Prova finale di Reti Logiche

Piermarco Gerli
(cod_persona: 10729408, matricola: 955980)

Gabriele Gessaghi
(cod_persona: 10660853, matricola: 939491)

30 Marzo 2023

POLITECNICO
MILANO 1863

Indice

1 Introduzione

La specifica della Prova Finale (Progetto di Reti Logiche) per l'Anno Accademico 2022/2023 richiede l'implementazione di un modulo hardware descritto in VHDL che possa interfacciarsi con una memoria e soddisfare le specifiche indicate. In generale, il sistema riceve informazioni sulla locazione di memoria il cui contenuto deve essere indirizzato verso uno dei quattro canali di uscita disponibili. Il canale di uscita e l'indirizzo di memoria vengono forniti tramite un ingresso seriale a un bit, mentre le uscite del sistema forniscono tutti i bit del dato ottenuto dalla memoria in parallelo. La sequenza di input è valida quando il segnale `START` è alto (=1) e termina quando il segnale `START` è basso (=0). Il segnale DONE (output del componente) indica che l'operazione è stata completata con successo e che il valore ottenuto dalla memoria è disponibile sull'uscita corrispondente al canale indicato.

Un possibile esempio di applicazione di questo componente potrebbe essere in un sistema di ricezione di segnale, che deve indirizzare i pacchetti in arrivo su uno dei quattro canali di uscita disponibili in base all'indirizzo di destinazione del singolo pacchetto e, solo una volta caricato completamente il dato sull'uscita, lo rende disponibile all'invio.

2 Architettura

2.1 Early Designs

Nella fase di stesura iniziale del progetto ci siamo concentrati sul cercare una soluzione implementabile che permettesse di risolvere due principali problemi: acquisire correttamente l'indirizzo (fornito tramite un numero di bit variabile tra 0 e 16) ed estenderlo con la codifica zero-extended.

Una prima soluzione a cui abbiamo pensato è stata quella di realizzare un circuito sequenziale che tenesse traccia del numero di bit trasmessi come "indirizzo" (quindi tutti quei bit successivi al secondo per i quali il segnale `start` fosse ancora alto) e che successivamente andasse ad aggiungere tanti zeri quanti ne mancassero per arrivare alla lunghezza prestabilita di 16 bit. Il numero di "bit" acquisiti veniva utilizzato anche come segnale per pilotare un multiplexer per la selezione della casella corrispondente di un array a 16 bit: l'idea era quindi di realizzare 16 "registri" di 1 bit ciascuno, identificati da un numero progressivo tra 0000_{bin} e 1111_{bin}.

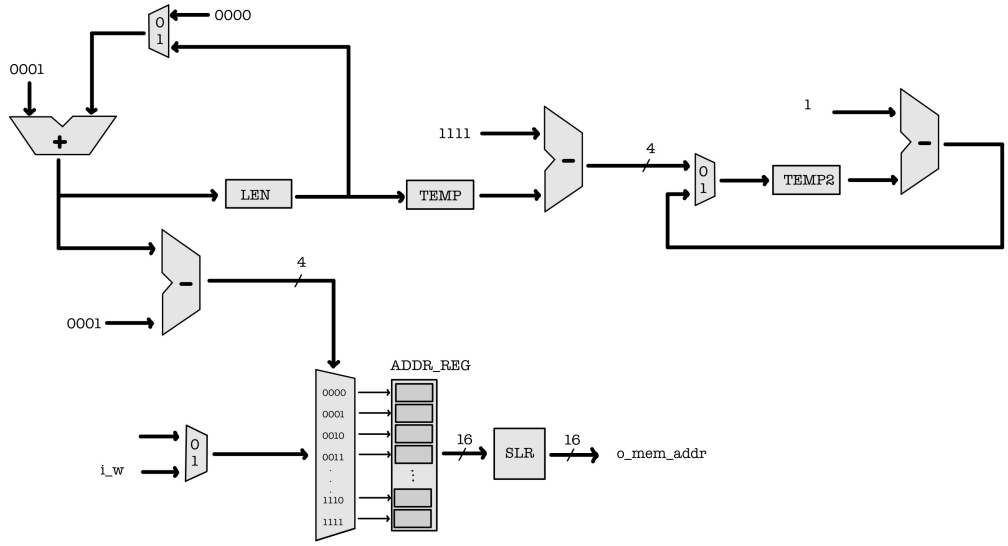

Figure 1: Design iniziale del processo di acquisizione da input seriale

L'architettura però non eseguiva in modo corretto l'estensione degli zeri e i due contatori erano ridondanti: il primo infatti si occupava solamente di conteggiare quanti bit di indirizzo venivano letti e il secondo dava tanti input per fare lo shift logico quanti zeri mancavano per arrivare alla lunghezza dei 16 bit. La soluzione trovata per evitare di dover estendere manualmente gli zeri è stata di inizializzare il vettore indirizzo a 0, eliminando contemporaneamente la necessità di conteggiare il numero di bit letti.

2.2 Modulo di acquisizione

La prima parte del processo di questo componente riguarda l'ottenere due indicazioni fondamentali a partire da un numero di bit trasmessi al componente stesso: 2 bit di indirizzamento e 16 di indirizzo di memoria. L'idea proposta per l'acquisizione dei bit di canale era di utilizzare un moltiplicatore esplicito (come si vede dalla Figura 2), ma è stata abbandonata in favore di un'implementazione dello shift register di tipo SIPO (serial input - parallel output), che consente il caricamento seriale (quindi bit a bit) dell'indirizzo del canale e la propagazione parallela del risultato al termine dell'acquisizione stessa, al fine di minimizzare il numero di cicli di clock richiesti dal processo.

Il processo di acquisizione dell'indirizzo (nonostante la maggiore lunghezza dell'indirizzo) è stato realizzato allo stesso modo, andando a popolare il vettore di tipo `STD_LOGIC_VECTOR` di lunghezza 16 bit un bit alla volta. Il dato parziale dell'indirizzo viene reso disponibile ad ogni iterazione successiva, ma

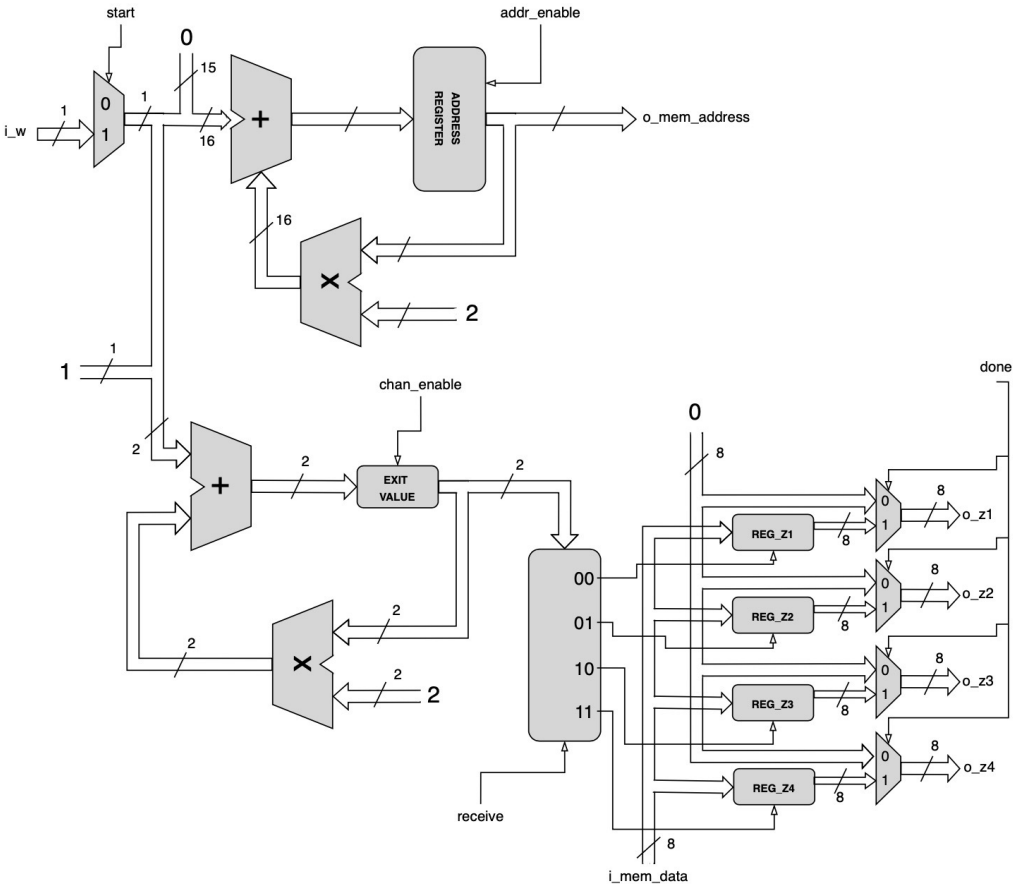

Figure 2: Design adottato per la creazione del componente

viene considerato valido solo all'abilitazione del segnale `o_mem_enable`, comandato dalla FSM.

Algorithm 1 Esempio del codice utilizzato per lo shift register

```
1 PROCESS (sensitivityList)
2 BEGIN
3     IF resetConditionsApplies THEN
4         VectorAddress <= (others => '0');
5     ELSIF rising_edge(clk) and addr_en = '1' THEN
6         VectorAddress <= VectorAddress(14 downto 0)
7                          & inputBit;
8     END IF;
9 END PROCESS;
```

2.3 Modulo di indirizzamento

Il modulo di indirizzamento comprende due processi: il primo abilita il segnale corrispondende all'uscita identificata dai bit di indirizzamento precedentemente acquisiti e il secondo che si occupa di indirizzare effettivamente i dati in uscita sul canale corretto.

La codifica utilizzata è "00" per il canale `o_z0`, "01" per il canale `o_z1`, "10" per il canale `o_z2`, "11" per il canale `o_z3`. Una volta ottenuta la codifica, quando il segnale `receive` comandato dalla macchina a stati finiti diventa alto, il corretto segnale di enable del canale viene posto a "1", a questo punto il canale selezionato viene attivato e i dati ricevuti dalla memoria salvati nel registro opportuno.

Le uscite del componente diventeranno visibili terminato il caricamento dei dati, tramite un ulteriore segnale chiamato `temp_done` (implementazione interna del segnale `o_done`). Quando questo segnale non è attivo (ovvero finchè non è terminato il processo di acquisizione e caricamento dei dati) le uscite del componente restano al valore di default "00000000".

2.4 Reset del componente

Durante lo sviluppo del componente ci siamo accorti che oltre al reset "imposto" dall'esterno, gestito localmente dai processi che portano le variabili e i vettori allo stato inziale, era necessario implementare anche un reset ciclico al termine dell'elaborazione dell'indirizzo corrente: una volta esposte le uscite o_z_n bisogna azzerare il vettore `sum_address` e `channel_selector`.

2.5 Macchina a Stati Finiti

La macchina a Stati Finiti in questo progetto ha la funzione di aggiornare i segnali di controllo a seconda dello stato in cui si trova: manipolando i segnali è possibile abilitare o disabilitare i diversi processi del circuito e permettere quindi l'avanzamento del processo. Di seguito l'elenco degli stati utilizzati e una loro descrizione:

- Stato S0: stato iniziale della FSM nel quale si attende che il segnale `i_start` assuma valore '1'. Questo stato funge anche come stato di `RESET`;
- Stato S1: stato di accettazione del primo bit del canale di uscita (in particolare del bit più significativo) tramite il segnale `chan_en = 1`;
- Stato S2: stato di accettazione del secondo bit del canale di uscita (bit meno significativo), il segnale di `chan_en` resta alto;
- Stato S3: terminata l'accettazione del canale di uscita viene salvato l'indirizzo e tramite il segnale `addr_en` viene abilitato il processo dello shift register responsabile dell'acquisizione dell'indirizzo. La FSM resta in questo stato fino a che il segnale `i_start` rimane a '1', ovvero finché ci sono ancora bit di indirizzo che vengono trasmessi al mio componente;
- Stato S4: stato che si occupa di trasmettere in output l'indirizzo acquisito;
- Stato S5: stato di accettazione dei dati restituiti dalla memoria;
- Stato S6: stato che si occupa di trasmettere in output i valori salvati nei registri di uscita ponendo il segnale `temp_done=1`. Questo stato inoltre pone a '1' anche il segnale di `internal_rst` che permette di resettare lo shift register dell'indirizzo e del canale di uscita, permettendo la successiva acquisizione di nuovi valori;

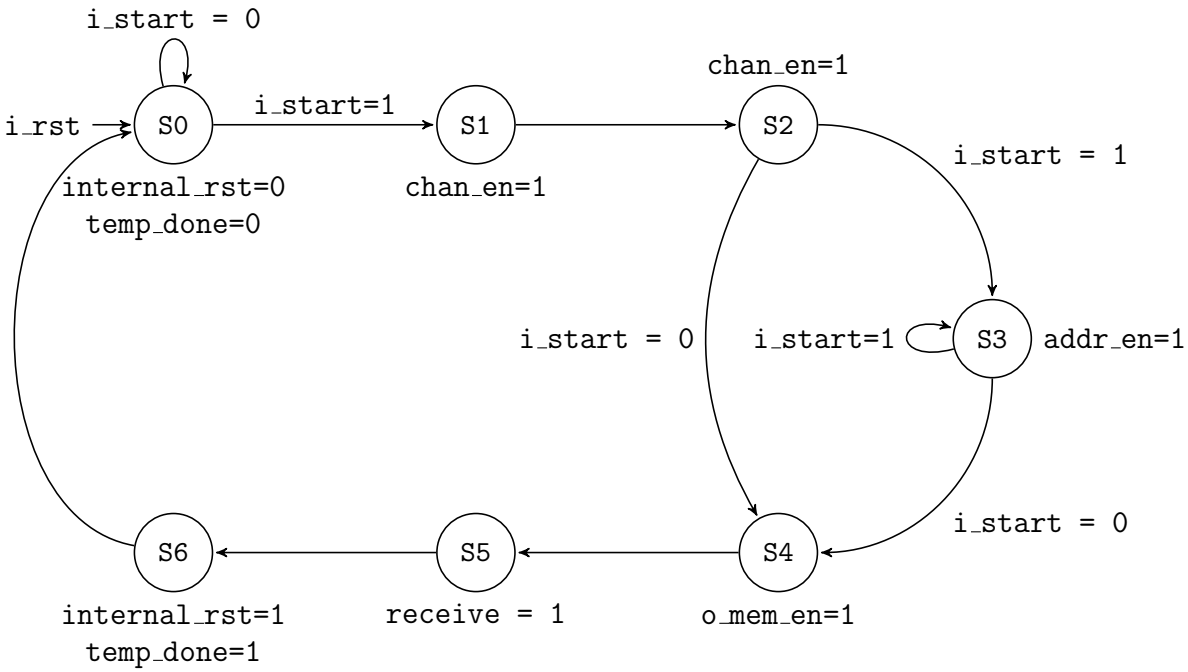

Figure 3: Diagramma degli stati della Macchina a Stati Finiti

Di seguito viene mostrato lo schema RTL finale del componenete realizzato:

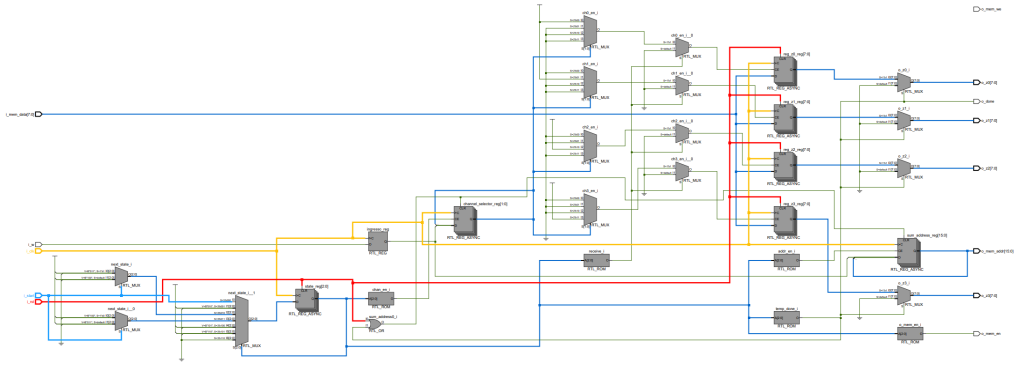

Figure 4: RTL

3 Risultati sperimentali

3.1 Sintesi - report di sintesi

Il componente da noi progettato e implementato, sottoposto ai test-bench forniti (presenti a questo `tbset_new`), non presenta nessuna criticità sia per quanto riguarda la `Behavioral Simulation` sia per quanto riguarda `Post-Synthesis Functional Simulation` e `Post-Synthesis Timing Simulation`.

Ottenuto un componente funzionante, che riuscisse quindi a superare tutti i test proposti in tbset_new, ci siamo concentrati sull'ottimizzazione dello stesso, andando a ridurre il numero di multiplexer utilizzati. Inoltre eseguendo il comando `report_utilization` ci siamo resi conto che venivano utilizzati dei `Latch Register`, i quali potrebbero creare criticità e instabilità durante l'utilizzo del componente. Dopo le opportune modifiche al codice abbiamo ottenuto degli `RTL_REG_ASYNC` e un numero significativamente minore dei multiplexer utilizzati, così da avere un' architettura più solida e consistente.

Per mostrare le modiche apportate al codice vengono mostrati in `Figura 5` e `Figura 6`, rispettivamente, il prima e dopo della stessa porzione dello schematico RTL:

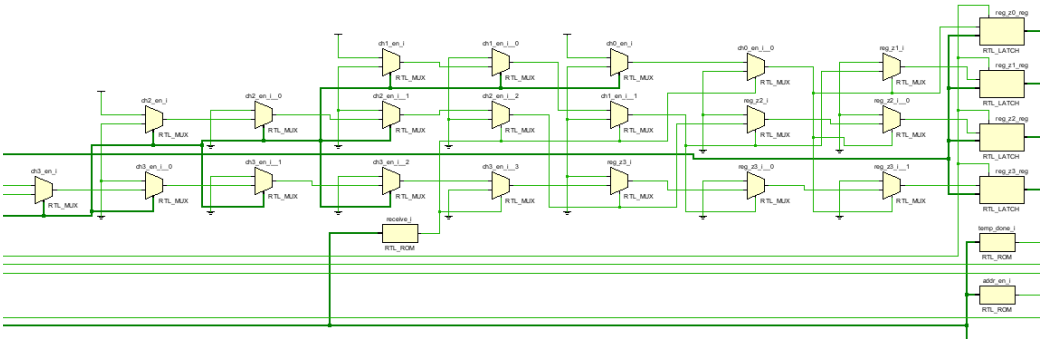

Figure 5: pre ottimizzazione

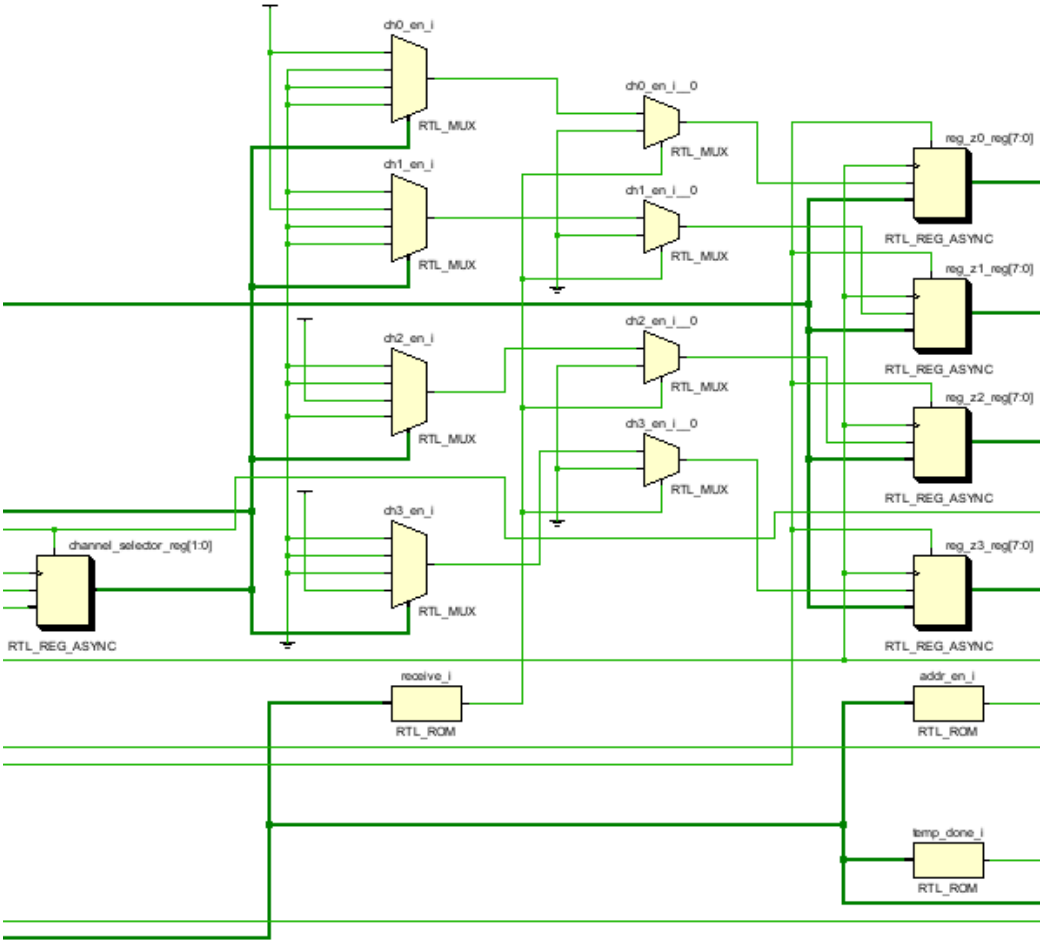

Figure 6: post ottimizzazione (vedi Figura 4 per schema completo)

Di seguito vengono mostrati i report di sintesi, dopo le ottimizzazioni sopra descritte, ottenuti attraverso la TCL Console di Vivavdo con i comandi `report_utilization` e `report_timing`:

1. Slice Logic

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs*	25	0	0	134600	0.02
LUT as Logic	25	0	0	134600	0.02
LUT as Memory	0	0	0	46200	0.00
Slice Registers	58	0	0	269200	0.02
Register as Flip Flop	58	0	0	269200	0.02
Register as Latch	0	0	0	269200	0.00
F7 Muxes	0	0	0	67300	0.00
F8 Muxes	0	0	0	33650	0.00

Figure 7: **report_utilization**

```
Timing Report

Slack (MET) :             97.443ns  (required time - arrival time)
  Source:                 FSM_onehot_state_reg[6]/C
                            (rising edge-triggered cell FDCE clocked by clock  {rise@0.000ns fall@5.000ns period=100.000ns})
  Destination:            channel_selector_reg[0]/CLR
                            (recovery check against rising-edge clock clock  {rise@0.000ns fall@5.000ns period=100.000ns})
  Path Group:             **async_default**
  Path Type:              Recovery (Max at Slow Process Corner)
  Requirement:            100.000ns  (clock rise@100.000ns - clock rise@0.000ns)
  Data Path Delay:        1.968ns  (logic 0.751ns (38.161%)  route 1.217ns (61.839%))
  Logic Levels:           1  (LUT2=1)
  Clock Path Skew:        -0.145ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD):    2.100ns = ( 102.100 - 100.000 )
    Source Clock Delay      (SCD):    2.424ns
    Clock Pessimism Removal (CPR):    0.178ns
  Clock Uncertainty:      0.035ns  ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
    Total System Jitter     (TSJ):    0.071ns
    Total Input Jitter      (TIJ):    0.000ns
    Discrete Jitter          (DJ):    0.000ns
    Phase Error              (PE):    0.000ns
```

Figure 8: **report_timing**

Guardando allo `slack (MET)`, possiamo notare come il nostro componente potrebbe girare ad una frequenza di clock notevolmente maggiore rispetto a quella usata in fase di test.

3.2 Simulazioni e testbench

Una volta progettato e implementato il nostro componente abbiamo iniziato a simulare il nostro componente utilizzando diverse tipologie di simulation sources, in primo luogo ci siamo soffermati su quelle proposte nella pagina del corso, verificando per prima cosa che per ognuna di esse venisse superata la Behavioral Simulation, poi abbiamo spostato l'attenzione sui comportamenti durante la Post-Synthesis Functional Simulation.
Ultimata questa fase, abbiamo poi generato in maniera casuale attraverso uno script python, un set di testbench (disponibile nella repo github del progetto) per cercare di coprire quante più casistiche possibili e poter essere sicuri di avere un componente affidabile.

3.2.1 Casi Limite

Verificato che non venisse riscontrata nessuna problematica abbiamo analizzato quali potevano essere i casi limite di particolare interesse, così da poter costruire delle simulation sources adatte a testarli e poter verificare anche in questi casi il comportamento del componente.
Vengono elencati di seguito i casi limite analizzati:

- nessun indirizzo (coperto anche dal tb_7 fornito)
- utilizzo di tutti i canali
- inserimento di tutti 0
- inseriemeto di tutti 1 (coperto anche dal tb_6 fornito)
- inserimento di 1 e poi tutti 0
- inserimento di 0 e poi tutti 1
- reset multipli
- indirizzo completo

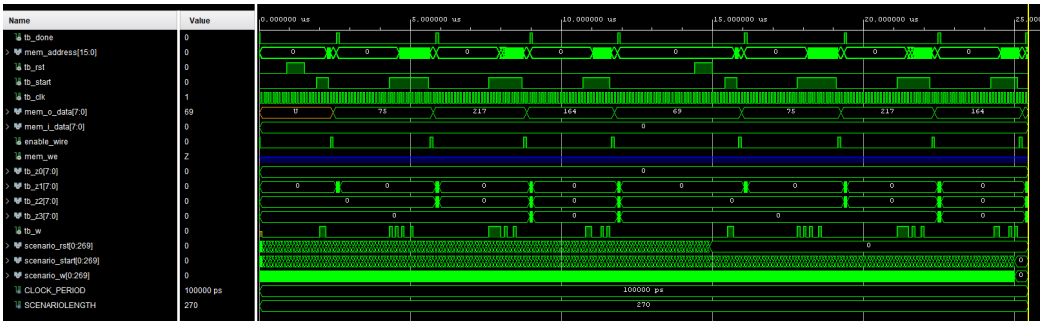

Figure 9: **multiple reset**

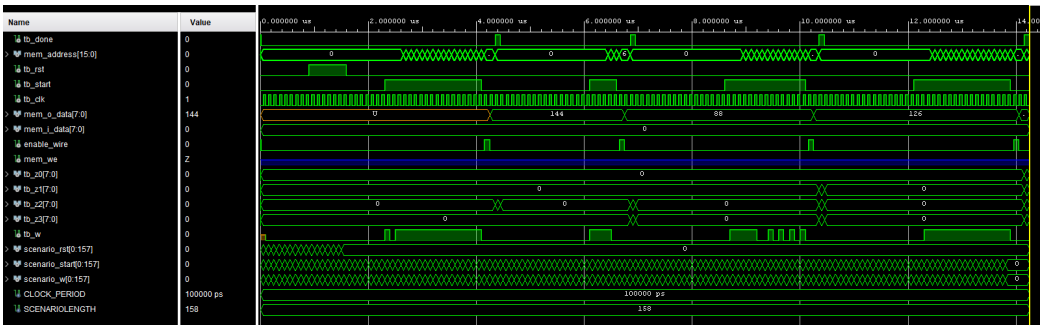

Figure 10: **full address**

4 Conclusione

Scopo del progetto era quello di realizzare un componente che andasse ad effettuare il processo di dereferenziazione di un indirizzo di memoria, così da poter accedere al contenuto in memoria ed esporlo nel canale di uscita indicato.
Riteniamo che il componente da noi pensato, progettato e sviluppato soddisfi in maniera esaustiva le speicifiche di progetto richieste e che svolga quanto richiesto in maniera efficiente, come dimostrato attraverso il report di sintesi dove possiamo notare come il periodo di clock utilizzato per portare a termine le task sia di gran lunga inferiore al massimo consentito.
L'architettura proposta inoltre è stata anche oggetto di sostanziali ottimizzazioni nel corso dello sviluppo così da poter offrire, nella versione finale, una versione molto compatta e di facile implementazione, pur mantenendo integrità e performance, garantite dai numerosi test effettuati sia per quanto riguarda i casi limite (e quindi andando a stressare il componente in quelle casistiche dove potevano essere riscontrate delle criticità) sia per quanto riguarda l'affidabilità e persistenza nel soddisfare i requisiti richiesti attraverso la sottoposizione a numerosi test generati in modo pseudo-casuale.

Anche se non richiesto, sui testbench forniti sulla pagina del corso è stata effettuata la Post-Synthesis Timing Simulation, risultata positiva in tutti i casi, che ne prova l'effettivo funzionamento sulla FPGA utilizzata.