Esempio: trovare il massimo in un array

- Con indici

```
int const N = ...;
int A[N] = {...};

int max = A[0]
for (i = 1; i != N; ++i) {
    if (max < A[i]) {
        max = A[i];
    }
}
```

- Supponiamo
  - $s0 indirizzo di A
  - $s1 indirizzo di N
  - $s2 indirizzo di max
  - c'è almeno un elemento nell'array

Esempio: trovare il massimo in un array

- Con indici

```
        lw   $t4, 0($s0)        # t4 e' il max corrente
                                # t4 e' inizializzato con A[0]
        lw   $t3, 0($s1)        # carica N
        ori  $t0, $zero, 1      # i = 1

LOOP:   beq  $t0, $t3, ENDL     # se i == N vai alla fine
        sll  $t1, $t0, 2        # t1 = i * 4
        add  $t1, $t1, $s0      # t1 += A
        lw   $t2, 0($t1)        # t2 = A[i]
        slt  $t5, $t4, $t2      # test max < A[i]
        beq  $t5, $zero, ENDIF # se no, vai alla fine dell'if
        or   $t4, $t2, $zero    # altrimenti A[i] e' il nuovo max
ENDIF: addi $t0, $t0, 1         # ++i
        j    LOOP               # torna all'inizio del loop
ENDL:   sw   $t4, 0($s2)        # salva il massimo in memoria
```

Notare che la variabile max in memoria è stata aggiornata solo alla fine del ciclo e non in ogni singola iterazione

Esempio: trovare il massimo in un array

- Con puntatori

```
int const N = ...;
int A[N] = {...};

int* p = A;
int* A_end = A + N
int max = *p;
for (++p; p != A_end; ++p) {
  if (max < *p) max = *p;
}
```

- Supponiamo
  - $s0 indirizzo di A
  - $s1 indirizzo di N
  - $s2 indirizzo di max
  - c'è almeno un elemento nell'array

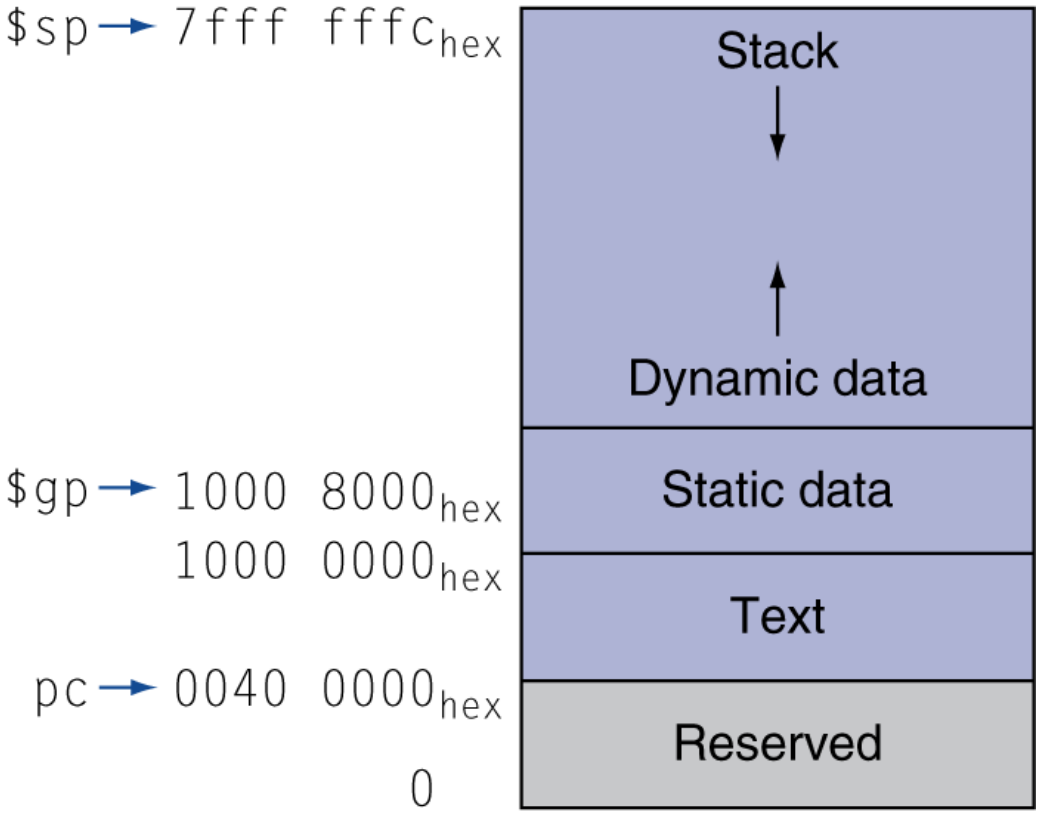
Esempio: trovare il massimo in un array

- Con puntatori

```
        or $t0, $s0, $zero     # t0 e' p
        lw $t1, 0($s1)         # carica N
        sll $t1, $t1, 2        # t1 *= 4
        add $t1, $t1, $s0      # t1 += A (t1 e' &A[N], A_end)
        lw $t4, 0($t0)         # t4 e' max
                               # t4 inizializzato con A[0]
        addi $t0, $t0, 4       # ++p
LOOP:   beq  $t0, $t1, ENDL    # se p == A_end vai alla fine
        lw   $t2, 0($t0)       # t2 = *p
        bge  $t4, $t2, ENDIF   # se max >= *p, vai alla fine dell'if
        move $t4, $t2          # altrimenti *p e' il nuovo max
ENDIF: addi $t0, $t0, 4        # ++p
        j    LOOP              # vai all'inizio del loop
ENDL:   sw $t4, 0($s2)         # salva il massimo in memoria
```

- Notare
  - l'uso delle pseudoistruzioni bge (branch greater or equal) e move.
  - che la variabile max in memoria è stata aggiornata solo alla fine del ciclo e non in ogni singola iterazione

Memory layout di un processo



- *Text*: istruzioni del programma
  - il *program counter* contiene l'indirizzo della prossima istruzione da eseguire
- *Static data*
  - variabili globali, variabili statiche, costanti, …
  - il valore di $gp permette di accedere facilmente a parte di queste variabili, usando ±offsets di 16 bit
    - $[(1000\ 0000)_{16}, (1000\ ffff)_{16}]$
- *Dynamic data*: *heap*
  - malloc in C
- *Stack*: *automatic storage*
  - es. variabili locali di una funzione

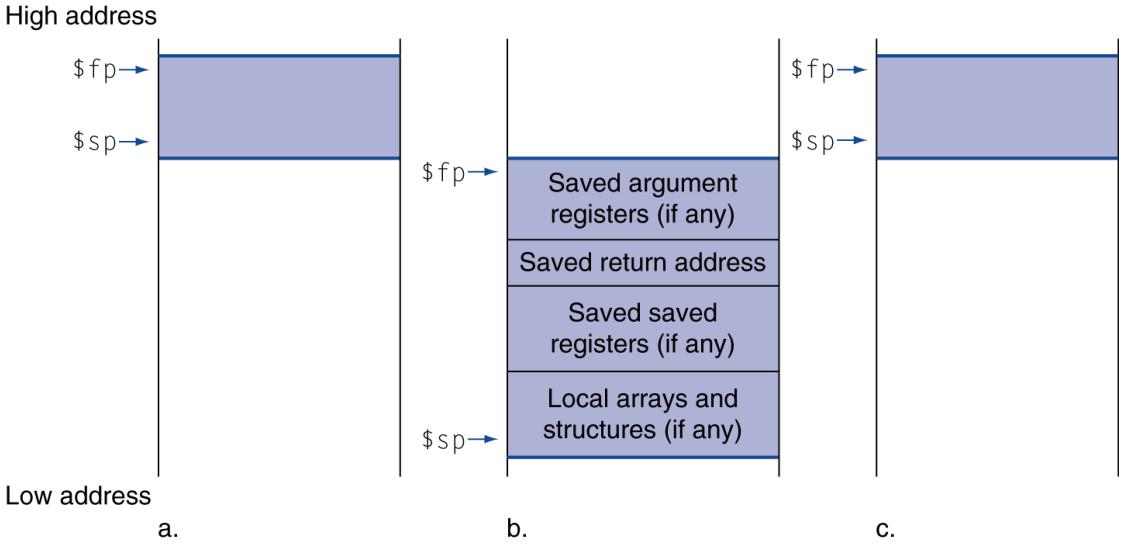
Chiamata di funzione

```
void swap(int* i, int* j) { … }
int max(int* A, int N) { … }
int increment(int i) { … }

int main()
{
  int const N = …;
  int A[N] = { … };
  int i = …;

  int o = increment(i);
  swap(&o, &i);
  int m = max(A, N);
}
```

Chiamata di funzione

- Passi necessari
  - Mettere i parametri dove la funzione chiamata li possa accedere
  - Trasferire il controllo alla funzione
  - Acquisire spazio di memoria per l'esecuzione della funzione
  - Eseguire le operazioni della funzione
  - Mettere il risultato dove la funzione chiamante li possa accedere
  - Ritornare il controllo al chiamante

Stack



NB: questa disposizione è un esempio

- Stack: coda LIFO (Last-In, First-Out)
- Lo stack di solito cresce verso il basso
  - da indirizzi alti a indirizzi bassi
- Per ogni chiamata di funzione viene allocata sullo stack un'area dedicata (*frame* o *activation record*)
  - il frame viene deallocato alla fine della chiamata

Cosa ci va in uno stack frame?

- Dipende
  - dall'hardware, dal compilatore, …
  - definisce un'**interfaccia**
- Dati necessari per la gestione della catena delle chiamate
  - frame pointer ($fp) se usato
  - return address ($ra) se necessario
- Argomenti passati alla funzione
  - tutti o solo quelli che non stanno nei registri $a0 - $a3
  - potrebbe essere allocato spazio anche per salvare $a0 - $a3
- Registri da preservare durante una chiamata di funzione
  - a cura del chiamante o del chiamato
- Variabili locali (automatiche)

Cosa ci va in uno stack frame?

- Noi useremo un approccio semplificato
- Allochiamo un frame solo se necessario
- Per le variabili locali di una funzione useremo spesso solo registri, senza allocare spazio sullo stack
- Non usiamo il frame pointer
  - l'accesso alla memoria sullo stack avviene solo attraverso lo stack pointer
- Ci limitiamo a massimo 4 argomenti
  - usiamo solo i registri $a0 – $a3, non la memoria
- Se abbiamo bisogno di preservare registri, usiamo quelli saved ($s0 – $s7) che sono preservati dal chiamato
- Ignoriamo problemi di allineamento, padding, ecc.

Uso dei registri

Name	Register number	Usage	Preserved on call?
$zero	0	The constant value 0	n.a.
$v0–$v1	2–3	Values for results and expression evaluation	no
$a0–$a3	4–7	Arguments	no
$t0–$t7	8–15	Temporaries	no
$s0–$s7	16–23	Saved	yes
$t8–$t9	24–25	More temporaries	no
$gp	28	Global pointer	yes
$sp	29	Stack pointer	yes
$fp	30	Frame pointer	yes
$ra	31	Return address	yes

Istruzioni di chiamata di funzione

- Chiamata: jump and link

  `jal function_label`

  - L'indirizzo dell'istruzione successiva è salvato in $ra
  - Salta all'indirizzo di destinazione
- Ritorno: jump register

  `jr $ra`

  - Copia $ra nel program counter

Esempio: funzione swap

```
void swap(int* l, int* r)
{
  int t = *l;
  *l = *r;
  *r = t;
}
```

- Argomenti in $a0 (l) e $a1 (r)
- Non c'è valore di ritorno
- Funzione terminale
  - non chiama altre funzioni

Esempio: funzione swap

- Versione con caller-saved (temporary) registers

```
swap:
    lw $t0, 0($a0)   # carica l in t0
    lw $t1, 0($a1)   # carica r in t1
    sw $t0, 0($a1)   # salva t0 in r
    sw $t1, 0($a0)   # salva t1 in l

    jr $ra           # ritorna il controllo al chiamante
```

- Notare la non necessità di allocare uno stack frame perché non c'è bisogno di salvare e poi ripristinare registri, né di tenere variabili locali in memoria

Esempio: funzione swap

- Versione con callee-saved (saved) registers

```
swap:
    addiu $sp, $sp, -8 # fa spazio sullo stack per due registri
    sw    $s0, 0($sp)  # salva $s0
    sw    $s1, 4($sp)  # salva $s1
    lw    $s0, 0($a0)  # carica l in s0
    lw    $s1, 0($a1)  # carica r in s1
    sw    $s0, 0($a1)  # salva s0 in r
    sw    $s1, 0($a0)  # salva s1 in l
    lw    $s0, 0($sp)  # ripristina $s0
    lw    $s1, 4($sp)  # ripristina $s1
    addiu $sp, $sp, 8  # dealloca il frame
    jr    $ra          # ritorna il controllo al chiamante
```

- E' necessario allocare uno stack frame per salvare e poi ripristinare i saved registers
  - non serve invece salvare il return address, né gli argomenti

Esempio: funzione increment

```
int increment(int i)
{
  int r = i + i;
  return r;
}
```

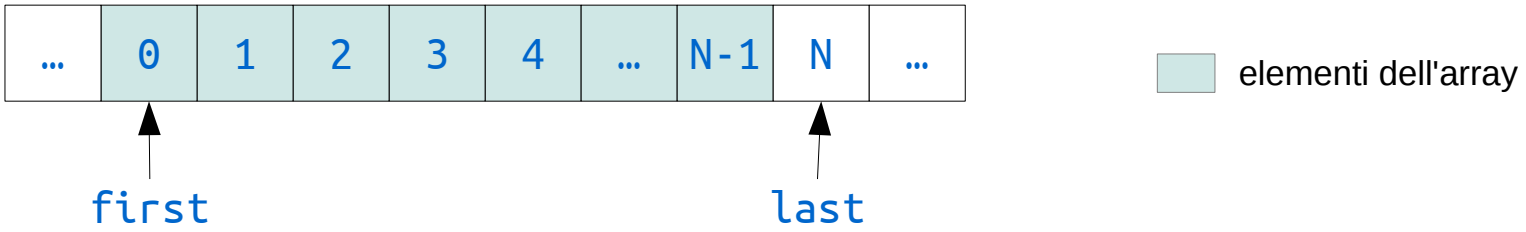
- Argomento in $a0
- Risultati parziali in $s0
  - bisogna salvare il valore precedente di $s0 nel frame sullo stack
- Una variabile locale r
- Risultato in $v0

Esempio: funzione increment

```
increment:
    addiu $sp, $sp, -8      # alloca uno stack frame per un registro
                            # e una variabile locale
    sw    $s0, 0($sp)       # salva $s0 sullo stack

    addi  $s0, $a0, 1       # r = i + 1
    sw    $s0, 4($sp)       # salva s0 in memoria nella var r
    lw    $v0, 4($sp)       # legge v0 dalla memoria

    lw    $s0, 0($sp)       # ripristina s0
    addiu $sp, $sp, 8       # dealloca lo stack frame
    jr    $ra               # ritorna il controllo al chiamante
```

- r mostra l'allocazione e l'uso di variabili locali in una funzione
  - in questo caso il salvataggio in r di $s0 è gratuito

Esempio: funzione reverse




```
void reverse(int* first, int* last)
{
  if (first == last) return;
  --last;
  while (first < last) {
    swap(first, last);
    ++first;
    --last;
  }
}
```

- Funzione che chiama un'altra funzione
  - bisogna salvare i registri argomenti e il return address

Esempio: funzione reverse

```
reverse: # prologo
    addiu $sp, $sp, -20
    sw    $ra,  0($sp)
    sw    $a0,  4($sp)
    sw    $a1,  8($sp)
    sw    $s0, 12($sp)
    sw    $s1, 16($sp)
...

r_done: # epilogo
    lw    $ra,  0($sp)
    lw    $a0,  4($sp)
    lw    $a1,  8($sp)
    lw    $s0, 12($sp)
    lw    $s1, 16($sp)
    addiu $sp, $sp, 20
    j     $ra
```

- Allochiamo un frame sufficiente a contenere 5 registri
  - 20 byte
- Dobbiamo chiamare altre funzioni, quindi salviamo:
  - il registro con il return address
  - i registri con gli argomenti
- Inoltre salviamo nel prologo e ripristiniamo nell'epilogo alcuni registri saved che usiamo

Esempio: funzione reverse

```
    move  $s0, $a0          # first
    move  $s1, $a1          # last
    beq   $s0, $s1, r_done  # se first == last vai alla fine
    addiu $s1, $s1, -4      # --last
r_loop:
    bge   $s0, $s1, r_done  # se first >= last vai alla fine
    move  $a0, $s0
    move  $a1, $s1
    jal   swap              # swap(first, last)
    addiu $s0, $s0,  4      # ++first
    addiu $s1, $s1, -4      # --last
    j     r_loop
```

- Notare l'uso di pseudo-istruzioni
- Se avessimo avuto bisogno di usare i valori originali degli argomenti a `reverse()`, avremmo potuto prenderli da `4($sp)` e `8($sp)`

struct

```
struct S {
  int i, j, k;
};

void fun()
{
  struct S s;
  s.i = s.j = s.k = 1;
  ...
  ... = s.i;
  ...
}
```

- Ricordarsi che lo stack cresce verso il basso

```
fun:
    addiu $sp, $sp, -16
    sw    $ra, 0($sp)
    # 3 parole del frame contengono s

    li    $t0, 1
    sw    $t0,  4($sp)   # s.i = ...
    sw    $t0,  8($sp)   # s.j = ...
    sw    $t0, 12($sp)   # s.k = ...
...
    lw    $t1,  4($sp)   # per leggere s.i
...
f_end:
    lw    $ra, 0($sp)
    addiu $sp, $sp, 16
    jr    $ra
```

Floating point

- Rappresentazione di numeri non interi
  - Tipi `float` e `double` in C
- Consideriamo solo la parte “decimale”
  - sappiamo già gestire la parte intera

$$\ldots . c_{-1} c_{-2} \ldots c_{-n} \ldots = \ldots + c_{-1} \cdot B^{-1} + c_{-2} \cdot B^{-2} + \ldots + c_{-n} \cdot B^{-n} + \ldots$$

- NB: i floating point sono molto di più di quanto trattato di seguito

Valore di un numero rappresentato in base B

- Notare che per rappresentare tale valore scegliamo un'altra base
  - tipicamente base 10, ma non necessariamente
  - già per esprimere la base con simboli numerici dovrei scegliere una base

$$\mathbf{(0.456)_{otto} = (4 \cdot 8^{-1} + 5 \cdot 8^{-2} + 6 \cdot 8^{-3})_{dieci} = (0.58984375)_{dieci}}$$

$$(0.456)_{otto} = (4 \cdot 8^{-1} + 5 \cdot 8^{-2} + 6 \cdot 8^{-3})_{sedici} = (0.97)_{sedici}$$

$$\mathbf{(0.1101)_{due} = (1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-4})_{10} = (0.8125)_{10}}$$

$$(0.8125)_{dieci} = (8 \cdot A^{-1} + 1 \cdot A^{-2} + 2 \cdot A^{-3} + 5 \cdot A^{-4})_{sedici} = (0.D)_{sedici}$$

$$(0.75)_{dieci} = (111 \cdot 1010^{-1} + 101 \cdot 1010^{-2})_{due} = (0.11)_{due}$$

Rappresentazione di un numero in base B

- $.f = (0.b_{-1}\ldots b_{-m}\ldots)_B$

  $= b_{-1}\cdot B^{-1} + b_{-2}\cdot B^{-2} + b_{-3}\cdot B^{-3} + \ldots$

- Se moltiplico $.f$ per B e prendo la parte intera ottengo b_{-1}

  $.f\cdot B = b_{-1} + b_{-2}\cdot B^{-1} + b_{-3}\cdot B^{-2} + \ldots = (b_{-1}.b_{-2}b_{-3}\ldots)_B$

- Se moltiplico per B la parte decimale di $.f\cdot B$ e prendo la parte intera ottengo b_{-2} e così via

- In generale la procedura non termina e ci si ferma quando si raggiunge il numero di cifre voluto

Rappresentazione di un valore in base B

- Esempi:
  - Notare che partiamo di nuovo da una rappresentazione del valore, per comodità in base 10

$$
\begin{aligned}
0.8125 \cdot 2 &= 1.625 \rightarrow 1 \\
0.625 \cdot 2 &= 1.25 \rightarrow 1 \\
0.25 \cdot 2 &= 0.5 \rightarrow 0 \\
0.5 \cdot 2 &= 1 \rightarrow 1
\end{aligned}
\qquad
0.8125 \cdot 16 = 13 \rightarrow D
$$

$$(0.8125)_{10} = (0.1101)_2 = (0.D)_{16}$$

$$
\begin{aligned}
0.2 \cdot 2 &= 0.4 \rightarrow 0 \\
0.4 \cdot 2 &= 0.8 \rightarrow 0 \\
0.8 \cdot 2 &= 1.6 \rightarrow 1 \\
0.6 \cdot 2 &= 1.2 \rightarrow 1 \\
0.2 \cdot 2 &= 0.4 \ldots
\end{aligned}
\qquad
0.2 \cdot 5 = 1 \rightarrow 1
$$

$$(0.2)_{10} = (0.\overline{0011})_2 = (0.1)_5$$

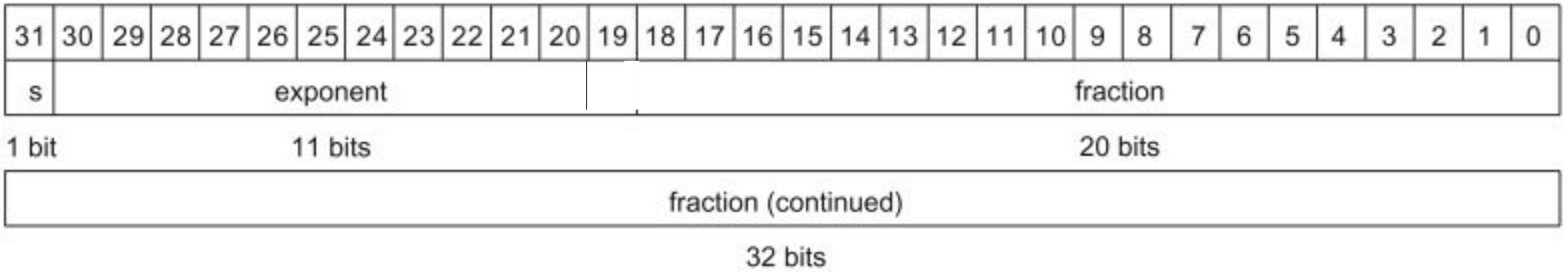
Rappresentazione floating point

- Necessariamente un'approssimazione dei numeri reali
  - tipi float e double in C
- Notazione scientifica
  - $-4.8744 \cdot 10^{-9}$
  - $-1343.8383 \cdot 10^{5}$
  - $+0.0002$
- Notazione scientifica normalizzata
  - $\pm d.dddd \cdot 10^{e}$ (decimale)
  - $\pm 1.bbbb \cdot 2^{e}$ (binaria)
- Definita nello standard IEEE 754
  - singola (32 bit) e doppia (64 bit) precisione

Rappresentazione floating point

32 bit

64 bit



$$v = (-1)^{s} \cdot (1 + \text{fraction}) \cdot 2^{(\text{exponent} - \text{bias})}$$

Rappresentazione floating point

$$v = (-1)^s \cdot (1 + \text{fraction}) \cdot 2^{(\text{exponent} - \text{bias})}$$

- s è il sign bit (`1` → numero negativo, `0` → non negativo)
- Il bit prima della virgola è implicito a 1
  - 1 + fraction si chiama mantissa (significand)
  - `1.0 ≤ |mantissa| < 2.0`
- L'esponente è in notazione *biased*: valore + bias
  - `00...00` è il valore più piccolo, *bias* è `0`, `11...11` è il valore più grande
  - FP: bias = `127 (0111 1111)`; DP: bias = `1023 (011 1111 1111)`
- La rappresentazione consente di confrontare i floating point come se fossero interi in complemento a 2

Floating point: esempio

- Rappresentare il numero $(-0.75)_{10} = (-0.11)_2$

  $(-0.11)_2 = (-1)^1 \cdot 1.1 \cdot 2^{-1}$
- S = 1
- Fraction = 10000...0000
- Exponent = -1 + bias
  - SP: $-1 + 127 = 126 = (01111110)_2$
  - DP: $-1 + 1023 = 1022 = (01111111110)_2$
- SP: 1 01111110 100000...00000000000
- DP: 1 01111111110 100000...00000000000

Floating point: esempio

- Che numero è rappresentato dal SP float?

  `1 10000001 01000...00`
- `S = 1`
- `Fraction = 01000...00`
- `Exponent = 10000001 = (129)`$_{10}$

$$v = (-1)^s \cdot (1 + \text{fraction}) \cdot 2^{(\text{exponent} - \text{bias})}$$
$$= (-1)^1 \cdot (1 + 0.01)_2 \cdot 2^{(129 - 127)}$$
$$= -1.25 \cdot 2^2$$
$$= -5 \text{ (come FP)}$$

Riepilogo codifica floating point

Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	Nonzero	0	Nonzero	± denormalized number
1–254	Anything	1–2046	Anything	± floating-point number
255	0	2047	0	± infinity
255	Nonzero	2047	Nonzero	NaN (Not a Number)

Intervallo singola precisione

- Esponenti `00000000` e `11111111` sono riservati
- Valore più piccolo (in valore assoluto)
  - Esponente: `00000001`

    → esponente reale = `1 - 127 = -126`
  - Frazione: `000…00` → mantissa = `1.0`
  - $\pm 1.0 \cdot 2^{-126} \approx \pm 1.2 \cdot 10^{-38}$ $\quad (38 \approx 126 \cdot \log_{10}(2) \approx 126 \cdot 0.3)$
  - Sotto è underflow (denormal a parte)
- Valore più grande
  - Esponente: `11111110`

    → esponente reale = `254 - 127 = +127`
  - Frazione: `111…11` → mantissa ≈ `2.0`
  - $\pm 2.0 \cdot 2^{+127} \approx \pm 3.4 \cdot 10^{+38}$
  - Sopra è overflow

Intervallo doppia precisione

- Esponenti `00000000000` e `11111111111` sono riservati
- Valore più piccolo
  - Esponente: `00000000001`

    → esponente reale = `1 - 1023 = -1022`
  - Frazione: `000…00` → mantissa = `1.0`
  - $\pm 1.0 \cdot 2^{-1022} \approx \pm 2.2 \cdot 10^{-308}$
  - Sotto è underflow (denormal a parte)
- Valore più grande
  - Esponente: `11111111110`

    → esponente reale = `2046 - 1023 = +1023`
  - Frazione: `111…11` → mantissa ≈ `2.0`
  - $\pm 2.0 \cdot 2^{+1023} \approx \pm 1.8 \cdot 10^{+308}$
  - Sopra è overflow

Addizione floating point

- Esempio con 4 cifre binarie

  $1.000_2 \cdot 2^{-1} + -1.110_2 \cdot 2^{-2}$ $(0.5 + -0.4375)_{10}$

- Allineare i punti decimali (binari)
  - Shift del numero con l'esponente più piccolo

    $1.000_2 \cdot 2^{-1} + -0.111_2 \cdot 2^{-1}$

- Sommare le mantisse

  $1.000_2 \cdot 2^{-1} + -0.111_2 \cdot 2^{-1} = 0.001_2 \cdot 2^{-1}$

- Normalizzare il risultato e controllare over/underflow

  $1.000_2 \cdot 2^{-4}$ (no over/underflow)

- Arrotondare e rinormalizzare se necessario

  $1.000_2 \cdot 2^{-4}$ (nessun cambiamento) $= 0.0625$

Moltiplicazione floating point

- Esempio con 4 cifre binarie

  $1.000_2 \cdot 2^{-1} \cdot -1.110_2 \cdot 2^{-2}$ (0.5 · -0.4375)

- Sommare gli esponenti
  - Unbiased: -1 + -2 = -3
  - Biased: -3 + 127 = +124
- Moltiplicare i significands

  $1.000_2 \cdot 1.110_2 = 1.110_2 \rightarrow 1.110_2 \cdot 2^{-3}$

- Normalizzare il risultato e controllare over/underflow

  $1.110_2 \cdot 2^{-3}$ (nessun cambiamento e no over/underflow)

- Arrotondare e rinormalizzare se necessario

  $1.110_2 \cdot 2^{-3}$ (nessun cambiamento)

- Determinare il segno: + · - → -

  $-1.110_2 \cdot 2^{-3} = -0.21875$

Istruzioni floating point

- L'hardware FP è considerato un coprocessore
- Registri FP separati
  - 32 in singola precisione: $f0, $f1, …, $f31
  - Accoppiati per doppia precisione: $f0/$f1, $f2/$f3, …, $f30/$f31
- Le istruzioni FP operano solo su registri FP

- Istruzioni FP per load e store
  - `lwc1, ldc1, swc1, sdc1`

```
ldc1 $f8, 32($sp) # riempie $f8/$f9 con 2 word a
                  # partire da 32($sp)
```

Istruzioni floating point

- Aritmetica in singola precisione
  - `add.s, sub.s, mul.s, div.s`

    `add.s $f0, $f1, $f6`
- Aritmetica in doppia precisione
  - `add.d, sub.d, mul.d, div.d`

    `mul.d $f4, $f4, $f6`
- Confronti in singola e doppia precisione
  - `c.xx.s, c.xx.d` (xx è `eq, lt, le`, ...)
  - Assegna 1 (true) o `0` (false) al bit *FP condition-code*

    `c.lt.s $f3, $f4`
- Branch in base al valore true o false del FP condition-code
  - `bc1t, bc1f`

    `bc1t TargetLabel`

Sincronizzazione

- Consideriamo due processori/processi/thread/agenti che condividiono un'area di memoria
  - r1, r2 registri (privati)
  - x, y locazioni di memoria (condivise)

P1	P2
`y = 1;` `r1 = x;`	`x = 1;` `r2 = y;`

Esecuzione 1	Esecuzione 2	Esecuzione 3

`y = 1;`
`r1 = x;`
`x = 1;`
`r2 = y;`
`// r1 == 0, r2 == 1`
	`x = 1;`
	`r2 = y;`
	`y = 1;`
	`r1 = x;`
	`// r1 == 1, r2 == 0`
		`y = 1;`
		`x = 1;`
		`r2 = y;`
		`r1 = x;`
		`// r1 == 1, r2 == 1`

- Esiste una *data race* se P1 e P2 non si sincronizzano
  - Il risultato dipende dall'ordine degli accessi

Sincronizzazione

- Per la sincronizzazione è necessario supporto dall'hardware sotto forma di operazioni *atomiche* di read/write in memoria
  - Non sono ammessi altri accessi alla locazione di memoria interessata tra la read e la write
- Diversi approcci
  - singola istruzione
    - esempio: swap atomico registro ↔ memoria
  - o una coppia di istruzioni atomiche
    - adottato in MIPS

Load linked, Store conditional

- Load linked: `ll rt, offset(rs)`
- Store conditional: `sc rt, offset(rs)`
  - Ha successo se il contenuto della locazione non è cambiato da quando è stata eseguita la `ll`

    → ritorna `1` in `rt`
  - Fallisce se il contenuto è cambiato

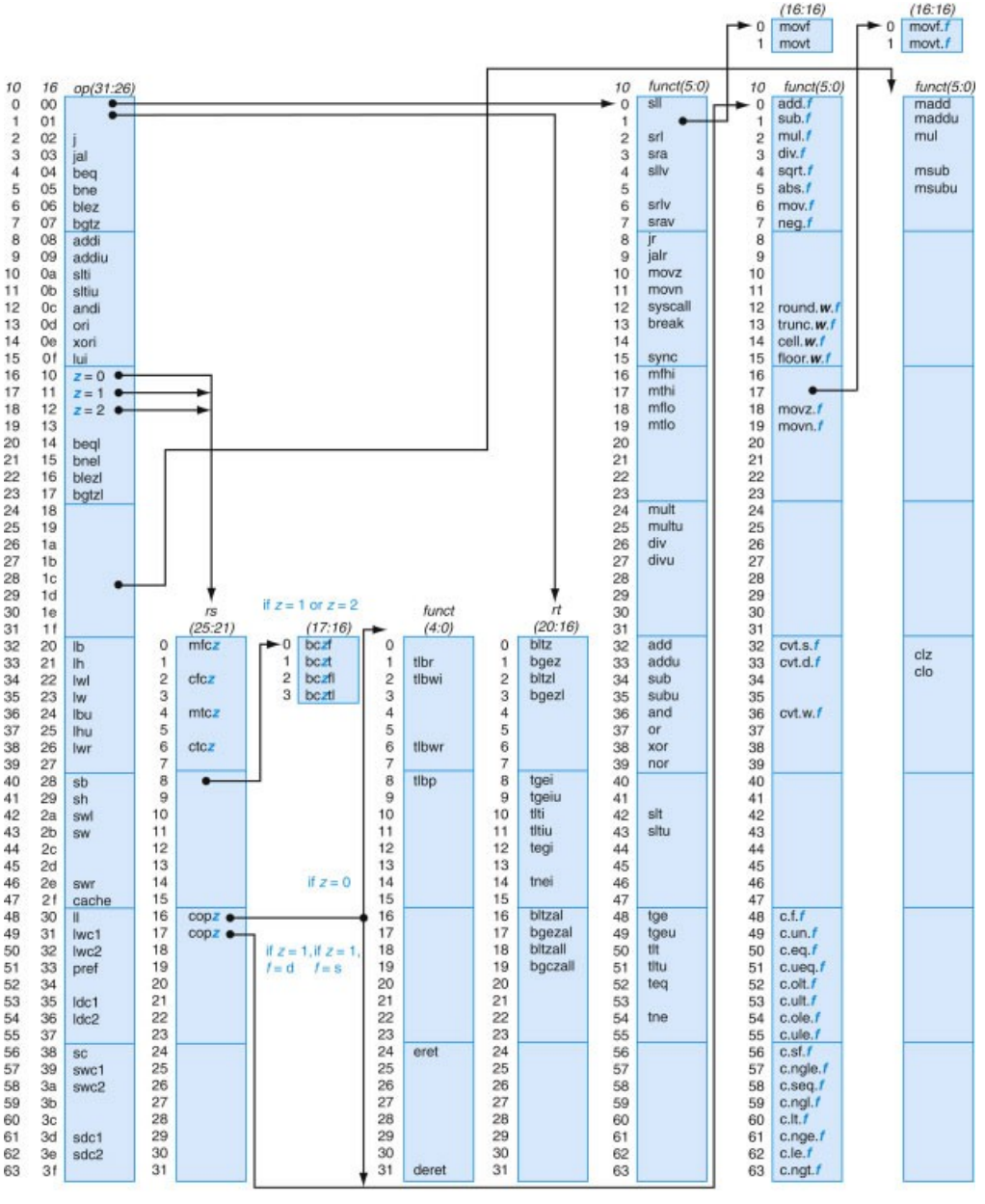
    → ritorna `0` in `rt`
- Esempio: atomic swap (to test/set lock variable)

```
# scambia il valore in $s0 con il valore in Mem[$s1]
try: move $t0, $s0         # t0 = s0
     ll   $t1, 0($s1)      # t1 = Mem[s1]
     sc   $t0, 0($s1)      # Mem[s1] = t0 (= s4)
     beq  $t0, $zero, try # fallimento, riprova
     move $s0, $t1         # successo, s0 = t1 (= Mem[s1])
```

Rappresentazione delle istruzioni

- Anche le istruzioni sono rappresentate in binario
  - codice macchina
- Tutte le istruzioni MIPS sono rappresentate in 32 bit
  - codice dell'istruzione, registri, costanti, indirizzi, …
- Tre formati
  - **R** per istruzioni aritmetico/logiche con tre registri
  - **I** per istruzioni aritmetico/logiche con una costante e per istruzioni di load/store
  - **J** per istruzioni di salto

Codifica istruzioni



non è da imparare a memoria

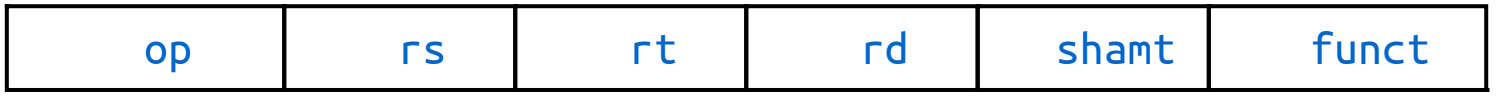
Formato R

- Campi
  - op: codice dell'operazione (opcode)
  - rs: numero del primo registro sorgente
  - rt: numero del secondo registro sorgente
  - rd: numero del registro destinazione
  - shamt: numero di bit dello shift
  - funct: codice della funzione (estende l'opcode)

R[rd] = R[rs] op/funct R[rt]

R[rd] = R[rt] << shamt

Formato R - esempio

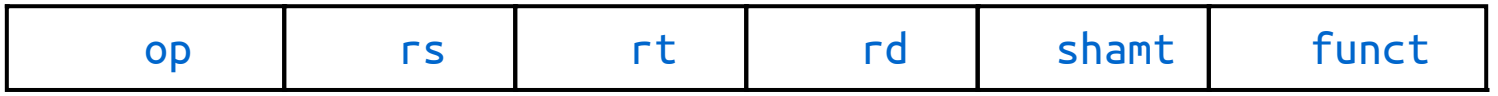



```
add $t0, $s0, $s1
```

R-format	16	17	8	0	add
000000	10000	10001	01000	00000	100000

```
0000 0010 0001 0001 0100 0000 0010 0000 = 0x02114020
```

Formato R - esempio

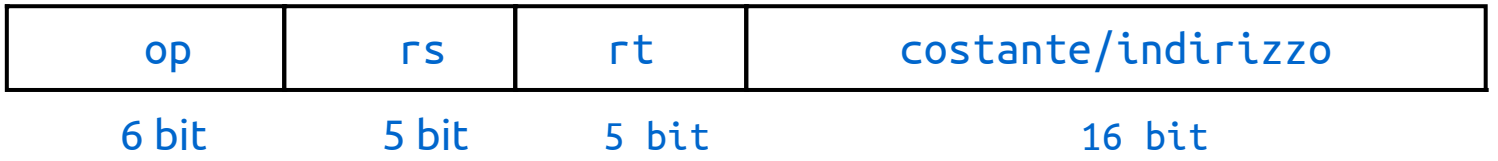



```
sll $t0, $s0, 2
```

R-format	0	16	8	2	sll
000000	00000	10000	01000	00010	000000

```
0000 0000 0001 0000 0100 0000 1000 0000 = 0x00104080
```

Formato I



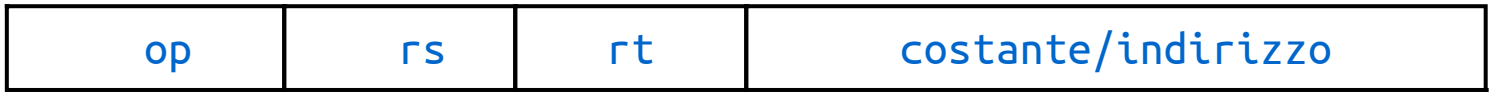
- Campi
  - op: codice dell'operazione (opcode)
  - rs: numero del registro sorgente
  - rt: numero del registro destinazione (sorgente per una store)
  - costante/indirizzo: operando *immediate* (sign-extended)
    - 16 bit → $[0, 2^{16}-1]$ (unsigned), $[-2^{15}, +2^{15}-1]$ (signed)

R[rt] = R[rs] op costante

R[rt] = Mem[R[rs] + indirizzo] # load word (lw)

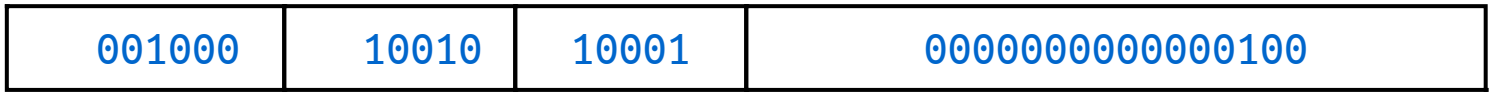
Mem[R[rs] + indirizzo] = R[rt] # store word (sw)

Formato I - esempio



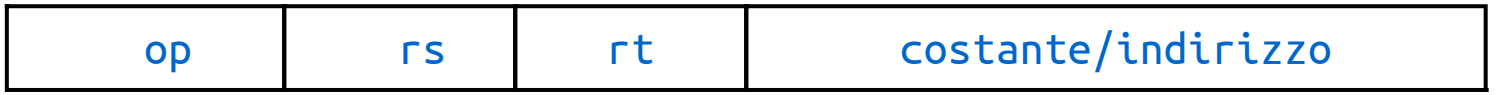
6 bit 5 bit 5 bit 16 bit

```
addi $s1, $s2, 4
```

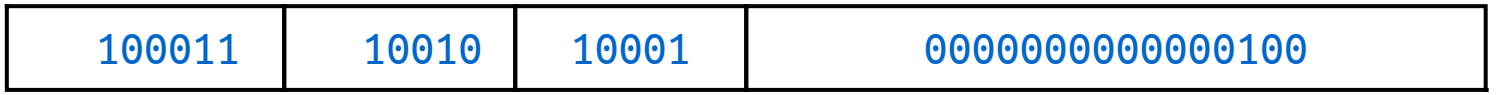



```
0010 0010 0101 0001 0000 0000 0000 0100 = 0x22510004
```

Formato I - esempio

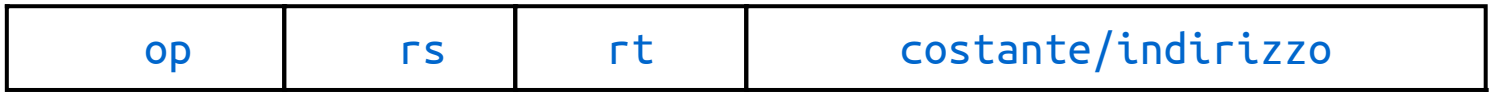



```
lw $s1, 4($s2)
```

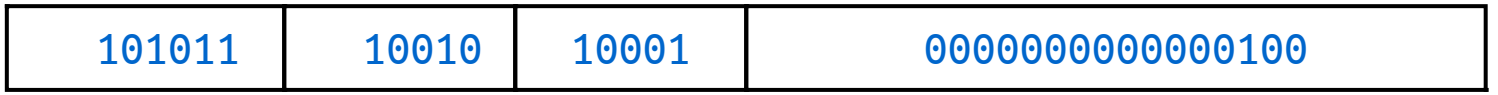



```
1000 1110 0101 0001 0000 0000 0000 0100 = 0x8E510004
```

Formato I - esempio

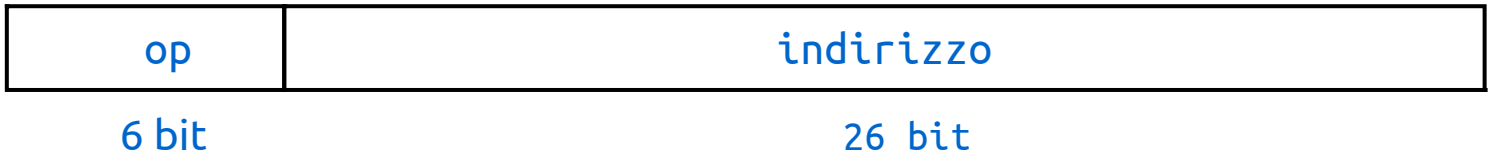



```
sw $s1, 4($s2)
```




```
1010 1110 0101 0001 0000 0000 0000 0100 = 0xAE510004
```

Formato J



- Campi
  - op: codice dell'operazione (opcode)
  - indirizzo: destinazione del salto
  - l'indirizzo si riferisce a istruzioni

    → multiplo di 4 byte → i 2 bit meno significativi sarebbero `0`

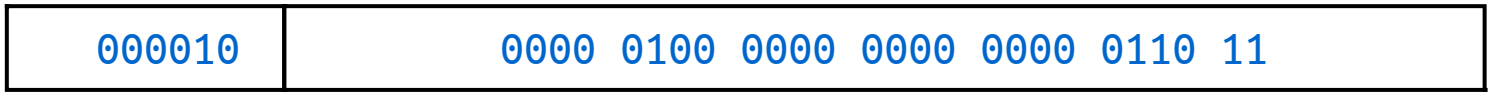
    → posso risparmiarli

    → è come se l'indirizzo fosse a 28 bit

    → l'intervallo di indirizzi è $[0, 2^{28}-1]$, ma a multipli di 4
    - i 4 bit più significativi sono presi dal program counter

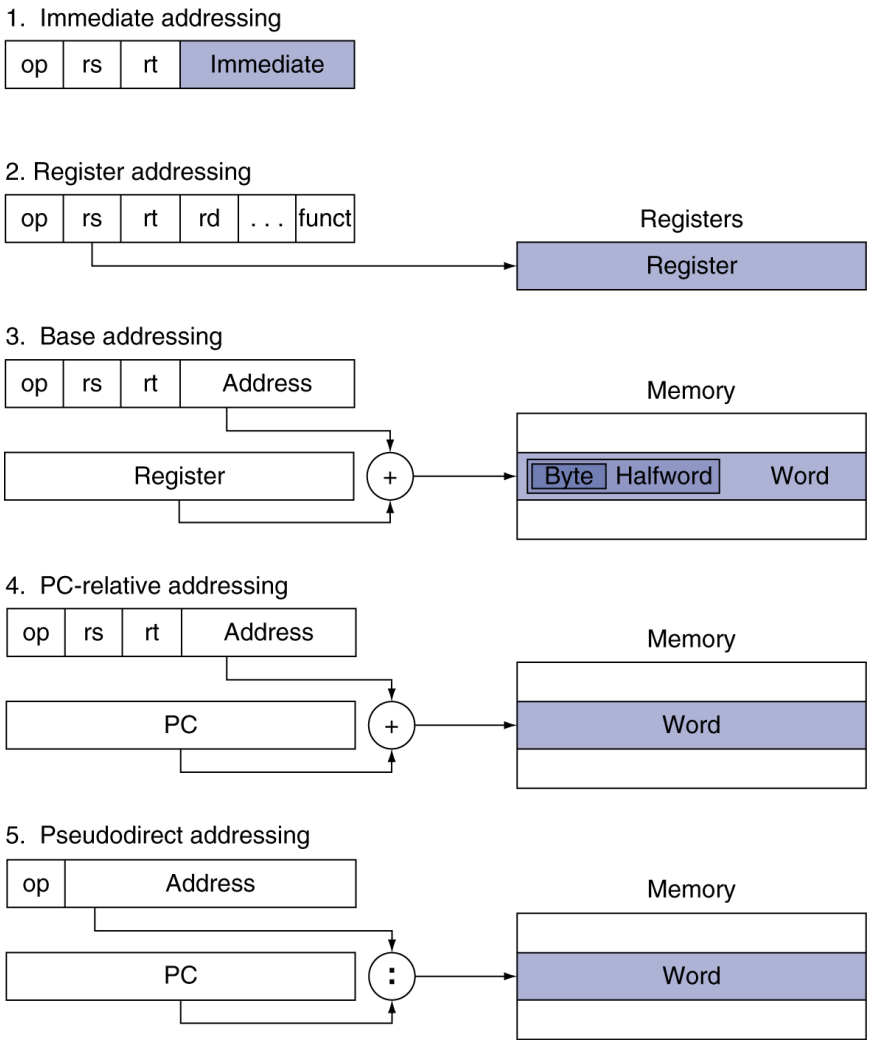
Formato J - esempio


```
j 1048603
```



0000 0000 0100 0000 0000 0000 0110 1100 = 0x0040006c

Addressing modes - riepilogo




```
addi $s0, $s1, 1
```

```
addi $s0, $s1, $s2
```

```
lw $s0, 0($sp)
```

```
beq $s0, $s1, LOOP
```

```
jal reverse
```