Prova Finale:
Relazione di Progetto

Anno Accademico 2019/2020

Simone Sarti – 887069

Sommario

1 Introduzione

Scopo di questa prova finale è l'implementazione di un modulo hardware capace di realizzare una versione semplificata, poiché in grado di gestire indirizzi di 7 bit, del metodo di codifica degli indirizzi denominato *Working Zone*.

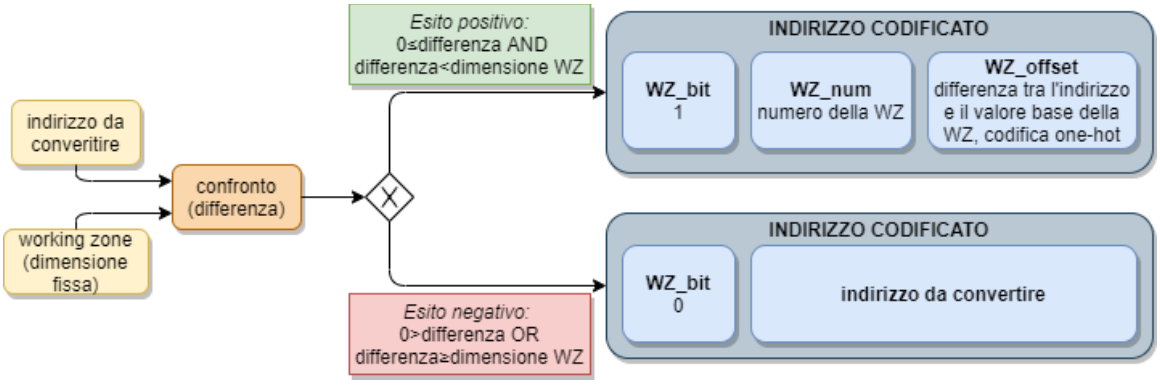
Il modulo deve essere in grado di interfacciarsi con una memoria RAM (già fornita), la quale contiene l'indirizzo da codificare, gli indirizzi base delle WZ, ed in cui, a computazione finita, deve essere inserito il risultato della codifica.

Il modulo è stato realizzato utilizzando il linguaggio di descrizione dell'hardware VHDL

1.1 Schema di funzionamento della codifica WZ con esempio

L'indirizzo ottenuto dalla codifica avrà una lunghezza di 8 bit.

L'indirizzo da codificare dovrà essere confrontato con 8 indirizzi base di WZ ($8=2^3$, 3 bit per WZ_num). Ogni WZ avrà una dimensione fissa di 4 indirizzi (one-hot su 4 bit per WZ_off). Il bit rimanente sarà quello destinato a WZ_bit.



1. Voglio codificare l'indirizzo 80. Suppongo ora che tra le varie WZ, la quinta abbia 77 come indirizzo di base. Il confronto tra i due valori dà risultato 3. Poiché il confronto ha esito positivo, l'indirizzo codificato sarà formato come segue:
    a. WZ_bit =1
    b. WZ_num = 101 (5)
    c. WZ_off = 1000 (3 one-hot)

2. Voglio codificare l'indirizzo 80. Suppongo ora che tra le varie WZ, la quinta sia quella con indirizzo più simile a quello da codificare, avendo 76 come indirizzo base. Il confronto tra i due valori dà risultato 4. Poiché il confronto ha esito negativo, l'indirizzo codificato sarà formato come segue:
    a. WZ_bit =0
    b. Indirizzo da codificare (7 bit) = 1000110 (80)

1.2 Software utilizzati

Il codice che descrive il modulo HW, così come quello dei Test Bench, è stato scritto e modificato usando un editor di testo.

Per visualizzare il modulo e i suoi elementi costitutivi, simularlo e sintetizzarlo, è stato utilizzato il software Vivado 2019.2 WebPACK di Xilinx.

2 Architettura

2.1 Punti focali nella realizzazione

2.1.1 Ottimizzazione temporale

Il modulo è stato progettato con lo scopo di ottenere un prodotto finale in grado di svolgere una computazione completa nel minor tempo possibile.

L'ottimizzazione temporale più importante riguarda la capacità del modulo di smettere di richiedere nuovi valori di base di WZ alla RAM non appena trovata, se presente, la WZ in cui si colloca l'indirizzo da codificare. In questo caso, il modulo passa subito alla scrittura del risultato in memoria e poi termina.

Una seconda ottimizzazione riguarda la capacità del modulo di effettuare richieste e confronti tra indirizzi parallelamente. Questo permette di mitigare il più possibile il ritardo di 1 ciclo di clock e 1 ns che intercorre tra la richiesta del valore alla RAM e il suo effettivo arrivo al modulo.

2.1.2 Leggibilità del codice e facilitazione nella ricerca di errori

Il codice è stato scritto in maniera tale che il funzionamento del modulo risulti il più chiaro possibile. Da ciò deriva la decisione di tenere separati i processi che ne compongono l'architecture, sebbene li si sarebbe potuti raggruppare tutti in uno solo. Questo approccio ha facilitato notevolmente l'individuazione e la correzione degli errori evidenziati dai test svolti.

Inoltre, il modulo è stato descritto in modo che in seguito ad un reset tutti i segnali vengano riportati ad un valore di default. Lo stesso succede a computazione terminata (ad eccezione dei segnali legati ai registri, che potranno essere sovrascritti da una eventuale computazione successiva senza dare problemi). Benché ciò vada a "impattare" sul numero di componenti di memoria necessari, viene resa estremamente più semplice l'individuazione di errori che si possono generare in casi di start multipli e reset asincroni, poiché permette di identificare facilmente gli elementi che non si sono comportati nella maniera appropriata.

Nota: il valore di default è 0, eccetto per `o_address`, che ha valore 000f. Le ragioni di questa scelta sono due:

- 0000 è un indirizzo valido della memoria, non voglio puntarlo se non quando devo accedervi.
- Sempre volendo evitare i primi 9 indirizzi, usare 000f è una scelta stilistica che permette di effettuare un controllo visivo più rapido del fatto che il modulo non rimanga incastrato all'indirizzo 0000 (ultimo letto) alla fine della computazione, rispetto all'uso del più simile 000a (soprattutto a simulazione in corso). Ovviamente presentare in uscita valore 15 e non 10 significa "aver bisogno" di 5 celle di RAM in più, la soluzione sarebbe stata diversa se il modulo venisse realizzato sul serio.

2.2 Entity del modulo

La Entity del modulo è costituita dai seguenti segnali:

- `i_clk`: segnale d'ingresso di tipo std_logic, è il segnale di clock generato dal Test Bench.
- `i_start`: segnale d'ingresso di tipo std_logic, è il segnale di start generato dal Test Bench.
- `i_rst`: segnale d'ingresso di tipo std_logic, è il segnale di reset del modulo.
- `i_data`: segnale d'ingresso di tipo std_logic_vector a 8 bit, è il segnale che contiene il valore che arriva dalla memoria in seguito ad una richiesta di lettura.
- `o_address`: segnale d'uscita di tipo std_logic_vector a 16 bit, è il segnale che contiene l'indirizzo di memoria di cui il modulo richiede il valore.
- `o_done`: segnale d'uscita di tipo std_logic, è il segnale che comunica la fine dell'elaborazione.

- o_en: segnale d'uscita di tipo std_logic, è il segnale di enable da dover mandare alla memoria per poter comunicare, sia in lettura che in scrittura.
- o_we: segnale d'uscita di tipo std_logic, è il segnale di write enable da dover alzare per poter scrivere in memoria. Per leggere da memoria esso deve essere 0.
- o_data: segnale d'uscita di tipo std_logic_vector a 8 bit, è il segnale che contiene il valore codificato da scrivere in memoria.

2.3 Architecture del modulo

Il modulo è stato realizzato adottando uno stile di modellazione principalmente behavioural per specificarne l'architecture. In piccola parte è stato utilizzato anche uno stile di tipo structural, per poter sfruttare due registri come componenti. Non sono state utilizzate macchine a stati.

2.3.1 Processi utilizzati

L'architecture del modulo è costituita da tre processi, qui descritti nell'ordine in cui appaiono nel codice.

2.3.1.1 Processo 1: INTERAZIONE CON LA MEMORIA

È il processo più complesso dei tre, si occupa di gestire l'interazione tra il modulo realizzato e la memoria RAM. Nella sua lista di sensibilità compaiono il segnale di reset e il segnale di clock.

Se il processo viene attivato dal segnale di reset (anche in maniera asincrona), i segnali di output vengono impostati al loro valore di default. Lo stesso succede ai segnali di supporto fine e ciclo_completo. Il contatore (counter) viene re-inizializzato.

In assenza di reset, questo processo opera sul fronte di salita del clock, e ad ogni ciclo di clock, basandosi sul valore del contatore (meno una costante), assegna ad o_address l'indirizzo di memoria RAM da cui si vuole prelevare il valore dell'indirizzo di base della WZ. Visto che tali indirizzi possono essere distribuiti senza un ordine preciso all'interno delle prime 8 celle di RAM, e che in ogni caso il primo passo deve essere la lettura dell'indirizzo da codificare, il quale risiede all'indirizzo di memoria 0008, il modulo assegnerà valori al segnale o_address in maniera decrescente, fino al raggiungimento di 0000 oppure all'interruzione delle richieste alla memoria dovuta ai processi seguenti.

In particolare, il secondo caso si verifica quando viene confermata l'appartenenza dell'indirizzo da codificare ad una WZ, ciò è vero se il segnale WZ_bit assume valore 1. In tal caso, il modulo provvede a porre o_address a 0009, alzare il segnale di WE e assegnare ad o_data il segnale valore_convertito. Fatto ciò, al ciclo di clock successivo, mentre la memoria salverà il valore, il processo alzerà il segnale o_done per indicare la fine della computazione, e riporterà i segnali di sua competenza ai loro valori di default.

Ciò che cambia nel caso in cui l'indirizzo non appartenga a nessuna WZ è che, una volta inviata la richiesta per il valore all'indirizzo 0000 e averlo testato, in caso di esito negativo, il processo si comporterà esattamente come descritto al punto precedente, ma ad o_data verrà assegnato indirizzo_convertire_out anziché valore_convertito.

2.3.1.2 Processo 2: INSTRADAMENTO DEGLI INDIRIZZI

La sua lista di sensibilità contiene i segnali di reset e di clock.

In caso venga attivato dal segnale di reset, che può essere anche asincrono, il processo si occupa di reimpostare al loro valore di default i segnali di ingresso ai due componenti di tipo address_register. Il valore interno ai registri si resetta in autonomia grazie alla loro descrizione comportamentale.

Il processo opera sul fronte di discesa del clock e si occupa, sulla base del valore del segnale `counter`, di inoltrare l'input ricevuto dalla memoria (e letto sul fronte di salita precedente dal processo 1) verso l'`address_register` corretto. In particolare, il primo indirizzo ricevuto dovrà essere instradato verso il registro `indirizzo_convertire`, mentre i successivi verso `base_WZ`.

Sul primo fronte di salita, il registri salveranno i valori ricevuti.

NB: Sebbene i valori ricevuti dalla memoria siano lunghi 8 bit, la conversione riguarda solo i primi 7. Tuttavia, poiché da specifica viene indicato che il MSB sarà sempre 0, essi possono (e vengono, per scelta di progetto) letti e instradati tutti e 8, in quanto lo 0 iniziale non influirà in nessun modo sulla conversione, la quale dipende solo da una differenza.

2.3.1.3 Processo 3: CONVERSIONE

Come i primi due, anche questo processo ha nella sua lista di sensibilità i segnali di reset e di clock.

Alla ricezione del segnale di reset, il processo provvede a riportare al valore di default i segnali `WZ_bit` e `valore_convertito`.

Alternativamente, il processo opera sul fronte di discesa del clock. Esso prende i valori salvati nei due `address_register` sul fronte di salita precedente, e verifica, svolgendo una differenza, se l'indirizzo da codificare si trova in tale WZ. In caso il confronto dia esito positivo, il segnale `WZ_bit` viene portato a 1 ed al segnale `valore_convertito` viene assegnato il valore codificato secondo il protocollo WZ.

Il valore di WZ_num non sarà altro che quello del segnale `counter`, pari al numero della WZ che sto testando. Per ottenere WZ_off viene fatto uno shift a sinistra del valore 0001 pari al risultato della differenza.

Una volta alzato `WZ_bit`, il processo 1 si occuperà di scrivere il risultato in memoria e di terminare l'esecuzione.

L'ultimo compito di questo processo è quello di riportare ai valori di default i segnali `WZ_bit` e `valore_convertito`, ciò viene fatto sul fronte di discesa successivo all'innalzamento di `o_done`.

NB: Nei due cicli di clock che intercorrono tra la scoperta della WZ e la terminazione, questo processo verrà eseguito altrettante volte (sempre che ci siano almeno altri due confronti da fare). Il risultato ottenuto non potrà comunque cambiare, sia perché il risultato positivo viene inviato alla memoria prima di questi nuovi confronti, sia perché nel metodo di codifica WZ, le WZ non possono essere sovrapposte, pertanto non sarà mai possibile ottenere un nuovo assegnamento a `valore_convertito`. Queste esecuzioni in più sono dovute allo sfasamento tra richiesta di valori alla RAM e l'effettivo svolgimento dei confronti.

2.3.2 Componenti utilizzati

È stato descritto a parte il funzionamento di un registro parallelo-parallelo a 8 bit, con reset asincrono e aggiornamento sul fronte di salita del clock. Il nome della sua entity è `address_register`, poiché i due componenti di questo tipo istanziati nel progetto vengono utilizzati rispettivamente per salvare l'indirizzo da codificare e gli indirizzi delle basi delle WZ. I due componenti, denominati rispettivamente `indirizzo_convertire` e `base_WZ`, sono riconoscibili nello schematic del progetto per il loro colore azzurro.

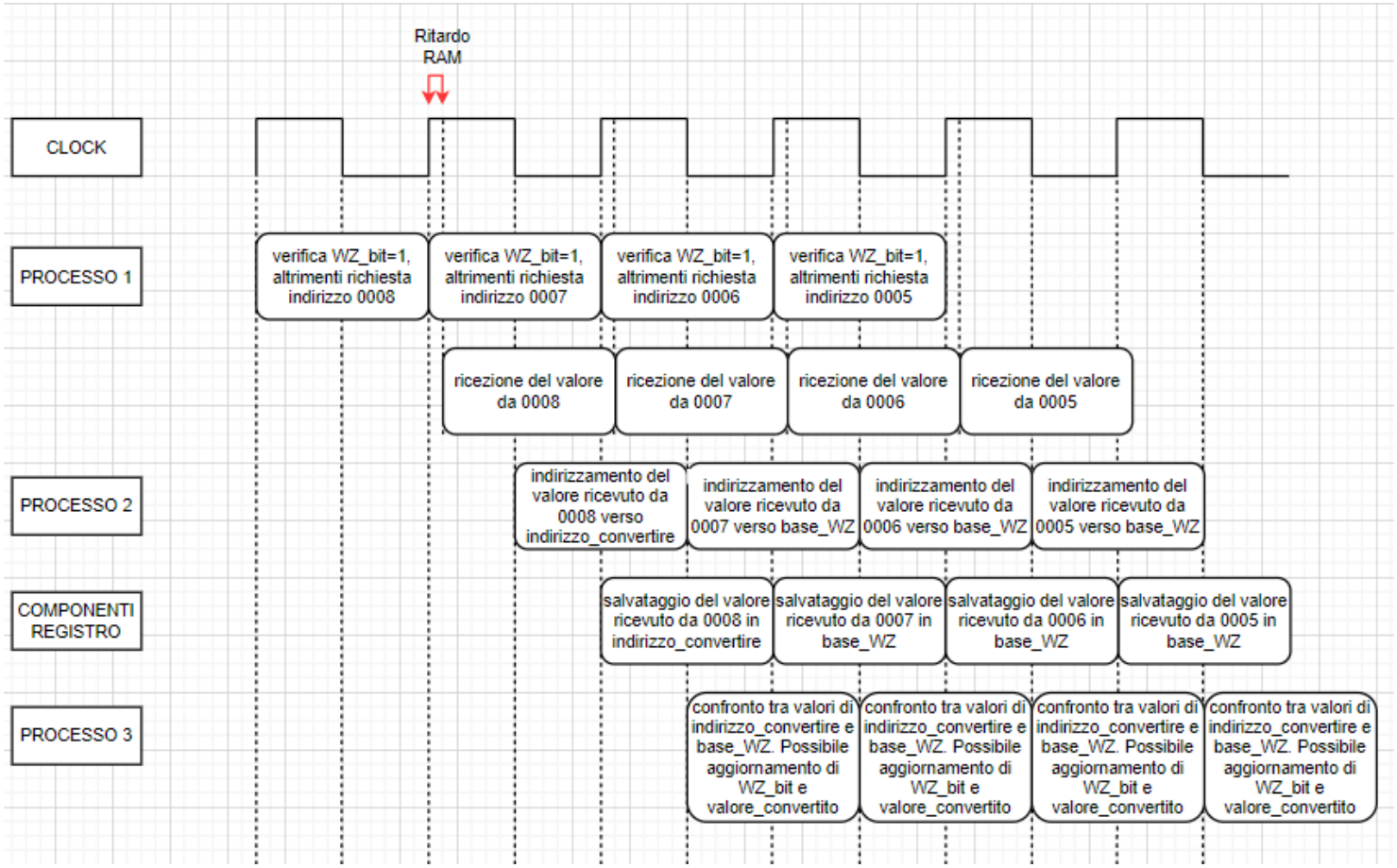
2.3.3 Segnali utilizzati

- **`ciclo_completo`**
  Segnale di tipo std_logic, viene alzato quando viene trovata una WZ valida oppure quando tutte le WZ sono state testaste ma nessuna è risultata valida. Quando `ciclo_completo` è alto, il primo processo si occupa di alzare il segnale `fine` e di riportare a valori di default il contatore e i segnali di uscita del modulo, in modo che il esso sia pronto a svolgere la computazione successiva.
- **`fine`**
  Segnale di tipo std_logic, è il segnale che controlla `o_done`. È stato introdotto per poter leggere il valore di `o_done`, cosa non possibile direttamente in quanto segnale di output, e stabilire quando abbassarlo. Si è poi rivelato utile anche per stabilire quando riportare `WZ_bit` e `valore_convertito` ai loro valori di default durante l'ultimo ciclo di esecuzione.
- **`WZ_bit`**
  Segnale di tipo std_logic, viene alzato quando un confronto tra l'indirizzo da codificare e l'indirizzo base di una WZ dà esito positivo. Quando `WZ_bit` viene posto a 1, il primo processo, invece di continuare a richiedere valori alla RAM, passa direttamente alla fase di terminazione e provvede a scrivere il risultato (ottenuto dal terzo processo) in memoria.
- **`valore_convertito`**
  Segnale di tipo std_logic_vector a 8 bit, è il segnale a cui viene assegnato il valore codificato nel caso in cui il confronto tra l'indirizzo da codificare e l'indirizzo base di una WZ dia esito positivo.
- **`indirizzo_convertire_in`**
  Segnale di tipo std_logic_vector a 8 bit, è il segnale d'ingresso al componente registro che si occupa di salvare l'indirizzo da codificare.
- **`indirizzo_convertire_out`**
  Segnale di tipo std_logic_vector a 8 bit, è il segnale d'uscita dal componente registro che si occupa di salvare l'indirizzo da codificare. Questo segnale viene usato nel confronto con gli indirizzi base delle WZ.
- **`base_WZ_in`**
  Segnale di tipo std_logic_vector a 8 bit, è il segnale d'ingresso al componente registro che si occupa di salvare l'indirizzo base della WZ che sto considerando ad un determinato ciclo di clock.
- **`base_WZ_out`**
  Segnale di tipo std_logic_vector a 8 bit, è il segnale d'uscita dal componente registro che si occupa di salvare l'indirizzo base della WZ considerata ad un determinato ciclo di clock. Questo segnale viene usato nel confronto con l'indirizzo da codificare.
- **`counter`**
  Segnale di tipo intero, è il segnale su cui si basa la richiesta dei valori alla RAM. Inoltre, dal suo valore dipende quello di WZ_num, il quale compone `valore_convertito`.

2.4 Schema di funzionamento

Viene riportato qui sotto uno schema di funzionamento del modulo. Scopo di questo esempio è mostrare in maniera chiara le modalità secondo cui i tre processi e i due registri interagiscono tra loro. Dallo schema è anche facile riconoscere l'ottimizzazione temporale di parallelismo lettura/confronti a cui ci si riferiva nella sezione 2.1.1, utilizzata per far fronte all'altrettanto evidente sfasamento temporale.

L'esempio vuole dare solo una visione generale del funzionamento, infatti si è supposto che nelle prime tre WZ non venisse trovata nessuna corrispondenza indirizzo-WZ. Ciò che succede in tale caso sarà mostrato dai diagrammi presentati nella sezione riguardante i risultati dei Test Bench.


Ritardo
RAM
CLOCK
PROCESSO 1
verifica WZ_bit=1, altrimenti richiesta indirizzo 0008
verifica WZ_bit=1, altrimenti richiesta indirizzo 0007
verifica WZ_bit=1, altrimenti richiesta indirizzo 0006
verifica WZ_bit=1, altrimenti richiesta indirizzo 0005
ricezione del valore da 0008
ricezione del valore da 0007
ricezione del valore da 0006
ricezione del valore da 0005
PROCESSO 2
indirizzamento del valore ricevuto da 0008 verso indirizzo_convertire
indirizzamento del valore ricevuto da 0007 verso base_WZ
indirizzamento del valore ricevuto da 0006 verso base_WZ
indirizzamento del valore ricevuto da 0005 verso base_WZ
COMPONENTI REGISTRO
salvataggio del valore ricevuto da 0008 in indirizzo_convertire
salvataggio del valore ricevuto da 0007 in base_WZ
salvataggio del valore ricevuto da 0006 in base_WZ
salvataggio del valore ricevuto da 0005 in base_WZ
PROCESSO 3
confronto tra valori di indirizzo_convertire e base_WZ. Possibile aggiornamento di WZ_bit e valore_convertito
confronto tra valori di indirizzo_convertire e base_WZ. Possibile aggiornamento di WZ_bit e valore_convertito
confronto tra valori di indirizzo_convertire e base_WZ. Possibile aggiornamento di WZ_bit e valore_convertito
confronto tra valori di indirizzo_convertire e base_WZ. Possibile aggiornamento di WZ_bit e valore_convertito

3 Risultati di sintesi

3.1 Risultati dei report

Il report d'utilizzo mostra che il modulo sintetizzato da Vivado faccia uso di 40 LUT e 61 FF.

Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAM	URAM	DSP
✓ synth_1	constrs_1	synth_design Complete!								40	61	0.0	0	0

In particolare:

Tipologia	Utilizzati	Disponibili	% Utilizzo
LUT	40	134600	0.03
Register as Flip Flop	61	269200	0.02
Register as Latch	0	269200	0.00

Come indicato in precedenza, il riportare i segnali ai loro valori di default in caso di reset o al termine di ogni computazione ha un impatto sul numero di FF utilizzati, questo perché si rende necessario sintetizzare i segnali come registri (che è esattamente ciò che fa Vivado). Tuttavia, vista la percentuale di utilizzo dei FF, ovvero lo 0,02% dei disponibili, queste "mancate ottimizzazioni" sono insignificanti. Le facilitazioni in termini di debugging hanno avuto invece un'importanza non indifferente.

Altro punto da sottolineare è che non siano stati generati latch.

3.2 Warning ottenuti

La sintesi del modulo ha prodotto due warning:

1. [*Synth 8-6014] Unused sequential element diff_reg was removed*

   Questo warning è legato alle linee di codice 181 e 183:

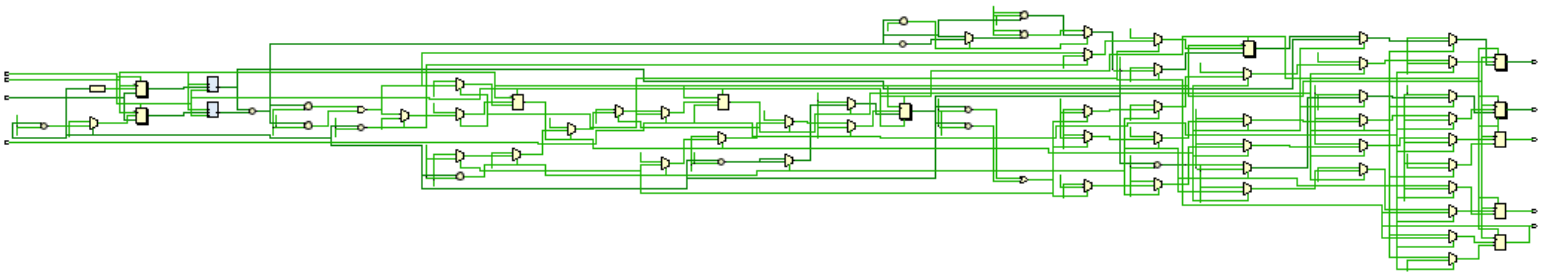
   181) diff := (signed(indirizzo_convertire_out))-(signed(base_WZ_out));
   183) if (diff>="00000000" and diff<="00000011") then ….

   Il motivo per cui si presenta è che la variabile diff non viene sintetizzata come hardware da Vivado. Diff è una variabile di supporto inserita nel terzo processo, quello che si occupa della conversione, per aumentare la leggibilità del codice. Si sarebbero potuti utilizzare direttamente i due segnali per formare le condizioni nell'if.

2. [*Constraints 18-5210] No constraints selected for write*

   Questo warning è legato al fatto che il progetto non contenga elementi di tipo Constraints Sources, ma solo Desing Sources e Simulation Sources

3.3 Schematic del modulo



4 Test Bench

4.1 Test Bench forniti

Insieme alla specifica sono stati forniti due Test Bench, per permettere di svolgere una prima serie di verifiche del funzionamento del modulo nei due casi principali di computazione: il caso in cui l'indirizzo da codificare appartenga a una delle WZ, e quello in cui non appartenga a nessuna.

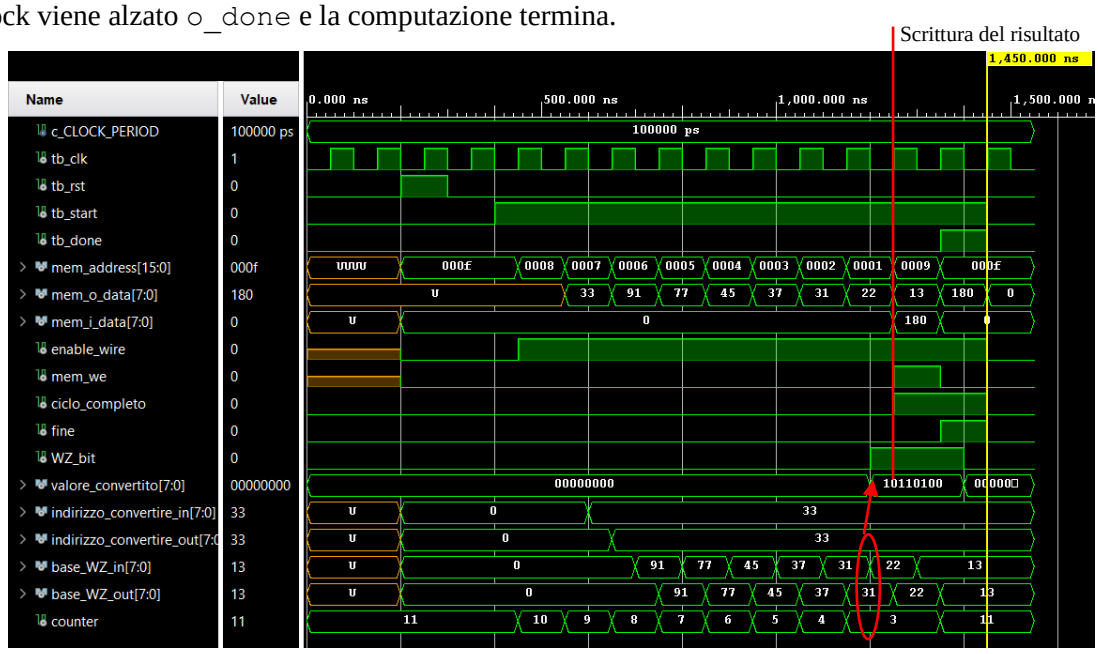
4.1.1 Indirizzo dentro una WZ

SCOPO: Lo scopo di questo test è quello di verificare che il modulo sia in grado di effettuare una codifica corretta dell'indirizzo, nel caso in cui esista una WZ a cui tale indirizzo appartiene. La tabella riporta i valori in input per questo test, così come l'output atteso.

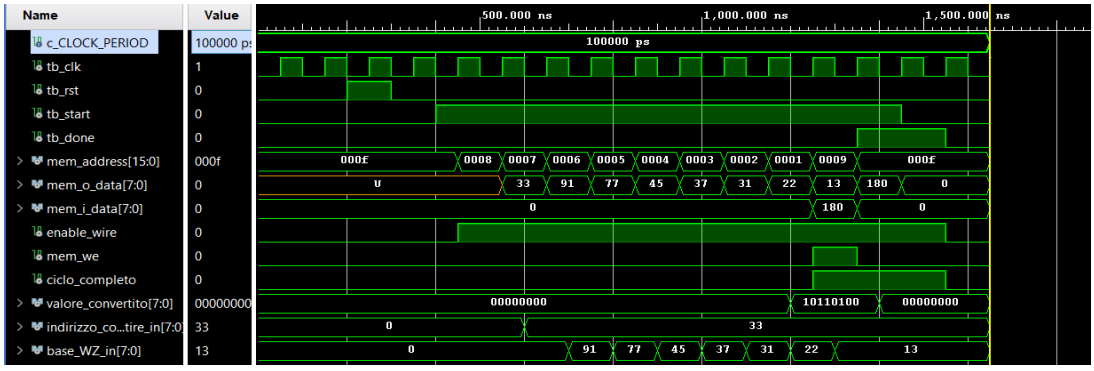
INDIRIZZO DI MEMORIA	VALORE CONTENUTO	FUNZIONE
0	4	Base WZ 0
1	13	Base WZ 1
2	22	Base WZ 2
3	31	Base WZ 3
4	37	Base WZ 4
5	45	Base WZ 5
6	77	Base WZ 6
7	91	Base WZ 7
8	33	Indirizzo da codificare
9	180 (1-011-0100)	Risultato da ottenere

RISULTATO: Il modulo supera sia il test comportamentale che il test post-sintesi, come evidenziato dalle forme d'onda riportate. Si può vedere che per `counter`=3, sul fronte di discesa del clock, `WZ_bit` viene alzato e `valore_convertito` aggiornato col valore codificato. Successivamente, sul primo fronte di salita, `ciclo_completo` e `WE` vengono alzati e ad `o_data` viene assegnato `valore_convertito`. Al successivo ciclo di clock viene alzato `o_done` e la computazione termina.

SIMULAZIONE COMPORTAMENTALE



SIMULAZIONE POST-SINTESI



4.1.2 Indirizzo fuori dalle WZ

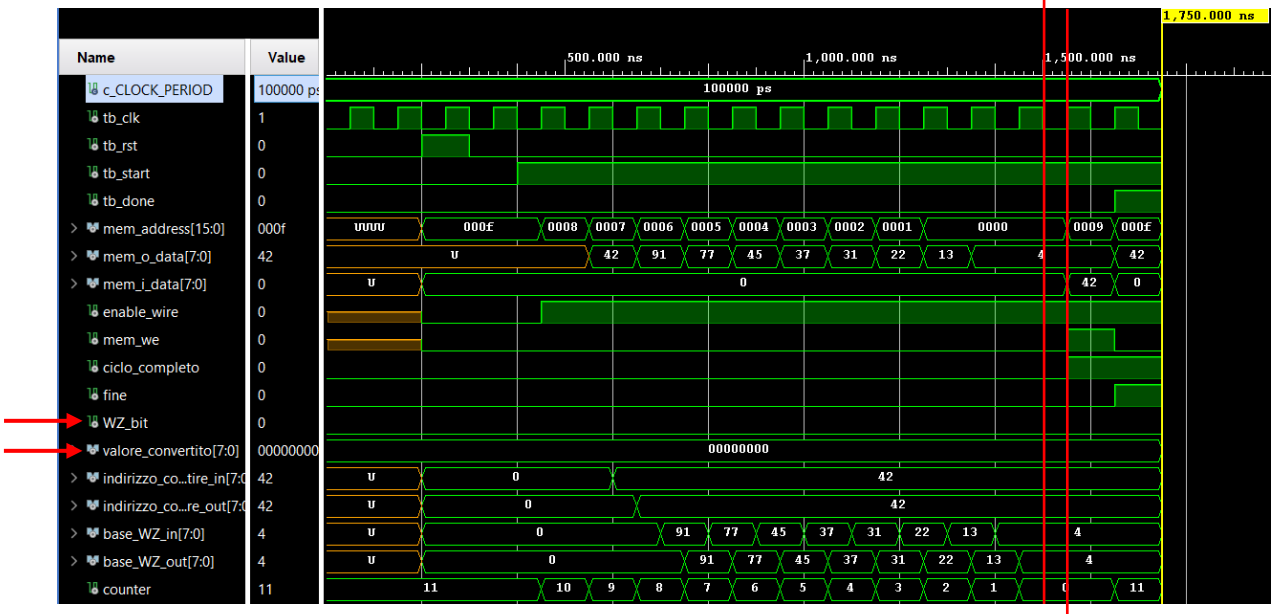
SCOPO: Lo scopo di questo test è quello di verificare che il modulo operi correttamente anche nel caso in cui l'indirizzo da codificare non appartenga a nessuna delle WZ testate. I valori in input per questo test sono gli stessi del test precedente, tuttavia cambia l'indirizzo da codificare e quindi il risultato atteso.

INDIRIZZO DI MEMORIA	VALORE CONTENUTO	FUNZIONE
0	4	Base WZ 0
1	13	Base WZ 1
2	22	Base WZ 2
3	31	Base WZ 3
4	37	Base WZ 4
5	45	Base WZ 5
6	77	Base WZ 6
7	91	Base WZ 7
8	42	Indirizzo da convertire
9	42 (0-0101010)	Risultato da ottenere

RISULTATO: Il modulo supera sia il test comportamentale che il test post-sintesi, come evidenziato dalle forme d'onda riportate. Tutte e 8 le WZ vengono testate e nessun confronto dà esito positivo. Come si può constatare, sia `WZ_bit` che `valore_convertito` rimangono fissi a 0. Il modulo si limita pertanto a scrivere in memoria uno zero concatenandogli l'indirizzo da codificare stesso, una volta effettuati tutti i confronti.

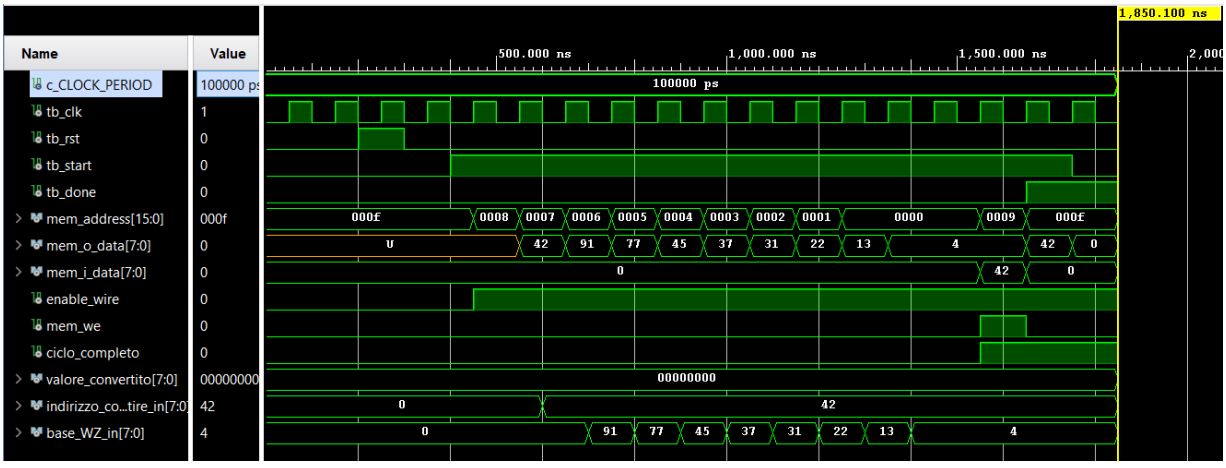
SIMULAZIONE COMPORTAMENTALE

Ultimo confronto eseguito



Scrittura del risultato

SIMULAZIONE POST-SINTESI



4.2 Test Bench personali

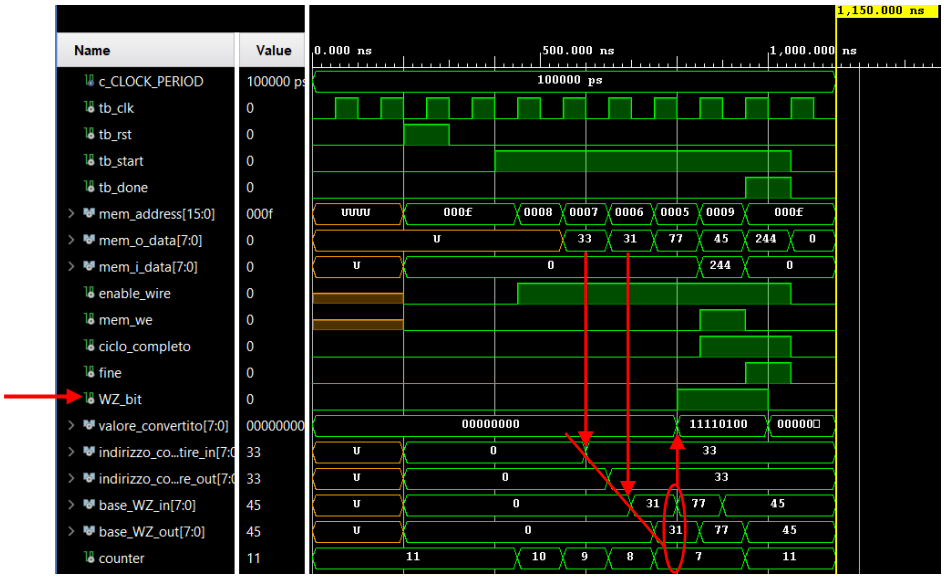
Benché utili come test iniziali, i Test Bench forniti non testano nessun caso limite. Per questo motivo è stato necessario creare un insieme di Test Bench personalizzati, i quali andassero a testare il funzionamento del modulo anche in situazioni più particolari.

Da ora in poi con "test superato" si intende che sono stati passati sia il test comportamentale che il test post-sintesi, ma verrà riportata solo la forma d'onda del test comportamentale poiché visualizza in maniera più chiara ciò che succede.

4.2.1 Prima WZ letta

SCOPO: Lo scopo di questo test è quello di verificare che il modulo sia in grado di effettuare una codifica corretta fin dal primo valore di base di WZ letto. Inoltre, il test verifica la capacità del modulo di interrompere subito le richieste di nuovi indirizzi alla RAM (tenendo ovviamente conto degli sfasamenti temporali tra la richiesta di un indirizzo e la sua eventuale codifica).

RISULTATO: Il modulo supera il test. Esso è in grado sia di effettuare "subito" (appena arriva il valore) una codifica corretta, sia di passare alla fase di scrittura appena il risultato positivo della codifica viene reso noto mediante il segnale `WZ_bit`, come indicato in figura.



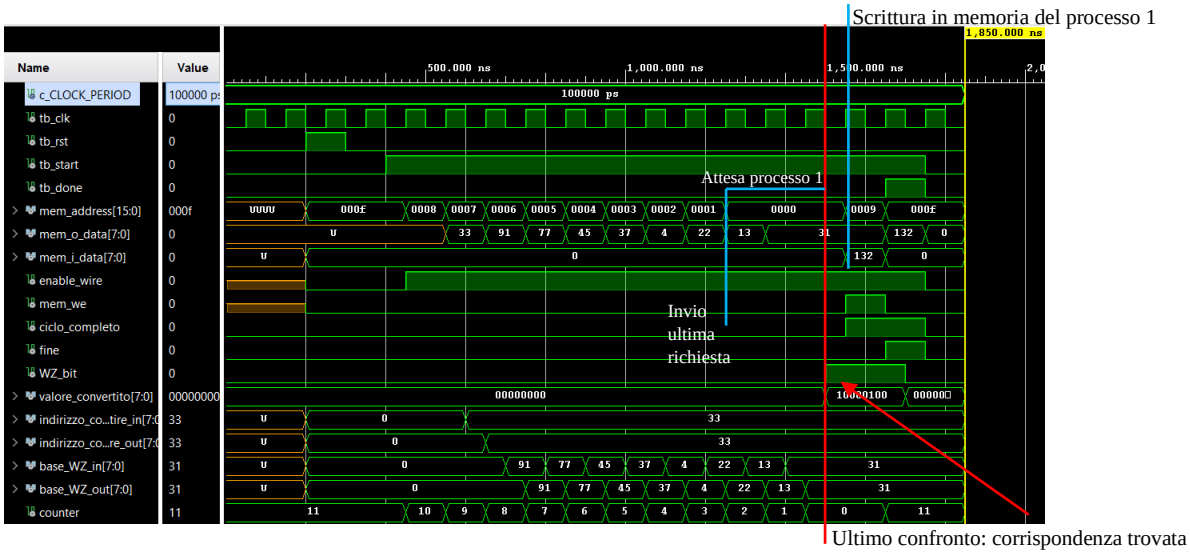
4.2.2 Ultima WZ letta

***SCOPO*:** Lo scopo di questo test è verificare che il modulo sia in grado di aspettare che anche l'ultimo confronto sia stato eseguito, e di controllarne il risultato ottenuto, prima di riportare i segnali al loro valore di default e terminare.

RISULTATO: Il modulo supera il test. Questo test è il motivo per cui il contatore parte da 11 invece che da 8 (come il numero di WZ da leggere). In particolare, durante gli ultimi due cicli di decremento del contatore il primo processo deve rimanere in attesa degli ultimi due confronti (ed eventuale codifica), che devono essere svolti dal terzo processo. Se ciò non avvenisse, il primo processo terminerebbe l'esecuzione senza aver conosciuto l'esito reale di tali confronti, e questo sarebbe un errore. Lo stesso problema si presenta nel secondo Test Bench fornito (assenza WZ), ma un settaggio errato del contatore può passare (ed era passato) inosservato perché il valore scritto in memoria non era influenzato dall'ultimo confronto, facendo risultare il test come superato nonostante l'errore.

La terza unità aggiuntiva nel contatore non riguarda il test, ma dipende dal fatto che `counter` e il primo processo vengano aggiornati entrambi sullo stesso fronte di salita, ne segue che il primo processo legga il

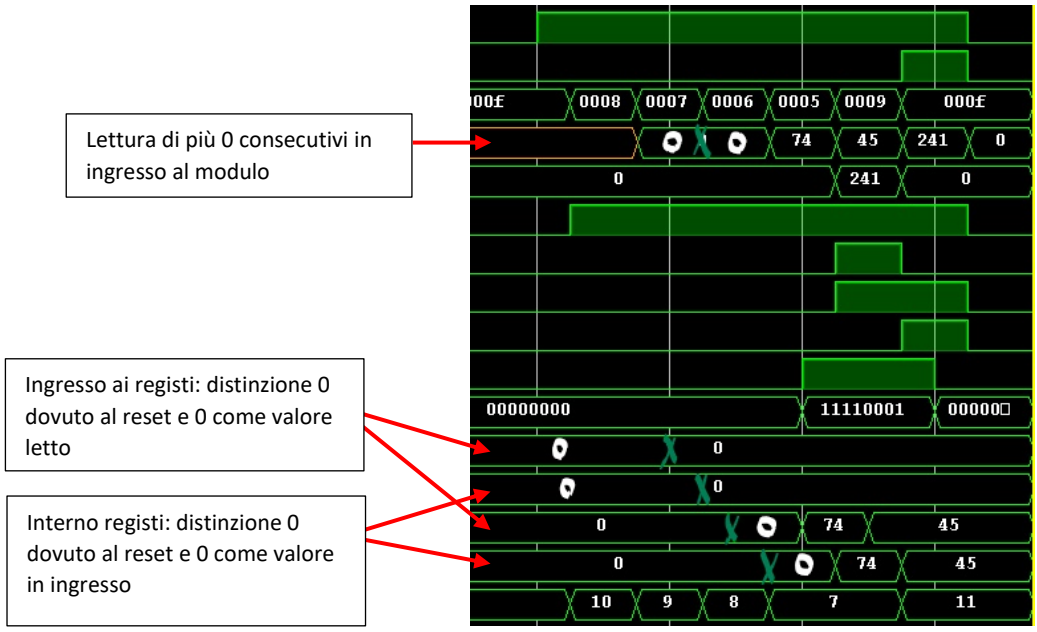
valore di `counter` prima che questo venga aggiornato. Supponendo per un attimo che gli altri due cicli aggiuntivi non esistano, il processo 1 aggiornerebbe il valore da richiedere alla RAM a 0000 ma, nel frattempo, `counter` verrebbe aggiornato da 0 a -1 (si sono voluti evitare valori negativi).



4.2.3 Distinzione dei valori di default da valori di computazione

SCOPO: Lo scopo di questo test è quello di verificare che la pratica di riportare i segnali a valori di default non generi errori, e in particolare che il modulo sia in grado di distingue gli 0 dovuti al reset dagli 0 da utilizzare nella computazione.

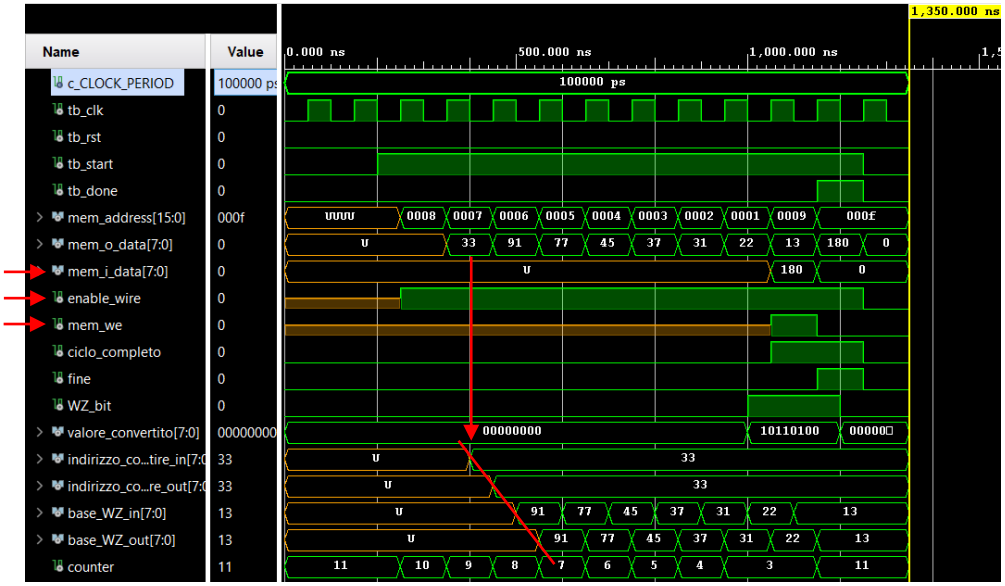
RISULTATO: Il modulo supera il test. Visto che la lettura del valore d'ingresso è legata al valore del contatore e non al valore di per sé, il modulo non ha nessun problema a capire in che momento lo 0 è un segnaposto e in che momento è un valore da utilizzare. Lo stesso principio si applica al salvataggio dei valori letti nei registri. Purtroppo, non è possibile osservare questo comportamento dalle forme d'onda perché Vivado aggiorna il valore riportato nell'onda solo quando c'è un cambio di valore, ma non un aggiornamento al valore stesso. La figura riportata mostra quello che succede davvero.



4.2.4 Assenza di Reset inziale

SCOPO: Lo scopo di questo test è quello di verificare che il modulo sia in grado di funzionare correttamente anche se al primo avvio non viene fornito un reset precedentemente allo start.

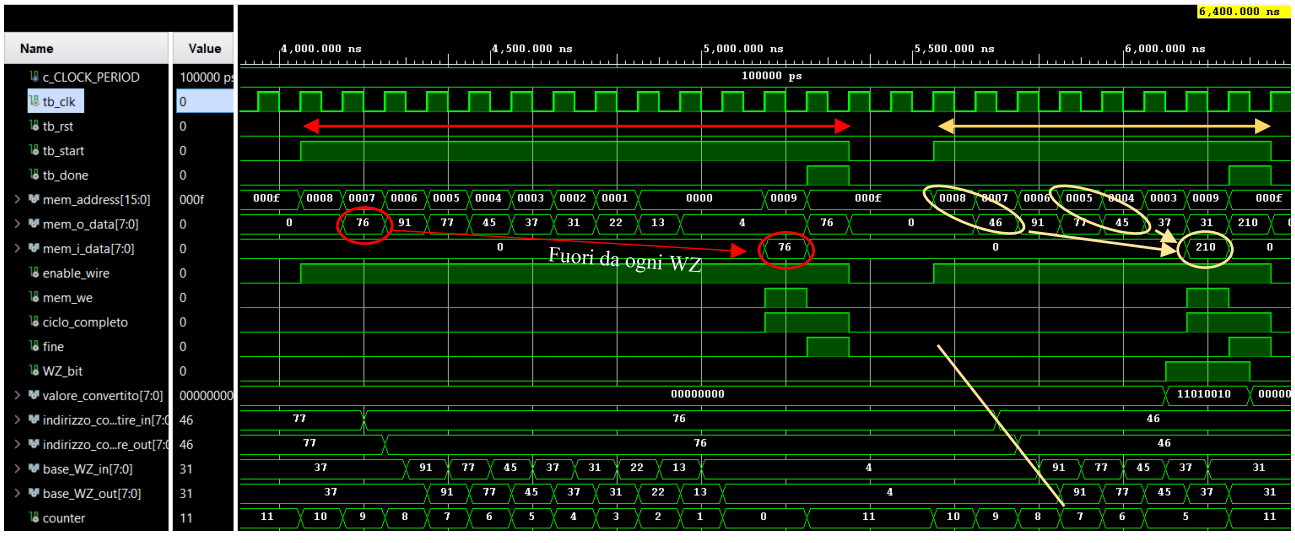
RISULTATO: Il modulo supera il test. La lettura e il salvataggio dei valori avvengono sempre e comunque sulla base del contatore, pertanto non è condizione necessaria per il corretto funzionamento che i segnali abbiano già un valore di inizializzazione dovuto ad un reset. Lo stesso vale per l'innalzamento dei segnali di enable, write enable e per l'assegnamento a `o_data`.



4.2.5 Start multipli

SCOPO: Lo scopo di questo test è verificare che il modulo sia in grado di svolgere correttamente una serie consecutiva di codifiche senza avere necessità del segnale di reset prima di ognuna di esse. La specifica asserisce che in assenza di reset le WZ non cambino, tuttavia non specifica se possa cambiare l'indirizzo da codificare. Si è assunto che l'indirizzo da codificare possa cambiare valore tra computazioni consecutive e senza bisogno del segnale di reset.

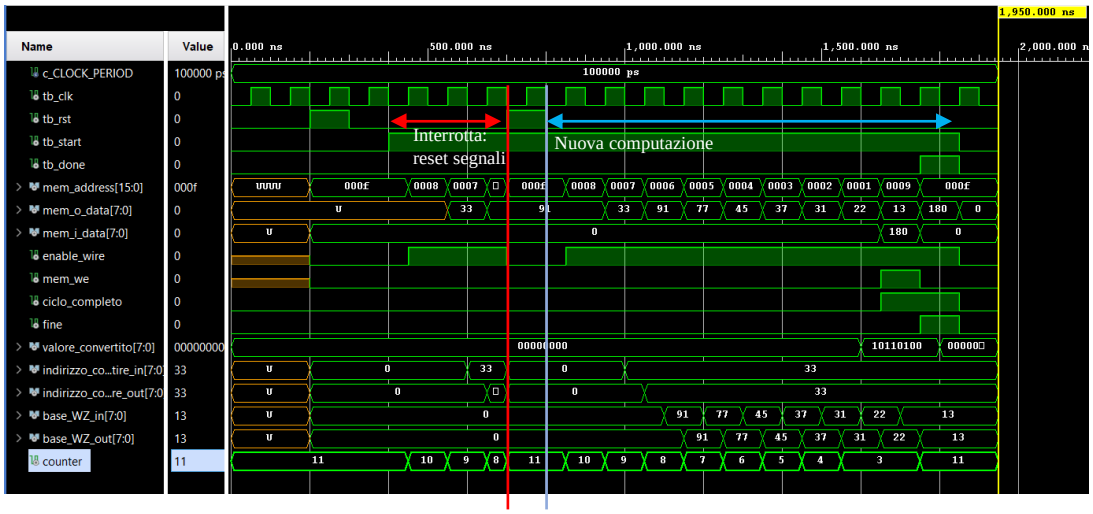
RISULTATO: Il modulo supera il test. Dopo ogni computazione, prima di alzare `o_done`, i tutti i segnali di rilievo per la computazione sono riportati a dei valori tali da poter consentire una nuova computazione, a prescindere che questa ci sia. Non è necessario resettare i valori in ingresso ai due componenti registro, né il loro valore interno, poiché, come spiegato nelle sezioni 4.2.3 e 4.2.4, la lettura e la validità di un valore sono legate al contatore. Le forme d'onda riportate mostrano un esempio di start multiplo in cui l'indirizzo da codificare iniziale (76) non appartenga a nessuna WZ (ris=76=0-1000000), mentre il secondo (46) appartenga alla quinta WZ (ris=210=1-101-0010).



4.2.6 Reset asincrono 1

SCOPO: Lo scopo di questo test è verificare che il modulo sia in grado di interrompere una computazione nel caso in cui venga alzato il segnale di reset, per poi ripartire ed essere in grado di fornire il valore corretto in uscita. In questa versione del test, il reset viene alzato prima che venga trovata un WZ valida.

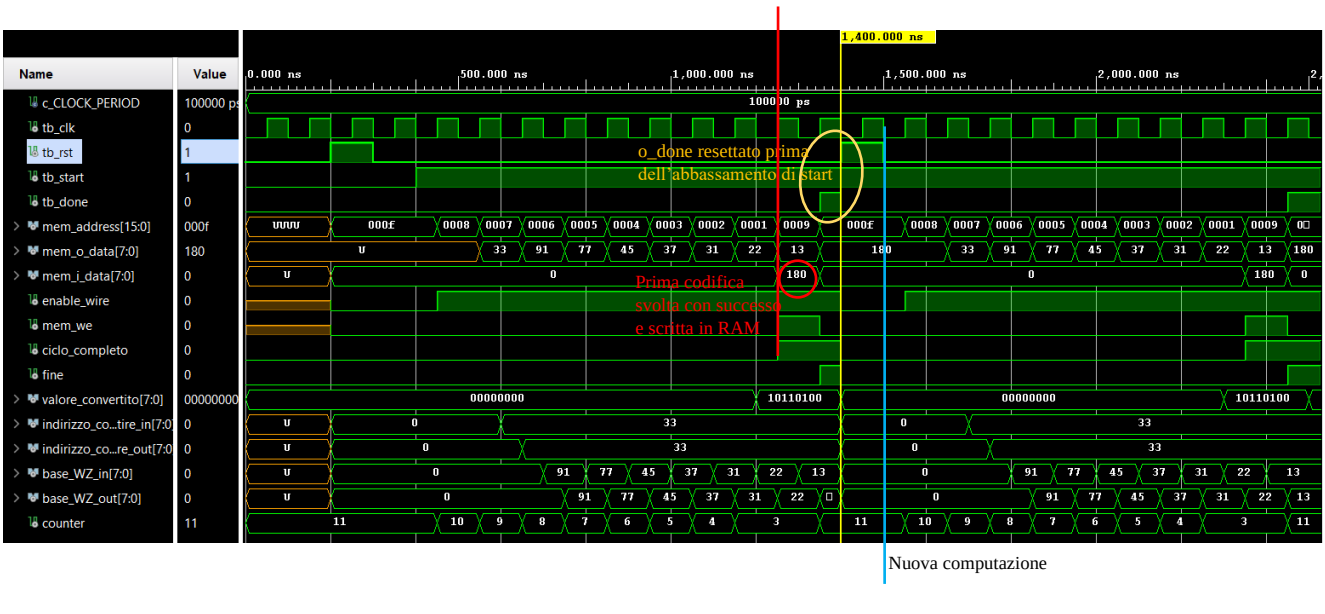
RISULTATO: Il modulo supera il test. Non appena reset diventa 1, tutti i segnali vengono reimpostati. Visto che `o_done` non era mai stato alzato, start rimane alto, e non appena reset viene abbassato la computazione riparte da capo con i nuovi valori, fino a terminare. Supponendo di mantenere la convenzione secondo cui il risultato viene verificato all'abbassamento di start, la codifica da controllare in memoria è quella relativa alla computazione successiva al reset asincrono. Se il controllo fosse fatto prima del termine della computazione post-reset, in RAM si troverebbe il valore di inizializzazione della cella, non essendo state fatte scritture.



4.2.7 Reset asincrono 2

SCOPO: Lo scopo di questi test è verificare che il modulo sia in grado di interrompere una computazione nel caso in cui venga alzato il segnale di reset, per poi ripartire ed essere in grado di fornire il valore corretto in uscita. In questa versione del test, il modulo viene sottoposto al caso estremo in cui, non solo il reset venga alzato dopo che sia stata trovata una corrispondenza indirizzo-WZ valida, ma anche dopo che `o_done` sia stato alzato, e tuttavia prima che start venga abbassato dal Test Bench.

RISULTATO: Il modulo supera il test. Poiché start non fa in tempo ad essere abbassato prima che `o_done` e tutti gli altri segnali vengano resettati, esso rimane alto. Per questo motivo, quando reset verrà abbassato, una nuova computazione inizierà da capo come nel caso precedente. Anche in questo caso la verifica del risultato va fatta sulla seconda computazione, ma se si andasse a vedere il valore in memoria prima della fine di questa, il valore trovato sarebbe quello relativo alla prima codifica, essendo stata effettuata la scrittura.



4.2.8 Test a ripetizione

SCOPO: Questo test serve a simulare il funzionamento pseudo-reale del modulo. Il test è composto da molte computazioni successive, in cui ciclicamente si presentano start multipli e reset asincroni, ed in cui i valori cambiano da computazione a computazione, andando a testare così entrambi i casi di presenza/non presenza di una corrispondenza indirizzo-WZ. Questo test è sostanzialmente un assortimento dei test precedenti, quindi riportare nuovamente le forme d'onda sarebbe ridondante.

RISULTATO: Il test viene superato.

5 Conclusioni

Riassumendo, il codice VHDL scritto permette a Vivado di sintetizzare un modulo HW in grado di realizzare la codifica WZ per indirizzi a 7 bit, e capace di superare tutti i test comportamentali e post-sintesi a cui è stato sottoposto. Il modulo è ottimizzato rispetto al fattore tempo, ciò nonostante occupa anche un’area molto piccola.