
Manuale di EduMIPS64

Release 1.2

**Andrea Spadaccini ed il team di sviluppo di EduMIPS64
(traduzione italiana di Simona Ullo)**

01 June 2013

Indice

1	Formato dei file sorgenti	3
1.1	La sezione <i>.data</i>	3
1.2	La sezione <i>.code</i>	4
1.3	Il comando <i>#include</i>	6
2	Il set di istruzioni	7
2.1	Le istruzioni ALU	7
2.2	Istruzioni load/store	11
2.3	Istruzioni di controllo del flusso	12
2.4	L'istruzione <i>SYSCALL</i>	13
2.5	Altre istruzioni	15
3	Floating Point Unit	17
3.1	Valori speciali	17
3.2	Configurazione delle eccezioni	18
3.3	Direttiva <i>.double</i>	18
3.4	Registro FCSR	19
3.5	Instruction set	19
4	L'interfaccia utente	25
4.1	La barra dei menù	25
4.2	Finestre	27
4.3	Finestre di dialogo	28
4.4	Opzioni da riga di comando	28
4.5	Eseguire EduMIPS64	29
5	Listati di esempio	31
5.1	<i>SYSCALL</i>	31

EduMIPS64 è un simulatore di Instruction Set Architecture (ISA) MIPS64, progettato per eseguire piccoli programmi che utilizzino il sottoinsieme dell'Instruction Set MIPS64 implementato dal simulatore stesso; permette all'utente di vedere come le istruzioni si comportino nella pipeline, come gli stalli siano gestiti dalla CPU, lo stato di registri e memoria e molto altro. È classificabile sia come simulatore, sia come debugger visuale.

Il sito web del progetto è <http://www.edumips.org>, ed il codice sorgente è disponibile presso <http://github.com/lupino3/edumips64>. Per segnalare bug o inviare suggerimenti sul simulatore, è possibile aprire una issue su github o inviare una mail a bugs@edumips.org.

EduMIPS64 è stato progettato e sviluppato da un gruppo di studenti dell'Università degli Studi di Catania, ed ha tratto spunto, come interfaccia e come funzionamento, dal simulatore WinMIPS64, sebbene vi siano alcune differenze importanti con quest'ultimo.

Questo manuale vi introdurrà ad EduMIPS64, e spiegherà come utilizzarlo.

Il primo capitolo del manuale riguarda il formato dei file sorgente accettato dal simulatore, descrivendo i tipi di dato e le direttive, oltre ai parametri da linea di comando.

Nel secondo capitolo è presentata una panoramica del set di istruzioni MIPS64 utilizzato da EduMIPS64, con tutti i parametri richiesti e le indicazioni per il loro utilizzo.

Il terzo capitolo descrive la Floating Point Unit ed il suo insieme di istruzioni.

Il quarto capitolo è una descrizione dell'interfaccia utente di EduMIPS64, che espone lo scopo di ciascuna finestra e di ciascun menù, insieme ad una descrizione delle finestre di configurazione, del Dinero frontend, del manuale e delle opzioni da linea di comando.

Il quinto capitolo contiene alcune esempi pratici di utilizzo del simulatore.

Questo manuale si riferisce ad EduMIPS64 versione 1.2.

Formato dei file sorgenti

EduMIPS64 si propone di seguire le convenzioni usate negli altri simulatori MIPS64 e DLX, in modo tale da non creare confusione riguardante la sintassi per i vecchi utenti.

All'interno di un file sorgente sono presenti due sezioni: quella dedicata ai *dati* e quella in cui è contenuto il *codice*, introdotte rispettivamente dalle direttive *.data* e *.code*. Nel seguente listato è possibile vedere un semplice programma:

```
; Questo è un commento
    .data
label: .word 15      ; Questo è un commento in linea

    .code
daddi  r1, r0, 0
syscall 0
```

Per distinguere le varie parti di ciascuna linea di codice, può essere utilizzata una qualunque combinazione di spazi e tabulazioni, visto che il parser ignora spazi multipli.

I commenti possono essere introdotti utilizzando il carattere ”;” qualsiasi cosa venga scritta successivamente ad esso verrà ignorata. Un commento può quindi essere usato “inline” (dopo una direttiva) oppure in una riga a sè stante.

Le etichette possono essere usate nel codice per fare riferimento ad una cella di memoria o ad un’istruzione. Esse sono case insensitive. Per ciascuna linea di codice può essere utilizzata un’unica etichetta. Quest’ultima può essere inserita una o più righe al di sopra dell’effettiva dichiarazione del dato o dell’istruzione, facendo in modo che non ci sia nulla, eccetto commenti e linee vuote, tra l’etichetta stessa e la dichiarazione.

1.1 La sezione *.data*

La sezione *data* contiene i comandi che specificano il modo in cui la memoria deve essere riempita prima dell’inizio dell’esecuzione del programma. La forma generale di un comando *.data* è:

```
[etichetta:] .tipo-dato valore1 [, valore2 [, ...]]
```

EduMIPS64 supporta diversi tipi di dato, che sono descritti nella seguente tabella.

Tipo	Direttiva	Bit richiesti
Byte	<i>.byte</i>	8
Half word	<i>.word16</i>	16
Word	<i>.word32</i>	32
Double Word	<i>.word</i> or <i>.word64</i>	64

Dati di tipo *doubleword* possono essere introdotti sia dalla direttiva *.word* che dalla direttiva *.word64*.

Esiste una differenza sostanziale tra la dichiarazione di una lista di dati utilizzando un'unica direttiva oppure direttive multiple dello stesso tipo. EduMIPS64 inizia la scrittura a partire dalla successiva *double word* a 64 bit non appena trova un identificatore del tipo di dato, in tal modo la prima istruzione *.byte* del seguente listato inserirà i numeri 1, 2, 3 e 4 nello spazio di 4 byte, occupando 32 bit, mentre il codice delle successive quattro righe inserirà ciascun numero in una differente cella di memoria, occupando 32 byte:

```
.data
.byte    1, 2, 3, 4
.byte    1
.byte    2
.byte    3
.byte    4
```

Nella seguente tabella, la memoria è rappresentata utilizzando celle di dimensione pari ad 1 byte e ciascuna riga è lunga 64 bit. L'indirizzo posto alla sinistra di ogni riga della tabella è riferito alla cella di memoria più a destra, che possiede l'indirizzo più basso rispetto alle otto celle in ciascuna linea.

0	0	0	0	0	4	3	2	1
8	0	0	0	0	0	0	0	1
16	0	0	0	0	0	0	0	2
24	0	0	0	0	0	0	0	3
36	0	0	0	0	0	0	0	4

Ci sono alcune direttive speciali che devono essere discusse: *.space*, *.ascii* e *.asciiz*.

La direttiva *.space* è usata per lasciare dello spazio vuoto in memoria. Essa accetta un intero come parametro, che indica il numero di byte che devono essere lasciati liberi. Tale direttiva è utile quando è necessario conservare dello spazio in memoria per i risultati dei propri calcoli.

La direttiva *.ascii* accetta stringhe contenenti un qualunque carattere ASCII, ed alcune "sequenze di escape", simili a quelle presenti nel linguaggio C, che sono descritte nella seguente tabella, ed inserisce tali stringhe in memoria.

Sequenza di escape	Significato	Codice ASCII
\0	Byte nullo	0
\t	Tabulazione orizzontale	9
\n	Carattere di inizio nuova linea	10
\"	Doppi apici	34
	Backslash	92

La direttiva *.asciiz* si comporta esattamente come il comando *.ascii*, con la differenza che essa pone automaticamente alla fine della stringa un byte nullo.

1.2 La sezione *.code*

La sezione *code* contiene le istruzioni che saranno eseguite dal simulatore a run-time. La forma generale di un comando *.code* è:

```
[etichetta:] istruzione [param1 [, param2 [, param3]]]
```

Essa può essere indicata anche con la direttiva *.text*.

Il numero e il tipo di parametri dipendono dall'istruzione stessa.

Le istruzioni possono accettare tre tipi di parametri:

- *Registri* un parametro di tipo registro è indicato da una “r” maiuscola o minuscola, o da un carattere “\$”, a fianco del numero di registro (tra 0 e 31). Ad esempio, le scritture “r4”, “R4” e “\$4” identificano tutt’e tre il quarto registro;
- *Valori immediati* un valore immediato pu’o essere un numero o un’etichetta; il numero può essere specificato in base 10 o in base 16. I numeri in base 10 sono inseriti semplicemente scrivendo il numero utilizzando l’usuale notazione decimale; i numeri in base 16 si inseriscono aggiungendo all’inizio del numero il prefisso “0x”;
- *Indirizzi* un indirizzo è composto da un valore immediato seguito dal nome di un registro tra parentesi. Il valore del registro sarà usato come base, quello dell’immediato come offset.

La dimensione dei valori immediati è limitata al numero di bit disponibili nella codifica associata all’istruzione.

è possibile utilizzare gli alias standard MIPS per i primi 32 registri, mettendo in coda ai prefissi standard per i registri (“r”, “\$”, “R”) uno degli alias indicati nella seguente tabella.

Registro	Alias
0	<i>zero</i>
1	<i>at</i>
2	<i>v0</i>
3	<i>v1</i>
4	<i>a0</i>
5	<i>a1</i>
6	<i>a2</i>
7	<i>a3</i>
8	<i>t0</i>
9	<i>t1</i>
10	<i>t2</i>
11	<i>t3</i>
12	<i>t4</i>
13	<i>t5</i>
14	<i>t6</i>
15	<i>t7</i>
16	<i>s0</i>
17	<i>s1</i>
18	<i>s2</i>
19	<i>s3</i>
20	<i>s4</i>
21	<i>s5</i>
22	<i>s6</i>
23	<i>s7</i>
24	<i>t8</i>
25	<i>t9</i>
26	<i>k0</i>
27	<i>k1</i>
28	<i>gp</i>
29	<i>sp</i>
30	<i>fp</i>
31	<i>ra</i>

1.3 Il comando *#include*

Nei sorgenti può essere utilizzato il comando **#include* nomefile*, che ha l'effetto di inserire, al posto della riga contenente questo comando, il contenuto del file *nomefile*. Questo comando è utile se si vogliono includere delle funzioni esterne, ed è dotato di un algoritmo di rilevamento dei cicli, che impedisce di eseguire inclusioni circolari tipo “*#include A.s*” nel file *B.s* e “*#include B.s*” nel file *A.s*.

Il set di istruzioni

In questa sezione verrà illustrato il repertorio delle istruzioni MIPS64 riconosciute da EduMIPS64. È possibile effettuare due differenti classificazioni: una basata sulla funzionalità delle istruzioni e l'altra basata sul tipo di parametri.

La prima classificazione suddivide le istruzioni in tre categorie: istruzioni ALU, istruzioni Load/Store, istruzioni di controllo del flusso. I prossimi tre paragrafi descriveranno ciascuna categoria e le istruzioni che vi appartengono.

Il quarto paragrafo descriverà le istruzioni che non rientrano in nessuna delle tre categorie sopraelencate.

2.1 Le istruzioni ALU

L'unità logico-aritmetica (ALU) fa parte dell'unità esecutiva di una CPU ed assume il ruolo di esecuzione di operazioni logiche ed aritmetiche. Il gruppo di istruzioni ALU conterrà quindi quelle istruzioni che effettuano questo tipo di operazioni.

Le istruzioni ALU possono essere suddivise in due gruppi: *tipo R* e *tipo I*.

Quattro di esse utilizzano due registri speciali: LO e HI. Tali registri sono interni alla CPU ed è possibile accedere al loro valore mediante le istruzioni *MFLO* e *MFHI*.

Ecco la lista delle istruzioni ALU di tipo R.

- *AND rd, rs, rt*
Esegue un AND bit a bit tra rs ed rt, e pone il risultato in rd.
- *ADD rd, rs, rt*
Somma il contenuto dei registri a 32-bit rs ed rt, considerandoli come valori con segno, e pone il risultato in rd. Lancia un'eccezione in caso di overflow.
- *ADDU rd, rs, rt*
Somma il contenuto dei registri a 32-bit rs ed rt, e pone il risultato in rd. Non si verificano eccezioni di overflow.
- *DADD rd, rs, rt*
Somma il contenuto dei registri a 64-bit rs ed rt, considerandoli come valori con segno, e pone il risultato in rd. Lancia un'eccezione in caso di overflow.

- *DADDU rd, rs, rt*

Somma il contenuto dei registri a 64-bit rs ed rt, e pone il risultato in rd. Non si verificano eccezioni di overflow.

- *DDIV rs, rt*

Esegue la divisione tra i registri a 64-bit rs ed rt, ponendo i 64-bit del quoziente in LO ed i 64-bit del resto in HI.

- *DDIVU rs, rt*

Esegue la divisione tra i registri a 64-bit rs ed rt, considerandoli come valori senza segno e ponendo i 64-bit del quoziente in LO ed i 64-bit del resto in HI.

- *DIV rs, rt*

Esegue la divisione tra i registri a 32-bit rs ed rt, ponendo i 32-bit del quoziente in LO ed i 32-bit del resto in HI.

- *DIVU rs, rt*

Esegue la divisione tra i registri a 32-bit rs ed rt, considerandoli come valori senza segno e pone i 32-bit del quoziente in LO ed i 32-bit del resto in HI.

- *DMULT rs, rt*

Esegue il prodotto tra i registri a 64-bit rs ed rt, ponendo i 64 bit bassi del risultato nel registro speciale LO e i 64 bit alti del risultato nel registro speciale HI.

- *DMULTU rs, rt*

Esegue il prodotto tra i registri a 64-bit rs ed rt, considerandoli come valori senza segno e ponendo i 64 bit bassi del risultato nel registro speciale LO e i 64 bit alti del risultato nel registro speciale HI.

- *DSLL rd, rt, sa*

Effettua uno shift verso sinistra del registro a 64-bit rt, di un numero di bit indicato nel valore immediato (positivo compreso tra 0 e 63) sa, e pone il risultato in rd. I bit liberi vengono posti a zero.

- *DSLLV rd, rt, rs*

Effettua uno shift verso sinistra del registro a 64-bit rt, di un numero di bit specificato nei 6 bit bassi del registro rs che verrà letto come valore senza segno, e pone il risultato in rd. I bit liberi vengono posti a zero.

- *DSRA rd, rt, sa*

Effettua uno shift verso destra del registro a 64-bit rt, di un numero di bit specificato nel valore senza segno immediato (positivo compreso tra 0 e 63) sa, e pone il risultato in rd. I bit liberi vengono posti a zero se il bit più a sinistra di rs è zero, altrimenti vengono posti a uno.

- *DSRAV rd, rt, rs*

Effettua uno shift verso destra del registro a 64-bit rt, di un numero di bit specificato nei 6 bit bassi del registro rs che verrà letto come valore senza segno, e pone il risultato in rd. I bit liberi vengono posti a zero se il bit più a sinistra di rs è zero, altrimenti vengono posti a uno.

- *DSRL rd, rs, sa*

Effettua uno shift verso destra del registro a 64-bit rt, di un numero di bit specificato nel valore immediato (positivo compreso tra 0 e 63) sa, e pone il risultato in rd. I bit liberi vengono posti a zero.

- *DSRLV rd, rt, rs*

Effettua uno shift verso destra del registro a 64-bit rt, di un numero di bit specificato nei 6 bit bassi del registro rs che verrà letto come valore senza segno, e pone il risultato in rd. I bit liberi vengono posti a zero.

- *DSUB rd, rs, rt*

Sottrae il valore del registro a 64-bit *rt* al valore del registro a 64-bit *rs*, considerandoli come valori con segno, e pone il risultato in *rd*. Lancia un'eccezione in caso di overflow.

- *DSUBU rd, rs, rt*

Sottrae il valore del registro a 64-bit *rt* al valore del registro a 64-bit *rs*, e pone il risultato in *rd*. Non si verificano eccezioni di overflow.

- *MFLO rd*

Copia il contenuto del registro speciale LO in *rd*.

- *MFHI rd*

Copia il contenuto del registro speciale HI in *rd*.

- *MOVN rd, rs, rt*

Se *rt* è diverso da zero, copia il contenuto di *rs* in *rd*.

- *MOVZ rd, rs, rt*

Se *rt* è uguale a zero, copia il contenuto di *rs* in *rd*.

- *MULT rs, rt*

Esegue il prodotto tra i registri a 32-bit *rs* ed *rt*, ponendo i 32 bit bassi del risultato nel registro speciale LO e i 32 bit alti del risultato nel registro speciale HI.

- *MULTU rs, rt*

Esegue il prodotto tra i registri a 32-bit *rs* ed *rt*, considerandoli come valori senza segno e ponendo i 32 bit bassi del risultato nel registro speciale LO e i 32 bit alti del risultato nel registro speciale HI.

- *OR rd, rs, rt*

Esegue un OR bit a bit tra *rs* ed *rt*, e pone il risultato in *rd*.

- *SLL rd, rt, sa*

Effettua uno shift verso sinistra del registro a 32-bit *rt*, di un numero di bit indicati nel valore immediato (positivo compreso tra 0 e 63) *sa*, e pone il risultato nel registro a 32-bit *rd*. I bit liberi vengono posti a zero.

- *SLLV rd, rt, rs*

Effettua uno shift verso sinistra del registro a 32-bit *rt*, di un numero di bit specificato nei 5 bit bassi del registro *rs* che verrà letto come valore senza segno, e pone il risultato nel registro a 32-bit *rd*. I bit liberi vengono posti a zero.

- *SRA rd, rt, sa*

Effettua uno shift verso destra del registro a 32-bit *rt*, di un numero di bit specificato nel valore immediato (positivo compreso tra 0 e 63) *sa*, e pone il risultato nel registro a 32-bit *rd*. I bit liberi vengono posti a zero se il bit più a sinistra di *rs* è zero, altrimenti vengono posti a uno.

- *SRAV rd, rt, rs*

Effettua uno shift verso destra del registro a 32-bit *rt*, di un numero di bit specificato nei 5 bit bassi del registro *rs* che verrà letto come valore senza segno, e pone il risultato nel registro a 32-bit in *rd*. I bit liberi vengono posti a zero se il bit più a sinistra di *rs* è zero, altrimenti vengono posti a uno.

- *SRL rd, rs, sa*

Effettua uno shift verso destra del registro a 32-bit *rt*, di un numero di bit specificato nel valore immediato (positivo compreso tra 0 e 63) *sa*, e pone il risultato nel registro a 32-bit *rd*. I bit liberi vengono posti a zero.

- *SRLV rd, rt, rs*

Effettua uno shift verso destra del registro a 32-bit rt, del numero di bit specificato nei 5 bit bassi del registro rs che verrà letto come valore senza segno, e pone il risultato nel registro a 32-bit rd. I bit liberi vengono posti a zero.

- *SUB rd, rs, rt*

Sottrae il valore del registro a 32-bit rt al valore del registro a 32-bit rs, considerandoli come valori con segno, e pone il risultato in rd. Lancia un'eccezione in caso di overflow.

- *SUBU rd, rs, rt*

Sottrae il valore del registro a 32-bit rt al valore del registro a 32-bit rs, e pone il risultato in rd. Non si verificano eccezioni di overflow.

- *SLT rd, rs, rt*

Pone il valore di rd ad 1 se il valore contenuto in rs è minore di quello contenuto in rt, altrimenti pone rd a 0. Questa istruzione esegue un confronto con segno.

- *SLTU rd, rs, rt*

Pone il valore di rd ad 1 se il valore contenuto in rs è minore di quello contenuto in rt, altrimenti pone rd a 0. Questa istruzione esegue un confronto senza segno.

- *XOR rd, rs, rt*

Esegue un OR esclusivo (XOR) bit a bit tra rs ed rt, e pone il risultato in rd.

Ecco la lista delle istruzioni ALU di tipo I.

- *ADDI rt, rs, immediate*

Effettua la somma tra il registro a 32 bit rs ed il valore immediato, ponendo il risultato in rt. Questa istruzione considera gli operandi come valori con segno. Lancia un'eccezione in caso di overflow.

- *ADDIU rt, rs, immediate*

Effettua la somma tra il registro a 32 bit rs ed il valore immediato, ponendo il risultato in rt. Non si verificano eccezioni di overflow.

- *ANDI rt, rs, immediate*

Esegue un AND bit a bit tra rs ed il valore immediato, ponendo il risultato in rt.

- *DADDI rt, rs, immediate*

Effettua la somma tra il registro a 64 bit rs ed il valore immediato, ponendo il risultato in rt. Questa istruzione considera gli operandi come valori con segno. Lancia un'eccezione in caso di overflow.

- *DADDIU rt, rs, immediate*

Effettua la somma tra il registro a 64 bit rs ed il valore immediato, ponendo il risultato in rt. Non si verificano eccezioni di overflow.

- *DADDUI rt, rs, immediate*

Effettua la somma tra il registro a 64 bit rs ed il valore immediato, ponendo il risultato in rt. Non si verificano eccezioni di overflow.

- *LUI rt, immediate*

Carica la costante definita dal valore immediato nella metà superiore dei 32 bit inferiori di rt, effettuando l'estensione del segno sui 32 bit superiori del registro.

- *ORI rt, rs, immediate*

Effettua l'OR bit a bit tra rs ed il valore immediato, ponendo il risultato in rt.

- *SLTI rt, rs, immediate*

Pone il valore di rt ad 1 se il valore di rs è minore di quello dell'immediato, altrimenti pone rt a 0. Questa operazione effettua un confronto con segno.

- *SLTUI rt, rs, immediate*

Pone il valore di rt ad 1 se il valore di rs è minore di quello dell'immediato, altrimenti pone rt a 0. Questa operazione effettua un confronto senza segno.

- *XORI rt, rs, immediate*

Effettua l'OR esclusivo bit a bit tra rs ed il valore immediato, ponendo il risultato in rt.

2.2 Istruzioni load/store

Questa categoria contiene tutte le istruzioni che effettuano trasferimenti di dati tra i registri e la memoria. Ognuna di esse è espressa nella forma:

[etichetta] *istruzione* rt, *offset*(base)

In base all'utilizzo di un'istruzione load oppure store, rt rappresenterà di volta in volta il registro sorgente o destinazione; offset è un'etichetta o un valore immediato e base è un registro. L'indirizzo è ottenuto sommando al valore del registro 'base' il valore immediato di *offset*.

L'indirizzo specificato deve essere allineato in base al tipo di dato che si sta trattando. Le istruzioni di caricamento che terminano con "U" considerano il contenuto del registro rt come un valore senza segno.

Ecco la lista delle istruzioni di caricamento (LOAD):

- *LB rt, offset(base)*

Carica il contenuto della cella di memoria all'indirizzo specificato da offset e base nel registro rt, considerando tale valore come byte con segno.

- *LBU rt, offset(base)*

Carica il contenuto della cella di memoria all'indirizzo specificato da offset e base nel registro rt, considerando tale valore come byte senza segno.

- *LD rt, offset(base)*

Carica il contenuto della cella di memoria all'indirizzo specificato da offset e base nel registro rt, considerando tale valore come una double word.

- *LH rt, offset(base)*

Carica il contenuto della cella di memoria all'indirizzo specificato da offset e base nel registro rt, considerando tale valore come una half word con segno.

- *LHU rt, offset(base)*

Carica il contenuto della cella di memoria all'indirizzo specificato da offset e base nel registro rt, considerando tale valore come una half word senza segno.

- *LW rt, offset(base)*

Carica il contenuto della cella di memoria all'indirizzo specificato da offset e base nel registro rt, considerando tale valore come una word con segno.

- *LWU rt, offset(base)*

Carica il contenuto della cella di memoria all'indirizzo specificato da offset e base nel registro rt, considerando tale valore come una word senza segno.

Ecco la lista delle istruzioni di memorizzazione (STORE):

- *SB rt, offset(base)*

Memorizza il contenuto del registro rt nella cella di memoria specificata da offset e base, considerando tale valore come un byte.

- *SD rt, offset(base)*

Memorizza il contenuto del registro rt nella cella di memoria specificata da offset e base, considerando tale valore come una double word.

- *SH rt, offset(base)*

Memorizza il contenuto del registro rt nella cella di memoria specificata da offset e base, considerando tale valore come una half word.

- *SW rt, offset(base)*

Memorizza il contenuto del registro rt nella cella di memoria specificata da offset e base, considerando tale valore come una word.

2.3 Istruzioni di controllo del flusso

Le istruzioni di controllo del flusso sono utilizzate per alterare l'ordine delle istruzioni prelevate dalla CPU nella fase di fetch. È possibile fare una distinzione tra tali istruzioni: tipo R, tipo I e tipo J.

Tali istruzioni eseguono il salto alla fase di Instruction Decode (ID), ogni qual volta viene effettuato un fetch inutile. In tal caso, due istruzioni vengono rimosse dalla pipeline, ed il contatore degli stalli dovuti ai salti effettuati viene incrementato di due unità.

Ecco la lista delle istruzioni di controllo del flusso di tipo R:

- *JALR rs*

Pone il contenuto di rs nel program counter, e salva in R31 l'indirizzo dell'istruzione che segue l'istruzione JALR, che rappresenta il valore di ritorno.

- *JR rs*

Pone il contenuto di rs nel program counter.

Ed ecco le istruzioni di controllo del flusso di tipo I:

- *B offset*

Salto incondizionato ad offset.

- *BEQ rs, rt, offset*

Salta ad offset se rs è uguale ad rt.

- *BEQZ rs, offset*

Salta ad offset se rs è uguale a zero.

- *BGEZ rs, offset*

Effettua un salto relativo al PC ad offset se rs è maggiore di zero.

- *BNE rs, rt, offset*

Salta ad offset se rs non è uguale ad rt.

- *BNEZ rs, offset*

Salta ad offset se rs non è uguale a zero.

Ecco la lista delle istruzioni di controllo del flusso di tipo J:

- *J target*

Pone il valore immediato nel program counter

- *JAL target*

Pone il valore immediato nel program counter, e salva in R31 l'indirizzo dell'istruzione che segue l'istruzione JAL, che rappresenta il valore di ritorno.

2.4 L'istruzione SYSCALL

L'istruzione SYSCALL offre al programmatore un'interfaccia simile a quella offerta da un sistema operativo, rendendo disponibili sei differenti chiamate di sistema (system call).

Le system call richiedono che l'indirizzo dei loro parametri sia memorizzato nel registro R14, e pongono il loro valore di ritorno nel registro R1. Tali system call sono il più possibile fedeli alla convenzione POSIX.

2.4.1 SYSCALL 0 - exit()

SYSCALL 0 non richiede alcun parametro nè ritorna nulla, semplicemente ferma il simulatore.

è opportuno notare che se il simulatore non trova SYSCALL 0 nel codice sorgente, o una qualsiasi istruzione equivalente (HALT TRAP 0), terminerà automaticamente alla fine del sorgente.

2.4.2 SYSCALL 1 - open()

SYSCALL 1 richiede due parametri: una stringa (che termini con valore zero) che indica il percorso del file che deve essere aperto, ed una double word contenente un intero che indica i parametri che devono essere usati per specificare come aprire il file.

Tale intero può essere costruito sommando i parametri che si vogliono utilizzare, scelti dalla seguente lista:

- *O_RDONLY (0x01)* Apre il file in modalità sola lettura;
- *O_WRONLY (0x02)* Apre il file in modalità sola scrittura;
- *O_RDWR (0x03)* Apre il file in modalità di lettura/scrittura;
- *O_CREAT (0x04)* Crea il file se non esiste;
- *O_APPEND (0x08)* In modalità di scrittura, aggiunge il testo alla fine del file;
- *O_TRUNC (0x08)* In modalità di scrittura, cancella il contenuto del file al momento della sua apertura.

È obbligatorio specificare una delle prime tre modalità. La quinta e la sesta sono esclusive, non è possibile specificare O_APPEND se si specifica O_TRUNC (e viceversa). Inoltre non si può specificare O_CREAT se si specifica O_RDONLY (oppure O_RDWR).

È possibile specificare una combinazione di modalità semplicemente sommando i valori interi ad esse associati. Ad esempio, se si vuole aprire un file in modalità di sola scrittura ed aggiungere il testo alla fine del file, si dovrà specificare la modalità $2 + 8 = 10$.

Il valore di ritorno delle chiamate di sistema è il nuovo descrittore del file (file descriptor) associato al file, che potrà essere utilizzato con le altre chiamate di sistema. Qualora si verifichi un errore, il valore di ritorno sarà -1.

2.4.3 SYSCALL 2 - close()

SYSCALL 2 richiede solo un parametro, il file descriptor del file che deve essere chiuso.

Qualora l'operazione termini con successo, SYSCALL 2 ritornerà 0, altrimenti -1. Possibili cause di errore sono il tentativo di chiudere un file inesistente, o di chiudere il file descriptor 0, 1 o 2, che sono associati rispettivamente allo standard input, allo standard output ed allo standard error.

2.4.4 SYSCALL 3 - read()

SYSCALL 3 richiede tre parametri: il file descriptor da cui leggere, l'indirizzo nel quale i dati letti dovranno essere copiati, il numero di byte da leggere.

Se il primo parametro è 0, il simulatore permetterà all'utente di inserire un valore mediante un'apposita finestra di dialogo. Se la lunghezza del valore immesso è maggiore del numero di byte che devono essere letti, il simulatore mostrerà nuovamente la finestra.

La chiamata di sistema ritorna il numero di byte effettivamente letti, o -1 se l'operazione di lettura fallisce. Possibili cause di errore sono il tentativo di leggere da un file inesistente, o di leggere dal file descriptor 1 (standard output) o 2 (standard error), oppure il tentativo di leggere da un file di sola scrittura.

2.4.5 SYSCALL 4 - write()

SYSCALL 4 richiede tre parametri: il file descriptor su cui scrivere, l'indirizzo dal quale i dati dovranno essere letti, il numero di byte da scrivere.

Se il primo parametro è 2 o 3, il simulatore mostrerà la finestra di input/output dove scriverà i dati letti.

Questa chiamata di sistema ritorna il numero di byte che sono stati scritti, o -1 se l'operazione di scrittura fallisce. Possibili cause di errore sono il tentativo di scrivere su un file inesistente, o sul file descriptor 0 (standard input), oppure il tentativo di scrivere su un file di sola lettura.

2.4.6 SYSCALL 5 - printf()

SYSCALL 5 richiede un numero variabile di parametri, il primo è la cosiddetta "format string" o stringa di formato. Nella stringa di formato possono essere inseriti alcuni segnaposto, descritti nella seguente lista: * %s parametro di tipo stringa; * %i parametro di tipo intero; * %d si comporta come %i; * %% carattere %

Per ciascuno dei segnaposto %s, %d o %i la SYSCALL 5 si aspetta un parametro, partendo dall'indirizzo del precedente.

Quando la SYSCALL trova un segnaposto per un parametro intero, si aspetta che il corrispondente parametro sia un valore intero, quando trova un segnaposto per un parametro stringa, si aspetta come parametro l'indirizzo della stringa stessa.

Il risultato visualizzato nella finestra di input/output, ed il numero di byte scritti posto in R1.

Qualora si verifichi un errore, R1 avrà valore -1.

2.5 Altre istruzioni

In questa sezione sono descritte istruzioni che non rientrano nelle precedenti categorie.

2.5.1 *BREAK*

L'istruzione *BREAK* solleva un'eccezione che ha l'effetto di fermare l'esecuzione se il simulatore è in esecuzione. Può essere utilizzata per il debugging.

2.5.2 *NOP*

L'istruzione *NOP* non fa nulla, ed è utilizzata per creare pause nel codice sorgente.

2.5.3 *TRAP*

L'istruzione *TRAP* è deprecated, rappresenta un'alternativa all'istruzione *SYSCALL*.

2.5.4 *HALT*

L'istruzione *HALT* è deprecated, rappresenta un'alternativa all'istruzione *SYSCALL* 0, che ferma il simulatore.

Floating Point Unit

Questo capitolo ¹ descrive le caratteristiche della Floating Point Unit (FPU) di EduMIPS64.

Nel primo paragrafo vengono introdotti il formato double, i valori speciali floating point definiti dallo standard che ne disciplina il trattamento nei calcolatori (IEEE 754), e le condizioni di eccezioni che i calcoli floating point possono provocare.

Nel secondo paragrafo viene illustrato come EduMIPS64 consenta di attivare e disattivare le trap relative alle condizioni di eccezione IEEE.

Nel terzo paragrafo si parla del modo in cui i valori double e i valori speciali vengono accettati dal simulatore per essere caricati in memoria.

Nel quarto paragrafo è introdotto il registro FCSR, usato dalla FPU per autogestirsi. In esso vengono memorizzate le informazioni riguardanti l'arrotondamento, i risultati booleani delle istruzioni di confronto e le politiche di gestione delle eccezioni IEEE.

Infine, nel quinto paragrafo sono elencate tutte le istruzioni dell'ISA MIPS64 implementate in EduMIPS64.

Prima di iniziare la trattazione della FPU, definiamo il dominio dei numeri in virgola mobile (floating point) a doppia precisione come $[-1.79E308, -4.94E-324] \cup \{0\} \cup [4.94E-324, 1.79E308]$.

3.1 Valori speciali

L'aritmetica in virgola mobile dei calcolatori è caratterizzata dal fatto che, anche in presenza di un'operazione matematica non valida, si potrebbe scegliere di continuare la computazione ignorando quanto è accaduto. In questo scenario, operazioni come divisioni tra zeri, oppure radici quadrate di numeri negativi devono generare comunque un risultato che, non essendo un numero (Not a Number), è trattato come qualcosa di diverso.

3.1.1 NaN o Invalid Operation

Lo standard IEEE 754, il quale regola la manipolazione dei numeri floating point nei calcolatori, ha definito che le operazioni matematiche non valide possono sia provocare una segnalazione durante l'esecuzione del programma (trap per la condizione di eccezione IEEE **Invalid Operation**), che fornire, come risultato, il valore speciale QNaN (Quit Not a Number). Un altro valore NaN, che genera incondizionatamente la stessa trap appena viene rilevato come

¹ Il presente capitolo è tratto dalla tesi di laurea di Massimo Trubia: "Progetto e implementazione di un modello di Floating Point Unit per un simulatore di CPU MIPS64".

operando, è SNaN (Signalling NaN). Tale valore è raramente utilizzato nelle applicazioni, e storicamente è stato usato per inizializzare le variabili.

3.1.2 Zeri o Underflow

Un altro valore speciale definito dallo standard è lo zero. Dal momento che il formato double non include lo zero nel dominio dei valori rappresentati, esso è considerato alla stregua di un valore speciale. Esistono uno zero positivo e uno negativo: quando si tenta di rappresentare un valore negativo molto vicino allo zero, ma fuori dal dominio rappresentabile dai double ($\in]-4.94\text{E}-324, 0[$), e si desidera a tutti i costi un risultato (non una trap per la condizione di eccezione **Underflow**), allora il numero restituito è -0. Analogamente, viene restituito +0 se si tenta di rappresentare un numero nell'intervallo $[0, 4.94\text{E}-324[$, e non si è preferito avere la stessa trap.

3.1.3 Infiniti od Overflow

Quando si tenta di rappresentare un numero estremamente grande ($\in]1.79\text{E}308, +\infty[$), o estremamente piccolo ($\in]-\infty, -1.79\text{E}308[$), al di fuori cioè del dominio rappresentabile dal formato double, vengono restituiti i valori speciali $+\infty$, nel primo caso, e $-\infty$ nel secondo caso. In alternativa, si può avere una trap per via della condizione di eccezione **Overflow**.

Gli infiniti potrebbero anche essere restituiti da una divisione per zero, nelle quali si effettua il prodotto tra il segno dello zero e quello del divisore, per restituire un opportuno infinito. Nel caso in cui non si voglia alcun valore restituito, si verifica la trap per la condizione di eccezione **Divide by zero**.

3.2 Configurazione delle eccezioni

EduMIPS64 consente di abilitare o disabilitare le trap relative alle 4 delle 5 condizioni di eccezione IEEE, implementate dalla scheda *Eccezioni FPU* della finestra *Configura* → *Impostazioni*. Se esse sono disabilitate, verrà fornito un risultato per qualunque operazione speciale la FPU effettui (si veda il Paragrafo *Valori speciali*). .. Nel .. caso illustrato in Figura *fig-exception_cfg*, in cui alcune .. caselle di controllo sono spuntate, se la CPU non maschera le eccezioni .. sincrone nel simulatore (Figura *fig-exception_mask_cfg*), .. verrà simulata una trap relativa alla condizione di eccezione IEEE che si è .. verificata (Figura *fig-invalid_operation_trap*). *_fig-exception_cfg*:
.. .. figure:: ../img/exception_cfg.png .. :scale: 50% Configurazione delle trap per le eccezioni IEEE
.. *_fig-exception_mask_cfg*: figure:: ../img/exception_mask_cfg.png .. :scale: 50% Opzio-
ne che maschera le eccezioni sincrone (disabilita tutte le trap) *_fig-invalid_operation_trap*: figure::
../img/invalid_operation_trap.png .. :scale: 50% Finestra che notifica la trap

3.3 Direttiva .double

La direttiva `.double`, da inserire nella sezione `.data` del file sorgente (`.s`), consente di allocare una cella della memoria di EduMIPS64, dove inserire un valore formattato *double*.

Le sintassi valide del comando sono:

```
nome_variabile: .double numero_double  
nome_variabile: .double parola_chiave
```

dove `numero_double` può essere rappresentato sia in forma estesa (`1.0, 0.003`), oppure in notazione scientifica (`3.7E-12, 0.5E32`). Invece, `parola_chiave` può assumere i valori `POSITIVEINFINITY`, `NEGATIVEINFINITY`, `POSITIVEZERO`, `NEGATIVEZERO`, `SNAN` e `QNAN`, consentendo l'inserimento diretto in memoria dei valori speciali.

3.4 Registro FCSR

L'FCSR (Floating point Control Status Register) è il registro che controlla i diversi aspetti funzionali della FPU. Esso è lungo 32 bit e, fino alla ridisegnazione grafica di EduMIPS64, sarà posto nella finestra delle statistiche.

Il campo **FCC** è costituito da 8 bit, identificati con numeri da 0 a 7. Le istruzioni condizionali (`C.EQ.D`, `C.LT.D`) lo utilizzano per memorizzare il risultato booleano di un confronto tra due registri.

I campi Cause, Enables e Flag gestiscono la dinamica delle eccezioni IEEE, illustrate nel Paragrafo *Valori speciali*. Essi sono costituiti, ognuno, da 5 bit identificati con le lettere V (Invalid operation), Z (Divide by zero), O (Overflow), U (Underflow) e I (Inexact); quest'ultimo bit non viene al momento utilizzato.

Il campo **Cause** indica se si è verificata una qualunque eccezione IEEE durante la simulazione, presentando un 1 nel relativo bit. È utile quando si esegue un programma dall'inizio alla fine senza fermarlo, per determinare se si è verificata una qualunque eccezione.

Il campo **Enable** mostra le eccezioni IEEE per le quali è attiva la trap. I bit di questo campo vengono modificati, anche senza resettare il simulatore, dalla finestra di configurazione. ... della Figura *fig-exception_cfg*.

Il campo **Flag** mostra le eccezioni IEEE che si sono verificate ma, non avendo la relativa trap attivata, hanno fornito come risultato dei valori speciali, illustrati nel Paragrafo *Valori speciali*.

Il campo **RM** mostra la modalità di arrotondamento corrente usata, in EduMIPS64, per le istruzioni che convertono numeri floating point in interi (si veda l'istruzione `CVT.L.D` per ulteriori dettagli).

3.5 Instruction set

Per una consultazione efficiente, le istruzioni dell'ISA MIPS64, implementate in EduMIPS64, vengono elencate in ordine alfabetico. Le operazioni eseguite vengono rappresentate mediante uno pseudocodice in cui l'*i*-esima cella di memoria è indicata con `memory[i]`, i bit del campo FCC del registro FCSR mediante `FCSR_FCC[cc]`, con `cc` ∈ [0,7].

In alcune istruzioni, per evitare ambiguità, i registri sono indicati come `GPR[i]` e `FPR[i]`, con `i` ∈ [0,31], ma nella maggior parte dei casi essi vengono indicati qualitativamente con la notazione `rx` o `fx`, dove `x` ∈ {d,s,t}. Le tre lettere servono solo a distinguere, al più, tre registri per ogni istruzione. Infine, i valori ritornati dalle operazioni di conversione vengono indicati con la notazione `convert_tipoconversione(registro[,tipo arrotondamento])`, dove il parametro tra parentesi quadre è presente solo in certe circostanze.

Per prendere confidenza con le istruzioni floating point, alcuni file sorgenti possono essere scaricati dal link <http://www.edumips.org/attachment/wiki/Upload/FPUMaxSamples.rar>.

- *ADD.D fd, fs, ft*

Descrizione: `fd = fs + ft`

Eccezioni: Le trap di Overflow e Underflow vengono generate se il risultato non può essere rappresentato secondo lo standard IEEE 754. Invalid Operation è generata se `fs` o `ft` contengono QNaN o SNaN, o se viene eseguita un'operazione non valida ($+\infty - \infty$).

- *BC1F cc, offset*

Descrizione: `if FCSR_FCC[cc] == 0 then branch`

Verifica se il valore booleano `FCSR_FCC[cc]` è falso ed effettua, in tal caso, un salto PC-relative. Questa istruzione può accedere al registro FCSR solo in lettura; l'informazione dev'essere scritta da una precedente istruzione condizionale del tipo `C.condizione.D`.

Esempio:

```
C.EQ.D 7, f1, f2
BC1F 7, label
```

In questo esempio `C.EQ.D` verifica l'uguaglianza tra i registri `f1` ed `f2`, scrivendo il risultato booleano del confronto nel bit 7 del campo FCC del registro FCSR. Dopodichè `BC1F` verifica se quel bit vale 0 (falso) e salta a `label`.

- *BCIT cc, offset*

Descrizione: `if FCSR_FCC[cc] == 1 then branch`

Verifica se il valore booleano `FCSR_FCC[cc]` è vero ed effettua, in tal caso, un salto PC-relative. Questa istruzione può accedere al registro FCSR solo in lettura; l'informazione dev'essere scritta da una precedente istruzione condizionale del tipo `C.condizione.D`.

Esempio:

```
C.EQ.D 7, f1, f2
BC1T 7, label
```

In questo esempio, `C.EQ.D` verifica l'uguaglianza tra i registri `f1` ed `f2`, scrivendo il risultato booleano del confronto nel bit 7 del campo FCC del registro FCSR. Dopodichè `BC1T` verifica se quel bit vale 1 (vero) e salta a `label`.

- *C.EQ.D cc, fs, ft*

Descrizione: `FCSR_FCC[cc] = (fs==ft)`

Verifica il predicato “uguale a” tra i due registri `fs` ed `ft` e salva il risultato booleano in `FCSR_FCC[cc]`. Questo valore verrà utilizzato da un'istruzione successiva per effettuare un salto condizionato (branch) o un movimento di dati condizionato tra registri floating point.

Esempio:

```
C.EQ.D 2, f1, f2
MOVT.D f8, f9, 2
```

In questo esempio `C.EQ.D` verifica l'uguaglianza tra i registri `f1` ed `f2`, scrivendo il risultato booleano del confronto nel bit 2 del campo FCC del registro FCSR. Dopodichè `MOVT.D` verifica se quel bit vale 1 (vero), e copia il registro `f9` su `f8`.

Eccezioni: Invalid Operation è lanciata quando `fs` o `ft` contengono valori QNaN (se attiva, si ha una trap) o SNaN (si ha sempre una trap).

- *C.LT.D cc, fs, ft*

Descrizione: `FCSR_FCC[cc] = (fs<ft)`

Verifica il predicato “minore di” (Less Than) tra i due registri `fs` ed `ft`, e salva il risultato booleano in `FCSR_FCC[cc]`. Questo valore verrà utilizzato da un'istruzione successiva, per effettuare un salto condizionato (branch), o per un movimento di dati condizionato tra registri floating point.

Esempio:

```
C.LT.D 2, f1, f2
BC1T 2, target
```

In questo esempio, `C.LT.D` verifica se `f1` è minore di `f2`, scrivendo il risultato booleano del confronto nel bit 2 del campo FCC del registro FCSR. Dopodichè, `BC1T` verifica se quel bit vale 1 (vero), e salta a `target`.

Eccezioni: Invalid Operation è lanciata quando `fs` o `ft` contengono valori QNaN (se attiva, si ha una trap) o SNaN (si ha sempre una trap).

- *CVT.D.L* *fd, fs*

Descrizione: `fd = convert_longToDouble(fs)`

Converte un long in un double.

Esempio:

```
DMTC1 r6, f5
CVT.D.L f5, f5
```

In questo esempio, DMTC1 copia il valore del GPR r6 nell'FPR f5. Successivamente, CVT.D.L converte il numero in f5 da long a double. Supponendo r6=52, dopo l'esecuzione di DMTC1, l'equivalente binario di 52 viene copiato nel registro f5 (f5 non contiene ancora il valore 52.0 perchè non è stato formattato ancora come double). Dopo l'esecuzione di CVT.D.L, f5=52.0.

Eccezioni: Invalid Operation è lanciata quando fs contiene QNaN, SNaN o Infinito

- *CVT.D.W* *fd, fs*

Descrizione: `fd = convert_IntToDouble(fs)`

Converte un int in un double

Esempio:

```
MTC1 r6, f5
CVT.D.W f5, f5
```

In questo esempio, MTC1 copia i 32 bit più bassi del GPR r6 nell'FPR f5. Successivamente, CVT.D.W, leggendo prima f5 come int, lo sovrascrive in double.

Supponendo `r6=0xAAAAAAAABBBBBBBB`, dopo l'esecuzione di MTC1 si ha che `f5=0xxxxxxxxxxxxxxxxxxxx`; si noti che i suoi 32 bit più alti (XX.X) sono UNDEFINED (non sono stati sovrascritti). CVT.D.W legge f5 come int (`f5=-1145324613`), formattandolo poi in double (`f5=0xC1D1111111400000 = -1.145324613E9`).

Eccezioni: Invalid Operation è lanciata quando fs contiene QNaN, SNaN o Infinito

- *CVT.L.D* *fd, fs*

Descrizione: `fd = convert_doubleToLong(fs, CurrentRoundingMode)`

Converte, dapprima arrotondandolo, un double in un long

Esempio:

```
CVT.L.D f5, f5
DMFC1 r6, f5
```

In questo esempio, CVT.L.D converte il double in f5 in un long. Dopodichè, DMFC1 copia l'FPR f5 nel GPR r6. Il risultato di questa istruzione dipende dalla modalità di arrotondamento corrente, che viene impostata dalla scheda *Arrotondamenti FPU* della finestra *Configura* → *Impostazioni*.

Tabella 3.1: Esempi sui tipi di arrotondamento

Tipo	Campo RM	Registro f5	Registro r6
Al più vicino	0	6.4	6
Al più vicino	0	6.8	7
Al più vicino	0	6.5	6 (al pari)
Al più vicino	0	7.5	8 (al pari)
Verso lo zero	1	7.1	7
Verso lo zero	1	-2.3	-2
Verso ∞	2	4.2	5
Verso ∞	2	-3.9	-3
Verso $-\infty$	3	4.2	4
Verso $-\infty$	3	-3.9	-4

- *CVT.W.D fd, fs*

Descrizione: `fd = convert_DoubleToInt(fs, CurrentRoundingMode)`

Converte un double in un int utilizzando la modalità di arrotondamento corrente, illustrata per l'istruzione `CVT.L.D`

Eccezioni: Invalid Operation è lanciata quando fs è Infinito, XNaN, o il risultato è fuori dall'intervallo degli interi con segno $[-2^{31}, 2^{31}-1]$

- *DIV.D fd, fs, ft*

Descrizione: `fd = fs \div ft`

Eccezioni: Le trap di Overflow e Underflow vengono generate se il risultato non può essere rappresentato secondo lo standard IEEE 754. Invalid Operation è generata se fs o ft contengono QNaN o SNaN, o se viene eseguita un'operazione non valida ($0 \div 0, \infty \div \infty$). Divide by zero è generata se è eseguita una divisione per zero che non ha per dividendo un XNaN ($5 \div 0$).

- *DMFC1 rt, fs*

Descrizione: `rt = fs`

Copia l'intero contenuto binario dell'FPR fs nel GPR rt. Nessun controllo di formato viene eseguito su fs prima della copia.

- *DMTC1 rt, fs*

Descrizione: `fs = rt`

Copia il contenuto binario del GPR rt nell' FPR fs.

- *L.D ft, offset(base)*

Descrizione: `ft = memory[GPR[base] + offset]`

Carica una doubleword (64 bit) dalla memoria all'FPR ft. Questa istruzione non appartiene all'ISA MIPS64; si consiglia l'utilizzo di `LDC1`.

Nota: *L.D* è un'istruzione non presente nell'ISA MIPS64, inclusa in EduMIPS64, solo per compatibilità con WinMIPS64

- *LDC1 ft, offset(base)*

Descrizione: `ft = memory[GPR[base] + offset]`

Carica una doubleword (64 bit) dalla memoria all'FPR ft

- *LWC1 ft, offset(base)*

Descrizione: `ft = memory[GPR[base] + offset]`

Carica una word (32 bit) dalla memoria all'FPR `ft`

- *MFC1 rt, fs*

Descrizione: `rt = readInt(fs)`

Legge l'FPR `fs` come int e scrive il GPR `rt` come long *Esempio:*

```
MFC1 r6, f5
SD r6, inmemoria(R0)
```

Si supponga `f5=0xAAAAAABBBBBBBB`; *MFC1* legge `f5` come int, cioè i 32 bit più bassi (interpreta `BBBBBBBB` come `-1145324613`), e lo scrive in `r6` (64 bit). Dopo l'esecuzione di *MFC1*, `r6=0xFFFFFFFFBBBBBBBB`, che equivale a `-1145324613` leggendo questo registro come long. Quindi in memoria, pur utilizzando l'istruzione *SD*, verrà scritta una doubleword con valore `-1145324613`. Questa operazione di conversione è chiamata *estensione del segno*, il cui approfondimento esula dagli scopi di questo manuale.

- *MOV.FD fd, fs, cc*

Descrizione: `if FCSR_FCC[cc] == 0 then fd=fs`

Verifica se la condizione di confronto booleana in `FCSR_FCC[cc]` è falsa e copia `fs` su `fd`. Nessun controllo sul formato viene realizzato su `fs`.

- *MOVT.D fd, fs, cc*

Descrizione: `if FCSR_FCC[cc] == 1 then fd=fs`

Verifica se la condizione di confronto booleana in `FCSR_FCC[cc]` è vera, e copia `fs` su `fd`. Nessun controllo sul formato viene realizzato su `fs`.

- *MOV.D fd, fs*

Descrizione: `fd = fs`

Copia `fs` su `fd` senza alcun controllo del formato di `fs`

- *MOVN.D fd, fs, rt*

Descrizione: `if rt != 0 then fd=fs`

Copia `fs` su `fd`, senza alcun controllo del formato di `fs`, se il GPR `rt` è diverso da zero

- *MOVZ.D fd, fs, rt*

Descrizione: `if rt == 0 then fd=fs`

Copia `fs` su `fd`, senza alcun controllo del formato di `fs`, se il GPR `rt` è uguale a zero

- *MTC1 rt, fs*

Descrizione: `fs = rt0..31`

Copia la word più bassa di `rt` scrivendola sull'FPR `fs`.

Esempio:

```
MTC1 r6, f5
```

Si supponga `r5=0xAAAAAABBBBBBBB`; *MTC1* legge i 32 bit più bassi di `r5` copiandoli nei 32 bit più bassi di `f5`. Dopo l'esecuzione di *MTC1*, `f5=0xFFFFFFFFBBBBBBBB`; si noti che i suoi 32 bit più alti (XX. .X) sono UNDEFINED (non sono stati sovrascritti).

- *MUL.D* *fd, fs, ft*

Descrizione: $fd = fs \times ft$

Eccezioni: Overflow e Underflow vengono generati se il risultato non può essere rappresentato secondo lo standard IEEE754. Invalid Operation è generata se *fs* o *ft* contiene QNaN o SNaN, o se si effettua un'operazione $\times \infty$, QNaN $\times$ numero)

- *S.D* *ft, offset(base)*

Descrizione: $memory[base+offset] = ft$

Copia la doubleword (64 bit) dell'FPR *ft* in memoria.

Nota: *S.D* è un'istruzione non presente nell'ISA MIPS64, inclusa in EduMIPS64, solo per compatibilità con WinMIPS64

- *SDC1* *ft, offset(base)*

Descrizione: $memory[base+offset] = ft$

Salva la doubleword (64 bit) dell'FPR *ft* in memoria.

- *SUB.D* *fd, fs, ft*

Descrizione: $fd = fs - ft$

Eccezioni: Overflow and Underflow vengono generati se il risultato non può essere rappresentato secondo lo standard IEEE754. Invalid Operation è generata se *fs* o *ft* contengono QNaN o SNaN, o se viene eseguita un'operazione non valida ($\infty - \infty$).

- *SWC1* *ft, offset(base)*

Descrizione: $memory[base+offset] = ft$

Salva la word (32 bit) dell'FPR *ft* in memoria.

L'interfaccia utente

L'interfaccia grafica EduMIPS64 si ispira a quella di WinMIPS64. Infatti, la finestra principale è identica, eccetto che per qualche menù.

La finestra principale di EduMIPS64 è caratterizzata da sei frame, che mostrano i differenti aspetti della simulazione. è inoltre presente una barra di stato, che ha il duplice scopo di mostrare il contenuto delle celle di memoria e dei registri quando vengono selezionati e di notificare all'utente che il simulatore è in esecuzione quando la simulazione è stata avviata ma la modalità verbose non è stata attivata. Maggiori dettagli sono descritti nelle sezioni a seguire.

4.1 La barra dei menù

La barra del menù contiene sei opzioni:

4.1.1 File

Il menù File contiene comandi per l'apertura dei file, per resettare o fermare il simulatore e per scrivere i trace file.

- *Apri...* Apre una finestra che consente all'utente di scegliere un file sorgente da aprire.
- *Apri recente* Mostra la lista dei file recentemente aperti dal simulatore, dalla quale l'utente può scegliere il file da aprire.
- *Resetta* Inizializza nuovamente il simulatore, mantenendo aperto il file che era stato caricato ma facendone ripartire l'esecuzione.
- *Scrivi Tracefile Dinero...* Scrive i dati di accesso alla memoria in un file, nel formato xdin.
- *Esci* Chiude il simulatore.

La voce del menù *Scrivi Tracefile Dinero...* è disponibile solo quando un file sorgente è stato completamente eseguito ed è stata già raggiunta la fine dell'esecuzione.

4.1.2 Esegui

Il menu Esegui contiene voci riguardanti il flusso di esecuzione della simulazione.

- *Ciclo singolo* Esegue un singolo passo di simulazione.

- *Completa* Inizia l'esecuzione, fermandosi quando il simulatore raggiunge una SYSCALL 0 (o equivalente) o un'istruzione di *BREAK*, oppure quando l'utente seleziona la voce Stop del menù (o preme F9).
- *Cicli multipli* Esegue un certo numero di passi di simulazione, tale valore può essere configurato attraverso la finestra di configurazione.
- *Ferma* Ferma l'esecuzione quando il simulatore è in modalità "Completa" o "Cicli multipli", come descritto precedentemente.

Il menu è disponibile solo quando è stato caricato un file sorgente e non è ancora stato raggiunto il termine della simulazione. La voce *Stop* del menù è disponibile solo in modalità "Completa" o "Cicli multipli" mode.

4.1.3 Configura

Il menu Configura fornisce l'opportunità di personalizzare l'aspetto ed il funzionamento di EduMIPS64.

- *Impostazioni...* Apre la finestra di configurazione, descritta nella prossima sezione di questo capitolo;
- *Selezione lingua* Consente di modificare la lingua usata dall'interfaccia utente. Attualmente sono supportate solo inglese ed italiano. Questa modifica riguarda ogni aspetto dell'interfaccia grafica, dal titolo delle finestre al manuale in linea ed i messaggi di errore o le notifiche.

La voce di menù *Impostazioni...* non è disponibile quando il simulatore è in modalità "Completa" o "Cicli multipli", a causa di possibili race conditions.

4.1.4 Strumenti

Questo menù contiene solo una voce, utilizzata per aprire la finestra del Dinero frontend.

- *Dinero Frontend...* Apre la finestra del Dinero Frontend..

Questo menù non è disponibile finchè non è stata portata a termine l'esecuzione del programma

4.1.5 Finestra

Questo menù contiene voci relative alle operazioni con le finestre.

- *Tile* Ordina le finestre visibili in modo tale che non vi siano più di tre finestre in una riga, tentando di massimizzare lo spazio occupato da ciascuna finestra.

Le altre voci del menù modificano semplicemente lo stato di ciascuna finestra, rendendola visibile o riducendola ad icona.

4.1.6 Aiuto

Questo menù contiene voci relative all'aiuto in linea.

- *Manuale...* Mostra la finestra di help.
- *Informazioni su...* Mostra una finestra contenente i nomi di coloro che hanno collaborato al progetto ed i loro ruoli.

4.2 Finestre

L'interfaccia grafica è composta da sette finestre, sei delle quali sono visibili per default, mentre una (la finestra di I/O) è nascosta.

4.2.1 Cicli

La finestra Cicli mostra l'evoluzione del flusso di esecuzione nel tempo, visualizzando in ogni istante quali istruzioni sono nella pipeline, ed in quale stadio si trovano.

4.2.2 Registri

La finestra Registri mostra il contenuto di ciascun registro. Mediante un click col tasto sinistro del mouse è possibile vedere il loro valore decimale (con segno) nella barra di stato, mentre con un doppio click verrà aperta una finestra di dialogo che consentirà all'utente di cambiare il valore del registro

4.2.3 Statistics

La finestra Statistiche mostra alcune statistiche riguardanti l'esecuzione del programma.

4.2.4 Pipeline

La finestra Pipeline mostra lo stato attuale della pipeline, visualizzando ciascuna istruzione con il suo stadio. I differenti colori evidenziano i vari stadi della pipeline stessa.

4.2.5 Memoria

La finestra Memoria mostra il contenuto delle celle di memoria, insieme alle etichette ed i commenti, tratti dal codice sorgente. Il contenuto delle celle di memoria, come per i registri, può essere modificato con un doppio click, e mediante un singolo click del mouse verrà mostrato il loro valore decimale nella barra di stato. La prima colonna mostra l'indirizzo esadecimale della cella di memoria, e la seconda il valore della cella stessa. Le altre colonne mostrano invece informazioni aggiuntive provenienti dal codice sorgente.

4.2.6 Codice

La finestra Codice visualizza le istruzioni caricate in memoria. La prima colonna mostra l'indirizzo dell'istruzione, mentre la seconda mostra la rappresentazione esadecimale dell'istruzione stessa. Le altre colonne mostrano infine informazioni aggiuntive provenienti dal codice sorgente.

4.2.7 Input/Output

La finestra Input/Output fornisce un'interfaccia all'utente per la visualizzazione dell'output creato dai programmi mediante le SYSCALL 4 e 5. Attualmente non è utilizzata per l'input di dati, ed al suo posto viene utilizzata una finestra di dialogo che viene mostrata quando una SYSCALL 3 tenta di leggere dallo standard input, ma future versioni includeranno una casella di testo per l'input.

4.3 Finestre di dialogo

Le finestre di dialogo sono utilizzate da EduMIPS64 per interagire con l'utente in vari modi. Ecco un riassunto delle più importanti:

4.3.1 Impostazioni

Nella finestra di configurazione possono essere configurati vari aspetti del simulatore. La selezione del tasto "OK" causa il salvataggio delle modifiche apportate alla configurazione, mentre il tasto "Cancel" (o la semplice chiusura della finestra) comporta il mancato salvataggio delle stesse.

La sezione "Impostazioni generali" consente di configurare il forwarding ed il numero di passi da effettuare nella modalità Cicli multipli.

La sezione "Comportamento" permette di abilitare o disabilitare gli avvisi durante la fase di parsing, l'opzione "sincronizza la GUI con la CPU nell'esecuzione multi step", quando abilitata, sincronizzerà lo stato grafico delle finestre con lo stato interno del simulatore. Ciò implicherà una simulazione più lenta, ma con la possibilità di avere un resoconto grafico esplicito di ciò che sta avvenendo durante la simulazione. L'opzione "intervallo tra i cicli", qualora sia abilitata, influenzerà il numero di millisecondi che il simulatore dovrà attendere prima di cominciare un nuovo ciclo. Tali opzioni hanno effetto solo quando la simulazione è avviata utilizzando le opzioni "Completa" o "Cicli multipli" dal menu Esegui.

Le ultime due opzioni stabiliscono il comportamento del simulatore quando si verifica un'eccezione sincrona. È importante notare che se le eccezioni sincrone sono mascherate, non succederà nulla, anche se l'opzione "Termina se si verifica un'eccezione sincrona" è abilitata. Se le eccezioni non sono mascherate e tale opzione è abilitata, apparirà una finestra di dialogo, e la simulazione sarà fermata non appena tale finestra verrà chiusa.

L'ultima sezione permette di modificare i colori associati ai diversi stadi della pipeline. Abbastanza inutile, ma carino.

4.3.2 Dinero Frontend

La finestra di dialogo Dinero Frontend consente di avviare un processo DineroIV con il trace file generato internamente mediante l'esecuzione del programma. Nella prima casella di testo c'è il percorso dell'eseguibile DineroIV, e nella seconda devono essere inseriti i parametri opportuni.

La sezione più in basso contiene l'output del processo DineroIV, dal quale è possibile prelevare i dati di cui si necessita.

4.3.3 Aiuto

La finestra di Aiuto contiene tre sezioni con qualche indicazione riguardo l'utilizzo del simulatore. La prima è una breve introduzione ad EduMIPS64, la seconda contiene informazioni riguardanti l'interfaccia grafica e la terza contiene un riassunto delle istruzioni supportate.

4.4 Opzioni da riga di comando

Sono disponibili tre opzioni da linea di comando. Esse sono descritte di seguito, con il nome per esteso scritto tra parentesi. Nomi abbreviati e per esteso possono essere utilizzati indifferentemente.

- *-v* (*-version*) stampa la versione del simulatore ed esce.
- *-h* (*-help*) mostra un messaggio di aiuto per le opzioni da linea di comando ed esce.
- *-f* (*-file*) *filename* apre *filename* nel simulatore.

- *-r* (*-reset*) ripristina i valori predefiniti per tutti i parametri di configurazione
- *-d* (*-debug*) attiva la modalità di debugging.

Nella modalità di debugging è disponibile una nuova finestra, la finestra Debug, che mostra il resoconto delle attività interne di EduMIPS64. Tale finestra non è utile per l'utente finale, è stata infatti ideata per poter essere utilizzata dagli sviluppatori di EduMIPS64.

4.5 Eseguire EduMIPS64

Il file *.jar* di EduMIPS64 può funzionare sia come file *.jar* eseguibile che come applet, quindi può essere eseguito in entrambi i modi, che richiedono il Java Runtime Environment, versione 5 o successiva.

Per eseguire il file come applicazione a sè stante, l'eseguibile *java* deve essere avviato nel seguente modo: *java -jar edumips64-version.jar*, dove la stringa *version* deve essere sostituita con la versione attuale del simulatore. Su alcuni sistemi, potrebbe essere possibile eseguire il programma semplicemente con un click sul file *.jar*.

Per eseguire il file come applet deve essere utilizzato il tag *<applet>*. Il sito web di EduMIPS64 presenta una pagina già contenente l'applet, in modo tale che chiunque possa eseguire il programma senza il problema dell'utilizzo da linea di comando.

Listati di esempio

In questo capitolo sono presenti degli esempi di codice utili per comprendere il funzionamento del simulatore.

5.1 SYSCALL

Gli esempi per le SYSCALL 1-4 si riferiscono al file *print.s*, che è l'esempio per la SYSCALL 5. Se si desidera eseguire gli esempi, è prima necessario copiare il contenuto di quell'esempio in un file denominato *print.s*, e salvarlo nella stessa directory contenente l'esempio che si sta eseguendo.

Alcuni esempi si aspettano che esista un file descriptor, e non contengono il codice per aprire alcun file. Per eseguire questi esempi, eseguire prima la SYSCALL 1.

5.1.1 SYSCALL 0

L'effetto dell'esecuzione della SYSCALL 0 è l'interruzione dell'esecuzione del programma. Esempio:

```
.code
daddi    r1, r0, 0      ; salva il valore 0 in R1
syscall 0                ; termina l'esecuzione
```

5.1.2 SYSCALL 1

Programma d'esempio che apre un file:

```
                .data
error_op:       .asciiz  "Errore durante l'apertura del file"
ok_message:     .asciiz  "Tutto ok."
params_sys1:    .asciiz  "filename.txt"
                .word64   0xF

                .text
open:           daddi     r14, r0, params_sys1
                syscall   1
                daddi     $s0, r0, -1
                dadd      $s2, r0, r1
                daddi     $a0, r0, ok_message
```

```

                bne      r1,$s0,end
                daddi    $a0,r0,error_op

end:            jal      print_string
                syscall  0

                #include print.s

```

Nelle prime due righe, vengono salvate in memoria le stringhe che contengono i messaggi di errore e di successo, che saranno poi passati come parametri alla funzione *print_string*, ed a ciascuno di essi viene associata un'etichetta. La funzione *print_string* è presente nel file *print.s*.

Successivamente, vengono salvati in memoria i dati richiesti dalla SYSCALL 1, il percorso del file da aprire (che deve esistere se si apre il file in modalità sola lettura o lettura/scrittura) e, nella cella successiva, un intero che definisce la modalità di apertura.

In questo esempio, il file è stato aperto utilizzando la seguente modalità: *O_RDWR|O_CREAT|O_APPEND*. Il numero 15 (0xF in base 16) deriva dalla somma dei valori di queste tre modalità modes (3 + 4 + 8).

Questi due parametri hanno un'etichetta, in modo che in seguito possano essere utilizzati.

Nella sezione *.text*, come prima cosa l'indirizzo di *param_sys1* - che per il compilatore è un numero - viene salvato in *r14*; successivamente viene chiamata la SYSCALL 1, ed il contenuto di *R1* viene salvato nel registro *\$s2*, in modo che possa essere utilizzato nel resto del programma (ad esempio, con un'altra SYSCALL).

Infine viene chiamata la funzione *print_string*, passando come parametro *error_op* se *R1* contiene il valore -1 (righe 13-14), altrimenti utilizzando *ok_message* (righe 12-16).

5.1.3 SYSCALL 2

Programma di esempio che chiude un file:

```

                .data
params_sys2:    .space 8
error_cl:      .asciiz  "Errore durante la chiusura del file"
ok_message:    .asciiz  "Tutto a posto"

                .text
close:          daddi    r14, r0, params_sys2
                sw       $s2, params_sys2(r0)
                syscall  2
                daddi    $s0, r0, -1
                daddi    $a0, r0, ok_message
                bne      r1, $s0, end
                daddi    $a0, r0, error_cl

end:            jal      print_string
                syscall  0

                #include print.s

```

Nota: Questo esempio richiede che in *\$s2* ci sia il file descriptor del file da chiudere.

Come prima cosa viene allocata della memoria per l'unico parametro di SYSCALL 2, il file descriptor del file da chiudere, e a questo spazio viene associata un'etichetta in modo da potervicisi riferire successivamente.

Successivamente vengono salvate in memoria le stringhe contenenti i messaggi di successo e di errore.

Nella sezione *.text*, l'indirizzo di *param_sys2* viene salvato in *R14*; successivamente viene chiamata la SYSCALL 2.

Infine viene chiamata la funzione *print_string*, stampando il messaggio d'errore se ci sono problemi (riga 13) o, se tutto è andato a buon fine, il messaggio di successo (riga 11).

5.1.4 SYSCALL 3

Programma di esempio che legge 16 byte da un file e li salva in memoria:

```

        .data
params_sys3: .space      8
ind_value:   .space      8
             .word64     16
error_3:     .asciiz      "Errore durante la lettura da file."
ok_message:  .asciiz      "Tutto ok."

value:       .space      30

        .text
read:      daddi         r14, r0, params_sys3
           sw           $s2, params_sys3(r0)
           daddi        $s1, r0, value
           sw           $s1, ind_value(r0)
           syscall      3
           daddi        $s0, r0, -1
           daddi        $a0, r0, ok_message
           bne          r1, $s0, end
           daddi        $a0, r0, error_3

end:       jal          print_string
           syscall      0

#include   print.s

```

Le prime 4 righe della sezione *.data* contengono i parametri della SYSCALL 3, il file descriptor da cui si devono leggere i dati, l'indirizzo della cella di memoria dove la SYSCALL deve salvare i dati letti, il numero di byte da leggere. Successivamente sono presenti in memoria i messaggi di successo e di errore.

Nella sezione *.text*, come prima cosa viene salvato l'indirizzo di *param_sys3* in *r14*, il file descriptor viene salvato nell'area di memoria dedicata ai parametri della SYSCALL, ed a seguire lo stesso destino tocca all'indirizzo dell'area di memoria adibita a contenere i dati letti.

Successivamente viene chiamata la SYSCALL 3 e viene stampato un messaggio di successo o di errore, a seconda dell'esito della SYSCALL.

5.1.5 SYSCALL 4

Programma di esempio che scrive su file una stringa:

```

        .data
params_sys4: .space      8
ind_value:   .space      8
             .word64     16
error_4:     .asciiz      "Errore durante la scrittura su stringa."
ok_message:  .asciiz      "Tutto ok."
value:       .space      30

        .text

```

```
write:      daddi      r14, r0,params_sys4
            sw         $s2, params_sys4(r0)
            daddi      $s1, r0,value
            sw         $s1, ind_value(r0)
            syscall    4
            daddi      $s0, r0,-1
            daddi      $a0, r0,ok_message
            bne        r1, $s0,end
            daddi      $a0, r0,error_4

end:        jal        print_string
            syscall    0

            #include   print.s
```

La struttura di quest'esempio è identica a quella dell'esempio di SYSCALL 3.

5.1.6 SYSCALL 5

Programma di esempio che contiene una funzione che stampa su standard output la stringa contenuta nell'indirizzo di memoria a cui punta \$a0:

```
            .data
params_sys5: .space 8

            .text
print_string:
            sw         $a0, params_sys5(r0)
            daddi      r14, r0, params_sys5
            syscall    5
            jr         r31
```

La seconda riga alloca spazio per la stringa che sarà stampata dalla SYSCALL, che è riempito dalla prima istruzione della sezione .text, che assume che l'indirizzo della stringa da stampare sia in \$a0.

L'istruzione successiva salva in r14 l'indirizzo di questa stringa, e successivamente la SYSCALL 5 viene chiamata, stampando quindi la stringa. L'ultima istruzione varia il program counter, impostandolo al valore di r31 - che secondo le convenzioni di chiamata di funzione MIPS contiene l'indirizzo dell'istruzione successiva alla chiamata di funzione.

5.1.7 Un esempio di utilizzo della SYSCALL 5 più complesso

La SYSCALL 5 utilizza un meccanismo di passaggio parametri non semplicissimo, che sarà illustrato nel seguente esempio:

```
            .data
format_str: .asciiiz  "%d %s:\nTest di %s versione %i.%i!"
s1:         .asciiiz  "Giugno"
s2:         .asciiiz  "EduMIPS64"
fs_addr:    .space    4
            .word      5
s1_addr:    .space    4
s2_addr:    .space    4
            .word      0
            .word      5

test:       .code
```

```
daddi    r5, r0, format_str
sw       r5, fs_addr(r0)
daddi    r2, r0, s1
daddi    r3, r0, s2
sd       r2, s1_addr(r0)
sd       r3, s2_addr(r0)
daddi    r14, r0, fs_addr
syscall  5
syscall  0
```

L'indirizzo di memoria della stringa di formato viene inserito in R5, il cui contenuto viene quindi salvato in memoria all'indirizzo *fs_addr*. Gli indirizzi dei parametri di tipo stringa sono salvati in *s1_addr* ed *s2_addr*. Questi due parametri saranno inseriti al posto dei due segnaposto *%s* all'interno della stringa di formato.

Nel caso di stringhe di formato complesse, come mostrato da questo esempio, le word che corrispondono ai segnaposto vanno inserite in memoria subito dopo l'indirizzo della stringa di formato.