Progetto di Reti Logiche

Prof. Fabio Salice

Paolo Gennaro
Codice Persona: 10766315
Matricola : 961072

Anno Accademico 2022/2023

POLITECNICO
MILANO 1863

Indice

1 Introduzione

1.1 Obiettivo

Questo progetto ha per obiettivo la realizzazione di un modulo hardware in VHDL che si interfacci con una memoria per indirizzare il contenuto di una sua locazione in uno dei quattro canali di uscita disponibili.

1.2 Specifica

Il modulo ha due ingressi primari da 1 bit (**i_w** e **i_start**) e cinque uscite primarie, di cui quattro da 8 bit (**o_z0, o_z1, o_z2 e o_z3**) e una da 1 bit (**o_done**). Il segnale di clock e il segnale di reset sono unici per tutto il sistema.
All'istante iniziale, cioè quello relativo al reset del sistema, le uscite **o_z0, o_z1, o_z2 e o_z3** devono valere "0000 0000", mentre **o_done** deve valere "0".

Il modulo funziona attraverso l'ingresso seriale **i_w**, organizzato come sequenza di bit che contiene 2 bit di intestazione seguiti da N bit di indirizzo della memoria (al massimo 16). Gli N bit permettono di costruire un indirizzo di memoria in cui è memorizzato il messaggio da 8 bit che deve essere indirizzato verso un canale di uscita. Se il numero di bit di N è inferiore a 16, l'indirizzo viene esteso con degli "0" sui bit più significativi.

I due bit di intestazione identificano il canale d'uscita (quindi **o_z0, o_z1, o_z2 o o_z3**) sul quale deve essere indirizzato il messaggio. Il primo bit è il bit più significativo del canale di uscita, il secondo quello meno significativo.

Il segnale **i_start** segnala l'inizio della sequenza di ingresso quando il suo valore è 1, e può rimanere alto per al massimo 18 cicli di clock e per non meno di 2 cicli di clock, e ne segnala la fine quando il suo valore è "0". Durante la scrittura del messaggio sul canale, il segnale **o_done** passa da "0" a "1" e rimane attivo per un solo ciclo di clock. Quando **o_done** è basso tutte le uscite primarie ad 8 bit hanno valore "0000 0000". Inoltre, il modulo deve anche tenere traccia delle precedenti esecuzioni.In altre parole, quando **o_done** è alto il canale associato al messaggio cambia il suo valore, mentre gli altri canali mostreranno l'ultimo valore trasmesso derivato dai precedenti messaggi.

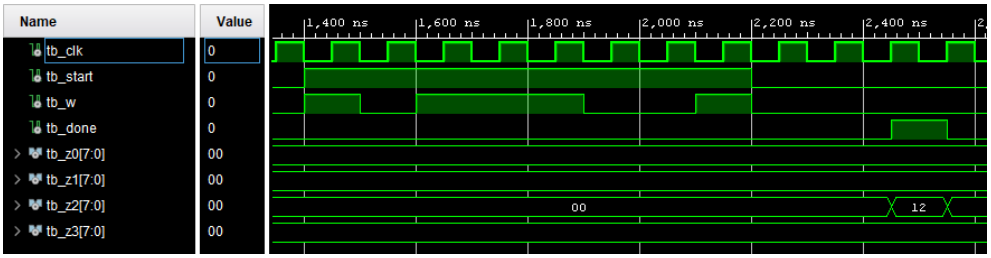


Figure 1: Esempio di diagramma temporale

1.3 Entity del componente

È stata fornita per il progetto 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_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;
```

Listing 1: Entity del progetto fornita dal docente

In particolare:

- **i_clk** è il segnale di CLOCK in ingresso generato dal Test Bench;
- **i_rst** è il segnale di RESET che inizializza la macchina pronta per ricevere il primo segnale di START;
- **i_start** è il segnale di START generato dal Test Bench;
- **i_w** è il segnale W precedentemente descritto e generato dal Test Bench;
- **o_z0, o_z1, o_z2, o_z3** sono i quattro canali di uscita;
- **o_done** è il segnale di uscita che comunica la fine dell'elaborazione;
- **o_mem_addr** è il segnale (vettore) di uscita che manda l'indirizzo alla memoria;
- **i_mem_data** è il segnale (vettore) che arriva dalla memoria in seguito ad una richiesta di lettura;
- **o_mem_en** è il segnale di ENABLE da dover mandare alla memoria per poter comunicare (sia in lettura che in scrittura);
- **o_mem_we** è il segnale di WRITE ENABLE da dover mandare alla memoria (=1) per poter scriverci. Per leggere da memoria esso deve essere 0.

2 Architettura

Il modulo è stato realizzato partendo dalla rappresentazione di un datapath e di un FSM, che hanno reso più facile la descrizione in VHDL.

2.1 Datapath

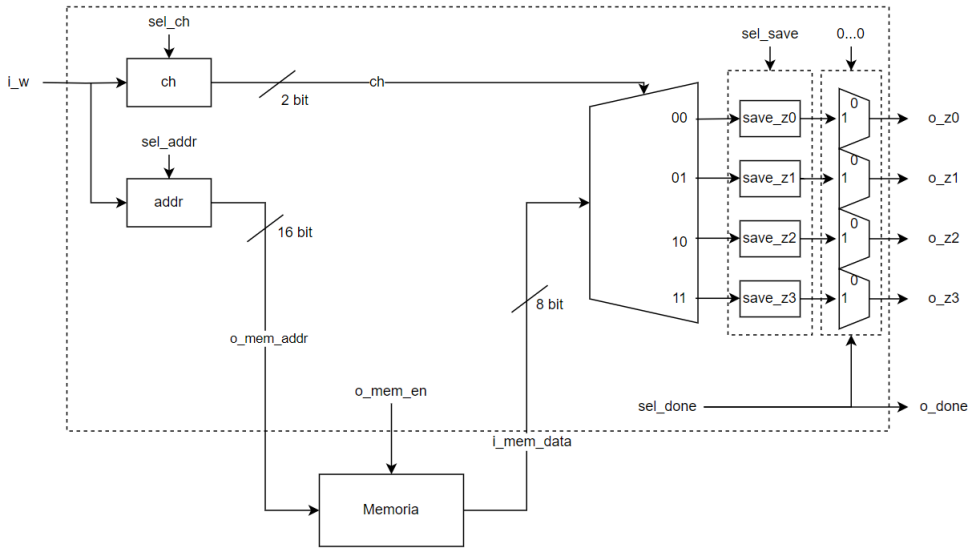


Figure 2: Datapath del modulo (semplificato)

Il modulo è composto da 14 processi: 6 di questi si occupano della gestione dei registri mostrati nel datapath, mentre i restanti processi implementano il funzionamento della macchina stati e delle configurazioni dei segnali di controllo ad essa associati.

2.1.1 Utilizzo e Definizione dei Registri

Definiamo adesso il comportamento dei registri mostrati nel datapath a seconda delle varie casistiche.

ch: è un registro a 2 bit che viene inizializzato a "00" e che viene gestito da un segnale di controllo: **sel_ch**. Lo scopo del registro è quello di salvare il valore del canale di uscita in cui andremo a mostrare il messaggio ricevuto dalla memoria.
Quando **i_start** è alto e **sel_ch** vale "01", il modulo scrive nel MSB del registro il valore di **i_w**, mentre se **sel_ch** vale "10", il modulo scrive nel LSB del registro il valore di **i_w**.

addr: è un registro a 16 bit che viene inizializzato a "0..0", e che viene gestito da un segnale di controllo: **sel_addr**. Lo scopo del registro è quello di salvare l'indirizzo di memoria in cui è salvato il messaggio che poi andrà mostrato in uno dei canali di uscita.
Quando i_start è alto e **sel_addr** vale "1", il modulo scrive nel LSB del registro il valore di **i_w**; se **i_start** rimane alto allora viene eseguito un left shift sul registro e viene inserito il nuovo valore di **i_w**; questo procedimento avviene in loop fino a quando **i_start** non torna ad essere "0".

Registri di save: sono 4 registri da 8 bit che vengono inizializzati a "0..0", e che vengono gestiti da un segnale di controllo: **sel_save**. Lo scopo di questi registri è quello di salvare il contenuto dei messaggi ricevuti dalla memoria.
Quando **sel_save** vale "1" il modulo scrive il contenuto del messaggio ricevuto dalla memoria in uno dei quattro registri; il registro in cui il valore viene salvato dipende dal valore contenuto nel registro ch (save_z0 se ch = "00"; save_z1 se ch = "01"; save_z2 se ch = "10"; save_z3 se ch = "11").

2.1.2 Processi di gestione delle uscite

Per la gestione delle uscite ho creato 5 processi (1 per ogni canale di uscita ed 1 per o_done) gestiti da un unico segnale di controllo: **sel_done**.
Quando **sel_done** viene impostato ad 1 dalla FSM il valore di **o_done** si alza e vengono mostrati tutti i messaggi salvati nei registri; quando invece **sel_done** torna basso, **o_done** si abbassa e tutte le uscite mostreranno "0000 0000".

2.2 Macchina a Stati

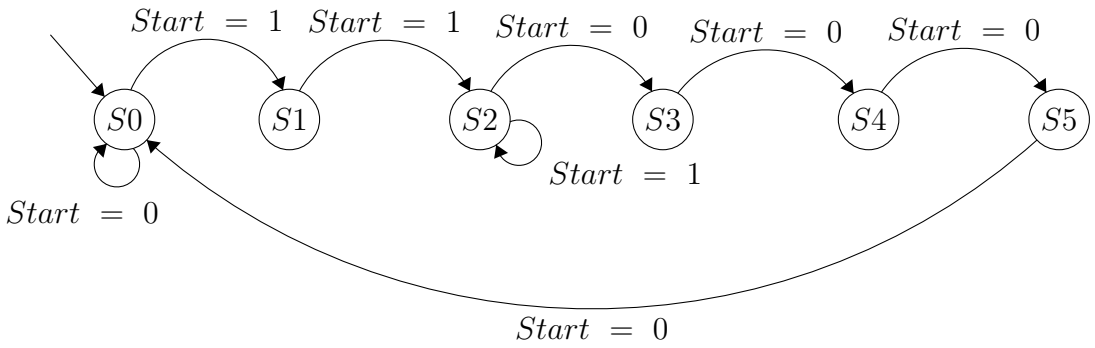


Figure 3: FSM del Progetto

La macchina a stati gestisce i segnali di controllo dell'intero circuito, e ne rende facile la comprensione. È definita da 3 processi:

1. il primo processo (**state_reg**) è sensibile sia al reset che al clock e definisce il comportamento della macchina nel caso in cui uno dei due segnali diventi alto;

2. il secondo processo (**nextstate_FSM**) è sensibile allo stato corrente ed a **i_start**, e definisce il passaggio tra gli stati della macchina;

3. il terzo processo (**output_FSM**) è sensibile soltanto allo stato corrente e si occupa della modifica dei selettori di controllo per ogni stato.

2.2.1 Output FSM

Di seguito il codice che definisce la configurazione dei segnali di controllo per ogni stato:

```
case current_state is
    when S0 =>
        sel_ch <= "01";
    when S1 =>
        sel_ch <= "10";
    when S2 =>
        sel_ch <= "00";
        sel_addr <= '1';
    when S3 =>
        sel_addr <= '0';
        o_mem_en <= '1';
    when S4 =>
        o_mem_en <= '0';
        sel_save <= '1';
    when S5 =>
        sel_save <= '0';
        sel_done <= '1';
 end case;
```

Listing 2: Righe di codice estrapolate dal output_FSM process

3 Risultati sperimentali

3.1 Sintesi

Il modulo progettato è sintetizzabile utilizzando l'FPGA xc7a200tfbg484-1.

Report Utilizzation:

Il modulo ha richiesto l'uso di 53 Flip Flop per l'implementazione dei registri e 0 Latch.

1. Slice Logic

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	35	0	134600	0.03
LUT as Logic	35	0	134600	0.03
LUT as Memory	0	0	46200	0.00
Slice Registers	53	0	269200	0.02
Register as Flip Flop	53	0	269200	0.02
Register as Latch	0	0	269200	0.00
F7 Muxes	0	0	67300	0.00
F8 Muxes	0	0	33650	0.00

Figure 4: report_utilizzation da terminale

Report Timing:

Il modulo soddisfa i "constraints" imposti dalla specifica del progetto. Questo è riscontrabile dalla figura seguente, che riporta le prime righe del comando "report_timing" che evidenziano un valore positivo di Slack(MET).

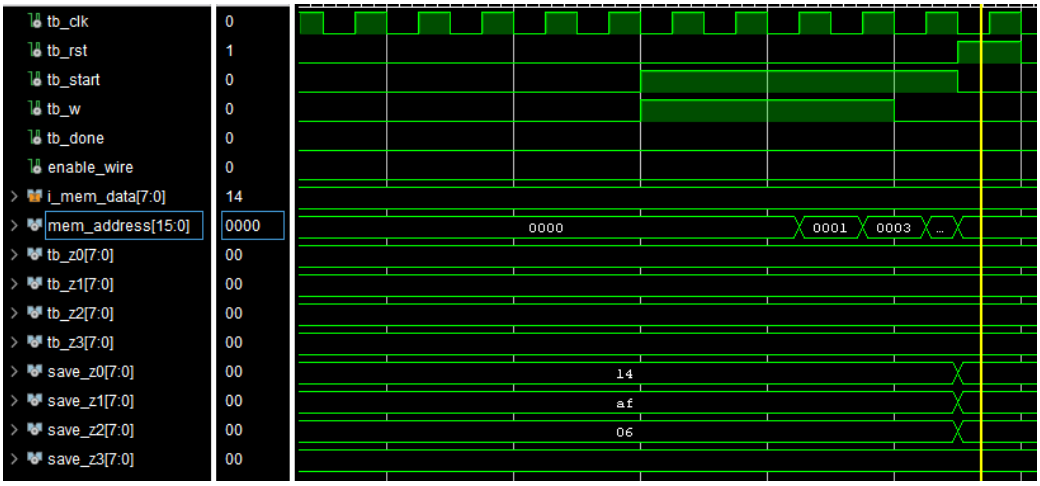
```
------------------------------------------------------------------------------------
| Tool Version : Vivado v.2018.3.1 (win64) Build 2489853 Tue Mar 26 04:20:25 MDT 2019
| Date         : Sat May 13 17:14:50 2023
| Host         : vwresxaas06 running 64-bit major release  (build 9200)
| Command      : report_timing
| Design       : project_reti_logiche
| Device       : 7a200t-fbv484
| Speed File   : -1  PRODUCTION 1.23 2018-06-13
------------------------------------------------------------------------------------

Timing Report

Slack (MET) :             97.528ns  (required time - arrival time)
  Source:                 FSM_sequential_current_state_reg[2]/C
                            (rising edge-triggered cell FDCE clocked by clock  {rise@0.000ns fall@5.000ns period=100.000ns})
  Destination:            save_z0_reg[0]/CE
                            (rising edge-triggered cell FDCE clocked by clock  {rise@0.000ns fall@5.000ns period=100.000ns})
  Path Group:             clock
  Path Type:              Setup (Max at Slow Process Corner)
  Requirement:            100.000ns  (clock rise@100.000ns - clock rise@0.000ns)
```

Figure 5: report_timing da terminale

3.2 Simulazioni

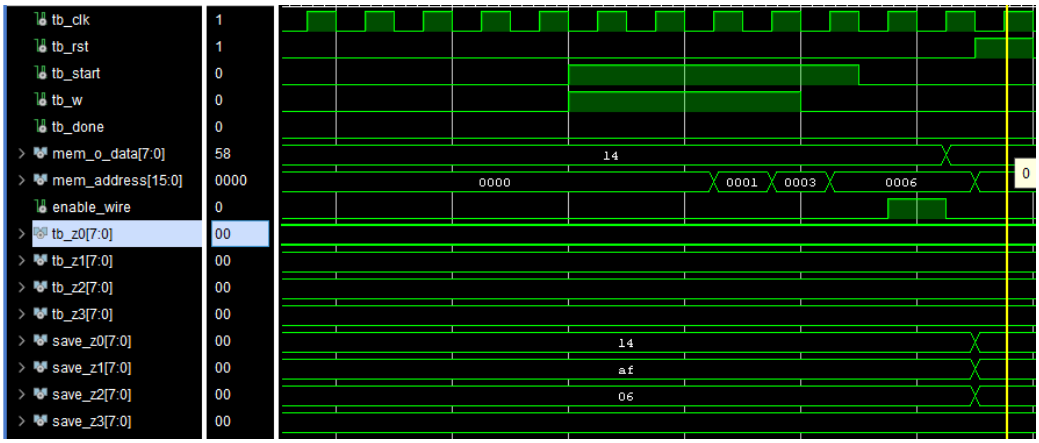
Il modello è stato testato su diversi Test Bench sia in **"Behavioral Simulation"** sia in **"Post-Synthesis Functional Simulation"**.
I primi test eseguiti sono stati quelli forniti dal docente, ne sono stati effettuati anche altri per testare il modulo anche in casi "critici".
Si riportano ora degli esempi di output di alcuni dei Test effettuati:

Reset subito dopo la discesa di i_start



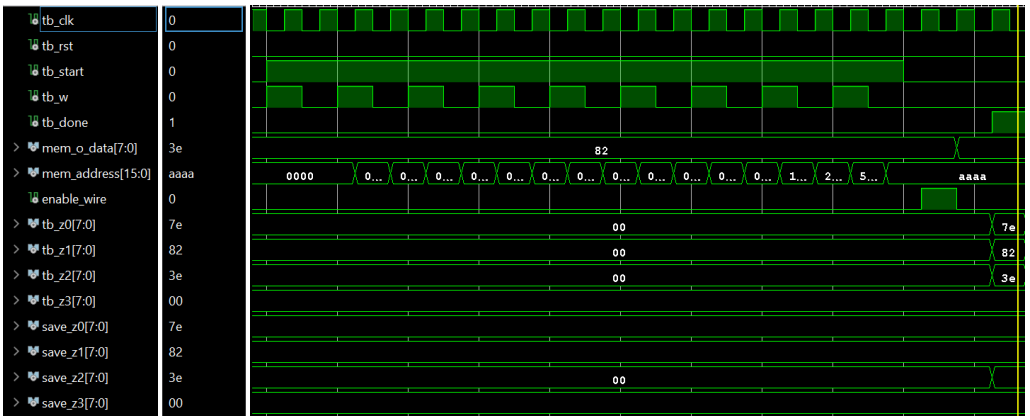
Letto il reset = "1", il modulo deve tornare nello stato di partenza, non portando a 1 il segnale di **o_mem_enable** e assegnando il valore "0000 0000" a tutti i **"registri di save"**. Il modulo ha funzionato correttamente.

Reset prima di o_done alto



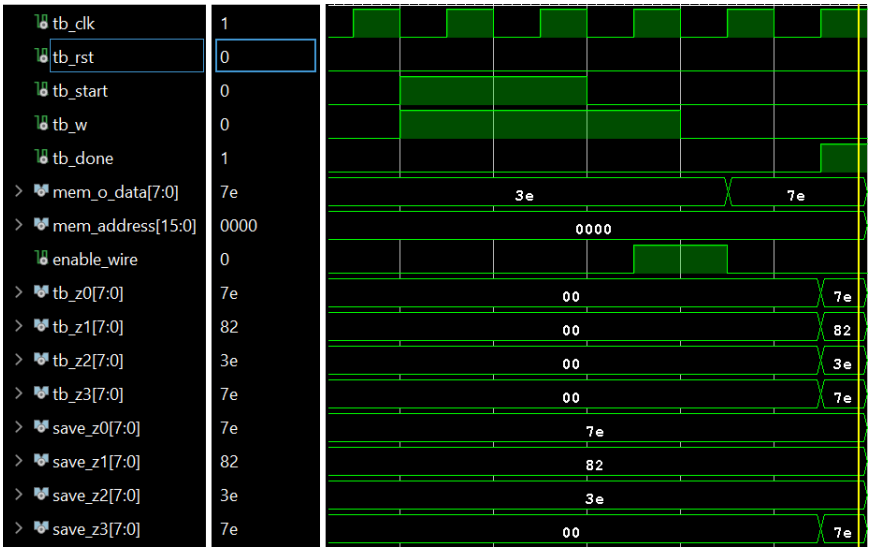
Il reset si alza dopo aver ricevuto il messaggio dalla memoria, quindi deve tornare allo stato di partenza; **o_done** non deve alzarsi e non devono essere mostrati nelle uscite i contenuti dei registri, ma deve essere assegnato il valore "0000 0000" a tutti i **"registri di save"**. Il modulo ha funzionato correttamente.

i_w alternato e i_start alto per 18 cicli di clock



In questo Test si prova la macchina nel caso in cui **i_start** resti alto il più a lungo possibile, cioè 18 cicli di clock da specifica. **i_w** invece è stato volutamente scritto in maniera alternata per mostrare il corretto funzionamento dei registri che si occupano del canale di intestazione e dell'indirizzo di memoria.

i_start alto per 2 cicli di clock e i_w alto per 3 cicli di clock



In questo Test si prova la macchina nel caso in cui **i_start** resti alto per il minore tempo possibile, cioè 2 cicli di clock da specifica. **i_w** è alto per 3 cicli di clock e 2 di questi contemporaneamente ad **i_start**. Anche in questo caso la macchina funziona correttamente.

4 Conclusioni

A seguito dello studio affrontato durante il corso di Reti Logiche, ho trovato molto interessante la stesura del codice in VHDL di questo progetto e la possibilità di applicare quanto studiato nella realizzazione del modulo hardware. Il primo approccio con Vivado e con VHDL non è stato semplice per la difficoltà del passare dalla programmazione software alla descrizione hardware. Partire da un datapath e dalla descrizione di un FSM mi ha permesso di scrivere il codice in quanto in VHDL più agevolmente.

Dopo aver finito di scrivere il codice, e dopo avere risolto qualche errore di descrizione, sono riuscito a fare funzionare il modulo in **Behavioral Simulation** e in **Post Synthesis Functional Simulation** senza grandi difficoltà. Non soddisfatto di alcune scelte implementative e nonostante il progetto fosse già funzionante, ho deciso di ottimizzare il numero di stati del FSM iniziale passando da 7 a 6 stati. In questo modo, dopo ulteriori modifiche, ho potuto diminuire anche il numero di Flip Flop utilizzati.