**ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA**

---

**SCUOLA DI INGEGNERIA E ARCHITETTURA**

Dipartimento di Informatica – Scienza e Ingegneria

*Corso di Laurea Triennale in Ingegneria Informatica*

**TESI DI LAUREA**

in

*Calcolatori Elettronici – T*

# Estensione del simulatore del processore DLX

**CANDIDATO**

*Umberto Laghi*

**RELATORE**

*Stefano Mattoccia*

*Anno Accademico: 2021/2022*

# Indice

# 1 Introduzione

L'obiettivo della tesi è quello di correggere i bug del simulatore ed aggiungere nuove funzionalità allo stesso, partendo da ciò che Fabrizio Maccagnani, Alessandro Foglia, Federico Pomponii e Filippo Comastri hanno fatto prima di me.

Il simulatore nasce con lo scopo di emulare un sistema a processore basato sulle architetture RISC-V e DLX, con l'idea di rendere quanto più concreti e vicini alla realtà gli argomenti affrontati nel corso di Calcolatori Elettronici T, in modo da renderli più comprensibili a tutti gli studenti del corso.

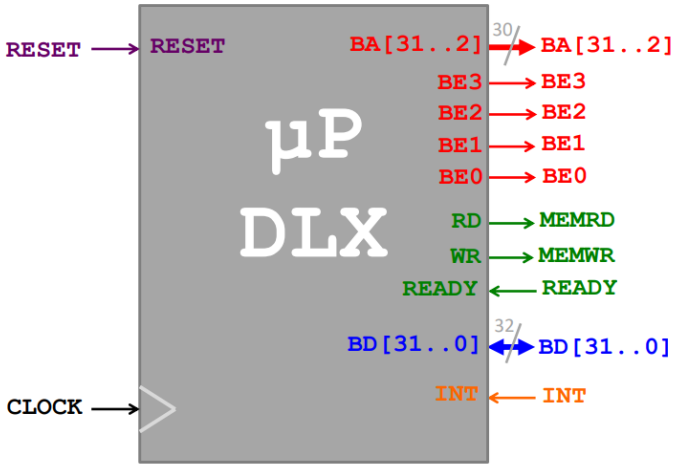
La prima parte della tesi riguarda la correzione dei diversi bug riscontrati nell'utilizzo e che, alle volte, ne rendono frustrante l'utilizzo. La seconda parte prevede l'inserimento di una rappresentazione dinamica dei cicli di bus che esegue il processore DLX per accedere in memoria.

## 1.1 Architettura DLX

DLX è un'architettura per microprocessori sviluppata da John L. Hennessy e David A. Patterson, partendo dall'architettura MIPS e semplificandola per scopi didattici. Lo studio del DLX è molto diffuso, perché il suo ISA è molto simile a quello che è realmente implementato nei processori basati su RISC-V [1].

L'ISA del DLX prevede 32 registri da 32bit ciascuno (R0-R31): R0 è cablato a 0, R31 viene usato per salvare l'indirizzo di ritorno per le istruzioni che lo prevedono, mentre tutti gli altri registri sono *general purpose*. Inoltre, sono presenti anche altri registri speciali, sempre a 32 bit, a cui il programmatore non può accedere. Questi registri sono: il Program Counter (PC), che memorizza l'indirizzo dell'istruzione successiva di cui fare il fetch; l'Interrupt Address Register (IAR), che contiene l'indirizzo di ritorno al main nel caso della ricezione di una interruzione o interrupt; il Memory Address Register (MAR), che preserva l'indirizzo di memoria in cui vogliamo fare una lettura o scrittura; il Memory Data Register (MDR), che contiene il dato appena letto o da scrivere [2].

Come si può notare dall'immagine sotto riportata, il DLX comunica con il resto del sistema tramite bus, poiché basato sul modello di Von Neumann [3].

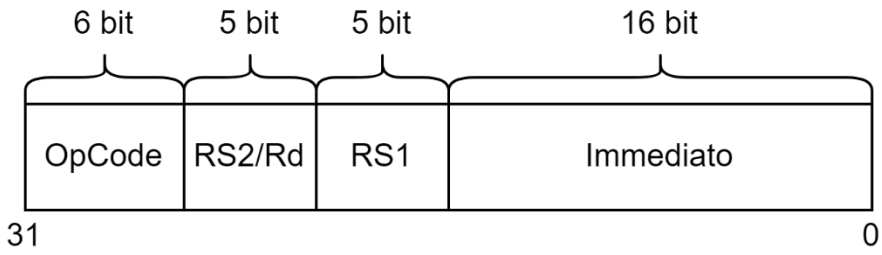



Il processore DLX ai “morsetti” [4]

### 1.1.1 Istruzioni DLX

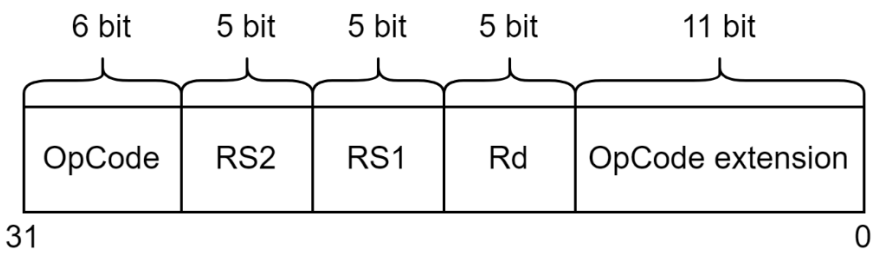
L’ISA del DLX prevede 42 istruzioni, divise in 3 gruppi distinti dal formato.

Al gruppo I appartengono tutte quelle istruzioni che nella firma hanno un immediato, ovvero un valore di 4 byte espresso in formato esadecimale.



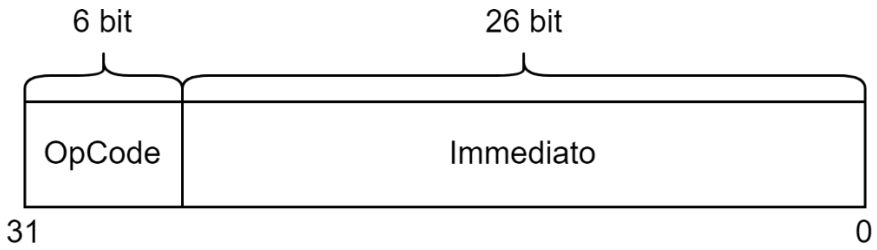

L’OpCode è la stringa di 6 bit associata ad ogni istruzione, RS2/Rd indica il secondo registro sorgente oppure il registro destinazione, ovvero quello in cui verrà salvato il risultato, RS1 è il registro sorgente.

Il gruppo R riguarda tutte le istruzioni che operano tra registri.



RS1 ed RS2 sono i registri sorgente da cui vengono prelevate le stringhe di bit, OpCpde extension è una stringa di 11 bit utilizzabile per fare un'ulteriore differenziazione tra le istruzioni nel caso in cui l'ISA venga esteso.

Il gruppo di istruzioni J comprende le istruzioni di salto incondizionato con o senza ritorno con l'utilizzo di un immediato.




In questo caso l'immediato ha lunghezza di 26 bit, perché i bit occupati dai registri non sono più utilizzati.

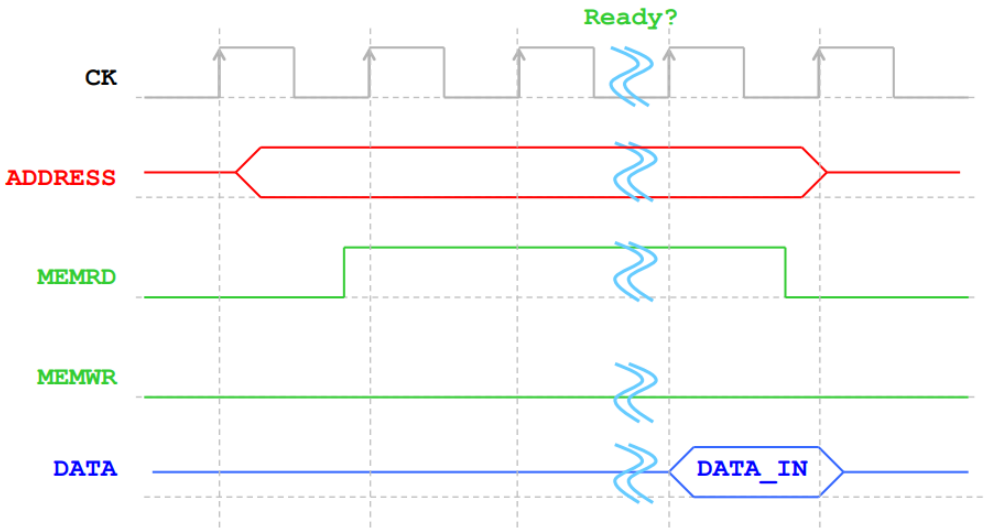
L'unica costante tra questi tre tipi di istruzioni è che l'OpCode sia posto nei 6 bit più significativi, questo è stato fatto per semplificare la progettazione dell'Unità di Controllo, che così conosce già a priori la posizione dell'OpCode [2].

### 1.1.2 Modalità di accesso alla memoria

Nel DLX l'accesso alla memoria avviene in maniera indiretta, ovvero l'indirizzo a cui accederà il DLX viene calcolato a tempo di esecuzione, facendo la somma tra il valore contenuto nel registro e una costante detta immediato. In questo modo è possibile cambiare il valore del registro a tempo di esecuzione e ottenere quindi un indirizzo diverso [2].

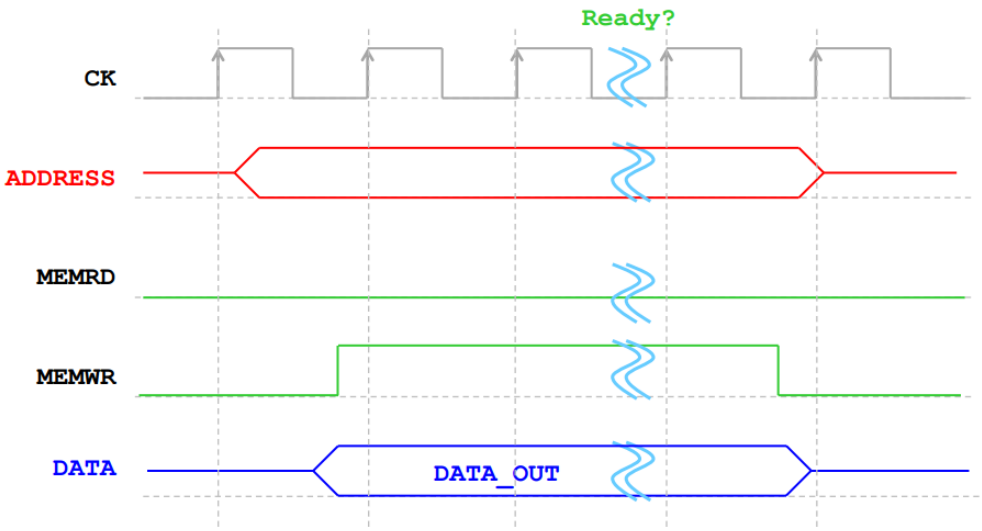
### 1.1.3 Cicli di bus

Il DLX accede alla memoria in maniera sincrona, tramite i cicli di bus. Di seguito un ciclo di bus di lettura: il segnale CK indica il clock della CPU, ADDRESS il bus degli indirizzi, usato per attivare per attivare i chip di memoria, MEMRD il segnale prodotto dal DLX per indicare ai chip di memoria che vuole effettuare una lettura, MEMWR generato dal DLX per indicare ai chip di memoria che vuole effettuare una scrittura, DATA il bus su cui viaggiano i dati, e Ready è una costante che mi dice se il dispositivo di memoria è pronto per l'operazione di lettura o scrittura. Il numero di cicli di clock necessari per fare un'operazione di I/O dipende dalla velocità della periferica.




Ciclo di bus di lettura [3]

Quando si vuole fare un'operazione di lettura, il DLX genera i segnali sul bus degli indirizzi e li mantiene asseriti fino alla fine dell'operazione, perché i chip di memoria devono essere abilitati sin da subito. Dopo ADDRESS viene abilitato il segnale MEMRD che serve ad abilitare i three state che collegano i chip al bus dati e rimane asserito fino al termine dell'operazione. Dopo che le periferiche hanno inviato il segnale di READY, mettono sul bus dati (DATA) i dati salvati all'indirizzo indicato.




Ciclo di bus di scrittura [3]

Nel caso di un'operazione di scrittura, viene messo sul bus degli indirizzi l'indirizzo della periferica interessata, subito dopo viene asserito il segnale MEMWR e vengono messi sul bus dati i dati, in modo che appena pronta la periferica possa leggerli. Il ciclo di clock successivo alla ricezione del segnale READY, non vengono più asseriti i segnali DATA, MEMWR e ADDRESS [3].

# 2 Strumenti utilizzati

## 2.1 Angular

I progettisti originali del simulatore scelsero di realizzare una Web Application utilizzando il framework Angular di Google.

Angular nasce nel 2009 con il nome di AngularJS, come estensione di JavaScript per lo sviluppo di Single Page Application, sviluppato principalmente da Google, ma comunque open-source.
A partire dalla release di settembre 2016 prende il nome di Angular 2+ o Angular CLI, ma per brevità lo si chiama semplicemente Angular. Questa versione e le successive non sono compatibili con le precedenti, poiché è stato totalmente riscritto; infatti, questa versione si basa sul linguaggio di programmazione TypeScript [4].

TypeScript è un linguaggio di programmazione open-source sviluppato da Microsoft, con lo scopo di avere un linguaggio di scripting per front-end che fosse più sicuro e robusto di JavaScript. TypeScript fa lo *static checking*, ovvero controlla che il codice sia corretto prima dell'esecuzione; è più tipato rispetto a JavaScript, conseguentemente certe operazioni lanciano errore per incompatibilità tra i tipi. La sintassi di TypeScript è fatta in modo che qualsiasi codice JavaScript funzionante sia corretto anche in TypeScript, senza preoccuparsi di come è scritto [5].

Angular segue la filosofia dei componenti, ovvero è possibile definire "classi" costituite da un file HTML, un file TypeScript contenente gli script, un file CSS contente gli stili. Una volta che un componente è stato definito, è possibile riutilizzarlo nel progetto.

Una componente fondamentale di Angular è Angular CLI, che consente di fare diverse operazioni, inserendo dei comandi da terminale. Per esempio, con il comando `ng new <nome-progetto>` viene creato un nuovo workspace Angular nella cartella corrente. Invece, con `ng serve` il progetto viene compilato e messo in esecuzione in locale, è possibile accedervi all'indirizzo `http://localhost:4200`. Con il comando `ng build <nome-progetto>` viene compilato il progetto e viene salvato nella cartella /dist. Per ottimizzare l'eseguibile è possibile usare l'opzione `--configuration production`, che sostituisce il deprecato `--prod`. Questa opzione consente di fare diverse ottimizzazioni all'applicazione web, per esempio introducendo la compilazione AOT (Ahead Of Time), rimuovendo il compilatore JIT (Just In Time), in modo da necessitare di meno risorse hardware per l'esecuzione [6].

## 2.2 Version Control System

Durante lo sviluppo è stato utilizzato un Version Control System (VCS), che consente di tenere traccia di tutte le modifiche apportate al codice, di mantenere più versioni di uno stesso codice e di lavorare in gruppo sul medesimo codice. Il codice sorgente e le sue versioni precedenti vengono salvate nella cosiddetta repository. Una funzionalità dei VCS è il branching, ovvero copiare il codice da una repository in un'altra, le modifiche apportate al branch non si propagano all'originale. L'operazione opposta è il merging, che consiste nell'importare le modifiche del branch nel ramo principale dello sviluppo. Una volta che il codice è stato modificato e si intende salvare tali modifiche si esegue il commit, cioè si invia codice modificato verso la repository, nel caso in cui sul codice lavorino più sviluppatori c'è il rischio di conflitti [7].

In particolare, è stato utilizzato Git, il VCS ideato da Linus Torvalds, per tenere traccia delle modifiche apportate al kernel di Linux. Git non prevede una repository centralizzata, ma ogni sviluppatore ha la propria in locale e si sincronizzano tramite patch [7].

Per una gestione più sicura del codice è stato utilizzato anche GitHub, una piattaforma di storage online per la conservazione del codice. Il codice sorgente presente su GitHub è disponibile a chiunque, è possibile fare una fork e copiarlo nella propria repository online, per poi poterlo modificare [8].

# 3 Correzione di bug

Prima di correggere i bug è stato necessario un processo di collaudo, per valutare cosa non funzionasse per poi risolverlo.

Il collaudo è cominciato provando tutte le istruzioni del DLX, inserendo anche casi limite. Il risultato è che le istruzioni funzionano correttamente, inoltre, analizzando il codice è parso tutto coerente.

Il passaggio dal formato esadecimale a decimale non funzionava correttamente. L'errore consisteva in un errato valore decimale, mentre nel formato esadecimale era corretto. Questo era dovuto alla funzione di conversione, che faceva uno shift logico a destra di 4bit, che poi è stato sistemato facendo uno shift logico a destra di 0 bit. Questo shift logico fa sì che l'interprete JavaScript tratti il numero come unsigned, così il valore decimale risulta coerente con quello esadecimale.

Diversi bug riguardano il contatore inserito da Filippo Comastri e documentato nella sua tesi [9]. Il primo bug era piuttosto evidente, poiché eseguendo uno stesso codice il risultato era diverso ogni volta, poiché non erano stati implementati dei metodi per gestire gli input A_SET e A_RESET del Flip-Flop D che gestisce il segnale U/D* del contatore. Conseguentemente, all'avvio il FFD si ritrovava in uno stato di metastabilità ed era quindi impossibile sapere a priori l'output del FFD. Inserendo queste funzioni per il reset sincrono e asincrono, il contatore dà sempre lo stesso risultato eseguendo lo stesso codice.

```
// reset asincrono del FFD
public a_reset_ffd(){
  //asserendo il segnale di A_RESET, il FFD inizializza l'output a 0
  this.up_down_value = 0;
}

// set asincrono del FFD
public a_set_ffd(){
  //asserendo il segnale di A_SET, il FFD inizializza l'output a 1
  this.up_down_value = 1;
}
```

Un altro bug riguarda il non corretto funzionamento del reset asincrono del contatore. Infatti, nonostante il contatore avesse come input di reset asincrono il segnale RESET, generato all'avvio dal DLX, il contatore non veniva resettato a ogni nuova esecuzione del codice. Per risolverlo è stato sufficiente inserire un metodo che gestisse il reset asincrono del contatore e che venisse invocato all'avvio dell'esecuzione.

Sempre nel contatore è stato riscontrato un problema relativo alla lettura del valore del contatore, tramite l'attivazione di un three state collegato al bus dati. Il problema era dovuto

al valore di uscita del contatore che veniva aggiornato solo in caso di lettura, rendendo così la lettura corretta solo nel caso in cui il contatore abbia l'input del clock collegato a MEMWR*. È stato risolto rendendo l'aggiornamento del valore esterno ogni qual volta si attivi il three state.

```
// reset asicnrono del contatore
public a_reset() {
  this.currentValue = 0 ;
    /*aggiorno cs_read_value_counter con il nuovo valore di currentValue. Se non lo facessi facendo
      successivamente una lettura a cs_read_value_counter non otterrei il valore aggiornato di
      currentValue
    */
    this.setCS("CS_READ_VALUE_COUNTER", this.min_address ,this.currentValue);
}
```

```
//operazioni di avvio del Counter
public startOp() {
  //pone a 0 il valore del counter
  if(this.a_reset_value.includes("RESET"))
    this.a_reset();
  //pone a 0 il valore di output del FFD
  if(this.a_set_value_ffd.includes("RESET"))
    this.a_set_ffd();
  //pone a 1 il valore di output del FFD
  if(this.a_res_value_ffd.includes("RESET"))
    this.a_reset_ffd();
}
```

Le operazioni di avvio sono state inserite tutte in uno stesso metodo.

Per invocare questi metodi all'avvio è stato necessario modificare il metodo *startResetSignal()* in *editor.component.ts* affinché vengano invocati all'avvio.

```
startResetSignal() {
  // WHEN THE APPLICATION START SEND THE RESET SIGNAL TO THE LOGICAL NETWORKS
  this.memoryService.memory.devices.forEach(el => {
    if(el.devType.includes("Start"))
      (el as StartLogicalNetwork).startOp();
    if(el.devType.includes("Led"))
      (el as LedLogicalNetwork).startOp();
    if(el.devType.includes("FF-D"))
      (el as FFDLogicalNetwork).startOp();
    if(el.devType.includes("Counter"))
      (el as Counter).startOp();
  });
}
```

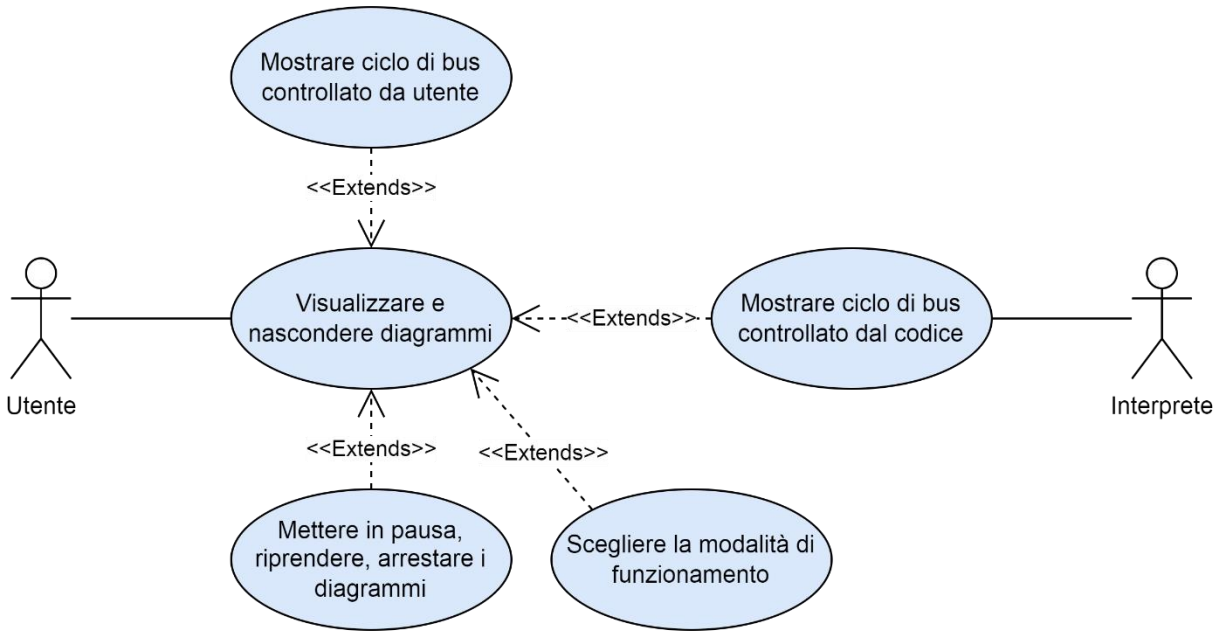
# 4 Rappresentazione in tempo reale dei cicli di bus

## 4.1 Analisi dei requisiti

### 4.1.1 Tabella dei requisiti

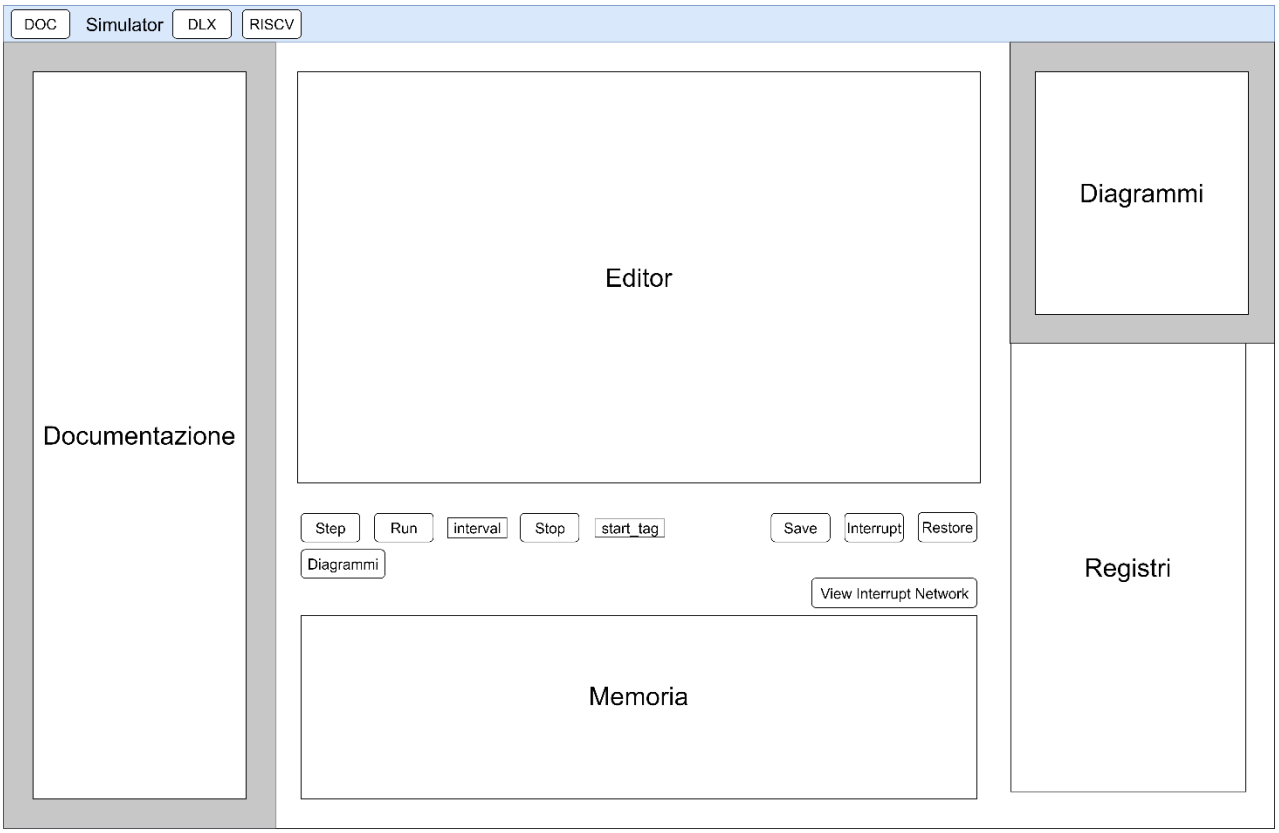
| ID | Requisito | Tipo |
|---|---|---|
| R1F | I diagrammi vengono avviati in base al codice eseguito dall'editor. | Funzionale |
| R2F | I diagrammi possono essere avviati dall'utente. | Funzionale |
| R3F | La visualizzazione dei diagrammi può essere abilitata e disabilitata dall'utente. | Funzionale |
| R4F | L'utente può mettere in pausa, riavviare e arrestare i diagrammi. | Funzionale |
| R5F | L'utente può decidere se lasciare il funzionamento dei diagrammi in mano all'editor oppure se gestirli lui. | Funzionale |
| R1NF | I diagrammi devono essere posti nella pagina in modo che sia possibile vedere bene la relazione tra un'istruzione e i segnali generati. | Non funzionale |
| R2NF | La rappresentazione dei diagrammi deve avere un overhead ridotto. | Non funzionale |

### 4.1.2 Casi d'uso

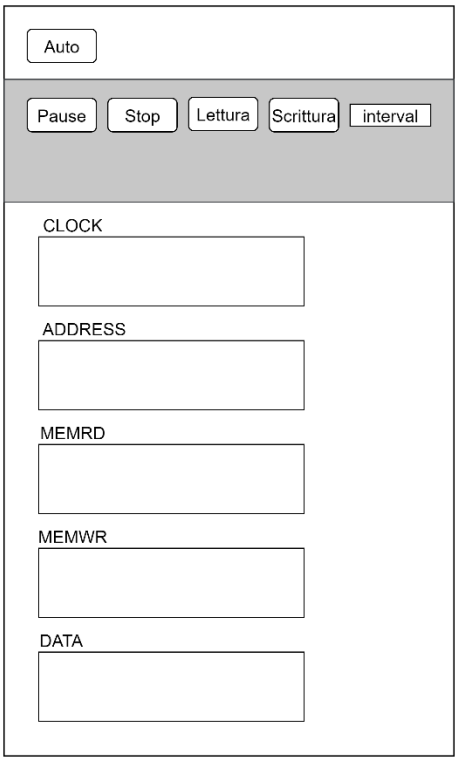


- Visualizzare e nascondere i diagrammi: dà la possibilità all'utente di scegliere se visualizzare o meno i diagrammi.
- Mettere in pausa, riprendere: arrestare i diagrammi, consiste nella gestione dell'andamento dei diagrammi.
- Mostrare ciclo di bus controllato dall'utente: l'utente può innescare la rappresentazione di un ciclo di bus, che sia di lettura o di scrittura.
- Scegliere la modalità di funzionamento: l'utente può decidere se controllare lui i diagrammi, oppure delegare al sistema, in questo caso identificato come l'Interpreter.
- Mostrare ciclo di bus controllato dal codice: l'Interprete gestisce i diagrammi secondo le istruzioni che vengono eseguite.

### 4.1.3 Interfacce Grafiche




Nell'immagine sopra riportata è mostrata l'interfaccia della pagina, i diagrammi sono a scomparsa e la visualizzazione viene abilitata dal pulsante con la dicitura *Diagrammi*. Alla visualizzazione dei diagrammi, viene mostrata una barra di scorrimento verticale, che consente di scorrere la sola porzione di destra, contente i diagrammi e i registri, così da non compromettere la visibilità di questi ultimi.




In questa seconda immagine è rappresentata l'interfaccia della sezione *Diagrammi*, il bottone *Auto* abilita o disabilita la visualizzazione dei bottoni sottostanti, che sono a disposizione dell'utente. Nel caso in cui la modalità d'uso sia indicata come non Auto (e quindi *Manual*), l'utente, tramite gli appositi bottoni, può:

- Avviare l'animazione di un ciclo di bus di lettura;
- Avviare l'animazione di un ciclo di bus di scrittura;
- Mettere in pausa l'animazione e se in pausa può riprenderla;
- Inserire la durata complessiva dell'animazione;

## 4.2 Scelte progettuali

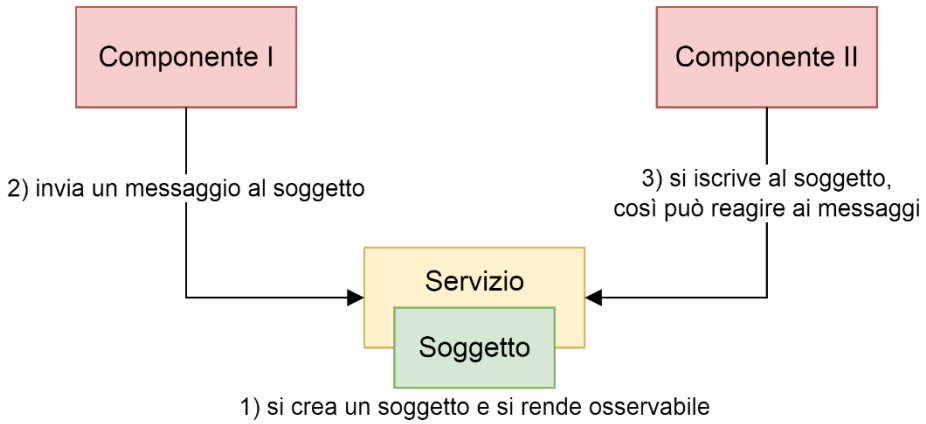
Lo scopo della tesi è quello di estendere il simulatore, di conseguenza si deve sottostare alla scelta precedentemente fatta di realizzare una applicazione web basata sul framework Angular.

La libertà in fatto di scelte progettuali rimane circoscritta ai soli Diagrammi.
Il principio dietro le animazioni è quello di avere un'immagine statica che viene fatta traslare verso sinistra, in questo modo sembra che il segnale venga generato dinamicamente. Per mantenere la formattazione dei diagrammi mostrata nei lucidi, le immagini hanno tutte la stessa dimensione e vengono traslate alla stessa velocità.
Per gestire le animazioni dei diagrammi sono stati usati script in formato .sass.

SASS (Syntactically Awesome Style Sheets) è un preprocessore che converte script in formato CSS, migliorando le prestazioni e rendendo la scrittura degli script più veloce. In SASS è possibile definire delle variabili utilizzabili in tutto il resto del foglio di stile, inoltre consente l'annidamento, che riduce il codice da scrivere, dà la possibilità di importare codice da altri file ed infine prevede l'ereditarietà [10].

Allo scopo che i diagrammi possano essere controllati da altri componenti o classi del sistema, si è ricorso all'uso dei servizi, ossia dei componenti singleton iniettabili che fungono da tramite tra due componenti, indipendentemente dalla loro relazione.
Il servizio è un elemento che include al proprio interno un soggetto, ovvero un elemento che è contemporaneamente osservato e osservatore. Ciò si traduce nella capacità di intercettare un evento e di inoltrarlo ad altri elementi, che poi reagiranno di conseguenza. Per poter avere una reazione da parte dei componenti interessati, questi devono essere iscritti al soggetto [11].

Di seguito una schematizzazione del funzionamento del servizio:

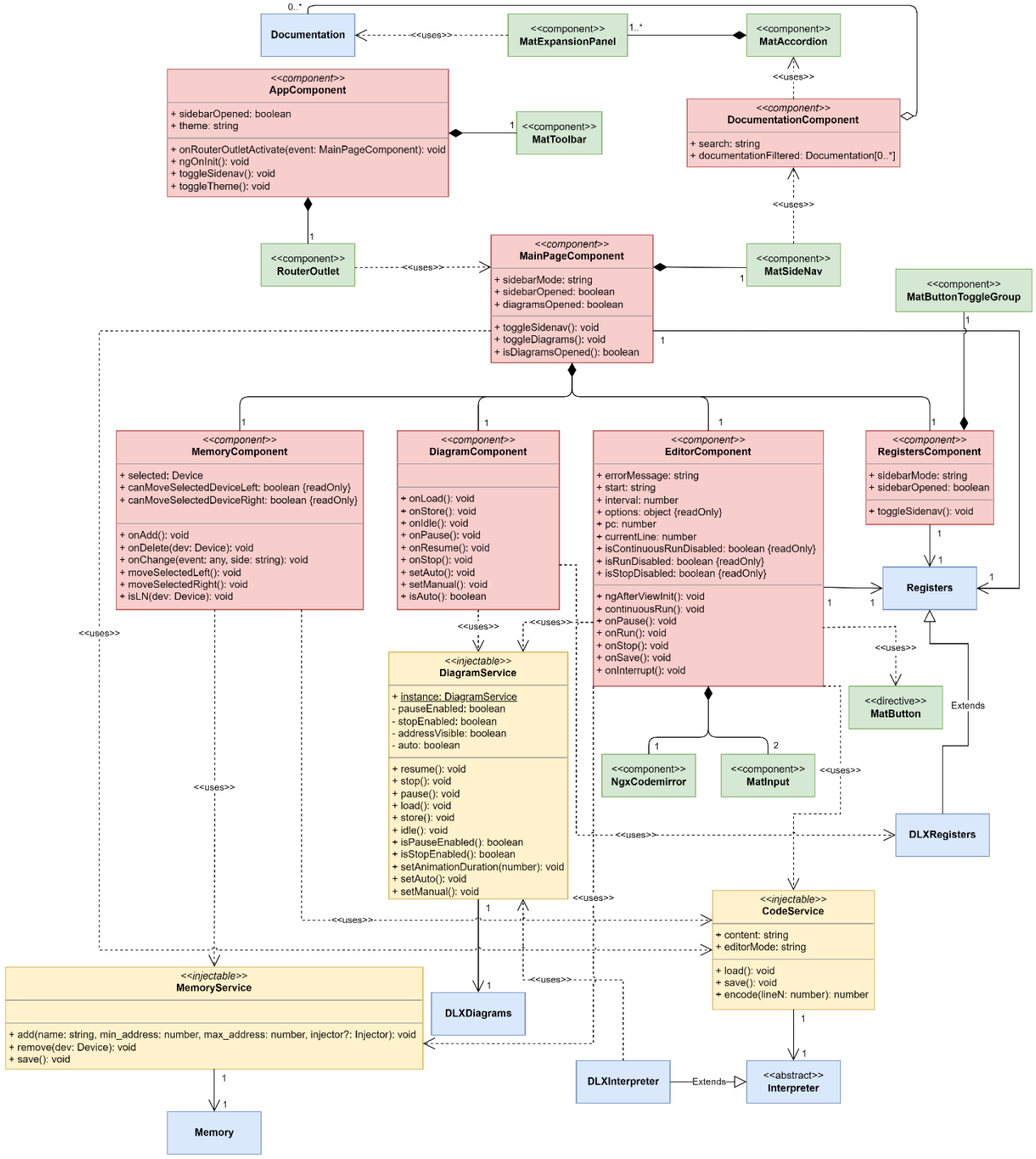



Nel sistema si è quindi realizzato un servizio denominato *DiagramService*, che contiene metodi pubblici invocabili da chiunque per interagire con i diagrammi. In particolar modo il servizio viene utilizzato da:

- *EditorComponent*: per gestire lo start, pause e stop dei diagrammi;
- *DLXInterpreter*: per avviare i cicli di bus di lettura e scrittura;
- *DiagramComponent*: per la gestione dei diagrammi da parte dell'utente.

## 4.2.1 Diagramma delle classi

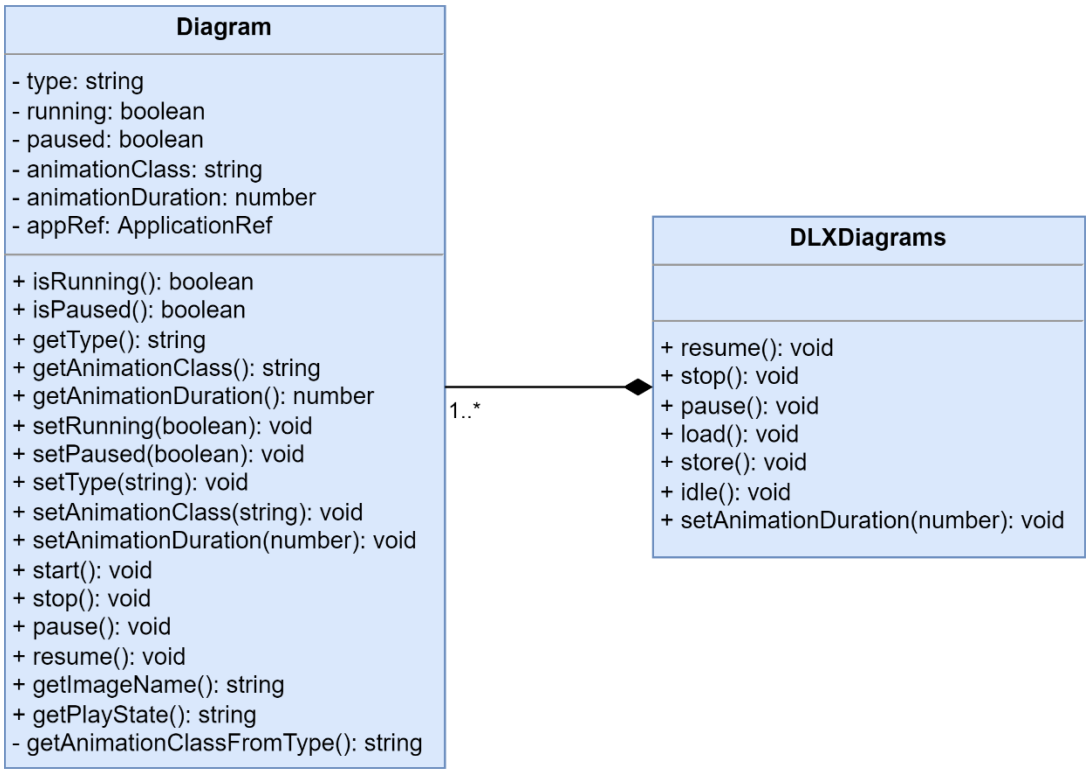
Di seguito il diagramma delle classi, esteso con gli elementi necessari per i Diagrammi:




Si noti che *DiagramService* presenta tra gli attributi un'istanza statica di sé stesso, questo perché altrimenti sarebbe impossibile far utilizzare a *DLXInterpreter* il servizio e perciò non si potrebbero far muovere i diagrammi secondo le istruzioni eseguite.

### 4.2.2 Modello

Partendo dai principi di design studiati nel corso di Ingegneria del Software, si è pensato di realizzare una classe che gestisca un singolo diagramma e poi, tramite composizione, avere tutti i diagrammi in una seconda classe. Lo scopo della classe *DLXDiagrams* è quello di gestire i pattern di animazione per i cicli di bus. Inoltre, l'architettura è stata realizzata in modo che sia relativamente semplice aggiungere un nuovo diagramma e gestire nuovi pattern.



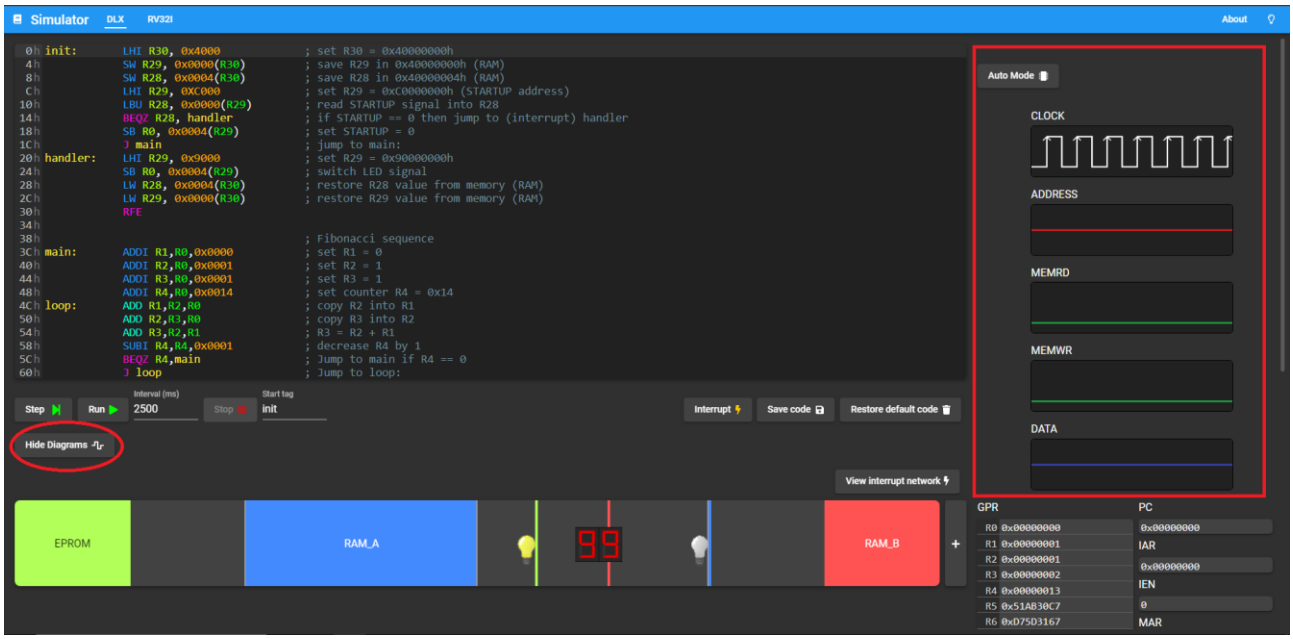

Tra gli attributi della classe *Diagram* figura *ApplicationRef*, che è il riferimento alla pagina in cui è in esecuzione l'applicazione. Il suo utilizzo è necessario per il metodo *stop()*, l'idea dietro questo metodo è quella di impostare la classe dell'animazione a *'none'*, innescare il refresh con il metodo *tick()* di *ApplicationRef* e poi reimpostare la classe dell'animazione a quella corretta.

# 5 Sperimentazione

Per mostrare il funzionamento del sistema è stato utilizzato il codice presente di default nel simulatore, ovvero lo sviluppo della successione di Fibonacci.

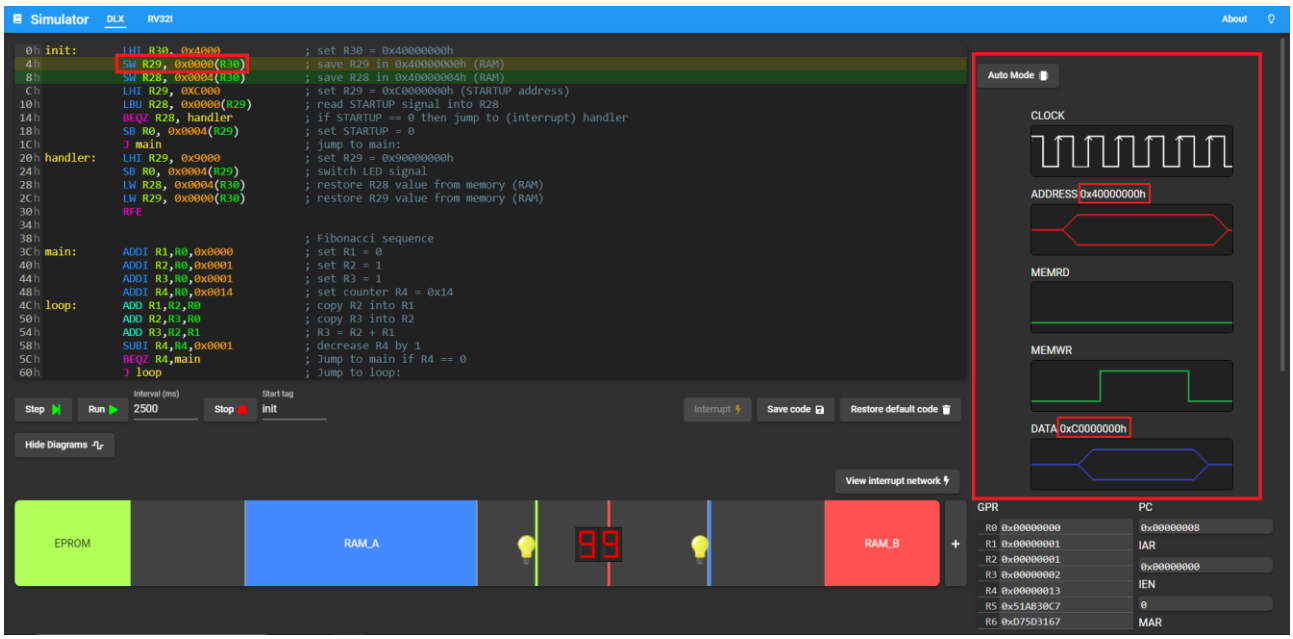
Di seguito sono riportati alcune schermate che mostrano il funzionamento del componente integrato nel sistema.

## 5.1 Abilita/disabilita diagrammi



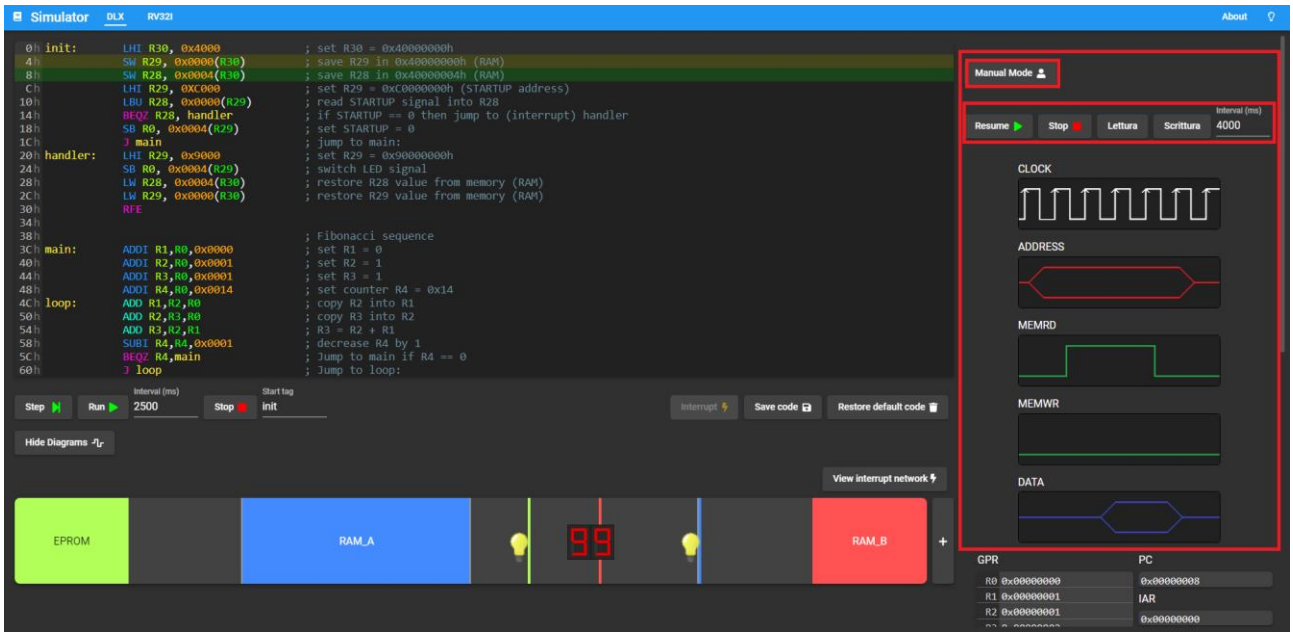

Dopo aver interagito con il bottone con dicitura *Show Diagrams*, viene aperto sulla destra una visualizzazione dei diagrammi, di default, la modalità di utilizzo è *Auto* e quindi i diagrammi si muovono seguendo l'esecuzione delle istruzioni. In questa modalità la durata dell'animazione è data dal valore di *interval*.

## 5.2 Funzionamento automatico




All'esecuzione di un'operazione di store, viene avviato un ciclo di bus di scrittura, e dato che siamo in modalità *Auto*, la pausa, la ripresa e l'arresto dell'animazione vengono controllati dagli stessi bottoni usati per gestire l'esecuzione. Si noti come vengano riportati vicino ad *ADDRESS* e a *DATA* i valori rispettivamente del *MAR* e del *MDR*.

## 5.3 Funzionamento manuale




In modalità *Manual* il funzionamento dei diagrammi è delegato all'utente che può simulare un ciclo di bus di lettura o di scrittura, può mettere in pausa, riprendere o arrestare

l’animazione. Infine, la durata dell’animazione è controllata tramite la casella di input con dicitura *Interval*.

## 5.4 Prestazioni

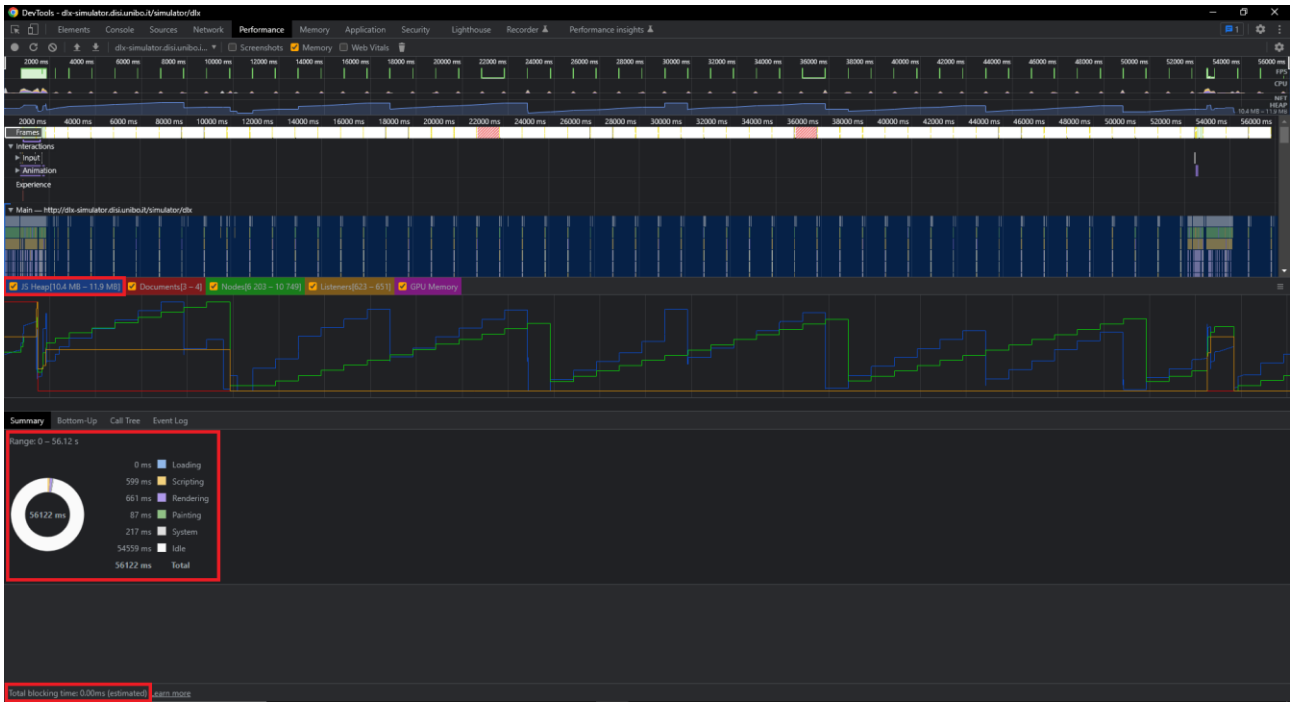
Tra i requisiti non funzionali è previsto un utilizzo ridotto delle risorse, per avere una maggiore reattività ed evitare attese frustranti.

Per avere un metro di paragone oggettivo tra la vecchia e la nuova versione è stato allestito un apposito benchmark.
Il benchmark prevede un loop di operazioni di lettura e scrittura sulla memoria utilizzando un intervallo di 1000 ms, questo perché si vuole costringere il server web a fare il render delle animazioni con una frequenza elevata. Il benchmark viene fatto girare per il tempo necessario per eseguire tutte le istruzioni 6 volte.

```
 0h init:      LHI R30, 0xFFFF
 4h store:     SW R29, 0x0000(R30)
 8h            SB R3, 0x0008(R30)
 Ch            SH R8, 0x000C(R30)
10h load_u:    LBU R18, 0x0000(R30)
14h            LHU R19, 0x0008(R30)
18h load:      LB  R20, 0x0000(R30)
1Ch            LH  R21, 0x000C(R30)
20h            LW  R22, 0x0004(R30)
24h            J store
```

Codice del benchmark

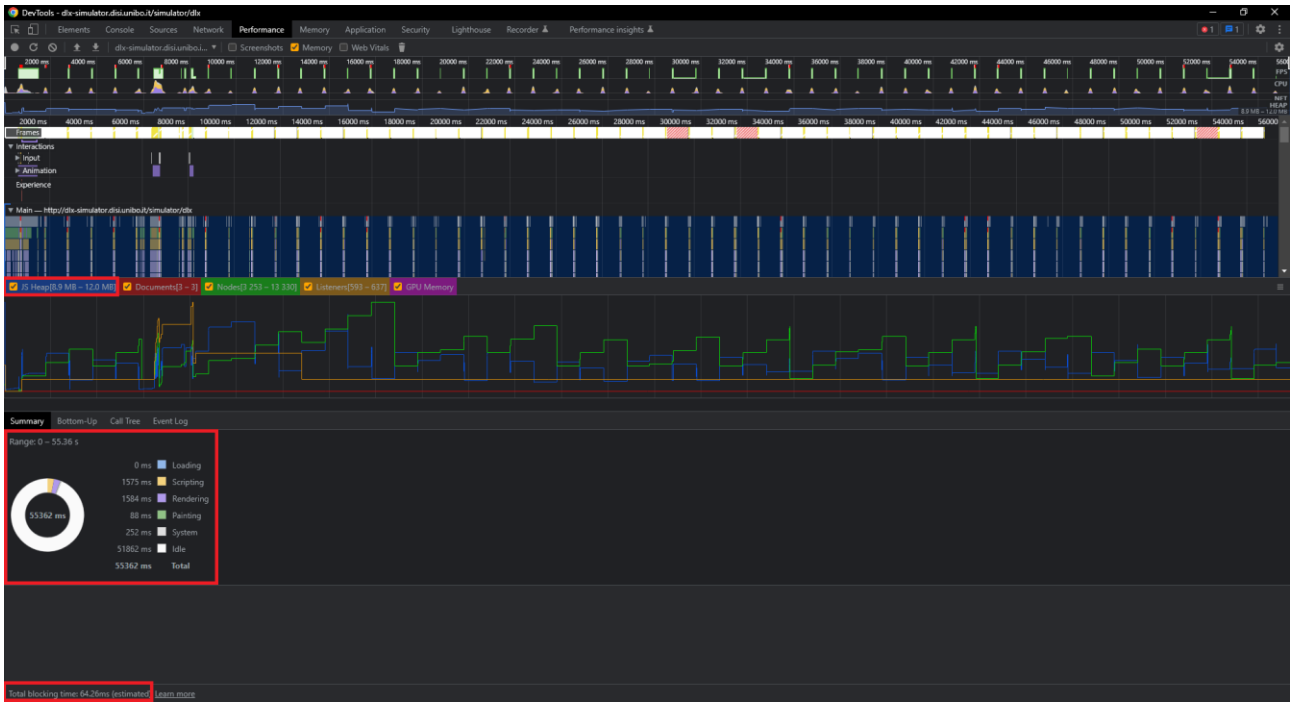
### 5.4.1 Vecchia versione




Si noti come l'heap utilizzato dall'interprete JavaScript utilizzi tra i 10.4 e 11.9 MB di memoria. Inoltre, la maggior parte del tempo è in idle, con una ridotta percentuale di tempo utilizzato nello scripting e nel rendering, rispettivamente lo 1,067% e 1,178% della durata totale del benchmark.

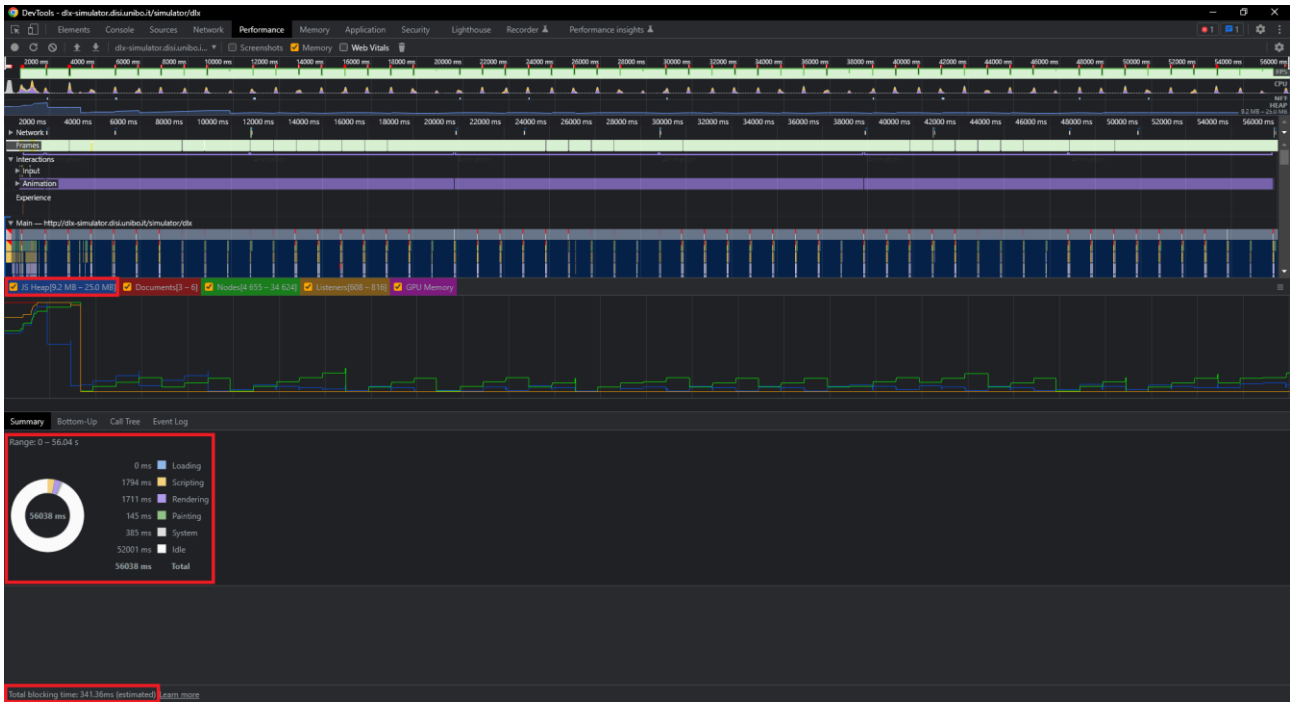
L'ultimo parametro degno di attenzione è il Total Blocking Time (TBT), ovvero il tempo in cui il main thread rimane bloccato senza poter gestire l'interazione con l'utente. Se questo valore dovesse superare i 50 ms, si potrebbe percepire lentezza e scarsa responsività [12]. In questo il TBT stimato è di 0 ms, un valore ottimo.

### 5.4.2 Nuova versione senza diagrammi a vista




Senza avere i diagrammi a vista, con la nuova versione l'interprete JavaScript mantiene quasi invariato l'heap utilizzato, però sia il tempo impiegato nel rendering sia nello scripting sono aumentati rispettivamente al 2,845% e al 2,861%. Il TBT è arrivato a 64,26 ms, ma rimane comunque tollerabile.

### 5.4.3 Nuova versione con diagrammi a vista



Con i diagrammi a vista, la memoria di heap utilizzato dall'interprete JavaScript arriva a 25 MB, il tempo impiegato in scripting e rendering rimane invariato rispetto al caso precedente. Il TBT arriva a 341,36 ms e ciò, in casi limite, potrebbe causare un'esperienza dell'utente un po' frustrante.

### 5.4.4 Conclusioni

Come si è potuto osservare, la nuova versione del sistema richiede più risorse rispetto alla precedente. Il testing eseguito sulla mia piattaforma non ha evidenziato scarsa responsività, però per un testing più approfondito e realistico bisognerà provare il simulatore sul campo.

# 6 Conclusioni

In un'ottica di sviluppi futuri per il simulatore si potrebbe ottimizzare, se possibile, il componente dei diagrammi, in modo da avere un overhead ridotto sul server e ridurre la possibilità di avere una scarsa responsività nelle interazioni.
Tra le possibili estensioni di questo progetto si potrebbero aggiungere i diagrammi per tutti i segnali di I/O del DLX, per avere una visione più completa del sistema: la base è già stata realizzata e aggiungere nuovi diagrammi non è troppo complesso.
Un ulteriore spunto può essere quello di realizzare un componente per la creazione di reti logiche costituite da FFD o Contatori tramite un'interfaccia grafica, per poi implementarle nel sistema e vedere come interagiscono con il DLX.

L'obiettivo della tesi è stato quello di correggere diversi bug e inserire la funzionalità per visualizzare in tempo reale i cicli di bus, senza modificare l'architettura del sistema e mantenendo uno stile grafico coerente.

Numerose informazioni necessarie per lo sviluppo dell'elaborato sono state reperite dalle slide del corso Calcolatori Elettronici T tenuto dal professore Stefano Mattoccia, ma è stata fondamentale la documentazione lasciata dai colleghi, in particolare Filippo Comastri che si è mostrato estremamente disponibile.

I requisiti posti sono stati soddisfatti, inoltre il progetto è stato mantenuto scalabile e il nuovo componente è stato progettato per l'estensione.

# Bibliografia


[1] Philipt M.Sailer, David R.Kaeli: *The DLX Instruction Set Architecture Handbook*

[2] Stefano Mattoccia - Slide corso Calcolatori Elettronici T Ingegneria Informatica: *Linguaggio macchina*

[3] Stefano Mattoccia - Slide corso Calcolatori Elettronici T Ingegneria Informatica: *Introduzione*

[4] Dave Gavigan - Sito web *Medium*: *The History of Angular* https://medium.com/the-startup-lab-blog/the-history-of-angular-3e36f7e828c7

[5] Documentazione ufficiale di Typescript: *TypeScript for the New Programmer* https://www.typescriptlang.org/docs/handbook/typescript-from-scratch.html

[6] Sito web *Learn Tutorial*: *Angular-cli* https://learntutorials.net/it/angular2/topic/8956/angular-cli

[7] Marco Patella - Slide corso Ingegneria del Software T Ingegneria Informatica: *Version Control Systems*

[8] Sito web *Kinsta*: *Git contro Github: Qual è la Differenza e Come Iniziare con Entrambi* https://kinsta.com/it/knowledgebase/git-contro-github/

[9] Filippo Comastri - Tesi di laurea triennale: *Estensione di un simulatore del processore DLX* http://dlx-simulator.disi.unibo.it/simulator/assets/pdf/Tesi_Filippo_Comastri.pdf

[10] Tammy Coron - Sito web *CREATIVE BLOQ*: *What is Sass? Your guide to this top CSS preprocessor* https://www.creativebloq.com/web-design/what-is-sass-111517618

[11] Canale Youtube *Codevolution*: *Angular Component Interaction - 14 - Using a Service* https://www.youtube.com/watch?v=oj6Tae2oSo0&list=PLC3y8-rFHvwgKhaLU8GTyF-5Bb8qT-wzV&index=16

[12] Philip Walton – sito web *web.dev*: *Total Blocking Time (TBT)* https://web.dev/tbt/