# 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 nell'andare a descrivere un modulo HW che si interfacci ad una memoria, estraendone dati sulla base di indirizzi letti da input dal modulo stesso e caricando tali dati su una delle apposite 4 uscite, gestendo la visione dei dati tramite un segnale "**DONE**".

### 1.1.2 Funzionamento

Più in dettaglio: il funzionamento del modulo prevede di avere inizialmente tutte le uscite e il segnale "**DONE**" a 0. Questa situazione rimane tale finchè il segnale in ingresso "**START**" rimane a 0, nel momento in cui il segnale "**START**" passa a 1, si possono iniziare 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, in particolare:

- I primi due bit letti compongono l'indirizzo di una delle 4 uscite, su cui verrà inserito il dato letto da memoria.
- I restanti bit (da un minimo di 0, ad un massimo di 16) verranno usati come indirizzo per accedere alla memoria e leggere il dato presente in tale indirizzo.

Da notare come il segnale "**START**" rimane pari a 1 per un minimo di 2 cicli di clock (2 bit) fino ad un massimo pari a 18 cicli di clock (18 bit).

A questo punto l'indirizzo letto può essere usato per leggere il dato in memoria, una volta fatto ciò tale dato deve essere caricato sull'uscita indicata dai primi due bit letti dal segnale in ingresso "W".
Ciononostante, il valore delle uscite (compresa quella su cui è stato caricato il dato) deve rimanere nullo fintanto che "**DONE**" rimane 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 deve essere mostrato (anche quelli caricati su altre uscite in iterazioni precedenti) per un solo ciclo di clock, dopodichè il segnale "**DONE**" torna a 0, cosi come le uscite.

## 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;
```

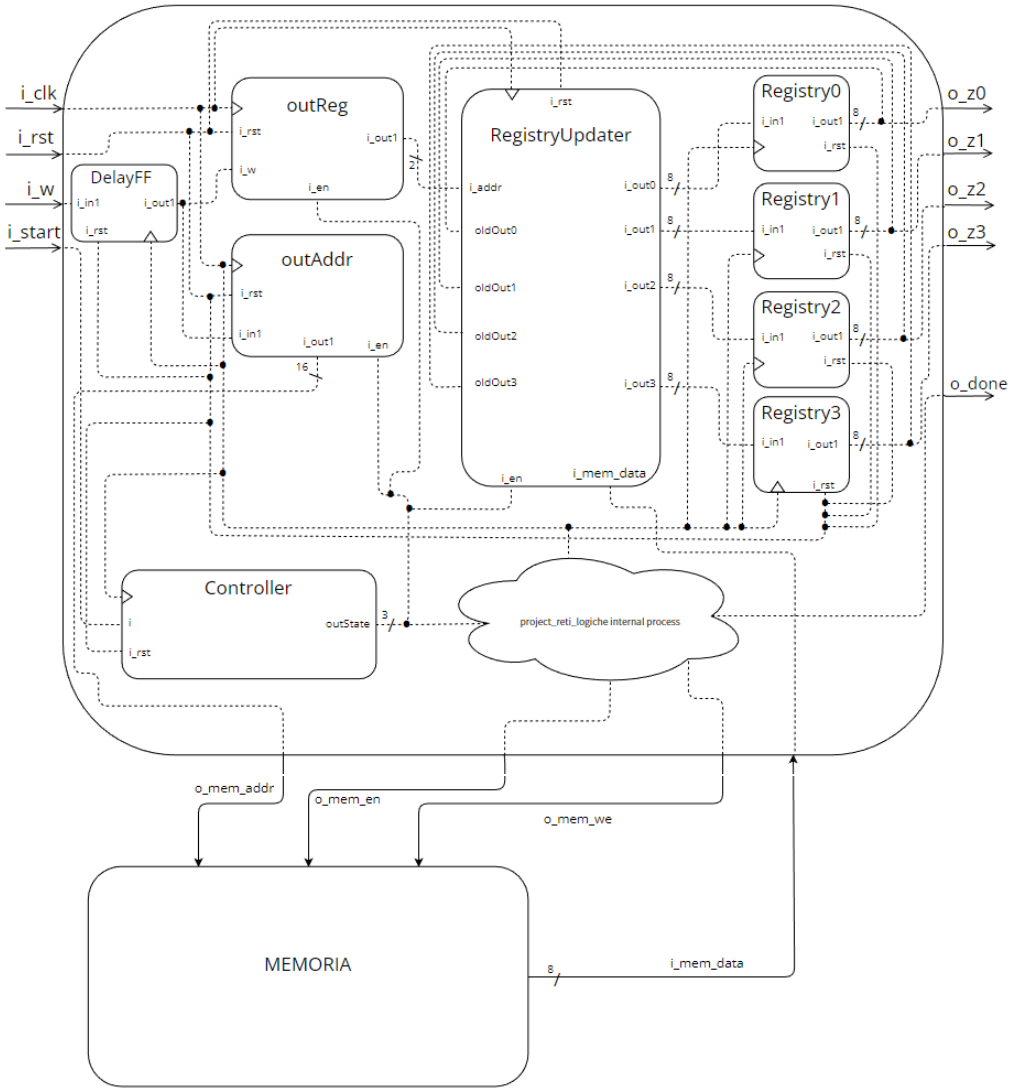
Più dettagliatamente:

1. i_clk: segnale di "**CLOCK**" generato dal testbench
2. i_rst: segnale di "**RESET**" asincrono generato dal testbench, usato 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 che permette di inviare l'indirizzo dal quale leggere il dato 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 sia in scrittura che in lettura con la memoria.
13. o_mem_we: segnale di "**WRITE ENABLE**", necessario per poter scrivere nella memoria.

# 2 Architettura

## 2.1 Descrizione ad alto livello

Il modulo HW consiste di 6 differenti sottomoduli, ognuno realizzato per una specifica funzionalità necessaria al corretto comportamento del modulo stesso. Tali sottomoduli hanno tutti in comune il segnale di "**CLOCK**" proveniente dall'esterno del modulo HW e quello di "**RESET**", necessario per appunto resettare il contenuto/funzionamento dei singoli sottomoduli.



### 2.1.1 delayFF

Sottomodulo che si occupa di ritardare il segnale "**W**" di un ciclo di clock. Questo accorgimento si è reso necessario nella realizzazione del nostro progetto per come siamo andati a gestire la lettura dei singoli bit tramite i sottomoduli *OutReg* e *OutAddr*, e per come abbiamo gestito l'implementazione della *FSM*. Infatti, senza l'utilizzo di questo sottomodulo, abbiamo riscontrato il problema di non riuscire a leggere il primo bit della sequenza in ingresso tramite il segnale "**W**", cosa che andava ad influire sull'inserimento del dato letto da memoria nella giusta uscita, poichè la *FSM* necessita di leggere almeno un bit pari a 1 dal segnale start, per poter "abilitare" alla lettura il sottomodulo *OutReg*.

### 2.1.2 outReg

Sottomodulo molto semplice. 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 scegliere il registro corretto nel quale andare a memorizzare il dato in uscita.

### 2.1.3 outAddr

Sottomodulo altrettanto semplice. Si occupa della lettura dei bit successi ai primi due presenti sul segnale "**W**", nel mentre che "**START**" si trova a 1. La lettura è stata implementata in modo che il modulo legga un bit alla volta dall'ingresso "**W**", inserendo sempre nella posizione 0 (quella più a destra) di un vettore, e ad ogni ciclo di "**CLOCK**", tale segnale temporaneo viene shiftato a sinistra, ripetendo l'azione fintanto che il sottomodulo rimane attivo (ovvero per come è stato costruita la *FSM*, fino a quando "**START**" ritorna a 0). Infine questo sottomodulo si occupa anche di inviare in uscita l'indirizzo di memoria letto per poter recuperare dalla memoria stessa il dato contenuto in tale indirizzo di memoria, che verrà spedito al *RegistryUpdater*.

### 2.1.4 RegistryUpdater

Sottomodulo che si occupa di gestire l'aggiornamento dei 4 registri visibili nell'immagine soprastante, utilizzati per memorizzare il valore che deve avere l'uscita fintanto che questa non venga sovrascritta, oppure ci sia un reset della macchina (in quel caso il contenuto dei registri viene completamente azzerato). L'aggiornamento viene gestito sulla base dell'indirizzo di due bit

letto dal sottomodulo *OutReg*, il quale determina quale dei 4 registri riceverà il dato letto nel frattempo dalla memoria per poter essere aggiornato. Oltre a tutto questo, il *RegistryUpdater* riceve in input il valore memorizzato all'interno dei 4 *Registry*, questo per fare 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.

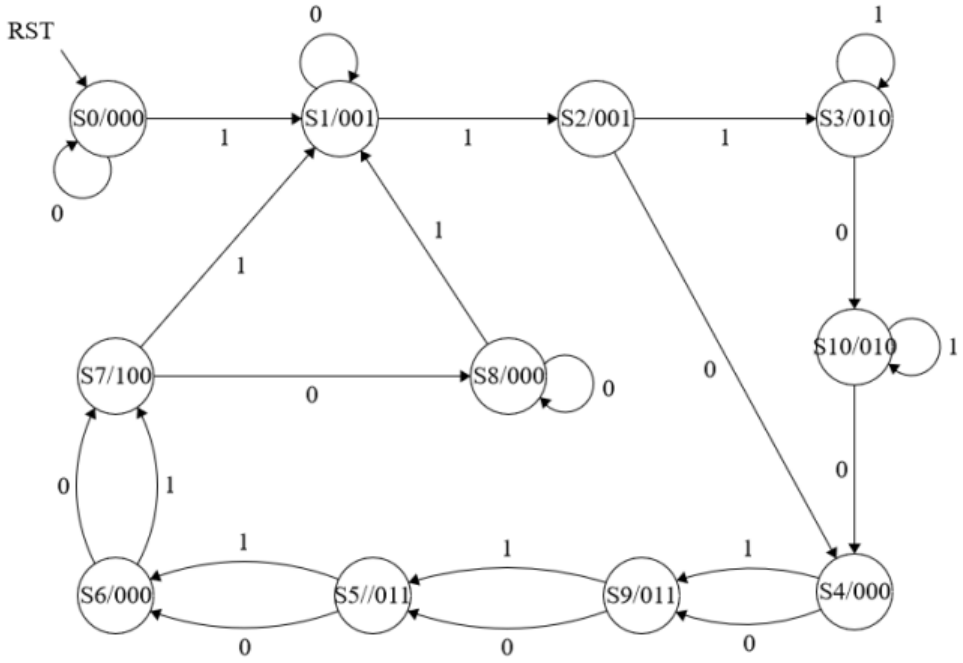
### 2.1.5 Registry8bit

Sottomodulo che si occupa di tenere memorizzato al suo interno il dato da mostrare in uscita, qualora "**DONE**" diventi pari a 1. Tale modulo riceve in ingresso il dato da memorizzare dal *RegistryUpdater*

### 2.1.6 Controller

Sottomodulo che si occupa dell'implementazione della *FSM* nel componente, è l'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, basandosi sui segnali "**START**" e "**RST**" capisce in quali stati deve andare.
Lo stato "**S0**" è lo stato iniziale, raggiunto dalla macchina dopo aver ricevuto il primo segnale di reset in input sul canale "**RST**". L'input della *FSM* che verrà considerato sarà allora solo il segnale "**START**", gli stati successivi allo stato "**S0**" servono solo 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, per capire a quale segnale corrisponde l'attivazione di quale componente.

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




# 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 generico fornitoci in partenza, il quale contiene alcune iterazioni, un singolo segnale di reset per inizializzare il modulo, una generica durata di "**START**" e una generica sequenza di bit sul segnale "**W**". Abbiamo usato questo test per verificare il corretto funzionamento del modulo in un caso generico senza condizioni particolari.

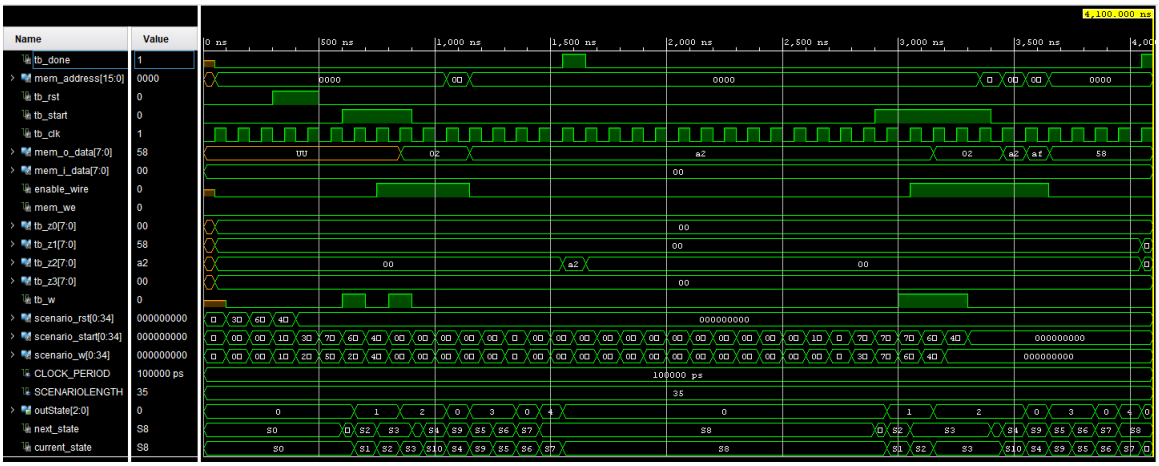



Figura 1: Test generico

### 3.2.2 TB2

Un caso particolare che abbiamo voluto testare, è quello in cui vi sia la lettura di 16 bit pari a uno consecutivi sul segnale "**W**".

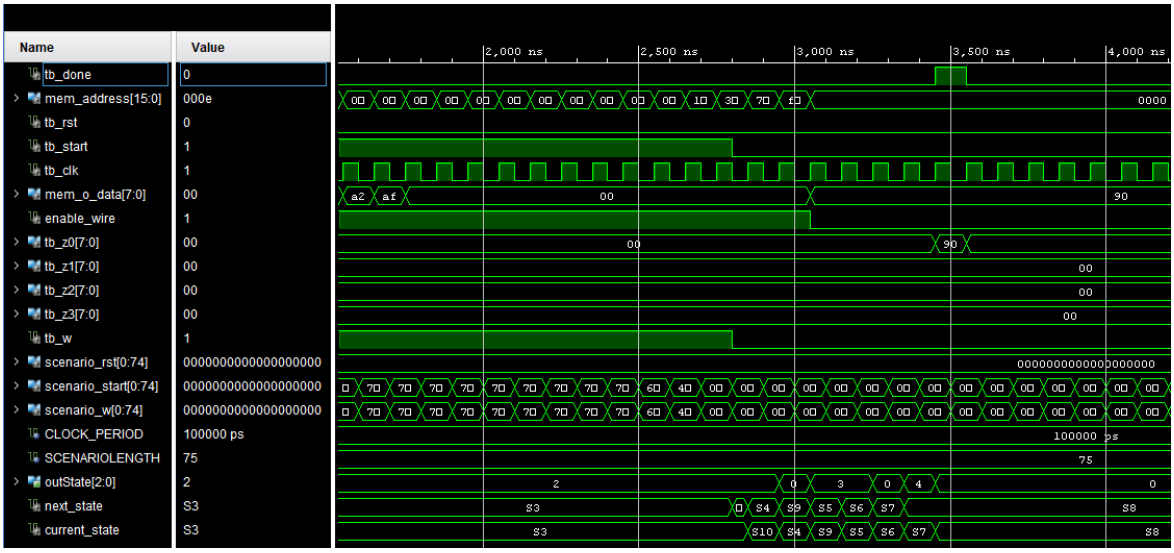



Figura 2: 16 bit a 1

### 3.2.3 TB3

Un ulteriore caso particolare che abbiamo voluto testare, è il caso in cui vi sia la lettura di due qualsiasi bit seguiti da 16 zero consecutivi sul segnale "**W**". Caso duale al TB2.

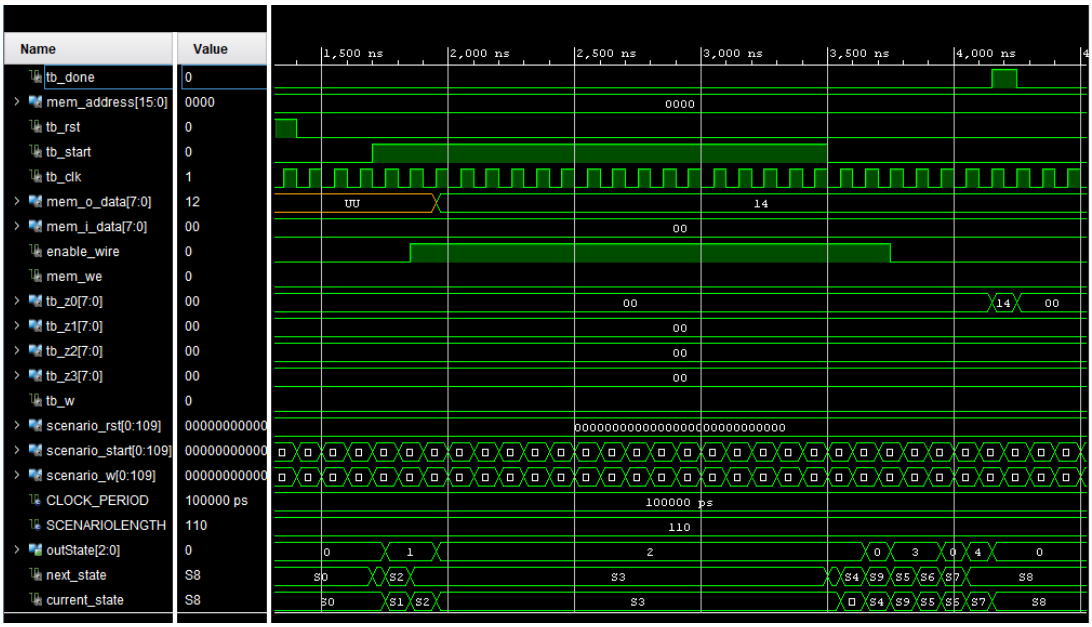



Figura 3: 16 bit a 0

### 3.2.4 TB4

Un quarto test, lo abbiamo fatto per verificare che il comportamento del modulo HW rimanga coerente nel corso del tempo, superando un notevole numero di iterazioni consecutive, e che quindi il comportamento delle uscite (ovvero il mantenimento del valore precedentemente mostrato) fosse corretto resettando di tanto in tanto il modulo tramite segnali di reset.


Figura 4: 300 iterazioni consecutive

# 4 Conclusioni

In conclusione, alla luce dei risultati ottenuti dai test effettuati (tutti con successo, sia in behavioural che in post sintesi), il modulo sembrerebbe funzionare correttamente, come da specifica. Dal punto di vista dell'ottimizzazione, abbiamo deciso, al momento della progettazione della FSM, di utilizzare un quantitativo di stati superiore a quello effettivamente necessario per tutelarci in caso di eventuali tempi di propagazione dei segnali e delle porte nella realtà. Ad esempio, abbiamo deciso di introdurre in fase di progettazione gli stati "**S10**", "**S4**", "**S9**", "**S8**" utilizzati rispettivamente: i primi due per avere la certezza che il sottomodulo *OutAddr* fosse in grado di completare i suoi compiti anche considerando eventuali ritardi (in particolare il primo mantiene attivo il sottomodulo, mentre il secondo è usato più come intermezzo tra l'operazione di lettura degli indirizzi di uscita e di memoria e la memorizzazione del dato nel registro selezionato), il terzo per avere la certezza che il sottomodulo *RegistryUpdater* avesse il tempo di completare i suoi compiti anche considerando eventuali ritardi, ed infine il quarto per semplice comodità in fase di progettazione della *FSM* (infatti quest'ultimo è facilmente rimovibile collegando direttamente "**S7**" a "**S1**" quando viene letto 1 sull'ingresso della *FSM*, ovvero su "**START**"). Concludendo quindi,

con molta probabilità questi stati potrebbero essere rimossi per velocizzare la generazione del segnale "**DONE**", ma siccome da specifica è richiesto che tale segnale venga generato entro 20 cicli di clock, abbiamo deciso di mantenerli siccome la specifica dataci è comunque rispettata (infatti il segnale di "**DONE**" viene generato a metà del settimo ciclo di clock e termina a metà dell'ottavo).