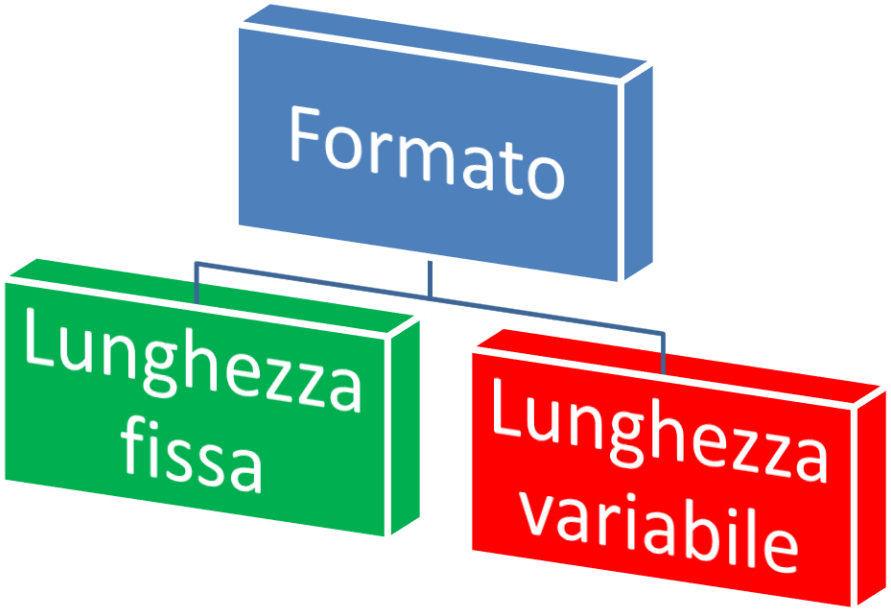
Architettura
degli Elaboratori Elettronici

Dott. Franco Liberati
liberati@di.uniroma1.it

ISTRUZIONI

Generalità

- Una **istruzione** è una stringa binaria che indica all'elaboratore elettronico dei compiti da svolgere
- Una istruzione è suddivisa in sottostringhe denominate **campi**
- La suddivisione in campi individua il **formato dell'istruzione**



ISTRUZIONI

Generalità: campi

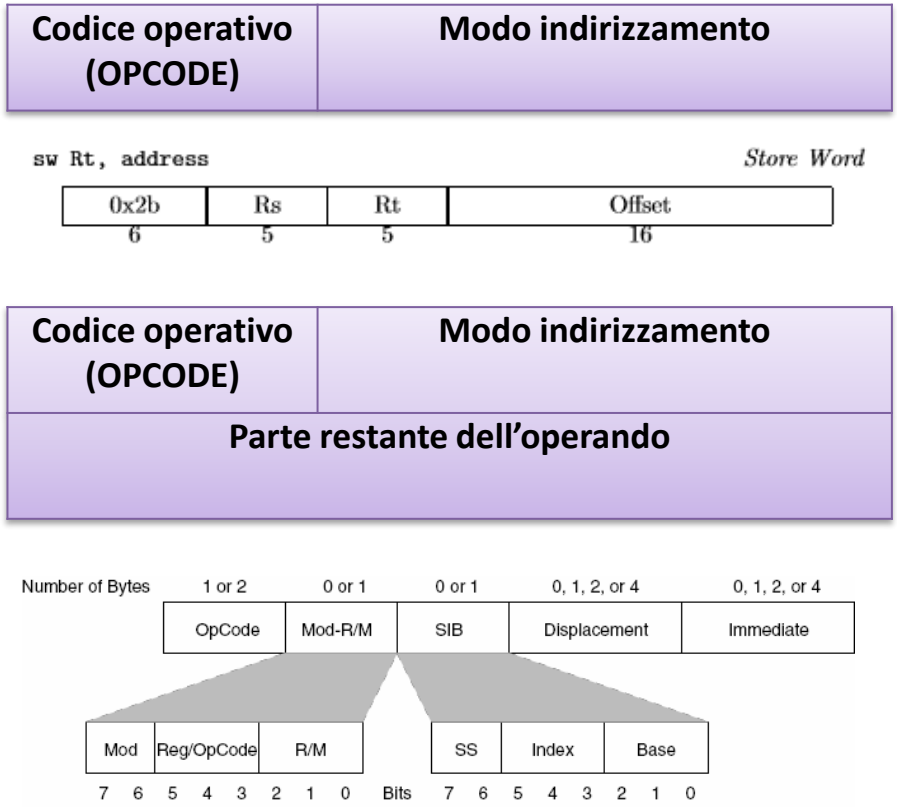
- I campi principali sono:
  - il **codice operativo** (o OPCODE), che specifica il tipo di operazione da eseguire (addizione, trasferimento dati,…)
  - l'**operando**, che indica il dato su cui devono essere effettuate le operazioni indicate dal codice operativo
  - L'operando può essere un valore numerico (come avviene nell'indirizzamento immediato) o, come spesso accade, si ha un **riferimento**: cioè un indirizzo di memoria in cui è immagazzinato un operando (indirizzamento diretto) o una etichetta che specifica un registro

Codice operativo (OPCODE)	Operando/riferimento
LW	$t0,133
ADD	$t0,$t1,$t2
MOVE	$t0,$t1
LW	$t0,($a0)
LH	$t1,4-($a2)

ISTRUZIONI

Generalità: formato

- Il **formato a lunghezza fissa** prevede un insieme di istruzioni (*instruction set*) con una dimensione predefinita (una sottoclasse di questa sono le **istruzioni a referenziamento implicito**, cioè quelle dotate di solo opcode)
- In alternativa, un set di istruzioni può avere una **lunghezza variabile**: in relazione al tipo di istruzione cambia la dimensione
  - Un istruzione a lunghezza variabile di solito ha i **bit in eccesso** – cioè non rappresentabili nella parola - ospitati nella parola successiva (richiede più accessi in memoria)



ISTRUZIONI

Generalità: formato Intel x86 a lunghezza variabile

- I processori intel X86 hanno un **formato a lunghezza variabile**
- Dopo l'opcode ci sono dei campi che specificano quanti bit appartengono al campo MODE

MOVE[b,w,l,q] sorgente, destinazione

Istruzione	Valore in RAX
movb 100,%RAX	0000000000000010
movw 100,%RAX	0000000000003210
movl 100,%RAX	0000000076543210
movq 100,%RAX	fedcba9876543210

IND	VAL
100	10
101	32
102	54
103	76
104	98
105	ba
106	dc
107	fe

ISTRUZIONI

Generalità: formato del MIPS a lunghezza fissa

- Il MIPS ha un **formato a lunghezza fissa a 32 bit**
- Qualora si usi un indirizzamento assoluto (o immediato) in cui il riferimento (o l'operando) richieda più di 16bit l'istruzione è suddivisa in due istruzioni elementari che consentono il riempimento dell'operando/indirizzo in un registro

li $t0,3000000000

10110010 11010000 01011110 00000000

B2 50 5E 00

Diventa

li $at, B250

In $at

10110010 11010000 00000000 00000000

ori $t0,$at,5E00

In $t0

10110010 11010000 01011110 00000000

ISTRUZIONI

Linguaggio macchina

- Le istruzioni sono eseguite quando sono scritte in **linguaggio macchina** (nei primi elaboratori esisteva solo questo tipo di linguaggio)

Salto incondizionato in MIPS [**J ciclo** (dove *ciclo* è l'indirizzo in memoria 68768]

Istruzioni a lunghezza fissa a 32 bit

OPCODE						*MODE*																							
0	**0**	**0**	**0**	**1**	**0**	***0***	***0***	***0***	***0***	***0***	***0***	***0***	***0***	***0***	***1***	***0***	***0***	***0***	***0***	***1***	***1***	***0***	***0***	***1***	***0***	***1***	***0***	***0***	***0***	***0***	***0***

ISTRUZIONI

Linguaggio assemblativo

- Il programmatore ricorre ad una rappresentazione simbolica delle istruzioni, utilizzando codici mnemonici che possono essere interpretati in maniera più comoda rispetto alle sequenze binarie: **istruzioni assembly**
- La **sintassi** di una istruzione assembly è costituita da:
  - **un indirizzo**, dove risiede l'istruzione in memoria (spesso omesso, perché impostato dall'assemblatore)
  - Una **etichetta** (opzionale)
  - Una **istruzione**:
    - un **codice mnemonico**, che descrive l'istruzione con pochi, ma significativi, caratteri
    - Il **modo di indirizzamento**, cioè i dati su cui operare o il luogo dove essi risiedono
  - i **commenti**,indispensabili per la comprensione del codice

100	CICLO:	ADD	$t0,$t1$t2	#somma i valori dei registri $t0=t1 + t2

ISTRUZIONI

Linguaggio assemblativo

❑ L'insieme delle istruzioni assembly definiscono un **linguaggio assemblativo** (*assembly* o *assembly language*)

Una istruzione aritmetica come la somma è, in assembly MIPS, così rappresentata:

```
add$t0,$t1,$t2
```

ADD	$t0,$t1$t2	#somma i valori dei registri $t0=t1 + t2

ISTRUZIONI

Linguaggio assemblativo

- Il legame che intercorre tra istruzione macchina e istruzione assembly è di **uno a uno**, nel senso che ad ogni istruzione macchina corrisponde una ed una sola istruzione assembly
- Per comodità molti linguaggi assembly utilizzano delle **pseudoistruzioni** ovvero delle istruzioni che sono composte da una o più istruzione assembly elementare

```
        LW $t0,x
        LW $t1,y
        BGT $t0,$t1, SALTA #$t0>$t1 salta
        LW $t2,z
Salta:
        ...

        LW $t0,x
        LW $t1,y
        SLT   $1,$9,$8 #set del registro AT a 1 se $t0>$t1
        BNE $1,$0,0x000002 #se AT!=0 salta una istruzione
        LW $t2,z
Salta:
        ...
```

ISTRUZIONI

Linguaggio assemblativo: macro

- Un linguaggio assembly consente di definire delle **macro**: una macro sostituisce una serie di istruzioni

- Ogni volta che si richiama la macro l'assemblatore riscrive le istruzioni definite nella macro

ESEMPIO DI MACRO IN MIPS

```
.macro end
    li $v0,10
    syscall
.end_macro

.text
.globl main

main:
        …
end
```

```
.macro end
    li $v0,10
    syscall
.end_macro

.text
.globl main

main:
        …

li $v0,10
    syscall
```

ISTRUZIONI

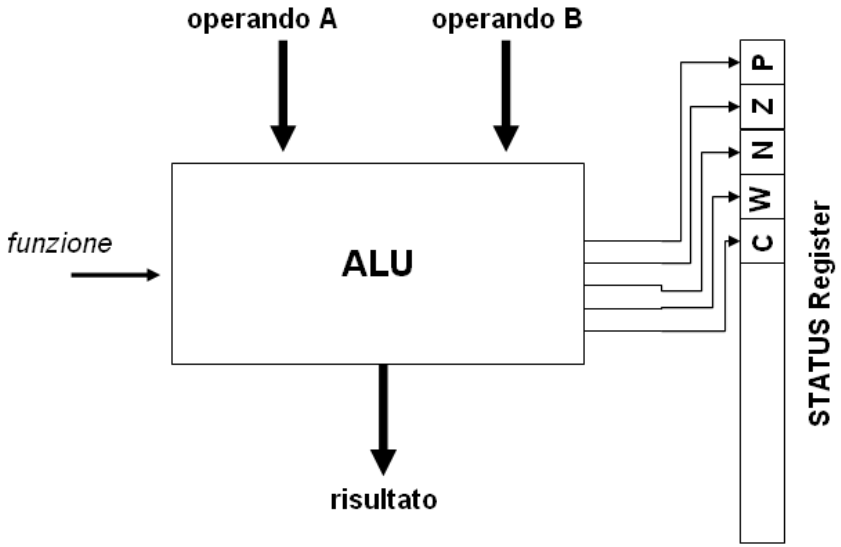
Linguaggio assemblativo: fase di pre-assemblaggio

- Prima della fase di assemblaggio a doppia passata si effettua un **pre-assemblaggio** dove accadono queste operazioni:
  - **si risolvono le macro**
  - **si risolvono le pseudo istruzioni**
  - **si includono eventuali file esterni**
  - **si analizzano le direttive**

ISTRUZIONI

Esecuzione istruzioni logiche aritmetiche

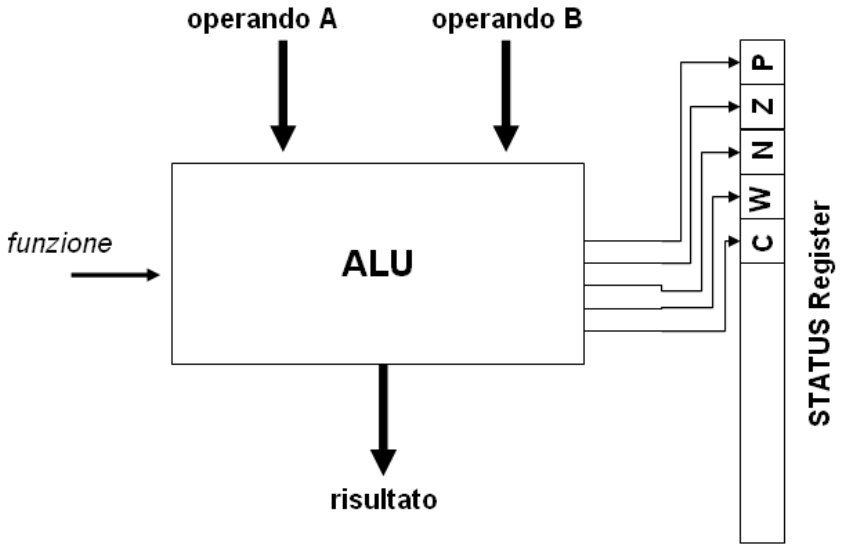
- Ad ogni tempo, dettato dal clock, l'elaboratore esegue una istruzione
- Ogni **istruzione logico-aritmetica**, produce dei bit, definiti **flags** (codici di condizione, o *condition code*), che saranno implicitamente memorizzati nel **registro di stato** (PSW, *processor status word*, o STATUS *register*)
- I Condition Codes svolgono un **ruolo fondamentale per le istruzioni di salto condizionato**



ISTRUZIONI

Codici di condizione

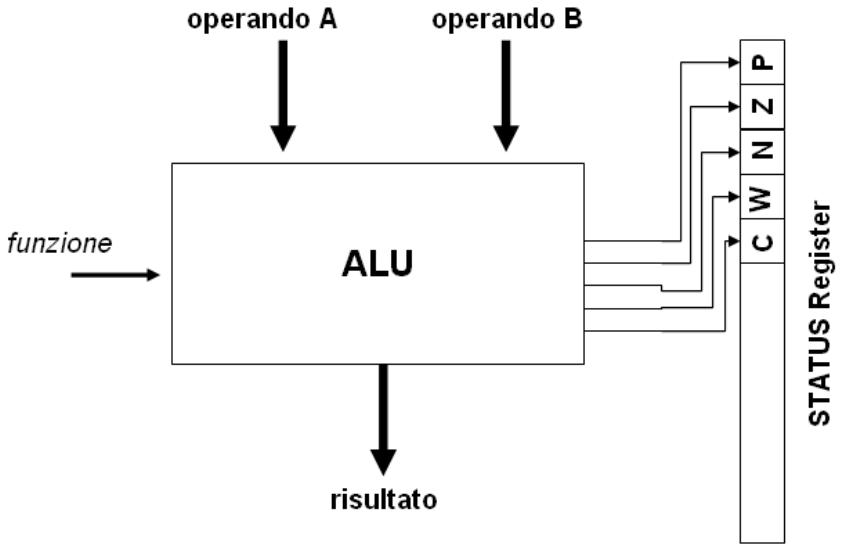
- I **principali flags** sono:
  - **C - Carry**: Individua il trabocco ed è impostato ad 1 se l'ultima operazione effettuata dall'ALU ha prodotto un riporto (addizione) o un prestito (sottrazione) a sinistra del bit più significato del risultato, 0 altrimenti
  - **N - Negative**: impostato ad 1 se l'ultima operazione effettuata dall'ALU ha prodotto un risultato negativo, 0 altrimenti. Ovvero Negative è una copia del bit più significativo del risultato
  - **Z - Zero**: impostato ad 1 se l'ultima operazione effettuata dall'ALU è nulla, 0 altrimenti.



ISTRUZIONI

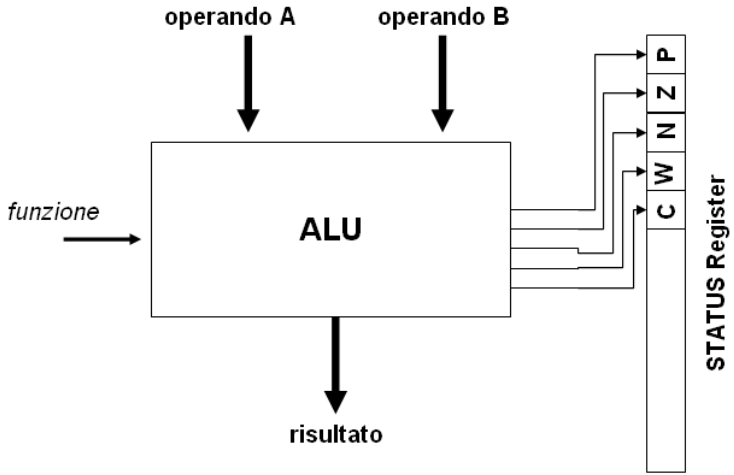
Codici di condizione

- I principali flags sono:
  - **W - Overflow**: impostato ad 1 se l'ultima operazione effettuata dall'ALU ha superato la capacità di rappresentazione data dalla lunghezza della parola, 0 altrimenti
  - **P - Parity**: impostato ad 1 se l'ultima operazione effettuata dall'ALU ha dato un risultato con un numero pari di 1; 0 altrimenti



ISTRUZIONI

Codici di condizione




```
li $t0, -1
li $t2, 1
add $t0, $t1, $t2
```

N = 0 (il risultato infatti è 0, che è considerato positivo)

Z = 1 (perchè si è verificato che il risultato è 0)

C = 1 (perchè si verifica un trabocco)

W = 0 (perchè non c'è overflow)

Classi di Istruzioni

ISTRUZIONI: CLASSI

- Classi di Istruzione
  - istruzione di spostamento dati
  - istruzioni logico ed aritmetiche
  - istruzioni di salto:
    - condizionato
    - non condizionato
    - a funzione (o a subroutine)
    - trap
  - istruzione di controllo macchina

ISTRUZIONI DI SPOSTAMENTO

Generalità

- Le **istruzioni per lo spostamento dei dati** servono a ricopiare un dato da una sorgente ad una destinazione e cioè da:
  - memoria a registro
  - registro a memoria
  - registro a registro
  - memoria a memoria

Codice Mnemonico	Sorgente	Destinazione

ISTRUZIONI DI SPOSTAMENTO

Esempi

- Le istruzioni di spostamento possono interessare la CPU e la Memoria (LOAD, STORE, PUSH e POP) o solamente i registri nella CPU (MOVE)
- Il contenuto della destinazione non si modifica rispetto alla sorgente

CODICE	OPERANDI	Commento
LOAD	<Sorgente><Destinazione>	Legge l'operando dalla sorgente (memoria) e lo copia nella destinazione (tipicamente un registro)
STORE	<Sorgente><Destinazione>	Legge l'operando dalla sorgente (tipicamente un registro) e lo copia nella destinazione (una cella di memoria esplicitata)
MOVE	<Sorgente><Destinazione>	Sposta il contenuto di un registro Sorgente ad un registro Destinazione
PUSH	<Sorgente>	Sposta un operando da una Sorgente(un registro o una cella in memoria) in cima allo stack/pila Equivale a STORE sorg,-($SP)
POP	<Destinazione>	Sposta un operando dalla cima dello stack/pila in una Destinazione (un registro o una cella in memoria) Equivale a LOAD ($SP)+,dest

ISTRUZIONI LOGICHE-ARITMETICHE

Generalità

- Le **istruzioni aritmetiche** consentono di effettuare le operazioni su numeri interi binari rappresentati in complemento a due (in alcuni casi le ALU possono svolgere operazioni anche con numeri in virgola mobile, ma spesso queste operazioni sono demandate ad una unità di calcolo – il coprocessore matematico - che è visto come un dispositivo di I/O)
- Le funzioni di base offerte dalla ALU sono il complemento, la comparazione e l'addizione; operazioni come la moltiplicazione o la divisione e la sottrazione possono essere ricavati sfruttando algoritmi che impiegano le operazioni elementari sopra citate

Codice Mnemonico	Destinazione	Sorgente 1	Sorgente 2

ISTRUZIONI LOGICHE-ARITMETICHE

Istruzioni aritmetiche

- Le istruzioni aritmetiche sono eseguite dall'ALU la quale produce due linee di uscita:
  - il risultato dell'operazione;
  - un vettore di bit, o *flags* (anche CC o *condition code*), che viene implicitamente caricato nello Status Register

CODICE	OPERANDI	Commento
ADD	<Destinazione><Sorgente><Sorgente>	Legge gli operandi dalla sorgente (memoria/registri), effettua la somma ed il risultato è trasferito nella destinazione (tipicamente un registro).
CMP	<Destinazione><Sorgente><Sorgente>	Legge gli operandi dalla sorgente (memoria/registri), effettua la comparazione ed il risultato è trasferito nella destinazione (tipicamente un registro)
NEG	<Destinazione><Sorgente>	Legge l'operando dalla sorgente (memoria/registro), effettua la negazione ed il risultato è trasferito nella destinazione (tipicamente un registro)

ISTRUZIONI LOGICHE-ARITMETICHE

Istruzioni logiche

- Le **operazioni logiche** permettono l'esecuzione delle più importanti operazioni definite nell'algebra booleana su stringhe binarie. Come per le operazioni aritmetiche, anche in questo caso, le operazioni avvengono per tutti i bit in posizione corrispondente
- La sintassi è simile alle istruzioni aritmetiche e l'operando sorgente può essere in una locazione di memoria, in un registro, o un dato costante (residente dopo l'istruzione); mentre l'operando destinazione è di solito un registro. Anche in questo caso i passi elementari che costituiscono la fase di decodifica ed esecuzione sono analoghi per tutte le istruzioni. Le istruzioni logiche permettono di modificare alcuni bit di un registro, di esaminare il loro valore o di settarli tutti a 0 o 1.

CODICE	OPERANDI	Commento
AND	Registro <Sorgente>, <Sorgente>	Legge gli operandi dalla sorgente (memoria/registri) ed effettua l'AND riportando il risultato in un registro
OR	Registro <Sorgente>, <Sorgente>	Legge gli operandi dalla sorgente (memoria/registri) ed effettua l'OR riportando il risultato in un registro
XOR	Registro <Sorgente>, <Sorgente>	Legge gli operandi dalla sorgente (memoria/registri) ed effettua l'XOR riportando il risultato in un registro
NOT	Registro <Sorgente>	Legge l'operando dalla sorgente (memoria/registri) ed effettua l'NOT riportando il risultato in un registro

ISTRUZIONI LOGICHE-ARITMETICHE

Istruzioni logico-aritmetiche

- Le istruzioni di **rotazione** e **shift** che operano su un solo dato posto in un registro. Queste istruzioni cambiano l'ordine dei bit nel registro ed hanno un significato:
  - **logico**: per effettuare lo scorrimento dei bit del registro nella direzione e nel numero di posizioni specificati. Il bit C (carry o trabocco) dello Status Register riceve l'ultimo bit che fuoriesce dal registro;
  - **aritmetico**: è opportuno ricordare che uno shift a destra equivale a dividere l'operando per 2^k (con k il numero di posizioni scorse), mentre uno scorrimento verso sinistra equivale a moltiplicare l'operando per 2^k (con k il numero di posizioni scorse)

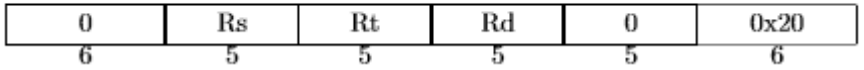
CODICE	OPERANDI	Commento
SL	Registro, k	Shifit a sinistra di k posti del registro
SR	Registro, k	Shifit a destra di k posti del registro
ROL	Registro, k	Ruota a sinistra di k posti del registro
ROR	Registro, k	Ruota a destra di k posti del registro

ISTRUZIONI LOGICHE-ARITMETICHE

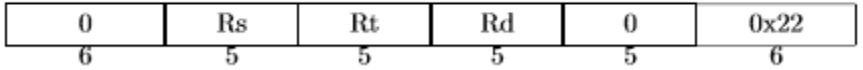
Istruzioni logico-aritmetiche MIPS

- Le istruzioni logico-aritmetiche e nel MIPS prevedono un OPECODE comune 000000 che individua la classe e poi una sottodivisione negli ultimi 6 bit che specifica il tipo di funzione

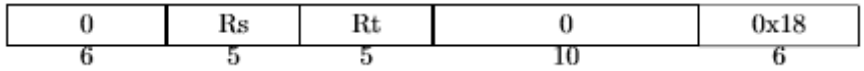
`add Rd, Rs, Rt` — *Addition (with overflow)*



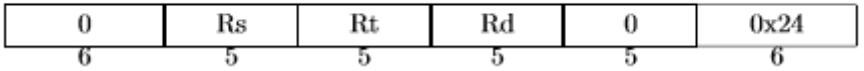
`sub Rd, Rs, Rt` — *Subtract (with overflow)*



`mult Rs, Rt` — *Multiply*



`and Rd, Rs, Rt` — *AND*



ISTRUZIONI LOGICHE-ARITMETICHE

Istruzioni IMPLICITE

- Esistono inoltre istruzioni, con referenziamento implicito, che consento di operare sui singoli bit del Registro di Stato

CODICE	Commeto	CODICE	Commeto
CLRC	Imposta a 0 il flag C	**SETC**	Imposta a 1 il flag C
CLRN	Imposta a 0 il flag N	**SETN**	Imposta a 1 il flag N
CLRZ	Imposta a 0 il flag Z	**SETZ**	Imposta a 1 il flag Z
CLRW	Imposta a 0 il flag W	**SETW**	Imposta a 1 il flag W

ISTRUZIONI LOGICHE-ARITMETICHE

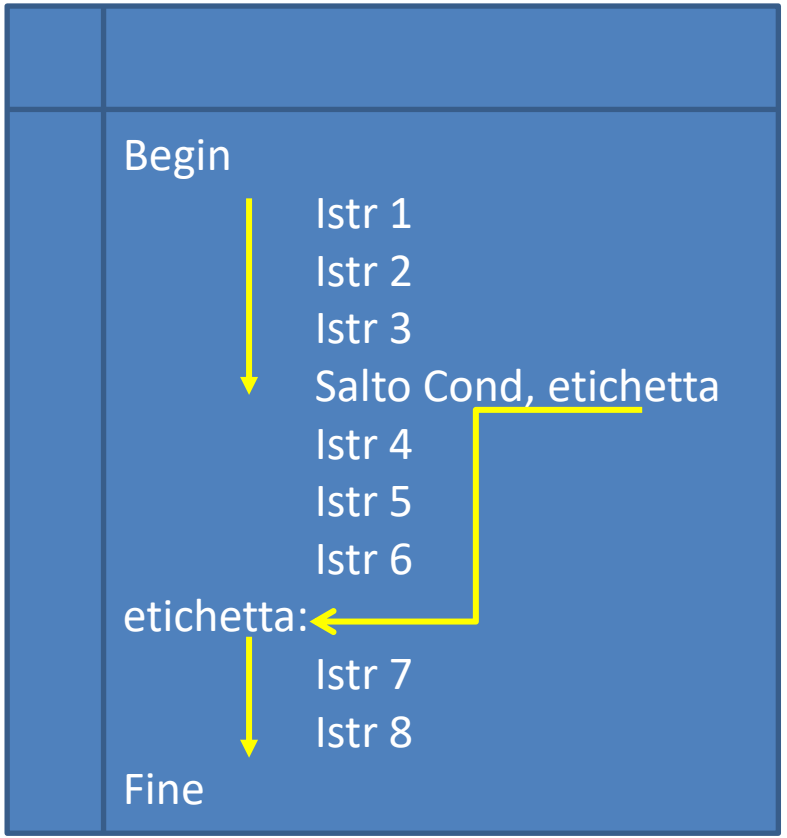
Modifiche dei Condition Code

Codice	C	N	Z	W	P
ADD	1/0	1/0	1/0	1/0	1/0
CMP	1/0	1/0	1/0	1/0	1/0
NEG	1/0	1/0	1/0	1/0	1/0
SUB	1/0	1/0	1/0	1/0	1/0
AND	0	1/0	1/0	0	1/0
OR	0	1/0	1/0	0	1/0
XOR	0	1/0	1/0	0	1/0
NOT	0	1/0	1/0	0	1/0
SL	1/0	1/0	1/0	1/0	1/0
SR	1/0	1/0	1/0	1/0	1/0
ROL	1/0	0	0	0	1/0
ROR	1/0	0	0	0	1/0
CLRC	0	-	-	-	-
CLRN	-	0	-	-	-
CLRZ	-	-	0	-	-
CLRW	-	-	-	0	-
CLRP	-	-	-	-	0

ISTRUZIONI DI SALTO

Generalità

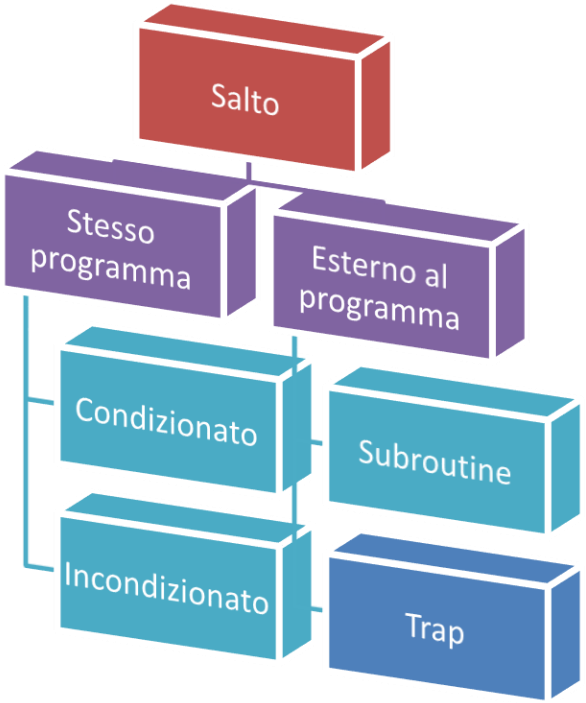
- Le **istruzioni di salto** individuano una classe particolare in quanto non agiscono direttamente sui dati, ma sono utilizzate per modificare l'ordine sequenziale di esecuzione delle istruzioni del programma stesso o uno esterno



ISTRUZIONI DI SALTO

Classificazione

- Le istruzioni di salto si dividono in:
  - salto all'interno dello stesso programma
    - **condizionato**: il salto viene eseguito in base ad una certa condizione fissata dal programmatore (Branch)
    - **incondizionato**: il salto viene sempre eseguito (Jump)
  - salto ad un altro programma: **salto a subroutine** (salto a sottoprogramma)
  - ***trap*** (o interruzioni software)



ISTRUZIONI DI SALTO

Condizionato e incondizionato

ESEMPIO SALTO CONDIZIONATO

La decodifica ed esecuzione di una istruzione di *brach*,

BEQZ $t0,0x100,

può essere così descritta

Decodifica:

Unità di Controllo ← BEQZ

Esecuzione:

Set z=1 se $t0 è zero (confronto con registro $zero)

Se Z=1 ⇒ PC← 0x100

Se Z=0 ⇒ *non fa nulla*

ESEMPIO SALTO INCONDIZIONATO

La decodifica ed esecuzione di una istruzione di *jump*,

J 0x100

può essere così descritta

Decodifica:

Unità di Controllo ← J

Esecuzione:

PC← 0x100

ISTRUZIONI DI SALTO

Condizionato e incondizionato

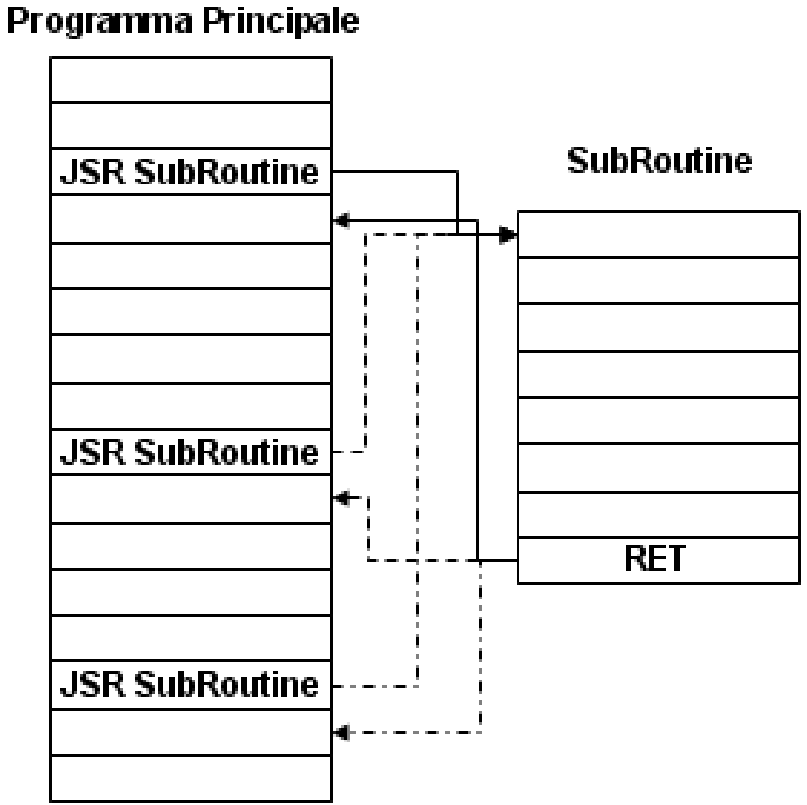
- Le istruzioni di salto sono fondamentali perché rompono la sequenzialità offrendo la possibilità di **effettuare scelte**, cioè prendere decisioni e perché consentono di eseguire più volte una parte di programma (es.: costrutto IF; ciclo while)

CODICE	OPERANDI	Commento
BEQZ	<Sorg1>, Indirizzo	Se l'operando contenuto in una sorgente (registro/memoria) è uguale a zero salta all'indirizzo specificato
BGT	<Sorg1>,<Sorg2>,Indirizzo	Legge gli operandi dalla sorgente (memoria/registri) e salta all'indirizzo se Sorg1 è maggiore della Sorg2
BLT	<Sorg1>,<Sorg2>,Indirizzo	Legge gli operandi dalla sorgente (memoria/registri) e salta all'indirizzo se Sorg1 è minore della Sorg2
J	Indirizzo	Salto incondizionato all'indirizzo specificato

ISTRUZIONI DI SALTO

Salto a subroutine

- L'**istruzione di salto a subroutine** (o chiamata a funzione) permette di saltare da un programma (il programma principale) ad un sottoprogramma, di eseguirlo e di tornare alla istruzione immediatamente successiva a quella di chiamata
- L'utilizzo di subroutine è utile quando un determinato insieme di istruzioni deve essere eseguito più volte e per avere un codice più chiaro e compatto. Inoltre le subroutine possono essere realizzate da terzi, essere scambiate e modificate ai propri fini



ISTRUZIONI DI SALTO

Salto a subroutine

CODICE	OPERANDI	Commento
JSR	Indirizzo	Salva il valore del PC incrementato nello Stack e salta all'indirizzo specificato che individua l'inizio del sottoprogramma
RET		Ritorna al programma principale ripristinando il valore del PC recuperato nello stack

ISTRUZIONI DI SALTO

Salto a subroutine

SALTO A SUBROUTINE

La decodifica ed esecuzione di una istruzione di *salto a funzione*,

JSR subroutine

può essere così descritta (se la sub routine è a posizione 0x100) :

Decodifica:

Unità di Controllo ← JSR 0x100

Esecuzione:

(SP) ←PC+1 # istruzione successiva

SP ←SP+1 #spostamento stack

PC ←0x100 #salto

NB: equivalente ad una PUSH

RITORNO DA SUBROUTINE

La decodifica ed esecuzione di una istruzione di ritorno da subroutine

RET

può essere così descritta

Decodifica:

Unità di Controllo ← RET

Esecuzione:

SP← SP-1 #decremento stack

PC← (SP) #estrazione del PC conservato #nello stack

NB: equivalente ad una POP

ISTRUZIONI DI SALTO

Salto a subroutine MIPS

SALTO A SUBROUTINE MIPS

In MIPS un salto a subroutine è ottenuto salvando il valore del PC in un registro speciale **$ra**

Così

JAL subroutine

può essere così descritta (se la sub routine è a posizione 0x100) :

Decodifica:

Unità di Controllo ← JAL 0x100

Esecuzione:

$RA ←PC+1 # istruzione successiva

PC ←0x100 #salto

RITORNO DA SUBROUTINE

La decodifica ed esecuzione di una istruzione di ritorno da subroutine

JR $ra

può essere così descritta

Decodifica:

Unità di Controllo ← JR $ra

Esecuzione:

PC← ($ra) #aggiornamento PC con #indirizzo di ritorno

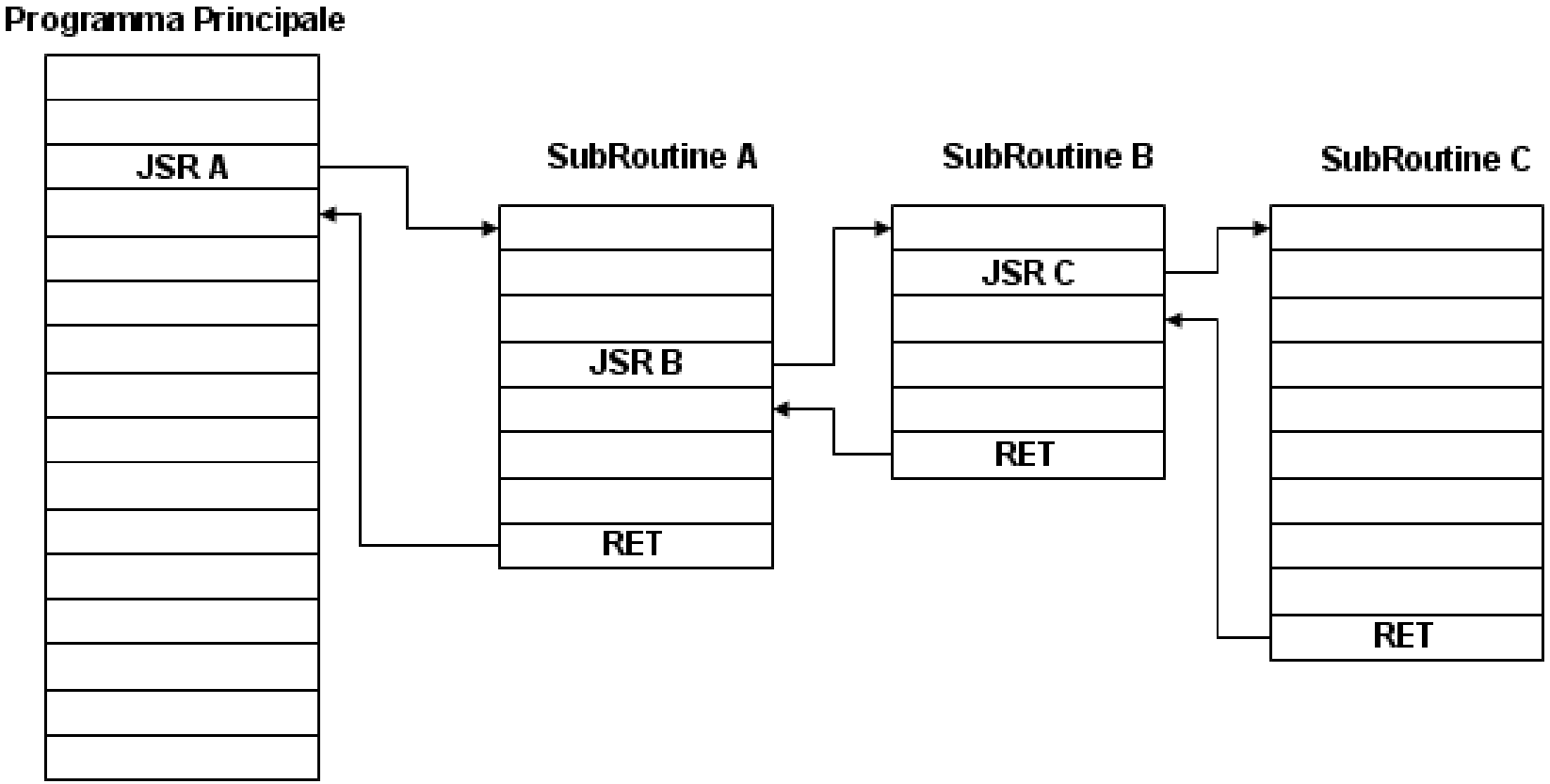
ISTRUZIONI DI SALTO

Annidamento di subroutine

- Molto spesso però i sottoprogrammi possono a loro volta chiamare altri programmi e così via. Può avverarsi cioè un **annidamento di subroutine** (*nested subroutine*)
- In questo caso è fondamentale salvare i diversi indirizzi di ritorno

ISTRUZIONI DI SALTO

Annidamento di subroutine



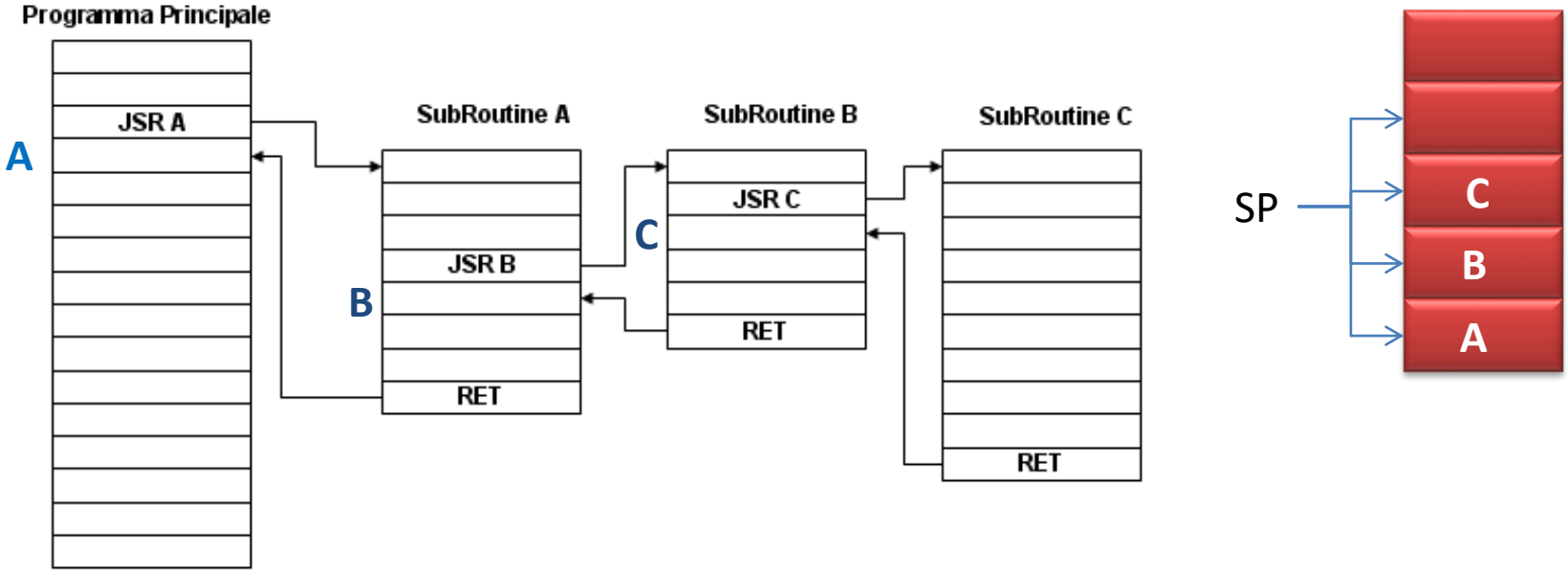
ISTRUZIONI DI SALTO

Annidamento di subroutine

- La gestione di funzioni ricorsive o l'annidamento di funzioni avviene grazie all'utilizzo della **pila** (**stack** o *canasta*)
- Lo stack è una zona di memoria riservata per il passaggio di parametri e la memorizzazione di informazioni gestita nella **modalità LIFO** (*Last in First Out*): ovvero l'ultimo elemento immesso nella pila è anche il primo ad uscire

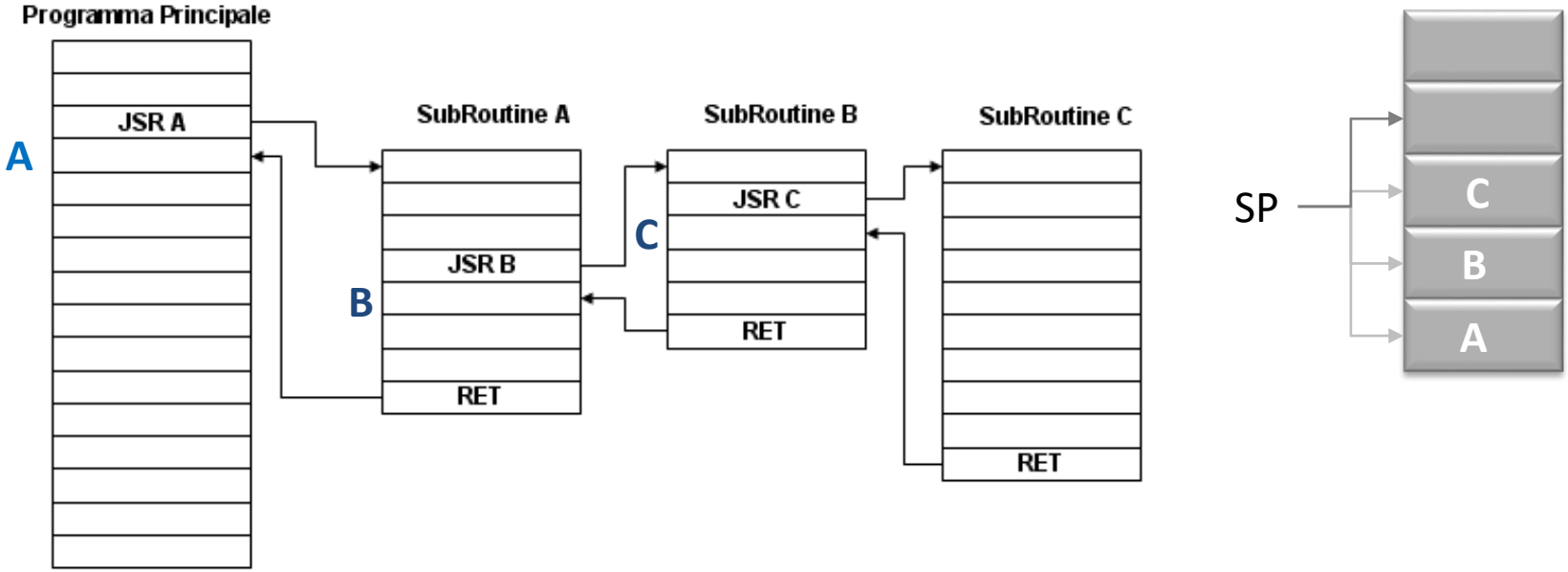
ISTRUZIONI DI SALTO

Annidamento di subroutine



ISTRUZIONI DI SALTO

Annidamento di subroutine



ISTRUZIONI I/O

Generalità

- Per interagire con i dispostivi di I/O si può ricorre ad un set di istruzioni dedicato (**IO canonico**) o riservare un'area di memoria agli scambi con i dispositivi di I/O ed operare con le istruzioni della macchina (**IO programmato**)

ISTRUZIONI I/O

4 INTEL

- SSE4 è un set d'istruzioni proposto della Intel ed utilizzato dai processori Core 2 Duo, Conroe e Merom del 2007
- Prevede:
  1. “Accelerazione video”: 14 istruzioni per i calcoli utilizzati nell'elaborazione di contenuti video
  2. "Graphics Building Blocks“: 32 istruzioni primitive orientate alla grafica
  3. "Streaming Load“: utile per accedere e ricevere dati di ritorno da dispositivi di memoria che non sono presenti nella cache del sistema

- Si opera in parallelo e si lavora con istruzioni in floating point come "Floating Point Dot Product" (DPPS, DPPD) e "Floating Point Round" (ROUNDPS, ROUNDSS, ROUNDPD, ROUNDSD), coinvolti nell'ottimizzazione delle scene 2D e 3D (videogiochi).

ISTRUZIONI I/O

SSE4:esempio

- Alcuni esempi:

PMINUD xmm1,xmm2 — **compara i valori contenuti in xmm1 e xmm2 e mette il MINIMO in xmm1**

PINSRD xmm1,r/m32,imm8 — **inserisce a partire dalla posizione specifica in imm8 il contenuto del registro o della cella di memoria a 32 bit nella destinazione di 64bit in xmm1**

MPSADBW xmm1,xmm2/m128,imm8

somma la differenza assoluta di 4 byte selezionati a partire da m128 e con posizione espressa in imm8 con 4byte in xmm1

ISTRUZIONI CONTROLLO MACCHINA

- Le **istruzioni di comando** (o istruzioni di controllo macchina) non operano né sui dati né sui registri né interessano il contatore di programma, ma intervengono direttamente sullo stato della CPU
- Le istruzioni di comando sono caratteristiche di ogni CPU: il loro numero può variare da poche unità, per macchine semplici, a decine per macchine complesse

CODICE	Commento
HALT	Interruzione di sistema
NOP	Nessuna operazione. ***È utile per il Delay Slot del pipeling***
BREAK	Interruzione di programma

Un esempio pratico

ESEMPIO

Progettazione di un elaboratore elettronico

Realizzazione di un elaboratore che svolga la sola funzione di l'elevamento a potenza di un numero (con esponente>0) e che utilizza istruzioni a dimensione fissa

es.: $2^3=8$ $3^3=27$ $5^2=25$

ESEMPIO

Programma da realizzare

- **Realizzazione di un elaboratore che svolga la sola funzione di l'elevamento a potenza di un numero (con esponente>0)**

  e che utilizza istruzioni a dimensione fissa

  es.: $2^3=8$ $3^3=27$ $5^2=25$

Possibile implementazione

```
        LOAD $R0, BASE
        LOAD $R1,ESPONENTE
        LOAD $R2, UNO
        LOAD $R3,MENOUNO
CICLO:
        BEQZ $R1,FINE
        MUL $R2,$R2,$R0
        ADD $R1,$R1,$R3
        JUMP  CICLO
FINE:
        STORE $R2, RISULTATO
```

ESEMPIO

Elenco elementi da considerare

Possibile implementazione

```
        LOAD $R0, BASE
        LOAD $R1,ESPONENTE
        LOAD $R2, UNO
        LOAD $R3,MENOUNO
CICLO:
        BEQZ $R1,FINE
        MUL $R2,$R2,$R0
        ADD $R1,$R1,$R3
        JUMP  CICLO
FINE:
        STORE $R2, RISULTATO
```

Di cosa si ha bisogno:

- ❑ 4 registri enumerati da R0 a R3
- ❑Una ALU che faccia 4 operazioni: moltiplicazione, somma, salto condizionato al valore zero, salto incondizionato
- ❑Istruzioni di caricamento e archiviazione dati in memoria
- ❑Spazio in memoria per archiviare i dati e gli operandi

3bit per individuare istruzioni

2bit per individuare un registro

ESEMPIO

Definizione linguaggio assemblativo

- Numero di istruzioni: **6**
- Registri: **4** (+ 1 di ausilio alla macchina trasparente al programmatore settato a 0 e non modificabile)

LOAD <registro di destinazione>, <indirizzo dove risiede l'operando>
STORE <registro sorgente>, <indirizzo dove copiare l'operando>
BEQZ<registro con valore di confronto>, <indirizzo dove saltare>
MUL<registro destinazione>, <registro con moltiplicatore 1><registro con moltiplicatore 2>
ADD<registro destinazione>, <registro con operando 1><registro con operando 2>
JUMP <indirizzo dove saltare>

Quanto deve essere lunga la parola per indirizzare almeno 255 locazioni di memoria (126 per ospitare il programma e 127 per ospitare i dati)?

ESEMPIO

Linguaggio macchina

LOAD <registro di destinazione>, <indirizzo dove risiede l'operando>

STORE <registro sorgente>, <indirizzo dove copiare l'operando>

ADD<registro destinazione>, <registro con operando 1><registro con operando 2>

MUL<registro destinazione>, <registro con moltiplicatore 1><registro con moltiplicatore 2>

ESEMPIO

Linguaggio macchina

BEQZ<registro con valore di confronto>, <indirizzo dove saltare>

JUMP <indirizzo dove saltare>

Programma in memoria da locazioni: 0-127
Dati in memoria da locazioni: 128-255

ESEMPIO

Codice Eseguibile

00000000		0	0	0	0	0	1	0	0	0	0	0	0	0	LOAD $R0, BASE
00000001		0	0	0	0	1	1	0	0	0	0	0	1	0	LOAD $R1,ESPONENTE
00000010		0	0	0	1	0	1	0	0	0	0	1	0	0	LOAD $R2,UNO
00000011		0	0	0	1	1	1	0	0	0	0	1	1	0	LOAD $R2,MENOUNO
00000100		1	0	0	0	1	0	0	0	0	1	0	0	0	BEQZ $R1,FINE
00000101		0	1	1	1	0	1	0	0	0					MUL $R2,$R2,$R0
00000110		0	1	0	0	1	0	1	1	1					ADD $R1,$R1,$R3
00000111		1	0	1	0	0	0	0	0	1	0	0			JUMP CICLO
00001000		0	0	1	1	0	1	0	0	0	1	0	0	0	SW $R2, RISULTATO

Fine