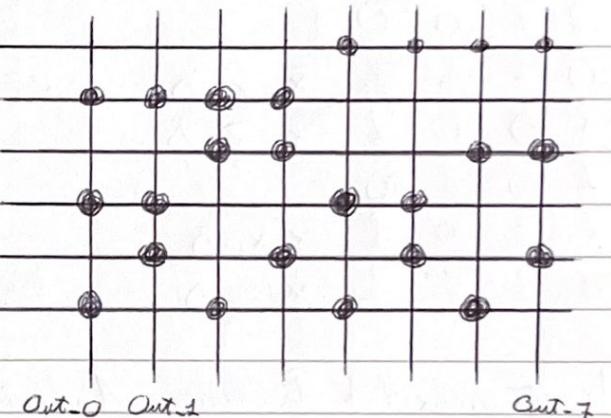
Il codificatore altro  
 non è che una serie di  
 porte OR.

Può essere anche rappresentato come una griglia

Decodificatore ha  $m$  input ed  $n$  output legati dalla relazione  $2^m = n$   
 Realizza la f. inversa del codificatore: da una sequenza di bit  
 genera il simbolo corrispondente. Una sola uscita vale uno, le altre 0



A	B	C	0	1	2	3	4	5	6	7
000	1	0	0	0	0	0	0	0	0	0
001	0	1	0	0	0	0	0	0	0	0
010	0	0	1	0	0	0	0	0	0	0
011	0	0	0	1	0	0	0	0	0	0
100	0	0	0	0	1	0	0	0	0	0
101	0	0	0	0	0	1	0	0	0	0
110	0	0	0	0	0	0	1	0	0	0
111	0	0	0	0	0	0	0	1	0	0



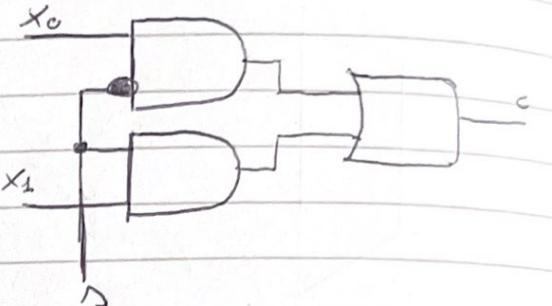
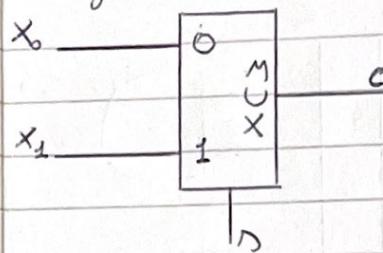
Il decodificatore è un insieme di porte AND

Può essere rappresentato come una griglia e a volte torna utile separare una variabile dal suo negato

Notare che ogni output è un mintermine, per cui è anche detto generatore di mintermini

È rappresentato anche come un rettangolo

Multiplexer ha  $m$  input,  $n$  linee di selezione ed una linea di output  $c$ .  
 Seleziona quale degli input verrà dato in output tramite i segnali di selezione. La relazione è  $n = \log_2 m$



$x_0$	$x_1$	$\sigma$	$c$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	$\bar{x}_0 \cdot x_1 \cdot \sigma$
1	0	0	$x_0 \cdot \bar{x}_1 \cdot \bar{\sigma}$
1	0	1	0
1	1	0	$x_0 \cdot x_1 \cdot \bar{\sigma}$
1	1	1	$x_0 \cdot x_1 \cdot \sigma$

$$c = \bar{x}_0 \cdot x_1 \cdot \sigma + x_0 \cdot \bar{x}_1 \cdot \bar{\sigma} + x_0 \cdot x_1 \cdot \bar{\sigma} + x_0 \cdot x_1 \cdot \sigma = \\ (\text{f. canonica SOP}) \\ = x_2 \cdot \sigma + x_0 \cdot \bar{\sigma}$$

espressione minimale

Dunque se  $\sigma = 0$  passa  $x_0$ ; se  $\sigma = 1$  passa  $x_1$

In un mux 4:1 troviamo 4 input ( $x_0, \dots, x_3$ ), 2 segnali ( $\sigma_0, \sigma_1$ ) e  $c$

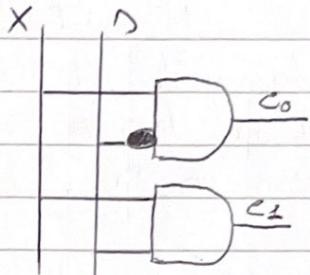
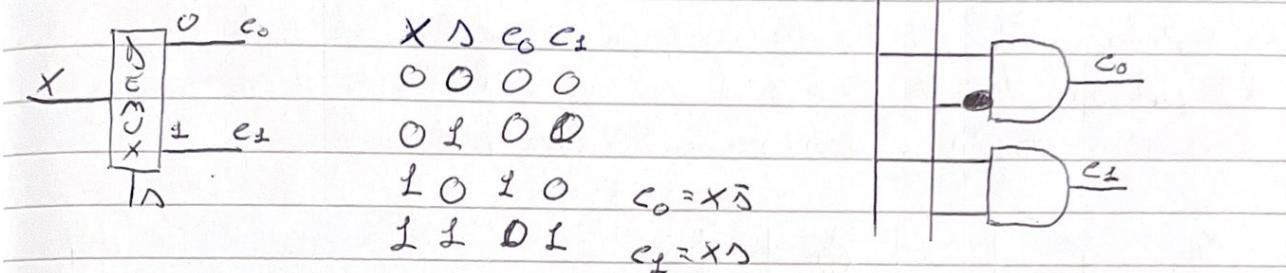
Possiamo immediatamente osservare il comportamento:

- Se  $\sigma_1 \sigma_0 = 00$  allora  $c = x_0$ .
- Se  $\sigma_1 \sigma_0 = 01$  allora  $c = x_1$ .
- Se  $\sigma_1 \sigma_0 = 10$  allora  $c = x_2$ .
- Se  $\sigma_1 \sigma_0 = 11$  allora  $c = x_3$ .

E quindi il circuito sarà composto da 4 porte AND a 3 ingressi ed un'unica porta OR.

Ponendo le  $n$  variabili sulle linee di selezione e la tavola di verità della funzione sulle  $2^n$  linee dati, un MUX può calcolare qualsiasi funzione booleana di  $n$  variabili.

Demultiplexer ha un solo input  $x$ ,  $m = \log_2 m$  linee di selezione ed  $m$  output. Indipendentemente dai segnali di selez., se l'input è 0 allora tutti gli output saranno 0; se l'input è 1, allora un solo output sarà 1.



Con un PLA è possibile rappresentare un insieme di funzioni; ha  $n$  input e  $o$  output e presenta  $m$  porte AND e  $o$  porte OR

Si consideri una funzione con  $A, B, C$  input ed  $D, E, F$  output:

$D=1$  se almeno uno degli input è 1;  $E=1$  se esattamente due input sono 1;  $F=1$  se solo se tutti e tre gli input sono 1

$A \ B \ C \ | \ D \ E \ F$

0 0 0 | 0 0 0

0 0 1 | 1 0 0

0 1 0 | 1 0 0

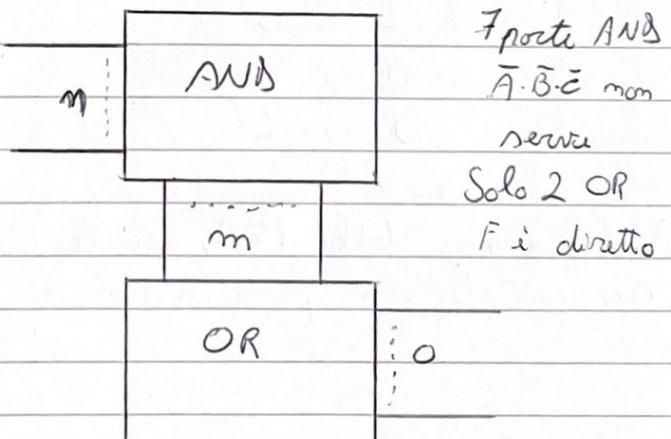
0 1 1 | 1 1 0

1 0 0 | 1 0 0

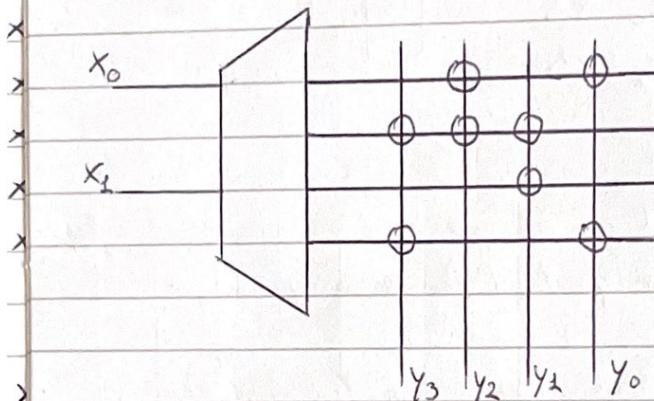
1 0 1 | 1 1 0

1 1 0 | 1 1 0

1 1 1 | 1 0 1



La **ROM** serve a memorizzare informazioni che non è necessario modificare.  
 È una tabella di **height** righe e **width** colonne in cui ciascuna cella rappresenta una locazione di memoria. La **PROM** è programmabile un'unica volta, mentre la **EPROM** è cancellabile con luce ultravioletta.  
 È costituita da un decoder legato a una catena di porte OR:  
 l'input al decoder indica il numero di locazioni di memoria,  
 e ciascun'uscita è connessa a ciascuna porta OR.



$X_0$	$X_1$	$Y_3$	$Y_2$	$Y_1$	$Y_0$
0	0	0	1	0	1
0	1	1	1	1	0
1	0	0	0	1	0
1	1	1	0	0	1

A differenza della PLA, qui le funzioni non possono essere rappresentate in forma minimale.

Data la tavola di verità del NAND

$$A \ B \ \bar{A} \cdot \bar{B}$$

$$0 \ 0 \ 1$$

$$0 \ 1 \ 1$$

$$1 \ 0 \ 1$$

$$1 \ 1 \ 0$$

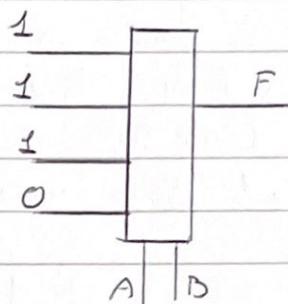
$$F = \bar{A} \cdot \bar{B} + \bar{A} \cdot B + A \cdot \bar{B} \quad (\text{canonica SOP})$$

La funzione realizzata dal MUX 4:1

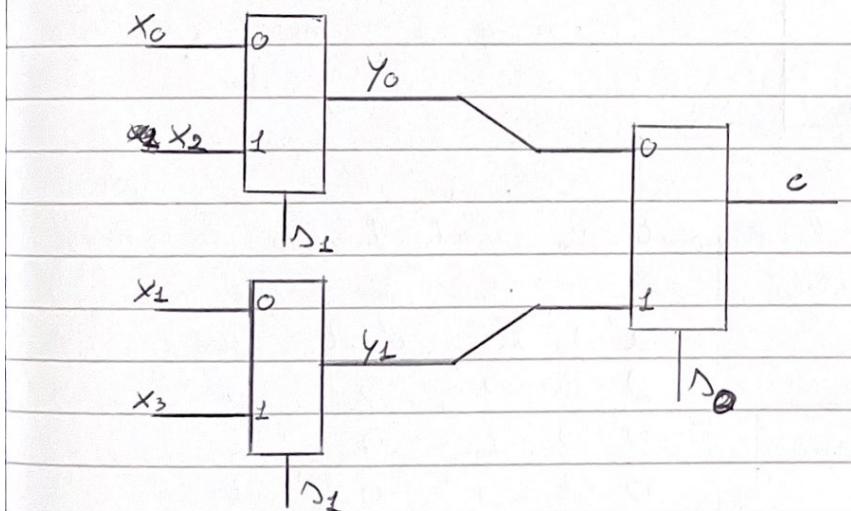
$$x_0 \bar{x}_1 \bar{x}_0 + x_1 \bar{x}_2 x_0 + x_2 x_1 \bar{x}_0 + x_3 x_2 x_0$$

restituire la f. NAND se

$$x_0 = 1, x_1 = 1, x_2 = 1, x_3 = 0$$



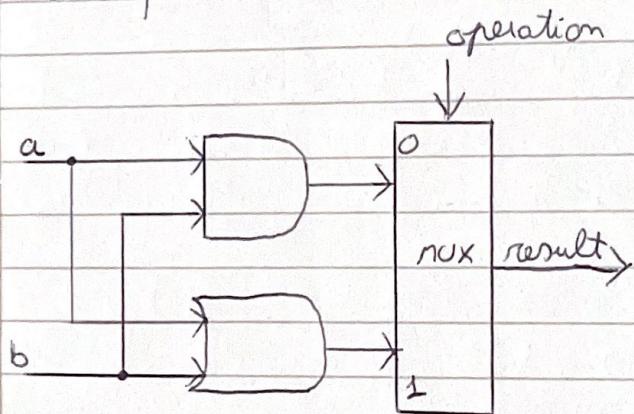
Realizzare un MUX 4:1 con tre MUX 2:1



L'Arithmetic Logic Unit è un circuito combinatorio che trova nel processore e si occupa di eseguire:

- Funzioni logiche: AND, OR e NOR (qui si aggiungono)
- Funzioni aritmetiche: somma e sottrazione
- Istruzioni di confronto:  $a < b$ ;  $a = b$ ;  $a \neq b$

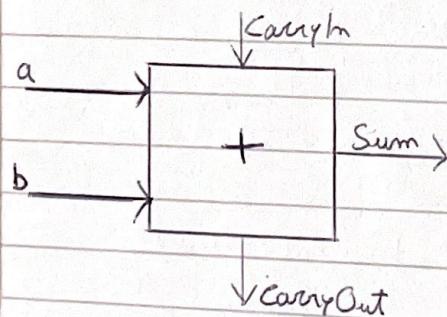
Nel MIPS, la parola macchina è a 32 bit e quindi useremo 32 copie in cascata di una ALU a 1 bit. Quest'ultima è costituita da un MUX 4:1 (AND, OR, somma, confronti) i cui 2 segnali di controllo sono forniti dalla unità di controllo (CU)



Lo costruiremo passo dopo passo iniziando con un MUX 2:1 in cui

$$\begin{aligned} \text{se } op. = 0 &\Rightarrow \text{result} = a \text{ AND } b \\ \text{se } op. = 1 &\Rightarrow \text{result} = a \text{ OR } b \end{aligned}$$

L'addizionatore a 1 bit è la componente che consente di eseguire la somma.



a	b	C <sub>in</sub>	C <sub>out</sub>	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Ricaviamo quindi il funzionamento:

$$C_{out} = \bar{a}bC_{in} + a\bar{b}C_{in} + ab\bar{C}_{in} + a\bar{b}C_{in}$$

← canonica SOP

$$= (a \oplus b) \cdot C_{in} + ab$$

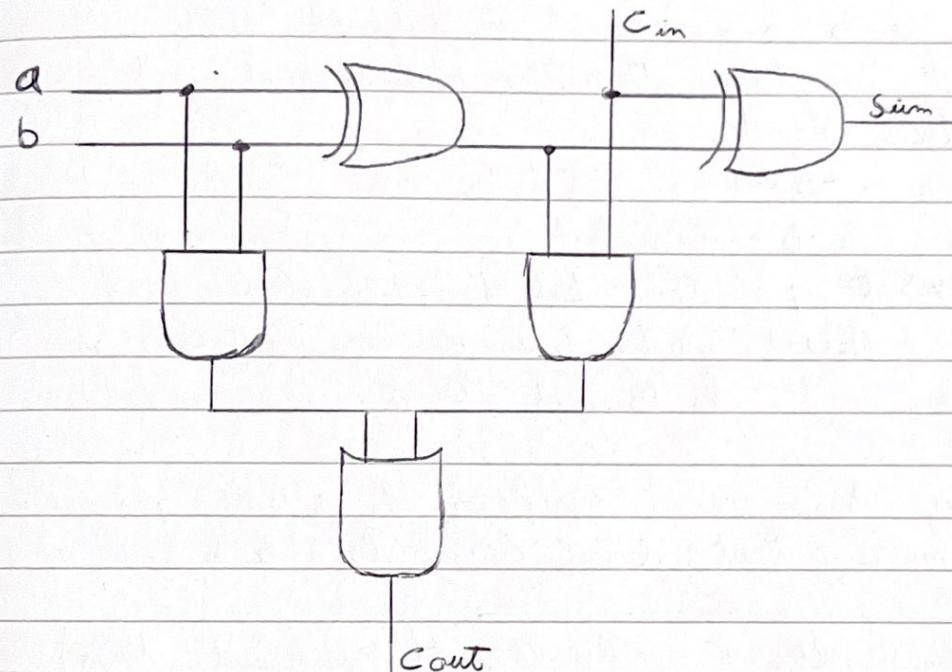
← minimale

$$Sum = \bar{a}\bar{b}C_{in} + \bar{a}b\bar{C}_{in} + a\bar{b}\bar{C}_{in} + abC_{in}$$

← canonica SOP

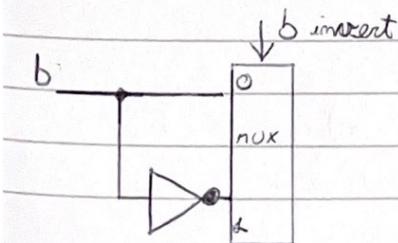
$$= a \oplus b \oplus C_{in}$$

← minimale



Aggiungendo questa nuova componente ad un immagazzinatore MUX a tre input, non c'è ditta che replicarlo fino ad avere 32 copie, tenendo conto che  $a_0$  è il bit meno significativo. A questo punto aggiungeremo una modifica per poter calcolare  $a - b$ :

$a - b = a + (-b)$  e sappiamo che in e2 l'opposto di  $b$  si ottiene invertendo i bit di  $b$  e sommando 1



Questo MUX 2:1 decide se invertire o meno  $b$  e inoltre il  $C_{in}$  della ALU, sarà ~~sempre~~ posto ad 1 anziché 0

L'istruzione di confronto è SLT:

se  $a < b$  il risultato è 1, 0 altrimenti;

set Result to 1 if a is less than b

L'output è una stringa a 32bit in cui l'1 e lo 0 sono posti come cifra meno significativa im fondo a una stringa di zero.

Per stabilire se  $a < b$  si osserva il risultato della differenza tra due il bit più significativo della sottrazione sarebbe chiamato Set e viene dato in output dalla ALU31:

se  $a < b$  allora  $a - b < 0$  e Set = 1;

se  $a \geq b$  allora  $a - b \geq 0$  e Set = 0.

Per cui  $\text{Result}_0 = \text{Set}$ ; l'output Set di ALU 31 viene suddiviso sull'input Less di ALU 0, mentre tutti gli altri input Less sono posti a 0. Notiamo inoltre che Borrow e CarryIn sono inutilizzabili.

Per il test di egualanza serve verificare che  $a - b = 0$  per cui è necessario un nuovo output: il bit Zero, che vale 1 se  $\text{Result} = 0$ .

---

$$\text{Zero} = \text{Result}_{31} + \text{Result}_{30} + \dots + \text{Result}_0 \leftarrow (\text{Richiede porta NOR})$$

Aggiungiamo anche l'output overflow a 1bit all'ALU31

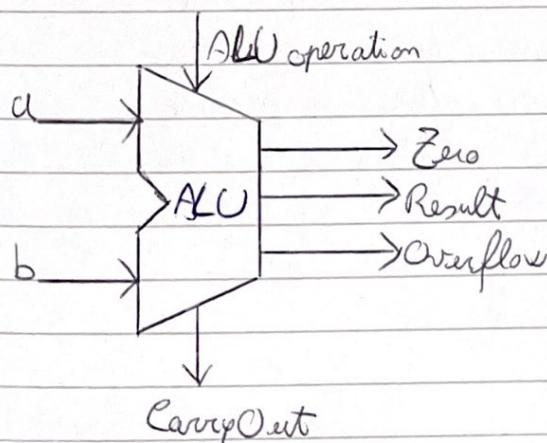
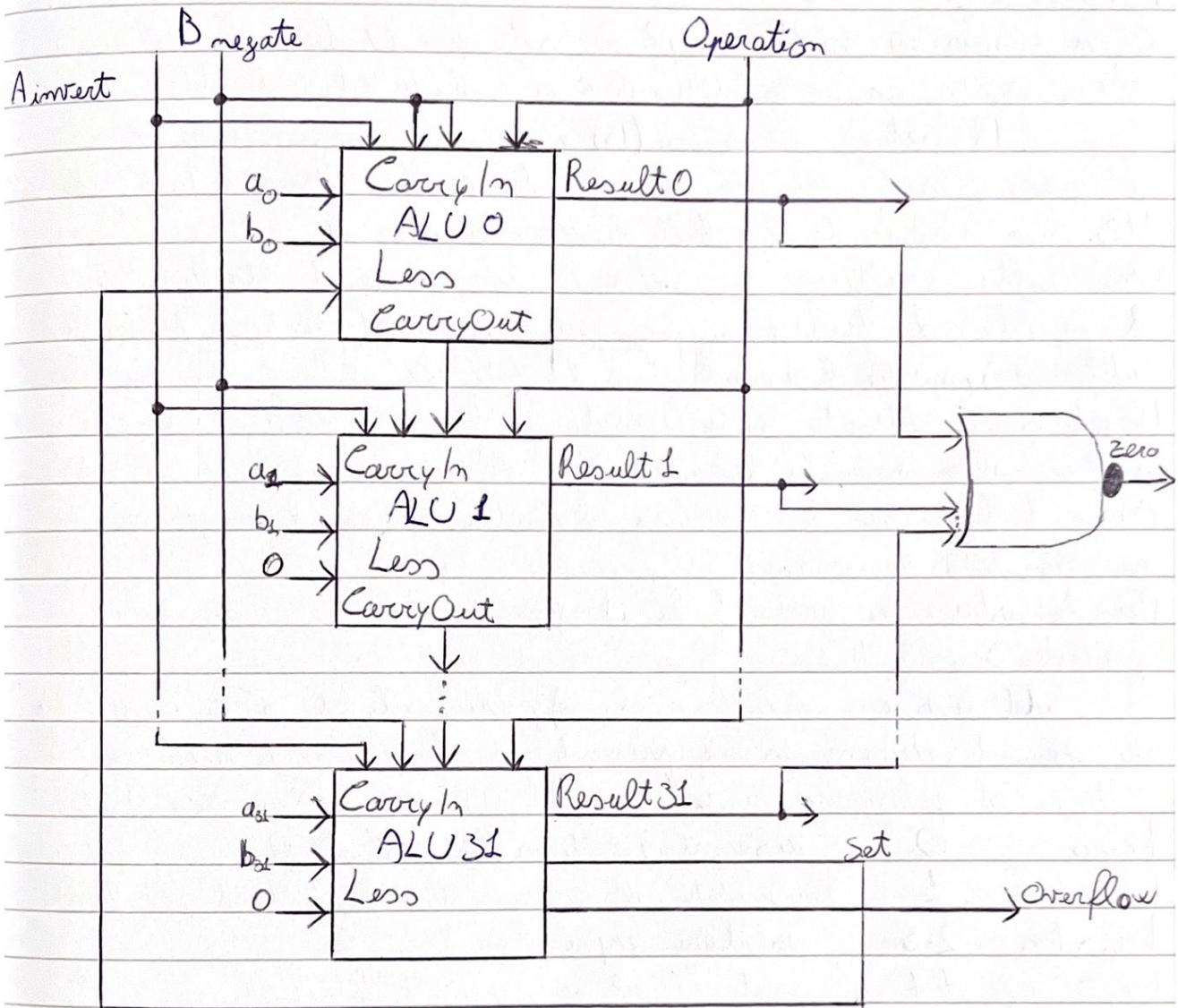
Dovendo introdurre anche la funzione  $\text{NOR}(a, b)$  ricordando che:

$$\text{NOR}(a, b) = \text{NOT}(\text{OR}(a, b)) = \text{AND}(\text{NOT}(a), \text{NOT}(b))$$

Invertiamo anche a usando un MUX 2:1 ed il segnale di controllo AimExt. Poiché

$$\text{NAND}(a, b) = \text{NOT}(\text{AND}(a, b)) = \text{OR}(\text{NOT}(a), \text{NOT}(b))$$

Poniamo anche aggiungere il comando NAND invertendo a e b e usandoli come input di una porta OR



control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	slt
1100	NOR
1101	NAND

## PROCESSORE MIPS

È un processore molto semplice usato per Nintendo 64, PS, PS2, PSP, che presenta un'architettura RISC cioè Reduced Instruction Set Computer. Il set di istruzioni (IS) è il "linguaggio" di un processore e ognuno presenta il suo, che presenta poche differenze dagli altri.

N.B.: non saranno trattate tutte le istruzioni presenti

Una volta scritto un programma in linguaggio ad alto livello, il compilatore lo traduce in linguaggio assembly (riferito al processore utilizzato), mentre l'assemblatore lo traduce in linguaggio macchina. Il file oggetto ottenuto viene passato al linker, o link editor, che collega altri moduli e librerie per ~~col~~ creare un singolo eseguibile. Prima di lanciarne l'esecuzione, il Loader carica il programma in un'area della memoria.

Per le istruzioni aritmetiche il formato del MIPS è rigido e prevede 3 operandi:

add a, b, c      ~~(un commento)~~ corrisponde ad  $a = b + c$

Gli operandi devono essere presenti in locazioni di memoria interne al processore: i registri. Nel MIPS ce ne sono 32 a 32bit.

\$zero      0      riservato; contiene la costante 0

\$at      1      riservato all'assemblatore per costanti molto lunghe

\$v<sub>0</sub> - \$v<sub>1</sub>      2-3      valutare espressione

\$a<sub>0</sub> - \$a<sub>3</sub>      4-7      usati per gli argomenti delle operazioni

\$t<sub>0</sub> - \$t<sub>7</sub>      8-15      usati per variabili temporanee

\$d<sub>0</sub> - \$d<sub>7</sub>      16-23      usati per memorizzare variabili

\$e<sub>8</sub> - \$e<sub>15</sub>      24-25      usati per variabili temporanee

\$k<sub>0</sub> - \$k<sub>1</sub>      26-27      riservati al sistema operativo

\$gp      28

\$sp      29

\$fp      30

\$ra      31

Per evitare di allocare memoria per le costanti, esistono op. immediati:

$\$r_3$  contiene  $f$  l'operazione  $f = f + 4$  diventa

addi  $\$r_3, \$r_3, 4$  ↑ anche negativa

Poiché si opera tra registri, abbiamo due operazioni volte al trasferimento:

Load va dalla memoria al registro

Store va dal registro alla memoria

Poiché ognuna delle  $2^{32}$  celle di memoria può contenere un byte, ogni parola necessita di 4 locazioni consecutive e dunque si possono memorizzare al massimo  $2^{30}$  parole, le quali sono memorizzate in formato esadecimale.

lw  $\$t_0, 4(\$r_3)$  ✖ carica in  $t_0$  la word che si trova all'indirizzi

✖ dato dal contenuto di  $r_3$  a cui si somma 4

sw  $\$t_0, 32(\$r_3)$  ✖ registra la word contenuta in  $t_0$  nella

✖ loc. d. m. al cui indirizzo si dato dal

✖ contenuto di  $r_3$  a cui si somma 32

La stringa corrispondente ad add  $\$t_0, \$r_1, \$r_2$  è

op	rs	rt	rd	shamt	funct
000000	10001	10010	01000	00000	100000

Questo è detto formato R:

OP è il codice operativo della ~~istruzione~~

rs primo operando sorgente }

rt secondo = = } corrispondono alla num. del registro

rd operando destinazione }

shamt shift amount (richiesto solo per alcune istruzioni)

funct definisce la funzionalità specifica dell'istruzione

Nota: OP è la stessa per add e sub, mentre funct è risp. 32 e 34.

Per le operazioni con costanti e quelle di trasferimento c'è il formato I:

OP a 6 bit

n a 5 bit

nt a 5 bit

costante o indirizzo a 26 bit

Le operazioni logiche AND OR NOR XOR sono in formato R e sono anche presenti due operazioni immediate: andi e ori. Per quanto riguarda il NOT, basta fare un NOR <sup>tra</sup> il registro \$to e quello \$zero.

Per supportare XOR dovranno ingrandire la nostra ALU per aggiungere la porta corrispondente agli input possibili.

Ci sono poi le operazioni di scorrimento/traslazione/shift che vengono eseguite da un'altra unità, lo shift register.

Lo shift logico a sinistra srl inserisce un certo numero di zeri a partire dalla pos. meno signif. di una word. L'operazione corrisponde ad una moltiplicazione per  $2^n$  a patto che il risultato rientri nell'intervallo di rapp. con 32bit in C2.

Lo shift logico a dx srl inserisce n zeri a partire dalla cifra più significativa. Può corrispondere ad una divisione intera, a patto che il numero di partenza sia positivo.

Lo shift aritmetico (a dx) sra inserisce n bit uguali al bit di segno a partire dalla cifra più significativa, e corregge il problema che si ha volendo usare lo srl per le divisioni.

Lo shift aritmetico a sx sra comporta come quello logico, per cui è raramente implementato.

Anche queste sono in formato R e ~~selezionano~~ utilizzano il campo shift amount.

**ES.**  $a = !b \parallel (c \& d);$  con  $a$  in  $s_0; b s_1; c s_2 \text{ e } d s_3$

and  $\$t_0, \$s_2, \$s_3$  # il reg.  $t_0$  contiene  $c \& d$   
nor  $\$t_2, \$s_2, \$zero$  #  $t_2$  contiene  $!b$   
or  $\$s_0, \$t_2, \$t_0$  #  $s_0$  contiene  $!b \parallel (c \& d)$

**ES.**  $f = 2^*(g + A[4]);$  fin  $s_0; g s_2; A s_2$  (ind. base)

lw  $\$t_0, 16(\$s_2)$  # carica in  $t_0$  la word all'indirizzo  $s_2 + 16$  eroi  $A[4]$   
add  $\$t_0, \$s_2, \$c_0$  #  $t_0$  contiene  $g + A[4]$   
sll  $\$s_0, \$t_0, 1$  #  $s_0$  contiene  $2^*(g + A[4])$

**ES.**  $B[8] = A[i-j];$   $i$  in  $s_0; j s_1;$  ind. base  $A s_2;$  ind. b.  $B s_3$

sub  $\$t_0, \$s_0, \$s_1$  #  $t_0$  contiene  $i-j$   
sll  $\$t_1, \$t_0, 2$  #  $t_1$  contiene  $4^{*(i-j)}$   
add  $\$t_2, \$s_2, \$t_1$  #  $t_2$  contiene l'ind. b. di  $A + 4^{*(i-j)}$   
lw  $\$t_3, 0(\$t_2)$  # carica in  $t_3$  la word contenuta in  $t_2,$  eroi  $A[i-j]$   
sw  $\$t_3, 32(\$s_3)$  # registra la word contenuta in  $t_3$  all'indirizzo di  
#  $s_3$  a cui va sommato 32, eroi  $B[8]$

**ES.**  $B[42] = A[ij] + A[5j];$   $i$  in  $s_0; j s_1;$  ind. b.  $A s_6;$  ind. b.  $B s_7$

sll  $\$t_0, \$s_0, 2$  #  $t_0$  contiene  $4^*i$   
add  $\$t_0, \$s_6, \$t_0$  #  $t_0$  contiene l'ind. b. di  $A$  sommato a  $4^*i$   
sll  $\$t_1, \$s_1, 2$  #  $t_1$  contiene  $4^*j$   
add  $\$t_2, \$s_6, \$t_1$  #  $t_2$  contiene l'ind. b. di  $A$  sommato a  $4^*j$   
lw  $\$t_3, 0(\$t_2)$  # carica in  $t_3$  la word co  $A[ij]$   
lw  $\$t_3, 0(\$t_1)$  # carica in  $t_3$   $A[5j]$   
add  $\$t_4, \$t_2, \$t_3$  #  $t_4$  contiene  $A[ij] + A[5j]$   
sw  $\$t_4, 48(\$s_7)$  # registra la word di  $t_4$  in  $B[42]$

**[ES]**  $f$  in  $t_0$ ;  $g$  in  $t_1$ , ind  $b$  in  $t_0$ ; ind  $b$ .  $B$  in  $t_2$

sll \$t\_0, \$t\_0, 2      \*  $t_0$  contiene  $f^*$   
add \$t\_0, \$t\_0, \$t\_0    \*  $t_0$  contiene ind  $b$ .  $A + f^*$   
sll \$t\_2, \$t\_2, 2      \*  $t_2$  contiene  $g^*$   
add \$t\_2, \$t\_2, \$t\_2    \*  $t_2$  contiene ind  $b$ .  $B + g^*$   
lw \$t\_0, 0(\$t\_0)      \* carica in  $t_0$  la word  $A[f]$   
addi \$t\_2, \$t\_0, 4      \*  $t_2$  contiene  $t_0 + 4$   
lw \$t\_0, 0(\$t\_2)      \* carica in  $t_0$  la word  $A[f+1]$   
add \$t\_0, \$t\_0, \$t\_0    \*  $t_0$  contiene  $A[f] + A[f+1]$   
sw \$t\_0, 0(\$t\_2)      \* registra in  $B[g]$  la word di  $t_0$

Il codice corrisponde a  $B[g] = A[f] + A[f+1]$

Il MIPS offre istruzioni di salto condizionato e incondizionato:  
Branch if Equal e Branch if NOT Equal hanno formato I,  
presentando operazione, i due registri da confrontare e l'etichetta  
o indirizzo di salto. Per calcolare quest'ultimo si usa il registro  
PC (Program Counter), il quale contiene l'indirizzo dell'istruzione  
corrente e viene incrementato non appena un'istruzione viene  
prelevata. Per sommare a questo i 16 bit dell'ind. di salto, questo  
subisce prima un'estens. del segno per essere portato a 32 bit e poi  
si applica uno shift logico a sx di 2 posti. Il procedimento è  
detto Indirizzamento relativo al Program Counter

N.B.: si preferisce l'uso di bne perché dopo l'istruzione di salto  
c'è l'istruzione corrispondente alla prima direzione della scelta.

L'istruzione Jump ha codice op 2 e 26 bit dedicati all'etichetta  
(formato J). Il calcolo dell'indirizzo avviene per Indirizzamento pseudo-  
diretto: 4 bit più segn. del PC e contenuto di J dopo uno shift a sx di  
2 posti

Esistono infine altre 3 modalità di **indirizzamento**:

immediato, tramite registro e tramite registro base e spostamento

Tramite l'istruzione Set if Less Than (immediato) è possibile confrontare il contenuto di un registro con un altro (o con una costante), settando a 1 il valore di un registro nel caso la condizione sia verificata.

È possibile usarla assieme alle istruzioni bne e bne ed al registro \$zero per i salti su condizione di disegualanza.

Esistono anche le versione unsigned

Le **procedure** o **funzioni** permettono di dividere il programma in moduli riutilizzabili e sono composti da **caller** e **callee**.

La prima deve rendere disponibili i parametri di ingresso e cedere il controllo, mentre la seconda prima alloca lo spazio necessario e dopo aver svolto il proprio compito rende il risultato accessibile e restituisce il controllo.

L'istruzione per saltare alla procedura è **Jal** (Jump And Link), di formato I e codice op 3, che salta all'indirizzo con etichetta Label e salva l'indirizzo di ritorno in \$ra.

Per restituire il controllo c'è invece **JR** (Jump Register), di formato R con funct=8, con rs = 31 (perché salta a \$ra) e altri campi = 0. Prima che i registri vengano riportati ai valori pre-chiamata i loro valori vengono copiati in memoria. Il registro \$sp ha un puntatore all'ultimo dato inserito nello **stack**, una struttura dati di tipo Last-In-First-Out che si trova in memoria e cresce da indirizzi alti verso indirizzi bassi. Il valore di \$sp si aggiorna ad ogni inserimento di nuovi valori (**push**) e ad ogni prelievo (**pop**). Per il PUSH si tiene documentato di 4 e c'è una svolta mentre per il POP c'è una dw e poi un incremento di 4.

Altro a \$ra per l'indirizzo, conventionalmente il MIPS usa i registri \$a<sub>0</sub>, ..., \$a<sub>4</sub> per il passaggio dei parametri e \$v<sub>0</sub>, \$v<sub>1</sub> per la restituzione dei valori.

int somma (int g, int h, int i, int j)  
{  
 int f;  
 f = (g + h) - (i + j);  
 return f;  
}

È una procedura  
foglia perché non  
chiama altre procedure

add \$sp, \$sp, -12      \* apre uno spazio per 3 word  
sw \$s0, 8(\$sp)      \* salva nello stack il registro \$s0 (per f)  
sw \$s2, 4(\$sp)      \*        "                                  \$s2 (per g+h)  
sw \$s2, 0(\$sp)      \*        "                                  \$s2 (per i+j)

add \$s2, \$a<sub>0</sub>, \$a<sub>1</sub>      \* in \$s2 c'è g+h  
add \$s2, \$a<sub>2</sub>, \$a<sub>3</sub>      \* in \$s2 c'è i+j  
sub \$s0, \$s2, \$s2

add \$v<sub>0</sub>, \$s0, \$zero      \* restituzione di f  
lw \$s2, 0(\$sp)      } \* ripristina i registri con un pop  
lw \$s2, 4(\$sp)      } (deallocazione memoria)  
lw \$s0, 8(\$sp)  
add \$sp, \$sp, 12      \* aggiorna \$sp dopo l'eliminazione di 3 elem  
jr \$ra      \* ritorno al programma

La procedura è divisa in prologo - corpo - epilogo

Per evitare di salvare e ripristinare registri il cui valore non sarà più utilizzato, il MIPS usa questa divisione:

$\$t_0, \dots, \$t_r$  per reg. temporanei il cui contenuto non viene salvato  
 $\$r_0, \dots, \$r_s$  per reg. il cui contenuto deve essere salvato

Il codice si riduce quindi a:

add  $\$t_1, \$r_0, \$r_1$

add  $\$t_2, \$r_2, \$r_3$

sub  $\$r_0, \$t_1, \$t_2$

$\$r_2$

**[ES.1]**  $\$r_0$  contiene 1;  $\$r_2$  contiene 20

slt  $\$t_0, \$r_0, \$r_1$   $\nabla t_0 = 1$  perché  $1 < 20$

brne  $\$t_0, \$r_2, \text{LABEL}$   $\nabla$  salto eseguito

add  $\$r_0, \$r_0, \$r_1$   $\nabla$  non eseguita

LABEL: sll  $\$r_0, \$r_0, 1$   $\nabla r_0$  shiftato di 1 pos.  $\rightarrow$  contiene 2

**[ES.2]**  $\$r_0$  contiene 2 +  $\$r_2$  contiene 6

LOOP: beg  $\$r_0, \$r_1, \text{EXIT}$   $\nabla$  fin quando  $\$r_0 \neq \$r_2$   $\Rightarrow \$r_0$  verrà incrementato di 1, dunque a fine ciclo  $\$r_0 = \$r_2 = 6$

addi  $\$r_0, \$r_0, 1$

$\nabla$  LOOP

EXIT: ...

**[ES.3]**  $\$r_2$  variabile intera result;  $\$r_0$  ind. base di M

addi  $\$t_1, \$zero, 0$   $\nabla t_1 = 0$

CICLO: lxi  $\$t_2, 0(\$r_0)$   $\nabla$  carica in  $\$r_2$  la word  $M[0]$

add  $\$r_2, \$r_2, \$r_2$   $\nabla r_2$  contiene result +  $M[0]$

addi  $\$r_0, \$r_0, 4$   $\nabla r_0$  contiene l'ind. di  $M[2]$

add  $\$t_1, \$t_1, 1$   $\nabla t_1 = 1$

slti  $\$t_2, \$t_1, 100$   $\nabla t_2 = 1$  perché  $1 < 100$

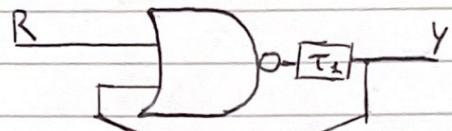
brne  $\$t_2, \$zero, \text{CICLO}$   $\nabla$  torna a ciclo perché  $1 \neq 0$

for ( $i=0$ ;  $i < 100$ ;  $i++$ ) {

    result +=  $M[i];$

}

Nelle reti sequentiali (in cui l'output dipende anche dallo stato del sistema, cioè da input precedenti) vengono usate forme d'onda temporali binarie; O è il valore istante per istante, ma viene presa in considerazione anche la componente tempo. Dopo le porte logiche verrà quindi inserito un blocco  $\boxed{\tau}$  dove  $\tau = \text{ritardo di propagazione}$ . Il circuito sequenziale di base è il latch S-R



Due porte NOR interconnesse con un segnale di feedback



$$Y = \overline{R + Z} = \overline{R + (\overline{S + Y})} = \overline{R}(S + Y)$$

seppur la stessa  $Y$ , queste sono prese in momenti diversi per via del ritardo

Supponendo  $Y=1$  e  $Z=0$  all'istante  $t_0$ , e com  $R=S=0$  si ha

- $Y=1$  all'istante  $t_0 + \tau_1$  Stabilità = lo stato non varia
- $Z=0$  all'istante  $t_0 + \tau_1 + \tau_2$

Se a  $t_1$  si ha  $R=1$ :

- $Y=0$ , mentre  $Z=1$   $\Rightarrow$  cambiare R li ha invertiti

Se a  $t_2$  R torna a valer 0:

- $Y=0$  e  $Z=1$   $\Rightarrow$  cambiare R li ha lasciati invariti

Analogamente, se a  $t_0$  abbiamo  $Y=0$ ,  $Z=1$  e  $R=S=0$

Se  $S=1$  a  $t_1$ :

- $Z=0$  e  $Y=1$

Se a  $t_2$  S torna a 0:

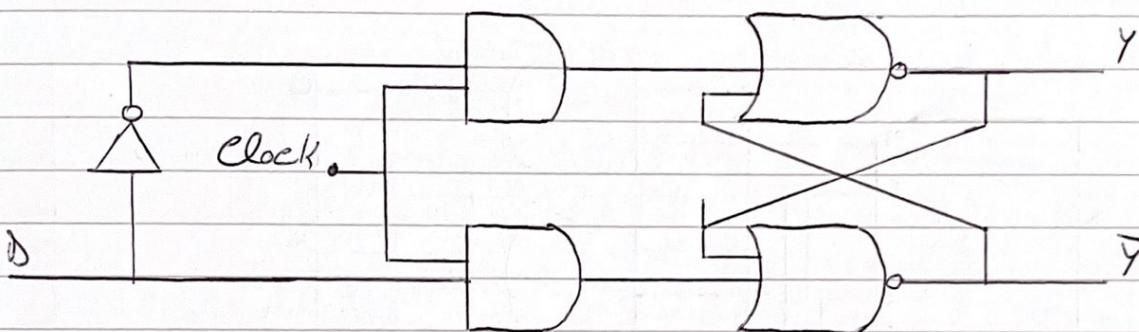
- $Z=0$  è mantenuta, così come  $Y=1$

Notiamo quindi che vale la relazione  $Z = \overline{Y}$ . Si viene detto Set perché fa diventare  $Y=1$ , mentre R è Reset

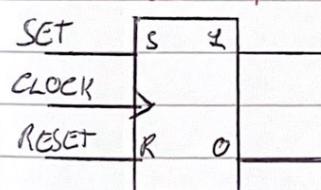
Va notato che  $R=S=1$  non può avvenire, dunque la configurazione è detta illegale e vale la condizione  $R \text{ AND } S = 0$

Per controllare gli istanti in cui un elemento di memoria può cambiare stato si usa il **clock**, una forma d'onda che fa da input a due porte AND anteposte al latch S-R. Evita che un dato sia contemporaneamente letto e scritto.

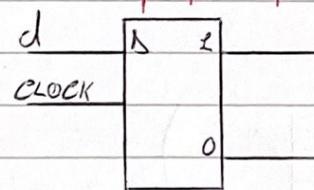
Ottieniamo quindi un **Flip-Flop SR** che va modificato per impedire la conf.  $R=S=1$  aggiungendo una porta NOT, ottenendo la res. **D**



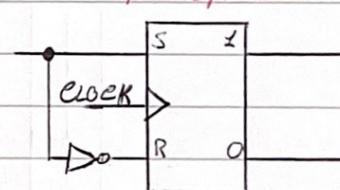
S. circuitale Flip-Flop SR



Flip-Flop D



Flip-Flop D

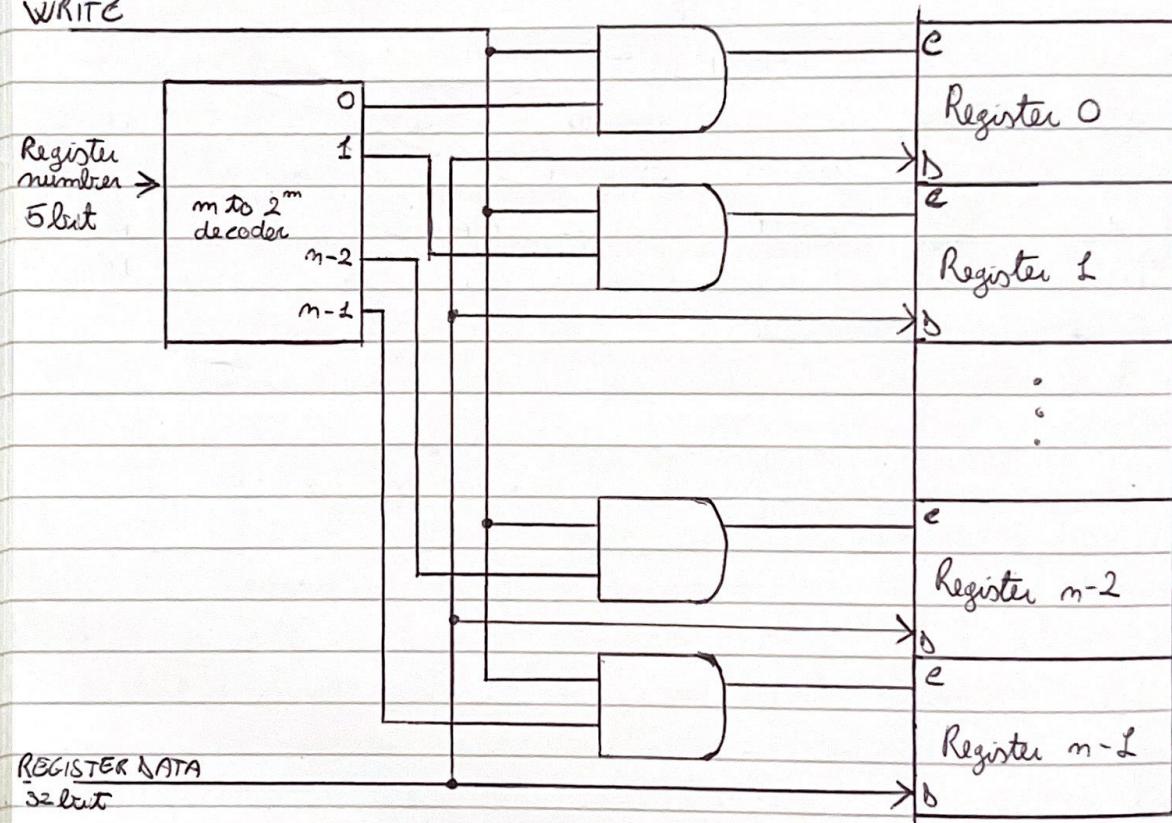


Un **registro** del MIPS è un array di 32 Flip-Flop D dotati di uno stesso input **clock** in modo da essere sincronizzati. L'input proviene da una porta AND che riceve un segnale di **clock** e uno di **Write**: se **Clock=Write=1** si sta scrivendo, cambiando quindi l'output dei Flip-Flop D, mentre se **Clock AND Write=0** lo stato non viene cambiato e si sta eseguendo una lettura.

Il Banco dei registri o Register file costituisce l'insieme dei 32 registri. Per la lettura di un registro viene usato un MUX 32:1 che riceve in input i contenuti dei 32 registri, trae i 5 segnali di controllo dai 5 bit del register number e restituisce il contenuto del registro indicato.

Per quanto riguarda la scrittura, viene usato un decoder a 5 input per determinare il registro in cui scrivere ed il suo output entra in una porta AND assieme al Write per abilitare la scrittura del registro corrispondente.

WRITE



Registri e banco dei registri costituiscono i blocchi per piccole memorie all'interno del processore.

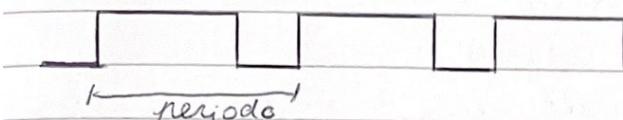
Maggiori quantità di memoria sono fornite da SRAM e DRAM.

## Implementazione a ciclo singolo del MIPS

Ciascuna istruzione viene eseguita in un ciclo singolo di clock ed il periodo viene misurato in picosecondi ( $10^{-12}$ ) o nanosec. ( $10^{-9}$ )

Hertz (clic al sec):

Mega  $10^6$  e Giga  $10^9$



Le istruzioni iniziano sul fronte di salita e ~~hanno~~ tutte stessa durata: sono fatte per la più lenta (e sprecano tempo)

Ogni istruzione condivide i primi due passaggi:

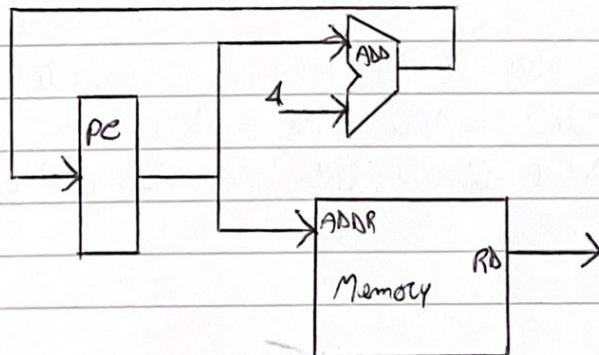
- Prelievo dell'istruzione dalla memoria (fetch)
- Lettura dei valori dei registri operandi

Mentre poi si presentano le differenze: quelle di accesso alla memoria prelevano o salvano il dato in memoria, quelle aritmetico logiche memorizzano il dato nel reg. di destinazione e quelle di salto condizionato (anche a non richiedere l'ALU) cambiano l'indirizzo dell'istruzione successiva.

Per incrementare il PC si utilizza un sommatore, un circuito combinatorio ottenuto impostando una ALU sempre su somma

La Memoria Istruzioni fornisce solo un accesso in lettura per fornire l'istruzione a 32 bit a partire dall'indirizzo

La Memoria dati ha come input l'indirizzo del dato e il dato da scrivere e restituisce il dato letto ricevendo a due segnali di controllo: MemWrite e MemRead (mutually excl.)

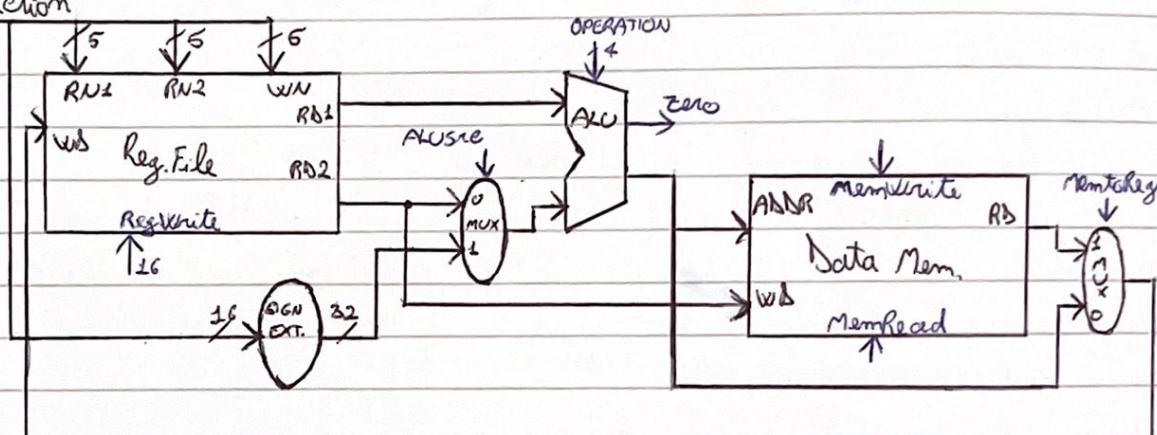


Le istruzioni in formato R non necessitano di componenti aggiuntivi, mentre quelle in formato I (load e store) hanno bisogno di una unità per l'estensione del segno per avere un offset a 32 bit.

Quest'ultima serve anche per i salti condizionati, i quali richiedono una ulteriore unità per eseguire lo shift a sx di 2 posti.

Combinando il tutto si aggiungono i multiplexer necessari per selezionare dati dalla sorgente richiesta.

Instruction



RN1 (2) = Read Register 1 (2)

WN = Write register

RD1 (2) = Read data 1 (2)

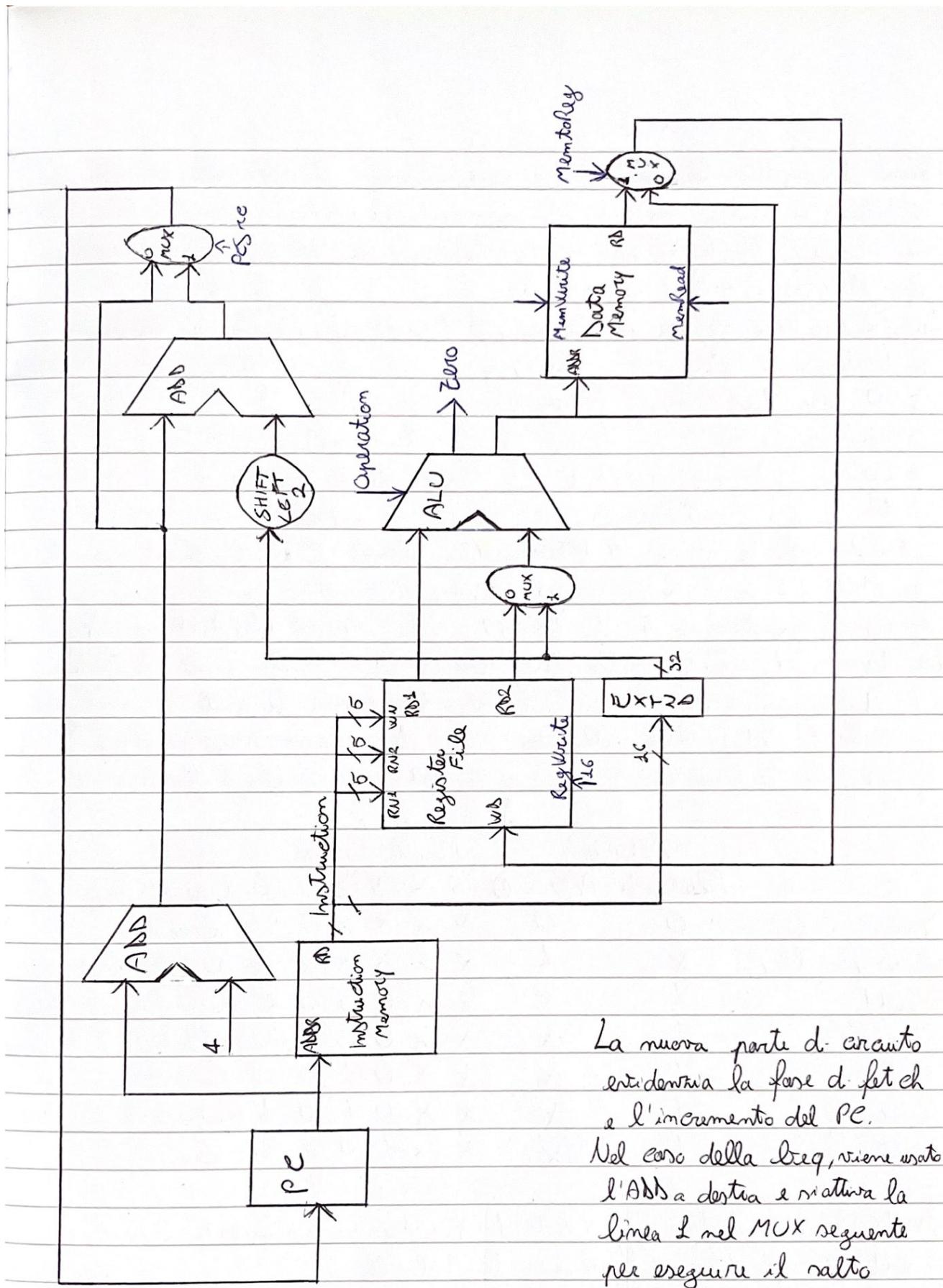
WD = Write data

ADDR = Address

• Il primo MUX dà all'ALU se il secondo input è il dato di un registro o un indirizzo

• Il secondo MUX sceglie se il dato da memorizzare sia il dato prelevato in memoria o il risultato dell'operazione

R: tra registri, RD1 e RD2 all'ALU (MUX 0), direttamente all'altro MUX (0) e indietro l.w.: RN1, WN, est. e MUX 1 all'ALU, ADDR  $\Rightarrow$  RD, MUX 1 e indietro r.w.: RN1 e 2, est. e RD1 all'ALU, Data Mem = ALU + RD2



La nuova parte d- circuito  
estendeva la fase d- fetch  
e l'incremento del PC.

Nel corso della breq, viene usata  
l'ADD a destra e riattiva la  
linea l nel MUX seguente  
per eseguire il salto

Prima di implementare la vera e propria unità di controllo, c'è da costruire un'unità combinatoria più piccola: l'unità di controllo della ALU. Escludendo l'istruzione NOR e quindi il segnale  $A_{Inv}$ , consideriamo una ALU con 3 segnali di controllo.

La ALU control riceve il campo funct dell'istruzione e un campo di controllo su 2 bit, detto ALUOp:

- 00 per lw e sw (operazione: somma)
- 01 per breq ( // : sottrazione)
- 10 per istr. in formato R ( " dipende da funct)
- 11 è una combinazione che non si ha mai

Notando che c'è una differenza tra formato R ed I, viene aggiunto un MUX 2:1 per selezionare il registro di scrittura/target, che compare rispettivamente da 15 a 12 e da 20 a 26 (RegDst segnale).

La tavola di verità della ALU Control è in realtà molto piccola perché ci interessa solo un piccolo sottoinsieme di tutte le combinazioni possibili ed inoltre molti dei valori di ingresso sono indifferenti cioè non influenzano il comportamento e sono quindi trascurabili.

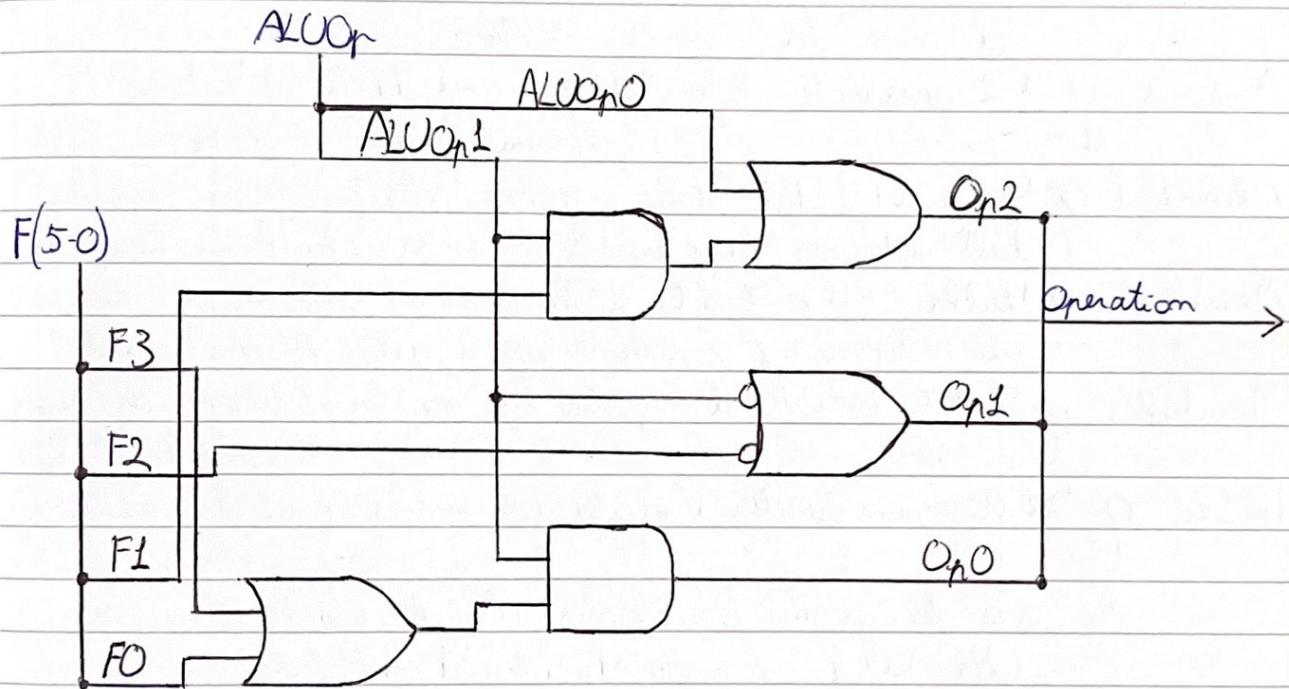
ALUOp	Funct Field								
	ALUOp 1	ALUOp 0	F5	F4	F3	F2	F1	FO	Operation
Somma per lw/sw	0	0	X	X	X	X	X	0010	
Sub per breq	X	1	X	X	X	X	X	0110	
add	1	X	X	X	0	0	0	0010	
sub	1	X	X	X	0	0	1	0	0110
and	1	X	X	X	0	1	0	0	0000
or	1	X	X	X	0	1	0	1	0001
slt	1	X	X	X	1	0	1	0	0111

Andiamo a considerare i 3 bit d'operation singolarmente, considerando soltanto gli 1 e compilando le tabelle

**Operation 2** vale 1 quando compare  $X1$  ad  $ALUOp_1$  e quando compare  $1X$  ad  $ALUOp_1$  e al campo  $F1$  vale 1

**Operation 1** vale 1 quando  $ALUOp_1 = 0$  e quando abbiamo  $ALUOp_1 = 1X$  e al campo  $F2$  vale 0

**Operation 0** vale 1 quando abbiamo  $ALUOp_1 = 1X$  e non ha una tra  $F0 = 1$  oppure  $F3 = 1$



$$Op_2 = ALUOp_0 \cdot O + ALUOp_1 \cdot F1$$

$$Op_1 = \overline{ALUOp_1} + \overline{F2}$$

$$Op_0 = ALUOp_1 \cdot (F0 + F3)$$

Costruiamo ora l'Unità di Controllo Principale, che prende in input il codice operativo OpCode (6 bit) dell'istruzione da eseguire e restituisce in output sia l'input per ALUOp che gli altri segnali di controllo:

**RegDst** O: Il num. del reg. di scrittura proviene dal campo rt (20-16)  
I: // // // dal campo rd (15-11)

**RegWrite** O: Nulla I: Il dato viene scritto nel reg. individuato dal num. del reg. di scrittura

**ALUSrc** O: Il 2° operando della ALU proviene da Read Data 2.  
I: // // proviene dall'estensione di segno

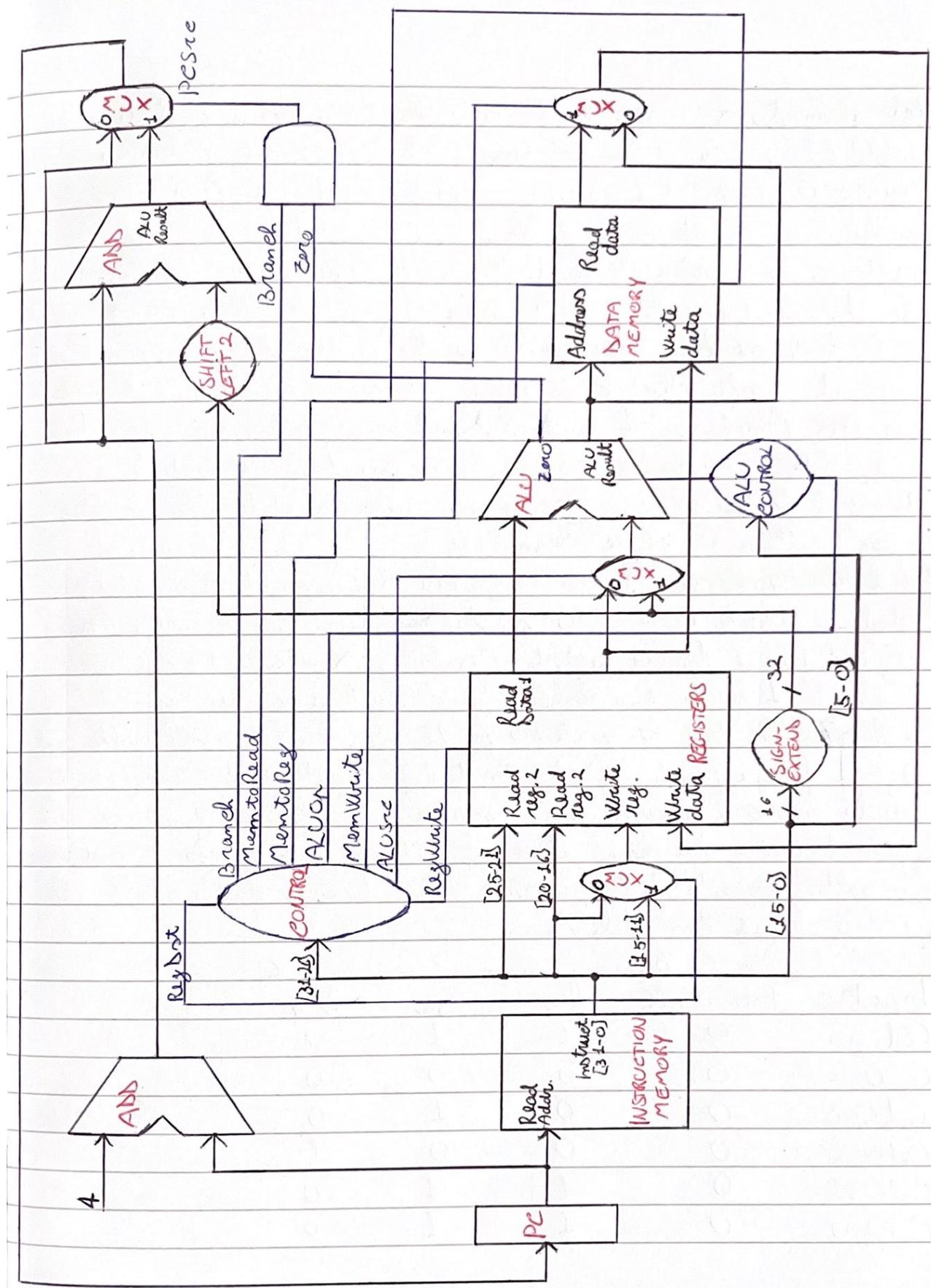
**MemRead** O: Nulla I: Il dato della memoria nella posizione puntata dall'indirizzo viene inviato in uscita sulla linea Read Data

**MemWrite** O: Nulla I: Il contenuto della memoria nella posiz. puntata dall'indir. viene sostituito con il dato di Write Data

**MemtoReg** O: Il dato inviato al register file per la scrittura viene dalla ALU  
I: // // // viene dalla Memoria Dati

**PCSrc** O: Nel PC viene scritta l'uscita del sommatore che calcola PC+4  
I: // // // calcola l'indirizzo di salto

Per generare usiamo una porta AND che prende l'uscita Zero della ALU e un nuovo segnale, Branch, vero solo con Breg



Nel formato R, i segnali di controllo sono così impostati:

RegDst = 1 perché il reg. destinazione è nel campo rd e dunque AluSrc = 0 perché il 2° operando viene dal campo rt. MemtoReg = 0 perché il risultato viene dalla ALU e RegWrite = 1 perché il ris. va scritto in un registro. MemRead = MemWrite = 0 perché non c'è accesso alla mem. dati e infine Branch = 0 perché la prossima ist. è PC + 4

Per l'istruzione **lw**: RegDst = 0 perché il reg. destinazione è nel campo rt, mentre AluSrc = 1 perché il 2° operando viene dall'estens. a 32 bit. MemtoReg = 1 perché la word viene dalla memoria dati e RegWrite = 1 perché va scritta in un registro. MemRead = 1 e MemWrite = 0 perché c'è accesso alla memoria dati solo in lettura e non scrittura e infine Branch = 0

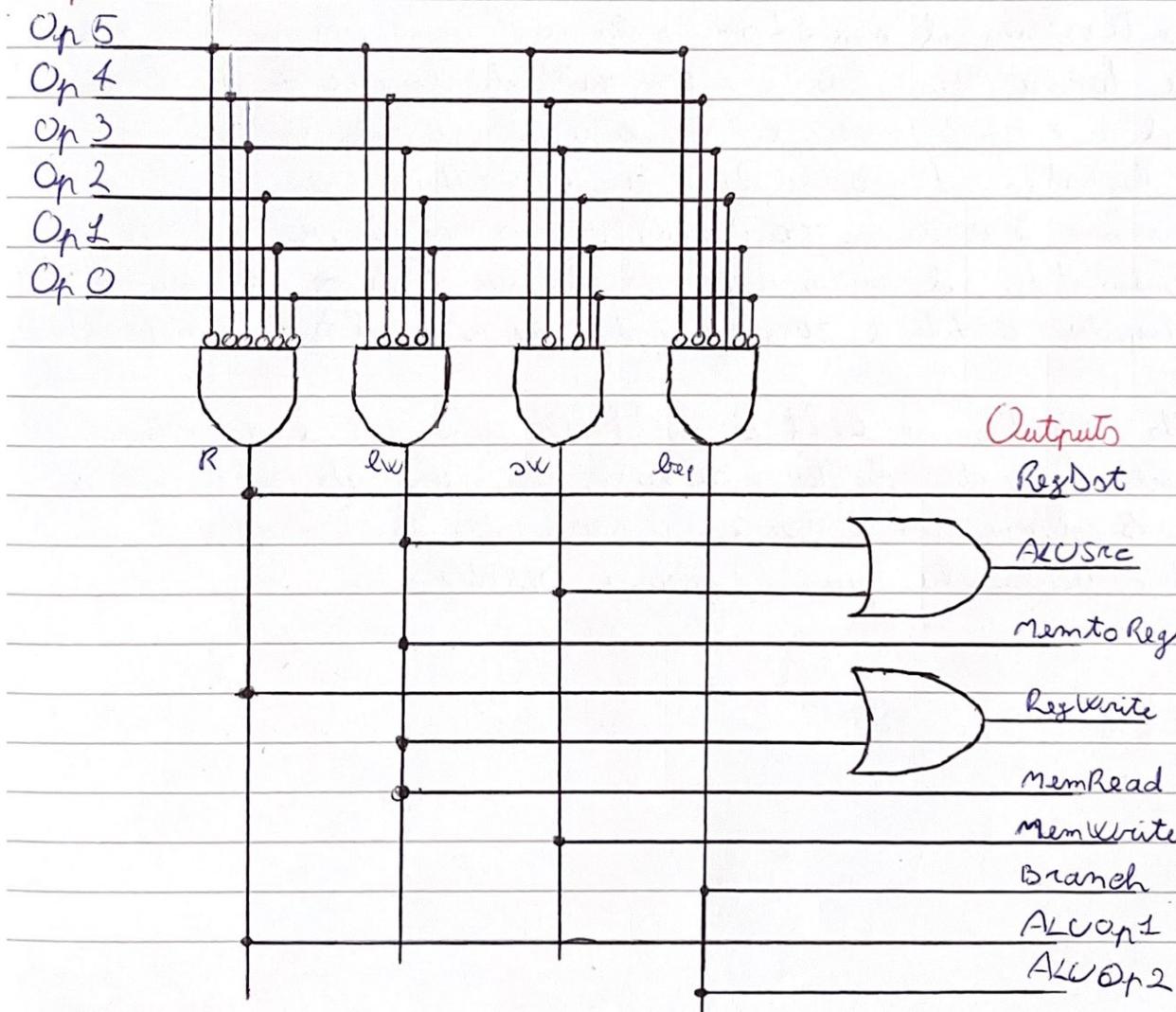
Per l'istruzione **sw**: AluSrc = 1 perché il 2° operando viene dalla estens. a 32 bit, RegWrite = 0 perché non viene scritto nulla nel register file e dunque RegDst = MemtoReg = X. MemRead = 0 e MemWrite = 1 perché l'accesso è solo in scrittura e Branch = 0

Per l'istruzione **lreq**: RegWrite = 0 perché non viene scritto nulla nel register file e quindi RegDst = MemtoReg = X. MemRead = MemWrite = 0 perché non c'è accesso alla memoria dati. AluSrc = 0 perché il 2° operando viene dal campo rt e infine Branch = 1 perché l'indirizzo di salto dipende della prossima ist. dipende dall'ind. di salto e dall'output Zer dell'ALU.

Input	Formato R	$0_{20}$	$35_{20}$	$42_{20}$	$4_{20}$
CodOp5	0	1	1	0	
CodOp4	0	0	0	0	
CodOp3	0	0	1	0	
CodOp2	0	0	0	1	
CodOp1	0	1	1	0	
CodOp0	0	1	1	0	

Output	Format	R	lw	sw	lreq
RegDst	1	0	X	X	
ALUSrc	0	1	L	0	
MemtoReg	0	1	X	X	
RegWwrite	1	1	0	0	
MemRead	0	1	0	0	
MemWrite	0	0	1	0	
Branch	0	0	0	1	
ALUOp1	1	0	0	0	
ALUOp2	0	0	0	1	

### Inputs



Aggiungiamo ora nuove istruzioni al set considerato:

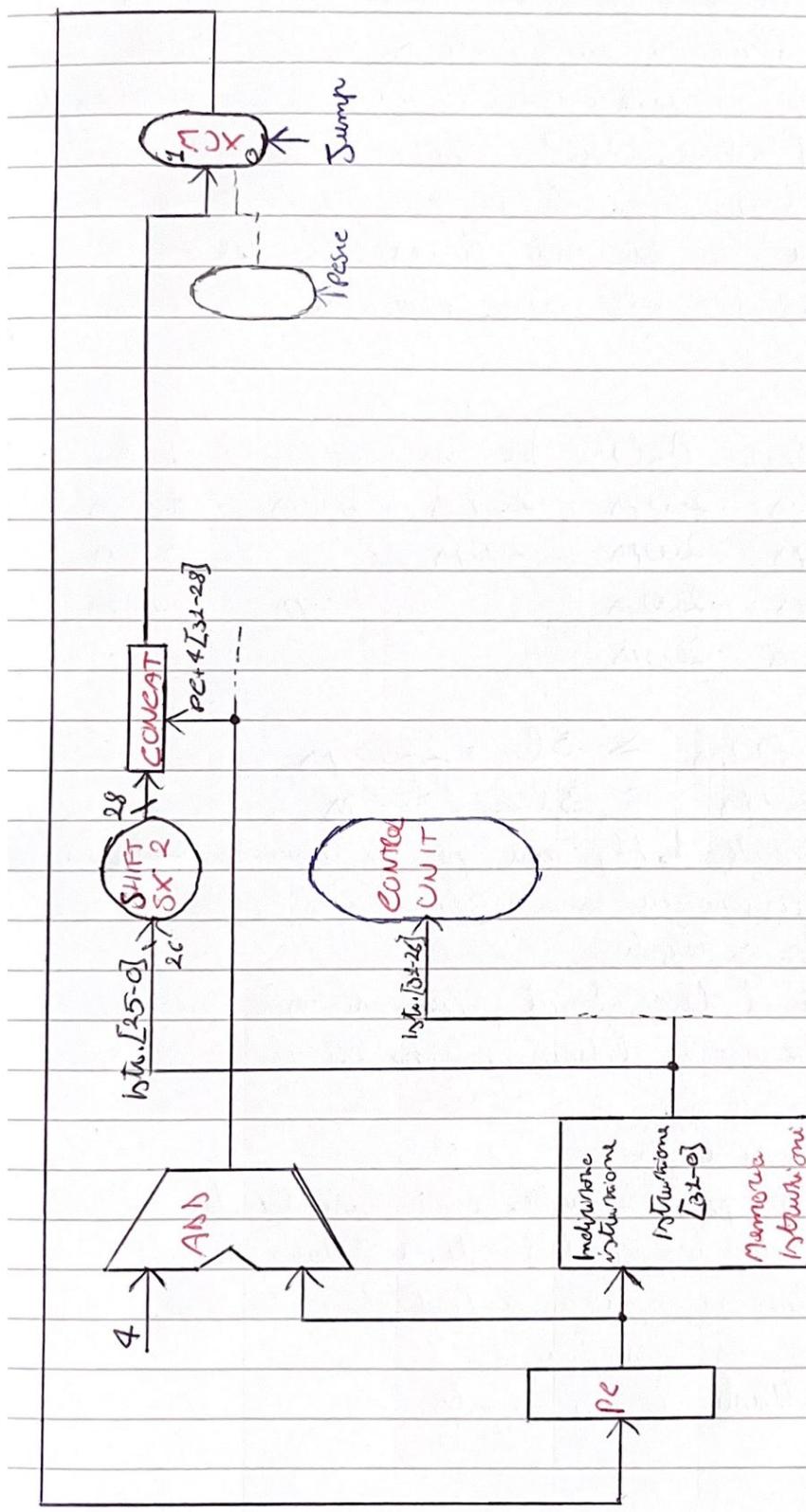
~~addi~~ (tipo I) ha operando immediato e come per le due addizioni una somma tra reg. e costante, ma qui non si parla di indirizzi, dunque MemRead  $\Rightarrow 0$  e MemToReg  $\Rightarrow 0$  e inoltre non servono molti componenti.

le ~~2~~ istruzioni di tipo J (che richiederanno hardware):

J (salto incond.) richiede uno shifter a 32 posti capace di prendere una stringa da 26 bit e restituire una da 28 ed un mux 2:1 con segnale di controllo jump da poche prime del PC  
Jal (Jump and link) segue lo stesso iter della precedente, ma serve l'indirizzo di ritorno nel registro \$ra (32). Per farci ciò trasforma il mux 2:1 in uscita alla memoria in un mux 4:1 e dunque MemToReg = 2 indica che WriteData è PC + 4 ed anche il mux 2:1 in uscita al banco dei registri diventa un 4:1, con RegDest = 2 ad indicare che il reg. di scrittura è \$ra

ed infine l'istruzione SJ (tipo R) che semplicemente richiede una linea dati nel nuovo mux inserito per l'istr. J, che quindi diventa un 4:1, e Jump = 2 indica che parla il dato letto L dal Register

Nello specifico, un Left Logic Shifter a 2 posti è un caso particolare di shifter a 32 bit, che è una rete combinatoria di 5 colonne con  $2^5$  MUX 2:1 e nessuna che ha il proprio segnale di controllo, che qui è fisso a 00010.



L'implementazione a ciclo singolo è in realtà inutilizzata poiché tarando il clock all'istruzione più lenta si spreca molto tempo motivo per cui, ispirandosi alle catene di montaggio introdotte da Henry Ford, è nata la Pipeline. Questo approccio permette di eseguire più azioni contemporaneamente in quanto consente di "dividere il processore" in più stadi, 5 nel caso del MIPS, e tara il ciclo di clock non sull'intera istruzione ma sullo stadio più lento.

Type	Lett. istr.	Lett. reg.	ALU	Ace Mem	Ser. Reg.	Tot.
lw	200 ps	200 ps	200 ps	200 ps	100 ps	800 ps
sw	200 ps	100 ps	200 ps	200 ps		700 ps
R	200 ps	200 ps	200 ps		100 ps	600 ps
lreq	200 ps	100 ps	200 ps			500 ps

Ciclo singolo :  $clock = 800 \mu s \rightarrow 3lw = 2400 \mu s$

Pipeline :  $clock = 200 \mu s \rightarrow 3lw = 1200 \mu s$

Il rapporto tra le due valle 1,71, ma per un numero maggiore di istruzioni si può raggiungere un incremento di velocità pari al numero di stadi considerati

Questa implem. aumenta il throughput, num. di istr. eseguiti nell'unità di tempo, ma non la latenza, tempo per una singola esecuz.

I cinque stadi della pipeline sono:

IF (Instruction Fetch) prelievo istr. e incremento del PC

ID (Inst. Decode) decodifica dell'istr. e lettura registri

EX (Execution) esecuzione operazione / calcolo indirizzo

MEM (Memory) accesso in memoria

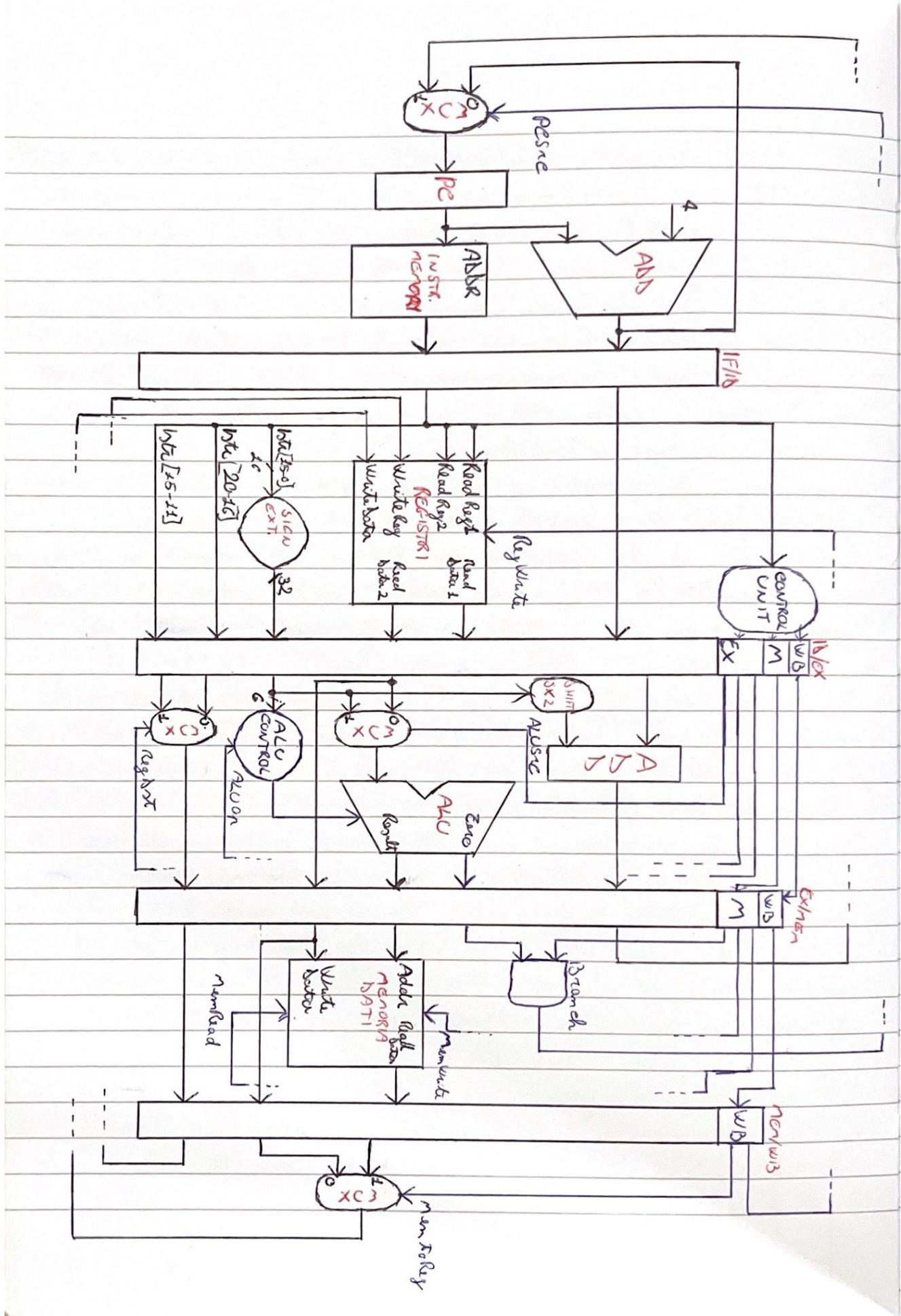
WB (Write Back) scrittura banco registro

Per "salvare" le informazioni correnti e condividerle con lo stadio successivo, inserivamo **4 registri di pipeline**, che si occupano di trasferire i dati e pertanto sono copiati a sufficienza. Analizzeremo quindi il comportamento della line:

- durante la fase di fetch, vengono copiati in IF/ID sia l'istruzione che ~~il PC4~~, che potrebbe tornare utile;
- al secondo stadio vengono copiati in ID/EX non solo l'offset a 32 bit, il dato letto 1 ma anche il dato letto 2, per facilità di implementazione
- l'ALU esegue la somma e scrive in EX/MEM
- il dato letto in memoria viene scritto in MEM/WB
- scrivere nel reg. file presenta un problema, il reg. di destinazione era contenuto in IF/ID, ma esso è stato ~~solo~~ sovrascritto:  
~~quindi~~ anch'esso è stato in realtà copiato tramite una linea diretta che collega i registri.

Per quanto riguarda **l'unità di controllo**, essa non velede mai segnali in quanto i registri vengono riscritti ad ogni ciclo di clock e quelli esistenti vanno solo raggruppati per stadio e quindi salvati nei registri di volta in volta:

- da IF/ID l'istruzione è inviata alla Control Unit
- ID/EX li memorizza tutti e 9 ed usa RegDst, ALUOp e ALUSrc per la fase EXE
- EX/MEM ne salva 5 ed usa Branch, MemRead e MemWrite per la fase MEM
- MEM/WB infine salva ed usa RegWrite e MemToReg



L'implementazione pipeline presenta però una molteplicità di problemi diversi derivati proprio al suo comportamento: gli

**hazard o conflitti**, che possono essere di diverso tipo:

- **Strutturali**: stadi diversi necessitano delle stesse risorse hardware nello stesso ciclo di clock

- **Su dati**: un'istruzione dipende dal risultato di una precedente non ancora completa

- **Sul controllo**: prendere decisioni su condizioni non ancora valutate. Quelli strutturali sono stati già risolti con la progettazione stessa del processore tramite la separazione di memoria istruzioni e memoria dati e facendo sì che la scrittura dei reg. file avvenisse nella 1<sup>a</sup> metà del clock e la lettura nella 2<sup>a</sup>.

Esempi di hazard su dati sono:

add \$2, \$1, \$3

IF | ID | EX | MEM | WB

sub \$1, \$2, \$5

IF | ID | EX | MEM | WB

lw \$2, 20(\$1)

IF | ID | EX | MEM | WB

sub \$4, \$2, \$5

IF | ID | EX | MEM | WB

risolubili con modifiche hardware:

- Forwarding, cioè propagazione

- Inserimento di bolle o stall.

oppure di tipo software:

- Inserimento di istruzioni nop

- Riordino delle istruzioni.

Esistono 4 tipi diversi che generano hazard su dati:

1) EX/MEM. Registro Rd = ID/EX, Registro R<sub>j</sub>

2) = = = ID/EX. Registro R<sub>k</sub>

Riguardano istruzioni che si apprestano a passare dallo stadio ID allo stadio EX ma uno dei due operandi dipende dal risultato di una ist. prec. che viene prodotto in fase EX.

$$3) \text{MEM/WB.Registro Rd} = \text{ID/EX.Registro Rs}$$

$$4) \quad // \quad = \text{ID/EX.Registro Rt}$$

Quando uno dei due operandi di una ist. che deve entrare nello stadio EX dipende dal risultato di una ist. precedente che viene prodotto nello stadio MEM.

N.B.: Se RegWrite = 0 non c'è necessità di propagare

Per consentire agli input dell'ALU di provengere da uno qualsiasi dei registri di pipeline, vengono aggiunti 2 MUX 4:1 i cui segnali sono generati da una **Unità di Propagazione**:

Propaga A (B) = 00 primo (secondo) operando da reg. file;

// = 10 1° (2°) operando prep. dalla ALU nel caso prec.

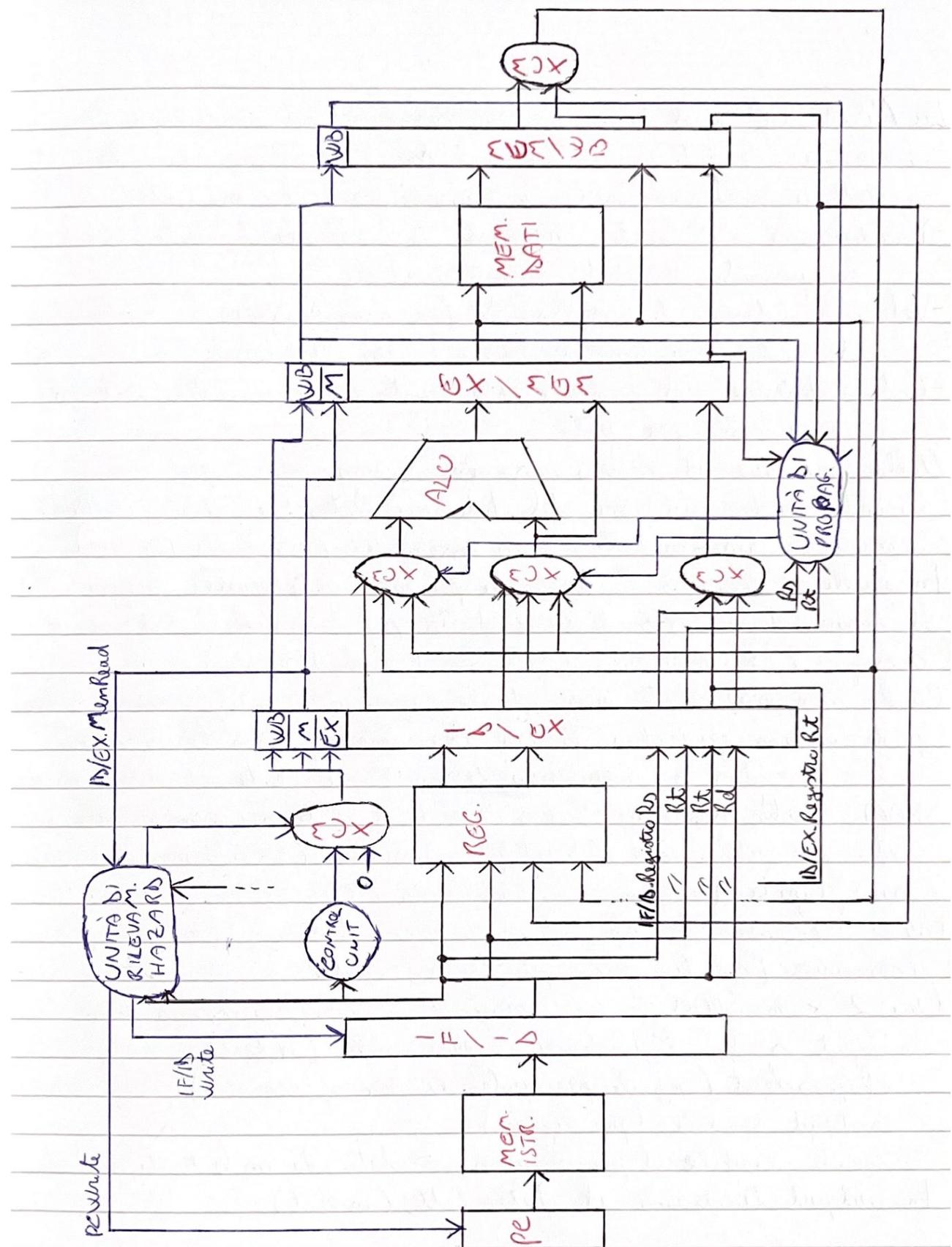
// = 01 1° (2°) // // dalla memoria dati

Per quanto riguarda il secondo esempio, non basta la sola propagazione: entra in gioco la **Hazard Detection Unit**:

- Rileva l'errore controllando se ID/EX.MemRead sia assento e in caso affermativo se si ha ID/EX.Registro Rt = IF/ID.Registro Rs (Rt)
- In caso di verifica, viene inserita una bollo: si impostano a 0 i segnali di controllo e si fa in modo che non vengano prelevate nuove istruzioni né venga incrementato il PC

Generalmente si hanno hazard sul controllo nel caso dei salti e per risolverli si può ricorrere a diversi metodi:

- Inserire bolle fai quando non viene calcolato il risultato
- Anticipare il confronto, che si chiede sia di calcolare in anticipo l'ind. di salto spostando il sommatore allo stadio ID e avendo già i dati necessari e ~~per~~<sup>ma</sup> valutare il confronto, per cui si richiede hardware aggiuntivo e si rischiano hazard sui dati
- Predizione statica, immaginando che il salto non avvenga e portando avanti normalmente le istruzioni, che eventualmente possono essere cancellate cancellando i segnali di controllo



La **MEMORIA** è il luogo in cui sono contenuti programmi in esecuzione e i dati di cui necessitiamo e possiamo vederla come un vettore unidimensionale in cui ogni cella ha un indirizzo.

- **Dimensione o capacità** misurata in multipli di byte, indica la quantità di dati memorizzabili

- **Velocità o tempo di accesso**  $t_c$  (in nanosec.) tra la richiesta del dato e il momento in cui è reso disponibile

- **Costo** dipende dal volume d'acquisto: più aumenta, minore è il costo per bit

Nelle **memorie ad accesso casuale**, il tempo di accesso non dipende dalla posizione del dato richiesto: è tipico delle memorie a semiconduttori (es. memoria principale (RAM))

In quelle **ad accesso sequenziale**, tipicamente quelle magnetiche, il tempo di accesso dipende dalla posizione del dato e un esempio è la memoria secondaria (dischi e nastri)

Poiché non ne esiste una ideale, vengono usate diverse memorie poste in una **gerarchia** in cui troviamo in cima le piccole, veloci e costose e scendendo le più ampie, lente ed economiche.

**SRAM**: Static Random Access Memory: statica (memorizza il dato fin quando c'è alimentazione), costosa (6-8 transistor per bit), rapida (e quindi vicina alla CPU).

**DRAM**: Dynamic Random Access Memory: dinamica (refresh periodica) economica (un transistor per bit), lenta

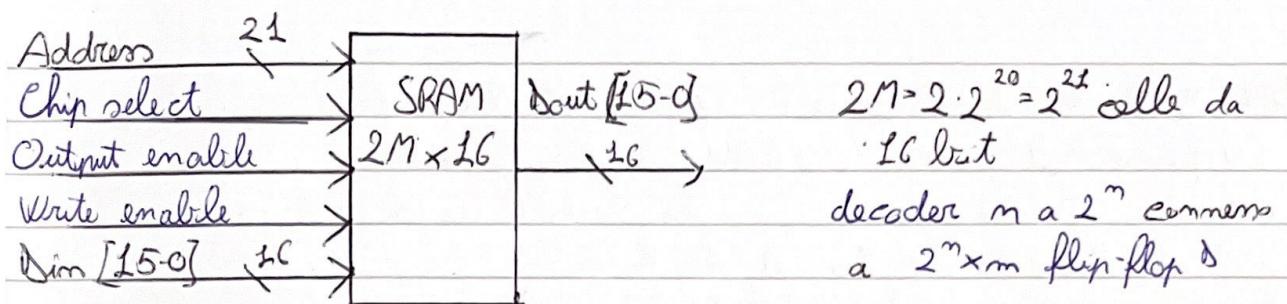
Una  $2^n \times m$  SRAM è un array di  $2^n$  celle, ciascuna di  $m$  bit:  
In input si ha l'indirizzo della cella ( $n$  bit)

chip select (segno per poter leggere o scrivere)

output enable (per leggere)

write enable (per scrivere)  $\Rightarrow$  dato da  $m$  bit da scrivere

In output riceviamo il dato letto ( $m$  bit)



La cache è una SRAM posta tra Registro e Memoria centrale, pertanto contiene un sottoinsieme dei dati del livello sottostante e tutti quelli contenuti nel livello superiore. I dati usati più spesso sono posti più in alto, mentre quelli più rari sono posti in basso (principio di località). I dati vengono allocati dinamicamente, cioè spostati automaticamente e rapidamente tra i livelli, rispettando sia la località temporale (tendenza di un dato ad essere riletto) che quella spaziale (tendenza a leggere dati adiacenti).

Blocco o linea più piccola quantità di dati trasferibile.

Hit: dato richiesto è presente nel livello adiacente, Miss altrimenti.

Il tempo di hit è il tempo nel trovare il dato richiesto nel livello superiore, la penalità di miss invece è il tempo impiegato nel trovare un dato in caso di miss. L'hit rate, frequenza di successo, è usato come indice delle prestazioni della gerarchia.

Analisi: Consideriamo un processore che carica una parola per volta e blocchi anch'essi ~~che una~~ parole e studiamone il comportamento:

Il processore richiede la parola  $X_m$ , producendo un miss in quanto sono presenti le parole  $X_1, \dots, X_{m-1}$ . È possibile verificarlo poiché esiste una corrispondenza univoca:

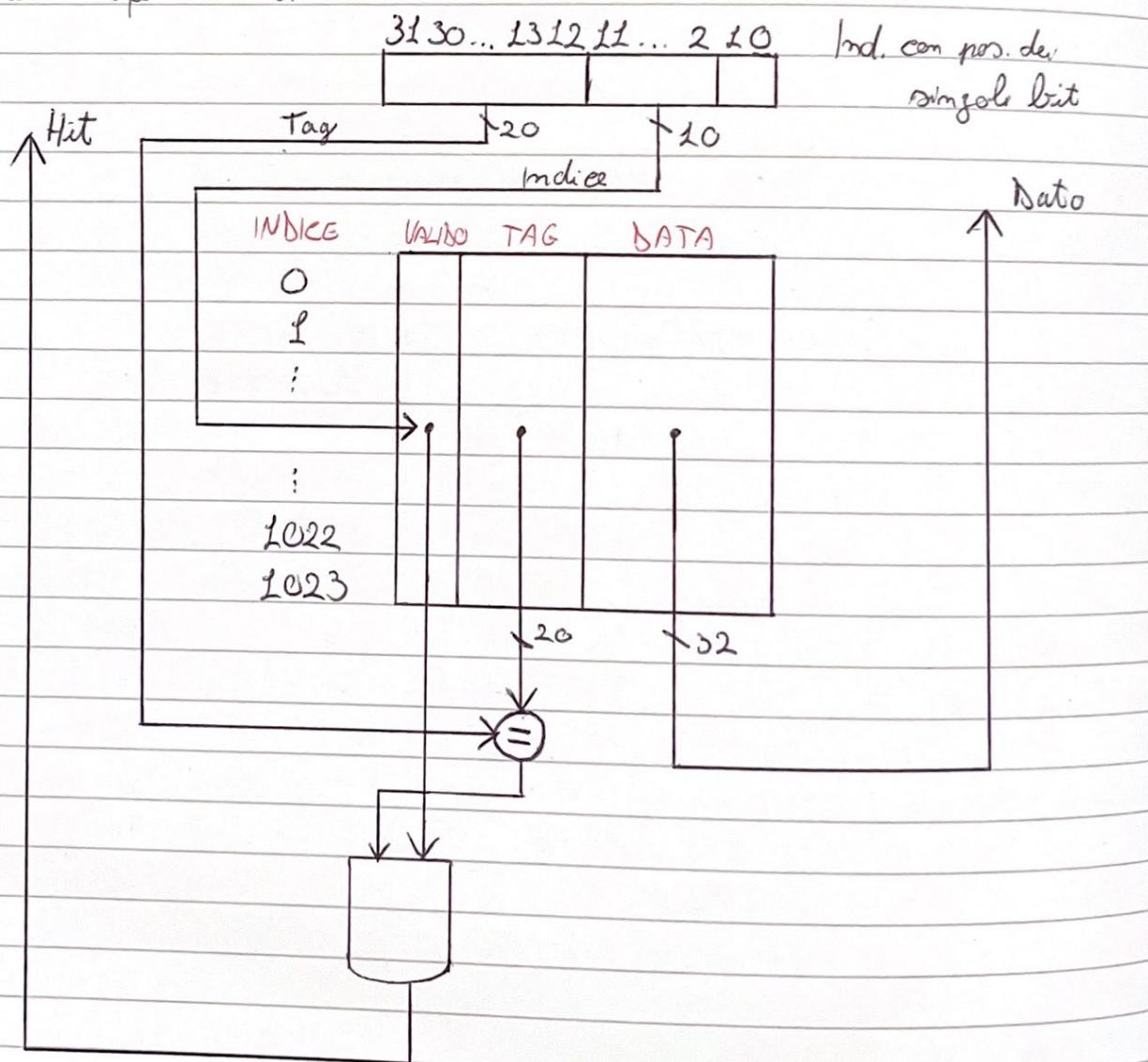
$\text{ind cache} = \text{ind mem modulo } N_B$  con  $N_B >$  numero di blocchi avendo  $N_B$  come potenza di due, basta considerare i  $\log_2 N_B$  bit meno significativi e mapparli nello stesso blocco di cache: i blocchi di memoria che terminano con gli stessi bit meno sig.

I bit restanti formano il tag che serve a distinguere i blocchi. E c'è poi il campo validità che indica se il dato può essere letto

Nel MIPS troviamo invece i primi 20 bit a formare il tag, 10 per l'indice e gli ultimi due per l'offset.

La cache è formata da  $2^{10}$  blocchi formati da 32 bit ciascuno, per una dimensione totale di 4KB.

- Accedendo alla cache si confronta il tag dell'indirizzo con il tag della linea di cache individuata tramite l'indice, poi si controlla il bit di validità e infine viene segnalato l'hit e trasferito il dato.



ES.  $a = b + (c/2)$  con  $a$  in  $\$r_0$ ,  $b$  in  $\$r_1$  e  $c$  in  $\$r_2$   
 ldr  $r_0, r_2, L$   $\ast$  shift  $a$  dx di un posto  $c/2$  in  $r_0$   
 add  $r_0, r_1, r_0$

ES.  $V[4] = 8^*a - (b + V[3])$   $a$  in  $\$r_0$ ,  $b$  in  $\$r_1$ , ind base di  $V$  in  $\$r_2$   
 lw  $t_0, 4(\$r_2)$   $\ast$  carica in  $t_0$   $\$r_2 + 4$   
 add  $t_0, r_1, t_0$   $\ast$   $t_0 = b + V[3]$   
 sll  $r_0, r_0, 3$   $\ast$   $r_0 = 8^*a$   
 sub  $r_0, r_0, t_0$   
 sw  $r_0, 4(\$r_2)$   $\ast$  registra la word in  $\$r_2 + 4$

ES.  $W[i] = V[i+5]$   $i$  in  $\$r_0$ ,  $j$  in  $\$r_1$ , ind  $b$   $V$  in  $\$r_2$ , ind  $b$   $W$  in  $\$r_3$   
 add  $t_0, r_0, r_2$   
 sll  $t_2, t_0, 2$   $\ast$   $t_2 = 4^*(i+5)$   
 add  $t_2, r_2, t_2$   $\ast$   $t_2 = V + 4^*(i+5)$   
 lw  $t_3, 0(t_2)$   $\ast$  carica in  $t_3$   $V[i+5]$   
 sll  $t_4, r_0, 2$   $\ast$   $t_4 = 4^*i$   
 add  $t_5, r_3, t_4$   $\ast$   $t_5 = W + 4^*i$   
 sw  $t_3, 0(t_5)$   $\ast$  registra la word  $t_3$  in  $W[i]$

ES.  $lw r_0, 4(r_2)$  cond. iniz.:  $r_0$  ha 5,  $r_2$  1024  
 $lw r_0, 4(r_2)$  all'ind. 1028 c'è 10

prima salva la word di  $r_0$  in  $\$r_2 + 4$  ( $1028$  c'è 5)  
 poi carica in  $r_0$  il contenuto di  $\$r_2 + 4$  (da  $r_0$  c'è 5)

ES.  $\text{SLL } t_0, s_0, 2$  cond. init. :  $i \text{ in } s_0, j \text{ in } s_1$   
 $\text{add } t_1, s_2, t_0$  ind  $b$   $V$  in  $s_2$ , ind  $b$   $W$  in  $s_3$   
 $\text{lw } t_2, 0(t_1)$   $\#$  carica in  $t_2$   $V + (i * 4)$   
 $\text{addi } t_3, t_2, -8$   $\# t_3 = W[i-8] V + (i * 4) - j$   
 $\text{lw } t_4, 0(t_3)$   $\#$  carica in  $t_4$   $V[i-2]$   
 $\text{add } t_5, t_2, t_4$   $\# t_5 = V[i] + V[i-2]$   
 $\text{SLL } t_6, s_1, 2$   $\# t_6 = j * 4$   
 $\text{add } t_7, s_3, t_6$   
 $\text{addi } t_8, t_7, 4$   $\# t_8 = W + (j * 4) + 4$   
 $\text{sw } t_5, 0(t_8)$   $\#$  registra in  $t_8$  la word contenuta in  $t_5$

$$W[j+4] = V[i] + V[i-2]$$

ES.  $\text{if } (a != b) d = c + 2; \text{ else } d = c * 4; \quad a \text{ } s_0, b \text{ } s_1, c \text{ } s_2, d \text{ } s_3$   
 $\text{Breq } s_0, s_2, \text{ ELSE } \# \text{ salta a ELSE se } a = b$   
 $\text{addi } s_3, s_2, 2$   
 $j \text{ ESEI}$   
 ELSE:  $\text{SLL } s_3, s_2, 2$   
 ESEI: ...

ES.  $\text{if } (a <= b) d = c * 3; \text{ else } d = c * 4, \quad a \text{ } s_0, b \text{ } s_1, c \text{ } s_2, d \text{ } s_3$   
 $\text{SLL } s_3, s_2, 2 \quad \# d = c * 4 \text{ (se ne in ogni caso)}$   
 $\text{slt } t_0, s_1, s_0 \quad \# t_0 = 1 \text{ se } b < a \text{ (a} > b)$   
 $\text{bne } t_0, \text{zero}, \text{ESEI} \quad \# \text{ salta a ESEI se } t_0 = 1 \text{ (se a} > b)$   
 $\text{sub } s_2, s_3, s_2 \quad \# d = d - c = c * 3$   
 ESEI: ...  $\#$  fine decisione

ES  $i=1; s = V[7];$  while ( $i \neq 5$ )  $x = 7^i \cdot i;$   $i \geq 15 \Rightarrow V[s_2]$   
 addi  $s_0, zero, 1 \quad \# i=1$   
 lw  $s_1, 28(s_2) \quad \# carica in s_1 V[7]$   
 CICLO: breq  $s_0, s_1, ESCI \quad \# salta a ESCI se  $i=5$$   
 sll  $t_0, s_0, 3 \quad \# t_0 = i * 8$   
 sub  $s_0, t_0, s_0 \quad \# s_0 = t_0 - s_0 \Rightarrow i = 8^k i - i = 7^k i$   
 J CICLO

ESCI:

ES. for ( $i=0; i < 50, i++\{$   $i$  in  $s_0$ , somma in  $s_2$   
 somma +=  $V[i];$  and base  $V$  in  $s_2$   
 $\}$

addi  $s_0, zero, 0 \quad \# i=0$

CICLO: lw  $t_0, 0(s_2) \quad \# carica in t_0 il contenuto di V[i]$   
 add  $s_2, s_2, t_0 \quad \# somma = somma + V[i]$   
 addi  $s_2, s_2, 4 \quad \# and V+4 (list. succ.)$   
 addi  $s_0, s_0, 1 \quad \# i = i+1$   
 slti  $t_1, s_0, 50 \quad \# t_1 = 1 se s_0 < 50 (i < 50)$   
 bne  $t_1, zero, CICLO \quad \# se t_1 \neq 0 salta a CICLO$

ES Se in un latch S-R il segnale Set è 1, allora:

B) forma  $Y$  ad 1

Se in un Flip-Flop S-R si ha  $R=1$ , allora:

C) forma  $\bar{Y}=1$  se si ha  $Clock=1$

Se in un Flip-Flop D si ha  $Clock=0$ , allora:

A) Flip-Flop chiuso e uscite invariate

Se in un Flip-Flop D il segnale di uscita  $Y$  è 1, allora:  $D=x$

B) Nessuna delle precedenti (se si ha  $Clock=1-D$  o  $Clock=0$ )

$$f_{\text{mem}} = 20_{\text{esa}} = 2 \cdot 16 = 32_{10}$$

ES. add  $s_0, s_1, s_2$   $s_0$  contiene 138,  $s_1$  ha -64,  $s_2$  ha 39

stringa istruzione: 000000|10001|10010|10000|00000|~~100000~~  
~~010100~~

l'ALU ottiene sulle uscite a 32 bit i contenuti di  $s_1$  ed  $s_2$

$$s_1 = -64_{10} = 1000000_{10} \rightarrow 1111111111111111111111000000$$

$$s_2 = 39_{10} = \cancel{00000000000000000000000000000000}010011_{10} \rightarrow 00000000000000000000000000000000010011$$

dato da scrivere nella banca dei registri:  $-25_{10} = 100111_{10}$   
 $1111111111111111111111000111$

e numero del reg.: 26

MemWrite = 0 nessun accesso in memoria

ES. se  $s_2, 24(s_2)$  qual è il 1° ingresso della ALU?

c) Il contenuto di  $s_2$

il pipelining

B) Aumenta il num. di istru. eseguibili nell'unità di tempo  
 se in un Flip-Flop ho Clock  $K=1$

A) Flip-Flop aperto e output Q = 8

in una memoria cache si ha hit quando:

D) (A?)