

ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

DIPARTIMENTO DI
INGEGNERIA DELL'ENERGIA ELETTRICA E
DELL'INFORMAZIONE
"GUGLIELMO MARCONI"

Corso di Laurea in INGEGNERIA ELETTRONICA

TITOLO DELL'ELABORATO

VIRTUAL SHOCK

**Uno strumento Open Source di supporto alla didattica per la simulazione e la
programmazione in Logisim della micro-architettura sequenziale del DLX**

Elaborato

in

CALCOLATORI ELETTRONICI

Relatore:

Luca ROFFIA

Presentato da:

Gregorio MONARI

Correlatrice:

Elisa RIFORGIATO

Anno Accademico 2020/21

ABSTRACT

Con questo elaborato si vuole proporre una metodologia per costruire e visualizzare la struttura circuitale dell'Architettura di un Calcolatore elettronico mediante un tool grafico di simulazione, Logisim. La programmazione del sistema utilizzato avverrà mediante l'ausilio di compiler esterni sviluppati in Matlab e sarà possibile visualizzare l'output del sistema direttamente in Logisim.

L'impiego di Logisim nello studio di un Calcolatore risulta un potente strumento soprattutto per la Didattica, consentendo allo studente di avere un supporto aggiuntivo nella comprensione architeturale di un Calcolatore in tutte le sue possibili implementazioni.

Pur fornendo delle linee guida utili per lo studio di un'architettura in generale, il presente elaborato fa riferimento all'architettura della CPU presentata nel corso di Calcolatori Elettronici della Laurea in Ingegneria elettronica per l'energia e l'informazione, Università di Bologna, Campus di Cesena. In particolare, la CPU presentata nel corso è ispirata al DLX, un'architettura per microprocessori RISC sviluppata da John L. Hennessy e David A. Patterson, della quale viene presa in considerazione l'implementazione sequenziale multiciclo e di cui viene fatta l'estensione per la gestione di spazi di I/O e memoria separati.

ABSTRACT		pag. 2
INTRODUZIONE		pag. 5
CAPITOLO 1	Progetto Virtual Shock	
1.1	Prefazione	pag. 7
1.2	Cos'è Virtual Shock	pag. 9
1.3	Cosa si intende con "Macchina Programmabile"	pag. 9
1.4	Scrittura del Codice Macchina	pag. 10
1.5	Implementazione tramite microcodice	pag. 11
CAPITOLO 2	Logisim	
2.1	Introduzione a Logisim	pag. 12
2.2	Aggiungere gates	pag. 13
2.3	Collegare i componenti (<i>wires</i>)	pag. 14
2.4	Aggiungere testo	pag. 15
2.5	Testare il circuito	pag. 16
2.6	Explorer Pane e Librerie	pag. 18
2.7	Attributes Table	pag. 19
2.8	Sottocircuiti	pag. 20
2.9	Memorie	pag. 21
CAPITOLO 3	Architettura CPU Virtual Shock	
3.1	Introduzione	pag. 24
3.2	Struttura interna della CPU	pag. 25
3.3	Registri: Flip Flop in parallelo	pag. 26
3.4	Spostamenti tra registri: Flip Flop in serie	pag. 32
3.5	Registri del Data Path	pag. 34
3.6	ALU	pag. 38
3.7	Struttura logica del Data Path e codice RTL	pag. 43
3.8	Registri specializzati e concetto di ISTRUZIONE	pag. 46
3.9	Costruzione del Data Path e visualizzazione dell'esecuzione sequenziale di Microistruzioni	pag. 49
3.10	Mappatura dei segnali di controllo	pag. 56
3.11	Udc: Rete Sequenziale	pag. 59
3.12	Instruction Set Architecture del DLX	pag. 65
3.13	Implementazione Tramite Microcodice	pag. 67
3.14	Programmabilità di una Macchina di Von Neumann	pag. 88
3.15	Implementazione Ibrida dell'UDC mediante Master Control	pag. 97
3.16	Architettura di BUS, Memorie e Periferiche	pag. 103
CAPITOLO 4	Compiler	
4.1	Introduzione	pag. 106
4.2	File System	pag. 107
4.3	Compiler Assembly	pag. 108
4.4	Sintassi e Compilazione nel Compiler Assembly	pag. 109
4.5	Compiler RTL	pag. 112
4.6	Sintassi e Compilazione nel Compiler RTL	pag. 112

CAPITOLO 5	Esempio Pratico di Programmazione della CPU	
5.1	Flusso di Progetto	pag.114
5.2	Progetto dell'istruzione: microistruzioni	pag.114
5.3	Modifica Hardware Udc	pag.116
5.4	Modifica del Compilatore RTL	pag.117
5.5	Scrittura e compilazione del codice RTL	pag.119
5.6	Gestione memoria COP	pag.120
5.7	Aggiungere la nuova istruzione al Compiler ASSEMBLY	pag.121
5.8	Realizzazione del programma BLINK	pag.121
5.9	Visualizzazione dell'esecuzione del programma nella CPU	pag.125
	Conclusioni	pag.134
	Sitografia	pag.135
	Ringraziamenti	pag.135

INTRODUZIONE

Visualizzare e comprendere il funzionamento dell'Architettura di un Calcolatore è un'operazione molto complessa, specialmente per uno studente al suo primo approccio con la materia.

Non essendo possibile infatti “aprire” una CPU e osservarla dal vivo mentre lavora, bisogna ricorrere a rappresentazioni grafiche e schemi logici per schematizzarne il funzionamento.

Al fine di supportare il processo di apprendimento può essere utile un software Open Source che consenta la programmazione, la simulazione e la visualizzazione in tempo reale di questi sistemi, e al tempo stesso che abbia un'interfaccia semplice e un facile utilizzo. Un programma che rispetti le caratteristiche elencate è Logisim, strumento principale del progetto sviluppato e presentato in questa tesi: Virtual Shock. Lo sviluppatore di Logisim creò questo software proprio per insegnare concretamente ai propri studenti il funzionamento di una vera CPU.

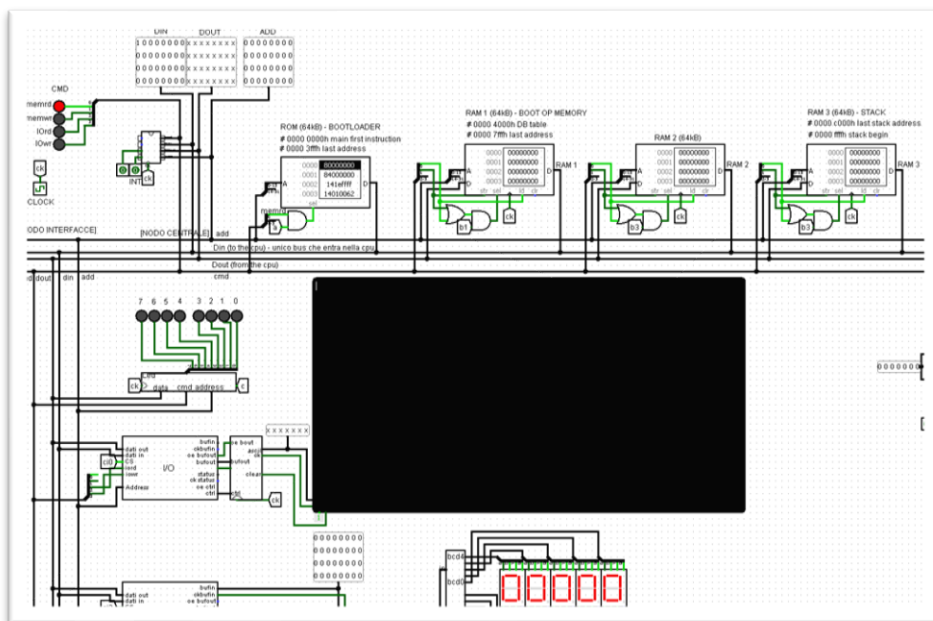


Figura 1: Schema logico della CPU VirtualShock realizzata in Logisim

Nel Capitolo 1 vengono introdotti gli elementi principali di un Calcolatore Elettronico, dell' Informazione e del Programma, in aggiunta ai punti principali affrontati dal progetto.

Il Capitolo 2 introduce il funzionamento del tool grafico Logisim.

Nel Capitolo 3 viene mostrato come usare Logisim per Costruire l'Architettura di una CPU e visualizzarne il comportamento durante i singoli fronti di salita del segnale di Clock.

Il Capitolo 4 introduce il concetto di Compiler come traduttore di linguaggio

Nel Capitolo 5 viene mostrata step-by-step la programmazione di una CPU realizzata in Logisim.

Il Capitolo 6 si focalizza sui vantaggi di affiancare lo studio teorico con un tool di visualizzazione grafica, suggerendo riflessioni per possibili future implementazioni del progetto.

CAPITOLO 1: PROGETTO VIRTUAL SHOCK

1.1-Prefazione

Con il termine Calcolatore Elettronico si intende una macchina automatica in grado di elaborare informazioni in ingresso mediante calcoli matematici e produrre un flusso di informazioni in uscita dipendente dalla programmazione dello sviluppatore. Una macchina automatica e programmabile viene anche detta elaboratore o computer. Laptop, smartphone e Xbox sono esempi anch'essi di calcolatori elettronici dotati di un input, come la tastiera o il joystick, e un output, come il desktop o lo schermo del televisore. La vasta diffusione di questi sistemi deriva proprio dal fatto che il funzionamento della macchina dipende principalmente dal programma o software memorizzato in essa.

A differenza dei calcolatori meccanici, le operazioni elaborate da un calcolatore elettronico vengono eseguite mediante componenti elettronici di tipo digitale, i transistor. Il comportamento elettronico di questi componenti è stato analizzato ampiamente nel corso di Elettronica Digitale del professor Tartagni, nel caso di Calcolatori però risulta più semplice analizzarne il comportamento secondo la Logica dell'informazione: collegando tra loro molteplici transistor è possibile realizzare strutture più complesse dette *porte logiche*, come mostrato in *figura 4*. Un sistema realizzato mediante l'uso di componenti logici viene anche detto *rete logica* e tali reti una volta progettate possono essere stampate su dei fogli di silicio detti chips o microchips.

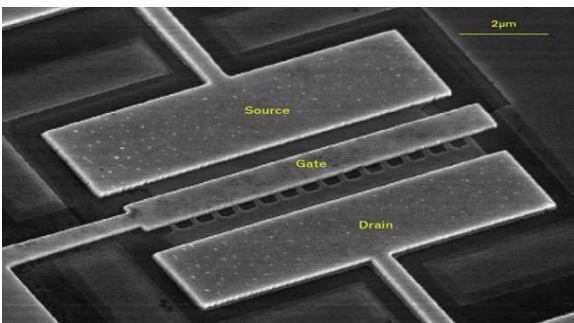


Figura 2: Fotografia zoomata di un transistor Mosfet (www.student-circuit.com)

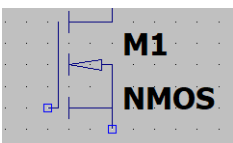


Figura 3: Simbolo circuitale del Mosfet in LTSpice

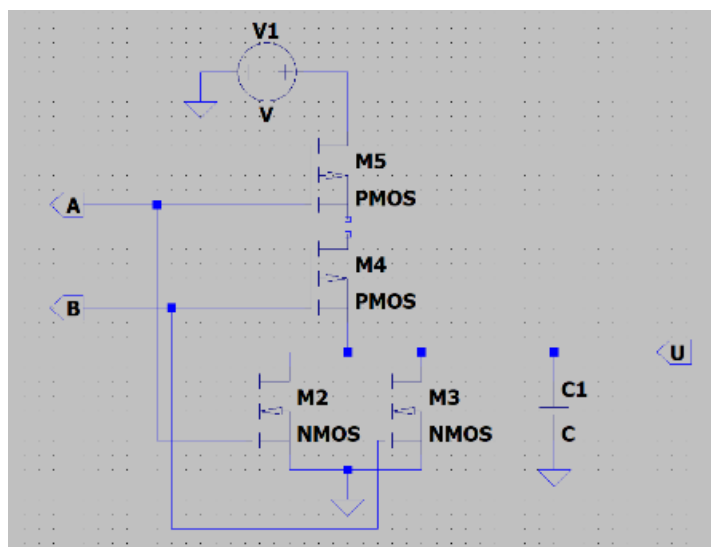


Figura 4: Schema circuitale di un Nand Gate in LTSpice

La componente principale di un calcolatore elettronico è la sua CPU, ovvero Central Processing Unit: essa è l'unità di elaborazione dove sono effettuati tutti i calcoli matematici, è quindi considerato il cervello del calcolatore.



Figura 5: Esempio di CPU Intel Core I7 (www.amazon.com)

I componenti digitali, comprese le porte logiche, seguono le regole dell'elettronica digitale, i transistor infatti si comportano come delle valvole che possono avere solamente due stati: aperto o chiuso. Un sistema simile implementa un tipo di logica detta *binaria* e la sua solidità è il motivo per il quale è stata scelta per veicolare l'informazione nei calcolatori, attraverso un codice per l'appunto chiamato *codice binario*. Concettualmente, come spiegato nell'introduzione delle slides del Modulo 2 di Calcolatori, un'informazione è una scelta tra più opzioni e l'unità più piccola di informazione presente in un sistema digitale è il bit. Il bit rappresenta gli stati dei transistor in quanto il valore che può assumere è: zero o uno. Stringhe ordinate di questi bit permettono la rappresentazione di numeri decimali e quindi l'implementazione di operazioni aritmetiche tra numeri.

La struttura dei calcolatori elettronici è particolarmente densa ed intricata, in generale ci si riferisce ad essa con il termine Architettura del Calcolatore. Per dare un'idea di grandezza, i moderni microchip contano miliardi di transistor e gates logici interconnessi tra loro.

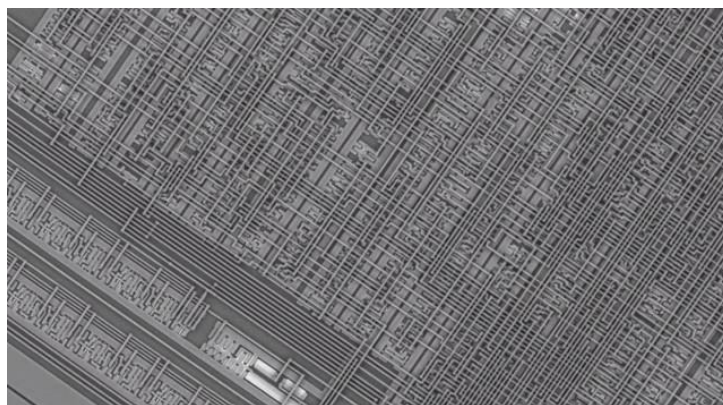


Figura 6: Fotografia zoomata della struttura di un microchip (www.extremetech.com)

Virtual Shock è un viaggio all'interno dell'Architettura di un Calcolatore, con un focus particolare nella realizzazione di una intera CPU, reso possibile proprio da questo programma di visualizzazione creato all'inizio degli anni 2000.

1.2 - Cos'è Virtual Shock

Mediante Logisim è possibile simulare non solo reti logiche combinatorie e sequenziali ma anche reti molto complesse come un'intera CPU. Il progetto Virtual Shock è volto alla simulazione e visualizzazione di tale sistema, accompagnando il lettore step-by-step nella costruzione dell'architettura e nella programmazione di una CPU Virtual Shock a 32Bit funzionante. Il primo passo è comprendere l'Hardware della macchina, ovvero lo schema della disposizione e dei collegamenti dei singoli gates logici, per poi analizzare il Software, ovvero il codice che consente di definirne il funzionamento finale permettendo di avere un calcolatore operativo.

1.3 – Cosa si intende con Macchina Programmabile

Il funzionamento di un calcolatore elettronico non viene definito solamente dalla sua architettura. L'enorme diffusione di tali sistemi, infatti, è stata resa possibile proprio grazie alla loro elevata flessibilità: è il software usato dal programmatore che determina le operazioni eseguite dalla CPU per ottenere in uscita l'output desiderato. Tale software prende il nome di "linguaggio Assembly" e viene caricato nella Memoria dell'elaboratore. Senza una memoria con all'interno un programma sviluppato, una CPU rimane un circuito inerte, come un cervello senza informazioni, e dal nulla, ovvero da una memoria vuota, non può scaturire altro che nulla. L'architettura completa di un elaboratore, mostrata in *figura 7*, conta quindi 3 blocchi essenziali: CPU, memoria e periferiche di Input/Output (come ad esempio tastiera, desktop, joystick, ma anche LEDs).

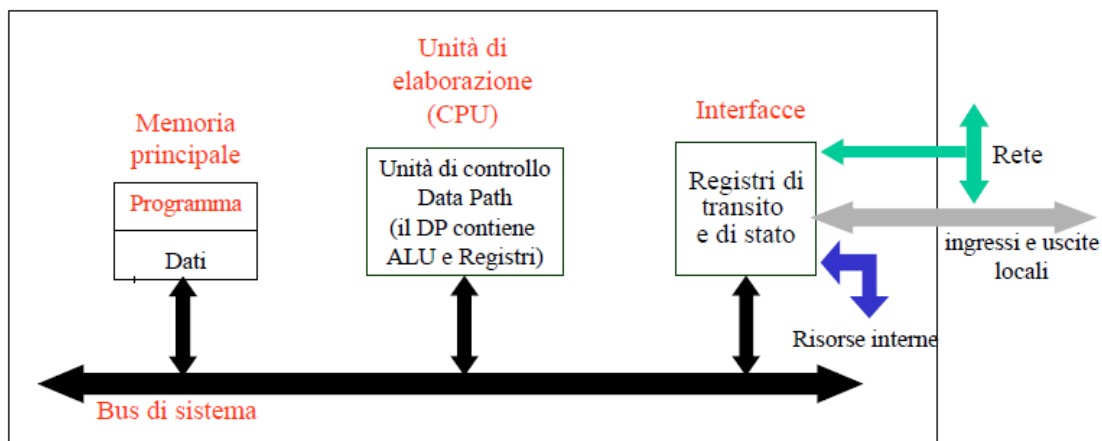


Figura 7: Schema a blocchi dell'Architettura di un Calcolatore (tratta dalle Slides del corso di Calcolatori Elettronici – A.A 2019)

Un sistema in grado di eseguire una serie di istruzioni in maniera automatica e programmabile prende il nome di Macchina di Von Neumann. La programmazione di un calcolatore può essere quindi divisa in tre fasi, come mostrato in *figura 8*:

- Scrittura del codice Assembly
- Caricamento del codice nell'elaboratore
- Simulazione del comportamento del sistema e lettura degli output



Figura 8: Flusso della Programmazione di una CPU

Studiare l'architettura di una CPU non è quindi sufficiente per comprenderne il funzionamento, e Virtual Shock non sarebbe completo senza la possibilità di programmare l'architettura appena costruita.

1.4 – Scrittura del Codice Macchina

Tra i componenti hardware messi a disposizione da Logisim sono presenti le memorie RAM e ROM. Dal punto di vista Hardware sono matrici bidimensionali di celle, contenenti degli zeri e degli uni. Un programmatore che volesse scrivere del codice per poi caricarlo su tali memorie, dovrebbe eseguire un processo lungo e tedioso di scrittura del codice Assembly in linguaggio *human readable*, in aggiunta poi alla traduzione di ogni singola istruzione in una stringa di uni e zeri, detto *Codice Macchina*, ed infine dovrebbe caricare tutto il codice in memoria e far partire la simulazione. Viene mostrato un esempio in *figura 9*, dove a sinistra è presente codice Assembly, mentre a destra viene mostrato il contenuto delle celle di una ROM in codice Esadecimale.

addi r1 r0 1h	14010001
beqz r0 2h	40000002
addi r1 r0 b	14010062
out ttybufout(r0) r1	9c200001
addi r1 r0 o	1401006f
out ttybufout(r0) r1	9c200001

Figura 9: Codice Assembly e Macchina a confronto

Sebbene questo processo sia fattibile per una o due istruzioni, con l'aumentare della complessità del codice diventa praticamente impossibile, e soprattutto inutile ai fini dell'apprendimento tradurre tutto il codice a mano.

A tal scopo risulta necessario sviluppare un algoritmo che consenta di effettuare in modo automatico le operazioni di traduzione da codice Assembly a codice Macchina. Tali algoritmi prendono il nome di Compilatori e sono tra gli strumenti più importanti per gli sviluppatori di software. In particolare, il funzionamento del compilatore Assembly per Architettura Virtual Shock verrà spiegato nel dettaglio nel capitolo 5, dopo aver introdotto l'Architettura della CPU.

1.5 – Implementazione tramite Microcodice

Come anticipato la CPU si occupa anche della parte di automatizzazione della Macchina Di Von Neumann. Sono diversi i modi in cui si può ottenere questo comportamento. Il primo tra questi consiste nel costruire una rete sequenziale molto complessa che gestisca l'interpretazione del Codice Macchina in operazioni matematiche. Risulta immediato notare come tale soluzione non sia molto pratica: per aggiungere una nuova istruzione sarebbe necessario modificare l'Hardware della CPU, senza contare il lungo processo per progettare reti sequenziali di tali dimensioni. Per dare un ordine di idee della complessità di tale rete, il calcolo delle tabelle di Karnaugh per tutta la rete sequenziale richiederebbe centinaia di ore, supponendo di mantenere un ritmo costante come una macchina.

Nello sviluppo dell'elaborato è stata suggerita una soluzione alternativa e qualitativamente migliore dal professor Testoni dell'Università di Bologna. È possibile, infatti, creare un anello mancante tra Hardware e Software, che riesca a gestire componenti hardware, ma che funzioni come una sorta di software. Tale implementazione viene chiamata Microcodice, detto anche *Firmware*, un ponte tra Hardware e Software. Il Microcodice affiancato a una rete sequenziale molto semplificata riesce quindi a ottenere lo stesso risultato di una rete sequenziale estremamente complessa. Inoltre, il Microcodice riesce a mantenere la stessa flessibilità che ha l'Assembly per la CPU: per aggiungere una nuova istruzione non serve modificare la rete sequenziale e di conseguenza l'Hardware, ma è sufficiente modificare il microcodice che verrà poi caricato sulla ROM.

Il processo di sviluppo del Microcodice è molto simile a quello del Codice Macchina: la parte di linguaggio più *human readable* viene chiamato *linguaggio RTL*, che andrà tradotto in uni e zeri nel Microcodice. Similmente al caso Assembly, è stato sviluppato un compilatore che possa aiutare in questo processo, che verrà descritto nel dettaglio nel capitolo 4.

CAPITOLO 2: Logisim

2.1-Introduzione a Logisim

Logisim è un tool grafico open source per il design e la simulazione di circuiti logici, sviluppato nel 2001 da Carl Burch , quando ancora rivestiva la carica di professore di Computer Science all'Hendrix College. Logisim presenta un'interfaccia semplice e la possibilità di simulare circuiti mentre vengono costruiti. Inoltre, è uno strumento:

- Sufficientemente intuitivo per imparare i concetti di base associati ai circuiti logici;
- Abbastanza potente per simulare intere CPU.



L'interfaccia Logisim è composta da cinque aree principali, evidenziate in figura 10:

1. Menu Bar, barra in alto a sinistra con le voci file, edit, project, simulate, window, help;
2. Tool Bar, sotto la Menu Bar, comprende le icone dei gates e il simbolo del cursore;
3. Explorer Pane, l'elenco di cartelle sulla sinistra sotto la Tool Bar;
4. Attributes Table, riquadro sito sotto l'Explorer Pane;
5. Canvas, la grande area bianca che occupa la parte destra dello schermo;

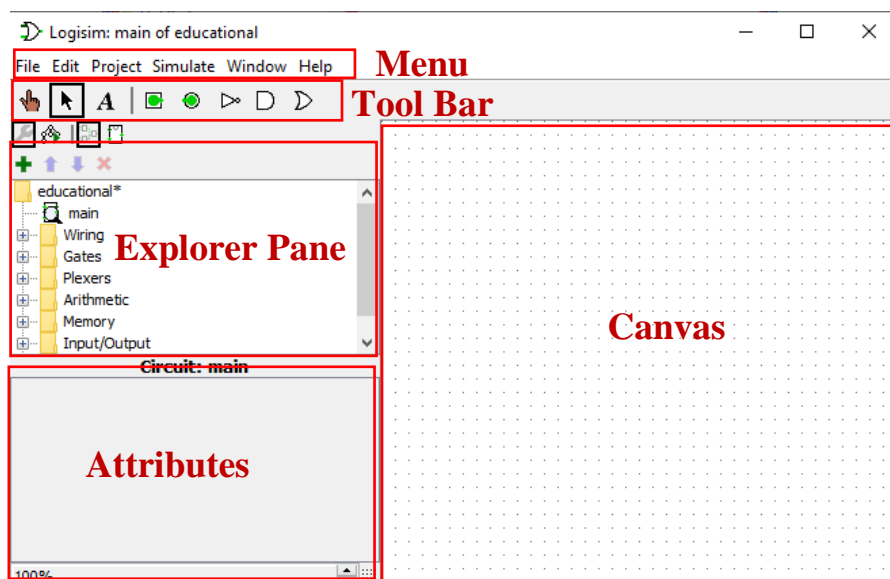


Figura 10: Interfaccia mostrata all'avvio del programma Logisim

La Menu Bar permette di gestire i files di progetto e la simulazione del circuito, mentre la Tool Bar permette di modificare e aggiungere componenti al Canvas (la *tela bianca* sulla quale vengono costruiti gli schemi logici dei circuiti). Questa tela è il fulcro del software, ed essendo realizzata mediante una interfaccia di tipo grafico il suo funzionamento verrà presentato mediante la costruzione e simulazione di una porta logica: lo XOR gate. L'Explorer Pane e l'Attributes Table verranno introdotti alla conclusione di questo capitolo.

2.2-Aggiungere gates

Nella costruzione di un circuito si procede prima aggiungendo i gates che fanno da struttura portante, connettendoli successivamente con dei fili (*wires*).

Nel caso di un circuito XOR ad esempio bisogna selezionare i componenti AND, NOT e OR dalla toolbar (ultime tre icone) e poi cliccare nel Canvas per posizionarli, come mostrato in *figura 11*:

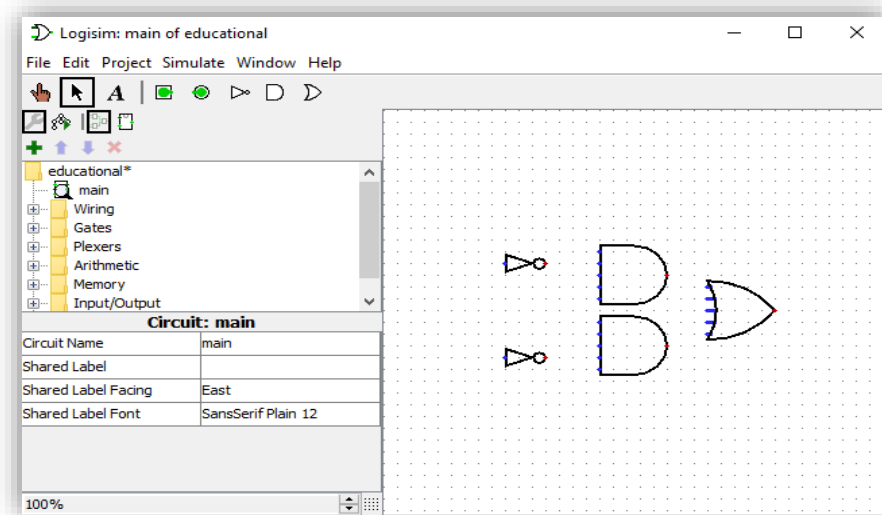


Figura 11: Simboli circuitali in Logisim delle porte Logiche NOT, AND, OR

I gates di default vengono posizionati con 5 pin in entrata, in questo caso negli AND e negli OR ne sono sufficienti 2 per la loro logica.

Ora bisogna aggiungere i due pin di input del circuito e l'output, come mostrato in *figura 12*:

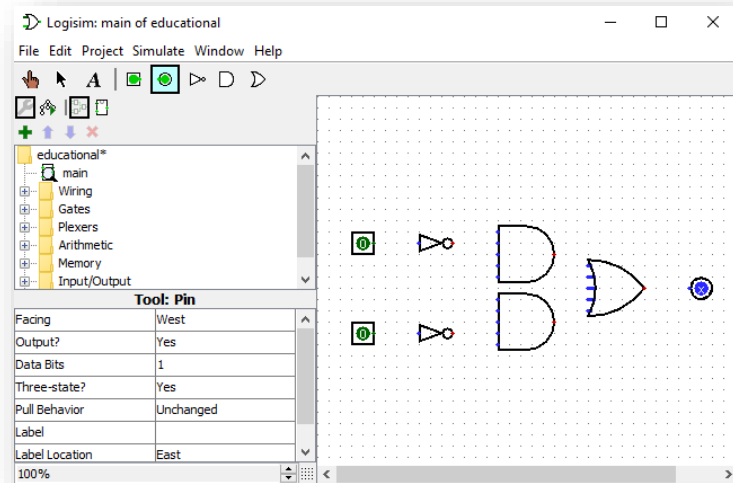


Figura 12: Aggiunta di pin di I/O

I pin posizionati rappresentano la sorgente (o uscita) di un bit di informazione, di default possono quindi assumere i valori 0 o 1.

2.3-Collegare i componenti

Una volta posizionati tutti i componenti necessari, si può procedere ad aggiungere le piste che li collegheranno.

Dopo aver premuto l'icona del cursore nella toolbar (seconda icona da sinistra), per creare un collegamento è sufficiente muovere il cursore su un pin di ingresso o uscita per poi cliccare e trascinarlo sul secondo pin che si vuole collegare. Il risultato viene mostrato in *figura 13*.

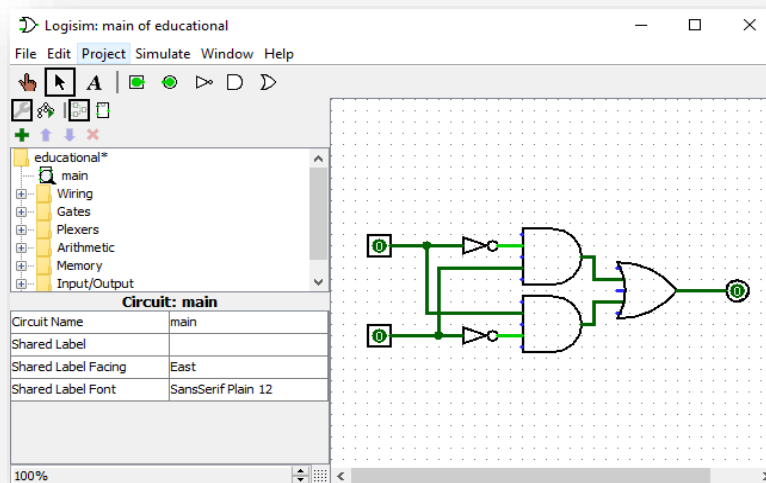


Figura 13: Aggiunta delle piste di collegamento

Una volta collegati i pin di ingresso e uscita, il simulatore calcola istantaneamente lo stato della linea e lo visualizza attraverso due componenti visive:

- il colore della linea: illuminata se lo stato è HIGH (1), spenta per lo stato LOW(0)
- lo stato dei bit in I/O: se un bit è collegato a una pista con valore HIGH, esso prenderà lo stesso valore e lo mostrerà nella sua interfaccia. Stesso discorso nel caso di uno zero.

2.4-Aggiungere testo

Ai fini di leggibilità del sistema può risultare comodo aggiungere dei commenti direttamente sul Canvas: selezionando dalla Toolbar l'icona con la lettera A (terza da sinistra) e cliccando nell'area di editing, apparirà un box nella quale è possibile scrivere del testo. Premendo invio il commento viene "fissato" nel Canvas, come mostrato in *figura 14*.

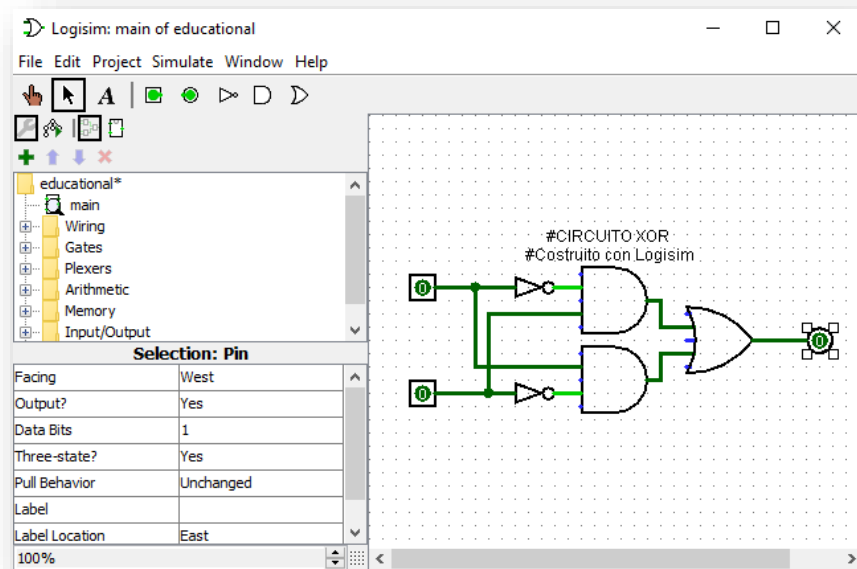


Figura 14: Fissaggio dei commenti direttamente nel Canvas

Per dare un'etichetta ad un pin è consigliato fare doppio click sul pin stesso come in *figura 15*, in tal modo l'etichetta si sposterà insieme al pin nel caso venga spostato nel Canvas.

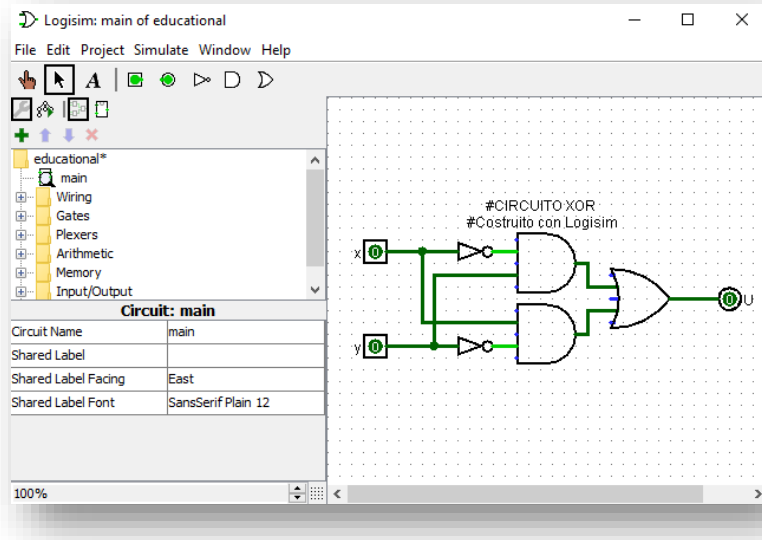


Figura 15: Aggiunta di Etichette ai pin di I/O

2.5-Testare il circuito

L'ultimo passo nella progettazione di un circuito è testare il suo funzionamento confrontando gli output della rete con quelli attesi. L'espressione logica della funzione XOR può essere rappresentata usando la Tabella di Verità, come mostrato in figura 16.

La tabella presenta due colonne per gli input e una per l'output. Una funzione combinatoria presenta 2^n uscite possibili, dove n rappresenta il numero di ingressi, quindi in questo caso 2^2 ovvero 4 righe.

X	Y	U= X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0

Figura 16: Tabella di verità di uno XOR gate

Come è già stato verificato nei precedenti paragrafi, quando i due input sono a 0 anche l'uscita è a 0. È possibile cambiare il valore dei bit premendo sull'icona della Toolbar a forma di mano (la prima a sinistra) e cliccando sul bit si vogliono invertire.

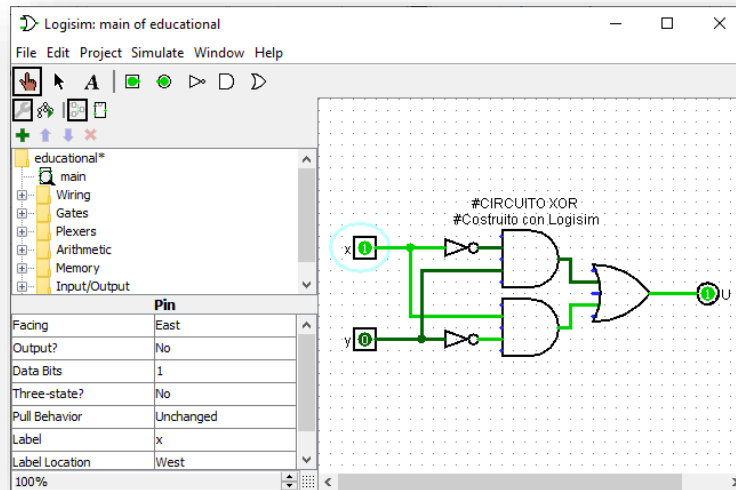


Figura 17: Uscita dello XOR nel caso di due input diversi

Come si può notare dalla *figura 17*, modificando uno dei due bit l'uscita si accende. L'ultimo caso da analizzare è quello dove entrambi i bit sono a 1, visualizzato in *figura 18*.

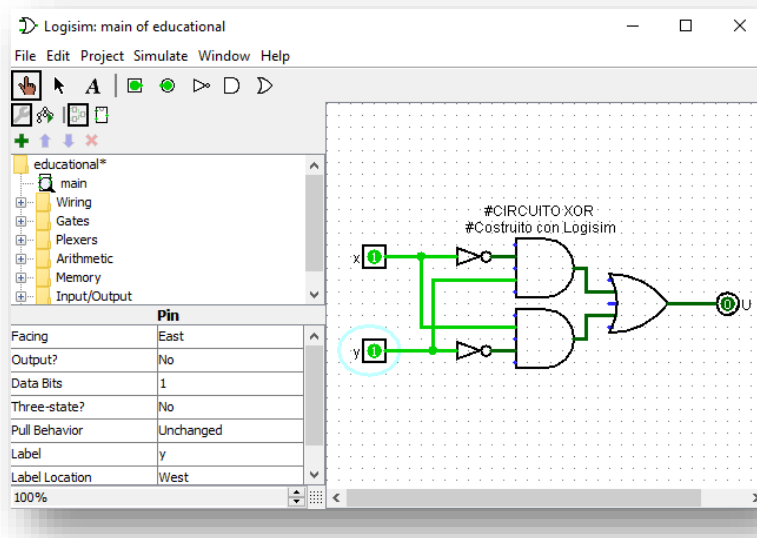


Figura 18: Uscita dello XOR nel caso di due input uguali

Tutte le casistiche corrispondono a quelle mostrate nella tabella di verità: è il comportamento atteso di uno XOR gate. Il circuito è ora funzionante e utilizzabile appieno.

2.6-Explorer Pane e Librerie

I tre gates usati nel precedente circuito non sono gli unici componenti disponibili nel simulatore. Nell'Explorer Pane, evidenziato in rosso in *figura 19*, è possibile trovare una serie di componenti organizzati in sottocartelle, dette *Libraries*.

Quando viene creato un nuovo progetto, vengono aggiunte di default diverse librerie, come:

- *Wiring*: componenti usati per interagire con le piste di connessione;
- *Gates*: componenti che eseguono semplici funzioni logiche;
- *Memory*: componenti dotati di memoria, come flip flops e RAMs.

Aprendo la libreria *gates* e cliccando sul simbolo dello XOR possiamo, ad esempio, posizionare sul Canvas un componente che calcola direttamente lo XOR dai due input, senza bisogno della circuiteria aggiuntiva presente nei precedenti paragrafi.

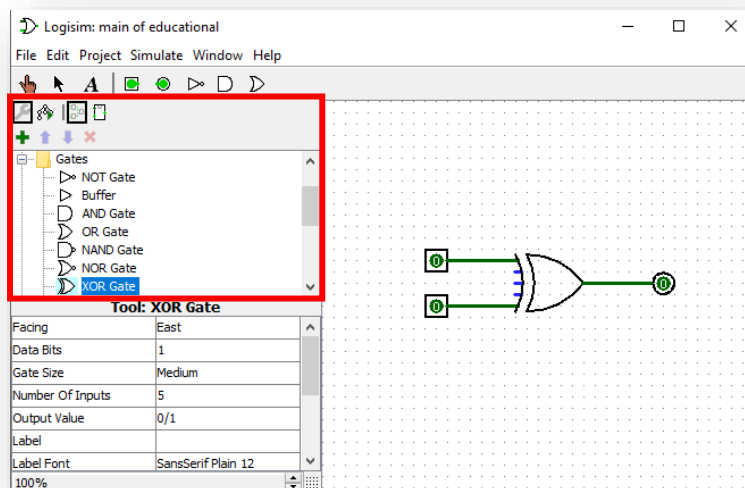


Figura 19: Pannello delle Librerie Built-in (Explorer Pane)

Logisim permette anche di aggiungere nuove librerie, selezionando il sottomenù *Load Library* dalla voce *Project* nella Menu Bar.

Esistono tre tipi di librerie in Logisim:

- *Librerie Built-in*: librerie incluse in logisim create dallo sviluppatore stesso
- *Librerie Logisim*: sono le schematiche di sottocircuiti che possono essere salvate su disco e riutilizzate in altri circuiti più complessi. I sottocircuiti verranno trattati nel paragrafo 2.8-Sottocircuiti.
- *Librerie JAR*: sono librerie realizzate in JAVA, senza quindi l'utilizzo di componenti logisim. Realizzare una libreria di questo tipo è molto più

difficile rispetto a una libreria Logisim, ma permette di creare qualsiasi tipo di componente, da uno speaker a una virtual Lan collegata ad internet attraverso l'interfaccia del pc.

È possibile rimuovere una libreria caricata su Logisim scegliendo *unload library* dal sottomenù della voce *Project*.

2.7-Attributes Table

La maggior parte dei componenti Logisim possiede degli attributi, ovvero delle proprietà che definiscono il comportamento o l'aspetto del componente. Tali proprietà vengono visualizzate nell'Attributes Table.

Per visualizzare le proprietà di un componente specifico è sufficiente cliccarvi sopra dopo aver selezionato l'icona del cursore nella Tool Bar, come mostrato in *figura 20*.

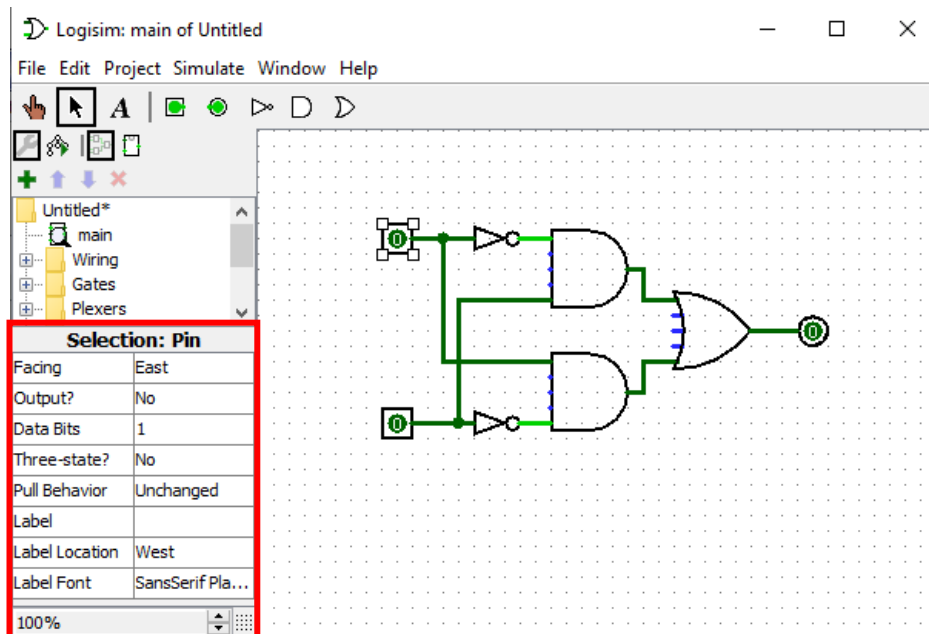


Figura 20: Tabella degli Attributi

Selezionando un qualsiasi campo della Attributes Table del componente viene data la possibilità di modificare la proprietà corrispondente. In figura vengono mostrati ad esempio i campi *Facing* per gestire l'orientamento del componente, *Data Bits* per modificare la Bit-Width del componente e *Label* per assegnare un'etichetta al componente.

2.8-Sottocircuiti

Un sottocircuito è un blocco che rappresenta una rete logica da un punto di vista di astrazione più alto, ovvero tenendo conto solamente degli ingressi della rete e della loro relazione rispetto all' uscita. Nel concetto di sottocircuito viene messo da parte lo schema circuitale della rete, permettendo quindi di riutilizzare tale blocco per costruire reti più complesse mediante moduli. Qualsiasi circuito realizzato in Logisim viene automaticamente reso disponibile dal simulatore sotto forma di sottocircuito durante la costruzione: per visualizzare e modificare il sottocircuito associato alla rete corrente è sufficiente cliccare sull'icona a forma di blocco al di sotto della Top Bar, evidenziata in rosso in *figura 21*. Il confronto tra schema circuitale e sottocircuito della rete XOR viene mostrato in *figura 21 e 22*.

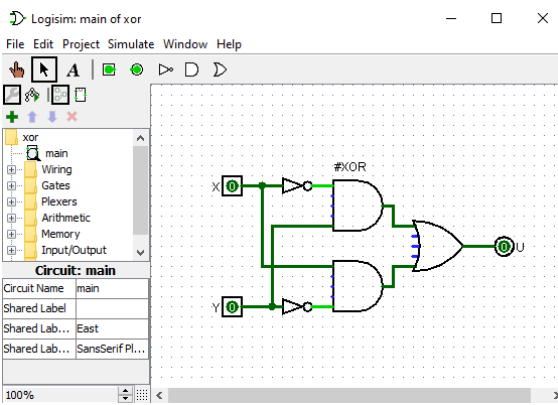


Figura 21: Schema circuitale

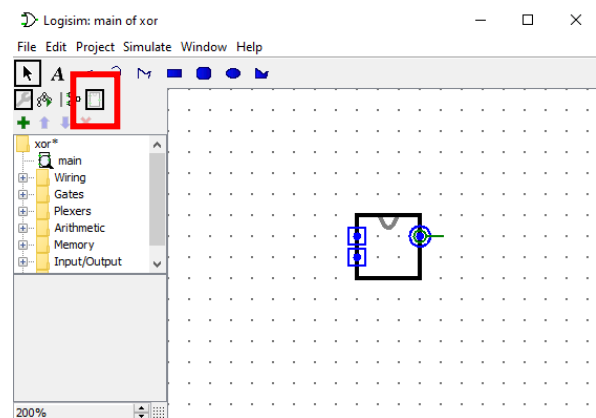


Figura 22: Simbolo associato al Sottocircuito

Dopo aver modificato il simbolo del sottocircuito, è possibile usarlo in un nuovo progetto premendo l'icona con il simbolo "+" in alto a sinistra dell'Explorer Pane. Verrà creato un altro progetto visualizzabile nell'elenco dell'Explorer Pane e selezionandone la voce cliccando due volte ne verrà visualizzato il Canvas. Cliccare una sola volta sulla voce del circuito XOR precedentemente realizzato è identico a selezionare un componente dalla Tool Bar: nel canvas apparirà il simbolo del sottocircuito, che ora potrà essere posizionato e collegato ai nuovi input, come mostrato in *figura 23*.

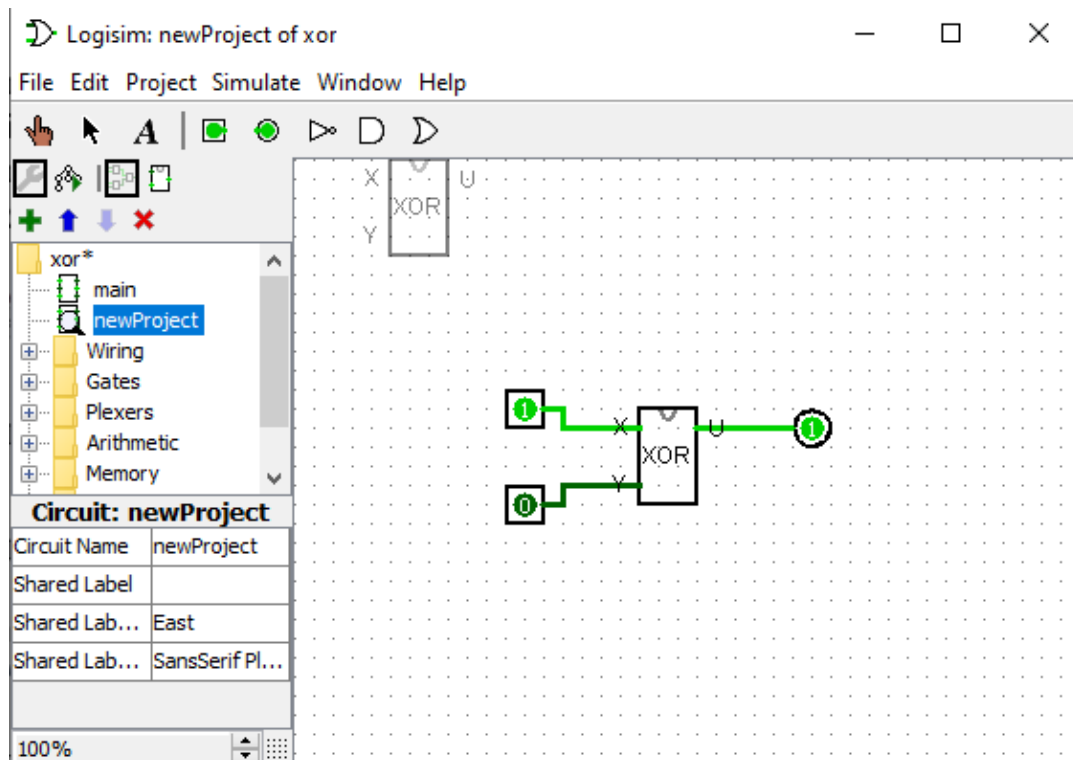


Figura 23: Impiego di un sottocircuito nel Canvas di un nuovo progetto

2.9-Memorie

Costruire reti combinatorie può essere utile per avere un primo approccio con l'interfaccia Logisim, esplorando però più a fondo l'Explorer Pane si può notare come tra la moltitudine di Librerie sia presente anche una voce chiamata *Memory*. Dentro questa Libreria sono contenuti diversi componenti dotati di memoria, tra cui Flip Flop, Registri e Memorie.

Le Memorie in Logisim sono disponibili in due tipologie: ROM e RAM. Una Memoria ROM dispone solamente di un ingresso per il segnale di abilitazione, un ingresso per gli indirizzi e una uscita per i dati contenuti nelle celle, ma il suo contenuto viene salvato anche alla chiusura del software Logisim. Una Memoria RAM invece dispone di modalità di lettura e scrittura, ma i suoi contenuti vengono azzerati alla chiusura del software o al reset della simulazione, simulando la volatilità

di una RAM reale. Il collegamento circuitale delle memorie ROM e RAM in Logisim viene mostrato in *figura 24 e 25*.

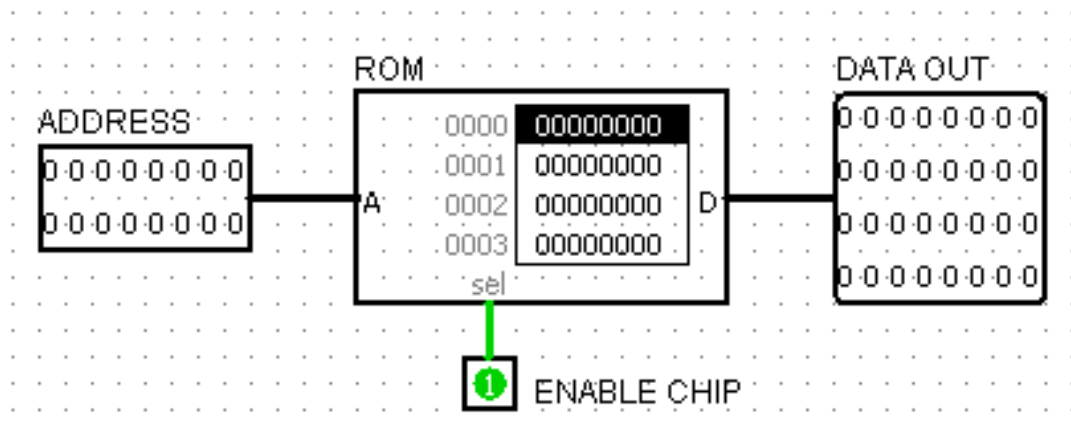


Figura 24: Schema circuitale di una Memoria ROM

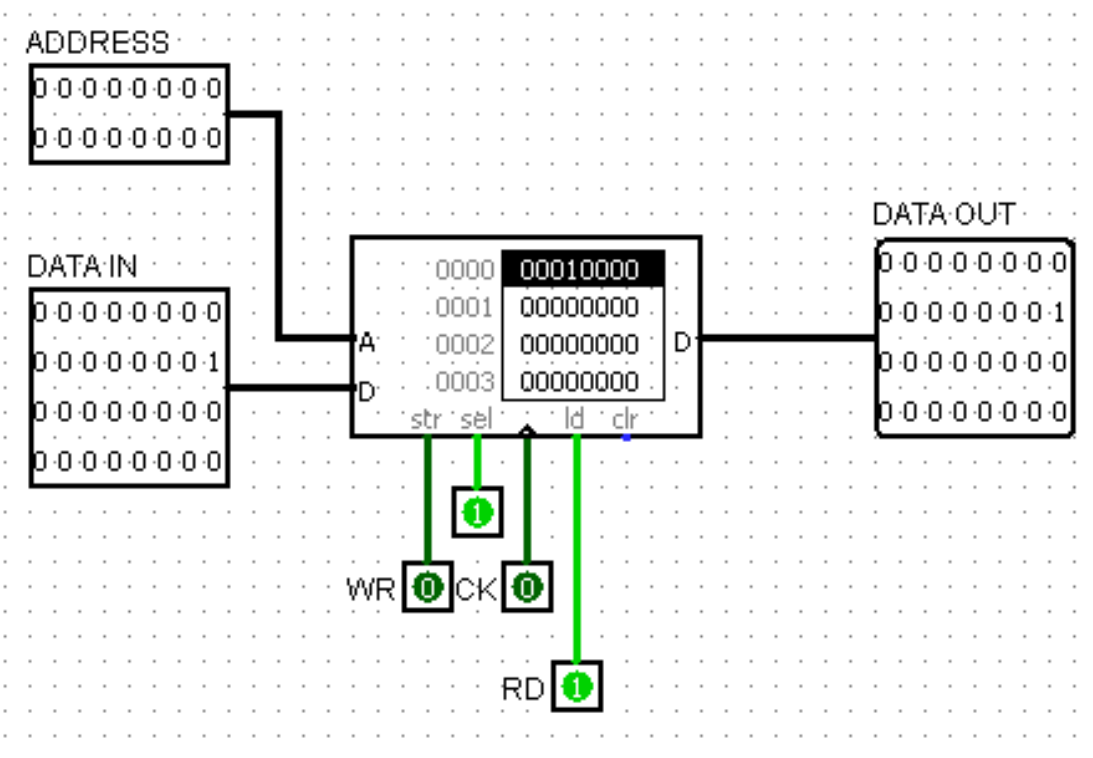


Figura 25: Schema circuitale di una Memoria RAM

La comodità di utilizzare questi componenti in Logisim sta nella possibilità di osservare il contenuto delle celle di una memoria direttamente nel canvas durante la simulazione, mediante un'interfaccia grafica presente sulle memorie, come mostrato

nelle figure. Quando viene fornito l'indirizzo di una cella alla memoria inoltre la cella corrispondente viene evidenziata con una barra nera. Questa caratteristica di visualizzazione tornerà molto utile per visualizzare il comportamento di una Macchina di Von Neumann dal vivo.

Un'ulteriore particolarità delle Memorie Logisim è la possibilità di modificarne il contenuto manualmente. Cliccando con il tasto destro sulla Memoria e selezionando la voce "edit" content, verrà visualizzata una finestra simile a un foglio Excel dove può essere modificato il contenuto di ogni singola cella, come mostrato in *figura 26*.

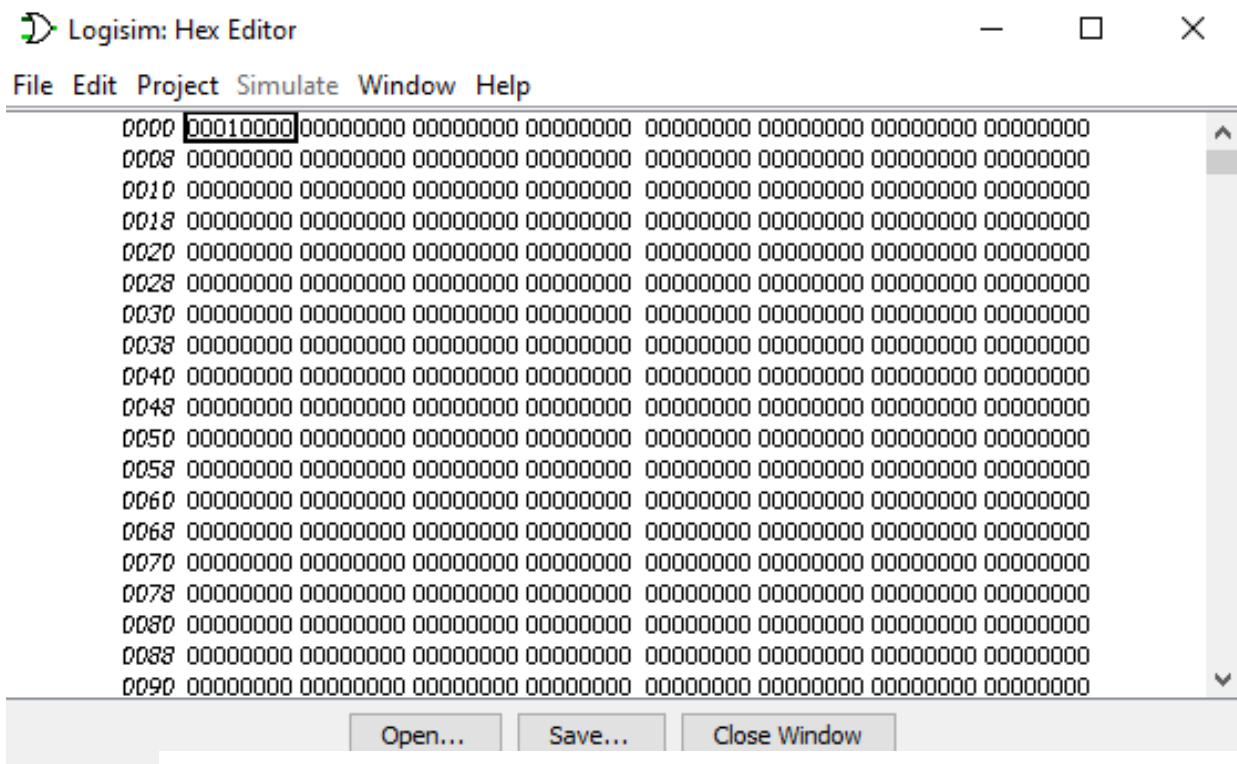


Figura 26: Editor delle celle di Memoria

CAPITOLO 3: Architettura Virtual Shock

3.1-Introduzione

Il modello di riferimento di un Calcolatore Elettronico consiste in una *Macchina digitale a Esecuzione sequenziale e Programma memorizzato*, ovvero una rete sequenziale sincrona universale in grado di eseguire istruzioni memorizzate in un programma.

Il funzionamento desiderato della macchina non dipende quindi dalla realizzazione circuitale, ma dalla sequenza di istruzioni scritta in memoria. È proprio l'universalità e la flessibilità di questo modello che ha permesso l'ampia diffusione dei calcolatori elettronici. Il tradeoff di un sistema di questo tipo consiste nella possibilità di trattare problemi molto più complessi di una rete logica tradizionale, a scapito di una minore efficienza. Nella pratica quindi il set di Istruzioni del calcolatore permette di creare una interfaccia tra Hardware e Software.

In *figura 27* viene mostrata la struttura di un calcolatore mediante uno schema a blocchi, ogni blocco è costituito da circuiti digitali e in logisim è considerabile come un Sottocircuito:

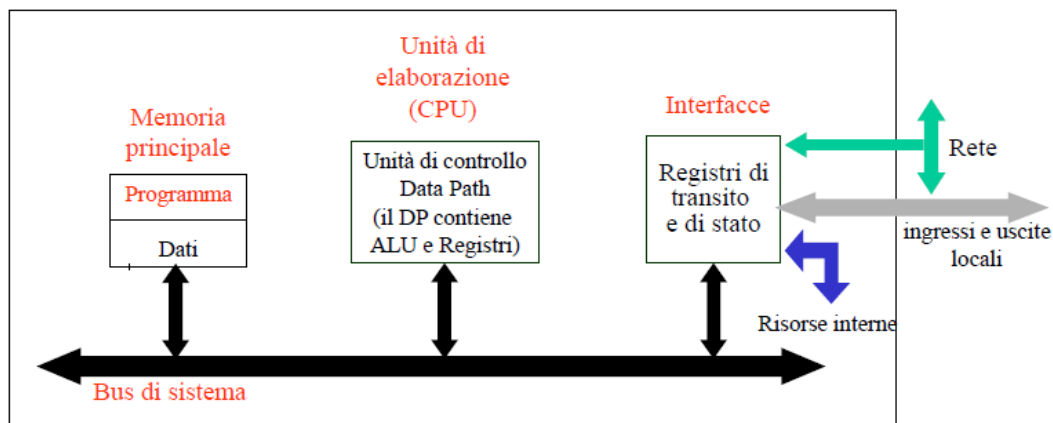


Figura 27: Schema a blocchi di un Calcolatore Elettronico (tratta dalle Slides del corso di Calcolatori Elettronici – A.A 2019)

Il blocco centrale del sistema è la CPU, ovvero il *cervello* del calcolatore. Mediante essa la Macchina è in grado di interagire con la sua Memoria e le periferiche ed eseguire le sue funzioni.

È difficile parlare di un calcolatore elettronico senza aver analizzato prima il suo blocco principale, la CPU, che permette di interpretare un programma e generare un Output. Il programma Logisim fornisce un aiuto importante per visualizzare il comportamento di un circuito complesso come una CPU, per questo motivo l'analisi dell'architettura del calcolatore presentata nei prossimi paragrafi sarà affiancata dal simulatore.

3.2-Struttura interna della CPU

La struttura logica di una CPU può essere suddivisa in due macroblocchi (o sottocircuiti): l'UDC e il DATA-PATH. L'UDC è un blocco di tipo MASTER, ovvero che controlla il comportamento di altri blocchi, in questo caso il DATA-PATH, di tipo SLAVE. La struttura interna della CPU viene mostrata in *figura 28*, dove è stata presa come punto di riferimento la struttura della CPU introdotta nelle slides di Calcolatori, mentre nella *figura 29* è mostrato il corrispettivo schema circuitale in Logisim.

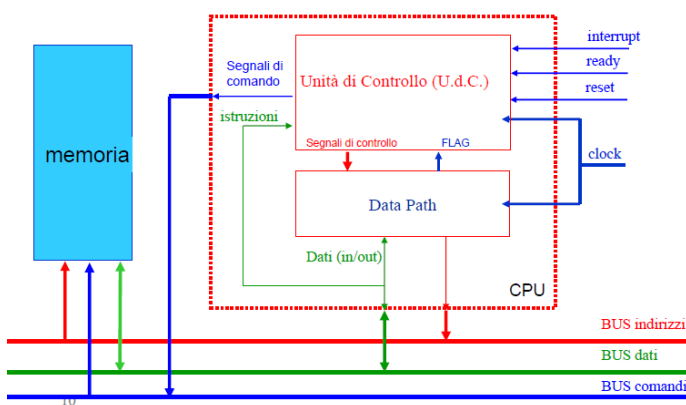


Figura 28: Schema a Blocchi della CPU (tratto dalle Slides del corso di Calcolatori Elettronici – A.A 2019)

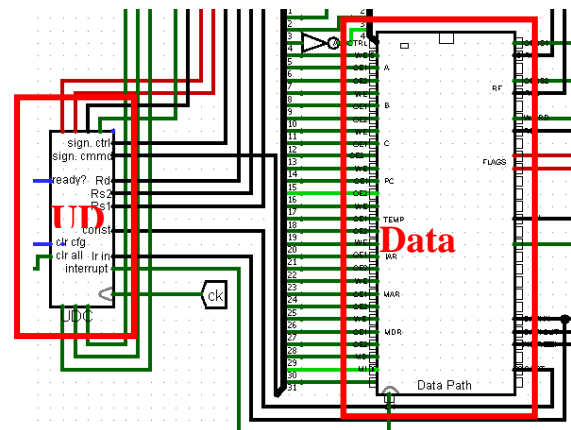


Figura 29: Schema circuitale in Logisim della CPU

Come anticipato nell'introduzione, affinché la CPU possa eseguire istruzioni è necessario un blocco che contenga in maniera ordinata queste istruzioni e gli output che tali istruzioni producono: la Memoria, evidenziata in blu in *figura 22*.

Il primo blocco che verrà affrontato nell'Architettura della CPU è il DATA-PATH, ovvero la Rete di Elaborazione. Nella pratica il Data Path è in grado di effettuare operazioni tra dati prelevati dalla memoria o forniti direttamente dalle istruzioni, e di rendere il risultato dell'operazione disponibile alle operazioni successive.

La struttura logica interna del Data Path viene presentata in *figura 30*.

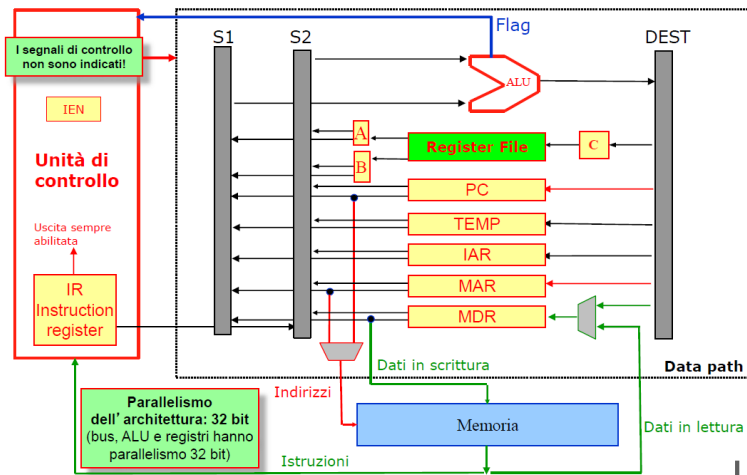


Figura 30: Visualizzazione schematica della struttura logica del DATA PATH (tratta dalle Slides del corso di Calcolatori Elettronici – A.A 2019)

Per comprendere il funzionamento di questa Rete di Elaborazione è necessario prima di tutto esaminare nel dettaglio i due blocchi principali necessari alla sua costruzione:

- Registri (come A, B, PC, TEMP), ovvero Reti Sequenziali
- ALU (blocco con outline rosso in alto), ovvero una Rete Combinatoria

3.3-Registri: Flip Flop in Parallelo

Per quanto riguarda la realizzazione logica, un registro del Data Path è innanzitutto una collezione di N flip flops posizionati in PARALLELO, con N pari alla larghezza in bit di una Word (il dato contenuto nel registro). I flip flops sono reti sequenziali, ovvero reti logiche le cui uscite non dipendono solo dagli ingressi ma anche dal tempo. Il loro scopo è campionare un dato in ingresso e aggiornare l'uscita in seguito al campionamento, ad ogni fronte di salita, mediante un colpo di clock. Si può pensare ai registri quindi come a semplici "contenitori" di dati.

Logisim fornisce già nelle *Built-in Libraries* un componente di tipo *Register*, ma verrà ugualmente mostrato un esempio di realizzazione circuitale e funzionamento usando dei flip flops, similmente al caso dello XOR precedentemente mostrato.

I componenti dotati di memoria sono accessibili aprendo la libreria Memory dall'Explorer Pane, come mostrato in *figura 31*. Si può quindi procedere a selezionare un D Flip-Flop e posizionarlo nel Canvas, come mostrato nella medesima figura.

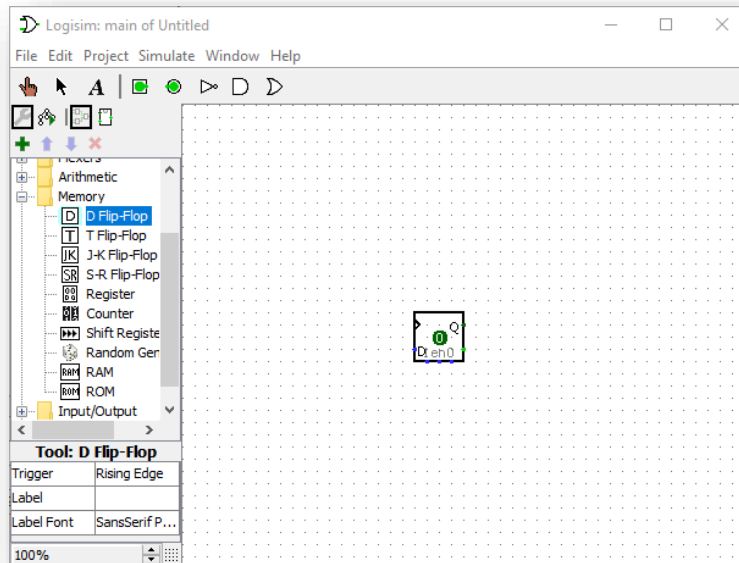


Figura 31: Componente Logisim rappresentante un Flip Flop

Si procede ora con i passaggi analoghi a quelli mostrati nel paragrafo due, aggiungendo quindi input, output e wiring al circuito. In figura 32 viene mostrata la realizzazione circuitale di un Registro con quattro Flip-Flop in parallelo, quindi un 4Bit-Register. Il circuito però non è ancora completo: bisogna collegare il segnale che fornisce il fronte di salita (Ck in figura 33) e alzare l'Enable dei Flip Flop, per renderli funzionanti. Il risultato viene mostrato in figura 33, dove sono stati aggiunti dei commenti per migliorare la leggibilità del circuito.

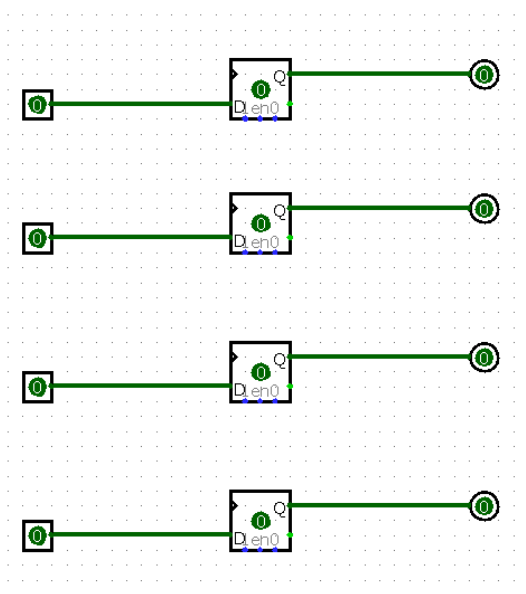


Figura 32: Flip Flop in Parallelo

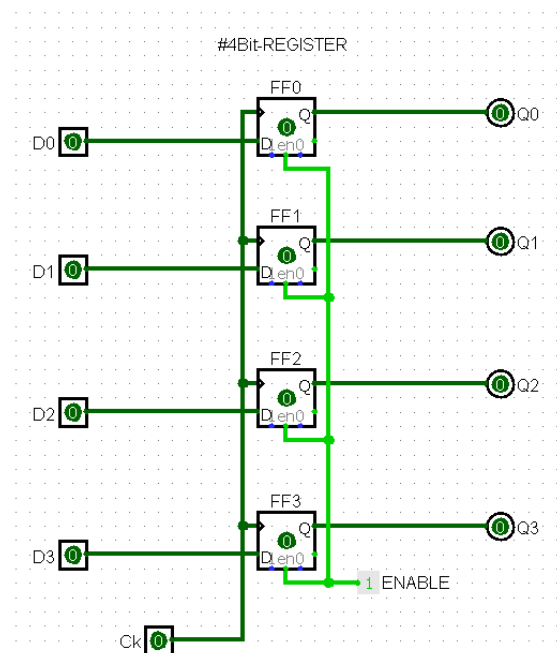


Figura 33: Collegamento del segnale di Clock ed Enable

A questo punto è possibile testare il funzionamento della rete. A livello logico, un flip flop e un registro differiscono dal fatto che mentre un singolo flip flop può memorizzare un unico bit, un registro può memorizzare numeri binari formati da N bit, con N pari al numero di flip flop in parallelo. Questo è reso possibile dal posizionamento “ordinato” dei flip flop. Il flip flop denominato FF0 si occuperà di memorizzare il primo bit del dato in ingresso, FF1 memorizza il secondo bit, e così per gli altri.

Verrà ora analizzato il comportamento del registro nel caso di un dato in ingresso pari a D=1010. Innanzitutto bisogna alzare gli input D1 e D3 dal canvas, come mostrato in figura 34. Si può notare come nonostante gli input siano cambiati l’uscita rimanga a 0. Alzando il bit denominato Ck viene simulato un fronte di salita: come mostrato in figura 35 nel momento in cui Ck si alza, i flip flop campionano gli ingressi e aggiornano le uscite col dato campionato.

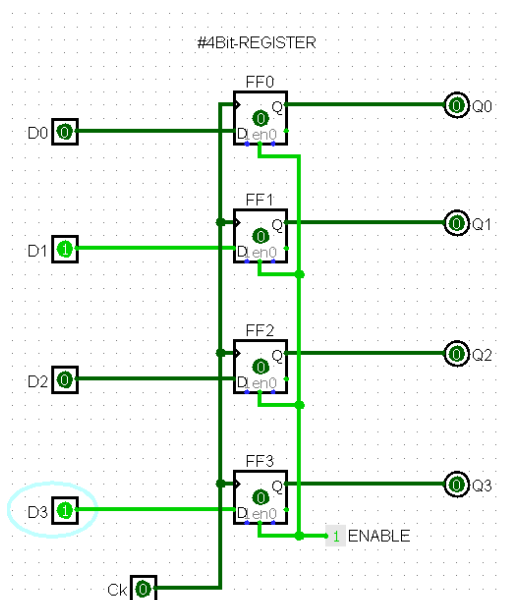


Figura 34: Visualizzazione del dato in ingresso a un Registro

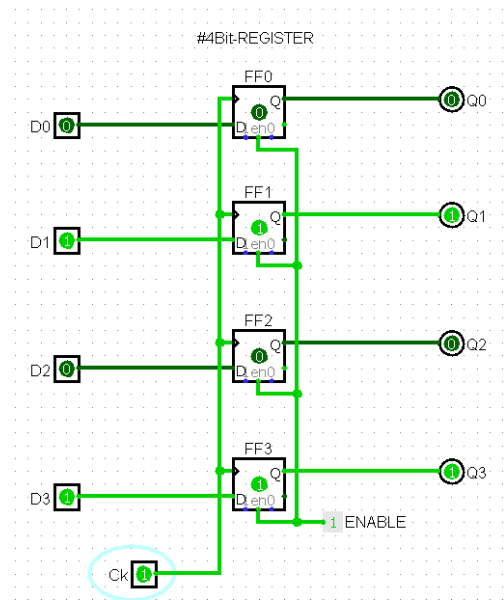
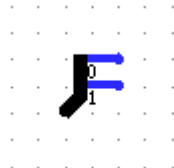


Figura 35: Campionamento dell'ingresso al fronte di salita

Il circuito realizzato è funzionale, ma sono necessarie alcune modifiche per usarlo all'interno dell'Architettura della CPU. Essendo la CPU a 32Bit, i Registri saranno composti necessariamente da 32Flip Flop. Un tale numero di input, output, FF e wires renderà sicuramente molto più difficile visualizzare il comportamento del circuito. Per far fronte a queste problematiche, Logisim mette a disposizione un componente detto *Splitter*. Uno Splitter è un tool GRAFICO che consente di accorpare un insieme di collegamenti in un unico filo, detto BUS. Selezionando il componente Splitter dalla libreria *wiring* e posizionandolo sul Canvas, si può notare come di default abbia un ingresso e due uscite (figura 36).

Nella *figura 37* viene mostrata la tabella degli attributi dello splitter: appare automaticamente quando viene cliccato il componente sul Canvas. La seconda riga permette di modificare il Fan Out del componente, mentre la terza il Fan In. È possibile anche specificare la mappatura precisa dei bit in ingresso e uscita.

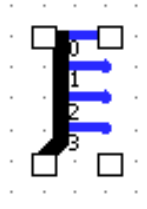


Selection: Splitter	
Facing	East
Fan Out	2
Bit Width In	2
Appearance	Left-handed
Bit 0	0 (Top)
Bit 1	1 (Bottom)

Figura 36: Pinout di uno splitter in Logisim

Figura 37: Attribute Table di uno Splitter

Nel caso del registro a 4bit è necessario un bus a quattro bit, quindi con Fan In pari a 4 e Fan Out pari a 4. In *figura 38* viene mostrato lo splitter modificato.



Selection: Splitter	
Facing	East
Fan Out	4
Bit Width In	4
Appearance	Left-handed
Bit 0	0 (Top)
Bit 1	1
Bit 2	2
Bit 3	3 (Bottom)

Figura 38: Splitter con Pin-Out pari a 4

Figura 39: Attribute Table di uno Splitter modificato

Va notato come mentre i fili dei bit in uscita aumentano all'aumentare del Fan Out, il filo in entrata rimane sempre uno. In realtà quello che cambia (graficamente) non è il numero dei fili in entrata, bensì la “densità” di informazione contenuta in quel filo.

Ora è possibile collegare lo splitter in ingresso e in uscita ai 4 FF. Va notato come mostrato in *figura 40* che i collegamenti degli splitter sono bi-direzionali.

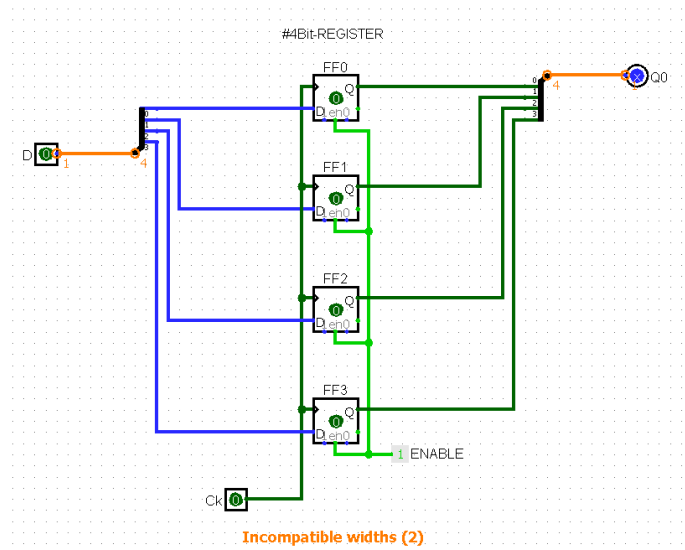


Figura 40: Collegamento di Splitter e Flip Flop in parallelo

Questa realizzazione ha però un problema. Quando ci sono degli errori di progettazione nel circuito le linee si colorano di arancione e viene visualizzato un messaggio di errore, come mostrato nella figura precedente.

In questo caso, gli splitter hanno un Fan In di 4 bit, ma gli input sono ancora a un singolo bit. In Logisim un input e un output collegati devono avere la stessa bit-width (stesso numero di bit). È necessario quindi modificare gli input e output per rispettare questa condizione attraverso la loro Tabella degli Attributi, come mostrato in *figura 41 e 42*.

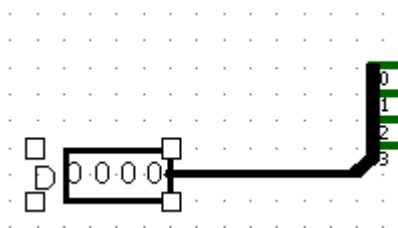


Figura 41: Pin modificato con Bit-Width pari a 4

Selection: Pin	
Facing	East
Output?	No
Data Bits	4
Three-state?	No
Pull Behavior	Unchanged
Label	D
Label Location	West
Label Font	SansSerif Plain 12

Figura 42: Attribute Table di un Pin con Bit-Width pari a 4

Ora il circuito realizzato è funzionante e avrà l'aspetto mostrato in *figura 43*.

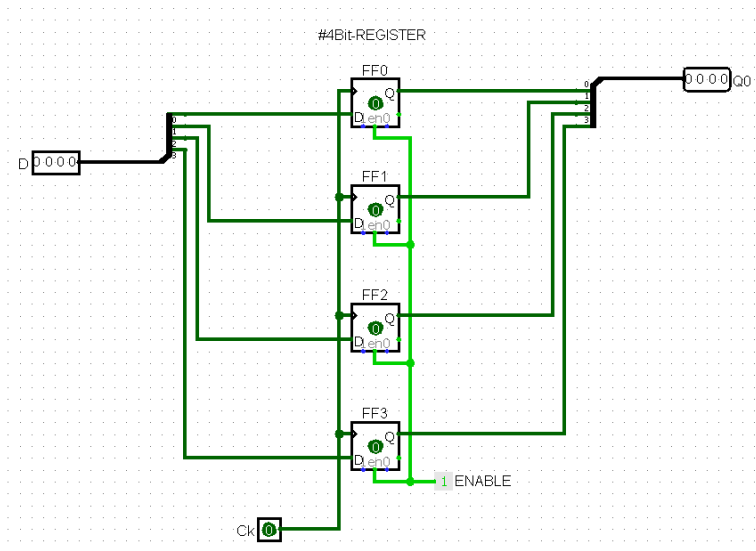


Figura 43: Schema circuitale di un 4 bit Register

A differenza di prima, il nuovo circuito presenta un ingresso al posto di quattro e una uscita, sempre al posto di quattro. I BUS, gli input multi bit e i sottocircuiti consentono di aumentare progressivamente e armonicamente il livello di astrazione, permettendo di raggiungere il livello proposto con lo schema a blocchi della macchina di Von Neumann mostrato nella prima figura del Capitolo 3- Architettura Virtual Shock.

Il passo successivo di questa realizzazione parte da un presupposto tecnico: ogni flip flop è un componente scritto in JAVA che realizza una funzione logica. I registri messi a disposizione da Logisim sono anch'essi scritti in JAVA, ma hanno un vantaggio: simulare un registro Logisim da 32b è meno pesante in termini di calcoli di simulare 32 flip flop da 1 bit. Una versione migliorata del circuito realizzato fino ad ora consiste quindi nell'impiego di un registro Logisim che sostituisca tutti i flip flop in parallelo, come mostrato in figura 45. Va notato che il circuito dal punto di vista logico è identico in entrambi i casi.

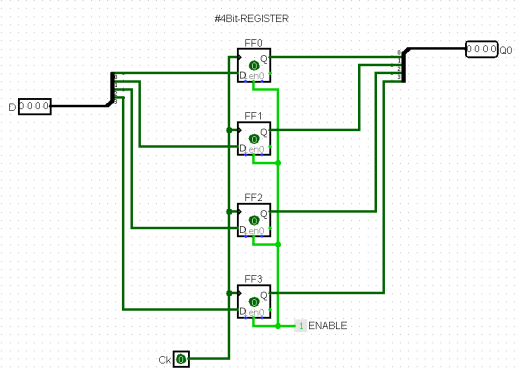


Figura 44: Schema circuitale del registro a 4Bit

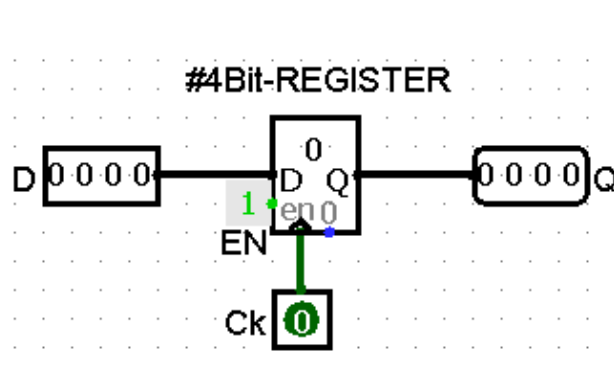


Figura 45: Componente integrato del registro Logisim

In questo tipo di realizzazione inoltre non è necessario usare gli splitter, poiché sia gli input che il registro hanno bit width pari a 4 bit. La bit width del registro può essere modificata dal pannello degli attributi, come per qualsiasi altro componente. I registri Logisim hanno un'altra caratteristica interessante: il loro contenuto viene visualizzato direttamente sul componente, rendendone l'analisi molto più comoda. Questa proprietà verrà approfondita nei successivi paragrafi.

3.4-Spostamenti tra registri: Flip Flop in Serie

Realizzare registri mediante l'editor grafico Logisim consente di visualizzare in maniera chiara e precisa il loro comportamento. Nel paragrafo precedente ad esempio è stato mostrato il comportamento di quattro Flip Flops collegati in parallelo e come ottimizzare la costruzione di tale circuito. In questo paragrafo verrà analizzata un'altra caratteristica della connessione tra FF: il collegamento in serie, e cosa comporta tale tipo di connessione.

Anche in questo caso si può iniziare partendo dal caso con bit width pari a uno. In *figura 46* viene mostrato un esempio di collegamento di due FF posti in serie. Va notato come l'uscita Q del primo flip flop viene collegata all'entrata D del secondo. Inoltre, nonostante siano presenti due flip flop, l'ingresso rimane uno solo (collegamento in serie).

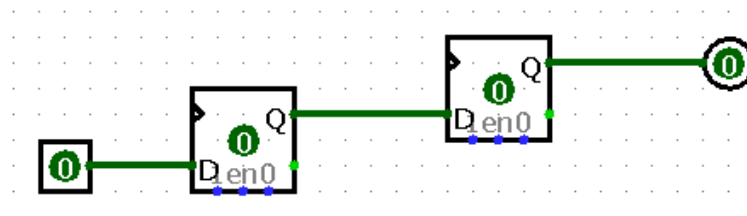


Figura 46: Due Flip Flop collegati in serie

Aggiungendo la circuiteria mancante, quindi clock e Enable, si ottiene il circuito mostrato in *figura 47*.

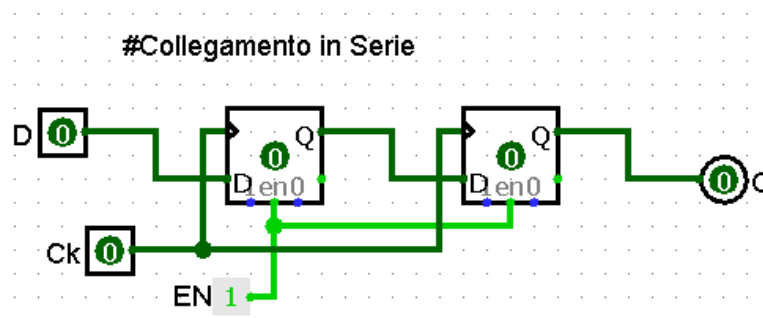


Figura 47: Schema circuitale del collegamento in serie di due Flip Flop

Uno strumento interessante per visualizzare il comportamento di una rete logica in Logisim consiste nell'impiego di LEDs per visualizzare lo stato di una linea. In *figura 48* sono stati collegati due led all'uscita del primo e del secondo FF, per visualizzare lo stato della linea Q0 e quello di Q1.

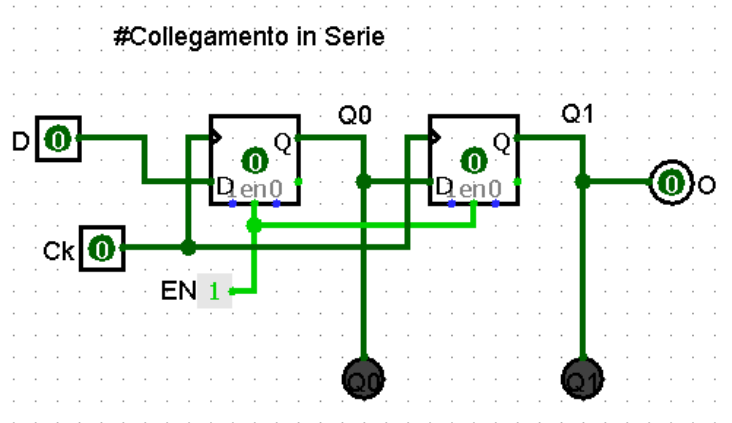


Figura 48: Collegamento di Leds alle uscite dei Flip Flop

Adesso è possibile analizzare il comportamento del circuito. Nel caso in cui l'uscita rimanga bassa entrambe le uscite rimangono a zero, come ci si aspettava. Quando D va a 1 e viene dato un colpo di clock, il primo FF campiona lo stato dell'ingresso, in questo caso 1, e aggiorna di conseguenza l'uscita alzando Q0, ci si aspetta quindi di vedere illuminarsi il primo LED, come mostrato in *figura 49*.

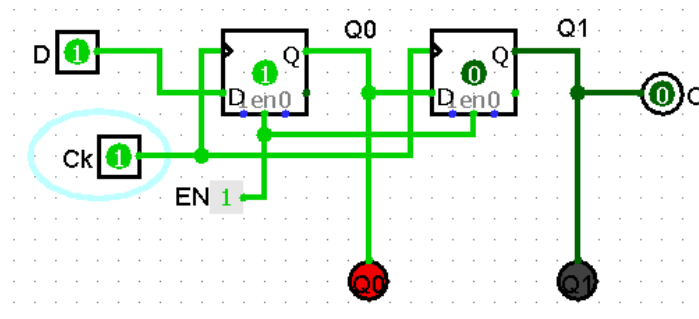


Figura 49: Visualizzazione del flusso di informazione tra Flip Flop in serie

Si può notare inoltre come Q1 sia rimasta a 0. Al momento del fronte di salita infatti essendo pilotati dallo stesso clock entrambi i FF campionano. Bisogna però considerare che l'uscita di un FF si aggiorna dopo un certo periodo tau, come una rete RC. Quindi mentre Q0 campiona l'ingresso D e comincia ad alzare l'uscita, Q1 ha già campionato l'uscita di Q0, che al momento del fronte di salita era ancora bassa. Di conseguenza Q1 rimane spento. All'arrivo del secondo colpo di clock, anche Q1 campiona un 1 e alza la sua uscita, come mostrato in *figura 50*.

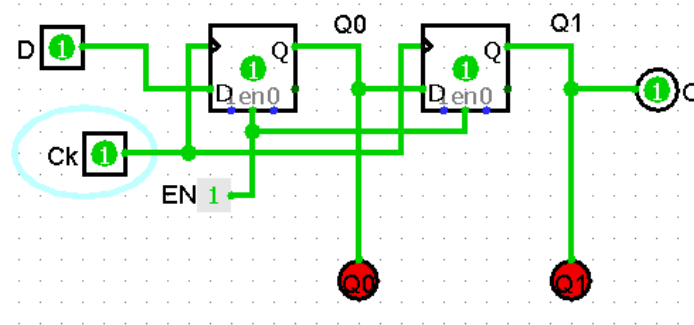


Figura 50: Stato del circuito al secondo colpo di clock

Collegare due FF in serie corrisponde quindi a “trasportare” il dato da un FF all’altro, in questo caso il dato è formato da un solo bit.

Nel caso di registri in parallelo il discorso è identico, l’unica differenza è la bit width maggiore del dato che viene trasferito. Collegare registri in serie come mostrato in figura 52 equivale quindi a effettuare “spostamenti di dati fra registri”.

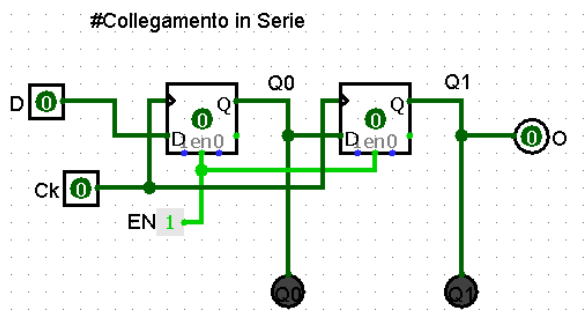


Figura 51: Collegamento in serie di Flip Flop

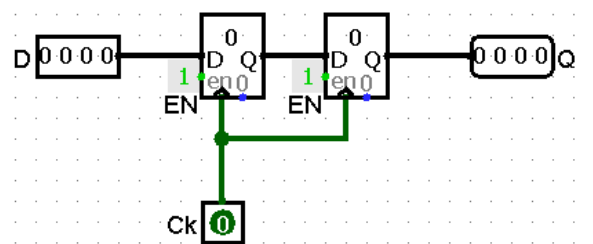


Figura 52: Collegamento in serie di Registri

3.5-Registri nel Data Path

Il collegamento in serie consente il trasferimento di dati da un registro all’altro, tuttavia tale sistema non offre un controllo sullo spostamento. Non è possibile ad esempio scegliere se scrivere o meno un registro, o se abilitare la sua uscita affinché venga letta o meno dal registro successivo. Con solo il collegamento in serie, un registro non può fare altro che campionare l’entrata e conseguentemente aggiornare l’uscita.

A tale scopo i registri del Data Path dispongono di circuiteria aggiuntiva per gestire il flusso dell’informazione in maniera controllata. Si ponga ad esempio il caso di voler abilitare o meno l’uscita di un registro, consentendo a dei registri collegati in serie di leggerla o meno. La soluzione più semplice a questo problema consiste nel collegare un Buffer Tri-State all’uscita. Un Buffer è un componente elettronico in grado

fisicamente di creare circuiti aperti o corto circuiti all'interno della linea, a seconda che il bit che lo pilota sia a UNO o a ZERO. Il nome Tri-State deriva proprio da questo comportamento: all'uscita di un buffer disabilitato infatti non è presente uno zero logico, ma un circuito elettronicamente aperto. Le implicazioni di questo terzo stato verranno mostrate più avanti. Nelle *figura 53 e 54* viene mostrato un esempio di registro con buffer in uscita.

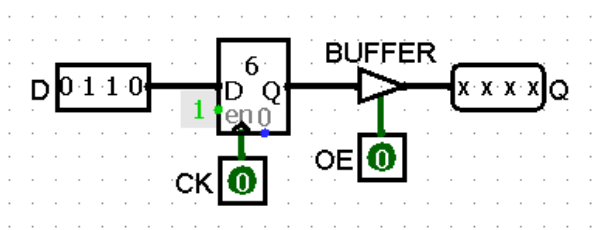


Figura 53: Registro con buffer disabilitato

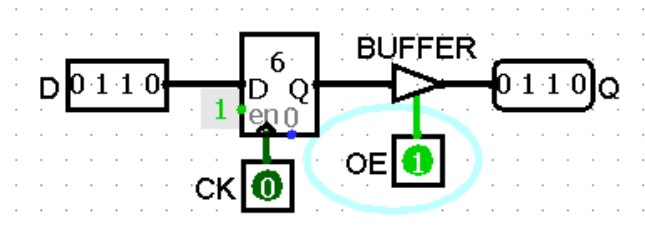


Figura 54: Registro con buffer abilitato

Il segnale OE mostrato in figura è il bit incaricato di pilotare il buffer in uscita e prende il nome di Output Enable. Quando OE è pari a 1, il buffer chiude il circuito e l'uscita diventa disponibile, come mostrato nella seconda figura, mentre nel caso di OE pari a 0 il circuito si apre e l'uscita non è più visibile all'esterno del registro. In Logisim un'uscita collegata a un circuito aperto causato da un Tri-State mostra delle X al posto dei classici 0 o 1, come mostrato in *figura 53*.

Le implicazioni di avere una uscita controllata da buffer sono molteplici, una di queste è la possibilità di sdoppiare l'uscita, come mostrato in *figura 55*.

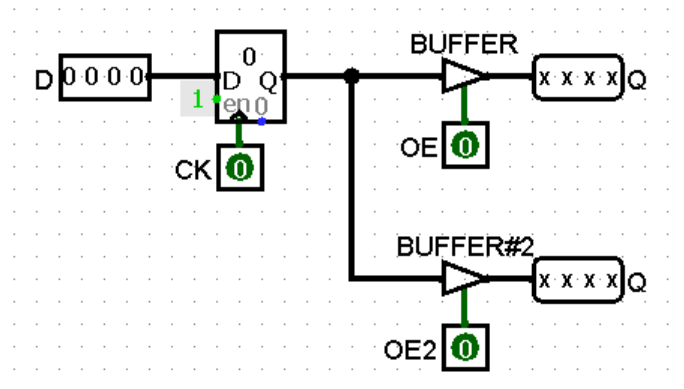


Figura 55: Registro con doppio Output Enable e doppia uscita

Sdoppiando la pista in uscita dal Registro e collegando ogni capo a un buffer diverso pilotato da un OE diverso, è possibile scegliere se l'uscita sarà visibile dalla prima o dalla seconda linea. In questo modo è possibile scegliere la direzione nella quale verrà trasferita l'informazione.

Collegando ad esempio un registro nella prima uscita e uno nella seconda, è possibile notare come abilitando il primo OE il dato venga trasferito nel primo ma non nel secondo registro (figura 56), e viceversa.

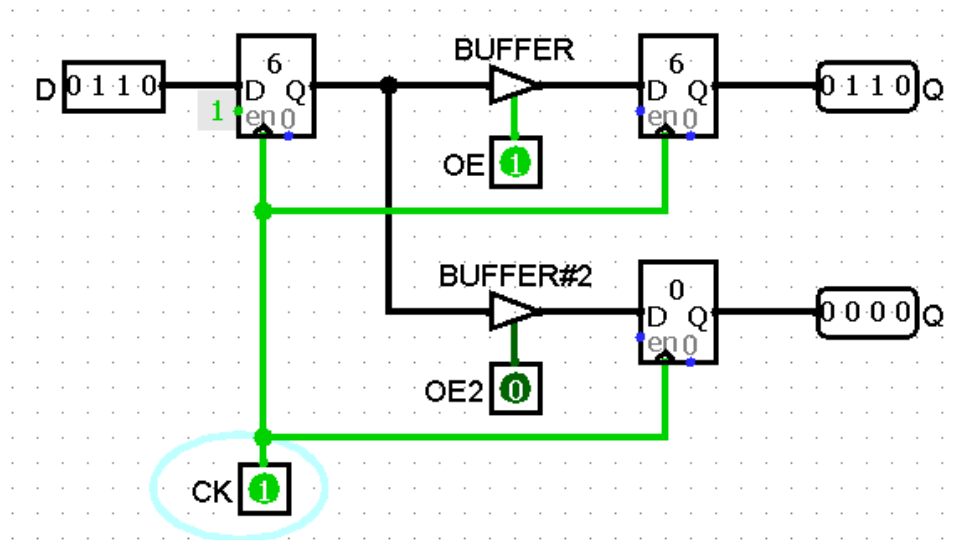


Figura 56: Esempio di funzionamento del doppio Output Enable

Si ponga ora il caso di voler abilitare o meno la lettura dei dati in ingresso di un registro, mantenendo quindi il valore del registro costante anche a fronte di un cambiamento del valore dell'ingresso. Un primo modo che potrebbe venire in mente consiste nel creare un circuito aperto all'ingresso del registro usando un buffer come per l'Output Enable. A meno che il circuito non venga chiuso, l'informazione non sarà in grado di passare e il registro non potrà campionare l'ingresso. Si può notare però che la soluzione impiegata non rispetta esattamente la consegna: il registro effettivamente non legge più i dati in ingresso, ma nella realtà non smette mai di campionare. Essendo l'ingresso del registro con circuito aperto pari a ZERO, il registro continuerà a campionare il valore ZERO, sovrascrivendo qualsiasi dato presente sul registro. Si perde quindi l'informazione contenuta nel registro, ottenendo l'effetto opposto a quello desiderato. La soluzione al problema richiede l'impiego della retroazione per mantenere o meno il dato sul registro.

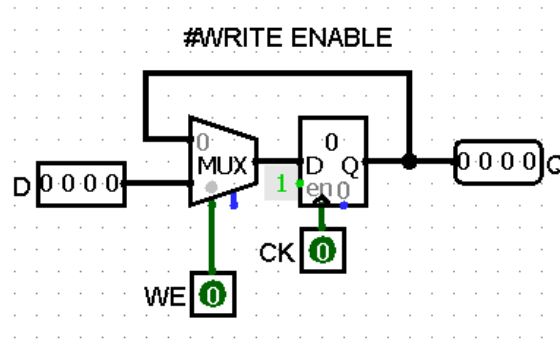


Figura 57: Schema di un Registro con Write Enable

Si consideri il circuito mostrato in *figura 57*: si può notare come all'ingresso sia stato posto un Multiplexer, pilotato da un segnale di controllo chiamato Write Enable. Se il Write Enable viene posto a 1, la situazione è uguale a quella esaminata nei paragrafi precedenti: il Mux è come se non esistesse e l'ingresso viene letto e campionato normalmente dal registro che procederà ad aggiornare l'uscita, come mostrato in *figura 58 e 59*.

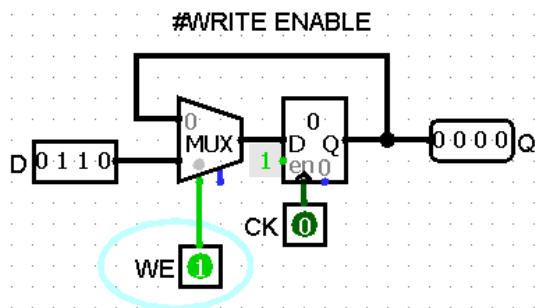


Figura 58: Stato del circuito prima del fronte di salita

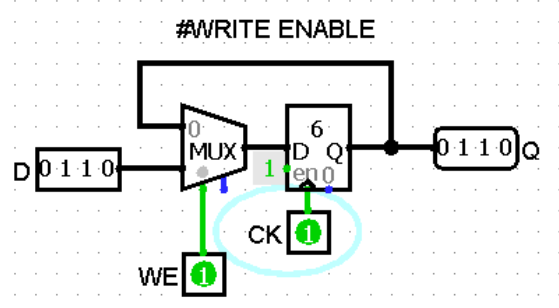


Figura 59: Aggiornamento dell'uscita dopo il fronte di salita

Nel caso in cui WE venga posto a ZERO, viene selezionato invece il primo ingresso del MUX. Ma tale ingresso mediante una linea retroazionata viene collegato all'uscita Q del registro. Questo significa che il registro sta ancora campionando, ma questa volta la sua stessa uscita. Essendo quindi l'ingresso mutato, il registro manterrà l'informazione al suo interno anche nel caso di un cambiamento dell'uscita, che è il comportamento che si voleva ottenere, come mostrato in *figura 60 e 61*.

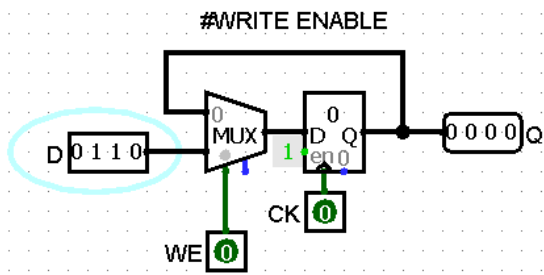


Figura 60: Circuito senza WE prima del fronte di salita

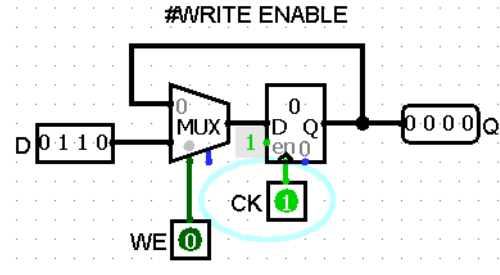


Figura 61: Circuito senza WE dopo il fronte di salita

Unendo la circuiteria di OUT e WRITE ENABLE si ottiene lo schema circuitale dei registri che verranno usati nella costruzione del Data Path, come mostrato in *figura 62*.

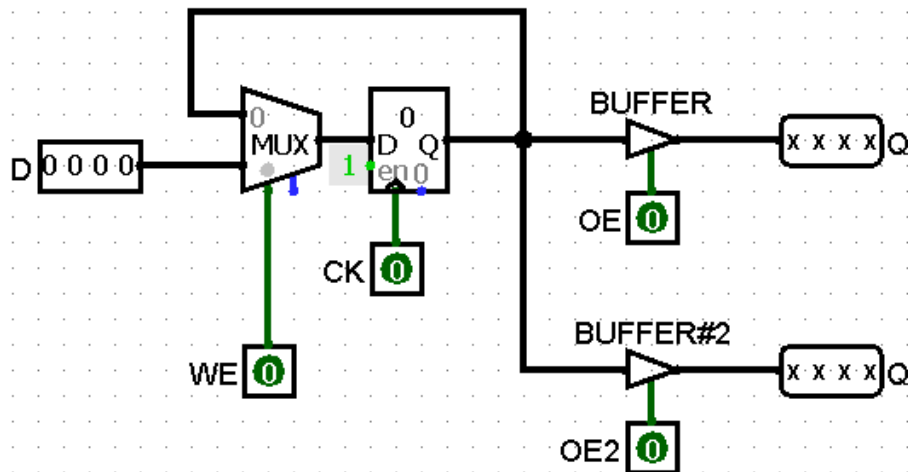


Figura 62

3.6-Alu

Il Data Path non è solamente in grado di “spostare” informazioni attraverso registri. Considerando due registri collegati in serie, interporre una rete combinatoria tra di essi corrisponde ad avere una Funzione di Trasferimento tra ingresso e uscita. Si supponga ad esempio di avere una rete combinatoria che incrementa di 1 il valore aritmetico del dato in ingresso, e di connettere a monte e a valle della rete dei registri. La relazione tra il dato contenuto nel registro in uscita (R_d) e quello in entrata (R_s) è la seguente: $R_d = R_s + 1$, in maniera analoga a $y = x + 1$. La situazione illustrata viene mostrata in *figura 63*.

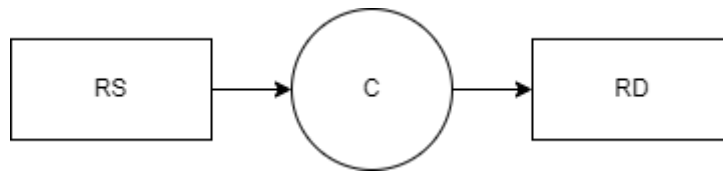


Figura 63: Visualizzazione dello spostamento e trasformazione di dati tra registri con schema a blocchi

Un sistema di questo tipo permette un controllo totale del “flusso” dell’informazione, riuscendo a operare su “direzione” dei dati e “trasformazione” degli stessi.

Una rete combinatoria in grado di eseguire una sola operazione non è però molto utile. È a questo scopo che viene introdotto un componente migliorato e più plastico: la ALU, che grazie alla sua circuiteria è in grado di effettuare una moltitudine di operazioni di tipo, come suggerisce il nome, Aritmetico e Logico. Con Aritmetico si intendono ad esempio operazioni di Somma e Sottrazione, mentre le operazioni logiche sono ad esempio AND, OR, XOR.

Una rete combinatoria in grado di eseguire diverse operazioni può essere realizzata in diversi modi, ad esempio costruendo una rete diversa per ogni operazione desiderata e usando un MUX per scegliere quale operazione effettuare. Alternativamente, usando le leggi della logica binaria è possibile modificare gli ingressi per modificare l’operazione tramite una rete detta di “Pre-Elaborazione”, che verrà presentata più avanti. Ciò consente di cambiare operazione senza cambiare la rete di calcolo, che rimane sempre la stessa: il Sommatore. Un sommatore è una rete combinatoria in grado di effettuare un’unica operazione molto importante: la Somma Aritmetica. Una somma aritmetica binaria è uguale a quella nel sistema decimale, con l’ unica differenza di usare un alfabeto in base 2.

Il principio con la quale si calcola la somma per numeri composti da più bit è lo stesso della somma in colonna: se la somma di due numeri in colonna va oltre il valore consentito dall’alfabeto (1 in questo caso) allora in uscita dalla colonna abbiamo un 1 di riporto, che andrà sommato poi alla colonna successiva.

Scomponendo il problema in pezzi, innanzitutto si deve realizzare una somma di un’unica colonna, ovvero due bit in entrata (A e B) e due bit in uscita (Somma e Carry Out). Questo circuito prende il nome di Half Adder, e la sua tabella di verità è mostrata in *figura 64*. La realizzazione circuitale di un Half Adder è invece mostrata in *figura 65*.

A	B	$S = A \text{ XOR } B$	$Co = A \text{ AND } B$
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Figura 64: Tabella di verità di un Half Adder

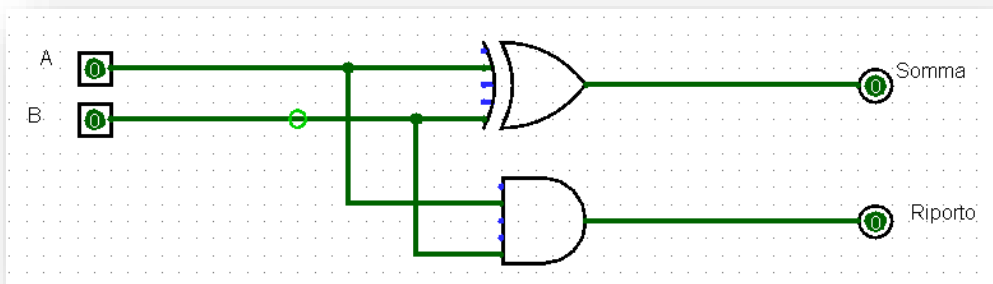


Figura 65: Schema circuitale di un Half Adder

Si può notare dalla tabella di verità del circuito precedente che l'Half Adder in realtà non calcola la somma in colonna completa. Manca una cosa: il riporto in entrata, proveniente dalla colonna precedente.

Il circuito completo per la somma in colonna viene chiamato Full Adder, la sua tabella di verità viene mostrata in *figura 66*, lo schema circuitale viene invece mostrato in *figura 67*.

Cl	A	B	S	Co
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Figura 66: Tabella di verità di un Full Adder

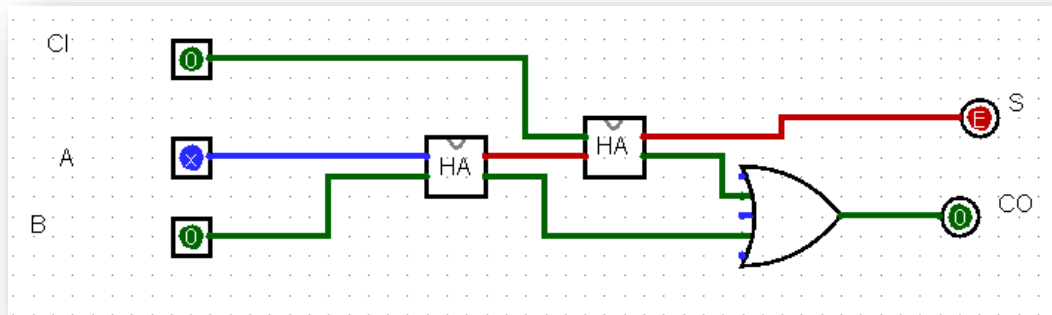


Figura 67: Schema circuitale di un Full Adder

Ora il circuito Full Adder è in grado di effettuare somme in colonna tra due bit. Collegando i Carry dei Full Adder in serie e i bit degli addendi in parallelo è possibile effettuare la somma in colonna di numeri formati da più bit. Il circuito risultante prende il nome di *sommatore*, in figura 68 viene mostrato un sommatore a 4Bit, che prende in ingresso due addendi da 4bit l'uno e manda in uscita un output da 4bit.

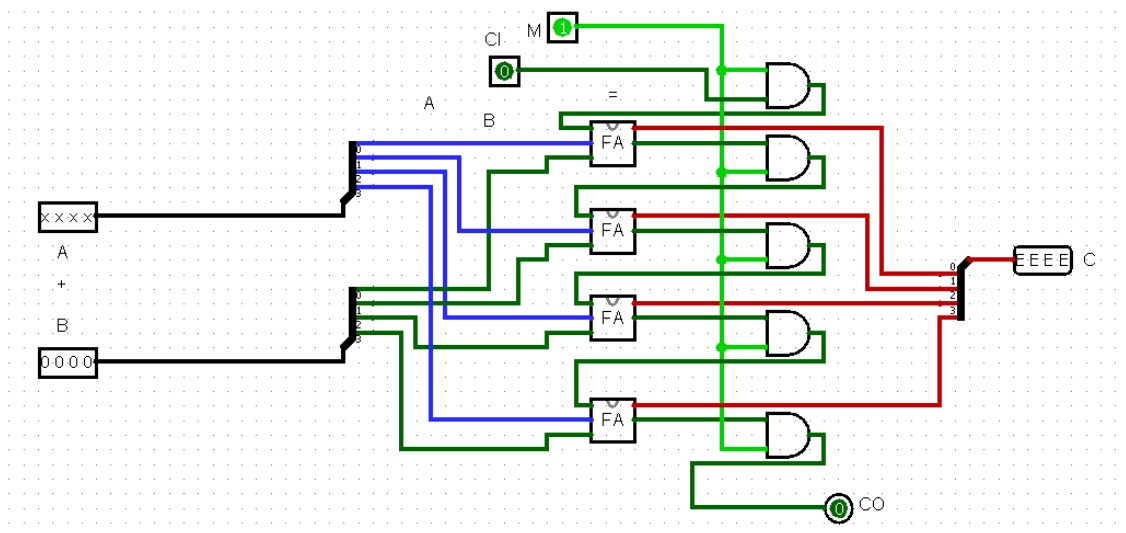


Figura 68: Schema circuitale di un sommatore a 4Bit

L'architettura è sviluppata a 32bit, perciò tutti i componenti dovranno avere 32 ingressi. Un sommatore da 32 bit avrà quindi 32 full adder in parallelo con il Carry in serie. I sottocircuiti e i bus aiutano molto in questi casi, poiché riducono l'inquinamento visivo causato da eccessive matasse di fili, come mostrato nella figura 69, dove per realizzare un sommatore a 32bit sono stati usati 8 sommatore da 4bit.

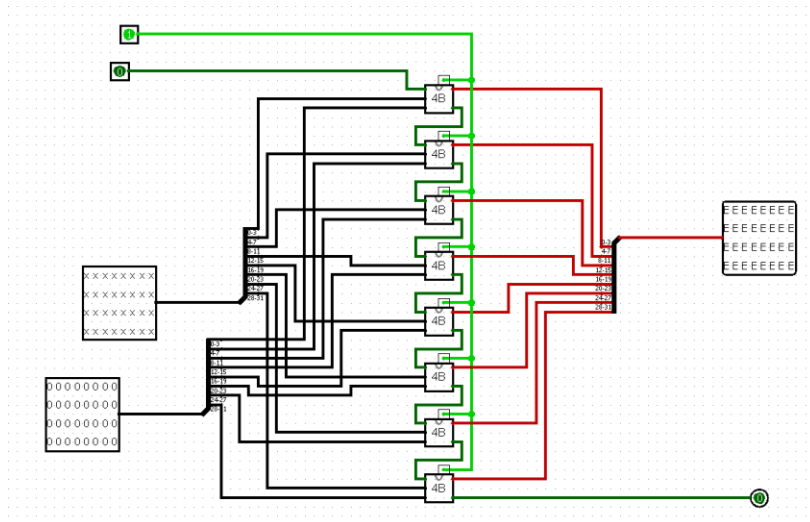


Figura 69: Schema circuitale di un sommatore a 32Bit

La rete realizzata fino a questo punto è quindi in grado di eseguire somme aritmetiche tra numeri espressi in codice binario. Questa rete però viene chiamata Sommatore, non ALU. Lo scopo della ALU è svolgere diversi tipi di operazione, e come anticipato la flessibilità dell'ALU è ottenuta mediante l'impiego di circuiteria aggiuntiva, la Rete di Preelaborazione. Si consideri ad esempio il caso di una operazione di sottrazione: $A-B$. Come spiegato in calcolatori modulo 1, mediante il complemento a 2 tale operazione può essere riscritta come: $A-B = A+{}^2B = A+\text{NOT}(B)+1$. Modificando gli ingressi e mantenendo di base l'operazione di somma, sia possibile eseguire operazioni diverse dalla somma. Lo scopo della rete di preelaborazione è proprio quello di rendere il sistema versatile nella scelta di operazioni da effettuare. Il suo schema Logico viene mostrato in figura 70, mentre la realizzazione in Logisim viene mostrata in figura 71.

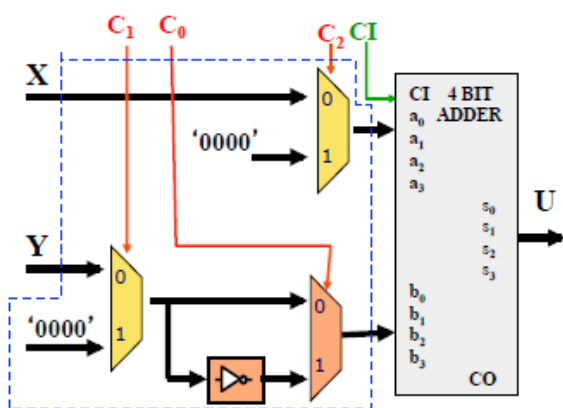


Figura 70: Schema logico di una Rete di Preelaborazione (tratto dalle Slides del corso di Calcolatori Elettronici – A.A 2019)

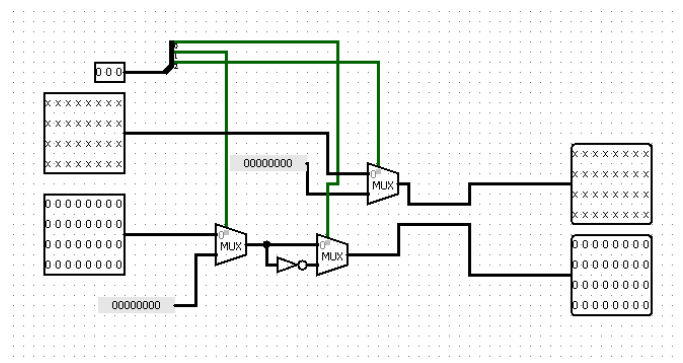


Figura 71: Circuito della Rete di Preelaborazione realizzato in Logisim

Un sommatore dotato di Rete di Preelaborazione viene chiamata ALU, come specificato nelle slides di Calcolatori Elettronici Modulo 1, e con l'aggiunta di un

AND tra i riporti in cascata per realizzare il Bit di Modalità Logica o Aritmetica è possibile ottenere tutte le operazioni riportate in *figura 72*:

$C_2C_1C_0$	$M = 1$		$M = 0$
	CI		
	0	1	
000	$X + Y$	$X + Y + 1$	$X \oplus Y$
001	$X - Y - 1$	$X - Y$	$X \oplus Y'$
010	X	$X + 1$	X
011	$X - 1$	X	X'
100	Y	$Y + 1$	Y
101	$-Y - 1$	$-Y$	Y'
110	0	1	0000
111	-1	0	1111

Figura 72: Configurazione dei bit $C_2C_1C_0$, M e I per eseguire operazioni diverse nella ALU

3.7-Struttura logica del Data Path e codice RTL

L'impiego della rete combinatoria della ALU unita alla possibilità di trasferire l'informazione tramite i registri consente di assemblare la rete di elaborazione anticipata nel paragrafo 3.2, il Data Path. Lo schema logico di tale rete viene richiamato in *figura 73*.

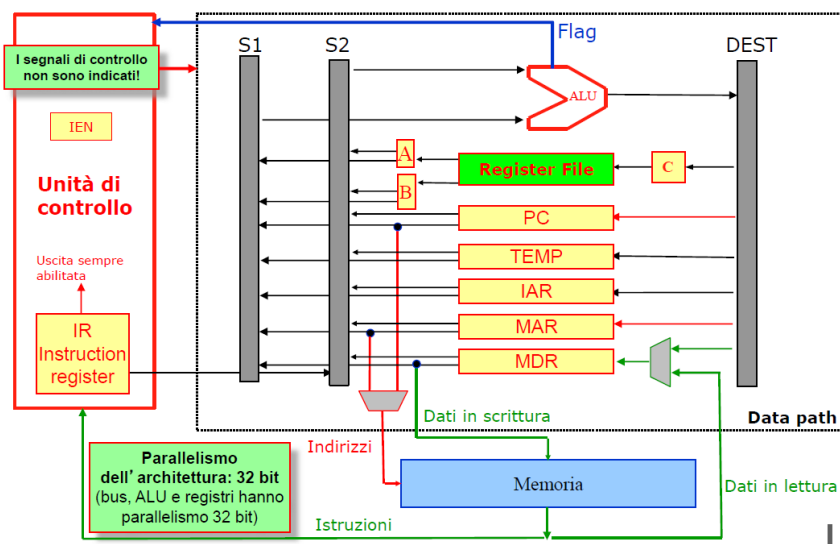


Figura 73: Schema Logico della Rete di Elaborazione del Data Path (tratta dalle Slides del corso di Calcolatori Elettronici – A.A. 2019)

Come anticipato nel paragrafo 3.4, sebbene il collegamento in serie di registri consenta il trasferimento di dati tra essi, sono i segnali di ENABLE che permettono il controllo del flusso. Non a caso, i segnali che pilotano gli Enable del Data Path vengono chiamati *Segnali di Controllo*. Conseguentemente, la modifica dei dati, e quindi il calcolo delle operazioni, viene affidata alla rete logica della ALU.

Partendo proprio da quest'ultima, è possibile notare come al suo ingresso siano presenti due BUS verticali: il primo, denominato S1, corrisponde al primo operando della ALU, mentre S2 corrisponde al secondo. L'uscita è collegata invece al BUS DEST, posto in verticale sulla destra, dove andrà il risultato finale dell'operazione. Nel caso si voglia ad esempio effettuare una somma mediante l'alu, quindi il caso $Op1+Op2=Res$, Op1 andrà messo su S1, Op2 su S2 e Res andrà invece su DEST.

È possibile notare inoltre come tutti i registri di colore giallo del Data Path in *figura 73* siano collegati sia ai bus Sorgente che al bus Destinazione, con la ALU interposta tra i due BUS. Questo è il caso introdotto all'inizio del paragrafo 3.5, ovvero dove due registri, uno sorgente e uno destinazione, sono collegati in serie con una rete combinatoria interposta fra i due (*figura 74*).

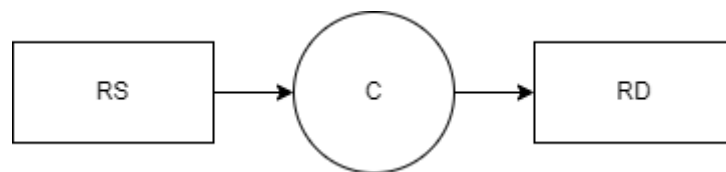


Figura 74: Schema logico del collegamento in serie di due registri Rs e Rd con rete combinatoria

Riordinando la posizione di RS e RD della *figura 74* per creare una situazione visiva più simile a quella del Data Path si può ottenere lo schema di flusso mostrato in *figura 75*:

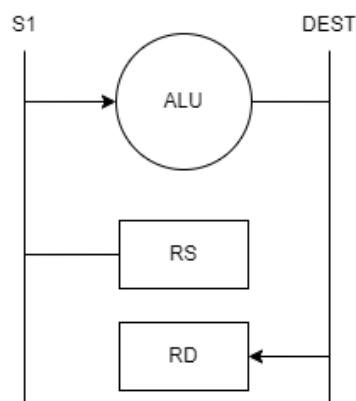


Figura 75: Adattamento dello schema logico del collegamento in serie con la struttura interna del Data Path

La differenza in questo caso rispetto al paragrafo 3.5 sta nel fatto che in ingresso alla ALU non c'è un unico registro, bensì molteplici registri collegati tutti allo stesso BUS. Rimane da capire come è possibile avere uscite di reti diverse collegate tra loro.

Normalmente se le uscite di reti diverse vengono collegate alla stessa pista, si verifica un conflitto nel nodo di collegamento e il dato in uscita non contiene più informazione ma rumore senza significato. I registri del Data Path però sono diversi da una semplice rete sequenziale: sono provvisti di OUTPUT ENABLE. Questo consente di scegliere quale registro collegare, e mediante lo sdoppiamento dell'uscita è possibile scegliere se usare tale registro come primo o secondo operando sorgente per la ALU. La condizione di funzionamento di questo sistema è che, su ogni BUS sorgente, ci sia un solo registro abilitato alla volta, tutti gli altri devono avere uscita Tri-state. In questo modo non ci saranno conflitti tra le uscite dei registri.

Per quanto riguarda il BUS destinazione invece, nonostante vi siano collegati molteplici registri, sono le entrate di tali registri a essere collegate. Sebbene sia possibile leggere un solo registro alla volta, per quanto riguarda la scrittura non ci sono restrizioni e si possono scrivere tutti i registri che si desidera in contemporanea.

L'operazione di spostamento di dati tra registri mostrato in *figura 74* passando attraverso la rete combinatoria della ALU ha un significato molto importante per un calcolatore elettronico: questo tipo di operazione viene chiamata Microspostamento o Microistruzione. Come mostrato nel paragrafo dei registri in serie, uno spostamento tra due registri adiacenti avviene in un periodo di clock, di conseguenza anche una microistruzione impiegherà un unico periodo di clock per essere portata a termine. Uno spostamento di dati tra registri del Data Path però non avviene semplicemente come nel caso dei registri in serie. Si ricorda che i registri sono stati modificati per avere il controllo del flusso dei dati tramite aggiunta di Mux in ingresso (WE) e buffer in uscita (OE). Come mostrato in *figura 76*, l'OUTPUT ENABLE del registro sorgente e il WRITE ENABLE del registro destinazione devono essere entrambi attivi nello stesso periodo di clock, altrimenti il registro sorgente non avrà modo di rendere visibile la sua uscita, e il registro destinazione non sarà in grado di campionare il dato reso disponibile dalla sorgente.

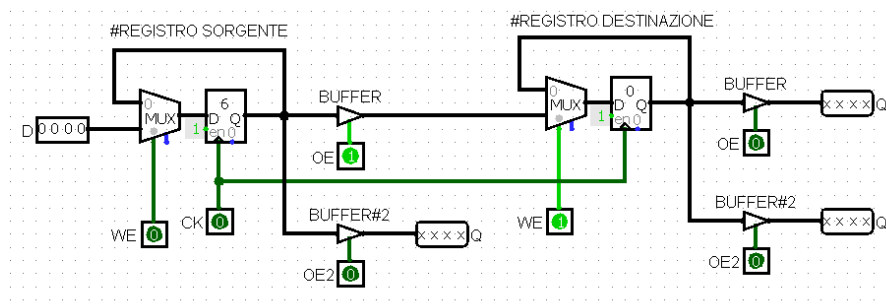


Figura 76: Stato del circuito prima del fronte di salita con Write Enable e Output Enable abilitati

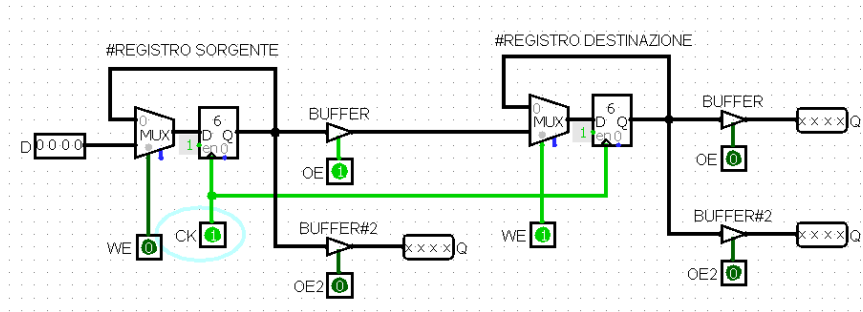


Figura 77: Al fronte di salita il registro destinazione campiona correttamente il dato

Come si può notare dalla figura, affinché sia consentito il passaggio un dato deve superare le due “barriere” dell’Output Enable e del Write Enable. Il contenuto del Registro Sorgente quindi viene spostato nel Registro destinazione in un periodo di clock. Dal punto di vista dei segnali di controllo, vanno quindi alzati contemporaneamente il pin denominato WE del registro destinazione e il pin denominato OE del registro sorgente, come mostrato in *figura 77*. Graficamente parlando, parlare di spostamenti di registri in termini di segnali di Enable risulta molto macchinoso. È per questo motivo che è stato proposto un metodo più intuitivo di rappresentazione delle micro-operazioni: il Codice RTL.

Supponendo ad esempio di voler esprimere il microspostamento di dati tra il registro A e il registro C del Data Path, il codice RTL per rappresentare tale spostamento è il seguente: $C \leftarrow A$. Per convenzione nel campo di destra si scrive il registro sorgente, mentre a sinistra viene scritto il registro destinazione. Scrivere $C \leftarrow A$ equivale quindi a inviare nello stesso periodo di clock i segnali Write Enable di C e Output Enable di A. Uno spostamento di dati da A a C deve passare inevitabilmente dalla rete combinatoria dell’ALU. Per evitare che il dato in A venga modificato dalla rete, è necessario impostare i segnali di controllo per l’ALU affinché effettui l’operazione $Op1+0$. In tal modo l’espressione completa dello spostamento diventa $C \leftarrow A+0$, che equivale a scrivere $C \leftarrow A$. Modificando i segnali di controllo della ALU è possibile modificare il valore del dato che andrà caricato in C. Settando la ALU in modalità somma, ad esempio, la relazione ingresso – uscita diventa $Op1+Op2=Ris$, questo consente quindi di scrivere l’istruzione RTL $C \leftarrow A + B$.

Per definire una Microistruzione quindi è necessario non solo definire gli Enable dei registri, ma anche i segnali di controllo della ALU, che non è mai disabilitata ed effettua sempre una operazione ad ogni spostamento.

3.8-Registri Specializzati e concetto di ISTRUZIONE

I nomi dei registri gialli mostrati nella *figura 73* sono assegnati convenzionalmente e rappresentano la funzione di tali registri nell’Architettura.

- MAR e MDR mediante i Mux colorati in *figura 73* possono essere usati per lo spostamento di dati da o verso Memoria e Periferiche seguendo il concetto di Dati e Indirizzi spiegato nel paragrafo 2.8. Si ricorda infatti che una Memoria altro non è che una serie di Registri in Parallelo e un indirizzo è sostanzialmente un modo per ordinare logicamente e aritmeticamente gli enable che la pilotano. Il concetto di Indirizzo quindi indica la traduzione di un dato contenuto in MAR in un enable usato per selezionare la cella di memoria desiderata, mediante circuiteria aggiuntiva che verrà analizzata in seguito. MDR invece fornisce o contiene il dato vero e proprio, è quindi in serie con gli ingressi dei registri della Memoria.
- PC contiene anch'esso un indirizzo e il suo scopo è legato alla PROGRAMMABILITA' del calcolatore
- IAR viene usato per la gestione degli Interrupt
- TEMP può essere usato come registro di appoggio per i calcoli
- I Registri A,B e C invece rappresentano intuitivamente i “contenitori” degli operandi sorgente e destinazione della ALU, e hanno lo scopo di collegare il componente mostrato in verde in *figura 73* detto Register File con il resto del Data Path.

Il Register File è una collezione di 32 Registri posizionati in parallelo tra loro. Essendo composto da Registri, anche il Register File viene controllato mediante degli enable, diversi per ogni registro. Si ricorda inoltre che ogni registro singolo è a sua volta composto da 32 Flip Flop, anch'essi in parallelo tra loro. La sua funzione è di Memoria Temporanea per i risultati di operazioni eseguite nel Data Path. Alcune operazioni complesse infatti possono richiedere il calcolo di diversi risultati intermedi, i quali senza l'ausilio del Register File andrebbero immagazzinati in memoria, rallentando quindi l'esecuzione della Macchina.

Il Register File dal punto di vista del funzionamento logico è quindi paragonabile proprio a una memoria, come quella esterna al Data Path. I segnali di controllo necessari per pilotare il Register File sono quindi gli stessi di una memoria: essendo 32 i registri selezionabili saranno necessari 5 bit di indirizzo più circuiteria per tradurre indirizzo in enable, più 3 enable per programmare la memoria in modalità scrittura (Registro Destinazione) o lettura (Registri Sorgente). Per realizzare il Register File in Logisim è quindi stata impiegata proprio una memoria RAM, come mostrato in *figura 78*.

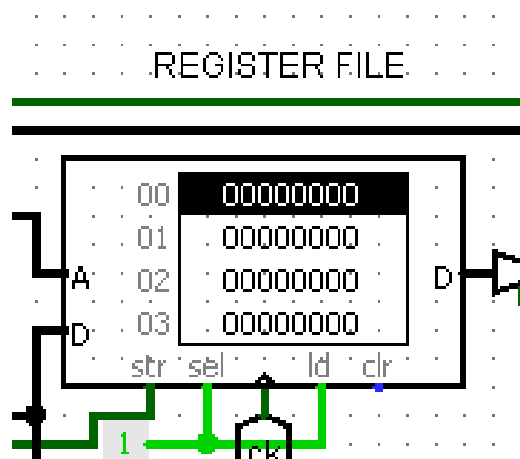


Figura 78: Memoria RAM impiegata in Logisim per realizzare il Register File

Il concetto di Registro Destinazione e Sorgente è molto importante per comprendere il funzionamento logico delle micro-operazioni RTL. Gli spostamenti nel Data Path infatti ora non sono più tra registri casuali, bensì tra “contenitori” con un nome e uno scopo logico specifico: non più spostare dati tra registri, bensì prelevare i dati contenuti nel Register File, modificarli mediante la ALU per poi caricarli nuovamente in un diverso registro del Register File, aggiornandone il contenuto, per poi eventualmente caricare il risultato in memoria o prelevare nuovi operandi mediante i registri MAR e MDR. In *figura 79* è mostrata la rappresentazione logica del flusso di questi spostamenti dal punto di vista dei Registri del Register File.

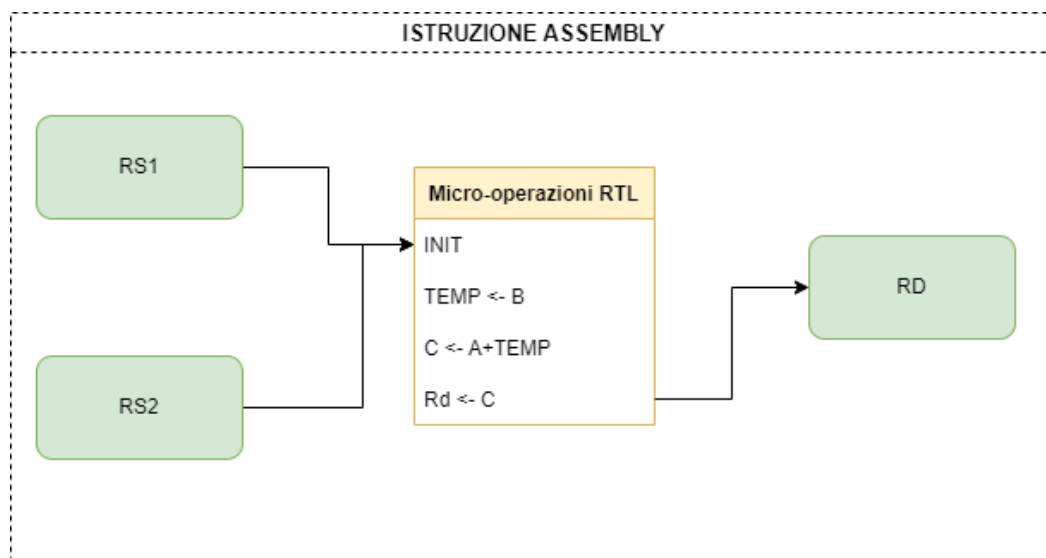


Figura 79: Schema logico del flusso di una istruzione Assembly, Rs1 Rs2 e Rd sono registri appartenenti al Register File

Definire un’istruzione equivale a specificare l’indirizzo dei registri sorgente e destinazione in aggiunta all’insieme di tutte le MICROISTRUZIONI (e quindi dei restanti segnali di controllo) eseguite in maniera ordinata nel tempo.

Esiste una moltitudine di tipi diversi di istruzione capaci di compiere ad esempio operazioni di calcolo e gestione della memoria, e il linguaggio impiegato per definirle è quello Assembly. L'astrazione consentita dal concetto di istruzione ne determina il passaggio da componente HARDWARE (come le Micro-operazioni) a SOFTWARE.

Nel capitolo 3.11 verrà esaminato nel dettaglio il funzionamento e la sintassi di una istruzione ASSEMBLY, dopo aver mostrato un esempio pratico di esecuzione di Microistruzioni nel Data Path.

3.9-Costruzione del Data Path e visualizzazione dell'esecuzione sequenziale di Microistruzioni

All'interno di questo paragrafo verrà mostrato un esempio di esecuzione di una serie di microistruzioni attraverso il Data Path realizzato nel simulatore Logisim. Il risultato dell'esecuzione sequenziale di queste microistruzioni è lo svolgimento di una istruzione di Somma tra registri del Register File chiamata ADD. Come si può notare dalla *figura 80*, la struttura del circuito realizzato è identica a quella mostrata in *figura 73*. In Logisim vengono mostrati anche i segnali di controllo, ovvero i Write Enable e Output Enable, rappresentati da tutti i pin raggruppati in verticale sul lato sinistro del circuito. Sarà proprio abilitando o meno questi bit che verrà analizzata l'esecuzione dell'istruzione ADD.

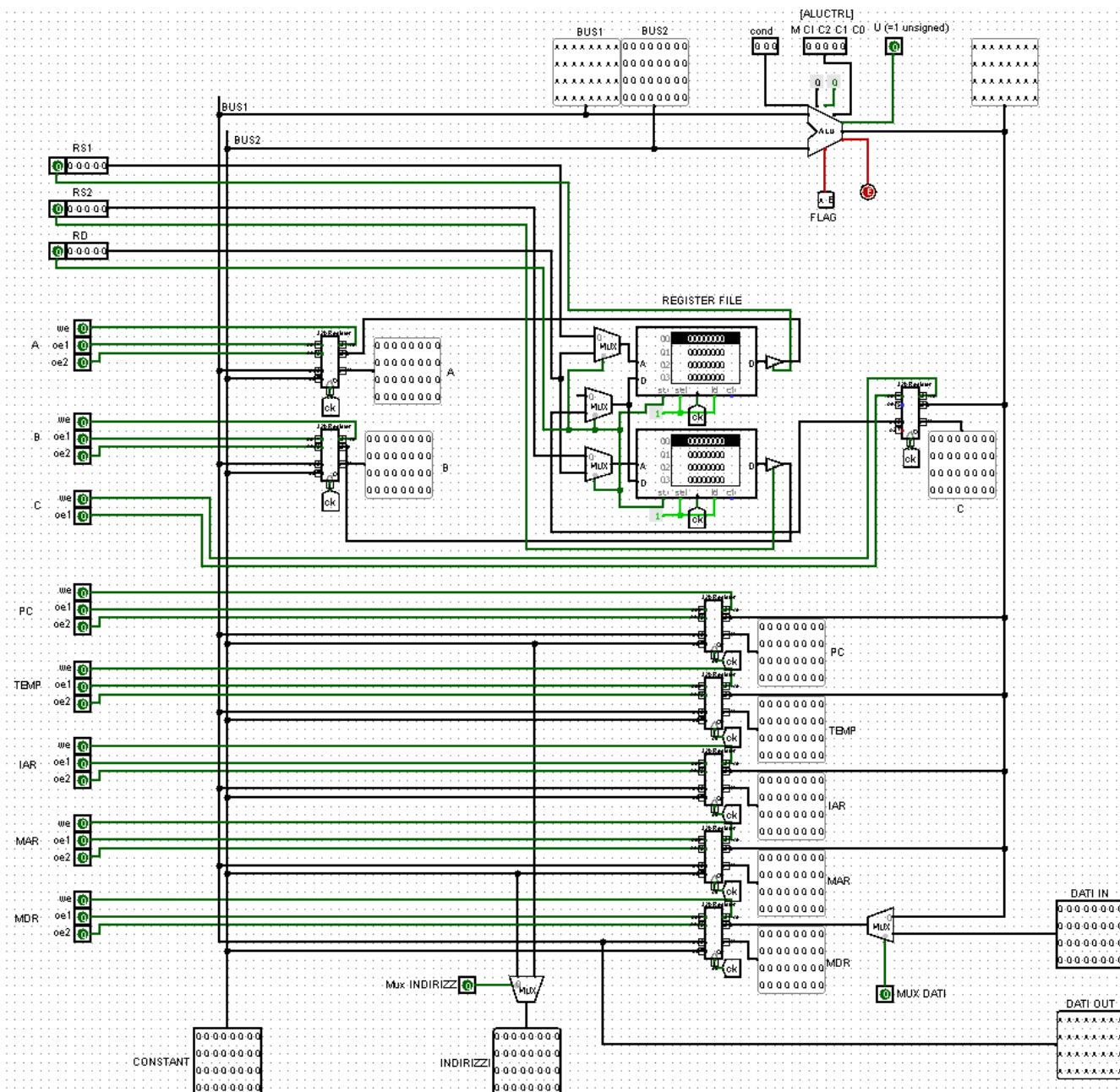


Figura 80: Schema circuitale della rete logica del Data Path costruito e visualizzato mediante Logisim

Sulla carta, per rappresentare il flusso di microistruzioni RTL di una istruzione ASSEMBLY risulta comodo usare la rappresentazione con metodo grafico chiamato *diagramma di flusso*. Un esempio di diagramma di flusso per l'istruzione di ADD viene mostrato in *figura 81*.

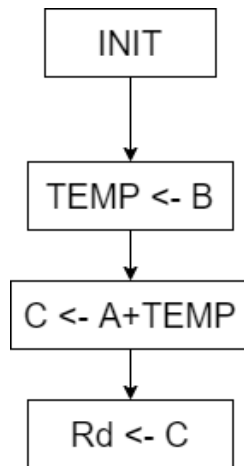


Figura 81: Diagramma di Flusso di una Istruzione Assembly, la ADD

I riquadri in figura rappresentano Microistruzioni scritte in codice RTL. Di conseguenza, ogni riquadro avrà durata di esattamente 1 CLOCK. La fase INIT dura 2 CLOCK e verrà discussa in seguito, essa è collegata alla “programmabilità” della macchina di Von Neumann. Al momento le microistruzioni che verranno visualizzate sono le ultime 3, ovvero quelle necessarie per portare a termine una operazione di Somma tra registri.

Si supponga ora di voler sommare il contenuto dei registri del Register File R1 e R2 e di voler salvare il risultato su R3. In linguaggio Assembly tale comportamento viene descritto dalla sintassi:

ADD R3 R1 R2

Si supponga inoltre che, al momento dell’inizio dell’esecuzione dell’istruzione, il registro R1 contenga il valore 15h mentre R2 sia pari a 5h. La situazione descritta del Register File viene mostrata in figura 82.

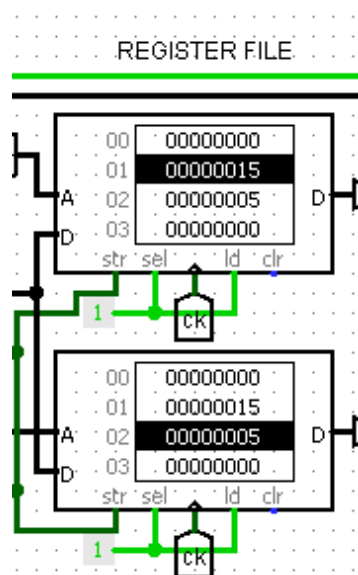


Figura 82: Stato dei Registri sorgente del Register File R1 e R2 all’inizio dell’istruzione

Prima di iniziare il calcolo della somma vera e propria è necessario caricare il contenuto dei registri R1 e R2 rispettivamente in A e B, poichè il Register File non è direttamente collegato all'ALU. Questa operazione fa parte della fase di INIT e viene eseguita per ogni istruzione. Spostare il contenuto di R1 in A equivale a selezionare l'indirizzo di R1 e l'OE1 del RF e in contemporanea alzare il WE di A, stesso discorso per R2 e B, come mostrato in *figura 83*.

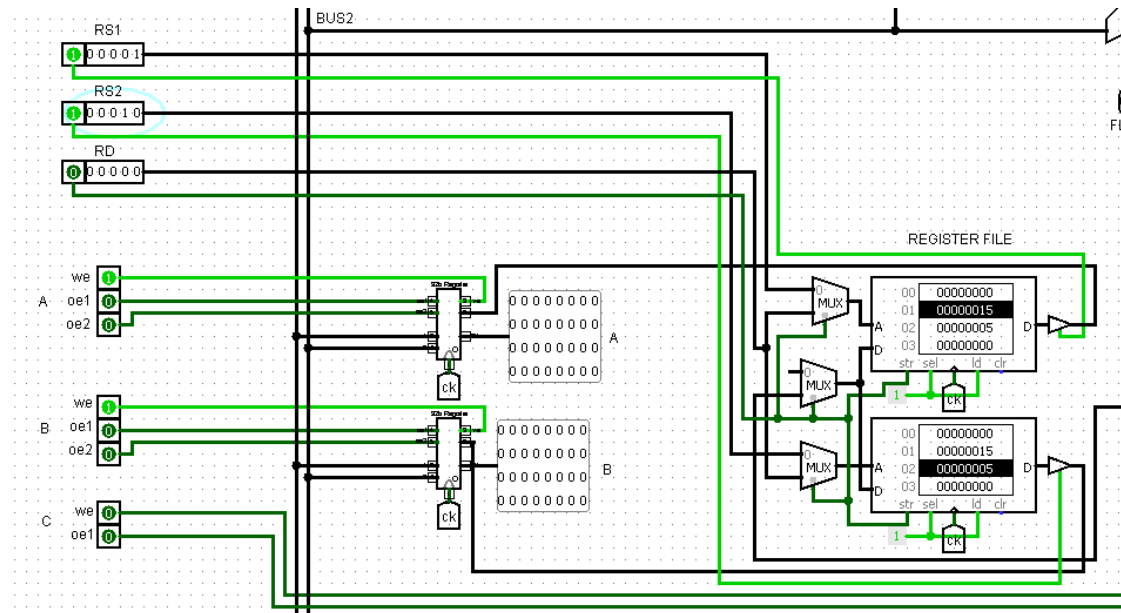


Figura 83: Visualizzazione dei segnali di Enable nel Data Path necessari al trasferimento degli operandi sorgente in A e B

Dalla *figura 83* si può notare che il contenuto di A e B non è cambiato. Questo perché ancora non è partito il colpo di Clock e i flip flop non hanno ancora campionato. Al successivo fronte di salita quindi si può vedere come il contenuto di A si aggiorni col valore 15h, mentre B diventa 5h, come mostrato in *figura 84*.

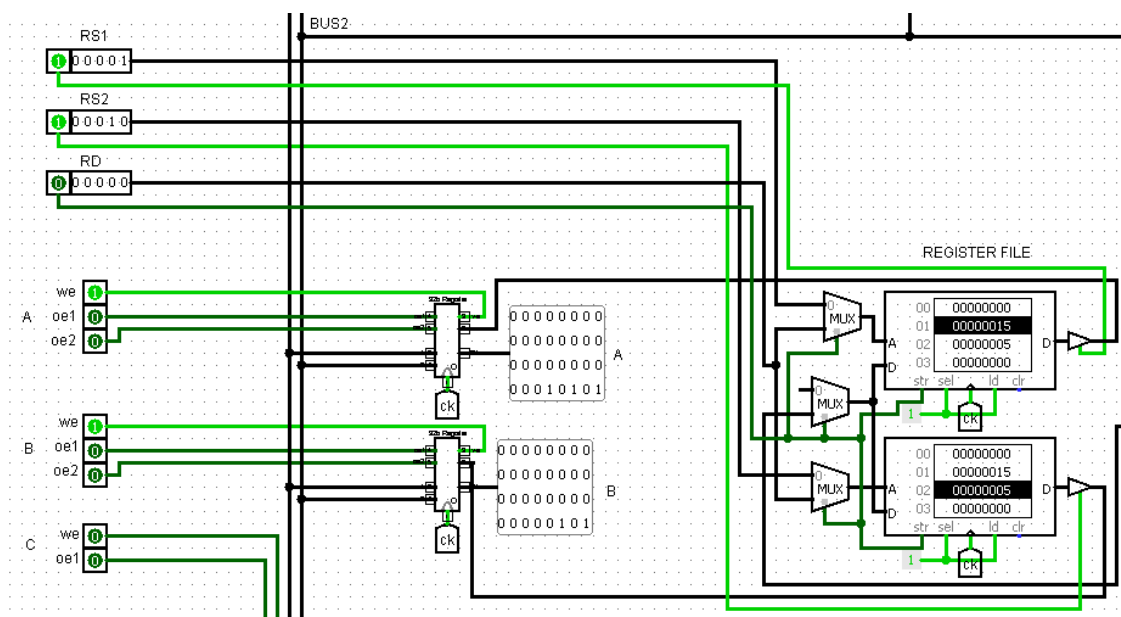


Figura 84: Trasferimento degli operandi sorgente nei registri A e B al fronte di salita

A seguire inizia la fase dell'istruzione chiamata di EXECUTE. Durante questa fase viene effettuata l'operazione di Somma vera e propria. La prima microistruzione della fase di EXECUTE consiste nel trasferire il contenuto del registro B nel registro di appoggio TEMP. Sarà quindi alzato l'OE di B e il WE di TEMP. La ALU va configurata con bit di Modalità M=1, CI=0, [C2 C1 C0]=[0 1 0] per eseguire l'operazione $Y=X$, in modo tale da trasferire il dato puro, come mostrato in figura 85.

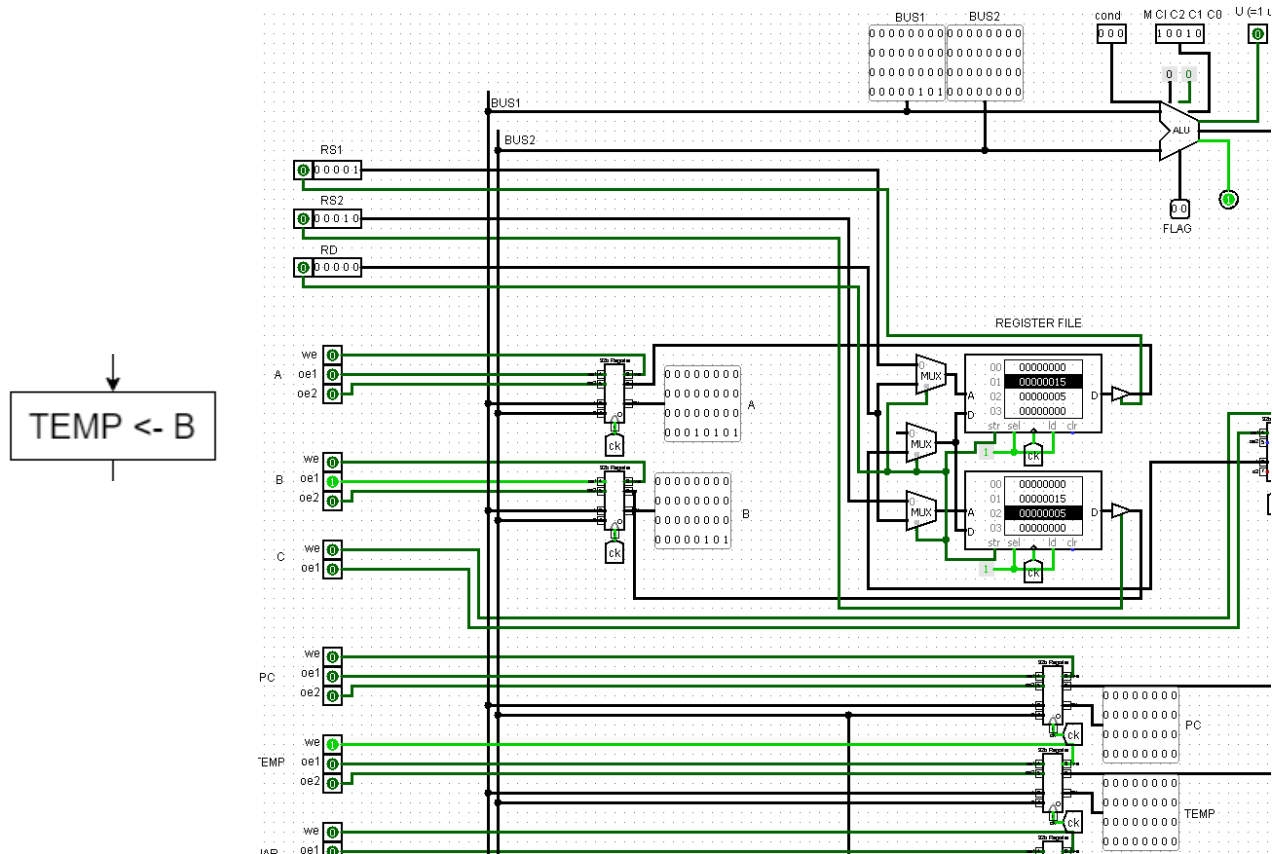


Figura 85: Segnali di Enable del Data Path necessari al trasferimento di B in TEMP

Al successivo colpo di clock il contenuto di B verrà trasferito in TEMP, che prenderà il valore 5h, come mostrato in figura 86.

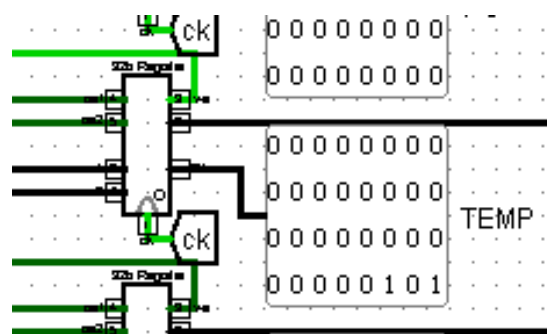


Figura 86: Zoom sul registro TEMP dopo il fronte di Salita

Nella seconda microistruzione della fase di EXECUTE viene eseguita la somma tra i contenuti del registro TEMP e A. Il risultato andrà messo in C. A tal fine è necessario alzare l'OE1 di A e l'OE2 di TEMP in contemporanea al WE di C. La ALU va configurata in modalità Somma, quindi M=1, CI=0, [C2 C1 C0]=[0 0 0], come mostrato in figura 87.

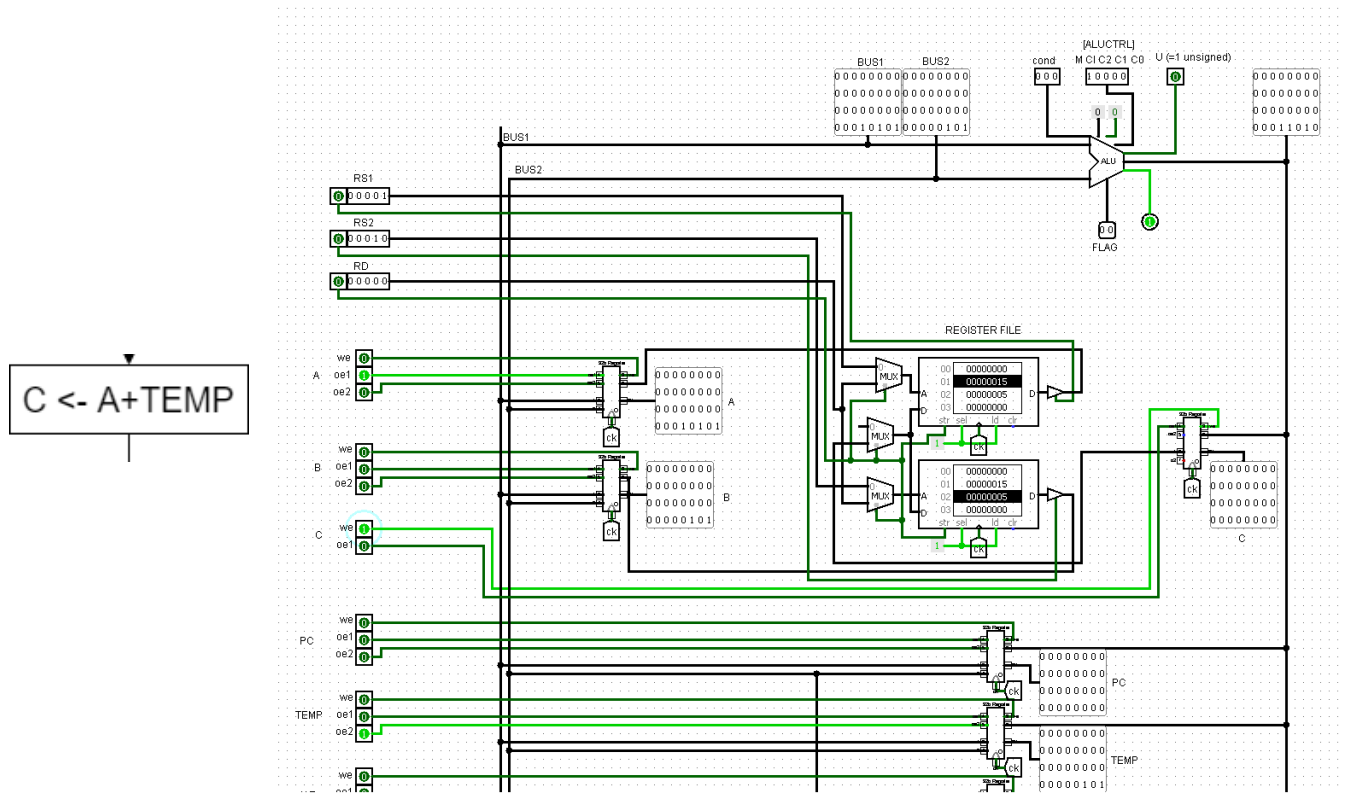


Figura 87: Segnali di Enable necessari al trasferimento della somma di A e TEMP in C

Al colpo di clock successivo in C verrà trasferita la somma di A e TEMP, ovvero 1ah, come mostrato in figura 88.

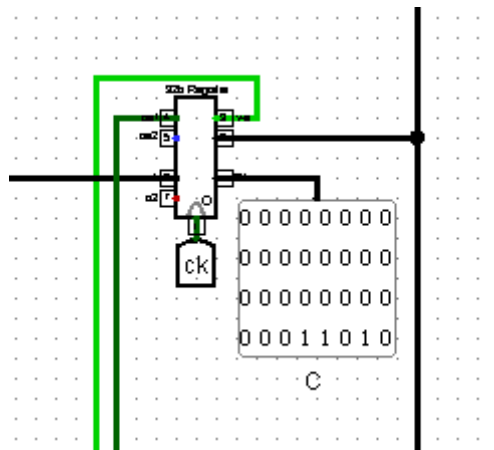


Figura 88: Zoom sul registro C dopo il fronte di salita

Infine, è necessario salvare il valore di C nel registro R3 del Register File, come specificato dall'istruzione ASSEMBLY. Tale fase viene chiamata di WRITE BACK e per effettuarla è sufficiente selezionare l'indirizzo corrispondente del registro destinazione del Register File e abilitarne il WE, e in contemporanea alzare l'OE di C, come mostrato in *figura 89*.

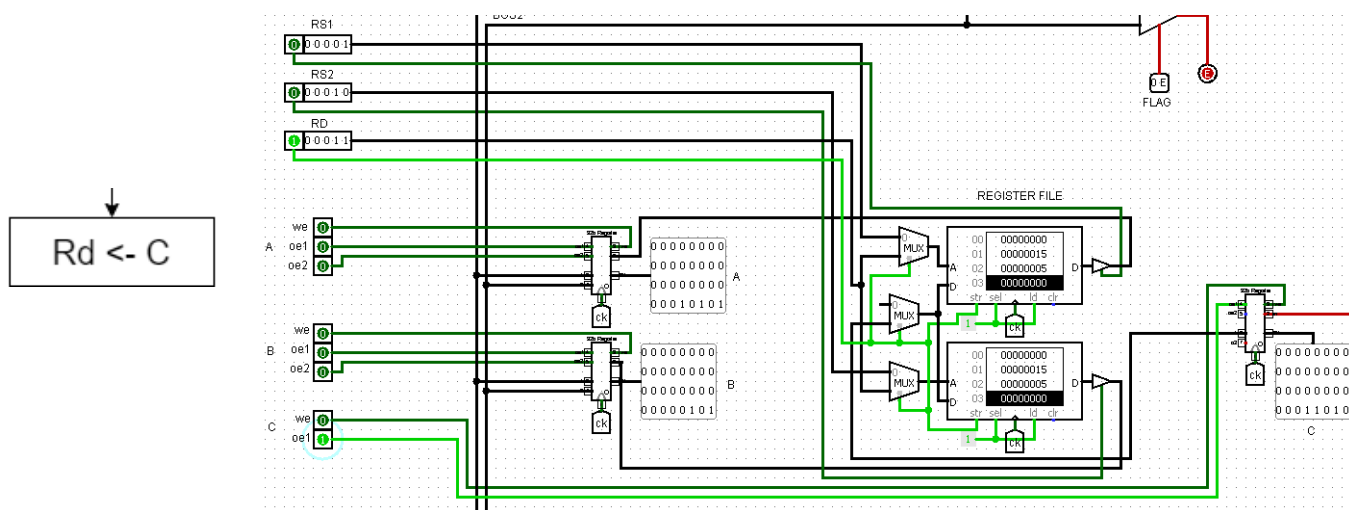


Figura 89: Segnali di Enable necessari al trasferimento di C nel registro destinazione Rd, ovvero fase di Write Back

Al successivo colpo di clock lo stato dei registri è quello atteso, quindi in R3 è ora presente il valore della somma tra il contenuto dei registri R1 e R2, come mostrato in *figura 90*.

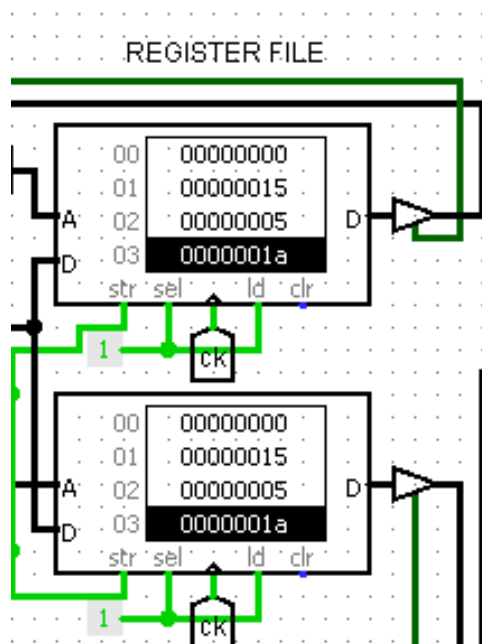


Figura 90: Stato dei registri del Register File al termine dell'istruzione

L'esecuzione dell'istruzione a questo punto è completa e in output si è ottenuto il valore desiderato.

3.10- Mappatura dei segnali di controllo

La visualizzazione delle Micro-operazioni direttamente nel Data Path fornisce un punto di vista molto interessante sulle Microistruzioni RTL. Nel codice binario, una stringa di bit che rappresenta un numero decimale contiene due tipi di informazione: il valore dei singoli bit e la posizione degli stessi. La posizione dei bit ne determina infatti il peso e insieme al valore dei bit consente di calcolare il valore decimale della stringa di bit.

Si prenda ora lo schema circuitale del Data Path e si supponga di ruotarlo di 90°, come mostrato in *figura 91*.

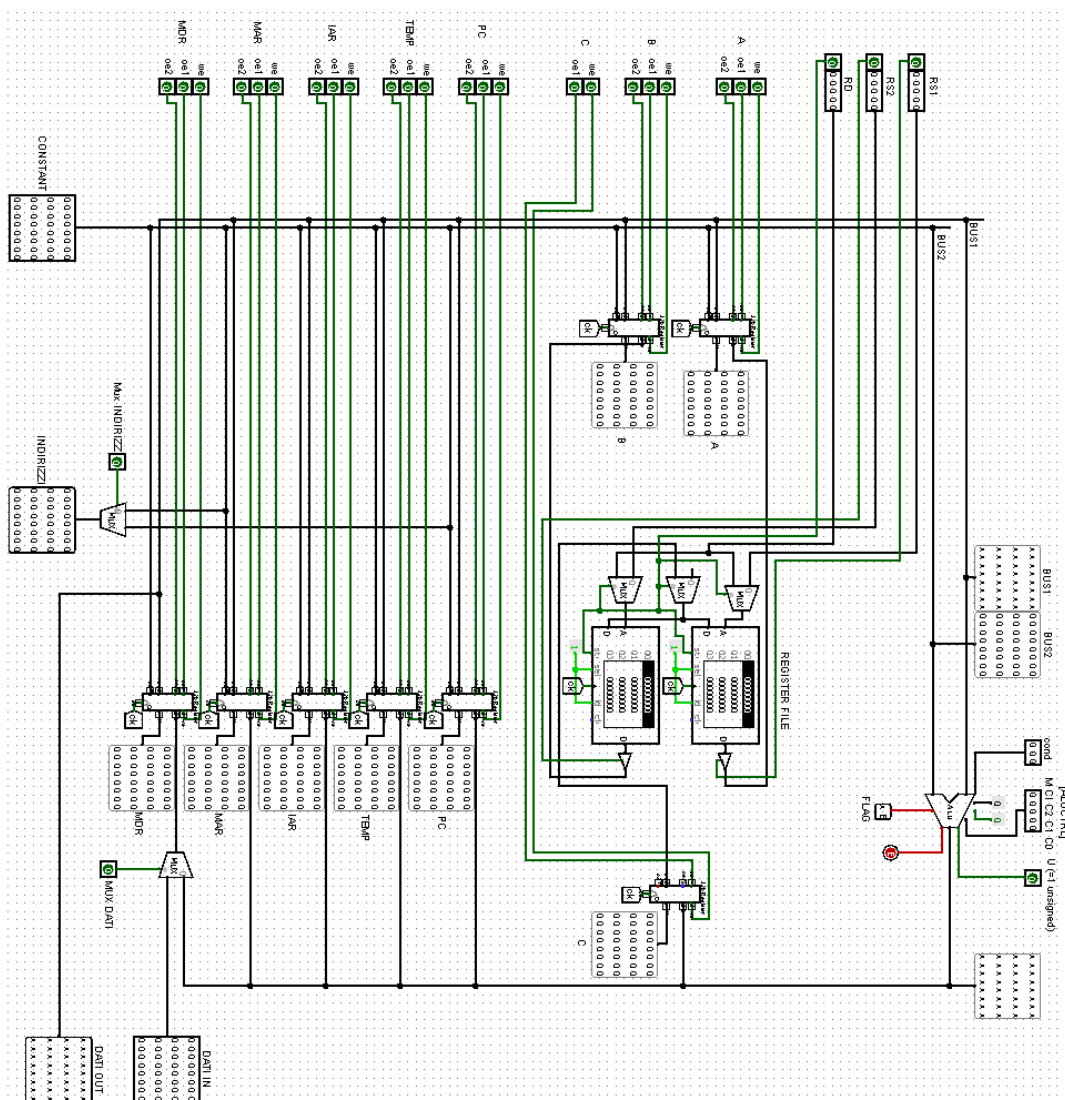


Figura 91: Schema circuitale del Data Path ruotato di 90° in senso orario

Tralasciando tutto ciò che non è strettamente un segnale di controllo, rimane la colonna che in *figura 80* era in verticale (dopo la rotazione risulterà orizzontale, come mostrato in *figura 92*) contenente tutti i bit dei segnali di controllo del Data Path.

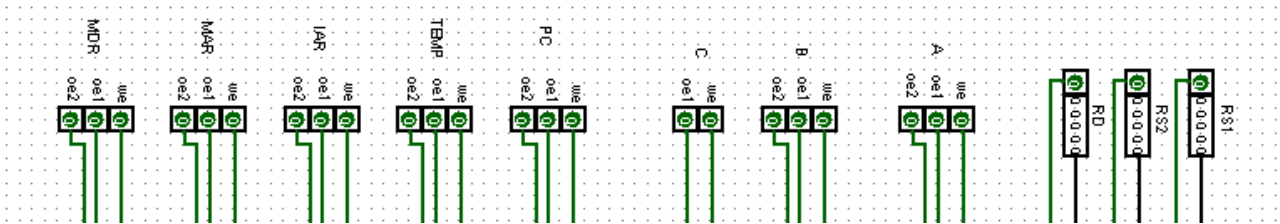


Figura 92: Visualizzazione orizzontale dei Segnali di Enable del Data Path, detti anche Segnali di Controllo

Tale colonna è una stringa di bit (come nell'esempio precedente del numero decimale), con la differenza che in questo caso la posizione non determina il peso del bit, bensì il suo significato. Il primo bit a partire da destra, infatti, non ha il valore di $2^0 = 1$, esso indica invece l'Output Enable del Registro Sorgente 1 del Register File.

Si prenda ad esempio nuovamente il caso dell'istruzione ADD. La prima microistruzione della fase di EXECUTE è $TEMP \leftarrow B$. Osservando lo stato dei bit dei segnali di controllo prima di effettuare il colpo di clock si ottiene la situazione mostrata in *figura 93*.

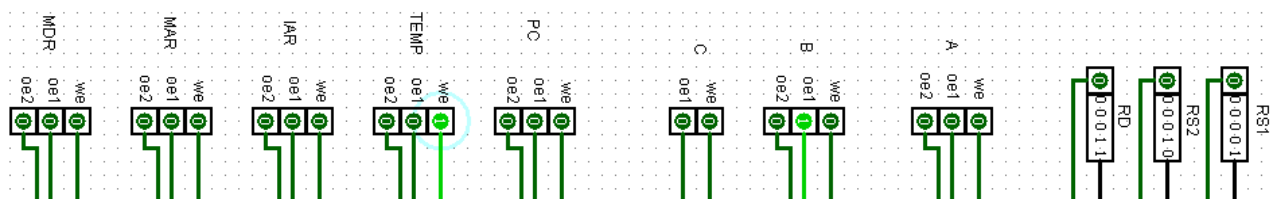
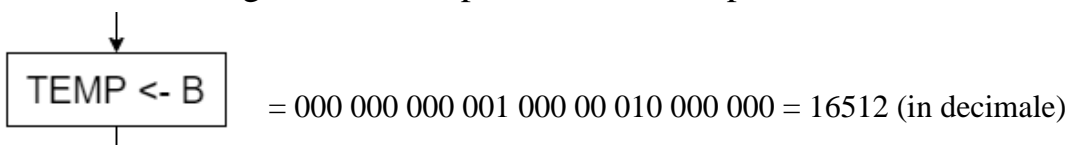


Figura 93: Stato dei Segnali di Enable del Data Path necessari all'esecuzione della Microistruzione $TEMP \leftarrow B$

Annotando i valori dei bit mostrati in figura si ottiene la seguente stringa binaria (MSB è l'OE2 di MDR): 000 000 000 001 000 00 010 000 000 (tralasciando gli indirizzi del Register File).

Una singola microistruzione RTL quindi si rappresenta attraverso una stringa di bit, prodotta a ogni colpo di clock, con un valore logico e quindi posizione ben definiti; cablando nel modo corretto questi bits nel Data Path è possibile vederne l'effetto, che consiste nell'eseguire il micropostamento corrispondente.



Il cablaggio dei bit della stringa quindi è ciò che ne determina l'effetto finale sul Data Path. Modificando i collegamenti tra stringa di bit ed enable dei registri infatti si cambia totalmente il significato della microistruzione, e questo è il motivo per cui il

codice RTL è così importante: architetture diverse avranno cablaggi diversi e quindi stringhe di bit diverse, ma l'effetto è sempre il medesimo, ovvero lo stesso microspostamento, espresso dallo stesso codice RTL. Mentre l'Architettura cambia, il significato del codice RTL rimane sempre invariato.

Per concludere, nonostante la stringa di bit non abbia significato aritmetico, bensì logico, è possibile comunque assegnare un valore decimale all'insieme dei bits: il primo 1 a sinistra ha peso pari a 2^{14} , mentre il secondo vale 2^7 . Il valore decimale della stringa è quindi pari a $2^{14}+2^7=16512$.

Con una serie di numeri provvisti di significato è quindi possibile sintetizzare tutte le informazioni viste fino nei paragrafi 3.6 e 3.7.

L'intera mappatura dei segnali di controllo del Data Path del sistema Virtual Shock nella stringa di bit, compresi segnali di controllo per ALU e MUX, viene mostrata in *figura 94*, dove per comodità visiva i segnali sono rappresentati in una matrice di bit, e in *figura 95*, dove vengono invece mostrati i restanti segnali in orizzontale, più altri segnali che verranno introdotti in seguito.

31	30	29	28	27	26	25	24
RIS	MI	MD	OE2	OE1	WR	OE2	OE1
MUX			MDR			MAR	
23	22	21	20	19	18	17	16
WR	OE2	OE1	WR	OE2	OE1	WR	OE2
IAR				TEMP		PC	
15	14	13	12	11	10	9	8
OE1	WR	OE2	OE1	WR	OE2	OE1	WR
C				B			
7	6	5	4	3	2	1	0
OE2	OE1	WR	M	CI	C2	C1	C0
A				ALU CTRL			

Figura 94: Mappatura dei segnali di controllo del Data Path

16	15	14	13	12	11	10	9	8
SET	BR	MUX	U	CO2	CO1	CO0	RX	RY
	7	6	5	4	3	2	1	0
	RZ	CEN	IOWR	IORD	MEMWR	MEMRD	IEN	WEIEN

Figura 95: Mappatura dei segnali di controllo del Data Path

In totalità sono presenti 49 segnali di controllo.

3.11-UDC: Rete Sequenziale

La rete di elaborazione del Data Path è completa e funzionante anche da sè, ed è in grado di eseguire operazioni complesse controllata dai segnali di controllo. Un Data Path da solo però non è una rete molto pratica. Concretamente è come avere una calcolatrice molto difficile da usare, dove per eseguire anche solo una somma sono necessari diversi passaggi macchinosi. Un essere umano non è in grado di usare efficientemente tale rete per fini pratici. È necessario che a pilotare tale rete ci sia una nuova rete di “controllo” che riesca a emettere i relativi segnali di controllo nell’ordine giusto per portare a termine istruzioni molto rapidamente e sfruttare la potenza di calcolo del Data Path, senza la necessità di un aiuto umano. Se la rete di controllo è molto veloce infatti, non importa che eseguire una somma sia una operazione macchinosa: verrà comunque eseguita più velocemente di un essere umano.

La possibilità di funzionare senza interventi esterni è proprio una delle caratteristiche principali di una Macchina di Von Neumann. La rete logica UDC, ovvero l’ Unità di Controllo, renderà automatico questo procedimento. L’UDC produrrà gli ENABLE in maniera sequenziale, sostituendosi al lavoro che altrimenti sarebbe costretto a fare un essere umano.

Nella pratica, in ingresso l’UDC deve prendere una ISTRUZIONE ASSEMBLY, come la ADD realizzata nell’ultimo esempio del paragrafo 3.9, e in uscita deve produrre una serie di stringhe di bit sequenzialmente nel tempo, che altro non sono che i segnali di controllo che codificano le Microistruzioni necessarie a portare a termine quella precisa istruzione. Il comportamento dell’UDC va quindi analizzato nel tempo: una istruzione infatti richiede diversi colpi di clock per essere completata. Questo concetto viene rappresentato dallo schema di flusso in *figura 96*:

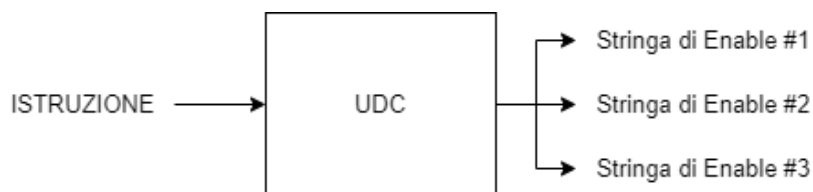


Figura 96: Schema logico del comportamento nel tempo dell’UDC

Come anticipato in precedenza, una ISTRUZIONE viene interamente definita dagli enable dei registri sorgente e destinazione del Register File e dagli enable del data path, ma questa in realtà è solo la definizione di come ESEGUIRE una istruzione.

Allo stesso modo del codice RTL, che codifica uno SPOSTAMENTO HARDWARE tra registri con una stringa di ENABLE (es. 000100011000) a cui viene associato un NOME (es. `temb <- B`), ASSEMBLY codifica una SERIE di spostamenti Hardware con una stringa di bit alla quale viene associato un nome ovvero il NOME dell'istruzione (es. ADD vista prima). Di conseguenza, sempre prendendo il caso della ADD, il nome serve all'occhio umano per riconoscere il tipo di operazione, che verrà poi codificata in UNA UNICA stringa di bit, che verranno poi tradotti dall'UDC in una serie di Micro-operazioni sequenziali.

La visualizzazione del concetto di nome, codifica ed enable viene mostrata in *figura 97*, ampliando la precedente figura. Si ricorda che i bit mostrati sono in parallelo.

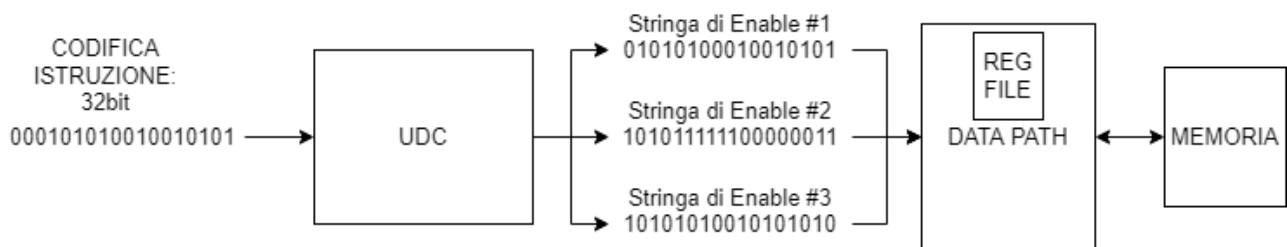


Figura 97: Schema logico del comportamento nel tempo dell'UDC con visualizzazione dei segnali in ingresso e uscita

Focalizzandosi ora sull'UDC, si può notare come una rete in grado di leggere un input e produrre una uscita dipendente dal tempo sia proprio una Rete Sequenziale, per intendersi simile alla rete dei Flip Flop, dove ad ogni stato viene mandata in uscita la stringa di bit di segnali di controllo mostrata nel paragrafo precedente. Una rappresentazione della Rete Sequenziale dell'UDC viene mostrata in *figura 98*.

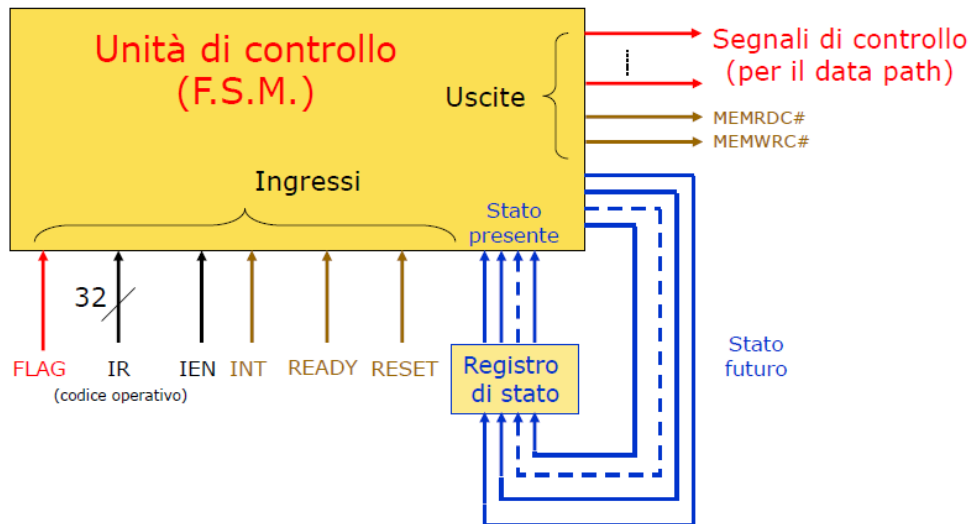


Figura 98: Schema Logico di una UDC realizzata mediante Rete Sequenziale (tratto dalle Slides del corso di Calcolatori Elettronici – A.A 2019)

Una Rete Sequenziale è definita da un ingresso, degli stati, una relazione che lega il passaggio da uno stato all'altro e le uscite che vanno accese ad ogni stato. Mediante l'uso del Diagramma degli Stati è possibile rappresentare in maniera astratta questo concetto. Ad esempio nel caso di un riconoscitore di sequenze, in ingresso la rete avrà un unico bit che però cambierà nel tempo, e l'uscita dovrà andare a 1 quando la forma d'onda dell'ingresso corrisponde alla sequenza da riconoscere. Il Diagramma degli Stati di tale rete viene mostrato in *figura 99*.

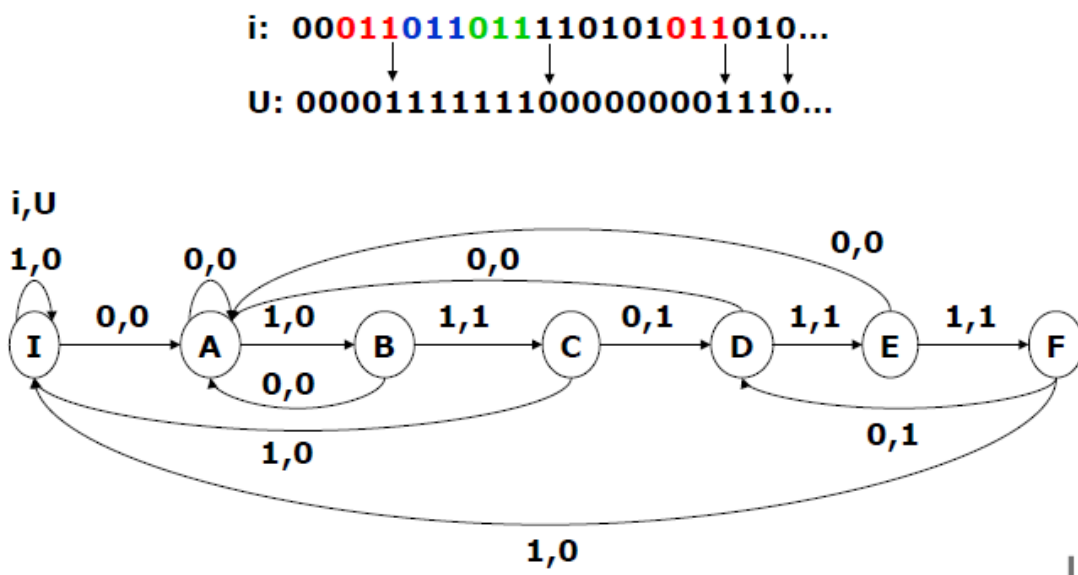


Figura 99: Diagramma degli Stati di un Riconoscitore di Sequenze

Una rete simile è abbastanza semplice: in ingresso e in uscita si ha un unico bit, e gli stati sono sette. Nulla impedisce però la realizzazione di reti molto più complesse, con diversi bit in ingresso e uscita e molti più stati.

Si supponga ad esempio di voler realizzare una Rete Sequenziale in grado di automatizzare lo svolgimento di un'unica istruzione: la ADD, che consiste nel sommare due registri sorgente del Register File, per poi caricare il risultato nel registro destinazione. L'obiettivo consiste quindi nel produrre ad ogni stato della rete una uscita costituita da 49 bit, ovvero tutti i segnali di Controllo del Data Path, Register File Compreso. Se si considera la traduzione in enable di ogni singolo blocco del diagramma di flusso delle microistruzioni come il valore che deve prendere l'uscita della rete sequenziale durante un determinato stato, è possibile associare il diagramma di flusso con il diagramma degli stati come mostrato in *figura 100*.

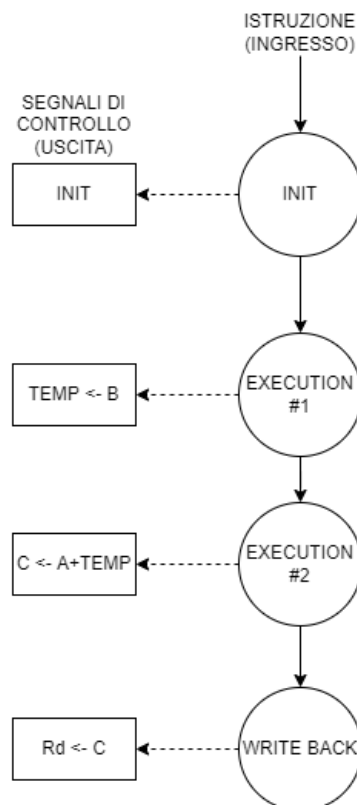


Figura 100: Diagramma degli Stati della Rete Sequenziale usata per implementare l'esecuzione delle Microistruzioni di una ADD

Come nel caso del riconoscitore di sequenze, l'uscita sarà solo un po' più complessa, ed inoltre, si hanno meno stati. Traducendo ogni microistruzione RTL in una stringa di ENABLE, si ottiene quindi la vera e propria uscita in bit della rete sequenziale ad

ogni stato, come mostrato in *figura 101*. Questa volta lo schema viene mostrato orizzontalmente per far risaltare la somiglianza con il riconoscitore di sequenze.

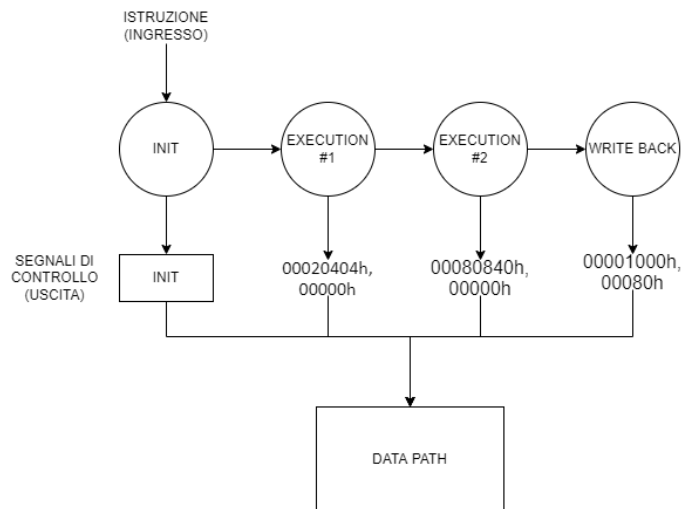


Figura 101: Visualizzazione orizzontale della rete con uscita rappresentata mediante stringhe di Bit collegate al Data Path

Come anticipato gli enable andranno a pilotare le Micro-operazioni del Data Path, motivo per quale il Data Path è stato incluso nella figura. Fino ad adesso è stato definito il comportamento delle uscite della rete sequenziale (secondo microistruzioni sequenziali da eseguire in ordine una alla volta), rimane da analizzare come possa essere gestito l'ingresso della rete. Come anticipato, l'ingresso della rete sequenziale UDC è la STRINGA DI 32bit corrispondente alla codifica associata a una ISTRUZIONE ASSEMBLY, in questo specifico caso di una ADD. Supponendo ad esempio di codificare una ADD con una stringa di 32 zeri, la rete sequenziale non deve fare nulla se vede in ingresso qualsiasi numero diverso da zero, mentre deve procedere allo stato successivo nel momento in cui riconosce il codice dell'istruzione.

Una volta quindi definito e compreso il funzionamento di tale rete, è possibile costruirla con una rete di AND, OR, NOT gates e un registro, usando il classico metodo delle mappe di Karnaugh. Fin qui la progettazione dell'UDC mediante una Rete Sequenziale sembra fattibile. Si ricorda però che tale rete permette di automatizzare un'unica istruzione: la ADD. Il codice Assembly comprende una moltitudine di istruzioni dalle diverse funzioni, la rete Sequenziale dell'UDC quindi deve essere ampliata per gestire anche microistruzioni associate a istruzioni diverse.

Si supponga ad esempio di voler aggiungere una istruzione alla rete dell'UDC: la SUB, ovvero una sottrazione tra registri sorgente. Nel diagramma degli stati questo equivale ad aggiungere un altro "ramo" con gli stati e le uscite necessari a eseguire la nuova istruzione. La fase di INIT quindi non serve più a decidere se eseguire o meno una istruzione, bensì a QUALE istruzione eseguire. L'ampliamento del diagramma degli stati viene mostrato in *figura 102*.

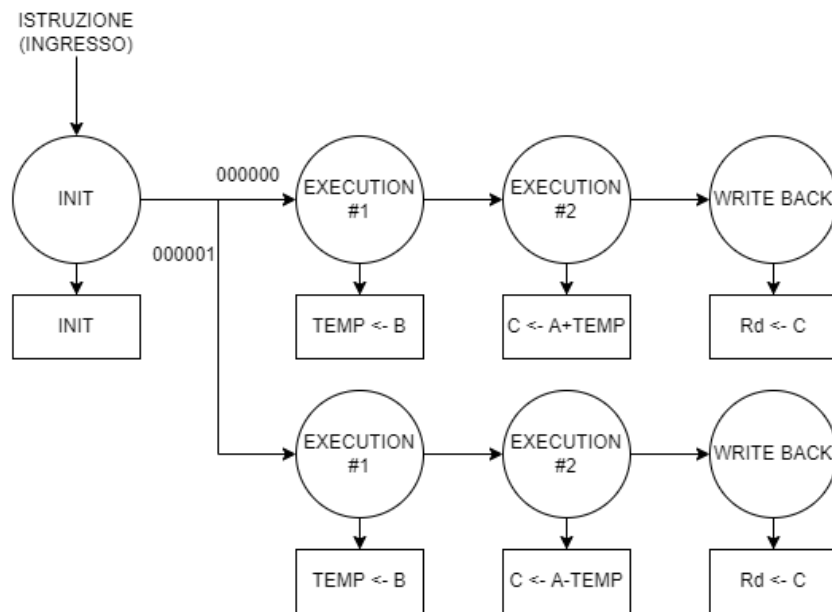


Figura 102: Aggiunta di un Ramo al Diagramma degli Stati della Rete Sequenziale dell'UDC

Come si può notare dalla figura, le due istruzioni condividono la fase di INIT e i “nomi” delle fasi successive. Il funzionamento è simile, ma nel ramo in basso appartenente alla SUB si può notare come la seconda microistruzione di EXE sia una sottrazione tra registri: $C \leftarrow A - TEMP$. Nella fase di INIT la rete si valuta se in ingresso ci sono una serie di zeri. Nel caso in cui ci fossero si procede ad eseguire la prima EXE della ADD, altrimenti cambia ramo ed esegue la prima EXE della SUB.

Questa operazione può essere ripetuta per tutti i tipi di istruzione ASSEMBLY che si desidera aggiungere alla rete, e lo schema mostrato in *figura 103* visualizza in maniera astratta lo schema degli stati di una rete Sequenziale in grado di eseguire tutte le istruzioni necessarie per il funzionamento della Macchina.

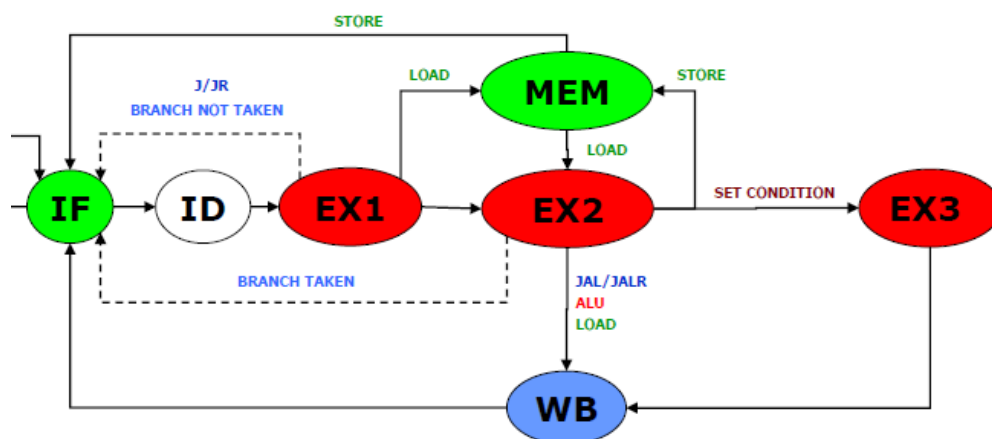


Figura 103: Diagramma degli Stati di una UDC realizzata mediante Rete Sequenziale (tratto dalle Slides del corso di Calcolatori Elettronici – A.A 2019)

La codifica e struttura delle istruzioni Assembly verrà spiegata nel dettaglio nel prossimo paragrafo.

3.12-Instruction Set Architecture e DLX

Una istruzione viene definita dal suo nome e dagli indirizzi dei registri Sorgente e Destinazione del Register File (*figura 79*). Bisogna quindi trovare un modo per utilizzare i 32 bit a disposizione per la codifica dell'istruzione per "comunicare" queste informazioni in ingresso all'UDC, in modo che possa iniziare l'esecuzione automatica dell'istruzione desiderata, dopo aver effettuato la decisione nella fase di INIT. Nell'ISA del DLX una istruzione ASSEMBLY viene codificata come mostrato in *figura 104*.

	IR[31..26]	IR[25..21]	IR[20..16]	IR[15..11]	IR[10..0]
Tipo	6 bit	5 bit	5 bit	5 bit	11bit
R	op-cod	S1	S2	D	ext-op-cod
I	op-cod	S1	S2	operando immediato (16 bit)	
J	op-cod	offset (26 bit)			

Figura 104: Codifica ISA di una istruzione Assembly

Esistono tre tipi di istruzione: R, I e J, ma tutti condividono lo stesso campo sulla sinistra: il COP. Formato da 6Bit, il codice op-cod, detto anche COP è il "nome" binario dell'istruzione e consente di distinguere una istruzione dalle altre. Il COP quindi altro non è che l'INGRESSO della rete sequenziale dell'UDC: grazie a esso infatti la rete è in grado di riconoscere l'istruzione e scegliere il ramo giusto del diagramma degli stati per eseguire le microistruzioni corrispondenti. Tale concetto viene illustrato in *figura 105*.

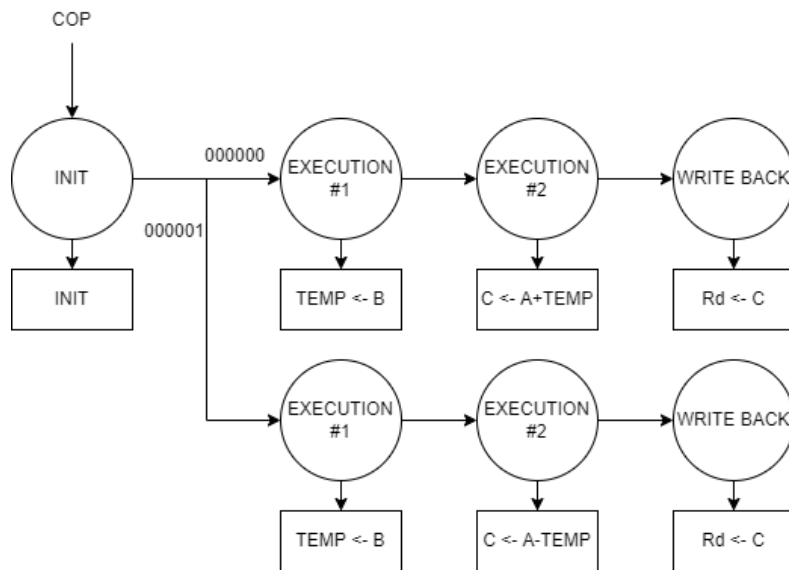


Figura 105: Diagramma degli Stati dell'UDC a Rete Sequenziale con COP in ingresso

La scelta di associare un determinato COP a una determinata istruzione è totalmente arbitraria, a patto di non usare lo stesso COP per due istruzioni diverse. Nel caso della *figura 105* alla ADD è stato associato il COP 000000, mentre alla SUB il COP 000001.

Per quanto riguarda invece i restanti 26 bit dell'istruzione, in essi è contenuta l'informazione degli indirizzi dei registri sorgente e destinazione dell'istruzione (si ricorda che il Register File è come una memoria, vanno specificati gli indirizzi dei registri e non il contenuto). Si può notare come i tipi di istruzione vengano distinti proprio in base a questi campi: il tipo R o REGISTER è quello a cui appartiene la ADD e prevede due registri sorgente e uno destinazione, il tipo I o IMMEDIATE invece consente la possibilità di sostituire all'indirizzo di uno dei due registri sorgente una costante scritta direttamente nel codice dell'istruzione (come se si scrivesse direttamente il contenuto del registro invece che il suo indirizzo), per ultimo il tipo J o JUMP non prevede registri sorgente bensì una unica costante. L'UDC prevede circuiteria aggiuntiva per interpretare nella maniera corretta la stringa di bit contenente gli indirizzi dei registri in base al tipo di istruzione.

Oltre che al tipo di operando, le istruzioni ASSEMBLY vengono divise in base alla loro FUNZIONE. Esistono 4 tipi principali di istruzione ASSEMBLY:

- ALU: qualsiasi istruzione che consenta di portare a termine un'operazione aritmetica o logica mediante l'uso della ALU, come la ADD
- LOAD/STORE: consentono di effettuare spostamenti in/da memoria
- BRANCH: consentono di modificare il flusso del programma mediante valutazione di determinate condizioni (simile a un IF)

- JUMP: consentono di effettuare salti nell'esecuzione del programma in qualsiasi caso, senza valutare condizioni

Ogni istruzione va quindi codificata in una stringa di bit seguendo un criterio preciso partendo dal nome "umano" e dalla funzione dell'istruzione. Nel caso della ADD ad esempio, sommare i registri sorgente r1 e r2 per poi salvarne il risultato in r3 equivale a scrivere: add r3 r1 r2, dove possono essere individuati nome e registri. Come anticipato la ADD ha cop 000000, l'indirizzo di r1 è pari a 00001, r2 invece è 00010 mentre r3 è pari a 00011. Ricostruendo i campi quindi la stringa di bit associata a questa istruzione è: 000000 00001 00010 00011 0000000000. Tradurre una istruzione Assembly in una stringa di bit equivale a scrivere quello che viene chiamato "Codice Macchina", che è proprio il programma da caricare nella memoria del calcolatore, denominata Memoria DLX.

Mediante la codifica delle istruzioni ASSEMBLY e una rete sequenziale che possa eseguirle mediante il Data Path è possibile realizzare l'Automatizzazione di una Macchina di Von Neumann, ma è a questo punto che sorge un problema. La Rete Sequenziale presentata finora può essere costruita unicamente in maniera teorica: si può notare infatti come continuando ad aggiungere istruzioni la rete continui a crescere di complessità fino ad arrivare a un circuito ingestibile e soprattutto molto scomodo da modificare, essendo che bisogna ricalcolare tutta la rete di nuovo. È necessario quindi trovare un modo per realizzare lo stesso comportamento di una rete sequenziale, senza usare una rete sequenziale. Questo problema è stato risolto grazie all'aiuto del Professore Testoni, che ha suggerito di ricreare la funzione della Rete Sequenziale mediante una implementazione particolare detta Microcodice, consentendo al progetto Virtual Shock di continuare. Oltre alla sua semplicità, la particolarità di questa implementazione è la possibilità di osservarne il comportamento in maniera dettagliata attraverso le memorie di Logisim, come verrà presentato nel prossimo capitolo.

3.13-Implementazione tramite Microcodice

Si ritorni innanzitutto al flusso di microistruzioni di una istruzione ADD. Come già spiegato, ogni microistruzione corrisponde a una stringa di segnali di controllo da inviare al Data Path, la quale rappresentazione mediante diagramma di flusso viene mostrata in *figura 106*.

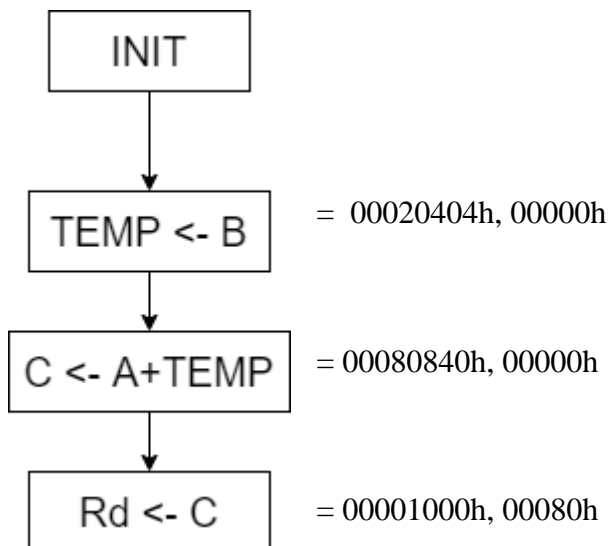


Figura 106: Diagramma di Flusso dell'istruzione ADD messo a confronto con i segnali di Enable che ogni riquadro rappresenta

Si ricorda inoltre che lo scopo dell'UDC con implementazione a microcodice rimane produrre le tre stringhe di bit sequenzialmente nello stesso ordine, una ad ogni colpo di clock, allo stesso modo della Rete Sequenziale introdotta nel paragrafo precedente.

Stringhe di bit, messe in colonna, in ordine secondo un criterio preciso... Questa situazione è perfettamente riproducibile con una memoria, che altro non è che un insieme di celle di bit posizionate in colonna secondo un ordine preciso, ovvero gli indirizzi.

Si consideri ad esempio la situazione seguente, dove le tre stringhe di bit delle microistruzioni della ADD sono state caricate su delle RAM di logisim nelle prime 3 celle, come mostrato in *figura 107*. Le RAM impiegate sono due poiché la massima bit-width di un componente in Logisim è 32Bit, ma in questo caso i segnali di controllo sono 49. In ogni caso le due RAM possono essere pensate come un'unica RAM con bit width di 49 bit.

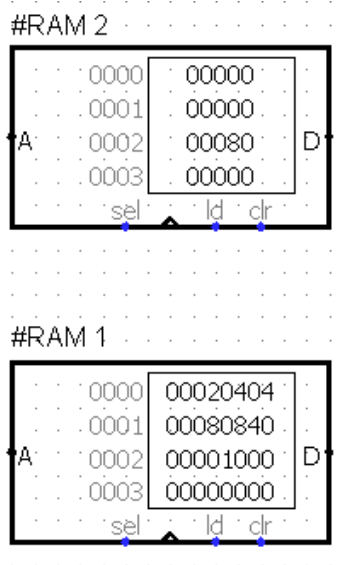


Figura 107: Memorie RAM Logisim con caricato il Microcodice dell'istruzione ADD

Si procede ad aggiungere tutti i collegamenti necessari per pilotare le memorie: input, output, clock e wires. Il risultato viene mostrato in figura 108.

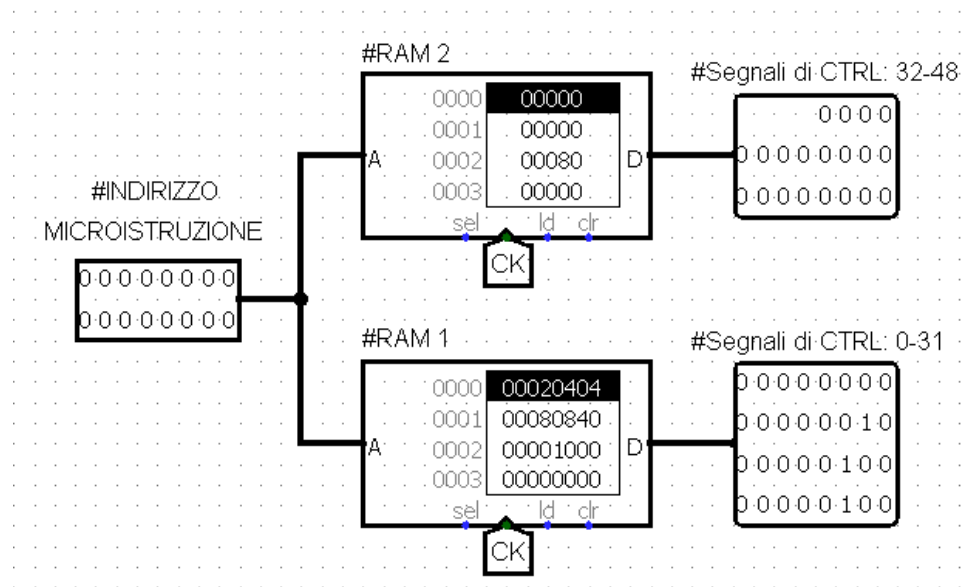


Figura 108: Collegamento delle RAM del Microcodice

A scopo di chiarezza, va tenuto presente che l'output di queste memorie va collegato ai pin dei segnali di controllo del Data Path, mostrati nella versione "orizzontale" in figura 109.

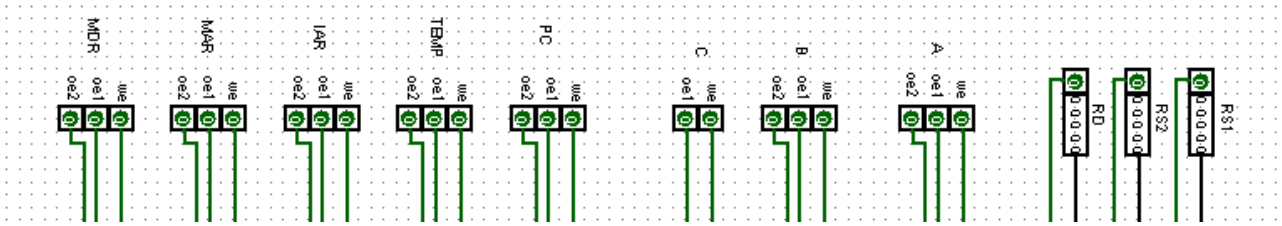


Figura 109: Segnali di Controllo del Data Path

È possibile poi verificare come l'output delle memorie corrisponda effettivamente agli enable dei segnali mappati necessari a effettuare ad esempio $TEMP \leftarrow B$. Il confronto fra output e mappatura viene mostrato in figura 110 e 111.

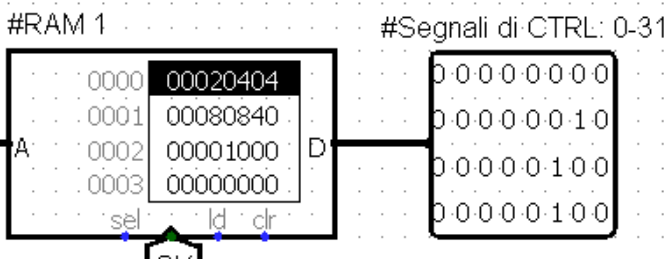


Figura 110: Il dato in uscita dalla RAM viene trasformato in segnali di controllo

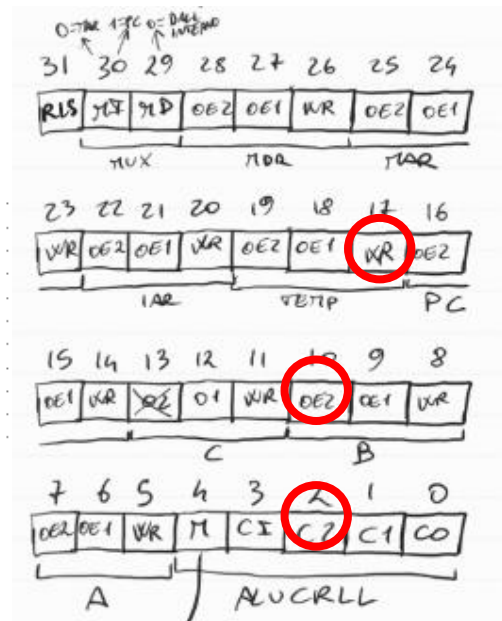


Figura 111: Mappatura dei Segnali di Controllo

Si può notare dalla figura come i segnali di controllo stiano alzando il WE di TEMP e l'OE di B. In più c2 c1 e c0 vengono settati per effettuare l'operazione di identità nella ALU. Collegando infatti nella maniera corretta i segnali di CTRL in uscita dalle RAM ai pin dei segnali di controllo del Data Path si "accenderanno" i pin corretti, come mostrato in figura 112. Come spiegato in precedenza, questo corrisponde a spostare il contenuto di B in TEMP nel Data Path, al successivo colpo di clock.

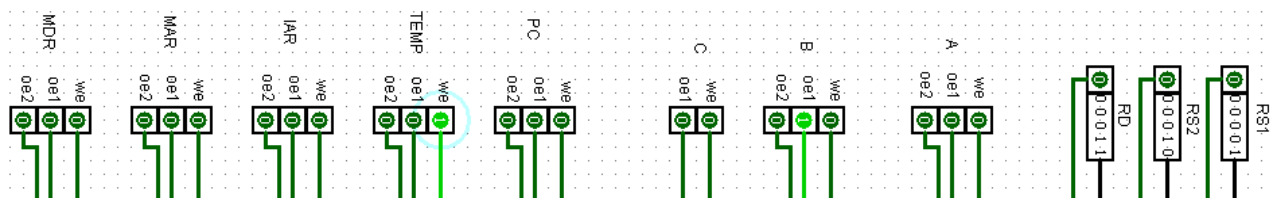


Figura 112: Visualizzazione dei pin dei segnali di controllo del Data Path collegati con l'output delle RAM del Microcodice

L'impiego delle memorie per contenere codice RTL consente quindi il passaggio ad un livello di astrazione superiore: una volta scritta sulla RAM, per "selezionare" una

microistruzione non serve più specificare tutti i segnali di controllo, basta semplicemente specificare l'indirizzo della cella nella quale è stata caricata la microistruzione. Questo concetto viene riassunto in *figura 113*.

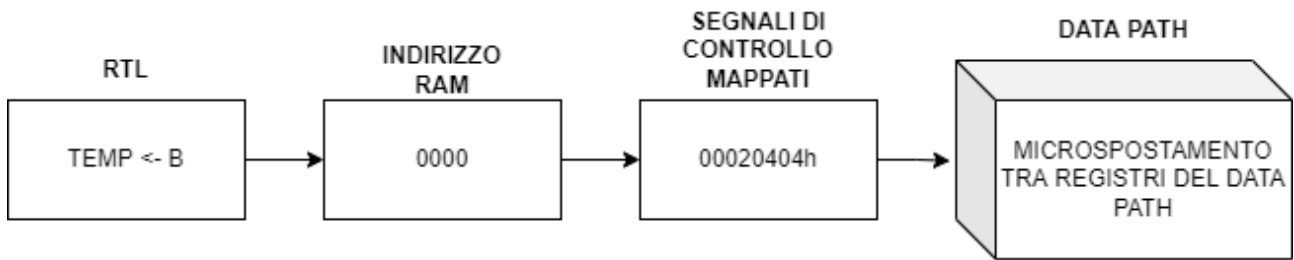


Figura 113: Schema dei passaggi per tradurre una Microistruzione RTL in un Microspostamento tra registri del Data Path mediante Microcodice

Le stringhe associate ai diagrammi di flusso quindi adesso non sono più segnali di controllo, bensì indirizzi, e ad ogni fase del diagramma di flusso corrisponderà un indirizzo e quindi un output diverso delle memorie, come mostrato in *figura 114*:

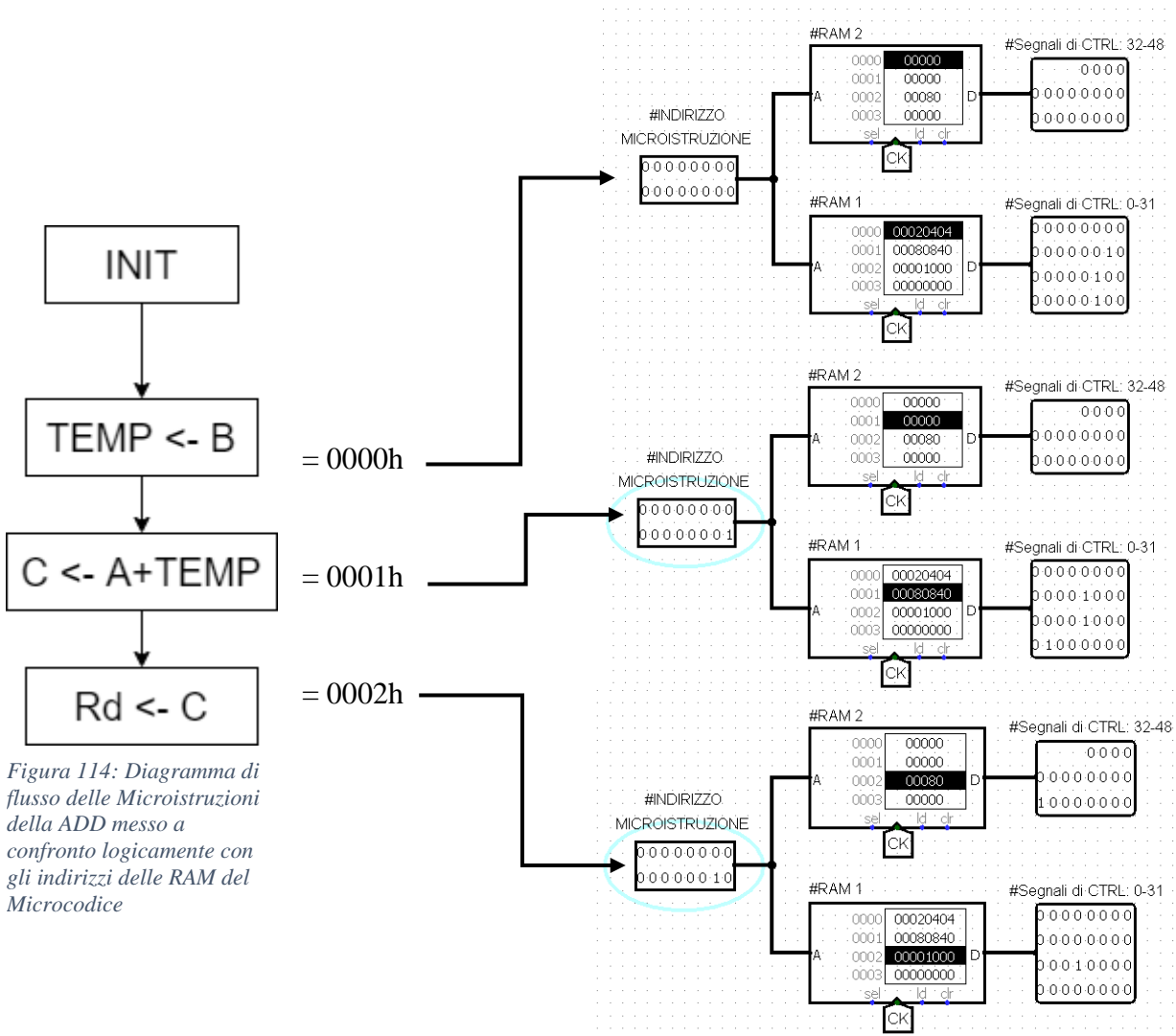


Figura 114: Diagramma di flusso delle Microistruzioni della ADD messo a confronto logicamente con gli indirizzi delle RAM del Microcodice

Con questo stratagemma è possibile semplificare il discorso e aumentare il livello di astrazione, tuttavia non si è ancora raggiunto lo scopo principale: l'esecuzione sequenziale di tali microistruzioni, allo stesso modo di una Rete Sequenziale. Sebbene il Data Path ora sia pilotato dalle RAM, gli indirizzi vanno comunque ancora inseriti manualmente. L'impiego delle RAM comporta però un vantaggio sostanziale: gli indirizzi sono ordinati, ovvero legati tra loro da un legame aritmetico.

Si consideri ad esempio l'indirizzo della prima microistruzione $TEMP \leftarrow B$, ovvero 0000h. La cella della microistruzione successiva $C \leftarrow A + TEMP$ si troverà all'indirizzo successivo, ovvero 0001h, mentre $Rd \leftarrow C$ avrà indirizzo 0002h. Generalizzando, se la prima microistruzione si trova ad un qualsiasi indirizzo $IND1$, la seconda si troverà in $IND1+1$, la terza in $IND1+2$, e così via. Sostanzialmente da tre indirizzi diversi il problema si è spostato nell'indicare un indirizzo di partenza e uno spostamento da tale indirizzo, ovvero un OFFSET. L'indirizzo di una qualsiasi microistruzione nelle RAM si esprime quindi come $IND1 + OFFSET$ e lo schema logico di questo concetto viene mostrato in *figura 115*, ampliando lo schema precedentemente mostrato in *figura 114*.

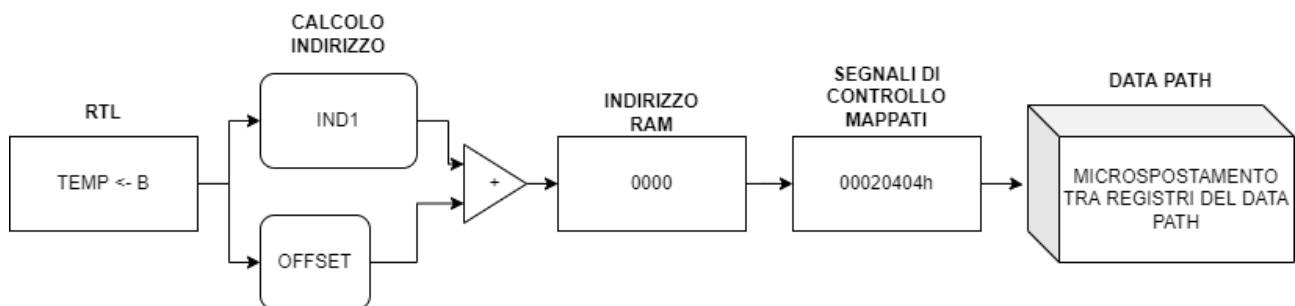


Figura 115: Schema dei passaggi per tradurre una Microistruzione RTL in un Microspostamento tra registri del Data Path mediante Microcodice ampliato con il blocco Sommatore e gli Input IND1 e OFFSET

Tale situazione in Logisim può essere ottenuta ampliando lo schema delle RAM aggiungendo un nuovo input per l'OFFSET e un sommatore, in questo caso una ALU, per il calcolo dell'indirizzo completo. Lo schema circuitale in logisim viene mostrato in *figura 116*.

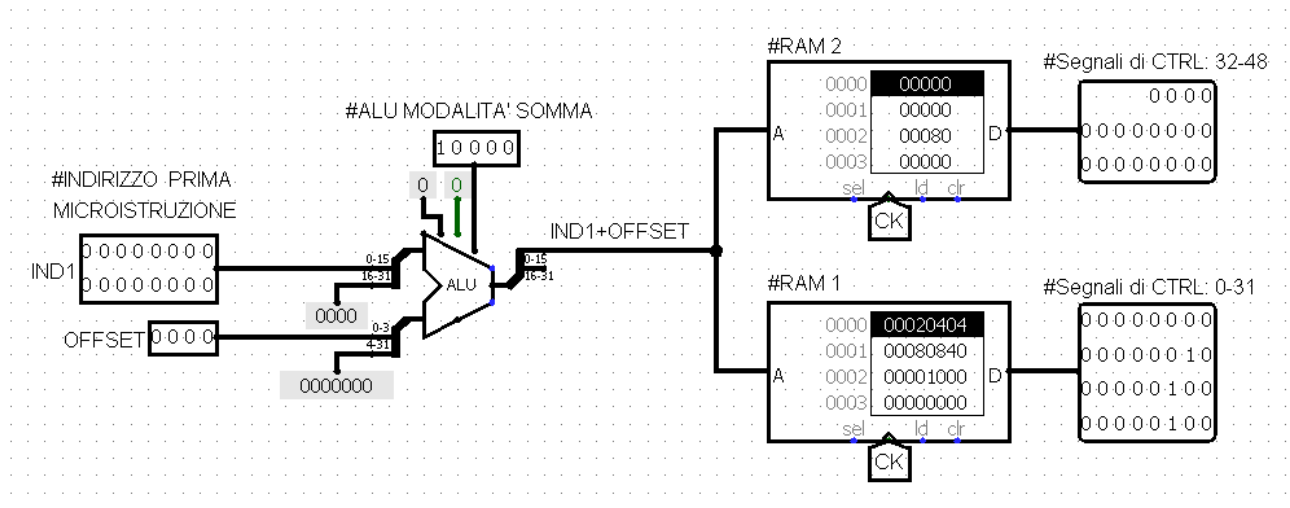


Figura 116: Schema logico in Logisim di Microcodice caricato in RAM pilotate da una coppia di IND1 e OFFSET sommati

Per selezionare la seconda microistruzione ad esempio basta quindi impostare l'OFFSET pari a 1, come in figura 117:

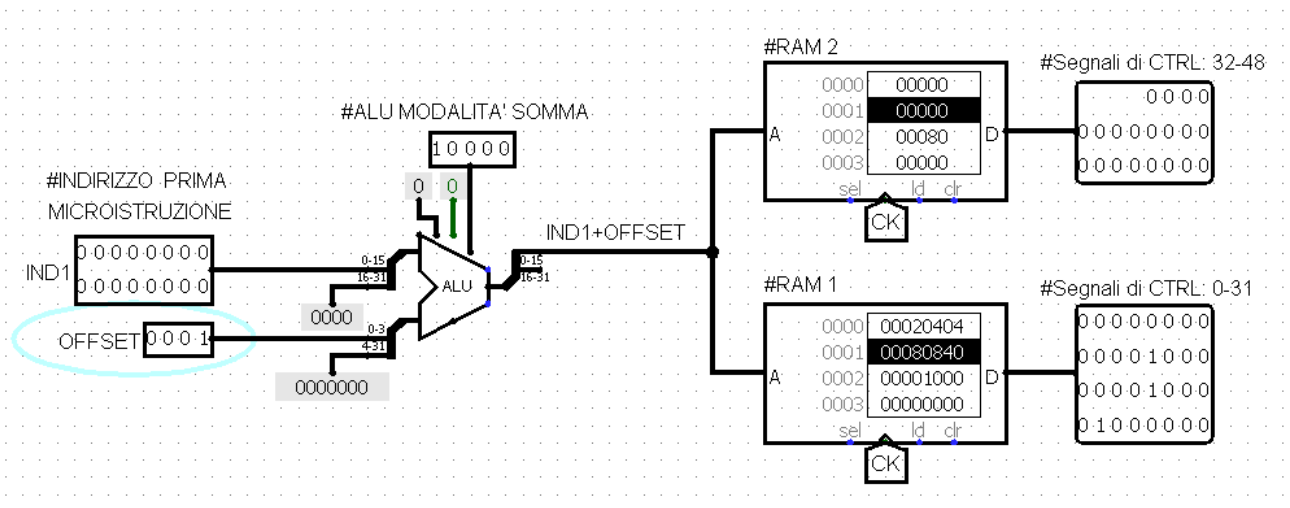


Figura 117: Stato delle Memorie RAM a seguito dell'incremento dell' Offset di 1

Va notato come IND1 rimane invariato ed è solo l'offset a cambiare.

Il nuovo flusso delle microistruzioni dell'istruzione ADD si riduce quindi ad esprimere l'indirizzo di partenza e un offset per ogni microistruzione, come mostrato in figura 118:

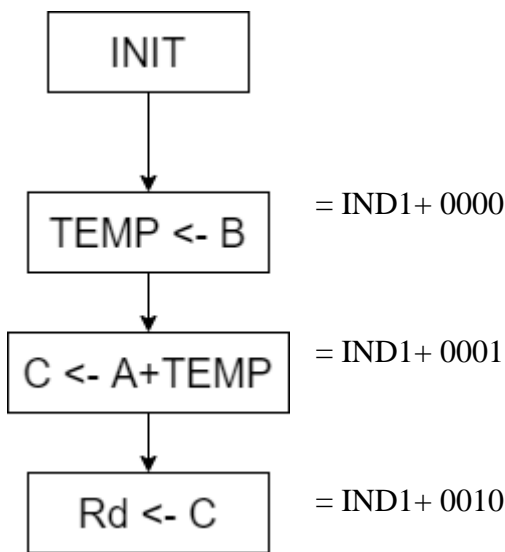


Figura 118: Diagramma di Flusso di una Istruzione ADD messo a confronto con IND1 e OFFSET corrispondenti a ogni riquadro

Supponendo ora di avere un circuito in grado di emettere un offset che incrementi di 1 ogni clock, si riuscirebbe ad eseguire la serie di microistruzioni sequenziali in maniera automatica, allo stesso modo della realizzazione mediante Rete Sequenziale dell'UDC vista nel paragrafo precedente. Senza il concetto di offset, le microistruzioni non avrebbero nessun legame aritmetico tra loro e non sarebbe possibile introdurre una automatizzazione logica.

Una rete che incrementa la sua uscita di 1 ad ogni clock altro non è che un COUNTER. Per quanto riguarda la quantità necessaria di bits del counter, solitamente una istruzione Assembly non supera le 7-8 microistruzioni, quindi l'offset sarà quasi sicuramente minore di 8. Per stare larghi viene quindi impiegato un COUNTER a 4 bit, quindi questa realizzazione circuitale consente di eseguire istruzioni composte da massimo 16 microistruzioni, quindi dalla durata massima di 16 clock. Naturalmente non c'è limite alla grandezza dell'offset, ma aumentandola troppo si rischia di tendere verso una architettura CISC piuttosto che la comoda RISC che si sta tentando di realizzare. Lo schema logico di un 2Bit COUNTER viene mostrato in *figura 119*. Non è necessario però realizzare lo schema circuitale in logisim, poiché i counter sono componenti già presenti nelle librerie Built-in (*figura 120 e 121*).

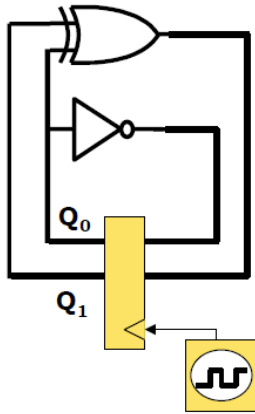


Figura 119: Schema logico di un 2Bit Counter (tratta dalle Slides del corso di Calcolatori Elettronici – A.A 2019)

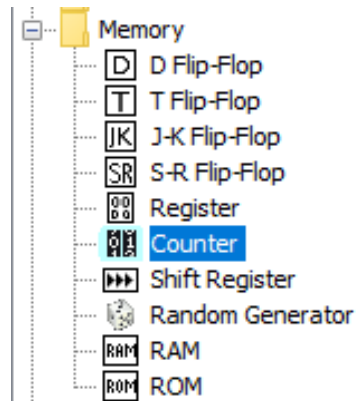


Figura 120: Voce "Counter" nella libreria Memory

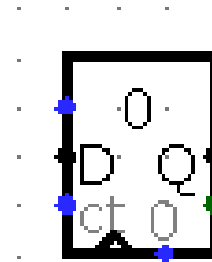


Figura 121: Componente del Contatore nel Canvas Logisim

Selezionando il componente Counter dalla libreria Memory e posizionandolo nel Canvas al posto dell'input OFFSET, si ottiene lo schema circuitale mostrato in figura 122.

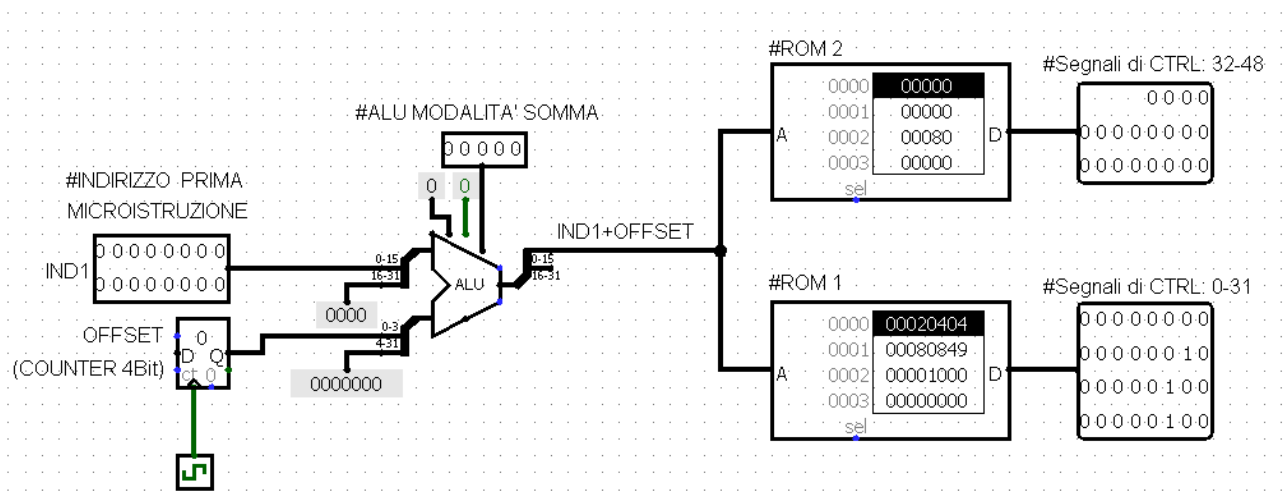


Figura 122: Schema logico delle ROM del Microcodice pilotate da IND1 e un counter collegato al Clock

Come si può notare dall'immagine, sono state modificate 3 cose nel circuito:

- 1 – Aggiunta del Counter
- 2 – Aggiunta del segnale di Clock (il quadrato con il simbolo dell'onda quadra sotto il counter)
- 3 – Sostituzione delle RAM con delle ROM

Il segnale di Clock è un componente molto importante, poiché aiuta molto nell'analisi delle reti sequenziali. Fino ad ora l'impulso di clock veniva dato con un pin uguale a

un normale input da un bit: premendo sul pin si produce un fronte di salita, mentre premendolo di nuovo si produce un fronte di discesa, e così via. Il blocco del Clock riproduce la stessa situazione, ma in maniera automatica. Selezionando la voce del menu Simulate apparirà infatti una finestra che consente il controllo preciso del segnale prodotto dal Blocco di Clock. Una soluzione ancora più comoda è usare i Key-Bindings forniti dal software Logisim:

- CTRL+T: simulazione di un unico fronte di salita o discesa
- CTRL+K: abilitazione del clock automatico
- CTRL+R: reset simulazione (ritorno allo stato iniziale)

Nel caso si voglia usare il segnale di clock automatico, sempre dal menu Simulate è possibile impostare la frequenza di clock, da 0.25Hz a 4.1kHz.

La possibilità di simulare un unico fronte di salita alla volta permette l'analisi di una rete sequenziale passo per passo, prendendosi tutto il tempo necessario tra un colpo di clock e l'altro per comprendere la situazione. È un modo per "spacchettare" il tempo discreto nella quale esiste un calcolatore.

Un problema del controllo della simulazione con blocco di Clock integrato consiste nel fatto che, una volta premuto CTRL+R, non solo viene resettata la simulazione, ma il contenuto di tutte le RAM viene cancellato e riportato a zero. Inoltre, il contenuto delle RAM viene resettato anche alla chiusura del programma. È immediato capire come questo componente, esattamente come nella realtà, non sia adatto a contenere del codice "permanente", mentre è perfetto invece per simulare una memoria volatile.

Logisim offre una alternativa alla memoria RAM: le ROM. Acronimo di Read Only Memory, come suggerisce il nome le ROM sono memorie fatte per funzionare in sola lettura. Non è possibile modificare il contenuto di una ROM attraverso circuiti logici, ma ovviamente tramite l'editor è possibile caricare manualmente il codice permanentemente sulla memoria. Per realizzare una memoria che contenga del microcodice le ROM sono quindi i componenti più adeguati.

Mediante l'impiego di counter, ALU, e ROM è quindi possibile realizzare una automatizzazione parziale dell'esecuzione delle microistruzioni. "Parziale" poiché sebbene l'offset sia controllato da un Counter automatico, l'indirizzo della prima microistruzione IND1 deve essere ancora calcolato manualmente e fornito alla rete da una "mano umana". Ciononostante, una volta fornito IND1 la rete realizzata finora è in grado di eseguire la sequenza di microistruzioni in maniera automatica, come mostrato nelle successive *figure 123 124 e 125*.

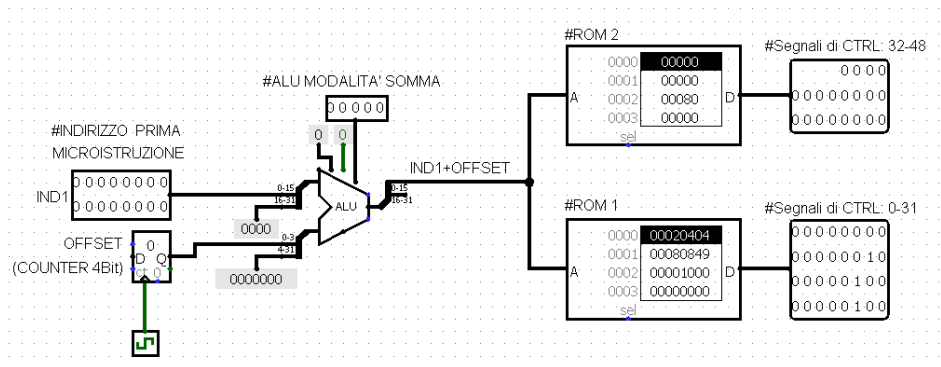


Figura 123: Stato delle ROM del Microcodice dell'UDC prima del primo fronte di salita

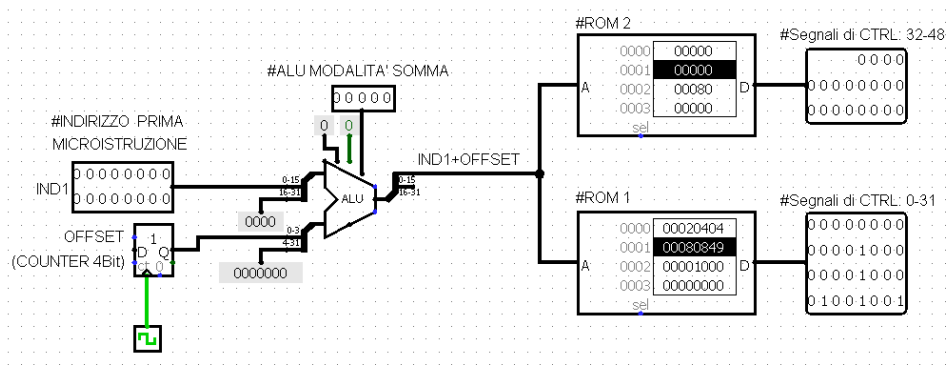


Figura 124: Stato della rete dopo il primo fronte di salita, la cella selezionata nelle ROM cambia

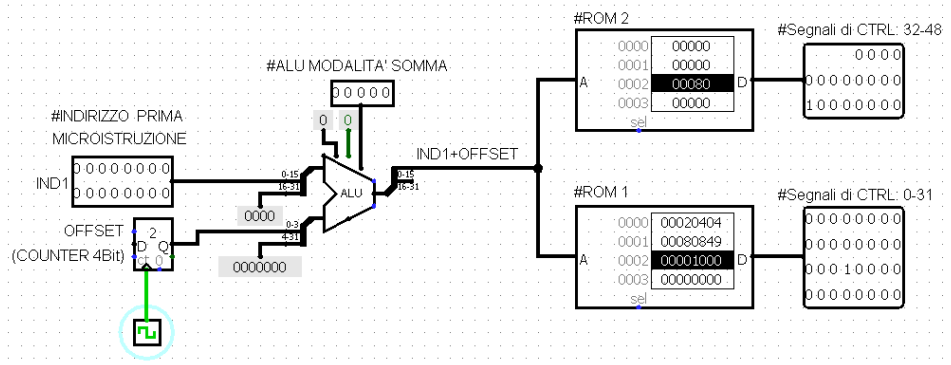


Figura 125: Stato della rete dopo il secondo fronte di salita, viene selezionata la cella contenente l'ultima microistruzione, ovvero la fase di Write Back

La situazione in cui si è giunti ora è quindi paragonabile al Diagramma degli Stati mostrato nel precedente paragrafo, riproposto in *figura 126*:

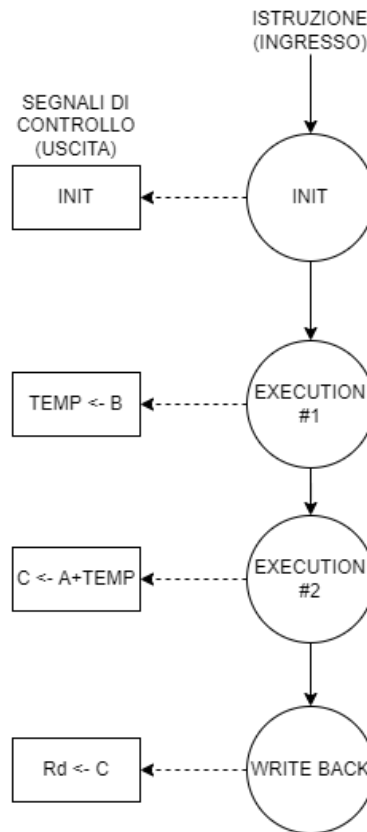


Figura 126: Diagramma degli Stati messo a confronto con le microistruzioni della ADD

È a questo punto però che i vantaggi del microcodice diventano sostanziali: aggiungere nuove istruzioni alla rete è molto più semplice ed è proprio per questo che questo tipo di implementazione viene chiamata a “Microcodice”. Supponendo ad esempio di voler aggiungere l’istruzione SUB alla rete, il processo è il medesimo della ADD: è sufficiente caricare le stringhe di bit corrispondenti alle microistruzioni della SUB nelle prime celle libere dopo le microistruzioni della ADD, come mostrato in figura 127.

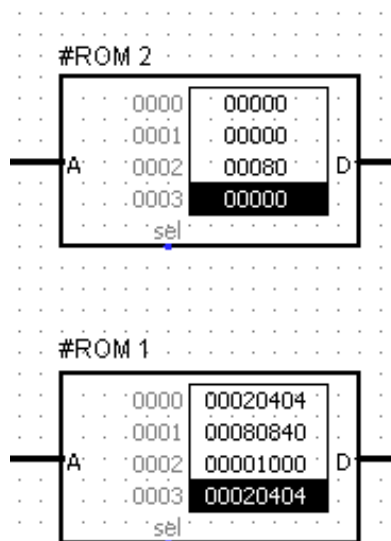


Figura 127: Microcodice dell’istruzione SUB caricato nelle ROM dell’UDC

Questo meccanismo è reso possibile proprio dalla coppia IND1 + OFFSET.
 Modificando IND1 infatti è possibile specificare il primo indirizzo delle microistruzioni della SUB, grazie al counter questo indirizzo verrà incrementato come nel caso della ADD, ma stavolta partendo non più da 0 bensì da IND1. Una volta quindi configurato l'INPUT della rete, avviando il clock è possibile osservare l'esecuzione sequenziale delle 3 microistruzioni della fase di EXECUTE della SUB in maniera identica alla ADD, come mostrato nelle successive *figure 128, 129 e 130*:

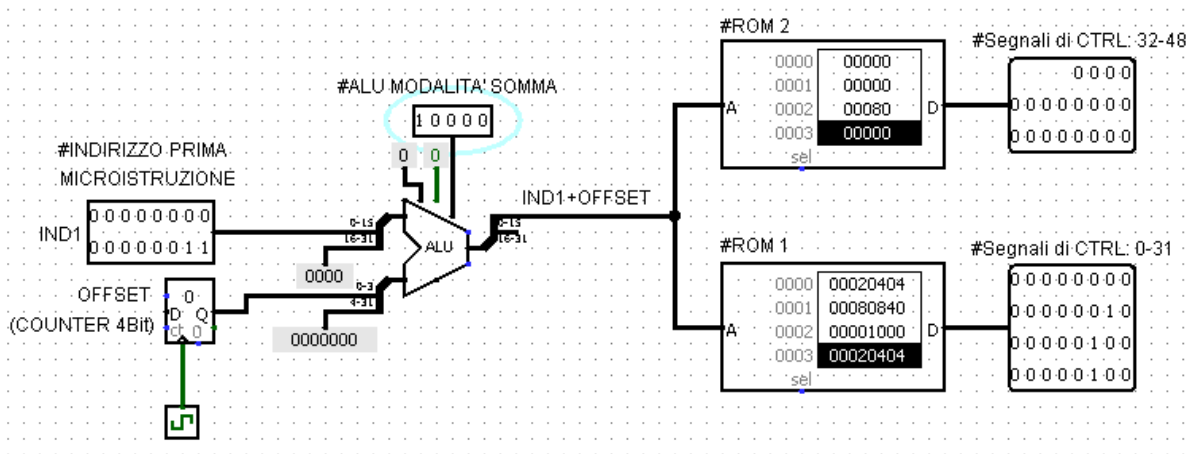


Figura 128: Stato delle ROM prima del primo fronte di salita, viene selezionata la cella della prima microistruzione della SUB

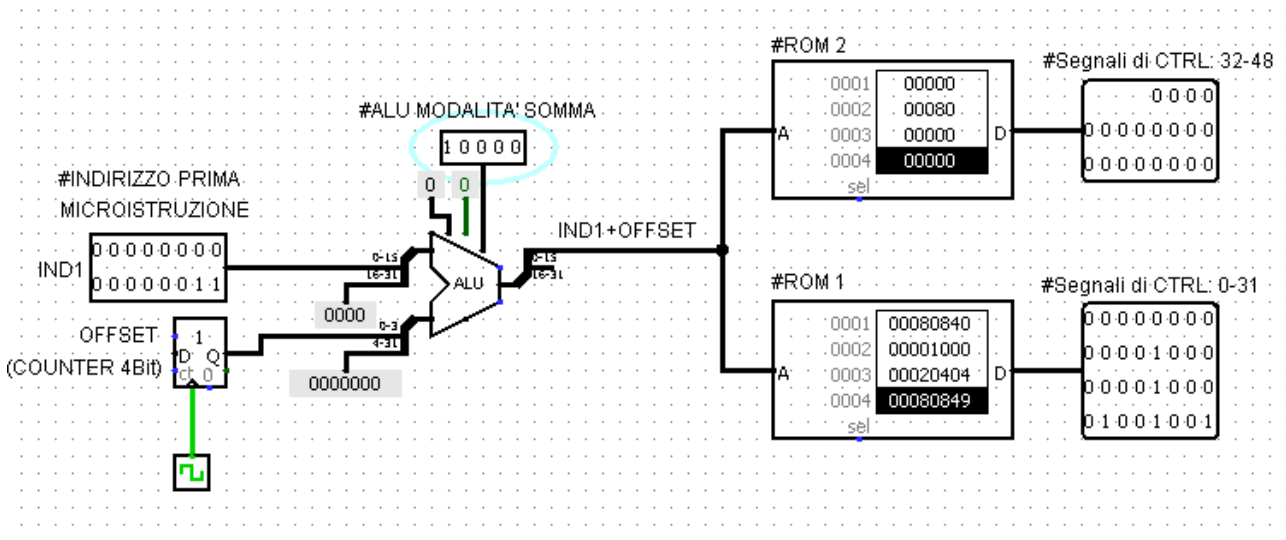


Figura 129: Stato delle ROM al primo fronte di salita

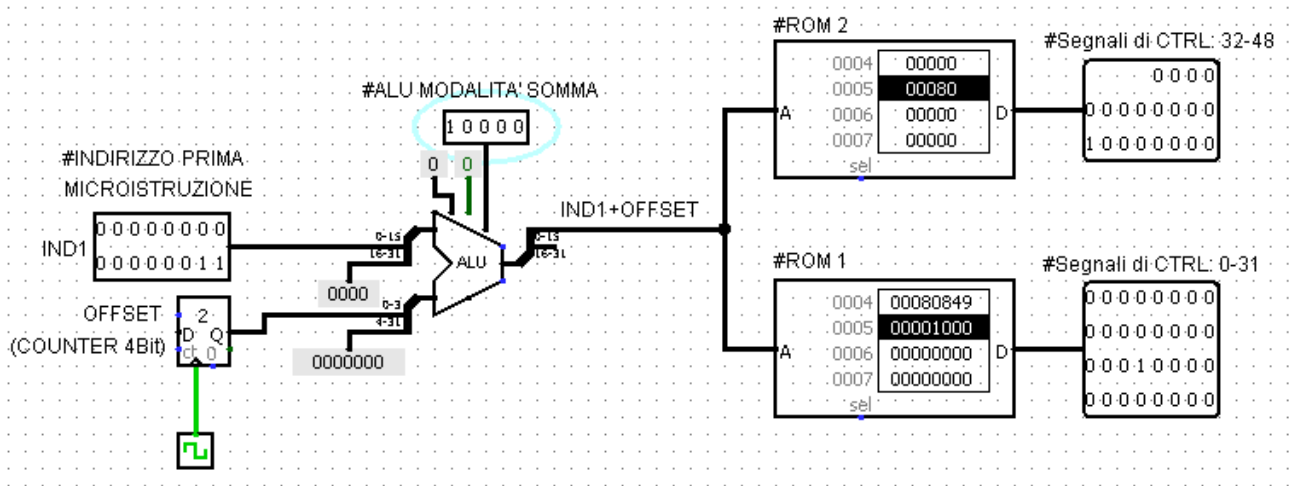


Figura 130: Stato delle ROM dopo il secondo fronte di salita

Osservando le ROM e i loro indirizzi in *figura 128*, la prima cella ad essere selezionata è quella ad indirizzo 3, che è effettivamente la prima microistruzione della SUB. Sebbene al clock successivo il counter valga comunque uno, l'indirizzo effettivo selezionato è 4, che è la seconda microistruzione della SUB, e così via. Cambiare il tipo di istruzione eseguita mediante l'impiego di IND1 è equivalente a cambiare il ramo del diagramma degli stati raffigurante le due istruzioni mostrato in *figura 131*:

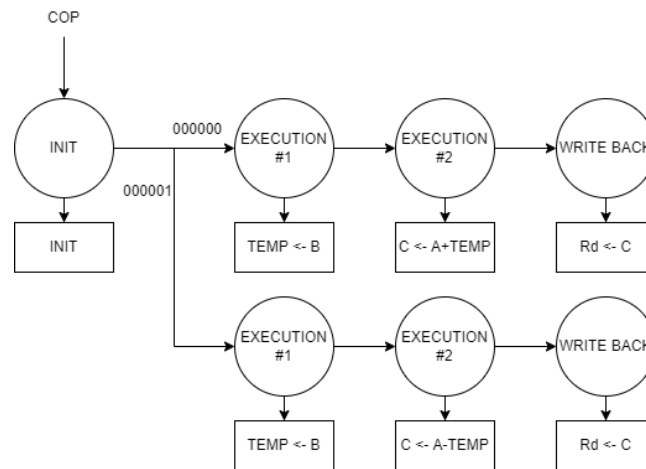


Figura 131: Diagramma degli Stati dell'UDC a Rete Sequenziale

È chiaro ora come l'indirizzo IND1 abbia rispetto al microcodice lo stesso significato del COP per la Rete Sequenziale, la differenza però è che in questo caso non è necessario fare alcuna modifica HARDWARE alla rete per aggiungere una nuova istruzione. È sufficiente scrivere UNA VOLTA SOLA le microistruzioni nelle ROM per avere lo stesso "effetto" sui segnali di controllo che aveva la Rete Sequenziale dell'UDC, ed è proprio questa funzione di "traduzione" da Software (istruzione) ad Hardware (Data Path) che rende il microcodice un linguaggio di tipo FIRMWARE.

Supponendo di nuovo di voler eseguire una ADD usando le ROM con caricato il microcodice di tutte le istruzioni, nella simulazione è possibile osservare come, dopo il terzo fronte di salita, il counter non si ferma da solo, bensì continua ad incrementare il valore di OFFSET, che ora è a 3, come mostrato in *figura 132*.

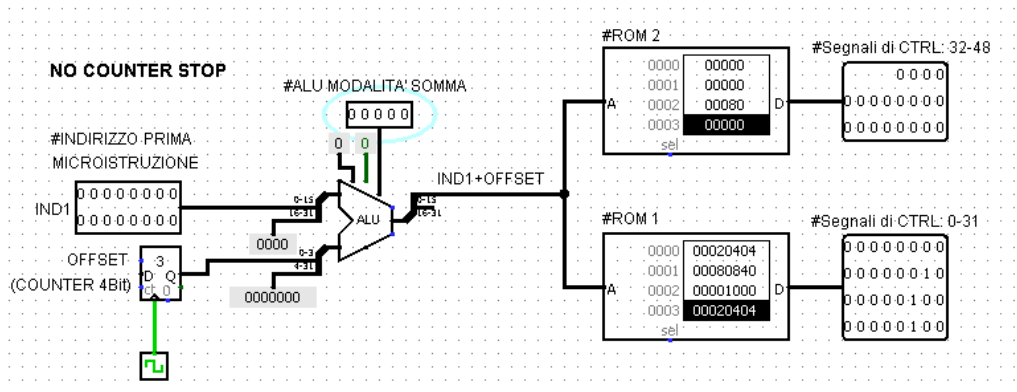


Figura 132: Overflow del Counter al terzo fronte di salita

La situazione è problematica, poiché la microistruzione all'indirizzo 3 non fa parte della ADD, bensì è la prima microistruzione della SUB. L'OFFSET continua a incrementare indiscriminatamente, eseguendo in maniera ciclica tutte le istruzioni delle ROM fino ad arrivare a overflow. Questo comportamento nel diagramma degli stati è rappresentabile nello schema in *figura 133*:

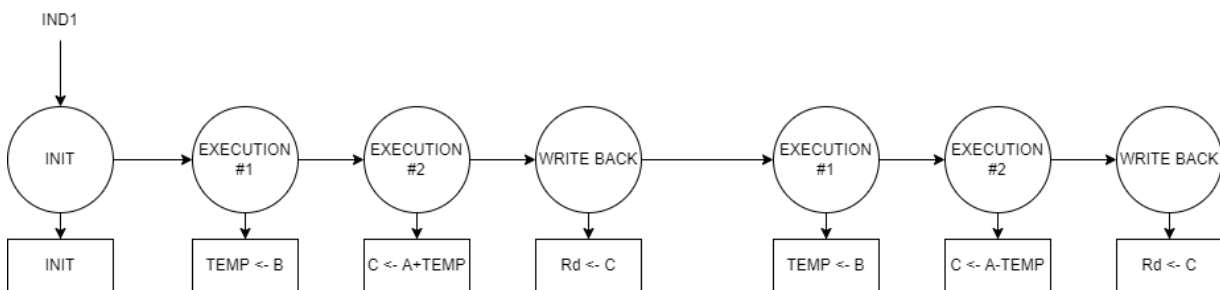


Figura 133: Diagramma degli Stati dell'UDC a Rete Sequenziale modificato a causa del bug

Il motivo di questo “bug” è la mancanza di una informazione essenziale per la rete: il numero di Clock per istruzione. Senza di essa il counter non sa quando fermarsi e continua a contare all'infinito. Il problema consiste quindi nel fornire al Counter un segnale che gli comunichi quando smettere di contare nel giusto periodo di Clock.

Una prima soluzione consiste intuitivamente nel fermare “manualmente” il Counter mediante il suo pin di Reset Asincrono dopo aver eseguito l'ultima microistruzione, come mostrato in *figura 134*.

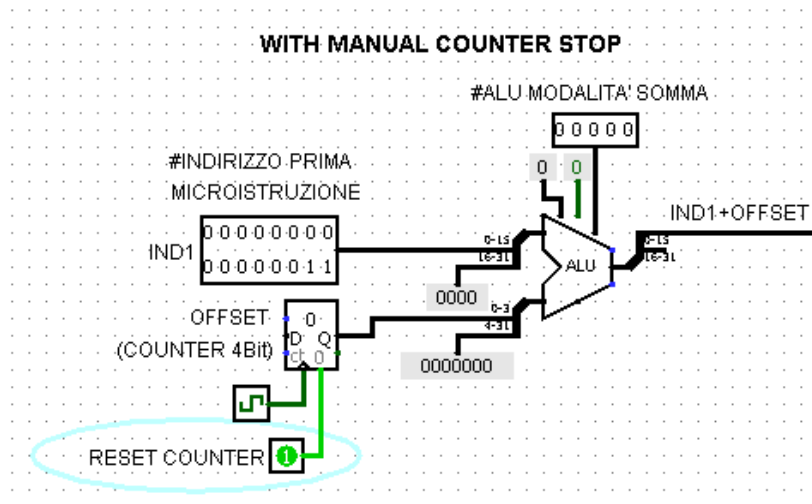


Figura 134: Contatore con segnale di reset manuale

Questa implementazione però richiederebbe di contare gli stati dell'istruzione guardando gli indirizzi delle ROM durante l'esecuzione. Questo è un passo indietro rispetto al livello di automazione raggiunto finora, dove una volta fornito l'indirizzo iniziale IND1 in ingresso all'ALU si affidava il resto del lavoro al Counter. Si provi però a ragionare come un essere umano che deve pilotare manualmente la rete: questo essere umano deve fermare il counter quanto sono passati i clock necessari a eseguire gli stati dell'istruzione. Questo equivale a: leggere il valore fino alla quale è arrivato il counter, confrontarlo con i CPI dell'istruzione e decidere se resettarlo o meno.

1. Leggere il valore del Counter equivale a leggerne l'uscita
2. Confrontare il valore del Counter con i CPI equivale a confrontare due numeri: XOR bit a bit
3. Collegando tutti gli XOR ad un AND in uscita si ha un unico bit che rappresenta proprio il segnale di Reset da inviare al Counter

La realizzazione circuitale della situazione descritta dai tre passaggi precedenti viene mostrata in *figura 135*, dove vengono usati XNOR e AND per generare il valore corretto del segnale di Reset Counter e il valore dei CPI viene fornito da un nuovo INPUT denominato CPI.

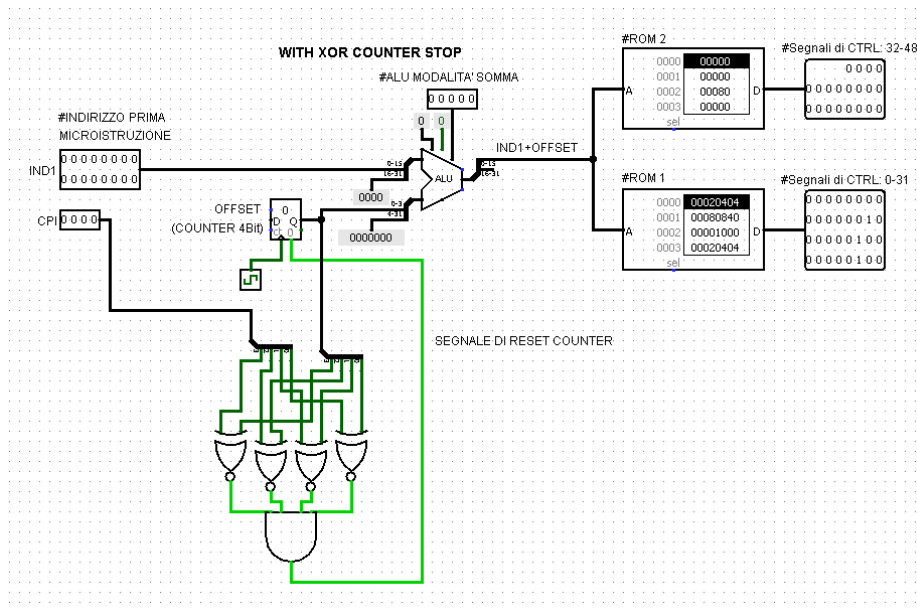


Figura 135: Schema circuitale dell'UDC con rete di confronto che fornisce il segnale di reset al Counter

Per motivi di visualizzazione tale circuito viene semplificato mediante l'impiego di un SubCircuit per realizzare la Rete di Confronto come mostrato in figura 136:

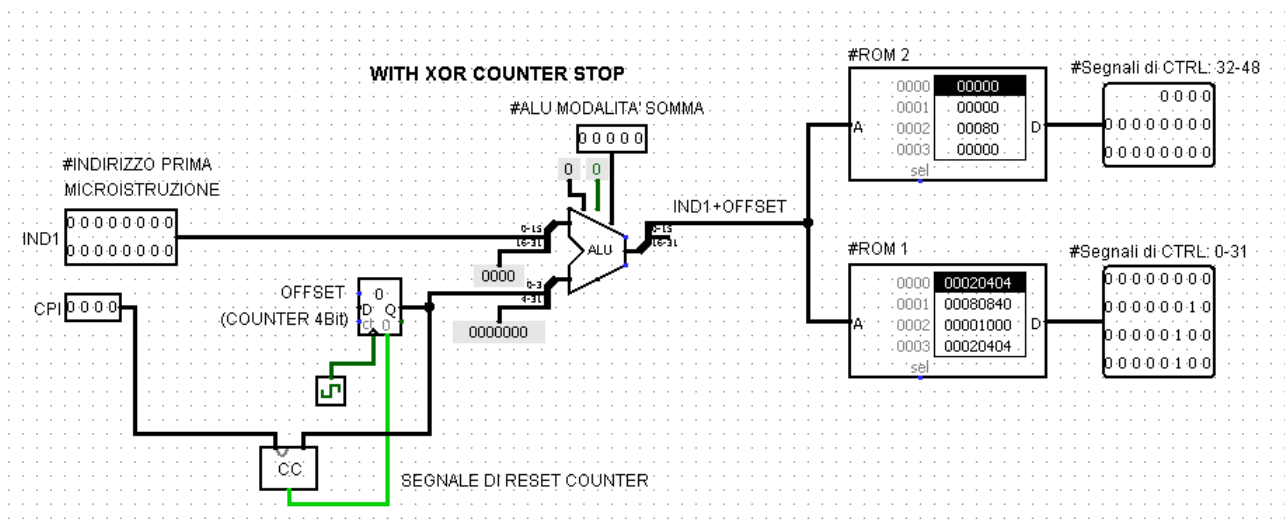


Figura 136: Schema circuitale dell'UDC con rete di confronto realizzata mediante un sottocircuito

La rete ora ha due input: IND1 e CPI. Adesso la scelta dell'istruzione non consiste più nel fornire solamente IND1, in aggiunta bisogna fornire anche i CPI dell'istruzione mediante il nuovo INPUT.

Successivamente viene mostrato il confronto tra le implementazioni con e senza rete di confronto: nel primo caso in figura 137 il counter continua a incrementare come mostrato in precedenza, mentre nel secondo caso in figura 138 il counter viene resettato correttamente.

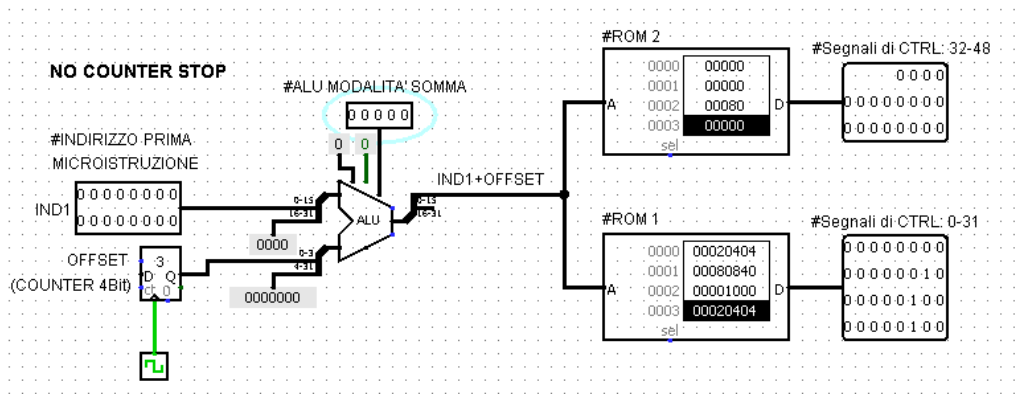


Figura 137: Overflow del Counter dell'UDC dopo il fronte di salita

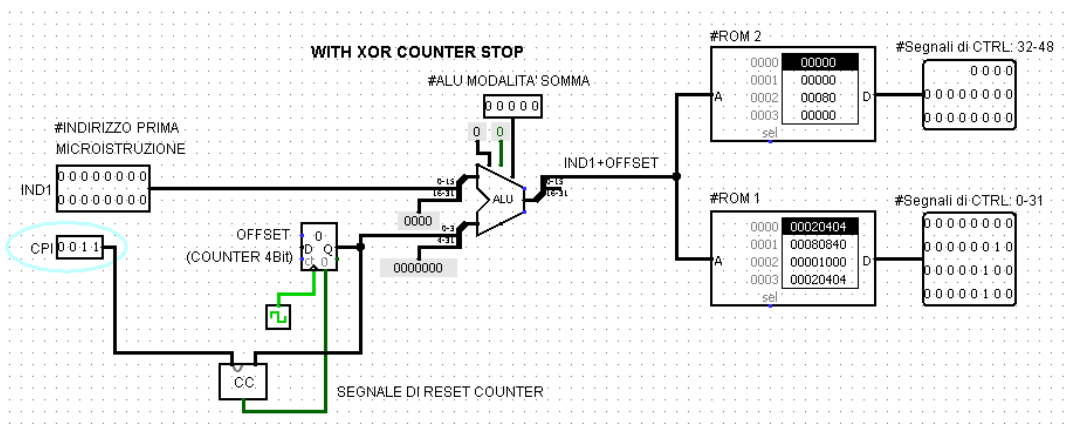


Figura 138: Reset Asincrono del Counter dopo il fronte di salita

Il Segnale di Reset Counter non apparirà mai acceso: essendo asincrono, nell'istante in cui si alza il counter viene resettato, ma a questo punto la sua uscita vale zero e quindi la rete di confronto cessa di emettere il segnale di Reset, che è rimasto acceso per un istante.

Rimane ora un'ultima differenza tra il Microcodice e la Rete Sequenziale dell'UDC. Si ricorda in figura 139 come in ingresso la Rete Sequenziale prenda una istruzione Assembly codificata in Codice Macchina.

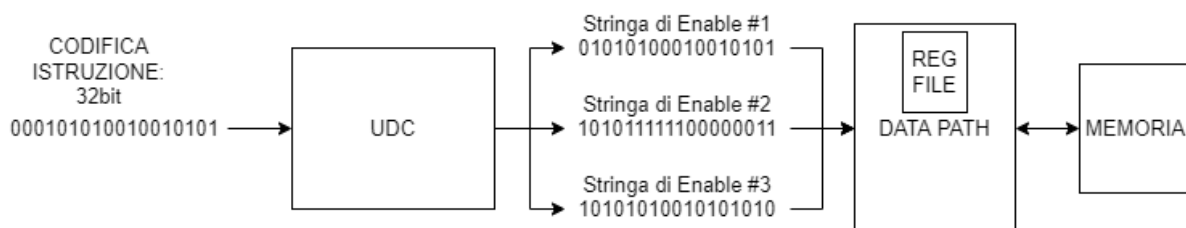


Figura 139: Schema logico del comportamento sequenziale della rete dell'UDC con segnali di Enable esplicitati

Nella rete a Microcodice realizzata finora però l'ingresso è formato dai due campi IND1 e OFFSET, come mostrato in *figura 140*.

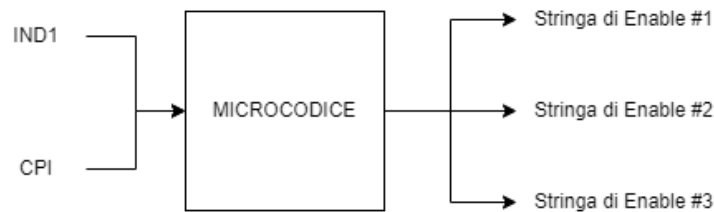


Figura 140: Schema Logico del rapporto ingresso-uscita dell'UDC realizzata mediante Microcodice

Affinché la rete sia compatibile con il codice Assembly, è necessario quindi tradurre i 32 bit di codice Macchina in ingresso con la coppia IND1 e CPI. Più precisamente, è il COP dell'istruzione che deve essere tradotto, poiché i campi restanti corrispondono ai registri che vengono trattati con circuiteria aggiuntiva. Il problema diventa quindi tradurre i 6 Bit di COP nei bit corrispondenti a IND1 e OFFSET.

Come visto finora, quando si tratta di tradurre stringhe di bit in linguaggi diversi, le Memorie sono lo strumento più comodo per raggiungere il fine. Anche in questo caso verrà usata una ROM, dove in ingresso verrà fornito il COP, mentre in uscita verrà restituita la coppia IND1 + OFFSET. Tale ROM viene per questo motivo definita COP MEMORY e lo schema circuitale completo di memoria di traduzione viene mostrato in *figura 141*.

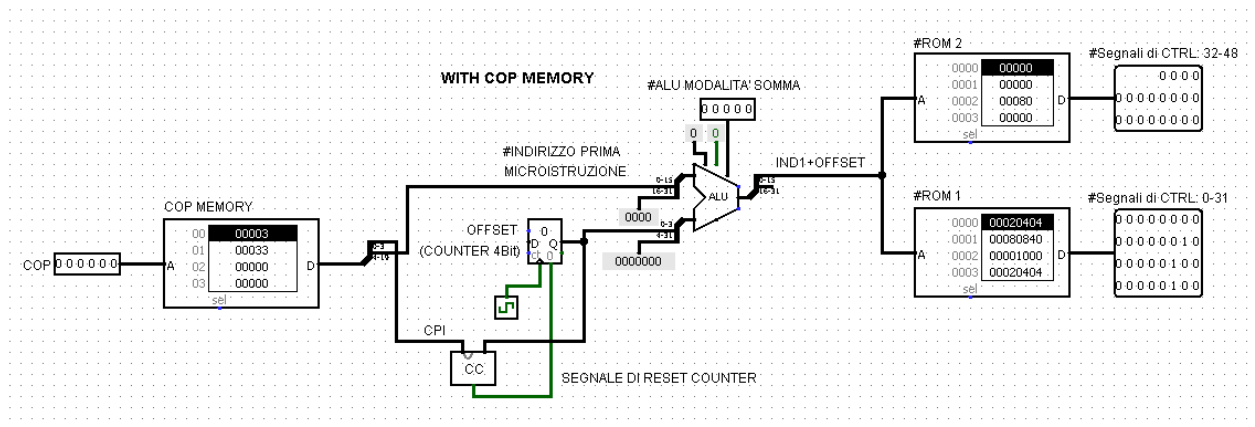


Figura 141: Schema circuitale della rete dell'UDC con aggiunta di COP MEMORY per migliorare la compatibilità

Osservando il contenuto della COP MEMORY in figura, è possibile notare come nella prima cella sia presente il valore 0003h mentre nella seconda sia presente 0033h. I primi 4 bit della sequenza a partire da destra corrispondono ai CPI dell'istruzione, mentre i restanti rappresentano IND1. Mediante uno splitter è quindi possibile incanalare queste due informazioni agli input corrispondenti della rete a Microcodice realizzata in precedenza. Come mostrato in *figura 141*, il COP diventa quindi fisicamente un indirizzo di 6 bit e la COP memory di conseguenza potrà tradurre 2^6 COP in coppie di IND1 e CPI.

Selezionando ad esempio un COP pari a 00001, ovvero la SUB, avviando il CLOCK è possibile notare come innanzitutto venga selezionata la seconda cella della COP MEMORY, mandando quindi in uscita il valore 0033h. I primi 4 bit, ovvero 3h=0011 vengono splittati e mandati in ingresso alla rete di Confronto del Counter, mentre i restanti vengono mandati all'ingresso IND1, selezionando quindi la cella di valore 3h nelle ROM del microcodice, come mostrato in figura 142.

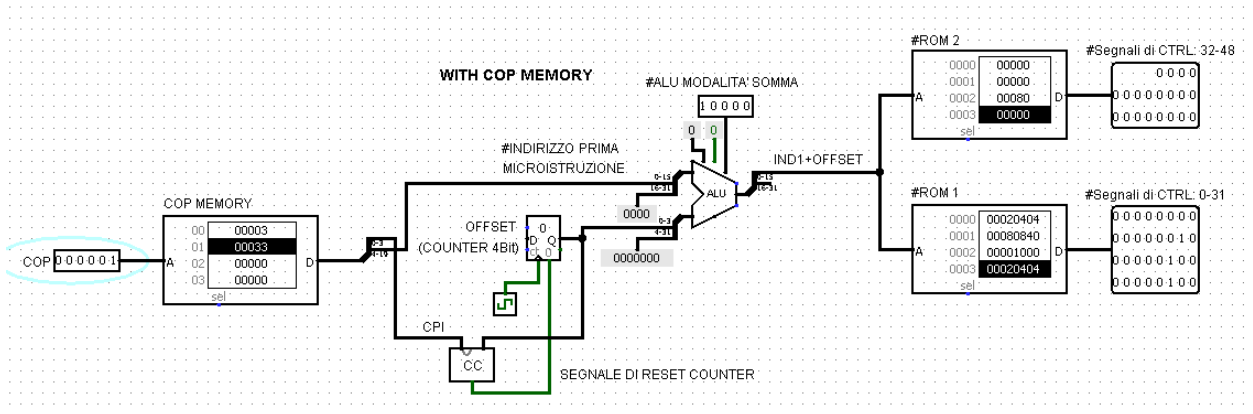


Figura 142: Stato della COP Memory dopo aver ricevuto in ingresso il codice COP dell'istruzione SUB

Ora premendo CTRL+K è possibile avviare il CLOCK automatico, alzare le mani, e osservare come la rete procede ad eseguire l'istruzione di SUB (figure 143 e 144) per poi tornare alla fase iniziale resettando il Clock in automatico (figura 145).

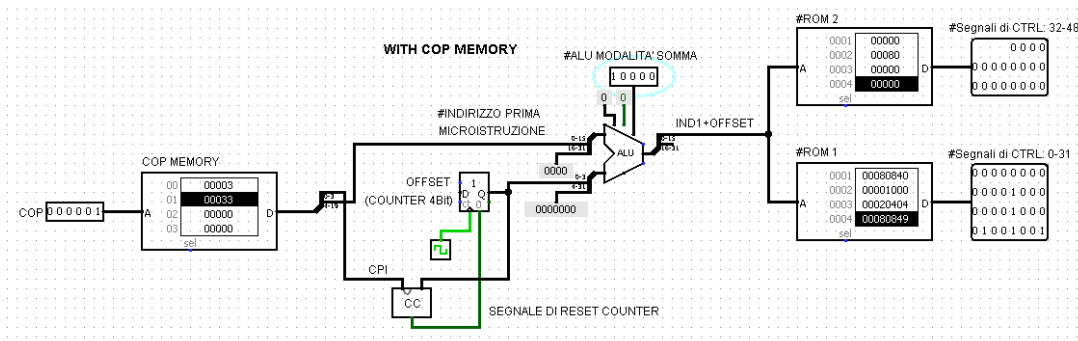


Figura 143: Stato della rete dopo il primo fronte di salita

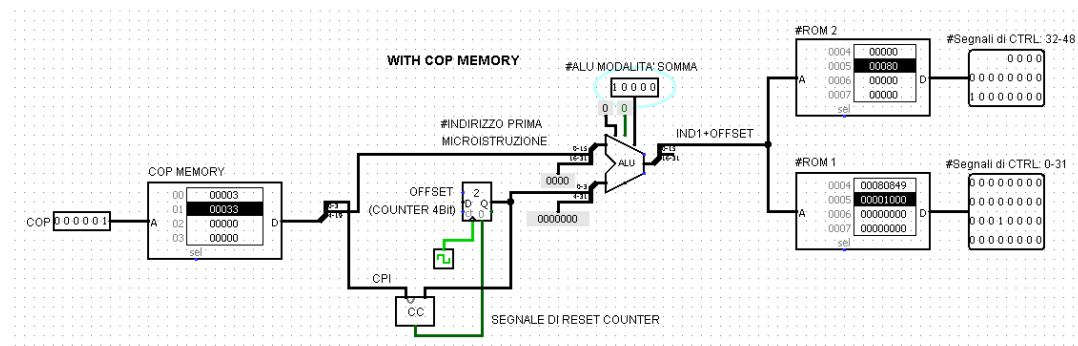


Figura 144: Stato della rete dopo il secondo fronte di salita

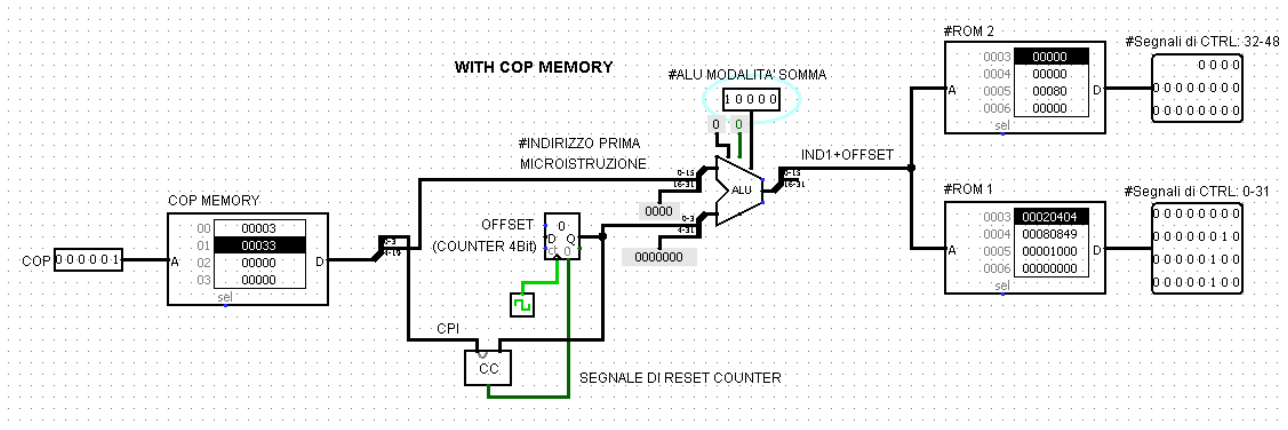


Figura 145: Stato della rete dopo il terzo fronte di salita: il Counter viene resettato correttamente

A questo punto però, sebbene il counter sia stato resettato, il Clock continua ad andare avanti, e la rete, trovandosi nella stessa situazione iniziale, ripete gli stessi passaggi. Se quindi il COP dell'istruzione rimane lo stesso, la rete eseguirà la stessa istruzione in un Loop infinito, seguendo lo schema mostrato in figura 146.

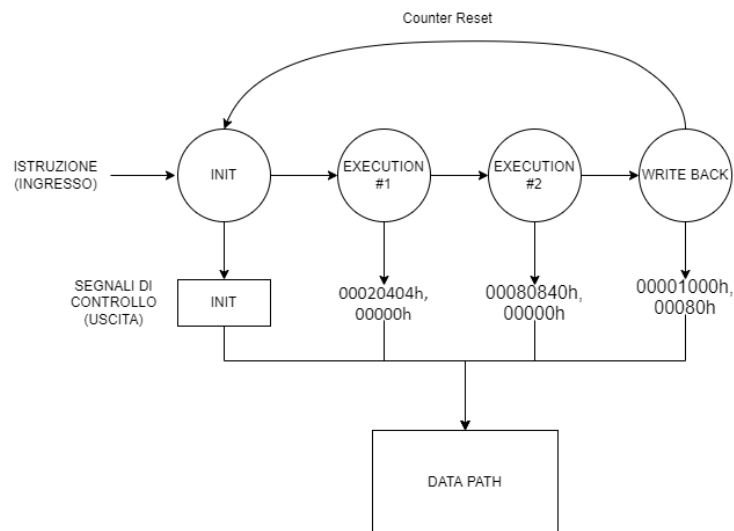


Figura 146: Completamento del Diagramma degli Stati dell'UDC realizzata mediante rete sequenziale con loop infinito

La Macchina ora è completamente Automatica: essendo chiusa in un Loop infinito non smetterà mai di funzionare. Ogni ciclo di un Loop corrisponde quindi ad eseguire totalmente tutte le microistruzioni necessarie all'esecuzione di una Istruzione Assembly, e cambiare COP in ingresso equivale a cambiare la sequenza di microistruzioni eseguite ad ogni Loop.

3.14-Programmabilità di una Macchina di Von Neumann

Il concetto di “programmazione” della Macchina di Von Neumann, che all’inizio aveva un connotato molto vago, adesso è diventato un problema concreto: come già anticipato, un “programma Assembly” altro non è che una serie di istruzioni Assembly scritte in fila una di seguito all’altra, che a loro volta vanno tradotte in codice Macchina. Eseguire un programma Assembly in una Macchina di Von Neumann corrisponde quindi a “presentare” all’ingresso dell’UDC i COP delle istruzioni Assembly, forniti uno alla volta in ordine e in maniera Sequenziale, come mostrato in *figura 147*.

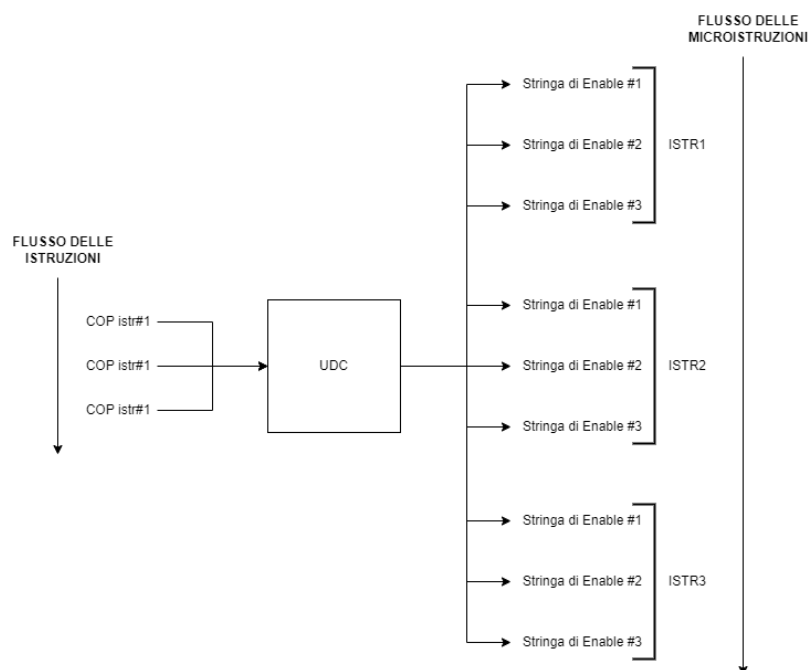


Figura 147: Confronto tra esecuzione sequenziale di Istruzioni e Microistruzioni

Mentre l’ Hardware, che lavora in modo automatico, permette l’esecuzione di una ISTRUZIONE mediante MICROISTRUZIONI IN SERIE, la Programmabilità Software è una astrazione di livello superiore, ovvero l’esecuzione automatica di un PROGRAMMA mediante ISTRUZIONI IN SERIE, che a loro volta saranno composte ognuna da microistruzioni in serie. È adesso che viene quindi fatto il salto da Hardware a Software, reso possibile dal ponte del Firmware. Mentre l’Hardware è legato all’Automaticità, il Software è in questo modo legato alla Programmabilità.

Da un punto di vista Logico, i due Concetti di Automaticità e Programmabilità sono simili: entrambi sono legati al concetto di sequenzialità, con la differenza che le Istruzioni Assembly possono essere scritte in qualsiasi ordine, non come il Microcodice che invece va scritto un’unica volta (da questo punto di vista il

microcodice è simile ad un Hardware). È proprio l'ordine e il tipo di Istruzioni Assembly infatti che determina il comportamento della Macchina.

Come mostrato di seguito, un esempio di programma Assembly può anche essere una semplice ADD e SUB scritte una di seguito all'altra:

```
add r1 r2 r3
```

```
sub r4 r5 r6
```

Questa rappresentazione è solo un aiuto per rendere il codice leggibile all'occhio umano. A ogni riga del codice può essere associata la stringa di 32 Bit corrispondente all'istruzione e l'insieme di tutte le stringhe di bit delle istruzioni viene definita "Codice Macchina", in questo caso le stringhe associate alle due istruzioni sono:

```
00430800h
```

```
04a62000h
```

L'aspetto dei bit rappresentati in figura risulta molto familiare, è una situazione identica al caso del microcodice dove erano presenti una serie di stringhe di bit posizionate una di seguito all'altra. Quindi, una serie di istruzioni può essere "caricata" in una memoria, seguendo proprio lo stesso criterio dell'RTL. Ed è proprio quello che è stato fatto: in *figura 148* viene mostrato l'esempio di una Memoria con caricate le stringhe corrispondenti al codice Macchina anticipato.

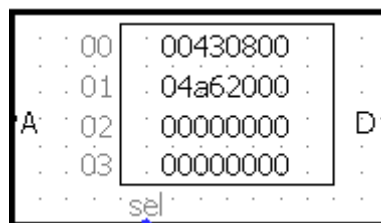


Figura 148: Memoria ROM Logisim con due istruzioni memorizzate nelle prime due celle

La Memoria contenente il codice Macchina è molto importante, ed è la "colonna" sulla quale viene implementato il concetto di Programmazione, in maniera simile al Microcodice.

Il problema della Programmazione ora diviene più chiaro: essendo il codice dell'istruzione salvato in memoria, per "riferirsi" a una istruzione è sufficiente specificare l'indirizzo in memoria alla quale è salvata, e l'output della cella di memoria selezionata conterrà il codice binario dell'istruzione corrispondente. La situazione alla quale si è giunti è rappresentabile dallo schema in *figura 149*.

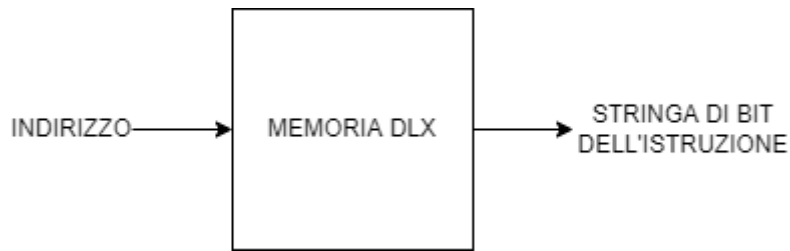


Figura 149: Rappresentazione dello schema logico della conversione da indirizzo a istruzione effettuata dalla memoria principale

Come si può notare dalla *figura 149*, lo schema è molto simile al caso delle ROM dell'RTL. La Memoria agisce da traduttore, questa volta da indirizzi a Codice Macchina. Rimane il vantaggio fornito dagli indirizzi, ottenendo una relazione matematica anche tra istruzioni diverse. Supponendo di caricare la prima istruzione all'indirizzo 0, l'istruzione successiva sarà contenuta all'indirizzo 1, e così via. La Programmabilità della Macchina di Von Neumann quindi consiste nell'Automazione del processo di prelievo sequenziale delle istruzioni in Memoria.

Risulta spontaneo a questo punto domandarsi se è possibile gestire l'automatizzazione dell'esecuzione sequenziale delle Istruzioni mediante una rete Sequenziale, come nella Rete Sequenziale dell'UDC. Ciò non è possibile perchè mentre la Rete Sequenziale dell'UDC, seppur complessa, deve gestire un numero FINITO di istruzioni, una rete sequenziale di istruzioni dovrebbe gestire qualsiasi combinazione di istruzioni possibile. Ma i programmi scrivibili in Assembly sono infiniti, come lo sarebbe quindi una rete sequenziale di questo tipo. L'unica alternativa possibile rimane dunque la medesima del Microcodice: usare un Counter con gli Indirizzi. Essendo le celle delle Memorie ordinate, la soluzione del Microcodice è implementabile infatti in maniera simile anche per le Istruzioni.

Si consideri innanzitutto lo schema logico del Data Path, riproposto in *figura 150*:

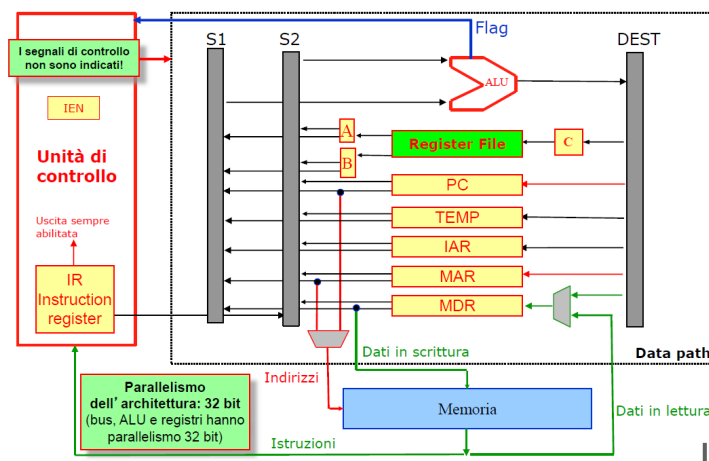


Figura 150: Schema logico della rete del Data Path (tratto dalle Slides del corso di Calcolatori Elettronici – A.A 2019)

Come visibile dalla figura, il registro PC è collegato al BUS INDIRIZZI: nel caso della Memoria del programma, specificare una istruzione equivale a specificare

l'indirizzo di una cella, quindi il registro PC, essendo in grado di emettere indirizzi, avrà la stessa funzione di indirizzo di OFFSET che veniva prodotto dal Counter nell'UDC. Per questa ragione il Counter può essere chiamato anche Micro Program Counter, a differenza di PC che invece indica il Program Counter.

Specificare un'istruzione ora quindi equivale a effettuare una Lettura dalla Memoria Principale, per poi caricare il dato in uscita dalla memoria all'ingresso dell'UDC, che verrà poi interpretato dalla COP MEMORY per dare inizio alla sequenza di microistruzioni. Questo processo di trasferimento dati è identico allo spostamento di dati tra registri introdotto all'inizio del capitolo 3: ogni cella di memoria può essere considerata come un registro. È per questo motivo che dentro l'UDC prima della COP MEMORY viene posizionato un registro che funga da REGISTRO DESTINAZIONE: IR. La COP MEMORY infatti deve rimanere attiva sulla stessa cella durante l'esecuzione dell'istruzione, e collegarla direttamente al BUS dati porterebbe a non pochi conflitti durante l'esecuzione.

Lo schema completo della situazione descritta viene mostrato in *figura 151*:

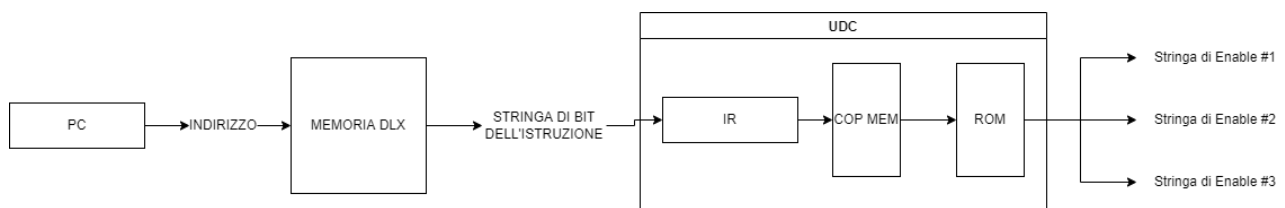


Figura 151: Visualizzazione schematizzata del processo di trasferimento del codice dell'istruzione da Memoria a UDC

Come anticipato, il Data Path è già in grado di effettuare spostamenti tra registri e tra registri e memoria, mediante i classici segnali di Enable. Tali spostamenti possono essere espressi con il codice RTL esattamente come qualsiasi altro microspostamento. La rete dell'UDC costruita finora poi è in grado di automatizzare l'esecuzione di queste Micro-operazioni grazie al Microcodice, qualsiasi essi siano. Modificando il Microcodice è infatti possibile modificare o aggiungere istruzioni.

Seguendo il principio delle Micro-operazioni, prelevare l'istruzione caricata nella cella di memoria ad indirizzo PC e spostarla in IR equivale a scrivere: $IR \leftarrow M[PC]$. Ma il codice RTL come già spiegato altro non è che una serie di Enable, quindi anche prelevare l'istruzione da memoria è gestibile da una serie di Enable, e la rete che fornisce gli Enable a tutto il sistema è sempre l'UDC con il Microcodice. Non a caso, la microistruzione $IR \leftarrow M[PC]$ corrisponde proprio al primo clock della fase di INIT, chiamato anche FASE DI IF.

Per iniziare l'esecuzione dell'istruzione quindi è sufficiente "presentare" la stringa di codice macchina all'UDC tramite il registro IR, una volta disponibile il COP quindi è

possibile procedere all'esecuzione automatica dell'istruzione, cosa che l'UDC è già progettata per fare.

L'UDC, quindi, non può iniziare l'esecuzione dell'istruzione prima di AVERE l'istruzione, di conseguenza è imperativo eseguire il prelievo dell'istruzione PRIMA dell'INIZIO di una istruzione. Essendo $IR \leftarrow -M[PC]$ una microistruzione RTL esattamente come le altre viste finora, è possibile scrivere la nuova microistruzione direttamente nelle ROM del Microcodice, come in *figura 152*:

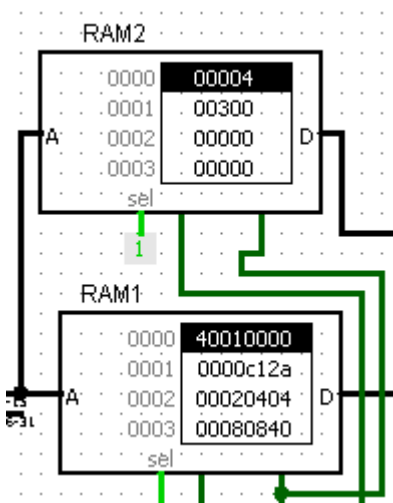


Figura 152: Aggiunta di una nuova microistruzione nelle ROM del Microcodice

	A	B	C	D	E	F
1	ir	mpc				
2	pc	pc+1	a	rs1	b	rs2
3	temp	b				
4	c	a+temp				
5	rd	c				

Figura 153: Codice RTL della nuova Microistruzione

Questa Microistruzione andrà ripetuta all'inizio di ogni istruzione. In un certo senso è l'UDC stessa ad Autofornirsi l'Istruzione che andrà poi eseguita. Questo "Autoprelievo" dell'istruzione da eseguire è un grande passo avanti verso la Programmabilità della Macchina di Von Neumann, poiché consente di "Automatizzare" il prelievo delle istruzioni introdotto all'inizio del paragrafo.

In *figura 154* viene mostrato lo stato del Data Path durante l'esecuzione della microistruzione $IR \leftarrow -M[PC]$. Affinché PC sia in grado di fornire un indirizzo, è necessario fornirgli l'OE2 più configurare il Mux indirizzi per mandare PC sul BUS indirizzi.

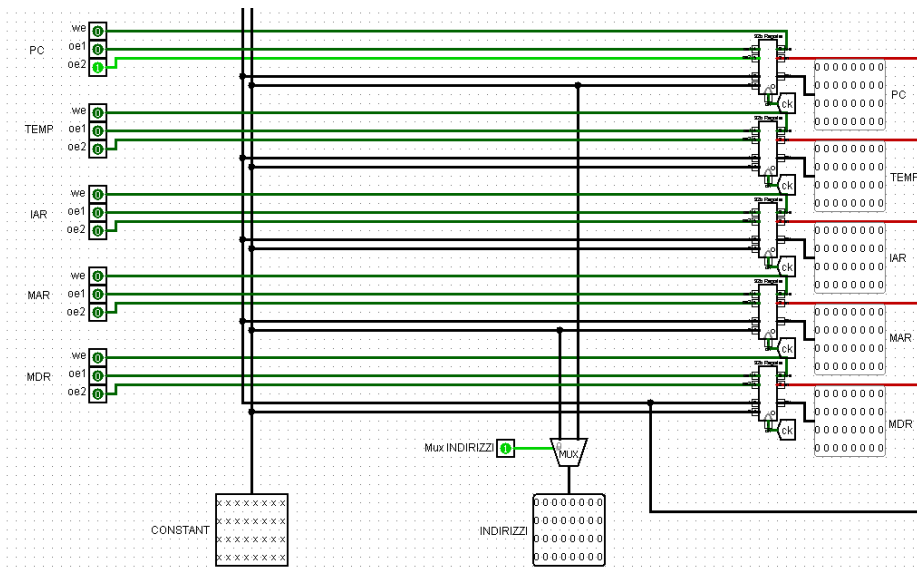


Figura 154: Output Enable del Registro PC nel Data Path

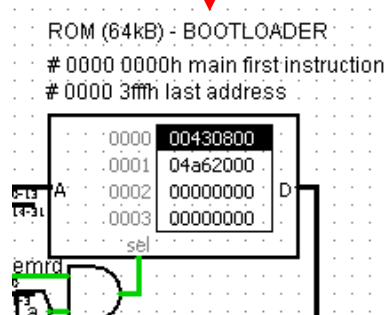


Figura 155: Zoom sulla memoria principale

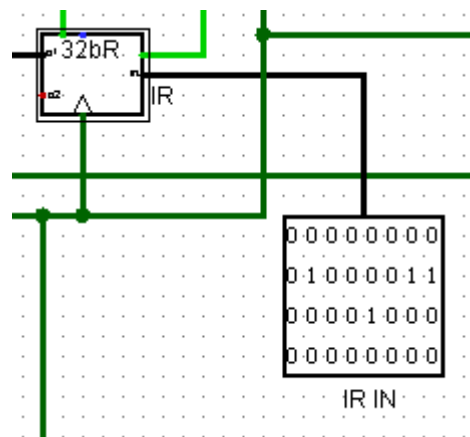


Figura 156: Zoom sul registro IR nell'UDC

In figura 155 viene mostrata la memoria principale dove è stato salvato il programma: si può notare come la cella ad indirizzo zero sia selezionata, il dato quindi ora è pronto per essere trasferito in IR (mostrato nello Zoom dell'UDC in figura 156) affinché possa iniziare l'esecuzione dell'istruzione. Naturalmente, anche

IR è pilotato da un WE, che deve essere quindi alzato per consentire il completamento del trasferimento. La situazione mostrata replica esattamente lo schema concettuale mostrato in *figura 151*.

L'UDC adesso sa da dove prelevare l'istruzione, ma allo stato attuale rimane da comprendere cosa effettivamente prelevi. Essendo il registro PC all'inizio dell'esecuzione pari a ZERO, allora verrà prelevata l'istruzione di indirizzo PC, ovvero $IR \leftarrow M[PC] = IR \leftarrow M[0]$. Una volta disponibile in IR, la COP memory selezionerà la cella della prima microistruzione della fase di EXE dell'istruzione e comincerà l'esecuzione vera e propria dell'istruzione. Alla successiva fase di INIT però, verrà eseguita la stessa prima microistruzione: $IR \leftarrow M[PC]$. Nessuno ha modificato PC, quindi l'istruzione eseguita sarà la stessa, ovvero quella ad indirizzo 0. Quello che manca stavolta è l'incremento dell'indirizzo, ovvero il lavoro che il Microprogramcounter faceva per le ROM dell'UDC. In maniera identica alla fase di INIT, è possibile risolvere il problema aggiungendo una nuova microistruzione specializzata al microcodice: $PC \leftarrow PC + 1$. Questo è l'incremento necessario per rendere possibile l'esecuzione sequenziale il programma.

Le fasi della realizzazione della Programmabilità della CPU sono quindi le seguenti:

1. Prelievo dell'istruzione dal DLX mediante $IR \leftarrow M[PC]$ (PC=0)
2. Incremento di uno di PC (PC=1)
3. Esecuzione delle microistruzioni (PC=1)
4. Prelievo dell'istruzione dal DLX mediante $IR \leftarrow M[PC]$ (PC=1)
5. Incremento di uno di PC (PC=2)

La fase di incremento di PC viene chiamata "DECODE" e mediante tale fase viene resa possibile l'esecuzione "sequenziale" delle istruzioni di un programma Assembly. In *figura 157* viene mostrato lo stato dei segnali di Enable del Data Path necessari a incrementare PC: in questo caso è necessario aggiornare lo stesso registro con un nuovo valore, andranno quindi alzati WE e OE di PC allo stesso momento, in più l'ALU andrà configurata con CI pari a uno.

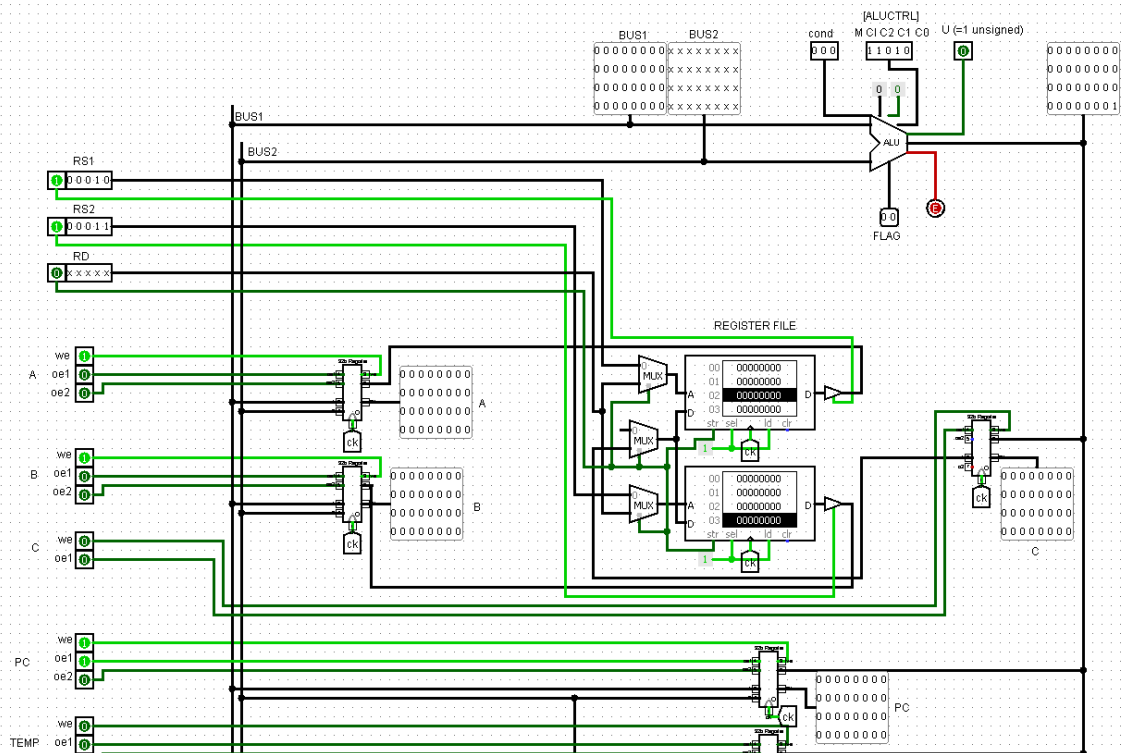


Figura 157: Segnali di Enable del Data Path necessari all'incremento di PC

Al termine di questa Microistruzione, il valore di PC verrà aggiornato da zero a uno, come mostrato in figura 158:

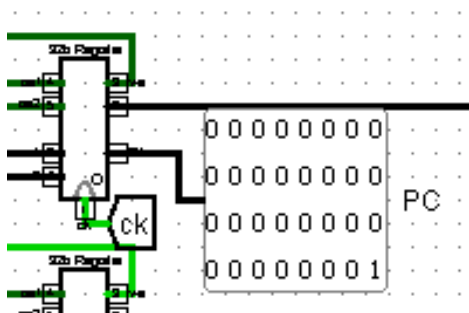


Figura 158: Zoom sul registro PC del Data Path al successivo fronte di salita

Terminata quindi poi la fase di EXECUTE dell'istruzione corrente, al colpo di clock successivo si torna alla fase di prelievo dell'istruzione, ovvero la fase di IF. Questa volta però, PC contiene il valore uno, verrà quindi selezionata la cella della Memoria principale con indirizzo uno, come mostrato in figura 159 e 160.

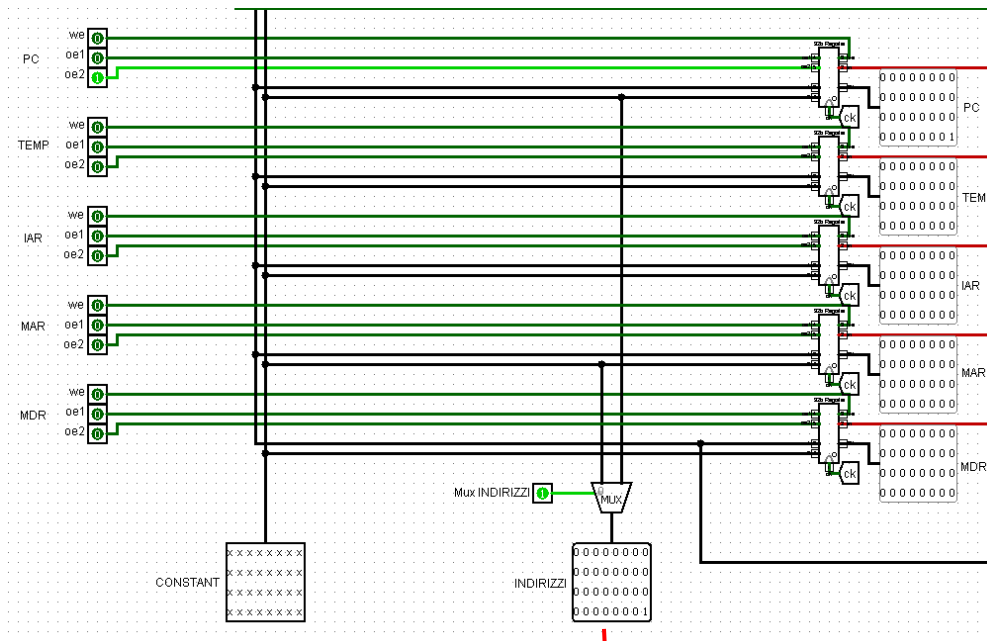


Figura 159: Output Enable del registro PC nel Data Path

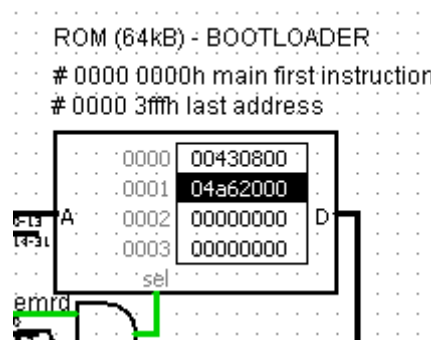


Figura 160: Zoom sulla memoria principale

Ripetendo il processo all'infinito, il valore di PC continuerà ad incrementare fino ad arrivare a overflow, senza bisogno di intervenire nel processo, prelevando quindi in maniera ordinata una istruzione alla volta dalla Memoria principale per poi eseguirla in modo automatico grazie al MicroProgramCounter. Incrementando PC di un valore diverso da uno si salterà quindi l'esecuzione di alcune istruzioni, permettendo il controllo del flusso del programma, che è lo scopo delle istruzioni di salto, come la JUMP, e di salto condizionato, come la BEQZ.

La Macchina è ora programmabile, e l'ordine delle istruzioni eseguite determina il funzionamento vero e proprio della Macchina.

L'incremento di uno di PC può essere svolto in contemporanea al prelievo degli operandi sorgente, viene quindi creato un unico stato dove vengono effettuate entrambe le operazioni. In *figura 161* viene quindi mostrato il Diagramma di Flusso completo delle Istruzioni Assembly, compreso di fase IF e DECODE appena introdotte:

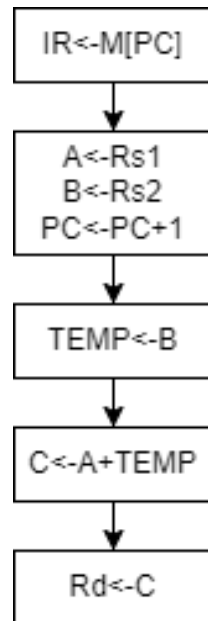


Figura 161: Diagramma di flusso completo dell'istruzione ADD

Automatizzare quindi la gestione delle Istruzioni è la definizione di Programmabilità, e per comprendere il funzionamento della Macchina di Von Neumann sono necessari entrambi i concetti. I due flussi di esecuzione mostrati in *figura 147* sono quindi intrecciati, e agiscono uno sull'esecuzione sequenziale di Istruzioni in Memoria, l'altro sull'esecuzione sequenziale di Microistruzioni nelle Ram del Microcodice. L'esempio pratico completo del funzionamento di una Macchina di Von Neumann verrà presentato nel capitolo 6, dopo aver introdotto una ultima modifica all'UDC.

3.15-Implementazione Ibrida dell'UDC mediante Master Control

La rete realizzata finora è funzionale, ma non è ancora completa. Ci sono tanti piccoli problemi che la CPU non è ancora in grado di gestire: se ad esempio una memoria impiega un unico colpo di clock per rendere disponibile il contenuto allora non ci sono problemi, ma se la memoria è lenta e impiega ad esempio 2 colpi di clock, l'UDC incrementa di 1 il Counter e continua comunque l'esecuzione, di una istruzione che però non è mai arrivata. L'implementazione a Microcodice funziona

benissimo per risolvere un particolare problema, ma perde la plasticità degli stati veri e propri di una Rete Sequenziale.

La soluzione Finale quindi per implementare l'UDC consiste in una rete ibrida, composta in parte da una rete sequenziale, e in parte dalla circuiteria necessaria a implementare il Microcodice. La Rete Sequenziale impiegata nella costruzione dell'UDC Ibrida prende il nome di Master Control, poiché è la rete con il livello di astrazione e controllo più alto nel sistema.

La rete del Master Control realizza le parti essenziali della implementazione a Rete Sequenziale dell'UDC, lasciando le parti complesse al Microcodice. Per intendersi, la Rete Sequenziale del Master Control può essere riassunta con il diagramma degli stati mostrato in *figura 162*:

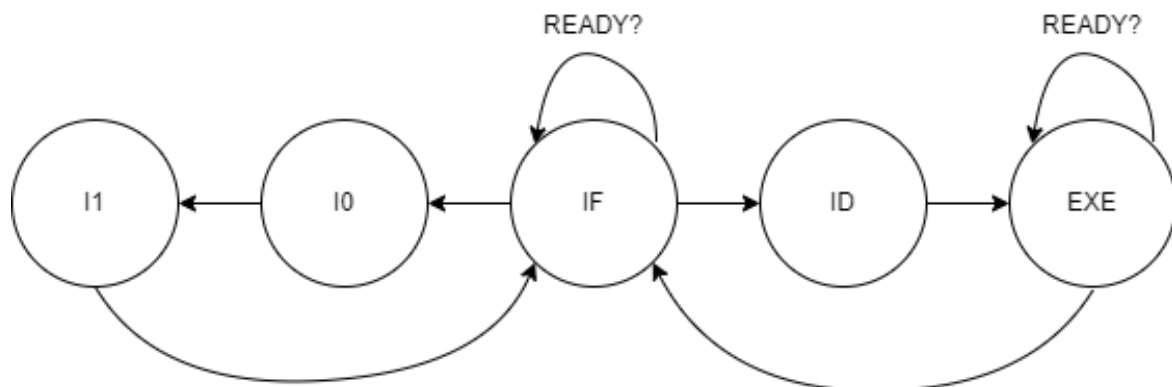


Figura 162: Diagramma degli stati del Master Control

In Logisim il Master Control è stato posizionato all'interno dell'UDC ed è stato realizzato mediante una rete sequenziale con una memoria che traduca i bit dello stato in segnali di controllo per l'UDC, mediante questi segnali di controllo personalizzati è possibile effettuare un fine-tuning del funzionamento della CPU per migliorarne il funzionamento o realizzare nuove funzionalità, nonché rendere più chiari i collegamenti all'interno dell'UDC. Lo schema del Master Control in Logisim viene mostrato in *figura 163*:

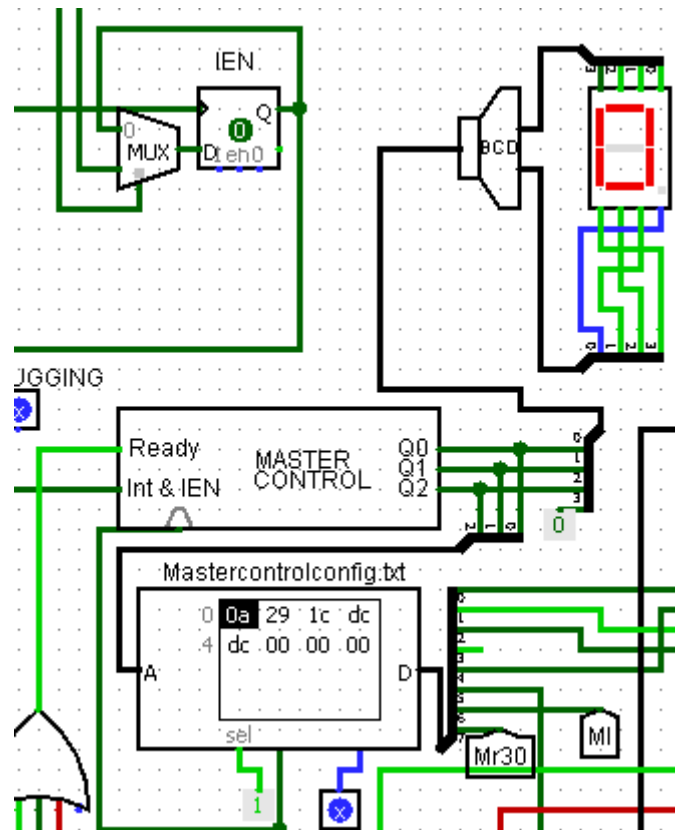


Figura 163: Zoom sulla rete di Master Control all'interno dell'UDC

Come si può notare dalla figura, ai bit di stato del Master Control è stato collegato un Pannello BCD che consentisse la visualizzazione immediata dello stato corrente della CPU: 0 corrisponde alla fase IF, 1 per la fase DECODE, 2 per la fase EXE, 3 per I0 e 4 per I1.

Il funzionamento Master Control può essere spiegato attraverso le sue tre modalità di funzionamento: Modalità INIT, Modalità EXE, Modalità INTERRUPT. La modalità INIT si occupa di portare a termine i primi due stati dell'UDC, IF e DECODE. In pratica le microistruzioni $IR \leftarrow M[PC]$, $PC \leftarrow PC+1$, $A \leftarrow Rs1$, $B \leftarrow Rs2$, invece che essere riscritte ogni volta all'inizio del microcodice di una istruzione, vengono scritte una sola volta nelle prime due celle delle ROM dell'UDC. A questo punto mediante un MUX la COP MEMORY viene temporaneamente disattivata e il Master Control prende il controllo delle ROM, selezionando "manualmente" la cella di indirizzo 0 nella fase IF, e la cella di indirizzo 1 nella fase DECODE, tramite gli Enable pilotati dalla memoria MasterControlConfig. Va aggiunto come il Master Control sia molto comodo per pilotare i MUX che controllano l'interpretazione dei registri del Register File come Sorgente o Destinazione nella fase 1. Nella fase 0, come già anticipato, è necessario attendere che la memoria sia pronta per fornire il dato. Se nel microcodice per implementare questo comportamento è richiesta diversa circuiteria aggiuntiva, al Master Control è sufficiente aggiungere una condizione di attesa nel primo stato, che

verifica lo stato del pin READY: se l'ingresso è alto allora si può procedere con lo stato successivo, come mostrato in *figura 164*, altrimenti si aspetta un altro colpo di clock.

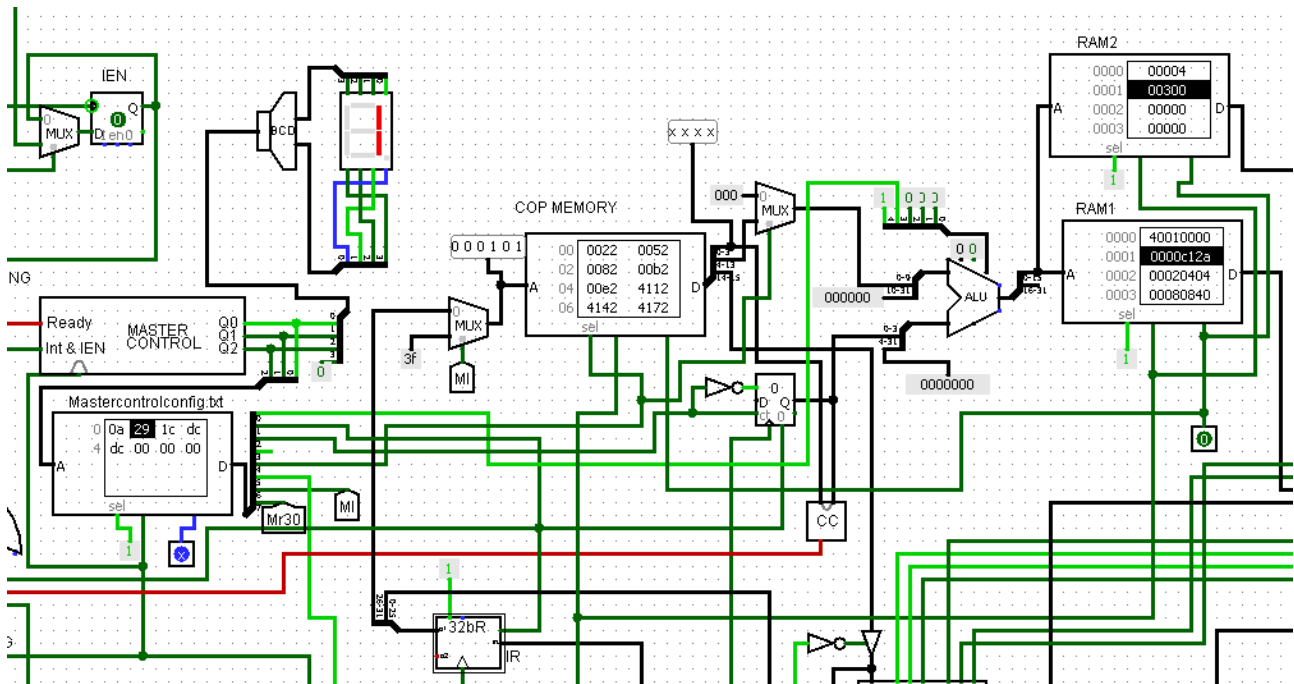


Figura 164: Stato della rete UDC dopo il segnale di Ready

Al terzo colpo di Clock, se non sono presenti eventi inattesi (INTERRUPT), la rete passa nello stato 2: la fase di EXECUTE, come mostrato in *figura 165*. Qui entra in gioco nuovamente il Microcodice: il Master Control abilita nuovamente la COP MEMORY, che adesso grazie alla fase 0 e 1 vede in ingresso il COP dell'istruzione corrente, fornito dai bit splittati di IR. Conseguentemente viene anche avviato il Counter.

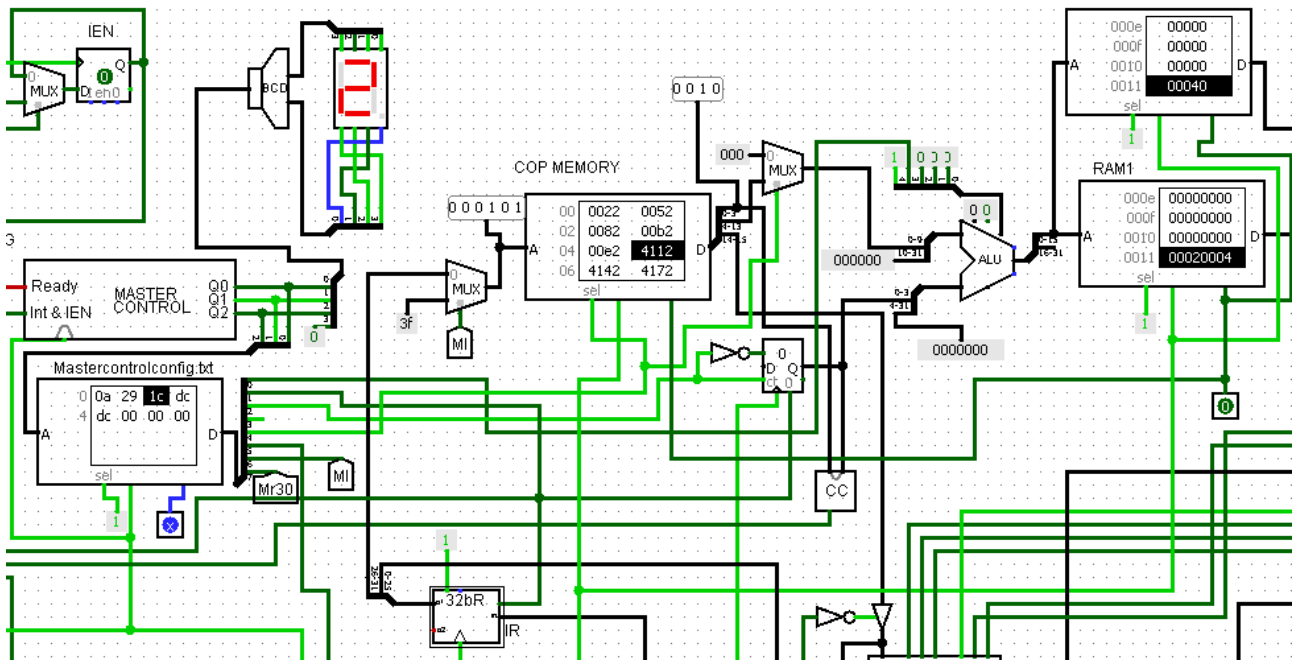


Figura 165: Stato dell'UDC durante la fase di Execute

Questa volta il reset del Counter viene lasciato al Master Control: anche nella fase 2 è infatti presente un READY, e sarà il COUNTER a fornire il READY finale della fase di EXECUTE, permettendo al Master Control di Ricominciare dalla fase INIT dell'istruzione successiva.

Per concludere, la CPU è abilitata anche all'esecuzione di INTERRUPT. Un interrupt è un evento inatteso che può essere segnalato alla CPU mediante un segnale chiamato di "Interrupt". All'arrivo di un Interrupt, la CPU deve essere in grado di gestire l'evento inatteso mediante un codice Assembly apposito, anch'esso salvato in memoria. Questo equivale ad effettuare un "salto" di PC per passare il controllo al nuovo codice chiamato "routine di Interrupt", una volta eseguito il codice sarà ovviamente necessario ripristinare il valore di PC per riprendere l'esecuzione normale del programma. L'ingresso di Interrupt per una maggiore flessibilità è collegato a un AND gate, come mostrato in figura 166, insieme a l'OUTPUT del registro IEN: caricando un uno sul registro gli interrupt sono abilitati, mentre caricando uno zero vengono disabilitati. Il controllo del registro IEN è lasciato alle istruzioni CLI e STI, sviluppate sempre in microcodice.

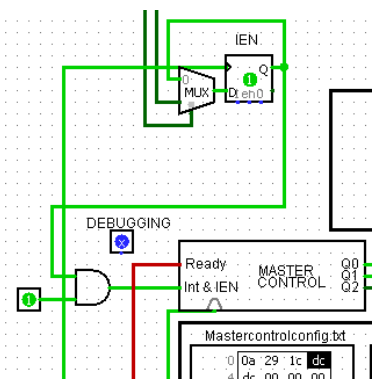


Figura 166: Collegamento di INT e IEN in ingresso al Master Control

Una volta abilitati, la "reazione" della CPU a un interrupt per passare il controllo alla routine viene mostrata in figura 167:

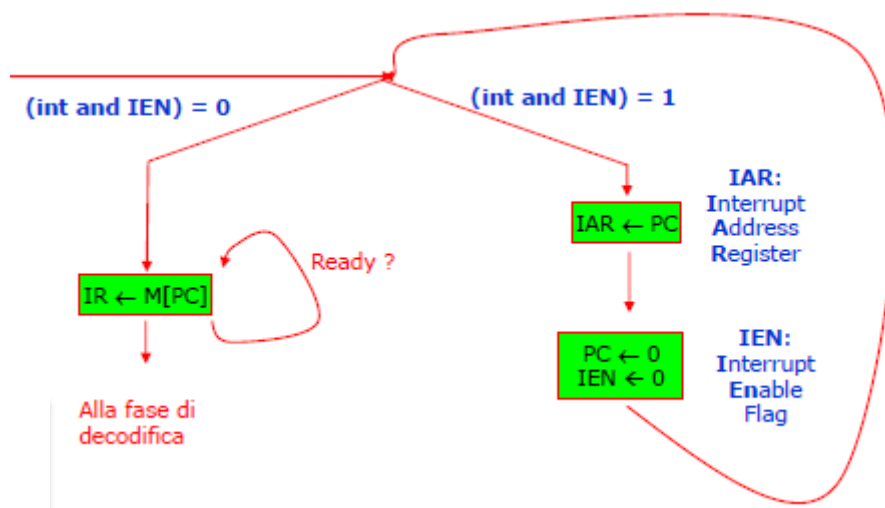


Figura 167: Schema di flusso delle Microistruzioni in caso di evento inatteso di Interrupt (tratta dalle Slides del corso di Calcolatori Elettronici – A.A 2019)

Non esiste però un unico modo di reagire a un interrupt, per consentire quindi allo sviluppatore di personalizzare il funzionamento della CPU il trasferimento del controllo alla routine di interrupt viene lasciato a una istruzione speciale, di COP 111111, che viene selezionata automaticamente dal Master Control una volta ricevuto un Interrupt. Personalizzando il codice RTL dell'istruzione nelle ROM del microcodice è possibile quindi realizzare diversi "tipi" di interrupt, come anche quello di tipo Vettorizzato. L'inizio della fase di execute dell'istruzione speciale avviene nello stato 3 dell'UDC, come mostrato in figura 168.

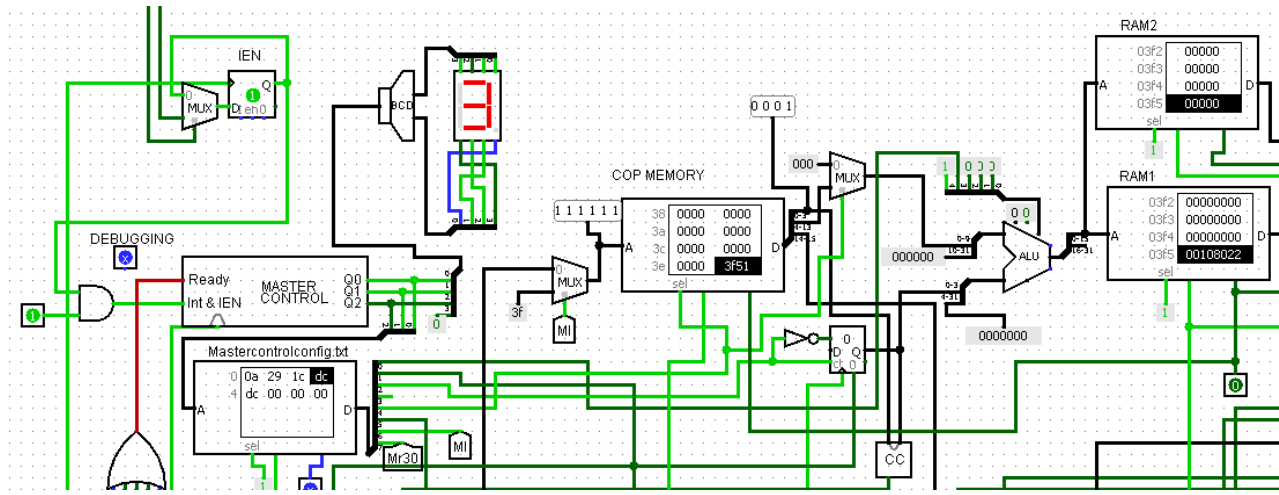


Figura 168: Stato dell'UDC all'arrivo di un segnale di Interrupt

L'UDC è ora completa ed è in grado di gestire qualsiasi aspetto della Macchina di Von Neumann, come Automatizzazione e Programmabilità, inoltre può essere implementata agilmente tramite l'implementazione a Microcodice e la rete di Master Control.

Ora non resta altro che trasformare UDC e Data Path in due sottocircuiti e unire le loro uscite in un nuovo circuito di livello superiore. Il circuito in questione è proprio

quello della CPU! In *figura 169* viene mostrato il collegamento tra i due Macroblocchi che compongono la struttura della CPU:

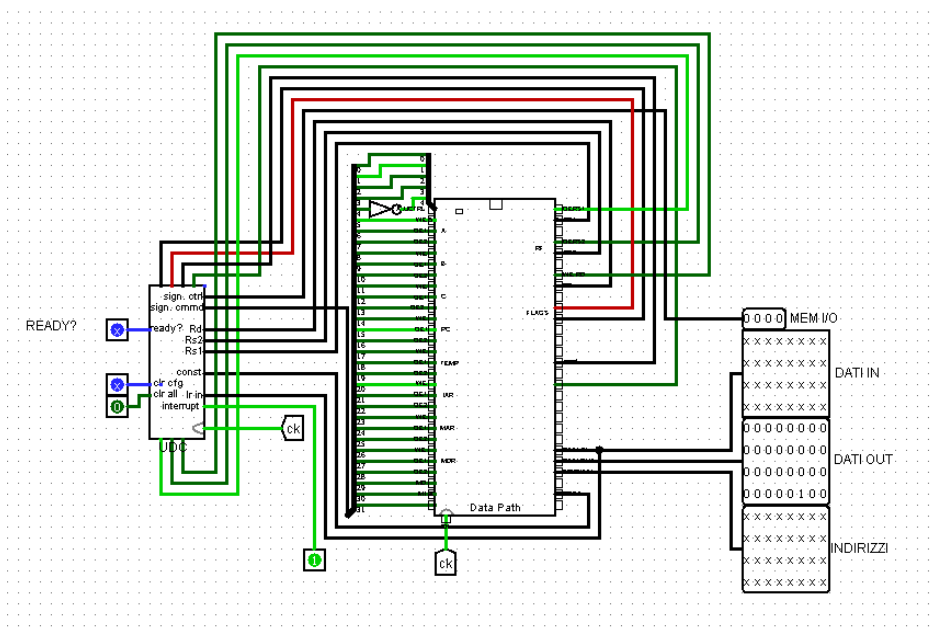


Figura 169: Collegamento delle reti dell'UDC e Data Path in Logisim

Infine, trasformando anche questo circuito in un sottocircuito, è possibile ottenere il blocco finale della CPU, come mostrato in *figura 170*:

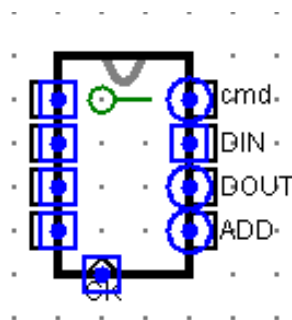


Figura 170: Simbolo del Sottocircuito della CPU realizzata in Logisim



Figura 171: La CPU di Virtual Shock è una versione semplificata di una vera CPU (www.amazon.com)

3.16-Architettura di Bus, memorie e periferiche

Ora che il “cervello” del calcolatore è completo, è il momento di connettere la CPU con la memoria dei programmi e dati e alle periferiche, in modo tale da poter visualizzare l’output dei programmi Memorizzati nella ROM principale. La disposizione dei bus in Logisim è stata scelta per consentire allo sviluppatore di aggiungere nuovi componenti in modo Modulare. Sulla linea Orizzontale è possibile disporre tutte le memorie, mentre sulla linea Verticale vengono posizionate tutte le periferiche di I/O, come mostrato in *figura 172*.

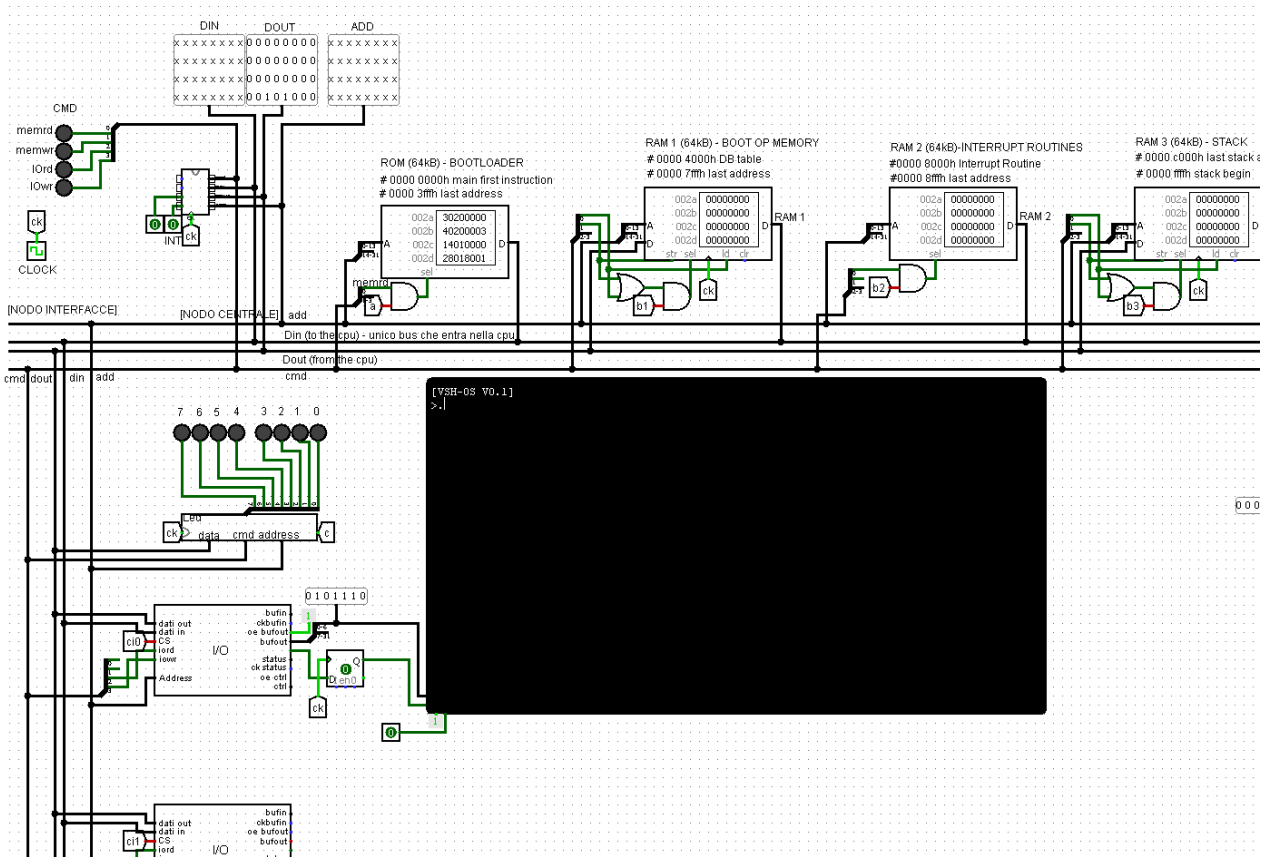


Figura 172: Schema circuitale completo della CPU collegata a periferiche e Memorie in Logisim

Come si può notare dalla *figura 172*, sono presenti 4 memorie separate da 64kB e non una unica, come il DLX della Architettura di Von Neumann. Tale disposizione consente di caricare programmi e dati in memorie separate portando notevoli vantaggi, come ad esempio la possibilità di caricare il programma sulla ROM, mentre i Dati possono essere immagazzinati nelle RAM. È stata inoltre aggiunta una memoria dedicata alle Routines di Interrupt e una per lo Stack, gestito dalle istruzioni Push e Pop.

La struttura logica di una periferica di I/O è mostrata in *figura 173*:

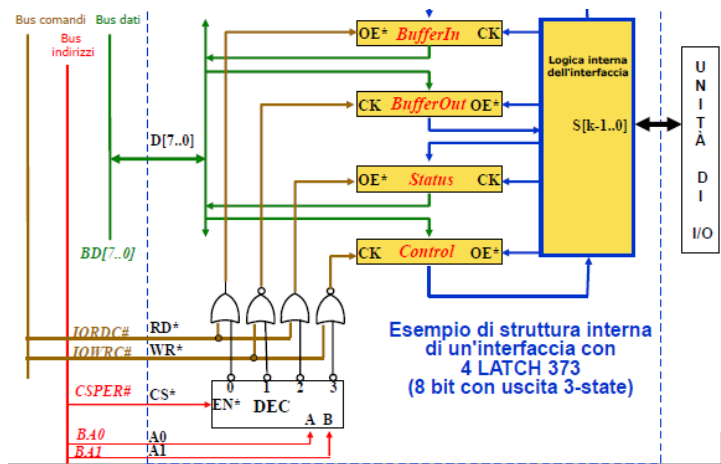


Figura 173: Struttura logica di una periferica di I/O (tratto dalle Slides del corso di Calcolatori Elettronici – A.A 2019)

La stessa struttura mostrata in *figura 173* in logisim appare come mostrato in *figura 174*.

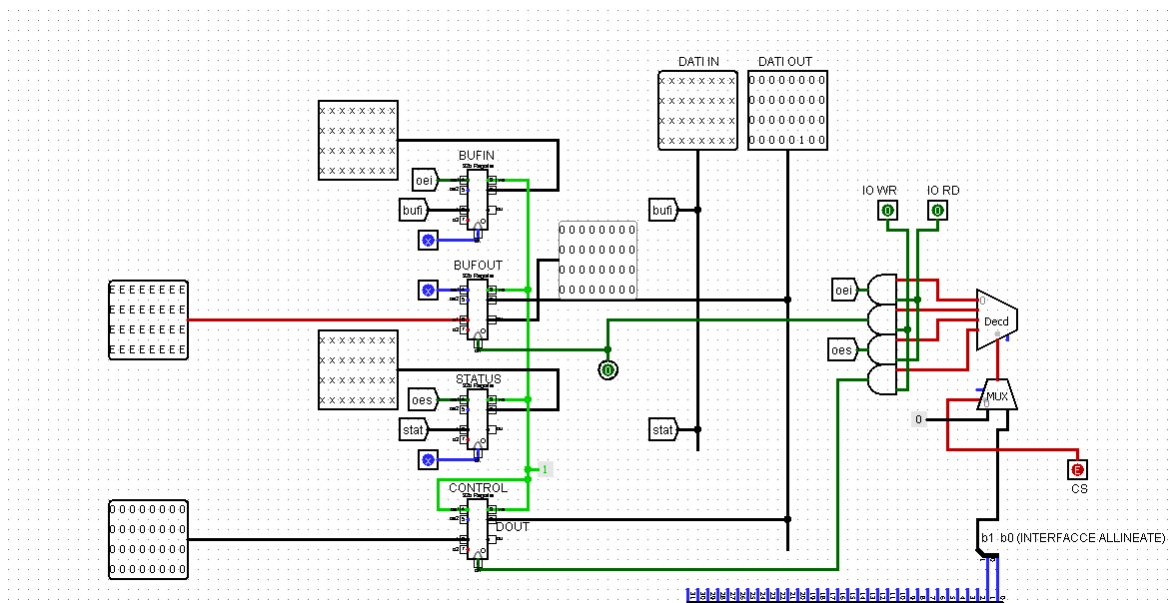


Figura 174: Schema circuitale di una periferica di I/O in Logisim

Mediante queste periferiche è possibile interfacciare una moltitudine di dispositivi al sistema. Al fine di mostrare gli OUTPUT del programma, ad esempio, è stata aggiunta una interfaccia TTY, ovvero un display in grado di mostrare solamente caratteri Ascii e non pixel, collegata a una di queste periferiche come mostrato in *figura 172*. In tal modo è possibile eseguire l'equivalente della funzione *printf* in linguaggio C. La funzione *printf* consente di stampare l'output di un programma in una interfaccia con caratteri leggibili dall'uomo, come un TTY.

Adesso non resta altro quindi che scrivere un programma, caricarlo in memoria e osservarne il funzionamento. Ed è qui che Virtual Shock ha affrontato il suo ultimo problema: la programmazione Assembly.

Un codice Assembly, sebbene possa essere scritto in linguaggio *human readable*, nella pratica consiste in una serie di zeri e uni da caricare in memoria, il Codice Macchina. Nelle ROM di Logisim vanno quindi fisicamente caricate delle sequenze di Bit, tradotte dal linguaggio Assembly. Il processo di traduzione di Codice Assembly in Codice Macchina è lungo e tedioso, e la probabilità di commettere errori è molto alta. Virtual Shock non sarebbe completo senza la possibilità di programmare comodamente la CPU, per poi consentire di sviluppare programmi sempre più complessi. È a tal scopo che è stato realizzato un sistema per tradurre automaticamente il codice umano Assembly in codice Macchina: tale sistema prende il nome di Compiler Assembly, e verrà introdotto nel prossimo capitolo.

CAPITOLO 4: Compiler

4.1-Introduzione

Come già anticipato, un Compiler è un programma in grado di tradurre un linguaggio di alto livello in un codice o linguaggio di livello più basso. Nel caso di Assembly, lo scopo del Compiler è tradurre Assembly in Codice Macchina. A differenza ad esempio del linguaggio C, il linguaggio Assembly è un codice molto “pulito” dal punto di vista visivo. Consiste infatti in una serie di istruzioni scritte in fila una di seguito all’altra. Tradurre con un algoritmo questo genere di codice è immediato. È sufficiente leggere una riga alla volta all’interno di un ciclo while, eseguire uno switch-case sulle parole della riga per tradurre il nome dell’istruzione nel COP e interpretare l’indirizzo dei registri nel modo corretto. In uscita quindi il Compiler deve restituire una serie di stringhe di bit una di seguito all’altra e corrispondenti alla codifica delle istruzioni in codice Macchina. C’è una corrispondenza biunivoca tra le righe di codice Assembly e le righe di codice Macchina, come mostrato in *figura 175*.

cli	84000000
addi r30 r0 STACKPOINTER	141effff
addi r29 r0 IVT //interrupt table//	141d8000
addi r2 r0 MAIN	14020005
sti	80000000
addi r1 r0 hm	14010048
out ttybufout(r0) r1	9c200001
addi r1 r0 e	14010065
out ttybufout(r0) r1	9c200001
addi r1 r0 l	1401006c
out ttybufout(r0) r1	9c200001
addi r1 r0 l	1401006c
out ttybufout(r0) r1	9c200001
addi r1 r0 o	1401006f
out ttybufout(r0) r1	9c200001
addi r1 r0 spazio	14010020
out ttybufout(r0) r1	9c200001
addi r1 r0 wm	14010057
out ttybufout(r0) r1	9c200001
addi r1 r0 o	1401006f
out ttybufout(r0) r1	9c200001
addi r1 r0 r	14010072
out ttybufout(r0) r1	9c200001
addi r1 r0 l	1401006c
out ttybufout(r0) r1	9c200001
addi r1 r0 d	14010064
out ttybufout(r0) r1	9c200001
addi r1 r0 !	14010021
out ttybufout(r0) r1	9c200001

Figura 175: Programma Assembly di un Hello World

Figura 176: Codice Macchina di un Hello World

Una volta quindi eseguita la compilazione, il codice Macchina è pronto per essere caricato direttamente sulle ROM di Logisim, tenendo presente che le Memorie possono essere scritte “manualmente” usando unicamente fogli di testo.

Anche nel caso dell’RTL è possibile realizzare un Compiler, che si occuperà di trasformare RTL in Microcodice. In questo caso in ingresso non si ha un file di testo bensì una matrice di celle contenenti i nomi dei registri per codificare Microoperazioni, e in uscita dopo la compilazione verrà reso disponibile un foglio di testo, come nel caso Assembly, questa volta però contenente il Microcodice da caricare sulle ROM dell’UDC.

Il Compiler sostanzialmente deve lavorare tra matrici di righe o celle, è stato scelto quindi come linguaggio di programmazione dei Compiler il linguaggio Matlab, per la sua capacità di gestire matrici di grandi dimensioni con elevata velocità. Matlab inoltre fornisce un’interfaccia grafica intuitiva: con la possibilità di visualizzare il contenuto delle variabili direttamente dal WorkSpace permettendo un debugging rapido ed efficace.

4.2-File System

Entrambi i Compiler si basano quindi sullo stesso principio, ovvero di tradurre un file in ingresso in un file di uscita scritto in un altro linguaggio. È necessario poi quindi un passaggio aggiuntivo per caricare il risultato direttamente nelle memorie Logisim, come mostrato in *figura 177*:

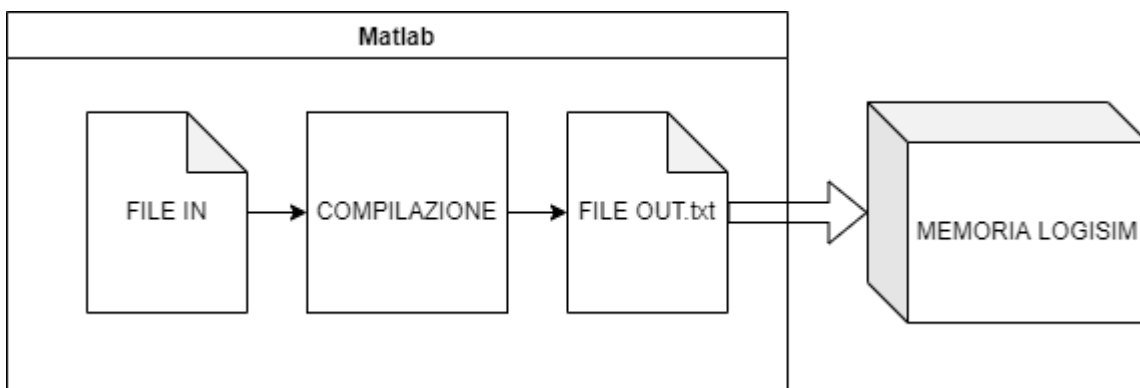


Figura 177: Schema logico del flusso di compilazione mediante compiler Matlab

L’organizzazione del File System in entrambi i Compiler è quindi molto simile: in un’unica cartella vengono contenuti tutti i Files Sorgente, compresi script, funzioni, files in e files out. In *figura 178* viene mostrata la cartella del Compiler Assembly, contenente lo script “COMPILER_ASS_BIN_v2.m” e il file DLX.txt, dove andrà scritto il codice Assembly in versione Umana.

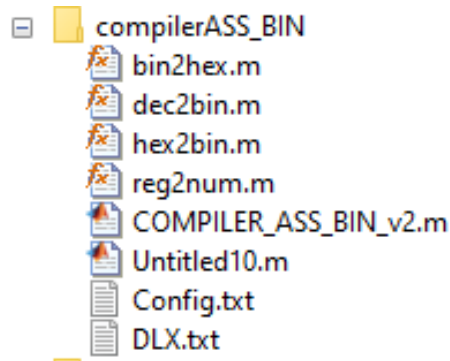


Figura 178: Cartella del Compiler Assembly

In figura 179 viene invece mostrata la cartella del Compiler RTL. In maniera simile ad Assembly, in essa è contenuto il file in ingresso, questa volta Excel, e lo script “COMPILER_RTL_MC.m”

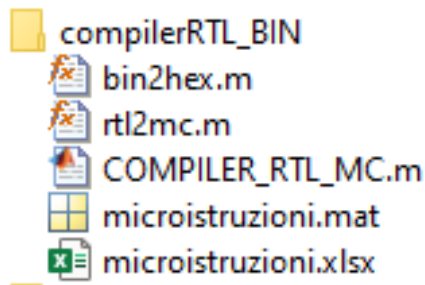


Figura 179: Cartella del Compiler RTL

4.3-Compiler ASSEMBLY

Una volta aggiunta la cartella compilerASS_BIN alla PATH di Matlab, è possibile cominciare a lavorare con il compiler. Il codice assembly può essere scritto direttamente dentro il file “DLX.txt”, oppure il file può essere aperto direttamente dalla cartella di lavoro in matlab, in tal caso il file verrà visualizzato all’interno dell’interfaccia matlab e salverà automaticamente e istantaneamente qualsiasi modifica. Per rendere la scrittura e visualizzazione del codice più agile, è stata aggiunta la possibilità per il Compiler di ignorare i commenti, che vanno delimitati dalla doppia sbarra all’inizio e alla fine del commento, ad esempio *//questo è un commento//*. Il codice, ad eccezione dei commenti, va scritto in MINUSCOLO.

Una volta scritto e salvato il codice, basta eseguire lo script COMPILER_ASSEMBLY_CM.m e il codice tradotto verrà salvato nel file di destinazione “DLXBINARIO.txt”. Durante l’esecuzione del Compiler verrà mostrato in output nella Command Window il Log dell’operazione (insieme di informazioni che descrivono il risultato delle singole operazioni svolte dall’algoritmo), come mostrato in figura 180.

```

[EXE] COMPILAZIONE INIZIATA...
[EXE] lettura istruzione CLI
[EXE] lettura istruzione IMMEDIATE
[EXE] lettura istruzione IMMEDIATE
[INFO] reading comment..
[INFO] comment read
[EXE] lettura istruzione IMMEDIATE
[EXE] lettura istruzione STI
[EXE] lettura istruzione IMMEDIATE
[EXE] lettura istruzione OUT
[EXE] lettura istruzione IMMEDIATE
[EXE] lettura istruzione OUT
[EXE] lettura istruzione IMMEDIATE

```

Figura 180: Output nella Command Window di Matlab

Queste informazioni sono molto utili per il debugging del codice.

4.4-Sintassi e Compilazione nel Compiler Assembly

All'inizio del file .txt vanno inserite le direttive #include <>; la direttiva base per il funzionamento è “#include <stdass>”, che contiene il compilatore base; è presente anche la direttiva “#include <stdequ>”, che consente di usare la direttiva EQU per creare variabili interne al compilatore e semplificare la scrittura del codice.

Le costanti in Assembly infatti possono essere scritte direttamente ad esempio nelle istruzioni immediate, ma nel caso in cui una istruzione venga ripetuta più volte è necessario copiare la costante su tutte le istruzioni. Modificare il valore della costante implica quindi modificare tutte le istruzioni che ne hanno fatto uso, rallentando la programmazione. È a tal scopo che sono state introdotte le direttive EQU: sono l'equivalente del #DEFINE in C e permettono di sostituire una parola a un valore, prima di effettuare la compilazione vera e propria. Scrivendo una direttiva equ con la forma NOME equ VALORE verrà aggiunto quindi il termine “NOME” ad una lista di parole speciali, e ogni volta che il compilatore incontrerà durante la compilazione la parola NOME essa verrà sostituita e tradotta in codice macchina nel VALORE corrispondente. Nel Compiler Assembly una direttiva EQU deve essere scritta all'interno del BLOCCO EQU, delimitato da “read” e graffe mostrato in *figura 181*:

```

//DIRETTIVE EQU//
read{
ledctrl equ 8000h
ledbuf equ 8001h
LEDOFFSET equ 3h
MAIN equ 5h
IVT equ 8000h
STACKPOINTER equ ffffh
ttybufout equ 0001h
ttybufctrl equ 0003h

```

Figura 181: Esempio di scrittura del Blocco EQU

Una direttiva equ è quindi STATICA, ovvero non può cambiare valore durante l'esecuzione del programma.

Allo stesso modo di una direttiva equ, le istruzioni vanno scritte nel blocco MAIN, delimitato da "\$main" e graffe, come mostrato in *figura 182*:

```
$main{  
  
cli  
addi r30 r0 STACKPOINTER  
addi r29 r0 IVT //interrupt table//  
addi r2 r0 MAIN  
sti  
  
addi r1 r0 hm  
out ttybufout(r0) r1  
addi r1 r0 e  
out ttybufout(r0) r1  
addi r1 r0 l  
out ttybufout(r0) r1  
addi r1 r0 l  
out ttybufout(r0) r1|
```

Figura 182: Esempio di scrittura del Blocco Main

Nel caso di istruzioni di tipo I o J la costante può essere scritta sia in binario che in esadecimale, nel secondo caso però va usato il suffisso 'h'

Es. subi r15 r16 0011 1110

subi r15 r16 3eh

Due istruzioni da trattare in maniera particolare sono la STORE e la LOAD. Esse consentono di eseguire operazioni di trasferimento in o da Memoria specificando un indirizzo e un valore. La Memoria però è libera da usare e non c'è nessuno che dica allo sviluppatore se una certa cella di memoria è libera o è già stata occupata da un altro programma. Lo spazio di indirizzamento in memoria per i dati va quindi calcolato a mano, e questo può portare a degli errori. È a tal scopo che sono quindi state introdotte le direttive DB: scritte nella forma NOME db DIMENSIONE nel blocco DB, permettono di definire variabili, e quindi preallocare spazio in memoria, con gli indirizzi calcolati direttamente dal Compiler. Si consideri ad esempio di voler preallocare lo spazio per una variabile che occupi 10 celle in Memoria. Supponendo di specificare il primo indirizzo libero in Memoria al compiler (tramite file config.txt), ad es 4000h, allora la variabile occuperà le celle da indirizzo 4000h a 4009h. La prima cella libera è ora 400ah, e quindi la variabile successiva potrà essere salvata da 400ah in poi. In maniera simile allo Stack Pointer, l'indice dell'ultima cella

libera viene sempre incrementato del valore di occupazione in celle della variabile precedente. Dal punto di vista pratico una direttiva DB viene trattata in maniera simile alle EQU. La parola NOME viene salvata in un elenco di parole speciali diverso dalla EQU, e se il compiler trova la parola nel codice viene sostituita con un valore, che stavolta è proprio il primo indirizzo dell'area di memoria occupata dalla variabile.

Ora che tutte le caratteristiche del compiler sono state introdotte, è possibile scrivere il codice, compilarlo e caricare il file di uscita DLX_MC.txt nella Memoria DLX in Logisim, collegata direttamente alla CPU, come mostrato in *figura 183*:

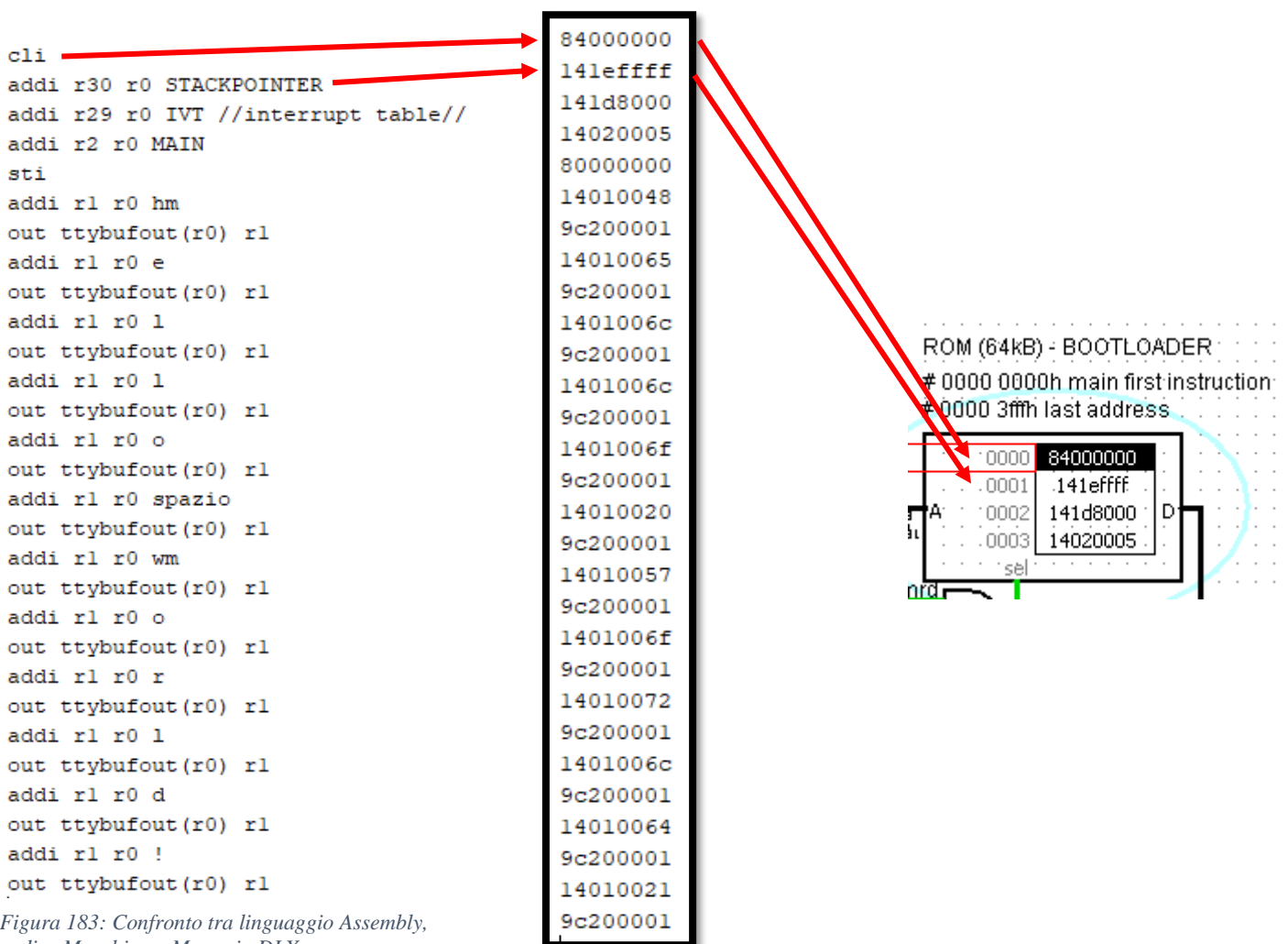


Figura 183: Confronto tra linguaggio Assembly, codice Macchina e Memoria DLX

4.5-Compiler RTL

Le ROM presenti nell'UDC invece contengono il codice macchina necessario per tradurre ogni fase di una istruzione in microistruzioni RTL.

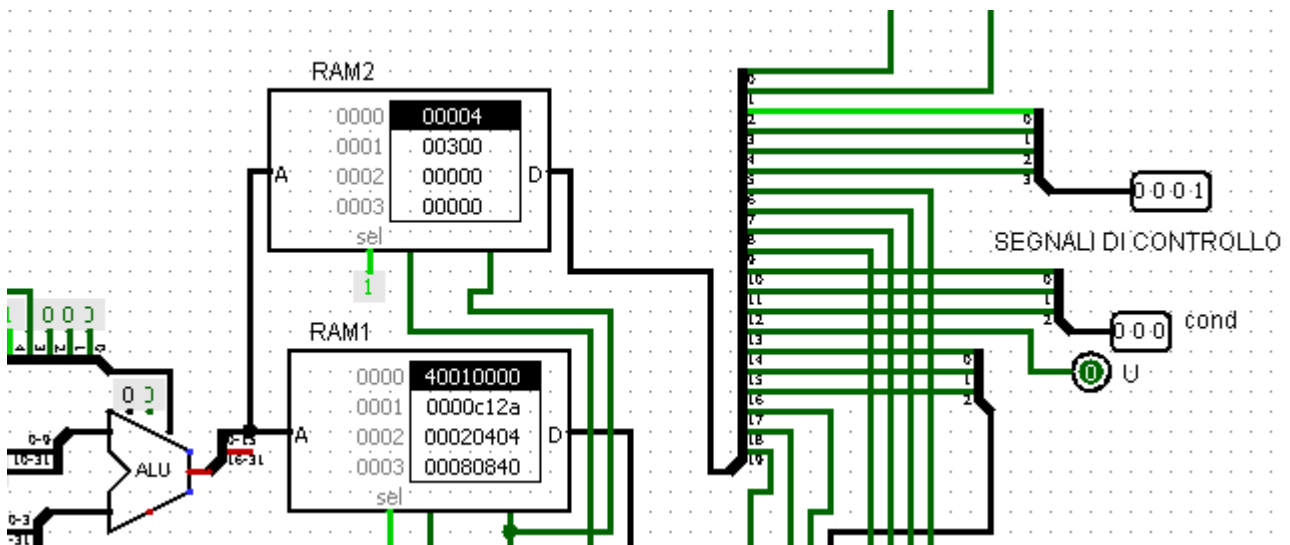


Figura 184: Zoom sulle ROM del Microcodice nell'UDC

Si ricorda che ogni microistruzione consiste in una serie di Write Enable e Output enable che pilotano gli spostamenti di dati tra registri nel data path. Il codice RTL rende più leggibile questa serie di spostamenti.

Tuttavia, nelle ROM vanno comunque caricate delle stringhe di codice binario, e la conversione da RTL a stringhe di 49 enable risulta essere un processo lungo e tedioso. Si è scelto quindi di programmare un compiler RTL nel linguaggio MATLAB per aiutare uno sviluppatore ad aggiungere nuove istruzioni eseguibili per la CPU.

Allo stesso modo di Matlab, dopo aver aggiunto alla Path la cartella "compilerRTL_BIN" è subito possibile cominciare a lavorare con il compiler.

4.6-Sintassi e Compilazione nel Compiler RTL

La struttura a celle del programma EXCEL ricorda molto le celle di una memoria, per questo motivo è stato scelto per scrivere istruzioni RTL che verranno poi convertite in celle con codice macchina.

Nella colonna A vanno inseriti i nomi dei registri destinazione, mentre nella B vanno inseriti i nomi dei registri sorgente. Il nome del registro rimane lo stesso nel caso in cui sia sorgente o destinazione, se si trova a sinistra il compiler tradurrà la parola con un WE del registro corrispondente, mentre se si trova a destra il registro verrà tradotto con un OE.

Ogni riga corrisponde a uno spostamento tra registri e quindi a una microistruzione.

La logica dell'UDC consente di eseguire le microistruzioni contenute in questa ram una alla volta sequenzialmente, in modo tale da portare a termine una istruzione.

	A	B	C	D
51	pc	pc+cost		
52	temp	b		
53	r	a==temp		
54	c	ris		
55	rd	c		
56	temp	b		
57	r	a!=temp		
58	c	ris		
59	rd	c		
60	temp	b		
61	r	a<temp		
62	c	ris		
63	rd	c		
64	temp	b		
65	r	a>temp		
66	c	ris		
67	rd	c		
68	temp	b		
69	r	a<=temp		
70	c	ris		
71	rd	c		
72	temp	b		
73	r	a>=temp		
74	c	ris		
75	rd	c		
76	temp	cost		
77	r	a==temp		

Figura 185: esempio di foglio EXCEL contenente le Microistruzioni da caricare nelle ROM del Microcodice

Una volta quindi scritto il foglio EXCEL come in *figura 185*, eseguendo lo script “COMPILER_RTL_MC.m” verranno prodotte in Output le stringhe di segnali di Enable che compongono il Microcodice: in maniera simile ad Assembly, il compiler legge ciclicamente tra le righe, valutando le due parole contenute nella singola riga, per poi eseguire uno switch case sulle parole e scrivere gli enable corrispondenti. L’output è composto da due files, che andranno caricati singolarmente sulle due ROM dell’UDC, in maniera identica al compiler Assembly.

CAPITOLO 5: ESEMPIO PRATICO DI PROGRAMMAZIONE DELLA CPU

5.1-Flusso di Progetto

Nei successivi paragrafi verranno mostrati i passaggi necessari per progettare una nuova istruzione denominata “BEQZ” e poi usarla nel compiler Assembly. Non essendo stata ancora implementata nell’UDC, l’istruzione andrà aggiunta modificando:

1. Struttura hardware dell’UDC
2. Contenuto della Cop Memory
3. Microcodice
4. Compiler Assembly

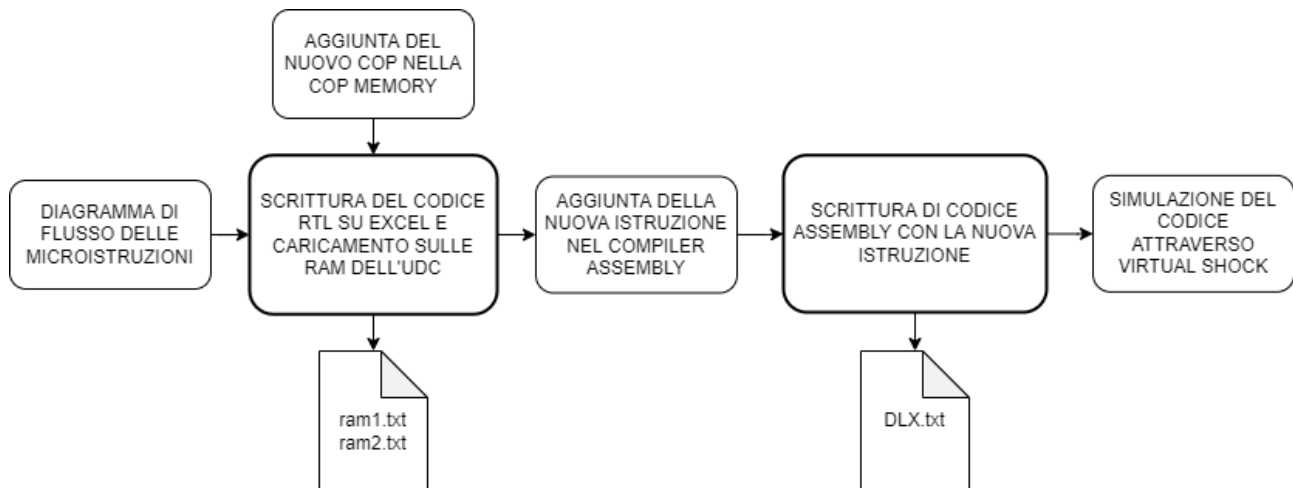


Figura 186: Flusso di progetto di un programma Assembly con aggiunta di una nuova istruzione

Una volta aggiornato il sistema, sarà quindi possibile utilizzare la nuova istruzione nel codice di un programma Assembly, che dopo essere stato scritto potrà essere caricato nella ROM del DLX. A scopo illustrativo verrà quindi mostrato un esempio di programma Assembly che riesce a far lampeggiare una periferica LED: il programma “BLINK”.

5.2-Progettazione dell’istruzione: microistruzioni

Come discusso nel Capitolo 3, il Data Path è una autostrada di informazioni, le microistruzioni altro non sono che le “direzioni” degli spostamenti fra registri.

Grazie all'ALU, un dato può non solo essere spostato, ma anche essere modificato.

Una istruzione è composta da un insieme di microistruzioni eseguite sequenzialmente, il suo funzionamento può essere descritto in maniera intuitiva da un diagramma particolare chiamato Diagramma di Flusso: un esempio di diagramma di flusso per la BEQZ viene mostrato in *figura 187*.

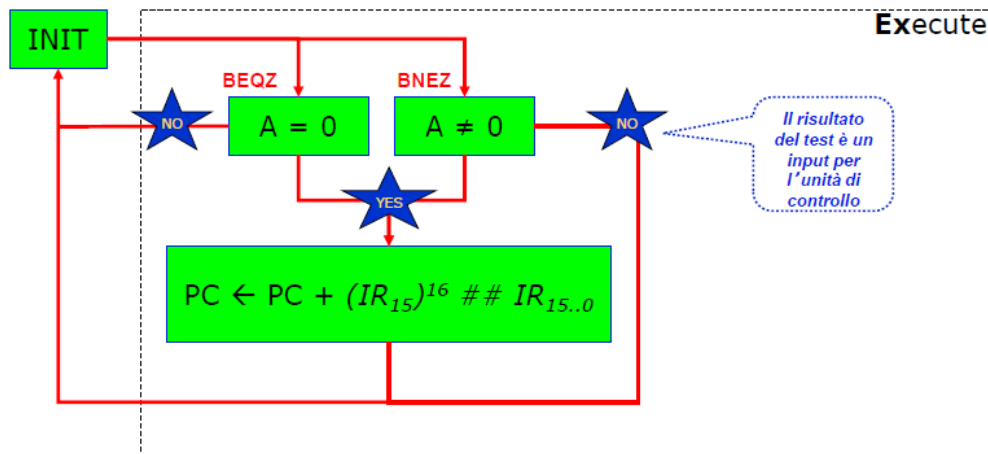


Figura 187: Diagramma di Flusso della istruzione BEQZ (tratto dalle Slides del corso di Calcolatori Elettronici – A.A 2019)

La fase INIT è uguale per tutte le microistruzioni: $IR \leftarrow M[PC]$, $PC \leftarrow PC + 1$, $A \leftarrow Rs1$, $B \leftarrow Rs2$.

Subito dopo si ha una operazione di confronto: nel caso della BEQZ se nel registro A non è contenuto uno 0 allora non dobbiamo fare nulla, altrimenti viene fatta una operazione di JUMP, modificando il valore di PC.

Per quanto riguarda il confronto, la ALU dispone già di circuiteria aggiuntiva che la rende in grado di effettuare tali operazioni.

A questo punto però ci si potrebbe chiedere come scegliere una microistruzione tra due da eseguire. L'UDC è progettata per eseguire microistruzioni sequenzialmente, ma non può fare questo tipo di decisione.

È necessario quindi modificare l'Hardware dell'UDC per eseguire questo nuovo tipo di istruzione.

5.3-Modifica HARDWARE UDC

Una possibile implementazione di una istruzione di questo tipo è la seguente: innanzitutto nella RAM dell'RTL si scrive l'istruzione completa, quindi con anche $PC \leftarrow PC + COST$. Ora l'UDC sarebbe anche in grado di eseguire l'istruzione, ma solo nel caso in cui $A=0$. Se $A=1$ il flusso sarà lo stesso e avremo un errore nell'esecuzione del software. Il problema però adesso si è ridotto a se eseguire o meno l'ultima microistruzione del blocco.

Questo problema può essere risolto abbastanza facilmente: una volta entrata nella fase di execute, la rete sequenziale del Master Control è progettata per tornare allo stato iniziale, quindi di fetch, una volta ricevuto un segnale di READY. Non importa da chi proviene, quando il segnale READY viene portato a 1, la CPU pensa di aver finito l'istruzione. Gli input del Master Control sono mostrati in *figura 188*.



Figura 188: Zoom sul Master Control

Normalmente il segnale READY viene mandato dal contatore quando raggiunge il numero di CPI dell'istruzione, segnalando la fine della fase di EXECUTE. Questo segnale può però essere sovrascritto: se arriva un ready da qualsiasi altra fonte prima che l'esecuzione finisca, la fase di EXECUTE viene “tagliata”. Il segnale di ready viene fatto provenire dallo ZERO FLAG: ogni volta che un dato passa dalla ALU, lo zero flag viene automaticamente calcolato: se il risultato è zero, viene alzato il flag, e viceversa. Il flag è un segnale binario composto da un solo bit, è adatto quindi ad essere collegato direttamente al ready del Master Control.

Quindi: quando lo zero flag è a 1, allora $A=0$ e dobbiamo eseguire l'ultima microistruzione, altrimenti la tagliamo e passiamo all'istruzione successiva.

Rimane un problema in questa implementazione: lo zero flag viene calcolato sempre, anche durante istruzioni diverse dalla BEQZ. Questa è una condizione pericolosa, poiché il “taglio” della fase di EXECUTE potrebbe accadere durante l'esecuzione di una qualsiasi istruzione, non solo della BEQZ. È a questo punto viene usata la ROM2 dell'UDC.

Questa ROM contiene principalmente i segnali di controllo usati per implementare istruzioni complesse, diverse da ADD o SUB. Questi segnali non pilotano soltanto il Data Path, ma anche alcuni componenti interni all'UDC. Nella *figura 189* viene mostrato il pinout della ROM2.

16	15	14	13	12	11	10	9	8
SET	BR	MUX	U	CO2	CO1	CO0	RX	RY
	7	6	5	4	3	2	1	0
	RZ	CEN	IOWR	IORD	MEMWR	MEMRD	IEN	WEIEN

Figura 189: Mappatura dei segnali di Controllo della ROM 2

Possiamo vedere tra i 16 segnali di controllo in posizione 15 il segnale “BR”, mentre in 14 il segnale “MUX”. Questi due segnali aiuteranno a portare a termine correttamente una istruzione di branch senza dare problemi alle le altre istruzioni. Il segnale BR è molto importante, poiché rende lo ZERO FLAG trasparente o meno attraverso l’uso di un MUX, chiamato MUX BRANCH MODE.

Nell’entrata corrispondente a 1 arriva il risultato dello ZF, mentre in 0 è presente uno 0 costante, mutando sostanzialmente il valore dello ZF. Lo stesso comportamento è ottenibile con un AND. Quindi quando il segnale “BR” è a zero, l’UDC funziona normalmente.

All’alzarsi di BR l’UDC entra in BRANCH MODE e il Master Control diventa “sensibile” allo zero flag. Adesso l’UDC è in grado di eseguire una microistruzione di confronto e salto per la BEQZ. Ma per quanto riguarda la BNEQZ?

Sostanzialmente le due istruzioni sono uguali, ma il confronto avviene in maniera inversa: per la BNEQZ se $A \neq 0$ allora si taglia, altrimenti si esegue l’ultima istruzione. Per distinguere tra le due istruzioni è sufficiente invertire il valore dello ZF, comportamento ottenuto aggiungendo un secondo mux, denominato MUX di BRANCH.

Quando il segnale “MUX” è a 0, viene preso il valore nottato dello ZF, mentre per 1 viene preso il valore dello ZF senza modifiche.

Ora l’UDC è in grado di eseguire le microistruzioni per implementare la BEQZ e BNEQZ, ma prima di eseguire l’intera istruzione bisogna scrivere il codice RTL all’interno delle RAM, come verrà illustrato nel paragrafo successivo.

5.4-Modifica del COMPILATORE

Sebbene le RAM1 e 2 possano essere scritte “a mano”, nella pratica è facilmente verificabile come questo processo sia lungo e difficile.

Inoltre scrivere queste parti di codice a mano comporta un notevole aumento della probabilità di errori di scrittura, rendendo lo sviluppo un vero e proprio incubo.

Per questo motivo viene impiegato l'uso del Compiler scritto in linguaggio Matlab, descritto nel Capitolo 4.

Essendo la fase di INIT uguale per tutte le istruzioni, questa fase viene gestita dal Master Control e non è necessario ripetere ogni volta il codice RTL necessario per l'esecuzione dell'INIT.

Nelle RAM viene quindi scritto il codice relativo alla fase di EXECUTE.

Nel caso di una istruzione BEQZ il codice è abbastanza semplice: dobbiamo effettuare il confronto e poi incrementare o meno il PC.

Sul foglio EXCEL il codice RTL avrà l'aspetto mostrato in *tabella 190*.

	A=0
PC	PC+COST

Figura 190: esempio di microistruzioni della fase di execute della EXE

Adesso verrà mostrato come il compilatore è in grado di riconoscere che quella appena vista è una istruzione di branch.

Normalmente in RTL le operazioni di confronto si fanno tra il registro A e B, o tra A e una costante COST, se quindi nell'RTL troviamo un confronto con l'operatore 0, siamo sicuri che stiamo facendo una operazione di branch.

Per effettuare la prima microistruzione quindi viene usato il codice presentato di seguito, che alza i bit corrispondenti agli enable e mux branch.

```
%USO IL ZF
p1=15; %branch mode on
p2=ctrlwide-p1;
machinecontrol(i,p2)=1;
p1=14; %mux branch on =
p2=ctrlwide-p1;
machinecontrol(i,p2)=0;
if (secondword=='!')
p1=14; %mux branch on !=
p2=ctrlwide-p1;
machinecontrol(i,p2)=1;
end
```

La seconda riga invece è una normale operazione tra registri, il compilatore è già equipaggiato per gestire questo tipo di microistruzioni.

5.5-Scrittura e compilazione del codice RTL

Si può ora procedere a scrivere le nuove microistruzioni sul foglio EXCEL.

Sorge però spontanea una domanda: in quale punto della RAM vanno posizionate le microistruzioni? La posizione è arbitraria o vanno scritte in delle celle precise? La posizione delle microistruzioni non è arbitraria e vanno scritte secondo un criterio preciso, dipendente dalla topologia dell'UDC.

Nel caso di Virtual Shock, il funzionamento dell'UDC è stato affrontato nel Capitolo 3: come anticipato, la selezione della microistruzione corrente da eseguire è affidata a una memoria a valle delle due RAM, denominata COP MEMORY, e al counter associato al sommatore, che consente di eseguire le microistruzioni sequenzialmente.

Le microistruzioni all'interno delle RAM sono ordinate secondo il seguente criterio:

- Dall'indirizzo 02h a 10h sono contenute le principali istruzioni REGISTER, come ADD,SUB,...
- Da 11h a 1fh istruzioni IMMEDIATE come ADDI,...
- STORE e LOAD occupano da 20h a 26h
- Da 27h a 2eh istruzioni di JUMP
- BEQZ e BNEQZ occupano da 2fh a 32h
- Le operazioni di confronto vanno da 33h a 62h
- PUSH e POP da 63h a 69h
- STI, CLI, RFE da 6ah a 6eh

Quindi lo spazio riservato per la BEQZ ad esempio va da 2fh a 30h, ovvero 2 celle, che è tutto il necessario per gestire la fase di EXECUTE dell'istruzione.

Ora si ha tutto il necessario per riempire il foglio EXCEL delle microistruzioni con le microistruzioni delle nostre nuove istruzioni: BEQZ e BNEQZ. La realizzazione viene mostrata in *figura 191*.

48	r	a==0
49	pc	pc+cost
50	r	a!=0
51	pc	pc+cost

Figura 191: Microistruzioni delle fasi di EXE delle istruzioni BEQZ e BNEQZ

Possiamo verificare il corretto posizionamento della microistruzione considerando che: 2fh è l'indirizzo della prima microistruzione di BEQZ.

2fh= 101111= indirizzo 47

Considerando che excel comincia a contare da 1, ci aspettiamo che la prima cella abbia indice ind+1=48, che è ciò che succede nella figura mostrata sopra.

5.6-Gestione Memoria COP

Rimane ora un ultimo passaggio da fare prima che la CPU sia in grado di eseguire la nuova istruzione.

Come accennato prima, la scelta delle microistruzioni da eseguire dipende dal COP delle istruzioni, seguendo il sistema mostrato in *figura 192*.

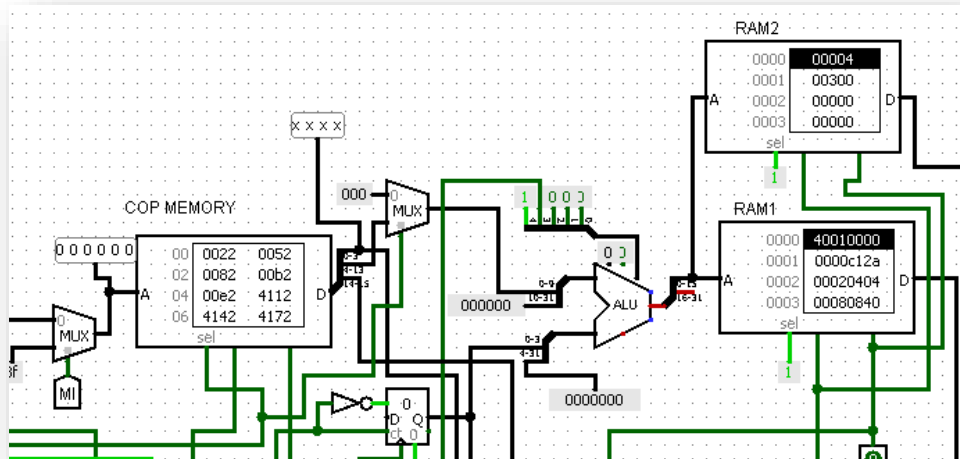


Figura 192: Schema circuitale della COP Memory in Logisim

Il codice COP prelevato dall'IR diventa l'INDIRIZZO della cop memory, che in uscita avrà i segnali di controllo necessari all'esecuzione delle microistruzioni.

Le celle della COP memory contengono 16 bit e sono organizzate nel seguente modo:

- 15:14 tipo di istruzione (R=00, I=01, J=11)
- 13:2 indirizzo nella ROM della PRIMA MICROISTRUZIONE dell'istruzione
- 1:0 tempo di wait per concludere la fase di execute

Di seguito viene mostrato un esempio di come costruire la cella della COP MEMORY corrispondente all'istruzione BEQZ .

1) TIPO DI ISTRUZIONE

La BEQZ è di tipo I, quindi 15:14 saranno pari a [01]

2) PRIMO INDIRIZZO

Come visto in precedenza il primo indirizzo nella RAM dell'istruzione BEQZ è 2fh, quindi i bit 13:3 prenderanno il valore [00 0010 1111].

3) TEMPO DI WAIT

Ovvero il NUMERO DI TEMPI DI CLOCK necessari per uscire dalla fase di EXECUTE dopo aver eseguito la prima istruzione.

Se ad esempio la fase EXECUTE di BEQZ impiega 2 tempi di clock per essere eseguita, i bit 3:0 prenderanno il valore $(\text{Clock_per_EXE} - 1) = 2 - 1 = 1$, quindi in binario [0001].

Ricomponendo i campi otteniamo una cella dal valore: 0100 0010 1111 0001, che in esadecimale corrisponde a 42f1h.

All'uscita dalla COP RAM i campi della cella vengono quindi separati: il primo indirizzo entra direttamente come primo addendo della ALU, il clock comincia a contare fino al raggiungimento del TEMPO DI WAIT, al termine del quale manda un segnale di ready all'UDC, mentre il tipo di istruzione viene usato per interpretare correttamente il valore dei registri sorgente e destinazione.

Ora si ha quindi il valore corretto della cella della COP MEMORY, questo valore va inserito all'INDIRIZZO DEL COP corrispondente: nel nostro caso viene scelto come COP della BEQZ il valore 16=10000, mentre per la BNEQZ 17=10001.

Una volta caricato, si avrà quindi la situazione:

$\text{Mcop}[\text{COP_BEQZ}] = \text{control_sgn} \rightarrow \text{M}[10000] = 0100\ 0000\ 1011\ 1101$

Completato anche questo passaggio, la CPU è ora abilitata anche all'esecuzione di istruzioni di tipo BEQZ e BNEQZ.

5.7-Aggiungere l'istruzione al compiler Assembly

A questo punto si è in grado di caricare il codice macchina della nuova istruzione nel DLX e osservare il comportamento del sistema.

Tradurre codice ASSEMBLY in CODICE MACCHINA a mano, come per il caso dell'RTL, è un processo tedioso che può portare ad avere diversi errori software.

Anche per questo passo viene quindi usato il supporto del compiler ASSEMBLY descritto nel CAPITOLO 5.

Essendo la BEQZ una istruzione di tipo I, possiamo tranquillamente trattarla come tutte le istruzioni simili, sarà l'RTL a gestire il branching.

5.8-Realizzazione del programma BLINK

In questo paragrafo sarà mostrato come scrivere un programma ASSEMBLY che faccia uso della nuova istruzione.

Il processo è semplice: basta aprire il file DLX.txt nella cartella compilerASS_BIN e scrivere tra le graffe della funzione "main" le istruzioni desiderate.

Si vuole ad esempio far eseguire al calcolatore questa funzione, “astratta” con un codice simile a C:

```
int R1=0; //DICHIARA VARIABILE
while (true){ //CICLO WHILE INFINITO
    if (R1==1){ //SE R1 è diverso da 0, allora spegnere i led e mettere R1 a 0
        led off
        R1=0;
    }else{ //Altrimenti accendere Led e mettere R1 a 1
        Led on
        R1=1;
    }
}
```

Questa funzione viene spesso chiamata “blink”, in pratica accende e spegne periodicamente un led in maniera alternata, facendolo quindi “lampeggiare” a una frequenza determinata.

Il codice Assembly impiegato per realizzare tale funzione viene mostrato in *figura 193*:

```
11 //DIRETTIVE EQU//
12 readf
13 ledctrl equ 8000h
14 ledbuf equ 8001h
15 LEDON equ 3h
16 MAINON equ 0h
17 MAINOFF equ 0h
18 }
19
20
21 $main{
22
23 beqz r1, LEDON //INDIRIZZO 0 IN MEMORIA, OVVERO MAIN//
24 addi r1 r0 0h
25 sw ledbuf(r0) r1 //Spengo i led//
26 jr r0 //Torna a MAIN//
27 addi r1 r0 1h //INDIRIZZO 4 IN MEMORIA, OVVERO LEDON//
28 sw ledbuf(r0) r1 //Accendo i led//
29 jr r0
30
31 }
```

Figura 193: Codice Assembly impiegato per realizzare il programma Blink

Nel codice C la prima istruzione è una inizializzazione di variabile, in Assembly questo non è necessario, poiché per questo esempio vengono usati direttamente i registri del Register File come contenitori di variabili. In questo particolare codice i confronti verranno effettuati sul registro R1.

Di seguito c'è l'istruzione While (true), ovvero un ciclo infinito. Nel caso Assembly, basta pensare che a ogni nuova istruzione il valore di PC incrementa automaticamente grazie alla fase di Decode dell'istruzione. Senza alcuna istruzione di salto, le istruzioni di un codice Assembly verranno quindi eseguite dalla prima all'ultima partendo da PC=0 fino ad arrivare al valore massimo di PC. Supponendo però di introdurre una istruzione di JUMP in un punto qualsiasi del codice e di provocare quindi un salto, se questo salto porta ad un indirizzo precedente rispetto a quello della JUMP, allora si creerà un Loop infinito dove la macchina continua a eseguire istruzioni, tornare indietro e eseguire le stesse istruzioni, per poi tornare nuovamente indietro e così via. Nel programma blink viene quindi inserita una istruzione jump register alla fine per saltare all'indirizzo 0, ovvero l'inizio del MAIN. La mappa concettuale di questo salto viene mostrata in *figura 194*:

```

11 //DIRETTIVE EQU//
12 read{
13 ledctrl equ 8000h
14 ledbuf equ 8001h
15 LEDON equ 3h
16 MAINON equ 0h
17 MAINOFF equ 0h
18 }
19
20
21 $main{
22
23 beqz r1, LEDON //INDIRIZZO 0 IN MEMORIA, OVVERO MAIN//
24 addi r1 r0 0h
25 sw ledbuf(r0) r1 //Spengo i led//
26 jr r0 //Torna a MAIN//
27 addi r1 r0 1h //INDIRIZZO 4 IN MEMORIA, OVVERO LEDON//
28 sw ledbuf(r0) r1 //Accendo i led//
29 jr r0
30
31 }

```




Figura 194: Salto in PC dato dall'istruzione BEQZ

All'interno del ciclo while va effettuato un confronto: se il registro R1 è diverso da 0 allora i led sono accesi e vanno spenti, mentre se R1 è pari a 0 i led vanno accesi. È qui che entra in gioco la "BEQZ". La beqz consente infatti di eseguire un'operazione molto importante detta di "controllo del flusso", permette quindi di scegliere se eseguire o meno dei pezzi di codice, in questo caso accendere o spegnere un led. Come mostrato in figura, al momento dell'esecuzione della BEQZ se r1 è pari a zero viene effettuato un salto in LEDON, che tramite la direttiva equ scritta prima del MAIN viene sostituito con 3h.

Viene quindi eseguita l'istruzione di addi per caricare 1 su R1, e con la store successiva tale valore viene caricato nell'area di memoria dove sono mappati i led, quindi all'indirizzo ledbuf pari a 8001h. L'unione di queste due operazioni permette quindi di Accendere i led. Il flusso del codice in questo caso viene mostrato in *figura 195*.

```
11 //DIRETTIVE EQU//
12 read{
13 ledctrl equ 8000h
14 ledbuf equ 8001h
15 LEDON equ 3h
16 MAINON equ 0h
17 MAINOFF equ 0h
18 }
19
20
21 $main{
22
23 beqz r1, LEDON //INDIRIZZO 0 IN MEMORIA, OVVERO MAIN//
24 addi r1 r0 0h
25 sw ledbuf(r0) r1 //Spendo i led//
26 jr r0 //Torna a MAIN//
27 addi r1 r0 1h //INDIRIZZO 4 IN MEMORIA, OVVERO LEDON//
28 sw ledbuf(r0) r1 //Accendo i led//
29 jr r0
30
31 }
```

Figura 195: Accensione del LED e salto all'istruzione zero tramite jump

Dopo essere saltati nuovamente all'indirizzo 0, lo stato dei registri è diverso rispetto a prima. La BEQZ trovando che R1 è ora diverso da zero non effettuerà nessun salto, lasciando che il programma scorra normalmente. Verranno eseguite la add e la store a indirizzo 1 e 2, con la differenza che verrà caricato uno zero nell'area di memoria mappata dei led. Questo corrisponde a spegnere i led. Va notato che il programma in questo caso continuerebbe ad eseguire non solo le due istruzioni desiderate, ma anche le due istruzioni del caso precedente, poiché sono scritte di seguito. La soluzione è aggiungere una nuova JUMP dopo le due istruzioni per saltare all'indirizzo MAIN evitando di eseguire il pezzo di codice indesiderato. Il flusso del codice in questo caso viene mostrato in *figura 196*.

```

11 //DIRETTIVE EQU//
12 read{
13 ledctrl equ 8000h
14 ledbuf equ 8001h
15 LEDON equ 3h
16 MAINON equ 0h
17 MAINOFF equ 0h
18 }
19
20
21 $main{
22
23 beqz r1, LEDON //INDIRIZZO 0 IN MEMORIA, OVVERO MAIN//
24 addi r1 r0 0h
25 sw ledbuf(r0) r1 //Spengo i led//
26 jr r0 //Torna a MAIN//
27 addi r1 r0 1h //INDIRIZZO 4 IN MEMORIA, OVVERO LEDON//
28 sw ledbuf(r0) r1 //Accendo i led//
29 jr r0
30
31 }

```

Figura 196: Spegnimento dei LED e salto all'istruzione zero

Il programma Blink alterna quindi tra questi due comportamenti all'infinito, consentendo di far lampeggiare una periferica Led.

5.9-Visualizzazione dell'esecuzione del programma nella CPU

Il codice Assembly deve ora essere caricato nella memoria della CPU. Come spiegato nel Capitolo 5, dopo l'esecuzione dello script COMPILER_ASS_BIN.m l'output DLX_MC.txt è pronto per essere caricato nella memoria ROM dedicata ai programmi. Una volta aperto il circuito della CPU dal programma Logisim, scegliendo la voce "Load Circuit" e scegliendo il file /VirtualShock_src/1_Nanosphere22/1_NANOSPHERE.circ, dopo aver caricato il codice la memoria principale appare come in figura 197.

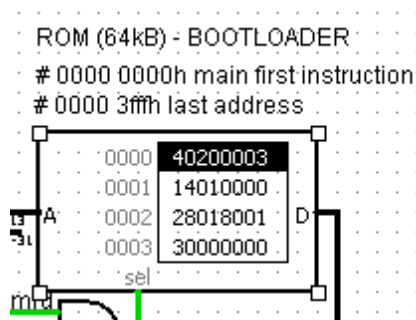


Figura 197: Zoom sulla memoria ROM principale

È possibile notare dalla figura come la cella correntemente selezionata in una memoria si illumini con una banda nera. L'istruzione all'indirizzo 0 correntemente illuminata è la BEQZ, che sarà quindi la prima ad essere eseguita: mediante essa

viene controllato se il registro r1 è pari a 0, per poi incrementare il PC di 3 se la condizione è verificata. In fondo alla cella della prima istruzione è possibile vedere il valore 3, che sarà l'entità del salto (+1 dato dall'incremento di PC). Essendo il registro r1 all'inizio dell'esecuzione pari a 0, ci si aspetta un salto nel valore di PC e quindi negli indirizzi delle celle della ROM.

Si comincia facendo partire il clock. Con ctrl+r si resetta la simulazione, mentre con ctrl+t si avanza di un colpo di clock. Ora che si è entrati nella fase di IF dell'istruzione si procede ad "aprire" l'UDC per vedere cosa sta succedendo al suo interno, come mostrato in *figura 198 e 199*.

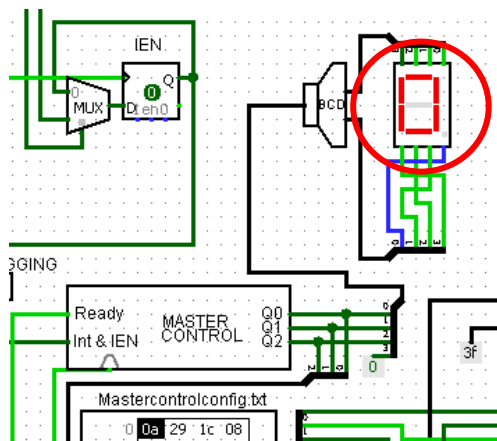


Figura 198: All'inizio del programma l'UDC è nella fase IF

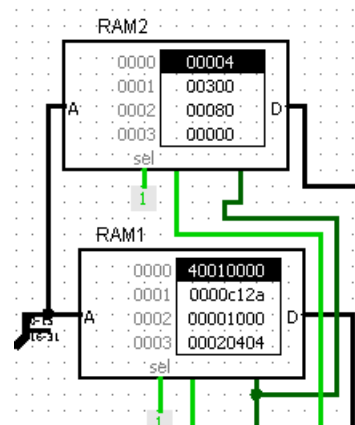


Figura 199: Zoom sulle ROM dell'UDC

La fase dell'istruzione corrente viene identificata dal numero sul display BCD: si parte da 0, ovvero la fase IF. Si può notare guardando la RAM1 e 2 in *figura 200* come la fase di IF corrisponda alla prima cella, mentre la fase DECODE eseguita al clock successivo alla seconda cella.

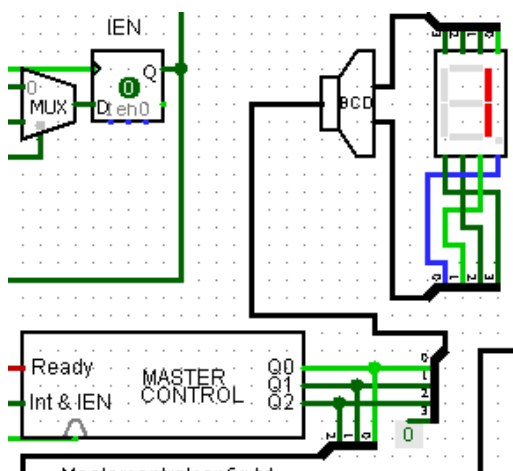


Figura 199: Il numero 1 del pannello BCD indica l'ingresso nella fase di Decode

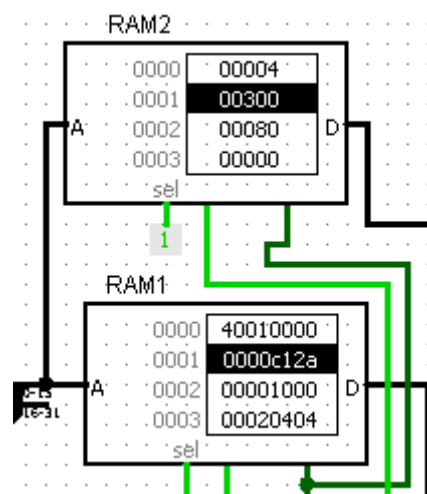


Figura 200: Zoom sulle ROM dell'UDC

Infine si entra nella fase EXECUTE. Si può vedere come la COP memory si attiva e seleziona la cella 2f della RAM, ovvero la PRIMA MICROISTRUZIONE della BEQZ, come mostrato in *figura 201 e 202*.

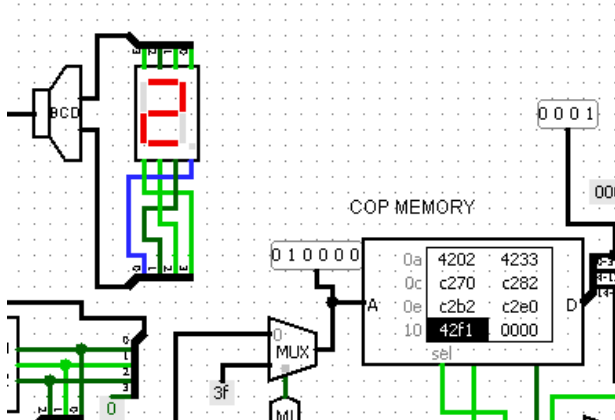


Figura 201: Attivazione della COP Memory nella fase di Execute

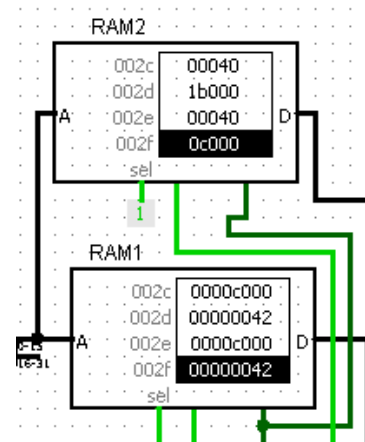


Figura 202: Zoom sulle ROM dell'UDC

In questa fase la alu sta già effettuando il confronto $A=0$, il cui risultato viene messo nello zero flag, come mostrato in *figura 203*.

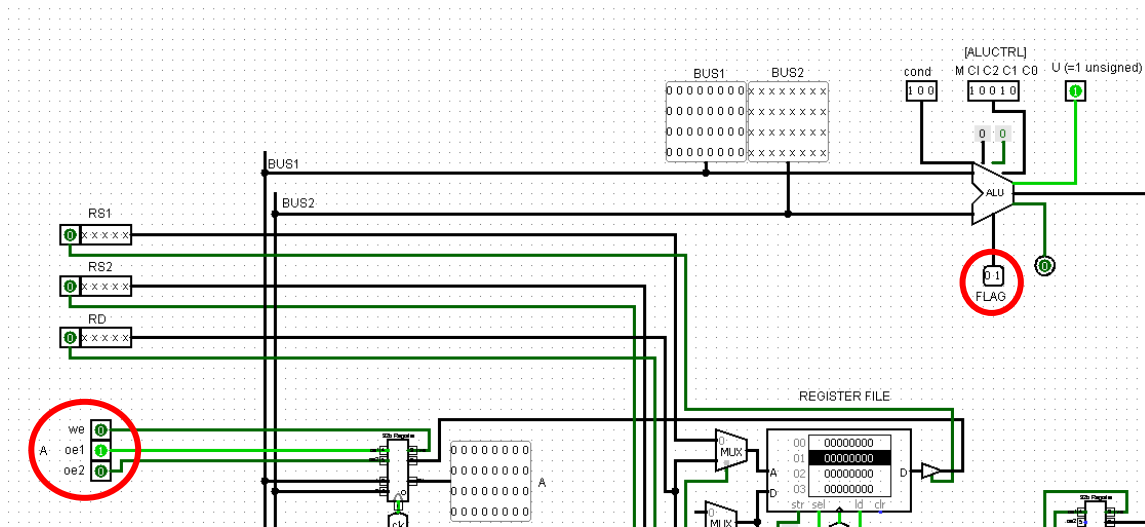


Figura 203: Zoom della ALU configurata in modalità confronto

A questo punto entra in gioco il circuito del paragrafo 6.4, costituito dai mux che consentono la scelta della modalità BRANCH, come mostrato in *figura 204*.

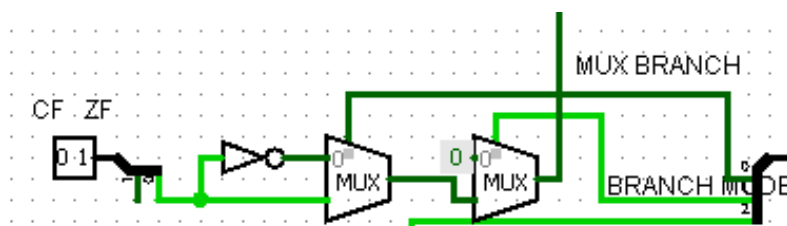


Figura 204: Zoom sulla circuiteria dell'UDC impiegata per realizzare il branching

Viene scelto il branch dello ZF nottato, che in questo momento è a 0, essendo ZF a 1. L'uscita di questo circuito è il ready del Master Control, che quindi è pari a zero. Ci si aspetta quindi un ultimo periodo di clock, come mostrato in *figura 205*, dove la BEQZ effettuerà l'operazione di incremento di PC.

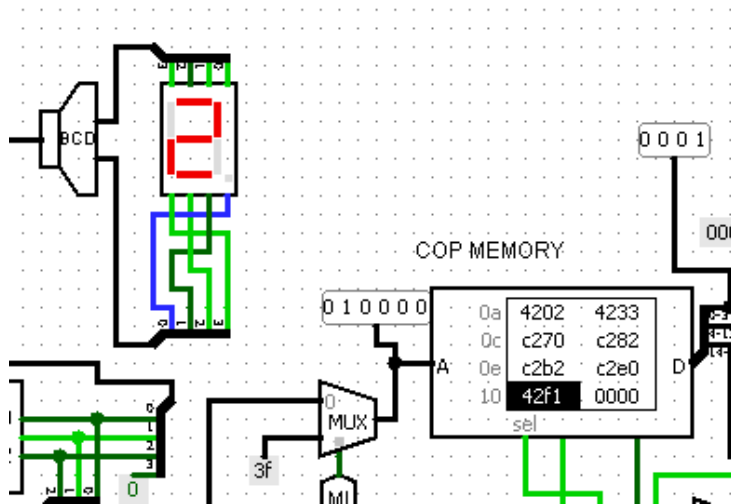


Figura 205: Il pannello BCD indica la cifra 2, si sta ancora eseguendo la fase di Execute

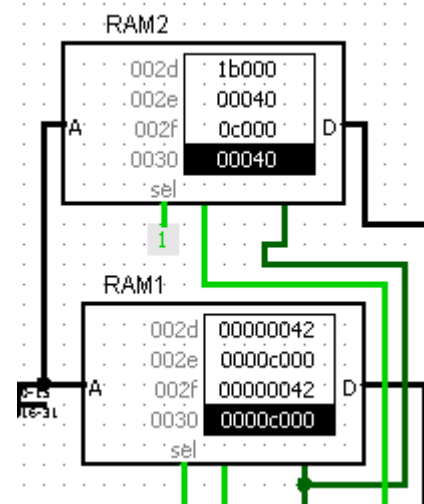


Figura 206: Zoom sulle ROM dell'UDC

Al successivo colpo di clock si può notare dalla figura come il pannello BCD indichi ancora 2, e le RAM dell'UDC sono passate alla microistruzione successiva, che è quella di incremento del PC.

L'istruzione sembra funzionare, ma per sicurezza si può andare a osservare cosa sta succedendo nel Data Path, mostrato in *figura 207*.

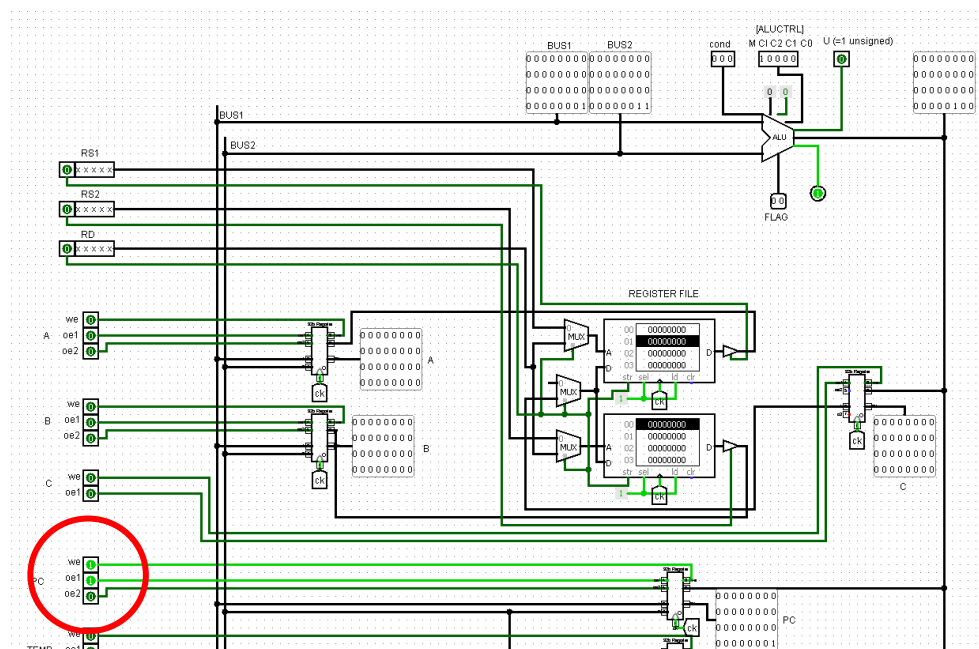


Figura 207: Zoom sul Data Path

La ALU sta incrementando di 3 (valore sul bus 2) il valore corrente di PC. Ci si aspetta quindi un salto all'istruzione di indirizzo 4.

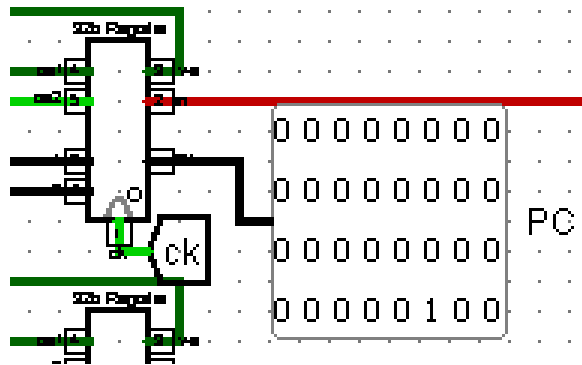


Figura 208: Zoom sul registro PC del Data Path

Si può verificare che il PC è effettivamente il registro incrementato osservandone il contenuto al successivo colpo di clock, come mostrato in figura 208. Si osservi ora l'istruzione selezionata nella ROM principale dei programmi, mostrata in figura 209:

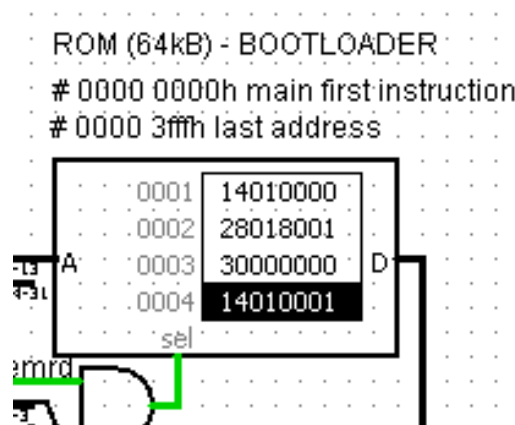


Figura 209: Stato della Memoria DLX a seguito dell'incremento di PC

Come atteso, è avvenuto il salto all'istruzione numero quattro. La situazione nel flusso del codice Assembly viene mostrata in figura 210.

```

22
23 beqz r1, LEDON //INDIRIZZO 0 IN MEMORIA, OVVERO MAIN//
24 addi r1 r0 0h
25 sw ledbuf(r0) r1 //Spengo i led//
26 jr r0 //Torna a MAIN//
27 addi r1 r0 1h //INDIRIZZO 4 IN MEMORIA, OVVERO LEDON//
28 sw ledbuf(r0) r1 //Accendo i led//
29 jr r0
30

```

Figura 210: Visualizzazione nel codice del salto causato da BEQZ

La prossima istruzione da eseguire quindi è la ADDI. Al termine della sua esecuzione ci si aspetta di trovare il valore 1 nel registro R1 del Register File, mostrato in *figura 211*.

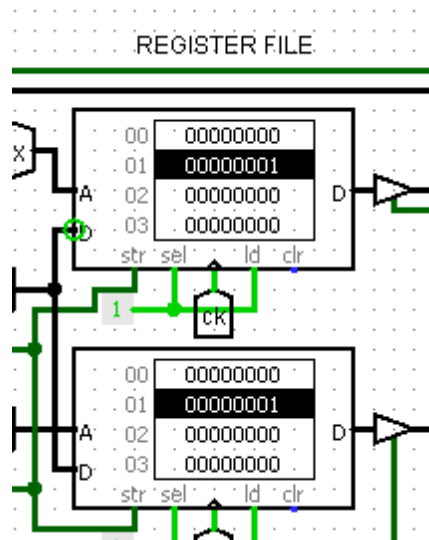


Figura 211: Il registro R1 del Register File contiene il valore 1

Come si può constatare dalla figura, R1 è stato correttamente portato a 1. Ora questo valore verrà caricato nell'area di memoria dei LED tramite la Store Word. Al termine dell'esecuzione della STORE, nel circuito MAIN si potrà vedere la periferica LED accendere il primo led sulla destra, come mostrato in *figura 213* insieme al flusso del codice mostrato in *figura 212*.

```

22
23 beqz r1, LEDON //INDIRIZZO 0 IN MEMORIA, OVVERO MAIN//
24 addi r1 r0 0h
25 sw ledbuf(r0) r1 //Spengo i led//
26 jr r0 //Torna a MAIN//
27 addi r1 r0 1h //INDIRIZZO 4 IN MEMORIA, OVVERO LEDON//
28 sw ledbuf(r0) r1 //Accendo i led//
29 jr r0
30

```

Figura 212: Visualizzazione del codice impiegato per l'accensione dei LED

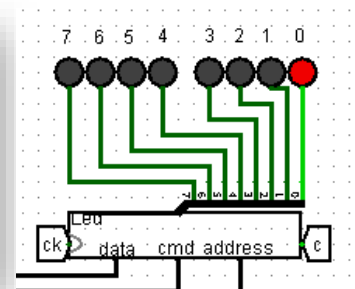


Figura 213: Il primo LED viene acceso

A questo punto entra in gioco l'ultima istruzione: la jump register. Dalla durata di un clock, questa semplice istruzione deve caricare il valore zero in PC per effettuare un salto di indirizzi all'indietro.

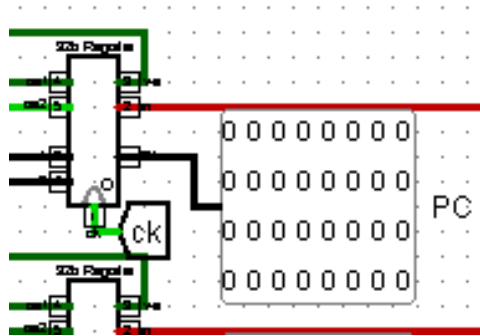


Figura 214: Zoom sul Registro PC del Data Path, resettato a zero correttamente

Come mostrato in *figura 214* valore di PC è stato modificato correttamente. Guardando la ROM principale, è possibile vedere come l'istruzione selezionata sia di nuovo quella ad indirizzo zero, come mostrato in *figura 215*.

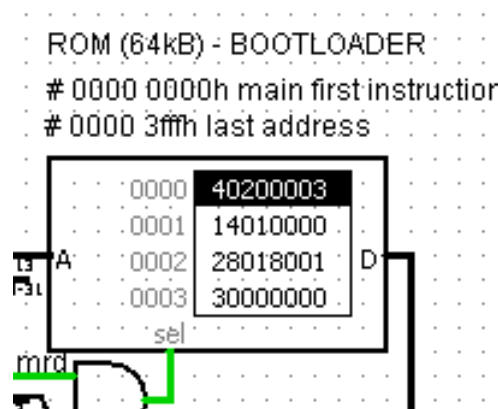


Figura 215: Zoom sulla memoria principale, viene selezionata la prima cella

E' stata quindi eseguita correttamente l'istruzione JUMP, la raffigurazione del salto nel flusso del programma Assembly viene mostrata in *figura 216*.

```

22
23 beqz r1, LEDON //INDIRIZZO 0 IN MEMORIA, OVVERO MAIN//
24 addi r1 r0 0h
25 sw ledbuf(r0) r1 //Spengo i led//
26 jr r0 //Torna a MAIN//
27 addi r1 r0 1h //INDIRIZZO 4 IN MEMORIA, OVVERO LEDON//
28 sw ledbuf(r0) r1 //Accendo i led//
29 jr r0
30

```

Figura 216: Salto all'indirizzo zero visualizzato nel flusso del codice

Ora però la situazione è diversa da quella iniziale: il registro R1 non è più vuoto, bensì contiene un uno, che è diverso da zero. Ci si aspetta quindi che la BEQZ ora

non faccia alcun salto e che la macchina passi all'istruzione successiva, come mostrato in *figura 217 e 218*.

```

22
23 beqz r1, LEDON //INDIRIZZO 0 IN MEMORIA, OVVERO MAIN//
24 addi r1 r0 0h
25 sw ledbuf(r0) r1 //Spengo i led//
26 jr r0 //Torna a MAIN//
27 addi r1 r0 1h //INDIRIZZO 4 IN MEMORIA, OVVERO LEDON//
28 sw ledbuf(r0) r1 //Accendo i led//
29 jr r0
30

```

Figura 217: Flusso di esecuzione atteso del codice, senza alcun salto

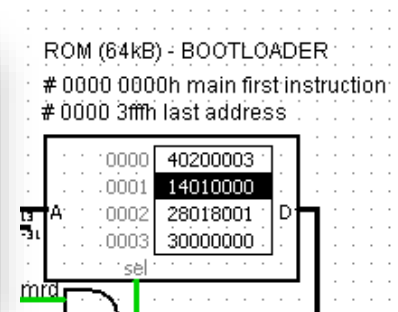


Figura 218: Viene selezionata la seconda cella nella Memoria ROM principale

La ADDI eseguita adesso è diversa da quella del caso precedente: viene infatti resettato a zero il valore di R1, come mostrato in *figura 219* nel Register File.

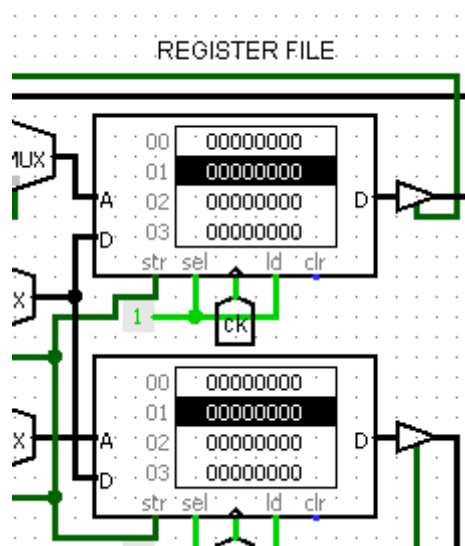


Figura 219: Il registro R1 del Register File contiene il valore zero

Il valore che la STORE andrà a caricare nei led quindi sarà zero, che equivale a spegnere tutti i led accesi, come mostrato in *figura 220 e 221*.

```

22
23 beqz r1, LEDON //INDIRIZZO 0 IN MEMORIA, OVVERO MAIN//
24 addi r1 r0 0h
25 sw ledbuf(r0) r1 //Spengo i led//
26 jr r0 //Torna a MAIN//
27 addi r1 r0 1h //INDIRIZZO 4 IN MEMORIA, OVVERO LEDON//
28 sw ledbuf(r0) r1 //Accendo i led//
29 jr r0
30

```

Figura 220: Esecuzione delle istruzioni ADDI e STORE

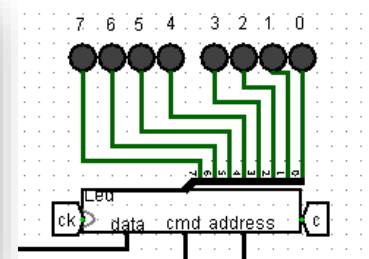


Figura 221: Il led viene spento correttamente

L'istruzione successiva è la JUMP, che in maniera identica al caso precedente riporta il PC a ZERO, come mostrato nelle *figure 220,221 e 222*.

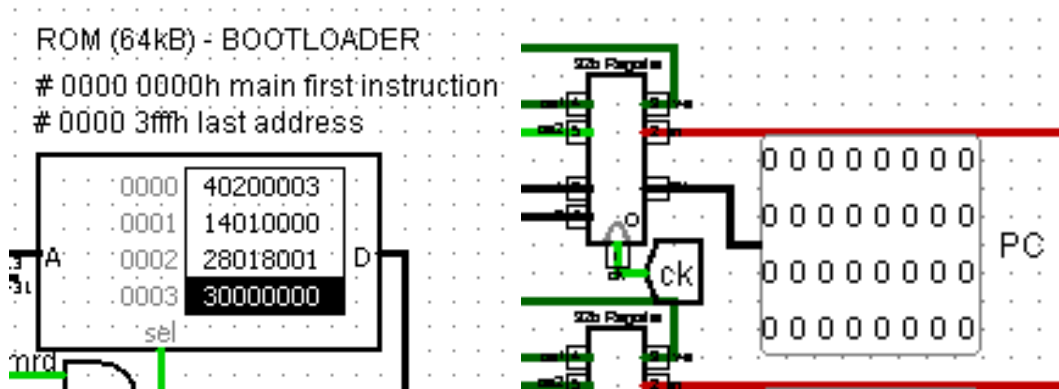


Figura 220: Cella contenente la jump in memoria principale

Figura 221: Zoom sul registro PC nel Data Path

```

23 beqz r1, LEDON //INDIRIZZO 0 IN MEMORIA, OVVERO MAIN//
24 addi r1 r0 0h
25 sw ledbuf(r0) r1 //Speno i led//
26 jr r0 //Torna a MAIN//
27 addi r1 r0 1h //INDIRIZZO 4 IN MEMORIA, OVVERO LEDON//
28 sw ledbuf(r0) r1 //Accendo i led//
29 jr r0
30

```

Figura 222: Visualizzazione nel codice del salto di indirizzi di PC

Adesso il caso è identico a quello analizzato per primo: R1 vale ZERO e la BEQZ salterà, e così via all'infinito.

Il programma Blink è il corrispettivo di Hello World per l'informatica, e permette di avere una prima esperienza con le due proprietà della Macchina di Von Neumann: l'Automatizzabilità e la Programmabilità. Non è più necessario infatti inserire manualmente gli indirizzi delle RAM dell'UDC o pilotare gli enable del Data Path: dopo aver premuto ctrl+k e aver fatto partire il clock automatico, è possibile osservare la Macchina andare avanti da sola. L'obiettivo principale del progetto Virtual Shock è finalmente stato raggiunto.

Conclusioni

Il Tool grafico Logisim permette di dare vita alle slides del corso di Calcolatori Elettronici: la possibilità di copiare lo schema logico di una rete direttamente nel Canvas di Logisim per poi simularla pilotandone input e clock permette di risolvere i dubbi incontrati durante lo studio teorico e di consolidare i concetti già appresi, fornendo uno strumento potente di supporto alla didattica. I sottocircuiti Logisim permettono poi di alzare gradualmente il livello di astrazione, fornendo una rappresentazione qualitativa del funzionamento di una rete complessa mantenendo però un solido livello tecnico di base. Non bisogna infatti pensare a una CPU come un semplice insieme di transistor, bensì come una rete di Elaborazione pilotata da un Controllore che automatizza tutte le operazioni in un ordine specificato dal programma.

Dagli spostamenti tra registri all'esecuzione sequenziale di microistruzioni alla programmazione in Assembly, il progetto Virtual Shock spazia tutti i 3 moduli di calcolatori, fornendo allo studente un sistema già funzionante in ogni suo aspetto, affrontabile e analizzabile step-by-step.

Le possibilità di miglioramento e aggiornamento di Virtual Shock inoltre sono molto vaste: la sua struttura modulare consente di implementare nuovi sistemi facilmente connettabili ai BUS del circuito principale, permettendo di aggiungere una moltitudine di nuove funzionalità.

Future nuove implementazioni prevedono:

- Aggiunta di nuove istruzioni Assembly mediante aggiornamento del Microcodice dell'UDC
- Sviluppo di Interrupt Vettorizzato
- Sviluppo di circuiteria aggiuntiva per la gestione di componenti JAVA in Logisim
- Porting dell'intero circuito della CPU Virtual Shock su una versione migliorata di Logisim sviluppata di recente, denominata Logisim Evolution, che oltre a fornire componenti aggiuntivi e una ottimizzazione migliore consente di definire nuovi componenti e sottocircuiti in linguaggio VHDL;
- Realizzazione di un nuovo compiler con interfaccia migliorata mediante APP open source realizzata in HTML, CSS e JS;
- Realizzazione di nuovi componenti Logisim in linguaggio JAVA.

Una volta completata la comprensione del progetto Virtual Shock, questa tesi può quindi essere considerata come un manuale di istruzioni per future implementazioni.

SITOGRAFIA

- Nella sezione di *Calcolatori Elettronici* del sito <https://virtuale.unibo.it/> sono reperibili le slides del corso
- Il manuale completo di Logisim e i files sorgente possono essere reperiti interamente e gratuitamente dal sito <http://www.cburch.com/logisim/> nella sezione *Documentation e Download*.
- La documentazione per il software Matlab impiegato per la realizzazione dei compiler è reperibile all'indirizzo <https://it.mathworks.com/help/matlab/>
- Le mappe concettuali impiegate nel corso dei paragrafi sono state realizzate mediante il software *Draw IO*, scaricabile gratuitamente dal sito <https://app.diagrams.net/>
- www.extremetech.com
- www.student-circuit.com
- www.amazon.com

RINGRAZIAMENTI

Si ringraziano i professori dell'Università Alma Mater Studiorum di Bologna UNIBO Roffia Luca per gli spunti dati durante l'insegnamento di Reti Logiche che hanno innescato l'interesse nel progetto Virtual Shock e Testoni Nicola per il suggerimento e spiegazione della programmazione mediante microcodice che ha consentito la realizzazione del progetto finale.