

MIC1-SYS

Paolo Amato

Sommario

MIC1-SYS	1
1) Processo di sviluppo software.....	3
2) Fase di avvio del progetto.....	5
2.1) Vision	5
2.2) Documentazione delle specifiche supplementari	7
2.3) Glossario	8
3) Specifica dei requisiti.....	10
3.1) Attori e obiettivi	10
3.2) Modello dei casi d'uso	11
3.2.1) EseguiProgramma	11
3.2.2) MostraInformazioniArchitettura.....	12
3.2.3) AssemblaCodice	13
3.3) Lista delle funzionalità.....	14
3.3.1) Emulazione dell'ambiente IJVM	14
3.3.2) Compilazione	15
3.3.3) Interfaccia grafica	16
3.3.4) Gestione lista programmi e lista microprogrammi	17
4) Analisi dei requisiti.....	18
4.1) Prima iterazione	18
4.1.1) Vista casi d'uso	18
4.1.2) Architettura logica.....	20
4.1.3) Logica applicativa	21
4.1.4) Gestione degli eventi provenienti dall'esterno.....	28
4.2) Seconda iterazione	31
4.2.1) Considerazioni	31
4.2.2) Logica di Business	31
4.2.3) GUI.....	33
4.2.4) Persistenza: File System	36

4.2.5) Gestione lista programmi e lista microprogrammi	42
5) Progettazione funzionale e non-funzionale	43
5.1) Prima iterazione	43
5.1.1) Architettura logica del componente emulatore.....	43
5.1.2) Raffinamento della dinamica del caso d'uso EseguiProgramma	46
5.1.3) Vista “Componenti e Connettori”	60
5.2) Seconda Iterazione	62
5.2.1) Architettura logica del componente emulatore.....	62
5.2.2) Raffinamento della dinamica del caso d'uso EseguiProgramma	64
5.2.3) Interfaccia grafica	72
5.2.4) Gestione flussi di controllo concorrenti e trasporto delle informazioni sullo stato	76
5.2.5) Gestione persistenza: lista programmi e lista microprogrammi	86
6) Prospettiva dell'implementazione	95
6.1) Prima iterazione	95
6.1.1) Prospettiva di deployment	95
6.2) Seconda iterazione	97
6.2.1) WindowsForms.....	97
6.2.2) Configurazione del progetto e delle librerie.	98
6.2.3) Resharper	99
7) Descrizione del testing	100

1) Processo di sviluppo software

Per la realizzazione del sistema software da presentare in sede d'esame si è deciso di adottare un processo di sviluppo del tipo **Agile UP**, sviluppato da **Scott Ambler**. In sostanza, questo significa che si farà uso del framework che **UP** mette a disposizione e di conseguenza dei concetti da esso forniti come gli artefatti da creare, le differenti discipline, le fasi di ideazione ed elaborazione e così via. Tuttavia, tale framework viene utilizzato focalizzandosi su **un modo di lavorare agile**, vale a dire:

- Approcciare la realizzazione del software mettendo da parte l'idea di poter avere a disposizione, sin dalle prime fasi del progetto, di tutti i requisiti software funzionali e non da implementare. È importante rispettare le tre caratteristiche fondamentali di un tale processo di sviluppo: il software va sviluppato in modo incrementale, iterativo ed evolutivo. **I requisiti dovranno evolvere con il passare delle iterazioni.**
- **Eseguire in maniera frequente dei workshop di breve durata.** Questa caratteristica si presta evidentemente ad un lavoro di gruppo. Tuttavia, pur essendo da solo, ho approfittato di questi momenti per individuare nuovi requisiti, problemi nei requisiti già presenti, eventualmente andando a ragionare su possibili problematiche in termini di progettazione ed implementazione.
- **Modellare secondo lo spirito agile.** Conformarsi al modo di lavorare agile non significa affatto non effettuare modellazione. L'obiettivo della modellazione è quello non di documentare, ma di aiutare a comprendere. Secondo questo spirito, la modellazione relativa ad aspetti meno importanti del sistema software risulterà leggera o completamente assente, mentre si riserverà una maggiore attenzione ed importanza a quegli aspetti del sistema che possono essere ritenuti più critici.
- **Concentrazione su attività ad elevato valore.** Sulla base di quanto già accennato con la modellazione agile, in generale si può affermare quanto segue: l'attenzione deve concentrarsi sulle attività che contano realmente, e non su ogni cosa possibile che possa accadere durante un progetto.
- **Fare uso dello strumento denominato GitHub,** di modo tale da poter controllare in maniera efficace le diverse versioni degli artefatti di volta in volta prodotti e, dunque, gli output delle diverse iterazioni.

Non ho potuto ovviamente non tenere conto della mia inesperienza pratica nel gestire la costruzione di un sistema software secondo quelle che sono le pratiche fondamentali di questo

processo di sviluppo software, motivo per cui molto spesso ho dovuto ragionare attentamente sul corretto modo di procedere ed operare, realizzando, ogni volta, quella che mi è parsa la scelta più giusta.

Per concludere questo paragrafo riguardante la descrizione del processo di sviluppo, brevemente andrò ad indicare le pratiche agili fondamentali alle quali ho fatto riferimento:

- **Sviluppo iterativo, incrementale ed evolutivo**
- **Design semplice**
- **Versioning**
- **Refactoring**
- **Prioritization:** Lo sviluppo della soluzione può cominciare solo dopo aver messo in priorità gli obiettivi, dai quali deriveranno i requirements e le features
- **Semplicità:** semplicità nel codice, semplicità nella documentazione, semplicità nella progettazione, semplicità nella modellazione; i risultati così ottenuti sono una migliore leggibilità dell'intero progetto ed una conseguente facilitazione nelle fasi di correzione e modifica;

2) Fase di avvio del progetto

2.1) Vision

Il principale scopo del sistema software che si vuole costruire è quello di **emulare una architettura hardware**. Ciò che fondamentalmente il sistema fornisce è un emulatore, vale a dire una applicazione che, dalla prospettiva comportamentale, funziona in maniera del tutto analoga al dispositivo hardware che si vuole emulare. Ponendo attenzione al fattore complessità, e alle conoscenze pregresse di cui dispongo, sono giunto alla decisione di produrre un sistema software in grado di **emulare la IJVM e in particolare il MIC-1**, un processore inventato da Tanenbaum a scopo didattico in ambito universitario. Si tratta di una architettura da me studiata al corso di “Calcolatori Elettronici II”, che viene presentata per due motivi essenziali:

- mostrare come, usando elementi logici di base, sia possibile realizzare una microarchitettura che implementi un semplice ma completo set di istruzioni;
- mostrare come anche la realizzazione di un sistema hardware apparentemente complesso, come un processore, si riduca in realtà alla progettazione di un’unità operativa e di un’unità di controllo, e del modo in cui devono comunicare.

Un emulatore di una generica architettura hardware, in quanto tale, deve mettere a disposizione tutte le funzionalità dello strato hardware che sta emulando. Si tenga presente che, in generale, non è necessario, come già sottolineato in precedenza, realizzare una emulazione a livello elettrico, in termini di porte logiche, componenti combinatori e sequenziali. Lo stesso risultato, infatti, può essere ottenuto analizzando il funzionamento del dispositivo in termini comportamentali. Si terrà conto di entrambe le prospettive: quella elettrica, infatti, risulta utile per capire come i vari componenti del datapath della CPU collaborano per consentire l’esecuzione delle istruzioni. Sarà, a questo punto, sufficiente replicare il comportamento dell’architettura hardware attraverso opportuni strumenti tecnologici e sfruttando una macchina sulla quale sarà presente l’ambiente di esecuzione attraverso il quale verrà eseguito lo strato software che sintetizza l’emulatore.

Più nel dettaglio, **si vuole produrre non soltanto un emulatore per tale architettura, ma anche un assemblatore** che sia capace di tradurre il codice sorgente scritto dall’utente in codice macchina eseguibile proprio dall’emulatore. Ovviamente, l’utente sarà “obbligato” a scrivere codice secondo quello che è il modello di programmazione del processore MIC-1.

L’utente deve poter interagire con l’assemblatore e costruire l’eseguibile da dare in pasto all’emulatore secondo quelle che sono le opzioni messe a disposizione dal sistema stesso.

All’utente quindi, sarà reso possibile utilizzare una qualsiasi istruzione appartenente all’ISA della CPU considerata.

Inoltre, l’utente potrà stabilire se caricare sull’emulatore un programma scritto con l’assemblatore appartenente al sistema software oppure un programma già compilato.

Si tenga presente che la IJVM è un processore virtuale a tutti gli effetti, quindi ha il processore, ossia il MIC-1, la memoria, un ambiente di esecuzione, e così via.

L’emulatore dovrà occuparsi non soltanto dell’effettiva esecuzione di istruzioni e dunque di linee di codice. A tale componente è affidata anche la visualizzazione, mediante un’interfaccia grafica, di alcune informazioni interne al dispositivo hardware emulato:

- **Comportamento del processore**
- **Comportamento dei registri**
- **Comportamento dell’unità di controllo**
- **Memoria principale**
- **Memoria di controllo**
- **Struttura dello stack e informazioni in esso salvate**

Per concludere il documento di “Vision”, si può in definitiva affermare che **è centrale il desiderio di produrre un emulatore affidabile, ma è importante anche curare in maniera opportuna aspetti di contorno per l’emulatore stesso**, che in ogni caso hanno una loro importanza, soprattutto nell’ottica di facilitare l’utente a comprendere il modo di funzionare del sistema emulato.

2.2) Documentazione delle specifiche supplementari

In questa sezione ci si focalizza su tutto ciò che non sarà compreso nei casi d'uso, in particolare vengono analizzati quelli che sono i **requisiti non-funzionali** che il sistema software dovrebbe soddisfare.

Una applicazione di questo tipo, il cui compito è quello di emulare una già esistente architettura hardware, si rivela davvero utile nel momento in cui la sua **logica di emulazione è indipendente dalla maniera in cui i programmi sono predisposti per l'esecuzione**. Sin dall'inizio sarà fondamentale andare a realizzare una architettura fatta da **componenti e/o sottosistemi che siano facilmente riusabili e modificabili**, così da poter conformare l'interfaccia utente all'emulatore stesso, senza inficiare quella che è la sua logica di elaborazione.

L'approccio che intendo seguire richiede che il sistema soddisfi la proprietà della **modularità**, disaccoppiando i due sottosistemi fondamentali da cui il sistema software è costituito: emulatore e assemblatore. Difatti, l'assemblatore non deve generare codice interpretabile ed eseguibile esclusivamente dall'emulatore appartenente a tale sistema.

Si tenga presente che si sta adottando un processo di sviluppo software **iterativo, incrementale ed evolutivo**. Dunque, i requisiti non-funzionali di **manutenibilità ed evolvibilità** sono importanti da soddisfare. **Emulatore ed assemblatore dovrebbero essere suddivisi in moduli tra loro scarsamente accoppiati**, consentendo così di adoperare un tipo di sviluppo focalizzato sulle parti più critiche del sistema, per poi estendere e raffinare quanto fatto nelle prime iterazioni.

In generale, un emulatore potrebbe risultare scarsamente **usabile**, ragion per cui le scelte di progetto effettuate dovranno in qualche modo far sì che il sistema costruito sia il più **interattivo ed user-friendly** possibile.

2.3) Glossario

Vengono di seguito presentati dei termini che compariranno di frequente all'interno di tale documento, fondamentali per capire in maniera adeguata le parti più tecniche e specifiche del dominio applicativo. Comincerò mostrando tutte quelle parole significative del dominio utilizzate nella prima iterazione, inserendone poi altre con il proseguire dello sviluppo.

- **ALU:** questo termine identifica la parte del processore che si occupa di svolgere calcoli matematici e operazioni logiche. Si tratta generalmente di una macchina combinatoria.
- **Architettura a stack:** si tratta di un particolare tipo di struttura di processore, nel quale si presuppone che se devo eseguire una qualsiasi istruzione, gli operandi di cui essa necessita si trovano in cima allo stack della memoria con la quale il processore interloquisce.
- **BUS:** canale di comunicazione che permette a periferiche e componenti di un sistema elettronico di interfacciarsi tra loro scambiandosi informazioni o dati di sistema attraverso la trasmissione e la ricezione di segnali.
- **CPU:** Unità centrale di elaborazione, corrispondente in questo caso al processore MIC-1, del quale verrà implementato l'ISA.
- **IJVM:** architettura di elaborazione che prende in considerazione il sottoinsieme delle istruzioni della JVM che lavorano soltanto sui numeri interi.
- **IR:** registro della CPU che immagazzina l'istruzione in fase di elaborazione.
- **Istruzione:** tale concetto ingloba in sé due tipi di informazione: la codifica binaria dell'Opcode da far eseguire alla CPU, e la codifica binaria degli operandi da usare.
- **MAL:** linguaggio usato per semplificare la scrittura del microprogramma.
- **MAR:** registro interno all'unità operativa della CPU collegato al bus indirizzi della RAM ed utilizzato insieme al registro MDR per accedere in lettura/scrittura all'area dati della memoria centrale.
- **MDR:** registro interno all'unità operativa della CPU, collegato direttamente al bus dati della RAM.
- **Memoria di controllo:** si tratta di una ROM utilizzata nelle unità di controllo costruite in logica microprogrammata. Essa contiene per ogni istruzione dell'ISA, la relativa microprecedura.
- **Microistruzione:** parola ad n bit che fornisce i valori che i segnali di controllo, mandati dall'unità di controllo all'unità operativa, devono assumere.
- **Microprecedura:** la sequenza di microistruzioni relativa ad una data istruzione dell'ISA costituisce la microprecedura che implementa quella istruzione.
- **Microprogrammazione:** tecnica specifica utilizzata per la realizzazione dell'unità di controllo. Si contrappone alla logica cablata.
- **MIR:** registro della CPU che immagazzina la microistruzione in fase di elaborazione.
- **Opcode:** codice operativo in base al quale l'unità di controllo comanda l'unità operativa e stabilisce cosa quest'ultima deve fare.
- **PC:** registro interno alla CPU utilizzato per mantenere l'indirizzo della successiva istruzione da eseguire.
- **Programma:** Blocco di istruzioni che il MIC-1 è in grado di interpretare.
- **RAM:** Random Access Memory, si tratta della memoria principale utilizzata dal processore.

- **Shift Register:** registro a scorrimento collegato in uscita all'ALU.
- **Stack Pointer:** registro interno all'unità operativa della CPU utilizzato per mantenere l'indirizzo della cima dello stack.
- **Stack:** è un tipo di dato astratto che viene usato in diversi contesti per riferirsi a strutture dati, le cui modalità d'accesso ai dati in essa contenuti seguono una modalità LIFO. Lo stack è un elemento dell'architettura dei processori, e fornisce il supporto fondamentale per l'implementazione del concetto di subroutine.
- **Unità di controllo:** Il blocco a cui è delegato il controllo della parte operativa è costituito dall'unità di controllo. Non sussiste unità di controllo senza unità operativa; se non è stata dapprima sviluppata la parte da controllare, non ha senso sviluppare il controllo di tale parte.
- **Unità operativa:** Questo blocco incapsula la parte inherente all'elaborazione dei dati ottenuti in ingresso sulla base degli ingressi di controllo forniti dalla unità di controllo, e risponde fornendo uno stato che sarà utile alla unità di controllo per scandire le prossime operazioni da comandare al blocco di elaborazione. Al termine delle elaborazioni, in uscita verrà rilevato il prodotto di queste ultime.

3) Specifica dei requisiti

In questa parte della documentazione l'attenzione è posta sulla realizzazione di artefatti rappresentativi di use-case che siano incentrati sugli obiettivi degli utenti che andranno a fare uso del sistema software in questione. Proseguendo, ci si focalizzerà sulle differenti funzionalità messe a disposizione dall'applicazione, utilizzabili dall'utente, anche se non collegate ad un determinato obiettivo o scopo che l'utente stesso intende perseguire.

3.1) Attori e obiettivi

Attori:

- **Utente**

Obiettivi utente:

- **Esegui Programma**
- **Assembla Codice**
- **Setup Emulazione**
- **Mostra Informazioni Architettura**

Di seguito viene riportata una descrizione dei casi d'uso secondo quello che in UP è definito "formato breve":

- **EseguiProgramma:** Lo sviluppatore, dopo aver caricato un microprogramma e dopo aver selezionato un programma, potrà eseguire il programma stesso in modalità normale oppure Step-by-Step.
- **AssemblaCodice:** Consente di compilare il codice scritto generando nel file system locale un file assemblato che l'emulatore è in grado di eseguire.
- **SetupEmulazione:** Prima di caricare ed eseguire un programma, l'utente sarà in grado di vedere e modificare proprietà tipiche dell'emulazione come i valori iniziali dei registri, la velocità di esecuzione, i dispositivi di I/O ecc.
- **MostraInformazioniArchitettura:** Lo sviluppatore potrà scegliere di visualizzare informazioni su specifici elementi dell'architettura emulata, di modo tale da verificare il comportamento di tali componenti architettonici.

3.2) Modello dei casi d'uso

3.2.1) EseguiProgramma

Nome caso d'uso	EseguiProgramma
Attore primario	Utente
Portata	Livello Utente
Parti interessate ed interessi	Scegliere il programma e mandarlo in esecuzione
Pre-condizioni	Modo di esecuzione selezionato
Post-condizioni	Visualizzazione dell'esecuzione del programma
Scenario principale	<ol style="list-style-type: none">1) Scegli un microprogramma dalla lista2) Carica microprogramma3) Scegli un programma dalla lista4) Avvia programma
Estensioni/Scenari alternativi	<ol style="list-style-type: none">4a) L'utente mette in pausa il programma4b) L'utente termina il programma

3.2.2) MostraInformazioniArchitettura

Nome caso d'uso	MostraInformazioniArchitettura
Attore primario	Utente
Portata	Livello Utente
Parti interessate ed interessi	Visualizzare come lo stato dell'architettura e dei suoi diversi componenti evolve
Pre-condizioni	Un programma è già stato selezionato ed avviato
Post-condizioni	Visualizzazione informazioni riguardanti lo stato del dispositivo hardware emulato
Scenario principale	1) Scegli componente
Estensioni/Scenari alternativi	<p>1a) Se viene scelta la CPU</p> <ol style="list-style-type: none"> 1) Visualizza i valori dei registri, i segnali di controllo e gli ingressi inviati ai vari elementi del datapath, le operazioni da essi effettuate e i risultati da essi prodotti <p>1b) Se viene scelta la memoria centrale</p> <ol style="list-style-type: none"> 1) Seleziona intervallo di indirizzi 2) Visualizza la parte di memoria corrispondente <p>1c) Se viene scelta la memoria di controllo</p> <ol style="list-style-type: none"> 1) Visualizza le microprocedture e le microistruzioni ad essa appartenenti

3.2.3) AssemblaCodice

Nome caso d'uso	AssemblaCodice
Attore primario	Utente
Portata	Livello Utente
Parti interessate ed interessi	Produrre un file eseguibile a partire dal codice scritto
Pre-condizioni	L'utente dispone di un file contenente codice
Post-condizioni	File eseguibile generato
Scenario principale	1) Carica file contenente codice 2) Compila codice
Estensioni/Scenari alternativi	2a) Se il codice è valido, viene restituito un file eseguibile 2b) Se il codice è errato, viene restituito un messaggio di errore

3.3) Lista delle funzionalità

3.3.1) Emulazione dell'ambiente IJVM

Questo è sicuramente un **requisito funzionale**, che presenta al suo interno uno specifico insieme di altre features. Il nostro interesse è rivolto all'**emulazione di una CPU**, in particolare del processore **MIC-1**, ma è evidente che un qualsiasi processore, preso così com'è, non possa funzionare correttamente. Ha bisogno senza dubbio di una memoria centrale con la quale effettuare scambi di dati in lettura e/o scrittura. Tra l'altro, il MIC-1 è costruito in **logica microprogrammata**, ragion per cui un ruolo di fondamentale importanza è svolto dalla **memoria di controllo**, che può essere considerata alla stregua di una ROM. Di seguito viene riportata, in forma semplificata, l'architettura della IJVM:

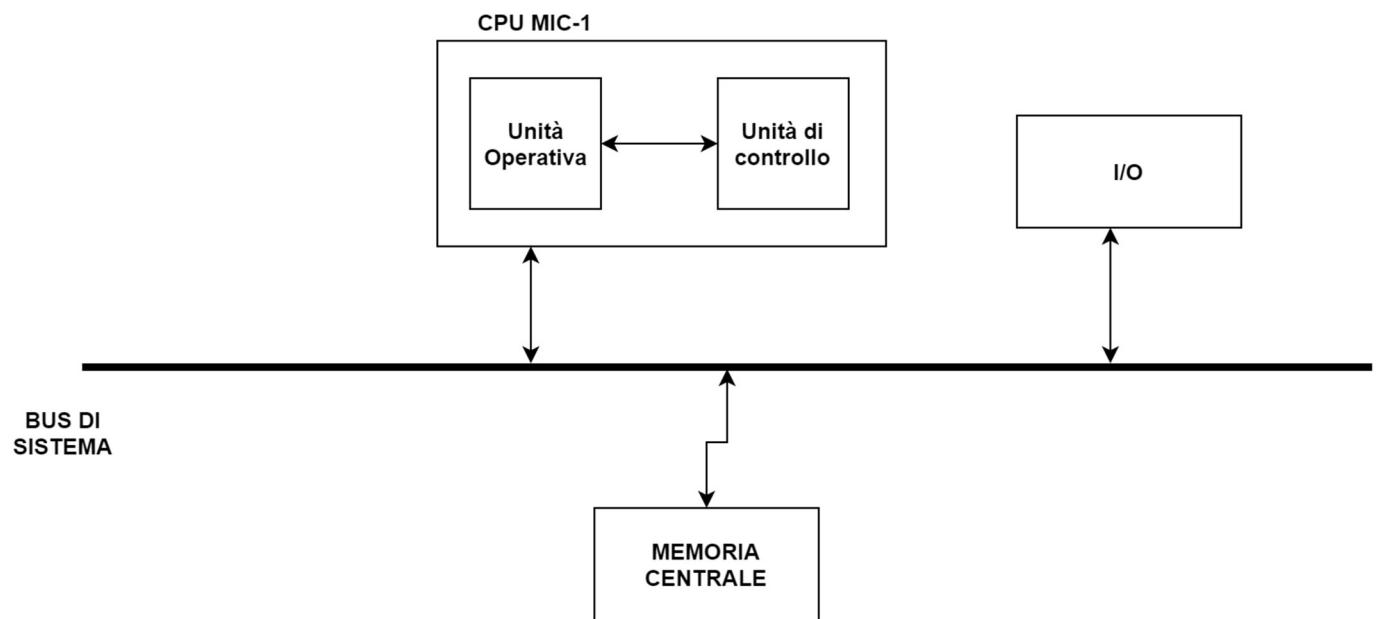


Figura 1: Architettura IJVM

La ROM (memoria di controllo) non è qui visibile in quanto componente interno alla CPU, appartenente alla sua unità di controllo. Tenere bene a mente questa architettura sarà di aiuto nel realizzare un componente emulatore la cui logica applicativa rispecchi in maniera adeguata l'hardware emulato. **Scopo principale dello sviluppo secondo la proprietà dell'iteratività è capire quali sono gli elementi più importanti e focalizzarsi sulla loro comprensione, per poi passare allo sviluppo completo del sistema.**

- **Linguaggio assembly del MIC-1:** L'architettura appena mostrata è dotata di un processore. Come per un qualsiasi altro processore, anche il MIC-1 ha un comportamento che può essere descritto attraverso il ciclo di Von Neumann. È possibile interagire con tale CPU sfruttando il suo modello di programmazione. **Un processore altro non è che un interprete:** presa in ingresso una istruzione, la esegue e fornisce in uscita dei risultati.

Siamo interessati a capire come la CPU, grazie al suo modello, riesce a rispondere alle istruzioni ad essa fornite. È inoltre di rilievo andare ad esplodere la struttura della singola istruzione: siamo nel caso di un processore realizzato in logica microprogrammata; dunque, è presente una memoria di controllo che, per ogni istruzione dell'ISA, memorizza la corrispondente microprocedura. Ogni microprocedura è fatta da un insieme di microistruzioni, e ogni microistruzione governa il datapath per un singolo ciclo di clock. Quando una microistruzione viene letta, ciascun segnale di controllo assume il valore del bit corrispondente.

- **Linguaggio MAL:** Scrivere a mano le microistruzioni che formano la microprocedura di una data istruzione dell'ISA è perfettamente possibile, ma è molto semplice commettere errori; inoltre, il microprogramma così ottenuto sarebbe tedioso da comprendere e modificare. Il linguaggio usato per semplificare la scrittura del microprogramma è denominato MAL (MicroAssembly Language). L'emulatore, come sappiamo, deve essere in grado di interpretare il linguaggio assembly del MIC-1, ossia deve saper interpretare le istruzioni dell'ISA della IJVM. Questo significa che tale componente deve essere in grado di interpretare le microistruzioni delle diverse microprocedure e, dunque, codice scritto in linguaggio MAL.

3.3.2) Compilazione

Anche questo è un requisito funzionale. **L'utente, alla consegna del sistema, potrà interagire con esso per generare un eseguibile interpretabile dall'emulatore** dopo aver caricato dal file system un file contenente codice conforme al linguaggio assembly del MIC-1. **Inoltre, l'utente potrà usare il sistema per generare, a partire da un microprogramma scritto in linguaggio MAL, un microprogramma compilato** ed effettivamente caricabile nella memoria di controllo.

3.3.3) Interfaccia grafica

Il sistema software in questione deve essere capace di gestire dinamicamente le interazioni con l'utente, ragion per cui, nella lista delle funzionalità, compare **l'interfaccia grafica**. L'utente, alla consegna dell'applicazione, potrà interagire con il sistema per stabilire proprietà dell'emulazione, la modalità di esecuzione ecc. **L'utente potrà caricare il file macchina del programma da eseguire selezionandolo da una lista di programmi mantenuta localmente dal sistema sotto forma di file opportunamente strutturato.**

L'utente potrà caricare il file macchina del microprogramma necessario per interpretare le istruzioni dell'ISA selezionandolo da una lista di microprogrammi mantenuta localmente dal sistema sotto forma di file opportunamente strutturato.

Le due disponibili modalità di esecuzione sono:

- **Utente:** in questa modalità l'unico scopo dell'utente è quello di poter usare l'emulatore come semplice esecutore di programmi, avendo soltanto la possibilità di visualizzare lo stack per prelevare i risultati di una elaborazione.
- **Esperto/Programmatore:** Questa è una modalità molto più tecnica. Si potrà non soltanto fare tutto ciò messo a disposizione nella modalità utente, ma anche vedere le informazioni in continuo cambiamento nei registri della CPU, nell'ALU, nello Shift Register, nei due bus interni al processore (B e C), nella memoria principale, nella memoria di controllo, utilizzare l'assemblatore e così via. Sarà possibile inoltre sospendere l'esecuzione, riprenderla, terminarla o eseguirla passo per passo. L'esecuzione step by step è utile nel momento in cui si vuole analizzare nel dettaglio il comportamento del programma caricato da parte del programmatore.

Data una prima interfaccia di configurazione, in base alla modalità selezionata, verrà mostrata una seconda interfaccia la cui struttura dipende dal particolare modo di funzionamento selezionato (Utente o Programmatore).

3.3.4) Gestione lista programmi e lista microprogrammi

È prevista una gestione dei programmi intesi come dati di tipo binario salvati su sistemi di accumulazione dati persistenti. Analogamente vale per i microprogrammi. **L'applicazione deve consentire la gestione di programmi e microprogrammi attraverso le classiche operazioni che rientrano sotto l'acronimo CRUD.** L'accesso alla generica fonte di dati persistente deve avvenire in maniera trasparente all'utente.

4) Analisi dei requisiti

L'analisi dei requisiti è stata effettuata all'inizio di ogni iterazione e ogni volta si è fatto riferimento ad un sottoinsieme di requisiti funzionali da realizzare e requisiti non-funzionali da soddisfare e rispettare. Data la complessità del sistema da realizzare, alcuni requisiti hanno richiesto uno studio e un approfondimento sicuramente più corposi di altri.

4.1) Prima iterazione

La prima cosa da fare è stata individuare gli elementi critici del sistema software da dover sviluppare. Dall'inizio molta enfasi è stata posta sulla logica del componente emulatore appartenente al sistema software che si sta costruendo. L'elemento principale dell'architettura e anche quello più critico è senza dubbio il processore. Prima di questo, però, il sistema è stato organizzato secondo una prospettiva statica e strutturato per soddisfare i requisiti non-funzionali.

4.1.1) Vista casi d'uso

MIC1-SYS si presta ad avere pochi casi d'uso associabili a degli obiettivi dell'utente. La prospettiva dei casi d'uso riportata di seguito è dunque molto scarna. È fondamentale osservare che, come primo caso d'uso (**si fa riferimento al diagramma "UC_Emulatore"**), si è preso in considerazione quello denominato "**Eseguiprogramma**", che permetterà all'utente di poter eseguire un applicativo scritto secondo il linguaggio assembly supportato dal processore MIC-1, dopo aver caricato un opportuno microprogramma. In questa prima iterazione si è deciso di non implementare, nemmeno parzialmente, alcuna funzionalità dell'assemblatore, ragion per cui non

è stato prodotto nessun diagramma dei casi d'uso relativo a tale sottosistema/componente.

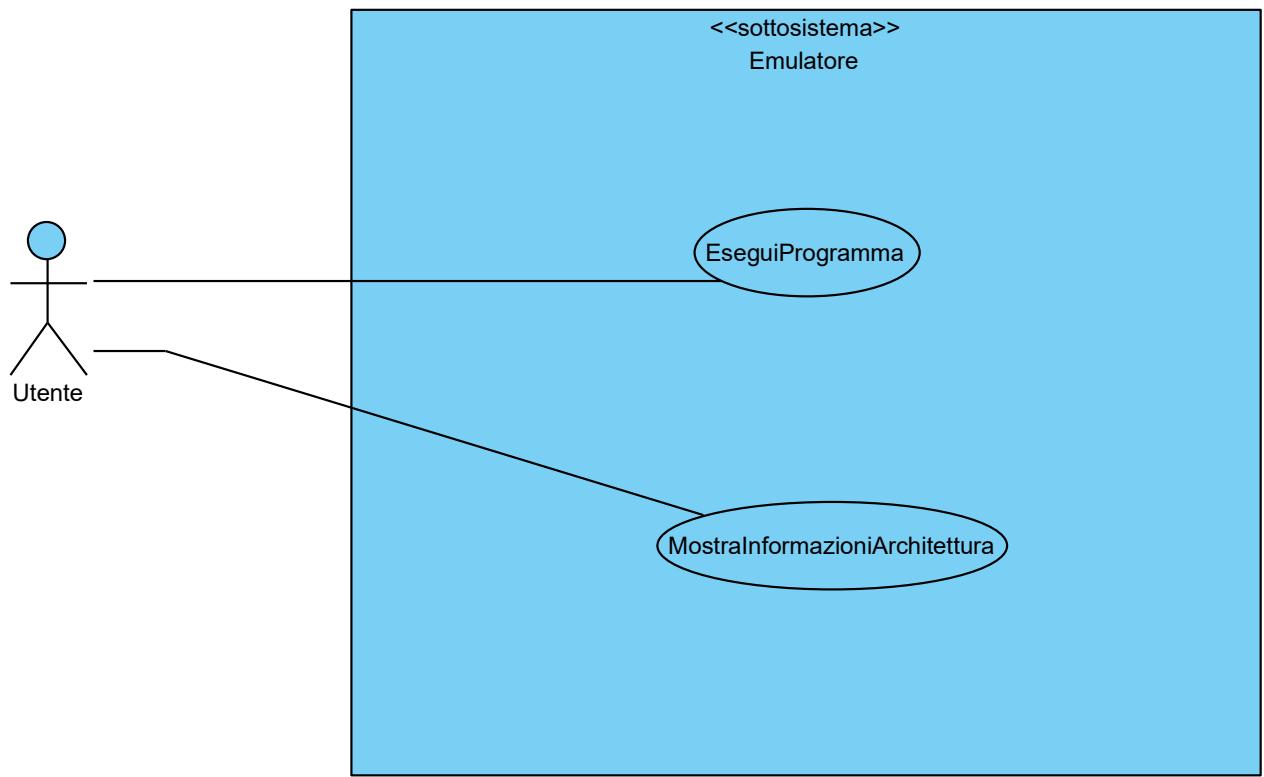


Diagramma 1: UC_Emulatore

4.1.2) Architettura logica

Viene per prima cosa effettuata la decomposizione modulare dei due sottosistemi da cui MIC1-SYS è formato, vale a dire assemblatore ed emulatore. Di seguito viene riportato un primo diagramma (“CD_MIC1-SYS_Architettura”) rappresentativo dei moduli fondamentali che costituiscono il sistema software

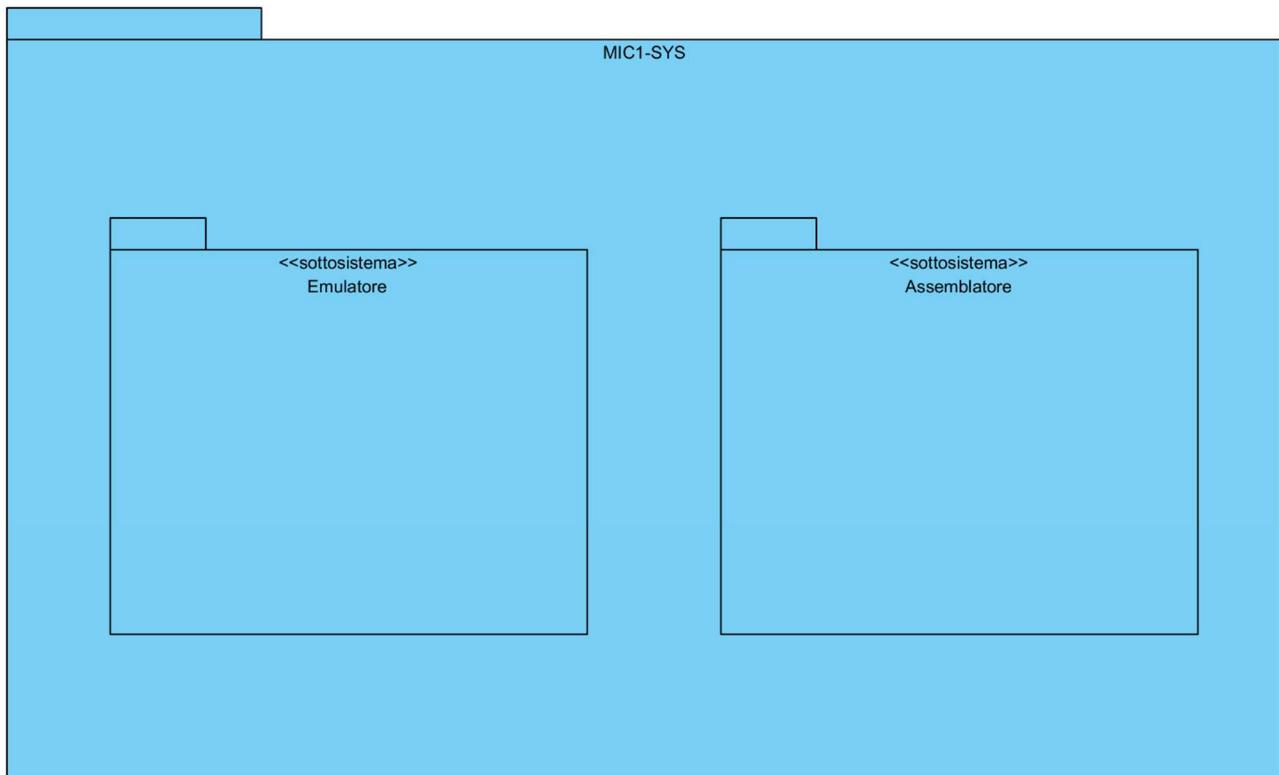


Diagramma 2: CD_MIC1-SYS_Architettura

4.1.3) Logica applicativa

L'approccio seguito è quello suggerito dal libro di Craig Larman. Si vanno quindi ad individuare le più importanti entità costitutive che influenzano il design in questa prima iterazione. A tal proposito, risulta necessario descrivere il componente che catalizza l'attenzione in tale iterazione, ossia la CPU.

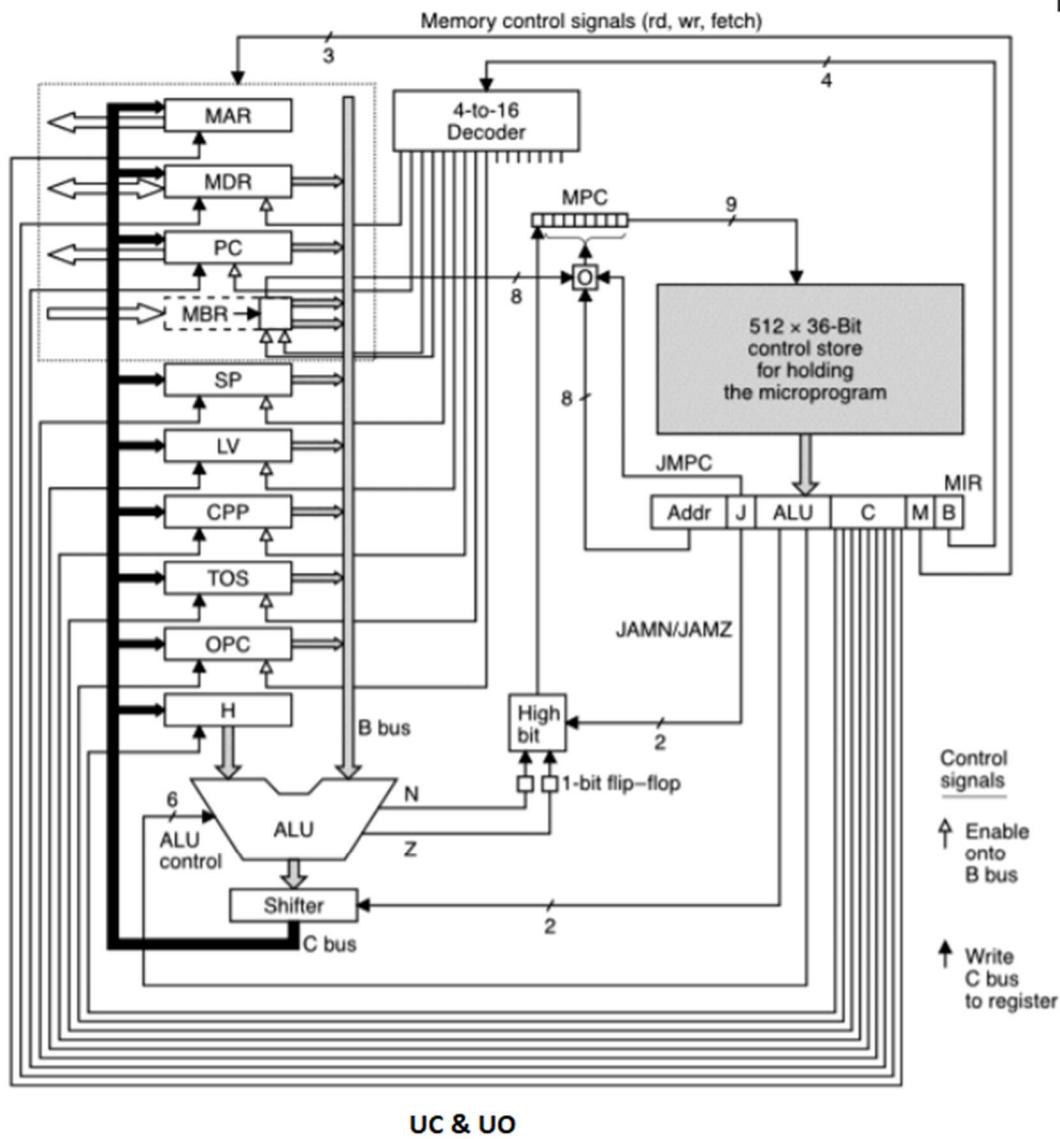


Figura 2: Struttura completa della CPU MIC-1

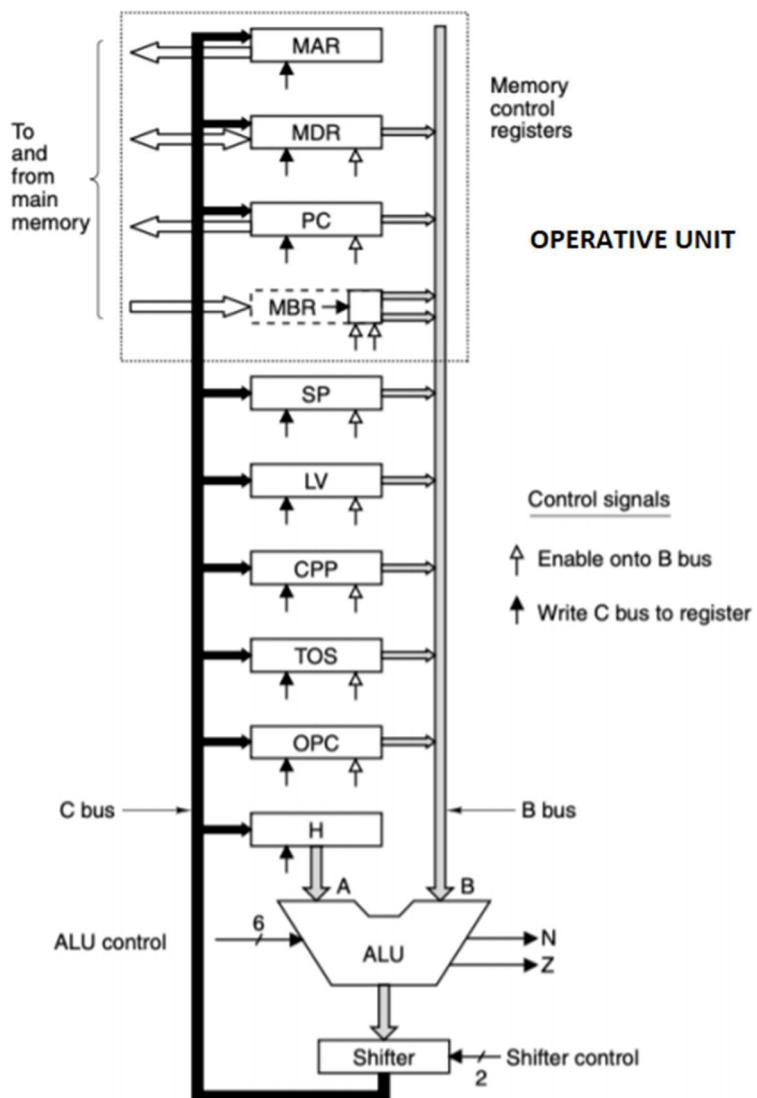


Figura 3: Unità operativa della CPU MIC-1

Nella prima figura è riportata la CPU con la sua struttura completa, includendo unità operativa e unità di controllo. Nella seconda figura, per semplicità, è riportata la sola unità operativa, ossia il datapath.

Questo processore è una macchina a stack, non a registri generali. Tutte le operazioni danno per scontato che gli operandi si trovino sullo stack.

L'unità operativa del processore comprende l'ALU, i suoi ingressi, e le sue uscite (tra cui i registri che si interfacciano con la memoria). **I registri hanno dimensione di 32 bit e non sono accessibili al programmatore, ma solo al microprogramma.** Il microprogramma è tipicamente memorizzato

in una ROM interna al processore. Quando diciamo che i registri non sono accessibili al programmatore intendiamo che essi non fanno parte del modello di programmazione, ossia non vengono utilizzati esplicitamente come operandi delle istruzioni. Vengono invece utilizzati dal microprogramma per implementare le istruzioni stesse. **L'unità operativa dispone di due bus, indicati con B e C, collegati rispettivamente al secondo ingresso e all'uscita dell'ALU; il primo ingresso dell'ALU è invece collegato esclusivamente al registro H (holding).**

Con alcune eccezioni, i registri dispongono di una coppia di segnali di controllo che permettono:

- di abilitare il collegamento del registro al bus B, rendendolo effettivamente l'operando B dell'ALU;
- di abilitare la scrittura sul registro del risultato fornito dall'ALU sul bus C.

Solo un registro può essere collegato al bus B in un determinato istante, mentre il risultato dello Shift Register (ossia il dato sul bus C) può essere scritto su più registri se necessario.

Registri dell'interfaccia con la memoria:

- MAR
- MDR
- PC
- MBR

$\begin{cases} \text{MAR} \\ \text{MDR} \end{cases}$ interfaccia per l'accesso in lettura/scrittura all'area dati della memoria principale

$\begin{cases} \text{PC} \\ \text{MBR} \end{cases}$ interfaccia per l'accesso in lettura all'area istruzioni della memoria principale

Registro che mantiene il primo operando dell'ALU:

- **H – Holding**

Gli altri registri sono funzionalmente equivalenti, ed i loro nomi sono assegnati sulla base dell'uso che se ne fa nel microprogramma:

- SP – stack pointer
- TOS – top of stack
- LV – local variables
- CPP – constant pool pointer
- OPC – scratch register

Ovviamente, tutta la CPU sarà temporizzata attraverso un opportuno segnale di clock.

Della CPU siamo interessati non soltanto al suo comportamento esterno, ma anche alla sua evoluzione dal punto di vista elettrico, quindi vogliamo capire quali sono i segnali di volta in volta inviati verso i diversi componenti dell'unità operativa e come essi si traducono in specifici comportamenti da parte di questi ultimi.

Il modello di programmazione del MIC-1 può essere facilmente delineato con un riferimento ai codici operativi e alle modalità di indirizzamento. Per quanto riguarda il secondo fattore, gli

operandi di una istruzione, come già accennato, sono sempre in cima allo stack: **siamo ben lontani dall'avere una architettura quasi-ortogonale come invece accade, ad esempio, nel 68000.**

Nome	Operandi	Descrizione
BIPUSH	byte	Scrive un byte in cima allo stack
DUP	N/A	Legge la prima parola sulla stack e compie push duplicandola
ERR	N/A	Stampa un messaggio di errore e arresta il simulatore
GOTO	nome etichetta	Salto incondizionato
HALT	N/A	Interrompe il simulatore
IADD	N/A	Sostituisce le due parole in cima allo stack con la loro somma
IAND	N/A	Sostituisce le due parole in cima allo stack con il loro AND logico
IFEQ	nome etichetta	Estrae la parola in cima allo stack ed esegue un salto se ha valore zero
IFLT	nome etichetta	Estrae la parola in cima allo stack ed esegue un salto se ha valore negativo
IF_ICMPEQ	nome etichetta	Estrae le due parole in cima allo stack ed esegue un salto se sono uguali
IINC	nome variabile byte	Somma una costante a una variabile locale
ILOAD	nome variabile	Scrive una variabile locale in cima allo stack
IN	N/A	Legge un carattere dal buffer della tastiera e lo scrive in cima allo stack. Se il carattere non è disponibile scrive il valore di 0
INVOKEVIRTUAL	nome metodo	Invoca un metodo
IOR	N/A	Sostituisce le due parole in cima allo stack con il loro OR logico
IRETURN	N/A	Termina un metodo restituendo un valore intero
ISTORE	nome variabile	Estrae la parola in cima allo stack e la memorizza in una variabile locale
ISUB	N/A	Sostituisce le due parole in cima allo stack con la loro differenza
LDC_W	nome costante	Scrive una costante proveniente dalla constant pool in cima allo stack
NOP	N/A	Nessuna operazione
OUT	N/A	Estrae la prima parola in cima allo stack e la scrive sulla periferica standard di output
POP	N/A	Estrae una parola dalla cima dello stack
SWAP	N/A	Scambia la posizione delle due parole in cima allo stack
WIDE	N/A	Istruzione Prefisso: l'istruzione seguente ha un indice a 16 bit

Figura 4: Istruzioni supportate dal processore MIC-1

Per quanto riguarda le istruzioni, **nella IJVM il formato dell'istruzione ha pochissimi campi ed è estremamente ridotto: c'è solo il codice operativo ed al massimo un solo operando.** Di conseguenza, la maggior parte delle istruzioni è codificata con un singolo byte, altre invece richiederanno due bytes per la codifica e altre ancora quattro (estensione WIDE).

Viene ora presentato il **formato adottato per le microistruzioni**, ciascuna rappresentata su 36 bit.

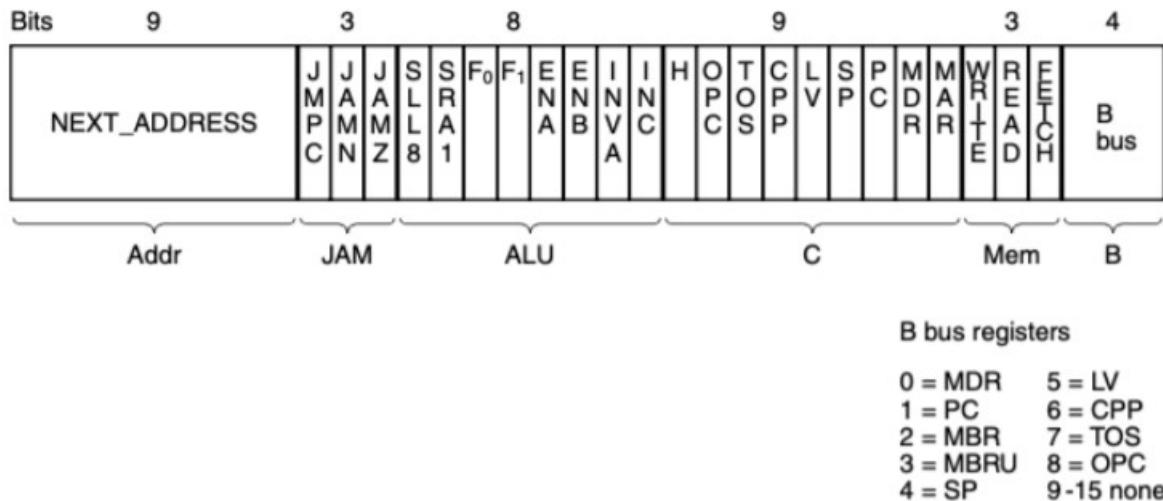


Figura 5: Formato di una generica microistruzione

Ogni microistruzione comprende i seguenti campi:

- **Addr** - Indirizzo di una potenziale prossima microistruzione;
- **JAM** - Determina come è selezionata la prossima microistruzione;
- **ALU** - Controllo dell'ALU e dello shifter;
- **C** - Controlla quali registri vengono scritti dal bus C;
- **Mem** - Controlla le operazioni di memoria;
- **B** - Seleziona il registro connesso al bus B

System Domain Model

Viene ora riportato il System Domain Model ("CD_SystemDomainModel"), realizzato secondo la notazione UML e facendo uso di quei costrutti che sono tipici di un Class Diagram.

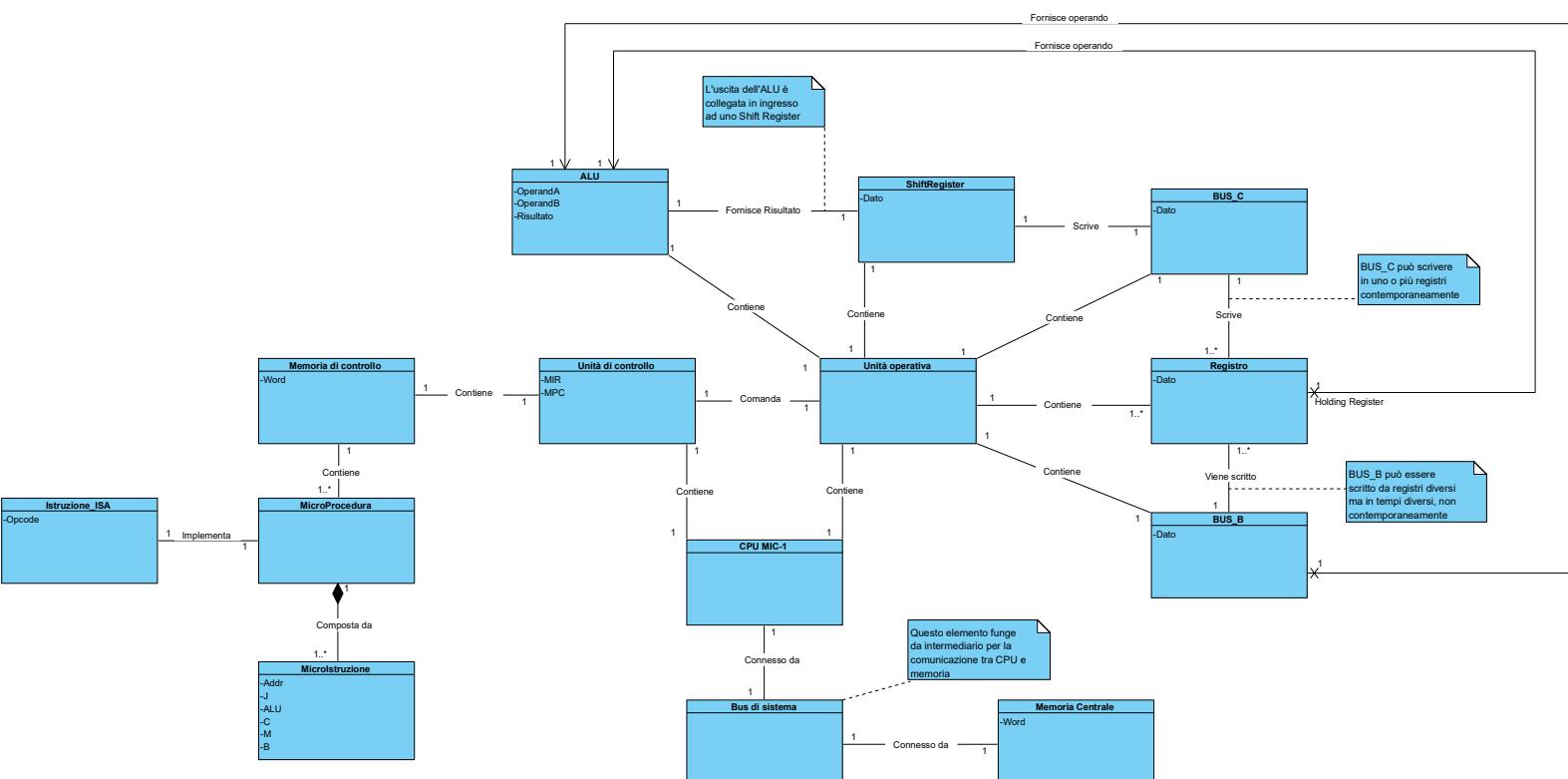


Diagramma 3: CD_SystemDomainModel

Lo scopo di questo diagramma è quello di catturare le prime entità concettuali appartenenti al dominio applicativo di interesse. Le classi presenti in questo diagramma sono rappresentative di concetti: hanno una propria **intensione**, ossia un proprio significato intrinseco, e una propria **estensione**, ossia un proprio modo di relazionarsi con le altre classi. Le classi presenti in questo diagramma non fanno riferimento ad alcuna classe software. Adesso, brevemente, andiamo a descrivere le classi concettuali individuate:

- **CPU MIC-1** – Il concetto rappresentato da tale classe è immediato: un’entità che si occupa di eseguire le istruzioni prelevate dalla memoria principale e che può essere schematizzata in termini di UC e UO.
- **Unità operativa** – Questa classe è direttamente collegata alla Unità di controllo. UO e UC sono due concetti ben distinti. L’UO evolve il proprio stato sulla base degli ordini ad essa impartiti da parte della UC.
- **Unità di controllo** – Una entità che, in base all’istruzione corrente di cui dispone, coordina l’esecuzione di un programma interagendo con opportune classi.
- **Registro** – I registri sono quelli presenti nell’architettura dell’UO e ne rappresentano lo stato; essi evolvono sulla base delle operazioni effettuate dall’UO.
- **Bus di sistema** – Questa classe rappresenta una entità che si occupa di gestire e direzionare la comunicazione tra i diversi elementi dell’architettura.
- **Memoria Centrale** – Questa entità è rappresentativa di uno spazio di informazioni dal quale la CPU può leggere o scrivere. Funge da sorgente dati che consente alla CPU di gestire il problema dovuto alla limitatezza dei registri di cui dispone.
- **ALU** – Questa classe rappresenta quell’entità del dominio, presente nell’unità operativa, che si occupa di gestire le operazioni logiche ed aritmetiche per la CPU.
- **Shift Register** – Questa classe è rappresentativa del registro a scorrimento presente nell’unità operativa e collegato in uscita all’ALU. Serve a “raffinare”, eventualmente, il risultato prodotto dall’ALU.
- **BUS_B** – Questa classe rappresenta una entità che si occupa di garantire il trasferimento di un dato da un registro dell’UO al secondo ingresso dell’ALU.
- **BUS_C** – Questa classe rappresenta una entità che si occupa di garantire il trasferimento di un dato dallo Shift Register ad uno o più registri dell’UO.
- **Memoria di controllo** – Questa entità è rappresentativa di uno spazio di informazioni utilizzato dall’unità di controllo per stabilire la successiva microistruzione da far eseguire all’unità operativa.
- **Microprocedura** – Questa classe serve ad esprimere il concetto di microprocedura. La CPU MIC-1 è realizzata in logica micropogrammata, quindi l’esecuzione di una generica istruzione ISA corrisponde all’esecuzione della sua relativa microprocedura.
- **Microlistruzione** – Questa classe rappresenta l’omonimo concetto, ossia una word codificata su n bit che serve a stabilire il modo in cui l’UC dovrà comandare l’UO nel corrente ciclo di clock.
- **Istruzione_ISA** – Questa classe rappresenta, in quanto concetto, una operazione supportata dalla CPU e imponibile in maniera diretta alla CPU stessa.

4.1.4) Gestione degli eventi provenienti dall'esterno.

Un problema che deve essere analizzato in maniera prioritaria è quello relativo alla cattura e alla gestione degli eventi esterni generati dall'utente del sistema. È stato prodotto un diagramma (“CD_Domain_Bound_Controller”), riportato di seguito, che estende il System Domain Model con nuove informazioni.

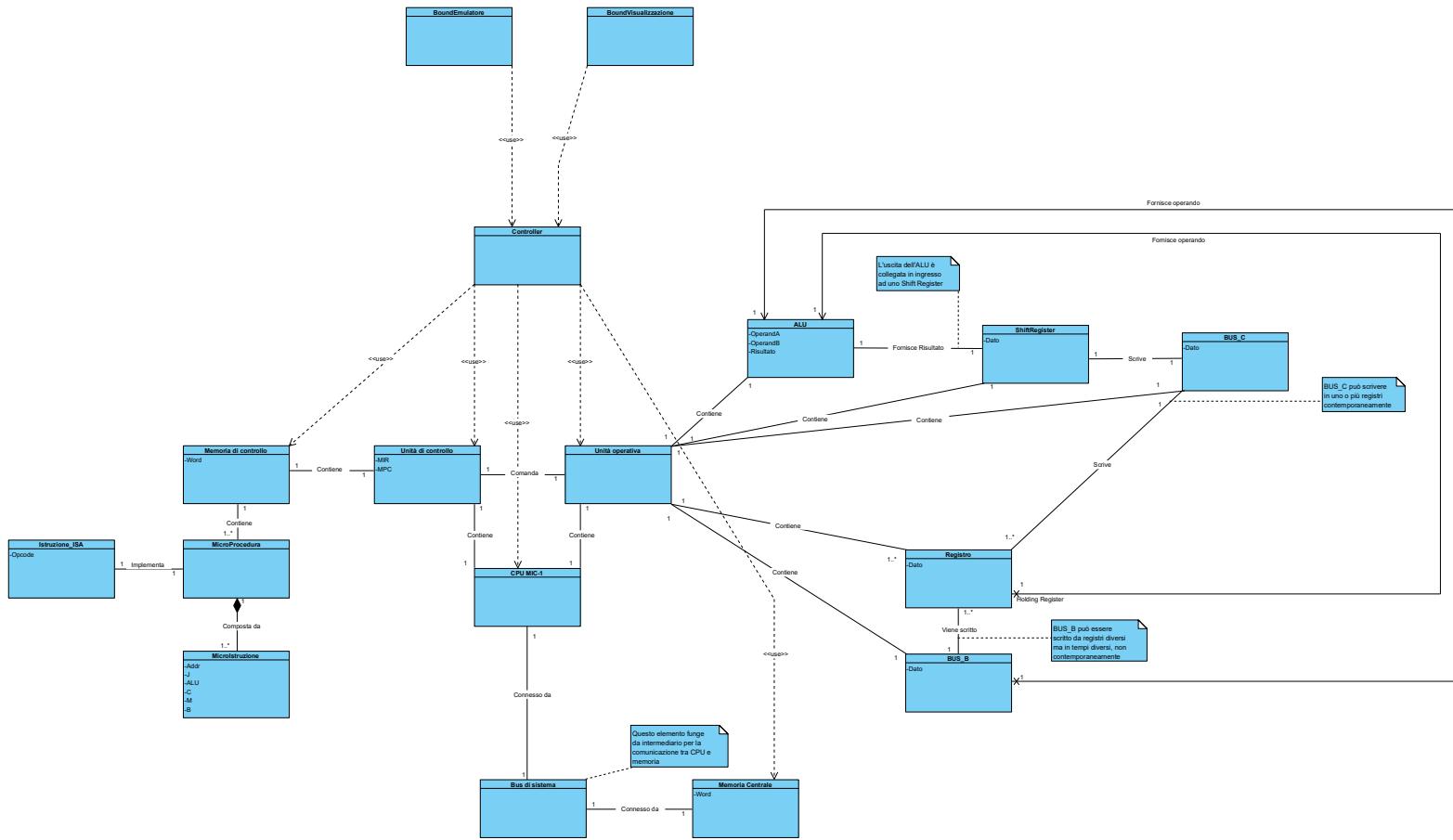


Diagramma 4: CD_Domain_Bound_Controller

Rispetto al System Domain Model, sono presenti tre nuove classi. Due di queste (**BoundEmulatore** e **BoundVisualizzazione**) hanno in sé concetti relativi ad una possibile interfaccia grafica che, difatti, rientra nella lista delle features. **La classe BoundEmulatore si dovrà occupare di iniziare la comunicazione con l'utente utilizzatore del sistema**, permettendo all'utente stesso di comunicare con il componente emulatore. **La classe BoundVisualizzazione, invece, sarà responsabile di inviare all'utente informazioni di tipo grafico**. Come osservabile dal diagramma qui inserito, si è

fatto uso del pattern “GRASP” denominato “Controller”: è stata introdotta una classe “Controller” il cui obiettivo è quello di delegare le richieste provenienti dagli oggetti del livello UI agli oggetti appartenenti al livello di dominio. Quindi il Controller riceve una richiesta dal livello UI e poi controlla/coordina altri oggetti del livello di dominio per soddisfare la richiesta.

A questo punto, è possibile mostrare un primo diagramma della dinamica da tenere in considerazione nella fase di design. Questo diagramma di sequenza (“**SQD_EseguiProgramma**”) ci consente di visualizzare in maniera essenziale il flusso di controllo associato al caso d’uso **EseguiProgramma**:

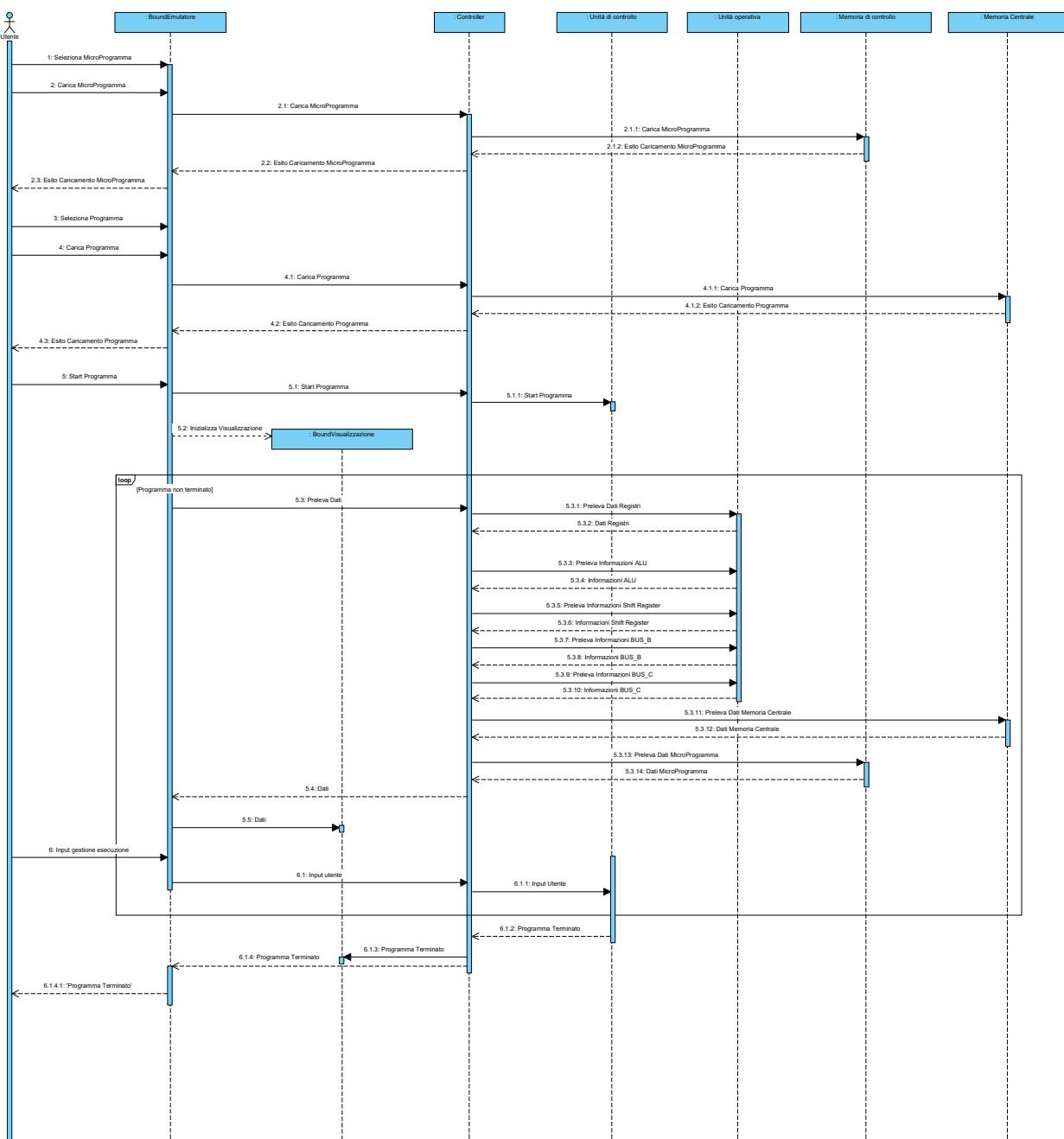


Diagramma 5: SQD Eseguiprogramma

L'utente, sfruttando l'interfaccia grafica, interagisce con la classe denominata

“BoundEmulatore”: dapprima si sceglie il microprogramma, caricato attraverso la classe “Memoria di Controllo”, poi si sceglie il programma, caricato attraverso la classe “Memoria Principale”; si ha successivamente l'avvio del programma, l'inizializzazione della visualizzazione, il prelievo ripetuto dello stato del dispositivo hardware emulato e la possibilità di gestire determinati input forniti dall'utente.

4.2) Seconda iterazione

4.2.1) Considerazioni

Come già anticipato, le decisioni di progetto intraprese nella prima iterazione hanno reso le modifiche facilmente effettuabili ed hanno inoltre favorito l'introduzione di nuovi requisiti funzionali. Il processo di sviluppo software adottato si basa sul concetto di evoluzione dei requisiti, per cui, in questa nuova iterazione, ci si focalizzerà anche su di una revisione di quanto già fatto in precedenza.

In questa iterazione mi sono dedicato anche al testing di integrazione di tutte le funzionalità implementate. Il testing è stato effettuato facendo riferimento ad alcuni libri di testo che mostrano la corretta esecuzione di alcuni specifici programmi per l'ambiente IJVM con CPU MIC-1.

4.2.2) Logica di Business

L'architettura closed layers adottata per il sottosistema emulatore non è stata modificata, così come non è variato lo stile architettonicale ad interprete utilizzato nel livello di logica applicativa. Grazie alla pratica effettuata nella prima iterazione, sono riuscito a raggiungere una maggiore consapevolezza nell'utilizzo degli strumenti di modellazione a disposizione. Questo ha portato ad alcuni cambiamenti nel System Domain Model: è stato possibile rappresentare le associazioni tra le entità concettuali in maniera più matura e coscientiosa.

Viene di seguito riportato il System Domain Model (“CD_SystemDomainModel”) aggiornato:

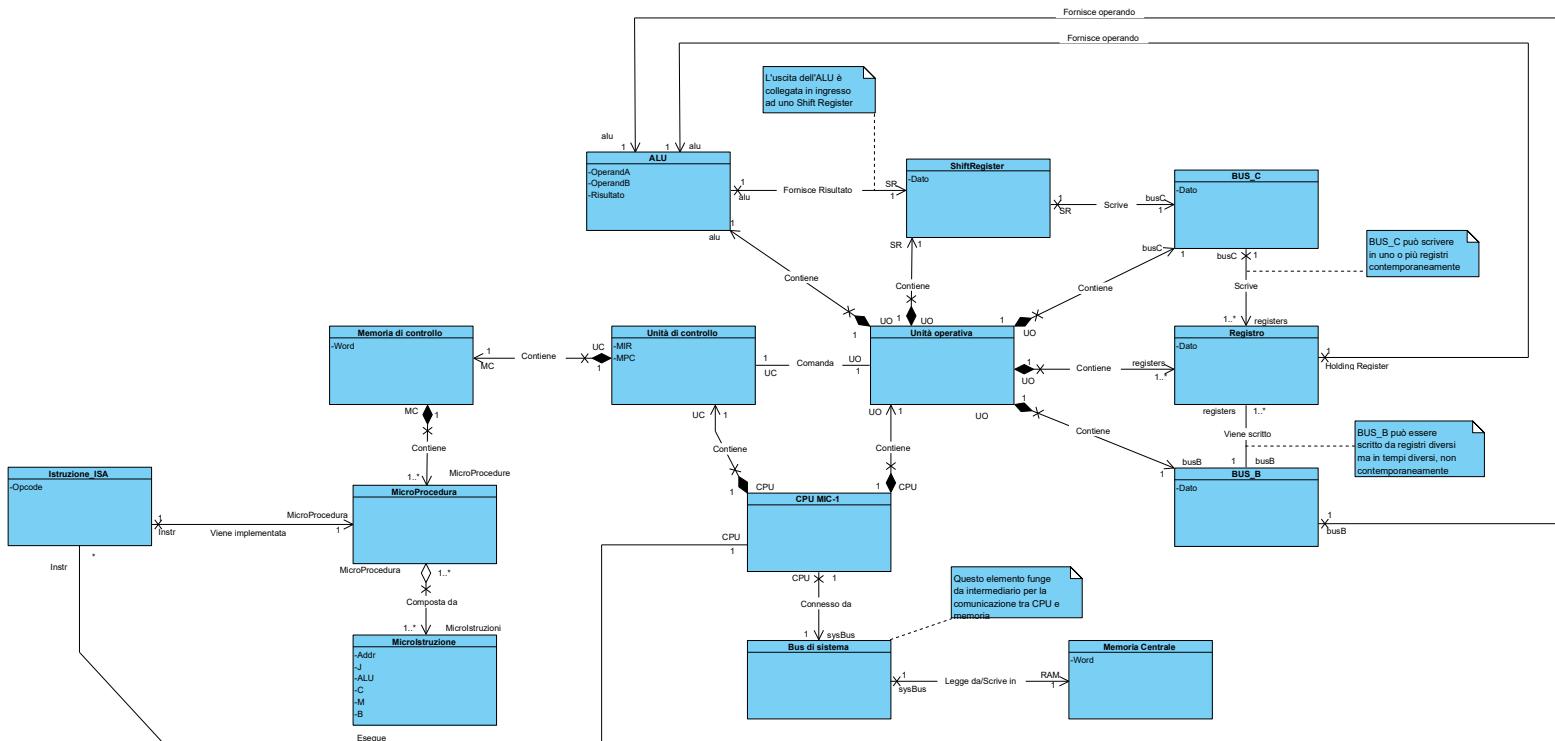


Diagramma 6: CD_SystemDomainModel

Queste stesse modifiche sono state realizzate anche sul diagramma (“CD_Domain_Bound_Controller”) che estende il System Domain Model con le classi “BoundVisualizzazione”, “BoundEmulatore” e “Controller”:

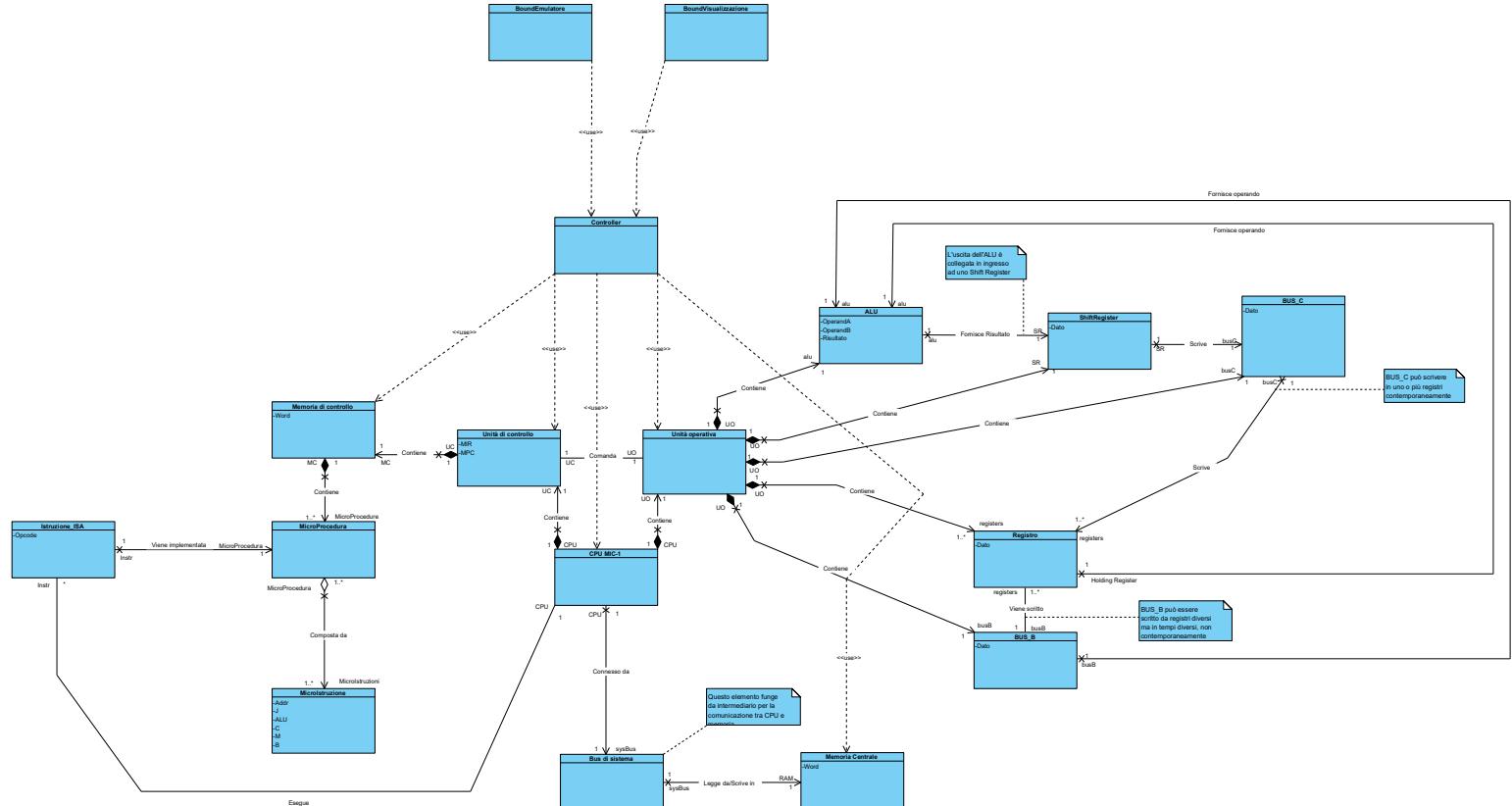


Diagramma 7: CD Domain Bound Controller

4.2.3) GUI

Durante questa iterazione l'attenzione è per la maggior parte rivolta allo sviluppo di due nuovi requisiti funzionali: interfaccia grafica e gestione della persistenza.

Le due classi già menzionate altrove, “**BoundEmulatore**” e “**BoundVisualizzazione**” non corrispondono effettivamente ad entità appartenenti al dominio applicativo di interesse, ma sono comunque importanti perché **servono per delineare i confini dell'applicazione rispetto all'utente che con essa interagisce**: queste due classi rappresentano infatti i porti attraverso i quali viene realizzato l'interfacciamento tra utilizzatore del sistema e sistema software stesso.

“**BoundEmulatore**” si occupa dell'interfacciamento primario con l'utente, e **dovrà** inoltre, come visibile nel diagramma di attività (“**Activity_EseguiProgramma**”) realizzato nella prima iterazione e raffinato nella seconda, **preoccuparsi di creare differenti flussi di esecuzione che operano in maniera concorrente**.

Di seguito viene riportato un aggiornamento di “**Activity_EseguiProgramma**”, in cui **si è tenuto conto delle richieste che l'unità operativa deve effettuare per prelevare le informazioni sull'intero datapath dell'architettura**:

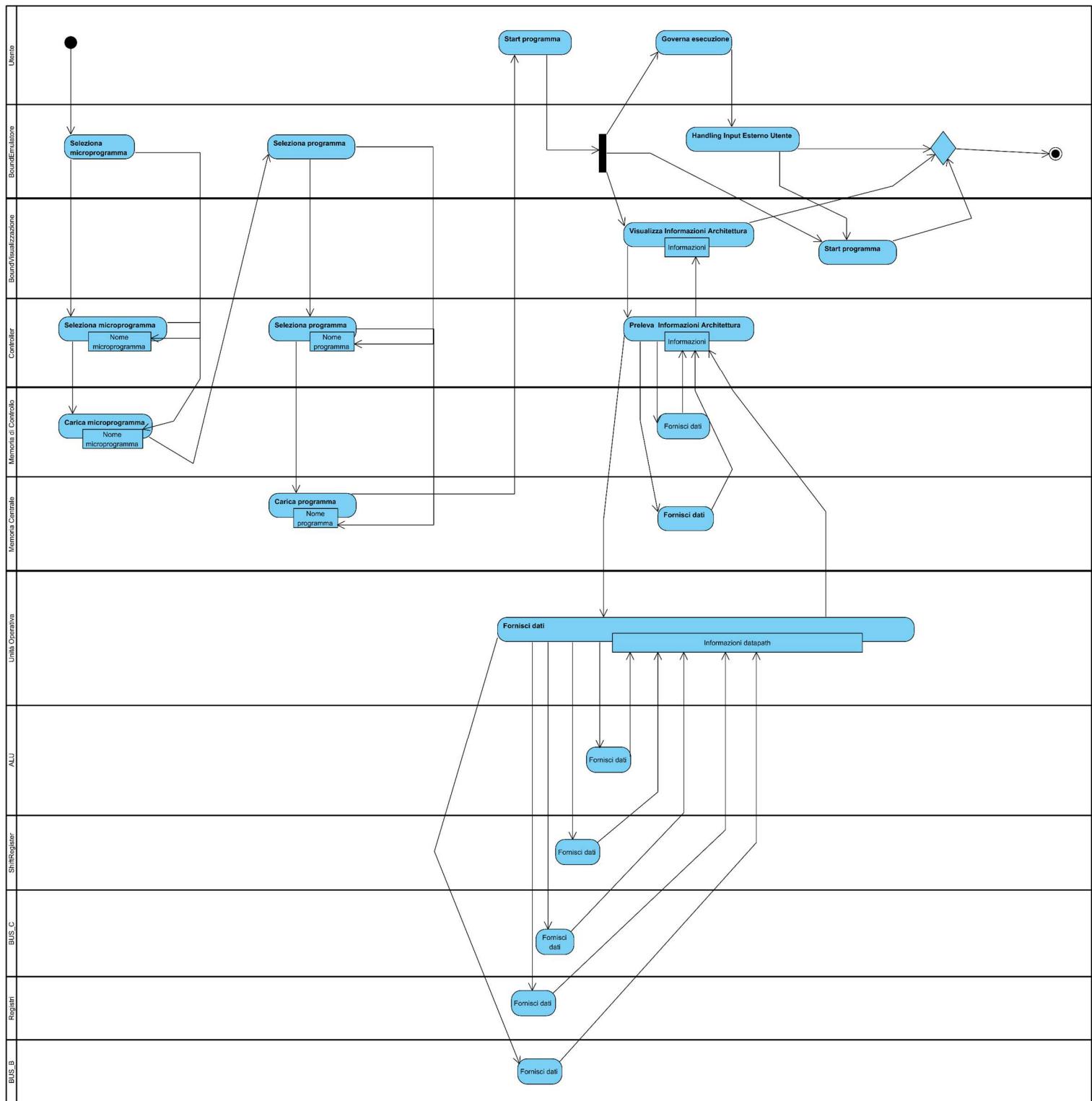


Diagramma 8: Activity_EseguiProgramma

Sempre guardando tale diagramma di attività, è facile intuire che la classe “**BoundEmulatore**” deve gestire la selezione del programma e l’esecuzione di tale programma.

La classe “**BoundVisualizzazione**”, invece, deve occuparsi dell’interfacciamento con lo stato del dispositivo e dovrà quindi aggiornare la vista dello stato fornita all’utente ogni volta che lo stato stesso viene aggiornato.

Quindi, in definitiva, sono presenti **tre flussi di esecuzione concorrenti**:

- **Flusso di logica applicativa:** a questo flusso è demandata la responsabilità di inizializzare ed avviare il processo di emulazione.
- **Flusso di Handling degli input utente:** questo flusso deve essere capace di prelevare gli input forniti dall'utente e, conseguentemente, sulla base dello specifico input catturato, di poter mettere in pausa l'esecuzione, riprenderla, terminarla, riavviarla etc.
- **Flusso di visualizzazione delle informazioni sull'architettura:** questo flusso ha il compito di estrarre lo stato del processo di emulazione e di modificare di conseguenza la vista sullo stesso mostrata all'utente.

La classe “BoundVisualizzazione” può creare dei flussi concorrenti per mostrare lo stato dei singoli componenti.

Sulla base di quanto detto, ossia, tenuto conto della presenza di flussi di esecuzione concorrenti, un diagramma di attività è la scelta più corretta per modellare il funzionamento dell'operazione di visualizzazione di informazioni circa lo stato corrente dell'architettura.

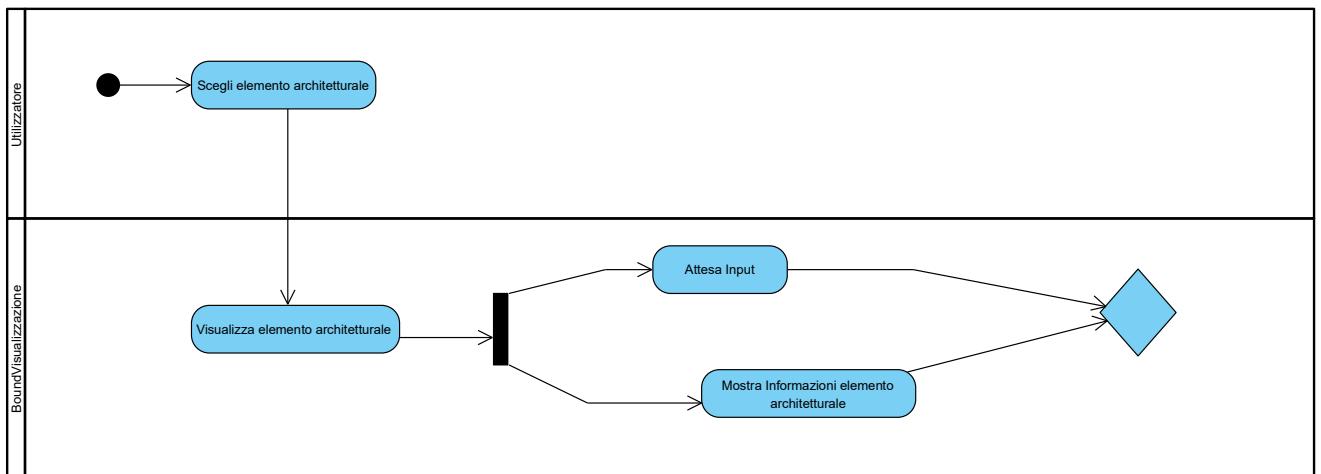


Diagramma 9: Activity_Diag_VisualizzaInformazioniArchitettura

4.2.4) Persistenza: File System

Il secondo requisito funzionale di interesse in questa iterazione riguarda la gestione della persistenza. **È prevista la presenza di una sola fonte di dati persistenti, ossia il File System.** Lo scopo che ci si prefigge è quello di realizzare un meccanismo che consenta di caricare un programma applicativo e/o un microprogramma dal File System. Risulta pertanto necessario andare ad analizzare come un file rappresentativo di un programma eseguibile in ambiente IJVM è strutturato e come invece è strutturato un file contenente un microprogramma da inserire nella memoria di controllo.

Cominciamo con la struttura di un programma eseguibile attraverso l'emulatore. Si consideri il seguente codice:

```
.constant num  
15  
.endconstant  
  
.main  
LDC_W num  
BIPUSH 10  
IADD  
.endmethod
```

.constant	num
	15
.endconstant	
.main	
LDC_W	num
BIPUSH	10
IADD	
.endmethod	

Il funzionamento di questo programma è molto intuitivo:

- Si definisce una costante con valore decimale “15”
- Attraverso l’istruzione **LDC_W** la costante in questione viene prelevata dalla constant pool e inserita in cima allo stack
- Attraverso l’istruzione **BIPUSH** viene caricato sulla cima dello stack il valore decimale “10”
- Attraverso l’istruzione **IADD** vengono prelevati i due valori correntemente in cima allo stack (15 e 10), viene fatta la somma, e il risultato di tale operazione viene introdotto in testa allo stack.

Dopo aver sottoposto questa porzione di codice al processo di compilazione, si ottiene un file di questo tipo:

La memoria centrale nell’architettura hardware che costituisce l’ambiente di esecuzione IJVM è organizzata in word da 32 bit ciascuna. Allo stesso modo è stata “costruita” la memoria centrale all’interno dell’emulatore: **una classe che contiene al suo interno un array di stringhe, ciascuna stringa da 32 caratteri, ciascun carattere rappresentativo di un singolo bit.**

Nel file compilato, i valori decimali preceduti dal simbolo "@" indicano l'indice di memoria centrale a partire dal quale le word che seguono devono essere memorizzate. È sufficiente dunque sviluppare una lettura da file word per word facendo attenzione agli indici, e riempire di conseguenza l'array che rappresenta lo spazio dati della memoria centrale.

Passiamo adesso alla struttura di un microprogramma caricabile all'interno della memoria di controllo. **Si tenga a mente che un microprogramma altro non è che una collezione di microprocedure, e ogni microprocedura è una collezione di opportune microistruzioni.** Ogni microistruzione viene codificata su 36 bit.

Viene di seguito riportato un esempio di microprogramma:

```
# MIC-1 Microprogram

        goto mic1_entry

main:
    PC = PC + 1; fetch; goto (MBR)

nop = 0x00:
    goto main

iadd = 0x65:
    MAR = SP = SP - 1; rd
    H = TOS
    MDR = TOS = MDR + H; wr; goto main

isub = 0x5C:
```

```
MAR = SP = SP - 1; rd  
H = TOS  
MDR = TOS = MDR - H; wr; goto main
```

```
iand = 0x7E:  
    MAR = SP = SP - 1; rd  
    H = TOS  
    MDR = TOS = MDR AND H; wr; goto main
```

```
ior = 0xB6:  
    MAR = SP = SP - 1; rd  
    H = TOS  
    MDR = TOS = MDR OR H; wr; goto main
```

```
dup = 0x57:  
    MAR = SP = SP + 1  
    MDR = TOS; wr; goto main
```

```
pop = 0x59:  
    MAR = SP = SP - 1; rd  
    empty  
    TOS = MDR; goto main
```

```
swap = 0x5F:  
    MAR = SP - 1; rd  
    MAR = SP  
    H = MDR; wr  
    MDR = TOS  
    MAR = SP - 1; wr  
    TOS = H; goto main
```

```
bipush = 0x10:  
    SP = MAR = SP + 1  
    PC = PC + 1; fetch  
    MDR = TOS = MBR; wr; goto main
```

```
iload = 0x15:  
    H = LV  
    MAR = MBRU + H; rd  
iload_cont:  
    MAR = SP = SP + 1  
    PC = PC + 1; fetch; wr  
    TOS = MDR; goto main
```

```
istore = 0x36:  
    H = LV  
    MAR = MBRU + H  
istore_cont:
```

```

MDR = TOS; wr
SP = MAR = SP - 1; rd
PC = PC + 1; fetch
TOS = MDR; goto main

wide = 0xCF:
    PC = PC + 1; fetch; goto (MBR OR 0x100)

wide_iload = 0x115:
    PC = PC + 1; fetch
    H = MBRU << 8
    H = MBRU OR H
    MAR = LV + H; rd; goto iload_cont

wide_istore = 0x136:
    PC = PC + 1; fetch
    H = MBRU << 8
    H = MBRU OR H
    MAR = LV + H; goto istore_cont

ldc_w = 0x20:
    PC = PC + 1; fetch
    H = MBRU << 8
    H = MBRU OR H
    MAR = H + CPP; rd; goto iload_cont

iinc = 0x84:
    H = LV
    MAR = MBRU + H; rd
    PC = PC + 1; fetch
    H = MDR
    PC = PC + 1; fetch
    MDR = MBR + H; wr; goto main

ijvm_goto = 0xA7:
    OPC = PC - 1
goto_cont:
    PC = PC + 1; fetch
    H = MBR << 8
    H = MBRU OR H
    PC = OPC + H; fetch
    goto main

iflt = 0x9D:
    MAR = SP = SP - 1; rd
    OPC = TOS
    TOS = MDR
    N = OPC; if (N) goto T; else goto F

```

```

ifeq = 0x99:
    MAR = SP = SP - 1; rd
    OPC = TOS
    TOS = MDR
    Z = OPC; if (Z) goto T; else goto F

if_icmpeq = 0xA1:
    MAR = SP = SP - 1; rd
    MAR = SP = SP - 1
    H = MDR; rd
    OPC = TOS
    TOS = MDR
    Z = OPC - H; if (Z) goto T; else goto F

T:
    OPC = PC - 1; fetch; goto goto_cont

F:
    PC = PC + 1
    PC = PC + 1; fetch
    goto main

invokevirtual = 0xB9:
    PC = PC + 1; fetch
    H = MBRU << 8
    H = MBRU OR H

mic1_entry:
    MAR = CPP + H; rd
    OPC = PC + 1
    PC = MDR; fetch
    PC = PC + 1; fetch
    H = MBRU << 8
    H = MBRU OR H
    PC = PC + 1; fetch
    TOS = SP - H
    TOS = MAR = TOS + 1
    PC = PC + 1; fetch
    H = MBRU << 8
    H = MBRU OR H
    MDR = SP + H + 1; wr
    MAR = SP = MDR
    MDR = OPC; wr
    MAR = SP = SP + 1
    MDR = LV; wr
    PC = PC + 1; fetch
    LV = TOS; goto main

```

```
ireturn = 0xAD:  
    MAR = SP = LV; rd  
    empty  
    LV = MAR = MDR; rd  
    MAR = LV + 1  
    PC = MDR; rd; fetch  
    MAR = SP  
    LV = MDR  
    MDR = TOS; wr  
    Z = PC - 1; if (Z) goto mic1_exit; else goto main
```

mic1_exit:
halt

Ad ogni istruzione appartenente all'ISA della CPU MIC-1, come visibile, è associato un valore esadecimale che indica, nella memoria di controllo, l'indice della locazione di memoria della prima microistruzione della relativa microprocedura. Inoltre, si osservi che all'interno del microprogramma possono essere utilizzati tutti i registri presenti nel datapath, che non appartengono al modello di programmazione.

Dopo aver sottoposto questo microprogramma al processo di compilazione, si ottiene un file di questo tipo:

Un qualsiasi microprogramma compilato altro non è che un file con 512 linee di testo: una linea per ogni microistruzione. All'interno di tale file, le diverse microistruzioni sono disposte secondo l'ordine nel quale devono essere caricate nella memoria di controllo.

La memoria di controllo è fatta proprio di 512 locazioni ciascuna da 36 bit.

È sufficiente dunque sviluppare una lettura da file linea per linea, ossia microistruzione per microistruzione, e riempire di conseguenza l'array di stringhe (ogni stringa di 36 caratteri, ciascun carattere rappresentativo di un bit) che rappresenta lo spazio dati della memoria di controllo.

4.2.5) Gestione lista programmi e lista microprogrammi

Altro aspetto importante da trattare in questa iterazione, legato alla gestione della persistenza, riguarda lo sviluppo della gestione della lista dei programmi e della lista dei microprogrammi. **Sul file system, saranno presenti due file di tipo “contenitore”: un primo file mantiene l’insieme dei programmi appartenenti ad una lista che viene mostrata all’utente, un secondo file mantiene l’insieme dei microprogrammi appartenenti ad una seconda lista che viene mostrata all’utente.** Su tali file di tipo “contenitore”, l’utilizzatore può effettuare le classiche operazioni **CRUD**, ossia **Create, Read, Update, Delete.** La struttura dei due files appena menzionati viene sviscerata in una sezione apposita della documentazione.

5) Progettazione funzionale e non-funzionale

Dopo aver individuato e analizzato i requisiti del prodotto, **nella fase di design le scelte progettuali effettuate hanno lo scopo di soddisfare i requisiti funzionali e non-funzionali**. Questa sezione, come già fatto con la sezione di “Analisi dei requisiti”, viene suddivisa in differenti sottosezioni, una per ogni iterazione svolta.

5.1) Prima iterazione

5.1.1) Architettura logica del componente emulatore

Per costruire l'emulatore si è deciso di fare uso dello stile architettonico a livelli. Inoltre, data la natura di questo componente, ossia dato che lo scopo di questo sottosistema è quello di interpretare istruzioni dell'ISA del MIC-1 e quindi le microistruzioni contenute nelle relative microprecedure, **nel livello di “logica di business” o “logica applicativa” viene adoperato lo stile architettonico ad interprete, trascurando però lo strato di interfacciamento verso l'esterno:** questo perché le istruzioni ISA da eseguire e quindi le microistruzioni da eseguire, vengono di volta in volta determinate attraverso, rispettivamente, le modifiche che il Program Counter e il Micro Program Counter subiscono durante l'esecuzione di un programma.

Di seguito viene riportato il diagramma “CD_Architettura_Emulatore”:

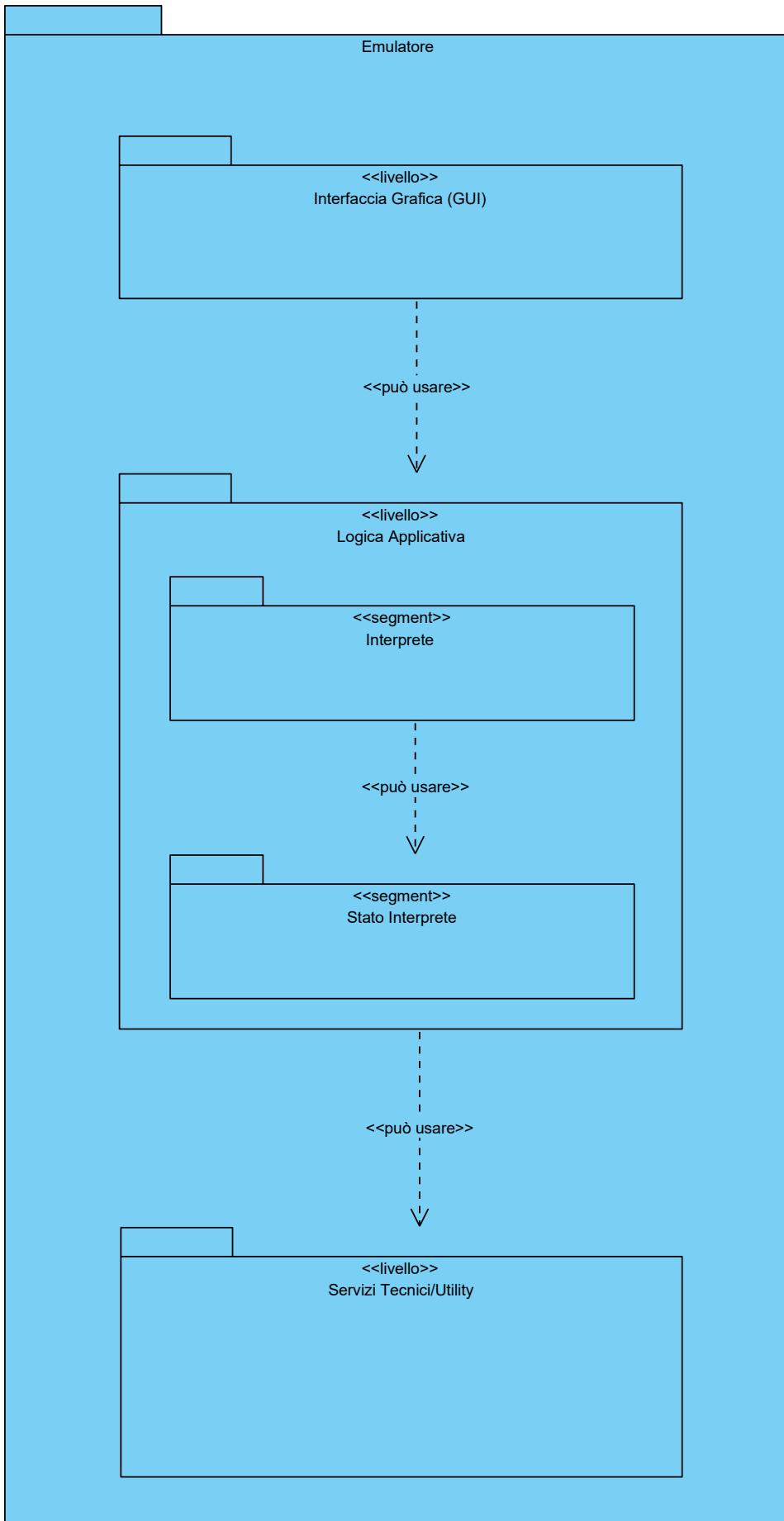


Diagramma 10: CD_Architettura_Emulatore

Ritengo che le motivazioni che mi hanno portato a scegliere lo stile architettonale a livelli (“**Closed layers**”) siano abbastanza valide:

- **Un’architettura software a livelli è facilmente modificabile e dunque evolvibile;** le responsabilità associate ai diversi livelli sono ben distinte e, nel caso in cui si verificassero delle richieste di modifica o cambiamento relative a requisiti già sviluppati e costruiti, l’integrazione e il soddisfacimento di queste richieste non provocherebbe grandi difficoltà: questo è vero perché esiste un’indipendenza tra i diversi livelli dell’architettura e, inoltre, il flusso di dati e il flusso di controllo possono essere facilmente tracciati.
- Sin dall’inizio ho specificato l’importanza di avere una architettura che sia facilmente modificabile. Questo perché, **diverse parti del sistema software che si sta realizzando, verranno analizzate nel dettaglio più avanti, e quindi diventa fondamentale riuscire a disaccoppiare tra loro i vari moduli.**

Detto questo, vale quanto segue:

- **L’interazione con l’utente viene gestita mediante una GUI resa disponibile dal package “Interfaccia Grafica”**
- **La logica applicativa, per realizzare la quale si farà uso dello stile Interpreter, è messa a disposizione dal package “Logica Applicativa” ed in esso è contenuta.**
- **È infine presente un livello – quello più in basso – denominato “Servizi Tecnici/Utility”. Esso espone un insieme di servizi che permettono la gestione di una lista di programmi e di una lista di microprogrammi nel file system del sistema operativo su cui l’applicazione è eseguita.**

5.1.2) Raffinamento della dinamica del caso d'uso EseguìProgramma

Il passo successivo è stato quello di arricchire l'architettura logica dell'emulatore con l'introduzione di nuove classi software, tenendo a mente, per il livello di dominio applicativo, le classi già presentate nei due diagrammi delle classi mostrati in precedenza. **Raffinare la dinamica del caso d'uso considerato in questa iterazione, ossia "EseguìProgramma"**, rappresenta il passo necessario per modellare correttamente, in termini di design e scelte progettuali, il **class diagram relativo al componente emulatore**. Si è già sottolineato che, per il componente emulatore, si è deciso di adottare uno stile architettonico a livelli. Si è visto inoltre che, per il livello di logica applicativa, si adopera uno stile ad interprete. **Nel particolare caso di questo sistema software, lo stile ad interprete si presta ad essere marcatamente dinamico**: c'è una dipendenza sia dallo stato interno del processore sia dalla particolare microistruzione ottenuta in ingresso. Questa porzione del caso d'uso può essere più facilmente rappresentata attraverso un diagramma degli stati (**"State_Diag_UnitàControllo"**).

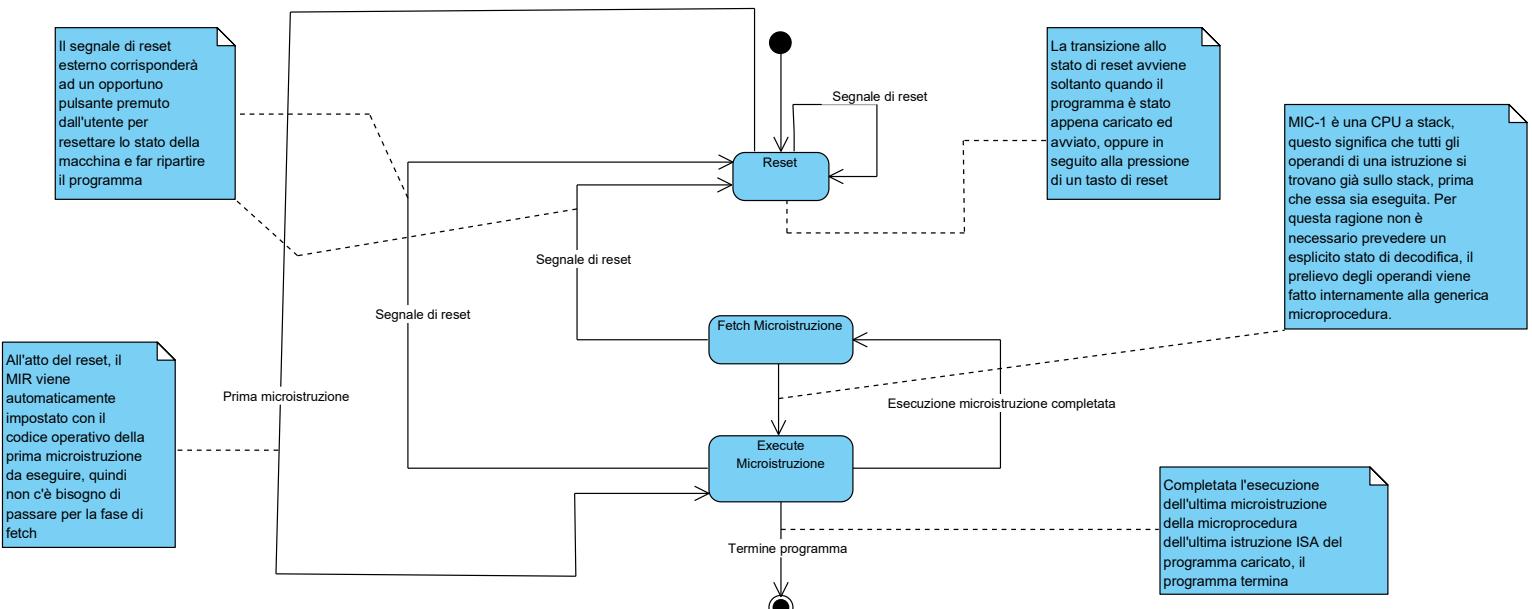


Diagramma 11: State_Diag_UnitàControllo

Questo diagramma degli stati mostra il modo in cui l'unità di controllo si comporta ed evolve il suo stato durante l'esecuzione di un programma precedentemente caricato. Con un tale tipo di diagramma la descrizione del comportamento dell'unità di controllo risulta molto più semplice rispetto al caso in cui si voglia utilizzare un diagramma di sequenza. Le diverse possibili microistruzioni supportate, e dunque le differenti istruzioni ISA disponibili, fanno sì che il comportamento sia molto variabile. Affinché i diagrammi realizzati in fase di design possano effettivamente coadiuvare la parte di implementazione, questi diversi comportamenti dovranno essere descritti in modo indipendente.

Risulta necessario, in tale contesto, utilizzare una programmazione concorrente. Infatti, avere un unico flusso di controllo che si occupa di eseguire il programma porterebbe ad una violazione di alcuni dei requisiti non-funzionali menzionati nel documento di specifiche supplementari. Con un singolo thread l'interfaccia risulterebbe “non reattiva”, potrebbe da un momento all'altro smettere di rispondere per un intervallo di tempo indefinito.

Di seguito viene riportato un diagramma di attività (“**Activity_EseguiProgramma**”).

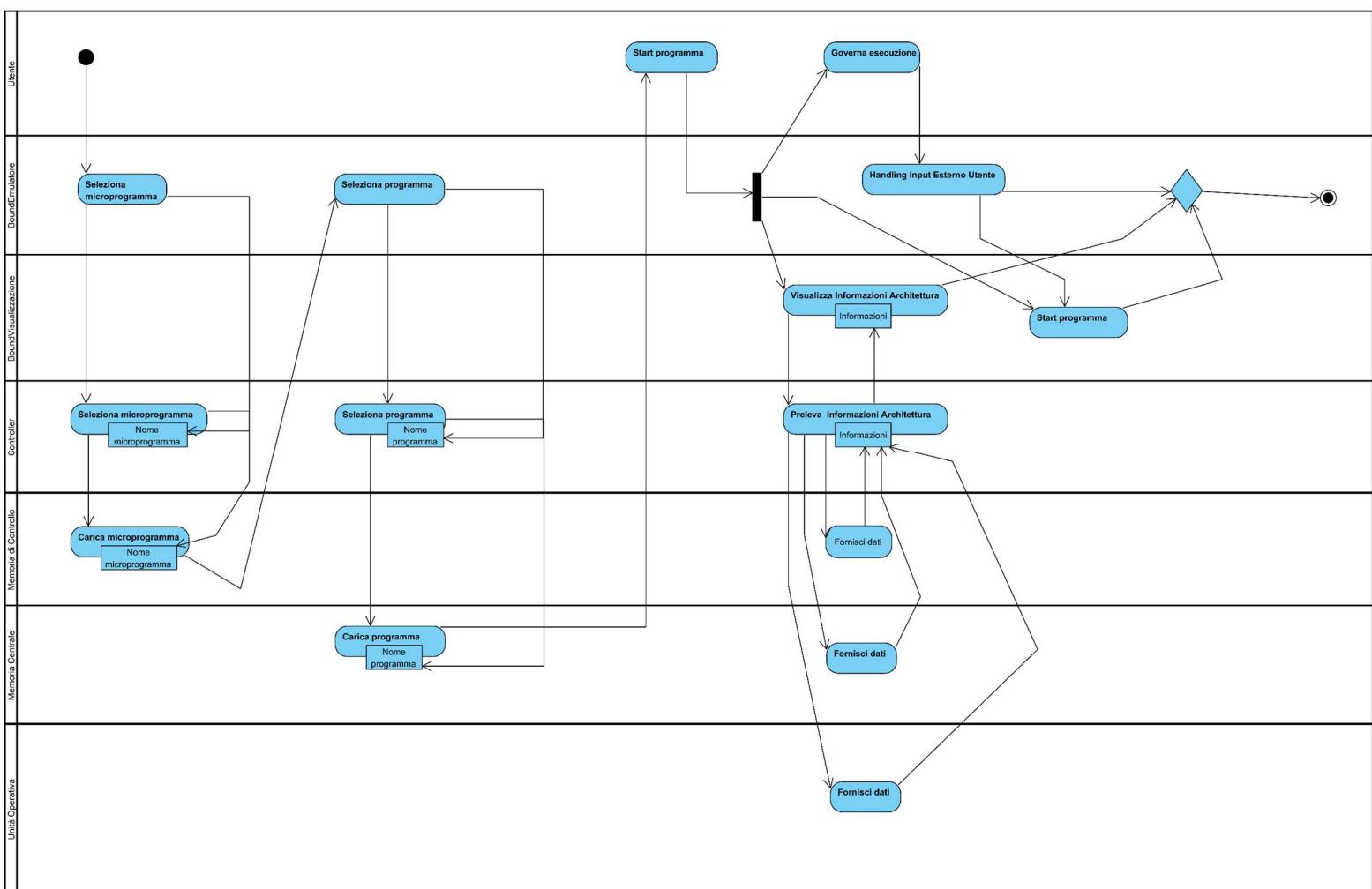


Diagramma 12: Activity_EseguìProgramma

Attraverso un diagramma di attività possiamo visualizzare in maniera chiara la presenza di flussi di controllo che procedono in maniera concorrente, raggiungendo dunque una progettazione più vicina all’implementazione. In tale diagramma le diverse attività sono rappresentate in maniera “sintetica” e si possono osservare accenni ad altri requisiti funzionali di cui il sistema software sarà dotato, come, ad esempio, l’handling degli input generati dall’utente e la visualizzazione delle informazioni sullo stato attuale del dispositivo emulato.

Adesso è possibile passare al class diagram di dettaglio per il sottosistema “Emulatore”.

Vengono utilizzati i seguenti design pattern:

- **Facade pattern:** grazie a questo design pattern si riesce a garantire l’esistenza di un solo punto di accesso ad un livello mediante un unico oggetto che fornisce l’interfaccia per lo specifico livello considerato.
- **State pattern:** questo design pattern viene adoperato per descrivere i diversi stati in cui l’unità di controllo può transitare.

- **Singleton pattern:** questo design pattern viene utilizzato in maniera ricorrente. Il particolare dominio applicativo in cui il sistema software si trova fa sì che per diverse classi in gioco il numero di istanze necessarie sia pari proprio ad 1. Inoltre, grazie al fatto che con il singleton pattern l'accesso ad un oggetto viene fatto attraverso una istanza statica, il generico elemento su cui questo pattern viene adoperato è in grado di mantenere il suo stato per tutta la durata dell'esecuzione dell'applicazione.

Di seguito vengono riportate due porzioni del class diagram precedentemente menzionato (**“Class_Diag_Emulator”**).

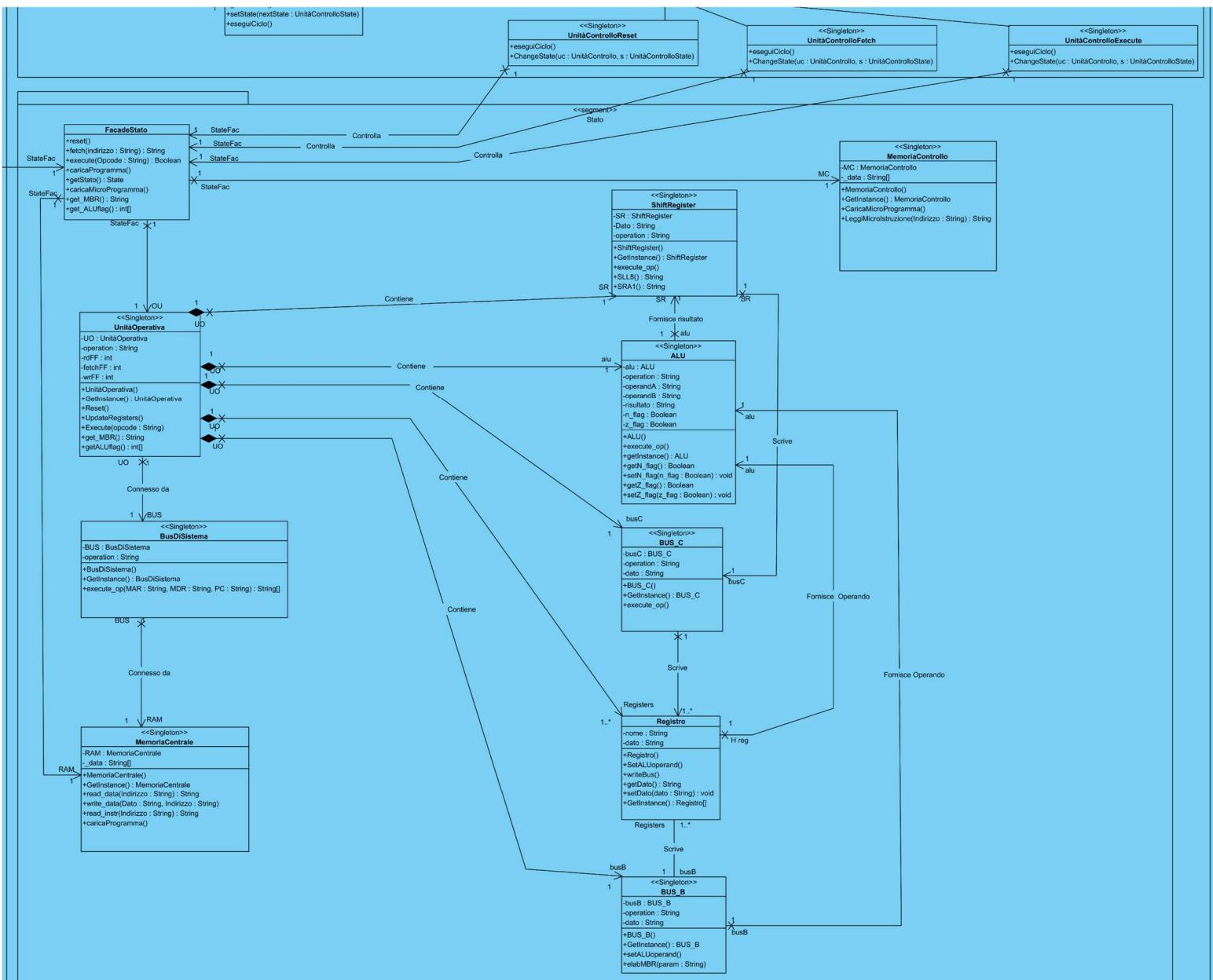


Diagramma 13: Porzione di Class_Diag_Emulator

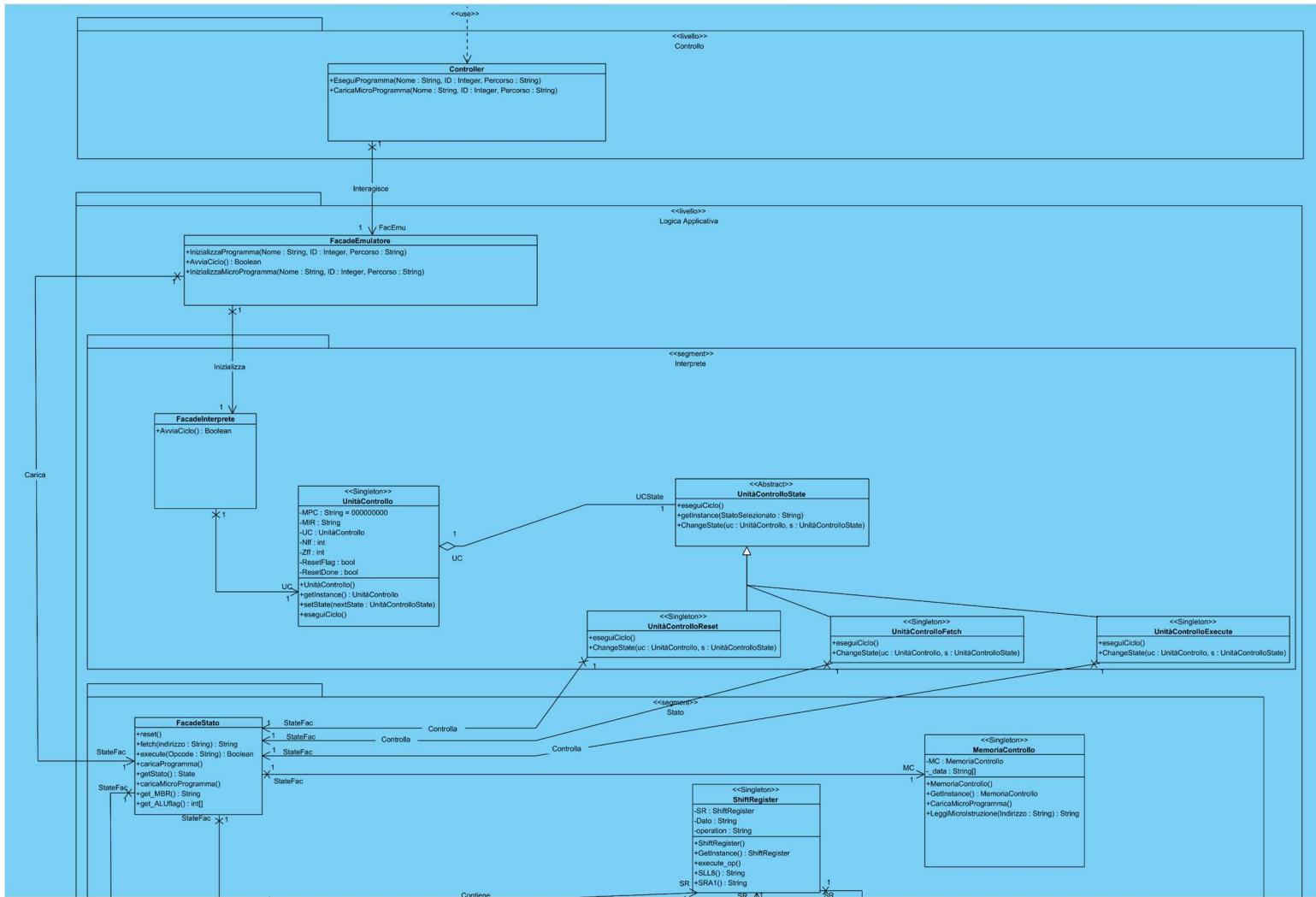


Diagramma 14: Porzione di Class_Diag_Emulatore

Come visibile da queste due immagini, sono stati assegnati degli stereotipi ad alcune classi secondo quanto ritenuto opportuno. **Le classi presentate in questo diagramma sono molto vicine all'implementazione, sono utili per aiutare il programmatore nella fase di implementazione, e si differenziano dalle classi del System Domain Model: in quest'ultimo caso si tratta di classi puramente concettuali, che hanno lo scopo di catturare le entità del dominio.**

Infine, si osservi che è presente un **livello denominato “Controllo”**: esso possiede le responsabilità relative alla gestione degli input dell’utente inviati dall’interfaccia grafica verso gli altri livelli dell’applicazione.

Vengono di seguito mostrati alcuni dei più significativi sequence diagram prodotti in questa prima iterazione per realizzare il modello dinamico del sistema software:

“Seq_Diag_reset_eseguiCiclo”:

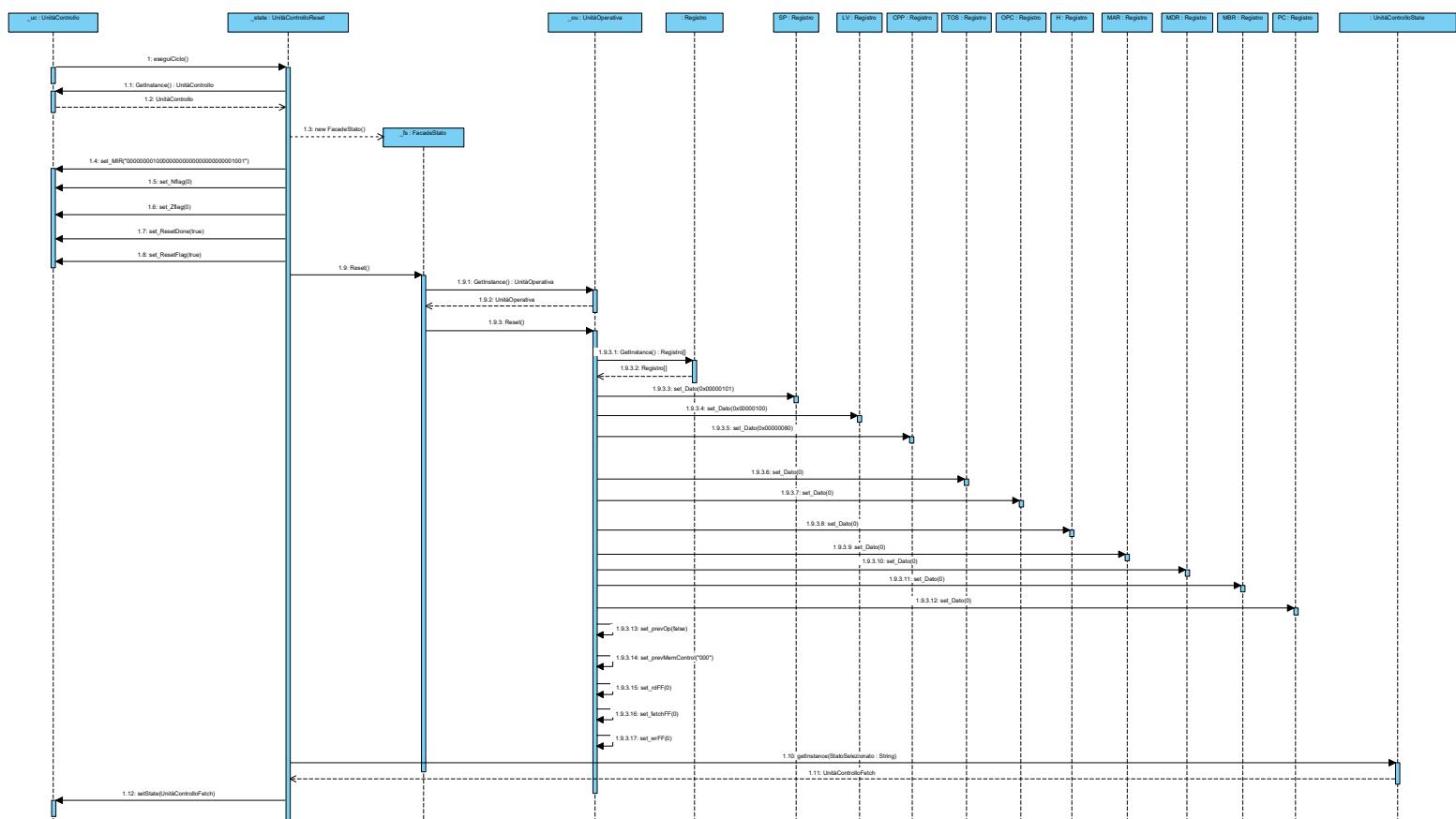


Diagramma 15: Seq_Diag_reset_eseguiCiclo

“Seq_Diag_fetch_eseguiCiclo”:

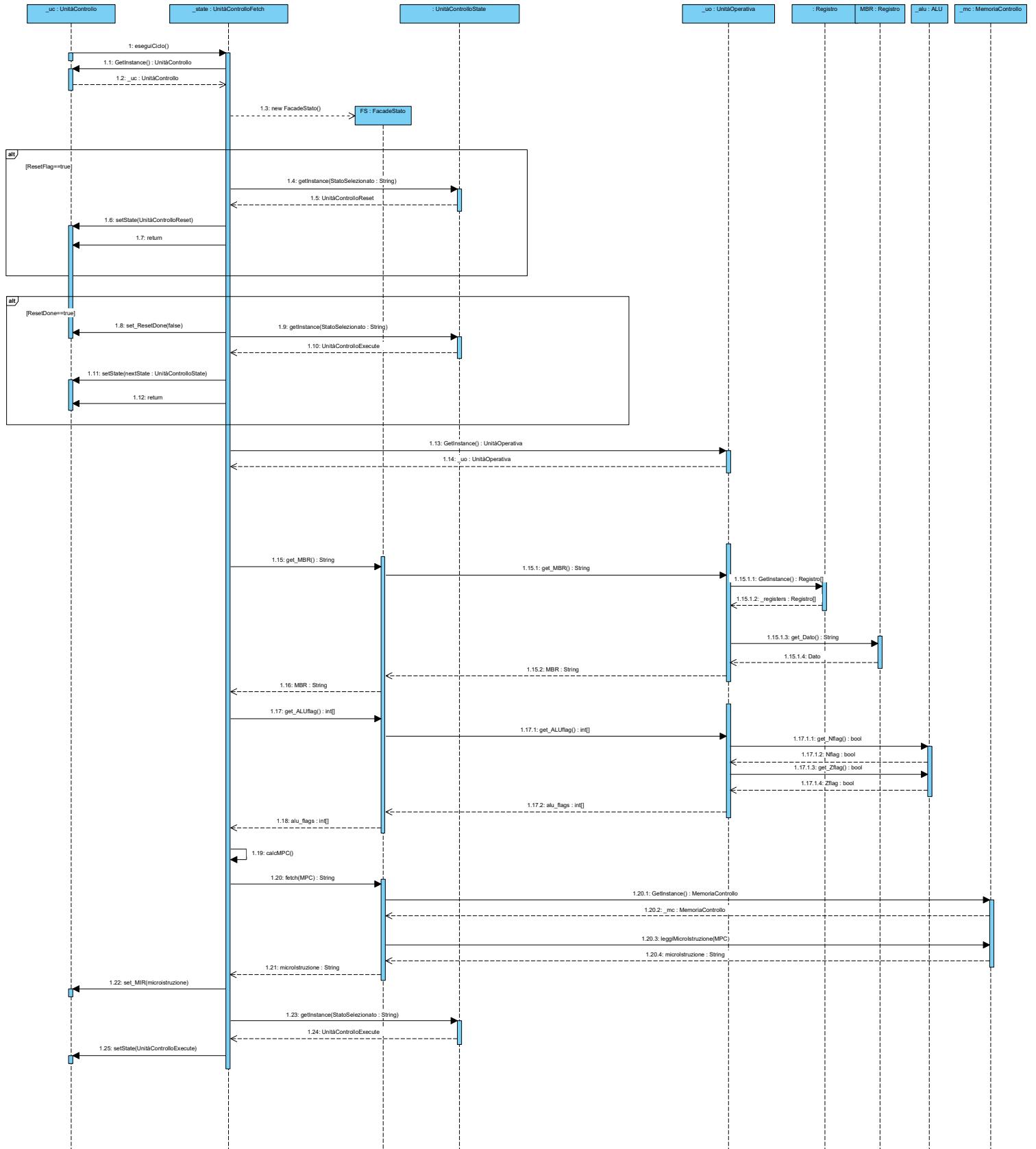


Diagramma 16: Seq_Diag_fetch_eseguiCiclo

“Seq_Diag_execute_eseguiCiclo”:

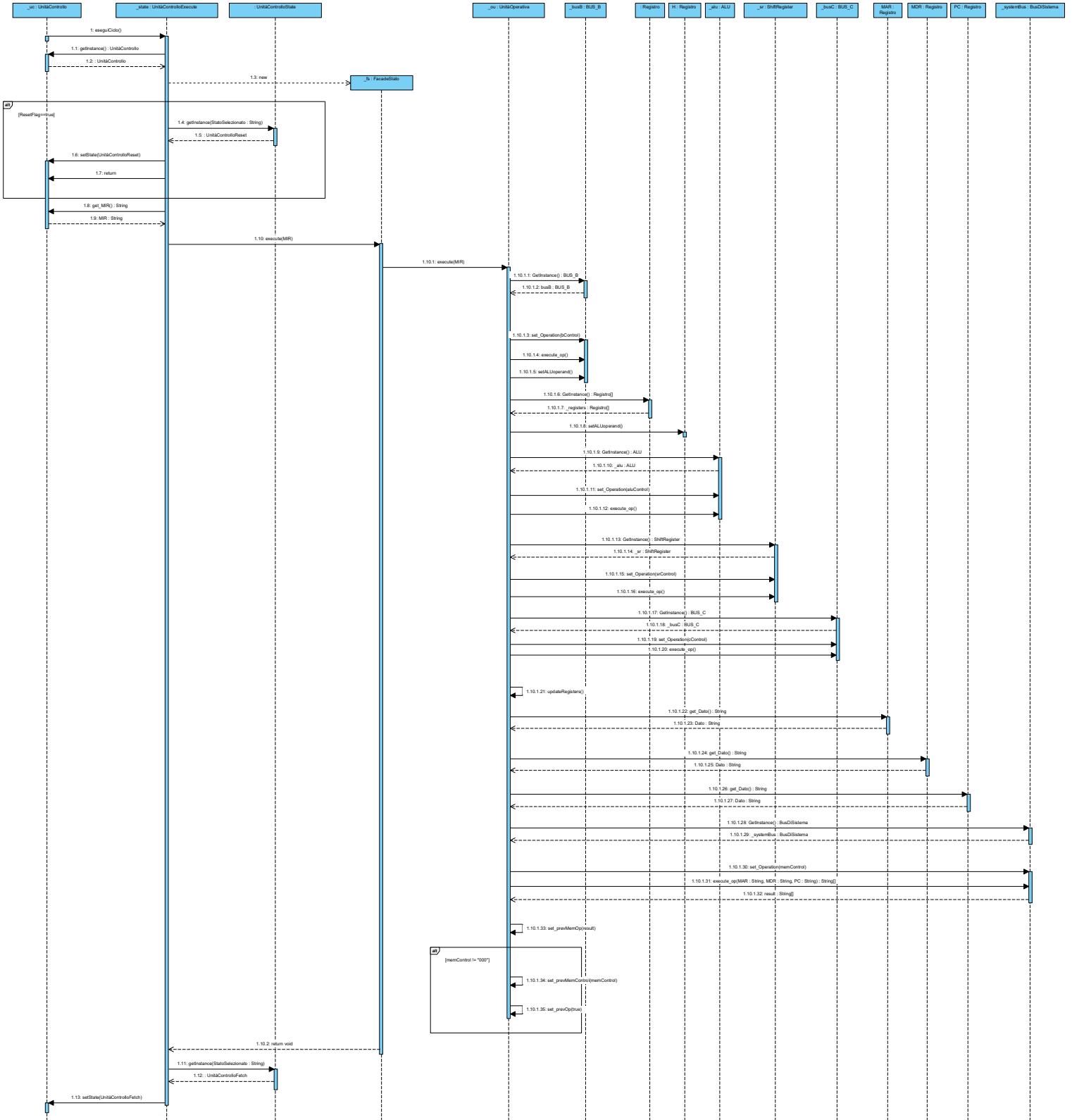


Diagramma 17: Seq_Diag_execute_eseguiCiclo

Questi tre diagrammi di sequenza mostrano il comportamento dell'unità di controllo nei suoi tre rispettivi possibili stati: "Reset", "Fetch Microlistruzione", "Execute Microlistruzione".

Sono stati inoltre realizzati appositi diagrammi di sequenza per descrivere in maniera dettagliata il comportamento dei diversi elementi che costituiscono il datapath del processore.

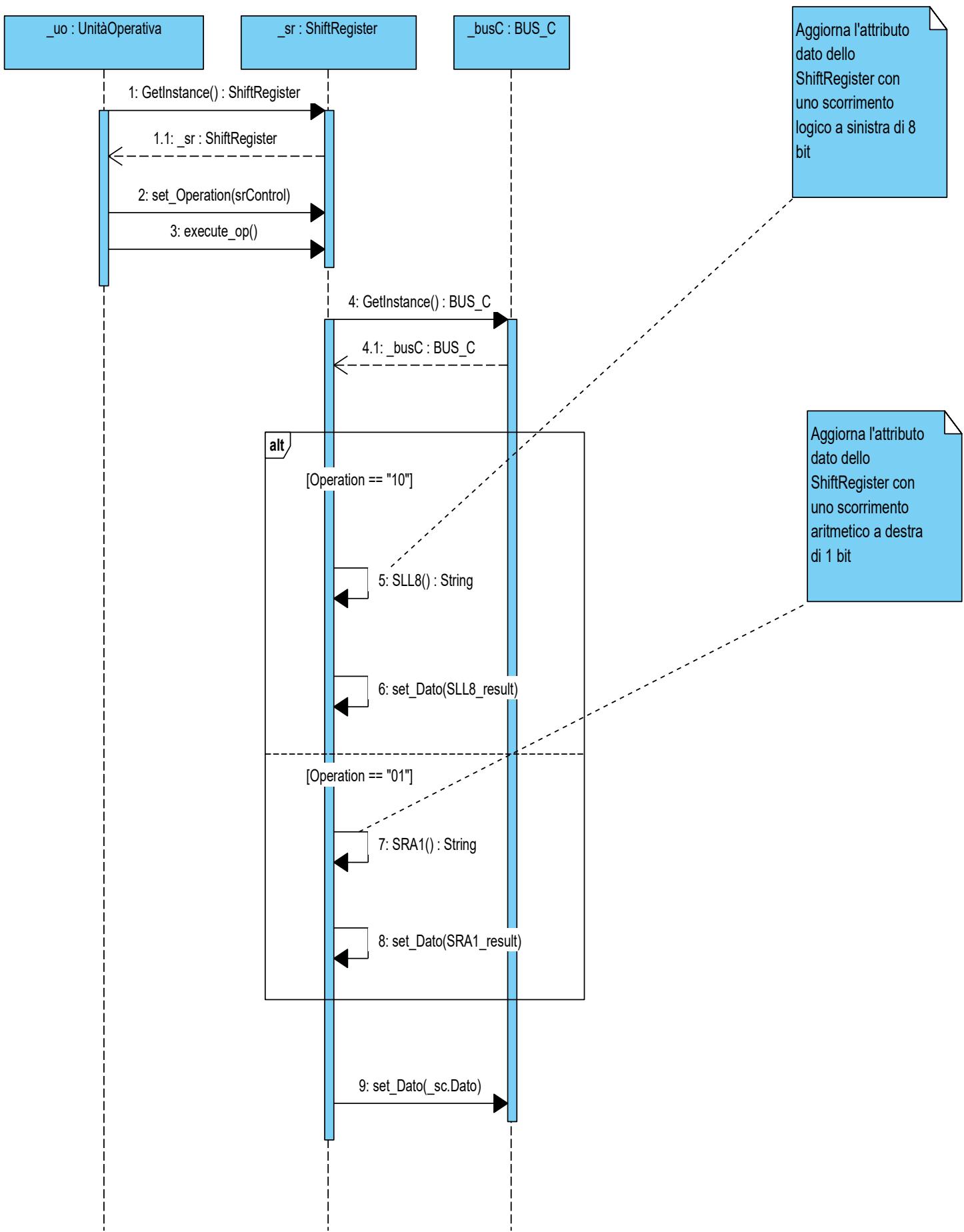


Diagramma 18: Seq_Diag_ShiftReg_executeop

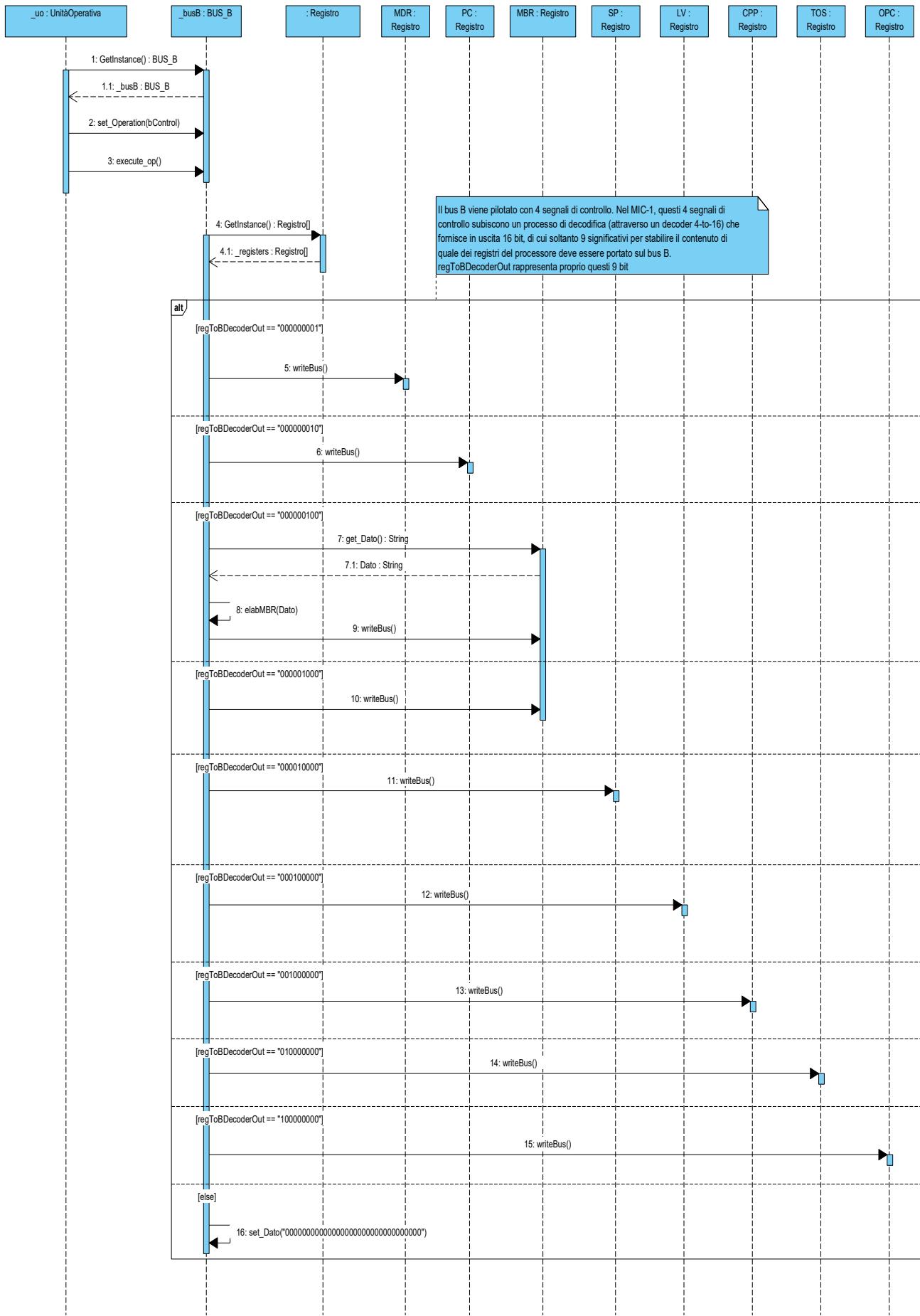
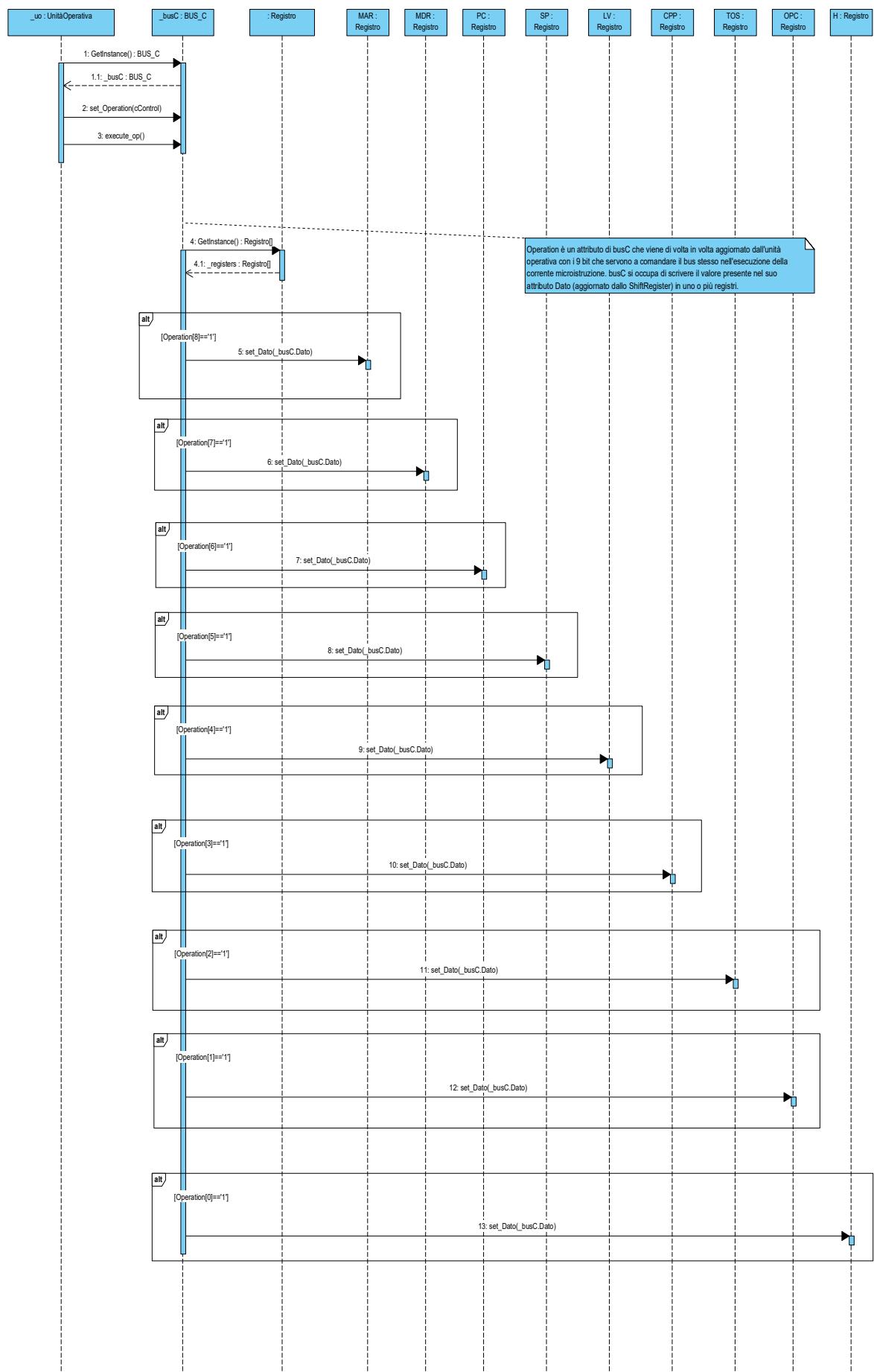


Diagramma 19: Seq_Diag_busB_executeop



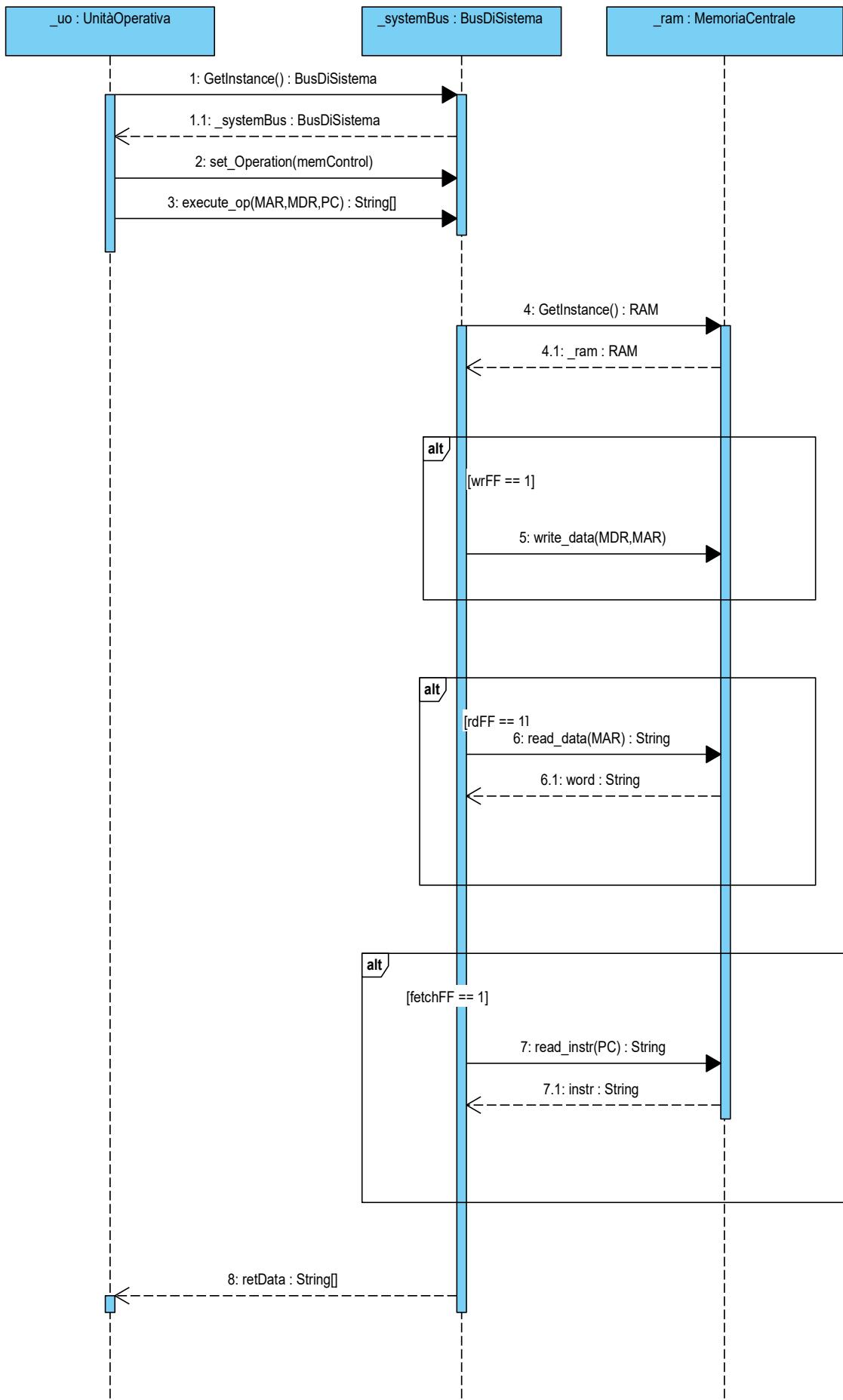


Diagramma 21: Seq_Diag_systembus_executeop

5.1.3) Vista “Componenti e Connettori”

Con la sola vista modulare non riusciamo ad esprimere informazioni significative circa gli elementi che formano il sistema software a tempo di esecuzione. Per questo motivo, **bisogna introdurre nella documentazione delle opportune viste e quindi diagrammi, che siano capaci di rappresentare correttamente le entità del sistema a tempo di esecuzione.** Per raggiungere questo risultato, vengono utilizzati i cosiddetti diagrammi dei componenti.

- **Un primo diagramma dei componenti costituisce il “catalogo”: in esso vengono rappresentati i diversi tipi di componenti che figurano nell’applicazione.**
- **Un secondo diagramma dei componenti consentirà invece di visualizzare gli elementi costitutivi del sistema a runtime come istanze dei tipi presenti nel catalogo.**

L’idea di base è che il componente assemblatore dovrebbe pubblicare dei programmi compilati in uno spazio dati al quale ha accesso anche l’emulatore. In generale, anche trascurando l’assemblatore, si vuole che il componente emulatore sia dotato di un proprio repository sul quale mantenere i programmi da eseguire. Analoghe considerazioni valgono per i micropogrammi. Si utilizza dunque uno stile architettonico del tipo “shared data”: ci sono degli opportuni componenti detti **repository** e i componenti del sistema software interagiscono con essi per prelevare dati da una sorgente condivisa.

Di seguito sono riportati i due diagrammi dei componenti precedentemente menzionati (rispettivamente “**Comp_Diag_Catalogo**” e “**Comp_Diag_MIC1-SYS**”).

“**Comp_Diag_Catalogo**”:

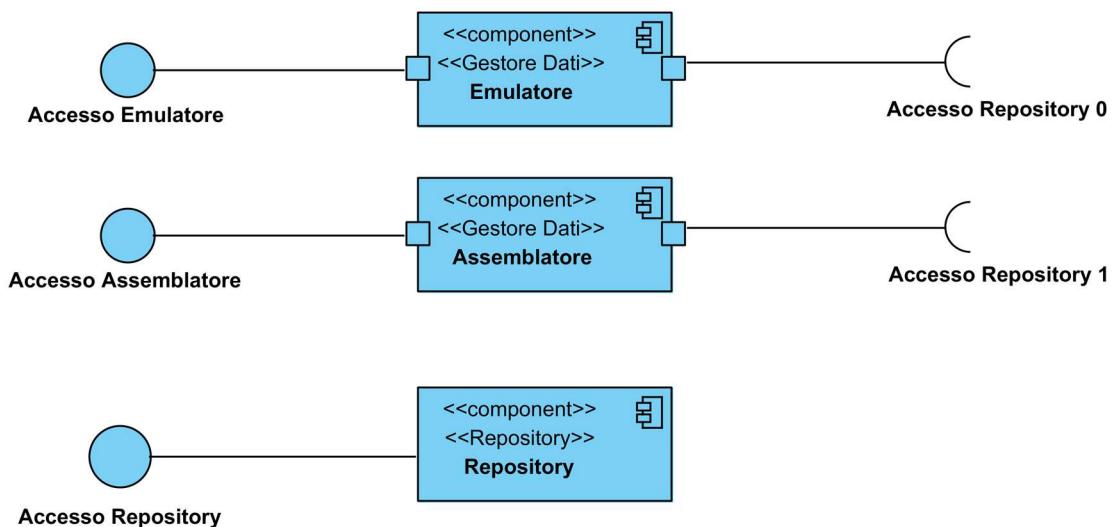


Diagramma 22: *Comp_Diag_Catalogo*

“**Comp_Diag_MIC1-SYS**”:

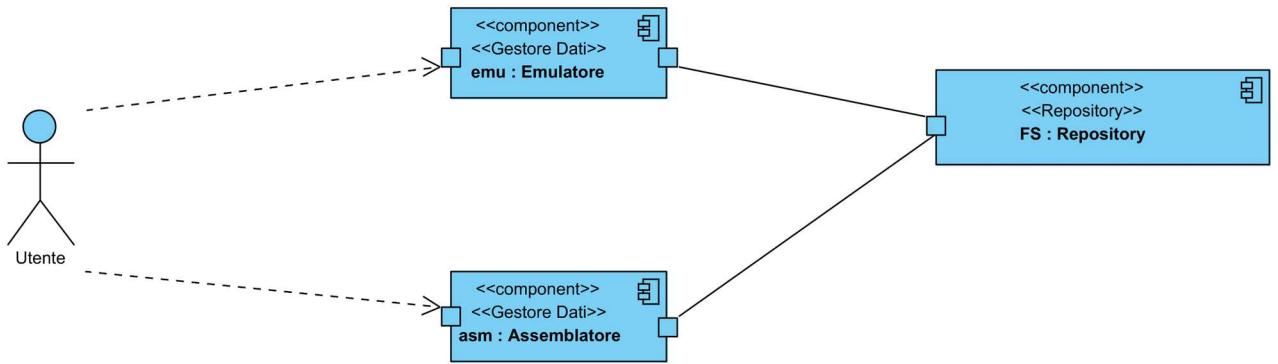


Diagramma 23: Comp_Diag_MIC1-SYS

FS sta per “File System”. Ho ipotizzato che ci sia soltanto il file system come sorgente possibile di dati. Eventualmente, magari in futuro, sarà possibile aggiungere una qualche altra fonte persistente di dati, come un database.

5.2) Seconda Iterazione

5.2.1) Architettura logica del componente emulatore

L'architettura logica del componente emulatore è stata modificata apportando migliorie e correzioni agli aspetti statici e dinamici della modellazione già realizzati in precedenza ed è stata estesa introducendo nuove funzionalità. Grazie alle scelte progettuali effettuate e ai design pattern adoperati nella prima iterazione, estendere il sistema è risultato semplice.

Parte Statica - Logica applicativa

Per quanto riguarda la parte statica, il livello di logica applicativa è stato impreziosito e migliorato. In particolare, è stato aggiunto un nuovo stato per l'unità di controllo ed è stata inserita una classe "Stato" nel sottopackage "Stato" appartenente al livello di "Logica Applicativa", utilizzata per mantenere le informazioni sullo stato corrente dell'architettura, informazioni che vengono opportunamente aggiornate ogni volta che viene eseguita una nuova microistruzione.

Del diagramma delle classi "Class_Diag_Emulator" vengono di seguito riportate soltanto quelle parti che evidenziano le modifiche effettuate in questo contesto.

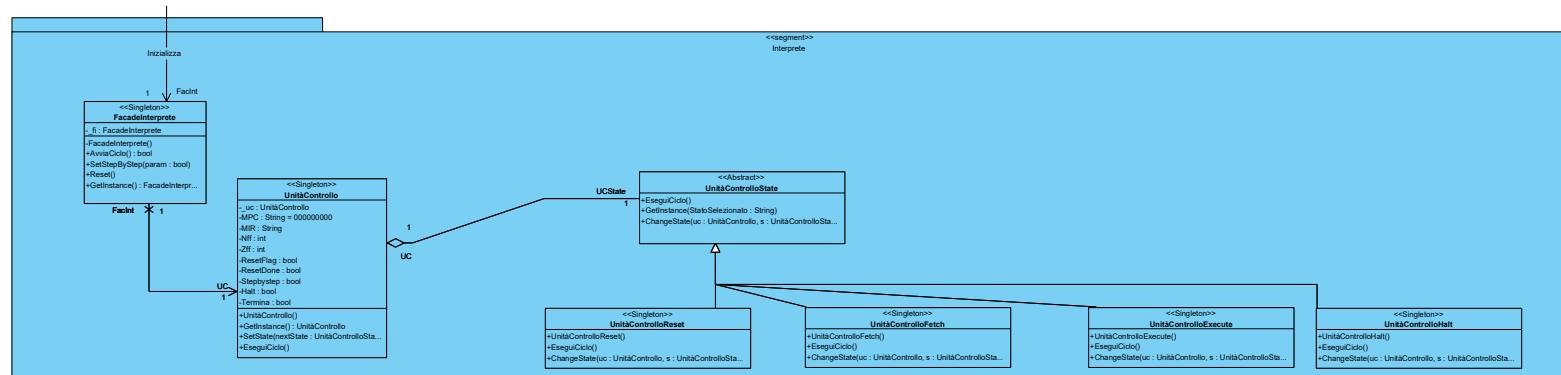


Diagramma 24: Porzione Class_Diag_Emulator

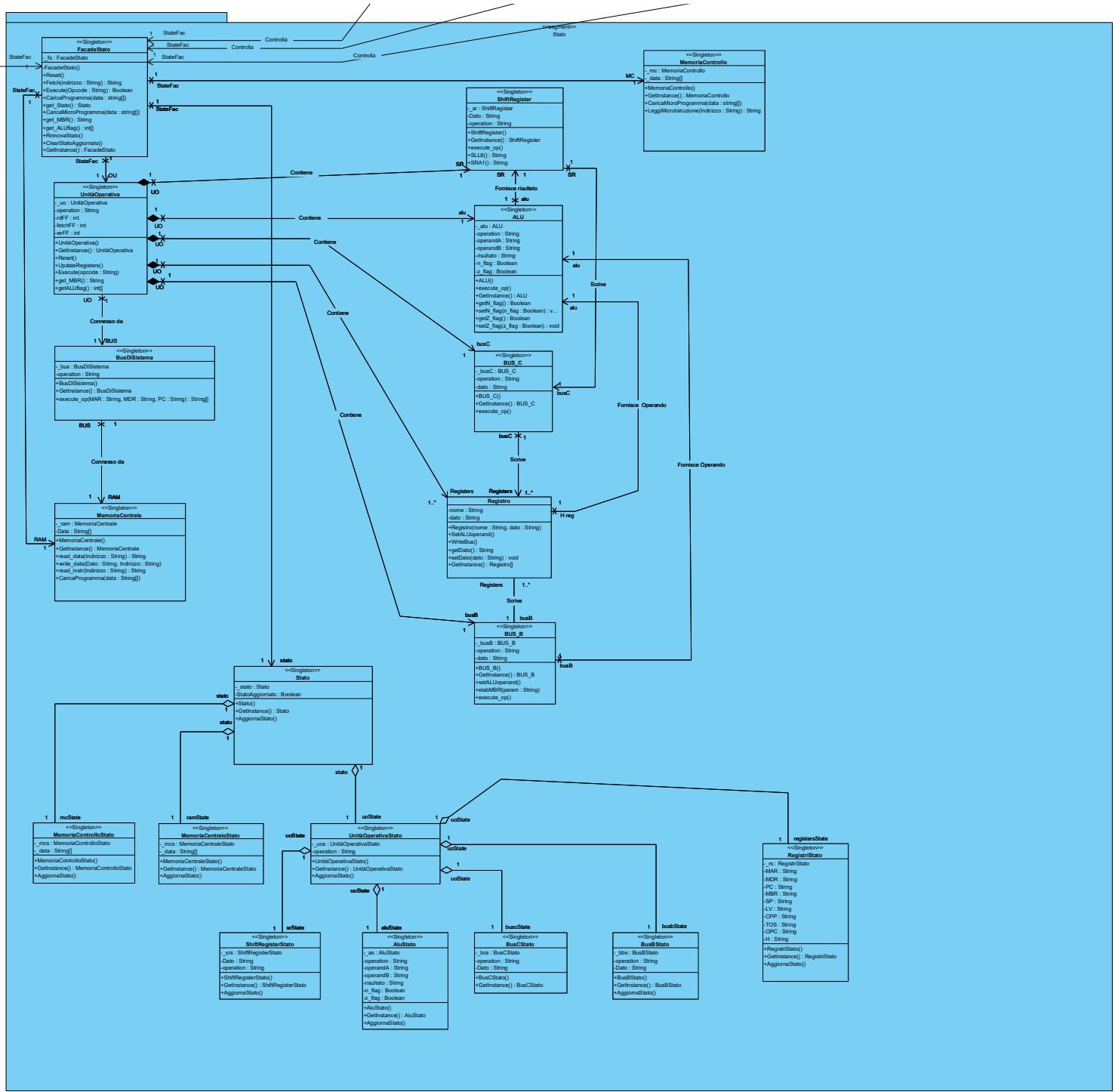


Diagramma 25: Porzione Class_Diag_Emulatore

Si osservi inoltre che a tutte le classi “Facade” è stato applicato il Design Pattern “Singleton”: questa scelta è coerente con il fatto che di ogni classe prodotta mediante il Design Pattern “Facade” deve esserci una ed una sola istanza.

5.2.2) Raffinamento della dinamica del caso d'uso EseguìProgramma

Le modifiche presentate per la parte statica della modellazione hanno ovviamente conseguenze sulla parte dinamica del sistema. Il diagramma degli stati relativo all'unità di controllo è stato modificato con l'aggiunta di un nuovo stato, denominato "Halt".

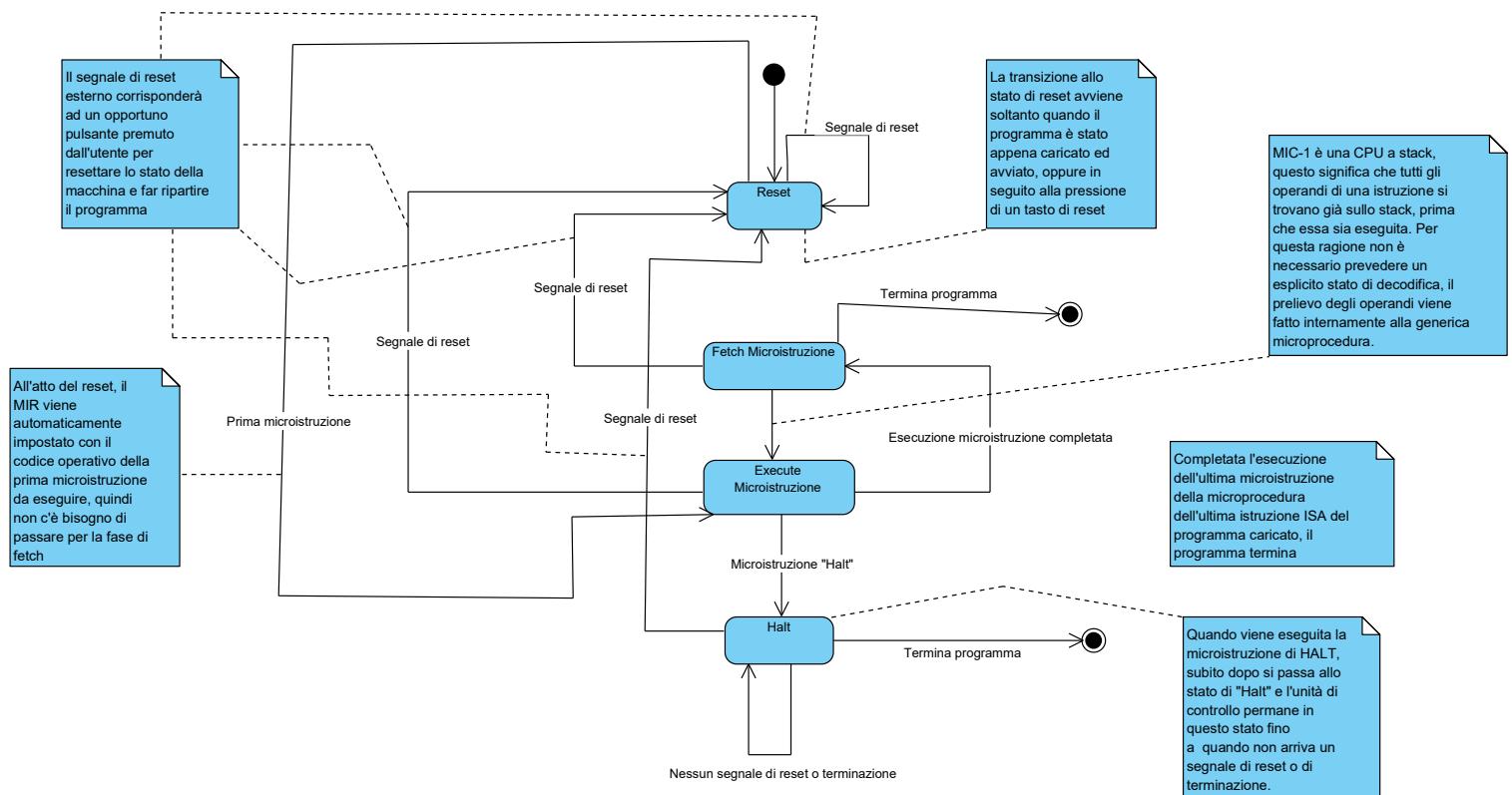


Diagramma 26: State_Diag_UnitàControllo

Il punto di uscita può essere preso o nello stato di "Fetch" o nello stato di "Halt", supposto che siano verificate le condizioni. **Con le modifiche effettuate a questo diagramma di stato, si è reso necessario aggiornare i sequence diagram relativi ai due stati di "Fetch" e "Execute" e introdurre un nuovo diagramma di sequenza relativo proprio al nuovo stato "Halt".**

“Seq_Diag_halt_eseguiCiclo”:

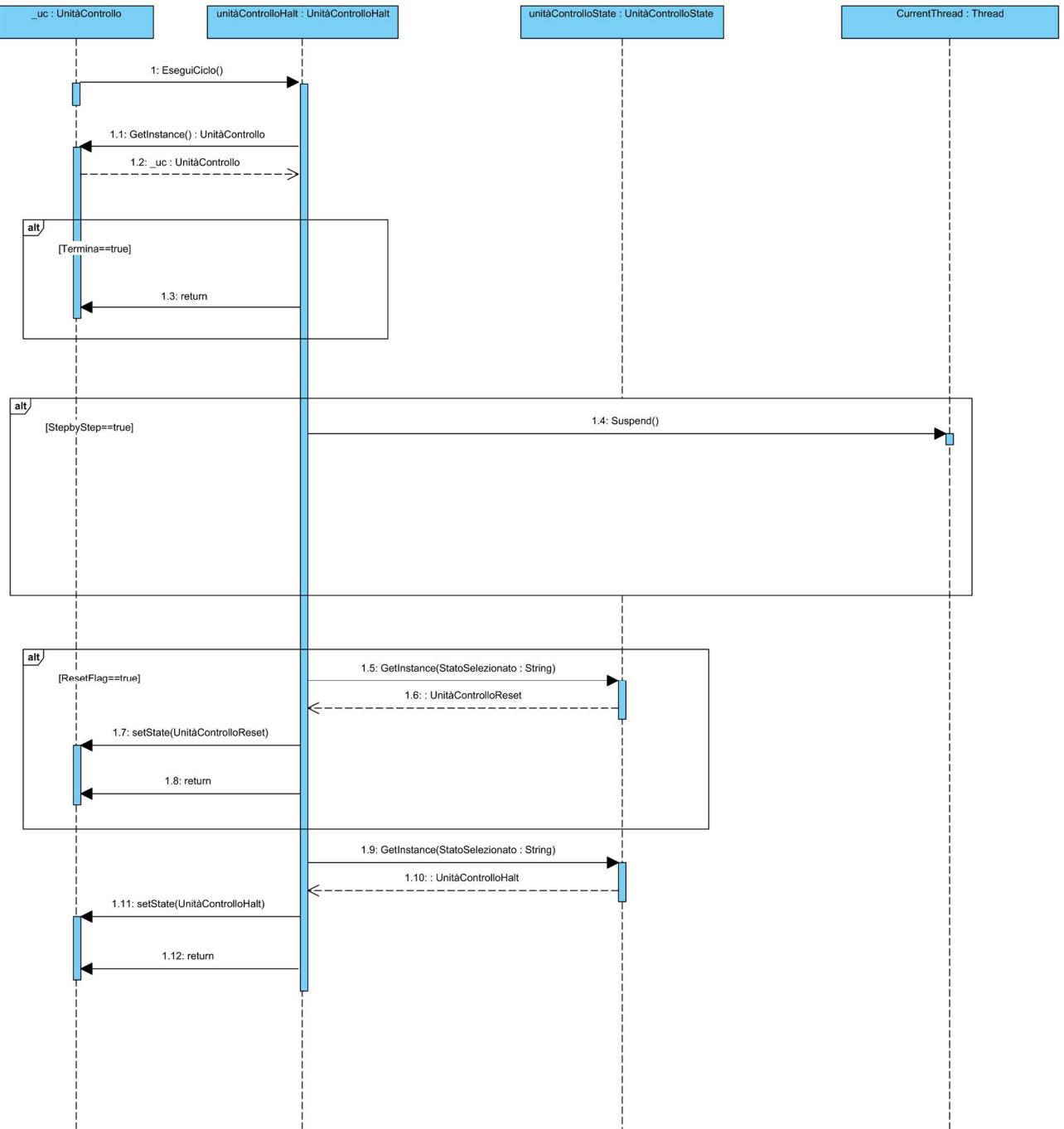
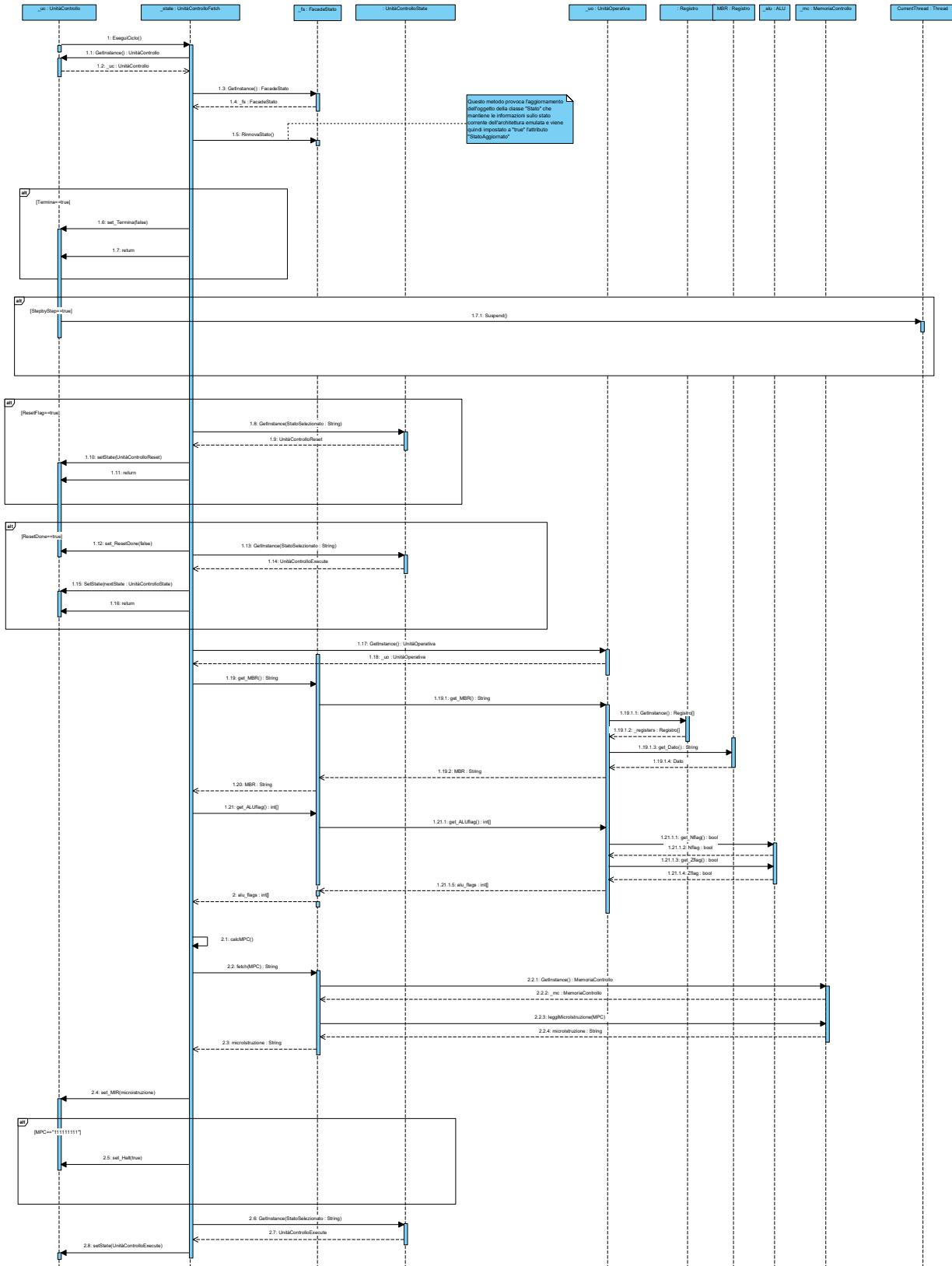
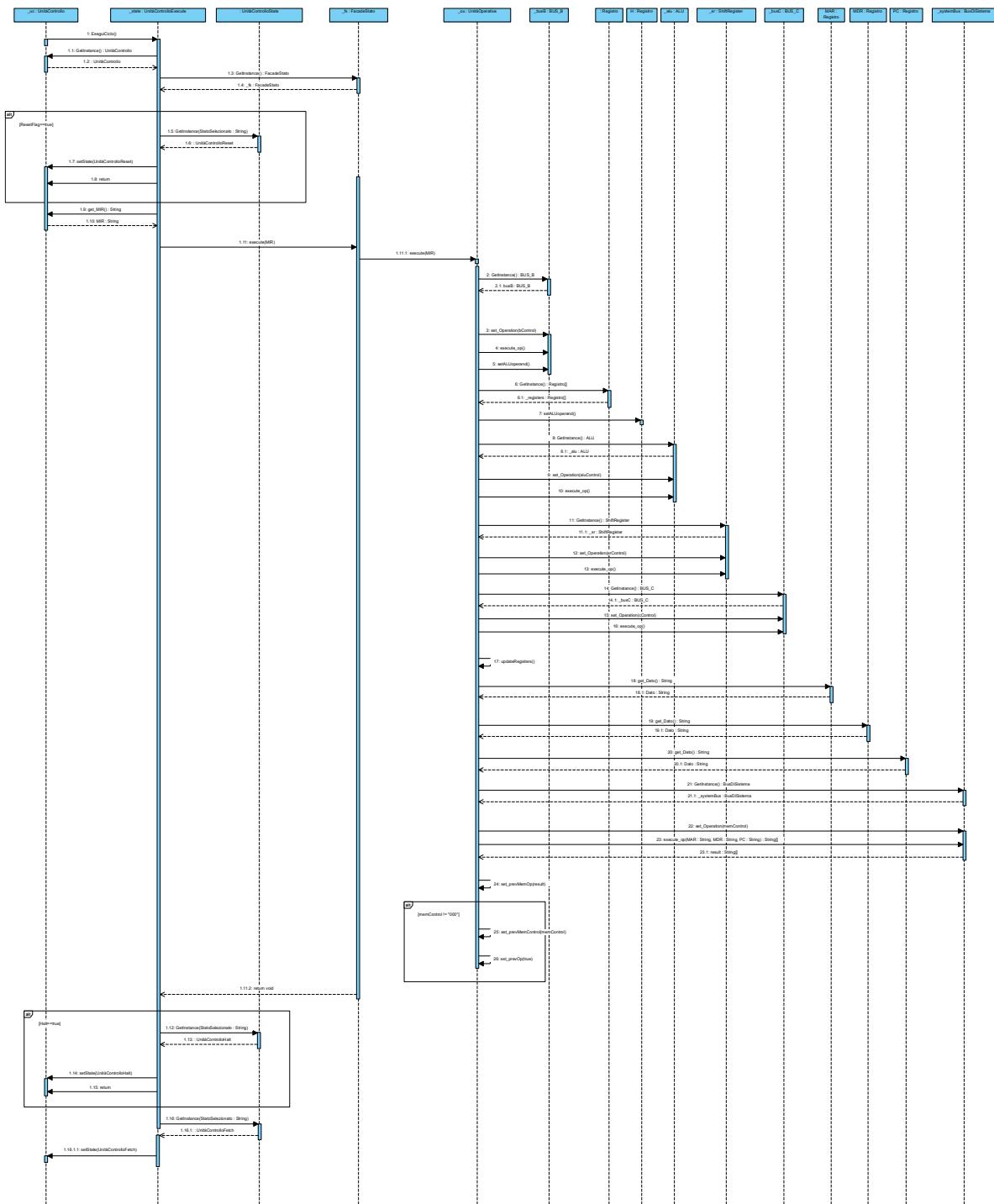


Diagramma 27: Seq_Diag_halt_eseguiCiclo





Adesso viene presentato il diagramma di sequenza che mostra ad un livello di dettaglio maggiore la dinamica del caso d'uso “EseguiProgramma”:

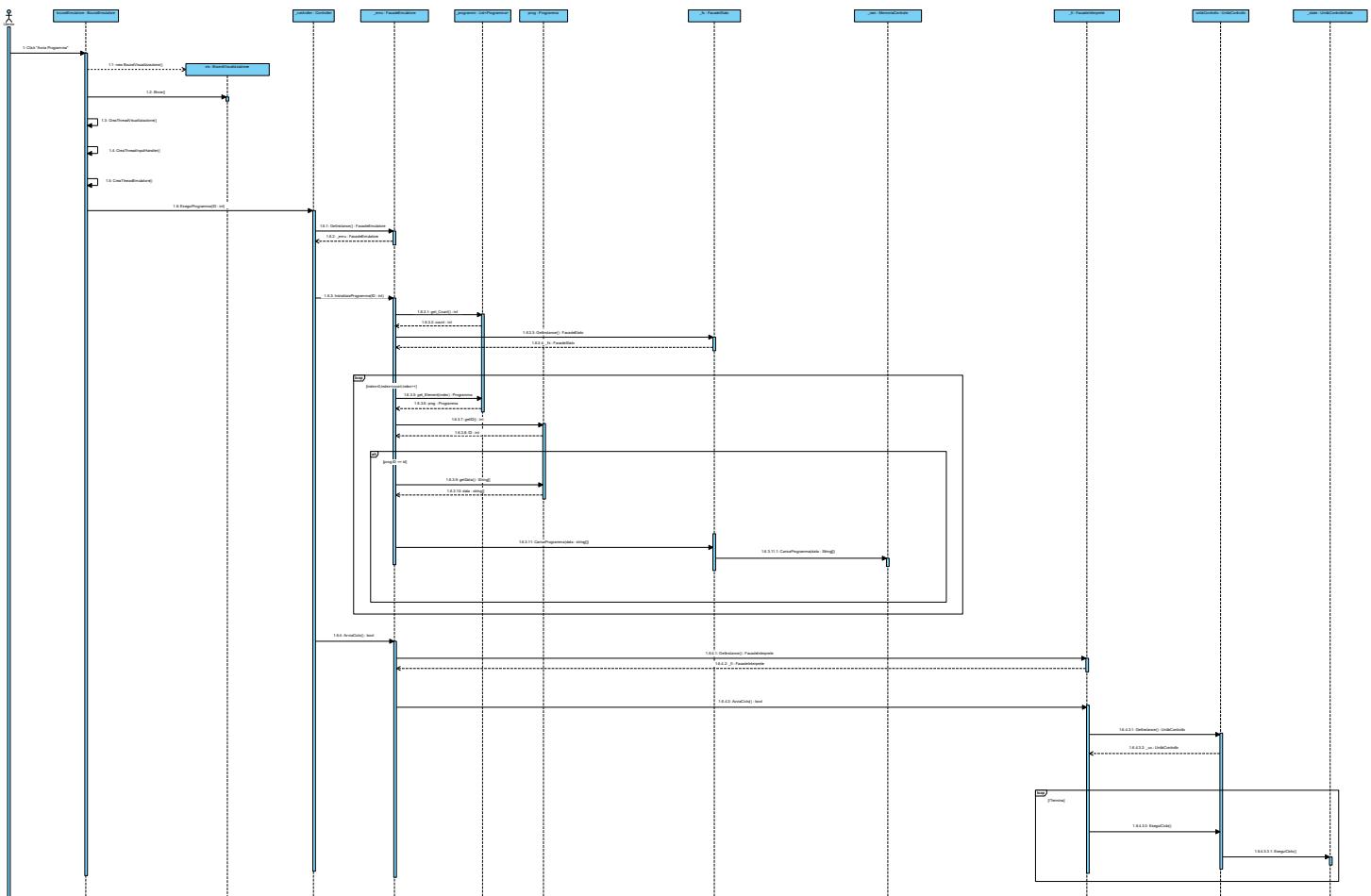


Diagramma 30: Seq_Diag_EseguiProgramma

In questo Sequence Diagram si presuppone che la lista dei programmi e la lista dei microprogrammi siano già state caricate e che il microprogramma sia già stato inserito in memoria di controllo. La scelta di un programma da eseguire dalla lista dei programmi si basa esclusivamente sull'ID e sul nome del programma considerato. Analogamente vale per la scelta di un microprogramma.

In particolare:

- L'utente, dopo aver selezionato un programma da eseguire, clicca su di un opportuno pulsante che ne consente l'avvio.
- Questo click provoca per prima cosa la visualizzazione del Form “BoundVisualizzazione”
- Si ha poi la creazione dei tre thread che operano in maniera concorrente: “ThreadEmulazione”, “ThreadVisualizzazione”, “ThreadInputHandler”
- Proseguendo, facendo riferimento al thread “ThreadEmulazione”, l'oggetto “BoundEmulatore” invoca la funzione “EseguiProgramma(ID : int)” sull'oggetto della classe “Controller”

- L'oggetto della classe “Controller” invoca la funzione “InizializzaProgramma(ID : int)” dell'oggetto della classe “FacadeEmulatore”
- Tra tutti gli oggetti di tipo “Programma” che la classe “FacadeEmulatore” possiede, viene selezionato quello il cui codice identificativo corrisponde all’ID passato in ingresso alla funzione.
- Di tale oggetto della classe “Programma” viene prelevato il contenuto, ossia il codice in formato binario, e sfruttando la funzione “CaricaProgramma(data : string[])” della classe “FacadeStato” e poi della classe “MemoriaCentrale” si ha il caricamento del programma da eseguire nella memoria principale.
- Al termine del caricamento, l'oggetto della classe “Controller” invoca la funzione “AvviaCiclo()” sull'oggetto della classe “FacadeEmulatore” e quest’ultimo, a sua volta, invoca l’omonima funzione sull’oggetto della classe “FacadeInterprete”
- L’oggetto della classe “FacadeInterprete”, all’interno di un ciclo infinito, invoca la funzione “EseguiCiclo()” sull’oggetto della classe “UnitàControllo” e quest’ultimo fa la stessa cosa sulla classe che implementa quella astratta denominata “UnitàControlloState” e che attualmente rappresenta lo stato corrente dell’unità di controllo
- Ovviamente questo ciclo infinito si ripete fino a quando l’utente non termina o sospende esplicitamente l’esecuzione del programma.

Un’osservazione importante: come già esplicitato nella sezioni di specifica e analisi dei requisiti, prima di poter eseguire un qualunque programma risulta necessario caricare in memoria di controllo un microprogramma. Per descrivere questa prima fase, è stato realizzato un ulteriore diagramma di sequenza:

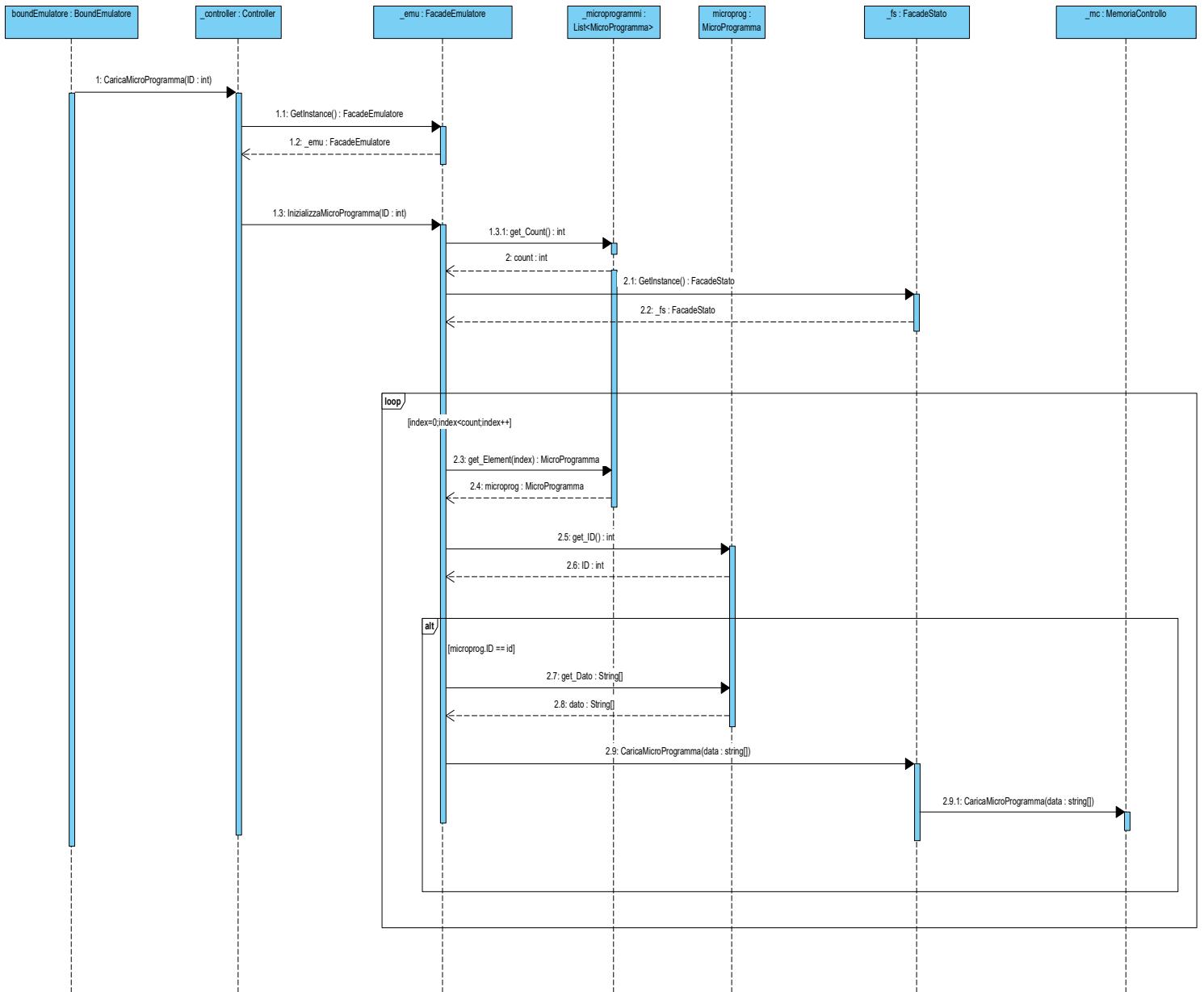


Diagramma 31: Seq_Diag_CaricaMicroProgrammaInROM

5.2.3) Interfaccia grafica

Ci si focalizza adesso sul livello che, all'interno dell'architettura closed layers dell'emulatore, si trova più in alto di tutti, ossia il livello “Interfaccia Utente”.

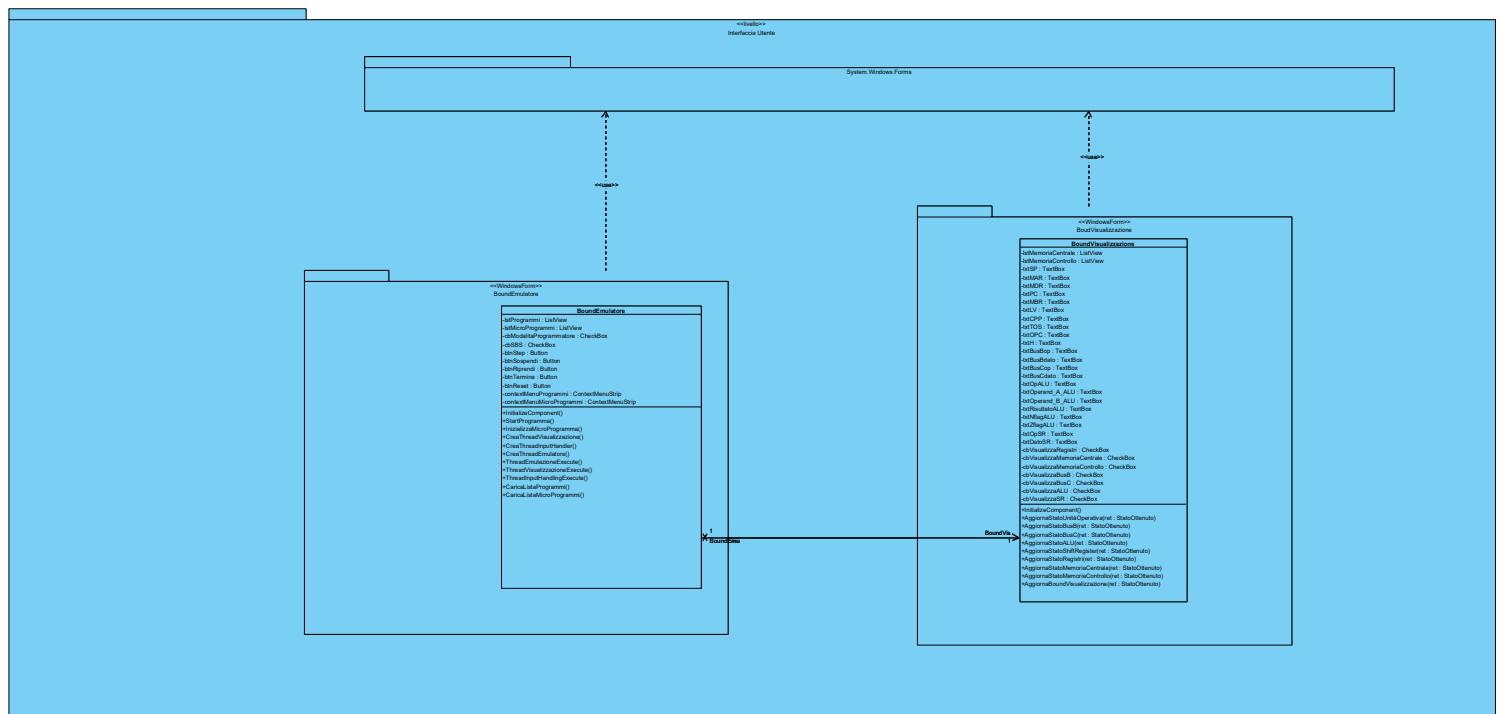


Diagramma 32: Porzione Class Diagram Emulatore

All'interno del package **“Interfaccia Utente”** sono presenti due package:

“BoundEmulatorePackage” e “BoundVisualizzazionePackage”. La classe **“BoundEmulatore”** funge da **Creator** rispetto alla classe **“BoundVisualizzazione”**: la prima ha una responsabilità di controllo nei confronti della seconda; la seconda si comporta sostanzialmente come una vista.

“BoundEmulatore” deve fornire dei metodi per linizializzazione dei componenti, per il caricamento del microprogramma e per il caricamento e l'avvio del programma. Si può in questa immagine osservare che questa classe dispone anche di diverse operazioni per l'avvio di flussi di esecuzione che operano in maniera concorrente, ma questo aspetto viene approfondito in una apposita sezione ulteriore della documentazione.

“BoundEmulatore” interloquisce con la classe “Controller” nel livello immediatamente sottostante per poter usufruire dei suoi servizi di caricamento microprogramma, caricamento e avvio programma.

Per la realizzazione dell’interfaccia grafica si è fatto uso dell’insieme di librerie note sotto il nome di **“WindowsForm”**, ossia la parte di GUI del framework Microsoft .NET.

Per quanto riguarda l’ottenimento dei dati sullo stato dell’architettura dal livello di logica applicativa si ha quanto segue: tali dati vengono racchiusi in una classe **“StatoOttenuto”** che viene passata dalla classe Controller alla classe BoundVisualizzazione nel momento in cui la classe **“BoundEmulatore”** lo richiede.

Come già detto, esisteranno tre flussi di esecuzione concorrenti. Per evitare eventuali problemi di sincronizzazione, mantenendo comunque una certa “velocità” nel sistema, si fa uso della tecnica di accesso sincronizzato alle funzioni delle classi di tipo Singleton.

La classe “BoundEmulatore”, sfruttando un opportuno thread, invia una richiesta di stato al “Controller” di modo tale che esso vada a popolare la classe “StatoOttenuto” già esposta. Dopo aver ottenuto lo stato, la classe “BoundEmulatore” si occupa di aggiornare la classe del package “BoundVisualizzazione”. A questo punto, quindi, la classe “BoundVisualizzazione” può aggiornare lo stato visibile della memoria di controllo, della memoria centrale, e dell’unità operativa con tutte le sue sotto parti.

L’interfaccia grafica relativa alla classe **“BoundVisualizzazione”** presenta una serie di sezioni, ciascuna mostra il contenuto di una porzione specifica dello stato estratto dagli elementi del modello, ciascuna visualizzabile o meno a scelta dell’utente a tempo di esecuzione.

La classe **“BoundVisualizzazione”** possiede una serie di operazioni che vengono utilizzate dal thread **“ThreadVisualizzazione”**: questi metodi consentono la visualizzazione dello stato aggiornato dei diversi componenti dell’architettura hardware emulata.

In termini di pure prestazioni, data l’interazione attiva con l’interfaccia, soluzioni asincrone avrebbero consentito l’ottenimento di risultati migliori, rendendo l’aggiornamento dei dati meno caotico di quanto in effetti sia, ma nel complesso quanto fatto si può ritenere soddisfacente.

“Seq_Diag_AggiornaBoundVisualizzazione”:

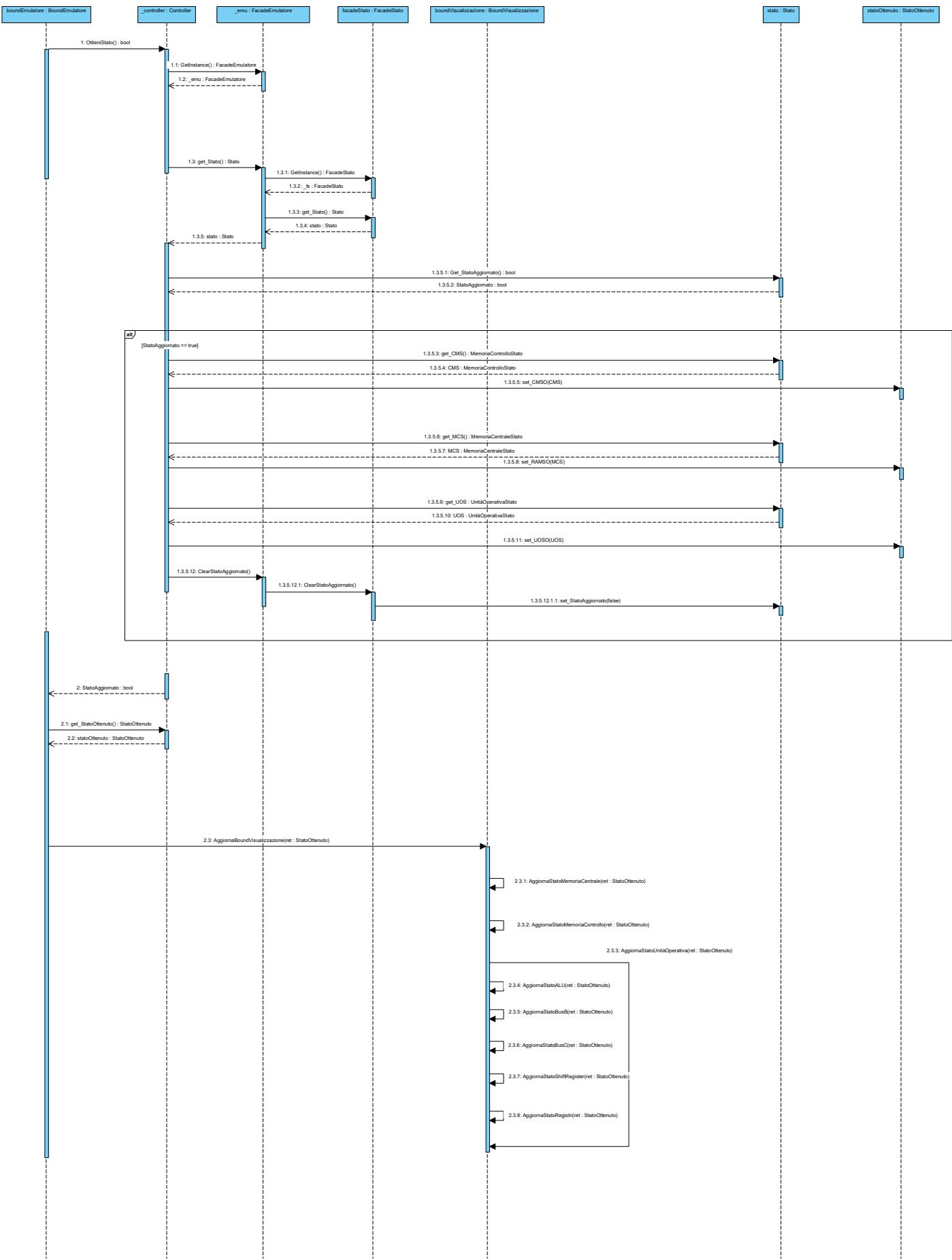


Diagramma 33: Seq_Diag_AggiornaBoundVisualizzazione

5.2.4) Gestione flussi di controllo concorrenti e trasporto delle informazioni sullo stato

In questa iterazione, coerentemente con i requisiti funzionali da sviluppare, è di rilievo capire come trasportare lo stato del dispositivo emulato ai livelli superiori dell'architettura. La procedura è la seguente: lo stato viene incapsulato in uno specifico oggetto, questo oggetto viene trasferito al controller e sarà poi quest'ultimo a fornire lo stato alla GUI. Per descrivere meglio la soluzione adottata, conviene riportare la parte del class diagram

“Class_Diag_Emulatore” contenente il livello “Controllo”:

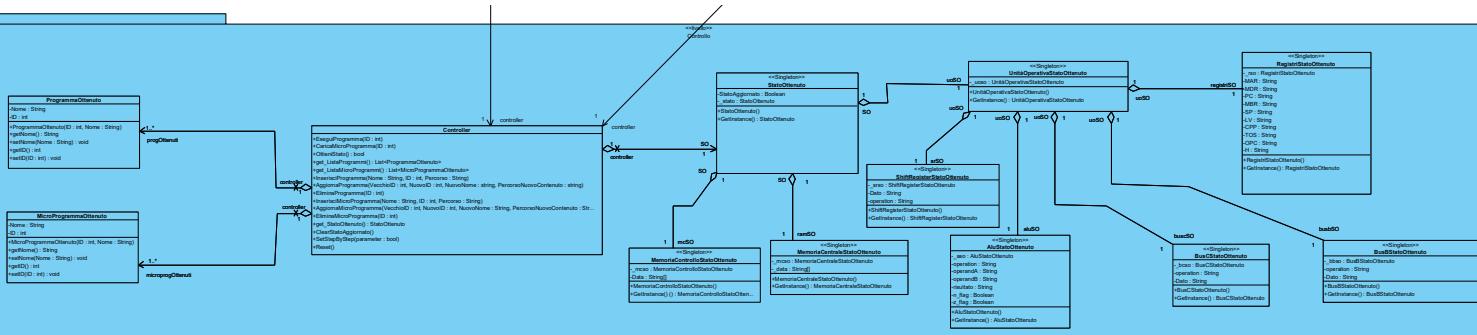


Diagramma 34: Porzione Class_Diag_Emulatore

È presente la classe **“StatoOttenuto”**, che viene aggiornata con le informazioni circa lo stato corrente dell'architettura emulata ogni qual volta viene invocata la funzione **OttieniStato()** della classe **Controller**. Attraverso questa funzione si esegue l'aggiornamento dello stato e successivamente si ha l'invio dello stato stesso all'interfaccia grafica, che riuscirà quindi ad aggiornare tutti quei parametri che fanno parte del concetto di stato. Dovrà esserci un thread che perennemente richiede lo stato mediante la classe **Controller** e quest'ultima andrà ad aggiornare lo stato fornito soltanto nel momento in cui lo stato dell'architettura vera e propria ha subito modifiche rispetto al prelievo di stato precedente. Non viene utilizzata in tale sede alcuna soluzione specifica di comunicazione tra threads diversi, semplicemente questo thread opera in modalità **“polling”**. La parte di trasferimento dello stato del dispositivo emulato è visualizzabile nel diagramma di sequenza mostrato poco sopra.

Focalizziamoci adesso sui flussi di esecuzione concorrenti. Nel momento in cui l'utilizzatore del sistema, dopo aver selezionato un microprogramma e averlo caricato in memoria di controllo e dopo aver selezionato un programma, decide di avviare quest'ultimo, vengono creati tre diversi threads che operano in parallelo:

- **ThreadEmulatore:** Si tratta del thread che gestisce il processo di emulazione dell'architettura hardware di interesse.
- **ThreadVisualizzazione:** Si tratta del thread che, in modalità polling, verifica continuamente se c'è stato un cambiamento di stato nell'architettura. Se così è, questo thread andrà a prelevare i dati relativi allo stato sottostante e andrà ad aggiornare “BoundVisualizzazione”.

- **ThreadInputHandling:** Si tratta del thread che si occupa di catturare gli input inviati dall'utente e di gestirli opportunamente. Sostanzialmente, gli input che l'utente può fornire all'applicativo consentono di manipolare il flusso di esecuzione del processo di emulazione (Reset, Sospendi, Riprendi, Termina e Step nel caso di modalità Step-by-Step).

In termini di design, per descrivere e rappresentare correttamente questi threads si è fatto uso di diagrammi di stato. Ovviamente, **dettagli significativi circa il funzionamento effettivo dei diversi threads possono essere visualizzati facendo riferimento ai rispettivi diagrammi di sequenza.** Il generico flusso di esecuzione (thread) per prima cosa deve essere creato. Dopo la creazione si troverà nello stato “**Unstarted**”, ad indicare il fatto che, pur essendo stato generato, non è stato ancora messo in esecuzione. Nel momento in cui viene messo in esecuzione si passa allo stato complesso denominato “**Running**”. Questo stato, essendo complesso, verrà descritto attraverso un ulteriore diagramma di stato. Tale stato, inoltre, si occupa di incapsulare l'esecuzione specifica del processo. Ovviamente, il generico thread potrà ricevere un comando di sospensione, passando quindi nello stato di “**Suspended**” e, eventualmente, in seguito, un comando di ripresa, passando nuovamente nel macro stato “**Running**”. Il generico thread, se necessario, potrà essere abortito: quando esso riceve un comando di terminazione, non viene subito distrutto, passa nello stato “**Aborted**” in cui esegue, se presente, una porzione di codice per la gestione di un'eccezione “**ThreadAbortException**” e, dopo aver fatto ciò, passa nello stato “**Stopped**”, dove sarà completamente terminato e deallocato.

Di seguito vengono riportati i diagrammi di stato prima citati.

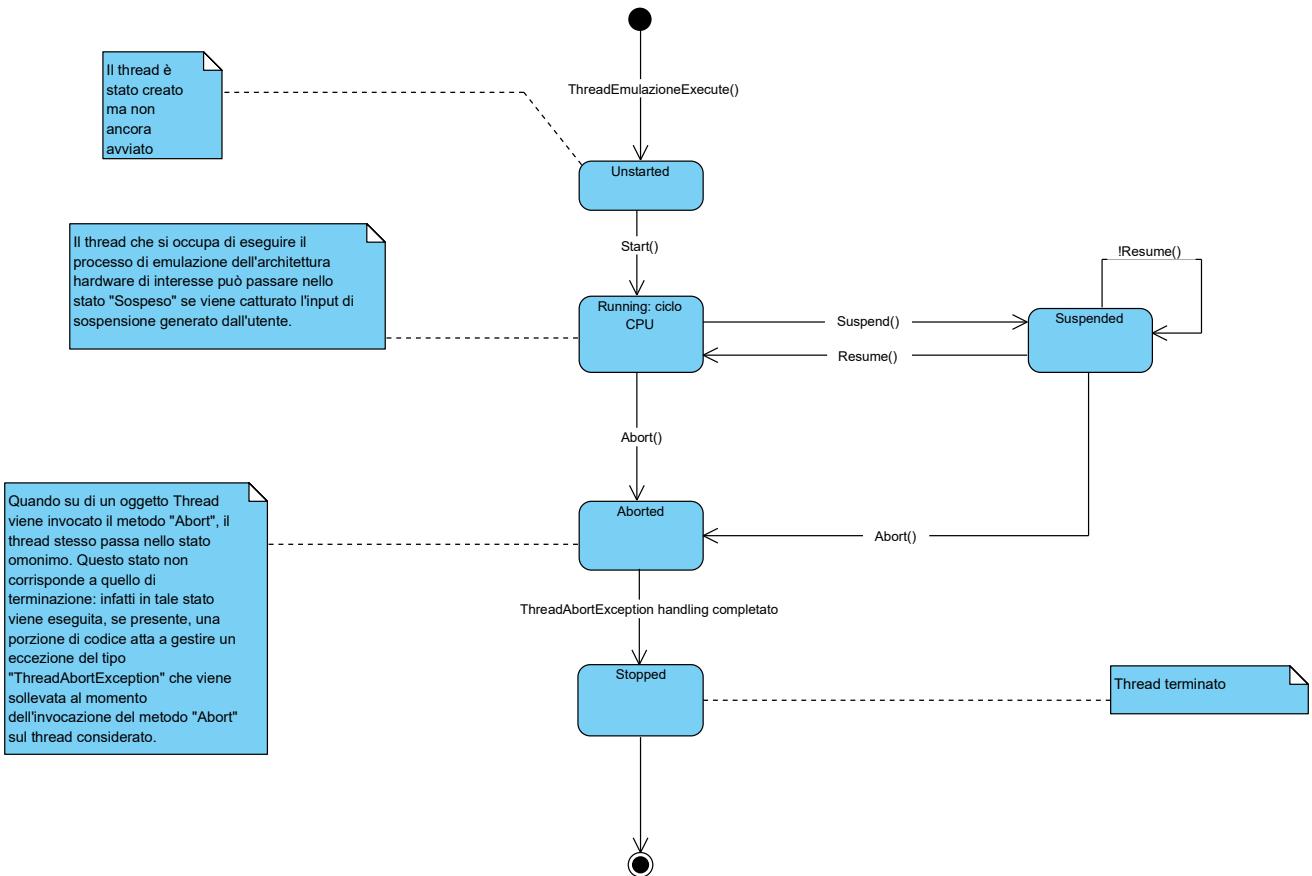


Diagramma 35: State_Diag_ThreadEmulazione

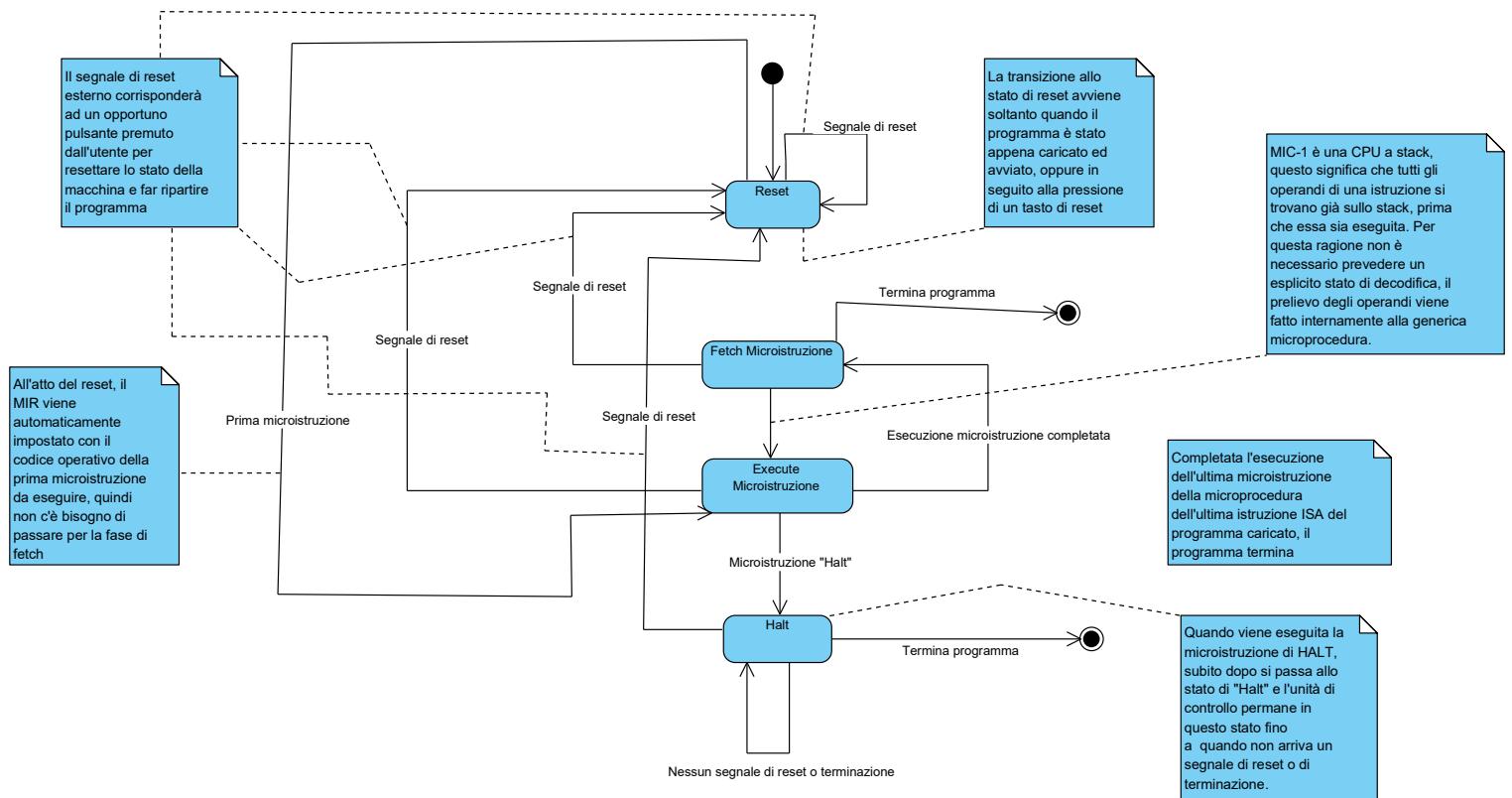


Diagramma 36: State_Diag_UnitàControllo

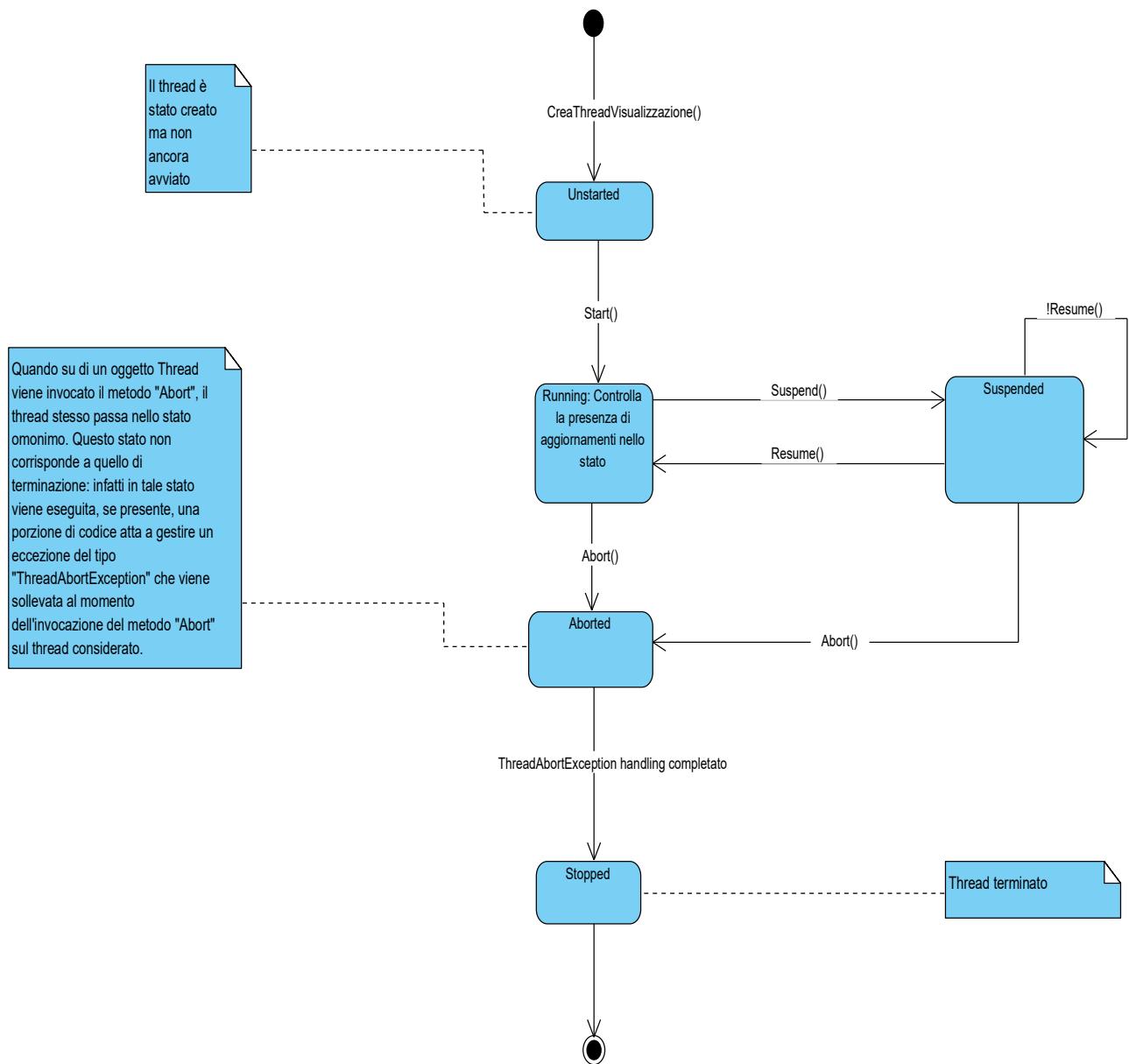


Diagramma 37: State_Diag_ThreadVisualizzazione

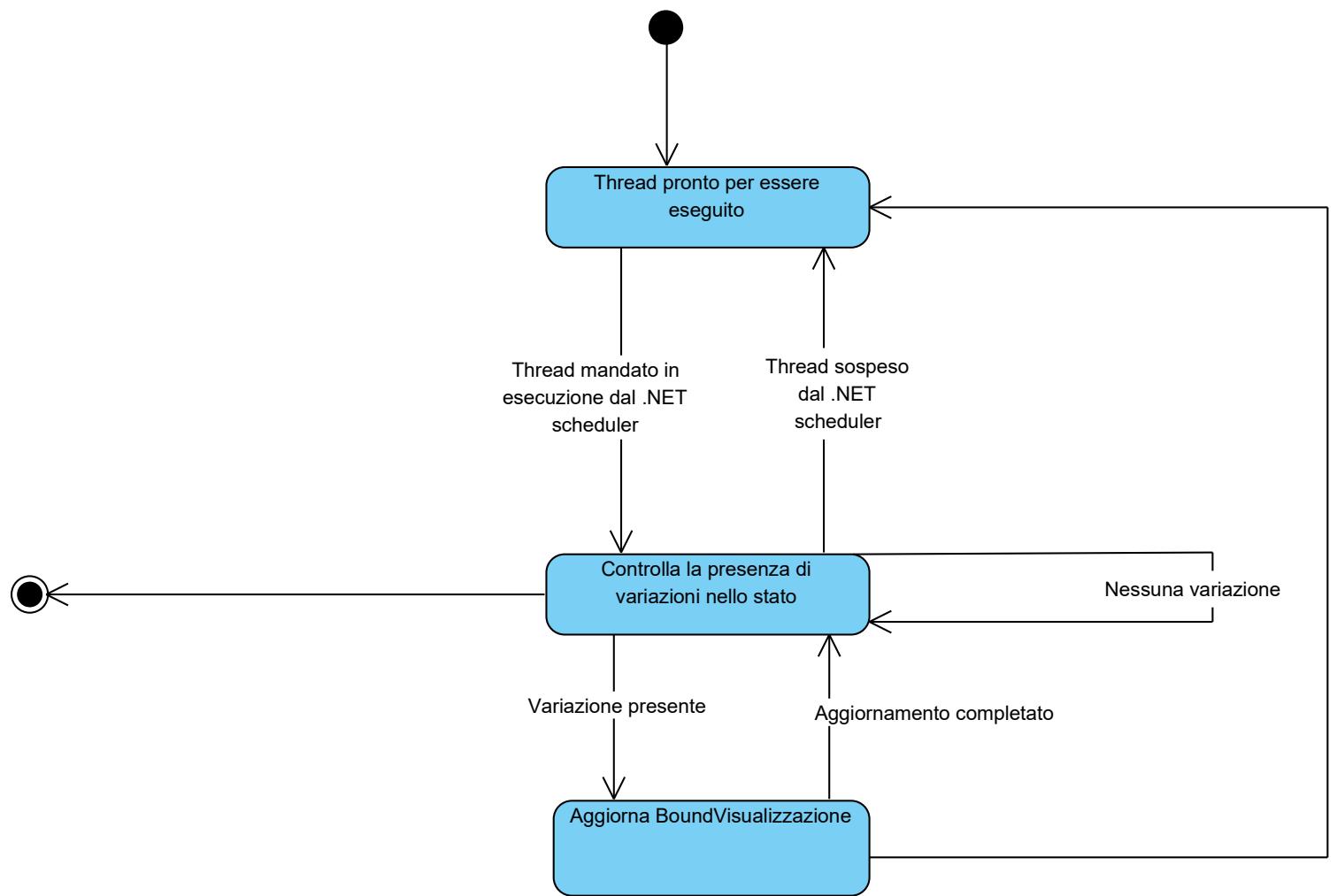


Diagramma 38: State_Diag_ControllaAggiornamenti

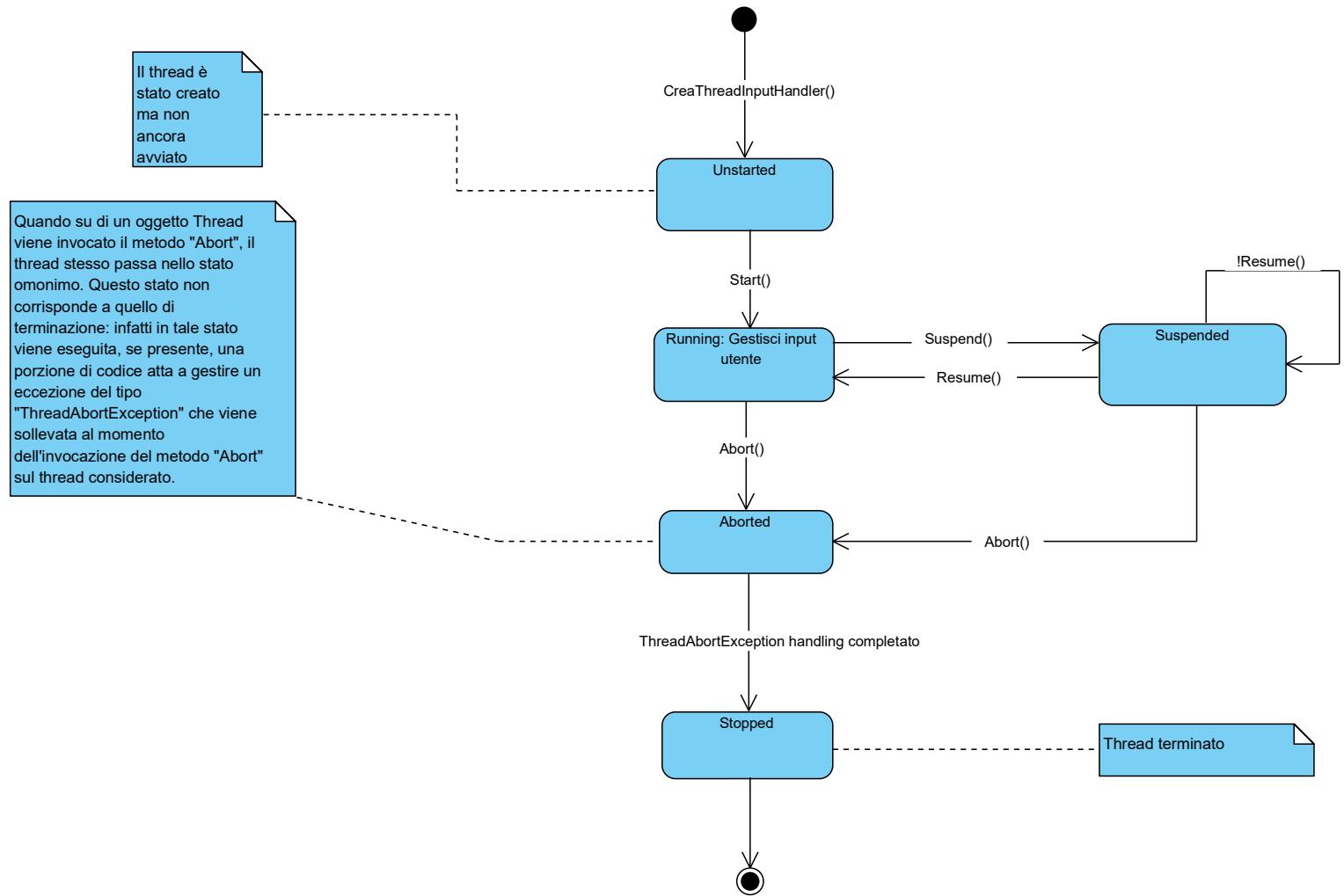


Diagramma 39: State_Diag_ThreadInputHandling

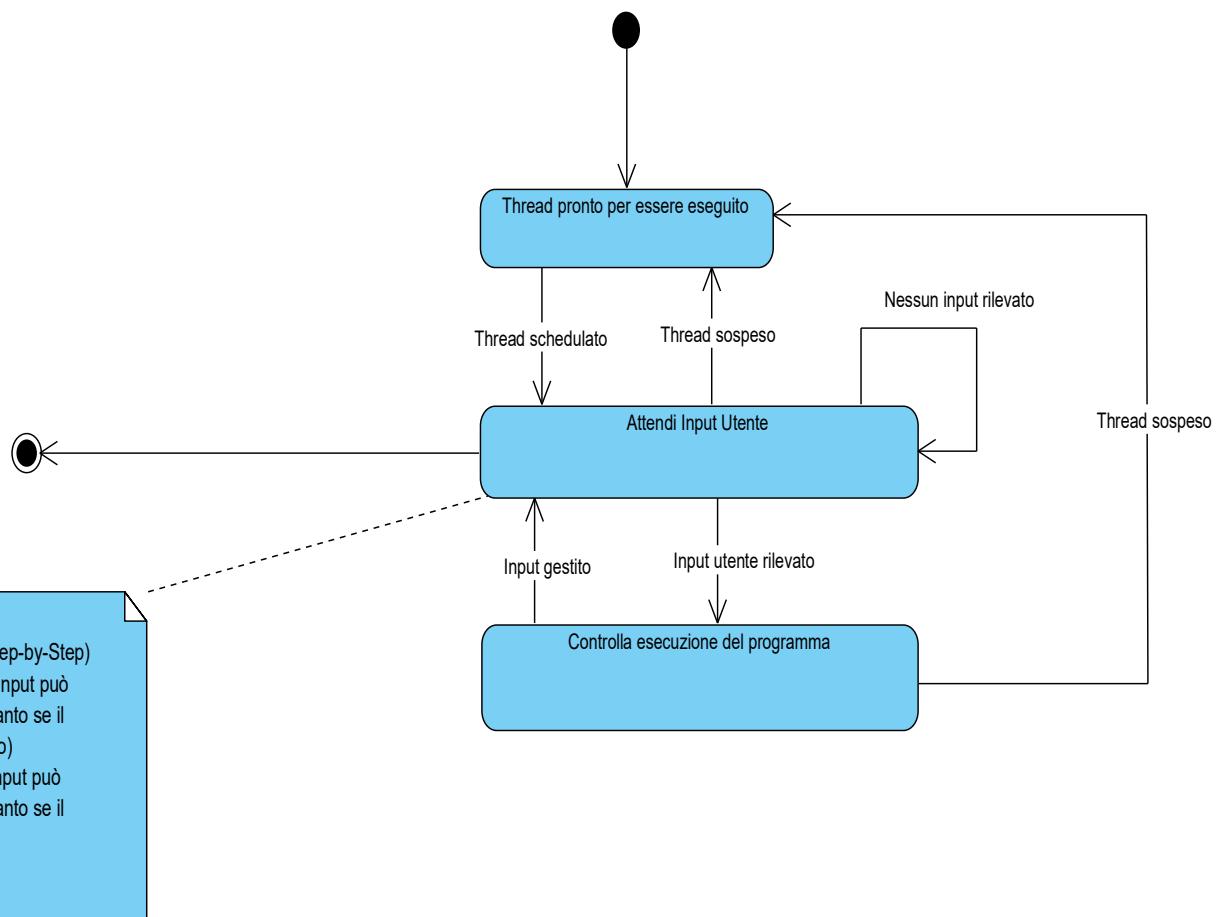


Diagramma 40: State_Diag_GestisciInputUtente

Per chiudere il cerchio, viene in questa sede riportato il diagramma di sequenza relativo al processo di gestione degli input utente.

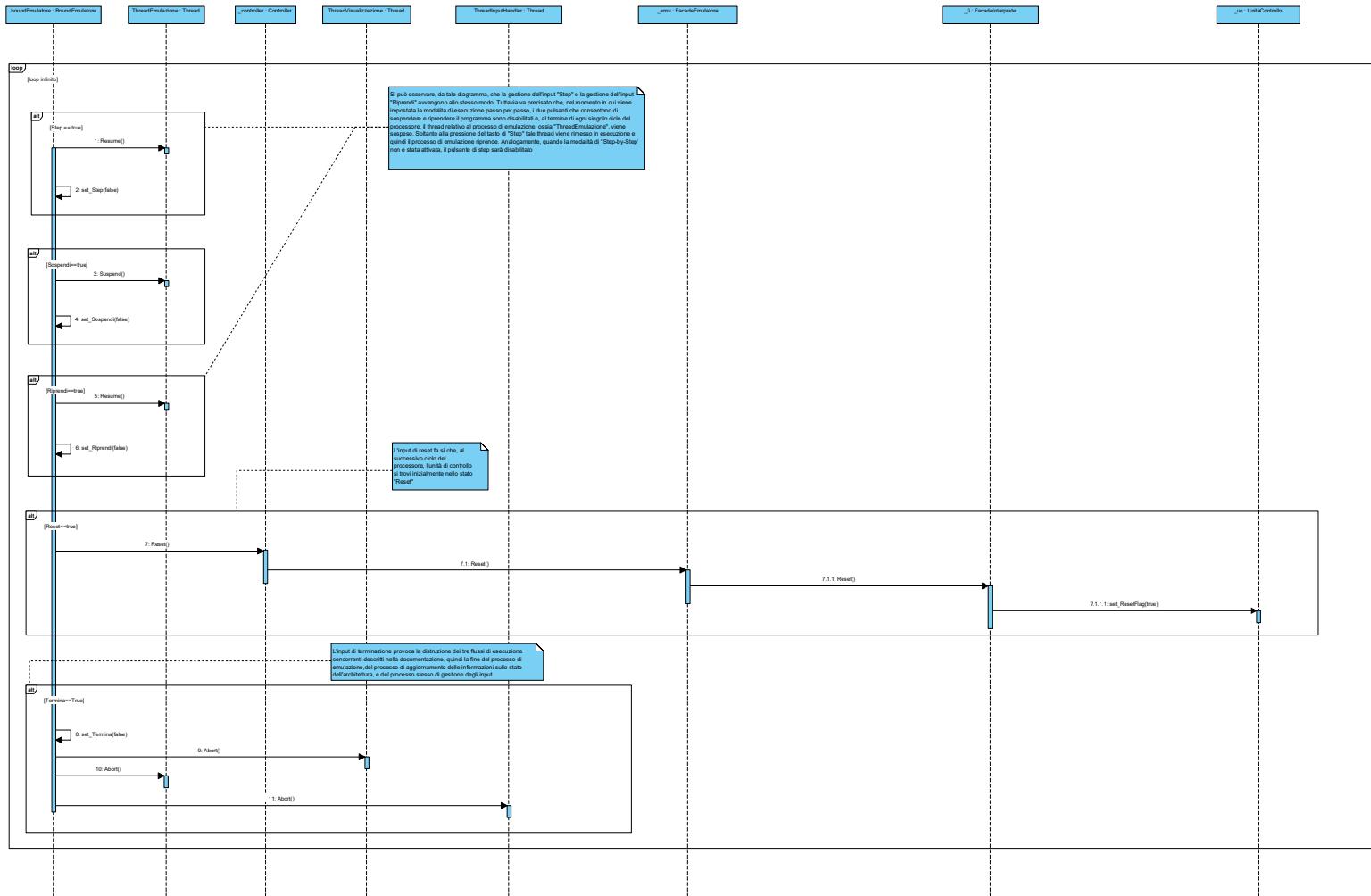


Diagramma 41: Seq_Diag_GestisciInputUtente

5.2.5) Gestione persistenza: lista programmi e lista microprogrammi

Uno degli scopi di questa iterazione è consentire il caricamento di un qualsiasi microprogramma nella memoria di controllo e di un qualsiasi programma nella memoria centrale ai fini dell'emulazione. Si faccia riferimento alla seguente porzione del class diagram

“Class_Diag_Emulatoro”:

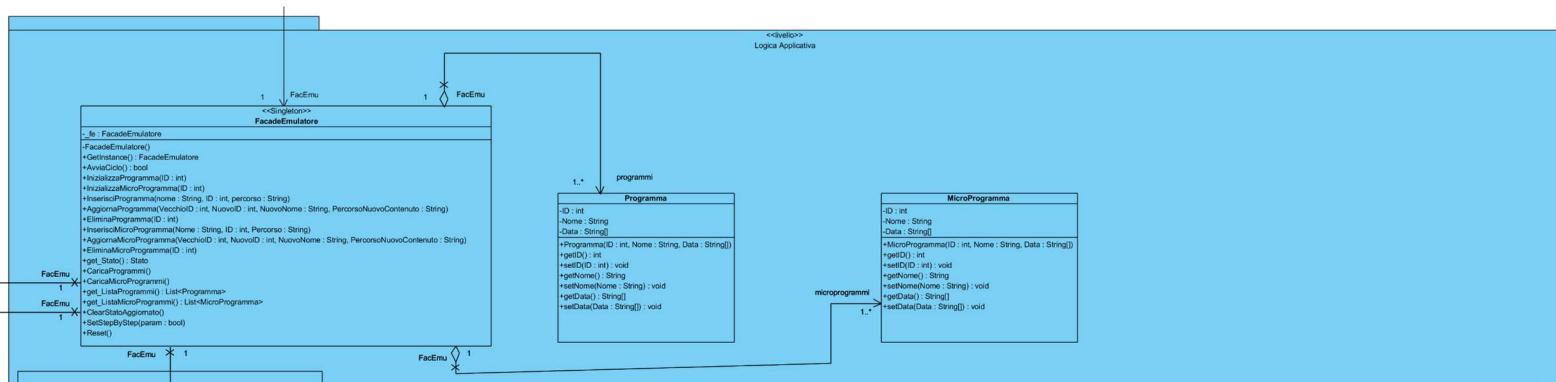


Diagramma 42: Porzione Class_Diag_Emulatoro

Sono presenti due nuove importanti classi: **“Programma”** e **“MicroProgramma”**.

Un oggetto della classe “Programma” mantiene le informazioni relative ad uno specifico programma che l’utente è interessato ad emulare.

Un oggetto della classe “MicroProgramma” mantiene le informazioni relative ad uno specifico microprogramma che l’utente è interessato a caricare.

Ogni programma o microprogramma, inteso come file presente nel file system, può essere visto come caratterizzato da tre parametri:

- Un primo parametro in cui viene specificato un codice identificativo
- Un secondo parametro in cui viene specificato il nome
- Un terzo parametro in cui viene specificato il contenuto

La classe “FacadeEmulatoro” è “information expert” degli oggetti relativi alla classe Programma e degli oggetti relativi alla classe “MicroProgramma”. Dunque, la classe “FacadeEmulatoro” non solo si occupa della creazione di tali oggetti ma anche della comunicazione con gli elementi propri di queste due classi.

Di seguito viene riportata la porzione del diagramma **“Class_Diag_Emulatoro”** relativa al livello **“Servizi Tecnici”**:

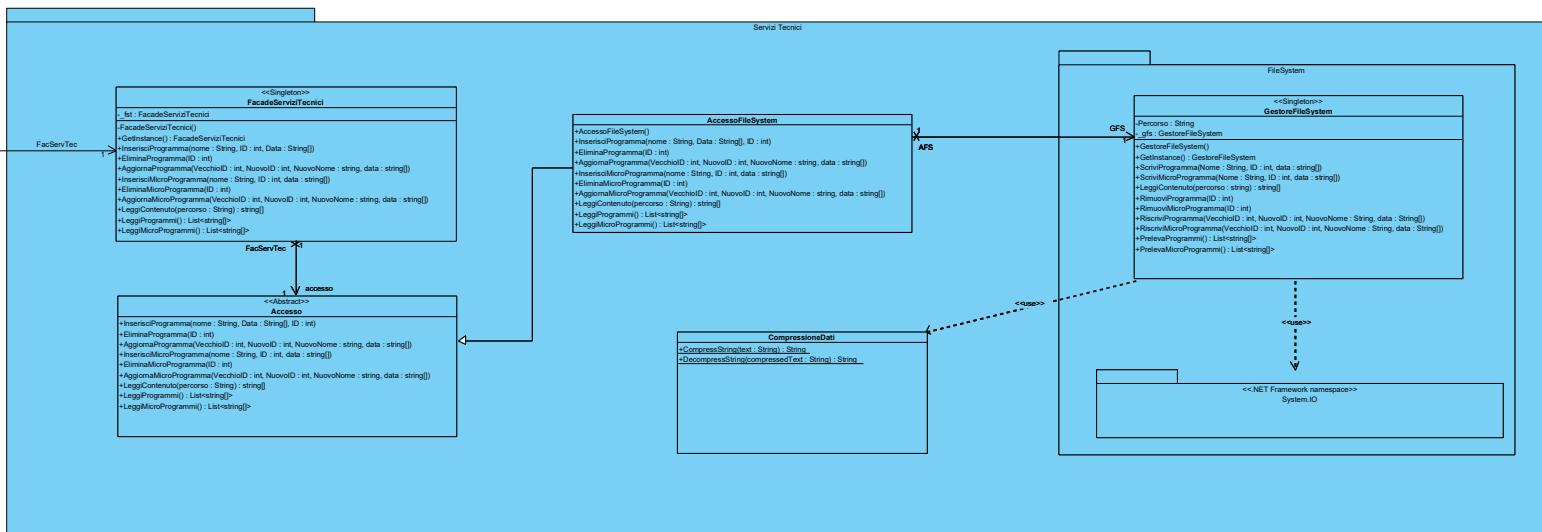


Diagramma 43: Porzione Class_Diag_Emulator

Questa parte del class diagram mostra l’interfacciamento con il File System. È stato usato anche qui il Design Pattern “Facade” e sulla classe “FacadeServiziTecnici” così ottenuta è stato poi applicato il Design Pattern “Singleton”. Attraverso la classe “FacadeServiziTecnici” si possono sfruttare i servizi per l’accesso al file system. La classe “AccessoFileSystem” comunica con la classe “GestoreFileSystem” che fornisce una serie di operazioni per la gestione dei programmi e dei microprogrammi. Ovviamente, tutte le operazioni legate a file che rappresentano programmi o microprogrammi devono opportunamente tenere conto della struttura dei file stessi.

La gestione della lista dei microprogrammi e della lista dei programmi è realizzata rendendo disponibili al livello di interfaccia utente alcune operazioni contenute nel controller. Sebbene lo abbia già fatto, riporto nuovamente la porzione di “Class_Diag_Emulator” relativa al livello “Controllo”, di modo tale da poter comprendere meglio quanto sto per dire.

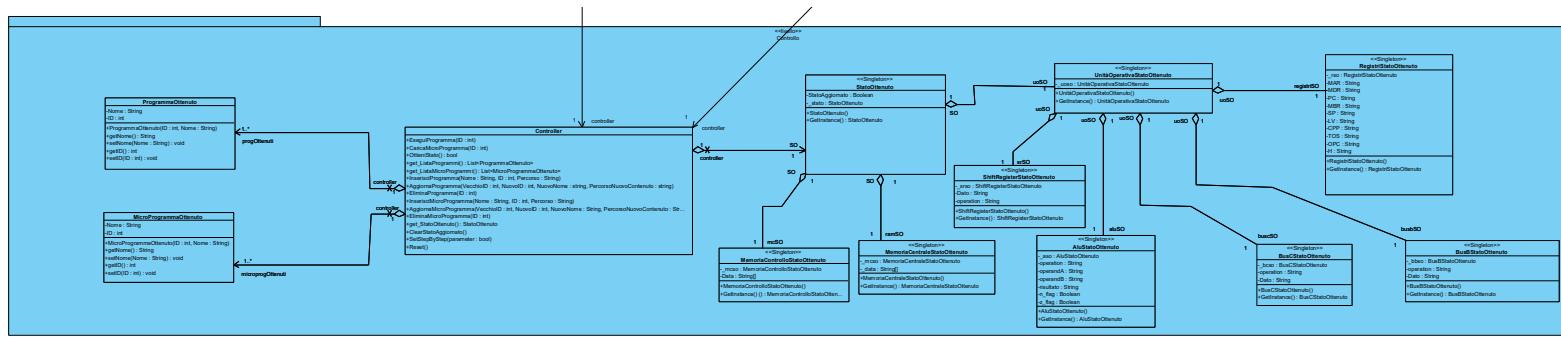


Diagramma 44: Porzione Class_Diag_Emulator

Di interesse sono le due classi denominate rispettivamente “**ProgrammaOttenuto**” e “**MicroProgrammaOttenuto**”. La classe “**Controller**” fornisce, quando richiesto, oggetti di tali classi all’interfaccia grafica. In particolare, ciò avviene attraverso i metodi “**get_ListaProgrammi**” e “**get_ListaMicroProgrammi**”. Inoltre, la classe “**Controller**” mette a disposizione una serie di operazioni per la gestione delle due liste:

- “InserisciProgramma”
- “AggiornaProgramma”
- “EliminaProgramma”
- “InserisciMicroProgramma”
- “AggiornaMicroProgramma”
- “EliminaMicroProgramma”
- “CaricaListaProgrammi”
- “CaricaListaMicroProgrammi”

L’**handling di queste operazioni richiede, evidentemente, l’utilizzo del livello denominato “Servizi Tecnici”.**

La lista programmi è rappresentata su disco mediante un file di questo tipo:

```

1 -----
2 #####
3 Nome Programma: Test_Constant
4 #####
5 ID Programma: 23
6 #####
7 Codice Programma:
8 UQEAAAB+LCAAAAAAABABzMDSy4DIgAPAOMDQwNMSrAKgCZIIDTjWGyLYYYpOAMBkQXEOQrYYwB1QM0wqQIL1
9 -----
10 -----
11 -----
12 #####
13 Nome Programma: IOR
14 #####
15 ID Programma: 113
16 #####
17 Codice Programma:
18 7gAAAB+LCAAAAAAABACFj7cBACAIBHum4aws3X8qEyJqIQ3hj1RIWfRjUtQYuCTPZRcaZByMgACgAdj+BLr
19 -----

```

Figura 6: Esempio lista programmi su disco

C’è una sezione per ogni programma memorizzato. La generica sezione è così strutturata:

- **Nome del programma**
- **ID del programma**
- **Codice del programma, non riportato in forma originale, ma opportunamente compresso per risparmiare spazio su disco.**

Analoghe considerazioni valgono per la lista dedicata ai microprogrammi.

```

1 -----
2 ######
3 Nome MicroProgramma: Default_MicroProgram
4 #####
5 ID MicroProgramma: 5
6 #####
7 Codice MicroProgramma:
8 0xcAAB+LCAAAAAAABAC9WP9z27oN/11/Bap4dZxFUpym7Vty7uYXu63bxM7ZTtPevNa0RntM9cUVpTi+l/e/I
9 -----
10 -----
11 #####
12 Nome MicroProgramma: Modded_MicroProgram
13 #####
14 ID MicroProgramma: 7
15 #####
16 Codice MicroProgramma:
17 /0kAAB+LCAAAAAAABADtmlF2GyEMRf+9Gt7+NxfHA9ITIGB63KZuHx85znAtJCFpBEkp3wMoyXhO4PH8CeRMl
18 -----

```

Figura 7: Esempio lista micropogrammi su disco

Quindi, una qualunque delle operazioni CRUD che si può effettuare sulla generica lista, agisce sul corrispondente file modificandolo opportunamente e preservandone la struttura.

Di seguito vengono riportati, a titolo di esempio, i diagrammi di sequenza relativi al caricamento della lista dei programmi.

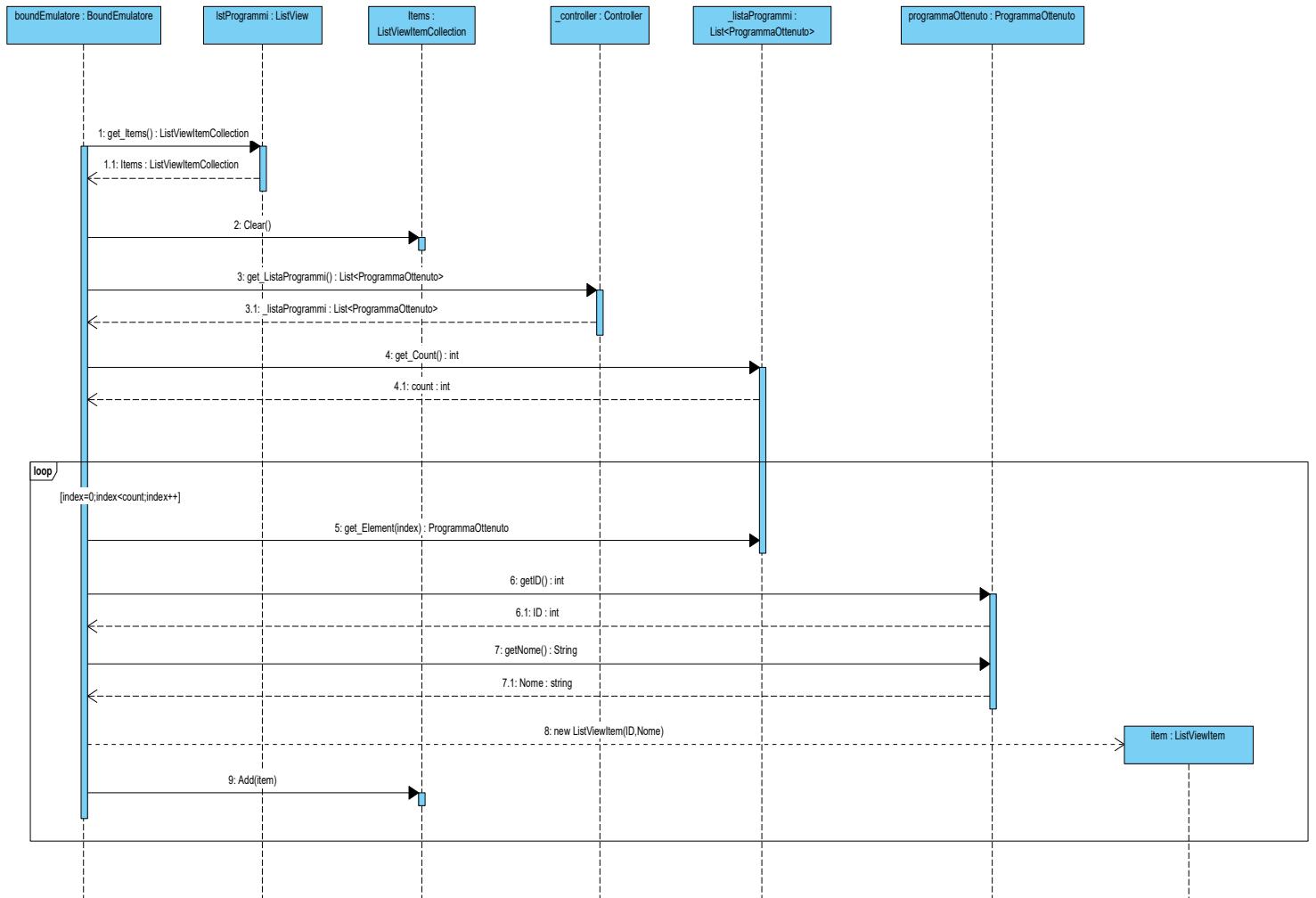


Diagramma 45: Seq_Diag_CaricaListaProgrammi

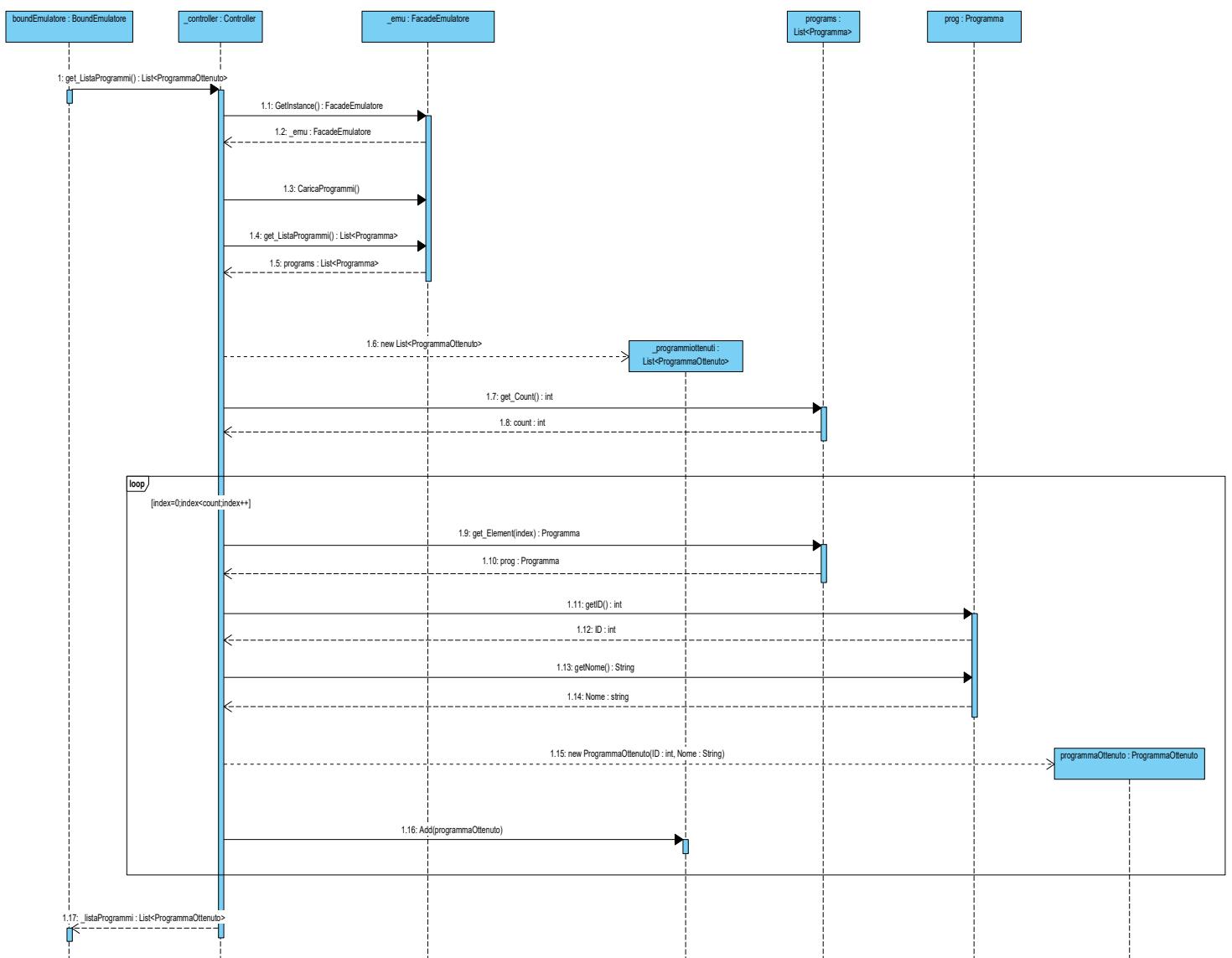


Diagramma 46: Seq_Diag_getListaProgrammi

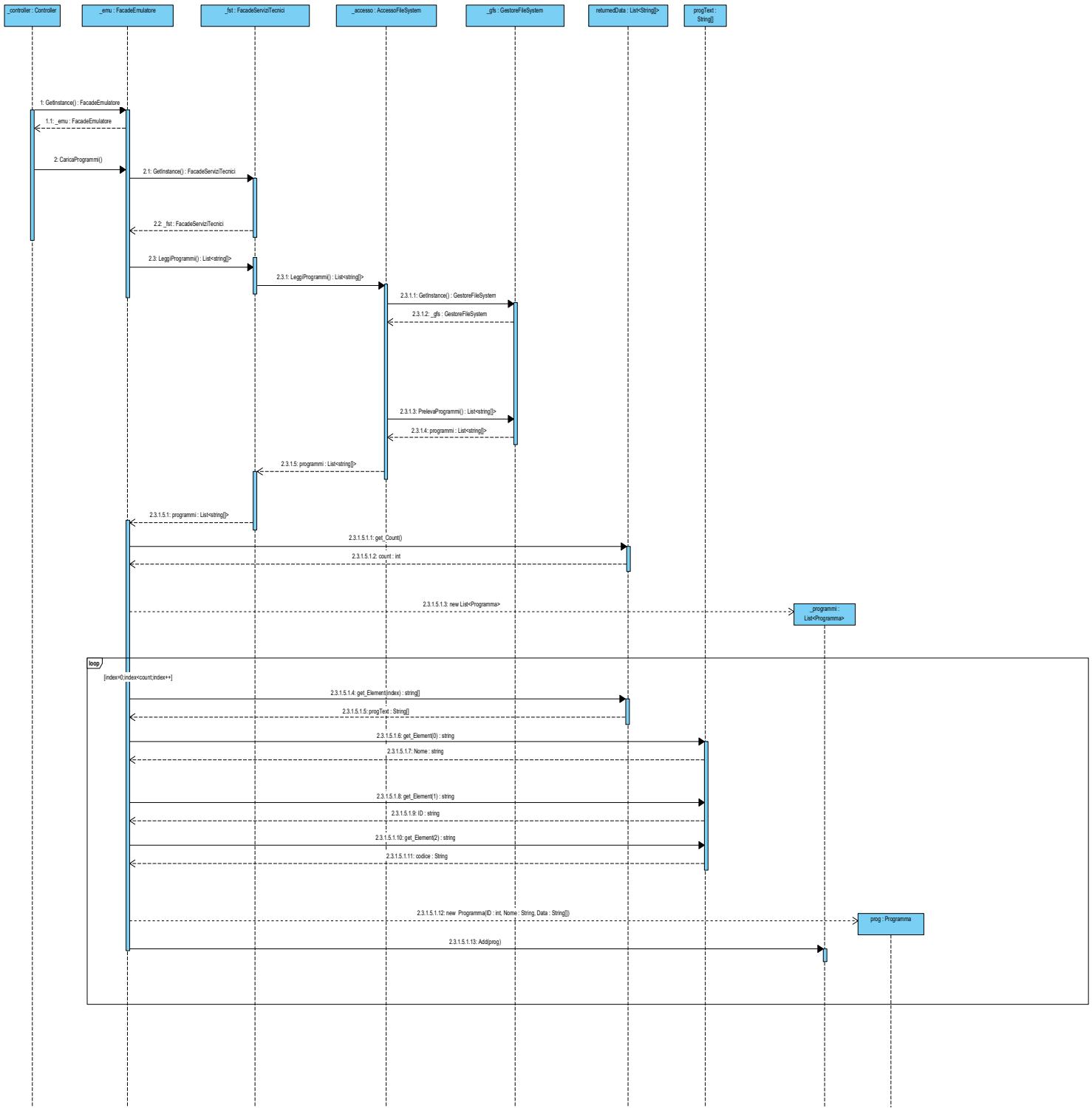


Diagramma 47: Seq_Diag_CaricaProgrammi

L'operazione di caricamento di una lista, comprensiva anche dell'aggiornamento opportuno dell'interfaccia grafica con i programmi caricati, è complessa, quindi il relativo sequence diagram, come visibile da queste immagini è stato suddiviso in tre diagrammi di sequenza, di modo tale da consentire una comprensione più chiara del flusso.

In maniera analoga si può descrivere il caricamento della lista dei microprogrammi: i relativi diagrammi di sequenza sono disponibili nel file con estensione “vpp”.

Ancora, a titolo di esempio, vengono riportati i sequence diagram relativi alle operazioni di aggiunta di un programma alla lista, modifica di un programma appartenente alla lista ed eliminazione di un programma dalla lista.

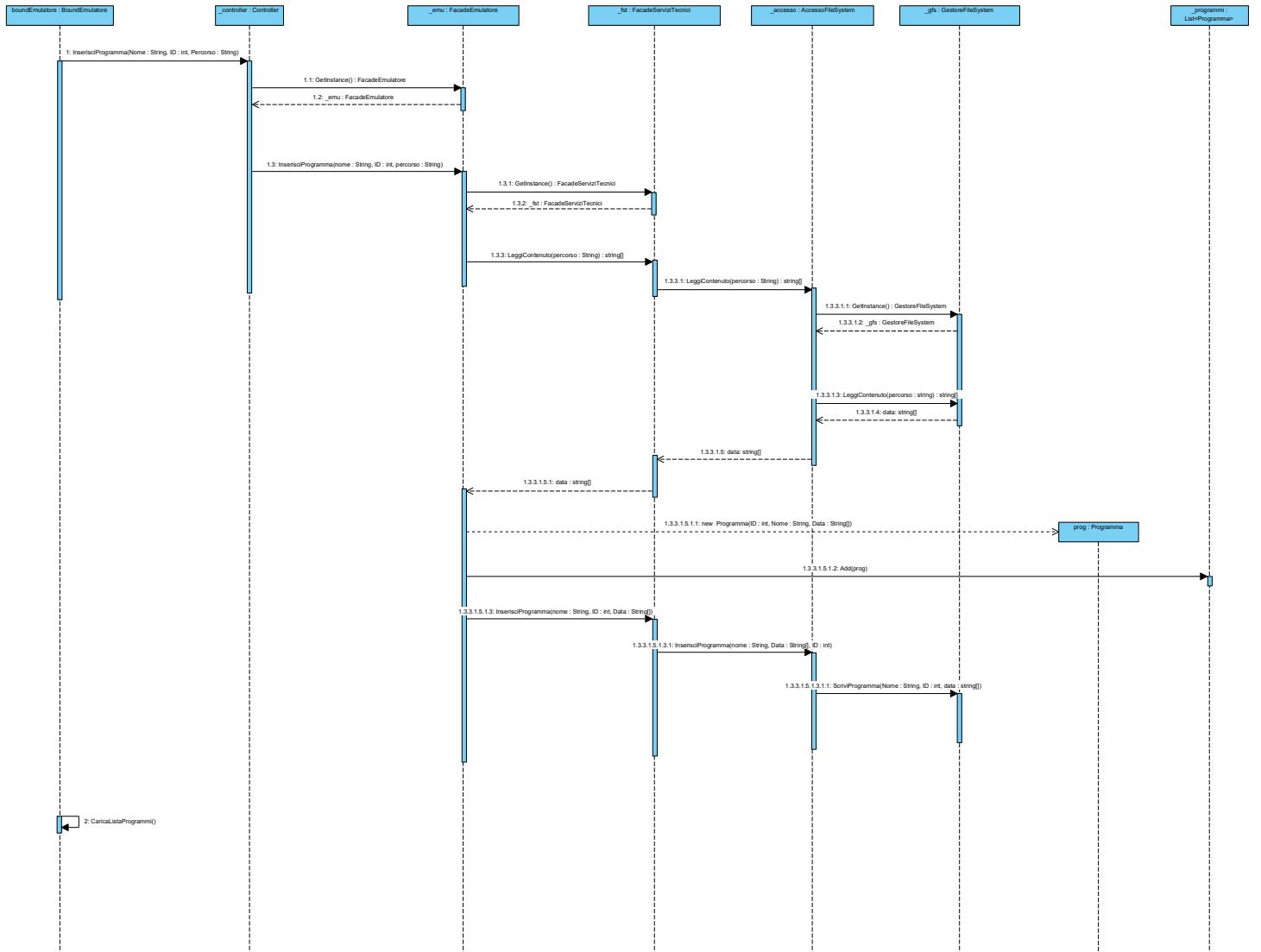


Diagramma 48: Seq_Diag_Inserisci_Programma

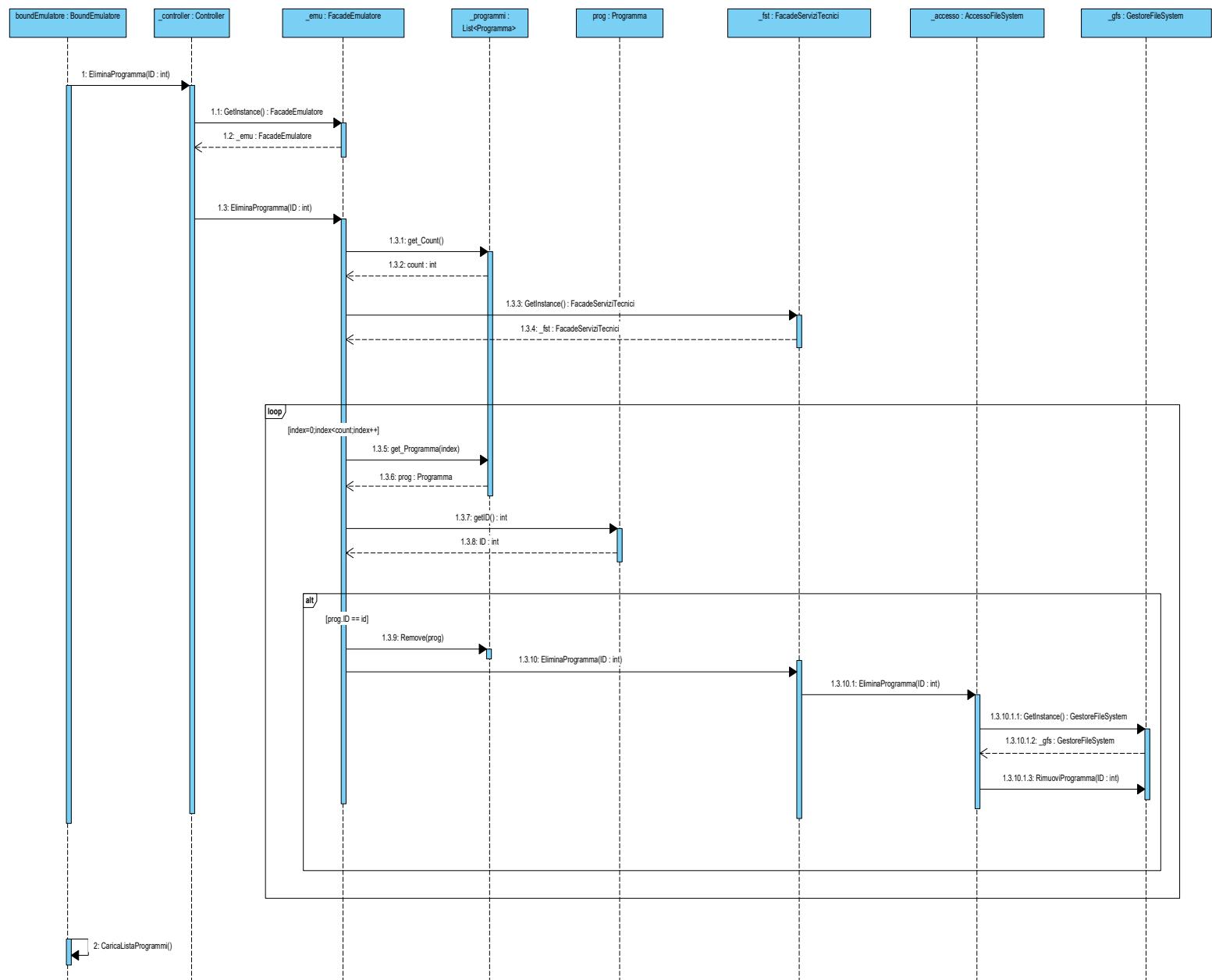


Diagramma 49: Seq_Diag_Elimina_Programma

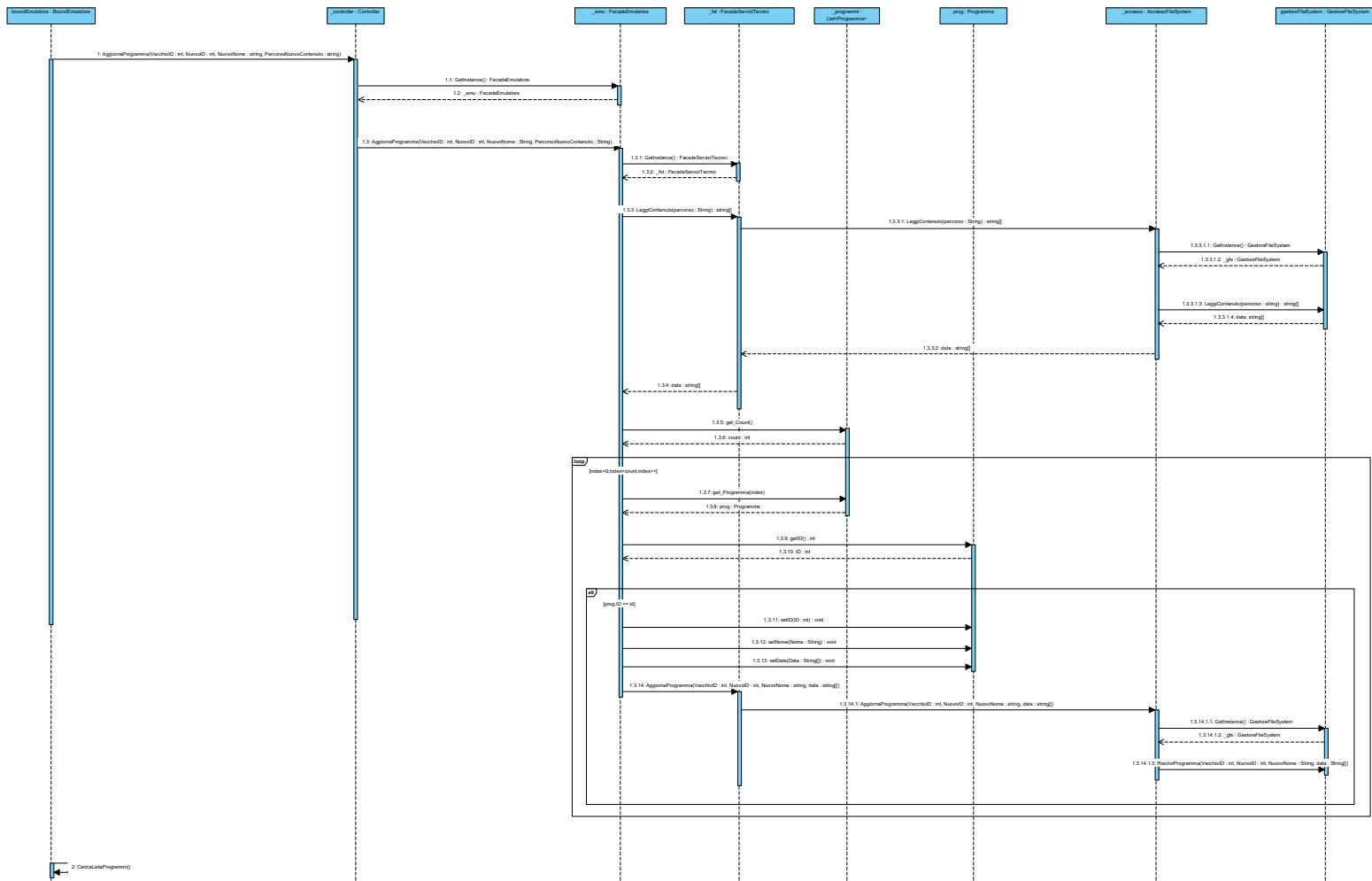


Diagramma 50: Seq_Diag_Modifica_Programma

In modo simile avvengono le omonime operazioni sulla lista dei microprogrammi.

6) Prospettiva dell'implementazione

In questa parte della documentazione ci si focalizza su due aspetti del sistema software:

- **Le scelte tecnologiche effettuate ai fini dello sviluppo dell'applicazione**
- **Gli artefatti che dovranno essere prodotti alla fine del progetto**

6.1) Prima iterazione

6.1.1) Prospettiva di deployment

Tecnologie utilizzate:

- **Linguaggio di programmazione C# (ambiente di sviluppo Visual Studio con estensione Resharper).**
- **CLR – Common Language Runtime, l'ambiente d'esecuzione del Common Intermediate Language, ossia il linguaggio intermedio in cui i compilatori della piattaforma .NET traducono i linguaggi ad alto livello supportati dalla piattaforma stessa (C#, VB.NET, F#, etc.).**
- **File System messo a disposizione dall'ambiente di esecuzione che fornisce il CLR. Nel mio caso, si tratta di Windows 10.**

Di seguito viene riportato un diagramma di deployment (“**Depl_Diag_MIC1-SYS**”) che schematizza quanto appena detto. Inoltre, in tale diagramma, per ogni istanza di uno specifico componente, viene rappresentato l'artefatto che lo manifesta.

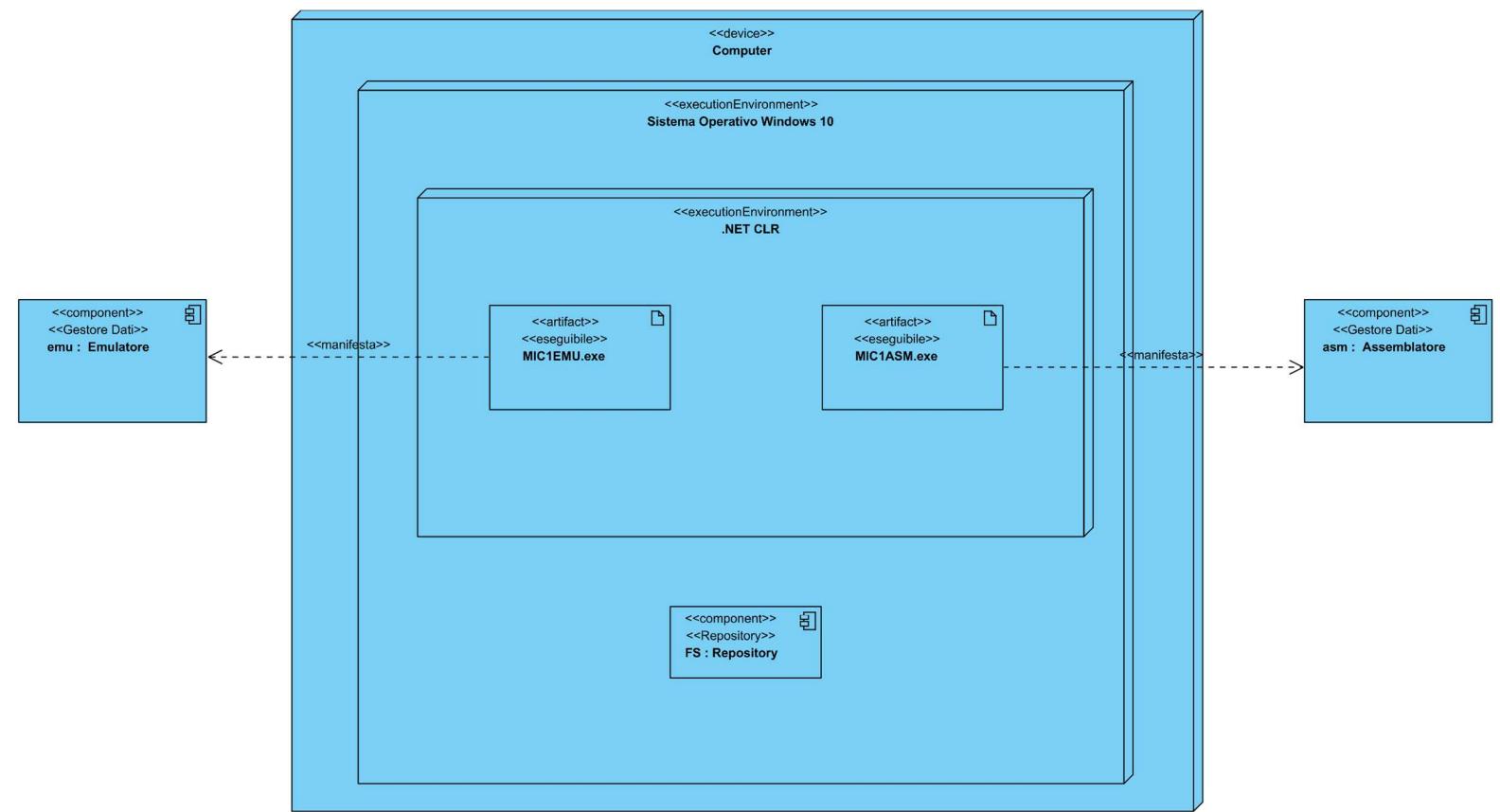


Diagramma 51: *Depl_Diag_MIC1-SYS*

6.2) Seconda iterazione

In questa iterazione si sfruttano i servizi messi a disposizione da alcuni framework e/o librerie inglobati all'interno del .NET framework. Ho ritenuto necessario descrivere brevemente quali specifici framework sono stati utilizzati, come e con che scopo, e di quali classi essi dispongono.

Viene infine fornito un breve accenno ad una estensione di Visual Studio, Resharper, che mi ha aiutato moltissimo durante lo sviluppo: un vero alleato nella scrittura di codice pulito, efficiente e ben formattato!

6.2.1) WindowsForms

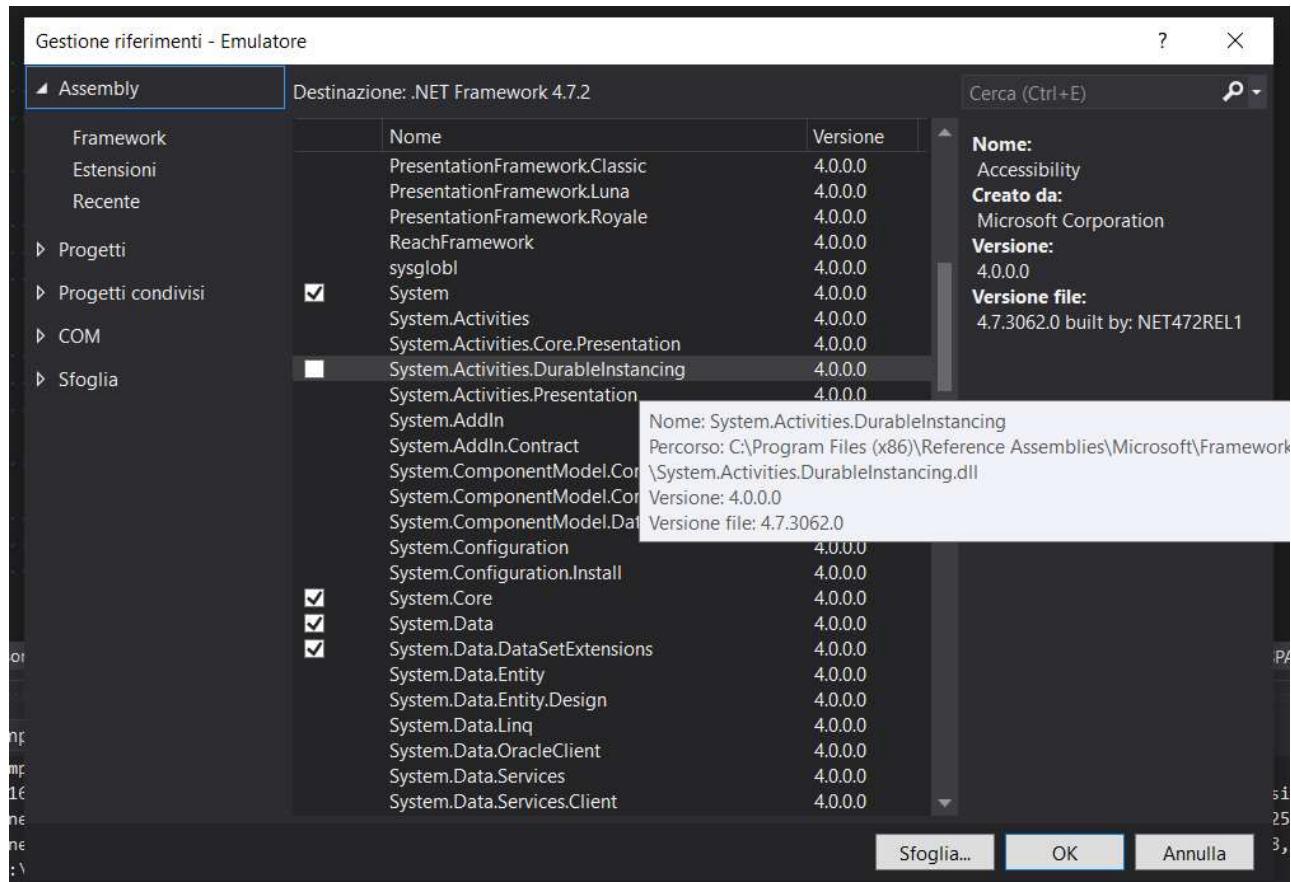
Per la costruzione dell'interfaccia grafica ho utilizzato il principale framework di cui .NET dispone per il raggiungimento di un tale obiettivo: “**WindowsForms**”. Lo spazio dei nomi “**System.Windows.Forms**” contiene classi per la creazione di applicazioni basate su Windows che sfruttano le funzionalità avanzate dell'interfaccia utente disponibili nel sistema operativo Microsoft Windows.

Si è scelto infatti di utilizzare un modo di lavorare agile, dunque il riuso è fondamentale. Di conseguenza, la scelta più sensata è stata proprio quella di adottare un framework che fornisse un insieme di classi già pronte ed efficienti per la gestione sia della parte grafica sia del prelievo degli input inviati dall'utente tramite il mouse o la tastiera.

Inoltre, “**WindowsForms**” permette la creazione di threads che operano in maniera concorrente grazie alla classe “**BackgroundWorker**”. Nonostante questa possibilità, per ragioni tecniche legate ad una più specifica gestione dei threads, per poter realizzare e gestire opportunamente i vari flussi di esecuzione concorrenti già menzionati in altre parti della documentazione, si è scelto di utilizzare la classe “**Threads**” messa a disposizione dal .NET framework, indipendente da “WindowsForms” ma comunque utilizzabili congiuntamente.

6.2.2) Configurazione del progetto e delle librerie.

L'ambiente di sviluppo “Visual Studio” è molto user-friendly; si fa carico di tutta una serie di problematiche relative alla configurazione delle librerie che con altre IDE bisogna per forza affrontare esplicitamente. Per l'utilizzo delle classi messe a disposizione per la creazione dell'interfaccia grafica e per la gestione dei Threads è sufficiente avere nel progetto un riferimento alle librerie **“System”** e **“System.Windows.Forms”**. In Visual Studio, per aggiungere un riferimento ad una libreria è sufficiente cliccare col tasto destro sull'icona del progetto e selezionare “Aggiungi...” e poi “Aggiungi riferimento”.



Si aprirà una schermata di questo tipo, dalla quale è possibile selezionare i riferimenti da aggiungere al progetto e infine applicare i cambiamenti. **Non è necessario inserire le librerie precedentemente menzionate all'interno della cartella del progetto: esse saranno già tutte presenti nel sistema operativo in uso dopo aver installato la versione più recente del .NET Framework.**

Per evitare di incorrere in problemi nella compilazione del progetto, è consigliato disporre dell'ultima versione di tale framework, ossia (al momento della realizzazione di questa applicazione) la 4.7.2. Ad ogni modo, dalla versione otto di Windows il framework è divenuto parte integrante del sistema operativo e viene generalmente aggiornato in maniera automatica.

6.2.3) Resharper

Resharper è una estensione di Visual Studio che fornisce un vasto insieme di soluzioni per il refactoring del codice. Si è rivelata un'estensione davvero interessante: difatti, il concetto di refactoring è strettamente legato agli approcci di sviluppo agili.

Con Resharper è possibile:

- Effettuare una analisi di qualità del codice: vengono messi in evidenza errori e avvertimenti nel codice, ma anche suggerimenti e indizi per migliorare il codice stesso (es: usare un costrutto switch invece di else if ripetuti oppure suggerimenti sulla visibilità di classi, attributi o metodi).
- Individuare ed eliminare i cosiddetti “bad smells” nel codice.
- Generare codice: il costruttore o distruttore di una classe, un metodo etc.
- Rinominare in maniera sicura classi, metodi e attributi.
- Estrarre una classe da una classe già esistente.
- Estrarre un metodo da un metodo già esistente.
- Etc...

Inoltre, Resharper si occupa di aggiungere automaticamente tutti i riferimenti alle librerie necessarie per la compilazione del progetto.

7) Descrizione del testing

Il testing ha rappresentato un aspetto cruciale nello sviluppo di questo sistema software.

Verificare il giusto funzionamento di caratteristiche come mostrare graficamente lo stato dell'architettura (segnali di controllo forniti ai diversi componenti, valori dei registri, operazioni svolte dall'ALU, operazioni svolte dallo Shift Register, contenuto della memoria centrale, contenuto della memoria di controllo etc.), gestire gli input utente, gestire la lista programmi, gestire la lista microprogrammi non è stato difficile.

Tuttavia, è stato complesso il testing relativo al cuore dell'architettura, ossia il processore, a causa della logica implementativa abbastanza complessa.

La scelta dello sviluppo di un tale sistema software, in questo senso, ha aiutato.

Nel libro di Tanenbaum, denominato "Architettura dei Calcolatori", il processore MIC-1 e l'ambiente IJVM sono analizzati nei minimi dettagli. C'è moltissimo materiale sull'architettura e, inoltre, le istruzioni supportate dal processore sono sviscerate nei loro singoli passi di esecuzione, il che fornisce una base ottima per verificare il corretto funzionamento dell'applicazione.

Terminata l'implementazione del processore, sono state realizzate delle funzioni di debug di modo tale da tracciare in maniera dettagliata l'esecuzione di diversi programmi di test, così da poter confrontare il log generato con le informazioni presenti nel libro. Questo ha permesso di individuare errori e correggerli.

Viene di seguito riportata una porzione di un esempio di log, relativo ad uno dei programmi di test utilizzati.

Tra l'altro, avendo già studiato questa architettura in due distinti corsi (“Architettura dei sistemi di elaborazione” e “Calcolatori Elettronici II”) da prospettive diverse, mi era già chiaro il corretto funzionamento di molte delle istruzioni e delle relative microprocedure.

Tutto ciò ha reso il testing molto più semplice, permettendomi di stabilire il corretto funzionamento dell'emulazione in pochi giorni.