POLITECNICO
MILANO 1863

Prova Finale del Corso di Reti Logiche

Politecnico di Milano

Lorenzo Gadolini Giuseppe Lischio.

Contents

1 Introduzione

Il metodo di codifica a Working-Zones 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 supposti non ripetuti, ed inoltre si suppone 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[1] esempio di indirizzo che non cade in alcuna WZ
```

[1] Il simbolo & indica concatenazione di bit

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.
- 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/o avvio per una successiva codifica.

1.2.1 Entity del componente

Il componente hardware è stato descritto 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.

```vhdl
--- 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.

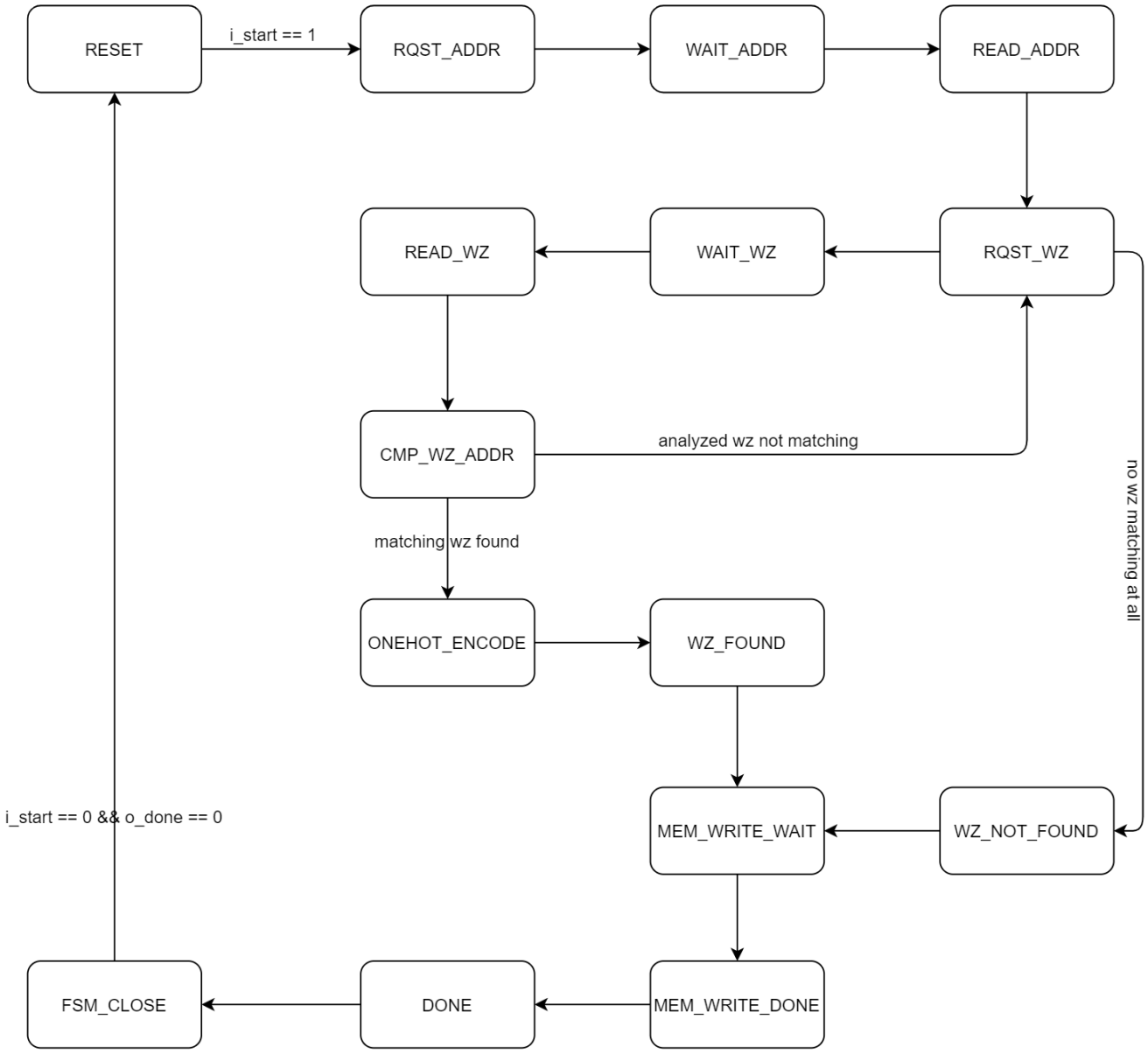
2.3 Appartenenza

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 tali valori di offset già tradotti in codifica OneHot. La macchina si occupa di assegnare al valore di uscita uno dei valori di offset tradotti. Una volta scritto in memoria il primo quartetto di bit, il componente passa alla traduzione della tripletta di bit contenente l'indirizzo della Working-Zones. Questa fase scrive i tre bit che corrispondono al valore binario naturale del byte di memoria che contiene l'indirizzo base nel registro di out.
Infine il componente pone ad 1 il MSB del valore di uscita, cosi 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 sopra enunciato.

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



Nel paragrafo 2.4.1 verranno specificati 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 o che termina una computazione. 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 ciclo di 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 portando il segnale di completamento ad '1'.
- FSM_ CLOSE: La macchina attende che il segnale di avvio torni a '0' per poter portare a '0' il segnale di completamento e successivamente passare allo stato di reset per avviare un'eventuale computazione successiva.

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 avvio torni ad '1', iniziando una nuova computazione.

Nel grafico per lo stesso motivo è stato escluso l'autoanello sullo stato FSM_ CLOSE. La macchina in tale stato attende ciclando sull'autoanello che il segnale di avvio vada a '0'.
Sono stati inoltre esclusi tutti gli archi che da ogni singolo stato portano allo stato di reset. `i_rst` potrebbe venire posto 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 tale segnale, e se riscontrato ad '1' la macchina esegue una transizione dallo stato in cui si trova a RESET.

2.5 Il Codice VHDL

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 elencati 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.
```

- `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).

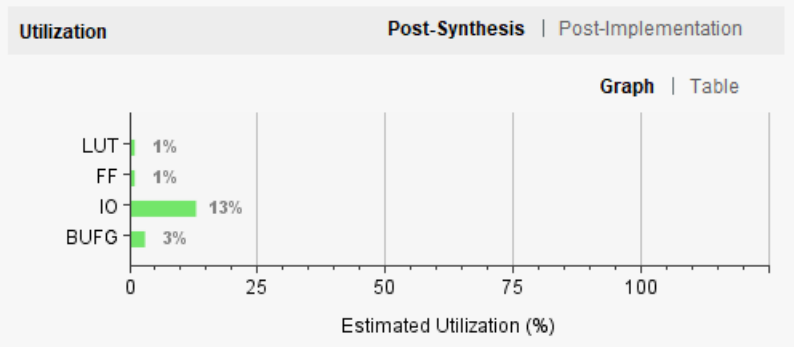
- `encodedOutput`

  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 testato con successo sia con test Behavioral che Post-Synthesis, 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, 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.

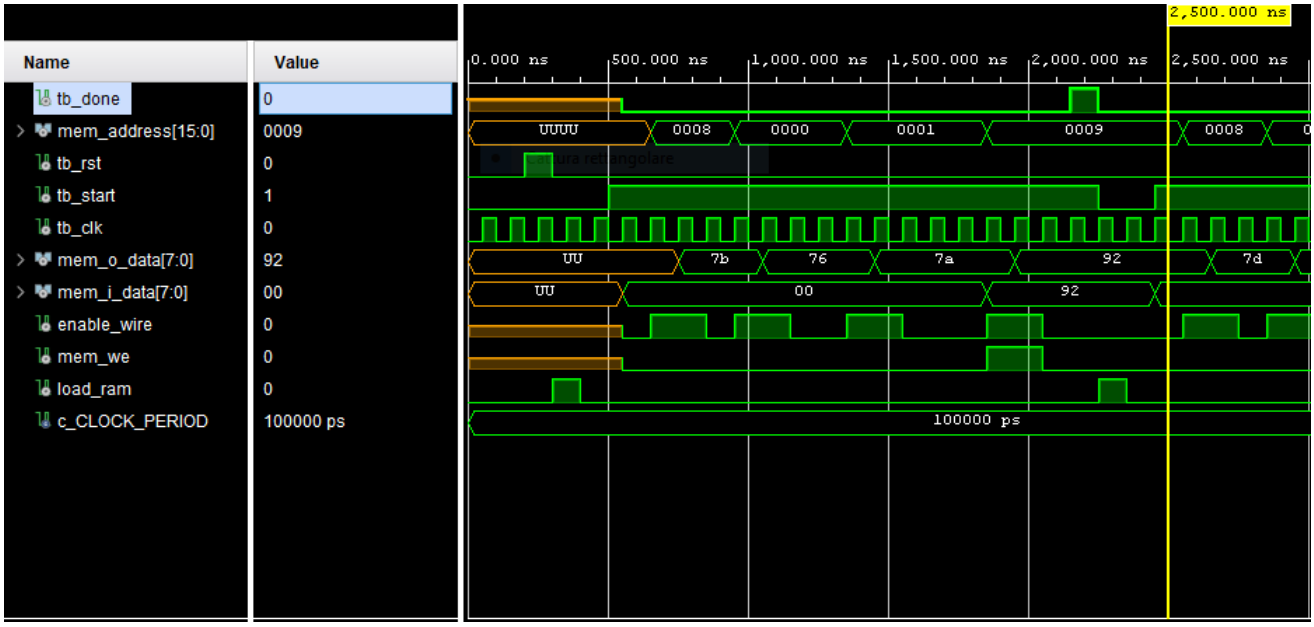
4.1 Test Benches

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:

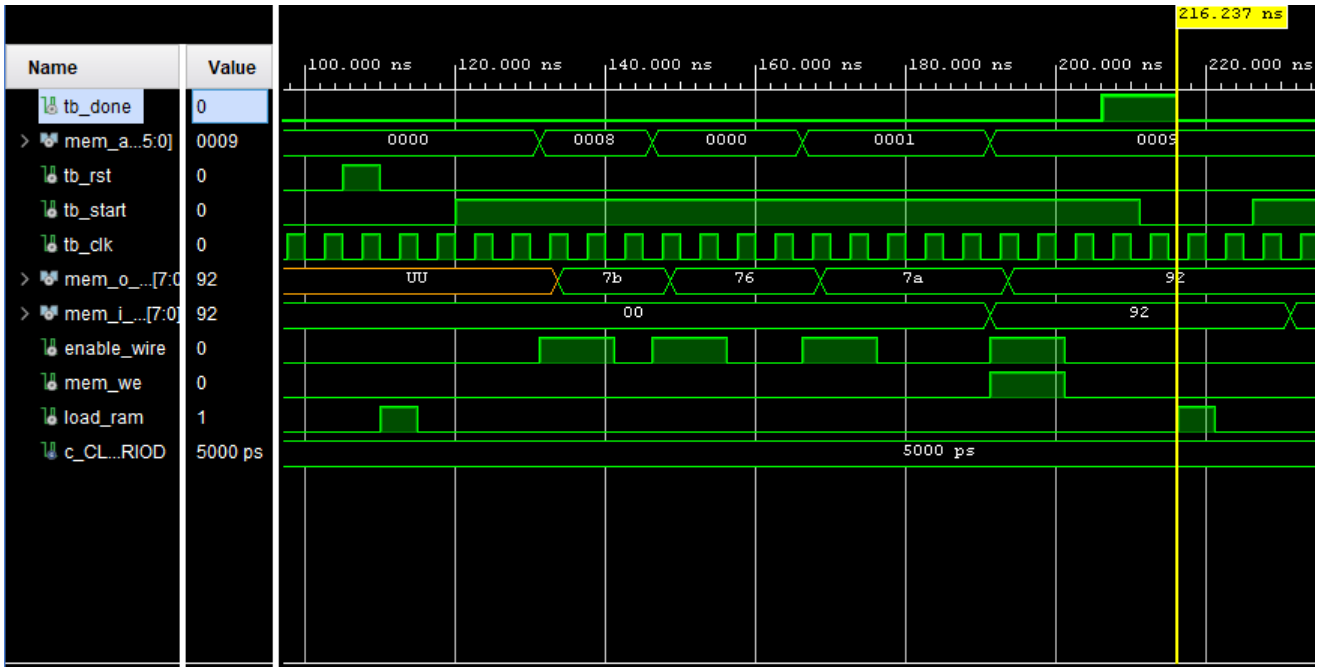
- Test Bench singolo: test analogo a quello fornito dal personale docente, che sottopone alla macchina un singolo caso di test, per poi terminare.
- Test Bench sequenziale: vengono sottoposti alla macchina dei casi di test da effettuare in maniera sequenziale. Per effettuare questo tipo di test, si è pensato di aggiungere al test bench singolo una gestione dei valori in input tramite un file di testo, contente un caso di test per ogni riga. La macchina, dopo aver terminato la computazione ed aver riportato il segnale *tb_done* a '0', riceve un segnale creato ad hoc, chiamato *load_ram*, grazie al quale viene fatta partire la lettura di un nuovo caso di test dal file, 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. In particolare durante l'esecuzione di un caso di test, e dopo un intervallo di tempo prefissato, viene portato ad '1' il segnale di reset senza attendere che la macchina abbia terminato la sua fase di calcolo.

4.2 Screenshots

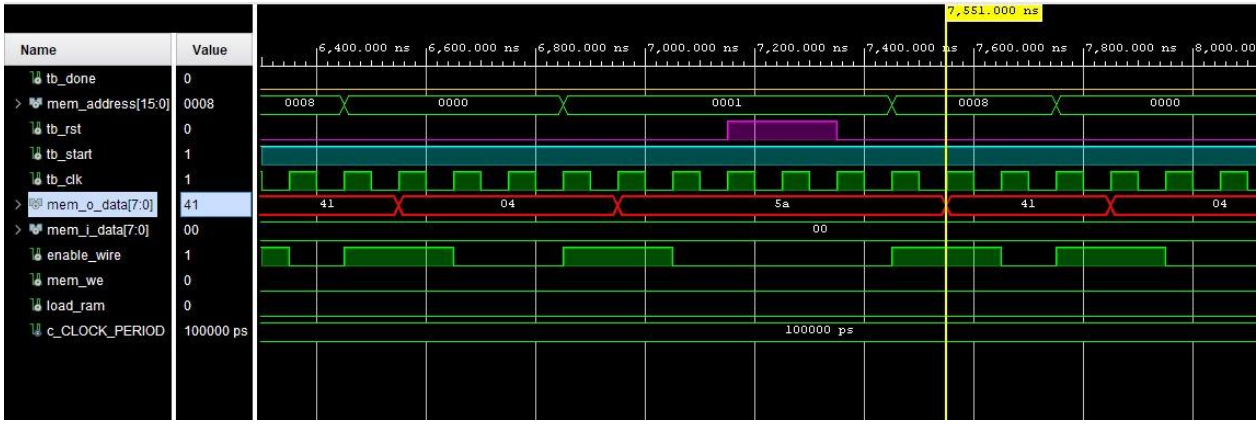
Simulazione Behavioral sequenziale Durante l'esecuzione di più casi di test consecutivi, si può notare come a seguito della memorizzazione del risultato computato, e dopo l'abbassamento - in questo caso contemporaneo - dei segnali *tb_done* e *tb_start*, il segnale *load_ram* venga portato ad '1' per 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 cicli di clock di durata 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.



Simulazione Behavioral, Reset Asincrono L'innalzamento del segnale di reset durante l'esecuzione di una computazione viene gestito correttamente dalla macchina. Nel dettaglio, dopo aver ricevuto tale segnale , la macchina è in grado di riavviare la computazione interrotta e giungere al termine del caso di test. Ciò accade perchè il segnale di reset ripristina la macchina ma non la memoria del testbench, per cui i dati letti restano identici all'esecuzione precedentemente interrotta.



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;
```

```
test : process is
begin
    testCount := 0;
    load_ram <= '0';
    wait for 100 ns;
    wait for c_CLOCK_PERIOD;
    tb_rst <= '1'; --inizialmente si porta la macchina nello stato di reset
    wait for c_CLOCK_PERIOD;
    tb_rst <= '0';

    --loop fino a che non si arriva alla fine del file di testo
    --contenente i casi di test sequenziali
    while (not endfile(testValuesFile)) loop
      testCount := testCount + 1;
      load_ram <= '1'; --alzando il segnale vengono caricati i nuovi valori da testare
      wait for c_CLOCK_PERIOD;
      load_ram <= '0';
      wait for c_CLOCK_PERIOD;
      tb_start <= '1'; --segnale di 'start' alto per iniziare una nuova computazione
      wait for c_CLOCK_PERIOD;
      wait until tb_done = '1'; --attesa finchè la macchina non pone 'done' ad 1
      wait for c_CLOCK_PERIOD;
      tb_start <= '0';
      wait until tb_done = '0'; --dopo aver abbassato 'start', si attende la discesa di 'done'
      -- Assertions
      assert RAM(9) = std_logic_vector(to_unsigned(expectedResult, 8)) report "TEST #" &
       integer'image(testCount) & " FALLITO. Expected " & integer'image(expectedResult) &
        " but found: " & integer'image(to_integer(unsigned(RAM(9))))  severity failure;
      report integer'image(testCount) & "test fatto" & integer'image(expectedResult);
    end loop;

    assert false report "Simulation Ended!, SUPERATI TUTTI I " &
     integer'image(testCount) & " TEST" severity failure;
end process test;

end projecttb;
```

5 Conclusioni

L'algoritmo presentato come soluzione al problema ha complessità di calcolo al più lineare col numero di indirizzi che si presentano in ingresso. Questa affermazione si può dedurre dal fatto che l'automa ha un numero finito di stati (da cui il nome "FSA", finite state automata) che vengono percorsi un numero finito di volte anche se ciclicamente.

Il componente descritto è stato simulato in differenti situazioni di test che vanno dal caso semplice ad interruzioni impreviste della computazione, e anche se sottoposto a questi edge case è stato in grado comunque di portare a termine il suo compito.

Nel paragrafo 4.2 si analizza un caso di test in cui si sottopone una sintesi di componente a periodi di clock 20 volte più brevi rispetto alla specifica. Durante le fasi di test Post Synthesis Timing si è cercato un limite inferiore per la durata di un periodo di clock che non inficiasse l'esecuzione di una codifica, e questo limite si è scoperto aggirarsi attorno ai 3950 *ps*.