Prova Finale (Progetto di Reti Logiche)

Prof. Fabio Salice – Anno 2019/2020

Arianna Galzerano (Codice Persona 10563365 – Matricola 886979)

Indice:

1 Introduzione

Scopo del progetto

Lo scopo del progetto è sintetizzare un componente hardware, descritto in VHDL, che implementi una versione del metodo di codifica delle "Working-Zone", pensato per diminuire la dissipazione di potenza. Esso è utilizzato dai bus di indirizzi per trasformare un indirizzo quando viene trasmesso e appartiene a certi intervalli; sono questi intervalli di indirizzi di dimensione fissa ad essere chiamati working-zone.

Per la realizzazione del componente sono stati utilizzati: - XILINX VIVADO WEBPACK;
- Target FPGA xc7a200tfbg484-1.

Specifiche generali

Si può rappresentare in modo riassuntivo il funzionamento di base del componente attraverso un numero finito di passi da seguire (che saranno poi gli stati della MSF):

1. Reset e attesa del segnale di start.
2. Ricevuto l'indirizzo da trasmettere, se ne salva il valore in un registro.
3. Si legge il contenuto del primo indirizzo della RAM (0) e si scorrono gli indirizzi della RAM in cui sono salvati i valori delle WZ, facendo il confronto per ciascun valore se l'indirizzo dato faccia parte della working-zone.
4. Se viene rispettata la condizione di verifica e dunque l'indirizzo appartiene a una wz esso viene codificato in questo modo: il bit addizionale *WZ_BIT* è posto a 1, mentre i bit di indirizzo vengono divisi in 2 sotto campi rappresentanti: Il numero della working-zone al quale l'indirizzo appartiene (*WZ_NUM*), codificato in binario; l'offset rispetto all'indirizzo di base della working-zone (*WZ_OFFSET*), codificato come one-hot (cioè il valore da rappresentare è equivalente all'unico bit a 1 della codifica).
5. Se l'indirizzo da trasmettere non appartiene a nessuna working-zone, esso viene trasmesso così come è, e un bit addizionale rispetto ai bit di indirizzamento (*WZ_BIT*) viene messo a 0.
6. Il valore (indirizzo) codificato o meno viene poi salvato nell'indirizzo della RAM successivo a dove era presente il valore dell'ultima working-zone.

Nella versione da implementare il numero di bit da considerare per l'indirizzo da codificare è 7. Il che definisce come indirizzi validi quelli da 0 a 127.
Il numero di working-zone è 8 mentre la dimensione della working-zone è 4 indirizzi incluso quello base. Questo comporta che l'indirizzo codificato sarà composto da 8 bit: 1 bit per *WZ_BIT* + 7 bit per *ADDR*, oppure 1 bit per *WZ_BIT*, 3 bit per codificare in binario a quale tra le 8 working zone l'indirizzo appartiene, e 4 bit per codificare one-hot il valore dell'offset di ADDR rispetto all'indirizzo base.
Il modulo da implementare leggerà l'indirizzo da codificare e gli 8 indirizzi base delle working- zones e dovrà produrre l'indirizzo opportunamente codificato.

Interfaccia del componente

Il componente da descrivere ha un'interfaccia così definita:

```
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;
```

In particolare:

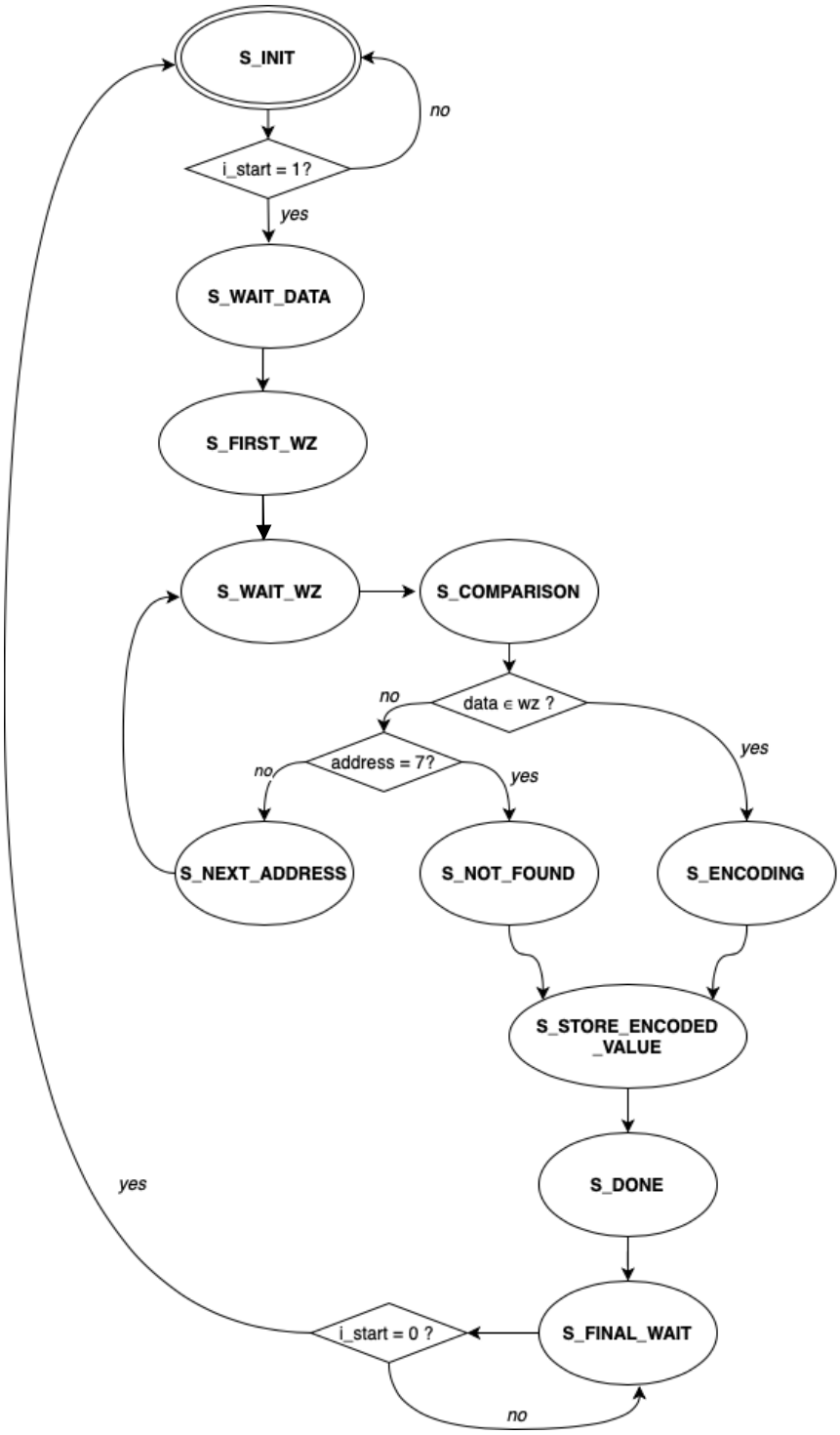
- i_clk è il segnale di CLOCK in ingresso generato dal test bench;
- i_start è il segnale di START generato dal test bench;
- i_rst è il segnale di RESET che inizializza la macchina pronta per ricevere il primo segnale di START;
- i_data è il segnale (vettore) che arriva dalla memoria in seguito ad una richiesta di lettura;
- o_address è il segnale (vettore) di uscita che manda l'indirizzo alla memoria;
- o_done è il segnale di uscita che comunica la fine dell'elaborazione e il dato di uscita scritto in memoria;
- o_en è il segnale di ENABLE da dover mandare alla memoria per poter comunicare (sia in lettura che in scrittura);
- o_we è il segnale di WRITE ENABLE da dover mandare alla memoria (=1) per poter scriverci. Per leggere da memoria esso deve essere 0;
- o_data è il segnale (vettore) di uscita dal componente verso la memoria.

Per quanto riguarda la memoria, nella specifica viene sottolineato che essa è già istanziata all'interno del Test Bench e non va sintetizzata. Inoltre, essa è derivata dalla User guide di VIVADO disponibile al seguente link: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_3/ug901-vivado- synthesis.pdf

2 Architettura

Il funzionamento alla base del componente è stato implementato attraverso una FSM che usa come segnale di ingresso *i_start*. Gli stati utilizzati sono in totale 11.

Figura 1: Macchina a Stati Finiti Implementata



Stati della FSM

S_INIT: stato iniziale, in questo stato si attende finchè non viene alzato *i_start* e cioè è possibile ricevere l'indirizzo da codificare. L'address da cui viene letto il valore in memoria è il Generic *ADDRESS_WITH_VALUE*, ovvero RAM(8) da specifica. Torno qui quando i*_rst* viene alzato. Si va poi in *S_WAIT_DATA*;

S_WAIT_DATA: stato in cui si attende la risposta dalla memoria in seguito alla richiesta del dato. Si passa poi allo stato *S_FIRST_WZ*;

S_FIRST_WZ: stato in cui viene inserito il primo address di memoria in cui è contenuto il valore di una wz, da specifica RAM(0), e la lettura è abilitata. In questo stato inoltre viene salvato l'indirizzo da codificare in *data*. Si passa poi a *S_WAIT*;

S_WAIT_WZ: stato in cui si attende che la memoria risponda dopo aver richiesto il valore di una wz. Si passa poi a *S_COMPARISON*;

S_COMPARISON: questo è lo stato in cui avviene l'accertamento di appartenenza o meno dell'indirizzo da codificare a una wz. Si svolge il confronto tra questo valore e quello della wz appena ottenuto dalla RAM; nel caso di mancato soddisfacimento della condizione di appartenenza si passa allo stato *S_NEXT_MEM*, nel caso invece di soddisfacimento si settano correttamente *wz_bit, wz_num e wz_offset* che andranno a costituire l'indirizzo codificato e si passa allo stato *S_ENCODING*; finite le wz si passa allo stato *S_NOT_FOUND*;

S_NEXT_ADDRESS_MEMORY: stato in cui avviene l'iterazione dell'address da cui leggere nella RAM per ottenere il successivo valore della wz salvato in memoria, fino a che l'address non arrivi a 8, da qui si ritorna allo stato *S_WAIT_WZ*;

S_ENCODING: stato in cui avviene la codifica e concatenamento per creare il nuovo indirizzo codificato da inserire in RAM(8). Si passa poi a *S_DONE*;

S_NOT_FOUND: stato in cui viene unicamente settato il primo bit dell'address a 0, lasciando la restante parte dell'indirizzo che doveva essere codificato uguale poiché non è stato trovato un'appartenenza a una delle wz proposte. Si passa a *S_DONE*;

S_DONE: stato in cui il nuovo valore dell'indirizzo fornito inizialmente (codificato o meno) viene salvato in memoria, setto *o_done* a 1 e passo a *S_FINAL_WAIT*;

S_FINAL_WAIT: stato in cui attendo che il segnale *i_start* si abbassi per poi tornare ad S_INIT;

Scelte Progettuali

La principale scelta progettuale effettuata è stata quella di descrivere il componente con un unico processo:
in esso viene gestita sia la parte sequenziale della macchina riguardante i registri e come essi vengono manipolati, che la parte riguardante la FSM e cioè l'evoluzione dello stato corrente a seconda dei segnali in ingresso.
Le operazioni logiche utilizzate per determinare il risultato finale sono l'AND del concatenamento per la creazione del dato codificato e l'ADD per incrementare l'indirizzo di memoria in cui leggere il valore della wz.
Nonostante il numero degli stati della FSM realizzata non sia eccessivamente elevato, l'obiettivo principale del mio progetto, oltre a creare un design funzionante che rispetti le specifiche, è stato piuttosto quello di realizzare un codice lineare e ordinato, senza ripetizioni e di facile manutenzione e comprensione, dando quindi una singola funzione a ciascuno stato. Le variabili inserite nel processo hanno anch'esse una funzione specifica, in particolare:

-***data***: salva il dato proveniente dalla memoria e restituisce il valore a fine elaborazione codificato da inserire in RAM(8);
-***wz_num, wz_offset, wz_bit***: rappresentano le tre parti in cui il dato codificato deve essere suddiviso. Esse vengono assegnate nello stato addetto alla codifica e poi concatenate per formare il dato codificato da inserire in memoria;
-***valdata***: integer usato per convertire il data (stdlogic) in integer e per poter svolgere il confronto;
-***valconfronto***: integer usato per convertire il valore della wz (stdlogic) in integer e per poter svolgere il confronto;
-***address***: contiene l'address della memoria da cui voglio leggere/scrivere a seconda dello stato;

Ho inoltre deciso di utilizzare un approccio che prevedesse di mantenere memorizzate meno informazioni possibili. Il valore di ciascuna working-zone viene salvato solamente per il confronto con l'indirizzo da codificare correntemente e sovrascritta con il valore della working-zone successiva.
Una scelta progettuale riguardante una possibile estendibilità del progetto è stata fatta inserendo due Generic, in modo da facilitare l'aumento di numero di wz da confrontare con il valore dell'indirizzo dato per verificarne l'appartenenza. Uno per indicare l'address della RAM dove è presente il dato iniziale da codificare *(ADDRESS_WITH_VALUE),* n.8 nella specifica, e uno per indicare l'address dove va salvato il dato dopo la codifica *(ADDRESS_WITH_ENCODED_VALUE),* n.9 nella specifica.

3 Risultati Sperimentali

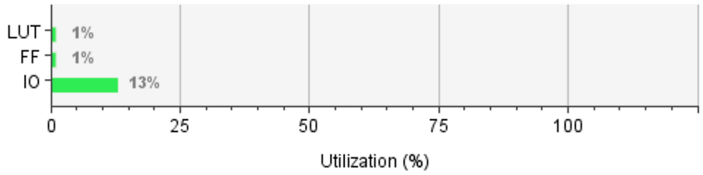
Il componente sintetizzato supera correttamente tutti i test specificati nelle 3 simulazioni: *Behavioral, Post-Synthesis Functional e Post-Synthesis Timing.*

I warning generati durante la sintesi, in particolare warning per inferring latches e per mancanza di segnali nella sensitivity list, sono stati risolti.

Utilization Report

Figura 2: Utilization Report

Resource	Utilization	Available	Utilization %
LUT	66	134600	0.05
FF	63	269200	0.02
IO	38	285	13.33



Attraverso il *Report Utilization* è stato possibile osservare quanta memoria è stata occupata dal componente, attraverso alcune ottimizzazioni è stato possibile arrivare a raggiungere 66 Look Up Table e 63 Flip Flop, che corrispondono entrambi a <0,1% rispetto alla disponibilità dell'FPGA.

Inoltre, nel Report generato dopo aver sintetizzato il componente è possibile trovare i registri utilizzati nel codice. Viene quindi segnalata la creazione di 10 registri (per un totale di 63 Flip Flop a singolo bit utilizzati).

Timing Report

Inserendo come *constraint* un clock di periodo 100ns (come indicato nella specifica), assegnato al port *i_clock*, è stato possibile nel *Report Timing Summary* ottenere il valore del *Worst Negative Slack*, pari a 94,750, indice del tempo necessario per percorrere il peggior percorso rispetto al tempo totale. Conoscendo anche il ritardo di risposta della RAM è possibile calcolare il periodo minimo applicabile al design creato. Valutando trascurabile il ritardo di riposta della RAM (TRAM) si ottiene:

$$Tmin = Tcurr - WNS + Tram = 100ns - 94.750 + Tram \approx 5{,}25ns \quad (fmax = 1/Tmin)$$

Figura 3: Design Timing Summary

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	94,750 ns	Worst Hold Slack (WHS):	0,144 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:	108	Total Number of Endpoints:	108	Total Number of Endpoints:	64

All user specified timing constraints are met.

4 Simulazioni

Sono stati eseguiti un certo numero di test, alcuni in cui si sono generati casualmente i parametri dei dati memorizzati in memoria, altri mirati a verificare il funzionamento del componente in casi particolari/limite, ovvero in situazioni con più probabilità di generare errori. Tutti questi casi di test hanno terminato la simulazione con esito positivo.

Test benches forniti con la specifica

Vengono di seguito mostrate le waveform di segnali in simulazione *Behavioral* e *Post -Synthesis* dei due test forniti che, nonostante non vadano a testare casi particolari, sono utili per la verifica del corretto funzionamento del componente.

Figura 4: Test Bench 1, waveform dei segnali in simulazione *Post-Synthesis*

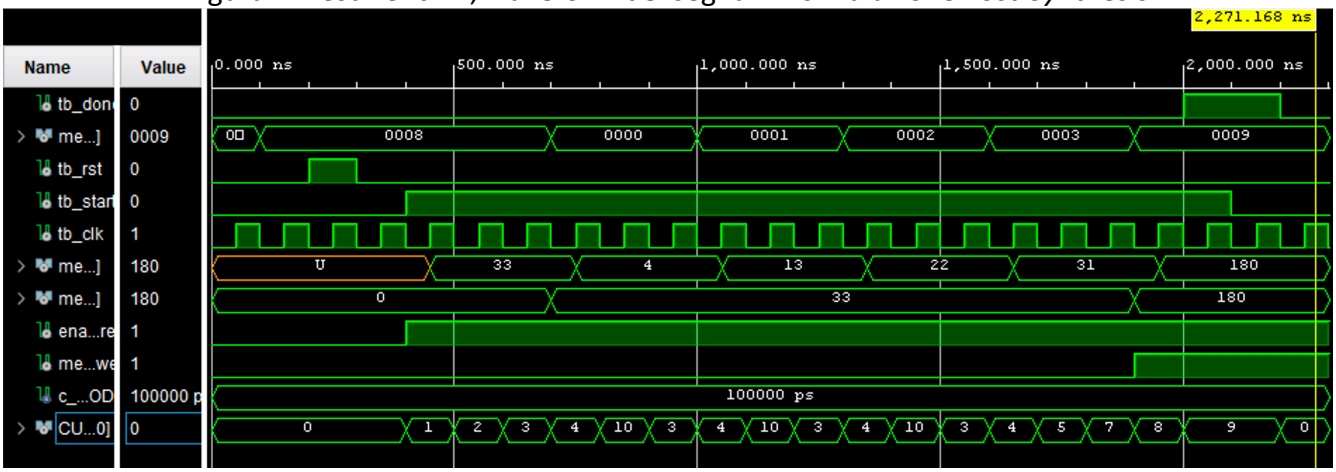
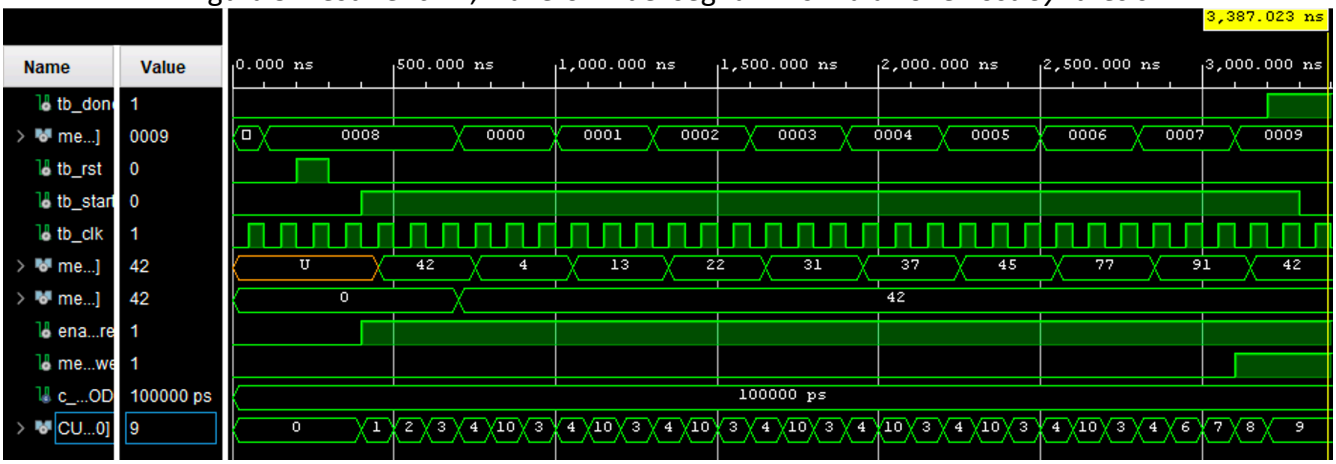


Figura 5: Test Bench 2, waveform dei segnali in simulazione *Post-Synthesis*

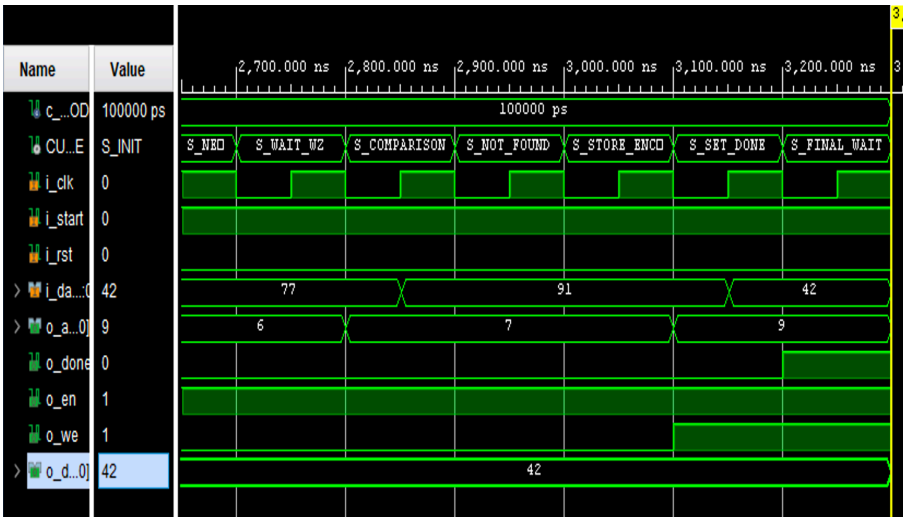


Nelle due waveform dei segnali nella *Post-Synthesis* Simulation si mostra il comportamento del componente una volta sintetizzato nel caso dei due test forniti con la specifica. In entrambe le immagini:

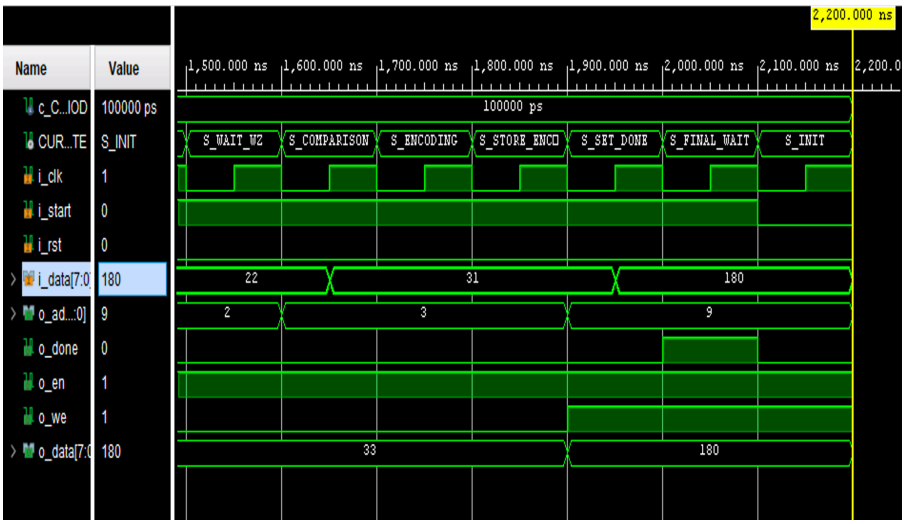
- *we* viene alzato nel momento in cui in il dato codificato è disponibile e può essere salvato;
- *done* viene alzato al termine della codifica e abbassato solo dopo che lo *start* viene abbassato;

Inoltre, nel primo test il valore da codificare appartiene a una wz che viene correttamente identificata, provocando la codifica del valore 33. Dall'immagine si osserva che poiché la wz era contenuta in RAM(3) , si scorrono i primi 4 indirizzi della RAM, passando 4 volte per i tre stati *S_WAIT_WZ* - *S_COMPARIOSN* - *S_NEXT_MEM* (3-4-10) ; Nel secondo test, il valore da codificare non appartiene a nessuna wz e dunque si scorrono gli indirizzi della RAM fino al 7.

Figure 6-7: TB 1-2, waveform dei segnali in Behavioral Simulation



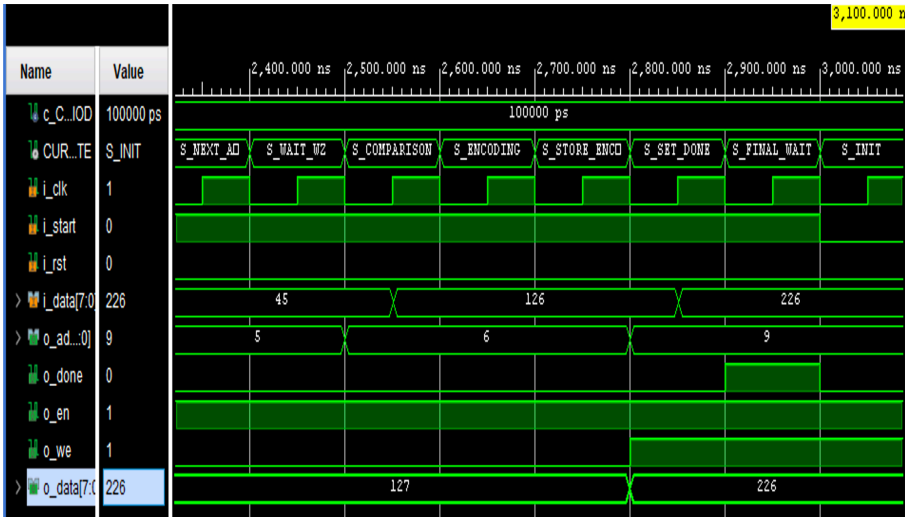
Test Bench 1: (fornito)
Nella figura sono mostrati i segnali nella parte finale della simulazione Behavioral. Si può osservare che la FSM è passata per lo stato *S_NOT_FOUND* e che *o_data* che andrà scritto in RAM(9) è il valore non codificato 42.



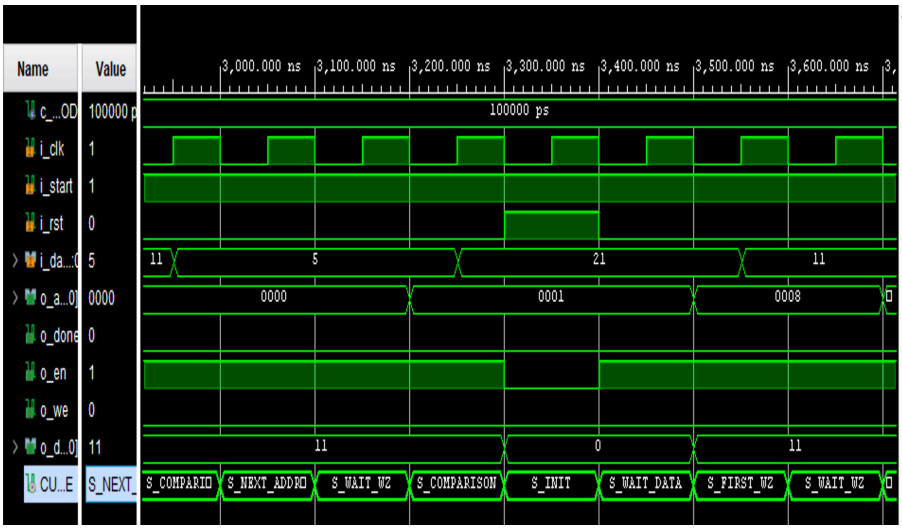
Test Bench 2: (fornito)
Nella figura sono mostrati i segnali nella parte finale della simulazione Behavioral. Si può osservare che la FSM è passata per lo stato *S_ENCODING* e che o_data che andrà scritto in RAM(9) è il valore 180, ovvero 33 codificato correttamente :
1 (*wz_bit*=1) &&
011 (*wz_num*=RAM(3)) &&
0100 (*wz_offset*=33-31=2)
= **1-011-0100** (180)

Altri Test Benches
Qui di seguito si fornisce una breve descrizione e figure che mostrano l'andamento dei segnali, tra cui *CURRENT_STAT*E per poter verificare la corretta successione degli stati della FSM, nei momenti significativi della simulazione di test benches per testare alcuni casi limite.
Sono anche stati testati casi particolari quali: reset asincrono, wz adiacenti e valore non appartenente a una wz ma al limite. Non li vediamo in dettaglio siccome alla fine seguono tutti lo stesso principio di esecuzione.

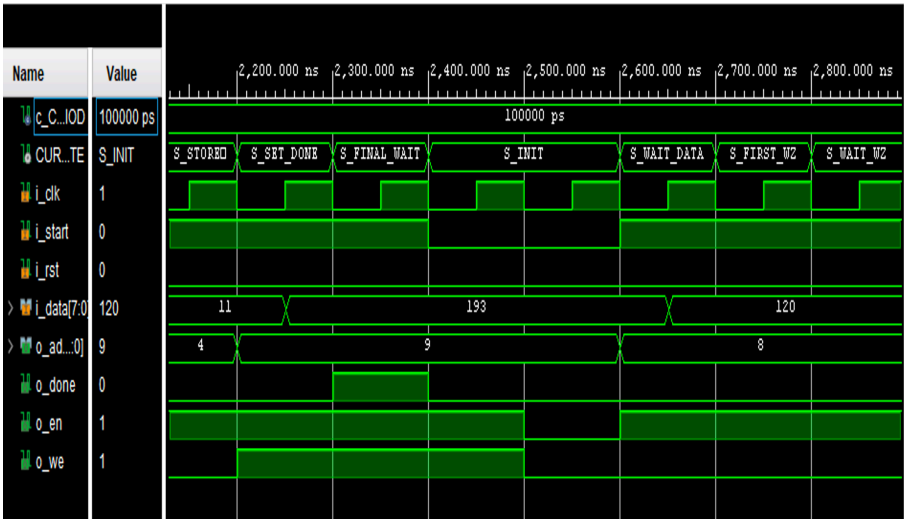
Figure 8-10: Test benches, waveform dei segnali in Behavioral Simulation



Test Bench Address Limit:
Il tb andava a verificare il caso in cui il valore da codificare è pari a 127 ovvero ultimo indirizzo disponibile.Tra le wz era presente l'address 126 e nella figura si può osservare che la FSM passa per lo stato S_ENCODING in cui avviene correttamente la codifica:
1 (wz_bit=1) &&
110 (wz_num=RAM(6)) &&
0010 (wz_offset=127-126=1)
= **1-110-0010** (226)



Test Bench Multi Reset:
Il tb andava a verificare il caso di un secondo reset durante la codifica. Dalla figura si osserva che quando il reset è attivato si ritorna allo stato S_INIT e si riprende il valore da codificare presente in RAM(8), ripartendo dall'inizio a svolgere la codifica correttamente.



Test Bench Multi Start:
Il tb andava a verificare il caso di un secondo start, senza reset, dopo una prima codifica, senza cambiamenti nelle working-zone, ma con il cambiamento dell'indirizzo da codificare. Dalla figura si osserva che quando lo start è attivato si ritorna allo stato S_INIT e si riprende il valore da codificare, ripartendo dall'inizio a svolgere la codifica correttamente.

5 Conclusione

In conclusione, è stato possibile realizzare un componente funzionante nelle simulazioni richieste, che utilizza 0.05% di LUT e 0.02% di FF e che rispetta la specifica data.
Si è cercato inoltre durante tutto il progetto di rendere il codice facilmente comprensibile e suddividere la FSM in stati con funzioni diversificate, prestando particolare attenzione ad evitare sovrapposizioni di segnali. Si è inoltre cercato di diminuire al minimo il numero di variabili utilizzate durante il processo.
La scelta di memorizzare meno informazioni possibili, ovvero di sovrascrivere il valore di ciascuna working-zone con la successiva dopo il confronto senza salvarlo, offre inoltre maggiore scalabilità in quanto, se bisognasse aumentare il numero di working-zones da analizzare, non sarebbero necessarie modifiche al codice se non quella di cambiare le dimensioni dei vettori.
Tra le scelte progettuali in aggiunta, sono stati inseriti due *generic* per permettere estendibilità riguardo al numero di working-zones su cui lavorare.