UNIVERSITÀ
di VERONA

Elaborato SIS

Architettura degli Elaboratori
A.A. 2023/2024 - Corso di Laurea in Informatica

Giuseppe Caltroni
VR489402

Diego Arroyo
VR500824

Indice

1 Introduzione

L'obiettivo di questo elaborato è la progettazione e l'implementazione di un dispositivo digitale in grado di gestire partite del gioco della morra cinese, conosciuto anche come "sasso-carta-forbici", attraverso l'uso di SIS e Verilog.

Il gioco della morra cinese ha una logica ben definita. Due giocatori scelgono simultaneamente una delle tre possibili mosse:

- Sasso, che batte forbici,
- Forbici, che battono carta,
- Carta, che batte sasso.

Se entrambi i giocatori scelgono la stessa mossa, la manche termina in pareggio. Il progetto prevede che una partita si svolga su un minimo di quattro manche e un massimo di diciannove, con una serie di regole che rendono il gioco più avvincente, come l'obbligo per il vincitore della manche precedente di non ripetere la stessa mossa.

Il progetto è stato sviluppato implementando un circuito digitale per gestire le partite e le manche della morra cinese. Inizialmente, abbiamo utilizzato Logisim per creare una rappresentazione fisica del circuito, con l'obiettivo di visualizzare e comprendere meglio la logica di funzionamento del sistema. Questa fase preliminare ci ha permesso di testare le basi del progetto e di avere un punto di riferimento chiaro per le fasi successive.

Una volta completata questa fase, abbiamo trasposto il progetto in SIS, un software di sintesi logica, dove ci siamo concentrati sull'ottimizzazione del circuito in termini di area e ritardo. In SIS, abbiamo eseguito una prima versione non ottimizzata del circuito, che successivamente è stata migliorata per ottenere prestazioni migliori, come richiesto dalle specifiche.

Infine, il progetto è stato implementato in Verilog per la descrizione completa del comportamento del circuito a livello di codice hardware.

Lo scopo principale dell'elaborato è dimostrare l'equivalenza funzionale tra le due versioni, validando la correttezza del progetto tramite simulazioni e confronti. In particolare, viene generato un file di output dal modello Verilog che serve come riferimento per il modello SIS. Il confronto tra i risultati dei due modelli garantisce la coerenza e correttezza della soluzione.

2 Architettura Generale del Circuito

2.1 Schema FSMD (Finite State Machine with Datapath)

L'architettura generale del circuito per la gestione delle partite di morra cinese si basa su un modello FSMD (Finite State Machine with Datapath). Questo schema è composto da due componenti principali: una Macchina a Stati Finiti (FSM) e un Datapath, che collaborano per gestire la logica di controllo e l'elaborazione dei dati necessari al funzionamento del dispositivo.

2.2 Descrizione della FSM

La FSM rappresenta il controllore logico del sistema, ed è responsabile della gestione degli stati del gioco. In questo progetto, la FSM si occupa di gestire:

- L'inizio della partita e la transizione tra le manche,
- La determinazione del vincitore di ciascuna manche in base alle scelte dei giocatori,
- Il monitoraggio delle condizioni di fine partita, ossia quando uno dei giocatori raggiunge due vittorie in più rispetto all'avversario dopo aver giocato almeno quattro manche,
- Il vincolo secondo cui il vincitore di una manche non può ripetere la stessa mossa nella manche successiva, invalidando la manche nel caso ciò accada.

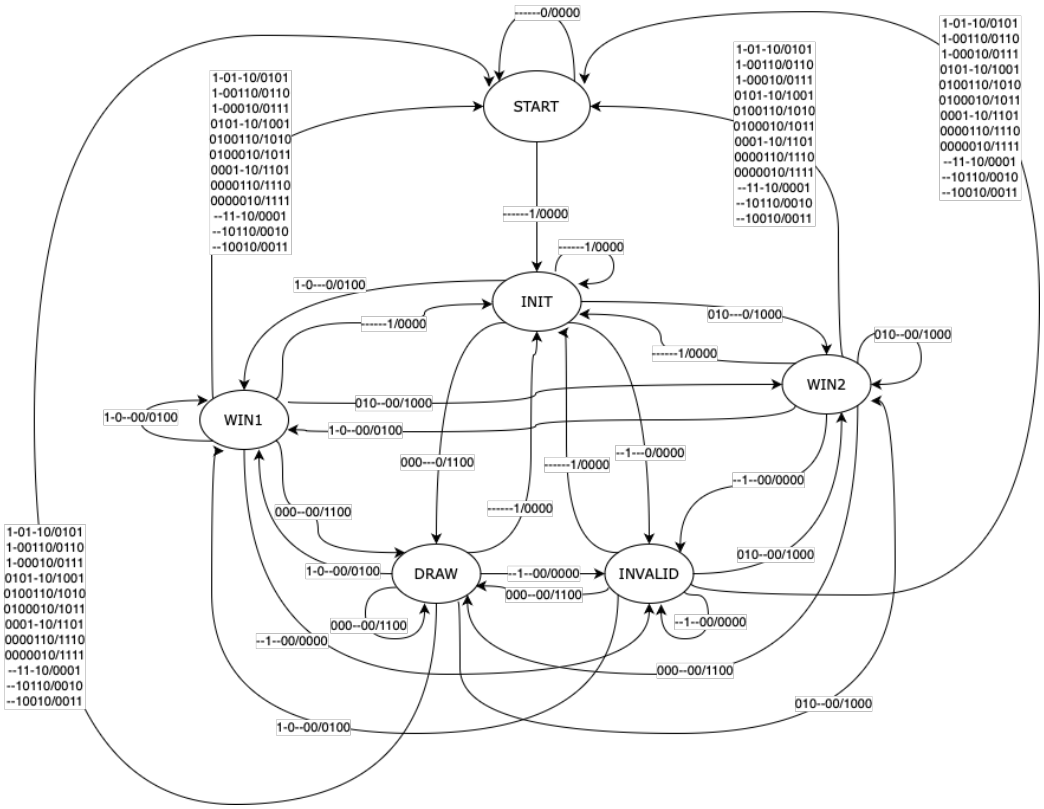


Figura 1: Grafo di transizione dello stato della FSM(Mealy)

La FSM si aggiorna ad ogni ciclo di clock e gestisce le transizioni tra stati in base alle condizioni della partita e del segnale di START.

Gli stati della FSM sono:

- START (000): resetta i valori iniziali.
- INIT (001): inizializza la partita se il segnale di start è vero.
- WIN1 (010): indica che P1 ha vinto la manche.
- WIN2 (011): indica che P2 ha vinto la manche.
- DRAW (100): indica che la manche è finita in pareggio.
- INVALID (101): indica una manche non valida.

2.3 Architettura del Datapath

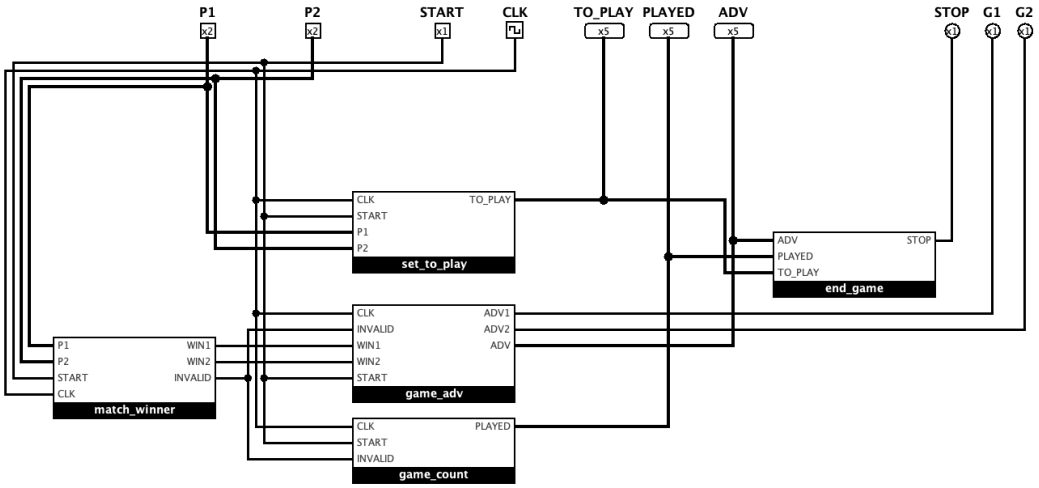


Figura 2: Struttura del Datapath rappresentato con Logisim.

Il datapath è composto da diversi moduli principali, ognuno dei quali svolge un ruolo specifico nell'elaborazione del gioco:

match_winner: Questo modulo permette, controllando se START è alzato oppure no, il vincitore della manche attuale attraverso una PLA ed un sistema di MUX per verificarne la validità della mossa.

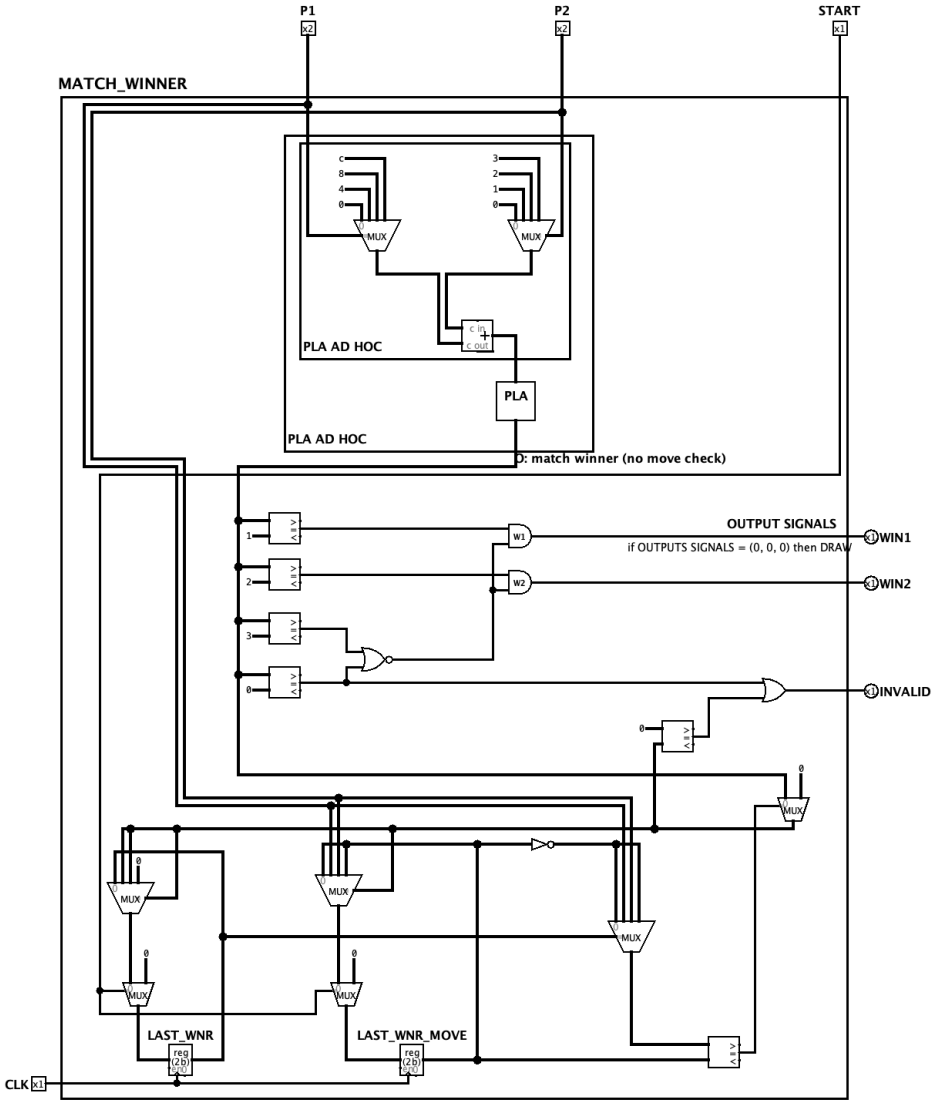
game_adv: Il modulo `game_adv` tiene traccia del vantaggio accumulato da uno dei due giocatori.

set_to_play: il numero massimo di round, che verrà sommato al numero minimo di round (4 round), viene creato concatenando il valore dei due input nel caso che START = 1.

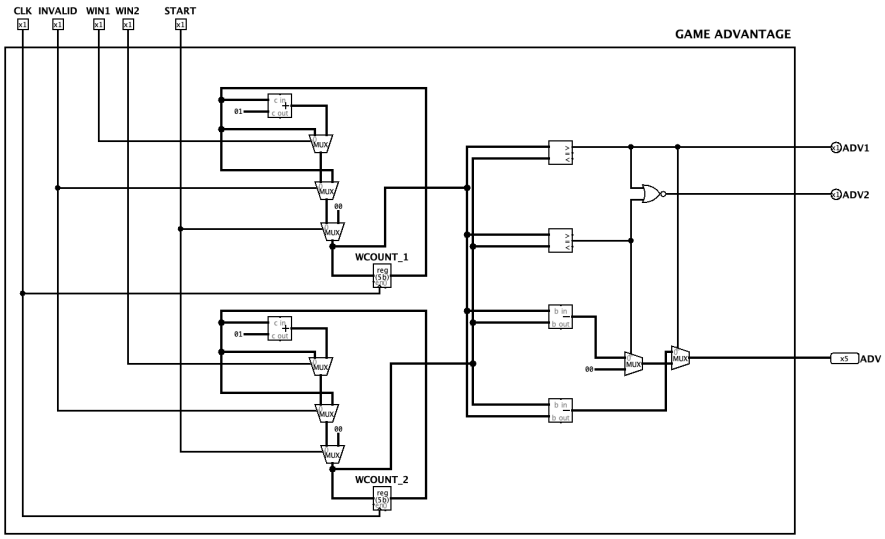
game_count: Un altro componente fondamentale è il modulo `game_count`, responsabile del conteggio delle manche giocate.

end_game: Questo modulo è un controllore che decide quando la partita deve terminare. Prende in input il numero di partite giocate (PLAYED), il numero massimo di partite da giocare (TO_PLAY), e il vantaggio accumulato da un giocatore (ADV). Il modulo confronta il numero di partite giocate con il massimo consentito e verifica se uno dei giocatori ha un vantaggio sufficiente per concludere la partita. Se una delle condizioni di fine partita è soddisfatta (tutte le partite sono giocate o un giocatore ha accumulato un vantaggio decisivo), il modulo attiva un segnale di uscita che indica la fine

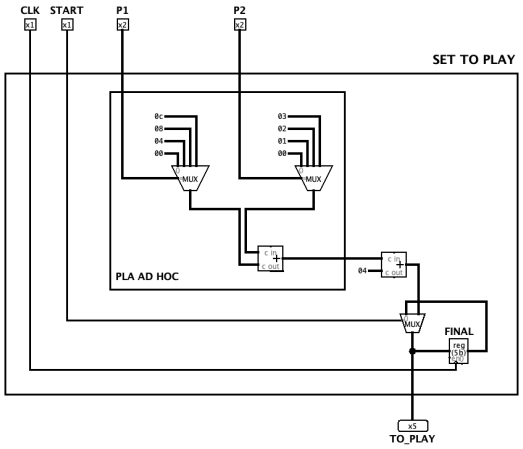
della partita (STOP) e identifica il vincitore.



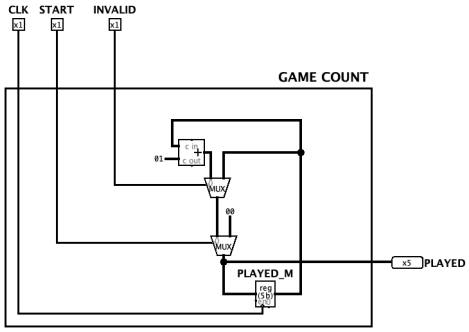
(a) match_winner



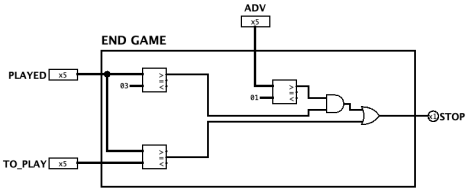
(a) game_adv



(b) set_to_play



(c) game_count



(d) end_game

Figura 3: I moduli del datapath rappresentati con Logisim.

3 Implementazione in Verilog

3.1 Input e Output

Input:

- clk: segnale di clock.
- P1 e P2: mosse dei giocatori (2 bit per ciascuno).
- START: segnale di avvio per iniziare la partita.

Output:

- ROUND: risultato della manche (00 = non valida, 01 = vinta da P1, 10 = vinta da P2, 11 = pareggio).
- GAME: stato della partita (00 = non finita, 01 = vinta da P1, 10 = vinta da P2, 11 = pareggio).

```
1     module MorraCinese(
2
3         // Inputs
4         input clk,
5         input[1:0] P1,
6         input[1:0] P2,
7         input START,
8
9         // Outputs
10        output reg[1:0] ROUND = 2'b00,
11        output reg[1:0] GAME = 2'b00
12    );
```

3.2 Registri e variabili

- CURRENT_STATE e NEXTT_STATE: gestiscono gli stati della FSM (Finite State Machine) della partita.
- ROUNDT_WINNER e PREV_ROUND_WINNER: tengono traccia dei vincitori delle manche.
- PLAYED e TO_PLAY: tengono traccia delle manche giocate e quelle rimanenti.

- ADV: variabile usata per determinare l'andamento del gioco e calcolare il vincitore finale.

```
    // Registers and variables
    reg[2:0] CURRENT_STATE = 3'b000;
    reg[2:0] NEXT_STATE;

    reg[1:0] CURRENT_ROUND_WINNER = 2'b00;
    reg[1:0] PREV_ROUND_WINNER = 2'b00;
    reg[1:0] PREV_WINNING_MOVE = 2'b00;

    reg[4:0] PLAYED = 5'b00000;
    reg[4:0] TO_PLAY = 5'b00000;

    reg[3:0] ADV = 4'b0100;

```

3.3 FSM

La FSM del modulo gestisce il flusso della partita tramite stati. I principali stati sono:

- **START (000)**: resetta i valori iniziali.
- **INIT (001)**: inizializza la partita se il segnale di start è vero.
- **WIN1 (010)**: indica che P1 ha vinto la manche.
- **WIN2 (011)**: indica che P2 ha vinto la manche.
- **DRAW (100)**: indica che la manche è finita in pareggio.
- **INVALID (101)**: indica una manche non valida.

Aggiornamento FSM:
La FSM si aggiorna ad ogni ciclo di clock e gestisce le transizioni tra stati in base alle condizioni della partita e del segnale di START.

```
    always @(posedge clk) begin : FSM
```

Caso 1: Quando la FSM, trovandosi nello stato iniziale di partenza, riceve come input START=1, lo stato prossimo viene aggiornato a INIT, e gli output indicano che nessun giocatore ha vinto né la partita né la manche.

```
    always @(posedge clk) begin : FSM

        /*        Case 1:
                Start signal evaluates True, no matter the
                ↪  state, the FSM
                transits to INIT state.
        */
                if(START) begin
                        NEXT_STATE = 3'b001;
                        ROUND = 2'b00;
                        GAME = 2'b00;

```

Caso 2: Quando la FSM, trovandosi nello stato iniziale di partenza, riceve come input START=0, indipendentemente dagli input delle mosse dei 2 giocatori, allora lo stato prossimo viene aggiornato sempre allo stato iniziale, sempre con output entrambi nulli. In questo modo si aspetta START=1 come input per iniziare la partita.

```
            /*        Case 2:
                Start signal evaluates False, if current state
                ↪  is reset state
                then FSM slef-transits to reset state.
        */
                end else if(CURRENT_STATE == 3'b000) begin
                        NEXT_STATE = 3'b000;
                        ROUND = 2'b00;
                        GAME = 2'b00;

```

Caso 3: se non ci troviamo nello stato iniziale e START è 0, allora viene assegnato all'output ROUND il valore del valore della manche. Se la partita non è ancora finita (ossia se nessuno dei due giocatori ha un vantaggio di

almeno 2 round, e se mancano partite da giocare, oppure se le partite giocate sono meno di 4) (righe 161-162), allora in base al valore di ROUND viene aggiornato lo stato prossimo.
(Righe 161->170)
Più in particolare, se: -ROUND = 00 (manche non valida) allora lo stato prossimo si aggiorna INVALID -ROUND = 01 (manche vinta da P1) allora lo stato prossimo si aggiorna a WIN1 -ROUND = 10 (manche vinta da P2) allora lo stato prossimo si aggiorna a WIN2 -ROUND = 11 (manche pareggiata) allora lo stato prossimo si aggiorna a DRAW
(righe 173->183)
Se la partita invece è finita, ossia se uno dei due giocatori ha un vantaggio di 2 sull'altro avendo giocato almeno 4 round, o se è stato raggiunto il numero massimo di round, allora se P1 è in vantaggio viene aggiornato GAME a W1 (ossia che vince P1), mentre se p2 è in vantaggio viene aggioranto GAME a W2 (vittoria di P2). Nel caso che non ci sia un vantaggio allora la partita viene terminata con un pareggio (GAME = 11). Una volta terminata la partita lo stato prossimo viene aggiornato a START.

```
1           /* Case 3:
2                   Start signal evaluates False, if current state
                    ↪  is any other state
3                   but reset state, FSM transits to next state
                    ↪  based on the round outcome
4           */
5                   end else begin
6
7                           ROUND = CURRENT_ROUND_WINNER;
8
9                           // If game not done yet
10                          if(((ADV > 4'b0010 && ADV < 4'b0110) &&
                            ↪  PLAYED < TO_PLAY) || PLAYED < 4)
                            ↪  begin
11                              GAME = 2'b00;
12
13                              // Case to determins the Next
                                ↪  State
14                              case (ROUND)
15                                      2'b00: NEXT_STATE =
                                        ↪  3'b101;
16                                      2'b01: NEXT_STATE =
                                        ↪  3'b010;
```

```
17                                          2'b10: NEXT_STATE =
                                            ↪  3'b011;
18                                          2'b11: NEXT_STATE =
                                            ↪  3'b100;
19                                  endcase
20
21                          // If game is done
22                          end else begin
23                                  // Game winner:
24                                  if(ADV > 4'b0100)
                                    ↪  begin                                    //
                                    ↪  Game won by P1
25                                      GAME = 2'b01;
26                                  end else if(ADV < 4'b0100)
                                    ↪  begin        // Game won by
                                    ↪  P2
27                                      GAME = 2'b10;
28                                  end else if(ADV == 4'b0100)
                                    ↪  begin        // Game draws
29                                      GAME = 2'b11;
30                                  end
31                                  NEXT_STATE = 3'b000;
32                          end
```

3.4 Aggiornamento della FSM

Ad ogni ciclo di clock viene aggiornato lo stato della FSM.

```
1       always @(posedge clk) begin : UPDATE_STATE
2                   CURRENT_STATE = NEXT_STATE;
3           end
```

3.5 Datapath

Reset e inizializzazione dei valori (righe 49-54) Vengono riportati i valori del sistema alla configurazione iniziale:

- Il vincitore della manche corrente (CURRENT_ROUND_WINNER), il vincitore precedente (PREV_ROUND_WINNER) e la mossa vincente precedente (PREV_WINNING_MOVE) vengono resettati a 00.
- PLAYED viene resettato, indicando che non sono ancora state giocate manche.
- ADV, che tiene traccia del vantaggio accumulato da uno dei due giocatori, viene inizializzato a 4.

```
1       if(START || CURRENT_STATE == 3'b000) begin
2                               CURRENT_ROUND_WINNER = 2'b00;
3                               PREV_ROUND_WINNER = 2'b00;
4                               PREV_WINNING_MOVE = 2'b00;
5                               PLAYED = 5'b00000;
6                               ADV = 4'b0100;
7                               if(START) begin
8                                       TO_PLAY = TO_PLAY + {P1, P2};
9                               end else begin
10                                      TO_PLAY = 5'b00100;
11                              end
```

Impostazione del numero di manche (righe 55-59) Se il segnale START è attivo, il numero di manche da giocare viene calcolato sommando la concatenazione dei bit di P1 e P2 al valore predefinito. Se il segnale START non è attivo, e ci troviamo nello stato inizaile, viene impostato il numero minimo di manche da giocare (4).

```
1       if(START) begin
2                       TO_PLAY = TO_PLAY + {P1, P2};
3           end else begin
4                       TO_PLAY = 5'b00100;
5           end
```

Condizioni di validità per giocare una manche (riga 70) verifica se le condizioni sono valide per giocare una manche. Le condizioni includono:

- ADV (il vantaggio di uno dei giocatori) deve essere tra 2 e 6.
- Il numero di manche giocate deve essere inferiore a quelle ancora da giocare o inferiore a 4 (il numero minimo).

```
1                    end else if(((ADV > 4'b0010 && ADV <
                     ↪  4'b0110) && PLAYED < TO_PLAY) || PLAYED
                     ↪  < 4) begin
```

Esito manche non valida (righe 78-79) Se il vincitore della manche precedente ripete la stessa mossa, o se uno dei giocatori non ha inserito una mossa valida (P1 == 00 o P2 == 00), la manche è considerata non valida e il vincitore corrente è impostato a 00.

```
1      if({PREV_ROUND_WINNER, PREV_WINNING_MOVE} == {2'b01, P1} ||
       ↪  {PREV_ROUND_WINNER, PREV_WINNING_MOVE} == {2'b10, P2}
       ↪  || P1 == 2'b00 || P2 == 2'b00) begin
2                    CURRENT_ROUND_WINNER = 2'b00;
```

Esito manche valida (righe 82-106) Viene controllato il risultato della manche confrontando le mosse di P1 e P2. A seconda della combinazione:

- Se P1 vince, i registri del vincitore vengono aggiornati e il vantaggio di P1 (ADV) viene incrementato.
- Se P2 vince, i registri vengono aggiornati e il vantaggio di P2 viene decrementato.
- In caso di pareggio, CURRENT_ROUND_WINNER viene impostato a 11 e i registri relativi alla mossa vincente precedente vengono azzerati.

Infine, il numero di manche giocate (PLAYED) viene incrementato.

```
1                end else begin
2                        case({P1, P2})
3                             // Wins P1
4                             4'b0111, 4'b1001, 4'b1110: begin
5                                       CURRENT_ROUND_WINNER =
                                        ↪  2'b01;
6                                       PREV_ROUND_WINNER =
                                        ↪  2'b01;
7                                       PREV_WINNING_MOVE = P1;
8                                       ADV = ADV + 1;
9                                end
10                               // Wins P2
11                               4'b1101, 4'b0110, 4'b1011:
                                 ↪  begin
```

```
12                                      CURRENT_ROUND_WINNER =
                                        ↪  2'b10;
13                                      PREV_ROUND_WINNER =
                                        ↪  2'b10;
14                                      PREV_WINNING_MOVE = P2;
15                                      ADV = ADV - 1;
16                                  end
17                                  // Draw
18                                  4'b0101, 4'b1010, 4'b1111:
                                    ↪  begin
19                                      CURRENT_ROUND_WINNER =
                                        ↪  2'b11;
20                                      PREV_ROUND_WINNER =
                                        ↪  2'b00;
21                                      PREV_WINNING_MOVE =
                                        ↪  2'b00;
22                                      end
23                          endcase
24                          PLAYED = PLAYED + 1;
25                  end
```

4 Implementazione in SIS

4.1 Input e Output

Input :

- clk: segnale di clock.
- P1 e P2: mosse dei giocatori (2 bit per ciascuno).
- START: segnale di avvio per iniziare la partita.

Output :

- ROUND: risultato della manche (00 = non valida, 01 = vinta da P1, 10 = vinta da P2, 11 = pareggio).
- GAME: stato della partita (00 = non finita, 01 = vinta da P1, 10 = vinta da P2, 11 = pareggio).

4.2 Moduli che compongono il circuito SIS

4.2.1 Modulo principale

All'interno del modulo **fsmd.blif** troviamo tutte le componenti del circuito SIS descritto di seguito.

4.2.2 FSM

- **fsm.blif**

4.2.3 Datapath

- datapath.blif

4.2.4 Componenti base

- LOGICA:
  - and.blif
  - or.blif

  - nor.blif
  - not1.blif
  - not2.blif
  - not5.blif
  - xnor.blif
  - xor.blif
- ARITMETICA:
  - adder1.blif
  - adder5.blif
  - eql2.blif
  - eql5.blif
  - grt2.blif
  - grt5.blif
  - subtractor5.blif
- MEMORIE:
  - reg1.blif
  - reg2.blif
  - reg5.blif
- PLA:
  - pla.blif (PLA usata per decidere il vincitore della manche)
- COSTANTI:
  - const_0.blif
  - const_0.blif
- MULTIPLEXER:
  - mux1_2.blif (MUX con ingressi a 2 bit e selettore a 1 bit)
  - mux1_5.blif (MUX con ingressi a 5 bit e selettore a 1 bit)
  - mux2_1.blif (MUX con ingressi a 1 bit e selettore a 2 bit)
  - mux2_2.blif (MUX con ingressi a 2 bit e selettore a 2 bit)

4.2.5 Altri moduli

- **end_game**: Questo modulo decreta, in base ai segnali in entrata, le condizioni per terminare la partita.
- **game_count.blif**: Questo modulo si occupa del conteggio delle manche giocate.
- **game_adv.blif**: Questo modulo tiene traccia del vantaggio che hanno i giocatori l'uno in confronto dell'altro. All'interno infatti c'è un registro il cui valore oscillo intorno a un valore predefinito.
- **match_winner.blif**: Questo modulo dichiara il vincitore della manche effettuando gli appositi controlli.
- **set_to_play.blif**: Questo modulo stabilisce quante aggiuntive manche giocare oltre le 4 obbligatorie.

5 Ottimizzazione

Nel processo di ottmizzazione abbiamo deciso di monitorare le statistiche del circuito sia prima che dopo la procedura, lo script impiegato è lo **script.rugged** ripetuto per mille volte. Questo approccio è stato scelto perchè, essendo lo script in grado di generare ottimizzazione buone ma non ottime, è molto probabile che sia necessario applicare più volte gli algoritmi di ritrutturazione e ottimizzazione presenti nello script, con l'obbiettivo di raggiungere il minimo locale nel miglior intorno di condizione

5.1 Statistiche pre-ottimizzazione FSMD

Prima di essere ottimizzato tramite lo script.rugged il circuito presenta le seguenti caratteristiche:

```
sis> ps
fsmd           pi= 5 po= 4 nodes=278 latches=26
lits(sop)=1167 lits(fac)=1094
```

5.2 Minimizzazione del circuito

Al circuito viene applicato uno script.rugged che esegue su tale circuito una minimizzazione multi-livello, applicando diverse tecniche di minimizzazione.

```
sis> source script.ruggedx   (sono x1000 script.rugged)
```

5.3 Statistiche post-ottimizzazione

```
fsmd           pi= 5 po= 4 nodes= 66 latches=26
lits(sop)= 404 lits(fac)= 346
```

5.4 Mapping per area

Abbiamo mappato il circuito usando la libreria tecnologica synch.genlib e abbiamo usato i parametri -m 0 -s per ottimizzare la mappatura per area.

```
sis> read_library synch.genlib
sis> map -m 0 -s
warning: unknown latch type at node '{[15]}' (RISING_EDGE assumed)
warning: unknown latch type at node '{[16]}' (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:          30
total gate area:       8320.00
maximum arrival time: (54.80,54.80)
maximum po slack:     (-7.00,-7.00)
minimum po slack:     (-54.80,-54.80)
total neg slack:      (-727.00,-727.00)
# of failing outputs:  30
>>> before removing parallel inverters <<<
# of outputs:          30
total gate area:       8208.00
maximum arrival time: (54.80,54.80)
maximum po slack:     (-7.00,-7.00)
minimum po slack:     (-54.80,-54.80)
total neg slack:      (-727.00,-727.00)
# of failing outputs:  30
# of outputs:          30
total gate area:       7872.00
```

```
maximum arrival time: (54.60,54.60)
maximum po slack:     (-7.00,-7.00)
minimum po slack:     (-54.60,-54.60)
total neg slack:      (-721.00,-721.00)
# of failing outputs:  30
sis>
```

6 Scelte Progettuali

1. Abbiamo deciso di utilizzare 5 bit per rappresentare il numero di partite massime, nonostante il numero massimo rappresentabile sia notevolmente superiore. Questa decisione garantisce semplicità ed immediatezza ad un costo irrisorio (1 bit in più) rispetto a soluzioni che avrebbero potuto prevedere 4 bit, ma con una minor intuitività e chiarezza.

2. Abbiamo deciso di salvare la mossa del giocatore vincente e mantenerla come discriminante qualora venisse ripetuta dallo stesso giocatore, anche nel caso in cui un round sia annullato. Questo garantisce maggior correttezza, impedendo al vincitore della manche precedente di annullare di proposito il round per ottenere, durante il turno successivo, una scelta di mosse non sottoposta a restrizioni.

3. Una volta terminato il gioco in maniera regolare (per vittoria su vantaggio o numero di mani giocate), abbiamo stabilito di far transitare la FSM allo stato di reset START e, da lì, mantenerlo fino a quando non venga alzato nuovamente il segnale di START a 1. Questo perché il segnale di START è l'unico segnale in grado di far ricominciare una nuova partita e, al contempo, l'unico a poter resettare la partita corrente per ricominciare immediatamente una nuova.

4. Conseguentemente al punto 3, abbiamo dedotto e preferito che, contemporaneamente alla salita del segnale START, le mosse dei giocatori, concatenate tra di loro, assumessero il valore dei round aggiuntivi da giocare.