Prova Finale (Progetto di Reti Logiche)

Andrea Altomare [Cod. Persona: 10608961 - Matricola: 891365]

30 Marzo 2020

Politecnico di Milano

Professore: William Fornaciari

Indice

Sommario

Obiettivo di tale prova è la descrizione VHDL (*VHSIC Hardware Description Language*) di un modulo hardware basato su FPGA volto alla realizzazione del metodo di codifica a bassa disspazione di potenza basato su "Working Zones" [1]).

1 Introduzione

Tale metodo di codifica è pensato per il Bus Indirizzi: si utilizza per tradurre un indirizzo, quando questo viene trasmesso, secondo uno schema di codifica basato su intervalli (detti, appunto, Working Zones). In particolare, la specifica prevede un determinato numero di intervalli di dimensione fissata (Dwz) i quali hanno un certo indirizzo di base. All'interno dello schema di codifica possono esistere molteplici Working Zones (Nwz). La particolare specifica da implementare prevede, per ogni ciclo di elaborazione, un indirizzo da codificare (tradurre) e 8 Working Zones. Ogni intervallo ha dimensione 4, e gli indirizzi che ne fanno parte sono contigui a partire dall'indirizzo di base (che è incluso nell'intervallo); è garantito inoltre che le Working Zones non si sovrappongono. Di seguito, lo schema concettuale delle Working Zones:

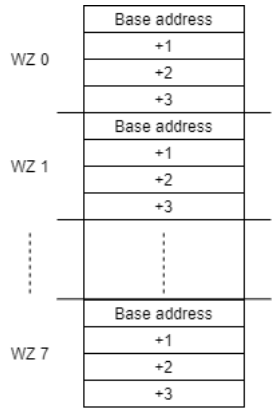


Figura 1: Schema concettuale Working Zones

Il protocollo è il seguente:

- Se l'indirizzo da trasmettere (ADDR) non appartiene ad alcuna Working Zone (*WZ*), esso viene semplicemente ritrasmesso in output senza nessuna variazione. Il bit addizionale (posto come MSB) WZ_BIT assume il valore 0 ed indica che non vi è stata alcuna operazione di codifica. L'output assumerà la seguente forma: WZ_BIT & ADDR (& rappresenta il simbolo di concatenazione);
- Se l'indirizzo da trasmettere (ADDR) appartiene ad una WZ, si procede con la traduzione. In output si avrà un indirizzo formato da tre gruppi di bit: WZ_BIT (posto a 1), WZ_NUM (rappresentante la codifica in binario naturale del numero della Working Zone coinvolta) e WZ_OFFSET (ovvero l'offset di ADDR rispetto all'indirizzo base della WZ alla quale ADDR appartiene) codificato come *one-hot*. L'output assumerà la seguente forma: WZ_BIT & WZ_NUM & WZ_OFFSET.

Tutti gli indirizzi iniziali (basi e address da codificare) hanno dimensione pari a 7 bit, il MSB sarà sempre 0.

Il modulo, a seguito di un segnale di *START*, dovrà leggere da memoria l'indirizzo da codificare e i base address delle WZ, dunque dovrà eseguire la traduzione e scrivere in RAM il risultato. Infine asserirà un segnale di *DONE*, che rimarrà alto fino all'abbassamento di *START*.

La memoria al quale si fa riferimento è una RAM con indirizzamento al Byte (celle da 8 bit ciascuna) da 64 KiB (16 bit per indirizzare ciascuna delle 2^{16} celle).

Gli indirizzi base delle Working Zones non cambiano tra un ciclo di codifica e un altro: tale situazione può verificarsi solo a seguito di un segnale di *RESET*.

2 Architettura

2.1 Considerazioni generali

Il flusso di progettazione dell'hardaware è basato su una logica *top-down* e vede una prima fase di analisi della specifica nella quale ho fissato gli obiettivi nel dettaglio del modulo da realizzare: il design del componente si divide in due macro parti, una macchina a stati finiti (FSM, "*Finite State Machine*") responsabile di gestire gli stadi della procedura di traduzione, ed il Datapath, al quale è affidato il compito di implementare il sistema algoritmo di codifca; segue dunque il momento della progettazione in cui ho definito le caratteristiche generali che avrebbero dovuto avere la FSM ed il datapath.

Mi sono concentrato in particolare sulla velocità di traduzione effettuando un trade-off e, a tale scopo, impiegando più spazio; le ragioni di questa scelta sono prevalentemente due:

- Essendo, il prodotto finale, un componente hardware, il contesto di lavoro in cui dovrebbe essere inserito prevede il privilegiare l'aspetto temporale rispetto a quello spaziale: le prestazioni temporali a basso livello, differentemente da quelle di un sistema software ad alto livello, devono essere quanto più possibile elevate;
- L'impiego di un maggiore spazio, in questo specifico caso, non fa degenerare né i costi di produzione della possibile realizzazione fisica su FPGA, né i costi relativi al consumo energetico, pertanto il compromesso effettuato risulta giustificato.

La strategia implementata si basa sul fatto che, all'interno di una stessa esecuzione, gli indirizzi base delle WZ non cambiano. Diverse esecuzioni sono scandite da segnali di *RESET*, per cui, per velocizzare le traduzioni dopo aver svolto la prima, è sufficiente salvare i suddetti indirizzi in appositi registri per poterli riutilizzare ogni volta.

Ogni ciclo di codifica inizia con un segnale di *START*, una volta caricati dalla memoria tutti gli indirizzi necessari, si procede con l'algoritmo di traduzione vero e proprio. Per ogni WZ:

1. Si sottrae l'indirizzo base dall'ADDR da codificare, tale operazione è svolta con segno (complemento a 2);
2. Il risultato della sottrazione è utilizzato come segnale di selezione in un multiplexer (MUX) per stabilire se ADDR si trova o meno nella WZ considerata;
3. Dunque tramite un altro MUX si sceglie la codifica per ADDR.

Una volta ottenuta la traduzione, viene eseguita la scrittura del risultato in RAM.

Il processo di codifica viene effettuato in una singola passata in parallelo con tutte le WZ: i componenti necessari allo svolgimento dei passaggi sopra descritti sono riportati per ogni Working Zone, in questo modo è possibile in soli due cicli di clock stabilire se un indirizzo è parte o meno di un certo intervallo e codificarlo opportunamente.

È opportuno infine notare che l'architettura di entrambe le *entity* è stata codificata mediante una descrizione VHDL di tipo *Behavioral*.

2.2 Moduli descritti

2.2.1 entity project_reti_logiche

Il modulo principale che implementa l'interfaccia (ingressi e uscite) del sistema complessivo con l'esterno (in particolare, con il testbench) e la FSM. Essa riceve i segnali dall'esterno e svolge le opportune computazioni. In particolare al suo interno sono descritti tre processi che implementano, rispettivamente, il registro per il salvataggio dello stato corrente, la funzione di stato prossimo Delta la quale calcola e scandisce le transizioni di stato, e la funzione d'uscita Lambda: è calcolata a partire dal solo stato, in modo da realizzare una macchina di *Moore*, ed asserisce opportunamente i segnali di controllo per il Datapath. Questa entità è inoltre collegata al suo interno con un altro modulo: il Datapath.

Gli stati della FSM sono 14.

idle e ready: I primi due stati sono quelli di inizio computazione: la macchina è in attesa del primo segnale di *START*. La differenza sta nel fatto che, dovendo inizialmente memorizzare tutti gli indirizzi, dallo stato di *idle* si passerà a quello di fetch della prima WZ, mentre dallo stato di ready si passerà direttamente al fetch dell'ADDR da codificare. Dopo un segnale di *RESET* si tornerà dunque sempre allo stato di *idle*; mentre, dopo la prima computazione, se non occorrono segnali di *RESET* si tornerà sempre allo stato di *ready*, in modo da sfruttare gli indirizzi base già memorizzati.

fetch: Gli stati *ftch1, ftch2, . . . , ftch8* servono a leggere da RAM e salvare le WZ, quindi in *ftchAddr* si procede alla lettura dell'indirizzo da tradurre.

encode: Lo stato *encode* rappresenta il cuore del meccanismo di traduzione e memorizza nei registri intermedi del datapath le codifiche parziali;

output: nello stato di *output* si prepara la scrittura in memoria del risultato della traduzione. Questi due stati in particolare potrebbero essere riuniti in un unico "macrostato", tuttavia, come spiegato in seguito nella descrizione del datapath, dividere la fase di codifica dalla fase di output ha dei vantaggi.

finish: L'ultimo stato è quello di *finish*, nel quale si asserisce semplicemente il segnale di *DONE* per comunicare che la computazione è terminata, come richiesto da specifica.

È da osservare, ai fini della comprensione della transizione di stato quando il componente viene testato, che i vari *process* sono sincronizzati sul fronte di discesa del clock, e non, come più di consueto, su quello di salita. La funzione VHDL utilizzata in particolare è "*falling_ edge(i_ clk)*", ed è utile a non avere problemi quando si interagisce con la memoria RAM: così facendo si può infatti richiedere un dato e salvarlo in un registro nello stesso ciclo di clock (l'interazione con la memoria impiega 1 ns, leggendo il dato sul fronte di discesa si riesce ad attendere questo lasso di tempo senza impiegare uno stato in più della FSM).

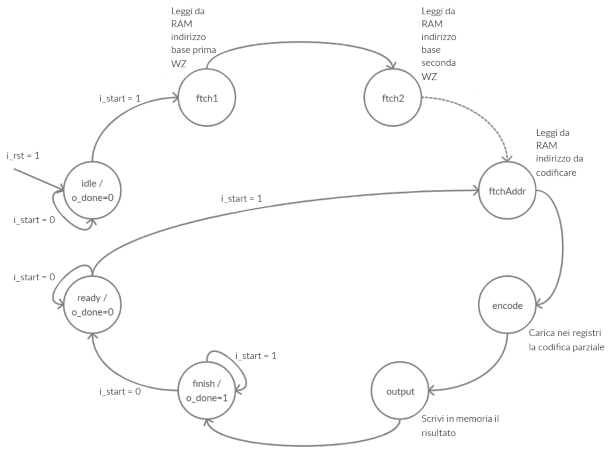


Figura 2: Macchina a Stati Finiti

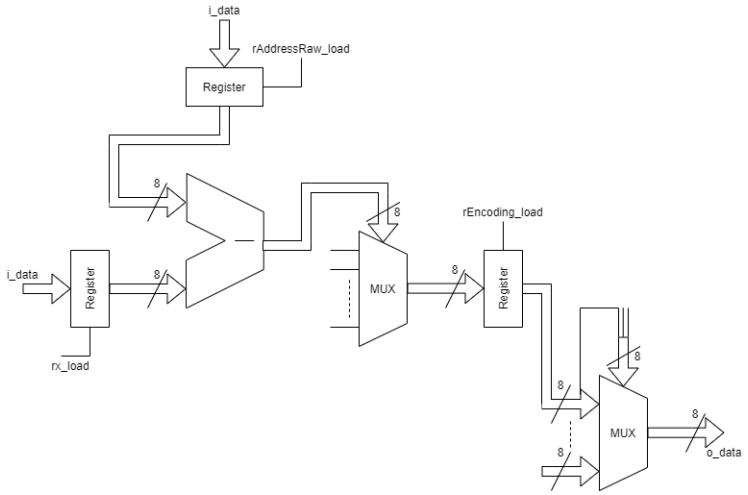
2.2.2 entity datapath

Questo modulo implementa l'algoritmo di codifica vero e proprio: si basa sul procedimento sopra descritto. Oltre al registro per memorizzare l'ADDR da tradurre, per ogni Working Zone sono descritti, tramite opportuni processi, i seguenti componenti: un registro per l'indirizzo di base della WZ; un Subtractor per effettuare la sottrazione tra i due indirizzi; un Multiplexer per la selezione di una *codifica parziale* che verrà

salvata in un apposito registro; e un ultimo Multiplexer che permetterà di scegliere infine la codifica corretta per l'ADDR considerato.

In particolare, è opportuno notare che il segnale di selezione del primo MUX è dato proprio dal risultato della sottrazione: se tale risultato è un numero compreso fra 0 e 3 (inclusi), vuol dire che ADDR è nella WZ, dunque verrà selezionata direttamente la sua codifica finale; ciò è possibile poiché ogni WZ ha la propria "catena" di componenti, pertanto il MUX *numero uno* selezionerà la codifica in binario naturale del numero uno (001), il MUX *numero due* selezionerà la codifica 002 e così via. Il WZ_OFFSET verrà scelto in base al risultato della sottrazione: non essendo possibile utilizzare direttamente tale valore (poiché WZ_OFFSET deve essere codificato in one-hot), si sceglie l'equivalente one-hot della relativa codifica binaria (es: $01 \implies 0010, 11 \implies 1000$). Il WZ_BIT sarà sempre 1 in questi casi. Se il risultato non è compreso fra 0 e 3 (il sistema riconosce anche risultati negativi, essendo la sottrazione imposta con tipi numerici *signed*) verrà semplicemente riportato un codice costituito solo da 0.

Queste codifiche parziali sono salvate in un registro di appoggio: questa scelta progettuale è stata effettuata poiché, in questo modo, è possibile ottenere frequenze di funzionamento più alte per il sistema. Se tutte le operazioni venissero effettuate in una volta, sarebbe necessario un periodo di clock più lungo rispetto a quello che si può ottenere "spezzando" la procedura di codifica in più stadi. L'ultimo passaggio è svolto dal secondo MUX, il quale riceve come segnale di selezione un bus di 8 bit costituito dai MSB delle codifiche parziali calcolate in precedenza: dato che le Working Zones non possono essere sovrapposte (da specifica), se l'indirizzo finale è già stato calcolato, esso è anche unico, dunque la posizione dell'unico bit a 1 stabilirà anche il numero del registro dal quale prendere l'indirizzo correttamente tradotto; se invece non vi è alcun bit a 1 significa che ADDR non appartiene ad alcuna WZ, pertanto verrà semplicemente dato, così come è, in uscita.

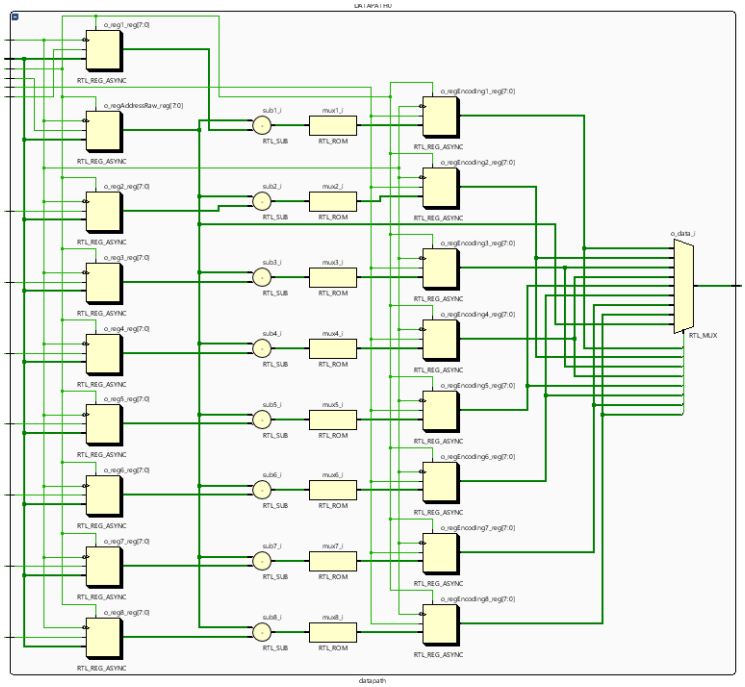


3 Risultati sperimentali

Di seguito sono riportati i risultati ottenuti dopo aver opportunamente scritto il codice VHDL, utilizzando l'ambiente *Xilinx Vivado 2019.2*.

3.1 RTL Analysis

Osservando lo *Schematic* in *RTL ANALYSIS* e concentrandosi sul datapath si può notare come i componenti logici indotti da Vivado in seguito all'analisi del codice sono esattamente quelli previsti in fase di design del sistema: la struttura rispecchia fedelmente lo schema sopra riportato.



3.2 Sintesi

La sintesi del componente procede senza errori, dunque quanto scritto in codice VHDL è correttamente sintetizzabile da Vivado. La scheda FPGA utilizzata (target) è la *xc7a200tfbg484-1*. I *Warning* rilevanti ottenuti, visibili nella scheda *"Messages"*, sono relativi al fatto che la porta *o_address* del componente (il bus di 16 bit in cui va posto l'indirizzo della cella di RAM in cui leggere/scrivere) ha i bit dal 4 al 15 posti costantemente a 0. Ciò è dovuto al fatto che la RAM, come spiegato precedentemente, è da 64 KiB, dunque necessita di 16 bit per indirizzare ogni cella; tuttavia gli effettivi indirizzi usati sono quelli che vanno da 0 a 9, cosicché bastano i soli bit da 0 a 3 di *o_address* per rappresentarli: i restanti non verranno mai utilizzati.

> [Synth 8-3917] design project_reti_logiche has port o_address[15] driven by constant 0 (11 more like this)

La FSM è inferita da Vivado codificando lo stato con una codifica *binario naturale*. Ciò è dovuto al fatto che 14 è un numero potenzialmente elevato di stati per essere codificato in *one-hot* (servirebbero più Flip-Flop), per cui il tool di sintesi non opererà tale scelta.

```
INFO: [Synth 8-802] inferred FSM for state register 'cur_status_reg' in module 'project_reti_logiche'
---------------------------------------------------------------------------------------------------
                   State |                     New Encoding |               Previous Encoding
---------------------------------------------------------------------------------------------------
                    idle |                             0000 |                             0000
                   ftch1 |                             0001 |                             0010
                   ftch2 |                             0010 |                             0011
                   ftch3 |                             0011 |                             0100
                   ftch4 |                             0100 |                             0101
                   ftch5 |                             0101 |                             0110
                   ftch6 |                             0110 |                             0111
                   ftch7 |                             0111 |                             1000
                   ftch8 |                             1000 |                             1001
                ftchaddr |                             1001 |                             1010
                  encode |                             1010 |                             1011
                  output |                             1011 |                             1100
                  finish |                             1100 |                             1101
                   ready |                             1101 |                             0001
---------------------------------------------------------------------------------------------------
INFO: [Synth 8-3354] encoded FSM with state register 'cur_status_reg' using encoding 'sequential' in module 'project_reti_logiche'
---------------------------------------------------------------------------------------------------
```

Infine, è opportuno notare che, al termine del processo di sintesi, il device ottenuto sarà composto complessivamente da 137 LUT (*Look-up Tables*) e 116 FF (*Flip-Flop*). Tali numeri possono essere giustificati dal fatto che, come spiegato nelle considerazioni sulla fase di design, effettuando tutte le codifiche in parallelo

è necessario avere i componenti che operano l'algoritmo di traduzione per ogni WZ. Dunque si occupa più area, ma le computazioni saranno più veloci.

Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF
synth_1	constrs_1	synth_design Complete!								137	116

4 Simulazioni

I test bench realizzati mirano a verificare che l'algoritmo di traduzione (dunque il datapath) e la FSM funzionino correttamente.

Per il datapath i test bench realizzati sono pochi: per l'implementazione realizzata è sufficiente infatti verificare che la codifica funzioni anche con valori agli estremi dell'intervallo di quelli possibili per affermare che funzionerà anche con tutti gli altri. Inoltre ciò che è cruciale controllare è il sistema di sottrazione (in complemento a 2), in quanto è il perno su cui poggia concettualmente tutto il flusso di codifica.

Quanto necessario per testare il giusto funzionamento della FSM è invece più complesso, poiché ha a che vedere con la temporizzazione dei segnali inviati alla RAM, la corretta suddivisione delle fasi della computazione nei relativi stati della macchina, e l'interazione con il modulo che si occupa di testarla (il test bench).

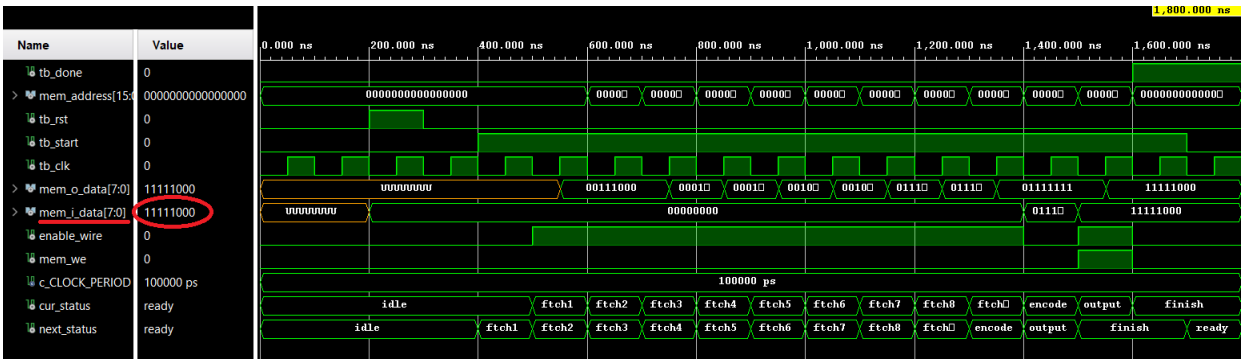
Di seguito sono riportate le descrizioni dei casi di test più significativi. Tutti i test realizzati sono stati passati con successo dal componente realizzato, sia in pre-synthesis (*Behavioral Simulation*) che in post-synthesis (*Post-Synthesis Functional Simulation*).

4.1 Tb_Datapath_1

Il valore da codificare è 127 (7 bit posti a *'1'*) e il primo confronto viene effettuato con l'indirizzo base 0 (tutti i bit posti a *'0'*). ADDR in questo caso non appartiene ad alcuna WZ, per cui in uscita, in RAM(9) dovrà essere presente proprio il valore iniziale 127 (*"0 - 1111111"*).

4.2 Tb_Datapath_2

Il valore da codificare è 127, stavolta però appartiene alla ottava WZ ($WZ\_NUM = 7$). L'indirizzo base di tale WZ è 124 (il massimo valore per un indirizzo base, da specifica); il valore ottenuto in uscita è 248 (*"1 - 111 - 1000"*).

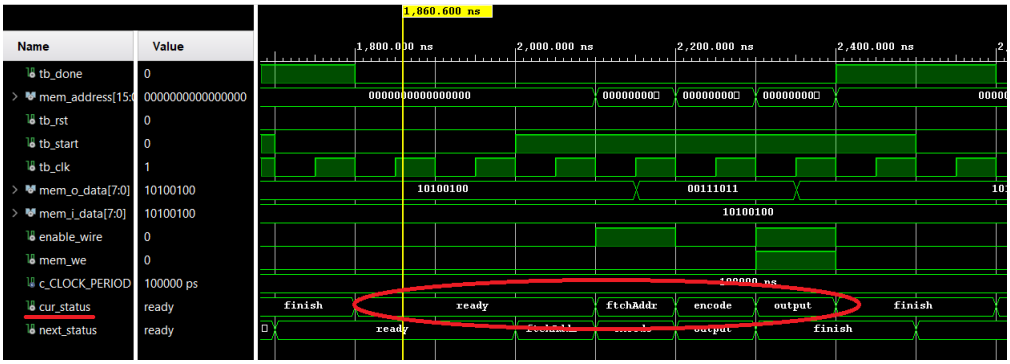


4.3 Tb_Datapath_3

Altri test sono stati condotti con valori da codificare casuali, in modo da avere una maggiore certezza del regolare funzionamento dell'algoritmo di codifica e dell'entity datapath.

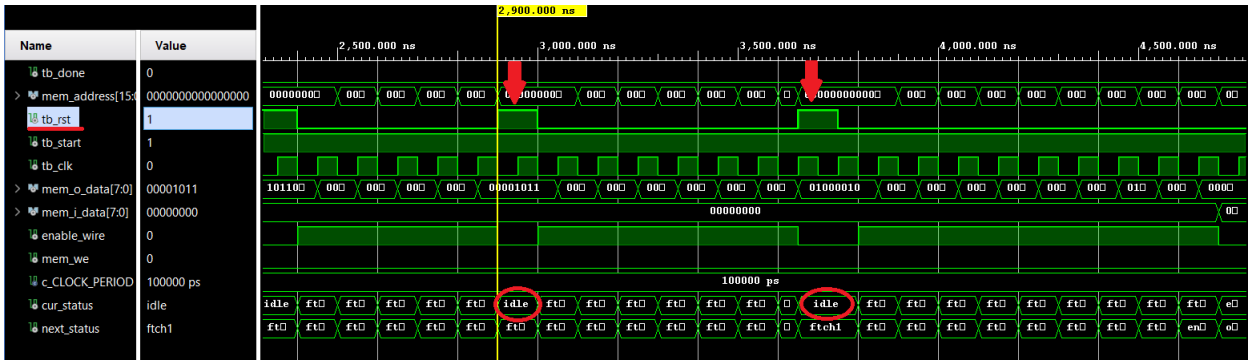
4.4 Tb_FSM_1_multi_start

Il primo test per la FSM prevede un segnale di *START* dato più volte (senza cambiare il contenuto della RAM), in modo da verificare che, se non vi sono segnali di *RESET*, la macchina a stati partirà con le successive computazioni semplicemente dalla fase di fetch di ADDR.



4.5 Tb_FSM_2_reset_asynchronous

Questo test bench mira a verificare due elementi essenziali: il primo è quello per cui dato un segnale di *RESET* la macchina si riporti nello stato iniziale, di *idle*, in attesa di un nuovo segnale di *START* (dopo il quale dovrà ricominciare la computazione innanzitutto eseguendo la fetch degli indirizzi base di tutte le WZ); il secondo è invece relativo al regolare funzionamento *indipendentemente da quando venga dato il segnale di RESET*: che tale input arrivi durante la fase di codifica, in un periodo in cui il *CLOCK* è alto o basso, sul fronte di salita o di discesa, non deve interferire con il corretto funzionamento del sistema, il quale si dovrà in ogni caso riportare stabilmente in stato di *idle*.

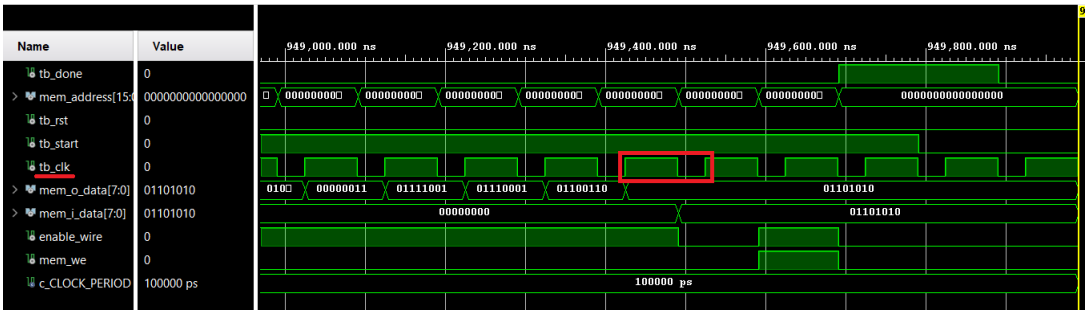


4.6 Tb_FSM_3_500MHz

Il codice di questo caso di test permette di capire fino a quale frequenza può funzionare il componente. Nello specifico, il periodo di clock è di 2 ns, ed è anche il periodo minimo al quale il modulo progettato può funzionare in post-synthesis. Ciò vuol dire che la frequenza di funzionamento massima è di 500 MHz.

4.7 Tb_FSM_4_Different_Duty_Cycle

Questo test bench verifica che il modulo funzioni bene anche con un Duty Cycle del clock diverso dal 50%.

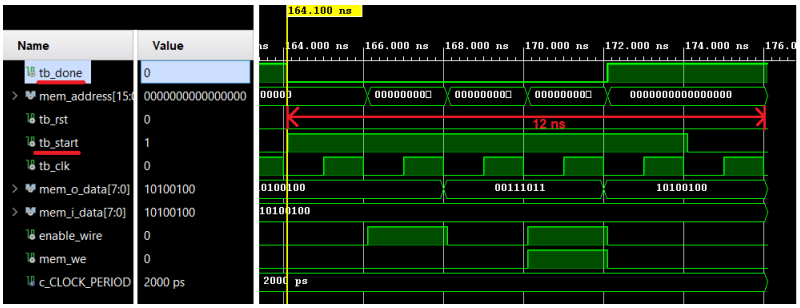


Il seguente frammento di codice mostra come si è potuto manipolare tale parametro del segnale di *CLOCK*.

```
p_CLK_GEN : process is
begin
    wait for c_CLOCK_PERIOD/3;
    tb_clk <= not tb_clk;
    wait for 2*(c_CLOCK_PERIOD/3);
    tb_clk <= not tb_clk;
end process p_CLK_GEN;
```

5 Conclusioni

Il design realizzato privilegia il risparmio di tempo rispetto al consumo di area: ciò permette di ottenere migliori performance in diversi tipi di sistemi (ipotizzando un utilizzo reale del modulo), ad esempio in soluzioni dove il fattore temporale è critico per il corretto funzionamento. Inoltre, un altro importante vantaggio nella computazione "in parallelo" dei confronti di ADDR con tutte le WZ sta nel fatto che la codifica richiederà sempre lo stesso tempo, permettendo una più semplice integrazione in un sistema ed un'analisi di funzionamento più immediata in un contesto reale. In definitiva, il *caso pessimo* corrisponde al *caso medio* e al *caso ottimo* di computazione; posto che la prima traduzione sia già stata effettuata, e quindi le WZ siano già state memorizzate, il componente impiegherà, alla massima frequenza (500 MHz), un tempo di soli 12 ns per eseguire una codifica.



Riferimenti bibliografici

[1] E. Musoll, T. Lang and J. Cortadella, "Working-zone encoding for reducing the energy in microprocessor address buses," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 6, no. 4, pp. 568-572, Dec. 1998.