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

Sviluppo di un simulatore DLX per scopi didattici

CANDIDATO
Federico Pomponii

RELATORE
Stefano Mattoccia

Anno Accademico 2019/20

1 Introduzione

Alla base di questa tesi di laurea vi è l'obiettivo di estendere un progetto di un simulatore DLX e RISC-V realizzato precedentemente, nelle loro tesi di laurea , dagli studenti Alessandro Foglia [1] e Fabrizio Maccagnani [2].
L'obiettivo è quello di immettere nuove funzionalità nel simulatore, mantenendo invariato il duplice funzionamento tra DLX e RISC-V, in ottica di un utilizzo da parte degli studenti nei corsi universitari, nell'ambito dei Calcolatori Elettronici.
L'interfaccia finale del progetto permette di aggiungere o rimuovere spazi di memoria, configurarne le caratteristiche e definire i diversi metodi di interazione. L'apposito editor di testo permette di scrivere codice assembly e lo script di esecuzione mostra a *run-time* gli effettivi risultati delle operazioni eseguite. E' presente inoltre una documentazione che fornisce informazioni riguardo i comandi da utilizzare per il DLX o RISC-V.
All'avvio l'applicazione fornisce un codice di esempio che permette di visualizzare il funzionamento di una rete logica per l'accensione o spegnimento di un LED, è possibile inoltre modificare la rete o crearne delle nuove, personalizzando il modo in cui queste agiscono.

La tesi è articolati in 6 capitoli: nel primo capitolo viene fornita un'introduzione agli argomenti trattati, fornendo una presentazione completa di tutto ciò che avviene nel simulatore. Il secondo capitolo introduce le tecnologie e gli strumenti utilizzati nel corso dello sviluppo. Il terzo capitolo si concentra sulle nuove funzioni introdotte, analizzate nel dettaglio. Nel quarto capitolo sono trattate le fasi di sperimentazione. Nel quinto e sesto capitolo invece sono individuati possibili sviluppi futuri e sono presentate delle considerazioni finali sul progetto.

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].

Di seguito un'immagine raffigurante il set di istruzioni del RISC-V a 32 bit

32-bit RISC-V instruction formats

Format	Bit																													
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Register/register	funct7							rs2					rs1					funct3			rd					opcode						
Immediate	imm[11:0]												rs1					funct3			rd					opcode						
Upper immediate	imm[31:12]																				rd					opcode						
Store	imm[11:5]							rs2					rs1					funct3			imm[4:0]					opcode						
Branch	[12]	imm[10:5]						rs2					rs1					funct3			imm[4:1]				[11]	opcode						
Jump	[20]	imm[10:1]										[11]	imm[19:12]								rd					opcode						

- **opcode (7 bits):** Partially specifies which of the 6 types of *instruction formats*.
- **funct7, and funct3 (10 bits):** These two fields, further than the *opcode* field, specify the operation to be performed.
- **rs1 (5 bits):** Specifies, by index, the register containing first operand (i.e., source register).
- **rs2 (5 bits):** Specifies the second operand register.
- **rd (5 bits):** Specifies the destination register to which the computation result will be directed.

Figura 1.1 - Istruzioni Risc-V 32 bit

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 definisce 32 GPR a 32 bit, R0-R31. Il registro R0 ha sempre valore 0. 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 [2].

Nel IR (Instruction Register) lungo 32 bit è contenuta l'istruzione in corso di esecuzione. Le istruzioni hanno lunghezza fissa di 32 bit, di cui un campo da 6 bit per l'operatore che quindi codifica al massimo 64 istruzioni, anche se il set standard è composto da 50 istruzioni; e campi da 5 bit per selezionare ognuno dei 32 registri general purpose usati come operandi. Le istruzioni DLX sono di tre tipi, le istruzioni *R* le istruzioni *I* e le istruzioni *J* [7].

1.2.1 Operazioni DLX

Le istruzioni R lavorano esclusivamente tra registri, utilizzano un operatore e tre operandi riferiti a registri, tutto in una lunghezza di 32 bit ovviamente.
Le istruzioni I sono simili ma utilizzano due operandi riferiti a registri ed un operando immediato da 16 bit con segno, che quando è trasferito in un registro da 32 bit viene "esteso in segno".
Le istruzioni J invece sono istruzioni di salto e contengono un operatore e uno spiazzamento da 26 bit con segno che esteso e sommato all'indirizzo contenuto nel PC (Registro Program Counter) fornisce l'indirizzo di destinazione del salto [7].

1.3 Gestione interrupt

Un interrupt è un evento inatteso che interrompe il normale flusso di esecuzione del programma e causa il trasferimento del controllo ad una apposita procedura di servizio in grado di gestire opportunamente l'evento che ha originato l'interruzione. Terminata la procedura di servizio, l'esecuzione riprende secondo il flusso normale [5].
Un segnale di interrupt può essere generato via software, dall'esterno della CPU oppure da un'eccezione. Un interrupt ad esempio viene generato nel momento in cui un processo tenta di eseguire una divisione per zero; in questo caso il processo non prosegue e viene segnalato al sistema di dover gestire correttamente l'errore.

2 Strumenti utilizzati

Per lo sviluppo del progetto è stato utilizzato Angular, uno dei framework Javascript open-source più popolari, utilizzato da numerosi sviluppatori in tutto il mondo per creare applicazioni web dinamiche grazie a una serie di funzionalità e strumenti che semplificano lo sviluppo delle applicazioni stesse garantendo contemporaneamente ottimi risultati in termini di prestazioni.

La prima versione di Angular, a cui spesso si fa riferimento con il nome AngularJS (o Angular.js) fu inizialmente sviluppata nel 2009 da Miško Hevery e Adam Abrons come progetto secondario con lo scopo di semplificare il processo di sviluppo di applicazioni web. L'intenzione era quella di poter facilitare la realizzazione delle applicazioni attraverso l'uso di un'estensione del linguaggio HTML. Il nome Angular deriva semplicemente dal fatto che i tag HTML sono racchiusi da parentesi angolari. È possibile creare un'applicazione Angular usando diversi linguaggi di programmazione. È ovviamente possibile utilizzare Javascript e sfruttare le funzionalità introdotte a partire da ES2015. Tuttavia il linguaggio consigliato è Typescript [3].

TypeScript è un linguaggio di programmazione open source sviluppato da Microsoft. Si tratta di un Super-set di JavaScript che basa le sue caratteristiche su ECMAScript 6. Per essere eseguito su browser meno recenti il codice TypeScript bisogna essere compilato in Javascript.

Il linguaggio estende la sintassi di JavaScript in modo che qualunque programma scritto in JavaScript sia anche in grado di funzionare con TypeScript senza nessuna modifica. È stato progettato per lo sviluppo di grandi applicazioni ed è destinato a essere compilato in JavaScript per poter essere interpretato da qualunque web browser o app [4].

In aggiunta è stato utilizzato Angular Material, un *toolkit* che combinato con il Flex layout, facilità lo sviluppo e l'inserimento di nuovi componenti in un'applicazione web-based.

3 Nuove funzionalità

Rispetto all'ambiente precedentemente sviluppato in altre tesi di laurea, nel simulatore sono state aggiunte le seguenti funzionalità:

- Possibilità di modificare e personalizzare la rete di avvio
- Utilizzare la rete logica per l'accensione o spegnimento di un LED
- Visualizzare il valore di un determinato indirizzo in memoria

3.1 Gestione interrupt

Nell'istante in cui l'applicazione riceve un interrupt la procedura viene passata all'interrupt handler, ossia una routine deputata a gestire tale evento. Alla ricezione di un interrupt il valore del registro Interrupt Address Resolution (IAR) diventa PC + 4 mentre Program Counter (PC) è impostato a 0, a questo punto dunque il processo effettua una fetch all'indirizzo 0 ed il ritorno dalla procedura di interrupt avviene con l'istruzione RFE, che porta il valore di PC a IAR.
Considerando che anche a regime viene effettuata una lettura all'indirizzo 0, è stata introdotta una rete logica capace di diversificare la procedura di avvio e la procedura di gestione di un interruzione.
L'elemento principale della rete utilizzata è il Flip Flop-D, vale a dire un circuito sequenziale molto semplice, utilizzato per esempio come dispositivo di memoria elementare [4].
Un Flip-Flop ha in ingresso un dato D, un segnale di sincronizzazione Clock e un'uscita Q. Inoltre sono presenti due segnali asincroni di A_SET e A_RES che rispettivamente portano il valore logico dell'uscita ad 1 o 0.
In corrispondenza del fronte di salita del segnale di Clock il FF-D trasferisce l'ingresso in uscita e ve lo mantiene fin quando non cambia il suddetto ingresso [4].
Di seguito sono rappresentati lo schema ai morsetti ed il diagramma logico di un FF-D standard.

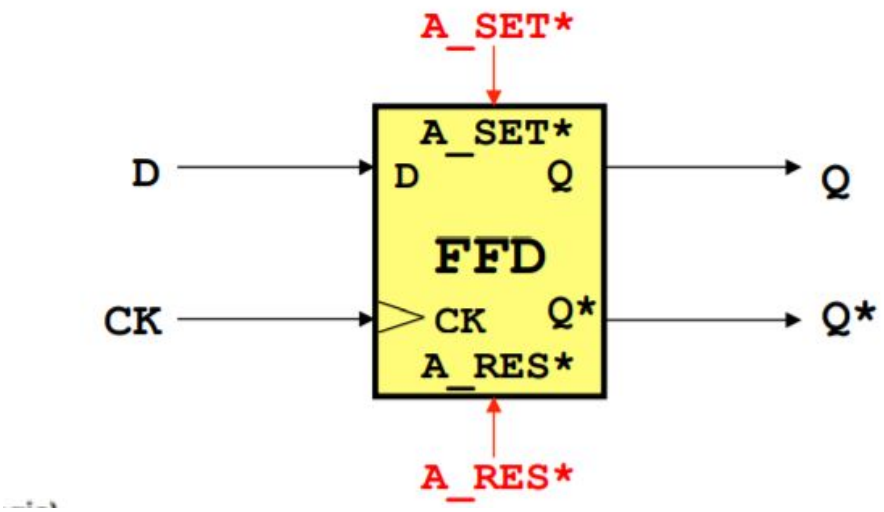

Figura 3.1 - Schema FF-D

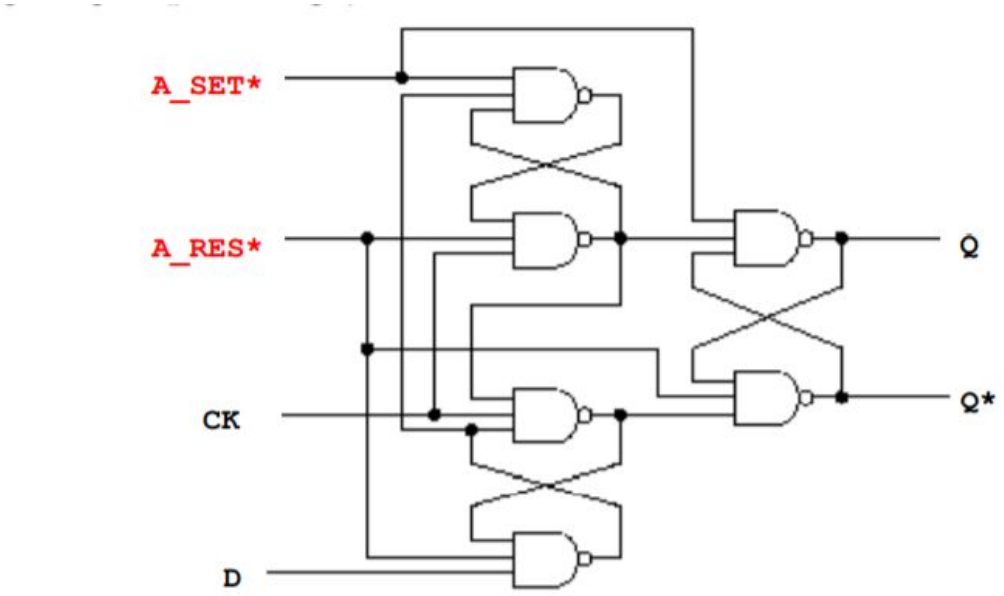

Figura 3.2 - Diagramma logico FF-D

Questa rete ha la responsabilità di impostare l'uscita del FF-D a 1 o 0, a seconda se ci troviamo in una procedura di avvio o in una routine di gestione di un interrupt.

Per settare ad 1 il valore dell'uscita, chiamata STARTUP, abbiamo collegato ad A_SET il segnale RESET, ossia un segnale che all'avvio assume il valore logico 1 prima di tornare a 0 dopo un breve lasso di tempo.

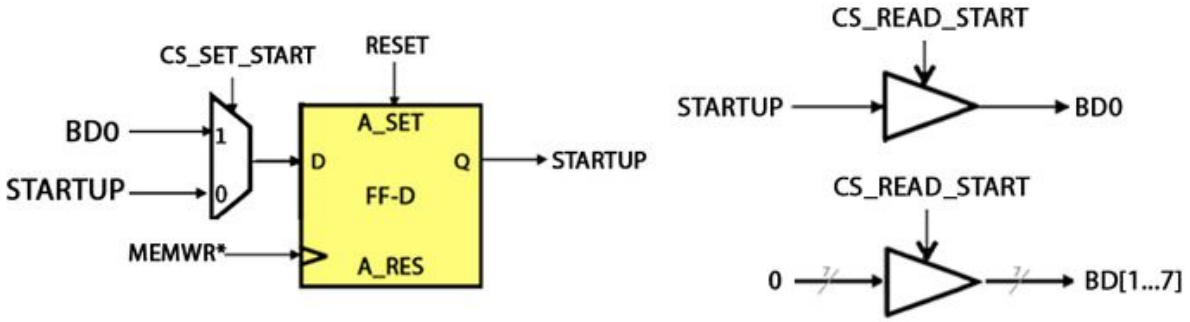

Figura 3.3 - Rete logica start

Il valore di STARTUP può essere modificato mediante opportune scrittura a CS_SET_START e può essere letto a CS_READ_START. A regime la rete verifica se il valore di STARTUP è 0, in questo caso affida la gestione all'interrupt handler, viceversa la rete imposta il valore di STARTUP a 0 ed effettua una Jump al main. Inoltre finchè il valore di STARTUP è diverso da 0 non sarà possibile generare un segnale di interrupt.

Figura 3.4 - Rete logica interrupt

Il codice (DLX) della procedura di start è il seguente

```
 0h LHI R30, 0×4000                  ; load RAM A 40000000h
 4h SW 0×0000(R30),R29               ; store R29 in 40000000h
 8h SW 0×0004(R30),R28               ; store R28 in 40000004h
 Ch LHI R29, 0XC000                  ; load address C0000000h in R29
10h LBU R28, 0×0000(R29)             ; load STARTUP value in R28
14h BEQZ R28, handler                ; if STARTUP == 0 then jump to interrupt handler
18h SB 0×0004(R29), R0               ; set startup = 0
1Ch J main                           ; jump to main
20h handler:
24h         LHI R29, 0×9000          ; load address 90000000h in R29
28h         SB 0×0004(R29), R0       ; switch led state
2Ch         LW R28, 0×0004(R30)      ; restore R28 value
30h         LW R29, 0×0000(R30)      ; restore R29 value
34h         RFE
```

Figura 3.5 - Screenshot procedura di avvio

In questo specifico caso i due Chip-Select: *CS_READ_START* e *CS_SET_START* sono mappati rispettivamente agli indirizzi *C0000000* e *C0000004*.
L'istruzione *"LBU R28, 0x0000(R29)"* legge il valore di STARTUP in R28.
Invece l'istruzione *"SB 0x0004(R29), R0"* imposta il valore di STARTUP a 0, decretando la fine della procedura di avvio.

Inoltre, per mantenere la consistenza dei dati e per far si che l'arrivo di un interrupt non interferisca con l'esecuzione del main, all'interno dell'interrupt handler i registri utilizzati vengono salvati in memoria prima di eseguire le operazioni e vengono ripristinati alla fine della procedura. In questo modo soltanto il registro utilizzato dall'handler (in questo caso specifico il registro R30) non potrà essere utilizzato nel main.

La configurazione della rete e i corrispondenti metodi di interazione con essa sono definiti nel file start.logical-network.ts.
StartLogicalNetwork estende la classe ***LogicalNetwork***, nella quale sono definiti gli elementi e i metodi di accesso comuni alle reti logiche presenti nel progetto.

3.2 Aggiunta rete logica LED

Un'altra nuova implementazione consiste nella possibilità di aggiungere una rete logica predisposta allo spegnimento o all'accensione di un LED.
Cliccando sul pulsante "+", nella sezione dei device, è possibile infatti decidere se inserire uno spazio di memoria o se inserire una rete logica per l'accensione/spegnimento di un LED.

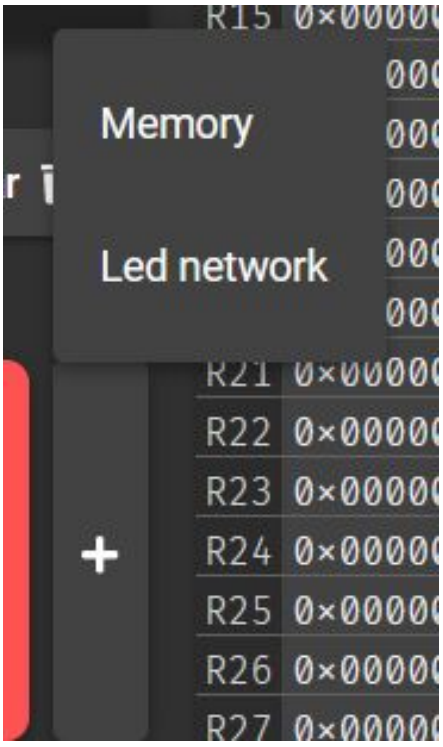

Figura 3.6 - Esempio aggiunta rete logica

Una volta inserita la rete logica, selezionandola nella sezione inferiore del simulatore, sarà possibile visualizzarla cliccando sul pulsante *"Visualizza rete logica"*.

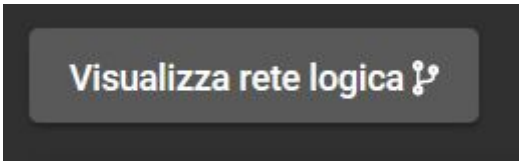

Figura 3.7 - Pulsante Visualizza rete logica

A questo punto verrà mostrata l'immagine corrispondente dello schema ai morsetti della rete e sarà inoltre possibile modificare gli indirizzi di ogni singolo

Chip-Select in modo tale da personalizzare i metodi di accesso o lettura delle informazioni necessarie.

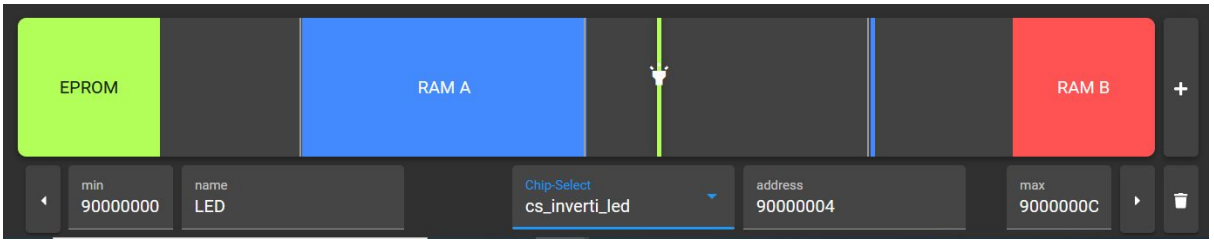

Figura 3.8 - Sezione gestione device

Infine è stata aggiunta anche la possibilità di gestire i comandi A_SET e A_RES decidendo se impostarli a 0, se collegarli al segnale RESET o se collegarli rispettivamente ai Chip-Select CS_SET e CS_RESET.
Per quanto riguarda il segnale di Clock invece è possibile decidere se utilizzare il segnale MEMWR* di fine scrittura in memoria o se utilizzare MEMRD*, ossia il segnale di fine lettura.

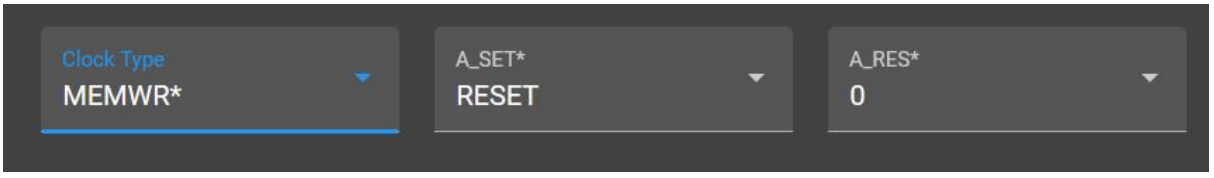

Figura 3.9 - Configurazione CLK, A_SET e A_RES**

3.2.1 Funzionamento Rete LED

Nel form per la gestione degli spazi di memoria, questo tipo di reti saranno contraddistinte da un'icona raffigurante un led in corrispondenza degli indirizzi dove la rete è collocata.
L'icona dinamicamente cambierà colore, da bianco (spento) a giallo (acceso), a seconda dello stato del Led in un determinato istante.

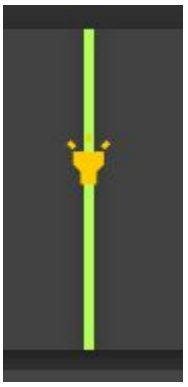

Figura 3.10 - Icona Led acceso

La rete presenta i seguenti Chip-Select:

- CS_INVERTI_LED
- CS_READ_LED
- CS_RESET
- CS_SET

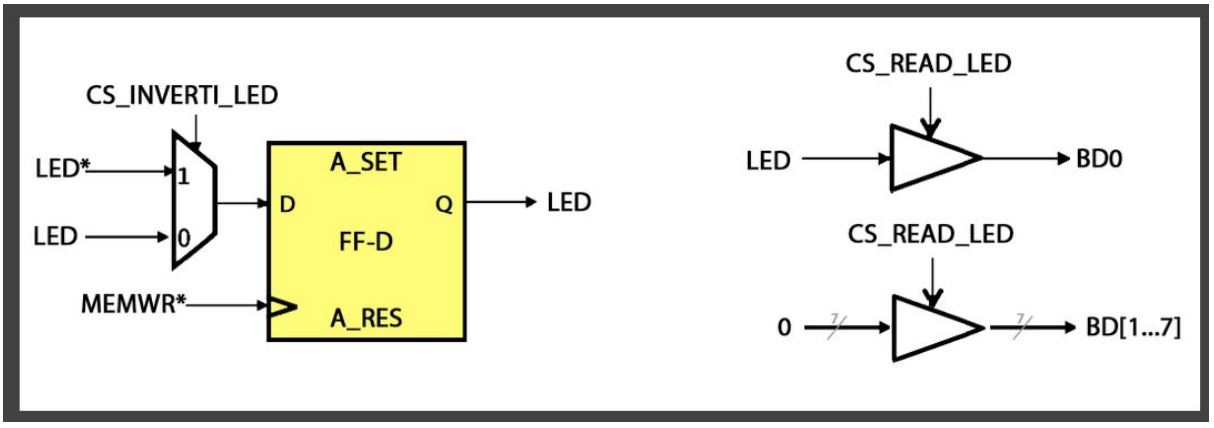

Figura 3.11 - Schema della rete logica LED

La figura 3.13 indica lo schema della rete logica. E' possibile invertire lo stato del LED, inizialmente acceso, mediante opportune scritture a CS_INVERTI_LED.

```
20h handler:
24h         LHI R29, 0×9000             ; load address 90000000h in R29
28h         SB 0×0004(R29), R0          ; switch led state
```

Figura 3.12 - Esempio scrittura a CS_INVERTI_LED

La coordinazione della rete è affidata a ***LedLogicalNetwork***, che estende la classe ***LogicalNetwork***, descritta precedentemente.

```
public a_set() {
  this.ffd_q = true;
}

public a_reset() {
  this.ffd_q = false;
}

public mux = (zero,one,sel) => {
  return sel==0 ? zero : one;
}

public tri = (input,en) => {
  return input && en;
}
```

Figura 3.13 - Funzioni definite in LogicalNetwork

I chip-select delle reti invece sono gestiti dalla classe *Device*, che espone i metodi load() e store() al componente *Memory* **,** in questo modo dopo un'operazione è possibile delegare la gestione alle singole reti logiche ed eseguire le corrette operazioni.
Le due funzioni updateCsMax() ed updateCsMin() si occupano di gestire gli indirizzi dei CS nel momento in cui viene aggiornato lo spazio di indirizzamento di una rete.

3.3 Visualizzazione valori in memoria

E’ stata aggiunta una nuova funzionalità che permette all’utente di visualizzare i valori salvati in memoria, a partire da un indirizzo scelto.

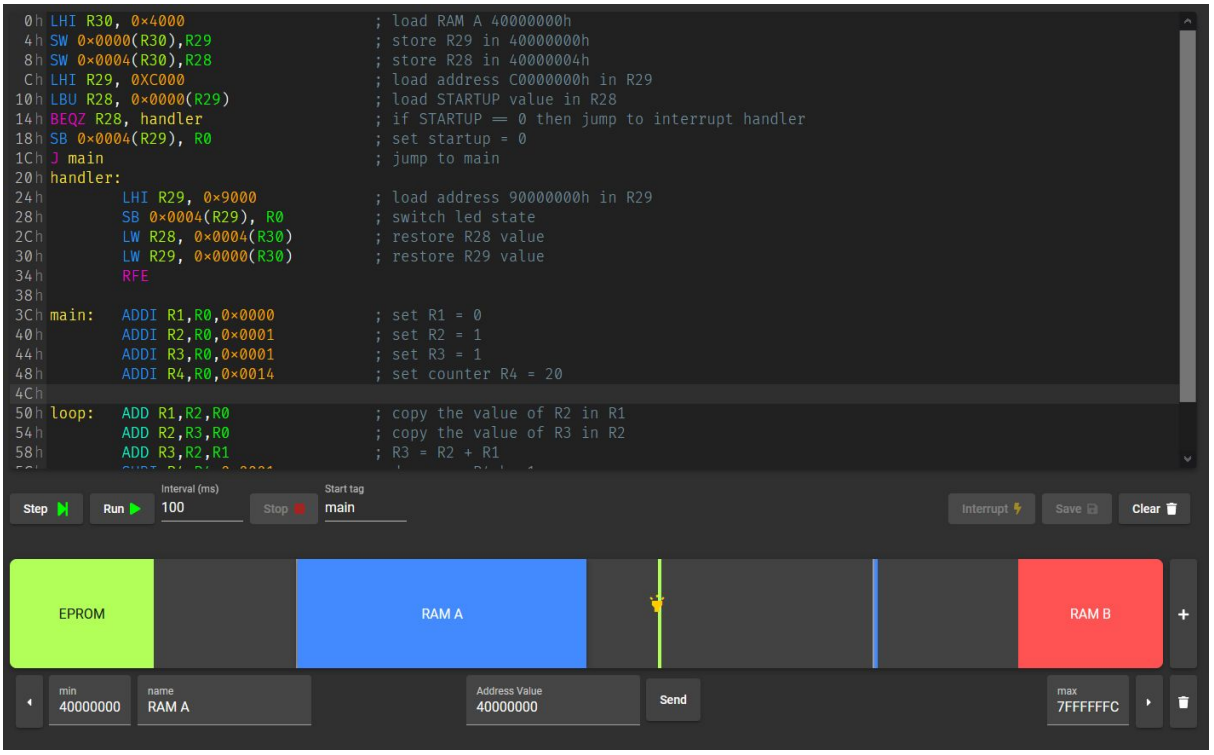

Figura 3.14 - Screenshot interfaccia

Nella sezione dei Device, nella parte inferiore dell’interfaccia, se si seleziona una memoria sarà presente un campo di input chiamato *“Address value”*. Impostando un indirizzo e cliccando sul pulsante “Send” sarà mostrata una sequenza di blocchi con il rispettivo valore a partire dall’indirizzo scelto.

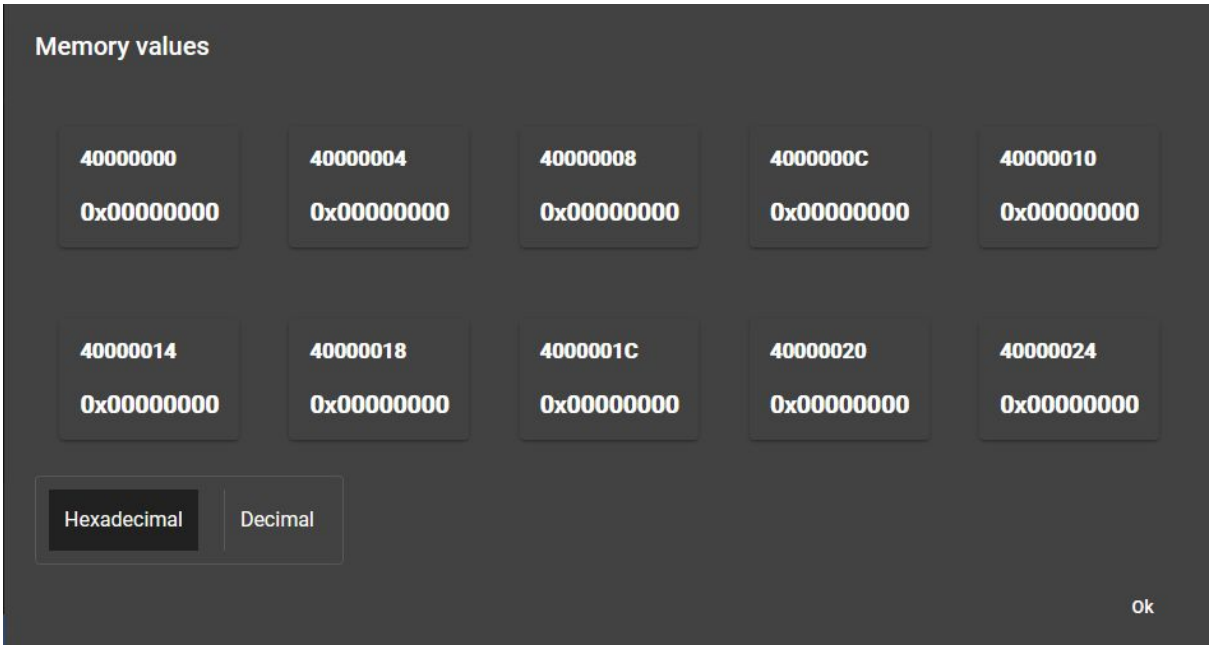

Figura 3.15 - Esempio visualizzazione valori in memoria

Nella figura è mostrato un esempio della visualizzazione degli indirizzi nella memoria "RAM A", a partire dall'indirizzo 40000000.

Per ottenere il valore effettivo degli indirizzi, a partire dall'indirizzo inserito dall'utente, viene effettuata la seguente conversione in TypeScript.

```
let iv = parseInt(addr, 16);
if (iv || iv === 0) {
  finalAddr = iv >>> 2;
}
```

Figura 3.16 - Conversione indirizzo in input

Dopo aver ottenuto i successivi 10 indirizzi, distanziati di 0004h l'uno dall'altro, convertiamo il risultato nuovamente in esadecimale per restituirlo all'utente.

```
public getAddressHex = (addr) => {
  return ((addr << 2) >>> 0).toString(16).toUpperCase().padStart(8, '0');
}
```

Figura 3.17 - Conversione indirizzo in Esadecimale

E’ possibile formattare i valori sia in esadecimale che in decimale cliccando sui pulsanti in basso a sinistra.

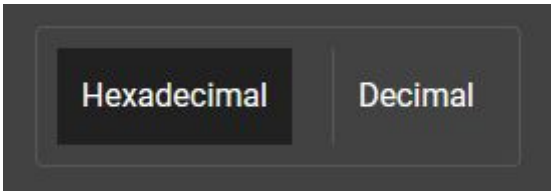

Figura 3.18 - Bottoni scelta Esadecimale e decimale

Il metodo che effettua la conversione dei valori è mostrato nella figura seguente

```
transform(n: number, type: 'dec' | 'bin' | 'hex' | 'oct', length: number = 32): any {
  switch (type) {
    case 'dec':
      return n;
    case 'bin':
      return (n >>> 0).toString(2).padStart(length, '0');
    case 'hex':
      return '0x' + (n >>> 0).toString(16).padStart(Math.ceil(length / 4), '0').toUpperCase();
    case 'oct':
      return (n >>> 0).toString(8).padStart(Math.ceil(length / 2), '0').toUpperCase();
  }
}
```

Figura 3.19 - Funzione conversione formati di rappresentazione

4 Sperimentazione

La parte centrale del progetto riguarda il testing dell'ambiente e delle nuove funzionalità precedentemente descritte. All'avvio l'applicazione avrà la seguente configurazione.

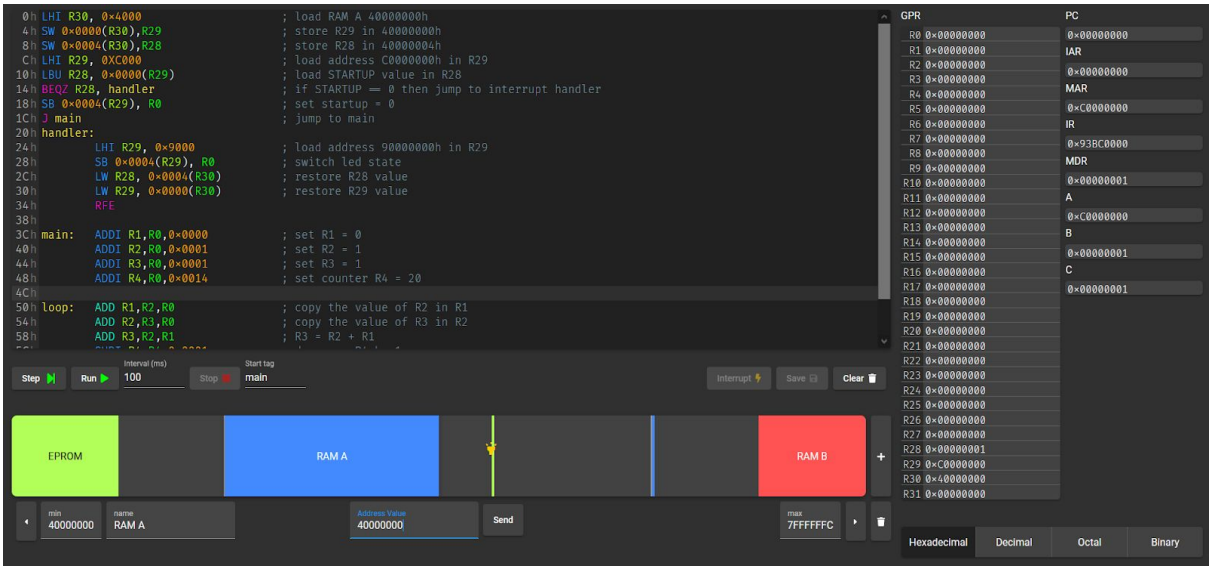

Figura 4.0 - Screenshot interfaccia all'avvio

Sono preconfigurate una EPROM, una RAM, una Rete Logica di un LED e una rete logica per l'avvio, mappate nel seguente modo:

- EPROM mappata da 00000000 a 1FFFFFFC
- RAM A mappata da 40000000 a 7FFFFFFC
- CS_INVERTI_LED mappato a 90000004
- CS_READ_START mappato a C0000000
- CS_SET_START mappato a C0000004

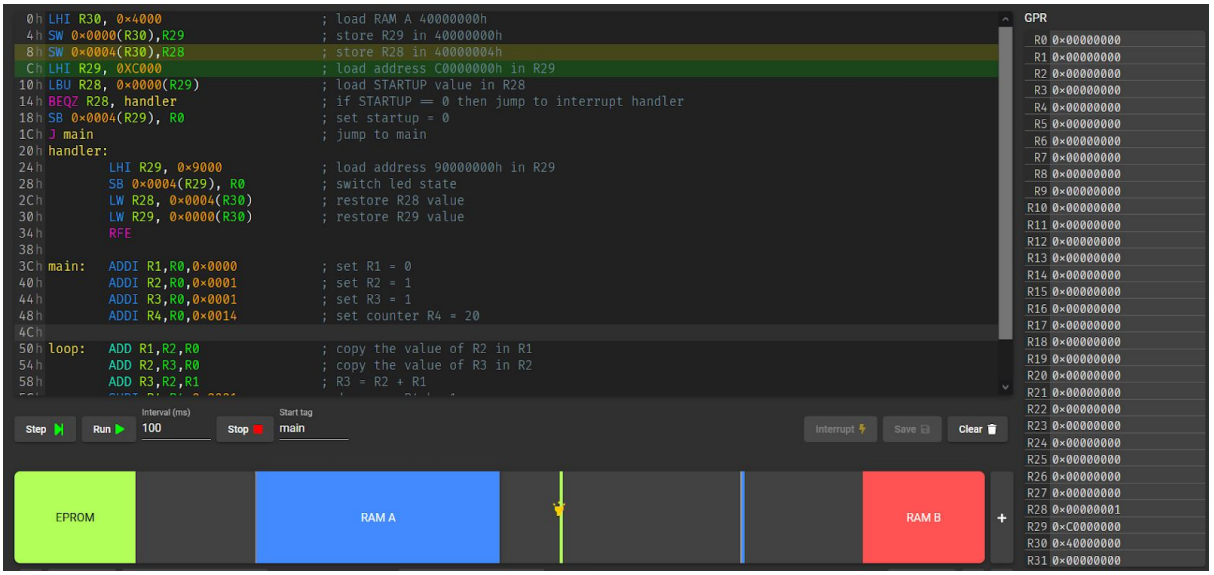

Figura 4.1 - Salvataggio dei registri in memoria

Le prime operazioni eseguite riguardano il salvataggio del valore dei registri R29 e R28 in memoria agli indirizzi 40000000h e 40000004h. Questa operazione consente di preservare la consistenza dei dati in modo che il codice in esecuzione non sia influenzato dall'arrivo o meno di un interrupt. A questo punto possiamo utilizzare i registri nel nostro handler e possiamo ripristinare il loro valore prima di ritornare dall'interrupt con l'istruzione RFE. Così facendo l'unico registro che non può essere utilizzato nel main è il registro R30, in quanto viene utilizzato nell'interrupt handler.
Nella figura 4.2 è mostrato il codice dell'handler che si occupa di ripristinare il valore dei registri R28 ed R29.

```
20h handler:
24h         LHI R29, 0×9000          ; load address 90000000h in R29
28h         SB 0×0004(R29), R0       ; switch led state
2Ch         LW R28, 0×0004(R30)      ; restore R28 value
30h         LW R29, 0×0000(R30)      ; restore R29 value
34h         RFE
```

Figura 4.2 - Ripristino dei registri R28 ed R29

```
 Ch LHI R29, 0XC000                ; load address C0000000h in R29
10h LBU R28, 0×0000(R29)           ; load STARTUP value in R28
14h BEQZ R28, handler              ; if STARTUP == 0 then jump to interrupt handler
18h SB 0×0004(R29), R0             ; set startup = 0
1Ch J main                         ; jump to main
```

Figura 4.3 - Lettura valore STARTUP

La porzione di codice mostrata nella figura 4.3 si occupa di leggere il valore di STARTUP e di decidere se affidare la gestione ad un handler o se effettuare una Jump al main.

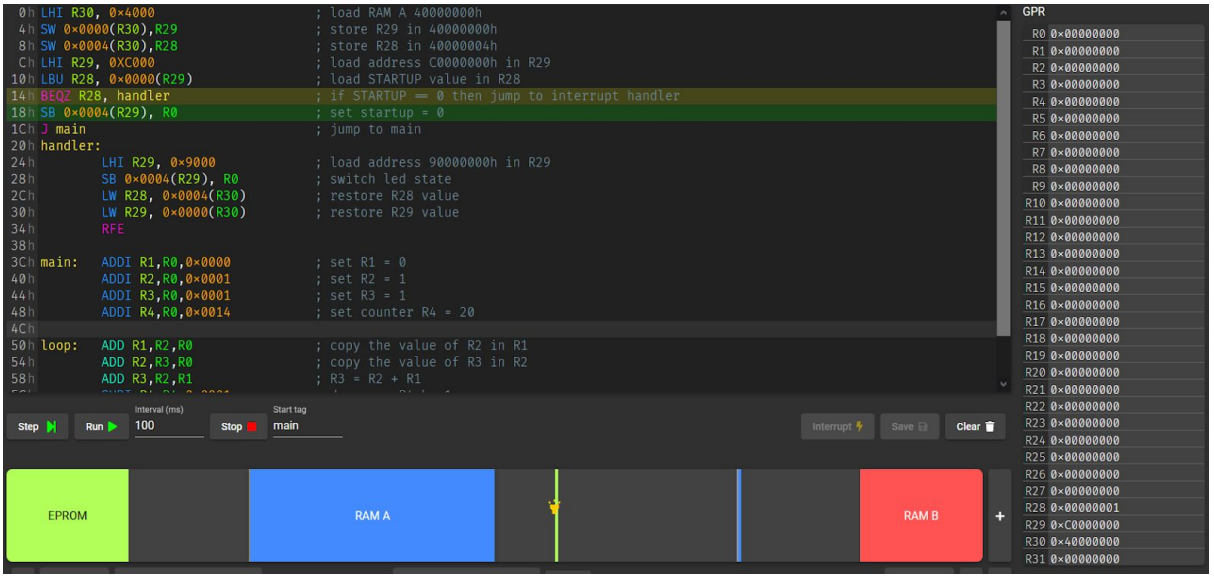

Figura 4.4 - Lettura a CS_READ_START

All'avvio il valore di R28 (risultato di una lettura a CS_READ_START) è uguale a 1, di conseguenza il programma procede scrivendo a CS_SET_START. Quindi imposta STARTUP a 0 ed infine effettua una Jump al main. Da questo punto in poi sarà possibile generare degli interrupt ed il programma saprà gestirli correttamente, delegando la procedura all'handler.

4.1 Sequenza di Fibonacci

Nel main dell'applicazione è presente un codice che simula la sequenza di Fibonacci.

```
3Ch main:   ADDI R1,R0,0×0000          ; set R1 = 0
40h         ADDI R2,R0,0×0001          ; set R2 = 1
44h         ADDI R3,R0,0×0001          ; set R3 = 1
48h         ADDI R4,R0,0×0014          ; set counter R4 = 20
4Ch
50h loop:   ADD R1,R2,R0               ; copy the value of R2 in R1
54h         ADD R2,R3,R0               ; copy the value of R3 in R2
58h         ADD R3,R2,R1               ; R3 = R2 + R1
5Ch         SUBI R4,R4,0×0001          ; decrease R4 by 1
60h         BEQZ R4,main               ; Jump to main if R4 = 0
64h         J loop                     ; Jump to loop
```

Figura 4.5 - Main per il calcolo della sequenza di Fibonacci

La sequenza di Fibonacci è una successione di numeri interi i cui primi due elementi sono 1 e 1 e ciascun altro elemento è uguale alla somma dei due precedenti. I termini della serie di Fibonacci dunque sono i seguenti:
1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,...

Nella figura 4.6 si può vedere il codice del main che rappresenta questa sequenza, effettuando una somma fino a N volte con N = 20.
Nel registro R4 è salvato il counter del ciclo, impostato a 20 con l'istruzione *"ADDI R4,R0, 0x0014"*.
I Registri R1, R2 ed R3 invece servono per effettuare le somme, salvando in R3 l'ultimo risultato della sequenza.
Una volta inizializzati i registri, il programma esegue ciclicamente la somma di R2 ed R1 salvando il risultato in R3 e contemporaneamente decrementa di 1 il valore di R4. Nel momento in cui il valore di R4 è pari a 0 il programma esce dal ciclo e torna nel main, inizializzando nuovamente i registri e cominciando da capo la sequenza.

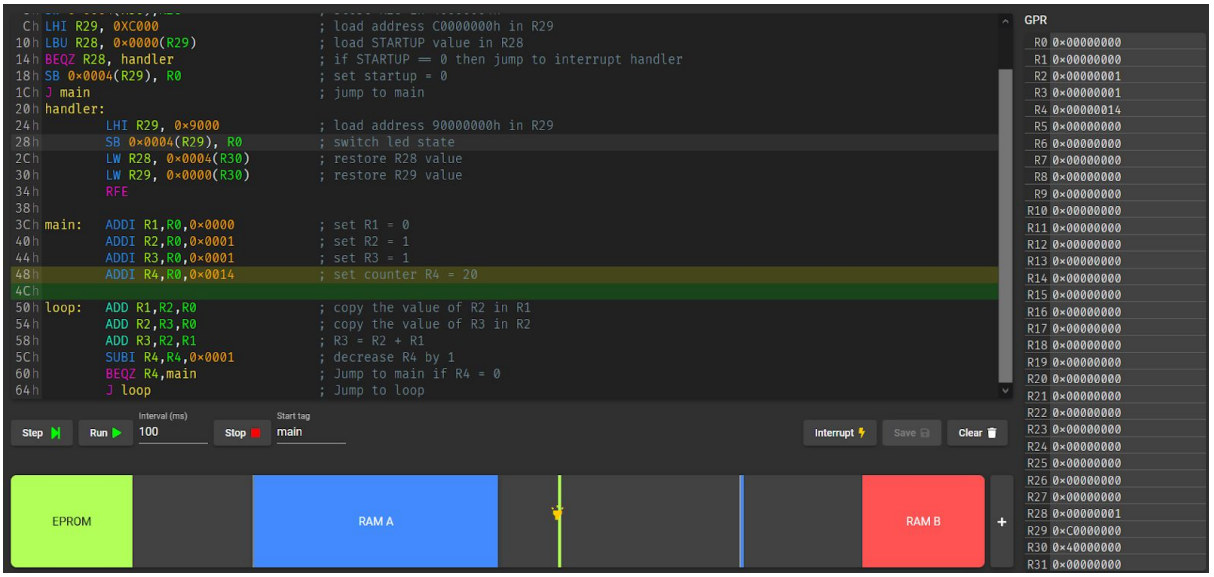

Figura 4.6 - Registri inizializzati nel main

Nella figura 4.7 si può notare come il valore dei registri R1,R2,R3 ed R4 sia cambiato dopo le operazioni nel main.
A questo punto ciclicamente il valore di R2 viene copiato in R1, il valore di R3 viene copiato in R2 ed infine R3 è il risultato della somma tra R2 e R1.

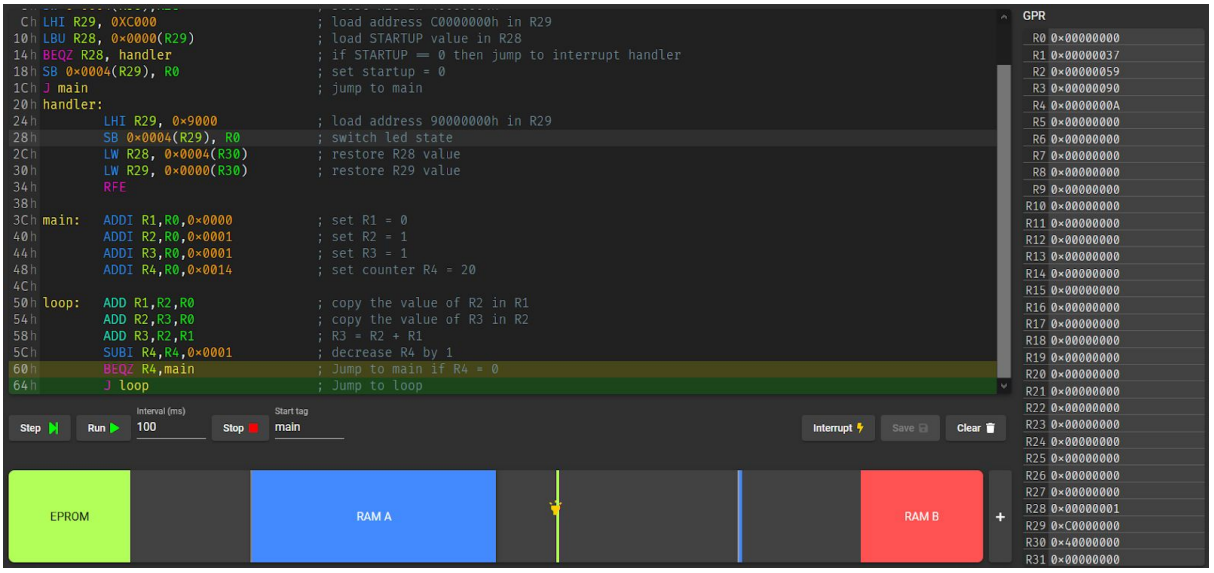

Figura 4.7 - Risultati iterazioni ciclo

Nella figura 4.8 sono mostrati i risultati alla decima iterazione del ciclo. I valori decimali di R3 ed R4 sono rispettivamente 144 e 10.

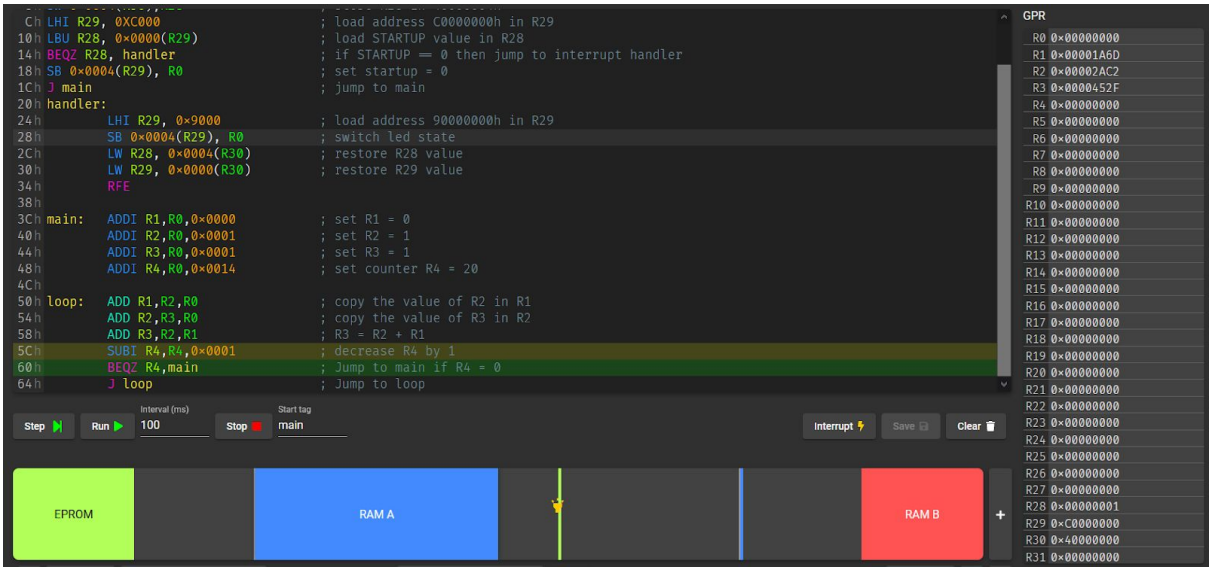

Figura 4.8 - Conclusione del calcolo

Alla ventesima iterazione il risultato è mostrato in figura 4.9.
Il valore di R4 è 0, dunque l'istruzione "*BEQZ R4, main*" fa si che si esca dal ciclo per tornare nel main. A questo punto vengono eseguite le istruzioni di partenza e la sequenza inizia da capo.

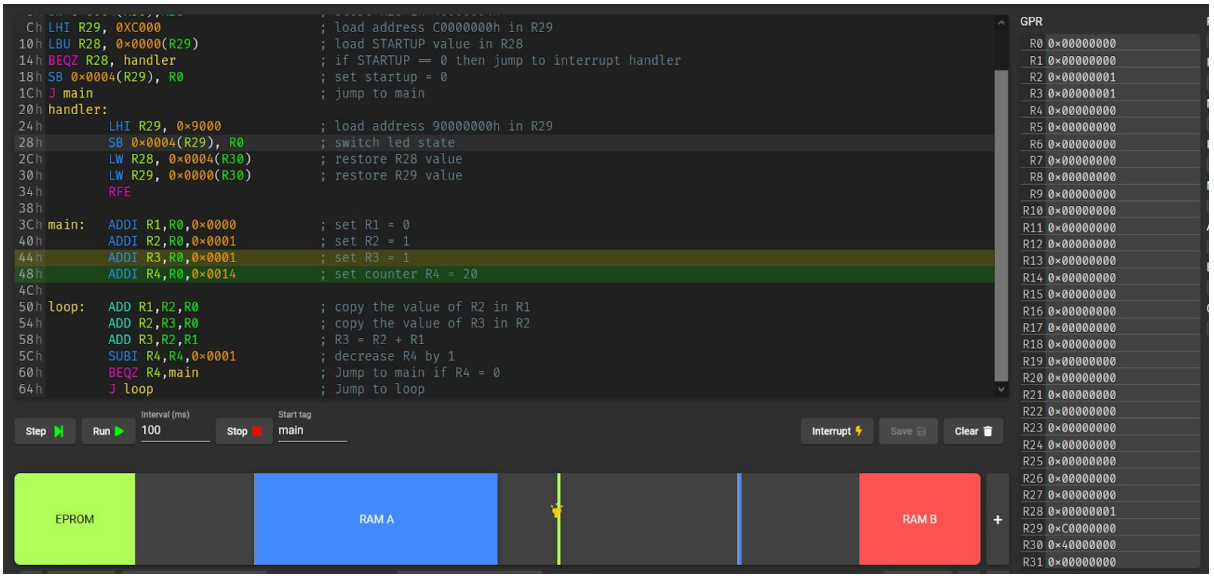

Figura 4.9 - Fine del ciclo e ritorno al main

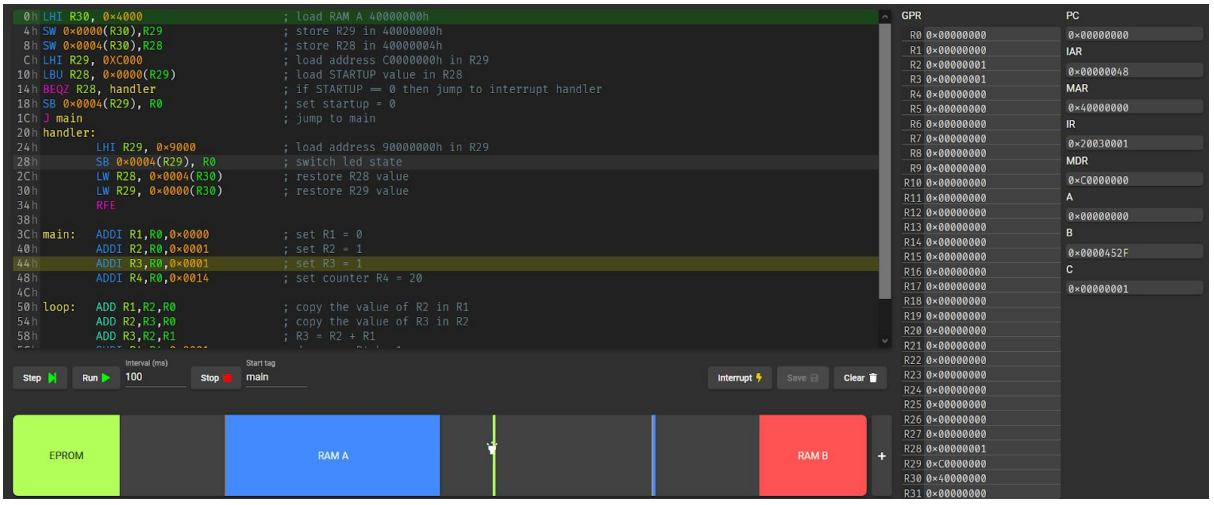

Figura 4.10 - Esempio segnale di interrupt

Nella figura 4.11 è mostrato un esempio nel caso in cui il sistema riceva un segnale di interrupt durante la procedura descritta sopra. Il valore di PC è 0x00000000, di conseguenza la prossima operazione eseguita sarà quella all'indirizzo 0h. A questo punto il valore della lettura a CS_READ_START è 0, per cui l'istruzione "*BEQZ R28, handler*" affida il controllo della procedura all'handler.

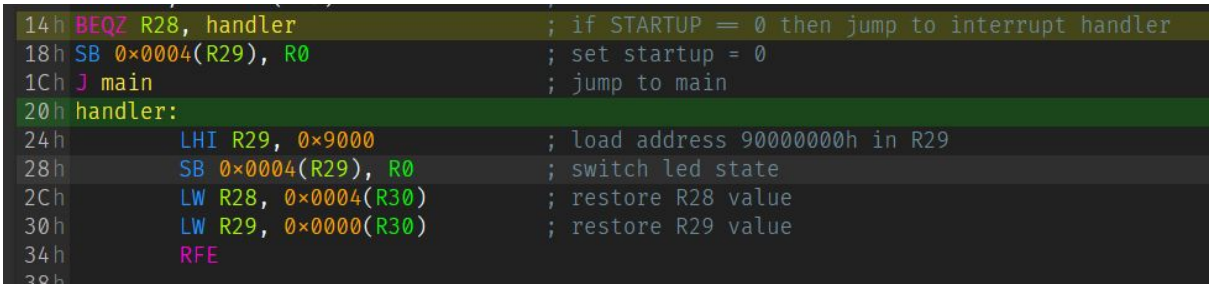

Figura 4.11 - Esempio Interrupt Handler

Il codice dell'interrupt handler, già ampiamente descritto precedentemente, inverte lo stato del led effettuando una scrittura a CS_INVERTI_LED e ripristina lo stato dei registri R28 ed R29. A questo punto il controllo passa al main che torna ad eseguire la sequenza di Fibonacci.

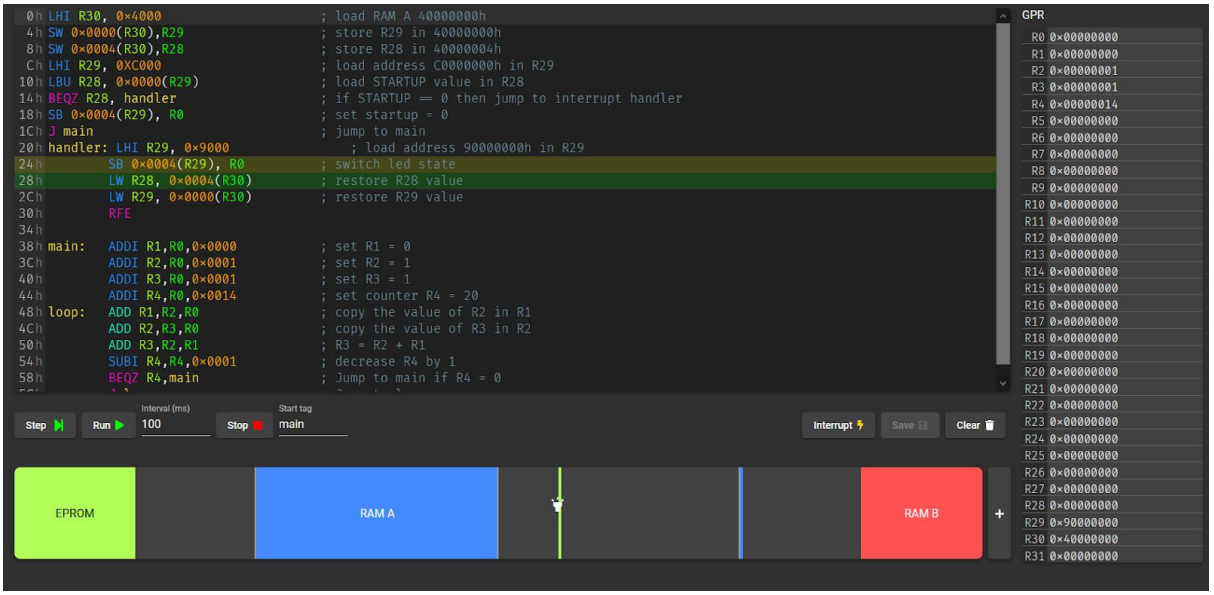

Figura 4.12 - Spegnimento LED nell'interrupt handler

Per avere la prova dell'effettivo salvataggio dei registri R28 ed R29 sulla memoria "RAM A" possiamo utilizzare il tool per la visualizzazione degli indirizzi. Infatti visualizzando gli indirizzi a partire da 40000000h il risultato sarà il seguente

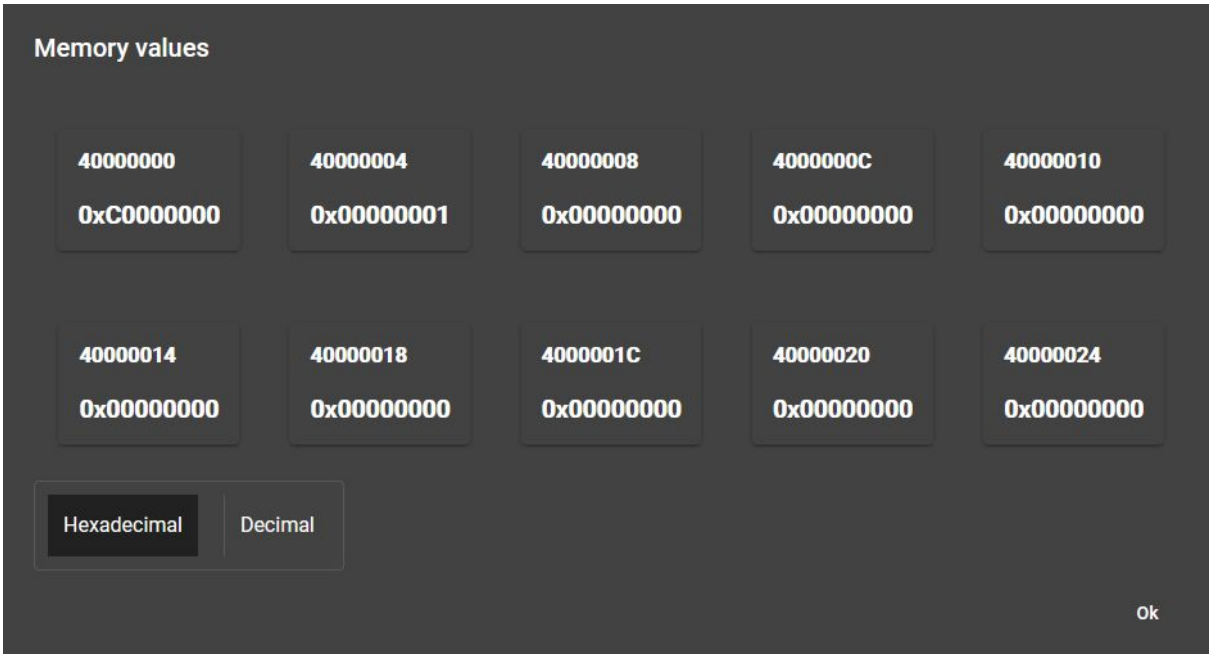

Figura 4.13 - Valori in memoria esadecimali

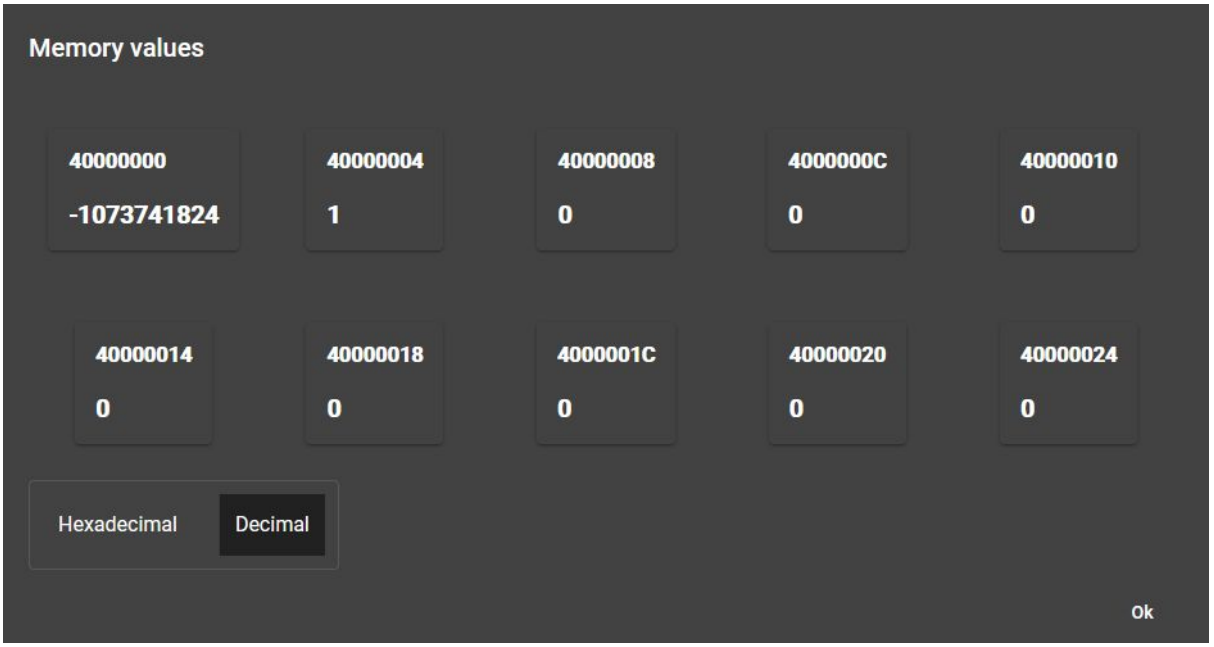

Figura 4.14 - Valori in memoria decimali

5 Sviluppi Futuri

Attualmente è possibile inserire o rimuovere delle reti logiche ma non è possibile configurare il modo in cui esse sono composte. In futuro si potrebbe pensare di affiancare all'applicazione un tool o un editor che permetta di creare delle reti logiche più complesse e articolate.

Inoltre potrebbe essere utile inserire nel progetto la gestione di periferiche I/O non troppo complesse e simularne i processi di lettura/scrittura

5.1 Possibili ottimizzazioni

Si potrebbe migliorare la gestione dei Chip Select delle reti logiche, magari inserendo un apposito service che gestisca correttamente il collegamento tra Chip-Select e le reti logiche, piuttosto che affidare il compito ai singoli componenti. Inoltre bisognerebbe configurare i Chip-Select in modo tale che sia possibile aggiungere più reti logiche dello stesso tipo con CS differenti.

Inoltre, ai fini di favorire la comprensione del modo in cui il programma agisce, sarebbe utile inserire nell'applicativo una sezione dove sia possibile caricare degli esempi di codice pre-impostati. Il tutto con un wizard e con dei popup che guidino l'utente nella comprensione di ciò che accade passo dopo passo.

6 Conclusioni

L’obiettivo del progetto è stato quello di estendere e, dove possibile, migliorare i componenti creati da Alessandro Foglia [1] e Fabrizio Maccagnani [2] nelle loro tesi di laurea, senza stravolgere la natura del loro elaborato.

Il lavoro svolto consiste nella realizzazione di componenti per la gestione e la simulazione di alcune reti logiche standard, ampiamente viste nel corso degli studi universitari. Le informazioni utili a svolgere questo lavoro sono state reperite principalmente nelle slide del corso Calcolatori Elettronici T del professore Stefano Mattoccia e nelle tesi dei colleghi.

Il risultato finale soddisfa correttamente i requisiti posti, è stata preservata la scalabilità del progetto e tutti gli sviluppi fatti sono stati pensati per agevolare l’aggiunta di nuove parti in futuro.

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] mrw.it - Introduzione ad Angular
https://www.mrw.it/javascript/introduzione-angular_12716.html

[4] Wikipedia - Flip-Flop
https://it.wikipedia.org/wiki/Flip-flop

[5] Materiale del corso di Calcolatori Elettronici T, Ingegneria Informatica, Università di Bologna, tenuto da Stefano Mattoccia. Sezione 04 Interruzioni
http://vision.deis.unibo.it/~smatt/DIDATTICA/Calcolatori_Elettronici_T/PDF/04_Interruzioni.pdf

[6] Wikipedia - Rete Logica
https://it.wikipedia.org/wiki/Rete_logica

[7] Wikipedia - DLX
https://it.wikipedia.org/wiki/DLX_(informatica)

Ringraziamenti

Giunto alla fine di questo percorso sono molto contento di avercela fatta, questi anni a Bologna sono stati stupendi ed è davvero difficile descrivere quanto io sia contento in questo giorno.

Innanzitutto ringrazio il professore Stefano Mattoccia, il mio relatore, per la disponibilità dimostrata durante tutto il periodo di stesura e per avermi dato l'opportunità di realizzare questa tesi.

Un ringraziamento speciale va ai miei genitori, che mi sono sempre stati vicini, mi hanno sempre supportato ed hanno sempre creduto in me.
Ringrazio particolarmente anche i nonni, gli zii e tutta la mia famiglia che è sempre stata al mio fianco in ogni momento.

Un grazie anche a tutte le persone incontrate a Bologna che hanno reso unici questi anni.

Infine un ringraziamento particolare va a tutti i miei amici e alla mia fidanzata che mi sono sempre stati vicini e con cui ho condiviso dei momenti fantastici, è impossibile descrivere in così poche parole quanto siete importanti per me.