Indice

Elenco delle figure

Sommario

> "Le persone che scoprono la potenza e la bellezza delle idee di alto livello di astrazione spesso commettono l'errore di credere che le idee concrete ai livelli inferiori siano tutto sommato inutili e possano essere dimenticate.
>
> Al contrario, i migliori informatici sono sempre saldamente radicati nei concetti basilari che governano il funzionamento dei calcolatori, ed in verità, l'essenza dell'informatica è l'abilità di comprendere e governare molti livelli di astrazione contemporaneamente."

Questo intervento di Donald Knuth, all' ottava Conferenza Annuale dell' ITiCSE-03, sottolinea come un'approfondita comprensione del reale funzionamento di un calcolatore sia fondamentale per ben padroneggiare l'informatica.

Questa tesi ha come obiettivo quello di definire, ad un livello di astrazione opportuno, una semplice CPU virtuale a scopo didattico e successivamente quello di prototipare sia l'architettura che l'interfaccia utente dell'emulatore per essa.

vCPU non vuole essere una CPU in grado di competere con le tecnologie moderne: il suo scopo principale è quello di rendere disponibile, ad un pubblico di studenti, uno strumento in grado di facilitare l'apprendimento dei concetti base dell'architettura dei calcolatori e quindi, nello spirito della citazione di Knuth, di fornire delle solide basi per la prosecuzione degli studi in informatica.

Parte I

Stato dell'Arte

Prima di affrontare la specifica di vCPU e la progettazione di un prototipo di emulatore per essa, è bene analizzare lo stato dell'arte di software simili, come ad esempio simulatori di CPU virtuali od anche emulatori, pur se per CPU reali. In particolare, si avrà modo di capire la differenza che intercorre tra un emulatore e un simulatore, comprendendo l'importanza di entrambe a livello didattico.

Capitolo 1

Emulatori

Un emulatore è un software che ricrea l'hardware di un sistema, solitamente differente da quello di cui si dispone, permettendo di far eseguire programmi di vario genere come se fossero eseguiti nell'hardware emulato. Nel seguito ne vengono analizzati alcuni allo scopo di presentare lo stato dell'arte di questo genere di software.

1.1 Emu8086

L'emulatore ha come obiettivo quello di creare un ambiente in grado di eseguire programmi scritti per i8086 e mostrare la struttura interna della CPU, permettendo così di comprendere al meglio il funzionamento sia del programma che del processore stesso. Come molti altri emulatori, Emu8086 permette varie modalità di esecuzione, tra cui quella di tracking e di debugging: ciò permette di interagire con il flusso del programma, permettendo di comprenderne il funzionamento. La GUI del software, mostrata in figura 1.1 si compone di più finestre, ognuna delle quali è adibita a contenere sia diversi aspetti della CPU, sia strumenti per la programmazione. Di notevole importanza sono quelle che mostrano lo stato dello stack, dei flag, della ALU e quella che mostra l'output del programma.

L'interfaccia presenta anche la possibilità di editare programmi in linguaggio assembly, rendendo così l'utente indipendente da altri editor esterni. Tra gli altri strumenti si trovano anche un convertitore ed una calcolatrice, per facilitare il lavoro al programmatore.

Una delle funzionalità principali è la possibilità di compilare i programmi scritti in assembly, generando file COM eseguibili su qualunque computer x86 con un sistema compatibile con DOS. Attualmente, il software è reperibile al seguente URL:

http://www.geocities.com/emu8086/

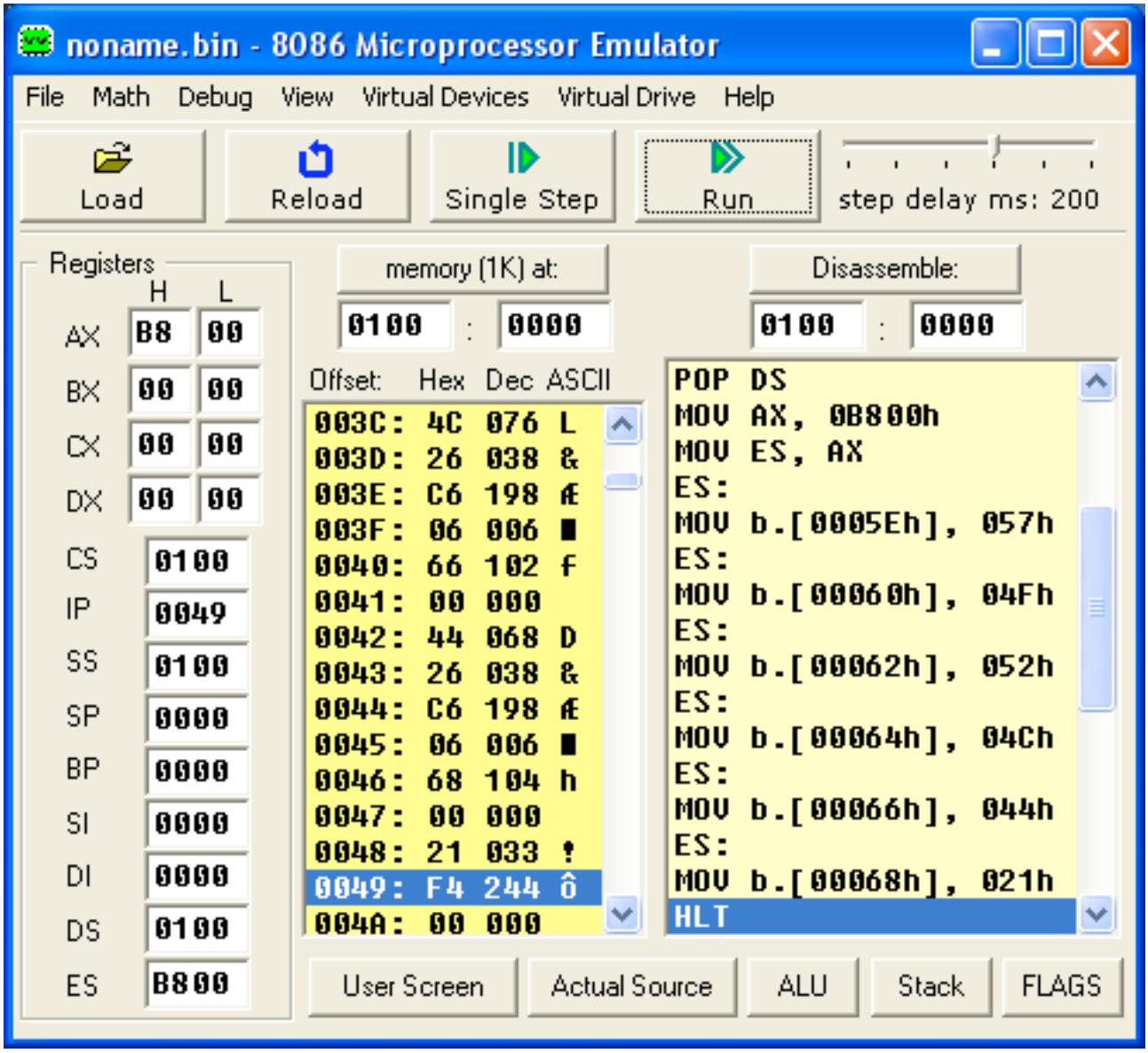


Figura 1.1: GUI di Emu8086

1.2 Emu486

Questo software nasce per essere un emulatore di CPU i80486. Scritto interamente per ambiente DOS, riesce tuttavia ad essere compatibile anche con i sistemi operativi più moderni, come ad esempio WindowsXP. Il software offre varie finestre (fig. 1.2) che mostrano il contenuto dei registri, dei flag e della memoria. Interessante è la finestra che mostra le istruzioni, in sintassi Intel, a partire dal contenuto della memoria: ciò permette una agevole lettura del codice dei programmi.

L'emulatore dispone di una serie di comandi in grado sia di variare la modalità di esecuzione, sia di entrare in modalità di debug. Ad esempio è possibile impostare e rimuovere dei breakpoint, ottenerne la lista completa, entrare in modalità di tracing, etc.

Il software è accompagnato da un piccolo sistema operativo, RDOS, con lo scopo di rende più semplice lo sviluppo di applicazioni per questo emulatore, introducendo delle routine che implementano le funzionalità di base presenti in ogni sistema operativo.

Attualmente, il software è reperibile al seguente URL:

http://www.chez.com/mobali/sperso/emu486/html/welcome.html.

1.3 S85

Questo software è utilizzato per emulare le CPU i8085. Si presenta con una interfaccia testuale molto semplice: la schermata è divisa in sei aree, ognuna delle quali rappresenta una finestra verso l'interno della CPU o verso risorse come la memoria e le porte. La figura 1.3 mostra l'aspetto dell'applicazione. Tra le principali finestre si segnalano quella dei registri, della memoria e del programma. Quest' ultima consiste in un dump di una porzione di memoria interpretata, per poter mostrare il programma sotto forma di istruzioni in sintassi Intel.

Uno degli aspetti che rende interessante questo emulatore è proprio l'architettura emulata: infatti, l'i8085 presenta registri molto semplici a 8 bit e una memoria indirizzabile a 16 bit. Questa semplicità predispone l'emulatore ad essere un valido strumento per l'apprendimento dell' assembly.

Scritto per il DOS, l'emulatore è comunque compatibile con i sistemi operativi moderni, come ad esempio Windows XP.

Attualmente, il software è reperibile al seguente URL:

http://www.freecolormanagement.com/s85/

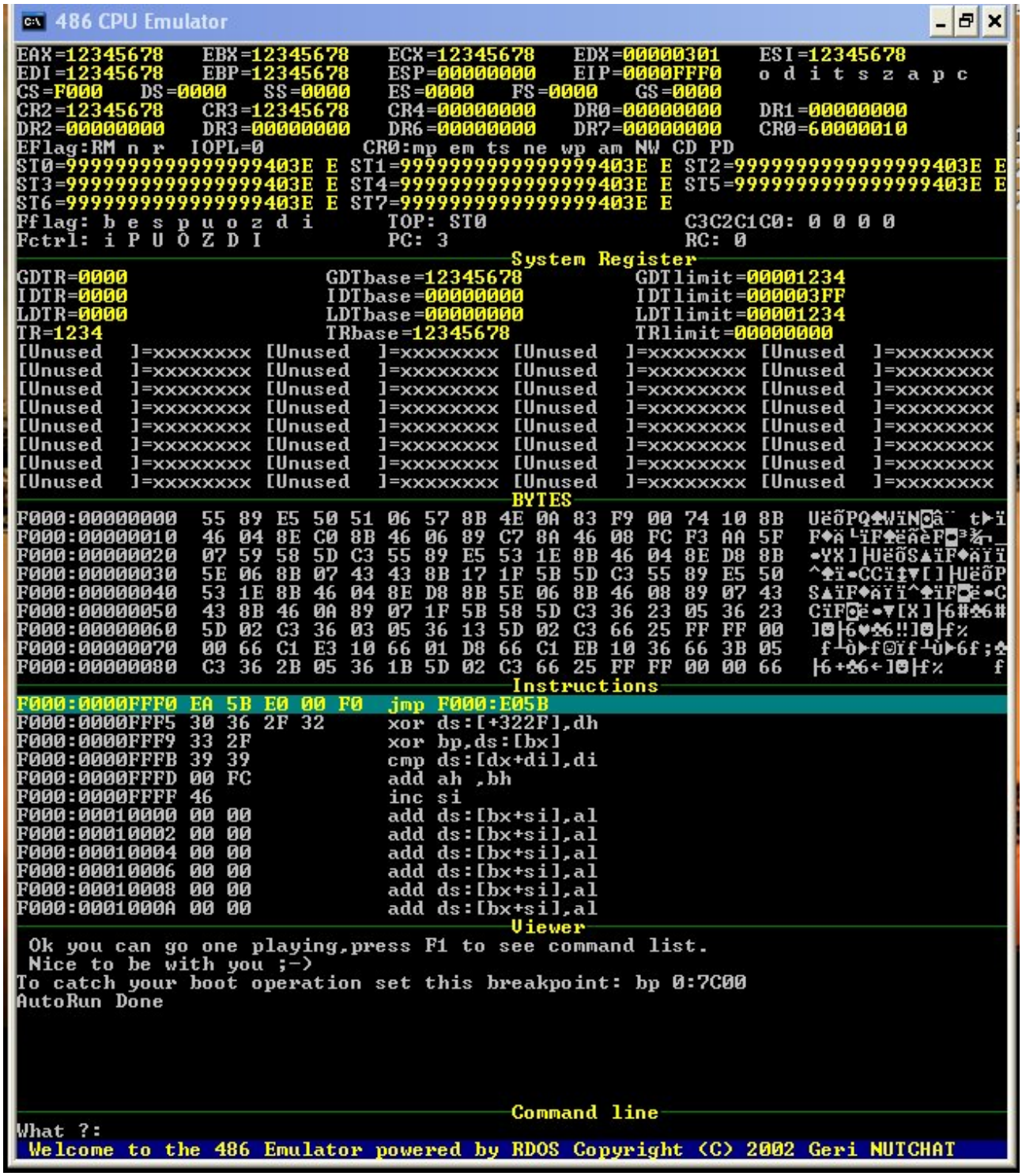


Figura 1.2: GUI di Emu486

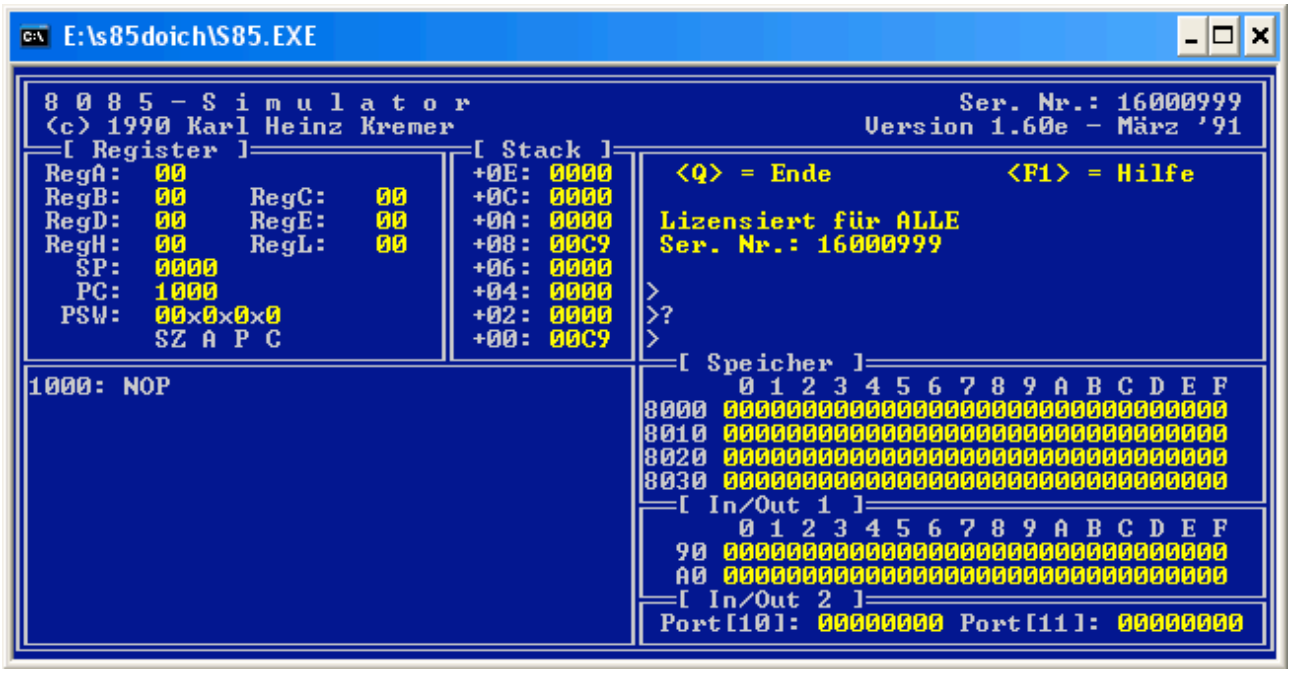


Figura 1.3: GUI di s85

Capitolo 2

Simulatori

Un simulatore è un software che imita l'hardware di un sistema. La differenza tra emulatori e simulatori è abbastanza sottile e non universalmente accettata. La sostanziale differenza tra emulatore e simulatore è che mentre il primo ha come obiettivo primario il funzionamento del software scritto per l'hardware emulato, per il secondo tale obiettivo è secondario, e l'attenzione è principalmente rivolta alla rappresentazione degli aspetti architetturali e funzionali dell'hardware simulato. Nel seguito si analizzano alcuni simulatori esistenti.

2.1 SPIM MIPS Simulator

Spim è un simulatore che esegue programmi scritti nel linguaggio assembly MIPS32 e, data la sua natura di simulatore, non esegue codice compilato. Il software fornisce un semplice debugger e un minimo dei servizi offerti da un sistema operativo. Il set di istruzioni implementato corrisponde quasi all'intero assembler MIPS esteso: le istruzioni non implementate sono quelle relative al confronto tra numeri in virgola mobile e quelle legate alla gestione della memoria. L'interfaccia di spim (fig. 2.1) mette a disposizione cinque finestre, di cui tre sono adibite a visualizzare il contenuto delle varie risorse della CPU, come i registri, il segmento dei dati e il segmento del codice. Nelle restanti due si trovano la finestra con i vari comandi e la finestra con i messaggi per l'utente inviati dallo spim stesso.

Il simulatore supporta varie modalità di esecuzione: quella classica e quella step-by-step. Tra le altre funzionalità vi è anche la possibilità di inserire dei breakpoint nel programma.

Spim è distribuito in due versioni: una testuale e una grafica. Per la versione grafica, vi sono implementazioni differenti in base al sistema operativo: per i sistemi Unix-like è disponibile *xspim*, mentre per i sistemi Microsoft si trova *PCSpim*.

Esiste una ulteriore variante di spim, *spimsal*, che si basa sulle sue prime versioni. La

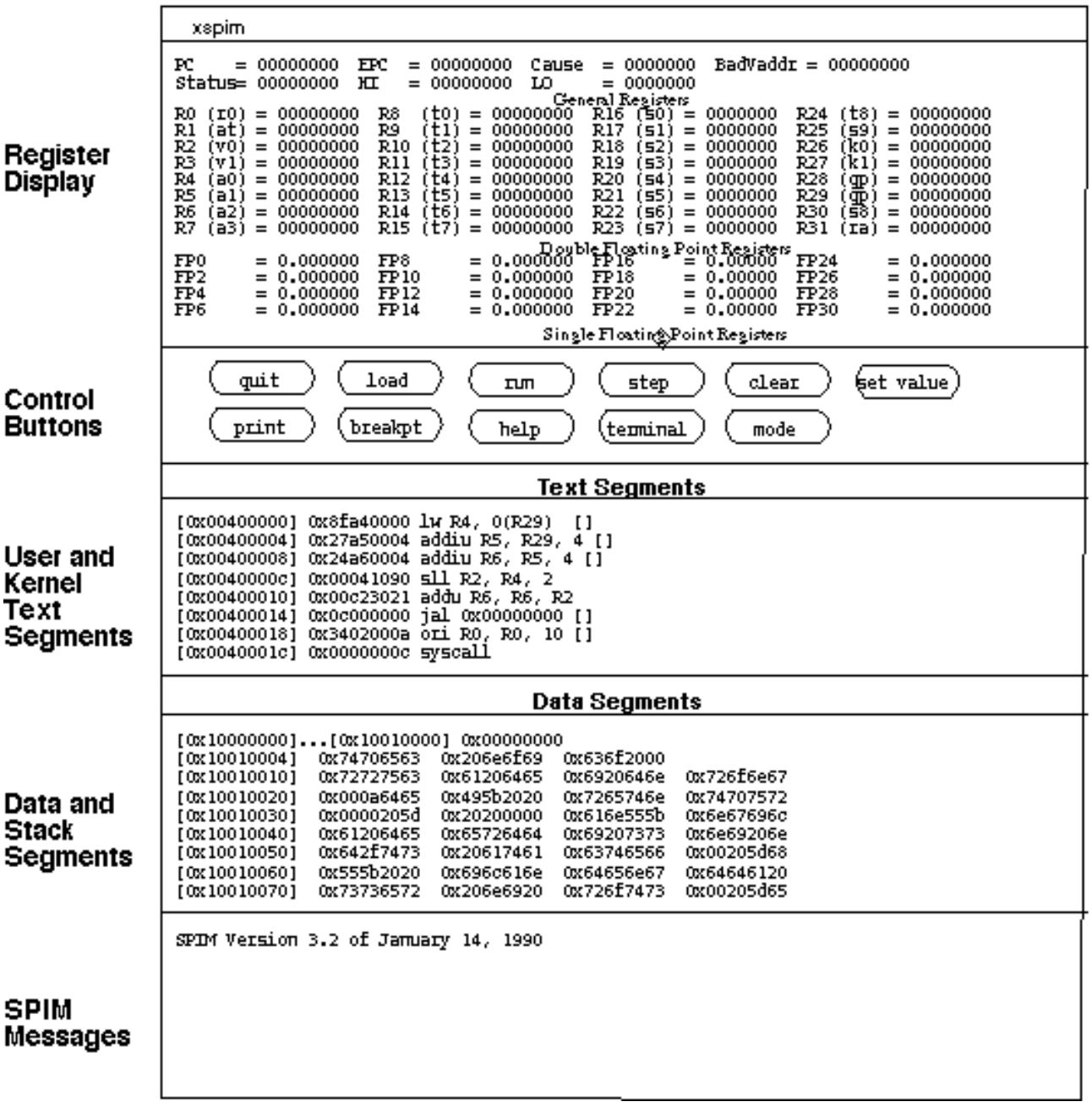


Figura 2.1: GUI di xspim

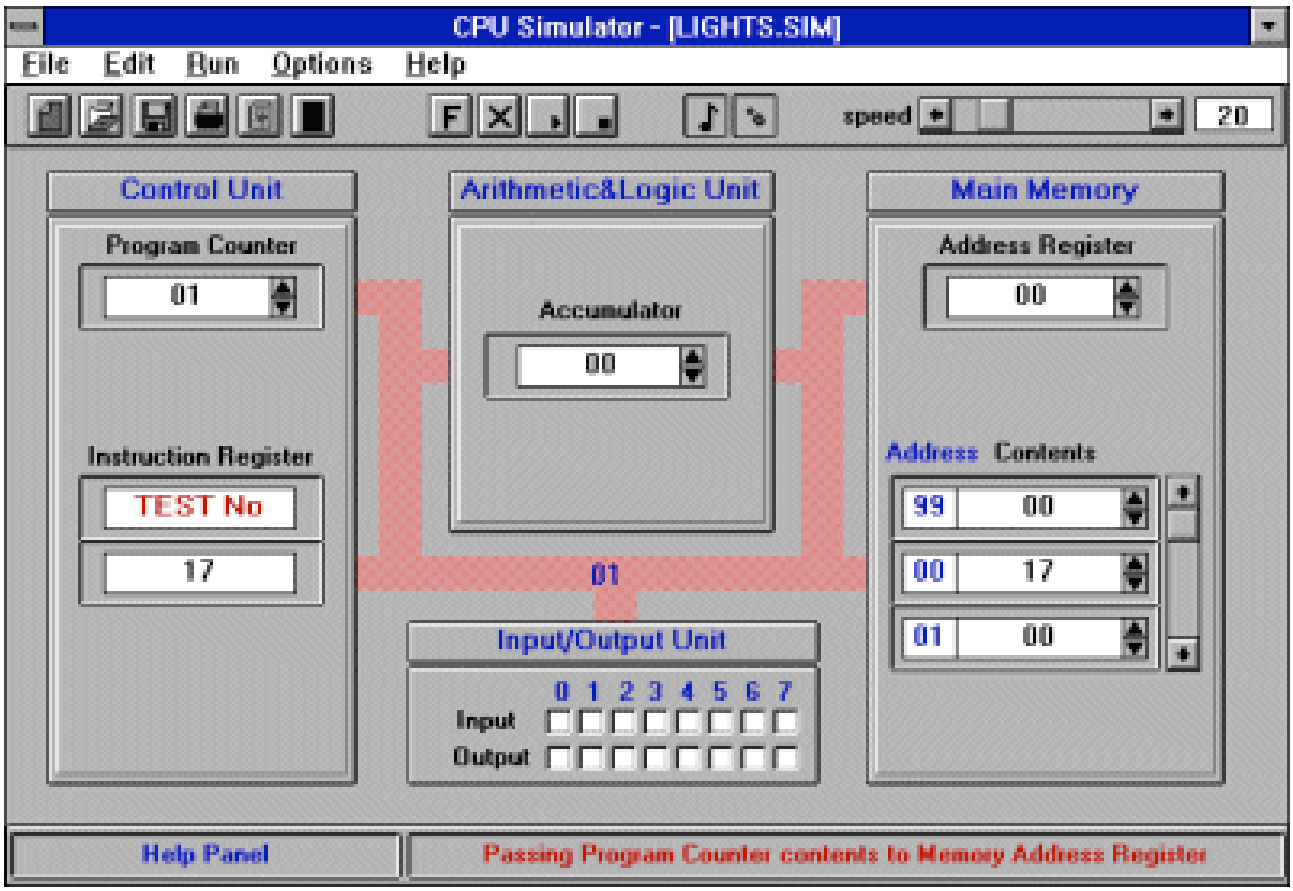


Figura 2.2: GUI di CPU Simulator

caratteristica interessante sta nel fatto che è compatibile con Windows 3.1 e alcuni vecchi Macintosh.

Attualmente, il software è reperibile al seguente URL:

http://www.cs.wisc.edu/ larus/spim.html

2.2 CPU Simulator

Questo prodotto commerciale simula una CPU, mostrandone in dettaglio la struttura interna così articolata: l'unità di controllo, la ALU, la memoria principale e le porte di I/O. Il simulatore visualizza in dettaglio come funziona ogni parte del ciclo fetch-decode-execute. Si può osservare (fig. 2.2) come la CPU prelevi (fetch) l'istruzione, la interpreti (decode), estragga dati addizionali, se necessario, ed effettui l'operazione (execute) come ad esempio dei calcoli. La semplicità e il livello di dettaglio della CPU permette un facile apprendimento di come il processore sia in grado di eseguire programmi, talvolta anche molto complessi.

Attualmente, il software è reperibile al seguente URL:

http://www.spasoft.co.uk/cpusim.html

2.3 Very Simple CPU Simulator

Questo simulatore, implementato come applet java (fig. 2.3), è utilizzato per insegnare la progettazione di microprocessori. Il software permette all'utente di simulare il flusso dei dati all'interno della CPU, passando per le fasi di fetch, decode ed exeute. Vengono impiegate delle animazioni per illustrare il flusso dei dati tra le componenti ed evidenzia i segnali di controllo all'interno della CPU. L'utente ha anche la capacità di simulare l'unità di controllo sia hard-wired che microcoded. Il simulatore utilizza una memoria abbastanza contenuta, composta solamente da 64 byte. Ovviamente una memoria simile è sufficiente per analizzare il funzionamento interno della CPU, ma può non essere sufficiente se si vogliono sviluppare piccoli programmi per apprendere il linguaggio assembly.

Attualmente, il software è reperibile al seguente URL:

http://media.pearsoncmg.com/aw/aw_carpinel_compsys_1/vscpu/index.html

2.4 Von-Neumann-CPU Sim

Il simulatore è una applicazione java in grado di fornire una visuale interna della CPU. In particolare sono visibili i registri. La memoria è visibile in due modalità: interpretata e non interpretata. La prima traduce in stringa le istruzioni presenti in memoria, mentre la seconda ne fornisce il contenuto in formato numerico. La figura 2.4 mostra l'aspetto della finestra principale del software. E' possibile inserire dei breakpoint e regolare la velocità di esecuzione: in questo modo risulta più semplice comprendere il funzionamento del programma. Infine, si può informare il simulatore sul punto di inizio e il punto di fine dell'applicazione da interpretare.

Attualmente, il software è reperibile al seguente URL:

http://homepage.ruhr-uni-bochum.de/Daniel.Reinert/Software/von_Neumann.htm

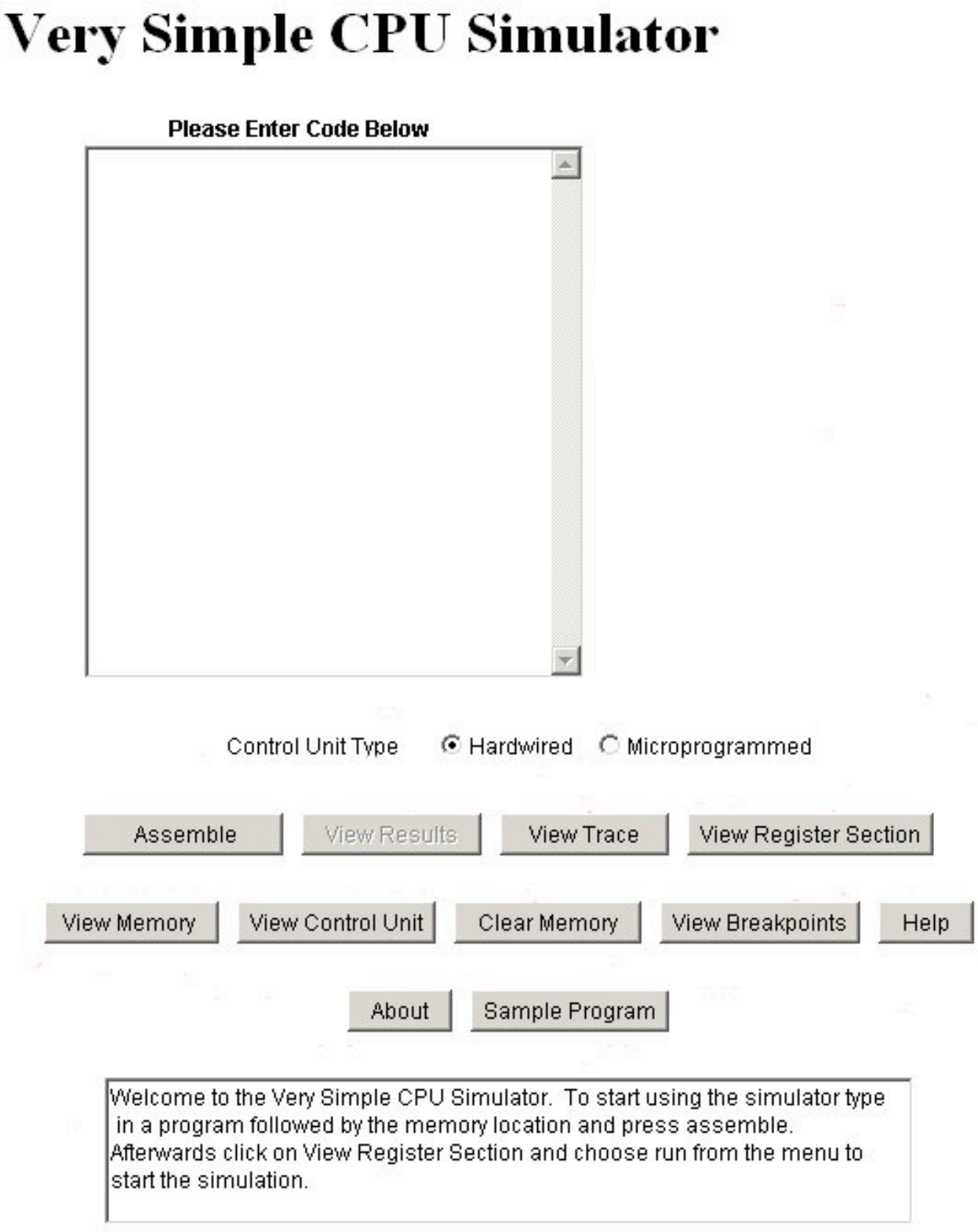


Figura 2.3: GUI di Very Simple CPU Simulator

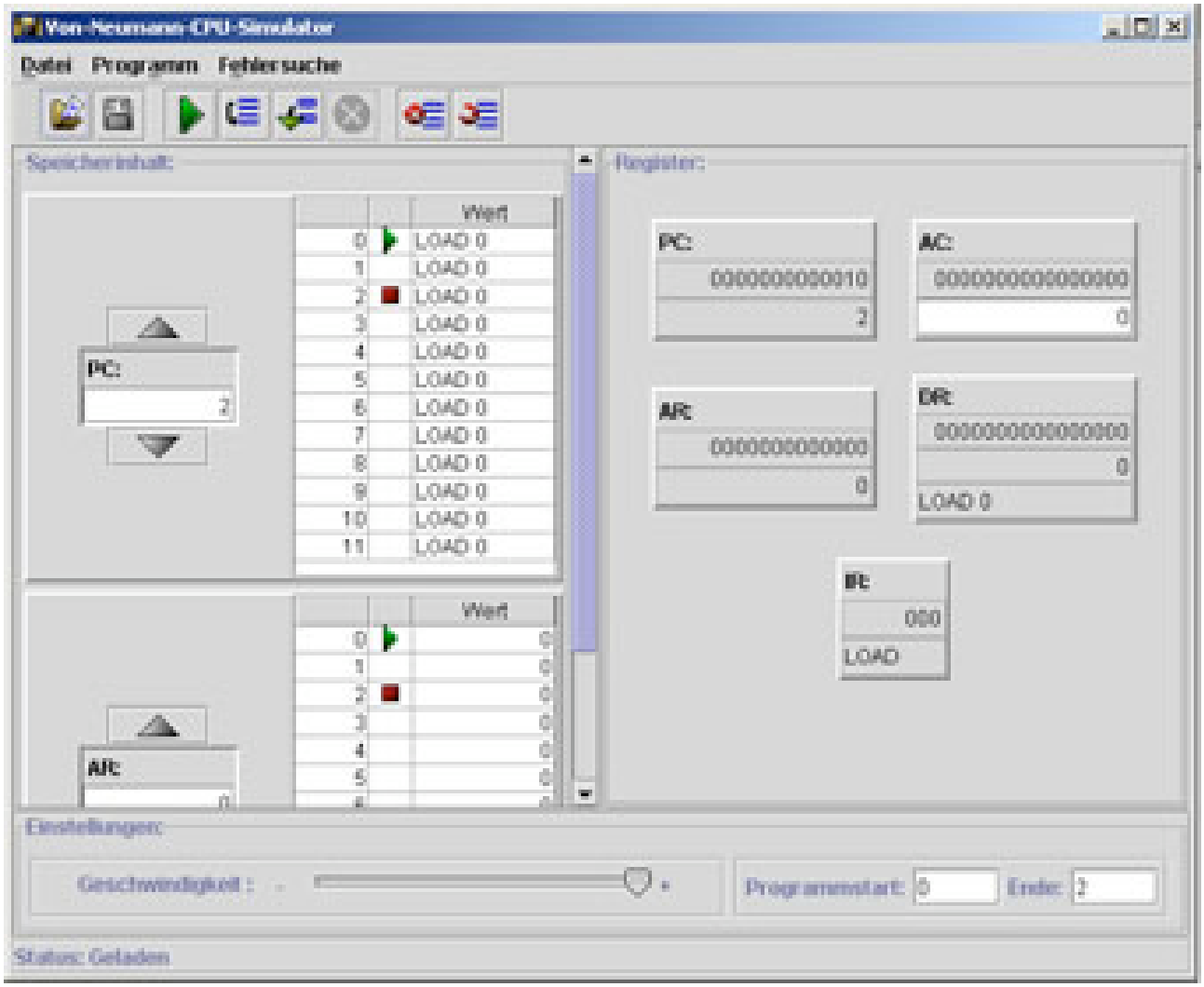


Figura 2.4: GUI di Von Neumann CPU Sim

Parte II

Specifica

Lo scopo di questa parte è definire, ad un livello di astrazione opportuno, una semplice CPU che da qui in poi chiameremo *vCPU*. Non si vuole progettare una CPU in grado di competere con le tecnologie moderne: il suo scopo principale è, come visto anche per i software analizzati nella parte I, quello di rendere disponibile ad un pubblico di studenti uno strumento in grado di facilitare l'apprendimento dei concetti base dell'architettura dei calcolatori.

Capitolo 3

Organizzazione di vCPU

In questo capito vengono affrontati tutti gli aspetti architetturali e organizzativi di vCPU. Dapprima si partirà con una visione d'insieme e successivamente si passerà ad analizzare ogni aspetto in dettaglio, come il formato delle istruzioni, l'organizzazione della memoria, i registri, i flag, la rappresentazione degli interi, la ALU e l'I/O.

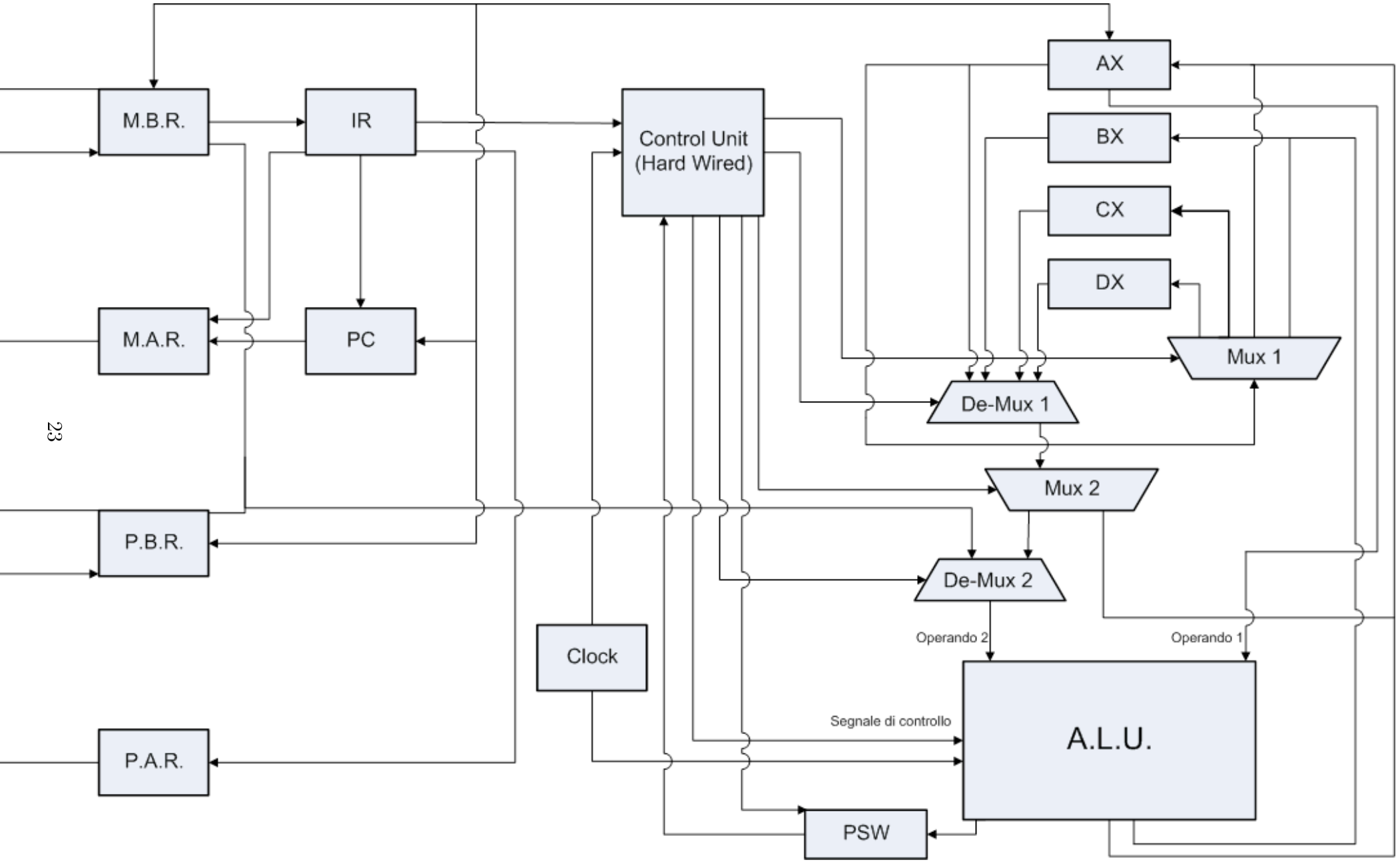
3.1 Visione d'insieme

Per comprendere il funzionamento e le scelte progettuali che hanno caratterizzato i vari aspetti di vCPU, come ad esempio il set di istruzioni, è necessario analizzare la sua struttura. Nella figura 3.1 è presentato lo schema circuitale che mostra le varie componenti di vCPU e le loro relazioni.

A tal proposito si noti che, essendo vCPU un'unità che non esiste nella realtà, il suo schema circuitale non è dato a priori, ma è stato definito sulla base di criteri di semplicità e significatività. In altre parole, si è scelta un'architettura non troppo complicata dal punto di vista circuitale, in modo che potesse essere compresa anche senza approfondite conoscenze di elettronica, ma sufficientemente sofisticata per permettere un reale apprendimento delle problematiche di base dell'architettura dei calcolatori.

Il punto di contatto tra la CPU e le risorse esterne, sono i registri MBR, MAR, PBR e PAR. I primi due si occupano, rispettivamente, di memorizzare il dato in transito da o verso la memoria e di contenere l'indirizzo della cella in questione; gli altri due, invece, si dedicano rispettivamente a memorizzare il dato in transito da o verso le porte ed a contenere il numero di porta al quale si sta riferendo.

l'IR (*Instruction Register*) è un registro adibito a contenere principalmente l'istruzione corrente appena prelevata dalla memoria. Il registro è collegato con la *Control Unit*, per consentire


M.B.R.
IR
Control Unit
(Hard Wired)
AX
BX
CX
DX
M.A.R.
PC
Mux 1
De-Mux 1
Mux 2
P.B.R.
De-Mux 2
Clock
Operando 2
Operando 1
Segnale di controllo
A.L.U.
P.A.R.
PSW

la fase di esecuzione dell'istruzione e quindi la decodifica della stessa. Inoltre, è collegato anche con il registro PC (*Program Counter*), MAR e PAR. per poterli valorizzare con un eventuale indirizzo.

Infine, il registro PC il *Program Counter*, contiene l'indirizzo della cella contenente l'istruzione successiva da eseguire.

La CPU è fornita di ulteriori cinque registri: AX, BX, CX, DX e PSW. Il primo, AX, è un accumulatore. Ciò lascia intendere che si tratta di un registro con funzioni speciali: ad esempio, è sempre un operando e luogo di memorizzazione del risultato in ogni operazione effettuata dalla ALU. Gli altri tre registri, BX, CX, DX, sono a scopo generico: sono utilizzati per contenere informazioni. Ad esempio, possono essere utilizzati per memorizzare risultati intermedi di una lunga computazione.

Successivamente troviamo il registro PSW, adibito a contenere i flag della CPU. Come si avrà modo di approfondire nelle sezioni successive, e in particolare nella sezione 3.4, i flag hanno un ruolo fondamentale nella programmazione: sono in grado di far prendere decisioni sul flusso del programma, basandosi sull'esito di operazioni precedenti.

La *Control Unit* è il cuore della CPU: esso si occupa di decodificare l'istruzione e di eseguire l'operazione associata ad essa. Per permettergli di prendere decisioni in merito alle istruzioni precedenti, il *Control Unit* riceve in ingresso i flag memorizzati nel registro *PSW*. Da come si evince dalla figura 3.1, questa unità è in grado di veicolare il flusso delle informazioni all'interno della CPU, anche per mezzo dei multiplexer e dei de-multiplexer: ad esempio, è in grado di scegliere da quale registro prelevare il dato, per mezzo del de-multiplexer 1, e in quale registro inserire un determinato valore, per mezzo del multiplexer 1. Un'altra funzionalità importante è quella di poter controllare la ALU: tramite un segnale di controllo, è in grado di selezionare l'operazione da effettuare sugli operandi in ingresso. Permette inoltre di stabilire, tramite il de-multiplexer 2, la provenienza del secondo operando: in parcolar modo ha la possibilità di scegliere tra un registro, il PC, l'MBR, il PBR e l'IR. Il compito del multiplexer 2 è quello di permettere ai dati contenuti nei registri di essere o passati come argomento alla ALU o di essere memorizzati dentro il registro AX.

La ALU, acronimo di *Arithmetic and Logic Unit*, permette di effettuare delle operazioni sia logiche che aritmetiche. In particolare, come vedremo nella sezione 3.7, la ALU di cui dispone vCPU ha la possibilità di fare dieci operazioni differenti. Come mostrato in figura, la ALU accetta sempre come operando AX, mentre l'altro operando varia in base al segnale di controllo inviato al de-multiplexer 2. In uscita troviamo invece i flag, memorizzati nel registro PSW, e AX e BX. Come si vedrà nella sezione 3.7, il risultato tipicamente viene inserito solo in AX: BX

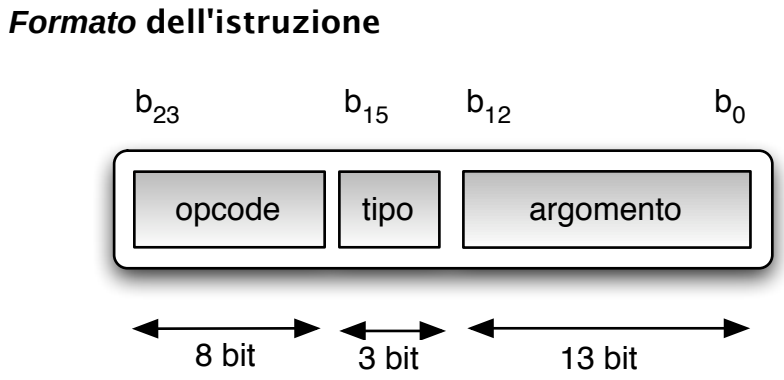


Figura 3.2: Formato dell' istruzione

è utilizzato per operazioni che richiedono maggiore spazio per memorizzare il risultato, come ad esempio l'operazione di moltiplicazione.

3.2 Formato delle istruzioni

Nei casi reali il formato delle istruzioni di una CPU è legato strettamente alla sua architettura e a vincoli tecnologici. Data la natura didattica di vCPU possiamo invece progettarlo in modo relativamente indipendente, dal momento che deve necessariamente essere compatibile con lo schema circuitale utilizzato. Ad esempio, la scelta architetturale di avere uno degli ingressi della ALU sempre costituito dal registro AX, chiaramente rende poco utile progettare un formato delle istruzioni con spazio per due argomenti, dal momento che uno dei due argomenti deve essere obbligatoriamente il registro AX.

Puntando alla semplicità, abbiamo quindi scelto un formato delle istruzioni a 24 bit. Su questi 24 bit, ne dedichiamo 8 per il *codice operativo* (detto anche *opcode*, che specifica quale operazione deve essere effettuata), e 16 bit per specificare l'argomento. Di questi, ne usiamo 3 per il *tipo* (che specifica la natura dell'argomento) e gli altri 13 per *l'argomento* (che specifica con quale operando deve essere effettuata l'operazione). (fig. 3.2). L'argomento di una istruzione può essere infatti di varia natura: potrebbe essere l'indirizzo di una cella contenente l'operando necessario all'istruzione, oppure il valore dell' operando direttamente fruibile dall'istruzione o altro ancora. Per necessità, o per loro natura se vogliamo, alcune istruzioni possono usare argomenti di diversi tipi, altre usano un argomento sempre dello stesso tipo e altre ancora non necessitano della specifica dell'argomento. Da queste differenze è possibile distinguere le istruzioni in due classi:

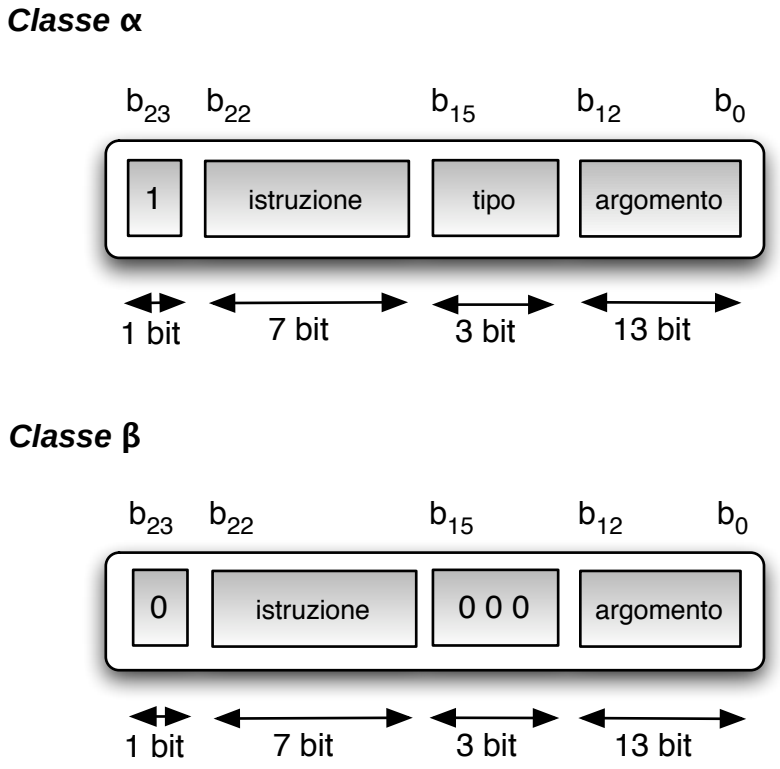


Figura 3.3: Classe α e classe β

- Argomento a tipologia variabile: istruzioni che usano argomenti di diversi tipi.
- Argomento a tipologia prefissata: istruzioni che usano un argomento sempre dello stesso tipo o che non necessitano della specifica dell'argomento.

Per rappresentare questa distinzione vengono introdotti due classi per gli 8 bit dell' opcode: la *classe* α per le istruzioni che usano argomenti di diversi tipi, e la *classe* β per quelle con argomenti a tipologia prefissata. Le classi sono distinte dal bit più significativo, b_{23}: la *classe* α e la *classe* β hanno il bit b_{23} configurato, rispettivamente, a 1 e a 0.

Come si osserva in figura 3.3, la *classe* α permette di rappresentare 2^7 istruzioni, ognuna delle quali può avere un argomento di 2^3 tipi differenti, per un totale di 2^{10} istruzioni effettive; la *classe* β permette di rappresentare solamente 2^7 istruzioni, in quanto i 3 bit di tipo, $b_{15}...b_{13}$, sono configurati sempre a 0.

3.3 Organizzazione della memoria e dimensione delle celle

Normalmente, le celle di memoria (dette anche parole) hanno una dimensione pari alla grandezza del formato delle istruzioni, molto più raramente un suo multiplo o sottomultiplo, per le

complicazioni che ciò causa nella fase di prelievo dell'istruzione. Nel nostro caso la scelta è per semplicità di 24 bit. Il numero di parole costituenti la memoria è anch'esso legato al formato dell'istruzione.

Le parole della memoria sono associate univocamente a dei valori numerici contigui, crescenti a partire da zero, chiamati **i**ndirizzi. Nel caso specifico, quindi, la memoria contiene 2^{12} parole, poiché abbiamo adottato la codifica del complemento a 2 per rappresentare gli interi e abbiamo i 13 bit dell'argomento per contenere l'indirizzo. Inoltre, un indirizzo si definisce più *basso* rispetto ad un altro se viene numericamente prima; si definisce invece più *alto* nel caso opposto.

Si vuole far notare che in teoria, grazie all' indirizzamento indiretto, sarebbe possibile fare riferimento a 2^{24} celle distinte, ma che per semplicità sono state vincolate a 2^{12} , come visto in precedenza.

3.4 I Registri

Internamente alla vCPU, sono presenti in totale sette registri:

- **AX**, l'accumulatore;
- **BX, CX, DX**, registri general purpose;
- **PSW**, il registro dei flag.
- **PC**, il program counter;
- **IR**, il registro delle istruzioni;

Ognuno di questi registri ha per semplicità di progettazione una dimensione pari a quella di una parola. AX è l'accumulatore, cioè un registro che costituisce l'operando implicito di alcune istruzioni e la destinazione implicita per la memorizzazione del risultato. I registri BX CX e DX, invece, sono utilizzatati sia per memorizzare temporaneamente informazioni, sia per essere usati come operandi nelle operazioni.

Il PSW, acronimo di *Processor Status Word*, usa ognuno dei bit in modo indipendente per rappresentare la verità o falsità di una proprietà logica di vCPU: al momento attuale ne usa solo 5 su 24, illustrati in dettaglio successivamente.

Il PC, è utilizzato per memorizzare l'indirizzo dell'istruzione successiva da eseguire. A questo registro si possono quindi assegnare degli indirizzi, ad esempio per effettuare un salto verso una cella di memoria che non sia necessariamente la seguente a quella appena eseguita.

Analogamente al registro PSW, il PC non utilizza tutti i 24 bit a disposizione, bensì i 13 meno significativi, necessari e sufficienti a rappresentare un indirizzo.

Si vuole sottolineare che il registro IR non è utilizzabile direttamente dal programmatore, in quanto usato internamente da vCPU per memorizzare l'istruzione corrente.

3.5 Rappresentazione degli interi

Uno degli aspetti principali di una CPU è la codifica adottata per la rappresentazione degli interi. Per semplicità di progettazione, vCPU è attualmente dotata di istruzioni in grado di elaborare solo interi relativi. La codifica adottata per rappresentare questi interi è quella del complemento a 2, in quanto presenta alcuni vantaggi rispetto ad altre notazioni. Questi vantaggi sono principalmente due:

- Ha una sola rappresentazione dello zero;
- Permette di effettuare le operazioni aritmetiche utilizzando la stessa circuiteria in grado di compiere operazioni aritmetiche su interi senza segno.

Un numero intero con segno (o numero relativo) X viene rappresentato dalla sequenza di cifre

$$b_n b_{n-1} ... b_1 b_0$$

dove $b_i \in \{0, 1\}$ tali che

$$-b_n 2^n + b_{n-1} 2^{n-1} + ... + b_1 2^1 + b_0 2^0 = X$$

Si rappresentano in tal modo tutti i numeri relativi che vanno da -2^n a $2^n - 1$ inclusi. A causa di questa definizione, in cui il bit significativo ha un peso negativo, la rappresentazione degli interi con segno richiede di definire a priori la grandezza dell'intervallo di rappresentazione, cioè il numero di bit dedicati alla rappresentazione del numero stesso. Ad esempio, rappresentare il numero -3 con 3 bit dà luogo a $(101)_2$, mentre rappresentandolo con 4 bit, otteniamo $(1101)_2$. D'altro canto, dato un numero rappresentato in complemento a 2 usando N bit se ne può facilmente ottenere la rappresentazione usando $N + K$ bit ($K > 0$) semplicemente aggiungendo di fronte al MSB per K volte il valore del MSB stesso. Ad esempio da $(-3)_{10} = (101)_2$ con 3 bit si passa a $(11101)_2$ con 5 bit. Si osservi, per concludere, che data una stringa di bit, ad esempio 1101, la sua interpretazione non è univoca, ma dipende dalla rappresentazione cui essa fa riferimento. Se si tratta di interi senza segno, essa equivale all'intero $(7)_{10}$, mentre se si

tratta di interi relativi in complemento a 2 usando 4 bit per la rappresentazione, allora equivale all'intero relativo $(-3)_{10}$.

3.6 I Flag

I flag, cioè i bit del registro PSW, sono essenziali per prendere delle decisioni in merito all'esito delle istruzioni eseguite precedentemente, per poter così cambiare il flusso di esecuzione del programma.

I flag si distinguono in *semplici* e *complessi*: i flag *semplici* sono quelli per i quali il meccanismo che ne determina il valore non dipende da quale operazione è stata effettuata, ma solo dallo stato finale della ALU; i flag *complessi* sono quelli per i quali esistono differenti meccanismi che determinano il valore a seconda dell'operazione effettuata. Nel seguito, nella sezione 3.6.1, si affronteranno i primi, mentre i secondi verranno analizzati nella sezione 3.7.1, dopo un'adeguata panoramica sulla ALU di vCPU esaminata nella sezione 3.7.

3.6.1 Flag semplici

. Questa categoria di flag è considerata *semplice* in quanto sono frutto di una computazione logica sul risultato dell'operazione. Segue una descrizione più adeguata:

- **SI**, è a 1 se l'ultima operazione ha generato un risultato in cui il bit più significativo è a 1, 0 altrimenti;
- **ZE**, è a 1 se il risultato dell'ultima operazione è zero, 0 altrimenti;
- **EV**, è a 1 se la somma dei bit a 1 del risultato dell'ultima operazione è even (pari), 0 altrimenti.

È importante notare che ogni flag, sia *semplice* che *complesso*, è semplicemente la rappresentazione di una parte dello stato finale a cui è arrivata la ALU nell'effettuazione dell'ultima operazione. Per quanto riguarda i flag semplici, tale rappresentazione fornisce anche in modo esplicito la loro interpretazione dal punto di vista aritmetico (qualora tale interpretazione sia significativa, vedi sezione 5.2.1).

In sezione 5.2.1 viene spiegato come interpretare il valore dei flag complessi da un punto di vista aritmetico.

3.7 La ALU e le sue funzionalità

La ALU di vCPU ha la capacità di effettuare operazioni sia logiche che aritmetiche. In particolar modo, si possono eseguire operazioni di somma, sottrazione, divisione, moltiplicazione, negazione, modulo, and, or, or esclusivo (xor) e not. La ALU ha quattro ingressi (due di dati, uno di controllo e il segnale di clock) e varie uscite. Come scelta progettuale, il secondo operando e il segnale di controllo sono specificati nell'istruzione, mentre il primo operando è sempre AX. Nella maggior parte delle operazioni, l'uscita va sempre in AX ma, nel caso specifico della moltiplicazione, l'uscita è distribuita in BX e AX. Insieme al risultato escono anche i valori che configurano i flag nel registro PSW.

L'ingresso di controllo serve per specificare l'operazione che deve essere eseguita. I due ingressi degli operandi hanno tanti bit quanti una parola di memoria, mentre il risultato può essere della dimensione o di una parola o di due parole di memoria. In figura 3.6 è visibile la ALU nella sua completezza.

Per poter comprendere la struttura della ALU di vCPU, nel seguito verrà descritta in modo incrementale, tale da arrivare alla sua comprensione integrale per passi di complessità crescente.

In prima istanza nella ALU evidenziamo la presenza di un *Adder*, implementato per mezzo di un *Full Adder*, in grado di effettuare delle addizioni come mostrato in figura 3.4. Gli operandi dell'operazione vengono inizialmente memorizzati nei registri interni alla ALU, B0 e B1, dove vengono rispettivamente memorizzati il valore di AX e l'operando specificato nell'istruzione. La presenza di un circuito di complementazione a 2, indicato nella figura come *Complementor*, permette inoltre di eseguire le sottrazioni.

Il *Governor* è un modulo in grado di decodificare l'opcode in input alla ALU e di attivare la circuiteria per effettuare l'operazione richiesta. In particolare, quando l'opcode identifica una sottrazione, il *Governor* attiva il *Complementor* che trasforma il secondo operando nel suo complemento a 2 e rende così possibile effettuare la sottrazione dal primo operando mediante l'*Adder*. Ad esempio, per eseguire $(3)-(5)$ la ALU esegue $(3)+(\textit{complemento a 2 di } 5)$, ovvero $(0011)_2 + (1011)_2 = (1110)_2$.

Al termine dell'operazione, il risultato viene memorizzato nel registro R0, da dove vengono prelevati i valori necessari per il calcolo dei *flag semplici*. In particolare, il flag SI coincide con il bit più significativo in R0, ZE con l'AND di tutti i bit in R0 negati e l'EV con lo XOR negato di tutti i bit in R0. Per quanto riguarda i flag complessi dell'*Adder* si rimanda alla sezione 3.7.1.1. Il registro R0 è inoltre connesso ad AX.

Ad un successivo livello di complessità, evidenziamo nella ALU (vedi fig 3.5) il *Multiplier*,

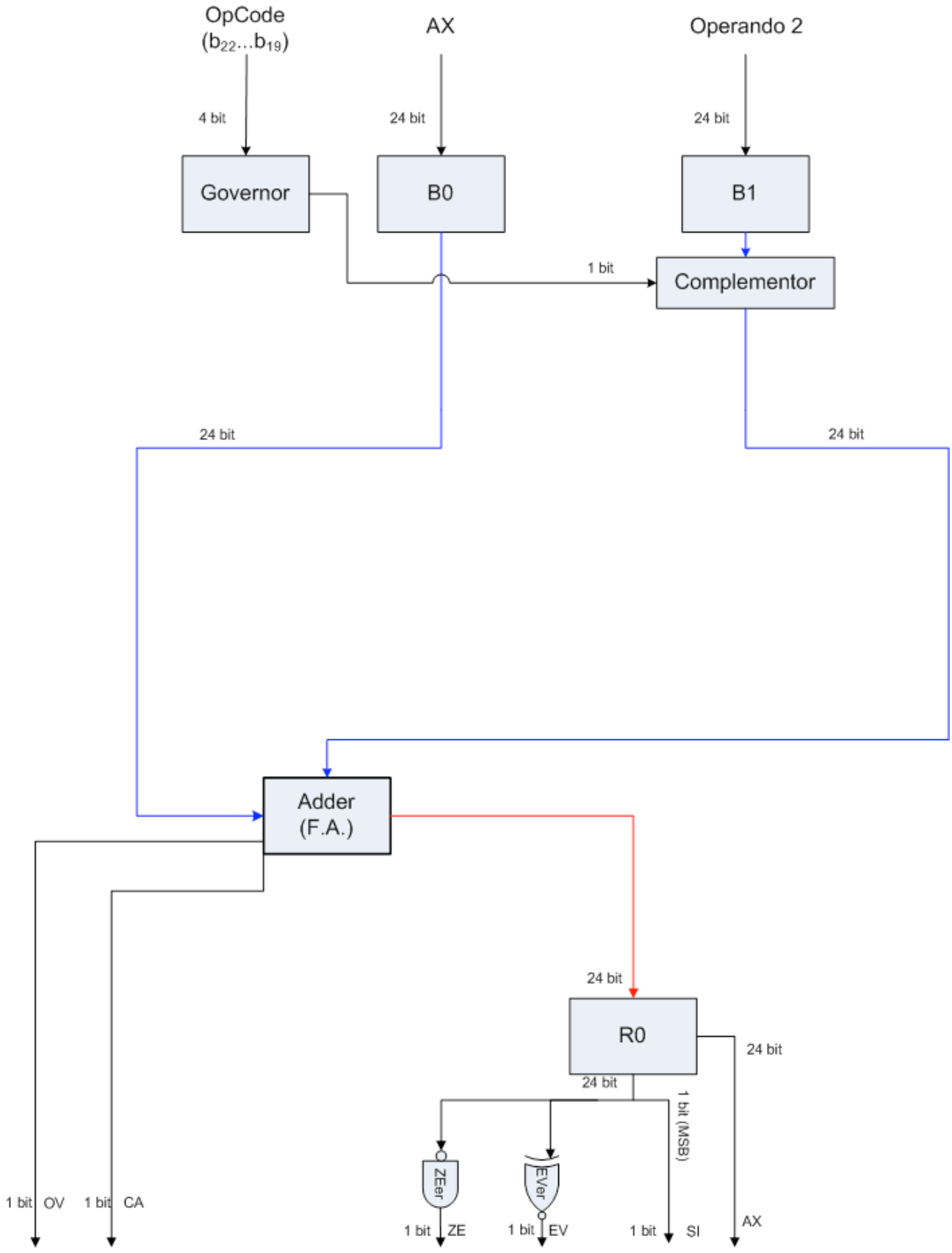


Figura 3.4: Una semplice ALU in grado di effettuare somme e sottrazioni.

modulo in grado di effettuare le operazioni di moltiplicazione. L'aggiunta di un secondo modulo, comporta l'introduzione di due de-multiplexer, guidati dal *Governor*, in grado di indirizzare gli operandi verso il modulo d'interesse. La natura dell'operazione di moltiplicazione comporta l'introduzione di un secondo registro per la memorizzazione del risultato: il registro R1. A differenza di altri moduli, il *Multiplier* utilizza sempre il registro R0 e il registro R1, per memorizzare rispettivamente la parte meno significativa del risultato e quella più significativa. Conseguentemente all'introduzione del nuovo registro, la logica per il calcolo dei flag semplici cambia: nel calcolo del flag ZE ed EV vengono considerati anche i bit in R1 e per il calcolo di SI viene utilizzato un de-multiplexer, pilotato dal *Governor*, in grado di selezionare il bit più significativo che andrà a valorizzare il flag. In particolar modo, se l'operazione da effettuare è una moltiplicazione, verrà utilizzato il bit più significativo in R1, altrimenti quello in R0. Come l'*Adder* anche il *Mutiplier* genera dei flag complessi secondo un criterio che viene descritto nella sezione 3.7.1.2. Per poter avere solo un bit in uscita sia per il flag OV che per CA, sono state introdotte due porte OR in grado di riunire i segnali uscenti dai moduli *Adder* e *Multiplier* in uno solo. Tenendo presente che le porte OR hanno output 1 nel caso in cui almeno uno degli ingressi valga 1 e che grazie ai de-multiplexer collegati con gli operandi i moduli *Adder* e *Multiplier* non possono essere attivi contemporaneamente, i flag OV e CA in uscita dalla ALU hanno sempre il valore generato dal modulo attivato.

Analogamente ai flag complessi, anche il registro R0 riceve in ingresso l'OR di tutte le uscite dei moduli. Come visto prima, grazie ai de-multiplexer, può essere attivo solo un modulo alla volta e il segnale in uscita dal modulo attivato, in OR con tutte le uscite a 0 degli altri moduli, coincide proprio con il risultato dell'operazione richiesta. Conseguenza di ciò è che non si ha la necessità di avere dei segnali di abilitazione per ogni modulo.

Infine, la descrizione della struttura della ALU di vCPU, viene completata con la presenza degli altri moduli necessari per implementare il resto delle operazioni. La figura 3.6 mostra lo schema circuitale completo dell'interno della ALU e di seguito riportiamo l'elenco completo dei moduli presenti:

Adder effettua la somma di due interi relativi.

$$R0 = Operando1 + Operando2$$

Complementor restituisce l'opposto di un intero relativo.

$$R0 = -Operando1$$

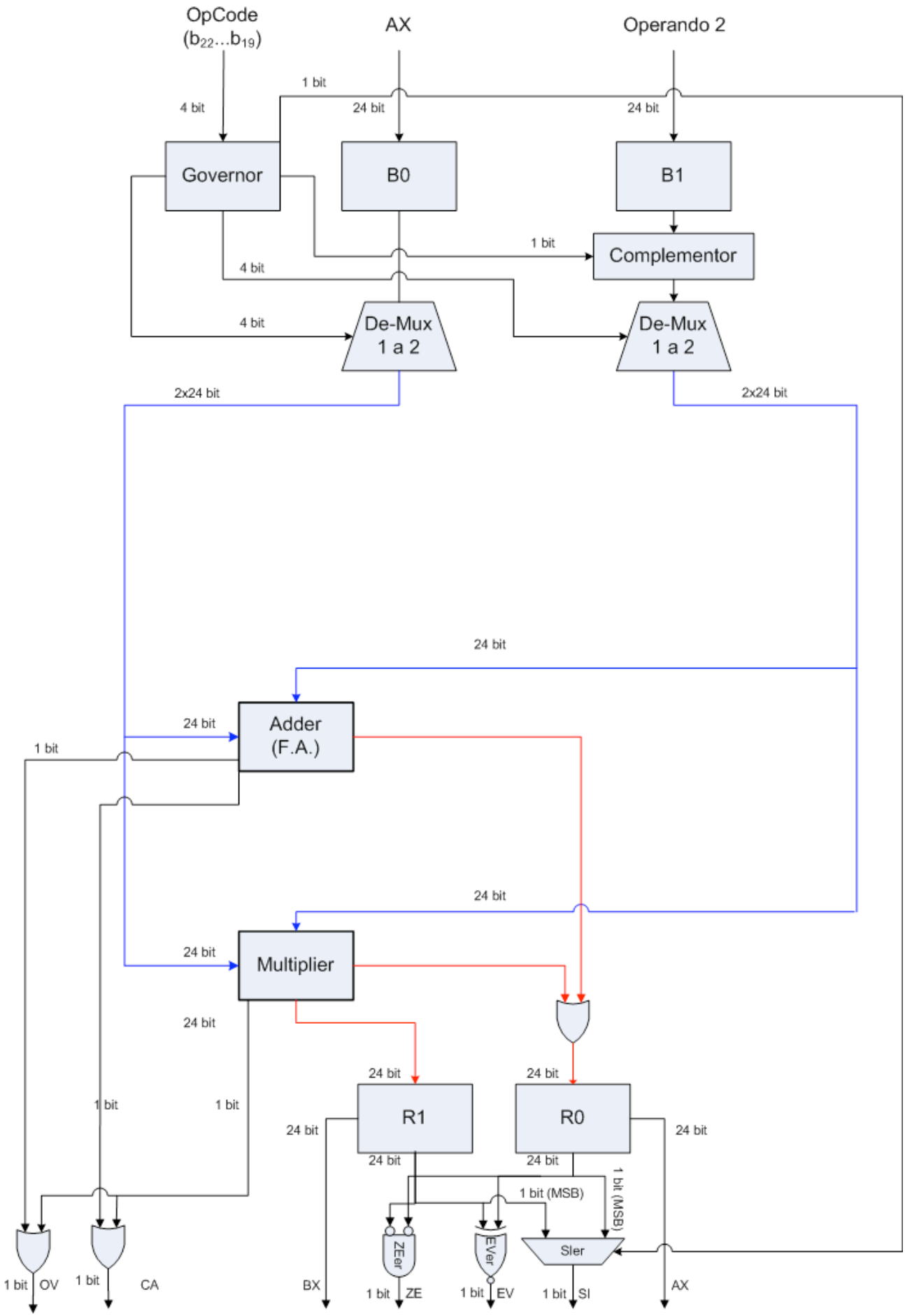


Figura 3.5: Una semplice ALU in grado di effettuare somme, sottrazioni e moltiplicazioni.

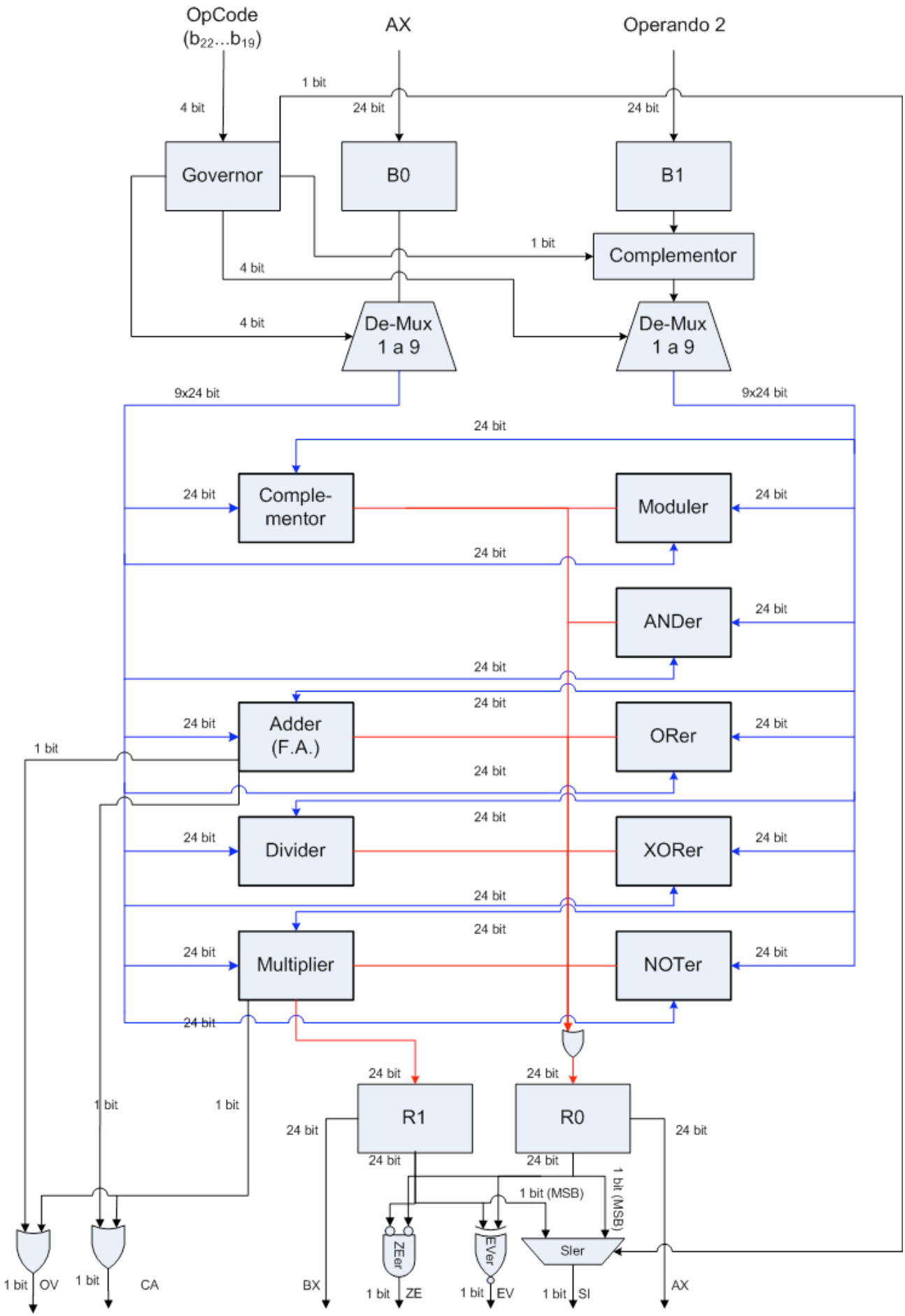


Figura 3.6: La ALU di vCPU.

Multiplier effettua il prodotto di due interi relativi.

$$R1, R0 = Operando1 * Operando2$$

Divider effettua la divisione tra due interi relativi.

$$R0 = Operando1/Operando2$$

Moduler effettua il modulo della divisione tra due interi relativi.

$$R0 = Operando1\ mod\ Operando2$$

ANDer effettua l'AND logico bit a bit tra due interi relativi.

$$R0 = Operando1\ AND\ Operando2$$

ORer effettua l'OR logico bit a bit tra due interi relativi.

$$R0 = Operando1\ OR\ Operando2$$

XORer effettua l'OR esclusivo bit a bit tra due interi relativi.

$$R0 = Operando1\ XOR\ Operando2$$

NOTer effettua il NOT logico bit a bit di un intero relativo.

$$R0 = \sim Operando1$$

È di particolare interesse esaminare con maggior dettaglio la struttura del *Multiplier*: in figura 3.7 sono visibili i moduli e le connessioni che lo compongono. Il *Multiplier* si basa sostanzialmente su un moltiplicatore S&A (Shift and Add) per numeri interi di 24 bit estesi a 48 bit. Il risultato che si ottiene moltiplicando gli operandi viene memorizzato nei registri R1 (la parte più significativa) e R0 (la parte meno significativa). Come è possibile vedere in figura 3.7, gli operandi vengono estesi utilizzando il proprio bit di segno. Quando viene fatta la trasformazione dell' operando in complemento a 2 da 24 a 48 bit, viene impiegato un *replicatore*, in grado di configurare tutti i bit aggiunti nell'estensione con lo stesso valore del bit più significativo dell' operando. Nella figura 3.7 si può notare come il bit di segno dei registri M00 ed M10 vengano replicati, rispettivamente, nei registri M01 ed M11. Una volta effettuata questa trasformazione, gli operandi a 48 bit possono essere passati come argomento al moltiplicatore S&A, ottenendo così un risultato corretto in complemento a 2.

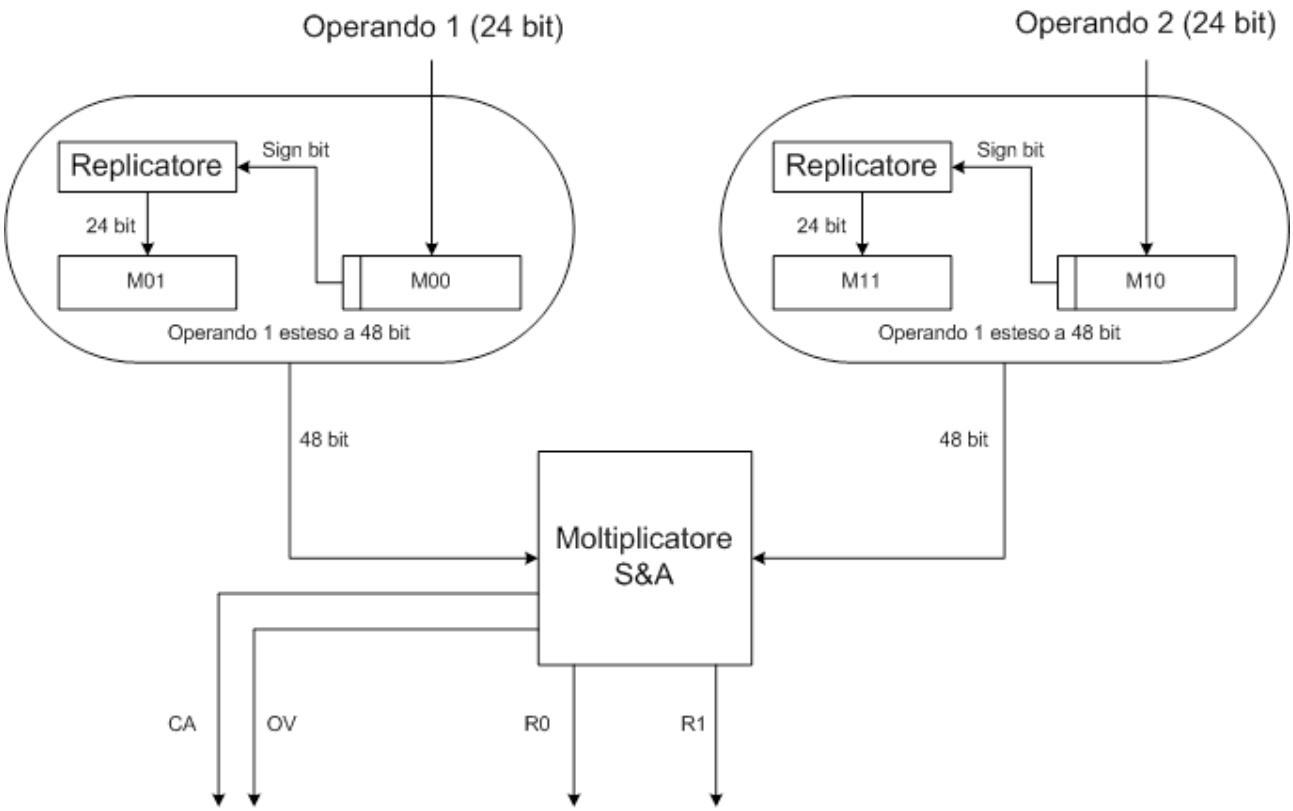


Figura 3.7: l'interno del Multiplier

In figura 3.8 è mostrato l'interno del moltiplicatore S&A. Si trovano i registri *MLND*, *MLER* e *PROD* che memorizzano, rispettivamente, il moltiplicando, il moltiplicatore e il prodotto. Gli *Shifter* hanno lo scopo di shiftare (verso sinistra il moltiplicando e verso destra il moltiplicatore), introducendo degli zeri, i registri *MLND* e *MLER*. Come si può notare, uno dei due ingressi dell'adder è il registro *PROD*, utilizzato per memorizzare le somme dei prodotti parziali e che costituisce quindi anche il registro di uscita dell'adder. Il valore memorizzato in *PROD*, alla fine della computazione, rappresenta il prodotto tra il registro *MLND* e *MLER*. In particolar modo, la computazione ha termine quando il registro *MLER* contiene 0.

Infine, in figura 3.9, è visibile il diagramma di flusso che mostra come il moltiplicatore S&A effettua la moltiplicazione.

3.7.1 Flag complessi

Questi flag, a differenza di quelli semplici, vengono configurati anche in base a criteri legati alle operazioni: in particolar modo le operazioni coinvolte sono l'addizione e la moltiplicazione. Si rimanda alla sezione 5.2.1 per l'interpretazione a livello aritmetico di questi flag.

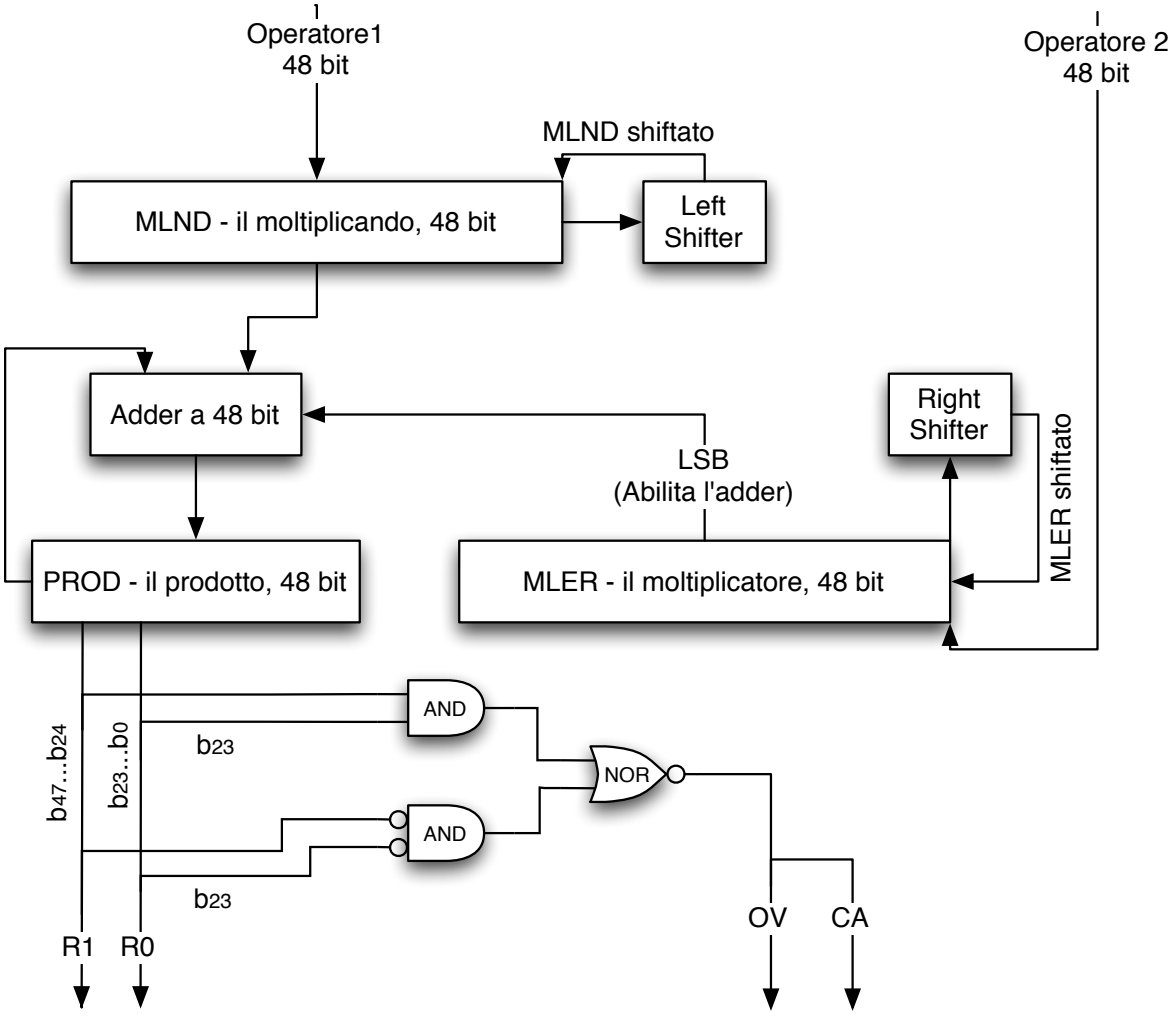


Figura 3.8: l'interno del moltiplicatore S&A a 48 bit

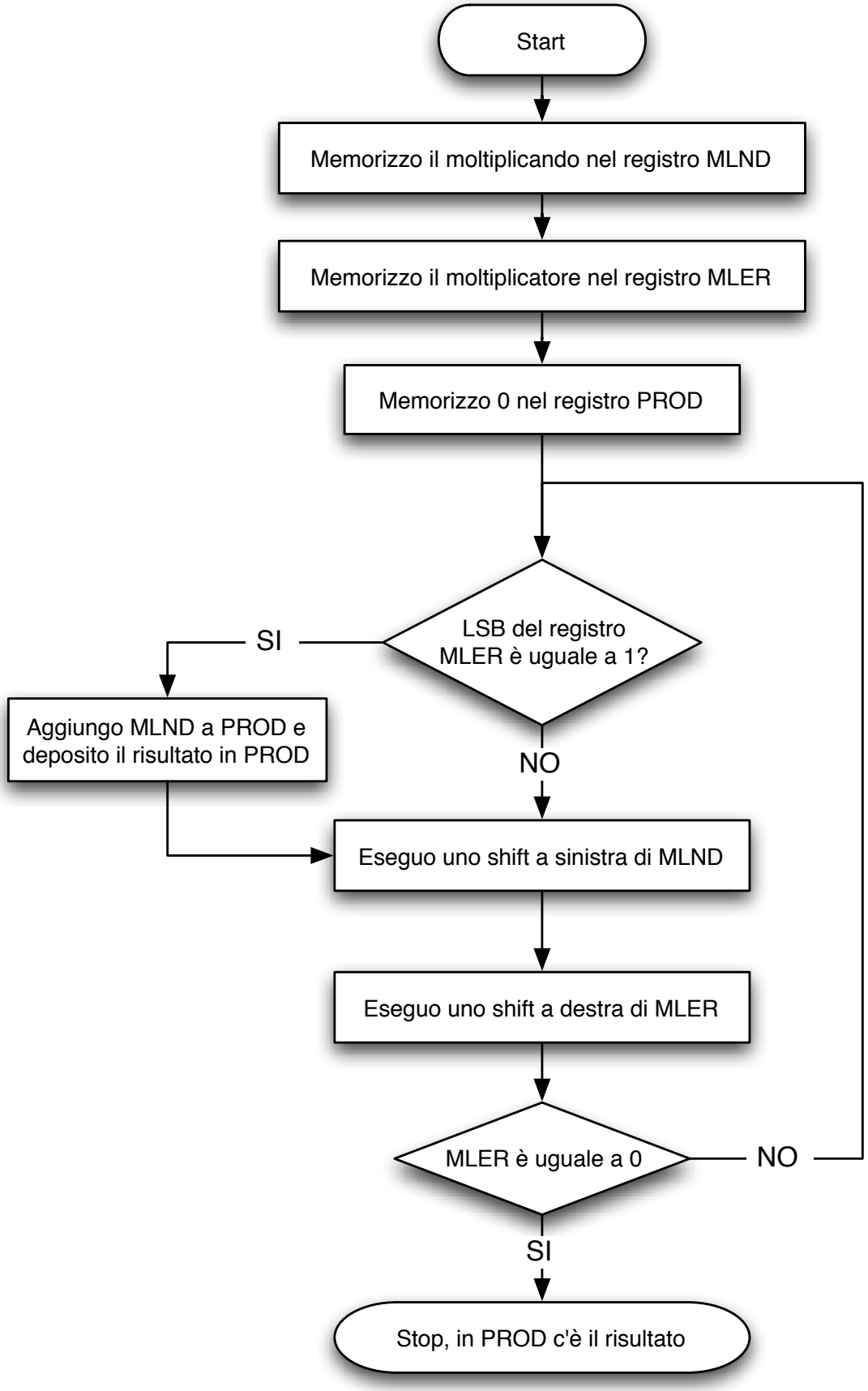


Figura 3.9: Flow Chart del moltiplicatore S&A

3.7.1.1 Adder

CA : è a 1 se il bit di riporto dell' Adder vale 1 nell'ultima operazione, 0 altrimenti.

OV : è a 1 se l'ultima operazione ha generato un risultato il cui bit più significativo differisce da entrambi i bit più significativi degli operandi, 0 altrimenti;

3.7.1.2 Multiplier

CA : viene configurato secondo lo stesso criterio con cui questa operazione configura OV.

OV : viene configurato a 0 quando i bit in R1 e il bit più significativo in R0 sono tutti uguali tra di loro, 1 altrimenti.

3.8 Ingresso/Uscita

Ogni architettura che si possa definire funzionale necessita di una modalità di comunicazione con l'esterno. Questa esigenza nasce anche dalla semplice necessità di voler importare dei dati da elaborare o di esportare quelli elaborati ad un qualche altro sottosistema elettronico. Per permettere questo flusso di dati, vCPU è stata dotata dell' I/O isolato, un meccanismo di I/O realizzato utilizzando istruzioni ad hoc per il flusso di dati su delle porte. L'insieme delle porte usate nell'I/O isolato fa parte di una sorta di memoria disgiunta da quella contenente il programma e i dati.

Analogamente alle decisioni prese per la memoria in sezione 3.3, anche le porte sono indirizzate da 2^{12} valori distinti: ciò comporta che vCPU dispone di 4096 porte di I/O per comunicare con l'esterno.

A differenza di altre CPU, in vCPU non si dispone di un sistema di interrupts per permettere di eseguire un'applicazione in modo più efficiente: l'unico modo per comunicare con un dispositivo è utilizzare la tecnica del *Polling*. Questa tecnica affronta la comunicazione con un dispositivo effettuando letture continue su delle porte specifiche, normalmente chiamate porte di controllo. Questa tecnica ha senza dubbio il difetto di essere inefficiente, ma sicuramente rende più chiara e semplice la comunicazione con un dispositivo.

Capitolo 4

Il formato delle istruzioni di vCPU

In questo capitolo viene presentato e discusso in dettaglio il formato delle istruzioni e i criteri usati per assegnare gli opcode alle varie operazioni.

4.1 Argomenti e operandi

Nel Capitolo 3 avevamo informalmente introdotto i termini *argomento* e *operando*. Ne forniamo adesso una definizione:

Argomento : è il valore dei bit $b_{12}...b_0$ presenti nell'istruzione;

Operando : è il valore effettivo utilizzato dall'operazione intrapresa dall'istruzione.

Come già accennato nel Capitolo 1, le istruzioni hanno un argomento che può essere di diversa natura. Nella vCPU ci sono 5 tipi di argomento:

immediato : l'operando coincide con l'argomento.

registro : l'operando è il valore contenuto nel registro specificato come argomento.

indirizzo diretto : l'operando è il valore contenuto nella cella di memoria specificata come argomento.

indirizzo indiretto : l'operando è il valore contenuto nella cella di memoria il cui indirizzo è contenuto nella cella di memoria specificata come argomento.

registro indiretto : l'operando è il valore contenuto nella cella di memoria il cui indirizzo è contenuto nel registro specificato come argomento.

Come si vede dagli esempi, il prefisso '@' indica che il valore effettivo usato dall'istruzione non corrisponde direttamente all'argomento, ma si ottiene da esso. Esempi di questi cinque tipi di argomenti, per una ipotetica istruzione SOMMA_ACC che somma il valore dell'accumulatore a quello specificato dall'argomento sono:

- SOMMA_ACC 100
- SOMMA_ACC @100
- SOMMA_ACC @@100
- SOMMA_ACC BX
- SOMMA_ACC @BX

4.2 Criterio di assegnazione degli opcode alle istruzioni

Sono stati assegnati alle varie istruzioni i codici di operazione in modo razionale e in modo tale che sia facile ricordare quale sia il codice operativo di una data istruzione. Nelle sezioni seguenti verrà descritto il criterio di assegnazione degli opcode in relazione sia alla classe α che alla classe β.

4.2.1 Istruzioni nella classe α (tipologia variabile)

4.2.1.1 Codifica delle operazioni

Come abbiamo già accennato in precedenza, la classe α (bit $b_{23} = 1$) permette di rappresentare 2^7 istruzioni differenti. Queste istruzioni possono essere divise nelle seguenti categorie:

Prodotti : contiene le istruzioni relative a moltiplicazioni e divisioni;

Somme : contiene le istruzioni relative ad addizioni e sottrazioni;

Memorizzazione : contiene le istruzioni che permettono il flusso di dati da e verso la memoria;

Logiche ed altro : contiene le istruzioni relative ad operazioni logiche ed altro.

Dei 7 bit disponibili per codificare l'istruzione dedichiamo quindi i primi due, $b_{22}b_{21}$, per rappresentare la categoria mentre i successivi, $b_{20}...b_{16}$, codificano la specifica operazione all'interno della categoria come mostrato in figura 4.1. In realtà, dato il numero relativamente ridotto di

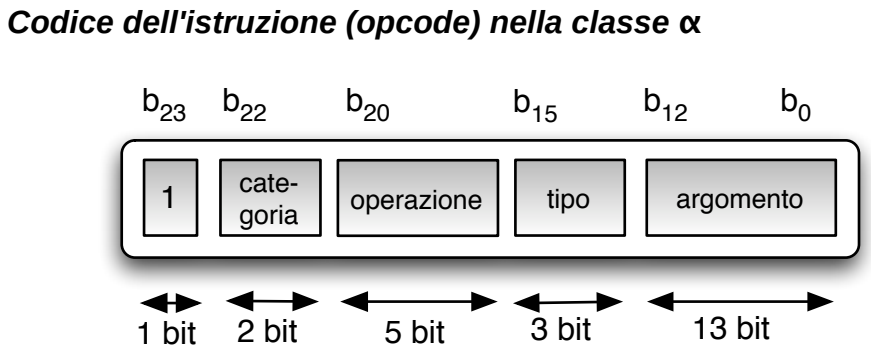


Figura 4.1: Dettaglio B22-B19 sulla classe α

istruzioni appartenenti a questa categoria, i bit dedicati realmente alla codifica di queste sono solamente i bit $b_{20}b_{19}$ e i bit $b_{18}...b_{16}$ sono sempre configurati a 0.

I bit $b_{22}b_{21}$ codificano la categoria come illustrato dalla seguente tabella:

	b_{22}	b_{21}
Prodotti	1	1
Somme	1	0
Memorizzazione	0	1
Logiche ed altro	0	0

Si noti che $b_{22} = 1$ contraddistingue le operazioni aritmetiche. La codifica delle specifiche operazioni per ognuna delle categorie, realizzata mediante i bit $b_{20}b_{19}$, avviene inoltre come illustrato dalle seguenti tabelle.

Prodotti

	b_{20}	b_{19}
DIV	1	1
MUL	1	0

Notare che solo il bit b_{19} contraddistingue le due istruzioni.

Somme

	b_{20}	b_{19}
INC	1	1
DEC	1	0
ADD	0	1
SUB	0	0

Notare che il bit b_{20} è impostato ad 1 per le istruzioni più elementari di somma e sottrazione, cioè l'incremento e il decremento.

Memorizzazione

	b_{20}	b_{19}
ST	1	1
LD	0	0

Le restanti configurazioni in questa categoria non sono assegnate.

Logiche ed altro

	b_{20}	b_{19}
AND	1	1
OR	1	0
XOR	0	1
MOD	0	0

Notare che entrambi i bit b_{20} e b_{19} sono a 0 per l'istruzione di modulo, l'unica a non essere classificabile come istruzione logica.

4.2.1.2 Codifica dei Registri

In molte istruzioni si ha la possibilità di specificare un registro come argomento. Per fare ciò è necessario assegnare una codifica ad ognuno di esso. La codifica adottata da vCPU è la seguente:

	b_{12}	$b_{11}...b_3$	b_2	b_1	b_0
AX	0	0...0	0	0	0
BX	0	0...0	0	0	1
CX	0	0...0	1	1	0
DX	0	0...0	1	1	1
PC	0	1...1	1	1	1

Si noti che il registro PSW non ha una codifica in quanto non può essere usato come operando di una istruzione (vedi sez. 3.4).

4.2.1.3 Codifica del tipo di argomento

Come già discusso, le istruzioni nella classe α possono avere argomenti di tipi differenti. La tipologia dell'argomento viene specificata dai 3 bit di *tipo*, $b_{15}...b_{13}$, come segue. Il bit b_{15} identifica la natura dell'argomento dell'istruzione: se è impostato ad 1 si indica che l'argomento è un nome di registro; altrimenti, se è a 0, si indica che è un numero naturale. Gli altri due bit, $b_{14}b_{13}$, assumono di conseguenza significati diversi in base alla natura dell'argomento. Nel primo caso, in cui l'argomento è un registro, i bit $b_{14}b_{13}$ indicano:

$b_{14} = 0$,$b_{13} = 0$: il registro contiene l'operando (argomento di tipo *registro*).

$b_{14} = 0$,$b_{13} = 1$: il registro contiene l'indirizzo della cella in cui è contenuto l'operando (argomento di tipo *registro indiretto*).

Nel secondo caso, invece, i bit $b_{14}b_{13}$ indicano:

$b_{14} = 0$,$b_{13} = 0$: il numero è l'operando (argomento di tipo *immediato*): è quindi un numero relativo.

$b_{14} = 0$,$b_{13} = 1$: il numero è l'indirizzo della cella che contiene l'operando (argomento di tipo *indirizzo diretto*): in questo caso si tratta invece di un numero relativo positivo.

$b_{14} = 1$,$b_{13} = 0$: il numero è l'indirizzo della cella che contiene l'indirizzo della cella contenente l'operando (argomento di tipo *indirizzo indiretto*): anche in questo caso è un numero positivo.

4.2.2 Istruzioni nella classe β (tipologia prefissata)

Con la classe β (bit $b_{23} = 0$) abbiamo a disposizione 7 bit per codificare i codici di operazione e i bit di *tipo*, $b_{15}...b_{13}$, sempre configurati a 0. Questo significa che possiamo codificare al più 2^7 istruzioni differenti. Il set di istruzioni di vCPU si può dividere in quattro categorie:

I/O : contiene le istruzioni usate per effettuare operazioni di I/O;

Aritmetico-logiche : sono le operazioni di tipo aritmetico o logiche con operando a tipologia prefissata;

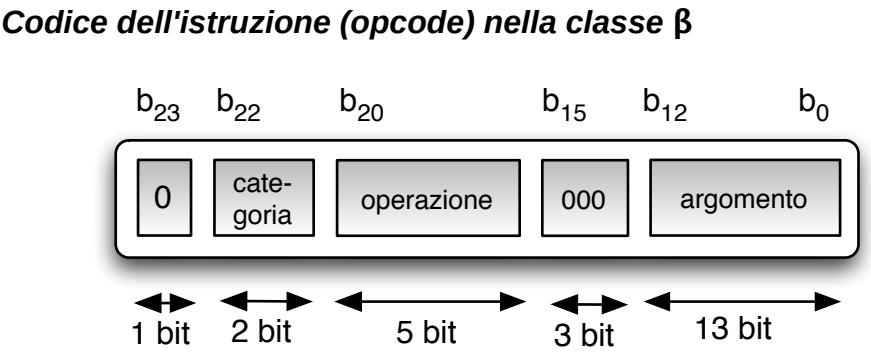


Figura 4.2: Dettaglio B22-B16 sulla classe β

Salto : troviamo in questa categoria tutte le istruzioni di salto sia condizionato dal valore dei flag che non;

Altro : in questa categoria ricadono tutte quelle istruzioni che non possono essere inserite nelle categorie precedenti.

Dei 7 bit disponibili per codificare l'istruzione ne dedichiamo quindi 2 per rappresentare la categoria di appartenenza dell'istruzione, mentre gli altri 5 codificano la specifica operazione all'interno della categoria (figura 4.2). I bit $b_{22}b_{21}$ codificano la categoria nel seguente modo:

	b_{22}	b_{21}
I/O	1	1
Aritmetico-logiche	1	0
Salto	0	1
Altro	0	0

La codifica delle operazioni specifiche di ogni categoria, realizzata mediante i bit $b_{20}...b_{16}$, avviene come specificato dalle seguenti tabelle

I/O

	b_{20}	b_{19}	b_{18}	b_{17}	b_{16}
IN	0	0	0	0	1
OUT	0	0	0	0	0

Le due sole istruzioni di questa categoria sono distinte dal valore di b_{16}. Per entrambe queste istruzioni, l'argomento è sempre un numero naturale che identifica il numero di porta sulla quale operare. Le ulteriori configurazioni in questa categoria non sono assegnate.

Aritmetico-logiche

	b_{20}	b_{19}	b_{18}	b_{17}	b_{16}
NOT	0	0	0	0	1
NEG	0	0	0	0	0

Le due sole istruzioni di questa categoria sono distinte dal valore di b_{16}. Come si vedrà meglio in seguito NOT è l'operazione booleana che complementa l'argomento bit per bit, mentre NEG è l'operazione aritmetica che nega l'argomento. Le ulteriori configurazioni in questa categoria non sono assegnate.

Salto

Rispetto alle altre categoria, quella di "Salto" conta un numero di istruzioni più alto: per questo motivo è bene suddividere i bit $b_{20}...b_{16}$ in tre parti: la prima parte, composta dal solo bit b_{20}, per identificare il tipo di salto; la seconda, composta dai bit $b_{19}...b_{17}$, per identificare il tipo di test; e la terza, composta dal bit b_{16}, per identificare la modalità del test.

Possiamo distinguere due tipi di salto: quelli condizionati dal valore di un singolo flag e quelli condizionati dal valore contemporaneo di più flag. Questi tipi di test vengono detti, rispettivamente, *semplici* e *complessi*. Si possono distinguere i due tipi di salto guardando il bit b_{20}: se è a 0 si sta codificando un test semplice; se è a 1, invece, si sta codificando un test complesso.

Nel caso di test *semplice* ($b_{20} = 0$) abbiamo le seguenti istruzioni, in base al valore dei bit $b_{19}...b_{16}$:

	b_{20}	b_{19}	b_{18}	b_{17}	b_{16}
Salta sempre	0	0	0	0	0
Salta se CA = 1	0	0	0	1	1
Salta se CA = 0	0	0	0	1	0
Salta se OV = 1	0	0	1	0	1
Salta se OV = 0	0	0	1	0	0
Salta se SI = 1	0	0	1	1	1
Salta se SI = 0	0	0	1	1	0
Salta se EV = 1	0	1	0	0	1
Salta se EV = 0	0	1	0	0	0
Salta se ZE = 1	0	1	0	1	1
Salta se ZE = 0	0	1	0	1	0

Come si vede, l'effetto del bit b_{16} nei test semplici è correlato alla condizione di salto. Se b_{16} è configurato a 1, allora il salto avverrà qualora il flag relativo avrà valore 1: analogamente, se è configurato a 0, allora il salto avverrà solo se il flag relativo è configurato a 0.

Nel caso dei test complessi ($b_{20} = 1$), invece, la situazione è la seguente:

	b_{20}	b_{19}	b_{18}	b_{17}	b_{16}
Salta se CA=0 e ZE=0	1	0	0	0	0
Salta se CA=1 e ZE = 1	1	0	0	1	1
Salta se ZE = 0 e SI = OV	1	0	1	0	0
Salta se SI = OV	1	0	1	0	1
Salta se SI != OV	1	1	1	1	0
Salta se ZE = 1 e SI != OV	1	1	1	1	1

Bisogna notare che in realtà esistono dei test complessi che coincidono con quelli semplici: in particolar modo, più avanti vedremo due istruzioni di salto complesse, JAE e JB, le cui condizioni coincidono rispettivamente con quelle di JNC e JC. Per mantenere entrambe le coppie di salto senza però compromettere l'unicità della codifica dell'istruzione, vengono assegnate le stesse codifiche ai salti JAE, JNC e a JB, JC.

L'ultima categoria della classe β comprende due sottocategorie: la prima contiene due istruzioni che non possono essere classificate in nessuna categoria analizzata precedentemente, la seconda contiene alcune istruzioni inserite in questa categoria per motivi di opportunità. Nessuna di queste istruzioni ha un argomento.

In particolare, la prima sottocategoria (contraddistinta da $b_{20} = b_{19} = 0$) contiene le istruzioni elencate di seguito:

	b_{20}	b_{19}	b_{18}	b_{17}	b_{16}
HLT	0	0	0	0	1
NOP	0	0	0	0	0

Inoltre, è da notare che l'istruzione NOP viene codificata ponendo tutti i bit per l' opcode $b_{23}...b_{16}$ a 0. Ulteriori configurazioni in questa sottocategoria non sono assegnate.

La seconda sottocategoria (contraddistinta da $b_{20} = b_{19} = 1$) contiene le istruzioni di complemento dei flag, che in realtà dovrebbero appartenere alla categoria di "Salto", in quanto hanno una funzione relativa ai flag. Il motivo per il quale queste istruzioni appartengono a questa categoria è dato dal fatto che i bit utili alla codifica delle istruzioni di "Salto" risultavano insufficienti. Questa insufficienza impediva di dare alla codifica delle istruzioni di salto una struttura logica. Per questo si è preferito spostare questo genere di istruzioni in una categoria relativamente più vuota rispetto a quella dei salti.

	b_{20}	b_{19}	b_{18}	b_{17}	b_{16}
CMC	1	1	0	0	0
CMO	1	1	0	0	1
CMS	1	1	0	1	0
CME	1	1	0	1	1
CMZ	1	1	1	0	0

Le ulteriori configurazioni in questa sottocategoria non sono assegnate.

Capitolo 5

Le istruzioni di vCPU

In questo capitolo presentiamo e discutiamo tutte le istruzioni di vCPU. Esse sono suddivise nelle seguenti categorie, prescindendo dalla loro classe, per essere analizzate in base alla loro funzione:

Aritmetico logiche sono le istruzioni che effettuano operazioni sui dati: sono sia aritmetiche che logiche.

Memorizzazione sono le istruzioni che permettono il flusso di dati dai registri alla memoria e viceversa.

Test, salto e controllo del flusso del programma sono le istruzioni che possono deviare il flusso del programma.

I/O sono le istruzioni che permettono il flusso dei dati da e verso l'esterno.

Di seguito, per ogni categoria, è presente una tabella organizzata nel seguente modo: nella prima colonna troviamo il codice macchina dell'istruzione; nella seconda troviamo la sua sintassi; nella terza troviamo la semantica scritta in un semplice formalismo; nella quarta sono elencati i flag che possono essere modificati dall'esecuzione dell'istruzione; infine, nella quinta e ultima colonna, troviamo un esempio. In tali tabelle, ogni riga contiene tutte le possibili varianti di una istruzione relativamente al tipo di argomento. Nel presentare e discutere le istruzioni useremo inoltre da un punto di vista sintattico le seguenti convenzioni:

R : per indicare un nome di un registro (vedi sezione 3.4);

N : per indicare un intero intero relativo positivo in complemento a 2 sui 13 bit dell'argomento, appartenente quindi all'intervallo $[0, +4095]$;

Z : per indicare un intero relativo in complemento a 2 sui 13 bit dell'argomento, appartenente quindi all'intervallo $[-4096, +4095]$.

5.1 Il formalismo per la semantica delle istruzioni

Il significato (o semantica) dell'istruzione, ovvero ciò che avviene con la sua esecuzione, è specificato con un formalismo molto semplice, consistente nello specificare un'*azione* costituita da due fasi: la prima consiste nel computare una espressione matematica che usa gli operatori tradizionali; la seconda parte, invece, consiste in un'assegnazione, ovvero nella memorizzazione del risultato dell'espressione in un luogo in grado di contenere il dato, come i registri o la memoria. Le due fasi dell'azione sono rappresentate con la seguente sintassi:

$$espr \rightarrow dest$$

dove *espr* è l'espressione da computare e *dest* è la destinazione del risultato della computazione. In alcuni casi l'azione può essere preceduta da una condizione, con l'interpretazione che l'azione viene eseguita solo quando la condizione è verificata. In questo caso non si parla più di azione, ma di *azione condizionata*. Una azione condizionata può essere rappresentata nella seguente forma:

$$cond \Rightarrow espr \rightarrow dest$$

Nel caso in cui la condizione *cond* non sia vera, l'azione associata all'istruzione non viene eseguita e quindi lo stato del programma non viene influenzato.

In questo formalismo di illustrazione della semantica l'uso delle parentesi tonde è utilizzato per descrivere la modalità di indirizzamento ad una cella di memoria. Se per esempio vogliamo riferirci al contenuto della cella 123, allora questo verrà indicato per mezzo delle parentesi come (123). Se vogliamo riferirci al contenuto della cella il cui indirizzo è contenuto nella cella 123, allora lo indicheremo come ((123)). Quando indichiamo il nome di un registro, intendiamo riferirci al suo valore. Il nome di un registro tra parentesi viene interpretato secondo le convenzioni sopra specificate. Quindi, se il registro BX contiene il valore 123, allora $27 + BX$ indica la somma tra 27 e 123, mentre $27 + (BX)$ indica la somma fra 27 ed il contenuto della cella di memoria 123 e $(27) + (BX)$ indica la somma tra il contenuto della cella 27 e quello della cella 123.

Per indicare che un indirizzo si riferisce alle porte di I/O piuttosto che alla memoria, vengono usate delle parentesi quadre attorno al numero di porta. Ad esempio, se l'argomento è del tipo

$$[N]$$

ci si sta riferendo alla porta N.

A volte è necessario riferirsi contemporaneamente a due registri. Per venire incontro a questa esigenza, il formalismo utilizza l'operatore ':' preceduto e succeduto dal nome di due registri. Nel dettaglio, il registro che segue l'operatore memorizza la porzione più bassa del dato, mentre il registro che precede l'operatore memorizza quella più alta, poiché non è possibile inserirla nel primo. Quindi, se si vuole indicare che la parte alta del risultato di una operazione viene memorizzato in BX e la parte bassa in AX, si può utilizzare la seguente sintassi:

$$operazione \rightarrow BX : AX$$

L'esigenza di un tale formalismo scaturisce dal fatto che alcune operazioni possono generare un risultato più grande rispetto alla parola di memoria adottata.

Le parentesi graffe vengono utilizzate per raccogliere gli operandi dei predicati logici di primo ordine. Infatti, questi vengono utilizzati nel formalismo di illustrazione della semantica. Ne segue una loro breve descrizione:

AND{x,y} Indica l'AND logico applicato bit a bit agli argomenti x e y.

OR{x,y} Indica l'OR logico applicato bit a bit agli argomenti x e y.

XOR{x,y} Indica l'OR esclusivo (XOR) logico applicato bit a bit agli argomenti x e y.

NOT{x} Indica il NOT logico applicato ad ogni singolo bit dell'argomento x.

Per la semantica che esce dal campo d'azione del formalismo adottato, verrà utilizzata la lingua italiana.

5.2 Istruzioni aritmetico logiche

5.2.1 Interpretazione aritmetica della configurazione dei flag

In questa sezione forniamo, specificatamente per le varie istruzioni, informazioni di dettaglio su come interpretare correttamente dal punto di vista aritmetico il valore dei flag complessi.

5.2.1.1 ADD, SUB, INC, DEC

OV : Se viene configurato ad 1, indica che la somma ha generato un risultato troppo grande, in modulo, per essere memorizzato in una parola (overflow) e pertanto non è possibile fruirne. Inoltre, quando questo flag è configurato ad 1, il valore degli altri flag perde di significato.

CA : Indicherebbe una condizione di overflow se l'operazione fosse stata effettuata su numeri in notazione ordinaria. Inoltre, il numero ottenuto affiancando questo bit, come il più significativo, al risultato della ALU in AX, rappresenta sempre il risultato anche in condizioni di overflow.

5.2.1.2 MUL

CA : il significato aritmetico di questo flag è lo stesso di OV.

OV : viene configurato ad 1 se il risultato è contenuto in AX:BX, 0 se è stato possibile memorizzarlo solo su AX. Non esistendo overflow per questa operazione, il significato che viene associato a questo flag è quello di overflow relativamente alla dimensione di una parola.

5.2.2 Elenco riassuntivo

OPCODE E TIPO	SINTASSI	SEMANTICA	FLAGS	ESEMPIO
11001000000	ADD Z	$AX + Z \rightarrow AX$	CA, SI, ZE, OV, EV.	ADD -10
11001000001	ADD @N	$AX + (N) \rightarrow AX$	CA, SI, ZE, OV, EV.	ADD @10
11001000010	ADD @@N	$AX + ((N)) \rightarrow AX$	CA, SI, ZE, OV, EV.	ADD @@10
11001000100	ADD R	$AX + R \rightarrow AX$	CA, SI, ZE, OV, EV.	ADD BX
11001000101	ADD @R	$AX + (R) \rightarrow AX$	CA, SI, ZE, OV, EV.	ADD @BX
11000000000	SUB Z	$AX - Z \rightarrow AX$	CA, SI, ZE, OV, EV.	SUB -5
11000000001	SUB @N	$AX - (N) \rightarrow AX$	CA, SI, ZE, OV, EV.	SUB @10
11000000010	SUB @@N	$AX - ((N)) \rightarrow AX$	CA, SI, ZE, OV, EV.	SUB @@10
11000000011	SUB R	$AX - R \rightarrow AX$	CA, SI, ZE, OV, EV.	SUB BX
11000000100	SUB @R	$AX - (R) \rightarrow AX$	CA, SI, ZE, OV, EV.	SUB @BX
11110000000	MUL N	$AX * N \rightarrow BX : AX$	CA, SI, ZE, OV, EV.	MUL 10
11110000001	MUL @N	$AX * (N) \rightarrow BX : AX$	CA, SI, ZE, OV, EV.	MUL @10
11110000010	MUL @@N	$AX * ((N)) \rightarrow BX : AX$	CA, SI, ZE, OV, EV.	MUL @@10
11110000100	MUL R	$AX * R \rightarrow BX : AX$	CA, SI, ZE, OV, EV.	MUL BX
11110000101	MUL @R	$AX * (R) \rightarrow BX : AX$	CA, SI, ZE, OV, EV.	MUL @BX
11111000000	DIV N	$AX \ / \ N \rightarrow AX$	SI, ZE, EV.	DIV 20
11111000001	DIV @N	$AX \ / \ (N) \rightarrow AX$	SI, ZE, EV.	DIV @10
11111000010	DIV @@N	$AX \ / \ ((N)) \rightarrow AX$	SI, ZE, EV.	DIV @@10
11111000100	DIV R	$AX \ / \ R \rightarrow AX$	SI, ZE, EV.	DIV BX
11111000101	DIV @R	$AX \ / \ (R) \rightarrow AX$	SI, ZE, EV.	DIV @BX
10000000000	MOD Z	$AX \ mod \ Z \rightarrow AX$	SI, ZE, EV.	MOD 2
10000000001	MOD @N	$AX \ mod \ (N) \rightarrow AX$	SI, ZE, EV.	MOD @10
10000000010	MOD @@N	$AX \ mod \ ((N)) \rightarrow AX$	SI, ZE, EV.	MOD @@10
10000000100	MOD R	$AX \ mod \ R \rightarrow AX$	SI, ZE, EV.	MOD BX
10000000101	MOD @R	$AX \ mod \ (R) \rightarrow AX$	SI, ZE, EV.	MOD @BX
10011000000	AND Z	$AND\{AX, Z\} \rightarrow AX$	SI, ZE, EV.	AND 6
10011000001	AND @N	$AND\{AX, (N)\} \rightarrow AX$	SI, ZE, EV.	AND @10
10011000010	AND @@N	$AND\{AX, ((N))\} \rightarrow AX$	SI, ZE, EV.	AND @@10
10011000100	AND R	$AND\{AX, R\} \rightarrow AX$	SI, ZE, EV.	AND BX
10011000101	AND @R	$AND\{AX, (R)\} \rightarrow AX$	SI, ZE, EV.	AND @BX
10010000000	OR Z	$OR\{AX, Z\} \rightarrow AX$	SI, ZE, EV.	OR -10

10010000001	OR @N	$OR\{AX,(N)\} \rightarrow AX$	SI, ZE, EV.	OR @10
10010000010	OR @@N	$OR\{AX,((N))\} \rightarrow AX$	SI, ZE, EV.	OR @@10
10010000100	OR R	$OR\{AX,R\} \rightarrow AX$	SI, ZE, EV.	OR BX
10010000101	OR @R	$OR\{AX,(R)\} \rightarrow AX$	SI, ZE, EV.	OR @BX
10001000000	XOR Z	$XOR\{AX,Z\} \rightarrow AX$	SI, ZE, EV.	XOR 10
10001000001	XOR @N	$XOR\{AX,(N)\} \rightarrow AX$	SI, ZE, EV.	XOR @10
10001000010	XOR @@N	$XOR\{AX,((N))\} \rightarrow AX$	SI, ZE, EV.	XOR @@10
10001000100	XOR R	$XOR\{AX,R\} \rightarrow AX$	SI, ZE, EV.	XOR BX
10001000101	XOR @R	$XOR\{AX,(R)\} \rightarrow AX$	SI, ZE, EV.	XOR @BX
11011000001	INC @N	$(N)+1 \rightarrow (N)$	CA, SI, ZE, OV, EV.	INC @10
11011000010	INC @@N	$((N))+1 \rightarrow ((N))$	CA, SI, ZE, OV, EV.	INC @@10
11011000100	INC R	$R+1 \rightarrow R$	CA, SI, ZE, OV, EV.	INC BX
11011000101	INC @R	$(R)+1 \rightarrow (R)$	CA, SI, ZE, OV, EV.	INC @BX
11010000001	DEC @N	$(N)-1 \rightarrow (N)$	CA, SI, ZE, OV, EV.	DEC @10
11010000010	DEC @@N	$((N))-1 \rightarrow ((N))$	CA, SI, ZE, OV, EV.	DEC @@10
11010000100	DEC R	$R-1 \rightarrow R$	CA, SI, ZE, OV, EV.	DEC BX
11010000101	DEC @R	$(R)-1 \rightarrow (R)$	CA, SI, ZE, OV, EV.	DEC @BX
01000001000	NOT	$NOT\{AX\} \rightarrow AX$	SI, ZE, EV.	NOT
01000000000	NEG	$NOT\{AX\}+1 \rightarrow AX$	SI, ZE, EV.	NEG

5.3 Istruzioni di memorizzazione

OPCODE E TIPO	SINTASSI	SEMANTICA	FLAGS	ESEMPIO
10100000000	LD Z	$Z \rightarrow AX$		LD -2
10100000001	LD @N	$(N) \rightarrow AX$		LD @10
10100000010	LD @@N	$((N)) \rightarrow AX$		LD @@10
10100000100	LD R	$R \rightarrow AX$		LD BX
10100000100	LD @R	$(R) \rightarrow AX$		LD @BX
10111000001	ST @N	$AX \rightarrow (N)$		ST @10
10111000010	ST @@N	$AX \rightarrow ((N))$		ST @@10
10111000100	ST R	$AX \rightarrow R$		ST BX
10111000100	ST @R	$AX \rightarrow (R)$		ST @BX

5.4 Test, salto e controllo del flusso del programma.

OPCODE E TIPO	SINTASSI	SEMANTICA	FLAGS	ESEMPIO
00011000000	CMC	$NOT\{CA\} \rightarrow CA$	CA.	CMC
00011001000	CMO	$NOT\{OV\} \rightarrow OV$	OV.	CMO
00011010000	CMS	$NOT\{SI\} \rightarrow SI$	SI.	CMS
00011011000	CMP	$NOT\{EV\} \rightarrow EV$	EV.	CMP
00011100000	CMZ	$NOT\{ZE\} \rightarrow ZE$	ZE.	CMZ
00000001000	NOP	Nessuna operazione.		NOP
00000000000	HLT	Termina l'esecuzione.		HLT
00100000000	JMP N	$N \rightarrow IP$		JMP 10
00101010000	JZ N	ZE = 1 $\Longrightarrow N \rightarrow IP$		JZ 10
00101011000	JNZ N	ZE = 0 $\Longrightarrow N \rightarrow IP$		JNZ 10
00100010000	JC N	CA = 1 $\Longrightarrow N \rightarrow IP$		JC 10
00100011000	JNC N	CA = 0 $\Longrightarrow N \rightarrow IP$		JNC 10
00100100000	JO N	OV = 1 $\Longrightarrow N \rightarrow IP$		JO 10
00100101000	JNO N	OV = 0 $\Longrightarrow N \rightarrow IP$		JNO 10
00100110000	JS N	SI = 1 $\Longrightarrow N \rightarrow IP$		JS 10
00100111000	JNS N	SI = 0 $\Longrightarrow N \rightarrow IP$		JNS 10
00101000000	JP N	EV = 1 $\Longrightarrow N \rightarrow IP$		JP 10
00101001000	JNP N	EV = 0 $\Longrightarrow N \rightarrow IP$		JNP 10
00110000000	JA N	AND{CA = 0, ZE = 0} $\Longrightarrow N \rightarrow IP$		JA 10
00100001000	JAE N	CA = 0 $\Longrightarrow N \rightarrow IP$		JAE 10
00100000000	JB N	CA = 1 $\Longrightarrow N \rightarrow IP$		JB 10
00110011000	JBE N	AND{CA = 1, ZE = 1} $\Longrightarrow N \rightarrow IP$		JBE 10
00110100000	JG N	AND{ZE = 0, SI = OV} $\Longrightarrow N \rightarrow IP$		JG 10
00110101000	JGE N	SI = OV $\Longrightarrow N \rightarrow IP$		JGE 10
00110110000	JL N	SI != OV $\Longrightarrow N \rightarrow IP$		JL 10
00110111000	JLE N	AND{ZE = 1, SI != OV} $\Longrightarrow N \rightarrow IP$		JLE 10

5.5 I/O

OPCODE E TIPO	SINTASSI	SEMANTICA	FLAGS	ESEMPIO

01100001000	IN N	$[N] \rightarrow AX$		IN 10
01100000000	OUT N	$AX \rightarrow [N]$		OUT 10

Parte III

Architettura dell'emulatore

Lo scopo dell'emulatore è quello di rendere possibile sia l'esecuzione di programmi, che di rendere comprensibile tutto ciò che accade all'interno di un calcolatore. Come si avrà modo di vedere nei prossimi capitoli, la struttura dell'emulatore è divisa in tre parti: *vCPU*, *Devices* e *GUI*. *vCPU* contiene tutte le componenti che, interagendo, emulano vCPU specificata nella parte II. *Devices* contiene tutto ciò che concerne i dispositivi collegabili all'emulatore: è utile pensare a queste componenti software come se fossero a loro volta emulatori di dispositivi. Infine, *GUI* interessa la veste grafica dell'emulatore. È bene sottolineare che l'architettura adottata per la parte *vCPU* è tale da consentire di avere diverse vesti grafiche.

In particolare *vCPU* è divisa in due parti: *Core* e *Instructions*.
Il *Core* racchiude le componenti di base dell'emulatore come, ad esempio, la memoria, le porte e i registri; l'*Instructions* riguarda esclusivamente le istruzioni. Pur essendo le istruzioni parte integrante di una vera CPU, *Instructions* è tuttavia esterno all'emulatore, in quanto queste devono poter essere inserite dinamicamente. L'inserimento dinamico delle istruzioni in vCPU evita la necessità di dover ricompilare tutto l'emulatore: con ciò si intende che se si vuole aggiungere una nuova istruzione, è necessario implementarla come specificato nella sezione 6.4.2, compilarla ed inserirla nel percorso laddove l'emulatore ha modo di trovarla. In questo modo è possibile estendere il set di istruzioni senza dover necessariamente toccare l'implementazione dell'emulatore.

Un approccio simile al precedente è stato adottato anche per i dispositivi: se si vuole aggiungere un nuovo dispositivo, è necessario implementarlo come specificato nella sezione 7.2, compilarlo e inserirlo nel percorso dove l'emulatore ha modo di trovarlo.

Come si avrà modo di approfondire più avanti, la differenza che intercorre tra l'implementazione di una istruzione e di un dispositivo sta nel fatto che il primo non ha interfaccia grafica, mentre il secondo ha la possibilità di averla. Questo aspetto ne cela anche un altro: le istruzioni vengono eseguite nel thread principale del programma, mentre i dispositivi su uno proprio.

In questa parte verrà introdotta l'architettura del prototipo di un emulatore per vCPU. In particolar modo si analizzerà il *Core*, nucleo del software. Verrà inoltre affrontato il design della parte *Devices*, anche se con minor dettaglio. Il design della parte *GUI* non verrà affrontata: in compenso, nella parte IV verrà illustrato un prototipo per la veste grafica.

Nei diagrammi che seguiranno, verrà utilizzata la convenzione UML per descrivere l'architettura dell'emulatore. In particolar modo le interfacce si distinguono dalle classi in quanto hanno una *I* (i maiuscola) alla fine del nome. Ad esempio, *MemoryI* è una interfaccia, mentre *DefaultMemory* è una classe. Se un modulo ha il nome che non termina per *I* ma è una classe astratta, verrà esplicitato con una nota.

Capitolo 6

vCPU

Questa parte rappresenta il nucleo dell'emulatore: qui troviamo l'executer che regola le fasi di esecuzione di un programma, la memoria, le porte e i registri, inclusi i flag. La figura 6.1 rappresenta un diagramma UML che evidenzia come le componenti di vCPU siano interrelate.

6.1 Elementi di base: Word e Address

Per gestire il flusso di dati tra le componenti dell'emulatore, sono stati introdotte due classi: *Word* e *Address*. La prima è utilizzata per rappresentare una parola di memoria a 24 bit, mentre la seconda rappresenta un indirizzo a 13 bit in complemento a 2. Nella figura 6.2 vengono mostrate le due classi.

Come è possibile vedere in figura, la classe *Word* è dotata di metodi in grado di accedere sia alle singole parti di una istruzione, sia alla parola nella sua interezza. Le singole parti vengono accedute tramite i seguenti metodi:

getOpcode() utilizzato per accedere all'opcode dell'istruzione.

getArgumentType() utilizzato per ottenere il tipo di argomento.

getWordArgument() utilizzato per ottenere l'argomento sotto forma di oggetto *Word*.

getAddressArgument() utilizzato per ottenere l'argomento sotto forma di *Address*.

Per accedere al valore della parola stessa si utilizza il metodo *getModel()*. I metodi *equals(Object)*, *greatherThan(Word)*, e *lessThan(Word)* sono utilizzati per poter confrontare due *Word*. I valori rappresentabili da una *Word* variano da -2^{23} a $+2^{23}-1$. Analogamente a *Word*, anche *Address*

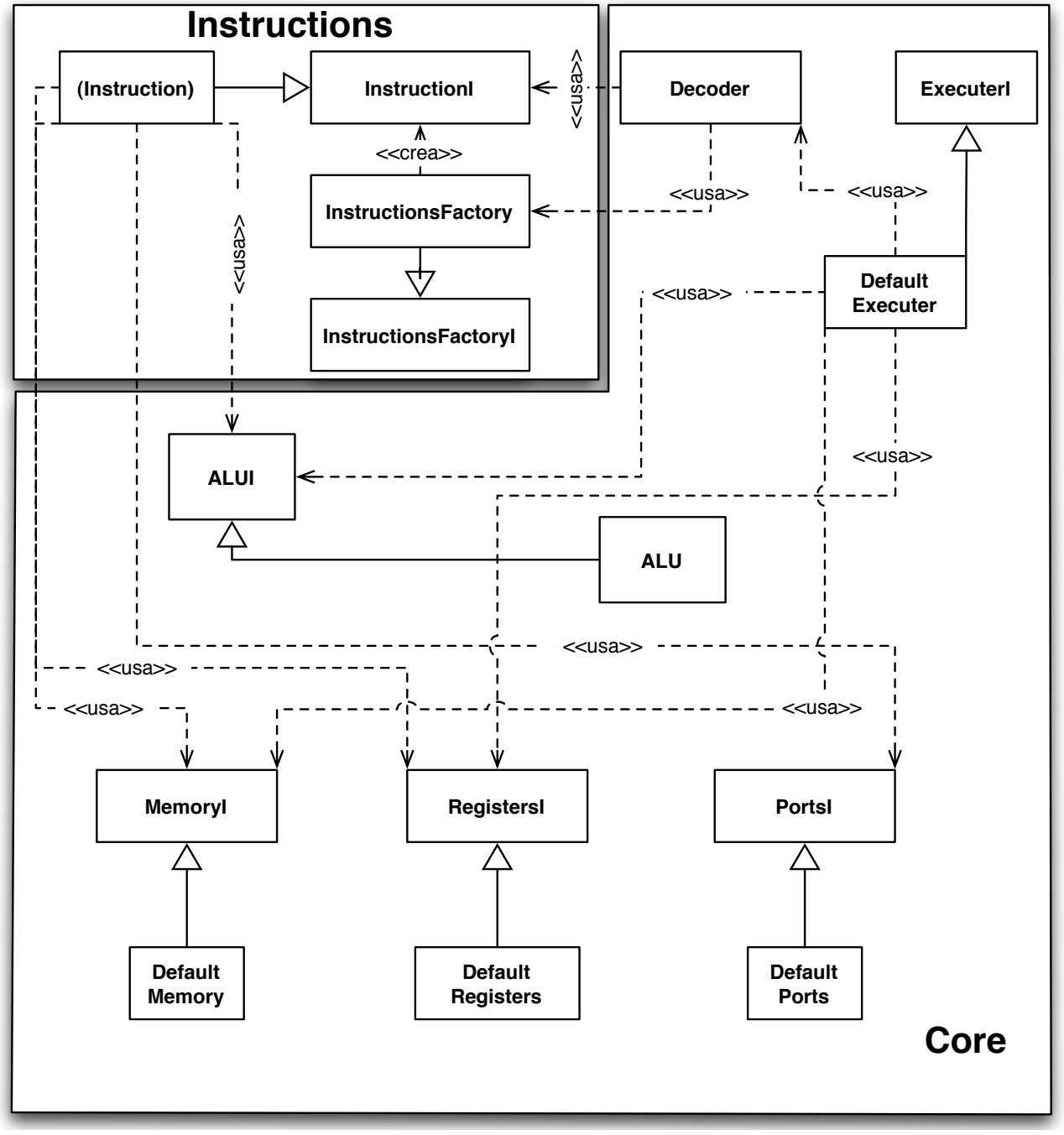


Figura 6.1: UML di vCPU

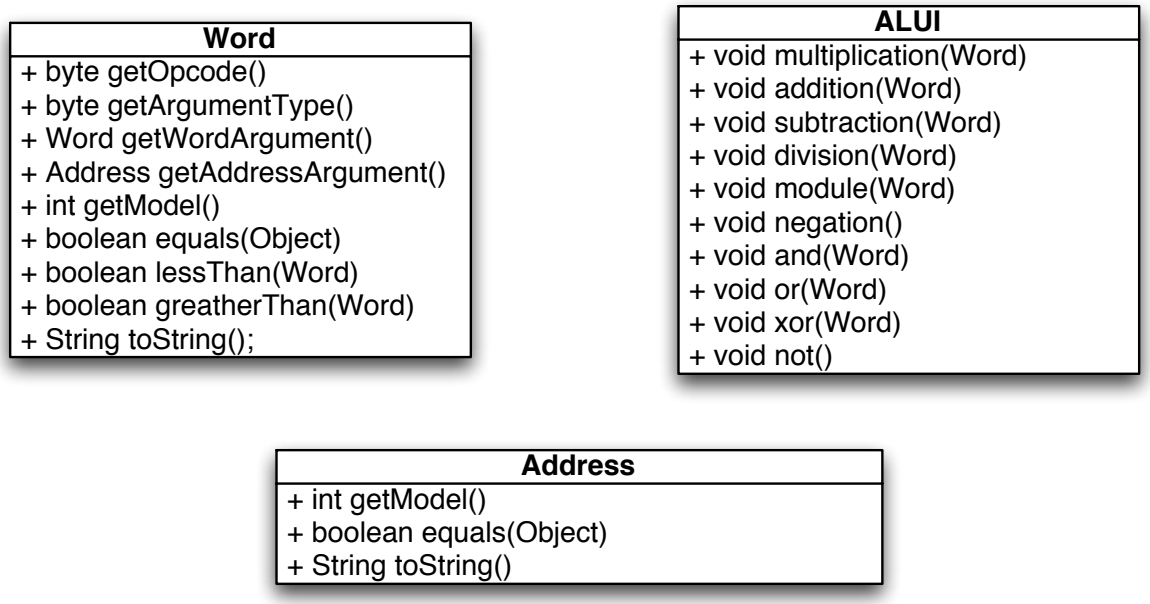


Figura 6.2: Word, Address ed ALUI

ha dei limiti: può rappresentare solo valori entro 0 e $2^{12} - 1$. Gli unici metodi rilevanti di cui la classe *Address* dispone sono *getModel()*, per accedere al valore dell'indirizzo, ed *equals()*.

6.2 I Locator

L'architettura di *vCPU* è stata progettata per avere una facile manutenzione. Per raggiungere questa proprietà sono state introdotte delle interfacce per ogni elemento fondamentale: in questo modo si raggiunge un buon livello di disaccoppiamento. Per rendere del tutto disaccoppiati gli elementi, sono stati introdotti anche dei locator, illustrati nelle figure 6.3 e 6.4. In questo modo, i moduli di *vCPU* sono disgiunti anche nella creazione degli oggetti. *VCPU* dispone di due locator: *VCPULocator* che permette di creare tutto il necessario per utilizzare l'emulatore, *VCPUInternalLocator* che si occupa di localizzare le risorse interne. In dettaglio, *VCPULocator* localizza oggetti di tipo *MemoryI*, *PortsI*, *RegistersI* e *ExecuterI*;

VCPUInternalLocator, invece, localizza oggetti di tipo *ALUI* e *InstructionsFactoryI*.

È importante che le risorse siano create come singleton: ciò permette di avere una istanza unica disponibile in tutto il ciclo di vita del processo dell'emulatore. Se così non fosse, non si avrebbe l'effetto desiderato: infatti, la memoria usata dall'emulatore deve essere unica.

In realtà, la memoria come altre risorse, è singleton perché il *vCPULocator* restituisce la

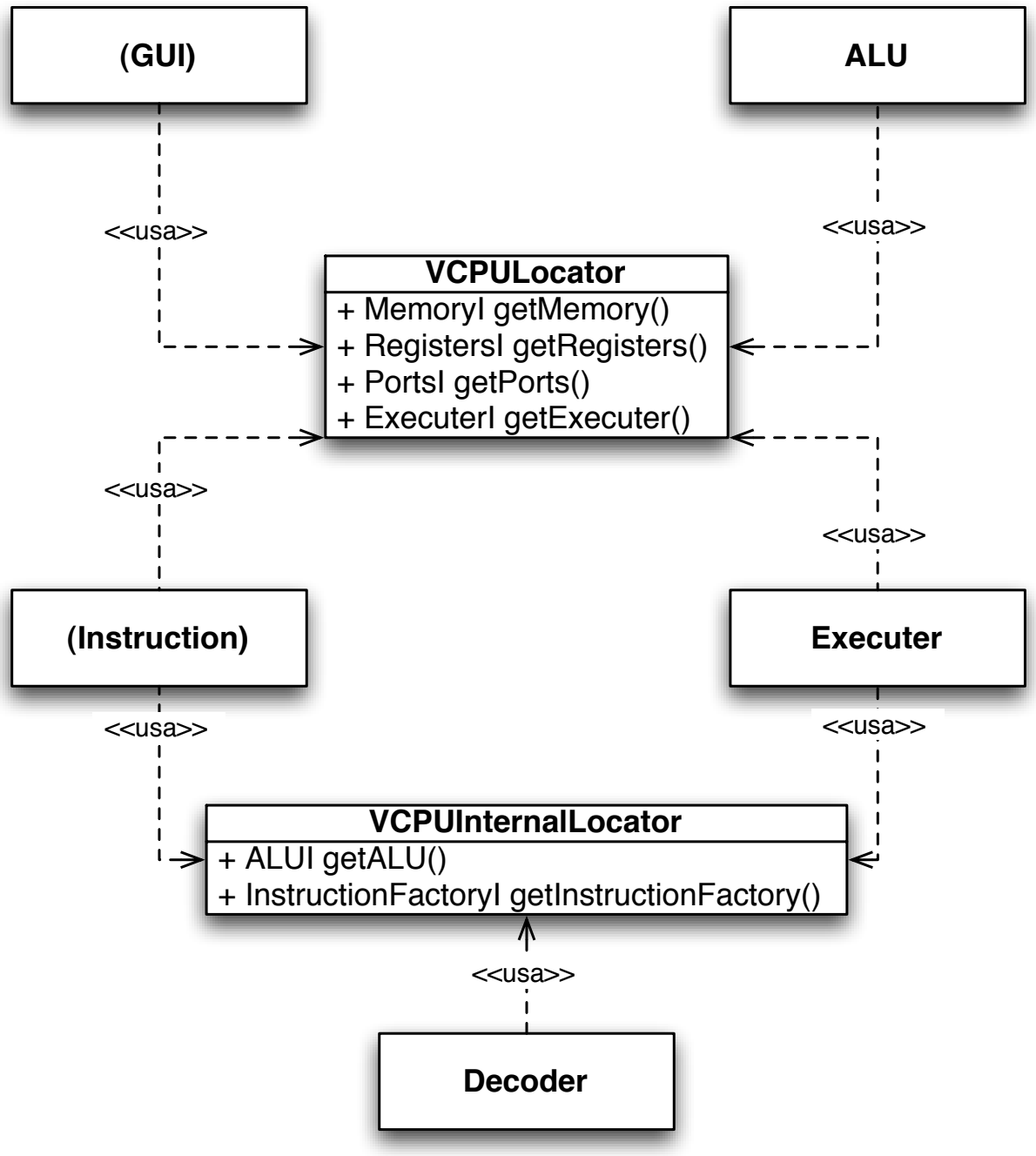


Figura 6.3: UML del locator di vCPU: utilizzo da parte delle componenti

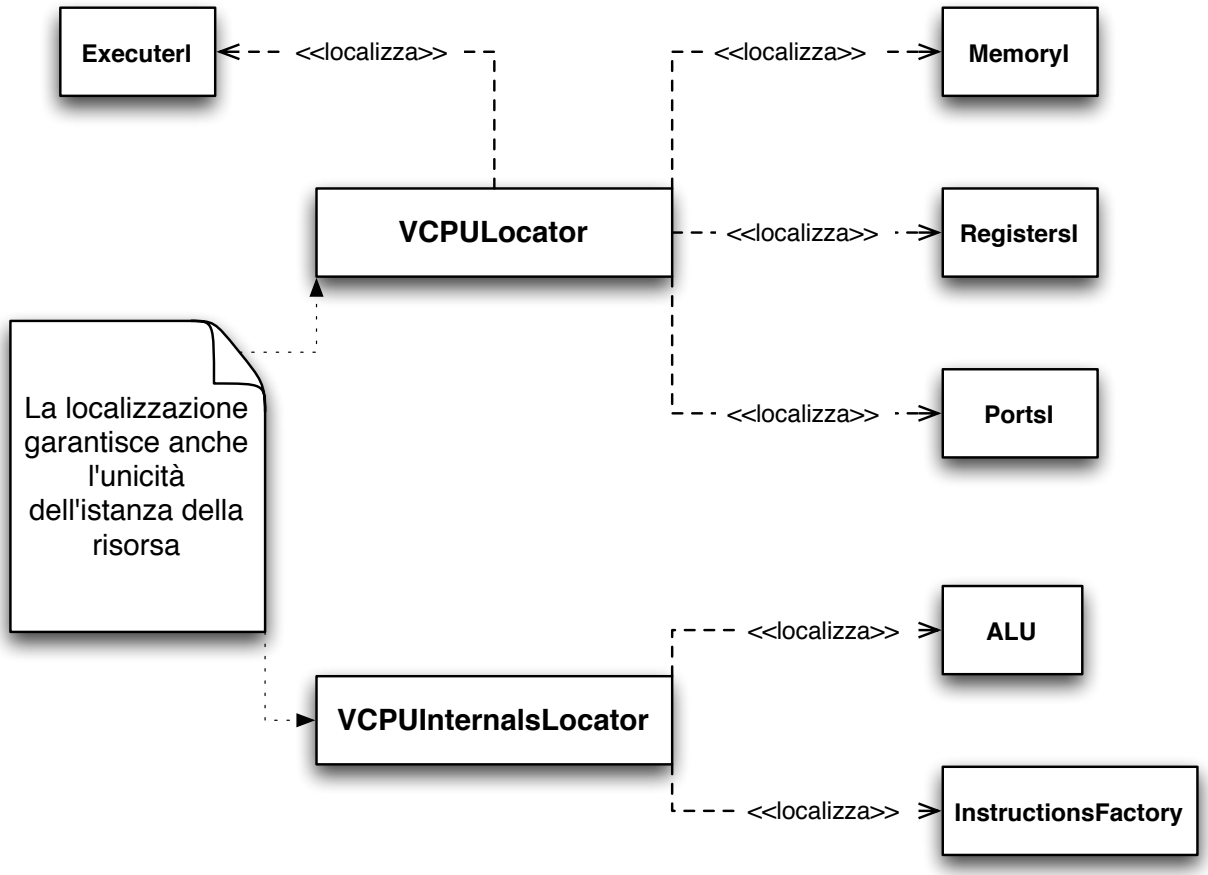


Figura 6.4: UML del locator di vCPU: localizzazione delle risorse

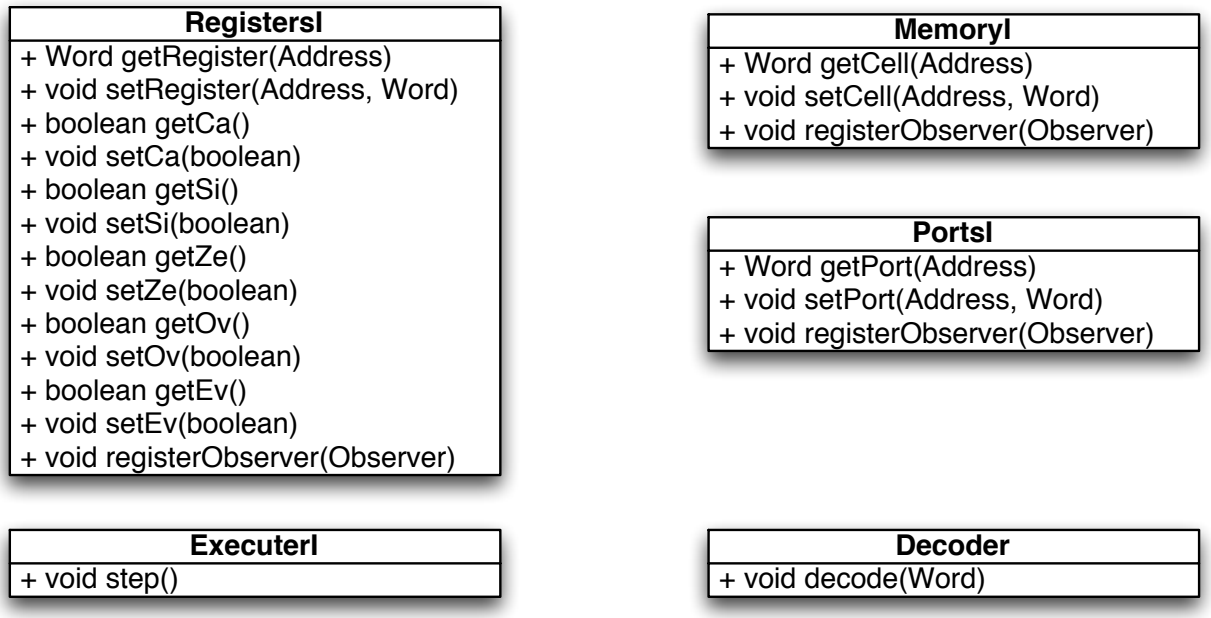


Figura 6.5: UML delle risorse del Core

stessa istanza a qualunque richiedente. Un altro approccio sarebbe stato quello di implementare il pattern *Singleton* nell'implementazione delle risorse.

6.3 Core

Il *Core*, come anticipato in precedenza, è il nucleo dell'emulatore. Tutta l'emulazione infatti avviene al suo interno e le risorse principali, come le porte e la memoria, sono qui contenute. Gli elementi di base del *Core* sono mostrati in figura 6.5. Questo inoltre implementa una messaggistica basata su eventi: l'infrastruttura adottata viene affrontata successivamente nella sezione 6.3.6.

6.3.1 La ALU: ALUI

Le funzionalità della ALU di vCPU sono raccolte nell'interfaccia ALUI. L'implementazione delle funzionalità della ALU è localizzata nella classe *DefaultALU* che, come è possibile vedere in figura 6.1, implementa l'interfaccia *ALUI*. Come la ALU descritta nella sezione 3.7, *ALUI* impone che sia il primo operando che il risultato siano sempre contenuti nel registro AX, mentre il secondo operando, se presente, venga specificato come argomento. Ogni metodo implementato da *DefaultALU* configura automaticamente i flag, sia semplici che complessi, in base all'esito dell'operazione. Questa componente è fornita dei metodi visibili in figura 6.2

6.3.2 l'Executer: ExecuterI

Questa componente si può considerare il punto di contatto tra la *GUI* e *vCPU*. Tramite questa classe è infatti possibile eseguire i programmi scritti in memoria. Il compito fondamentale di tale componente rimane comunque quello di effettuare il ciclo di fetch-decode-execute per eseguire il codice in memoria. L'unico metodo di cui dispone *ExecuterI* è *step()*: questo non accetta nessun parametro, ed esegue un singolo ciclo fetch-decode-execute. In questo modo si lascia completamente alla *GUI* il compito di implementare tutte le modalità di esecuzione che si vogliono fornire all'utente. Non tutte le fasi del ciclo fetch-decode-execute sono implementate internamente all'*Executer*: infatti, della codifica dell'istruzione se ne occupa il *Decoder*, cercando la classe che implementa l'istruzione associata al codice operativo della *Word* passatagli. In figura 6.5 viene mostrato l'*ExecuterI* e in figura 6.1 è possibile vedere sia la sua implementazione, *DefaultExecuter*, che il *Decoder*. Il *Decoder* dispone di una interfaccia molto semplice: consiste nel solo metodo *decode(Word)*. Questo metodo accetta come parametro una *Word* contenente una istruzione da decodificare. Sostanzialmente, la codifica consiste nel comprendere l'istruzione, associata all'opcode nella *Word* passatagli come argomento, e nell'invocare la classe di tipo *InstructionI* che la implementa.

6.3.3 La Memoria: MemoryI

Indubbiamente, la memoria è tra i componenti più interessanti. L'implementazione della memoria per l'emulatore è, come si può vedere in figura 6.1, *DefaultMemory* e la sua interfaccia è specificata in *MemoryI*. Lo scopo che ha la memoria è quello di memorizzare sia il codice del programma che i dati utili al suo funzionamento. Le implementazioni possibili per la memoria sono molteplici: tra le più semplici vi è quella basata su mappa. Questa consiste nell'adottare una mappa per associare ad un indirizzo (la chiave) un contenuto (il valore). In particolar modo, l'implementazione adottatata associa un oggetto di tipo Word (valore) ad un oggetto di tipo Address (la chiave), univoco nella mappa. In questo modo si ottiene una struttura dati di facile consultazione, senza rischiare di creare inconsistenze di alcun tipo.

6.3.4 Le Porte: PortsI

Per permettere ad una CPU di comunicare con l'esterno, le architetture più diffuse comprendono anche delle porte di I/O. Come è stato possibile vedere nella sezione 3.8, anche vCPU adotta questo genere di sistema per la comunicazione con l'esterno. Per rendere semplice l'implementazione e l'apprendimento delle funzionalità dell'emulatore, le porte utilizzano la stessa

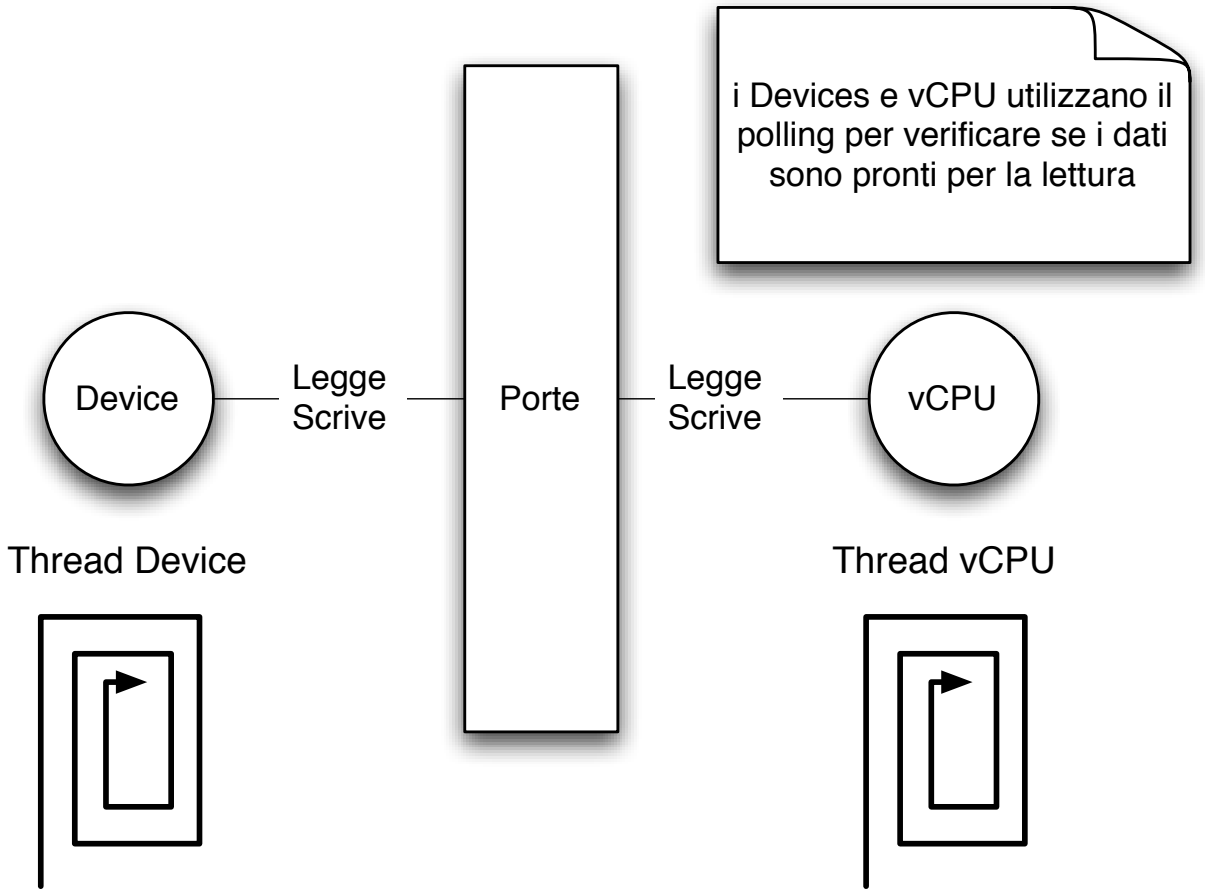


Figura 6.6: UML dei dispositivi

architettura adottata per la memoria. DefaultPort, la classe che implementa le porte del sistema, ha molti aspetti in comune con DefaultMemory (fig. 6.1). Anche questa implementazione usa una mappa per associare dei valori ai numeri di porta.

A differenza della memoria, alle porte accedono anche i vari dispositivi collegati all'emulatore. Non esiste un sistema di messagistica basata su eventi, come nel caso tra le porte e la GUI, ma vengono utilizzate semplici scritture e letture (polling) su determinate porte (fig. 6.6). Ciò rende più realistica la modalità di comunicazione tra una semplice CPU priva di interrupt, quale è vCPU, e i dispositivi.

6.3.5 I Registri: RegistersI

I registri rappresentano uno degli aspetti più importanti dell'architettura di una CPU. La loro implementazione è rappresentata dalla classe *DefaultRegisters*, che implementa l'interfaccia *RegistersI* come è mostrato in figura 6.1. Questa risorsa è organizzata in modo tale da avere un'interfaccia che renda possibile accedere sia ai registri, che ai singoli bit del registro

PSW descritti nelle sezioni 3.6.1 e 3.7.1. In figura 6.5 viene mostrata l'interfaccia *RegisterI* con tutti i suoi metodi. Come è possibile vedere, sono disponibili vari metodi *accessor* per poter leggere o modificare il valore dei flag, e i metodi *getRegister(Address)*, *setRegister(Address, Word)* per poter leggere o modificare il valore dei registri AX, BX, CX, DX e PC. Gli argomenti di tipo *Address* rappresentano i registri e vengono utilizzati come parametri ai metodi *getRegister(Address)* e *setRegister(Address, Word)* per specificarne uno in particolare.

6.3.6 Gli Eventi

L'emulatore è stato progettato per rendere il più possibile indipendente l'implementazione di vCPU da quella della *GUI*. Per fare ciò è stato necessario introdurre un sistema di messagistica ad eventi in grado di disaccoppiare l'implementazione di vCPU con il resto dell'emulatore. L'architettura adottata è visibile in figura 6.7. In questo modo, quando avviene un cambiamento ad esempio nella memoria, tramite un evento, viene notificato il *vCPUObserverI*, che si occupa di aggiornare la GUI dell'emulatore implementata dalla parte *GUI*. In questo modo l'implementazione del *Core* non conosce altro che l'interfaccia *vCPUObserverI*. Questo aspetto è particolarmente utile qualora si decidesse, ad esempio, di cambiare GUI all'emulatore.

Come è possibile vedere in figura 6.7, le interfacce *RegistersI*, *PortsI* e *MemoryI* estendono l'interfaccia *vCPUObservableI*, in modo da poter gestire il flusso degli eventi. L'interfaccia *vCPUObservableI* fornisce i metodi utili alla registrazione (*registerObserver(vCPUObserverI)*) e deregistrazione (*unregisterObserver(vCPUObserverI)*) dei *vCPUObserverI*, mentre l'interfaccia *vCPUObserverI* viene implementata da tutte le componenti interessate agli eventi generati dai *vCPUObservableI*. Il codice da eseguire in risposta ad un evento, viene inserito nel metodo *modified(AddressableEvent)*, il quale accetta un parametro di tipo *AddressableEvent*.

AddressableEvent è una classe che rappresenta un evento di vCPU, in quanto estende la classe *vCPUEvent*. La sua caratteristica è che rappresenta un evento generato da una risorsa indirizzabile, come la memoria.

Il *vCPUMulticaster* è una classe in grado di gestire un elenco di *vCPUObserverI* registrati e di notificarli in caso di eventi. L'introduzione di questa classe di utilità rende semplice per le risorse come *MemoryI* interagire con set di *vCPUObserverI*. Come è possibile vedere in figura 6.7, le varie implementazioni delle risorse delegano ciò che concerne il lancio degli eventi al *vCPUMulticaster*. L'unico loro aspetto, che viene trattato separatamente in ogni risorsa, è la loro creazione: ad esempio, il *DefaultMemory* ha la necessità di costruire, a fronte di una modifica, un nuovo *AddressableEvent* contenente l'indirizzo di memoria modificato.

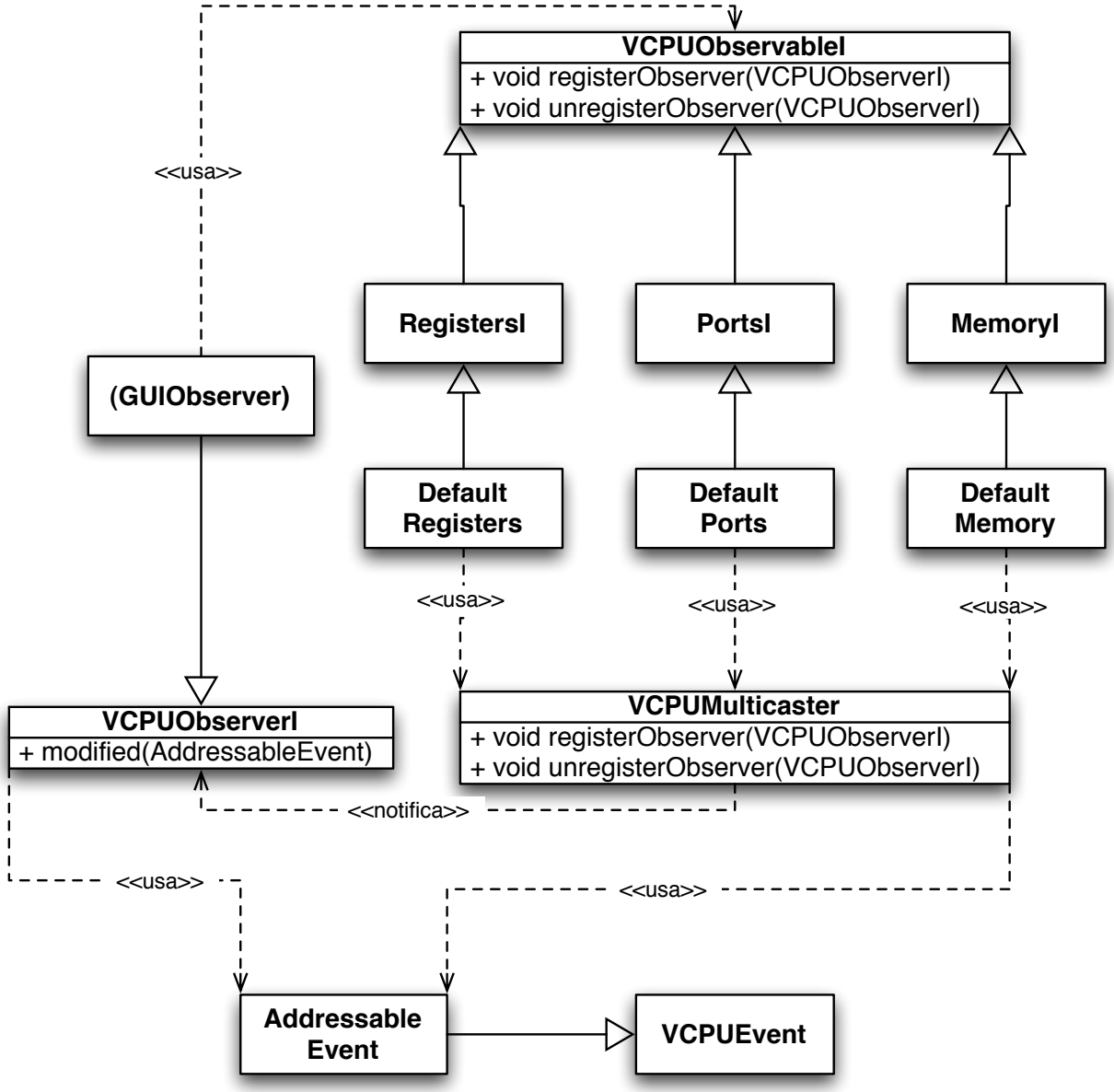


Figura 6.7: UML della gestione degli eventi di vCPU

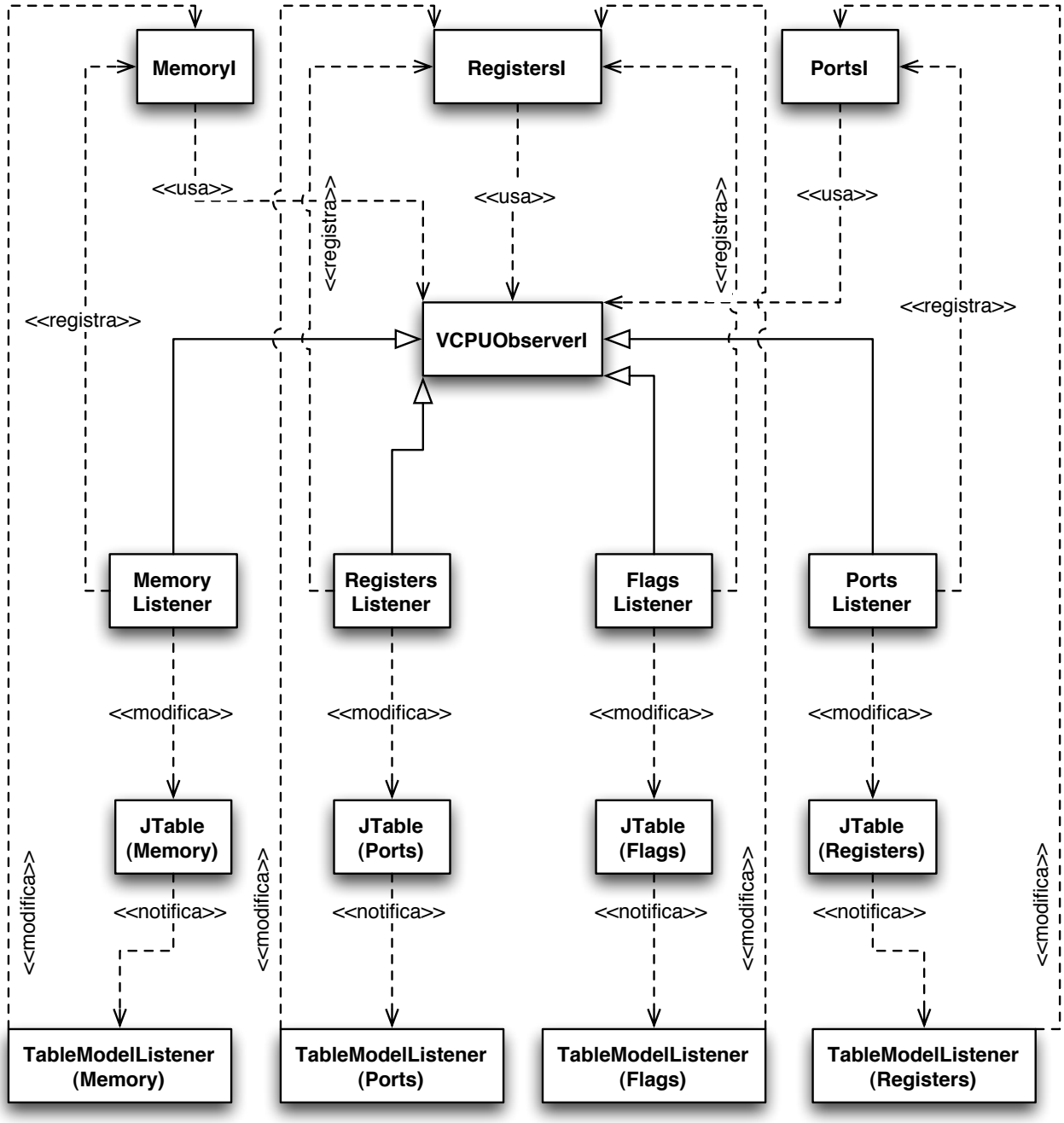


Figura 6.8: UML del flusso degli eventi di vCPU

La figura 6.8 mostra un esempio di flusso degli eventi tra il *Core* e la *GUI*. Come è possibile vedere, vengono creati dei *VCPUObserverI* uno per ogni risorsa di vCPU. A fronte di un evento, questi observer andranno a modificare il contenuto della relativa tabella sulla *GUI*, per mostrare la modifica all'utente. Nel caso opposto, quando l'utente modifica una cella delle tabelle nella *GUI*, il relativi listener andranno a modificare il contenuto della memoria.

6.4 Instructions

Nel seguito verrà affrontata la parte *InstructionI* e verranno analizzati due argomenti fondamentali per l'architettura delle istruzioni: il loro caricamento e la loro struttura. Il primo argomento è rilevante in quanto rende noto come avviene il caricamento dinamico delle istruzioni nell'emulatore; il secondo invece lo è in quanto descrive la struttura di una *InstructionI*, indicando quindi le modalità di implementazione delle istruzioni dell'emulatore.

6.4.1 Caricamento delle istruzioni

Il caricamento delle istruzioni segue un processo particolare. Essendo caricabili dinamicamente, quando viene eseguito il programma, si analizza un percorso specifico, dove l'emulatore sa di trovare le implementazioni delle varie istruzioni. Nell'analisi di questo percorso, tutte queste classi vengono caricate dall'*InstructionsFactory* che implementa l'interfaccia *InstructionsFactoryI* (fig. 6.9). La presenza di questo modulo è giustificata dal fatto che esiste la necessità di disaccoppiare il *Core* dalla modalità di caricamento delle istruzioni.

Ciò che resituirà il metodo *getInstructions()* dell' *InstructionFactory* è una collezione di *IstruzioniI*, sotto forma di Set contenente tutte le istruzioni caricate. Questa collezione è utilizzata direttamente dal Decoder, per sapere di quali *IstruzioniI* dispone l'emulatore.

6.4.2 Struttura di una istruzione

Il caricamento dinamico delle istruzioni prevede che ogni singola istruzione implementi l'interfaccia *InstructionI* (fig. 6.9). In questo modo, l'*InstructionFactory* è in grado di restituire una collezione di oggetti che implementano la stessa interfaccia, senza dover entrare nel dettaglio implementativo di ogni singola istruzione.

Inoltre, il fatto che di ogni istruzione si conosca solo l'interfaccia, permette di costruire un *Core* più semplice. In particolare, il *Decoder* è in grado di compiere il suo lavoro sapendo solo a quale opcode sono associate.

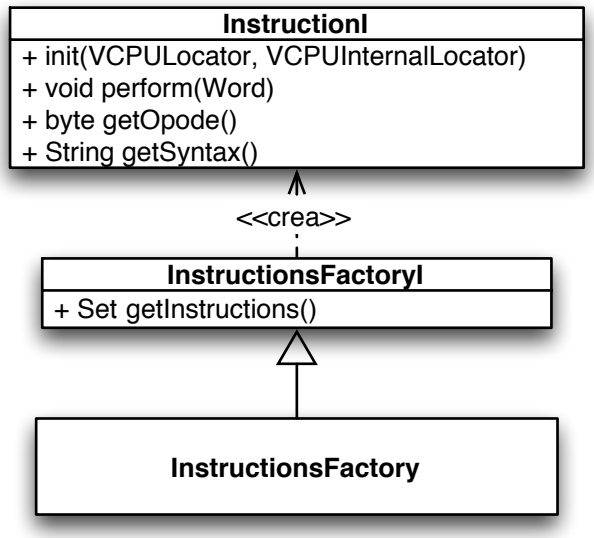


Figura 6.9: UML dell'istruzione

Osservando la figura 6.9 si può notare come una istruzione sia dotata di metodi come *void perform(Word)* per eseguire l'operazione implementata. Il metodo *init(VCPULocator, VCPUInternalLocator)*, chiamato unicamente dall'*InstructionFactory*, inizializza l'istruzione indicandogli quali sono i locator da utilizzare per individuare le risorse da utilizzare. I metodi *byte getOpCode()* e *String getSyntax()* invece servono rispettivamente per fornire l'opcode associato all'operazione e per restituire l'etichetta che corrisponde alla sintassi dell'istruzione. Il parametro passato come argomento al metodo *void perform(Word)* è la parola che contiene l'intera istruzione. In questo modo si avranno a disposizione tutte le informazioni disponibili, come la tipologia dell'argomento e l'argomento stesso.

Capitolo 7

I Dispositivi

La presenza dei dispositivi rende sicuramente più interessante la programmazione di vCPU. In questo modo è possibile, ad esempio, connettere un dispositivo che simuli un display e codificare un programma in grado di effettuare dei disegni su di esso. Un altro esempio più complesso potrebbe essere quello in cui vengono collegati un display e una tastiera: in questo modo è possibile scrivere programmi che possano interagire con l'utente.

Nel seguito verranno analizzati vari aspetti dei dispositivi per l'emulatore. In particolar modo si esamineranno la modalità di caricamento e la struttura dei dispositivi.

7.1 Caricamento dei dispositivi

Il caricamento dei dispositivi nell'emulatore avviene in modo analogo a quello delle istruzioni: esiste infatti un *DevicesFactory* con il metodo *getDevices()* in grado di restituire, sotto forma di *Set*, i dispositivi collegabili all'emulatore.

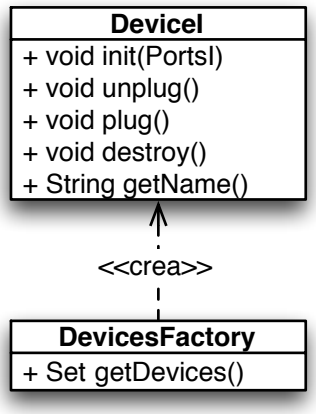


Figura 7.1: UML dei dispositivi

7.2 Struttura di un dispositivo

Un dispositivo si basa sull'interfaccia *DeviceI*. Questa interfaccia presenta una serie di metodi in grado di inizializzare, chiudere, collegare e scollegare un dispositivo. Ne segue una breve descrizione:

init(PortsI) invocato quando si vuole inizializzare il dispositivo. Vengono inoltre passate le porte con le quali il dispositivo deve interagire.

unplug() invocato quando il dispositivo viene scollegato.

plug() invocato quando il dispositivo viene collegato.

destroy() invocato quando il dispositivo viene spento.

getName() invocato per conoscere l'etichetta associata al dispositivo.

In generale, alla chiamata di *init(PortsI)*, viene creato un nuovo thread, in stato di pausa, in grado di emulare il dispositivo. Il metodo *plug()* invece si occupa di far partire questo thread, per iniziare l'emulazione del dispositivo. Una chiamata a *unplug()* invece provoca l'arresto del dispositivo, come se venisse messo in pausa. Infine, il metodo *destroy()* viene invocato alla chiusura del dispositivo: tipicamente si occupa di operazioni come la liberazione di memoria allocata in modo atipico o la chiusura di stream aperto con qualche risorsa.

Parte IV

Interfaccia dell'emulatore

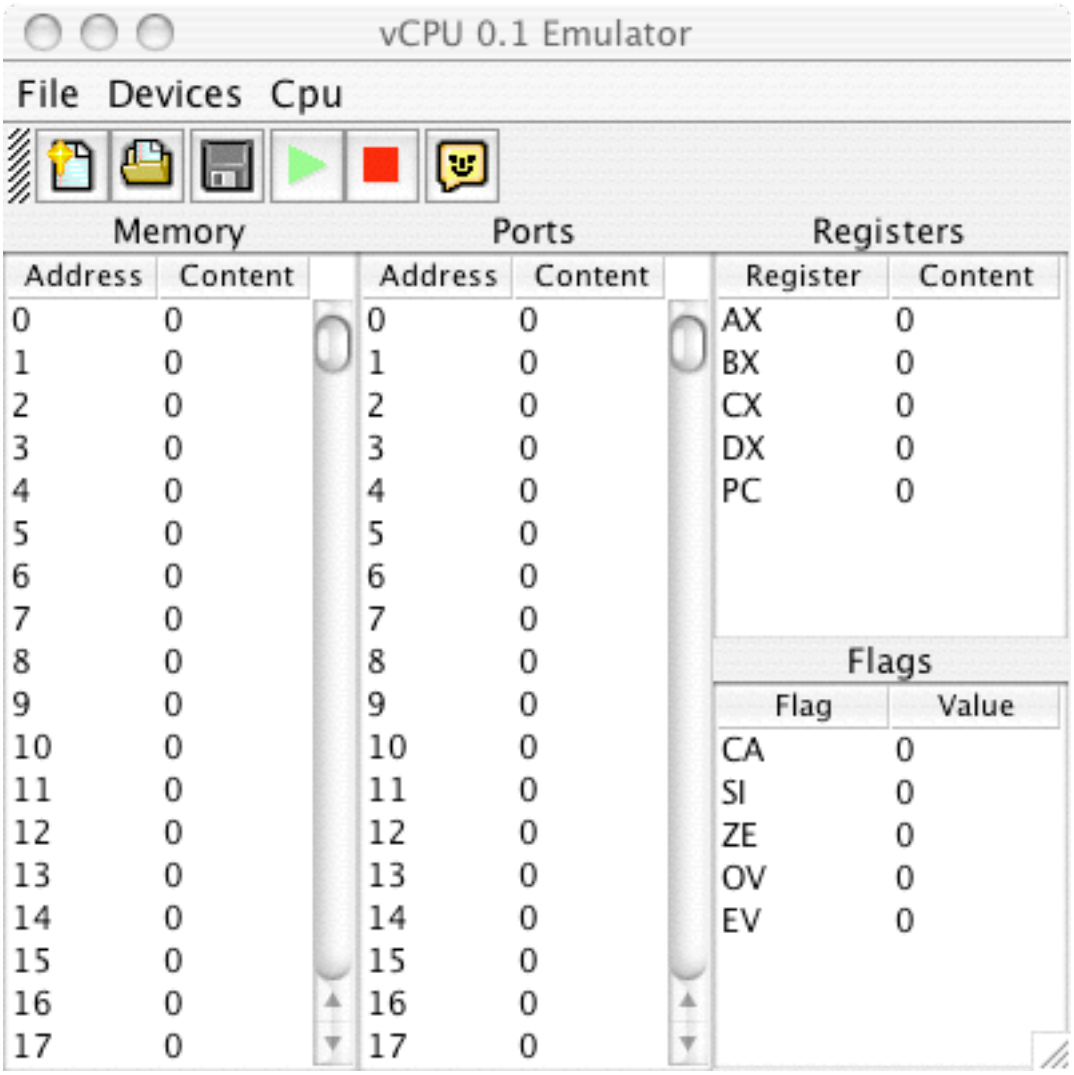


Figura 7.2: Finestra principale.

L'interfaccia grafica del prototipo dell'emulatore di vCPU è stata progettata per essere il più semplice possibile, ma al contempo anche completa per rendere l'emulatore fruibile. Tramite la realizzazione di un prototipo, è stato possibile affrontare lo studio dell'interfaccia nel modo più ampio, che verrà descritta nelle seguenti sezioni. In figura 7.2 viene mostrata l'interfaccia della finestra principale.

Capitolo 8

Finestra principale

La finestra principale si riduce alla visualizzazione dell'intero della CPU e delle risorse ad essa associate, come la memoria e le porte. L'aspetto delle varie aree della finestra sono simili e, ancor più importante, sono semplici da utilizzare: consistono in una tabella di due colonne, una per il contenuto e una per visualizzare l'etichetta associata al contenuto stesso e di un numero di righe in relazione al numero di entità presenti nella risorsa in esame. Ad esempio, sia la memoria che le porte di I/O dispongono di una tabella di due colonne e 4096 righe, ognuna delle quali rappresenta ogni singola cella.

8.1 Memoria

Come anticipato nella sezione precedente, la memoria è organizzata in una tabella di due colonne e 4096 righe, ognuna delle quali rappresenta una parola. Le parole sono ordinate in base al loro indirizzo e si presentano dalla più bassa alla più alta a partire dalla prima riga della tabella. Gli indirizzi, presenti nella prima colonna, non sono in alcun modo modificabili: rimane possibile comunque cambiare, tramite il menù del programma, il formato numerico adottato per la loro visualizzazione. Con ciò si intende che è possibile visionare gli indirizzi nel formato decimale (usato di default) ed esadecimale. Il formato binario e ottale non vengono interpellati in quanto poco utili per vari motivi: utilizzare il formato binario per visualizzare un indirizzo può essere scomodo, tanto quanto quello ottale, che inoltre è stato abbandonato dalla maggior parte dei software. In figura 8.1 è mostrata la porzione di finestra dedicata alla memoria.

Le modalità di editing della memoria possono essere molteplici: allo stato attuale è possibile inserire il codice in formato esadecimale, decimale, binario e sotto forma di stringa, nella sintassi descritta nel capitolo 5. Questo permette di arrivare a scrivere il codice in modalità, e quindi difficoltà, differenti. Il vantaggio di avere sia la possibilità di scrivere le istruzione per mezzo

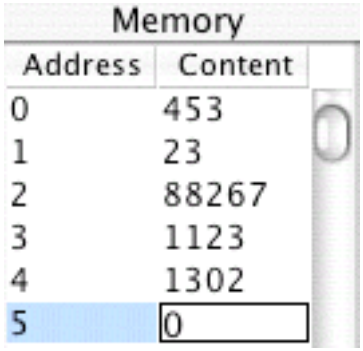


Figura 8.1: La memoria.

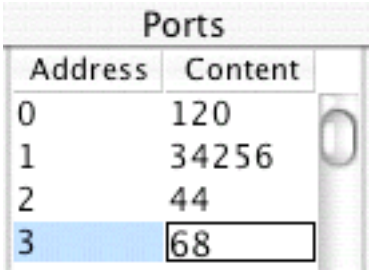


Figura 8.2: Le porte.

di una stringa e sia per mezzo del codice binario è che nel primo caso è possibile scrivere i programmi rapidamente, e nel secondo caso invece è possibile arrivare a scrivere il programma considerando la struttura e l'organizzazione dei codici operativi assegnati alle istruzioni. Il codice esadecimale gode degli stessi aspetti del codice binario, con la sola differenza che risulta più compatto e, nella maggior parte dei casi, anche più agevole.

8.2 Porte

Anche le porte adottano le stesse modalità di visualizzazione della memoria. Ciò comporta una omogeneità nell'interfaccia che risulta gradevole all'utente, in quanto non lo porta a dover apprendere diverse GUI per diverse risorse: conseguenza di ciò è una veloce comprensione dell'interfaccia grafica. Come la memoria, il valore delle porte possono essere modificate. L'unica differenza che si ha con la modalità di editing della memoria è che, ovviamente, non è possibile inserire delle stringhe, dato che non si può assegnare una istruzione come valore di porta. In figura 8.2 è mostrata la porzione di finestra dedicata alle porte.

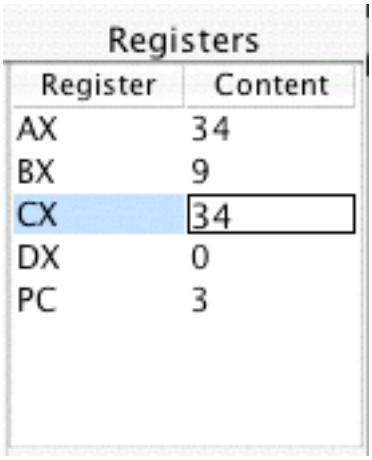


Figura 8.3: I registri.

8.3 Registri

L'area dedicata ai registri è molto più piccola rispetto a quella dedicata alla memoria e alle porte, ma con ciò non si vuole sottolineare una particolare differenza: in realtà, al posto degli indirizzi, nella prima colonna sono presenti i nomi dei registri. Analogamente alle porte, è possibile modificare il valore dei registri. I formati numerici adottati anche qui sono il binario, il decimale e l'esadecimale. In figura 8.3 è mostrata la porzione di finestra dedicata ai registri.

8.4 I Flag

Anche quest'area presenta delle somiglianze con le altre. La prima colonna, relativa alle etichette, presenta i nomi dei registri di vCPU. Gli unici valori possibili per i flag sono due: 0 e 1. Questo è dovuto al fatto che i flag vengono memorizzati in singoli bit, e pertanto le combinazioni numeriche che possono essere memorizzare sono due. Il numero ridotto di valori porta ad assumere una diversa modalità di input, effettuato per mezzo di una casella combinata con i soli valori 0 e 1 disponibili. In questo caso, avere più formati di editing è totalmente inutile, perché non si potrebbero distinguere: i valori 0 e 1 sono presenti in ogni formato numerico. In figura 8.4 è mostrata la porzione di finestra dedicata ai flag.

8.5 Plug e Unplug dei dispositivi

La possibilità di effettuare il Plug e l'Unplug dei dispositivi ha chiaramente un riscontro anche nell'interfaccia dell'emulatore. Tramite il menù *Devices* è possibile accedere ai sottomenù *Plug* e *Unplug* che rispettivamente contengono la lista dei dispositivi collegati e scollegati dal sistema.

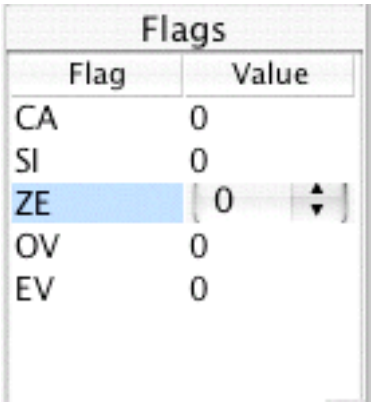


Figura 8.4: I flag.

Una volta che si decide di collegare un dispositivo al sistema, è necessario selezionare, come anticipato prima, il menù *Devices* e il sottomenù *Plug*. Qui si trovano una serie di voci di menù, ognuna delle quali rappresenta un determinato dispositivo. Selezionando una delle voci, si provoca l'attivazione del medesimo. Ciò comporta il lancio di un thread separato, che agisce da periferica: verifica eventuali cambiamenti sulle porte, scrive i risultati su delle altre, e così via. Una volta attivato un dispositivo, la voce ad esso associata si sposta nel sottomenu *Unplug* per permettere di scollegarla in qualsiasi momento. Una volta deciso di staccare un dispositivo, si deve selezionare la relativa voce nel menù *Unplugged* ed essa ritornerà nel sottomenù *Plugged*, pronta per essere nuovamente collegata.

8.6 Esecuzione dei programmi

Come ogni emulatore, il software è in grado di poter eseguire dei programmi scritti in memoria. Attualmente è disponibile la modalità di esecuzione a velocità variabile, per poter permettere di osservare meglio il cambiamento dello stato della CPU.

Quando viene avviata l'esecuzione del programma da una situazione di stop, essa avviene sempre a partire dalla prima cella di memoria. Se invece si riparte dallo stato di pausa, l'esecuzione riprenderà dall'istruzione successiva all'ultima, relativamente al flusso del programma, che è stata eseguita. Per far partire, mette in pausa e fermare la CPU, sono disponibili delle voci nel menù *CPU*. Inoltre, è presente un sottomenù, *Speed*, che dispone di varie velocità, espresse in millisecondi. Scegliendone una, si vincola l'emulatore a rispettare delle pause tra l'esecuzione di una istruzione e quella successiva.

Bibliografia

[1] William Stallings. *Architettura e organizzazione dei calcolatori*. Quinta edizione. Jackson Libri.

[2] Andrew S. Tanenbaum, Albert S. Woodhull. *Sistemi Operativi*. Seconda edizione. UTET Libreria.

[3] *On the Design of a New CPU Architecture for Pedagogical Purpose*. D. Ellard, D. Holland, N. Murphy, M.Seltzer.

[4] *Hillside Net*.
http://www.hillside.net/patterns/.

[5] *PIC16F86 Instruction Set*.
http://www.eee.bham.ac.uk/woolleysi/teaching/pic/16f84inst.htm.