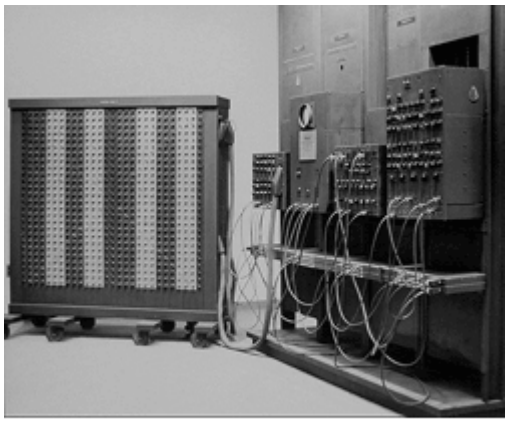
Indice

Prefazione

Cos'è ENIAC?

E.N.I.A.C., è l'acronimo di Electronic Numerical Integrator And Computer, il primo computer elettronico della storia. Concepito nel 1946 usando valvole termoioniche anticipò il transistor di due anni e trovò impieghi vari dal campo militare a quello civile. La programmazione dipendeva dai collegamenti dei cavi su pannelli: a seconda della loro combinazione poteva essere risolto un diverso problema.



The ENIAC Today

Nel 2006 si è quindi celebrato il 60° anniversario, non solo di ENIAC ma della storia della intera programmazione; proprio per questo motivo, quindi per un tributo dovuto, è stato deciso di dare il nome ENIAC all'argomento oggetto di questa tesi di laurea. ENIAC è un'implementazione per vCPU (virtual C.P.U.), originariamente oggetto di tesi di laurea del collega Mauro Codella; in questa è stata progettata la specifica, quindi un set d'istruzioni e tutto l'hardware necessario ad eseguirle.

1 Cenni su vCPU

Verranno ora riportate alcune delle specifiche alla base di vCPU. Si raccomanda di fare riferimento alla tesi originale redatta dal collega Mauro Codella:
"vCPU, un'unità di elaborazione ad uso didattico: specifica dell'architettura e prototipazione di un emulatore".

1.1 Caratteristiche

L'architettura di vCPU è a 24 bit. Sono quindi usate parole della lunghezza di 24 bit pari a 16777216 senza segno, o comprese tra -83886608 e +83886608 con segno in complemento a due; tutti i dispositivi che saranno brevemente trattati sono pensati, progettati e implementati proprio per lavorare a 24 bit.

1.2 A.L.U.

Acronimo di Arithmetic Logic Unit è la vera e propria unità di calcolo, che permette di effettuare tutte le operazioni di base, tra cui:

- Somma;
- Sottrazione;
- Incremento;
- Decremento;
- Divisione;
- Modulo;
- Moltiplicazione;
  - Particolare: poiché il prodotto di due numeri a 24 bit ne produce uno a 48 bit, verrà diviso il risultato in due parti: R0 e R1;
- AND Logico;
- OR Logico;
- XOR Logico;
- NOT Logico.

L'ALU è concepita per eseguire operazioni tra parole a 24 bit con segno, ma senza virgola.

1.3 La Memoria

Data l'architettura a 24 bit, per semplicità, le celle di memoria possono contenere parole della medesima dimensione.
La memoria è costituita da 2^{12} celle.

1.4 Le Porte

Data l'architettura a 24 bit, per semplicità, le celle delle porte possono contenere parole della medesima dimensione. Servono per caricare dispositivi di I/O esterni.
Le porte sono costituite da 2^{12} celle.

1.5 I Registri

Fondamentalmente esistono cinque registri, adatti a scopi diversi:

- AX: detto anche accumulatore viene usato come registro predefinito per la maggior parte delle operazioni;
- BX,CX,DX: registri general purpose usabili dal programmatore;
- PC: è il Program Counter ovvero il registro che si occupa di salvare l'indirizzo della cella di memoria che contiene l'istruzione successiva da eseguire.

1.6 I Flag

I flags sono dei registri di dimensione 1 bit, hanno la funzione di determinare il flusso del programma. Per comodità d'ora in poi quando si parla di registri, si intenderà quelli del paragrafo (1.5) mentre per i registri flags si userà il solo termine flags.
Possono avere valore 1 o 0, si dividono in due categorie e sono impostati dal risultato dell'ultima operazione:

- Semplici;

- SI: flag di segno, vale 1 se il risultato è negativo, 0 altrimenti;
- ZE: flag di zero, vale 1 se il risultato è pari a zero, 0 altrimenti;
- EV: flag di parità, vale 1 se la somma dei bit a 1 del risultati è pari, 0 altrimenti;

- Complessi; sono specifici delle operazioni di somma e moltiplicazione:
  - Somma:
    - CA: vale 1 se si è verificato un riporto, 0 altrimenti;
    - OV: vale 1 se l'MSB del risultato differisce dagli MSB degli operandi, 0 altrimenti;
  - Prodotto:
    - CA: vale 0 se i bit in R1 e l'MSB in R0 sono uguali tra loro, 1 altrimenti;
    - OV: identico a CA.

1.7 Le Istruzioni

Sono state pensate due classi di istruzioni: *alpha* e *beta*. Hanno compiti diversi poiché la classe *alpha* permette soprattutto di gestire operazioni di base, oltre che logiche e di memorizzazione; ogni istruzione può arrivare ad avere fino a cinque diversi tipi di operando in argomento:
(Indicando con "Operando" il valore da raggiungere)

- Immediato: l'operando è esattamente il valore da usare;
  - Sintassi esempio: ADD **Z**;
- Indirizzo Diretto: l'operando è nella cella di memoria indicata;
  - Formato: Memoria -> Operando;
  - Sintassi esempio: ADD **@N**;
- Indirizzo Indiretto: l'operando è nella cella di memoria il cui indirizzo è nella cella di memoria indicata;
  - Formato: Memoria -> Memoria -> Operando;
  - Sintassi esempio: ADD **@@N**;
- Registro: l'operando è nel registro indicato;

  - Formato: Registro -> Operando;
  - Sintassi esempio: ADD **R**;
- Registro Indiretto: l'operando è nella cella di memoria il cui indirizzo è nel registro indicato;
  - Formato: Registro -> Memoria -> Operando;
  - Sintassi esempio: ADD **@R**;

Per l'operando, sono dedicati i tredici bit più a destra; bisogna quindi fare delle precisazioni sul significato di Z,N,R:

- **Z**: intero compreso tra -4096 e +4095, poiché si tratta di 2^{13} in complemento a due;
- **N**: poiché le celle di memoria sono 2^{12} l'indirizzo è compreso tra 0 e +4095;
- **R**: i registri sono cinque, saranno dunque usati cinque interi positivi per l'indirizzo di cui, per comodità, è riportata la codifica in intero piuttosto che in binario:
  - AX: 0;
  - BX: 1;
  - CX: 6;
  - DX: 7;
  - PC: 4095.

Differentemente, il tipo *beta* non ha sottotipi e racchiude principalmente istruzioni per il controllo del flusso del programma, di salto condizionato ed incondizionato.

Per non creare confusione, in tutta la tesi si farà riferimento alla "classe alpha o classe beta" con "tipo alpha o tipo beta".

2 Dettagli di progettazione

Verranno ora trattati alcuni concetti inerenti la base di ENIAC.

2.1 Strumenti di sviluppo

ENIAC è stato sviluppato utilizzando i seguenti tools:

- Sun Java Development Kit ver. 1.6;
- EclipseSDK ver. 3.2.1;
- Junit ver. 4.1;
- Apache ANT ver. 1.7.0;
- Subversion ver. 1.1.9.

2.2 Metodologia di sviluppo e progettazione

Come si può evincere dal 2.1, dato l'uso di Junit, è stata usata la metodologia di sviluppo T.D.D., meglio nota come Test Driven Development o metodologia di sviluppo guidata dai test. E' una metodologia impiegata esclusivamente per lo sviluppo del simulatore, quindi per le risorse all'interno del package ENIAC.

Inoltre per questioni di disaccoppiamento, quindi per consentire il più possibile il riuso del codice e per rendere espandibile in un futuro l'architettura del simulatore, si è fatto uso dei Design Patterns.

Successivamente quando si parlerà delle classi di ENIAC, nello specifico, saranno trattati i patterns implementati.

2.3 Perché il T.D.D.

Il TDD[1] è una metodologia di sviluppo dotata di aspetti sia positivi che negativi, uno degli aspetti peculiari è che allunga notevolmente i tempi di sviluppo di un'applicazione dato che richiede lo sviluppo di pari passo sia della classe obiettivo che del suo test:

[1] Vedi: http://en.wikipedia.org/wiki/Test-driven_development

1. Creazione della classe obiettivo;
2. Minime modifiche del codice della classe obiettivo;
3. Creazione della classe di test, in cui quest'ultimo fallisca;
4. Modifica della classe al punto (3.), affinché il test vada a buon fine;
5. Ritorno al punto (2.).

Il pregio dei test è inequivocabilmente che all'ottenimento del test finale funzionante si otterrà anche la classe obiettivo perfettamente funzionante, che senza ulteriori controlli verrà subito inserita nel repository.
Il TDD é stato particolarmente utile nello sviluppo di risorse quali la memoria, le porte, i registri, l'ALU, il bus ma anche le numerose istruzioni che vCPU è in grado di riconoscere.

2.4 Perché i Design Patterns

I Design Patterns hanno avuto un ruolo importante nello sviluppo di ENIAC: laddove possibile sono stati usati patterns per migliorare l'aspetto progettuale. Nel simulatore, spesso, si sono presentate situazioni dove non si poteva fare altrimenti che riusare il medesimo codice per risolvere un problema identico o simile.
Il ruolo dei Design Patterns è proprio quello di descrivere una soluzione generale, semplice ed elegante per risolvere un problema ricorrente.
I patterns implementati sono i seguenti:

- Interface;
- Factory Method;
- Locator;
- Implementation;
- Observer.

In seguito, attraverso l'uso di diagrammi UML ed esempi di codice sorgente, saranno discussi relativamente all'implementazione in ENIAC: Capitolo 4, Design Patterns in ENIAC.

2.5 Descrizione dei Task di ANT

ANT ha lo scopo di automatizzare una serie di operazioni solitamente ripetitive nello sviluppo di ENIAC, ad esempio la compilazione.
ANT quando eseguito dalla linea di comando come prima operazione cerca il file *build.xml,* ed esegue il task specificato.
Premesso che la compilazione avviene nella directory *bin*, e la pubblicazione del prodotto finale nella directory *dist,* si procede alla descrizione dei task attualmente implementati nel file *build.xml* e le dipendenze da altri task:

1. *clean*: pulisce tutto ciò precedentemente creato da ANT come i files compilati nella directory *bin,* i files necessari all'esecuzione in *dist*, la documentazione in *docs*;
2. *compile*: esegue le compilazione di tutti i sorgenti, ponendoli quindi nella directory *bin*;
   - dipendenza: (1.);
3. *dist*: prepara i files necessari all'esecuzione, come *ENIAC.jar*, all'interno della directory *dist*;
   - dipendenza: (1.) (2.);
4. *execute*: esegue *ENIAC.jar* situato in *dist*;
   - dipendenza: (1.), (2.), (3.);
5. *javadoc*: genera Javadoc in base ai tag specifici inseriti nei sorgenti; la documentazione, consistente di documenti html, viene creata nella directory *docs/javadoc*;
   - dipendenza: (1.), (2.);
6. *test-compile*: esegue la compilazione di tutti i test, ponendoli quindi nella directory *bin;*
   - dipendenza: (1.), (2.);
7. *test*: esegue tutti i test e contestualmente crea un report dell'esito dei test; la documentazione, consistente di documenti html, viene creata nella directory *docs/report* e *docs/xml*;
   - dipendenza: (1.), (2.), (6.).

Una particolare menzione la merita il task (3.), in quanto oltre a generare *ENIAC.jar* procede anche alla copia del contenuto delle directory *resources* contenente le immagini, *lib* contenente librerie utili e del package *eniac.vcpu.instruction.module* dalla directory *bin* che contiene tutte le classi delle istruzioni con il loro codice operativo.

2.6 Gerarchia

Viene ora descritta la gerarchia adottata per i package e le directory dei sorgenti:

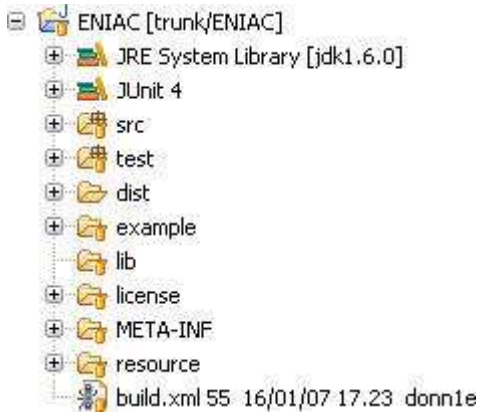


Figura 2.6.1

Come è possibile vedere dall'immagine, i package principali sono due:

- *src*;
- *test*.

Il package *test*, al suo interno, contiene tutti i sorgenti dei test di tipo Junit utilizzati per lo sviluppo del simulatore; in *src*, invece, sono presenti tutti i sorgenti che si dividono in due rami: *eniac* e *gui.*

Sono inoltre presenti directory diverse:

- *dist*: contiene il file *ENIAC.jar* e l'occorrente necessario all'esecuzione;
- *example*: programmi scritti in codice assembly di vCPU mostrato in esadecimale come il calcolo di un fattoriale;
- *lib*: librerie necessario al corretto funzionamento del simulatore e dei suoi test;

- *license*: contiene la licenza GPL del simulatore per la sua pubblicazione sulla comunità open source di sourceforge.net;
- *META-INF*: contiene *MANIFEST.MF*, file di configurazione necessario ad ANT per compilare correttamente il progetto;
- *resource*: risorse esterne come le immagini;
- *dist*: usata dal task di ANT *dist* per pubblicare i file necessari all'esecuzione.

2.7 Notazioni

Notazioni particolari sono state assunte per i nomi delle classi. Laddove il nome è terminante con:

- *A*: il file è una classe astratta;
- *I*: il file è un'interfaccia;
- *S*: è un file con metodi *static*; usato solo per le factories.

Mentre se invece inizia con:

- *Test*: si tratta di una classe test.

Inoltre il codice delle classi presenta alcune notazioni:

- Le variabili di istanza, sono precedute da un "_".

Quando si farà riferimento a programmi scritti con istruzioni il cui formato è riconosciuto da ENIAC, si farà riferimento ad essi come programmi scritti in linguaggio assembly di vCPU.

Notazione adottata negli schemi UML (o Unified Modelling Language):

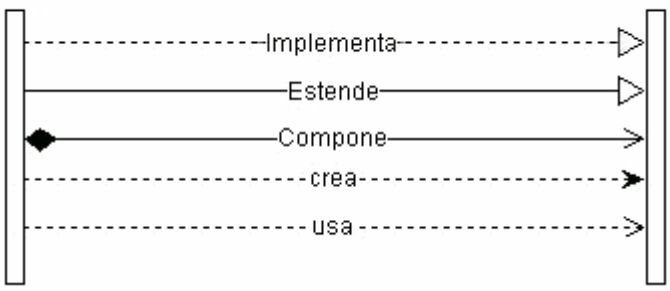


Figura 2.7.1

3 ENIAC: il simulatore

Di seguito è trattata la *business logic* del simulatore.

3.1 Struttura dei package

L'immagine che segue rappresenta il numero, i nomi e la disposizione dei package:

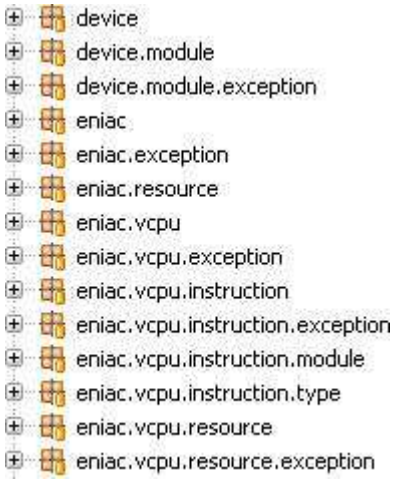


Figura 3.1.1

ma è opportuno andare per ordine.

Fondamentalmente, all'interno del package *eniac* si è cercato di disporre le varie risorse come se si fosse in presenza di hardware reale; per esempio se la memoria e le porte sono esterne alla CPU, lo stesso non vale per i registri, i flag, l'ALU e le istruzioni che questa è in grado di riconoscere.

Per questo motivo tutto ciò che fa parte della CPU è stato inserito all'interno del package *eniac.vcpu,* mentre le restanti risorse in *eniac*; per chiarire l'esposizione segue un diagramma UML:

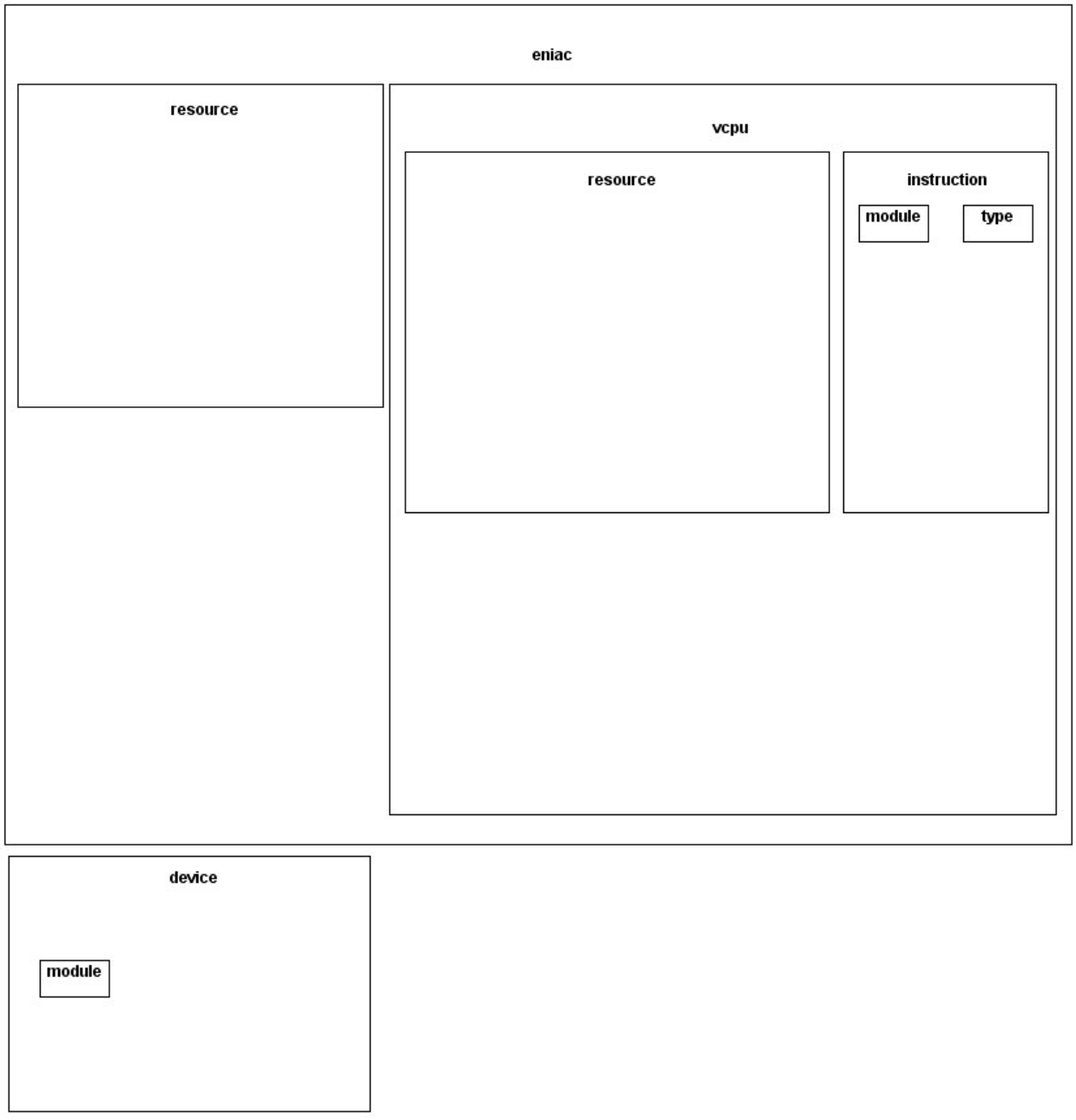

eniac
resource
vcpu
resource
instruction
module
type
device
module

Figura 3.1.2

3.2 Parole e Indirizzi

- package *eniac*

Qualsiasi dato (sia istruzione, sia costante) all'interno di ENIAC viene rappresentato come parola a 24 bit: da questo il nome *Word* per la classe che gestisce questo tipo di dati.

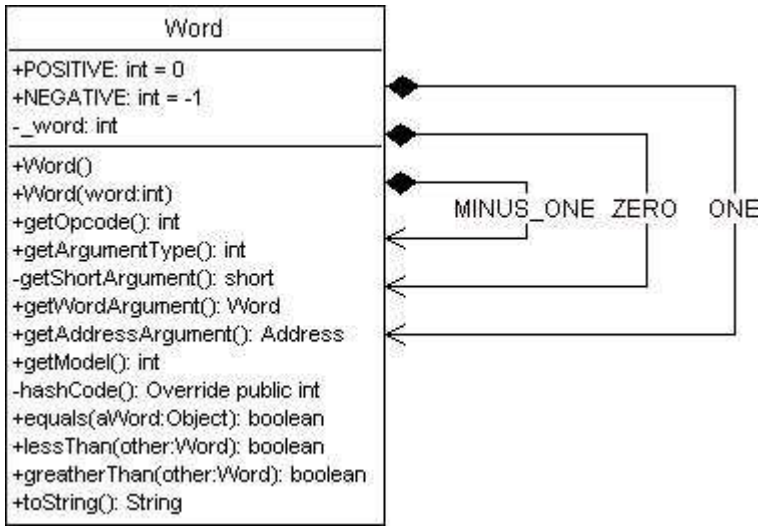


Figura 3.2.1

Dal diagramma UML si può osservare che la classe implementa diversi metodi attraverso i quali è possibile gestire il dato che essa rappresenta; infatti, nel caso di una istruzione, i 24 bit che compongono il dato sono suddivisi in campi che hanno significati particolari[2] e può risultare utile leggerli.

Sono presenti inoltre metodi per confrontare il dato che essa rappresenta con un altro dato dello stesso tipo situato in un'altra parola in argomento:

- *lessThan(…)*;
- *greaterThan(…)*.

Inoltre il costruttore di default dichiara dati Word che saranno trattati come

[2] Rif. pag. 52 fig. 4.1 e pag. 56 fig. 4.2 tesi vCPU.

costanti, generalmente utili nei confronti:

- Word(-1);
- Word(0);
- Word(1).

Una particolare menzione la merita il secondo costruttore che implementa il seguente codice per la creazione di nuovi dati Word:

```
if(
  ((word & 0xFF000000) == 0xFF000000) ||
  ((word & 0xFF000000) == 0)
)
  _word = word;
else throw new OutOfBoundsException("……");
```

Figura 3.2.2

Bisogna innanzitutto fare una premessa: per java (ma anche in molti altri linguaggi di programmazione) non esistono "in natura" dati primitivi con segno, senza virgola, a 24 bit, ma esistono dati con caratteristiche simili, come indicati nella tabella seguente:

Tipo	Dimensione	Minima	Massima
Byte	8 bit	-2^7	2^7-1
Short	16 bit	-2^{15}	$2^{15}-1$
Int	32 bit	-2^{31}	$2^{31}-1$
Long	64 bit	-2^{63}	$2^{63}-1$

Tabella 3-1

Il dato che più si avvicina alle parole a 24 bit usate in ENIAC è quindi il dato di tipo intero, che però essendo a 32 bit necessità di qualche accorgimento nelle limitazioni superiore e inferiore.

Il codice appena citato (3.2.2) permette di riconoscere l'effettiva lunghezza della parola in complemento a due; quindi se il dato rappresentato dalla

parola è compreso nell'intervallo -8388608 e +8388607 (o comunque un valore che entra in 24 bit segno compreso) viene accettato, in caso contrario sarà lanciata un'eccezione.

Tipo	Dimensione	Minima	Massima
Word	24 bit	-2^{23}	$2^{23}-1$

Tabella 3-2

Esempio

Java lavora nativamente in complemento a due, perciò in presenza di un numero intero negativo rappresentabile in soli 24 bit, ne verrebbero occupati i primi 24 bit con il valore, mentre i bit successivi, da b24 a b31, verrebbero impiegati per l'estensione del segno. Si presenterebbe la seguente situazione:

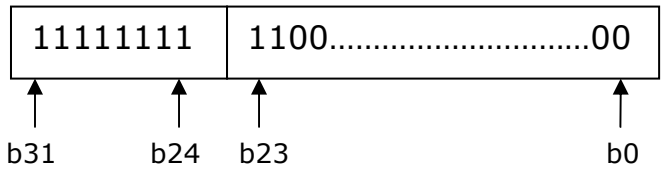


Figura 3.2.3

Adesso, caso opposto, un numero che per essere rappresentato ha bisogno di 29 bit (non accettabile), da b0 a b28; l'estensione del segno quindi va dal bit b29 al b31:

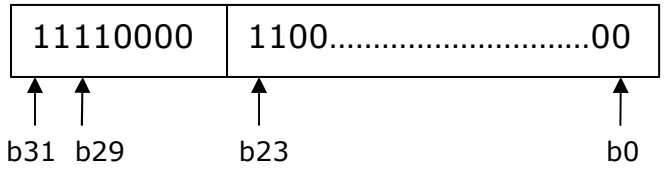


Figura 3.2.4

Il codice citato nella 3.2.2 fa esattamente quanto appena detto, cioè,

controlla gli 8 bit più significativi e, se li rileva come una semplice estensione del segno dei primi 24 bit, accetta il dato come parola valida; viceversa non viene accettato e quindi avverte tramite una eccezione di tipo *OutOfBoundsException*.

Allo stesso livello di *Word*, è anche situata la classe *Address*:

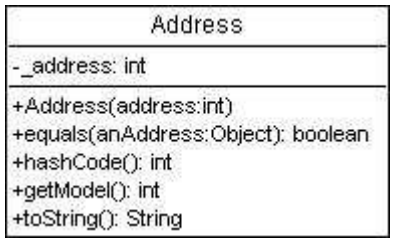


Figura 3.2.5

rappresenta un indirizzo ed è utilizzata per indicizzare le parole in memoria, nelle porte e nei registri; quando sarà necessario prelevare un valore presente in una delle risorse citate, sarà necessario fare riferimento ad un particolare *Address*.
E' importante sottolineare che il costruttore di *Address* accetta solo indirizzi compresi tra 0 e 4095, in quanto sono gli estremi delle celle di memoria e delle porte ammesse (dati tipo **N**), e comprendono anche le codifiche dei registri (dati tipo **R**).
Dato che gli *Address* sono usati all'interno di strutture dati come chiavi di ricerca (successivamente trattate), per rendere efficiente quest'ultima si è resa necessaria l'implementazione dei metodi:

```
public boolean equals(Object anAddress) {
        if (!(anAddress instanceof Address) || (anAddress == null))
                return false;
        else
                return (_address == ((Address) anAddress)._address);
}

public int hashCode() {
        return _address;
}
```

Figura 3.2.6

in quanto per ottenere un determinato dato, bisogna usare esattamente un dato di tipo *Address* che contenga esattamente lo stesso indirizzo.

3.3 Memoria e Porte

- package *eniac.resources*

La Memoria e le Porte (definite dalle classi *DefaultPorts* e *DefaultMemory*) sono ottenute mediante degli oggetti di tipo *Map*, più concretamente le *HashMap*.

Le *HashMap,* per essere più espliciti, sono delle strutture dati; la particolarità che le contraddistingue è la necessità di specificare due campi per ogni inserimento, uno per un determinato oggetto e l'altro per la sua chiave, quindi una coppia del tipo:

Key	Object

Tabella 3-3

Per ottenere l'oggetto è necessario fornire la chiave associata; tale caratteristica si è rilevata molto adatta per rappresentare diverse risorse di ENIAC.

Seguono quindi i loro schemi UML:

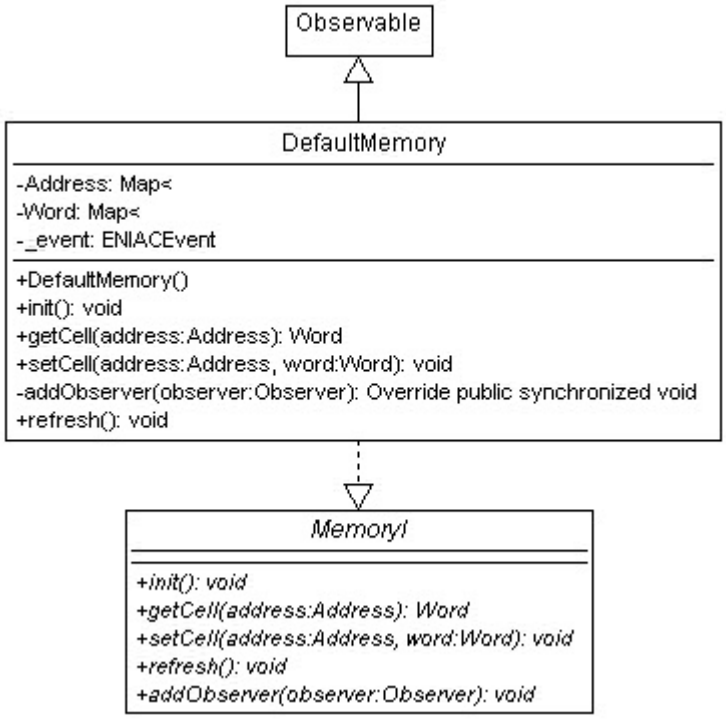


Figura 3.3.1

e quello relativo alle porte:

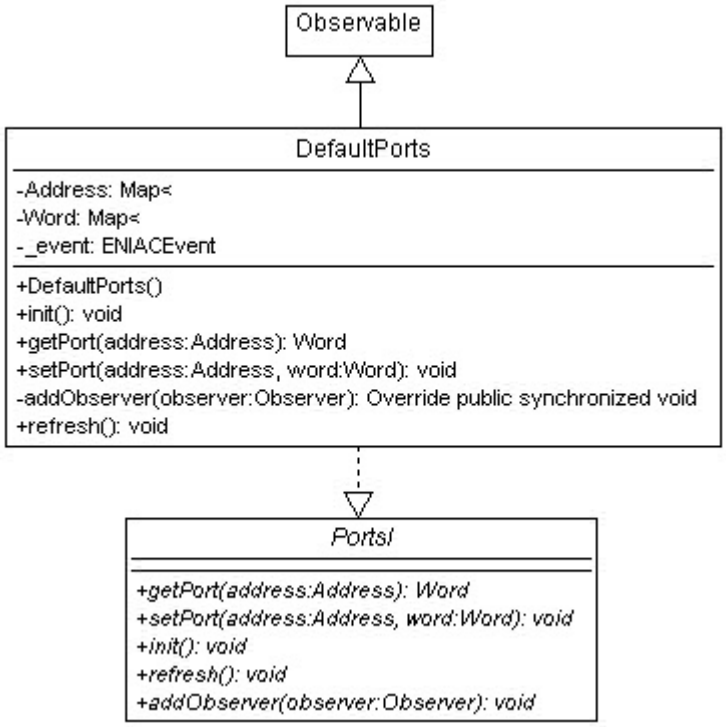


Figura 3.3.2

Come si può osservare, le porte (vedi figura 3.3.2), hanno la medesima struttura della memoria (vedi figura 3.3.1); all'interno delle *HashMap* sono usati come chiavi degli *Address,* del valore compreso tra 0 e 4095, e i dati corrispondenti a queste sono delle *Word*.
Le classi hanno alcuni metodi di gestione che permettono l'inserimento, il recupero di parole a determinati indirizzi e la inizializzazione:

- *init()*: inizializza tutto il contenuto della risorsa cancellando quanto precedentemente scritto;
- *refresh()*: effettua una lettura dell'intero contenuto della risorsa.

Patterns impiegati:

- *Interface*;
- *Observer*.

3.4 Registri

- package *eniac.vcpu.resources*

I Registri hanno la medesima struttura della memoria e delle porte ma con una limitazione: come precedentemente indicato nei paragrafi (1.5) e (1.7) i registri di ENIAC possono essere solo cinque, ognuno di loro contiene una sola parola di 24 bit ed è raggiungibile solo attraverso un indirizzo specifico, perciò per ogni registro è stato creato l'equivalente oggetto di tipo *Address* e inserito nella *HashMap*:

Registro	Indirizzo	Key
AX	0	Address(0)
BX	1	Address(1)
CX	6	Address(6)
DX	7	Address(7)
PC	4095	Address(4095)

Tabella 3-4

Infine, attraverso un accorgimento nel metodo di inserimento dei dati nel registro, viene effettuato un controllo sull'effettiva veridicità dell'indirizzo del registro:

```
public void setRegister(Address register, Word value)
        throws InexistentRegisterException {
        if(_registers.containsKey(register)){
                (……)
        }
        else throw new InexistentRegisterException("…");
}
```

Figura 3.4.1

nel caso non sia un indirizzo pertinente a quelli effettivi verrà scartato l'inserimento e generata una eccezione di tipo *InexistentRegisterException.*
La classe possiede alcuni metodi di gestione, che permettono l'inserimento, il recupero e la reinizializzazione:

- *init()*: inizializza tutto il contenuto della risorsa cancellando quanto precedentemente scritto;
- *refresh()*: effettua una lettura dell'intero contenuto della risorsa.

Patterns usati:

- *Interface*;
- *Observer*.

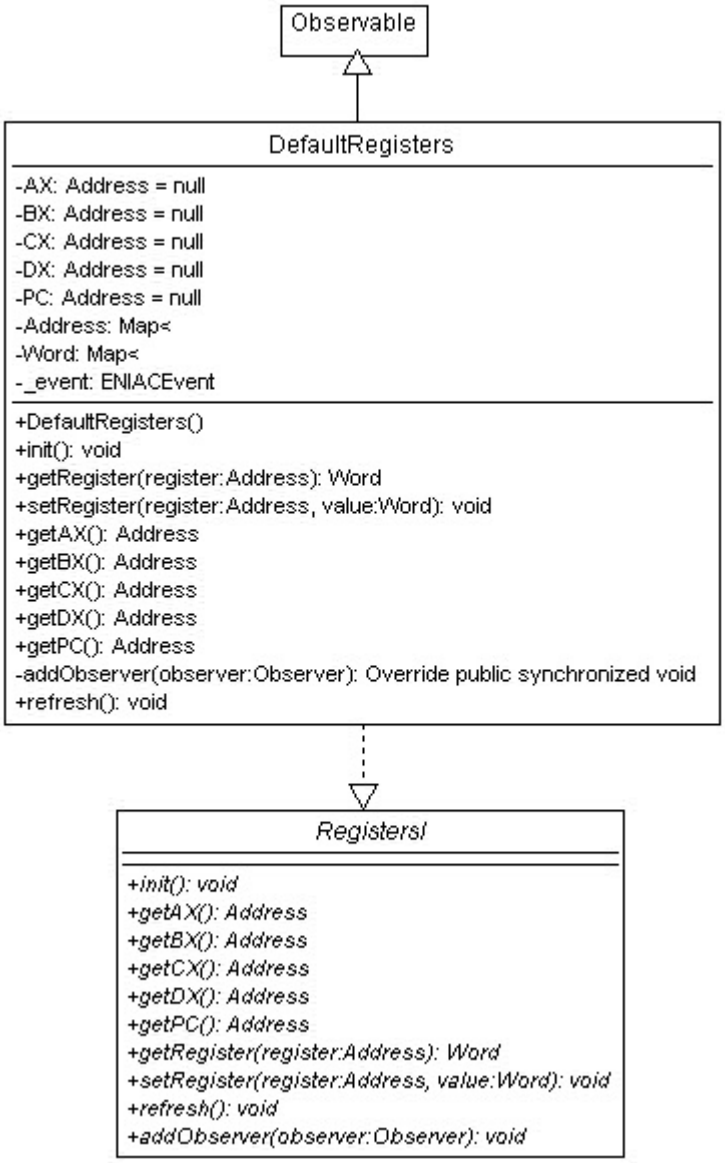

Observable
DefaultRegisters
-AX: Address = null
-BX: Address = null
-CX: Address = null
-DX: Address = null
-PC: Address = null
-Address: Map<
-Word: Map<
-_event: ENIACEvent
+DefaultRegisters()
+init(): void
+getRegister(register:Address): Word
+setRegister(register:Address, value:Word): void
+getAX(): Address
+getBX(): Address
+getCX(): Address
+getDX(): Address
+getPC(): Address
-addObserver(observer:Observer): Override public synchronized void
+refresh(): void
RegistersI
+init(): void
+getAX(): Address
+getBX(): Address
+getCX(): Address
+getDX(): Address
+getPC(): Address
+getRegister(register:Address): Word
+setRegister(register:Address, value:Word): void
+refresh(): void
+addObserver(observer:Observer): void

Figura 3.4.2

3.5 Flags

- package *eniac.vcpu.resources*

La struttura dei Flags è leggermente semplificata rispetto alle precedenti. Infatti, come visto nel paragrafo (1.6), detti flags sono cinque, hanno la dimensione di 1 bit ed è sufficiente usare dei valori boolean per rappresentarli (tra l'altro in java i boolean hanno proprio la dimensione di 1 bit):

- Se true, il flag è a 1;
- Se false, il flag è a 0.

Questa scelta è anche di convenienza, non è necessario usare addirittura una *HashMap,* visto che quest'ultima costa molto più in termini di memoria oltre che le operazioni di mantenimento hanno una complessità maggiore penalizzando le prestazioni.

La classe possiede alcuni metodi di gestione, che permettono l'inserimento, il recupero e la inizializzazione:

- *init()*: inizializza tutto il contenuto della risorsa cancellando quanto precedentemente scritto;
- *refresh()*: effettua una lettura dell'intero contenuto della risorsa.

Patterns usati:

- *Interface*;
- *Observer*.

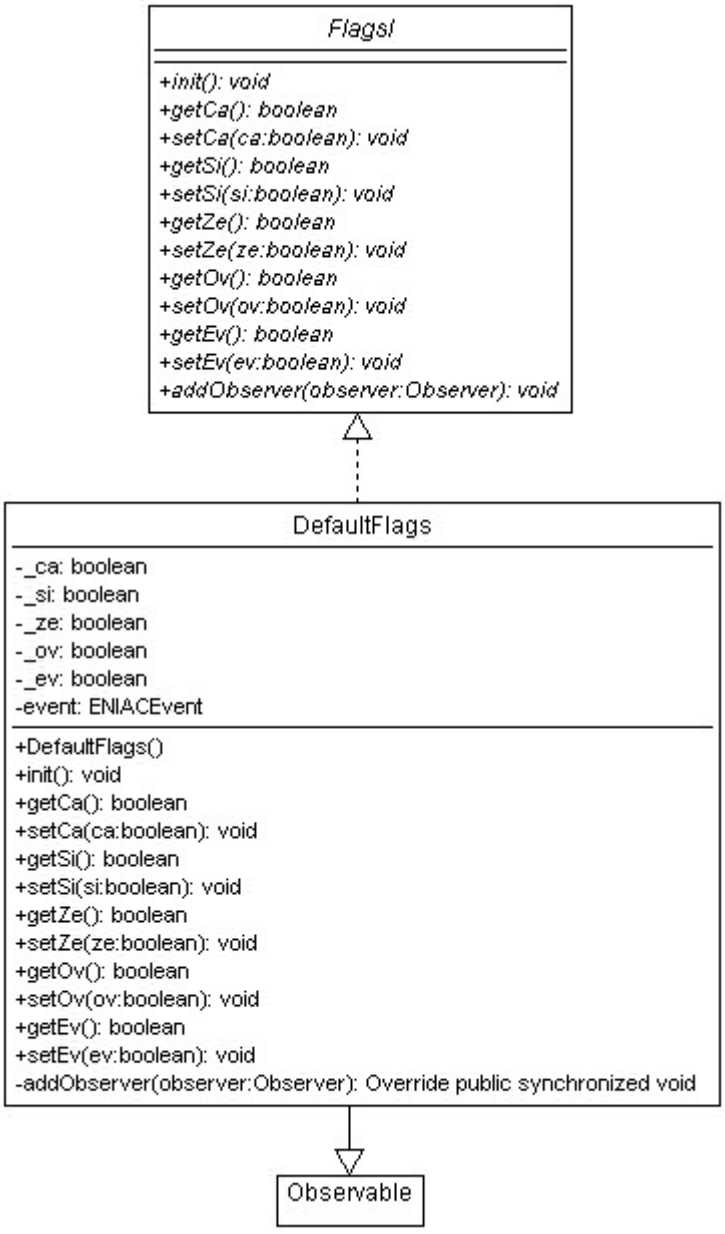

FlagsI
+init(): void
+getCa(): boolean
+setCa(ca:boolean): void
+getSi(): boolean
+setSi(si:boolean): void
+getZe(): boolean
+setZe(ze:boolean): void
+getOv(): boolean
+setOv(ov:boolean): void
+getEv(): boolean
+setEv(ev:boolean): void
+addObserver(observer:Observer): void
DefaultFlags
-_ca: boolean
-_si: boolean
-_ze: boolean
-_ov: boolean
-_ev: boolean
-event: ENIACEvent
+DefaultFlags()
+init(): void
+getCa(): boolean
+setCa(ca:boolean): void
+getSi(): boolean
+setSi(si:boolean): void
+getZe(): boolean
+setZe(ze:boolean): void
+getOv(): boolean
+setOv(ov:boolean): void
+getEv(): boolean
+setEv(ev:boolean): void
-addObserver(observer:Observer): Override public synchronized void
Observable

Figura 3.5.1

3.6 A.L.U.

- package *eniac.vcpu.resource*

L'ALU, vero e proprio cervello di ENIAC, esegue tutte le operazioni accessibili dal set di istruzioni. Dato che ENIAC viene concepito come un simulatore, al suo interno l'ALU esegue le operazioni attraverso gli operatori aritmetici di java, e non bit a bit come si potrebbe pensare.
Inoltre poiché lo stato dei flags dipende dai risultati delle operazioni, i flags sono calcolati sempre all'interno della ALU.

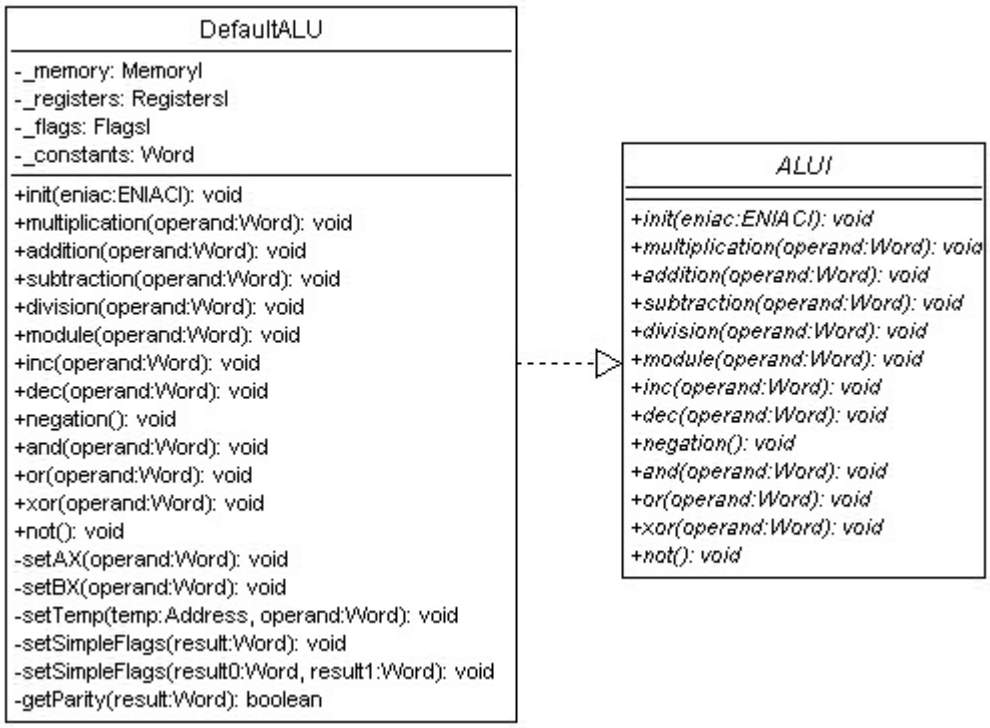


Figura 3.6.1

Come è possibile constatare dal diagramma UML (figura 3.6.1) della classe *DefaultALU*, l'ALU ha completo accesso all'hardware di ENIAC perché le operazioni eseguite dall'ALU solitamente coinvolgono i registri come l'accumulatore (o registro AX), la memoria e i flags.
Sebbene come comportamento generale le operazioni svolte dall'ALU lavorino sull'operando presente nell'accumulatore (precedentemente

caricatovi manualmente, o risultato di altra operazione e ivi rimasto) e sull'operando passato in argomento al metodo dell'operazione, per poi salvare nuovamente il risultato nell'accumulatore, ci sono operazioni che meritano menzioni a parte.

Nella specifica di vCPU le istruzioni INC e DEC[3] operano direttamente sul dato passatogli per argomento, eventualmente lasciando immutato il valore presente in AX; per questo motivo il metodo *inc(…)*, ma anche *dec(…)*, indicato in figura 3.6.1 ed esplicitato nella figura 3.6.2, si appoggia a una cella di memoria salvando mediante il metodo *setTemp(…)* l'incremento (o il decremento) nella cella di memoria 4095.

```
public void inc(Word operand)
        throws OutOfBoundsException, InexistentRegisterException {
                (……)
                Address AddrTemp = new Address(4095);
                setTemp(AddrTemp, result);
}
```

Figura 3.6.2

Un'altra menzione particolare la merita il multiplier; da un punto di vista aritmetico una moltiplicazione tra due operandi a 24 bit:

$$Mul = Op1 \times Op2 = 2^{24} \times 2^{24} = 2^{24+24} = 2^{48}$$

un dato troppo grande che deve essere gestito.

Nella tesi di vCPU[4] c'è una logica di controllo; in ENIAC il moltiplicatore effettua quanto segue: il prodotto è salvato in un dato di tipo *long* (a 64 bit) e poichè java in automatico lavora in complemento a due, estende il bit di

[3] Rif. pag. 67, tesi vCPU.

[4] Rif. pag. 44 e pag. 45, tesi vCPU.

segno per tutti i 16 bit rimanenti; però come sappiamo il risultato può riempire al massimo i primi 48 bit, perciò attraverso l'operatore di *bitwise* (>>) vengono create due parole: la prima con 24 bit più significativi (MSB) e l'altra con 24 bit meno significativi (LSB).

```
(……..)
resultHigh.equals(_constants.MINUS_ONE)&& resultLow.lessThan(_constants.ZERO)
(……..)
```

Figura 3.6.3

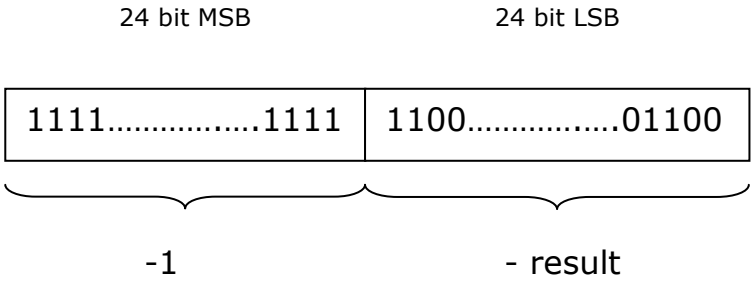


Figura 3.6.4

A questo punto è necessario comprendere se la parola con 24 bit MSB è soltanto un'estensione di segno di quella con 24 bit LSB; i controlli sono eseguiti per mezzo di metodi propri degli oggetti Word, e sono mirati a verificare il segno e la dimensione delle parole.
Il codice in figura 3.6.3 controlla la veridicità di due condizioni: se la parola nei 24 bit MSB, ha valore -1 (quindi 24 volte il valore 1, perciò potrebbe essere solo un'estensione di segno) e se nei 24 bit LSB il valore della parola è minore di 0 (cioè negativa, conferma quanto detto nell'MSB) come presentato in figura 3.6.4; se entrambe sono vere allora è sufficiente solo la parola nei 24 bit LSB per rappresentare il valore del risultato negativo .

```
(……..)
(resultHigh.equals(_constants.ZERO) &&
(resultLow.equals(_constants.ZERO) || resultLow.greatherThan(_constants.ZERO)))
(……..)
```

Figura 3.6.5

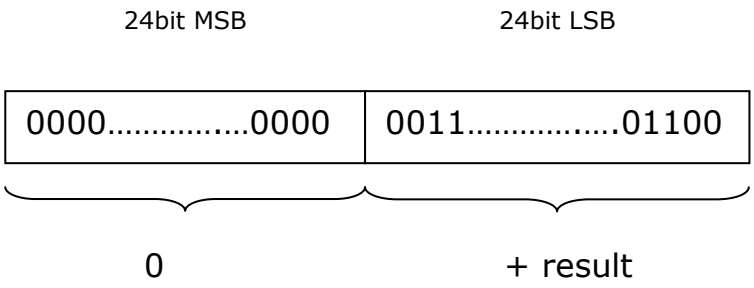


Figura 3.6.6

Viceversa il codice in figura 3.6.5 effettua il controllo sulla positività; se la parola nei 24 bit MSB vale 0 (24 bit tutti di valore 0, quindi positivo) ed è anche vero che la parola nei 24 bit LSB vale anch'essa esattamente 0, o comunque un valore maggiore di 0 (perciò positivo come l'MSB), allora sono sufficienti i 24 bit LSB per rappresentare il risultato positivo, come l'esempio mostrato in figura 3.6.6.

Nei casi delle figure 3.6.4 e 3.6.6 sarà sufficiente quindi salvare il contenuto dei 24 bit LSB nel solo registro AX attraverso il metodo *setAX(…)*, e calcolare sempre e solo su questa parola i flags.

In tutti gli altri casi, è richiesto il salvataggio in due registri (AX e BX) mentre i flags verranno calcolati in base alle due parole.

Per calcolare i flags nell'ALU, sono presenti dei metodi *private* (figura 3.6.7) perché l'uso è riservato all'ALU stessa in quanto sono definiti in base al risultato di un'operazione; tutte le operazioni hanno effetto sullo stato dei flags semplici mentre solo multiplication(…), addition(…), subtraction(…), inc(..) e dec(…) hanno effetto su quelli complessi (vedi par. 1.6).

```
private void setSimpleFlags(Word result) {
        _flags.setZe(result.equals(_constants.ZERO));
        _flags.setSi(result.lessThan(_constants.ZERO));
        _flags.setEv(getParity(result));
}

O anche (usato solo dal multiplier):

private void setSimpleFlags(Word result0, Word result1) {
        _flags.setZe(result0.equals(_constants.ZERO) &&
                result1.equals(_constants.ZERO));
        _flags.setSi(result0.lessThan(_constants.ZERO));
                _flags.setEv(getParity(result0) == getParity(result1));
}
```

Figura 3.6.7

3.7 Istruzioni

- package *eniac.vcpu.instruction*

Come indicato nel paragrafo 1.7, vCPU riconosce due tipi di Istruzioni: alpha e beta; attorno a questa definizione è stato progettato il formato di istruzioni di ENIAC.

E' stato opportuno stabilire una struttura comune a tutte le istruzioni, dove ai livelli più bassi della gerarchia venissero stabilite delle specializzazioni; inoltre le classi delle istruzioni dovrebbero essere in grado di operare sull'hardware e avere una codifica che le identificasse.

Per questi criteri, si è scelto di utilizzare un'interfaccia comune di nome *InstructionI*, dotata di due metodi che devono essere implementati dalle classi delle istruzioni:

```
public interface InstructionI {

        public void perform(ENIACI eniac, Word argument)
                throws OutOfBoundsException,
                        InexistentRegisterException,
                        EndOfProgramException,
                        DivisionByZeroException;

        public int getOpCode();

        public String getOpRegex();

}
```

Figura 3.7.1

I metodi sottoposti alla procedura dell'implementazione, dovranno svolgere le seguenti funzioni:

- *perform(…)*: conoscendo l'hardware e una parola su cui operare, esegue la funzione associata all'istruzione;
- *getOpCode()*: restituisce il codice operativo (i bit dal b13 al b23) che identifica l'istruzione;
- *getOpRegex()*: restituisce una stringa contenente un'espressione regolare adatta al riconoscimento dell'istruzione e del suo argomento, ad esempio se "ADD 100" oppure "ADD @100".

Una trattazione nel dettaglio di questi avverrà nel capitolo 5.

Tutte le istruzioni di ENIAC, di tipo alpha e beta, devono quindi implementare questi due metodi, ovviamente con le varianti del caso; segue un esempio della gerarchia inerente alle istruzioni "ADD" e "JMP":

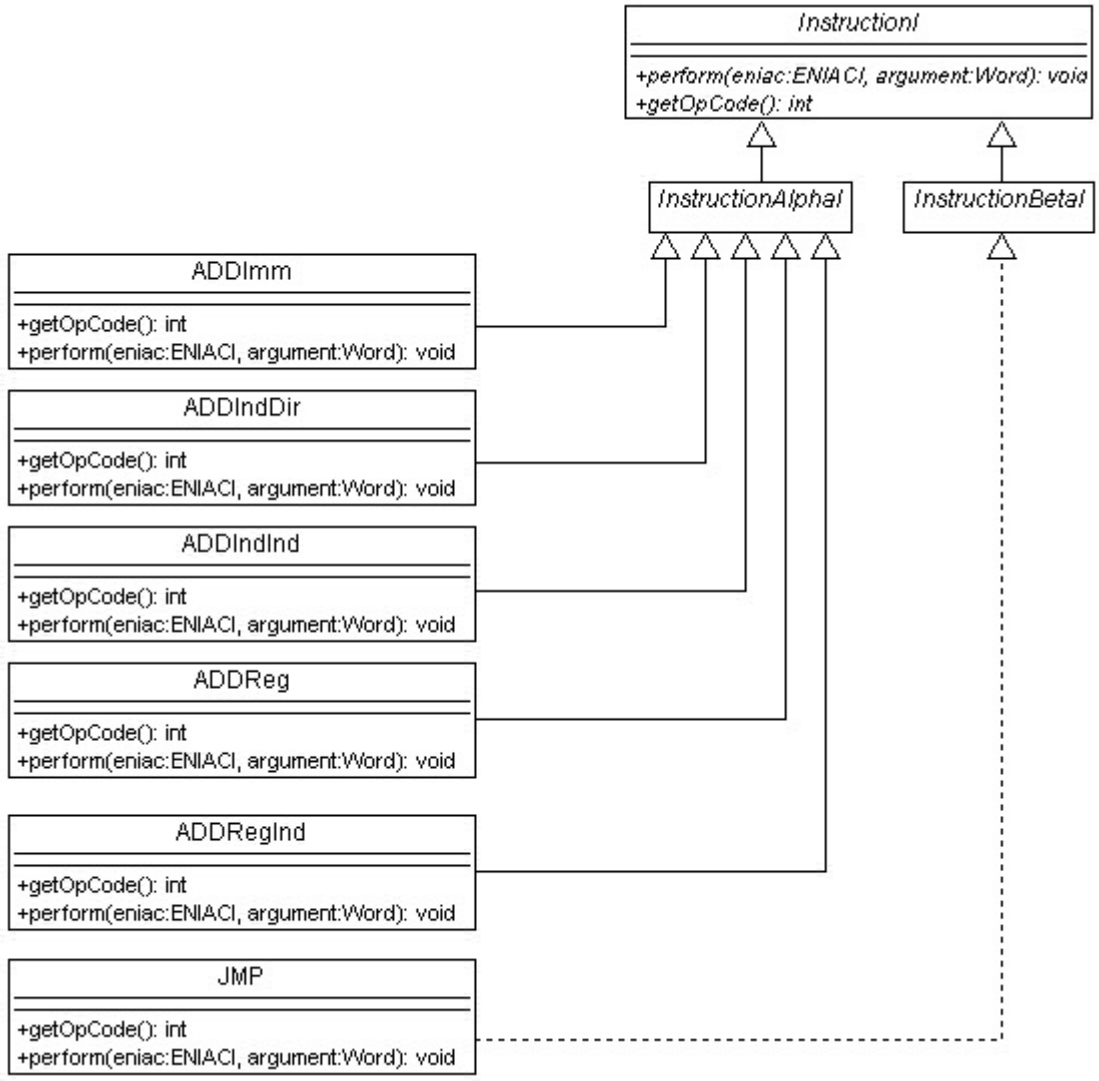


Figura 3.7.2

Nella figura 3.7.2, ad esempio, sono indicate le istruzioni ADD e JMP rispettivamente di tipo alpha e beta; allo scopo di specializzarle, oltre che per imprimere una maggiore chiarezza e facilitare l'implementazione di nuove istruzioni, sono state dichiarate altre due interfacce *InstructionAlphaI* e *InstructionBetaI* che estendono *InstructionI* (figura 3.7.3):

```
Tipo Alpha:

        public interface InstructionAlphaI extends InstructionI {

        }

Tipo Beta:

        public interface InstructionBetaI extends InstructionI {

        }
```

Figura 3.7.3

Come può essere osservato il corpo è vuoto.

Tutto questo per un valido motivo: definire uno standard comune alle istruzioni dei due tipi, senza però perdere la natura e quindi l'identificabilità del tipo.

Ulteriori dettagli e precisazioni sulla gerarchia delle istruzioni sarà discussa nel capitolo 4.

3.8 Instruction Factory

- package *eniac.vcpu.instruction*

Trattandosi di una factory a tutti gli effetti merita una trattazione più adeguata, fornita nel paragrafo 4.2: il pattern Factory Method.

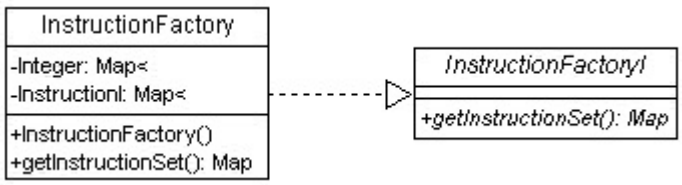


Figura 3.8.1

Lo scopo è restituire un oggetto di tipo *HashMap,* in cui sono caricate coppie del tipo in tabella 3-5:

Key	Object
int	InstructionI

Tabella 3-5

perciò una coppia per ognuna delle istruzioni presenti nel package *eniac.vcpu.instruction.module*; la chiave è l'intero che viene ritornato dal metodo *getOpCode()* identificato al par. 3.8.

3.9 Dispositivi

- package *device*

I Dispositivi rappresentano funzionalità aggiuntive che possono essere programmate ed interfacciate alle risorse di ENIAC a seconda delle necessità. Fondamentalmente i dispositivi hanno accesso in lettura ed in scrittura sulle porte. Per implementare correttamente un dispositivo è necessario attenersi all'interfaccia *DeviceI*:

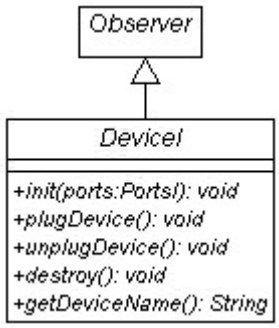


Figura 3.9.1

Questa interfaccia consta di cinque metodi:

- *init(PortsI)*: invocato quando si vuole inizializzare il dispositivo; vengono inoltre passate le porte con le quali il dispositivo deve interagire;
- *plugDevice()*: invocato quando il dispositivo viene collegato;
- *unplugDevice()*: invocato quando il dispositivo viene scollegato;
- *destroy()*: invocato quando il dispositivo viene spento;
- *getDeviceName()*: restituisce l'etichetta associata al dispositivo.

Gli oggetti implementanti l'interfaccia *DeviceI*, per definizione, sono situati nel package *device.module*.

3.10 Device Factory

- package *device*

La Device Factory è identica all'instruction factory e verrà anch'essa trattata nel paragrafo 4.2:

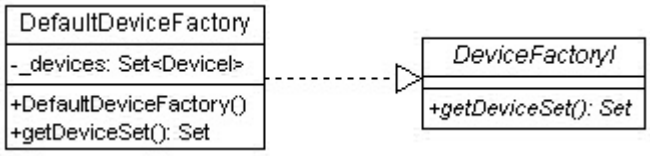


Figura 3.10.1

Come si evince dalla figura 3.10.1, questo tipo di factory possiede un solo metodo:

- *getDeviceSet()*: restituisce un oggetto di tipo *Set*, ovvero una collezione di dispositivi di tipo *DeviceI* (vedi parag. 3.9).

3.11 Decoder

- package *eniac.vcpu*

Il Decoder è così chiamato perché teoricamente decodifica la parola in memoria per poi rintracciarne l'istruzione corrispondente; in pratica confronta il codice operativo di una generica parola passata (quindi i bit dal b13 al b23), con le istruzioni effettivamente supportate da ENIAC.

La parola nel caso di ENIAC è stata precedentemente prelevata dall'Executer (vedi parag. 3.12) mentre le istruzioni sono conosciute tramite una *HashMap* precedentemente popolata dalla factory delle istruzioni (vedi par. 3.8 e 4.2).

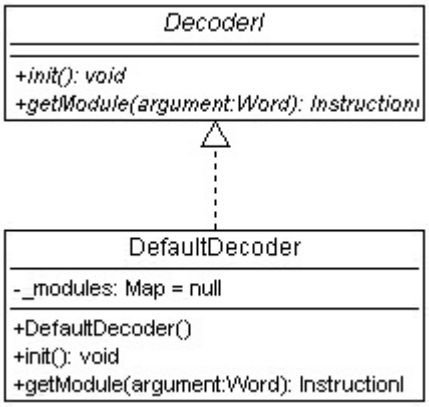


Figura 3.11.1

La ricerca avviene per mezzo del metodo *getModule(…)*, nel quale viene usato il metodo *get(…)* fornito dalle *HashMap;* nel caso quest'ultimo trovasse corrispondenza tra una chiave e il codice operativo passato in argomento, verrà restituito un oggetto di tipo *InstructionI* (vedi par 3.7).
Segue il codice del metodo *getModule(…)*:

```
public InstructionI getModule(Word argument) throws DecoderException {
        int opcode = argument.getOpcode();
        int type = argument.getArgumentType();
        int format = (opcode | type);
        InstructionI module = (InstructionI)_modules.get(format);
        if(module != null)
                return module;
        else throw new DecoderException(…);
}
```

Figura 3.11.2

Il codice operativo dell'istruzione (*format*) viene ottenuto tramite due metodi forniti dalla classe *Word* precedentemente accennati nel diagramma UML di figura 3.2.1:

- *getOpCode()*: restituisce i bit dal b16 al b23;
- *getArgumentType()*: restituisce i bit dal b13 al b15.

Come evidenziato dalla figura 3.11.2 viene restituita una *DecoderException* in caso che l'istruzione non sia valida.

3.12 Executer

- package *eniac.vcpu*

L'Executer ha il compito di coordinare il funzionamento dei vari dispositivi progettati e implementati all'interno di ENIAC; il suo scopo principale è quello di eseguire il ciclo di Fetch - Decode – Execute, attraverso il metodo *step()*.

L'Executer, mediante lo *step()*, rappresenta lo strumento tramite cui un'applicazione esterna, come la GUI del simulatore, potrà eseguire i programmi scritti in linguaggio assembly di vCPU.

Inoltre è l'Executer ad inizializzare risorse quali l'ALU e il Decoder, in quanto le usa direttamente.

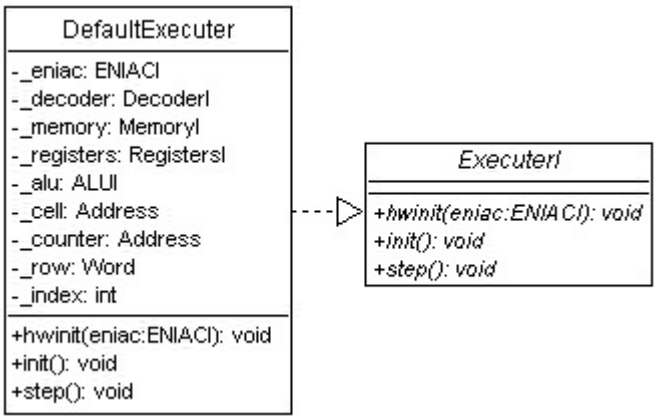


Figura 3.12.1

Descrizione delle fasi:

- Fetch: legge la parola contenuta dal registro PC; essendo questa riconducibile a un indirizzo permetterà di prelevare dalla memoria il dato che dovrà essere processato;
- Decode: usa il decoder per riconoscere l'istruzione ed localizzare il codice relativo;
- Execute: esegue il codice specifico dell'istruzione.

Alcuni particolari meritano attenzione: dopo la fase di Execute si procede a un controllo avente lo scopo di verificare se, a causa dell'esecuzione

dell'ultima istruzione, il PC è cambiato; in caso negativo sarà l'Executer ad incrementare il PC all'indirizzo della cella di memoria successiva all'ultima eseguita. Il motivo va ricercato in istruzioni come i salti, esempio il "JMP N", i quali possono modificare il contenuto del registro PC per puntare a una cella N e cambiare così il flusso di esecuzione.
Altro aspetto importante è costituito dalle eccezioni: infatti in caso di un errore verificatosi in una qualsiasi componente di ENIAC durante il ciclo di Fetch – Decode – Execute passerà attraverso l'Executer per poi essere rimappata all'esterno, per esempio all'interfaccia grafica.
Vi è inoltre un metodo *init()* che ha la funzione di reimpostare il PC per puntare la prima cella (*Address(0)*) della memoria.

3.13 Eccezioni

Tra i vari package di ENIAC ne sono presenti diversi denominati *exception* (vedi figura 3.1.1); essi contengono le eccezioni che saranno eventualmente sollevate dai vari oggetti.
Una nota la merita però l'eccezione *VCPUException* nel package *eniac.vcpu.exception*; questa è l'eccezione base di tutte le eccezioni generate da ENIAC, in quanto tale tutte le altre dentro i vari package *exception* estendono questa eccezione.
VCPUException è creata estendendo la classe *Exception*; questo pone in evidenza una scelta progettuale, cioè che *VCPUException* e tutte le altre saranno pertanto eccezioni di tipo *checked*: fattore che comporta una gestione attenta del flusso del programma.

3.14 Riepilogo

Allo stato attuale, la situazione può essere riassunta dal seguente schema:

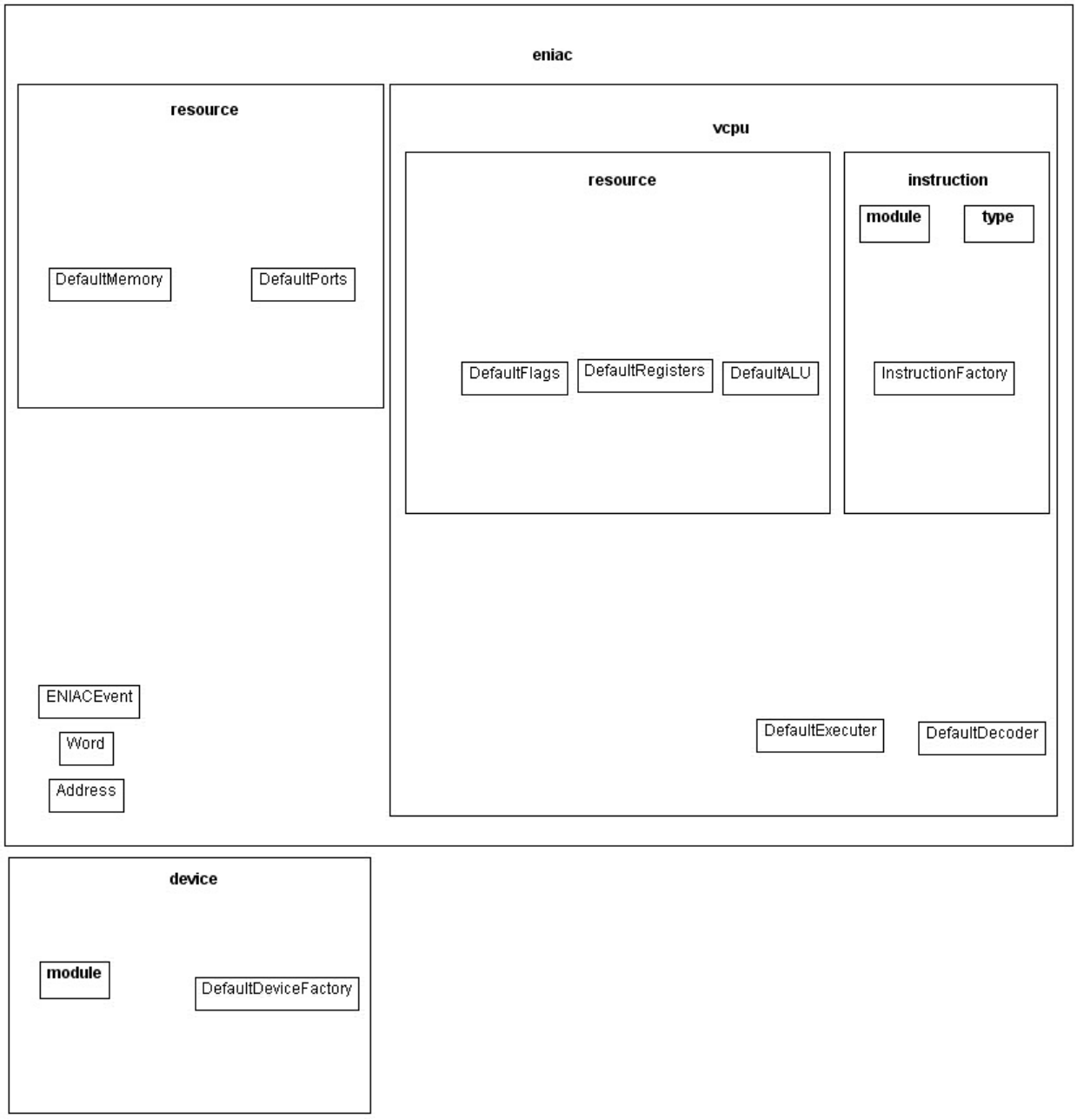

eniac
resource
DefaultMemory
DefaultPorts
vcpu
resource
DefaultFlags
DefaultRegisters
DefaultALU
instruction
module
type
InstructionFactory
ENIACEvent
Word
Address
DefaultExecuter
DefaultDecoder
device
module
DefaultDeviceFactory

Figura 3.14.1

4 I Design Patterns in ENIAC

Descrizione dell'applicazione di alcuni patterns nell'architettura di ENIAC.

4.1 Interface

Le classi che costituiscono il simulatore, normalmente implementano una propria interfaccia (tabella 4-1):

Risorsa	Interfaccia
DefaultMemory	MemoryI
DefaultPorts	PortsI
DefaultDecoder	DecoderI
DefaultExecuter	ExecuterI
DefaultInstructionFactory	InstructionFactoryI
DefaultALU	ALUI
DefaultRegisters	RegistersI
DefaultFlags	FlagsI
ENIAC	ENIACI
DefaultVCPU	VCPUI
DefaultDeviceFactory	DeviceFactoryI

Tabella 4-1

Generalmente da una lettura del codice di una qualsiasi classe nella quale venga fatto uso di risorse di ENIAC, si può osservare come, quando venga creata un'istanza delle classi il cui nome è presente nella colonna "Risorsa", si faccia riferimento ad un oggetto il cui tipo è della corrispondente interfaccia (figura 4.1.1):

```
(……..)
MemoryI _memory = new DefaultMemory();
(……..)
```

Figura 4.1.1

In questo modo si garantisce l'indipendenza dell'oggetto creato da una particolare istanza; perciò ogni volta che verrà utilizzato l'oggetto si farà riferimento all'interfaccia piuttosto che all'oggetto stesso: questo viene definito come disaccoppiamento dall'oggetto stesso.
La situazione perciò rispetto al diagramma 3.14.1 si evolve nel seguente:

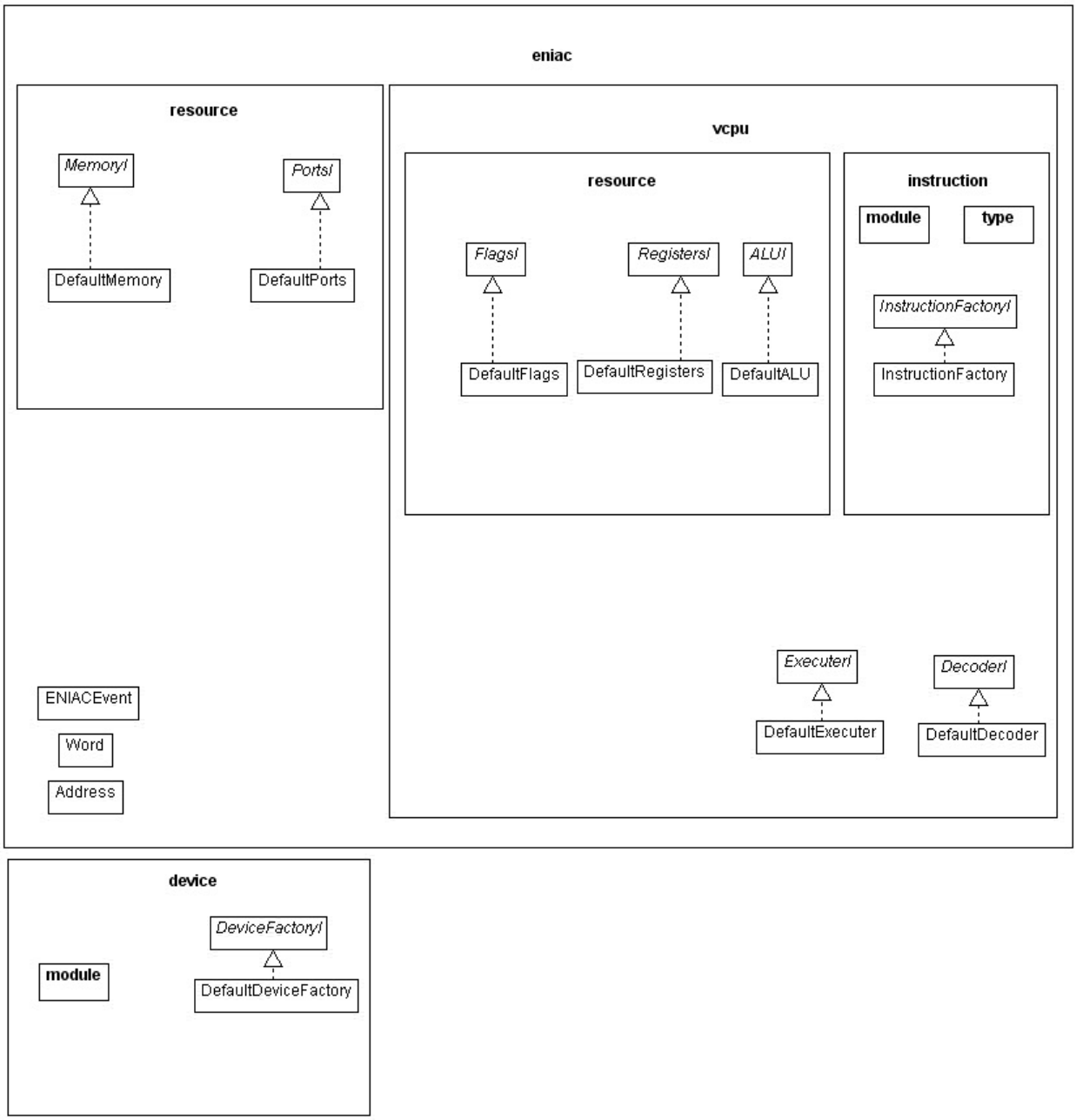


Figura 4.1.2

4.2 Factory method

In ENIAC sono presenti più Factory Method; generalmente sono presenti in package dove sono locate risorse di ENIAC. Lo scopo delle factories è descrivere un "ponte" attraverso il quale poter utilizzare le risorse di ENIAC presenti in quel package; segue un esempio di factory per il package *eniac.resource*:

```
public class ResourceFactoryS {
        public static MemoryI getMemory() {
                MemoryI memory = new DefaultMemory();
                return memory;
        }
        public static PortsI getPorts() {
                PortsI ports = new DefaultPorts();
                return ports;
        }
}
```

Figura 4.2.1

La factory in figura 4.2.1 permette, attraverso due metodi *static*, di avere accesso diretto sia alla memoria che alle porte di ENIAC; tutte le altre factory sono concepite sullo stesso principio, se venisse aggiunta una nuova risorsa in un package si tratterebbe di aggiornare la factory relativa a questo.

Un caso particolare è presentato dai package *eniac.vcpu.instructions* e *device*, in quanto come detto precedentemente all'interno di questi vi sono rispettivamente le interfacce *InstructionI* e *DeviceI* (vedi parag. 3.7 e 3.9), che vengono implementate da tutte le istruzioni e i dispositivi che vogliono essere riconosciuti come validi da ENIAC; in ognuno di questi package è presente una seconda factory, la *DefaultInstructionFactory* e la *DefaultDeviceFactory,* con il compito di leggere tutti i file presenti nel package *eniac.vcpu.instructions* e *device,* e di istanziare una alla volta tutte

le istruzioni nei rispettivi packages *module*, con chiamate come ad esempio la seguente:

```
(……..)
module = Class.forName("eniac.vcpu.instruction.module.JMP"));
InstructionI instance = (InstructionI)module.newInstance();
(……..)
```

Figura 4.2.2

La figura 4.2.2 mostra l'esempio specifico per l'istruzione "JUMP". L'architettura generale si sviluppa ulteriormente, con le sei factories il cui nome termina per *FactoryS*, più le due factories particolari appena descritte. Una nota sull'implementazione: si sarebbe potuto scrivere questi factory method aggiungendovi poche righe di codice per ottenere il comportamento del pattern *singleton*. In pratica la factory restituisce all'oggetto chiamante la stessa sempre le medesima istanza della risorsa; in altre parole usare più simulatori ENIAC sulle stesse risorse hardware. L'idea, però, a seguito di riflessioni è stata accantonata per permettere la coesistenza di più simulatori sulla stessa macchina: ad esempio per l'uso di un dispositivo comune, connesso alle porte di due diversi simulatori.

Ricapitolando:

Package	Factory
eniac	ENIACFactoryS
eniac.resource	ResourceFactoryS
eniac.vcpu	VCPUFactoryS
eniac.vcpu.resouce	VCPUResourceFactoryS
eniac.vcpu.instruction	InstructionFactoryS
eniac.vcpu.instruction.module	DefaultInstructionFactory
device	DefaultDeviceFactory

Tabella 4-2

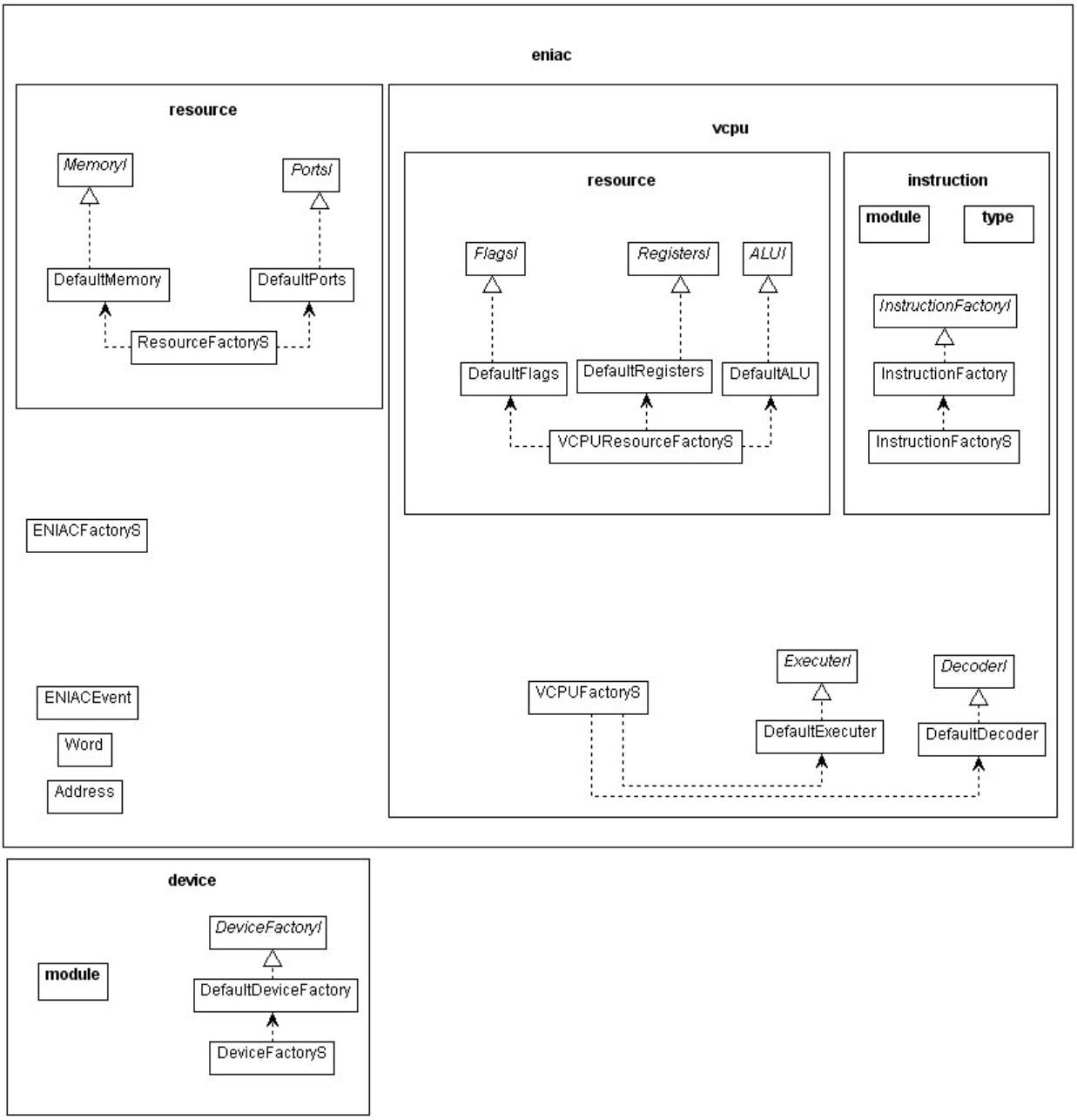

eniac
resource
MemoryI
PortsI
DefaultMemory
DefaultPorts
ResourceFactoryS
vcpu
resource
FlagsI
RegistersI
ALUI
DefaultFlags
DefaultRegisters
DefaultALU
VCPUResourceFactoryS
instruction
module
type
InstructionFactoryI
InstructionFactory
InstructionFactoryS
ENIACFactoryS
ENIACEvent
Word
Address
VCPUFactoryS
ExecuterI
DecoderI
DefaultExecuter
DefaultDecoder
device
module
DeviceFactoryI
DefaultDeviceFactory
DeviceFactoryS

Figura 4.2.3

4.3 Locator

La funzione dei Locator in ENIAC può essere vista come un bus di sistema per comunicare con l'hardware: offre un canale di comunicazione per localizzare tutte le factories e usare le risorse istanziate tramite queste ultime. Come detto nel 4.2 le factories sono costituite da metodi *static*, il compito dei Locator è usare questi e restituire oggetti rappresentanti risorse usabili come le porte:

```
public class ENIAC implements ENIACI {
        private PortsI _ports;
        private VCPUI _vcpu;
        public ENIAC() {
                _ports = ResourceFactoryS.getPorts();
                _vcpu = VCPUFactoryS.getVCPU();
        }
        public PortsI getPorts(){
                return _ports;
        }
        public VCPUI getVCPU(){
                return _vcpu;
        }
        (……)
}
```

Figura 4.3.1

La figura 4.3.1 mostra lo localizzazione delle porte, così come il metodo che restituirà l'oggetto richiesto (le porte) alla classe richiedente (ad esempio l'Executer). E' importante sottolineare alcune scelte implementative; i Locator per motivi di ordine, sono due:

- *ENIAC*: situato nel package *eniac*, implementa l'interfaccia *ENIACI*; localizza tutte le risorse del simulatore;
- *DefaultVCPU*: situato nel package *eniac.vcpu*, implementa l'interfaccia *VCPUI*; normalmente localizza solo quelle risorse che possono essere considerate parte della CPU, e se una classe esterna vuole accedervi

bisogna farlo attraverso il Locator *ENIAC* (nella figura 4.3.1 sono presenti i metodi che effettuano questo collegamento).

Per concludere entrambi implementano una propria interfaccia, per i motivi del paragrafo 4.1 e inoltre ambedue sono raggiungibili attraverso rispettive factories dei package di appartenenza. In figura 4.3.2 sono aggiunti i Locator.

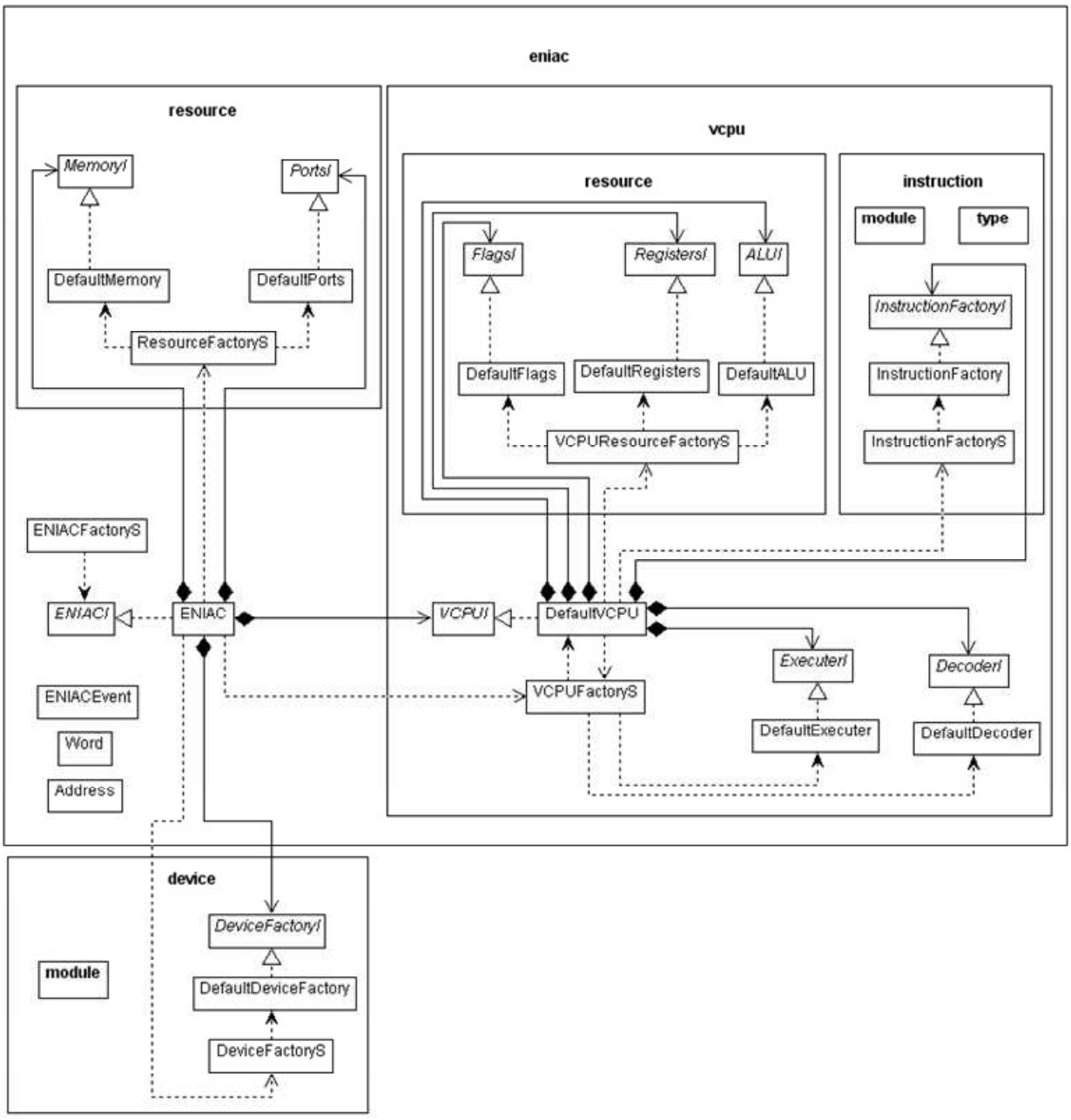


Figura 4.3.2

4.4 Implementation

- package *eniac.vcpu.instruction.type*

Nel diagramma UML di figura 3.7.1 è presente (volutamente) una discrepanza; le classi delle istruzioni di tipo alpha dovrebbero implementare l'interfaccia *IstructionAlphaI*, mentre nello schema è indicata una relazione di ereditarietà.

Il motivo va ricercato nella mancanza di un livello di astrazione; tra le classi concrete (l'istruzione) e l'interfaccia (*IstructionAlphaI*) è presente una classe astratta che mette a disposizione delle classi concrete dei metodi che servono a specializzarle.

Perché astratta? - Come descritto nel paragrafo 1.7 sono presenti cinque possibili tipi di argomento per le istruzioni di tipo alpha e ogni classe astratta presente nel package ha proprio il compito di far conoscere a ogni istruzione che la estende il suo tipo di argomento; quindi darle la conoscenza del tipo di dato che essa sarà in grado di manipolare, ma senza per questo stravolgere l'architettura di fondo:

Classe astratta	Tipo di argomento	Identificativo
ImmA	Immediato	Z
IndDirA	Indirizzo Diretto	@N
IndIndA	Indirizzo Indiretto	@@N
RegA	Registro	R
RegIndA	Registro Indiretto	@R

Tabella 4-3

All'interno di ognuna queste classi astratte sono presenti tre metodi:

- *getTypeCode()*: restituisce il codice operativo (compreso tra i bit da b13 a b15) che corrisponde al tipo di argomento trattato;
- *getTypeRegex()*: restituisce una stringa contenente una espressione regolare adatta a riconoscere il tipo di argomento, esempio se "@100"

oppure "@@100";

- *setResolved(…)*: risolve il tipo e scrive il dato in una determinata risorsa;
- *getResolved(…)*: risolve il tipo e legge il dato da una determinata risorsa.

Prendendo per esempio la classe *IndDirA*:

```
public abstract class IndDirA implements InstructionAlphaI {

        protected int getTypeCode(){
                return TypeCodesI.IndDir;
        }

        protected String getTypeRegex(){
                return TypeCodesI.IndDir;
        }

        protected Word getResolved(ENIACI eniac, Word argument)
                throws OutOfBoundsException {
                        (…….)
        }

        protected void setResolved(ENIACI eniac, Word source,
                Word argument) throws OutOfBoundsException {
                        (…….)
        }

}
```

Figura 4.4.1

Generalmente nei due metodi *setResolved(…)* e *getResolved(…)*, quando si tratta di risolvere indirizzi diretti, indiretti, nonché registri, sono usate chiamate ai metodi della classe *Word*:

- *getAddressArgument()*: ottiene un dato di tipo *Address* dai 13 bit dal b0 al b12 dell'argomento della parola;
- *getWordArgument()*: ottiene un dato di tipo *Word* dai 13 bit dal b0 al b12 dell'argomento della parola (caso in cui avviene l'estensione del segno ai

rimanenti bit).

Un'osservazione la meritano anche i metodi *getTypeCode()* e *getTypeRegex()*: all'interno di questi sono presenti riferimenti a costanti, situate nell'interfaccia *TypeCodesI;* quest'ultima contiene tutte le possibile codifiche dei tipi di argomenti supportati da ENIAC, identificate dai bit dal b13 al b15 nonché le espressioni regolari che identificano il tipo di argomento, come mostrato nella figura 4.4.2:

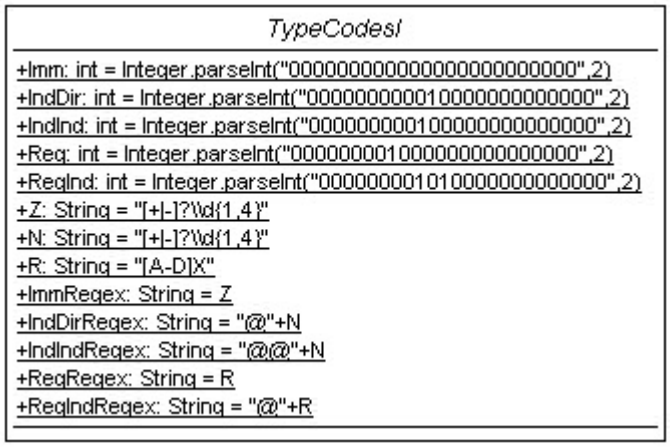


Figura 4.4.2

Il diagramma UML di figura 3.7.1 diventa a questo punto:

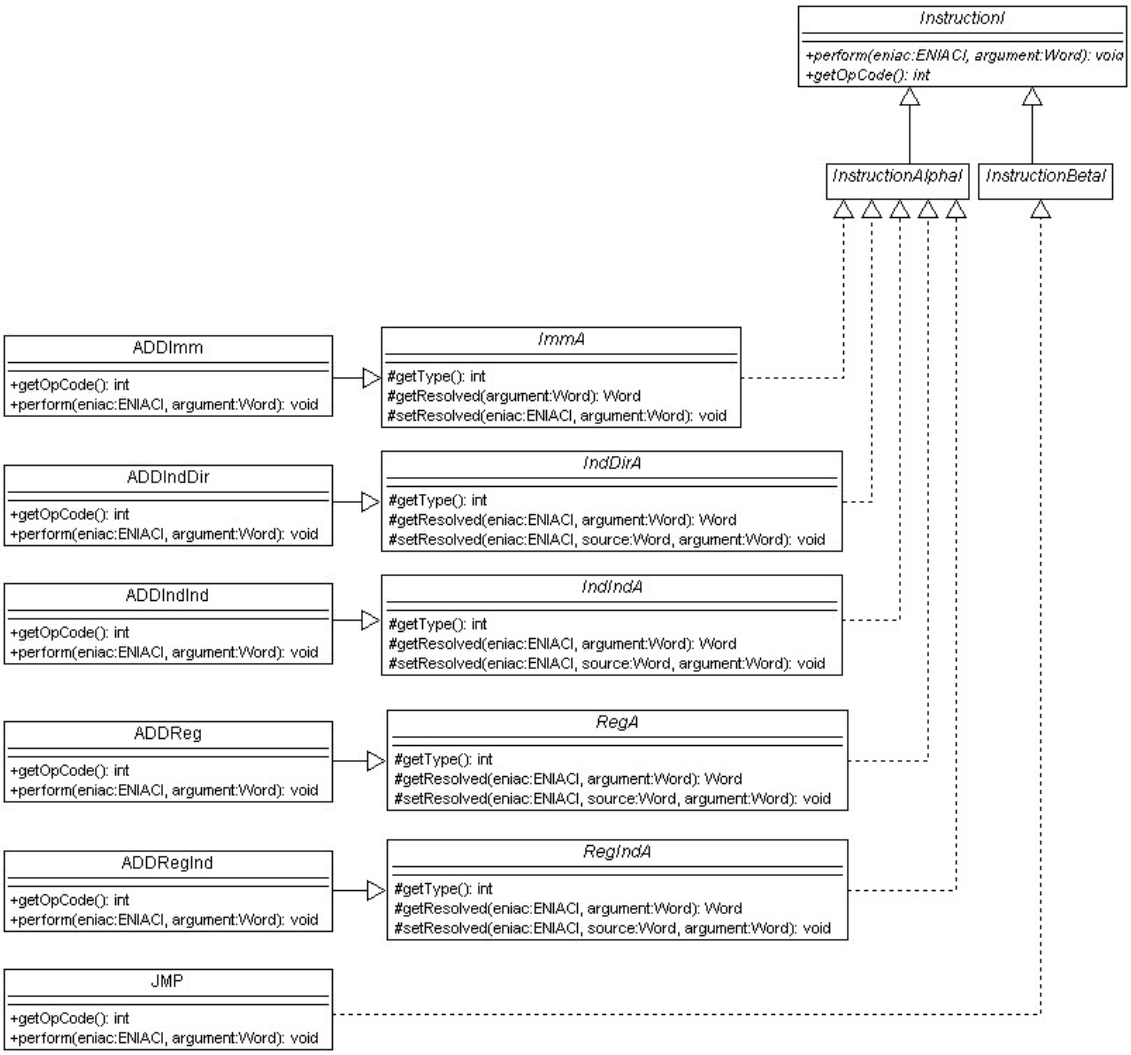


Figura 4.4.3

4.5 Observer

Dai diagrammi UML 3.3.1, 3.3.2, 3.4.1, 3.5.1 può essere osservato che le classi che rappresentano la memoria, le porte, i registri e i flags oltre implementare le interfacce, estendano una classe, la *Observable*.

Questa fa parte delle librerie standard di Java, precisamente il package *java.util* che, come più in avanti trattato, ha il compito di rendere le classi

che la estendono come dei "client" registrabili e monitorabili da classi server. Questo strumento, come si evincerà, risulta indispensabile per mantenere sincronizzate l'interfaccia grafica con lo stato delle *HashMap* che rappresentano parti dell'hardware del simulatore.
L'estensione della classe *Observable,* da parte di un'altra classe, mette a disposizione di quest'ultima dei metodi per notificare a una terza classe osservatrice che è avvenuto un cambiamento in quell'oggetto.
Prendendo ad esempio il metodo per l'inserimento dati in memoria:

```
public class DefaultMemory extends Observable implements  MemoryI {
        (…….)
        public void setCell(Address address, Word word) throws OutOfBoundsException {
                if(word.equals(new Word(0))){
                        (…….)
                }
                else {
                        _memory.put(address, word);
                        setChanged();
                        _event.setAddress(address);
                        _event.setWord(word);
                        notifyObservers(_event);
                }
        }
}
```

Figura 4.5.1

Il codice in figura 4.5.1, all'interno dell'*else*, mostra dei passaggi che in genere sono usati all'interno di tutte quelle risorse di ENIAC monitorate dall'interfaccia grafica:

1. *setChanged()*: avverte l'oggetto osservatore che un dato è stato aggiunto (o rimosso) dall'*HashMap* della memoria;
2. Viene preparato un oggetto contenente *l'Address* e la *Word* oggetto di modifica nella memoria;
3. *notifyObservers(ENIACEvent)*: avverte la classe osservatrice che lo stato

della memoria è cambiato inerentemente al dato passato.

Il dato passato al punto (3.), *ENIACEvent*, è un oggetto di tipo *Bean* creato ad hoc per il compito assegnatogli; constatato che in totale sono quattro le risorse monitorate (la memoria, le porte, i registri, i flags) e che sono fondamentalmente simili tra loro, si è optato per un oggetto dotato di due campi *private* con relativi metodi per scriverne e leggerne i dati; segue l'UML:

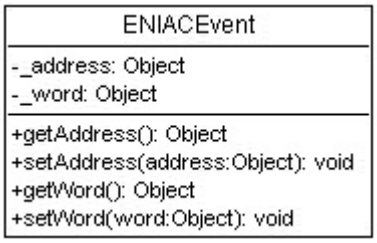


Figura 4.5.2

Alla luce di questi nuovi elementi, lo schema generale di ENIAC si completa nel seguente:

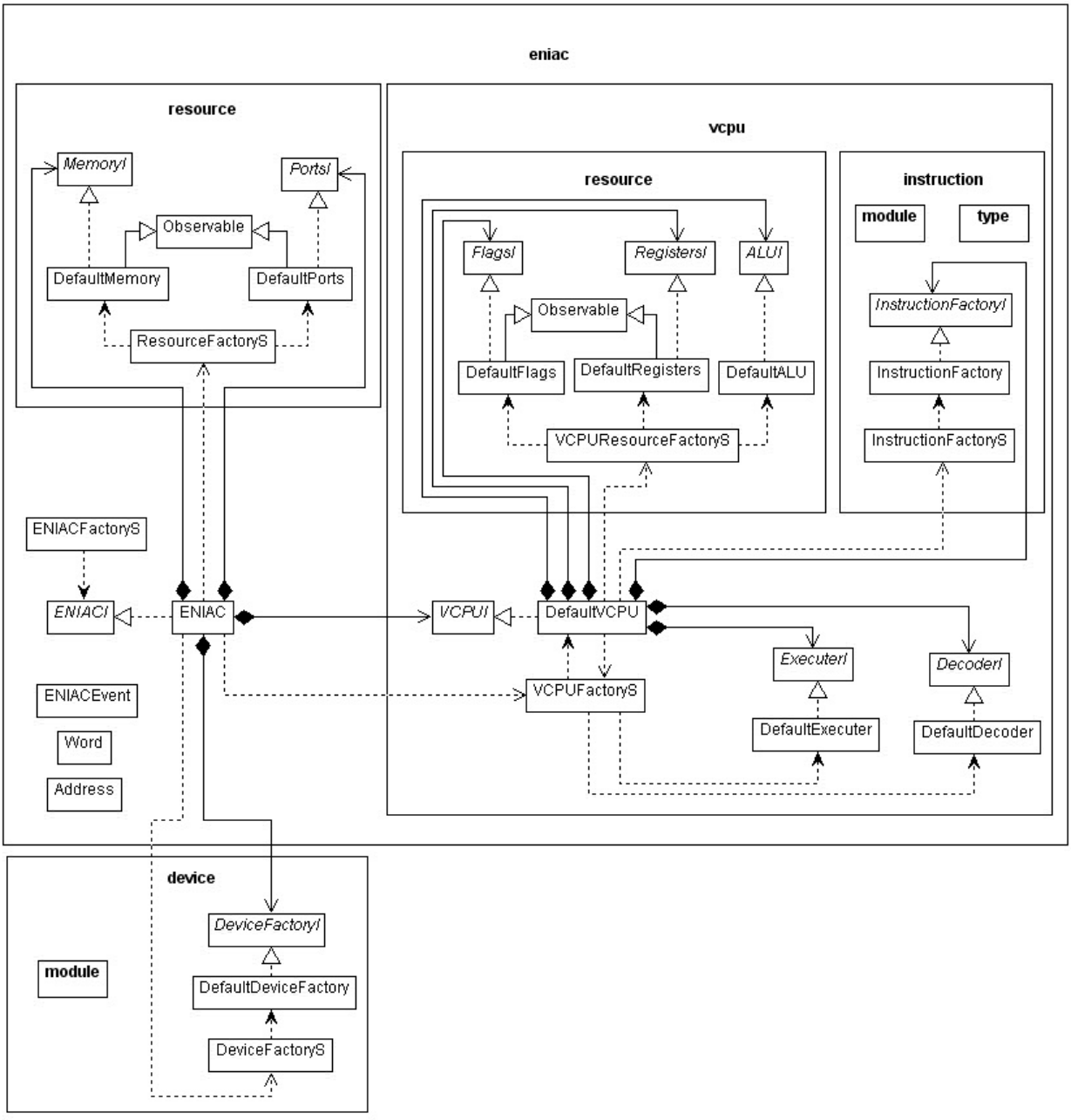


Figura 4.5.3

Quella di figura 4.5.3 è l'effettiva struttura del simulatore finale.

5 Istruzioni di ENIAC

Panoramica delle istruzioni supportate da ENIAC, e direttive per aggiungerne di nuove.

5.1 Istruzioni originali

Il set di istruzioni originale di vCPU[5] è stato ricreato in ENIAC, ovviamente rifacendosi alla specifica dei capitoli 3 e 4 quanto a progettazione. Seguono i diagrammi UML di tutte le istruzioni implementate in ENIAC ordinate in base al tipo di argomento trattato dall'istruzione (e quindi in base alla superclasse astratta descritta nel paragrafo 4.4).

Istruzioni che riconoscono il tipo di argomento Immediato:

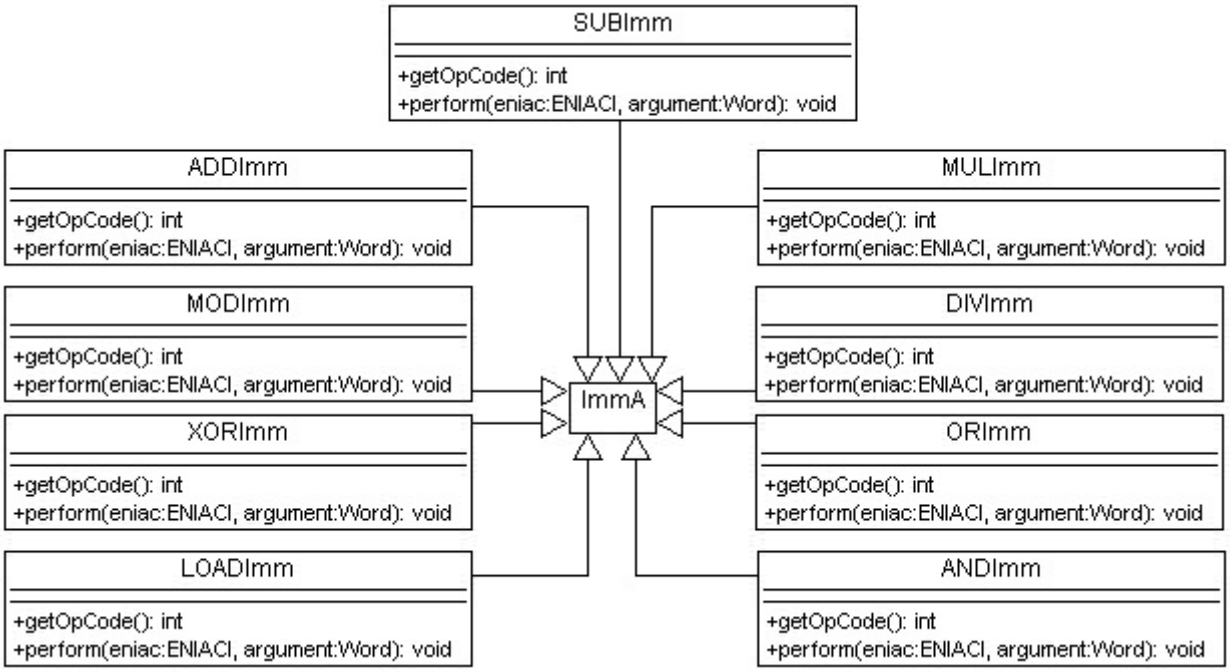


Figura 5.1.1

[5] Rif. pag. 66,67,68,69, tesi vCPU.

Istruzioni che riconoscono il tipo di argomento Indirizzo Diretto (oppure "@"):

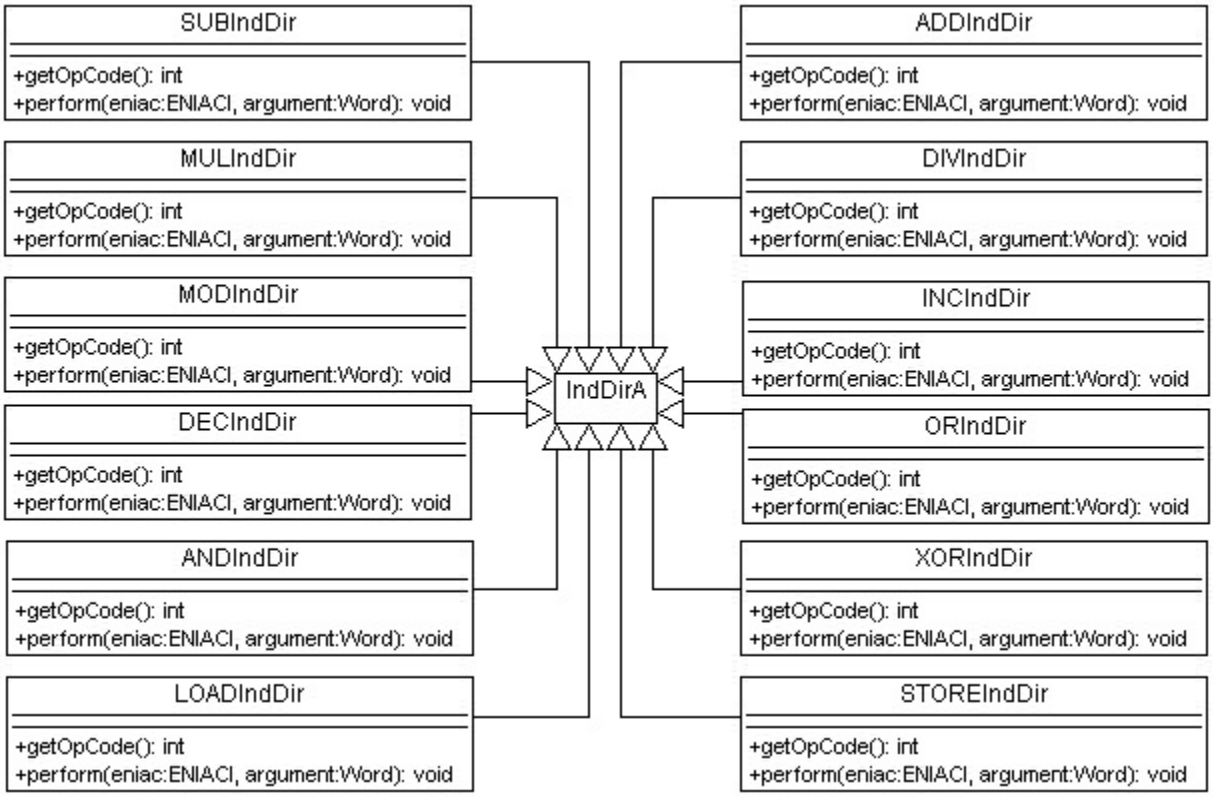


Figura 5.1.2

Istruzioni che riconoscono il tipo di argomento Indirizzo Indiretto (oppure "@@"):

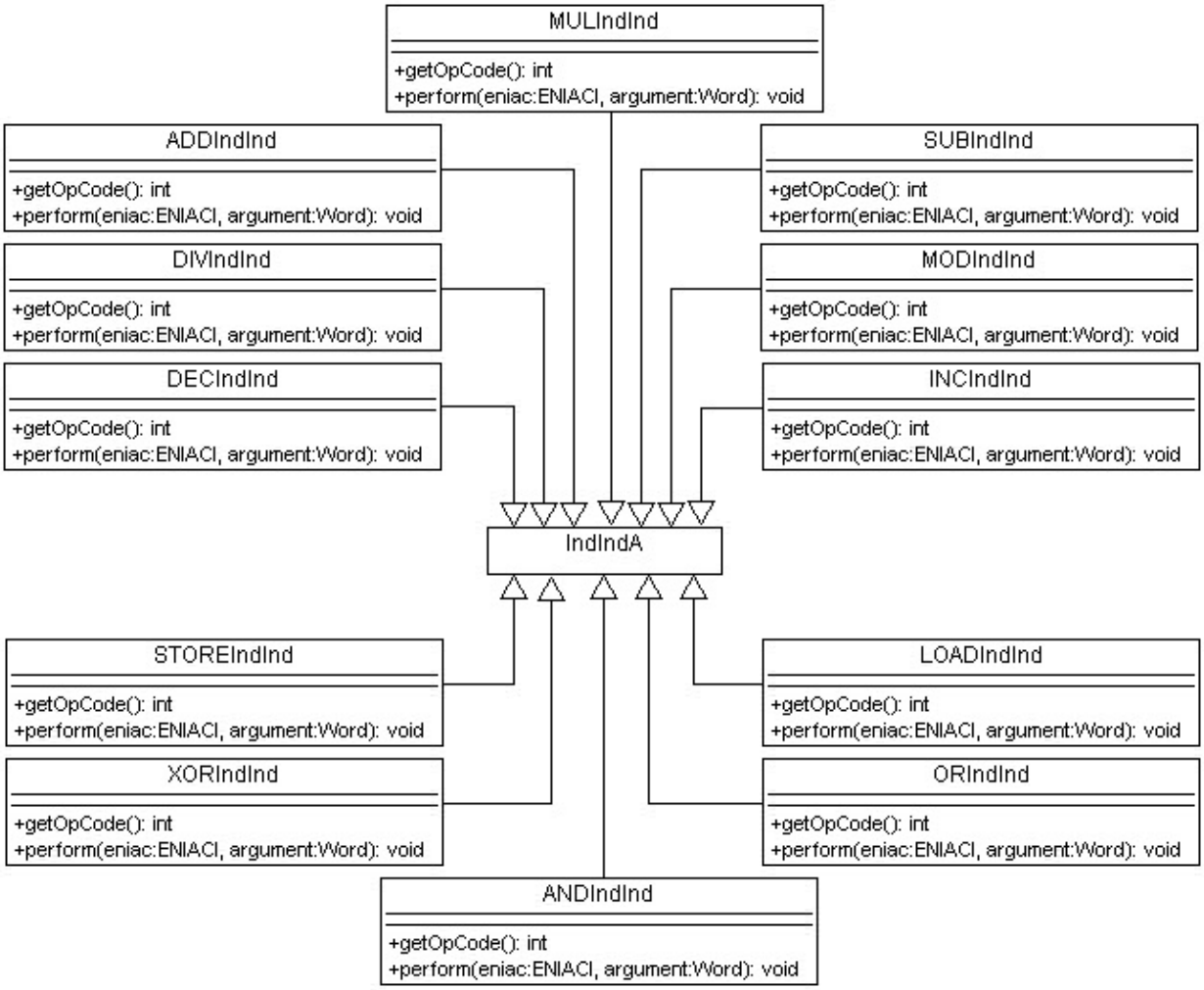


Figura 5.1.3

Istruzioni che riconoscono il tipo di argomento Registro (oppure "R"):

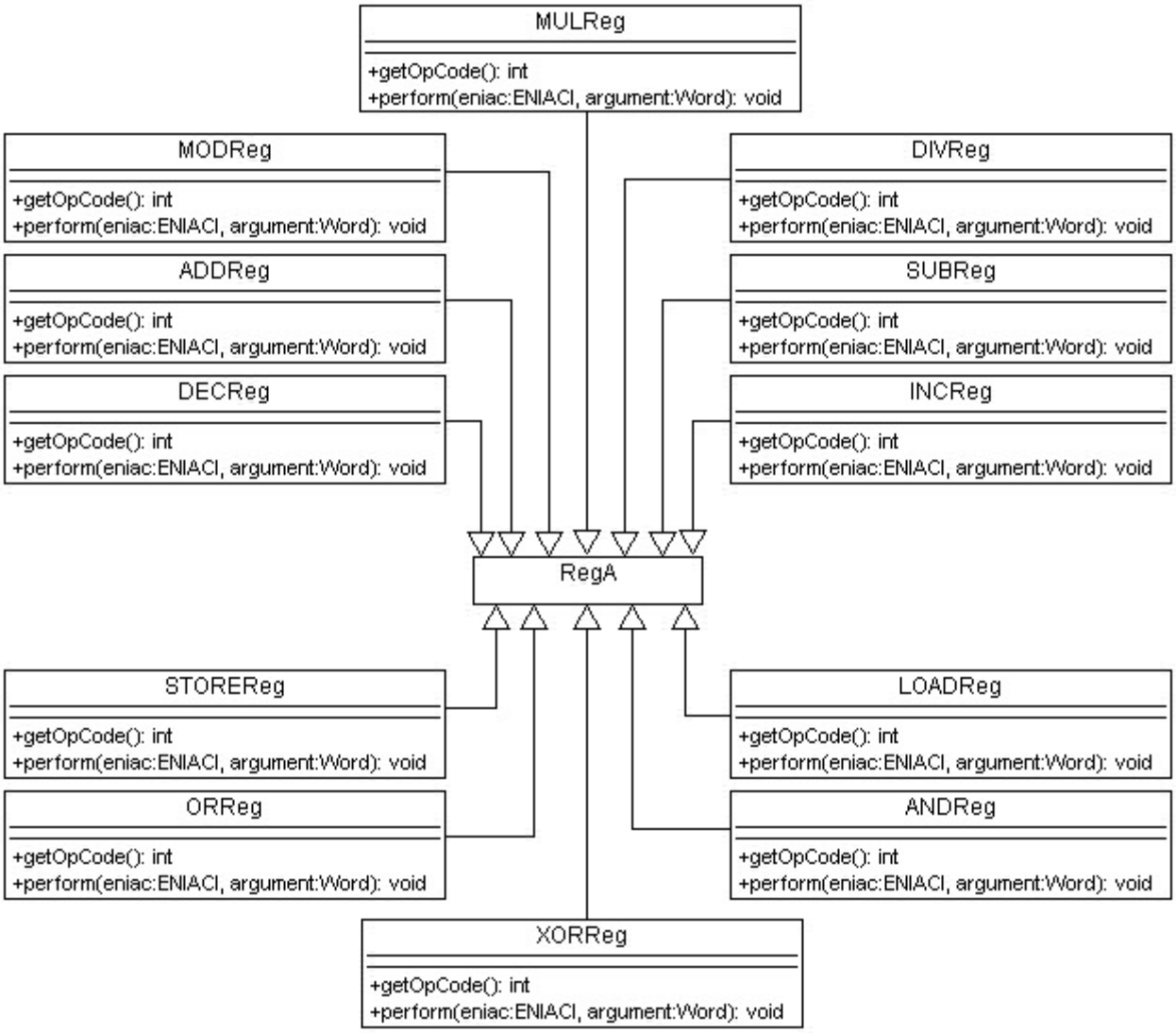


Figura 5.1.4

Istruzioni che riconoscono il tipo di argomento Registro Indiretto (oppure "@R"):

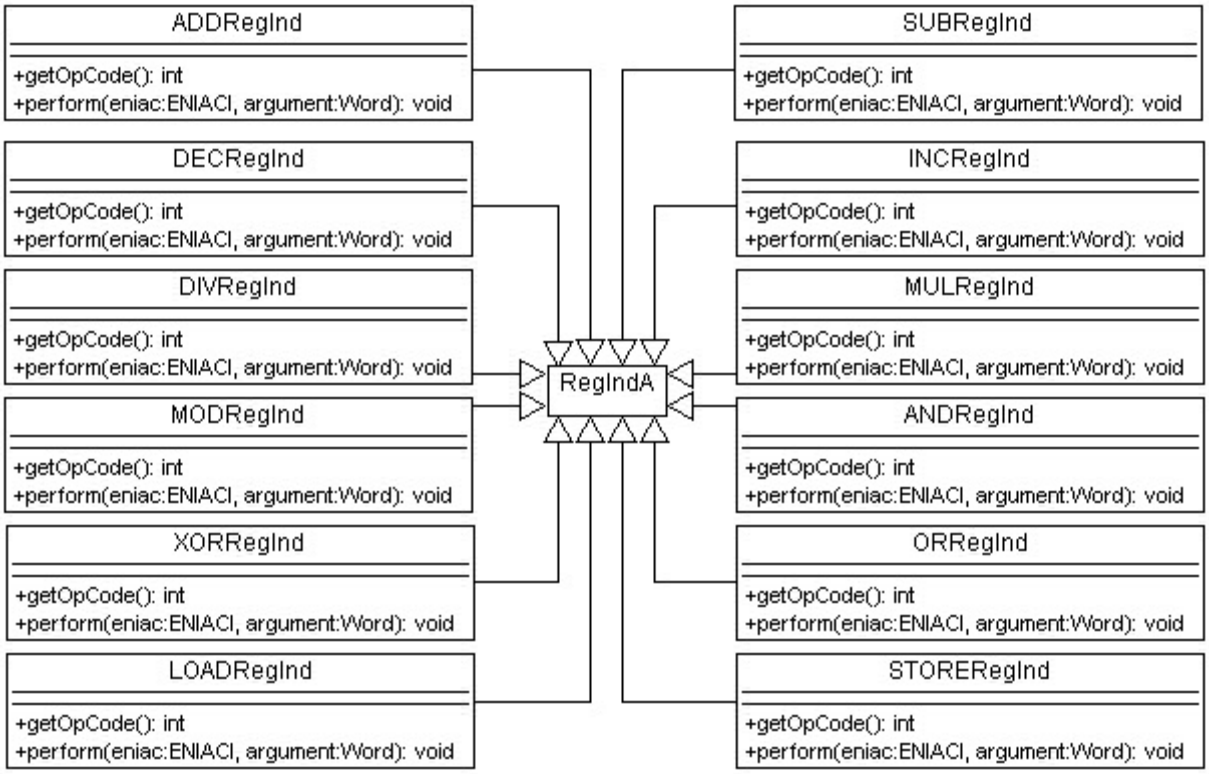


Figura 5.1.5

5.2 Nuove istruzioni

Il programmatore può cimentarsi nella scrittura di nuove istruzioni in ENIAC, attenendosi scrupolosamente a determinate regole.

All'interno del package *eniac.vcpu.instruction* è presente un'interfaccia denominata *OpCodesI;* all'interno di essa sono dichiarate esclusivamente variabili, ognuna delle quali porta il nome di una istruzione di tipo assembly supportata da vCPU, il suo valore è la codifica generica di ogni istruzione e cioè i bit dal b23 al b16:

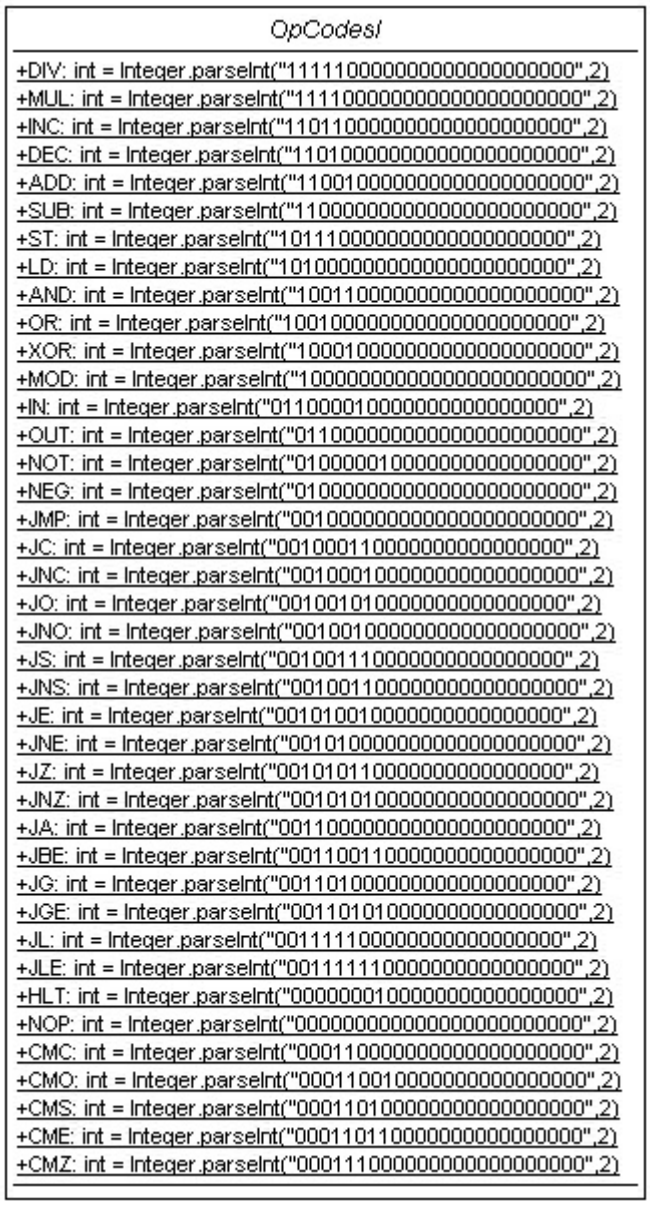


Figura 5.2.1

Una volta identificato l'ambito dell'istruzione, di tipo alpha o beta, la successione dei passi per aggiungere correttamente un'istruzione è:

1. aggiungere il codice operativo nell'interfaccia *OpCodesI* (figura 5.2.1) facendo attenzione che non si tratti di un codice operativo già esistente;
2. porre attenzione al nome dell'istruzione (es. istruzioni identiche ma con tipo di argomento accettabile diverso), creare la classe dell'istruzione nel package *eniac.vcpu.instruction.modules*;
3. in base al tipo di istruzione deciso, estendere una delle classi astratte della tabella 4-3 oppure implementare una delle interfacce *InstructionAlphaI* o *InstructionBetaI*;
4. implementare quindi i tre metodi di figura 3.7.1, facendo attenzione che:
   - il metodo *getOpCode()* prelevi il codice operativo dall'interfaccia *OpCodesI* dove sicuramente sarà corretto per via del punto (1.) e nel metodo *getOpRegex()* sia presente una stringa che rappresenti una espressione regolare valida che riconosca il nome dell'istruzione;
   - inoltre se estesa una delle classi astratte della tabella 4-3 prelevarne, tramite i metodi esposti al punto precedente, il codice operativo dell'argomento e anche la stringa dell'espressione regolare della superclasse tramite i metodi nel paragrafo 4.4;
   - scrivendo il *perform(…)* venga usato l'hardware attraverso il Locator *ENIACI* in argomento, poiché alle istruzioni è permesso l'accesso all'hardware del simulatore.

Segue un esempio di classe che implementa l'istruzione assembly "ADD", nel caso di argomenti Immediati:

```
public class ADDImm extends ImmA {
        public int getOpCode() {
                return (OpCodesI.ADD | super.getTypeCode());
        }

        public int getOpCode() {
                return ("ADD[]+"+super.getTypeRegex());
        }

        public void perform(ENIACI eniac, Word argument)
                throws OutOfBoundsException, InexistentRegisterException,
                EndOfProgramException, DivisionByZeroException {
                try {
                (……)
                } catch (OutOfBoundsException e) {
                        throw new OutOfBoundsException(…);
                }
        }
}
```

Figura 5.2.2

Come si evince, i messaggi di errore generati dalle eccezioni nel metodo *perform(…)* delle classi delle istruzioni, acquisiscono una certa importanza per il programmatore che si cimenterà nello scrivere programmi in linguaggio assembly di vCPU; occorre quindi particolare accortezza nello scrivere dei messaggi di errore eloquenti.

6 Interfaccia utente

Dettagli grafici e implementativi dell'interfaccia utente.

6.1 Descrizione GUI

Fino a questo momento l'oggetto di discussione è stato il simulatore, o meglio la *business logic* del progetto.
GUI è l'acronimo di Graphics User Interface, ovvero il *front end* tramite cui poter utilizzare il simulatore ENIAC (figura 6.1.1).
L'obiettivo principale è mettere a disposizione del programmatore che vuole cimentarsi nella scrittura di programmi in linguaggio assembly di vCPU, un'interfaccia grafica semplice e funzionale, che permetta di gestire agevolmente tutte quelle risorse coinvolte nell'esecuzione di un programma, quali memoria, porte, registri, flags, ma non solo: la GUI di ENIAC è stata pensata mirandola, in modo particolare, a scopo didattico.
E' stata quindi aggiunta la possibilità di modificare le risorse di cui sopra, durante l'esecuzione di un programma, allo scopo di alterarne il flusso d'esecuzione; ad esempio, salti condizionati e incondizionati dipendono dai flags: "JP N" è un'istruzione di salto dipendente dal flag EV, se quest'ultimo vale 0 ma per un imprecisato motivo si desidera che salti comunque alla cella N specificata, sarà sufficiente agire manualmente nella casella dei flag, modificare il campo a 1 alla voce EV per cambiare il flusso del programma e saltare alla suddetta cella.
Inoltre sempre per favorire il lato istruttivo sono stati aggiunti controlli sul flusso di esecuzione dei programma caricato, per permettere l'esecuzione di pari passo di ogni singola riga dei programmi scritti in linguaggio assembly di vCPU.
Seguono dettagli grafici e implementativi per giungere a questi risultati.

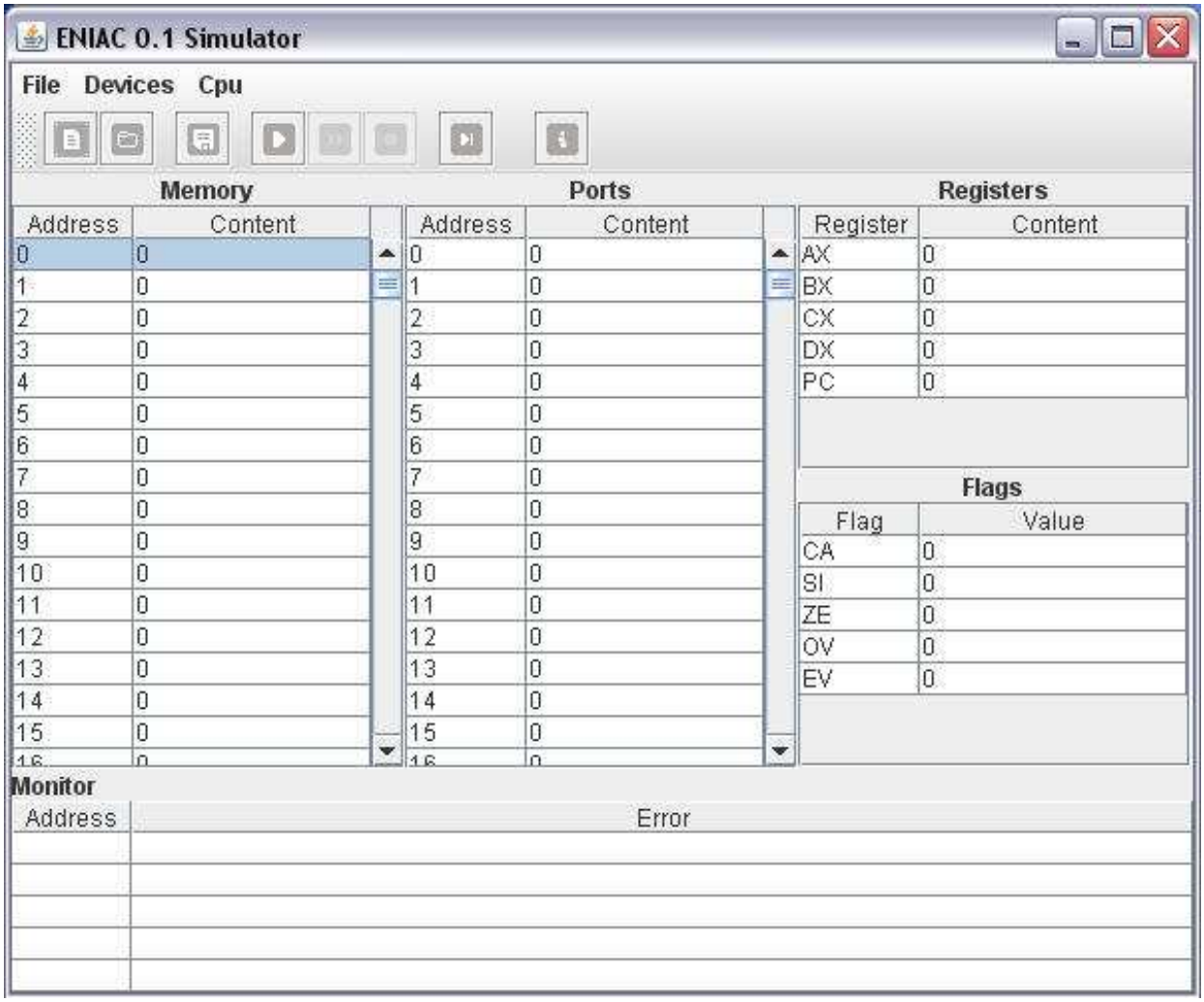


Figura 6.1.1

6.2 Inizializzazione

L'interfaccia grafica, nel simulatore ENIAC, contiene il metodo *main(…)* ossia il metodo da cui viene inizializzato il simulatore; due sono le fasi principali:

- Inizializzazione hardware: consiste nell'uso della *ENIACFactoryS* (figura 4.5.3) da cui ottenere una istanza di tipo *ENIACI* ovvero il Locator che permette di accedere a tutto l'hardware;
- Inizializzazione della GUI: è un oggetto *ENIACFrame*, di tipo *JFrame*, che viene inizializzato attorno all'oggetto *ENIACI* del punto precedente. Attraverso l'istanza del Locator infatti, verranno localizzate e inizializzate

tutte le risorse, comprese quelle che potranno poi essere controllate tramite la GUI; per motivi al paragrafo 4.1 ovviamente gli oggetti ottenuti attraverso il Locator saranno mappati su oggetti delle rispettive interfacce:

- *MemoryI*;
- *PortsI*;
- *RegistersI*;
- *FlagsI*;
- *ExecuterI*;

Di queste, le prime quattro rappresentano le risorse che potranno essere visualizzate e controllate attraverso la GUI, mentre l'oggetto di tipo *ExecuterI* permetterà di gestire il flusso di esecuzione dei programmi scritti in linguaggio assembly di vCPU tramite l'interfaccia grafica. Le restanti risorse di ENIAC non sono localizzate né inizializzate, in quanto non utili ai fini grafici (quindi sono inizializzate altrove).
Inoltre la classe *ENIACFrame* consta di alcuni metodi di gestione, atti ad automatizzare varie operazioni.

6.3 Componenti grafici

- package *gui*

Il *front end* di ENIAC (*ENIACFrame*) utilizza ampiamente le librerie *swing* e *awt* di Java; la finestra principale è un *JFrame* (in quanto estende quest'ultima):

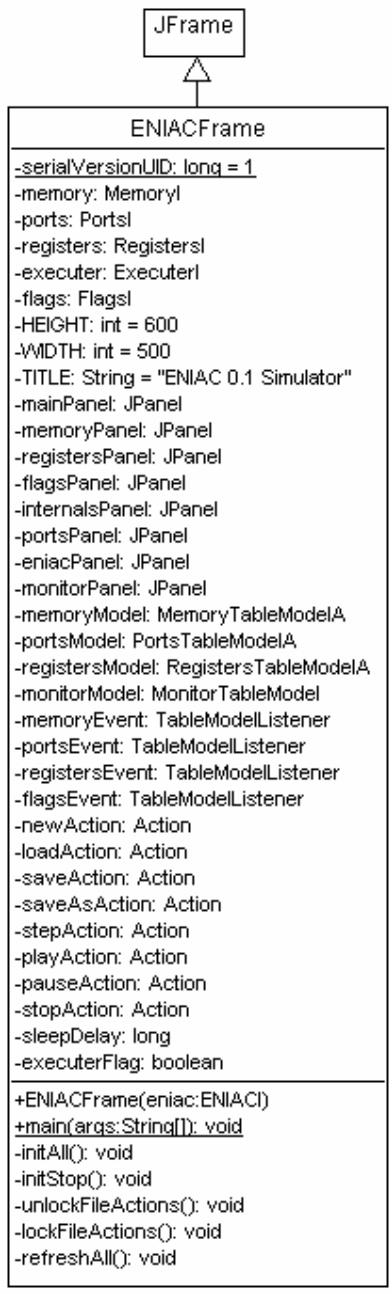


Figura 6.3.1

Su di esso, attraverso la composizione, sono stati poi applicati diversi oggetti di tipo *JPanel*:

- *memoryPanel()*;
- *portsPanel()*;
- *registersPanel()*;
- *flagsPanel()*.

Per ottenerne un preciso posizionamento si è anche reso necessario l'uso di layout del tipo:

- *BorderLayout()*;
- *GridLayout(…)*.

Tramite questi ultimi si sono potuti disporre gli oggetti *JPanel* come rappresentati nella figura 6.3.2:

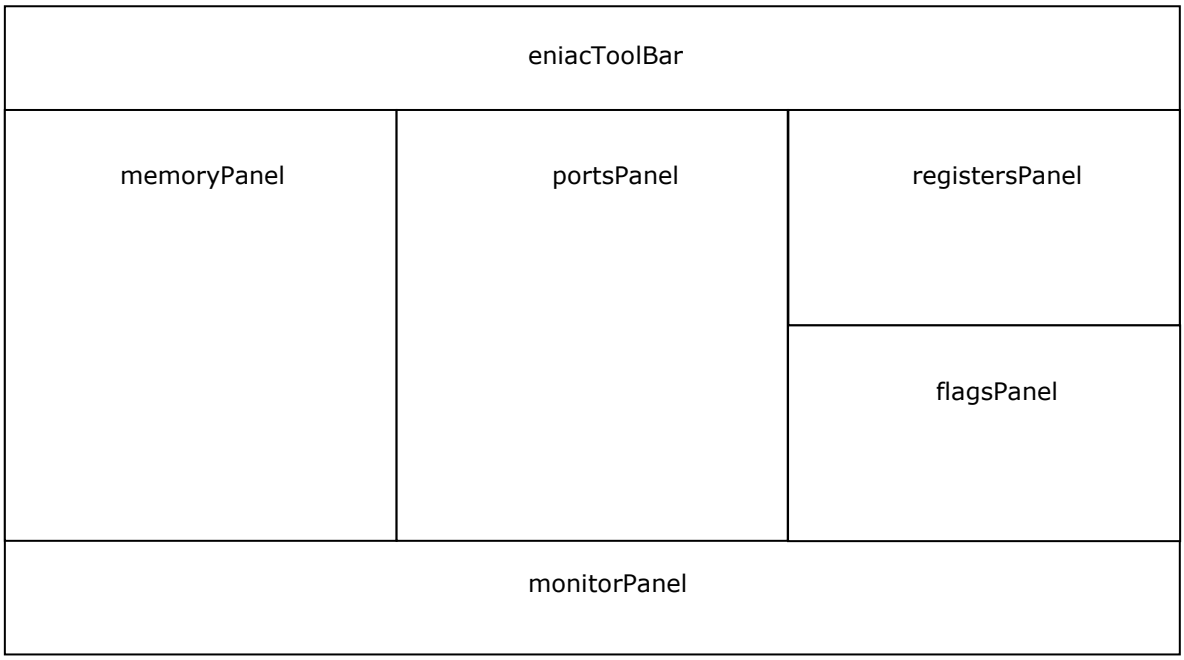


Figura 6.3.2

Partendo dall'alto (*eniacToolBar()*) e cioè dai menu, c'è stata la ricerca di un approccio utente presente nei software di uso comune, quindi un menù testuale completo nella parte più alta, seguito successivamente da una barra degli strumenti. Java fornisce, a disposizione del programmatore, una serie

di strumenti precompilati:

- *JMenuBar()*: una barra dei menu da allegare al *JFrame*; sono costituiti da *JMenu()* ovvero menu da comporre. In ENIAC ne sono presenti tre;
- *JToolBar()*: una barra contenente dei bottoni che permettono l'accesso rapido ad alcune funzioni altrimenti raggiungibili solo con le voci nei *JMenu().*

Seguono ora immagini raffiguranti il risultato dell'uso di questi oggetti.

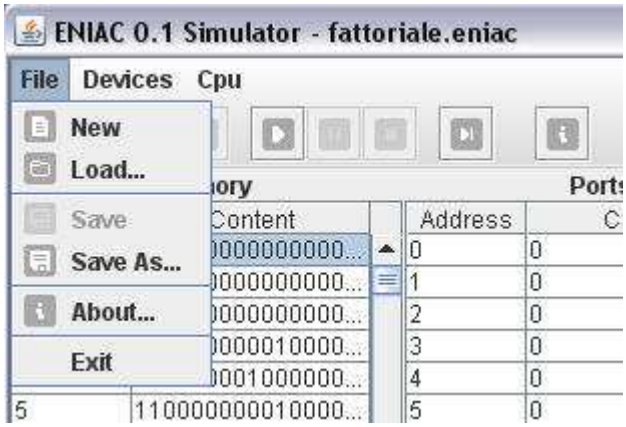


Figura 6.3.3

Menù file: operazioni tramite cui poter aprire, creare, salvare un file in linguaggio assembly di vCPU o anche uscire dal programma.

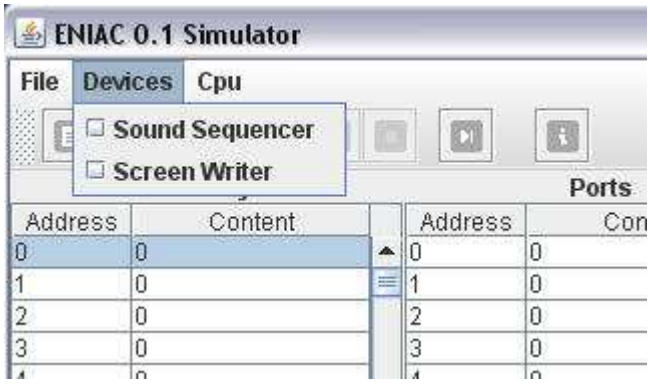


Figura 6.3.4

Menù Devices: attraverso questo è possibile attivare o disattivare dispositivi da connettere alle porte.

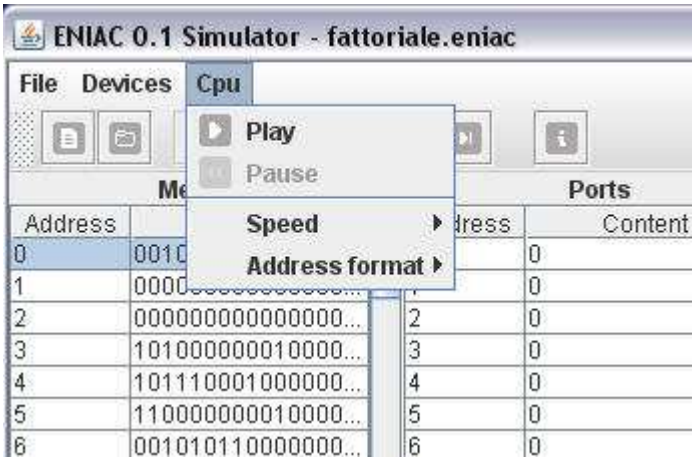


Figura 6.3.5

Menù Cpu: controlli vari per il flusso di esecuzione come Step, Play, Pause, Stop, ma anche alcuni sottomenù.

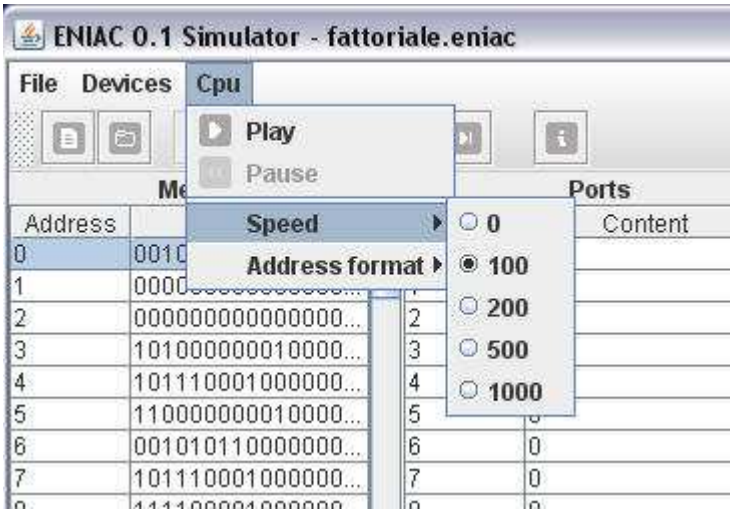


Figura 6.3.6

Sottomenù Speed: tramite esso è possibile controllare la velocità di esecuzione dei programmi scritti in linguaggio assembly di vCPU. I numeri esprimono il tempo di pausa in millisecondi tra l'esecuzione di una istruzione e la successiva.

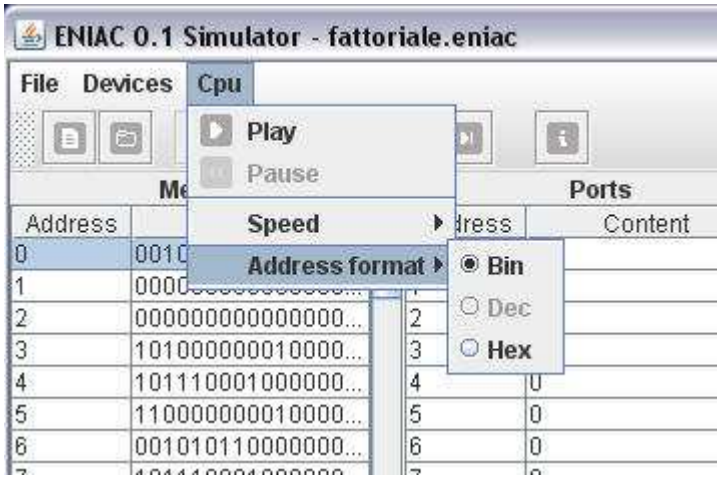


Figura 6.3.7

Sottomenù Address Format: permette di cambiare le modalità di visualizzazione dei dati presenti all'interno delle risorse di ENIAC. Ad esempio se Bin, il contenuto della memoria, delle porte, e dei registri verrà visualizzato con numeri in base 2; ad eccezione dei flag, che in qualsiasi modalità saranno sempre visualizzati con 1 oppure 0.
Sempre dalle precedenti immagini può essere notato che, sottostante la barra dei menù, sono presenti i bottoni di accesso diretto, costruiti tramite la *JToolBar*.

Tutti i comandi discussi sinora, sia dei menù che della toolbar, da un punto di vista implementativo generano degli eventi, che devono essere catturati per permettere l'esecuzione del codice adeguato. Diverse erano le possibilità, ma la scelta che permetteva il miglior riuso di codice è ricaduta sull'oggetto *AbstractAction()* del package *swing*; in quanto questo tipo di classi permettono non solo di gestire l'evento generato da un componente grafico, ma anche di impostarne dati come lo stato di selezione di default, etichette, icone e altre che possono essere colte graficamente sia nei bottoni della *JToolBar* che nelle voci dei menù.
Alla luce di quanto detto, è stato sufficiente creare *classi interne* alla classe *ENIACFrame,* che però ereditassero l'oggetto *AbstractAction()* così da poter

customizzare sia lo stato grafico che l'azione associata all'evento generato. Nell'UML di figura 6.3.1 sono presenti le firme di alcuni oggetti di tipo *Action* (in quanto l'oggetto *AbstractAction()* implementa l'interfaccia *Action*), di cui ne viene fornita una lista e relative descrizioni:

Classe interna	Descrizione
NewAction	Inizializza un nuovo foglio di lavoro
LoadAction	Carica un file sorgente esterno
SaveAction	Salva nel file precedentemente aperto
SaveAsAction	Salva in un nuovo file
AboutAction	Mostra i credits del programma
ExitAction	Esce dal programma
StepAction	Esegue un ciclo di fetch – decode – execute
PlayAction	Esegue più cicli, fino al raggiungimento dell'HLT
PauseAction	Ferma temporaneamente l'esecuzione
StopAction	Ferma e reinizializza l'esecuzione
Sleep0Action	Pausa di 0 secondi tra un ciclo e il successivo
Sleep100Action	Pausa di 100 secondi tra un ciclo e il successivo
Sleep200Action	Pausa di 200 secondi tra un ciclo e il successivo
Sleep500Action	Pausa di 500 secondi tra un ciclo e il successivo
Sleep1000Action	Pausa di 1000 secondi tra un ciclo e il successivo
BinFormatAction	Modalità di visualizzazione binaria
SynFormatAction	Modalità di visualizzazione con sintassi
DecFormatAction	Modalità di visualizzazione in decimale
HexFormatAction	Modalità di visualizzazione esadecimale
DeviceAction	Attiva o disattiva un dispositivo assegnatogli

Tabella 6-1

6.4 Classi interne come azioni

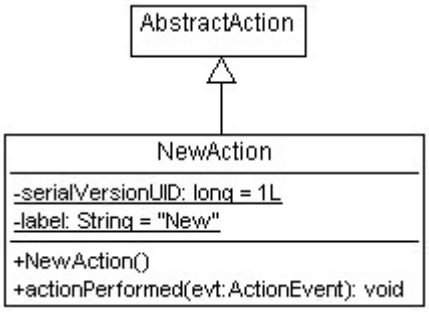


Figura 6.4.1

La figura 6.4.1 mostra lo schema di una classe interna, la *NewAction*; in questa è dichiarato il costruttore base e un metodo:

- *NewAction()*: costruttore, ha il compito di specificare impostazioni come etichetta e icona da visualizzare sullo schermo;
- *actionPerformed(…)*: questo è un metodo che viene eseguito automaticamente dal gestore degli eventi, in esso sono specificati i comandi dell'azione richiesta. Ad esempio, nel caso della *NewAction()* verranno inizializzate tutte le risorse hardware di ENIAC, comprese le loro rappresentazioni a schermo.

Tutte le classi interne seguono il medesimo schema, ma ci sono alcuni casi che meritano menzioni particolari: su tutte la *PlayAction,* le classi interne del tipo *xxxFormatAction* e la *DeviceAction*.

PlayAction è una classe interna il cui scopo è permettere l'esecuzione di un intero programma caricato nella memoria di ENIAC, scritto in linguaggio assembly di vCPU. Fondamentalmente esegue il metodo *step(),* descritto nel paragrafo 3.10*,* di un oggetto di tipo *ExecuterI* (), fino al raggiungimento dell'istruzione di halt (oppure "HLT"); inoltre per introdurre elementi utili ai fini didattici, si è pensato di aggiungere la possibilità di sospendere momentaneamente l'esecuzione dei programmi scritti in linguaggio assembly di vCPU, se non addirittura interromperla definitivamente.

Dopo alcune considerazioni sulle varie possibilità concesse da Java (ad esempio la classe *JTimer*), gli obiettivi sono stati raggiunti attraverso la creazione di un nuovo Thread all'interno del processo del simulatore (soluzione considerata semplice ed elegante). In pratica, quando ad ogni pressione della voce Play nell'interfaccia grafica viene eseguito il metodo *actionPerformed(…)* della classe *PlayAction,* viene creato ed eseguito il nuovo Thread, identico a quello principale, che quindi affiancherà quest'ultimo; per questo motivo la classe interna *PlayAction* oltre a mantenere la struttura indicata nel diagramma di figura 6.4.1, implementa anche l'interfaccia *Runnable*:

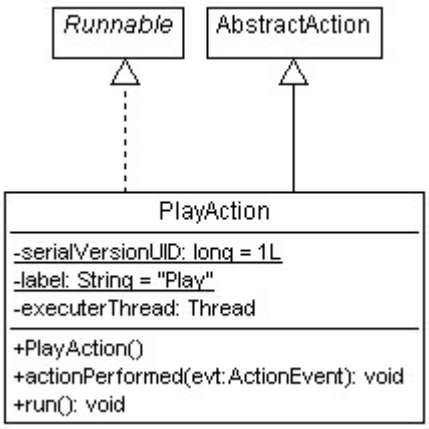


Figura 6.4.2

Dall'interfaccia viene ereditato il metodo *run(),* nel cui corpo è stato inserito un ciclo *while(…)* controllato da due variabili d'istanza; è da quest'ultimo che sono eseguite le chiamate al metodo *step()* precedentemente accennato:

- *executerFlag*: tipo *boolean*, viene letta ad ogni iterazione del ciclo *while(…)*; se *false* viene meno il ciclo e conseguente terminazione del Thread;
- *sleepDelay*: un intero che specifica il numero di millisecondi per cui il Thread deve essere sospeso; effetto ottenuto da una chiamata *Thread.sleep(…)* situata all'interno del ciclo *while(…)*, che pone in attesa tutte le operazioni in atto in quel Thread, compresa proprio

quell'iterazione del ciclo (tale variabile viene impostata dalle *SleepxxxxAction*).

Alla luce di quanto detto potrebbe sorgere una domanda: perché non usare la chiamata *Thread.sleep(…)* nel Thread principale piuttosto che in uno secondario? Analizzando il comportamento dei Thread è facile intuire che questa, non si sarebbe limitata a porre in attesa solo quell'iterazione del ciclo *while()* ma addirittura qualsiasi altra operazione all'interno dello stesso Thread, come i comandi di pausa e stop.
Il motivo per cui è necessario creare un nuovo Thread ad ogni pressione del comando Play (e conseguente *PlayAction*) dell'interfaccia grafica risiede in una limitazione imposta da Java, che non permette successivamente di riavviare lo stesso.
Per ciò che riguarda le classi interne di tipo *xxxFormatAction,* queste contengono il codice necessario al cambio della modalità di visualizzazione dei dati. Vista la vastità dell'argomento si ritiene utile, per maggior chiarezza, dedicarne l'argomentazione nel paragrafo successivo.
Anche le *DeviceAction* meritano una trattazione a parte.

6.5 Gestione modalità di visualizzazione

- package *gui.table*

Nel layout di figura 6.3.2 sono presenti cinque oggetti (i pannelli di tipo *JPanel*), sui quali sono applicate delle tabelle atte a visualizzare il contenuto delle memorie, delle porte, dei registri, dei flags nonché il monitor degli errori.
Di queste vengono analizzate le prime quattro, in quanto devono presentare una certa "interattività" con l'utente che si cimenta nella scrittura di programmi in linguaggio assembly di vCPU, interattività che risiede in una gestione dei flussi di dati "da e verso" le risorse del simulatore; infatti, come già portato in evidenza, grazie all'architettura ideata il simulatore e la GUI

sono nettamente separate. In pratica il simulatore costituisce la *business logic,* che quindi come da regola di buona programmazione è separato dal codice del *front end*.

I flussi dei dati da analizzare fondamentalmente sono due:

- dati inseriti mediante caricamento file o manualmente attraverso le tabelle delle risorse del simulatore;
- dati che a causa delle operazioni eseguite dal simulatore sono cambiati; detto cambiamento deve essere visualizzato nelle tabelle.

Fondamentalmente come prima attività sono stati creati dei modelli delle tabelle: Java allo scopo fornisce le classi *AbstractTableModel*, che una volta estese permettono di ridefinire alcuni metodi che riguardano il comportamento delle tabelle, come la possibilità di editare i valori delle celle ed elaborare i dati inseriti. Personalizzare gli *AbstractTableModel* conferisce la flessibilità necessaria agli oggetti *EniacTable* (in seguito trattati*)* per la visualizzazione dei dati nel *front end* di ENIAC, infatti le tabelle vengono create attorno a questi modelli.

Segue un esempio di modello per la tabella della memoria, basato su una classe astratta (il motivo di tale scelta sarà indicato a breve):

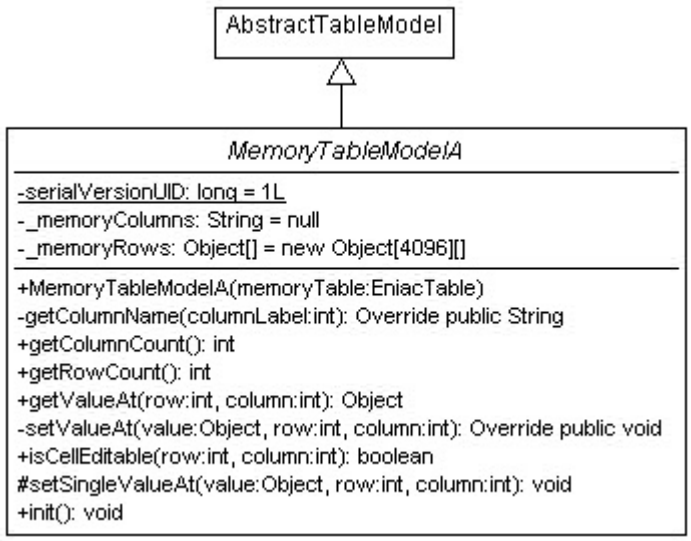


Figura 6.5.1

Dopo queste premesse si potrà passare all'esame dei due punti di cui sopra, partendo dal secondo.

Come si potrà rammentare dal paragrafo 4.5, quattro classi rappresentanti risorse di ENIAC estendono la classe *Observable*, attraverso metodi della quale ad ogni inserimento o cambiamento di dato, vengono inoltrati degli oggetti *ENIACEvent* ad una non precisata classe server; quest'ultima può adesso essere svelata e va ricercata nei modelli delle tabelle usate nel *front end* di ENIAC.

Infatti sono quest'ultime le classi *Observer*, così definite perché oltre ad estendere la classe *AbstractTableModel* implementano anche l'interfaccia *Observer*, che consta di un unico metodo:

- *void update(Observable, Object)*: dove il primo indica la classe osservata che ha generato l'evento, mentre il secondo non è altro che l'oggetto *ENIACEvent* del paragrafo 4.5, contenente il dato che deve essere rappresentato in una determinata cella della tabella.

```
public abstract class MemoryTableModelA
        extends AbstractTableModel
        implements Observer {
        (……)
}
```

Figura 6.5.2

Dopo un'attenta analisi si è arrivati alla conclusione che le operazioni di conversione del tipo di dato da visualizzare (indirizzo e parola a 24 bit nel dato *ENIACEvent*) nel formato scelto (binario, decimale, esadecimale o sintassi), potevano essere subordinate solo al metodo *update(…)* piuttosto che a qualche altro metodo del modello della tabella; a causa di questa scelta si è optato per rendere classi astratte i modelli delle tabelle che venivano quindi estese da altre classi dotate del metodo *update(…)*. Segue un esempio per la classe che si occuperà di visualizzare i dati della memoria

in formato binario nella relativa tabella:

```
public class BinMemoryModel extends MemoryTableModelA {
        (……)

        public void update(Observable o, Object arg) {
                (……)
        }
}
```

Figura 6.5.3

A dimostrazione di quanto detto si mostra il diagramma UML delle classi che entrano in gioco nel modello della tabella della memoria, da considerare come un ulteriore livello verso il basso di quello di figura 6.5.1:

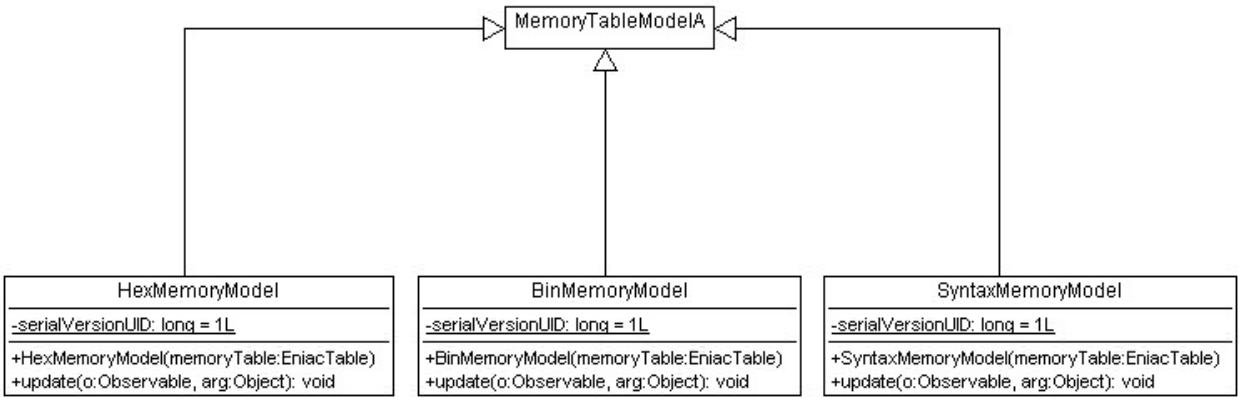


Figura 6.5.4

Ricapitolando, quanto detto sinora ovviamente viene ricalcato per le tabelle che mostrano i dati delle seguenti risorse:

- Memoria;
- Porte;
- Registri;
- Flags.

I motivi che hanno portato a queste scelte, sono principalmente due:

- come mostrato dal diagramma di figura 6.5.4 c'è un cospicuo riuso del codice dei modelli delle tabelle; è stato necessario riscrivere solo il codice delle classi che specializzassero il dato da visualizzare;
- si è venuta a creare una associazione biunivoca tra *l'Observer* (il modello della tabella) e la relativa risorsa *Observable* (le risorse del simulatore ENIAC); si sarebbe potuto creare una sola classe *Observer*, ad esempio la classe principale *ENIACFrame*, ma questo avrebbe costretto la scrittura di codice di riconoscimento della classe *Observable* generatrice dell'evento, codice che sicuramente avrebbe aumentato la complessità temporale.

In definitiva si fornisce un ulteriore schema generale:

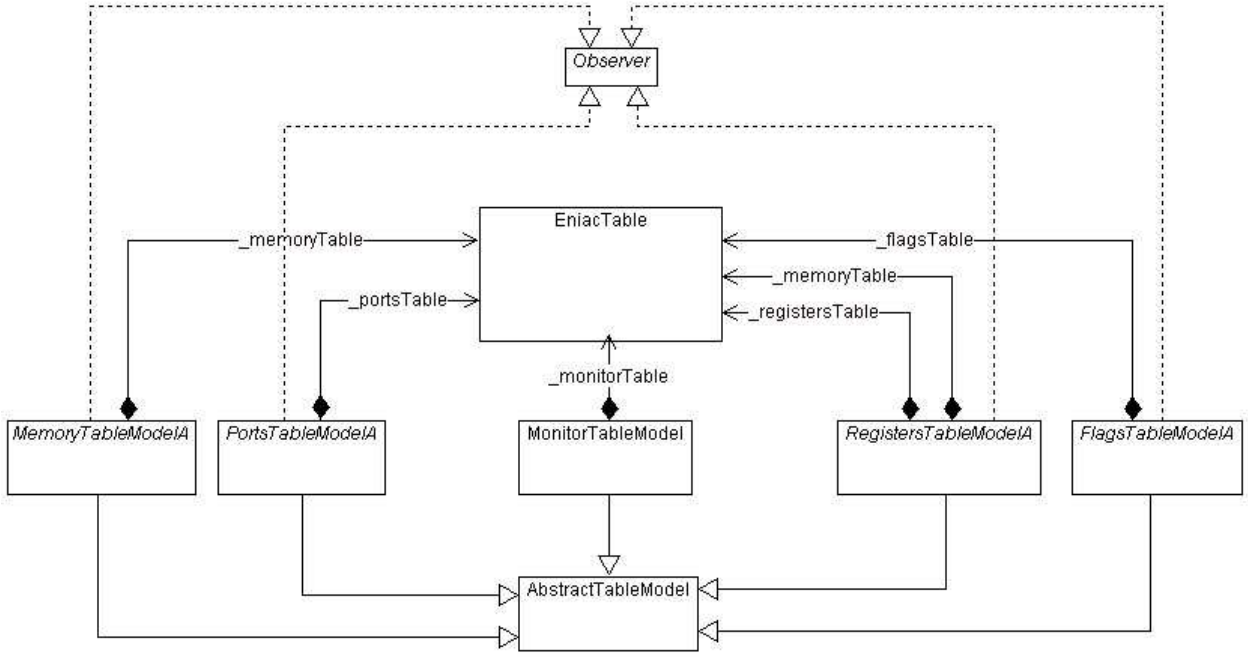


Figura 6.5.5

Per ciò che riguarda le tabelle visualizzate nel *front end*, poiché l'oggetto *JTable* aveva alcuni comportamenti grafici inappropriati alla situazione, è stato creato *EniacTable* e cioè un oggetto *JTable* con personalizzazioni come la larghezza delle colonne o menù a tendina onde inserire i dati nelle celle; inoltre per permettere una maggiore dinamicità, come mostrato in figura

6.5.5, gli oggetti dei modelli delle tabelle hanno anche un riferimento alla tabella stessa di cui fanno parte e quando necessario possono interagire con essa, ad esempio per effettuarne il *repaint()*.

Volendo trattare il primo punto bisogna fare una premessa: le celle delle tabelle sono normalmente editabili, ed al cambiamento del dato, questo viene rilevato automaticamente dal modello della tabella.

Proprio dalla classe *AbstractTableModel* viene ereditato un metodo:

- *addTableModelListener(TableModelListener)*: ha il compito di registrare su quel modello della tabella un oggetto *TableModelListener* in grado di ascoltarne i cambiamenti e gestirli.

Quindi fondamentalmente un oggetto in grado di ascoltare gli eventi legati a una tabella deve implementare il metodo *tableChanged(…)* dell'interfaccia *TableModelListener*, metodo il cui compito nel caso del *front end* di ENIAC è riconoscere la validità dell'input immesso nella tabella e trasformarlo in un dato di tipo *Word,* che potrà quindi essere inserito direttamente nella risorsa equivalente di ENIAC.

Poiché, come già evidenziato, i dati visualizzati a schermo possono essere rappresentati in formato binario, esadecimale, decimale o sintassi, a seconda della tabella sono dichiarate più classi di tipo *TableModelListener* ognuna delle quali verrà registrata a seconda della scelta dell'utente:

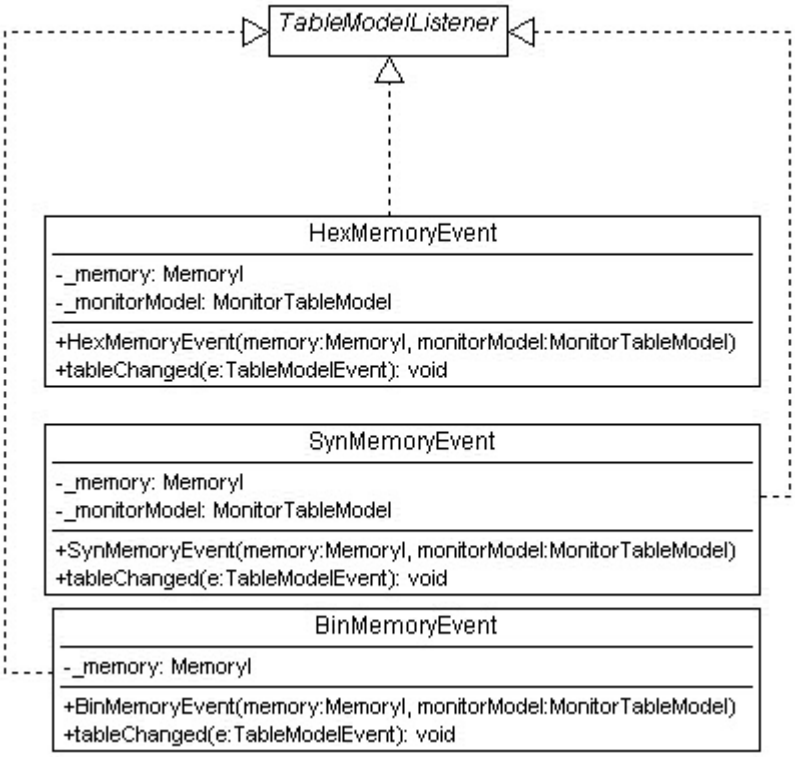


Figura 6.5.6

Quindi in definitiva il metodo *actionPerformed(…)* nelle *classi interne* di tipo *xxxFormatAction,* ha il compito, in base alla modalità di visualizzazione dei dati, di selezionare per ogni tabella il modello *AbstractTableModel* esatto e registrare su esso le corrette classi *Observable*, cioè la risorsa, e il gestore degli eventi *TableModelListener*; si allega un esempio per la tabella della memoria nel caso di visualizzazione dei dati in base due (o binario):

```
private class BinFormatAction extends AbstractAction {
        (…….)
        public void actionPerformed(ActionEvent evt) {
                memoryModel = new BinMemoryModel(memoryTable);
                memoryTable.setModel(memoryModel);
                memoryTable.setLeftColumn();
                memory.addObserver(memoryModel);
                memoryEvent = new BinMemoryEvent(memory, monitorModel);
                memoryModel.addTableModelListener(memoryEvent);
        }
}
```

Figura 6.5.7

Nella parte bassa della GUI è presente un'ultima tabella che costituisce il Monitor; è stato definito anche in questo caso il modello della tabella, il *MonitorTableModel*.

Attraverso il modello sono mostrati gli errori occorsi durante la scrittura o l'esecuzione di programmi scritti in linguaggio assembly di vCPU. Questi errori altro non sono che le eccezioni, di vario tipo, generatesi per qualche motivo dalla *business logic* (il simulatore) e dalla GUI, queste vengono catturate per poi rimapparne il messaggio nel *MonitorTableModel*.

6.6 Accesso ai dispositivi

Occorre fare una premessa: in riferimento alla figura 3.9.1, si può osservare che l'interfaccia *DeviceI* estende l'oggetto *Observer*, la cui funzione è descritta nel parag. 6.5; questa è una conseguenza del fatto che i dispositivi sono interfacciati ad ENIAC attraverso le porte, perciò quando un dato viene scritto in queste, dall'utente o da ENIAC stesso, l'effetto desiderato è che sia notificato al dispositivo.

Come poter usare i dispositivi? – nei paragrafi 3.9 e 3.10 sono stati introdotti rispettivamente i dispositivi e la factory dei dispositivi.

Quest'ultima, come già detto, restituisce una struttura dati di tipo *Set* che

verrà usata dalla classe *ENIACFrame* in fase di costruzione del menù dell'interfaccia denominato "Device": per ogni elemento contenuto nel *Set* viene istanziato un oggetto *JCheckBoxMenuItem* al quale è associata una *DeviceAction*, così strutturata:

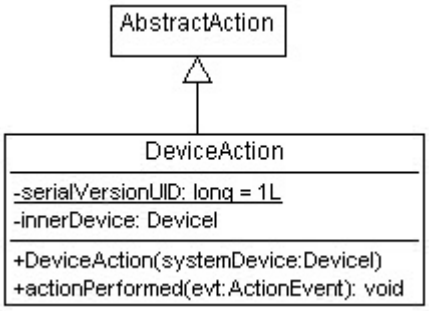


Figura 6.6.1

Come evidente, sono presenti un costruttore e un metodo come di seguito descritti:

- DeviceAction(DeviceI): il costruttore, riceve un oggetto di tipo *DeviceI* (vedi parag. 3.9), al quale sarà associata questa *AbstractAction*;
- actionPerformed(ActionEvent): in questo metodo, che verrà eseguito alla pressione del relativo *JCheckBoxMenuItem*, sono associati due diversi comportamenti:
  - alla selezione del *JCheckBoxMenuItem*, verrà registrato il dispositivo rappresentato come *Observer* dell'oggetto PortsI (che come si ricorda è di tipo *Observable*), e successivamente eseguito il metodo *plugDevice()* che attiverà il dispositivo;
  - viceversa, alla deselezione del *JCheckBoxMenuItem*, verrà eseguito il metodo *unplugDevice()* per disattivare il dispositivo, dopodiché verrà rimosso come *Observer* delle porte.

6.7 Sintassi di ENIAC

- package *eniac.util*

Tra le modalità di visualizzazione selezionabili c'è anche il formato in

linguaggio assembly riconosciuto da vCPU; ma come è possibile il riconoscimento in ENIAC?

E' stato necessario programmare un analizzatore lessicale, rappresentato dalla classe *ENIACSyntax*; questo strumento usa largamente gli oggetti *Pattern* di Java, oggetti che in base a una espressione regolare possono riconoscere la corrispondenza o meno con una data stringa di testo.

Poiché il numero di istruzioni, come indicato nel paragrafo 5.2, può evolvere è stato necessario implementare un meccanismo che, a tempo di esecuzione del simulatore, notificasse in qualche modo all'analizzatore le istruzioni presenti; per questo scopo è stata sfruttata la *DefaultInstructionFactory*.

Come indicato nel paragrafo 4.2 la *DefaultInstructionFactory* crea dinamicamente, una alla volta, istanze di classi delle istruzioni che poi verranno inserite all'interno di una *HashMap*; proprio durante tale processo, per ogni istruzione istanziata, viene creato un oggetto di tipo *InstructionData* sulla base della informazioni estrapolate da essa:

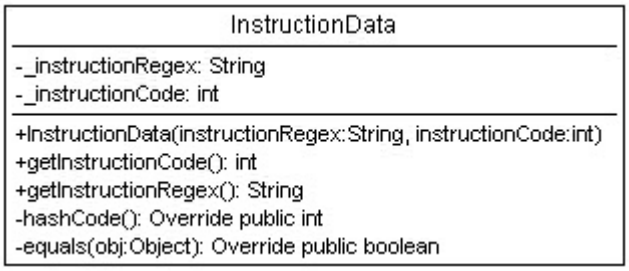


Figura 6.7.1

Nella figura 6.7.1 è mostrato un costruttore accettante due dati:

- *instructionRegex*: contenente la stringa dell'espressione regolare per l'istruzione processata in quel momento; essa comprende non solo il formato per il riconoscimento del nome dell'istruzione (esempio "ADD"), ma anche del suo argomento (esempio "@100");
- *instructionCode*: l'intero che risponde al codice operativo di quella data istruzione, quindi il valore dei bit compresi dal b13 al b23.

A questo punto è possibile passare alla spiegazione dei metodi di *ENIACSyntax*:

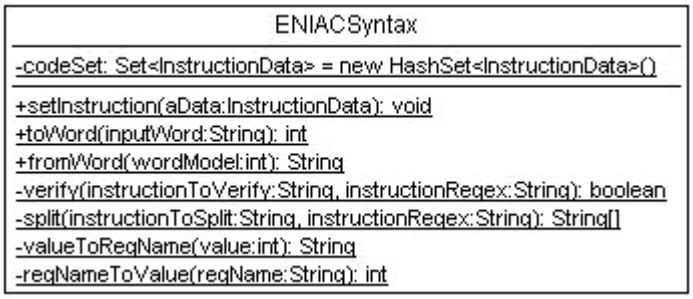


Figura 6.7.2

Dal diagramma UML 6.7.2 sono visibili alcuni componenti *private,* come dei metodi che contengono codice per operazioni che vengono eseguite con una certa frequenza e una struttura dati rappresentata da un oggetto *HashSet()*; quest'ultimo viene popolato di oggetti di tipo *IstructionData*, attraverso il metodo *setInstruction(…)*, che come già detto vengono creati dalla factory delle istruzioni a tempo di esecuzione. La scelta di una struttura dati è subordinata ai vincoli e al tipo di operazioni necessarie al programmatore; ne segue una lista di quanto osservato:

- le istruzioni sono univoche, pertanto lo saranno anche gli oggetti *InstructionData*;
- nel passaggio da stringa a parola e viceversa, verrà effettuato un confronto con ogni dato presente nella lista, per verificarne la presenza o meno;
- il fattore velocità, influenzato dalla complessità degli algoritmi di gestione della struttura dati.

Per questi motivi la scelta è ricaduta sul tipo *HashSet()* in quanto quella che, rispetto a un *ArrayList()* o ad altre strutture dati concesse da Java, meglio risponde a questi requisiti.
Sono presenti altri metodi, usati quando attivata dalla GUI la modalità di visualizzazione in formato linguaggio assembly di vCPU, quali:

- *toWord(…)*: usato dal gestore degli eventi delle tabelle, converte la stringa passata come argomento in un intero, che solo successivamente diventerà una parola a 24 bit destinata ad essere caricata in una risorsa;
- *fromWord(…)*: usato dai modelli delle tabelle, nel caso avvenga una modifica nello stato di una risorsa (esempio: cambio del valore di una parola nella risorsa); converte il valore intero dentro una generica parola di 24 bit nella sua rappresentazione in stringa, se la circostanza la rende possibile.

Questi metodi, come pure l'oggetto *HashSet(),* sono dichiarati *static* perciò per accedervi non è necessario istanziare l'oggetto *ENIACSyntax;* inoltre più istanze del simulatore su una stessa macchina faranno uso sempre della stessa istanza generica di *ENIACSyntax* perciò ci sarà un risparmio di memoria.

Bibliografia

- Thinking in Java 3rd edition, Bruce Eckel, 2003 – APOGEO;
- Java 2 I fondamenti Sesta edizione, Cay S. Horstmann – Gary Cornell, 2004 – McGraw Hill;
- Sun Java Tutorial: http://java.sun.com/docs/books/tutorial/
- Thinking in Patterns, Bruce Eckel, 2003;
- GoF's Design Patterns in Java, Franco Guidi Polanco, 2002;
- ANT User Manual, vari autori, 2001 – Apache Software Foundation;

Glossario

MSB: **M**ost **S**ignificant **B**it, ovvero il bit più significativo del dato, situato alla sua estrema sinistra; su 24bit che vanno da b23.....b0 il valore di MSB è quello di b23.

LSB: **L**ess **S**ignificant **B**it, ovvero il bit meno significativo del dato, situato alla sua estrema destra; su 24bit che vanno da b23.....b0 il valore di LSB è quello di b0.

Complemento a due: ovvero una modalità di rappresentazione dei numeri relativi; il segno è dato dall'MSB, se MSB vale 1 il numero è negativo, se MSB vale 0 il numero è positivo.
Inoltre se su 24bit si dovesse verificare che un dato sia rappresentabile solo con 12bit, i restanti 12bit altro non farebbero che estendere il segno; quindi se negativo sarebbero 12bit posti a 1, viceversa a 0.
Supposto a 1 l'MSB di 24bit: $-2^{23}+2^{22}+\ldots\ldots\ldots..+2^{1}+2^{0;}$
Supposto a 0 l'MSB di 24bit: $+2^{22}+\ldots\ldots\ldots..+2^{1}+2^{0;}$