Edge detector CPU

Realizzazione di una CPU per l'estrazione contorni da una immagine

Gruppo di lavoro:

Giorgio Lanzi

Mauro Laurenti

Michele Marino

Tommaso Villani

Esame di sistemi digitali

Prof. Mauro Olivieri

Università "La Sapienza" Roma

A.A. 2003/2004

Realizzazione hardware/VHDL

L'edge detector *CPU* (*Central Processing Unit*) è un processore la cui funzione è quella di estrarre i contorni di un'immagine presente in una memoria esterna al dispositivo e memorizzare la nuova immagine rappresentante i contorni nella memoria stessa.

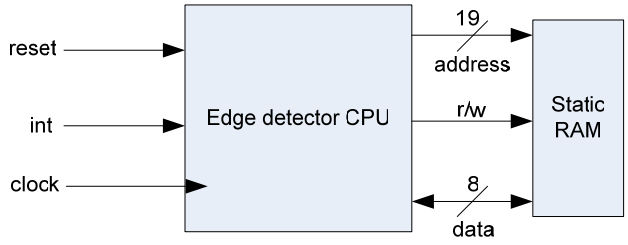


figura 1 – Segnali di ingresso e di uscita della *CPU* e relativo collegamento con la *RAM* esterna

Come si può vedere dalla figura 1, la *CPU* possiede un segnale di reset, un segnale di interrupt mediante il quale viene avviata la procedura di estrazione dei contorni e un segnale di clock per la sincronizzazione delle azioni. Le immagini che il dispositivo può trattare hanno una dimensione di *512x512* pixel, con una codifica di colore di 8 bit per pixel (scala di grigio). La memoria esterna è organizzata per righe e quindi ciascuna riga rappresenta un pixel (8 bit). L'immagine sorgente e quella di destinazione sono memorizzate consecutivamente e, poiché ogni immagine richiede *512*512=262144* celle di memoria a 8 bit, la memoria dovrà avere *512*512*2=524288* celle di memoria e quindi un *BUS* di indirizzamento a *19* bit, come illustrato in figura 2.

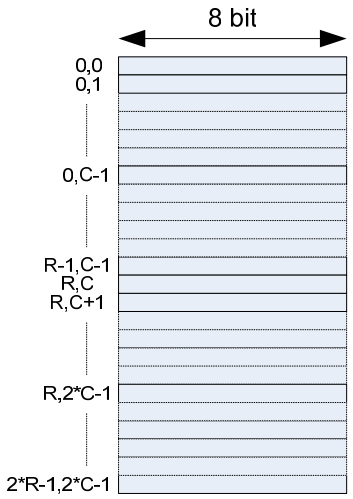


figura 2 – Organizzazione della memoria *RAM* esterna

Il *BUS* di comunicazione dati con la memoria è ad 8 bit poiché ogni pixel è codificato con 8 bit. Inoltre è presente il segnale *r/w* che consente di leggere un byte presente ad un certo indirizzo quando è a livello logico basso, di scriverlo in caso contrario.

La scansione dell'immagine sorgente avviene per righe per cui vengono letti due pixel adiacenti controllando se la differenza di luminosità supera una soglia prefissata, distinguendo due casi:

- Se la differenza è positiva, viene annerito il pixel corrente in quanto siamo nel caso in cui ha inizio un contorno.
- Se la differenza è negativa, viene annerito il pixel successivo in quanto siamo nel caso in cui finisce un contorno.

Per implementare l'algoritmo di estrazione contorni utilizziamo un set di 8 istruzioni (*ISA – Instruction Set Architecture*) da 16 bit suddivise in quattro classi:

Operazioni di ALU	
ADD	*r r r*
SUB	*r r r*
Operazioni di branch	
BRGT	*r r r*
RETURN	-
Operazioni memoria	
LOADB	*r r*
STORB	*r #*
Operazioni di trasferimento	
MOVE	*r #*
SHIFTL	*r r #*

Il primo registro a sinistra per tutte le istruzioni, tranne per la *BRGT*, rappresenta il registro di destinazione dell'operazione eseguita. La codifica delle istruzioni viene riportata nella figura 3.

L'istruzione di salto condizionato (*BRGT – Branch If Greather Than*) viene gestita caricando prima dell'istruzione di salto, il nuovo indirizzo del *Program Counter* (*PC*)nel registro 14 del register file attraverso l'istruzione *MOVE*. Se il salto viene preso, viene caricato il *Program Counter* con il valore presente nel registro 14 del register file, altrimenti viene incrementato di 1 e si continua l'esecuzione sequenziale del codice. Quindi il blocco di 4 bit contrassegnato da "*PC reg value*" fa riferimento al valore binario 1110 che rappresenta il registro 14 del register file.

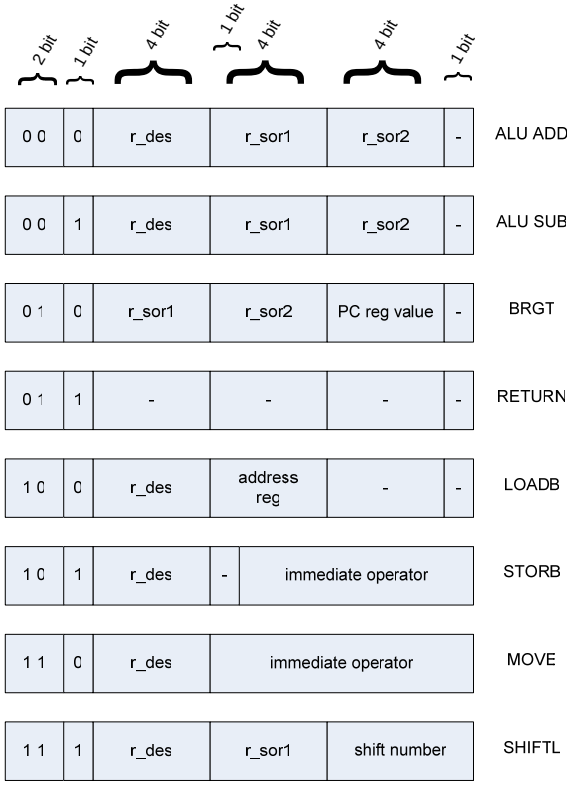


figura 3 – Instruction set architecture dell'edge detector *CPU*

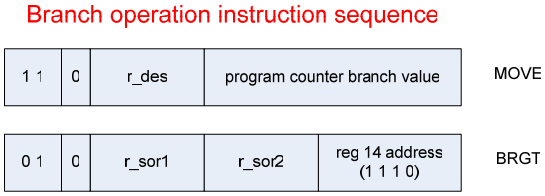


figura 4 – Gestione dell'istruzione di salto condizionato

Il programma risiede in una memoria *ROM* interna alla *CPU* con un *BUS* di indizzamento e dati ad 8 bit per cui ogni istruzione occupa due locazioni di memoria. Ogni istruzione viene letta nel seguente modo:

- Con il primo accesso nella *ROM* vengono letti gli 8 bit meno significativi e posti nella parte bassa del registro istruzione a 16 bit (*IR – Instruction Register*).
- Con il secondo accesso nella *ROM* vengono letti gli 8 bit più significativi e posti nella parte alta dell'*IR* ottenendo così l'istruzione completa a 16 bit.

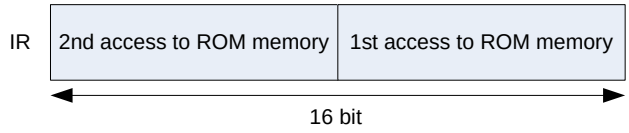
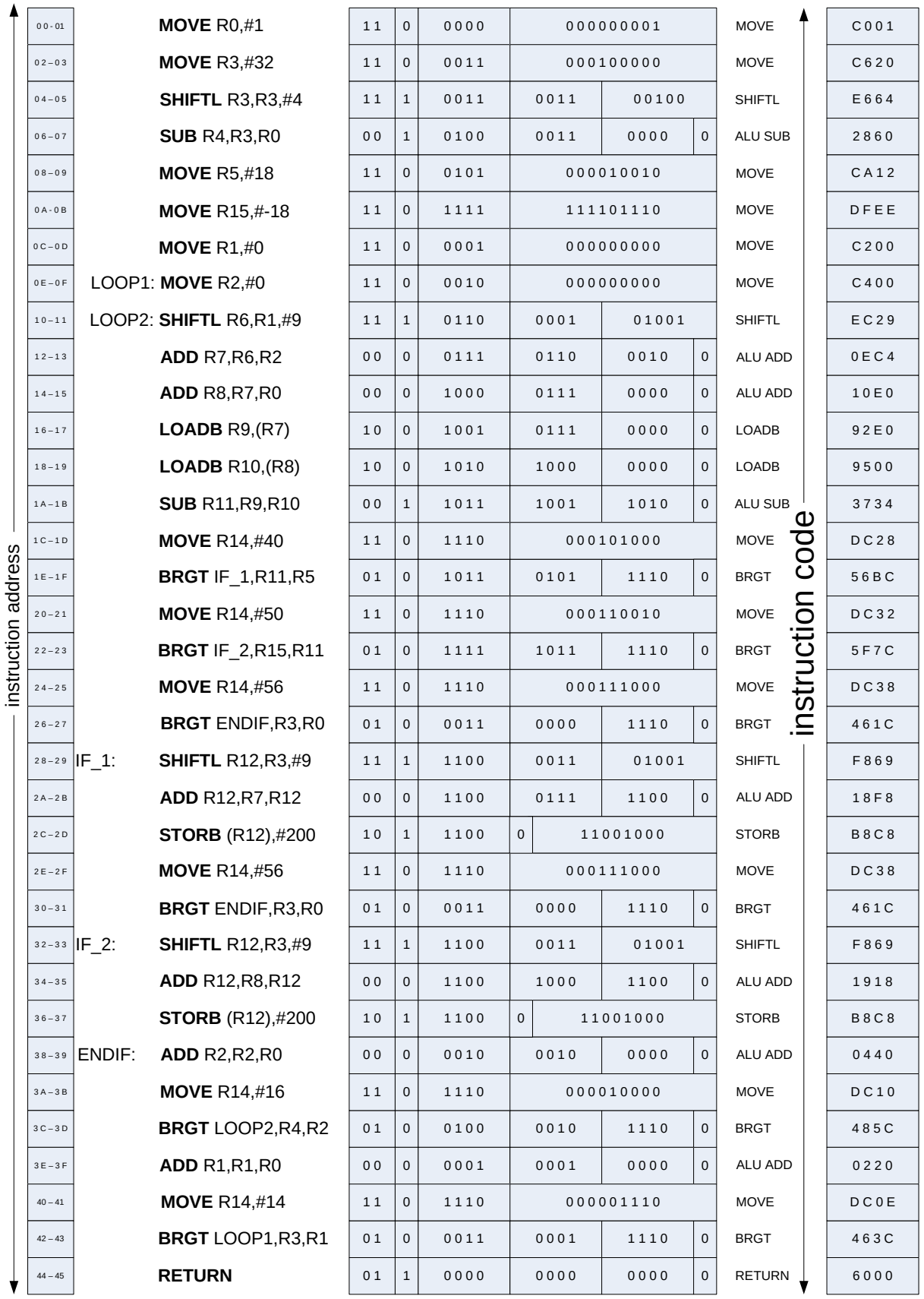


figura 5 – Caricamento del registro istruzione a 16 bit

Nella figura 6 è riportato il listato assembler del programma che il processore deve eseguire.



instruction address										instruction code	hex code
00-01		**MOVE** R0,#1	11	0	0000	000000001			MOVE		C001
02-03		**MOVE** R3,#32	11	0	0011	000100000			MOVE		C620
04-05		**SHIFTL** R3,R3,#4	11	1	0011	0011	00100		SHIFTL		E664
06-07		**SUB** R4,R3,R0	00	1	0100	0011	0000	0	ALU SUB		2860
08-09		**MOVE** R5,#18	11	0	0101	000010010			MOVE		CA12
0A-0B		**MOVE** R15,#-18	11	0	1111	111101110			MOVE		DFEE
0C-0D		**MOVE** R1,#0	11	0	0001	000000000			MOVE		C200
0E-0F	LOOP1:	**MOVE** R2,#0	11	0	0010	000000000			MOVE		C400
10-11	LOOP2:	**SHIFTL** R6,R1,#9	11	1	0110	0001	01001		SHIFTL		EC29
12-13		**ADD** R7,R6,R2	00	0	0111	0110	0010	0	ALU ADD		0EC4
14-15		**ADD** R8,R7,R0	00	0	1000	0111	0000	0	ALU ADD		10E0
16-17		**LOADB** R9,(R7)	10	0	1001	0111	0000	0	LOADB		92E0
18-19		**LOADB** R10,(R8)	10	0	1010	1000	0000	0	LOADB		9500
1A-1B		**SUB** R11,R9,R10	00	1	1011	1001	1010	0	ALU SUB		3734
1C-1D		**MOVE** R14,#40	11	0	1110	000101000			MOVE		DC28
1E-1F		**BRGT** IF_1,R11,R5	01	0	1011	0101	1110	0	BRGT		56BC
20-21		**MOVE** R14,#50	11	0	1110	000110010			MOVE		DC32
22-23		**BRGT** IF_2,R15,R11	01	0	1111	1011	1110	0	BRGT		5F7C
24-25		**MOVE** R14,#56	11	0	1110	000111000			MOVE		DC38
26-27		**BRGT** ENDIF,R3,R0	01	0	0011	0000	1110	0	BRGT		461C
28-29	IF_1:	**SHIFTL** R12,R3,#9	11	1	1100	0011	01001		SHIFTL		F869
2A-2B		**ADD** R12,R7,R12	00	0	1100	0111	1100	0	ALU ADD		18F8
2C-2D		**STORB** (R12),#200	10	1	1100	0	11001000		STORB		B8C8
2E-2F		**MOVE** R14,#56	11	0	1110	000111000			MOVE		DC38
30-31		**BRGT** ENDIF,R3,R0	01	0	0011	0000	1110	0	BRGT		461C
32-33	IF_2:	**SHIFTL** R12,R3,#9	11	1	1100	0011	01001		SHIFTL		F869
34-35		**ADD** R12,R8,R12	00	0	1100	1000	1100	0	ALU ADD		1918
36-37		**STORB** (R12),#200	10	1	1100	0	11001000		STORB		B8C8
38-39	ENDIF:	**ADD** R2,R2,R0	00	0	0010	0010	0000	0	ALU ADD		0440
3A-3B		**MOVE** R14,#16	11	0	1110	000010000			MOVE		DC10
3C-3D		**BRGT** LOOP2,R4,R2	01	0	0100	0010	1110	0	BRGT		485C
3E-3F		**ADD** R1,R1,R0	00	0	0001	0001	0000	0	ALU ADD		0220
40-41		**MOVE** R14,#14	11	0	1110	000001110			MOVE		DC0E
42-43		**BRGT** LOOP1,R3,R1	01	0	0011	0001	1110	0	BRGT		463C
44-45		**RETURN**	01	1	0000	0000	0000	0	RETURN		6000

figura 6 – Codice assembler del programma di estrazione contorni

A questo punto possiamo passare alla specifica *RTL* (*Register Transfer Level*) attraverso un diagramma *ASM*, riportato nella figura 7.

Come si può vedere, la *CPU* inizialmente rimane in attesa di una interrupt: se *int_req* è zero non viene eseguita nessuna operazione, ma non appena questo diventa uno, ha inizio l'esecuzione della sequenza di operazioni. La prima operazione riguarda il caricamento del registro *rom_mar* (*ROM Memory Address Register*) che contiene l'indirizzo della locazione della *ROM* da cui leggere ovvero la parte bassa dell'istruzione (8 bit meno significativi). Successivamente viene trasferito il byte letto in registro ausiliario (a 8 bit) chiamato temp e nello stesso tempo viene incrementato di 1 l'indirizzo della *ROM* puntanto così al byte successivo. A questo punto vengono letti gli 8 bit più significativi dell'istruzione e vengono concatenati nel registro *IR* con gli 8 bit meno significati presenti nel registro *temp* ottenendo così l'istruzione completa a 16 bit all'interno del registro istruzione *IR*. Il passo successivo consiste nel riconoscere la classe dell'istruzione letta attraverso un controllo sui bit 15 e 14 del registro istruzione abilitando i relativi segnali di controllo.

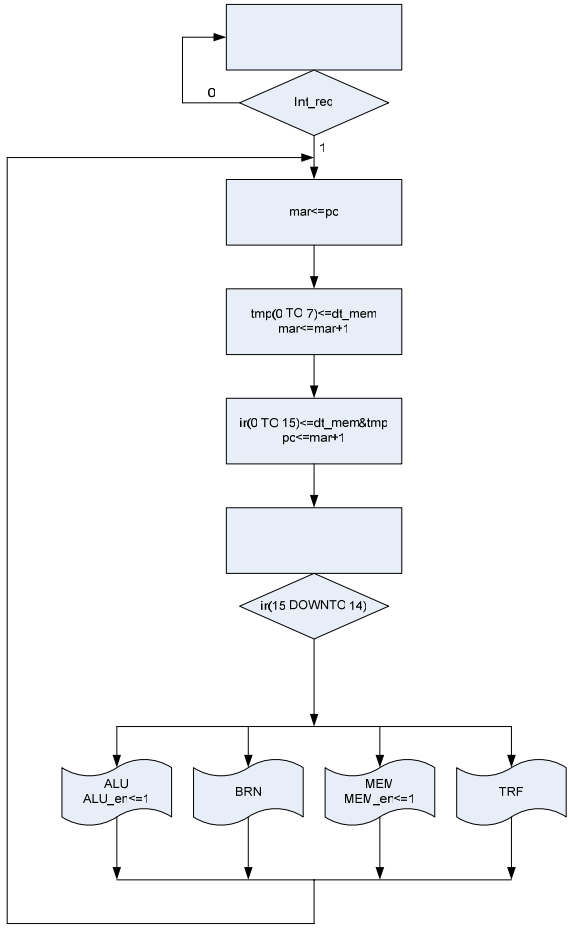


figura 7 – Diagramma *ASM* dell'edge detector *CPU*

In relazione ai segnali di abilitazione vengono eseguite le varie operazioni, i cui digrammi *ASM* sono riportati qui di seguito. Se viene asserito il segnale *alu_en* si tratta di una operazione di *ALU* per cui, attraverso l'unità di decodifica dell'istruzione (*ID – Instruction decode*) vengono decodificati gli indirizzi degli operandi. A questo punto viene eseguito un controllo sul bit 13 del registro istruzione al fine di distinguere le istruzioni all'interno della classe corrispondente: se il valore è 0 viene eseguita un'addizione, se il valore è 1 viene eseguita una sottrazione.

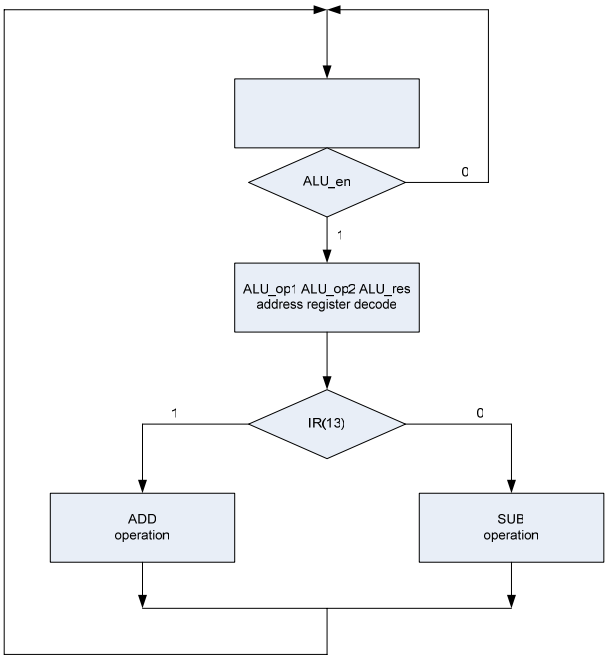


figura 8 – Diagramma *ASM* relativo alle istruzioni *ADD* e *SUB*

Se l'istruzione riguarda un'operazione di trasferimento[1], anche qui si effettua un controllo sul bit 13 del registro istruzione: se vale 0 viene eseguita l'istruzione *MOVE* e in particolare viene decodificato il registro di destinazione *r_des*, il valore immediato a 9 bit identificato da *imm_op*, dopodichè questo viene trasferito nel registro di destinazione; se vale 1 viene eseguita l'istruzione *SHIFTL*. Viene decodificato l'indirizzo di destinazione *r_des*, quello sorgente *r_sor* e il numero di shift da eseguire sul valore presente nel registro *r_sor* (nel nostro caso 9) identificato dal blocco a 5 bit "shift number". A questo punto viene "shiftato" il valore del registro *r_sor* e il risultato viene trasferito nel registro *r_des*.

[1] Si tenga presente che l'istruzione *SHIFTL* è in realtà un'operazione di *ALU*. Per poter suddividere le istruzioni in quattro classi e quindi per semplicità, consideriamo quest'ultima come un'operazione di trasferimento.

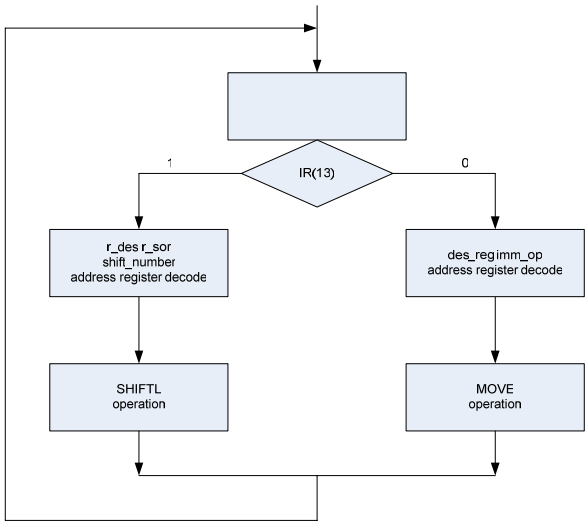


figura 9 – Diagramma *ASM* relativo alle istruzioni *SHIFTL* e *MOVE*

Se l'istruzione è un'operazione di salto e, se il bit 13 del registro istruzione è 1, si esegue l'istruzione *RETURN* caricando il valore zero nel *PC* al fine di puntare alla prima istruzione da eseguire. Quindi in queste condizioni si riporta la *CPU* ad attendere un nuovo impulso sul segnale di interrupt al fine di avviare nuovamente l'esecuzione del programma. Se il tredicesimo bit del registro istruzione vale 0 viene eseguita l'istruzione di salto condizionato *BRGT*. In particolare, vengono prima decodificati gli indirizzi *r_sor1*, *r_sor2* e vengono confrontati tra loro: se *r_sor1*>*r_sor2* viene caricato nel *PC* l'indirizzo dell'etichetta alla quale saltare, mentre nel caso contrario viene continuata l'esecuzione sequenziale del codice.

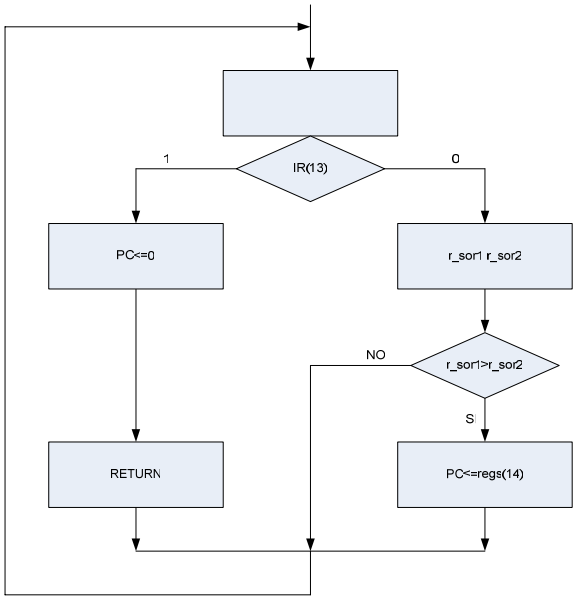


figura 10 – Diagramma *ASM* relativo alle istruzioni *RETURN* e *BRGT*

Infine, se viene asserito il segnale *mem_en* l'istruzione è un'operazione di memoria. Se il bit 13 del registro istruzione vale 0 viene eseguita l'istruzione *LOADB* che carica un byte dalla memoria esterna. In particolare vengono decodificati gli indirizzi del registro che contiene l'indirizzo della locazione di memoria alla quale puntare e del registro di destinazione che contiene il valore letto. A questo punto viene eseguita l'operazione vera e propria scrivendo il byte ricevuto dalla memoria nel registro facente riferimento all'indirizzo precedentemente decodificato. Se il tredicesimo bit del registro istruzione vale 0, viene eseguita l'istruzione *STORB* mediante la quale viene scritto un byte in memoria. Quindi viene decodificato l'indirizzo che contiene l'indirizzo della locazione di memoria nella quale scrivere e il valore dell'operatore immediato, dopodichè vengono abilitati i segnali relativi alla scrittura scrivendo il valore dell'operatore immediato (8 bit) nella locazione puntata dal relativo indirizzo precedentemente decodificato.

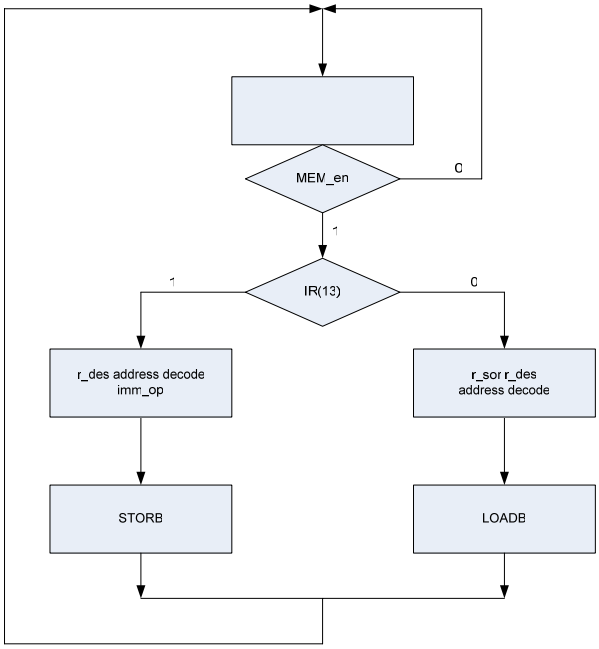


figura 11 – Diagramma *ASM* relativo alle istruzioni *LOADB* e *STORB*

Per quanto riguarda la struttura interna della *CPU*, questa è costituita da un'unità di fetch delle istruzioni, un'unità di decodifica per l'indirizzamento degli operandi, una *ALU* a 19 bit, un'unità di esecuzione, una *ROM* interna di 256 locazioni da 8 bit ciascuna e un banco di 16 registri a 19 bit. Nella figura 12 è visibile il *data path* dell'edge detector *CPU* dove si possono notare dei *BUS* interni a 19 bit per le operazioni di *ALU*, per il register file e *BUS* a 8 bit per la lettura e/o scrittura dei dati in memoria.

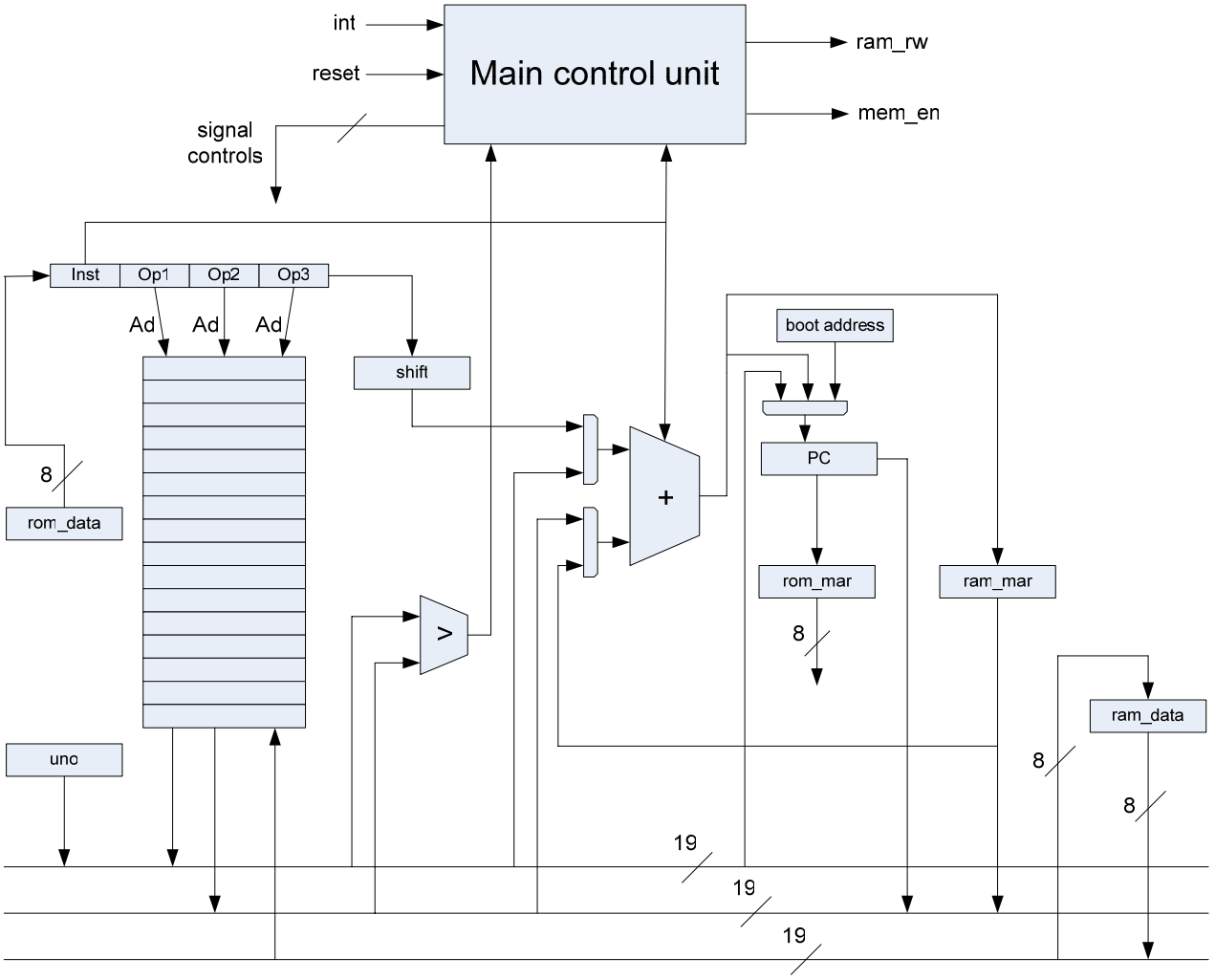


figura 12 – Data path dell'edge detector *CPU*

In definitiva il funzionamento della *CPU*, e quindi i vari passi da seguire per l'esecuzione delle istruzioni, sono i seguenti:

1.

1.1. Caricamento del *PC* nel registro rom_mar per puntare al byte basso dell'istruzione da eseguire, abilitazione del segnale di lettura della *ROM* e caricamento degli operandi nell'*ALU* (fronte di salita del clock).

1.2. Lettura del byte basso dell'istruzione dalla memoria *ROM* interna ed esecuzione dell'operazione di somma tra il registro rom_mar e il registro cablato "1" (fronte di discesa del clock).

2.

2.1. Scrittura del byte basso nel registro istruzione, scrittura del risultato dell'operazione di *ALU* precedente nel registro *rom_mar* per puntare al byte alto

dell'istruzione da eseguire, bypass del risultato dell'*ALU* sull'operando 1 e abilitazione del segnale di lettura della *ROM* (fronte di salita del clock).

2.2. Lettura del byte alto dell'istruzione dalla memoria *ROM* interna ed esecuzione della somma tra il risultato dell'operazione di *ALU* precedente (*rom_mar* + *1*) e il registro cablato "1" (fronte di discesa del clock).

3.

3.1. Caricamento del *PC* con il risultato dell'*ALU* (*rom_mar*+2 per puntare all'istruzione successiva), scrittura del byte alto nel registro istruzione e disabilitazione del segnale di lettura della *ROM* (fronte di salita del clock).

3.2. Nessuna operazione (fronte di discesa del clock).

4.

4.1. Controllo dei bit 13, 14 e 15 del registro istruzione (*IR*) per la discriminazione dell'istruzione da eseguire e decodifica degli indirizzi degli operandi (fronte di salita del clock).

4.2. Nessuna operazione (fronte di discesa del clock).

5. Esecuzione dell'istruzione:

5.1. Se è l'istruzione *MOVE*:

5.1.1. viene caricato il valore dell'operando immediato nel registro di destinazione (fronte di salita del clock).

5.1.2. nessuna operazione (fronte di discesa del clock)

5.2. Se è l'istruzione *SHIFTL*:

5.2.1. viene shiftato di n posizioni il registro sorgente e viene scritto il risultato nel registro di destinazione (fronte di salita del clock).

5.2.2. nessuna operazione (fronte di discesa del clock)

5.3. Se è un'operazione di *ALU*:

5.3.1. vengono caricati gli operandi (fronte di salita del clock)

5.3.2. esecuzione dell'operazione di *ALU* (fronte di discesa del clock)

5.3.3. scrittura del risultato nel registro di destinazione (fronte di salita del clock) – l'esecuzione di questa operazione avviene in parallelo al *fetch* dell'istruzione successiva.

5.4. Se è l'istruzione *STORB*:

5.4.1. viene scritto l'indirizzo della cella di memoria nella quale scrivere sul *BUS* di indirizzamento e il dato da memorizzare sul *BUS* dati (fronte di salita del clock)

5.4.2. nessuna operazione (fronte di discesa del clock)

5.4.3. scrittura del dato presente sul *BUS* dati nella cella di memoria indirizzata dal *BUS* di indirizzamento (fronte di salita del clock) – l'esecuzione di questa operazione avviene in parallelo al *fetch* dell'istruzione successiva.

5.5. Se è l'istruzione *LOADB*:

5.5.1. viene scritto l'indirizzo della cella di memoria dalla quale leggere il dato sul *BUS* di indirizzamento (fronte di salita del clock)

5.5.2. nessuna operazione (fronte di discesa del clock)

5.5.3. scrittura del dato letto dalla *RAM* presente sul *BUS* dati nel registro di destinazione (fronte di salita del clock) - l'esecuzione di questa operazione avviene in parallelo al *fetch* dell'istruzione successiva.

5.6. Se è l'istruzione *RETURN*:

5.6.1. viene caricato il valore 0 nel *PC* e viene riportata la *CPU* nella condizione iniziale, cioè in attesa di un segnale di interrupt (fronte di salita del clock)

5.6.2. nessuna operazione (fronte di discesa del clock)

5.7. Se è l'istruzione *BRGT*:

5.7.1. valutazione dell'operazione di confronto (fronte di salita del clock)

5.7.2. nessuna operazione (fronte di discesa del clock – *1° ciclo*)

5.7.2.1. se la condizione è vera carica l'indirizzo presente nel registro 14 nel *PC* (fronte di salita del clock)

5.7.2.2. se la condizione è falsa continua l'esecuzione sequenziale (fronte di salita del clock)

5.7.3. nessuna operazione (fronte di discesa del clock – *2° ciclo*).

E' bene precisare che la *CPU* in questione lavora su entrambi i fronti di clock. Infatti, mentre l'unità di decodifica, gli accessi nella *RAM* esterna, gli accessi ai registri avvengono sul fronte di salita del clock, l'*ALU* e l'unità di *fetch* delle istruzioni avvengono sul fronte di discesa del clock. Questo per ridurre la latenza di esecuzione delle istruzioni al fine di creare una sorta di pipeline interna (in particolare per ridurre la latenza delle istruzioni di

memoria e delle istruzioni di salto). Attraverso l'attivazione di alcuni segnali interni, la *CPU* è in grado di eseguire il *fetch* dell'istruzione successiva e allo stesso tempo portare a termine l'esecuzione dell'istruzione precedentemente avviata senza perdere ulteriori cicli di clock. Il codice *VHDL* è diviso in due file dove nel primo è possibile notare la presenza di una funzione di *shift*, un processo di controllo, un processo operativo e un processo per l'assegnazione dello stato prossimo; nel secondo file sono definite le unità di memoria *ROM* e *RAM*, l'*ALU* e la funzione per la gestione del *BUS* dati bi-direzionale a 8 bit della *RAM* esterna.

Per quanto riguarda la sintesi, il sistema viene mappato su una FPGA or2c12aM84 prodotta da Lattice semiconductor. Di seguito viene riportato il log della sintesi mediante Leonardo Spectrum.

Dispositivi utilizzati per il chip or2c12aM84			
Risorse	Utilizzate	Disponibili	Utilizzo
IOs	31	64	48.44%
LUTs	1250	1300	96.15%
PFUs	313	325	96.31%
Flipflops	441	1300	33.92%

Per ulteriori approfondimenti e particolari sulla struttura hardware è possibile visionare i commenti presenti nel codice *VHDL*.

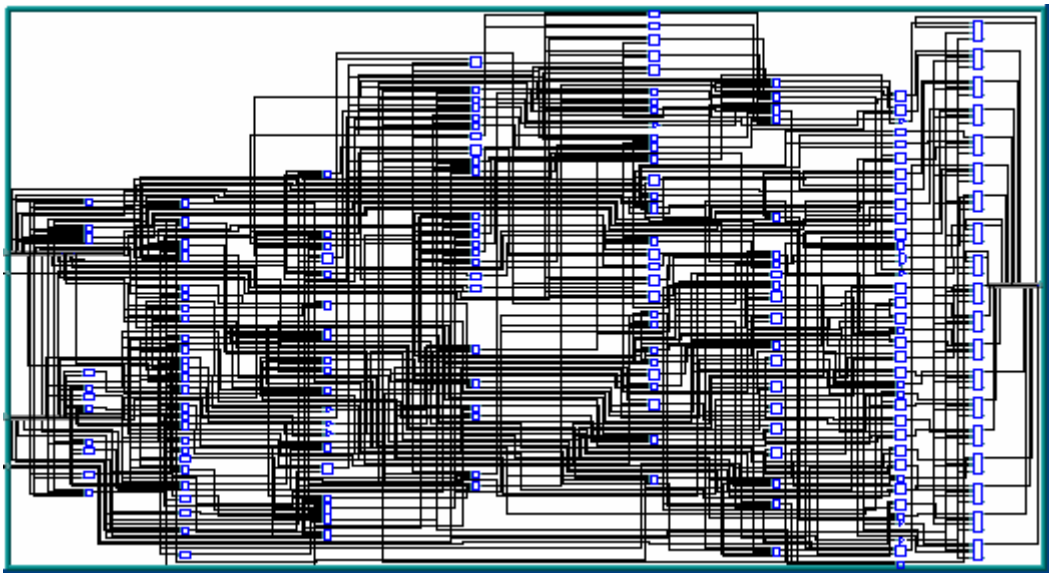


figura 13 – Realizzazione circuitale su FPGA dell'*ALU*

Nella figura 13 è riportato lo schema dell'ALU relativo alla sintesi mediante Leonardo spectrum, mentre nella figura 14 vengono riportati gli schemi dell'Edge detector, delle unità di memoria e dell'ALU.

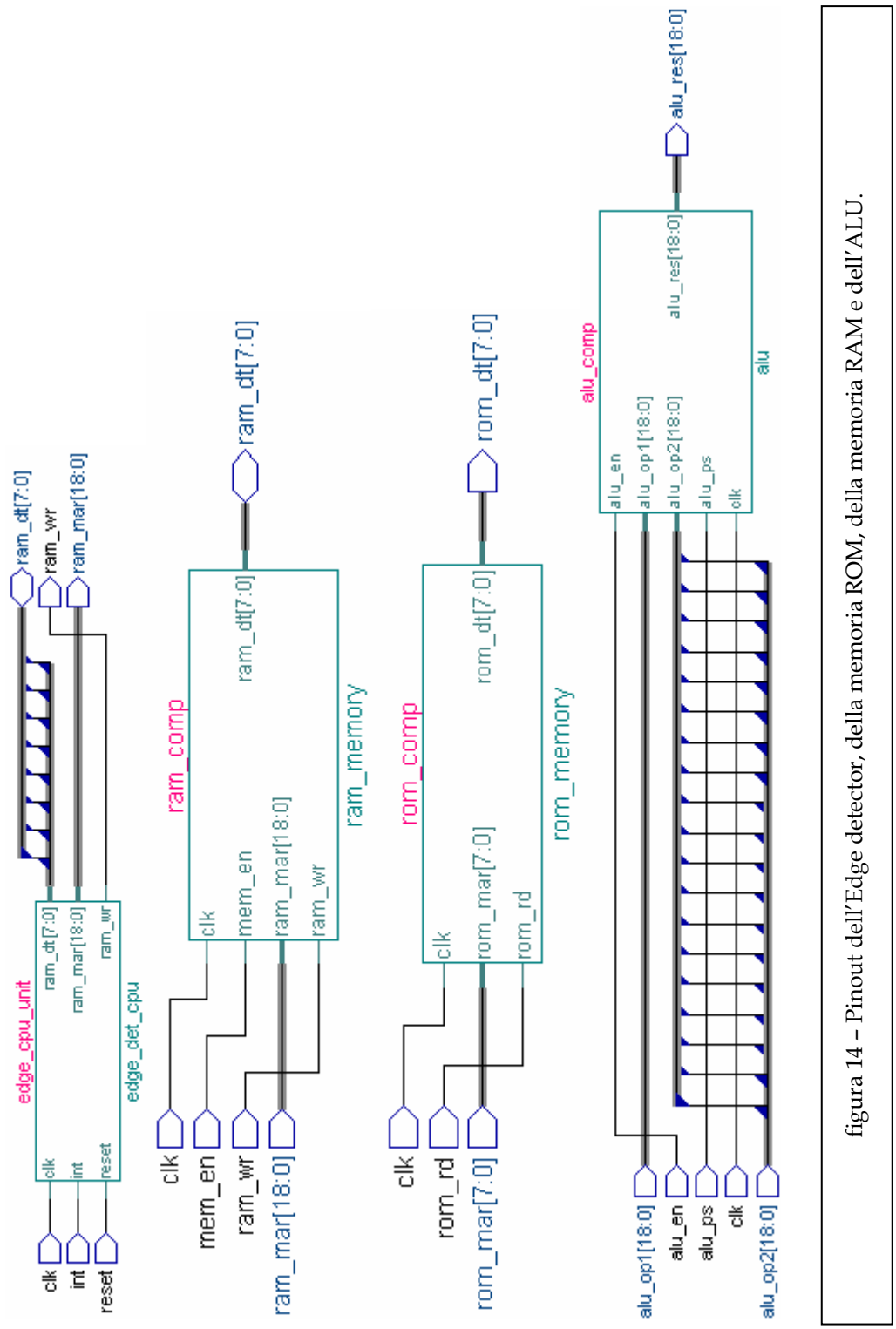


figura 14 – Pinout dell'Edge detector, della memoria ROM, della memoria RAM e dell'ALU.

AMBIENTE DI SVILUPPO

Al fine di rendere lo sviluppo delle applicazioni facente ipoteticamente uso di tale dispositivo; si è sviluppato un ambiente di sviluppo che racchiude i principali *Tools* di cui si può avere bisogno. L'ambiente di cui sopra è composto dai seguenti componenti :

- Editor *ASCII*
- Compilatore
- Simulatore
- Convertitore d'immagine

L'analisi di tale ambiente avverrà partendo dal livello di astrazione più alto in modo da permettere l'uso dello stesso senza scendere nei dettagli realizzativi, i quali verranno trattati solo in parte, lasciando ai ricchi commenti dei sorgenti la spiegazione dei dettagli.

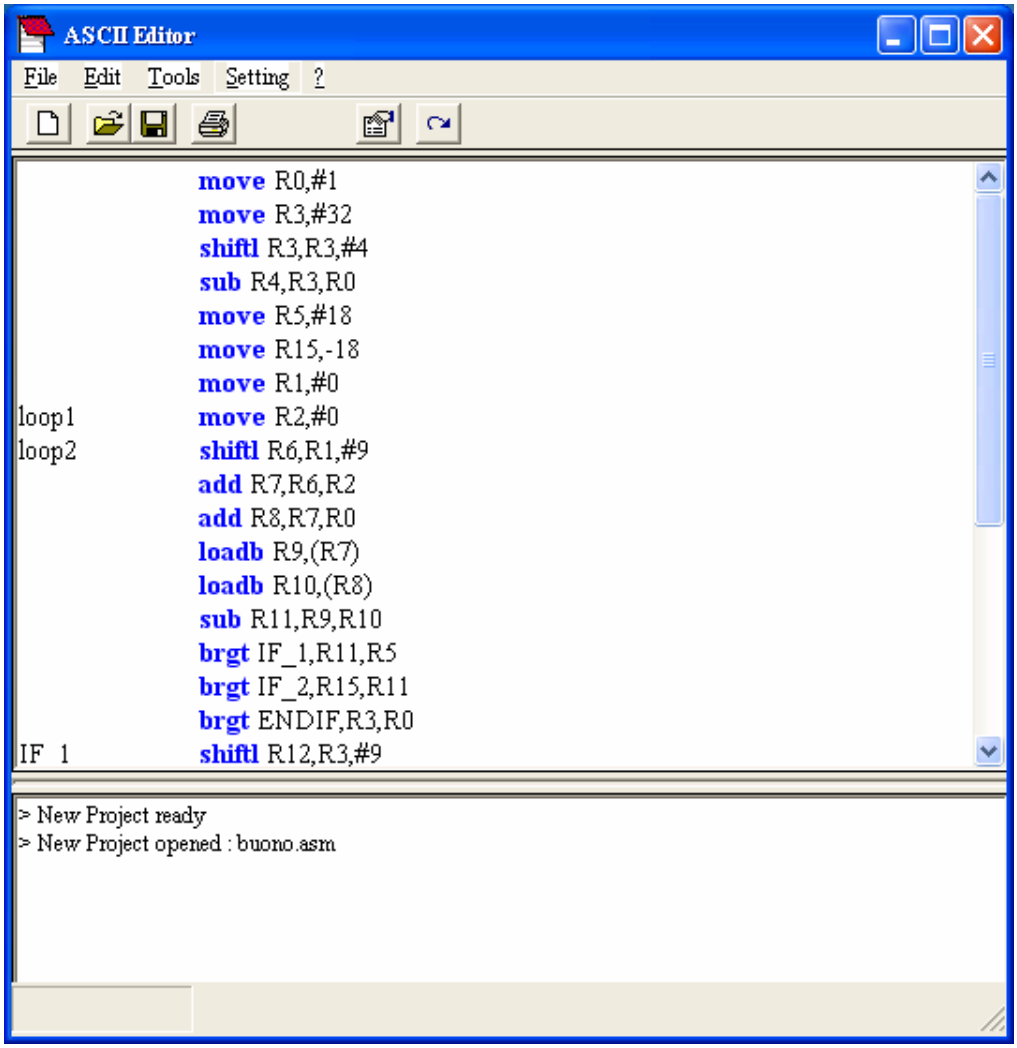


figura 15 – Schermata principale dell'editor *ASCII*

Lo sviluppo delle applicazioni avviene per mezzo di un'unica applicazione "padre" dalla quale vengono eseguiti processi "figli". L'applicazione padre è *Editor.exe* sviluppato in *Visual Basic* per sistemi operativi a 32 bit, in figura 15 è riportata la schermata principale.

È possibile subito osservare che l'editor è *Key Sensitive*; le parole chiave vengono automaticamente formattate in grassetto e di colore blu.

Dal menù *File* ed *Edit* è possibile svolgere le tipiche operazioni di sistemi per ambiente *Windows*. Per mezzo del menù *Setting* → *Path* è possibile impostare i percorsi relativi alle applicazioni che l'editor può richiamare[2] (figura 16).

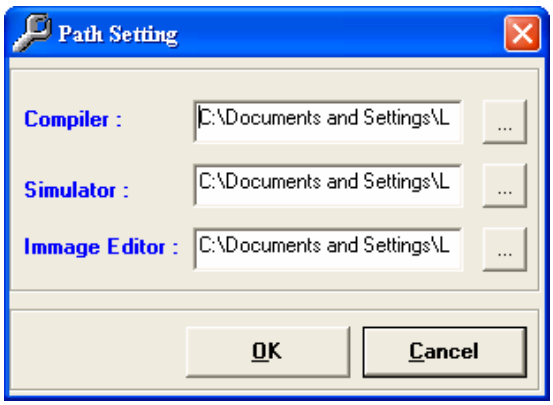


figura 16 – Schermata per il settaggio del percorso delle applicazioni

Le applicazioni che è possibile richiamare sono quelle presenti nel menu *Tools* o nella *ToolBar* . Ogni applicazione rappresenta un programma assestante, tale filosofia di progetto ha permesso di focalizzare di volta in volta i vari problemi realizzativi; in particolare *l'Image Editor* è stato realizzato in *Visual Basic* e permette di convertire una immagine bitmap in una immagine il cui formato è compatibile con la funzione di accesso dati a file usata nel sorgente *VHDL* durante la simulazione. Il convertitore di immagini pur accettando immagini a colori, durante la conversione le trasforma in scala di grigi compresi in un intervallo 0-255. In figura 17 viene riportata la videata principale dell'applicazione *Picture.exe*; in figura 18 viene invece riportato l'inizio del file immagine ottenuto dopo la conversione.

Il compilatore e il simulatore sono stati realizzati in *C*, questo ha permesso di ottenere diversi vantaggi; per quanto riguarda il simulatore, visto il numero elevato di istruzioni da simulare, (ogni istruzione viene interpretata ed eseguita) si sono raggiunti livelli di

[2] Tale operazione è necessaria dopo aver istallato l'applicazione da *CD-ROM*.

ottimizzazione temporale notevoli. Questo lo si può notare paragonando il tempo di scrittura del file da convertire (*Picture.exe*) e il tempo di scrittura del file contenete i contorni (*Simulatore.exe*).

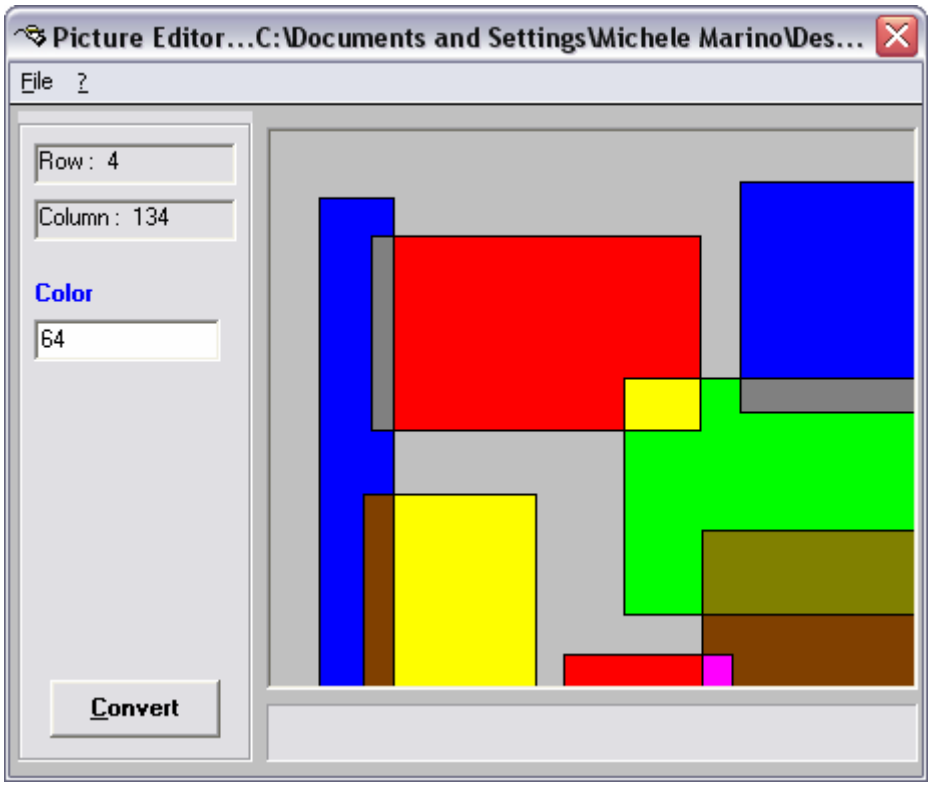


figura 17 – Schermata principale dell'editor delle immagini

```
width: 8
default: FF
00000: 90 40 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F
00010: 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F
00020: 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F
00030: 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F
00040: 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F
00050: 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F
```

figura 18 – Struttura dei dati per l'emulazione della memoria

La scrittura di un sorgente in *C* permette inoltre l'esecuzione dell'applicazione in ambiente *DOS;* la stesura dei sorgenti è stata fatta utilizzando solo funzioni dello standard *ANSI C* il che permette di ricompilarli anche su piattaforme utilizzanti altri sistemi operativi quali *LINUX*.

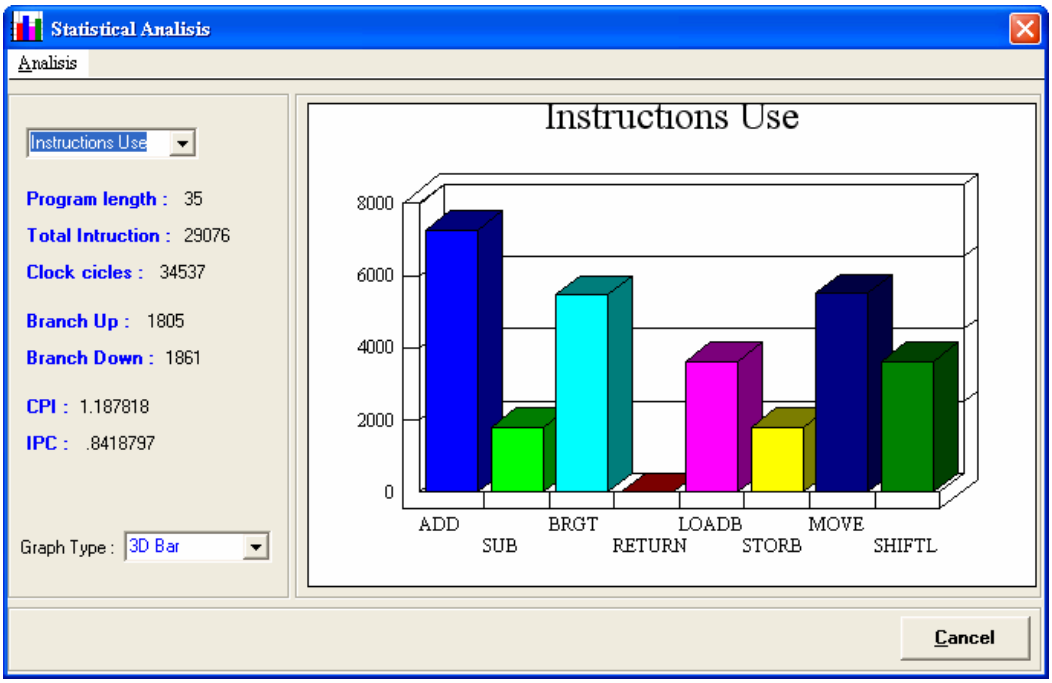


figura 19 – Schermata per la visualizzazione delle analisi statistiche

Nel menù *Tools* è presente anche la possibilità di analizzare graficamente i risultati, riportati in un file *.sta, dal processo *Simulator.exe*, al termine della simulazione. In figura 20 viene riportata la finestra grafica dopo aver aperto un file del simulatore.

Sulla parte destra della finestra vengono riportati i principali parametri che caratterizzano il risultato della simulazione. Sul grafico viene riportato il numero di volte che viene eseguita ogni singola istruzione (sono presenti 8 istruzioni); è possibile selezionare anche la voce *Registers Use* oltre a *Instructions Use*, visualizzando così il numero di accessi per ogni registro (sono presenti 16 registri).

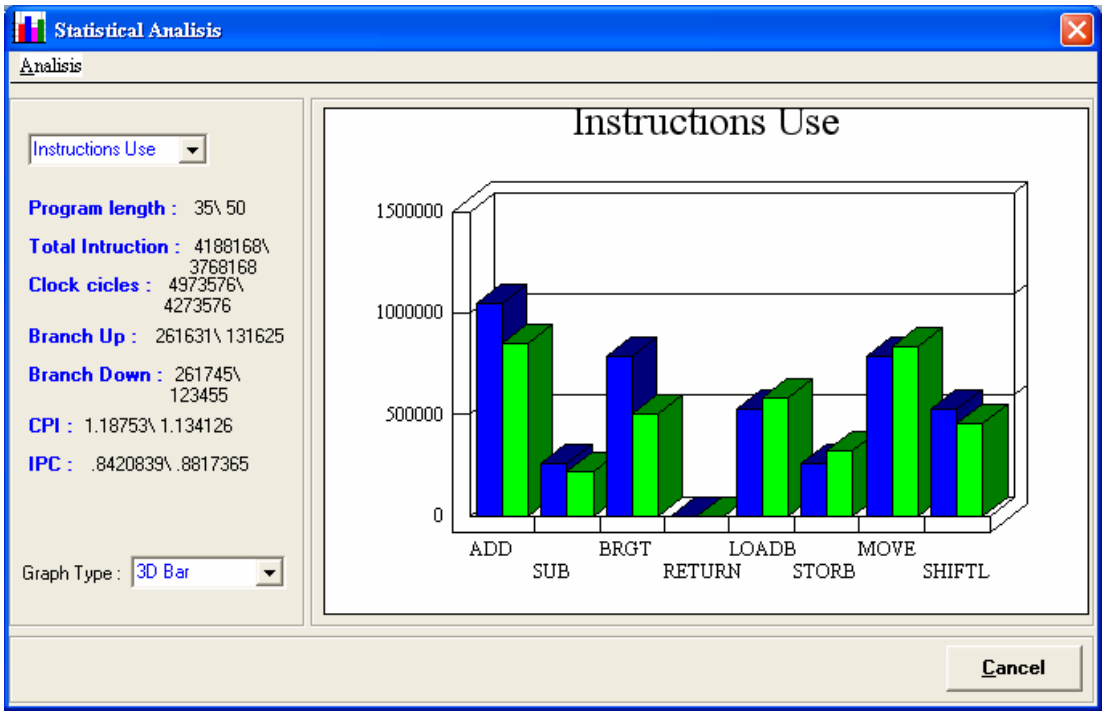


figura 20 - Schermata per la visualizzazione delle analisi statistiche

Il grafico può essere formattato in *3D Bar*, *2D Bar* e *2D Pie*. Dal menu *Analisis* →*Load Compare File* è possibile caricare un ulteriore file *.sta (figura 20) in modo da comparare i risultati statistici al variare del file immagine o del programma sorgente[3] (come per esempio nel caso di un *loop unroolling*) .

COMPILATORE

Il compilatore è stato realizzato al fine di agevolare la riscrittura e la codifica di programmi sorgenti. Al pari del resto del sistema anche tale programma è stato suddiviso in sottoprogrammi in modo da rendere la funzione *main* quanto più snella possibile; questo ha permesso di ritoccare il programma focalizzando senza problemi i punti di intervento. Oltre al file *Compiler.c* si sono realizzati i file header seguenti :

```
#include "myConstant.h"  // dichiara le costanti degli operandi e di programma
#include "myVariable.h"  // variabili globali
#include "myFunction.h"  // dichiara funzioni matematiche IntToBin BinToInt EsaToInt...
#include "Compute.h"     // funzioni di codifica delle istruzioni
```

Tale programma può essere eseguito anche in ambiente *DOS* e se ricompilato può essere eseguito anche con sistemi operativi differenti.

[3] In questo caso la rappresentazione grafica *2D Pie* non è impostabile.

Per eseguire il programma è necessario passargli come parametro il file da compilare.

```
C:\Compiler prova.asm
```

Se la compilazione viene eseguita tramite il programma *Editor.exe* come parametro viene automaticamente passato il file sorgente del progetto caricato.
Dal nome del file passato vengono creati due nuovi file; il primo è il file di log (estensione *.clg) che viene creato ogni volta che viene avviata una compilazione indipendentemente dal fatto che questa abbia esito positivo o meno. In questo file vengono scritte tutte le operazioni che vengono svolte dal compilatore con relative segnalazioni di errore nel caso di interruzione[4]. Il secondo file generato (estensione *.hex) viene invece generato solo se la compilazione termina senza errori.
La prima operazione che viene svolta dal compilatore è la ricerca delle etichette e relativi indirizzi puntati, la logica seguita è "tutto ciò che non è un'istruzione è un'etichetta". I valori trovati vengono memorizzati all'interno di una tabella così strutturata :

```
typedef struct
{
     char label [LABEL_LEN];  // nome etichetta
     short int target;        // indirizzo puntato
} Record;

Record etichette[NUM_LABEL];  // record di etichette con relativi indirizzi puntati
```

Tale tabella permette, una volta creata, di fare salti sia all'indietro che in avanti in maniera più snella. L'operazione loggata come "*Label Finder*" non comporta ancora la compilazione di alcuna istruzione.
Durante questa scansione vengono trovati, eventualmente, una prima tipologia di errori di sintassi; questo viene fatto con l'ausilio della variabile *flag* che permette di evitare che siano presenti due label consecutive.
Creata la tabella dei riferimenti avviene la compilazione delle istruzioni precedentemente scansionate. Tale operazione viene svolta rileggendo il file dall'inizio saltando tutto ciò che

[4] Oltre agli errori vengono loggate le varie fasi della compilazione permettendo di verificare, e questo è stato importane durante la stesura del programma, la correttezza degli operandi prelevati e della codifica svolta.

non è un'istruzione, ovvero le label[5]. Una volta rilevata una istruzione viene richiamata la relativa funzione che la codificherà in linguaggio macchina; nel caso di una istruzione *SUB* si avrebbe :

```
SubAdd (prg,log,PrgMemory);
```

alla funzione viene passato il puntatore del file *.asm in modo da poter andare a leggere gli operandi, il puntatore del file di log *.clg in modo da poter aggiornare il file con le nuove operazioni svolte e il puntatore alla memoria *RAM*, precedentemente allocata tramite la funzione *malloc*, dove risiedono le istruzioni compilate (in formato long int).
La lettura delle istruzioni da compilare termina con la fine del file. Giunti a questo punto, la compilazione deve essere avvenuta correttamente altrimenti il processo verrebbe interrotto prematuramente con la relativa segnalazione d'errore, viene generato il file *.hex in formato compatibile con la funzione di accesso ai dati in file *ASCII* presente nelle librerie di *ModelSim*. All'interno di ogni funzione relativa alla codifica di un'istruzione avviene la rilevazione di ulteriori errori:

`Miss parameter after the instruction add.`
Viene segnalato per ogni istruzione se mancano dei parametri per fine file.
`[label] is not recognize like any instruction.`
Viene segnalato se dopo un'etichetta non viene trovata un'istruzione valida (ci sarebbero due label a puntare lo stesso indirizzo).
`The label [ %s ] is not previously defined.`
Viene segnalato durante la codifica del BRGT se l'etichetta di salto non è nella tabella dei riferimenti.
`Shift position overflow.`
Viene segnalato quando lo shift è maggiore di 31 (i registri sono comunque di 16 bit ma per lo shift si hanno comunque 5 bit a disposizione)
`Immediate operator overflow.`

[5] Se si è giunti a questo punto della compilazione, senza interruzioni, dopo una label è sicuramente presente un'istruzione.

Viene segnalato se si tenta di caricare, nelle istruzione *MOVE* e *STORB* un operando maggiore di *512*.

`One or more Registers file are not available.`

Viene segnalato quando si fa involontariamente uso di un registro superiore a *R15*.

`R14 is not a user Register file.`

Viene segnalato quando si fa uso del registro *R14*

Quest'ultimo errore merita una nota in più. Si ricorda che l'istruzione *BRGT* non contiene tra gli operandi l'indirizzo a cui saltare, a causa del numero esiguo di bit a disposizione; l'indirizzo puntato dall'etichetta, contenuta tra gli operandi della *BRGT*, è contenuto in *R14*; tale indirizzo deve essere preventivamente caricato prima dell'istruzione BRGT. Questa operazione risulterebbe onerosa per il programmatore, per questo è stata mascherata dal compilatore che introduce una *MOVE R14,[indirizzo]* in maniera automatica. In questo modo il programmatore può effettuare salti in avanti o indietro facendo solo riferimento alle etichette da lui definite, ignorando il modo con cui verrà poi eseguita l'istruzione di *BRGT*.

Viene ora riportato un esempio di un semplice programma con relativo file di log.

```
loop    move R0,#1
        move R2,#30
        move R3,#8
        shiftl R3,R3,#2
loop1   sub R3,R3,R0
        brgt loop1,R3,R2
        return
```

Il file *.clg generato è il seguente:

```
Set Source file : Prova.asm
Open Source File : Prova.asm
Program's Array memory allocated


************** LABEL FINDER **************

loop is set like a label.
Pointed Memory : 0

loop1 is set like a label.
Pointed Memory : 8
```

```
************ FINDER TERMINATED ************

*********** COMPILER OPERATIONS ***********

[ move R0,#1 ] instruction is compiling.
Destination Register : 0
Immediate operator : 1
Word compiled like : 1100000000000001

[ move R2,#30 ] instruction is compiling.
Destination Register : 2
Immediate operator : 30
Word compiled like : 1100010000011110

[ move R3,#8 ] instruction is compiling.
Destination Register : 3
Immediate operator : 8
Word compiled like : 1100011000001000

[ shiftl R3,R3,#2 ] instruction is compiling.
Destination Register : 3
Source 1 Register : 3
Shift position : 2
Word compiled like : 1110011001100010

[ sub R3,R3,R0 ] instruction is compiling.
Destination Register : 3
Source 1 Register : 3
Source 2 Register : 0
Word compiled like : 0010011001100000

[ brgt loop1,R3,R2 ] instruction is compiling.
Source 1 Register : 3
Source 2 Register : 2
The label is : loop1
Pointed Memory : 8
Word [ move ] compiled like : 1101110000001000←istruzione move inserita dal compilatore
Word [ brgt ] compiled like : 0100011001011100

[ return ] instruction is compiling.
Word compiled like : 0110000000000000

*************** END COMPILE ***************

Hex File created.

Total lines compiled : 8

---------------------------------

Compiled program :

C001
C41E
C608
E662
2660
DC08
465C
6000
---------------------------------
```

```
Log END
```

Il file *.hex generato tiene conto del fatto che la memoria è a 8 bit.

width: 8
default: FF
0000: 01
0001: C0
0002: 1E
0003: C4
0004: 08
0005: C6
[…]
000D: 46
000E: 00
000F: 60

Per ulteriori dettagli sul programma si rimanda ai commenti presenti sul file sorgente.

SIMULATORE

Anche il processo simulatore necessita in ingresso di alcuni parametri, secondo la seguente sintassi:

`c:\Simulator <*.hex file> <*.ram Image File> <-log| -unlog>`

Il file *.hex viene passato in funzione del file *.asm del progetto corrente mentre il file immagine e l'opzione log o unlog dipendono dalla finestra di dialogo riportata in figura 21.

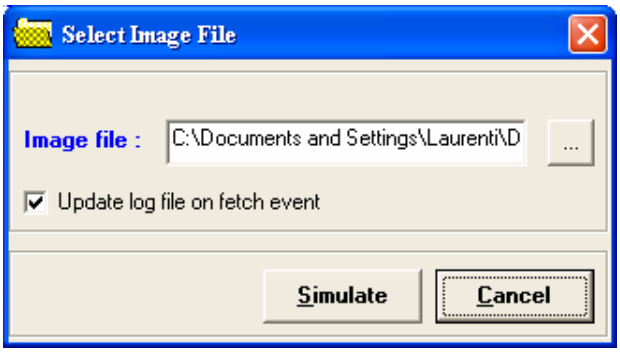


figura 21 – Schermata per la selezione del file del file immagine

L'opzione "*Update log file on fetch event*" permette di abilitare la scrittura del file di log ogni volta che viene caricata una nuova istruzione; in realtà vengono loggate tutte le fasi successive ovvero quella di decode e di execute riportando in ultimo il *dump* dei registri

interni. Tale opzione è di particolare utilità per controllare parti di programma critiche (la sua abilitazione rallenta la simulazione a causa del continuo accesso al file di *.slg). Il simulatore al pari del compilatore può essere eseguito in ambiente *DOS* o altri sistemi operativi, se ricompilato.

Il programma principale *Simulator.c* include i seguenti file header (oltre ad alcuni header delle librerie standard):

```
#include "myVariable.h"        // variabili globali
#include "myConstant.h"        // dichiara le costanti degli operandi e di programma
#include "myFunction.h"        // dichiara funzioni matematiche IntToBin BinToInt…
#include "exeIstr.h"           // funzioni relative alle istruzioni  da eseguire
#include "compute.h"           // ha le funzioni fetch decode execoute
```

Alcune, utilizzate nel programma di compilazione, sono state riutilizzate anche in questo programma. Dopo la lettura e il controllo dei parametri passati il programma provvede ad allocare degli spazi di memoria in modo da simulare la *RAM* del microprocessore e la rispettiva *ROM*. Allocati gli spazi opportuni viene letto il file esadecimale e successivamente il file immagine. Lo spazio di memoria *RAM* dedicata all'immagine destinazione viene settato a *FFh* tramite programma piuttosto che leggere e riportare in *RAM* il contenuto del file. Tutte le operazioni di cui sopra vengono loggate nel file *.slg. Prima di avviare la simulazione vengono inizializzati gli array di contatori nei quali vengono conteggiati gli accessi ai registri interni e l'uso delle varie istruzioni.

Complessivamente ai fini dell'analisi statistica sono utilizzate le seguenti variabili :

```
int NumeroIstruzioni;     // conta il numero di istruzioni lette

long int counterReg [15];           // conta accesso ai singoli registri
long int counterIstr [15];          // conta esecuzione delle singole istruzioni (8)
long int branchUp =0;
long int branchDown =0;
```

Prima di avviare la simulazione viene posta la variabile *PC=0*. L'intera simulazione viene svolta all'interno del seguente ciclo infinito, alla fine del quale viene creato un file di output contenente l'immagine di uscita e il file di log viene chiuso.

```
while (1) // ciclo infinito
{    printf (".");                          // stampa sullo schermo punto/istruzione
     fetch (Istruzione,PrgMemory,log);
     decode (Istruzione,log);
     PC++;                                  // il BRGT cambierà eventualmente il PC
     if (execut (ImgMemory,log)) break; // ritorna 1 se esegue RETURN (esco se 1)
}
```

Da quanto spiegato si capisce l'utilità dei file header, grazie ai quali si è potuta mantenere una linearità concettuale nella stesura del programma. Le funzioni fetch, decode ed execute sono contenute nel file *compute.h*; da questo file vengono poi richiamate le rispettive funzioni (contenute nel file *exeIstr.h*) che implementano le istruzioni di volta in volta caricate e decodificate. Si riporta sotto il file di log *.slg relativo alla simulazione del file precedentemente compilato; non si fa uso della memoria destinata all'immagine destinazione.

```
Set Program file : C:\WINDOWS\Desktop\Prova.hex
Set Image file : C:\WINDOWS\Desktop\Digitale\Sorgente\img_mem.img
Program's Array memory allocated
Image's Array memory allocated
Open Program file : C:\WINDOWS\Desktop\Prova.hex
Program loaded.
Open Image file : C:\WINDOWS\Desktop\Digitale\Sorgente\img_mem.img
Image loaded.
Image's destination RAM is set to 0xFF.

The loaded Program is :

0  : 1100000000000001
1  : 1100010000011110
2  : 1100011000001000
3  : 1110011001100010
4  : 0010011001100000
5  : 1101110000001000
6  : 0100011001011100
7  : 0110000000000000

********** START SIMULATION **********

Interrupt received
Internal Registers clear

Fetch intstruction  0 : 1100000000000001
Instruction Decode like : MOVE
Opcode : 6
Destination Register : 0
Immediate Operand : 1
Execute done.

Register Dump

Reg0  : 0000000000000000001
Reg1  : 0000000000000000000
Reg2  : 0000000000000000000
Reg3  : 0000000000000000000
[…]
Reg14 : 0000000000000000000
Reg15 : 0000000000000000000

Fetch intstruction  1 : 1100010000011110
Instruction Decode like : MOVE
Opcode : 6
Destination Register : 2
Immediate Operand : 30
```

```
Execute done.

Register Dump

Reg0  : 0000000000000000001
Reg1  : 0000000000000000000
Reg2  : 0000000000000011110
Reg3  : 0000000000000000000
[…]
Reg14 : 0000000000000000000
Reg15 : 0000000000000000000

Fetch intstruction  2 : 1100011000001000
Instruction Decode like : MOVE
Opcode : 6
Destination Register : 3
Immediate Operand : 8
Execute done.

Register Dump

Reg0  : 0000000000000000001
Reg1  : 0000000000000000000
Reg2  : 0000000000000011110
Reg3  : 0000000000000001000
[…]
Reg14 : 0000000000000000000
Reg15 : 0000000000000000000

Fetch intstruction  3 : 1110011001100010
Instruction Decode like : SHIFTL
Opcode : 7
Destination Register : 3
Source Register 1 : 3
Immediate Operand : 2
Execute done.

Register Dump

Reg0  : 0000000000000000001
Reg1  : 0000000000000000000
Reg2  : 0000000000000011110
Reg3  : 0000000000000100000
[…]
Reg14 : 0000000000000000000
Reg15 : 0000000000000000000

Fetch intstruction  4 : 0010011001100000
Instruction Decode like : SUB
Opcode : 1
Destination Register : 3
Source Register 1 : 3
Source Register 2 : 0
Execute done.

Register Dump

Reg0  : 0000000000000000001
Reg1  : 0000000000000000000
Reg2  : 0000000000000011110
Reg3  : 0000000000000011111
[…]
```

```
Reg14 : 0000000000000000
Reg15 : 0000000000000000

Fetch intstruction  5 : 1101110000001000
Instruction Decode like : MOVE
Opcode : 6
Destination Register : 14
Immediate Operand : 8
Execute done.

Register Dump

Reg0  : 0000000000000001
Reg1  : 0000000000000000
Reg2  : 0000000000011110
Reg3  : 0000000000011111
[…]
Reg14 : 0000000000001000
Reg15 : 0000000000000000

Fetch intstruction  6 : 0100011001011100
Instruction Decode like : BRGT
Opcode : 2
Load PC from Register : 14
Source Register 1 : 3
Source Register 2 : 2
Execute done.

Register Dump

Reg0  : 0000000000000001
Reg1  : 0000000000000000
Reg2  : 0000000000011110
Reg3  : 0000000000011111
[…]
Reg14 : 0000000000001000
Reg15 : 0000000000000000

Fetch intstruction  4 : 0010011001100000
Instruction Decode like : SUB
Opcode : 1
Destination Register : 3
Source Register 1 : 3
Source Register 2 : 0
Execute done.

Register Dump

Reg0  : 0000000000000001
Reg1  : 0000000000000000
Reg2  : 0000000000011110
Reg3  : 0000000000011110
[…]
Reg14 : 0000000000001000
Reg15 : 0000000000000000

Fetch intstruction  5 : 1101110000001000
Instruction Decode like : MOVE
Opcode : 6
Destination Register : 14
Immediate Operand : 8
Execute done.
```

```
      Register Dump

      Reg0  : 0000000000000001
      Reg1  : 0000000000000000
      Reg2  : 0000000000011110
      Reg3  : 0000000000011110
      […]
      Reg14 : 0000000000001000
      Reg15 : 0000000000000000

      Fetch intstruction  6 : 0100011001011100
      Instruction Decode like : BRGT
      Opcode : 2
      Load PC from Register : 14
      Source Register 1 : 3
      Source Register 2 : 2
      Execute done.

      Register Dump

      Reg0  : 0000000000000001
      Reg1  : 0000000000000000
      Reg2  : 0000000000011110
      Reg3  : 0000000000011110
      […]
      Reg14 : 0000000000001000
      Reg15 : 0000000000000000

      Fetch intstruction  7 : 0110000000000000
      Instruction Decode like : RETURN
      Opcode : 3

      ***********  SIMULATION END  ***********

      Statistics file created : C:\WINDOWS\Desktop\Prova.sta
      Image file output created : C:\WINDOWS\Desktop\Prova.img
Log END
```

Simulazioni

Qui di seguito si riportano i risultati della simulazione di alcune istruzioni effettuate sia con il software di simulazione appositamente realizzato per il processore, sia con ModelSim 5.7f.

Fetch delle istruzioni

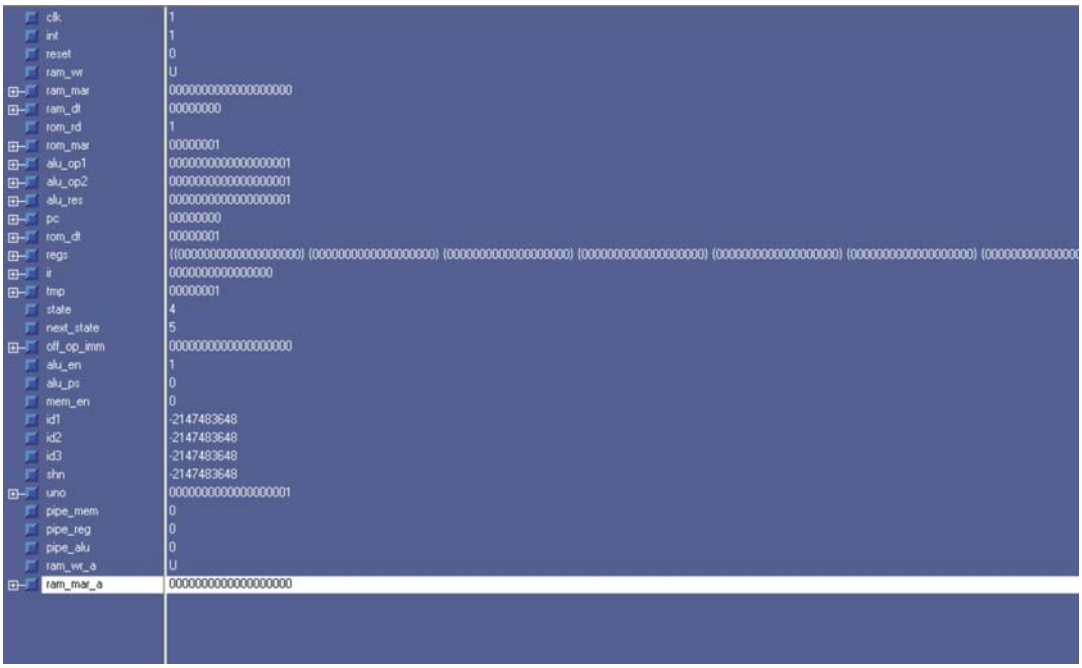


figura 22 - Lista completa dei segnali e dei registri del sistema

La figura 22 mostra un elenco di tutti i segnali presenti nel nostro sistema.

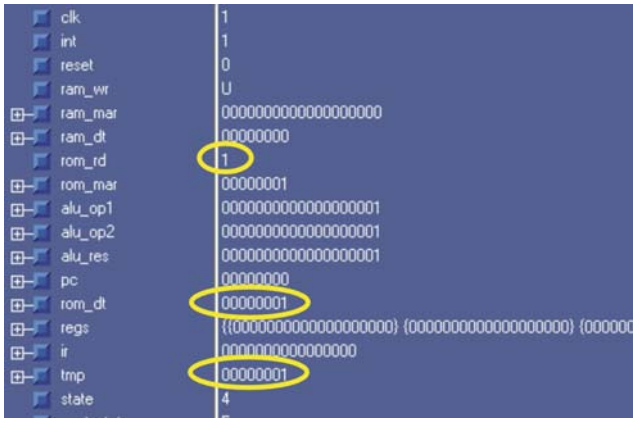


figura 23 - Caricamento prima parte dell'istruzione

La figura 23 mostra l'attivazione del segnale rom_rd (abilitazione lettura dalla rom) e l'aggiornamento dei registri rom_dt e temp (vengono caricati i primi 8 bit dell'istruzione).

ns / delta	/edge_det_cpu/rom_dt	/edge_det_cpu/tmp	/edge_det_cpu/ir	/edge_det_cpu/state	/edge_det_cpu/next_state
0 +0	00000000	0000000000000000	00000000	-2147483648	-2147483648
0 +1	00000000	0000000000000000	00000000	-2147483648	1
100 +1	00000000	0000000000000000	00000000	1	1
100 +2	00000000	0000000000000000	00000000	1	2
200 +1	00000000	0000000000000000	00000000	2	2
200 +2	00000000	0000000000000000	00000000	2	3
300 +1	00000000	0000000000000000	00000000	3	3
300 +2	00000000	0000000000000000	00000000	3	4
350 +1	00000001	0000000000000000	00000000	3	4
400 +1	00000001	0000000000000000	00000001	4	4
400 +2	00000001	0000000000000000	00000001	4	5
450 +1	11000000	0000000000000000	00000001	4	5
500 +1	11000000	1100000000000001	00000001	5	5
500 +2	11000000	1100000000000001	00000001	5	7

figura 24 - Caricamento dell'intera istruzione nell'IR

Come si vede dalla figura 24 il caricamento dell'istruzione nell'IR avviene attraverso diverse fasi. La fase cerchiata in rosso rappresenta l'operazione (che avviene sul fronte di discesa) di caricamento della prima metà di istruzione nel registro rom_dt. In seguito (cerchio giallo) il valore caricato in rom_dt viene riversato in un registro di appoggio (tmp). Successivamente viene caricata in IR l'intera istruzione (rom_dt&tmp). E' interessante notare come sia necessario un tempo delta per la determinazione dello stato prossimo. L'istruzione caricata in questo caso è una istruzione di trasferimento, in particolare una "MOVE", (i bit più significativi dell'IR sono a 110).

Istruzione MOVE

ns / delta	/edge_det_cpu/off_op_imm	/edge_det_cpu/id3	/edge_det_cpu/state	/edge_det_cpu/next_state
600 +2	00000000000000000001	0	7	2
700 +1	00000000000000000001	0	2	2
700 +2	00000000000000000001	0	2	3
800 +1	00000000000000000001	0	3	3
800 +2	00000000000000000001	0	3	4
900 +1	00000000000000000001	0	4	4
900 +2	00000000000000000001	0	4	5
1000 +1	00000000000000000001	0	5	5
1000 +2	00000000000000100000	3	5	7
1100 +1	00000000000000100000	3	7	7
1100 +2	00000000000000100000	3	7	2
1200 +1	00000000000000100000	3	2	2
1200 +2	00000000000000100000	3	2	3
1300 +1	00000000000000100000	3	3	3
1300 +2	00000000000000100000	3	3	4

figura 25 - Variazione di registri implicati nell'operazione di MOVE

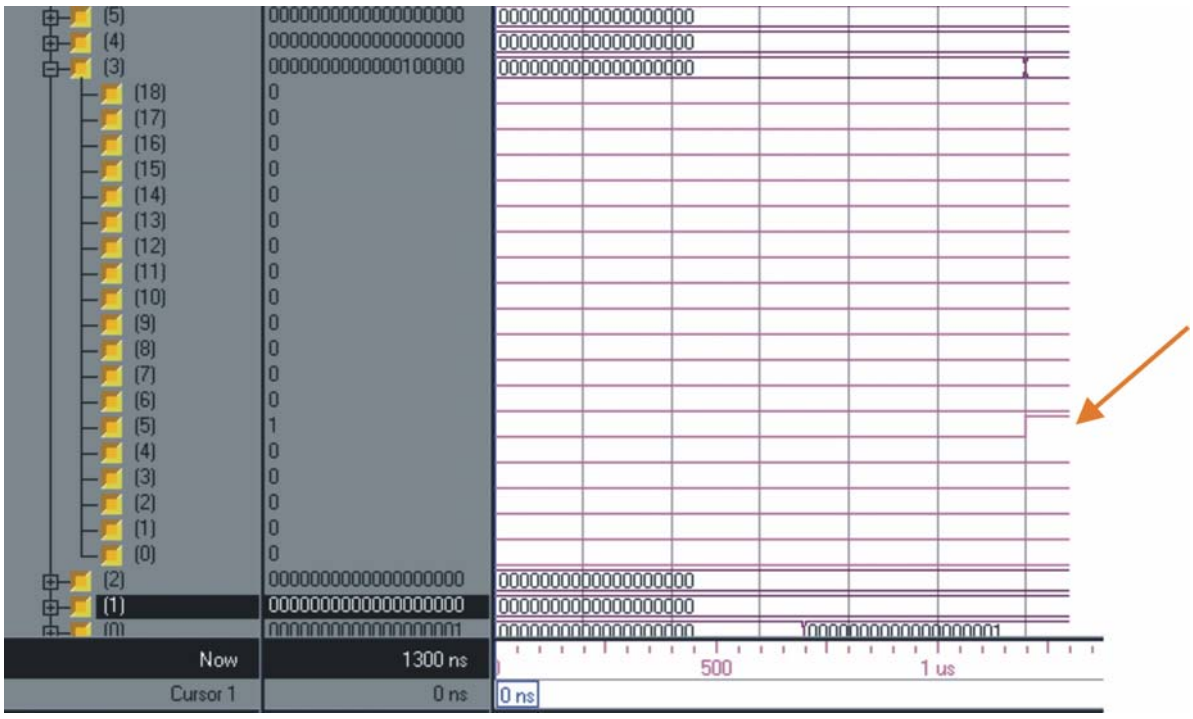


figura 26 - Risultato di una MOVE sul registro R3 (MOVE R3, #32)

Dalla figura 25 si vedono le varie fasi che portano all'esecuzione dell'istruzione MOVE R3, #32. La prima fase è quella del caricamento (vista in precedenza) che ha luogo a partire da t=700ns. Da t=1000ns viene caricato nel registro off_op_imm il valore 32, mentre nel registro di indirizzamento-registri (id3) viene caricato il valore 3. Nell'istante t=1200ns il valore 32 viene caricato in R3, come illustrato in figura 26.

```
Fetch instruction  1 :
1100011000100000
Instruction Decode like : MOVE
Opcode : 6
Destination Register : 3
Immediate Operand : 32
Execute done.

Registers Dump

Reg0  : 0000000000000000001
Reg1  : 0000000000000000000
Reg2  : 0000000000000000000
Reg3  : 0000000000000100000
Reg4  : 0000000000000000000
Reg5  : 0000000000000000000
Reg6  : 0000000000000000000
Reg7  : 0000000000000000000
Reg8  : 0000000000000000000
Reg9  : 0000000000000000000
Reg10 : 0000000000000000000
Reg11 : 0000000000000000000
Reg12 : 0000000000000000000
Reg13 : 0000000000000000000
Reg14 : 0000000000000000000
Reg15 : 0000000000000000000
```

figura 27 - Risultato di una MOVE sul registro R3 (MOVE R3, #32) fornito dal nostro simulatore

Si vede anche dal file di log prodotto dal simulatore da noi sviluppato, che l'operazione viene eseguita correttamente.

Istruzione SHIFTL

ns delta	/edge_det_cpu/id2	/edge_det_cpu/id3	/edge_det_cpu/shn	/edge_det_cpu/state	/edge_det_cpu/next_state
1200 +2	1	3	-2147483648	2	3
1300 +1	1	3	-2147483648	3	3
1300 +2	1	3	-2147483648	3	4
1400 +1	1	3	-2147483648	4	4
1400 +2	1	3	-2147483648	4	5
1500 +1	1	3	-2147483648	5	5
1500 +2	3	3	4	5	8
1600 +1	3	3	4	8	8
1600 +2	3	3	4	8	2
1700 +1	3	3	4	2	2

figura 28 - Variazione dei registri implicati nell'operazione di SHIFTL

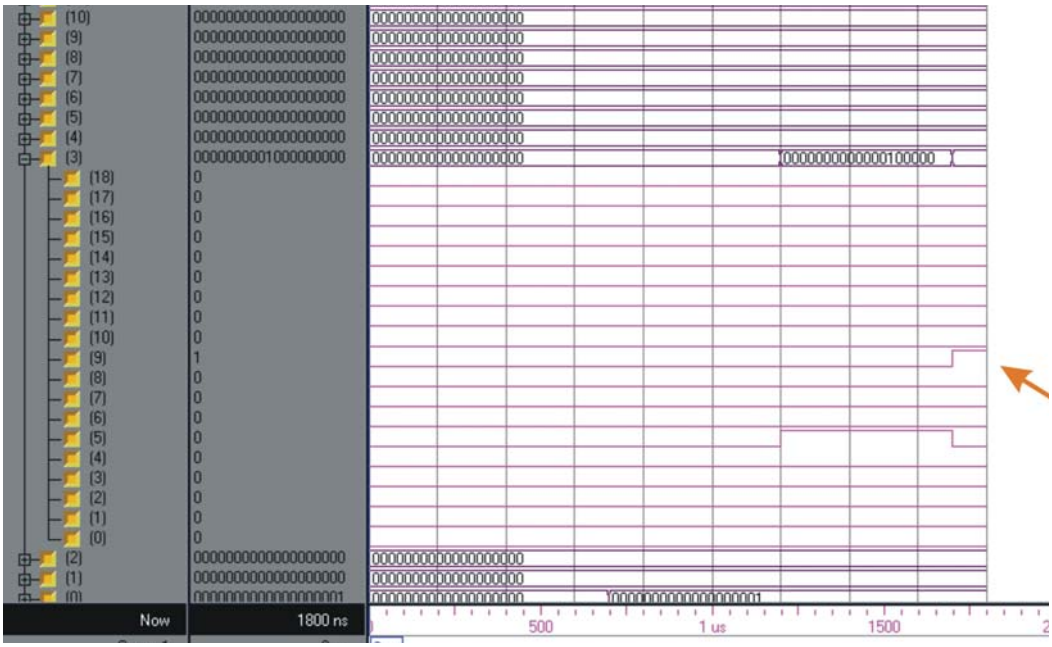


figura 29 - Risultato di una SHIFTL sul registro R3 (SHIFTL R3, R3, #4)

```
Fetch instruction  2 :
1110011001100100
Instruction Decode like : SHIFTL
Opcode : 7
Destination Register : 3
Source Register 1 : 3
Immediate Operand : 4
Execute done.

Registers Dump

Reg0  : 0000000000000000001
Reg1  : 0000000000000000000
Reg2  : 0000000000000000000
Reg3  : 0000000001000000000
Reg4  : 0000000000000000000
Reg5  : 0000000000000000000
Reg6  : 0000000000000000000
Reg7  : 0000000000000000000
Reg8  : 0000000000000000000
Reg9  : 0000000000000000000
Reg10 : 0000000000000000000
Reg11 : 0000000000000000000
Reg12 : 0000000000000000000
Reg13 : 0000000000000000000
Reg14 : 0000000000000000000
Reg15 : 0000000000000000000
```

figura 30 - Risultato di una SHIFTL sul registro R3 (SHIFTL R3, R3, #4) fornito dal nostro simulatore

La figura 28 mostra le variazioni che interessano i registri implicati nell'operazione SHIFTL R3, R3, #4. Si nota che l'operazione, come al solito, inizia col caricamento dell'istruzione (in questo caso a partire da t=1200ns), poi prosegue col caricamento del valore 3 in id3 (identifica il registro destinazione), del valore 3 in id2 (individua il registro da cui prender il dato da shiftare), il valore 4 in shn (contiene il numero di posizioni di cui il dato deve essere shiftato). Come si vede dalla figura 29 e dalla figura 30, l'operazione di shift viene eseguita correttamente (t=1700 ns), sia con ModelSim che con il nostro simulatore.

Istruzione SUB

ns / delta	/edge_det_cpu/alu_op1	/edge_det_cpu/alu_op2	/edge_det_cpu/alu_res	/edge_det_cpu/state	/edge_det_cpu/next_state
2200 +1	0000000001000000000	0000000000000000001	0000000000000000110	2	2
2200 +2	0000000001000000000	0000000000000000001	0000000000000000110	2	3
2250 +1	0000000001000000000	0000000000000000001	0000000000111111111	2	3
2300 +1	0000000000000001000	0000000000000000001	0000000000111111111	3	3

figura 31 - Variazione dei registri implicati nell'operazione di SUB

La figura 31 mostra come un'operazione SUB alteri i registri alu_op1, alu_op2, alu_res. L'uscita dell'ALU (alu_res) rappresenta la differenza tra i due operandi di ingresso. Il risultato viene reso disponibile in uscita sul fronte di discesa del clock.

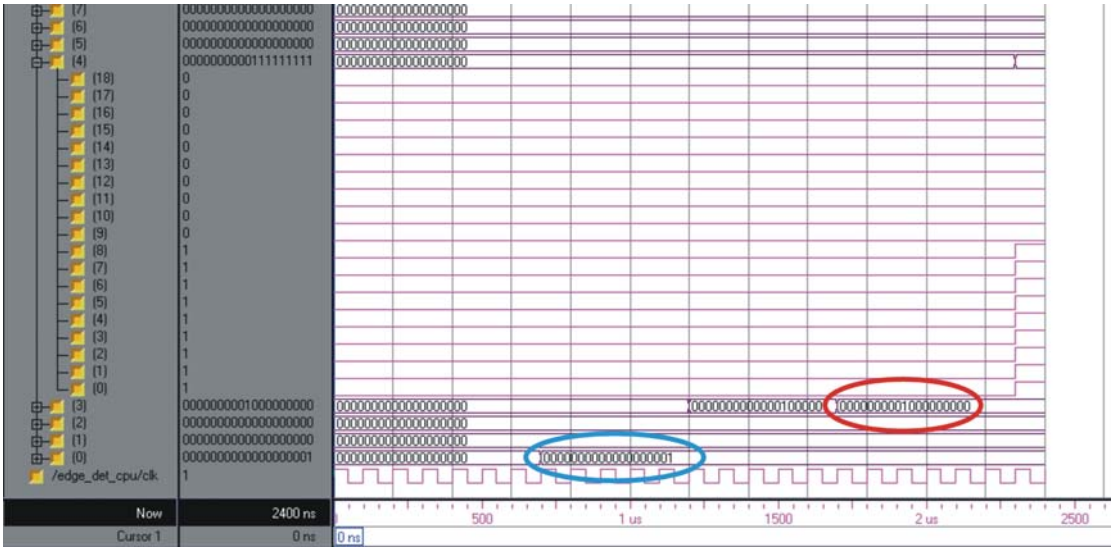


figura 32 - Risultato di una SUB sul registro R4 (SUB R4, R3, R0)

```
Fetch instruction  3 :
0010100001100000
Instruction Decode like : SUB
Opcode : 1
Destination Register : 4
Source Register 1 : 3
Source Register 2 : 0
Execute done.

Registers Dump

Reg0  : 0000000000000000001
Reg1  : 0000000000000000000
Reg2  : 0000000000000000000
Reg3  : 0000000001000000000
Reg4  : 0000000000111111111
Reg5  : 0000000000000000000
Reg6  : 0000000000000000000
Reg7  : 0000000000000000000
Reg8  : 0000000000000000000
Reg9  : 0000000000000000000
Reg10 : 0000000000000000000
Reg11 : 0000000000000000000
Reg12 : 0000000000000000000
Reg13 : 0000000000000000000
Reg14 : 0000000000000000000
Reg15 : 0000000000000000000
```

figura 33 - Risultato di una SUB sul registro R4 (SUB R4, R3, R0) fornito dal nostro simulatore

Dalla figura 32 e figura 33 invece viene presa in esame la variazione del registro R4 in base ai valori di R3 ed R0. Anche in questo caso l'operazione viene eseguita correttamente sia da ModelSim che dal nostro simulatore.

Istruzione ADD

Un ragionamento del tutto analogo può essere seguito nel caso di un'operazione di ADD. Nelle figure seguenti viene mostrata la variazione degli operandi dell'ALU (figura 34) e dei registri coinvolti nell'operazione ADD R8, R7, R0 (figura 35)

ns delta	/edge_det_cpu/alu_op1	/edge_det_cpu/alu_op2	/edge_det_cpu/alu_res	/edge_det_cpu/state	/edge_det_cpu/next_state
5700 +1	0000000000000000000	0000000000000000001	0000000000000010110	2	2
5700 +2	0000000000000000000	0000000000000000001	0000000000000010110	2	3
5750 +1	0000000000000000000	0000000000000000001	0000000000000000001	2	3
5800 +1	0000000000000010110	0000000000000000001	0000000000000000001	3	3

figura 34 - Variazione dei registri implicati nell'operazione di ADD

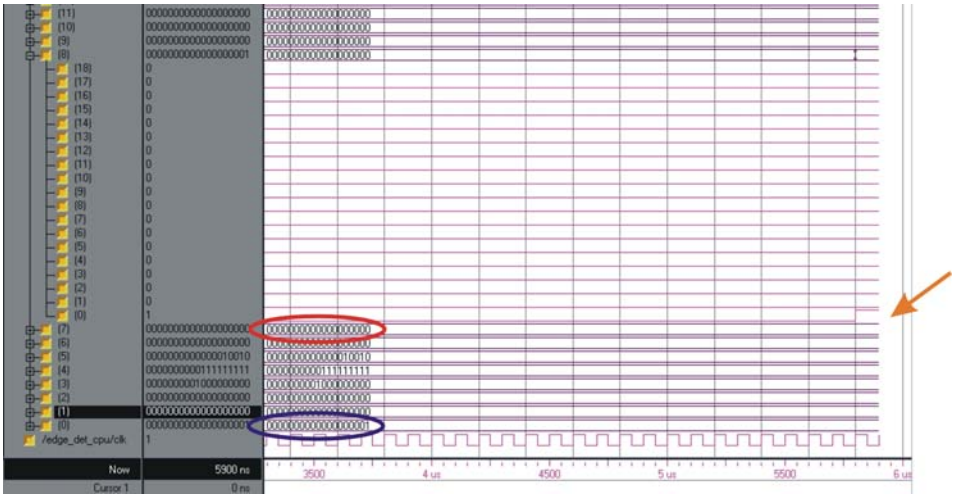


figura 35 - Risultato di una ADD sul registro R8 (ADD R8, R7, R0)

```
Fetch instruction  10 :
0001000011100000
Instruction Decode like : ADD
Opcode : 0
Destination Register : 8
Source Register 1 : 7
Source Register 2 : 0
Execute done.

Registers Dump

Reg0  : 0000000000000000001
Reg1  : 0000000000000000000
Reg2  : 0000000000000000000
Reg3  : 0000000001000000000
Reg4  : 0000000000111111111
Reg5  : 0000000000000010010
Reg6  : 0000000000000000000
Reg7  : 0000000000000000000
Reg8  : 0000000000000000001
Reg9  : 0000000000000000000
Reg10 : 0000000000000000000
Reg11 : 0000000000000000000
Reg12 : 0000000000000000000
Reg13 : 0000000000000000000
Reg14 : 0000000000000000000
Reg15 : 1111111111111101110
```

figura 36 - Risultato di una ADD sul registro R8 (ADD R8, R7, R0) fornito dal nostro simulatore

La figura 36 mostra che anche il nostro software di simulazione effettua l'operazione correttamente.

Istruzione LOADB

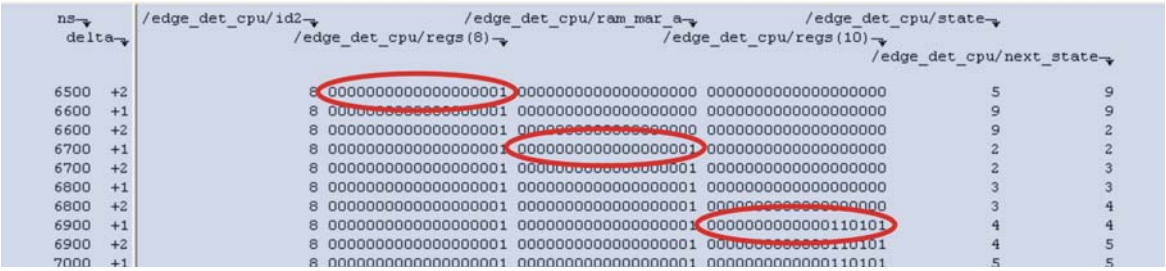


figura 37 - Variazione dei registri implicati nell'operazione di LOADB

```
width: 8
default: FF
00000: 35 35 35 35 35 35 35 35 35 35 35 35 35 35 35 35
00010: 35 35 35 35 35 35 35 35 35 35 35 35 35 35 35 35
00020: 35 35 35 35 35 35 35 35 35 35 35 35 35 35 35 35
00030: 35 35 35 35 35 35 35 35 35 35 35 35 35 35 35 35
00040: 35 35 35 35 35 35 35 35 35 35 35 35 35 35 35 35
00050: 35 35 35 35 35 35 35 35 35 35 35 35 35 35 35 35
00060: 35 35 35 35 35 35 35 35 35 35 35 35 35 35 35 35
```

tabella 1 - Contenuto del file sorgente img_mem.img

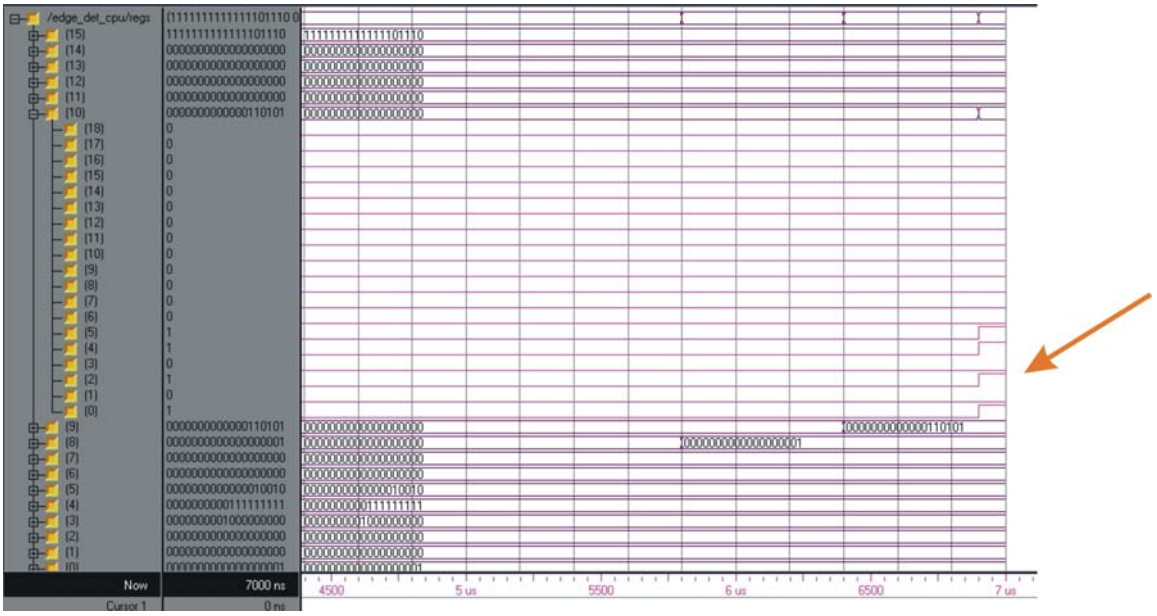


figura 38 - Risultato dell'operazione LOADB R10, (R8)

La figura 37 mostra la variazione del contenuto del registro ram_mar_a dovuta all'istruzione LOADB R10, (R8). Ad ir2 viene assegnato il valore contenuto in R8, dopodichè in ram_mar_a viene caricato il contenuto della locazione di memoria esterna

(che contiene l'immagine) puntata da ir2. Il valore sarà pronto in ram_mar_a all'istante t = 6700ns mentre il registro R10 sarà aggiornato all'istante t = 6900ns. Si nota che in R10 è stato scritto il valore esadecimale 35, contenuto nella locazione di memoria di indirizzo 000001(hex) come si può vedere dal confronto tra la tabella 1 e la figura 37, figura 38 e figura 39.

```
Fetch instruction  12 :
1001010100000000
Instruction Decode like : LOADB
Opcode : 4
Destination Register : 10
Source Register 1 : 8
Execute done.

Registers Dump

Reg0  : 0000000000000000001
Reg1  : 0000000000000000000
Reg2  : 0000000000000000000
Reg3  : 0000000001000000000
Reg4  : 0000000000111111111
Reg5  : 0000000000000010010
Reg6  : 0000000000000000000
Reg7  : 0000000000000000000
Reg8  : 0000000000000000001
Reg9  : 0000000000000110101
Reg10 : 0000000000000110101
Reg11 : 0000000000000000000
Reg12 : 0000000000000000000
Reg13 : 0000000000000000000
Reg14 : 0000000000000000000
Reg15 : 1111111111111101110
```

figura 39 - Risultato dell'operazione LOADB R10, (R8) fornita dal nostro simulatore

Istruzione STORB

```
00000: 87 35 35 35 35 35 35 35 35 35 35 35 35 35 35 35
00010: 35 35 35 35 35 35 35 35 35 35 35 35 35 35 35 35
00020: 35 35 35 35 35 35 35 35 35 35 35 35 35 35 35 35
00030: 35 35 35 35 35 35 35 35 35 35 35 35 35 35 35 35
00040: 35 35 35 35 35 35 35 35 35 35 35 35 35 35 35 35
00050: 35 35 35 35 35 35 35 35 35 35 35 35 35 35 35 35
```

tabella 2 - Contenuto del file sorgente img_mem.img

La tabella 2 mostra il contenuto del file che rappresenta la nostra memoria in cui è contenuta l'immagine (i valori sono espressi in base esadecimale). Come si vede la differenza tra il pixel in posizione 00000 (cerchiato in rosso) è maggiore di 18 (12hex, soglia fissata per l'individuazione di un contorno).

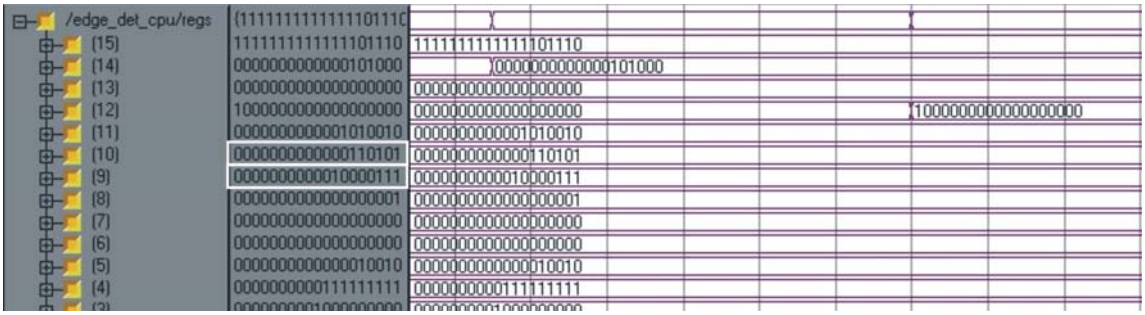


figura 40 - Contenuto dei registri R9 ed R10 dopo la lettura della memoria

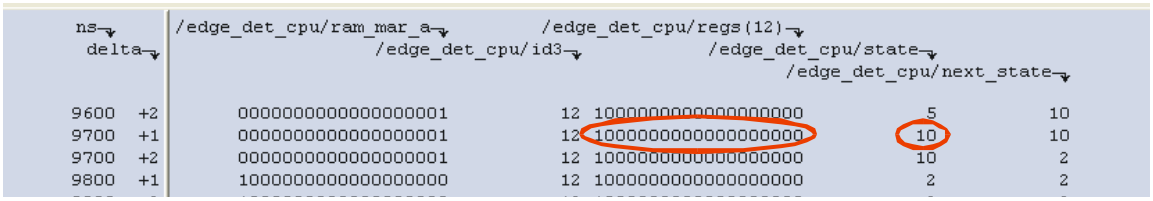
Tali valori sono caricati rispettivamente nei registri R9 ed R10, come si vede dalla figura 40 e figura 41 (che mostrano lo stesso risultato ottenuto con i due diversi sistemi di simulazione) . In R11 è caricata invece la loro differenza.

```
Fetch instruction  22 :
1011100011001000
Instruction Decode like : STORB
Opcode : 5
Destination Register : 12
Immediate Operand : 200
Execute done.

Registers Dump

Reg0  : 0000000000000000001
Reg1  : 0000000000000000000
Reg2  : 0000000000000000000
Reg3  : 0000000001000000000
Reg4  : 0000000000111111111
Reg5  : 0000000000000010010
Reg6  : 0000000000000000000
Reg7  : 0000000000000000000
Reg8  : 0000000000000000001
Reg9  : 0000000000010000111
Reg10 : 0000000000000110101
Reg11 : 0000000000001010010
Reg12 : 0000000000000000001
Reg13 : 0000000000000000000
Reg14 : 0000000000000101000
Reg15 : 1111111111111101110
```

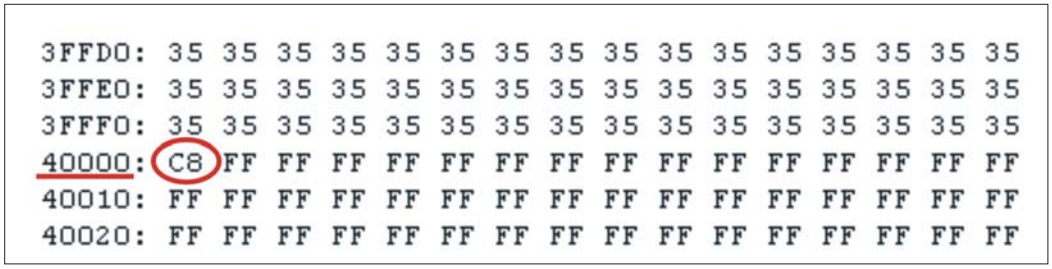
figura 41 - Contenuto dei registri R9 ed R10 dopo la lettura della memoria fornito dal nostro simulatore



ns delta	/edge_det_cpu/ram_mar_a /edge_det_cpu/id3	/edge_det_cpu/regs(12)	/edge_det_cpu/state	/edge_det_cpu/next_state
9600 +2	0000000000000000001	12 1000000000000000000	5	10
9700 +1	0000000000000000001	12 1000000000000000000	10	10
9700 +2	0000000000000000001	12 1000000000000000000	10	2
9800 +1	1000000000000000000	12 1000000000000000000	2	2

figura 42 - Contenuto dei registri R12 e ram_mar durante l'operazione STORB (stato 10)

Visto che la differenza è superiore a 18 si verifica la condizione che avevamo imposto per l'esecuzione della STORB. In R12 viene memorizzato il valore dell'indirizzo della memoria esterna (la stessa memoria img_mem.img) in cui andremo a scrivere il valore 200 (C8hex).

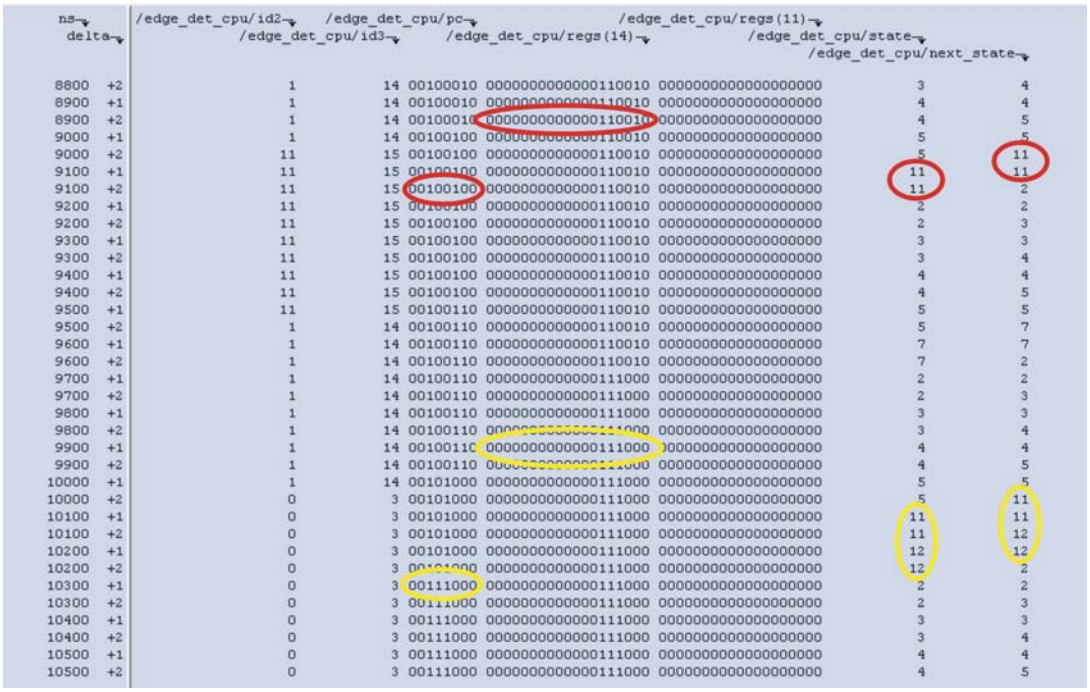


```
3FFD0: 35 35 35 35 35 35 35 35 35 35 35 35 35 35 35 35
3FFE0: 35 35 35 35 35 35 35 35 35 35 35 35 35 35 35 35
3FFF0: 35 35 35 35 35 35 35 35 35 35 35 35 35 35 35 35
40000: C8 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
40010: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
40020: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
```

tabella 3 - Contenuto del file img_mem.img dopo l'operazione di STORB

La tabella 3 conferma quanto appena detto, infatti il valore C8 viene caricato nella posizione 40000hex (1000000000000000000 in binario, come si vede dalla figura 42).

Istruzione BRGT



ns / delta	/edge_det_cpu/id2	/edge_det_cpu/id3	/edge_det_cpu/pc	/edge_det_cpu/regs(14)	/edge_det_cpu/regs(11)	/edge_det_cpu/state	/edge_det_cpu/next_state
8800 +2	1	14	00100010	00000000000000110010	00000000000000000000	3	4
8900 +1	1	14	00100010	00000000000000110010	00000000000000000000	4	4
8900 +2	1	14	00100010	00000000000000110010	00000000000000000000	4	5
9000 +1	1	14	00100100	00000000000000110010	00000000000000000000	5	5
9000 +2	11	15	00100100	00000000000000110010	00000000000000000000	5	11
9100 +1	11	15	00100100	00000000000000110010	00000000000000000000	11	11
9100 +2	11	15	00100100	00000000000000110010	00000000000000000000	11	2
9200 +1	11	15	00100100	00000000000000110010	00000000000000000000	2	2
9200 +2	11	15	00100100	00000000000000110010	00000000000000000000	2	3
9300 +1	11	15	00100100	00000000000000110010	00000000000000000000	3	3
9300 +2	11	15	00100100	00000000000000110010	00000000000000000000	3	4
9400 +1	11	15	00100100	00000000000000110010	00000000000000000000	4	4
9400 +2	11	15	00100100	00000000000000110010	00000000000000000000	4	5
9500 +1	11	15	00100110	00000000000000110010	00000000000000000000	5	5
9500 +2	1	14	00100110	00000000000000110010	00000000000000000000	5	7
9600 +1	1	14	00100110	00000000000000110010	00000000000000000000	7	7
9600 +2	1	14	00100110	00000000000000110010	00000000000000000000	7	2
9700 +1	1	14	00100110	00000000000000111000	00000000000000000000	2	2
9700 +2	1	14	00100110	00000000000000111000	00000000000000000000	2	3
9800 +1	1	14	00100110	00000000000000111000	00000000000000000000	3	3
9800 +2	1	14	00100110	00000000000000111000	00000000000000000000	3	4
9900 +1	1	14	00100110	00000000000000111000	00000000000000000000	4	4
9900 +2	1	14	00100110	00000000000000111000	00000000000000000000	4	5
10000 +1	1	14	00101000	00000000000000111000	00000000000000000000	5	5
10000 +2	0	3	00101000	00000000000000111000	00000000000000000000	5	11
10100 +1	0	3	00101000	00000000000000111000	00000000000000000000	11	11
10100 +2	0	3	00101000	00000000000000111000	00000000000000000000	11	12
10200 +1	0	3	00101000	00000000000000111000	00000000000000000000	12	12
10200 +2	0	3	00101000	00000000000000111000	00000000000000000000	12	2
10300 +1	0	3	00111000	00000000000000111000	00000000000000000000	2	2
10300 +2	0	3	00111000	00000000000000111000	00000000000000000000	2	3
10400 +1	0	3	00111000	00000000000000111000	00000000000000000000	3	3
10400 +2	0	3	00111000	00000000000000111000	00000000000000000000	3	4
10500 +1	0	3	00111000	00000000000000111000	00000000000000000000	4	4
10500 +2	0	3	00111000	00000000000000111000	00000000000000000000	4	5

figura 43 - Variazione dei registri implicati in un'istruzione di BRANCH

Ora vediamo il test di funzionamento di istruzioni di BRANCH. La prima istruzione di salto (BRGT IF_2 R15, R11) non deve dare luogo a salto, infatti (come da figura 43) il PC viene incrementato di sole due posizioni e dallo stato 11 (verifica della condizione per il salto) si torna allo stato 2 (stato iniziale). La seconda BRANCH (BRGT ENDIF R3, R0) sarà invece eseguita in quanto in R0 (vedi figura 44) è contenuto un valore minore di quello contenuto in R3. Si passerà quindi per lo stato 12 (evidenziato in giallo) in cui il PC viene aggiornato con il valore contenuto in R14.

Si noti che prima dello stato 11 si passa nello stato 7 (MOVE) per pre-caricare in R14 l'eventuale valore del PC nel caso in cui il salto fosse da eseguire. Se il salto viene eseguito PC<=R14 altrimenti PC<=PC+2. Questo si evince dalle figg. 17a e 17b prese dal file di log del nostro simulatore.

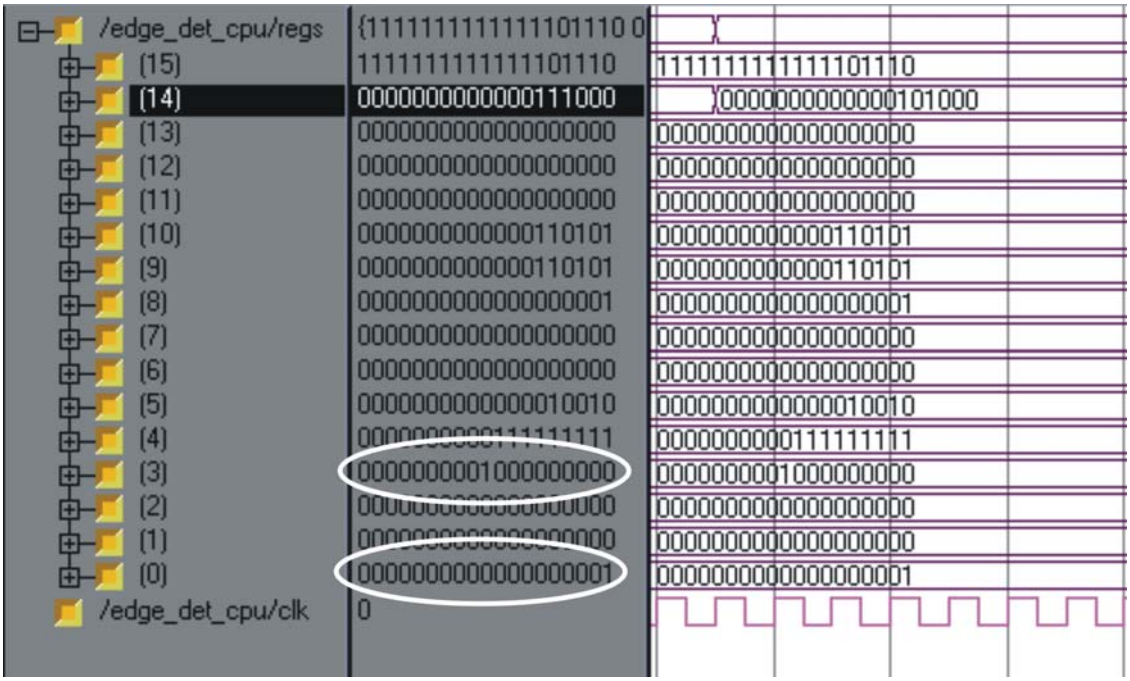


figura 44 – Registri implicati nella condizione di salto BRGT ENDIF,R3,R0

La figura 45 mostra il caso in cui un salto non venga preso. Infatti l'indicatore dell'istruzione caricata successivamente è incrementato di uno rispetto al precedente (dall'istruzione 17 si passa alla 18). La figura 46 invece mostra il caso in cui un salto avvenga. L'istruzione successiva, di cui viene fatto il fetch, ha il valore relativo al salto (in questo caso si passa dall'istruzione 19 all'istruzione 28).

```
Fetch instruction  16 :
1101110000110010
Instruction Decode like : MOVE
Opcode : 6
Destination Register : 14
Immediate Operand : 50
Execute done.
Registers Dump
Reg0  : 0000000000000000001
Reg1  : 0000000000000000000
Reg2  : 0000000000000000000
Reg3  : 0000000001000000000
Reg4  : 0000000000111111111
Reg5  : 0000000000000010010
Reg6  : 0000000000000000000
Reg7  : 0000000000000000000
Reg8  : 0000000000000000001
Reg9  : 0000000000000110101
Reg10 : 0000000000000110101
Reg11 : 0000000000000000000
Reg12 : 0000000000000000000
Reg13 : 0000000000000000000
Reg14 : 0000000000000110010
Reg15 : 1111111111111101110

Fetch instruction  17 :
0101111101111100
Instruction Decode like : BRGT
Opcode : 2
Load PC from Register : 14
Source Register 1 : 15
Source Register 2 : 11

Execute done.
Registers Dump
Reg0  : 0000000000000000001
Reg1  : 0000000000000000000
Reg2  : 0000000000000000000
Reg3  : 0000000001000000000
Reg4  : 0000000000111111111
Reg5  : 0000000000000010010
Reg6  : 0000000000000000000
Reg7  : 0000000000000000000
Reg8  : 0000000000000000001
Reg9  : 0000000000000110101
Reg10 : 0000000000000110101
Reg11 : 0000000000000000000
Reg12 : 0000000000000000000
Reg13 : 0000000000000000000
Reg14 : 0000000000000110010
Reg15 : 1111111111111101110

Fetch instruction  18 :
1101110000111000
Instruction Decode like : MOVE
Opcode : 6
Destination Register : 14
Immediate Operand : 56
Execute done.
```

figura 45 - Fetch di istruzione seguente ad un salto "non preso"

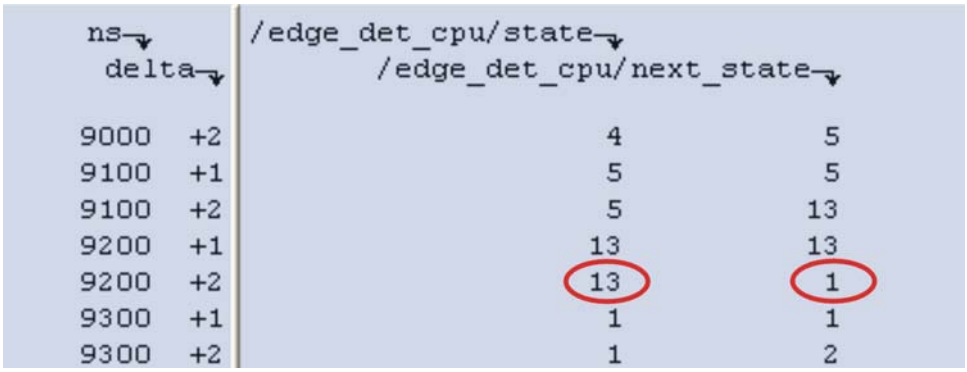
```
Fetch instruction  18 :
1101110000111000
Instruction Decode like : MOVE
Opcode : 6
Destination Register : 14
Immediate Operand : 56
Execute done.
Registers Dump
Reg0  : 0000000000000000001
Reg1  : 0000000000000000000
Reg2  : 0000000000000000000
Reg3  : 0000000001000000000
Reg4  : 0000000000111111111
Reg5  : 0000000000000010010
Reg6  : 0000000000000000000
Reg7  : 0000000000000000000
Reg8  : 0000000000000000001
Reg9  : 0000000000000110101
Reg10 : 0000000000000110101
Reg11 : 0000000000000000000
Reg12 : 0000000000000000000
Reg13 : 0000000000000000000
Reg14 : 0000000000000111000
Reg15 : 1111111111111101110

Fetch instruction  19 :
0100011000011100
Instruction Decode like : BRGT
Opcode : 2
Load PC from Register : 14
Source Register 1 : 3
Source Register 2 : 0
Execute done.
Registers Dump
Reg0  : 0000000000000000001
Reg1  : 0000000000000000000
Reg2  : 0000000000000000000
Reg3  : 0000000001000000000
Reg4  : 0000000000111111111
Reg5  : 0000000000000010010
Reg6  : 0000000000000000000
Reg7  : 0000000000000000000
Reg8  : 0000000000000000001
Reg9  : 0000000000000110101
Reg10 : 0000000000000110101
Reg11 : 0000000000000000000
Reg12 : 0000000000000000000
Reg13 : 0000000000000000000
Reg14 : 0000000000000111000
Reg15 : 1111111111111101110

Fetch instruction  28 :
0000010001000000
Instruction Decode like : ADD
Opcode : 0
Destination Register : 2
Source Register 1 : 2
Source Register 2 : 0
Execute done.
```

figura 46 - Fetch di un'istruzione seguente un salto "preso"

Istruzione RETURN



ns delta	/edge_det_cpu/state	/edge_det_cpu/next_state
9000 +2	4	5
9100 +1	5	5
9100 +2	5	13
9200 +1	13	13
9200 +2	13	1
9300 +1	1	1
9300 +2	1	2

figura 47 - Variazione del registro di stato dopo l'istruzione RETURN

L'operazione RETURN (stato 13) ci riporta nello stato 1 (attesa dell'interrupt) come previsto.