UNIVERSITÀ DEGLI STUDI DI MILANO

Relazione di : Andrea Zubenko (908857)
Laboratorio di architettura degli elaboratori I (turno B)
Anno Accademico 2017/2018

Specifiche del progetto **2**
Input 2
Output 2
Stato iniziale 2
Stato finale 2
Ciclo di utilizzo 2
Sottocircuiti implementati **4**
CheckNove 4
Super Comparator 6
Input filter 7
Input filter 4 7
Equals 0 8
SommaNove 9
Circuito principale **10**
Griglia del sudoku 10
Controllo degli errori 12
Controllo di fine partita 13
Considerazioni finali **14**
Possibili migliorie future 14

Specifiche del progetto

Il progetto consiste in un circuito che permette all'utente di giocare al classico gioco del sudoku.

Input

- 81 contatori "nascosti" all'interno dei display, utilizzati per inserire un numero all'interno di una cella

Output

- 81 display che rappresentano la griglia del sudoku
- Un led per indicare la presenza di errori nella griglia (ad esempio due numeri uguali nella stessa riga)
- Un led per indicare il termine della partita (vinta)

Stato iniziale

Tutti i display risultano vuoti (viene visualizzato "-").
Entrambi i led sono spenti.

Stato finale

Al termine del gioco, nessun display risulterà vuoto, il led di fine partita sarà acceso mentre quello di errore spento.

Ciclo di utilizzo

L'utente seleziona un display (cella della griglia) con il mouse, usando il "poke tool" di logisim.
Dopodichè scrive una cifra da tastiera, che verrà impostata sul display (0 per resettarlo).
Lo scopo del gioco è riempire tutte e 81 le celle con le cifre tra 1 e 9, facendo in modo che nessuna riga, colonna o sotto-griglia 3x3 contenga cifre ripetute.

Sottocircuiti implementati

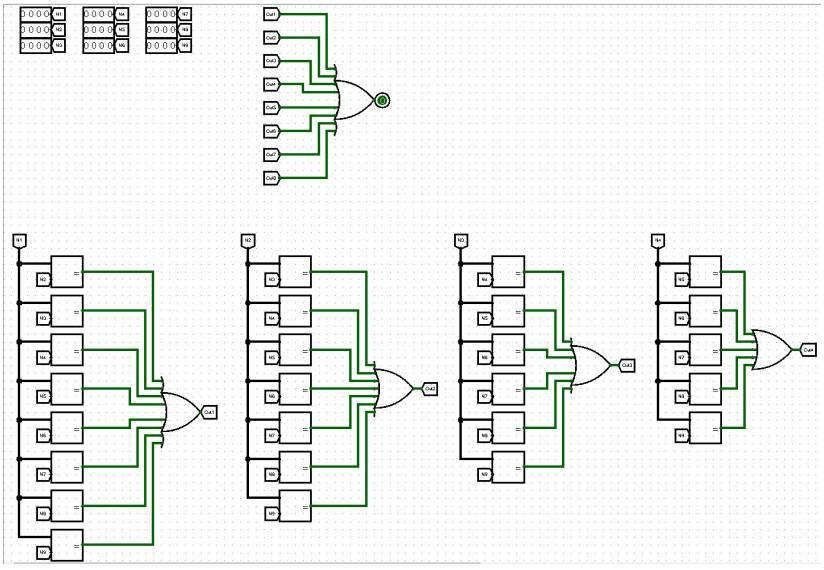
CheckNove

CheckNove è un circuito che prende in input nove numeri interi e controlla se almeno due di questi sono uguali tra loro, nel quale caso restituendo in output 1 (e 0 altrimenti).
Verrà utilizzato nel circuito principale per rilevare errori in righe, colonne o sotto-griglie, che sono formate sempre da 9 celle.
Proprio per questo, due “0” non sono considerati uguali, dal momento che non si vuole segnalare errore in presenza di due o più celle vuote.

La prima cifra viene confrontata con le 8 rimanenti per rilevare eventuali doppioni.
La seconda con le rimanenti 7 (risulta inutile confrontare con la prima, dato che la relazione di uguaglianza è simmetrica e il confronto num2==num1 risulterebbe uguale a quello num1==num2, già effettuato), la terza con le altre 6 e così via, diminuendo quindi man mano il numero di controlli.

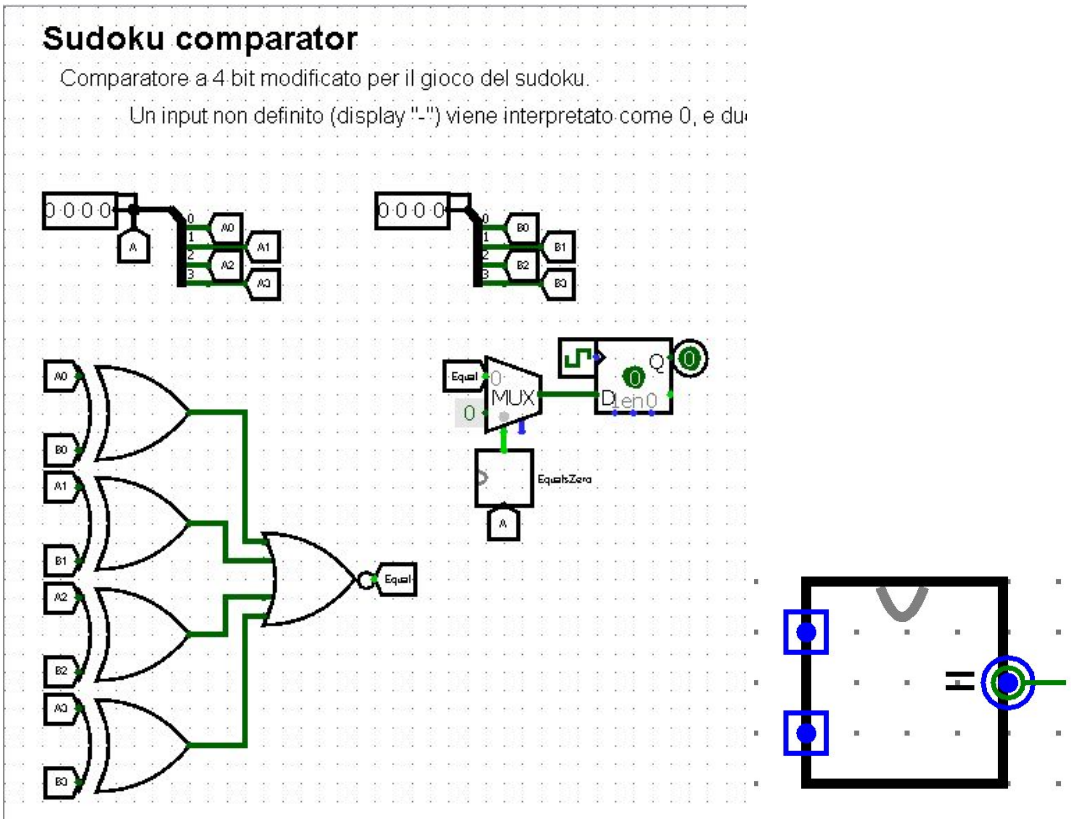
Per eseguire il confronto tra due cifre, ho scelto di usare un componente personalizzato (comparatore) in quanto

- Il comparatore standard di logisim, oltre a stabilire l’uguaglianza, permette anche di stabilire quale tra i due input è il maggiore. Quindi dovrebbe essere più lento di un comparatore più semplice.
- Per motivi che saranno chiari più avanti, avevo la necessità di vedere un input indefinito (UNKNOWN in logisim) come zero.
- Per come è stato progettato il circuito, lo zero rappresenta la cella vuota. Di conseguenza, il mio comparatore, usato come rivelatore di errori, non dovrebbe dare in output “1” (errore) nel caso vengano confrontati due zeri.



Mi sono tuttavia scontrato con un problema: a causa di limitazioni tecniche di logisim, il mio comparatore personalizzato, perfettamente funzionante e teoricamente senza problemi di prestazioni rispetto al comparatore standard con le stesse funzionalità, risultava completamente inutilizzabile. Il consumo di RAM di logisim raggiungeva il GB, il tempo di caricamento del circuito superava il minuto, e il circuito stesso, una volta usati sufficienti “checkNove”, semplicemente smetteva di funzionare per il troppo lag. Questo è dovuto al fatto che nel caso di un circuito personalizzato, logisim è costretto a simulare in tempo reale ogni singola porta, mentre i componenti standard sono semplici oggetti java con delay prefissati. Usando i miei componenti personalizzati, logisim si trovava costretto a simulare svariate migliaia di porte, con un ovvio crollo delle prestazioni.
Per ovviare al problema, ho in prima istanza ristrutturato il progetto riducendo all’osso il numero di componenti non originali, soluzione che ha di fatto risolto il problema (il circuito funzionava correttamente e velocemente), ma non mi ha lasciato soddisfatto.
Ho quindi deciso di creare una mia libreria java per logisim contenente i componenti a me necessari, arrivando di fatto ad avere componenti con le stesse identiche funzionalità di quelli che avevo creato, ma senza alcun problema di prestazioni.
Di seguito illustro il funzionamento del sottocircuito comparatore originale, anche se di fatto non è stato usato per i motivi spiegati sopra.

Super Comparator



Questo speciale comparatore verifica l'uguaglianza di due numeri da 4 bit in input, interpretando un input non definito ("-" sul display) come 0 e non considerando uguali due zeri.

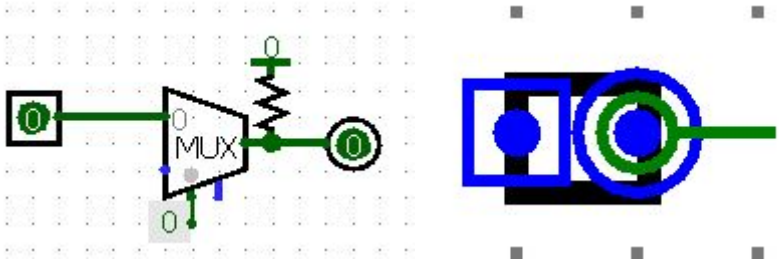
Il circuito lavora in tre fasi:

1. "Filtra" l'input, cioè trasforma un eventuale input non definito in zero, tramite altro componente personalizzato
2. Effettua la normale comparazione tramite quattro porte xor e una nor
3. Verifica se almeno uno dei due numeri è uno zero, nel caso inviando "0" in output a prescindere dal risultato della comparazione

Prima di inviare il risultato in output (1 o 0 a seconda che gli input siano uguali o meno), inoltre, questo viene salvato in un flip flop comandato dal clock, nel tentativo di risolvere eventuali problemi di data races.

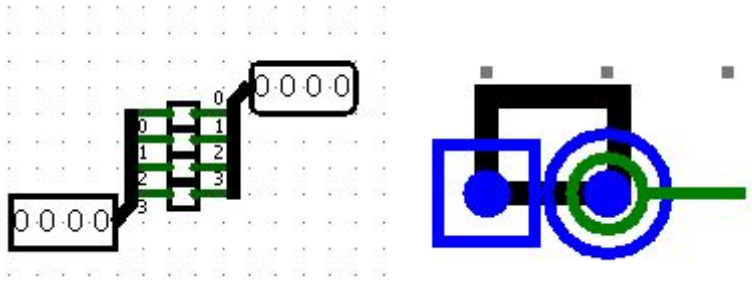
Input filter

Piccolo circuito che non fa altro che propagare l'input, avendo l'accortezza di modificare l'input indefinito in zero tramite un pull resistor.
Questo componente viene reso necessario dal fatto che applicare il pull resistor direttamente al cavo va a modificare il valore del cavo stesso, nel mio caso trasformando il display da "-" a "0".

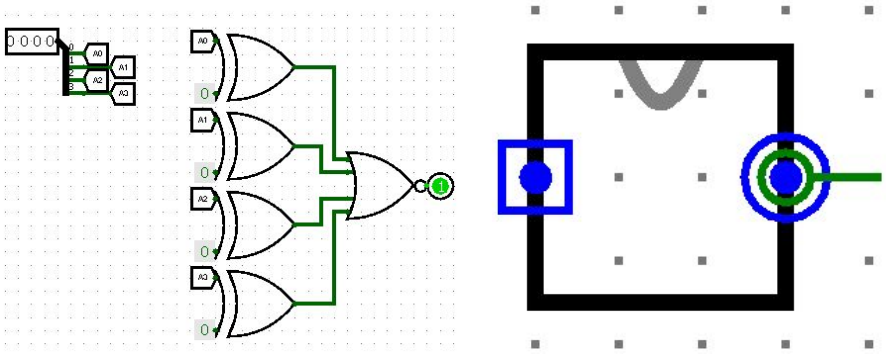


Input filter 4

Nient'altro che la versione a 4 bit di filtered input

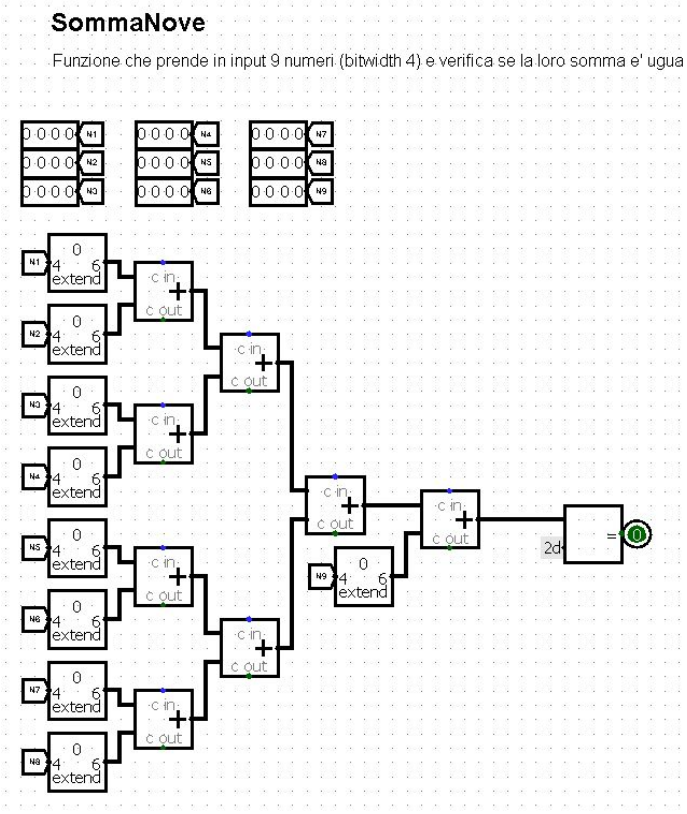


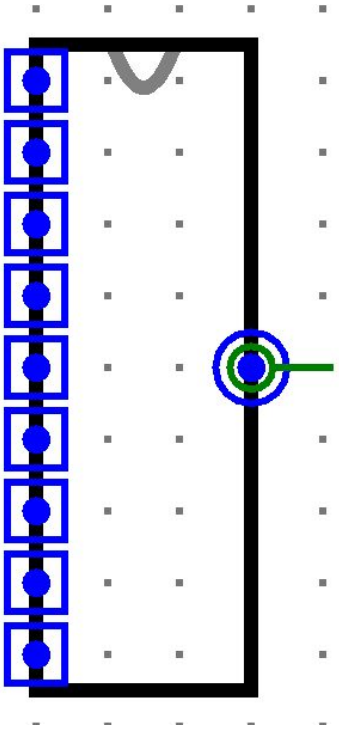
Equals 0



Comparatore “di comodo” che verifica l’uguaglianza a 0 di un input a 4 bit. L’idea originale era di sostituire le porte xor e not con altre più efficaci tramite costruzione di una tabella di verità, ma quest’idea non è stata sviluppata dato che il componente stesso non è più stato usato.

SommaNove





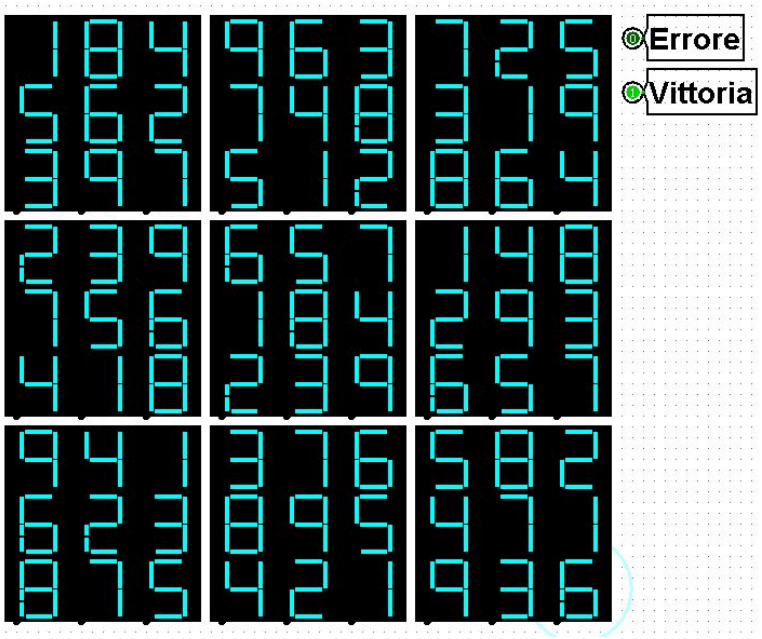
Circuito che somma 9 numeri interi in input, quindi restituisce in output 1 o 0 a seconda che la somma sia pari a 45 (2d) o meno.
Il numero 45 non è altro che la somma dei primi 9 naturali, utilizzato dato che la funzione di sommaNove (in accoppiata con un controllo di duplicati) è quella di verificare se all'interno di una riga, colonna o sottogriglia sono presenti tutte le cifre da 1 a 9.

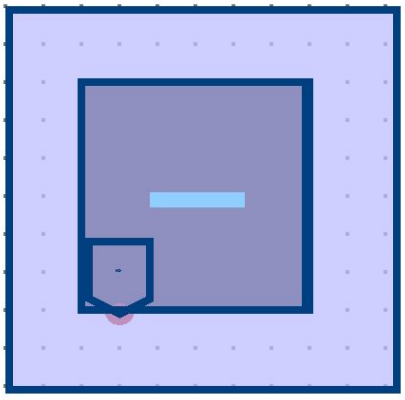
Dal momento che il minimo numero di bit necessari per rappresentare 45 è 6 (2^6 = 64), tutti gli input da 4 bit devono venire estesi a 6 bit tramite aggiunta di zeri.

Circuito principale

Il circuito principale si suddivide in tre sezioni: griglia del sudoku, controllo degli errori e controllo di fine partita

Griglia del sudoku





La griglia è composta da

- 81 display esadecimali modificati
- 81 contatori modificati
- 81 tunnel

I display sono una versione modificata del normale display esadecimale, con le seguenti modifiche:

- L'input zero viene visualizzato in automatico come se fosse input vuoto ("-")
- La dimensione è maggiorata rispetto al display classico, e inoltre la forma passa dall'essere rettangolare (alta) ad un vero e proprio quadrato, per questioni puramente estetiche
- Rimossa la porta clear, in quanto non usata

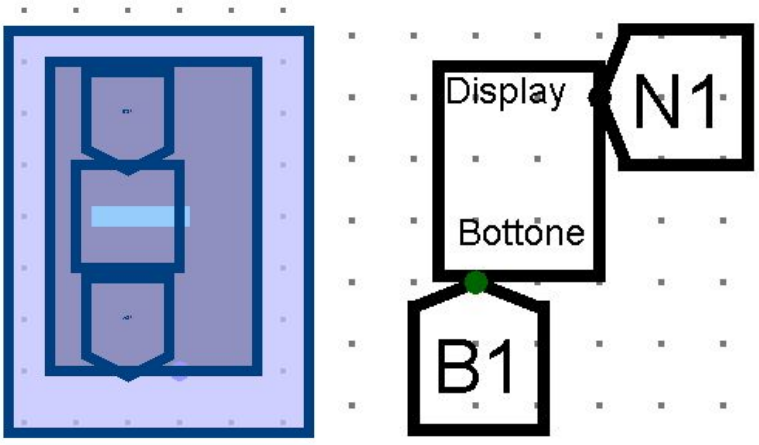
Il colore scuro dello sfondo è stato scelto appositamente per nascondere la “circuiteria” interna.

I contatori sono quasi del tutto identici al contatore standard, tranne che per

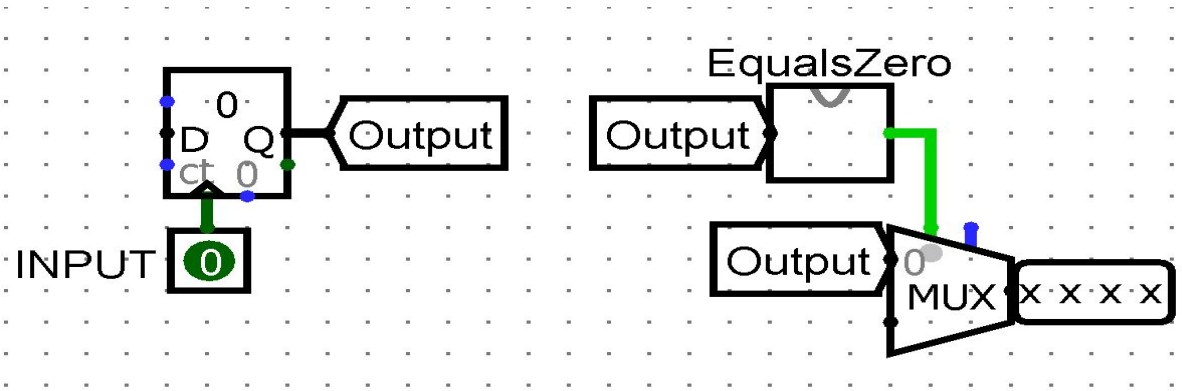
- la dimensione, resa molto simile (ma non identica, in quanto logisim non permette la completa sovrapposizione di componenti) a quella dei display in modo da renderne più semplice la selezione
- Il numero e posizionamento delle porte: la porta di output è stata posiziona in modo da sovrapporsi perfettamente a quella di input del display, mentre sono state nascoste quelle di “Data In” e clock in quanto non utili nel progetto
- La label contenente il numero attualmente memorizzato nel contatore, nascosta al fine di non sovrapporsi al display

Infine, i tunnel servono per trasferire informazione, cioè a far sapere al resto del circuito il valore contenuto in ogni cella. In questo caso, la dimensione del font delle label è stata ridotta al minimo, sempre per non sovrapporsi al display.

Prima di creare la libreria java, ho usato solo componenti standard per creare la griglia. La vecchia versione (ancora disponibile all’interno del progetto) prevedeva l’uso di un bottone, collegato a un contatore collocato altrove, per la gestione dell’input. Al premere del bottone, la cifra indicata sul display veniva aumentata di uno, ciclando agli estremi (9 diventa “-”, che diventa 1)

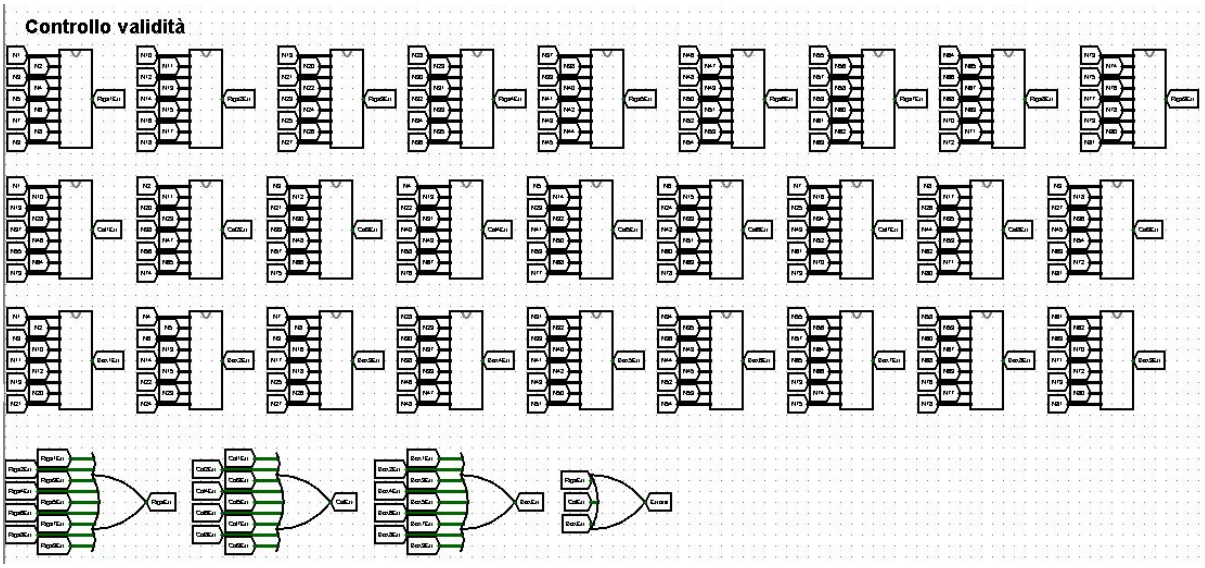


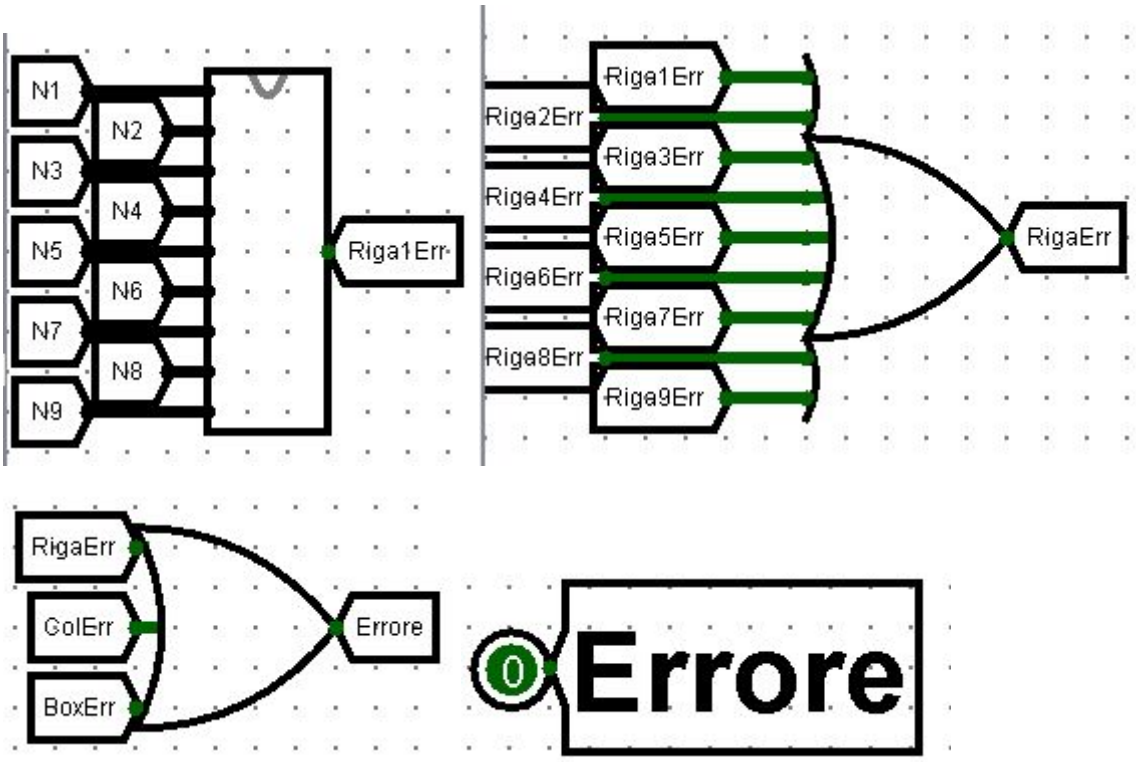
Il tunnel B (con numero da 1 a 81) veniva collegato al bottone, mentre il tunnel N agiva come input al display stesso, fornendo ad esso il valore attualmente memorizzato nel contatore personalizzato, che funzionava fondamentalmente come un contatore normale con la sola differenza di mandare in output UNKNOWN al posto di 0, in modo da visualizzare “-” sul display.



Controllo degli errori

Il controllo degli errori viene effettuato molto semplicemente usando il componente “checkNove” illustrato in precedenza.
Per ogni riga, colonna e sotto-griglia 3x3 (da me chiamata “box”) viene effettuato un controllo per rilevare eventuali doppioni, e nel caso ne venga trovato almeno uno il led di errore viene acceso.
Di seguito è possibile vedere le parti salienti del processo, che è molto ripetitivo

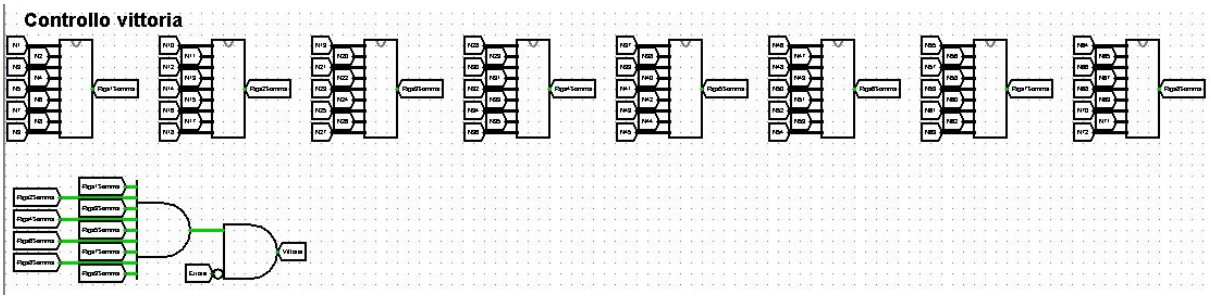


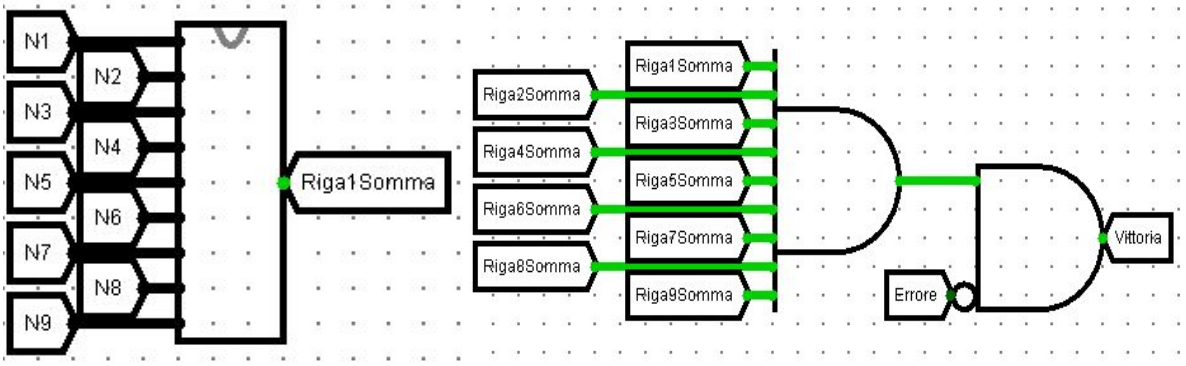


Controllo di fine partita

Il controllo di fine partita viene effettuato principalmente col componente “SommaNove”.
Si va a controllare, per ogni riga, se la somma delle cifre in essa contenuta è pari a 45 (la somma dei primi 9 naturali).
Se questa condizione è verificata per ogni riga, e non ci sono errori (doppioni), allora il gioco è concluso.

Esistono molte altre soluzioni al problema di determinare il termine della partita (per esempio, usare un register file per tenere traccia delle occorrenze di ogni cifra), ma ritengo che la mia sia il miglior compromesso tra semplicità e prestazioni.




N1
N2
N3
N4
N5
N6
N7
N8
N9
Riga1Somma
Riga2Somma
Riga4Somma
Riga6Somma
Riga8Somma
Riga1Somma
Riga3Somma
Riga5Somma
Riga7Somma
Riga9Somma
Errore
Vittoria

Considerazioni finali

Nello scegliere il progetto, ho preferito creare qualcosa di interessante ma non eccessivamente complesso, cercando di focalizzarmi invece sul creare un circuito chiaro e ordinato (cosa che spero di essere riuscito a fare).
Nel cercare di fare questo, ho incontrato delle limitazioni tecniche di logisim, che mi hanno "costretto" a leggerne il sorgente e creare una libreria java.
Proprio quest'ultima attività (comunque a mio parere non scollegata dall'argomento del corso, data la necessità di giostrarsi tra varie variabili come porte, clock, banda ecc.) è stata complessa e dispendiosa in termini di tempo, data la quasi totale assenza di documentazione (o anche solo commenti nel codice). Tuttavia, mi ritengo soddisfatto del risultato, in quanto il risultato finale è senza dubbio migliore di quello che avrei potuto costruire utilizzando solo i componenti standard.

Possibili migliorie future

Il progetto è senz'altro migliorabile, in particolare mi vengono in mente tre estensioni molto interessanti:

- Inserire un registro all'interno del display in modo da usare solo due componenti (display e tunnel), eliminando quindi il contatore "nascosto"
- Permettere di "bloccare" determinate celle, proprio come in un vero sudoku dove il valore di alcune celle è dato e non modificabile
- Far risolvere un dato sudoku direttamente al calcolatore

Il primo punto è sicuramente fattibile, ma decisamente più complicato di quello che sembra, anche a causa della mancanza di documentazione. Il mio intento originale era di fare proprio questo, ma non sono riuscito a concludere in tempo e quindi ho optato per il contatore nascosto.

La seconda modifica richiede qualche stravolgimento in più (a cominciare da una porta di input da 1 bit su ogni display o contatore per indicare che la cella deve essere bloccata), ma senza dubbio realizzabile.

Infine, l'ultima idea è chiaramente più complicata delle altre due, ma anche questa potenzialmente attuabile: gli ingredienti (la possibilità di leggere e impostare il valore di una cella) ci sono tutti, si tratta "solo" di implementare un algoritmo di risoluzione (per esempio, per backtracking).