Relazione Progetto di Reti Logiche

Prof. Gianluca Palermo - Anno accademico 2023 - 2024

Vitali Matteo (Cod. Persona: 10800443 - Matricola: 981804)

Contents

1 Introduzione

1.1 Scopo del progetto

Sia data una memoria indirizzata al byte (con indirizzi a 16 bit) che contiene, a partire dall'indirizzo `i_add`, la sequenza (anche vuota) così definita:

$$w_0,\ \gamma_0,\ w_1,\ \gamma_1,\ \ldots\ w_{k-1},\ \gamma_{k-1}$$

dove $0 < k < 512$, le parole w_i sono *unsigned* tali che $w_i < 255 \quad \forall i \leq k$ e γ_i è la credibilità, inizialmente nulla, associata alla parola w_i, $0 \leq \gamma_i \leq 31 \quad \forall i \leq k$.

L'obbiettivo del modulo da progettare è quello di aggiornare i valori di credibilità (e talvolta le parole) in accordo alla seguente specifica:

$$\gamma_i = 31 \iff w_i \neq 0$$

$$w_i = w_{i-1} \wedge \gamma_i = \max\{0, \gamma_{i-1} - 1\} \iff w_i = 0 \wedge w_{i-1} \neq 0 \quad (i \geq 1)$$

$$w_i = 0 \wedge \gamma_i = 0 \iff w_0 = 0 \wedge \forall k \leq i \quad w_k = 0$$

Ad esempio, se la sequenza, lunga 10, è inizialmente:

i_add	*i_add + 1*	*i_add + 2*	*i_add + 3*	*i_add + 4*	*i_add + 5*	*i_add + 6*	*i_add + 7*	*i_add + 8*	*i_add + 9*
128	0	64	0	0	0	25	0	0	0

Figure 1: Situazione iniziale

deve diventare:

i_add	*i_add + 1*	*i_add + 2*	*i_add + 3*	*i_add + 4*	*i_add + 5*	*i_add + 6*	*i_add + 7*	*i_add + 8*	*i_add + 9*
128	31	64	31	64	30	25	31	25	30

Figure 2: Situazione finale

1.2 Interfaccia del componente

Il componente da progettare ha la seguente interfaccia:

```
entity project_reti_logiche is
    port(
        i_clk      : in std_logic;
        i_rst      : in std_logic;
        i_start    : in std_logic;
        i_add      : in std_logic_vector(15 downto 0);
        i_k        : in std_logic_vector(9 downto 0);

        o_done     : out std_logic;

        o_mem_addr : out std_logic_vector(15 doento 0);
        i_mem_addr : in std_logic_vector(7 downto 0);
        o_mem_data : out std_logic_vector(7 downto 0);
        o_mem_en   : out std_logic;
        o_mem_we   : out std_logic;
    );
end project_reti_logiche;
```

In particolare:

1. `i_clk`: segnale di clock comune con il test bench.
2. `i_rst`: segnale asincrono di reset della macchina.
3. `i_start`: segnale di start generato dal testbench.
4. `i_k`: vettore che rappresenta la lunghezza della sequenza da analizzare.
5. `i_add`: vettore che rappresenta l'indirizzo a partire dal quale è memorizzata la sequenza.
6. `o_done`: segnale che comunica la fine dell'elaborazione.
7. `o_mem_addr`: vettore in uscita contenenete un indirizzo.
8. `i_mem_data`: vettore in entrata dalla memoria che contiene il dato in seguito ad una richiesta di lettura.
9. `o_mem_data`: vettore in uscita contenente il valore da scrivere in memoria.
10. `o_mem_en`: segnale di enable della memoria per poterci comunicare (in lettura o scrittura).
11. `o_mem_we`: segnale di *write enable* della memoria: 1 permette la scrittura, 0 la lettura.

1.3 Dati e descrizione della memoria

La memoria è indirizzata al byte. Le parole w_i e i valori di credibilità γ_i sono *unsigned* di 8bit. Gli indirizzi $[\texttt{i\_add}, \texttt{i\_add} + 2\ , ...,\ \texttt{i\_add} + 2 \cdot (\texttt{i\_k} - 1)]$ sono utilizzati per memorizzare le parole e $[\texttt{i\_add} + 1, \texttt{i\_add} + 3\ , ...,\ \texttt{i\_add} + 2 \cdot \texttt{i\_k} - 1]$ i valori di credibilità. È possibile pensare alla memoria con cui il modulo si interfaccia come una tabella:

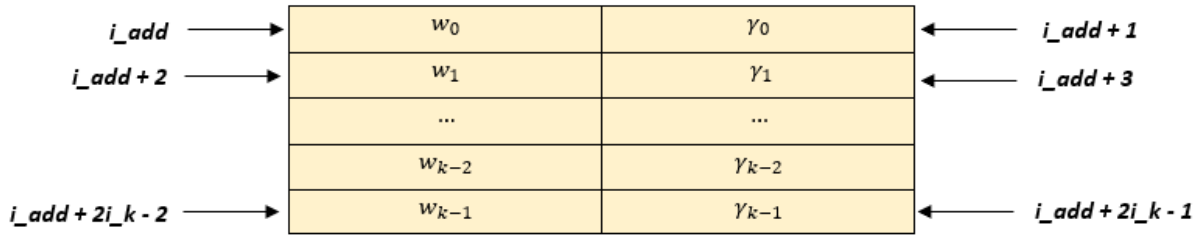


Figure 3: Tabella rappresentante la memoria

1.4 Design

Quando il segnale `i_start` viene asserito il modulo progettato inizia l'elaborazione, notificandone il termine alzando l'uscita `o_done`; a quel punto, l'utilizzare (in particolare il testbench) può, in qualsiasi momento, porre `i_start` = 0 a cui fa seguito `o_done` = 0.
Il segnale `i_rst` asincrono resetta la macchina: deve essere garantito che prima del primo fronte di salita di `i_start`, `i_rst` sia stato asserito almeno una volta, mentre tra un'elaborazione e l'altra `i_rst` può anche essere sempre 0.

2 Datapath

2.1 Entità utilizzate

A livello generale, il modulo progettato utilizza le seguenti entità:

1. *ALU (per somma e sottrazione)*:

```
component ALU
generic(
    N : integer
);
port(
    first_operand        : in std_logic_vector(N - 1 downto 0);
    second_operand       : in std_logic_vector(N - 1 downto 0);
    operation            : in std_logic;
    result               : out std_logic_vector(N - 1 downto 0)
);
end component;
```

   Per la somma `operation` $= 0$ e per la sottrazione `operation` $= 1$. In questo caso, se `first_operand` $<$ `second_operand` il valore di `result` è 0.

2. *Shifter*:

```
component shifter
generic(
    N : integer
);
port(
    value_in  : in std_logic_vector(N - 1 downto 0);
    value_out : out std_logic_vector(N downto 0)
);
end component;
```

   Il componente realizza lo shift logico: il valore in ingresso viene moltiplicato per 2.

3. *Comparatori*:

```
component comparator
generic(
    N : integer
);
port(
    first_operand  : in std_logic_vector(N - 1 downto 0);
    second_operand : in std_logic_vector(N - 1 downto 0);
    result         : out std_logic
);
end component;
```

   In questo caso, si è scelto di avere `result` $= 0$ quando i due ingressi sono diversi, `result` $= 1$ altrimenti.

4. *Registri*:

```
component memory_register
generic(
    N : integer
);
port(
```

```
    clock : in std_logic;
    enable : in std_logic;
    reset  : in std_logic;
    value_in : std_logic_vector(N - 1 downto 0);
    value_out : out std_logic_vector(N - 1 downto 0)
    );
end component;
```

Il segnale di reset è asincrono. Il valore `value_in` viene memorizzato nel registro sul fronte di salita del *clock*.

5. *Multiplexer (MUX)*:

```
component multiplexer
    generic(
        N : integer
    );
    port(
        control   : in std_logic;
        value_0   : in std_logic_vector(N - 1 downto 0);
        value_1   : in std_logic_vector(N - 1 downto 0);
        value_out : out std_logic_vector(N - 1 downto 0)
    );
end component;
```

2.2 Blocchi funzionali

I componenti istanziati sono organizzati in blocchi che realizzano una specifica funzione:

1. ***Generazione dell'indirizzo di memoria finale***

   Calcola il valore dell'ultimo indirizzo di memoria valido, cioè:

   $$\texttt{signal\_last\_valid\_address} = \texttt{i\_add} + 2 \cdot \texttt{i\_k} - 1$$

   In particolare:

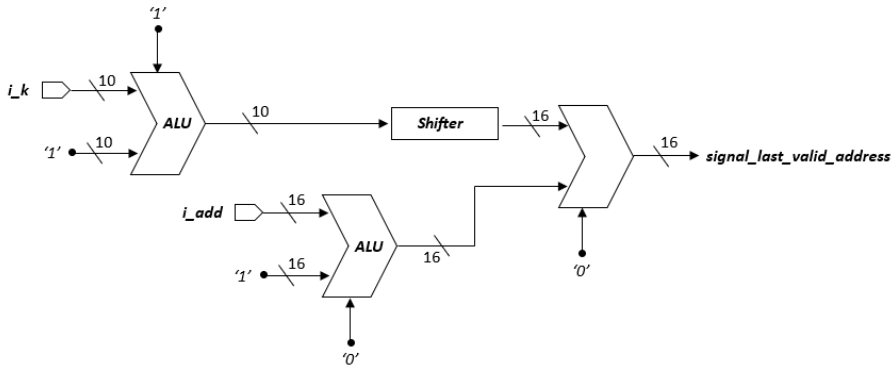


Figure 4: Blocco di generazione dell'indirizzo di memoria finale

   Notare che per non avere *overflow* nella somma tra `i_add` e `i_k` e non dover istanziare una ALU con 3 ingressi, si è scelto di calcolare prima `i_add`$+1$ (che non può generare *overflow*) e $2 \cdot$(`i_k`-1), dopodiché sommarli. Infatti, l'ultimo indirizzo valido è:

   $$\texttt{signal\_last\_valid\_address} = \texttt{i\_add} + 2 \cdot (\texttt{i\_k} - 1) + 1 = \texttt{i\_add} + 2 \cdot \texttt{i\_k} - 1$$

   Se `i_k` fosse 0, si avrebbe `signal_last_address` $=$ `i_add` $+ 1$ (analogamente al caso `i_k` $= 1$): questa situazione è trattata dal blocco di gestione dell'indirizzo corrente.

2. ***Gestione indirizzo corrente***

   Si memorizza il valore corrente dell'indirizzo in un registro a 16 bit.
   L'incremento è eseguito dalla ALU e propagato nel registro sul fronte di salita del *clock* se `enable_address_register` $= 1$ e $\texttt{MUX}_3 = 1$.
   Il comparatore viene utilizzato per confrontare l'indirizzo corrente con quello finale ($\texttt{MUX}_2 = \texttt{MUX}_6 = 1, \texttt{MUX}_5 = 0$) o iniziale ($\texttt{MUX}_2 = 0, \texttt{MUX}_6 = 1$) e per determinare se la sequenza è vuota ($\texttt{MUX}_2 = \texttt{MUX}_5 = 1, \texttt{MUX}_6 = 0$).

   Si noti che il registro non viene mai resettato: il valore precedentemente memorizzato viene sovrascritto all'esecuzione successiva.

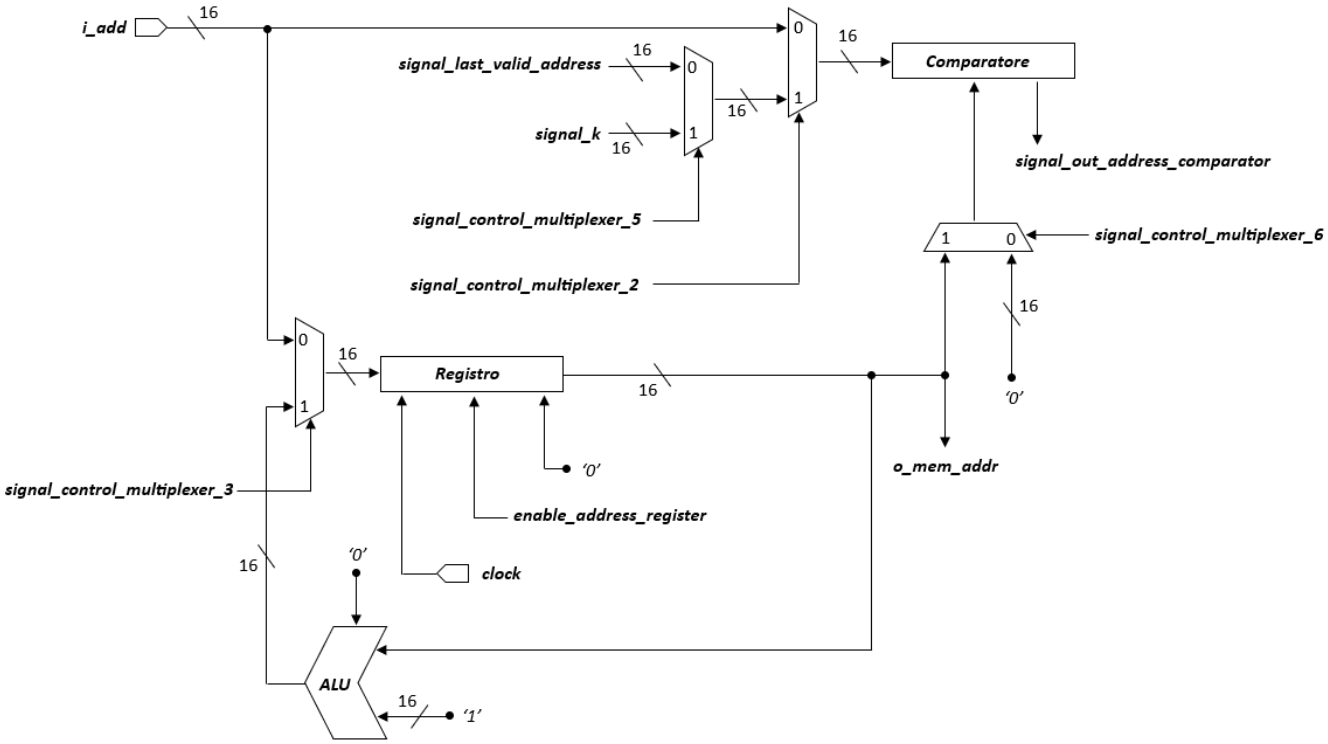


Figure 5: Blocco di gestione dell'indirizzo corrente

Nel seguito, per motivi di impaginazione, si indicherà con σ_1 segnale di uscita dal comparatore degli indirizzi.

3. ***Gestione delle parole***

   L'ultima parola valida viene mantenuta in un registro da 8 bit.
   Il risultato fornito dal comparatore viene utilizzato dalla logica della FSM per stabile se alzare `enable_data_register` e, quindi, memorizzare il nuovo valore sul fronte di salita del *clock* successivo.
   Il registro può essere anche resettato.

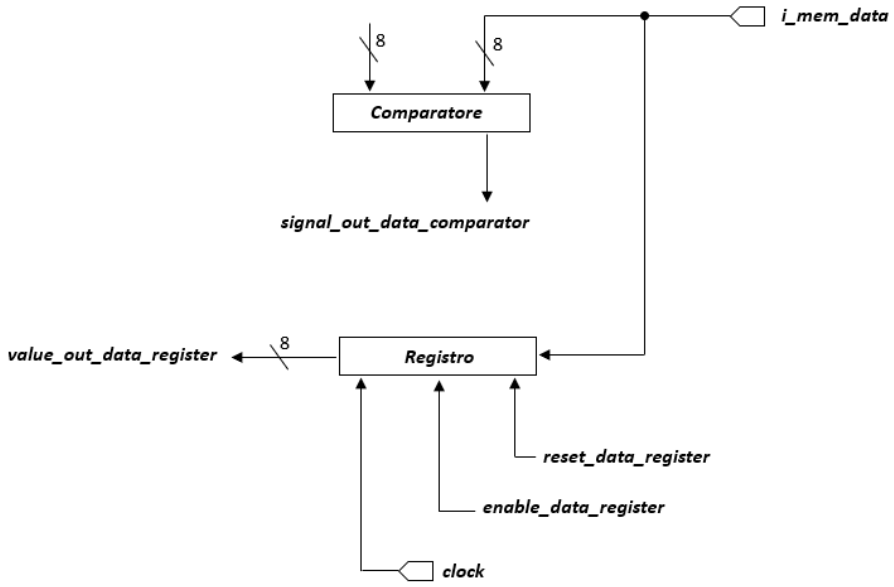


Figure 6: Blocco di gestione delle parole

Nel seguito, si indicherà con σ_2 segnale di uscita dal comparatore delle parole.

4. ***Gestione della credibilità***

La credibilità viene memorizzata in un registro a 8 bit.
Mediante un blocco ALU, che lavora come sottrattore, si calcola il valore di credibilità successivo. La FSM può scegliere di aggiornare il registro con il risultato della ALU ($\texttt{MUX}_4$ = `enable_credibility_register` = 1) al fronte di salita del *clock*, con il valore 31 ($\texttt{MUX}_4$ = 0) oppure resettarlo a 0. Infine, con un ulteriore MUX si seleziona il valore da scrivere in memoria.

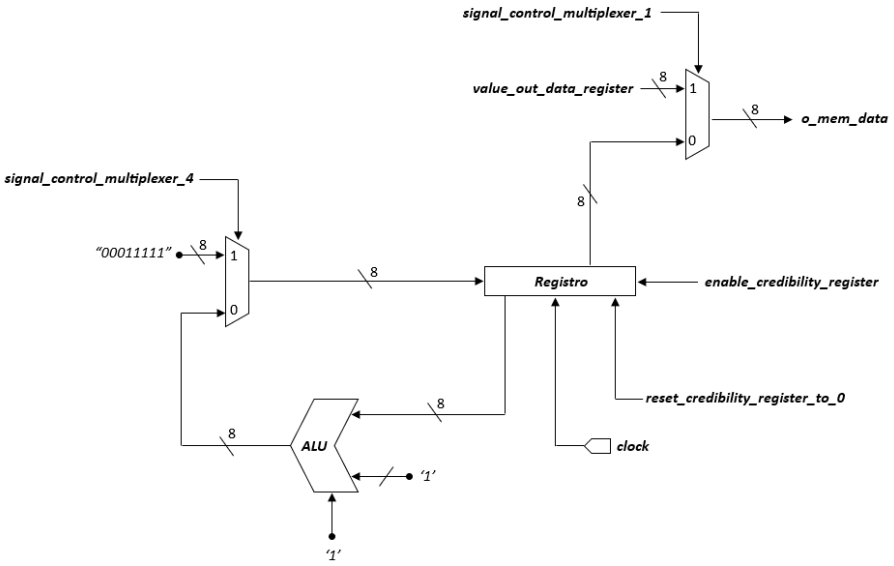


Figure 7: Blocco di gestione dei valori di credibilità

3 FSM

La FSM gestisce le i moduli istanziati, la memoria e le loro interazioni. La sua interfaccia è:

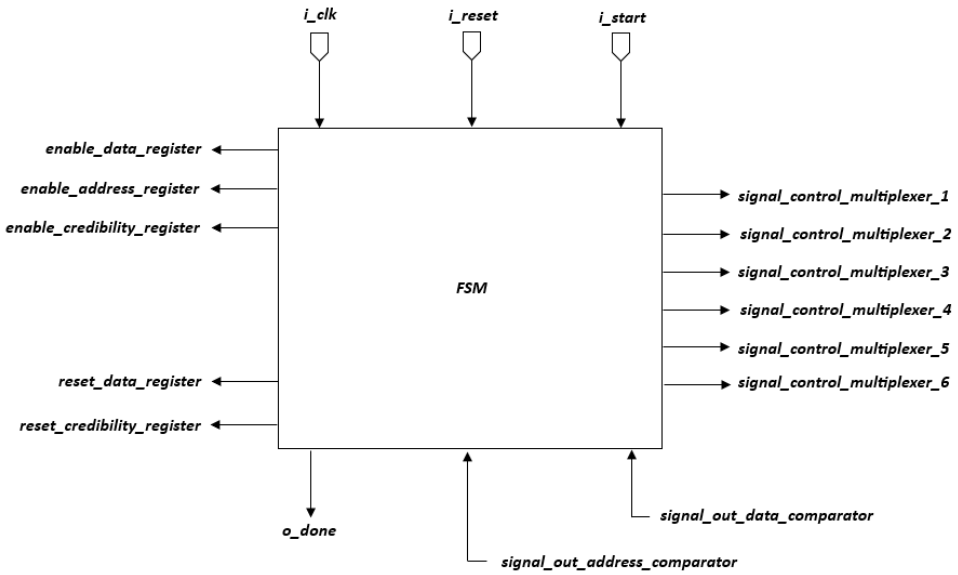
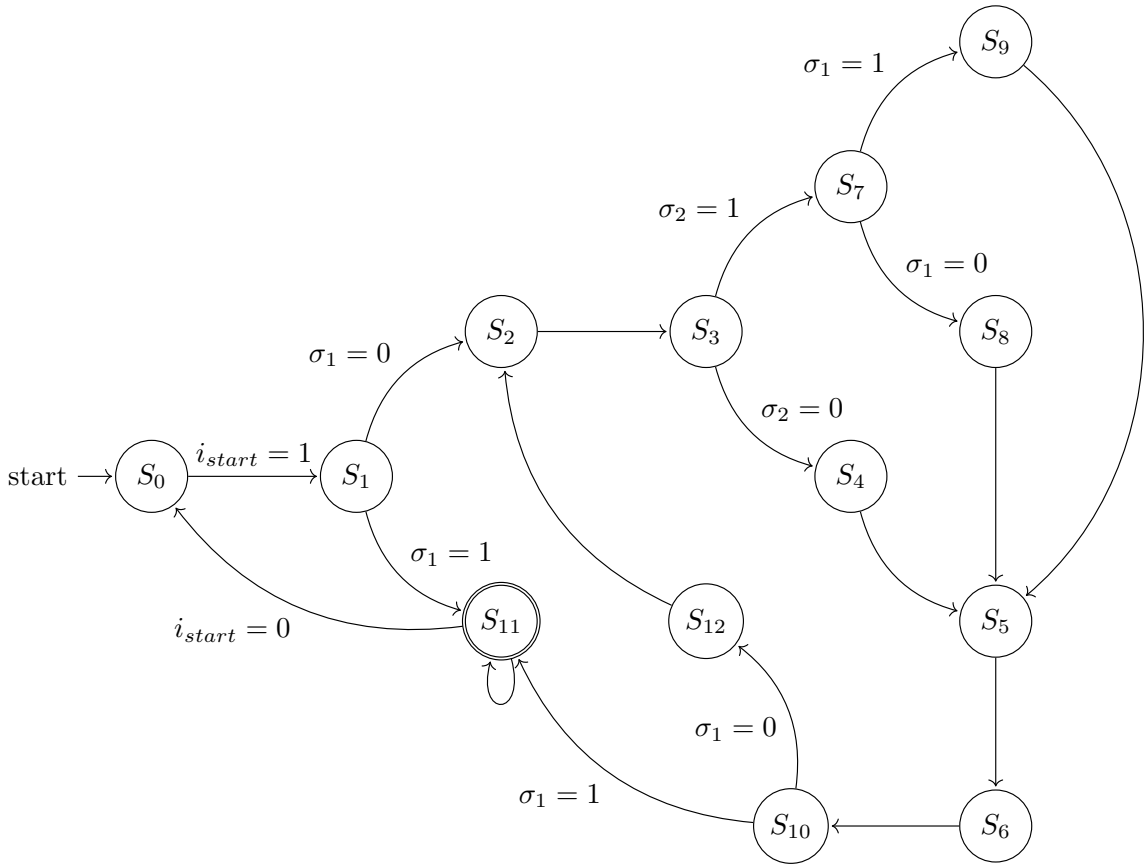


Figure 8: Blocco FSM

3.1 Diagramma degli stati

La macchina è formata da 13 stati, organizzati nel diagramma seguente:



3.2 Descrizione degli stati

Si analizzano, ora, uno per uno, gli stati della FSM:

1. `IDLE` (S_0)

   È lo stato iniziale della FSM. Si resettano il registro delle parole e quello delle credibilità. Si attende che l'utilizzatore asserisca `i_start`.

2. `CHECK_K_ZERO_AND_STORE_I_ADD` (S_1)

   Si controlla che la sequenza non sia vuota, $\texttt{MUX}_2 = \texttt{MUX}_5 = 1$ e $\texttt{MUX}_6 = 0$. Se lo è, $\sigma_1 = 1$, l'elaborazione termina. Parallelamente, mediante $\texttt{MUX}_3 = 0$ si memorizza l'indirizzo di partenza nel registro.

3. `ASK_MEMORY` (S_2)

   Alla memoria viene richiesta la lettura della parola il cui indirizzo è memorizzato nel registro degli indirizzi.

4. `CHECK_WORD` (S_3)

   Si attende la risposta della memoria e si esegue il controllo $\texttt{i\_mem\_data} = 0$.

5. `WORD_IS_NOT_ZERO` (S_4)

   La parola w_i letta è maggiore di 0. Occorre, quindi, memorizzarla nel registro delle parole: $\texttt{enable\_data\_register} = 1$.
   Il valore γ_i va posto a 31, perciò $\texttt{MUX}_4 = 0$ e $\texttt{enable\_credibility\_register} = 1$. Il valore non viene riscritto in memoria.

6. `NEXT_ADDRESS_CREDIBILITY` (S_5)

   È lo stato preparatorio prima della scrittura in memoria di γ_i. $\texttt{MUX}_3 = 1$, quindi si incrementa il valore memorizzato nel registro degli indirizzi.

7. `WRITE_CREDIBILITY` (S_6)

   Viene scritto in memoria il valore di credibilità: $\texttt{o\_mem\_en = o\_mem\_we} = 1$ e $\texttt{MUX}_4 = 0$.

8. `READ_ZERO` (S_7)

   La parola w_i letta è 0. È necessario determinare se tale parola è la prima della sequenza: $\texttt{MUX}_2 = 0$, $\texttt{MUX}_6 = 1$, cioè il comparatore confronta `i_add` e `value_out_address_register`.

9. `FIRST_ELEMENT_NOT_ZERO` (S_8)

   $w_i = 0$ non è la prima parola. Bisogna decrementare la credibilità ($\texttt{MUX}_4 = 1$ e $\texttt{enable\_credibility\_register} = 1$). Il registro dei valori non va aggiornato e il suo valore scritto in memoria ($\texttt{MUX}_1 = \texttt{o\_me\_en} = \texttt{o\_mem\_we} = 1$).

10. `FIRST_ELEMENT_IS_ZERO` (S_9)

    $w_0 = 0$. Il registro delle credibilità e dei valori vanno resettati ($\texttt{reset\_data\_register} = 1$ e $\texttt{reset\_credibility\_regiter} = 1$).

11. `CHECK_FINISH` (S_{10})

    La FSM controlla che siano stati scritti in memoria tutti i valori
    (cioè $\texttt{signal\_current\_address} = \texttt{signal\_last\_address}$): $\texttt{MUX}_2 = \texttt{MUX}_6 = 1, \texttt{MUX}_5 = 0$.

12. `DONE` (S_{11})

    La FSM, inizialmente, alza il segnale `o_done`. Quando l'utilizzatore abbassa `i_start` anche `o_done` viene abbassato e si torna in `IDLE`, altrimenti si rimane in `DONE`.

13. `NEXT_ADDRESS_WORD` (S_{12})

    Se la sequenza non è terminata, occorre aggiornare il registro degli indirizzi: `enable_address_register` = $\texttt{MUX}_3 = 1$.

A valle della sintesi, l'*encoding* scelto dal tool è stato il seguente:

State	New Encoding	Previous Encoding
idle	0000	0000
check_k_zero_and_store_i_add	0001	0001
ask_memory	0010	0100
check_word	0011	0101
read_zero	0100	1001
first_element_is_zero	0101	1011
first_element_not_zero	0110	1010
word_is_not_zero	0111	0110
next_address_credibility	1000	0111
write_credibility	1001	1000
check_finish	1010	0010
next_address_word	1011	0011
done	1100	1100

Figure 9: Encoding a valle della sintesi

3.3 Progetto della FSM

Per progettare la FSM è stato utilizzato l'approccio *behavioural*. In particolare, sono stati definiti tre processi:

1. `delta`: funzione di uscita (δ), gestisce tutti i segnali della macchina (macchina di Moore).

   ```
   delta : process(current_state)
   ```

2. `lambda`: funzione di stato prossimo (λ), determina `next_state` in base a `current_state` e agli ingressi.

   ```
   lambda : process(current_state, signal_out_address_comparator,
                    signal_out_data_comparator, i_start)
   ```

3. `state_reg`: realizza l'effettiva commutazione dello stato sul fronte di salita del *clock* e gestisce il segnale `i_rst`. Il codice che lo implementa è il seguente:

   ```
   state_reg : process(i_clk, i_rst)
       begin
           if i_rst = '1' then
               current_state <= IDLE;
           else
               if rising_edge(i_clk) then
                   current_state <= next_state;
               end if;
           end if;
   end process;
   ```

4 Testing del modulo

Oltre a quello fornito, il modulo è stato ulteriormente testato con altri 15 *testbench*, alcuni dei quali mirati a verificare possibili *corner case* e corse critiche della macchina. Di seguito, si riportano quelli più significativi e di alcuni anche il grafico (*post-synthesis functional*) generato dal *waveform viewer*:

1. *Sequenza senza zeri*

   La memoria contiene una sequenza arbitraria di parole $w_i > 0 \ \ \forall i$.

2. *Sequenza mista*

   La sequenza è di lunghezza arbitraria e $0 \leq w_i \leq 255 \ \ \forall i$. Più di 31 zeri consecutivi non possono presentarsi e $w_0 \neq 0$.

3. *Sequenza con zero iniziale*

   La sequenza è di lunghezza arbitraria e $0 \leq w_i \leq 255 \ \ \forall i$. $w_0 = 0$ ed $w_i \neq 0$ per qualche i. Più di 31 zeri consecutivi non possono presentarsi.

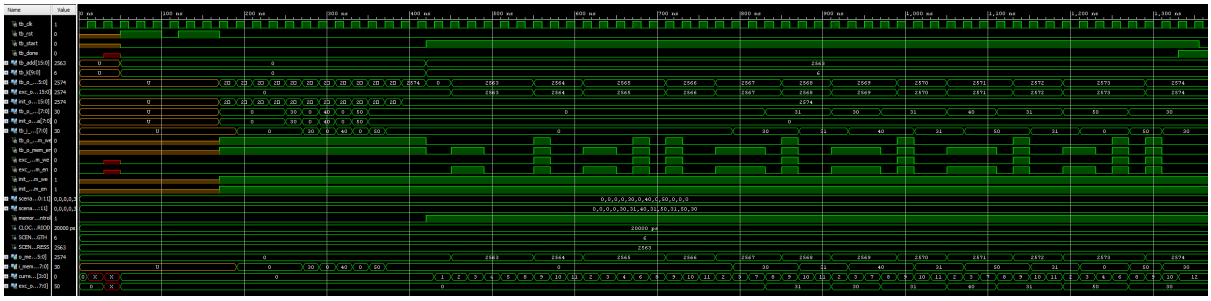


Figure 10: simulazione completa e superamento del test

4. *Sequenza di soli zeri*

   La sequenza è di lunghezza arbitraria e $w_i = 0 \ \ \forall i$.

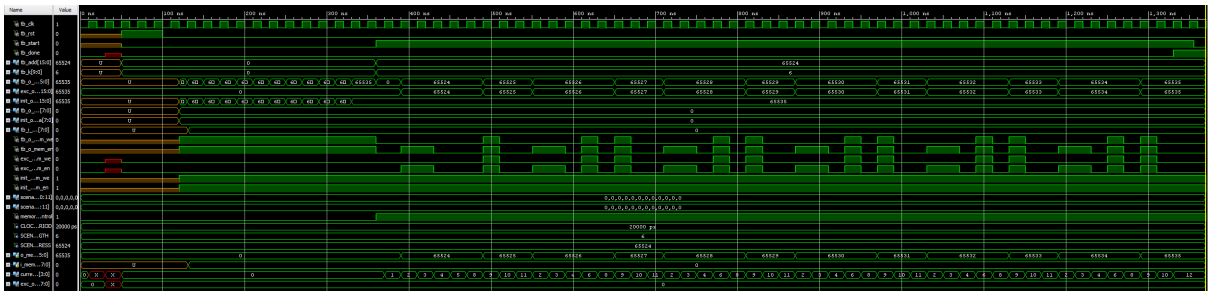


Figure 11: simulazione completa e superamento del test con $w_i = 0 \ \ \forall i$

5. *Sequenza con almeno* 31 *zeri consecutivi* ($w_0 \neq 0$)

   La sequenza è di lunghezza arbitraria e contiene una sequenza di almeno 31 parole 0; $w_0 \neq 0$.

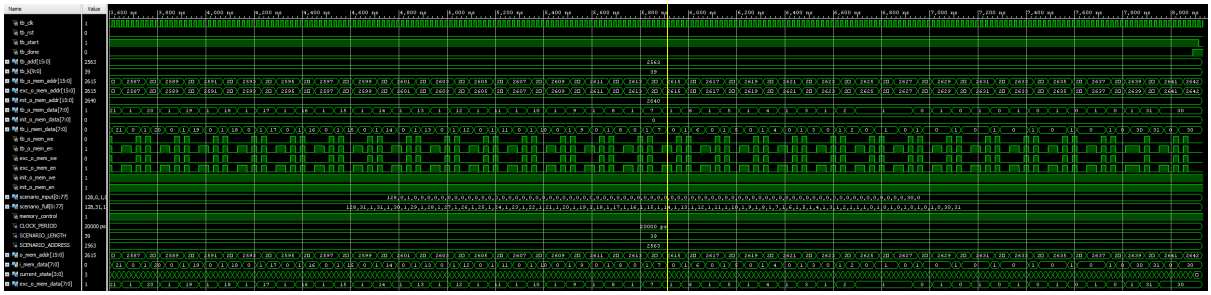


Figure 12: ultima fase della simulazione e superamento del test

6. *Sequenza di lunghezza massima*

   Poiché k è a 10bit, la lunghezza massima della sequenza da elaborare è 1022.

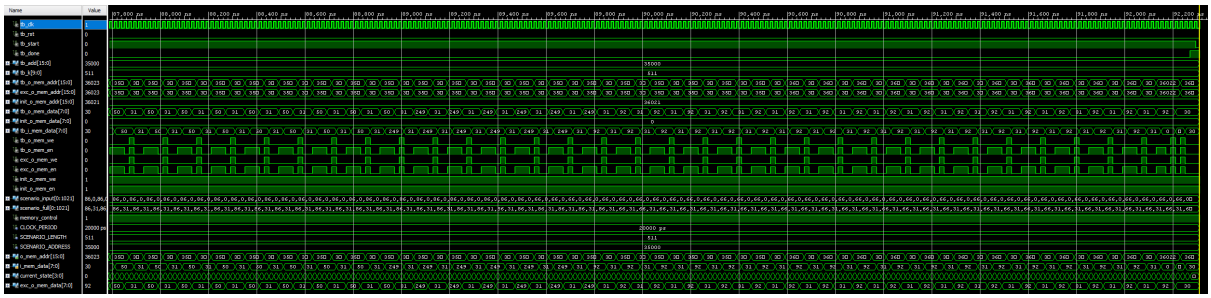


Figure 13: ultima fase della simulazione e superamento del test con `i_k = 511`

7. *Sequenza di lunghezza zero*

   La sequenza è vuota, cioè `i_k` $= 0$.

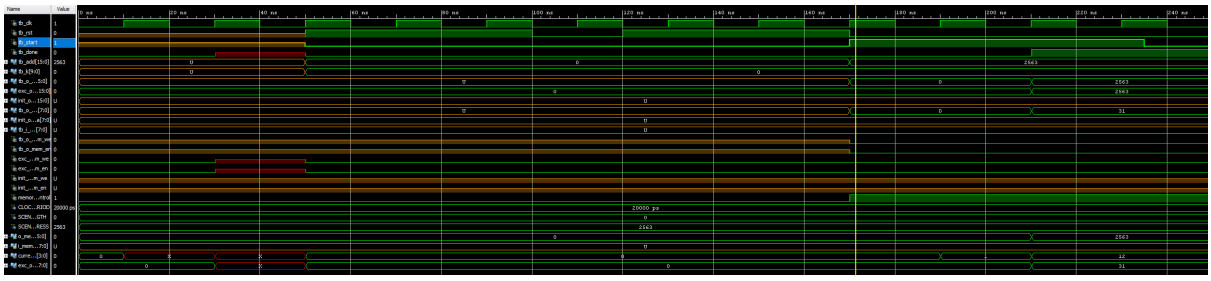


Figure 14: ultima fase della simulazione e superamento del test con `i_k = 0`

8. *Sequenza di lunghezza unitaria*

   Questo caso di test deriva dalla struttura *datapath* scelta per calcolare `signal_last_valid_address`.

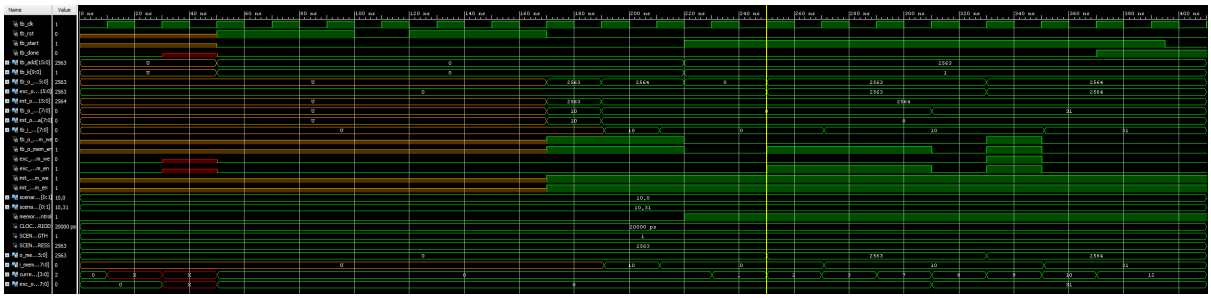


Figure 15: ultima fase della simulazione e superamento del test con `i_k = 1`

9. *Ultima parola in posizione* $2^{16} - 2 = 65534$

   La sequenza è arbitraria. Si sceglie `i_add` e `i_k` in modo che $i_{add} + 2i_k = 2^{16}$

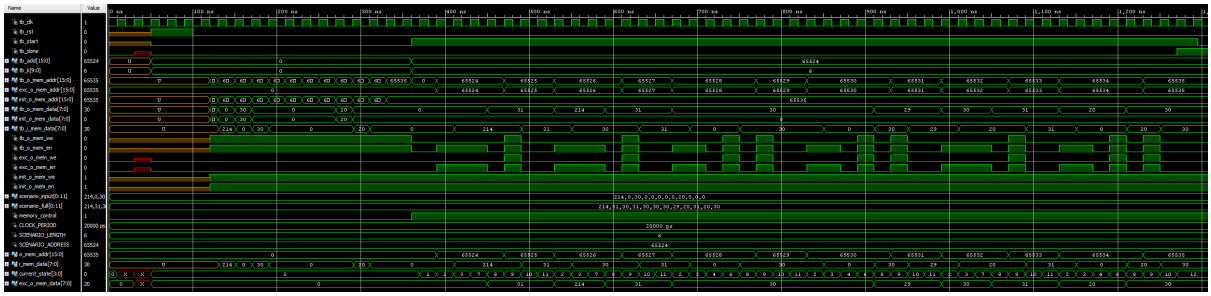


Figure 16: simulazione completa e superamento del test con $i_{add} + 2i_k = 2^{16}$

10. *Doppia esecuzione con reset intermedio*

    Al componente sono chieste due elaborazioni successive intervallate dall'asserimento del segnale di reset.

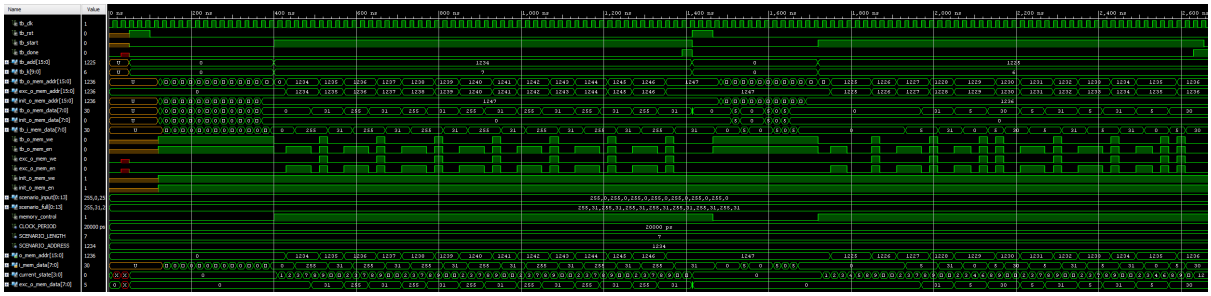


Figure 17: simulazione completa e superamento del test

11. *Doppia esecuzione senza reset intermedio*

    Al componente sono richieste due elaborazioni successive tra le quali `i_rst` è tenuto a 0.

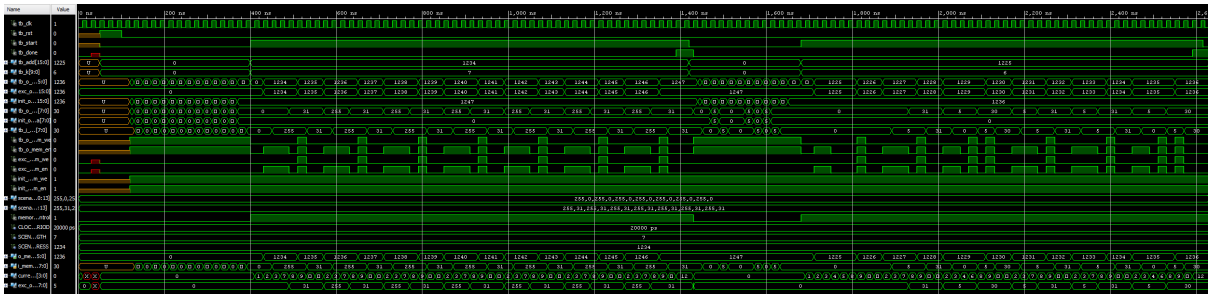


Figure 18: simulazione completa e superamento del test

12. *Sequenze (parzialmente) sovrapposte senza reset intermedio*

    Al componente sono richieste due elaborazioni successive, senza reset intermedio, con sequenze che iniziano allo stesso indirizzo di memoria.

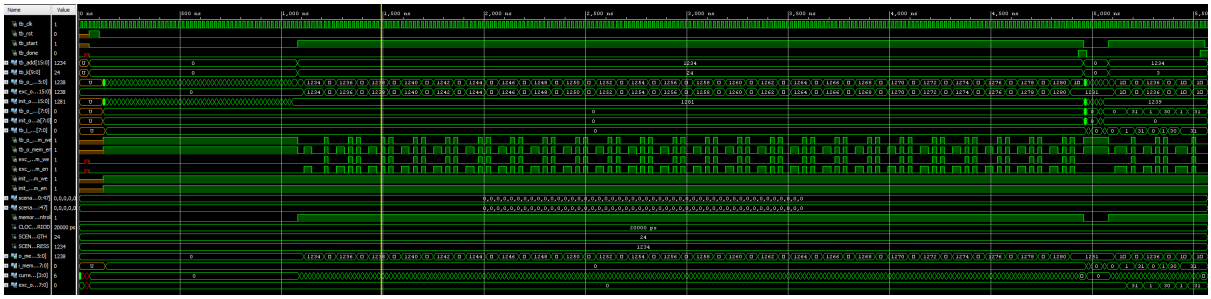


Figure 19: simulazione completa e superamento del test

13. *Fronti di transizione di* `i_start` *molto ravvicinati*

    Al componente sono chieste due elaborazioni successive, senza reset intermedio e con fronti di transizione di `i_start` molto ravvicinati (20`ns`).

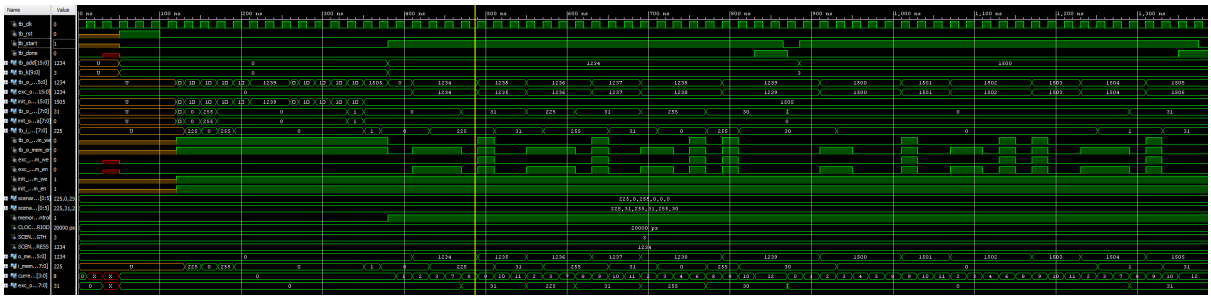


Figure 20: simulazione completa e superamento del test

Infine, la scrittura di casi di test è stata automatizzata. Mediante un programma Java, dopo aver specificato la lunghezza del *test case*, si ottengono i vettori `scenario_input` e `scnario_full` da inserire nel *test bench*.

5 Risultati

Il componente progettato è stato testato, ottenendo sempre un risultato positivo, nelle modalità *behavioural* e *post-synthesis functional*.

La complessità temporale della macchina è $O(n)$. Quella spaziale, invece, è costante, $O(1)$ perché si utilizzano solamente $16 + 8 + 8 = 32$ registri per gli indirizzi, le parole e i valori di credibilità (nella soluzione *datapath*, senza tener conto della codifica degli stati).

Il requisto temporale del *clock* è soddisfatto. Infatti, il comando `report_timing` fornisce un $\Delta =$ `required time` $-$ `arrival time` $= 16.557$`ns`. Di seguito la schermata:

```
Timing Report

Slack (MET) :             16.557ns  (required time - arrival time)
  Source:                 c_address_register/stored_value_reg[3]/C
                            (rising edge-triggered cell FDRE clocked by clock  {rise@0.000ns fall@5.000ns period=20.000ns})
  Destination:            FSM_sequential_current_state_reg[0]/D
                            (rising edge-triggered cell FDCE clocked by clock  {rise@0.000ns fall@5.000ns period=20.000ns})
  Path Group:             clock
  Path Type:              Setup (Max at Slow Process Corner)
  Requirement:            20.000ns  (clock rise@20.000ns - clock rise@0.000ns)
  Data Path Delay:        3.293ns  (logic 1.531ns (46.493%)  route 1.762ns (53.507%))
  Logic Levels:           6  (CARRY4=2 LUT3=1 LUT5=1 LUT6=2)
  Clock Path Skew:        -0.145ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD):    1.693ns = ( 21.693 - 20.000 )
    Source Clock Delay      (SCD):    2.001ns
    Clock Pessimism Removal (CPR):    0.163ns
```

Figure 21: risultati comando `report_timing`

Il circuito finale, sintetizzato dal tool, contiene 0 latch e 35 FF, csoì suddivisi:

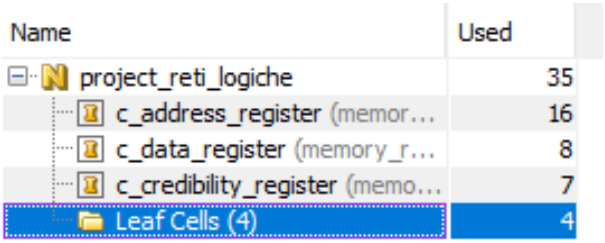


Figure 22: risultati comando `report_utilizazion` → *register as flip flop*

6 Ottimizzazioni

L'ottimizzazione più importante messa in atto è la riduzione del numero degli stati e dei componenti. Inizialmente, infatti, per memorizzare gli indirizzi e i valori di credibilità erano impiegati registri appositamente progettati: sul fronte di salita del *clock* il primo aumenta di 1 l'indirizzo se `signal_sum = 1` e il secondo diminuisce di 1 la credibilità se `signal_subtract = 1`. Ora, invece, sono entrambi `memory_register` (affiancati dalla logica mostrata), così due stati sono stati eliminati.
Anche gli stati che si occupano di memorizzare la credibilità sono comuni: `WORD_IS_NOT_ZERO`, `FIRST_ELEMENT_NOT_ZERO` e `FIRST_ELEMENT_IS_ZERO` confluiscono tutti e tre in `NEXT_ADDRESS_CREDIBILITY`.
Infine, uno stato vien eliminato grazie al fatto che la scrittura in memoria della parola e l'aggiornamento della credibilità possono essere eseguite in insieme: `FIRST_ELEMENT_NOT_ZERO` (S_8) fa $\texttt{MUX}_4 = \texttt{enabl\_credibility\_register} = 1$ e $\texttt{MUX}_1 = \texttt{o\_mem\_en} = \texttt{o\_mem\_we} = 1$ parallelamente.

Il numero di scritture in memoria è minimo: infatti, si è optato per non riscrivere in memoria parole $w_i \neq 0$, ma passare subito alla definizione della credibilità.