Esercitazione di Assembly MIPS32 - Introduzione

1. I registri e convenzioni sui registri.

Il MIPS contiene 32 registri general-porpouse a 32bit per operazioni su interi (**$0..$31**), 32 registri general-porpouse per operazioni in virgola mobile a 32bit (**$FP0**..**$FP31**) più un certo numero di registri speciali, sempre a 32bit, usati per compiti particolari.
Tra questi ultimi, il **Program Counter** (**PC**) contiene l'indirizzo dell'istruzione da eseguire, **HI** e **LO** usati nella moltiplicazione e nella divisione e **Status** contiene i flag di stato della CPU.
Il MIPS è una CPU **RISC** ad architettura **load-store** che sfrutta massicciamente il **pipelining**. Ciò significa che il set di istruzioni disponibili nativamente sulla CPU è estremamente ridotto (Reduced Istruction Set Cpu) e limitato ad istruzioni che sfruttano pienamente la struttura a fasi successive (*pipeline*); il caricamento di una costante in un registro, ad esempio, è ottenuto con una sequenza equivalente di operazioni aritmetiche. Per analoghi motivi le operazioni da e verso la memoria sono limitate al solo caricamento e salvataggio dei registri (*load-store*). Non è possibile ad esempio sommare direttamente un dato contenuto in memoria con un registro: occorre prima spostare il dato in memoria in un registro della CPU.
Allo scopo di semplificare la programmazione in assembly MIPS sono state adottate alcune convenzioni sull'uso dei registri. Queste convenzioni permettono di creare delle *pseudo-istruzioni* facilmente traducibili nel set base del MIPS, di definire delle interfacce standard per le chiamate a sotto-procedure e di definire l'uso dei registri nelle chiamate stesse in modo da evitare corruzioni involontarie nello stato dei registri.
Le più importanti convenzioni sui registri sono:

- il registro **$0 ($zero)** settato costantemente al valore **0**
- il registro **$1 ($at)** che viene usato come variabile temporanea nell'implementazione delle pseudo-istruzioni.

Le convenzioni adottate per i registri interi del MIPS sono riassunti di seguito:

Nome	Numero	Utilizzo	Preservato durante le chiamate
$zero	0	costante zero	*Riservato MIPS*
$at	1	riservato per l'assemblatore	*Riservato Compiler*
$v0-$v1	2-3	valori di ritorno di una procedura	No
$a0-$a3	4-7	argomenti di una procedura	No
$t0-$t7	8-15	registri temporanei (non salvati)	No
$s0-$s7	16-23	registri salvati	Si
$t8-$t9	24-25	registri temporanei (non salvati)	No
$k0-$k1	26-27	gestione delle eccezioni	*Riservati OS*
$gp	28	puntatore alla global area (dati)	Si
$sp	29	stack pointer	Si
$s8	30	registro salvato (fp)	Si
$ra	31	indirizzo di ritorno	No

2. Le pseudo istruzioni

Le pseudo-istruzioni sono istruzioni assembly non implementate direttamente dalla CPU ma ottenute come macro di istruzioni realmente implementate. Il compilatore, all'atto della compilazione, sostituisce le pseudo-istruzioni con il codice delle macro corrispondenti. Ad esempio, la pseudo-istruzione load con indirizzamento immediato:

li $v0 , 4

che carica la costante 4 in $v0 è realizzata attraverso l'istruzione:

ori $vo, $zero, 4

che realizza un *or logico* con indirizzamento immediato tra il registro **$0** (che vale sempre **0**, elemento neutro per l'*or*) e la costante **4** e mette il risultato in **$v0** (altre codifiche sono possibili).
Pseudo-istruzioni più complicate possono fare uso del registro **$at** riservato per questo scopo. Ad esempio, la pseudo-istruzione load con indirizzamento immediato di una word:

la $v0 , 0x12345678

che carica la costante 0x12345678 in $v0 è realizzata attraverso la coppia di istruzioni:

lui $at, 0x1234
ori $vo, $at, 0x5678

la prima istruzione carica in $at la parte alta della word, mentre la seconda completa la word e la salva nel registro appropriato (altre codifiche sono possibili).

Di seguito sono indicate le più usate pseudo-istruzioni con le relative codifiche adottate dal compilatore **Pcspim**:

li $rx , half-word	**ori $rx, $zero, half-word**
la $rx, word	**lui $at,word-hi** **ori $rx,$at,word-lo**
move $rx, $ry	**addu $rx, $zero, $ry**
blt $rx, $ry, Loop	**slt $at, $rx, $ry** **bne $at, $zero, Loop**
ble $rx, $ry, Loop	**slt $at, $ry, $rx** **beq $at, $zero, Loop**
bgt $rx, $ry, Loop	**slt $at, $ry, $rx** **bne $at, $zero, Loop**
bge $rx, $ry, Loop	**slt $at, $rx, $ry** **beq $at, $zero, Loop**

Esistono anche pseudo-istruzioni che permettono indirizzamenti più complessi del semplice indirizzamento con base implementato sul MIPS (vedi manuale SPIM).

3. Convenzioni di chiamata e salvataggio per le chiamate a procedura

La convenzione per le chiamate a procedura prevede che i parametri in ingresso alla procedura debbano essere collocati nei registri $a0-$a3. Nel caso occorra passare più di quattro word i dati aggiuntivi devono essere passati attraverso lo stack. La procedura, se necessario, restituirà in uscita al chiamante al più due word attraverso i registri $v0-$v1. Se occorre comunicare più dati, i dati aggiuntivi devono essere passati tramite lo stack. Nel caso i dati da passare siano *tanti* (ad esempio un array o una stringa) è preferibile comunicare i dati *per riferimento* ossia comunicare alla procedura non i dati veri e propri ma un indirizzo della memoria dove i dati possono essere recuperati.

Una procedura deve garantire che, al termine della procedura, i registri **$s0-$s7** , **$gp**, **$sp**, **$fp** siano ripristinati al valore che avevano al momento della chiamata. Per chiamare una procedura occorre usare l'istruzione **jal** che memorizza in **$ra** il valore corretto di ritorno (eventuali valori precedenti di $ra devono essere preservati prima della chiamata se necessario). Prima di iniziare la procedura avra cura di preservare nello stack i registri indicati sopra che verranno alterati.Al termine della procedura, dopo aver ripristinato i registri, per tornare al punto della chiamata occorre eseguire una istruzione **jr $ra**.

Es: Si implementi una funzione che realizza la somma di due interi (file: somma_proc.asm):

```
#Sum(A,B)
sum:    add $v0, $a0, $a1    #ricevo A in $a0, B in $a1
        jr $ra               #restituisco la somma in $v0

#Sum_scorretta(A,B)
sum:    move $s0,$a1         #altero $s0 !!non rispetta le convenzioni!!
        add $v0, $a0, $s1
        jr $ra

#Sum_corretta(A,B)
sum:    move $t0, $s0        #preservo $s0 in $t0 ($t0 lo posso cambiare)
        move $s0,$a1         #altero $s0
        add $v0, $a0, $s1
        move $s0,$t0         #rimetto a posto $s0, le convenzioni sono rispettate!
        jr $ra
```

4. La struttura della memoria: area testo, area dati, area stack

Di seguito è riportata la suddivisione della memoria usata nel MIPS32. Lo Stack e lo Heap condividono l'area dati lasciata libera dai dati statici.

<table>
<tr><td>0x9000.0000-</td><td colspan="3">Kernel data</td></tr>
<tr><td>0x8000.0000-0x8fff.ffff</td><td colspan="3">Kernel</td></tr>
<tr><td rowspan="3">0x1000.0000-0x7fff.ffff</td><td rowspan="3">Area Dati</td><td colspan="2">Stack ⇓</td></tr>
<tr><td colspan="2">Heap ⇑</td></tr>
<tr><td rowspan="2">Dati statici</td><td>....</td></tr>
<tr><td>0x1000.FFFF
0x1000.0000</td></tr>
<tr><td>0x0040.0000-0x0fff.ffff</td><td colspan="3">Area Testo</td></tr>
<tr><td>0x0000.0000-0x003f.ffff</td><td colspan="3">Riservata</td></tr>
</table>

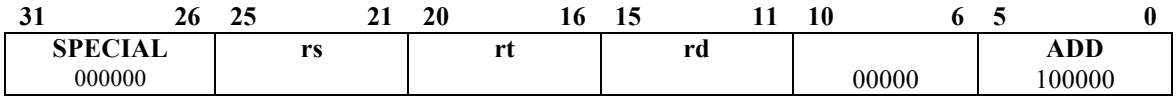
I primi 64K byte dell'area dati possono essere indirizzati agevolmente attraverso il registro **$gp** che è inizializzato con il valore centrale di questa sottoarea, solitamente **0x1000.8000**. In questo modo con una sola istruzione di load/strore è possibile trasferire dati tra registri e memoria dati.
Lo stack è implementato per convenzione alla fine dell'area dati e viene incrementato all'indietro verso indirizzi di memoria via via decrescenti. E' possibile inserire dati nello stack fino a che non si raggiunge l'area dati statici o eventualmente l'area HEAP, se implementata.

5. Modi di indirizzamento e trama bitcode.

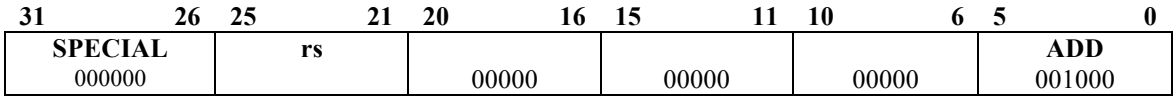
Di seguito sono elencate alcuni possibili indirizzamenti e le relative trame disponibili in MIPS32:

- **A registro** (trama di tipo R)

L'operando (l'indirizzo) è il contenuto di un registro della CPU: il nome (numero di indice del registro) del registro è specificato nell'istruzione.



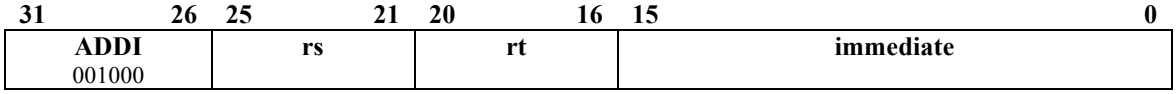
Format: `ADD rd, rs, rt` **MIPS32**
Purpose: To add 32-bit integers. If an overflow occurs, then trap.
Description: `GPR[rd] ← GPR[rs] + GPR[rt]`



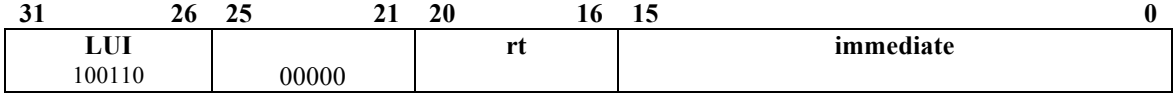
Format: `JR rs`
Purpose: To execute a branch to an instruction address in a register
Description: `PC ← GPR[rs]`

- **Immediato** (trama di tipo I)

L'operando (o uno degli operandi) è una costante il cui valore è contenuto nell'istruzione.



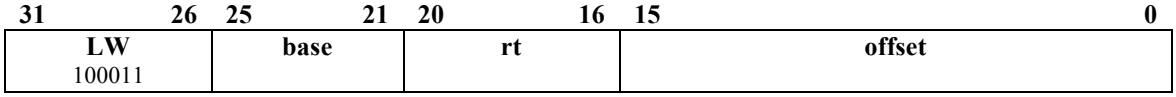
Formato: `ADDI rt, rs, immediate`
Purpose: To load a constant into the upper half of a word
Description: `GPR[rt] ← immediate || 00000000 00000000`



Formato: `LUI rt, immediate`
Purpose: To load a constant into the upper half of a word
Description: `GPR[rt] ← immediate || 0`$_{16}$

- **Indirizzamento con base** (trama di tipo I)

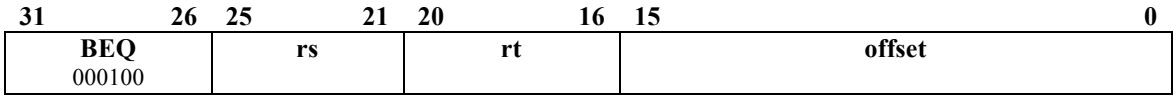
L'operando è in una locazione di memoria il cui indirizzo si ottiene sommando il contenuto di un registro base ad un valore costante (*offset o spiazzamento*) contenuto nell'istruzione.



Format: `LW rt, offset(base)`
Purpose: To load a word from memory for an atomic read-modify-write
Description: `GPR[rt] ← memory[GPR[base] + offset]`

- **Relativo** (trama di tipo I)

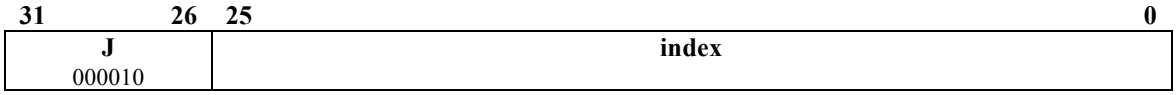
L'istruzione è in una locazione di memoria il cui indirizzo si ottiene sommando il contenuto del Program Counter ad un valore costante (offset o spiazzamento) contenuto nell'istruzione:



Format: `BEQ rs, rt, offset`
Purpose: To provide a conditional branch
Description: `IF (GPR[rs]==GPR[rt]) PC←PC+4+offset*4 ELSE PC←PC+4`

- **Pseudo-assoluto** (trama di tipo J)

Una parte dell'indirizzo è presente come valore costante nell'istruzione ma deve essere completato nei suoi bit più significativi. Questa costante si può intendere come offset rispetto alla posizione 0.



Format: `J target`
Purpose: To branch within the current 256 MB-aligned region
Description: `PC ← (PC+4)31..28 || index || 00`

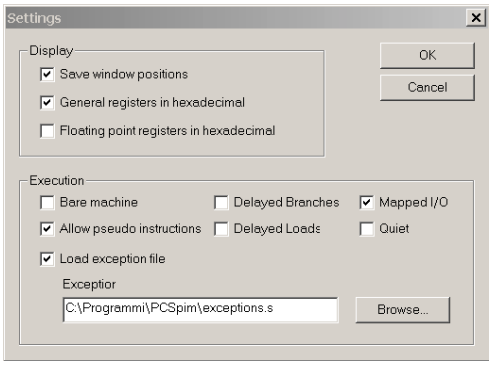
6. Il simulatore MIPS32 PCSpim

PCSpim è un simulatore di CPU MIPS. Tramite **PCSpim** è possibile caricare programmi per MIPS in linguaggio assembly MIPS, tradurli in linguaggio macchina ed eseguirli sia in modalità passo-passo (viene eseguita una sola istruzione per volta quindi il sistema attende una conferma per proseguire con la successiva) che in modalità continua (il codice viene eseguito senza interruzioni). E' possibile vedere prima, durante e dopo l'esecuzione lo stato dei registri ed alterarli se necessario. Sono disponibili un certo numero di funzioni predefinite nel S.O. richiamabili attraverso le **syscall**.

Il **PCSpim** permette di simulare MIPS con diversi comportamenti. Attraverso il menu **Setting..** è possibile specificare:

- se le pseudo-istruzioni sono accettabili o no
- se si desidera implementare salti e brach ritardati (che altera la sequenza ed i tempi di esecuzione di istruzioni eseguite dal MIPS a causa del pipelining) **Nota: se attivate i salti e i caricamenti ritardati dovete considerare il delay slot dopo i salti ed i caricamenti.**
- se si desidera simulare il MIPS reale (*bare-machine*) in tutte le sue caratteristiche senza considerare le convenzioni, ad esempio, per poter usare il registro **at**. **Nota: l'opzione *bare-machine* implica implicitamente i salti/branch ritardati.**
- la routine di gestione delle eccezioni (di default Pcspim carica una routine minimale presente nella cartella dove è stato installato).

Di seguito sono sono indcati i settaggi di default, necessari per eseguire i programmi Assembly con pseudo-istruzioni senza delay slot/branches:



La GUI è composta da quattro aree: l'area di visualizzazione dei registri, l'area codice (assembly ed LM), l'area dati (data, stack e kernel) ed un'area di log in cui è possibile vedere quali istruzioni sono effettivamente eseguite dalla CPU (utile se si usa la CPU con salti/caricamenti ritardati).

Un programma Assembly è composto *(in linea di principio)* principalmente da due segmenti distinti: il segmento **dati**, contenente i dati da elaborare, ed il segmento **codice o testo**, contenente il programma da eseguire (modello di calcolo **Harvard**[1]). Nel formato Assembly utilizzato da PCSpim le due aree sono indicate rispettivamente con **data:** e

[1] http://it.wikipedia.org/wiki/Architettura_Harvard

text:. l'area **dati** viene caricata a partire dalla locazione 0x10000000 mentre l'area **testo** viene caricata da **0x00400024**.

L'area codice effettivamente definita da PCSpim parte da 0x00400000. Il programma Assembly viene caricato e rilocato a partire dalla locazione 0x00400024 poiché PCSpim, prima di lanciare effettivamente il programma, effettua alcune operazioni di servizio. Dalla locazione 0x00400000 alla 0x00400024, è presente un frammento di codice che prepara i registri per permettere al programma di accedere a dati definibili dall'utente a run-time (dati passati al programma al momento del lancio tramite una finestra di dialogo). Successivamente chiama l'indirizzo **main:** (0x00400024) con una chiamata a sub-routine per permette un'uscita dall'esecuzione del programma "elegante" (vedi più avanti nel testo).

Tramite **F10** è possibile eseguire un programma in modalità passo-passo mentre **F5** esegue il programma in modalità continua fino al primo **break-point**, definibili tramite la combinazione **Ctrl+B**. Con **Symolator->Set value..** è possibile cambiare lo stato di un registro o di un area di memoria.

Il simulatore PCSpim mette a disposizione alcune procedure di sistema che possono essere usate per compiere delle azioni di I/O verso la console. Di seguito sono riportate le più comuni con un esempio di utilizzo:

Chiamate di sistema:

print_int:

```
                                # $a0 deve contenere l'intero da stampare,
                                # ex: move $a0 , $s0
            li $v0, 1           # $v0 codice della print_int
            syscall             # stampa della stringa
```

print_string:

```
            .data
str:        .asciiz "Inserire un numero intero:"
            [...]
            .text
            [...]
            li $v0, 4           # $v0 codice della print_string
            la $a0, str         # $a0 indirizzo della stringa con label str
            syscall             # stampa della stringa
```

read_int:

```
            li $v0, 5           # $v0 codice della print_string
            syscall             # stampa della stringa
```

exit:

```
            li $v0, 10          # $v0 codice della exit
            syscall             # exit
```

Per eseguire un programma in Pcspim occorre caricare il programma assembly con **File->Open..**. Una volta terminata l'esecuzione occorre ricaricare il programma tramite **Symulator->Reload**. **Symulator->Reinitialize** riporta Pcspim allo stato iniziale di apertura.

7. Le costanti Assembly

Attraverso delle direttive al compilatore è possibile definire delle costanti che verranno create nell'area dati. Di seguito sono elencate le direttive più usate:

- .space n — inserisce n byte inizializzata a 0
- .byte $b_1,b_2,...,b_n$ — inserisce i bytes $b_1,b_2,...b_n$ nella memoria dati
- .word $w_1,w_2,...,w_n$ — inserisce le word $w_1,w_2,...w_n$ nella memoria dati. Ogni word viene memorizzata con il byte meno significativo all'indirizzo di memoria più basso.
- .asciiz "...." — inserisce una stringa codificata in ASCII terminata da un byte a 0.

Altre direttive possono essere reperite nella manualistica di PCSpim.

8. Uso delle costanti tramite indirizzamento immediato.

Attraverso l'indirizzamento immediato è possibile impostare il valore di una delle due metà di un registro ad una costante intera a 16bit (ogni istruzione MIPS è lunga sempre 32 bit; tolto lo spazio per il codice operativo dell'istruzione, non rimane spazio sufficiente per una costante a 32bit). Nel caso sia necessario caricare in un registro tutti i 32bit, es. caricare in **$a0** l'indirizzo di inizio di una stringa**,** si usa la pseudo-istruzione **la** implementata dall'assemblatore in due passi: prima vengono caricati i 16 bit più significativi della parola tramite l'istruzione **lui** (*load upper immediate*) quindi vengono caricati i 16 bit meno significativi con un'istruzione **ori**.

Nota: *l'istruzione* ***lui*** *resetta i bit meno significativi del registro che manipola; ogni valore precedentemente presente nel registro è perso. Occorre quindi chiamarla sempre prima dell'operazione* ***ori.***

Ex: Programma dimostrativo dell'uso di costanti (file: somma.asm)

```
# Somma di due costanti.
# Dimostrativo dell'uso delle costanti
# e del modo di indirizzamento immediato
        .data
pad:    .space 1            # spazio inserito per fare diventare un valore
                            # diverso da 0 la semi-word bassa di str
str:    .asciiz "10 + 15 = "
                            # creo la stessa stringa con la direttiva byte
str2:   .byte 0x31,0x30,0x20,0x2b
        .byte 0x20,0x31,0x15,0x20
        .byte 0x3d,0x20,0x00
inutile: .byte 0xff         # aggiungo un dato inutile
        .text
        .globl main
main:
        li $t1, 10          # carica il valore decimale 10 nel reg. $t1
        li $t2, 15          # carica il valore decimale 15 nel reg. $t2
        add $s0, $t1, $t2   # $s0 = $t1 + $t2
```

```
        li $v0, 4          # stampa la stringa che inizia all'indirizzo str
        la $a0, str2       # pseudo-istruzione che carica una costante a 32 bit
        syscall

        li $v0, 1          # stampa risultato (10 + 15 = 25)
        move $a0 , $s0     #
        syscall

        li $v0, 10         # exit
        syscall
```

Nota: *il simulatore PCSpim usa la notazione little-endian, invece che la notazione big-endian che è definita per il MIPS. Di seguito è riportata l'immagine dell'area dati dopo il caricamento del programma somma.asm nel simulatore PCSpim:*

```
        DATA
[0x10000000]...[0x10010000]     0x00000000
[0x10010000]                    0x20303100  0x3531202b  0x00203d20  0x2b203031
[0x10010010]                    0x20153120  0xff00203d  0x00000000  0x00000000
[0x10010020]...[0x10040000]     0x00000000
```

Da notare che i dati vengono presentati a blocchi di una word invertiti rispetto all'effettivo ordine in memoria. Ad esempio, la prima word, 0x20303100, ha come byte meno significativo 0x00, corrispondente allo spazio creato con la direttiva ***.space 1.*** *Seguono poi i codici ASCII di "10 ", 0x31 0x30 0x20 corrispondenti all'inizio della stringa* ***str****. La stringa è terminata nella terza word dal valore 0x00. Di seguito è stata definita la stessa stringa utilizzando invece la direttiva* ***.byte****. Anche in questo caso la sequenza è terminata dal valore 0x00.*

Ex2: Versione compatta (file: somma2.asm)

```
# Somma di due costanti (seconda versione compatta)
# Dimostrativo dell'uso delle costanti
# e del modo di indirizzamento immediato
        .data
str:    .asciiz "10 + 15 = "
        .text
        .globl main
main:
        li $t1, 10         # carica il valore decimale 10 nel reg. $t1
        addi $s0, $t1, 15  # $s0 = $t1 + 15

        li $v0, 4          # stampa la stringa che inizia all'indirizzo str
        la $a0, str        # pseudo-istruzione che carica una costante a 32 bit
        syscall

        li $v0, 1          # stampa risultato (10 + 15 = 25)
        move $a0 , $s0     #
        syscall

        li $v0, 10         # exit
        syscall
```

9. Allocazione ed allineamento dei byte in memoria e nei registri.

Il MIPS trasferisce i dati da/verso la memoria in parole formate da **4 byte (word)** allineati a indirizzi multipli di 4. Nei trasferimenti che interessano word o half-word è necessario che i dati in memoria siano opportunamente allineati; in caso contrario si verifica una eccezione (*Exception 4 [Address error in inst/data fetch]*).

Ex: Programma dimostrativo per gli allineamenti (file: wordmem.asm).

```
# Programma dimostrativo per gli allineamenti delle word in memoria e nei registri.

        .data
carat:  .byte  0x21
        .byte  0x22
        .align 2         # riallineo i dati alle word = 2bit di indirizzo
testo:  .asciiz "0123456789"
space:  .asciiz " "
        .text
        .globl main
main:
        li $v0 , 4       # seleziono con $v0 la syscall print_str (4)
        la $a0 , testo   # indico in $a0 la stringa da stampare
        syscall          # invoco la syscall

        li $v0 , 4       # stampo uno spazio
        la $a0 , space
        syscall

        la $a0 , testo   # carico i primi 4 caratteri come fossero una word
        lw $t0 , 0($a0)
        #lw $t0 , 1($a0) # questa linea genera un eccezione perchè
                         # la word puntata non è allineata correttamente
        li $v0 , 1       # seleziono con $v0 la syscall print_int (1)
        move $a0 , $t0   # indico in $a0 l'intero da stampare
        syscall

        li $v0 ,....     # stampo uno spazio (vedi segmento di codice precedente)

        la $a0, carat    # carico l'indirizzo del byte puntato da carat
        lbu $t0, 0($a0)
        move $a0 , $t0
        li $v0 , 1
        syscall

        li $v0 ,....     # stampo uno spazio (vedi segmento di codice precedente)

        la $a0, carat    # carico l'indirizzo del byte puntato da carat
        lbu $t0, 1($a0)  # qui l'allineamento è irrilevante perchè stiamo
        li $v0 , 1       # trasferendo un byte.
        move $a0 , $t0
        syscall

        li $v0 ,....     # stampo uno spazio (vedi segmento di codice precedente)

        la $a0, carat    # carico l'indirizzo del byte puntato da carat
        lhu $t0, 2($a0)  # qui l'allineamento è importante perchè stiamo
        li $v0 , 1       # trasferendo una half-word.
        move $a0 , $t0
        syscall

        li $v0 , 1       # exit
        syscall
```

10. Salti condizionati

Il MIPS mette a disposizione due istruzioni base per la realizzazione di salti condizionati, **beq** e **bne**; il salto viene deciso testando l'uguaglianza tra due registri. La destinazione del salto viene calcolata relativamente alla posizione dell'istruzione di salto considerando un offset relativo su 16bit con segno.

La specifica MIPS32 indica che l'offset del salto va calcolato rispetto l'istruzione che segue il salto:

> ***Description:*** *if GPR[rs] = GPR[rt] then branch*
> *An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.*
> *[...]*

In altre parole, l'offset viene calcolato sottraendo l'indirizzo di arrivo del salto con l'indirizzo dell'istruzione che segue l'istruzione di salto e dividendo il risultato per 4.

Esempio: *In questo esempio* **Loop:** *indica l'indirizzo* **0x00400044** *mentre l'istruzione* **beq** *viene rilocata dall'assemblatore in* **0x00400054.**

```
Loop: 0x00400044    .....
                    .....
      0x00400054    beq $t0,$t1, Loop
      0x00400058    .....
```

Da cui:

Loop - 0x00400058 = 0xFFFFFEC (CA2) **= -0x14 = -20_{10}**

-20/4=-5

$-5_{10} = -101_2$

$-101_2 \rightarrow$ 0xFFFB in CA2 su 16 bit.

Nota: *il simulatore PCSpim si comporta in maniera leggermente diversa da quanto detto sopra. L'offset viene calcolato in maniera diversa a seconda se il salto è in avanti o all'indietro. Se il salto risulta all'indietro, come in questo caso, allora l'offset viene calcolato rispetto all'istruzione di salto e non alla successiva.*

L'offset su 16 bit limita a circa 2^{15} bytes la distanza di salto possibile. Se occorre saltare più lontano bisogna ridisegnare il codice in modo da aggiungere un istruzione **j** e realizzare un salto "lungo" (o **jr** se il salto è veramente "lungo", vedi sezione "*Salti oltre i 2 GigaBytes***"**).

Esempio: *Lo spezzone di codice:*

```
0x00400000      beq 0x01000000   # salto non realizzabile con
                                 # indirizzamento relativo
0x00400004      add ......
....
0x00401000      # Fine dell'area programma.
                # Inizia una zona libera della memoria
```

si può tradurre come:

```
0x00400000      bne 0x00401008   # inverto la condizione e creo una zona
                                 # libera per l'istruzione di salto
0x00400004      j 0x01000000     # realizzo un salto lungo.
0x00400008      add ......
```

o anche:

```
0x00400000      beq 0x00401004  # salto in una zona libera raggiungibile
0x00400004      add ......
....
0x00401000      #fine dell'area utilizzata dal programma.
0x00401004      j 0x01000000
....
```

Per realizzare test più complessi si costruiscono pseudo-istruzioni usando l'istruzione **slt** (**S**et on **L**ess **T**han). Vale infatti la seguente catena di equivalenze:

$$A<=B \leftrightarrow A-B<=0 \leftrightarrow B-A>=0 \leftrightarrow \text{sign}(B-A) = 0$$

La pseudo istruzione **ble $t0, $s0, Loop** (**B**ranch if **L**ess or **E**qual)viene tradotta nel seguente modo:

```
slt $at , $so , $t0
beq $1 , $0 , -16
```

Alcune traduzioni di pseudo-istruzioni di salto condizionato:

bgt rs ,rt , loop → rs>rt → rs-rt>0 → not (rs-rt<=0) → not (rt-rs>=0) → **slt $1,rt,rs ; bne $1,$0, loop**

bge rs ,rt , loop → rs>=rt → rs-rt>=0 → **slt $1,rs,rt ; beq $1,$0, loop**

blt rs ,rt , loop → rs<rt → rs-rt<0 → not(rs-rt>=0) → **slt $1,rs,rt ; bne $1,$0, loop**

ble rs ,rt , loop → rs<=rt → rs-rt<=0 → rt-rs >=0 → **slt $1,rt,rs ; beq $1,$0, loop**

11. Uso del registro $ra.

Ogni qualvolta viene eseguita l'istruzione **jal** il MIPS memorizza l'indirizzo dell'istruzione seguente nel registro **$31** (**$ra**). Se il registro non viene sovrascritto, alla fine della routine invocata è possibile usare questo valore per tornare indietro al punto di chiamata con l'istruzione

jr $ra.

Nota: Il PCSpim prima di lanciare effettivamente un programma caricato in memoria effettua alcune operazioni di servizio. Al termine di queste operazioni, prosegue l'esecuzione all'indirizzo indicato nel programma tramite l'etichetta ***main:*** *attraverso un'istruzione* ***jal*** *opportuna. Se il programma termina con un istruzione* ***jr $ra*** *e il registro* ***$ra*** *non è stato cambiato dal programma, allora l'esecuzione del programma ritornerà al termine alle istruzioni seguenti* ***jal main*** *che altro non sono che un'invocazione alla syscall* ***exit.***

Esempio*: Programma dimostrativo dell'uso del registro $ra (file: do_nothing.asm)*

```
# Programma che non fa niente.
# Dimostrativo dell'uso del registro $ra
        .data
        .text
        .globl main
main:
        move $s0 , $ra          # salvo $ra in $s0 poiché la subroutine
                                # do_nothing lo riscriverà quando invocata
        jal do_nothing          # richiamo una sub-routine che non fa niente
        move $ra , $s0          # ripristino $ra

        jr $ra                  # salto indietro all'exit di PCSpim

do_nothing:
        nop                     # non fare niente
        jr $ra                  # return
```

12. Implementazione ed uso degli array di word

Un array di word è una sequenza di word che si trovano in memoria una di seguito all'altra. Per riferirsi ad una data word si usa un indice numerico che rappresenta l'ordinale della word all'interno del set stesso. Normalmente si assegna alla prima word l'ordinale 0.
Dato l'ordinale **i**, l'indirizzo della word **i**-esima sarà memorizzzato nella word di indirizzo **inizio_array** + **i** * 4, dove **inizio_array** è l'indirizzo di inizio della tabella (a volte chiamata anche *tabella di lookup*).

Esempio: *Programma dimostrativo dell'uso degli array (* array.asm *).*

```
# Programma dimostrativo dell'uso degli array
# Traduzione in linguaggio macchina di A[1]=16
         .data
iarray:  .space 12
         .text
         .globl main
main:
         li $s0 , 1          # uso $s0 per memorizzare l'indice (in questo caso 1)
         la $s1 , iarray     # uso $s1 per memorizzare la base dell'array
         li $a0 , 0x10       # carico in $a0 il valore 16
         sll $t0, $s0, 2     # $t0 = i * 4
         add $t0, $s1, $t0   # $t0 = inizio_array + i*4
         sw $a0 , 0($t0)     # A[1]=16

         li $v0, 10            # exit
         syscall
```

13. Le Jump Address Table

Una Jump Address Table è un array di indirizzi corrispondenti al punto di ingresso di un set di procedure indirizzabili tramite l'indice di posizione. In questo modo è possibile saltare alla procedura i-esima della JAT tramite un salto ad indirizzamento indiretto (prima recupero l'indirizzo di salto dalla memoria poi salto all'indirizzo recuperato). E' usata di solito per definire le syscall e le routine associate agli interrupt (interrupt vettoriale).

Esempio: *Programma dimostrativo delle Jump Address Table (* lookup.asm *).*

```
# Programma dimostrativo dell'uso delle Jump Address Table
# per richiamare un set di procedure attraverso un indice
# (ex. syscall o switch)
        .data
lookup:  .space 12
msga:    .asciiz "Inserisci l'indice della funzione[0..2]: "
msg1:    .asciiz "Prima procedura"
msg2:    .asciiz "Seconda procedura"
msg3:    .asciiz "terza procedura"
         .text
         .globl main
main:
# carico la lookup table con gli indirizzi attuali
# poiché le procedure vengono rilocate a run-time.
         la $s0 , lookup   # $s0 contiene l'indirizzo della tabella lookup
         la $t0 , proc1    # salvo l'indirizzo dalla prima procedura
         sw $t0 , 0($s0)
         la $t0 , proc2    # salvo l'indirizzo dalla seconda procedura
         sw $t0 , 4($s0)
         la $t0 , proc3    # salvo l'indirizzo dalla terza procedura
         sw $t0 , 8($s0)

         li $v0 , 4        # stampo messaggio di richiesta
         la $a0 , msga
         syscall
         li $v0 , 5        # leggo un intero da console
         syscall           # numero in $v0

# calcolo l'indirizzo della procedura richiesta e la richiamo
         li $t0 , 4
         mul $t0 , $t0 , $v0   # $t0 = $vo * 4
         add  $t1 , $t0 , $s0  # $t1 = $t0 + lookup
         lw $t2 , 0($t1)       # $t2 = ($t1)
         jal $t2               # salto alla procedura

         li $v0, 10            # exit
         syscall

# set di procedure
proc1:   li $v0 , 4
         la $a0 , msg1
         syscall
         jr $ra
proc2:   li $v0 , 4
         la $a0 , msg2
         syscall
         jr $ra
proc3:   li $v0 , 4
         la $a0 , msg3
         syscall
         jr $ra
```

14. Salti oltre i 2 GigaBytes

Poiché l'istruzione **j** è in grado di memorizzare solo 26 bit dell'indirizzo di salto, il MIPS usa i 4 bit più significativi del PC per completare l'indirizzo di destinazione (i 2 bit meno significativi vengono messi a 0 poiché le istruzioni sono allineate alle word). Nel caso si debba saltare ad una locazione oltre il confine dei due gigabytes occorre caricare l'indirizzo di destinazione in un registro ed usare l'istruzione **jr**.

Esempio: *Si consideri il segmento di codice seguente:*

```
0x00400000      lui $t0, 0x1000     # Carico la parte alta dell'indirizzo
                ori $t0, $t0, 0x4   # Carico la parte bassa dell'indirizzo
                jr $t0              # Salto all'indirizzo 0x10000004
```

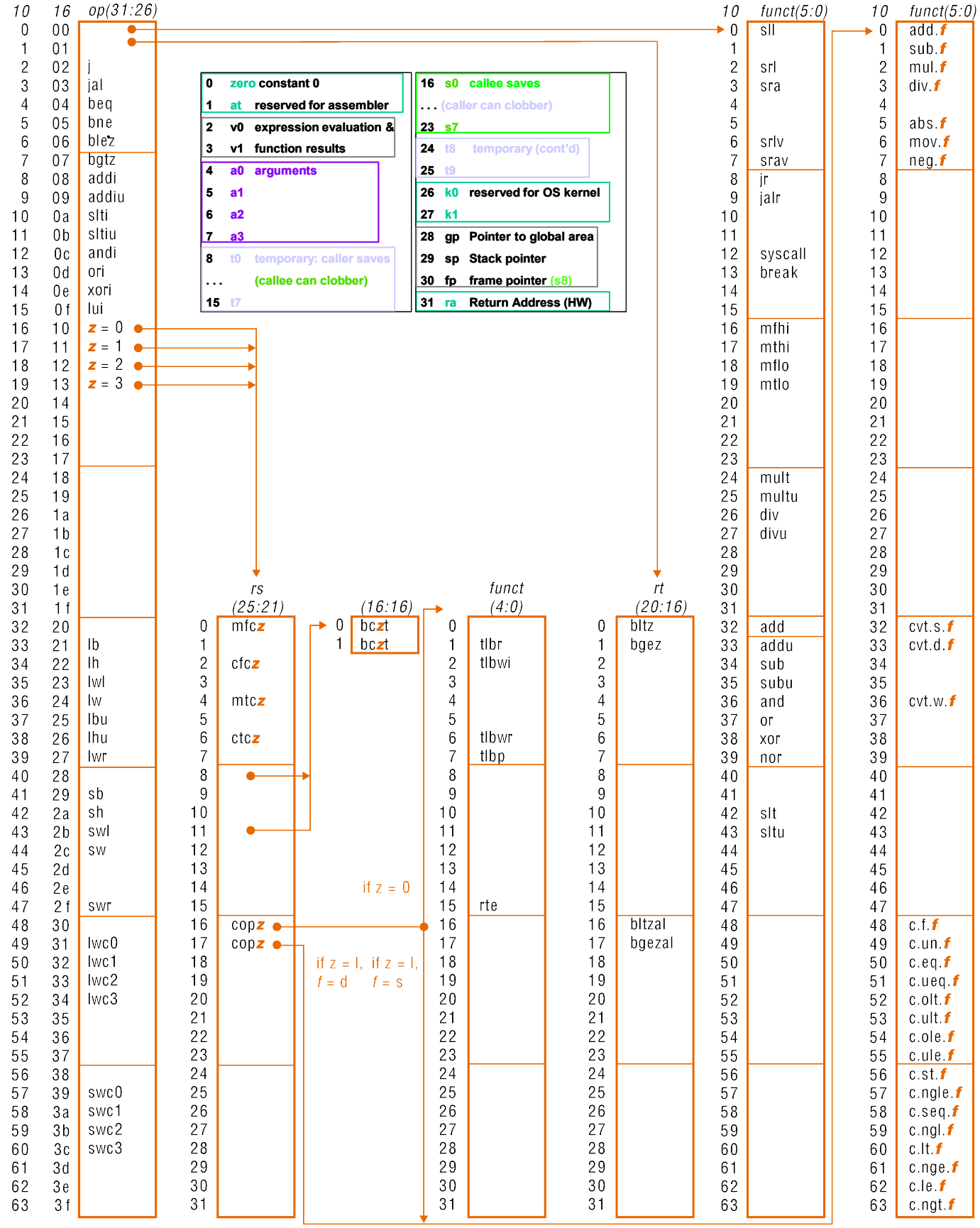


FIGURE A.19 MIPS opcode map. The values of each field are shown to its left. The first column shows the values in base 10 and the second shows base 16 for the op field (bits 31 to 26) in the third column. This op field completely specifies the MIPS operation except for 6 op values: 0, 1, 16, 17, 18, and 19. These operations are determined by other fields, identified by pointers. The last field (funct) uses "*f*" to mean "s" if rs = 16 and op = 17 or "d" if rs = 17 and op = 17. The second field (rs) uses "*z*" to mean "0", "1", "2", or "3" if op = 16, 17, 18, or 19, respectively. If rs = 16, the operation is specified elsewhere: if *z* = 0, the operations are specified in the fourth field (bits 4 to 0); if *z* = 1, then the operations are in the last field with *f* = s. If rs = 17 and *z* = 1, then the operations are in the last field with *f* = d.