Esame Bruno

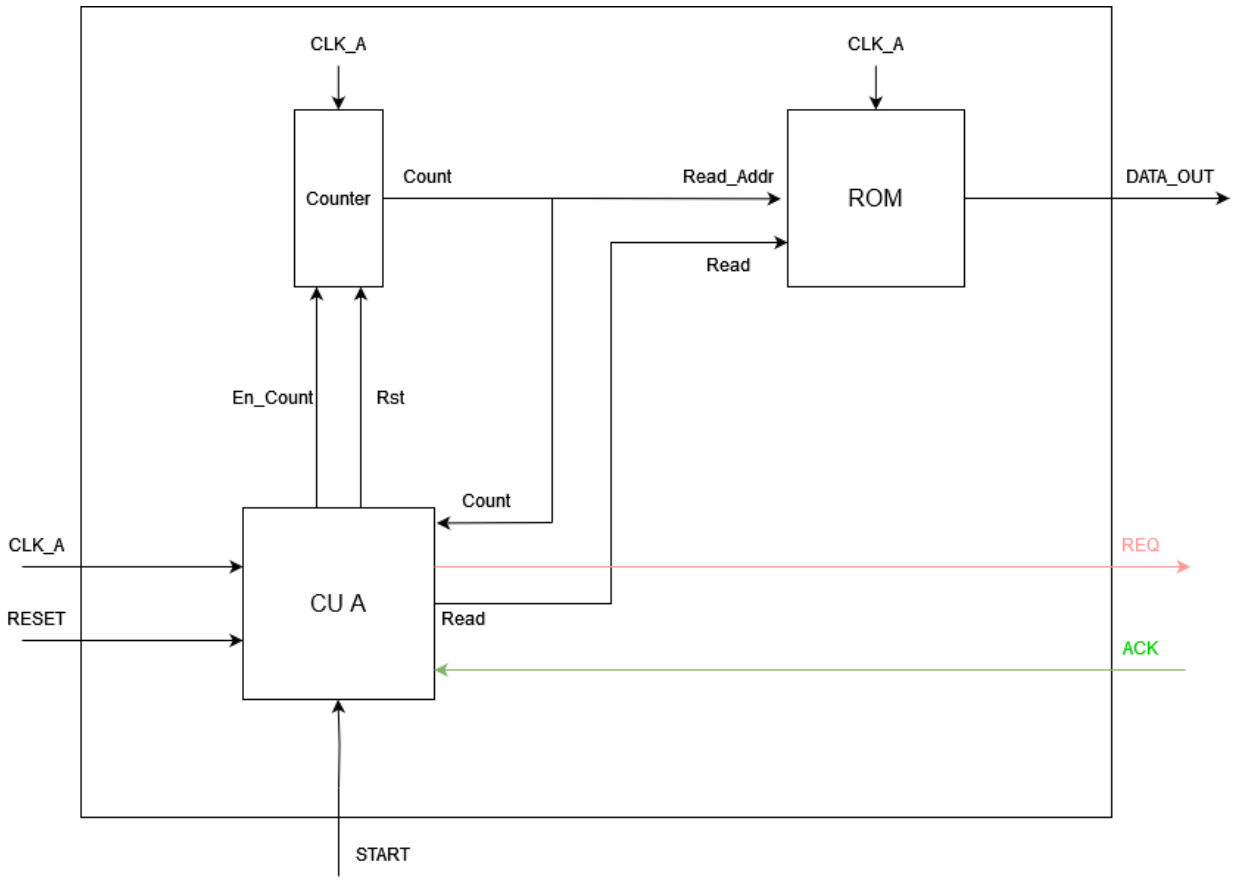
Traccia
Progetto e Architettura
    Nodo A
        CU A
    Nodo B
        CU B
Simulazione

Traccia

> Progettare, implementare in VHDL e simulare la seguente architettura. Un sistema è composto da 2 nodi, A e B. Il sistema A contiene una memoria ROM di N=8 locazioni da 8 bit ciascuna, da cui preleva un valore alla volta che invia a B mediante handshaking completo. B acquisisce la stringa e calcola il numero di bit alti al suo interno mediante una macchina M realizzata secondo un approccio strutturale. Il valore in uscita da M viene memorizzato da B in una memoria MEM insieme alla stringa di partenza solo se è dispari, altrimenti non viene salvato. Si implementi il sistema impiegando un approccio strutturale per l'interconnessione dei componenti e utilizzando per B un'unità di controllo microprogrammata.

Progetto e Architettura

Nodo A



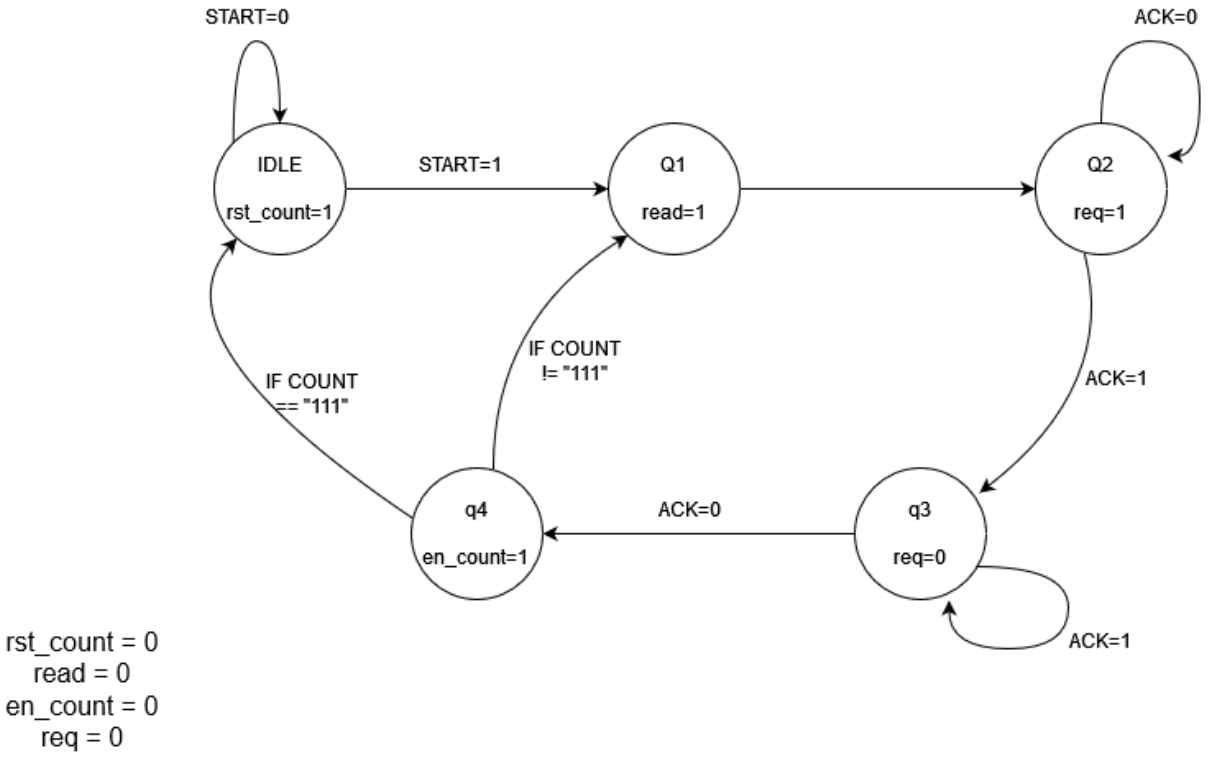
Architettura Nodo A

Questo nodo è stato realizzato in maniera strutturale interconnettendo 3 blocchi:

- ***Contatore (mod 8)*:** Il contatore, sviluppato con un approccio comportamentale, svolge un ruolo cruciale nel determinare quale delle 8 locazioni della ROM deve essere letta, fornendo quindi il `read_addr`. Inoltre, il conteggio viene trasmesso all'unità di controllo per decidere se continuare a inviare dati o se si è raggiunto il termine (`count="111"`) della scansione della ROM.
  - Questo contatore incrementa il suo valore ad ogni fronte di salita del clock quando il segnale di abilitazione (`En_count`) è attivo, e ritorna al suo stato iniziale quando il segnale di reset (`Rst`) è attivo.
- ***ROM:*** Questa memoria viene inizializzata con 8 valori esadecimali.
  - Il processo verifica se il segnale di lettura è alto ad ogni fronte di salita del clock. Se lo è, assegna alla `data_out` il valore della ROM all'indirizzo specificato da `addr`.
- ***Unità di Controllo:*** L'Unità di Controllo svolge un ruolo fondamentale nel coordinare i **segnali di controllo destinati alla ROM (`read`) e al contatore (`En_count`)**. Accoglie i segnali di `start` e `reset` provenienti dall'esterno, insieme al segnale di clock. Inoltre,

gestisce efficacemente la **comunicazione di controllo con il nodo B**, rispettando il protocollo di handshaking.

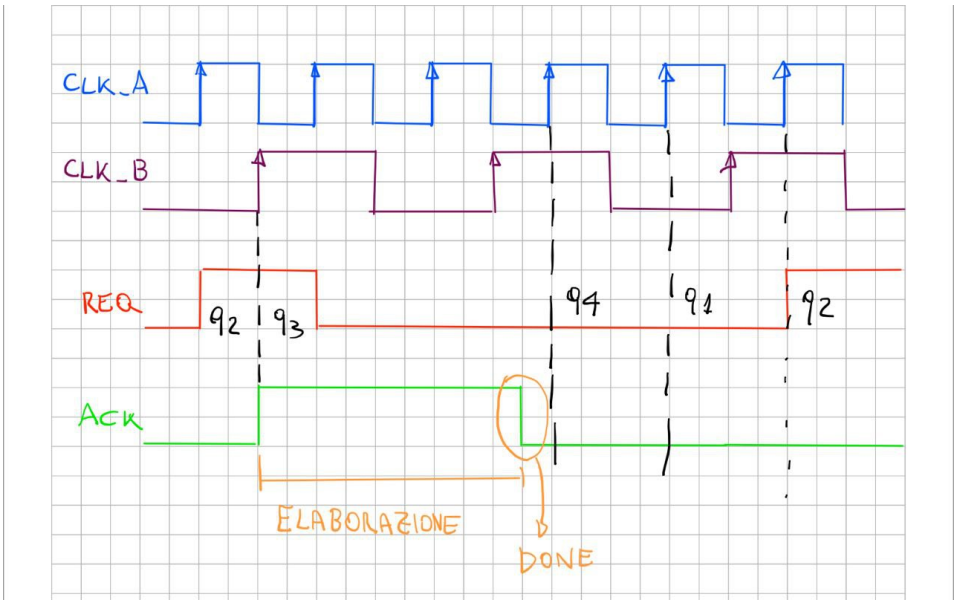
CU A



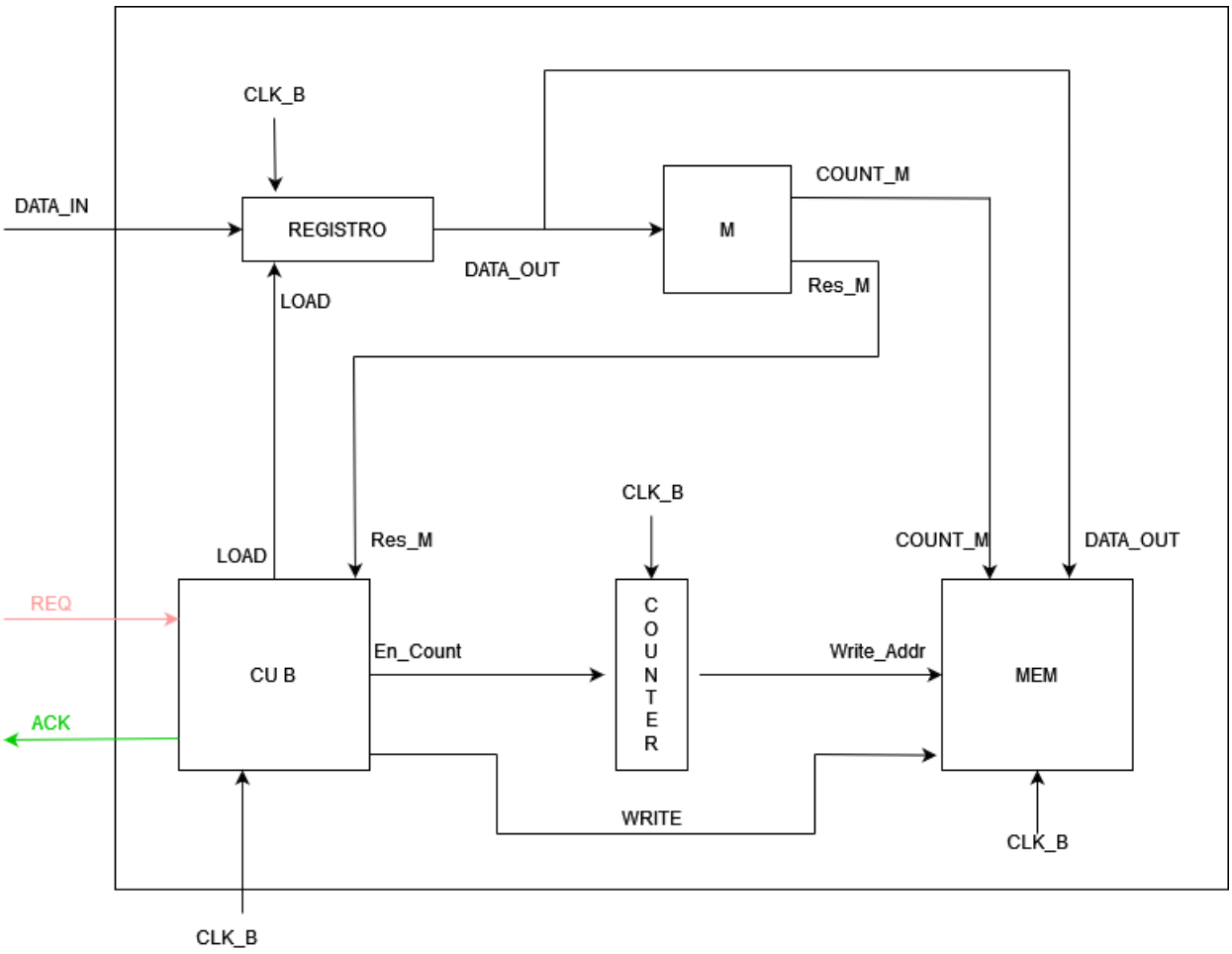
Quando `START=1`, dallo stato IDLE si passa allo stato ***q1:***

- ***q1:*** In questa fase, viene inviato il segnale di **lettura** alla ROM, che a sua volta trasmette il **dato sulla linea di comunicazione** connessa al nodo B.
- ***q2:*** Una volta preparato il dato, A emette un segnale di **REQ** a B per richiedere la trasmissione del dato. Il sistema rimane in questo stato fino a quando non riceve la conferma di disponibilità da B (`ack=1`).
- ***q3:*** Al ricevimento del segnale di ack, il segnale **REQ viene disattivato**, e il sistema attende la conclusione dell'elaborazione da parte di B, che coincide con la disattivazione di ACK.
  - Questo approccio consente di risparmiare l'utilizzo di un ulteriore cavo DONE, sfruttando invece il fronte di discesa di ACK.
- ***q4***: Al termine dell'elaborazione da parte di B, viene attivato il conteggio per scandire la posizione successiva della ROM e preparare un nuovo dato. Tuttavia, se il conteggio

raggiunge 8, indicando che tutte le 8 locazioni della ROM sono state scandite, il sistema ritorna allo stato IDLE.



Nodo B



Questo nodo è stato realizzato in maniera strutturale interconnettendo 5 blocchi:

- ***Registro***: Questo registro è un buffer che serve ad immagazzinare la stringa inviata da A.

  Il processo verifica prima se il segnale di reset è alto. Se lo è, azzera il registro (`stato`). Se non lo è, verifica **se il segnale di scrittura è alto. Se lo è, scrive `data_in` nel registro.**

  Infine, il segnale di uscita `data_out` viene assegnato il valore corrente del registro (`stato`).

- ***Counter (mod 8)***: stesso contatore del nodo A con la differenza che in questo caso il conteggio in uscita serve a specificare il write address della MEM.

- ***M:*** Questo componente è stata realizzata in maniera strutturale interconnettendo 2 blocchi:

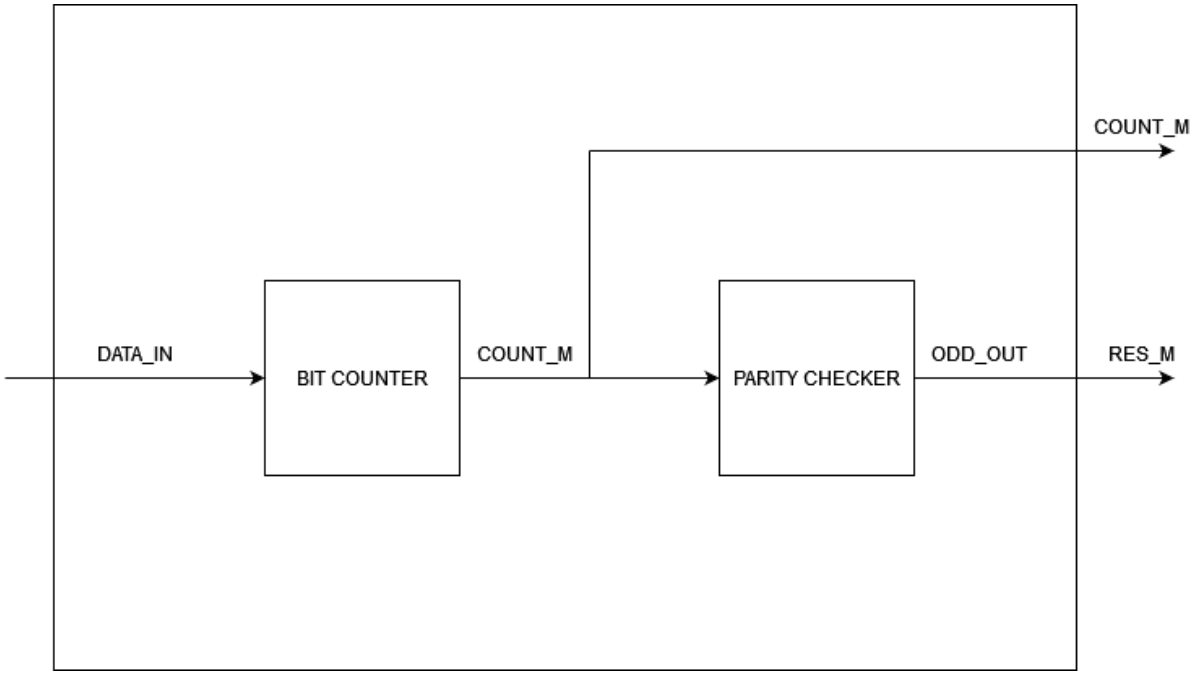
  1. **Bit Counter:** questo componente prende in ingresso la stringa in uscita dal registro buffer e conta quanti bit sono alti. L'uscita del contatore dunque è un intero che rappresenta il numero di bit alti nei dati in ingresso.

In questo caso, c'è un unico processo che viene eseguito ogni volta che `data_in` cambia. Il processo inizia impostando una variabile temporanea `temp_count` a zero. Poi, per ogni bit in `data_in`, se il bit è '1', incrementa `temp_count`. Alla fine del processo, assegna il valore di `temp_count` a `count_out`.

Questo conteggio poi verrà fornito sia alla MEM che al blocco Parity Checker.

2. **Parity Checker:** questo componente prende in ingresso il conteggio del Bit Counter e in uscita fornisce un segnale logico `odd_out`, che indica se il numero in ingresso è pari o dispari.

   In questo caso, c'è un unico processo che viene eseguito ogni volta che `count_in` cambia. Il processo verifica se `count_in` è dispari o pari. Se è dispari, assegna '1' a `odd_out`. Se è pari, assegna '0' a `odd_out`.



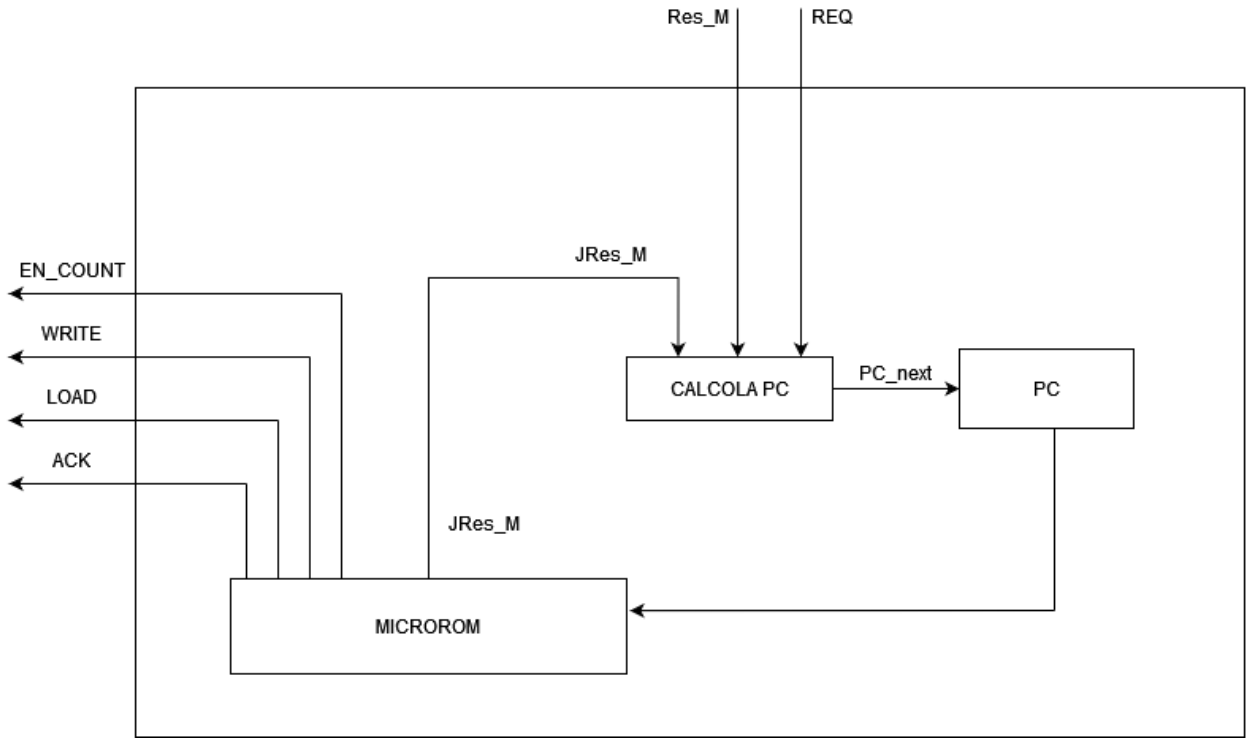
- ***MEM:*** Questa memoria presenta 8 locazioni da 11 bit. Questo perchè in input prende la concatenazione tra la stringa inviata da A (8 bit) e il numero di bit alti presenti nella stringa (3 bit).

  Il processo verifica se il segnale di abilitazione alla scrittura (**Write)** è alto ad ogni fronte di salita del clock. Se lo è, scrive `input` nella memoria all'indirizzo specificato da `write_addr`.

- **Control Unit:** In questo contesto, l'unità di controllo è stata realizzata utilizzando un approccio microprogrammato. Questo comporta l'uso di una **microRom**, che serve come deposito per tutte le microistruzioni. Queste microistruzioni, che corrispondono

agli stati dell'unità di controllo, forniscono i segnali di controllo necessari per ogni singolo passaggio del processo di elaborazione. All'interno dell'unità di controllo, troviamo un registro **Program Counter**. Questo registro svolge un ruolo fondamentale nell'indirizzare la memoria, permettendo la selezione della microistruzione corrente

CU B



MicroRom:

Questo componente serve come deposito per le microistruzioni che guidano il funzionamento del sistema. Ogni microistruzione è un insieme di segnali di controllo che determinano le operazioni da eseguire in un dato passo di elaborazione.

Il **PC** è un indicatore che tiene traccia della posizione corrente all'interno della memoria delle microistruzioni. Ad ogni ciclo di clock, il PC seleziona una microistruzione dalla memoria.

Le **microistruzioni stesse sono rappresentate come record**. In questo caso, ogni microistruzione contiene un insieme di segnali di controllo e il prossimo valore del PC.

```vhdl
entity MicroRom is
    port(
        PC: in unsigned(1 downto 0);
        PC_next: out unsigned(1 downto 0);
        JRes_M: out std_logic;
        write: out std_logic;
        load: out std_logic;
        ack: out std_logic;
        en_count: out std_logic
    );
end MicroRom;

architecture synth of MicroRom is

    type Controllo_type is record --struttura delle microistruzioni
        PC_next:  unsigned(1 downto 0);
        JRes_M:   std_logic;
        write:    std_logic;
        load:     std_logic;
        en_count: std_logic;
        ack:      std_logic;
    end record;
```

Quando il sistema è in funzione, la MicroRom legge la microistruzione corrente dalla memoria, come indicato dal PC. Poi, fornisce i segnali di controllo alla restante parte del sistema e aggiorna il PC al suo prossimo valore.

```
CONSTANT idle: Controllo_type := (
        PC_next => "00",
        JRes_M => '0',
        write => '0',
        load => '0',
        en_count => '0',
        ack => '0'
);
CONSTANT q1: Controllo_type := ( --load del registro buffer e invio ack
        PC_next => "10",
        JRes_M => '0',
        write => '0',
        load => '1',
        en_count => '0',
        ack => '1'
);
CONSTANT q2: Controllo_type := ( --controllo se il conteggio è pari o dispari
        PC_next => "11",
        JRes_M => '1',
        write => '0',
        load => '0',
        en_count => '0',
        ack => '1'  --continuo a tenere alzato ack
);
CONSTANT q3: Controllo_type := (--nel caso in cui il conteggio sia dispari
        PC_next => "00",
        JRes_M => '0',
        write => '1', --salvo il valore concatenato in MEM
        load => '0',
        en_count => '1', --passo alla prossima locazione di MEM
        ack => '1' --mantengo alto ack
);
```

```
    type ROM_TYPE is array(0 to 3) of Controllo_type;
    CONSTANT ROM: ROM_type := (
    --Contiene le 4 microistrizioni (corrispondono agli stati)
        0 => idle,
        1 => q1,
        2 => q2,
        3 => q3
    );

    signal Controllo: Controllo_type;

begin

    Controllo <= ROM(conv_integer(PC));

        PC_next <= Controllo.PC_next;
        JRes_M <= Controllo.JRes_M;
        write <= Controllo.write;
        load <= Controllo.load;
        en_count <= Controllo.en_count;
        ack <= Controllo.ack;

end synth;
```

Come abbiamo visto **le microistruzioni corrispondono agli stati** della FSM del controllo cablato, infatti ogni microistruzione deve contenere:

- I **segnali** da settare verso l'esterno (load, write, en_count, ack)
- **L'indirizzo della prossima microistruzione** (PC_next)

Per gestire il flusso di controllo in modo più flessibile, è necessario un meccanismo che permetta di caricare una microistruzione diversa da quella indicata da `PC_next`. A tale scopo, ho introdotto un **segnale interno al controllore (`JRes_M`)**, che viene associato a ciascuna microistruzione.

In particolare, quando ci troviamo nello stato **q2**, dopo aver ottenuto gli output dalla macchina M, dobbiamo decidere il prossimo stato in base al valore di `Res_M`. Se `Res_M` è dispari, dovremmo passare allo stato successivo **(q3)** e abilitare la scrittura nella MEM. Al contrario, se `Res_M` è pari, dovremmo tornare allo stato di IDLE.

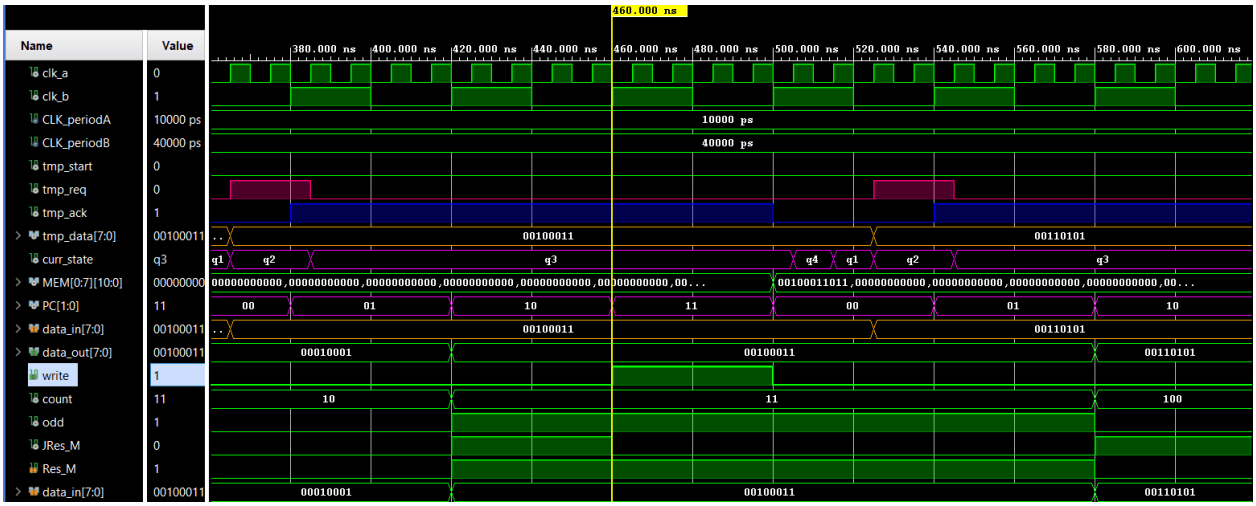
Per implementare questa logica, quando siamo nello stato **q2**, alziamo il segnale `JRes_M`. In questo modo, invece di passare direttamente al `PC_next`, il program counter controlla prima il valore di `Res_M` e, se necessario, salta direttamente allo stato IDLE.

```
reg_PC: process(clk_B)
begin
    if rising_edge(clk_B) then
        if (req = '1') then
            PC <= "01";
        elsif (JRes_M = '0') then
            PC <= PC_next;
        else
            if (Res_M = '0') then
                PC <= "00";
            else
                PC <= PC_next;
            end if;
        end if;
    end if;
end process reg_PC;
```

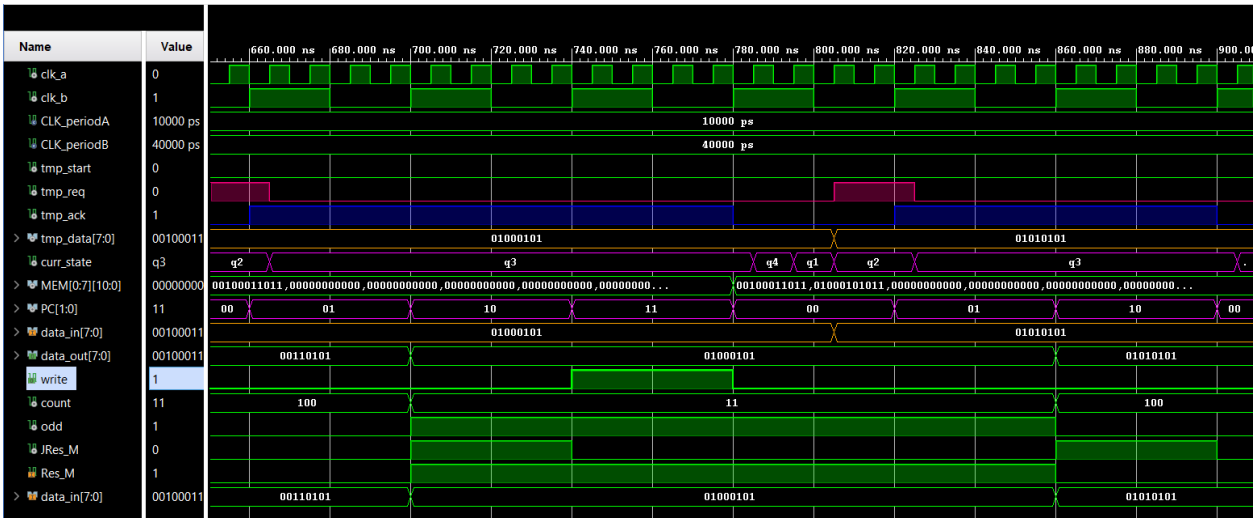
Process nell'unità di controllo per l'aggiornamento del Program Counter

Simulazione

I primi due valori inviati hanno un numero di "1" pari, dunque non vengono salvati nella MEM.



Il terzo valore invece ha un numero di “1” dispari, per questo motivo viene salvato nella MEM la concatenazione tra la stringa originale ed il conteggio di “1”.



Altro esempio di scrittura.