

**Prova Finale di Reti Logiche**

**15/05/2020**

Autore : Andrea Sestito

Matricola: 869695

Codice persona: 10559959



**Introduzione**

**Obiettivo della Prova Finale (Progetto di Reti Logiche) consiste nella descrizione di un componente logico in VHDL, il cui comportamento simuli una soluzione pensata per il trasferimento degli indirizzi all’interno dei microprocessori, con lo scopo di minimizzare le attività dei bus interni ed avere quindi una riduzione della dissipazione di potenza.**

**La simulazione del funzionamento prevede che dato un indirizzo (ADDR)da 7 bit, letto dall’indirizzo 8 della RAM, esso venga trasmesso attraverso due tipologie di codifica diverse, dipendentemente dal fatto che appartenga oppure no ad una Working Zone.**

**Le Working Zone (WZ) sono in totale 8, WZ\_NUM sarà quindi espresso su 3 bit (000-111), e l’indirizzo base di ciascuna di esse è situato in memoria RAM rispettivamente nei primi 8 indirizzi. Ogni Working Zone è composta da 4 indirizzi consecutivi da 7 bit e per identificare un indirizzo, si utilizza il suo indirizzo base (WZ\_BASE) assieme all’offset (WZ\_OFFSET), espresso in notazione *one-hot* su 4 bit, permettendo la rappresentazione dei valori da 0 a 3.**

**Risultato finale del processo, sarà la scrittura del risultato della codifica attraverso l’interfaccia d’uscita o\_data:** out std\_logic\_vector (7 downto 0), **codifica Little-endian, nella cella RAM(9), seguendo la politica sotto riportata:**

* **se ADDR appartiene ad una Working Zone,**

**RAM(9) conterrà l’indirizzo così codificato:**

Bit 7 : valore del bit WZ\_BIT=1;

Bit 6-4: valore codificato binario di WZ\_NUM;

Bit 3-0: valore codificato one-hot di WZ\_OFFSET;

* **se ADDR non appartiene ad alcuna Working Zone,**

**RAM(9) conterrà la seguente sequenza di bit:**

Bit 7 : WZ\_BIT=0;

Bit 6-0: valore binario di ADDR;

**1 - Architettura**

In questo capitolo analizzeremo l’Architettura del progetto, iniziando con una presentazione dell’interfaccia del componente utilizzato, per poi descriverne il processo e infine la logica di funzionamento.

* 1. **Interfaccia**

**Le specifiche del progetto vengono interamente svolte da un unico componente,** project\_reti\_logiche, in quanto non si è ritenuto necessario aggiungerne ulteriori a supporto dell’algoritmo, sebbene tale scelta non permetta ulteriore astrazione e modularità dell’Architettura.

Il componente si occupa dunque di leggere i segnali in input e gestire quelli di ouput tramite l’interfaccia riportata in basso:

**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, sono riportati in basso i segnali di IO:

* **i\_clk** riceve dal TestBench il segnale di CLOCK, con periodo impostabile; nello specifico il processo p\_CLK\_GEN si occupa di fornire la temporizzazione.
* **i\_start** è il segnale di START, che quando assume valore 1, indica l’inizio della codifica dell’addr.
* **i\_rst** è il segnale di RESET, che quando assume valore 1, permette di inizializzare i segnali per garantire l’esecuzione corrente e preparare quella successiva.
* **i\_data** contiene il valore del registro espresso in 8 bit proveniente dalla RAM.
* **o\_address** contiene l’indirizzo del registro della RAM dove si intende leggere o scrivere.
* **o\_done** è il segnale di DONE, che quando assume valore 1, indica la fine della sequenza di comandi principale.
* **o\_en** è il segnale di ENABLE, necessario all’attivazione del processo MEM del TestBench, al fine di comunicare alla RAM l’intenzione di accesso alla memoria.
* **o\_we** è il segnale di WRITE ENABLE, inviato al processo MEM del TestBench, al fine di comunicare alla RAM l’intenzione di scrittura in memoria (0 per leggere, 1 per scrivere).
* **o\_data** contiene il valore elaborato espresso in 8 bit da scrivere nella RAM all’indirizzo specificato da o\_address.

In sintesi, i segnali possono essere categorizzati in tre tipologie: la prima (i\_clk) è quella che realizza la temporizzazione, la seconda (i\_rst, i\_start, o\_done) si occupa della gestione del ciclo vita dell’esecuzione del progetto ed infine una terza categoria (o\_en, o\_address, o\_we, i\_data, o\_data) viene impiegata dal processo MEM per permettere il processo di lettura e scrittura dei dati nella RAM.

* 1. **Processo**

**Il processo viene utilizzato per gestire i segnali in entrata e in uscita dal componente, impiegando la discretizzazione temporale del clock (lista di sensibilità), parallelamente alle condizioni che scandiscono gli eventi.**

**Esso si occupa di gestire opportunamente i segnali in entrata dal processo test (i\_rst, i\_start) del Test Bench e produrre in uscita il segnale o\_done in modo da portare a termine correttamente la specifica e preparare una nuova esecuzione.**

**La prima condizione [1] permette al componente di resettare i valori non appena viene inviato il segnale i\_rst, e con l’aggiunta del segnale alla sensitivity list, si rende il processo di RESET istantaneo, asincrono e indipendente dalle altre parti.**

**La seconda condizione [2] permette di temporizzare i segnali in maniera tale che quando i\_clock commuta da 1 a 0, si inneschino le operazioni su di essi, mentre nel restante periodo, vengono effettivamente modificati e messi a disposizione per il ciclo di clock successivo.**

**Questo scandire ed effettuare operazioni diverse ad ogni ciclo, combacia perfettamente con l’idea di una macchina a stati finiti, motivo per il quale si è scelto di utilizzare questa tipologia di struttura per rappresentare la logica del componente e della quale verrà descritto il suo funzionamento nel dettaglio nei prossimi paragrafi.**

**Successivamente se il segnale i\_start viene alzato ad 1, si entra nella condizione [3]**, e l’esecuzione della codifica può finalmente avere inizio.

**Nel codice sottostante si evidenziano le parti principali che caratterizzano il ciclo vita del processo:**

processo: **process**(i\_clk, i\_rst)

**...**

**begin**

**if** i\_rst = '1' **then [1]**

curr\_state **<=** rst;

**else if** (i\_clk'event **and** i\_clk='0') **then [2]**

**case** curr\_state **is**

**when** rst **=>**

**...**

**if** i\_start = '1' **then [3]**

...

curr\_state **<=** read\_input\_addr;

**end if**;

**...**

**when** final\_state =>

**if** i\_start = '0' **then [4]**

o\_done **<=** '0';

curr\_state **<=** rst;

**else [5]**

o\_done **<=** '1';

**end if**;

**end case**;

**end if**;

**end if**;

**end process**;

**La terminazione della codifica e la predisposizione dei segnali ad una nuova è gestita dalle condizioni [5]** e **[4], nelle quali si controlla che i\_start si trovi ad 1, per poter segnalare la terminazione della codifica impostando o\_done ad 1. Successivamente, dal processo test del TestBench viene inviato i\_start = 0 che grazie alla [4]** porta il segnale DONE a 0 per poi tornare allo stato di RESET **nell’attesa di un nuovo segnale di START e quindi di una nuova codifica.**

* 1. **Segnali interni e variabili**

**In questo paragrafo vengono descritti i segnali interni al componente utilizzati a supporto dello sviluppo del codice e dell’architettura in generale, per permettere di memorizzare segnali temporanei e non interferire con i segnali di interfaccia, ma fornire ad essi i risultati solo ad elaborazione completata.**

* **addr segnale nel quale viene memorizzato l’indirizzo da codificare, letto della cella 8 della RAM.**
* **wz\_bit segnale costituito da un solo bit utilizzato nella codifica finale per indicare l’appartenenza (1) o non appartenenza (0) ad una specifica Working Zone.**
* **wz\_base segnale utilizzato per memorizzare l’indirizzo base delle Working Zone durante l’esecuzione.**
* **wz\_num segnale utilizzato per memorizzare il numero della Working Zone specifica nel caso in cui l’addr vi appartenga.**
* **wz\_offset\_binary segnale utilizzato per salvare la distanza in binario di un indirizzo da una Working Zone relativa.**
* **wz\_offset segnale utilizzato per salvare l’offset in notazione *one-hot.***
* **i variabile utilizzata come contatore incrementale per il ciclo della FSM.**
* **curr\_state segnale utilizzato per indicare lo stato corrente della FSM; può valere uno dei 9 stati definiti dal tipo type\_states.**
* **type\_states** tipo contentente tutti gli stati della FSM che curr\_state potrà assumere nel corso dell’esecuzione della specifica; in particolare, la definizione risulta

**type** type\_states **is** (rst, read\_input\_addr, find\_wz\_base, get\_wz\_base, scan\_wz, not\_in\_wz, in\_wz, set\_output, final\_state);

* 1. **Logica e stati della FSM**

**Nel seguente paragrafo viene descritto il funzionamento della macchina a stati finiti che governa la logica del progetto; ogni cambiamento di stato è dato dall’assegnamento di uno specifico stato al segnale curr\_state, così che al ciclo di clock successivo si possa entrare nello stato assegnato al segnale.**

**L’implementazione della FSM in VHDL è stata realizzata con uno *switch-case*, dove ogni *case* corrisponde ad uno stato di type\_states.**

**La codifica comincia dallo stato rst, il cui scopo è quello di inizializzare i segnali interni e di interfaccia, onde evitare valori di partenza inaspettati; nel momento in cui viene inviato il segnale di START la codifica può avere inizio.**

**Si passa quindi allo stato read\_input\_addr: in questo stato si legge all’indirizzo 8 della RAM il valore dell’indirizzo da codificare e lo si salva nel segnale interno addr.**

**Successivamente, si entra nel ciclo principale composto dagli stati find\_wz\_base, get\_wz\_base, scan\_wz, il cui scopo è scorrere tutte le Working Zone arrivando ad un punto in cui si prenderà una delle due diramazioni, dipendentemente dall’appartenenza dell’indirizzo alla WZ, not\_in\_wz oppure in\_wz.**

* **La funzione di find\_wz\_base è quella di impostare in o\_address l’indirizzo della i-esima Working Zone che si vuole andare a controllare.**
* **La funzione di get\_wz\_base, è quella di leggere dalla memoria l’indirizzo base della i-esima WZ e garantire la permanenza all’interno del ciclo [1] a patto che non siano finite le Working Zone da controllare (condizione else), e in quest’ultimo caso si entra nello stato not\_in\_wz. [2].**

**if i < 8 then // i inizializzata a 0 nello stato rst e**

**// incrementata in scan\_wz**

**wz\_base <=** **i\_data (6 downto 0);**

**curr\_state <=** **scan\_wz; [1]**

**else**

**...**

**curr\_state <=** **not\_in\_wz; [2]**

**end if;**

* **Nel caso [1] si procede con l’esecuzione dello stato scan\_wz, nel quale si controlla l’appartenenza di addr alla WZ numero i [3], il cui indirizzo base è stato salvato nel segnale wz\_base nello stato precedente.**

**In caso affermativo si entra nello stato in\_wz, [4] in caso negativo si torna allo stato find\_wz\_base collegamento che identifica l’esistenza di un loop all’interno della FSM, e si incrementa la i per controllare la i+1 – esima WZ.**

**if addr >= wz\_base and**

std\_logic\_vector**(**unsigned**(addr)-**unsigned**(wz\_base)) < "0000100" [3]**

**then**

**...**

**curr\_state <=** **in\_wz; [4]**

**else**

**i := i+1;**

**curr\_state <=** **find\_wz\_base;**

**end if;**

**In altre parole, se la variabile i raggiunge il valore 8 [2] significa che le WZ da scorrere sono finite e la condizione di appartenenza ad una WZ [3] non si è mai verificata.**

**In questo caso si procede quindi con la codifica come da specifica: viene scritto nella cella 9 della RAM, attraverso l’interfaccia di uscita del componente o\_data, la concatenazione di wz\_bit & addr, dove wz\_bit = 0.**

**Nel caso contrario invece viene fornito a o\_data la concatenazione di** wz\_bit = 1, **wz\_num (ottenuto convertendo in binario la variabile i) e wz\_offset.**

**Quest’ultimo segnale viene ottenuto eseguendo lo shift verso sinistra della sequenza 0001 di un numero di volte pari a wz\_offset\_binary, dove wz\_offset\_binary è la distanza espressa su 3 bit dell’addr dalla WZ, ottenuto dalla sottrazione binaria di questi due segnali.**

**Ad esempio, se wz\_offset\_binary fosse 011, si deve effettuare uno shift a sinistra per 3 volte dei bit 0001 ottenendo quindi 1000 (notazione *one-hot* del numero 3).**

Poiché nello stato in\_wz viene effettuata questa operazione, il valore di wz\_offset non sarebbe pronto per la codifica finale prima del ciclo di clock successivo, quindi e si è utilizzato uno stato aggiuntivo set\_output nel quale si concatena wz\_bit, wz\_num e wz\_offset e si assegna la sequenza di bit così ottenuta a o\_data.

Infine, dagli stati not\_in\_wz e set\_ouput si giunge nello stato final\_state dove si termina la codifica inviando il segnale di o\_done impostato ad 1 al TestBench e si lascia la gestione della terminazione della codifica e la predisposizione ad una nuova al processo e a quest’ultimo stato della FSM.

Viene riportato sotto lo schema ad alto livello del funzionamento della FSM.

i\_start

i\_clk

i\_reset

i\_data

rst

read

input

addr

find

wz

base

get

wz

base

scan

wz

not

in

wz

in

wz

set

output

final

state

o\_done

o\_addresss

o\_we

o\_data

⮚ segnali di input ⮘ segnali di output • stati FSM

**2 – Test Bench**

Viene ora descritta la struttura del TestBench, analiazzando alcune delle sue parti fondamentali.

Grazie alla seguente interfaccia si mappa la comunicazione fra il componente e il TestBench,

in modo da far corrispondere i segnali in entrata/uscita e permettere il passaggio di informazioni da un modulo all’altro.

UUT: project\_reti\_logiche

**port** map (

i\_clk **=>** tb\_clk,

i\_start **=>** tb\_start,

i\_rst **=>** tb\_rst,

i\_data **=>** mem\_o\_data,

o\_address **=>** mem\_address,

o\_done **=>** tb\_done,

o\_en **=>** enable\_wire,

o\_we **=>** mem\_we,

o\_data **=>** mem\_i\_data

);

Per scelta progettuale, la RAM viene fornita dal Test Bench e vengono inizializzati i primi 8 indirizzi con gli indirizzi base delle Working Zone:

**type** ram\_type **is** array (65535 **downto** 0)

of std\_logic\_vector (7 **downto** 0);

**signal** RAM: ram\_type := (

0 **=>** std\_logic\_vector(**to\_unsigned**(4 ,8)),

1 **=>** std\_logic\_vector(**to\_unsigned**(13,8)),

8 **=>** std\_logic\_vector(**to\_unsigned**(ADDR,8)),

**others =>** (**others =>**'0'))

La lettura e la scrittura su di essa avviene grazie al seguente processo:

MEM : **process**(tb\_clk)

**begin**

**if** tb\_clk'event **and** tb\_clk = '1' **then**

**if** enable\_wire = '1' **then**

**if** mem\_we = '1' **then**

RAM(conv\_integer(mem\_address) **<=** mem\_i\_data;

mem\_o\_data **<=** mem\_i\_data **after** 1 ns;

**else**

mem\_o\_data **<=** RAM(conv\_integer(mem\_address))

after 1 ns;

**end** **if**;

**end** **if**;

**end** **if**;

**end** **process**;

Esso si attiva nel momento in cui il segnale di clock assume valore alto; se l’operazione da effettuare è una scrittura il segnale enable\_wire dovrà essere impostato a 1, mentre se è una lettura a 0.

Tutti i processi sono sincronizzati con il segnale di clock che nel Test Bench è gestito dal processo p\_CLK\_GEN riportato in basso, dove c\_CLOCK\_PERIOD è impostato a 100ns.

p\_CLK\_GEN : **process is**

**begin**

**wait for** c\_CLOCK\_PERIOD/2;

tb\_clk <= not tb\_clk;

**end process** p\_CLK\_GEN;

Infine, vi è un ultimo processo che si occupa di inviare i segnali di RST, START e aspettare fino a quando il segnale di DONE non viene portato a 1, per poi poter avviare un’altra esecuzione con nuovi valori.

test: **process**

**begin**

**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;

**wait until** tb\_done = '1';

**wait for** c\_CLOCK\_PERIOD;

tb\_start **<=** '0';

**wait until** tb\_done = '0';

Da notare come questo processo potrebbe essere modificato in base al tipo di esecuzione che si vuole ottenere, e in ogni caso il modulo dovrebbe processare i valori in maniera opportuna. **Ad esempio, si potrebbe volere eseguire due codifiche all’interno della stessa esecuzione, quindi occorerebbe modificare il processo in maniera tale da fornire i segnali opportunamente.**

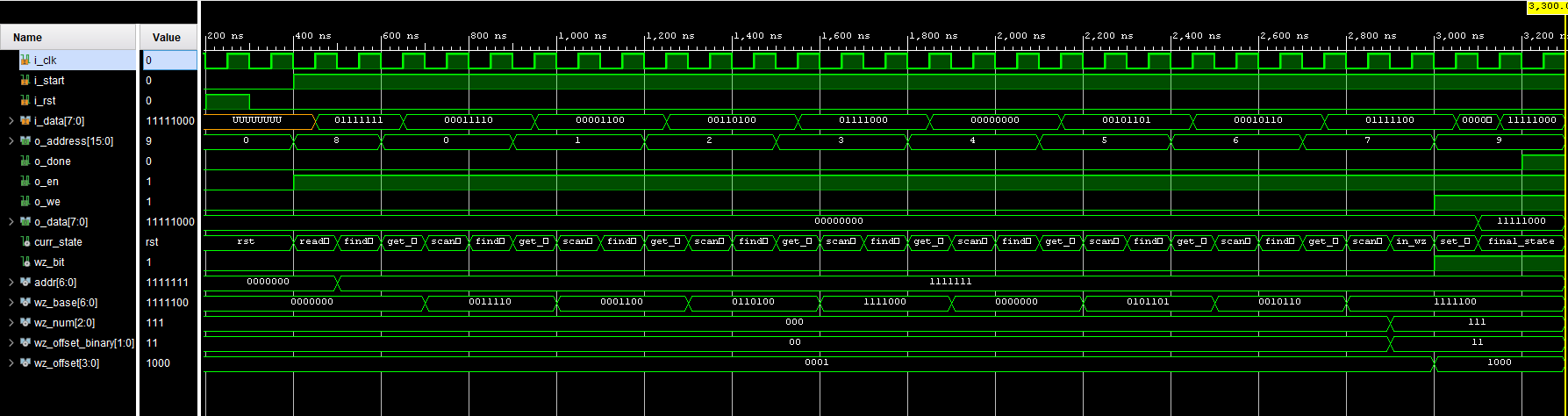
**In generale, quindi il processo test si occupa di impartire il ciclo-vita che il modulo dovrà seguire nel momento in cui viene fatta partire la simulazione.**

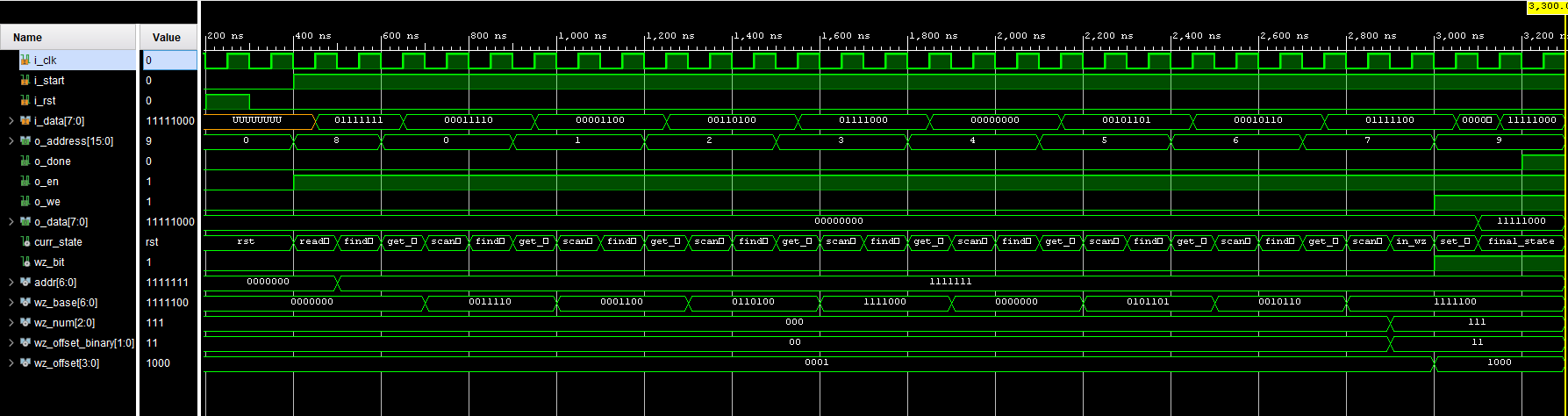
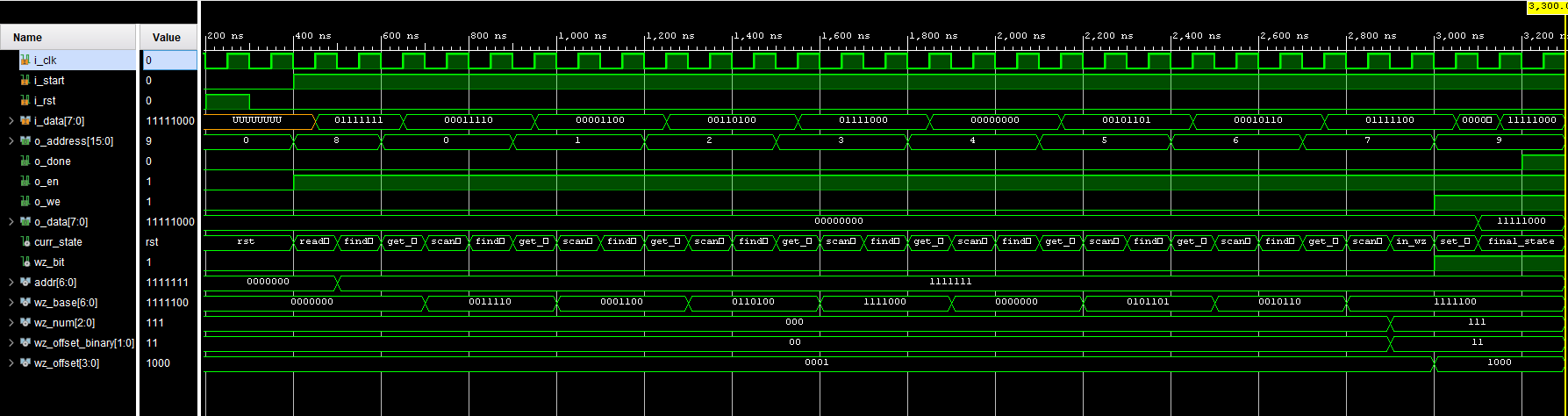
**3 – Risultati Sperimentali**

In questo capitolo verranno commentati i risultati ottenuti dalle esecuzioni delle simulazioni *Behavioural* e *Post-Functional Synthesis*.

**3.1 TestBench – Caso limite massimo**

Obiettivo di questo TestBench è quello di fornire i valori in ingresso in modo da avere il caso massimo da codificare: l’indirizzo da cercare è il massimo previsto su 7 bit e si trova nell’ultima Working Zone avente indirizzo base 124, che è il massimo previsto, poiché gli indirizzi della WZ in questione partono da 124 ed arrivano fino a 127.





Vengono rappresentati di seguito primi 9 indirizzi della RAM, i segnali interni e di interfaccia che entrano in gioco nel corso dell’esecuzione; in particolare, nella tabella si leggono gli indirizzi base delle WZ (0-7), l’addr (8) e infine il valore codificato (9).

A fianco della tabella si evidenziano i valori dei segnali interni nel final\_state.

|  |  |
| --- | --- |
| **RAM** | **value** |
| **0** | **30** |
| **1** | **12** |
| **2** | **52** |
| **3** | **120** |
| **4** | **0** |
| **5** | **45** |
| **6** | **22** |
| **7** | **124** |
| **8** | **127** |
| **9** | **248** |

**addr: 1111111**

**wz\_bit: 1**

**wz\_base: 1111100**

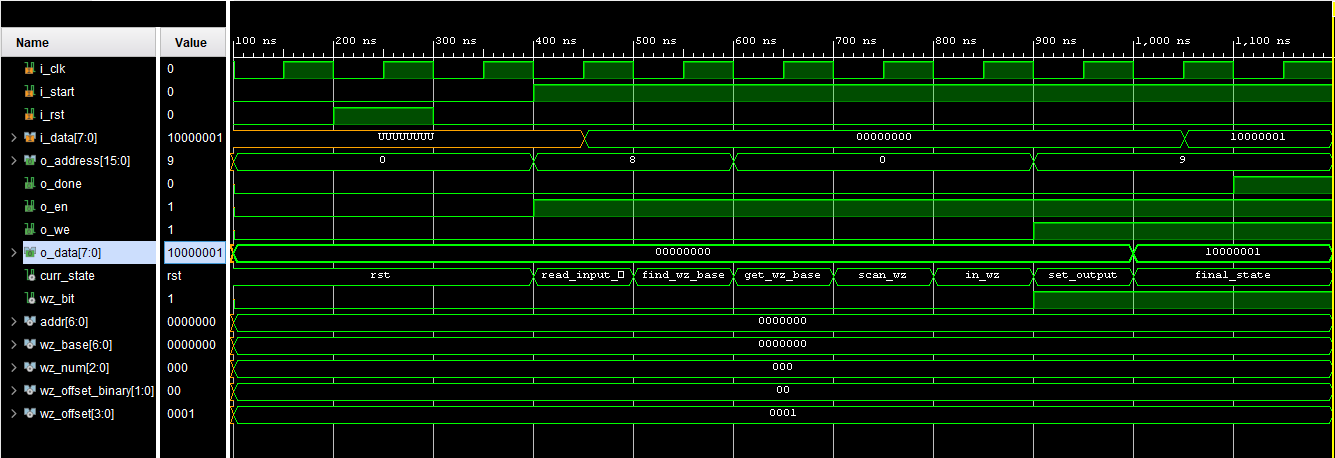
**wz\_num: 111**

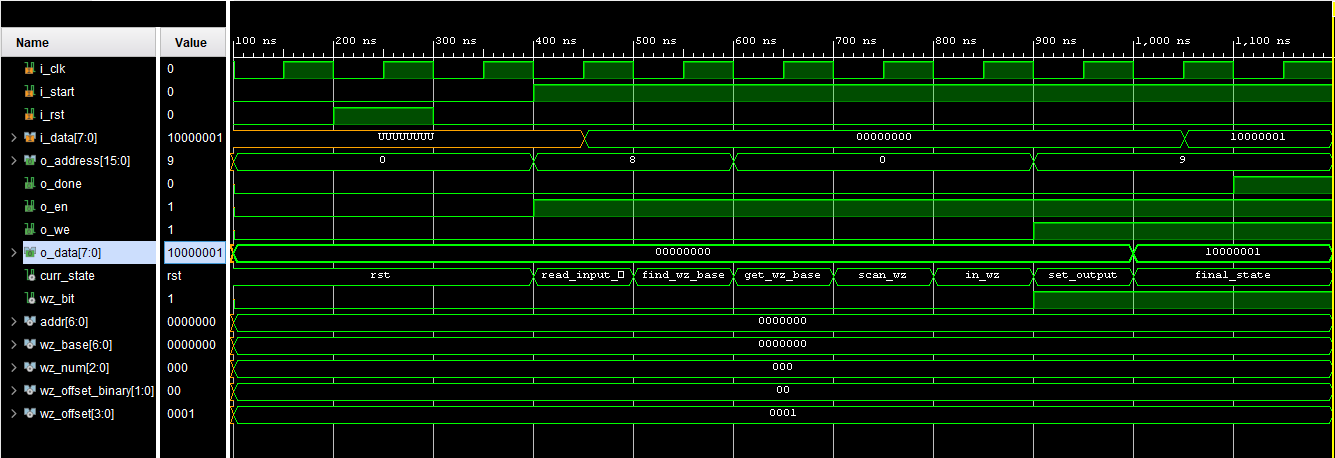
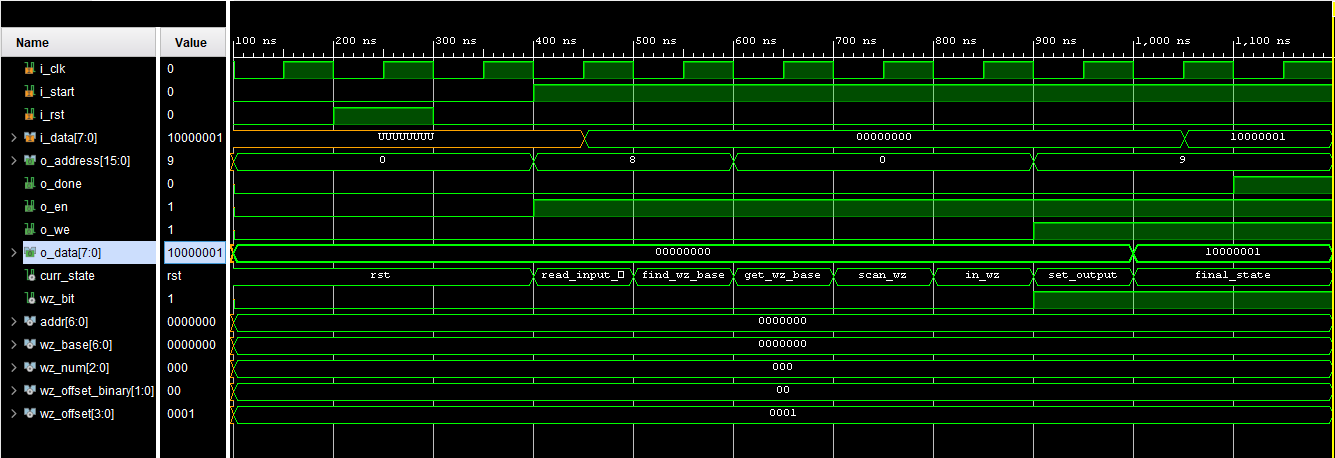
**wz\_offset\_binary: 11**

**wz\_offset: 1000**

**codifica finale: 1 111 1000**

**3.2 – TestBench – Caso limite minimo**

Obiettivo di questo TestBench è quello di fornire in ingresso l’indirizzo da codificare appartenente al primo indirizzo della Working Zone 0 (caso limite minimo).



|  |  |
| --- | --- |
| **RAM** | **value** |
| **0** | **0** |
| **1** | **64** |
| **2** | **70** |
| **3** | **10** |
| **4** | **15** |
| **5** | **99** |
| **6** | **4** |
| **7** | **122** |
| **8** | **0** |
| **9** | **129** |

**addr: 0000000**

**wz\_bit: 1**

**wz\_base: 0000000**

**wz\_num: 000**

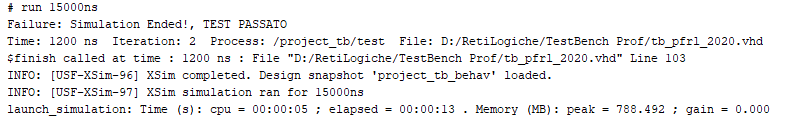
**wz\_offset\_binary: 00**

**wz\_offset: 0001**

**codifica finale: 1 000 0001**

**Viene qui di seguito riportato il log di esecuzione della *Behavioural Simulation* con esito positivo**



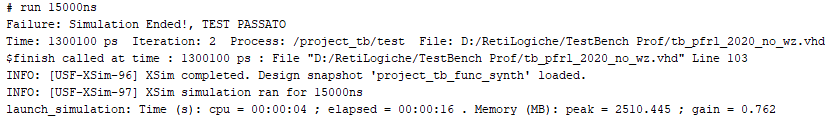


L’esecuzione non porta a nessun tipo di errore o warning, e viene eseguita nel tempo descritto nell’ultima riga del log.

**3.3 – Sintesi**

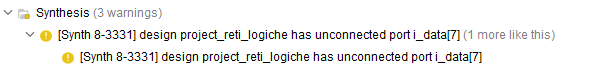
Il componente supera correttamente le simulazioni *Post-Synthesis Functional* insieme alla *Post-Synthesis Timing*.Di seguito vengono riportati i log della prima:





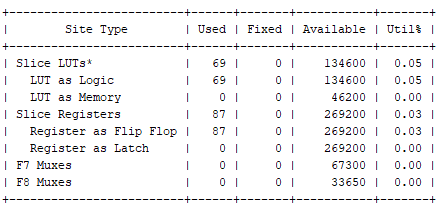
Nel report di sintesi non sono presenti né *errors*, né *critical warnings*. Gli unici *warnings* che vengono riportati riguardano il segnale i\_data, del quale si utilizzano solo 7 bit su 8, in quanto i segnali addr e wz\_base non richiedono bit ulteriori per essere rappresentati; per verifica empirica, non risulta comunque essere un rilevante problema per l’esecuzione della specifica.

****

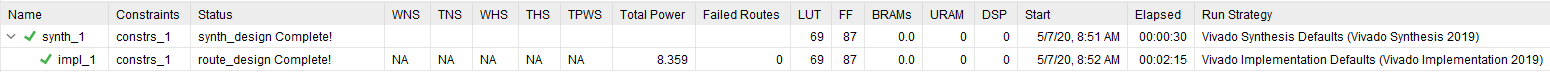


Infatti, al fine di eliminare questi *warning,* si è provato ad utilizzare 8 bit per addr ed altrettanti per wz\_base, e, nonostante il bit aggiuntivo non fosse strettamente necessario, si è riscontrato aumento non significativo dei LUT e dei FF, quindi si è preferito evitare di implementare la modifica.

Viene di sotto riportato il synth\_report\_utilization generato da *Vivado*, dal quale si ricava il numero di componenti elementari usati per la sintesi.

****

Come si può notare dalla tabella, viene utilizzata una parte molto piccola dell’FPGA, solo il 0.05% di LUT ed il 0.03% di FF; i risultati citati si considerano accettabili e non si è ritenuto necessario ridurre il numero FF e LUT; questa scelta avrebbe portato a rendere il codice meno semplice e lineare, con la perdita della divisione netta degli stati, che invece caratterizza il progetto in questione.

****

**Conclusioni**

Con questo progetto si è voluta dare una possibile soluzione alla specifica richiesta, concentrandosi sull’utilizzo di un’architettura compatta, composta da un singolo processo e allo stesso tempo da una logica estesa su più stati, in modo da separare le operazioni sintatticamente e semanticamente, offrendo una visione più semplice, e allo stesso tempo funzionale del componente. In definitiva, il componente passa correttamente i test *Behavioural Simulation*, *Post-Synthesis Functional Simulation* con i risultati descritti nei paragrafi precedenti, mantendosi essenziale nella struttura.