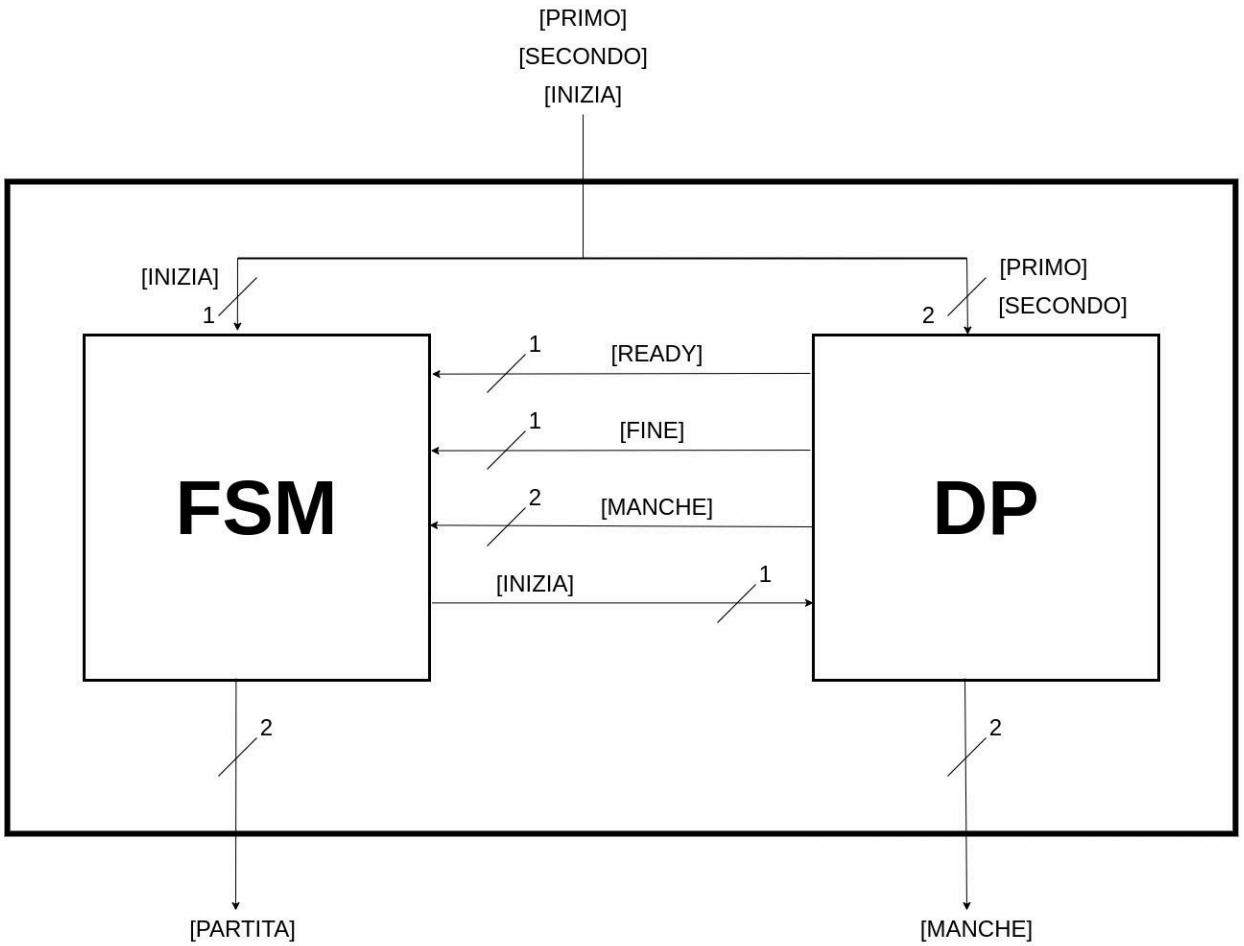
ARCHITETTURA DEGLI ELABORATORI: ELABORATO IN SIS E VERILOG

VR500402, VR500356
Loris Hoxhaj, Mattia Nicolis
A.A. 2023-24

Sommario

Si desidera realizzare un circuito per la gestione di una partita (composta da più manche) del "**Gioco della morra**" o "**sasso, carta, forbice**".
Ricevuti in ingresso le mosse dei due giocatori e il bit di inizio della partita, il programma restituisce alla fine di ogni manche il vincitore di essa e, infine, il vincitore dell'intera partita.
Quest'ultimo viene definito attraverso la condizione di vincita, ovvero, in caso uno dei due giocatori abbia un vantaggio di 2 manche sull'altro giocatore e a patto che siano state giocate almeno quattro manche, in questo caso la partita termina istantaneamente.

Architettura generale del circuito

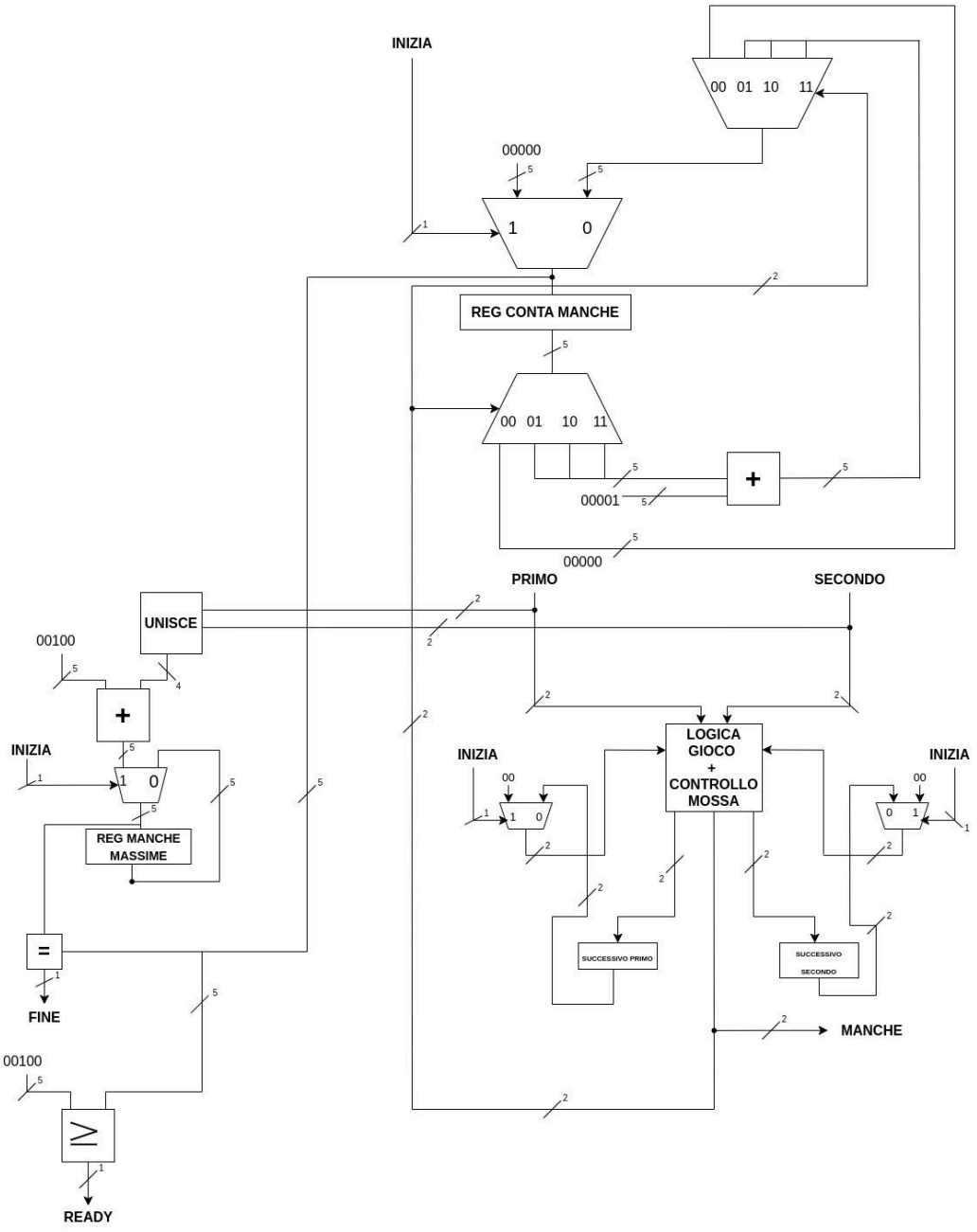


(FSMD)

Unità di elaborazione

L'unità di elaborazione è stata implementata tramite il modello DATAPATH schematizzato secondo la seguente struttura.

Datapath



Input

Input dell'unità di elaborazione:

1. **PRIMO**[2]: valore della mossa del primo giocatore; se vale **00** significa che non ha inserito "**nessuna mossa**", se vale **01** vuol dire che gioca la mossa "**sasso**", se vale **10** equivale alla mossa "**carta**", altrimenti **11** equivale alla mossa "**forbice**".
2. **SECONDO**[2]: valore della mossa del secondo giocatore; se vale **00** significa che non ha inserito "**nessuna mossa**", se vale **01** vuol dire che gioca la mossa "**sasso**", se vale **10** equivale alla mossa "**carta**", altrimenti **11** equivale alla mossa "**forbice**".
3. **VIA**[1]: valore di inizio/reset della partita, se vale 1 verrà azzerata la partita, altrimenti se vale 0 si continua a giocare la partita in corso, che viene passato dalla macchina a stati.

Output

Output dell'unità di elaborazione:

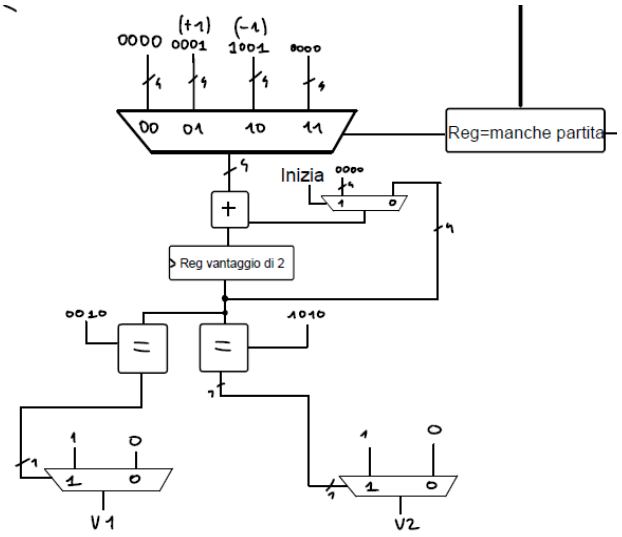
1. **MANCHE**[2]: ottenuto attraverso la logica di gioco e al controllo che l'eventuale mossa vincente non possa essere ripetuta al ciclo successivo.
   Manche viene codificata nel seguente modo:
   I. 00 = manche non valida
   II. 01 = manche vinta dal giocatore 1
   III. 10 = manche vinta dal giocatore 2
   IV. 11 = manche pareggiata
2. **READY**[2]: ottenuto attraverso il maggiore uguale tra il numero di manche giocate e le quattro manche di base. Se vale 1 vuol dire che sono state giocate almeno quattro manche e quindi se ci fosse un vantaggio di due di uno dei due giocatori, la partita potrebbe finire anticipatamente, se vale 0 vuol dire che non sono state giocate ancora quattro manche.
3. **FINE**[2]: ottenuto dal confronto tra il numero di manche giocate e il numero di manche totali. Se vale 1 vuol dire che sono state giocate tutte le manche e quindi la partita è terminata o è riiniziata, altrimenti se vale 0 la partita è ancora in corso.

Scelte progettuali

In seguito, viene analizzata nello specifico il funzionamento del DATAPATH.
Per capacità e chiarezza suddiviso in blocchi, che fanno tutti riferimento all'architettura rappresentata precedentemente, analizzando logicamente e progettualmente il significato e il compito di ogni parte dell'unità di elaborazione. La progettazione del DATAPATH nella seguente maniera è stato pensato affinché lo schema risulti funzionale, leggibile e fluido nel passaggio di informazione, così da evitare possibili ritardi e malfunzionamenti.
Inizialmente si era pensato di gestire il vantaggio attraverso il Datapath, ma poi si è riscontrato un problema con la macchina a stati, in quanto l'unità di elaborazione avrebbe svolto tutte le funzionalità necessarie affinché venissero rispettati i requisiti del progetto. Optare per questa procedura avrebbe comportato il rischio di una potenziale semplificazione della macchina a stati a un unico stato, e alla perdita di significato per quanto riguarda la finalità del progetto nel creare un FSMD.

Idea di come gestire il vantaggio con il Datapath

L'idea era infatti quella di gestire il vantaggio, attraverso la somma che dipendeva dal risultato della manche, in quanto se PRIMO[2] vinceva si andava a sommare 1 e quindi il registro vantaggio di 2, memorizzava 1, però nel caso in cui vincesse SECONDO[2], si sarebbe andati a sottrarre -1, cosa da gestire la situazione di vantaggio.
Poi attraverso degli operatori di confronto si sarebbero mandati due segnali di stato alla FSM che la informassero di chi era in vantaggio durante la partita.



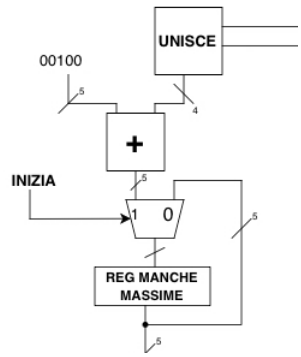
Poi tale schema tornerà utile durante la stesura della macchina a stati.

Perciò, si è deciso perciò di seguire la seconda strada ai fini della realizzazione del dispositivo.

Inoltre, durante la fase di progettazione del Datapath, sono state prese diverse decisioni per semplificare l'implementazione dell'unità di elaborazione. In particolare, la logica di gioco e il controllo delle mosse sono stati combinati in una singola tabella di verità. Abbiamo esteso questa approccio alla creazione di una tabella di verità che combina i bit corrispondenti di PRIMO[2] e SECONDO[2], consentendo così il calcolo del numero massimo di manche. Infine, è stata presa la decisione che nel caso in cui la partita terminasse senza un vantaggio di almeno due, essa sarebbe considerata un pareggio.

Analisi Datapath

Gestione del numero totale di manche

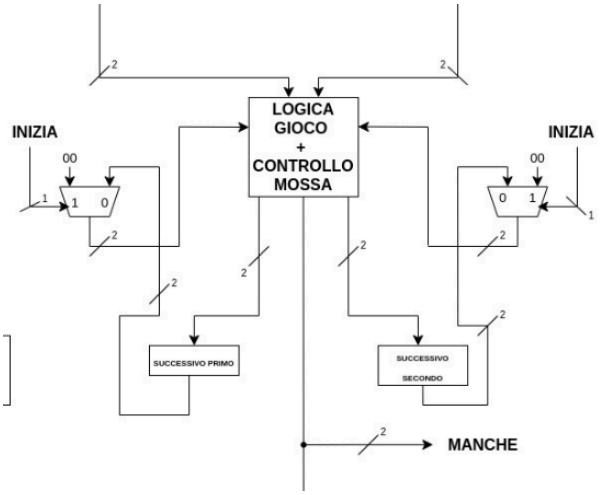


PRIMO e SECONDO ad ogni manche vengono mandati all'interno del blocco UNISCE.
Una volta uniti i bit vengono sommati a quattro perché è il numero minimo di manche da giocare.
Una volta ottenuto il numero totale di manche, il totale entra in multiplexer e se vale 1 pende il valore della somma e lo salva in un registro.
Se vale 0 e quindi per tutta la durata della partita continua a mantenere il numero di manche calcolato al primo ciclo e che viene preso dal registro sottostante.

Gestione del vincitore della manche e controllo della mossa vincente

Per stabilire il vincitore della manche abbiamo deciso di creare una tabella di verità ad otto entrate e due uscite.
Le entrate sono:

1. PRECEDENTE PRIMO[2]
2. PRECEDENTE SECONDO[2]
3. PRIMO[2]
4. SECONDO[2]
5. SUCCESSIVO PRIMO[2]
6. SUCCESSIVO SECONDO[2]

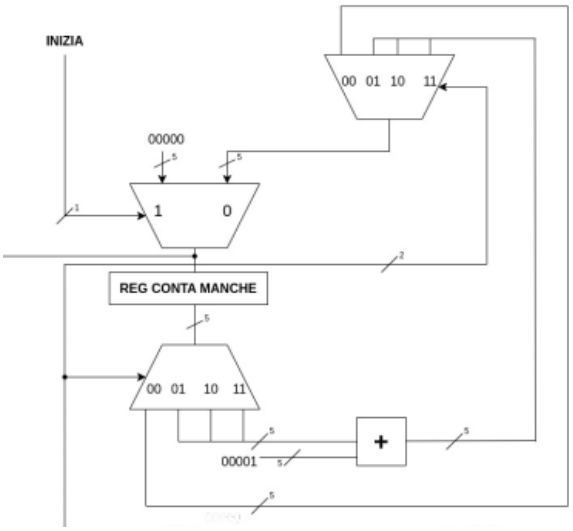


Le uscite sono:

1. MANCHE[2]
2. SUCCESSIVO PRIMO[2]
3. SUCCESSIVO SECONDO[2]

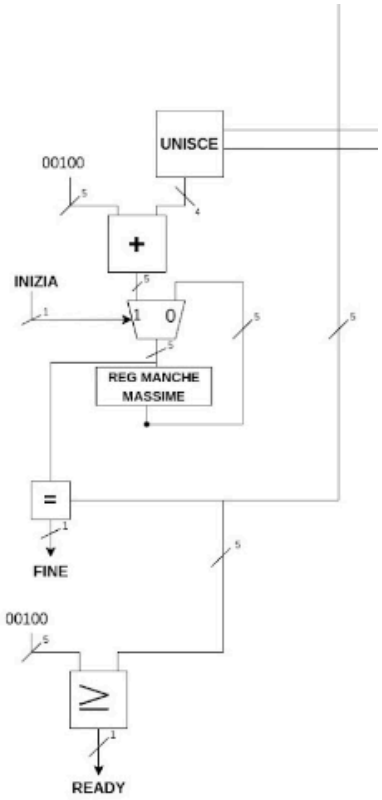
PRECEDENTE PRIMO e PRECEDENTE SECONDO vengono stabiliti dal multiplexer: se vale 1 e quindi, nel caso ci troviamo nella prima manche precedente vale 00, se invece vale 0 il multiplexer prende il valore di successivo e diventa precedente per la manche successiva.

Gestione del conteggio delle manche



Il ciclo di manche viene settato al primo ciclo: per settarlo bisogna inserire come valore di INIZIA 1.
Dalla seconda manche se INIZIA vale 0 viene passato il numero di manche conservato nel registro CONTA MANCHE.
il registro poi passa il numero delle manche al multiplexer che viene gestito da MANCHE[2] che proviene dalla logica di gioco. In caso di manche non valida il registro non viene aggiornato, mentre negli altri casi viene aggiunto +1 al numero di manche.

Gestione di FINE e READY

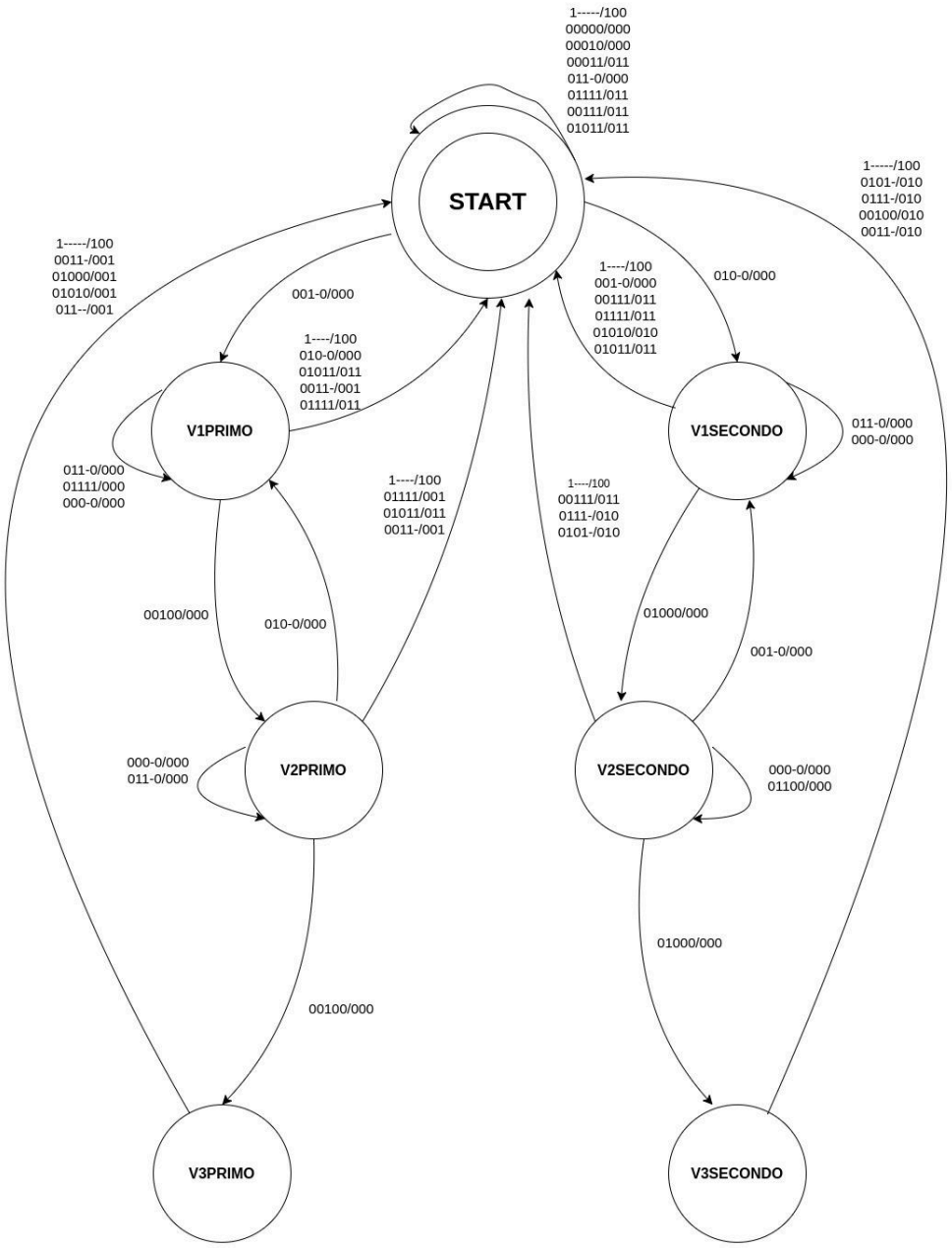


Per calcolare FINE prendiamo il filo che proviene dal multiplexer iniziale e lo compariamo al numero di manche totali che esce dal multiplexer sopra il registro che contiene il numero totale di manche.
Per calcolare READY, invece, prendiamo sempre il filo del multiplexer iniziale e se è maggiore o uguale a quattro (numero minimo) di manche.

Controllore

Il controllore è stato realizzato tramite una macchina a stati finiti; di seguito si riporta il grafico delle transizioni, con input e output corrispondenti, e una breve analisi del suo funzionamento.

State Transition Graph (STG)



Input

Input relativi al controllore:

4. **INIZIA**[1]: valore di inizio/reset della partita, se vale 1 verrà azzerata la partita, altrimenti se vale 0 si continua a giocare la partita in corso.
2. **READY**[2]: ottenuto attraverso il maggiore uguale tra il numero di manche giocate e le quattro manche di base. Se vale 1 vuol dire che sono state giocate almeno quattro manche e quindi se ci fosse un vantaggio di due di uno dei due giocatori, la partita potrebbe finire anticipatamente, se vale 0 vuol dire che non sono state giocate ancora quattro manche.
3. **FINE**[2]: ottenuto dal confronto tra il numero di manche giocate e il numero di manche totali. Se vale 1 vuol dire che sono state giocate tutte le manche e quindi la partita è terminata o è ri-iniziata, altrimenti se vale 0 la partita è ancora in corso.

Output

Output relativi al controllore:

1. **VIA**[1]: valore di inizio/reset della partita, se vale 1 verrà azzerata la partita, altrimenti se vale 0 si continua a giocare la partita in corso, che viene passata al datapath.
2. **PARTITA**[2]: risultato della partita e codificato nel seguente modo:
   I. 00 = partita non ancora terminata
   II. 01 = partita vinta dal giocatore 1
   III. 10 = partita vinta dal giocatore 2
   IV. 11 = partita pareggiata

Architettura del controllore

Il controllo è stato pensato avente i seguenti stati:

1. **START/PEREGGIO**: è lo stato iniziale della macchina, quando inizia (riinizia) una partita.

2. **V1PRIMO, V1SECONDO, V2PRIMO, V2SECONDO, V3PRIMO, V3SECONDO**: stati in cui la macchina è accesa e gestisce i diversi vantaggi dei due giocatori.

3. **PARTITA**: è lo stato finale in cui la macchina va quando si ha un vantaggio di +2 oppure sono finite tutte le manche.

FUNZIONAMENTO
Inizialmente ci troviamo nello stato START/PAREGGIO. Se **INIZIA** vale 1, resto in questo stato, e gli output sono tutti 0.
Se, invece, **MANCHE** vale 0 e a seconda del vincitore della manche, ci si sposta nello stato corrispondente: V1PRIMO se **MANCHE** vale 01, V1SECONDO se vale 01, START se vale 10, e, infine, START qualora **MANCHE** vale 00.
A secondo se si vince/pareggia o nel caso di manche non valida ci si sposta nei vari stati.

Nel caso in cui ci trovassimo in una situazione di vantaggio di +2 o +3 e con READY a 1 la partita termina e la macchina a stati fornice il vincitore della manche.
Se all'ultima manche ci si trova in un vantaggio di +1 con READY e FINE a 1 allora la FSM fornirà come risultato della partita 11.

Scelte progettuali

Si sarebbe potuto aggiungere un contatore per il numero di manche di vittorie di PRIMO e SECONDO, così poi nel caso in cui si arrivasse alla fine della partita con solo un vantaggio di +1 si sarebbe potuto stampare a video chi ha vinto più manche.
Ma per comodità abbiamo deciso, invece, di stampare a video come risultato partita 11.

Realizzazione del circuito in SIS

Per poter realizzare il circuito abbiamo creato la nostra FSM, descritta tramite la transizione degli stati scritti nel file blif. Una volta realizzata, attraverso *state_minimize stamina* abbiamo ottenuto la fsm equivalente minima, formata da 7 stati: *START*, *V1PRIMO*, V1SECONDO, V2PRIMO, V2SECONDO, V3PRIMO, V3SECONDO.
A questo punto attraverso il comando *state_assign*, *sis* ha assegnato una codifica agli stati e ha mappato il controllore trasformandolo in circuito sequenziale.

```
loris@loris-VirtualBox:~$ cd Scrivania/
loris@loris-VirtualBox:~/Scrivania$ cd VR500402_VR500356/
loris@loris-VirtualBox:~/Scrivania/VR500402_VR500356$ quit
Comando «quit» non trovato, ma può essere installato con:
sudo snap install quit
loris@loris-VirtualBox:~/Scrivania/VR500402_VR500356$ read_blif FSM[1].blif
read_blif: comando non trovato
loris@loris-VirtualBox:~/Scrivania/VR500402_VR500356$ sis
UC Berkeley, SIS 1.3.6 (compiled 2017-10-27 16:08:57)
sis> read_blif FSM[1].blif
sis> state_minimize stamina
Running stamina, written by June Rho, University of Colorado at Boulder
Number of states in original machine : 7
Number of states in minimized machine : 7
sis> state_assign jedi
Running jedi, written by Bill Lin,  UC Berkeley
sis> stg_to_network
sis> write_blif fsm.blif
sis>
```

Statistiche fsm

```
loris@loris-VirtualBox:~$ cd Scrivania/
loris@loris-VirtualBox:~/Scrivania$ cd VR500402_VR500356/
loris@loris-VirtualBox:~/Scrivania/VR500402_VR500356$ quit
Comando «quit» non trovato, ma può essere installato con:
sudo snap install quit
loris@loris-VirtualBox:~/Scrivania/VR500402_VR500356$ read_blif FSM[1].blif
read_blif: comando non trovato
loris@loris-VirtualBox:~/Scrivania/VR500402_VR500356$ sis
UC Berkeley, SIS 1.3.6 (compiled 2017-10-27 16:08:57)
sis> read_blif FSM[1].blif
sis> state_minimize stamina
Running stamina, written by June Rho, University of Colorado at Boulder
Number of states in original machine : 7
Number of states in minimized machine : 7
sis> state_assign jedi
Running jedi, written by Bill Lin,  UC Berkeley
sis> stg_to_network
sis> write_blif fsm.blif
sis> print_stats
FSM             pi= 5   po= 3   nodes=  6       latches= 3
lits(sop)= 147  #states(STG)=   7
```

Statistiche circuito

```
UC Berkeley, SIS 1.3.6 (compiled 2017-10-27 16:08:57)
sis> read_blif FSMD.blif
Warning: network `DATAPATH', node "m2" does not fanout
Warning: network `DATAPATH', node "m1" does not fanout
Warning: network `DATAPATH', node "m0" does not fanout
Warning: network `FSMD', node "m2" does not fanout
Warning: network `FSMD', node "m1" does not fanout
Warning: network `FSMD', node "m0" does not fanout
sis> print_stats
FSMD            pi= 5   po= 4   nodes=118       latches=17
lits(sop)=10561
sis> full_simplify
sis> print_stats
FSMD            pi= 5   po= 4   nodes=118       latches=17
lits(sop)=1042
sis> source script.rugged
sis> print_stats
FSMD            pi= 5   po= 4   nodes= 60       latches=17
lits(sop)= 516
```

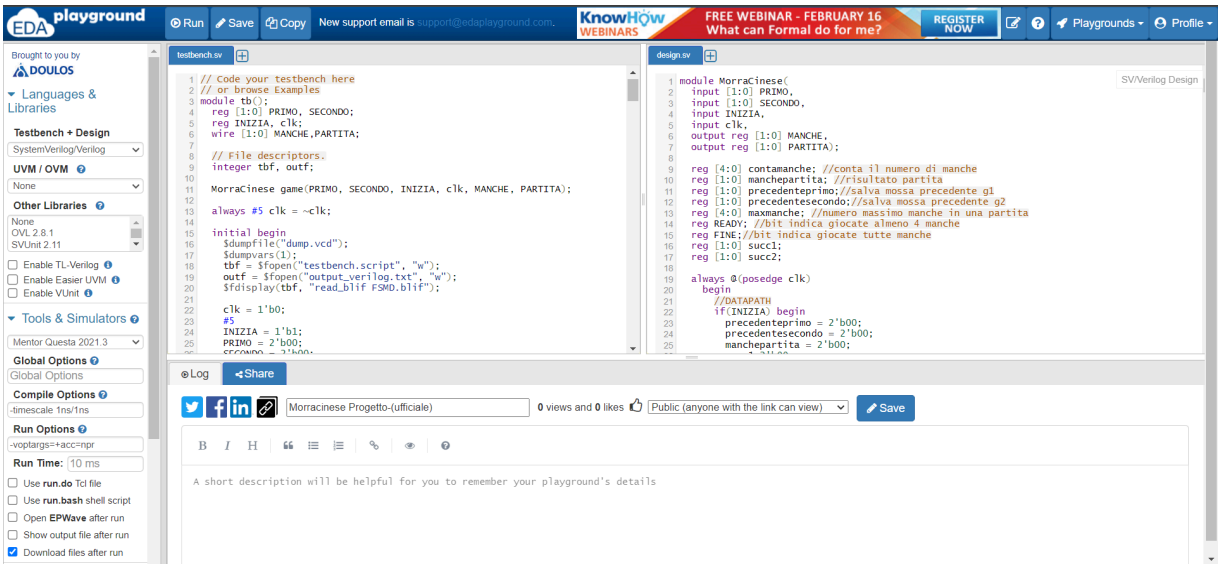
Statistiche finali elaborato

```
sis> read_library synch.genlib
sis> map -m 0 -s
warning: unknown latch type at node '{[23]}' (RISING_EDGE assumed)
warning: unknown latch type at node '{[24]}' (RISING_EDGE assumed)
warning: unknown latch type at node '{[25]}' (RISING_EDGE assumed)
WARNING: uses as primary input drive the value (0.20,0.20)
WARNING: uses as primary input arrival the value (0.00,0.00)
WARNING: uses as primary input max load limit the value (999.00)
WARNING: uses as primary output required the value (0.00,0.00)
WARNING: uses as primary output load the value 1.00
>>> before removing serial inverters <<<
# of outputs:          21
total gate area:       9720.00
maximum arrival time: (78.80,78.80)
maximum po slack:     (-5.00,-5.00)
minimum po slack:     (-78.80,-78.80)
total neg slack:      (-855.60,-855.60)
# of failing outputs:  21
>>> before removing parallel inverters <<<
# of outputs:          21
total gate area:       9496.00
maximum arrival time: (77.20,77.20)
maximum po slack:     (-5.00,-5.00)
minimum po slack:     (-77.20,-77.20)
total neg slack:      (-851.20,-851.20)
# of failing outputs:  21
# of outputs:          21
total gate area:       8408.00
maximum arrival time: (73.20,73.20)
maximum po slack:     (-5.00,-5.00)
minimum po slack:     (-73.20,-73.20)
total neg slack:      (-793.00,-793.00)
# of failing outputs:  21
sis>
```

Realizzazione del circuito in Verilog

Il circuito in Verilog è stato implementato adottando un approccio comportamentale. Abbiamo seguito la struttura del DATAPATH, implementando il blocco Logica di Gioco+Controllo Mossa attraverso un costrutto "case". Le condizioni "if..else" sono state utilizzate per gestire il conteggio delle manche e i segnali FINE e READY.

Successivamente, la FSM è stata realizzata tramite istruzioni "case" semplici. La corretta esecuzione del circuito è stata facilitata dall'implementazione di un testbench, che è stato cruciale per simulare ed eseguire i file. Questo ci ha permesso di identificare eventuali errori e correggerli, garantendo la robustezza e l'efficienza del nostro design.



Considerazioni finali

In generale, ciò che ci ha guidati nella realizzazione del nostro elaborato, è stato, oltre al garantire che il tutto funzionasse in ogni caso possibile e presentabile, la volontà di ottenere una macchina più efficiente ed ottimizzata possibile, ma allo stesso tempo anche flessibile e facilmente modificabile.
Si tratta infatti di una struttura aperta alle modifiche, che richiede poco impegno di risorse affinché vengano reindirizzati e cambiati segnali che permettono di variare il funzionamento della macchina, perciò la possiamo definire versatile e migliorabile in futuro in caso di aggiornamenti dei requisiti.
Inoltre, abbiamo affrontato questo progetto universitario come una sfida stimolante, che ci ha avvicinato molto alla teoria vista a lezione e ci ha permesso di comprenderla meglio e di metterci nei panni di essere dei progettisti.
Grazie a questo abbiamo acquisito competenze fondamentali nel campo della progettazione di circuiti digitali. L'esperienza è stata preziosa nel fornirci una visione approfondita delle metodologie di progettazione e nell'implementazione pratica dei concetti teorici appresi durante il corso.

L'utilizzo di Verilog come linguaggio di descrizione hardware ci ha permesso di tradurre in modo efficiente i nostri concetti di progettazione in implementazioni concrete.

In conclusione, questo progetto ha rappresentato un'opportunità formidabile e siamo grati per l'esperienza e crediamo che le conoscenze acquisite saranno di grande valore e utili nella nostra carriera scolastica e lavorativa futura.