02: Esercitazione 01

Gestione della memoria
Vettori
Controllo di flusso di un programma

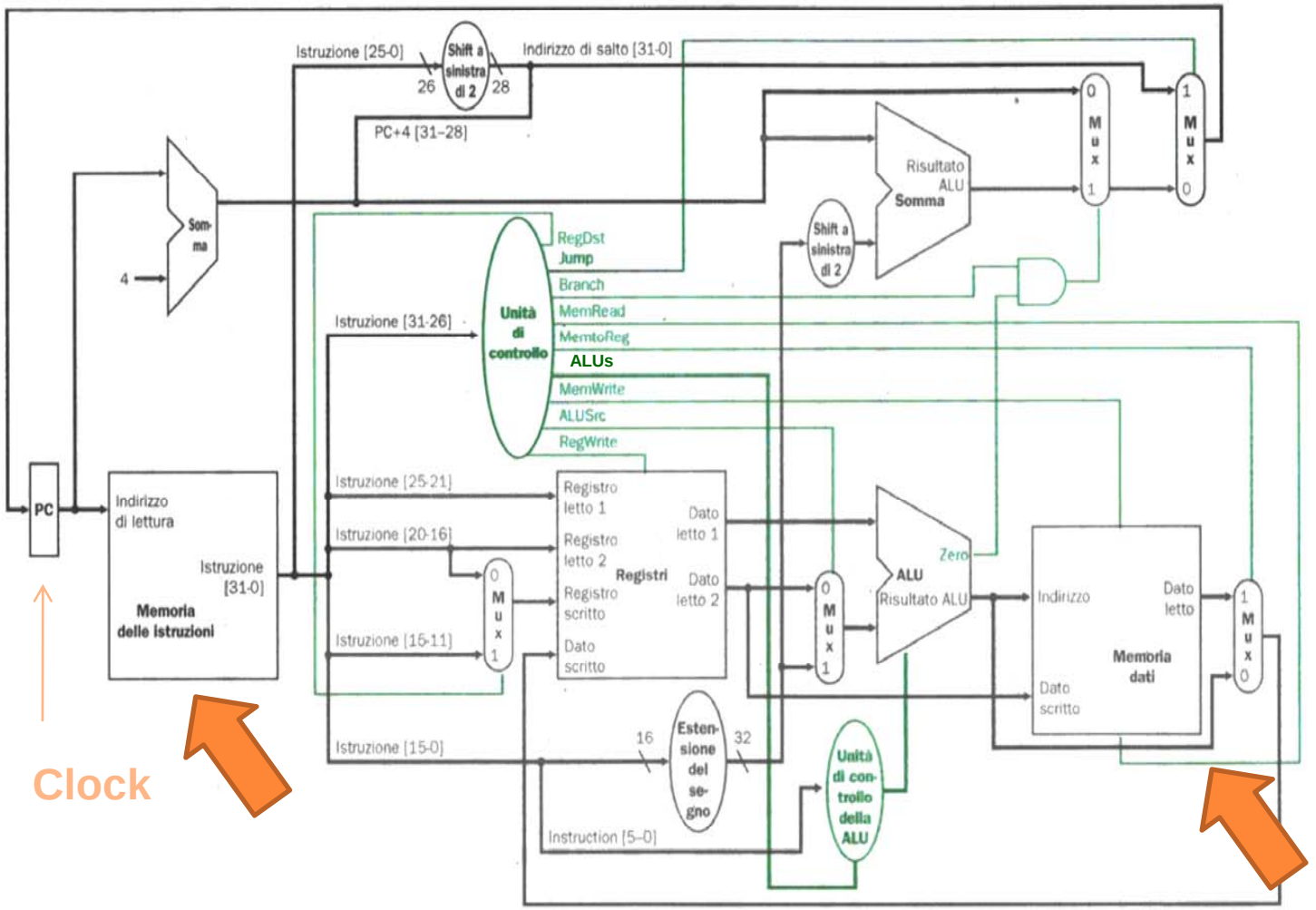
I. Frosio

21 March 2011

SOMMARIO

- Organizzazione della memoria 
- Istruzioni di accesso alla memoria
- Vettori
- Istruzioni MIPS di controllo di flusso (condizioni di uguaglianza)
- Istruzioni MIPS di controllo di flusso (condizioni di disuguaglianza).
- Costrutto switch

CPU + UC A SINGOLO CICLO

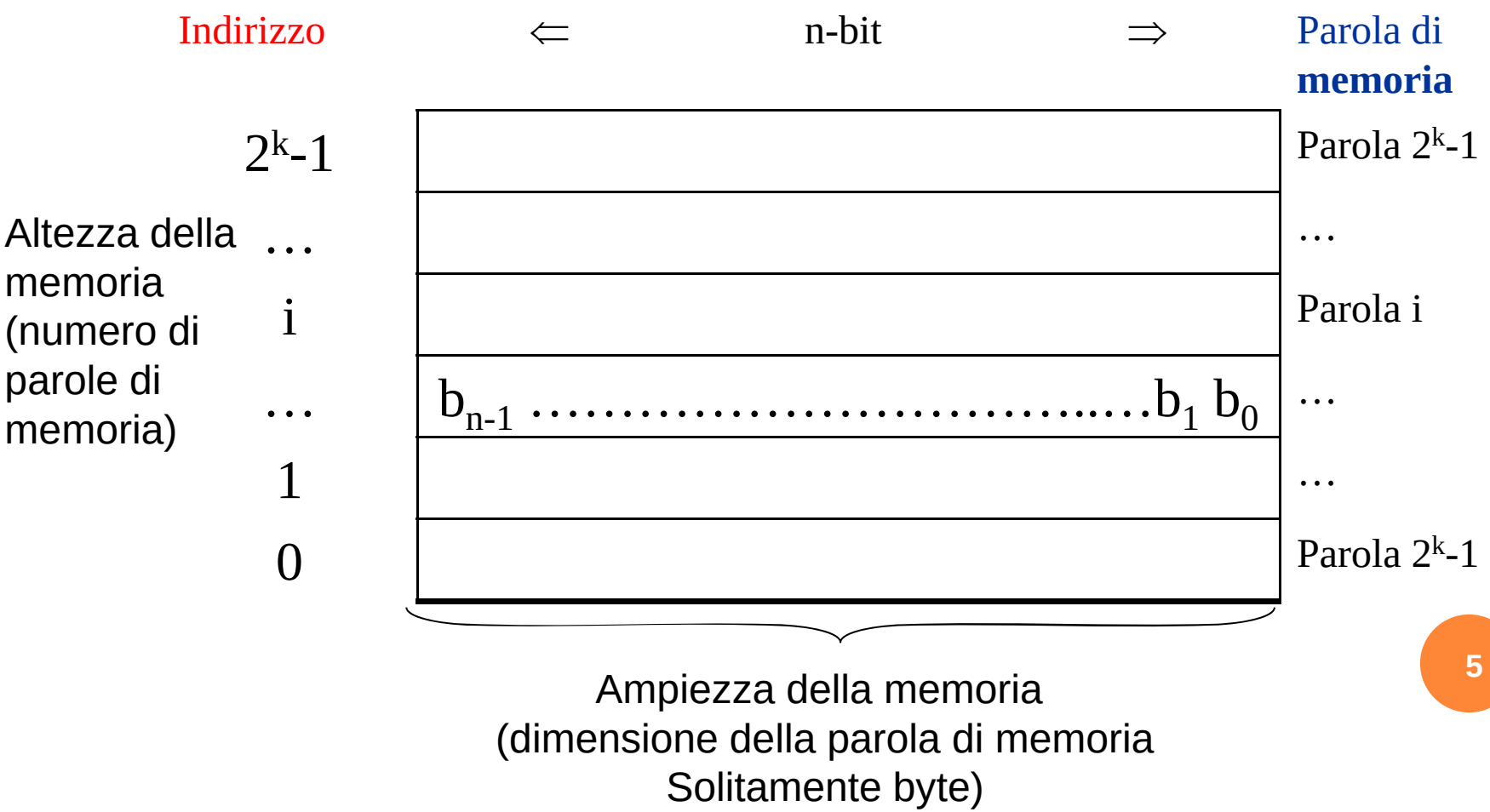


ORGANIZZAZIONE DELLA MEMORIA

- Le memorie in cui ogni locazione può essere raggiunta in un breve e prefissato intervallo di tempo misurato a partire dall'istante in cui si specifica l'indirizzo desiderato, vengono chiamate memorie ad accesso casuale *(Random Access Memory – RAM)*
- Nelle RAM il *tempo di accesso alla memoria (tempo necessario per accedere ad una* parola di memoria) è *fisso e indipendente dalla posizione della parola* alla quale *si* vuole accedere.
- Il contenuto delle locazioni di memoria può rappresentare **sia istruzioni che dati**, sui quali l'architettura sta lavorando.
- La memoria può essere visto come un array monodimensionale.

ORGANIZZAZIONE DELLA MEMORIA

- La memoria è vista come un unico grande array uni-dimensionale.
- Un **indirizzo di memoria costituisce un indice all'interno dell'array.**



ORGANIZZAZIONE DELLA MEMORIA

- La memoria è organizzata in *parole* composte da *n-bit* che possono essere indirizzati separatamente.
- Ogni parola di memoria è associata ad un indirizzo composto da *k-bit.*
- I 2^k indirizzi costituiscono lo *spazio di indirizzamento* del calcolatore. Ad esempio un indirizzo composto da *32-bit* genera uno spazio di indirizzamento di 2^{32} o *4Gbyte*.

MEMORIA PRINCIPALE E PAROLE

- In genere, la dimensione della parola di memoria non coincide con la dimensione dei registri contenuti nella *CPU*.

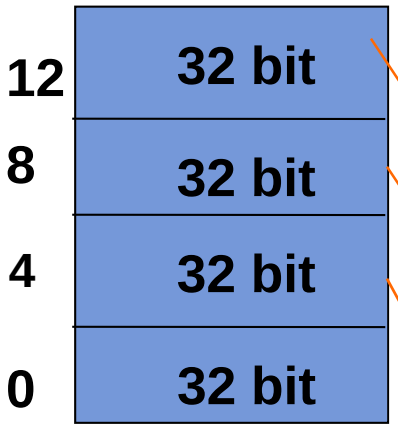
- Per ottimizzare i tempi, ad ogni trasferimento vengono trasferiti contemporaneamente un numero di byte pari o multiplo del numero di byte che costituisce la parola dell'architettura.

  $\Rightarrow$ l'operazione di *load/store* di una parola avviene in un singolo ciclo di clock del bus.

- Le parole hanno quindi generalmente indirizzo in memoria che è multiplo di 4. [Architettura a 32 bit del MIPS-32]

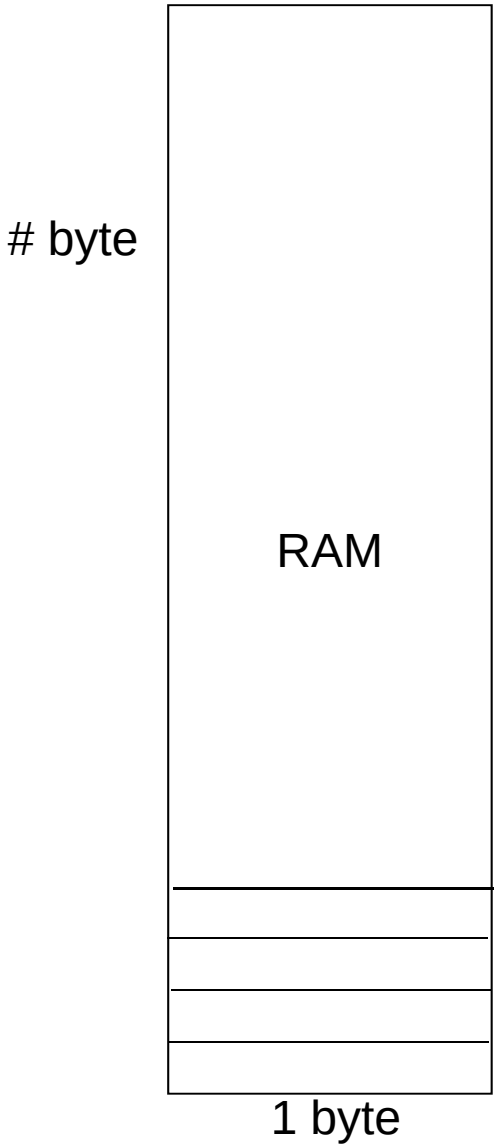
  $\Rightarrow$ Problema dell'allineamento dei dati.

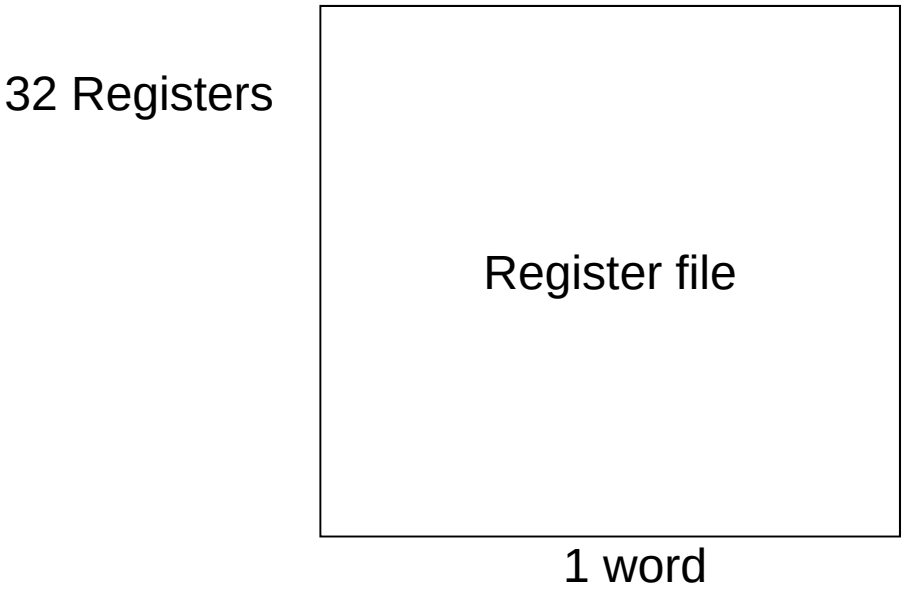
INDIRIZZAMENTO DELLA MEMORIA MIPS



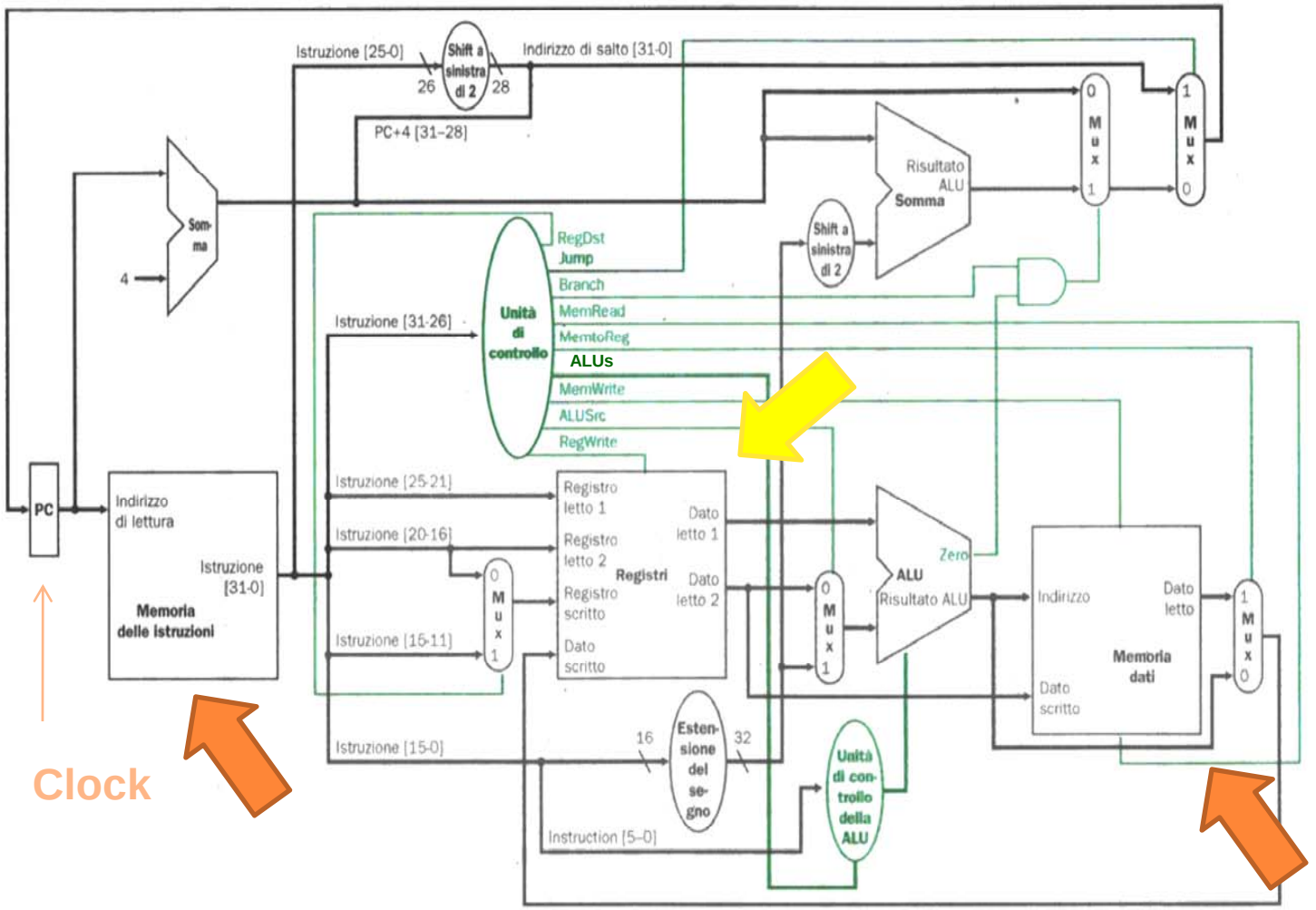
# byte			
2^k-4	2^k-3	2^k-2	2^k-1
8	9	10	11
4	5	6	7
0	1	2	3

MEMORIA E REGISTER FILE



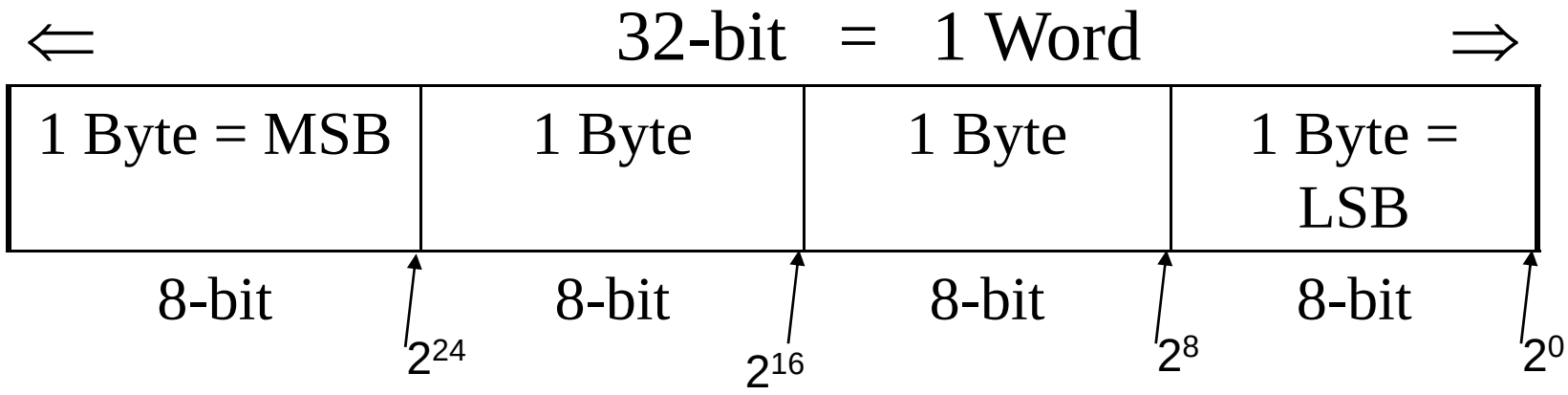


MEMORIA E REGISTER FILE



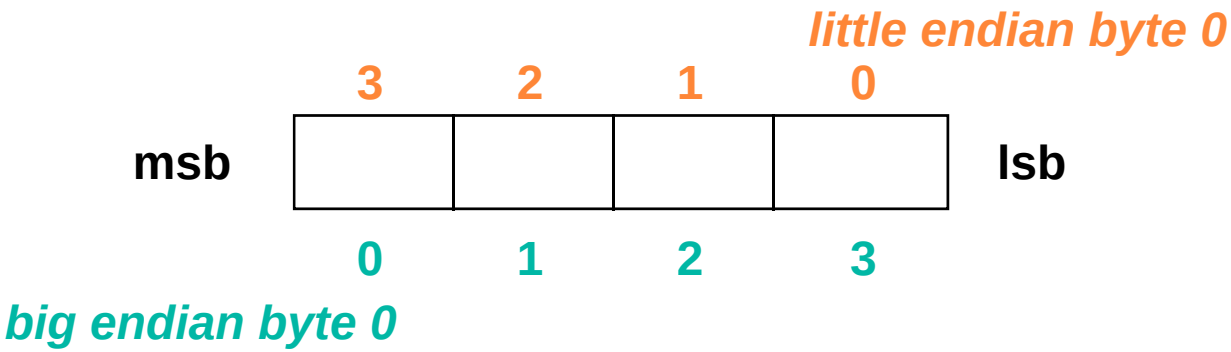
INDIRIZZAMENTO DEI BYTE ALL'INTERNO DELLA PAROLA

MIPS utilizza un **indirizzamento al byte**, cioè l'indice punta ad un byte di memoria.
Byte consecutivi hanno indirizzi consecutivi.
Indirizzi di parole consecutive (adiacenti) differiscono di un fattore 4 (8-bit x 4 = 32-bit): ad ogni indirizzo è associato un byte.

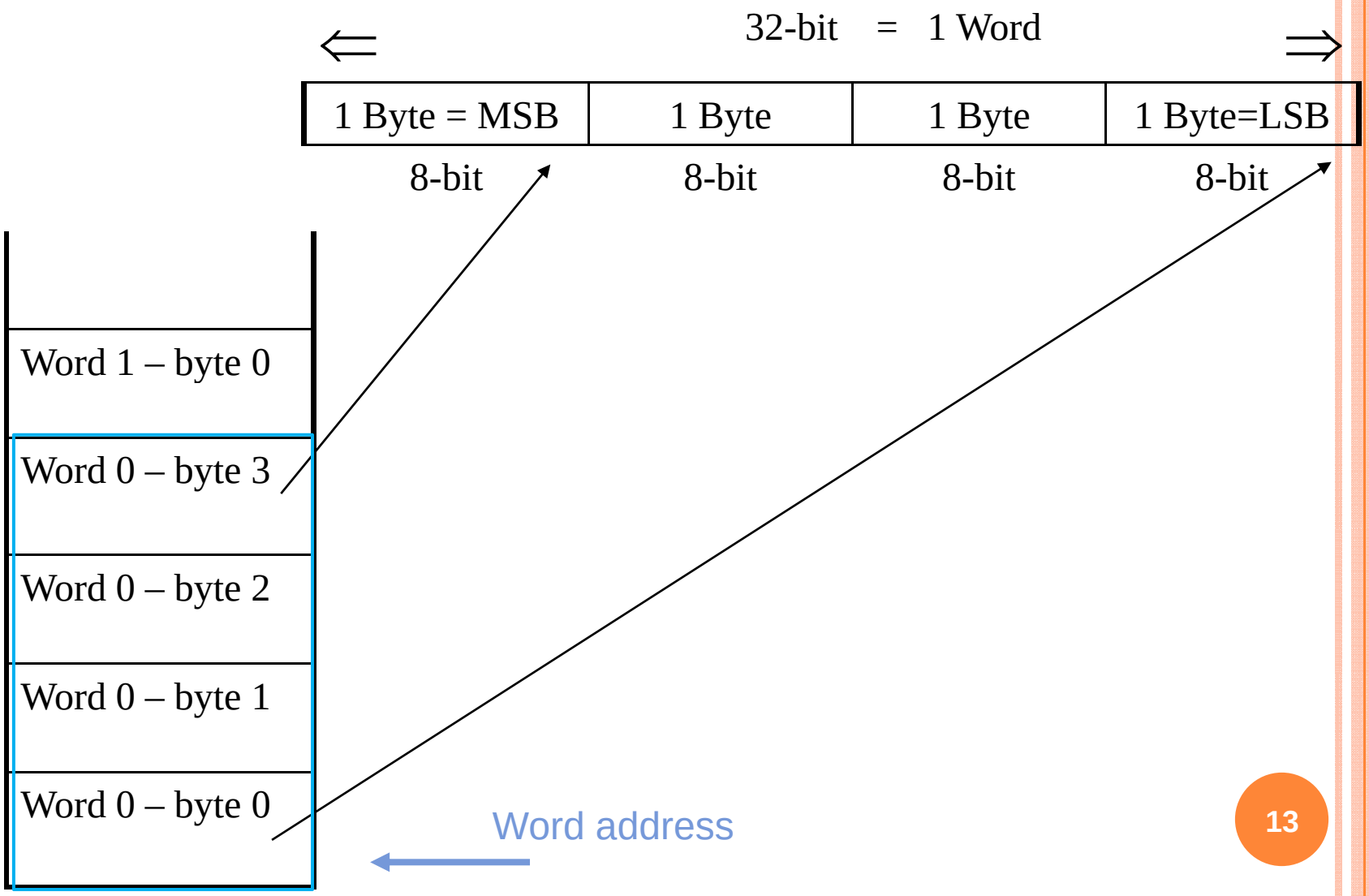


Addressing Objects: Endianess

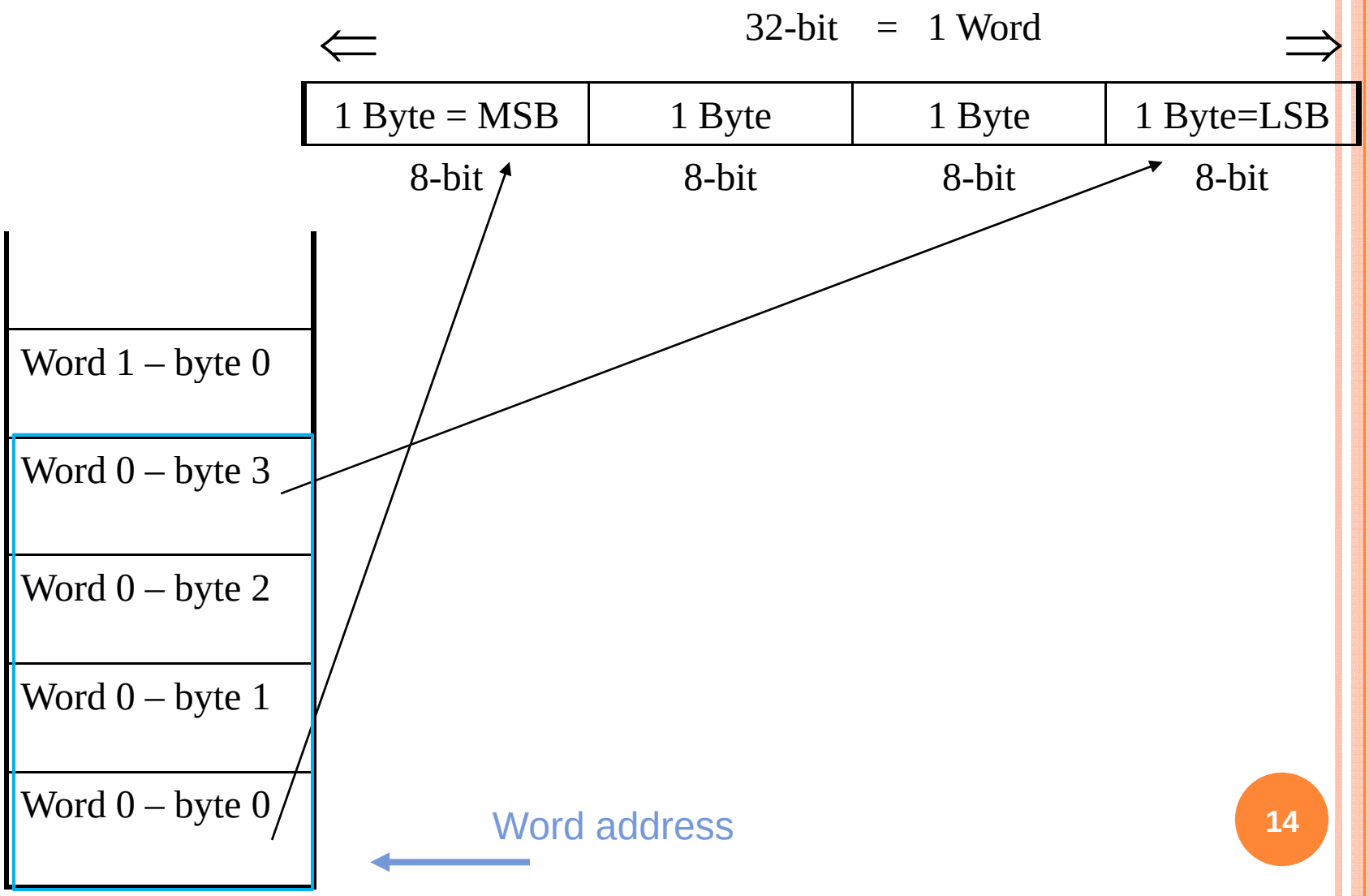
- Big Endian: address of most significant byte = word address (xx00 = Big End of word)
  - IBM 360/370, Motorola 68k, MIPS, Sparc, HP
- Little Endian: address of least significant byte = word address (xx00 = Little End of word)
  - Intel 80x86, DEC Vax, DEC Alpha (Windows NT)



DISPOSIZIONE IN MEMORIA: LITTLE ENDIAN



DISPOSIZIONE IN MEMORIA: BIG ENDIAN

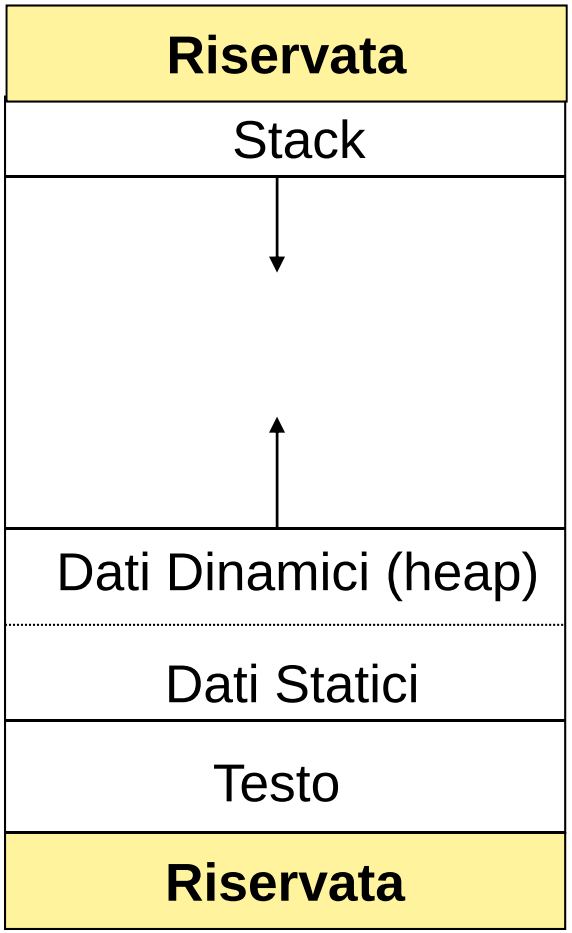


ORGANIZZAZIONE LOGICA DELLA MEMORIA

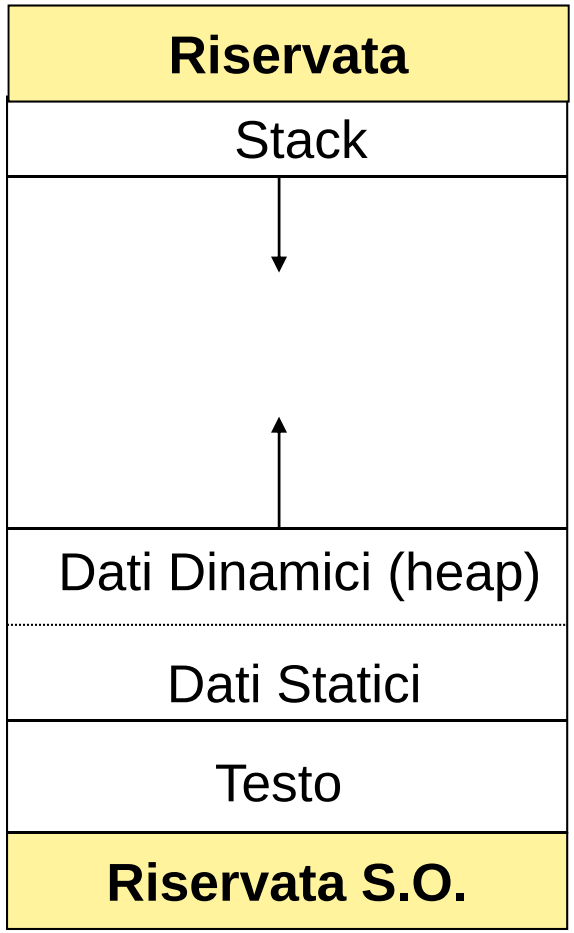
Nei sistemi basati su processore MIPS (e Intel) la memoria è solitamente divisa in **tre** parti:

- **Segmento testo**: contiene le **istruzioni** del programma
- **Segmento dati**: ulteriormente suddiviso in:
  - ***dati statici*:** contiene dati la cui dimensione è conosciuta al momento della compilazione e il cui intervallo di vita coincide con l'esecuzione del programma
  - ***dati dinamici*:** contiene dati ai quali lo spazio è allocato dinamicamente al momento dell'esecuzione del programma su richiesta del programma stesso [heap].
- **Segmento stack**: contiene lo stack allocato automaticamente da un programma durante l'esecuzione.

ORGANIZZAZIONE LOGICA DELLA MEMORIA



ORGANIZZAZIONE LOGICA DELLA MEMORIA



2 Gbyte

$7fffffff_{16}$

Max spazio di indirizzamento su 32 bit è di 2^{32} = 4Gbyte.

256Mbyte

10000000_{16}

4Mbyte

400000_{16}

0

0

REGISTRI O RAM?

- Gli operandi di una istruzione aritmetica devono risiedere nei registri che sono in numero limitato (32 nel MIPS), ma i programmi in genere richiedono un numero maggiore di variabili. Cosa succede ai programmi i cui dati richiedono più di 32 registri (32 variabili)?
  - Alcuni dati risiedono necessariamente in memoria.
- Quali dati mettere in memoria e quali tenere nei registri?
  - L'accesso ai dati in memoria richiede un tempo maggiore rispetto all'accesso dei dati nei registri.
  - Inoltre, non è possibile effettuare operazioni direttamente in memoria: è necessario copiare un dato nel registro, operare sul dato, ricopiare il dati in memoria.
  - si preferisce quindi mettere le variabili meno usate (o usate in futuro) in memoria. Tale tecnica viene chiamata ***Register Spilling***.

SOMMARIO

- Organizzazione della memoria
- Istruzioni di accesso alla memoria 
- Vettori
- Istruzioni MIPS di controllo di flusso (condizioni di uguaglianza)
- Istruzioni MIPS di controllo di flusso (condizioni di disuguaglianza).
- Costrutto switch

MIPS & ISTRUZIONI DI ACCESSO ALLA MEMORIA

- Servono inoltre istruzioni apposite per trasferire dati da memoria a registri e viceversa.

  - **lw (load word)**, per trasferire una parola di memoria in un registro della CPU. E' necessario specificare il registro destinazione e l'indirizzo di memoria contenente la parola che si vuole copiare.

  - **sw (store word)**, per trasferire il contenuto di un registro della CPU in una parola di memoria. E' necessario specificare il registro che si intende copiare e la destinazione in memoria della parola copiata.

INDIRIZZAMENTO DELLA MEMORIA

Address_final = Base_address + Offset

Un indirizzo in memoria viene identificato mediante due valori:

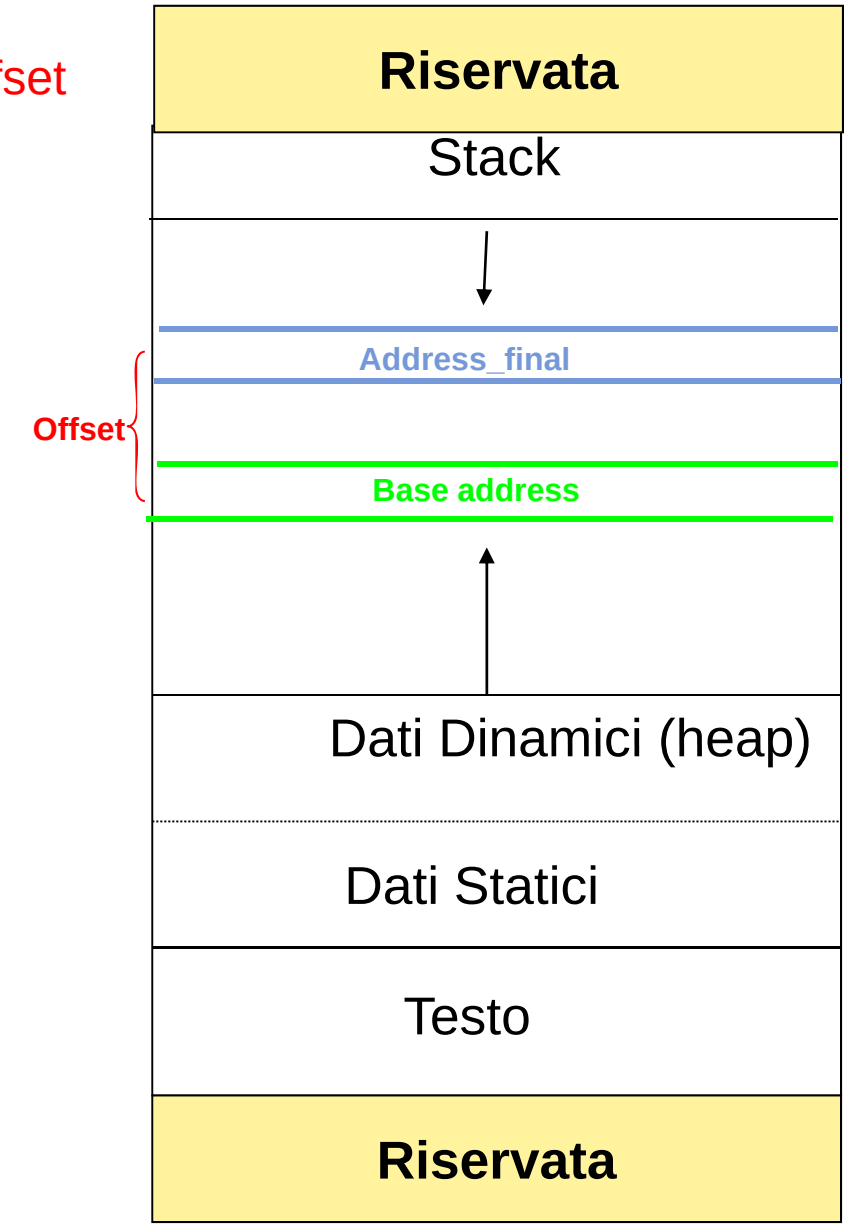
- **Un indirizzo di base**
- **Un offset**

In pratica, è come se la parola in memoria da indicizzare appartenesse ad un array.

Con A[100] indichiamo la posizione in memoria sfasata di 100 bytes rispetto ad A[0].

A[0] è l'indirizzo di base.
100 è l'offset.

Per una variabile scalare, l'offset sarà sempre pari a 0.



CARICAMENTO DALLA MEMORIA (LOAD)
SCRITTURA IN MEMORIA (STORE)

- Load (word):

  lw $s1, 100($s2) # $s1 ← M[[$s2] + 100]

- La memoria viene indicata sottoforma di vettore M[].
- L'istruzione prende il contenuto della parola che ha inizio in $s2+100 e la copia nel registro $s1. La memoria resta inalterata.

- Store (word):

  sw $s1, 100($s2) # $s1 ← M[[$s2] + 100]

- L'istruzione prende il contenuto del registro $s1 e lo copia nella parola che ha inizio in $s2+100. Il registro resta inalterato.

LW & SW: ESEMPIO DI COMPILAZIONE

Codice C: `A[12] = h + A[8];`

- Si suppone che:
  - la variabile **h** sia associata al registro **$s2**
  - l'indirizzo del primo elemento dell'array *(base address)* sia contenuto nel registro **$s3 (A[0])**

Codice MIPS:

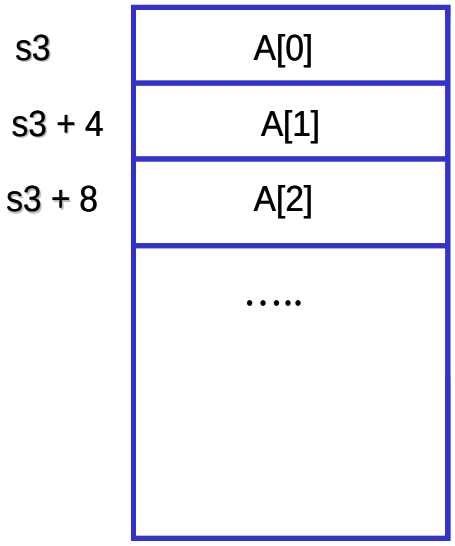
```
lw $t0, 32($s3)          # $t0 ← M[ [$s3] + 32]
add $t0, $s2, $t0        # $t0 ← $s2 + $t0
sw $t0, 48($s3)          # M[ [$s3] + 48] ← $t0
```

SOMMARIO

- Organizzazione della memoria
- Istruzioni di accesso alla memoria
- Vettori 
- Istruzioni MIPS di controllo di flusso (condizioni di uguaglianza)
- Istruzioni MIPS di controllo di flusso (condizioni di disuguaglianza).
- Costrutto switch

MEMORIZZAZIONE DI UN VETTORE

- L'elemento **i-esimo** di un array (32 byte per elemento) si troverà nella locazione:
  `br + 4 * i`
- dove:
  - **`br`** è il registro base;
  - **`i`** è l'indice ad alto livello;
  - il fattore **`4`** dipende dall'indirizzamento al byte della memoria nel MIPS



A[0]	0	1	2	3
	4	5	6	7
Offset = 2 (A[2])	8	9	10	11
	2^k-4	2^k-3	2^k-2	2^k-1

LETTURA DI UN ARRAY

- Sia A un array di N word. Realizziamo l'istruzione C: g = h + A[i]
- Si suppone che:
  - le variabili **g, h, i** siano associate rispettivamente ai registri **`$s1, $s2,`** ed **`$s4`**
  - l'indirizzo del primo elemento dell'array *(base address)* sia contenuto nel registro **`$s3`**
- L'elemento **i-esimo** dell'array si trova nella locazione di memoria di indirizzo **`($s3+ 4*i).`**
- Caricamento dell'indirizzo di A[i] nel registro temporaneo **`$t1:`**

```
muli $t1, $s4, 4        # $t1 ← 4 * i
add $t1, $t1, $s3       # $t1 ← add. of A[i]
                        # that is ($s3 + 4 * i)
```

- Per trasferire A[i] nel registro temporaneo **`$t0:`**

```
lw $t0, 0($t1)          # $t0 ← A[i]
```

- Per sommare h e A[i] e mettere il risultato in g:

```
add $s1, $s2, $t0       # g = h + A[i]
```

ARRAY: ARITMETICA DEI PUNTATORI

```
for (i=0; i<N; i+=2)
    g = h + A[i];
```

–l'indirizzo del primo elemento dell'array *(base address)* sia contenuto nel registro **$s3**

First iteration:

```
lw $t0, 0($s3)
```

All the other iterations:

```
addi $s3, $s3, 8
lw $t0, 0($s3)
```

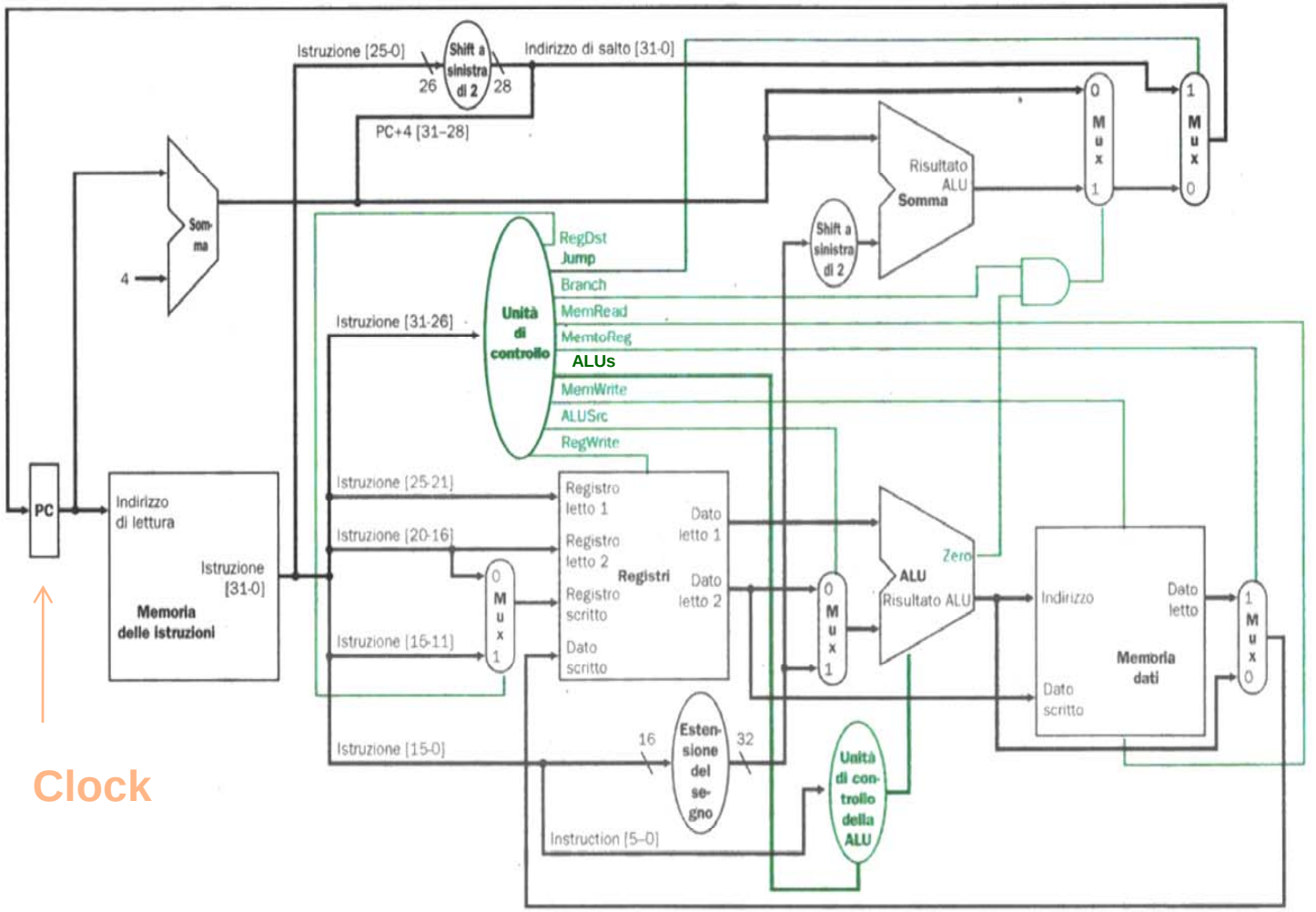
- Increment of the address of the location of A[i], inside $s3, by adding the proper offset.

ISTRUZIONI ARITMETICHE VS. LOAD/STORE

- Le istruzioni aritmetiche leggono il contenuto di due registri (operandi) , eseguono una computazione e scrivono il risultato in un terzo registro (destinazione o risultato)

- Le operazioni di trasferimento dati leggono e scrivono un solo operando senza effettuare nessuna computazione

CPU + UC A SINGOLO CICLO



SOMMARIO

- Organizzazione della memoria
- Istruzioni di accesso alla memoria
- Vettori
- Istruzioni MIPS di controllo di flusso (condizioni di uguaglianza) 
- Istruzioni MIPS di controllo di flusso (condizioni di disuguaglianza).
- Costrutto switch

STRUTTURE DI CONTROLLO DEL FLUSSO

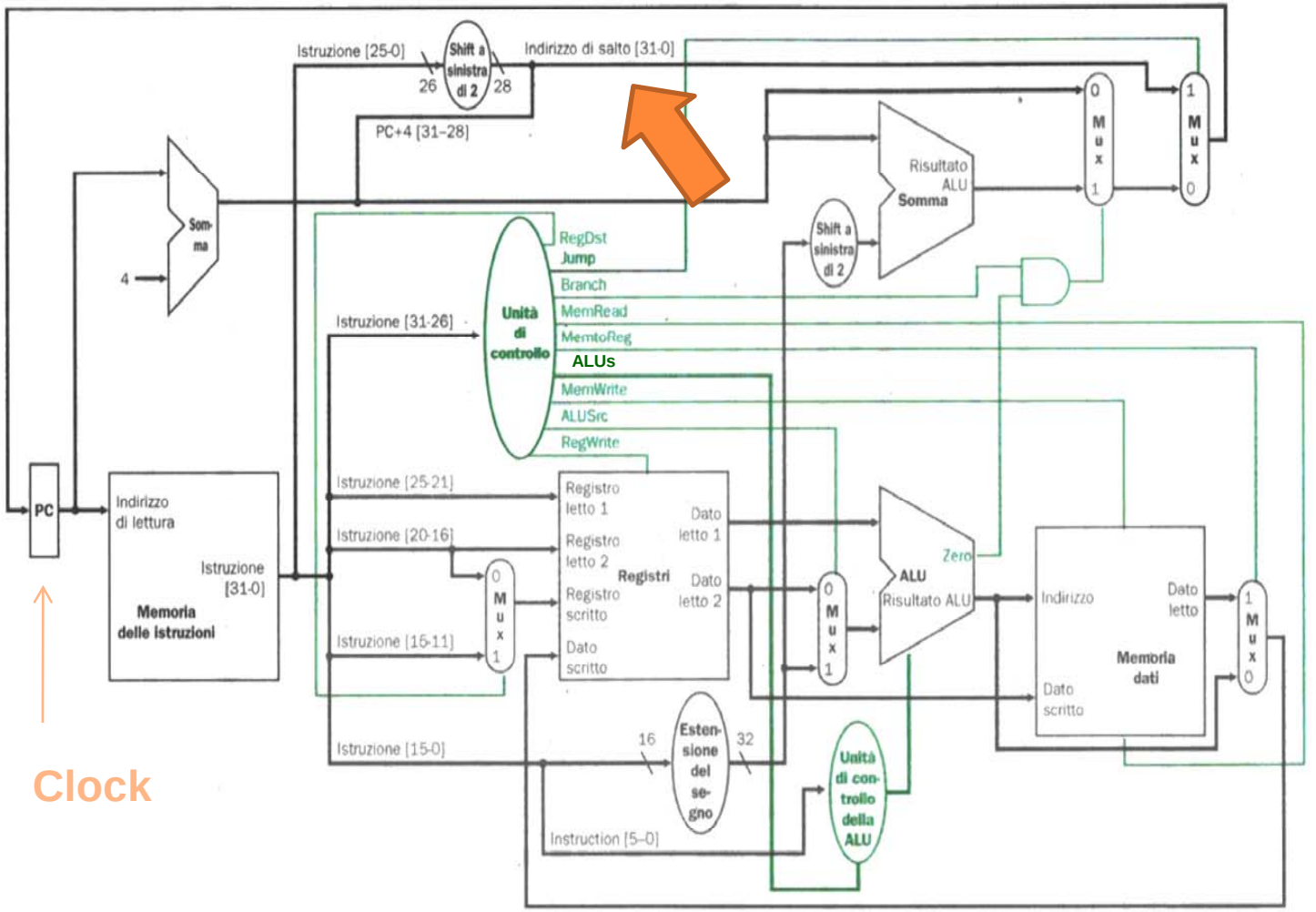
Strutture di controllo:

- cicli (for i = 0; i < n; i++)
- condizioni (if …. then …. else ….)
- salto (goto)

- Queste istruzioni (branch & jump):
  - Alterano l'ordine sequenziale di esecuzione delle istruzioni:
    - La prossima istruzione da eseguire non è l'istruzione successiva all'istruzione corrente
  - Permettono di eseguire cicli e condizioni

- In assembly le strutture di controllo sono molto semplici e primitive

CPU + UC A SINGOLO CICLO



SALTO CONDIZIONATO

- Istruzioni di salto: viene caricato un nuovo indirizzo nel contatore di programma (PC) invece dell'indirizzo seguente l'indirizzo di salto secondo l'ordine sequenziale delle istruzioni.

- Istruzioni di ***salto condizionato (conditional branch)*****:** il salto viene eseguito solo se una certa condizione risulta soddisfatta.

- Esempi: **`beq`** (*branch on equal*) e **`bne`** (*branch on not equal*)

```
beq rs, rt, L1  # go to L1 if (rs == rt)
bne rs, rt, L1  # go to L1 if (rs != rt)
```

SALTO INCONDIZIONATO

- Istruzioni di salto: viene caricato un nuovo indirizzo nel contatore di programma (PC) invece dell'indirizzo seguente l'indirizzo di salto secondo l'ordine sequenziale delle istruzioni.

- Istruzioni di ***salto incondizionato (unconditional jump):*** il salto viene sempre eseguito.
  Esempi: **`j`** (jump) e **`jr`** (jump register) e **`jal`** (jump and link)

```
j   L1          # go to L1
jr  r31         # go to add. contained in r31
jal L1          # go to L1. Save add. of next
                # instruction in reg. ra (ad
                # esempio return address).
```

ESEMPIO: IF... THEN

Codice C:

```
if (i==j)
    f=g+h;
```

- Si suppone che le variabili **f, g, h, i** e **j** siano associate rispettivamente ai registri **$s0, $s1, $s2, $s3 e $s4**

La condizione viene trasformata in codice C implementabile in Assembly:

```
    if (i != j)
        goto Etichetta;
    f=g+h;
Etichetta:
```

Codice MIPS:

```
        bne $s3, $s4, Etichetta     # go to Etichetta if i ≠ j
        add $s0, $s1, $s2           # f=g+h is skipped if i ≠ j
Etichetta:
```

ESEMPIO: IF... THEN... ELSE

Codice C:

```
if (i==j)
        f=g+h;
  else
        f=g-h;
```

- Si suppone che le variabili **f, g, h, i** e **j** siano associate rispettivamente ai registri **$s0, $s1, $s2, $s3 e $s4**

- Codice MIPS:

```
bne $s3, $s4, Else   # go to Else if i≠j
add $s0, $s1, $s2    # f=g+h skipped if i ≠ j
j   End              # go to End
Else:
sub $s0, $s1, $s2    # f=g-h skipped if i = j
End:
```

...

ESEMPIO: DO… WHILE (REPEAT)

- Codice C:

```
do
        g = g + A[i];
        i = i + j;
while (i != h)
```

- Si suppone che **g** e **h** siano associate a **$s1** e **$s2,** i e **j** associate a **$s3** e **$s4** e che **$s5** contenga il *base address* di **A = A[0].**
- Si noti che il corpo del ciclo modifica la variabile **i** ⇒ devo moltiplicare **i** per **4** ad ogni iterazione del ciclo per indirizzare il vettore A.

- Indirizzamento della memoria: indirizzo Base + Offset.
- Strategia utilizzata: sposto l'indirizzo base e considero sempre offset = 0.

ESEMPIO: DO... WHILE (REPEAT)

Codice C modificato:

```
        i = 0;
Ciclo:  g = g + A[i];
        i = i + j;
        if (i != h) goto Ciclo;
```

g e **h** -> **$s1** e **$s2**
i e **j** -> **$s3** e **$s4**
A[0] -> **$s5**

Codice MIPS:

```
        add $s3, $zero, $zero
Loop:   muli $t1, $s3, 4        # $t1 ← 4 * i : offset in byte
        add $t1, $t1, $s5       # $t1 ← address of A[i]
        lw $t0, 0($t1)          # $t0 ← A[i]
        add $s1, $s1, $t0       # g ← g + A[i]
        add $s3, $s3, $s4       # i ← i + j
        bne $s3, $s2, Loop      # go to Loop if i ≠ h
```

ESEMPIO: WHILE

Codice C:

```
// while (A[i] == k) {i=i+j};
Ciclo:        if (A[i] != k)
                    goto Fine;
              i = i + j;
              goto Ciclo;
Fine;
```

Si suppone che **i, j** e **k** siano associate a **$s3, $s4, e $s5** e che **$s6** contenga il *base address* di **A**

Codice MIPS:

```
Loop: muli $t1, $s3, 4        # $t1 ← 4 * i
      add  $t1, $t1, $s6      # $t1 ← address of A[i]
      lw  $t0, 0($t1)         # $t0 ← A[i]
      bne $t0, $s5, Exit      # go to Exit if A[i]≠k
      add $s3, $s3, $s4       # i ← i + j
      j Loop                  # go to Loop
Exit:
```

SOMMARIO

- Organizzazione della memoria
- Istruzioni di accesso alla memoria
- Vettori
- Istruzioni MIPS di controllo di flusso (condizioni di uguaglianza)
- Istruzioni MIPS di controllo di flusso (condizioni di disuguaglianza). 
- Costrutto switch

STRUTTURE DI CONTROLLO DEL FLUSSO

- Cosa posso fare se il contenuto di un registro è minore o maggiore del contenuto di un altro?

- MIPS mette a disposizione branch solo nel caso uguale o diverso, non maggiore o minore.

- Spesso è utile condizionare l'esecuzione di una istruzione al fatto che una variabile sia minore di una altra:
  - **`slt $s1, $s2, $s3`** **`# set on less than`**
    - Assegna il valore **1** a **`$s1`** se **`$s2 < $s3;`** altrimenti assegna il valore **0**

- Con **`slt, beq`** e **`bne`** si possono implementare tutti i test sui valori di due variabili **(=, !=, <, <=, >,>=)**

ESEMPIO

```
if (i < j) then
    k = i + j;
else
    k = i - j;
```

```
    if (i < j)
        flag = 1;
    If (flag == 0) goto Else;
    k = i + j;
    goto Exit;
Else:   k = i – j;
Exit:
```

```
#$s0 ed $s1 contengono i e j
#$s2 contiene k
```

```
        slt $t0, $s0, $s1

        beq $t0, $zero, Else
        add $s2, $s0, $s1
        j Exit
Else:   sub $s2, $s0, $s1
Exit:
```

CONDIZIONE DI DISUGUAGLIANZA CON PSEUDO-ISTRUZIONI (BGT)


```
if (i < j) then
    k = i + j;
else
    k = i - j;
```

```
    if (i < j)
        t = 1;
    If (t == 0) goto Else;
    k = i + j;
    goto Exit;
Else:   k = i – j;
Exit:
```

```
#$s0 ed $s1 contengono i e j
#$s2 contiene k

    bge $s0, $s1, Else

    add $s2, $s0, $s1
    j Exit
Else: sub $s2, $s0, $s1
Exit:
```

SOMMARIO

- Organizzazione della memoria
- Istruzioni di accesso alla memoria
- Vettori
- Istruzioni MIPS di controllo di flusso (condizioni di uguaglianza)
- Istruzioni MIPS di controllo di flusso (condizioni di disuguaglianza).
- Costrutto switch 

COSTRUTTO SWITCH

- Può essere implementata mediante una serie di *if-then-else*

- Alternativa: uso di una *jump address table* cioè di una tabella che contiene una serie di indirizzi di istruzioni alternative (espressività maggiore che in linguaggio ad alto livello – prossime lezioni)

```
switch(k)
 {
    case 0:    f = i + j; break;
    case 1:    f = g + h; break
    case 2:    f = g - h; break;
    case 3:    f = i - j; break;
   default:    break;
 }
```

SWITCH (PSEUDO-ASSEMBLY/C)

```
if (k < 0)
        t = 1;
else
        t = 0;
if (t == 1)                     // k < 0
        goto Exit;
t2 = k;
if (t2 == 0)                    // k = 0
        goto L0;
t2--; if (t2 == 0) // k = 1;
        goto L1;
t2--; if (t2 == 0) // k = 2;
        goto L2;
t2--; if (t2 == 0) // k = 3;
        goto L3;
goto Exit;                      // k > 3;
```

L0: f = i + j; goto Exit;
L1: f = g + h; goto Exit;
L2: f = g – h; goto Exit;
L3: f = i - j; goto Exit;

Exit:

SWITCH (ASSEMBLY)

```
#$s0, .., $s5 contengono f,..,k, k variabile di test
#$t2 contiene la costante 4

   slt $t3, $s5, $zero       # (t =) t3 = 1, nel caso in cui  $5 (= k) < 0,
   bne $t3, $zero, Exit      # if k<0
                                        #case vero e proprio
   beq $s5, $zero, L0        # se $s5 (=k) == 0, vai a L0
   addi $s5, $s5, -1         # k--
   beq $s5, $zero, L1        # se $s5 (=k) == 0, vai a L1 (inizialmente k==1)
   addi $s5, $s5, -1         # k--
   beq $s5, $zero, L2
   addi $s5, $s5, -1
   beq $s5, $zero, L3
   j Exit;                                # if k>3
L0: add $s0, $s3, $s4
    j Exit
L1: add $s0, $s1, $s2
    j Exit
L2: sub $s0, $s1, $s2
    j Exit
L3: sub $s0, $s3, $s4
Exit:
```

SWITCH (PSEUDO ASSEMBLY/C) – II VERSIONE

```
if (k < 0)
        t = 1;
else
        t = 0;
if (t == 1)                 // k < 0
        goto Exit;
t2 = 0;
if (t2 == k)                // k = 0
        goto L0;
t2++; if (t2 == k)          // k = 1;
        goto L1;
t2++; if (t2 == k)          // k = 2;
        goto L2;
t2++; if (t2 == k)          // k = 3;
        goto L3;
goto Exit;                  // k > 3;
L0: f = i + j; goto Exit;
L1: f = g + h; goto Exit;
L2: f = g – h; goto Exit;
L3: f = i - j; goto Exit;

Exit:
```

UTILIZZO 2 ELEMENTI DEL REGISTER FILE INVECE DI 1 PER MEMORIZZARE K E T2, MA È PIÙ LEGGIBILE.

SWITCH (ASSEMBLY) – II VERSIONE

```
#$s0, .., $s5 contengono f,..,k, k variabile di test
#$t2 contiene la costante 4

   slt $t3, $s5, $zero
   bne $t3, $zero, Exit      # if k<0
                                        #case vero e proprio
   mov $t3, $zero
   beq $t3, $s5, L0
   addi $t3, $t3, +1
   beq $t3, $s5, L1
   addi $t3, $t3, +1
   beq $t3, $s5, L2
   addi $t3, $t3, +1
   beq $t3, $s5, L3
   j Exit;                          # if k>3
L0: add $s0, $s3, $s4
    j Exit
L1: add $s0, $s1, $s2
    j Exit
L2: sub $s0, $s1, $s2
    j Exit
L3: sub $s0, $s3, $s4
Exit:
```