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 di un simulatore del processore DLX

CANDIDATO

Filippo Comastri

RELATORE

Stefano Mattoccia

Anno Accademico: 2020/2021

1 INTRODUZIONE

Lo scopo di questa tesi di laurea è quello di estendere e dove possibile migliorare un simulatore DLX e RISC-V già sviluppato in precedenti tesi dagli studenti Alessandro Foglia [1], Fabrizio Maccagnani [2] e Federico Pomponii [3].
Il simulatore funge da strumento didattico per gli studenti del corso universitario Calcolatori Elettronici-T che grazie ad esso possono approfondire e capire al meglio il funzionamento di un vero processore con architettura DLX o RISC-V.
A tal proposito va tenuto conto che l'obiettivo del simulatore è quello di riprodurre in maniera fedele il comportamento di un processore con architettura DLX o RISC-V ma senza riprodurre il funzionamento interno di un processore fisico.

1.1 Architettura RISC-V

Il RISC-V è uno standard open-source di istruzioni ISA (Instruction Set Architecture).
Sebbene sia conveniente parlare dell'ISA RISC-V, RISC-V è in realtà una famiglia di ISA correlati, di cui attualmente esistono quattro ISA di base [1].
Esistono due varianti per i set di istruzione base degli interi, RV32I e RV64I, che forniscono rispettivamente spazi di indirizzi a 32 o 64 bit. La versione presa in esame è RV32I utile anche a scopo didattico, e sarà utilizzato il termine XLEN per fare riferimento alla larghezza in bit di un registro [1].

1.2 Architettura DLX

DLX è un'architettura di microprocessori RISC (Reduced Instruction Set Computer) ideata da John Hennessy e Dave Patterson, 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 [2].
L'ISA DLX possiede un unico spazio di indirizzamento di 4 Gigabyte e definisce 32 registri General Purpose (GPR) a 32 bit, R0-R31. Il registro R0 ha sempre valore 0 mentre il registro R31 è usato per salvare l'indirizzo di ritorno per le istruzioni JAL (Jump And Link) e JALR (Jump And Link Register). I GPR possono essere usati come operandi o come destinazione della maggior parte delle istruzioni.

1.2.1 Modalità di accesso alla memoria

Il DLX prevede un'unica modalità di indirizzamento: indiretto. L'indirizzamento indiretto prevede che l'indirizzo di accesso alla memoria sia ottenuto sommando un valore costante presente nell'istruzione con il contenuto di un registro.

Indirizzo = costante + registro

Il registro è cablato nell'istruzione ma il suo contenuto può cambiare a tempo di esecuzione.
Nel DLX l'indirizzo (a 32 bit) è sempre ottenuto sommando un registro a 32 bit con un valore immediato a 16 bit esteso a 32 bit. [4]

1.2.2 Istruzioni DLX

Le istruzioni DLX hanno lunghezza costante di 32 bit e sono presenti 3 diversi formati: I, R, J.

1.2.2.1 Tipo I

Figura 1.1

Include le istruzioni di Load e Store, operazioni ALU con immediato, le istruzioni di Branch condizionali, 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. [2]
Nelle operazioni Load e ALU RS2/Rd è il registro destinazione (Rd). Nelle Store RS2/Rd è il registro che contiene il valore da salvare in memoria (RS2).
Il campo RS1 indica un registro che viene utilizzato come argomento dell'istruzione.

1.2.2.2 Tipo R

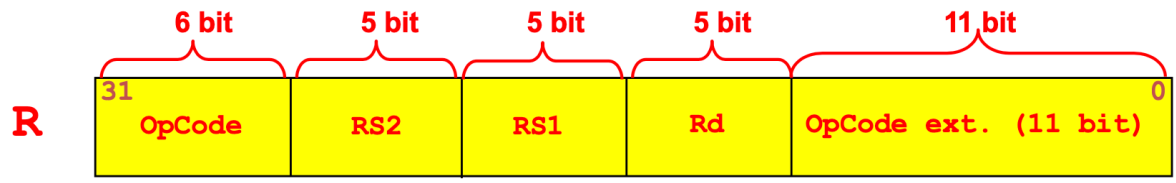

Figura 1.2

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 e le diverse operazioni sono distinte dal campo funzione.
Il campo RS1 indica il registro GPR che viene utilizzato come primo operando dell'istruzione.
Il campo RS2 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. [2]

1.2.2.3 Tipo J

Figura 1.3

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 immediato/offset 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 immediato/offset rappresenta un indirizzo assoluto, mentre per l'istruzione RFE non è utilizzato.

2 STRUMENTI UTILIZZATI

Il simulatore è stato sviluppato con Angular, uno dei framework Javascript più popolari per sviluppare applicazioni web a singola pagina.
AngularJS è la prima versione del framework che fu sviluppata nel 2009 con l'intenzione di poter facilitare la realizzazione delle applicazioni attraverso l'uso di un'estensione del linguaggio HTML.
A partire da Angular 2 il framework fu completamente riscritto rispetto ad AngularJS, rendendo le successive versioni incompatibili con quest'ultima. Quando si parla di Angular ci si riferisce solitamente alle versioni dalla seconda in poi che furono rilasciate a partire dal 2014 e che ebbero un notevole successo. Ad oggi siamo alla versione 12.1.1 rilasciata a giugno 2021.
Lo sviluppo di una nuova versione di Angular ha permesso di renderlo più semplice e snello e di diminuire il tempo necessario per imparare ad utilizzare il framework.
Le applicazioni Angular sono strutturate in Componenti che facilitano la riusabilità e la manutenibilità dell'applicazione.
È stato inoltre messo a disposizione degli sviluppatori uno strumento come Angular CLI (Command Line Interface) che semplifica notevolmente la fase di creazione, sviluppo, test e deploy di un progetto tramite semplici comandi.

Angular supporta Javascript ma è consigliato l'utilizzo del linguaggio Typescript, sviluppato da Microsoft, che è un'estensione di Javascript. Typescript non è però un linguaggio interpretato come 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.

Nel progetto è stato anche utilizzato Angular Material, una libreria grafica per Angular, che aggiunge gli elementi grafici di base seguendo le specifiche di Material Design.

Per le icone è stato utilizzato Font Awesome che permette, specificando solamente un prefisso di stile e il nome dell'icona, di utilizzarle ovunque.

3 BASE DI PARTENZA

Prima del mio lavoro il simulatore comprendeva già una serie di funzionalità, realizzate da altri studenti [1][2][3] nell'ambito della loro tesi di laurea, che verranno brevemente descritte di seguito.
È presente un editor di testo nel quale poter scrivere codice assembly con la possibilità di scegliere quali istruzioni utilizzare: DLX o RISC-V. Tramite un tasto è possibile salvare nella memoria locale del browser il contenuto dell'editor di testo.
In una sezione apposita è presente la documentazione delle istruzioni DLX o RISC-V disponibili.
È possibile eseguire il codice osservando a tempo di esecuzione lo stato dei registri. Durante l'esecuzione è anche possibile generare interrupt. Si può inoltre definire da quale tag far partire l'esecuzione e l'intervallo di tempo tra l'esecuzione di un'istruzione e la successiva.
In un'apposita area è possibile aggiungere o rimuovere spazi di memoria, configurare le loro caratteristiche e visualizzare il valore di un determinato indirizzo di memoria.
In questa stessa area è possibile modificare e personalizzare la rete di avvio e utilizzare una rete logica per l'accensione e lo spegnimento di un led. È anche possibile visualizzare lo schema ai morsetti delle reti logiche e modificare gli indirizzi dei Chip-Select tramite un'apposita interfaccia.

4 NUOVE FUNZIONALITÀ

Nell'ambito di questa tesi laurea sono state aggiunte le seguenti funzionalità:

- Possibilità di salvare in locale un file di testo contenente il codice scritto nell'editor
- Possibilità di visualizzare i valori in memoria in maniera più dettagliata con l'aggiunta di un'interfaccia grafica intuitiva per poter navigare tra le memorie
- Salvataggio automatico nella memoria del processore della codifica di ciascuna istruzione scritta nell'editor
- Aggiunta di un contatore tra le reti logiche che possono essere utilizzate all'interno del simulatore

4.1 Salvataggio codice in locale

Per permettere agli studenti di mantenere una copia del proprio lavoro svolto sul simulatore, è stata aggiunta la possibilità di salvare il codice contenuto nell'editor di testo su un file che viene scaricato in locale.

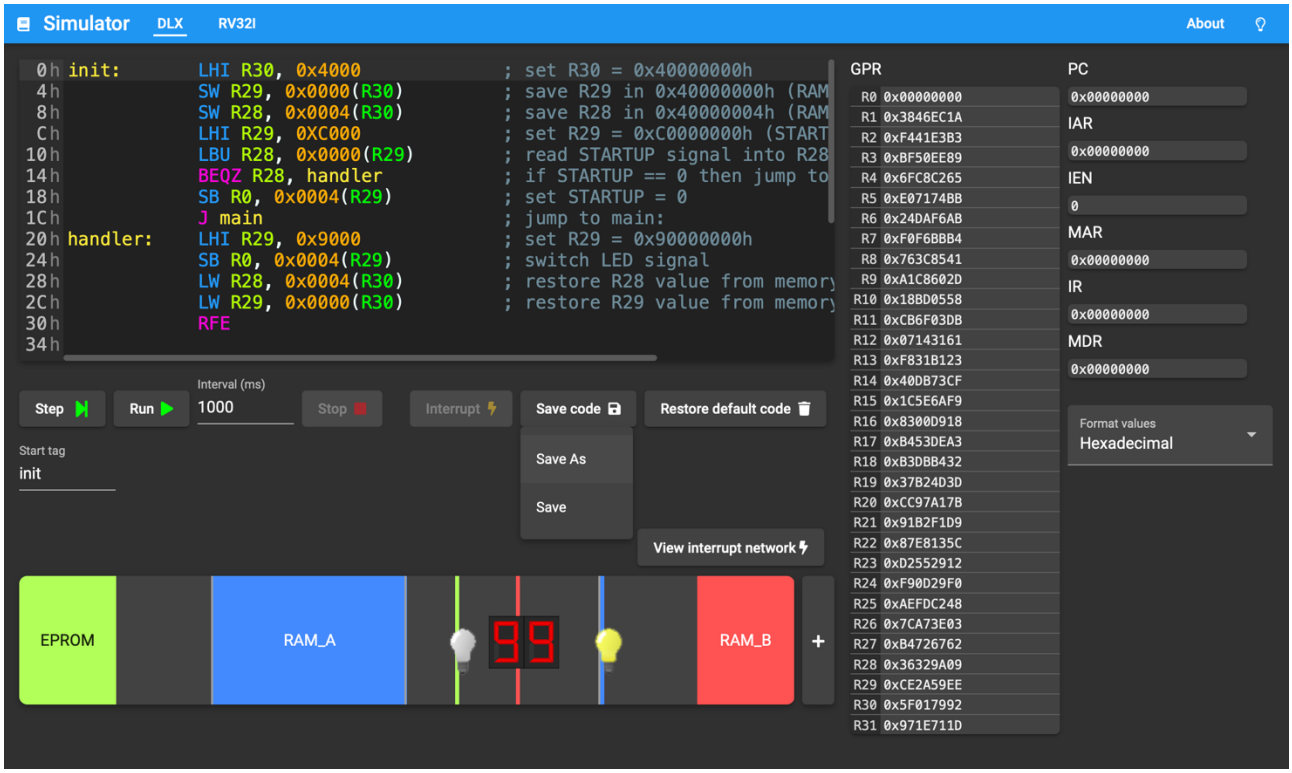

Figura 4.1

Come si vede in Figura 4.1 cliccando sul tasto “*Save Code*” si apre un menù a tendina nel quale si può scegliere che tipo di salvataggio effettuare. Cliccando su *“Save”* il contenuto dell’editor viene salvato nella memoria locale del browser. Cliccando su *“Save As”* invece si apre una finestra di dialogo (Figura 4.2) nella quale è possibile scegliere il nome del file nel quale viene salvato il codice scritto nell’editor.

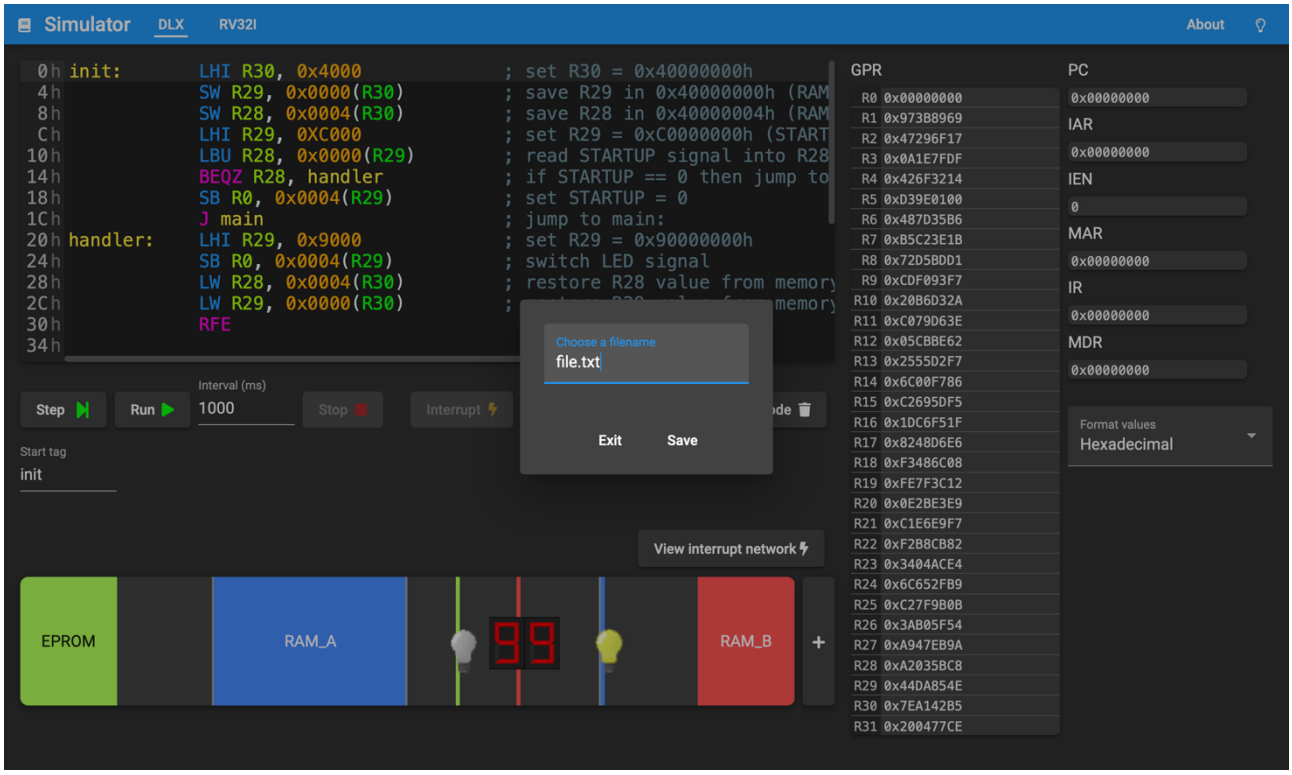

Figura 4.2

Dalla finestra di dialogo cliccando su *“Save”* il file verrà scaricato dal browser e sarà possibile visualizzarlo nella cartella dei Downloads.

4.2 Visualizzazione dettagliata della memoria

È stata migliorata la visualizzazione dei valori in memoria, aggiungendo la possibilità di visualizzare i valori non solamente distanziati di 0004h ma anche solamente di un byte. In questo modo è possibile analizzare in modo più preciso e puntuale il valore di ciascuna cella di memoria.

4.2.1 Visualizzazione byte per byte

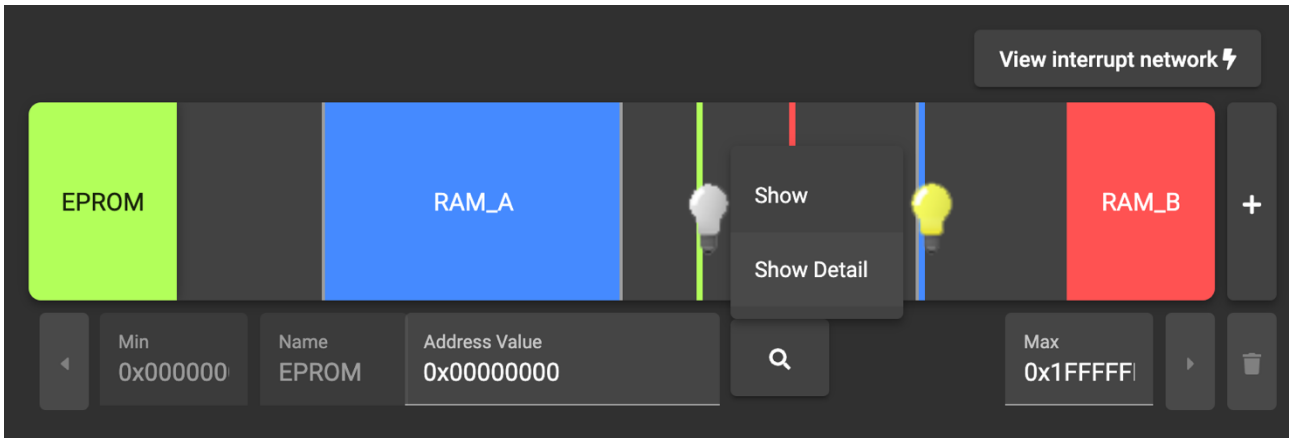

Figura 4.3

Nella sezione delle memorie (Figura 4.3) cliccando sull'icona della lente di ingrandimento si apre un menù a tendina: cliccando su *"Show"* si visualizza il contenuto della memoria a partire dell'indirizzo scritto nel campo *"Address Value"* con gli indirizzi separati di 0004h; cliccando su *"Show Detail"* viene aperta la visualizzazione più dettagliata della memoria con gli indirizzi distanziati di un byte.

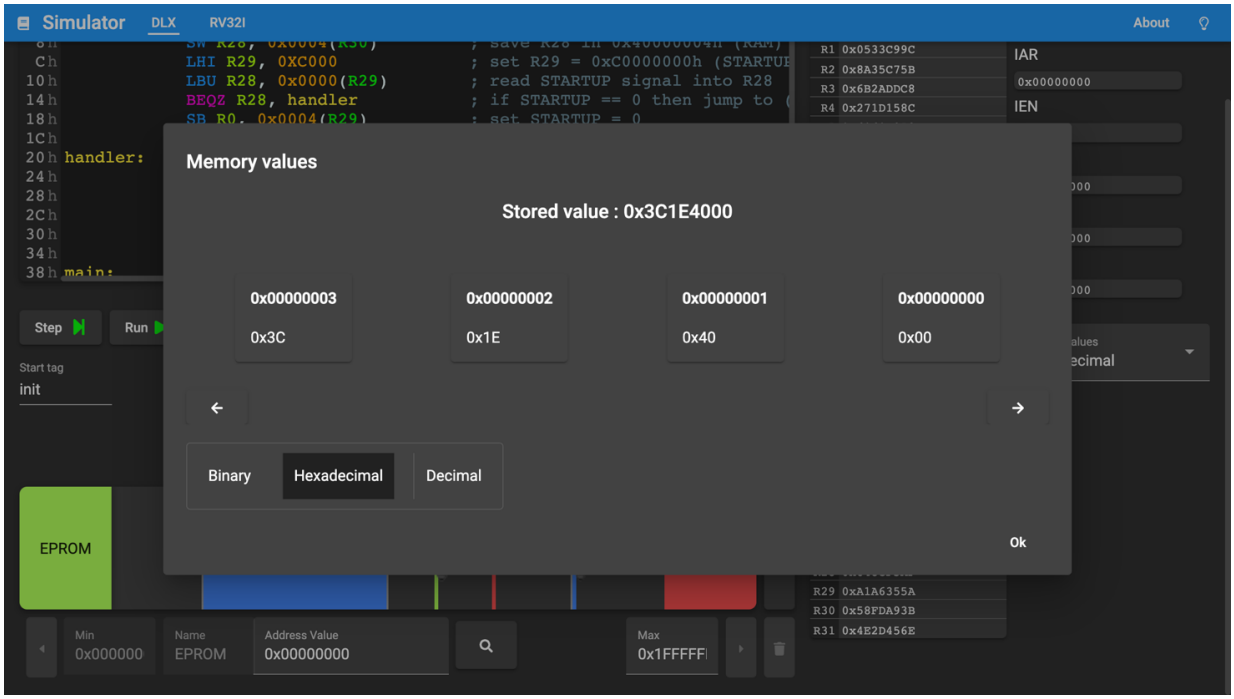

Figura 4.4

In Figura 4.4 vediamo l'interfaccia della visualizzazione dettagliata della memoria. Nei 4 blocchi vengono mostrati byte per byte i valori contenuti in memoria a partire dall'indirizzo selezionato nel campo *"Address Value"*. I valori vengono salvati in memoria rispettando la convenzione *Little Endian* secondo la quale una grandezza numerica viene memorizzato in memoria partendo dal byte meno significativo finendo con quello più significativo. Come esempio osserviamo la Figura 4.4. Il valore memorizzato a partire dall'indirizzo 0x0000000 è 0x3C1E4000 (*"Stored value"*). Tale valore è lungo 4 byte, quindi ciascun byte verrà memorizzato nel range di indirizzi 0x00000000 – 0x00000003. Secondo la convenzione *Little Endian* la cella di memoria corrispondente all'indirizzo 0x0000000 conterrà il byte meno significativo 0x00 mentre la cella corrispondente all'indirizzo 0x00000003 conterrà il byte più significativo 0x3C.
È anche possibile decidere in che formato visualizzare i valori salvati in memoria: esadecimale, binario o decimale.

4.2.2 Navigazione agile in memoria

Per facilitare la navigazione in memoria è stata sviluppata una funzionalità che permette di muoversi in modo intuitivo avanti e indietro tra le varie celle di memoria tramite due bottoni contenti l'icona di una freccia.

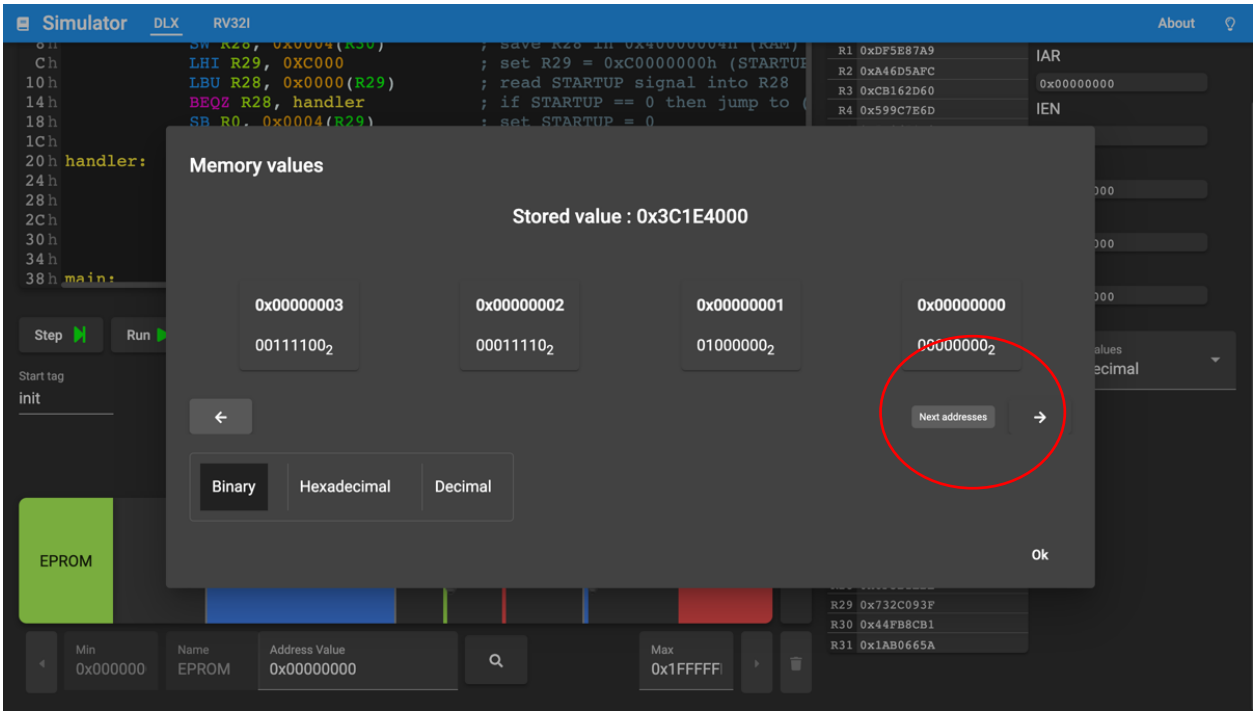

Figura 4.5

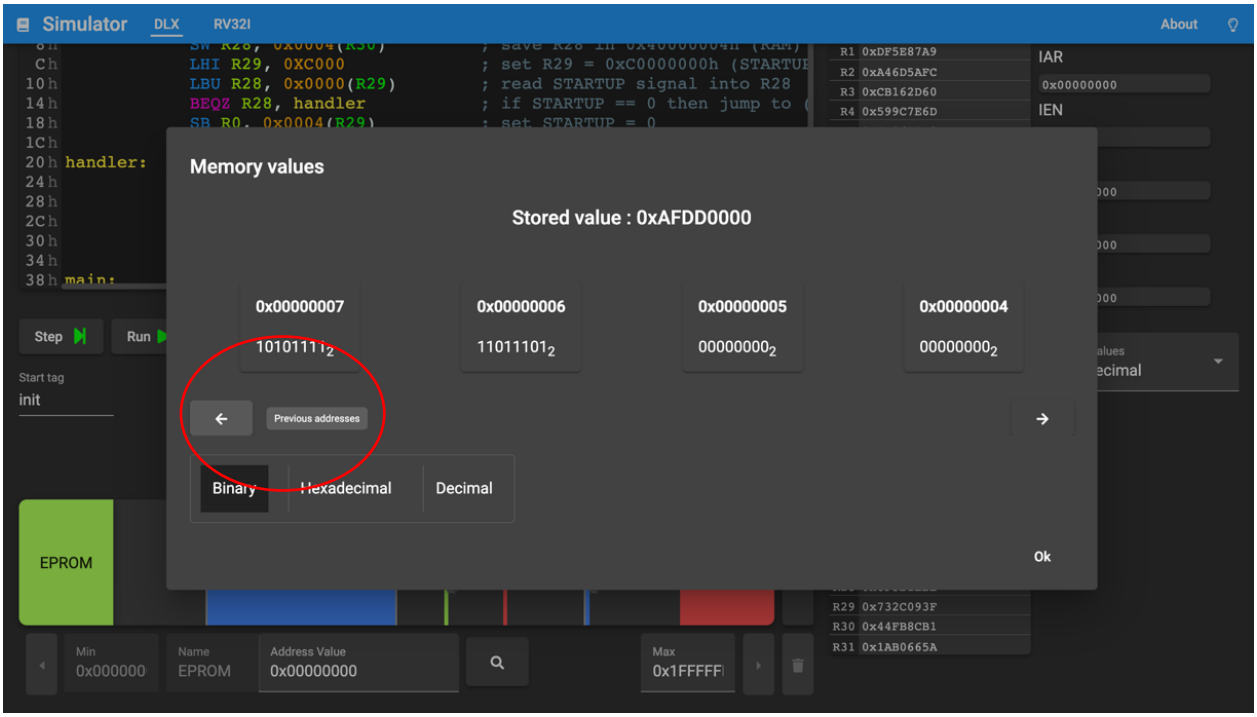

Figura 4.6

Nelle figure 4.5 e 4.6 sono evidenziati i due bottoni che permettono di muoversi avanti (Figura 4.5) e indietro (Figura 4.6) in memoria. Passando con il mouse sopra il bottone compare anche una breve descrizione della funzionalità che svolgono per favorire l'usabilità.
Se ad esempio ci troviamo nella situazione della Figura 4.5 e clicchiamo sul bottone con la freccia verso destra ci "si sposterà" avanti nella memoria e verranno visualizzate le successive 4 celle di memoria separate di un byte l'una dall'altra.
Ci si trova così nella situazione che vediamo in Figura 4.6 e da questa interfaccia ci si potrà spostare ancora in avanti di 4 celle di memoria oppure tornare indietro cliccando sul bottone con la freccia verso sinistra. In questo modo si tornerà a visualizzare l'interfaccia delle Figura 4.5.
Questo meccanismo intuitivo e rapido permette di muoversi agilmente in memoria, senza dover ogni volta chiudere l'interfaccia di visualizzazione dei valori in memoria e dover scrivere nuovamente l'indirizzo di cui si vogliono visualizzare i valori.

4.3 Salvataggio in memoria del codice

Il codice assembly risiede in memoria e andando quindi a leggere l'indirizzo di memoria nel quale è salvata la codifica dell'istruzione dovremo visualizzare quest'ultima.
In particolare, nel simulatore il codice viene sempre salvato a partire dall'indirizzo 0x0000000h e l'indirizzo di ciascuna istruzione viene indicata a lato dell'editor di testo per ciascuna riga di codice.

```
 0h init:      LHI R30, 0x4000          ; set R30 = 0x40000000h
 4h            SW R29, 0x0000(R30)      ; save R29 in 0x40000000h (RAM)
 8h            SW R28, 0x0004(R30)      ; save R28 in 0x40000004h (RAM)
 Ch            LHI R29, 0XC000          ; set R29 = 0xC0000000h (STARTUP
10h            LBU R28, 0x0000(R29)     ; read STARTUP signal into R28
14h            BEQZ R28, handler        ; if STARTUP == 0 then jump to (
18h            SB R0, 0x0004(R29)       ; set STARTUP = 0
1Ch            J main                   ; jump to main:
20h handler:   LHI R29, 0x9000          ; set R29 = 0x90000000h
24h            SB R0, 0x0004(R29)       ; switch LED signal
28h            LW R28, 0x0004(R30)      ; restore R28 value from memory
2Ch            LW R29, 0x0000(R30)      ; restore R29 value from memory
30h            RFE
34h
38h                                     ; Fibonacci sequence
```

Figura 4.7

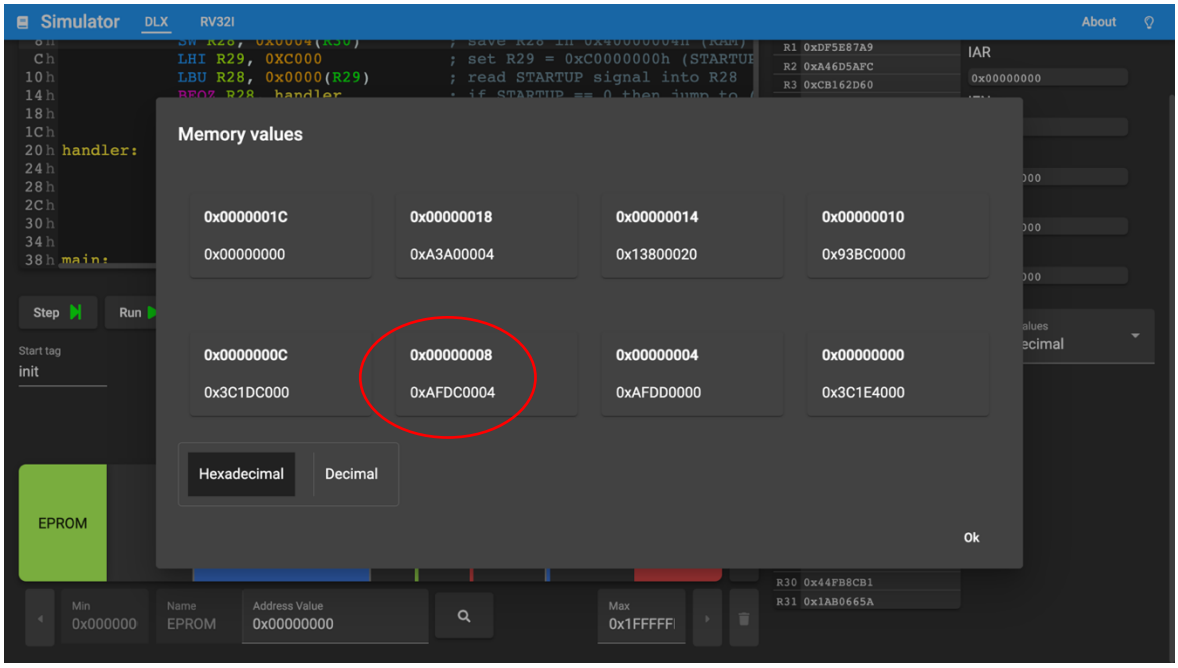

Figura 4.8

Osserviamo ad esempio in Figura 4.7 che la prima riga di codice contiene l'istruzione *"SW R28, 0x0004(R30)"* e a lato è indicato che la codifica di quest'ultima è salvata all'indirizzo 0x0000008h. Vediamo in Figura 4.8 che l'indirizzo di memoria 0x00000008h contiene la codifica dell'istruzione sopra citata.

4.3.1 Esempio codifica istruzione

Utilizziamo la visualizzazione byte per byte della memoria per analizzare come viene salvata la codifica dell'istruzione *"SW R28, 0x0004(R30)".*

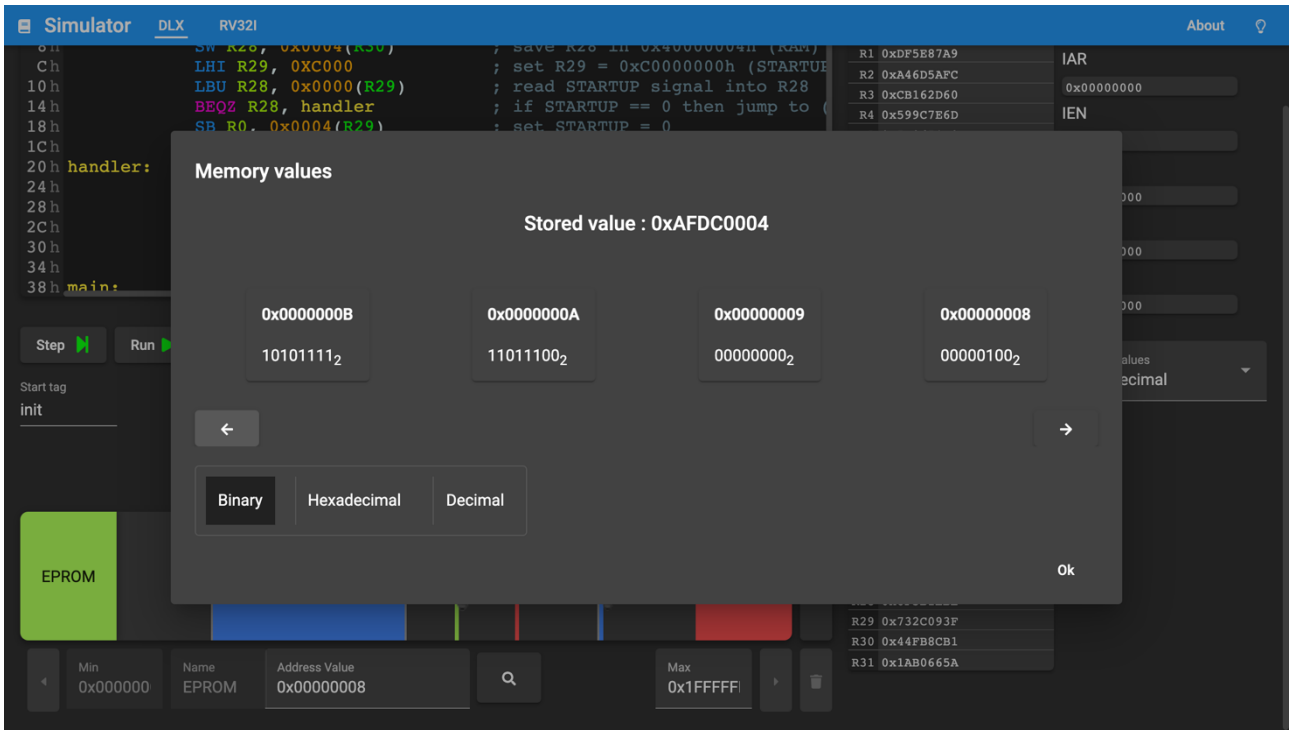

Figura 4.9

L'istruzione *Store* rientra nelle istruzioni di tipo I che come già ben spiegato nel capitolo *1.2.2.1* ha un formato che comprende nell'ordine: un codice operativo (6 bit), una codifica del registro destinazione (5 bit), una codifica del registro che contiene il valore da salvare in memoria (5 bit) e un immediato a 16 bit.
Il codice operativo per la SW è 101011 [5].
Il registro destinazione è il numero 30 che codificato in binario con 5 bit produce il codice 11110.
Il registro che contiene il valore da salvare è il numero 28 che codificato in binario con 5 bit produce il codice 11100.
L'immediato è 0x0004 che codificato in binario con 16 bit produce il codice $(0)^{13}100$.
Vediamo in Figura 4.9 che le celle di memoria contengono concatenati fra loro i codici sopra elencati, con l'ordine che va dal byte meno significativo a quello più significativo seguendo sempre la convenzione *Little Endian*.

4.4 Aggiunta Contatore

Tra le novità introdotte nel simulatore c'è la possibilità di aggiungere nello spazio di indirizzamento un contatore che può essere comandato tramite letture o scritture a specifici Chip-Select.

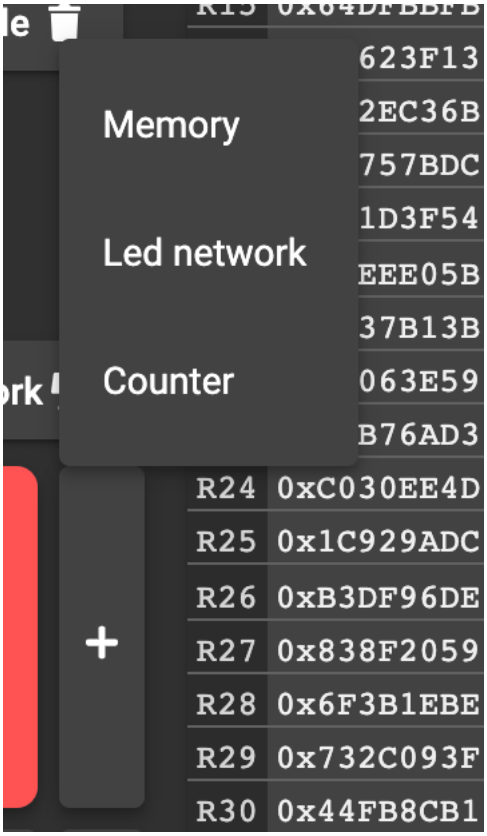

Figura 4.10

Cliccando sul tasto "+" collocato nella zona dei device è possibile aggiungere un contatore cliccando su *"Counter"*. Comparirà quindi tra i device l'icona del contatore.

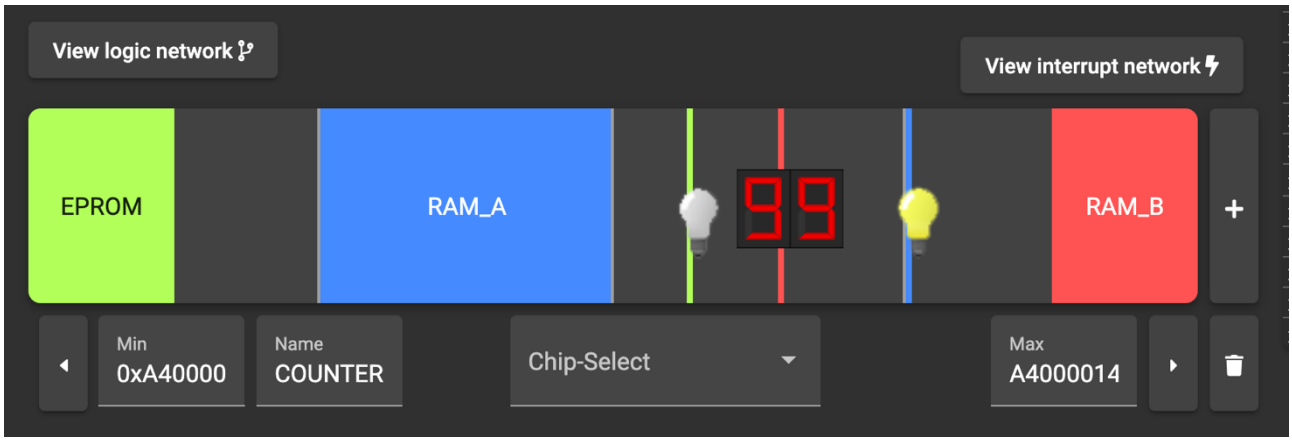

Figura 4.11

Come per le memorie e per il led è possibile scegliere gli indirizzi in cui mappare il contatore e gli indirizzi in cui mappare i chip select di quest'ultimo.
Inoltre, selezionando il contatore dalla sua icona è possibile visualizzare lo schema ai morsetti della rete logica cliccando su *"View logic network"*.

Figura 4.12

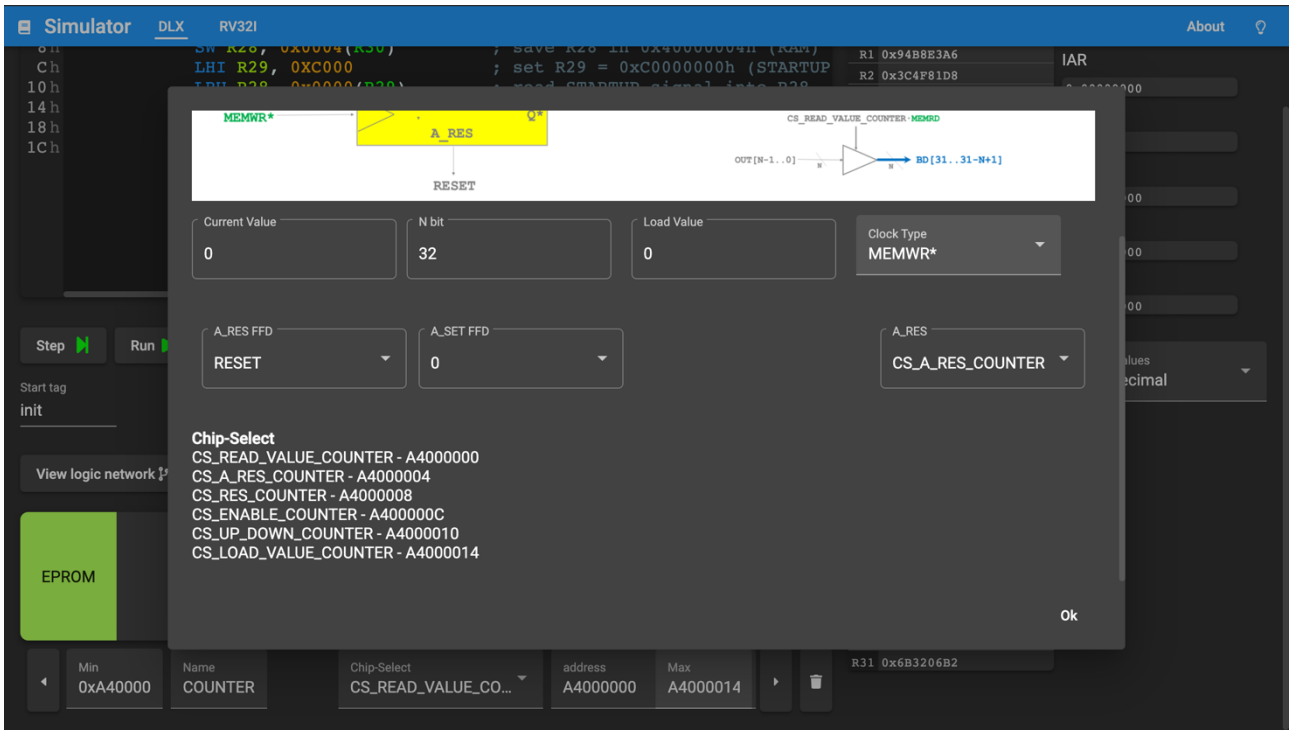

Figura 4.13

In Figura 4.12 vediamo lo schema ai morsetti del contatore e nella parte bassa dell'interfaccia una serie di campi che permettono di personalizzare i chip select.
Scorrendo in basso la finestra di dialogo compare una lista dei chip select e degli indirizzi in cui sono mappati. (Figura 4.13)
Il campo *"N bit"* permette di selezionare la base di conteggio del contatore mentre nel campo *"Current Value"* possiamo osservare il valore corrente mantenuto dal contatore.
Dal campo *"Clock type"* è possibile decidere se utilizzare il segnale MEMWR* di fine scrittura in memoria o se utilizzare MEMRD*, ossia il segnale di fine lettura. Tutti i chip select saranno quindi sincroni al segnale selezionato. Se è selezionato come clock MEMRD* le letture impartiranno il comando associato al chip select, viceversa se è selezionato MEMWR* saranno le scritture ad impartire il comando.
Di seguito analizzeremo il significato di ciascuno dei chip select:

- *CS_UP_DOWN_COUNTER*: stabilisce se il conteggio va effettuato in avanti o all'indietro. Di default il conteggio parte in avanti ed effettuando letture o scritture all'indirizzo in cui è mappato il chip select si inverte la direzione. È necessario un flip-flop per mantenere la corrente direzione di conteggio.
- *CS_ENABLE_COUNTER*: abilita o disabilita il conteggio. Effettuando letture o scritture a questo chip select si incrementa o decrementa il valore del contatore.
- *CS_LOAD_COUNTER*: imposta il valore di conteggio ad un valore fornito dall'esterno sugli ingressi L [N-1..0]. Il valore degli ingressi L è selezionabile dal campo *"Load Value"* dov'è possibile scrivere il valore che si desidere immettere.
- *CS_RES_COUNTER*: resetta il contatore in modo sincrono.
- *CS_A_RES_COUNTER*: resetta il contatore in modo asincrono.
- *CS_READ_VALUE_COUNTER*: effettuando una lettura a questo chip select si ottiene il valore corrente memorizzato nel contatore. È ovviamente messo in and logico con il segnale di MEMRD perché il valore può essere ottenuto solamente tramite una lettura, una scrittura a questo chip select non avrebbe alcun senso.

4.4.1 Esempi di utilizzo del contatore

Vediamo ora un semplice esempio di codice assembly che utilizza il contatore e analizziamo i comportamenti ottenuti.

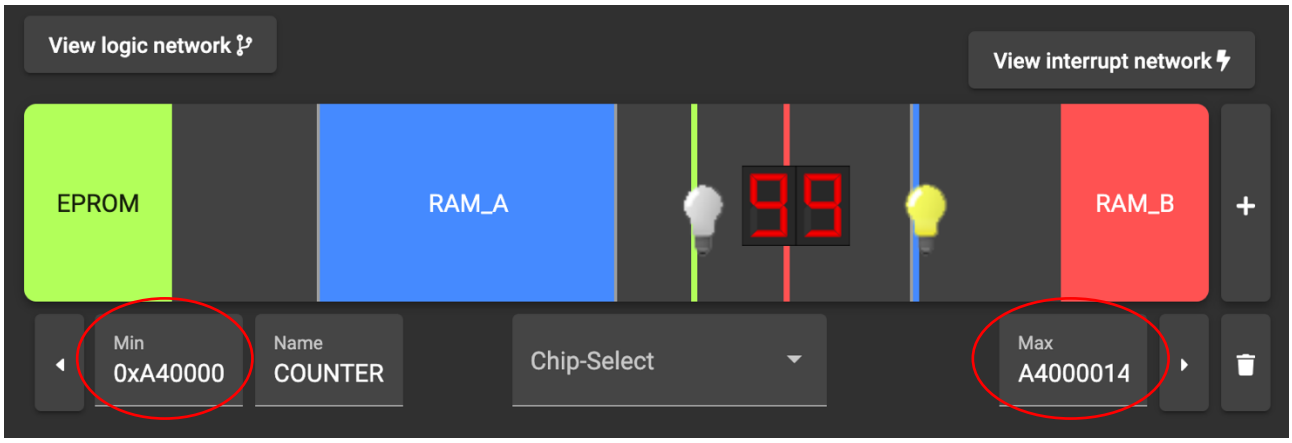

Figura 4.14

Il contatore è mappato (Figura 4.14) nel range di indirizzi che va da 0xA4000000 a 0xA4000014 mentre gli indirizzi dei chip select sono quelli in Figura 4.13.

4.4.1.1 Esempio 1

```
 0h          LHI R30, 0xA400          ;
 4h          SW R29, 0x000C(R30)      ; increments counter
 8h          SW R29, 0x000C(R30)      ; increments counter
 Ch          SW R29, 0X0010(R30)      ; reverse counting
10h          SW R29, 0x000C(R30)      ; decrements counter
14h          SW R29, 0x0008(R30)      ; syncronous reset
```

Figura 4.15

La prima riga di codice (*0h*) prepara il registro R30 per essere utilizzato successivamente nelle store.
La seconda e terza istruzione (*4h, 8h*) effettuano delle scritture al chip select *CS_ENABLE_COUNTER*. Essendo selezionato come clock type il MEMWR* verrà impartito il comando di enable ed essendo di default impostato il conteggio in avanti, il contatore verrà incrementato di due.

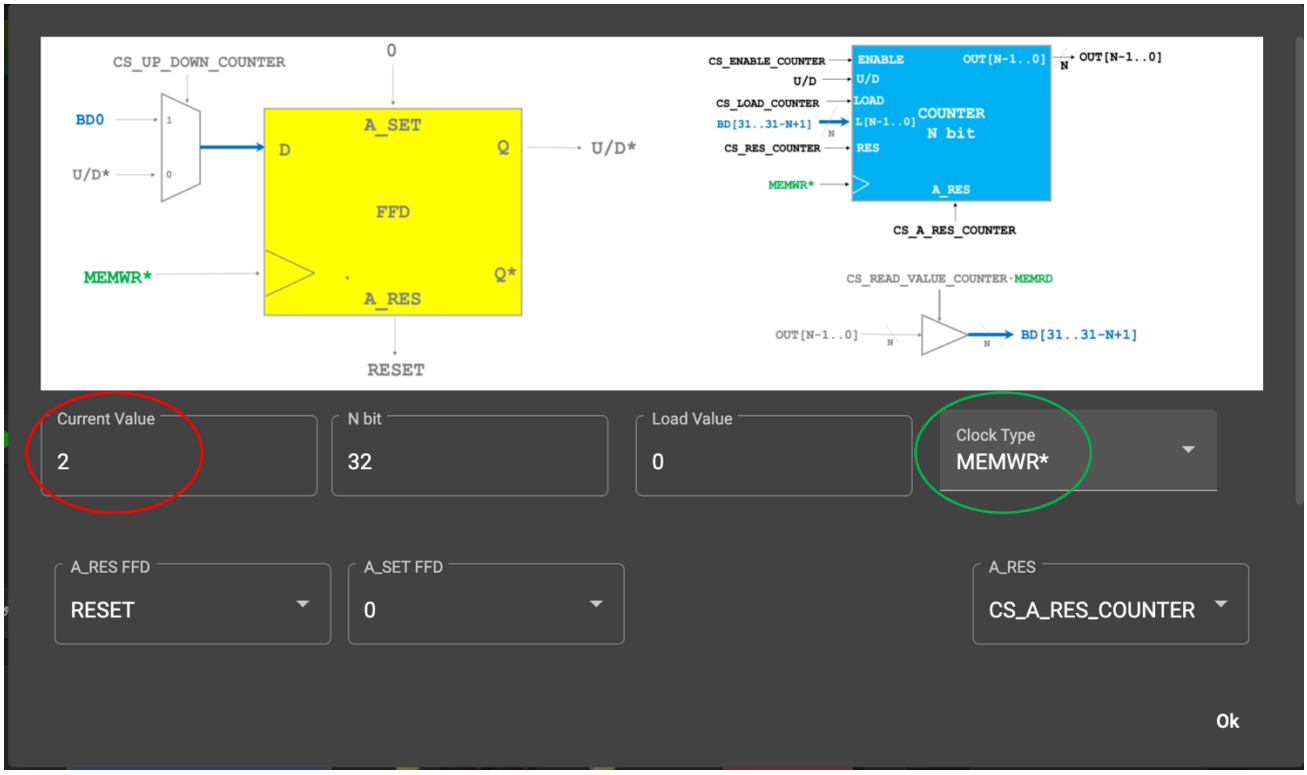

Figura 4.16 - Contatore incrementato di due e clock type impostato a MEMWR*

La quarta istruzione (*Ch*) effettua una scrittura a *CS_UP_DOWN_COUNTER* invertendo la direzione del conteggio, quindi la successiva scrittura (*10h*) al chip select *CS_ENABLE_COUNTER* decrementerà il valore corrente del contatore di uno.
L'ultima istruzione effettua una scrittura al chip select *CS_RES_COUNTER* che resetta in modo sincrono il contatore portando il valore corrente a 0.

4.4.1.2 Esempio 2

```
0h      LHI R30, 0xA400 ;
4h      LW R29, 0x0014(R30) ;
8h      SW R29, 0x000C(R30) ;
Ch      LW R29, 0x000C(R30) ;
10h     LW R28, 0X0000(R30) ;
```

Figura 4.17

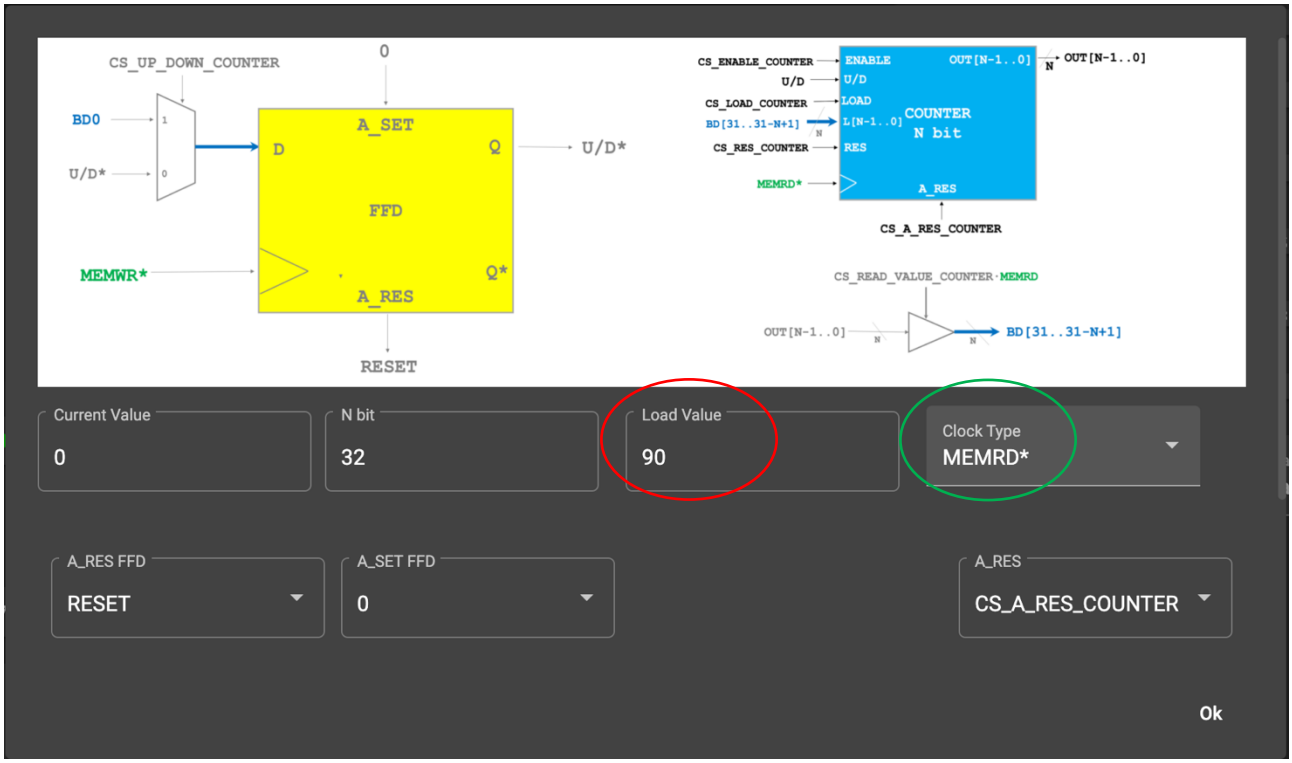

Figura 4.18

In questo esempio abbiamo selezionato come clock type MEMRD* quindi per attivare un comando bisognerà effettuare una lettura al chip select associato. Notiamo anche in Figura 4.18 che abbiamo impostato sugli ingressi L il valore 90.
La prima riga di codice (*0h*) prepara il registro R30 per essere utilizzato successivamente nelle load.
La seconda riga di codice (*4h*) carica nel contatore il valore presente sugli ingressi L effettuando una lettura al chip select *CS_LOAD_VALUE_COUNTER*.

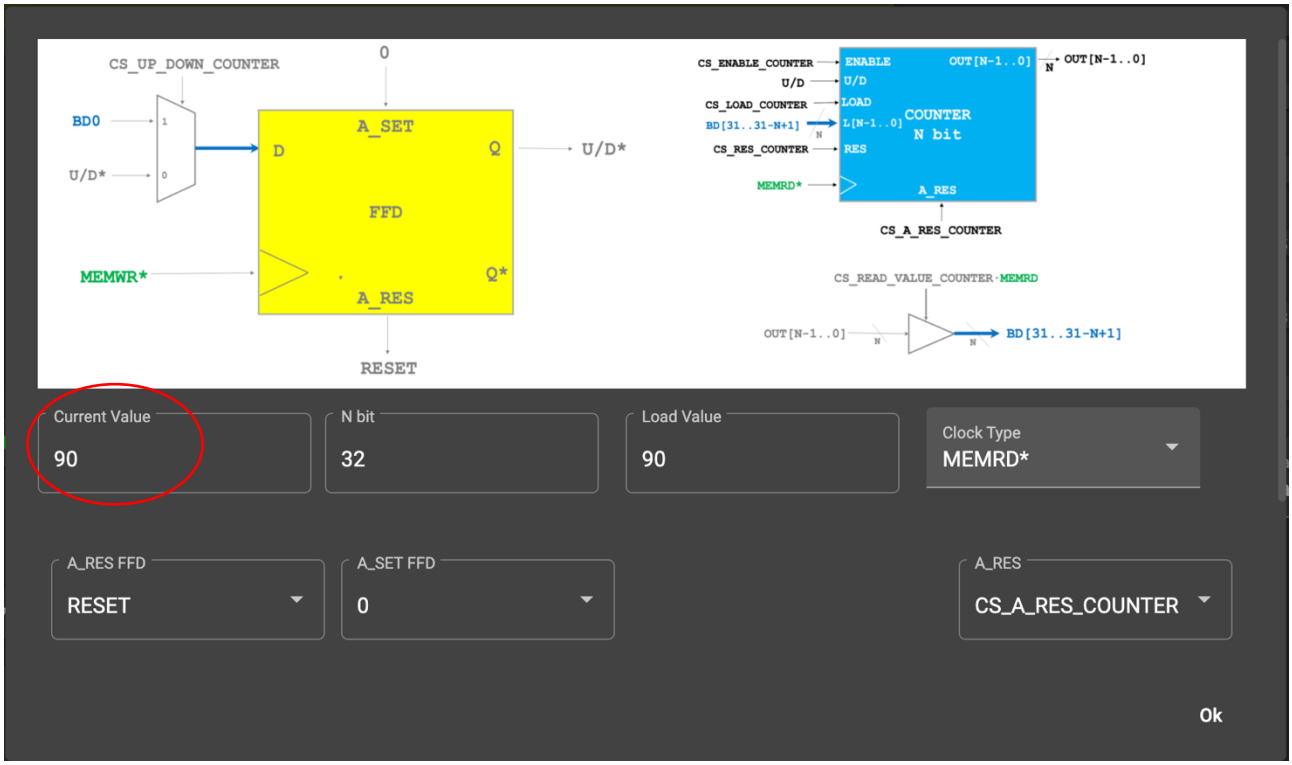

Figura 4.19

In Figura 4.19 notiamo come il valore corrente del contatore sia stato inizializzato con quello presente sugli ingressi L.
La terza istruzione (*8h*) non produce alcun effetto perché essendo impostato come clock MEMRD* il comando viene eseguito in corrispondenza delle letture e non delle scritture.
La quarta istruzione (*Ch*) invece effettua una lettura al *CS_ENABLE_COUNTER* incrementando il valore del contatore di uno.
L'ultima istruzione (*10h*) effettua una lettura al *CS_READ_VALUE_COUNTER* andando a salvare il valore del contatore nel registro R28.

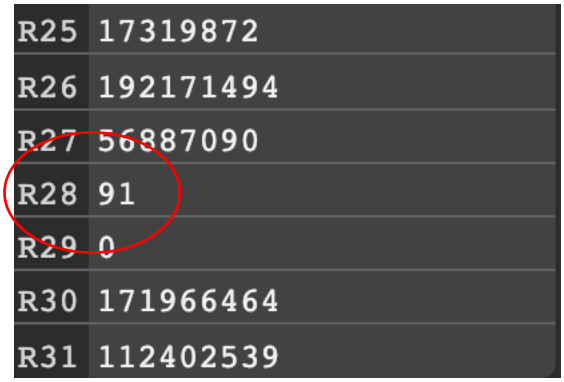

Figura 4.20

Vediamo in Figura 4.20 che il registro R28 contiene il valore 91 che è il valore iniziale del contatore incrementato di 1.

5 SVILUPPI FUTURI

La parte del simulatore nella quale possono essere apportate significative migliorie è quella che riguarda la simulazione del clock e dei cicli di bus in lettura e scrittura. Ad oggi è solamente possibile selezionare in modo statico quale tipo di segnale collegare al clock, ma i segnali di MEMRD* e MEMWR* sono fittizi non essendo prodotti da veri cicli di bus. Sarebbe utile affiancare al simulatore già esistente un altro applicativo in grado di simulare i cicli di bus in lettura e scrittura ogni qualvolta avvengano delle load o store in memoria, così da rendere molto più realistici e fedeli alla realtà i segnali che vengono collegati al clock.
Con questa novità sarebbe possibile anche inserire nel progetto la gestione di periferiche I/O e simularne i processi di lettura e scrittura.
Sarebbe inoltre utile affiancare all'applicazione un tool o un editor che permetta di creare delle reti logiche più complesse e articolate oltre a quelle già presenti nel simulatore.

6 CONCLUSIONI

Lo scopo del progetto di estendere e migliorare il simulatore è stato raggiunto e tutte le nuove funzionalità introdotte funzionano correttamente e svolgono il compito per il quale sono state progettate. Sono stati inoltri corretti alcuni piccoli errori sorti durante l'utilizzo del simulatore.
Tutte le modifiche sono state apportate seguendo le linee guida progettuali scelte dagli studenti che hanno lavorato al simulatore prima di me, cercando inoltre di favorire future modifiche preservando scalabilità e modularità.
Gli argomenti trattati durante questa tesi di laurea sono stati principalmente quelli già visti durante il corso di Calcolatori Elettronici T, il che mi ha permesso di approfondirli e di consolidare le mie conoscenze.
Ho inoltre approfondito le mie conoscenze per quel che riguarda le tecnologie web avendo avuto l'occasione di lavorare con Angular, un framework che non avevo mai utilizzato in passato.

BIBLIOGRAFIA

[1] Alessandro Foglia - “Progetto di un simulatore di RISC-V per scopi didattici”, Tesi di laurea AA 2018/19

[2] Fabrizio Maccagnani - “Progetto di un simulatore di DLX per scopi didattici”, Tesi di laurea AA 2018/19

[3] Federico Pomponii – “Sviluppo di un simulatore DLX per scopi didattici”, Tesi di laurea AA 2019/20

[4] Materiale del corso di Calcolatori Elettronici T, Ingegneria Informatica, Università di Bologna, tenuto da Stefano Mattoccia. Sezione 03 Linguaggio Macchina http://vision.deis.unibo.it/~smatt/DIDATTICA/Calcolatori_Elettronici_T/PDF/03_Linguaggio_macchina.pdf

[5] Codici Operativi per le istruzioni del Dlx https://www.csd.uoc.gr/~hy425/2002s/dlxmap.html