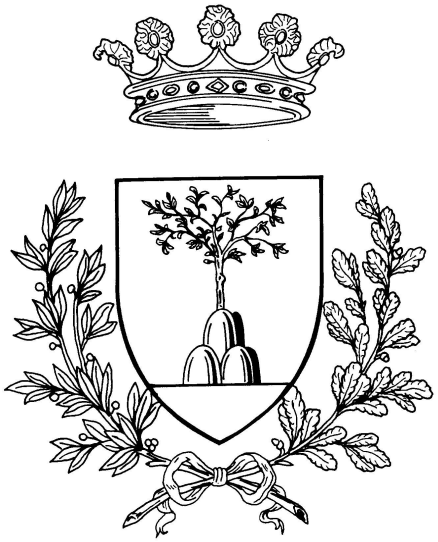


Architettura degli Elaboratori e Laboratorio

Matteo Manzali

Università degli Studi di Ferrara

Anno Accademico 2016 - 2017

Syscall

- Le syscall (chiamate di sistema) sono un insieme di servizi di sistema invocabili attraverso l'istruzione **syscall**.
- Le syscall possono essere usate per operazioni di input/output, terminazione del programma, etc…
- Ogni syscall ha un codice associato.
- Uso delle syscall:
  - si carica il codice della syscall in $v0
  - si caricano gli argomenti (se ci sono) nei registri argomento
  - si esegue l'istruzione syscall
  - si recuperano i risultati (se ci sono) in registri specifici

Syscall

Service	System call code	Arguments	Result
print_int	1	$a0 = integer	
print_float	2	$f12 = float	
print_double	3	$f12 = double	
print_string	4	$a0 = string	
read_int	5		integer (in $v0)
read_float	6		float (in $f0)
read_double	7		double (in $f0)
read_string	8	$a0 = buffer, $a1 = length	
sbrk	9	$a0 = amount	address (in $v0)
exit	10		
print_char	11	$a0 = char	
read_char	12		char (in $a0)
open	13	$a0 = filename (string), $a1 = flags, $a2 = mode	file descriptor (in $a0)
read	14	$a0 = file descriptor, $a1 = buffer, $a2 = length	num chars read (in $a0)
write	15	$a0 = file descriptor, $a1 = buffer, $a2 = length	num chars written (in $a0)
close	16	$a0 = file descriptor	
exit2	17	$a0 = result	

Syscall

- Alcuni esempi di chiamate di sistema in MIPS:
  - per terminare l'esecuzione del programma:

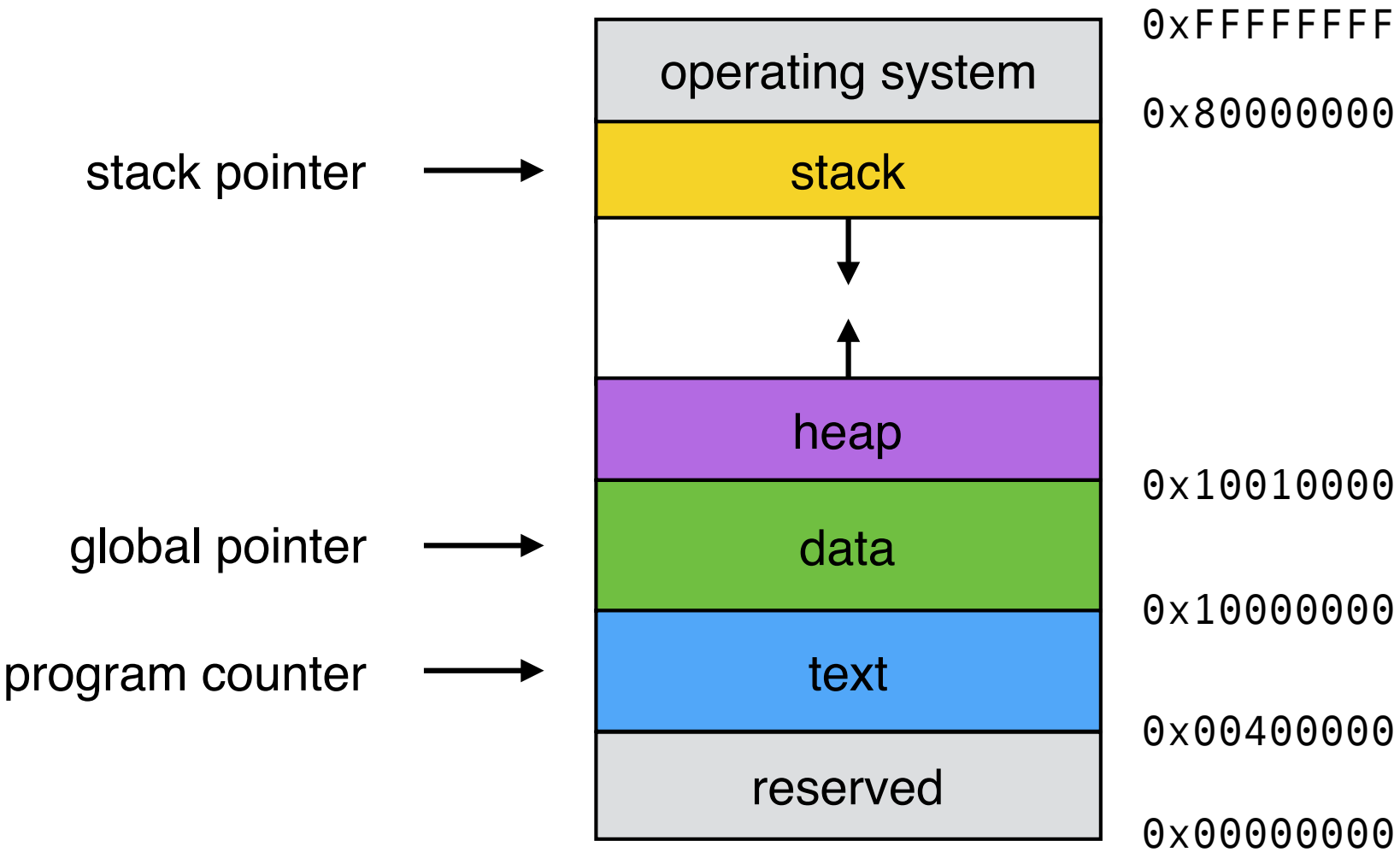
```
ori $v0, $zero, 10    # carico il codice della syscall exit in $v0
syscall               # chiamata di sistema
```

  - per stampare a terminale un intero:

```
ori $v0, $zero, 1     # carico il codice della syscall print_int in $v0
ori $a0, $zero, 45    # carico il numero da stampare in $a0
syscall               # chiamata di sistema
```

- D'ora in avanti useremo la syscall exit per uscire dai programmi.

Mappatura memoria



Mappatura memoria

- **reserved**: area di memoria (4MB) riservata al sistema.
- **text**: area di memoria contenente le istruzioni da eseguire.
  - Contiene le istruzioni identificate dalla direttiva “.text”.
  - E’ di dimensione fissa (massimo numero di istruzioni prefissato).
- **program counter** (PC): registro che contiene l’indirizzo della prossima istruzione da eseguire.
  - Viene aggiornato in automatico all’esecuzione di ogni istruzione.
- **data**: area di memoria contenente solo i dati statici globali (caricati con direttiva .extern).
  - esempio: “.extern LABEL 20”

Mappatura memoria

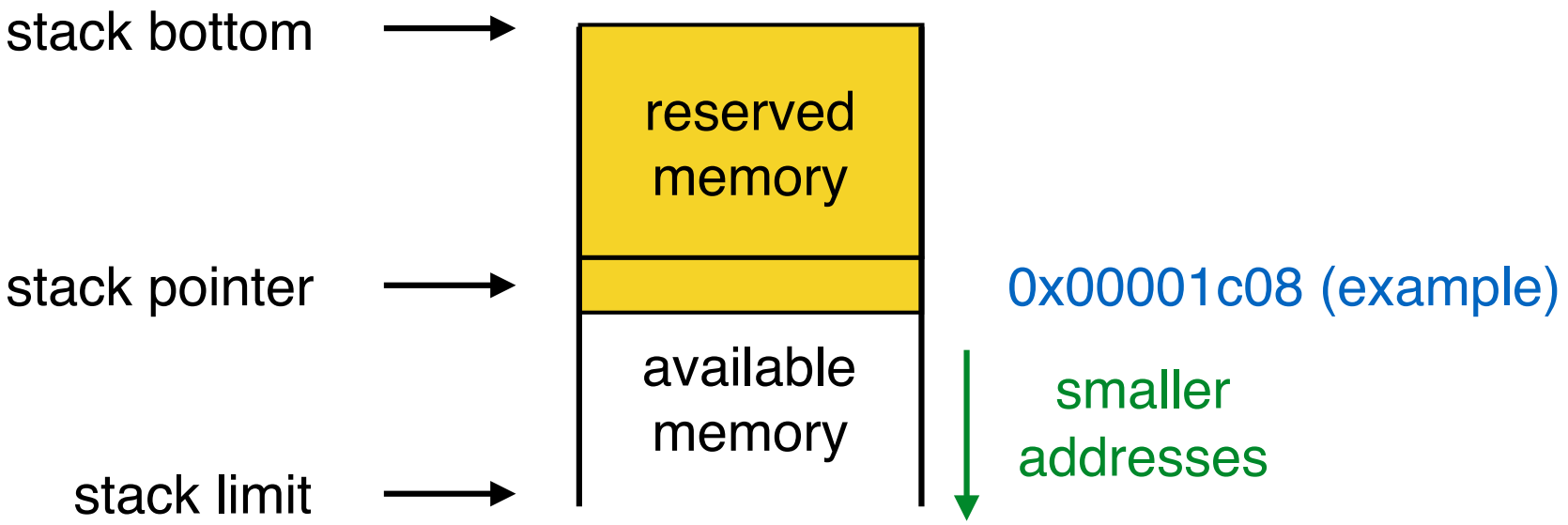
- **global pointer** ($gp): registro usato per accedere tramite un offset alla sezione data (default value 0x10008000).
  - Si usa solo per accedere ai dati globali.
- **heap**: area di memoria contenente:
  - i dati caricati con la direttiva .data
  - i dati allocati dinamicamente
- In MIPS si può allocare dinamicamente della memoria usando una syscall dedicata (prossime slides).
- In C si può allocare dinamicamente della memoria usando *malloc*, in C++ invece si usa *new*.

Stack

- All'inizio dell'esecuzione di un programma viene riservata una sezione contigua della memoria principale chiamata **stack**.
- Lo stack è una coda LIFO (last-in first-out) e cresce verso il basso:
  - da indirizzi di memoria alti verso quelli bassi
- Lo **stack pointer** è un registro (**$sp**) che contiene l'indirizzo di inizio dello stack:
  - gli indirizzi di memoria più grandi di $sp sono riservati
  - gli indirizzi di memoria più piccoli sono disponibili (fino al raggiungimento del limite dello stack)

Stack pointer

- All'inizio del programma lo stack pointer punta all'inizio dello stack (stack bottom).
- Può crescere verso il basso fino al raggiungimento del limite dello stack (se cresce ulteriormente si ha **stack overflow**).



Cosa va nello stack

- Lo stack viene usato per salvare temporaneamente i dati che non possono essere tenuti nei registri:
  - dati necessari a gestire le chiamate a funzione
  - variabili temporanee
- I dati nello stack sono allocati staticamente e sono locali (la loro vita è la durata della funzione in cui sono stati creati).
  - Al contrario l'heap contiene dati allocati dinamicamente e globali (esistono fino a quando non vengono eliminati).
- Programmando ad alto livello non ci si preoccupa di gestire stack e heap.
  - In MIPS invece è necessario farlo in maniera esplicita!

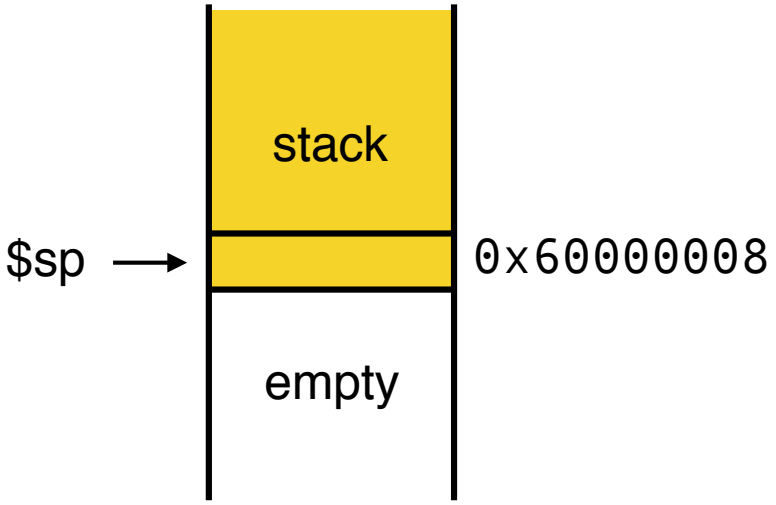
Come usare lo stack

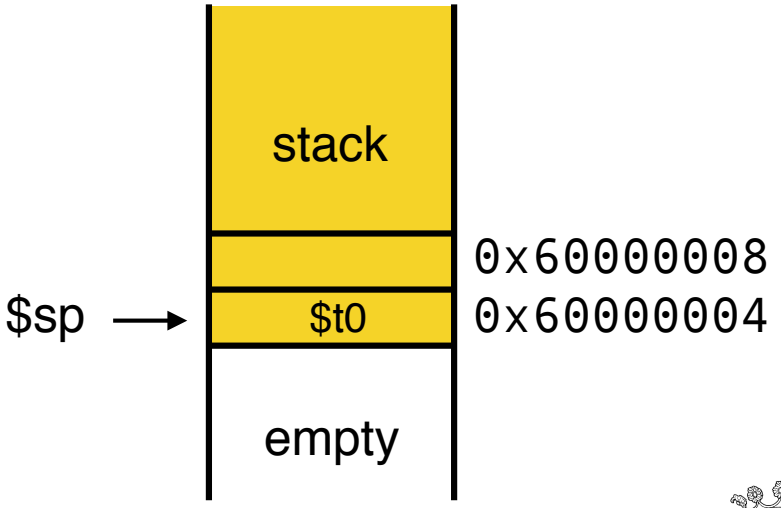
- I dati possono essere inseriti nello stack tramite un'operazione di **push**:
  - si decrementa $sp della dimensione del dato da inserire
  - si scrive in memoria il dato (all'indirizzo puntato da $sp)
- Operazione inversa per rimuovere un dato dallo stack (operazione **pop**):
  - si legge da memoria il dato (all'indirizzo puntato da $sp)
  - si incrementa $sp della dimensione del dato da rimuovere
- Lavorando con registri a 32 bit si decrementerà/incrementerà $sp di 4 (byte) per ogni valore che vogliamo inserire/rimuovere.

Esempio di push

- Esempio: inserisco nello stack il contenuto di $t0.

```
…
addi $sp, $sp, -4
sw $t0, 0($sp)
…
```

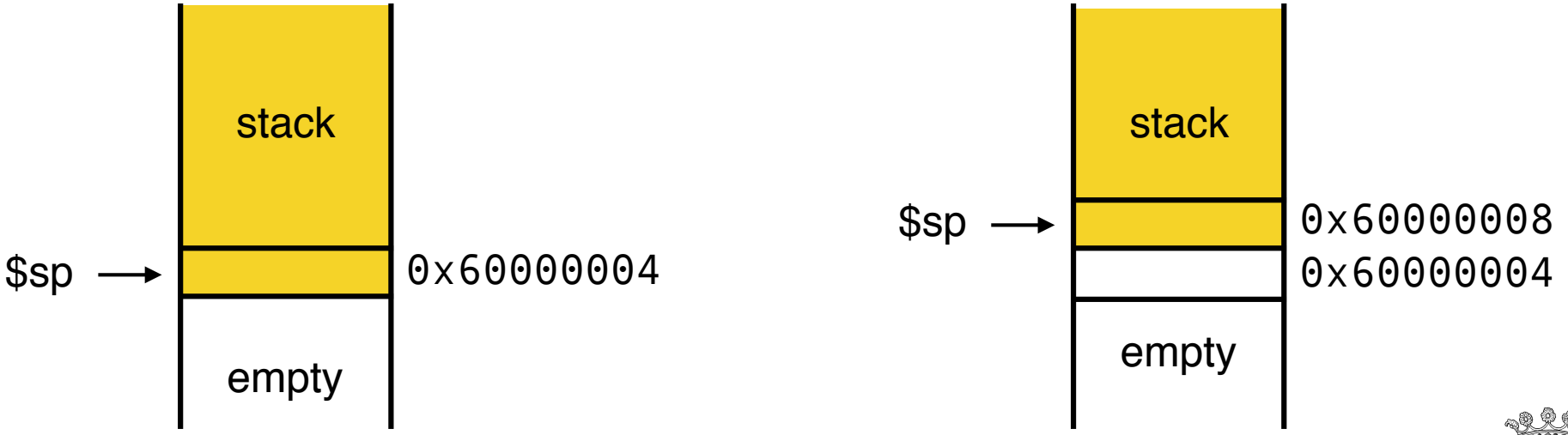




Esempio di pop

- Esempio: rimuovo nello stack la word più recente e la metto in $t1.

```
…
lw $t1, 0($sp)
addi $sp, $sp 4
…
```



Esempio di pop

- Come potete notare rimuovere un elemento dallo stack non significa “cancellarlo” fisicamente:
  - il valore rimane lì ma è considerato **invalido**
  - verrà **sovrascritto** al prossimo push
- Se vi è mai capitato di ritornare da una funzione un puntatore ad una variabile locale, potreste aver notato che:
  - inizialmente il valore sembra rimanere valido
  - dopo qualche istruzione il valore si corrompe (spesso si finisce in segmentation fault)
- Questa è la ragione di quel comportamento apparentemente anomalo!

Esempio di push (2)

- Esempio: inserisco nello stack il contenuto di $t0, $t1 e $t2.

```
…
addi $sp, $sp -12      # decremento $sp di 12 (spazio per 3 word)
sw $t0, 0($sp)         # salvo $t0 nello stack
sw $t1, 4($sp)         # salvo $t1 nello stack
sw $t2, 8($sp)         # salvo $t2 nello stack
…
```

- Notare come effettuo solo una volta il decremento di $sp.
- Avrei potuto fare 3 decrementi da 4 byte ciascuno, ma sarebbe stato meno efficiente.
- Potete inoltre notare l'utilizzo dell'offset nella sw.

Esempio di pop (2)

- Esempio: leggo dallo stack le 3 word più recenti e le scrivo in $t0, $t1 e $t2.

```
…
lw $t0, 0($sp)          # leggo dallo stack e scrivo in $t0
lw $t1, 4($sp)          # leggo dallo stack e scrivo in $t1
lw $t2, 8($sp)          # leggo dallo stack e scrivo in $t2
addi $sp, $sp 12        # incremento $sp di 12 (spazio per 3 word)
…
```

Funzioni

- Raggruppano una serie di istruzioni.
- Possono essere richiamate più volte da diversi punti del programma.
- Sono identificate da:
  - un nome
  - un valore di ritorno
  - zero, uno o più parametri
  - il corpo della funzione (con le istruzioni da eseguire)
- N.B.: Nell'esempio il main è la **funzione chiamante** mentre max è la **funzione chiamata**.

```
int max(int a, int b) { … }
int max(int* arr) { … }

int main () {
    int a = … ;
    int b = … ;
    int const N = … ;
    int arr [N] = { … };

    int m = max(a, b);
    m = max(&arr);
}
```

Funzioni

- I passi necessari per effettuare una chiamata a funzione sono:

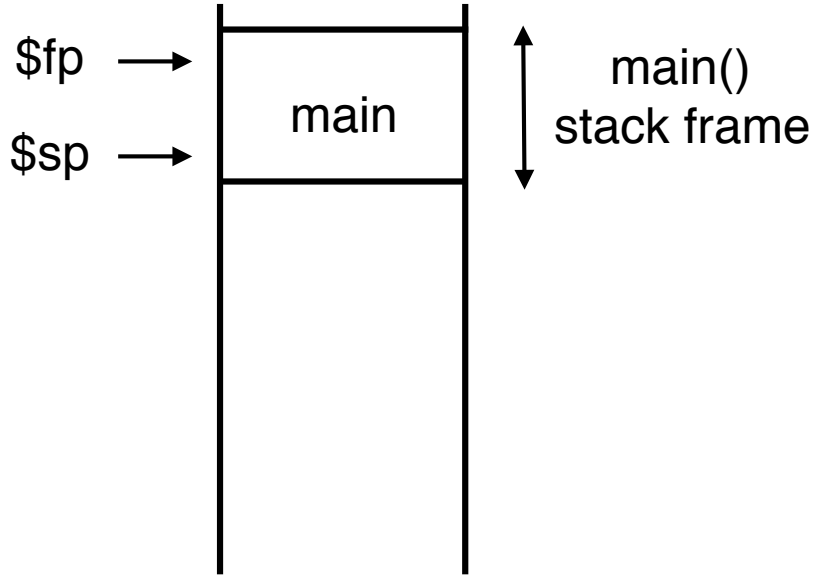
  1. mettere i parametri dove la funzione chiamata li possa accedere
  2. trasferire il controllo alla funzione
  3. acquisire spazio di memoria per l'esecuzione della funzione
  4. eseguire le operazioni della funzione
  5. mettere il risultato dove la funzione chiamante lo possa accedere
  6. ritornare il controllo al chiamante

Stack e funzioni

- Le funzioni vengono implementate attraverso lo stack:
  - ad ogni chiamata di funzione viene riservata una porzione dello stack (**stack frame**)
  - il **frame pointer** ($fp) è un ulteriore registro che contiene l'indirizzo di inizio dello stack frame corrente
    - ha lo stesso valore di $sp all'inizio di una funzione ($sp può venire modificato durante la funzione, per salvare variabili temporanee)
  - una volta che la funzione ritorna, lo stack frame viene rimosso (aggiornando $sp e $fp)
  - Nello stack frame ci sono tutti i dati necessari alla funzione.

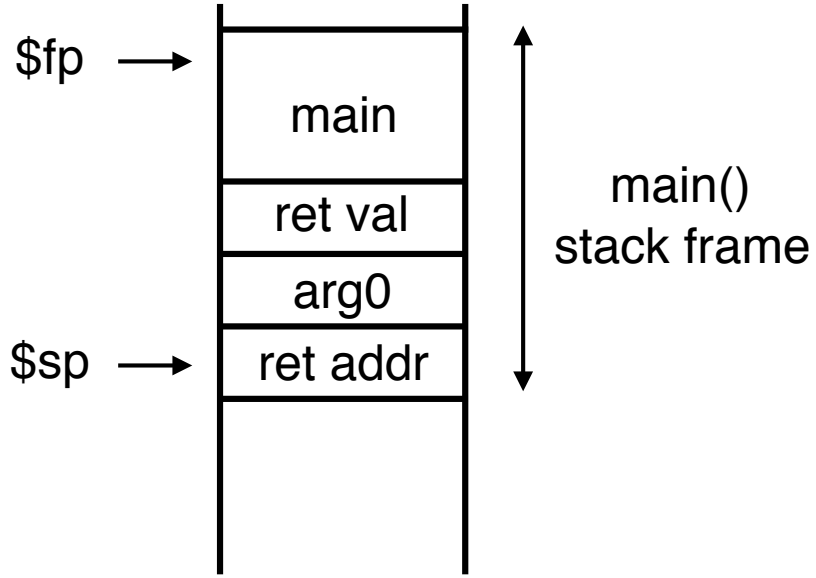
Stack e funzioni - esempio

- Pensiamo ad esempio alla funzione main() di un programma C.
- Ci sarà uno stack frame dedicato al main (con relativo frame pointer).



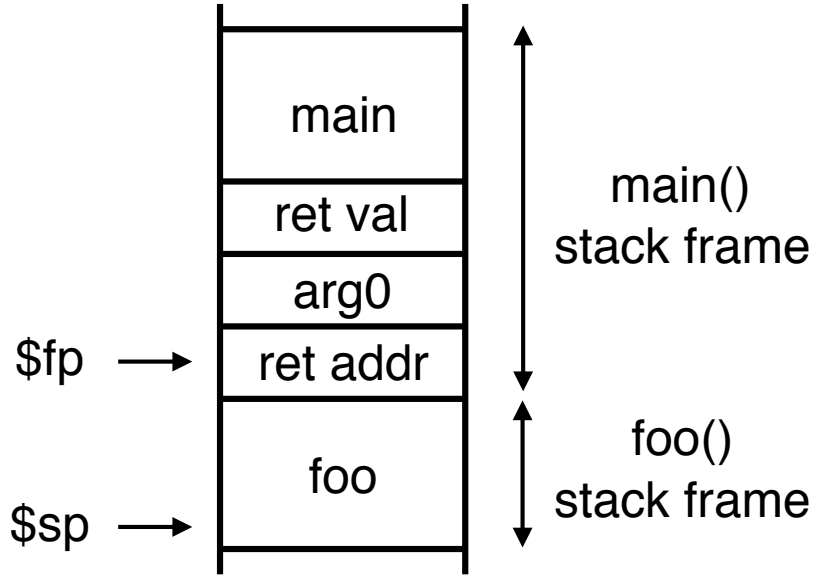
Stack e funzioni - esempio

- Supponiamo che dentro al main ci sia una chiamata ad una funzione foo() che richiede un argomento.
- Nello stack frame di main verrà fatto spazio per il valore di ritorno, gli argomenti (uno in questo caso) e l'indirizzo di ritorno (per sapere a quale istruzione ritornare).



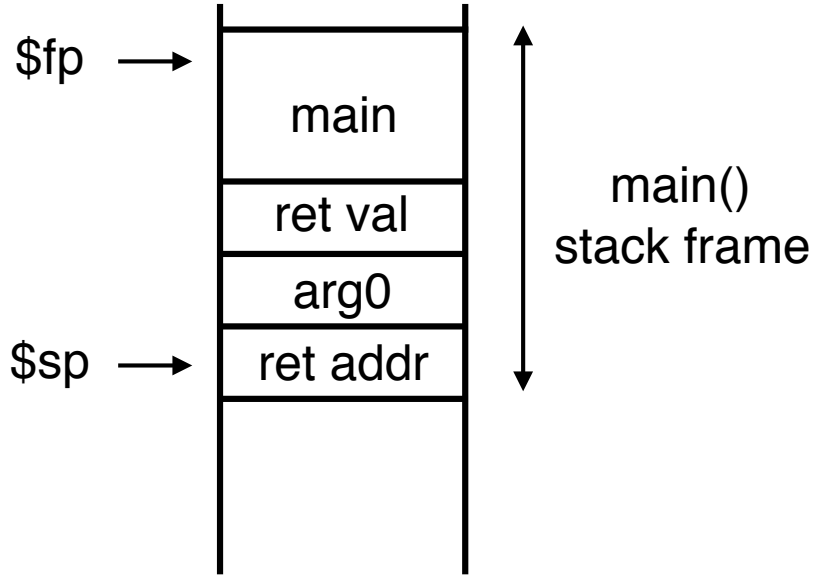
Stack e funzioni - esempio

- Inizia l'esecuzione di foo().
- foo() mette nel suo stack frame alcune variabili locali, spostando $sp.
- Notare lo spostamento del frame pointer $fp che rimane all'inizio dello stack frame corrente.



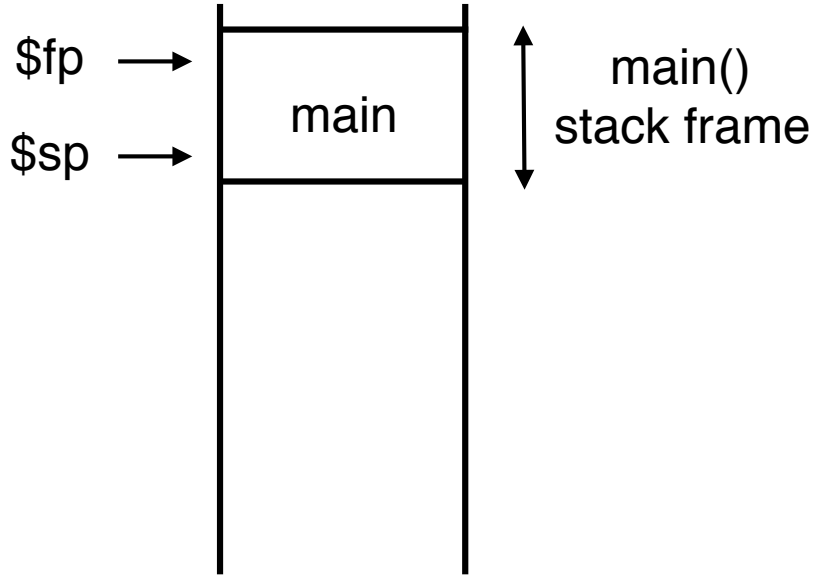
Stack e funzioni - esempio

- Termina l'esecuzione di foo().
- Viene rimosso lo stack frame di foo().
- Lo stack pointer assume il valore del frame pointer e il frame pointer riprende il suo vecchio valore.



Stack e funzioni - esempio

- Nel main viene letto il valore ritornato dalla funzione.
- Ora si possono rimuovere dallo stack frame l'indirizzo di ritorno, il valore di ritorno ed i parametri di foo().



Stack e funzioni in MIPS

- Il procedimento descritto è quello generalmente adottato (con diverse varianti da architettura ad architettura) per implementare le chiamate a funzione.
- In MIPS si è cercato di ridurre la quantità di accessi allo stack necessari all'implementazione delle chiamate a funzione.
  - lo stack è in RAM → tempi di accesso molto alti
- Per questo motivo vengono usati i registri per passare la maggior parte di informazioni tra la funzione chiamante e la funzione chiamata.
  - sono stati creati registri appositamente per questo scopo
  - non è però sempre possibile evitare l'uso del stack (chiamate ricorsive, etc…)

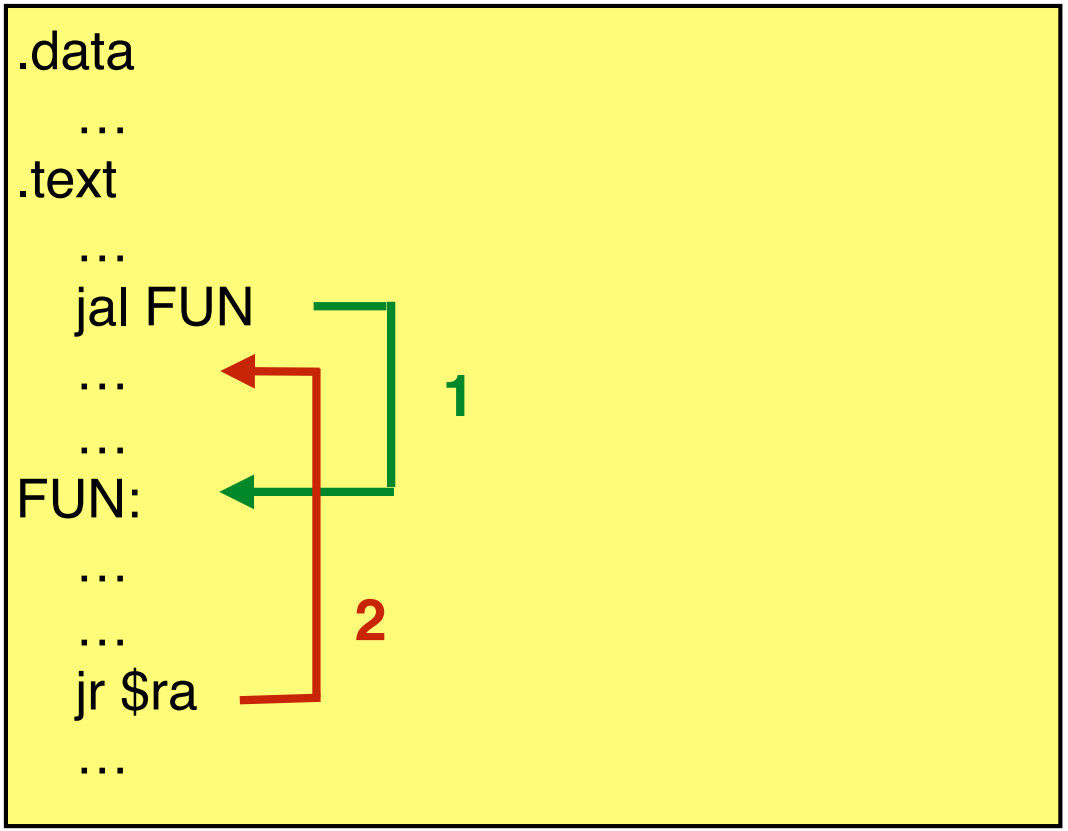
Chiamata a funzione

- Per chiamare una funzione è necessario prima di tutto identificarla:
  - si utilizzano le etichette
- E' necessario inoltre ricordarsi l'indirizzo dell'istruzione successiva del chiamante.
- Si utilizza l'istruzione "jump and link":
  - **jal label**
  - salta all'etichetta "label" che definisce l'inizio della funzione
  - salva nel registro $ra (return address) l'indirizzo dell'istruzione successiva del chiamante

Ritorno da funzione

- Per tornare alla funzione è necessario effettuare un jump all'indirizzo contenuto nell'apposito registro.
- Si utilizza la funzione "jump register":
  - **jr $ra**
  - effettua un jump all'indirizzo contenuto nel registro passato come parametro
  - utilizzo $ra perchè è il registro che contiene l'indirizzo di ritorno (inserito dall'istruzione jal)

Esempio



Argomenti e valore di ritorno

- Per il passaggio dei parametri (argomenti di funzione) si utilizzano dei registri dedicati:
  - i registri $a0, $a1, $a2 e $a3
  - per più di 4 parametri si utilizza lo stack
- Per il valore di ritorno esistono due registri:
  - $v0 e $v1
- L'ordine di utilizzo dei registri è incrementale:
  - 1 parametro → $a0
  - 2 parametri → $a0, $a1
  - etc…

Esempio

```
.data

.text
    addi $a0, $zero, 5          # a0 = 5
    ori $a1, $zero, 3           # a1 = 3
    jal FUN                     # v0 = a0 + a1 • 2
    or $s0, $v0, $zero          # s0 = v0
    addi $v0, $zero, 10         # carico exit in $v0
    syscall                     # syscall exit

FUN:
    ori $t0, $zero, 2           # t0 = 2
    mult $a1, $t0               # t0 = a1 • t0
    mflo $t0
    add $v0, $a0, $t0           # v0 = a0 + t0
    jr $ra
```

Preservare i registri

- Come già detto in precedenza i registri hanno scopi ben definiti.
- Seguendo le convenzioni MIPS i valori di alcuni registri devono essere “preservati” dalla funzione chiamata.

Name	Register Number	Usage	Should preserve on call?
$zero	0	the constant 0	no
$v0 - $v1	2-3	returned values	no
$a0 - $a3	4-7	arguments	no
$t0 - $t7	8-15	temporaries	no
$s0 - $s7	16-23	saved values	yes
$t8 - $t9	24-25	temporaries	no
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	return address	yes

Regole per il chiamante

- Secondo la convenzione MIPS i registri che una funzione deve salvare nello stack prima di fare una chiamata a funzione sono:
  - i registri $v0 e $v1
  - i registri argomento ($a0 - $a3)
  - i registri temporanei ($t0 - $t9)
- **La funzione chiamata può sovrascrivere tali registri senza l'obbligo di salvaguardarne il vecchio valore.**
- Tuttavia è buona norma non fare affidamento sul contenuto dei registri temporanei dopo aver fatto una chiamata a funzione:
  - meglio utilizzare i registri permanenti ($s0 - $s7)
  - stesso discorso per i registri $v0 e $v1

Regole per il chiamato

- Secondo la convenzione MIPS i registri che una funzione deve salvare nello stack come prima operazione (e ricaricarne il vecchio valore prima di uscire) sono:
  - i registri permanenti ($s0 - $s7)
  - il global pointer ($gp)
  - il frame pointer ($fp)
  - il return address ($ra)
- **Questi registri vanno preservati solo se è previsto che il loro valore cambi dentro la funzione.**

Regole per il chiamato

- Difficilmente userete il global pointer:
  - anche nel caso di utilizzo di variabili globali non siete tenuti ad usarlo se non volete
- Il frame pointer può essere utile ma non indispensabile:
  - potete sempre accedere ai dati dello stack frame con un offset a partire dallo stack pointer
  - nel caso vogliate utilizzare il frame pointer dovete gestirlo voi (non viene settato in automatico quando chiamate una funzione)

Esempio - inc()

- Vediamo le regole appena descritte attraverso una serie di esempi pratici.
- Prendiamo una funzione “inc()” che dato un intero come parametro ritorna lo stesso intero incrementato di una unità.

```
void main() {
    int a = 2;
    a = inc(a);
    // a = 3
}

int inc (int a) {
    return a + 1;
}
```

```
.text
    ori $a0, $zero, 2          # a = 2
    jal INC
    or $a0, $v0, $zero         # a = 3
    addi $v0, $zero, 10        # carico exit in $v0
    syscall                    # syscall exit

INC:
    addi $v0, $a0, 1           # a = a + 1
    jr $ra                     # return
```

Esempio - inc()

- Ora dentro ad inc() vogliamo salvare in $s0 il parametro passato:
  - da convenzione è necessario preservare il suo vecchio valore

```
.text
    ori $a0, $zero, 2          # a = 2
    jal INC
    or $a0, $v0, $zero          # a = 3
    addi $v0, $zero, 10        # carico exit in $v0
    syscall                    # syscall exit
INC:
    addi $sp, $sp, -4          # alloco stack frame
    sw $s0, 0($sp)             # inserisco $s0 nello stack
    ori $s0, $a0, 0            # uso $s0 (cambia valore)
    addi $v0, $s0, 1           # a = a + 1
    lw $s0, 0($sp)             # ricarico vecchio valore $s0
    addi $sp, $sp, 4           # elimino lo stack frame
    jr $ra                     # return
```

Esempio - swap()

- Prendiamo una funzione “swap()” che dati due puntatori scambia i valori puntati.

```
void swap(int* a, int* b) {
    int t = *a;
    *a = *b;
    *b = t;
}
```

```
SWAP:
    lw $t0, 0($a0)          # carico *a in t0
    lw $t1, 0($a1)          # carico *b in t1
    sw $t0, 0($a1)          # scrivo t0 in *b
    sw $t1, 0($a0)          # scrivo t1 in *a
    jr $ra                  # return
```

- Notare come non ho bisogno di allocare lo stack frame in SWAP visto che non ho registri da preservare.

Esempio - swap()

- Ora mi impongo di usare i registri permanenti in SWAP invece dei registri temporanei.
- In questo caso devo preservare i loro vecchi valori.

```
SWAP:
    addi $sp, $sp, -8       # alloco un frame per 2 registri
    sw $s0, 0($sp)          # salvo nello stack frame il valore di s0
    sw $s1, 4($sp)          # salvo nello stack frame il valore di s1
    lw $s0, 0($a0)          # carico *a in s0
    lw $s1, 0($a1)          # carico *b in s1
    sw $s0, 0($a1)          # scrivo s0 in *b
    sw $s1, 0($a0)          # scrivo s1 in *a
    lw $s0, 0($sp)          # ricarico vecchio valore $s0
    lw $s1, 4($sp)          # ricarico vecchio valore $s1
    addi $sp, $sp, 8        # rimuovo il frame dallo stack
    jr $ra                  # return
```


Esempio - inc_arr()

- La funzione “inc_arr()” scorre un array di interi e incrementa di 1 il valore di ogni elemento richiamando la funzione inc() vista nelle slides precedenti.
- La funzione accetta come parametri il puntatore all’array e la dimensione dell’array.
- In questo caso abbiamo una funzione che chiama un’altra funzione:
  - dobbiamo preservare sicuramente il registro $ra (viene sovrascritto con la chiamata jal)
  - dobbiamo anche preservare il registro $a0 in quanto deve essere modificato in inc_arr() per passare il parametro a inc()

Esempio - inc_arr()

- Una possibile versione in C:

```
void inc_arr(int* a, int n) {
    int i = 0;
    for (; i < n; ++i) {
      a[i] = inc(a[i]);
    }
}

int inc (int a) {
    return a + 1;
}
```

Esempio - inc_arr()

```
INC_ARR:                    # etichetta della funzione

    addi $sp, $sp, -8       # creo un stack frame per 2 registri
    sw $a0, 0($sp)          # salvo a0
    sw $ra, 4($sp)          # salvo il return address

    or $s0, $a0, $zero      # s0 = a0 (puntatore a inizio array)
    or $t0, $t0, $zero      # t0 = 0 (è il contatore)

FOR_B:

    beq $t0, $a1, FOR_E # se i == N allora salto a fine ciclo
    sll $t1, $t0, 2         # t1 = t0 * 4 (offset)
    add $t1, $t1, $s0       # t1 = t1 + &arr[0]
    …
```

Esempio - inc_arr()

```
    …
    lw $a0, 0($t1)          # a0 = array[i]
    jal INC                 # chiamo inc()
    sw $v0, 0($t1)          # array[i] = inc(array[i])

    addi $t0, $t0, 1        # t0++ (incremento il contatore)
    j FOR_B                 # salto a inizio ciclo

FOR_E:                      # il ciclo è finito

    lw $a0, 0($sp)          # carico il vecchio valore di a0
    lw $ra, 4($sp)          # carico il vecchio valore di ra
    addi $sp, $sp, 8        # elimino lo stack frame
    jr $ra                  # ritorno al main
```

Memorandum su funzioni

- Quando compilate un programma in C è il compilatore che decide (in base all'architettura ed in base a come il compilatore stesso è stato istruito) come implementare le chiamate a funzione:
  - se mettere tutto sullo stack
  - o massimizzare l'uso dei registri
  - quali registri utilizzare
  - quali valori locali salvaguardare
- Quando scrivete in codice MIPS questa responsabilità spetta a voi.
- Prima di scrivere del codice cercate di capire quali risorse (registri) vi servono e quali tra quei registri devono o meno essere salvaguardati in caso di chiamata a funzione (dalla parte del chiamante e dalla parte del chiamato).

Istruzioni floating point

- In MIPS l'hardware che gestisce le operazioni floating point (FP) è un **coprocessore separato** (chiamato coprocessore 1).
- Questo coprocessore ha i propri registri FP:
  - 32 registri in singola precisione $f0, $f1, …, $f31
  - possono essere accoppiati per ottenere 16 registri in doppia precisione: $f0/$f1, $f2/$f3, …, $f30/$f31
- Le istruzioni per eseguire operazioni FP operano solo su questi registri.

Load / store floating point

- Per fare operazioni di load / store ci sono istruzioni dedicate
  - **lwc1** → load word coprocessor 1
  - **swc1** → store word coprocessor 1
  - **ldc1** → load double coprocessor 1
  - **sdc1** → store double coprocessor 1
- Es.:
  - swc1 $f1, 0($sp) → scrive sullo stack il numero FP in singola precisone contenuto nel registro $f1
  - ldc1 $f8, 0($sp) → carica dallo stack frame il numero FP in doppia precisione (double, 8 byte) e lo scrive nella coppia di registri $f8/$f9 (essendo double un solo registro non basta)

Aritmetica floating point

- Le operazioni aritmetiche sono simili a quelle sui numeri interi.
- Aritmetica in singola precisione:
  - **add.s, sub.s, mul.s, div.s**
  - esempio: add.s $f0, $f1, $f6
- Aritmetica in doppia precisione:
  - **add.d, sub.d, mul.d, div.d**
  - esempio: mul.d $f4, $f4, $f6

Confronto floating point

- Esistono delle istruzioni di confronto per numeri FP che settano un bit speciale a 1 se la condizione è vera, altrimenti 0:
  - singola precisione: **c.XXX.s**
  - doppia precisione: **c.XXX.d**
  - dove XXX è uno tra i seguenti controlli:
    - eq (equal), ne (not equal), lt (less than), le (less than or equal), gt (greater than), ge (greater than or equal)
  - esempio:

    c.lt.s $f3, $f4 → setta il bit di controllo a 1 se il numero FP singola precisione contenuto in $f3 è minore rispetto al numero FP singola precisione contenuto in $f4.

Confronto floating point

- Una volta eseguita l'istruzione di controllo, è possibile effettuare un salto condizionato in base al valore del bit di controllo:
  - **bc1t** → salta se il bit di controllo è a 1
  - **bc1f** → salta se il bit di controllo è a 0
- Esempio:

  bc1t LABEL → se il bit di controllo è a 1 allora salta all'etichetta

Trasferimento FP / interi

- Esistono istruzioni per copiare i valori tra registri interi e FP e viceversa:
  - **mtc1** → copia da intero a FP singola precisione
  - **mtc1.d** → copia da intero a FP doppia precisione
  - **mfc1** → copia da FP singola precisione ad intero
  - **mfc1.d** → copia da FP doppia precisione ad intero
  - esempio: mtc1 $zero, $f1 → scrive 0 nel registro $f1
  - esempio: mfc1 $t0, $f8 → copia il contenuto di $f8 in $t0
- Il primo registro è sempre intero ed il secondo è sempre FP.
- Queste istruzioni non fanno conversione, solo copia bit-a-bit.

Conversione FP / interi

- Per convertire un numero FP in intero e viceversa:
  - **cvt.TO.FROM fd, fs**
  - dove TO e FROM possono valere:
    - s → FP singola precisione
    - d → FP doppia precisione
    - w → intero (word)
  - fd e fs sono rispettivamente i registri FP di destinazione e sorgente.
  - esempio: cvt.d.w $f0, $f2 → converte l'intero contenuto in $f2 in un FP doppia precisione dentro a $f0.

Istruzioni binarie

- Nelle prossime slides vedremo come vengono rappresentate le istruzioni MIPS in linguaggio macchina (codice binario).
- Tutte le istruzioni MIPS sono rappresentate da una sequenza binaria di 32 bit e le istruzioni si suddividono in 3 grandi gruppi:
  - **formato R** → istruzioni aritmetico/logiche con tre registri o di shift
  - **formato I** → istruzioni aritmetico/logiche con una costante, load/store, branch condizionato
  - **formato J** → istruzioni di branch incondizionato
- Ogni formato segue regole ben precise che permettono di determinare in maniera univoca tutti i campi.

Formato R

- Campi:
  - op: codice dell'operazione (opcode) → sempre a 0 per R
  - rs: numero del primo registro sorgente
  - rt: numero del secondo registro sorgente
  - rd: numero del registro destinazione
  - shamt: numero di bit dello shift (solo per operazione di shift)
  - funct: codice della funzione (estende l'opcode)

Formato R

op	rs	rt	rd	shamt	funct
6 bit	5 bit	5 bit	5 bit	5 bit	6 bit

```
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

op	rs	rt	rd	shamt	funct
6 bit	5 bit	5 bit	5 bit	5 bit	6 bit

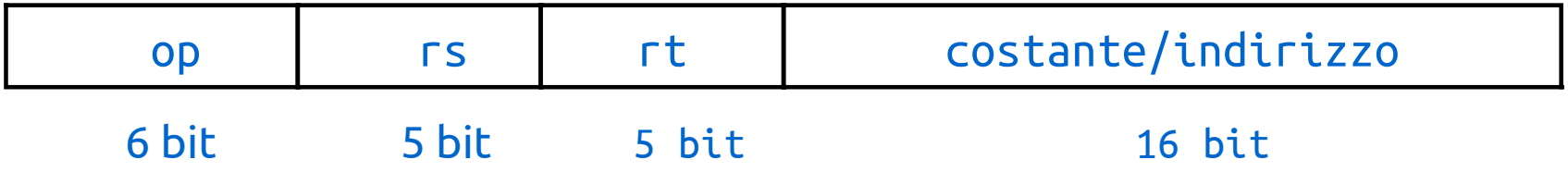
```
sll $t0, $s0, 2
```

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 o operando immediate (16 bit):
    - unsigned → $[0, 2^{16}-1]$
    - signed → $[-2^{15}, +2^{15}-1]$

Formato I

op	rs	rt	costante/indirizzo
6 bit	5 bit	5 bit	16 bit

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

8	18	17	4
001000	10010	10001	0000000000000100

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

Formato I

op	rs	rt	costante/indirizzo
6 bit	5 bit	5 bit	16 bit

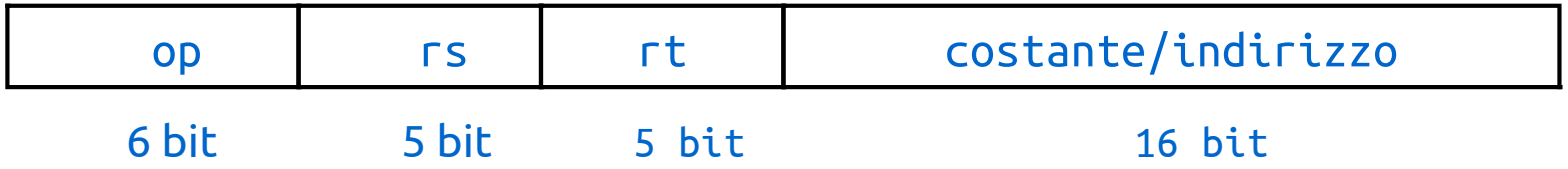
```
lw $s1, 4($s2)
```

35	18	17	4

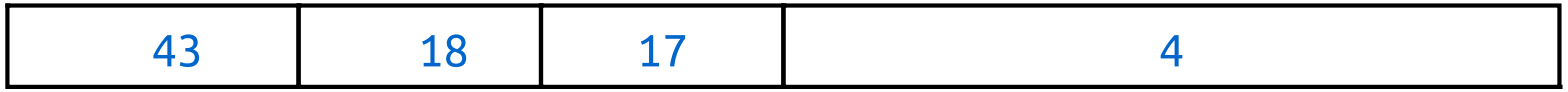
100011	10010	10001	0000000000000100

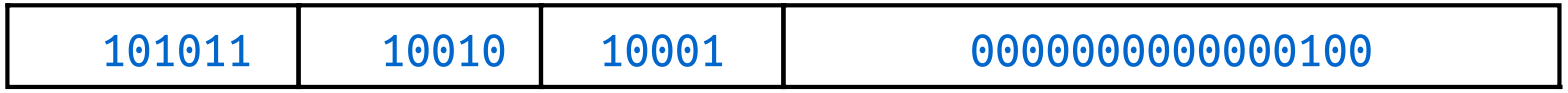
```
1000 1110 0101 0001 0000 0000 0000 0100 = 0x8E510004
```

Formato I



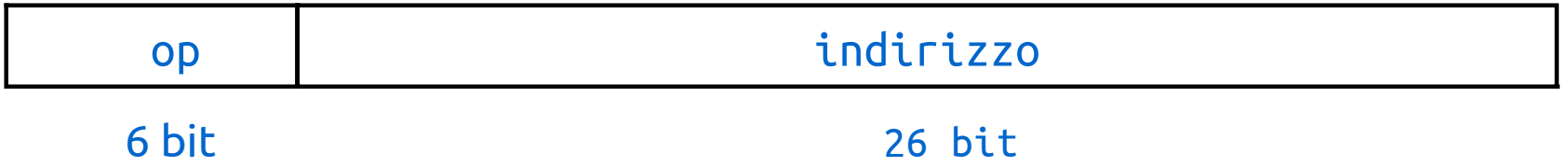

```
sw $s1, 4($s2)
```





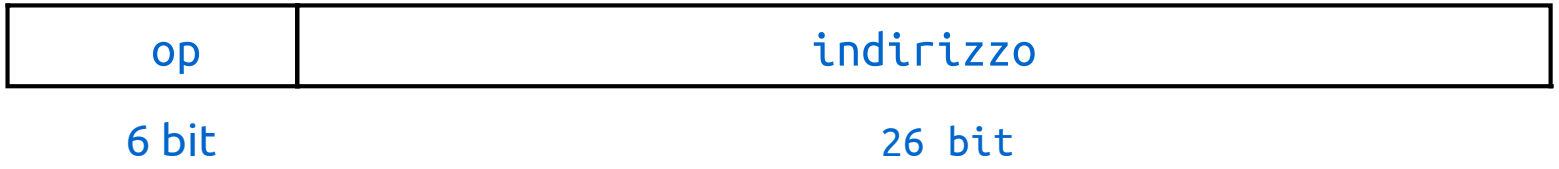

```
1010 1110 0101 0001 0000 0000 0000 0100 = 0xAE510004
```

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 non considerarli
    - assumo che i 4 bit più significativi non cambino rispetto all'indirizzo dell'istruzione corrente

Formato J



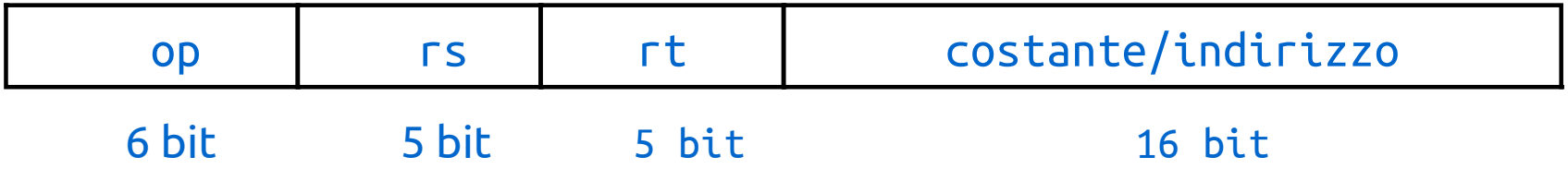

```
j 1048603 (0x10001B)
```


```
0000 1000 0001 0000 0000 0000 0001 1011 = 0x0810001B
```

indirizzo destinazione*: 0000 0000 0100 0000 0000 0000 0110 1100 = 0x000040006C

* supponendo i quattro bit “alti” del PC siano 0

Formato I - beq e bne



- Le istruzioni di branch condizionato (beq, bne) appartengono al formato I e non al formato J.
- Sappiamo che un indirizzo è formato da 32 bit mentre nel formato I abbiamo a disposizione 16 bit.
- Questo ci costringe ad usare quei 16 bit come **offset da applicare al Program Counter** (PC):
  - possiamo shiftare l'indirizzo di 2 come nel caso del formato J
  - un branch condizionato potrà quindi saltare a qualunque istruzione nell'intervallo [PC - 2^{17}, PC + 2^{17} - 4]