PROVA FINALE DI RETI LOGICHE

Prof. Salice Fabio – Anno 2022/2023

Andrea Giangrande

Codice Persona: 10743167

Matricola: 958433

INDICE:

POLITECNICO MILANO 1863

1. INTRODUZIONE

Obbiettivo:

Lo scopo del Progetto è la realizzazione di un componente hardware in VHDL che interagisca con una memoria e indirizzi il contenuto di una sua specifica cella in un predeterminato canale di uscita.

Specifica:

Il modulo presenta cinque ingressi e cinque uscite. Gli ingressi primari sono due, *i_w* e *i_start*, entrambi composti da un singolo bit. Gli altri ingressi sono i segnali di clock e reset, *i_clk* e *i_rst*, unici per tutto il sistema, e *i_mem_data*, segnale che arriva dalla memoria, composto da 8 bit e contenente il dato da mostrare in uscita. Le uscite comprendono i quattro canali di uscita, su cui vengono mostrati i relativi dati, composti da 8 bit ciascuno e denominati *o_z0, o_z1, o_z2, o_z3*. Gli altri segnali di uscita sono *o_done*, che comunica la fine dell'elaborazione, ed *o_mem_addr*, che comunica alla memoria l'indirizzo della cella da cui estrarre il dato che si cerca.

Il componente dovrà gestire i segnali come segue: quando *i_start* vale "1", dovrà leggere sincronicamente con il clock il valore di *i_w*. Tramite i valori raccolti deve individuare il canale di uscita corretto indicato dai primi due bit utili* letti, e l'indirizzo di memoria dai successivi bit. Una volta comunicato con la memoria, per un solo ciclo di clock il valore di *o_done* dovrà valere "1", e nello stesso ciclo di clock verrà mostrato sul corretto canale di uscita il dato appena letto da memoria; sugli altri canali dovrà invece essere mostrato l'ultimo valore contenuto durante il precedente ciclo di clock in cui *o_done* era alto. Se *o_done* non è

alto allora i canali di uscita devono contenere “0000 0000”. Se viene letto in ingresso il segnale di reset alto, allora nel successivo fronte di salita del segnale *o_done* i canali di uscita non dovranno più tenere conto dei valori contenuti prima del segnale di reset. (*Nota: con bit “utili” si intendono i bit letti durante i cicli di clock in cui *i_start* è alto)

Entity:

L’interfaccia del componente è stata fornita direttamente dal docente ed è la seguente:

```
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;
```

Comportamento dei segnali:

Viene ora spiegato come si comportano i vari segnali che fanno parte dell’ interfaccia del componente.

i_clk è il segnale di clock, generato dal TestBench.

i_rst è il segnale di reset che inizializza la macchina: quando viene letto in ingresso *i_rst* con valore “1” allora nel seguente ciclo di clock in cui il segnale *o_done* ha valore “1”, le uscite diverse da quelle specificate dai primi due bit utili di *i_w* devono mostrare “0000 0000”.

i_start è il segnale generato dal TestBench che, quando il suo valore è “1”, segnala l’inizio della sequenza di bit in ingresso da leggere. Può rimanere alto per un minimo di 2 cicli di clock fino ad un massimo di 18 cicli di clock.

i_w è il segnale in ingresso generato dal TestBench che viene letto quando *i_start* è alto. La sequenza di bit generata da *i_w* costituisce per i primi due bit la codifica del canale di uscita su cui mostrare il dato e dal terzo bit in poi (che potrebbe non essere già letto) costituisce l’indirizzo della memoria di cui si vuole leggere il dato da mostrare in uscita.

o_z0, o_z1, o_z2, o_z3 sono i quattro canali di uscita: devono mostrare “0000 0000” fintanto che *o_done* ha valore “0”.

o_done è il segnale che comunica la fine dell’elaborazione: deve valere “0” fino a quando non si ha il corretto dato da mostrare disponibile. Varrà “1” per un solo ciclo di clock.

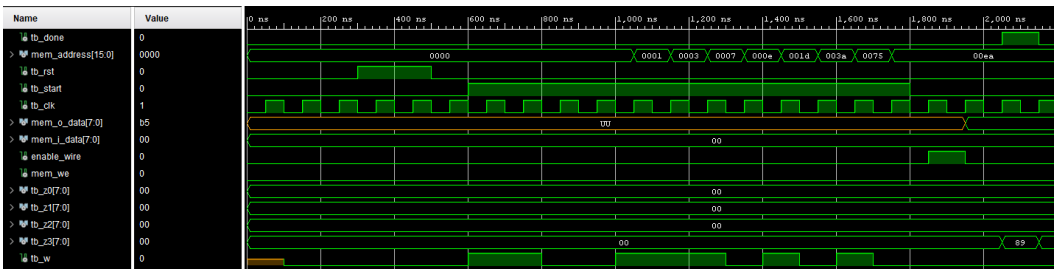
o_mem_addr è il segnale di 16 bit che contiene l’indirizzo della memoria da cui si vuole leggere il dato.

i_mem_data è il segnale di 8 bit che arriva dalla memoria dopo una richiesta di lettura.

o_mem_en è il segnale da mandare alla memoria per poter comunicare (leggere e/o scrivere del contenuto) con essa.

o_mem_we è il segnale da mandare alla memoria per poter scrivere su di essa. Nel modulo esso non verrà mai considerato in quanto la scrittura in memoria non è necessaria ai fini del progetto.

Figura 1: Esempio di funzionamento



2. ARCHITETTURA

Per realizzare il progetto sono stati definiti come prima cosa lo schema di un **datapath** e di una **macchina a stati**. Di seguito i dettagli.

1. Datapath:

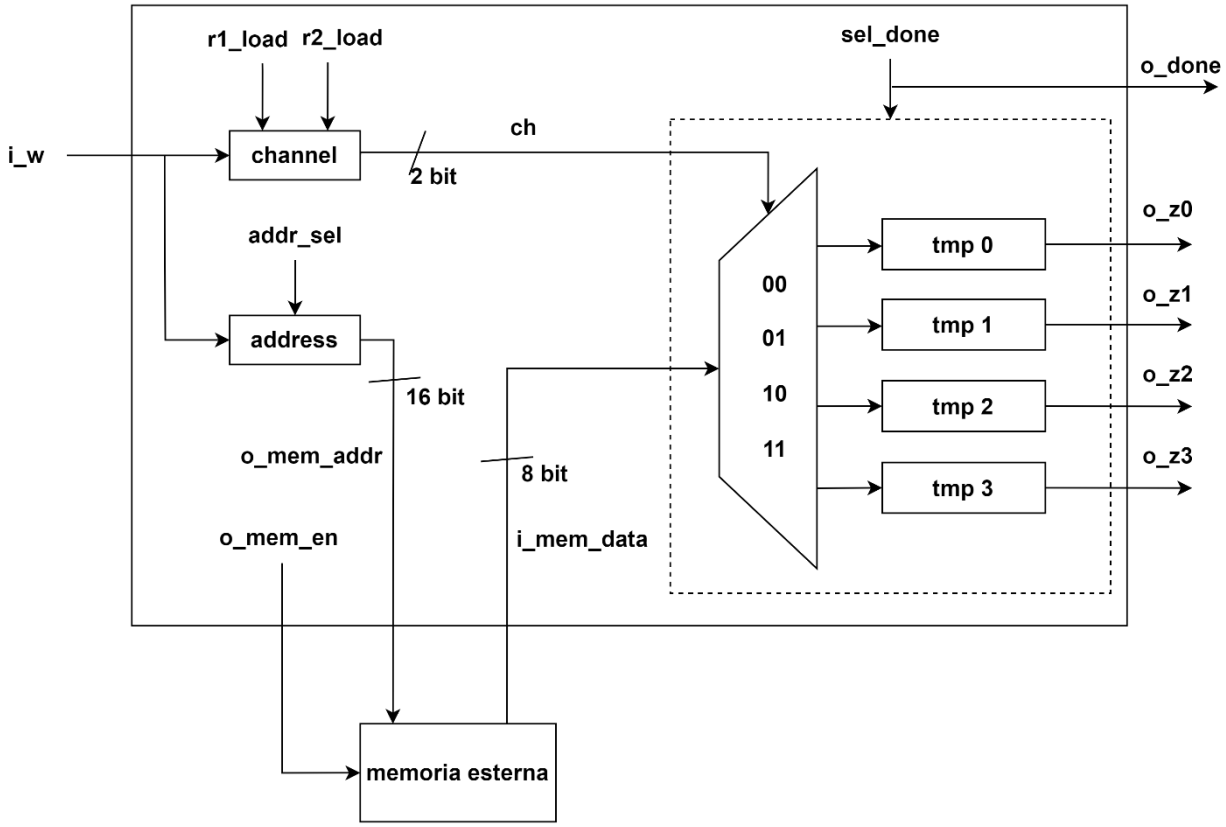


Figura 2: datapath (semplificato) del modulo

Il datapath mostrato contiene i seguenti registri:

- *channel* (nel codice rinominato *ch* per brevità) contiene i due bit di prefisso che indicano il canale di uscita corrispondente. In questo registro vengono salvati i primi due bit utili di *i_w* ed è controllato dai selettori *r1_load* e *r2_load*, che permettono l'assegnamento dei bit al MSB ed al LSB di *ch*.
- *address* (nel codice rinominato *addr* per brevità) è inizializzato con 16 bit di valore "0" e contiene alla fine dell'elaborazione l'indirizzo della memoria di cui si vuole leggere il contenuto. È controllato dal selettore *addr_sel*; in particolare quando

addr_sel vale “1” il valore di *i_w* viene inserito nel LSB del registro e il resto del contenuto viene shiftato a sinistra.

- I registri *tmp0, tmp1, tmp2, tmp3* contengono i valori da mostrare in uscita sui corrispondenti canali. L’assegnamento di *i_mem_data* al registro corretto è garantito da un MUX, direttamente dipendente dal registro del canale, *ch*.

Le uscite sono controllate dal selettore *sel_done*. Esso sarà alto per un solo ciclo di clock, nel quale alle uscite *o_z0, …, o_z3* sono assegnati i valori di *tmp0, …, tmp3,* e, nello stesso ciclo di clock, il segnale di *o_done* è posto a “1”.

2. Macchina a stati:

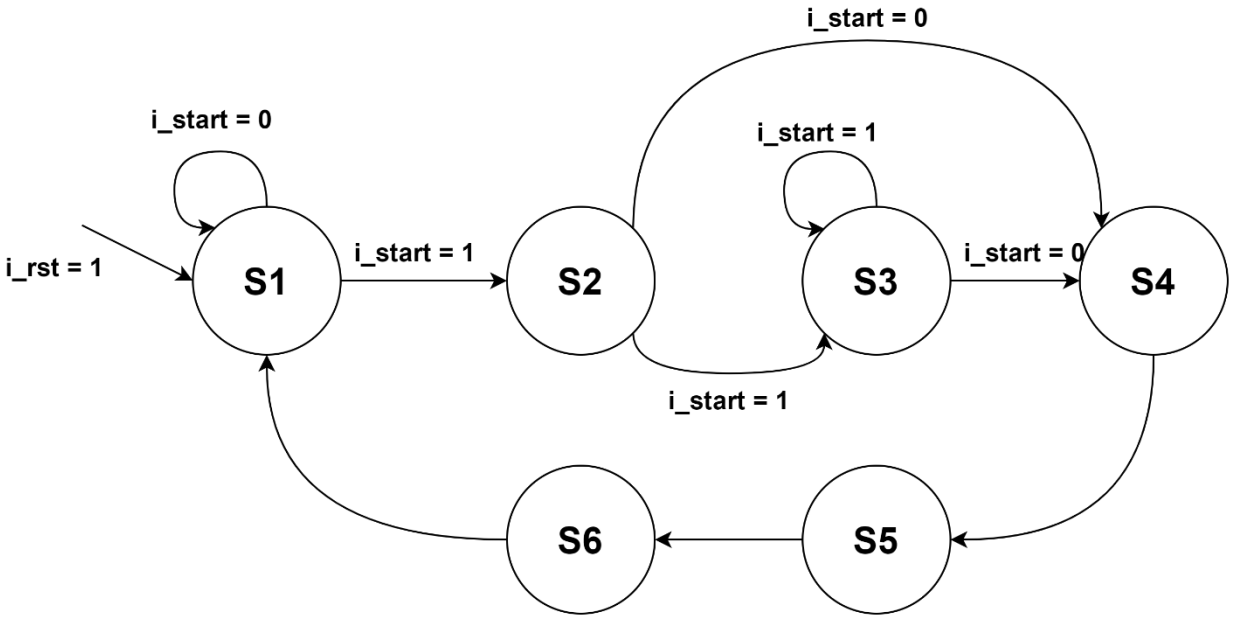


Figura 3: macchina a stati

La macchina a stati sviluppata per il progetto è composta da sei stati. In dettaglio il comportamento di ogni stato:

Stato **S1**:

Stato iniziale della macchina; ogni volta che viene letto il segnale di reset alto, la macchina è riportata in questo stato. Inizializza tutti i segnali a "0" e rimane in attesa che *i_start* sia alto; quando avviene, legge il primo bit utile di *i_w* e lo salva nel registro *ch(1)*.

Stato **S2**:

Legge il secondo bit utile di *i_w* e lo salva nel registro *ch(0)*. Se *i_start* è ancora alto, il prossimo stato sarà **S3**, altrimenti **S4**.

Stato **S3**:

Se *i_start* resta alto per meno di tre cicli di clock, questo stato viene saltato. Altrimenti legge il valore di *i_w*, lo salva nel registro *addr(0)* e shifta a sinistra tutti gli altri bit di quest'ultimo registro. Continua questa operazione fintanto che *i_start* rimane alto (garantito per un massimo di 16 cicli di clock), poi il prossimo stato sarà **S4**.

Stato **S4**:

Questo stato abilita la lettura in memoria portando a "1" l'uscita *o_mem_en*.

Stato **S5**:

In questo stato è letto in ingresso il valore di *i_mem_data* ed è salvato nei registro corrispondente tra *tmp0, …, tmp3*.

Stato **S6**:

Stato finale; il segnale *o_done* è portato a "1" e i valori contenuti nei registri *tmp0, …, tmp3* sono mostrati nelle corrispondenti uscite *o_z0, …, o_z3*. Nel ciclo di clock successivo lo stato corrente sarà **S1**.

3. IMPLEMENTAZIONE E OTTIMIZZAZIONE

Per quanto riguarda la macchina a stati, la prima bozza del progetto conteneva uno stato in più, che serviva per inizializzare i registri dopo il reset. Il progetto risultava funzionante, ma è stato subito notata la possibilità di rimuovere lo stato "fondendolo" con l'attuale stato di reset **S1**, che inizialmente era stato sviluppato con il solo scopo di leggere il primo bit utile dal segnale *i_w*.

Per quanto riguarda il codice del datapath, inizialmente era stato pensato di gestire tutto con un singolo processo. Per questioni di leggibilità e adattabilità del codice si è preferito invece dividere in molteplici processi distinti. Al fine di debugging la scelta si è rivelata conveniente. Una versione contenente un solo processo verrà probabilmente e implementata in seguito a solo scopo di fare pratica, ma non sarà inclusa nella versione finale del progetto.

La versione finale del codice VHDL contiene sette processi, di cui tre per la gestione della macchina a stati e i restanti per la gestione dei segnali e delle uscite. Nella architettura è anche presente del codice non contenuto all'interno di processi. Tale scelta è stata fatta con lo scopo di scrivere del codice più leggibile, semplice e comprensibile possibile.

4. TEST E RISULTATI

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

È stato testato tramite la versione di Vivado 2018, ed ha superato con successo tutti i test che sono stati fatti, nelle modalità "*Behavioral Simulation*" e "*Post-Synthesis Functional Simulation*".

I test sono stati effettuati utilizzando tutti i sette TestBench forniti dai docenti su Webeep. Inoltre, sono state create appositamente altre dieci TestBench al fine di testare il comportamento del modulo progettato nel modo più completo possibile.

Risultati di maggior interesse:

- Report Utilization:
  La sintesi di Vivado (v 2018) ha richiesto l'utilizzo di 56 Flip-Flop e 0 Latch (Figura 4).
- Report Timing:
  Il modulo progettato soddisfa ampiamente i requisiti di tempo: a fronte di un limite superiore di 100ns, la sintesi di Vivado (v 2018) ha un tempo di circa 3ns (Figura 5).
- Test di casi particolari:
  Il modulo ha superato con successo simulazioni contenenti casi particolari che potrebbero verificarsi. In particolare, il caso in cui *i_start* diventi alto subito dopo il reset è di particolare interesse in quanto ha inizialmente causato bug e problematiche, ma è stato in fine risolto e gestito con successo. A seguito è riportato il test effettuato sul TestBench contenente il caso particolare citato; come si può notare è stato gestito correttamente (Figure 6 e 7).

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	33	0	134600	0.02
LUT as Logic	33	0	134600	0.02
LUT as Memory	0	0	46200	0.00
Slice Registers	56	0	269200	0.02
Register as Flip Flop	56	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

Figura 4: utilizzo di Flip Flop e Latch mostrato dopo avere digitato il comando "report_utilization" nel terminale di Vivado (v 2018).

```
Slack (MET) :             97.324ns  (required time - arrival time)
```

Figura 5: Slack = (Richiesto – Tempo di esecuzione) mostrato dopo avere digitato il comando "report_timing" nel terminale di Vivado (v 2018).

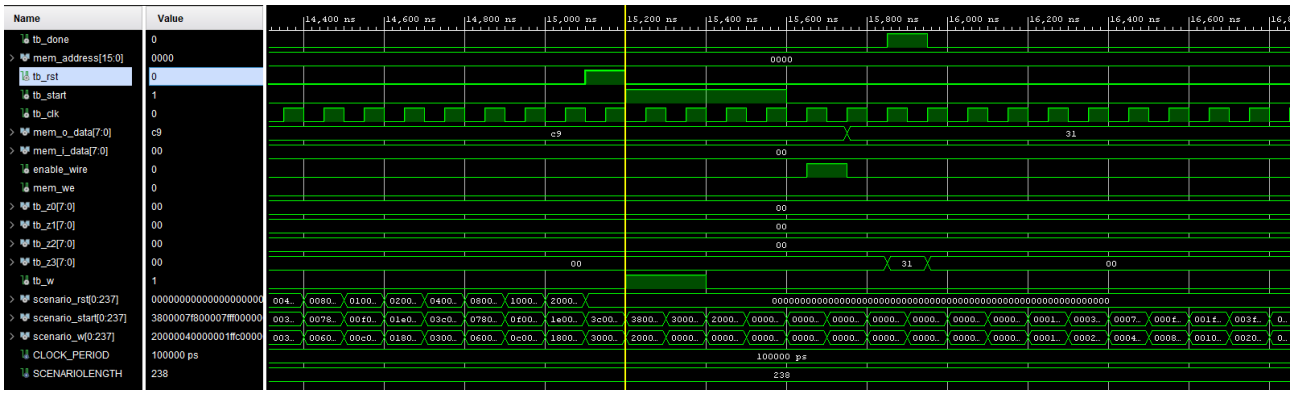


Figura 6: Test superato con successo. Il segnale *i_start* è alto subito dopo il reset e la lettura successiva è avvenuta correttamente.

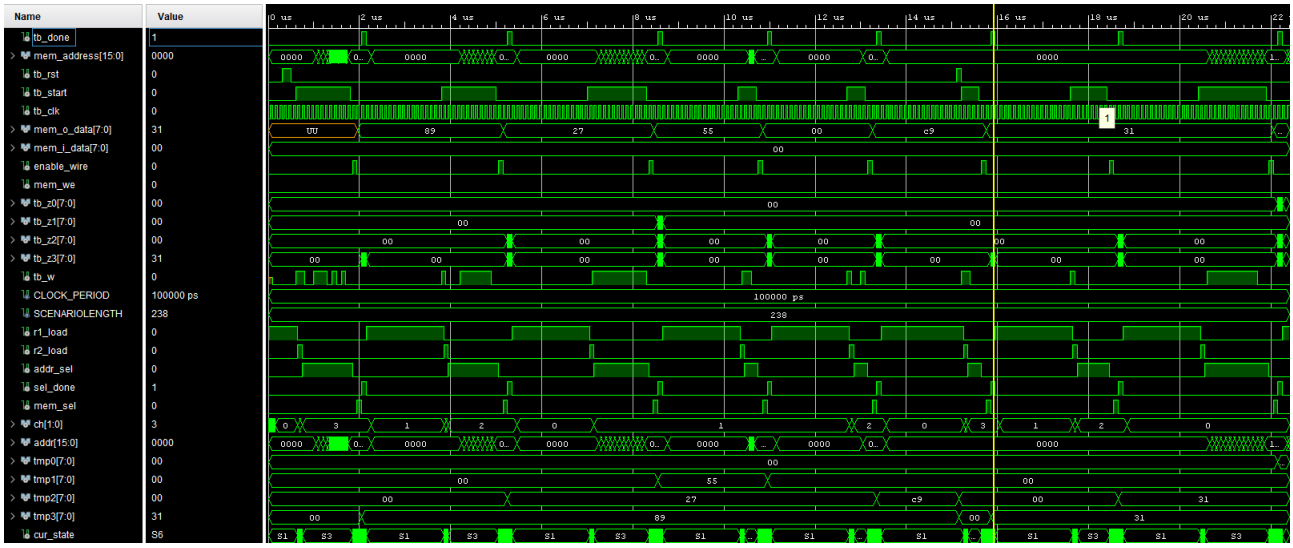


Figura 7: è riportata la simulazione del TestBench completo, mostrando anche i principali segnali del modulo implementato.

5. CONCLUSIONI

Di seguito le conclusioni tratte dallo svolgimento del progetto “Prova Finale di Reti Logiche” dell’ anno 2022/2023.

Dai test e dalle simulazioni effettuate si evince che il modulo progettato abbia raggiunto con successo gli obbiettivi e gli scopi della specifica, superando i test effettuati con successo. Ha inoltre rispettato i limiti di tempo della specifica, utilizzato un numero ragionevole di componenti hardware (flip-flop) e superato test specifici comprendenti casi di particolare interesse.

Futuri sviluppi potrebbero incentrarsi sull’ottimizzazione e l’efficienza del modulo hardware.

In conclusione, si ritiene di aver sviluppato un componente hardware in grado di risolvere in modo adeguato il problema proposto dalla specifica.