Progetto Finale di Reti Logiche

Simone Controguerra

Anno 2022-2023

Indice

1 Introduzione

1.1 Obiettivo del progetto

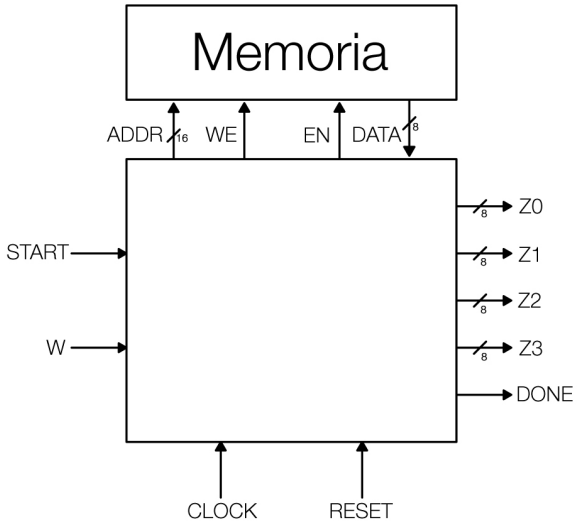
Lo scopo del progetto è quello di descrivere un modulo in linguaggio VHDL in grado di ricevere in ingresso una sequenza di bit di lunghezza variabile in ingresso e di inoltrare in uscita sequenze di 8 bit disponibili su più canali.

1.2 Interfacce del componente

I segnali di input sono: `START`, segnale ausiliario che, quando è alto, segnala la validità dei bit in ingresso; `W`, l'ingresso effettivo da cui vengono letti i bit; `DATA`, una sequenza di 8 bit ricevuti, in seguito a una richiesta, da una memoria esterna di tipo RAM; il `CLOCK` e il `RESET`.

I segnali di output, invece, sono: `ADDR`, una sequenza di 16 bit, passati alla memoria RAM, che funge da indirizzo per ricavare il dato da trasmettere in uscita; il `Write Enable`, segnale che permette la scrittura in memoria (in questo modulo sarà sempre basso poiché non è utile allo scopo del progetto); l'`ENable`, segnale che permette, quando è alto, di accedere alla memoria durante la richiesta del dato; i canali di uscita `Z0`, `Z1`, `Z2` e `Z3`, ovvero i canali dove verranno trasmessi i dati ottenuti dalla memoria; `DONE`, un segnale ausiliario indicatore della completa produzione del risultato e della sua trasmissione in uscita.

Di seguito viene riportato il modulo illustrato con tutti i segnali di input e di output.



1.3 Specifiche generali

Più precisamente, il componente è in grado di adottare il seguente comportamento: dati almeno 2 bit e massimo 18 bit, che verranno mandati in ingresso individualmente per ogni ciclo di clock, in cui un apposito segnale (`START`) rimane alto per segnalare la validità dei bit stessi, il modulo decodifica il canale di uscita, tramite la lettura dei primi 2 bit, dove verrà trasmesso il dato ricevuto ad una memoria esterna, alla quale verrà fornito un indirizzo di 16 bit composto dai bit ricevuti in ingresso successivi ai primi 2 bit utilizzati per determinare il canale di uscita. Qualora non vengano letti precisamente 16 bit, il modulo si occuperà di inserire a sinistra i bit 0 necessari per formare un indirizzo di dimensione corretta.

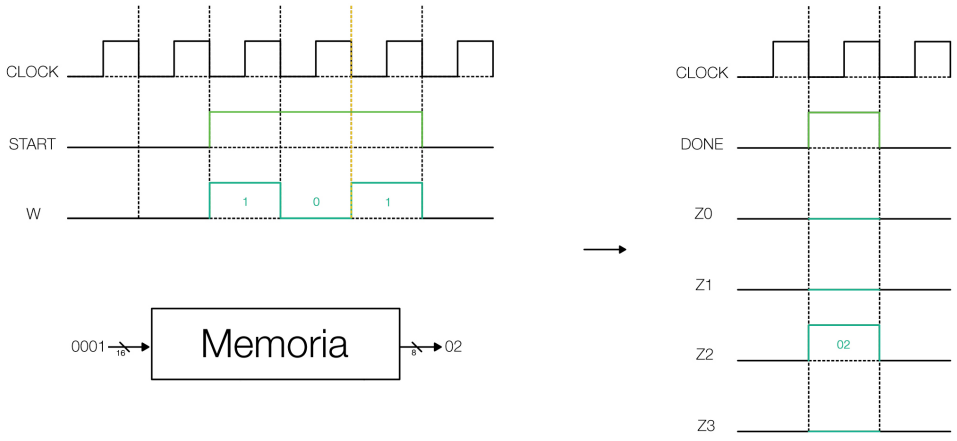
A meno di segnali alti di reset, il componente terrà memoria in ciascun canale il dato trasmesso in uscita, a meno di casi di sovrascrittura. Pertanto, ogni qualvolta verrà alzato un segnale apposito (`DONE`) per segnalare la completa produzione del risultato, ogni canale mostrerà il dato memorizzato, se presente. In qualunque momento, un segnale alto del reset provoca la re-inizializzazione del modulo che cancellerà tutti i dati salvati durante le letture precedenti.

1.4 Illustrazione esemplificativa

Per facilitare la comprensione della specifica, si prosegue con un esempio. Il grafico riportato presenta sulla sinistra l'input, mentre sulla destra l'output.

Si osserva che in un momento arbitrario il segnale `START` viene alzato per la durata di 3 cicli di clock: grazie alla lettura nei primi due cicli (si legge da `W` prima 1 e poi 0) viene scelta la seconda uscita Z2 (con codifica `10`). Nell'ultimo ciclo con `START` ancora alto viene letto un 1 che andrà a rappresentare dunque un indirizzo `0000 0000 0000 0001`.

La memoria RAM esterna comunica, in risposta all'indirizzo ricevuto, il dato `0000 0010`. Quando il segnale `DONE` viene alzato, sul canale `Z2` compare il dato ottenuto dalla memoria.



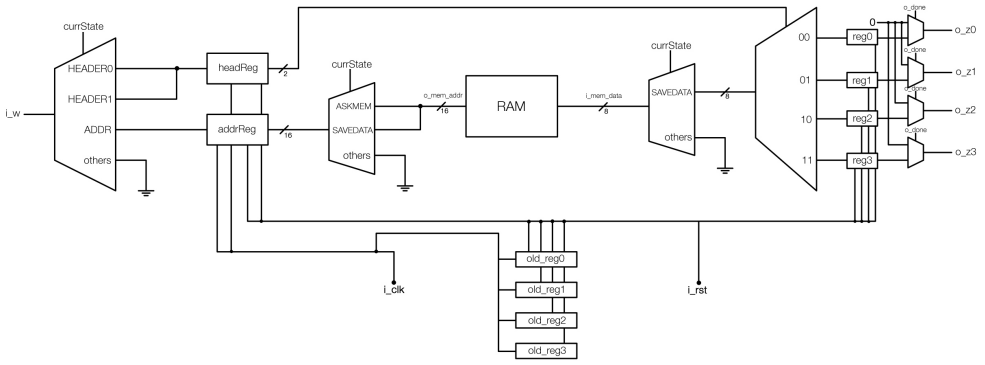
1.5 Interfaccia dell'entità

Di seguito viene riportata la descrizione dell'entità del modulo utilizzata per il progetto in linguaggio VHDL.

```
entity project_reti_logiche is
port (
    i_clk :         in std_logic;
    i_rst :         in std_logic;
    i_start :       in std_logic;
    i_w :           in std_logic;
    o_z0 :          out std_logic_vector(7 downto 0);
    o_z1 :          out std_logic_vector(7 downto 0);
    o_z2 :          out std_logic_vector(7 downto 0);
    o_z3 :          out std_logic_vector(7 downto 0);
    o_done :        out std_logic;
    o_mem_addr :    out std_logic_vector(15 downto 0);
    i_mem_data :    in std_logic_vector(7 downto 0);
    o_mem_we :      out std_logic;
    o_mem_en :      out std_logic
);
end project_reti_logiche;
```

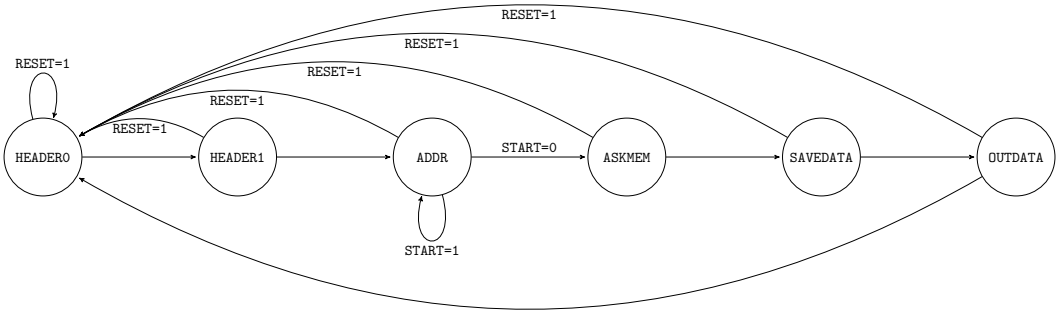
2 Architettura

Il modulo è caratterizzato da 4 processi sincroni e da una macchina a stati finiti, dotata di sei stati e implementata tramite due processi, per un totale complessivo di 6 processi. Ciascun processo si occupa di implementare le diverse parti che vengono illustrate nello schematico sintetico qui di seguito riportato.



2.1 Macchina a stati finiti

Il modulo è regolato da una macchina a stati finiti. Nel progetto viene implementata attraverso un processo sincrono, che si occupa di gestire le transizioni degli stati, e da un processo asincrono, che viene invece utilizzato per settare i diversi segnali utilizzati. Qui di seguito viene riportato un grafico rappresentante gli stati della macchina e le sue transizioni.



La macchina è caratterizzata da 6 stati:

`HEADER0` : è lo stato di reset della macchina ed è pronto a leggere il primo bit dall'ingresso sul fronte di salita del clock, appena il segnale `START` viene alzato. Dopo un ciclo di clock avviene la transizione allo stato successivo.

`HEADER1` : viene letto il secondo bit dall'ingresso sul fronte di salita del clock. Avviene la transizione allo stato successivo dopo un ciclo di clock.

`ADDR` : finché il segnale `START` è alto, vengono letti i bit dall'ingresso sul fronte di salita del clock. Quando il segnale `START` è basso, viene prodotto l'indirizzo da passare alla memoria esterna e avviene la transizione allo stato successivo sul fronte di salita del clock.

`ASKMEM` : il segnale `ENable` viene alzato e viene inviata alla memoria RAM l'indirizzo da 16 bit. Dopo un ciclo di clock avviene la transizione allo stato successivo.

`SAVEDATA` : il segnale `ENable` viene abbassato e viene ricevuta la sequenza di 8 bit dalla memoria esterna e viene salvata in un registro correlato al canale di uscita decodificato nello stato `HEADER1`. Dopo un ciclo di clock avviene la transizione allo stato successivo.

`OUTDATA` : il segnale `DONE` viene alzato per la durata di un ciclo di clock, durante il quale sui canali di uscita vengono trasmessi i dati contenuti nei registri correlati. Dopo un ciclo di clock il segnale `DONE` viene abbassato e si torna allo stato iniziale.

2.2 Determinazione del canale di uscita

Come già anticipato, il progetto è diviso in quattro processi. Il primo tra questi è l'`headerMaker`, ovvero quel processo che si occupa di determinare quale dei quattro canali di uscita è stato indicato dai primi due bit provenienti dall'ingresso.

Si tratta di un processo sincronizzato con i segnali di clock, affinché l'ingresso venga campionato su ogni fronte di salita, e con il segnale di reset, in modo tale da poter azzerare tutti i registri e i segnali ausiliari utilizzati in caso di segnale alto.

Il processo utilizza un registro `headReg` a scorrimento da 2 bit inizializzato a 0, dove verranno inseriti i primi bit letti da input che determineranno il canale d'uscita desiderato, ed un segnale secondario `secondHeaderBit`, utilizzato per tener conto del numero di bit letti.

Quando la macchina a stati finiti definita si trova nello stato `HEADER0`, il componente legge dall'ingresso `W` un bit che viene inserito nella posizione meno significativa. Appena avviene la transizione nello stato `HEADER1`, viene eseguita una concatenazione tra il bit appena inserito nel registro, durante il ciclo di clock precedente, e il bit appena letto dall'ingresso. Questo registro verrà utilizzato più avanti come filtro per selezionare il canale corretto dove inoltrare il dato. Nelle transizioni di stato successive, prima di tornare allo stato iniziale, il registro e il segnale ausiliario verranno re-inizializzati nello stato `OUTDATA`.

Di seguito, viene riportata l'implementazione in VHDL del processo.

```
headerMaker : process(i_clk, i_rst)
    begin
        if i_rst = '1' then
            headReg <= (others=>'0');
            secondHeaderBit <= '0';
        else
            if currState = HEADER0 or currState = HEADER1 then
               if i_start = '1' and rising_edge(i_clk) then
                   if secondHeaderBit = '0' then
                        headReg <= headReg(0) & i_w;
                        secondHeaderBit <= '1';
                   else
                        headReg <= headReg(0) & i_w;
                    end if;
                end if;
            end if;
            if currState = OUTDATA then
                headReg <= (others=>'0');
                secondHeaderBit <= '0';
            end if;
       end if;
    end process;
```

2.3 Costruzione dell'indirizzo

Qui viene trattato il processo `addressMaker` che si occupa di leggere da input tutti i bit che andranno a formare l'indirizzo da inoltrare alla memoria esterna.

È un processo sincronizzato con il segnale di clock, affinché tutti i bit vengano letti sul fronte di salita del clock, e con il segnale di reset, in modo da azzerare il registro utilizzato in questo processo se necessario.

Viene utilizzato un registro a scorrimento `addrReg` da 16 bit inizializzato a 0. Finché la macchina a stati finiti definita si trova nello stato `ADDR`, vengono inseriti uno ad uno i bit letti dall'ingresso `W` nella posizione meno significativa e viene fatto scorrere a sinistra il registro ad ogni lettura. Quando il segnale `START` si abbasserà, non potranno più avvenire campionamenti dall'ingresso e verrà salvato il registro `addrReg` nel segnale `o_mem_addr` definito nell'entità. Durante lo stato `OUTDATA`, ovvero prima di tornare allo stato di reset del modulo, il registro ausiliario viene azzerato.

Di seguito, viene riportata l'implementazione in VHDL del processo.

```
addressMaker : process(i_clk, i_rst)
    begin
        if i_rst = '1' then
            addrReg <= (others=>'0');
        elsif currState = ADDR then
            if i_start = '1' then
                if rising_edge(i_clk) then
                    addrReg <= addrReg(14 downto 0) & i_w;
                end if;
            end if;
        end if;
        if currState = OUTDATA then
            addrReg <= (others=>'0');
        end if;
    end process;
```

2.4 Salvataggio dei dati nei registri correlati

Il processo `loadingData` si occupa di salvare sui registri correlati ai canali di uscita i dati ottenuti dalla memoria esterna, dopo aver fatto richiesta tramite inoltro dell'indirizzo costruito nel processo precedente.

Si tratta di un processo sincronizzato con il segnale di clock e con il segnale di reset, affinché tutti i registri utilizzati possano essere re-inizializzati.

Vengono utilizzati quattro registri da 8 bit, ciascuno correlato ad un canale di uscita, inizializzati a 0. Quando la macchina a stati definita precedentemente si trova nello stato `SAVEDATA`, viene utilizzato il registro `headReg`, riempito a dovere in precedenza nel processo `headerMaker`, con lo scopo di salvare nel registro correlato corretto il dato disponibile sul segnale `i_mem_data`, ricevuto dalla memoria RAM.

Ai rimanenti registri vengono, invece, assegnati il loro vecchio valore conservato in ulteriori quattro registri ausiliari che verranno trattati nella prossima sottosezione.

Al di fuori dello stato `SAVEDATA`, i registri prenderanno i loro vecchi valori conservati nei registri secondari. Quando la macchina si troverà nello stato `OUTDATA`, il segnale `DONE` verrà alzato e il contenuto dei canali verrà trasmesso sui canali in uscita.

Di seguito, viene riportata l'implementazione in VHDL del processo.

```
loadingData : process(i_clk, i_rst)
    begin
        if i_rst = '1' then
            reg0 <= (others=>'0');
            reg1 <= (others=>'0');
            reg2 <= (others=>'0');
            reg3 <= (others=>'0');
        elsif currState = SAVEDATA then
            case headReg is
                when "00" =>
                    reg0 <= i_mem_data;
                    reg1 <= old_reg1;
                    reg2 <= old_reg2;
                    reg3 <= old_reg3;
                when "01" =>
                    reg0 <= old_reg0;
                    reg1 <= i_mem_data;
                    reg2 <= old_reg2;
                    reg3 <= old_reg3;
                when "10" =>
                    reg0 <= old_reg0;
                    reg1 <= old_reg1;
                    reg2 <= i_mem_data;
                    reg3 <= old_reg3;
                when "11" =>
                    reg0 <= old_reg0;
                    reg1 <= old_reg1;
                    reg2 <= old_reg2;
                    reg3 <= i_mem_data;
                when others =>
                    reg0 <= old_reg0;
                    reg1 <= old_reg1;
                    reg2 <= old_reg2;
                    reg3 <= old_reg3;
            end case;
        else
            reg0 <= old_reg0;
            reg1 <= old_reg1;
            reg2 <= old_reg2;
            reg3 <= old_reg3;
        end if;
    end process;
```

2.5 Utilizzo di registri temporanei per evitare la generazione di latch

Il quarto e ultimo processo `keepDATA` viene utilizzato con lo scopo di non generare alcun latch durante la sintesi del componente. Verranno trattati qui i registri `old_reg` utilizzati nel processo precedente.

Si tratta di un processo sincrono con il segnale di clock, affinché ogni operazione avvenga sul fronte di salita, e il segnale di reset, per far sì che i registri vengano azzerati se necessario.

Sul fronte di salita del segnale del clock, i quattro registri prendono i valori dei loro registri correlati utilizzati nel processo `loadingData`, che contengono i dati ottenuti dalla memoria RAM, affinché possano conservarne il loro vecchio valore. L'utilizzo di questi registri secondari si ritiene necessario affinché non venga generato alcun latch durante la sintesi per il mancato assegnamento di un valore a quei registri non corrispondenti al canale di uscita decodificato.

Di seguito, viene riportata l'implementazione in VHDL del processo.

```
keepDATA : process (i_rst, i_clk)
    begin
        if i_rst = '1' then
            old_reg0 <= (others=>'0');
            old_reg1 <= (others=>'0');
            old_reg2 <= (others=>'0');
            old_reg3 <= (others=>'0');
        elsif rising_edge(i_clk) then
            old_reg0 <= reg0;
            old_reg1 <= reg1;
            old_reg2 <= reg2;
            old_reg3 <= reg3;
        end if;
    end process;
```

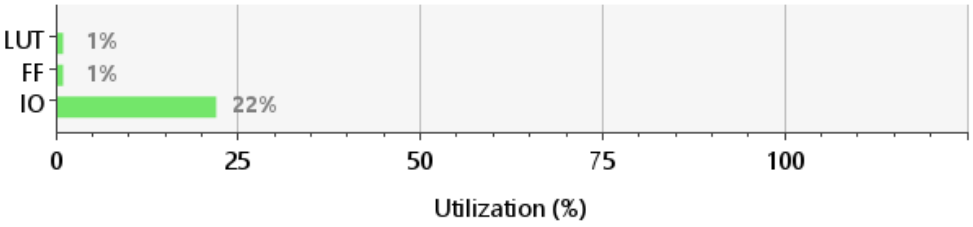
3 Risultati sperimentali

3.1 Report di sintesi

Il modulo definito è sintetizzabile e supera tutti i test, che verranno illustrati successivamente, sia in Behavioral sia in Post Sintesi Functional.

Il componente è stato sintetizzato come mostrato dal report riguardo l'utilizzo riportato di seguito, da cui si evince che sono stati utilizzati 56 registri Flip Flop, 87 lookup table e 66 buffer di input e output.

Resource	Utilization	Available	Utilization %
LUT	87	134600	0.06
FF	56	269200	0.02
IO	63	285	22.11



Come richiesto dalle specifiche del progetto, il componente è funzionate su un segnale di clock di almeno 100ns, come lo dimostra la voce Worst Negative Slack nella sezione Setup del report sul timing generato dalla sintesi.

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	97,332 ns	Worst Hold Slack (WHS):	0,139 ns	Worst Pulse Width Slack (WPWS):	4,500 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	90	Total Number of Endpoints:	90	Total Number of Endpoints:	57

All user specified timing constraints are met.

3.2 Simulazioni

Durante la progettazione del componente, non sono mancate considerazioni su quali potessero essere i casi limite in grado di testare il corretto funzionamento del modulo prodotto. Ne sono stati individuati sei che sono stati raggruppati e suddivisi in tre testbench.

3.2.1 Reset sincrono e asincrono

Il testbench è stato prodotto con lo scopo di testare il corretto funzionamento del componente in risposta ad un segnale alto del reset, sia sul fronte di salita del clock sia sul fronte di discesa.

I grafici ad onda quadra riportati qui di seguito riportano l'andamento dei segnali specificati nell'entità del modulo in seguito all'innalzamento del reset. In particolare si vuole mostrare il caso in cui il reset venga mandato proprio durante la fase di inoltro dei dati sui canali di uscita, con lo scopo di mostrare la corretta reazione in maniera più evidente.

Come si può notare, in entrambi i casi, appena il segnale di reset viene alzato la macchina a stati finiti effettua la transizione allo stato iniziale. Qualora il segnale venga alzato, come accade nel caso sincrono, durante il segnale `DONE` alto, questo viene abbassato e i canali di uscita vengono tutti azzerati.

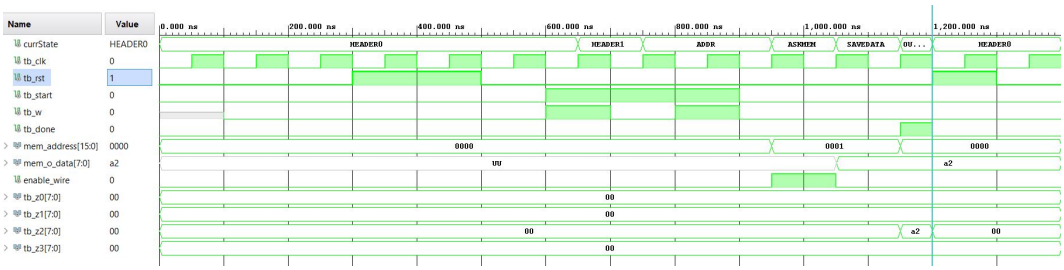


Figura 1: Il segnale `i_rst` viene alzato in maniera sincrona con gli altri segnali, ovvero sul fronte di discesa di `i_clk`.

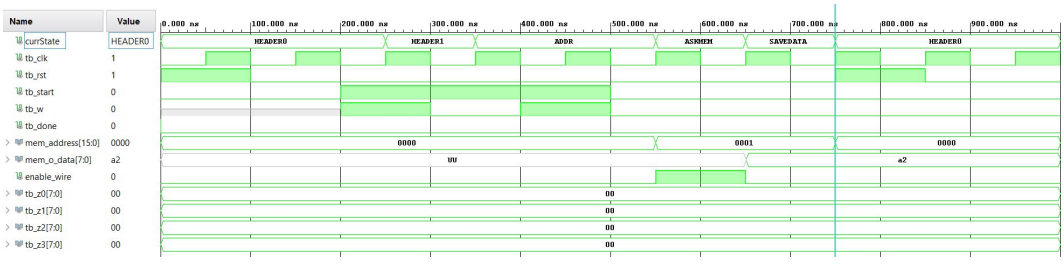
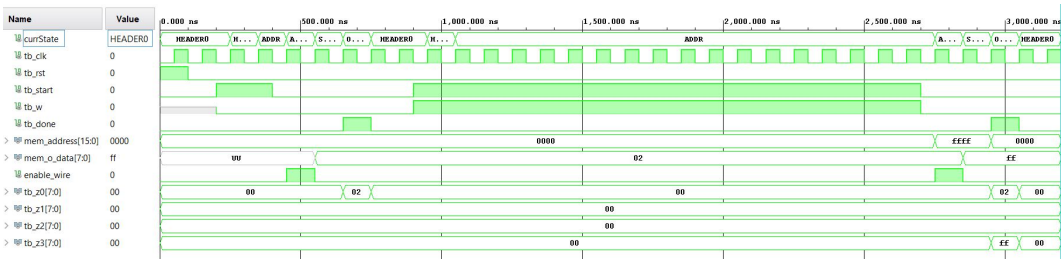


Figura 2: Il segnale `i_rst` viene alzato in maniera asincrona con gli altri segnali, ovvero sul fronte di salita di `i_clk`.

3.2.2 Indirizzo vuoto e indirizzo pieno

Questo testbench ha l'obiettivo di verificare il corretto funzionamento nel caso in cui il segnale `START` venga alzato prima per 2 cicli di clock e poi per 18 cicli di clock, con lo scopo di costruire gli indirizzi di memoria `0000 0000 0000 0000` e `1111 1111 1111 1111`.

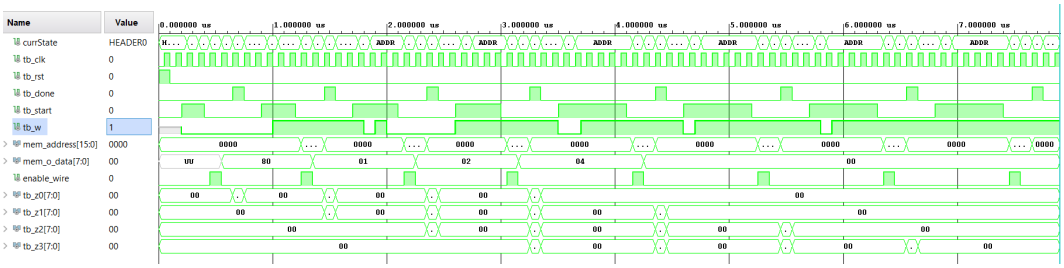
Nel grafico sotto riportato, infatti è possibile notare il segnale `mem_o_data` acquisire prima il valore `0000` e successivamente il valore `ffff`. Il testbench è stato costruito appositamente per riportare sul canale `tb_z0` il valore `02` e sul `tb_z1` il valore `ff`.



3.2.3 Scrittura e sovrascrittura di tutti i canali

Il testbench ha lo scopo di verificare che tutti i canali vengano scritti e sovrascritti contemporaneamente.

Viene riempito uno ad uno ciascun canale ed una volta aver scritto per la prima volta sul canale `tb_z3`, viene svuotato volta per volta ogni canale, sovrascrivendo il vecchio dato con una sequenza di 0.



4 Conclusioni

La documentazione termina dopo aver illustrato tutti gli aspetti architetturali del componente (tutti i segnali utilizzati, i processi e la macchina a stati finiti che regola il funzionamento globale del modulo) e averne provato la corretta esecuzione nei diversi casi limite individuati nello studio della sua progettazione.

Il componente si comporta in maniera corretta e coerente sia in simulazione comportamentale sia in simulazione post-sintesi, nel rispetto dei vincoli imposti dalla specifica del progetto, con riguardo verso l'ottimizzazione di tutto il modulo.

In particolare, l'ottimizzazione è stata concentrata sull'utilizzo del minimo numero di stati. Grazie all'impiego dei registri a scorrimento è stato possibile raggruppare i 16 stati astratti, utilizzati per poter leggere il massimo numero possibile di bit dall'ingresso, in un unico stato. Potrebbe essere effettuata un'ulteriore ottimizzazione unificando i due stati responsabili della lettura dei primi 2 bit, fondamentali per la codifica del canale d'uscita, ma per maggiore semplicità e leggibilità del codice si è preferito tenerli separati.

Infine, il modulo viene sintetizzato con successo dal software Vivado, rispettando i limiti temporali imposti dalla specifica.