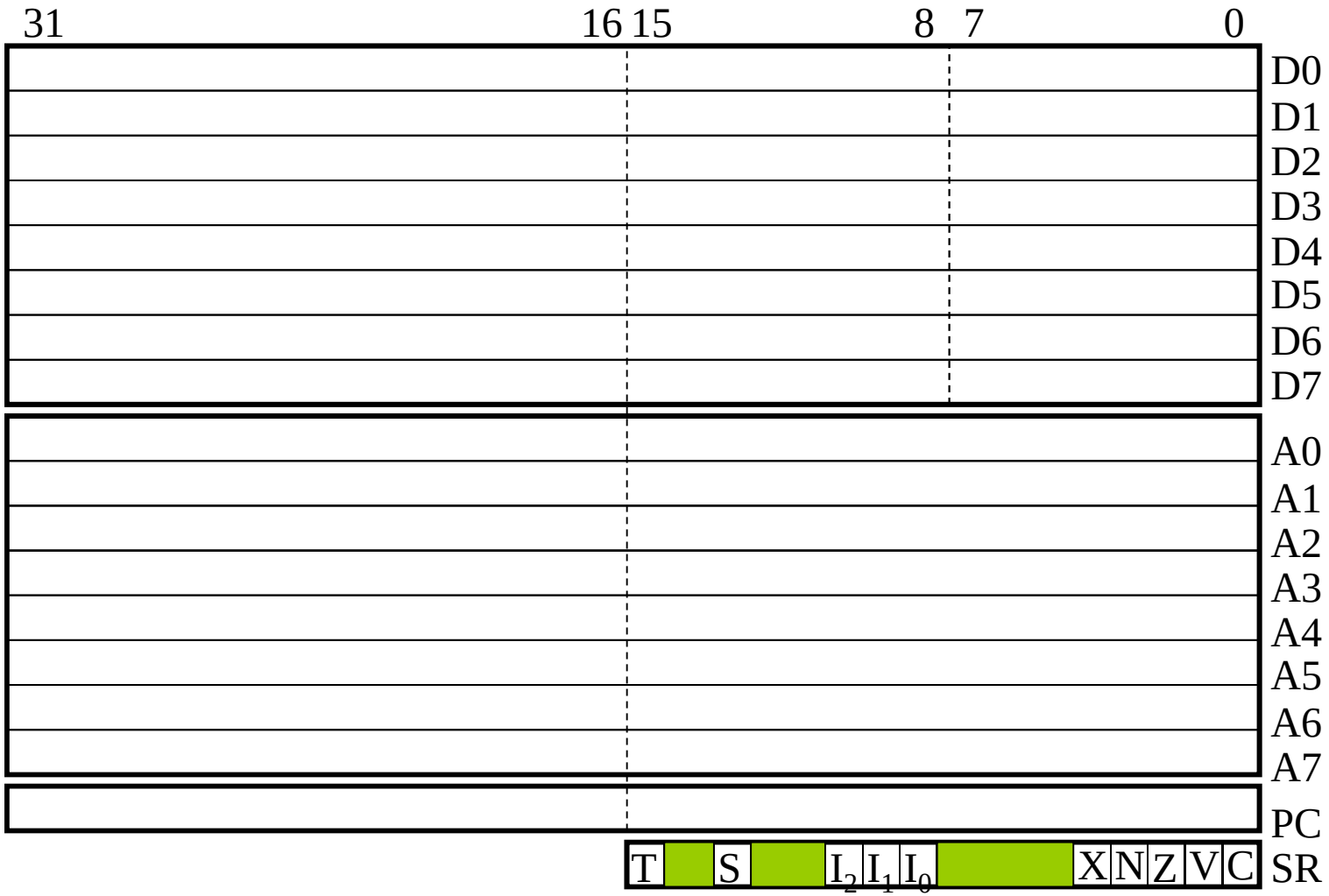
Corso di Calcolatori Elettronici I
A.A. 2012-2013

Introduzione al linguaggio assembly del processore Motorola 68000

ing. Alessandro Cilardo

Accademia Aeronautica di Pozzuoli
Corso Pegaso V “GArn Elettronici”

68000: Modello di programmazione

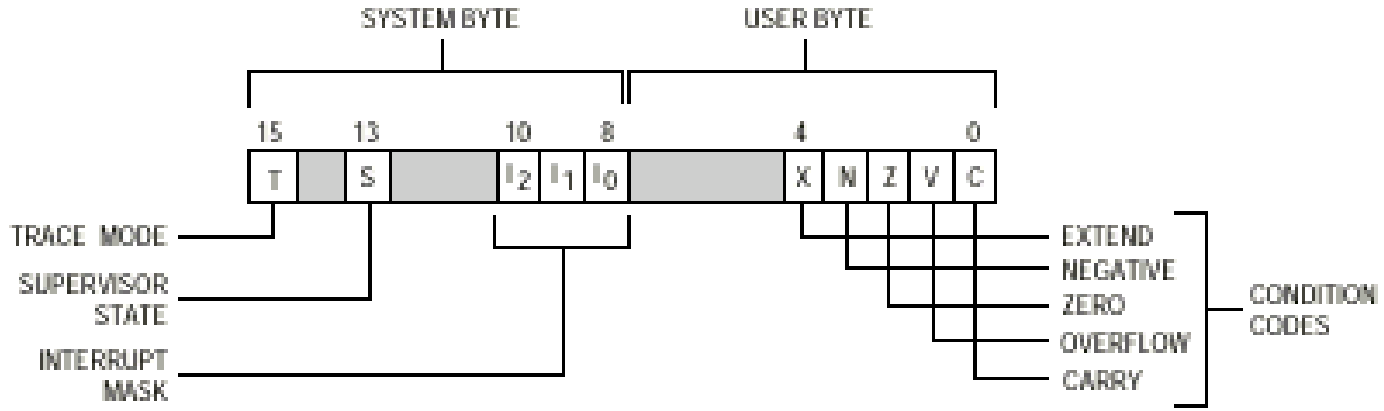


68000: Modello di programmazione

- 8 registri dato $\mathbf{D_i}$ utilizzabili per dati generici
  - possono essere usati tutti i 32 bit dei registri, o solo i 16 meno significativi (lasciando inalterati gli altri), o solo gli 8 bit meno significativi (lasciando inalterati gli altri)
- 8 registri indirizzo $\mathbf{A_i}$ utilizzati per contenere valori di indirizzi in memoria
  - possono essere utilizzati a 32 o a 16 bit
- Program counter
  - contenente l'indirizzo della prossima istruzione da eseguire
- Registro di stato
  - contenente informazioni sulle modalità di funzionamento e sullo stato del processore

68000: Status Register

- Contiene:
  - La interrupt mask (8 livelli)
  - I codici di condizione (CC) - oVerflow (V), Zero (Z), Negative (N), Carry (C), e eXtend (X)
  - Altri bit di stato - Trace (T), Supervisor (S)
  - I Bits 5, 6, 7, 11, 12, e 14 non sono definiti e sono riservati per espansioni future



68000: esempio di programma assembly

PLC	contenuto		label	opcode	operands	comments
00000000		1	* Programma per sommare i primi 17 interi			
00000000		2	*			
00008000		3		ORG	$8000	
00008000	4279 00008032	4	START	CLR.W	SUM	
00008006	3039 00008034	5		MOVE.W	ICNT,D0	
0000800C	33C0 00008030	6	ALOOP	MOVE.W	D0,CNT	
00008012	D079 00008032	7		ADD.W	SUM,D0	
00008018	33C0 00008032	8		MOVE.W	D0,SUM	
0000801E	3039 00008030	9		MOVE.W	CNT,D0	
00008024	0640 FFFF	10		ADD.W	#-1,D0	
00008028	66E2	11		BNE	ALOOP	
0000802A	4EF9 00008008	12		JMP	SYSA	
00008030	=00008008	13	SYSA	EQU	$8008	
00008030		14	CNT	DS.W	1	
00008032		15	SUM	DS.W	1	
00008034	=00000011	16	IVAL	EQU	17	
00008034	0011	17	ICNT	DC.W	IVAL	

Symbol Table

ALOOP	800C	CNT	8030	IVAL	0011
START	8000	SUM	8032	ICNT	8034

Formato dei programmi assembly

- Una linea di codice sorgente Assembly 68000 è costituita da quattro campi:
  - **LABEL** (*opzionale*)
    - Stringa alfanumerica
    - Definisce un nome simbolico per il corrispondente indirizzo
  - carattere **TAB** + **OPCODE**
    - Codice mnemonico o pseudo-operatore
    - Determina la generazione di un'istruzione in linguaggio macchina o la modifica del valore corrente del *Program Location Counter,* o PLC (l'indirizzo fisico che verrà assegnato alla riga corrente)
  - carattere **TAB** + **OPERANDI**
    - Oggetti dell'azione specificata dall'OPCODE
    - Variano a seconda dell'OPCODE e del modo di indirizzamento
  - carattere **TAB** + **COMMENTI** (*opzionale*)
    - Testo arbitrario inserito dal programmatore

Formato dei programmi assembly

- Ogni riga deve cominciare con una etichetta o con uno spazio/TAB
- Gli spazi bianchi tra i diversi campi fungono esclusivamente da separatori (vengono ignorati dall'assemblatore)
- Una linea che inizi con un asterisco ("*****") è di commento
- Nelle espressioni assembly, gli argomenti di tipo numerico si intendono espressi
  - in notazione decimale, se non diversamente specificato
  - in notazione esadecimale, se preceduti dal simbolo "**$**"
- Nell'indicazione degli operandi, il simbolo "**#**" denota un valore immediato inserito direttamente nell'istruzione

Lunghezza degli operandi

- La maggior parte delle istruzioni può operare su operandi di lunghezza configurabile:
  - 8 bit (Byte)
  - 16 bit (Word)
  - 32 bit (Long word)
- La lunghezza degli operandi è controllata tramite un suffisso aggiunto alle istruzioni: **`.B`**, **`.W`**, o **`.L`**
  - se il suffisso è omesso, si intende implicitamente **`.W`**
- A seconda del suffisso, vengono influenzate parti diverse dei registri, in particolare i registri $\mathbf{D}_i$:
  - gli 8 bit meno significativi (B) del registro
  - i 16 bit meno significativi (W) del registro
  - tutti i 32 bit (L) del registro

Pseudo-operatori

- NON sono istruzioni eseguite dal processore
  - sono direttive che regolano il processo di traduzione del programma assembler in programma eseguibile

- Lo pseudo-operatore **ORG**
  - viene usato per inizializzare il Program Location Counter (PLC), ovvero per **indicare a quale indirizzo** sarà posta la successiva sezione di codice o dati
  - **Esempio:** `ORG $8100`
- Lo pseudo-operatore **END**
  - viene usato per terminare il processo di assemblaggio ed impostare l'***entry-point*** (prima istruzione da eseguire) nel programma
  - **Esempio:** `END TARGETLAB`

Pseudo-operatori

- Lo pseudo-operatore **DS** (*Define Storage*)
  - viene usato per incrementare il Program Location Counter (PLC), in modo da riservare spazio di memoria per una variabile
  - **Esempio:** `LABEL DS.W NUMSKIPS`

- Lo pseudo-operatore **DC** (*Define Constant*)
  - viene usato per inizializzare il valore di una variabile
  - **Esempio:** `LABEL DC.W VALUE`

- Lo pseudo-operatore **EQU**
  - viene usato per definire una costante testuale usata nel sorgente assembler in sostituzione di un valore
  - **Esempio:** `LABEL EQU VALUE`

Etichette (label)

- Sono stringhe di testo arbitrarie (opzionali) anteposte ad una istruzione o ad un dato all'interno del programma assembler
- Servono a riferirsi al particolare indirizzo che contiene quella istruzione o dato
  - usate per gestire i salti
  - usate per gestire variabili (manipolate nel programma assembler attraverso le loro etichette in maniera simile alle variabili di un linguaggio di programmazione di alto livello)
- Ad esempio:
  - ALOOP è un'etichetta usata per riferirsi all'istruzione MOVE, SUM è una etichetta usata per gestire una variabile, mentre IVAL è una costante testuale

```
ALOOP    MOVE.W      D0,CNT
         ADD.W       SUM,D0
   …   …  …   …
SUM      DS.W        1
IVAL     EQU         17
   … … … … … …
```

68000: istruzioni base

- **CLR**: pone il valore 0 nella destinazione
- **MOVE**: sposta un dato da sorgente a destinazione
- **ADD**: somma il valore della sorgente alla destinazione
- **ADDQ**: somma un valore immediato alla destinazione
- **SUB**: sottrae il valore della sorgente dalla destinazione
- **SUBQ**: sottrae un immediato dalla destinazione
- **CMP**: confronta i due operandi
- **JMP**: salta ad un'istruzione diversa dalla prossima
- **B*cc***: salta ad un'istruzione diversa dalla prossima se è vera la condizione specificata dai flag ***cc***

Far riferimento al manuale del processore Motorola 68000 per una descrizione dettagliata della sintassi e del funzionamento delle istruzioni

Programma di esempio - 1

File: **programma001.a68**

- Inizializzare due registri del processore con valori immediati e sommare i due valori

le istruzioni vengono poste in memoria a partire dall'indirizzo decimale **$8000**

```
        ORG     $8000
pippo   MOVE.L  #9,D0
        MOVE.L  #8,D1
        ADD.L   D0,D1
        END     pippo
```

*L'indirizzo della prima istruzione viene qui simbolicamente indicato con l'etichetta testuale "***pippo***"*

Sposta i due valori immediati (#), ovvero indicati direttamente dall'istruzione, nei registri **D0** *e* **D1**

Somma il contenuto dei due registri e poni il risultato in **D1**

*La prima istruzione ad essere eseguita sarà quella posta all'indirizzo in memoria indicato con "***pippo***"*

Programma di esempio - 1

File: **programma001.a68**

- Osserviamo il corrispondente file .LIS generato dalla compilazione del file precedente:

indirizzo in memoria, o PLC *(il primo indirizzo è stato determinato dalla* **ORG***)*

righe del file sorgente originario

Notare che i valori "immediati" **#9** *e* **#8** *sono stati inseriti direttamente nella codifica delle istruzioni*

```
00008000             1              ORG   $8000
00008000             2
00008000  7009       3     pippo    MOVE.L  #9,D0
00008002  7208       4              MOVE.L  #8,D1
00008004  D280       5              ADD.L   D0,D1
00008006             6              END   pippo
```

Contenuto effettivo della memoria: è la codifica delle istruzioni. Notare che le pseudo-istruzioni **ORG** *e* **END** ***non*** *corrispondono ad alcuna istruzione fisicamente presente in memoria: esse sono infatti solo delle direttive al compilatore*

*L'etichetta "***pippo***" ha assunto qui il valore dell'indirizzo* **$8000***, quello dellla riga a cui l'etichetta era associata*

Programma di esempio - 2

File: `programma002.a68`

- Leggere un valore di 4 byte dalla memoria. Far sì che il bit Z dello Status Register diventi **1** se il numero è pari, **0** altrimenti

```
        ORG     $8000
MAIN    MOVE.L  $8100,D4
        AND.L   #1,D4
        ORG     $8100
        DC.L    17
        END     MAIN
```

le istruzioni vengono poste in memoria a partire dall'indirizzo decimale **$8000**

La **MOVE.L** *prende qui come primo operando un valore numerico (senza #). Esso viene in questo caso interpretato come un indirizzo: "sposta il contenuto della locazione all'indirizzo* **$8100** *nel registro* **D4***"*

La **AND.L** *con l'immediato* **#1** *maschera (pone a zero) tutti i bit di* **D4** *tranne quello meno significativo. Se il contenuto di* **D4** *era pari, questo bit è zero, quindi per l'***ALU** *transiterà il valore zero ed il bit* ***Z****ero dello* **Status Register** *diventerà* **1**

La sezione seguente, contenente dati, viene posta in memoria a partire dall'indirizzo decimale **$8100**

*Riserva in memoria una locazione di 4 byte (***.L***) e inserisce al suo interno il valore* **17** *(decimale)*

Programma di esempio - 2

File: **programma002.a68**

- Osserviamo il corrispondente file .LIS generato dalla compilazione del file precedente:

```
00008000                    1         ORG   $8000
00008000                    2
00008000 2839 00008100      3   MAIN  MOVE.L $8100,D4
00008006 C8BC 00000001      4         AND.L  #1,D4
0000800C                    5

00008100                    6         ORG    $8100
00008100 00000011           7         DC.L   17
00008104                    8
00008104                    9         END    MAIN
```

Notare che, per entrambe le istruzioni, la codifca contiene **6 byte**, *quattro dei quali usati per codificare l'operando numerico (***$8100** *per la* **MOVE** *e* **#1** *per la* **AND**)

La parte del programma che segue la seconda direttiva **ORG** *è posta in memoria a partire dall'indirizzo* **$8100**. *Essa contiene un'unica locazione di 4 byte contenente il valore* **17** *(***11** *esadecimale) con ordine big-endian*

Programma di esempio - 3

File: `programma003.a68`

- Confrontare i contenuti di due locazioni di memoria di 4 byte. Far sì che il bit Z dello Status Register diventi **1** se i due valore sono uguali, **0** altrimenti

```
        ORG     $8000
MAIN    MOVE.L  A,D0
        CMP.L   B,D0
        ORG     $8100
A       DC.L    18
B       DC.L    18
        END     MAIN
```

*La **MOVE.L** prende qui come primo operando un'etichetta (**A**) che è rappresentativa di un indirizzo: "sposta il contenuto della locazione di memoria il cui indirizzo è denotato con **A** nel registro **D0**"*

*La **CMP.L** sottrae il contenuto della locazione di indirizzo **B** al contenuto del registro **D0** (senza modificarlo). Se i due contenuti sono uguali, per l'**ALU** transiterà il valore zero ed il bit **Z**ero dello **Status Register** diventerà **1***

*La sezione seguente, contenente dati, viene posta in memoria a partire dall'indirizzo decimale **$8100***

*Riserva in memoria due locazioni di 4 byte (**.L**) e inserisce in entrambe il valore **18** (decimale). Gli indirizzi delle due locazioni sono denotati simbolicamente tramite le etichette **A** e **B***

Programma di esempio - 3

File: **programma003.a68**

- Osserviamo il corrispondente file .LIS generato dalla compilazione del file precedente:

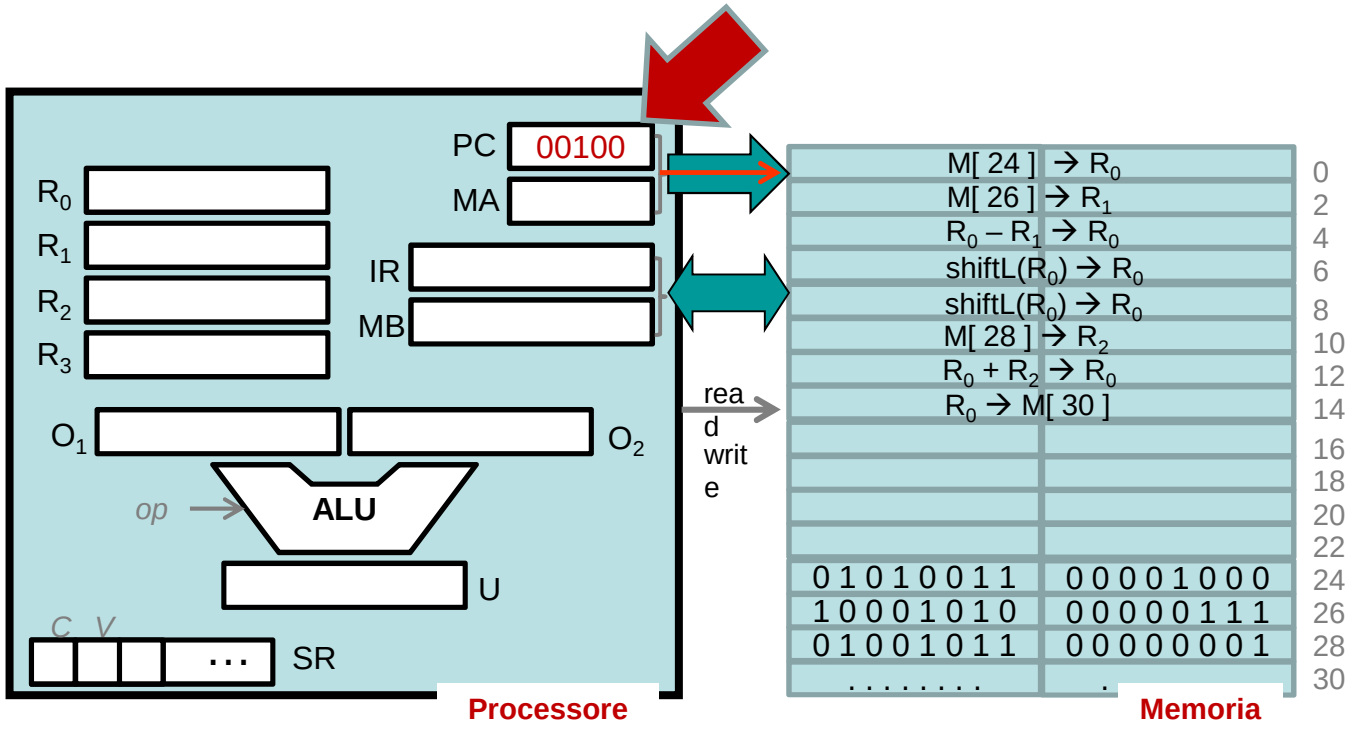
```
00008000                        1          ORG     $8000
00008000                        2
00008000 2039 00008100          3   MAIN   MOVE.L  A,D0
00008006 B0B9 00008104          4          CMP.L   B,D0
0000800C                        5
00008100                        6          ORG     $8100
00008100 00000012               7   A      DC.L 18
00008104 00000012               8   B      DC.L 18
00008108                        9
00008108                        10         END    MAIN
```

Notare che, per entrambe le istruzioni, la codifca contiene **6 byte**, *quattro dei quali usati per codificare l'operando numerico. Si tratta in entrambi i casi dell'indirizzo in memoria del primo operando dell'istruzione (***A** *per la* **MOVE** *e* **B** *per la* **CMP**)

La parte del programma che segue la seconda direttiva **ORG** *è posta in memoria a partire dall'indirizzo* **$8100**. *Essa contiene due locazioni di 4 byte contenenti entrambe il valore* **18** *(***12** *esadecimale) con ordine big-endian*

Istruzioni di salto

- Un'istruzione di **salto** inserisce un valore nel registro PC, "*forzando*" il processore ad eseguire un'istruzione diversa dalla successiva in memoria

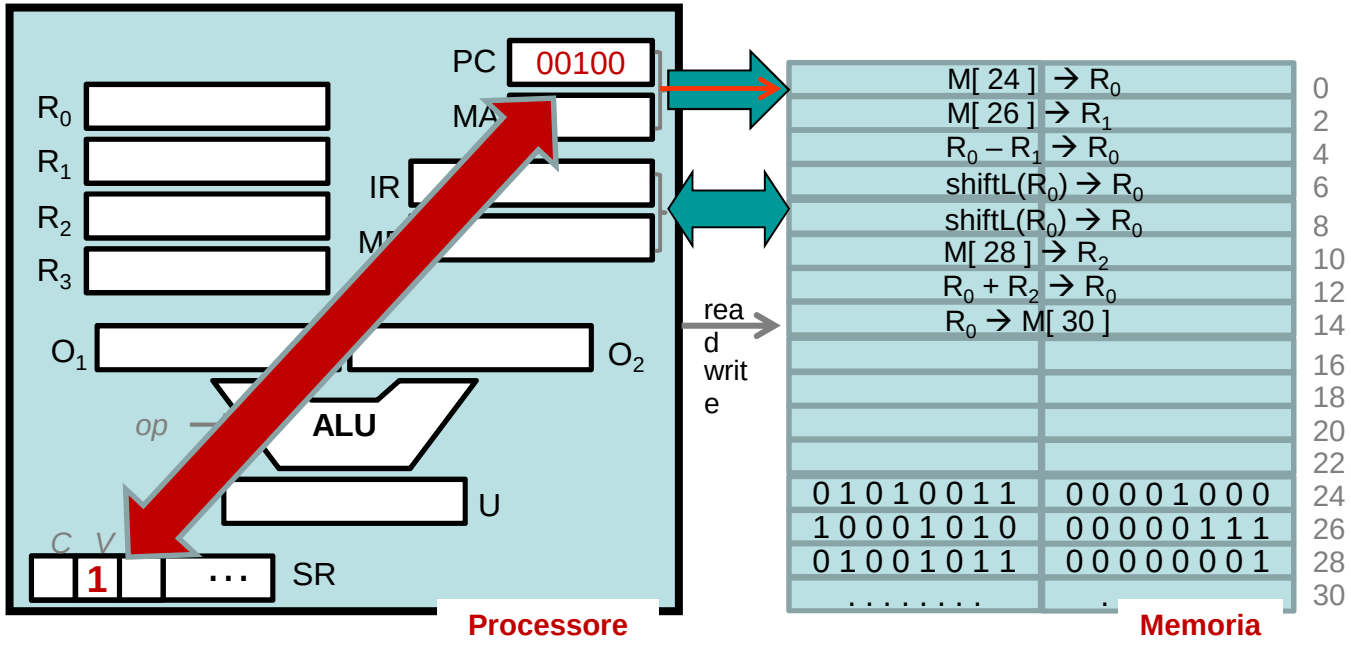


Istruzioni di salto

- Le istruzioni di salto consentono di passare da una sequenza di istruzioni all'altra in un programma, anche quando le diverse sequenze *non sono consecutive* in memoria
- Le istruzioni di salto possono essere "incondizionate" o "condizionate"
- In quest'ultimo caso, il salto ha effetto solamente se si verifica una specifica condizione all'interno del registro di stato:
  - ad esempio: condizione di o**V**erflow, condizione di **Z**ero, condizione di valore **N**egativo, etc.

Istruzioni di salto

- Un'istruzione di **salto condizionato** inserisce un valore nel registro PC, "*forzando*" il processore ad eseguire un'istruzione diversa dalla successiva in memoria, **solo se** si verifica una specifica condizione nel registro di stato



68000: Istruzioni di salto

- L'istruzione di salto incondizionato nel processore 68000 è la **JMP** (*Jump*)
- L'istruzione di salto condizionato nel processore 68000 è la **B*cc*** (*Branch on cc*)
  - dove ***cc*** denota una particolare combinazione di condizioni nel registro di stato che corrispondono a diversi esiti possibili di operazioni logico/aritmetiche (vedi lucido successivo)

Istruzione Branch (B*cc*)

Single bit	**Condizione nel Registro di Stato che innesca il salto**
• BCS branch on carry set	C = 1
• BCC branch on carry clear	C = 0
• BVS branch on overflow set	V = 1
• BVC branch on overflow clear	V = 0
• BEQ branch on equal (zero)	Z = 1
• BNE branch on not equal	Z = 0
• BMI branch on minus (i.e., negative)	N = 1
• BPL branch on plus (i.e., positive)	N = 0
Signed	
• BLT branch on less than (zero)	$N \oplus V = 1$
• BGE branch on greater than or equal	$N \oplus V = 0$
• BLE branch on less than or equal	$(N \oplus V) + Z = 1$
• BGT branch on greater than	$(N \oplus V) + Z = 0$
Unsigned	
• BLS branch on lower than or same	C + Z = 1
• BHI branch on higher than	C + Z = 0

Istruzione Branch (B*cc*): esempi

- **BEQ** (Branch on EQual): salta solo se il bit Z del Registro di Stato è 1
  - si usa spesso dopo un'istruzione di confronto (CMP):

    solo se questa si è trovata a confrontare due valori uguali, il bit Z sarà 1 ed il salto verrà effettuato
- **BNE** (Branch on Not Equal): ha il comportamento opposto rispetto alla BEQ
- **BLT** (Branch on Less Than): salta se il confronto precedente ha mostrato che il primo operando era minore del secondo
- etc…

Programma di esempio - 4

File: **programma004.a68**

- Data una variabile in memoria **N**, sommare i primi numeri interi da **1** a **N** e scrivere il risultato in memoria

```
        ORG     $8000
MAIN    CLR     D0
        CLR     D1
LOOP    ADD     #1,D1
        ADD     D1,D0
        CMP     N,D1
        BNE     LOOP
        MOVE    D0,SUM
        ORG     $8100
SUM     DC.W    0
N       DC.W    10
        END     MAIN
```

*La **CLR** pone il valore zero nei registri **D0** e **D1** (inizialmente potrebbe essere presente qualsiasi valore)*

*Incrementiamo il valore di **D1** aggiungendogli **#1***

*In **D0** memorizziamo la somma degli interi da **1** a **N**, che di volta in volta saranno contenuti in **D1**. La prima volta che eseguiamo questa istruzione, **D1** vale **1**.*

*Confrontiamo il valore raggiunto da **D1** con il valore **N***

*Se **D1** non ha raggiunto il valore di **N** (**10** nell'esempio), la **BNE** salta indietro all'indirizzo **LOOP**. Tutte le istruzioni in grigio verranno ripetute nuovamente! Altrimenti va avanti.*

*Copia il risultato finale da **D0** alla locazione di memoria **SUM***

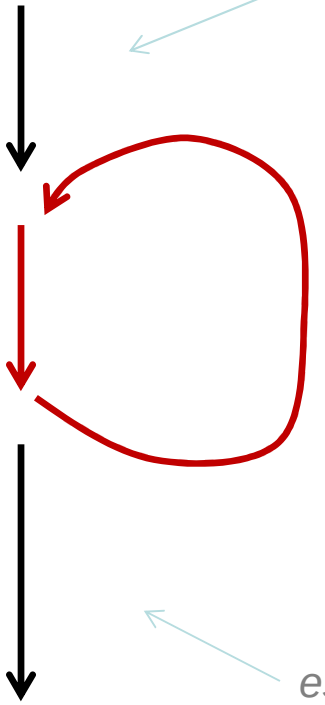
*Riserva in memoria due locazioni di 2 byte (**.W**). La prima (**SUM**) conterrà il risultato alla fine dell'esecuzione. La seconda contiene il valore **N** dato in ingresso al programma (**10** nell'esempio).*

Programma di esempio - 4

File: **programma004.a68**

- Data una variabile in memoria **N**, sommare i primi numeri interi da **1** a **N** e scrivere il risultato in memoria

```
        ORG     $8000
MAIN    CLR     D0
        CLR     D1
LOOP    ADD     #1,D1
        ADD     D1,D0
        CMP     N,D1
        BNE     LOOP
        MOVE    D0,SUM
        ORG     $8100
SUM     DC.W    0
N       DC.W    10
        END     MAIN
```



esecuzione sequenziale

***ciclo**: come effetto del salto causato dalla **BNE**, le istruzioni in grigio vengono ripetute fintantoché la **CMP** rileva che **D1** è diverso da **N**.*
*Inizialmente **D1** è pari a **1**, poi verrà incrementato ad ogni ripetizione. Appena **D1** diventa pari ad **N** (10 nell'esempio), la **CMP** segnala che i valori confrontati sono uguali ed il salto non viene più effettuato dalla **BNE**: si passa invece all'istruzione successiva in memoria*

esecuzione sequenziale

Programma di esempio - 5

File: **programma005.a68**

- Sommare gli elementi di un vettore di **N** elementi presente in memoria e scrivere il risultato in memoria

```
        ORG     $8000
START   MOVE.L  #VET,A0
        MOVE.L  #N,D0
        CLR     D2

CICLO   MOVE    (A0),D1
        ADD     #2,A0
        ADD     D1,D2
        SUBQ    #1,D0
        CMP     #0,D0
        BNE     CICLO
        MOVE    D2,RIS
N       EQU     $000A
        ORG     $80B0
VET     DC.W    1,3,1,5,-21,51,11,0,0,13
RIS     DS.W    1
        END     START
```

*Il registro **A0** contiene l'indirizzo in memoria da cui prelevare di volta in volta gli elementi del vettore (partendo dal primo)*

D0 *viene inizializzato con la dimensione del vettore* **N**

***Indirizzamento indiretto**: "Sposta il contenuto della locazione di memoria il cui indirizzo è dato dal registro **A0** nel registro **D1**"*

*Aggiorna **A0** in modo che contenga adesso l'indirizzo dell'elemento successivo nel vettore (ciascun elemento occupa 2 byte)*

*Decrementa **D0**. Se non ha raggiunto **0**, non abbiamo ancora finito: quindi salta indietro con la **BNE** e ripeti tutte le istruzioni in grigio. Altrimenti procedi con la prossima.*

N *(dimensione del vettore) è qui una costante testuale (**EQU**), non una variabile*

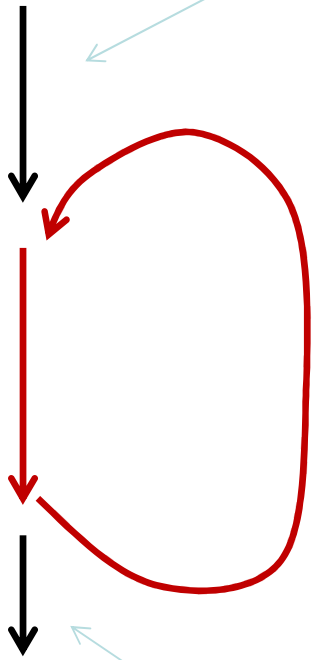
*Il vettore **VET** contiene 10 locazioni di due byte (**.W**), il cui contenuto è specificato tramite una **DC** sequita dai 10 valori*

Programma di esempio - 5

File: `programma005.a68`

- Sommare gli elementi di un vettore di **N** elementi presente in memoria e scrivere il risultato in memoria

```
        ORG     $8000
START   MOVE.L  #VET,A0
        MOVE.L  #N,D0
        CLR     D2
CICLO   MOVE    (A0),D1
        ADD     #2,A0
        ADD     D1,D2
        SUBQ    #1,D0
        CMP     #0,D0
        BNE     CICLO
        MOVE    D2,RIS
N       EQU     $000A
        ORG     $80B0
VET     DC.W    1,3,1,5,-21,51,11,0,0,13
RIS     DS.W    1
        END     START
```



ciclo**: come effetto del salto causato dalla **BNE**, le istruzioni in grigio vengono ripetute fintantoché la **CMP** rileva che **D0** è diverso da **0**. Ogni istruzione legge un elemento del vettore usando come indirizzo **A0**, lo copia in **D1** e lo somma nel registro **D2**. Inizialmente **D0** è pari a **N**, (10 nell'esempio), poi verrà decrementato ad ogni ripetizione. Appena **D0** diventa pari a **0**, la **CMP** segnala che i valori confrontati sono uguali ed il salto non viene più effettuato dalla **BNE**: si passa invece all'istruzione successiva in memoria, che copia il risultato da **D2** alla locazione **RIS