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

PROGETTO DI UN SIMULATORE DI DLX PER SCOPI DIDATTICI

CANDIDATO:

Fabrizio Maccagnani

RELATORE:

Prof. Stefano Mattoccia

Anno Accademico 2018/19 - Sessione III

Indice

Introduzione

Questa tesi consiste nella realizzazione di un simulatore di microprocessore DLX che permetta agli studenti del corso di Calcolatori Elettronici di testare il codice assembly imparato a lezione.
Il progetto è stato svolto in collaborazione con lo studente Alessandro Foglia che ha realizzato un simulatore RISC-V [1], siccome i due simulatori possono avere la stessa interfaccia grafica abbiamo collaborato nella realizzazione di questa.

Un simulatore è un software che all'esterno riproduce fedelmente il comportamento di un microprocessore, ma senza riprodurre il funzionamento interno identico a quello del processore fisico.

Il simulatore:

- deve avere un editor di codice nel quale scrivere il codice assembly
- deve mostrare in tempo reale durante l'esecuzione lo stato dei registri
- può consentire di modificare la configurazione della memoria legata al microprocessore
- deve fornire una documentazione delle istruzioni in linguaggio assembly

1 - Architettura DLX

DLX è un'architettura di microprocessori RISC (Reduced Instruction Set Computer) da John Hennessy e Dave Patterson [2], viene utilizzata a scopo didattico per la sua semplicità di realizzazione.
L'architettura DLX come tutte le architetture RISC consiste in un insieme di istruzioni ridotto, e molti registri ad utilizzo generale detti GPR (General Purpose Registers), a differenza delle architetture CISC (Complex Instruction Set Computer), che hanno molte più istruzioni complesse e meno registri GPR.

DLX ha 32 GPR raccolti nel cosiddetto register file e, 32 FPR (Floating Point Registers) e altri registri usati per l'interrupt handling e per le eccezioni delle operazioni floating point.
La lunghezza di una word in DLX è di 32 bit, i registri sono a 32 bit e la memoria è indirizzabile a 32 bit in modalità Big Endian (il byte più significativo a destra).

L'ISA (Instruction Set Architecture) DLX è composta da 92 istruzioni, nel nostro simulatore sono presenti soltanto le istruzioni che usano i GPR interi, e non quelli floating point, quindi parlerò soltanto delle 52 istruzioni che usano gli interi.

1.1 - Registri

1.1.1 - Registri GPR

L'ISA DLX definisce 32 GPR a 32 bit, R0-R31. Il registro R0 ha sempre valore 0, è possibile comunque usarlo come destinazione di un operando ma il suo valore non cambia. R0 può essere usato per caricare un immediato (valore fissato nel codice) a 16 bit in un altro registro mediante l'istruzione ADDI (somma immediato), mettendo R0 come primo operando.
Il registro R31 è usato per salvare l'indirizzo di ritorno per le istruzioni JAL e JALR, bisogna stare attenti a come si utilizza questo registro in modo da preservare l'indirizzo di ritorno.
I GPR possono essere usati come operandi o come destinazione della maggior parte delle istruzioni.
I registri GPR sono identificati nel register file da un indirizzo a 5 bit.

1.1.2 - Altri Registri

Ci sono altri due registri definiti nell'architettura DLX a interi: il program counter (*PC*) e l'interrupt address register (*IAR*).
Il program counter è un registro a 32 bit che contiene l'indirizzo dell'istruzione che sta venendo letta dalla memoria in un qualunque momento. Il *PC* viene modificato dalle istruzioni di jump, branch e trap. Se non viene modificato da un'istruzione il *PC* viene incrementato di 4 per puntare alla word da 32 bit successiva nella memoria.
L'interrupt address register è anch'esso un registro a 32 bit e serve per immagazzinare il valore del *PC* successivo ad un'istruzione TRAP o alla ricezione di un interrupt.

1.1.3 - Registri implementazione DLX sequenziale

Nell'implementazione sequenziale [3] dell'architettura DLX sono presenti altri registri che però non sono mai accessibili direttamente da codice: Memory Address Register (*MAR*), Memory Data Register (*MDR*), Instruction Register (*IR*), e tre registri utilizzati per gestire input e output dal register file dei GPR: *A*, *B*, *C*.
Il Memory Address Register viene utilizzato dalle funzioni di store e di load per contenere l'indirizzo della memoria nella quale si vogliono scrivere o dalla quale si vogliono leggere dei dati.
I dati da scrivere nella memoria sono temporaneamente memorizzati nel Memory Data Register, anche i dati letti dalla memoria prima di finire in un GPR passano dal registro *MDR*.
L'Instruction Register contiene invece l'istruzione letta all'indirizzo *PC* della memoria.I registri *A* e *B* sono gli output del register file mentre il registro *C* è l'input.

1.2 - Memorizzazione

La memoria è indirizzabile a 32 bit, quindi il massimo di memoria indirizzabile da DLX è di 4GB. L'accesso per halfword è ristretto ai soli indirizzi pari, mentre l'accesso per word è ristretto ad indirizzi divisibili per 4.
Dato che DLX è un'architettura puramente load/store, le sole istruzioni che accedono alla memoria sono le istruzioni di load e di store.

1.3 - Gestione interrupt

Gli interrupt sono eventi che interrompono la normale esecuzione del programma. Il processore DLX ha tre possibili interrupt: tramite segnale di Interrupt, quando avviene un'eccezione di overflow aritmetico e, l'istruzione TRAP.
Il segnale di interrupt è collegato ad eventi esterni al processore. L'eccezione di overflow aritmetico avviene durante una ADD, ADDI, SUB o SUBI se risulta un overflow del complemento a due.
Per questi due tipi di interrupt il *PC* è caricato nell'*IAR* e il *PC* viene messo a zero dove sarà scritta la routine di gestione dell'interrupt.
Invece l'istruzione TRAP dopo aver caricato il *PC* nell'*IAR* imposta il *PC* all'indirizzo assoluto indicato nell'istruzione. Per tornare all'esecuzione normale del programma l'istruzione Return From Exception (RFE) ricarica il contenuto dell'*IAR* nel *PC.*

1.4 - Istruzioni

Le istruzioni DLX rappresentano le principali primitive utilizzate nei programmi. Sono tutte istruzioni a 32 bit, esistono 3 formati diversi.

1.4.1 - Tipo I

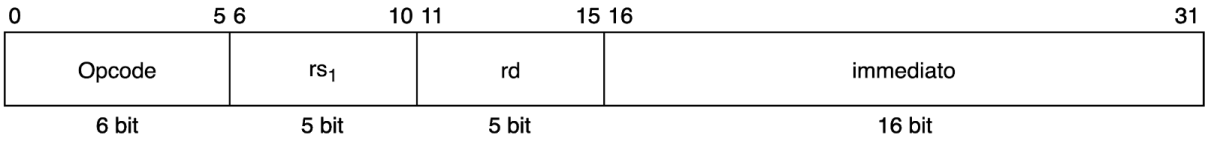

Figura 1.1 - Formato istruzione tipo I

Include le istruzioni di load e store, operazioni ALU con immediato, le istruzioni di branch condizionali, e le istruzioni di jump incondizionali Jump Register (JR) e Jump And Link Register (JALR). Il formato delle istruzioni di tipo I è illustrato nella figura 1.1.
Il campo *Opcode* specifica quale istruzione DLX deve essere eseguita, ed è comune a tutti i tipi di istruzione. Il campo rs_1 indica un registro GPR che viene utilizzato come argomento dell'istruzione. Il campo *rd* indica il registro GPR destinazione del risultato dell'operazione oppure nel caso delle istruzioni di store è il registro che contiene il valore da salvare in memoria. Il campo *immediato* contiene l'offset usato per calcolare l'MAR delle istruzioni di store e di load, o l'operando immediato per le operazioni ALU , o l'offset che viene sommato al PC per le operazioni di branch condizionale. Il campo *immediato* non viene utilizzato dalle istruzioni JR e JALR.

1.4.2 - Tipo R

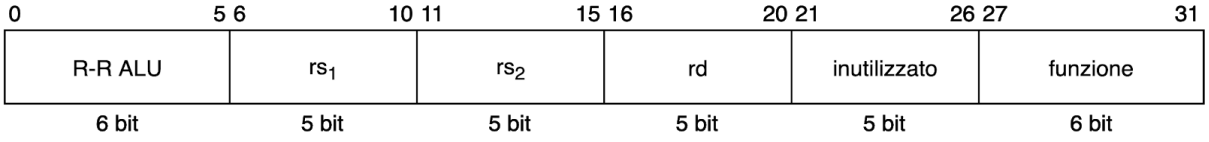

Figura 1.2 - Formato istruzione tipo R

Le istruzioni di tipo R sono usate per le operazioni ALU tra registri e per copiare il contenuto di registri GPR nei registri speciali e viceversa. Le istruzioni di tipo R del DLX a numeri interi hanno un solo Opcode cioè R-R ALU, le diverse operazioni sono distinte dal campo funzione.
I campi rs_1 indica il registro GPR che viene utilizzato come primo operando dell'istruzione.
I campi rs_2 indica il registro GPR che viene utilizzato come secondo operando dell'istruzione.
Il campo *rd* indica il registro GPR destinazione del risultato dell'operazione. I bit da 21 a 26 sono inutilizzati. Il campo funzione serve a distinguere quale operazione deve eseguire la ALU.

1.4.3 - Tipo J

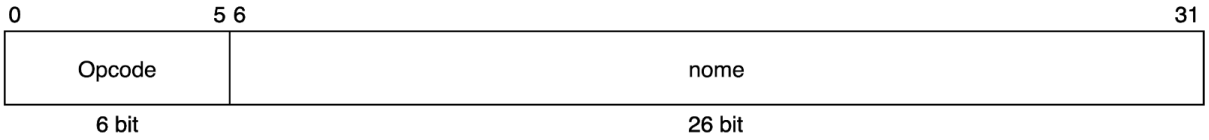

Figura 1.3 - Formato istruzione tipo J

Le istruzioni di tipo J includono Jump (J), Jump And Link (JAL), TRAP, e Return From Exception (RFE).
Il campo *Opcode* specifica quale istruzione DLX deve essere eseguita. il campo *nome* contiene l'offset che viene aggiunto all'indirizzo successivo del Program Counter (PC + 4) per ottenere il nuovo indirizzo. Per l'istruzione TRAP il campo *nome* rappresenta un indirizzo assoluto, mentre per l'istruzione RFE il *nome* non è utilizzato.

2 - Progetto - Specifiche

In questo capitolo vengono analizzate le specifiche per la realizzazione del simulatore, vengono analizzati i casi d'uso e le interfacce grafiche dell'applicazione.

2.1 - Analisi dei Requisiti

Id. Requisito	Requisito	Tipo
R1F	L'utente può scegliere l'architettura con la quale eseguire il codice assembly	Funzionale
R2F	L'utente può scrivere codice assembly	Funzionale
R3F	L'utente può scegliere da quale tag far partire l'esecuzione del codice	Funzionale
R4F	L'utente può eseguire passo passo il codice scritto	Funzionale
R5F	L'utente può far partire un'esecuzione continua del codice con delay tra un'operazione e l'altra impostabile.	Funzionale
R6F	Il programma mostra in tempo reale il valore dei registri	Funzionale
R7F	L'utente può modificare la configurazione della memoria	Funzionale
R8F	L'utente può consultare una documentazione delle istruzioni assembly	Funzionale
R1NF	Interfaccia intuitiva per la schermata con la quale interagisce l'utente	Non Funzionale
R2NF	L'interprete non deve essere troppo pesante	Non Funzionale
R3NF	Deve essere una applicazione web	Non Funzionale

2.1.1 - Casi d’uso

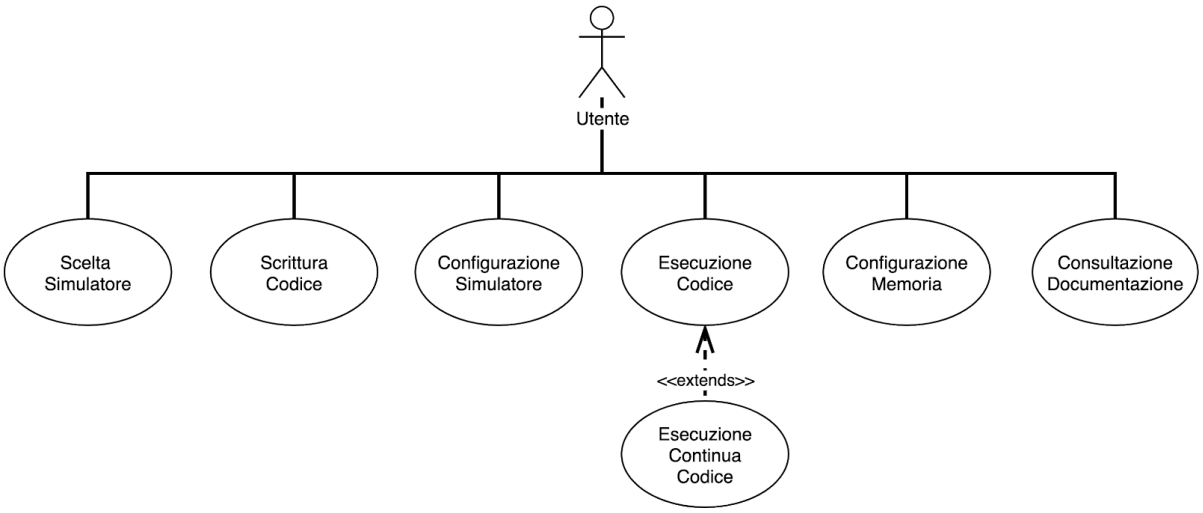

Figura 2.1 - Casi d’uso considerati

- Scelta Simulatore, tramite dei pulsanti posti nella barra superiore si può scegliere quale simulatore eseguire.
- Scrittura Codice, l’utente può scrivere il codice assembly in un editor di testo posto centralmente.
- Configurazione Simulatore, tramite degli appositi campi posti sotto l’editor di testo è possibile impostare l’intervallo di tempo usato dall’esecuzione continua e, impostare da quale tag parte l’esecuzione del codice.
- Esecuzione Codice, l’utente può eseguire il codice un’istruzione alla volta e vedere i registri che cambiano valore in tempo reale. Il codice viene eseguito da un interprete specifico per il linguaggio assembly dell’architettura selezionata.
- Esecuzione Continua Codice, l’utente può avviare l’esecuzione continua del codice, ogni tot millisecondi specificati nel campo interval della configurazione viene eseguita una linea di codice.
- Configurazione Memoria, l’utente può modificare, aggiungere o eliminare i blocchi di memoria.
- Consultazione Documentazione, l’utente può aprire e chiudere la barra laterale contenente la documentazione sul linguaggio assembly specifico dell’architettura selezionata.

2.1.2 - Interfacce Grafiche

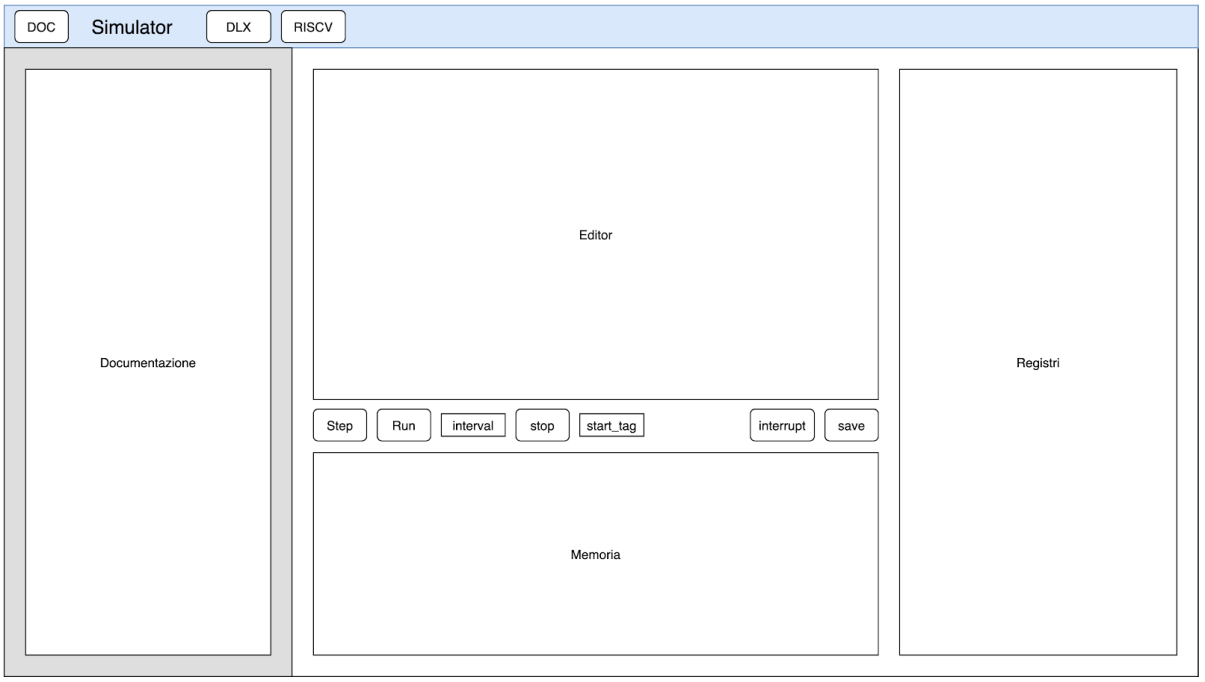

Figura 2.2 - Layout interfaccia grafica

Nella figura 2.2 viene mostrato il layout dell'applicazione, nella barra superiore troviamo i pulsanti DLX e RISCV che permettono di selezionare l'interprete che si andrà ad utilizzare.
La documentazione a sinistra si trova in una barra laterale che può essere aperta e chiusa dal pulsante doc in alto a sinistra. L'editor centrale è l'elemento grafico principale dell'applicazione, qui viene scritto il codice assembly. Il pulsante step consente l'esecuzione riga per riga del codice. Il pulsante run l'esecuzione continua. Interval è un campo numerico nel quale si seleziona l'intervallo di tempo in millisecondi, che passa tra l'esecuzione di una riga a quella dopo, nell'esecuzione continua. Il pulsante stop termina l'esecuzione. Il campo testuale start_tag contiene il nome del tag dal quale partirà l'esecuzione del programma. Il pulsante Interrupt permette di mandare al processore simulato un segnale di interrupt. Il pulsante Save serve a salvare il codice scritto. L'elemento grafico in basso è l'elemento della memoria, qui è possibile vedere e modificare la configurazione della memoria. Mentre a destra vengono mostrati i valori contenuti nei registri in tempo reale.

3 - Progetto - Sviluppo

3.1 - Scelte progettuali

Per la realizzazione del progetto è stato scelto di utilizzare un framework web, Angular, dato che è pensato per realizzare applicazioni web single page. Il linguaggio principale utilizzato in Angular è il typescript. Il typescript è un linguaggio emergente, è basato su javascript, al quale aggiunge una forte tipizzazione. Il typescript non è però un linguaggio interpretato come il javascript e per essere eseguito da un browser viene compilato in javascript. La compilazione genera del codice javascript ottimizzato e con i dovuti controlli di tipo. Angular fornisce un comodo sistema di template basato sull'html con alcune modifiche, che permettono in modo descrittivo di associare in modo bidirezionale il modello con la vista.
Sono state utilizzate anche altre librerie node costruite per Angular, ngx-codemirror e Angular Material.
Ngx-codemirror è un wrapper angular per la libreria Codemirror, Codemirror è la libreria javascript più usata per creare editor di testo nelle applicazioni web.
Angular Material è invece una libreria grafica per angular, che aggiunge gli elementi grafici di base seguendo le specifiche di Material Design.
Inoltre per le icone è stata utilizzata la libreria javascript Fontawesome.
Angular permette la creazione di diversi componenti che vengono poi mostrati in un componente principale che è l'applicazione stessa chiamato AppComponent.

3.1.2 - Struttura programma

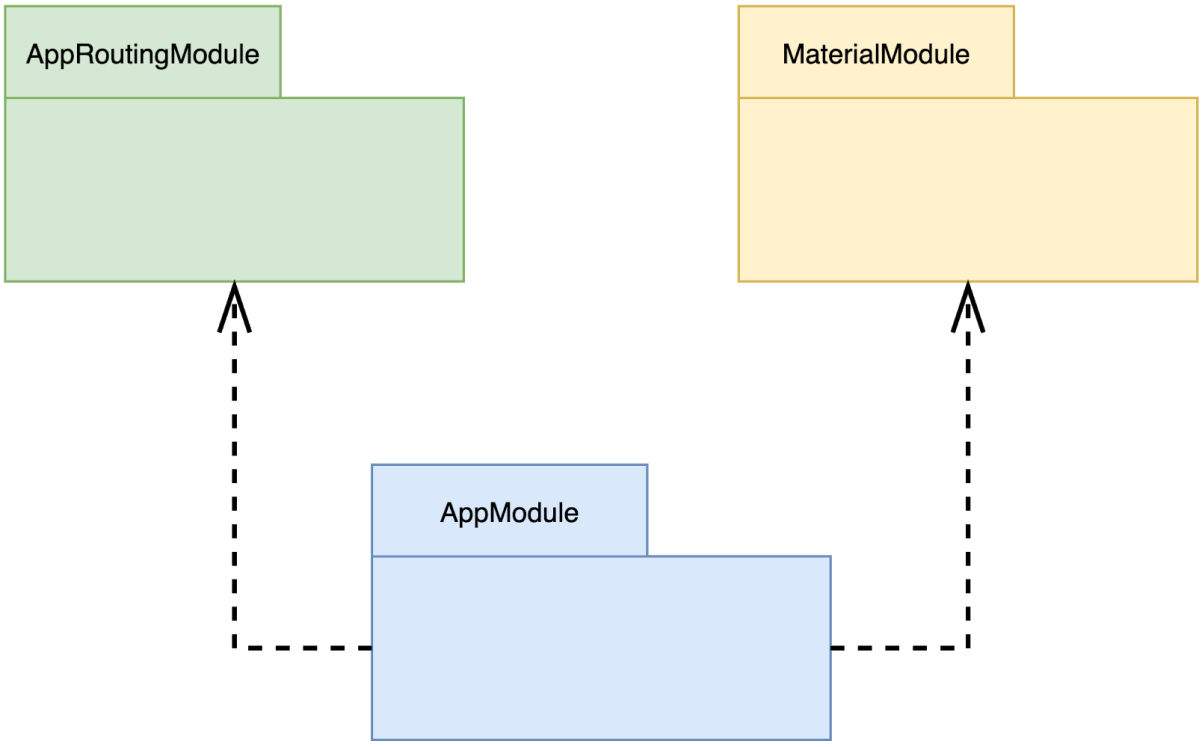

Figura 3.1 - Diagramma dei pacchetti

AppRoutingModule è un modulo in cui sono definite delle route. Una route ha 3 proprietà: path, component e data. Il path è il path relativo del server nell'url alla quale la route corrisponde. La proprietà component è la classe del componente che viene caricato nel RouterOutlet. Il RouterOutlet è un componente di Angular che presenta al suo interno un componente a seconda del path dell'url. Il RouterOutlet ci consente di cambiare l'interprete utilizzato, DLX o RISCV a seconda del path selezionato nell'url. La proprietà data invece è un oggetto generico e può essere utilizzato per passare argomenti ai componenti contenuti nel RouterOutlet, relativi alla route attiva.
Ad esempio la route /dlx è definita nel seguente modo.

```
{
    path: 'dlx',
    component: MainPageComponent,
    data: {
      interpreter: new DLXInterpreter(),
      editorMode: 'dlx',
      registers: new DLXRegisters(),
      documentation: DLXDocumentation
    }
},
```

MaterialModule è utilizzato per importare i componenti e le direttive della libreria Angular Material che vengono usate nella nostra applicazione. AppModule è il modulo principale del simulatore e contiene i componenti definiti da noi nell'applicazione.

3.1.3 - Diagramma delle classi

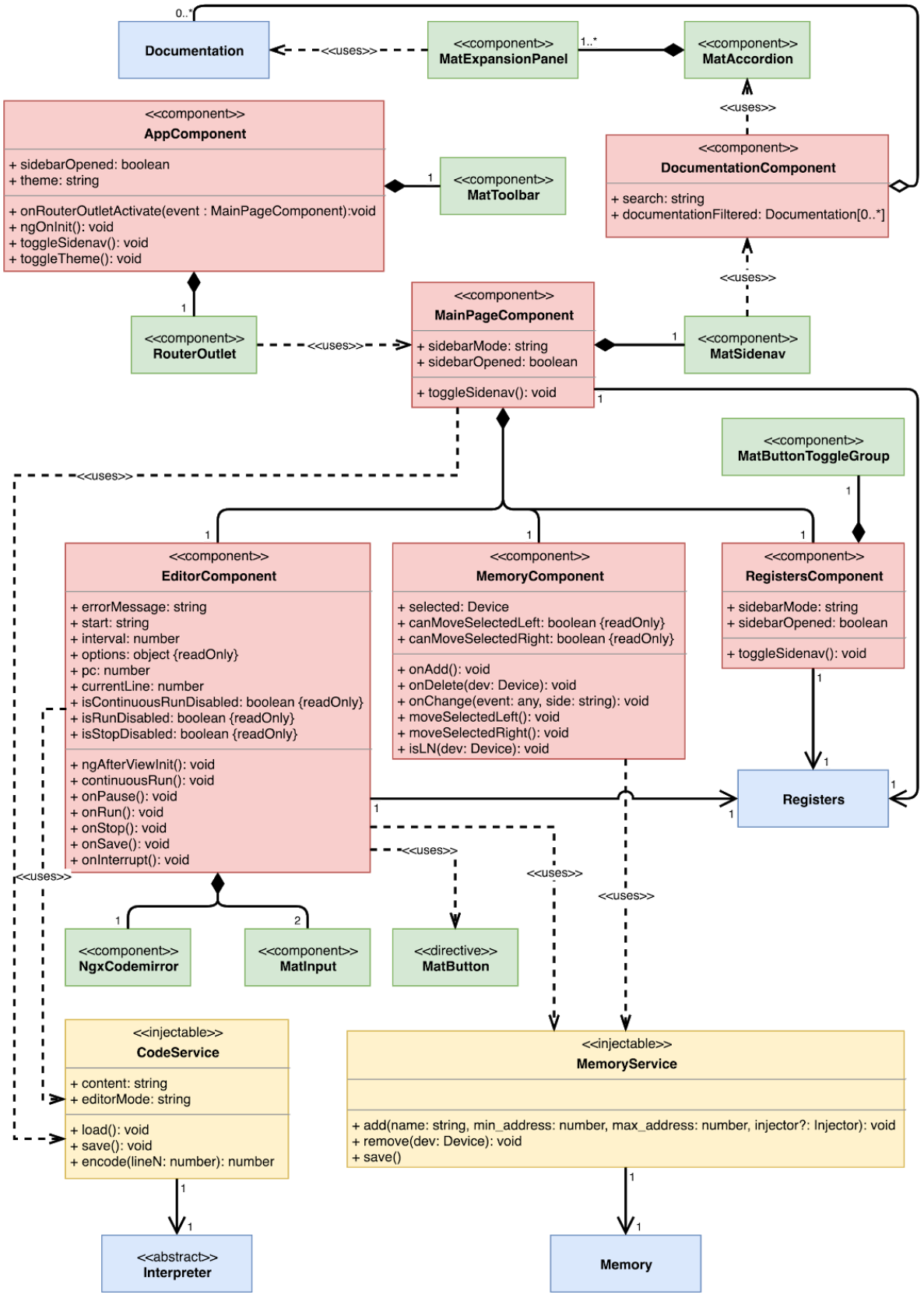

Figura 3.2 - Diagramma delle classi

In Angular un componente (<<component>>) è l'unione di una vista e di un controller, i componenti sono legati tra di loro in una struttura ad albero. Le direttive (<<directive>>) servono ad aggiungere dei comportamenti specifici ad un elemento grafico. I servizi (<<injectable>>) sono dei singleton che vengono passati ai costruttori degli altri elementi tramite pattern injection.

La figura 3.2 rappresenta la struttura dell'applicazione. In rosso ci sono i componenti dell'applicazione, in verde componenti e direttive di libreria, in blu classi del modello, in giallo i servizi dell'applicazione.

Nel componente AppComponent sono definite la barra superiore e un div principale che occupa il resto della finestra. Nel div principale è inserito il RouterOutlet. Il RouterOutlet visualizza un componente specificato dalla route, nel nostro caso mostra il componente MainPageComponent. Il componente MainPageComponent definisce la barra laterale che contiene il componente DocumentationComponent, mentre nel div centrale sono posizionati i tre componenti più importanti: EditorComponent, MemoryComponent e RegistersComponent.
DocumentationComponent ottiene dalla route una lista di oggetti Documentation specifica per l'architettura selezionata, e la mostra come una lista di campi espandibili.
EditorComponent ottiene invece dalla route un interprete e il nome del linguaggio, che viene utilizzato per il syntax highlight, esso contiene l'editor di testo CodeMirror e i pulsanti per il controllo di flusso. MemoryComponent mostra una rappresentazione grafica della memoria e dei suoi blocchi, consente di modificare la configurazione della memoria aggiungendo, togliendo, modificando i blocchi di memoria. RegistersComponent invece mostra lo stato di tutti i registri del processore selezionato, i registri GPR in una colonna e gli altri in un'altra colonna. EditorComponent utilizza CodeService per la memorizzazione del codice, e MemoryComponent usa MemoryService per salvare in locale nel client la configurazione della memoria.

3.2 - Modello

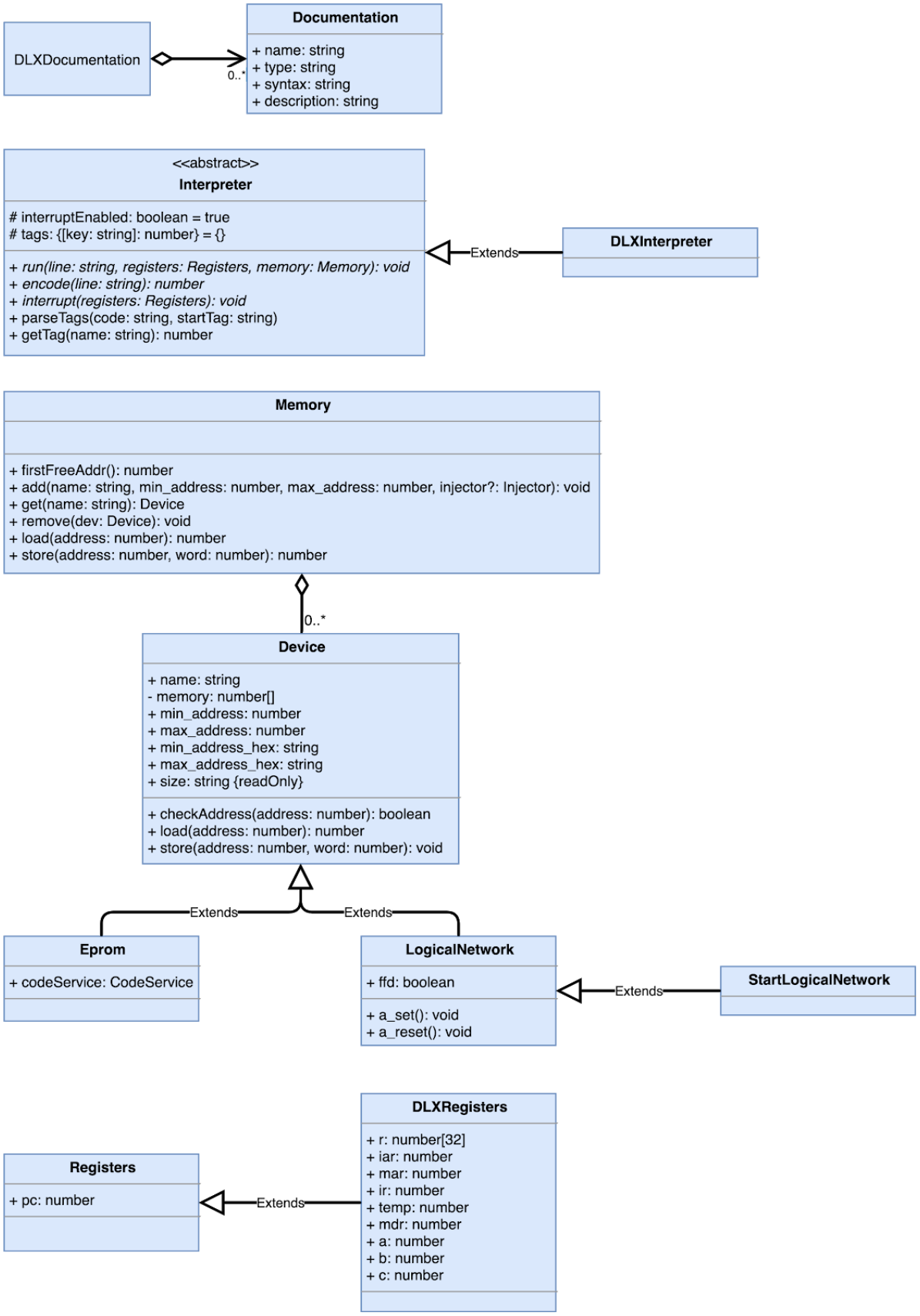

Figura 3.3 - Diagramma delle classi del modello

3.2.1 - Astrazione memoria

La memoria è rappresentata come un array di Device, ogni device ha un nome, un indirizzo minimo, un indirizzo massimo, e un array di number. La funzione checkAddress controlla se l'indirizzo effettivo diviso 4 è compreso tra l'indirizzo minimo e il massimo del device. Quando l'interprete fa una load/store la memoria cerca il device corrispondente all'indirizzo passatogli utilizzando checkAddress, poi chiama la funzione load/store del device trovato.
Load e store di Memory passano sempre delle word, l'interprete nel caso faccia una load/store di un byte o di una halfword si occupa di scomporre e ricomporre la word per la memoria.
I device hanno delle specializzazioni:

- Eprom, la eprom ridefinisce il metodo load in modo da utilizzare codeService per ritornare il codice scritto nell'editor codificato. Ridefinisce la store in modo che non faccia nulla. La Eprom non può essere rimossa dalla configurazione ne spostata dall'indirizzo 0.
- LogicalNetwork, modellizza una rete logica con un flip-flop D (ffd) [4] e implementa due funzioni: a_set() e a_reset() per rappresentare i segnali asincroni di set e reset dell'ffd.
  Per ora non è completamente implementata, potrebbe essere oggetto di futuri aggiornamenti, e serve solo come base per la classe StartLogicalNetwork.
- StartLogicalNetwork, modellizza la rete logica che viene utilizzata da DLX per capire se è appena stato avviato, dato che i processori DLX all'avvio hanno PC = 0 e iniziano quindi ad eseguire il codice da indirizzo 0. Stesso indirizzo al quale il PC viene settato per gli interrupt, quindi il codice deve essere in grado di riconoscere un avvio da un interrupt, per questo è necessaria questa rete. Per il motivo appena spiegato non è possibile rimuovere il device StartLogicalNetwork dalla configurazione.

3.2.2 - Astrazione registri

I registri sono rappresentati proprietà di tipo number, in javascript i number sono numeri a 32 bit in complemento a due, come per l'architettura DLX.
Per ottenere una rappresentazione binaria o esadecimale di una variabile javascript di tipo number esiste la funzione toString(base: number), base indica la base in quale si vuole rappresentare il numero, toString(2) restituisce quindi la rappresentazione binaria.
Ad esempio (3).toString(2) darà come risultato la stringa "11", e (13).toString(16) restituirà la stringa "d". La funzione toString di number è quindi molto comoda per mostrare il valore dei registri, l'unico problema è con i numeri negativi, non dà una rappresentazione corretta del valore dei bit. Ad esempio se scriviamo (-1).toString(16) il valore che ci aspettiamo è la stringa "FFFFFFFF" invece otteniamo "-1". Per risolvere questo problema di rappresentazione basta eseguire uno shift logico a destra di 0 bit in modo che l'interprete javascript tratti il numero come unsigned. Quindi usando (-1 >>> 0).toString(16) restituisce "FFFFFFFF".

3.3 - Interprete DLX

La classe Interpreter dichiara 3 metodi astratti run, encode e interrupt, e due metodi pubblici parseTags e getTag.
ParseTags viene chiamato dall'EditorComponent quando si preme il pulsante run o step se l'esecuzione era ferma (non in pausa). ParseTags cerca nel codice tutti i tag definiti e li inserisce nel dizionario tags dove la chiave è il nome del tag e il valore è l'indirizzo della linea di codice. I tags vengono poi utilizzati nelle funzioni di branch e di jump. GetTag serve per ottenere la posizione di un tag al di fuori dell'interprete, dato che tags è una proprietà protected della classe Interpreter.
Il metodo encode viene utilizzato dalla classe Eprom per ottenere il valore 32 bit corrispondente alla linea di codice richiesta.
Il metodo interrupt viene chiamato dall'EditorComponent quando si preme il pulsante interrupt e simula l'arrivo di un segnale di Interrupt.
Il metodo run viene chiamato dall'EditorComponent quando si preme il pulsante step oppure ogni clock del continuous run, prende in ingresso la linea di codice da eseguire, i registri su cui operare e la memoria.
La classe DLXInterpreter estende la classe Interpreter e dichiara la funzione privata processLine(line: string) che separa in token la linea di codice, il primo è l'istruzione, gli altri vengono convertiti in numeri. Se il token inizia per 'R' allora è un registro e viene quindi presa in considerazione solo il numero che poi verrà utilizzato come indice dell'array r della classe DLXRegistri. Se il token è un registro speciale allora viene associato all'indice di quel registro in un array che verrà poi utilizzato per risolvere il nome del registro speciale. Se il token inizia per '0x' allora è un immediato e viene convertito da stringa a intero con parseInt. Se il token corrisponde ad un tag contenuto nel dizionario tags allora viene convertito con l'indirizzo corrispondente al tag. Se il token non viene riconosciuto allora lancia un errore di formattazione. Alla fine processLine ritorna l'istruzione, gli argomenti modificati in numeri e la linea di codice senza tag e senza commento.
DLXInterpreter implementa le tre funzioni astratte di Interpreter:

- Encode. Invoca processLine per ottenere l'istruzione e gli argomenti sotto forma di array di numeri, poi ottiene il tipo di istruzione dal dizionario instructions dichiarato nel file dlx.instructions.ts, poi ottiene opCode e la func nel caso siano istruzioni di tipo R dal dizionario encoder dichiarato nel file dlx.encoder.ts. Infine concatena opcode, argomenti usando una funzione ottenuta dal dizionario inputs_encoder che a seconda del tipo di istruzione concatena i giusti argomenti, e la func nel caso sia definita. Così facendo si ottiene una stringa che è la rappresentazione in binario dell'istruzione, questa stringa viene parsata con parseInt() e ritornata dalla funzione.
- Interrupt. Se la proprietà interruptEnabled è true allora mette il contenuto del registro PC nel registro IAR, imposta PC a 0 e infine mette interruptEnabled a false. InterruptEnabled tornerà true solo dopo una RFE.
- Run. Chiama processLine per ottenere il nome dell'istruzione da eseguire, gli argomenti e la linea di codice senza tag e commento. Se l'istruzione esiste nel dizionario instructions allora come in encode codifica l'istruzione e la mette nel registro Instruction Register (IR). Poi sceglie dal dizionario process_instruction una funzione in baso al tipo di sintassi, e quindi di ordine in cui sono scritti gli argomenti, poi chiama questa funzione passandogli:

- la riga di codice senza tag e commenti per fare un check sintattico
- il nome dell'istruzione
- gli argomenti dell'istruzione
- la funzione associata all'istruzione
- i registri
- la memoria
- e se l'istruzione è unsigned o no, nelle tipo R se è unsigned allora non eseguirà il check sull'overflow aritmetico, nelle tipo I se è unsigned estenderà con zeri l'immediato da 16 a 32 bit, invece se non è unsigned estenderà con segno.

Le funzione contenute in process_instruction eseguono operazioni sui registri comuni a le istruzioni di quel tipo. Poi eseguono la funzione associata all'istruzione che prendono in ingresso passandogli solo i registri. Quest'ultima funzione è quella che si occupa di eseguire le operazioni di base.

4 - Sperimentazione

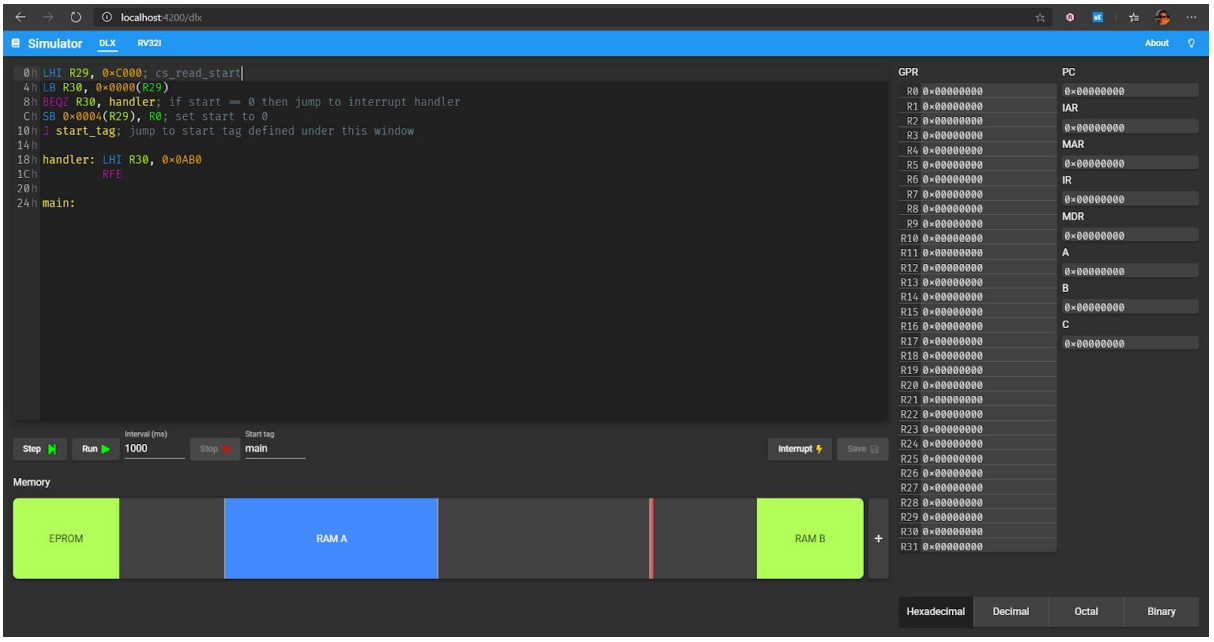

Figura 4.1 - Schermata iniziale dell'applicazione

Al primo avvio l'applicazione si mostra come nella figura 4.1, la memoria è già configurata con una configurazione di base. Il blocco EPROM non può essere né rimosso né spostato dall'indirizzo 0, ma si possono cambiare le sue dimensioni. La linea rossa rappresenta invece la rete logica Start, il segnale di set asincrono (a_set) del Flip-Flop D della rete Start è collegato all'avvio dell'esecuzione, che sia a step o continua. Il codice già scritto nell'editor legge dalla rete logica Start il valore del Flip-Flop D, se è zero allora il program counter è stato portato a zero da un interrupt, da un overflow o da un TRAP, e quindi si passa all'esecuzione dell'handler. Se invece il valore del Flip-Flop D di Start è uno allora il processore è appena stato avviato, quindi imposta il Flip-Flop D a zero e fa un salto incondizionato al tag predefinito start_tag. All'interno del codice è sempre possibile utilizzare start_tag, esso è collegato al tag scritto nel campo di testo "Start tag" sotto all'editor.
I registri sono tutti inizializzati a zero e il formato di visualizzazione di default è esadecimale.
Il pulsante save è disabilitato finché non viene apportata qualche modifica al codice o ai campi Interval e Start tag. Quando viene premuto il pulsante save il codice e il contenuto dei campi Interval e Start tag vengono salvati nel localstorage [5] del browser. Al refresh della pagina codice e configurazione del programma vengono caricati dal localstorage, se non sono presenti vengono caricati i valori di default, che sono quelli che si vedono nella figura 4.1.

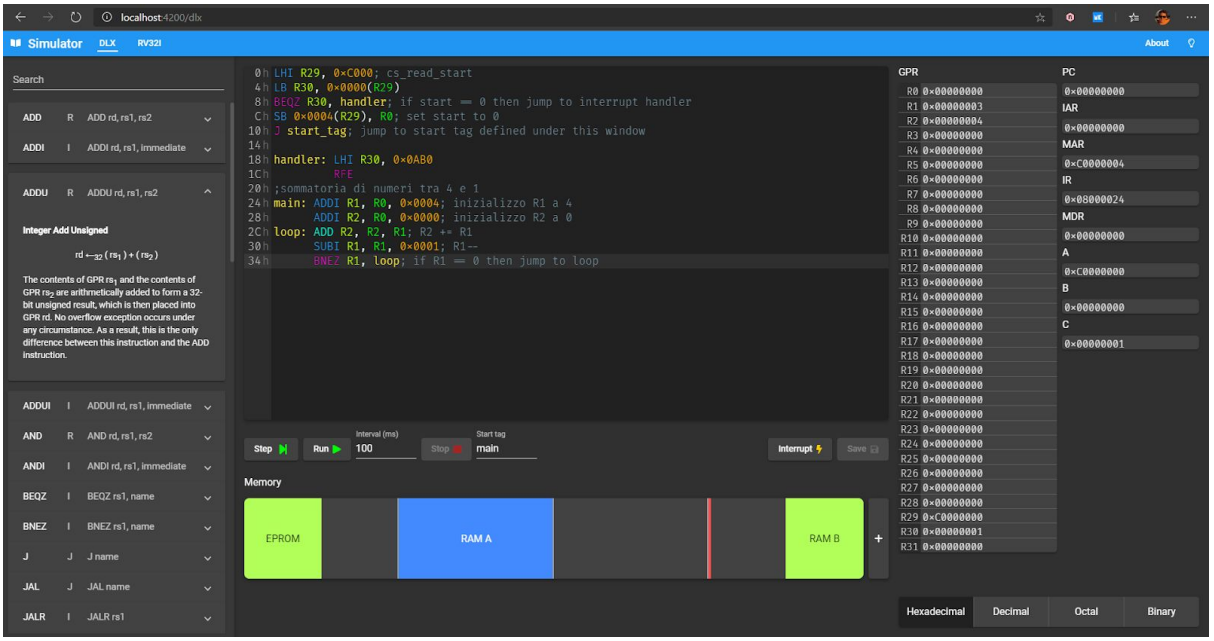

Figura 4.2 - Documentazione e aggiunta codice

Se si preme il pulsante in alto a sinistra compare la barra laterale della documentazione come si può vedere nella figura 4.2. Ogni istruzione è contenuta in un pannello espandibile, da chiuso si legge la codifica dell'istruzione, il tipo e la sintassi, da aperto contiene anche il nome completo dell'istruzione, l'espressione di quello che fa e una descrizione. In alto nel campo search è possibile filtrare la lista di istruzioni. Nella figura 4.2 si può notare anche che è stato scritto il codice per realizzare la sommatoria dei numeri da 1 a 4 con un ciclo, alla fine dell'esecuzione del programma R2 sarà uguale a 1+2+3+4.

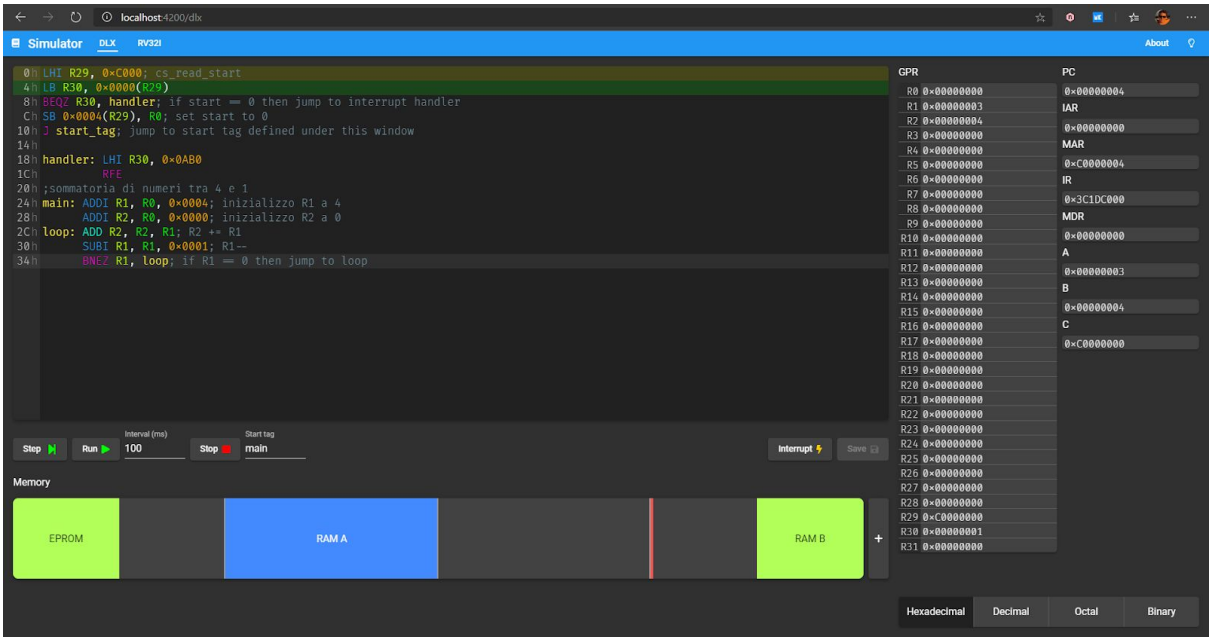

Figura 4.3 - Esecuzione primo step

Nella figura 4.3 si può vedere cosa succede se si preme il pulsante step per la prima volta. La riga evidenziata in giallo è la riga che è appena stata eseguita, mentre in verde viene

evidenziata la prossima riga che verrà eseguita in base al valore del registro Program Counter. Infatti come si può notare a destra il registro PC ha valore 4h che è lo stesso della seconda riga di codice. Ogni riga di codice occupa 32 bit, 4 byte, quindi ogni 4 indirizzi della memoria. Quando il programma è in esecuzione si attiva il pulsante stop che se premuto reimposta il simulatore allo stato iniziale, i registri e la memoria non vengono resettati. La funzione di stop viene eseguita anche se viene modificato il codice, questo per evitare degli effetti indesiderati.

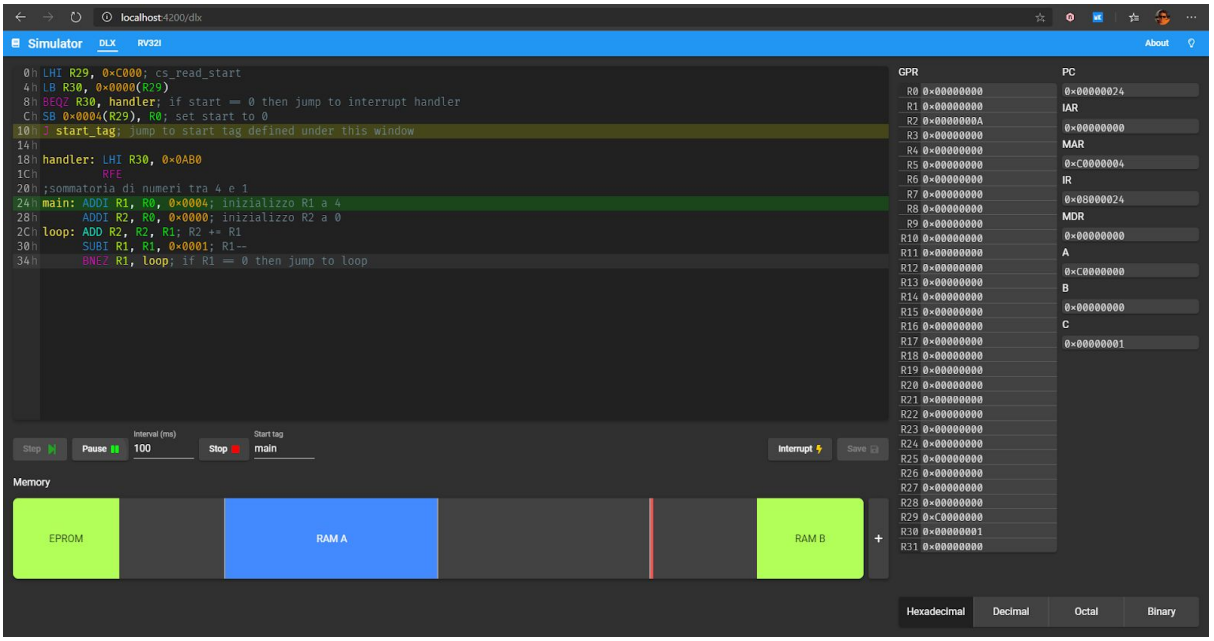

Figura 4.4 - Esecuzione continua ed esempio Jump

Nella figura 4.4 viene mostrato come con un'istruzione Jump agisce sul Program Counter e come la riga evidenziata in verde sia quella a cui punta il PC. Si può anche notare che il programma è in modalità esecuzione continua con un intervallo di 100ms che corrisponde ad un clock di 10Hz. Quando è in esecuzione continua è possibile anche metterla in pausa mediante il pulsante pause. Se in pausa si può riprendere l'esecuzione step by step oppure continua.

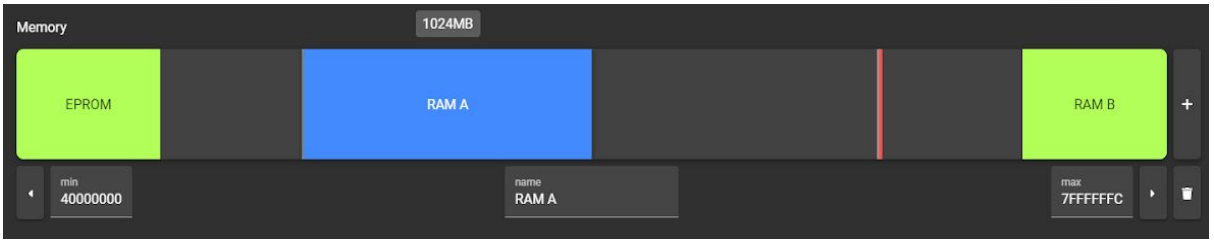

Figura 4.5 - Modifica memoria

In figura 4.5 viene mostrata l'interfaccia di modifica della memoria. Quando col mouse si passa sopra ad un blocco, compare un tooltip con la dimensione espressa in MegaByte del blocco, se non è un blocco normale ma una rete logica invece della dimensione compare il nome della rete e il valore del suo Flip-Flop D. Per motivi di visualizzazione è stata impostata come dimensione minima di un blocco, non di tipo rete logica, 128MB, un blocco di

dimensioni più piccole potrebbe diventare difficile da selezionare. Un blocco comune e una rete logica vengono mostrati in modo diverso, la larghezza del blocco nella vista dipende dalla sua dimensione mentre la rete logica ha larghezza 4 pixel. Quando si clicca su un blocco questo viene selezionato e compaiono i controlli che servono per spostarlo a destra e sinistra, cambiare le sue dimensioni, cambiare il nome ed eliminare il blocco. Se il blocco viene impostato come troppo piccolo allora compare un messaggio di errore. I blocchi EPROM e Start non possono essere eliminati. Il pulsante a destra della vista della memoria serve ad aggiungere un nuovo blocco. Se cliccato aggiunge un blocco di 128MB nel primo indirizzo libero. Ogni modifica fatta alla configurazione della memoria essa viene salvata nel local storage. Anche la configurazione della memoria viene caricata dal localstorage all'avvio dell'applicazione, se non è presente una configurazione nel localstorage allora viene caricata la configurazione di default.

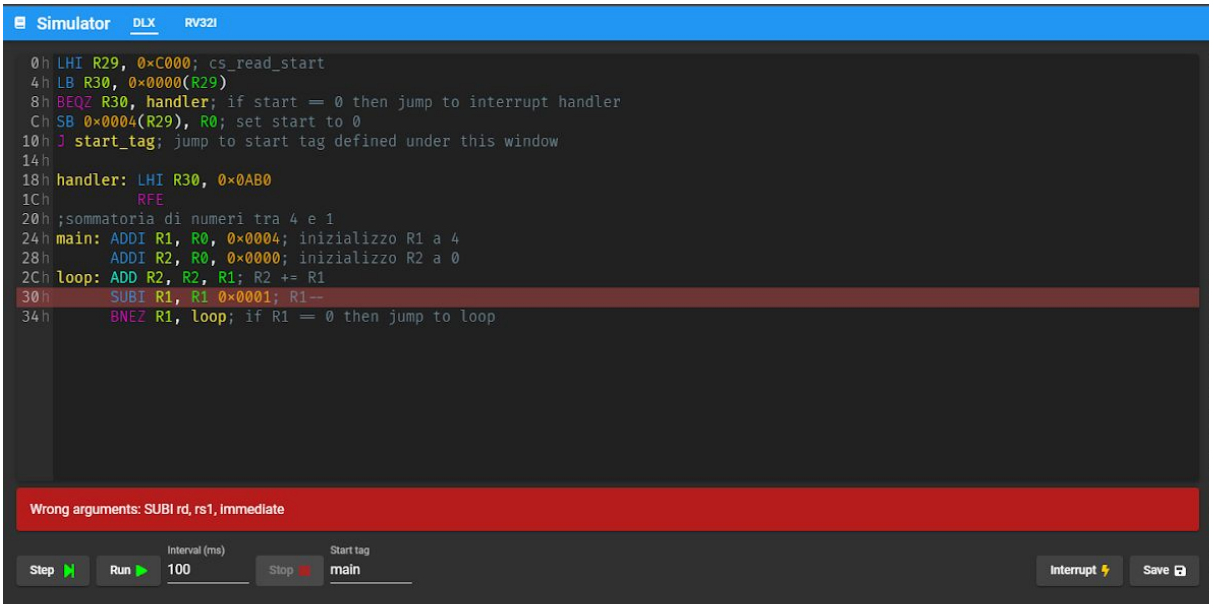

Figura 4.5 - Errore sintattico

Se durante l'esecuzione del codice si incontra un'istruzione che non è presente nel set di istruzioni DLX, o se ci sono errori sintattici nell'istruzione, o se in un'istruzione branch/jump il tag non esiste allora l'interprete lancia un errore che viene mostrato sotto l'editor, e la riga che ha dato errore viene evidenziata in rosso. Se si modifica il codice o si fa partire l'esecuzione l'errore scompare. Nella figura 4.5 si vede che la riga 30h è sintatticamente errata perchè manca una virgola, il programma notifica l'errore sintattico specificando la sintassi corretta per l'istruzione.

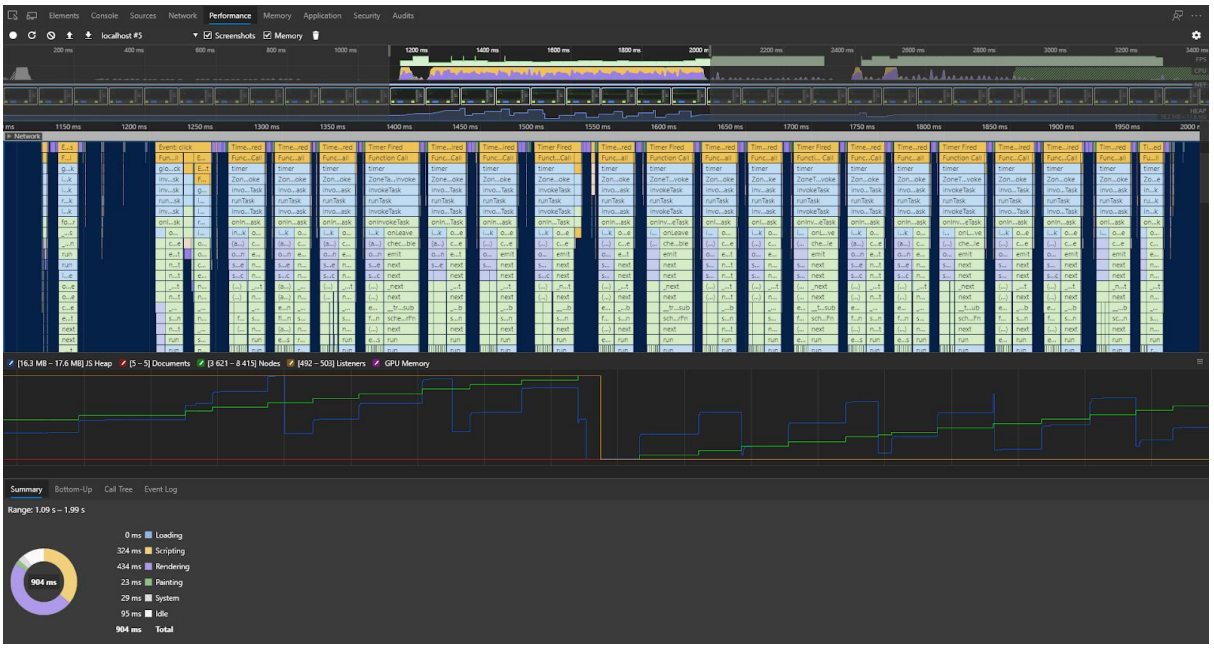

Figura 4.6 - Prestazioni registrate da chromium

Nella figura 4.6 si possono notare quanta ram viene utilizzata dall'applicazione.

5 - Conclusioni

La progettazione e la realizzazione dell'interfaccia grafica e del modello della memoria, in quanto parte in comune con l'altro simulatore, è stata svolta in collaborazione con Alessandro Foglia.

Il lavoro svolto individualmente consiste nella realizzazione dei modelli dei registri, delle documentazioni, l'evidenziazione della sintassi che cambia da architettura ad architettura e la realizzazione dell'interprete che è il cuore del simulatore. La documentazione delle istruzioni è stata ripresa dal libro "The DLX Instruction Set Architecture Handbook" [2] che è stato utile insieme alle slide del corso di Calcolatori Elettronici T del professore Stefano Mattoccia per studiare approfonditamente l'insieme di istruzioni e l'architettura DLX.

L'applicazione rispetta tutti i requisiti che erano stati posti.

In futuri sviluppi si potrebbero aggiungere le istruzioni a floating point di DLX, si potrebbero anche aggiungere facilmente nuove architetture oltre a DLX e RISC-V, andrebbe modificata la vista dei registri in modo da essere più dinamica in modo da funzionare anche con altre architetture.
Un'altra aggiunta sarebbe un sistema di creazione di semplici reti logiche accessibili tramite memoria modificando ed estendendo la classe LogicalNetwork, si potrebbe definire il funzionamento della rete logica tramite un linguaggio dichiarativo creato per questo scopo oppure con un interfaccia grafica. Si potrebbe creare un sistema di comunicazione tra il simulatore ed un'altra applicazione che realizza reti logiche. L'applicazione delle reti logiche potrebbe esportare la rete in un formato leggibile dal simulatore che dopo averla importata simulerebbe anche il funzionamento della rete.

Bibliografia

[1] Alessandro Foglia - “Simulatore RISC-V”

[2] Philip M.Sailer, David R.Kaeli - “The DLX Instruction Set Architecture Handbook”

[3] Stefano Mattoccia - Slide corso Calcolatori Elettronici T Ingegneria Informatica “DLX: implementazione sequenziale”
http://vision.deis.unibo.it/~smatt/DIDATTICA/Calcolatori_Elettronici_T/PDF/07_DLX_sequenziale.pdf

[4] Stefano Mattoccia - Slide corso Reti Logiche T Ingegneria Informatica “Progettazione Diretta”
http://vision.deis.unibo.it/~smatt/DIDATTICA/Reti_Logiche_T/PDF/1_2_Progettazione_diretta.pdf

[5] https://developer.mozilla.org/it/docs/Web/API/Window/localStorage

Ringraziamenti

Ringrazio la mia bellissima ragazza Bianca per il sostegno morale indispensabile.
Ringrazio i miei genitori, i nonni Bruni e tutta la mia famiglia per l'amore e il sostegno che mi hanno sempre dato.
Ringrazio i miei amici del team Anabola, Matt, Bruno, Delta, Bruce, Nick e Tommi, lavorare alla tesi fino alle 3 di notte non sarebbe stato lo stesso senza la loro compagnia.
Ringrazio i miei grandi amici Lollo e Cecca per essere sempre al mio fianco, anche in un traguardo così importante.
Ringrazio i miei compagni di squadra con cui condivido la passione per la pallavolo.
Infine ringrazio il Professor Mattoccia per il sostegno e per l'opportunità di realizzare questa tesi.