Architettura degli Elaboratori: Rock-Paper-Scissors FSMD Verilog/SIS

Simone Di Maria (VRXXXXXX), Pietro Secchi (VRXXXXXX)

21 febbraio 2024

Indice

1 Specifiche del Progetto

Viene chiesto di realizzare il circuito di un dispositivo per la gestione di partite di Morra Cinese, noto anche come Sasso-Carta-Forbici.
Il dispositivo è implementato in Verilog e SIS, due linguaggi di descrizione hardware. Il circuito riceverà in input le mosse di due player e verranno giocate da un minimo di 4 ad un massimo di 19 manche. Vince il primo giocatore che riesce a vincere due manche in più del proprio avversario, a patto di aver giocato almeno quattro manche. Si devono giocare un minimo di quattro (4) manche. Ad ogni manche, il giocatore vincente della manche precedente non può ripetere l'ultima mossa utilizzata. Nel caso lo facesse, la manche non sarebbe valida ed andrebbe ripetuta (quindi, non conteggiata). Ad ogni manche, in caso di pareggio la manche viene conteggiata. Alla manche successiva, entrambi i giocatori possono usare tutte le mosse.

2 Struttura del progetto

Il circuito è realizzato utilizzando il modello **FSMD** (Finite State Machine e Datapath) ed è quindi composto da 2 parti: un controllore relizzato tramite una Macchina a Stati Finiti (**FSM**) ed un'unità di elaborazione realizzata tramite il modello **DATAPATH**.
Di seguito sono riportati i pin Input/Output del circuito.

2.1 Inputs

- `PRIMO [2 bit]`: mossa scelta dal primo giocatore. Le mosse hanno i seguenti codici:
  - `00: Nessuna mossa`;
  - `01: Sasso`;
  - `10: Carta`;
  - `11: Forbice`;
- `SECONDO [2 bit]`: mossa scelta dal secondo giocatore. Le mosse hanno gli stessi codici del primo giocatore.
- `INIZIA [1 bit]`: quando vale `1`, riporta il sistema alla configurazione iniziale, considerato dunque come il segnale di "reset". Inoltre, la concatenazione degli ingressi `PRIMO` e `SECONDO` viene usata per specificare il numero massimo di manche oltre le quattro obbligatorie. Ad esempio, se si inserisse il valore `PRIMO = 00` e `SECONDO = 01`, si indicherebbe di giocare al più 5 manche (le 4 obbligatorie, più il valore `1` indicato da `0001`). Quando vale `0`, la manche prosegue normalmente.

2.2 Outputs

- MANCHE [2 bit]: fornisce il risultato della partita con la seguente codifica:
  - 00: La partita non è terminata;
  - 01: La partita è terminata, ed ha vinto il Player 1;
  - 10: La partita è terminata, ed ha vinto il Player 2;
  - 11: La partita è terminata in pareggio;
- PARTITA [2 bit]: fornisce il risultato dell'ultima manche giocata con la seguente codifica:
  - 00: Manche non valida;
  - 01: Manche vinta dal Player 1;
  - 10: Manche vinta dal Player 2;
  - 11: Manche pareggiata;

2.3 Finite State Machine e Datapath

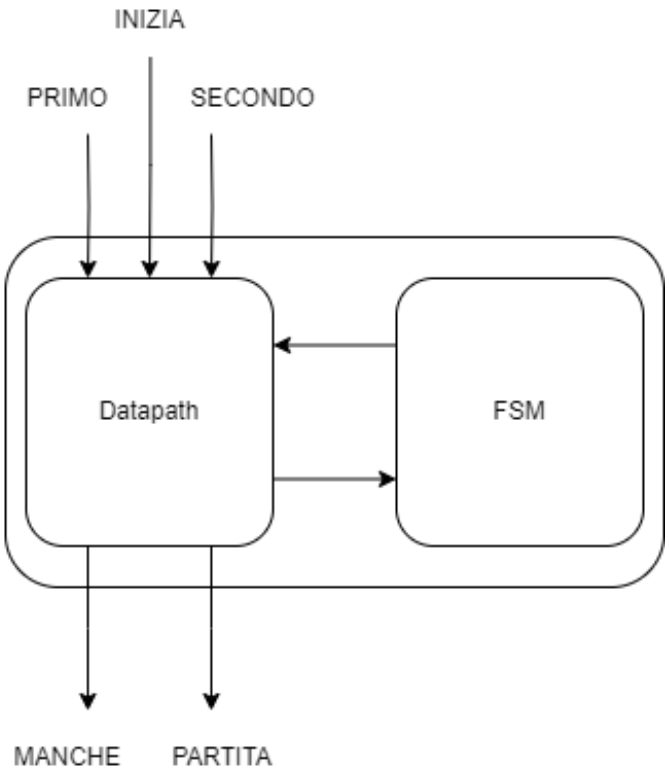
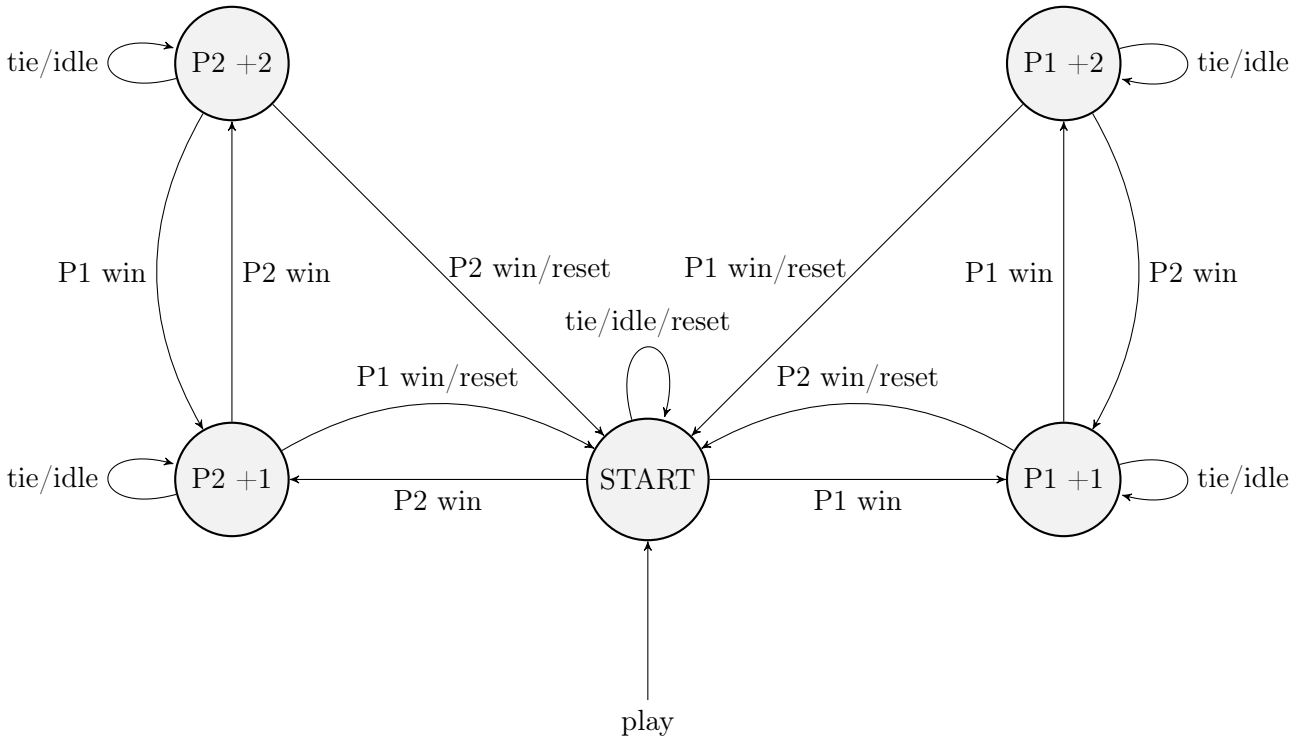


Figura 1: FSMD: Finite State Machine e Datapath

3 Controllore (FSM)

Il controllore, il cui diagramma di transizione degli stati (State Transition Graph) viene riportato qui di seguito, è progettato mediante il modello Finite State Machine di Mealy. Per le scelte di design invece, si fa spesso riferimento al [**paper1**] dei riferimenti bibliografici.

3.1 State Transitions Graph (STG)



Si noti l'assenza di uno stato finale. Si considera una partita terminata nel momento in cui il circuito determinerà un player vincitore. Dopodichè, viene richiesto all'utente il "reset" manuale del dispositivo. Siccome il compito dello STG è semplicemente quello di tenere traccia di quale giocatore e di quanto è in vantaggio, non è possibile determinare uno stato finale nel nostro STG. Di conseguenza, ogni stato potenzialmente è uno stato finale, in base al numero di manche giocate. Si consideri ad esempio, il caso in cui vengono giocate il numero massimo di manche, ma al momento della fine dell'ultima manche, nessun giocatore è in vantaggio di due punti, bensì uno dei due, di un solo punto. A quel punto verrà assegnata la vittoria al giocatore in vantaggio (quindi con il numero maggiore di vittorie). Un'alternativa considerata durante la fase di progettazione, è stata quella di aggiungere altri due stati, `P1_WINNER` e `P2_WINNER`, rendendoli stati finali; tuttavia ciò avrebbe portato complicazioni nell'implementazione in *SIS*.

Scelta Progettuale n. 1

Si è preferito definire il *STG* con l'insieme degli stati finali (F) vuoto, $F := \emptyset$. Tale scelta è stata guidata dalla volontà di utilizzare il minor numero possibile di stati, dal rendere facile l'implementazione in SIS, e dal voler mantenere il STG con il semplice compito di tenere traccia del giocatore in vantaggio.

3.2 Architettura del controllore

Il controllore è stato progettato con il seguente diagramma di blocco per le FSM di Mealy: Questa struttura ci permette di individuare due caratteristiche principali della nostra FSM:

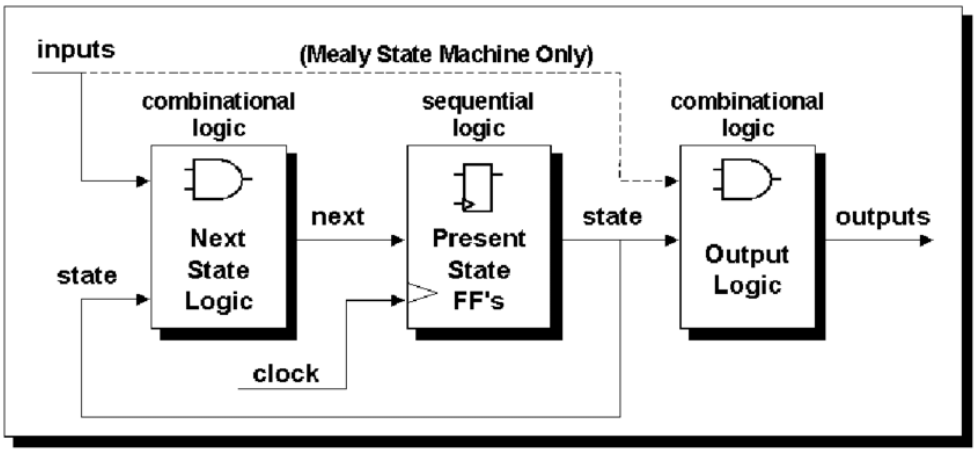


Figura 2: Finite State Machine (FSM) block diagram, from **[paper1]**

1. **Suddivisione tra blocchi "sequenziali" e blocchi "combinatori"**: Un circuito logico combinatorio è un circuito la cui uscita dipende *esclusivamente* dai suoi ingressi correnti e non ha alcuna memoria. Per questo motivo sarà anche *time-insensitive*. I circuiti sequenziali, invece, sono costruiti utilizzando circuiti combinatori ed elementi di memoria chiamati "flip-flop". Inoltre, il loro "trigger" è dato dal `clock`, rendendoli *time-sensitive*. Questi circuiti generano un output che dipende dallo stato attuale e da quello precedente. In particolare, nel nostro caso il blocco combinatorio gestirà l'esito delle manche restituendo in output, ad esempio, il prossimo stato, passandolo al blocco sequenziale che imposterà e memorizzerà lo stato corrente della macchina.

2. **Determinazione output**: Notiamo come l'`Output Logic` della nostra FSM [3.2] dipenda dallo stato attuale **e** dagli input. Di conseguenza la funzione di output della nostra FSM avrà come dominio l'insieme degli input che gli forniamo e lo stato corrente della macchina. Inoltre, deduciamo che le uscite avranno una reazione immediata agli ingressi. Nel caso la funzione di output non reagisse immediatamente alla modifica dell'inputs, e quindi dipendesse solo dallo stato corrente della macchina, staremmo parlando di FSM di Moore, e non più di Mealy.

Scelta Progettuale n. 2

Si è scelto di usare come modello di FSM, la FSM di Mealy. Questa scelta caratterizza la nostra FSM nella distinzione tra blocchi sequenziali e blocchi combinatori. In particolare, il blocco combinatorio che si occupa della logica di output, sarà funzione dell'insieme di inputs e dello stato corrente. Sebbene sia possibile implementare il circuito con una FSM di Moore, la FSM di Mealy ci permette più flessibilità e semplicità nella gestione delle manche di gioco.

3.3 Implementazione Verilog

Di seguito sono riportate le principali scelte di stile e codifica prese nell'implementazione del circuito in Verilog. Per il codice completo si invita il lettore a consultare direttamente il codice sorgente poichè ben commentato.

3.3.1 Coding Style: *lowRISC Verilog Coding Style Guide*

Nell'implementazione Verilog del circuito, si è deciso di aderire alle convenzioni elencate nella lowRISC Verilog Coding Style Guide. Tra i motivi di questa scelta è presente la volontà di scrivere codice nel modo più leggibile, coerente, pulito e standardizzato possibile per favorire la facile comprensione del codice ai lettori non-autori di questo progetto.

> *To quote the Google C++ style guide: "Creating common, required idioms and patterns makes code much easier to understand."*

Alcune delle "best practices" adottate nel codice sono: la Tabular Alignment, l'Indentation, il Naming, la preferenza al data type `logic`, l'uso dei costrutti `always_comb`, `unique case` e *blocking assignment* `=` per la logica combinatoria, l'uso dei costrutti `always_ff` e *non-blocking assignment* `<=` per la logica sequenziale, etc etc.

Scelta Progettuale n. 3

Nell'implementazione del circuito in Verilog, si è deciso di seguire la lowRISC Verilog Coding Style Guide per ottenere un risultato più leggibile, coerente, pulito e standardizzato possibile e favorire la comprensione del codice ai lettori non-autori di questo progetto.

3.3.2 Coding Style: *Two always block*

Il "Two always block" style è una buona pratica per l'implementazione della Finite State Machine (FSM) in Verilog. Questo approccio offre una serie di vantaggi che lo rendono particolarmente adatto per questa task. In primo luogo, una priorità comune anche ad altre scelte di design prese nel corso di questo progetto, è la leggibilità del codice. Per questo motivo, utilizzare due blocchi always separati per il comportamento combinatorio (Next State Logic e Output Logic) e lo stato sequenziale (Present State FF), rende più chiaro il partizionamento delle componenti. Inoltre, questo stile di codifica aiuta a mantenere una struttura modulare e organizzata. Separando il comportamento combinatorio e lo stato sequenziale, è più semplice per il progettista concentrarsi su ciascun aspetto separatamente. Inoltre, questo stile favorisce una migliore scalabilità e

	Two always block coding style	One always block coding style (12%-83% larger)	Onehot, two always block coding style	Three always block coding style w/ registered outputs
fsm_cc4 (4 states, simple)	37 lines of code	47 lines of code (12%-27% larger)	42 lines of code	40 lines of code
fsm_cc7 (10 states, simple)	50 lines of code	79 lines of code (32%-58% larger)	53 lines of code	60 lines of code
fsm_cc8 (10 states, moderate complexity)	80 lines of code	146 lines of code (70%-83% larger)	86 lines of code	83 lines of code

Figura 3: Lines of RTL code required for different FSM coding styles, from [**paper1**]

manutenibilità del codice. Se in futuro si desidera estendere o modificare la FSM per aggiungere nuovi stati o comportamenti, la separazione dei blocchi always semplifica il processo di aggiornamento del codice senza dover riscrivere completamente la logica esistente.
In generale il "Two always block" risulta spesso più efficace rispetto ad altri approcci come il "One always block". L'argomento è largamente discusso dal [paper1] menzionato e preso in esame più volte nel corso di questo report, a cui si invita la lettura per una comprensione approfondita.

> *The one always block FSM coding style is more verbose, more confusing and more error prone than a comparable Two always block coding style.*

Scelta Progettuale n. 4

Nell'implementazione in Verilog della FSM, si è deciso di seguire il "Two always block" style, che risulta più efficace di altri approcci come il "One always block" o il "Three always block" style.

3.3.3 Codifica Stati

Per rappresentare gli stati della nostra FSM, ci siamo trovati a dover fare una scelta (non banale) sulla codifica da implementare. Una codifica adeguata per questo scopo deve essere: compatta, leggibile, e deve produrre risultati di sintesi efficienti. Tra le varie codifiche possibili, abbiamo identificato tre principali classificazioni comuni utilizzate per descrivere la codifica dello stato di una FSM:
1) *One-Hot Encoding (OHE)* 2) *Binary Encoding* 3) *Gray Code.*

One-Hot Encoding (OHE):
L'OHE rappresenta ciascun stato mediante un vettore binario di lunghezza pari al numero di stati possibili. In questa codifica **solo uno** dei bit del vettore che rappresenta il nostro stato è `1`, chiamato per questo motivo "*hot*". Tutti gli altri bit sono `0`. La distanza di Hamming di

questa codifica fra ogni stato è 2. Considerando gli stati del rappresentati nel STG[3.1], la OHE li codifica come segue:

$$\begin{aligned}
\text{P2+2:} &\quad \begin{bmatrix}1\ 0\ 0\ 0\ 0\end{bmatrix} \\
\text{P2+1:} &\quad \begin{bmatrix}0\ 1\ 0\ 0\ 0\end{bmatrix} \\
\text{START:} &\quad \begin{bmatrix}0\ 0\ 1\ 0\ 0\end{bmatrix} \\
\text{P1+1:} &\quad \begin{bmatrix}0\ 0\ 0\ 1\ 0\end{bmatrix} \\
\text{P1+2:} &\quad \begin{bmatrix}0\ 0\ 0\ 0\ 1\end{bmatrix}
\end{aligned}$$

Essendo un solo bit attivo, è necessario un solo flip-flop per ogni stato della FSM. Di conseguenza, la macchina a stati è già "decodificata": lo stato della macchina viene determinato semplicemente leggendo quale flip-flop è attivo. Inoltre, questa codifica è più efficiente a livello di consumi rispetto alla Binary Encoding, grazie alla caratteristica di tenere solo 1 bit acceso. Questa tecnica di codifica riduce lo spazio del nostro circuito combinatorio e, di conseguenza, la macchina a stati richiede meno livelli logici tra i registri, riducendone la complessità e aumentando la velocità.

Binary Encoding:
In questa codifica, gli stati vengono assegnati in sequenza binaria dove gli stati sono numerati a partire da `0`. Il numero di bit b necessari per codificare un numero n di stati è dato dalla seguente relazione:

$$b = \lceil \log_2 n \rceil \tag{1}$$

Poiché la codifica binaria utilizza il numero minimo di bit (flip-flop) per codificare una macchina, i flip-flop vengono sfruttati al massimo. Di conseguenza, più logica combinatoria è necessaria per decodificare ogni stato rispetto alla OHE. Il Binary Encoding richiede meno flip-flop rispetto all'OHE, ma la distanza di Hamming può essere maggiore del numero di bit (b).

Gray Code:
Il codice Gray, noto anche come codice binario riflesso, gli stati vengono assegnati in modo tale che le codifiche di stati consecutivi differiscano solo di un bit (distanza di Hamming unitaria). In questa codifica anche la relazione tra numero di bit e numero di stati è definita dalla formula[1]:

$$b = \lceil \log_2 n \rceil$$

Il numero di flip-flop utilizzati e la complessità della logica di decodifica sono gli stessi della codifica binaria. Ma in questo caso la distanza di Hamming è sempre 1.

Codifiche a confronto: Pro e Contro

Tabella 1: Vantaggi e Svantaggi delle varie codifiche di stato per le FSM

Codifica Stati	Pros	Cons
Binary Encoding	• Richiede meno flip-flop rispetto all'encoding One Hot. • Facile da implementare e comprendere. • Adatto per FSM con pochi stati.	• Richiede logica extra per decodificare lo stato attuale. • Non ottimale a livello di consumo energetico.
One-Hot Encoding	• Transizioni di stato dirette, nessuna necessità di decodifica. • Facile da comprendere e leggere. • Facile da estendere e modificare senza impattare altri stati.	• Richiede un gran numero di flip-flop.
Gray Code	• Distanza di Hamming unitaria. • Riduce il consumo di potenza.	• Richiede logica extra per decodificare lo stato attuale. • Comprensione più complessa rispetto alla Binary Encoding e la OHE. • Non ottimale per FSM con pochi stati, poiché richiede più flip-flop rispetto all'encoding binario.

Considerando la rappresentazione degli stati con la codifica One-Hot, notiamo come potremmo sfruttare a nostro favore il "movimento" dei bit "hot". Si consideri infatti la codifica del vincitore della manche: nel caso vinca il Player 1, l'output `MANCHE` sarà `01`, nel caso in cui vinca Player 2, l'output `MANCHE` sarà `10`. Possiamo dunque aiutarci con la "direzione" del bit di `MANCHE`, per capire verso quale direzione si muoverà la FSM e in che direzione si muoverà il bit "hot" per rappresentare il prossimo stato (`next_state`).
Ad esempio:

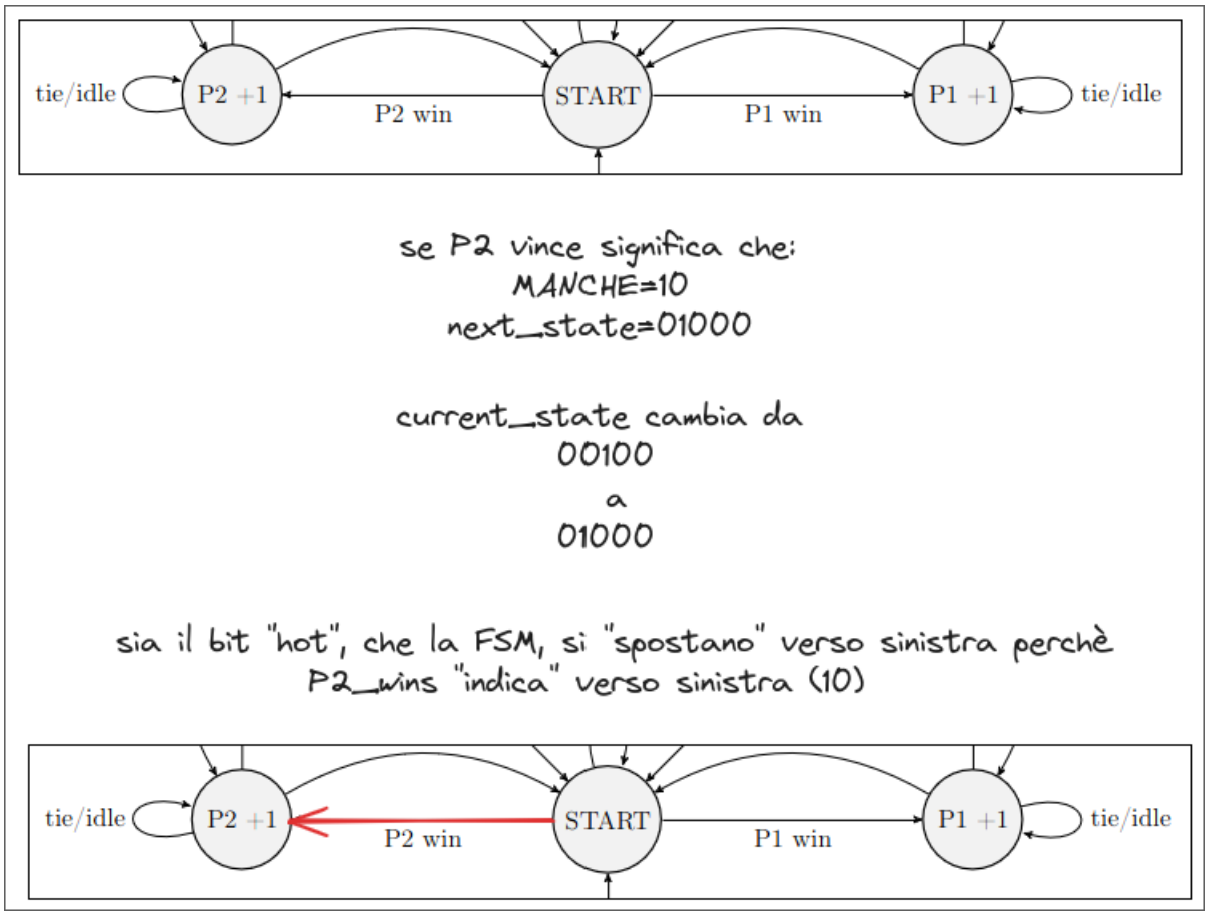


Figura 4: Rappresentazione del "movimento" dei bit e della FSM al vincere di un Player

La codifica OHE, di conseguenza, aiuterebbe molto alla comprensione e alla lettura del circuito.

Scelta Progettuale n. 5

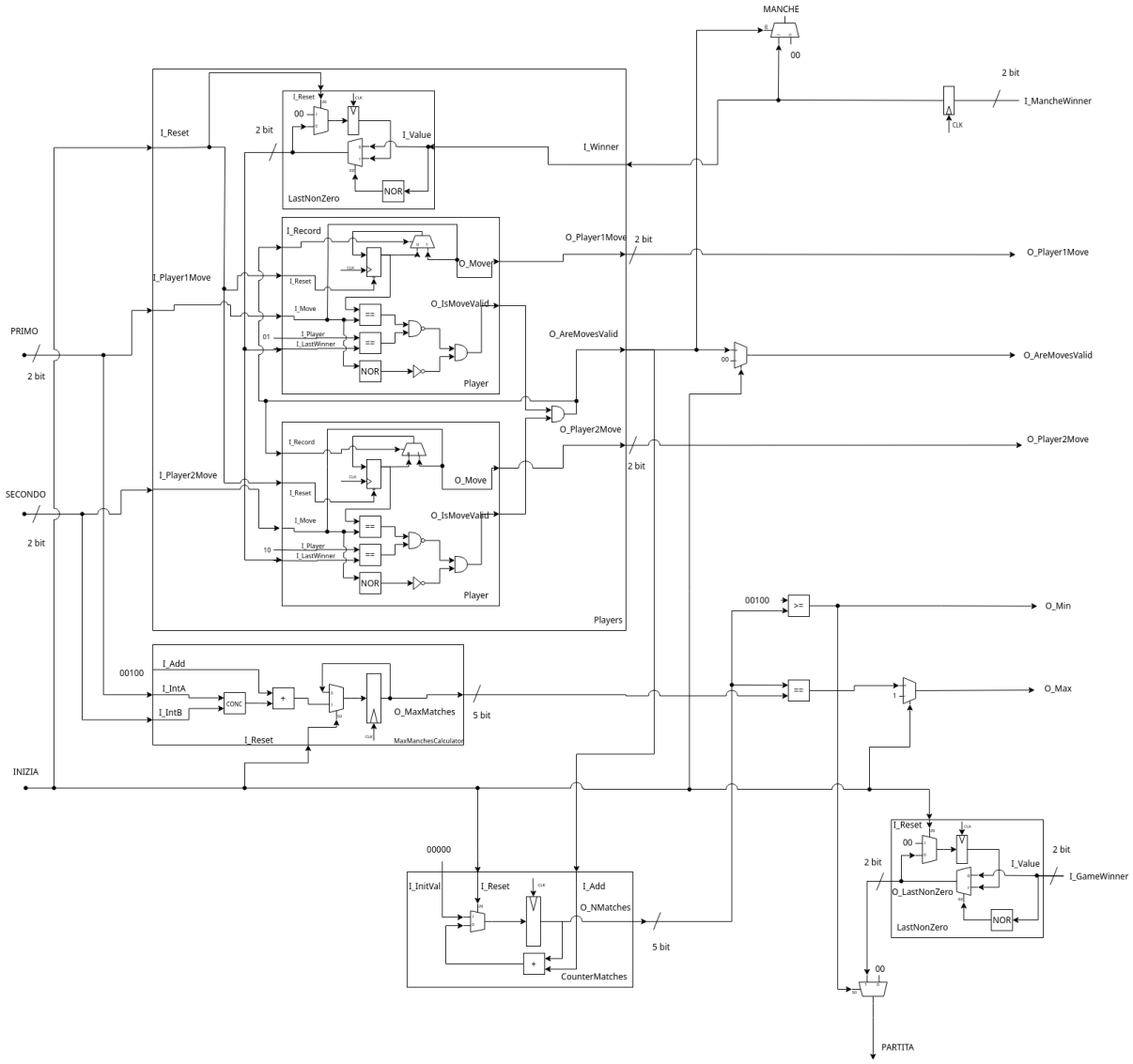
Per la codifica degli stati della FSM in Verilog è stata scelta la *One-Hot Encoding*. La OHE risulta il perfetto compromesso tra leggibilità, consumi, velocità ed area utilizzata. Per di più, secondo [**paper1**], questa codifica è la più coerente in contesto di progettazione FPGA:

> *FPGA vendors frequently recommend using a onehot state encoding style because flip-flops are plentiful in an FPGA and the combinational logic required to implement a onehot FSM design is typically smaller than most binary encoding styles. Since FPGA performance is typically related to the combinational logic size of the FPGA design, onehot FSMs typically run faster than a binary encoded FSM with larger combinational logic blocks.*

4 Unità di Elaborazione

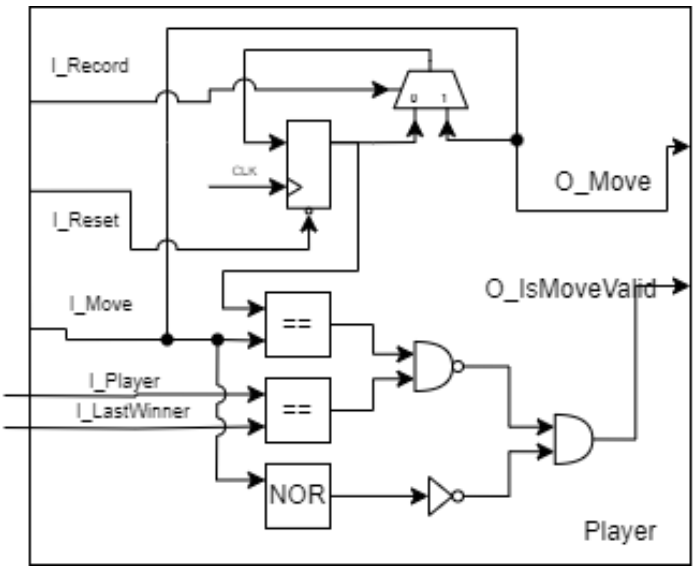
L'unità di elaborazione è realizzata tramite il modello ***datapath***, ed è così schematizzata:

4.1 Datapath



Di cui si hanno 5 sottomoduli principali:

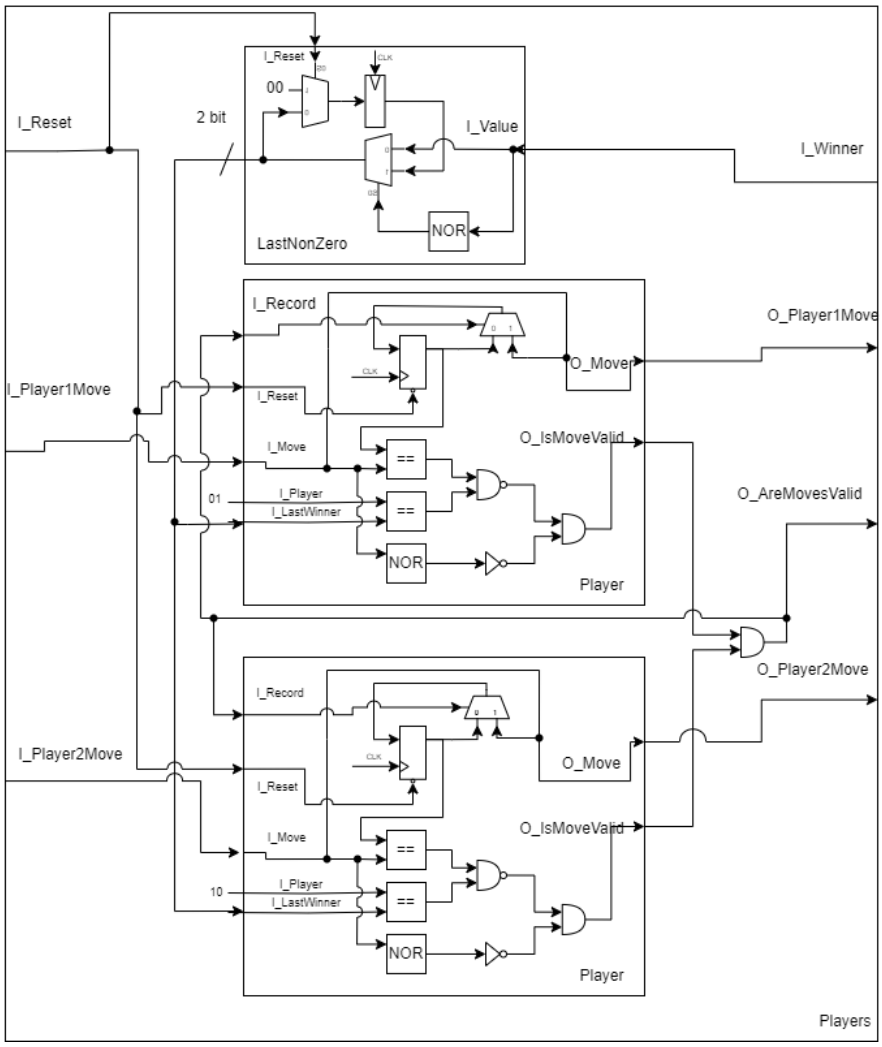
4.1.1 `Player` module



Il modulo **`Player`** si occupa di controllare la validità delle mosse, prende in input l'ultimo vincitore, il nome del giocatore che rappresenta (01 o 10), la mossa e un bit per decidere se registrare la mossa. Utilizza un registro `D` con `Reset` per resettare lo stato dell'ultima mossa in memoria. Per capire se la mossa è valida usa la seguente tabella di verità:

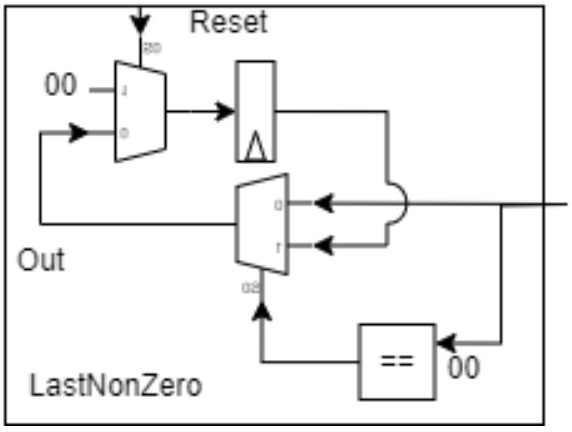
EqualLastMove	EqualLastWinner	Invalid	¬(EqualLastMove ∧ EqualLastWinner) ∧ ¬Invalid
F	F	F	T
F	F	T	F
F	T	F	T
F	T	T	F
T	F	F	T
T	F	T	F
T	T	F	F
T	T	T	F

4.1.2 `Players` module



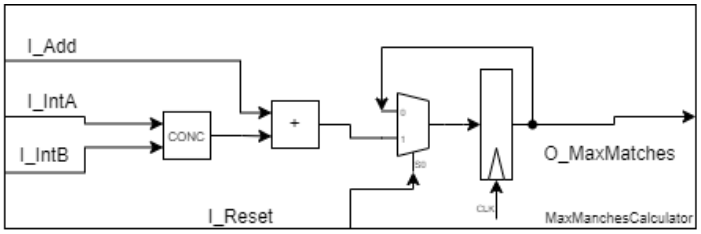
Il modulo **`Players`** include i 2 moduli **`Player`** e un modulo **`LastNonZero`**, esegue un operazione di `AND` con gli output dei **`Player`** e il risultato viene restituito di nuovo in input ai **`Player`** per registrare la mossa se valida (1) utilizzando un multiplexer.

4.1.3 `LastNonZero` module



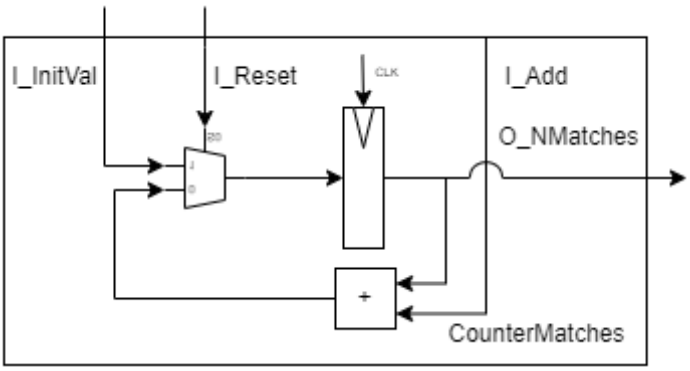
Questo modulo prende in input due bit e in output mette l'ultima coppia di bit dove entrambi non erano 0, questo viene usato in **`Player`** per dare ai singoli player l'ultimo vincitore valido e viene usato anche per mantenere in memoria lo stato del vincitore della partita. Ha un bit di reset per pulire i registri.

4.1.4 `MaxManchesCalculator` module



Entra in funzione quando `INIZIA` è a 1 ed esegue una concatenazione di `PRIMO` e `SECONDO`, sommando poi `I_Add`, che in questo caso è `00100` (4 in decimale) per avere in output il numero massimo di manche. Viene resettato con `I_Reset`.

4.1.5 `Counter` module



Conta il numero di manche giocate. Il modulo prende come input `I_Add` che in questo caso è `O_AreMovesValid` di **`Players`**, quindi se le mosse sono valide il counter viene incrementato.

Scelta Progettuale n. 6

Per risparmiare logica i controlli per confrontare un valore a `00` vengono fatti con un operatore `NOR`. L'input `I_Add` del counter di partite è direttamente `O_AreMovesValid` da **`Players`**. Le mosse dei giocatori prese in input dal modulo **`Players`** vengono passate direttamente ai moduli **`Player`**, che le restituiranno in output sempre, anche quando non sono valide. Si sarebbe potuto far passare il segnale delle mosse senza farlo passare per i suddetti moduli; abbiamo deciso però di rendere il datapath più pulito. Il numero di manche viene registrato con registri a `5 bit` poichè il range di valori che dobbiamo assumere è $[0, 19]$ e $\lceil \log_2 19 \rceil = 5$ abbiamo quindi bisogno di almeno 5 registri. Un'altra opzione sarebbe stata quella di usare un registro ad `8 bit` poichè più comune (essendo potenza di 2).

5 Realizzazione del circuito in formato `blif`

Per poter realizzare il circuito abbiamo prima di tutto assegnato alla **FSM** - descritta sotto forma di **STG** - gli stati mediante la funzione `state_assign jedi`, che si è occupata anche di mappare la **FSM** come circuito sequenziale. A questo punto, valutato il corretto funzionamento rispettivamente di **FSM** e **DATAPATH**, abbiamo unito elaboratore e controllore mediante un unico file `FSMD.blif`.

Warning

Nel caricare il file del circuito, vengono prodotti alcuni warning che indicano il mancato uso del `COUT` di qualche modulo. Poiché questi bit non sono rilevanti, siamo riusciti a sopprimere il warning assegnando i `COUT` ad un modulo extra che semplicemente ritorna in output quello che riceve in input, così da evitare di avere dei warning ogni volta che si carica il file `blif`.

5.1 Ottimizzazione e mapping del circuito

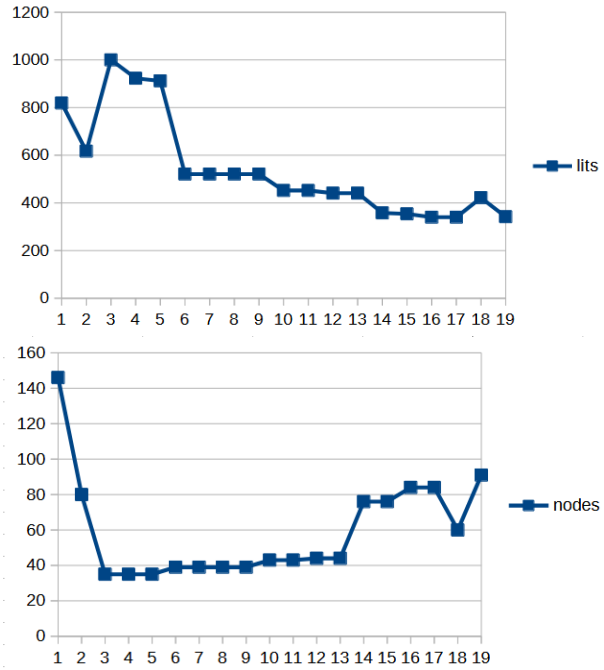
Siamo passati quindi all'ottimizzazione per area totale, utilizzando tutti gli script che sono disponibili all'interno della cartella di sis e questi sono i risultati per ogni comando nello script dopo diverse iterazioni (sono riportati di seguito anche i grafici che illustrano il variare di, rispettivamente, letterali e nodi, per ogni iterazione di ottimizzazione):

5.1.1 algebraic.script

```
stats pre-optimization: pi= 5 po= 4 nodes=146 latches=23 lits(sop)= 819

pi= 5 po= 4 nodes= 80 latches=23 lits(sop)= 617
pi= 5 po= 4 nodes= 35 latches=23 lits(sop)= 1000
pi= 5 po= 4 nodes= 35 latches=23 lits(sop)= 923
pi= 5 po= 4 nodes= 35 latches=23 lits(sop)= 912
pi= 5 po= 4 nodes= 39 latches=23 lits(sop)= 521
pi= 5 po= 4 nodes= 39 latches=23 lits(sop)= 521
pi= 5 po= 4 nodes= 39 latches=23 lits(sop)= 521
pi= 5 po= 4 nodes= 39 latches=23 lits(sop)= 521
pi= 5 po= 4 nodes= 43 latches=23 lits(sop)= 452
pi= 5 po= 4 nodes= 43 latches=23 lits(sop)= 452
pi= 5 po= 4 nodes= 44 latches=23 lits(sop)= 441
pi= 5 po= 4 nodes= 44 latches=23 lits(sop)= 441
pi= 5 po= 4 nodes= 76 latches=23 lits(sop)= 358
pi= 5 po= 4 nodes= 76 latches=23 lits(sop)= 354
pi= 5 po= 4 nodes= 84 latches=23 lits(sop)= 340
pi= 5 po= 4 nodes= 84 latches=23 lits(sop)= 340
pi= 5 po= 4 nodes= 60 latches=23 lits(sop)= 422

stats post-optimization: pi= 5 po= 4 nodes= 91 latches=23 lits(sop)= 342
```

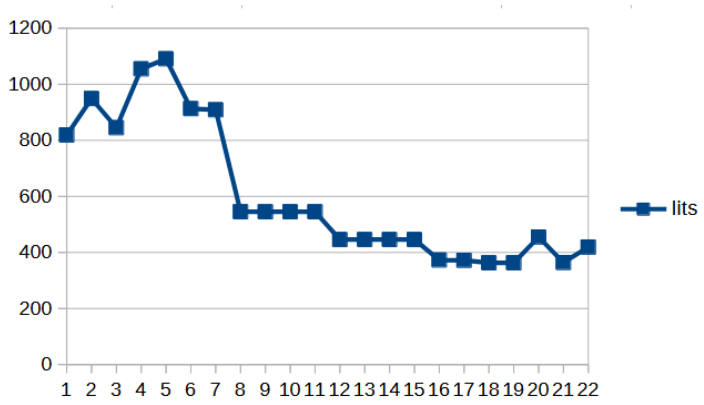

1200
1000
800
600
400
200
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
lits
160
140
120
100
80
60
40
20
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
nodes

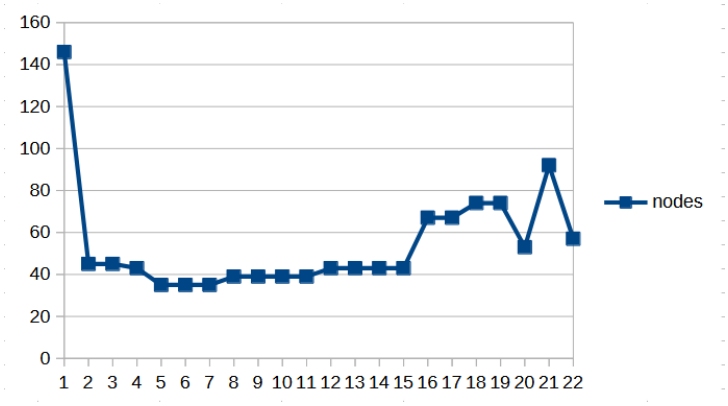
5.1.2 boolean.script

```
stats pre-optimization: pi= 5 po= 4 nodes=146 latches=23 lits(sop)= 819

pi= 5 po= 4 nodes= 45 latches=23 lits(sop)= 949
pi= 5 po= 4 nodes= 45 latches=23 lits(sop)= 845
pi= 5 po= 4 nodes= 43 latches=23 lits(sop)= 1055
pi= 5 po= 4 nodes= 35 latches=23 lits(sop)= 1091
pi= 5 po= 4 nodes= 35 latches=23 lits(sop)= 913
pi= 5 po= 4 nodes= 35 latches=23 lits(sop)= 909
pi= 5 po= 4 nodes= 39 latches=23 lits(sop)= 545
pi= 5 po= 4 nodes= 39 latches=23 lits(sop)= 545
pi= 5 po= 4 nodes= 39 latches=23 lits(sop)= 545
pi= 5 po= 4 nodes= 39 latches=23 lits(sop)= 545
pi= 5 po= 4 nodes= 43 latches=23 lits(sop)= 446
pi= 5 po= 4 nodes= 43 latches=23 lits(sop)= 446
pi= 5 po= 4 nodes= 43 latches=23 lits(sop)= 446
pi= 5 po= 4 nodes= 43 latches=23 lits(sop)= 446
pi= 5 po= 4 nodes= 67 latches=23 lits(sop)= 373
pi= 5 po= 4 nodes= 67 latches=23 lits(sop)= 372
pi= 5 po= 4 nodes= 74 latches=23 lits(sop)= 363
pi= 5 po= 4 nodes= 74 latches=23 lits(sop)= 363
pi= 5 po= 4 nodes= 53 latches=23 lits(sop)= 455
pi= 5 po= 4 nodes= 92 latches=23 lits(sop)= 364

stats post-optimization: pi= 5 po= 4 nodes= 57 latches=23 lits(sop)= 419
```



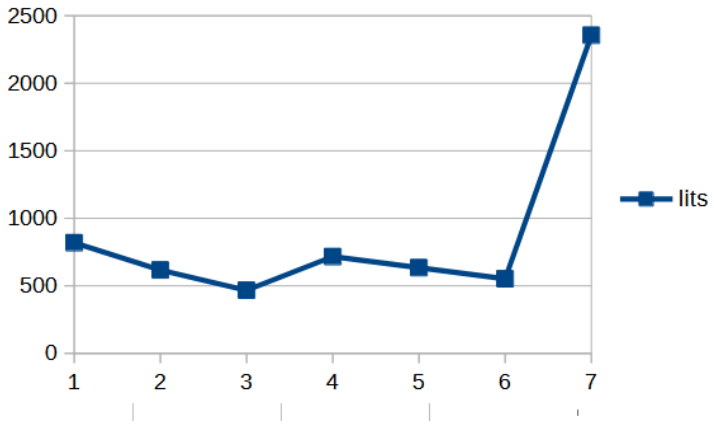


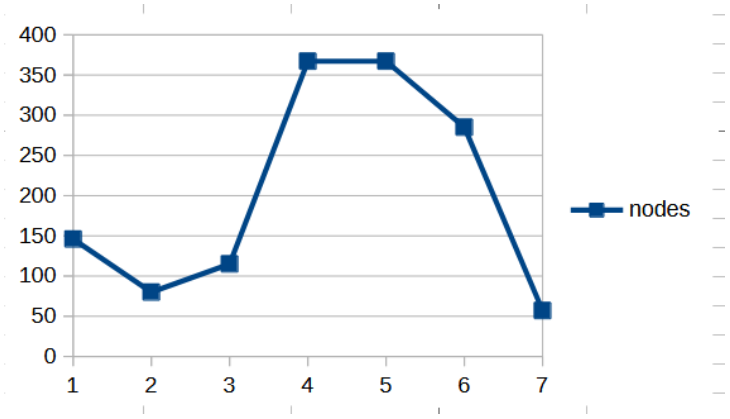
5.1.3 delay.script

```
stats pre-optimization: pi= 5 po= 4 nodes=146 latches=23 lits(sop)= 819

pi= 5 po= 4 nodes= 80 latches=23 lits(sop)= 617
pi= 5 po= 4 nodes=115 latches=23 lits(sop)= 467
pi= 5 po= 4 nodes=367 latches=23 lits(sop)= 716
pi= 5 po= 4 nodes=367 latches=23 lits(sop)= 634
pi= 5 po= 4 nodes=285 latches=23 lits(sop)= 552
pi= 5 po= 4 nodes= 57 latches=23 lits(sop)= 2355
hanging, stopping execution...
```

Al comando `red_removal`, che genera input causali per semplificare il `blif`, si blocca. Dopo 10 minuti di esecuzione non si riceve output e si considerà perciò bloccato.



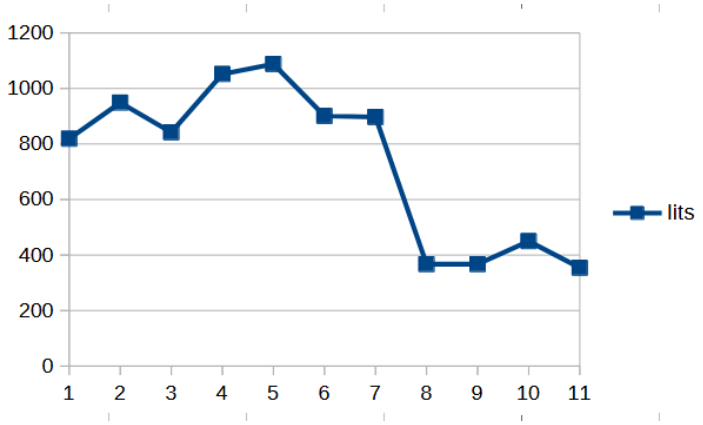


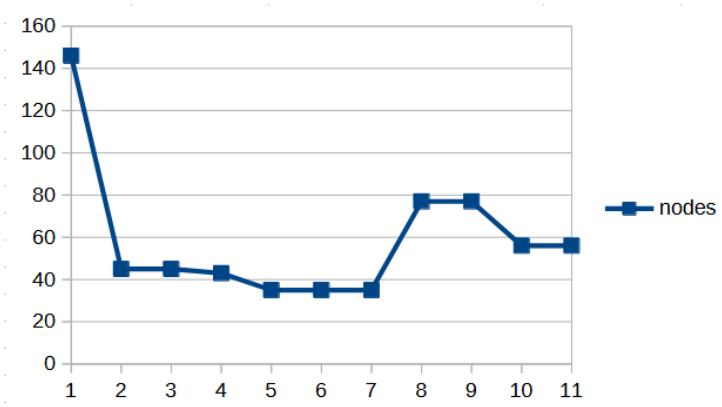
5.1.4 rugged.script

```
stats pre-optimization: pi= 5 po= 4 nodes=146 latches=23 lits(sop)= 819

pi= 5 po= 4 nodes= 45 latches=23 lits(sop)= 949
pi= 5 po= 4 nodes= 45 latches=23 lits(sop)= 842
pi= 5 po= 4 nodes= 43 latches=23 lits(sop)= 1052
pi= 5 po= 4 nodes= 35 latches=23 lits(sop)= 1088
pi= 5 po= 4 nodes= 35 latches=23 lits(sop)= 901
pi= 5 po= 4 nodes= 35 latches=23 lits(sop)= 897
pi= 5 po= 4 nodes= 77 latches=23 lits(sop)= 367
pi= 5 po= 4 nodes= 77 latches=23 lits(sop)= 367
pi= 5 po= 4 nodes= 56 latches=23 lits(sop)= 450

stats post-optimization: pi= 5 po= 4 nodes= 56 latches=23 lits(sop)= 354
```



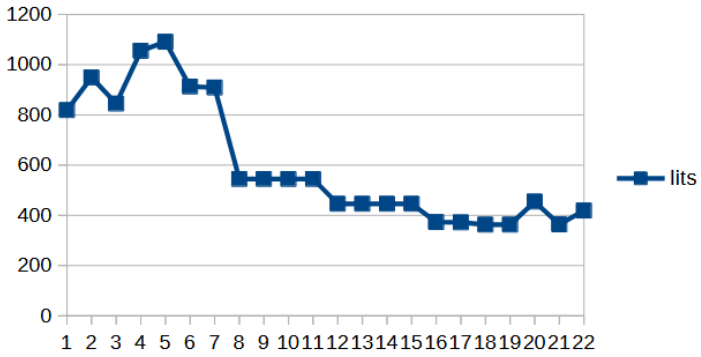

160
140
120
100
80
60
40
20
0
1
2
3
4
5
6
7
8
9
10
11
nodes

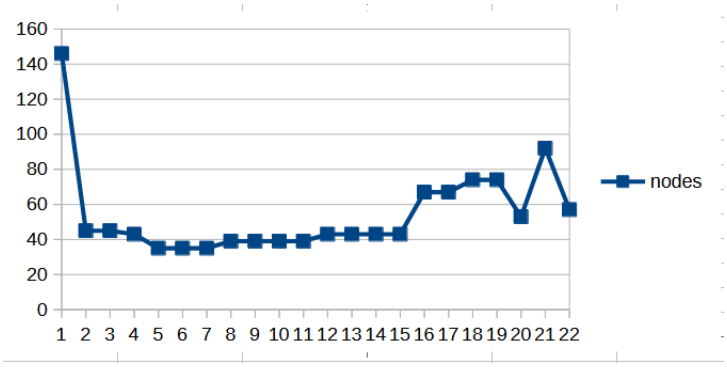
5.1.5 script

```
stats pre-optimization: pi= 5 po= 4 nodes=146 latches=23 lits(sop)= 819

pi= 5 po= 4 nodes= 45 latches=23 lits(sop)= 949
pi= 5 po= 4 nodes= 45 latches=23 lits(sop)= 845
pi= 5 po= 4 nodes= 43 latches=23 lits(sop)= 1055
pi= 5 po= 4 nodes= 35 latches=23 lits(sop)= 1091
pi= 5 po= 4 nodes= 35 latches=23 lits(sop)= 913
pi= 5 po= 4 nodes= 35 latches=23 lits(sop)= 909
pi= 5 po= 4 nodes= 39 latches=23 lits(sop)= 545
pi= 5 po= 4 nodes= 39 latches=23 lits(sop)= 545
pi= 5 po= 4 nodes= 39 latches=23 lits(sop)= 545
pi= 5 po= 4 nodes= 39 latches=23 lits(sop)= 545
pi= 5 po= 4 nodes= 43 latches=23 lits(sop)= 446
pi= 5 po= 4 nodes= 43 latches=23 lits(sop)= 446
pi= 5 po= 4 nodes= 43 latches=23 lits(sop)= 446
pi= 5 po= 4 nodes= 43 latches=23 lits(sop)= 446
pi= 5 po= 4 nodes= 67 latches=23 lits(sop)= 373
pi= 5 po= 4 nodes= 67 latches=23 lits(sop)= 372
pi= 5 po= 4 nodes= 74 latches=23 lits(sop)= 363
pi= 5 po= 4 nodes= 74 latches=23 lits(sop)= 363
pi= 5 po= 4 nodes= 53 latches=23 lits(sop)= 455
stats post-optimization:  pi= 5 po= 4 nodes= 92 latches=23 lits(sop)= 364

pi= 5 po= 4 nodes= 57 latches=23 lits(sop)= 419
```





Osservazioni

Possiamo notare che il numero di `latch` è rimasto uguale, questo significa che abbiamo utilizzato il minimo numero di latch necessari per il progetto.
Possiamo anche notare che **`script.rugged`** è stato il più performante nello semplificare i nodi, arrivando ad un numero di `56` nodi, ma è secondo nel semplificare i letterali con un risultato di `354` letterali, contro i `342` di **`algebraic.script`**.

5.2 Esecuzione del mapping

Qui di seguito viene riportato l'output dei comandi utilizzati per eseguire il mapping del progetto (dopo l'ottimizzazione) con la libreria tecnologica `synch.genlib`.

```
Mapping post-ottimizzazione
UC Berkeley, SIS 1.3.6 (compiled 2017-10-27 16:08:57)
sis> rl FSMD.blif
sis> print_stats
FSMD            pi= 5   po= 4   nodes=146       latches=23
lits(sop)= 819
sis> source script.rugged
sis> print_stats
FSMD            pi= 5   po= 4   nodes= 56       latches=23
lits(sop)= 354
sis>  read_library synch.genlib
sis>  map -m 0 -s
>>> before removing serial inverters <<<
# of outputs:          27
total gate area:       6248.00
maximum arrival time: (39.20,39.20)
maximum po slack:     (-4.00,-4.00)
minimum po slack:     (-39.20,-39.20)
total neg slack:      (-509.40,-509.40)
# of failing outputs:  27
>>> before removing parallel inverters <<<
# of outputs:          27
total gate area:       6120.00
maximum arrival time: (39.20,39.20)
maximum po slack:     (-4.00,-4.00)
minimum po slack:     (-39.20,-39.20)
total neg slack:      (-504.60,-504.60)
# of failing outputs:  27
# of outputs:          27
total gate area:       6024.00
maximum arrival time: (39.20,39.20)
maximum po slack:     (-4.00,-4.00)
minimum po slack:     (-39.20,-39.20)
total neg slack:      (-503.60,-503.60)
# of failing outputs:  27
sis> print_stats
FSMD            pi= 5   po= 4   nodes=152       latches=23
lits(sop)= 385
sis>
```

Osservazioni

Prima di ottimizzazione e di mapping tecnologico, il circuito presentava 146 nodi e 819 letterali. Dopo l'ottimizzazione, ma prima del mapping tecnologico il circuito presentava 56 nodi e 354 letterali. Infine, post-mapping e post-ottimizzazione notiamo come i letterali aumentino rispetto all'ottimizzazione, arrivando a 152 nodi e 385 letterali.

Riferimenti bibliografici

[1] Clifford E. Cummings (ICU-2002), The Fundamentals of Efficient Synthesizable Finite State Machine Design using NC-Verilog and BuildGates.

[2] Clifford E. Cummings, Heath Chambers (SNUG-2019), Finite State Machine (FSM) Design & Synthesis using SystemVerilog - Part I.