

Funzioni e Reti Logiche

Funzioni circuitali

- I circuiti elettronici non sono in grado di svolgere operazioni complesse o algebriche
- Le funzioni base che si riescono a svolgere con logica basata su commutatori bistabili (transistor) sono operazioni logiche (AND, OR, EX-OR) tra bit
- Dobbiamo imparare a maneggiare le funzioni logiche
- Per semplicità grafica useremo i simboli algebrici (+) al posto di OR e (\cdot) al posto di AND, se non ci sono ambiguità (\cdot) può venire omesso

Operatori logici

- AND (\cdot) – ha le stesse proprietà della moltiplicazione nell'algebra convenzionale, in particolare $1 \cdot X = X$
- OR (+) – ha le stesse proprietà della somma nell'algebra convenzionale, inoltre $1+X = 1$, da cui deriva ovviamente $1+1 = 1$
- Negazione (\bar{x}) – inverte il valore della variabile
- EX-OR (\oplus) – può essere espresso come una somma di prodotti $x \oplus y = x\bar{y} + y\bar{x}$ e assume valore 1 quando uno solo degli ingressi è uno

Funzioni logiche

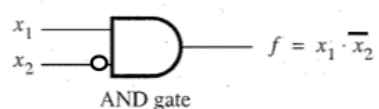
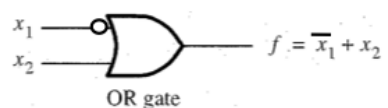
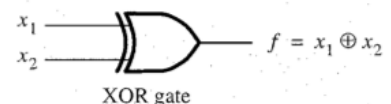
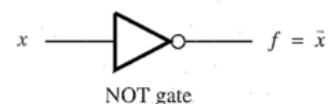
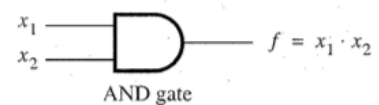
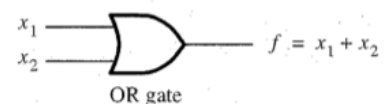
- Le funzioni logiche sono combinazioni di AND, OR, EX-OR e negazioni di variabili logiche
- Una combinazione logica (funzione) di variabili x_i porta alla definizione di una *mappa di verità*

Porte logiche

- Le operazioni logiche elementari vengono svolte da circuiti elettronici realizzati appositamente per effettuare la funzione, che vengono in genere chiamate "porte" o "gate" in inglese
- Ogni funzione ha una sua porta ... porta AND, porta OR, ...
- L'uscita della porta ha il valore della funzione logica implementata
- Ai fini della sintesi di un circuito che realizza una funzione logica è uso dare una rappresentazione grafica delle porte
- Le porte possono essere collegate tra loro per realizzare reti logiche, che sono la base di tutti i circuiti, siano essi integrati o a componenti discreti

Negazione degli ingressi

- Le porte logiche possono anche avere più di 2 ingressi
- è uso rappresentare gli ingressi negati con un semplice pallino sull'ingresso anziché con la porta NOT



Mappe di verità

- Una mappa di verità contiene tante righe quante sono le possibili combinazioni delle variabili, cioè 2^n dove n è il numero di variabili.
- Ciascuna riga della colonna corrisponde per cui la funzione vale 1 corrisponde all'AND logico di una data combinazione delle variabili o delle variabili negate
- La funzione logica complessiva è definita dall'OR logico di tutte le combinazioni che danno per risultato 1
- La mappa di verità definisce una funzione somma (OR) di prodotti (AND)

Sintesi di reti logiche

- Una rete logica può essere sintetizzata direttamente dalla tavola di verità, ovvero dalla sua forma "somma di prodotti"
- Risulta una rete a due stadi, il primo di porte AND, il secondo con una sola porta OR

Costo di una rete logica

- Per costo di una rete logica si intende normalmente la somma del numero di porte e del numero di ingressi della rete (indipendentemente dal fatto che siano positivi o negati)

Minimizzazione di funzioni logiche

- Data una funzione logica esistono molte reti logiche diverse per realizzarla ... non tutte equivalenti dal punto di vista del costo
- Dalla tavola di verità si ottiene una rete logica a due stadi, non necessariamente quella a costo minimo
- Una espressione logica si dice minima quando il costo della rete logica che la realizza è minimo
- È possibile che una rete minima abbia più di due stadi, cioè l'espressione corrispondente non è in forma somma-di-prodotti
- Le configurazioni con più di due stadi hanno applicazioni limitate perché introducono un ritardo maggiore nell'elaborazione (propagazione dei segnali dagli ingressi delle porte alle uscite delle porte e quindi agli ingressi dello stadio successivo)

Identità e regole algebriche booleane

Nome	OR (+)	AND (•)
identità o idempotenza	$x + x = x$	$xx = x$
commutativa	$x + y = y + x$	$xy = yx$
associativa	$(x + y)z = x + (y + z)$	$(xy)z = x(yz)$
distributiva	$x + yz = (x + y)(x + z)$	$x(y + z) = xy + xz$
complemento	$x + \bar{x} = 1$	$x\bar{x} = 0$
De Morgan	$\overline{x+y} = \bar{x}\cdot\bar{y}$	$\overline{x\cdot y} = \bar{x} + \bar{y}$
	$1 + x = 1$	$0 \cdot x = 0$
	$0 + x = x$	$1 \cdot x = x$

Minimizzazione di funzioni logiche

- La minimizzazione di alcune espressioni logiche è banale, in altri casi è necessario applicare le regole definite prima in modo "furbo"

Minimizzazione metodo "grafico"

- La minimizzazione algebrica di funzioni logiche non è un processo semplice e diretto
- Per poche variabili è possibile usare una forma particolare delle mappe di verità che consente una minimizzazione "a vista" della rete implementativa molto semplice
- Mappe di Karnaugh

Mappe di Karnaugh (Mdk)

- Una Mdk è una tabella a due dimensioni che riporta le variabili logiche come indici delle righe e delle colonne e come valore nelle caselle il valore della funzione
- L'idea di base per la minimizzazione con le Mdk è sfruttare l'adiacenza dei valori positivi, che indicano una possibile fattorizzazione o aggregazione dei termini moltiplicativi della tavola di verità

	A	0	1
B	0	0	0
	1	0	1

Mdk a 3 variabili

- è una tabella di 8 elementi
- In genere si mettono 2 variabili nelle colonne
- Si noti la codifica delle coppie di bit tipo "gray"
- $f = x_1x_2 + x_1x_2x_3 + x_1x_3$

		x_1x_2			
		00	01	11	10
x_3	0				
	1				

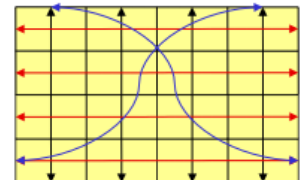
Mdk a 4, 5, 6, ... variabili

- Con 4 variabili è una tabella di 16 elementi
- Con 5 di 32
- Con 6 di 64
-
- Si noti ancora la codifica delle coppie di bit tipo "gray" che è fondamentale per la minimizzazione

		x_1x_2			
		00	01	11	10
x_3x_4	00				
	01				
	11				
	10				

Uso delle Mdk

- L'idea chiave è che "1" adiacenti nelle Mdk corrispondono a termini che differiscono per una sola variabile e indicano quindi una possibile semplificazione algebrica
- Una Mdk, disegnata sul piano, va interpretata come ripiegata su un toroide \Rightarrow le celle sui bordi sono adiacenti a quelle sul bordo opposto
- Raggruppando gli "1" adiacenti si trovano dei termini prodotto



		x_1x_2			
		00	01	11	10
x_3	0	0	0	1	0
	1	0	0	1	1

- Più è grande il gruppo meno variabili ci sono nel termine

	x_1x_2			
x_3	00	01	11	10
0	0	0	1	1
1	0	0	1	1

x_1

- Lo stesso "1" può far parte di diversi gruppi

	x_1x_2			
x_3	00	01	11	10
0	0	0	1	0
1	0	0	1	1

x_1x_3

x_1x_2

Condizioni di "don't care"

- Esistono situazioni in cui il valore di una variabile o di un gruppo di variabili non ha importanza per una funzione, cioè può assumere un valore "d".
Es: ($x=d$, $xyz=d$)
- Le MdK consentono di tenere conto di queste situazioni, perchè si può assegnare le caselle "d" a gruppi di "1", che generano un termine nella funzione oppure lasciarle a "0"

	x_1x_2			
x_3x_4	00	01	11	10
00	0	0	d	0
01	0	0	d	1
11	1	0	d	d
10	0	1	d	d

x_1x_4

$x_2x_3\bar{x}_4$

$\bar{x}_2x_3x_4$

Altre porte logiche

- Qualsiasi funzione logica si può realizzare con porte AND e OR
- Esistono però molti altri tipi di porte logiche: l'EX-OR ne è un esempio
- Nella pratica ingegneristica le porte più usate sono le porte NAND e NOR, cioè delle funzioni AND e OR con negazione dell'uscita
- La ragione è la facile realizzazione in tecnologia c-mos delle suddette porte
- In generale i circuiti integrati mettono a disposizione del progettista solo due tipi di porte, una per "moltiplicare" e una per "sommare"
- Qualsiasi funzione realizzabile con AND e OR può essere realizzata con NAND e NOR ... solo che la "logica inversa" è un po' meno intuitiva



(a) NAND



(b) NOR

Reti con reazione e memoria

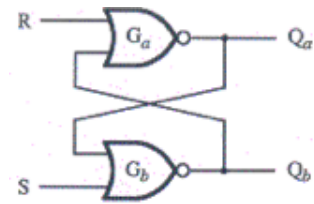
- Le funzioni logiche e le relative reti di implementazione visto fino ad ora sono note come "reti combinatorie"
- Le reti combinatorie non hanno una nozione "esplicita" del tempo e non hanno memoria del passato: in ogni istante di tempo l'uscita dipende solamente dagli ingressi nell'istante considerato
- In molte applicazioni è necessario introdurre memoria nel sistema ...
- In realtà si dà sempre per scontato che un elaboratore sia in grado di memorizzare informazioni
- La memoria in una rete logica si ottiene con una "reazione" cioè alimentando l'uscita di alcune porte sugli ingressi di porte del medesimo stadio in modo da formare un anello in cui gli ingressi dipendono dalle uscite (e viceversa)
- La reazione complica in modo significativo l'analisi e la sintesi di una rete

logica

- La memoria deriva dal fatto che gli ingressi "ricordano" il passato della rete attraverso il valore delle uscite passate

Elemento base di memoria (latch)

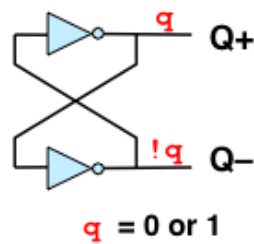
- Realizzazione con due porte NOR e schema di "temporizzazione" della tavola di verità



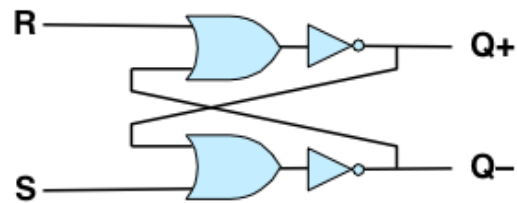
S	R	Q ^a	Q ^b
0	0	0/1	1/0 (No change)
0	1	0	1
1	0	1	0
1	1	0	0

Memorizzare un bit

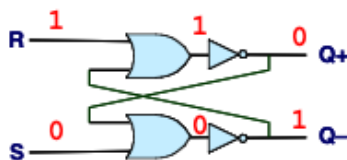
Elemento Bistabile



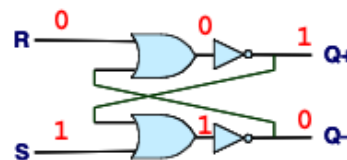
R-S Latch



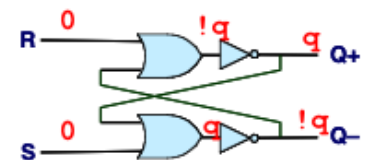
Reset



Set

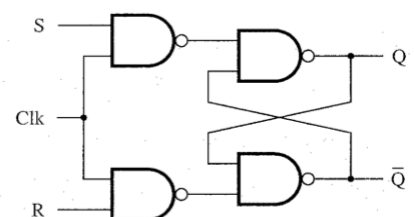
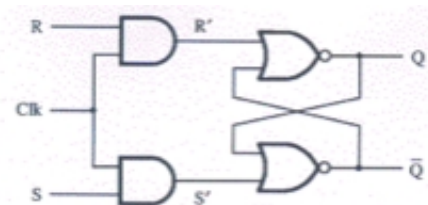


Memorizzare



Stati indecidibili e temporizzazione

- Dato che i segnali non si propagano in tempo nullo, l'effetto del cambio di un ingresso si propaga in tempo finito sulle uscite
- Se le uscite sono reazionate questo può creare problemi di indecidibilità dello stato di una rete con memoria
- Gli elementi di memoria sono quindi sempre temporizzati, cioè sono governati da un segnale speciale chiamato "clock"
- Un elemento base di memoria temporizzato viene normalmente indicato come "gated latch"
- Il clock viene inserito come "ingresso di abilitazione" attraverso porte AND: se ck è a zero la



rete reazionata ha gli ingressi forzati a zero e non può cambiare stato

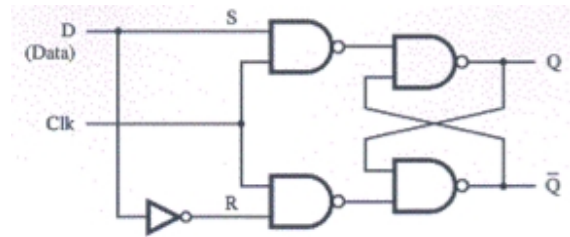
- Quando ck è a uno la gli ingressi della rete reazionata sono gli ingressi R ed S del circuito
- Circuiti di questo tipo hanno rappresentazione grafiche "standard"

Elementi di memoria "reali" celle D e flip-flop

- Le reti viste prima sono note come latch S-R (Set-Reset)
- Hanno il difetto di avere uno stato indecidibile (cioè l'uscita non può essere nota con certezza) quanto entrambi gli ingressi sono a uno
- In molti casi questo è inaccettabile
- Si può rimediare con latch-D (data) e flip-flop

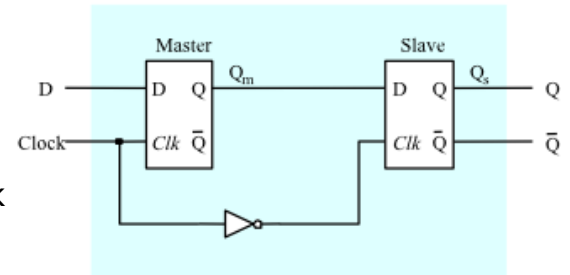
Latch tipo "D"

- Gli ingressi al circuito base sono ottenuti da una unica variabile
- Non vi può essere ambiguità
- Il circuito è abilitato durante tutta la fase positiva del clock



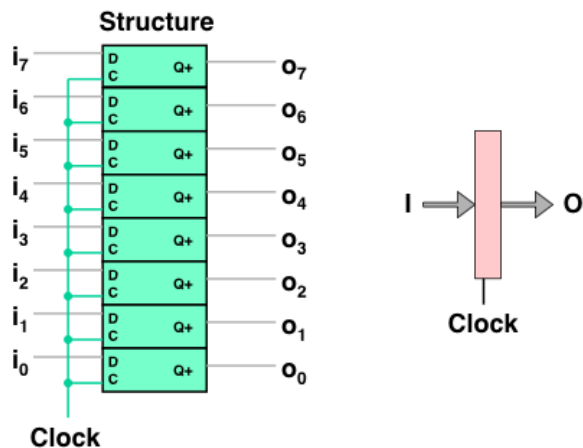
Flip-flop master-slave

- Configurazioni più complesse (come questa) consentono ad esempio di ottenere che l'uscita del circuito commuti esattamente al termine dell'impulso di clock



Registri

- Impiegati per registrare delle word di dati
- Collezione di latch edge-triggered
- Caricano gli input sul fronte in salita del clock



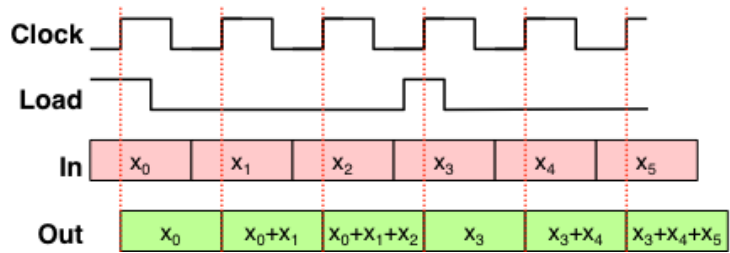
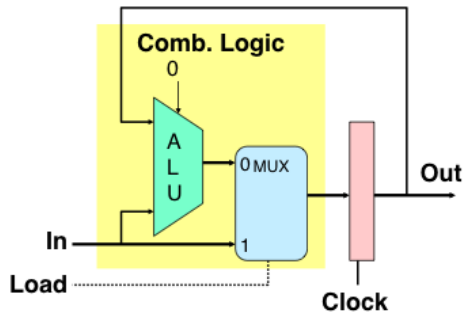
Operazioni su registri

- Memorizzano bit
- La maggior parte delle volte operano come una barriera tra input e output
- Sul fronte in salita del clock memorizzano l'input



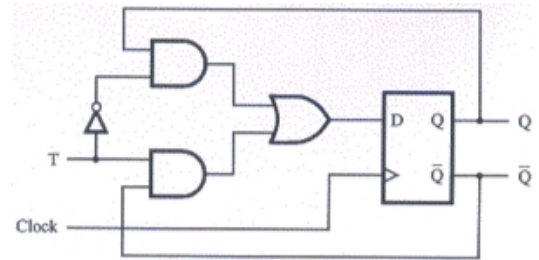
Esempio di macchina a stati

- Circuito accumulatore
- A ogni ciclo carica l'input e lo accumula



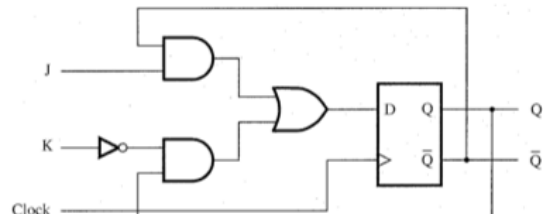
Flip-flop tipo "T"

- "T" sta per "toggle" cioè "oscillatore"
- Commuta lo stato dell'uscita ad ogni colpo di clock se l'ingresso è positivo
- Molto usato per realizzare contatori



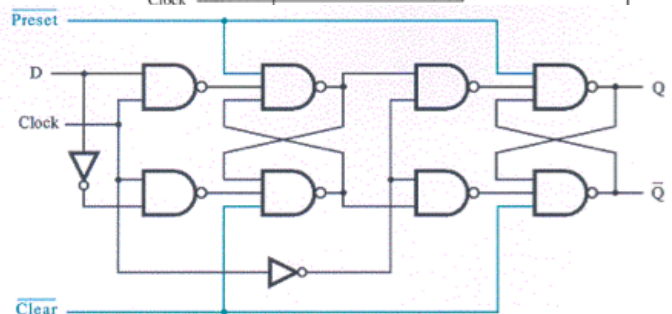
Flip-flop "J-K"

- Combina le proprietà dei flip-flop di tipo T con i latch S-R
- è un circuito versatile perchè può memorizzare dati o realizzare contatori



Flip-flop master-slave con preset e clear

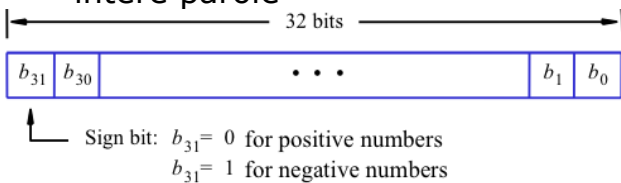
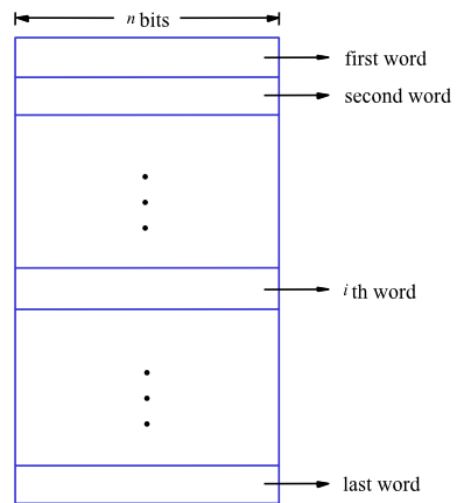
- Consente di "forzare" delle particolari configurazioni sulle uscite (zero o uno) indipendentemente dalla "storia" del circuito



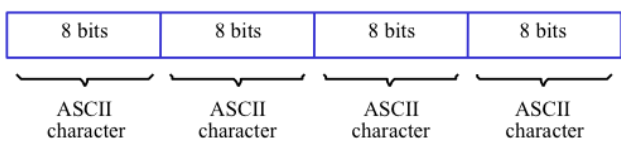
Istruzioni, Programmi e Indirizzamento Linguaggi "Assembler"

Locazioni di Memoria e Indirizzi

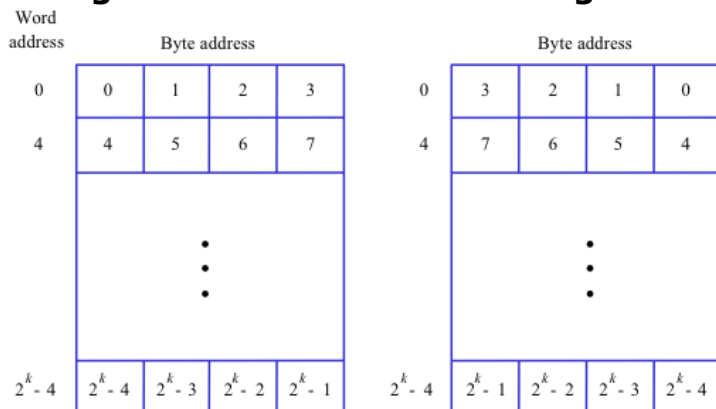
- Negli elaboratori la "minima" unità di dati è in genere il byte (ottetto)
- I byte sono organizzati in word (parole) la cui lunghezza è tipicamente 2^n byte, $n=0,1,2,3,4$
- Per convenzione si usa un significato modificato dei prefissi k, M, G, T:
 - k (kilo) = 2^{10} = (1024)
 - M (Mega) = 2^{20} = (1 048 576)
 - G (Giga) = 2^{30} = (1 071 741 824)
 - T (Tera) = 2^{40} = ...
- Ogni locazione di memoria deve essere identificabile \Rightarrow indirizzi
- Il numero di locazioni indirizzabile è noto come spazio di indirizzamento
- Un elaboratore può essere indirizzabile
 - **"al byte"** : la minima unità di dati indirizzabile è un byte oppure
 - **"alla parola"** : la minima unità di dati indirizzabile è una parola di 2^n byte; in questo caso non sarò mai in gradi di leggere o scrivere un singolo byte in memoria, ma solo intere parole



intero con segno



Assegnazione Little-Endian e Big-Endian



(a) Big-endian assignment

(b) Little-endian assignment

Allineamento degli indirizzi

- Un indirizzo che identifica l'inizio di una parola di memoria si dice "allineato"
- Un indirizzo che identifica una word non all'inizio di una parola di memoria si dice "non-allineato"
- Quasi tutti gli elaboratori moderni **non** consentono l'accesso di parole a indirizzi non allineati
- L'uso di un indirizzamento allineato
 - velocizza le operazioni di accesso in memoria
 - spreca memoria se devo memorizzare byte singoli
(in pratica nessun elaboratore consente la memorizzazione di singoli bit)

Numeri, caratteri, istruzioni

- Un numero viene memorizzato su una o più parole
- Un carattere ASCII isolato viene memorizzato su una parola
- Una stringa di j caratteri viene memorizzata su $\lceil j/n \rceil$ parole (n numero di byte per parola)
- Una istruzione (comando dell'elaboratore) viene memorizzata su una (o più) parole
- Numeri, caratteri e istruzioni sono indistinguibili se non so il "tipo" di dato che contengono

Istruzioni e notazione

- Per semplicità noi usiamo dei nomi o simboli mnemonici per le istruzioni, evitando la possibile confusione istruzioni/dati
- Esistono diversi tipi di notazione per identificare le operazioni
 - RTN (Register Transfer Notation): più vicina al modo di operare dell'Hardware
 - ALN (Assembler Language Notation): più vicina al modo di operare del software di alto livello
- Ogni elaboratore ha il suo assembler, ma è possibile "ragionare" su un linguaggio "generico"...
- RTN
 - $R1 \leftarrow [LOC]$
assegna al registro R1 il contenuto della locazione di memoria LOC
 - $R3 \leftarrow [R1] + [R2]$
assegna al registro R3 la somma dei contenuti di R1 ed R2

Si noti che la parte destra di una RTN denota sempre un valore da assegnare a una locazione

- ALN
 - Move LOC,R1
 - Add R1,R2,R3

Istruzioni e operandi

- Una istruzione elementare di un elaboratore può avere 1, 2, o 3 operandi
 - Gli operandi possono essere sorgenti o destinazioni
Move A , R1
(A sorgente, R1 destinazione)

- Istruzioni ad 1 operando danno per scontata la presenza di un registro "speciale" chiamato accumulatore

```
Load A
Add B
```

carica A nell'accumulatore e gli somma B (sempre nell'accumulatore)

Istruzioni e operandi Intel

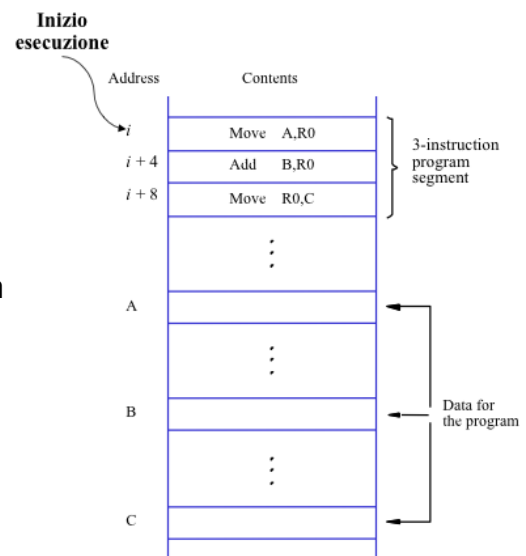
- Nell'assembler Intel (sintassi AT&T) si usa distinguere anche se l'istruzione opera su byte (b), su word a 16bit (w) o su word a 32 bit (l) e si usa % prima di ogni registro

– Es.

- movb %Ax, Dest (opera sul byte)
- movl %EAX, Dest (opera su word)
- Come accumulatore viene spesso usato EAX (32bit)
 - E' possibile usarne solo 16bit specificando AX:
 - AX a sua volta può essere partizionato
 - AL (8 bit meno significativi)
 - AH (8 bit più significativi)

Esecuzione delle Istruzioni

- Gli elaboratori sono macchine sequenziali
- L'esecuzione "naturale" delle istruzioni è "una-dopo-l'altra"
- Un programma viene caricato nel processore una istruzione alla volta nell'IR
- Il processore esegue l'istruzione e poi carica la successiva a cui punta il PC
- La fase iniziale si chiama "instruction fetch" (o load, o caricamento)
- La seconda fase di chiama "instruction execution" (o esecuzione)



Metodi di indirizzamento

- Implicitamente abbiamo già usato due diversi modi per indirizzare la memoria
 - modalità assoluta
 - modalità registro
- Il problema di effettuare cicli nel codice rimanda al problema di "generare" indirizzi e di come interpretarli
- Ci sono almeno una decina di modi diversi per generare indirizzi e usarli nelle istruzioni e questo ha una grossa influenza sull'organizzazione delle strutture dati usate per la programmazione

Indirizzamento in modo registro

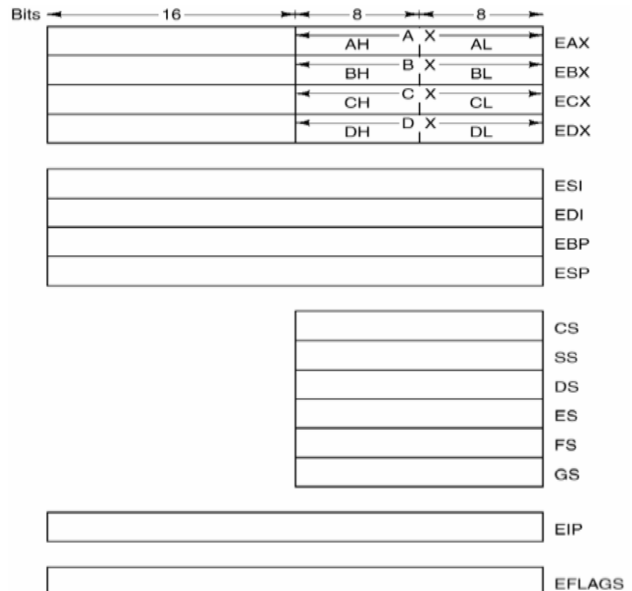
- L'operando è già contenuto in un registro del processore, che viene identificato con il nome standard "Ri"
- Es.


```
Move R1,R2
```
- In assembler INTEL questo indirizzamento è presente

- Es.
- I registri principali per uso generale sono:
`Movl %eax, %ebp`
`eax, ebx, ecx, edx, esi,`
`edi, ebp`

Registri IA32

- **EAX, EBX, ECX, EDX**: general purpose
 - **EAX**: accumulatore
 - **EBX**: puntatore a memoria
 - **ECX**: controllo cicli
 - **EDX**: estende EAX per operazioni a 64bit
- **ESI, EDI**: puntatori a stringhe
- **EBP**: puntatore alla testa del frame di attivazione
- **ESP**: puntatore alla testa della pila
- **EIP**: instruction pointer
- **CS, ..., GS**: selettori di segmento
- **EFLAGS**: registro con i flag di stato



Indirizzamento Immediato

- L'operando è una costante numerica e se ne usa il suo valore effettivo
- Il valore della costante è in genere preceduto da un diesis (#)
- Es.
`Move #A,R1`
`Add #3,R0`

Indirizzamento Immediato Intel

- Per l'Intel (sintassi AT&T), il valore della costante è in preceduto da un \$
- Es.
`Move $1, %EAX`
`Add $2, %EAX`

Indirizzamento Assoluto o Diretto

- L'operando contenuto in una locazione di memoria "LOC". LOC è una costante numerica
- Es.
`Move LOC,R2`
`Move A,B`
`Add A,B,R0`

Indirizzamento indiretto e puntatori

- L'operando è contenuto in una locazione di memoria il cui indirizzo è contenuto in un registro o in un'altra locazione
- L'operando è indicato tra parentesi tonde ()
- L'operando che indica un indirizzamento indiretto è detto "puntatore"

- Es.
 Move (LOC),R2
 Move (R1),(R2)
 Add (A),(R1),R0

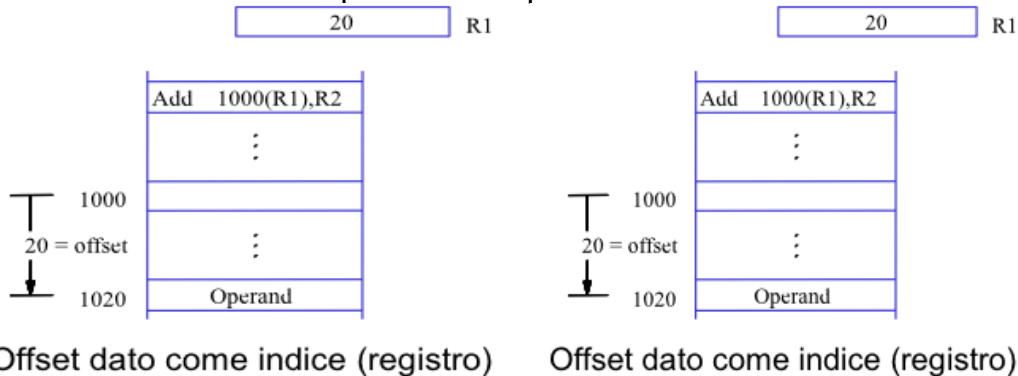
Indirizzamento Indicizzato

- L'operando è ottenuto sommando una costante (o il valore di un altro registro/localazione di memoria) ad un registro/localazione di memoria
- Normalmente si usa solo con i registri

- Es.

Move X(R1),R2 l'indirizzo effettivo è $X+[R1]$
 Add #5,A(B),R0 l'indirizzo effettivo è $A+[B]$

- Il valore che precede il registro è detto offset o spostamento
- Ovviamente esistono sempre 2 modi per ottenere lo stesso indirizzo



Indirizzamento Indicizzato

- L'indirizzamento indicizzato può essere effettuato rispetto al contenuto di un altro registro che viene chiamato base oppure in entrambi i modi

- Es.

(R1,R2) l'indirizzo effettivo è $[R1]+[R2]$
 X(R1,R2) l'indirizzo effettivo è $X+[R1]+[R2]$

Load #27(R1,R2),R0

Indirizzamento Relativo

- L'operando contenuto in una locazione di memoria definita a partire dal Program Counter "X(PC)"

- Es.

Load #20(PC),R1

- L'uso più tipico è per caricare istruzioni di salto nell'IR

Load #A(PC),IR

- Questa è la "traduzione" della più comune istruzione Branch LABEL

Indirizzamento Intel

- Nell'architettura intel tutte le modalità di indirizzamento sono sottocasi di:

$ind = base + (index * scale) + displacement$ dove

- base è uno dei registri eax, ebx, ecx, edx, esp, ebp, esi e edi
- index può essere uno dei precedenti (eccetto esp)
- scale può essere (1, 2, 4, o 8)

- displacement è un valore immediato a 8, 16 o 32 bit
- Ad esempio
 - `movl 10(%ebx,%ecx,2), %eax` porta in `eax` il contenuto di memoria in `ebx+2*ecx+10`

- Esempi

Global Symbol	MOVL x, %eax
Immediate	MOVL \$56, %eax
Register	MOVL %ebx, %eax
Indirect	MOVL (%esp), %eax
Base-Relative	MOVL -4(%ebp), %eax
Offset-Scaled-Base-Relative	MOVL -12(%esi, %ebx, 4), %eax

Indirizzamento indiretto e indicizzato - Formato assembler

- Per operare correttamente bisogna definire un formato univoco del linguaggio assembler
 - etichette (label), possono avere un nome logico oltre al valore numerico
 - istruzione (operation – instruction)
 - operandi (operands)
 - commenti (comments)

label	i-code	op1,op2,op3	...
-------	--------	-------------	-----

- Sintassi gcc per Intel

<code>.pippo:</code>	<code>movl</code>	<code>\$27,%eax</code>	<code>/*loads the number 27 in the accumulator*/</code>
	<code>addl</code>	<code>A,%eax</code>	<code>/*adds the content of memory location "A" in the accumulator*/</code>
	<code>movl</code>	<code>%eax,B</code>	<code>/*store the result into B*/</code>

Altre istruzioni assembler...

- Istruzioni logiche sui singoli bit di una parola

Not	dst	bitwise complement "dst", that may be a register or a memory location (mloc)
And	dst1, dst2	bitwise "ANDs" the content of dst1/2, that may be registers, mlocs or constants
Or	dst1, dst2	bitwise "ORs" the content of dst1/2, that may be registers, mlocs or constants
Comp	dst1, dst2	compares the content of dst1/2, that may be registers, mlocs or constants, returns "0" if <code>dst1=dst2</code>

- Istruzioni algebriche

Neg	dst	trasforma "dst" in "-dst"
Mul	dst1, dst2	calcola la moltiplicazione tra dst1/2 supposti

numeri interi, e memorizza il risultato in dst2, il cui contenuto viene distrutto; resta il problema che il risultato può andare in overflow, problema peraltro comune anche ad Add

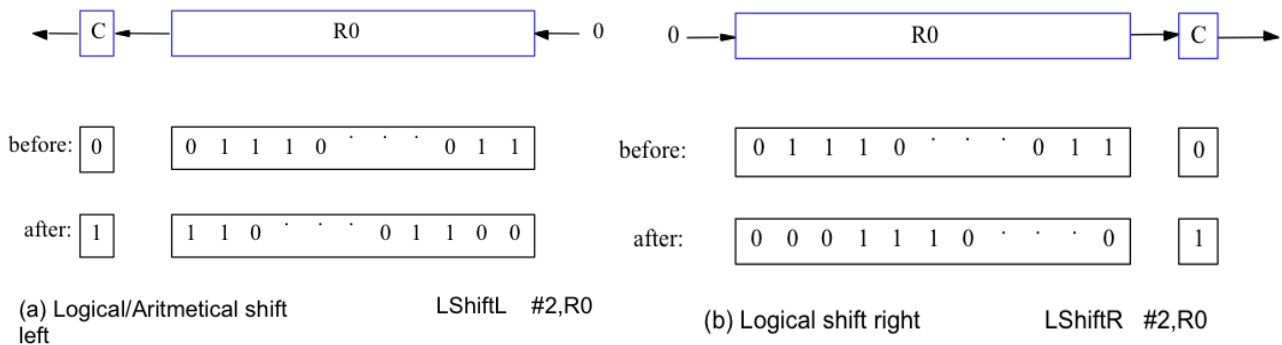
- In generale non vi sono operazioni "base" (primitive) per numeri FP, nè primitive di divisione a meno che il processore non abbia una ALU che contiene circuiti dedicati alle operazioni complesse
- Sub può essere realizzata da Neg e Add in sequenza
- Istruzioni di Shift logiche

LShfL N, Ri

sposta il contenuto del registro Ri a sinistra di N passi, N può usare un sistema di indirizzamento qualsiasi; i bit che escono dal registro sono persi, le posizioni a destra sono rimpiazzate da "0"

LShfR N, Ri

sposta il contenuto del registro Ri a destra di N passi, N può usare un sistema di indirizzamento qualsiasi; i bit che escono dal registro sono persi, le posizioni a sinistra sono rimpiazzate da "0"



- Istruzioni di Shift logiche Intel

shll \$N, %dst

sposta il contenuto del registro dst a sinistrabdi N passi, N può usare un sistema di indirizzamento qualsiasi; i bit che escono dal registro sono persi, le posizioni a destra sono rimpiazzate da "0" (operandi a 32bit)

shrl \$N, %dst

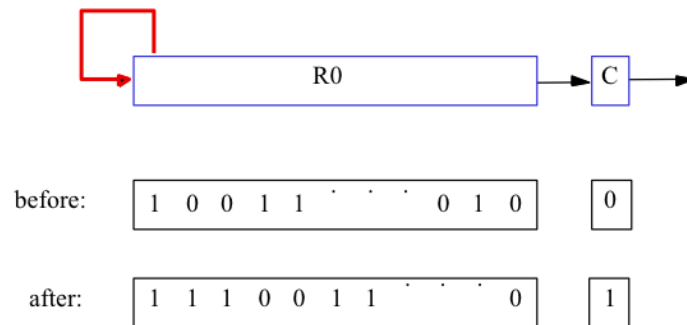
sposta il contenuto del registro Ri a destra di N passi, N può usare un sistema di indirizzamento qualsiasi; i bit che escono dal registro sono persi, le posizioni a sinistra sono rimpiazzate da "0" (operandi a 32bit)

- Istruzioni di Shift aritmetiche (moltiplicazione/divisione per due)

AShfL N, Ri

sposta il contenuto del registro Ri a sinistra di N passi, N può usare un sistema di indirizzamento qualsiasi; i bit che escono dal registro sono persi, le posizioni a destra sono rimpiazzate da "0" (identica a LShfL)

AShR N, Ri sposta il contenuto del registro Ri a destra di N passi, N può usare un sistema di indirizzamento qualsiasi; i bit che escono dal registro sono persi, le posizioni a sinistra sono rimpiazzate da "0" (operandi a 32bit)
 sposta il contenuto del registro Ri a destra di N passi, N può usare un sistema di indirizzamento qualsiasi; i bit che escono dal registro sono persi, le posizioni a sinistra sono rimpiazzate da bit uguali al bit di segno



(c) Arithmetic shift right ASHiftR #2,R0

• Istruzioni di Shift aritmetiche (moltiplicazione/divisione per due) Intel

sall \$N, %dest sposta il contenuto del registro dest a sinistra di N passi, N può usare un sistema di indirizzamento qualsiasi; i bit che escono dal registro sono persi, le posizioni a destra sono rimpiazzate da "0" (identica a LShfL)

sarl N, Ri sposta il contenuto del registro Ri a destra di N passi, N può usare un sistema di indirizzamento qualsiasi; i bit che escono dal registro sono persi, le posizioni a sinistra sono rimpiazzate da bit uguali al bit di segno

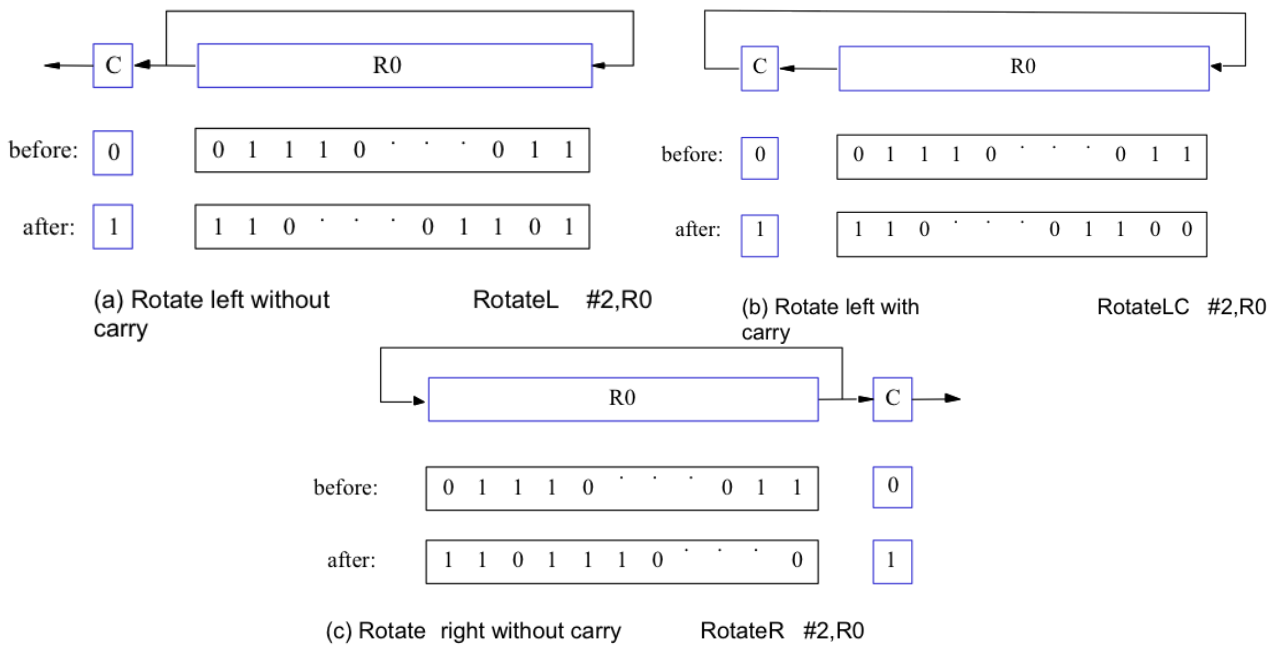
• Istruzioni di rotazione

RotL N, Ri sposta ciclicamente il contenuto del registro Ri a sinistra di N passi, N può usare un sistema di indirizzamento qualsiasi;

RotLC N, Ri come RotL, ma include il bit di carry

RotR N, Ri come RotL, ma verso destra

RotRC N, Ri come RotR, ma include il bit di carry



- Istruzioni di rotazione Intel

roll	\$N, %dst	sposta ciclicamente il contenuto del registro dst a sinistra di N passi, N può usare un sistema di indirizzamento qualsiasi;
rcll	\$N, %dst	come RotL, ma include il bit di carry
ror	\$N, %dst	come RotL, ma verso destra
rcr	\$N, %dst	come RotR, ma include il bit di carry

- Istruzioni di fine

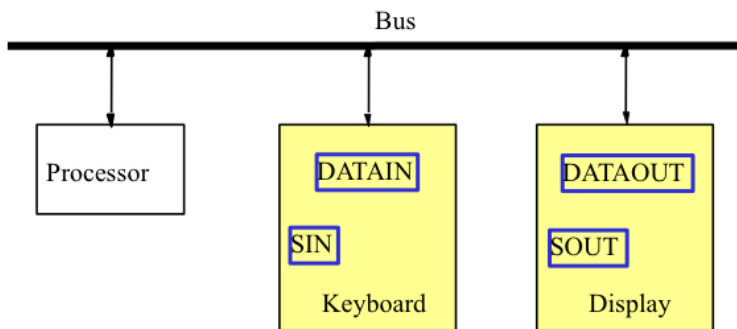
End	comunica all'assemblatore che il codice del programma è terminato
Return (IA32: ret)	comunica al sistema operativo (o a un altro programma, o al programma principale se stiamo considerando una subroutine) che il programma ha terminato l'esecuzione, contiene una opportuna etichetta di ritorno

Esecuzione dei programmi

- l'Assembler NON è il linguaggio macchina
- Assemblatore: traduce l'assembler in linguaggio macchina, risolvendo eventuali "conflitti" (es. salti in avanti)
 - simile a un compilatore
- Loader: carica un programma in memoria (ad esempio dal disco rigido) al momento dell'esecuzione, risolvendo gli indirizzi relativi e controllando che ci siano tutti i "pezzi" necessari all'esecuzione corretta

Funzioni fondamentali di I/O

- Qualsiasi elaboratore deve poter comunicare con l'esterno ... almeno tramite una tastiera e un video/ stampante



- è uso che il processore "veda" i registri periferici di interfaccia di tastiera e video come delle normali posizioni di memoria (memory mapped I/O)
 - DATAIN registro di ingresso
 - DATAOUT registro di uscita
- I/O sono molto più lenti del processore
- Normalmente i dispositivi periferici lavorano a byte e non a parole
- Ci serve una nuova istruzione...
 - MoveBy DATAIN, Ri carica il contenuto dell'input nel registro
 - MoveBy Ri, DATAOUT scrive il contenuto del registro sull'output
- Per funzionare correttamente le istruzioni devono essere abilitate dai rispettivi bit di "stato" SIN e SOUT

Cicli di lettura e scrittura

- Il processore deve poter aspettare che l'I/O sia abilitato, quindi deve avere un "loop" per leggere il registro solo al momento opportuno

READW

- TestBit SIN controlla lo stato dell'ingresso
- Branch=0 READW continua il loop se il dato non è disponibile
- MoveBy DATAIN, Ri legge il dato

WRITEW

- TestBit SOUT controlla lo stato dell'uscita
- Branch=0 WRITEW continua il loop se il display non può ricevere il dato
- MoveBy Ri, DATAOUT scrive il dato

- Intel

READW

- test \$3, SIN controlla lo stato dell'ingresso
- jnc READW continua il loop se il dato non è disponibile
- movb Din, %al legge il dato

WRITEW

- test \$3, SOUT controlla lo stato dell'uscita

jnc	WRITEW	continua il loop se il display non può ricevere il dato
movb	R%al, Dout	scrive il dato

Subroutines e gestione della memoria

- I programmi sono (auspicabilmente!!) composti di tante subroutine
- Il programma principale invoca una subroutine, che ne invoca un'altra, che ne invoca una successiva ...

Tecniche di gestione della memoria

- Coda
 - spazio di memoria in cui gli elementi che entrano per primi saranno anche i primi a uscire (FIFO – First In First Out)
 - Realizzabile in memoria tramite un buffer circolare gestito da puntatori dinamici
- Stack
 - spazio di memoria in cui gli elementi che entrano per primi gli ultimi a uscire (LIFO – Last In First Out)
 - Realizzabile in memoria tramite un buffer circolare gestito con un puntatore statico e uno dinamico

Coda

- Concettualmente una lista linkata di elementi di memoria
- Un puntatore mantiene l'indice dell'ultimo elemento entrato nella coda
- Un puntatore mantiene l'indice del primo elemento che deve uscire dalla coda
- Fondamentale in tutti i sistemi di trasmissione
- Poco adatta alla gestione della memoria nell'interazione tra subroutine
- Per evitare la crescita infinita della coda si gestisce come un buffer circolare tra un minimo e un massimo

Stack

- Come la pila dei giornali in salotto ...
- Un puntatore (fisso) segna il fondo della coda
- Un puntatore (mobile) segna l'ultimo elemento entrato nello stack e anche il primo ad uscire
- La gestione di uno stack in memoria si può fare semplicemente con una locazione di memoria (in genere un registro specializzato) che svolge il ruolo di puntatore e delle operazioni di move per scrivere e leggere gli elementi
- è utile effettuare dei controlli per evitare di leggere elementi da uno stack vuoto o di riempire lo stack all'infinito
- Le operazioni di lettura e scrittura in uno stack in genere si chiamano **PUSH ITEM** per la scrittura
e
POP ITEM per la lettura
- Le operazioni di push e pop sono normalmente realizzate con subroutine che controllano lo stato dello stack stesso

Subroutines e Stack

- Lo stack è la struttura usata abitualmente per il passaggio dei parametri e del

controllo tra subroutines

- Consente in modo naturale la ricorsione e l'annidamento di subroutines senza nessun vincolo a priori sul livello di annidamento
- Noi daremo per scontato che esistono le routines pop e push (con un solo parametro) e usiamo quelle per gestire lo stack, ovviamente invocando POP e PUSH ... usiamo lo stack!!

Salvataggio del PC

- Passando il controllo del programma ad una subroutine bisogna salvare il Program Counter per sapere da dove riprendere l'esecuzione
- L'istruzione Call salva il PC nello stack del processore e poi esegue un branch incondizionato all'istruzione di inizio della subroutine
- Bisogna sempre ricordare che una subroutine in assembler non è nient'altro che una sequenza di istruzioni in un'area di memoria

Subroutines e parametri

- Nei linguaggi di alto livello i parametri si passano ... tra parentesi
call subr(par1,par2,par3)
- In assembler invocare una subroutine vuol dire solamente effettuare un salto incondizionato a una locazione di memoria
Call SUBR ↔ Branch SUBR
- I parametri che userà la subroutine devono già essere in una opportuna area di memoria ... in cima allo stack
- I parametri restituiti dalla subroutine saranno anche loro passati attraverso lo stack
- Invocando una subroutine in un linguaggio di alto livello
es:

```
is_date(day,month,year)
```

- Si ottiene una sequenza di istruzioni in assembler del tipo

```
Push PAR3
```

```
Push PAR2
```

```
Push PAR1
```

```
Call ISDATE
```

- Dove PAR1, PAR2 e PAR3 sono le locazioni di memoria in cui sono state memorizzate le variabili day, month e year

Passaggio dei parametri

- I parametri vengono scritti nello stack (ad esempio in ordine inverso)
- Per ultimo viene salvato il PC in modo da sapere l'indirizzo dove si dovrà ritornare in uscita dalla subroutine
- I parametri possono anche essere variabili in uscita, che userà il programma chiamante

Salvataggio dei registri

- Se una subroutine deve usare dei registri (diciamo R0, R1 ed R2) deve garantire di non distruggere il loro contenuto, per poterlo ripristinare in uscita
- Li memorizza in cima allo stack

Variabili locali e Frame Pointer (FP)

- In molti linguaggi le subroutines possono avere variabili locali che non sono viste dal resto del programma e la cui memoria viene rilasciata all'uscita dalla subroutine
- Il luogo naturale di memorizzazione è nuovamente lo stack dove la memoria può essere assegnata all'ingresso in una subroutine e rilasciata in uscita
- Per mantenere chiaramente separate le variabili globali salvate nello stack e quelle locali si usa un ulteriore puntatore chiamato Frame Pointer
- FP separa la parte di memoria dedicata al passaggio dei parametri dalla memoria locale
- I registri R1 ed R0 sono salvati perchè assumiamo che la subroutine li userà
- FP non varia durante l'esecuzione della subroutine

IA32 – gcc convenzioni di chiamata

- Utilizzando gcc, per rendere una funzione assembler invocabile dal C, occorre rispettare le seguenti convenzioni:
 - I parametri vengono immessi sullo stack dal chiamante a partire dall'ultimo verso il primo.
 - as. es., all'interno della funzione troviamo in (%esp) l'indirizzo di ritorno, 4(%esp) il primo parametro, 8(%esp) il secondo (assumendo che il primo sia a 32 bit) ecc.
 - I registri esp, ebp, esi, edi, ebx devono essere preservati
 - se si usano bisogna prima salvarli sullo stack e poi ripristinarli
 - i parametri di ritorno, se interi o puntatori, vengono restituiti in eax

Codifica delle istruzioni macchina: Architetture CISC e RISC

- CISC: Complex Instruction Set Computer
 - istruzioni complesse di lunghezza variabile, le istruzioni sono più lente da eseguire
- RISC: Reduced Instruction Set Computer
 - istruzioni semplici tutte di una singola parola, le istruzioni sono più veloci, ma ne servono di più per svolgere lo stesso compito

La "CPU"

Processore

- Il termine "CPU" (Central Processing Unit) è un po' fuorviante nei sistemi moderni, dove spesso c'è più di una unità in grado di svolgere istruzioni (multiprocessori)
- Il (i) processore(i) sono le unità in grado di effettuare le operazioni specificate dal progetto dell'elaboratore stesso
- Il termine esatto sarebbe Instruction Set Processor, cioè "colui che esegue l'insieme delle istruzioni"

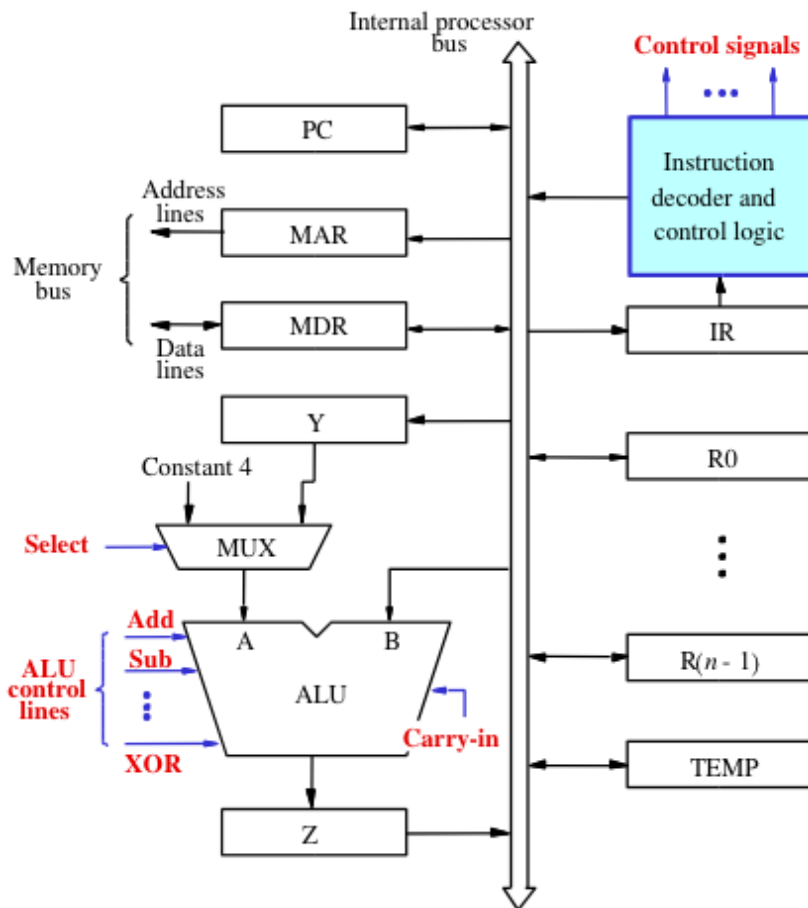
Notazione

- Per distinguere le operazioni interne del processore dal linguaggio assembler usiamo una notazione stile RTN
 - $R1 \leftarrow [LOC]$ carica in R1 il contenuto della locazione LOC
- Per indicare la configurazione dei segnali di controllo all'interno del processore, che in ultima analisi realizzano le istruzioni, usiamo la notazione $S1, S2, S3, \dots$ intendendo che i soli segnali $S1, S2, S3, \dots$ sono attivi in uno specifico colpo di clock

Concetti base

- Un processore esegue una istruzione alla volta
- Una istruzione è normalmente composta da tre fasi ben distinte
 1. caricamento dell'istruzione dalla memoria (fetch)
 $IR \leftarrow [[PC]]$
 2. incremento del program counter
 $PC \leftarrow [PC]+4$
 3. esecuzione dell'istruzione vera e propria, cioè realizzazione delle azioni specificate dall'istruzione
 S_i, S_j, S_k, \dots
 S_b, S_q, S_z, \dots
.....

Organizzazione interna di un processore a bus singolo



Organizzazione

- Registri Y,Z, TEMP non sono visibili all'esterno e memorizzano info solo durante l'esecuzione di una istruzione
- ALU: Arithmetic and Logic Unit
- Esiste un solo "percorso delle informazioni" (datapath) all'interno del processore: non posso svolgere operazioni in parallelo
- Il multiplexer all'ingresso della ALU seleziona se il secondo operando è il registro interno Y (SelectY) o semplicemente "4" (Select4) (l'istruzione più frequente è l'incremento di un puntatore in memoria!!)

Registri, comandi e memorizzazione locale

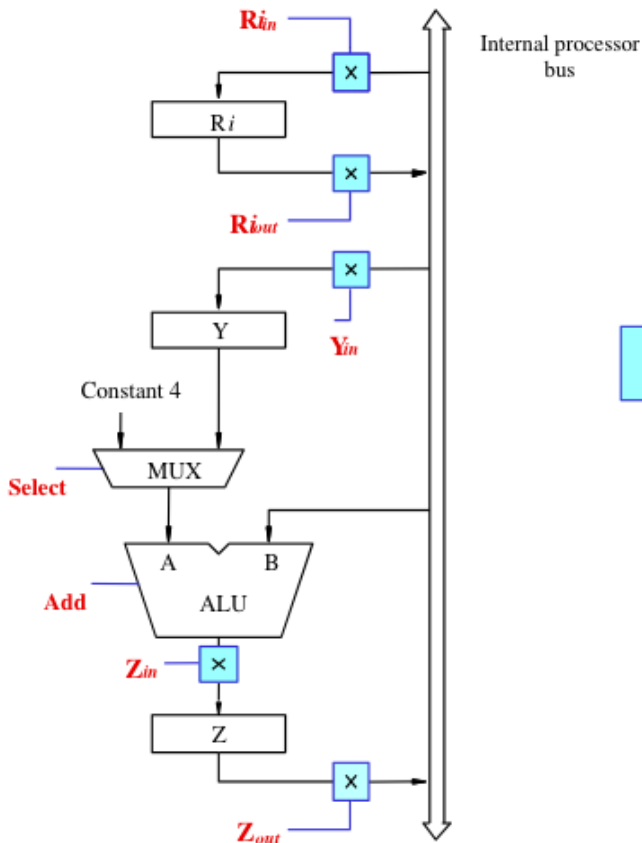
- Tutti i registri di un processore sono ottenuti con flip-flop
- Ipotizziamo che siano del ff "edge triggered", per cui sul fronte di salita di un colpo di clock le uscite assumono il valore degli ingressi
- Il processore evolve in cicli temporali sincroni con il clock
- I comandi sono "attivati" all'inizio di un ciclo, sul fronte successivo si ha il trasferimento dei dati da un punto ad un altro (reg↔bus e viceversa) e i comandi cambiano stato
- I comandi non sono nient'altro che "fili di abilitazione" dei diversi circuiti del processore
- Es.
 - un "comando" consente di accoppiare il registro R0 per trasferire il suo contenuto sul bus
 - un altro "comando" consente di accoppiare il bus con il registro R0 per


trasferire il suo contenuto in R0

- un terzo "comando" dice alla ALU di sommare (Add) i due ingressi consente

-

"Gating" dei registri al bus



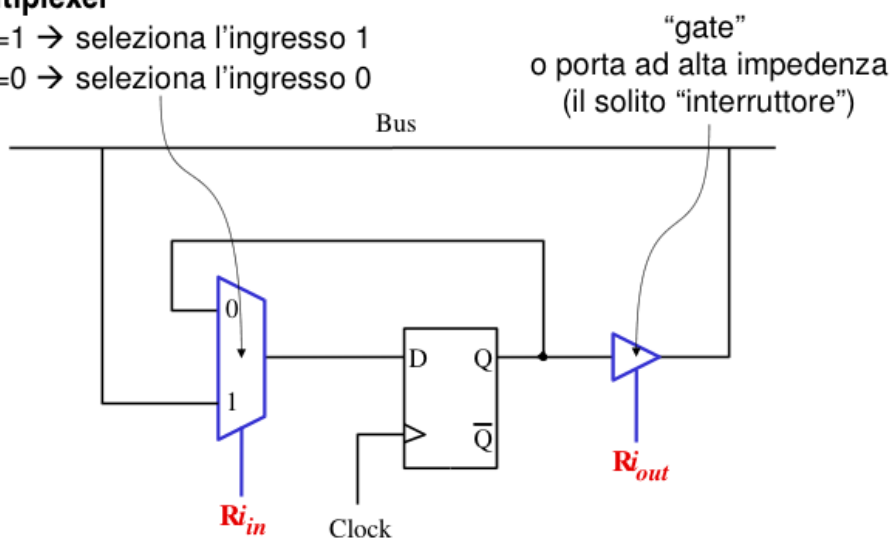

 è una porta di "abilitazione" se il segnale è valido allora l'ingresso/uscita del registro è connesso al bus altrimenti si trova in uno stato di "alta impedenza" cioè non è in comunicazione con il bus stesso

Schema di gating per un bit

multiplexer

$R_{j_{in}}=1 \rightarrow$ seleziona l'ingresso 1

$R_{j_{in}}=0 \rightarrow$ seleziona l'ingresso 0



Trasferimento di un registro - $R_i \leftarrow [R_j]$

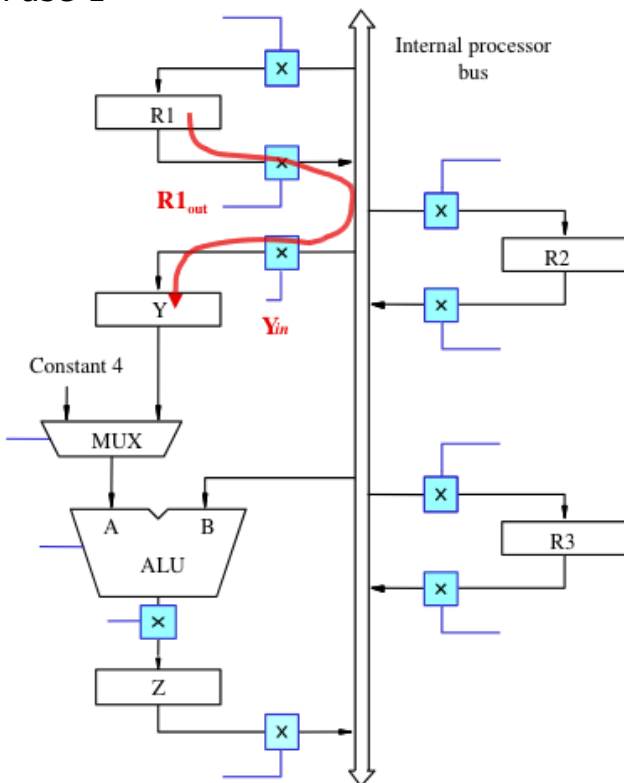
- Abilita il registro R_j con $R_{j_{out}}$, trasferendo così il suo contenuto sul bus

- Abilita il registro Ri con Ri_{in} , trasferendo così il valore presente sul bus nel registro Ri
- Il trasferimento può avvenire in un unico colpo di clock, con la nostra notazione:
 1. Rj_{out}, Ri_{in}

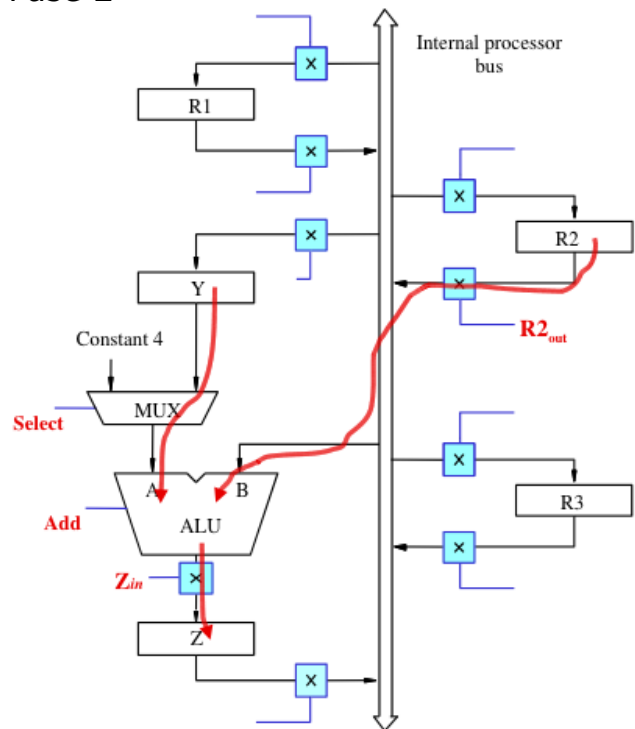
Operazioni Aritmetiche (es. "Add")

- Consideriamo l'istruzione
Add R1,R2,R3
- La parte di "esecuzione" avviene in 3 passi:
 1. $R1_{out}, Y_{in}$
 2. $R2_{out}, SelectY, Add, Z_{in}$
 3. $Z_{out}, R3_{in}$
- Torniamo all'architettura del processore per seguire l'esecuzione...

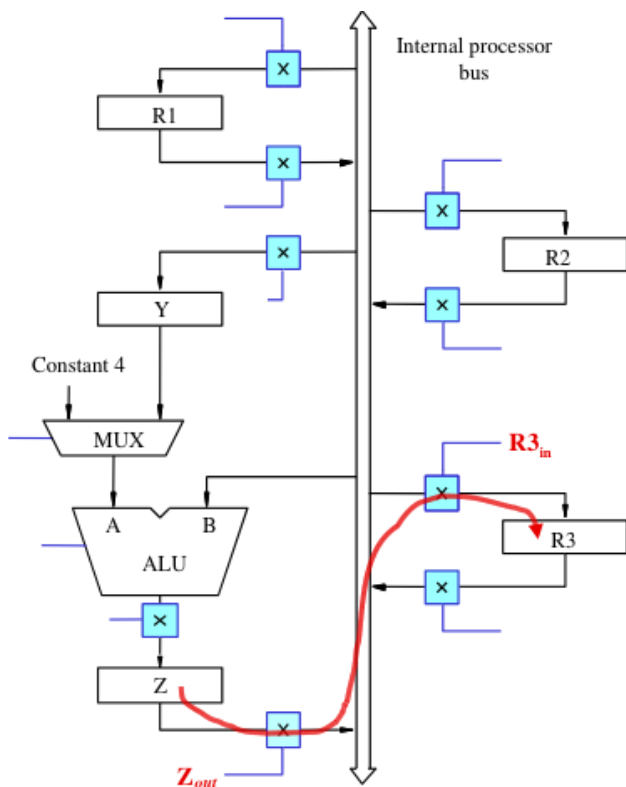
Fase 1



Fase 2

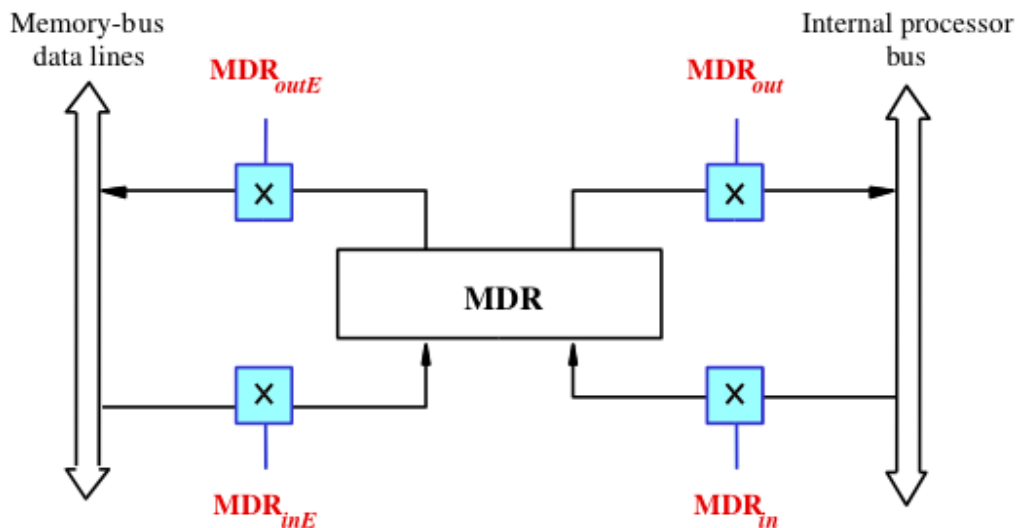


Fase 3



Accesso in memoria

- È più complicato perchè il registro di accesso alla memoria MDR è connesso sia al bus interno che al bus esterno



- La memoria è più lenta, quindi è possibile che il processore debba attendere il risultato, sia esso Read o Write
- Per "Read (R1),R2" abbiamo la sequenza logica:
 1. $MAR \leftarrow [R1]$
 2. inizia una Read sul bus esterno (memoria)
 3. aspetta un segnale MCF dalla memoria (es. slave ready)
 4. carica MDR dal bus esterno
 5. $R2 \leftarrow [MDR]$
- Cioè vengono attivati i comandi

1. $R1_{out}$, MAR_{in} , Read
2. MDR_{inE} , WMCF,
3. MDR_{out} , $R2_{in}$

- Una operazione di write è ovviamente analoga
- Per "Write $R1,(R2)$ " abbiamo la sequenza logica:
 1. $MAR \leftarrow [R2]$
 2. inizia una Write sul bus esterno (memoria)
 3. $MDR \leftarrow [R1]$
 4. carica MDR sul bus esterno
 5. aspetta un segnale MCF dalla memoria (es. slave ready)
- Cioè vengono attivati i comandi
 1. $R2_{out}$, MAR_{in}
 2. $R1_{out}$, MDR_{in} , Write
 3. MDR_{outE} , WMFC

Esecuzione di una istruzione completa: Add ($R3$), $R1$

- Caricamento dell'istruzione
 - caricamento dell'operando ($R3$), perchè $R1$ è già nel processore
 - Esecuzione dell'addizione
 - Caricamento del risultato in $R1$
1. PC_{out} , MAR_{in} , Read , Select4 , Add , Z_{in}
 2. Z_{out} , PC_{in} , Y_{in} , WMFC
 3. MDR_{out} , IR_{in}
 4. $R3_{out}$, MAR_{in} , Read
 5. $R1_{out}$, Y_{in} , WMFC
 6. MDR_{out} , SelectY , Add, Z_{in}
 7. Z_{out} , $R1_{in}$, End

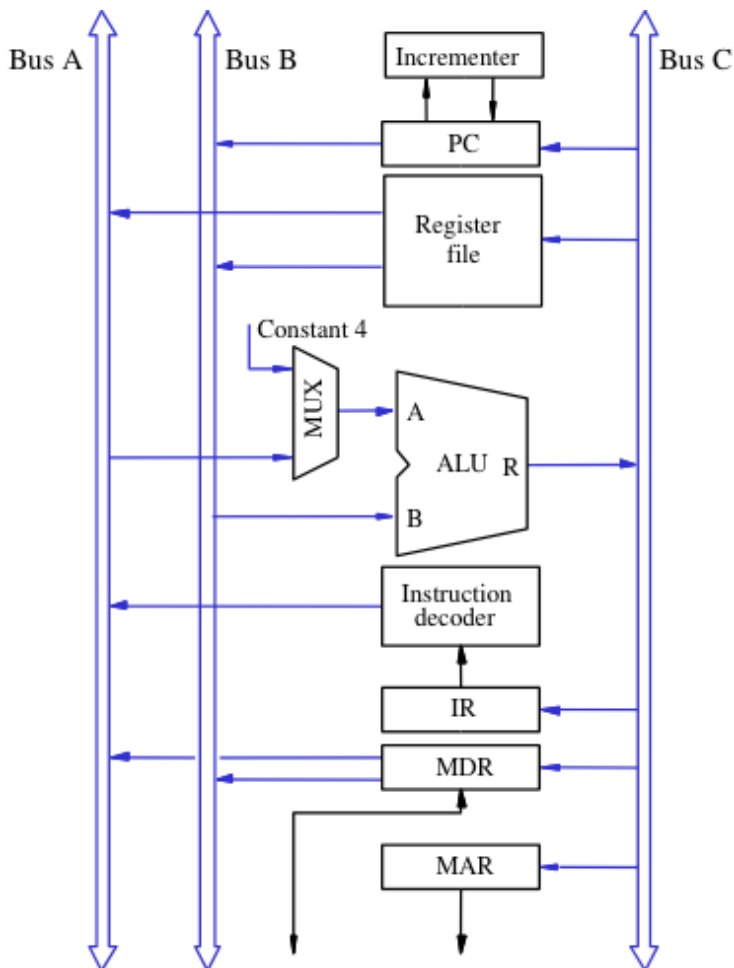
Istruzioni di branch

- Alterano il normale flusso del programma
 - Bisogna sommare l'offset (contenuto nell'istruzione) al PC
 - Per i branch condizionati bisogna aggiungere un "filo" di controllo dal registro di stato del processore
1. PC_{out} , MAR_{in} , Read , Select4 , Add , Z_{in}
 2. Z_{out} , PC_{in} , Y_{in} , WMFC
 3. MDR_{out} , IR_{in}
 4. $IR-Offset_{out}$, Add, Z_{in}
 5. Z_{out} , PC_{in} , End

Processori multi-bus

- La semplice architettura a bus singolo non è adatta a processori ad elevate prestazioni
- Avere più di un bus di comunicazione consente di eseguire più compiti in parallelo (es. lettura e scrittura di registri contemporaneamente)
- Visto che il PC viene sempre incrementato, tanto vale dedicargli un circuito
- Tutte le operazioni, tranne i trasferimenti, coinvolgono la ALU, bisogna quindi usarla al meglio

Processore a tre bus



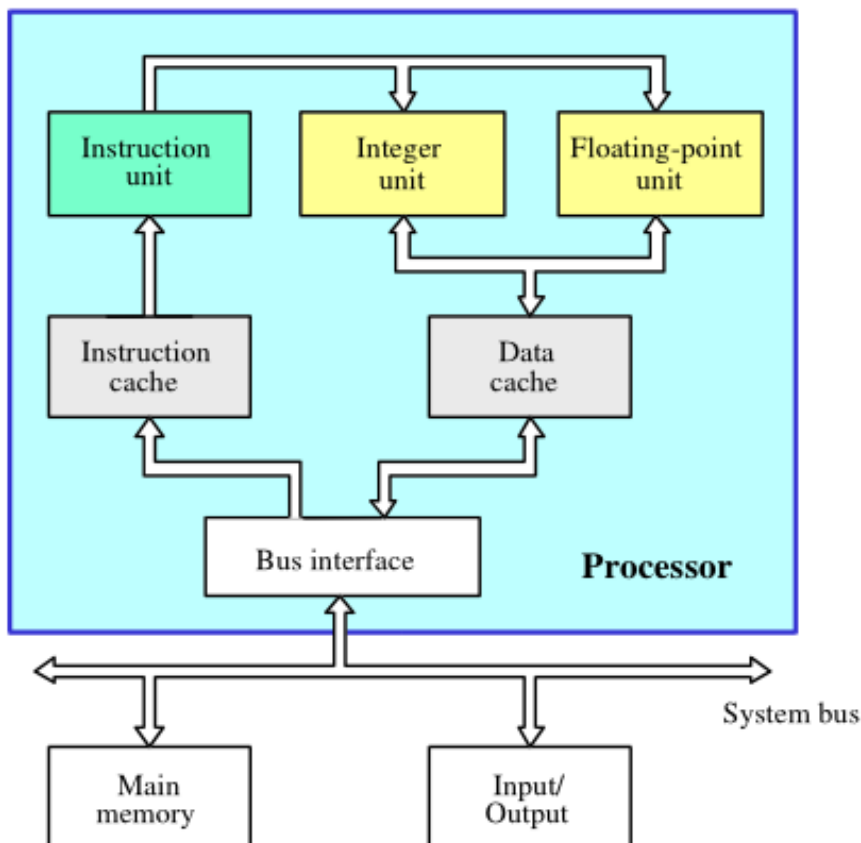
Esecuzione di una istruzione completa: Add R4, R5, R6

- Si aumenta il parallelismo, perché diversi trasferimenti possono avvenire su bus diversi
 - La ALU può semplicemente passare un segnale dai bus A o B sul C con i comandi R=A e R=B
 - Spariscono Y, Z e TEMP
1. PC_{out} , $R=B$, MAR_{in} , Read , $PCInc$
 2. WMFC
 3. MDR_{out} , $R=B$, IR_{in}
 4. $R4_{outA}$, $R5_{outB}$, SelectA, Add , $R6_{in}$, End

Tipi di processore

- Esistono molti tipi di processori, con varie architetture
- In tutti esiste una separazione funzionale "a blocchi" che rispettano le funzioni viste
- Possono avere 1 o più memoria cache
- Possono avere diverse "ALU"
 - 1 per op. logiche
 - 1 per op. tra interi
 - 1 per op. FP

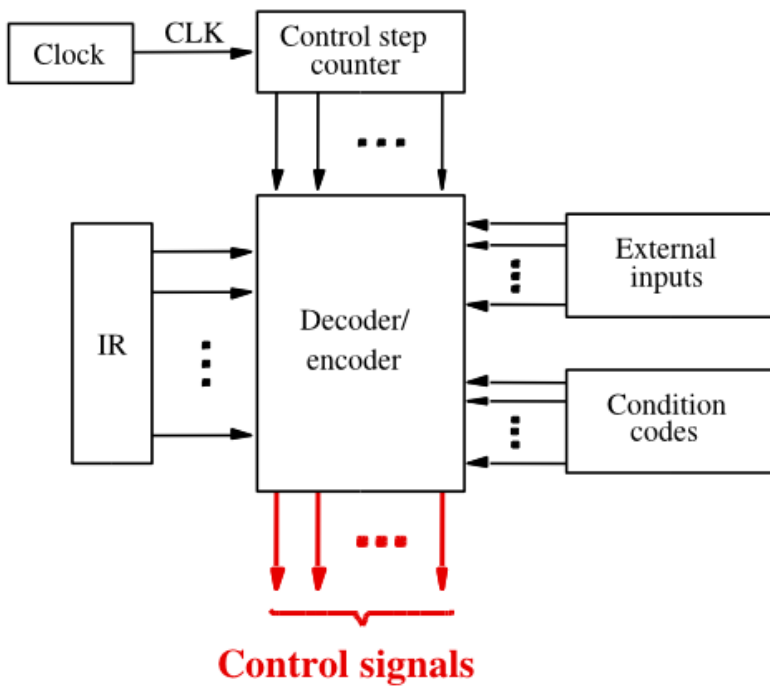
Diagramma a blocchi funzionali di un processore completo



Controllo e comandi

- Il "cuore" di una CPU, è l'unità di decodifica delle istruzioni e generazione dei comandi
- L'unità è un circuito combinatorio, ma ha bisogno di sapere quale "passo" dell'istruzione deve essere eseguita, quindi ha al suo interno un "contatore di passi", incrementato dal clock
- Nelle architetture tradizionali, il controllo avviene in modo "cablato" (hardwired)
- Nelle architetture moderne, invece, al posto di un circuito combinatorio esiste un "microcalcolatore embedded" che può essere "microprogrammato" per realizzare insiemi di comandi "interni" diversi a parità di istruzioni "esterne"

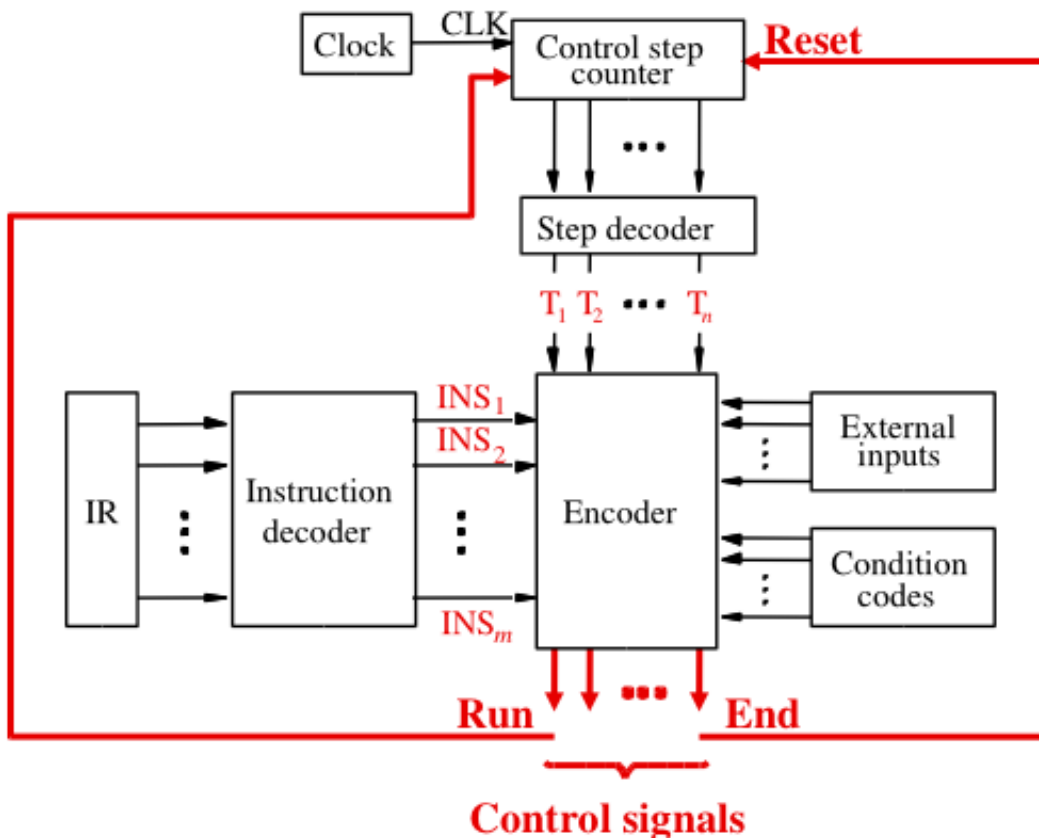
Organizzazione dell'unità di controllo



Comandi

- I "comandi" non sono nient'altro che funzioni logiche degli ingressi
- Per semplificare la logica combinatoria decodifica dell'istruzione e codifica dei comandi sono realizzati separatamente
- Anche il contatore di passi viene decodificato in modo da avere un solo segnale attivo ad ogni passo
- I comandi Run e End comandano ... i comandi

Organizzazione dettagliata dell'unità di controllo



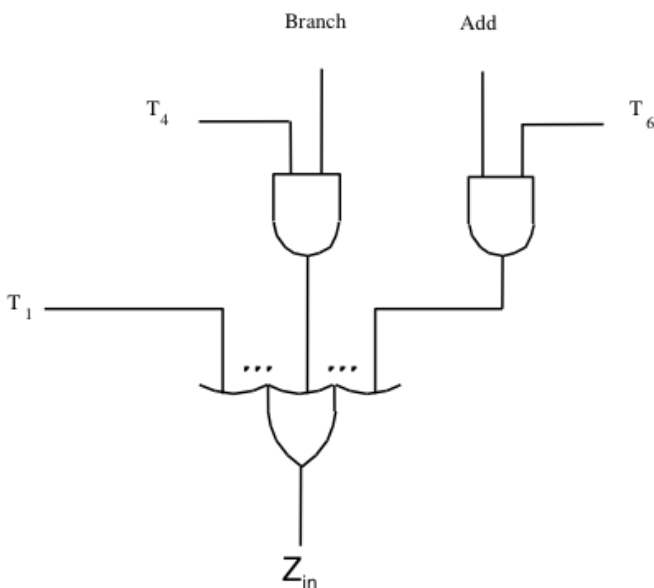
Generazione dei comandi

- I decodificatori di istruzione e del clock sono circuiti simili al decodificatore BCD visto come esempio di circuito combinatorio "complesso"
- I codici condizionali sono il risultato delle operazioni precedenti, cioè i bit del registro di stato (=0, <0, >0, ...)
- I segnali esterni sono, ad esempio, il segnale MFC, che segnala il termine di una operazione in memoria
- Run=0 ferma il contatore, ad esempio mentre un comando WMCF è attivo
- Il singolo comando, es. R2out ,SelectY, Add, Zin è una funzione logica, che deve tenere conto di tutte le volte che tale segnale deve essere attivo ... la realizzazione come somma di prodotti è la più logica e banale!!!
- Es. per il nostro processore a bus singolo:

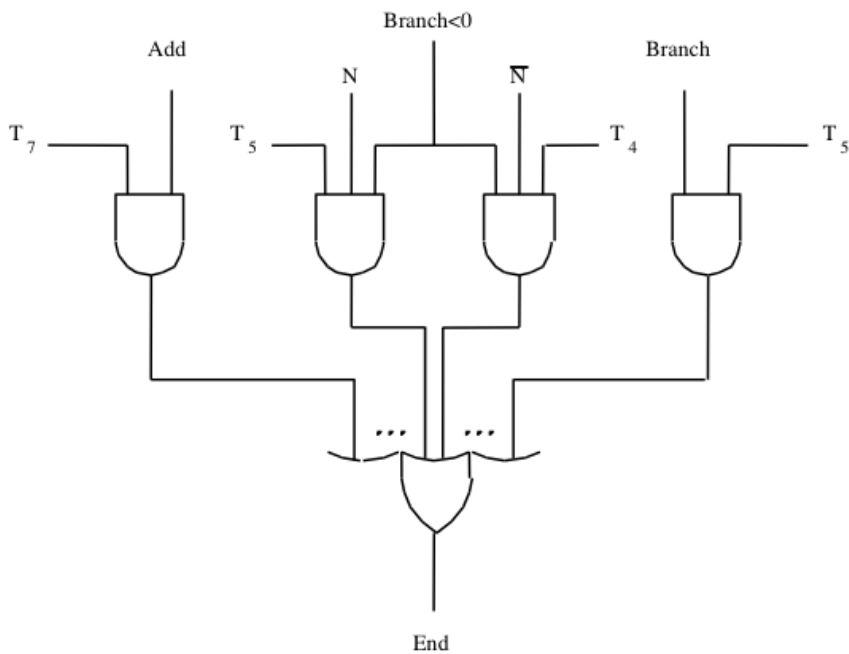
$$Z_{in} = T_1 + T_6 \cdot \text{Add} + T_4 \cdot \text{Branch} + \dots$$

$$\text{End} = T_7 \cdot \text{Add} + T_5 \cdot \text{Branch} + (T_5 \cdot N + T_4 \cdot N) \cdot \text{Branch} < 0 + \dots$$

Generazione del comando Zin



Generazione del comando End



Considerazioni sul controllo cablato

- Adatto a realizzare processori semplici e molto veloci
- Economico
- Poco flessibile
- Difficile realizzare set di istruzioni complessi
- Impossibile "adattare" un processore a eseguire diversi Instruction Sets
- Richiede una progettazione ex-novo per modificare un set di istruzioni

Controllo programmabile

- Anzichè realizzare le funzioni logiche di controllo attraverso una rete combinatoria si realizzano attraverso "parole di controllo" (Control Words-CW) o microistruzioni che contengono una codifica per l'insieme di comandi da attivare in un dato passo
- L'esecuzione di una istruzione diventa semplicemente una sequenza di CW
- La codifica più semplice prevede un bit per ogni possibile comando

Microprogrammazione

- L'insieme di CW che compongono una istruzione sono note come "microroutines"
- La "scrittura" di microroutines è la microprogrammazione di un processore
- Tramite dei microprogrammi è possibile ottenere grande flessibilità e un set di istruzioni macchina praticamente arbitrario
- L'insieme delle microistruzioni forma il "microcodice" che è contenuto in una speciale memoria chiamata "control store" (memoria di controllo)
- La decodifica di una istruzione si traduce semplicemente nel generare l'indirizzo iniziale della microroutine appropriata

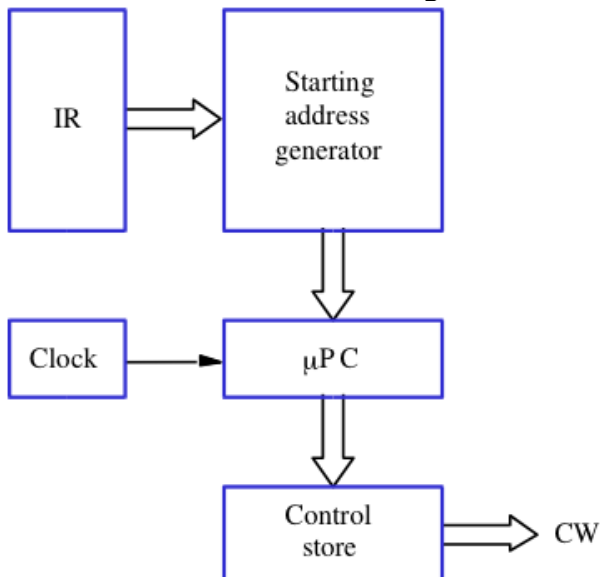
Codifica delle microistruzioni

- Le CW sono caricate in un circuito che genera i comandi
- Se a ciascun bit corrisponde un comando il circuito è banale
- L'uso di un bit per ogni comando può risultare non conveniente a causa delle grandi dimensioni delle CW (un processore può avere decine se non centinaia)

di comandi!)

- è possibile codificare le CW in modo più compatto, anche se cresce la complessità di decodifica e il rischio di rallentare la macchina

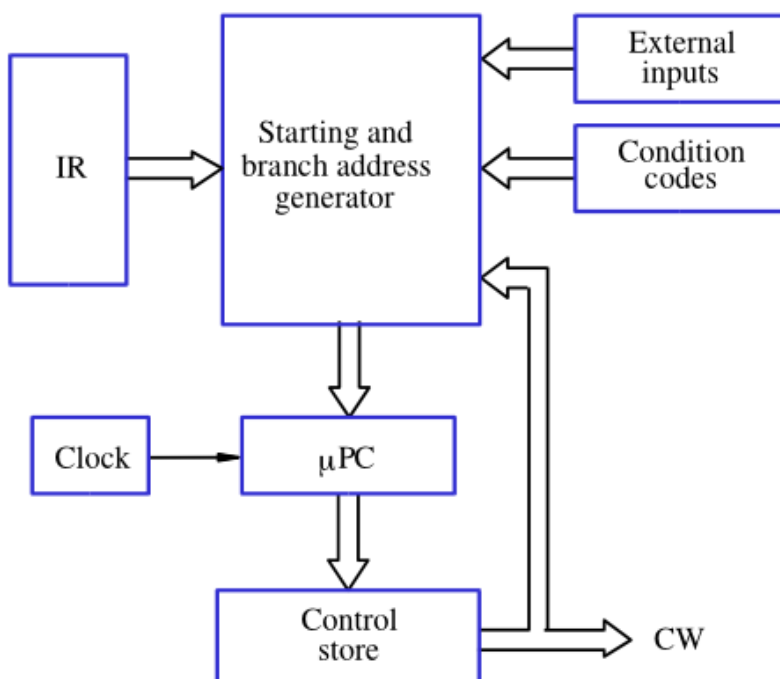
Architettura di base di un generatore di comandi microprogrammato



Branching

- Una istruzione di branch, soprattutto se condizionato, prevede una esecuzione non strettamente sequenziale
- Le condizioni di branch sono normalmente "esterne" all'istruzione
- L'organizzazione del microprogramma e del generatore di comandi microprogrammato deve prevedere la possibilità di istruzioni con branch e quindi di interazione con segnali esterni

Architettura di un generatore di comandi microprogrammato con possibilità di branching



Organizzazione delle microistruzioni

- In genere le microistruzioni sono organizzate in campi che governano insieme di segnali tra loro mutuamente esclusivi (es. abilitazione "out" dei registri, o comandi alla ALU)
- Ciò consente di codificare in modo compatto le istruzioni con la garanzia di non avere mai conflitti nella generazione dei comandi
- Particolare attenzione viene posta ai diversi modi di indirizzamento ammessi dal processore

Considerazioni sul controllo microprogrammato

- Tutti i processori "general purpose" sono microprogrammati
- La disponibilità di memorie "extraveloci" per il control store è fondamentale
- è necessario avere una struttura (che non abbiamo visto) che consente di caricare una microistruzione mentre la precedente viene eseguita, altrimenti si rallenta il processore
- è possibile "emulare" diversi processori (cioè diversi Instruction Set) a partire dallo stesso hardware

Il sistema di memorizzazione

Organizzazione gerarchica della memoria

- Un elaboratore senza memoria non funziona ...
- La memoria negli elaboratori non è tutta uguale
- Esistono compromessi tra costo, prestazioni e dimensione della memoria
- La memoria è tipicamente più lenta del processore e può limitarne le prestazioni
- è importante avere memoria veloce "vicino" al processore e grandi quantità di memoria complessiva

Basics

- La memoria serve a contenere dati, bisogna poter leggere e scrivere in memoria ...
- La memoria indirizzata direttamente (principale o cache) è
 - di tipo volatile, cioè il suo contenuto viene perso se si spegne l'elaboratore
 - limitata dallo spazio di indirizzamento del processore
- La memoria indirizzata in modo indiretto
 - di tipo permanente: mantiene il suo contenuto anche senza alimentazione
 - ha uno spazio di indirizzamento "software" non limitato dal processore
- Differenze radicali tra memoria volatile e permanente
- Memoria "RAM": è tipicamente la memoria principale dell'elaboratore
- Memoria "virtuale": è una parte dello spazio di indirizzamento diretto che non può essere accomodata nella RAM, ma viene mappata su un dispositivo periferico tipo disco, il cui accesso, pur trasparente al processore è molto più lento rispetto alla memoria "RAM"
- Le informazioni nella memoria principale (indirizzamento diretto) è accessibile al processore in qualsiasi momento
- Le informazioni nella memoria periferica (indirizzamento indiretto) devono prima essere trasferite nella memoria principale
- Il trasferimento dell'informazione tra memoria principale e memoria periferica è mediato dal software (tipicamente il S.O.)
- **Tempo di accesso:** tempo richiesto per una operazione di lettura/scrittura nella memoria
- **Tempo di ciclo:** tempo che intercorre tra l'inizio di due operazioni (es. due read) tra locazioni diverse; in genere leggermente superiore al tempo di accesso
- **Casuale (accesso):**
 - non vi è alcuna relazione o ordine nei dati memorizzati
 - tipico delle memorie a semiconduttori
- **Sequenziale (accesso):**
 - l'accesso alla memoria è ordinato o semi-ordinato
 - il tempo di accesso dipende dalla posizione
 - tipico dei dischi e dei nastri
- **RAM:** Random Access Memory
 - memoria scrivibile/leggibile a semiconduttori

- tempo di accesso indipendente dalla posizione dell'informazione
- **ROM: Read Only Memory**
 - memoria a semiconduttori in sola lettura
 - accesso casuale o sequenziale

Memoria principale

- Connessione "logica" con il processore

MAR: Memory Address Register

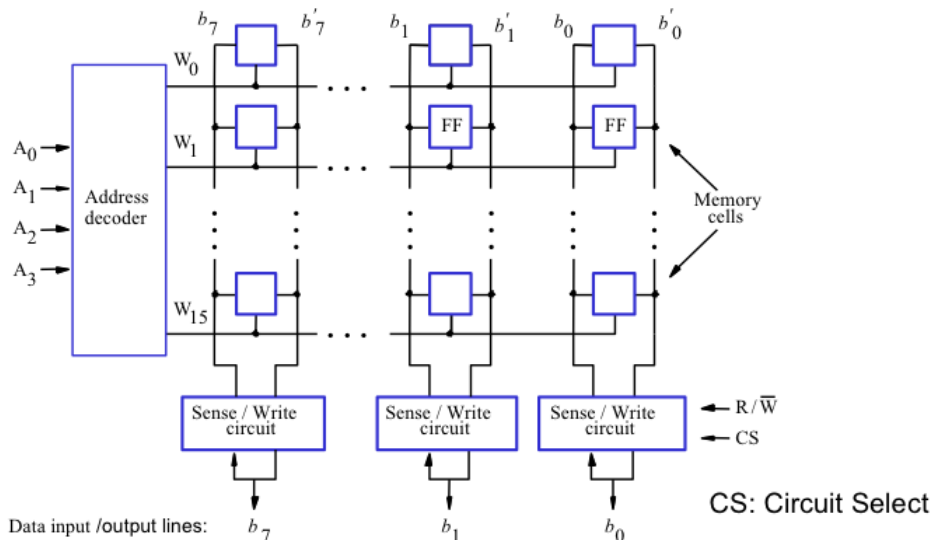
MDR: Memory Data Register

MFC: Memory Function Completed

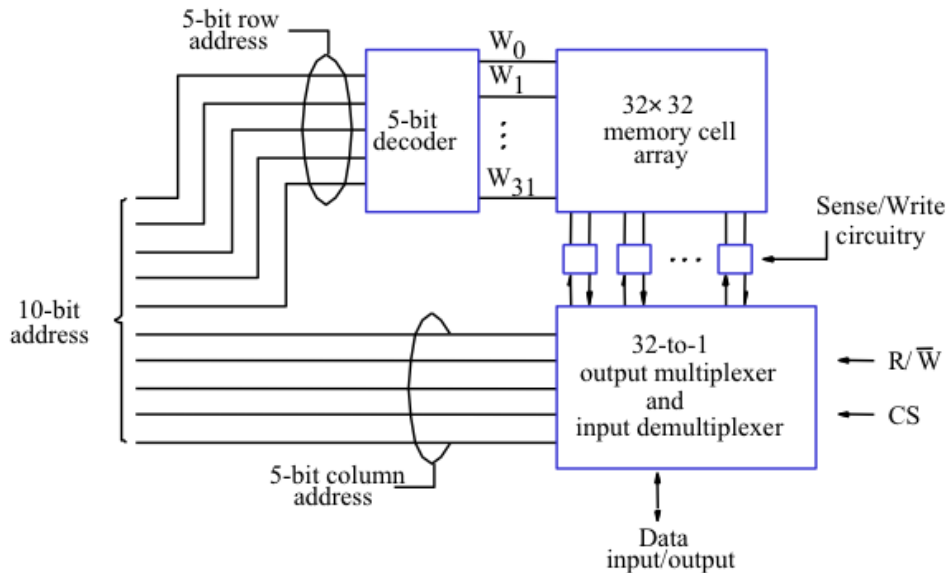
Memorie RAM a semiconduttori

- Memorizzano singoli bit, normalmente organizzati in byte e/o word
- Data una capacità N (es. 512 Kbit) la memoria può essere organizzata in diversi modi a seconda del parallelismo P (es. 8, 1 o 4)
 - 64K X 8
 - 512K X 1
 - 128K X 4
- L'organizzazione influenza in numero di pin di I/O dell'integrato (banco) che realizza la memoria

Organizzazione dei bit in un banco di memoria 16 X 8



Organizzazione di un banco di memoria 1K X 8

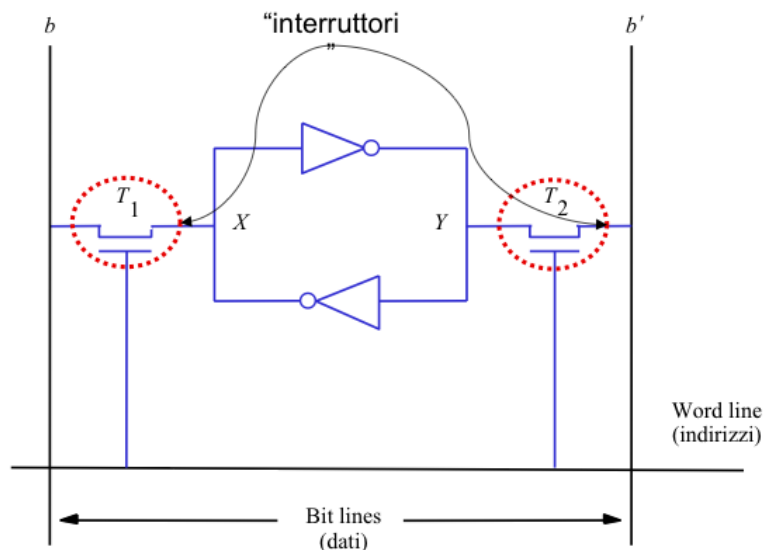


Tipi di memoria diversi

- L'organizzazione delle memorie RAM è sostanzialmente sempre uguale dato il "formato" (es. 4M X 8, 16 M X 16, Q X P)
- Il numero di piedini di accesso al chip (tralasciando quelli di alimentazione e controllo) è dato da $\log_2(Q) + P$
- A parità di memoria ($N=QXP$) ho meno piedini se il parallelismo è minore
- Cambiando il "tipo" di memoria RAM cambia la cella base di memorizzazione dei bit

Memorie Statiche (SRAM)

- Sono memorie in cui i bit possono essere tenuti indefinitamente (posto che non manchi l'alimentazione)
- Estremamente veloci (tempo di accesso di pochi ns)
- Consumano poca corrente (e quindi non scaldano)
- Costano care perchè hanno molti componenti per ciascuna cella di memorizzazione
- Cella di memoria:



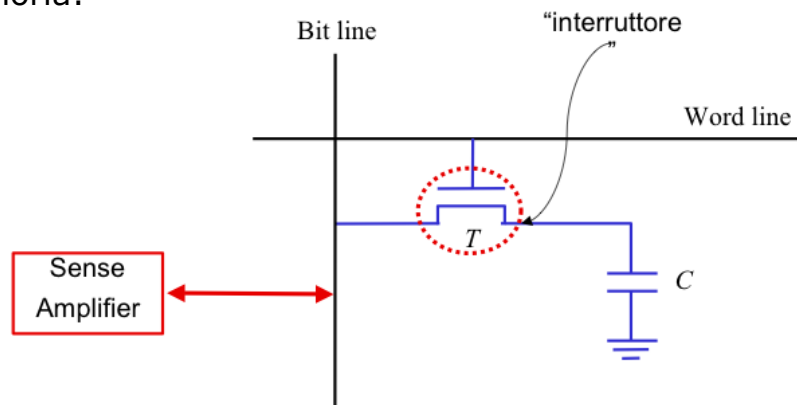
- $b' = \text{NOT}(b)$: i circuiti di terminazione della linea di bit (sense/write circuit) interfacciano il mondo esterno che non accede mai direttamente alle celle
- La presenza contemporanea di b e $\text{NOT}(b)$ consente un minor tasso di errori

- Scrittura: la linea di word è alta e chiude T_1 e T_2 , il valore presente su b e b' , che funzionano da linee di pilotaggio, viene memorizzato nel latch a doppio NOT
- Lettura: la linea di word è alta e chiude T_1 e T_2 , le linee b e b' sono tenute in stato di alta impedenza: il valore nei punti X e Y viene "copiato" su b e b'
- Se la linea di word è bassa T_1 e T_2 sono interruttori aperti: il consumo è praticamente nullo

RAM dinamiche (DRAM)

- Sono le memorie più diffuse nei PC e simili
- Economiche e a densità elevatissima (in pratica 1 solo componente per ogni cella)
 - la memoria viene ottenuta sotto forma di carica di un condensatore
- Hanno bisogno di un rinfresco continuo del proprio contenuto che altrimenti "svanisce" a causa delle correnti parassite
- Consumi elevati a causa del rinfresco continuo

- Cella di memoria:



- Scrittura: la linea di word è alta e chiude T , il valore presente su b viene copiato su C (carica il transistor)
- Lettura: la linea di word è alta e chiude T , un apposito circuito (sense amplifier) misura la tensione su C
 - se è sopra una soglia data pilota la linea b alla tensione nominale di alimentazione, ricaricando il condensatore C ,
 - se è sotto la soglia data mette a terra la linea b scaricando completamente il condensatore
- Tempi di rinfresco
 - Nel momento in cui T viene aperto il condensatore C comincia a scaricarsi (o caricarsi, anche se più lentamente) a causa delle resistenze parassite dei semiconduttori
 - È necessario rinfrescare la memoria prima che i dati "spariscano" (basta fare un ciclo di lettura)
 - In genere il chip di memoria contiene un circuito per il rinfresco (lettura periodica di tutta la memoria); l'utente non si deve preoccupare del problema
 - Una tipica DRAM moderna ha bisogno di un rinfresco ogni 64ms e funziona a 100 MHz ($T_c = 1/(100\text{MHz}) = 10\text{ns}$) o più
 - Il tempo totale di rinfresco T_r dipende dal numero di linee Q e dal

numero di cicli N_c necessari ad una lettura:

$$T_r = Q \times N_c \times T_c$$

- Se abbiamo un banco di memoria $32k \times P$ e ci vogliono 4 cicli per fare una lettura allora

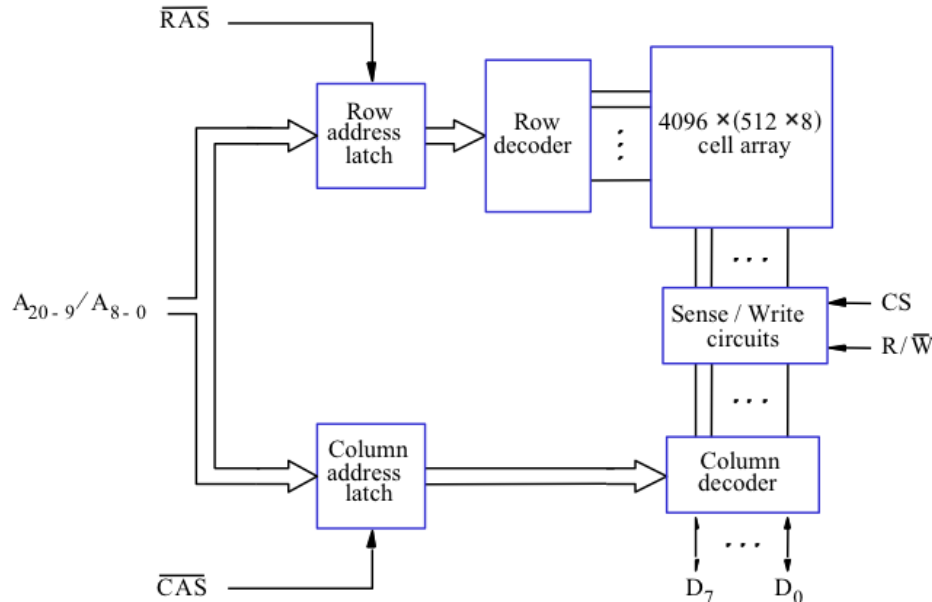
$$T_r = 32768 \times 4 \times 10 \text{ ns} \approx 1.3 \text{ ms}$$

- Lo spreco di tempo è $1.3/64 \approx 2\%$
- Più è grande una memoria maggiore è lo spreco

- **Multiplicazione degli indirizzi**

- Dato l'elevata integrazione delle DRAM il numero di pin di I/O è un problema
- È usuale moltiplicare nel tempo l'indirizzo delle righe e delle colonne negli stessi fili
- Normalmente le memorie non sono indirizzabili al bit, per cui righe e colonne si riferiscono a byte e non a bit
- Es. una memoria $2M \times 8$ (21 bit di indirizzo) può essere organizzata in 4096 righe (12bit di indirizzo) per 512 colonne (9bit di indirizzo) di 8 bit ciascuno

- **Organizzazione di una DRAM $2M \times 8$**



- **Modo di accesso veloce**

- Spesso i trasferimenti da/per la memoria avvengono a blocchi (o pagine)
- Nello schema appena visto, vengono selezionati prima 4096 bytes e poi tra questi viene scelto quello richiesto
- È possibile migliorare le prestazioni semplicemente evitando di "riselezionare" la riga ad ogni accesso se le posizioni sono consecutive
- Questo viene chiamato "fast page mode" (FPM) e l'incremento di prestazioni può essere significativo

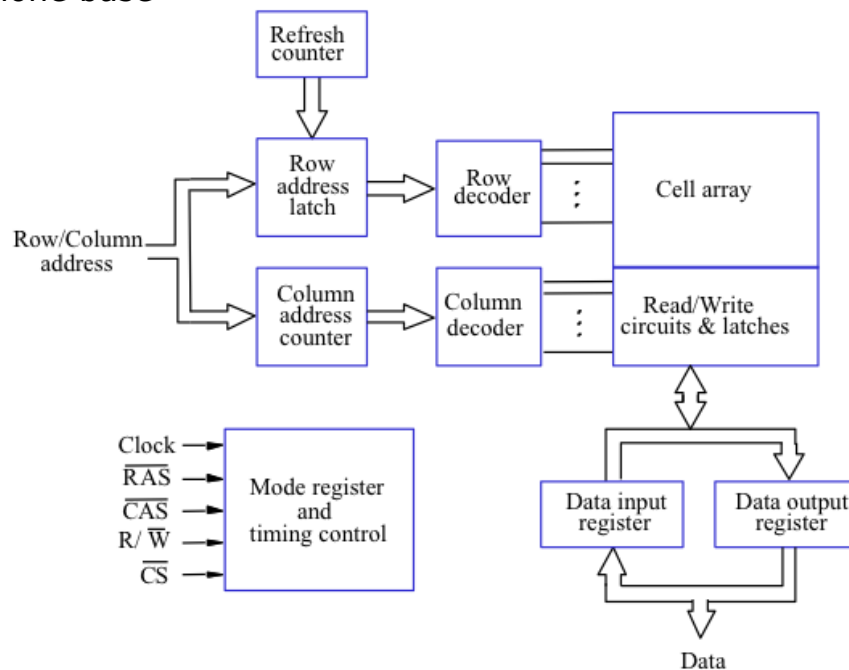
DRAM sincrone (SDRAM)

- Le DRAM visto prima sono dette "asincrone" perché non esiste una precisa temporizzazione di accesso, ma la dinamica viene governata dai segnali RAS e CAS

- Il processore deve tenere conto di questa potenziale "asincronicità"

- in caso di rinfresco in corso può essere fastidiosa

- Aggiungendo dei buffer (latch) di memorizzazione degli ingressi e delle uscite si può ottenere un funzionamento sincrono, disaccoppiando lettura e scrittura dal rinfresco e si può ottenere automaticamente un accesso FPM pilotato dal clock
- Organizzazione base



Velocità e prestazione

- **Latenza:** tempo di accesso ad una singola parola
 - è la misura "principe" delle prestazioni di una memoria
 - da una indicazione di quanto il processore dovrebbe poter aspettare un dato dalla memoria nel caso peggiore
- **Velocità o "banda":** velocità di trasferimento massima in FPM
 - molto importante per le operazioni in FPM che sono legate all'uso di memorie cache interne ai processori
 - è anche importante per le operazioni in DMA, posto che il dispositivo periferico sia veloce

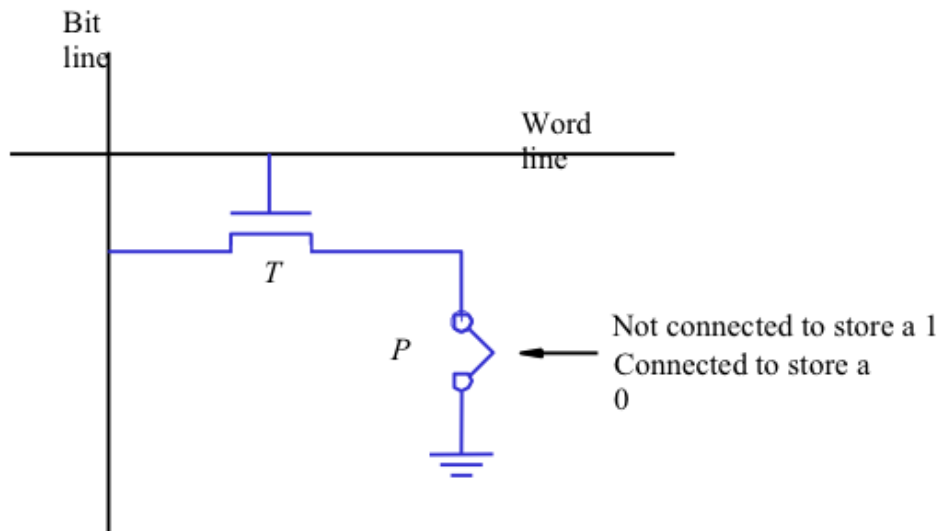
Double-Data-Rate SDRAM (DDR-SDRAM)

- DRAM statica che consente il trasferimento dei dati in FPM sia sul fronte positivo che sul fronte negativo del clock
- Latenza uguale a una SDRAM normale
- Banda doppia
- Sono ottenute organizzando la memoria in due banche separate
 - uno contiene le posizioni pari: si accede sul fronte positivo
 - l'altro quelle dispari: si accede sul fronte negativo
- Locazioni contigue sono in banche separate e quindi si può fare l'accesso in modo interlacciato

Read Only Memory (ROM)

- Memorie strutturalmente analoghe alle DRAM, ma con un "interruttore" al posto del condensatore:
 - se l'interruttore è aperto la cella contiene 1
 - se l'interruttore è chiuso la cella contiene 0

- Richiedono un processo "speciale" di scrittura che consente di aprire o chiudere l'interruttore



Tipi di ROM

- Una ROM "normale" viene scritta in fase di produzione:
 - richiede un nuovo "progetto" e una nuova "linea di produzione" anche solo per modificare un bit
 - va bene solo per grandissimi volumi di produzione
- PROM (Programmable-ROM):
 - al posto dell'interruttore viene inserito un fusibile
 - all'acquisto memorizza tutti "0"
 - bruciando fusibili posso introdurre "1" dove mi interessa
 - scrivibili una sola volta
 - serve un dispositivo "speciale" per bruciare i fusibili
- EPROM (Erasable PROM):
 - al posto dell'interruttore viene inserito uno speciale transistor che ha la capacità di funzionare o come un normale transistor (quando è alimentato conduce e quindi la cella memorizza "0") oppure di comportarsi come un circuito aperto e quindi di memorizzare "1"
 - la programmazione avviene iniettando della carica che viene intrappolata nei transistor speciali disabilitandoli
 - la cancellazione avviene per esposizione a luce ultravioletta che provoca il rilascio della carica intrappolata
 - programmazione e cancellazione devono avvenire in un dispositivo apposito (rimozione del chip)
- EEPROM (Electrically EPROM):
 - il principio è lo stesso delle EPROM normali
 - il transistor "speciale" può essere sia scritto che cancellato con diversi livelli di tensione
- alimentazione normale: lettura
- alimentazione più elevata: programmazione
- alimentazione ancora più elevata: cancellazione
 - posso operare senza rimuovere il chip
 - devo prevedere un sistema di alimentazione a tensione variabile

- Memorie Flash
 - sono un caso speciale di EEPROM
 - elevata densità
 - singola tensione di alimentazione per lettura, scrittura e cancellazione
 - lettura di singoli elementi
 - scrittura/cancellazione solo in blocchi
 - bassi consumi: adatte a sistemi a batteria

La CPU: organizzazione a Pipeline

Il problema dell'efficienza

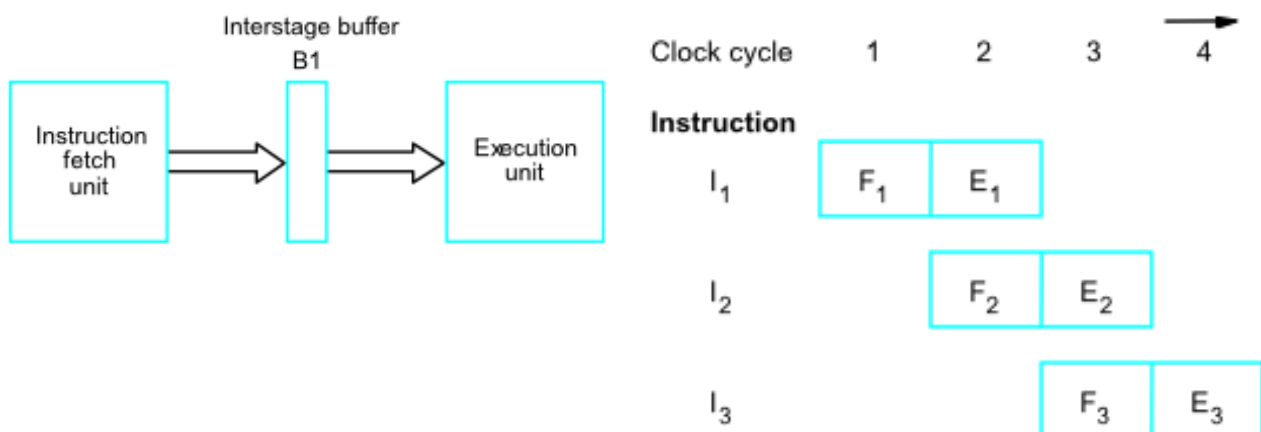
- I calcolatori moderni sono stati sviluppati con un'ossessione in mente: aumentare l'efficienza
- Questo comporta:
 - Riduzione dei colli di bottiglia (ad esempio tramite l'uso di memorie Cache)
 - Aumentare il parallelismo
- Il pipelining opera sul secondo fronte

Esecuzione sequenziale

- Come abbiamo visto le istruzioni sono eseguite tradizionalmente in maniera sequenziale
- La fase di fetch e di esecuzione si alternano
- Una possibile alternativa è che mentre eseguo un'istruzione mi preparo alla prossima
 - Modello catena di montaggio

L'idea del pipelining

- Introduco due unità:
 - Una specializzata nel fetch dell'istruzione
 - L'altra nell'esecuzione
- Le Unità comunicano tramite un buffer intermedio
- Il vantaggio è che a ogni ciclo di clock completo un'istruzione
- Notare come questo non abbassi assolutamente il tempo per eseguire un'istruzione (latenza)
- Quello che aumenta drasticamente è il numero di istruzioni al secondo (throughput)



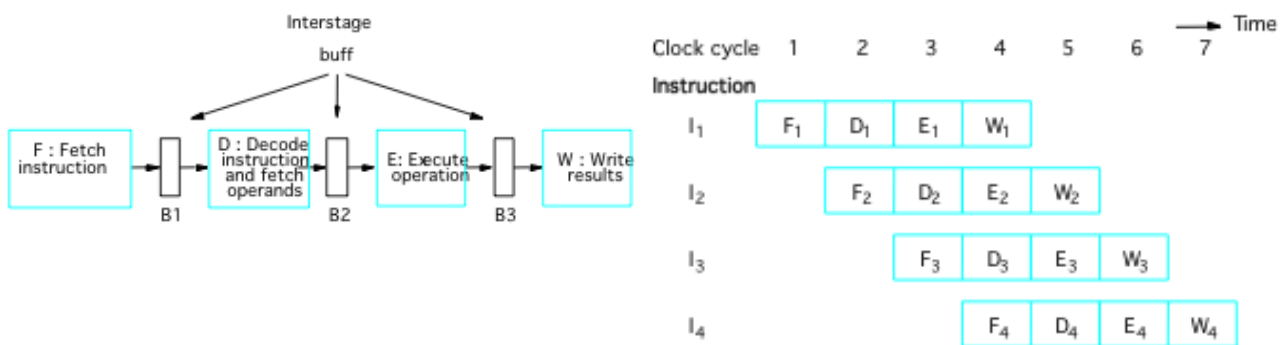
Problema fondamentale

- Perché il meccanismo funzioni tutti gli stadi devono andare alla stessa velocità
- Il ciclo di clock deve essere dimensionato sullo stadio più lento
- Il problema è che quando ho accessi in memoria (fase di fetch) posso avere bisogno di 10 cicli per portare il dato in memoria

- Questo problema viene risolto "quasi sempre" dalla cache di primo livello che da tempi di accesso pari a quelli dei registri
- Quando ho cache miss....beh gli altri stadi devono adeguare!!!!

Pipeline a più di due stadi

- Non c'è bisogno di limitarsi a due stadi per la pipeline....
- Posso avere due ulteriori stadi per prelevare gli operandi e per scrivere i risultati
 - In realtà nelle CPU moderne posso trovare molti più di 4 stadi
- Avere 4 stadi moltiplica per 4 il throughput ma anche la latenza (in termini di cicli di clock per fare un'istruzione)
- Tuttavia teniamo presente che il clock può essere reso più veloce se le operazioni sono più semplici

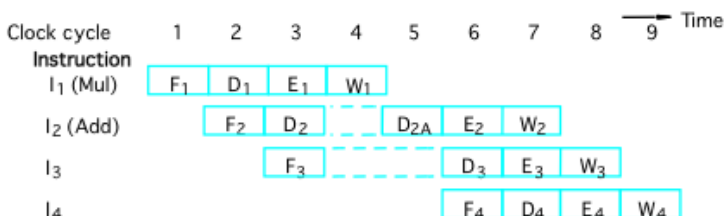


Performance della pipeline

- Quando la pipeline incontra un'operazione più lunga del previsto (esempio cache miss) si deve fermare tutto fino a che l'operazione non termina
- Questa situazione si chiama Hazard.
- Chiaramente la frequenza degli hazard ha un grande impatto sulla performance
 - La vera sfida del design è ridurre queste situazioni al minimo

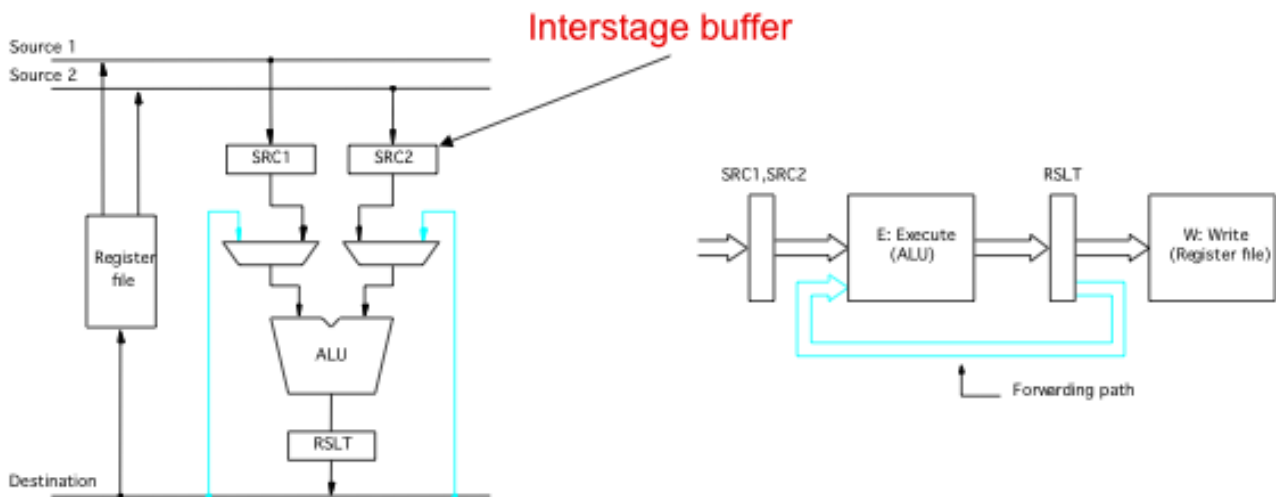
Data Dependency

- Consideriamo le seguenti istruzioni:
 MUL R2,R3,R4
 ADD R5,R4,R6
- La seconda ha bisogno del risultato della prima per eseguire



Data Dep.: operand forwarding

- In un caso come quello visto prima il problema è aspettare che il risultato sia scritto in un register file
- Un possibile rimedio è di modificare il datapath in modo che i risultati siano subito usati prima di scriverli nei registri



Strategia software

- In questo caso per evitare lo stallo il compilatore introduce delle NOP
 - Risparmiare sulla complessità dell'hardware vuol dire ridurre i ritardi e velocizzare il clock
 - Chiaramente le cose sono tutt'altro che semplici
 - In certi casi le data dependencies sono dovute a effetti collaterali (modifica dei flag)
- ```
MUL R2,R3,R4
NOP
NOP
ADD R5,R4,R6
```

### Instruction hazards

- La fase di fetch delle istruzioni è eseguita dal primo stadio della pipeline
- Per ridurre i conflitti con gli altri stadi della pipeline viene usata una cache dedicata alle istruzioni
- Spesso ci possono problemi anche con i salti
- Quando si fa un salto (non condizionato), può capitare che alcune istruzioni eseguite dagli stadi della pipeline siano sprecati (branch penalty)

### Early computation of branch address

- Una possibilità per ridurre il branch penalty può essere di avere una circuiteria avanzata che permetta di riconoscere il salto e calcolare l'indirizzo nelle fasi di decode

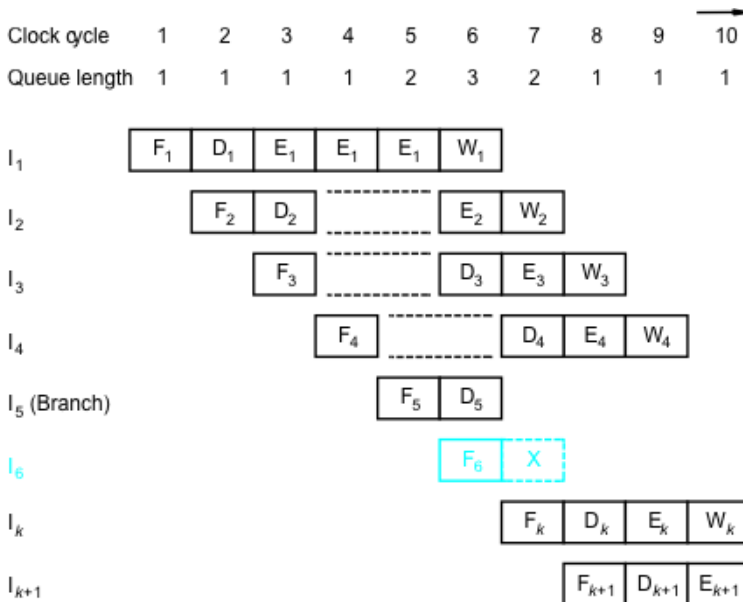
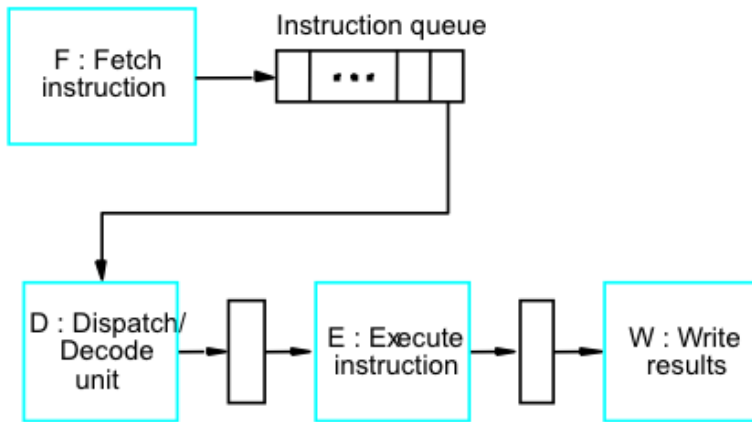
### Prefetching

- Durante la fase di fetch Cache miss e branch possono fare stall della pipeline
- Per ridurre il problema le moderne architetture adottano una sofisticata unità che fa il fetch di un po' di istruzioni e li mette in una coda
- Un'unità di dispatch prende le istruzioni dalla testa della coda e li manda alle unità di esecuzione
- L'unità di prefetch deve essere sofisticata in modo da riconoscere i salti e precaricare le istruzioni adeguate
- Avere una coda abbastanza lunga permette di disaccoppiare i due stadi della pipeline
  - Se la pipeline va in stallo per un data hazard l'unità di fetch può

continuare a prelevare istruzioni

- Se invece è l'unità di fetch ad andare in stallo possono continuare con l'esecuzione

Instruction fetch unit



- Al tempo 4,  $I_1$  genera stallo
- L'unità di fetch riempie la coda
- Al tempo 6, viene riconosciuto il branch e  $I_6$  viene buttata
- Tuttavia al tempo 7 non ho stallo dovuto a fetch (prendo dalla coda)
- Quindi le istruzioni vengono completate in perfetta sequenza senza perdere cicli (branch folding)

### Salti condizionali

- I salti condizionali combinano due difficoltà:
  - Data dependency
  - Instruction hazard
- Quindi hanno tutte le carte in regola per avere un impatto devastante sulle prestazioni
- ....Per fortuna esistono delle buone strategie per mitigarne l'effetto....

### Salti condizionali: delayed branch

- L'istruzione che segue un branch viene detta branch delay slot
- L'idea è di usare il compilatore per riordinare l'esecuzione di istruzioni in modo da riempire il branch delay slot con un'istruzione che va eseguita comunque (qualche che sia l'esito del jump)
- Analisi statistiche rivelano che questo si può fare nell'85% dei programmi !!

|      |            |       |
|------|------------|-------|
| LOOP | Shift_left | R1    |
|      | Decrement  | R2    |
|      | Branch=0   | LOOP  |
| NEXT | Add        | R1,R3 |

L'anticipo dell'operazione di shift mi permette di riempire tutti i branch Slot. Nel caso non riordinato dopo il branch dovrei fare stallo e discard di istruzioni



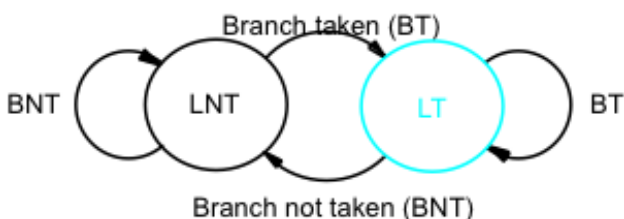
|      |            |       |
|------|------------|-------|
| LOOP | Decrement  | R2    |
|      | Branch=0   | LOOP  |
|      | Shift_left | R1    |
| NEXT | Add        | R1,R3 |

### Branch prediction

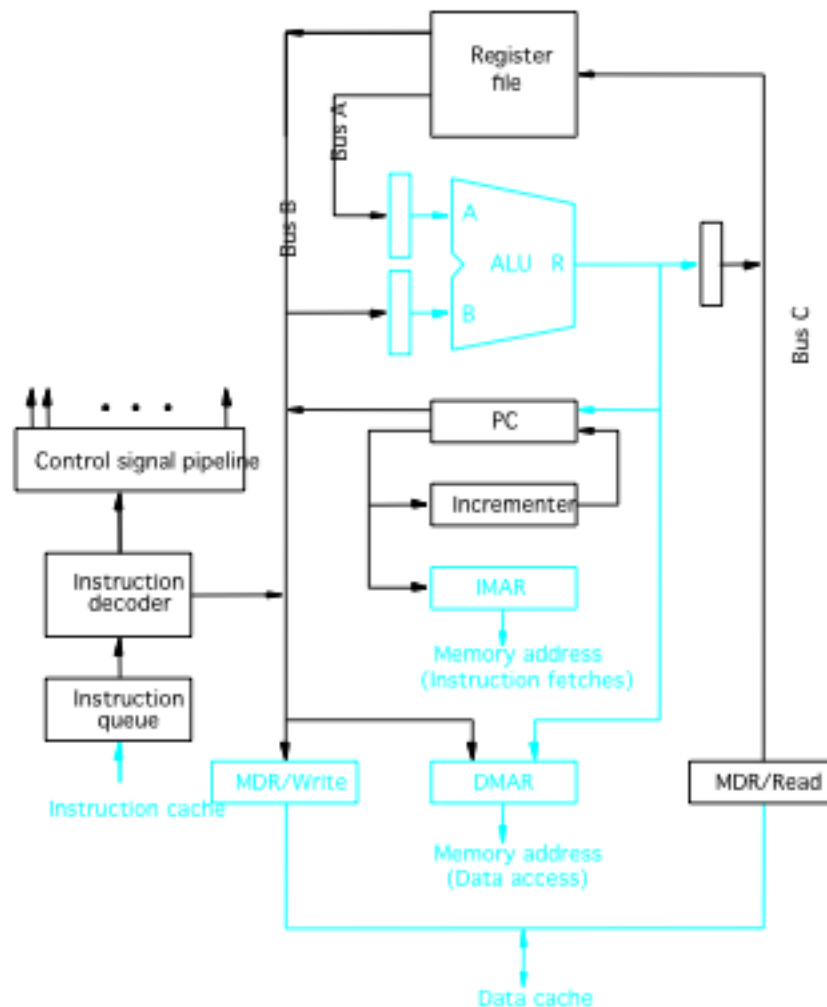
- Un'altra tecnica consiste nel cercare di prevedere l'esito del branch condizionale
- Ad esempio un branch all'inizio di un ciclo quasi spesso non viene preso, mentre un branch alla fine di un ciclo viene preso molto spesso
- In certe architetture (SPARC) il compilatore può usare un bit per indicare alla CPU qual'è la probabilità di prendere un branch
- Una volta che si fa una previsione si continua ad eseguire le istruzioni assumendo che un esito per il branching
- Siccome non si sa se questa esecuzione verrà richiesta o meno viene detta "speculativa"
  - Occorre fare attenzione all'aggiornamento di registri e memoria da parte di un'esecuzione speculativa (uso di registri temporanei)

### Branch prediction dinamico

- Negli esempi precedenti a ciascun branch si associa sempre la stessa predizione (branching statico)
- Complicando l'hardware è possibile anche fare delle predizioni dinamiche, associando una macchina a state a ciascun branch
- Il caso più semplice è prevedere sulla base dell'ultima esecuzione del branch (occorre un bit per ogni branch)
- Funziona molto bene all'interno di cicli
  - Si possono fare anche delle macchine più complesse



### Progetto per il piepeling



## Note

- ALU arricchita con i buffer per la comunicaz. interstage
- I registri MAR vengono divisi (IMAR, DMAR) per permettere di accedere a instruction e data cache
- Il PC viene connesso direttamente all'IMAR (il trasferimento del PC nell'IMAR) può avvenire in parallelo alle op. delle ALU
- In DMAR possiamo inserire dati sia dal register file che dall'ALU per supportare le varie modalità di indirizzamento
- MDR è diviso in due in modo da poterci operare parallelamente prelevando i dati direttamente i dati direttamente dal register file
- I registri SRC1, SRC2 e RSLT sono aggiunti sulle ALU per facilitare l'operand forwarding (per risolvere le dipendence dei dati)
- L'IR è stata sostituita da una coda di istruzioni
- I segnali di controllo generati vengono bufferizzati per i vari stadi della pipeline
- Operazioni che è possibile eseguire in parallelo
  - Leggere un'istruzione dalla Cache
  - Incrementare il PC
  - Decodificare un'istruzione
  - Leggere e scrivere nella cache dati
  - Leggere il contenuto di numero di registri fino a due
  - Scrivere in un registro
  - Effettuare un'operazione di ALU

- Ad esempio durante un ciclo di clock possiamo
  - Scrivere il risultato dell'istruzione I1 in un registro
  - Leggere gli operandi di I2
  - Decodificare I3
  - Prelevare I4 e incrementare il PC

### **Architetture super-scalari**

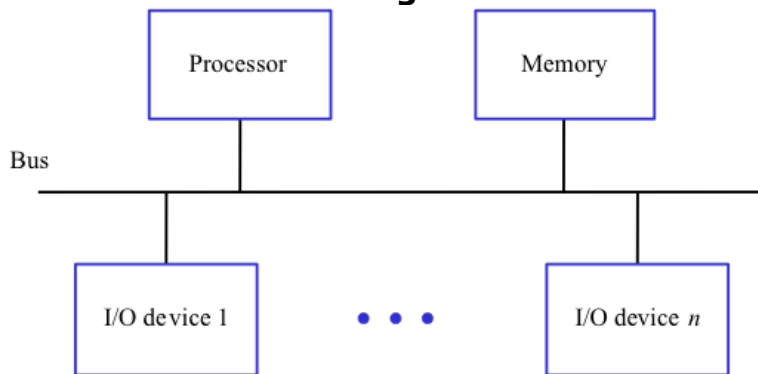
- L'idea delle architetture multiscalari è di avere più pipeline che eseguono in parallelo, ciascuna specializzata su diverse operazioni (es. pipeline per le operazioni float e pipeline per le operazioni intere)
- Unità di fetch e di dispatch devono operare su più istruzioni insieme

# Controllo dell' "I/O"

## Cosa vuol dire "I/O" ?

- Un elaboratore deve comunicare con il mondo esterno
- Le funzioni di ingresso e uscita dei dati (Input/Output – I/O) sono fondamentali per il buon funzionamento di una architettura
- I dispositivi di I/O sono (in genere) molto più lenti del processore e della memoria centrale
- Un elaboratore è costruito intorno a una struttura di comunicazione tra i diversi dispositivi chiamato "bus"

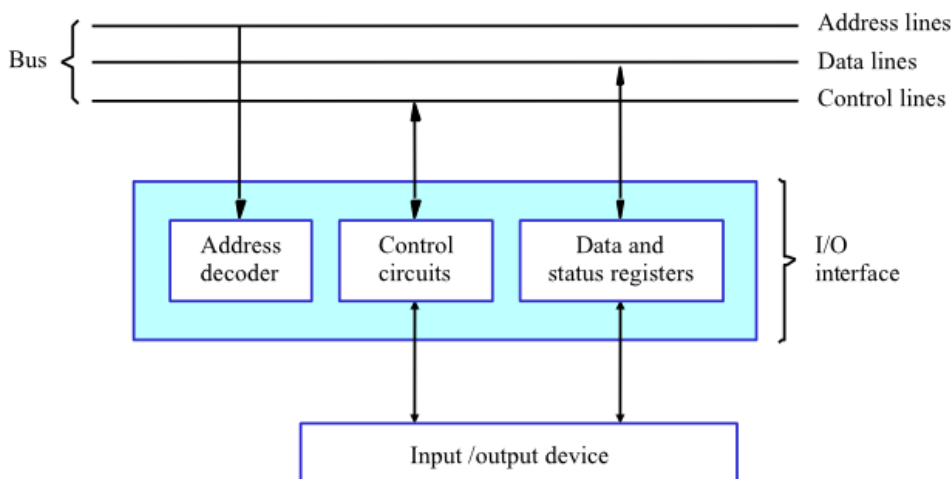
## Architettura a bus singolo



## Struttura di un bus

- Elevato parallelismo (negli elaboratori)
- Linee separate per:
  - trasferimento dei dati
  - indirizzamento dei dati
  - controllo del bus (es. chi scrive e chi legge)
- I dispositivi connessi ad un bus (incluso il processore) devono avere
  - una interfaccia di accesso al bus
  - un protocollo per usarlo correttamente
- L'indirizzamento dei diversi dispositivi può essere integrato (memory mapped I/O) oppure separato per ogni dispositivo

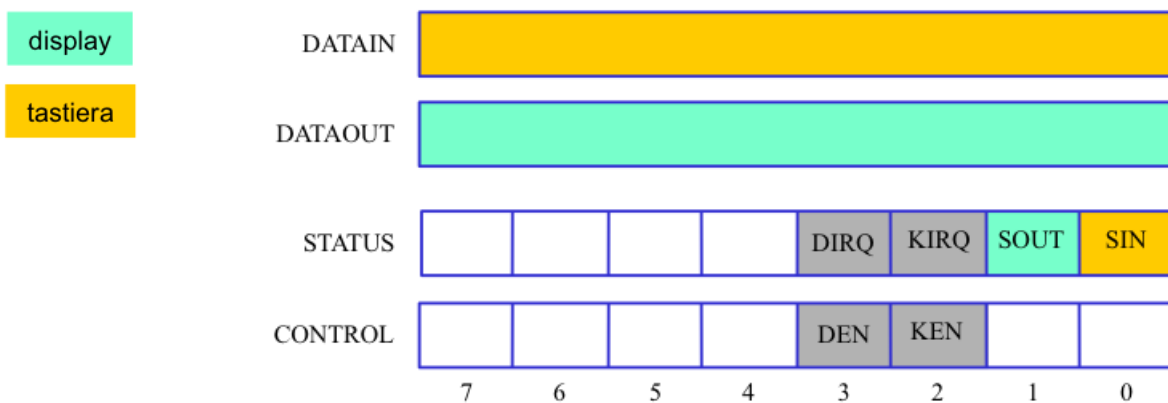
## Struttura dell'interfaccia di un dispositivo di I/O



## I/O controllato da programma

- Normalmente chiamato "polling"
- E' il modo più intuitivo e semplice di concepire l'I/O (già visto come esempio in assembler)
- Un programma controlla lo stato di un dispositivo ed effettua la lettura/scrittura ("move") del dato quando il dispositivo è disponibile
- Riprendiamo l'esempio già fatto e vediamo come si può leggere una intera linea di testo da una tastiera e farne l'eco su un display con interfaccia integrata come segue

## Registri di controllo e dati di tastiera e display



## Programma per leggere una linea dalla tastiera (memorizzarla e farne l'eco sul terminale)

|       |           |                |                                            |
|-------|-----------|----------------|--------------------------------------------|
| BEGIN | Move      | #MEM, R0       | carica l'indirizzo di memorizz.            |
| WAITK | TestBit   | #0, STATUS     | controlla SIN                              |
|       | Branch=0  | WAITK          | il loop se il car. non è disponibile       |
|       | MoveBy    | DATAIN, R1     | carica il dato in R1 (e resetta SIN)       |
| WAITD | TestBit   | #1, STATUS     | controlla SOUT                             |
|       | Branch=0  | WAITD          | loop se il display non è disp.             |
|       | MoveBy    | R1, DATAOUT    | fa l'eco e resetta SOUT                    |
|       | Move      | R1, (R0)       | memorizza il car.                          |
|       | Add       | #4, R0         | incrementa il puntatore                    |
|       | Comp      | #\$0D, R1      | Carriage Return??                          |
|       | Branch!"# | WAITK          | altro car. se la linea non è finita        |
|       | Move      | #\$0A, DATAOUT | LF sul display                             |
|       | Call      | PROCESS        | chiama una routine per processare la linea |

## I/O controllato da programma

- Inefficiente (loop infinito per aspettare la disponibilità del dispositivo)
- In caso di indisponibilità (o errore, o dispositivo inesistente) si blocca l'intero sistema
- Come gestisco le contese tra programmi che devono accedere allo stesso dispositivo?
- Es. gestione del mouse per spostare il puntatore ed eco della tastiera

## Interrupts

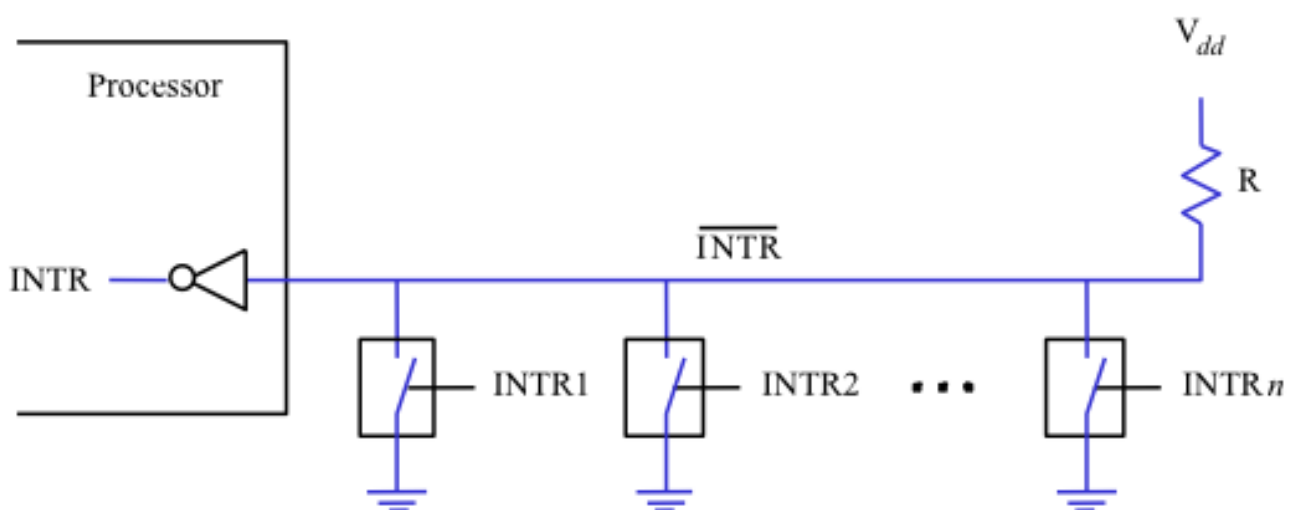
- Una gestione più efficiente dell'I/O si ottiene tramite dei segnali di richiesta di attenzione (o interruzione di ciò che il processore sta facendo, da cui interrupt) da parte dei dispositivi periferici
- Si può a tale scopo usare una o più linee dedicate della parte di controllo del bus
- Il segnale di interrupt forza il processore ad eseguire una speciale routine di "Service-Interrupt"

## Service-Interrupt

- E' una (quasi) normale subroutine
- Bisogna salvare lo stato del programma in esecuzione (si usa in genere lo stack)
- Servire un interrupt costa tempo, oltre che per l'esecuzione dell'I/O anche per il salvataggio e il ripristino dello stato
- Un interrupt è in generale un modo per trasferire il controllo da un programma ad un altro in modo non predefinito
- Come si possono gestire molti dispositivi con gli interrupt?

## Segnale di Interrupt

- E' in genere un linea che assume valore negato quando è presente almeno un interrupt
- Assumiamo che ci sia una porta NOT che ne complementa il valore



## Gestione degli interrupt

- Un interrupt fa invocare la *Service-Interrupt*
- Cosa succede se arriva un altro interrupt durante il servizio al primo?

- Interrupt disabilitati
- Interrupt annidati a priorità diversa
- I processori semplici hanno un solo livello di interrupt e l'inizio di un servizio provoca la disabilitazione di ulteriori interrupt fino al termine del servizio stesso
- Esiste un apposito bit (Interrupt Enable IE) nel registro di stato del processore
- In un sistema semplice (non i PC!) una tipica sequenza di interrupt è la seguente
  1. il dispositivo inoltra una richiesta attivando INTR
  2. il processore interrompe il programma in esecuzione in un punto opportuno e salva lo stato
  3. il processore disabilita ulteriori interrupt (IE=0)
  4. il dispositivo viene avvertito (es. segnale di ACK) che la richiesta è stata ascoltata e disattiva INTR
  5. l'azione richiesta viene svolta da Service-Interrupt
  6. gli interrupt vengono riabilitati (IE=1) e il processore riprende l'esecuzione del programma interrotto
- Cosa succede con tanti dispositivi?
  - Tante linee di interrupt?
  - Interrupt service diverse?
- Esistono molti modi diversi per gestire interrupt multipli, spesso mescolati tra loro nello stesso processore
- Bisogna anche gestire l'eventualità di interrupt contemporanei, quindi avere un sistema di arbitraggio per decidere chi servire prima

### **Interrupt vettoriale**

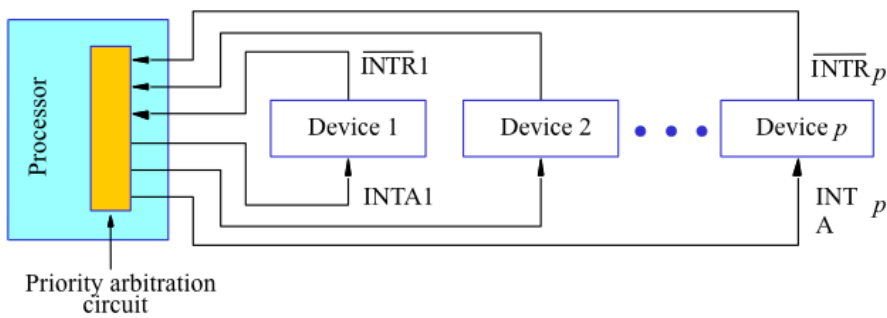
- La *Service-Interrupt* si limita a leggere da un registro opportuno dell'interfaccia del dispositivo un indirizzo che punta alla routine di servizio specifica del dispositivo
- L'indirizzo è in genere un offset su 4-8 bit che viene integrato dal processore con l'indirizzo base del vettore di routine di servizio
- Le routine specifiche dei dispositivi sono in genere chiamate "*device driver*"

### **Interrupt annidati**

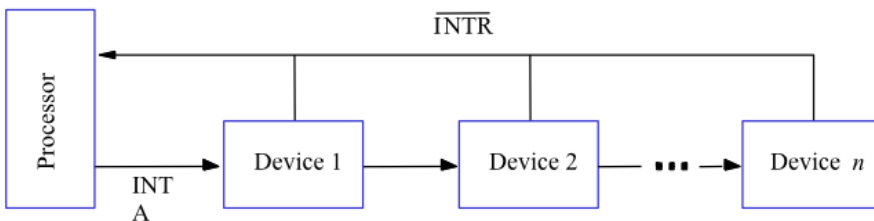
- Nei sistemi complessi (PC, etc.) un solo livello di interrupt non è sufficiente
- Esistono diverse priorità, un INTR a priorità più alta può interrompere una *Service-Interrupt* a priorità più bassa come se fosse un normale programma
- A seconda dei processori si possono avere fino a 8-16 livelli di priorità
- Es. un sensore di allarme ha priorità più elevata di un accesso a disco

### **Collegamento dei dispositivi**

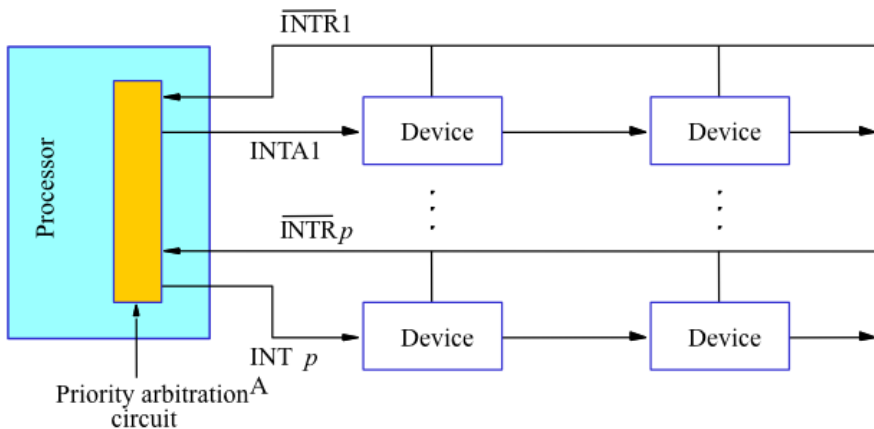
- Linee individuali:
  - semplice gestione delle priorità e dell'arbitraggio
  - esplosione del numero di linee all'aumentare dei dispositivi



- Linea condivisa:
  - gestione della priorità molto complessa (se non impossibile), arbitraggio "rigido" e lento
  - una sola linea connessione a "daisy chain,"



- Linee condivise ma separate per priorità
  - caratteristiche intermedie tra i casi precedenti



### Daisy Chain

- Collegamento a "festone" dei dispositivi
- Il processore accede ai registri di controllo in modo sequenziale, se il primo non è attivo passa al secondo e così via
- In pratica i primi dispositivi hanno priorità (esistono metodi per assegnare questa priorità a rotazione)
- L'accesso a tanti registri può rendere il sistema piuttosto lento
- Non si collegano mai più di alcune unità con questo metodo

### Interrupt

- |               |            |
|---------------|------------|
| • Main        | • ISR      |
| movl \$0, EOL | Pushl %EAX |
| movb \$4, %BL | Pushl %EBX |

```
orb %BL, CONTROL
STI
```

```
leal PNTR, %EAX
Movl DATAIN, %BL
Movl %BL, (%EAX)
Addl $4, %EAX
Movl %eax, PNTR
cmpb 0DH, %BL
JNE Fine
Movb $4, %BL
xorb %BL,CONTROL
MOV $1, EOL
FINE: Pop %EBX
 Pob %EAX
 IRET
```

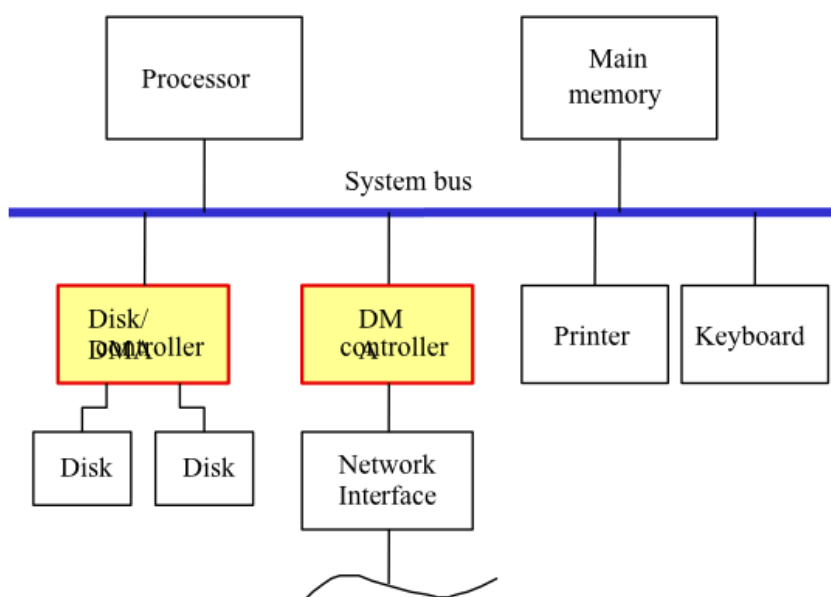
### Interrupt e Sistema Operativo

- Nei sistemi moderni un programma utente non accede mai direttamente a una periferica
- Il SO si fa carico della gestione delle periferiche per conto dell'utente
  - gli utenti non si devono preoccupare di come funzionano le periferiche
  - gli utenti non hanno il privilegio di accesso alle periferiche e quindi hanno meno possibilità di fare danni!
- Un interrupt altera la normale programmazione dei processi (il principale compito di un SO) inserendo le routine di Service-Interrupt opportune

### Direct Memory Access (DMA)

- Spesso i dispositivi periferici devono caricare molti dati in memoria prima che il processore li possa (o voglia) elaborare
- Passare ogni singolo byte da una periferica (lenta) al processore (veloce) alla memoria (lenta) è un enorme spreco di risorse
- Se dotiamo i dispositivi di capacità di indirizzamento e di un protocollo per la gestione del bus si può evitare di coinvolgere il processore in alcune fasi di I/O

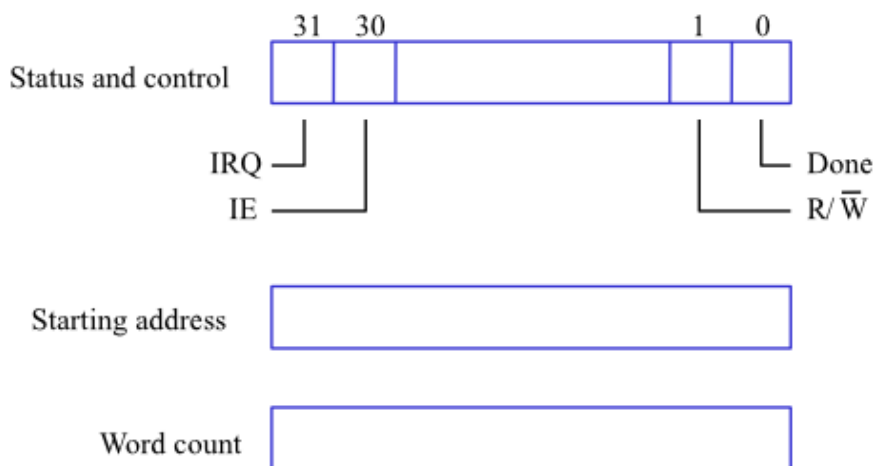
### Architettura con uso di DMA



## Interfaccia DMA

- La parte del dispositivo che gestisce l'accesso al bus in questo caso viene chiamato "DMA controller"
- Prima di poter accedere la bus per scrivere o leggere dati il DMA controller deve interagire con il processore (e gli altri dispositivi) per controllare lo stato del bus e prendere le opportune istruzioni
  - indirizzo iniziale di lettura/scrittura
  - numero di word da leggere scrivere
  - etc.

## Interfaccia e registri di un DMA controller



## Accesso in memoria in DMA

- Uno degli scopi del DMA è lasciare libero il processore di fare "altro" mentre una periferica accede alla memoria ...
- Mentre lavora il processore accede in memoria ...
- Le periferiche in DMA hanno normalmente priorità sul processore nell'accesso alla memoria
- "Cycle stealing": il DMA controller "ruba" cicli di accesso alla memoria per caricare/scaricare pochi dati
- "Burst mode": il DMA controller "negozia" un l'accesso alla memoria per un tempo sufficiente a caricare/ scaricare un blocco (grande) di dati

## Lettura/Scrittura ↔ Master/Slave

- Nelle comunicazioni sul bus c'è sempre un dispositivo che "scrive" sul bus, cioè trasmette e (almeno) uno che "legge" dal bus, cioè riceve
- Uno dei dispositivi (indipendentemente che legga o scriva) ha il controllo del bus e si chiama **master**
- L'altro (o gli altri) seguono i comandi del master e vengono chiamati **slave**
- Il master può cambiare dinamicamente durante l'uso del bus e, in generale è il dispositivo che ha fatto richiesta di effettuare la comunicazione

## Tecniche di arbitraggio del bus

- In DMA è necessario risolvere le contese per l'uso del Bus (oltre che della memoria, ma se ho risolto la contesa sul bus, l'ho risolta automaticamente

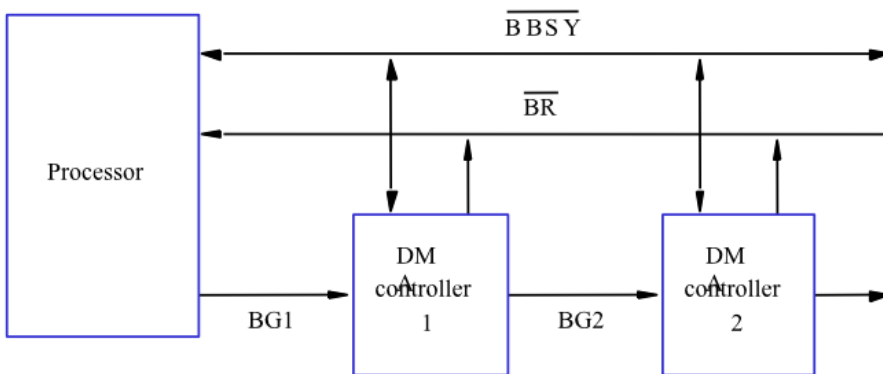
anche per la memoria)

- è necessario un "arbitraggio":  
chi vince diventa master
- Centralizzato: più semplice ma richiede o un dispositivo ad-hoc o l'uso del processore (poco efficiente), se l'arbitro non è disponibile si blocca tutto
- Distribuito: Più robusto, veloce e affidabile, ma più complesso

### Arbitraggio Centralizzato

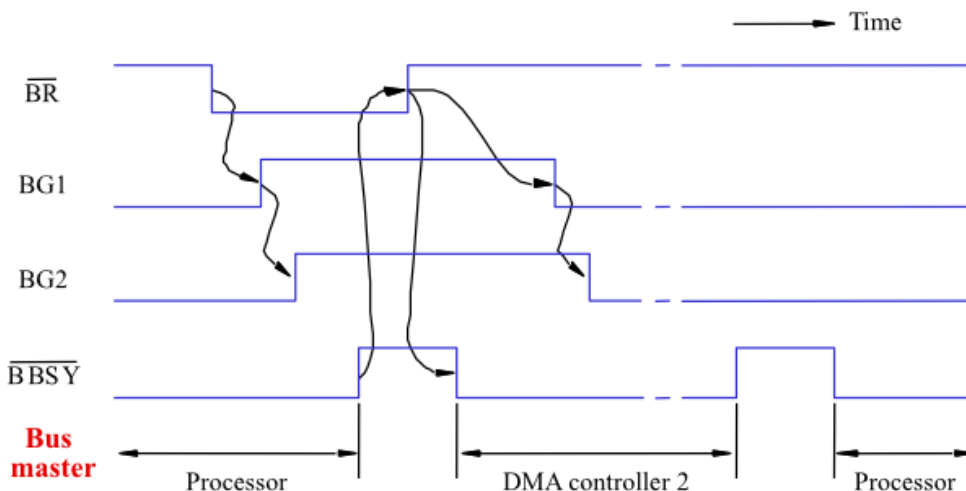
- Il processore è normalmente il master del bus
- Il processore raccoglie le richieste
- Mediante una qualsiasi logica risolve le contese
- Assegna l'uso del bus al dispositivo D che ne ha fatto richiesta
- D diventa a sua volta master
- D usa il bus per le sue comunicazioni fino ad un tempo massimo prefissato
- D rilascia il controllo del bus, che torna in generale al processore, salvo altre richieste pendenti

### Arbitraggio centralizzato con BR e BBSY



BG = Bus Grant

### Sequenze causa/effetto per l'accesso al bus su richiesta di DMA-C2



### Arbitraggio distribuito

- è necessaria avere dei circuiti logici di arbitraggio su ciascuna interfaccia DMA, in modo che alla fine dell'arbitraggio tutti sappiano qual'è stato l'esito

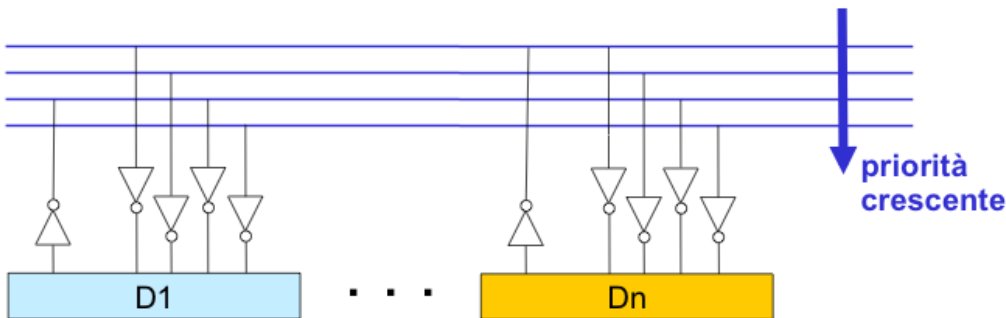
- Si usano in genere degli appositi fili di controllo su cui i dispositivi pongono il loro "codice" o indirizzo
- Le linee sono analoghe a quelle degli interrupt su cui il valore attivo è quello basso, in modo da ottenere l'OR equivalente delle richieste

### Arbitraggio distribuito

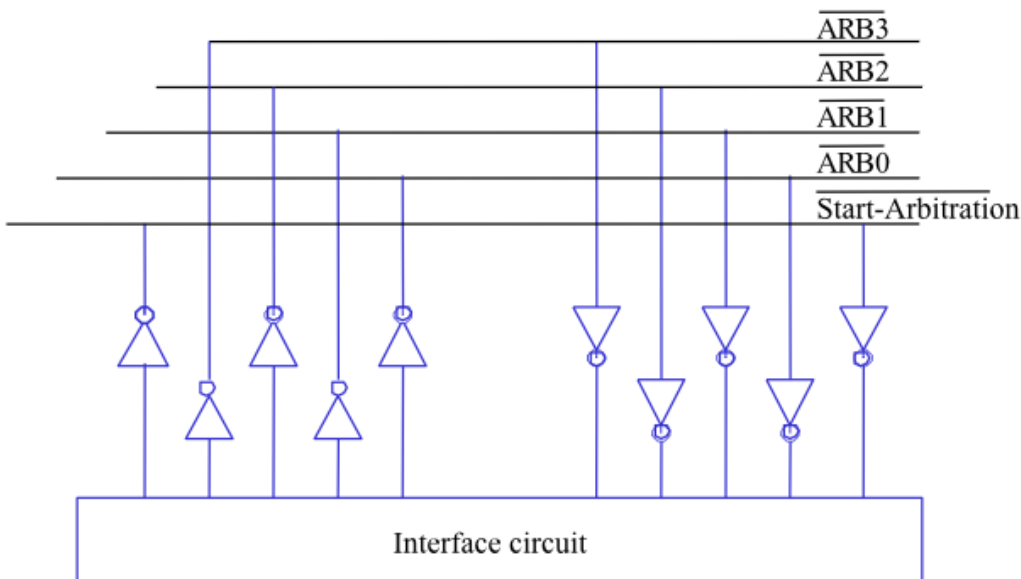
- Esistono molte tecniche diverse di arbitraggio distribuito
- L'interfaccia di arbitraggio è in grado di leggere e scrivere contemporaneamente sulle linee di arbitraggio
- I valori scritti vengono modificati in funzione di quanto viene letto per realizzare la funzione di arbitraggio
- Dopo alcuni cicli di scrittura lettura il valore delle linee si deve stabilizzare e identifica il vincitore

### Es. : 1 linea per dispositivo

- Le linee hanno priorità diversa
- Vince la contesa il dispositivo sulla linea a priorità più alta



### Arbitraggio distribuito (4 fili, 16 indirizzi)



- Tutti scrivono il proprio indirizzo
- Partendo dal bit più significativo se riscontrano una differenza con il proprio indirizzo mettono a zero il bit corrispondente e tutti quelli a priorità più bassa
- Il valore sulle linee può "oscillare" per un numero di cicli che è al massimo

pari al numero di fili, poi si stabilizza al valore uguale all'indirizzo di chi ha vinto, che riconosce il proprio indirizzo e diventa master del bus

# Comunicazioni nell'elaboratore (e oltre)

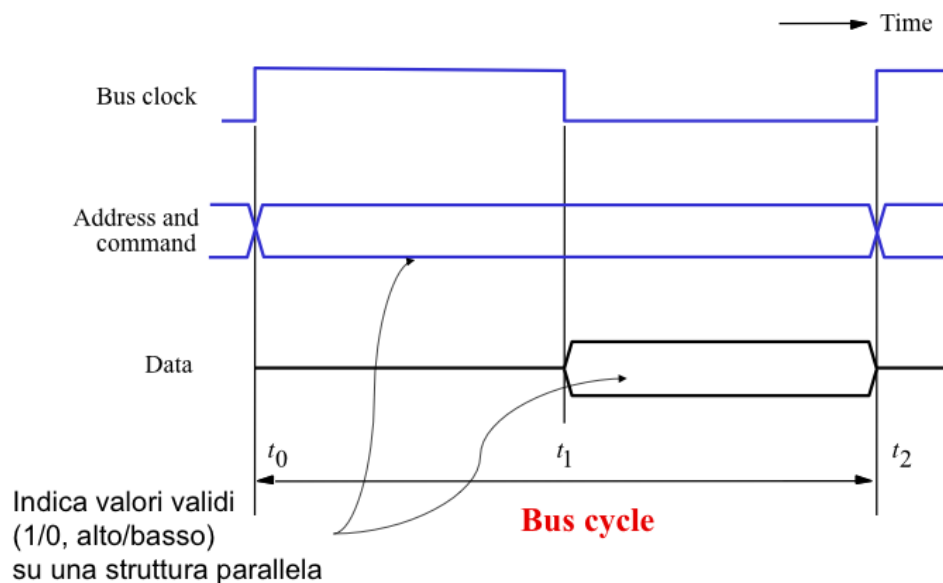
## Tipi di Bus

- Il bus è una struttura di comunicazione
- Esistono problemi di tempi di propagazione del segnale e quindi di sincronizzazione dei dispositivi
- Bus sincroni: funzionamento governato da un segnale di clock
- Bus asincroni: funzionamento governato solamente dall'interazione tra master e slave

## Bus sincrono

- Il processore distribuisce un segnale di temporizzazione sulle cui transizioni avvengono gli eventi
- Esiste un "ciclo del bus" che consente di sapere quando gli indirizzi, controlli e dati sono validi
- Indirizzi e controlli vengono fissati all'inizio de ciclo
- I dati sono validi nella seconda metà del ciclo
- La durata del ciclo (e quindi la velocità del bus) dipendono dai dispositivi e dai tempi di latenza/ propagazione

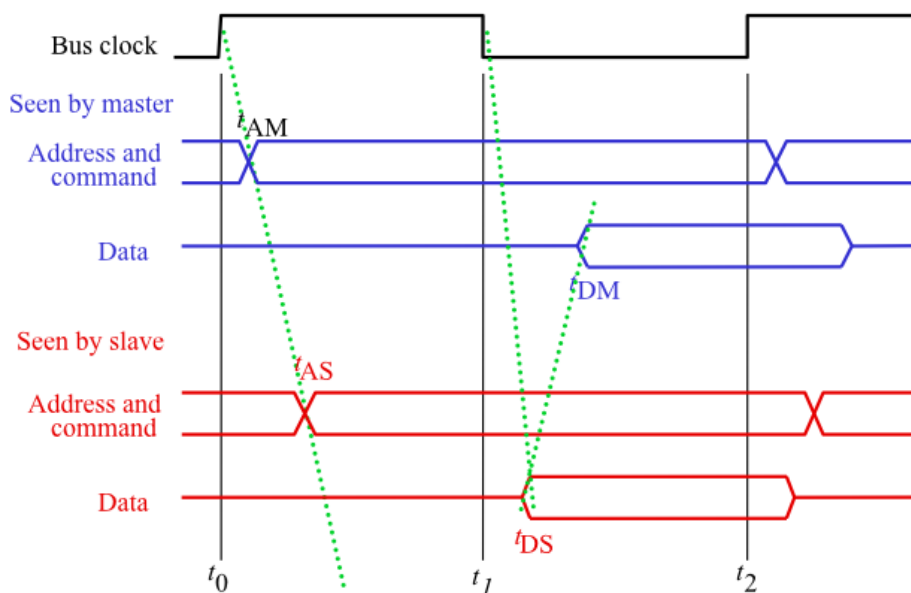
## Bus sincrono: funzionamento base



## Latenza e propagazione

- I tempi di propagazione fanno sì che i dispositivi "vedano" situazioni diverse del bus, o meglio le condizioni imposte da un dispositivo risultano valide per gli altri dopo un tempo legato alla propagazione dei segnali
- Questo è un vincolo di prestazioni insormontabile per un bus, quindi per avere prestazioni elevate è necessario o avere distanze molto piccole o avere strutture più sofisticate che consentano di compensare i ritardi di propagazione

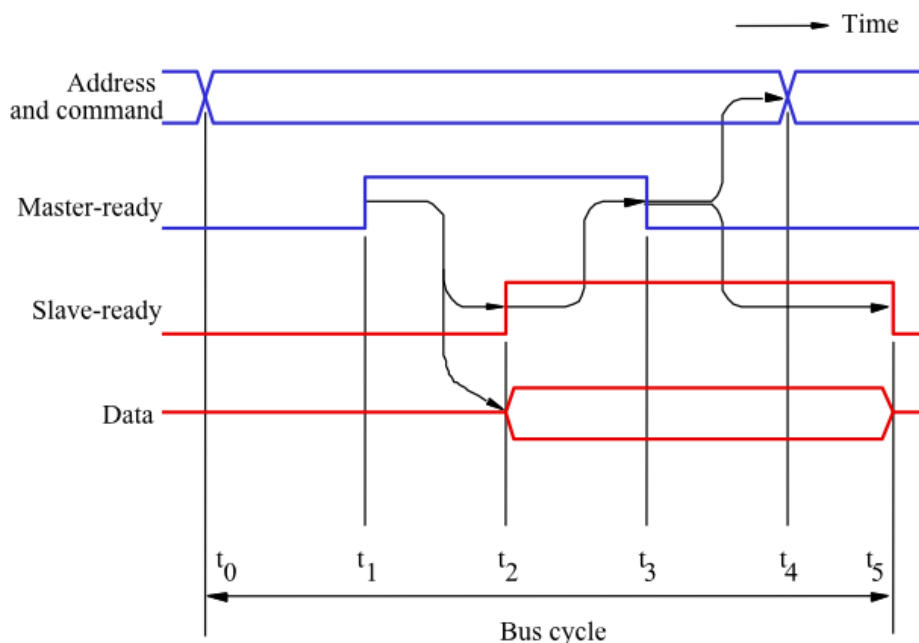
## Bus sincrono: effetto del tempo di propagazione



### Bus asincrono

- Al posto di un segnale di clock che garantisce il corretto funzionamento tenuto conto del massimo ritardo si usano due segnali di "handshake"
- Handshake (letteralmente "stretta di mano"): tecnica per sincronizzare due dispositivi o, più in generale, un trasmettitore ed un ricevitore
- Master ready: indica quando il master del bus ha inviato segnali ed indirizzi
- Slave ready: indica quando il dispositivo slave ha decodificato correttamente comandi e indirizzi

### Ciclo di un bus asincrono

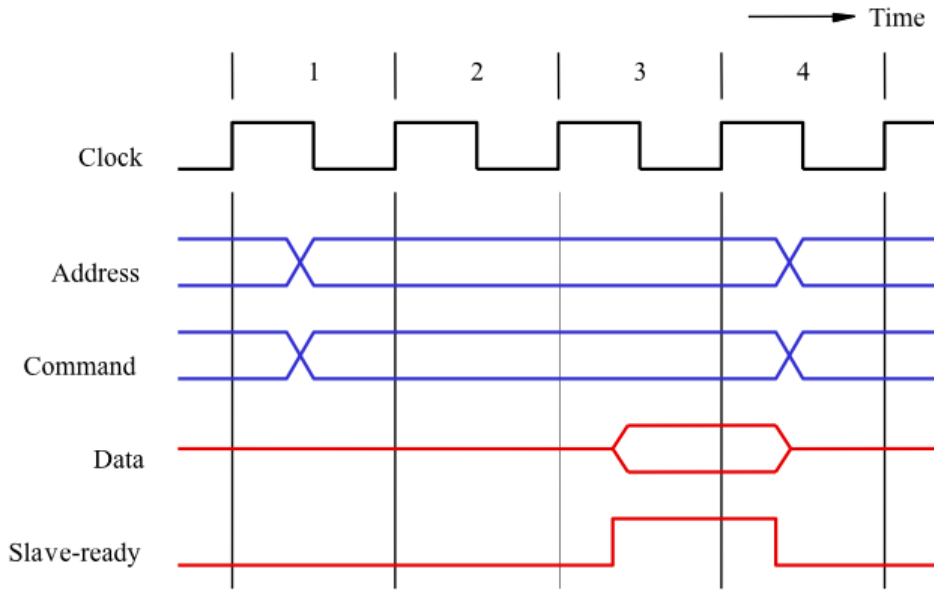


### Altre strutture (intermedie)

- E' possibile mettere insieme i vantaggi dei bus sincroni e asincroni usando un clock a velocità elevata ed un solo segnale di "handshake" che comunica quando lo slave ha letto/scritto i dati
- Consente di fare comunicazioni su più colpi di clock, consentendo la

coesistenza di dispositivi veloci e dispositivi lenti

### Bus sincrono "multiclock"



### Bus e ... Bus

- Un elaboratore può avere più di un bus di comunicazione
- Tipicamente esiste un bus "privato" del processore, che serve a far comunicare le diverse parti (ALU, Cache, ...) realizzate sul singolo circuito integrato
- All'esterno del processore è necessario comunicare con gli altri dispositivi e quindi le comunicazioni devono seguire uno standard

### Esempio di una architettura "moderna"

