

# Prova Finale di Reti Logiche

Politecnico di Milano

Lorenzo Gadolini,  
Giuseppe Lischio

March 22, 2020

## Contents

<b>Contents</b>	<b>I</b>
<b>1 Introduzione</b>	<b>1</b>
1.1 Dati Progettuali e Specifica . . . . .	1
1.2 La fase di traduzione . . . . .	1
1.2.1 Entity del componente . . . . .	2
<b>2 Architettura e Scelte Progettuali</b>	<b>2</b>
2.1 L'algoritmo . . . . .	2
2.2 Non Appartenenza . . . . .	2
2.3 Appartenenza . . . . .	3
2.4 Il design: La macchina a stati . . . . .	3
2.4.1 Descrizione degli stati . . . . .	4
2.5 Il Codice VHDL . . . . .	4
<b>3 Risultati della Sintesi</b>	<b>6</b>
<b>4 Simulazioni</b>	<b>6</b>
4.1 Test Benches . . . . .	6
4.2 Screenshots . . . . .	7
<b>5 Conclusioni</b>	<b>9</b>

# 1 Introduzione

Il metodo di codifica a working-zone propone un'ottimizzazione orientata alla riduzione del consumo di energia introdotto dall'input/output di un microprocessore.

Sia dato un generico sistema composto da un processore e una memoria esterna al chip referenziabile tramite un bus indirizzi. Questo metodo suggerisce che un programma in esecuzione sul dato sistema sfrutti durante la sua esecuzione un insieme di indirizzi "preferiti", e che quindi sia più efficiente racchiudere tutti questi indirizzi dentro uno spazio di lavoro (detto appunto Working-Zone).

Gli indirizzi di queste Working-Zones sono codificati tramite un sistema base e offset, così da ridurre ulteriormente la quantità di informazione trasmessa sul bus indirizzi, e di conseguenza ridurre anche l'energia dissipata.

## 1.1 Dati Progettuali e Specifica

Il componente da progettare ha come compito quello di stabilire se un indirizzo che riceve in input appartiene o meno ad una delle Working-Zones stabilite, e in caso affermativo di effettuare la traduzione dell'indirizzo da binario naturale a codifica WZE.

Il componente si interfaccia con una memoria indirizzabile al byte a partire dall'indirizzo 0, e in cui vengono inizializzati i dati necessari per effettuare la computazione. La memoria inoltre è indirizzabile tramite indirizzi di 16 bit.

Gli indirizzi di memoria da 0 a 7 contengono le basi delle 8 Working-Zones stabilite in fase di setup e scritte in binario naturale senza segno su 7 bit.

La cella di memoria 8 contiene l'indirizzo su cui effettuare la verifica e l'eventuale codifica, anch'esso scritto in binario naturale senza segno su 7 bit.

L'indirizzo di memoria 9 conterrà a fine computazione un numero di 8 bit senza segno, che come verrà descritto più avanti potrà essere una codifica o un indirizzo semplice.

I restanti indirizzi sono tutti inizializzati a 0 e non vengono sfruttati durante la computazione.

Gli indirizzi delle Working-Zones sono inoltre supposti non ripetuti, ed inoltre è supposto che non ci sia overlapping tra gli spazi delle varie Working-Zones. Per cui ogni indirizzo analizzato appartiene al più ad una Working-Zone.

## 1.2 La fase di traduzione

Nella fase di codifica il componente deve essere in grado di stabilire se un indirizzo depositato nella cella numero 8 della memoria appartiene o meno ad una Working-Zone.

Un indirizzo che non appartiene a nessuna Working-Zone fa entrare il componente in uno stato di "Non appartenenza", stato in cui la macchina secondo specifica deve depositare in uscita il dato così come le è stato fornito, preceduto da un bit posto a "0".

Il dato in uscita è un intero binario naturale senza segno di 8 bit

"0" & "0101010" --Codifica<sup>1</sup> esempio di indirizzo che non cade in alcuna WZ

Se il componente stabilisce che l'indirizzo appartiene ad una Working-Zone, scatta la fase di "Appartenenza", il dato necessita di encoding.

Un indirizzo codificato è un numero di 8 bit senza segno composto da tre sottogruppi di bit.

- Il bit più significativo viene posto ad "1", che indica "Codifica avvenuta";
- I tre bit successivi codificano posizionalmente la Working-Zone a cui tale indirizzo appartiene. Il dato in uscita è un numero binario naturale senza segno da 0 a 7 che indica che l'indirizzo appartiene alla "n-esima" Working-Zone.

---

<sup>1</sup>Il simbolo & indica concatenazione di bit

- I quattro bit finali codificano l'offset dell'indirizzo dalla base. Il numero codificato è un numero binario One Hot. L'offset indica la distanza posizionale in memoria dall'indirizzo base, ovvero l'indirizzo "si trova a" n indirizzi dalla base.

"1" & " 010" & " 0100" --Codifica esempio di indirizzo appartenente ad una WZ

Sia che l'indirizzo appartenga o che non appartenga ad una Working-Zone, il componente prosegue sempre lungo la stessa fase di output. Il dato viene depositato in memoria, salvato, e la macchina entra in fase di attesa di reset e avvio per una successiva codifica.

### 1.2.1 Entity del componente

Il componente hardware è stato progettato tramite linguaggio VHDL in base alle specifiche fornite dal tema di progetto.

La struttura input/output del componente in VHDL è stata fornita nelle specifiche ed è di seguito riportata.

```

--- ENTITY DEL PROGETTO ---
entity project_reti_logiche is
  Port ( i_clk : in STD_LOGIC;           --Input clock proveniente dal testbench.
        i_start : in STD_LOGIC;         --Segnale di avvio della computazione generato dal testbench.
        i_rst : in STD_LOGIC;           --Segnale di reset generato dal testbench.
        i_data : in STD_LOGIC_VECTOR (7 downto 0); --Input Byte proveniente dalla memoria esterna.
        o_address : out STD_LOGIC_VECTOR (15 downto 0); --Indirizzo di memoria per cui si richiede lettura/scrittura.
        o_done : out STD_LOGIC;          --Segnale di terminazione della computazione generato dal componente.
        o_en : out STD_LOGIC;            --Segnale di ENABLE per poter comunicare con la memoria.
        o_we : out STD_LOGIC;            --Segnale di abilitazione alla scrittura in memoria.
        o_data : out STD_LOGIC_VECTOR (7 downto 0)); --Dato da scrivere in memoria.
end project_reti_logiche;

```

## 2 Architettura e Scelte Progettuali

### 2.1 L'algoritmo

Il progetto si basa su un algoritmo alquanto semplice.

Dato uno stato iniziale di memoria, per prima cosa viene letto il dato di cui si necessita la codifica dall'indirizzo di memoria numero 8, successivamente il componente entra in un ciclo di calcolo. In questo ciclo di calcolo vengono letti uno alla volta i byte di memoria 0-7 contenenti gli indirizzi base delle Working zones.

Il componente sottrae l'indirizzo da codificare all'indirizzo base letto, e verifica che il risultato cada fra 0 e 3. Questo risultato ha il significato di offset, e come da specifica, un offset compreso fra 0 e 3 (inclusi) indica un'appartenenza alla working zone. Qualsiasi altro risultato viene interpretato come "Non appartenenza".

### 2.2 Non Appartenenza

Un risultato che non appartiene a nessuna working zone deve venire propagato in uscita così come è stato letto da memoria in entrata. Il componente ha terminato gli otto cicli di verifica e ha determinato che l'indirizzo in input non appartiene a nessuna working zone. A questo punto la macchina carica i 7 bit dell'indirizzo nel registro di output, il quale era stato inizializzato con 8 bit tutti posti a zero, così da rispettare la specifica che prevede di emettere in uscita uno 0 seguito dai 7 bit dell'indirizzo scritti in binario naturale senza segno.

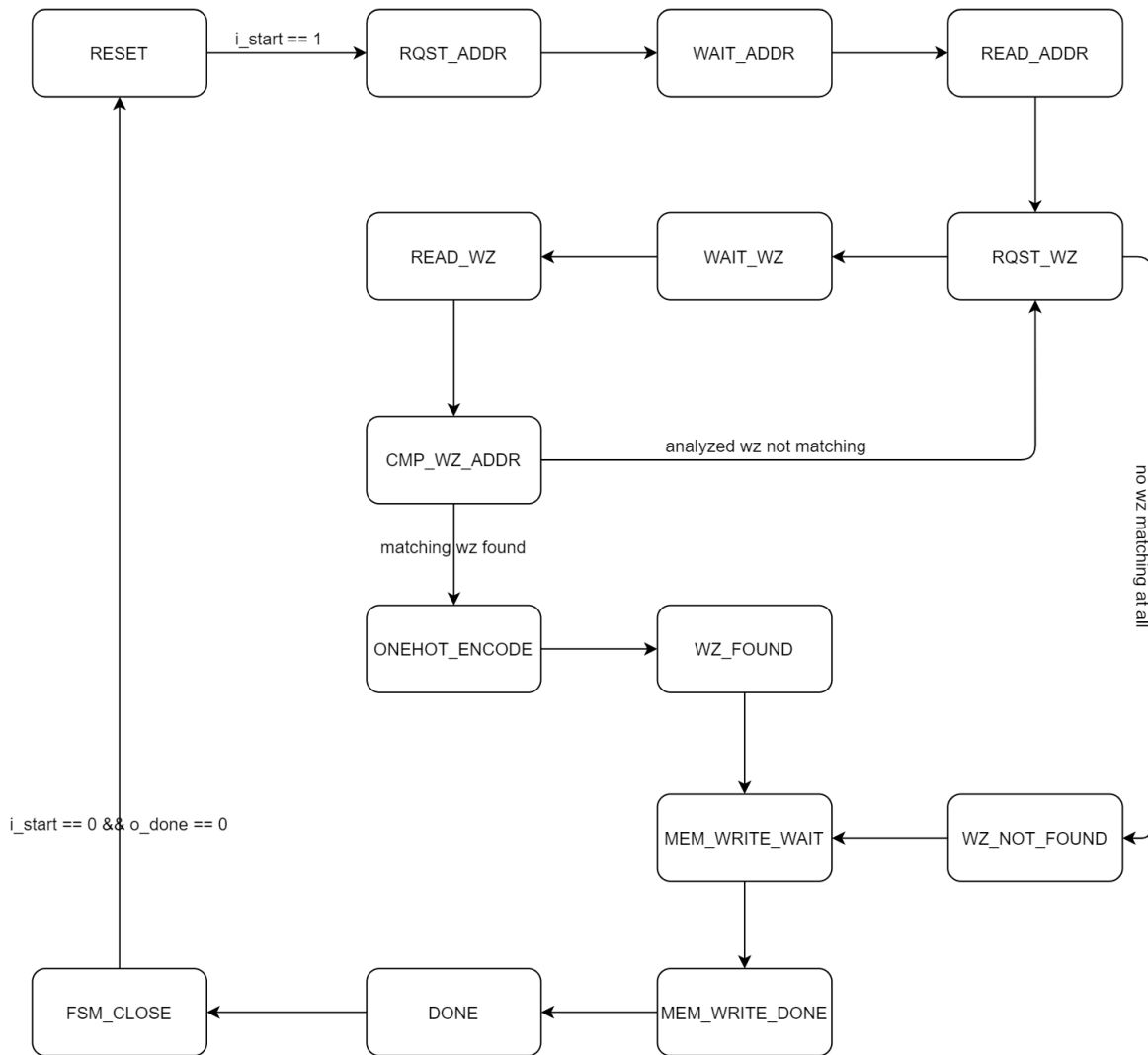
## 2.3 Appartenenza

Se il componente durante la sua esecuzione verifica che l'indirizzo da codificare cade all'interno di una delle basi, allora si avvia la procedura di encoding. Per prima cosa viene codificato l'offset. Il risultato della sottrazione appartiene ad un insieme finito e ristretto di valori, per questo si è scelto di creare una lookup table che contenesse le codifiche in onehot da assegnare al valore di uscita a seconda del valore di offset. Una volta scritto in memoria il primo quartetto di bit, il componente passa alla traduzione della tripletta di bit contenente l'indirizzo della working zone. Questa fase scrive i tre bit nel registro di out che corrispondono al valore binario naturale del byte di memoria che contiene l'indirizzo base. Infine il componente pone ad 1 il MSB del valore di uscita, così che chi riceve il valore tradotto a valle riconosce che è stato effettuato l'encoding.

## 2.4 Il design: La macchina a stati

Il progetto del componente parte dalla descrizione di una macchina a stati che esegua l'algoritmo poco fa descritto

La macchina a stati completa che esegue la codifica è disegnata qui:



Nel paragrafo 2.4.1 verranno descritti i compiti dei singoli stati e delle transizioni.

### 2.4.1 Descrizione degli stati

- **RESET:** Lo stato in cui la macchina viene avviata e a cui giunge una volta che riceve il segnale di reset. La macchina cicla su questo stato attendendo un segnale di start. Successivamente inizializza la memoria e salta al primo vero stato di esecuzione.
- **RQST\_ ADDR:** La macchina richiede alla memoria l'indirizzo da codificare.
- **WAIT\_ ADDR:** La macchina attende per un ciclo di clock la propagazione dei segnali alla memoria.
- **READ\_ ADDR:** La macchina ottiene il dato dalla memoria.
- **RQST\_ WZ:** La macchina richiede alla memoria il valore della n-esima base, dove n è compreso fra 0 e 7.
- **WAIT\_ WZ:** La macchina attende un ciclo di clock la propagazione dei segnali alla memoria.
- **READ\_ WZ:** La macchina ottiene la base n-esima dalla memoria.
- **CMP\_ WZ\_ ADDR:** La macchina esegue la sottrazione che determina l'appartenenza o meno e l'offset del dato da codificare.
- **ONEHOT\_ ENCODE:** La macchina codifica l'offset di un indirizzo che ha dato esito positivo in una comparazione.
- **WZ\_ FOUND:** La macchina prepara il byte contenente il dato codificato e notifica alla memoria che è pronto per essere scritto
- **WZ\_ NOT\_ FOUND:** La macchina notifica alla memoria che il dato non appartiene a nessuna Working Zone, e che è pronta a propagarlo senza codifica.
- **MEM\_ WRITE \_ WAIT:** La macchina attende un clock che la memoria riceva la sua richiesta di scrittura
- **MEM\_ WRITE\_ DONE:** La macchina completa la fase di scrittura.
- **DONE:** La macchina notifica il raggiungimento dello stato di terminazione.
- **FSM\_ CLOSE:** La macchina entra in attesa di una richiesta di reset.

Nel grafico della macchina è stato escluso l'autoanello dello stato di reset. L'autoanello è necessario perchè permette alla macchina di rimanere in attesa su tale stato che il segnale di start si alzi, dando il via ad una nuova computazione.

Sono stati inoltre esclusi tutti gli archi che da ogni singolo stato portano allo stato di reset. `i_rst` potrebbe venire posto alto (ad "1") in qualsiasi momento della computazione.

Data questa evenienza è necessario che la macchina in qualsiasi stato si trovi abbandoni l'encoding ed entri nello stato di reset.

Questo si ottiene con una gestione asincrona della verifica del segnale. Ad ogni ciclo di clock viene analizzato il segnale, e se posto ad "1" la macchina esegue una transizione allo stato in cui si trova a RESET.

## 2.5 Il Codice VHDL

Fino a questo momento non è stata data nessuna descrizione fisica del componente al di fuori delle specifiche che descrivono il problema. In questo piccolo paragrafo viene fatta un'analisi di come è stata descritta la macchina a stati in linguaggio VHDL, analizzandone i segnali e i registri necessari al funzionamento. Il componente è stato descritto tramite codice VHDL Behavioral che implementa tutti gli stati descritti nel paragrafo 2.4.1.

Oltre all'architettura fornita dalla specifica, il componente possiede altri segnali e registri che si sono resi necessari in fase di scrittura di codice.

```
-- BEHAVIORAL ARCHITECTURE --
architecture Behavioral of project_reti_logiche is
    type state_type is (
        RESET,
        RQST_ADDR,
        WAIT_ADDR,
        READ_ADDR,
        RQST_WZ,
        WAIT_WZ,
        READ_WZ,
        CMP_WZ_ADDR,
        ONEHOT_ENCODE,
        WZ_FOUND,
        WZ_NOT_FOUND,
        MEM_WRITE_WAIT,
        MEM_WRITE_DONE,
        DONE,
        FSM_CLOSE);
    signal state : state_type; --Segnale di gestione del flusso degli stati.

begin
    process (i_clk, i_rst)
        -- Variabili
        variable addressToEncode: std_logic_vector(7 downto 0); --Byte in cui salvare l'indirizzo da codificare.
        variable wzBase: std_logic_vector(7 downto 0); --Indirizzo base di una Working_Zone.
        variable wzCounter: integer range -1 to 8; --Contatore necessario a ciclare le 8 basi.
        variable wzOffset: integer; --Variabile che contiene l'offset rispetto ad una base.
        variable baseInteger: integer; --Variabile ausiliaria.
        variable addressInteger: integer; --Variabile ausiliaria.
        variable encodedOutput: std_logic_vector (7 downto 0); --Contiene l'indirizzo codificato rispetto ad una WZ.
        variable onehotOffset: std_logic_vector (3 downto 0); --Contiene la codifica onehot dell'offset.
    end process
end architecture;
```

- addressToEncode

Variabile di tipo std\_logic\_vector a 8 bit, viene utilizzata per salvare l'indirizzo che necessita di valutazione e codifica.

- wzBase

Variabile std\_logic\_vector a 8 bit che salva nel ciclo di computazione la base della Working-Zone e viene aggiornata ad ogni risultato di non appartenenza con la base successiva.

- wzCounter

Variabile integer che può assumere valori da -1 (default) ad 8. Viene sfruttata come flag per segnalare a quale valore di byte in memoria siamo arrivati. Raggiunto il valore 8 funge da flag per segnalare al componente che non ci sono più Working-Zones da valutare e che la macchina deve entrare in stato di "Non Appartenenza" (par. 2.2).

- wzOffset

Variabile integer che salva il valore dell'offset di un indirizzo rispetto alla sua base in caso la macchina determini che tale indirizzo appartiene ad una Working-Zone.

- encodedOutput

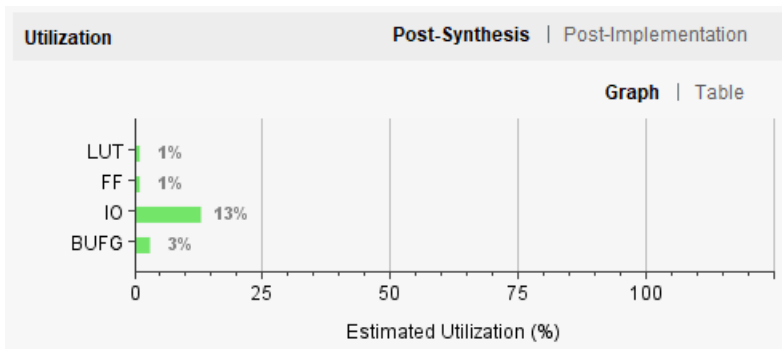
Variabile di tipo std\_logic\_vector a 8 bit che salva un indirizzo che è stato sottoposto a processo di encoding ( par. 2.3).

- encodedOutput

Variabile di tipo std\_logic\_vector a 4 bit che viene sfruttata dalla macchina per salvare il valore di offset in codifica onehot.

### 3 Risultati della Sintesi

Il componente progettato è stato sintetizzato in maniera virtuale tramite l'apposito tool di Vivado. Come FPGA target è stata utilizzata una Xilinx xc7a200tfbg484-1. La sintetizzazione ha dato un risultato positivo, ed il componente è stato successivamente testato anche post sintesi, con gli stessi test effettuati per il solo funzionamento logico. È emerso che il componente funziona correttamente anche impostando periodi di clock molto inferiori al requisito minimo della specifica (100 ns). Il report post sintesi di Vivado ha inoltre rilevato le seguenti percentuali di utilizzo delle risorse della FPGA utilizzata:



### 4 Simulazioni

In questo paragrafo verranno discussi i metodi utilizzati per verificare il corretto funzionamento del componente progettato. In particolare è stato usato il software Xilinx "Vivado" per scrivere i test bench e per effettuare vari tipi di simulazione:

- Simulazione **Behavioral**: test del funzionamento logico del componente, basato sul codice VHDL scritto, senza passare per la sintesi virtuale del componente.
- Simulazione **Post-Synthesis Functional**: test che permette di verificare il comportamento strutturale del componente descritto. Viene eseguito dopo aver terminato il processo di sintesi virtuale
- Simulazione **Post-Synthesis Timing**: test analogo al *Post-Synthesis Functional*, che tiene però conto di tutti i ritardi introdotti dalle porte logiche usate nella simulazione del componente virtuale.

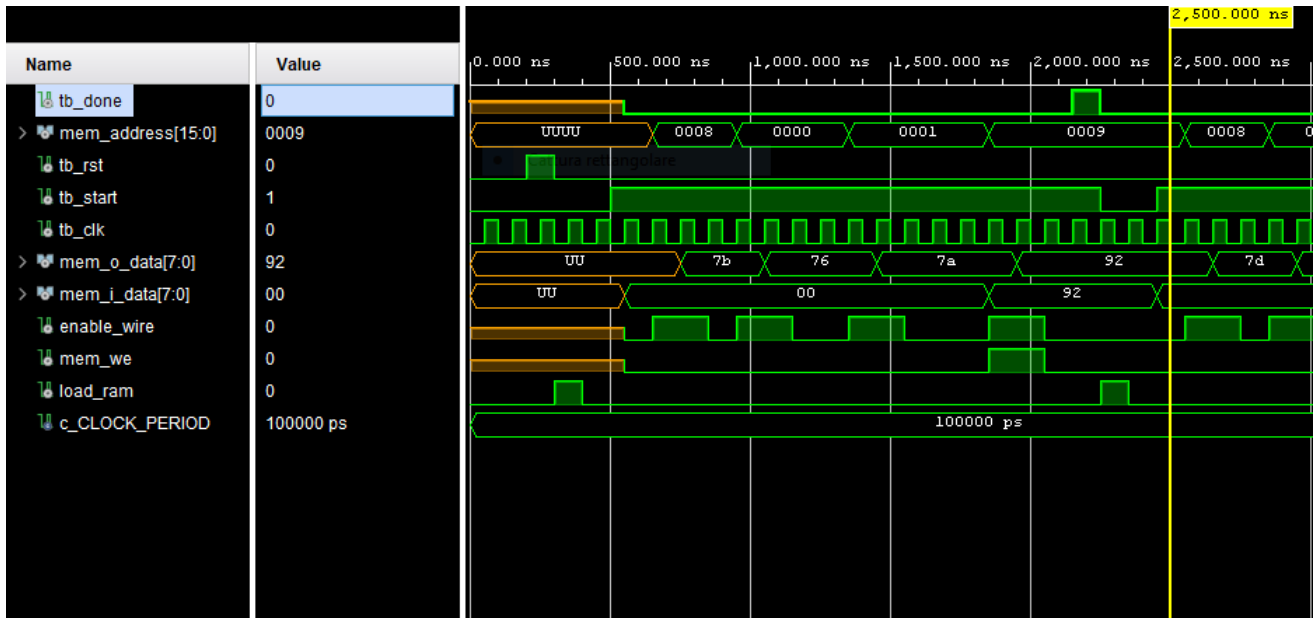
#### 4.1 Test Benches

Le tipologie di simulazione descritte sopra, sono state applicate su casi di test scritti ad hoc, per individuare eventuali punti deboli del funzionamento del componente. I test a cui la macchina è stata sottoposta sono:

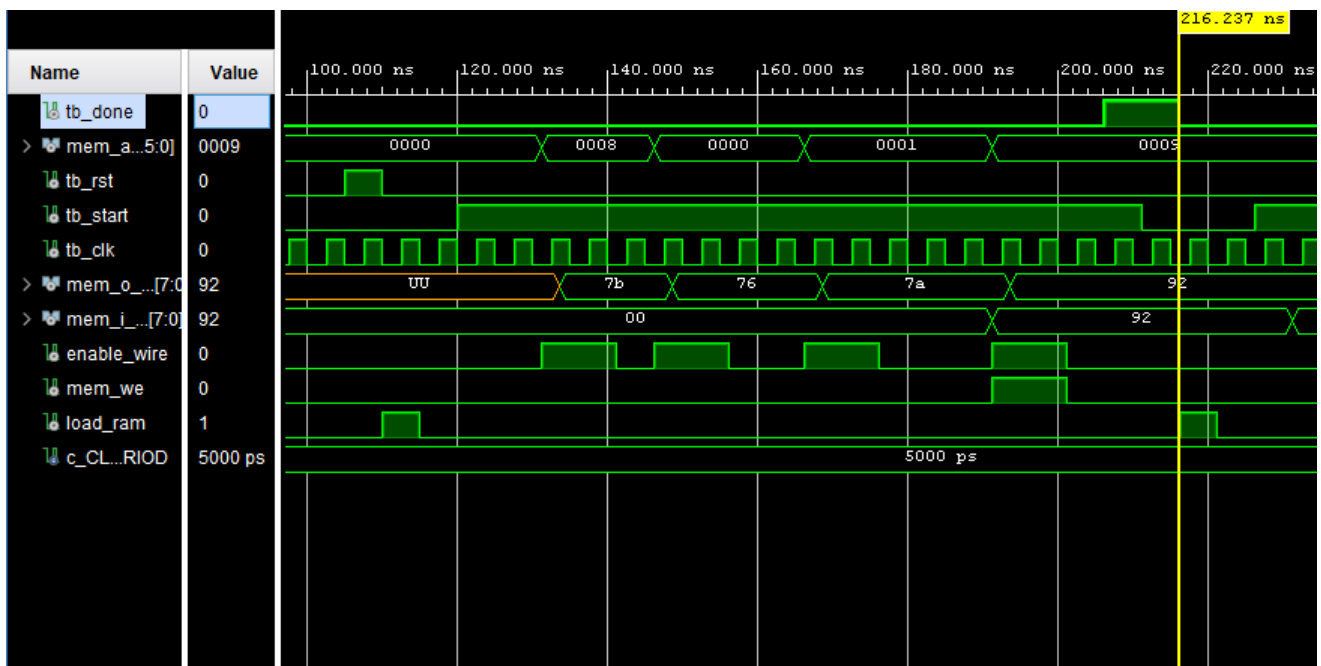
- Test Bench singolo: test analogo a quello fornito dal personale docente, che sottopone alla macchina un singolo caso di test, per poi terminare.
- Test Bench sequenziale: vengono sottoposti alla macchina dei casi di test da effettuare in maniera sequenziale. Per effettuare questo tipo di test, si è pensato di aggiungere al test bench singolo, una gestione dei valori in input tramite un file di testo, contenente un caso di test per ogni riga. La macchina, dopo aver terminato la computazione ed aver riportato il segnale *tb\_done* al livello basso, riceve un segnale creato ad hoc, chiamato *load\_ram*, grazie al quale viene fatta partire la lettura di un nuovo caso di test da input, i cui valori vengono copiati in memoria, e con i quali viene eseguita una nuova computazione. In questo modo si è potuto verificare che la macchina non ha problemi con il reset dei segnali e delle celle di memoria. Inoltre è stato possibile testare la macchina su decine di migliaia di casi di test diversi, generati automaticamente tramite degli appositi script.
- Test Bench reset asincrono: viene testata la capacità della macchina di gestire un reset asincrono durante l'esecuzione di casi di test sequenziali

## 4.2 Screenshots

**Simulazione Behavioral sequenziale** Durante l'esecuzione di più casi di test consecutivi, si può notare come a seguito della memorizzazione del risultato computato, e dopo l'abbassamento - in questo caso contemporaneo - dei segnali *tb\_done* e *tb\_start*, il segnale *load\_ram* venga alzato per consentire la lettura dei nuovi dati e poter cominciare un nuovo ciclo di computazione.



**Simulazione Post-Synthesis Timing 5ns** La macchina riesce ad effettuare la computazione anche con clock a 5ns. Si può notare come tra i fronti di discesa di *tb\_done* e *tb\_start* ci sia un piccolo ritardo di quasi un periodo di clock, dovuto alla rete combinatoria utilizzata per la sintesi virtuale del componente.





**Codice Test Bench Sequenziale** Qui di seguito si trovano alcuni snippet del codice del test bench sequenziale, opportunamente commentato.

```
22 --variabili e segnali per gestione file di testo
23 file testValuesFile : text open read_mode is "test13032020.txt";
24 shared variable row : line; --linea del file di testo che viene letto
25 shared variable dataIn : integer;
26 shared variable expectedResult : integer; --risultato atteso dopo la codifica dell'indirizzo
27 shared variable testCount : integer; --contatore dei casi di test che sono stati svolti
28 signal load_ram : std_logic; --segnale per avviare il caricamento di nuovi dati di test da file
29 type ram_type is array (65535 downto 0) of std_logic_vector(7 downto 0);

70 MEM : process(tb_clk, load_ram)
71 begin
72     --quando il segnale load_ram sale, viene letta una nuova riga nel file
73     --contenente i casi di test da effettuare in sequenza
74     if load_ram'event and load_ram = '1' then
75         readline(testValuesFile, row); --Lettura riga del file: ogni riga contiene un caso di test
76         for i in 0 to 8 loop --loop per inserire in ram gli indirizzi delle wz
77             read(row, dataIn); --Lettura di un singolo dato dalla riga
78             RAM(i) <= std_logic_vector(to_unsigned(dataIn, 8));
79         end loop;
80         read(row, expectedResult); --Lettura del valore atteso dopo la codifica
81         RAM(9) <= "00000000"; --inizializzazione dell'indirizzo in cui memorizzare il valore codificato
82     end if;

97 test : process is
98 begin
99     testCount := 0;
100    load_ram <= '0';
101    wait for 100 ns;
102    wait for c_CLOCK_PERIOD;
103    tb_rst <= '1'; --inizialmente si porta la macchina nello stato di reset
104    wait for c_CLOCK_PERIOD;
105    tb_rst <= '0';
106
107    --Loop fino a che non si arriva alla fine del file di testo
108    --contenente i casi di test sequenziali
109    while (not endfile(testValuesFile)) loop
110        testCount := testCount + 1;
111        load_ram <= '1'; --alzando il segnale vengono caricati i nuovi valori da testare
112        wait for c_CLOCK_PERIOD;
113        load_ram <= '0';
114        wait for c_CLOCK_PERIOD;
115        tb_start <= '1'; --segnale di 'start' alto per iniziare una nuova computazione
116        wait for c_CLOCK_PERIOD;
117        wait until tb_done = '1'; --attesa finchè la macchina non pone 'done' ad 1
118        wait for c_CLOCK_PERIOD;
119        tb_start <= '0';
120        wait until tb_done = '0'; --dopo aver abbassato 'start', si attende la discesa di 'done'
```

```

122      -- Assertions
123  v    assert RAM(9) = std_logic_vector(to_unsigned(expectedResult, 8)) report "TEST #" &
124  v    integer'image(testCount) & " FALLITO. Expected " & integer'image(expectedResult) &
125    " but found: " & integer'image(to_integer(unsigned(RAM(9)))) severity failure;
126    report integer'image(testCount) & "test fatto" & integer'image(expectedResult);
127  end loop;
128
129  v    assert false report "Simulation Ended!, SUPERATI TUTTI I " &
130    integer'image(testCount) & " TEST" severity failure;
131  end process test;
132
133  end projecttb;

```

## 5 Conclusioni

La macchina è stata quindi progettata attendendosi alla specifica e ne è stato testato il funzionamento. L'algoritmo adottato è stato pensato per effettuare la computazione riducendo quanto possibile gli accessi. Si è notato che è in grado di lavorare anche a periodi di clock vicini a 1ns.