Progetto Finale di Reti Logiche

Introduzione

I consumi energetici di un componente hardware sono un fattore fondamentale da tenere in considerazione per la progettazione di processori elettronici.
Le operazioni di lettura e di scrittura giocano un ruolo fondamentale nel calcolo dell'assorbimento energetico.
Per questi motivi è stato sviluppato un modello di componente elettronico che permette un notevole risparmio energetico privilegiando l'accesso ad un insieme di indirizzi impiegati più frequentemente.
Tale struttura è chiamata modello a "zone di lavoro" o **working zone**, e consiste nel codificare un bus di indirizzi sapendo che questi sono maggiormente impiegati dai programmi in esecuzione.
Il calcolo consiste nel verificare l'appartenenza o meno dell'indirizzo da codificare ad una "zona di lavoro" o **intervallo di indirizzi** opportunamente pre-caricati in memoria.
Secondo il criterio di appartenenza, al numero di working zone e all'offset (differenza *aritmetica* tra indirizzo da codificare ed indirizzo contenuto nella WZ) si codifica un nuovo indirizzo che verrà impiegato dal programma.
Per aumentare le prestazioni in fase di codifica, si usano delle strategie quali l'uso di un bit di appartenenza a WZ, il numero identificativo della WZ e codifica one-hot applicata all'offset.

"... The approach has been applied to several address streams, broken down into instruction-only, data-only, and instruction-data traces, to evaluate the effect on separate and shared address buses. The effect of instruction and data caches is evaluated. For the case without caches, the proposed scheme is especially beneficial for data address and shared buses, which are the cases where other codings are less effective. On the other hand, for the case with caches the best scheme for the instruction-only and data-only traces is the WZE, whereas for the instruction-data traces it is either the WZE or the bus-invert with four groups (depending on the energy overhead of these techniques)."

E. Musol, T. Lang, J. Cortadella

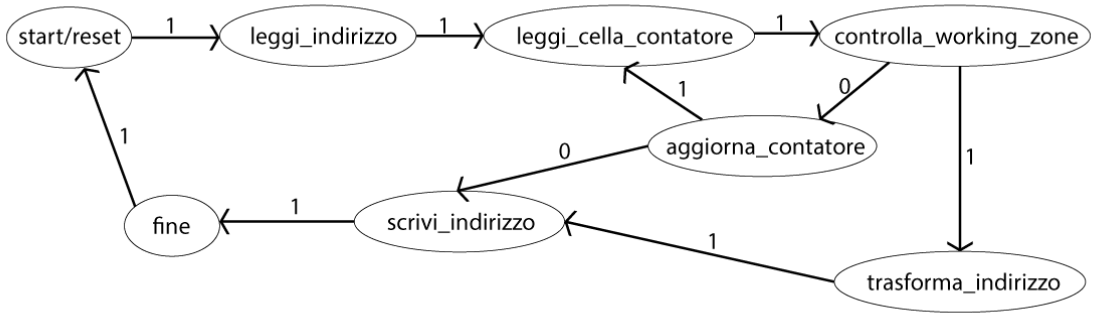
IEEE Transactions on Very Large Scale Integration (VLSI) Systems
(Volume: 6, Issue: 4, Dec. 1998)

In seguito viene presentato il progetto ed una proposta implementativa del modello di encoding di indirizzi basato su Working Zones orientata allo sviluppo e all'ottimizzazione hardware.

Progetto Finale di Reti Logiche

Studio analitico del progetto

Volendo dare una specifica formale al problema, si nota che può essere interpretato da una macchina a stati finiti, nella fattispecie:



Per dare un'idea dello sviluppo della computazione, immaginiamo di partire dallo stato "start/reset": tale stato permette di svolgere le operazioni preliminari, quali assicurare che i segnali di "start" e "done" siano coerenti, rimuovere informazioni da tutti i registri possibilmente impiegati in un precedente calcolo, ed inizializzare un'apposita variabile chiamata "contatore" al valore zero.

Procediamo, portando la macchina nello stato "leggi_indirizzo" per leggere da memoria il contenuto dell'indirizzo designato (che secondo i requisiti è la cella di indirizzo 8) e salvare in un apposito registro/variabile chiamata "indirizzo_da_valutare" il contenuto letto.

La macchina avanza allo stato "leggi_cella_contatore" che, come da nome, utilizza il valore della variabile "contatore" per per portarsi al corrispondente indirizzo in memoria dal quale verrà letto il contenuto e salvato in una variabile dedicata chiamata "contenuto_cella". Verrà inoltre predisposta una seconda variabile chiamata "contenuto_cella_dwz" che contiene la somma del valore di "contenuto_cella" e della codifica binaria di 4.

L'automa avanza nello stato "controlla_working_zone" che si occuperà della prima operazione decisionale descritta in questi termini:
se

$$\text{“}contenuto\_cella\text{”} \leq \text{“}indirizzo\_da\_valutare\text{”} < \text{“}contenuto\_cella\_dwz\text{”}$$

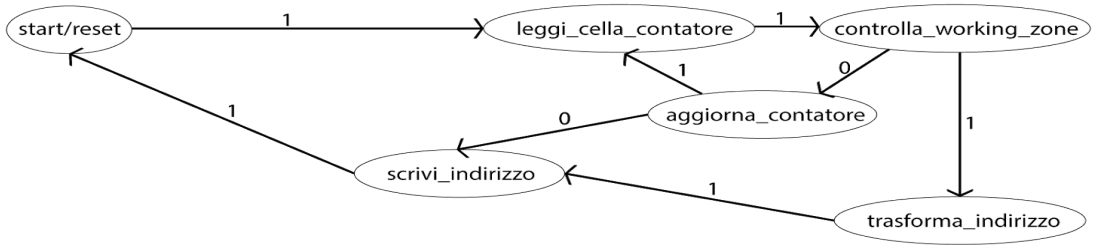
La computazione avanza nello stato "trasforma_indirizzo" descritto successivamente.
Altrimenti la macchina avanza nello stato "aggiorna_contatore" descritto in seguito.

Lo stato "aggiorna_contatore" si occupa di incrementare di 1 il valore della variabile "contatore" e di valutare una semplice operazione condizionale:
Se $\text{“}contatore\text{”} = 7$ (come da specifica progettuale) la macchina si occupa di copiare in una variabile chiamata "indirizzo_da_scrivere" il valore di "indirizzo_da_valutare" e raggiunge lo stato "scrivi_indirizzo" descritto successivamente.

In caso contrario la macchina raggiunge lo stato “leggi_cella_contatore”.

Lo stato “trasforma_indirizzo” si occupa di salvare in una variabile chiamata “indirizzo_da_scrivere” il valore ottenuto dalla computazione dell’indirizzo appartenente alla working zone secondo le specifiche di progetto.

Una volta raggiunto lo stato “scrivi_indirizzo” la macchina si occuperà di trascrivere all’indirizzo in posizione 9 della memoria il contenuto della variabile “indirizzo_da_scrivere” e la macchina procederà verso lo stato “fine” che si occupa di controllare la corretta sequenza dei segnali “start” e “done” per assicurare la conclusione della computazione e la correttezza del risultato. La macchina potrà eseguire un nuovo calcolo partendo nuovamente dallo stato “start/reset”.
Dalla precedente trattazione, utile a chiarire alcuni passaggi fondamentali per il buon esito della computazione ed una chiara comprensione di ogni passaggio, si evince che lo stato “leggi_indirizzo” è strettamente legato allo stato “start/reset” in quanto viene raggiunto come passaggio obbligato. Allo stesso modo “fine” è raggiunto obbligatoriamente in seguito al raggiungimento dello stato “scrivi_indirizzo”. Per questo motivo si può evitare di fare una distinzione esplicita tra gli stati legati tra loro ottenendo una macchina descritta come da immagine seguente:

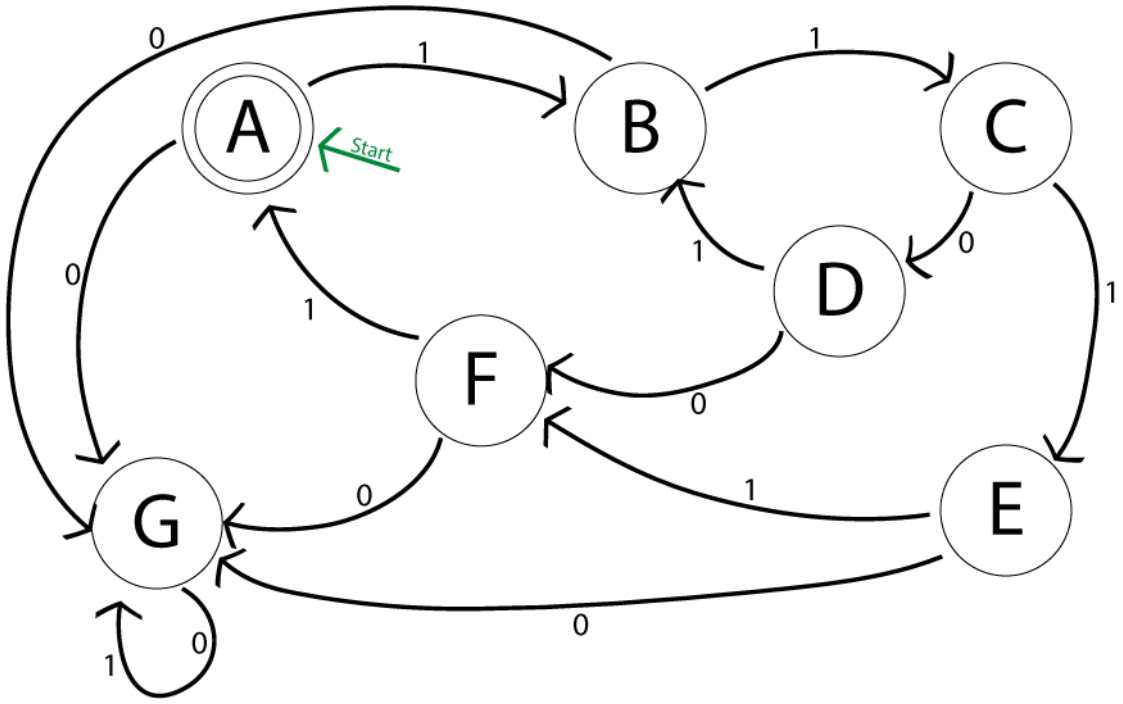


Ora è possibile definire uno stato di errore “errore” (non incluso nell’immagine per motivi di semplificazione della lettura) ed organizzare le operazioni che devono essere svolte in ogni stato in questo modo:

		IN=0	IN=1
A	start/reset	errore	leggi_cella_contatore
B	leggi_cella_contatore	errore	controlla_working_zone
C	controlla_working_zone	aggiorna_contatore	trasforma_indirizzo
D	aggiorna_contatore	scrivi_indirizzo	leggi_cella_contatore
E	trasforma_indirizzo	errore	scrivi_indirizzo
F	scrivi_indirizzo	errore	start/reset
G	errore	errore	errore

Con **stato iniziale e stato di accettazione:** “start/reset”.
Schema della macchina completa:

Progetto Finale di Reti Logiche



Risulta minima in quanto:

B	BC					
C	DG BE	DG CE				
D	FG	FG BC	DF BE			
E	BF	CF	DG EF	FG BF		
F	AB	AC	DG AE	FG AB	AF	
G	BG	CG	DG EG	FG BG	FG	AG
	A	**B**	**C**	**D**	**E**	**F**

Progetto Finale di Reti Logiche

Letterale	**in=0**	**in=1**		Binaria	**in=0**	**in=1**
A	G	B		**000**	110	001
B	G	C		**001**	110	010
C	D	E		**010**	011	100
D	F	B		**011**	101	001
E	G	F		**100**	110	101
F	G	A		**101**	110	000
G	G	G		**110**	110	110

Al fine di controllare più finemente l'architettura del componente da utilizzare senza rendere la macchina eccessivamente complessa, decido di sintetizzare una macchina a stati finiti usando tre flip-flop di tipo D: **D1, D2, D3** per rappresentare lo stato della macchina e la sua evoluzione rispetto all'ingresso **I**.

I	**Q1**	**Q2**	**Q3**		**D1=Q1***	**D2=Q2***	**D3=Q3***
0	0	0	0		1	1	0
0	0	0	1		1	1	0
0	0	1	0		0	1	1
0	0	1	1		1	0	1
0	1	0	0		1	1	0
0	1	0	1		1	1	0
0	1	1	0		1	1	0
1	0	0	0		0	0	1
1	0	0	1		0	1	0
1	0	1	0		1	0	0
1	0	1	1		0	0	1
1	1	0	0		1	0	1
1	1	0	1		0	0	0
1	1	1	0		1	1	0

Progetto Finale di Reti Logiche

Pongo D1, D2, D3 in funzione di (I, Q1, Q2, Q3) in questo modo:

$$D_1 = f_1(I, Q_1, Q_2, Q_3)$$
$$D_2 = f_2(I, Q_1, Q_2, Q_3)$$
$$D_3 = f_3(I, Q_1, Q_2, Q_3)$$

per trovare le espressioni minime adatte a rappresentare f_1, f_2, f_3 utilizzo il metodo delle mappe di karnaugh.

D1				
Q2-Q3 I-Q1	**00**	**01**	**11**	**10**
00	1	1	1	0
01	1	1	-	1
11	1	0	-	1
10	0	0	0	1

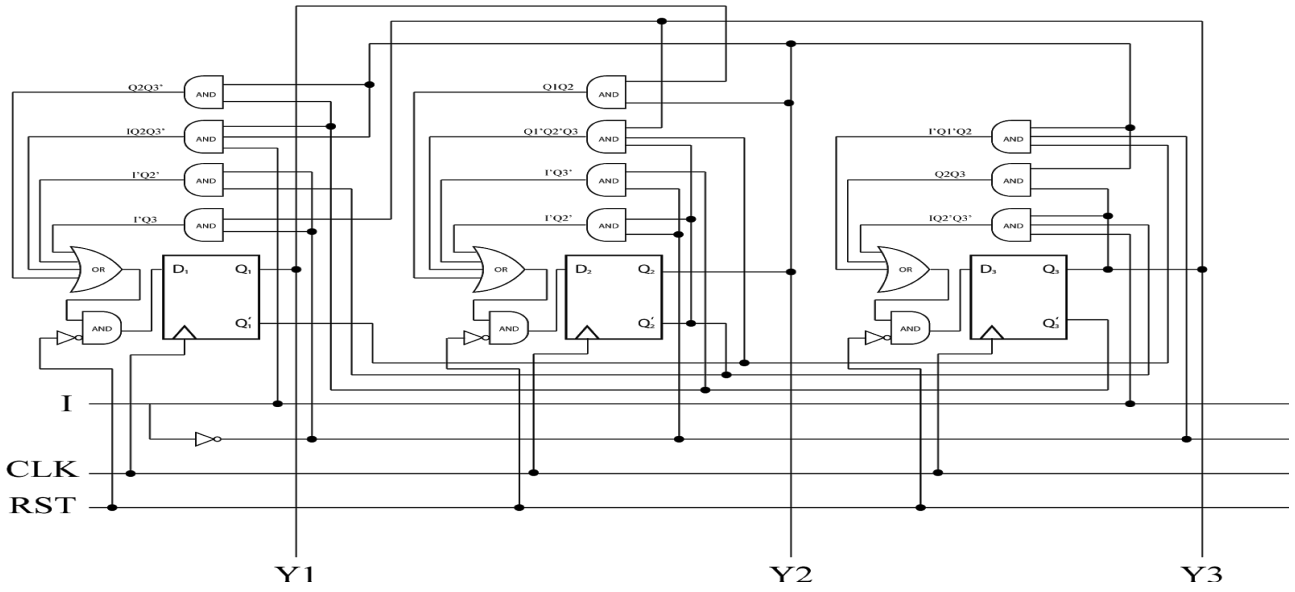
D2				
Q2-Q3 I-Q1	**00**	**01**	**11**	**10**
00	1	1	0	1
01	1	1	-	1
11	0	0	-	1
10	0	1	0	0

D3				
Q2-Q3 I-Q1	**00**	**01**	**11**	**10**
00	0	0	1	1
01	0	0	-	0
11	1	0	-	0
10	1	0	1	0

Da cui risulta:

$$D_1 = I'Q_2' + Q_2Q_3' + I'Q_3 + IQ_2Q_3'$$
$$D_2 = I'Q_2' + I'Q_3' + Q_1Q_2 + Q_1'Q_2'Q_3$$
$$D_3 = Q_2Q_3 + I'Q_1'Q_2 + IQ_2'Q_3'$$

Il progetto della macchina risultante è:



Progetto Finale di Reti Logiche

Si evince che la macchina descritta termina correttamente (non entra nello stato pozzo di errore e termina nello stato di accettazione) se e solo se riceve in ingresso un linguaggio del tipo:

$$1 \,.\, (1011)^{+} .\, 001$$

oppure

$$1 \,.\, (1011)^{+} .\, 111$$

in cui il punto indica l'operazione di concatenamento e l'operatore + indica che la stringa contenuta nel blocco di parentesi debba essere ripetuta almeno una volta, per indefinite volte (senza considerare i limiti che le specifiche di progetto prevedono).

Le uscite Y1, Y2, Y3 permettono ad altri componenti di conoscere lo stato della macchina in ogni istante. Tali componenti, a seconda dello stato della macchina si interfacciano con i registri per leggere il contenuto della memoria ed eseguire operazioni (descritte successivamente) che controllano l'ingresso I che governa la macchina a stati.

Le variabili (registri) necessarie, che concorrono alla definizione dello stato del componente, sono:

Nome	Descrizione	#bit
indirizzo_da_valutare	valore contenuto nella cella #8 secondo specifiche	8 da specifiche
contatore	necessario a scorrere gli indirizzi in maniera incrementale conoscendone sempre la posizione	3 si prevede di contare da 0 a 7
contenuto_cella	necessario per la valutazione che riguarda l'appartenenza o meno alla WZ	8 da specifiche
offset	contiene codifica one-hot di sottrazione	4 da specifiche one-hot
indirizzo_da_scrivere	variabile che contiene l'indirizzo da copiare nella cella #9	8 da specifiche
WZ_BIT	contiene il valore "booleano" di appartenenza o meno ad una working zone	1 "true o false"

Progetto Finale di Reti Logiche

Le operazioni di lettura, scrittura ed elaborazione dei dati si possono schematizzare nella seguente tabella:

Nome	Uscita	Descrizione
start/reset	1	Se il segnale **done** = 0 e **start** = 1 copia contenuto dell'indirizzo #8 in **indirizzo_da_valutare**. **contatore, contenuto_cella, indirizzo_da_scrivere = 0.**
leggi_cella_contatore	1	Legge il valore di **contatore** e si porta all'indirizzo di memoria letto, copia il contenuto dell'indirizzo in **contenuto_cella.**
Controlla_working_zone	**r**: 1 bit	Controlla l'appartenenza di **indirizzo_da_valutare** all'intervallo: **[contenuto_cella, contenuto_cella + 4)**: mette **WZ_BIT** a 1 se **indirizzo_da_valutare** è compreso(Condizione "A"), 0 altrimenti.
Trasforma_indirizzo	1	Calcola **offset =** one-hot(**indirizzo_da_valutare - contenuto_cella**). Concatena: **WZ_BIT** & **contatore** & **offset** e salva il vettore di 8 bit ottenuto in **indirizzo_da_scrivere.**
Aggiorna_contatore	**r**: 1 bit	se **n** = '**111**' restituisce **0** altrimenti Incrementa di 1 il valore di **contatore**, salva il nuovo valore nella variabile **contatore** e imposta l'uscita a **1** (Condizione "B").
Scrivi_indirizzo	**1**	Scrive il valore di **indirizzo_da_scrivere** nella cella di memoria #9. Controlla adeguatamente i segnali di **done** e **start**.
Errore	**1**	Porta un segnale di errore ad uno (che evita che il segnale di "done" venga portato ad 1).

Progetto Finale di Reti Logiche

Hardware Definition Language

01. Introduzione

Per ragioni legate all'estrema difficoltà di sintetizzare manualmente la macchina precedentemente descritta, la definizione e sintetizzazione del componente verrà delegata ai software di sintesi.
Oltre che facilitare la dichiarazione, l'utilizzo e la gestione dei registri permettono di definire la funzionalità del componente usando contemporaneamente la "strategia" strutturale e comportamentale.
Per quanto riguarda la dichiarazione dei registri e degli stati della macchina, che rappresentano la gran parte del lavoro da realizzare, verrà utilizzata la metodologia comportamentale o **behavioural**.
Verrà dunque valutata la possibilità di utilizzare logiche per ridurre la complessità del componente attraverso l'uso di architetture ad hoc definite con metodologia **structural**.

02. Specifiche

Il componente da sintetizzare sarà concepito come una FSM che si interfaccia alla macchina (fornita dai test-bench) dotata di RAM.
Per semplificare la stesura della routine di aggiornamento dello stato del componente, si può prevedere un cambiamento di stato ad ogni periodo/ciclo di clock.
In tal senso sarebbe utile concepire due processi che si occupano l'uno di impostare il cambio di stato al fronte di salita del clock e l'altro di aggiornare i segnali che definiscono lo stato corrente e quello successivo e che vengono sincronizzati al fronte di discesa del clock il tutto seguirebbe il paradigma **master-slave**.
Sfruttando la tecnologia process si può concepire questa struttura in tal senso:

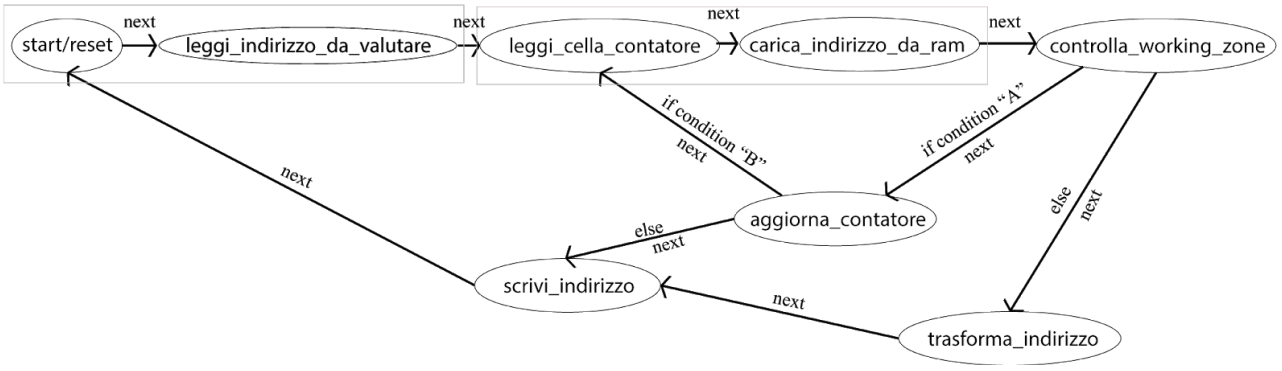
```
next_state_process: process(i_clk, i_rst)
      begin
      if(i_rst= '1') then
            current_state <= start_reset;
      elsif(rising_edge(i_clk)) then
            current_state <= next_state;
      end if;
end process;

Fsm: process(i_clk, i_start)
      begin
      if(i_start = '0') then o_done <= '0';
      elsif(falling_edge(i_clk) and i_start= '1') then
            case current_state is

            when start_reset =>
                  . . .
                  . . .
```

Progetto Finale di Reti Logiche

Al fine di attenersi ai requisiti di progetto si evince che lo scambio di segnali con la ram si svolge in fasi distinte:

- Il componente porta il segnale di uscita `o_address` al valore che rappresenta l'indirizzo della cella di memoria della quale vuole ricevere il contenuto.
- La macchina porta il segnale di uscita `mem_o_data` al valore contenuto dalla cella di memoria dell'indirizzo ricevuto. Il segnale mem_o_data viene portato all'ingresso del componente attraverso il segnale dell'interfaccia `i_data`.

Queste operazioni richiedono **un periodo** di clock ciascuna.
Siccome il componente deve richiedere dati alla ram macchina in maniera iterativa, al fine di sincronizzare il processo, si decide di espandere gli stati della FSM in modo tale da ridurre la complessità della progettazione della macchina mantenendo il paradigma che prevede il "cambio di stato" ad ogni ciclo di clock.
Queste considerazioni portano ad una concezione finale della FSM illustrata in seguito.



Le condizioni "A" e "B" sono state esaustivamente trattate precedentemente.

Progetto Finale di Reti Logiche

03. Segnali

I tipi ed i segnali utilizzati si attengono alle specifiche descritte precedentemente:

```
type machine_state is (
      start_reset,
      leggi_indirizzo_da_valutare,
      leggi_cella_contatore,
      carica_indirizzo_da_ram,
      controlla_working_zone,
      aggiorna_contatore,
      trasforma_indirizzo,
      scrivi_indirizzo
      );

      signal current_state, next_state : machine_state;

      signal indirizzo_da_valutare : UNSIGNED(7 downto 0) := "00000000";
      signal contatore : UNSIGNED (2 downto 0) := "000";
      signal contenuto_cella : UNSIGNED (7 downto 0) := "00000000";
      signal indirizzo_da_scrivere : STD_LOGIC_VECTOR(7 downto 0) :=
     "00000000";
      signal wz_bit : STD_LOGIC := '0';

      constant DWZ : unsigned(2 downto 0) := "100";
```

Con l'aggiunta della costante DWZ di valore 4.

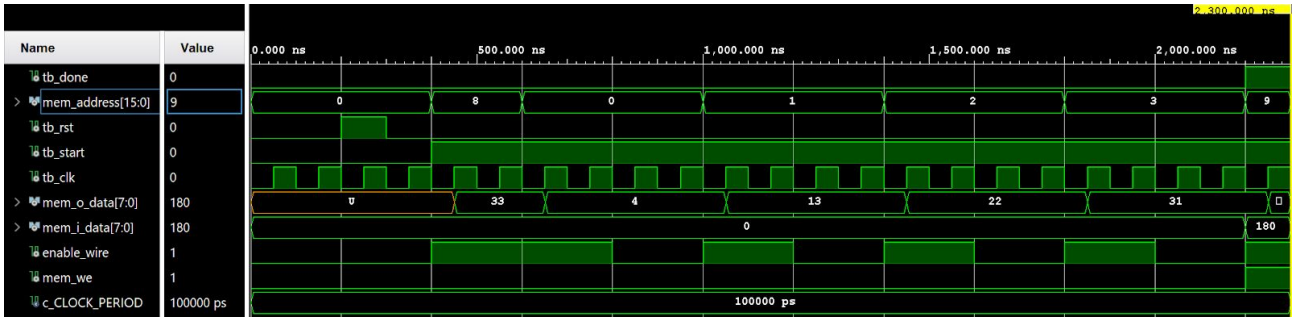
04. Implementazione

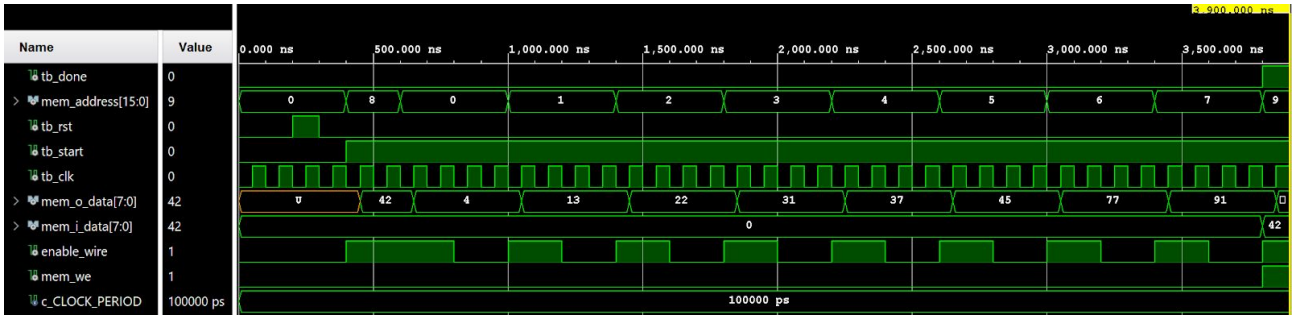
L'implementazione del codice è presente nel file .vhd presentato assieme alla documentazione.
Si è fatto uso di due processi distinti, il primo chiamato `next_state_process` che governa il cambiamento di stato della macchina ad ogni ciclo di clock, a patto che il segnale di reset sia congruo alle specifiche di funzionamento del componente, la sensitivity list del processo è formata appunto dal segnale di reset `i_rst` ed il clock `i_clk`.
`next_state_process` ha l'unico scopo di cambiare stato ad ogni fronte di salita del periodo di clock, facendo uso della variabile `next_state`.
Il secondo processo, chiamato `Fsm`, si occupa di eseguire operazioni di lettura e scrittura dei registri e della ram della macchina, che sono ovviamente diverse a seconda dello stato corrente della macchina, memorizzato nella variabile `current_state`.
`Fsm` è programmato per essere sensibile ai cambiamenti dei segnali di start `i_start` e del clock (sul fronte di discesa) `i_clk`, che fanno parte appunto della sensitivity list.
Alla fine dell'esecuzione della routine prevista da ogni stato, c'è sempre un'istruzione per il cambio di stato che prevede di memorizzare in `next_state`, il valore dello stato successivo come precedentemente spiegato.

Progetto Finale di Reti Logiche

05. Test e analisi della complessità pre-sintesi

Il codice di definizione del componente produce una simulazione pre-sintesi in accordo con i risultati aspettati. Si veda il grafico delle eccitazioni prodotto dall'esecuzione della simulazione per i casi di test forniti:





I test sono stati superati ed il componente si è comportato come previsto.

Al fine di affrontare casi di test più complessi (casi limite) e assicurare una corretta interpretazione dei dati conviene fare delle considerazioni:

- L'indirizzo da codificare non può assumere valori maggiori di 127
- L'indirizzo di valore "più grande" che si può ottenere dalla codifica è 1-111-1000 ovvero 248, si può usare un qualsiasi valore più grande per codificare un errore.
  Si userà il valore 1111-1111 o 255.
- La codifica del numero è **indipendente** dal valore iniziale contenuto in memoria.

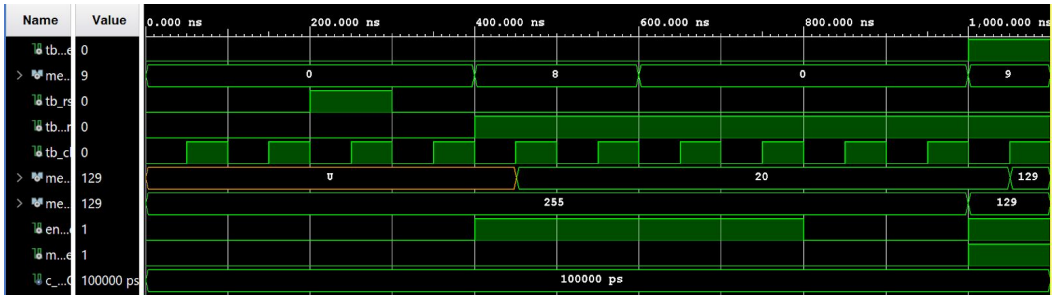
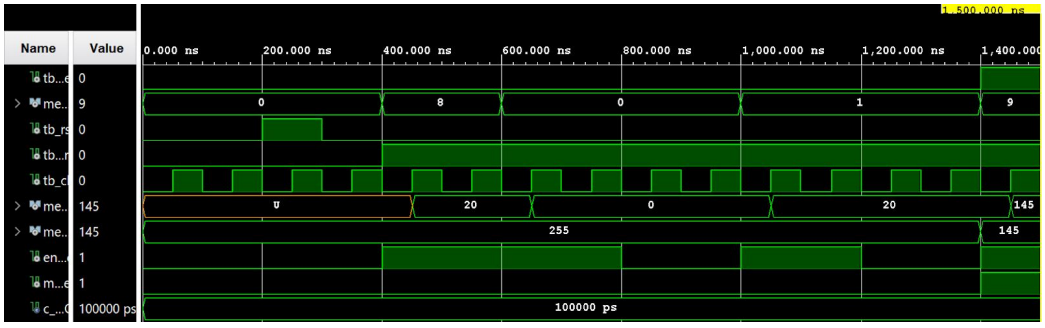
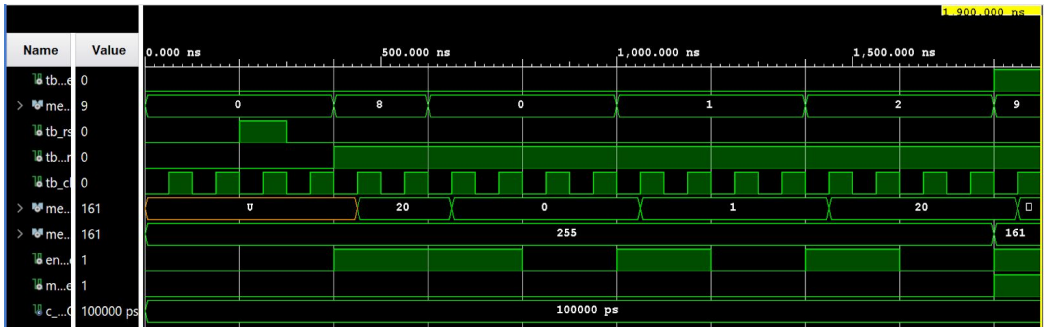
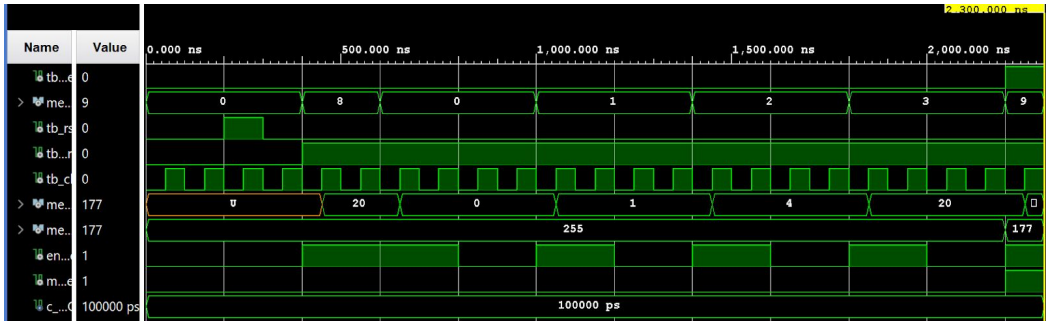
Progetto Finale di Reti Logiche

06. Test e analisi complessità post-sintesi

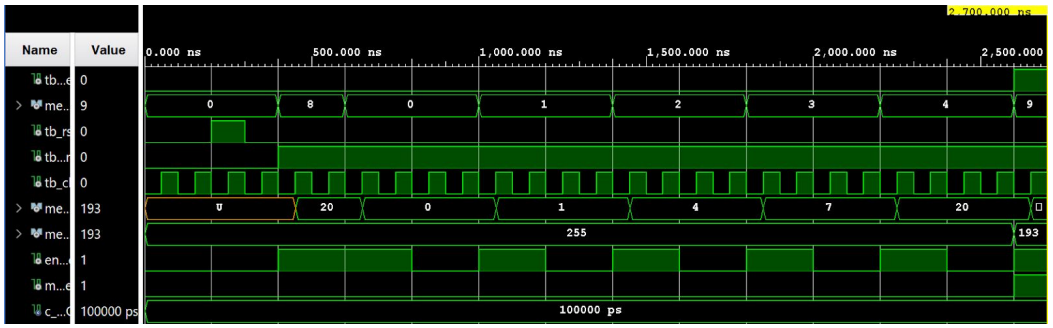
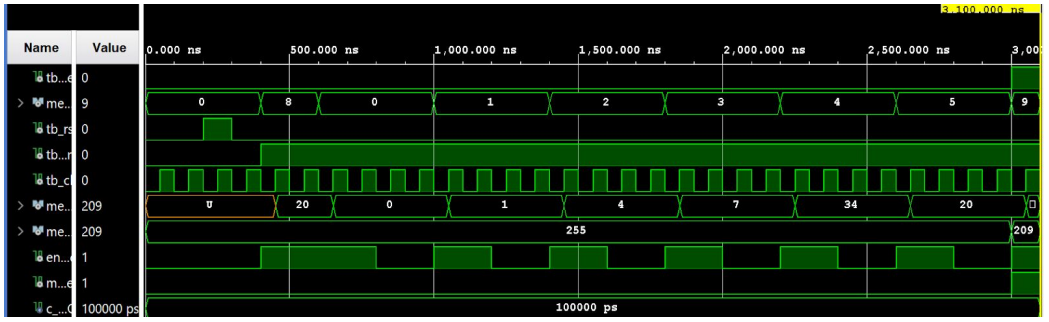
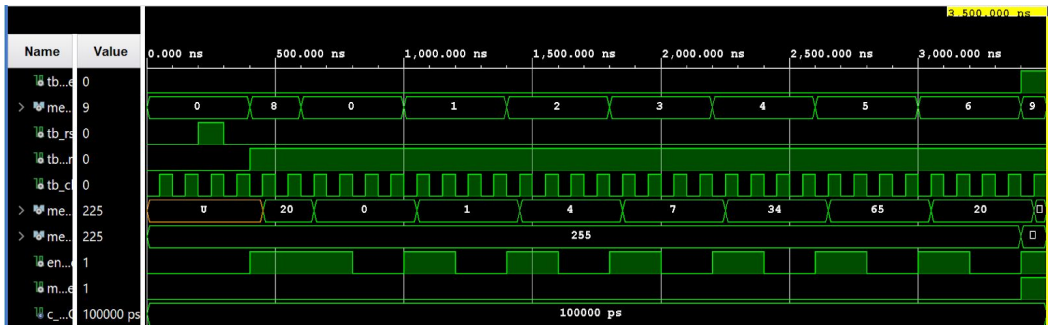
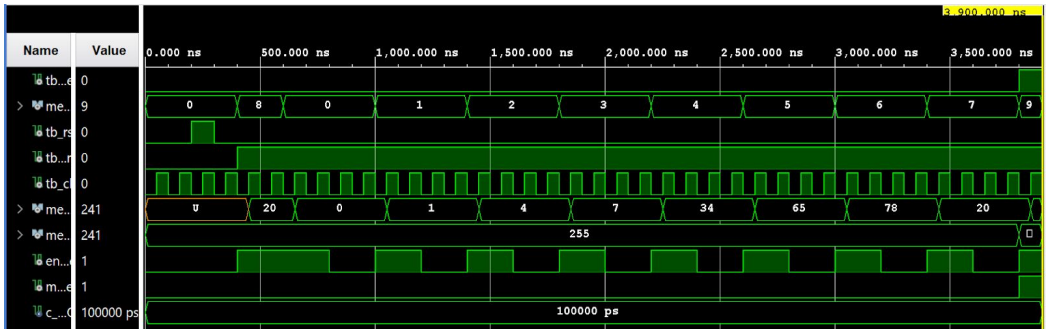
I test-bench forniti risultano simulabili senza errori post sintesi.
I casi che si andranno a testare sono riportati in seguito.
Tutti i test sono stati eseguiti con **cicli di clock** pari a **100ns.**

- **Valore presente in ogni cella**

Per assicurare il corretto funzionamento è testata la presenza di un valore da codificare in ogni cella

presente in 0 **1100 ns**	
presente in 1 **1500 ns**	
presente in 2 **1900 ns**	
presente in 3 **2300 ns**	

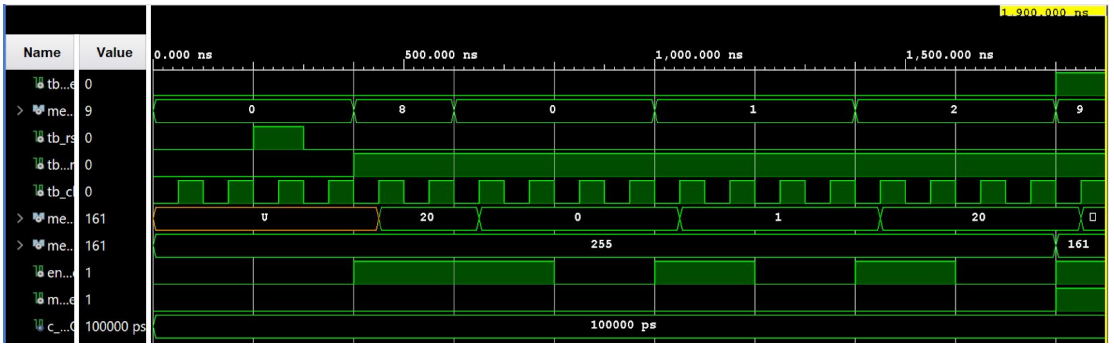
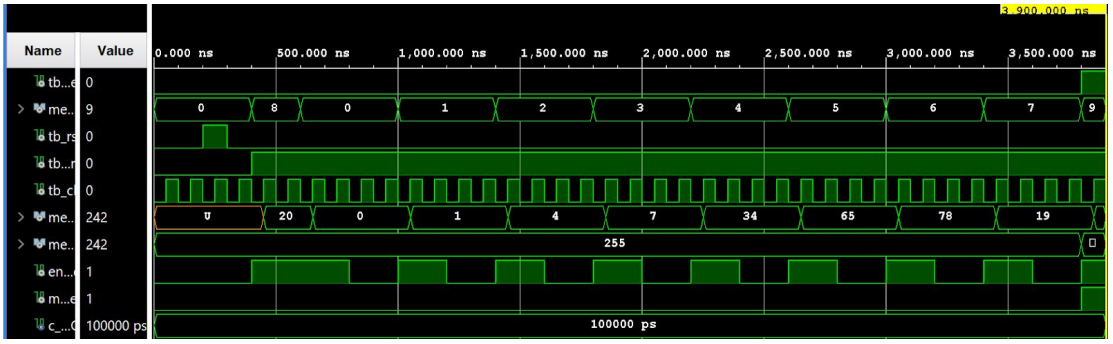
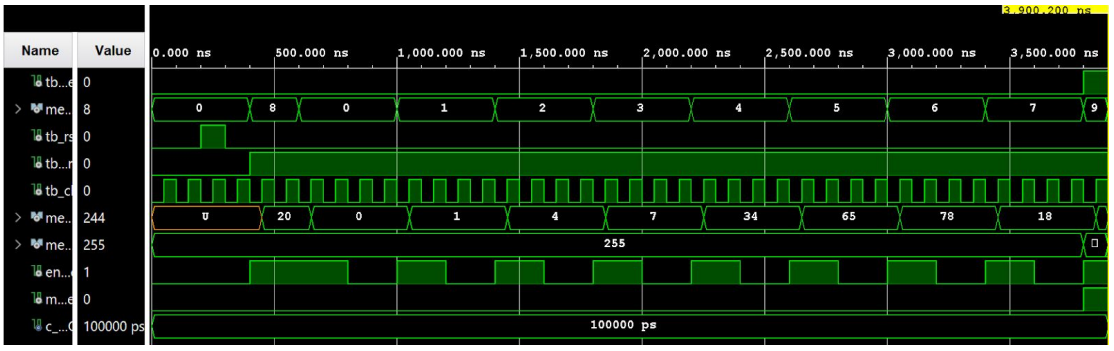
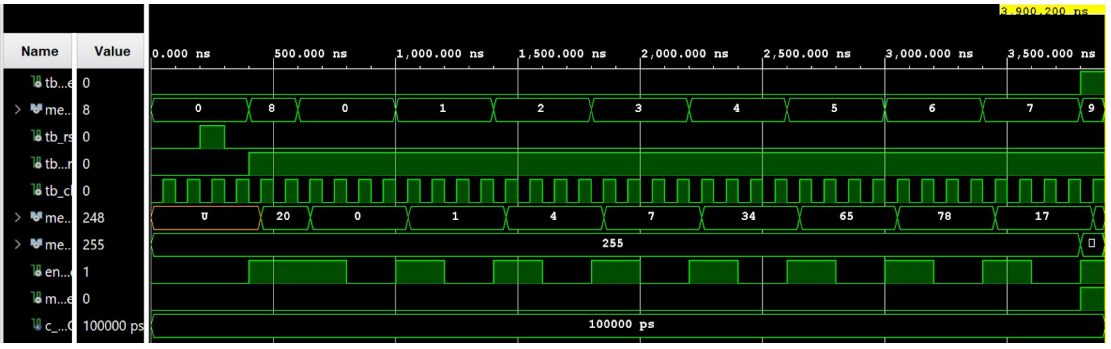
Progetto Finale di Reti Logiche

presente in 4 **2700 ns**	
presente in 5 **3100 ns**	
presente in 6 **3500 ns**	
presente in 7 **3900 ns**	

Si evince che il componente sintetizzato, seguendo lo schema architetturale, legge la ram in maniera progressiva e quindi impiega un tempo maggiore se l'indirizzo da codificare non corrisponde a nessuna WZ o se appartiene all'ultima WZ.
Si può affermare che il comportamento della macchina a stati finiti ha una complessità temporale proporzionale al numero di WZ o NWZ. In condizioni di corretto funzionamento il componente impiega un tempo compreso tra **1100 ns** e **3900 ns** a seconda della posizione della WZ da codificare. Il tempo di elaborazione dei segnali potrebbe essere migliorato col presupposto che gli indirizzi delle WZ siano ordinati (condizione non specificata), in questo caso si potrebbe ottenere un tempo proporzionale a $log(n)$ secondo i classici algoritmi di ricerca in liste ordinate.

Progetto Finale di Reti Logiche

- **Valore presente in ogni Working Zone**

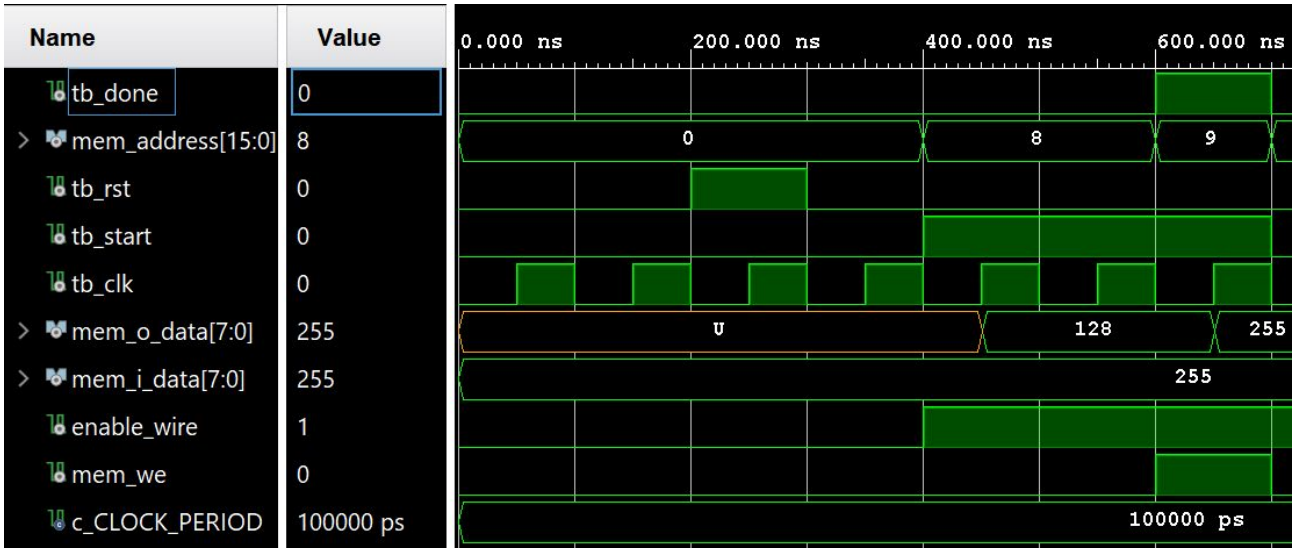
WZ = 0 (offset)	
WZ = 1 (offset)	
WZ = 2 (offset)	
WZ = 3 (offset)	

Il componente risponde adeguatamente codificando l'indirizzo correttamente ad ogni offset.
La codifica **one-hot** funziona come da specifiche.

- **Valore da codificare maggiore di 127 (NON CONSENTITO)**

in questo caso il componente interrompe la lettura ed entra immediatamente nello stato “scrivi indirizzo” scrivendo in posizione #9 della ram il valore 255 (non ammesso per ragioni di specifiche).
Vengono allegati i risultati:

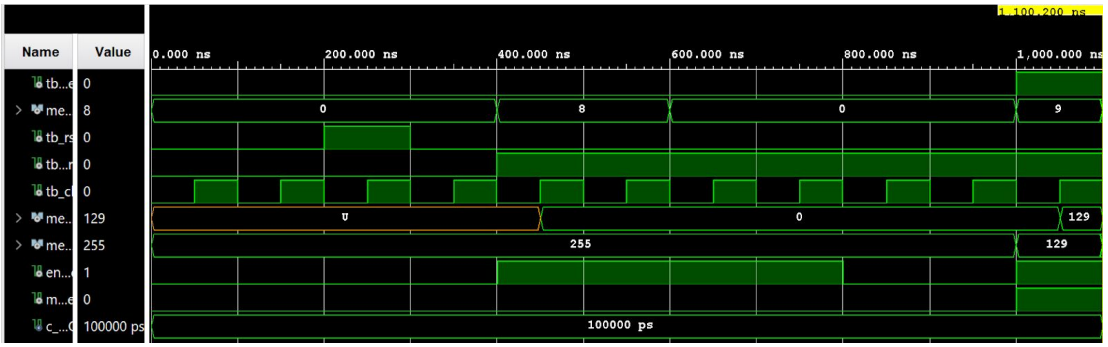
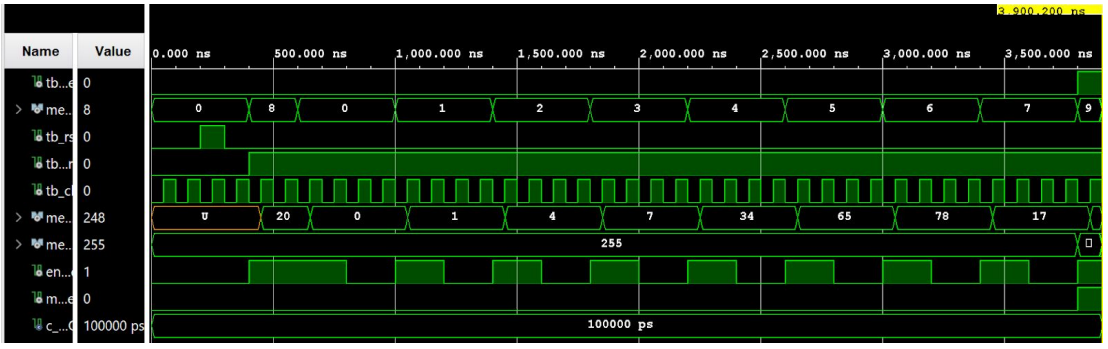
```
signal RAM: ram_type := (0 => std_logic_vector(to_unsigned( 4 , 8)),
                         1 => std_logic_vector(to_unsigned( 1 , 8)),
                         2 => std_logic_vector(to_unsigned( 11 , 8)),
                         3 => std_logic_vector(to_unsigned( 12 , 8)),
                         4 => std_logic_vector(to_unsigned( 37 , 8)),
                         5 => std_logic_vector(to_unsigned( 45 , 8)),
                         6 => std_logic_vector(to_unsigned( 77 , 8)),
                         7 => std_logic_vector(to_unsigned( 124 , 8)),
                         8 => std_logic_vector(to_unsigned( 128 , 8)),
                others => (others =>'0'));
```



si nota che il valore da scrivere rimane costantemente 255 in quanto è valore di default del signal.
In questo caso la computazione si interrompe a **600 ns.**

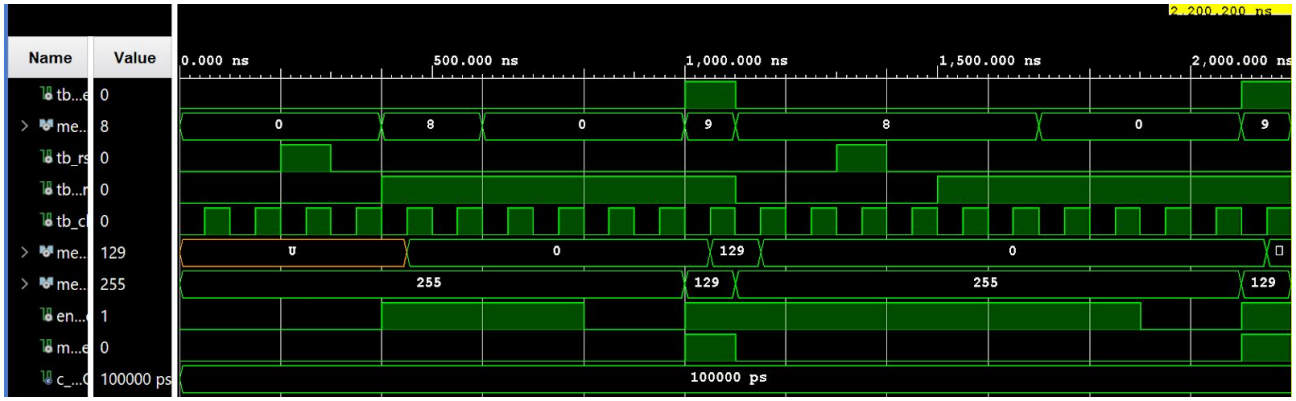
Progetto Finale di Reti Logiche

- **Valori da codificare agli “estremi” (casi limite)**

Caso **129**	
Caso **248**	

Il range di valori codificabili è stato ampiamente testato ed i risultati sono stati sempre coerenti con le specifiche, i casi limite hanno avuto esito positivo.

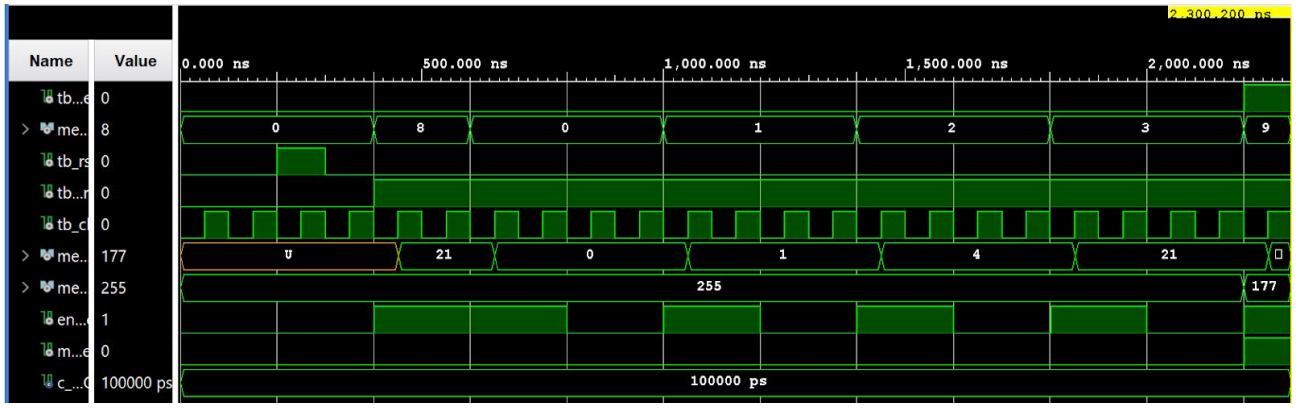
- **Letture ripetute da RAM**



Il componente sintetizzato risponde correttamente alle specifiche di fine lettura da ram manipolando in maniera adeguata i segnali di **start** e **done**. Di fatto il componente esegue indefinite volte l’operazione in quanto i test forniti lo prevedono, i segnali ed i risultati sono coerenti con le specifiche richieste.

- **Indirizzo presente più di una volta (multiple WZ)**

```
signal RAM: ram_type := (0 => std_logic_vector(to_unsigned( 0 , 8)),
                         1 => std_logic_vector(to_unsigned( 1 , 8)),
                         2 => std_logic_vector(to_unsigned( 4 , 8)),
                         3 => std_logic_vector(to_unsigned( 21 , 8)),
                         4 => std_logic_vector(to_unsigned( 19 , 8)),
                         5 => std_logic_vector(to_unsigned( 65 , 8)),
                         6 => std_logic_vector(to_unsigned( 78 , 8)),
                         7 => std_logic_vector(to_unsigned( 17 , 8)),
                         8 => std_logic_vector(to_unsigned( 21 , 8)),
             others => (others =>'0'));
```



Come è stato trattato precedentemente, l'interpretazione del progetto ha portato alla sintesi di un componente che esegue una routine di ricerca. Come previsto dagli algoritmi di ricerca in un insieme non necessariamente ordinato ed in cui ci possono essere elementi con molteplicità maggiore di uno, il componente ha un tempo di risoluzione proporzionale alla dimensione della lista di indirizzi che deve leggere. In questo modo, se erroneamente nella struttura della ram ci fossero indirizzi che appartengono potenzialmente a più working zones (codifica errata in ram), verrà codificato il primo presente in ram. Le immagini precedenti documentano la situazione illustrata.

Tutti i test sono stati eseguiti post sintesi.

Conclusioni

Lo strumento di sintesi ed implementation ha portato al seguente report:

Tcl Console | Messages | Log | Reports | **Design Runs**

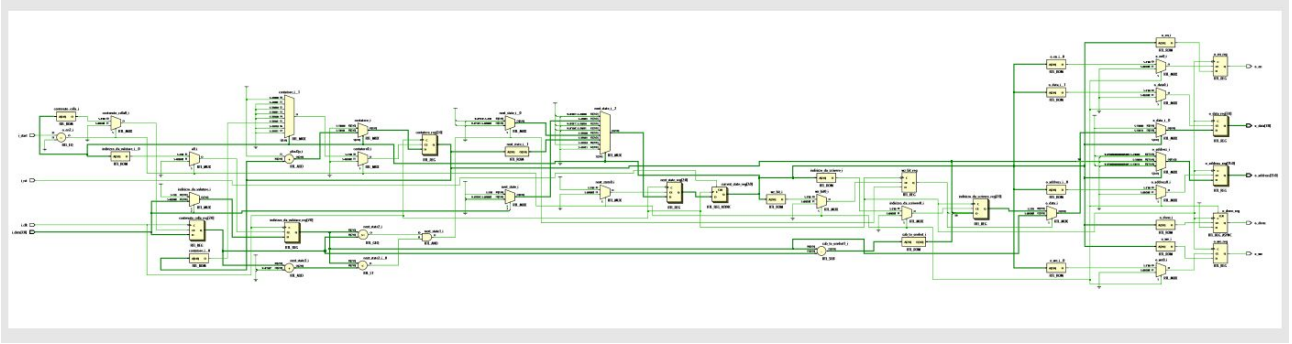
Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAM	URAM	DSP	Start
✓ synth_1	constrs_1	synth_design Complete!								53	59	0.0	0	0	6/7/20, 1
✓ impl_1	constrs_1	route_design Complete!	NA	NA	NA	NA	NA	6.135	0	53	59	0.0	0	0	6/7/20, 1

Progetto Finale di Reti Logiche

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	53	0	134600	0.04
LUT as Logic	53	0	134600	0.04
LUT as Memory	0	0	46200	0.00
Slice Registers	59	0	269200	0.02
Register as Flip Flop	59	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

Ref Name	Used	Functional Category
FDRE	31	Flop & Latch
OBUF	27	IO
LUT2	22	LUT
LUT4	20	LUT
FDSE	19	Flop & Latch
IBUF	11	IO
LUT3	9	LUT
FDCE	8	Flop & Latch
LUT6	6	LUT
CARRY4	5	CarryLogic
LUT5	4	LUT
LUT1	1	LUT
FDPE	1	Flop & Latch
BUFG	1	Clock

Si nota che durante il processo di ottimizzazione sono stati utilizzati solo LUT logici e solo registri di tipo Flip-flop.
Lo schema del componente si presenta come segue:



La versione ad alta risoluzione dello schema implementativo si può trovare all'indirizzo:

https://www.stefanomarzo.it/rl/schematic.pdf