Prova finale di Reti Logiche

Grassi Andrea 10741092
Motti Caterina 10717568

A.A. 2022/23

Indice

1 Introduzione

1.1 Obiettivo

Lo scopo del progetto è l'implementazione di un componente hardware in VHDL che, dati in ingesso un selettore e un indirizzo di memoria, mostri sul canale d'uscita selezionato il contenuto della memoria, senza cancellare i risultati ottenuti in precedenza.

Di seguito l'interfaccia del componente:

```
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;
```

Figura 1: Entity del progetto

1.2 Specifica

Il modulo si compone di:

- START: ingresso da 1 bit che indica l'inizio e la fine della sequenza di bit valida.
- W: ingresso da 1 bit che rappresenta la sequenza di bit.
- Z0, Z1, Z2, Z3: uscite da 8 bit su cui viene indirizzato il contenuto della memoria, codificate rispettivamente da "00" "01" "10" e "11".
- DONE: uscita da 1 bit che indica la fine dell'elaborazione.
- CLOCK e RESET: segnali unici per tutto il sistema.

Si interfaccia con la memoria tramite:

- MEM ADDRESS: uscita da 16 bit che indica l'indirizzo di memoria da cui prendere il dato.
- MEM DATA: ingresso da 8 bit ricevuto in seguito all'elaborazione dell'indirizzo.
- ENABLE: uscita da 1 bit che permette la comunicazione con la memoria (sia in lettura che in scrittura).
- WRITE ENABLE: uscita da 1 bit da dover mandare alla memoria per poter scriverci.

1.3 Funzionamento

Il modulo viene inizializzato ogni volta che il segnale *i_rst = 1*, in modo asincrono rispetto al clock.

L'elaborazione inizia con *i_start = 1* e la lettura di *i_w* avviene sul fronte di salita del clock. Appena *i_start = 0* il modulo si occupa di mandare una richiesta alla memoria e di selezionare il canale di uscita verso cui indirizzare il risultato.

Quando la memoria produce un risultato viene portato *o_done = 1* e contestualmente sul canale di uscita selezionato viene scritto il messaggio, mentre sugli altri canali viene mostrato l'ultimo valore trasmesso, il tutto in un singolo ciclo di clock. In questa fase il segnale di *i_start* è garantito a restare a 0 finché il segnale di *i_done* non torna a 0.

Il modulo è progettato considerando che, prima della prima elaborazione, venga sempre dato il segnale di *i_rst = 1*, in modo da poter inizializzare tutte le uscite. Una successiva elaborazione non aspetta il segnale di reset.

Di seguito un esempio di diagramma temporale del funzionamento del modulo (fig. 2).

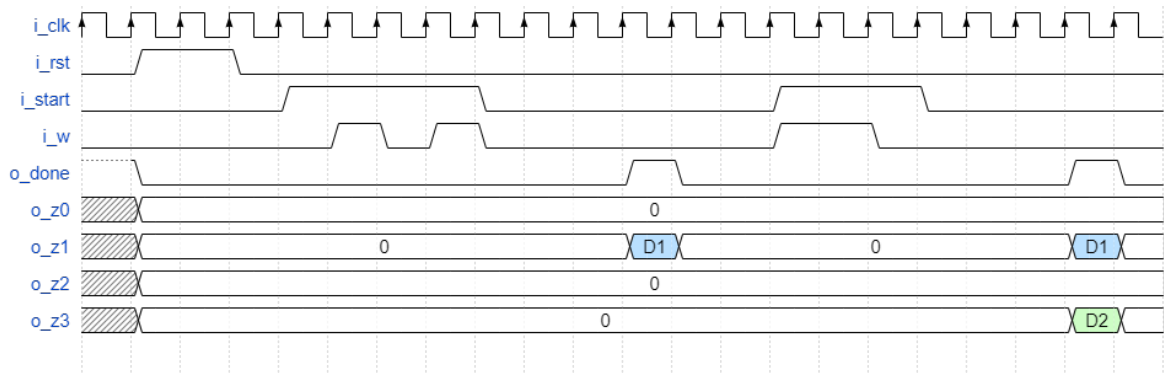

Figura 2: Diagramma temporale

Nella prima elaborazione dell'esempio l'uscita selezionata è *o_z1* codificata da "01", su cui viene mostrato D1, il contenuto dell'indirizzo di memoria "01" esteso con 0 sui bit più significativi ovvero *o_mem_addr = '0000000000000001'*.

In modo analogo nella seconda elaborazione l'uscita selezionata è *o_z3* codificata da "11", su cui viene mostrato D2 il contenuto dell'indirizzo di memoria *o_mem_addr = '0000000000000000'*.

Ad ogni elaborazione viene modificato un solo canale di output, e se tra diverse elaborazioni non viene dato il segnale di reset allora sugli altri canali vengono mostrati i risultati precedenti. Quindi in questo caso durante la seconda elaborazione viene mostrato anche il risultato precedente, oltre a quello nuovo.

2 Architettura

2.1 Scelte progettuali

L'implementazione è realizzata tramite un macchina a stati finiti che rappresenta l'algoritmo utilizzato, dalla lettura degli input fino alla scrittura in output. Il funzionamento dell'intero modulo è racchiuso in un singolo processo, in questo modo tutti i segnali vengono aggiornati in modo prevedibile e sincronizzato.

2.2 Segnali

Il processo utilizza i seguenti segnali:

- S_CURR: di tipo state_type (enumerazione degli stati) che memorizza lo stato corrente in cui si trova la FSM.
- O_Z0_NEXT, O_Z1_NEXT, O_Z2_NEXT, O_Z3_NEXT: di tipo std_logic_vector(7 downto 0), in cui vengono memorizzati gli output in modo da non perdere l'informazione delle elaborazioni precedenti.
- S_ADDR: di tipo std_logic_vector(15 downto 0), in cui viene memorizzato l'indirizzo in fase di READ.
- S_SELECT: di tipo std_logic_vector(1 downto 0), in cui viene memorizzata la codifica dell'uscita.
- FLAG: di tipo std_logic, che permette di memorizzare un solo bit di *i_w* in *s_select* durante la fase di READ.

2.3 FSM

La macchina a stati finti si compone dei seguenti stati:

- WAIT_START: è lo stato iniziale, che resta in attesa di *i_start = 1*. Si occupa di inizializzare a 0 le uscite e *s_addr* ad ogni nuova elaborazione. Sul fronte di salita del clock quando *i_start* passa da 0 a 1, *s_curr* passa allo stato successivo, quindi il primo bit di *s_select* viene memorizzato, altrimenti andrebbe perso, e inizializza *flag = 1*. In caso di reset si ritorna in questo stato.
- READ: finché *i_start = 1* si occupa di leggere l'ingresso *i_w*. Memorizza il secondo bit di *s_select* mettendo *flag = 0*. Si occupa dell'elaborazione di *s_addr*, tramite uno shift logico a sinistra e l'inserimento del bit in coda. Il *flag* è il segnale che permette di fare la distinzione dal bit che andrà salvato in *s_select* e i restanti. In questo modo l'indirizzo di memoria è pronto, inclusa l'estensione dei restanti bit a 0 grazie all'inizializzazione in WAIT_START.

  Prima di passare allo stato successivo con *i_start = 0* si occupa di impostare gli output necessari alla memoria, ovvero *o_mem_en = 1*, *o_mem_we = 0* e assegnare a *o_mem_addr* l'indirizzo appena costruito.
- WAIT_MEM: dura un solo ciclo di clock per permettere alla memoria di dare al modulo *i_mem_dato* in input. Idealmente la memoria ha bisogno di molto meno tempo di quello di un ciclo di clock per ottenere il dato, ma dato che statements come wait per ottimizzare non sono sintetizzabili abbiamo deciso di aspettare un ciclo di clock.

- ELAB: dura un solo ciclo di clock e si occupa di selezionare l'uscita simulando l'azione di un demultiplexer che usa *s_select* come selector, *i_mem_dato* come input e ha come output i vari *o_zi_next*.
- PRINT: dura un solo ciclo di clock nel quale porta *o_done = 1* e sui canali di uscita *o_zi* i valori salvati in *o_zi_next*, quindi il risultato dell'elaborazione corrente e, se presenti, delle precedenti.

Dato che il reset avviene in modo asincrono rispetto al clock non esiste uno stato di reset e lo stato di partenza è WAIT_START.

Dal diagramma della FSM (fig. 3) è possibile vedere come le transizioni tra WAIT_MEM, ELAB, PRINT e WAIT_START non dipendano da alcun parametro, per fare in modo la permanenza in ognuno di questi duri esattamente un solo ciclo di clock.

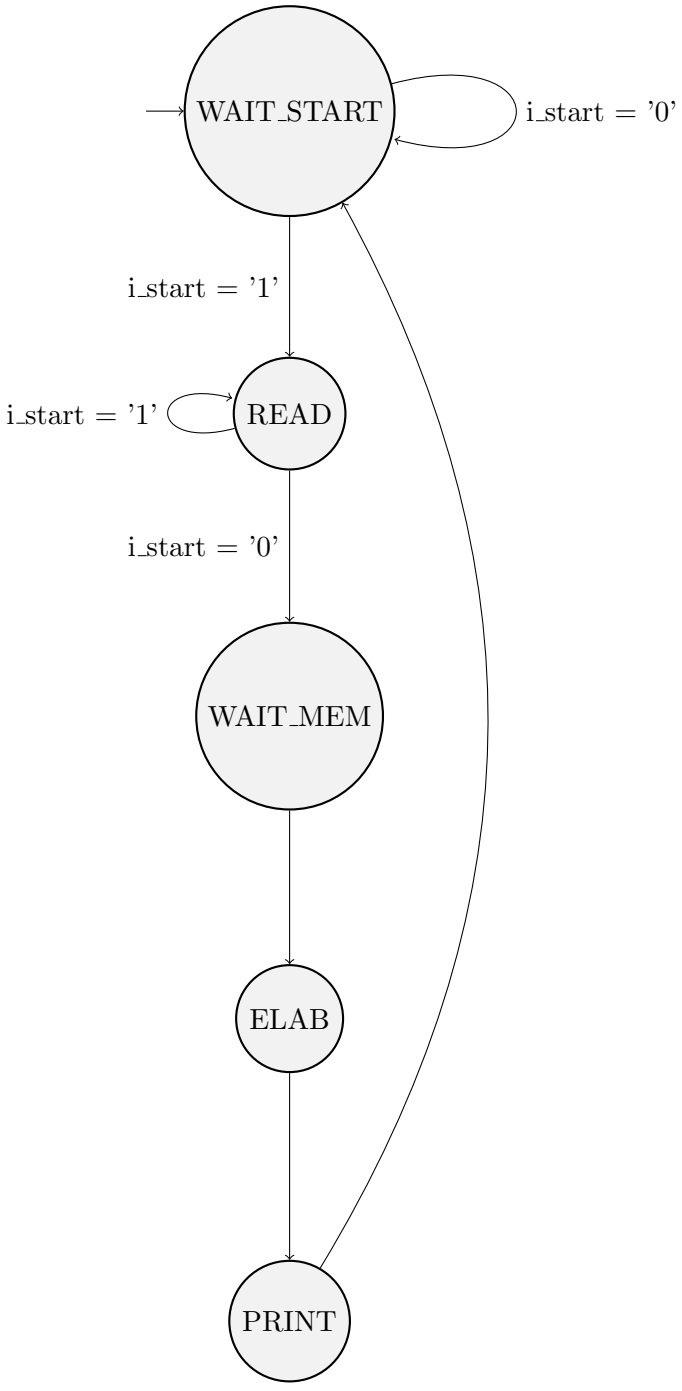

Figura 3: Diagramma delle transizioni di stato

3 Risultati sperimentali

3.1 Sintesi

L'hardware progettato è funzionante in tutte le sue parti. Per la sintesi del componente abbiamo utilizzato un FPGA Artix-7 (xc7a50tcsg324-1). Dall'utilization report si può notare come vengano utilizzati solo FF, non vi è quindi la presenza di inferred latches che potrebbero creare comportamenti imprevedibili.

Resource	Utilization	Available	Utilization %
LUT	66	32600	0.20
FF	105	65200	0.16
IO	63	210	30.00

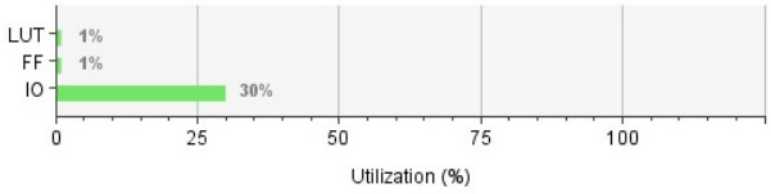

Figura 4: Utilization report

Come si può vedere nel riassunto di Timing (fig. 5) il componente presenta un buon Worst Negative Slack (WSN) in quanto si avvicina molto alla durata di un ciclo di clock (100ns).

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	97,357 ns	Worst Hold Slack (WHS):	0,134 ns	Worst Pulse Width Slack (WPWS):	49,500 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	174	Total Number of Endpoints:	174	Total Number of Endpoints:	106

All user specified timing constraints are met.

Figura 5: Timing summary

Da questi valori è anche possibile calcolare la massima frequenza a cui il componente può lavorare correttamente:

$$f_{max} = \frac{1}{T_{min}} = \frac{1}{T_{clk} - WNS} = \frac{1}{100ns - 97,367ns} = \frac{1}{2,633ns} = 379,8Mhz \qquad (1)$$

3.2 Simulazioni

Il modulo ha passato con successo tutti i test forniti, sia in behavioural simulation che in post-synthesis (sia functional che timing). A partire dai testbench di esempio abbiamo creato i seguenti che simulano condizioni limite:

- Reset asincrono casuale durante la computazione: il modulo funziona in modo corretto inizializzando ad ogni reset tutte le uscite a zero. Un primo caso si ha quando il reset avviene tra diverse elaborazioni cancellando quindi i valori precedenti (fig. 6).

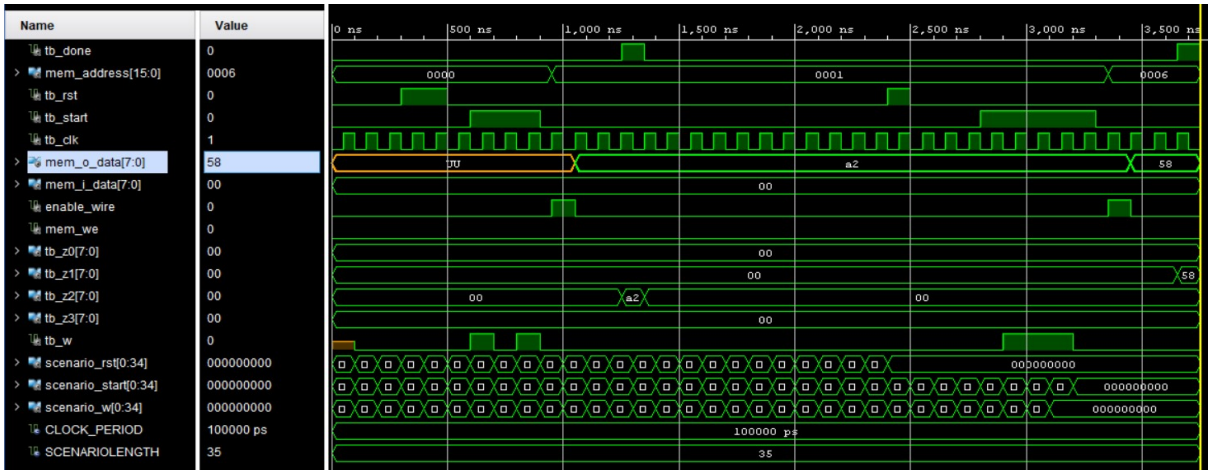

Figura 6: Reset tra due elaborazioni

Un altro caso si ha quando il reset avviene subito dopo la fine di un'elaborazione, prima che il modulo possa portare *o_done = 1* e quindi prima che il risultato di quest'ultima possa essere visibile in output (fig. 7).

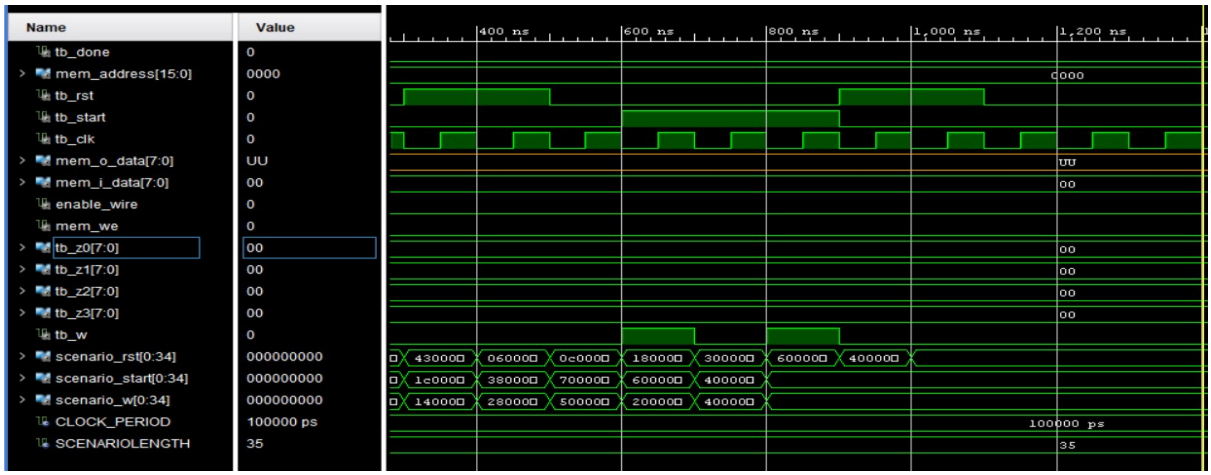

Figura 7: Reset subito dopo un'elaborazione

- Indirizzo di memoria 0000: il modulo funziona correttamente portando in uscita il contenuto della prima cella di memoria.

Figura 8: Indirizzo "0000"

- Indirizzo di memoria ffff: il modulo funziona correttamente portando in uscita il contenuto dell'ultima cella di memoria.

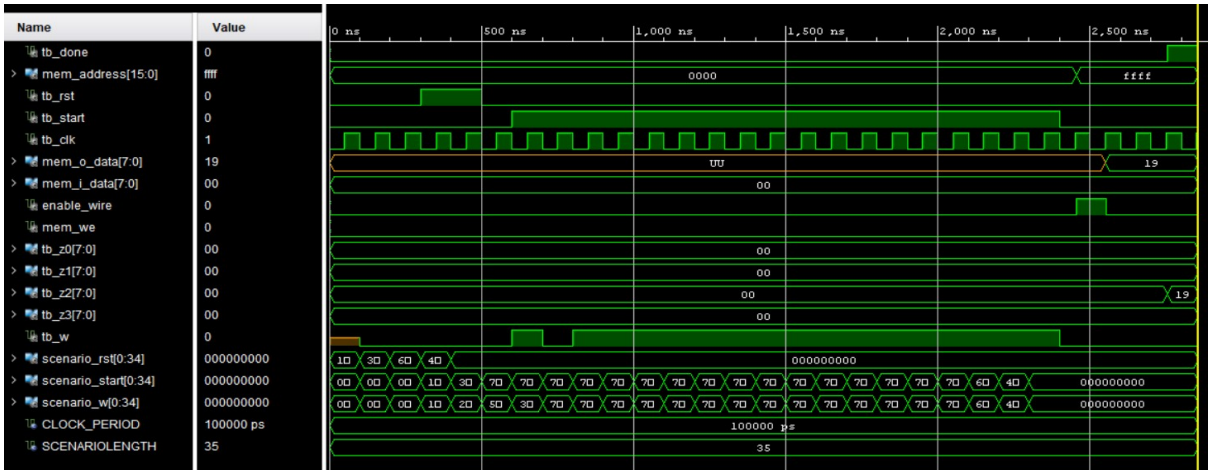

Figura 9: Indirizzo "ffff"

- Start alto per soli due cicli di clock: in caso di indirizzo "mancante" abbiamo deciso di prendere di default quello tutto a '0'.

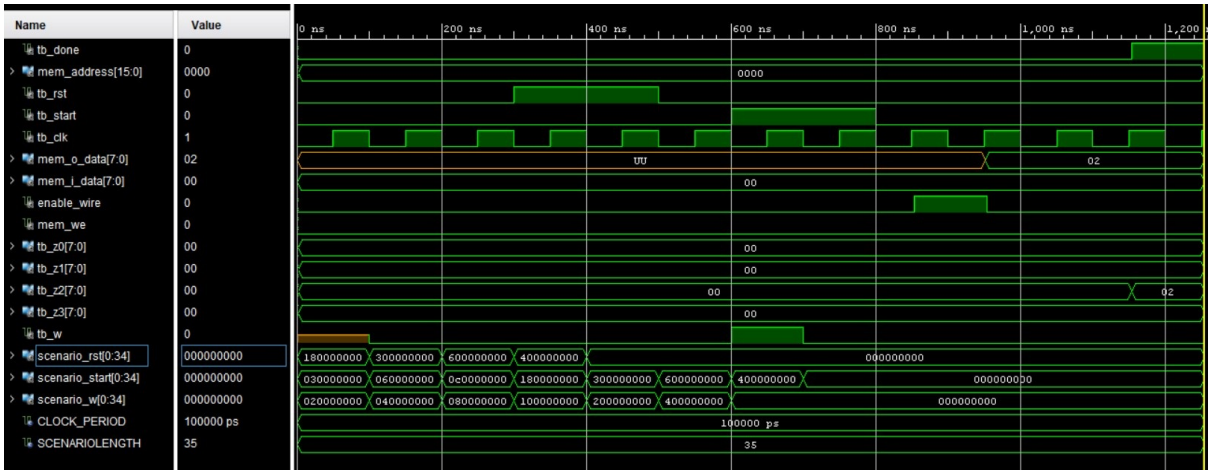

Figura 10: Start alto due soli cicli di clock

4 Conclusioni

Nel complesso il progetto è stato molto stimolante, ci ha dato la possibilità di approfondire la programmazione di hardware e il funzionamento delle FPGA.

Siamo partiti dalla comprensione della specifica e dalla stesura di tutti i casi limite a cui avremmo dovuto fare particolare attenzione. Disegnata una bozza della macchina a stati abbiamo scritto un primo codice a più processi che si occupavano di aspetti differenti di questa, ma provando a simularlo questo passava il behavioural ma non la post-sintesi. L'idea era quella di rendere il più possibile parallelo il tutto, ma così facendo abbiamo complicato inutilmente la specifica. Quindi nel risolvere i vari warning dovuti alla sintesi ci siamo accorti di star procedendo verso una semplificazione, fino ad arrivare alla soluzione finale a singolo processo, che oltre a passare tutte le simulazioni anche in post-sintesi dimostra una buona performance.

Infine siamo passati all'ottimizzazione riducendo il più possibile il numero degli stati, eliminandone alcuni che allungavano solo la computazione ed accorpando la lettura di i_w con l'elaborazione dell'indirizzo finale (estensione con 0) nella fase di READ.