Prova Finale Reti Logiche 2019-2020

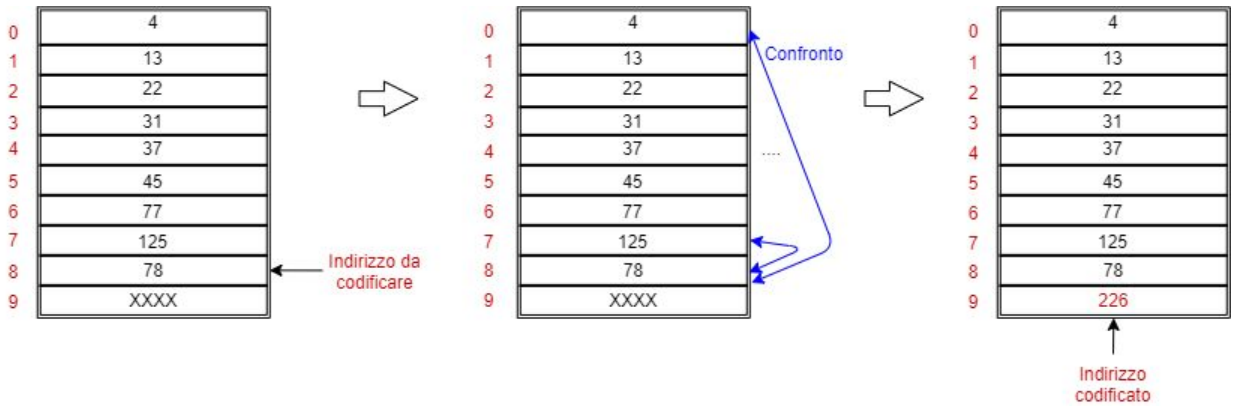
Cosimo Sguanci Codice Persona 10583516 - Matricola 889204
Roberto Spatafora Codice Persona 10557690 - Matricola 888883

Introduzione

La codifica Working Zone (WZ) è una tecnica di codifica del segnale che ha come obiettivo quello di diminuire il consumo di energia dovuto ai pin di I/O, che è una parte molto significativa del consumo di energia complessivo di un chip. Questo si ottiene prendendo in considerazione la località dei riferimenti di memoria: infatti le applicazioni, in ogni istante di tempo, fanno uso di poche “zone di lavoro” all’interno dello spazio di indirizzi. In particolare, dato un indirizzo contenuto in una di queste zone di lavoro (le quali avranno una dimensione, nel caso del nostro progetto fissata a 4 indirizzi), solo l’offset rispetto alla base della WZ (in codifica one-hot) ed un’indicazione sulla WZ di appartenenza sono inviati tramite il bus indirizzi.
La potenza dinamica dissipata da un circuito elettronico è direttamente proporzionale al fattore di attività e alla capacità di carico vista dall'uscita della porta logica. Nel caso di un bus, la capacità di carico è generalmente elevata poiché il bus deve essere collegato a più moduli e anche il fattore di attività è elevato. A causa del valore considerevole della capacità di carico e del fattore di attività, in un sistema tipico il consumo di energia del bus può contribuire fino al 50% del consumo totale di energia. La codifica WZ mira a ridurre questa potenza riducendo la quantità di attività (numero di interruttori) nelle linee del bus; la codifica one-hot dell’offset permette di ridurre il numero di transizioni a causa del ridotto numero di bit da modificare tra un invio di un address ed il successivo.
Nello specifico la codifica WZ opera andando innanzitutto a controllare se l’indirizzo da inviare tramite il bus appartenga ad una delle WZ. In tal caso, l’indirizzo viene codificato come segue:

1 - WZ - WZ_OFFSET

Dove WZ è la Working Zone di appartenenza, e WZ_OFFSET è l'offset rispetto alla base della WZ. Nel caso in cui l'indirizzo non faccia parte di nessuna delle WZ, viene trasmesso inalterato, concatenato ad uno 0 iniziale; come nel caso di corrispondenza individuata (in cui veniva concatenato un 1 iniziale), ci sarà quindi bisogno di un bit addizionale rispetto ai bit di indirizzamento.
Esempio (RAM con indirizzi a 16 bit come da specifica):



L'indirizzo da codificare è quello contenuto nella cella 8 della RAM. Facendo parte di una WZ (quella con indirizzo base 77), l'address dovrà essere codificato come segue:

1 - 110 - 0010

- 110 corrisponde alla WZ di appartenenza (quella il cui indirizzo base è contenuto nella cella 6 della RAM);
- 0010 corrisponde all'offset dell'indirizzo da codificare rispetto alla WZ di appartenenza, in codifica one-hot.

Architettura

Modulo principale

Il modulo principale del componente è stato sviluppato adottando un approccio di tipo behavioral, ed è costituito da un unico process. E' possibile fare un'analogia tra l'implementazione realizzata da tale process ed una macchina a stati finiti, e il diagramma di questa FSM è riportato di seguito. Da ora in avanti verrà quindi fatto riferimento al componente realizzato come se fosse una vera e propria FSM, anche se il process, nonostante abbia esattamente il comportamento descritto, implementa di fatto solo un'analogia con essa.

La progettazione della FSM ha seguito diverse fasi: in un primo momento abbiamo pensato all'equivalente di un contatore che sfruttasse il conteggio dei cicli di clock per passare da uno stato al successivo, come mostrato nella figura 1 di pagina 6. In questo modo abbiamo innanzitutto richiesto alla memoria il contenuto dell'indirizzo 8 (che indicheremo come "target"), contenente l'indirizzo da trasmettere. Leggere prima la cella di memoria numero 8 rispetto a quelle contenenti gli indirizzi base delle WZ ci permette di evitare l'utilizzo di signal aggiuntivi per memorizzare i valori delle WZ stesse. Ottenuto il valore del target, l'FSM implementata contiene uno stato per ogni WZ da confrontare. Una volta terminati i confronti con tutte le WZ lette dalla memoria, se viene individuata una corrispondenza, il signal "found" è posto a 1, interrompiamo il conteggio e ci portiamo in uno stato "trovato", nel quale effettuiamo la codifica dell'indirizzo e la FSM rimane in attesa per una nuova eventuale codifica. Al contrario, se dopo aver letto tutte le WZ, non c'è stato alcun matching, passiamo in uno stato "non_trovato", in cui scriviamo nella cella 9 della memoria il target, e come per il caso opposto rimaniamo in attesa di ulteriori codifiche da effettuare. In entrambi gli stati "finali" provvediamo a reinizializzare tutti i signal di interesse per essere pronti ad effettuare nuove codifiche.

In una fase successiva della progettazione, sfruttando sempre il concetto di conteggio dei cicli di clock, e continuando con l'analogia tra il componente ed una FSM, abbiamo deciso di collassare gli stati in cui precedentemente erano lette le basi delle WZ in un unico stato utilizzando un autoanello per la realizzazione di tutte le letture dalla memoria, uscendo dallo stato solo in caso di corrispondenza individuata o esaurimento delle WZ. In questo modo siamo riusciti ad utilizzare meno signal, andando così a rendere il codice meno

ridondante, a parità di risultato ottenuto. Inoltre abbiamo così potuto operare alcune ottimizzazioni, evitando di continuare ad effettuare confronti quando viene individuata la WZ di appartenenza del target, risparmiando cicli di clock.
Nella terza ed ultima fase di progettazione del componente, siamo andati ad ottimizzare il tipo dei signal utilizzati, convertendo tutti quello aventi tipo “integer” in vettori di bit (std_logic_vector). Abbiamo infatti notato che:

- Per rappresentare num_clk sarebbero bastati 4 bit (il massimo valore che può assumere è 12);
- target_group, utilizzato per indicare il gruppo di appartenenza del target, può assumere al massimo il valore 7, ed è quindi più conveniente fare uso di un std_logic_vector di dimensione 3;
- group_data, che è il signal responsabile del salvataggio della base della WZ di appartenenza (caso “found” = ‘1’), può essere rappresentato da un vettore di 8 bit (così come il target).

Mediante queste ottimizzazioni, siamo riusciti a diminuire notevolmente l’area occupata dal componente progettato, passando da 156 LUT e 99 FF a 54 LUT e 47 FF.

L’FSM che riproduce il comportamento del componente a seguito di tali modifiche è mostrata in figura 2.

Descrizione degli stati

Si ricorda che gli stati a cui si fa riferimento in questo paragrafo sono un’analogia con le fasi che vengono eseguite dal process implementato.

Il primo stato, lo stato “***richiesta_lettura_target***”, è lo stato da cui parte l’esecuzione di ciascuna codifica e da cui si riparte in presenza del segnale di reset. In questo stato viene richiesto alla memoria di fornire il contenuto all’indirizzo otto, contenente il valore che si vuole andare a codificare secondo la specifica. Viene inoltre incrementato il segnale “num_clk” attraverso cui viene gestito il conteggio dei cicli di clock della FSM.

Il secondo stato, denominato “***attesa_lettura_target***”, consente di ritardare l’esecuzione di un ciclo di clock al fine di utilizzare negli stati successivi il corretto “target”. Come nello stato precedente viene incrementato il valore contenuto nel signal “num_clk”. Questo stato consente di avere un valore corretto per il signal “target”, che sarà disponibile soltanto dopo due cicli di clock.

Nel terzo stato, ***"lettura_target_e_richiesta_wz0"***, una volta ottenuto il corretto valore di "target", viene richiesto il contenuto della memoria all'indirizzo zero così da poter successivamente iniziare una fase di controllo che coinvolgerà il target e il contenuto della cella di memoria richiesto. Come succede negli stati precedenti viene incrementato il valore del conteggio dei cicli di clock, facendo uso del signal "num_clk".

Il quarto stato, denominato "***attesa_lettura_wz0_e_richiesta_wz1***", come il secondo stato, consente di ritardare l'esecuzione di un ciclo di clock al fine di effettuare correttamente il confronto tra "target" e "i_data". All'interno di questo stato o_address viene portato al valore 1, così da uniformare il ritardo necessario affinchè negli stati successivi arrivi in i_data il valore richiesto dalla memoria, e ottenere correttamente il numero identificativo della WZ di appartenenza, nel caso in cui questa venga individuata.

Il quinto stato, "***controllo_target_wz***", mostra, di fatto, la miglioria apportata dalla soluzione proposta rispetto a quella precedentemente progettata. Vengono infatti condensati in questo unico stato otto stati della prima soluzione. Si passerà ad uno stato successivo non appena verrà verificata la non appartenenza a nessuna delle WZ, oppure al primo confronto che risulta positivo, il che significa trovare la WZ cui il target appartiene. Nella prima soluzione implementata non era possibile terminare il confronto tra il target e le WZ prima di aver letto tutte le WZ stesse, ed era quindi necessario eseguire tutti i controlli e solo dopo decidere, in funzione del valore del signal "found", in quale stato sarebbe finita la FSM; la nuova soluzione permette invece di terminare i confronti non appena individuata l'appartenenza del target ad una WZ, permettendo al componente di passare immediatamente alla fase di codifica dell'address. Al primo confronto che risulta positivo, il signal "found" viene portato al valore "1", attraverso cui si arriverà in uno stato "trovato", successivamente illustrato. Qualora nessuno dei confronti dovesse dare esito positivo si arriverà in uno stato caratterizzato dal valore "0" del signal "found". Il confronto qui implementato consiste nella verifica dell'appartenenza del valore "target" al range di valori compresi tra il valore dell'indirizzo base della WZ in esame e tre posizioni successive a quest'ultimo; tale range rappresenta la dimensione delle WZ come da specifica.

Lo stato "***trovato***", sopra citato, è uno stato al quale si può arrivare dallo stato "confronto_target_wz" in seguito al primo confronto risultato positivo che determina la WZ di appartenenza del target. In tale stato si procederà pertanto con la codifica del valore

contenuto nel target, per andare infine a scrivere nella cella di memoria 9 l'address correttamente codificato.

Infine, lo stato "***non_trovato***" è uno stato al quale si arriva solo quando sono stati effettuati i tutti i controlli con le WZ presenti in memoria senza trovarne una che includa il target. Come per lo stato "trovato", anche in questo stato, terminale per una singola esecuzione, si va a scrivere nella cella 9 della memoria, in particolare viene scritto lo stesso valore contenuto nel signal "target" (che, come detto, corrisponde all'address da codificare).

Figura 1

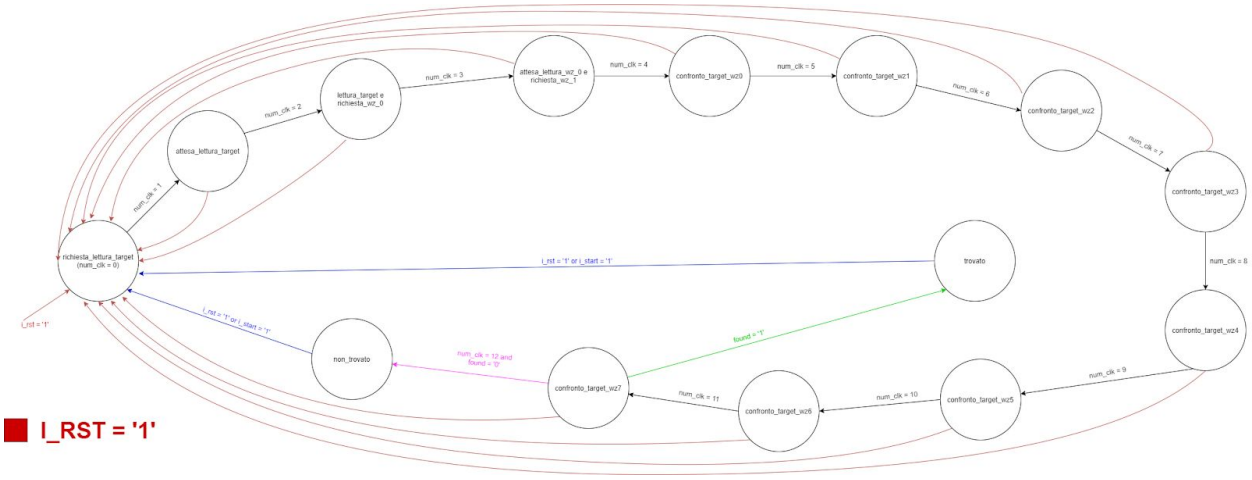
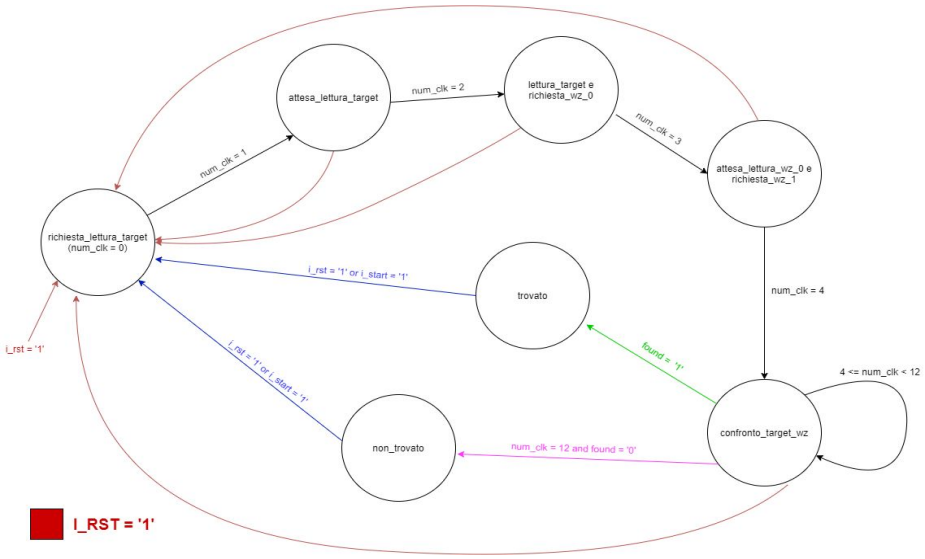


Figura 2



Risultati Sperimentali

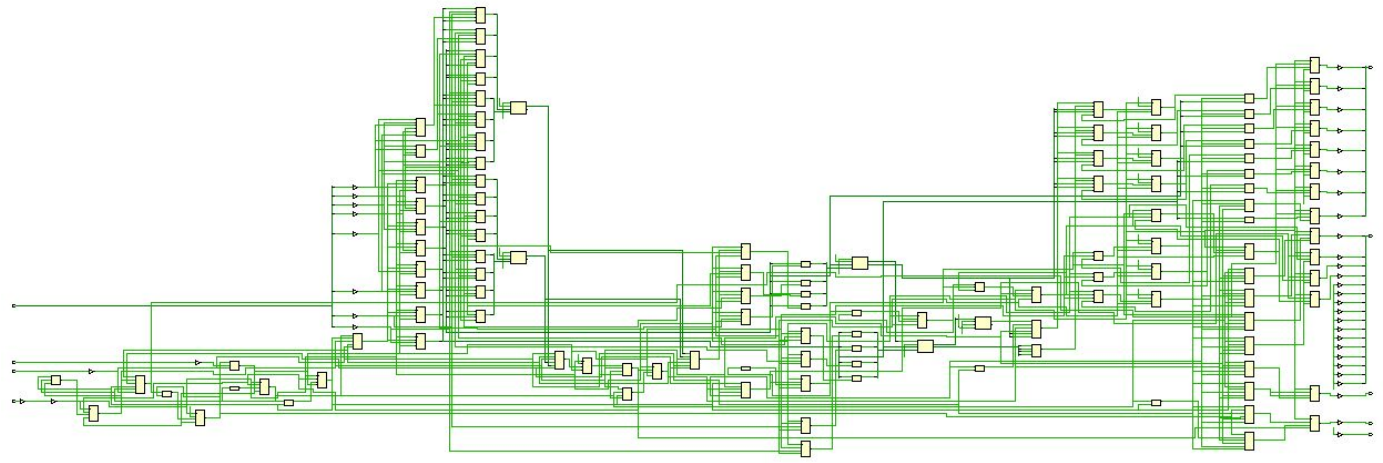
Sintesi

```
Report Cell Usage:
+------+-------+------+
|      |Cell   |Count |
+------+-------+------+
|1     |BUFG   |     1|
|2     |CARRY4 |     5|
|3     |LUT1   |     2|
|4     |LUT2   |    14|
|5     |LUT3   |    12|
|6     |LUT4   |    17|
|7     |LUT5   |    12|
|8     |LUT6   |    13|
|9     |FDRE   |    45|
|10    |FDSE   |     2|
|11    |IBUF   |    11|
|12    |OBUF   |    27|
+------+-------+------+
```

Il modulo consta di 161 celle, di cui, nella foto mostrata a sinistra, si notano i diversi tipi di componenti e le relative cardinalità.

```
Report Instance Areas:
+------+---------+-------+------+
|      |Instance |Module |Cells |
+------+---------+-------+------+
|1     |top      |       |   161|
+------+---------+-------+------+
```

161 Cells 38 I/O Ports 194 Nets



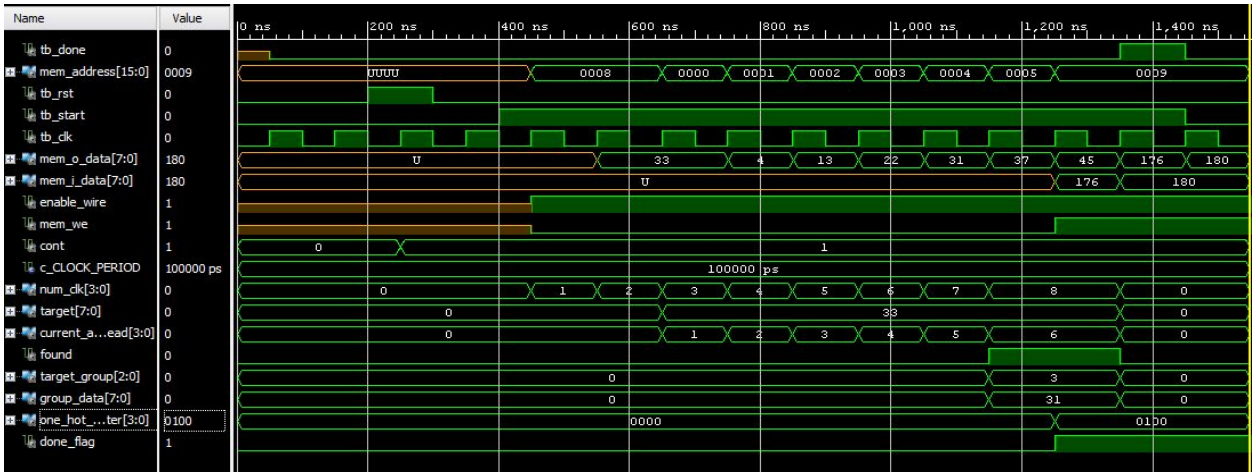
Simulazione

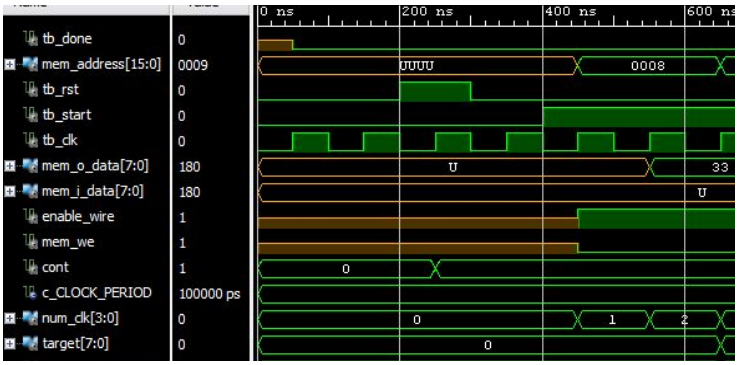
Durante la fase di simulazione, grazie alle waveform, è possibile notare come sia necessario gestire i ritardi generati tra la richiesta di un determinato dato e l'arrivo di questo nel segnale specificato: infatti per ogni dato richiesto o segnale portato ad un determinato valore, sarà necessario aspettare il ciclo clock successivo per vedere gli effetti dei vari assegnamenti.

Esempio 1

Un esempio di funzionamento del progetto è dato dal waveform relativo al seguente contenuto di memoria. Tale esempio riporta un caso in cui viene effettuata la codifica del numero letto, seguendo quanto descritto nell'introduzione.

```
-- come da esempio su specifica
signal RAM: ram_type := (0 => std_logic_vector(to_unsigned( 4 , 8)),
                         1 => std_logic_vector(to_unsigned( 13 , 8)),
                         2 => std_logic_vector(to_unsigned( 22 , 8)),
                         3 => std_logic_vector(to_unsigned( 31 , 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( 125 , 8)),
                         8 => std_logic_vector(to_unsigned( 33 , 8)),
              others => (others =>'0'));
```





La porzione d'immagine a sinistra, relativa all'esempio riportato sopra permette di notare la gestione della lettura, nell'apposito segnale detto "target", del valore che si vuole codificare. Risulta infatti necessario aspettare un ciclo di clock quando la FSM si trova nel secondo stato ("num_clk" = 1) in quanto in "mem_o_data" non è ancora disponibile il dato contenuto nell'indirizzo otto della memoria, pertanto non aspettando si commetterebbe un errore, nel resto dell'esecuzione, dovuto alla presenza di un valore errato contenuto dal segnale "target".

Un'altra peculiarità che vale la pena osservare è data dal fatto che, come nel secondo stato, anche nel quarto viene ritardata l'esecuzione così da evitare un confronto errato dovuto alla presenza del valore da codificare (ovvero il target) in "i_data", invece del contenuto della WZ0 (necessario per iniziare ad effettuare i confronti). Senza questo stato di attesa, il primo confronto darebbe pertanto sempre esito positivo, compromettendo l'esecuzione.

Test

Per agevolare la leggibilità è stato ritenuto opportuno suddividere i test effettuati in 3 categorie: test sulle variazioni dei segnali in input, test sui valori limite, test su particolari istanze di esecuzioni.

Test sui segnali in input

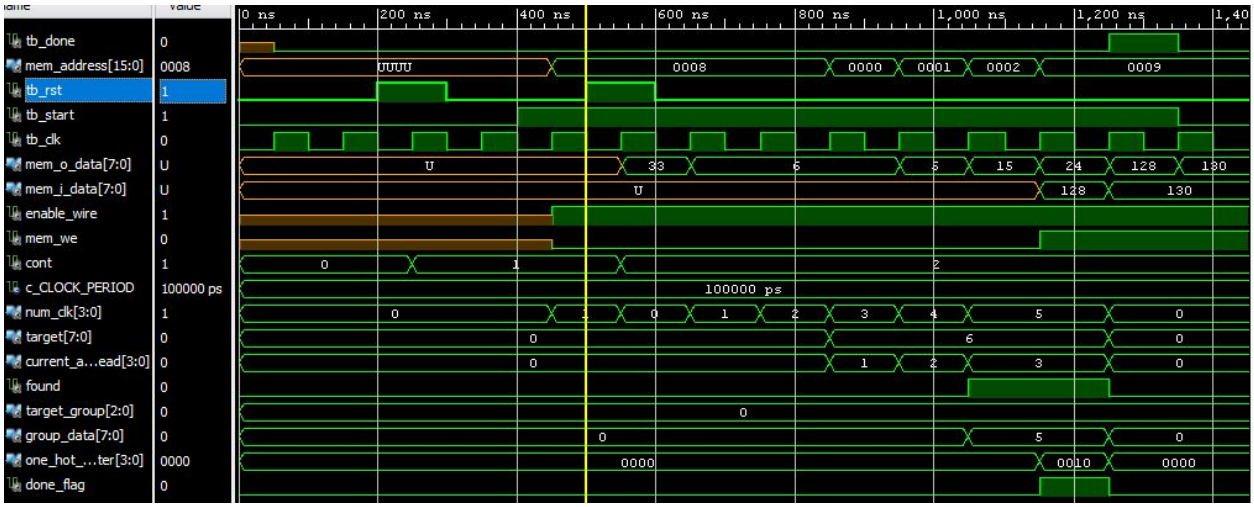
Segnale di RST a 1 con codifica in corso

```
test : process is
begin
    wait for 100 ns;
    wait for c_CLOCK_PERIOD;
    tb_rst <= '1';
    wait for c_CLOCK_PERIOD;
    tb_rst <= '0';
    wait for c_CLOCK_PERIOD;
    tb_start <= '1';
    wait for c_CLOCK_PERIOD;
    tb_rst <= '1';
    wait for c_CLOCK_PERIOD;
    tb_rst <= '0';
    wait until tb_done = '1';
    wait for c_CLOCK_PERIOD;
    tb_start <= '0';
    wait until tb_done = '0';
    wait for 100 ns;
```

Un test particolarmente significativo per verificare il corretto funzionamento del componente a fronte di segnali di reset consiste nel porre i_rst a 1 mentre è in corso un'altra codifica (quindi anche i_start è a 1). In questo modo, modificando anche i valori contenuti in RAM, viene testata la capacità di reinizializzare i signal utilizzati al fine di ripartire con una successiva codifica tralasciando in toto quella precedente.

```
MEM : process(tb_clk)
begin
    if tb_clk'event and tb_clk = '1' then
        if(tb_rst = '1') then
            if(cont = 1) then
                RAM(0) <= std_logic_vector(to_unsigned( 5 , 8));
                RAM(1) <= std_logic_vector(to_unsigned( 15 , 8));
                RAM(2) <= std_logic_vector(to_unsigned( 24 , 8));
                RAM(3) <= std_logic_vector(to_unsigned( 54 , 8));
                RAM(4) <= std_logic_vector(to_unsigned( 60 , 8));
                RAM(5) <= std_logic_vector(to_unsigned( 81, 8));
                RAM(6) <= std_logic_vector(to_unsigned( 30 , 8));
                RAM(7) <= std_logic_vector(to_unsigned( 45 , 8));
                RAM(8) <= std_logic_vector(to_unsigned( 6 , 8));
                cont <= cont + 1;
            else
               cont <= cont + 1;
            end if;
        end if;
```

A tal fine è stato modificato il test bench fornito dal docente aggiungendo un nuovo signal di tipo integer inizializzato a 0; all'interno del processo MEM tale signal viene incrementato di 1 e, ad ogni reset, viene cambiato il contenuto della porzione di memoria per noi significativa.

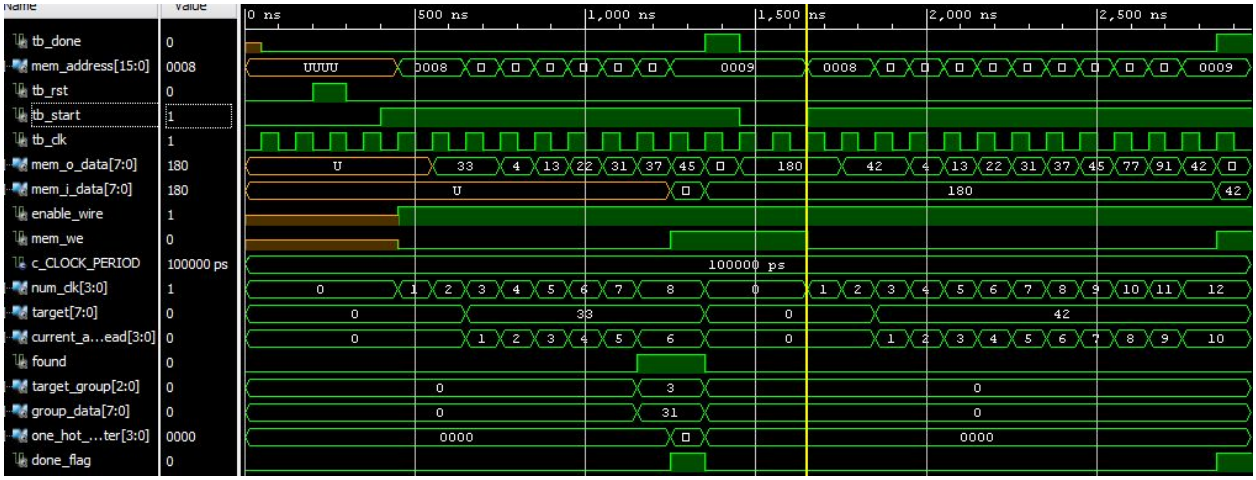


È bene notare dal waveform sopra riportato che, sebbene fosse cominciata la prima codifica richiesta, una volta riportato ad 1 il segnale i_rst (t = 500 ns), i signal sono stati correttamente reinizializzati e il componente ha proceduto con la nuova codifica, facendo uso del contenuto aggiornato della RAM.

Una variazione del test descritto ci ha permesso di verificare il risultato voluto in corrispondenza del valore del segnale di reset a 1 per N cicli di clock. In tal caso, aggiungendo altre N configurazioni della RAM nel processo MEM, seguendo quanto atteso, il risultato della codifica coinvolge solo l'N-esima configurazione della RAM.

Molteplici codifiche, stesse WZ

Terminata una codifica, nel caso in cui i_start torni ad 1 (senza che i_rst venga posto ad 1), è necessario iniziare una nuova codifica. In tal caso, l'unico valore che potrà cambiare è quello relativo all'indirizzo da codificare. Per testare questo comportamento è stato nuovamente modificato il processo MEM del testbench: ad ogni scrittura viene controllato l'indirizzo su cui questa sta avvenendo, se si tratta dell'indirizzo 9, vorrà dire che stiamo scrivendo l'indirizzo codificato, quindi si può andare a cambiare il contenuto della cella 8 della RAM, ovvero l'address da codificare. Il comportamento atteso è che il componente vada ad effettuare la codifica del nuovo indirizzo, utilizzando le stesse basi precedenti delle WZ.



Dal waveform si osserva che, terminata la prima codifica, si ha una situazione di attesa (i_start = '0'); nel momento in cui il segnale di start torna ad 1, l'address da codificare è stato cambiato, il componente leggerà nuovamente tutti i dati necessari dalla memoria e l'encoding viene effettuato correttamente anche la seconda volta.

Modifica del periodo di clock

Per testare la robustezza del componente a diverse velocità di esecuzione abbiamo eseguito dei test modificando il periodo di clock. Il risultato ottenuto è stato quello atteso, i tempi di esecuzione variano in funzione del periodo inserito mantenendo tuttavia costante il numero di cicli di clock per l'esecuzione del caso pessimo. Tuttavia è necessario notare che l'attesa dopo una richiesta di lettura o scrittura da memoria (nel tb di esempio è

implementato utilizzando ‘after 1ns’ nel processo MEM) deve essere minore del periodo di clock.

Test sui valori limite

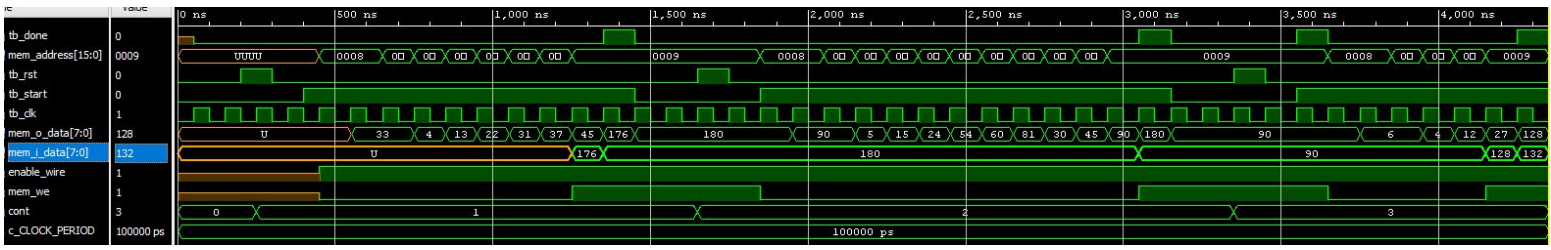
L’indirizzo da codificare è un unsigned a 7 bit, quindi i valori limite da considerare sono 0 e 127. È possibile considerare zero come gli altri possibili valori da codificare. Una maggiore attenzione è da prestare ai valori finali del range ammissibile, in quanto, qualora fossero presenti indirizzi base di WZ maggiori di 124, le WZ corrispondenti avrebbero una dimensione minore di 4, essendo 127 il valore massimo unsigned esprimibile con 7 bit.

In entrambi i casi descritti il test è andato a buon fine, visto che, potendo assumere che non ci siano mai unsigned di 8 bit come target, i valori limite sono gestiti esattamente come i valori “standard” all’interno del range ammissibile.

Test su particolari istanze di esecuzioni

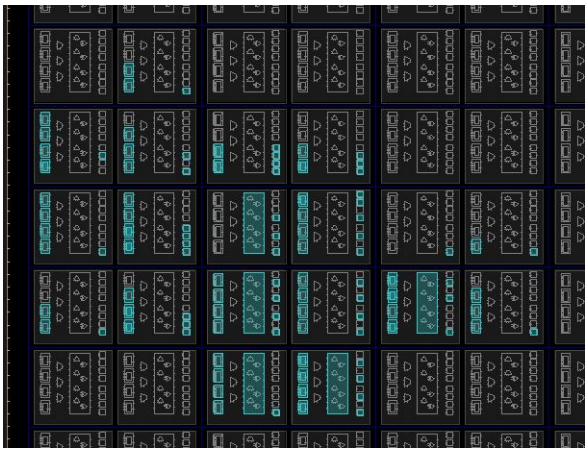
Molteplici codifiche in presenza di RST a 1

Una verifica rilevante da effettuare consiste nell’osservare il comportamento del componente realizzato in presenza di più codifiche richieste con il segnale i_rst che può essere portato ad 1 (a differenza del caso di test precedentemente analizzato). Infatti in tal caso, oltre all’address da codificare, potranno cambiare anche gli indirizzi base delle WZ.



Il caso di più istanze di esecuzioni consecutive non causa problemi al componente, infatti il process che implementa il modulo principale, ogni volta che una codifica è terminata (oppure se i_rst =’1’), pone ‘num_clk’ e tutti gli altri signal al valore iniziale prima di iniziare una nuova fase di encoding.

Conclusioni



Il componente realizzato è risultato idoneo in fase di sintesi, superando tutti i testbench sia in simulazione behavioral sia post-sintesi functional e post-sintesi timing. Il componente è inoltre risultato correttamente sintetizzabile e funzionante anche anche in simulazione functional post-implementazione.

In termini di prestazioni temporali, nel caso pessimo, ovvero quello in cui l'indirizzo da codificare non fa parte di nessuna WZ, il modulo termina le proprie operazioni in 1,3μs con periodo di clock 100ns (13 cicli di clock). Nel caso ottimo invece, in cui l'indirizzo da codificare appartiene alla prima WZ, il tempo impiegato è pari a 700ns.