POLITECNICO
MILANO 1863

Prova Finale – Progetto di Reti Logiche

Anno Accademico 2023/2024

Luca De Nicola
Cod. Persona: 10808901

1 Introduzione

1.1 Descrizione generale

Il modulo da implementare presenta tre ingressi a 1 bit utilizzati per la gestione dei segnali di **clock**, **reset** e **start**. Dispone di due ingressi principali per la ricezione dei dati: un segnale da 8 bit (**i_k**) e un segnale da 16 bit (**i_add**). L'output include un segnale a 1 bit (**o_done**) per indicare la terminazione dell'elaborazione.

Inoltre, il modulo interagisce con una memoria RAM attraverso i seguenti segnali:

- Due uscite a 1 bit: una per abilitare la comunicazione con la RAM, sia in lettura che in scrittura, (**o_mem_en**) e una per abilitarla in scrittura (**o_mem_we**),
- Un'uscita da 16 bit (**o_mem_addr**) per specificare l'indirizzo di memoria,
- Due porte dati a 8 bit: una in uscita (**o_mem_data**) e una in ingresso (**i_mem_data**), per lo scambio di dati con la memoria.

Il seguente diagramma descrive l'interfaccia così definita:

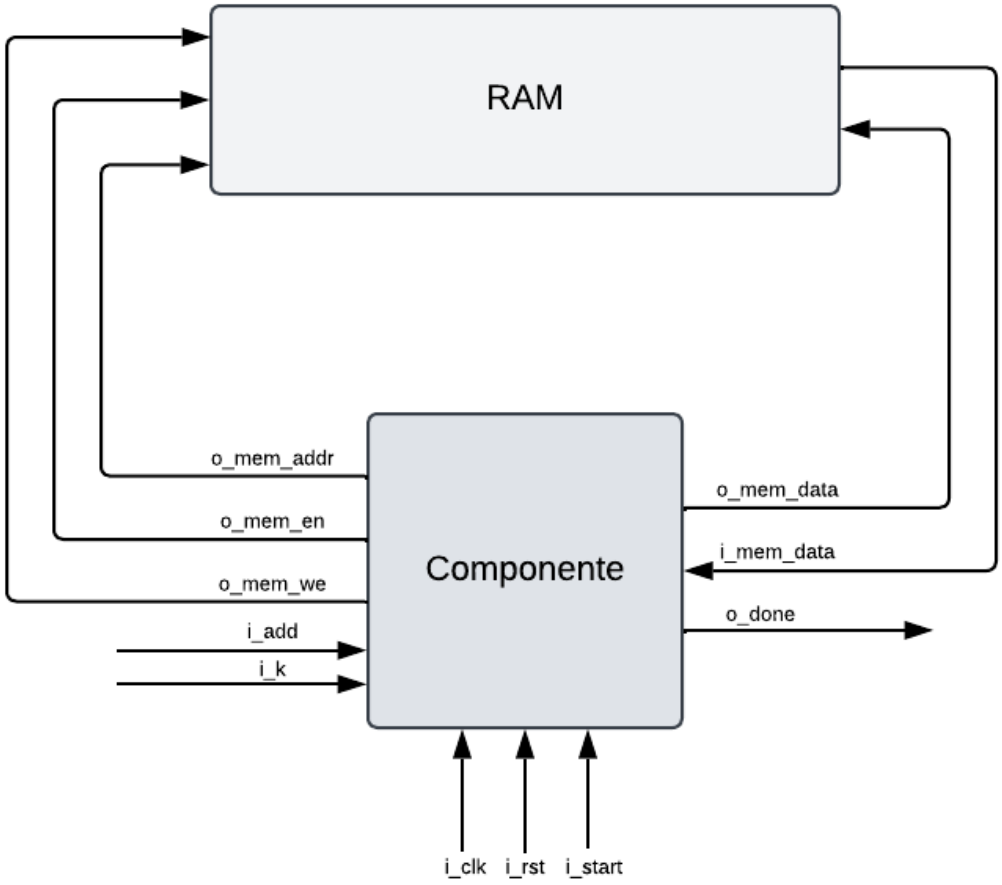


Figura 1: Modulo da implementare

1.2 Funzionamento

Il modulo legge una sequenza di K parole W, memorizzata a partire da un indirizzo iniziale e distribuita ogni 2 byte consecutivi. Durante l'elaborazione, il modulo sostituisce ogni zero con l'ultimo valore valido precedentemente letto e assegna un valore di "credibilità" C al byte successivo. Il valore C parte da 31 e viene decrementato ogni volta che si incontra un dato mancante, fino a un minimo di 0. C viene reimpostato a 31 al primo valore diverso da zero della sequenza.

Il modulo richiede tre ingressi principali: un segnale di clock (**i_clk**), un reset asincrono (**i_rst**), e un segnale di avvio (**i_start**), oltre a due segnali dati (**i_add**, 16 bit, e **i_k**, 10 bit). L'uscita principale è il segnale **i_done**, che notifica il completamento dell'elaborazione. Il comportamento del modulo è sincrono al fronte di salita del clock, fatta eccezione per il segnale di reset.

Durante l'inizializzazione, DONE deve essere impostato a 0, e il modulo avvierà l'elaborazione solo quando START è attivo (alto). Il segnale START rimane alto fino alla fine del processo, momento in cui viene alzato il segnale DONE. Un nuovo comando START non può essere ricevuto fin tanto che DONE è alto.

```
entity project_reti_logiche is
    port (
        i_clk   : in std_logic;
        i_rst   : in std_logic;
        i_start : in std_logic;
        i_add   : in std_logic_vector(15 downto 0);
        i_k     : in std_logic_vector(9 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_data : out std_logic_vector(7 downto 0);
        o_mem_we   : out std_logic;
        o_mem_en   : out std_logic
    );
end project_reti_logiche;
```

Figura 2: Interfaccia del componente

2 Architettura

2.1 Datapath

L'architettura del componente, progettato per soddisfare i requisiti e rispettare le specifiche indicate, è rappresentata nello schema a blocchi riportato di seguito.

Come evidenziato nello schema, l'architettura è composta da sette blocchi funzionali principali: **FSM** (Finite State Machine), **add_register**, **k_register**, **w_register**, **last_w_valid_register**, **c_register** e **MUX**. Ogni blocco svolge un ruolo specifico e contribuisce al corretto funzionamento del sistema, garantendo l'elaborazione conforme alle indicazioni fornite nella specifica.

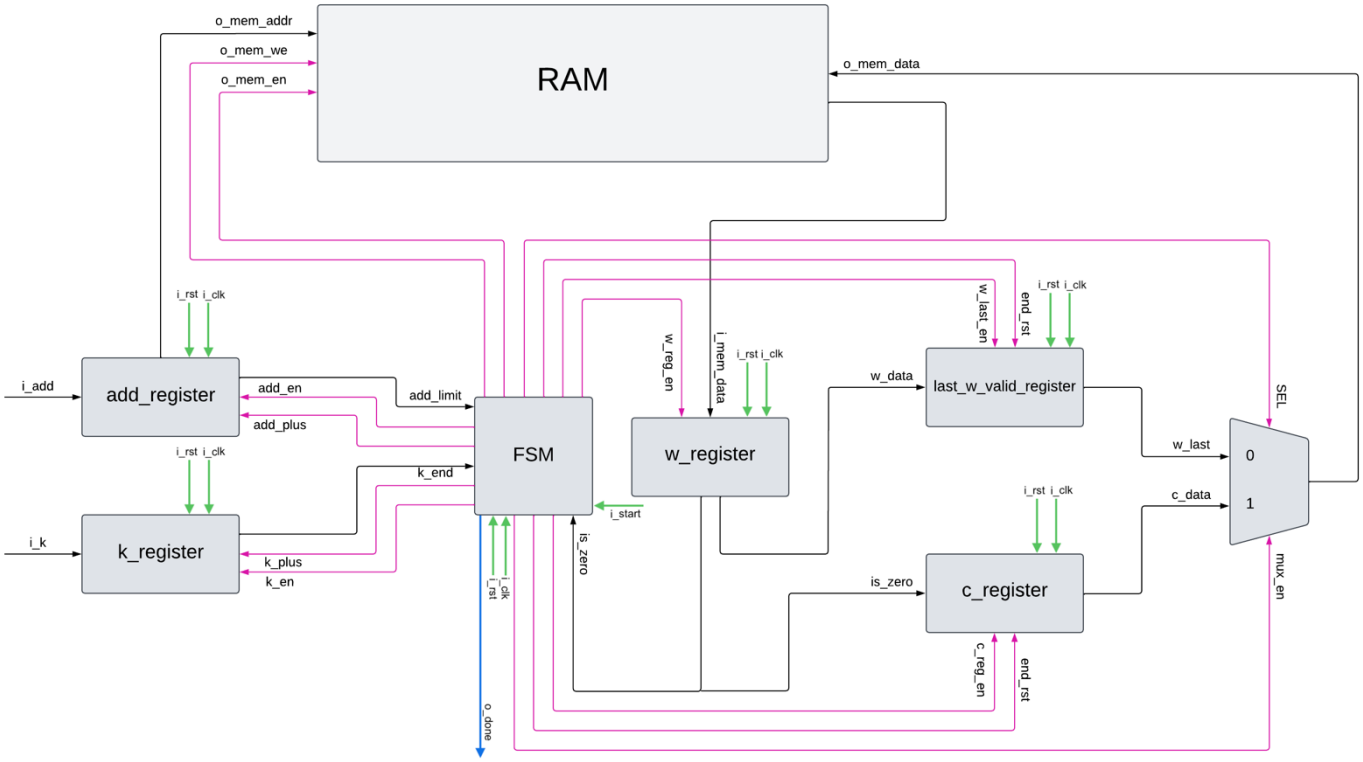


Figura 3: Datapath e architettura del componente

2.2 Moduli

2.2.1 Add Register

L'Add Register è il componente che gestisce il salvataggio e l'aggiornamento dell'indirizzo iniziale della RAM, contenuto nel segnale di ingresso **i_add**, rappresentante il punto di partenza della sequenza da elaborare. È composto da un processo sequenziale, sincronizzato al clock (**i_clk**), che aggiorna il registro **address** con il valore di **i_add** se il segnale di abilitazione (**add_en**) è attivo, oppure lo incrementa di 1 se il segnale **add_plus** è attivo. In caso di reset (**i_rst**), il registro viene azzerato. Un processo combinatorio verifica poi se il valore di **address** ha raggiunto il limite massimo di 16 bit (1111111111111111), impostando in tal caso il segnale

add_limit a 1; altrimenti, **add_limit** rimane a 0. Il valore corrente del registro **address** viene inviato al segnale di uscita **o_mem_addr**, rappresentando l'indirizzo attuale per l'accesso alla memoria. Questo componente assicura una gestione sicura e coerente degli indirizzi, rispettando i limiti del sistema.

```
add_register_1 : process(i_clk, i_rst)
-- Sequential process ADD_REGISTER
begin
    if i_rst = '1' then
        address <= (others => '0');
    elsif rising_edge(i_clk) then
        if add_en = '1' then
            address <= i_add;
        elsif add_plus = '1' then
            address <= std_logic_vector(unsigned(address) + 1);
        end if;
    end if;
end process add_register_1;
```

```
add_register_2 : process(address, add_plus)
-- Combinational process ADD_REGISTER
begin
    add_limit <= '0';                --Default value
    if address = "1111111111111111" then
        add_limit <= '1';
    end if;
end process add_register_2;

-- Output ADD_REGISTER
o_mem_addr <= address;
```

2.2.2 K Register

Il K Register è il componente che memorizza e gestisce il valore k, rappresentante il numero di parole W presenti nella sequenza da elaborare. È costituito da un processo sequenziale e uno combinatorio. Il processo sequenziale, sincronizzato al clock (**i_clk**) e al reset (**i_rst**), inizializza il registro **stored_k** a 0 quando il reset è attivo. Se il segnale di abilitazione (**k_en**) è alto, il valore di ingresso **i_k** viene salvato e il contatore temporaneo **k_temp**, che gestisce l'avanzamento interno per garantire un'elaborazione coerente con il valore k, viene impostato a 1. Il processo combinatorio verifica il completamento della sequenza impostando il segnale **k_end** a 1 nel caso particolare in cui **stored_k** è 0 o quando **k_temp** raggiunge il massimo valore. Se **k_plus** è attivo, **k_temp** viene incrementato fino a eguagliare **stored_k**. Una volta eguagliato l'output **k_end** segnala la fine della sequenza.

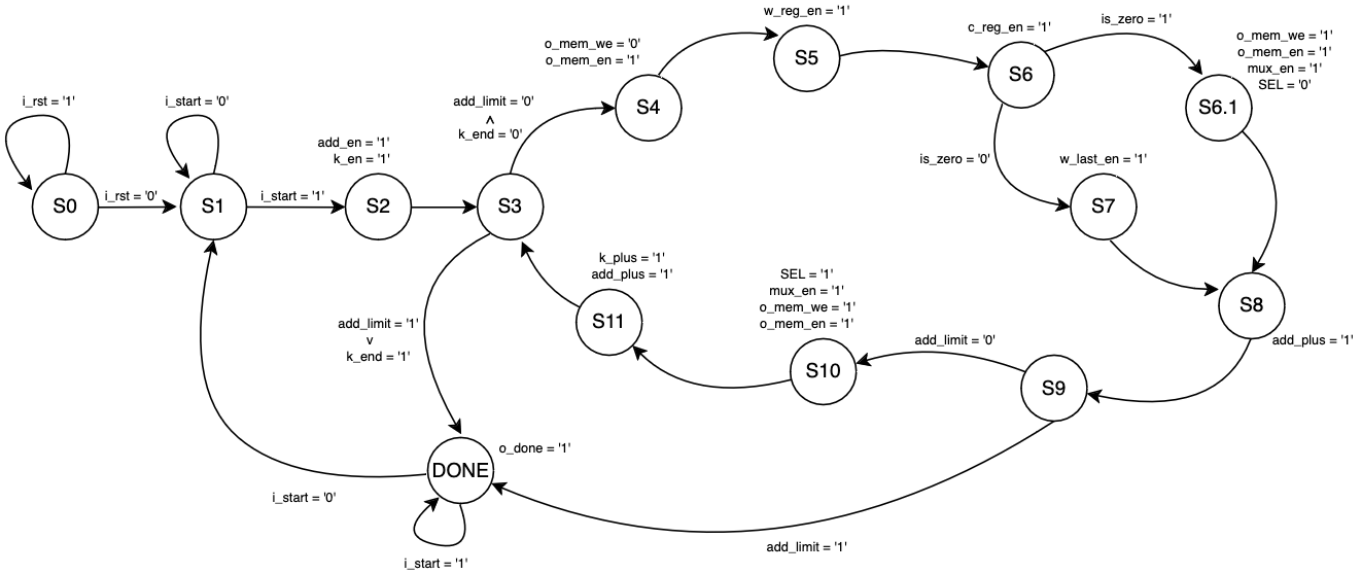
```
k_register_1 : process(i_clk, i_rst)
-- Sequential process K_REGISTER
begin
    if i_rst = '1' then
        stored_k <= (others => '0');
    elsif rising_edge(i_clk) then
        if k_en = '1' then
            stored_k <= i_k;
        k_temp <= "0000000001"; --parte da 1
        end if;
    end if;
end process k_register_1;
```

```
k_register_2 : process(stored_k, k_plus)
-- Combinational process K_REGISTER
begin
    k_end <= '0';       --Default
    k_temp <= k_temp;   --Default

    if stored_k = "0000000000" then
        k_end <= '1';
    elsif k_temp = "1111111111" then
        k_end <= '1';
    elsif k_plus = '1' then
        if k_temp = stored_k then
            k_end <= '1';
        else
            k_end <= '0';
            k_temp <= std_logic_vector(unsigned(k_temp) + 1);
        end if;
    end if;
end process k_register_2;
```

2.2.3 FSM

La FSM è il componente centrale del modulo, responsabile della gestione del flusso di controllo e del coordinamento dei segnali per l'elaborazione della sequenza. È composta da due processi distinti: un processo sequenziale, che gestisce la transizione tra gli stati in base al clock e alle condizioni specificate, e un processo combinatorio, che genera i segnali di controllo in uscita in base allo stato corrente. Questa separazione garantisce un design modulare e chiaro, in cui le transizioni degli stati e la logica di controllo operano in maniera indipendente ma coordinata, assicurando che tutte le operazioni vengano eseguite nell'ordine e nei tempi previsti.



Stati della FSM:

- **S0** (**Reset**): Stato di reset; il sistema è in attesa che il segnale di reset si abbassi.
- **S1** (**Attesa start**): Rimane in attesa del segnale di start; Attivazione del reset secondario (**end_rst** impostato a 1).
- **S2** (**Caricamento iniziale**): Abilitazione del caricamento di **i_add** e **i_k** nei rispettivi registri.
- **S3** (**Verifica**): Prosegue nello stato S4 se i segnali di terminazione dei registri add e k sono entrambi bassi, altrimenti passa allo stato di DONE.
- **S4** (**Abilitazione memoria**): Abilitazione della memoria in lettura.
- **S5** (**Scrittura del dato**): Salvataggio del dato letto nel registro w_register.
- **S6** (**Aggiornamento credibilità**): Aggiornamento del valore di credibilità contenuto nel c_register.
- **S6_1** (**Scrittura W**): Attivazione del multiplexer e scrittura dell'ultimo valore W valido in memoria.
- **S7** (**Salvataggio W**): Salvataggio dell'attuale valore W nel registro last_w_valid_register.
- **S8** (**Incremento indirizzo**): Incremento dell'indirizzo corrente.

- **S9** (**Verifica limite**): Controlla se è stato raggiunto il limite di indirizzamento. Se il limite è raggiunto, passa allo stato DONE; altrimenti prosegue nello stato S10.
- **S10** (**Scrittura**): Scrittura in memoria del valore aggiornato di credibilità C.
- **S11** (**Incremento**): Incremento dell'indiritto corrente e delle parole elaborate; Ritorna allo stato S3.
- **DONE** (**Conclusione**): Il modulo segnala la conclusione del processo.
  Ritorna a S1 se il segnale **i_start** è basso, altrimenti rimane nello stato DONE.

```
StateProcess : process(i_clk, i_rst) is
--Sequential Process
begin
    if i_rst = '1' then
        curr_state <= S0;
    elsif rising_edge(i_clk) then
        case curr_state is
            when S0 =>
                curr_state <= S1;
            when S1 =>
                if i_start = '1' then
                    curr_state <= S2;
                end if;
            when S2 =>
                curr_state <= S3;
            when S3 =>
                if add_limit = '1' or k_end = '1' then
                    curr_state <= DONE;
                else
                    curr_state <= S4;
                end if;
            when S4 =>
                curr_state <= S5;
            when S5 =>
                curr_state <= S6;
            when S6_1 =>
                curr_state <= S8;
            when S7   =>
                curr_state <= S8;
            when S8    =>
                curr_state <= S9;
            when S9   =>
                if add_limit = '1' then
                    curr_state <= DONE;
                else
                    curr_state <= S10;
                end if;
            when S10    =>
                curr_state <= S11;
            when S11   =>
                curr_state <= S3;
             when DONE =>
                if i_start = '1' then
                    curr_state <= DONE;
                else
                    curr_state <= S1;
                end if;
        end case;
    end if;
end process StateProcess;
```

```
ControlProcess : process(curr_state) is
--Combinational Process
begin
    -- Default outputs
    o_done      <= '0';
    o_mem_we    <= '0';
    o_mem_en    <= '0';
    add_en      <= '0';
    add_plus    <= '0';
    k_en        <= '0';
    k_plus      <= '0';
    w_reg_en    <= '0';
    w_last_en   <= '0';
    c_reg_en    <= '0';
    end_rst     <= '0';
    mux_en      <= '0';
    SEL         <= '0';

    case curr_state is

        when S0 =>
            --nothing
        when S1 =>
            end_rst     <= '1';
        when S2 =>
            add_en      <= '1';
            k_en        <= '1';
        when S3 =>
            --wait check
        when S4 =>
            o_mem_we    <= '0';
            o_mem_en    <= '1';
        when S5 =>
            w_reg_en    <= '1';
        when S6 =>
            c_reg_en    <= '1';
        when S6_1 =>
            mux_en      <= '1';
            SEL         <= '0';
            o_mem_we    <= '1';
            o_mem_en    <= '1';
        when S7 =>
            w_last_en   <= '1';
        when S8 =>
            add_plus    <= '1';
        when S9 =>
            --check
        when S10 =>
            mux_en      <= '1';
            SEL         <= '1';
            o_mem_we    <= '1';
            o_mem_en    <= '1';
        when S11 =>
            add_plus    <= '1';
            k_plus      <= '1';
        when DONE =>
            o_done      <= '1';
    end case;
end process ControlProcess;
```

2.2.4 W Register

Il W Register è responsabile della memorizzazione del valore W della sequenza e della verifica se esso è uguale a zero. Nel processo sequenziale, sincronizzato al clock (**i_clk**) e al reset (**i_rst**), il valore di **w_stored** viene azzerato in caso di reset, mentre, se il segnale di abilitazione (**w_reg_en**) è attivo, viene aggiornato con il dato in ingresso (**i_mem_data**). Il processo combinatorio analizza il valore memorizzato: se **w_stored** è pari a zero, il segnale **is_zero** viene impostato a 1, altrimenti a 0. L'uscita **w_data** fornisce il valore aggiornato, garantendo che il registro supporti il sistema nell'elaborazione della sequenza.

```
w_register_1 : process(i_clk, i_rst)
-- Sequential process W_REGISTER
begin
    if i_rst = '1' then
        w_stored <= (others => '0');
    elsif rising_edge(i_clk) then
        if w_reg_en = '1' then
            w_stored <= i_mem_data;
        end if;
    end if;
end process w_register_1;
```

```
w_register_2 : process(w_stored)
-- Combinational process W_REGISTER
begin
    if w_stored = "00000000" then
        is_zero <= '1';
    else
        is_zero <= '0';
    end if;
end process w_register_2;

--Output W_REGITER
w_data <= w_stored;
```

2.2.5 Last W Valid Register

Il Last W Valid Register è il componente incaricato di memorizzare l'ultimo valore valido della sequenza W, ovvero l'ultimo valore diverso da zero, e di fornirlo come uscita al multiplexer. Attraverso un processo sequenziale sincronizzato al clock (**i_clk**) e al reset (**i_rst**), il registro inizializza il valore **last_w_stored** a zero quando il reset è attivo. Durante il normale funzionamento, se il segnale di abilitazione (**w_last_en**) è alto, il registro aggiorna il valore memorizzato con quello proveniente dal W Register. Se il segnale di reset secondario (**end_rst**) è attivo, il registro viene nuovamente azzerato, assicurando la corretta inizializzazione per una nuova elaborazione. In assenza di abilitazione, il valore memorizzato rimane invariato. L'uscita **w_last** fornisce il valore salvato, permettendo al sistema di disporre sempre dell'ultimo dato valido per la gestione della sequenza.

```
last_reg : process(i_clk, i_rst)
    -- Sequential process LAST_W
begin
    if i_rst = '1' then
        last_w_stored <= (others => '0');
    elsif rising_edge(i_clk) then
        if end_rst = '1' then
            last_w_stored <= (others => '0');
        else
            if w_last_en = '1' then
                last_w_stored <= w_data;
            else
                last_w_stored <= last_w_stored;
            end if;
        end if;
    end if;
end process last_reg;

--Output LAST_W
w_last <= last_w_stored;
```

2.2.6 C Register

Il C Register è il componente che assegna e gestisce il valore di "credibilità" associato alla sequenza da elaborare, fornendo il risultato al multiplexer in uscita. Tramite un processo sequenziale sincronizzato al clock (**i_clk**) e al reset (**i_rst**), il registro inizializza il valore **c_stored** a zero in caso di reset. Durante il funzionamento, se il segnale di abilitazione (**c_reg_en**) è attivo, il comportamento dipende dal valore del segnale **is_zero**: se **is_zero** è basso (0), il registro viene impostato a 31; in caso contrario, e se il valore di **c_stored** è diverso da zero, viene decrementato di 1 (fino a un minimo di 0). L'uscita **c_data** fornisce il valore di credibilità aggiornato al multiplexer.

```
c_reg : process(i_clk, i_rst)
-- Sequential process C_REGISTER
begin
    if i_rst = '1' then
        c_stored <= (others => '0');
    elsif rising_edge(i_clk) then
        if end_rst = '1' then
            c_stored <= (others => '0');
        else
            if c_reg_en = '1' then
                if is_zero = '0' then
                    c_stored <= "00011111";  -- Set to 31
                elsif c_stored /= "00000000" then
                    c_stored <= std_logic_vector(unsigned(c_stored) - 1);
                end if;
             end if;
        end if;
    end if;
end process c_reg;

--Output
c_data <= c_stored;
```

2.2.7 Multiplexer

Il Multiplexer (MUX) è il componente responsabile dell'emissione del dato finale in uscita verso la memoria (**o_mem_data**). La sua operazione è regolata dalla FSM, che seleziona quale ingresso utilizzare tramite il segnale SEL e abilita il funzionamento del MUX attraverso il segnale **mux_en**. Gli ingressi provengono dal C Register e dal Last W Valid Register. In base al valore di SEL, il MUX trasmette in uscita **w_last** o **c_data**, garantendo che il dato corretto sia inviato alla memoria durante l'elaborazione della sequenza.

```
process(mux_en, SEL, c_data, w_last)
-- Combinational process MUX
begin
    o_mem_data <= (others => '0');  --Default value

    if mux_en = '1' then
        case SEL is
            when '0' => o_mem_data <= w_last;
            when '1' => o_mem_data <= c_data;
            when others => o_mem_data <= (others => 'X');
        end case;
     end if;
end process;
```

3 Risultati sperimentali

3.1 Risultati della Sintesi

Il componente progettato è completamente sintetizzabile e correttamente simulabile in post-sintesi, confermando la conformità alle specifiche funzionali. Di seguito vengono riportati i risultati relativi al Timing Report e all'Utilization Report, generati attraverso l'ambiente di sviluppo Vivado.

3.1.1 Report Utilization

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	61	0	134600	0.05
LUT as Logic	61	0	134600	0.05
LUT as Memory	0	0	46200	0.00
Slice Registers	64	0	269200	0.02
Register as Flip Flop	64	0	269200	0.02
Register as Latch	0	0	269200	0.00
F7 Muxes	0	0	67300	0.00
F8 Muxes	0	0	33650	0.00

Il report mostra l'assenza di latch, con tutti i registri implementati come flip-flop, confermando la correttezza delle scelte progettuali volte a evitare latch indesiderati.

3.1.2 Report timing

```
Slack (MET) :             16.259ns  (required time - arrival time)
  Path Group:             clock
  Path Type:              Setup (Max at Slow Process Corner)
  Requirement:            20.000ns  (clock rise@20.000ns - clock rise@0.000ns)
  Data Path Delay:        3.590ns  (logic 0.999ns (27.827%)  route 2.591ns (72.173%))
  Logic Levels:           3  (LUT4=2 LUT5=1)
```

Il report conferma che il requisito di timing è rispettato (**MET**) con uno slack positivo di 16,259 ns, evidenziando che il **Data Path Delay** è di 3,590 ns, ben al di sotto del **Requirement** di 20,000 ns, garantendo il corretto funzionamento nei vincoli temporali definiti.

3.2 Simulazioni

Sono state effettuate diverse simulazioni, oltre a quelle assegnate, con l'obiettivo di esplorare i corner case e verificare il funzionamento del componente sia in pre- che in post-sintesi. L'intento è stato quello di coprire il maggior numero possibile di scenari limite per garantire l'affidabilità e la robustezza del design.

3.2.1 Test bench 1

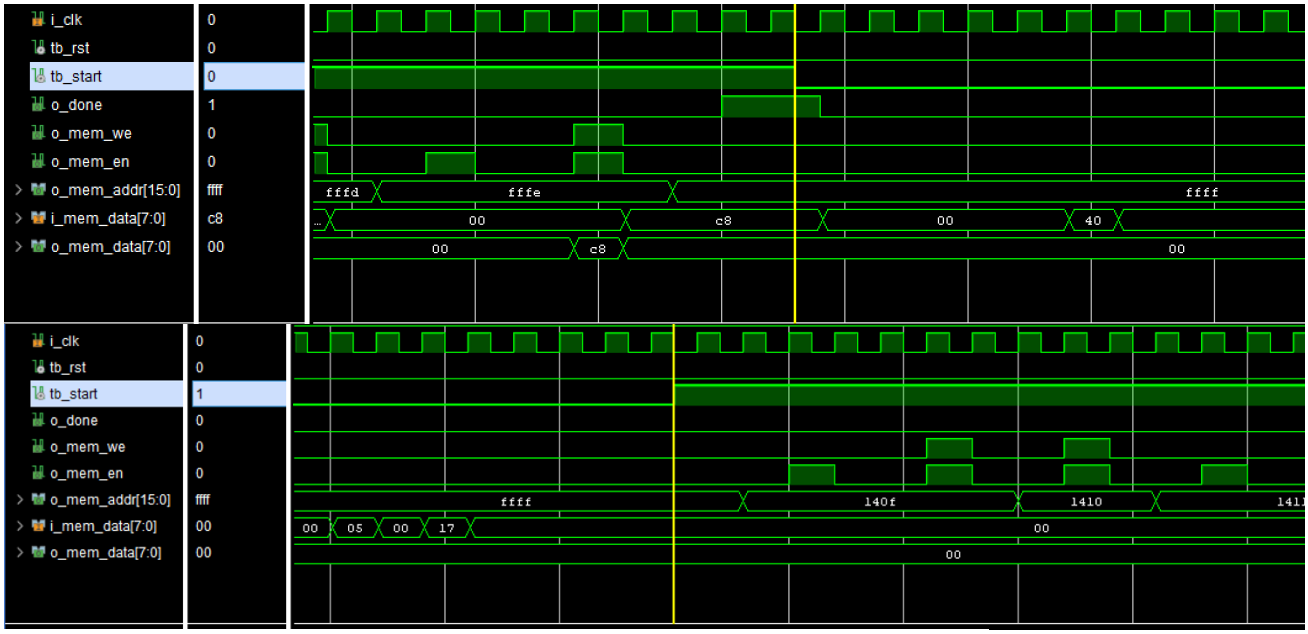


Figura 4: Simulazione di due sequenze in assenza di reset

Il test bench mostra il funzionamento del componente nell'elaborazione di due sequenze consecutive senza l'intervento del segnale di reset tra esse. La prima immagine illustra la corretta elaborazione della prima sequenza, con i segnali di controllo e gli indirizzi aggiornati coerentemente. La seconda immagine rappresenta l'inizio della seconda elaborazione, avviata con il segnale di start. La continuità tra le due sequenze conferma che il componente gestisce correttamente la transizione senza richiedere un reset, garantendo la funzionalità prevista. Le immagini sono state separate per migliorare la visibilità dei segnali.

3.2.2 Test bench 2

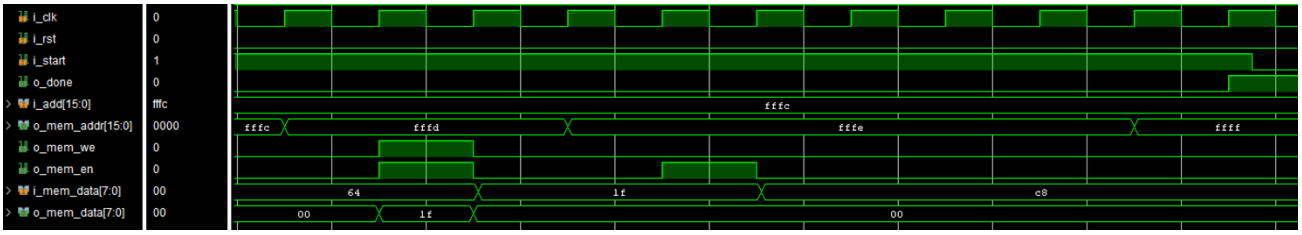


Figura 5: Simulazione del limite massimo degli indirizzi (ffff)

Il test bench conferma il corretto funzionamento del modulo nella gestione dell'indirizzo massimo (FFFF). Gli indirizzi in uscita (**o_mem_addr**) avanzano regolarmente fino al valore massimo senza causare overflow, mentre i segnali di controllo della memoria (**o_mem_we** e **o_mem_en**) si attivano in modo coerente con le operazioni previste. Al raggiungimento del limite, il segnale **o_done** si attiva correttamente, indicando la conclusione dell'elaborazione e confermando che il modulo gestisce in modo affidabile questa condizione limite.

3.2.3 Test bench 3

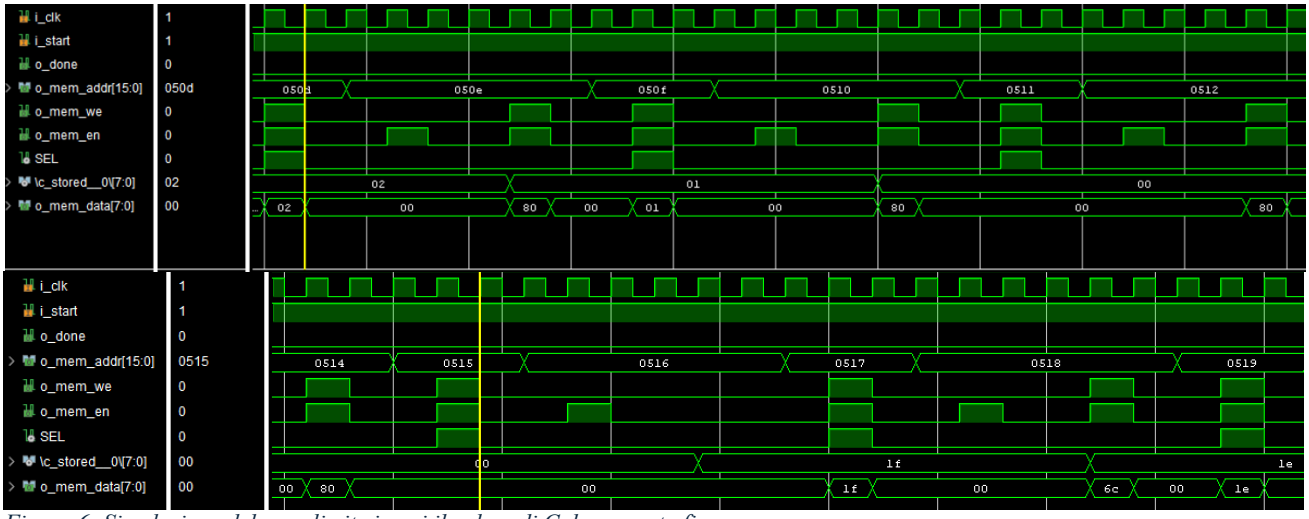


Figura 6: Simulazione del caso limite in cui il valore di C decrementa fino a zero

Il test bench rappresenta il caso limite in cui il valore di credibilità (**c_stored**) raggiunge zero durante l'elaborazione. Si osserva il corretto funzionamento del modulo, con il segnale **c_stored** che decrementa progressivamente fino a 00, dimostrando la corretta gestione dell'operazione di decremento del valore di credibilità. Inoltre, il valore di credibilità viene correttamente resettato a 31 non appena viene rilevata una parola W diversa da zero, confermando la conformità del comportamento del modulo alle specifiche progettuali.

4 Conclusioni

Il componente progettato è stato sviluppato seguendo i criteri richiesti dalle specifiche, garantendo il rispetto dei vincoli di temporizzazione, clock e sintesi. Fin da subito è stato adottato l'approccio suggerito dal docente, iniziando con la progettazione della rete su carta, seguita da un'analisi dettagliata di ogni componente ideato, fino alla stesura del codice in modo modulare, per renderlo più comprensibile e semplice da gestire. Sono state eseguite simulazioni approfondite per verificare il corretto funzionamento in pre- e post-sintesi, con particolare attenzione ai casi limite e alle condizioni critiche. I test bench utilizzati comprendono sia quelli forniti dai docenti sia quelli creati autonomamente, con l'obiettivo di coprire il maggior numero di scenari possibili.