RelazioneProgettoRetiLogiche

Francesco Spangaro - Luca Tosetti

April 2023

Matricola: 955106 - 956958
Codice persona: 10734844 - 10739865
Docente: Gianluca Palermo

POLITECNICO
MILANO 1863

1 Introduzione

1.1 Specifiche generali

1.1.1 Descrizione

Il progetto consiste nel descrivere un modulo hardware (HW) che si interfacci con una memoria. L'elaborazione dei dati avviene in diverse fasi: vengono forniti degli input da cui è ricavato un indirizzo, questo indirizzo, viene poi restituito alla memoria, che solo a questo punto risponderà con il dato salvato nella cella identificata. Infine, il dato viene messo in output su una delle 4 apposite uscite, insieme al segnale "**DONE**" che avvisa della fine dell'elaborazione.

1.1.2 Funzionamento

Il funzionamento del modulo prevede di avere inizialmente tutte le uscite ed il segnale "**DONE**" pari a 0. Questa situazione rimane invariata finchè il segnale in ingresso "**START**" è uguale a 0, nel momento in cui il segnale "**START**" passa a 1, il componente inizia a leggere i bit presenti sul secondo segnale (seriale) in ingresso, "**W**". La lettura di questi bit continua fintanto che "**START**" rimane pari a 1. La stringa letta è così divisa:

- I primi due bit compongono l'indirizzo di una delle 4 uscite, su cui verrà visualizzato il dato letto da memoria.
- I restanti bit (da un minimo di 0, ad un massimo di 16) compongono l'indirizzo di memoria che identifica la cella su cui è salvato il dato richiesto.

Il segnale "**START**" rimane pari ad 1 per un minimo di 2 cicli di clock (2 bit) fino ad un massimo di 18 cicli di clock (18 bit).

Una volta che il segnale "**START**" torna a 0, i bit letti fin'ora vengono unificati nella posizione meno significativa di una stringa da 16 bit, che sarà completata con tanti zeri nella posizione più significativa quanti bit mancano per arrivare al suo completamento. A questo punto l'indirizzo salvato viene utilizzato per leggere il dato da memoria. Il dato viene, quindi, caricato sull'uscita identificata dai primi due bit forniti dal segnale in ingresso "**W**".
Il valore delle uscite rimane nullo fintanto che "**DONE**" è pari a 0. Quando "**DONE**" viene settato a 1 (entro 20 cicli di clock dal passaggio di "**START**" da 1 a 0), il valore caricato sulle uscite viene mostrato. Sull'uscita identificata dai primi due bit del segnale "**W**" viene mostrato il dato restituito dalla memoria, sulle altre viene invece mostrato il dato relativo all'ultima iterazione che le ha coinvolte. Le uscite e "**DONE**" presentano i dati per un solo ciclo di clock, per poi tornare a 0.

1.2 Strumenti forniti

Tra gli strumenti fornitoci, possiamo trovare:

- Interfaccia di memoria:

```
-- Single-Port Block RAM Write-First Mode (recommended template)
--
-- File: rams_02.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity rams_sp_wf is
port(
  clk  : in  std_logic;
  we   : in  std_logic;
  en   : in  std_logic;
  addr : in  std_logic_vector(15 downto 0);
  di   : in  std_logic_vector(7 downto 0);
  do   : out std_logic_vector(7 downto 0)
);
end rams_sp_wf;

architecture syn of rams_sp_wf is
type ram_type is array (65535 downto 0) of std_logic_vector(7 downto 0);
signal RAM : ram_type;
begin
  process(clk)
    begin
    if clk'event and clk = '1' then
      if en = '1' then
        if we = '1' then
          RAM(conv_integer(addr)) <= di;
          do                      <= di after 2 ns;
        else
          do <= RAM(conv_integer(addr)) after 2 ns;
        end if;
      end if;
    end if;
  end process;
end syn;
```

- Interfaccia modulo HW:

```
entity project_reti_logiche is
    port (
        i_clk   : in std_logic;
        i_rst   : in std_logic;
        i_start : in std_logic;
        i_w     : in std_logic;

        o_z0    : out std_logic_vector(7 downto 0);
        o_z1    : out std_logic_vector(7 downto 0);
        o_z2    : out std_logic_vector(7 downto 0);
        o_z3    : out std_logic_vector(7 downto 0);
        o_done  : out std_logic;

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

I segnali sono:

1. i_clk: segnale di "**CLOCK**" generato dal testbench
2. i_rst: segnale di "**RESET**" asincrono generato dal testbench, utilizzato per inizializzare la macchina, pronta per ricevere il primo segnale di "**START**".
3. i_start: segnale di "**START**" generato dal testbench
4. i_w: segnale di input generato dal testbench
5. o_z0: primo canale d'uscita
6. o_z1: secondo canale d'uscita
7. o_z2: terzo canale d'uscita
8. o_z3: quarto canale d'uscita
9. o_done: segnale di uscita che comunica la fine dell'elaborazione
10. o_mem_addr: segnale di invio dell'indirizzo alla memoria
11. i_mem_data: segnale dal quale si riceve il dato letto dalla memoria
12. o_mem_en: segnale di "**ENABLE**", necessario per poter comunicare con la memoria.
13. o_mem_we: segnale di "**WRITE ENABLE**", necessario per poter scrivere sulla memoria.

2 Architettura

2.1 Descrizione ad alto livello

Il modulo HW consiste di 9 differenti sottomoduli, ogniuno realizzato per una specifica funzione necessaria al corretto comportamento del modulo stesso. Questi sottomoduli hanno tutti in comune il segnale di "**CLOCK**" proveniente dall'esterno del modulo HW e quello di "**RESET**", necessario per resettare il contenuto e funzionamento dei singoli sottomoduli.

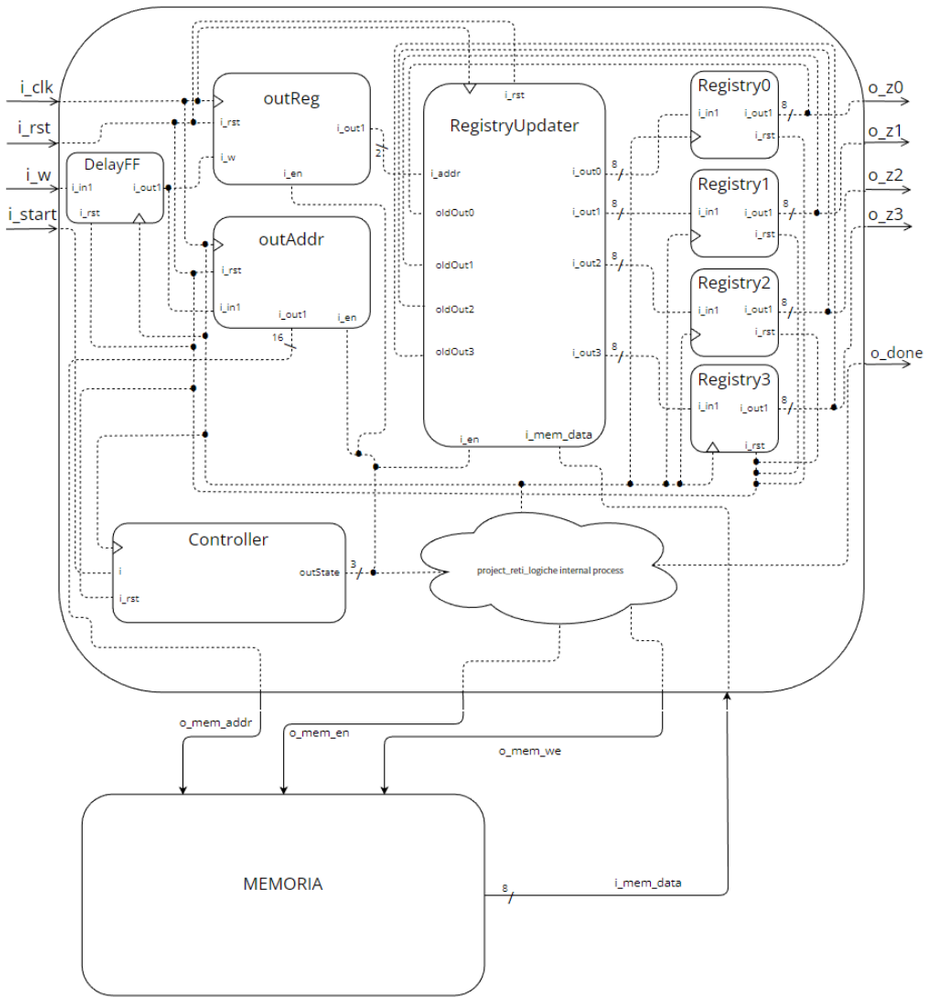

Figura 1: Descrizione grafica del modulo.

2.1.1 delayFF

Sottomodulo che si occupa di ritardare il segnale "**W**" di un ciclo di clock. Questo accorgimento si è reso necessario per la gestione della lettura dei singoli bit tramite i sottomoduli *OutReg* e *OutAddr*, e per l'implementazione della *FSM*. Il sottomodulo si accerta che tutti i bit del segnale "**W**" vengano letti correttamente.

2.1.2 outReg

Sottomodulo che si occupa della lettura dei primi due bit della sequenza in ingresso sul segnale "**W**". Tale combinazione di bit viene poi mandata in uscita

al sottomodulo *RegistryUpdater* per poter selezionare il registro corretto nel quale andare a memorizzare il dato in uscita.

2.1.3 outAddr

Sottomodulo che si occupa della lettura dei bit successivi ai primi due presenti sul segnale "**W**", mentre "**START**" è pari a 1. La lettura è stata implementata in modo che il modulo legga un bit alla volta dall'ingresso "**W**": il bit viene sempre inserito nella posizione 0 di un vettore, e ad ogni ciclo di "**CLOCK**", questo vettore viene shiftato a sinistra, ripetendo l'operazione finché "**START**" è pari a 1. Per poter recuperare il dato contenuto nella cella di memoria identificata dall'indirizzo appena letto, il sottomodulo invia l'indirizzo alla memoria tramite il canale "**o_mem_addr**".

2.1.4 RegistryUpdater

Sottomodulo che si occupa di gestire l'aggiornamento dei 4 registri visibili nella figura 1. L'aggiornamento viene gestito sulla base dell'indirizzo letto dal sottomodulo *OutReg*. Il sottomodulo *RegistryUpdater* riceve in ingresso il segnale "**i_mem_data**" contenente il dato da salvare sul registro identificato dal sottomodulo *OutReg*. Inoltre, il sottomodulo riceve in input il valore memorizzato all'interno dei 4 *Registry*, in modo che tutti i registri, ad eccezione di quello da aggiornare, continuino a tenere memorizzato il dato che contenevano al ciclo di "**CLOCK**" precedente.

2.1.5 Registry8bit

Sottomodulo che si occupa di tenere memorizzato il dato da mostrare in uscita quando "**DONE**" diventa pari a 1. Questo modulo riceve dal *RegistryUpdater* il dato da memorizzare.

2.1.6 Controller

Sottomodulo che si occupa dell'implementazione della *FSM* nel componente. Si tratta dell'effettivo controller del componente descritto nel progetto. La macchina specificata è una macchina di Moore, dato che l'output è definito solo dallo stato in cui si trova la *FSM*, e non dall'input. La macchina si basa infatti sui segnali "**START**" e "**RST**", capendo in quale stato si deve spostare.
Lo stato "**S0**" è lo stato iniziale, raggiunto dalla macchina dopo aver ricevuto il primo segnale di reset sul canale "**RST**". L'input della *FSM* che viene considerato è il segnale "**START**", gli stati successivi allo stato "**S0**" servono ad abilitare al momento corretto i diversi componenti, attraverso un vettore di segnali interni: l'uscita "**outState**".
I vari valori di "**outState**" possono essere mappati in una tabella, dalla quale possiamo trarre l'informazione del componente attivato da ogni segnale.

Controller	
000	Nessun componente interno abilitato
001	Componente *OutReg* abilitato
010	Componente *OutAddr* abilitato
011	Componente *RegistryUpdater* abilitato
100	Segnale "**DONE**" posto pari ad 1, le uscite passano da 0 al valore richiesto
101	Mai utilizzato, quindi DC
110	Mai utilizzato, quindi DC
111	Mai utilizzato, quindi DC

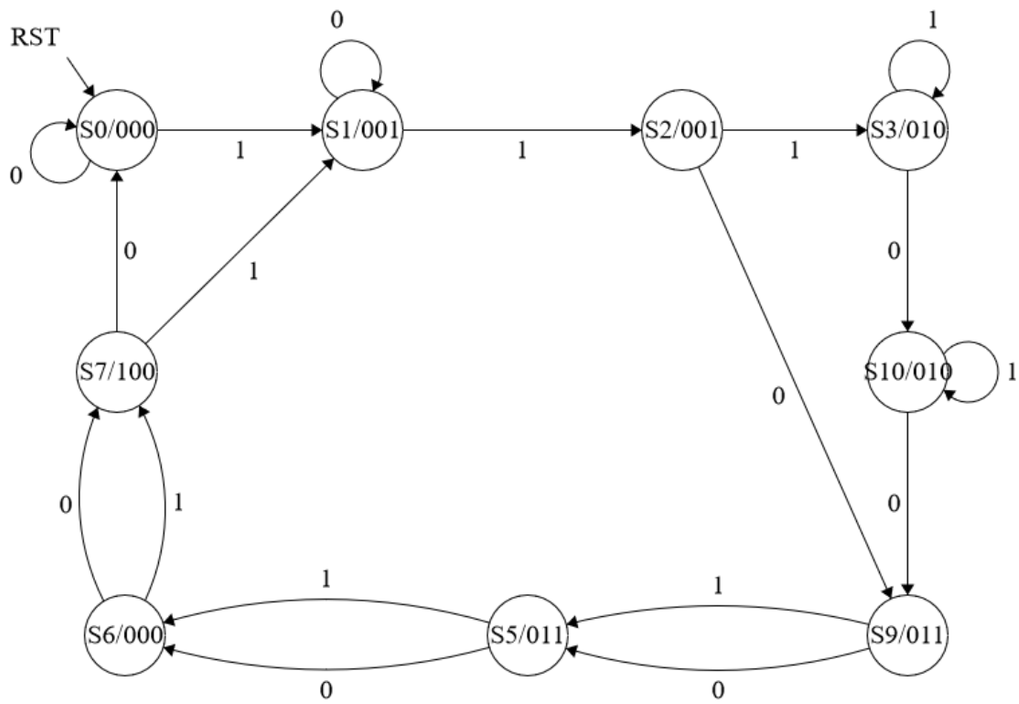

Figura 2: Schema della *FSM*.

3 Risultati sperimentali

3.1 Report di sintesi

3.1.1 Risorse utilizzate

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	121	0	134600	0.09
LUT as Logic	121	0	134600	0.09
LUT as Memory	0	0	46200	0.00
Slice Registers	171	0	269200	0.06
Register as Flip Flop	171	0	269200	0.06
Register as Latch	0	0	269200	0.00
F7 Muxes	0	0	67300	0.00
F8 Muxes	0	0	33650	0.00

3.1.2 FSM Codificata

State	New Encoding	Previous Encoding
s0	0000	0000
s1	0001	0001
s2	0010	0010
s3	0011	0011
s10	0100	1010
s4	0101	0100
s9	0110	1001
s5	0111	0101
s6	1000	0110
s7	1001	0111
s8	1010	1000

```
INFO: [Synth 8-3354] encoded FSM with state register 'current_state_reg' using encoding 'sequential'
 in module 'controller'
---------------------------------------------------------------------------------
Finished RTL Optimization Phase 2 : Time (s): cpu = 00:00:06 ; elapsed = 00:00:06 . Memory (MB):
peak = 385.605 ; gain = 154.164
---------------------------------------------------------------------------------
```

3.1.3 Statistiche RTL

```
---------------------------------------------------------------------------------
Start RTL Component Statistics
---------------------------------------------------------------------------------
Detailed RTL Component Info :
+---Adders :
      2 Input     32 Bit       Adders := 1
+---Registers :
                  32 Bit    Registers := 1
                  16 Bit    Registers := 2
                   8 Bit    Registers := 12
                   2 Bit    Registers := 2
                   1 Bit    Registers := 3
+---Muxes :
      2 Input     32 Bit        Muxes := 2
      2 Input     16 Bit        Muxes := 2
      2 Input      8 Bit        Muxes := 20
      5 Input      8 Bit        Muxes := 4
     19 Input      4 Bit        Muxes := 1
     11 Input      3 Bit        Muxes := 1
      2 Input      2 Bit        Muxes := 6
      2 Input      1 Bit        Muxes := 13
---------------------------------------------------------------------------------
```

3.1.4 Info Project_Reti_Logiche

```
Module project_reti_logiche
Detailed RTL Component Info :
+---Registers :
                   8 Bit    Registers := 4
                   1 Bit    Registers := 2
+---Muxes :
       2 Input      8 Bit        Muxes := 4
       2 Input      1 Bit        Muxes := 1
Module controller
Detailed RTL Component Info :
+---Muxes :
      19 Input      4 Bit        Muxes := 1
      11 Input      3 Bit        Muxes := 1
```

3.1.5 Info Controller

```
Module controller
Detailed RTL Component Info :
+---Muxes :
      19 Input      4 Bit        Muxes := 1
      11 Input      3 Bit        Muxes := 1
```

3.1.6 Info delayFF

```
Module delayFF
Detailed RTL Component Info :
+---Registers :
                   1 Bit    Registers := 1
```

3.1.7 Info outReg

```
Module outReg
Detailed RTL Component Info :
+---Adders :
       2 Input     32 Bit       Adders := 1
+---Registers :
                  32 Bit    Registers := 1
                   2 Bit    Registers := 2
+---Muxes :
       2 Input     32 Bit        Muxes := 2
       2 Input      2 Bit        Muxes := 6
       2 Input      1 Bit        Muxes := 5
```

3.1.8 Info outAddr

```
Module outAddr
Detailed RTL Component Info :
+---Registers :
                  16 Bit    Registers := 2
+---Muxes :
       2 Input     16 Bit        Muxes := 2
       2 Input      1 Bit        Muxes := 5
```

3.1.9 Info registryUpdater

```
Module deMuxMux
Detailed RTL Component Info :
+---Registers :
                   8 Bit    Registers := 4
+---Muxes :
       2 Input      8 Bit        Muxes := 8
       5 Input      8 Bit        Muxes := 4
       2 Input      1 Bit        Muxes := 2
```

3.1.10 Info registry8bit

```
Module registry8bit
Detailed RTL Component Info :
+---Registers :
                8 Bit    Registers := 1
+---Muxes :
      2 Input      8 Bit        Muxes := 2
```

3.2 Simulazioni

3.2.1 TB1

Uno dei primi test che abbiamo effettuato è stato verificare il comportamento della macchina tramite il testbench fornitoci in partenza. Il testbench contiene alcune iterazioni, un singolo segnale di reset per inizializzare il modulo, una generica durata di "**START**" ed una generica sequenza di bit sul segnale "**W**". Abbiamo utilizzato questo test per verificare il corretto funzionamento del modulo in un caso generale senza particolari condizioni.

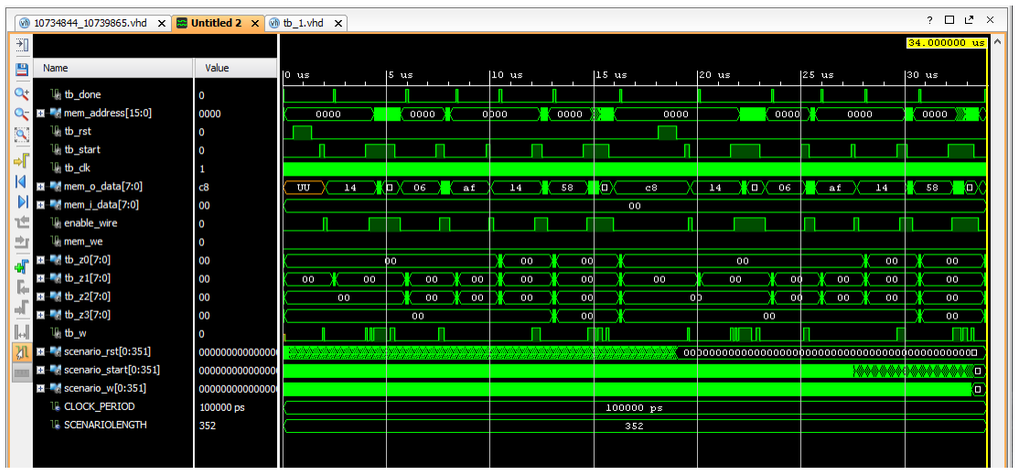

Figura 3: Test iniziale.

3.2.2 TB2

Un caso particolare che abbiamo voluto testare, è quello in cui il segnale "**W**" è pari ad una stringa di 18 bit pari a 1.

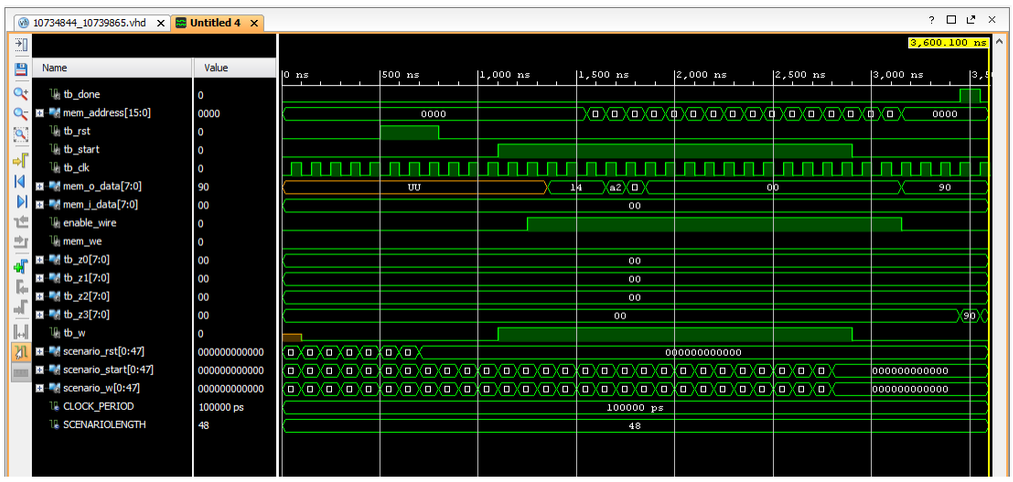

Figura 4: 18 bit a 1.

3.2.3 TB3

In questo caso abbiamo voluto testare la dualità della macchina rispetto al caso di test TB2 (18 bit a 0 sul segnale "**W**").

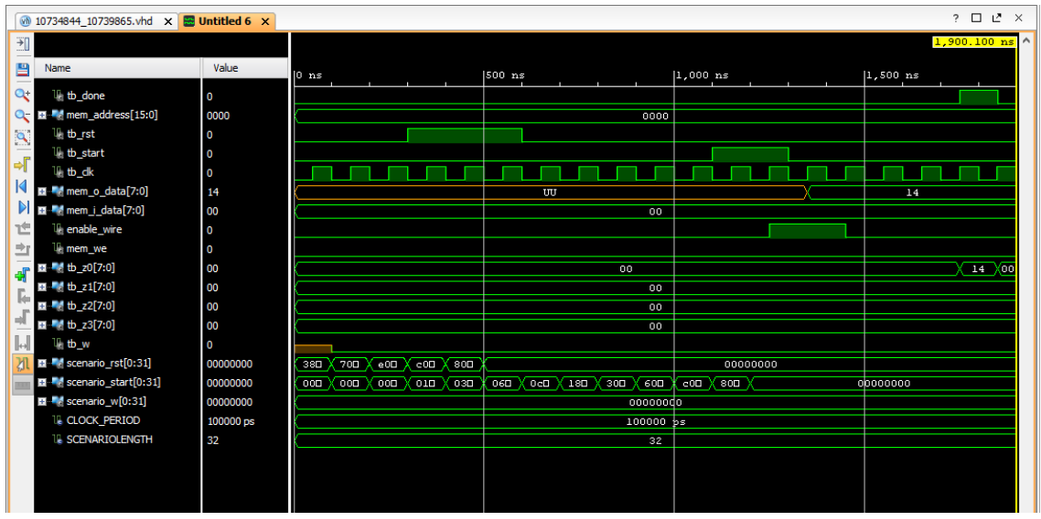

Figura 5: 18 bit a 0.

3.2.4 TB4

In questo test abbiamo verificato che il comportamento del modulo HW rimanga coerente nel corso del tempo, superando un notevole numero di iterazioni consecutive. Il comportamento delle uscite (ovvero il mantenimento del valore precedentemente mostrato) deve essere corretto resettando di tanto in tanto il modulo dopo la ricezione di segnali di reset.

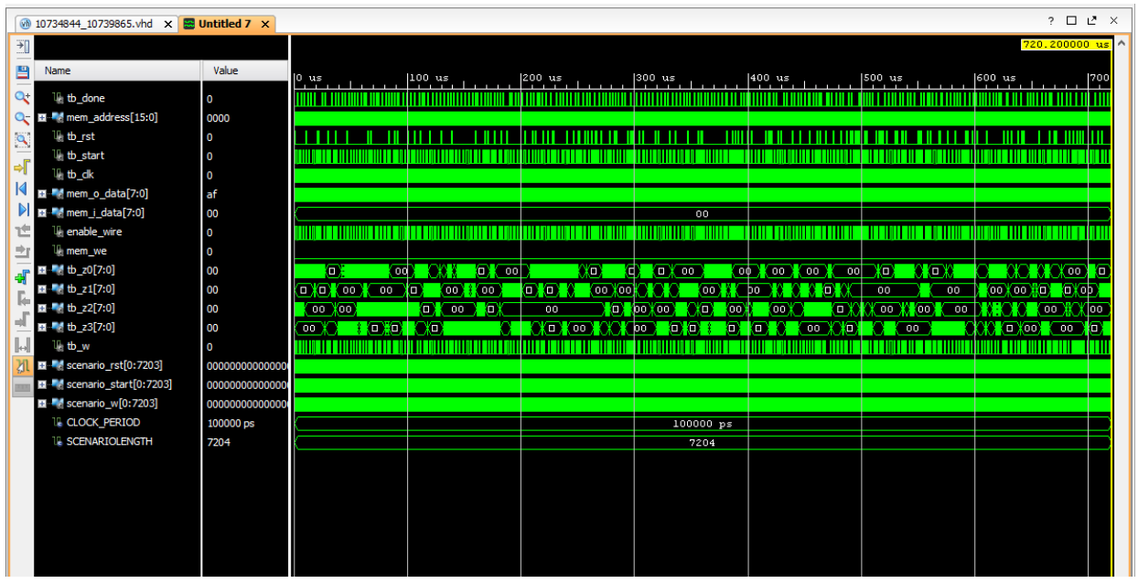

Figura 6: 300 iterazioni consecutive.

4 Conclusioni

Alla luce dei risultati ottenuti dai test effettuati (tutti con successo, sia in behavioural che in post sintesi), il modulo funziona correttamente, come da specifica. Abbiamo deciso, al momento della progettazione della FSM, di utilizzare un quantitativo di stati superiore a quello effettivamente necessario. Questa strategia ci assicura che, in caso di eventuali tempi di propagazione dei segnali e delle porte più lunghi di quelli riportati nei nostri test, il componente continui a funzionare correttamente. Ad esempio, abbiamo deciso di introdurre in fase di progettazione gli stati "**S10**" ed "**S9**" utilizzati rispettivamente: il primo per avere la certezza che il sottomodulo *OutAddr* sia in grado di completare l'elaborazione anche tenendo in considerazione eventuali ritardi, in particolare mantenendo attivo il sottomodulo durante la lettura; il secondo è stato introdotto per avere la certezza che i dati si propagassero in tempo tra i sottomoduli prima di iniziare l'elaborazione. Abbiamo gestito il caso in cui il segnale di "**RST**" arrivi in concomitanza della modifica del segnale "**DONE**" da 0 ad 1, portando "**DONE**" e i segnali di uscita a 0 senza curarci del ciclo di clock in cui devono rimanere settati ad 1. Considerando che questo preciso caso non è stato descritto nella specifica del componente, abbiamo deciso di valutare il segnale di "**RST**" prioritario rispetto all'elaborazione dei dati. In conclusione, nonostante l'aggiunta degli stati prima descritti e presa in esame la nostra ipotesi, siamo riusciti a descrivere correttamente il componente, rispettando la specifica in ogni suo termine.