POLITECNICO
MILANO 1863

Prova Finale di Reti Logiche

Prof. Gianluca Palermo

Manuel Tacca – 10707009

Anno Accademico 2022/23

Sommario

1. Introduzione

1.1 Descrizione della specifica

La specifica del Progetto Finale di Reti Logiche 2022/23 richiede l'implementazione di un modulo hardware descritto tramite VHDL.

Tale componente, che si interfaccia con una memoria già descritta, si occupa di trasmettere un certo valore da 8 bit su uno dei quattro canali output disponibili. Questo valore, insieme al canale di uscita, viene fornito al componente mediante un ingresso seriale da un bit. In particolare, i primi due bit ricevuti rappresenteranno il canale su cui trasmettere il valore ("00" rappresenterà il canale 0, "01" il canale 1, "10" il canale 2, "11" il canale 3), mentre i successivi indicheranno l'area di memoria da cui il componente dovrà prelevare il dato da portare in uscita.

L'input seriale è considerato valido fintantoché il segnale "*i_start*" è alto. Non appena questo si abbasserà, l'elaborazione incomincerà. "*i_start*" può restare alto da un minimo di 2 ad un massimo di 18 cicli di clock. I bit dell'area di memoria che non sono stati esplicitamente indicati dall'input, andranno posti a 0. Ad esempio, con un ingresso *w = "01" & "1010"*, si otterrà che dovremmo trasmettere sul canale 1 il valore contenuto nell'area di memoria "0000000000001010" (ossia 10).

L'elaborazione dell'ingresso seriale w deve avvenire entro 20 cicli di clock a partire da quando il segnale "*i_start*" si abbassa. Entro tale limite il componente dovrà:

- Leggere e salvare la codifica associata con il canale di uscita,
- Leggere e salvare l'indirizzo di memoria da cui reperire il dato,
- Salvare il dato letto dall'area di memoria di cui sopra nel registro corrispondente al canale comunicato in ingresso, eventualmente sovrascrivendolo,
- Alzare il segnale "*o_done*" e mandare in uscita il contenuto dei registri corrispondenti ai quattro canali output,
- Abbassare il segnale "*o_done*" e mettersi in attesa di un nuovo segnale di start.

Il componente dovrà tenere conto di ogni sequenza di ingresso che avviene tra due segnali di reset. Questo vuol dire che, ad esempio, dopo aver elaborato un primo ingresso, potrebbe presentarsi un nuovo "*i_start"*, che ci comunicherà nuovamente un canale ed un'area di memoria. A questo punto ci potremmo trovare davanti a due casi:

1. Il canale di uscita è stato utilizzato in precedenza: in questo caso verrà considerato soltanto il dato dell'elaborazione corrente e dovremmo sovrascrivere il registro corrispondente perdendo quanto già presente in tale registro.
2. Il canale di uscita non è mai stato utilizzato prima: in questo caso non ci sarà alcuna sovrascrittura.

Ogni volta che si dovrà mandare un nuovo valore in output, anche i valori appresi dalle precedenti elaborazioni dovranno apparire in uscita. Ad esempio, se in una prima elaborazione ottengo che sul canale 0 dovrà esserci il valore 10, nelle elaborazioni successive ci dovrà sempre essere 10 sul canale 0, salvo i casi in cui ci sia un reset oppure il canale 0 venga sovrascritto.

In ogni momento il segnale "*i_rst"* potrà essere alzato asincronicamente, in tal caso si dovrà reinizializzare il componente alle condizioni iniziali.

1.2 Interfaccia del componente ed esempio

1.2.1 Interfaccia del componente

Il componente desiderato si interfaccerà con una memoria RAM e con l'esterno tramite le seguenti porte:

```
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_clk*** è il segnale di CLOCK in ingresso generato dal Test Bench;

- ***i_rst*** è il segnale di RESET che inizializza la macchina rendendola pronta per ricevere il primo segnale di START;

- ***i_start*** è il segnale di START generato dal Testbench;

- ***i_w*** è il segnale W precedentemente descritto e generato dal Testbench;

- ***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;

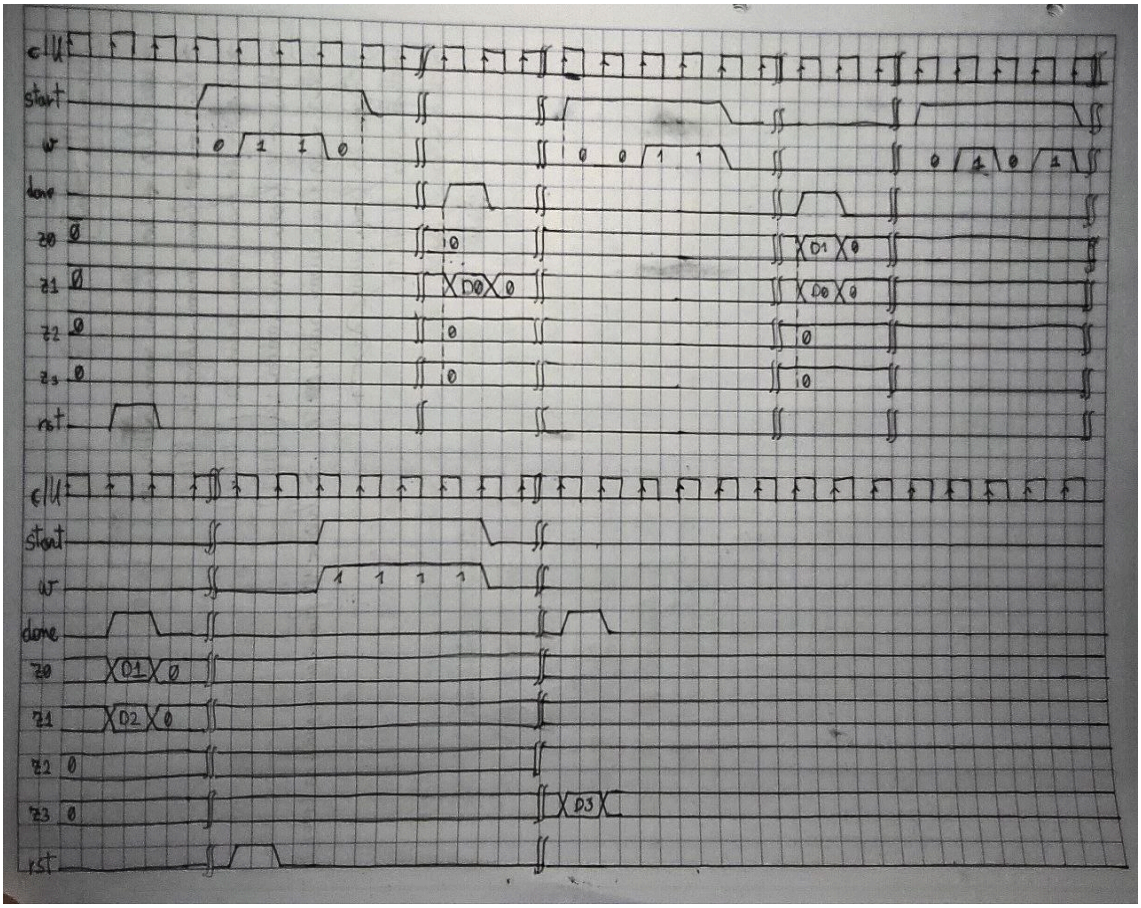
- ***i_mem_data*** è il segnale (vettore) che arriva dalla memoria in seguito ad una richiesta di lettura;

- ***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.

1.2.2 Esempio

Per comprendere del tutto il funzionamento del componente, prendiamo in considerazione un esempio: lo scenario comprende tre START consecutivi, seguiti da un RESET e da un altro START, così da coprire i casi fondamentali discussi. Questo esempio verrà esposto tramite un diagramma temporale:



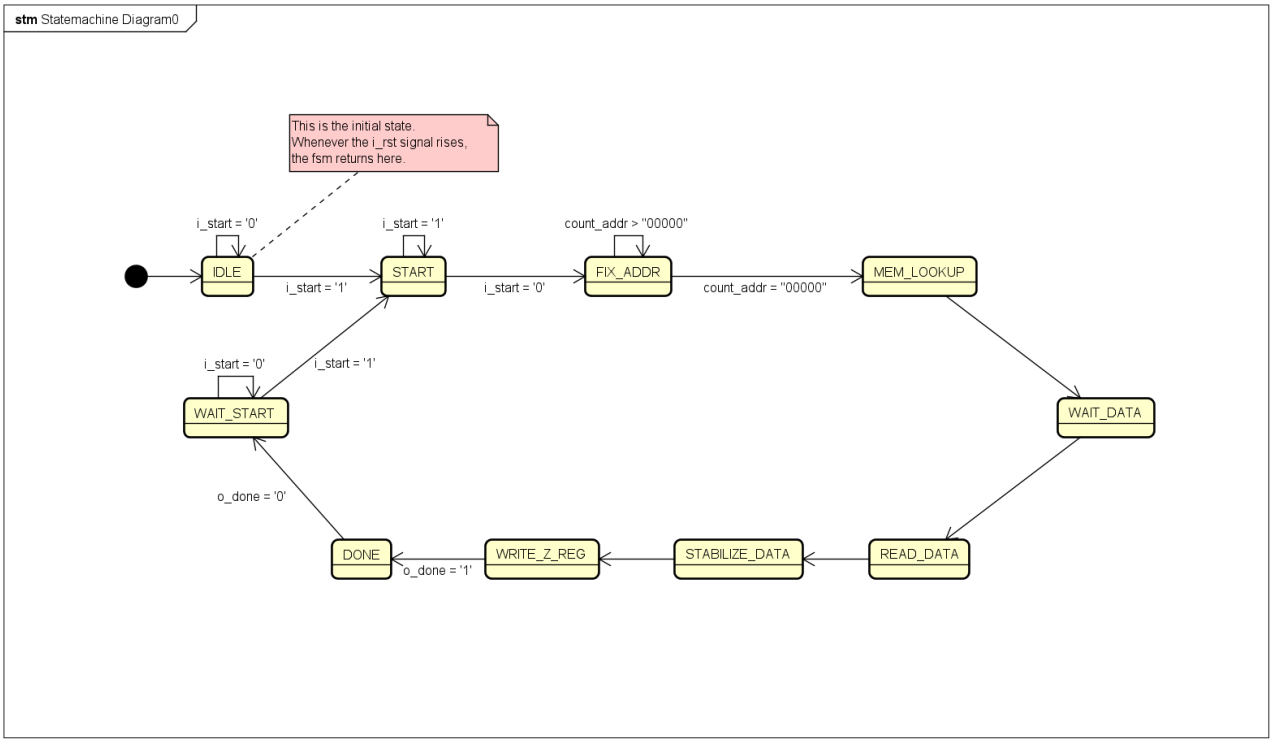
Abbiamo le prime tre sequenze seriali che portano ai seguenti rispettivi comportamenti:

- Input 1: sul canale “01” viene trasmesso il valore *D0*, contenuto nell’area di memoria 0000000000000010 (d’ora in poi l’area di memoria verrà espressa nei termini dei bit significativi)
- Input 2: sul canale “00” viene trasmesso il valore *D1*, contenuto nell’area di memoria 11 e sul canale “01” il valore *D0*, che è stato fornito dall’input 1
- Input 3: sul canale “01” viene trasmesso un nuovo valore *D2*, contenuto nell’area di memoria 01 e sul canale “00” il valore *D1*, fornito dall’input 2.

Dopodiché abbiamo un reset, che riporta il componente allo stato iniziale, seguito da un nuovo input che trasmette sul canale “11” il valore *D3* (uguale a *D1*), contenuto nell’area di memoria 11.

2. Architettura

L'architettura del componente consiste in una macchina a stati finiti che gestisce tutte le operazioni che il componente deve eseguire in modo sequenziale. È una macchina di Moore sincrona i cui passaggi di stato dipendono dai segnali provenienti dalla memoria a cui è collegata (*i_start, i_rst*). Di seguito la rappresentazione di tale FSM:



In questo diagramma, per semplicità, non sono mostrati gli archi che a partire da ogni stato portano ad IDLE quando *i_rst = '1'*.

Ogni stato ha una sua particolare funzione. Prima di andare nel loro dettaglio è tuttavia bene nominare e spiegare la funzione dei signal interni alla FSM:

```
signal count_chnl : UNSIGNED(1 DOWNTO 0)          := "10";
signal count_addr : UNSIGNED(4 DOWNTO 0)          := "00000";
signal c_help     : INTEGER                       := 1;
signal channel    : STD_LOGIC_VECTOR(1 DOWNTO 0)  := "00";
signal mem_inv    : STD_LOGIC_VECTOR(15 DOWNTO 0) := "0000000000000000";
signal mem_addr   : STD_LOGIC_VECTOR(15 DOWNTO 0) := "0000000000000000";
signal data       : STD_LOGIC_VECTOR(7 DOWNTO 0)  := "00000000";
signal z0_reg     : STD_LOGIC_VECTOR(7 DOWNTO 0)  := "00000000";
signal z1_reg     : STD_LOGIC_VECTOR(7 DOWNTO 0)  := "00000000";
signal z2_reg     : STD_LOGIC_VECTOR(7 DOWNTO 0)  := "00000000";
signal z3_reg     : STD_LOGIC_VECTOR(7 DOWNTO 0)  := "00000000";
```

- **count_chnl:** contatore utile a tenere conto di quanti bit del segnale in ingresso *i_w* relativi al canale di uscita sono stati letti fino a questo momento. Poiché questo dato va sempre fornito e poiché il numero di bit di codifica del canale è uguale a due, conviene inizializzare il *count_chnl* a "10".

- **count_addr:** contatore utile a tenere conto del numero di bit relativi all'area di memoria forniti in ingresso, parte da 0 e può raggiungere 15.
- **c_help:** contatore di utilità per l'inversione dei bit relativi alla memoria.
- **channel:** signal con il compito di mantenere la codifica del canale di uscita.
- **mem_inv:** signal che mantiene esattamente come vengono forniti i bit relativi all'area di memoria. Ad esempio, se abbiamo come indirizzo di memoria "10110", *i_w* trasmetterà 0 1 1 0 1 → che verrà memorizzato dal signal *mem_inv* esattamente così: "0000000000001101".
- **mem_addr:** signal che mantiene correttamente l'indirizzo di memoria da cui recuperare il dato da trasmettere. Con riferimento all'esempio del punto precedente, mem_addr = "0000000000010110"
- **data:** signal che mantiene il dato da trasmettere, letto dalla memoria all'indirizzo contenuto in *mem_addr*
- **zx_reg:** signal che svolge la funzione di registro, prima di alzare il segnale di '*o_done'*, il valore contenuto in *data* verrà memorizzato nel registro corrispondente. Per determinare quale sia il registro corretto è sufficiente consultare il contenuto del signal *channel*.

Ognuno dei dieci stati della FSM sopra rappresentata svolge una determinata operazione. Ora che sono chiari i segnali interni, vediamo di che cosa si occupa ogni stato:

1. **IDLE:** È lo stato iniziale, porta tutti i segnali interni alla condizione iniziale ed attende che si alzi il segnale *i_start.*

2. **START:** Si occupa di leggere i bit seriali *i_w* e di inserirli all'interno di *channel* e di *mem_inv*. Ad ogni fronte di salita del clock decrementa il contatore relativo al canale (*count_chnl*) finché non raggiunge il valore "00", dopodiché passa all'incremento di *count_addr* ed all'inserimento dei bit così come arrivano in *mem_inv*. Restiamo in questo stato finché il segnale *i_start* non si abbassa.

3. **FIX_ADDR:** Si occupa di invertire i bit significativi (quelli letti da *i_w* con *i_start='1'*), di modo da avere nel segnale *mem_addr* il corretto indirizzo di memoria. Ad ogni ciclo di clock, scambia il primo e l'ultimo bit di *mem_inv*, inserendoli in *mem_addr*, per poi passare al secondo ed al penultimo, al terzo ed al terzultimo eccetera. Ad ogni iterazione incrementa *c_help* e decrementa *count_addr* per tenere traccia delle posizioni da invertire al clock successivo. L'operazione termina quando questi due contatori sono di uguale valore (quando il numero di bit validi per l'indirizzo di memoria è dispari) o quando *count_addr < c_help* (quando il numero di bit validi per l'indirizzo di memoria è pari). Prima di passare allo stato successivo, alza il segnale di interfaccia *o_mem_en*.

4. **MEM_LOOKUP:** pone nel segnale di interfaccia *o_mem_addr* il valore contenuto nel segnale interno *mem_addr.*

5. **WAIT_DATA:** fa passare un ciclo di clock per attendere il dato da leggere

6. **READ_DATA:** inserisce nel segnale *data* quanto presente in *i_mem_data*

7. **STABILIZE_DATA:** abbassa *o_mem_en*, cancella il valore in *o_mem_data* e fa passare un ciclo di clock

8. **WRITE_Z_REG:** Si occupa di scrivere nel registro corretto il valore appena appreso. Per farlo si serve di un meccanismo simile ad un componente demultiplexer: a seconda del contenuto di *channel*, tramite un case when determina in quale dei quattro registri scrivere il valore.

9. **DONE:** alza il segnale *o_done* e mette in ogni rispettiva uscita *o_zx* il corrispondente valore contenuto in *zx_reg*.

10. **WAIT_START:** abbassa il segnale *o_done*, fa tornare a 0 i valori *o_zx*, reinizializza i valori di *channel, mem_addr, mem_inv, data, count_chnl, count_addr, c_help* e si mette in attesa di un nuovo start. Restiamo in questo stato fintantoché *i_start='0'.*

3. Risultati sperimentali

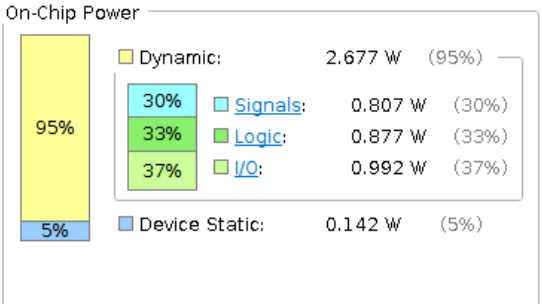
3.1 Sintesi

Il componente è correttamente sintetizzabile ed implementabile con un totale di 344 LUT e 167 FF.
Il consumo totale di potenza è 2.819W.

Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAM	URAM	PCIe %
synth_1	constrs_1	synth_design Complete!								344	167	0	0	0.000
impl_1	constrs_1	route_design Complete!	NA	NA	NA	NA	NA	2.590	0	313	167	0	0	0.000

Power estimation from Synthesized netlist. Activity derived from constraints files, simulation files or vectorless analysis. Note: these early estimates can change after implementation.

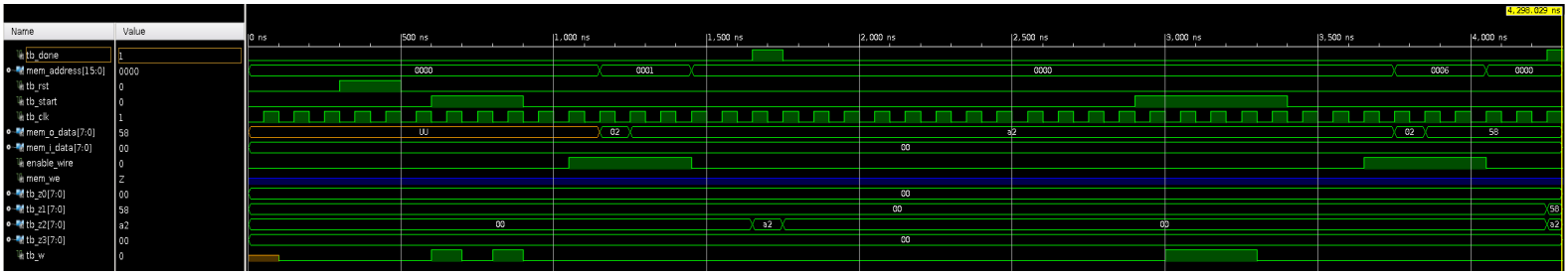
Total On-Chip Power:	**2.819 W**
Junction Temperature:	**32.0 °C**
Thermal Margin:	53.0 °C (21.2 W)
Effective θJA:	2.5 °C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low



3.2 Simulazioni

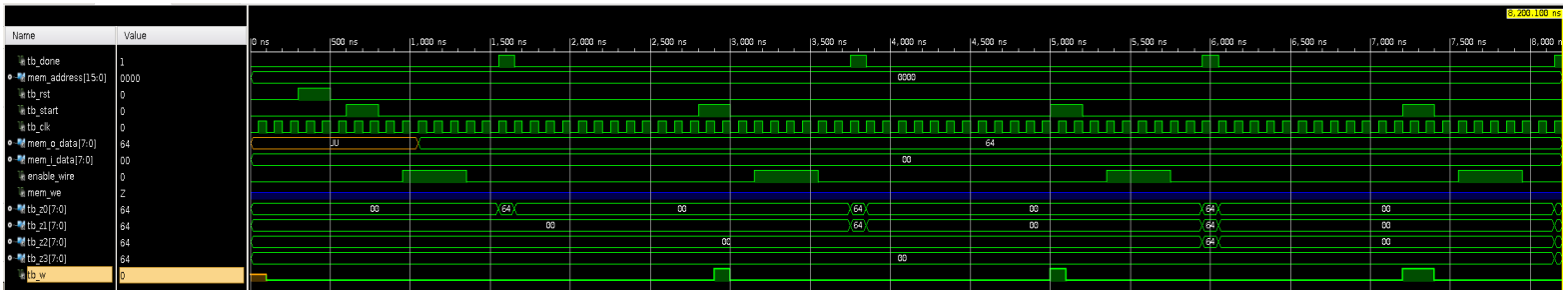
Per verificare il corretto funzionamento del componente ho deciso di utilizzare, oltre al testbench fornito dal professore, dei testbench sui casi limiti espressi nella specifica. I risultati ottenuti verranno mostrati di seguito:

- **Testbench del docente:**



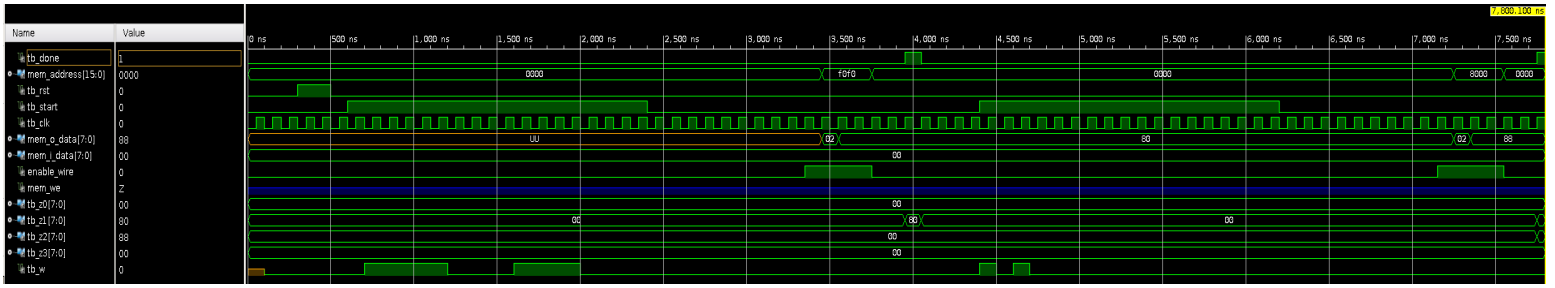
- **Testbench w=2:**

Testbench volto al testing del caso in cui abbiamo in ingresso il numero minimo di bit. Ossia, quando viene fornito soltanto il canale di uscita. Ho deciso di mettere nell'area di memoria 0 il valore 64 e di trasmetterlo in ordine dall'uscita 0 alla 4.



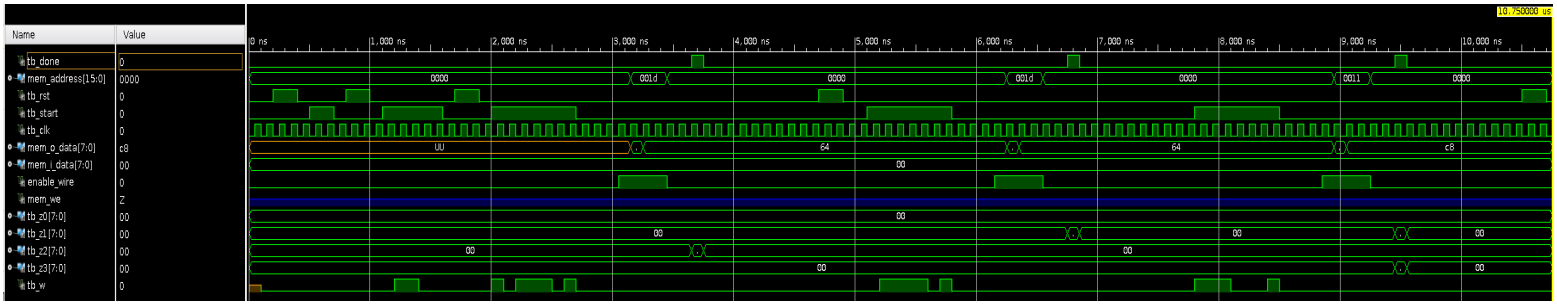
- **Testbench w=18:**

In questo testbench verifico il funzionamento del componente nel caso in cui *i_start* resta alto per 18 cicli di clock, ossia il tempo massimo. In questo caso non vanno aggiunti '0' infondo all'indirizzo di memoria poiché avremmo tutti bit validi. La difficoltà di questo testbench è di avere il risultato in output entro i 20 clk dopo che *i_start* si abbassa.



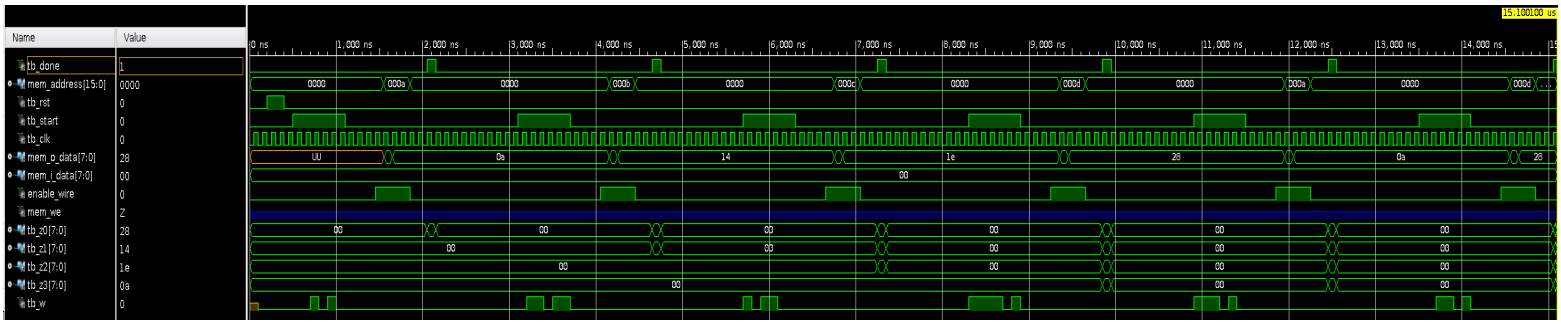
- **Testbench multireset:**

In questo testbench verifico il funzionamento del componente in termini del segnale asincrono di reset, alzandolo diverse volte in diversi stadi di elaborazione. Viene alzato dopo aver letto il *channel*, dopo aver letto *channel* e *mem_addr*, dopo il risultato di un'elaborazione (*o_done = '1'*), dopo due elabrazioni consecutive. In tutti i casi il comportamento si è rivelato quello desiderato.



- **Testbench multistart:**

In questo caso si verifica il comportamento corretto nel caso in cui ci siano molti start consecutivi. Ho usato 4 start per assegnare rispettivamente a ch0 <- 10, ch1 <- 20, ch2 <- 30, ch <- 40, seguiti da due start che vanno ad invertire i valori in ch0 e in ch3: ch3 <- 10, ch0 <- 40.



Oltre a questi testbench, il componente è stato testato anche su altri, più generici, risultando in tutti corretto sia in simulazione Behavioral, che in simulazione funzionale Post-Synthesis.

4. Conclusioni

Ho scelto di utilizzare un'implementazione seguendo l'utilizzo di una FSM siccome reputo questa la soluzione più semplice e intuitiva. Probabilmente si sarebbe potuto usare un approccio diverso, andando ad integrare diversi componenti con processi a sé stanti che sarebbero stati chiamati dalla FSM tramite dei segnali interni. Ad esempio, per quanto riguarda l'inversione dei bit che indicano l'area di memoria, oppure per la decisione del canale su cui scrivere, che sono operazioni funzionali, si sarebbero potuti utilizzare due componenti esterni interfacciati con la FSM, che non avrebbe più avuto logica interna, ma avrebbe svolto il compito di controllore, gestendo soltanto i passi da svolgere nell'elaborazione. Inizialmente ho utilizzato questo approccio, che poi ho abbandonato per uno più pratico. Tuttavia, un approccio simile a quello descritto sopra avrebbe diminuito l'utilizzo di LUTs, che in questo caso è molto alto, raggiugendo forse un risultato migliore in termini di prestazione e risorse.