

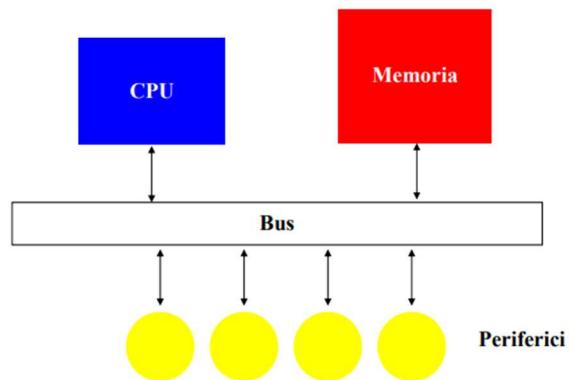
CALCOLATORI ELETTRONICI (ESAME)

0) Introduzione

Un **SISTEMA DI ELABORAZIONE** (es. Computer [ovvero Calcolatore]) riceve, elabora ed invia informazioni; sono:

- **GENERAL PURPOSE** = progettati e venduti senza una precisa applicazione (es. computer);
- **SPECIAL PURPOSE** = software già caricato nel sistema in quanto deve svolgere un preciso compito (es. lettore di badge) [ovvero i **SISTEMI EMBEDDED** che trovano un esempio nei **MICROCONTROLLORI**]. Una volta interconnessi creano l'**IOT** (Internet Of Things).

La struttura di un sistema di elaborazione è **APPLICAZIONI** → **OS** (Sistema Operativo) → **FIRMWARE** → **HARDWARE**, dove l'HARDWARE è riconducibile all'**ARCHITETTURA DI VON NEUMANN**:



⚠ Se la memoria non è singola, ma doppia (memoria dati e memoria codice [istruzioni]), è l'**ARCHITETTURA HARVARD**!

→ **LEGGE DI MOORE** = n° di transistor in un circuito raddoppia ogni 18/24 mesi e, con essi, il n° di bit immagazzinabili e la velocità di calcolo.

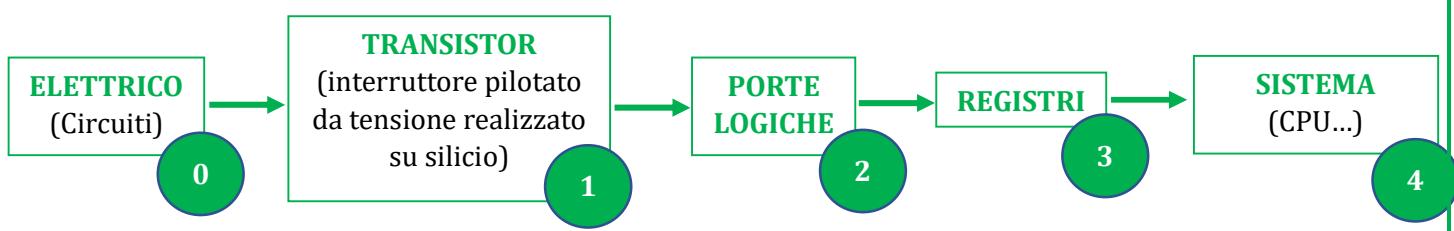
Altre definizioni utili sono:

- **ISA (Instruction Set Architecture)** = set di istruzioni che rimane costante nei processori successivi per mantenere la compatibilità delle applicazioni eseguibili, senza doverle riscrivere;
- **GPU (Graphics Processing Unit)** = composta da molti PE (Processing Element) che eseguono parallelamente la stessa istruzione sui propri dati (in questo modo sono più veloci sulle strutture dati regolari, come le matrici);
- **MCU (Microcontrollori)** = sono dei **SoC (System on Chip)** destinati ai sistemi special purpose (se il circuito integrato è fatto per una specifica applicazione, parliamo di ASIC).

1) Progetto 1 – CIRCUITI LOGICI e SEQUENZIALI

Un **sistema** è una collezione di componenti, il cui scopo è dato dalle funzioni svolte dai componenti singoli e da come questi sono interconnessi (**topologia**); un **circuito** è un sistema. Con "**PROGETTO**" si intende trovare la giusta correlazione tra le varie componenti in modo tale da ottenere il **comportamento cercato** e soddisfare le specifiche richieste (anche il **costo**, da contare come costo di progetto + di produzione + di manutenzione).

Un PROGETTO DI SISTEMI ELETTRONICI avviene in più livelli:

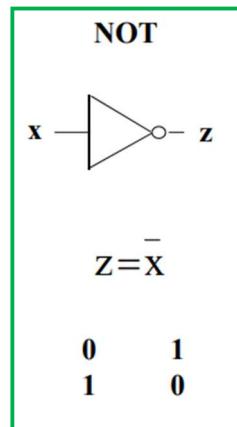
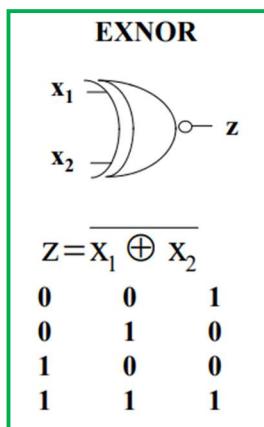
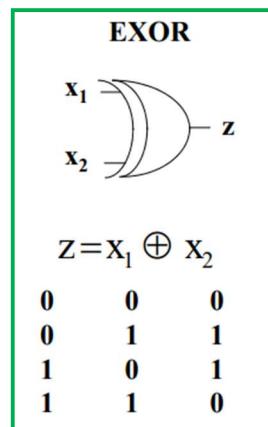
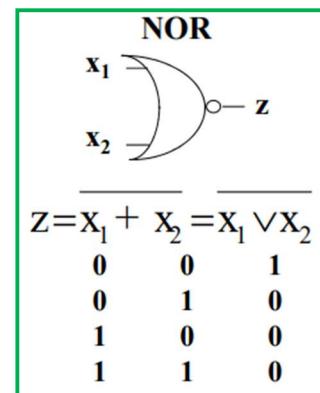
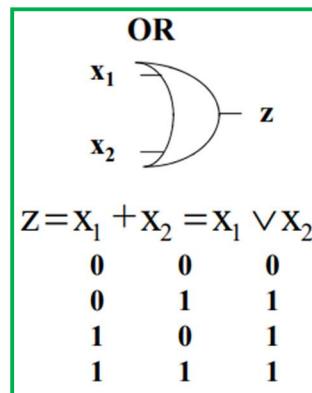
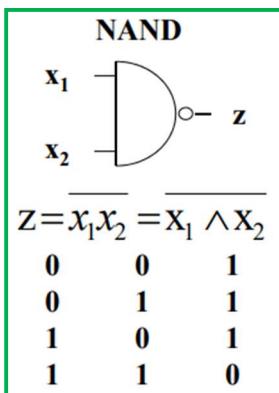
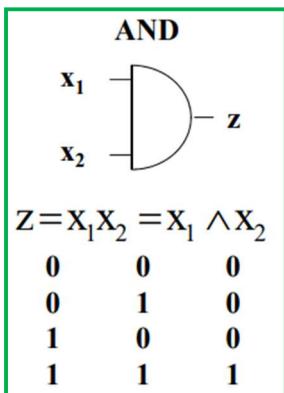


1

Il progetto a livello di **TRANSISTOR** si basa sul fatto che ogni punto del circuito a regime si trova alla **TENSIONE DI MASSA** (bit = 0) o alla **TENSIONE DI ALIMENTAZIONE** (bit = 1), impostando delle “soglie” per definire chi è a massa e chi ad alimentazione (i valori intermedi tra le due soglie sono “proibiti”).

2

Tramite l'**INVERTER NMOS** (ovvero un transistor in cui data tensione di alimentazione si ottiene tensione di massa e viceversa), riusciamo ad implementare le **PORTE LOGICHE** (**NAND** e **NOR** = 2 transistor; **NOT** = 1 transistor):



⚠ Se con un insieme di porte si possono realizzare tutti i circuiti combinatori, l’insieme si dice “completo”:

- {NAND}
- {NOR}
- {AND, NOT}
- {OR, NOT}
- {AND, OR, NOT}

⚠ Definiamo **FANIN** = n° segnali in input di una porta, e **FANOUT** = n° porte pilotate dall’output della porta!

→ Esistono 2 tipi di circuiti:

- A) **COMBINATORI** = output dipende solo da input corrente (es. sommatore);
- B) **SEQUENZIALI** = output dipende da input corrente e precedente (es. contatore).

A) I **COMBINATORI** sono descrivibili con **TAVOLE DI VERITÀ** (quelle viste sopra nelle porte logiche) e/o **FUNZIONI BOOLEANE** (ottenute dalle **MAPPE DI KARNAUGH**, ovvero metodo con cui passare dalla tavola di verità al circuito minino).

Per costruire un circuito da una tavola di verità, potrei adottare la soluzione “brutale” (1 porta AND per ogni 1 in output nella tavola + 1 porta OR in fondo per raccogliere i segnali), ma il nostro obiettivo è ottenere dei **CIRCUITI BEN FORMATI MINIMI** (CCBF MINIMI), ovvero che non abbiano dei cicli al loro interno e con il numero minimo di porte logiche (e quindi meno transistor) [l’obiettivo sarebbe avere 2 livelli di profondità del circuito]. Questo lo si fa **RIDUCENDO LE FUNZIONI BOOLEANE**:

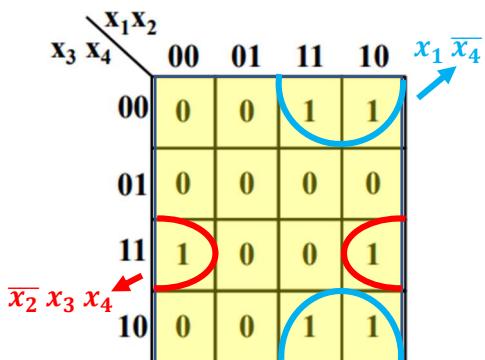
$$f = \overline{x_1} \overline{x_2} \overline{x_3} + \overline{x_1} \overline{x_2} x_3 + \overline{x_1} x_2 x_3 + x_1 x_2 x_3 = \overline{x_1} \overline{x_2} (\overline{x_3} + x_3) + x_2 x_3 (\overline{x_1} + x_1) = \overline{x_1} \overline{x_2} + x_2 x_3$$

Oppure USANDO LE **MAPPE DI KARNAUGH** (dove questa semplificazione avviene implicitamente); vediamo le regole per costruire la mappa di karnaugh:

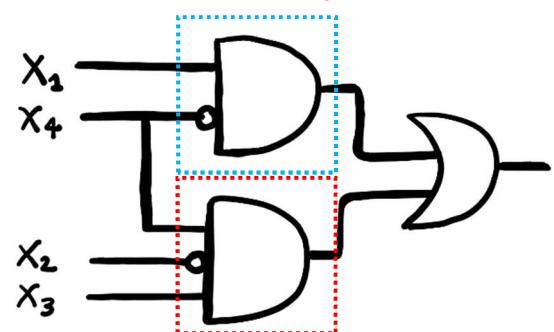
- Le colonne sono disposte in modo che cambi un solo bit per volta (00 – 01 – 11 – 10);
- Una volta costruita la mappa di karnaugh, effettuo la “copertura” (prendo gli 1 adiacenti creando il gruppo di 1 pari più grande possibile [facendo attenzione che anche l’ultima colonna e l’ultima riga sono adiacenti alla prima colonna e alla prima riga]).

ESERCIZI ESAME

1. Dalla mappa di karnaugh, realizzare il circuito minimo:



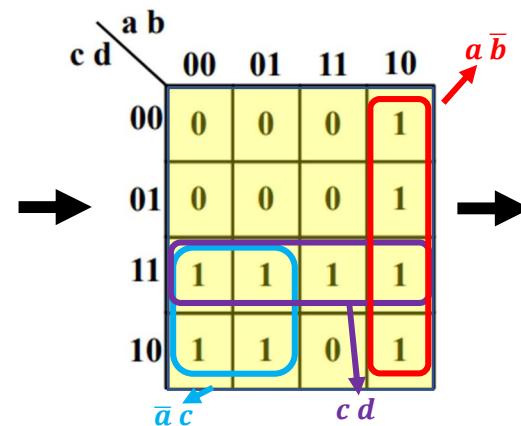
$$f = x_1 \bar{x}_4 + \bar{x}_2 x_3 x_4$$



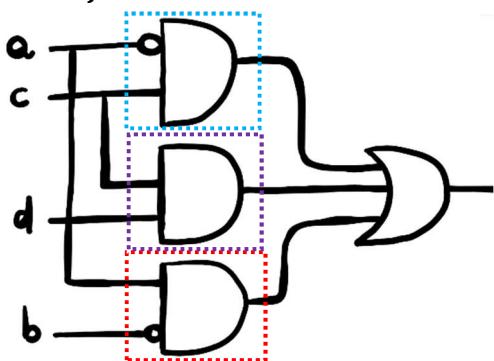
2. Dalla funzione booleana alla tavola di verità (da cui karnaugh e circuito minimo):

$$f = a \bar{b} + \bar{a} c + b c d$$

a	b	c	d	f
0	0	0	0	0
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

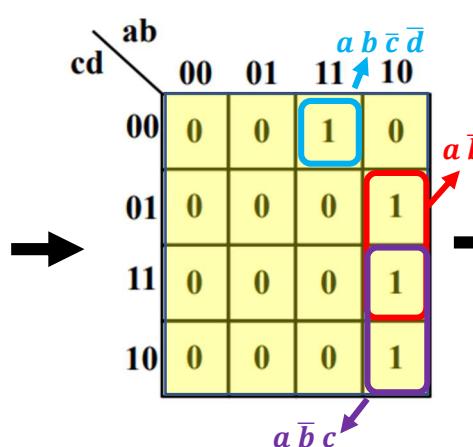


$$f = \bar{a} c + \bar{c} d + \bar{a} \bar{b}$$

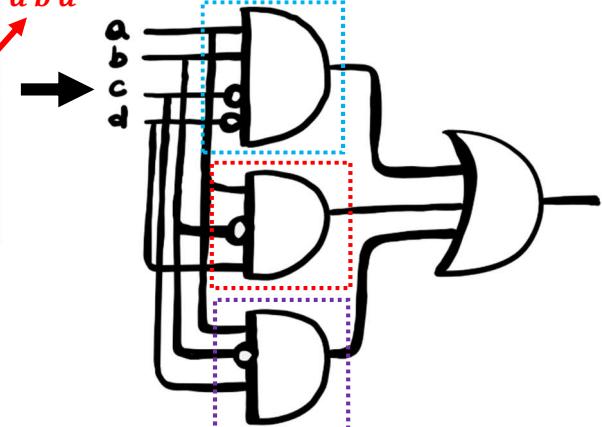


3. Dal linguaggio normale: 4 ingressi a, b, c, d con $f = 1$ se $8 < (abcd) < 13$:

a	b	c	d	f
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	0
1	1	1	1	0
1	1	1	1	0



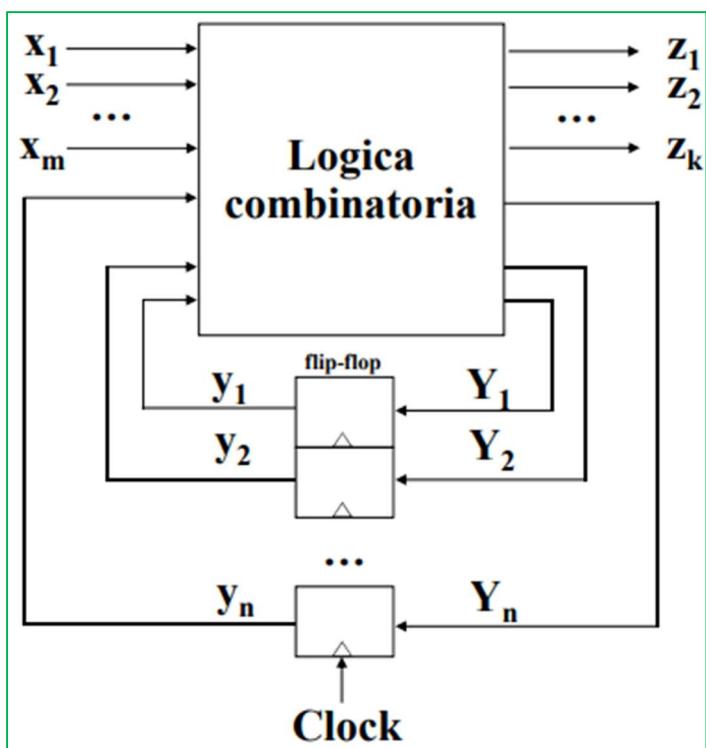
$$f = \bar{a} b \bar{c} \bar{d} + \bar{a} \bar{b} c \bar{d} + \bar{a} \bar{b} \bar{c} d$$



⚠ I risultati di combinazioni che non ci interessano/non possono avvenire, si esprimono come "DON'T CARE", ovvero possiamo metterli a 0 o 1 in modo tale da avere meno cubi e più grandi (circuito minimo più piccolo)!

⚠ Come già accennato, è importante che il circuito abbia profondità minima ($\cong 2$ livelli) perché, aumentando la profondità, aumenta anche il **RITARDO** (quindi ci va del tempo affinché l'output del circuito sia corretto); bisogna dunque fare in modo che il **CAMMINO CRITICO** (ovvero il cammino percorribile con più livelli) sia di 2 livelli [somma dei ritardi sul cammino critico = ritardo del circuito].

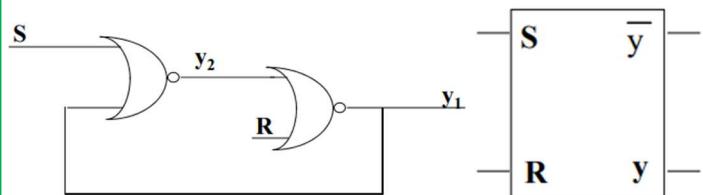
B) I **SEQUENZIALI** (o **FSM** = Finite State Machine) hanno l'output che dipende da **VALORI IN INPUT (X)** + **MEMORIA DEL CIRCUITO (Y)**; in genere lo "stato" transitorio del circuito non è osservabile dall'esterno. I circuiti sequenziali "sincroni" vengono implementati attraverso il MODELLO DI HUFFMAN:



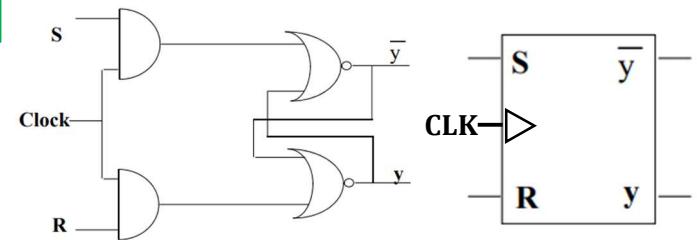
→ **LOGICA COMBINATORIA** = blocco che, tramite i valori in input (x_i) e il valore della variabile di stato (y_i oppure S_i), produce lo stato futuro (Y_i oppure F_i) e i valori di output (z_i);

→ **FLIP-FLOP** (o "bistabile") = contiene la variabile di stato (tempo illimitato); può essere:

- **ASINCRONO (SR)** = 2 porte NOR (4 transistor), y_1 opposto di y_2 ;

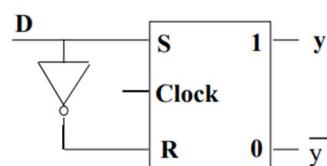


- **SINCRONO** = ASINCRONO + segnale di CLOCK (se CLK = 0, valori costanti; se CLK = 1, valori cambiati). Dunque 2 NOR (4 transistor) + 2 AND (4 transistor) = 8 transistor;



⚠ In realtà per i FF sincroni si usa il "**FLIP-FLOP D**" dove:

- CLK = 0, y = costante;
- CLK = 1, y = D.



⚠ I FF sono pilotati tutti dallo stesso segnale di clock!

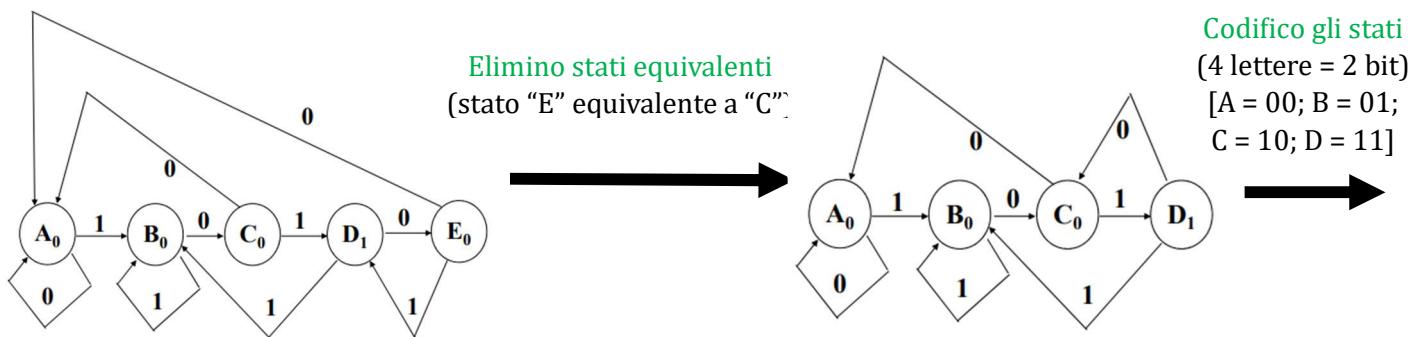
→ La **TAVOLA DEGLI STATI** è la TAVOLA DI VERITÀ DELLA LOGICA COMBINATORIA! Questo perché:

- n° ingressi = n° input (x_i) + n° FF (y_i oppure S_i);
- n° uscite = n° output (z_i) + n° FF (Y_i oppure F_i);
- n° stati = $2^{n_{FF}}$;
- n° righe tavola = $2^{n_{ingressi}} * n_{stati}$.

→ Dunque per progettare un circuito sequenziale sincrono (FSM) bisogna:

- costruire il **DIAGRAMMA DEGLI STATI**;
- **MINIMIZZARLO** (eliminare gli stati equivalenti) e **CODIFICARE** gli stati (4 stati = 2 FF);
- costruire la **TAVOLA DEGLI STATI** (tavola di verità);
- **SINTETIZZARLA** (mappe di karnaugh -> funzioni booleane -> circuito minimo).

ESERCIZIO ESAME → Dato il diagramma degli stati della macchina di Moore, arrivare al circuito minimo:



INPUT	S0	S1	F0	F1	OUTPUT
0	0	0	0	0	0
1	0	0	0	1	0
0	0	1	1	0	0
1	0	1	0	1	0
0	1	0	0	0	0
1	1	0	1	1	0
0	1	1	1	0	1
1	1	1	0	1	1
(input)	(stato)	(futuro)	(output)		

Creo la tabella degli stati
(da cui tiro fuori le funzioni output, F0, F1
tramite le mappe di karnaugh)

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

$$\text{Output} = S0 \cdot S1$$

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

$$F0 = S1 \bar{I} + S0 \bar{S1} I$$

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

$$F1 = I$$

⚠ Il disegno del circuito minimo di un sequenziale prevede il disegno anche dei FF, dunque facoltativo!

I circuiti sequenziali (FSM) possono essere di:

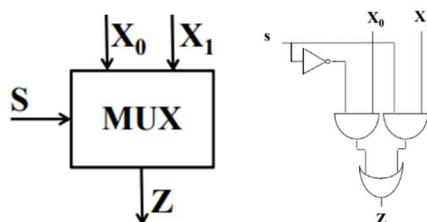
- MEALY = output (z_i) dipende da valori in input (x_i) e dagli stati (S_i);
- MOORE = output (z_i) dipende solo dagli stati (S_i).

⚠ Per garantire il funzionamento del circuito sequenziale sincrono, bisogna avere $T_{CLOCK} > \text{ritardo max}$ nella logica combinatoria (che dipende dalla tecnologia e dalla profondità del cammino critico). Inoltre, la FREQUENZA MASSIMA DI CLOCK [Hz] si ottiene sommando il ritardo di ogni porta logica e FF!

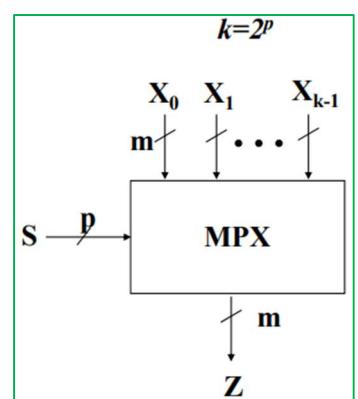
2) Progetto 2 – MODULI e REGISTRI

3 Il progettista a livello di registri manipola le WORD (parole) e usa:

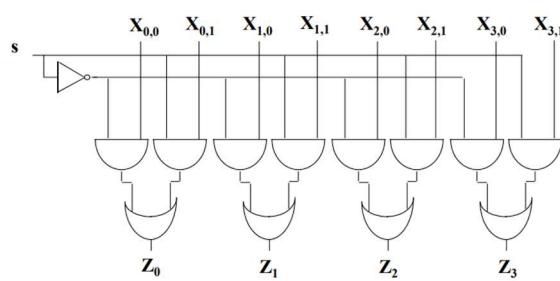
- **MULTIPLEXER** = riceve dei segnali in input (x_i) e produce come output alcuni di questi segnali (z_i), filtrandoli attraverso dei bit di selezione (S_i); Un esempio è il **MUX 2x1** (2 ingressi, 1 uscita), dove dato che ho 2 ingressi basta 1 bit di selezione ($x_0 \rightarrow S = 0; x_1 \rightarrow S = 1$):



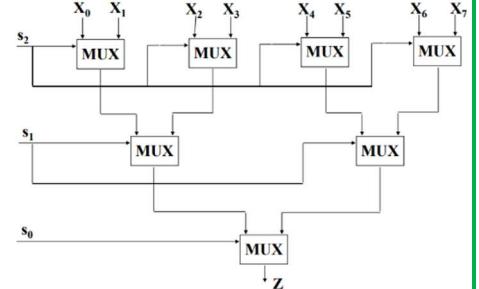
⚠ Internamente, un **MUX 2x1** è composto da 2 porte AND (per fare l'and tra l'input x_i e il bit di selezione S), 1 porta NOT (per far uscire $S = 0$ e $S = 1$) e 1 porta OR (per l'output z_i)!



→ Dunque, se si vuole realizzare un MUX 2x1 con input da 4 bit ciascuno (e dunque output da 4 bit), bisogna mettere in parallelo **4 MUX 2x1 collegati allo stesso bit di selezione**:



I bit di selezione (s_i) aumentano solo se aumentiamo gli ingressi (x_i) [per es. un MUX 8x1 con input da 1 bit]!

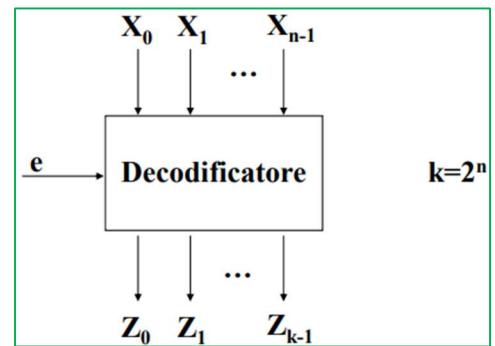
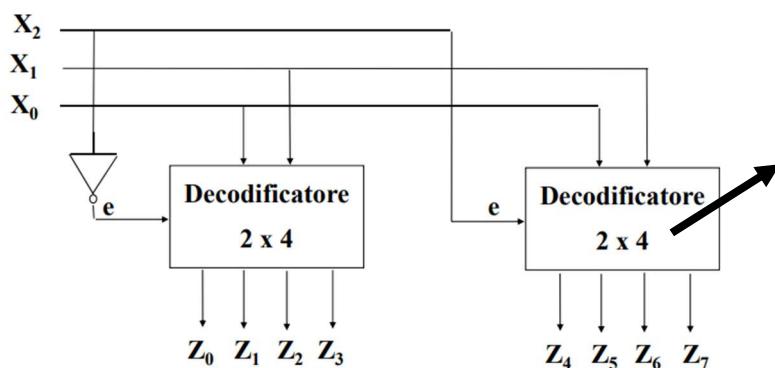


⚠ Dato che il MUX è composto da 2 AND, 1 OR e 1 NOT, questo modulo può implementare qualsiasi circuito combinatorio [perché {AND, OR} = insieme completo]!

- **DECODER (DECODIFICATORE)** = riceve n segnali in input (x_i) e produce 2^n linee di output (z_i), di cui però è attiva solo quella di indice uguale al valore in input; il segnale di enable (e) attiva l'uscita se $e = 1$ e la mette a zero se $e = 0$.

I decoder convertono 'BIT → DECIMALE'; hanno delle porte AND davanti alle uscite e porte NOT su 1 delle linee di input (per averli sia normali che negati).

⚠ Un esempio è il **DECODIFICATORE 3→8**:



X	e	Z
00	1	0001
01	1	0010
10	1	0100
11	1	1000
-	0	0000

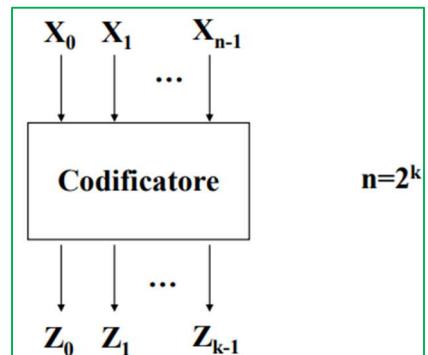
Qui x_2 (il bit più significativo [MSB]) viene usato come segnale di enable perché se voglio in uscita un:

- n° da 0 a 3 → $x_2 = 0$ [$000 = 0, 001 = 1, 010 = 2, 011 = 3$], dunque metto $e = \overline{x_2}$;
- n° da 4 a 7 → $x_2 = 1$ [$100 = 4, 101 = 5, 110 = 6, 111 = 7$], dunque metto $e = x_2$.

- **ENCODER (CODIFICATORE)** = opposto del decoder, ovvero riceve 2^n segnali in input (x_i) e produce n linee di output (z_i), di cui però è attiva solo quella di valore uguale all'indice in input.

La tavola di verità di un **CODIFICATORE 4→2** è opposta al decodificatore 2→4, ovvero:

X	Z
0001	00
0010	01
0100	10
1000	11



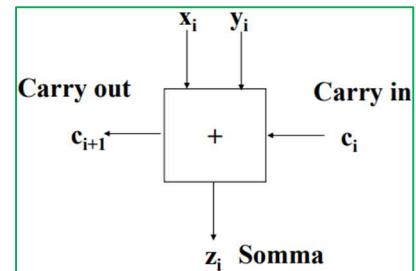
⚠ Nella realtà si usano i **CODIFICATORI PRIORITARI** ("priority encoder"), dove guardiamo i bit dal MSB al LSB (da sx a dx) e a seconda di dove troviamo il primo 1, capiamo che cosa dare in output [es. $x = 1011 \rightarrow z = 11; x = 0100 \rightarrow z = 10; x = 0111 \rightarrow z = 10; x = 0011 \rightarrow z = 01 \dots$].

⚠ In alcuni casi si aggiungono l'enable ($e = 0 \rightarrow$ tutte le uscite = 0) e l'input active ($e = 1 \wedge$ almeno 1 linea attiva → input active = 1).

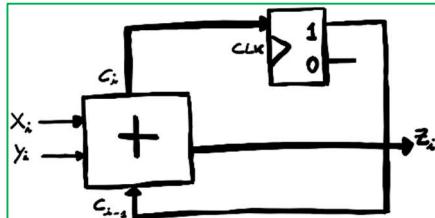
→ Da qui iniziano i **MODULI ARITMETICI**!

- **SOMMATORI** = per parlare dei sommatori, dobbiamo introdurre il **FULL-ADDER**, ovvero l'unità che fa la somma dei 2 operandi + il carry (riporto) in input e ci da 1 risultato + il carry in output (componete sommatori più grandi).

Ci sono diversi tipi di sommatori:

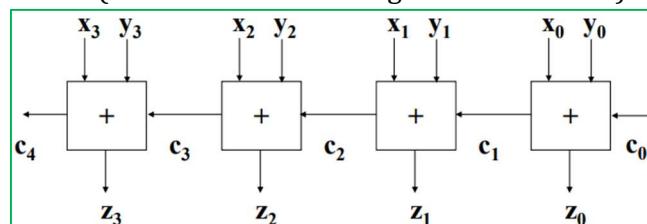


- **SERIALI** = il FF memorizza il carry di un bit e lo riporta sul bit successivo nel successivo clock, mentre il FULL-ADDER fa la somma degli operandi e del riporto del bit precedente (preso dal FF). Dunque:



- costo minimo;
- sequenziale;
- lento (se n = bit operandi, allora n periodi di clock).

- **COMBINATORI** = circuito a 2 livelli per sommare 2 numeri su n bit e ottenere $n + 1$ bit di output; dunque circuito minimo, veloce, ma al crescere di n diventa impegnativo.
- **RIPPLE CARRY ADDER** = costruito con n FULL-ADDER in cascata (1 per bit), quindi senza FF (il carry e il risultato va direttamente al prossimo FULL-ADDER) [solo combinatorio], ma con tempo di output pari a $t_{RISULTATO} = n * \Delta$ (con Δ = ritardo del singolo FULL-ADDER). Dunque ha costo proporzionale ad n .

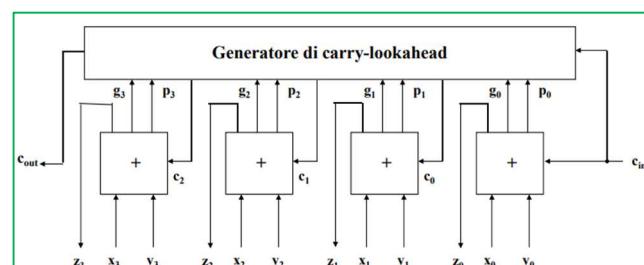


- **CARRY-LOOKAHEAD** = come il RIPPLE CARRY ADDER, ma i carry non vengono passati da un modulo all'altro in cascata, bensì vengono gestiti dal GENERATORE DI CARRY-LOOKAHEAD (una logica combinatoria), dove

$$c_{i+1} = x_i y_i + x_i c_i + y_i c_i = g_i + p_i c_i$$

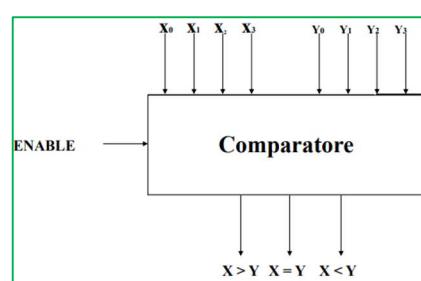
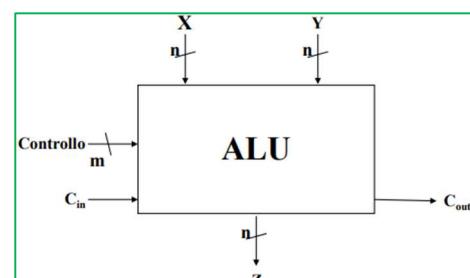
con **generazione** e **propagazione** del carry.

Dunque viene ridotto il ritardo (più veloce), ma ha costo che cresce esponenzialmente con n .



⚠ Si possono quindi usare anche soluzioni MISTE!

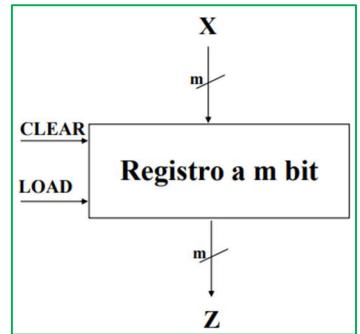
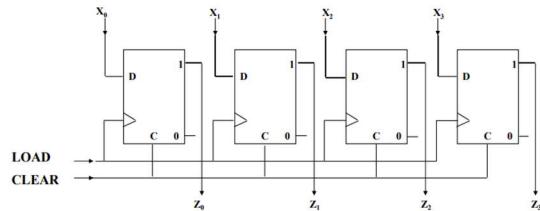
- **ALU (Unità Aritmetico-Logica)** = svolge le principali funzioni logiche e aritmetiche (somma, sottrazione, negazione, and, or, not, exor), che vengono selezionate da alcuni bit di controllo.
- **COMPARATORE** = compara 2 operandi partendo dai bit MSB:



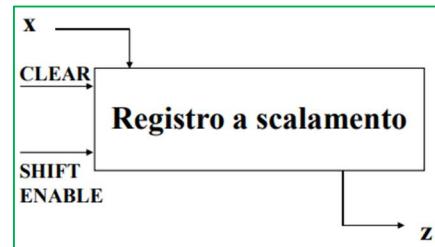
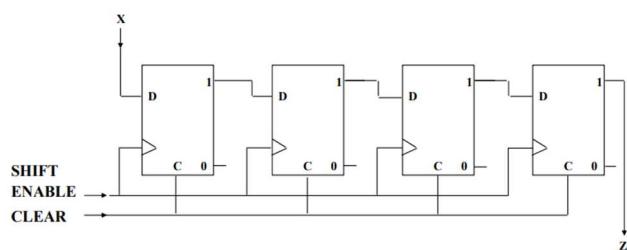
⚠ Unendo comparatori per n° a 4 bit, si ottengono comparatori per numeri su più bit (es. da 16 bit)!

→ Da qui iniziano i **MODULI SEQUENZIALI** (basati sui FF, dunque con funzione di memoria)!

- **REGISTRO** = batteria di FF che memorizza dei valori (clear = azzerza le uscite; load = memorizza i valori in input). Per esempio, in un registro a 4 bit, il load pilota il clock (mentre il clear azzerà):

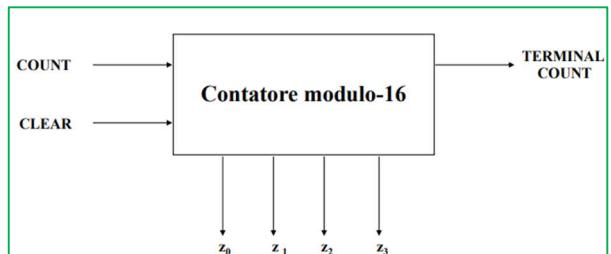
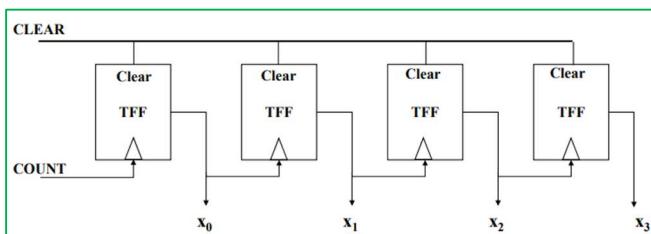


⚠ Importanti sono gli **SHIFT REGISTER (REGISTRI A SCALAMENTO)** che hanno anche il segnale di shift enable (possiamo scalare a sx/dx i valori contenuti nei FF [per lo shift completo, ovvero sx + dx, possiamo usare dei MUX per selezionare in quale verso scalare]). Questi registri sono utili per effettuare la **sll** e la **srl** richiesta dai processori (per moltiplicare/dividere per 2) ma anche per la conversione “seriale↔parallelo”.

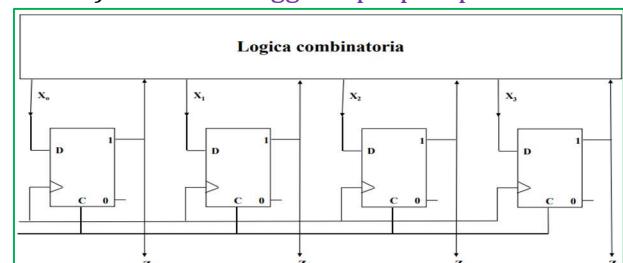


- **COUNTER (CONTATORE)** = si possono usare come **program counter**, **contatori di eventi** o **divisori di frequenza** (dato un segnale, lo passiamo al counter che con il terminal count [ovvero il risultato finale, associato ad un flag di terminazione] lo divide). Ci sono 2 tipi:

ASINCRONO (RIPPLE COUNTER) = contiene **Flip-Flop T** (FF T) [ovvero dei FF D con l'uscita connessa all'ingresso attraverso un inverter]. Il **costo è minimo**, ma **al crescere di n è più lento**.



SINCRONO = è un **registro collegato ad una logica combinatoria**, la quale prende il valore dei FF e lo incrementa contemporaneamente per tutti i FF (**tutte le uscite assumono contemporaneamente il valore corrente**). Ha costo maggiore proprio per la LC.



- **BUS e MEMORIE** (di cui parleremo in seguito);

⚠ Nel progetto si possono adottare **2 strategie** (a seconda della finalità e del costo):

- **ASIC (Application Specific IC)** = circuiti con **elevati costi di progettazione/produzione (in serie)**, ma **basso costo unitario**, più veloce ed efficiente (es. circuiti prodotti in serie per una grossa azienda);
- **FPGA (Field Programmable Gate Array)** = dispositivi **programmabili** per realizzare circuiti specifici; **bassi costi di progettazione (non produzione in serie)**, ma **elevato costo unitario**, meno veloce ed efficiente. Possono essere **OTP (One-Time Programmable)** oppure possono avere una **memoria** dove arriva il **bitstream** contenente le funzioni da svolgere.

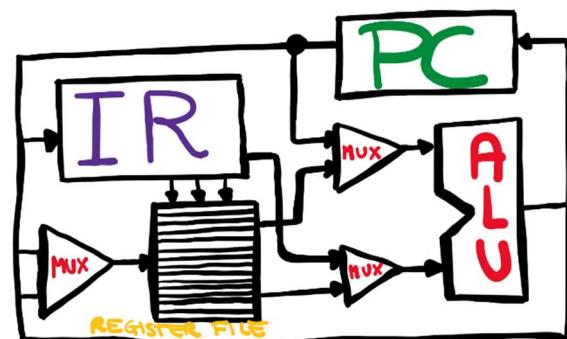
3) PROCESSORI e MIPS

Il **PROCESSORE (CPU)** è la **Central Processing Unit** di un “sistema a processore” (es. von neumann) e può essere una parte (**CORE**) di un SoC o di un microcontrollore (o direttamente un IC a sé). Il processore esegue le istruzioni contenute nel suo ISA (Instruction Set Architecture), le quali sono composte da 2 fasi:

- **FETCH** (lettura del codice da memoria [accesso in memoria]);
- **EXECUTE** (codice decodificato ed eseguito).

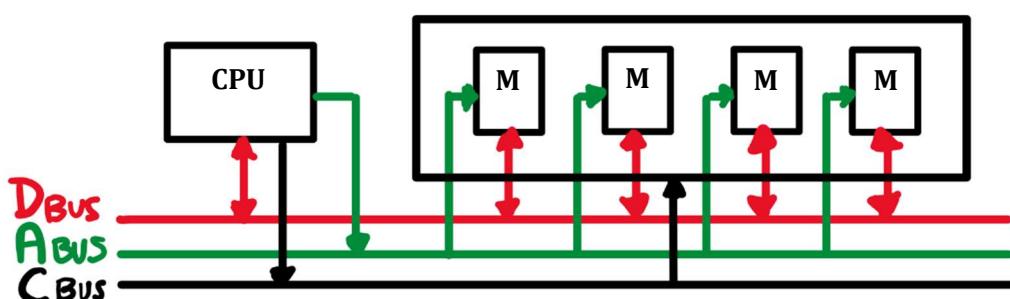
L'OS (Sistema Operativo) traduce il codice ad alto livello (es. C) in **CODICE MACCHINA** (eseguibile dalla cpu) e lo mette in memoria; dato che però gli accessi in memoria sono molto costosi a livello di tempo (bottle-neck cpu), nelle cpu troviamo i **REGISTRI** dove vengono salvati gli operandi (presi dalla memoria) e sono pronti all'uso [**MIPS** ha 32 registri da 32 bit] → nel mips (32 bit):

- **PROGRAM COUNTER (PC)** = registro che contiene l'indirizzo per il fetch della prossima istruzione;
- **INSTRUCTION REGISTER (IR)** = registro che contiene l'istruzione corrente;
- **REGISTER FILE** = i 32 registri che contengono gli operandi/istruzioni (32*32 bit = 1024 bit).

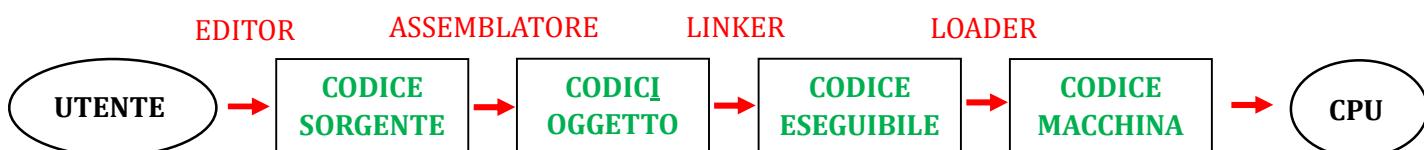


⚠ I calcoli sugli indirizzi vengono effettuati tramite la **ALU** (+4 per indirizzo successivo [guarda sotto perché])!

⚠ In generale le memorie sono organizzate a byte (1 word = 1byte = parallelismo di 8 bit), mentre il **MIPS** (e dunque il suo bus) ha un parallelismo di 4 BYTE (ovvero 32 bit, come già detto): questo avviene perché non ho 1 memoria ma un banco di 4 memorie, perciò se la CPU manda 1 indirizzo, ciascuna memoria risponde con la word a quell'indirizzo (ovvero sui 32 bit del DBus vengono messe le 4 word da 1 byte provenienti dalle 4 memorie). Questo ci garantisce 4 word (4 Byte di dati scambiati) con 1 solo accesso (1 Byte di indirizzo) [**BYTE – ADDRESSED**]! Dato che l'ABus del mips è da 32 bit, esso può accedere a 2^{32} word (4 GB di memoria)!



→ Ora vediamo infatti come funzionano i programmi per capire come funziona la CPU (lo faremo attraverso **ASSEMBLY MIPS**):



⚠ La differenza tra codice eseguibile e macchina è che l'eseguibile viene scritto senza sapere dove verrà messo, mentre il macchina viene sistemato proprio per funzionare dove verrà collocato (questo nei sistemi general purpose [dove l'OS carica il macchina nella RAM] perché negli special purpose so già dove il codice viene collocato [in quanto viene caricato direttamente in una flash per essere eseguito] e dunque eseguibile ≈ macchina).

→ A differenza dei linguaggi ad alto livello dove il **COMPILATORE** associa ad ogni nostra funzione una funzione di basso livello, in **ASSEMBLER** le istruzioni che scriviamo (in **ASCII**) sono le stesse che eseguirà la CPU (**l'ASSEMBLATORE** le tradurrà solo in bit [0,1]) [il compilatore è però più efficiente dell'uomo].

Il mips è un **RISC** (Reduced Instruction Set Computer) dunque opera su istruzioni semplici e regolari; infatti le istruzioni complesse vengono spaccate in tante piccole istruzioni semplici. Come abbiamo già detto, il **MIPS REGISTER SET** (ovvero l'elenco dei **REGISTER FILE** del mips) è composto da 32 registri da 32 bit che sono:

NOME	N°	USO	
\$0	0	Contiene la costante '0'	Non scrivibile (non modificabile)
\$at	1	Assembler Temporary	
\$v0, \$v1	2, 3	Valori di ritorno della funzione	Passaggio di procedure (syscall)
\$a0-\$a3	4-7	Argomenti della funzione	Passaggio di parametri (syscall)
\$t0-\$t7	8-15	Valori Temporanei [Temporaries]	Registri usabili dal programmatore (\$t) *
\$s0-\$s7	16-23	Variabili salvate	
\$t8, \$t9	24, 25	(Altri tmp)	*
\$k0, \$k1	26, 27	Valori tmp dell'OS	Usati dall'OS
\$gp	28	Global Pointer	
\$sp	29	Stack Pointer	Usati dal sistema
\$fp	30	Frame Pointer	
\$ra	31	Return Address della funzione	Return Address per le procedure

Il modo in cui l'assemblatore codifica le varie istruzioni su 32 bit, ci permette di dividerle in **3 CATEGORIE**:

- **R-TYPE** → operandi = 3 registri;

op	rs	rt	rd	shamt	funct
Codice Operativo {6 bit} (000000 R-Type)	2 Registri Operandi {5 bit + 5 bit}		Registro Risultato {5 bit}	Shift Amount (scalamenti) {5 bit}	Codice Funzione {6 bit}

- **I-TYPE** → operandi = 2 registri + 1 immediato (costante);

op	rs	rd	imm
Codice Operativo {6 bit}	Registro Operando {5 bit}	Registro Risultato {5 bit}	Immediato su 16 bit in CA2 {16 bit → $-2^{15} \leq x < 2^{15}$ }

- **J-TYPE** → operandi = address;

op	addr
Codice Operativo {6 bit}	Indirizzo dove devo saltare, da scrivere nel PC (Program Counter) {26 bit → $2^{26} = 64$ MB accessibili}

Ora vediamo tutte le **ISTRUZIONI UTILI** (a che categoria appartengono, come vengono codificate e il loro uso):

- **add** \$t2, \$t0, \$t1 # \$t2 = \$t0 + \$t1
- **sub** \$t2, \$t0, \$t1 # \$t2 = \$t0 - \$t1
- **and** \$t2, \$t0, \$t1 # se bit di \$t0 e bit di \$t1 entrambi = 1, allora bit di \$t2 = 1
- **or** \$t2, \$t0, \$t1 # se bit di \$t0 = 1 oppure bit di \$t1 = 1, allora bit di \$t2 = 1
- **xor** \$t2, \$t0, \$t1 # se solo bit di \$t0 = 1 oppure solo bit di \$t1 = 1, allora bit di \$t2 = 0
- **sll** \$t2, \$t0, 1 # scalo a sx \$t0 di 1 posizione (ovvero $\times 2$) e metto risultato in \$t2
- **srl** \$t2, \$t0, 2 # scalo a dx \$t0 di 2 posizioni (ovvero $\div 2^2$) e metto risultato in \$t2
- **sra** \$t2, \$t0, 1 # esiste perché **srl** (divisione) funziona solo per positivi/unsigned
- **slrv** \$t2, \$t0, \$t1 # queste 3 istruzioni sono uguali a prima, ma lo **shamt** è in \$t1
- **srlv** \$t2, \$t0, \$t1
- **srav** \$t2, \$t0, \$t1

⚠ Nelle posizioni scalate negli **SHIFT** vengono messi degli zeri!

- **addi** \$t2, \$t0, 4 # \$t2 = \$t0 + 4 → si usa anche per la **subi** mettendo un immediato negativo
- **andi** \$t2, \$t0, 25 # come le espressioni precedenti ma con un immediato su 16 bit
- **ori** \$t2, \$t0, 2
- **xori** \$t2, \$t0, 45

⚠ Quando si fanno operazioni tra registri (32 bit) e immediati (16 bit), vanno aggiunti 2 Byte (16 bit) davanti all'immediato per farli comunicare; se l'istruzione è:

- aritmetica → estensione del segno (se immediato positivo aggiungo 16 zeri, se negativo 16 “uni”);
- logica → estensione con 16 zeri.

- **lw** \$t2, (\$t0) # **Load Word** (Memory Read) → caricare in \$t2 la parola all'indirizzo \$t0
- **sw** \$s0, 8(\$t0) # **Store Word** (Memory Write) → salvo la parola di \$s0 in memoria all'indirizzo (\$t0 + 8) [8 è l'**OFFSET**]
- **lb** \$t2, 4(\$t0) # **Load Byte** → uguale a lw ma per singolo byte
- **sb** \$t2, 4(\$t0) # **Store Byte** → uguale a sw ma per singolo byte

⚠ Solo le istruzioni **lw**, **sw**, **lb**, **sb** (e le **PSEUDOISTRUZIONI li e la**) possono accedere in memoria; infatti meno ne mettiamo, più il programma sarà efficiente!

⚠ Abbiamo detto che con 1 Byte di indirizzo nel mips prendo 4 Byte (1 word da 32 bit), ma in che ordine vengono presi questi Byte? La word può essere memorizzata a partire dal:

- LSB (Least Significant Byte) → **LITTLE-ENDIAN**;
- MSB (Most Significant Byte) → **BIG ENDIAN**.

[È importante capire l'ordine specialmente per le **lb** e **sb**!]

- **beq** \$t1, 2, loop # **Branch if Equal** → salto a 'loop' se \$t1 == 2
- **bne** \$t0, \$t2, exit # **Branch if Not Equal** → salto a 'exit' se \$t0 != \$t2

⚠ Nonostante siano dei “**SALTI CONDIZIONATI**”, le istruzioni di **BRANCH** sono delle **I-Type** perché i 16 bit dell'immediato contengono il valore da sommare al Program Counter per avere l'indirizzo a cui saltare (bastano 16 bit perché il programmatore non farà grossi salti se sono condizionati)!

- **j** loop # **Jump** → Salta a 'loop' (in PC va indirizzo di 'loop')
- **jal** ciclo # **Jump and link** → Per saltare dal main alle procedure
- **jr** \$ra # **Jump Register** → Per uscire dalle procedure (o al posto di syscall 10)

⚠ La **jr** è una **R-Type** perché ha tutto a zero eccetto i 5 bit del registro a cui salto e il codice funzione (9)!

In assembler si possono anche GENERARE DELLE COSTANTI:

- 16 bit → **addi** \$s0, \$0, 0x4F3C
- 32 bit → **lui** \$s0, 0xF4CD # scrive la parte alta della costante nei 16 bit alti di \$s0
ori \$s0, \$s0, 0x875C # scrive la parte bassa della costante nei 16 bit bassi di \$s0
- **li** \$t0, 4 # carico 4 in \$t0
- **la** \$t0, msg # carico l'indirizzo di 'msg' in \$t0

⚠ Viene usato nelle PSEUDOISTRUZIONI (sequenza di istruzioni mascherata in un'unica istruzione) **li** e **la** dove:

- se l'immediato della li è da 16 bit basta una ori;
- se l'immediato della li è da 32 bit (oppure nella la) serve lui + ori con operando \$at.

⚠ Per fare le moltiplicazioni e divisioni abbiamo 2 alternative:

- Usare le ISTRUZIONI **mult** e **div** (2 operandi) e spostare i risultati dai registri **hi** e **lo** in registri accessibili con **mflo/mfhi** \$t0:
 - **mult** \$s1, \$s0 # il risultato (32 x 32 = 64 bit) su 64 bit viene spaccato in {hi, lo}
 - **div** \$s1, \$s0 # quoziente = lo, resto = hi
- Usare le PSEUDOISTRUZIONI **mul** e **div** (3 operandi) che fanno "mult/div + mflo":
 - **mul** \$t2, \$t1, \$t0 # \$t2 = \$t1 x \$t0
 - **div** \$t2, \$t1, \$t0 # \$t2 = \$t1 / \$t0

→ Altre pseudoistruzioni utili sono:

- **slt** \$t2, \$t0, \$t1 # if (\$t0 < \$t1) \$t2 = 1; else \$t2 = 0
- **slti** \$t2, \$t0, 4 # if (\$t0 < 4) \$t2 = 1; else \$t2 = 0
- **bge** \$t0, \$t1, loop # if (\$t0 >= \$t1), branch a loop
- **ble** \$t0, \$t1, loop # uguale ma con (\$t0 <= \$t1)
- **bgt** \$t0, \$t1, loop # uguale ma con (\$t0 > \$t1)
- **blt** \$t0, \$t1, loop # uguale ma con (\$t0 < \$t1)

⚠ Tutte le istruzioni e pseudoistruzioni viste finora esistono anche in VERSIONE UNSIGNED (add → addu)!

ESERCIZIO ESAME

1. Codificare l'istruzione R-Type add \$s0, \$s1, \$s2:

DECIMALE	0	17	18	16	0	32
BINARIO	000000	10001	10010	10000	00000	100000
ESADECIMALE	0x02328020					

2. Codificare l'istruzione I-Type lw \$t2, 32 (\$0):

DECIMALE	35	0	10	32
BINARIO	100011	00000	01010	0000000000100000
ESADECIMALE	0x8C0A0020			

→ Per quanto riguarda le collezioni di dati omogenei (vettori e matrici), in assembler devo gestirli io come dati aggregati, in quanto mi muovo da una cella all'altra del dato aggregato incrementando l'indirizzo +4 (muovo su riga) oppure +DIM*4 per le colonne delle matrici.

vett:	.word 1, 2, 3
mat:	.word 1, 2, 3 .word 4, 5, 6 .word 7, 8, 9

→ Ultima cosa di assembler sono le **SYSCALL** (System Call), dove gli argomenti della chiamata sono messi in **\$a0-\$a3**, mentre il tipo di operazione che deve fare l'OS è messa con un numero in **\$v0**; il valore di ritorno della syscall è in **\$v0**.

⚠ Per quanto riguarda la **CHIAMATA A PROCEDURE DAL MAIN**, lo schema è questo:

```

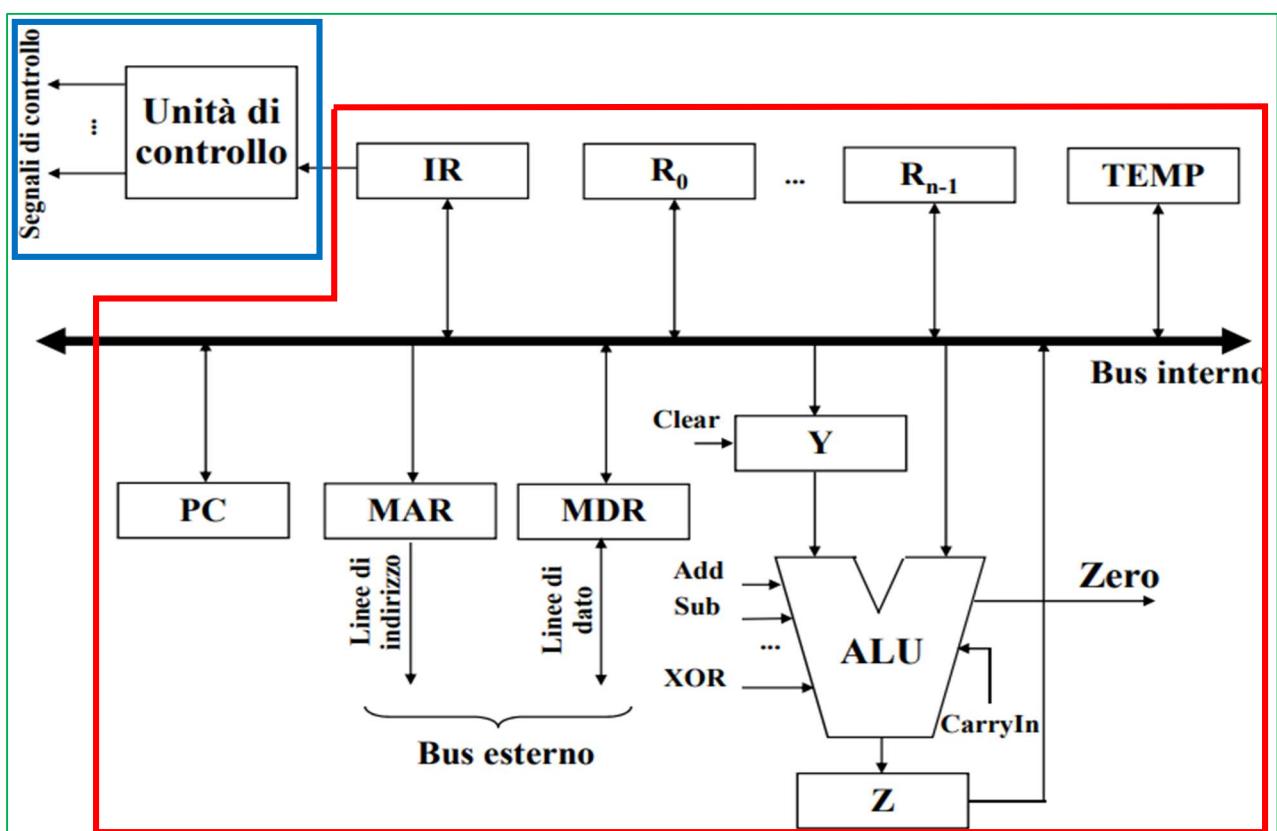
main:    subu $sp, $sp, 4          # creo spazio nello stack
         sw $ra, ($sp)           # metto il return address ($ra) nello stack
         ...
         jal funct               # salto alla procedura 'funct'
         lw $ra, ($sp)           # tornato al main, ripristino $ra
         addiu $sp, $sp, 4        # e resetto lo stack
         jr $ra                 # uso jr $ra al posto di (li $v0, 10 + syscall)
         .end main               # termino il main

funct:   ...                      # terminata la procedura, risalto al main con $ra
         jr $ra                 # termino la procedura 'funct'
         .end funct
  
```

STRUTTURA INTERNA DELLA CPU

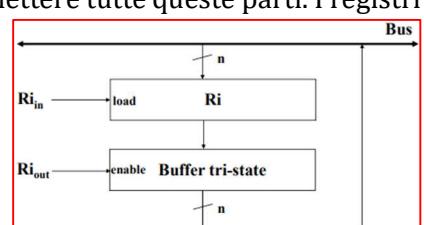
La CPU (processore) è composta da **2 parti**:

- 1) **UNITÀ DI ELABORAZIONE** (o DATA-PATH) = opera sui dati;
- 2) **UNITÀ DI CONTROLLO** (o UC) = genera i segnali di controllo.



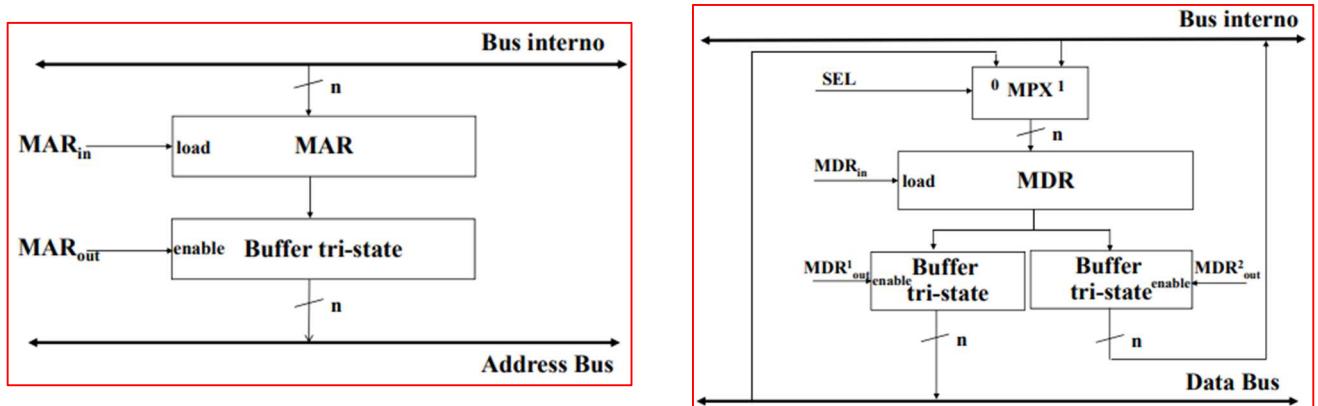
1) Nell'**UNITÀ DI ELABORAZIONE** troviamo i **Registri**, il **Program Counter (PC)**, l'**Instruction Register (IR)**, la **ALU**, dei **Registri di Appoggio** per la CPU e il **Bus Interno** al processore per connettere tutte queste parti. I registri sono attaccati al Bus Interno in modo che:

- segnale R_{in} pilota il **LOAD** nel registro dal bus;
- segnale R_{out} pilota il **BUFFER TRI-STATE** (mettendo enable = 1 e tutti gli altri enable = 0, fa sì che solo il registro di interesse faccia **STORE sul bus**).



Ora parleremo delle **MICROISTRUZIONI** (singole operazioni che esegue la CPU in un periodo di clock, che in sequenza ci danno l'istruzione), ma dobbiamo prima introdurre 2 particolari registri che mettono in comunicazione i bus interni ed esterni:

- **MAR (Memory Address Register)** = fa da tramite tra bus interno e ABus (contiene l'indirizzo da dare al ABus) [mette in comunicazione registri e ABus];
- **MDR (Memory Data Register)** = fa da tramite tra bus interno e DBus (contiene il dato da dare/ricevere al/dal DBus) [mette in comunicazione registri e DBus].



Ecco le **4 operazioni elementari** (scritte come sequenze di **MICROISTRUZIONI** che svolge la CPU) + **FETCH**:

LOAD (da memoria a registro)
MAR $\leftarrow R_1$ (R_1 = indirizzo sorgente)
READ
WAIT MFC
MDR \leftarrow BUS ESTERNO
R2 $\leftarrow MDR$ ($R2$ = registro destinazione)

STORE (da registro a memoria)
MDR $\leftarrow R_1$ (R_1 = dato)
MAR $\leftarrow R_2$ ($R2$ = indirizzo destinazione)
WRITE
WAIT MFC

MOVE (transfer tra registri)
R2 $\leftarrow R_1$
OPERATE (operazione + risultato in registro)
[add R3, R1, R2]
Y $\leftarrow R_1$
Z $\leftarrow Y + R_2$
R3 $\leftarrow Z$

⚠ **MFC (Memory Function Completed)** = segnale per capire quando la memoria (che è più lenta) ha terminato l'operazione (arriva direttamente poi all'UC che gestisce le operazioni dal Memory Controller)!

→ Prima di tutte le operazioni va aggiunto il **FETCH** che nel 1° clock legge dalla memoria l'indirizzo dell'istruzione da eseguire e nel 2° clock si aggiorna il PC con $PC += 4$ (non tocca i registri la fetch) [2 clock]:

FETCH
MAR $\leftarrow PC$
Y = 0
CARRY = 4
Z $\leftarrow PC + Y + CARRY$
READ
PC $\leftarrow Z$
WAIT MFC
MDR \leftarrow BUS ESTERNO
IR $\leftarrow MDR$

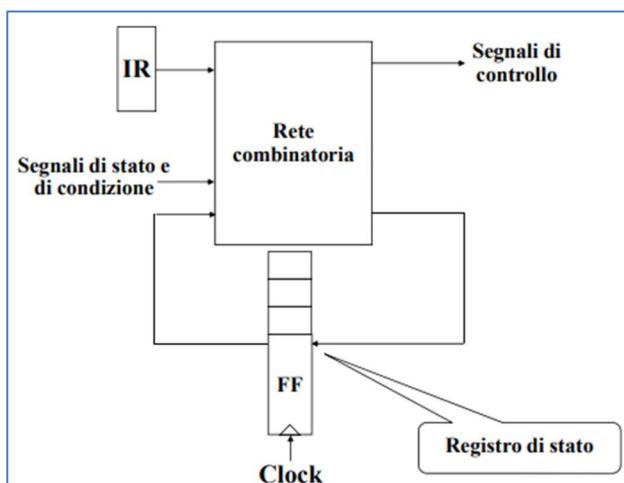
⚠ Posso ridurre il n° di microistruzioni (e i periodi di clock) facendo più istruzioni IN PARALLELO!

Per quanto riguarda i SALTI, se ho:

- **JUMP** (salto incondizionato) = $PC \leftarrow (\text{Operando di IR})_{out}$;
- **BRANCH** (salto condizionato) = faccio prima il confronto richiesto e poi uguale alla JUMP [nei processori x86 troviamo dei registri che svolgono il ruolo di "flag di condizione" per le branch].

2) Per quanto riguarda l'**UNITÀ DI CONTROLLO** ci sono 2 strategie:

- A. **UC CABLATA** = posso trattarla come un **CIRCUITO SEQUENZIALE (FSM)**, ovvero posso applicare i metodi visti a progetto ed implementarla con il **MODELLO DI HUFFMAN** (dunque la complessità è data dal numero di segnali di controllo e dal numero degli stati).



- Vantaggi:

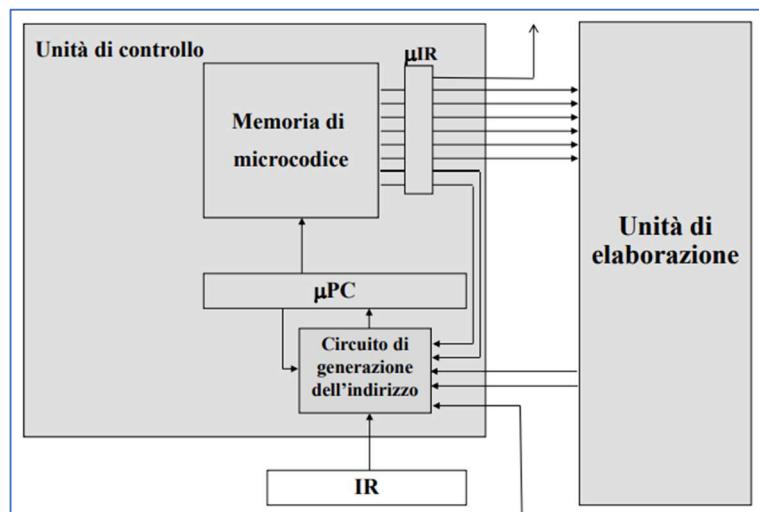
- Minimo uso silicio e **minimo costo circuito**;
- **Massima frequenza** di funzionamento;

- Svantaggi:

- Diagramma di stati grande = **difficile implementare**;
- **Difficile da modificare** (andrebbe riprodotto).

- B. **UC MICROPROGRAMMATA** = al posto di un circuito sequenziale, si usa una **MEMORIA DI MICROCODICE** (una ROM) dove viene memorizzato l'insieme delle **PAROLE DI CONTROLLO** (ovvero tutte le combinazioni dei segnali di controllo prodotti dalla UC, corrispondenti alle microistruzioni).

Come funziona? Si esegue una **read** sulla memoria di microcodice (usando l'indirizzo contenuto nel **micro-program counter [μPC]**) e si carica la word di controllo nel **micro-instruction register [μIR]**; questa pilota i segnali di controllo per l'unità di elaborazione e per generare l'indirizzo della prossima word di controllo (l'intero procedimento è eseguito in 1 solo periodo di clock).



- Vantaggi:

- **Maggiore flessibilità** (la modifica incide solo sulla memoria di microcodice);
- **Basso costo di progettazione** (no LC);

- Svantaggi:

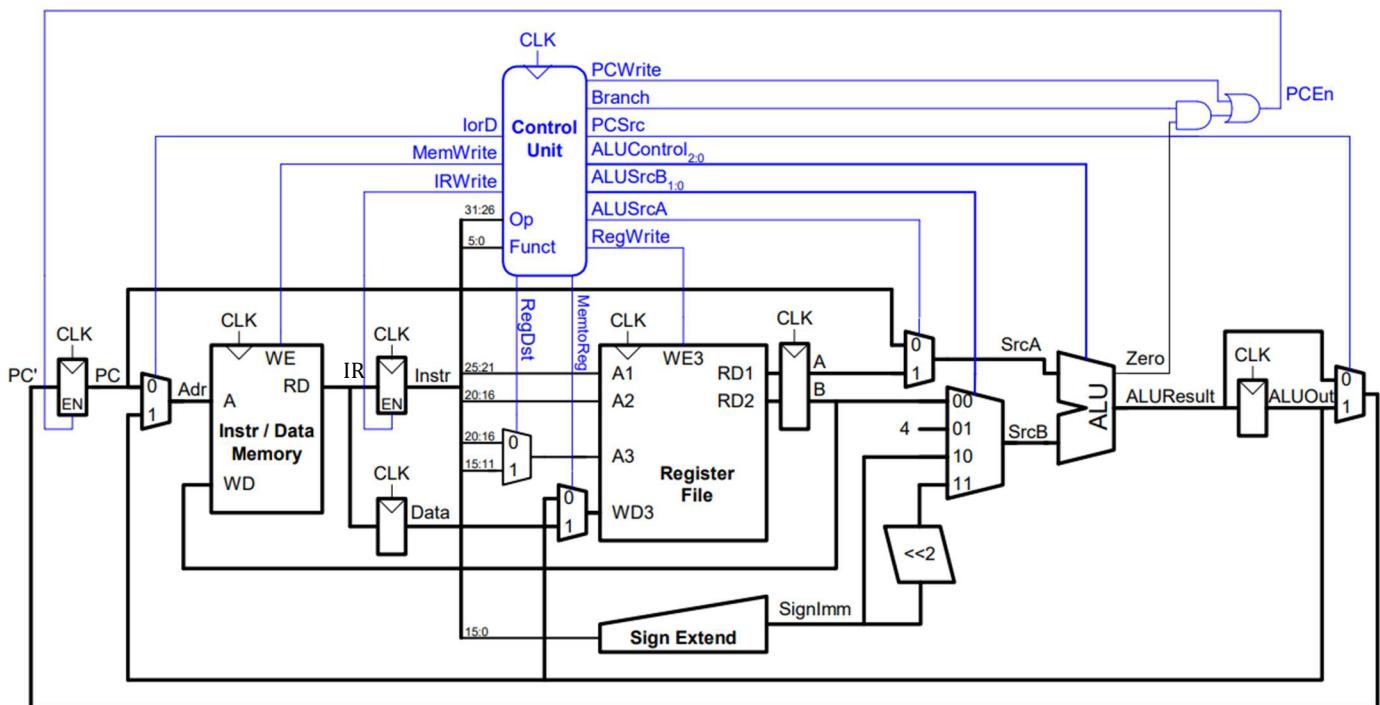
- **Minore velocità** (accesso a memoria di microcodice rende più lento);
- **Maggiore costo di produzione** (proprio per la memoria, dunque più silicio).

Esistono 2 tipi di **MICROPROGRAMMAZIONE** della memoria di microcodice:

- **ORIZZONTALE**: le word di controllo (microistruzioni) contengono 1 bit per ogni segnale di controllo (**più semplice**); dunque i segnali di controllo vengono pilotati direttamente (**massima velocità** di esecuzione), ma la lunghezza delle microistruzioni può diventare eccessiva e alcune combinazioni possono risultare inutili (**più hardware**);
- **VERTICALE**: i segnali di controllo vengono raggruppati in gruppi di bit (codificati); dunque è **meno veloce** (per i decoder che devono decodificare questi gruppi di segnali), ma le microistruzioni sono più corte [**gruppo di n segnali → log₂ n bit**] (**meno hardware**).

⚠ Tramite la ripartizione di segnali di controllo in **CLASSI DI COMPATIBILITÀ** (ovvero insieme di segnali che non vengono attivati insieme), possiamo minimizzare la lunghezza delle word di controllo (microistruzioni) (ovvero minimizzare il parallelismo della memoria di microcodice)!

Vediamo ora come si comportano i SEGNALI DI CONTROLLO quando eseguo le istruzioni nel mips (dato che il mips è un RISC, usiamo una versione del mips semplificata per questa parte che è una via di mezzo tra il CISC e la pipeline, ovvero l'architettura "multicycle"):



- **FETCH [2 CLOCK]:**

- 1° CLK = leggo dalla memoria (IorD = 0, MemWrite = 0) e scrivo su RD (CLK attivo nella Memory);
- 2° CLK = scrivo RD in IR (IRWrite = 1, CLK attivo in IR) e update PC += 4 (PCEn = 1, ALUSrcA = 0, ALUSrcB = 01 [ovvero il +4], ALUControl = 010 [somma] e PCSrc = 0).

⚠ I 2 clock di fetch precedono tutte le operazioni sotto elencate!

- **LOAD WORD (lw) [2 CLOCK di FETCH + 5 = 7 CLOCK]:**

- 3° CLK = lettura dell'IR in A1 (RegWrite = 0) che finisce in RD1 e viene scritto nel registro operando A (CLK attivo in A); in più si fa l'estensione del segno dell'immediato (ovvero dell'offset) dall'IR al MUX davanti alla ALU;
- 4° CLK = sommo A e l'immediato nella ALU (ALUSrcA = 1 [A], ALUSrcB = 10 [l'immediato], ALUControl = 010 [somma]) e scrivo il risultato nel registro risultato (CLK attivo in ALUResult);
- 5° CLK = scrivo l'indirizzo risultato nell'Address Register in memoria (ALUOut, IorD = 1, MemWrite = 0, CLK attivo in Memory);
- 6° CLK = scrivo RD nel Data Register (IRWrite = 0, CLK attivo in Data);
- 7° CLK = da Data scrivo in WD3 nel Register File con address scritto da IR in A3 (RegDst = 0, MemToReg = 1, RegWrite = 1, CLK attivo in Register File).

- **STORE WORD (sw) [2 CLOCK di FETCH + 3 = 5 CLOCK]:**

- 3° CLK = lettura dei due registri operandi (rs, rt) dall'IR in A1 e A2 (RegWrite = 0) che poi finiscono in RD1 e RD2, che scrivono nei registri operandi A e B (CLK attivo in A e B); in più si fa l'estensione del segno dell'immediato (ovvero dell'offset) dall'IR al MUX davanti alla ALU;
- 4° CLK = sommo A e l'immediato nella ALU (ALUSrcA = 1 [A], ALUSrcB = 10 [l'immediato], ALUControl = 010 [somma]) e scrivo il risultato nel registro risultato (CLK attivo in ALUResult);
- 5° CLK = scrivo l'indirizzo risultato nell'Address Register in memoria (ALUOut, IorD = 1, CLK attivo in Memory) e anche il registro operando B in WD (MemWrite = 1).

- **R-TYPE INSTRUCTION [2 CLOCK di FETCH + 3 = 5 CLOCK]:**

- 3° CLK = lettura dei due registri operandi (rs, rt) dall'IR in A1 e A2 (RegWrite = 0) che poi finiscono in RD1 e RD2, che scrivono nei registri operandi A e B (CLK attivo in A e B);

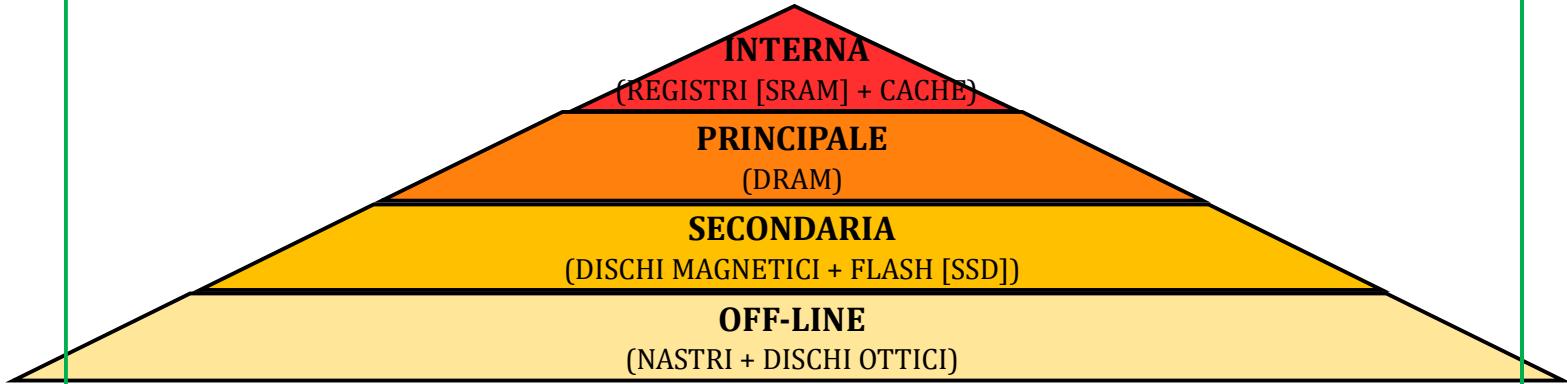
- 4° CLK = faccio l'operazione richiesta (**ALUControl = codice richiesto**) tra A e B (**ALUSrcA = 1 [A]**, **ALUSrcB = 00 [B]**) e scrivo il risultato nel registro risultato (**CLK attivo in ALUResult**);
 - 5° CLK = scrivo il risultato in WD3 nel Register File (**ALUOut**) con indirizzo da IR in A3 (**RegDst = 1**, **MemToReg = 0**, **RegWrite = 1** e **CLK attivo in Register File**).
- **BEQ** [2 CLOCK di FETCH + 2 = 4 CLOCK]:
 - 3° CLK = lettura dei 2 operandi da confrontare da IR in A1 e A2, poi da RD1 e RD2 in A e B (**RegWrite = 0**, **CLK attivo in A e B**); dunque **estensione del segno** dell'immediato (indirizzo) con somma tra PC e questo indirizzo ($PC \rightarrow MUX$, **ALUSrcA = 0 [PC]**, **ALUSrcB = 11 [indirizzo]**, **ALUControl = 010 [somma]**, **CLK attivo in ALUResult**);
 - 4° CLK = comparo A e B (**ALUSrcA = 1 [A]**, **ALUSrcB = 00 [B]**, **ALUControl = 110 [compare]**) e metto il risultato (0 o 1) in AND con il Branch e in OR con il PCWrite (**Branch = 1**, **PCWrite = 0**); inoltre metto l'indirizzo dell'eventuale salto **da ALUOut nel PC**.
- **J** [2 CLOCK di FETCH + 1 = 3 CLOCK]:
 - 3° CLK = prendo l'indirizzo da IR, lo moltiplico per 4 (sll 2) e lo combino con PC+4 (**PCWrite = 1**).

4) MEMORIE

L'obiettivo è realizzare un SISTEMA DI MEMORIE che faccia da compromesso tra costo e prestazioni (in modo da far attendere alla CPU il minor tempo possibile). Questi sistemi si fondano sui 2 principi di LOCALITÀ DEI RIFERIMENTI:

- **LOCALITÀ SPAZIALE** = riferimento in memoria tende ad essere vicino a quelli già usati;
- **LOCALITÀ TEMPORALE** = riferimento in memoria tende ad essere ripetuto dopo poco tempo.

Proprio a causa di questi 2 principi, bisogna fare in modo che i dati usati dalla CPU siano collocati nelle memorie più veloci, ovvero parliamo di **GERARCHIA DI MEMORIA** (costo e velocità ↑, dimensione e tempo di accesso ↓)



⚠ Quando si sceglie una memoria, si guarda costo e velocità (tempo di accesso, di ciclo e di trasferimento)!

Parlando di **AFFIDABILITÀ** delle memorie si ha:

- Mean Time To Failure (**MTTF**) = tempo prima di guasto permanente;
- Mean Time Between Failures (**MTBF**) = tempo tra 2 guasti transitori.

Le memorie si classificano anche a seconda del TIPO DI ACCESSO:

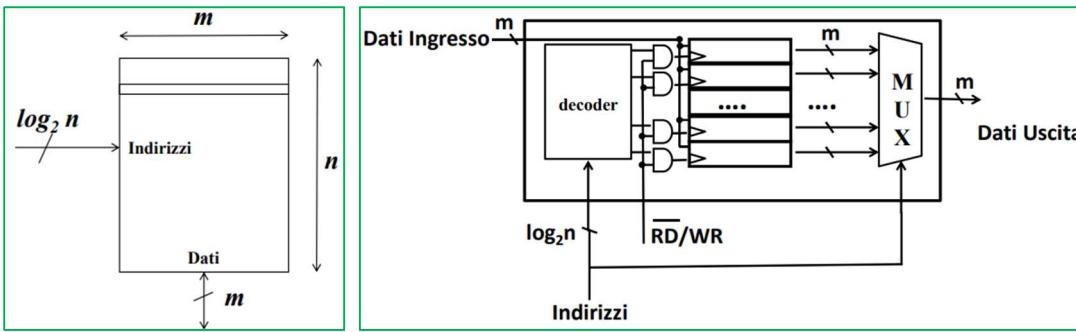
- **SEQUENZIALE** (nastri) = scandire ordine prefissato;
- **DIRETTO** (dischi) = ogni blocco ha un indirizzo e la ricerca del blocco è sequenziale;
- **CASUALE** (RAM) = accesso in ogni punto è uguale (read e write in ogni ordine);
- **ASSOCiativo** (alcune CACHES) = tramite il contenuto (una chiave).

MEMORIE AD ACCESSO CASUALE

Ogni cella può essere indirizzata indipendentemente ed è accessibile con Δt uguale; può essere:

- **SINCRONA** = memoria e chi la usa condividono il clock;
- **ASINCRONA** = memoria e chi la usa non condividono il clock e infatti necessitano di segnali:
 - CPU → MEMORIA [segnali di **controllo**]:
 - Chip Select (CS), da attivare per read/write;
 - Output Enable (OE), da attivare per write su bus condiviso;
 - WRITE ENABLE (WE), da attivare per il write.
 - MEMORIA → CPU [segnali di **stato**]:
 - Memory Function Completed (MFC, già visto prima), se memoria completato l'operazione;
 - Error, se memoria ha trovato una word con un dato errato.

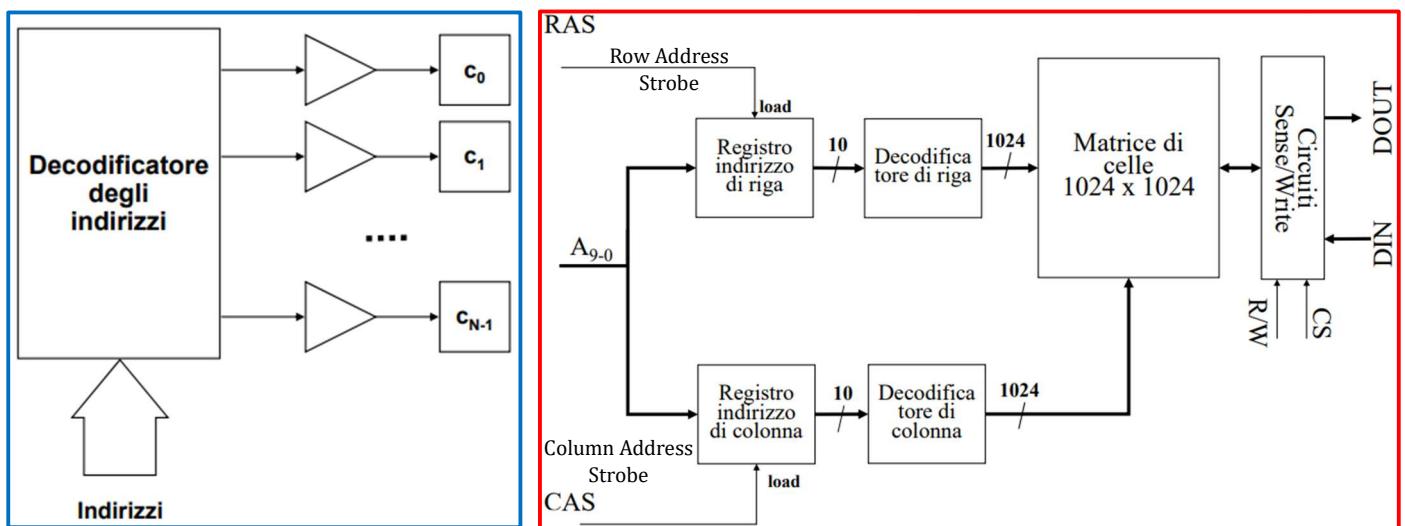
→ Internamente la memoria ha **n parole** (ovvero **n REGISTRI** [word register]) di **m bit ciascuna** (parallelismo di memoria); dunque ha un **DECODER** che riceve **$\log_2 n$ indirizzi** e un **MUX** per scegliere tra **m segnali di dato**. È anche presente un segnale di enable (che se messo a zero mette la memoria in stand-by) e della circuiteria per i segnali di controllo e di stato (una specie di UC della memoria) [es. LOAD attiva il clock dei FF delle word].



⚠ Se la memoria è grande, al posto del MUX posso usare un BUS INTERNO (sempre pilotato dal decoder)!

Ci sono 2 possibili organizzazioni:

- **ORGANIZZAZIONE A VETTORE** = il decoder riceve gli indirizzi e pilota 1 linea di output per ogni parola e dunque 1 driver per ciascuna linea (i driver aggiustano i valori di tensione);
- **ORGANIZZAZIONE A MATRICE** = per memorie più grandi; ci sono **2 decoder** (decoder delle righe e delle colonne) in modo da spacchettare il segnale in 2 parti e far passare sui pin metà dei bit di segnale per volta (se ho 20 bit di segnale, posso usare solo 10 pin, meno costi e più facile da realizzare su silicio).



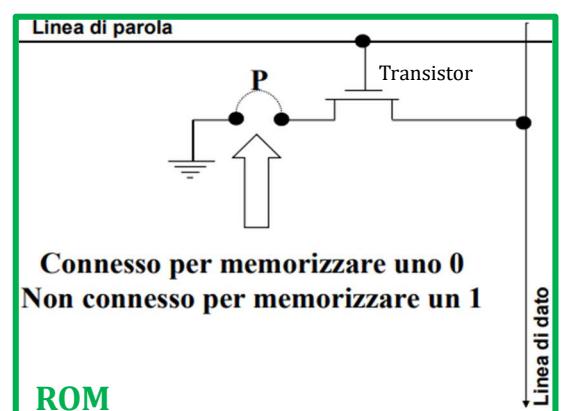
⚠ Se ho memoria organizzata a matrice e accesso sequenziale, conviene usare l'accesso in **PAGE MODE**, ovvero:

- Carico indirizzo di riga + RAS;
- Carico indirizzo di colonna + CAS;
- Accedo al dato;
- Riparto dal 2° step [perché cambia solo indirizzo di colonna, riga rimane uguale].

→ Le memorie sono i dispositivi a più alta densità su silicio; con le **MEMORIE A SEMICONDUTTORE**, vogliamo aumentare capacità e velocità e ridurre il consumo!

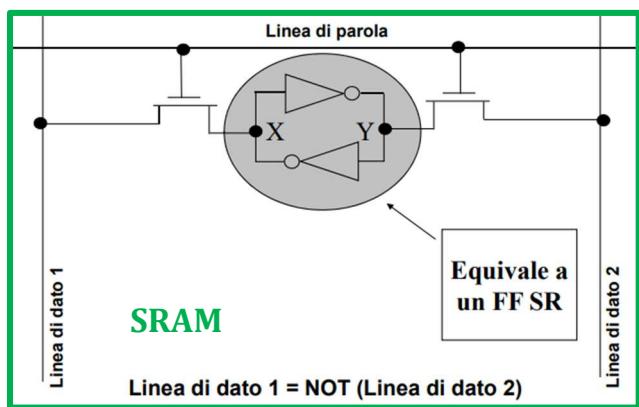
Vediamo ora la **CLASSIFICAZIONE** di queste memorie:

- **ROM (Read-Only Memory)** = memoria non volatile il cui contenuto è deciso prima della sua realizzazione (non riscrivibile, dunque solo lettura [no segnali di write]). Sono utili durante il boot di sistema.

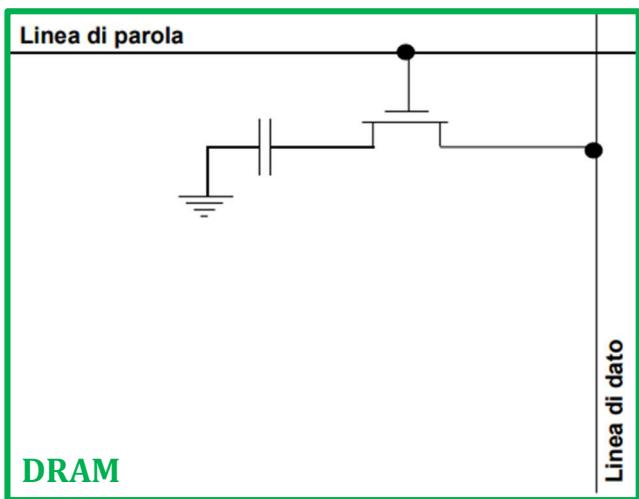


- **RAM** (Random Access Memory [accesso casuale*]) = memoria **volatile** e **riscrivibile** (read + write); 2 tipi:

- **SRAM (RAM Statica)** = ogni cella è composta da **1 FF SR** (ovvero la versione più semplice di FF composta da 2 NOR) [4 transistor] + **2 transistor**, dunque **ogni cella = 6 transistor** (è molto costosa infatti, ma è anche la memoria a semiconduttore più veloce esistente [registri della CPU]). Data la sua struttura si vede che le due linee di dato contengono sempre valori opposti; inoltre, la cella consuma solo quando conta (fintanto che è alimentata, il FF mantiene il valore);



- **DRAM** (RAM Dinamica) = ogni cella è composta da 1 transistor e da 1 condensatore (scarico = 0, carico = 1), dunque molto meno costosa; il condensatore però ha 2 problemi:
 - viene azzerato ogni volta che effettua una read, quindi ad ogni read segue una write per riscrivere il valore (**DESTRUCTIVE READOUT**);
 - tende a scaricarsi nel tempo, dunque ci vuole una circuiteria che faccia il **REFRESH** (rinfresco) delle word in memoria.



⚠ Se una cella di DRAM viene colpita da una **radiazione**, il suo valore può cambiare (specialmente con alta densità di integrazione); dunque per aumentare l'**AFFIDABILITÀ** della DRAM, a ciascun word viene associato un **CODICE DI PROTEZIONE**, che può essere:

- nel caso più semplice, 1 bit detto **CODICE DI PARITÀ** (calcolato e scritto durante il write della parola, ricalcolato durante la read e se \neq dalla write \rightarrow errore);
 - nella realtà, **CODICI DI HAMMING**, ovvero $(1 + \log_2 n)$ bit che possono correggere gli errori singoli e trovare gli errori doppi (**SECDED** = Single-Error Correction Double-Error Detection).

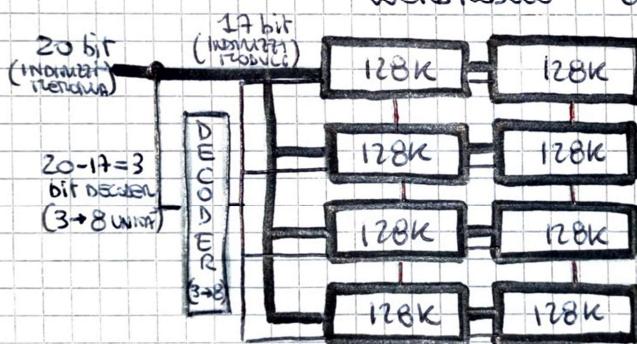
ESERCIZIO ESAME

Banco di memoria composto da 1M (2^{20}) parole da 16 (2^4) bit ciascuna, con dei moduli da 128K (2^{17}) parole da 1 Bye (2^3) ciascuna:

$$\text{QUANTI? } \text{BIT MEMORIA} = 2^{20} \cdot 2^4 = 2^{24} \quad \left. \begin{array}{l} \\ \text{BIT MODULO} = 2^{17} \cdot 2^3 = 2^{20} \end{array} \right\} \text{N° MODULI} = \frac{2^{24}}{2^{20}} = 2^4 = 16 \quad) \quad 2 \cdot 8 = 16$$

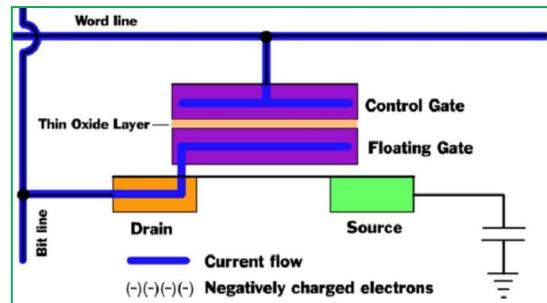
Cette 1

⑧ h° MODULI PER RIGA = $\frac{\text{Word memoria}}{\text{Word trasfuso}} = \frac{16 \text{ bit}}{8 \text{ bit}} = 2 \rightarrow \text{Dunque: } \begin{cases} \text{Colonne} = 2 \\ \text{Righe} = 8 \end{cases}$



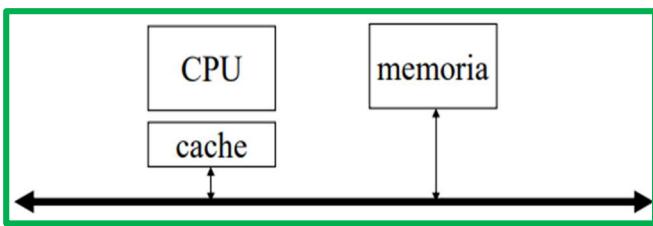
- **PROM** = scrivibili una sola volta (non-volatili), prima di essere montate nella scheda;
- **EPROM** = riscrivibili, ma per farlo vanno cancellate con dei raggi ultravioletti;
- **EEPROM** = più costose e meno dense delle EPROM, ma la scrittura avviene normalmente;

- **FLASH** (usate negli **SSD**) = riscrivibili e non-volatili; usano 1 transistor per bit (di tipo **Floating-Gate Transistor [MOSFET]**). Il read come per le RAM, ma il write più lento, a blocchi e con precedente cancellazione; 2 tipi:
 - o NAND FLASH **SLC** (Single-Level Cell) → 1 cella = 1 bit;
 - o NAND FLASH **MLC** (Multiple-Level Cell) → 1 cella = 2 bit.



- **CACHE** = sono memorie molto veloci (solitamente fatte di SRAM) interposte tra **CPU** e **MEMORIA PRINCIPALE**, create per collocarci i programmi da eseguire (principi spaziale e temporale di località dei riferimenti*). La probabilità che l'istruzione da eseguire si trovi nella cache è detta "**hit ratio**" (h), da cui:

$$t_{\text{medio di accesso della CPU in memoria}} = ht_{\text{accesso se in cache [HIT]}} + (1 - h)t_{\text{accesso se non in cache [MISS]}}$$



In caso di:

- **HIT** → CPU accede alla cache e non al BUS;
- **MISS** → CPU accede al BUS (ovvero alla memoria principale), facendo **LOAD-THROUGH** (ovvero bypassando la cache). Nel contempo si fa **EARLY RESTART** (viene messo nella cache il blocco seguente a quello appena acceduto in memoria principale).

⚠ Quando avviene un **MISS**, si riscrive in memoria principale il blocco corrente e si mette in cache al suo posto il blocco all'indirizzo di memoria che ha provocato il MISS. Come si riscrive il blocco in memoria? 2 meccanismi:

- o **WRITE BACK** = il DIRTY BIT (ogni linea ne ha 1) ci dice se la linea di cache è stata modificata; se è stata modificata viene riscritta in memoria, altrimenti è già scritta giusta;
- o **WRITE THROUGH** = ogni modifica avviene sia sulla cache sia sulla memoria principale (funziona se ho pochi operazioni di write).

⚠ Nei **SISTEMI MULTICORE**, per garantire la coerenza delle cache di ogni CPU con la memoria principale condivisa si usa il **BIT DI VALIDITÀ** (se disattivato, il blocco in quella linea ha un valore diverso dal blocco in memoria principale [in questo caso, ogni accesso al blocco comporta un **MISS**]!)

⚠ La **CACHE** è trasparente per la CPU (se il dato viene passato dalla cache o dal bus, alla CPU non importa)!

→ Ci sono diversi meccanismi di correlazione (MAPPING) tra le linee della cache e l'indirizzo emesso dalla CPU:

- **DIRECT MAPPING** → $k = i|N$ (k = linea cache; i = blocco memoria; N = n° linee cache);
 - o **Vantaggi**: facilmente implementabile;
 - o **Svantaggi**: se programma accede spesso a 2 blocchi corrispondenti alla stessa linea di cache, ad ogni accesso si verifica un **MISS**.
- **ASSOCIATIVE MAPPING** → $k = i$ ovvero ogni blocco può essere memorizzato in ogni linea della cache;
 - o **Vantaggi**: flessibilità;
 - o **Svantaggi**: memoria associativa costosa (accesso tramite contenuto, chiave*).
- **SET ASSOCIATIVE MAPPING** → $k = i|S$, ovvero uguale all'associative ma le linee della cache sono raggruppate in S insiemi (set). Per scegliere in quale linea dell'insieme memorizzare il nuovo blocco:
 - o **LRU** (Least Recently Used) = più usato;
 - o **FIFO** = più economico;
 - o **LFU** (Least Frequently Used) = il migliore, ma complesso da implementare;
 - o **Random** = semplice ed efficiente.

ESERCIZIO ESAME

1. Sistema a processore con memoria di 512 Byte (2⁹ Byte) e una cache direct-mapped da 4 linee (2² linee) da 4 Byte ciascuna (2² Byte). Inizialmente le 4 linee della cache contengono i blocchi 0, 1, 2, 3; dopo gli accessi qui sotto riportati, cosa contengono le 4 linee della cache?

$$\frac{2^9}{2^2} = 2^7 \rightarrow 7 \text{ BIT BUONI (dai 9 di inizio, quindi escludono gli ultimi 2)} \\ \text{DI QUESTI GLI ULTIMI 2 IDENTIFICANO LA LINEA (2^2 LINEE)}$$

ACCESSI
000010101
100010111
000011001
000000000
001110100
001010111
100011001
010111100

→ ACCESSI:

① 0000 101	$2^0 + 2^2 = 5$	→ MISS LINEA 1 → 5
② 1000 101	$2^0 + 2^2 + 2^6 = 69$	→ MISS LINEA 1 → 69
③ 0000 110	$2^1 + 2^2 = 6$	→ MISS LINEA 2 → 6
④ 0000 000	0	→ HIT → 0
⑤ 0011 101	$2^0 + 2^2 + 2^3 + 2^4 = 29$	→ MISS LINEA 1 → 29
⑥ 0010 101	$2^0 + 2^2 + 2^4 = 21$	→ MISS LINEA 1 → 21
⑦ 1000 110	$2^1 + 2^2 + 2^6 = 70$	→ MISS LINEA 2 → 70
⑧ 010 1111	$2^0 + 2^1 + 2^2 + 2^3 + 2^5 = 47$	→ MISS LINEA 3 → 47

QUINDI: [LINEA 0 = 0 ; LINEA 1 = 21 ; LINEA 2 = 70 ; LINEA 3 = 47]

2. Sistema a processore con memoria di 2¹⁶ Byte e una cache set-associative a 4 vie con 16 linee (2⁴ gruppi da 4 linee ciascuno) da 32 Byte (2⁵ Byte) con rimpiazzamento LRU (quindi in ordine). Inizialmente le linee sono vuote; dopo gli accessi qui sotto riportati, cosa contengono le 16 linee della cache?

$$\frac{2^{16}}{2^5} = 2^{11} \rightarrow 11 \text{ BIT BUONI (dai 16 di inizio, quindi escludono gli ultimi 5)} \\ \text{DI QUESTI GLI ULTIMI 2 IDENTIFICANO IL GRUPPO DI LINEE (2^2 GRUPPI)}$$

ACCESSI
0010110000010101
0100100111010101
1111101000101010
0101010110110101
0000111110101010
1111100100101010
0111001100101010

→ ACCESSI:

① 0010 1100 000	352	→ MISS GRUPPO 0 [1° POSTO]
② 0100 100 1110	590	→ MISS GRUPPO 2 [1° POSTO]
③ 1111 1010 001	2001	→ MISS GRUPPO 1 [1° POSTO]
④ 0101 0110001	685	→ MISS GRUPPO 1 [2° POSTO]
⑤ 0101 0111001	125	→ MISS GRUPPO 1 [3° POSTO]
⑥ 01000000001	1883	→ MISS GRUPPO 1 [4° POSTO] → GRUPPO PLENO
⑦ 01010101101	921	→ MISS GRUPPO 1 [GRUPPO PLENO → 1° POSTO] 2001 → 921

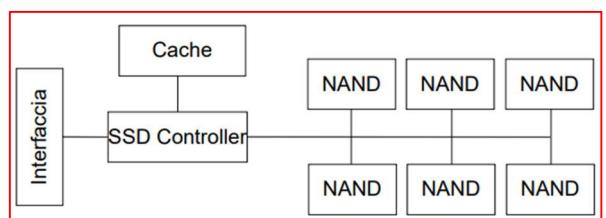
MEMORIA SECONDARIA e OFF-LINE

La **MEMORIA SECONDARIA** (Hard-Disk, SSD, USB...) e la **MEMORIA OFF-LINE** (nastri, cassette e dischi ottici) sono **non volatili, poco costose** e con **alti tempi di accesso**; l'accesso a queste memorie avviene mediante le periferiche connesse al bus. Vediamo queste memorie:

- **MEMORIE A DISCO MAGNETICO** (Hard-Disk) = sul disco magnetico ci sono delle **tracce concentriche** che contengono i dati (tutte le tracce hanno **= n° di bit**, ma **≠ lunghezza** [dunque la densità di bit aumenta andando verso il centro del disco]). Per **accedere ad un dato** sul disco:
 - la **testina** va **sulla traccia**;
 - il **settore** desiderato viene **ruotato** sotto la testina;
 - viene **letto** il **settore** (si usa la memorizzazione a settori perché leggere i dati del settore è più veloce che accedervi).

→ I dati vengono codificati attraverso delle **transazioni di campo magnetico** (in modo da avere 1 transizione = 1 bit e dunque evitare una traccia specifica del disco per la gestione del clock)!

- **SSD (Solid State Drive)** = memoria **non volatile** di tipo **NAND FLASH** (dove posso quindi fare read + write);



- **NASTRI MAGNETICI** (standard LTO);
- **MEMORIE OTTICHE (DISCHI OTTICI)** = grandi costi di accesso, capacità e affidabilità; 2 tipi:
 - **CAV (Constant Angular Velocity)** → come le memorie a disco magnetico, ma accesso sequenziale;
 - **CLV (Constant Linear Velocity)** → **CD** e **DVD**, con un laser che scandisce il disco.

⚠ Le memorie off-line sono molto usate per i **backup** (dove la priorità non è il tempo di accesso, ma l'**affidabilità**)!

MEMORIA VIRTUALE

È un meccanismo per **nascondere alla CPU la reale dimensione della memoria di sistema** (i programmi vengono resi **indipendenti dalla memoria** del sistema), ovvero viene mostrata alla CPU una memoria virtuale più grande di quella disponibile. Questo funziona perché il **dato cercato** dalla CPU viene **spostato dalle memorie secondarie alla principale**. Come funziona?

- CPU genera gli indirizzi logici;
- **MMU (Memory Management Unit)** [che sta **tra la CPU e la CACHE**] **traduce gli indirizzi logici in indirizzi fisici** tramite la **MAT (Memory Address Table)** [che sta **in memoria principale**] {la MAT ha un'entry per ogni page virtuale}:
 - Viene preso l'**indirizzo di base della MAT**;
 - Gli viene **sommato l'offset**, ovvero i **bit bassi dell'indirizzo logico**;
 - L'**indirizzo fisico generato** viene messo **nei bit alti**.
- **Se l'indirizzo cercato è in CACHE** (memoria principale), **MMU produce l'indirizzo fisico richiesto**. **Se invece non è in memoria principale [PAGE FAULT, ovvero MISS]**, l'**MMU** richiede all'**OS** di spostare il blocco dalla **memoria secondaria alla principale**.

⚠ Per **velocizzare l'accesso alla MAT** si usa la **cache della MAT**, detta **TLB** [che sta **in MMU**]!

5) BUS

Interconnette 2 o + dispositivi in maniera condivisa (ovvero il dato sul bus è visibile a tutti i dispositivi connessi) mediante dei **TRANSCEIVER** (**DRIVER** per pilotare le linee del bus [buffer tri-state] + **RECEIVER** per leggere i valori sul bus). Il bus è composto da:

- **DBus** = dati;
- **ABus** = indirizzi;
- **CBus** = controllo.

⚠ In alcuni casi le stesse linee portano segnali di tipo diverso (meno linee, ma circuiteria aggiuntiva), ovvero **BUS MULTIPLEXATI!**

⚠ Per facilitare lo sviluppo di dispositivi compatibili con il bus, ci sono degli standard!

⚠ A seconda dell'architettura, ci sono i **BUS SINGOLI** (1 Bus) e **MULTIPLI** (più Bus [es. bus memoria + bus I/O] connessi tra loro da dei bridge)!

→ L'utilizzo del bus richiede la soluzione di 2 problemi:

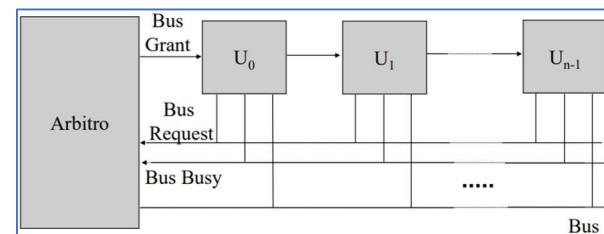
a) **TEMPISTICHE** con cui vengono svolte le operazioni → 2 tipi:

- **BUS SINCRONI** = unità master e slave usano lo stesso clock (il più lento tra i due), parte del bus stesso;
- **BUS ASINCRONI** = ogni comunicazione ha una velocità data dai segnali di controllo ("hand-shaking") [dunque più flessibilità, ma maggiore complessità];
- **SOLUZIONE MISTA** = **BUS SINCRONI** con segnali di **READY** (introduce dei CICLI DI WAIT per fare in modo che lo slave possa richiedere al master più tempo per eseguire l'operazione richiesta).

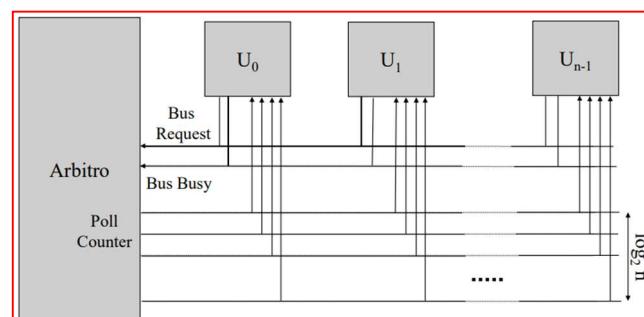
b) **GESTIONE DEI CONFLITTI** nell'accesso al bus (più unità master*) → **ARBITRAGGIO** di tipo:

- **DISTRIBUITO** = ogni modulo attaccato al bus contiene la logica per implementare l'arbitraggio (ogni linea di bus associata ad 1 dispositivo; quando il bus si libera (BUS BUSY inattivo), diventa master quello con priorità massima);
- **CENTRALIZZATO** = c'è un ARBITRO; 3 tipi:

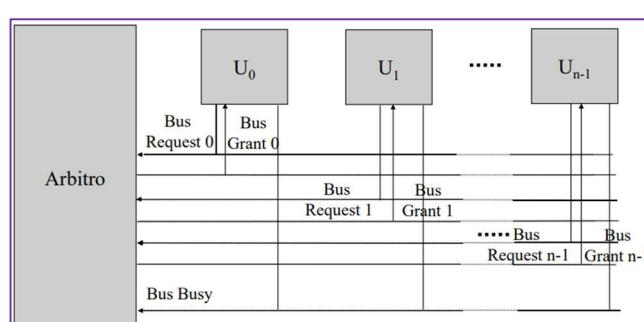
- **DAISY CHAINING** → solo 3 segnali di controllo (meno costoso): BUS BUSY e BUS REQUEST (condivisi, da I/O ad arbitro), BUS GRANT (propagato, da arbitro ad I/O) [data la propagazione del grant la priorità è **statica** e non è **tollerante ai guasti**];



- **POLLING** → BUS REQUEST e BUS GRANT (condivisi) + il grant (che qui si chiama POLL COUNTER) ha $\log(n)$ segnali per n unità [quindi tot segnali = **$2 + \log(n)$** , quindi **costoso**]; è **tollerante ai guasti** e la priorità è **dinamica** (l'arbitro decide la priorità);

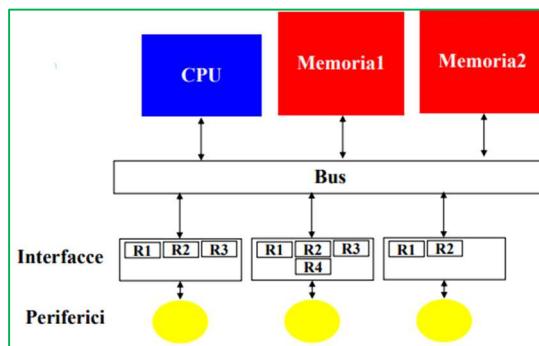


- **RICHIESTE INDIPENDENTI** → BUS REQUEST e BUS GRANT (1 per ogni unità) + BUS BUSY (condiviso). L'arbitro sa già chi fa request perché ogni dispositivo ha la sua linea di request [tot segnali = **$2n + 1$** , quindi è il più **costoso**], quindi priorità **dinamica**.



6) I/O + ECCEZIONI

A ciascun dispositivo di I/O è associata un'**INTERFACCIA** (che lo collega al bus), contenente dei REGISTRI (porte) a cui sono associati degli INDIRIZZI.



Ci sono 2 modi di CONNESSIONE:

- **ISOLATED I/O** → spazio di memoria e I/O **separato** (attivati alternativamente da un **bit di segnale**);
- **MEMORY-MAPPED I/O [MIPS]** → spazio di memoria e I/O **condiviso** (dunque **meno spazio richiesto** e **istruzioni per accedere a memoria = istruzioni per accedere a I/O**).

Dato che velocità CPU ≠ velocità periferiche, la CPU deve SINCRONIZZARSI con essi; 2 metodi:

- **I/O PROGRAMMATO** = CPU gestisce le periferiche e **aspetta le loro risposte** (poco costoso, poco efficiente) [dunque una sorta di I/O sincrono];
- **INTERRUPT** = le periferiche chiamano la CPU mediante un segnale asincrono detto INTERRUPT, in modo che la **CPU smette di fare ciò che sta facendo** (salvando però l'indirizzo del PC e lo stato per ritornare poi a ciò che stava facendo) **e assiste la periferica** [CPU opera solo quando serve, senza aspettare] {inoltre a volte è necessario disabilitare gli interrupt perché la CPU non può fermare quello che sta facendo}. Ma come fa la CPU ad IDENTIFICARE la periferica che ha inviato l'interrupt? 3 modi:
 - LINEE MULTIPLE → ogni dispositivo ha la sua linea di interrupt (per sistemi piccoli);
 - POLLING → 1 solo segnale di interrupt; la CPU scandisce tutte le parole di stato delle periferiche per trovare quella che ha fatto l'interrupt request (lento);
 - INTERRUPT VETTORIZZATO (il più usato e complesso):
 - Periferica manda richiesta all'Interrupt Controller (IC);
 - IC attiva interrupt request verso CPU;
 - Quando è pronta, la CPU invia interrupt acknowledge;
 - IC scrive sul DBus il codice identificativo della periferica;
 - CPU usa il codice per accedere all'Interrupt Vector Table (IVT) e avvia l'ISR (Interrupt Service Routine).

Oltre agli interrupt delle periferiche (INTERRUPT ESTERNI), esistono anche degli INTERRUPT INTERNI, detti **ECCEZIONI** [quindi in grado anch'essi di interrompere l'esecuzione del programma corrente, con procedure collocate nell'IVT]; 3 cause:

- **ECCEZIONI DI ERRORE** (dai circuiti di controllo delle memorie e dalla decodifica delle istruzioni);
- **ECCEZIONI DI DEBUG** (per il debug step-by-step e per i breakpoint);
- **ECCEZIONI DI PRIVILEGIO** (per fare in modo che l'utente non tocchi i dati dell'amministratore).

⚠ Quando si verifica un'eccezione, il controllo passa all'**EXCEPTION HANDLER** (un blocco di codice nel KERNEL SPACE della RAM per gestire le eccezioni) che, finita la gestione dell'eccezione, fa “eret” (return from exception, ovvero carica nel PC l'indirizzo contenuto nell'EPC e ritorna alla user mode) oppure “exit” (li \$v0, 10 + syscall). Inoltre, l'exception handler ha riservati i registri **\$k0** e **\$k1** e usa i dati nella “**.kdata**” (kernel data)!

Vediamo ora la **GESTIONE DELLE ECCEZIONI NEL MIPS**: definendo eccezioni sincrone (generate dal programma in esecuzione, es. syscall) e asincrone (imprevedibili), distinguiamo 3 tipi di eccezioni:

- **INTERRUPT HARDWARE** (asincrona, generata da periferica) → dato che il MIPS è **MEMORY-MAPPED con INTERRUPT A LINEE MULTIPLE** [solo 6 dispositivi di I/O], ci sono 6 linee con 3 segnali:
 - **Reset** = reset del sistema;
 - **Non-Maskable Interrupt** = problemi HW seri;
 - **IRQ_0 - IRQ_5** = inviati dalle 6 periferiche (non mascherabili).
- **INTERRUPT SOFTWARE** (sincrona, generata nell'esecuzione del programma [es. overflow]);
- **TRAP** (sincrona, provoca il controllo dell'OS [ovvero portano la CPU in **KERNEL MODE**]);
 - **Syscall** = chiama l'OS;
 - **Break** = breakpoint per il debugger.

⚠ Ci sono anche delle "istruzioni di trap" che fanno andare la CPU in kernel mode (come **teq \$t0, \$t1** [trap se $t0 = t1$], **tne \$t0, \$t1** [trap se $t0 \neq t1$] e **tlt \$t0, \$t1** [trap se $t0 < t1$])!

⚠ Ma cos'è la **KERNEL MODE**? Il MIPS ha 2 modalità operative (specificate da 1 bit detto **CPU MODE** [**KERNEL** = 1, **USER** = 0]) e quindi la memoria divisa in 2 parti:

- **KERNEL** (Supervisor) → CPU esegue l'OS (boot del dispositivo + eccezioni); si trova nel **KERNEL SPACE [80000000, FFFFFFFF]**;
- **USER** → CPU esegue programma utente; si trova nello **USER SPACE [00000000, 7FFFFFFF]**.

→ La **GESTIONE DELLE ECCEZIONI IN HW** non è della CPU, ma del **COPROCESSORE '0' (CP0)** [accessibile solo in kernel mode]; questo contiene dei registri utili:

- **EPC** (Exception Program Counter) = salva indirizzo di return dopo la gestione dell'eccezione;
- **Cause** = salva interrupt in sospeso e codifica la causa dell'eccezione;
- **BadVAddr** = salva l'indirizzo che ha causato l'eccezione;
- **Status** = salva l'eccezione in corso (+ usato per mascherare gli interrupt);
- **Count** (incrementato ogni 10 ms) + **Compare** (compara valori) = timer interrupt.

⚠ Ci sono anche delle "istruzioni" per comunicare con il CP0 (come **mfc0** [lettura da CP0], **mct0** [scrittura in CP0], **lwc0** [carica valore in CP0] e **swc0** [salva valore da CP0])!

⚠ Nel MIPS, ogni porta I/O è controllata da 2 registri, ovvero **TRIX** (ci dice se la porta è in input o output) e **PORT** (tiene il valore in input o output)!

Per trasferire grosse moli di dati tra memoria e periferica (es. unità disco rigido), si usa il **DIRECT MEMORY ACCESS (DMA)** gestito dal **DMA Controller (DMAC)** [che quindi fa da master del bus].

→ Per la connessione tra DMAC e i vari dispositivi I/O abbiamo alcuni registri del DMAC:

- **IOAR** = contiene l'indirizzo di partenza del blocco;
- **DC** = contiene il n° di byte (o word) da trasferire;
- **Controllo** = capire se i dati vanno letti o scritti.

→ Ci sono 3 MODALITÀ DI TRASFERIMENTO:

- **BURST TRANSFER** → DMAC controlla il bus fino a trasferimento concluso (CPU bloccata, velocità massima);
- **CYCLE STEALING** → blocchi spaccettati per bloccare per meno tempo la CPU (velocità minore);
- **TRANSPARENT DMA** → DMAC usa il bus solo quando la CPU non lo sta usando (velocità minima).

⚠ Il funzionamento del DMAC è:

- CPU programma il DMAC all'inizio riempiendo i 3 registri del DMAC;
- DMAC riceve richiesta di trasferimento da periferica;
- DMAC invia DMA Request alla CPU;

- CPU rilascia bus e attiva DMA Acknowledge;
- DMAC inizia il trasferimento e aggiorna IOAR e DC dopo ogni parola trasferita;
- DC arriva a zero, trasferimento termina;
- DMAC invia interrupt alla CPU.

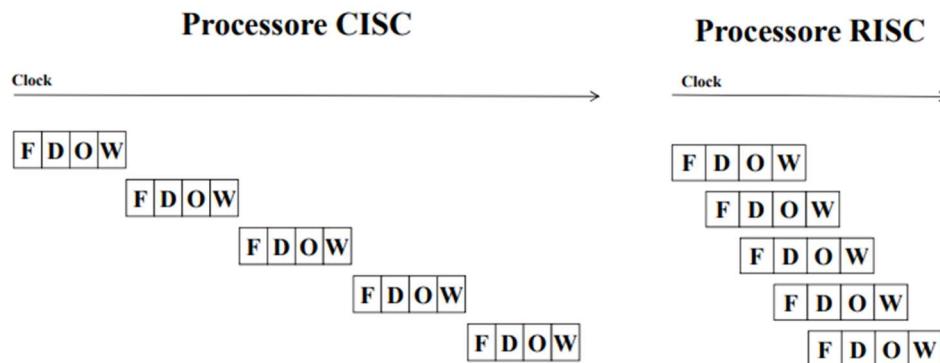
7) PIPELINE (RISC e SUPERSCALARI)

Definendo CPI (Clocks Per Instruction, ovvero il THROUGHPUT), i processori possono essere:

- **CISC** (Complex Instruction Set Computer) → CPI > 1;
- **RISC** (Reduced Instruction Set Computer) → CPI = 1 [ovvero il MIPS];
- **SUPERSCALARI** → CPI < 1.

I RISC e i SUPERSCALARI hanno CPI basso proprio per l'**ARCHITETTURA A PIPELINE** (equivalente elettronico della catena di montaggio), ovvero le istruzioni vengono spacchettate in fasi (stadi) e il processore è diviso in parti che lavorano in parallelo su una specifica fase. Le condizioni per la pipeline sono:

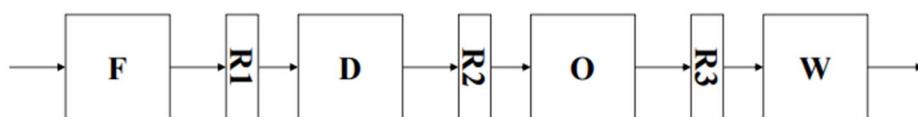
- ❖ Stessi stadi (fasi) per ogni istruzione;
- ❖ Stesso tempo per ogni stadio.



⚠ La pipeline viene infatti usata sui RISC perché le istruzioni devono essere tutte semplici e simil-spacchettabili. Infatti devo disporre di tanti registri (per i registri di pipeline), devo avere 2 cache (dati e codice) e UC cablata!

⚠ Il MIPS ha **5 stadi** (fasi): Fetch [F] + Decode [D] + Operate [O] + Write [W] + Memory (Load/Store) [M]!

Ogni parte della CPU che esegue uno stadio si può vedere come un blocco di LC (Logica Combinatoria) con 1 registro in input e 1 registro in output (**REGISTRI DI PIPELINE**, che servono per mantenere il valore da passare tra le varie parti, dunque con lo stesso clock).



→ A volte però, dopo 1 periodo di clock, uno stadio non è ancora terminato e dunque gli altri devono aspettare (ovvero si verifica uno **STALLO**, per esempio a causa di un miss della cache o per la lentezza di una specifica operate). Questo problema si risolve usando dei **BUFFER** al posto dei registri (vengono salvati più dati nel buffer e man mano che lo stadio va avanti, i dati vengono scalati e passati [**CODA DI ISTRUZIONI**]).

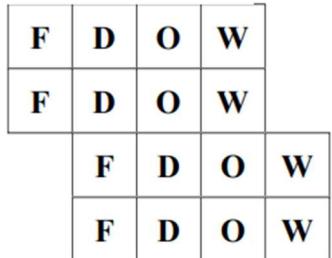
Un altro caso che può generare uno stallo è dovuto alle **DIPENDENZE DI DATO** (es. add R3, R1, R2 con subito a seguire un'operazione su R3; dunque la fase di Decode [lettura di R3] della seconda operazione precede la fase di Write [scrittura su R3 del risultato] della add precedente); queste dipendenze si possono risolvere con:

- SOLUZIONE HARDWARE = la CPU manda in STALLO il sistema per evitare il problema;
- SOLUZIONE SOFTWARE = il COMPILATORE inserisce dei NOP tra le 2 istruzioni problematiche (per ritardare la Decode della seconda istruzione, che dipende dalla prima) [meno portabilità del programma].

⚠ Anche per le **ISTRUZIONI DI SALTO** ci sono 2 soluzioni:

- **SOLUZIONE HARDWARE** = CPU svuota dalla pipeline le istruzioni caricate (+ **PREDIZIONE DI SALTO** = tecnica hardware dove l'unità di fetch fa previsioni sui salti condizionati → se la predizione è errata, tutte le istruzioni caricate dopo vengo abortite);
- **SOLUZIONE SOFTWARE** = COMPILATORE inserisce dei NOP.

→ Oltre ai RISC ci sono i **SUPERSCALARIS** (**CPI < 1**), che hanno pipeline dove **1 stadio = \oplus Unità** che lo gestiscono (dunque abbiamo più stadi uguali gestiti in contemporanea, cioè maggior INSTRUCTION LEVEL PARALLELISM), per esempio spaccando la mia operate in FPU (float) e ALU (interi) con davanti un'unità di smistamento.



⚠ Legati a questi processori, troviamo i **PROCESSORI**:

- MULTITHREAD → oltre alle istruzioni (RISC), partizioniamo anche parti del programma (**THREAD**);
- MULTICORE → processori contenenti più CPU (**CORE**); dunque alte prestazioni e bassi consumi.