MIC1-SYS

Paolo Amato

Sommario

1) Processo di sviluppo

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 quando 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.

- **CPU:** Unità centrale di elaborazione, corrispondente in questo caso al processore MIC-1, del quale verrà implementato l'ISA.

- **Unità operativa:** Questo blocco incapsula la parte inerente 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.
- **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.
- **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.
- **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
- **RAM:** Random Access Memory, si tratta della memoria principale utilizzata dal processore
- **Opcode:** codice operativo in base al quale l'unità di controllo comanda l'unità operativa e stabilisce cosa quest'ultima deve fare
- **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.
- **Programma:** Blocco di istruzioni che il MIC-1 è in grado di interpretare
- **Microprogrammazione:** tecnica specifica utilizzata per la realizzazione dell'unità di controllo. Si contrappone alla logica cablata
- **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 microprocedura
- **Microistruzione:** parola ad n bit che fornisce i valori che i segnali di controllo, mandati dall'unita di controllo all'unità operativa, devono assumere
- **Microprocedura:** la sequenza di microistruzioni relativa ad una data istruzione dell'ISA costituisce la microprocedura che implementa quella istruzione.
- **ALU:** questo termine identifica la parte del processore che si occupa di svolgere calcoli matematici e operazioni logiche
- **Shift Register:** registro a scorrimento collegato in uscita all'ALU
- **PC:** registro interno alla CPU utilizzato per mantenere l'indirizzo della successiva istruzione da eseguire
- **IR:** registro della CPU che immagazzina l'istruzione in fase di elaborazione.
- **IJVM:** architettura di elaborazione che prende in considerazione il sottoinsieme delle istruzioni della JVM che lavorano soltanto sui numeri interi.
- **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 del processore stesso.

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**
- **Inizializza 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 selezionato un programma dal file system, potrà eseguirlo 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.
- **InizializzaEmulazione:** Prima di caricare ed eseguire un programma, l'utente sarà in grado di vedere e modificare proprietà tipiche dell'emulazione come il microprogramma memorizzato nella memoria di controllo, le istruzioni ISA supportate, i valori iniziali dei registri ecc.
- **MostraInformazioniArchitettura:** Il programmatore potrà visualizzare informazioni su specifici elementi dell'architettura, di modo tale da verificare il comportamento di tali componenti architetturali.

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	1) Scegli un programma dal file system 2) Avvia programma
Estensioni/Scenari alternativi	2a) L'utente mette in pausa il programma 2b) 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
Scenario principale	1) Scegli componente
Estensioni/Scenari alternativi	1a) Se viene scelta la CPU 1) Visualizza i valori dei registri 1b) Se viene scelta la memoria centrale 1) Visualizza stack, constant pool, method area 1c) Se viene scelta la memoria di controllo 1) Visualizza le microprocedure e le microistruzioni ad esse appartenenti

3.2.3) AssemblaCodice

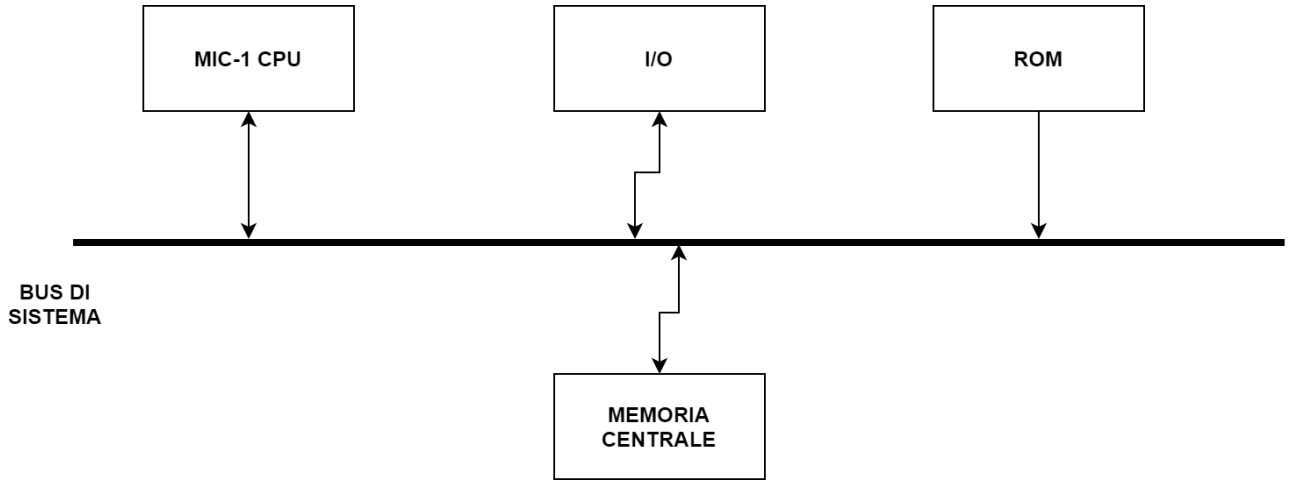
Nome caso d'uso	AssemblaCodice
Attore primario	Utente
Portata	Livello Utente
Parti interessate ed interessi	Produrre un file eseguibile a partire dal codice scritto
Pre-condizioni	L'utente dispone di un file contenente codice
Post-condizioni	File eseguibile generato
Scenario principale	1) Carica file 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:



La ROM è stata riportata in maniera esplicita, sebbene sia da ritenere un 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.

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 da eseguire prelevandolo dal file system.**

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, nella memoria principale, modificare il microprogramma nella memoria di controllo, utilizzare l'assemblatore e così via. Sarà possibile inoltre interrompere l'esecuzione, riprenderla, terminarla o eseguirla passo per passo. L'esecuzione step by step, che potrà essere fatta in termini di operazioni interne alle microistruzioni, singole microistruzioni, o singole istruzioni dell'ISA, 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).

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.

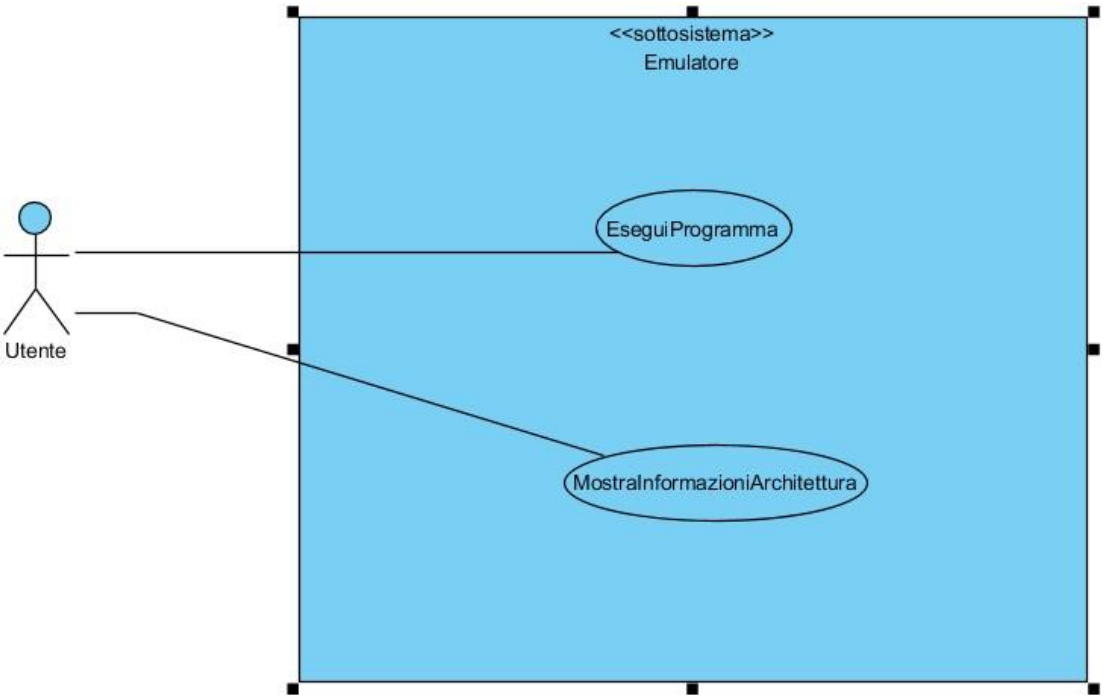
4.1) Prima iterazione

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 ala 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 **("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. In questa prima iterazione si è deciso di

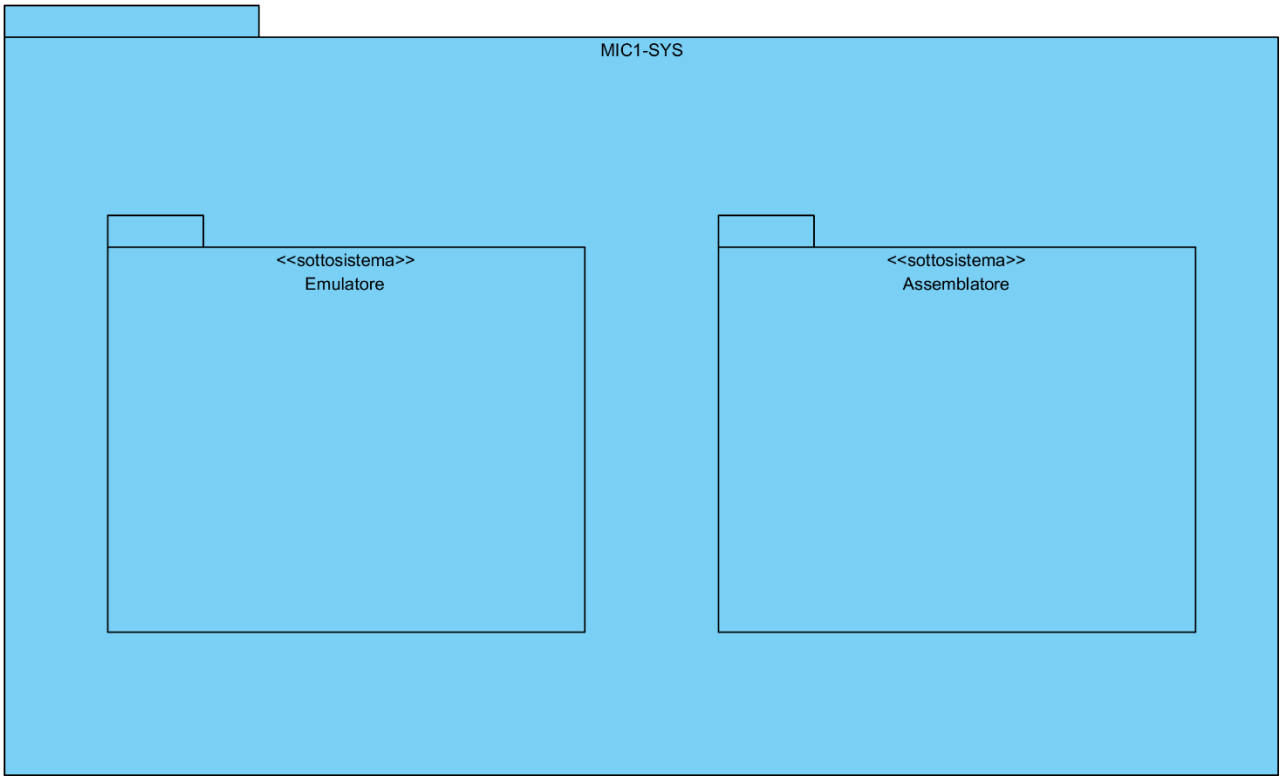
non implementare, nemmeno parzialmente, alcuna funzionalità dell'assemblatore, ragion per cui non è stato prodotto nessun diagramma dei casi d'uso relativo a tale sottosistema/componente.



4.1.2) Architettura logica

Viene per prima cosa effettuata la decomposizione modulare dei due sottosistemi di 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

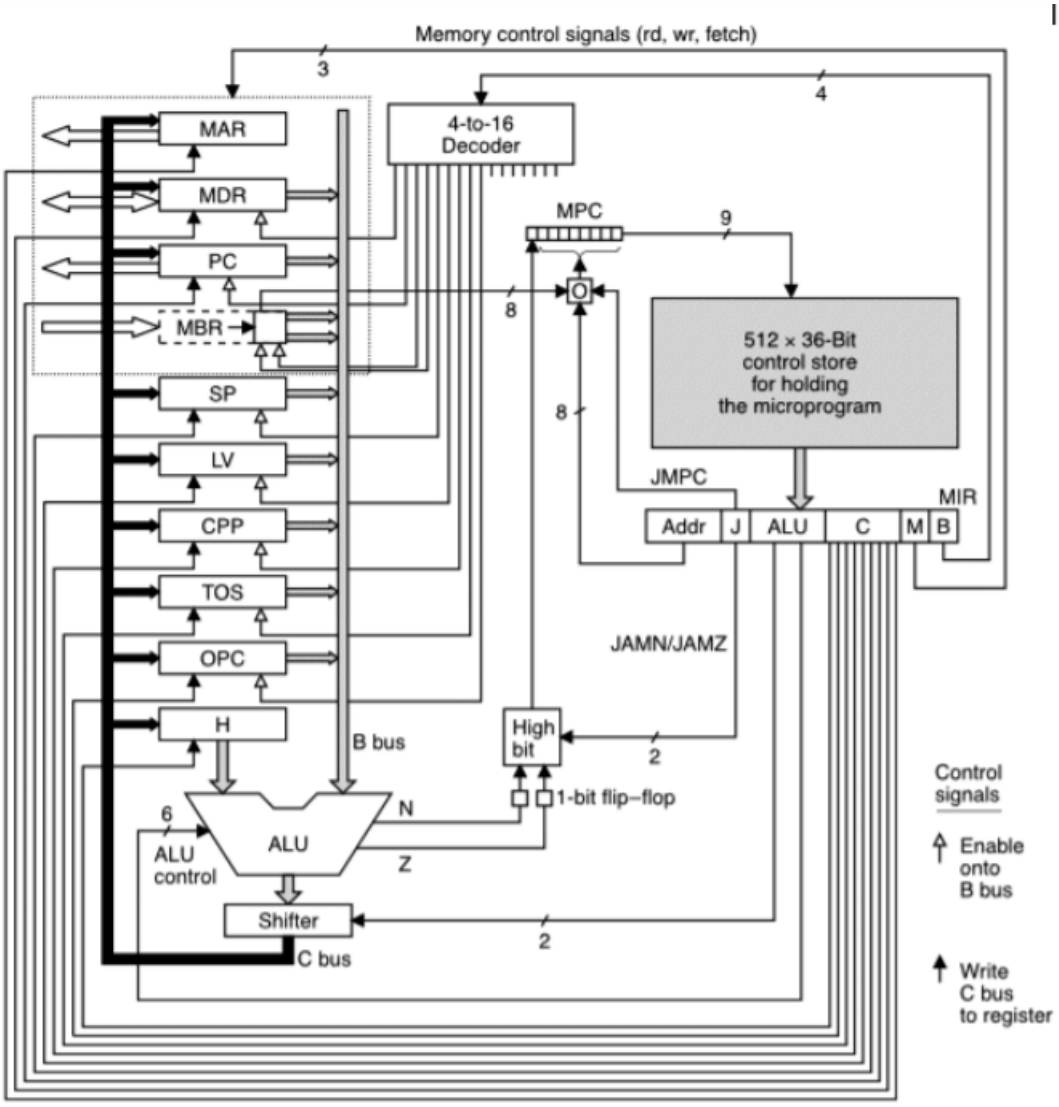
costituiscono il sistema software



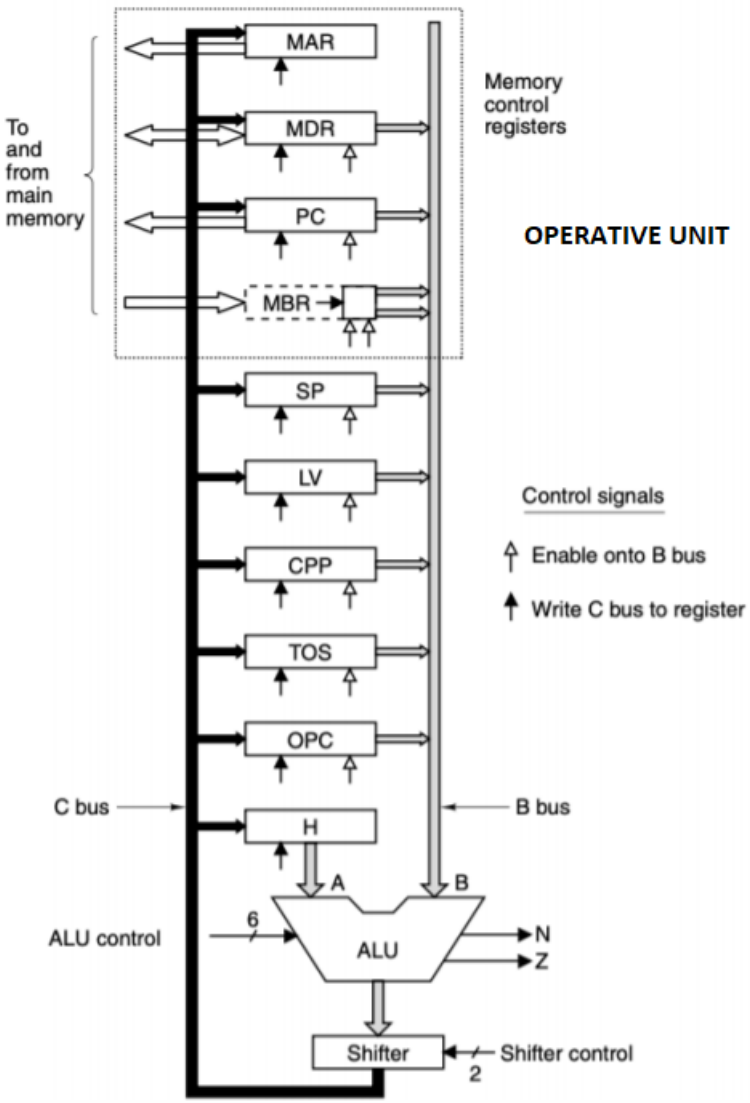
4.1.3) Logica applicativa

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.



UC & UO



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**

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 tempificata 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.

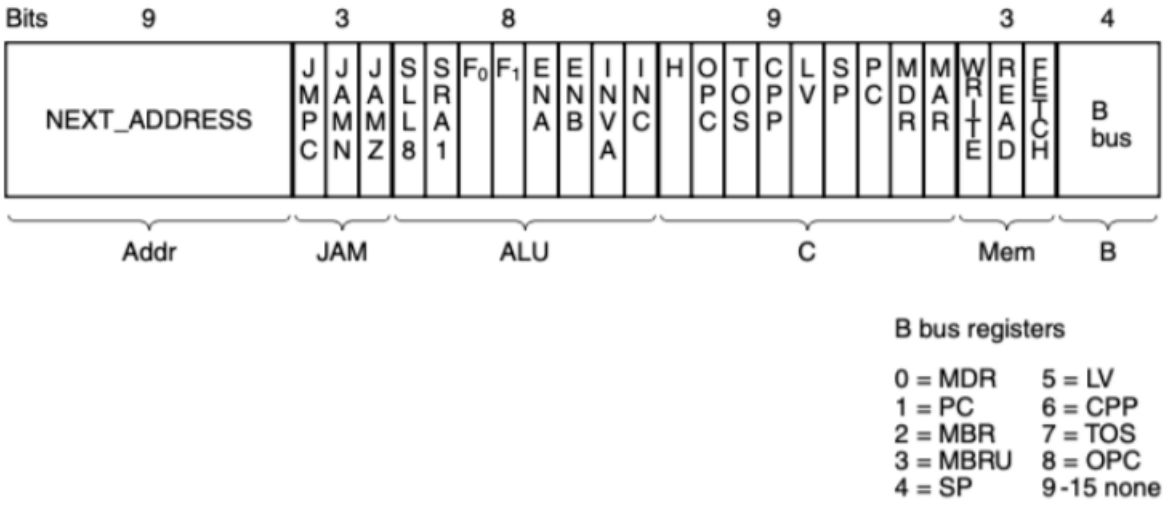
Il modello di programmazione del MIC-1 può essere facilmente delineato con un riferimento ai codici operativi e alle modalità di indirizzamento. Per quanto riguarda il secondo fattore, gli operandi di una istruzione, come già accennato, sono sempre in cima allo stack: siamo ben lontani dall'avere una architettura quasi-ortogonale come invece accade, ad esempio, nel 68000.

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).

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

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

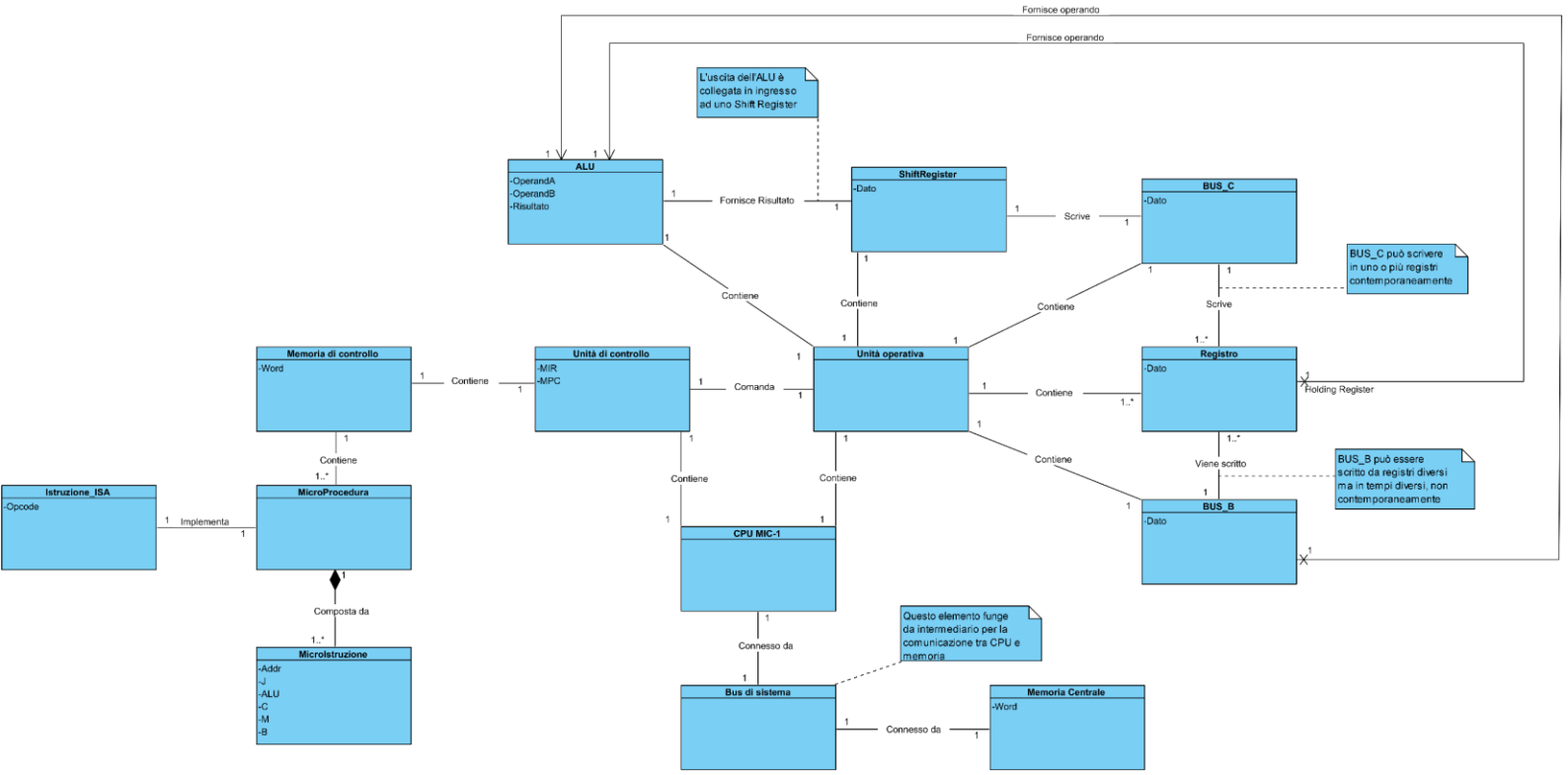


Ogni microistruzione comprende i seguenti campi:

- **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.



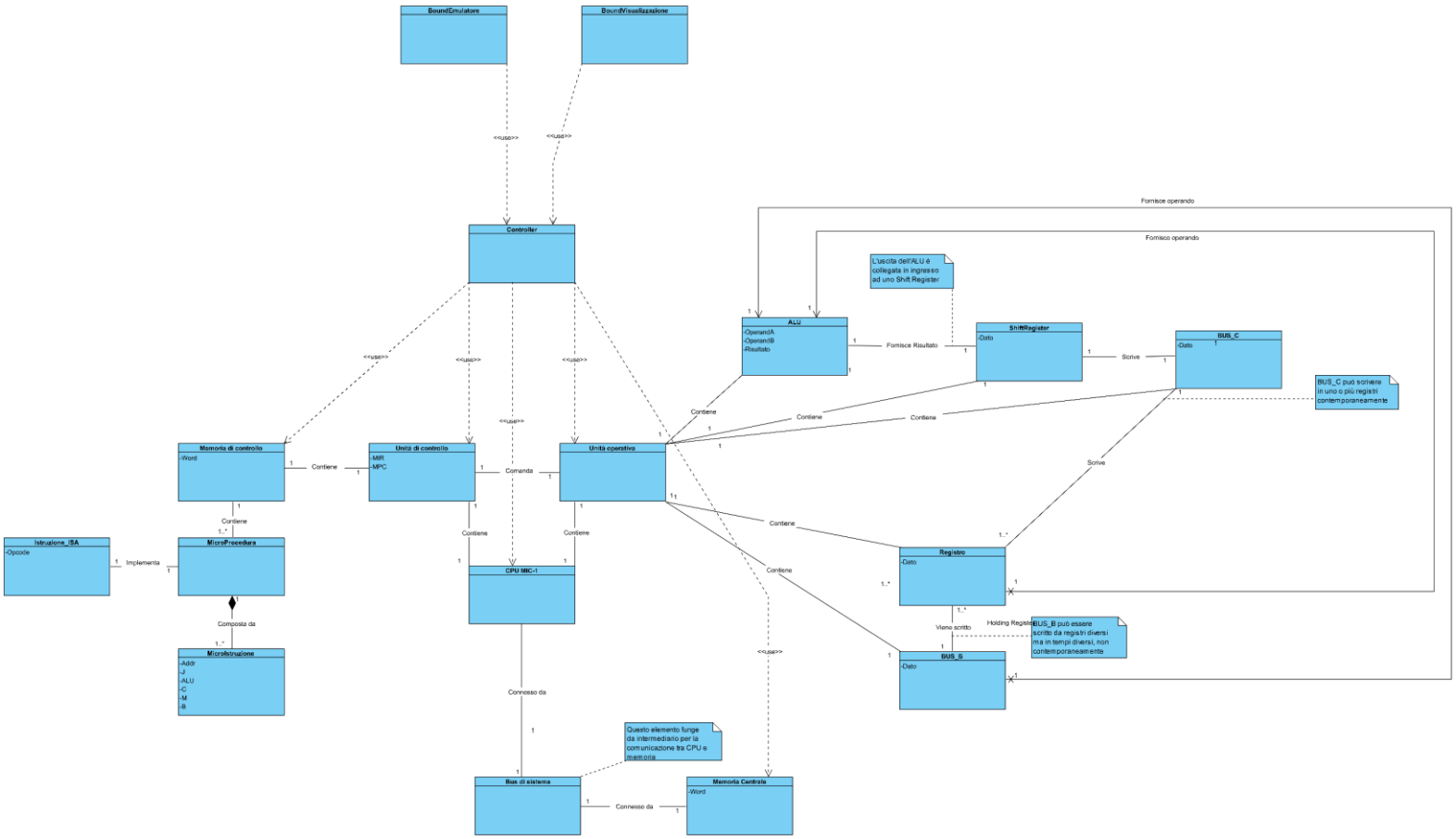
Lo scopo di questo diagramma è quello di catturare le prime entità concettuali appartenenti al dominio applicativo di interesse. Le classi presenti in questo diagramma sono rappresentative di concetti: hanno una propria **intensione**, ossia un proprio significato intrinseco, e una propria **estensione**, 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** – 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 microprogrammata, quindi l'esecuzione di una generica istruzione ISA corrisponde all'esecuzione della sua relativa microprocedura.
- **MicroIstruzione** – 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
- **Istruzione_ISA** – Questa classe rappresenta, in quanto concetto, una operazione supportata dalla CPU e imponibile in maniera diretta alla CPU stessa

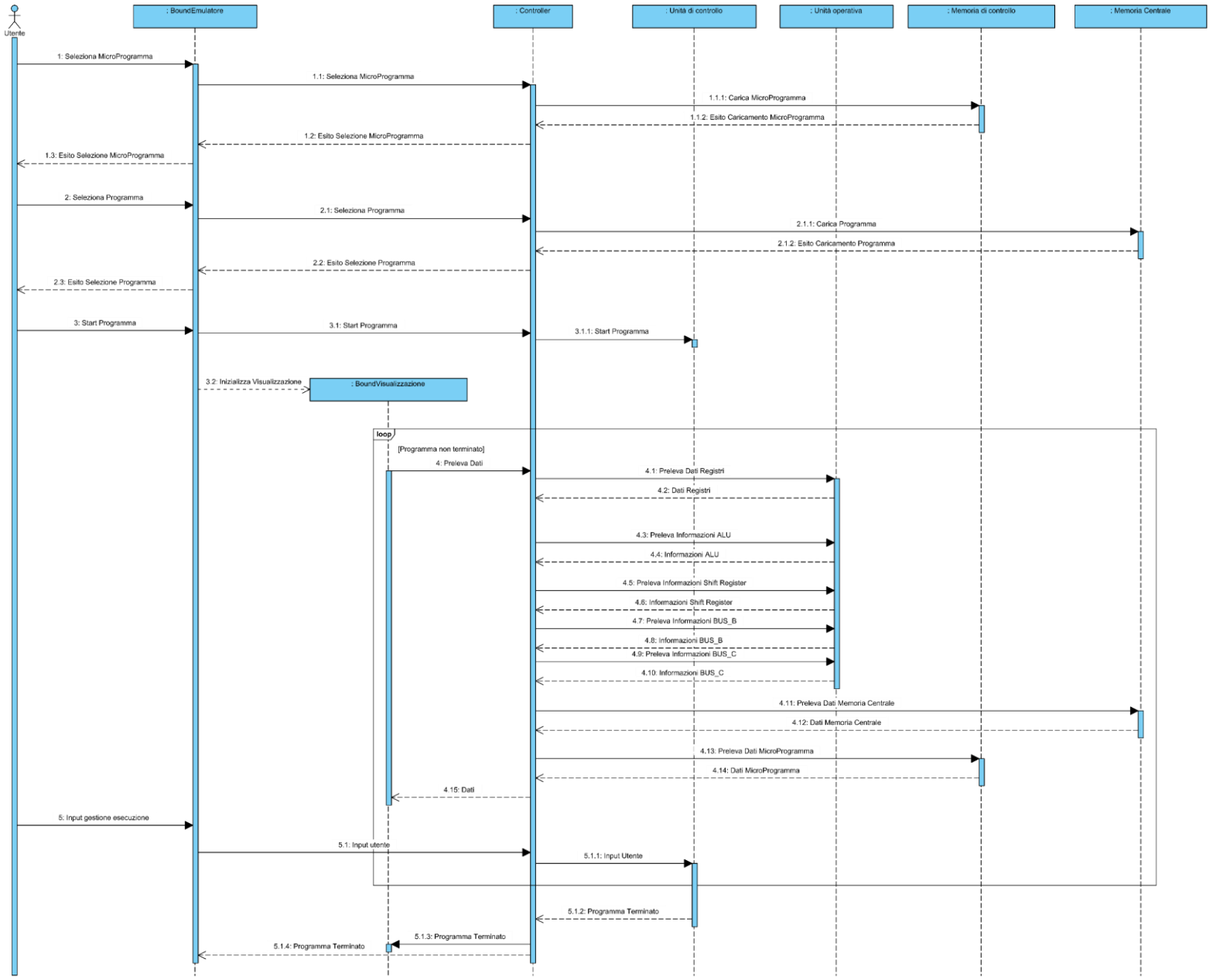
4.1.4) Gestione degli eventi provenienti dall'esterno.

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.



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:



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

(fare seq diag eseguiprogramma prima o poi)

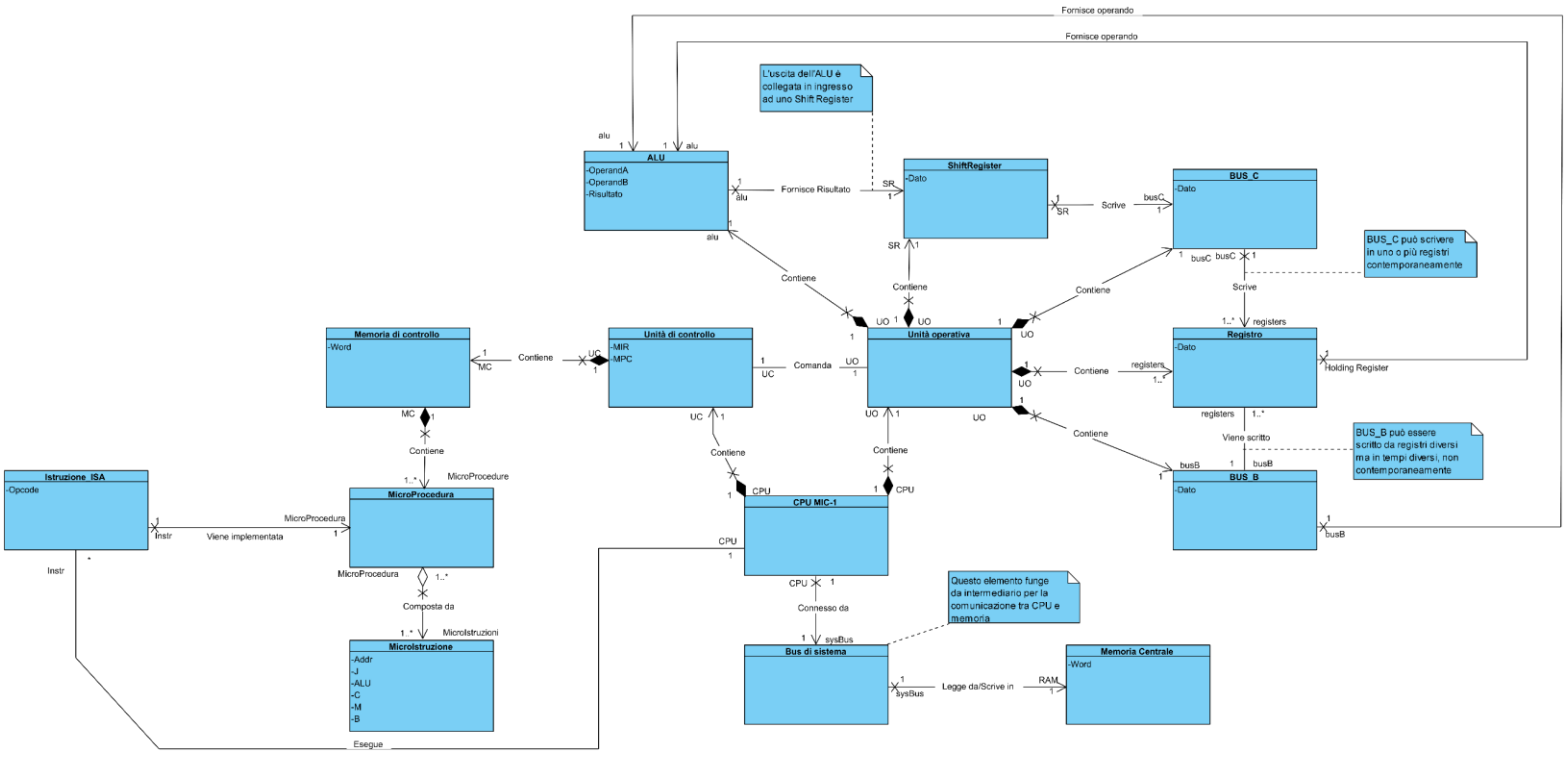
4.2.1) Considerazioni

Come già anticipato, le decisioni di progetto intraprese nella prima iterazione ha reso le modifiche facilmente effettuabili ed ha 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.

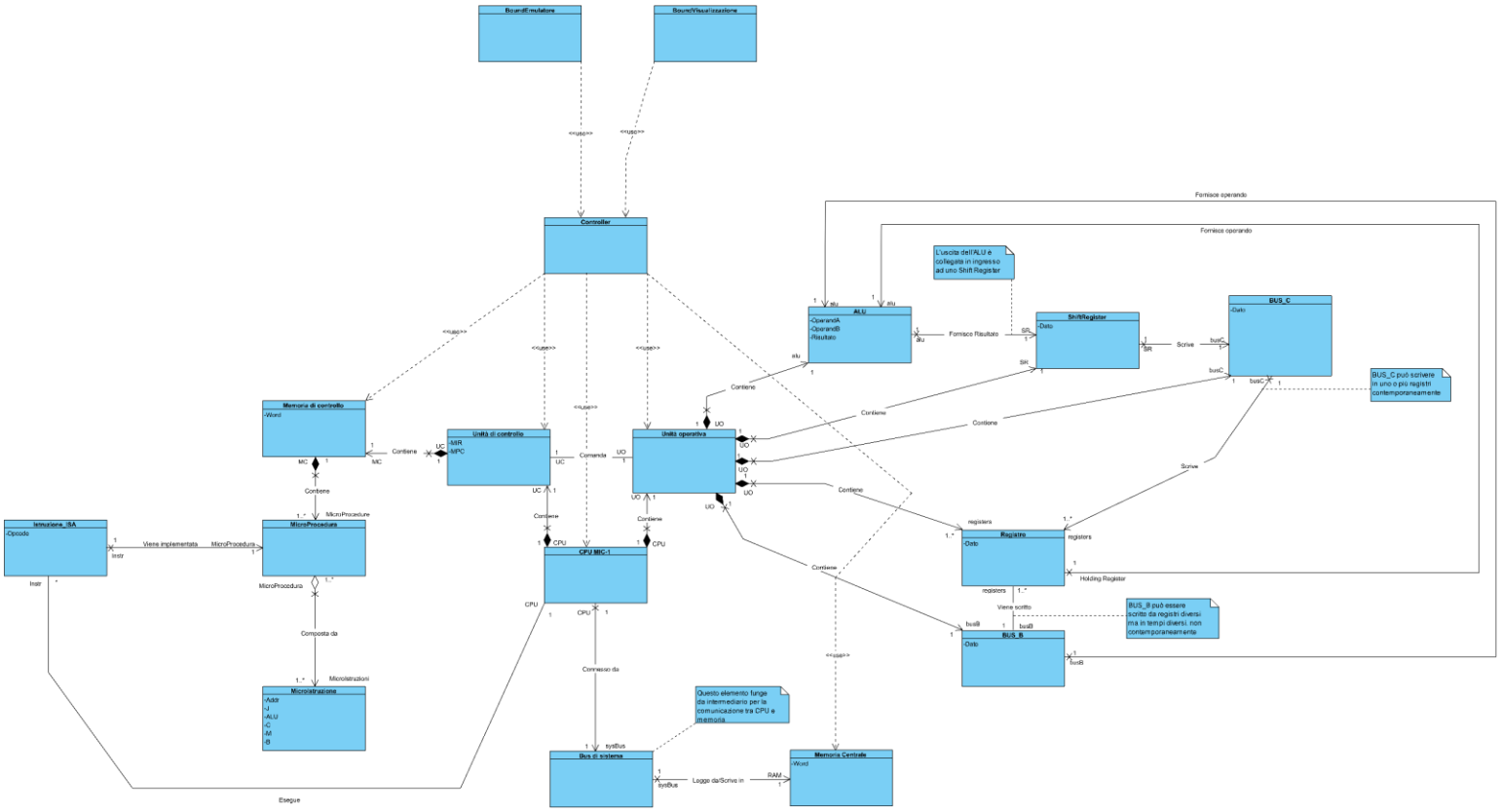
4.2.2) Logica di Business

L'architettura closed layers adottata per il sottosistema emulatore non è stata modificata, così come non è variato lo stile architetturale 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 gioco in maniera più matura e coscenziosa.

Viene di seguito riportato il System Domain Model **("CD_SystemDomainModel")** aggiornato:



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":



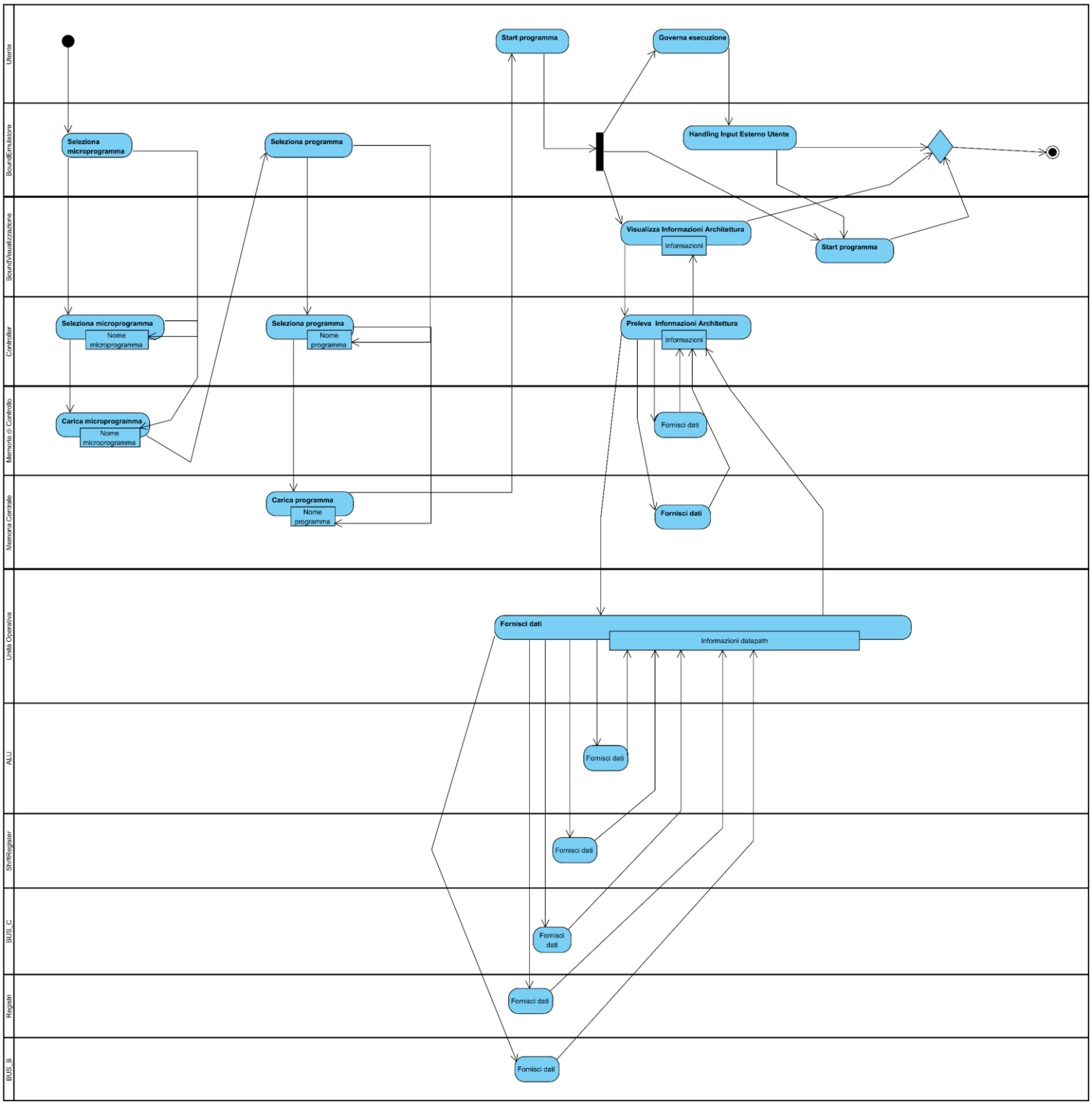
4.2.3) GUI

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, 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:



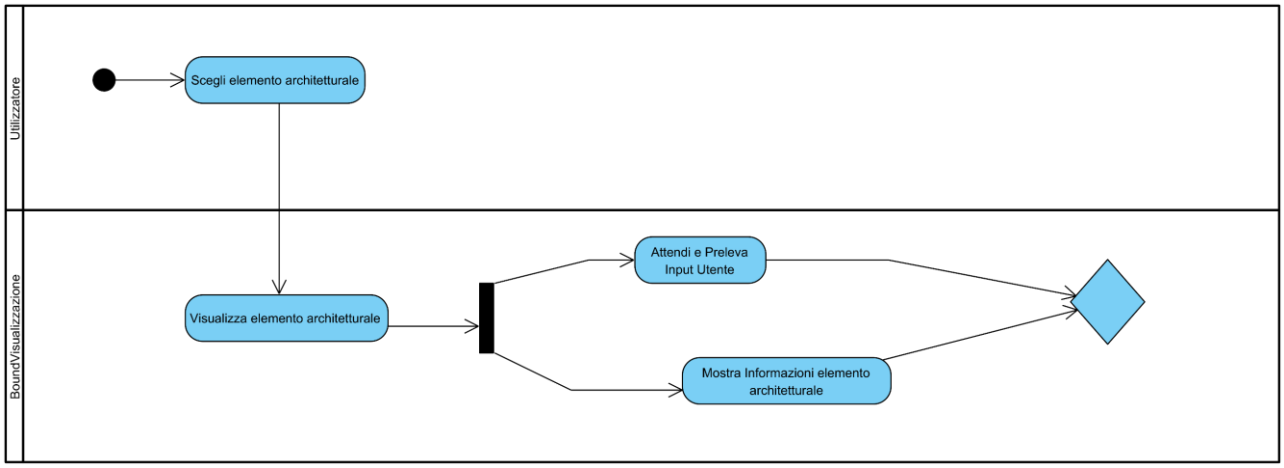
Sempre guardando a tale diagramma di attività, è facile intuire che la classe “BoundEmulatore” deve gestire la selezione del programma e l’esecuzione di tale programma.

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 stato stesso mostrata all'utente.

Si tenga presente che alla classe "BoundVisualizzazione" è consentito dar vita a nuovi flussi di esecuzione concorrenti, ciascuno si occuperà prelevare lo stato di uno specifico elemento dell'architettura hardware emulata e di mostrare le informazioni relative a tale stato all'utilizzatore del sistema, ossia all'utente. 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.



4.2.4) Persistenza: File System

Il secondo requisito funzionale di interesse in questa iterazione riguarda la gestione della persistenza. E' 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:

```

```

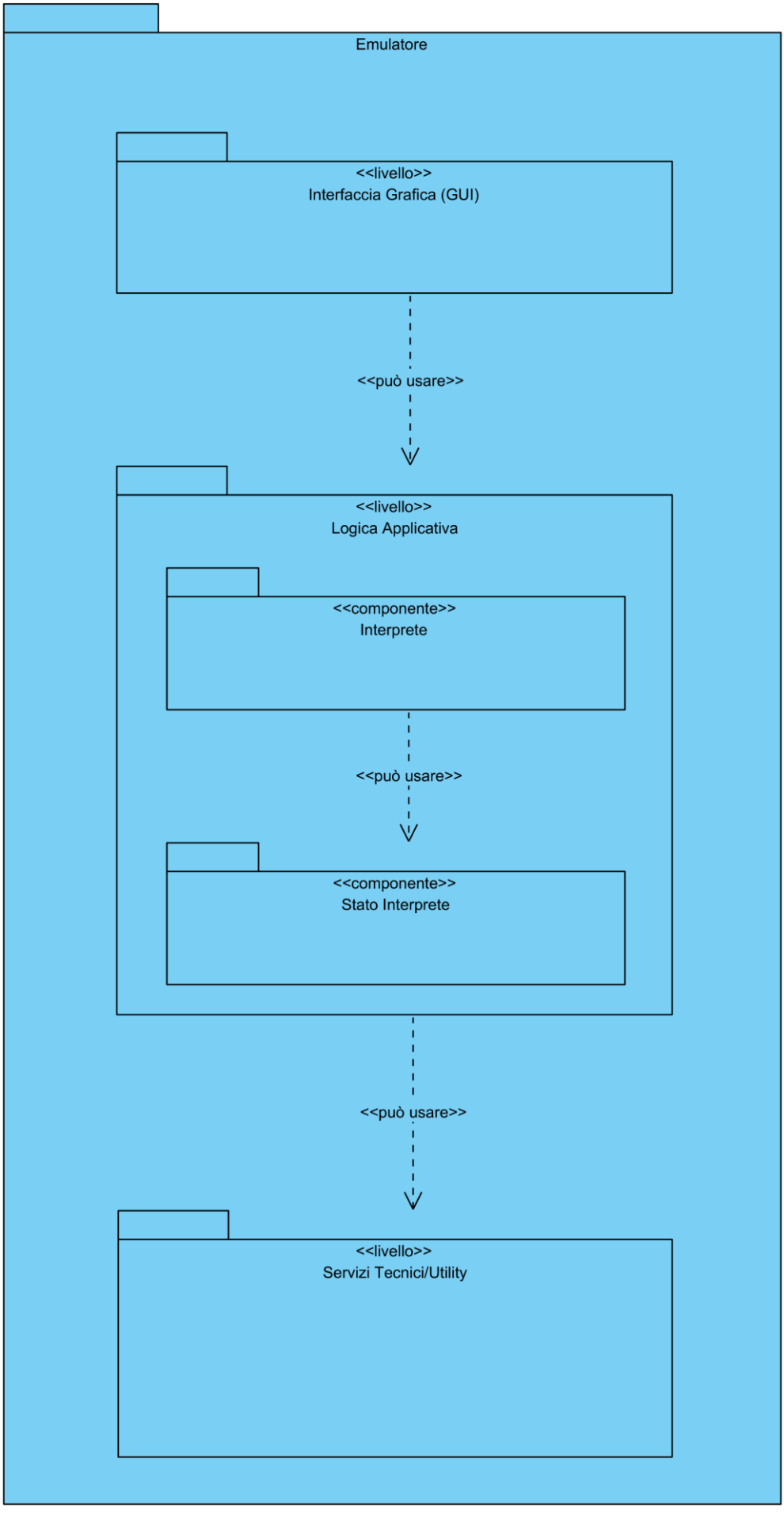
5) Design funzionale e non-funzionale

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 architetturale 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 microprocedure, **nel livello di "logica di bussines" o "logica applicativa" viene adoperato lo stile architetturale 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.

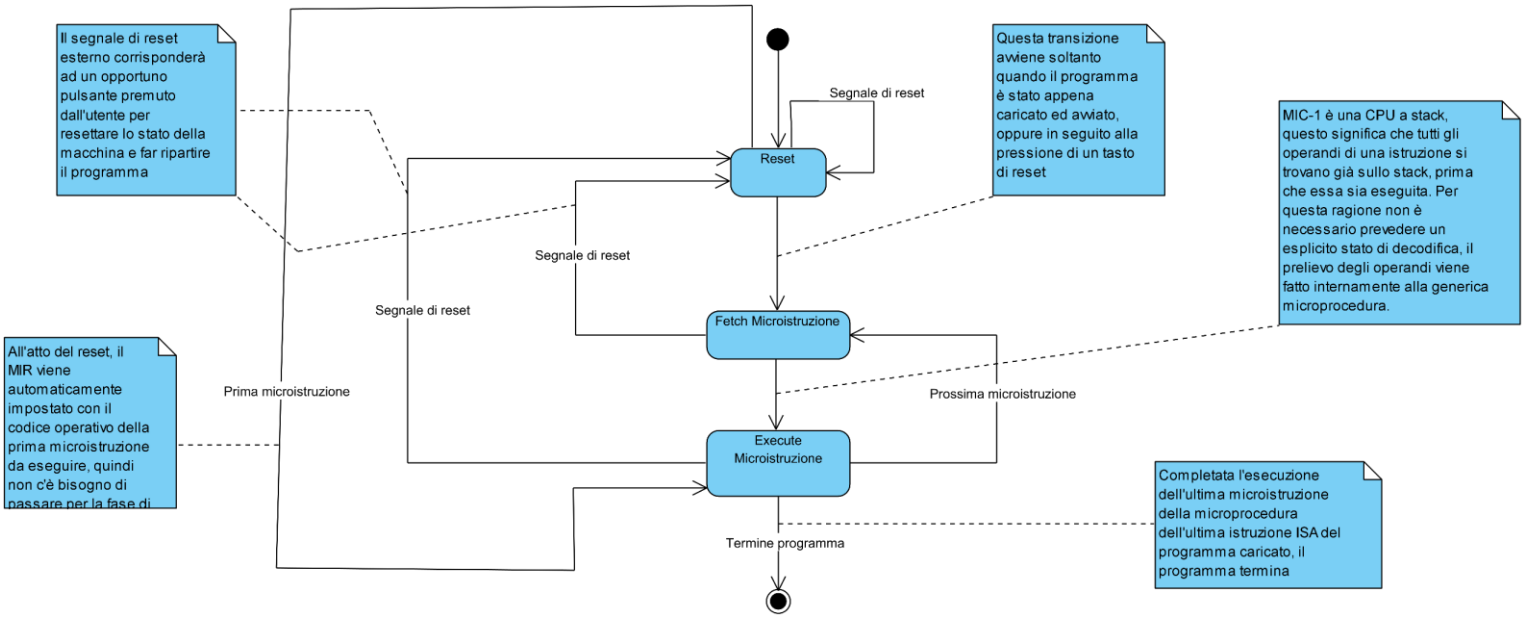

Emulatore
<<livello>>
Interfaccia Grafica (GUI)
<<può usare>>
<<livello>>
Logica Applicativa
<<componente>>
Interprete
<<può usare>>
<<componente>>
Stato Interprete
<<può usare>>
<<livello>>
Servizi Tecnici/Utility

Ritengo che le motivazioni che mi hanno portato a scegliere lo stile architetturale a livelli 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'importante 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.**
- **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 nel file system del sistema operativo su cui l'applicazione è eseguita.**

5.1.2) Raffinamento della dinamica del caso d'uso EseguiProgramma

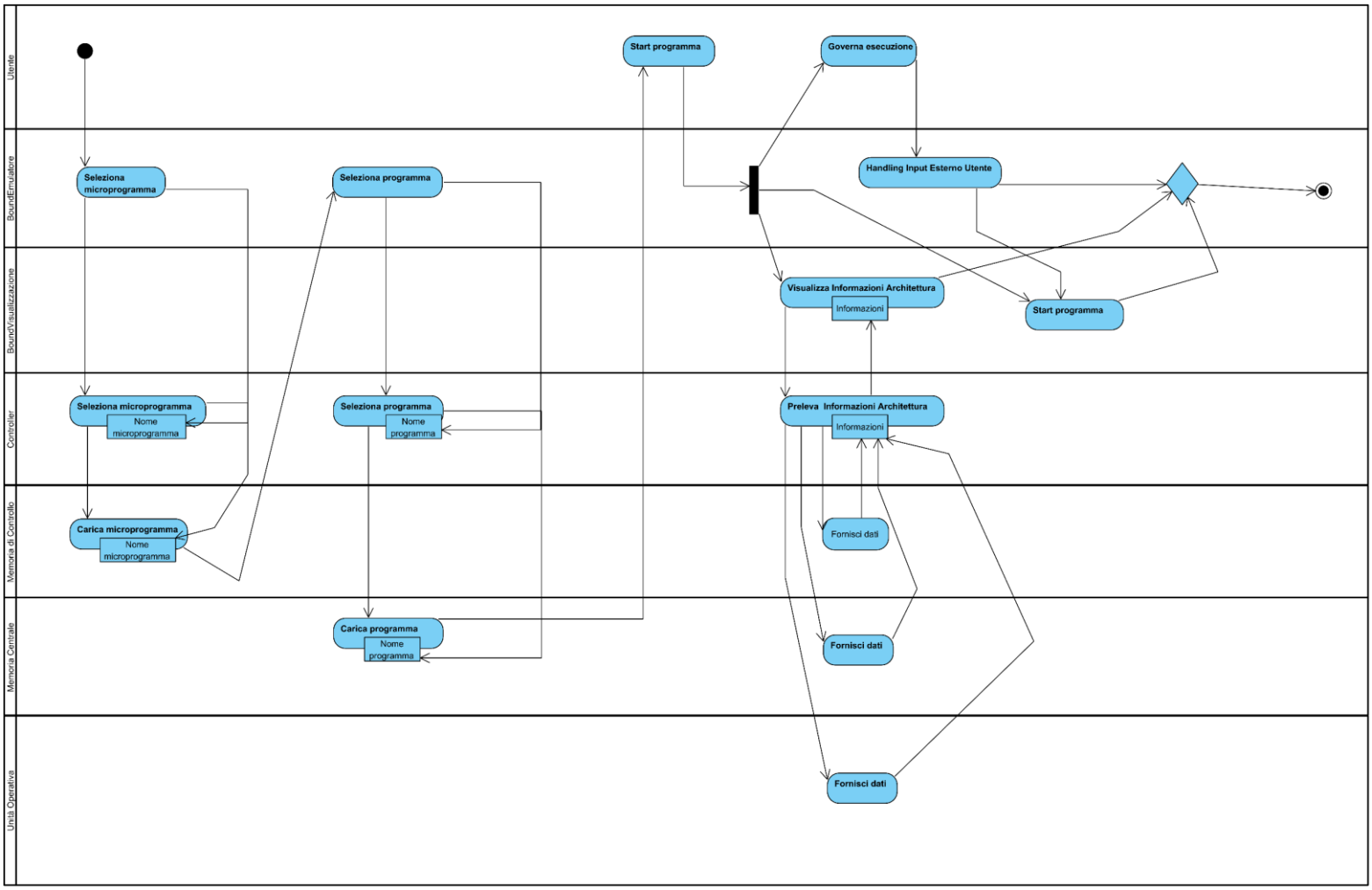
Il passo successivo è stato quello di arricchire l'architettura logica con l'introduzione di nuove classi software, tenendo a mente, per il livello di dominio applicativo, le classi già presentate nei due diagrammi delle classi mostrati in precedenza. **Raffinare la dinamica del caso d'uso considerato in questa iterazione, ossia "EseguiProgramma", 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 architetturale 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").**



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 "passiva", unresponsive.

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



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.**

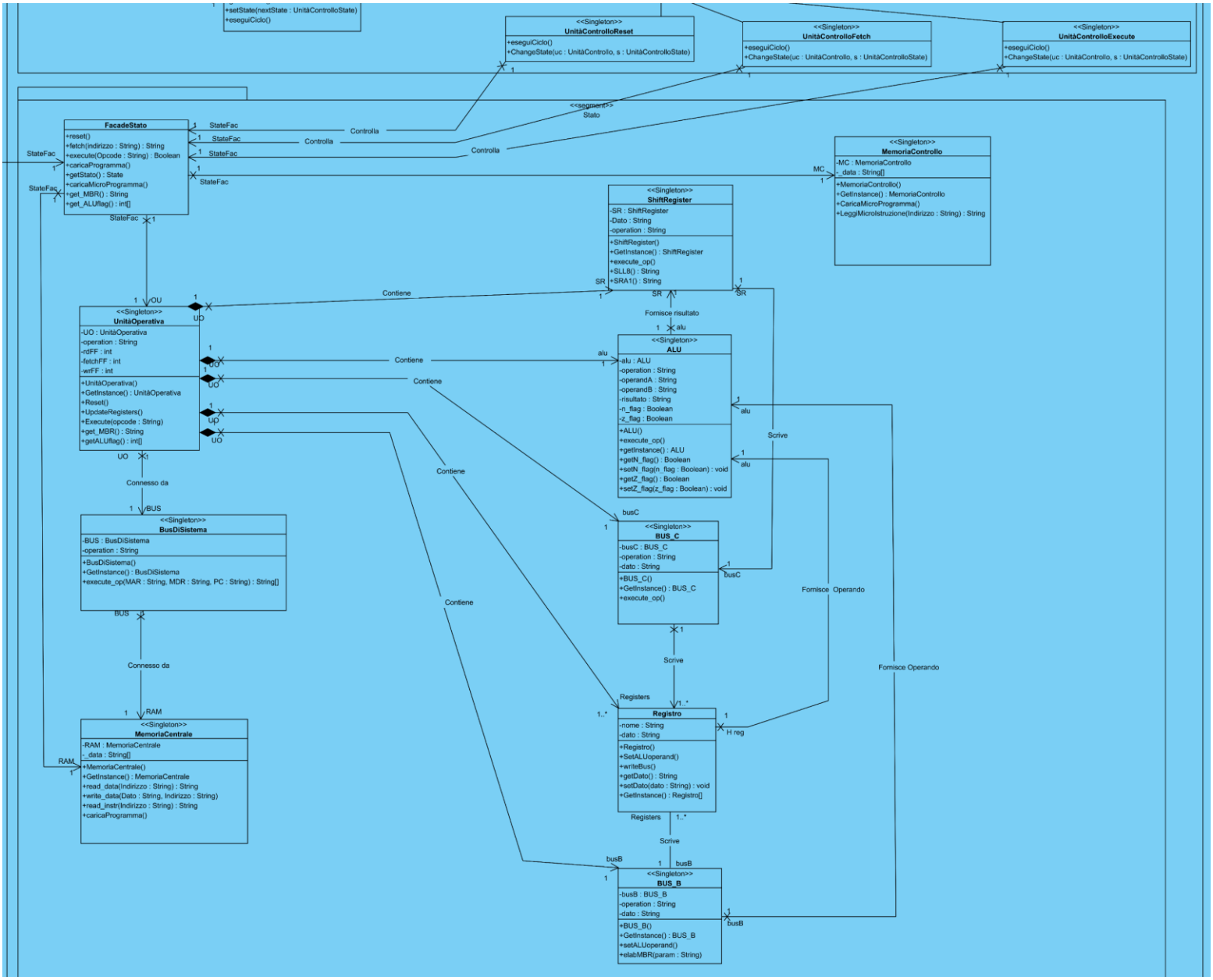
Adesso è possibile passare al class diagram di dettaglio per il sottosistema "Emulatore".

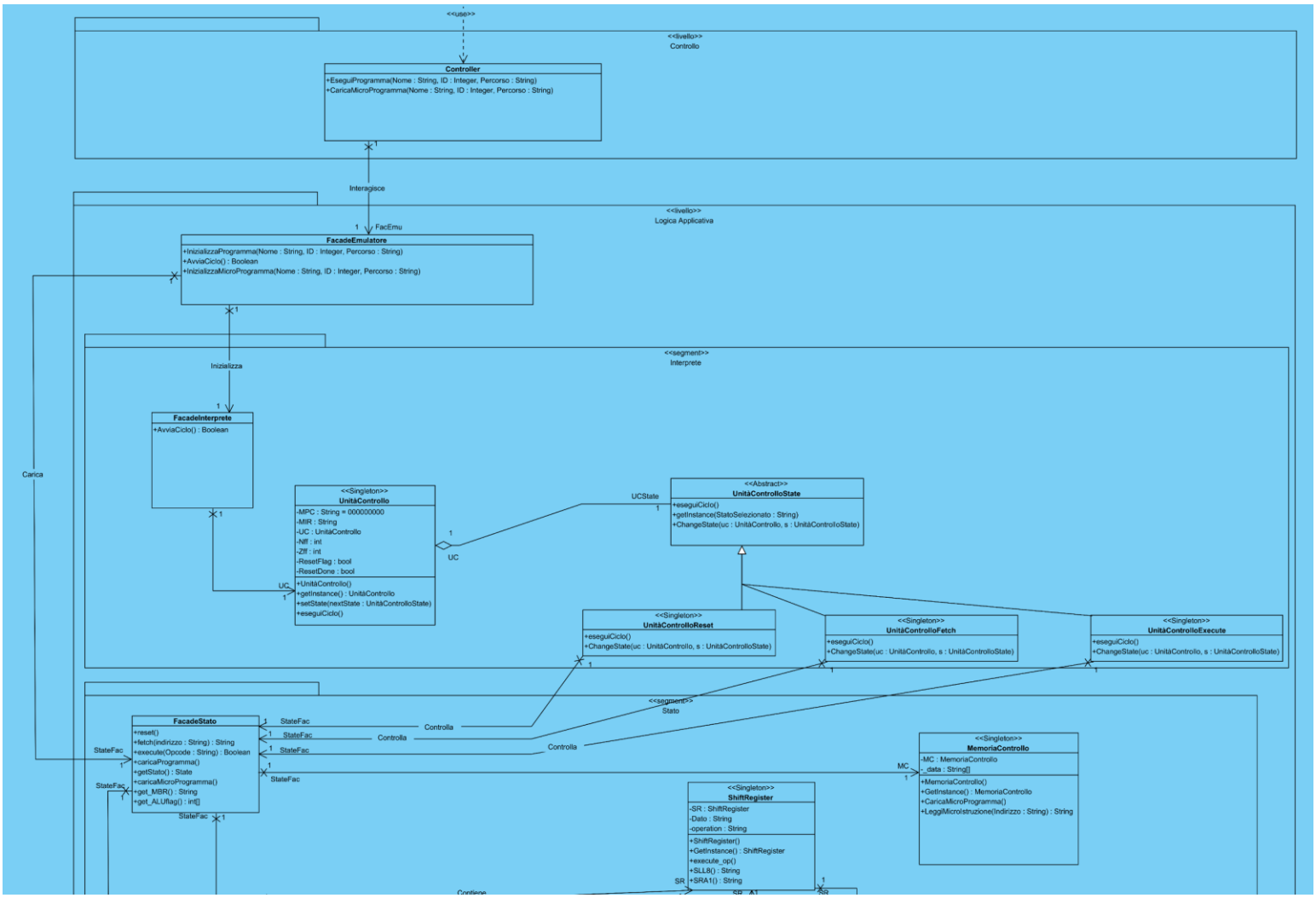
Vengono utilizzati i seguenti design pattern:

- **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_Emulatore").**


+setState(nextState : UnitàControlloState)
+eseguiCiclo()
<<Singleton>>
UnitàControlloReset
+eseguiCiclo()
+ChangeState(uc : UnitàControllo, s : UnitàControlloState)
<<Singleton>>
UnitàControlloFetch
+eseguiCiclo()
+ChangeState(uc : UnitàControllo, s : UnitàControlloState)
<<Singleton>>
UnitàControlloExecute
+eseguiCiclo()
+ChangeState(uc : UnitàControllo, s : UnitàControlloState)
<>
Stato
FacadeStato
+reset()
+fetch(indirizzo : String) : String
+execute(Opcode : String) : Boolean
+caricaProgramma()
+getStato() : State
+caricaMicroProgramma()
+get_MBR() : String
+get_ALUflag() : int[]
StateFac
Controlla
<<Singleton>>
MemoriaControllo
-MC : MemoriaControllo
-_data : String[]
+MemoriaControllo()
+GetInstance() : MemoriaControllo
+CaricaMicroProgramma()
+LeggiMicroIstruzione(Indirizzo : String) : String
MC
<<Singleton>>
ShiftRegister
-SR : ShiftRegister
-Dato : String
-operation : String
+ShiftRegister()
+GetInstance() : ShiftRegister
+execute_op()
+SLL8() : String
+SRA1() : String
SR
Contiene
Fornisce risultato
<<Singleton>>
UnitàOperativa
-UO : UnitàOperativa
-operation : String
-rdFF : int
-fetchFF : int
-wrFF : int
+UnitàOperativa()
+GetInstance() : UnitàOperativa
+Reset()
+UpdateRegisters()
+Execute(opcode : String)
+get_MBR() : String
+getALUflag() : int[]
UO
Connesso da
<<Singleton>>
ALU
-alu : ALU
-operation : String
-operandA : String
-operandB : String
-risultato : String
-n_flag : Boolean
-z_flag : Boolean
+ALU()
+execute_op()
+getInstance() : ALU
+getN_flag() : Boolean
+setN_flag(n_flag : Boolean) : void
+getZ_flag() : Boolean
+setZ_flag(z_flag : Boolean) : void
alu
Scrive
<<Singleton>>
BusDiSistema
-BUS : BusDiSistema
-operation : String
+BusDiSistema()
+GetInstance() : BusDiSistema
+execute_op(MAR : String, MDR : String, PC : String) : String[]
BUS
<<Singleton>>
BUS_C
-busC : BUS_C
-operation : String
-dato : String
+BUS_C()
+GetInstance() : BUS_C
+execute_op()
busC
Fornisce Operando
Registers
Registro
-nome : String
-dato : String
+Registro()
+SetALUoperand()
+writeBus()
+getDato() : String
+setDato(dato : String) : void
+GetInstance() : Registro[]
H reg
<<Singleton>>
MemoriaCentrale
-RAM : MemoriaCentrale
-_data : String[]
+MemoriaCentrale()
+GetInstance() : MemoriaCentrale
+read_data(Indirizzo : String) : String
+write_data(Dato : String, Indirizzo : String)
+read_instr(Indirizzo : String) : String
+caricaProgramma()
RAM
busB
<<Singleton>>
BUS_B
-busB : BUS_B
-operation : String
-dato : String
+BUS_B()
+GetInstance() : BUS_B
+setALUoperand()
+elabMBR(param : String)

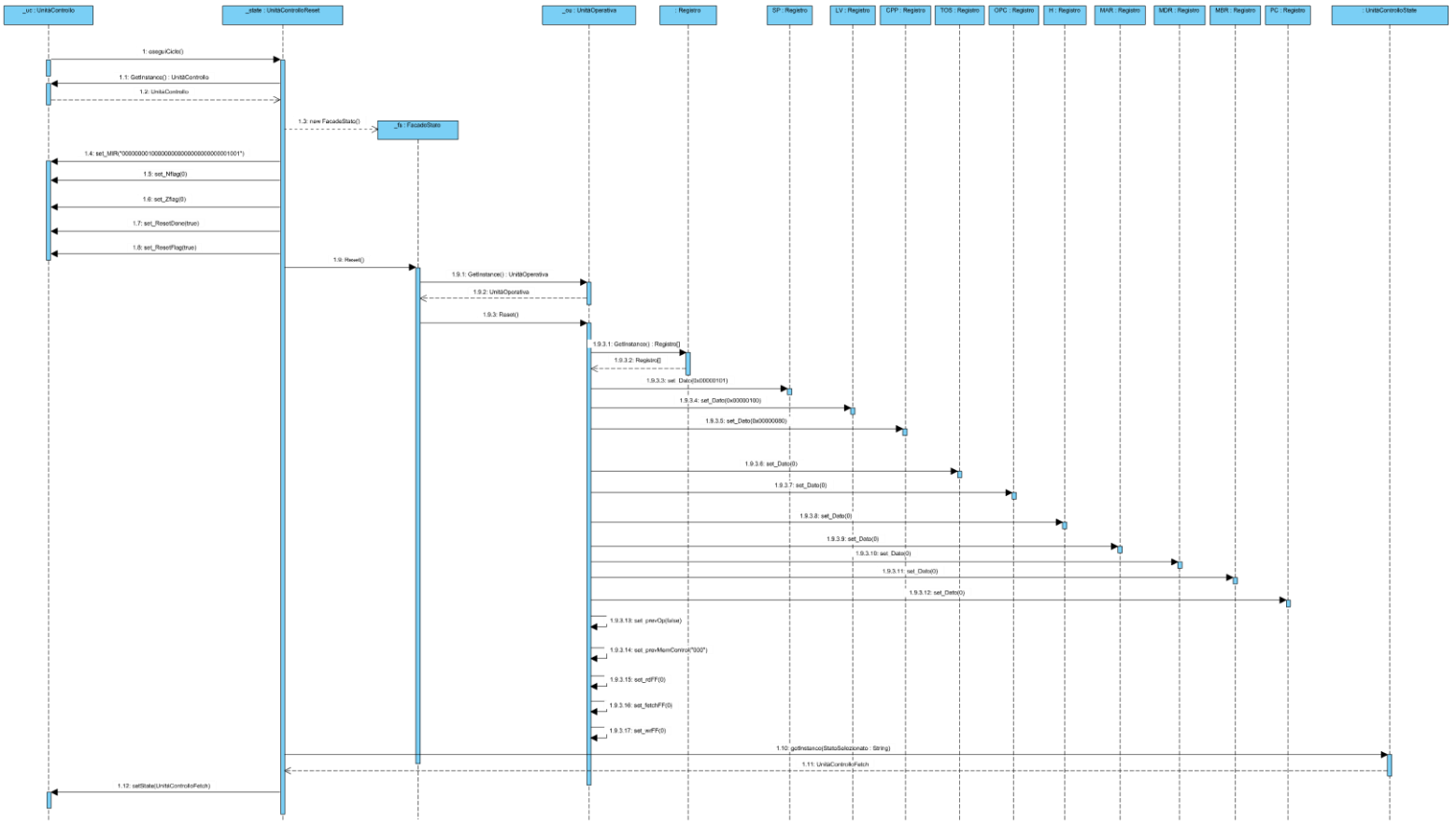

<<use>>
<<livello>>
Controllo
Controller
+EseguiProgramma(Nome : String, ID : Integer, Percorso : String)
+CaricaMicroProgramma(Nome : String, ID : Integer, Percorso : String)
Interagisce
<<livello>>
Logica Applicativa
FacEmu
FacadeEmulatore
+InizializzaProgramma(Nome : String, ID : Integer, Percorso : String)
+AvviaCiclo() : Boolean
+InizializzaMicroProgramma(Nome : String, ID : Integer, Percorso : String)
Inizializza
<>
Interprete
FacadeInterprete
+AvviaCiclo() : Boolean
Carica
<<Singleton>>
UnitàControllo
-MPC : String = 000000000
-MIR : String
-UC : UnitàControllo
-Nff : int
-Zff : int
-ResetFlag : bool
-ResetDone : bool
+UnitàControllo()
+getInstance() : UnitàControllo
+setState(nextState : UnitàControlloState)
+eseguiCiclo()
UC
UCState
<<Abstract>>
UnitàControlloState
+eseguiCiclo()
+getInstance(StatoSelezionato : String)
+ChangeState(uc : UnitàControllo, s : UnitàControlloState)
<<Singleton>>
UnitàControlloReset
+eseguiCiclo()
+ChangeState(uc : UnitàControllo, s : UnitàControlloState)
<<Singleton>>
UnitàControlloFetch
+eseguiCiclo()
+ChangeState(uc : UnitàControllo, s : UnitàControlloState)
<<Singleton>>
UnitàControlloExecute
+eseguiCiclo()
+ChangeState(uc : UnitàControllo, s : UnitàControlloState)
<>
Stato
FacadeStato
+reset()
+fetch(indirizzo : String) : String
+execute(Opcode : String) : Boolean
+caricaProgramma()
+getStato() : State
+caricaMicroProgramma()
+get_MBR() : String
+get_ALUflag() : int[]
StateFac
Controlla
MC
<<Singleton>>
MemoriaControllo
-MC : MemoriaControllo
- data : String[]
+MemoriaControllo()
+GetInstance() : MemoriaControllo
+CaricaMicroProgramma()
+LeggiMicroIstruzione(Indirizzo : String) : String
<<Singleton>>
ShiftRegister
-SR : ShiftRegister
-Dato : String
-operation : String
+ShiftRegister()
+GetInstance() : ShiftRegister
+execute_op()
+SLL8() : String
+SRA1() : String
SR

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.**

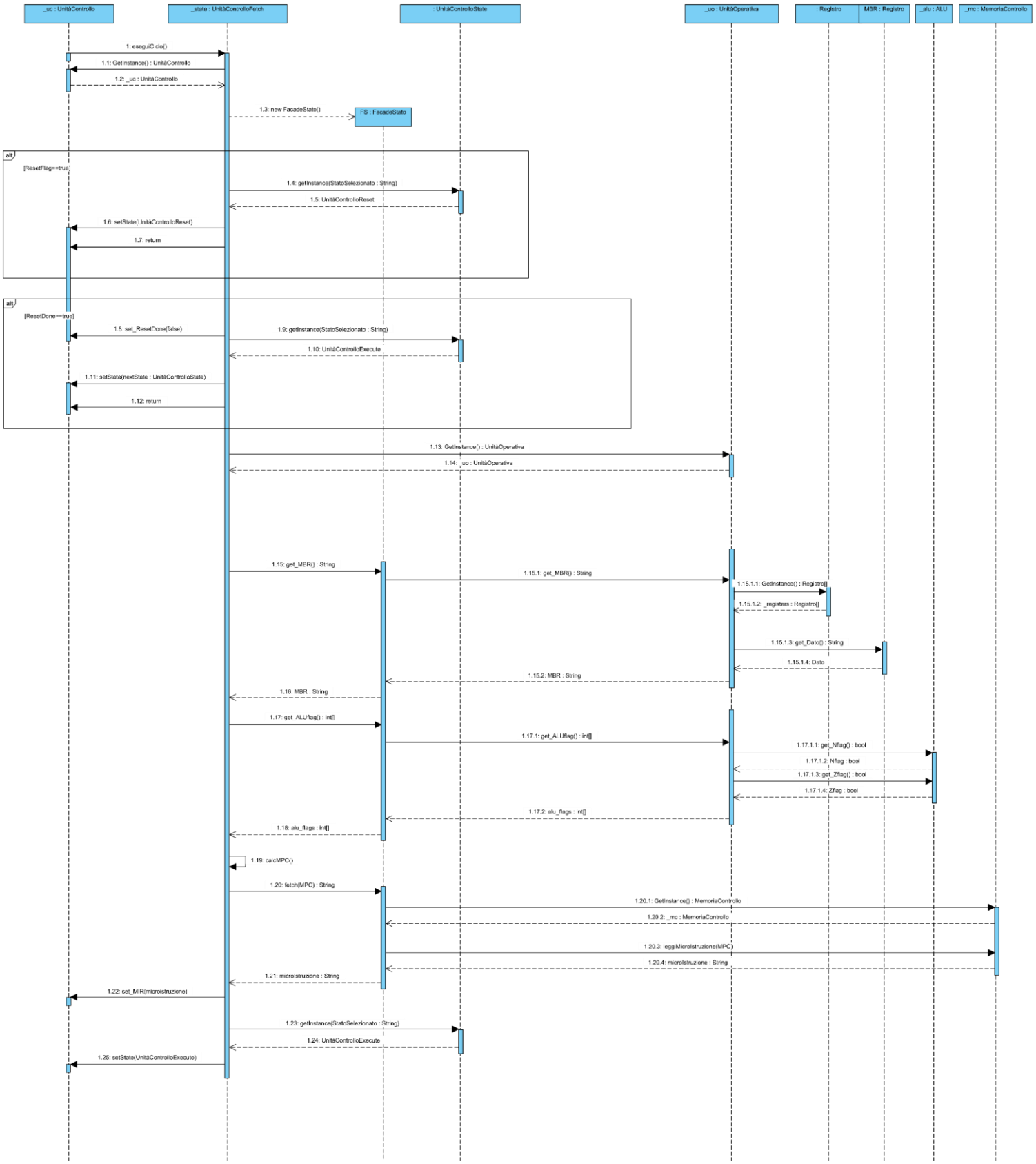
Infine, si osservi che è presente un **livello denominato "Controllo"**: esso possiede le responsabilità relative alla gestione degli input dell'utente inviati dall'interfaccia grafica verso gli altri livelli dell'applicazione.

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

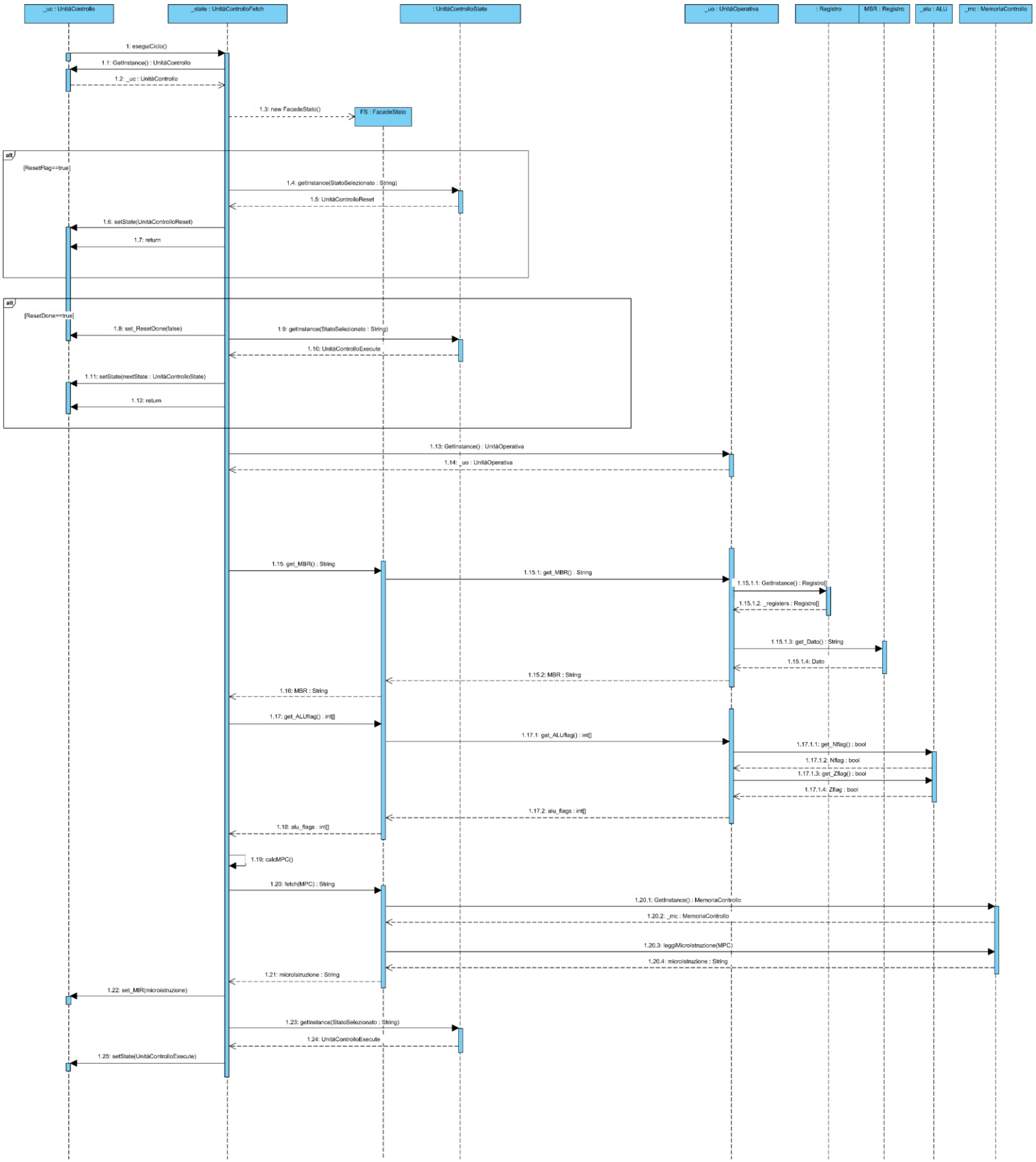
"Seq_Diag_reset_eseguiCiclo":



“Seq_Diag_fetch_eseguiCiclo”:


_uc : UnitàControllo
_state : UnitàControlloFetch
: UnitàControlloState
_uo : UnitàOperativa
: Registro
MBR : Registro
_alu : ALU
_mc : MemoriaControllo
1: eseguiCiclo()
1.1: GetInstance() : UnitàControllo
1.2: _uc : UnitàControllo
1.3: new FacadeStato()
FS : FacadeStato
alt
[ResetFlag==true]
1.4: getInstance(StatoSelezionato : String)
1.5: UnitàControlloReset
1.6: setState(UnitàControlloReset)
1.7: return
alt
[ResetDone==true]
1.8: set_ResetDone(false)
1.9: getInstance(StatoSelezionato : String)
1.10: UnitàControlloExecute
1.11: setState(nextState : UnitàControlloState)
1.12: return
1.13: GetInstance() : UnitàOperativa
1.14: _uo : UnitàOperativa
1.15: get_MBR() : String
1.15.1: get_MBR() : String
1.15.1.1: GetInstance() : Registro[]
1.15.1.2: _registers : Registro[]
1.15.1.3: get_Dato() : String
1.15.1.4: Dato
1.15.2: MBR : String
1.16: MBR : String
1.17: get_ALUflag() : int[]
1.17.1: get_ALUflag() : int[]
1.17.1.1: get_Nflag() : bool
1.17.1.2: Nflag : bool
1.17.1.3: get_Zflag() : bool
1.17.1.4: Zflag : bool
1.17.2: alu_flags : int[]
1.18: alu_flags : Int[]
1.19: calcMPC()
1.20: fetch(MPC) : String
1.20.1: GetInstance() : MemoriaControllo
1.20.2: _mc : MemoriaControllo
1.20.3: leggiMicroIstruzione(MPC)
1.20.4: microIstruzione : String
1.21: microIstruzione : String
1.22: set_MIR(microIstruzione)
1.23: getInstance(StatoSelezionato : String)
1.24: UnitàControlloExecute
1.25: setState(UnitàControlloExecute)

“Seq_Diag_execute_eseguiCiclo”:


_uc : UnitàControllo
_state : UnitàControlloFetch
: UnitàControlloState
_uo : UnitàOperativa
: Registro
MBR : Registro
_alu : ALU
_mc : MemoriaControllo
1: eseguiCiclo()
1.1: GetInstance() : UnitàControllo
1.2: _uc : UnitàControllo
1.3: new FacadeStato()
FS : FacadeStato
alt
[ResetFlag==true]
1.4: getInstance(StatoSelezionato : String)
1.5: UnitàControlloReset
1.6: setState(UnitàControlloReset)
1.7: return
alt
[ResetDone==true]
1.8: set_ResetDone(false)
1.9: getInstance(StatoSelezionato : String)
1.10: UnitàControlloExecute
1.11: setState(nextState : UnitàControlloState)
1.12: return
1.13: GetInstance() : UnitàOperativa
1.14: _uo : UnitàOperativa
1.15: get_MBR() : String
1.15.1: get_MBR() : String
1.15.1.1: GetInstance() : Registro[]
1.15.1.2: _registers : Registro[]
1.15.1.3: get_Dato() : String
1.15.1.4: Dato
1.15.2: MBR : String
1.16: MBR : String
1.17: get_ALUflag() : int[]
1.17.1: get_ALUflag() : int[]
1.17.1.1: get_Nflag() : bool
1.17.1.2: Nflag : bool
1.17.1.3: get_Zflag() : bool
1.17.1.4: Zflag : bool
1.17.2: alu_flags : int[]
1.18: alu_flags : int[]
1.19: calcMPC()
1.20: fetch(MPC) : String
1.20.1: GetInstance() : MemoriaControllo
1.20.2: _mc : MemoriaControllo
1.20.3: leggiMicroIstruzione(MPC)
1.20.4: microIstruzione : String
1.21: microIstruzione : String
1.22: set_MIR(microIstruzione)
1.23: getInstance(StatoSelezionato : String)
1.24: UnitàControlloExecute
1.25: setState(UnitàControlloExecute)

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

5.1.3) Vista "Componenti e Connettori"

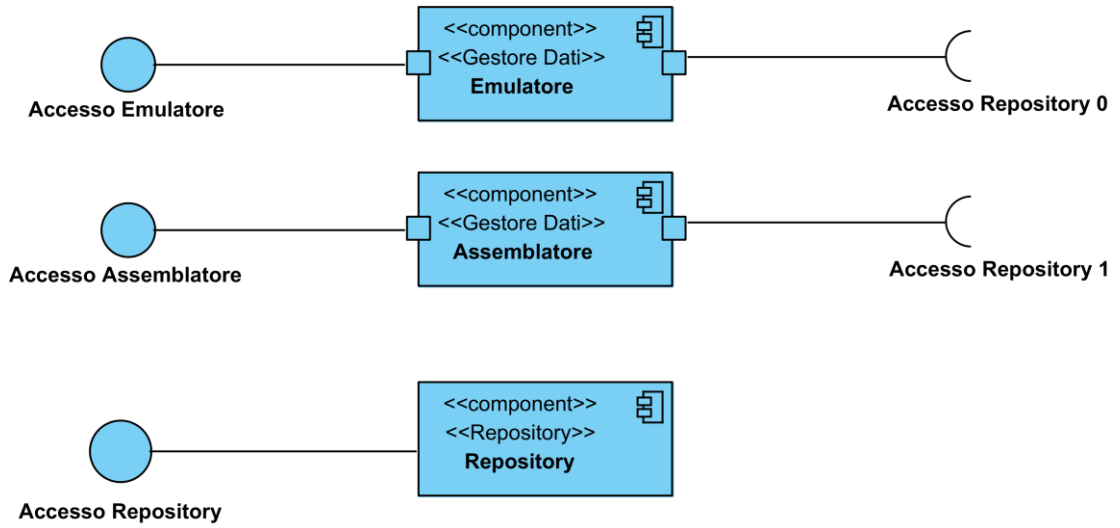
Con la sola vista modulare non riusciamo ad esprimere informazioni significative circa gli elementi che formano il sistema software a tempo di esecuzione. Per questo motivo, bisogna introdurre nella documentazione delle opportune viste e quindi diagrammi, che siano capaci di rappresentare correttamente 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 istante istanze dei tipi presenti nel catalogo.

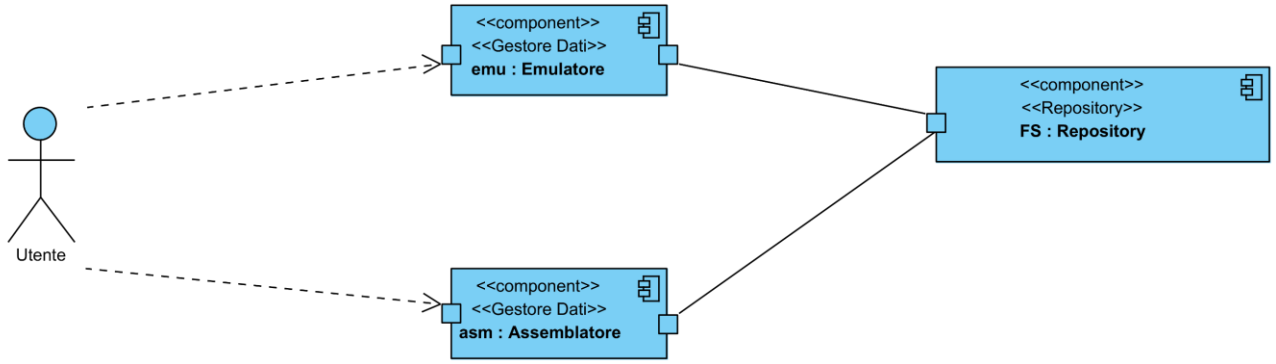
L'idea di base è che il componente assemblatore dovrebbe pubblicare dei programmi compilati in uno spazio dati al quale ha accesso anche l'emulatore. In generale, anche trascurando l'assemblatore, si vuole che il componente emulatore sia dotato di un proprio repository sul quale mantenere i programmi da eseguire. Si utilizza dunque uno stile architetturale 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":



“Comp_Diag_MIC1-SYS”:



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

6) Prospettiva dell'implementazione

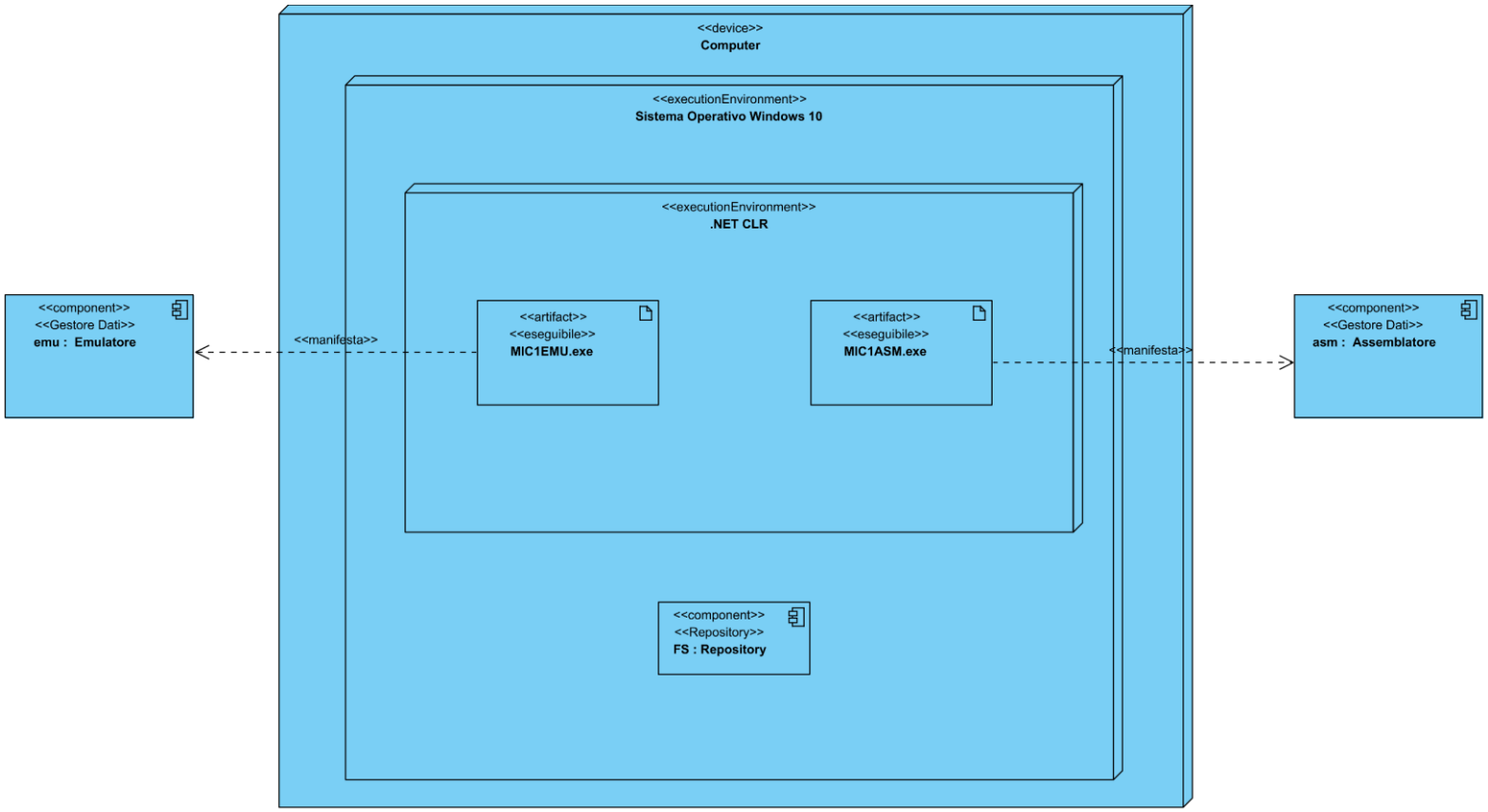
In questa parte della documentazione ci si focalizza su due aspetti del sistema software:

- 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#
- 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.

7) Testing