Progetto finale di Reti Logiche

Marco Balossini - Matricola n. 889075

Anno Accademico 2019/2020

Indice

1 Introduzione

1.1 Scopo del progetto

Si definiscano 8 intervalli di indirizzi di dimensione fissa, detti working zones. Lo scopo del progetto è implementare un componente hardware, descritto in VHDL, che riproduca il metodo di codifica a bassa dissipazione detto "Working Zone". Questo metodo prende in input un indirizzo e ne cambia la codifica se appartiene ad una delle working zones.

1.2 Specifiche generali

Questo metodo prende in input un indirizzo `ADDR_IN` da trasmettere e gli indirizzi iniziali di 8 working zones (intervalli di indirizzi di dimensione fissa) per stabilire l'appartenenza di `ADDR_IN` ad una delle working zones. Nel nostro caso consideriamo indirizzi di 7 bit ed un numero di working zones pari a 8.

- Se `ADDR_IN` non appartiene a nessuna WZ, viene trasmesso così com'è in output preceduto però da un bit posto a 0
- Se `ADDR_IN` appartiene ad una qualsiasi WZ, viene propagato un nuovo vettore di bit. Il più significativo sarà posto ad 1, i successivi tre indicheranno il numero della WZ di appartenenza, mentre gli ultimi 4 indicheranno l'offset in codifica one-hot.

Supponendo gli indirizzi dati in input come corretti, `ADDR_IN` sarà compreso tra 0 e 127, mentre gli indirizzi iniziali delle working zones saranno tra 0 e 124 (127 meno l'offset massimo).

WZ1	2		107	WZ1
WZ2	15		26	WZ2
WZ3	46		42	WZ3
WZ4	80		89	WZ4
WZ5	40		12	WZ5
WZ6	23		54	WZ6
WZ7	121		6	WZ7
WZ8	100		1	WZ8
`ADDR_IN`	5		125	`ADDR_IN`
OUT	`ADDR_OUT`		`ADDR_OUT`	OUT

Tabella 1: Esempi 1 e 2

Guardando la prima memoria d'esempio, notiamo che `ADDR_IN` appartiene alla prima working zone e `ADDR_OUT` si può ottenere concatenando:

- `'1'`, il bit più significativo viene posto ad uno per indicare una corrispondenza
- `"000"`, il numero della working zone corrispondente
- `"1000"`, l'offset in codifica one-hot, infatti `1000` corrisponde a `3=5+2`

Guardando la seconda memoria di esempio, notiamo che non ci sono corrispondenze, quindi `ADDR_OUT` sarà pari a `ADDR_IN`.

1.3 Descrizione della memoria

Il modulo comunica con la memoria RAM, avente un indice di indirizzamento al byte.

- Agli indirizzi da 0 a 7 sono salvati gli indirizzi di partenza delle 8 working zones
- All'indirizzo 8 si legge l'indirizzo da propagare
- L'indirizzo 9 verrà sovrascritto per salvare il risultato del processo

Come poi vedremo nell'interfaccia del componente, l'indirizzo è rappresentato come un vettore di 16 segnali logici, di cui però utilizzerò solamente i 4 meno significativi. La quasi totalità dei warning segnalati da Vivado è appunto dovuta al non utilizzo dei mancanti 12 bit.

Addr 0	WZ 1
Addr 1	WZ 2
Addr 2	WZ 3
Addr 3	WZ 4
Addr 4	WZ 5
Addr 5	WZ 6
Addr 6	WZ 7
Addr 7	WZ 8
Addr 8	ADDR IN
Addr 9	ADDR OUT

Tabella 2: Rappresentazione della memoria

1.4 Interfaccia del componente

```
entity project_reti_logiche is
        port (
                i_clk : in std_logic;
                i_start : in std_logic;
                i_rst : in std_logic;
                i_data : in std_logic_vector(7 downto 0);
                o_address : out std_logic_vector(15 downto 0);
                o_done : out std_logic;
                o_en : out std_logic;
                o_we : out std_logic;
                o_data : out std_logic_vector (7 downto 0)
        );
end project_reti_logiche;
```

`i_clk`	È il segnale di `CLOCK` dato in ingresso dal testbench
`i_start`	È il segnale di `START` generato dal testbench per iniziare l'esecuzione
`i_rst`	È il segnale di `RESET` che porta la macchina in `IDLE` e la prepara ad una nuova esecuzione
`i_data`	È un vettore di segnali che arriva dalla memoria RAM in seguito ad una richiesta di lettura
`o_address`	È il vettore di segnali in uscita che decide l'indirizzo di memoria al clock successivo
`o_done`	È il segnale di uscita, che comunica la fine dell'elaborazione
`o_en`	È il segnale di `ENABLE` che abilita l'accesso in memoria
`o_we`	È il segnale `WRITE ENABLE` che abilita la scrittura in memoria. Se è posto ad 1 allora scrivo in memoria, altrimenti leggo
`o_data`	È il segnale di output del componente che scrive in memoria

Tabella 3: Segnali dell'interfaccia

2 Architettura

2.1 Schema FSM

In questo capitolo spiegherò come è costituita la FSM che ho ideato partendo dallo schema.
Successivamente specificherò i singoli stati ed i segnali che ho utilizzato, ma che non sono presenti nell'interfaccia.

2.1.1 Rappresentazione grafica

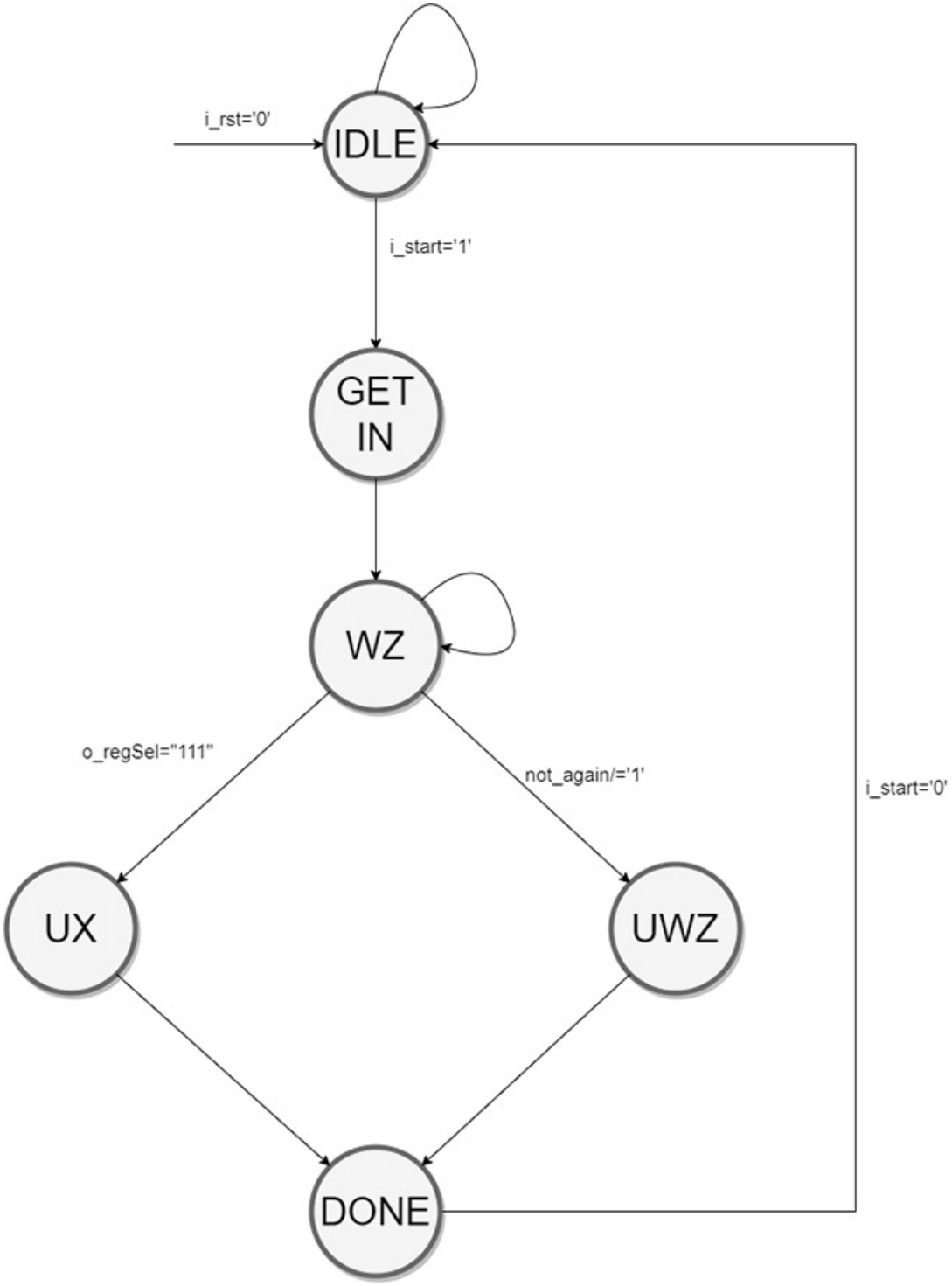


Figura 1: Macchina a Stati Finiti implementata

2.1.2 Tabella degli stati

IDLE	Stato di partenza della FSM, viene indotto anche asincronamente dal segnale i_rst. Alla ricezione di un segnale i_start si avanza allo stato GET_IN.
GET_IN	Stato iniziale, nel quale viene letto l'input dall'ottavo indirizzo di RAM. Da qui poi pongo i_address a "0000000000000000" per leggere nel successivo stato gli indirizzi iniziali delle working zones.
CTRL_WZ	Stato in cui il componente incrementa di '1' l'indirizzo ogni ciclo di clock, per confrontare ogni Working Zone con l'input. Quando il segnale not_again diventa diverso da "0000" la macchina esce da questo stato per andare nello stato UWZ. Se il segnale rimane a zero fin dopo l'ultimo controllo, il nuovo stato sarà UX.
UWZ	Questo stato viene raggiunto quando l'indirizzo da trasmettere è interno all'ultima Working Zone controllata. In questo stato il componente stampa in memoria all'indirizzo "0000000000001001" l'indirizzo propagato, nella forma '1' & o_regSel & o_regOH.
UX	Questo stato viene raggiunto quando tra le working zones non si trovano corrispondenze, ed il componente trasmette l'indirizzo iniziale.
DONE	In questo stato il segnale o_done viene posto a '1' per poi tornare in IDLE.

2.1.3 Segnali interni e registri

not_again	È un segnale posto a "0000" se l'indirizzo non è interno alla working zone corrente, altrimenti è pari all'offset one-hot. È calcolato con l'OR bit a bit tra i segnali ctrl0, ctrl1, ctrl2 e ctrl3. Ogni ctrln è definito come la codifica one-hot corrispondente al proprio offset, se l'indirizzo da trasmettere è pari all'indirizzo della working zone sommato a n, "0000" altrimenti. Ogni turno il valore di not_again viene poi salvato nel registro o_regOH
mux	È il multiplexer utilizzato per implementare un contatore da "000" a "111" per aumentare o_addr. Ad ogni turno viene salvato il suo valore nel registro o_regSel, da utilizzare come numero della working zone salvata.
o_regSel	È il registro in cui viene salvato il numero della working zone per da concatenare per comporre i risultati positivi.
mux	È il registro dove viene salvato l'offset in one-hot, se esiste, altrimenti 0.

2.1.4 Funzionamento nel dettaglio

La macchina parte dallo stato di IDLE, dal quale esce solo in seguito all'arrivo del segnale i_start. Con lo start essa entra in un primo stato dove legge l'indirizzo da propagare, ADDR_IN, e lo salva in o_regIN, azzera mux per far partire il contatore e poi il clock successivo entra nello stato di controllo delle working zones.
In questo stato la macchina, controlla una working zone per ogni ciclo di clock. Ogni ciclo immette il valore letto dalla memoria ADDR_WZ nel circuito (senza salvarlo in un registro) che controlla in parallelo la parità di ADDR_IN con ADDR_WZ, ADDR_WZ+1, ADDR_WZ+2 e ADDR_WZ+3. Se uno di questi valori è pari a ADDR_IN, allora il corrispondente vettore di controllo prende il valore dell'offset in codifica one-hot. Alla fine di ogni controllo, salvo il valore di not_again (or bit a bit tra i quattro vettori di controllo) nel registro o_regOH e, se è diverso da "0000", la macchina entra nello stato UWZ, dove salva l'output nella forma '1' & o_regSel & o_regOH. Se la macchina arriva all'ultima WZ senza trovare corrispondenze, allora entra nello stato UX dove salva in memoria ADDR_IN.
Da questi due stati si entra nello stato DONE, dove si alza il segnale o_done per concludere il lavoro e tornare nello stato di IDLE.

2.2 Scelte progettuali

Ho scelto di descrivere il componente con un solo processo. La scelta principale che ho effettuato è stata quella di non salvare gli indirizzi iniziali delle working zones, in modo da risparmiare in componenti e facilitare la **scalabilità** del componente. Parlando di scalabilità, infatti, per aumentare il numero di working sones bisognerebbe semplicemente aumentare la lunghezza dei vettori di segnali, senza dover però aumentare il numero di registri. Questa scelta, oltre ad i vantaggi già elencati, ha però lo svantaggio di dover rileggere tutte le working zones da memoria, anche nel caso di due esecuzioni diverse senza che vi sia stato il segnale di reset.
I controlli sui vari offset di ogni working zone sono eseguiti in parallelo per minimizzare il tempo di esecuzione.

---

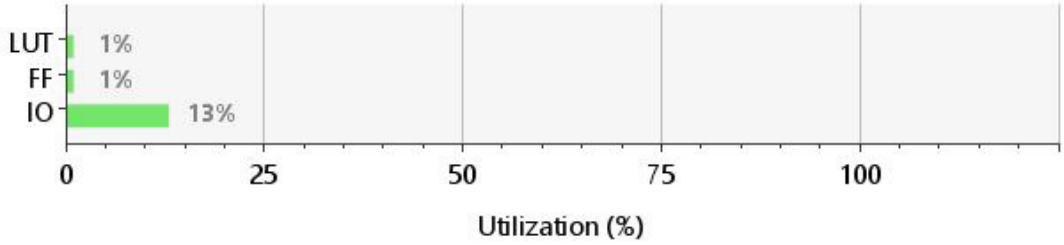
3 Sintesi

3.1 Report Utilization

Analizzando il "Vivado Synthesis Report" troviamo che i tre registri sono stati sintetizzati da FF senza latch inferiti. Dal Report Utilization leggiamo che il progetto utilizza 18 FF e 45 LUT, una percentuale irrisoria della disponibilità della FPGA. Infatti, date le modeste dimensioni del progetto, nelle scelte progettuali non ho ritenuto di grande importanza la riduzione dell'area occupata.

Summary

Resource	Utilization	Available	Utilization %
LUT	45	134600	0.03
FF	18	269200	0.01
IO	38	285	13.33



3.2 Timing Report

Analizzando il timing report, si può vedere quanto del periodo di clock è effettivamente utile per svolgere il lavoro. Si è ottenuto con il periodo di clock del testbench di 100ns un Worst Negative Slack pari a 95,695ns. Da questo valore, sapendo anche il ritardo di riposta della RAM, possiamo calcolare il clock minimo applicabile al design creato:

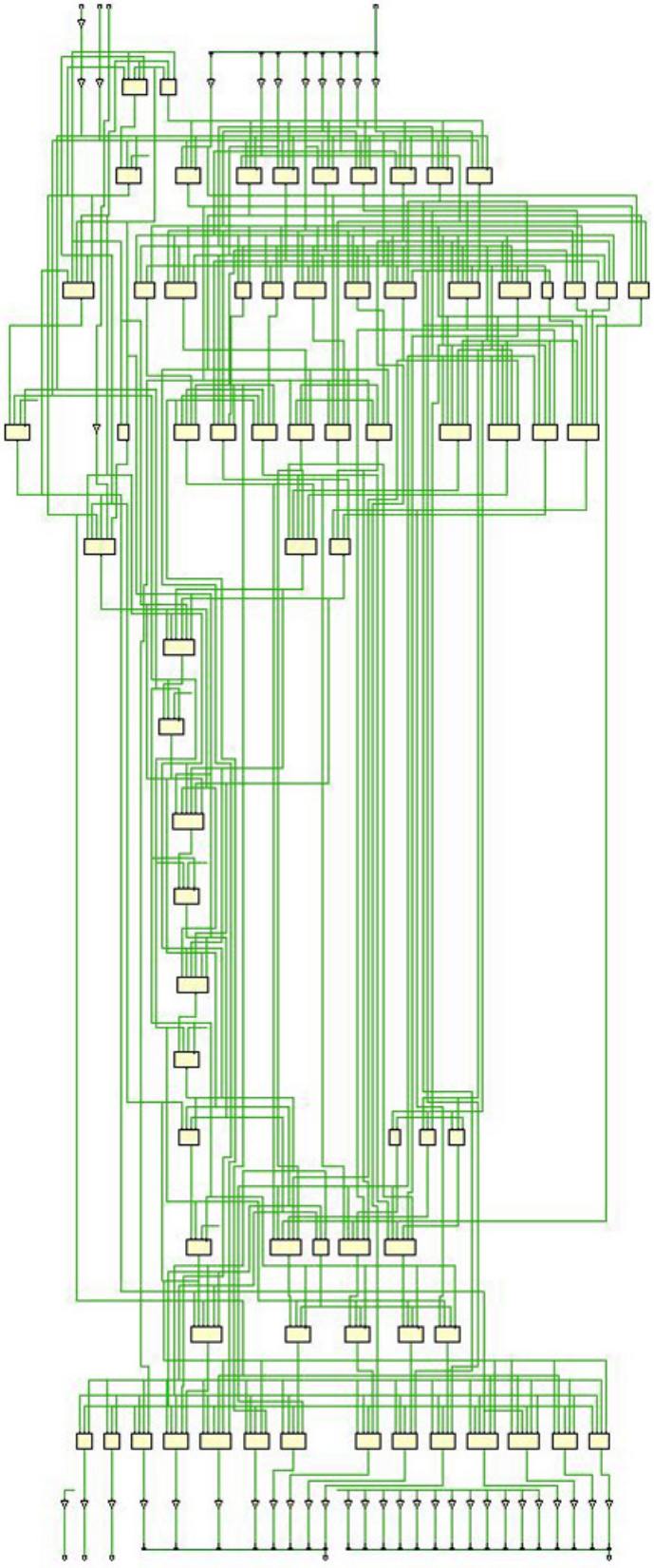
$$T_{min} = T_{curr} - WNS + T_{RAM} = 100ns - 95.695ns + 2ns = 6.305ns$$

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	95,695 ns	Worst Hold Slack (WHS):	0,146 ns	Worst Pulse Width Slack (WPWS):	4,500 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:	22	Total Number of Endpoints:	22	Total Number of Endpoints:	19

All user specified timing constraints are met.

3.3 Schema di sintesi

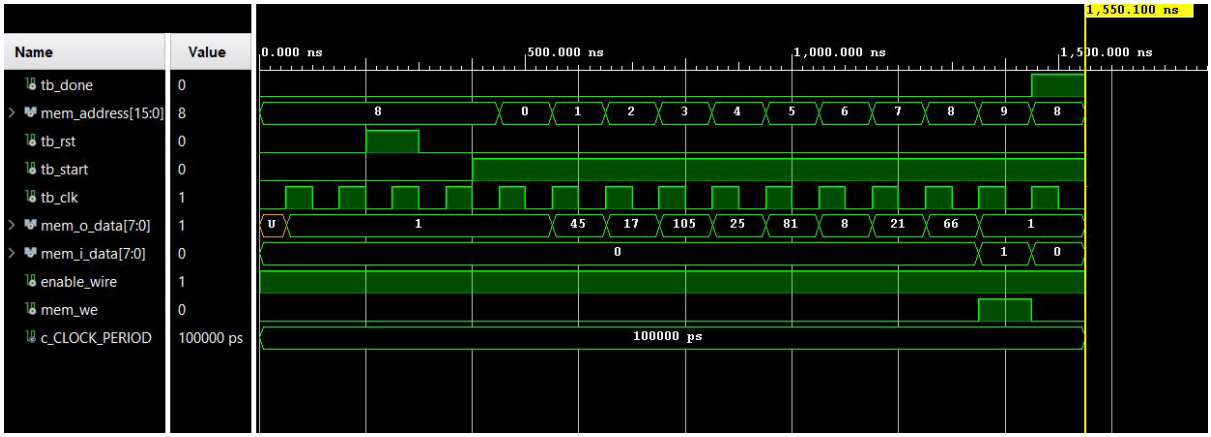


4 Simulazioni

Per poter testare il componente, dopo aver verificato il funzionamento generale senza controllare casi in particolare con i testbench forniti insieme alla specifica, ho scritto dei test bench per coprire i casi limite. Di seguito riporto i più significativi.
Tra i casi significativi non è presente una doppia esecuzione perché, date le caratteristica di assenza di memoria del mio componente, nella seconda esecuzione non ci sarebbero differenze rispetto alla prima.

4.1 Test Bench senza corrispondenze

Con questo test bench viene testato il caso in cui non esistano working zones che contengono l'indirizzo in ingresso. Come nel test bench dato con la specifica, testo un reset a macchina in idle.
Questo è il caso di test con tempo di esecuzione (calcolato in cicli di clock tra `i_start` e `o_done`) maggiore.

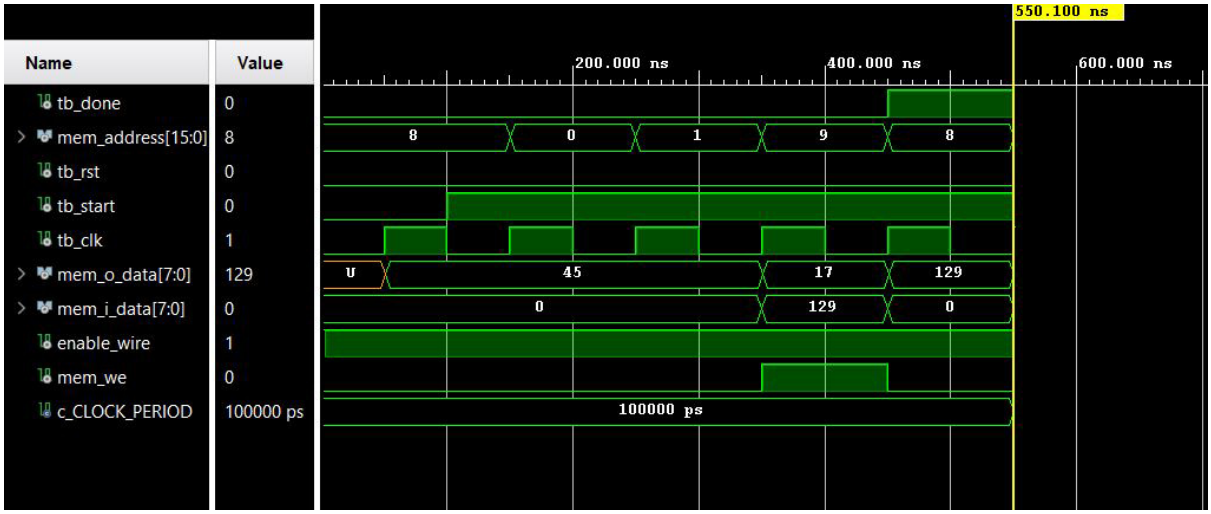


4.2 Test bench con corrispondenze agli estremi

Tutti i controlli sulle working zones si svolgono nello stesso stato della macchina.Verosimilmente se il componente funziona per una corrispondenza alla prima ed all'ultima wz allora funzionerà anche per tutte le altre nel mezzo (che comunque ho testato).

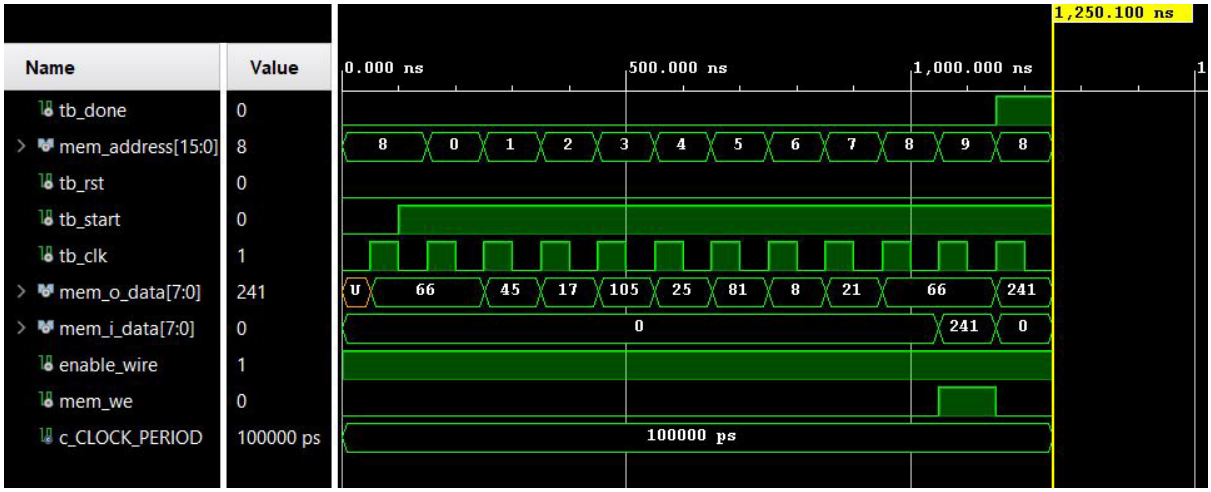
4.2.1 Test con corrispondenza alla prima working zone

Questo è il caso con il tempo di esecuzione più breve: quando la macchina esce dallo stato di `IDLE` in un ciclo di clock salva `ADDR_IN` in un registro, il successivo trova la corrispondenza, il terzo salva il risultato in RAM e l'ultimo pone `o_done` a `'1'`.



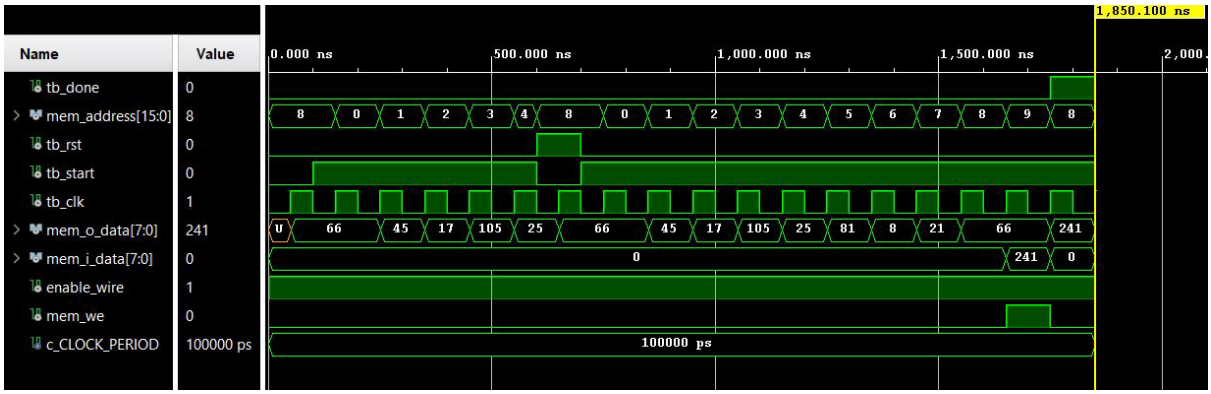
4.2.2 Test con corrispondenza all'ultima working zone

In questo caso testo di aver progettato correttamente la macchina perché il componente entri correttamente nello stato `UWZ` anche all'ultima working zone senza entrare nello stato `UX` in anticipo di un ciclo di clock.



4.3 Test Bench con reset asincrono

Questo caso testa che il trigger asincrono del segnale di reset non comprometta il funzionamento del componente, ma riporti semplicemente la macchina in stato di `IDLE`.



5 Conclusione

La realizzazione finale del progetto soddisfa gli obiettivi che mi ero posto:

- Ottimizzato dal punto di vista temporale, in modo da impiegare un solo clock per un controllo.
- Ottimizzato in modo che durante l'elaborazione venga sfruttata al massimo la RAM
- Facilmente scalabile nel caso aumenti il numero di working zones o la loro dimensione.
- Periodo di clock minimo pari a 6.305 ns.
- Utilizzo di 45 LUT.
- Utilizzo di 18 FF.

Il mio lavoro è iniziato con la costruzione del datapath e della macchina a stati. L'aver chiaro lo schema di funzionamento mi ha poi aiutato a scrivere il codice piuttosto semplicemente ed in modo ordinato. Dopo la scrittura del codice ho iniziato la fase di debug e corretto gli errori trovati. Infine la correzione di tutti i warning presenti mi ha permesso di ottenere subito un componente funzionante in postsintesi.