

UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

SCUOLA POLITECNICA E DELLE SCIENZE DI BASE

DIPARTIMENTO DI INGEGNERIA ELETTRICA E TECNOLOGIE DELL'INFORMAZIONE

Corso di Laurea Magistrale in Ingegneria Informatica



ELABORATO DI ARCHITETTURA DEI SISTEMI DIGITALI

Prof.ssa Alessandra De Benedictis

a.a. 2023-24

Studenti:

Alberto Petillo M63001604

Benedetta Gaia Varriale M63001603

Raffaele Imperato M63001661

CAPITOLO 1: RETI COMBINATORIE ELEMENTARI	4
ESERCIZIO 1: MULTIPLEXER 16:1	4
<i>Esercizio 1.1.</i>	4
<i>Esercizio 1.2.</i>	9
<i>Esercizio 1.3.</i>	15
ESERCIZIO 2: SISTEMA ROM + M	19
<i>Esercizio 2.1.</i>	19
<i>Esercizio 2.2.</i>	25
ESERCIZIO 3: RICONOSCITORE DI SEQUENZE	26
<i>Esercizio 3.1.</i>	26
<i>Esercizio 3.2.</i>	33
ESERCIZIO 4: SHIFT REGISTER.....	37
ESERCIZIO 5: CRONOMETRO	45
<i>Esercizio 5.1.</i>	45
<i>Esercizio 5.2.</i>	53
<i>Esercizio 5.3.</i>	56
ESERCIZIO 6: SISTEMA DI LETTURA-ELABORAZIONE-SCRITTURA PO_PC.....	58
<i>Esercizio 6.1.</i>	58
<i>Esercizio 6.2.</i>	66
ESERCIZIO 7: MOLTIPLICATORE DI BOOTH.....	67
<i>Esercizio 7.1.</i>	67
<i>Esercizio 7.2.</i>	84
ESERCIZIO 8: COMUNICAZIONE CON HANDSHAKING	89
ESERCIZIO 9: PROCESSORE	105
ESERCIZIO 10: INTERFACCIA SERIALE.....	111
ESERCIZIO 11: SWITCH MULTISTADIO	120
APPENDICE	130
MULTIPLEXER 2:1.....	130
<i>Progetto e architettura.</i>	130
<i>Implementazione.</i>	130
DEMULTIPLEXER 1:2	131
<i>Progetto e architettura.</i>	131
<i>Implementazione.</i>	131
MULTIPLEXER 4:1.....	132
<i>Progetto e architettura.</i>	132
<i>Implementazione.</i>	133
CONTATORE MODULO N	133
ROM SEQUENZIALE	135
MEMORIA.....	137
BOTTON DEBOUNCER.....	137

Capitolo 1: reti combinatorie elementari

Esercizio 1: Multiplexer 16:1

Esercizio 1.1

Progettare, implementare in VHDL e testare mediante simulazione un **multiplexer indirizzabile 16:1**, utilizzando un approccio di progettazione per composizione a partire da **multiplexer 4:1**.

Progetto e architettura

Per la risoluzione dell'esercizio abbiamo utilizzato un approccio modulare, ovvero, abbiamo decomposto la macchina da implementare in componenti più piccoli per poi arrivare gradualmente all'implementazione finale del multiplexer indirizzabile 16:1.

Come richiesto dalla traccia, il multiplexer 16:1 è stato progettato tramite la composizione di 5 multiplexer 4:1.

A loro volta, i multiplexer 4:1 sono stati realizzati tramite la composizione di 3 multiplexer 2:1. All'interno dell'appendice è possibile consultare la progettazione e l'implementazione prima del multiplexer 2:1 e poi del multiplexer 4:1 ottenuto per composizione.

Il multiplexer 16:1 presenta 16 ingressi e 1 uscita. Anche in questo caso si tratta di un multiplexer *indirizzabile* per questo avremo ulteriori 4 ingressi di selezione attraverso i quali viene trasmessa un'uscita y . Quindi in definitiva abbiamo **20 ingressi e 1 uscita**. Lo schema è il seguente:

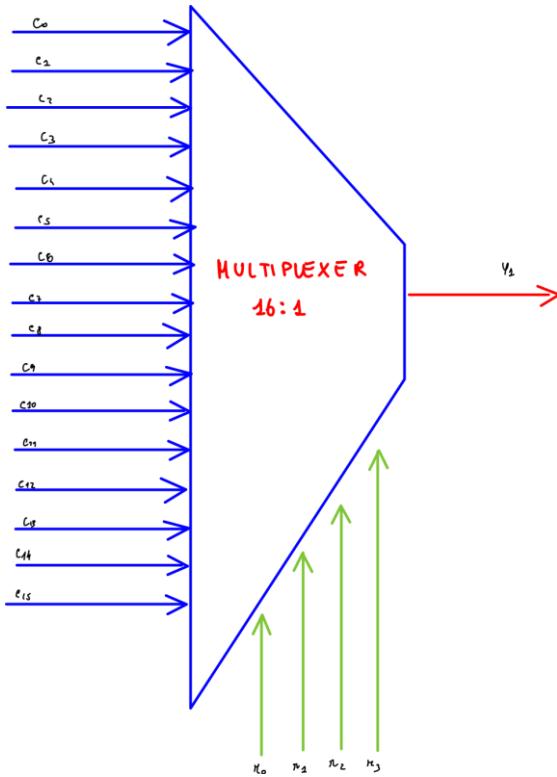


Figura 1 multiplexer 16:1

I 16 ingressi sono c_0, c_1, \dots, c_{15} , i 4 ingressi di selezione sono r_0, r_1, r_2, r_3 mentre l'uscita è y_1 .

Come già spiegato preliminarmente, per realizzare tramite composizione il multiplexer 16:1 vengono adoperati **5 multiplexer 4:1**. I primi 4 multiplexer 4:1 prenderanno in ingresso ciascuno 4 dei 16 ingressi del

multiplexer 16:1. Questi, tramite i primi 2 ingressi di selezione (r_0 ed r_1) produrranno ciascuno un'uscita in modo da avere in totale 4 segnali provenienti da altrettanti multiplexer 4:1 che saranno gli ingressi del **quinto** multiplexer 4:1, che, tramite i rimanenti ingressi di selezione r_2 ed r_3 produrrà l'uscita finale y .

Lo schema è mostrato nella pagina seguente:

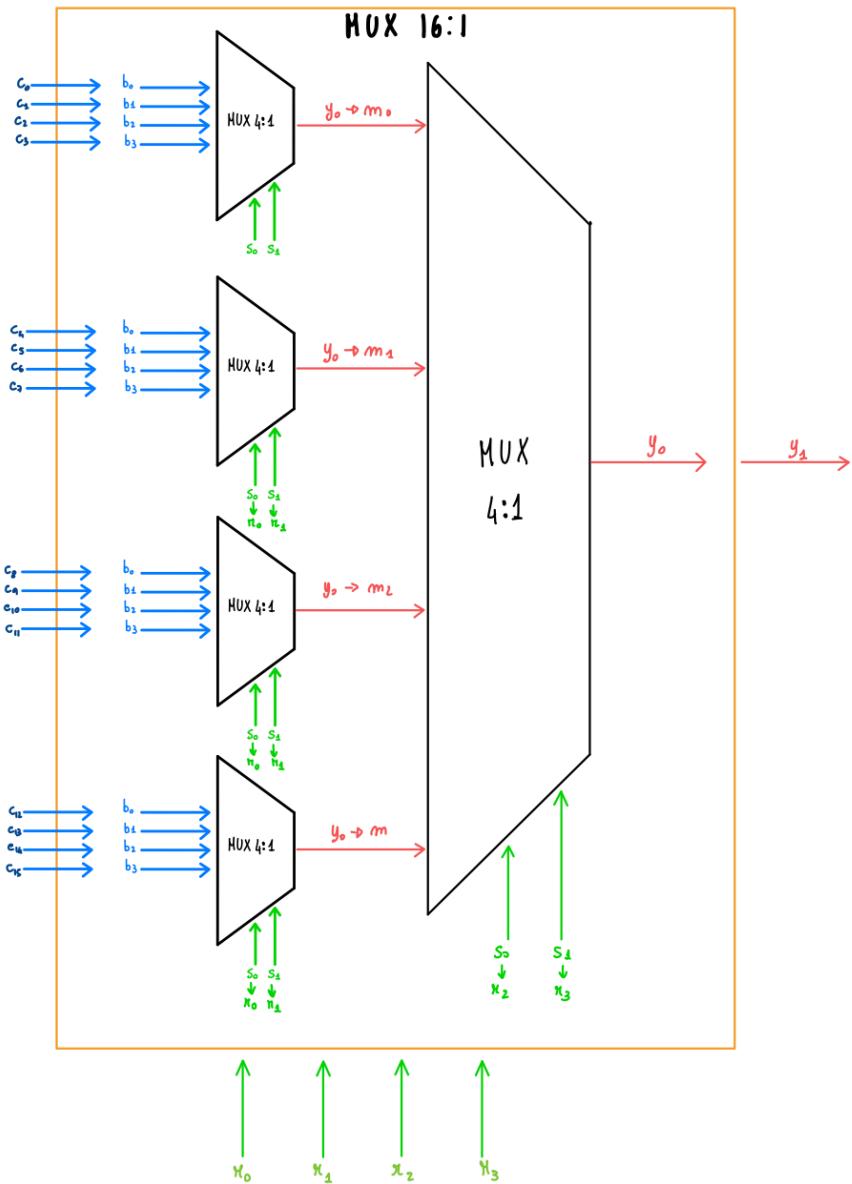


Figura 2.2 schema mux 16:1 per composizione di mux 4:1

Implementazione

In maniera quasi analoga a quanto fatto per il mux 4:1 abbiamo composto il mux 16:1. (si veda appendice per mux 4:1)

Innanzitutto, abbiamo definito la nostra entity provvista di 20 ingressi dichiarati come *std_logic* e 1 uscita anch'essa *std_logic*:

```

entity mux_16_1 is
    port(  c0 : in STD_LOGIC;
            c1 : in STD_LOGIC;
            c2 : in STD_LOGIC;
            c3 : in STD_LOGIC;
            c4 : in STD_LOGIC;
            c5 : in STD_LOGIC;
            c6 : in STD_LOGIC;
            c7 : in STD_LOGIC;
            c8 : in STD_LOGIC;
            c9 : in STD_LOGIC;
            c10 : in STD_LOGIC;
            c11 : in STD_LOGIC;
            c12 : in STD_LOGIC;
            c13 : in STD_LOGIC;
            c14 : in STD_LOGIC;
            c15 : in STD_LOGIC;
            r0 : in STD_LOGIC;
            r1 : in STD_LOGIC;
            r2 : in STD_LOGIC;
            r3 : in STD_LOGIC;
            y1 : out std_logic
        );
end mux_16_1;

```

Figura 1.3 entity mux_16_1

L'approccio usato è stato di tipo *strutturale* e abbiamo sfruttato il mux 4:1 che è stato istanziato come *component*.

```

architecture structural of mux_16_1 is
    signal m0 : std_logic := '0';
    signal m1 : std_logic := '0';
    signal m2 : std_logic := '0';
    signal m3 : std_logic := '0';

    component mux_4_1
        port(  b0 : in STD_LOGIC;
                b1 : in STD_LOGIC;
                b2 : in STD_LOGIC;
                b3 : in STD_LOGIC;
                s0 : in STD_LOGIC;
                s1 : in STD_LOGIC;
                y0 : out STD_LOGIC
            );
    end component;

    begin
        muxel0: mux_4_1
            Port map(
                b0 => c0,
                b1 => c1,
                b2 => c2,
                b3 => c3,
                s0 => r0,
                s1 => r1,
                y0 => m0
            );
        muxel1: mux_4_1
            Port map(
                b0 => c4,
                b1 => c5,
                b2 => c6,
                b3 => c7,
                s0 => r0,
                s1 => r1,
                y0 => m1
            );
        muxel2: mux_4_1
            Port map(
                b0 => c8,
                b1 => c9,
                b2 => c10,
                b3 => c11,
                s0 => r0,
                s1 => r1,
                y0 => m2
            );
        muxel3: mux_4_1
            Port map(
                b0 => c12,
                b1 => c13,
                b2 => c14,
                b3 => c15,
                s0 => r0,
                s1 => r1,
                y0 => m3
            );
        muxel4: mux_4_1
            Port map(
                b0 => m0,
                b1 => m1,
                b2 => m2,
                b3 => m3,
                s0 => r2,
                s1 => r3,
                y0 => y1
            );
    end structural;

```

Figura 1.4 descrizione strutturale mux 16:1

Come prima cosa all'interno dell'architettura abbiamo dichiarato 4 segnali: m0, m1, m2, m3. Questi 4 segnali rappresentano le uscite dei primi 4 multiplexer all'interno dell'architettura. Utilizziamo questi segnali per effettuare i collegamenti tra i vari componenti.

Successivamente dichiariamo il componente utilizzato ovvero il mux_4_1.

Dopo la dichiarazione istanziamo 4 multiplexer tramite port mapping.

In questo codice vengono create le istanze dei multiplexer 4:1 *mplex0*, *mplex1*, *mplex2* e *mplex3*. Ogni istanza è collegata a un gruppo di 4 ingressi e controllate dai segnali di selezione r0 e r1. I segnali interni m0, m1, m2 e m3 vengono utilizzati per immagazzinare i risultati dei quattro multiplexer 4:1.

L'istanza *mplex4* rappresenta il multiplexer 4:1 finale che seleziona l'uscita dei quattro multiplexer precedenti in base ai segnali di selezione r2 e r3 restituendo il risultato finale su y1.

Simulazione

Per effettuare la simulazione il primo passo da compiere è la stesura del **testbench**. Prima di discutere quanto fatto è bene osservare il codice in figura:

```
entity mux_16_1_tb is
end mux_16_1_tb;

architecture behavioral of mux_16_1_tb is

component mux_16_1
port(
    c0 : in STD_LOGIC;
    c1 : in STD_LOGIC;
    c2 : in STD_LOGIC;
    c3 : in STD_LOGIC;
    c4 : in STD_LOGIC;
    c5 : in STD_LOGIC;
    c6 : in STD_LOGIC;
    c7 : in STD_LOGIC;
    c8 : in STD_LOGIC;
    c9 : in STD_LOGIC;
    c10 : in STD_LOGIC;
    c11 : in STD_LOGIC;
    c12 : in STD_LOGIC;
    c13 : in STD_LOGIC;
    c14 : in STD_LOGIC;
    c15 : in STD_LOGIC;
    r0 : in STD_LOGIC;
    r1 : in STD_LOGIC;
    r2 : in STD_LOGIC;
    r3 : in STD_LOGIC;
    y1 : out std_logic
);
end component;

stim_proc : process
begin
    wait for 10 ns;
    input <= "1010101010101010";
    wait for 10 ns;
    control <= "0000";
    wait for 5 ns;
    control <= "0001";
    wait for 5 ns;
    control <= "0010";
    wait for 5 ns;
    control <= "0100";
    wait for 5 ns;
    control <= "1000";
    assert output = '0'
        report "errore0"
        severity failure;
    wait;
end process;
end;
```

```
signal input : std_logic_vector (15 downto 0) := (others => 'U');
signal control: std_logic_vector (3 downto 0) := (others => 'U');
signal output: std_logic:= 'U';

begin
    uut : mux_16_1
    Port map(
        c0 => input(0),
        c1 => input(1),
        c2 => input(2),
        c3 => input(3),
        c4 => input(4),
        c5 => input(5),
        c6 => input(6),
        c7 => input(7),
        c8 => input(8),
        c9 => input(9),
        c10 => input(10),
        c11 => input(11),
        c12 => input(12),
        c13 => input(13),
        c14 => input(14),
        c15 => input(15),
        r0 => control(0),
        r1 => control(1),
        r2 => control(2),
        r3 => control(3),
        y1 => output
    );

```

La prima cosa che abbiamo fatto è stata dichiarare una entity. Si nota che il corpo della entity è vuoto, questo perché non rappresenta un oggetto da realizzare ma ci serve solo per effettuare la simulazione e verificare il

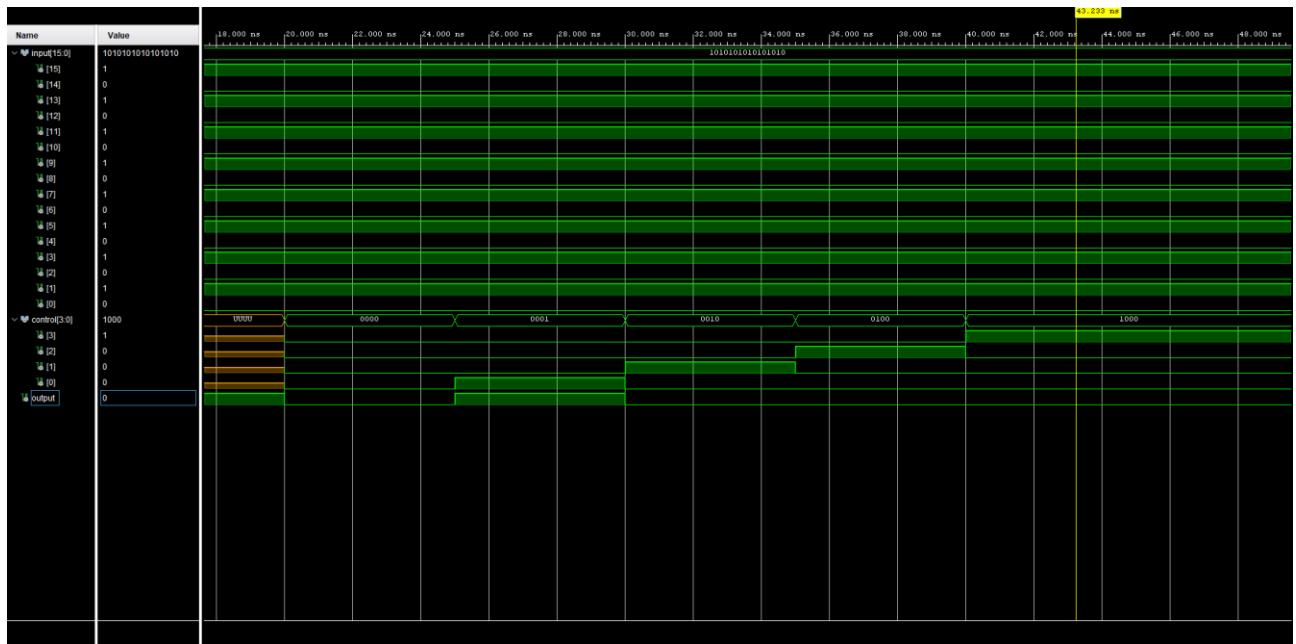
corretto funzionamento del sistema. Il testbench non ha segnali di ingresso e di uscita in quanto non può essere istanziato da nessun blocco ma istanzia al suo interno altri blocchi per effettuarne il test.

Visto che l'oggetto che vogliamo testare è il **multiplexer 16:1** lo istanziamo come **uut (unit under test)** e ne definiamo il port map collegando al mux 16:1 i segnali (*input* e *control*) che utilizzeremo per validare il funzionamento del nostro sistema. I segnali del testbench sono stati dichiarati prima dell'uut.

Il segnale *input* rappresenta gli ingressi del multiplexer, *control* rappresenta i segnali di controllo ed infine *output* rappresenta l'uscita del multiplexer.

Successivamente abbiamo definito un *process*, in particolare si tratta di uno **stim_process** (processo di stimolazione) che si occupa di fornire stimoli (input) al nostro componente e verificare che l'output sia quello atteso. Forniamo vari segnali di controllo e gli input vengono modificati nel tempo. Aspettiamo 10ns e diamo in ingresso il vettore “1010101010101010” e come vettore di selezione, dopo altri 5ns, diamo “0000” che in decimale corrisponde a 0. Con intervalli cadenzati ogni 5ns cambiamo la stringa di controllo. L'ultima stringa di controllo è “1000” che in decimale corrisponde ad 8. Ci aspettiamo quindi di vedere in uscita c8 che corrisponde a 0. Utilizzando un *assert* verifichiamo che l'output sia pari a 0, in caso contrario segnaliamo l'errore e arrestiamo il processo.

Lanciando la simulazione possiamo verificare quanto abbiamo appena spiegato.



Dal grafico possiamo notare come l'inizializzazione dell'input avviene correttamente, i valori sono coerenti con quanto scritto nel testbench. Possiamo fare un discorso analogo anche per il valore di control. Infatti, inizialmente è Undefined, dopo 20ns, esattamente come scriviamo nel codice, viene inizializzato a 0000. Il valore del segnale control cambia ogni 10ns esattamente come desiderato. In base alla variazione del segnale control anche il valore di output viene modificato. Infatti, avremo:

- Segnale di control pari a 0000 (0) -> l'output dovrà essere pari a *input[0]* ovvero 0;
- Segnale di control pari a 0001 (1) -> l'output dovrà essere pari a *input[1]* ovvero 1;
- Segnale di control pari a 0010 (2) -> l'output dovrà essere pari a *input[2]* ovvero 0;
- Segnale di control pari a 0100 (4) -> l'output dovrà essere pari a *input[4]* ovvero 0;
- Segnale di control pari a 1000 (1) -> l'output dovrà essere pari a *input[8]* ovvero 0;

Verifichiamo che i valori mostrati in output sono tutti corretti.

Esercizio 1.2

Utilizzando il componente sviluppato al punto precedente, progettare, implementare in VHDL e testare mediante simulazione una **rete di interconnessione a 16 sorgenti e 4 destinazioni**.

Progetto e architettura

Per la risoluzione dell'esercizio di progettazione della rete di interconnessione, abbiamo adottato un approccio modulare, suddividendo il problema in componenti più piccoli per semplificare l'implementazione complessiva. In particolare, la nostra implementazione coinvolge due componenti principali: un multiplexer 16:1 e un demultiplexer 1:4.

Il multiplexer 16:1 ha 16 ingressi, identificati come c_0, c_1, \dots, c_{15} , e 4 linee di selezione, indicate come r_0, r_1, r_2 e r_3 . L'uscita del multiplexer è denominata y_1 . Abbiamo adottato un approccio di composizione per realizzare il multiplexer 16:1 utilizzando un totale di 5 multiplexer 4:1.

In dettaglio, i primi 4 multiplexer 4:1 ricevono ciascuno 4 degli ingressi del multiplexer 16:1. Ogni multiplexer 4:1, a sua volta, utilizza due linee di selezione (r_0 e r_1) per generare un'uscita. Questo ci fornisce 4 segnali in uscita da questi primi 4 multiplexer 4:1. Tutti questi segnali diventano gli ingressi di un quinto multiplexer 4:1.

Il quinto multiplexer 4:1 utilizza le rimanenti due linee di selezione (r_2 e r_3) per selezionare uno dei segnali in ingresso dai primi quattro multiplexer 4:1. L'uscita di questo quinto multiplexer 4:1 è la nostra uscita finale y_1 , che rappresenta il risultato del multiplexer 16:1.

Il demultiplexer 1:4, a sua volta, riceve un singolo ingresso (in questo caso, y_1) e lo distribuisce su quattro uscite, controllate da due linee di selezione ($s_{_demux}(0)$ e $s_{_demux}(1)$).

Lo schema è il seguente:

RETE 16:4

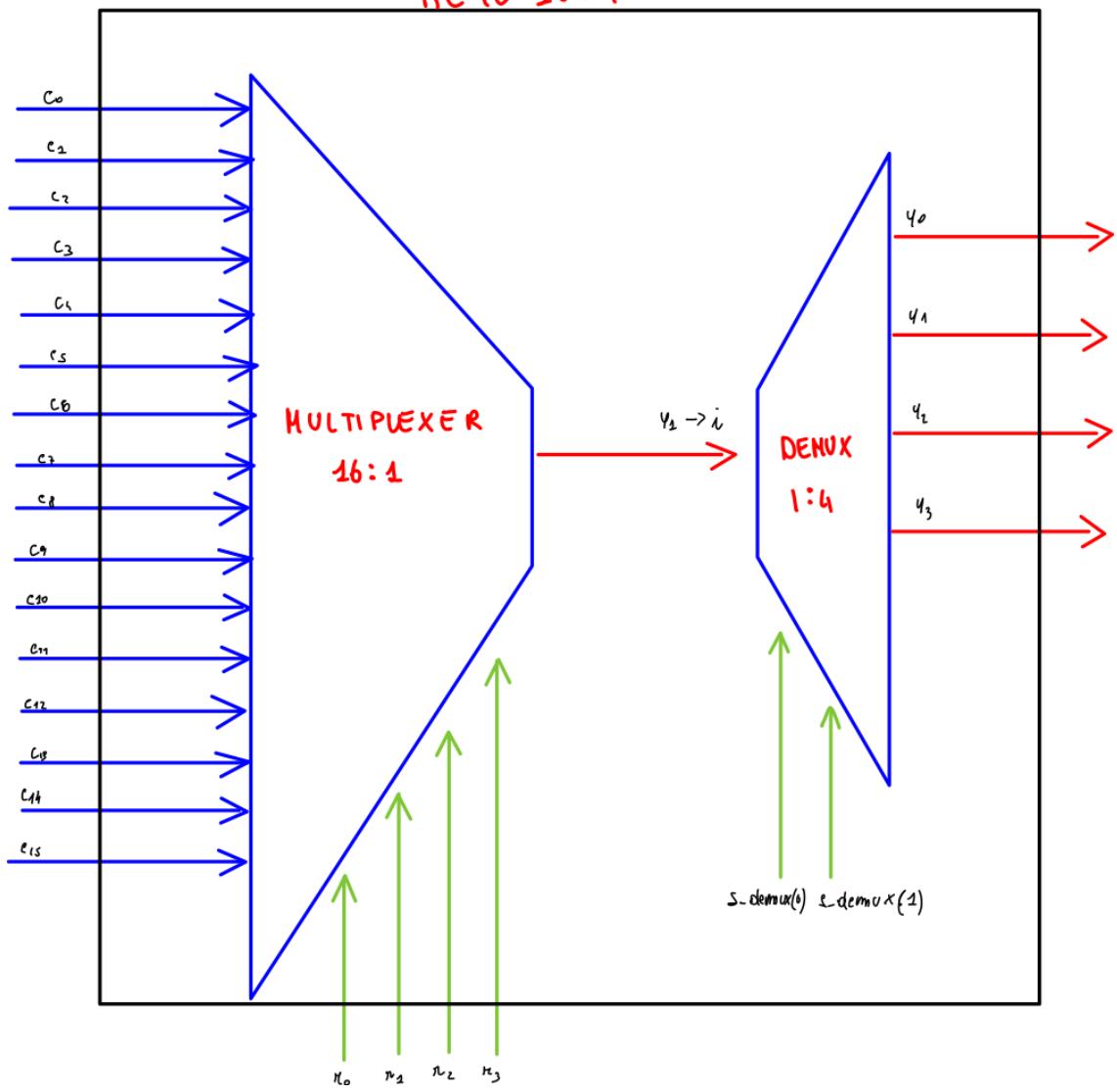


Figura 3 schema rete 16:4

Vediamo anche lo schema dove abbiamo esplicitato come avviene la composizione del multiplexer 16:1

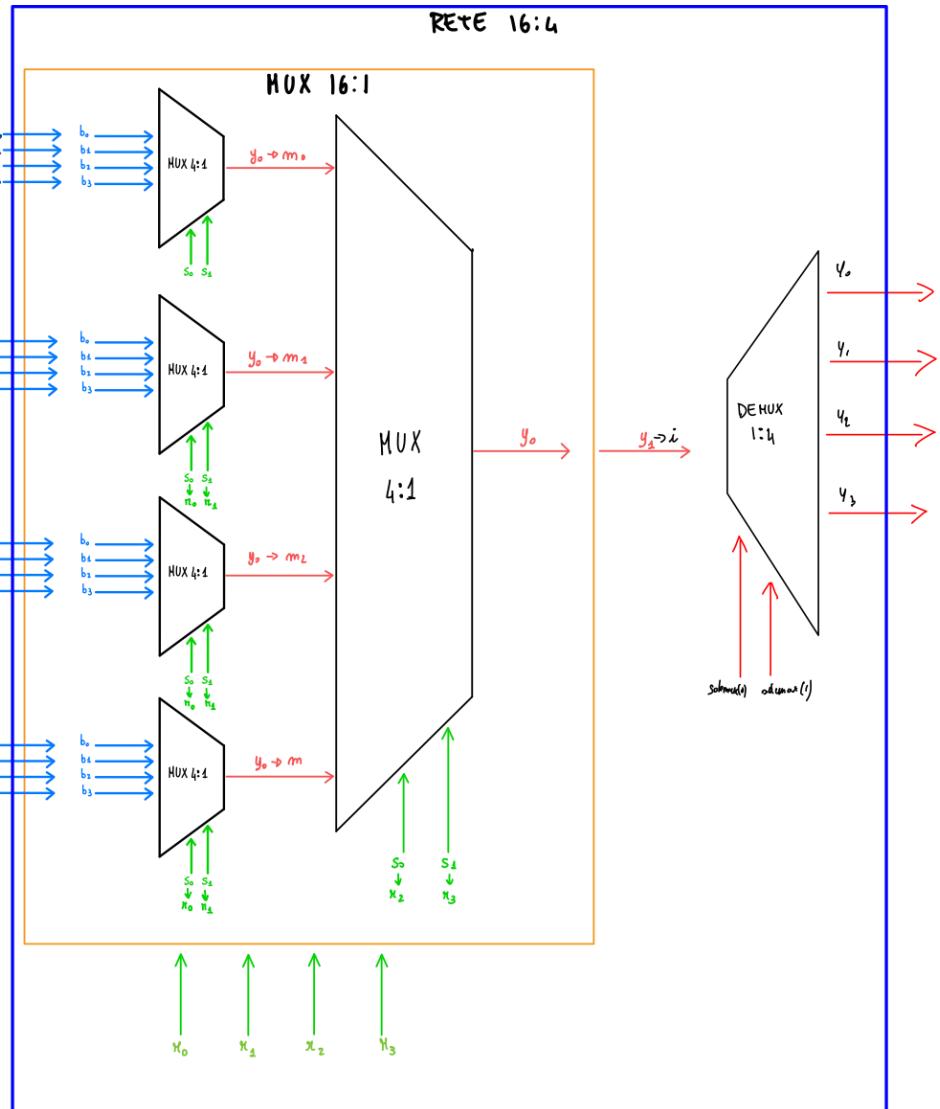


Figura 4 rete 16:4 architettura modulare

Implementazione

Anche in questo caso abbiamo utilizzato un approccio per composizione. Come prima cosa è stata definita l'entity **rete_16_4** che presenta 3 ingressi: **a**, **s_mux**, **s_demux**. Il segnale in ingresso della rete è **a** definito come *std_logic_vector*, rappresenta un insieme di 16 segnali di ingresso provenienti dalle sorgenti della rete di interconnessione. Ciascun elemento di **a** (ad esempio, **a(0)**, **a(1)**, ..., **a(15)**) rappresenta un singolo segnale logico.

I segnali di selezione del multiplexer e del demultiplexer sono rispettivamente **s_mux** e **s_demux**. In questo caso **s_mux** l'abbiamo dichiarato come *std_logic_vector* e rappresenta un insieme di 4 segnali mentre **s_demux** è di tipo *std_logic*. L'uscita è rappresentata da **y**, uno *std_logic_vector* che rappresenta un insieme di 4 segnali

```

entity rete_16_4 is
  port(
    a : in std_logic_vector(15 downto 0);
    s_mux : in std_logic_vector(3 downto 0);
    s_demux : in std_logic_vector(1 downto 0);
    y : out std_logic_vector(3 downto 0)
  );
end rete_16_4;

```

Abbiamo utilizzato un approccio strutturale sfruttando come componenti il **mux_16_1** e il **demux_1_4**:

```

architecture structural of rete_16_4 is

component mux_16_1 is
  port(
    c0 : in STD_LOGIC;
    c1 : in STD_LOGIC;
    c2 : in STD_LOGIC;
    c3 : in STD_LOGIC;
    c4 : in STD_LOGIC;
    c5 : in STD_LOGIC;
    c6 : in STD_LOGIC;
    c7 : in STD_LOGIC;
    c8 : in STD_LOGIC;
    c9 : in STD_LOGIC;
    c10 : in STD_LOGIC;
    c11 : in STD_LOGIC;
    c12 : in STD_LOGIC;
    c13 : in STD_LOGIC;
    c14 : in STD_LOGIC;
    c15 : in STD_LOGIC;
    r0 : in STD_LOGIC;
    r1 : in STD_LOGIC;
    r2 : in STD_LOGIC;
    r3 : in STD_LOGIC;
    y1 : out std_logic
  );
end component;

component demux_1_4 is
  port(
    i : in std_logic;
    s : in std_logic_vector(1 downto 0);
    y0, y1, y2, y3 : out std_logic
  );
end component;

```

```

begin
    muplex: mux_16_1
    Port map(
        c0 => a(0),
        c1 => a(1),
        c2 => a(2),
        c3 => a(3),
        c4 => a(4),
        c5 => a(5),
        c6 => a(6),
        c7 => a(7),
        c8 => a(8),
        c9 => a(9),
        c10 => a(10),
        c11 => a(11),
        c12 => a(12),
        c13 => a(13),
        c14 => a(14),
        c15 => a(15),
        r0 => s_mux(0),
        r1 => s_mux(1),
        r2 => s_mux(2),
        r3 => s_mux(3),
        y1 => link
    );
    demux: demux_1_4
    Port map(
        i => link,
        y0 => y(0),
        y1 => y(1),
        y2 => y(2),
        y3 => y(3),
        s(0) => s_demux(0),
        s(1) => s_demux(1)
    );
end structural;

```

In questo codice vengono create le istanze del multiplexer 16:1 (*muplex*) e del demultiplexer 1:4 (*demux*). Ogni istanza è collegata a un gruppo di ingressi e sono controllate dai segnali di selezione. Il segnale interno *link*, di tipo *std_logic* inizializzato a 0, viene utilizzato per trasmettere in ingresso al demultiplexer l'uscita del multiplexer, consente il trasferimento del segnale da una sorgente selezionata alle destinazioni desiderate nella rete di interconnessione.

Simulazione

Per poter effettuare la simulazione è stato scritto il **testbench**.

Il codice viene mostrato in figura nella pagina successiva:

```

entity rete_16_4_tb is
end rete_16_4_tb;

architecture in_out of rete_16_4_tb is
    component rete_16_4 is
        port(
            a : in std_logic_vector(15 downto 0);
            s_mux : in std_logic_vector(3 downto 0);
            s_demux : in std_logic_vector(1 downto 0);
            y : out std_logic_vector(3 downto 0)
        );
    end component;

```

La prima cosa che abbiamo fatto è stata dichiarare una entity. Si nota che il corpo della entity è vuoto, questo perché non rappresenta un oggetto da realizzare ma ci serve solo per effettuare la simulazione e verificare il corretto funzionamento del sistema. Il testbench non ha segnali di ingresso e di uscita in quanto non può essere istanziato da nessun blocco ma istanzia al suo interno altri blocchi per effettuarne il test.

All'interno dell'architettura del testbench dichiariamo il componente da testare, ovvero **rete_16_4**. Successivamente vengono dichiarati i segnali che fungono da input e da output per il componente da testare. I segnali sono 4 e si collegano alle 4 porte dichiarate del componente.

```

signal input : std_logic_vector (15 downto 0) := (others => 'U');
signal select_mux : std_logic_vector (3 downto 0) := (others => 'U');
signal select_demux : std_logic_vector (1 downto 0) := (others => 'U');
signal output : std_logic_vector (3 downto 0) := (others => 'U');

```

Figura 5 dichiarazione dei segnali

Istanziamo il componente utilizzando l'etichetta **dut** (design under test) e le porte del componente vengono collegate ai segnali dichiarati nel testbench (*input*, *select_mux*, *select_demux*, *output*).

In seguito, abbiamo definito un processo di simulazione **stim_proc** che controlla dinamicamente i valori dei segnali di input. In particolare, vengono applicati vari valori ai segnali di input in tempi specifici, simulando così condizioni diverse del sistema. Ad esempio, si inietta un valore "0000000000000001" nel segnale di ingresso (**input**) e si variano le selezioni del multiplexer e del demultiplexer.

```

begin
    dut : rete_16_4
    port map(
        a => input,
        s_mux => select_mux,
        s_demux => select_demux,
        y => output
    );

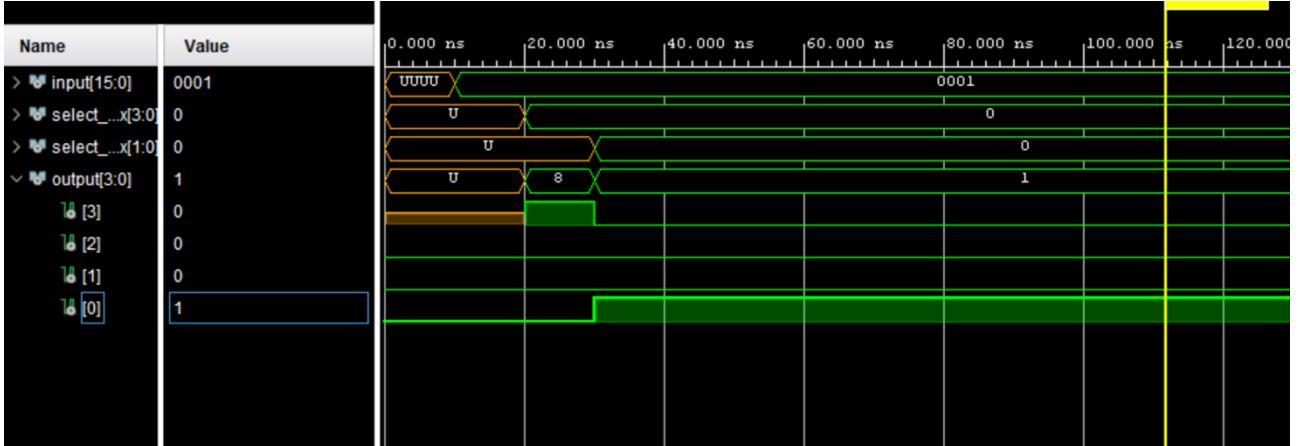
    stim_proc : process
    begin
        wait for 10 ns;
        input <= "0000000000000001";
        wait for 10 ns;
        select_mux <= "0000";
        wait for 10 ns;
        select_demux <= "00";
        wait for 10 ns;
        wait;
    end process;

end;

```

L'ultima istruzione di wait fa sì che la simulazione rimanga in esecuzione indefinitamente in modo da poter monitorare l'andamento del sistema durante l'intera durata della simulazione.

Avviando la simulazione abbiamo il seguente risultato:



Analizziamo il grafico per controllare che la simulazione sia andata buon fine. Come prima cosa notiamo che tutti i segnali inizialmente sono posti undefined proprio come dichiarato nel testbench.

Dopo la prima wait di 10ns viene settato il valore di input a 0000000000000001, dopo la seconda wait di 10ns (quindi quando ci troviamo a 20ns) l'ingresso di selezione del multiplexer viene posto a 0, dopo ulteriori 10 ns, quindi a 30ns, anche il segnale di demux viene posto a 0. Ciò implica che l'output debba essere pari a 1 e come possiamo osservare è proprio ciò che accade

Esercizio 1.3

Sintetizzare ed implementare su board il progetto della rete di interconnessione sviluppato al punto 1.2, utilizzando gli switch per fornire gli input di selezione e i led per visualizzare i 4 bit di uscita. Per quanto riguarda i 16 bit dato in input, essi devono essere immessi mediante switch, 8 bit alla volta, sviluppando

un'apposita “rete di controllo” per l’acquisizione che utilizzi due bottoni della board per caricare rispettivamente la prima e la seconda metà del dato in ingresso.

Progetto e architettura

Per implementare su board il progetto della rete di interconnessione siamo andati a riscrivere il codice del sistema. La prima cosa che è stata fatta è stata l’istanziazione di una entity **sistema**.

Anche in questo caso è stato seguito un approccio modulare. L’entity **sistema** rappresenta un sistema di interconnessione modulare che coinvolge due componenti principali: l’unità di controllo (**unita_controllo**) e la rete di interconnessione (**rete_16_4**). Questa progettazione modulare offre una soluzione flessibile e scalabile per gestire dati in ingresso e instradare selettivamente gli stessi in base alle esigenze del sistema.

```
entity sistema is
  Port (
    ingresso : in std_logic_vector(15 downto 0);
    selMux : in std_logic_vector(3 downto 0);
    selDemux : in std_logic_vector(1 downto 0);
    buttons : in std_logic_vector(1 downto 0);
    uscita : out std_logic_vector(3 downto 0)
  );
end sistema;
```

Figura 6 definizione entity sistema

Abbiamo definito 4 segnali di input: *ingresso*, *selMux*, *selDemux*, *buttons*.

- **ingresso:** (*std_logic_vector*) Rappresenta un vettore di 16 bit proveniente dagli switch sulla board e costituisce l’input principale per il sistema.
- **selMux:** (*std_logic_vector*) È un vettore di selezione di 4 bit che determina quale delle 16 sorgenti deve essere instradata nella rete di interconnessione.
- **selDemux:** (*std_logic_vector*) È un vettore di selezione di 2 bit che guida il demultiplexer nella rete di interconnessione per instradare il segnale verso una delle 4 destinazioni possibili.
- **buttons:** (*std_logic_vector*) Rappresenta un vettore di 2 bit associato ai bottoni sulla board. Viene utilizzato per controllare il funzionamento dell’unità di controllo.

Il segnale di output è solo uno, *uscita*, ed è un vettore di 4 bit che rappresenta la combinazione degli input selezionati dalla rete di interconnessione.

Per l’architettura abbiamo seguito un approccio strutturale. Sono stati definiti due componenti: *rete_16_4* e *unita_controllo*.

- L’unità di controllo (**unita_controllo**) è incaricata di gestire il condizionamento degli input provenienti dagli switch sulla board. Questi dati vengono opportunamente indirizzati e trasmessi alla rete di interconnessione attraverso il segnale **link**.
- La rete di interconnessione (**rete_16_4**) si occupa di selezionare, instradare e distribuire i dati in ingresso in base alle segnalazioni di selezione (**selMux** e **selDemux**). L’output risultante viene restituito come segnale **uscita**.

```

architecture Behavioral of sistema is

    component rete_16_4 is
        port(
            a : in std_logic_vector(15 downto 0);
            s_mux : in std_logic_vector(3 downto 0);
            s_demux : in std_logic_vector(1 downto 0);
            y : out std_logic_vector(3 downto 0)
        );
    end component;

    component unita_controllo is
        port (
            b1 : in std_logic;
            b2 : in std_logic;
            switchIN : in std_logic_vector(15 downto 0);
            switchOUT : out std_logic_vector(15 downto 0)
        );
    end component;

```

Figura 7 definizione componenti

L'unità di controllo cattura i dati dagli switch sulla board in risposta ai comandi provenienti dai bottoni (**b1** e **b2**).

I dati condizionati vengono trasmessi alla rete di interconnessione attraverso il segnale **link**.

La rete di interconnessione seleziona e instrada i dati in ingresso in base alle segnalazioni di selezione (**selMulx** e **selDemux**).

Il risultato finale è riflettuto nell'output del sistema (**uscita**), che rappresenta la combinazione degli input selezionati.

Prima di proseguire analizziamo meglio l'**unità_controllo** che abbiamo definito in un ulteriore file.

```

entity unita_controllo is
    Port (
        b1 : in std_logic;
        b2 : in std_logic;
        switchIN : in std_logic_vector(15 downto 0);
        switchOUT : out std_logic_vector(15 downto 0)
    );
end unita_controllo;

architecture Behavioral of unita_controllo is
    signal temp : std_logic_vector(15 downto 0);
begin
    process(switchIN, b1, b2) begin
        if (b1 = '1') then
            temp(7 downto 0) <= switchIN(7 downto 0);
        elsif (b2 = '1') then
            temp(15 downto 8) <= switchIN(15 downto 8);
        end if;
    end process;
    switchOUT <= temp;
end Behavioral;

```

Figura 8 entity unità di controllo

L'entity **unità_controllo** costituisce un componente chiave all'interno del sistema, responsabile di condizionare gli input provenienti dagli switch in base agli stati dei bottoni (**b1** e **b2**).

Gli input sono i seguenti:

- **b1 e b2:** (*std_logic*) Sono due segnali provenienti dai bottoni della board. Determinano lo stato operativo dell'unità di controllo.

- **switchIN**: (*std_logic_vector*) Rappresenta un vettore di 16 bit proveniente dagli switch sulla board. L'output è rappresentato dal segnale, di tipo *std_logic_vector*, **switchOUT** che rappresenta un vettore di 16 bit risultante dall'elaborazione dei dati in ingresso.

L'unità di controllo monitora lo stato dei pulsanti (**b1** e **b2**). In risposta all'attivazione di uno dei pulsanti, i dati appropriati provenienti dagli switch vengono copiati in una variabile temporanea (**temp**). Il vettore **temp** rappresenta la porzione di dati selezionata in base allo stato dei pulsanti. Il risultato (**switchOUT**) viene trasmesso alla rete di interconnessione attraverso il segnale **link**.

Ritornando al nostro sistema, dopo la definizione dei due componenti si ha l'istanziazione:

```
begin

    CU : unita_controllo
    port map(
        b1 => buttons(0),
        b2 => buttons(1),
        switchIN => ingresso,
        switchOUT => link
    );

    rete : rete_16_4
    port map(
        a => link,
        s_mux => selMulx,
        s_demux => selDemux,
        y => uscita
    );
end Behavioral;
```

Figura 9 architettura behavioral

I segnali di input vengono mappati ai segnali di ingresso del componente **unita_controllo** usando le seguenti associazioni:

- **b1** (pulsante 1) è collegato a **buttons(0)**.
- **b2** (pulsante 2) è collegato a **buttons(1)**.
- **switchIN** è collegato a **ingresso**.
- **switchOUT** è collegato a **link**.

L'unità di controllo gestisce gli input dai pulsanti e dagli switch, la **rete_16_4** manipola i segnali in base alle modalità da noi specificate.

Nella **rete_16_4** si effettua il seguente mapping:

- **a** è collegato a **link**.
- **s_mux** è collegato a **selMulx**.
- **s_demux** è collegato a **selDemux**.
- **y** è collegato a **uscita**.

Lo schema risultante è il seguente:

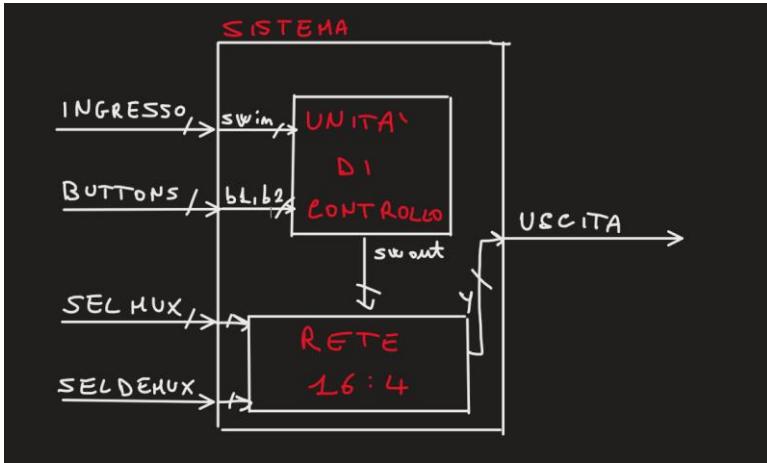


Figura 10 schema sistema

Per poter utilizzare la board è stato necessario effettuare alcune modifiche al file di configurazione **Nexys-A7-100T-Master.xdc**:

I dispositivi di input e output della board utilizzati sono stati collegati in modo opportuno usando i segnali citati qui sopra.

I 16 switch sono stati tutto collegati all'ingresso del multiplexer, l'uscita è stata collegata ai primi 4 led e per abilitare gli ingressi abbiamo fatto uso dei bottoni, in particolare, sono stati utilizzati il BTNL (il bottone che si trova a sinistra tra i 5 bottoni utilizzabili) per abilitare i primi 8 bit dell'ingresso e il BTNR (il bottone che si trova a destra tra i 5 bottoni utilizzabili) per abilitare gli altri 8 bit dell'ingresso.

Per la selezione degli ingressi del multiplexer e delle uscite del demultiplexer i valori sono presi di default (tutti '0'), ma per controllarli è possibile vedere l'illuminazione dei led, dato che tali segnali sono mappati sulla scheda, in modo da poter fare un debug.

```
##Switches
set_property -dict { PACKAGE_PIN J15 IOSTANDARD LVCMS33 } [get_ports { ingresso[0] }]; #IO_L24N_T3_RS0_15 Sch=sw[0]
set_property -dict { PACKAGE_PIN L16 IOSTANDARD LVCMS33 } [get_ports { ingresso[1] }]; #IO_L3N_T0_DQ5_EMCCCLK_14 Sch=sw[1]
set_property -dict { PACKAGE_PIN M13 IOSTANDARD LVCMS33 } [get_ports { ingresso[2] }]; #IO_L6N_T0_D08_VREF_14 Sch=sw[2]
set_property -dict { PACKAGE_PIN R15 IOSTANDARD LVCMS33 } [get_ports { ingresso[3] }]; #IO_L13N_T2_MRCC_14 Sch=sw[3]
set_property -dict { PACKAGE_PIN R17 IOSTANDARD LVCMS33 } [get_ports { ingresso[4] }]; #IO_L12N_T1_MRCC_14 Sch=sw[4]
set_property -dict { PACKAGE_PIN T18 IOSTANDARD LVCMS33 } [get_ports { ingresso[5] }]; #IO_L7N_T1_D10_14 Sch=sw[5]
set_property -dict { PACKAGE_PIN U18 IOSTANDARD LVCMS33 } [get_ports { ingresso[6] }]; #IO_L17N_T2_A13_D29_14 Sch=sw[6]
set_property -dict { PACKAGE_PIN R13 IOSTANDARD LVCMS33 } [get_ports { ingresso[7] }]; #IO_L5N_T0_D07_14 Sch=sw[7]
set_property -dict { PACKAGE_PIN T8 IOSTANDARD LVCMS18 } [get_ports { ingresso[8] }]; #IO_L24N_T3_34 Sch=sw[8]
set_property -dict { PACKAGE_PIN U8 IOSTANDARD LVCMS18 } [get_ports { ingresso[9] }]; #IO_25_34 Sch=sw[9]
set_property -dict { PACKAGE_PIN R16 IOSTANDARD LVCMS33 } [get_ports { ingresso[10] }]; #IO_L15P_T2_DQS_RDWR_B_14 Sch=sw[10]
set_property -dict { PACKAGE_PIN T13 IOSTANDARD LVCMS33 } [get_ports { ingresso[11] }]; #IO_L23P_T3_A03_D19_14 Sch=sw[11]
set_property -dict { PACKAGE_PIN H6 IOSTANDARD LVCMS33 } [get_ports { ingresso[12] }]; #IO_L24P_T3_35 Sch=sw[12]
set_property -dict { PACKAGE_PIN U12 IOSTANDARD LVCMS33 } [get_ports { ingresso[13] }]; #IO_L20P_T3_A08_D24_14 Sch=sw[13]
set_property -dict { PACKAGE_PIN U11 IOSTANDARD LVCMS33 } [get_ports { ingresso[14] }]; #IO_L19N_T3_A09_D25_VREF_14 Sch=sw[14]
set_property -dict { PACKAGE_PIN V10 IOSTANDARD LVCMS33 } [get_ports { ingresso[15] }]; #IO_L21P_T3_DQS_14 Sch=sw[15]

## LEDs
set_property -dict { PACKAGE_PIN H17 IOSTANDARD LVCMS33 } [get_ports { uscita[0] }]; #IO_L18P_T2_A24_15 Sch=led[0]
set_property -dict { PACKAGE_PIN K15 IOSTANDARD LVCMS33 } [get_ports { uscita[1] }]; #IO_L24P_T3_RS1_15 Sch=led[1]
set_property -dict { PACKAGE_PIN J13 IOSTANDARD LVCMS33 } [get_ports { uscita[2] }]; #IO_L17N_T2_A25_15 Sch=led[2]
set_property -dict { PACKAGE_PIN N14 IOSTANDARD LVCMS33 } [get_ports { uscita[3] }]; #IO_L8P_T1_D11_14 Sch=led[3]
set_property -dict { PACKAGE_PIN R18 IOSTANDARD LVCMS33 } [get_ports { selDemux[0] }]; #IO_L7P_T1_D09_14 Sch=led[4]
set_property -dict { PACKAGE_PIN V17 IOSTANDARD LVCMS33 } [get_ports { selDemux[1] }]; #IO_L18N_T2_A11_D27_14 Sch=led[5]
set_property -dict { PACKAGE_PIN U17 IOSTANDARD LVCMS33 } [get_ports { selMux[0] }]; #IO_L17P_T2_A14_D30_14 Sch=led[6]
set_property -dict { PACKAGE_PIN U16 IOSTANDARD LVCMS33 } [get_ports { selMux[1] }]; #IO_L18P_T2_A12_D28_14 Sch=led[7]
set_property -dict { PACKAGE_PIN V16 IOSTANDARD LVCMS33 } [get_ports { selMux[2] }]; #IO_L16N_T2_A15_D31_14 Sch=led[8]
set_property -dict { PACKAGE_PIN T15 IOSTANDARD LVCMS33 } [get_ports { selMux[3] }]; #IO_L14N_T2_SRCC_14 Sch=led[9]
```

Esercizio 2: Sistema ROM + M

Esercizio 2.1

Progettare, implementare in VHDL e testare mediante simulazione un **sistema S** composto da una **ROM** puramente combinatoria di 16 locazioni da 8 bit ciascuna e da una macchina combinatoria **M**

che opera come segue: fornito al sistema un indirizzo A di 4 bit, il sistema restituisce il valore contenuto nella ROM all'indirizzo A opportunamente “trasformato” attraverso la macchina M. Il comportamento della macchina M è totalmente a scelta dello studente, l'unico vincolo è che essa prenda in ingresso 8 bit e ne fornisca in uscita 4.

Progetto e architettura

L'architettura è modulare, abbiamo utilizzato due componenti principali: **rom_comb** e **macchina**. La **rom_comb** è la memoria ROM, mentre la **macchina** è la componente che esegue la trasformazione sugli 8 bit in ingresso per restituire 4 bit in uscita. Entrambi i componenti sono dichiarati come entità separate con relativi port e generici che consentono flessibilità nella configurazione.

La memoria ROM è implementata come un array di 16 locazioni, ognuna contenente un valore a 8 bit. L'indirizzo fornito al sistema determina quale locazione della memoria viene selezionata, e il valore corrispondente viene quindi inviato alla macchina combinatoria. La macchina a sua volta restituisce un risultato a 4 bit.

Lo schema del sistema è il seguente:

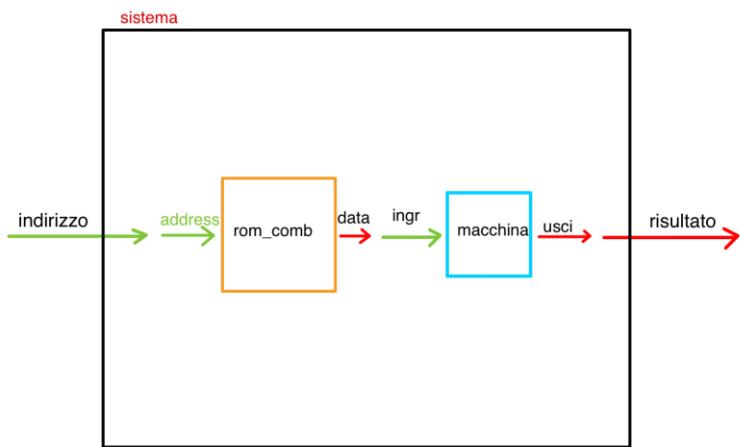


Figura 11 schema sistema

Il componente **rom_comb** viene collegato all'indirizzo fornito in ingresso al sistema (**indirizzo**), mentre la **macchina** riceve il valore dalla memoria e restituisce il risultato attraverso il segnale **risultato**.

Implementazione

La nostra soluzione implementa un sistema composto da una memoria ROM (**rom_comb**) di 16 locazioni da 8 bit ciascuna e una macchina combinatoria (**macchina**). La memoria ROM è progettata per essere puramente combinatoria, e la macchina combinatoria prende in ingresso 8 bit e restituisce 4 bit in base al valore del bit meno significativo dell'ingresso.

Prima di analizzare il sistema complessivo poniamo attenzione sull'implementazione degli altri due componenti ovvero la rom e la macchina.

Cominciamo dalla rom combinatoria (**rom_comb**).

```

entity rom_comb is
  generic(
    word_length : positive := 8;
    depth : positive := 4;
    numero_locazioni : positive := 2**4
  );
  port(
    address : in std_logic_vector(depth-1 downto 0); --indirizzo di memoria
    data : out std_logic_vector (word_length-1 downto 0) --dati in uscita
  );
end rom_comb;

```

Figura 12 entity rom comb

Nell'entity abbiamo dichiarato tre generici:

- **word_length**: Specifica la lunghezza di ciascuna parola di memoria nella ROM, ovvero il numero di bit contenuti in ciascuna locazione di memoria.
- **depth**: Indica la profondità della ROM.
- **numero_locazioni**: Determina il numero totale di locazioni di memoria. In particolare, è fissato a 2^{**4} , il che significa che il numero di locazioni è 2 elevato alla potenza di 4, ovvero 16.

Oltre ai generici sono stati dichiarati due segnali, uno di input e uno di output. In input la rom riceve il segnale di tipo *std_logic_vector address* che indica l'indirizzo in memoria. Il segnale di output, anch'esso di tipo *std_logic_vector*, è il segnale *data* che rappresenta i dati in uscita.

L'architettura della **rom_comb** è di tipo *dataflow*

```

architecture dataflow of rom_comb is

  type memory_16_4 is array(0 to numero_locazioni-1) of std_logic_vector(word_length-1
  downto 0); --rappresenta le celle di memoria
  constant myrom : memory_16_4 :=(
    2 => "11111111" , --255
    3 => "11010101" ,
    4 => "01101000" ,
    6 => "10011011" ,
    8 => "01101101" ,
    9 => "00110111" ,
    others => "00000000" ) ;
  begin
  process(address)
  begin
    data <= myrom(to_integer(unsigned(address))); --accedi alla locazione e ritira
    il dato
  end process;
end architecture dataflow;

```

Figura 13 architettura dataflow rom comb

Viene dichiarato un nuovo tipo *array* chiamato **memory_16_4** che rappresenta le celle di memoria della ROM. Questo array ha 16 elementi (locazioni di memoria), ciascuno dei quali è un vettore di lunghezza **word_length** (numero di bit in ciascuna parola di memoria).

Successivamente, viene dichiarata e inizializzata una costante **myrom** di tipo **memory_16_4** che rappresenta i dati precaricati nella ROM per le diverse locazioni. Ad esempio, la locazione 2 contiene il valore "11111111" (255 in binario), la locazione 3 contiene "11010101", e così via.

Infine, viene definito un processo con sensibilità al segnale **address** (l'indirizzo di memoria in ingresso). Ogni volta che il valore di **address** cambia, il processo viene attivato.

All'interno del processo, il valore di **address** viene convertito in un intero senza segno (**to_integer(unsigned(address))**) e utilizzato per accedere all'elemento corrispondente nella memoria **myrom**. Il valore associato a quell'indirizzo viene quindi assegnato al segnale **data** in uscita.

Adesso andiamo ad analizzare l'implementazione della **macchina**.

```
entity macchina is
  generic(
    lenght_in : positive := 8;
    lenght_out : positive := 4
  );
  port(
    ingr : in std_logic_vector(lenght_in-1 downto 0);
    usci : out std_logic_vector(lenght_out-1 downto 0)
  );
end macchina;
```

Figura 14 entity macchina

Nella sezione **generic**, vengono dichiarati due parametri generici:

- **lenght_in**: specifica la lunghezza in bit dell'ingresso (**ingr**), e il suo valore predefinito è 8.
- **lenght_out**: specifica la lunghezza in bit dell'uscita (**usci**), e il suo valore predefinito è 4.

Nella sezione **port**, vengono dichiarati i segnali di ingresso e di uscita:

- **ingr**: segnale di ingresso di tipo *std_logic_vector* con lunghezza specificata dal generico **lenght_in**.
- **usci**: segnale di uscita di tipo *std_logic_vector* con lunghezza specificata dal generico **lenght_out**.

Anche per l'architettura della macchina abbiamo utilizzato un approccio dataflow.

```
architecture dataflow of macchina is
begin
  process(ingr)
    begin
      if (ingr(0) = '1') then --se pari
        usci <= "1111";
      elsif(ingr(0) = '0') then --se dispari
        usci <= "0000";
      else
        usci <= "1010";
      end if;
    end process;
end architecture dataflow;
```

Viene dichiarato un processo sensibile al segnale **ingr**. Ciò significa che ogni volta cambia il valore del segnale si avrà l'esecuzione del processo. Tramite l'utilizzo dei costrutti **if**, in base al valore del segnale **ingr** assegniamo un determinato valore al segnale di output **usci**. In questo caso il valore dipende dal fatto che l'ingresso sia pari o dispari.

Dopo aver implementato la rom e la macchina uniamo tutto nell'implementazione del **sistema**.

L'entity del sistema è la seguente:

```

entity sistema is
  generic(
    lunghezza_uscita : positive := 4;
    lunghezza_addr : positive := 4
  );
  port(
    indirizzo : in std_logic_vector(lunghezza_addr-1 downto 0);
    risultato : out std_logic_vector(lunghezza_uscita-1 downto 0)
  );
end sistema;

```

Figura 15 entity sistema

L'entity ha due generici (**lunghezza_uscita** e **lunghezza_addr**) che rappresentano la lunghezza dei vettori di uscita e degli indirizzi, rispettivamente. I suoi port sono **indirizzo** (vettore di ingresso degli indirizzi) e **risultato** (vettore di uscita).

Abbiamo 1 segnale sia per ingresso che per uscita:

- **indirizzo**: È un vettore di ingresso di lunghezza **lunghezza_addr** e rappresenta l'indirizzo che sarà utilizzato per accedere alla memoria ROM.
- **risultato**: È un vettore di uscita di lunghezza **lunghezza_uscita** e rappresenta il risultato prodotto dalla macchina combinatoria.

In questo caso per l'architettura abbiamo scelto un approccio structural.

La prima cosa che facciamo è l'istanziazione dei due componenti ovvero della **macchina** e della **rom_comb**.

```

architecture structural of sistema is

  component macchina is
    generic(
      lenght_in : positive := 8;
      lenght_out : positive := 4
    );
    port(
      ingr : in std_logic_vector(lenght_in-1 downto 0);
      usci : out std_logic_vector(lenght_out-1 downto 0)
    );
  end component;

  component rom_comb is
    generic(
      word_lenght : positive := 8;
      depth : positive := 4;
      numero_locazioni : positive := 2**4
    );
    port(
      address : in std_logic_vector(depth-1 downto 0); --indirizzo di memoria
      data : out std_logic_vector (word_lenght-1 downto 0) --dati in uscita
    );
  end component;

```

rom: rom_comb: Questa istruzione istanzia un componente di tipo **rom_comb** chiamato **rom**. La **port map** specifica come i segnali interni del componente (**address** e **data**) sono collegati ai segnali esterni (**indirizzo** e **link**).

m: macchina: Questa istruzione istanzia un componente di tipo **macchina** chiamato **m**. La **port map** specifica come i segnali interni del componente (**ingr** e **usci**) sono collegati ai segnali esterni (**link** e **risultato**).

Il segnale **indirizzo** viene collegato all'input **address** del componente **rom_comb**. Questo segnale rappresenta l'indirizzo di memoria a 4 bit fornito al sistema.

Il segnale **link** funge da collegamento tra il componente **rom_comb** e il componente **macchina**. Il dato in uscita dalla ROM (**data**) viene collegato a **link**, che rappresenta l'input della macchina (**ingr**).

Il segnale **risultato** viene collegato all'output della macchina (**usci**), rappresentando il risultato finale del sistema.

Continuiamo ad analizzare il codice.

```
signal link : std_logic_vector(7 downto 0);
type memory_16_4 is array(0 to 15) of std_logic_vector(7 downto 0); --rappresenta le celle
di memoria
constant myrom : memory_16_4 :=(
2 => "11111111",
3 => "11010101",
4 => "01101000",
6 => "10011011",
8 => "01101101",
9 => "00110111",
others => "00000000" );
begin
    rom : rom_comb
    port map(
        address => indirizzo,
        data => link
    );
    m : macchina
    port map(
        ingr => link,
        usci => risultato
    );
end architecture structural;
```

Viene dichiarato il segnale **link** come *std_logic_vector* di lunghezza 8 bit. Viene utilizzato come ponte tra la ROM (**rom_comb**) e la macchina (**macchina**). La **port map** specifica che il segnale **link** è utilizzato per trasferire i dati dalla ROM alla macchina, consentendo la comunicazione tra i due componenti.

Simulazione

Per effettuare la simulazione è stato necessario scrivere il testbench.

La prima cosa che abbiamo fatto è stata dichiarare una entity. Si nota che il corpo della entity è vuoto, questo perché non rappresenta un oggetto da realizzare ma ci serve solo per effettuare la simulazione e verificare il corretto funzionamento del sistema. Il testbench non ha segnali di ingresso e di uscita in quanto non può essere istanziato da nessun blocco ma istanzia al suo interno altri blocchi per effettuarne il test.

Successivamente dichiariamo il componente da testare, ovvero il sistema.

```
entity sistema_tb is
end sistema_tb;

architecture behavioural of sistema_tb is

component sistema is
    generic(
        lunghezza_uscita : positive := 4;
        lunghezza_addr : positive := 4
    );
    port(
        indirizzo : in std_logic_vector(lunghezza_addr-1 downto 0);
        risultato : out std_logic_vector(lunghezza_uscita-1 downto 0)
    );
end component;
```

Figura 16 testbench

Successivamente vengono dichiarati i segnali che fungono da input e da output per il componente da testare.

```

signal input : std_logic_vector (3 downto 0) := (others => 'U');
signal output : std_logic_vector (3 downto 0) := (others => 'U');

begin
    dut : sistema
        port map(
            indirizzo => input,
            risultato => output
        );

    stim_proc : process
    begin
        wait for 10 ns;
        input <= "0010";
        wait for 10 ns;
        wait;
    end process;

end;

```

Vengono dichiarati due segnali, **input** e **output**, entrambi di tipo **std_logic_vector** con lunghezza 4. Questi segnali sono inizializzati con il valore 'U' (non definito).

All'interno del blocco architetturale abbiamo l'istanziazione del componente da testare **sistema** utilizzando l'etichetta **dut** (design under test).

La clausola **port map** associa il segnale **input** alla porta **indirizzo** e il segnale **output** alla porta **risultato** del componente **sistema**.

Viene dichiarato un processo **stim_proc** che rappresenta il generatore di stimoli del testbench. In questo caso, dopo 10 ns, il valore del segnale **input** viene impostato su "0010". Dopo ulteriori 10 ns, il processo entra in uno stato di attesa (**wait**) indefinito.

Avviando la simulazione possiamo osservare il seguente risultato:



Dal grafico possiamo osservare come l'inizializzazione del segnale di input sia corretta essendo che si presenta undefined prima dell'assegnazione del valore che avviene dopo 10ns.

Dopo 10ns il segnale di input viene posto a 0010 ovvero 2 in decimale. Essendo 2 un numero pari, per la logica con cui abbiamo implementato la macchina, dovremmo ottenere come output 1111 ed è ciò che accade.

Esercizio 2.2

Sintetizzare ed implementare su board il progetto del sistema ROM+M sviluppato al punto 2.1, utilizzando gli switch per fornire l'indirizzo della ROM da cui leggere i valori da trasformare e i led per visualizzare i 4 bit di uscita.

Progetto e architettura

Per ottenere il risultato richiesto non è stato necessaria la modifica di nessun file presente nell'implementazione originale. Infatti, è bastato collegare opportunamente l'indirizzo, che era l'ingresso del sistema, agli switch presenti sulla board, in modo da leggere dalla ROM i valori contenuti. Una volta letti i valori, essi vengono trasformati, secondo l'implementazione sviluppata nell'esercizio precedente, e mostrati in uscita sui led della board, quindi ai led è stata collegata l'uscita "risultato" del sistema.

Implementazione

Il file **Nexys-A7-100T-Master.xdc** è stato modificato nel modo seguente:

```
##Switches
set_property -dict { PACKAGE_PIN J15 IOSTANDARD LVCMOS33 } [get_ports { indirizzo[0] }]; #IO_L24N_T3_RS0_15 Sch=sw[0]
set_property -dict { PACKAGE_PIN L16 IOSTANDARD LVCMOS33 } [get_ports { indirizzo[1] }]; #IO_L3N_T0_D05_EMCLLK_14 Sch=sw[1]
set_property -dict { PACKAGE_PIN M13 IOSTANDARD LVCMOS33 } [get_ports { indirizzo[2] }]; #IO_L6N_T0_D08_VREF_14 Sch=sw[2]
set_property -dict { PACKAGE_PIN R15 IOSTANDARD LVCMOS33 } [get_ports { indirizzo[3] }]; #IO_L13N_T2_MRCC_14 Sch=sw[3]
#set_property -dict { PACKAGE_PIN R17 IOSTANDARD LVCMOS33 } [get_ports { X[4] }]; #IO_L12N_T4_MRCC_14 Sch=sw[4]
#set_property -dict { PACKAGE_PIN T18 IOSTANDARD LVCMOS33 } [get_ports { X[5] }]; #IO_L7N_T1_D10_14 Sch=sw[5]
#set_property -dict { PACKAGE_PIN U18 IOSTANDARD LVCMOS33 } [get_ports { X[6] }]; #IO_L17N_T2_A13_D29_14 Sch=sw[6]
#set_property -dict { PACKAGE_PIN R13 IOSTANDARD LVCMOS33 } [get_ports { X[7] }]; #IO_L5N_T0_D07_14 Sch=sw[7]
#set_property -dict { PACKAGE_PIN T8 IOSTANDARD LVCMOS33 } [get_ports { X[8] }]; #IO_L24N_T3_34 Sch=sw[8]
#set_property -dict { PACKAGE_PIN U8 IOSTANDARD LVCMOS33 } [get_ports { X[9] }]; #IO_25_34 Sch=sw[9]
#set_property -dict { PACKAGE_PIN R16 IOSTANDARD LVCMOS33 } [get_ports { SW[10] }]; #IO_L15P_T2_D05_RDNR_B_14 Sch=sw[10]
#set_property -dict { PACKAGE_PIN T13 IOSTANDARD LVCMOS33 } [get_ports { SW[11] }]; #IO_L23P_T3_A03_D19_14 Sch=sw[11]
#set_property -dict { PACKAGE_PIN H6 IOSTANDARD LVCMOS33 } [get_ports { SW[12] }]; #IO_L24P_T3_35 Sch=sw[12]
#set_property -dict { PACKAGE_PIN U12 IOSTANDARD LVCMOS33 } [get_ports { SW[13] }]; #IO_L20P_T3_A08_D24_14 Sch=sw[13]
#set_property -dict { PACKAGE_PIN U11 IOSTANDARD LVCMOS33 } [get_ports { SW[14] }]; #IO_L19N_T3_A09_D25_VREF_14 Sch=sw[14]
#set_property -dict { PACKAGE_PIN V10 IOSTANDARD LVCMOS33 } [get_ports { SW[15] }]; #IO_L21P_T3_D05_14 Sch=sw[15]

## LEDs
set_property -dict { PACKAGE_PIN H17 IOSTANDARD LVCMOS33 } [get_ports { risultato[0] }]; #IO_L18P_T2_A24_15 Sch=led[0]
set_property -dict { PACKAGE_PIN K15 IOSTANDARD LVCMOS33 } [get_ports { risultato[1] }]; #IO_L24P_T3_RS1_15 Sch=led[1]
set_property -dict { PACKAGE_PIN J13 IOSTANDARD LVCMOS33 } [get_ports { risultato[2] }]; #IO_L17N_T2_A25_15 Sch=led[2]
set_property -dict { PACKAGE_PIN N14 IOSTANDARD LVCMOS33 } [get_ports { risultato[3] }]; #IO_L8P_T1_D11_14 Sch=led[3]
#set_property -dict { PACKAGE_PIN R18 IOSTANDARD LVCMOS33 } [get_ports { LED[4] }]; #IO_L7P_T1_D09_14 Sch=led[4]
#set_property -dict { PACKAGE_PIN V17 IOSTANDARD LVCMOS33 } [get_ports { LED[5] }]; #IO_L18N_T2_A11_D27_14 Sch=led[5]
#set_property -dict { PACKAGE_PIN U17 IOSTANDARD LVCMOS33 } [get_ports { LED[6] }]; #IO_L17P_T2_A14_D30_14 Sch=led[6]
#set_property -dict { PACKAGE_PIN U16 IOSTANDARD LVCMOS33 } [get_ports { LED[7] }]; #IO_L18P_T2_A12_D28_14 Sch=led[7]
#set_property -dict { PACKAGE_PIN V16 IOSTANDARD LVCMOS33 } [get_ports { LED[8] }]; #IO_L16N_T2_A15_D31_14 Sch=led[8]
#set_property -dict { PACKAGE_PIN T15 IOSTANDARD LVCMOS33 } [get_ports { LED[9] }]; #IO_L14N_T2_SRCC_14 Sch=led[9]
#set_property -dict { PACKAGE_PIN U14 IOSTANDARD LVCMOS33 } [get_ports { LED[10] }]; #IO_L22P_T2_A05_D21_14 Sch=led[10]
#set_property -dict { PACKAGE_PIN T16 IOSTANDARD LVCMOS33 } [get_ports { LED[11] }]; #IO_L15H_T2_D05_DOUT_CSD_B_14 Sch=led[11]
#set_property -dict { PACKAGE_PIN V15 IOSTANDARD LVCMOS33 } [get_ports { LED[12] }]; #IO_L16P_T2_C51_B_14 Sch=led[12]
#set_property -dict { PACKAGE_PIN V14 IOSTANDARD LVCMOS33 } [get_ports { LED[13] }]; #IO_L22N_T3_A04_D20_14 Sch=led[13]
#set_property -dict { PACKAGE_PIN V12 IOSTANDARD LVCMOS33 } [get_ports { LED[14] }]; #IO_L20N_T3_A07_D23_14 Sch=led[14]
#set_property -dict { PACKAGE_PIN V11 IOSTANDARD LVCMOS33 } [get_ports { LED[15] }]; #IO_L21N_T3_D05_A06_D22_14 Sch=led[15]
```

Esercizio 3: riconoscitore di sequenze

Esercizio 3.1

Progettare, implementare in VHDL e testare mediante simulazione una macchina in grado di riconoscere la sequenza **101**. La macchina prende in ingresso un segnale binario **i** che rappresenta il dato, un segnale **A** di temporizzazione e un segnale **M** di modo, che ne disciplina il funzionamento, e fornisce un'uscita **Y** alta quando la sequenza viene riconosciuta. In particolare,

- se **M=0**, la macchina valuta i bit seriali in ingresso a gruppi di 3 (sequenze non sovrapposte),
- se **M=1**, la macchina valuta i bit seriali in ingresso uno alla volta, tornando allo stato iniziale ogni volta che la sequenza viene correttamente riconosciuta (sequenze parzialmente sovrapposte).

Progetto e architettura

Il riconoscitore da noi progettato è una macchina sequenziale che presenta il seguente schema:

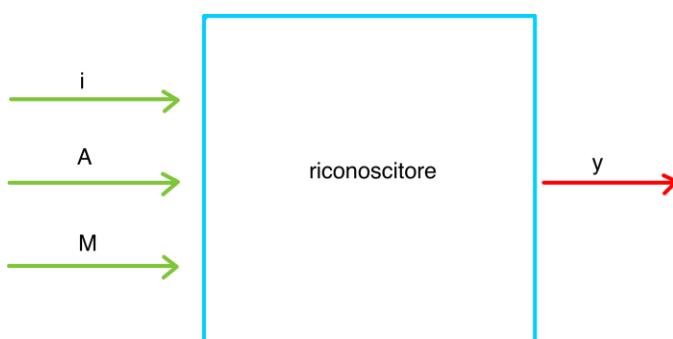


Figura 17 schema riconoscitore

L'architettura del riconoscitore è stata progettata per identificare la sequenza binaria "101".

I segnali in input del riconoscitore sono 3:

- **i**: bit in ingresso che viene fornito al riconoscitore per essere analizzato.
- **A**: il segnale di temporizzazione
- **M**: il segnale che indica il modo in cui deve essere effettuato il riconoscimento della sequenza ovvero se bisogna seguire un approccio di riconoscimento non sovrapposto o parzialmente sovrapposto

Prima di passare al codice abbiamo definito gli **automi a stati finiti** del riconoscitore in entrambe le modalità. In questo modo la scrittura del codice è risultata più semplice e priva di errori.

Iniziamo ad analizzare il caso in cui $M=0$ ovvero il caso in cui bisogna effettuare un riconoscimento non sovrapposto. L'automa è il seguente:

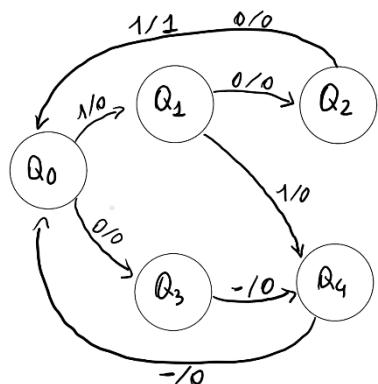


Figura 18 automa stati finiti modalità non sovrapposta

Analizziamo i vari stati:

- **Q0**: Stato iniziale. Se il segnale di ingresso (i) diventa '1', passa a $Q1$; se è '0', passa a $Q3$. In entrambi i casi, l'uscita è '0'.
- **Q1**: Se il segnale di ingresso (i) diventa '0', passa a $Q2$; se è '1', passa a $Q4$. In entrambi i casi, l'uscita è '0'.
- **Q2**: Se i è '1', ritorna a $Q0$ con l'uscita pari a '1'; altrimenti l'uscita è pari a '0'.
- **Q3**: Passa a $Q4$ con uscita pari a '0'.
- **Q4**: Ritorna a $Q0$ con uscita pari a '0'.

Dopo il riconoscimento della sequenza "101", la macchina ritorna allo stato iniziale $Q0$ per essere pronta a riconoscere nuove sequenze. Questa implementazione permette al riconoscitore di funzionare ciclicamente, analizzando sequenze di tre bit e riconoscendo la sequenza target durante il processo.

L'automa nel caso in cui $M=1$ e quindi il riconoscitore funziona in modalità parzialmente sovrapposta è il seguente:

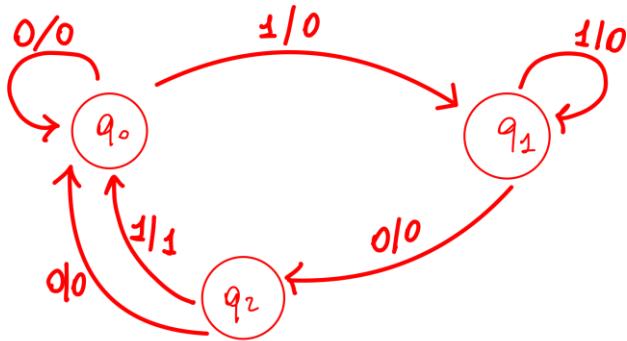


Figura 19 automa stati finiti modalità parzialmente sovrapposta

- **Stato Iniziale (Q0):** inizia in Q0 quando inizia l'analisi di una nuova sequenza di bit. Se il bit di ingresso è '1', la macchina transita a Q1 con y pari a '0'. Altrimenti, rimane in Q0 con y pari a '0'.
- **Stato Q1:** se la macchina è in Q1 e il bit successivo è '0', passa a Q2 con y pari a '0'. Se il bit successivo è pari a '1' resta in Q1 con y pari a '0'.
- **Stato Q2:** se il terzo bit è '1', la sequenza "101" è stata riconosciuta con successo. La macchina ritorna allo stato iniziale Q0 con y pari a '1'. Altrimenti, torna a Q0 con y pari a '0'.

In questo caso, quando M=1, l'automa a stati finiti valuta i bit seriali uno alla volta, tornando allo stato iniziale Q0 ogni volta che la sequenza "101" viene correttamente riconosciuta (anche in modo parziale). Questa implementazione consente alla macchina di essere più flessibile nel riconoscere sequenze parzialmente sovrapposte, come richiesto dalla modalità M=1.

Implementazione

La prima cosa che abbiamo fatto è stato definire l'entity del riconoscitore:

```

entity riconoscitore is
  port(
    i : in std_logic;
    y : out std_logic;
    A : in std_logic;
    M : in std_logic
  );
end riconoscitore;

```

Figura 20 entity riconoscitore

Il riconoscitore ha tre segnali di input e uno di output dichiarati come port.

- **i:** è un segnale di ingresso di tipo *std_logic*. Rappresenta il bit binario in ingresso che viene fornito al riconoscitore per essere analizzato.
- **y:** è un segnale di uscita di tipo *std_logic*. Indica l'uscita del riconoscitore, che diventa alta ('1') quando la sequenza target viene riconosciuta.
- **A:** è un segnale di ingresso di tipo *std_logic*. Rappresenta il segnale di temporizzazione, indicando quando il riconoscitore deve effettuare una transizione di stato.
- **M:** è un segnale di ingresso di tipo *std_logic*. Indica la modalità di funzionamento del riconoscitore. Quando M=0, il riconoscitore valuta i bit in gruppi di 3. Quando M=1, valuta i bit uno alla volta, tornando allo stato iniziale ogni volta che la sequenza viene correttamente riconosciuta.

Per l'architettura abbiamo utilizzato un approccio *behaviorial*.

```

architecture behavioural of riconoscitore is
  type stato is (Q0, Q1, Q2, Q3, Q4);
  signal stato_corrente : stato := Q0;
  signal stato_prossimo : stato;
  signal y_prossima : std_logic := 'U';

```

Figura 21 architettura behavioral

Definiamo un nuovo tipo enumerato denominato **stato**. Questo tipo può assumere uno dei valori Q0, Q1, Q2, Q3 o Q4, rappresentando gli stati possibili del riconoscitore. Gli stati sono utilizzati per tracciare la sequenza di bit e il progresso nel processo di riconoscimento.

Successivamente dichiariamo i segnali di stato

- **stato_corrente**: Questo segnale rappresenta lo stato corrente del riconoscitore. È inizializzato con Q0, indicando che il riconoscitore inizia la valutazione di una nuova sequenza da uno stato iniziale.
- **stato_prossimo**: Questo segnale rappresenta il prossimo stato calcolato durante l'analisi del bit in ingresso e del segnale di modalità. È utilizzato per determinare il passaggio da uno stato all'altro.

Abbiamo due variabili per lo stato in quanto abbiamo deciso di usare due process separati: il primo combinatorio per descrivere la FSM, il secondo invece attivato dalla variazione del segnale di temporizzazione in modo da aggiornare lo stato.

- **y_prossima**: Questo segnale è un'uscita temporanea utilizzata per memorizzare il valore dell'uscita che sarà assegnato al segnale di uscita **y** una volta completata la transizione di stato, in maniera sincrona col fronte di salita del segnale di temporizzazione.

In seguito, definiamo un process sensibile ai cambiamenti dei segnali *stato_corrente*, *i* ed *M*.

```

state_uscita : process(stato_corrente,i, M)
begin
  case M is
    when '1' =>
      case stato_corrente is
        when Q0 =>
          if (i = '1') then
            stato_prossimo <= Q1;
            y_prossima <= '0';
          else
            stato_prossimo <= Q0;
            y_prossima <= '0';
          end if ;
        when Q1 =>
          if (i = '0') then
            stato_prossimo <= Q2;
            y_prossima <= '0';
          else
            stato_prossimo <= Q1;
            y_prossima <= '0';
          end if ;
        when Q2 =>
          if (i = '1') then
            stato_prossimo <= Q0;
            y_prossima <= '1';
          else
            stato_prossimo <= Q0;
            y_prossima <= '0';
          end if ;
        when others =>
          stato_prossimo <= Q0;
          y_prossima <= '0';
      end case;
    when '0' =>
      case stato_corrente is
        when Q0 =>
          if (i = '1') then
            stato_prossimo <= Q1;
            y_prossima <= '0';
          else
            stato_prossimo <= Q3;
            y_prossima <= '0';
          end if ;
        when Q1 =>
          if (i = '0') then
            stato_prossimo <= Q2;
            y_prossima <= '0';
          else
            stato_prossimo <= Q4;
            y_prossima <= '0';
          end if ;
        when Q2 =>
          if (i = '1') then
            stato_prossimo <= Q0;
            y_prossima <= '1';
          else
            stato_prossimo <= Q0;
            y_prossima <= '0';
          end if ;
        when Q3 =>
          stato_prossimo <= Q4;
          y_prossima <= '0';
        when Q4 =>
          stato_prossimo <= Q0;
          y_prossima <= '0';
        when others =>
          stato_prossimo <= Q0;
          y_prossima <= '0';
      end case;
  end case;
end process;

```

Figura 22 FSM

All'interno del processo, viene utilizzata una struttura **case** per gestire il comportamento del riconoscitore in base al valore del segnale di modalità **M** e allo stato corrente.

Per ogni stato e modalità, vengono definite le condizioni e le azioni da eseguire per il calcolo del prossimo stato e del valore dell'uscita **y_prossima**. Per le transizioni tra i vari stati seguiamo quanto specificato negli automi a stati finiti visti in precedenza.

Dopo questo processo ne definiamo un altro sensibile agli eventi del segnale di temporizzazione A:

```
memoria : process(A)
begin
    if (A'event and A = '1') then
        stato_corrente <= stato_prossimo;
        y <= y_prossima;
    end if ;
end process;
end behavioural;
```

Figura 23 process memoria

Simulazione

Per effettuare la simulazione è stato necessario scrivere il testbench.

La prima cosa che abbiamo fatto è stata dichiarare una entity. Si nota che il corpo della entity è vuoto, questo perché non rappresenta un oggetto da realizzare ma ci serve solo per effettuare la simulazione e verificare il corretto funzionamento del sistema. Il testbench non ha segnali di ingresso e di uscita in quanto non può essere istanziato da nessun blocco ma istanzia al suo interno altri blocchi per effettuarne il test.

Successivamente dichiariamo il componente da testare ovvero il riconoscitore.

```
architecture rtl of riconoscitore_tb is
component riconoscitore is
    port(
        i : in std_logic;
        y : out std_logic;
        A : in std_logic;
        M : in std_logic
    );
end component;
```

Figura 24 dichiarazione componente da testare

Dopo dichiariamo i segnali che vengono utilizzati come input e output.

```
signal input : std_logic := 'U';
signal output : std_logic := 'U';
signal tempo : std_logic := 'U';
signal modo : std_logic := 'U';
```

Figura 25 dichiarazione segnali

- **input:** questo segnale rappresenta il bit di ingresso (i) che viene fornito al componente **riconoscitore** durante la simulazione. È inizializzato con il valore '**U**' (Undefined) per indicare che il suo valore non

è specificato all'inizio della simulazione. Durante la simulazione, il valore di **input** sarà modificato per testare il comportamento del componente quando si verifica una variazione del bit di ingresso.

- **output:** questo segnale rappresenta l'uscita del componente **riconoscitore**, ovvero il bit **y** restituito dal componente. È inizializzato con il valore '**U**' per indicare che il valore di **output** non è specificato inizialmente. Durante la simulazione, il valore di **output** sarà monitorato per verificare se il componente produce l'uscita desiderata in risposta ai vari stimoli di input.
- **tempo:** questo segnale rappresenta il segnale di temporizzazione (**A**) fornito al componente **riconoscitore** durante la simulazione. È inizializzato con il valore '**U**' per indicare che il suo valore non è specificato all'inizio della simulazione. Durante la simulazione, il valore di **tempo** sarà modificato per controllare quando il componente deve effettuare una transizione di stato.
- **modo:** Questo segnale rappresenta il segnale di modalità (**M**) fornito al componente **riconoscitore**. È inizializzato con il valore '**U**' per indicare che il suo valore non è specificato inizialmente. Durante la simulazione, il valore di **modo** sarà modificato per controllare se il componente deve operare in modalità "parzialmente sovrapposta" (**M=1**) o "non sovrapposta" (**M=0**).

All'interno del blocco architetturale abbiamo l'istanziazione del componente da testare **sistema** utilizzando l'etichetta **dut** (design under test).

La clausola **port map** associa il segnale **input** alla porta **i**, il segnale **output** alla porta **y**, il segnale tempo alla porta **A** e il segnale modo alla porta **M** del componente **riconoscitore**.

```
begin
dut: riconoscitore
port map(
    i => input,
    y => output,
    A => tempo,
    M => modo
);
```

Figura 26 istanziazione componente

Viene dichiarato un processo **stim_proc** che rappresenta il generatore di stimoli del testbench.

```

stim_process : process
begin
--M = 1 parzialmente sovrapposta
    modo <= '1';
    tempo <= '0';
    wait for 5 ns;
    tempo <= '1';
    wait for 5 ns;
    tempo <= '0';

    wait for 5 ns;
    input <= '1';
    wait for 5 ns;
    tempo <= '1';
    wait for 5 ns;
    tempo <= '0';
    wait for 5 ns;
    input <= '0';
    wait for 5 ns;
    tempo <= '1';
    wait for 5 ns;
    tempo <= '0';
    wait for 5 ns;
    input <= '1';
    wait for 5 ns;
    tempo <= '1';
    wait for 5 ns;
    tempo <= '0';
    wait for 10 ns;
    wait;
end process;
end;

```

All'inizio del processo, viene impostata la modalità (**modo**) a '1', indicando che desideriamo testare il riconoscitore con una modalità parzialmente sovrapposta.

Il segnale di temporizzazione (**tempo**) viene inizializzato a '0'.

Dopo un ritardo di 5 ns, il segnale di temporizzazione passa a '1' per indicare un cambio di stato.

Successivamente, dopo ulteriori 5 ns, il segnale di temporizzazione ritorna a '0'. Questa sequenza simula una condizione in cui il riconoscitore opera in modalità parzialmente sovrapposta e attraversa varie transizioni di stato.

Seguono diverse assegnazioni di valori al segnale di input (**input**) con ritardi temporali di 5 ns tra ciascuna assegnazione.

Inizialmente, il segnale di input è '1', poi diventa '0' dopo 5 ns, e successivamente ritorna a '1' dopo altri 5 ns.

Questa sequenza di input simula una situazione in cui il componente **riconoscitore** riceve una sequenza binaria specifica durante il funzionamento in modalità parzialmente sovrapposta.

Dopodiché, il processo reimposta la modalità (**modo**) a '0', indicando che ora desideri testare il riconoscitore in modalità non sovrapposta.

Il segnale di temporizzazione (**tempo**) viene nuovamente impostato a '0'.

Dopo 5 ns, il segnale di temporizzazione passa a '1' per indicare un cambio di stato.

Ancora una volta, dopo altri 5 ns, il segnale di temporizzazione ritorna a '0'.

Questa parte del processo simula una transizione del riconoscitore dalla modalità parzialmente sovrapposta alla modalità non sovrapposta.

Vengono nuovamente assegnati valori al segnale di input (**input**) con ritardi temporali di 5 ns tra ciascuna assegnazione.

La sequenza di input simula il funzionamento del componente **riconoscitore** in modalità non sovrapposta, dove valuta i bit uno alla volta, tornando allo stato iniziale ogni volta che la sequenza viene riconosciuta.

Infine, viene inserito un comando **wait**, che sospende il processo in modo indefinito.

Avviando la simulazione otteniamo il seguente risultato, che conferma il corretto funzionamento del componente sotto esame:



Figura 27 simulazione

Come possiamo vedere dal grafico, il segnale di modo viene subito settato a ‘1’ mentre il segnale di tempo dopo 5ns viene settato anch’esso a ‘1’ per poi essere abbassato a ‘0’ dopo ulteriori 5ns.

Passati altri 5ns l’input viene posto a 1, notiamo che l’uscita rimane ‘0’ finché non si ha il riconoscimento di ‘101’ in modalità parzialmente sovrapposta. Dopo il riconoscimento il segnale M diventa ‘0’ e si va ad analizzare l’altra modalità di funzionamento del riconoscitore. I risultati ottenuti dalla simulazione sono coerenti con quanto richiesto e progettato.

Esercizio 3.2

Sintetizzare e implementare su board la rete sviluppata al punto precedente, utilizzando uno switch S1 per codificare l’input i e uno switch S2 per codificare il modo M, in combinazione con due bottoni B1 e B2 utilizzati rispettivamente per acquisire l’input da S1 e S2 in sincronismo con il segnale di temporizzazione A, che deve essere ottenuto a partire dal clock della board. Infine, l’uscita Y può essere codificata utilizzando un led.

Progetto e architettura

Per implementare l’architettura del riconoscitore sulla board abbiamo sviluppato il componente *boardIO*, che prende in ingresso i segnali dalla board tramite gli switch, in particolare il bit della sequenza e il bit del modo, per passarli al riconoscitore. Il componente abilita l’acquisizione di tali ingressi soltanto in corrispondenza di un segnale alto proveniente dagli opportuni bottoni. Infine, se la sequenza ‘101’ viene riconosciuta, si illumina un led. L’architettura progettata ha il seguente schema:

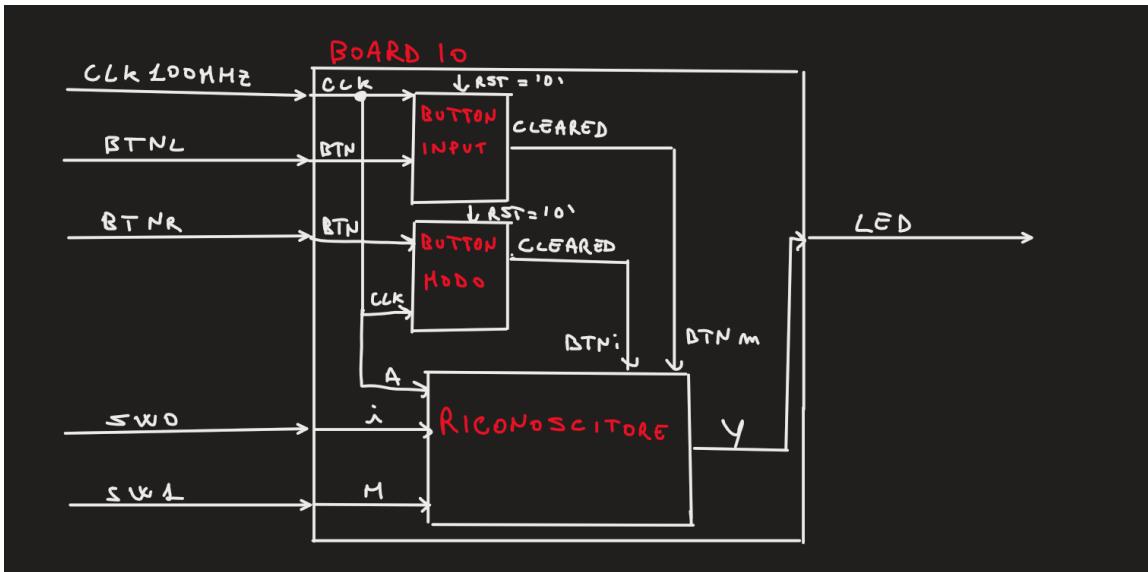


Figura 28 schema architettura

Implementazione

L'implementazione del riconoscitore è stata leggermente modificata, la differenza rispetto alla precedente è che in ingresso sono presenti i bottoni per abilitare l'acquisizione degli ingressi. Inoltre, sono stati istanziati i segnali di ingresso e modo per rendere possibile la lettura in corrispondenza delle abilitazioni. Il codice modificato è il seguente:

```

entity riconoscitore is
  port(
    btn_i : std_logic;
    btn_m : std_logic;
    i : in std_logic;
    y : out std_logic;
    A : in std_logic;
    M : in std_logic
  );
end riconoscitore;

architecture behavioural of riconoscitore is
  type stato is (Q0, Q1, Q2, Q3, Q4);
  signal stato_corrente : stato := Q0;
  signal stato_prossimo : stato;
  signal y_prossima : std_logic := '0';
  signal ingresso : std_logic := '0';
  signal modo : std_logic := '0';

  memoria : process(A)
  begin
    if (A'event and A = '1') then
      if btn_i = '1' then
        ingresso <= i;
        stato_corrente <= stato_prossimo;
      end if;
      if btn_m = '1' then
        modo <= m;
      end if;
      y <= y_prossima;
    end if ;
  end process;

```

Possiamo notare che ora l'ingresso viene letto soltanto in corrispondenza di un valore alto di "btn_i" e che il modo viene cambiato soltanto in corrispondenza di un valore alto di "btn_m".

Il componente boardIO invece si occupa di istanziare il riconoscitore e due botton debouncer, uno per l'ingresso e uno per il modo, la cui architettura è descritta nell'appendice. Nelle immagini seguenti è riportata l'implementazione di tale componente:

```
entity boardIO is
  Port (
    CLK100MHZ : in std_logic;
    BTNL : in std_logic;
    BTNR : in std_logic;
    SW0 : in std_logic;
    SW1 : in std_logic;
    LED : out std_logic
  );
end boardIO;
```

```
architecture Behavioral of boardIO is
  component riconoscitore is
    port(
      btn_i : std_logic;
      btn_m : std_logic;
      i : in std_logic;
      y : out std_logic;
      A : in std_logic;
      M : in std_logic
    );
  end component;

  component button_debouncer is
    generic (
      CLK_period: integer := 10; -- periodo del clock (della board) in nanosecondi
      btn_noise_time: integer := 10000000 -- durata stimata dell'oscillazione del bottone in nanosecondi
                                         | -- il valore di default ◆ 10 millisecondi
    );
    Port ( RST : in STD_LOGIC;
            CLK : in STD_LOGIC;
            BTN : in STD_LOGIC;
            CLEARED_BTN : out STD_LOGIC);
  end component;
  signal btn_ingresso, btn_modo : std_logic := '0';
begin
```

```
begin
  RICO : riconoscitore
  port map(
    btn_i => btn_ingresso,
    btn_m => btn_modo,
    i => SW0,
    y => LED,
    A => CLK100MHZ,
    M => SW1
  );
  BUTTON_INPUT : button_debouncer
  port map(
    RST => '0',
    clk => CLK100MHZ,
    BTN => BTNL,
    CLEARED_BTN => btn_ingresso
  );
  BUTTON_MODO : button_debouncer
  port map(
    RST => '0',
    clk => CLK100MHZ,
    BTN => BTNR,
    CLEARED_BTN => btn_modo
  );
end Behavioral;
```

Possiamo infine vedere il file modificato della board "**Nexys-A7-100T-Master.xdc**", dove sono stati mappati opportunamente gli ingressi e le uscite del componente boardIO:

```

## Clock signal
set_property -dict { PACKAGE_PIN E3      IOSTANDARD LVCMOS33 } [get_ports { CLK100MHZ }]; #IO_L12P_T1_MRCC_35 Sch=clk100mhz
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports {CLK100MHZ}];

##Switches
set_property -dict { PACKAGE_PIN J15      IOSTANDARD LVCMOS33 } [get_ports { SW0 }]; #IO_L24N_T3_RS0_15 Sch=sw[0]
set_property -dict { PACKAGE_PIN L16      IOSTANDARD LVCMOS33 } [get_ports { SW1 }]; #IO_L3N_T0_DQS_EMCCCLK_14 Sch=sw[1]

## LEDs
set_property -dict { PACKAGE_PIN H17      IOSTANDARD LVCMOS33 } [get_ports { LED }]; #IO_L18P_T2_A24_15 Sch=led[0]

##Buttons
#set_property -dict { PACKAGE_PIN C12      IOSTANDARD LVCMOS33 } [get_ports { CPU_RESETN }]; #IO_L3P_T0_DQS_AD1P_15 Sch=cpu_resetn
#set_property -dict { PACKAGE_PIN N17      IOSTANDARD LVCMOS33 } [get_ports { BTNC }]; #IO_L9P_T1_DQS_14 Sch=btnc
#set_property -dict { PACKAGE_PIN M18      IOSTANDARD LVCMOS33 } [get_ports { BTNU }]; #IO_L4N_T0_D05_14 Sch=btnu
set_property -dict { PACKAGE_PIN P17      IOSTANDARD LVCMOS33 } [get_ports { BTNL }]; #IO_L12P_T1_MRCC_14 Sch=btnl
set_property -dict { PACKAGE_PIN M17      IOSTANDARD LVCMOS33 } [get_ports { BTNR }]; #IO_L10N_T1_D15_14 Sch=btnr
#set_property -dict { PACKAGE_PIN P18      IOSTANDARD LVCMOS33 } [get_ports { BTND }]; #IO_L9N_T1_DQS_D13_14 Sch=btnd

```

Timing analysis

Un primary clock definisce un riferimento temporale per il progetto e viene utilizzato dal timing engine per derivare i requisiti del timing path.

```

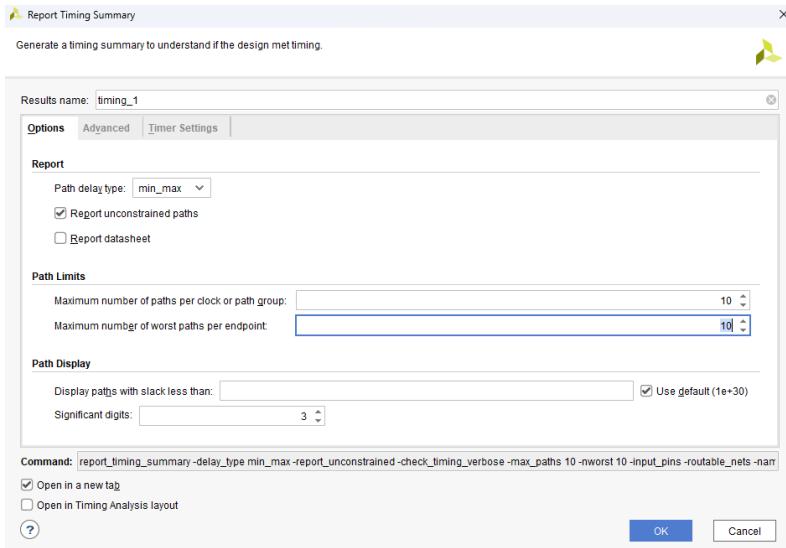
set_property -dict { PACKAGE_PIN E3      IOSTANDARD LVCMOS33 } [get_ports { CLK100MHZ }];
#IO_L12P_T1_MRCC_35 Sch=clk100mhz

create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports {CLK100MHZ}];

```

Con i comandi riportati sopra abbiamo creato un primary clock, con il nome di CLK100MHZ (nome di default all'interno del file Nexys-A7-100T-Master.xdc) con periodo di 10 ns e la forma d'onda desiderata: fronte di salita 0 ns e fronte di discesa 5 ns. Il clock è collegato al PIN E3 della board.

Per effettuare la timing analysis in Vivado è necessario prima di tutto effettuare la sintesi, che consentirebbe di eseguire già l'analisi ma con delle stime approssimate, quindi abbiamo effettuato anche l'implementazione. Nella schermata “Report Timing Summary” è possibile configurare i parametri della timing analysis, che specificano il tipo di report da generare e il contenuto da mostrare una volta eseguita l'analisi. In particolare, abbiamo selezionato **min** e **max**, come “Path delay type”, per misurare il **Worst Negative Slack (WNS)**, cioè il tempo che impiega un segnale di input a stabilizzarsi prima del fronte successivo del clock, tale che le uscite raggiungano il valore desiderato e il **Worst Hold Slack (WHS)**, cioè il tempo per cui un segnale di input deve restare stabile dopo il fronte del clock per consentire all'output di raggiungere il valore desiderato. Lo slack indica la differenza tra **require time** e **arrival time**. In questa analisi viene misurato anche il **Worst Pulse Width Slack (WPWS)** che indica il peggiore tra tutti i controlli considerando i ritardi sia minimi che massimi. Il parametro **maximum number of path** per clock indica quanti percorsi possono essere analizzati simultaneamente in un singolo ciclo di clock, mentre il **maximum number of worst paths** per endpoint consente di limitare il numero massimo di percorsi peggiori analizzati per ciascun endpoint.



È importante che il WNS sia positivo, perché altrimenti vuol dire che il percorso è fallito.

Di seguito sono riportati i risultati per quanto riguarda il clock creato precedentemente:

Design Timing Summary

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 5,027 ns	Worst Hold Slack (WHS): 0,235 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: 200	Total Number of Endpoints: 200	Total Number of Endpoints: 77

All user specified timing constraints are met.

Una volta controllato questo risultato, abbiamo calcolato anche la FMAX, cioè la frequenza massima di funzionamento. Essa non è fornita esplicitamente nei report ma l'abbiamo stimata con la seguente formula:

$\frac{1}{T - WNS}$, dove T è il periodo del clock target. Per trovare questo valore è possibile diminuire progressivamente il periodo del clock di design, finché non si ottiene un WNS negativo. In particolare, abbiamo diminuito il periodo fino a 3.5 ns con una forma d'onda con fronte di salita in 0 ns e fronte di discesa a 1.75 ns

-period 3.50 -waveform {0 1.75}

Il valore approssimato è 281 MHZ. Oltre questa frequenza i vincoli temporali non sono più rispettati.

Design Timing Summary

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): -0,054 ns	Worst Hold Slack (WHS): 0,250 ns	Worst Pulse Width Slack (WPWS): 1,250 ns
Total Negative Slack (TNS): -0,440 ns	Total Hold Slack (THS): 0,000 ns	Total Pulse Width Negative Slack (TPWS): 0,000 ns
Number of Failing Endpoints: 12	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 200	Total Number of Endpoints: 200	Total Number of Endpoints: 77

Timing constraints are not met.

Esercizio 4: Shift Register

Progettare, implementare in VHDL e testare mediante simulazione un registro a scorrimento di N bit in grado di shiftare a destra o a sinistra di un numero Y variabile di posizioni a seconda di una opportuna selezione. In particolare, i valori possibili di Y sono 1 e 2. L'utente tramite selezione deve scegliere di quante posizioni shiftare. Il componente deve essere realizzato utilizzando sia a) approccio comportamentale sia b) approccio strutturale.

Nota: il numero di bit del registro deve essere implementato come un generic, e dall'esterno deve poter essere scelta la modalità di funzionamento mediante opportuni segnali di selezione.

Architettura e progetto

Lo shift register da noi progettato presenta la seguente architettura blackbox:

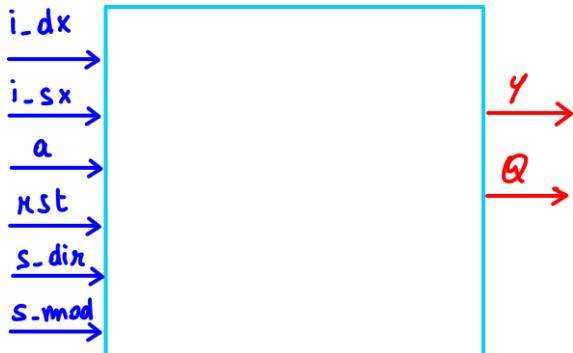


Figura 29 architettura blackbox

Gli ingressi sono 6: due di questi sono gli ingressi in base alla direzione (**i_dx** e **i_sx**), due segnali che vengono posti in ingresso ai multiplexer per decidere la modalità di scorrimento (**s_dir** e **s_mod**) e il segnale di temporizzazione **a** e il segnale di reset.

Le uscite sono due: **y** rappresenta il bit più significativo mentre **Q** viene utilizzato per fornire l'intero stato del registro a scorrimento includendo tutti i suoi bit.

Il registro è composto da 4 flip-flop che memorizzano i dati e 4 multiplexer 4:1. Ogni multiplexer è responsabile di un bit del registro a scorrimento.

I multiplexer servono a porre in ingresso il segnale in base a quanto deciso in base al valore dell'ingresso di selezione.

I multiplexer da noi progettati presentano due linee di selezione: una per la direzione (desta o sinistra) di scorrimento e una per la modalità (di 1 o 2 bit).

Lo schema dell'architettura del registro è il seguente:

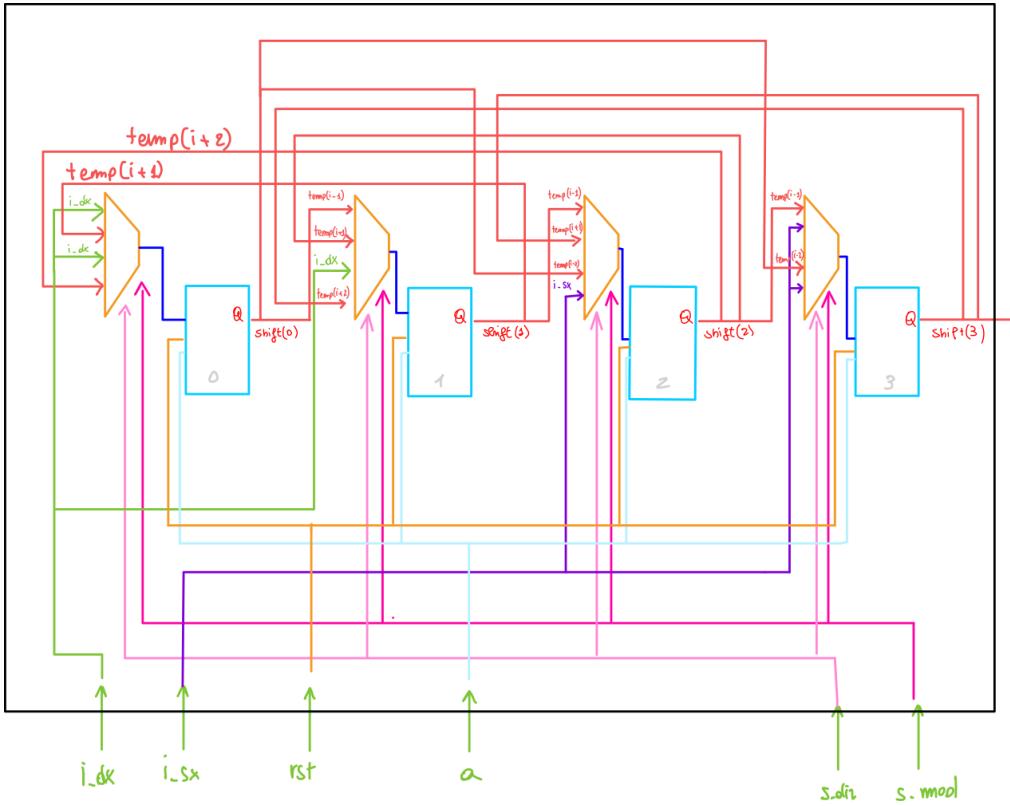


Figura 30 architettura shift register

Nello schema temp, shift e Q rappresentano i bit del registro.

Implementazione

Il primo passo fatto è stata la definizione dell'entity dello shift register:

```
entity shiftRegister is
  generic(
    N : positive := 4
  );
  port(
    i_dx: in std_logic;
    i_sx: in std_logic;
    a : in std_logic;
    rst : in std_logic;
    s_dir : in std_logic;
    s_mod : in std_logic;
    y : out std_logic;
    Q : out std_logic_vector(N-1 downto 0)
  );
end shiftRegister;
```

Figura 31 definizione entity shift register

Dichiariamo come parametro di tipo generic N che rappresenta la lunghezza del registro a scorrimento.

Gli ingressi e le uscite del componente sono i seguenti:

- **i_dx** e **i_sx** sono segnali di ingresso di tipo *std_logic*, in base alla direzione di shift a destra e a sinistra.
- **a** è un segnale di ingresso di tipo *std_logic*, utilizzato per la temporizzazione.
- **rst** è un segnale di ingresso di tipo *std_logic*, utilizzato per il reset del registro.
- **s_dir** è un segnale di ingresso di tipo *std_logic*, che controlla la direzione di shift.
- **s_mod** è un segnale di ingresso di tipo *std_logic*, utilizzato per selezionare la modalità di shift (1 o 2 bit).

- **y** è un segnale di uscita di tipo *std_logic*, rappresentante il bit più significativo del registro a scorrimento.
- **Q** è un segnale di uscita di tipo *std_logic_vector* con dimensione **N**, che rappresenta l'intero stato del registro a scorrimento.

Per l'implementazione abbiamo utilizzato due approcci, sia di tipo *behavioral* che *structural*.

Nella parte in cui si è utilizzato un approccio behavioral abbiamo descritto la logica dello shift nel registro.

```
architecture behavioral of shiftRegister is

  signal shift : std_logic_vector(N-1 downto 0);
begin
  ingresso_uscita : process(a)
  begin
    case s_mod is
      --Caso 1 bit
      when '0' =>
        case s_dir is
          --Caso 1 bit e shift a destra
          when '0' =>
            if (rst = '1') then
              shift <= (others => '0');
            elsif(a'event and a ='1') then
              shift <= shift(N-2 downto 0) & i_dx;
            end if ;
          --Caso 1 bit e shift a sinistra
          when '1' =>
            if (rst = '1') then
              shift <= (others => '0');
            elsif(a'event and a ='1') then
              shift <= i_sx & shift(N-1 downto 1);
            end if;
          when others =>
            shift <= (others => '0');
        end case ;
      --Caso 2 bit
      when '1' =>
        case s_dir is
          --Caso 2 bit e shift a destra
          when '0' =>
            if (rst = '1') then
              shift <= (others => '0');
            elsif(a'event and a ='1') then
              shift <= shift(N-3 downto 0) & i_dx & i_dx;
            end if ;
          --Caso 2 bit e shift a sinistra
          when '1' =>
            if (rst = '1') then
              shift <= (others => '0');
            elsif(a'event and a ='1') then
              shift <= i_sx & i_sx & shift(N-1 downto 2);
            end if ;
          when others =>
            shift <= (others => '0');
        end case ;
    end process;
    Q <= shift;
    y <= shift(N-1);
  end behavioral;
```

Viene dichiarato un segnale *shift*, un vettore di N bit, rappresentante il contenuto del registro a scorrimento. Successivamente si ha l'avvio di un processo *ingresso_uscita*.

Il processo **ingresso_uscita** è attivato dal fronte di salita del segnale di temporizzazione **a** ed è responsabile di gestire il comportamento di shift del registro in base ai segnali di controllo **s_mod** e **s_dir**. Analizziamo i casi principali:

1. Modalità di Shift a 1 Bit (**s_mod = '0'**):

In questo caso, il processo valuta la direzione di shift (**s_dir**).

- Se **s_dir = '0'**, il registro esegue uno shift a destra, caricando il bit in ingresso (**i_dx**) nel bit meno significativo.
- Se **s_dir = '1'**, il registro esegue uno shift a sinistra, caricando il bit in ingresso (**i_sx**) nel bit più significativo.

2. Modalità di Shift a 2 Bit (**s_mod = '1'**):

In questo caso, il processo valuta la direzione di shift (**s_dir**) e esegue uno shift a destra o a sinistra di due posizioni, a seconda di **s_dir**.

3. Azzeramento del Registro:

Se la modalità di shift non è né 1 né 2, il registro viene azzerato.

Le uscite **Q** e **y** del modulo vengono quindi assegnate in base allo stato corrente del vettore **shift**, che rappresenta il contenuto del registro.

Adesso andiamo ad analizzare l'architettura strutturale.

```

architecture structural of shiftRegister is

    component mux_4_1 is
        port( b0 : in STD_LOGIC;
              b1 : in STD_LOGIC;
              b2 : in STD_LOGIC;
              b3 : in STD_LOGIC;
              s0 : in STD_LOGIC;
              s1 : in STD_LOGIC;
              y0 : out STD_LOGIC
        );
    end component;

    signal shift : std_logic_vector(N-1 downto 0) := (others => '0');
    signal temp : std_logic_vector(N-1 downto 0):= (others => '0');

```

L'architettura **structural** del modulo **shiftRegister** utilizza componenti dichiarati precedentemente, in particolare il componente **mux_4_1**, per costruire la logica del registro a scorrimento.

Sono dichiarati due segnali: **shift**, che rappresenta il contenuto corrente del registro a scorrimento, e **temp**, utilizzato per memorizzare temporaneamente il valore di **shift** durante le operazioni.

Successivamente implementiamo la rete combinatoria:

```

begin
    --RETE COMBINATORIA
    MUX0toN : for i in 0 to N-1 generate
        primo : if i = 0 generate
            MUX0 : mux_4_1 port map(
                i_dx, temp(i+1), i_dx, temp(i+2), s_mod, s_dir, shift(i)
            );
        end generate;

        secondo : if i = 1 generate
            MUX1 : mux_4_1 port map(
                temp(i-1), temp(i+1), i_dx, temp(i+2), s_mod, s_dir, shift(i)
            );
        end generate;

        intermedi : if i > 1 and i < N-2 generate
            MUX_INTER : mux_4_1 port map(
                temp(i-1), temp(i+1), temp(i-2), temp(i+2), s_mod, s_dir, shift(i)
            );
        end generate;

        penultimo : if i = N-2 generate
            MUX_N_2 : mux_4_1 port map(
                temp(i-1), temp(i+1), temp(i-2), i_sx, s_mod, s_dir, shift(i)
            );
        end generate;

        ultimo : if i = N-1 generate
            MUX_N_1 : mux_4_1 port map(
                temp(i-1), i_sx, temp(i-2), i_sx, s_mod, s_dir, shift(i)
            );
        end generate;
    end generate;

```

Figura 32 rete combinatoria

Il ciclo **for** inizia da 0 e va fino a **N-1**, generando istanze multiple di multiplexer in base al valore di **i**. Ogni iterazione del ciclo genera un blocco di istruzioni che istanzia un multiplexer.

Vediamo le varie istanziazioni dei multiplexer:

- **Primo Multiplexer (MUX0):** viene istanziato un multiplexer con l'etichetta **MUX0** quando **i=0**. Questo multiplexer prende in ingresso i segnali **i_dx**, **temp(i+1)**, **i_dx**, **temp(i+2)**, e li seleziona in base alla modalità di scorrimento dei bit di selezione.

- **Secondo Multiplexer (MUX1):** analogamente, viene istanziato un secondo multiplexer (**MUX1**) quando **i = 1**. Questo multiplexer connette segnali in base alla logica specificata per **i = 1**.
- **Multiplexer Intermedi (MUX_INTER):** i multiplexer intermedi (**MUX_INTER**) sono generati quando **i** è compreso tra 1 e **N-2**. Questi multiplexer connettono segnali in base alla posizione corrente del ciclo, ad esempio, **temp(i-1)**, **temp(i+1)**, **temp(i-2)**, **temp(i+2)**, ecc.
- **Penultimo e Ultimo Multiplexer (MUX_N_2 e MUX_N_1):** Due multiplexer finali (**MUX_N_2** e **MUX_N_1**) sono istanziati quando **i** raggiunge **N-2** e **N-1** rispettivamente. Questi connettono segnali in base alla logica specificata per le posizioni finali del registro.

Dopo si ha la definizione di un ulteriore processo *memoria_reset*.

```
memoria_reset : process(a) begin
    if (a'event and a = '1') then
        if (rst = '1') then
            Q <= (others => '0');
            temp <= (others => '0');
        else
            temp <= shift;
            Q <= shift;
            y <= shift(N-1);
        end if;
    end if;
end process;
```

Figura 33 process memoria reset

Questo processo è responsabile della gestione dell'aggiornamento dell'uscita del registro a scorrimento, sul fronte di salita del segnale di temporizzazione.

Successivamente è presente un secondo blocco condizionale dove viene verificato se il segnale di reset sia uguale a '1'. Se il reset è attivo, allora si imposta l'uscita **Q** a tutti zeri ((**others => '0'**)), azzerando così il contenuto del registro.

Il contenuto corrente del registro (**shift**) viene copiato nella variabile temporanea **temp**. Questo passo è cruciale per conservare lo stato corrente del registro prima di apportare eventuali modifiche.

Infine, l'uscita **Q** del modulo viene aggiornata con il contenuto corrente del registro, garantendo che rifletta sempre lo stato corrente. Inoltre, l'uscita **y** viene impostata uguale all'ultimo bit del registro (**shift(N-1)**).

Simulazione

```
entity shiftRegister_tb is
end shiftRegister_tb;

architecture rtl of shiftRegister_tb is

    component shiftRegister is
        generic(
            N : positive := 4
        );
        port(
            i_dx: in std_logic;
            i_sx: in std_logic;
            a : in std_logic;
            rst : in std_logic;
            s_dir : in std_logic;
            s_mod : in std_logic;
            y : out std_logic;
            Q : out std_logic_vector(N-1 downto 0)
        );
    end component;
    FOR ALL : shiftRegister USE ENTITY WORK.shiftRegister (structural);
```

Figura 34 architettura rtl del testbench - dichiarazione componente sotto test

Utilizziamo l'istruzione **FOR ALL** per istanziare il componente **shiftRegister** e specifica che si desidera utilizzare l'entità **structural** del modulo **shiftRegister**. Questo significa che stiamo utilizzando l'architettura strutturale del modulo per il testbench.

Figura 35 dichiarazione dei segnali

I segnali dichiarati sono i seguenti:

- **input_dx, input_sx, output**: queste sono variabili di tipo **std_logic** che rappresentano i segnali di input e output del sistema. **input_dx** e **input_sx** rappresentano i segnali di input da destra e sinistra, rispettivamente. **output** rappresenta il segnale di output. Tutti sono inizializzati a '0'.
- **clk, dir, modo**: queste sono variabili di tipo **std_logic** utilizzate per rappresentare il segnale di clock (**clk**), il segnale di direzione (**dir**), e il segnale di modalità (**modo**). Anche queste sono inizializzate a 'U'.
- **reset**: questa variabile di tipo **std_logic** rappresenta il segnale di reset. È inizializzata a '0', indicando che il sistema è in uno stato non di reset.
- **registro**: questa variabile di tipo **std_logic_vector(3 downto 0)** rappresenta un registro a 4 bit. Inizializzato con tutti '0'.
- **test_sequence**: questa variabile di tipo **std_logic_vector(0 to 9)** rappresenta una sequenza di test. Contiene la stringa binaria "1011000110". Può essere utilizzata per applicare una sequenza di ingressi al sistema durante la simulazione e osservare il comportamento del sistema in risposta a questa sequenza.
- **PERIOD** invece è una costante di tipo **time** e rappresenta il periodo del clock. È impostata a 100 nanosecondi, indicando la durata di ciascun periodo del clock nel sistema.

Successivamente creiamo un'istanza del componente shift register le etichettiamo come **dut**:

```
begin
dut : shiftRegister
port map(
    i_dx => input_dx,
    i_sx => input_sx,
    a => clk,
    rst => reset,
    s_dir => dir,
    s_mod => modo,
    y => output,
    Q => registro
);
```

Figura 36 istanza del componente sotto test

Collegiamo utilizzando il port map le porte del componente con i segnali dichiarati in precedenza.

- **i_dx => input_dx**: collega il segnale di input **i_dx** del componente al segnale **input_dx** del testbench.
- **i_sx => input_sx**: collega il segnale di input **i_sx** del componente al segnale **input_sx** del testbench.
- **a => clk**: collega il segnale di input **a** del componente al segnale **clk** del testbench.
- **rst => reset**: collega il segnale di input **rst** (reset) del componente al segnale **reset** del testbench.

- **s_dir => dir**: collega il segnale di input **s_dir** (direzione di shift) del componente al segnale **dir** del testbench.
- **s_mod => modo**: collega il segnale di input **s_mod** (modalità di shift) del componente al segnale **modo** del testbench.
- **y => output**: collega il segnale di output **y** del componente al segnale **output** del testbench.
- **Q => registro**: collega il segnale di output **Q** (vettore di uscita) del componente al vettore **registro** del testbench.

Successivamente definiamo un process che genera un clock periodico:

```
clock: process begin
  clk <= '1';
  wait for PERIOD/2;
  clk <= '0';
  wait for PERIOD/2;
end process;
```

Figura 37 process del clock

Come ultima cosa andiamo a definire un process che genera le sequenze di input per poter effettuare la simulazione

```
inputs : process begin
  --1 bit shift a destra
  reset <= '1';
  modo <= '0';
  dir <= '0';
  wait for PERIOD;
  for i in 0 to 9 loop
    reset <= '0';
    input_dx <= test_sequence(i);
    wait for PERIOD;
  end loop;

  wait for 3*PERIOD/2;

  --1 bit shift a sinistra
  reset <= '1';
  modo <= '0';
  dir <= '1';
  wait for PERIOD;
  for i in 0 to 9 loop
    reset <= '0';
    input_sx <= test_sequence(i);
    wait for PERIOD;
  end loop;
  wait for PERIOD;

  --2 bit shift a destra
  reset <= '1';
  modo <= '1';
  dir <= '0';
  wait for PERIOD;
  for i in 0 to 9 loop
    reset <= '0';
    input_dx <= test_sequence(i);
    wait for PERIOD;
  end loop;
  wait;
end process;
```

Questo processo simula tutte le operazioni di shift a destra e sinistra a 1 e 2 bit, generando sequenze di input per il componente.

Avviando la simulazione avremo il seguente risultato:

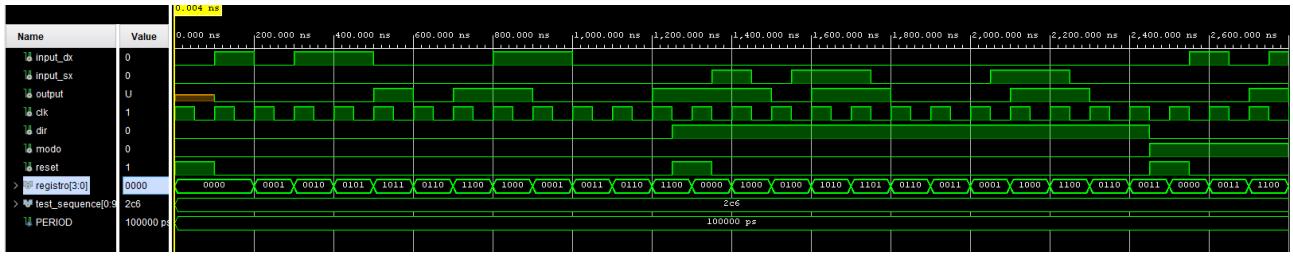


Figura 38: Simulazione 1 bit prima a destra e poi a sinistra

Prima di ogni nuovo caso di test viene ripulito il registro alzando il segnale di reset. Nel primo caso vediamo uno scorrimento a destra di 1 bit ($s_mod = 0$ e $s_dir = 0$). Dopo il secondo reset passiamo allo scorrimento a sinistra di 1 bit ($s_mod = 0$ e $s_dir = 1$). Come si evince dalla figura il risultato riflette il comportamento atteso.

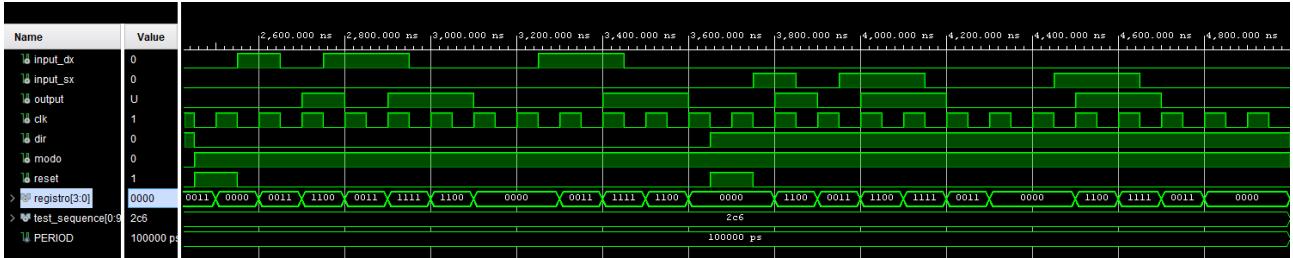


Figura 39: Simulazione 2 bit prima a destra e poi a sinistra

Stesso discorso vale per gli ultimi due casi in cui abbiamo uno shift a destra di 2 bit ($s_mod = 1$ e $s_dir = 0$) e shift a sinistra di 2 bit ($s_mod = 1$ e $s_dir = 1$).

Esercizio 5: Cronometro

Esercizio 5.1:

Progettare, implementare in VHDL e testare mediante simulazione un cronometro, in grado di scandire secondi, minuti e ore a partire da una base dei tempi prefissata (es. si consideri il clock a disposizione sulla board). Il progetto deve prevedere la possibilità di inizializzare il cronometro con un valore iniziale, sempre espresso in termini di ore, minuti e secondi, mediante un opportuno ingresso di *set*, e deve prevedere un ingresso di *reset* per azzerare il tempo.

Il componente deve essere realizzato utilizzando un approccio strutturale, collegando opportunamente dei contatori secondo uno schema a scelta

Progetto e architettura

L'approccio migliore per affrontare la complessità del cronometro è utilizzare un'architettura modulare. Invece di trattare il cronometro come un'unica entità complessa, suddivideremo il problema in parti più

gestibili. Ciascuna unità temporale (secondi, minuti e ore) sarà gestita da un contatore separato, rendendo il sistema più modulare e flessibile.

L'architettura blackbox è la seguente:



Figura 40 architettura cronometro

Per consentire un utilizzo flessibile del cronometro, è essenziale includere funzionalità di inizializzazione e caricamento. Gli utenti devono poter impostare manualmente il tempo iniziale del cronometro e avere la possibilità di azzerare il cronometro quando necessario.

I tre segnali di load servono per l'eventuale precaricamento del cronometro a un determinato valore iniziale definito da init. Le uscite sono tre ovvero secondi, minuti e ore.

Per implementare la funzione di conteggio del cronometro, è necessario utilizzare contatori modulo-n. Questi contatori sono in grado di contare fino a un valore massimo specificato (n) prima di ritornare a zero. Per esempio, il contatore dei secondi avrà un modulo di 60, mentre quelli dei minuti e delle ore avranno moduli rispettivamente di 60 e 24.

L'abilitazione di ciascun contatore deve essere gestita in modo sequenziale. Ad esempio, il contatore dei minuti non dovrebbe avanzare se il contatore dei secondi è in pausa. Progettiamo la logica di abilitazione in modo da rispettare questa sequenza gerarchica. Ciò implica l'uso di segnali di abilitazione intermedi che collegano i contatori in cascata.

Lo schema è il seguente:

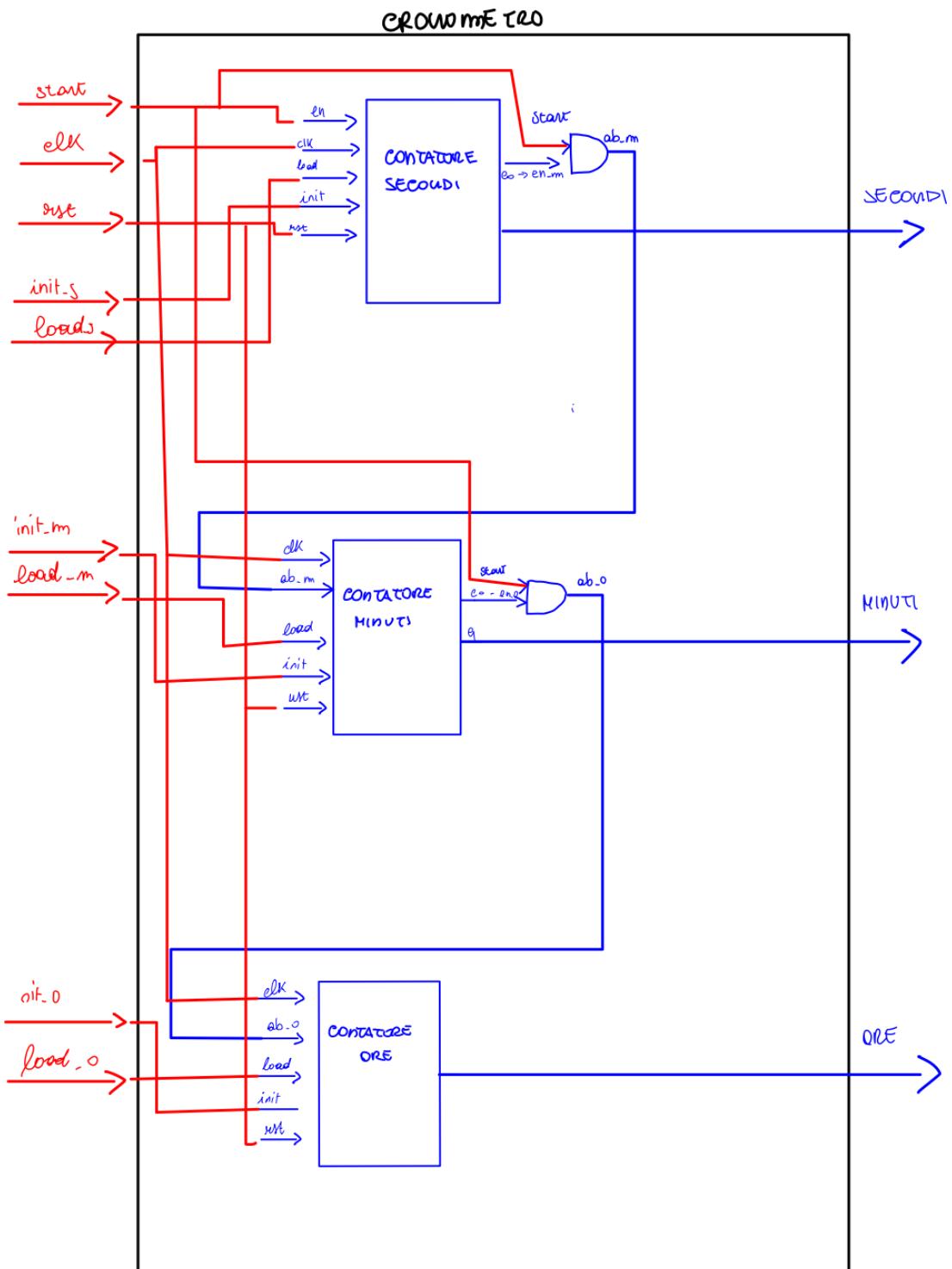


Figura 41 architettura cronometro

Implementazione

Andiamo a definire l'entity del nostro sistema cronometro:

```

entity cronometro is
    port (
        load_s : in std_logic;
        load_m : in std_logic;
        load_o : in std_logic;
        start : in std_logic;
        clk : in std_logic;
        rst : in std_logic;
        init_s : in std_logic_vector(5 downto 0);
        init_m : in std_logic_vector(5 downto 0);
        init_o : in std_logic_vector(4 downto 0);
        secondi : out std_logic_vector(5 downto 0);
        minuti : out std_logic_vector(5 downto 0);
        ore : out std_logic_vector(4 downto 0)
    );
end cronometro;

```

Figura 42 entity cronometro

- **load_s, load_m, load_o**: questi segnali di input indicano l'intenzione di caricare nuovi valori per i secondi, i minuti e le ore rispettivamente. Quando attivi, questi segnali influenzano i contatori corrispondenti del cronometro.
- **start**: questo segnale di input avvia o ferma il cronometro. Quando è attivo, il cronometro inizia a contare il tempo.
- **clk**: questo segnale di input rappresenta il clock del sistema e sincronizza le operazioni del cronometro.
- **rst**: questo segnale di input è utilizzato per azzerare il cronometro, riportando tutti i contatori a zero.
- **init_s, init_m, init_o**: questi segnali di input forniscono i valori iniziali per i contatori dei secondi, dei minuti e delle ore rispettivamente.
- **secondi**: questo segnale di output rappresenta il valore corrente del contatore dei secondi.
- **minuti**: questo segnale di output rappresenta il valore corrente del contatore dei minuti.
- **ore**: questo segnale di output rappresenta il valore corrente del contatore delle ore.

Successivamente andiamo a definire l'architettura utilizzando un approccio strutturale. Come prima cosa andiamo a definire il componente utilizzato dal cronometro ovvero il contatore modulo n. Non ci andremo a soffermare sulla sua implementazione per la quale rimandiamo all'appendice:

```
architecture structural of cronometro is

    component contatore_modn is
        generic(
            n : positive := 6;
            max : positive := 60
        );
        port(
            en : in std_logic;
            clk: in std_logic;
            load : in std_logic;
            init : in std_logic_vector(n-1 downto 0);
            rst : in std_logic;
            q : out std_logic_vector(n-1 downto 0);
            co : out std_logic
        );
    end component;

    signal en_m, en_o :std_logic;
    signal ab_m, ab_o :std_logic;
```

Figura 43 architettura strutturale

Dichiariamo anche i segnali che ci occorreranno per istanziare i vari componenti all'interno del nostro sistema. I segnali **en_m**, **en_o**, **ab_m**, e **ab_o** sono utilizzati per gestire l'abilitazione sequenziale dei contatori dei minuti e delle ore:

- **en_m e ab_m:**
 - **en_m:** è un segnale che indica il riporto del contatore dei secondi (contatore_secondi).
 - **ab_m:** è un segnale che rappresenta la logica AND tra il segnale di start (start) ed en_m. Questo significa che il contatore dei minuti è abilitato solo quando entrambi i segnali, start ed en_m, sono attivi contemporaneamente, ovvero quando i secondi hanno raggiunto il conteggio massimo e il cronometro è abilitato.
- **en_o e ab_o:**
 - **en_o:** è un segnale che indica il riporto del contatore dei minuti (contatore_minuti).
 - **ab_o:** è un segnale che rappresenta la logica AND tra il segnale di start (start) ed en_o. In questo modo, il contatore delle ore è abilitato solo quando entrambi i segnali, start ed en_o, sono attivi contemporaneamente.

Adesso istanziamo all'interno dell'architettura i tre contatori che andranno a costituire il nostro cronometro:

```
begin

    contatore_secondi : contatore_modn
    generic map(
        n => 6,
        max => 59
    )
    port map(
        en => start,
        clk => clk,
        load => load_s,
        init => init_s,
        rst => rst,
        q => secondi,
        co => en_m
    );
    ab_m <= start and en_m;
    contatore_minuti : contatore_modn
    generic map(
        n => 6,
        max => 59
    )
    port map(
        en => ab_m,
        clk => clk,
        load => load_m,
        init => init_m,
        rst => rst,
        q => minuti,
        co => en_o
    );
    ab_o <= start and en_o;
    contatore_ore : contatore_modn
    generic map(
        n => 5,
        max => 23
    )
    port map(
        en => ab_o,
        clk => clk,
        load => load_o,
        init => init_o,
        rst => rst,
        q => ore
    );
end structural;
```

Questa porzione di codice organizza gerarchicamente i contatori (contatore_modn) per i secondi, i minuti e le ore, assicurandosi che ciascun contatore inizi a contare solo quando il suo predecessore ha completato un ciclo. Questo approccio sequenziale è essenziale per la corretta misurazione del tempo in un cronometro.

Viene dichiarata un'istanza del componente contatore_modn con il nome **contatore_secondi**.

Il contatore dei secondi è configurato con un numero di bit del conteggio (n) pari a 6 e un valore massimo (max) di 59, che rappresenta il massimo conteggio.

Le porte del componente sono mappate ai segnali del cronometro. Ad esempio, il segnale di abilitazione (en) è collegato a start, il clock (clk) a clk, il segnale di caricamento (load) a load_s, il valore iniziale (init) a init_s, il segnale di reset (rst) a rst, l'uscita del contatore (q) a secondi, e il segnale di carry-out (co) a en_m.

Inoltre, viene definito il segnale ab_m che è il risultato della logica AND tra start ed en_m. Questo segnale sarà utilizzato per abilitare il contatore dei minuti.

Discorso analogo vale per gli altri contatori.

Simulazione

Per effettuare la simulazione è stato necessario scrivere il testbench.

```
entity cronometro_tb is
end cronometro_tb;

architecture rtl of cronometro_tb is

    component cronometro is
        port (
            load_s : in std_logic;
            load_m : in std_logic;
            load_o : in std_logic;
            start : in std_logic;
            clk : in std_logic;
            rst : in std_logic;
            init_s : in std_logic_vector(5 downto 0);
            init_m : in std_logic_vector(5 downto 0);
            init_o : in std_logic_vector(4 downto 0);
            secondi : out std_logic_vector(5 downto 0);
            minuti : out std_logic_vector(5 downto 0);
            ore : out std_logic_vector(4 downto 0)
        );
    end component;
```

Successivamente dichiariamo i segnali che utilizziamo per la simulazione:

```
signal load : std_logic := 'U';
signal start : std_logic := 'U';
signal clk : std_logic := 'U';
signal rst : std_logic := 'U';
signal init_s : std_logic_vector(5 downto 0) := (others => 'U');
signal init_m : std_logic_vector(5 downto 0) := (others => 'U');
signal init_o : std_logic_vector(4 downto 0) := (others => 'U');
signal s : std_logic_vector(5 downto 0) := (others => 'U');
signal m : std_logic_vector(5 downto 0) := (others => 'U');
signal o : std_logic_vector(4 downto 0) := (others => 'U');
constant SEMIPERIOD : time := 5 ns;
```

Figura 44 segnali di simulazione

- **load**: rappresenta l'input load del componente cronometro, abbiamo scelto di collegarlo a tutti i segnali di load dei contatori.
- **start**: rappresenta l'input start del componente cronometro
- **clk**: rappresenta l'input clk del componente cronometro.
- **rst**: rappresenta l'input rst del componente cronometro.
- **init_s**: rappresenta l'input init_s del componente cronometro, che è un vettore di bit di lunghezza 6.
- **init_m**: rappresenta l'input init_m del componente cronometro, che è un vettore di bit di lunghezza 6.
- **init_o**: rappresenta l'input init_o del componente cronometro, che è un vettore di bit di lunghezza 5.
- **s**: rappresenta l'output secondi del componente cronometro, che è un vettore di bit di lunghezza 6.
- **m**: rappresenta l'output minuti del componente cronometro, che è un vettore di bit di lunghezza 6.
- **o**: rappresenta l'output ore del componente cronometro, che è un vettore di bit di lunghezza 5.
- **constant SEMIPERIOD**: questa costante definisce la durata del semiperiodo del clock utilizzato nella simulazione del testbench. La durata è impostata a 5 nanosecondi.

Dopo aver fatto ciò, utilizzando questi segnali, andiamo ad istanziare il nostro componente sotto test:

```
begin
    dut: cronometro
    port map(
        load_s => load,
        load_m => load,
        load_o => load,
        start => start,
        clk => clk,
        rst => rst,
        init_s => init_s,
        init_m => init_m,
        init_o => init_o,
        secondi => s,
        minuti => m,
        ore => o
    );

```

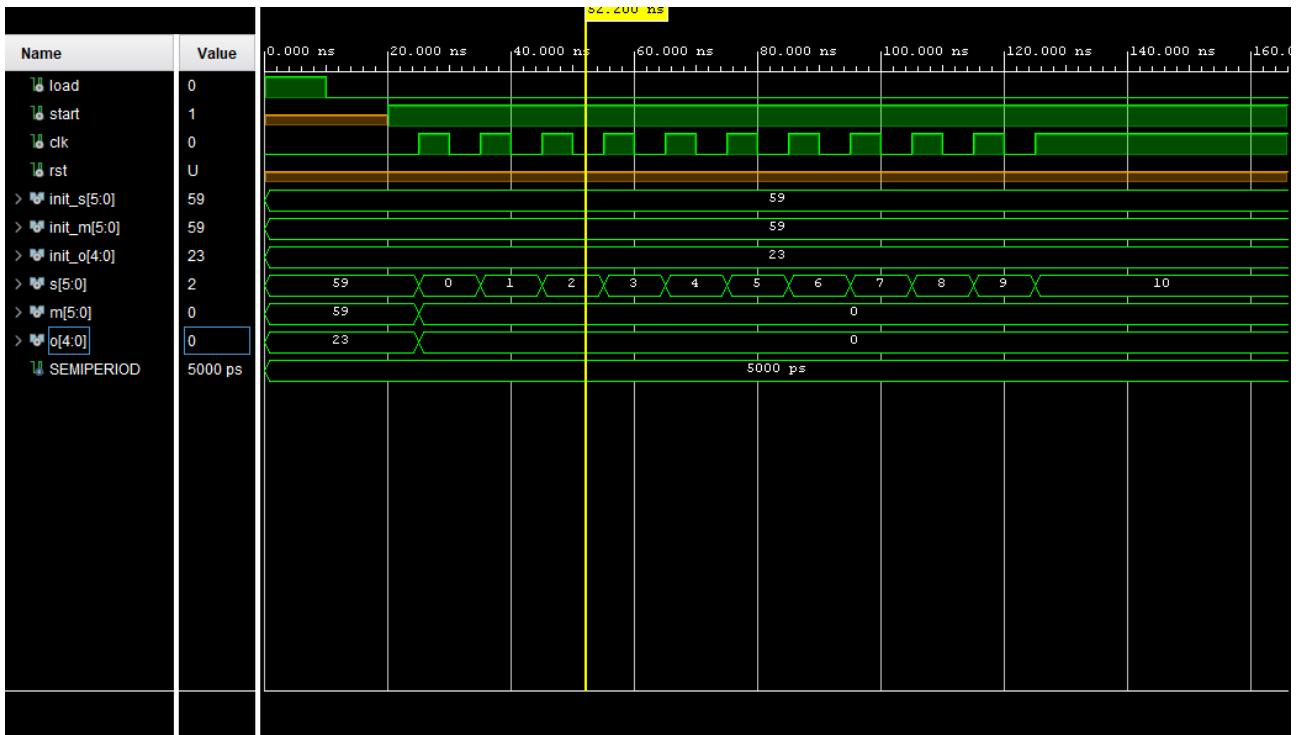
Definiamo poi il processo di simulazione:

```
stim_proc : process begin
    init_s <= "111011";
    init_m <= "111011";
    init_o <= "10111";
    clk <= '0';
    load <= '1';
    wait for 10 ns;
    load <= '0';
    wait for 10 ns;
    start <= '1';
    for i in 0 to 10 loop
        clk <= '0';
        wait for SEMIPERIOD;
        clk <= '1';
        wait for SEMIPERIOD;
    end loop;
    wait;
end process;

end rtl;
```

Questo processo genera una sequenza di segnali di input per simulare il funzionamento del cronometro. Imposta valori iniziali per le variabili di inizio (init_s, init_m, init_o), carica il cronometro con il segnale di load (load), attiva il segnale di start (start), e genera una serie di cicli di clock (clk) per simulare il passare del tempo.

Avviando la simulazione otteniamo il seguente risultato:



Nella nostra simulazione abbiamo voluto testare il caso in cui ci si trova a 23:59:59 e che quindi con il passare di un secondo bisogna azzerare il valore dei tre contatori.

Come possiamo osservare le inizializzazioni dei tre contatori avvengono correttamente, infatti, prima dell'avvio del segnale di star, abbiamo i valori di secondi, minuti e ore rispettivamente settati a 59, 59 e 23. Dopo 20ns il segnale di start si alza e si avvia il conteggio sul primo fronte di salita del clock. Quando ciò avviene osserviamo come tutti e 3 i contatori si azzerano proprio come previsto. In questo frammento di simulazione osserviamo come a ogni salita del fronte del clock si effettua l'incremento del valore del contatore dei secondi.

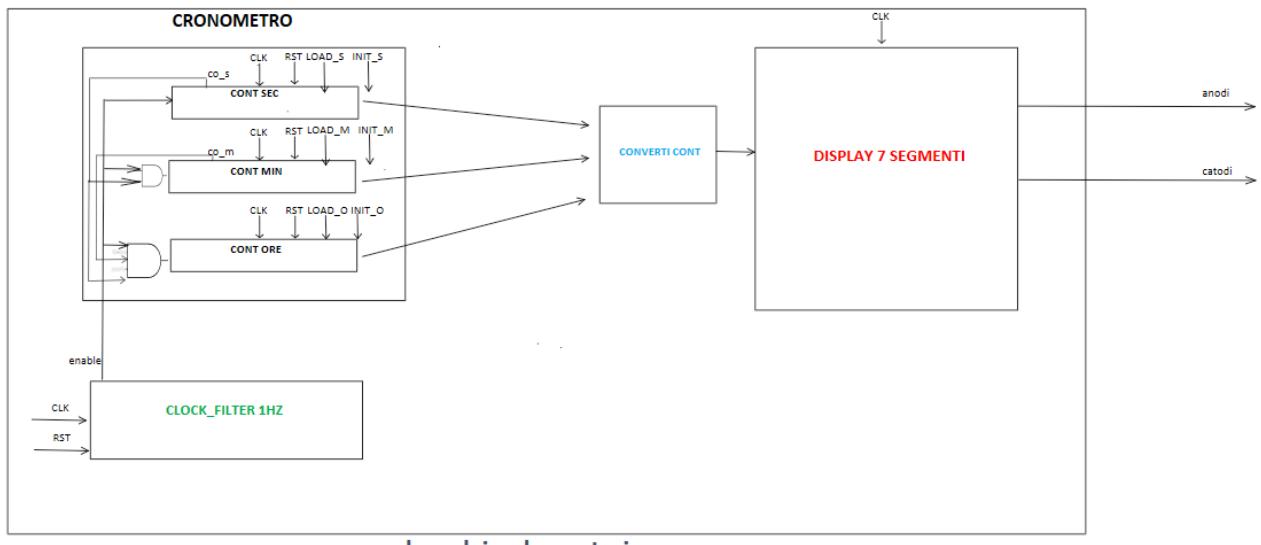
Esercizio 5.2

Sintetizzare ed implementare su board il componente sviluppato al punto precedente, utilizzando i display a 7 segmenti per la visualizzazione dell'orario (o una combinazione di display e led nel caso in cui i display a disposizione siano in numero inferiore a quello necessario), gli switch per l'immissione dell'orario iniziale e due bottoni, uno per il set dell'orario e uno per il reset. Si utilizzi una codifica a scelta dello studente per la visualizzazione dell'orario sui display (esadecimale o decimale).

Progetto e architettura

Dovendo utilizzare il display a 7 segmenti, per mostrare il cronometro su board, utilizziamo l'implementazione che ci è stata fornita. In questo caso il numero di cifre che vogliamo illuminare è 6, 2 per ogni parametro. Abbiamo scelto di accendere tutti i segmenti per visualizzare l'orario nel formato *ore - minuti - secondi* espresso in base 10. Quindi, utilizzando un approccio strutturale, abbiamo creato un componente che include il cronometro e il display, nonché 3 debouncer dei

bottoni utilizzati per il load di ore, secondi e minuti, e 1 debouncer per il reset. Le uscite dei cronometri devono essere opportunamente elaborate prima di andare in ingresso al display. È necessario concatenare i risultati dei secondi, minuti e ore per formare un'unica stringa e inizializzare a "1111" i bit corrispondenti ai trattini del formato specificato sopra. Essendo ogni cifra del display a 4 bit, abbiamo usato una cifra per rappresentare le decine e una per le unità. Questa elaborazione viene effettuata da una macchina combinatoria che prende in ingresso le uscite dei contatori, le converte in decine e unità e le concatena insieme ai bit dei trattini per formare un'unica stringa di ingresso per il display. Il risultato sarà una stringa del tipo *ore_decine + ore_unita + "1111" + minuti_decine + minuti_unità + "1111" + secondi_decine + secondi_unità*. Inoltre, per ridurre la frequenza del cronometro a 1Hz abbiamo utilizzato un secondo *clock_filter*, oltre a quello utilizzato dal display, in modo da abilitare correttamente i contatori del cronometro.



board_implementazione

Per non appesantire lo schema non abbiamo rappresentato anche i debouncer per i bottoni di load e reset. Il sistema completo prende in ingresso, attraverso gli switch, i bit di inizializzazione per i contatori, il clock della board a 100MHz e attraverso 4 bottoni i segnali di load e reset. In uscita abbiamo i segnali per i catodi e gli anodi del display. Come si vede dallo schema, l'abilitazione del contatore dei secondi corrisponde all'uscita del *clock_filter*, mentre per i minuti abbiamo una AND dello stesso segnale e il riporto del contatore dei secondi, e per le ore una AND dell'abilitazione dei minuti e il riporto del contatore dei minuti. In questo modo abbiamo ottenuto un aggiornamento corretto del cronometro alla frequenza di 1Hz. Segue il codice VHDL dei componenti, esclusi il display che ci è stato fornito, il debouncer presenti nell'appendice, e i contatori già visti prima.

Implementazione

La macchina combinatoria che elabora l'ingresso per il display è molto semplice. Estraie le decine e le unità delle uscite dei contatori utilizzando le operazioni di libreria *mod* e / rispettivamente, sfruttando alcune conversioni dei tipi di dato.

```

entity converti_count is
  Port (
    secondi : in std_logic_vector(5 downto 0);
    minuti : in std_logic_vector(5 downto 0);
    ore : in std_logic_vector(3 downto 0);
    outp : out std_logic_vector(31 downto 0)
  );
end converti_count;

architecture beh of converti_count is
  signal secondi_u : integer;
  signal secondi_d : integer;
  signal minuti_u : integer;
  signal minuti_d : integer;
  signal ore_u : integer;
  signal ore_d : integer;

  signal secondi_tot : std_logic_vector(7 downto 0);
  signal minuti_tot : std_logic_vector(7 downto 0);
  signal ore_tot : std_logic_vector(7 downto 0);
  signal uscita_temp : std_logic_vector(31 downto 0);

begin
  begin
    secondi_u <= to_integer(unsigned(secondi)) mod 10;
    secondi_d <= to_integer(unsigned(secondi)) / 10;
    minuti_u <= to_integer(unsigned(minuti)) mod 10;
    minuti_d <= to_integer(unsigned(minuti)) / 10;
    ore_u <= to_integer(unsigned(ore)) mod 10;
    ore_d <= to_integer(unsigned(ore)) / 10;

    secondi_tot(3 downto 0) <= std_logic_vector(to_unsigned(secondi_u, 4));
    secondi_tot(7 downto 4) <= std_logic_vector(to_unsigned(secondi_d, 4));

    minuti_tot(3 downto 0) <= std_logic_vector(to_unsigned(minuti_u, 4));
    minuti_tot(7 downto 4) <= std_logic_vector(to_unsigned(minuti_d, 4));

    ore_tot(3 downto 0) <= std_logic_vector(to_unsigned(ore_u, 4));
    ore_tot(7 downto 4) <= std_logic_vector(to_unsigned(ore_d, 4));

    uscita_temp(11 downto 8) <= "1111";
    uscita_temp(23 downto 20) <= "1111";
    uscita_temp(7 downto 0) <= secondi_tot;
    uscita_temp(19 downto 12) <= minuti_tot;
    uscita_temp(31 downto 24) <= ore_tot;

    outp <= uscita_temp;
  end;

```

Il file *board_implementazione* include, con approccio strutturale tutte le componenti, e costruisce i segnali di abilitazione per i contatori.

```

entity board_implementazione is
  Port (
    RST_tot : in STD_LOGIC;
    CLK_tot : in STD_LOGIC;
    anodes_out : out STD_LOGIC_VECTOR (7 downto 0);
    cathodes_out : out STD_LOGIC_VECTOR (7 downto 0);
    btn_s : in STD_LOGIC;
    btn_m : in STD_LOGIC;
    btn_o : in STD_LOGIC;
    init_s : in std_logic_vector(5 downto 0);
    init_m : in std_logic_vector(5 downto 0);
    init_o : in std_logic_vector(3 downto 0)
  );
end board_implementazione;

```

Vediamo come vengono istanziati i vari componenti:

```

signal o_co : std_logic;
signal m_co : std_logic;
signal s_co : std_logic;
signal abilita_minuti : std_logic;
signal abilita_ore : std_logic;

signal b_s_pulito : std_logic;
signal b_m_pulito : std_logic;
signal b_o_pulito : std_logic;
signal b_RST_pulito : std_logic;

signal val_temp : std_logic_vector(31 downto 0);

signal temp_s : std_logic_vector(5 downto 0) := (others => '0');
signal temp_m : std_logic_vector(5 downto 0) := (others => '0');
signal temp_o : std_logic_vector(3 downto 0) := (others => '0');

signal clock_BUONO : std_logic;

```

I primi 3 segnali rappresentano i riporti dei contatori, i successivi due sono le abilitazioni di ore e minuti. Il segnale *clock_buono* rappresenta l'uscita del *clock_filter*. I segnali del tipo *b_(nome bottone)_pulito* rappresentano le uscite dei debouncer. I segnali *temp_s/m/o* rappresentano le uscite dei contatori e *val_temp* l'uscita della macchina combinatoria che elabora l'ingresso del display, chiamata *converti_count*.

```

begin
    clk_filter: clock_filter
    generic map(
        CLKIN_freq => 100000000,
        CLKOUT_freq => 1
    )
    port map(
        clock_in => CLK_tot,
        reset => b_RST_pulito,
        clock_out => clock_BUONO
    );
    cont_secondi: contatore_modn
    generic map (
        max => 59,
        n => 6
    )
    port map (
        clk => CLK_tot,
        rst => b_RST_pulito,
        en => clock_BUONO,
        q => temp_s,
        co => s_co,
        init => init_s,
        btn_load => b_s_pulito
    );
    abilita_minuti <= clock_BUONO and s_co;
    cont_minuti: contatore_modn
    generic map (
        max => 59,
        n => 6
    )
    port map (
        clk => CLK_tot,
        rst => b_RST_pulito,
        en => abilita_minuti,
        q => temp_m,
        co => m_co,
        init => init_m,
        btn_load => b_m_pulito
    );
    abilita_ore <= abilita_minuti and m_co;
    cont_ore: contatore_modn
    generic map (
        max => 11,
        n => 4
    )
    port map (
        clk => CLK_tot,
        rst => b_RST_pulito,
        en => abilita_ore,
        q => temp_o,
        co => o_co,
        init => init_o,
        btn_load => b_o_pulito
    );
    seven_segment_array: display_seven_segments
    GENERIC MAP(
        CLKIN_freq => 10000000, --qui inserisco i parametri effettivi (clock della board e clock in uscita desiderato)
        CLKOUT_freq => 500 --inserendo un valore inferiore si vedranno le cifre illuminarsi in sequenza
    )
    PORT MAP(
        CLK => CLK_tot,
        RST => '0',
        value => val_temp,
        enable => "11111111", --stabilisco che tutti i display siano accesi
        dots => "00000000", --stabilisco che tutti i punti siano spenti
        anodes => anodes_out,
        cathodes => cathodes_out
    );

```

Esercizio 5.3

Estendere il componente sviluppato ai punti precedenti in modo che sia in grado di acquisire e memorizzare internamente fino ad un numero N di intertempi in corrispondenza di un ingresso di *stop*. Opzionalmente, il componente può prevedere una modalità di visualizzazione in cui, alla pressione di un bottone, vengano visualizzati sui display gli intertempi memorizzati (uno per ogni pressione).

Progetto e architettura

Abbiamo scelto di utilizzare una memoria per memorizzare gli N intertempi, indirizzata da un contatore che aggiorna l'indirizzo su cui scrivere gli intertempi ad ogni pressione del bottone di stop in uscita da un nuovo debouncer. Il contatore inoltre fornisce il segnale di *write* alla memoria per la scrittura all'indirizzo fornito. Inoltre, abbiamo deciso di visualizzare l'ultimo intertempo memorizzato attraverso i LED della board. Un'alternativa potrebbe essere utilizzare un banco di N shift register, ma nel caso in cui si volesse estendere l'implementazione in modo da visualizzare uno specifico intertempo, questo non sarebbe stato facilmente implementabile.

Implementazione

```

entity contatore_mem is
  generic(
    word_lenght : positive := 16;
    depth : positive := 2;
    numero_locazioni : positive := 4
  );
  Port (
    clk : in std_logic;
    btn_stop : in std_logic;
    address : out std_logic_vector(depth-1 downto 0);
    write: out std_logic
  );
end contatore_mem;

architecture Behavioral of contatore_mem is
signal count : std_logic_vector(depth-1 downto 0) := (others => '0');
begin
  process(clk) begin
    if (clk'event and clk = '1') then
      if btn_stop = '1' then
        write <= '1';
        address <= count;
        count <= std_logic_vector(unsigned(count) + 1);
      else
        write <= '0';
      end if;
    end if;
  end process;
end Behavioral;

```

Come si può vedere dal codice, in maniera sincrona, il segnale di stop alto determina un segnale di *write* alto, l'aggiornamento dell'indirizzo in ingresso alla memoria e l'incremento del contatore. I nuovi componenti vengono istanziati come segue:

```

intertempo <= temp_o & temp_m & temp_s;
cont_mem: contatore_mem
  generic map (
    word_lenght => 16,
    depth => 2,
    numero_locazioni => 4
  )
  port map (
    clk => CLK_tot,
    btn_stop => b_stop_pulito,
    address => address_mem,
    write => temp_w
  );

mem : memoria
generic map(
  word_lenght => 16,
  depth => 2,
  numero_locazioni => 4
)

port map(
  clk => CLK_tot,
  address => address_mem,
  stringa => intertempo,
  write => temp_w,
  data => LED
);

```

I segnali *temp_w* e *address_mem* servono semplicemente per collegare i due componenti. Il segnale *b_stop_debounce* rappresenta l'uscita del nuovo debouncer. Il segnale *intertempo* viene costruito concatenando le uscite dei contatori e rappresenta la stringa di bit da memorizzare in memoria.

Timing analysis

Design Timing Summary

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 3.572 ns	Worst Hold Slack (WHS): 0,151 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: 563	Total Number of Endpoints: 563	Total Number of Endpoints: 223

All user specified timing constraints are met.

Una volta controllato questo risultato, abbiamo calcolato anche la FMAX, cioè la frequenza massima di funzionamento. Essa non è fornita esplicitamente nei report ma l'abbiamo stimata con la seguente formula:

$\frac{1}{T - WNS}$, dove T è il periodo del clock target. Per trovare questo valore è possibile diminuire progressivamente il periodo del clock di design, finché non si ottiene un WNS negativo. In particolare, abbiamo diminuito il periodo fino a 3.5 ns con una forma d'onda con fronte di salita in 0 ns e fronte di discesa a 1.75 ns

-period 3.50 -waveform {0 1.75}

Il valore approssimato è 252 MHZ. Oltre questa frequenza i vincoli temporali non sono più rispettati.

Design Timing Summary

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): -0,457 ns	Worst Hold Slack (WHS): 0,143 ns	Worst Pulse Width Slack (WPWS): 1,250 ns
Total Negative Slack (TNS): -33,719 ns	Total Hold Slack (THS): 0,000 ns	Total Pulse Width Negative Slack (TPWS): 0,000 ns
Number of Failing Endpoints: 169	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 566	Total Number of Endpoints: 566	Total Number of Endpoints: 224

Timing constraints are not met.

Esercizio 6: Sistema di lettura-elaborazione-scrittura PO_PC

Esercizio 6.1

Progettare, implementare in VHDL e verificare mediante simulazione un sistema dotato di una memoria ROM di N locazioni da 8 bit ciascuna, una macchina combinatoria M in grado di trasformare (secondo una funzione a scelta dello studente) la stringa di 8 bit letta dalla ROM in una stringa di 4 bit, e una memoria MEM di N locazioni che memorizza la stringa in output da M.

Il sistema si avvia in corrispondenza di un segnale di START che viene fornito esternamente. Una volta avviato, tramite un'apposita unità di controllo che gestisce la tempificazione del sistema, viene scandita una locazione alla volta della ROM e viene scritta la corrispondente locazione di MEM. Gli indirizzi di memoria sono forniti da un contatore. Le memorie ROM e MEM hanno rispettivamente un read e un write sincrono.

Progetto e architettura

Il sistema che dobbiamo realizzare presenta tre segnali di input: inizio, fine e il clock. Il segnale di output è unico ed è il segnale che rappresenta il risultato finale, ovvero ciò che è stato scritto nella locazione della memoria. L'architettura blackbox è la seguente:

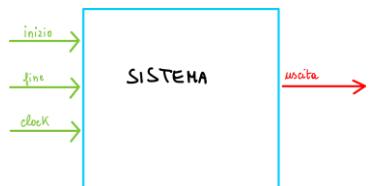


Figura 45 architettura black box del sistema

Per realizzare il sistema desiderato lo suddividiamo in tre parti: la memoria ROM, la macchina combinatoria M e la memoria MEM. Inoltre, è necessaria un'unità di controllo per gestire la tempificazione del sistema e un contatore per scandire le locazioni delle due memorie.

L'architettura del sistema è la seguente:

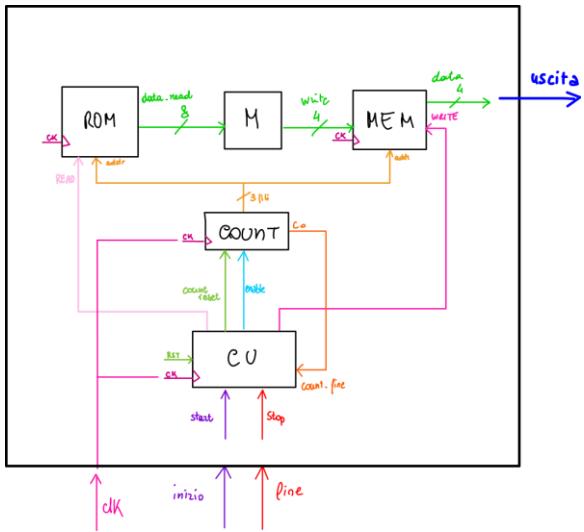


Figura 46 architettura sistema

Utilizziamo un contatore modulo 16 per generare sequenzialmente gli indirizzi per la lettura dalla ROM e la scrittura nella MEM. Il contatore viene abilitato e sincronizzato con il segnale di clock. Il segnale di reset lo riporta a zero.

L'**unità di controllo** scandisce le varie fasi del sistema, realizzando una macchina a stati finiti. Il segnale di inizio avvia il sistema, fine lo ferma e count_fine indica quando il contatore raggiunge il massimo e quindi il sistema deve essere fermato. L'unità di controllo sincronizza le operazioni in base al segnale di clock.

La ROM sequenziale legge i dati in base all'indirizzo e fornisce l'output del dato letto. Questo output viene trasformato da una stringa a 8 bit a una stringa a 4 bit utilizzando una macchina combinatoria. La macchina combinatoria restituisce in output il dato convertito in stringa da 4 bit da scrivere in memoria, in particolare abbiamo deciso di restituire tutti 1 in caso di stringa dispari e tutti 0 in caso di stringa pari.

La memoria scrive nella locazione specificata quando il segnale di scrittura è attivo.

Implementazione

Per l'implementazione del sistema abbiamo utilizzato un approccio strutturale.

Abbiamo implementato separatamente le varie componenti che costituiscono il sistema.

Iniziamo ad analizzare l'implementazione della control unit. Le implementazioni di macchina, rom sequenziale e contatore sono spiegate nell'appendice.

Come prima cosa abbiamo definito l'entity:

```

entity control_unit is
  port(
    start : in std_logic;
    stop : in std_logic;
    clk: in std_logic;
    count_fine : in std_logic;
    w: out std_logic;
    r: out std_logic;
    enable : out std_logic;
    count_rst: out std_logic
  );
end control_unit;

```

Figura 47 entity control unit

I segnali dichiarati come port sono i seguenti:

- **start**: questo segnale di ingresso indica l'intenzione di avviare il sistema. Quando **start** è attivo, l'unità di controllo deve avviare il sistema.
- **stop**: questo segnale di ingresso indica l'intenzione di fermare il sistema. Quando **stop** è attivo, l'unità di controllo deve fermare il sistema.
- **clk**: questo è il segnale di clock di ingresso che sincronizza le operazioni dell'unità di controllo.
- **count_fine**: questo segnale di ingresso indica che il contatore (utilizzato per generare gli indirizzi) ha raggiunto il massimo valore.
- **w**: questo è un segnale di uscita che indica alla memoria di dover scrivere nella locazione di memoria indicata dal contatore.
- **r**: questo è un segnale di uscita che indica alla rom di dover leggere dalla locazione di memoria indicata dal contatore.
- **enable**: Questo è un segnale di uscita che utilizziamo per abilitare il contatore.
- **count_rst**: Questo è un segnale di uscita utilizzato per resettare il contatore.

Per l'implementazione della control unit abbiamo utilizzato un approccio behavioral

```

architecture behavioral of control_unit is
begin
    state: process(clk) begin
        if clk'event and clk = '0' then
            state <= next_state;
        end if;
    end process;

    fsm: process(state, start, stop, count_fine) begin
        case state is
            when IDLE =>
                count_rst <= '0';
                if start = '1' then
                    next_state <= READ;
                    r <= '1';
                end if;
            when READ =>
                next_state <= OP;
                r <= '0';
            when OP =>
                next_state <= WRITE;
                w <= '1';
            when WRITE =>
                next_state <= COUNT;
                w <= '0';
                enable <= '1';
            when COUNT =>
                enable <= '0';
                if stop = '1' then
                    next_state <= IDLE;
                    count_rst <= '1';
                elsif count_fine = '1' then
                    next_state <= IDLE;
                    count_rst <= '1';
                else
                    next_state <= READ;
                    r <= '1';
                end if;
        end case;
    end process;
end behavioral;

```

Figura 48 architettura behavioral

Per prima cosa definiamo un tipo enumerativo **state_type** che rappresenta gli stati possibili della macchina a stati finiti. Gli stati sono IDLE, READ, OP, WRITE e COUNT.

Successivamente definiamo il segnale che rappresenta lo stato corrente della macchina a stati finiti, questo segnale lo inizializziamo a IDLE. Oltre a questo, inizializziamo il segnale che rappresenta lo stato successivo, anch'esso viene inizializzato a IDLE.

Dopo definiamo un processo sequenziale che sincronizza lo stato corrente con lo stato successivo al fronte di discesa del segnale di clock.

Il processo successivo implementa la logica della macchina a stati finiti. La transizione tra gli stati è gestita da un blocco case basato sullo stato corrente.

- Quando lo stato è IDLE, si verifica se il segnale **start** è attivo. Se è attivo, la macchina passa allo stato READ, attivando anche il segnale **r** per indicare l'inizio di un'operazione di lettura.
- Quando lo stato è READ, si passa allo stato OP e si abbassa il segnale **r**.
- Quando lo stato è OP, e si attende l'elaborazione della macchina, si passa allo stato WRITE e si attiva il segnale **w** per indicare l'inizio di un'operazione di scrittura.
- Quando lo stato è WRITE, si passa allo stato COUNT e si disattiva il segnale **w**, attivando anche il segnale **enable** per abilitare altre parti del sistema.

- Quando lo stato è COUNT, si disattiva il segnale **enable**. Se il segnale **stop** oppure è attivo **count_fine**, si ritorna allo stato IDLE, attivando anche il segnale **count_rst** per ripristinare il contatore.

Dopo aver analizzato questa implementazione andiamo ad analizzare il sistema.

Analogamente a prima andiamo a definire l'entity del sistema:

```
entity sistema is
    generic(
        word_length : positive := 4
    );
    port(
        inizio : in std_logic;
        fine : in std_logic;
        uscita : out std_logic_vector(word_length-1 downto 0);
        clock : in std_logic
        -- indirizzo_corrente : out std_logic_vector(word_length-1 downto 0) --debug
    );
end sistema;
```

Figura 49 entity sistema

Definiamo come parametro generic **word_length**, questo parametro indica la lunghezza intesa come numero di bit delle stringhe in ingresso e in uscita. Gli altri segnali sono i seguenti:

- **inizio**: input di tipo **std_logic** che rappresenta un segnale di avvio per il sistema.
- **fine**: input di tipo **std_logic** che rappresenta un segnale di fine per il sistema.
- **uscita**: output di tipo **std_logic_vector** con dimensione **word_length**. Rappresenta la stringa di output del sistema.
- **clock**: input di tipo **std_logic** che rappresenta il segnale di clock del sistema.

Come detto in precedenza abbiamo utilizzato un approccio structural. La prima cosa che abbiamo fatto è stata la definizione dei componenti implementati in precedenza.

Dopo aver fatto ciò passiamo alla definizione dei segnali che utilizziamo per effettuare le interconnessioni tra i vari componenti.

```
signal scrivi : std_logic;
signal leggi : std_logic;
signal reset conteggio : std_logic;
signal indirizzo : std_logic_vector(3 downto 0);
signal data_read : std_logic_vector(7 downto 0);
signal data_write : std_logic_vector(3 downto 0);
signal abilita conteggio : std_logic;
signal max_count : std_logic;
```

Figura 50 definizione segnali

- **scrivi**: segnale di controllo per indicare al componente **memoria** quando deve scrivere i dati. Quando è alto ('1'), il componente **memoria** è abilitato a scrivere i dati in ingresso alla locazione di memoria specificata dall'**indirizzo**.

- **leggi**: segnale di controllo per indicare al componente **rom_seq** quando deve leggere i dati. Quando è alto ('1'), il componente **rom_seq** è abilitato a leggere i dati dalla locazione di memoria specificata dall'**indirizzo**.
- **reset conteggio**: segnale di reset del contatore. Quando è alto ('1'), il contatore viene azzerato.
- **indirizzo**: vettore di segnali che rappresenta l'indirizzo corrente nella memoria. La lunghezza del vettore è 4 bit, indicando che può rappresentare fino a 16 locazioni di memoria.
- **data_read**: vettore di segnali che rappresenta i dati letti dalla rom.
- **data_write**: vettore di segnali che rappresenta i dati scritti nella memoria.
- **abilita conteggio**: segnale di controllo per abilitare/disabilitare il contatore. Quando è alto ('1'), il contatore è abilitato a contare.
- **max_count**: segnale che indica quando il contatore ha raggiunto il massimo valore.

Adesso colleghiamo le istanze dei diversi componenti e specifichiamo le interconnessioni tra essi.

```
begin
    contatore : contatore_mod16
    port map(
        en => abilita conteggio,
        clk => clock,
        rst => reset conteggio,
        q => indirizzo,
        co => max_count
    );

```

Figura 51 contatore

```
cu : control_unit
port map(
    stop => fine,
    enable => abilita conteggio,
    start => inizio,
    clk => clock,
    w => scrivi,
    r => leggi,
    count_rst => reset conteggio,
    count_fine => max_count
);

```

Figura 52 control unit

```
rom : rom_seq
port map(
    clk => clock,
    address => indirizzo,
    read => leggi,
    data => data_read
);

```

Figura 53 rom sequenziale

```

mem : memoria
port map(
    clk => clock,
    address => indirizzo,
    write => scrivi,
    data => uscita,
    stringa => data_write
);

```

Figura 54 memoria

L'ultimo componente istanziato è quello relativo alla macchina:

```

mac : macchina
port map(
    i => data_read,
    o => data_write
);
end architecture structural;

```

Figura 55 macchina

Simulazione

Per poter effettuare la simulazione abbiamo scritto il testbench.

```

entity sistema_tb is
end sistema_tb;

architecture rtl of sistema_tb is

component sistema is
    generic(
        word_length : positive := 4
    );
    port(
        inizio : in std_logic;
        fine : in std_logic;
        uscita : out std_logic_vector(word_length-1 downto 0);
        clock : in std_logic
        --indirizzo_corrente : out std_logic_vector(word_length-1 downto 0)
    );
end component;

```

Figura 56 componente sistema

Dichiariamo i segnali che utilizzeremo per controllare e monitorare il comportamento del sistema durante la simulazione.

```

signal start : std_logic := 'U';
signal stop : std_logic := 'U';
signal output : std_logic_vector(3 downto 0) := (others => 'U');
signal clk : std_logic := 'U';
constant PERIOD : time := 5 ns;

```

Figura 57 dichiarazione segnali

La clausola port map(...); viene utilizzata per collegare i segnali della componente cronometro con i segnali dichiarati all'interno del testbench. I collegamenti sono i seguenti:

```

begin
dut: sistema
port map(
    inizio => start,
    fine => stop,
    uscita => output,
    clock => clk
    --indirizzo_corrente => count
);

```

Figura 58 collegamento segnali

Successivamente definiamo un processo **stim_proc** che genera gli stimoli per il sistema durante la simulazione.

```

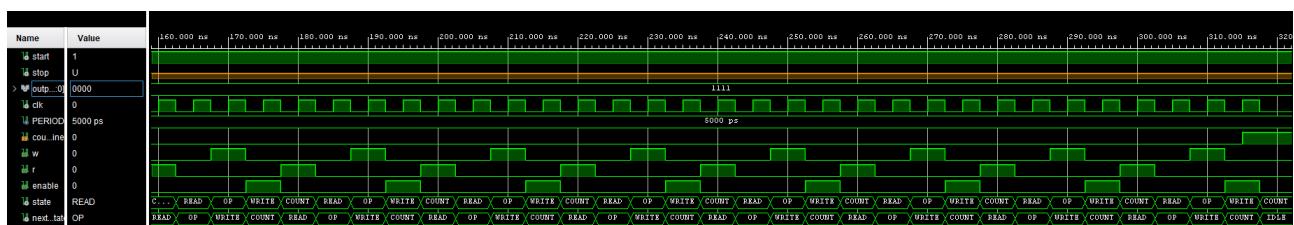
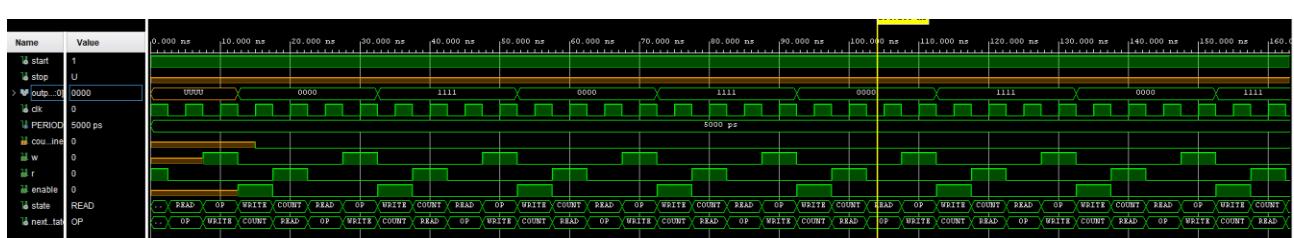
stim_proc : process begin
    start <= '1';
    for i in 0 to 63 loop
        clk <= '1';
        wait for PERIOD/2;
        clk <= '0';
        wait for PERIOD/2;
    end loop;
    wait for 5*PERIOD;
    wait;
end process;
end;

```

Figura 59 stim process

Questo processo genera un segnale **start** attivo alto all'inizio, quindi genera cicli del clock simulati (con un periodo specificato da **PERIOD**) per un totale di 64 cicli. Successivamente, attende per un periodo più lungo prima di rimanere in attesa indefinita.

Avviando la simulazione otteniamo il seguente risultato:



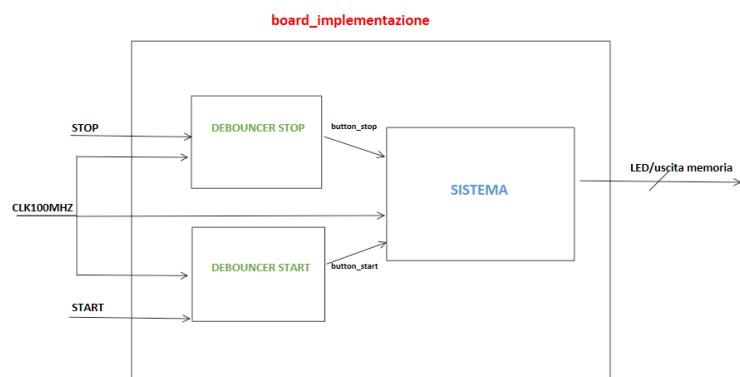
Come possiamo vedere dallo schema, essendo il valore di start pari a 1, si ha l'inizio della simulazione. Il valore di stop è undefined quindi il ciclo della macchina a stati finiti non termina fino al conteggio massimo del contatore. Andando ad analizzare anche i valori di *state* e *nextstate*, possiamo osservare come questi siano coerenti tra loro. Inoltre, anche i valori dei vari segnali in corrispondenza di un determinato stato sono coerenti con quanto progettato. Non sono presenti i risultati della scrittura in memoria delle uscite della macchina combinatoria che però risultano corretti, è presente solo l'uscita della memoria. Abbiamo ritenuto di maggiore interesse verificare il comportamento della FSM.

Esercizio 6.2

Sintetizzare ed implementare su board il componente sviluppato al punto precedente, utilizzando due bottoni per i segnali di *start* e *stop* rispettivamente e i led per la visualizzazione delle uscite della macchina istante per istante.

Progetto e architettura

Abbiamo bisogno di due debouncer per i segnali di *start* e *stop* provenienti dai bottoni, le loro uscite poi saranno l'ingresso per il sistema. In effetti il progetto risulta molto semplice: ad ogni segnale di *start* viene letto un valore dalla rom, elaborato dalla macchina e scritto nella memoria. Il segnale di *stop* resetta il contatore e riporta il sistema in *IDLE*.



Implementazione

Per questo esercizio, siccome il sistema lavora sul fronte di discesa del clock, abbiamo modificato il process del debouncer in modo da allinearla col resto del sistema. Per il resto è stato modificato leggermente il process della control unit, in quanto l'implementazione precedente non era adatta all'implementazione su FPGA. Siamo passati da un codice con doppio process a un codice con singolo process. In questo caso viene utilizzato un solo segnale per lo stato e la lista di sensitività contiene solo il clock. Segue il codice:

```

architecture behavioral of control_unit is
begin
  fsm: process(clk) begin
    if clk'event and clk = '0' then
      case state is
        when IDLE =>
          count_rst <= '0';
          if start = '1' then
            state <= READ;
            r <= '1';
          end if;
          if stop = '1' then
            state <= IDLE;
            count_rst <= '1';
          end if;
        when READ =>
          state <= OP;
          r <= '0';
        when OP =>
          state <= WRITE;
          w <= '1';
        when WRITE =>
          state <= COUNT;
          w <= '0';
          enable <= '1';
        when COUNT =>
          enable <= '0';
          if count_time = "1" then
            state <= IDLE;
            count_rst <= '1';
          else
            state <= IDLE;
          end if;
      end case;
    end if;
  end process;
end behavioral;
  
```

Per quanto riguarda la composizione strutturale di tutti i componenti, questa risulta molto semplice: vengono istanziati due debouncer per i bottoni di *start* e *stop* e le loro uscite vengono collegate ai rispettivi segnali di ingresso del sistema.

```

signal btn_start_pulito, btn_stop_pulito : std_logic := '0';
begin
  bouncer_start : button_debouncer
    generic map (
      CLK_period => 10,
      btn_noise_time => 10000000
    )
    Port map (
      RST => '0',
      CLK => CLK100MHZ,
      BTN => START,
      CLEARED_BTN => btn_start_pulito
    );
  bouncer_stop : button_debouncer
    generic map (
      CLK_period => 10,
      btn_noise_time => 10000000
    )
    Port map (
      RST => '0',
      CLK => CLK100MHZ,
      BTN => STOP,
      CLEARED_BTN => btn_stop_pulito
    );
  sis : sistema
    generic map (
      word_length => 4
    )
    port map(
      inizio => btn_start_pulito,
      fine => btn_stop_pulito,
      uscita => LED,
      clock => CLK100MHZ
    );
end board_implementazione;

```

Abbiamo utilizzato il clock a 100MHz della board, i bottoni *BTNL* e *BTRN* per i segnali di *start* e *stop* rispettivamente e abbiamo utilizzato i primi 4 LED per visualizzare l'ultimo valore scritto dalla memoria ricevuto dalla macchina combinatoria. Per completezza riportiamo il file relativo ai vincoli:

```

## Clock signal
set_property -dict { PACKAGE_PIN E3      IOSTANDARD LVCMOS33 } [get_ports { CLK100MHZ }]; #IO_L12P_T1_MRCC_35 Sch=clk100mhz
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports {CLK100MHZ}];

## LEDs
set_property -dict { PACKAGE_PIN H17      IOSTANDARD LVCMOS33 } [get_ports { LED[0] }]; #IO_L18P_T2_A24_15 Sch=led[0]
set_property -dict { PACKAGE_PIN K15      IOSTANDARD LVCMOS33 } [get_ports { LED[1] }]; #IO_L24P_T3_RS1_15 Sch=led[1]
set_property -dict { PACKAGE_PIN J13      IOSTANDARD LVCMOS33 } [get_ports { LED[2] }]; #IO_L17N_T2_A25_15 Sch=led[2]
set_property -dict { PACKAGE_PIN N14      IOSTANDARD LVCMOS33 } [get_ports { LED[3] }]; #IO_L8P_T1_D11_14 Sch=led[3]

set_property -dict { PACKAGE_PIN P17      IOSTANDARD LVCMOS33 } [get_ports { START }]; #IO_L12P_T1_MRCC_14 Sch=btnl
set_property -dict { PACKAGE_PIN M17      IOSTANDARD LVCMOS33 } [get_ports { STOP }]; #IO_L10N_T1_D15_14 Sch=bttr

```

Esercizio 7: Moltiplicatore di Booth

Esercizio 7.1:

Progettare, implementare in VHDL e simulare una macchina moltiplicatore di Booth in grado di effettuare il prodotto di 2 stringhe A e B da 8 bit ciascuna

Progetto e architettura

Il moltiplicatore di Booth è un moltiplicatore sequenziale. I moltiplicatori sequenziali sono particolari tipi di macchinari che presentano sia una rete combinatoria utilizzata per implementare un algoritmo specifico, sia una rete sequenziale che serve a gestirne il corretto funzionamento. Quando parliamo di moltiplicatore di Booth intendiamo un moltiplicatore sequenziale che utilizza l'algoritmo di Booth per effettuare la moltiplicazione. L'obiettivo dell'algoritmo è moltiplicare due numeri interi con segno, utilizzando l'algoritmo manuale della moltiplicazione. Iniziamo formalizzando il problema: abbiamo due operandi in ingresso, un moltiplicatore X registrato su n bit e un moltiplicando Y registrato su m bit. Il prodotto tra i due operandi, P, sarà memorizzato su un totale di n + m bit.

Ora, considerando di effettuare una moltiplicazione tra due numeri binari, entrambi di 4 cifre, il risultato che ci aspettiamo, dovrà essere su 8 cifre. Ad ogni iterazione dell'algoritmo, moltiplicheremo la cifra i-esima del moltiplicatore per il moltiplicando, ed effettueremo uno shift di **i posizioni** verso sinistra. Alla fine, sommeremo tutti i prodotti parziali ottenuti.

In questo modo il prodotto finale potrà essere definito nel seguente modo:

$$P = \sum_{j=0}^{n-1} Y$$

Questo approccio però risulta inefficiente in quanto si dovrebbero memorizzare tutti i prodotti parziali ed eseguire un numero di shift elevato.

Per questo motivo si segue un altro approccio che consiste nel calcolare il prodotto parziale i -esimo a partire dal prodotto parziale precedente. Ovviamente va considerato pari a 0 il prodotto parziale iniziale.

La formula diventerebbe la seguente:

$$P_{i+1} = P_i + x_i 2^i Y$$

Dove $P_0 = 0$.

In questo modo risolviamo il problema di dover memorizzare ogni prodotto, in quanto lavoriamo su di un unico registro per la memorizzazione del prodotto parziale. Il problema dell'elevato numero di shift però rimane.

Per questo motivo si utilizza un algoritmo alternativo che procede nel seguente modo:

1. Si effettua la moltiplicazione del bit X_i del moltiplicatore con il moltiplicando Y
2. Si somma il prodotto parziale appena ottenuto con P_i
3. Si effettua uno shift a destra del prodotto parziale P_i appena calcolato

$$P_i = P_i + x_i Y$$

$$P_{i+1} = 2^{-1} P_i$$

Utilizzando questo algoritmo il numero di prodotti parziali memorizzati e il numero di shift effettuati sono stati ridotti. Alla fine dell'algoritmo il registro AQ conterrà il risultato dell'operazione.

All'inizio dell'operazione il registro Q sarà vuoto per poi essere riempito di un bit ad ogni iterazione.

Ulteriori problemi relativi all'utilizzo di questo algoritmo nascono quando si vuole effettuare un prodotto tra due numeri interi relativi.

Se i numeri sono codificati in complementi a due una soluzione concettualmente semplice consiste nel negare tutti gli operandi negativi, effettuare un'operazione unsigned sui numeri positivi risultanti e poi negare il risultato se necessario. Questo porta ad utilizzare un numero di risorse maggiore in quanto sarà necessario modificare X e Y.

Una soluzione alternativa si basa sullo sfruttare alcune proprietà della rappresentazione in complementi a due.

Un numero in complementi può essere espresso nel seguente modo:

$$X = -2^{n-1} x^{n-1} + \sum_{i=0}^{n-2} 2^i x_i$$

Se il bit più significativo è nullo il numero è positivo altrimenti è negativo. Un numero negativo attribuisce un peso -1 al bit più significativo.

In base ai segni dei fattori X e Y possiamo formalizzare 4 casi:

- **X>0, Y>0:** si segue l'algoritmo illustrato in precedenza;

- **X<0, Y>0:** si segue l'algoritmo precedente effettuando una modifica all'ultimo passo. Nell'ultima iterazione prenderemo in considerazione il bit più significativo che sarà 1. Questo bit provocherà una sottrazione al prodotto parziale invece di un'addizione;
- **X>0, Y<0:** l'algoritmo procede normalmente tenendo in considerazione che, visto che Y è negativo, il prodotto parziale sarà per forza negativo. Per questo motivo lo shift in testa ai registri a scorrimento dovrà essere effettuato con un uno invece che con zero.
- **X<0, Y<0:** si segue il secondo caso.

Visto che vogliamo realizzare un moltiplicatore di Booth andiamo ad aggiungere a questo algoritmo la *codifica di Booth*.

Ricordiamo che un numero in complemento a due su n cifre si può rappresentare come composizione di un certo numero di cifre X_i . Definiamo:

$$y_0 = -x_0$$

$$y_1 = -x_1 + x_0$$

...

$$y_{n-1} = -x_{n-1} + x_{n-2}$$

Moltiplicando il generico y_i per la potenza 2^i e sommando ogni termine otteniamo:

$$y_{n-1}2^{n-1} + y_{n-2}2^{n-2} + \dots + y_02^0 = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_02^0$$

Rappresentazione posizionale Rappresentazione in complementi a due

Da qui notiamo che la rappresentazione secondo y è equivalente a un numero X in complemento a 2 su n cifre. Per questo motivo possiamo rappresentare x secondo la codifica alternativa di Booth. La rappresentazione Booth-1 di x può essere facilmente ottenuta sostituendo ciascuna coppia di bit adiacenti di x con un valore in {-1, 0, 1}, secondo la corrispondenza in tabella:

$x_j x_{j-1}$	codifica
00/11	0
01	+1
10	-1

Tramite questa codifica è possibile ridurre il numero di moltiplicazioni (e di somme) da effettuare per il prodotto fra numeri relativi. La situazione sarà la seguente:

$x_j x_{j-1}$	
00/11	Non viene effettuata né la somma né la sottrazione, ma solo lo shift
01	Y viene aggiunto al prodotto parziale corrente
10	Y viene sottratto dal prodotto parziale corrente

Per il progetto del moltiplicatore abbiamo utilizzato un approccio modulare utilizzando diversi componenti. Lo schema del moltiplicatore è il seguente:

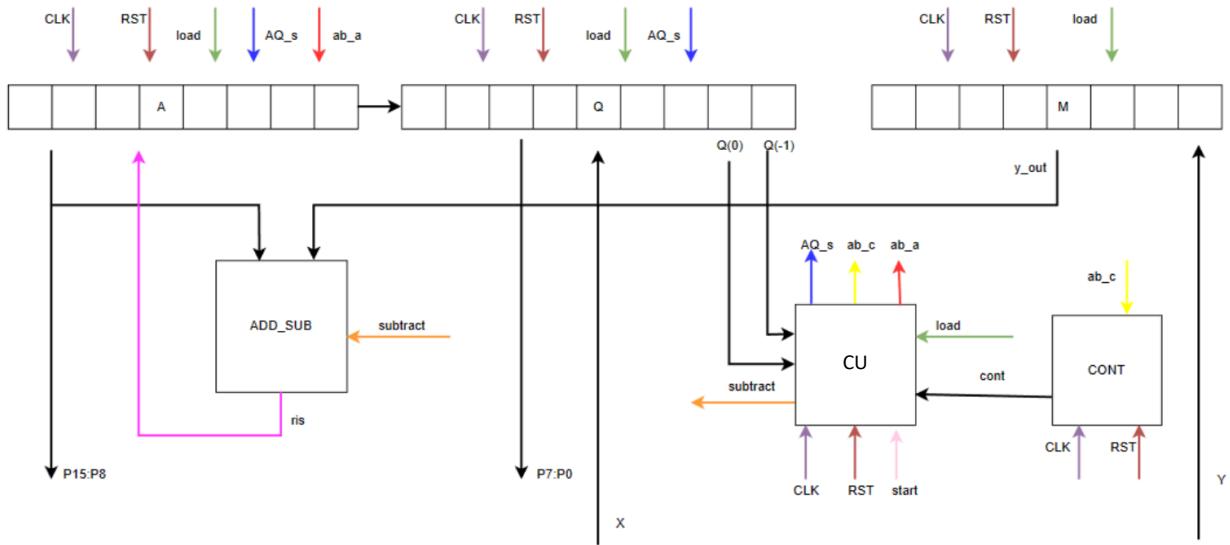


Figura 60 schema moltiplicatore di booth

A e **Q** sono i registri a scorrimento (entrambi da 8 bit) che conterranno alla fine dell'operazione il risultato. Come si può notare anche dallo schema il registro **Q** conterrà un bit extra posto inizialmente a 0 necessario per gestire correttamente la codifica di Booth. Il registro **M** a 8 bit contiene il moltiplicando.

Il **sommatore** riceverà in ingresso il valore di uscita di **A** e il valore di **M**. L'uscita del sommatore verrà posta in **A**, in base al valore di bit **Q[0]** e **Q[1]** verrà stabilito se si dovrà effettuare a sottrazione o l'addizione.

L'architettura del parallel adder utilizza un sommatore a riporto interno (**ripple carry adder**) per eseguire l'operazione di somma o sottrazione, a seconda del valore di **carry in** implementando così un sommatore-sottrattore a 8 bit. Il valore di **carry in** viene fornito dalla **control unit**.

La **control unit** gestisce le operazioni di shift (aritmetico a destra) e di somma/sottrazione, controlla il flusso dell'algoritmo basandosi sui valori dei bit del moltiplicatore e sugli stati interni e coordina l'operazione di shift, la somma e la sottrazione.

Il **counter** viene utilizzato per tenere traccia della posizione corrente (bit) del moltiplicatore durante l'algoritmo. Può essere incrementato di uno ad ogni iterazione di shift.

La **parte operativa** dell'architettura è composta dai registri **A**, **Q** ed **M**, dal sommatore e dal counter.

Implementazione

Come già spiegato prima, abbiamo utilizzato un approccio modulare per la realizzazione del moltiplicatore di Booth. In particolare, non analizzeremo come abbiamo implementato lo shift register in quanto la sua implementazione già è stata spiegata nell'esercizio 4. Comportamento analogo abbiamo seguito per il contatore in quanto già spiegato nell'appendice. Per risolvere questo esercizio, precisiamo che siamo partiti dall'implementazione del moltiplicatore di Robertson che ci è stata fornita e abbiamo apportato le opportune modifiche. Iniziamo con un'analisi dell'implementazione dell'unità operativa.

L'unità operativa è composta dai seguenti componenti: shift register (composto dai registri **A** e **Q**), registro8 (rappresenta il registro **M** in cui è presente il moltiplicando **Y**), counter, adder_sub e ripple_carry che insieme compongono il parallel adder.

Iniziamo con il **registro8**, un registro parallelo-parallelo che mantiene il valore del moltiplicando **Y**.

Ne riportiamo solo l'interfaccia in quanto è un semplice registro a 8 bit con funzionalità di caricamento sincrono e reset sincrono, che quindi riteniamo poco interessante.

```
--registro parallelo-parallelo che mantiene il valore del moltiplicando Y
entity registro8 is
    port( A: in std_logic_vector(7 downto 0);
          clk, res, load: in std_logic;
          B: out std_logic_vector(7 downto 0));
end registro8;
```

Figura 61 entity registro 8

Successivamente andiamo ad analizzare il **parallel adder**.

Prima di fare ciò ci sembra necessario fornire una breve spiegazione riguardo a cos'è e come funziona il parallel adder implementato con ripple carry.

Il **parallel adder** implementato con ripple carry è un circuito logico progettato per sommare contemporaneamente i bit corrispondenti di due numeri binari. La caratteristica "parallel" deriva dal fatto che tutte le addizioni vengono eseguite contemporaneamente in parallelo, accelerando il processo rispetto a un'implementazione sequenziale.

Il circuito include due operandi, X e Y, ciascuno a 8 bit. Inizialmente, viene calcolato il complemento a 1 di Y rispetto a un bit di carry in, ottenendo così **complementoy**. Questo passo è importante per la gestione delle sottrazioni. L'operazione di somma è eseguita bit per bit utilizzando un ripple carry adder, che è una struttura in cui il carry-out di un bit viene sommato al bit successivo.

Il circuito di ripple carry è composto da sette full adder collegati in cascata.

Ogni full adder è responsabile della somma di un bit specifico degli operandi, del bit di carry-in e produce il bit di somma e il carry-out. Il risultato finale della somma è ottenuto sommando i bit di somma prodotti da ciascun full adder.

L'utilizzo del ripple carry consente di propagare efficientemente il carry attraverso i vari bit.

Per l'implementazione di questi componenti abbiamo utilizzato un approccio strutturale.

Effettuiamo un'analisi bottom up per spiegare al meglio l'implementazione del parallel adder. Per questo partiamo dal full adder che viene utilizzato per realizzare il ripple carry, a sua volta utilizzato per implementare il parallel adder.

Iniziamo con la definizione dell'entity:

```
entity full_adder is
    port(
        a,b: in std_logic;
        cin: in std_logic;
        cout, s: out std_logic);
end full_adder;
```

Figura 62 entity full adder

Il full adder è un circuito logico utilizzato per sommare tre ingressi: a, b e un bit di riporto (carry in) cin.

Abbiamo dichiarato i seguenti port:

- **a** e **b**: input di tipo *std_logic*, sono gli ingressi corrispondenti ai bit di A e B da sommare.
- **cin**: input di tipo *std_logic*, rappresenta il bit di carry-in.

- **cout**: uscita di tipo *std_logic*, è l'uscita che rappresenta il bit di carry-out, ovvero il carry che viene propagato al bit successivo.
- **s**: uscita di tipo *std_logic*, è l'uscita che rappresenta il risultato della somma di A, B e del carry-in.

L'architettura del full adder è la seguente:

```
architecture rtl of full_adder is
begin
  s<= a xor b xor cin;
  cout<= (a and b) or (cin and (a xor b));
end rtl;
```

Figura 63 architettura full adder

Implementiamo la logica interna del full adder la quale si basa sul XOR e porte logiche AND e OR. Descriviamo come i segnali di output sono calcolati in base agli ingressi.

- **s <= a xor b xor cin**: calcola il risultato della somma (**s**). Utilizza l'operatore XOR (**xor**) per sommare i bit di A, B e il carry-in (**cin**). XOR restituirà '1' solo quando un numero dispari di ingressi è '1', rappresentando così la somma binaria.
- **cout <= (a and b) or (cin and (a xor b))**: calcola il bit di carry-out (**cout**). Utilizza le porte logiche AND (**and**) e OR (**or**). Il termine (**a and b**) rappresenta la generazione del carry, che è '1' solo quando sia A che B sono '1'. Il termine (**cin and (a xor b)**) rappresenta la propagazione del carry, che è '1' quando c'è un carry-in e la somma di A e B (senza il carry-in) è '1'. L'OR logico combina questi due contributi per determinare il bit di carry-out.

Dopo aver visto l'implementazione andiamo ad analizzare l'implementazione del ripple carry. Partiamo dalla definizione dell'entity:

```
entity ripple_carry is
  port( X, Y: in std_logic_vector(7 downto 0);
        c_in: in std_logic;
        c_out: out std_logic;
        Z: out std_logic_vector(7 downto 0));
end ripple_carry;
```

Figura 64 entity ripple carry

Questo componente accetta due operandi di 8 bit (**X** e **Y**), un bit di carry in (**c_in**), e produce una somma di 8 bit (**Z**) insieme al bit di carry out (**c_out**).

I port che abbiamo dichiarato sono i seguenti:

- **X, Y**: operandi in ingresso. Sono di tipo *std_logic_vector* di 8 bit ciascuno.
- **c_in**: bit di carry in, che rappresenta il carry proveniente da operazioni precedenti, segnale di input di tipo *std_logic*
- **c_out**: bit di carry out, che rappresenta il carry generato da questa operazione. È un segnale di output di tipo *std_logic*
- **Z**: Risultato della somma, un vettore di segnali logici di 8 bit. È un segnale di output di tipo *std_logic_vector*

L'idea chiave del ripple carry è che la somma di ciascun bit dipende dai bit corrispondenti negli operandi, dal carry in e dal carry generato dai bit meno significativi. Il risultato e il carry out sono quindi passati alla

successiva coppia di bit più significativi, generando una catena di somme e carry attraverso i bit dell'operazione.

Per l'architettura abbiamo adoperato un approccio structural:

```
architecture structural of ripple_carry is
    component full_adder is
        port(
            a,b: in std_logic;
            cin: in std_logic;
            cout, s: out std_logic);
    end component;

    signal temp: std_logic_vector(7 downto 0);

begin
    RA0: full_adder port map(X(0), Y(0), c_in, temp(0), Z(0));

    RA1to6: FOR i IN 1 TO 6 GENERATE
        RA: full_adder port map(X(i), Y(i), temp(i-1), temp(i), Z(i));
    END GENERATE;

    RA7: full_adder port map(X(7), Y(7), temp(6), c_out, Z(7));
end structural;
```

Figura 65 architettura ripple carry

Per implementare il ripple carry istanziamo il componente del full adder del quale abbiamo visto l'implementazione in precedenza.

In questa architettura mostriamo come i singoli componenti full adder siano collegati in serie per creare un sommatore ripple carry.

Dopo aver istanziato il componente full adder andiamo a dichiarare un segnale **temp** di tipo *std_logic_vector* di 8 bit. Utilizziamo questo segnale per conservare temporaneamente i risultati intermedi durante l'esecuzione.

Adesso vediamo l'implementazione della somma con ripple carry attraverso la connessione sequenziale di full adder in cascata.

- **RA0: full_adder port map(X(0), Y(0), c_in, temp(0), Z(0)):** questa istruzione collega il primo bit degli operandi (**X(0)** e **Y(0)**) al primo **full_adder**. L'ingresso di carry (**c_in**) viene collegato al carry iniziale (**c_in**), l'uscita di somma (**temp(0)**) viene collegata al primo elemento del vettore temporaneo **temp**, e l'uscita di carry (**Z(0)**) viene collegata al primo bit del risultato finale.
- **RA: full_adder port map(X(i), Y(i), temp(i-1), temp(i), Z(i)):** questa istruzione collega il bit corrente dell'operando (**X(i)** e **Y(i)**) al **full_adder**, l'ingresso di carry (**temp(i-1)**) è il risultato del **full_adder** precedente, l'uscita di somma (**temp(i)**) viene collegata al vettore temporaneo **temp**, e l'uscita di carry (**Z(i)**) viene collegata al bit corrispondente del risultato finale.
- **RA7: full_adder port map(X(7), Y(7), temp(6), c_out, Z(7)):** questa istruzione collega l'ottavo bit degli operandi (**X(7)** e **Y(7)**) all'ultimo **full_adder**. L'ingresso di carry (**temp(6)**) è il risultato del **full_adder** precedente, l'uscita di carry (**c_out**) viene collegata al carry finale, e l'uscita di somma (**Z(7)**) viene collegata all'ottavo bit del risultato finale.

Dopo aver visto l'implementazione del ripple carry andiamo ad analizzare l'**adder_sub**.

L'entity dichiarata è la seguente, i segnali hanno lo stesso significato del componente precedente:

```

entity adder_sub is
  port( X, Y: in std_logic_vector(7 downto 0);
        cin: in std_logic;
        Z: out std_logic_vector(7 downto 0);
        cout: out std_logic);
end adder_sub;

```

Figura 66 entity adder sub

L'entità **adder_sub** rappresenta un componente che esegue operazioni di somma o sottrazione su operandi a 8 bit, con la possibilità di specificare un carry-in e restituendo il risultato a 8 bit insieme al carry-out.

```

architecture structural of adder_sub is
  component ripple_carry is
    port( X, Y: in std_logic_vector(7 downto 0);
          c_in: in std_logic;
          c_out: out std_logic;
          Z: out std_logic_vector(7 downto 0));
  end component;

  signal complementoy: std_logic_vector(7 downto 0);

begin

  complementoy_y: FOR i IN 0 TO 7 GENERATE
    complementoy(i)<=Y(i) xor cin;
  END GENERATE;

  RA: ripple_carry port map(X, complementoy, cin, cout, Z);
end structural;

```

Figura 67 architettura adder sub

Come prima cosa istanziamo il componente ripple carry implementato in precedenza.

Successivamente dichiariamo un segnale **complementoy** di tipo *std_logic_vector*. Questo segnale è di 8 bit ed è un segnale temporaneo utilizzato per memorizzare il complemento a uno di **Y**. Otteniamo il complemento a uno effettuando una XOR tra ciascun bit di **Y** il segnale di carry in **cin**.

In seguito, eseguiamo il mapping delle porte del componente ripple carry.

Utilizziamo come input gli operandi **X** e **complementoy**, il carry in **cin** viene passato direttamente mentre i risultati **Z** e **cout** vengono ottenuti direttamente come output dal componente ripple carry.

Dopo aver analizzato i vari componenti che la costituiscono passiamo all'implementazione dell'**unità operativa**. Come sempre, partiamo dalla definizione dell'entity:

```

entity unita_operativa is
  port( X, Y: in std_logic_vector(7 downto 0);--moltiplicatore e moltiplicando
        clock, reset: in std_logic;
        loadAQ, shift, loadM, sub, selAQ, count_in: in std_logic;
        count: out std_logic_vector(2 downto 0);
        P: out std_logic_vector(16 downto 0));
end unita_operativa;

```

Questa entity rappresenta l'unità operativa per la moltiplicazione, gestendo gli operandi, il clock, i segnali di controllo e producendo l'output della moltiplicazione.

I port dichiarati nell'entity sono i seguenti:

- **X, Y: (in std_logic_vector(7 downto 0))** I due operandi della moltiplicazione, **X** (moltiplicatore) e **Y** (moltiplicando), entrambi a 8 bit.

- **clock, reset:** (*in std_logic*) Il segnale di clock per sincronizzare le operazioni dell'unità operativa e il segnale di reset per ripristinare l'unità ai valori iniziali.
- **loadAQ, shift, loadM, sub, selAQ, count_in:** (*in std_logic*) Segnali di controllo che determinano il comportamento dell'unità operativa:
 - **loadAQ:** Carica il registro AQ con il prodotto parziale o con l'operando **Y**.
 - **shift:** Esegue uno shift logico a sinistra sul registro AQ.
 - **loadM:** Carica il registro M (Multiplier) con l'operando **Y**.
 - **sub:** Segnale per abilitare la sottrazione.
 - **selAQ:** Segnale di selezione per il mux, per selezionare l'ingresso parallelo dello shift register: valore iniziale AQ_init oppure uscita dell'adder AQ_sum_in
 - **count_in:** Segnale di abilitazione del conteggio in ingresso per il contatore.
- **count:** (*out std_logic_vector(2 downto 0)*) Il segnale di conteggio in uscita, rappresentante il valore del contatore.
- **P:** (*out std_logic_vector(16 downto 0)*) L'output della moltiplicazione, rappresentato da un vettore a 16 bit (più uno extra per la questione della codifica), contenente il prodotto finale.

Anche in questo caso per la definizione dall'architettura abbiamo seguito un approccio strutturale istanziando i vari componenti implementati in precedenza.

Ricordiamo che i componenti dell'unità operativa sono i seguenti:

- **adder_sub:** il sommatore a 8 bit.
- **registro8:** il registro a 8 bit utilizzato per memorizzare il moltiplicando.
- **mux_21:** il multiplexer 2:1 per il caricamento dello shift register.
- **shift_register:** shift register a 17 bit (16 del risultato più il bit extra).
- **cont_mod8:** un contatore modulo-8 utilizzato per contare le iterazioni durante la moltiplicazione.

```

architecture structural of unita_operativa is

    component adder_sub is
        port( X, Y: in std_logic_vector(7 downto 0);
              cin: in std_logic;
              Z: out std_logic_vector(7 downto 0);
              cout: out std_logic);
    end component;

    component registro8 is
        port( A: in std_logic_vector(7 downto 0);
              clk, res, load: in std_logic;
              B: out std_logic_vector(7 downto 0));
    end component;

    component mux_21 is
        generic (width : integer range 0 to 17 := 8);
        port( x0, x1: in std_logic_vector(width-1 downto 0);
              s: in std_logic;
              y: out std_logic_vector(width-1 downto 0));
    end component;

    component shift_register is
        port( parallel_in: in std_logic_vector(16 downto 0);
              serial_in: in std_logic;
              clock, reset, load, shift: in std_logic;
              parallel_out: out std_logic_vector(16 downto 0));
    end component;

    component cont_mod8 is
        port( clock, reset: in std_logic;
              count_in: in std_logic;
              count: out std_logic_vector(2 downto 0));
    end component;

```

Figura 68 istanziazione dei componenti

Successivamente abbiamo dichiarato i seguenti segnali:

```

signal Mreg: std_logic_vector(7 downto 0); --segnale temporaneo tra reg8 M e mux 21
signal AQ_init: std_logic_vector(16 downto 0); --segnale in input all'SR
signal AQ_in: std_logic_vector(16 downto 0); --segnale in input all'SR
signal AQ_out: std_logic_vector(16 downto 0); --segnale temporaneo uscita dell'SR
signal sum: std_logic_vector(7 downto 0); --uscita del parallel adder
signal AQ_sum_in : std_logic_vector(16 downto 0);
signal riporto: std_logic; -- riporto in uscita dell'adder che non utilizziamo
signal SRserialIn: std_logic;

```

Figura 69 segnali unità operativa

- **Mreg** (*std_logic_vector(7 downto 0)*): questo segnale rappresenta il moltiplicando memorizzato nel registro **registro8**. Viene utilizzato come secondo operando per il parallel adder **adder_sub**
- **AQ_init** (*std_logic_vector(16 downto 0)*): segnale in input allo shift register durante la fase di inizializzazione. Contiene il valore iniziale da caricare nello shift register **SR** ed è formato concatenando zero, il moltiplicatore **X**, e un bit zero per il bit Q_{-1} .
- **AQ_in** (*std_logic_vector(16 downto 0)*): segnale in input allo shift register durante la fase operativa dopo aver effettuato la somma. Viene elaborato in uscita dal multiplexer **mux_21**.
- **AQ_out** (*std_logic_vector(16 downto 0)*): segnale temporaneo rappresentante l'uscita dello shift register **SR**. Contiene il risultato delle operazioni eseguite nello shift register. Gli 8 bit significativi rappresentano il primo operando del sommatore
- **sum** (*std_logic_vector(7 downto 0)*): Uscita del parallel adder. Rappresenta il risultato dell'addizione tra l'uscita dello shift register e il moltiplicando.
- **AQ_sum_in** (*std_logic_vector(16 downto 0)*): stringa di 16 bit da inserire nello shift register A.Q durante la fase operativa dopo aver effettuato la somma
- **riporto** (*std_logic*): riporto in uscita dall'adder che, in questa implementazione, non viene utilizzato dall'unità operativa.
- **SRserialIn** (*std_logic*): ingresso seriale dello shift register. Rappresenta il bit più significativo da mantenere durante lo shift in fase di finalizzazione.

Osserviamo ora l'istanziazione dei componenti:

```
begin
    -- 1) predisposizione del secondo operando della somma:
    -- registro moltiplicando
    M: registro8 port map(Y, clock, reset, loadM, Mreg);
```

Figura 70 stanziamento registro8

Successivamente predisponiamo i segnali di inizializzazione dello shift register a seconda del segnale di selezione **selAQ**:

```
-- 2) predisposizione del primo operando della somma:
--stringa da 16 bit da inserire nello shift register A.Q durante la fase di inizializzazione:
--ottenuta concatenando 00000000 con il moltiplicatore X
AQ_init <= "00000000" & X & "0"; --valore da inserire all'inizio nello shift register

--stringa di 16 bit da inserire nello shift register A.Q durante la fase operativa dopo aver effettuato la somma
AQ_sum_in <= sum & AQ_out(8 downto 0);

-- mux per selezionare l'ingresso parallelo dello shift register: valore iniziale AQ_init
-- oppure uscita dell'adder AQ_sum_in
MUX_SR_parallel_in : mux_21 generic map (width => 17) port map(AQ_init, AQ_sum_in, selAQ, AQ_in);
```

Figura 71 predisposizione primo operando della somma

Dopo predisponiamo l'ingresso seriale dello shift register:

```
-- 3) predisposizione dell'ingresso seriale dello shift register: deve prendere sempre F tranne che nel  
-- final shift, quando bisogna mantenere il bit piÙ significativo
```

```
SRserialIn <= AQ_out(16);
```

Figura 72 predisposizione ingresso seriale dello shift register

```
-- 4) shift register A.Q
```

```
SR: shift_register port map(AQ_in, SRserialIn, clock, reset, loadAQ, shift, AQ_out);
```

Figura 73 shift register

```
-- 5) sommatore
```

```
ADD_SUB: adder_sub port map(AQ_out(16 downto 9), Mreg, sub, sum, riporto);
```

Figura 74 sommatore

I segnali forniti al componente sono i bit 16-9 dell'uscita dello shift register **AQ_out**, il registro **Mreg** (moltiplicando), **sub** (che determina se l'operazione è un'addizione o una sottrazione), e i segnali **sum** e **riporto**.

Come ultima cosa istanziamo il contatore.

```
-- 6) contatore
```

```
CONT: cont_mod8 port map(clock, reset, count_in, count);
```

Figura 75 contatore

Infine, collegiamo l'uscita P del moltiplicatore all'uscita dello shift register **AQ_out**.

```
P<=AQ_out;
```

```
end structural;
```

Per ottenere l'implementazione completa del moltiplicatore di Booth manca ancora l'implementazione dell'unità di controllo che andremo ad analizzare adesso.

L'unità di controllo gestisce i vari componenti dell'architettura come il caricamento del moltiplicando, la gestione del contare, il controllo dello shifter, la selezione dell'ingresso dell'addizionatore/sottrattore e il segnale di stop per l'arresto dell'unità di controllo.

L'entità che andiamo a definire è la seguente:

```

entity unita_controllo is
  port( qLSB : in std_logic_vector(1 downto 0);
        clock, reset, start: in std_logic;--clock il clock della board, clock_div viene dal divisore di freq
        count: in std_logic_vector(2 downto 0);
        loadM, count_in, loadAQ, en_shift: out std_logic;
        selAQ, subtract, stop_cu: out std_logic);
end unita_controllo;

```

I segnali di input sono i seguenti:

- **qLSB** (*in std_logic_vector*): Un vettore di 2 bit rappresentante i bit meno significativi dello shift register.
- **clock**
- **reset** (*in std_logic*): Il segnale di reset per l'unità di controllo.
- **start** (*in std_logic*): Il segnale di avvio dell'operazione di moltiplicazione.
- **count** (*in std_logic_vector*): Un vettore di 3 bit rappresentante il valore corrente del contatore delle iterazioni dell'algoritmo.

I segnali di output sono fondamentalmente segnali di controllo per l'unità operativa e presentano gli stessi nomi visti prima. L'architettura dell'unità di controllo è basata su una macchina a stati finiti.

Prima di andare ad analizzare il codice andiamo a disegnare l'automa di stati finiti in modo da rendere più semplice la successiva implementazione tramite codice:

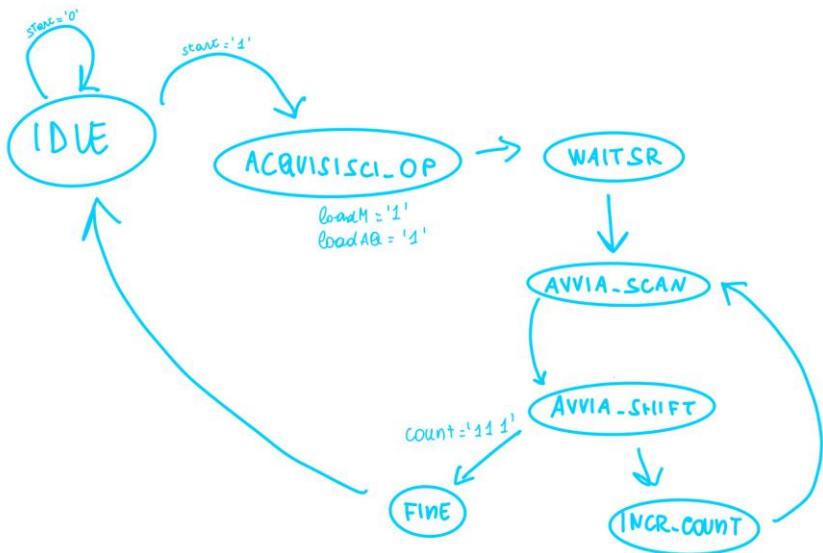


Figura 76 automa a stati finiti unità di controllo

Adesso passiamo all'analisi del codice.

Iniziamo con la definizione degli stati:

```

architecture structural of unita_controllo is
  type state is (idle, acquisisci_op, waitSR, avvia_scan, avvia_shift, incr_count, fine);
  signal current_state,next_state: state;

```

Gli stati che abbiamo definito sono i seguenti:

- **idle**: Stato iniziale quando l'unità di controllo è inattiva.
- **acquisisci_op**: Stato in cui l'unità di controllo acquisisce i dati iniziali, caricando il moltiplicando (**M**) e inizializzando lo shift register(**AQ**).
- **waitSR**: Stato in cui l'unità di controllo attende il completamento dell'inizializzazione dello shift register **AQ**.
- **avvia_scan**: Stato in cui l'unità di controllo analizza **qLSB**.
- **avvia_shift**: Stato in cui l'unità di controllo abilita lo shift nello shift register **AQ**.
- **incr_count**: Stato in cui l'unità di controllo abilita l'incremento del contatore.
- **fine**: Stato di fine dell'operazione di moltiplicazione.

Oltre a definire gli stati dichiariamo anche i segnali che vengono utilizzati per rappresentare lo stato attuale (**current_state**) e lo stato prossimo (**next_state**). Utilizziamo questi segnali per sincronizzare il cambiamento di stato con i fronti di salita del segnale di clock come abbiamo visto anche con il riconoscitore e il sistema.

```
begin

reg_stato: process(clock)
    begin
        if(clock'event and clock='1') then
            if(reset='1') then
                current_state <=idle;
            else
                current_state <=next_state;
            end if;
        end if;
    end process;
```

Figura 77 processo registrazione stato

Adesso implementiamo il processo che gestisce le variazioni tra i vari stati in base alle variazioni dei segnali

Il processo, infatti, è sensibile ai cambiamenti dei seguenti segnali: **current_state**, **start** e **count**.

```
comb: process(current_state, start, count)
begin

-- Attenzione! questo process si attiva ogni volta che c'è una variazione nei segnali della sensitivity list
-- current_state e count per loro natura variano sempre in corrispondenza del fronte di salita del clock
-- start viene dall'esterno: se non varia (sale e scende) col fronte del clock, si potrebbe avere una situazione
-- in cui il next_state varia ma non ha modo da stabilizzarsi (perché current_state non è ancora variato)
-- quando il moltiplicatore sarà messo su board, START dovrà essere generato come uscita del button debouncer

count_in <='0';
subtract <='0';
selAQ <= '0';
loadAQ <='0'; --carica nello shift register
loadM <='0'; --carica il moltiplicando nel registro M
stop_cu <='0';
en_shift <='0'; --segna che abilita lo shift durante le prime N-1 iterazioni
```

Figura 78 definizione process

All'interno del process inizializziamo a zero i vari segnali di controllo. Questo garantisce che tutti i segnali di controllo siano in uno stato noto prima di essere assegnati all'interno del processo.

Tramite un costrutto case andiamo a definire sotto quali condizioni logiche devono effettuarsi i cambi di stato evidenziati dall'automa a stati finiti.

```

CASE current_state is
WHEN idle =>

    if(start='1') then
        | next_state <= acquisisci_op;
    else
        | next_state <= idle;
    end if;

--fornisce i segnali di caricamento operandi
WHEN acquisisci_op =>

    loadM <='1'; --abilita il caricamento del moltiplicando nel registro M
    loadAQ <='1'; --abilita il caricamento del moltiplicatore e degli 8 zeri in testa
                    --nello shift register A.Q (perch selAQ=0)
    next_state <= waitSR;

--acquisisce gli operandi, su cui il sommatore inizia a lavorare immediatamente
WHEN waitSR =>

    | next_state <= avvia_scan;

WHEN avvia_scan =>

    if(qLSB = "01") then
        selAQ <= '1';
        loadAQ <= '1'; --fornisce il segnale di caricamento in A del risultato della somma
        next_state <= avvia_shift;
    elsif(qLSB = "10") then
        subtract <= '1';
        selAQ <= '1';
        loadAQ <= '1'; --fornisce il segnale di caricamento in A del risultato della somma
        next_state <= avvia_shift;
    elsif (qLSB = "00" or qLSB = "11") then
        next_state <= avvia_shift;
    end if;

--carica il risultato della somma in A e da fornisce il segnale di shift
WHEN avvia_shift =>

    en_shift <='1';
    if(count="111") then
        next_state <= fine;
    else
        next_state <= incr_count;
    end if;
--esegue lo shift, abilita incremento conteggio e predisponde per nuova iterazione
WHEN incr_count =>

    count_in <= '1';
    next_state <= avvia_scan;
WHEN fine =>

    stop_cu <='1';
    next_state <= idle;
end CASE;
end process;
end structural;

```

Analizziamo nel dettaglio le transizioni, questa volta ci concentreremo anche sulle modifiche dei vari segnali in base ai passaggi di stato; nel momento del disegno dell'automa a stati finiti avevamo tralasciato questo aspetto:

- **idle**: Se lo stato corrente è **idle** e il segnale **start** diventa '1', il prossimo stato sarà **acquisisci_op**, altrimenti rimarrà **idle**.
- **acquisisci_op**: Questo è lo stato in cui l'unità di controllo sta acquisendo gli operandi per l'operazione successiva. Le azioni associate a questo stato sono:
 - Viene alzato il segnale **loadM**, indicando che il moltiplicando deve essere caricato nel registro **M**.
 - Viene alzato il segnale **loadAQ** a 1, indicando che il moltiplicatore e gli 8 zeri iniziali e il bit Q_{-1} devono essere caricati nello shift register. Questo si verifica perché **selAQ** è a 0.
 - Dopo aver eseguito queste azioni, la macchina passerà allo stato successivo, che è **waitSR**. Quindi, l'unità di controllo si sposterà a prepararsi per eseguire operazioni successive.
- **waitSR**: Questo è lo stato in cui l'unità di controllo attende che lo shift register (**SR**) completi il caricamento degli operandi e che il sommatore sia pronto per iniziare a lavorare. L'unica azione associata a questo stato è: Dopo aver aspettato che lo shift register sia pronto, la macchina passerà allo stato successivo, che è **avvia_scan**.

- **avvia_scan:** In questo blocco **CASE** relativo allo stato **avvia_scan**, l'unità di controllo sta prendendo decisioni basate sul valore di **qLSB**:
 - Se i due bit meno significativi di **Q** sono "01", allora abbiamo la somma:
 - Imposta il segnale **selAQ** a 1 e abilita il caricamento dello shift register. Dopo aver effettuato queste azioni, la FSM passerà allo stato successivo, che è **avvia_shift**.
 - Se i due bit meno significativi di **Q** sono "10", allora abbiamo una sottrazione:
 - Imposta il segnale **subtract** a 1. Le altre azioni (**selAQ** e **loadAQ**) sono le stesse di prima. Dopo aver effettuato queste azioni, la FSM passerà allo stato successivo, che è **avvia_shift**.
 - Se i due bit meno significativi di **Q** sono "00" o "11", allora la FSM passerà direttamente allo stato successivo, che è **avvia_shift**.
- **avvia_shift:** quando la macchina è nello stato **avvia_shift**, abilita lo shift, e a seconda se il contatore ha raggiunto il valore massimo o meno, passerà allo stato **fine** o allo stato **incr_count** per continuare il processo.
- **Incr_count:** quando la FSM è nello stato **incr_count**, abilita l'incremento del contatore (**count_in <= '1'**) e si prepara per una nuova iterazione, passando allo stato **avvia_scan**. Questo ciclo continuerà fino a quando il processo sarà completato, e alla fine passerà allo stato **fine** e successivamente a **idle** terminando l'esecuzione.

Dopo aver implementato anche l'unità di controllo possiamo passare all'implementazione del moltiplicatore di Booth nella sua interezza.

L'entity dichiarata è la seguente:

```
entity molt_rob is
    port( clock, reset, start: in std_logic;
          X, Y: in std_logic_vector(7 downto 0);
          P: out std_logic_vector(15 downto 0);
          stop_cu: out std_logic);
end molt_rob;
```

Figura 79 entity moltiplicatore di booth

L'architettura è stata definita con un approccio strutturale componendo unità operativa e di controllo.

Nella parte finale dell'implementazione gestiamo l'output della moltiplicazione ed effettuiamo l'invio di informazioni all'unità di controllo tramite il segnale **tempqLSB**.

```
tempqLSB<=temp_p(1 downto 0); --invio all'unità di controllo il bit meno significativo del
--registro A.Q
P<=temp_p(16 downto 1);

-- la UO viene resettata sia se arriva un reset dall'esterno sia se l'operazione di
moltiplicazione termina
--temp_reset_in <= reset or temp_stop_cu;

stop_cu <= temp_stop_cu;
end structural;
```

Simulazione

Dichiariamo i segnali che utilizzeremo per controllare e monitorare il comportamento del sistema durante la simulazione.

```
signal inputx, inputy: std_logic_vector(7 downto 0);
signal prod: std_logic_vector(15 downto 0);
signal clk, res, start: std_logic;
signal t_stop_cu: std_logic;
constant clk_period : time := 20 ns;
signal end_sim : std_logic := '0';
```

Figura 80 dichiarazione segnali

Successivamente mappiamo questi segnali al componente principale che viene etichettato come **uut**:

```
uut: molt_rob port map(clk, res, start, inputx, inputy, prod, t_stop_cu);
```

Figura 81 mapping del componente

In seguito, definiamo un processo che generare un segnale di clock periodico:

```
clk_process : process
begin
  while (end_sim = '0') loop
    clk<= '1';
    wait for clk_period/2;
    clk <= '0';
    wait for clk_period/2;
  end loop;
  wait;
end process;
```

Figura 82 generazione clock periodico

Il processo **clk_process** viene eseguito in un loop continuo fintanto che **end_sim** è '0'. Durante ogni iterazione del loop. Infine, definiamo il processo di simulazione **sim**:

```

sim: process
begin

  wait for 100 ns;

  res<='1';
  wait for 20 ns;
  res<='0';

  -- -----
  -- operazione numero 1:
  -- 15*3=45 (002D)
  inputx<="00001111";
  inputy<="00000011";

  -- start deve essere visto da clk_div: poiché sarà generato dal button debouncer si
  aggiungerà anche il clk_div
  -- al button debouncer e il segnale di start deve durare quanto il periodo del clk
  rallentato
  wait for 40 ns;
  start<='1';
  wait for 20 ns;
  start<='0';
  wait;
end process;

end behavioural;

```

Figura 83 processo di simulazione

Per effettuare la simulazione assegniamo dei valori specifici a moltiplicatore e a moltiplicando ovvero 15 e 3.

Lanciando la simulazione otteniamo il seguente risultato:

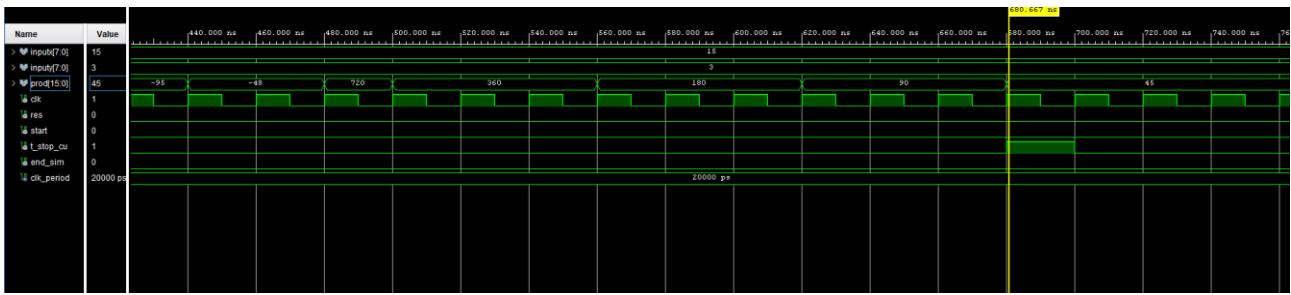


Figura 84 simulazione moltiplicatore di booth

Come possiamo vedere dal grafico i valori del moltiplicando e del moltiplicatore vengono settati correttamente a 15 (inputx) e a 3 (inputy). Anche il segnale di clock periodico è stato generato correttamente. quando si alza il segnale di stop della control unit e quindi si ferma l'elaborazione, nel registro del risultato prod possiamo osservare il valore corretto della moltiplicazione, ovvero 45. Controllando a mano tutti i passaggi dell'algoritmo, è possibile verificare il corretto funzionamento del sistema.

Esercizio 7.2:

Sintetizzare il moltiplicatore implementato al punto 7.1 su FPGA e testarlo mediante l'utilizzo dei dispositivi di input/output (switch, bottoni, led, display) presenti sulla board di sviluppo in dotazione. La modalità di utilizzo degli stessi è a completa discrezione degli studenti.

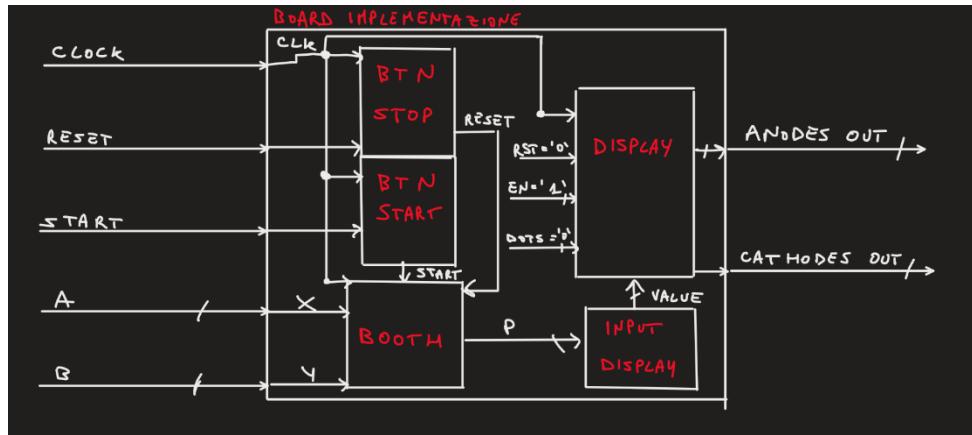
Progetto e architettura

La modalità di utilizzo scelta per il moltiplicatore di Booth è la seguente:

- operandi A e B in ingresso sugli switch;
- start e reset in ingresso sui bottoni;
- risultato della moltiplicazione in uscita sui display.

Per gestire i bottoni di start e reset abbiamo utilizzato i button debouncer, la cui implementazione è presente nell'appendice. Invece, per vedere il prodotto sui display è stato necessario implementare il componente "converti_count", che avendo in ingresso il prodotto ottenuto dal moltiplicatore lo converte in modo

opportuno. Infine, è stato utilizzato il componente “display_seven_segments” trovato nel materiale didattico. Quindi lo schema del componente “board_implementazione”, che comprende il moltiplicatore, i due button debouncer, il convertitore e il gestore del display è il seguente:



Implementazione

Il componente “board_implementazione” prende in ingresso i seguenti segnali:

- clock della board;
- bottone di reset;
- bottone di start;
- operando A (8 bit dagli switch);
- operando B (8 bit dagli switch).

E in uscita fornisce i bit per gestire anodi e catodi dei display.

Al suo interno vengono istanziate le componenti già citate, in modo da poter collegare i segnali per gestire il comportamento desiderato.

La sua implementazione in VHDL è la seguente:

```
entity board_implementazione is
  Port (
    clock, reset, start: in std_logic;
    A, B: in std_logic_vector(7 downto 0);
    anodes_out : out std_logic_vector (7 downto 0); --anodi e catodi delle cifre, sono un output del topmodule
    cathodes_out : out std_logic_vector (7 downto 0)
  );
end board_implementazione;

architecture Behavioral of board_implementazione is

  component molt_rob is
    port(
      clock, reset, start: in std_logic;
      X, Y: in std_logic_vector(7 downto 0);
      --stop: out std_logic; --a che serve?
      P: out std_logic_vector(15 downto 0);
      stop_cu: out std_logic
    );
  end component;

  component button_debouncer is
    generic (
      CLK_period: integer := 10; -- periodo del clock (della board) in nanosecondi
      btn_noise_time: integer := 10000000 -- durata stimata dell'oscillazione del bottone in nanosecondi
      | -- il valore di default → 10 millisecondi
    );
    Port (
      RST : in STD_LOGIC;
      CLK : in STD_LOGIC;
      BTN : in STD_LOGIC;
      CLEARED_BTN : out STD_LOGIC
    );
  end component;
```

```

component display_seven_segments is
Generic(
    CLKIN_freq : integer := 100000000;
    CLKOUT_freq : integer := 500
);
Port ( CLK : in STD_LOGIC;
       RST : in STD_LOGIC;
       VALUE : in STD_LOGIC_VECTOR (31 downto 0);
       ENABLE : in STD_LOGIC_VECTOR (7 downto 0); -- decide quali cifre abilitare
       DOTS : in STD_LOGIC_VECTOR (7 downto 0); -- decide quali punti visualizzare
       ANODES : out STD_LOGIC_VECTOR (7 downto 0);
       CATHODES : out STD_LOGIC_VECTOR (7 downto 0));
end component;

component converti_count is
Port (
    prodotto : in std_logic_vector(15 downto 0);
    outp : out std_logic_vector(31 downto 0)
);
end component;

signal btn_start_pulito, btn_reset_pulito : std_logic := '0';
signal temp_P : std_logic_vector(15 downto 0);
signal value_temp : std_logic_vector(31 downto 0);

```

```

begin

    BOOTH : molt_rob
    port map(
        clock => clock,
        reset => btn_reset_pulito,
        start => btn_start_pulito,
        X => A,
        Y => B,
        P => temp_P
    );

    BTN_START : button_debouncer
    generic map(
        CLK_period => 10,
        btn_noise_time => 10000000
    )
    Port map(
        RST => '0',
        CLK => clock,
        BTN => start,
        CLEARED_BTN => btn_start_pulito
    );

    BTN_STOP : button_debouncer
    generic map(
        CLK_period => 10,
        btn_noise_time => 10000000
    )
    Port map(
        RST => '0',
        CLK => clock,
        BTN => reset,
        CLEARED_BTN => btn_reset_pulito
    );

```

```

DISPLAY : display_seven_segments
Generic map(
    CLKIN_freq => 100000000,
    CLKOUT_freq => 500
)
Port map(
    CLK => clock,
    RST => '0',
    VALUE => value_temp,
    ENABLE => "11111111",
    DOTS => "00000000",
    ANODES => anodes_out,
    CATHODES => cathodes_out
);
INPUT_DISPLAY : converti_count
Port map(
    prodotto => temp_P,
    outp => value_temp
);
end Behavioral;

```

Per capire l'importanza del componente “converti_count” mettiamo il focus sull'ingresso value del componente “display_seven_segments” che è a 32 bit e sull'uscita P del moltiplicatore di booth, a 8 bit. Tale componente deve gestire questa conversione, in modo da visualizzare le cifre del risultato ottenuto, in decimale. Per farlo siamo ricorsi alla suddivisione del risultato (convertito prima in intero) in decine e unità nel seguente modo:

- Le unità sono ottenute dal resto della divisione per 10 del risultato;
- Le decine invece dalla parte intera della divisione per 10 del risultato.

Di seguito si può vedere l'implementazione in VHDL del componente:

```

entity converti_count is
  Port (
    prodotto : in std_logic_vector(15 downto 0);
    outp : out std_logic_vector(31 downto 0)
  );
end converti_count;

architecture beh of converti_count is
  signal prodotto_u : integer;
  signal prodotto_d : integer;

  signal prodotto_tot : std_logic_vector(7 downto 0);
  signal uscita_temp : std_logic_vector(31 downto 0);

begin
  prodotto_u <= to_integer(unsigned(prodotto)) mod 10;
  prodotto_d <= to_integer(unsigned(prodotto)) / 10;

  prodotto_tot (3 downto 0) <= std_logic_vector(to_unsigned(prodotto_u, 4));
  prodotto_tot (7 downto 4) <= std_logic_vector(to_unsigned(prodotto_d, 4));

  uscita_temp(31 downto 8) <= (others => '1');
  uscita_temp(7 downto 0) <= prodotto_tot;

  outp <= uscita_temp;
end beh;

```

Gli altri bit sono posti tutti pari a '1' per ottenere i trattini '-' sul display.

Timing analysis

Design Timing Summary

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 4,756 ns	Worst Hold Slack (WHS): 0,176 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: 313	Total Number of Endpoints: 313	Total Number of Endpoints: 128

All user specified timing constraints are met.

Una volta controllato questo risultato, abbiamo calcolato anche la FMAX, cioè la frequenza massima di funzionamento. Essa non è fornita esplicitamente nei report ma l'abbiamo stimata con la seguente formula: $\frac{1}{T - WNS}$, dove T è il periodo del clock target. Per trovare questo valore è possibile diminuire progressivamente il periodo del clock di design, finché non si ottiene un WNS negativo. In particolare, abbiamo diminuito il periodo fino a 3.5 ns con una forma d'onda con fronte di salita in 0 ns e fronte di discesa a 1.75 ns

-period 3.50 -waveform {0 1.75}

Il valore approssimato è 285 MHZ. Oltre questa frequenza i vincoli temporali non sono più rispettati.

Design Timing Summary

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): -0,002 ns	Worst Hold Slack (WHS): 0,186 ns	Worst Pulse Width Slack (WPWS): 1,250 ns
Total Negative Slack (TNS): -0,002 ns	Total Hold Slack (THS): 0,000 ns	Total Pulse Width Negative Slack (TPWS): 0,000 ns
Number of Failing Endpoints: 1	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 313	Total Number of Endpoints: 313	Total Number of Endpoints: 128

Timing constraints are not met.

Esercizio 8: comunicazione con handshaking

Progettare, implementare in VHDL e testare mediante simulazione un sistema composto da 2 nodi, A e B, che comunicano mediante un protocollo di handshaking. Il nodo A e il nodo B possiedono entrambi una memoria interna in cui sono memorizzate N stringhe di M bit, denominate X(i) e Y(i) rispettivamente ($i=0,\dots,N-1$). Il nodo A trasmette a B ciascuna stringa X(i) utilizzando un protocollo di handshaking; B, ricevuta la stringa X(i), calcola $S(i)=X(i)+Y(i)$ e immagazzina la somma in opportune locazioni della propria memoria interna.

Per il progetto è possibile considerare una implementazione di tipo comportamentale per effettuare la somma, mentre è necessario prevedere esplicitamente un componente contatore sia nel sistema A sia nel sistema B per scandire la trasmissione/ricezione delle stringhe e per terminare la comunicazione.

Progetto e architettura

Per realizzare il sistema abbiamo progettato separatamente, seguendo un approccio modulare, i due nodi A e B che abbiamo identificato rispettivamente come **nodo trasmittente** e **nodo ricevente**.

Il protocollo di handshaking è responsabile di coordinare la comunicazione tra A e B. Il protocollo di handshaking viene implementato attraverso i segnali accepted e ready che indicano la conferma di ricezione e la disponibilità di trasmissione rispettivamente. La trasmissione inizia quando si alza ready e finisce quando si abbassa. Il segnale di accepted indica sia la possibilità di ricevere il dato, sia la fine della sua elaborazione. Solo alla fine dell'elaborazione viene abbassato il segnale di ready. Questo tipo di convenzione permette di risparmiare la definizione di ulteriori segnali e una buona dose di complessità nel protocollo.

Un prima schema dell'architettura dove mostriamo come avviene la comunicazione tramite protocollo handshaking è il seguente:

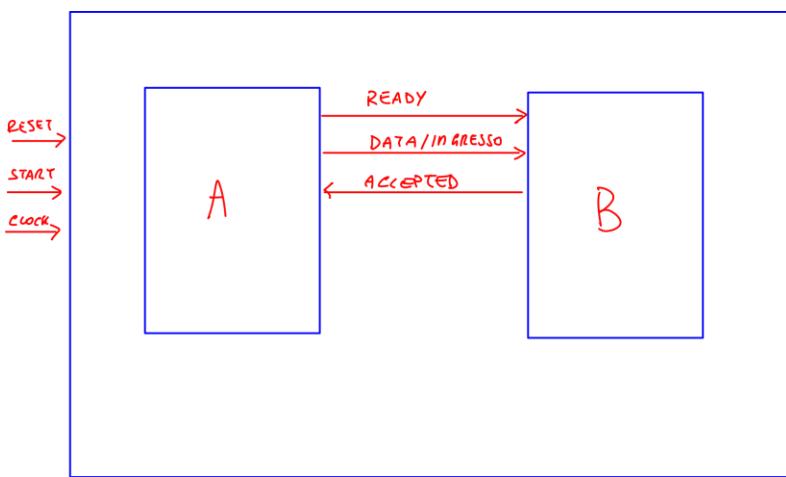


Figura 85 schema base del sistema

Quindi, il sistema è composto da due parti principali: il trasmittitore (nodo A) e il ricevitore (nodo B). Ciascun nodo è suddiviso in due blocchi principali, l'unità operativa e l'unità di controllo.

Analizziamo meglio come sono realizzati i nodi.

L'**unità operativa del trasmettitore** comprende un contatore per gestire gli indirizzi di memoria e una memoria per immagazzinare le stringhe da trasmettere. Il contatore è abilitato tramite un segnale di controllo ('enable'), e la memoria legge i dati in base all'indirizzo corrente del contatore.

L'**unità di controllo del trasmettitore** gestisce la logica di handshaking e la sequenza di operazioni. La macchina a stati finiti comprende gli stati IDLE, DATA_READY, WAIT_ACK, WAIT_ELAB, e CONTA. La transizione tra gli stati è guidata dagli input come start, count, accepted, e dagli output controllati come enable, read, fine, ready.

Per comprendere al meglio il funzionamento dell'unità di controllo abbiamo disegnato l'automa a stati finiti:

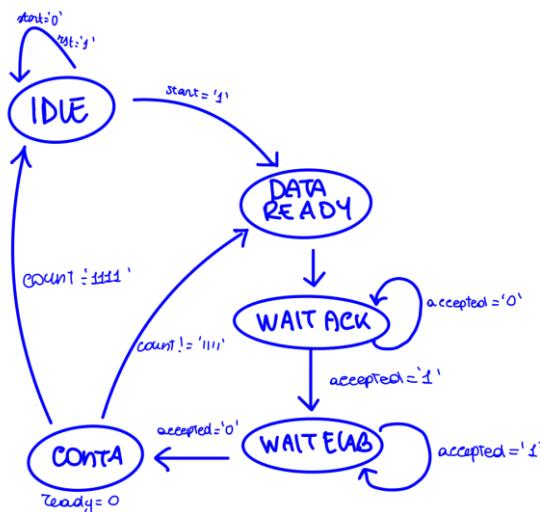


Figura 86 automa a stati finiti trasmettitore

Analizziamo meglio il funzionamento:

- **Stato iniziale – IDLE:** l'unità di controllo inizia in uno stato di riposo, chiamato IDLE. In questo stato, il sistema è in attesa di ricevere il segnale start che indica l'inizio di una nuova trasmissione. Se start diventa '1', la FSM passa allo stato successivo per prepararsi alla trasmissione dei dati.
- **Stato di trasmissione - DATA_READY:** una volta ricevuto il segnale start, il sistema entra nello stato DATA_READY. In questo stato, l'unità di controllo configura l'ambiente di trasmissione. Disabilita il segnale enable per garantire che il contatore non si muova durante la trasmissione, attiva il segnale read per leggere i dati dalla memoria e imposta ready a '1' per segnalare che il trasmettitore è pronto a trasmettere dati.
- **Stato di attesa della ACK - WAIT_ACK:** dopo aver preparato l'ambiente di trasmissione, il sistema entra nello stato WAIT_ACK. In questo stato, l'unità di controllo attende il segnale accepted dall'unità ricevente. Se l'unità ricevente è pronta ad accettare i dati (quindi accepted diventa '1'), passa allo stato successivo per iniziare l'elaborazione dei dati. Altrimenti, rimane in attesa.
- **Stato di attesa elaborazione - WAIT_ELAB:** una volta ricevuto il segnale accepted, il sistema entra nello stato WAIT_ELAB. In questo stato, l'unità di controllo aspetta che l'unità ricevente elabori i dati. Se accepted diventa '0', indica che l'elaborazione è completa, e il sistema passa allo stato CONTA. Altrimenti, rimane in attesa.
- **Stato di conteggio – CONTA:** in questo stato, l'unità di controllo abilita nuovamente il segnale enable per permettere al contatore di avanzare. Se il contatore raggiunge il valore "1111" (indicando che

sono stati trasmessi tutti i dati in memoria), il sistema passa allo stato IDLE. In caso contrario, torna allo stato DATA_READY per prepararsi alla trasmissione successiva.

Adesso analizziamo il ricevitore.

L'**unità operativa** del ricevitore include un contatore, e due registri per memorizzare le stringhe da usare per l'elaborazione della somma, e una memoria per immagazzinare la somma delle stringhe ricevute. Il contatore è abilitato tramite un segnale di controllo enable, i registri sono caricati secondo il segnale load, e la memoria legge e scrive in base all'indirizzo corrente del contatore.

L'**unità di controllo** del ricevitore gestisce la logica di handshaking e la sequenza di operazioni. La macchina a stati finiti comprende gli stati IDLE, WAIT_DATA_READ, ACK, ELAB, WAIT_FINE, e CONTA, duali rispetto al trasmettitore. La transizione tra gli stati è guidata dagli input start, count, ready, e dagli output controllati come enable, read, write, load, accepted, somma, fine.

Anche in questo caso abbiamo definito l'automa a stati finiti:

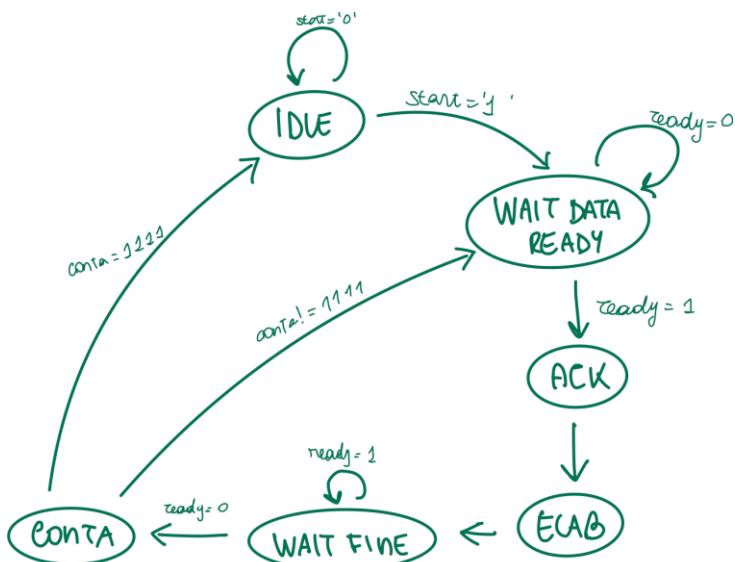


Figura 87 automa a stati finiti

Analizziamo i vari stati:

- **Stato iniziale – IDLE:** nello stato iniziale IDLE, l'unità di controllo è in attesa del segnale start. Se start diventa '1', indica l'inizio di una nuova trasmissione. In tal caso, la FSM passa allo stato successivo, WAIT_DATA_READY, altrimenti rimane nello stato IDLE.
- **Stato di attesa di trasmissione - WAIT_DATA_READY:** nello stato WAIT_DATA_READY, il segnale enable viene disabilitato, e il segnale read viene attivato per leggere dalla memoria in modo che questo sia già pronto per l'elaborazione. La FSM attende che il segnale ready diventi '1', indicando che i dati sono pronti per essere trasmessi. Quando ready è '1', viene attivato il segnale load e la FSM passa allo stato ACK. Se ready rimane '0', la FSM rimane nello stato WAIT_DATA_READY.
- **Stato di invio della ACK – ACK:** nello stato ACK, i segnali load e read vengono disattivati. Viene alzato il segnale accepted per l'invio della ACK, e il segnale somma. La FSM passa quindi allo stato successivo, ELAB.

- **Stato di elaborazione – ELAB:** nello stato ELAB, il segnale somma viene disattivato e il segnale write viene attivato, indicando che l'unità ricevente può scrivere la somma nella propria memoria interna. La FSM passa allo stato WAIT_FINE.
- **Stato di attesa di conferma della fine della trasmissione - WAIT_FINE:** nello stato WAIT_FINE, il segnale accepted viene disattivato e il segnale write viene spento. La FSM attende che il segnale ready diventi '0', indicando che l'unità trasmittente ha ricevuto l'ACK. La FSM passa quindi allo stato CONTA. Se ready è ancora '1', rimane in attesa nello stesso stato.
- **Stato di aggiornamento dell'indirizzo di memoria – CONTA:** nello stato CONTA, la FSM abilita nuovamente il segnale enable per permettere al contatore di avanzare. Se il contatore raggiunge il valore "1111", la FSM indica la fine delle trasmissioni impostando fine a '1' e torna allo stato IDLE. Altrimenti, torna allo stato WAIT_DATA_READY per iniziare la ricezione successiva.

Lo schema dell'architettura è il seguente:

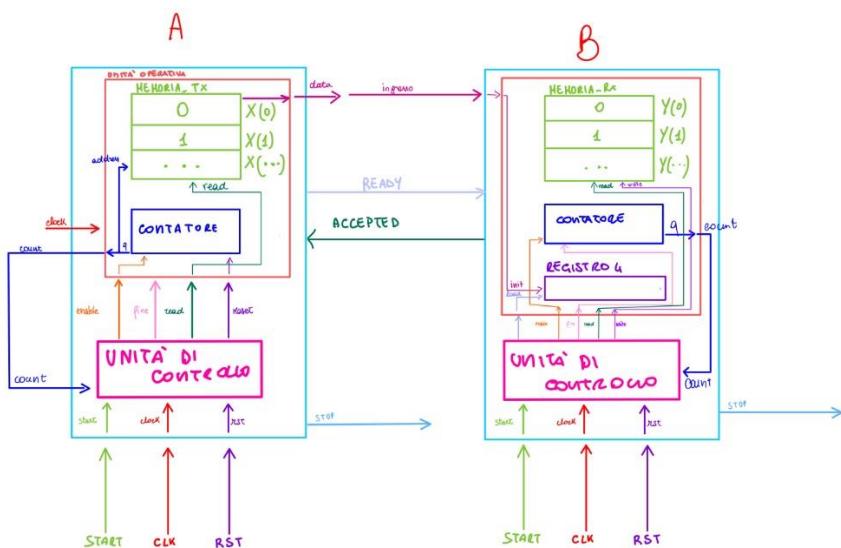


Figura 88: schema architettonico

Implementazione

Come già spiegato, abbiamo seguito un approccio modulare per l'architettura del sistema. Per analizzare l'implementazione consideriamo un modulo alla volta dell'architettura. Non specifichiamo l'implementazione del contatore modulo 16 in quanto è presente nell'appendice.

```

entity memoria_tx is
generic(
    word_lenght : positive := 4;
    depth : positive := 4;
    numero_locazioni : positive := 2**4
);

port(
    clk : in std_logic;
    address : in std_logic_vector(depth-1 downto 0);      --indirizzo della
    locazione da sovrascrivere
    read : in std_logic;
    data : out std_logic_vector (word_lenght-1 downto 0) --in uscita viene
    presentato l'ultimo valore scritto
);
end memoria_tx;

```

Figura 89 entity memoria trasmettitore

La dichiarazione dell'entità memoria_tx specifica le caratteristiche principali del componente, inclusi i parametri generici e le porte di input/output.

I generici dichiarati sono 3:

- **word_lenght:** specifica la lunghezza di ciascuna parola di memoria, ovvero il numero di bit in ogni locazione di memoria. È un parametro generico con valore predefinito di 4.
- **depth:** indica la profondità della memoria, ovvero il numero totale di locazioni di memoria disponibili. È un parametro generico con valore predefinito di 4.
- **numero_locazioni:** calcolato come 2 elevato alla potenza di depth. Rappresenta il numero totale di locazioni di memoria possibili.

Le porte invece sono:

- **clk (std_logic in):** segnale di clock che regola l'operazione della memoria.
- **address (std_logic in):** vettore di segnali che rappresenta l'indirizzo della locazione di memoria da leggere.
- **read (std_logic in):** segnale di controllo che attiva la lettura dalla memoria quando è '1'.
- **data (std_logic_vector out):** vettore di segnali in uscita che rappresenta il valore letto dalla memoria, con la lunghezza specificata da word_lenght.

L'entità memoria_tx rappresenta una memoria a sola lettura (RAM), in cui i dati possono essere letti dalla memoria utilizzando il segnale read. I parametri generici (word_lenght e depth) forniscono la flessibilità per configurare la dimensione delle parole e la profondità della memoria. La memoria è organizzata in locazioni, ognuna delle quali contiene una parola di lunghezza specificata.

Le operazioni di lettura sono sincronizzate dal segnale di clock, e il valore letto dalla memoria viene fornito attraverso il segnale data. L'indirizzo della locazione di memoria da leggere è specificato dal segnale address.

Per la definizione dell'architettura abbiamo utilizzato un approccio behavioral:

```

architecture behavioral of memoria_tx is
    type memory_16_4 is array(0 to numero_locazioni-1) of std_logic_vector(word_lenght-1
        downto 0); --rappresenta le celle di memoria
    signal mymem : memory_16_4 := (others => "0001");

    begin

        process(clk) begin
            if (clk'event and clk = '0') then --se si abbassa il bit di abilitazione....
                if(read = '1') then
                    data <= mymem(to_integer(unsigned(address)));
                end if;
            end if;
        end process;

    end behavioral;

```

Figura 90 dichiarazione architettura

Come prima cosa dichiariamo un tipo array: **memory_16_4** è un tipo array che rappresenta le celle di memoria. Ogni elemento dell'array è una stringa di bit (std_logic_vector) con lunghezza specificata da **word_lenght**.

Successivamente dichiariamo il segnale **mymem**.

È un segnale del tipo **memory_16_4** che rappresenta effettivamente la memoria. È inizializzato con un array di lunghezza **numero_locazioni** inizializzato con "0001" (un valore arbitrario) per ogni cella di memoria.

Dopo la dichiarazione del segnale si ha l'inizio del processo.

Il processo è sensibile ai cambiamenti del segnale di clock. Viene eseguito solo quando il fronte di discesa del segnale di clock è rilevato. All'interno del processo, viene verificato se il segnale **read** è '1', indicando una richiesta di lettura dalla memoria.

Se la condizione è vera, il valore di **mymem** all'indirizzo specificato da **address** viene assegnato al segnale **data**.

Dopo aver analizzato l'implementazione della memoria passiamo all'analisi dell'implementazione dell'**unità di controllo**.

Come sempre partiamo dall'entity:

```

entity unita_controllo_tx is
    port(
        start : in std_logic;
        clk: in std_logic;
        rst : in std_logic;

        --CONTATORE
        count : in std_logic_vector(3 downto 0);
        enable : out std_logic;
        fine: out std_logic;

        --MEMORIA
        read : out std_logic;

        --HANDSHAKE
        accepted: in std_logic;
        ready:out std_logic
    );
end unita_controllo_tx;

```

Figura 91 entity unità di controllo trasmettitore

La entity **unita_controllo_tx** rappresenta l'unità di controllo per il sistema di trasmissione dati. Questa entity gestisce il flusso di controllo durante l'invio di dati, includendo il contatore, il controllo della memoria e l'implementazione del protocollo di handshake.

I port sono i seguenti:

- **start** (*std_logic in*): segnale di avvio del sistema.
- **clk** (*std_logic in*): segnale di clock.
- **rst** (*std_logic in*): segnale di reset.
- **count** (*std_logic_vector in*): vettore di 4 bit rappresentante il contatore.
- **enable** (*std_logic out*) segnale di abilitazione per il contatore.
- **fine** (*std_logic out*): segnale indicante la fine delle trasmissioni.
- **read** (*std_logic out*): segnale di lettura per la memoria.
- **accepted** (*std_logic in*): segnale che rappresenta la ACK ricevuta dal nodo ricevitore.
- **ready** (*std_logic in*): segnale di inizio trasmissione.

L'architettura behavioral implementa il comportamento interno dell'unità di controllo. La descrizione è suddivisa in due processi principali: reg_stato e comb. Il primo come già visto in precedenza aggiorna lo stato corrente.

```
architecture behavioral of unita_controllo_tx is
  type state_type is (IDLE, DATA_READY, WAIT_ACK, WAIT_ELAB ,CONTA);
  signal state, next_state : state_type := IDLE;
begin
  reg_stato: process(clk)
    begin
      if(clk'event and clk='1') then
        if(rst='1') then
          state <=idle;
        else
          state <=next_state;
        end if;
      end if;
    end process;
```

Figura 92 architettura e processo reg stato di unità di controllo trasmettitore

```
comb: process(state, start, count, accepted) begin
  case state is
    when IDLE =>
      if (start = '1') then
        next_state <= DATA_READY;
      else
        next_state <= IDLE;
      end if;

    when DATA_READY =>
      enable <= '0';
      read <= '1';
      ready <= '1';
      next_state <= WAIT_ACK;

    when WAIT_ACK =>
      read <= '0';
      if (accepted = '1') then
        next_state <= WAIT_ELAB;
      else
        next_state <= WAIT_ACK;
      end if;

    when WAIT_ELAB =>
      if (accepted = '0') then
        next_state <= CONTA;
        ready <= '0';
      else
        next_state <= WAIT_ELAB;
      end if;
    when CONTA =>
      if (count = "1111") then
        fine <= '1';
        next_state <= IDLE;
      else
        enable <= '1';
        next_state <= DATA_READY;
      end if;
  end case;
end process;
```

Figura 93 processo comb

Questo processo è responsabile dell'implementazione delle diverse fasi del protocollo di trasmissione che abbiamo descritto prima.

Dopo aver analizzato implementazione di unità operativa e unità di controllo del trasmettitore analizziamo come tutto ciò viene unito nel sistema di trasmissione tramite l'implementazione del modulo **sistema_tx**.

Definiamone l'entity:

```
entity sistema_tx is
  Port (
    start : in std_logic;
    clk: in std_logic;
    rst : in std_logic;
    stop : out std_logic;

    --HANDSHAKE
    accepted : in std_logic;
    ready : out std_logic;
    data : out std_logic_vector(3 downto 0)
  );
end sistema_tx;
```

Figura 94 entity sistema trasmissione

```
architecture Behavioral of sistema_tx is
component unita_operativa_tx is
  port (
    clk: in std_logic;
    rst : in std_logic;

    --CONTATORE
    count : out std_logic_vector(3 downto 0);
    enable : in std_logic;
    fine: in std_logic;

    --MEMORIA
    read : in std_logic;
    data : out std_logic_vector(3 downto 0)
  );
end component;

component unita_controllo_tx is
  port(
    start: in std_logic;
    clk: in std_logic;
    rst : in std_logic;

    --CONTATORE
    count : in std_logic_vector(3 downto 0);
    enable : out std_logic;
    fine: out std_logic;

    --MEMORIA
    read : out std_logic;

    --HANDSHAKE;
    accepted : in std_logic;
    ready:out std_logic
  );
end component;
```

Figura 95 definizione dei componenti all'interno dell'architettura

La dichiarazione dei componenti **unita_operativa_tx** e **unita_controllo_tx** indica che sono componenti esterni che verranno utilizzati all'interno dell'architettura principale (**sistema_tx**).

In seguito, dichiariamo i segnali necessari per collegare le porte dei componenti:

```

signal temp_count : std_logic_vector(3 downto 0);
signal temp_enable : std_logic;
signal temp_fine : std_logic;
signal temp_read : std_logic;
signal temp_load : std_logic;

```

Figura 96 dichiarazione dei segnali

Dopo la dichiarazione dei segnali procediamo con l'instanziare i componenti:

```

begin
    CU: unita_controllo_tx
    port map(
        start => start,
        clk => clk,
        rst => rst,

        --CONTATORE
        count => temp_count,
        enable => temp_enable,
        fine => temp_fine,

        --MEMORIA
        read => temp_read,

        --HANDSHAKE
        accepted => accepted,
        ready => ready
    );

    U0 : unita_operativa_tx
    port map(
        clk => clk,
        rst => rst,

        --CONTATORE
        count => temp_count,
        enable => temp_enable,
        fine => temp_fine,

        --MEMORIA
        read => temp_read,
        data => data
    );

    stop <= temp_fine;
end Behavioral;

```

Figura 97 istanziazione dei componenti

Dopo aver analizzato il sistema di trasmissione passiamo all'analisi del **sistema di ricezione**. Anche il sistema di ricezione è composto da unità operativa ed unità di controllo. L'unità operativa è composta da contatore, memoria e registro a 4 bit. Anche in questo caso per l'implementazione del contatore rimandiamo all'appendice. L'implementazione della memoria_rx è molto simile all'implementazione della memoria del trasmittitore con l'unica differenza che non è read-only.

```

entity memoria is
    generic(
        word_length : positive := 4;
        depth : positive := 4;
        numero_locazioni : positive := 2**4
    );
    port(
        clk : in std_logic;
        address : in std_logic_vector(depth-1 downto 0);      --indirizzo della locazione da sovrascrivere
        stringa : in std_logic_vector(word_length-1 downto 0); --stringa in ingresso
        write: in std_logic;                                     --segnale di scrittura della stringa presente all'indirizzo di memoria
        read : in std_logic;
        data : out std_logic_vector (word_length-1 downto 0) --in uscita viene presentato l'ultimo valore scritto
    );
end memoria;

```

Figura 98 entity memoria ricevitore

```

architecture behavioral of memoria is
begin
process(clk) begin
    if (clk'event and clk = '0') then --se si abbassa il bit di abilitazione...
        if(write = '1') then
            mymem(to_integer(unsigned(address))) <= stringa;
            data <= stringa;
        end if;
        if(read = '1') then
            data <= mymem(to_integer(unsigned(address)));
        end if;
    end if;
end process;
end behavioral;

```

Figura 99 architettura ricevitore

Dopo aver analizzato l'implementazione della memoria passiamo all'implementazione del registro. Questo è del tutto simile al registro visto nell'implementazione di booth solo che è a 4 bit.

```

entity registro4 is
port (
    ingresso : in std_logic_vector(3 downto 0);
    clk : in std_logic;
    load : in std_logic;
    rst : in std_logic;
    data : out std_logic_vector(3 downto 0)
);
end registro4;

```

Figura 100 entity registro

```

architecture Behavioral of registro4 is
begin
process(clk) begin
    if (clk = '1' and clk'event) then
        if(rst = '1') then
            temp <= "0000";
        elsif(load = '1') then
            temp <= ingresso;
        end if;
    end if;
end process;
data <= temp;
end Behavioral;

```

Figura 101 architettura registro

Dopo aver visto i moduli dell'unità operativa analizziamo l'implementazione di quest'ultima. L'entity è la seguente:

```

entity unita_operativa_rx is
port (
    clk: in std_logic;
    rst : in std_logic;

    --CONTATORE
    count : out std_logic_vector(3 downto 0);
    enable : in std_logic;
    fine: in std_logic;

    --MEMORIA
    write : in std_logic;
    read : in std_logic;
    somma : in std_logic;

    --REGISTRO
    load : in std_logic;
    init : in std_logic_vector(3 downto 0)
);
end unita_operativa_rx;

```

Figura 102 entity unità operativa ricevitore

Come prima cosa dichiariamo i componenti:

```
architecture Structural of unita_operativa_rx is

component contatore_mod16 is
    generic(
        n : positive := 4;
        max : positive := 2**4-1
    );
    port(
        en : in std_logic;
        clk: in std_logic;
        rst : in std_logic;
        q : out std_logic_vector(n-1 downto 0)
    );
end component;

component registro4 is
    port (
        ingresso : in std_logic_vector(3 downto 0);
        clk : in std_logic;
        load : in std_logic;
        rst : in std_logic;
        data : out std_logic_vector(3 downto 0)
    );
end component;

component memoria is
    generic(
        word_length : positive := 4;
        depth : positive := 4;
        numero_locazioni : positive := 2**4
    );
    port(
        clk : in std_logic;
        address : in std_logic_vector(depth-1 downto 0); --indirizzo della locazione da sovrascrivere
        stringa : in std_logic_vector(word_length-1 downto 0); --stringa in ingresso
        write: in std_logic; --segnale di scrittura della stringa presente all'indirizzo di memoria
        read : in std_logic;
        data : out std_logic_vector (word_length-1 downto 0) --in uscita viene presentato l'ultimo valore scritto
    );
end component;
```

Figura 103 architettura strutturale

Poi definiamo i segnali temporanei necessari per collegare le porta dei componenti:

```
signal temp_X : std_logic_vector(3 downto 0) := (others => '0');
signal temp_Y : std_logic_vector(3 downto 0) := (others => '0');
signal temp_count : std_logic_vector(3 downto 0):= (others => '0');
signal temp_somma : std_logic_vector(3 downto 0) := (others => '0');
signal temp_data : std_logic_vector(3 downto 0) := (others => '0');
```

Figura 104 segnali interni

Dopo aver definito i segnali possiamo effettuare i collegamenti e quindi istanziare i componenti:

```
begin
    registroX : registro4
    port map(
        ingresso => init,
        clk => clk,
        load => load,
        rst => rst,
        data => temp_X
    );

    registroY : registro4
    port map(
        ingresso => temp_data,
        clk => clk,
        load => load,
        rst => rst,
        data => temp_Y
    );

    mem : memoria
    port map(
        clk => clk,
        address => temp_count,
        stringa => temp_somma,
        write => write,
        read => read,
        data => temp_data
    );

    cont : contatore_mod16
    port map(
        en => enable,
        clk => clk,
        rst => rst,
        q => temp_count
    );

    temp_somma <= std_logic_vector(unsigned(temp_X)+ unsigned(temp_Y) ) when (somma = '1');
    count <= temp_count;
end structural;
```

Figura 105 istanziamento componenti

Per gestire i dati in ingresso e supportare l'inizializzazione del sistema, vengono impiegati due istanze di registro4. La prima istanza è registroX e la seconda registroY.

registroX:

- **ingresso => init**: Inizializzazione del registro con il segnale init che rappresenta il dato ricevuto.
- **clk => clk**
- **load => load**: Abilitazione del caricamento dei dati.
- **rst => rst**
- **data => temp_X**: Output del registro, collegato a temp_X.

registroY:

- **ingresso => temp_data**: Dati in ingresso dal componente memoria.
- **clk => clk**
- **load => load**: Abilitazione del caricamento dei dati.
- **rst => rst**: Segnale di reset per azzerare il registro.
- **data => temp_Y**: Output del registro, collegato a temp_Y

I collegamenti effettuati con il componente della memoria sono i seguenti:

- **clk => clk**
- **address => temp_count**: indirizzo di memoria, determinato da temp_count.
- **stringa => temp_somma**: dati da scrivere in memoria, derivati dalla somma di temp_X e temp_Y.
- **write => write**: segnale per l'operazione di scrittura.
- **read => read**: segnale per l'operazione di lettura.
- **data => temp_data**: dati letti dalla memoria, collegati a temp_data.

Per il contatore invece avremo:

- **en => enable**: Abilitazione del contatore.
- **clk => clk**
- **rst => rst**
- **q => temp_count**: Output del contatore, collegato a temp_count.

Dopo aver istanziato i componenti definiamo la logica della somma.

temp_X e temp_Y sono sommati quando il segnale somma è attivo ('1'). Il risultato viene memorizzato in temp_somma.

Adesso analizziamo l'implementazione dell'**unità di controllo**, anche in questo caso rispecchia la FSM descritta prima:

```

< entity unita_controllo_rx is
  port(
    start: in std_logic;
    clk: in std_logic;
    rst : in std_logic;

    --CONTATORE
    count : in std_logic_vector(3 downto 0);
    enable : out std_logic;
    fine: out std_logic;

    --MEMORIA
    write : out std_logic;
    read : out std_logic;
    somma : out std_logic;
    | --REGISTRO
    load : out std_logic;

    --HANDSHAKE
    accepted : out std_logic;
    ready: in std_logic
  );
end unita_controllo_rx;

```

Figura 106 entity unità di controllo

Passiamo alla definizione dell'architettura tramite approccio behavioral:

```

architecture behavioral of unita_controllo_rx is
  type state_type is (IDLE, ACK, WAIT_DATA_READY, CONTA, ELAB, WAIT_FINE);
  signal state, next_state : state_type := IDLE;
begin
  reg_state: process(clk)
    begin
      if(clk'event and clk='1') then
        if(rst='1') then
          state <=idle;
        else
          state <=next_state;
        end if;
      end if;
    end process;

```

Figura 107 architettura

Adesso vediamo il processo **comb**:

```

comb: process(state, start, count, ready) begin
  case state is
    when IDLE =>
      if (start = '1') then
        next_state <= WAIT_DATA_READY;
      else
        next_state <= IDLE;
      end if;

    when WAIT_DATA_READY =>
      enable <= '0';
      read <= '1';
      if(ready = '1') then
        load <= '1';
        next_state <= ACK;
      else
        next_state <= WAIT_DATA_READY;
      end if;

    when ACK =>
      load <= '0';
      read <= '0';
      accepted <= '1';
      somma <= '1';
      next_state <= ELAB;

    when ELAB =>
      somma <= '0';
      write <= '1';
      next_state <= WAIT_FINE;

    when CONTA =>
      if (count = "1111") then
        fine <= '1';
        next_state <= IDLE;
      else
        enable <= '1';
        next_state <= WAIT_DATA_READY;
      end if;
  end case;
end process;
end behavioral;

```

Ora che abbiamo analizzato l'implementazione sia di unità di controllo che unità operativa mettiamo tutto insieme nell'implementazione di **sistema_rx**. L'entità è la seguente:

```
entity sistema_rx is
  Port (
    start : in std_logic;
    clk: in std_logic;
    rst : in std_logic;
    stop : out std_logic;

    --HANDSHAKE
    accepted : out std_logic;
    ready : in std_logic;

    ingresso : in std_logic_vector(3 downto 0)
  );
end sistema_rx;
```

Figura 108 entity sistema rx

L'istanziazione dei componenti è pressocché identica a quella del sistema trasmettitore.

Simulazione

Dopo aver dichiarato i componenti da testare ovvero il sistema trasmettitore e il sistema ricevente, dichiariamo i segnali che utilizziamo per manipolare il nostro sistema in modo da effettuare il test:

```
signal startA  : std_logic;
signal startB : std_logic;
signal clockA: std_logic;
signal clockB: std_logic;
constant PERIODA : time := 20 ns;
constant PERIODB : time := 25 ns;
signal reset : std_logic;
signal fineA : std_logic;
signal fineB : std_logic;
signal input : std_logic_vector(3 downto 0);

--HANDSHAKE
signal ack : std_logic;
signal data_ready : std_logic;

signal end_sim : std_logic := '0';
```

Figura 109 segnali interni testbench

periodA e **periodB**: sono costanti di periodo che specificano la durata di un ciclo di clock per i moduli TX e RX. periodA indica il periodo del clock del modulo TX, mentre periodB indica il periodo del clock del modulo RX. **input** è un segnale utilizzato dal modulo TX per trasmettere dati al modulo RX.

Utilizzando questi segnali creiamo le istanze dei componenti sotto test:

```
A : sistema_tx
  Port map(
    start => startA,
    clk => clockA,
    rst => reset,
    stop => fineA,
    --HANDSHAKE
    accepted => ack,
    ready => data_ready,
    data => input
  );
B : sistema_rx
  Port map(
    start => startB,
    clk => clockB,
    rst => reset,
    stop => fineB,
    --HANDSHAKE
    accepted => ack,
    ready => data_ready,
    ingresso => input
  );
```

Figura 110 port mapping per istanziare i componenti

Successivamente dichiariamo due processi per la generazione del clock per il sistema trasmittitore e per il sistema ricevente:

```
clkA_proc : process begin
  while(end_sim = '0') loop
    clockA <= '1';
    wait for PERIODA/2;
    clockA <= '0';
    wait for PERIODA/2;
  end loop;
  wait;
end process;
clkB_proc : process begin
  while(end_sim = '0') loop
    clockB <= '1';
    wait for PERIODB/2;
    clockB <= '0';
    wait for PERIODB/2;
  end loop;
  wait;
end process;
```

Figura 111 processi per la generazione del clock

Infine, definiamo il processo di simulazione:

```
stim_proc : process begin
  reset <= '0';
  startA <= '1';
  startB <= '1';
  wait for 30 ns;
  startA <= '0';
  startB <= '0';
  wait for 2050 ns;
  end_sim <= '1';
  wait;
end process;
end Behavioral;
```

Figura 112 processo di simulazione

Questo processo è responsabile di fornire i segnali di controllo iniziali per la simulazione. Inizialmente, imposta reset a '0', startA e startB a '1', attende per 30 ns e poi abbassa startA e startB a '0'. Dopo ulteriori 2050 ns, imposta end_sim a '1', indicando che la simulazione dovrebbe terminare. Abbiamo scelto di aspettare tutte le trasmissioni.

Lanciando la simulazione otteniamo i seguenti risultati:

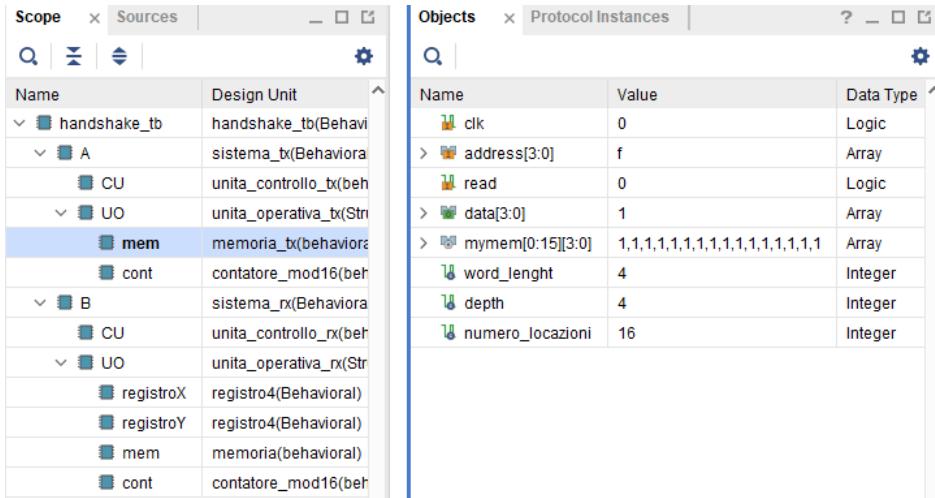


Figura 113 memoria trasmittente

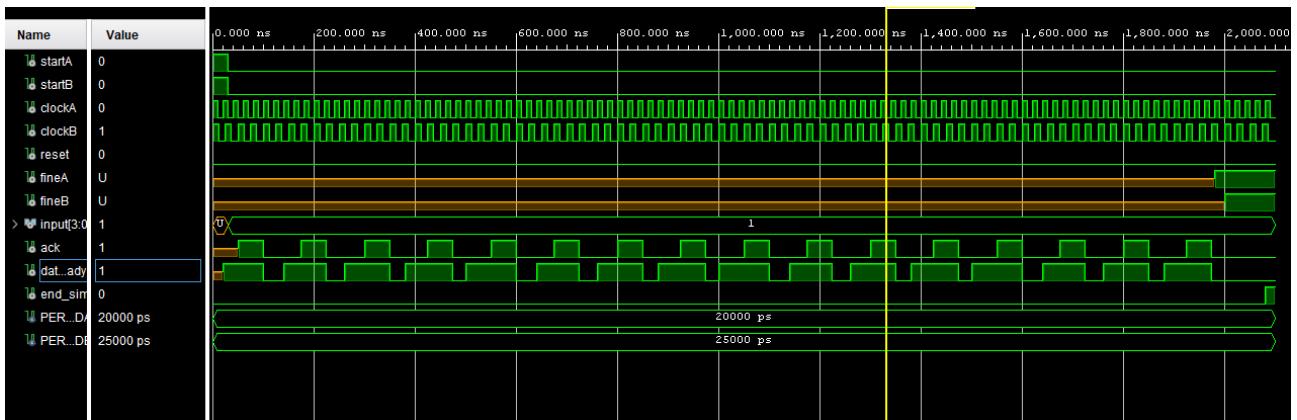


Figura 114 simulazione handshake

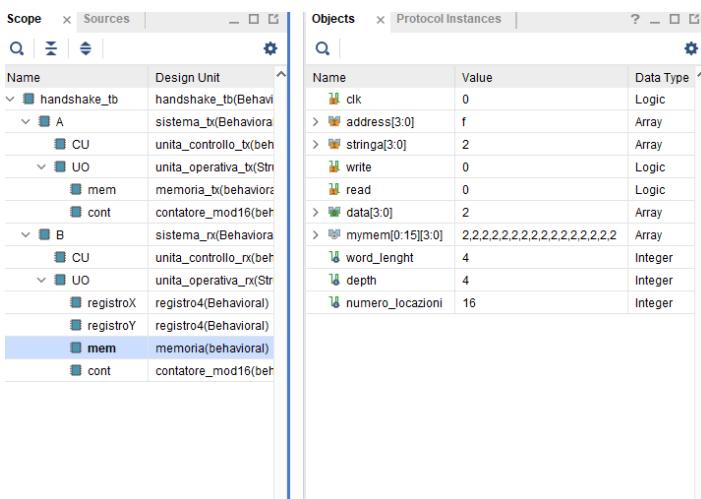


Figura 115 memoria ricevente

Come si vede dalla simulazione e il contenuto delle memorie, l'unità A ha trasmesso tutto il contenuto della sua memoria e l'unità B ha ricevuto i segnali, eseguito la somma, e scritto il risultato correttamente in memoria.

Esercizio 9: Processore

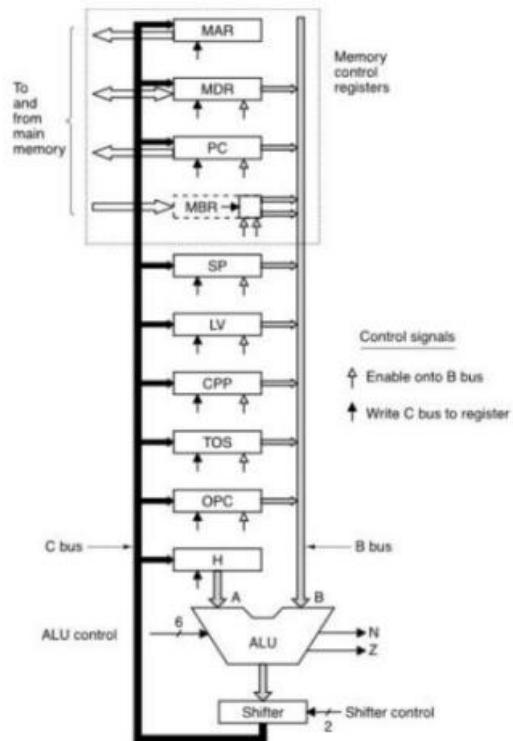
A partire dall'implementazione fornita del processore operante secondo il modello IJVM,

- si proceda all'analisi dell'architettura mediante simulazione e si approfondisca lo studio del suo funzionamento per due istruzioni a scelta,
- si modifichi un codice operativo a scelta, documentando tutte le modifiche effettuate

Analisi dell'architettura e istruzioni

Il processore Mic-1 nasce come interprete hardware del bytecode. Esso presenta un'architettura a stack e una logica microprogrammata. Il livello della microarchitettura descrive come le istruzioni *ISA* (Instruction Set Architecture) vengono interpretate ed eseguite dall'hardware. Ogni istruzione è dotata di 1 o 2 campi. Il primo campo dell'istruzione è l'*OPCODE* che identifica l'istruzione univocamente, il secondo campo, se presente, specifica gli operandi. L'istruzione è tratta dal programma assembler scritto da un programmatore. Per eseguire l'istruzione bisogna implementare le microistruzioni necessarie.

Il Mic-1 è composto da parte operativa (*PO*) e unità di controllo (*UC*). La *PO* è formata da dieci registri (di cui quattro per l'accesso alla memoria), due *bus* (tramite cui possono essere scambiati i dati tra i registri), l'*ALU* (per eseguire operazioni logico-aritmetiche) e uno *shift register*. Di seguito vediamo l'unità operativa del processore e ne analizziamo i componenti.

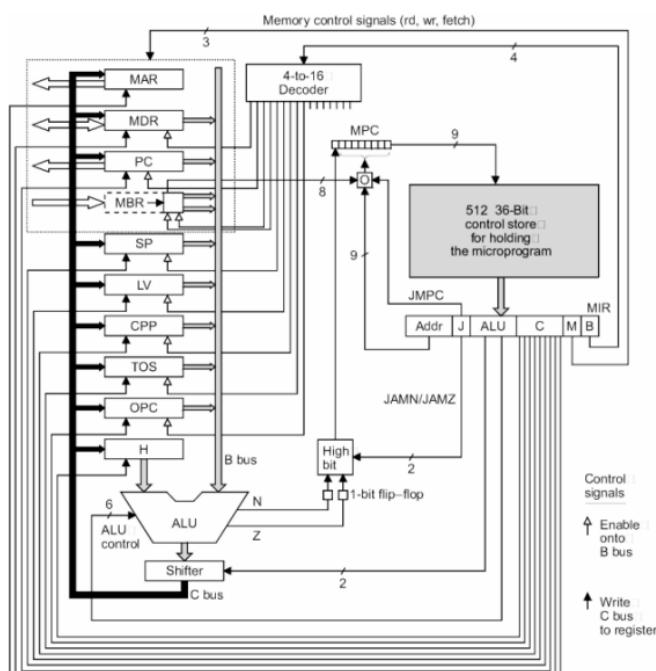


Ci sono quattro registri utilizzati per comunicare con la memoria: la coppia *MAR* (Memory Address Register) e *MDR* (Memory Data Register) per la lettura/scrittura dei dati e la coppia *PC* (Program Counter) ed *MBR* (Memory Byte Register) per leggere il programma eseguibile permettendo il fetch delle istruzioni *ISA*. *MAR* specifica l'indirizzo di memoria in cui si desidera leggere o scrivere una parola. *MDR* contiene il dato a 32 bit che sarà letto o scritto all'indirizzo di memoria specificato da *MAR*. Il *PC* è un registro a 32 bit che indica l'indirizzo di memoria della prossima istruzione *ISA* da caricare (fetch) e il *MBR* (Memory Byte Register)

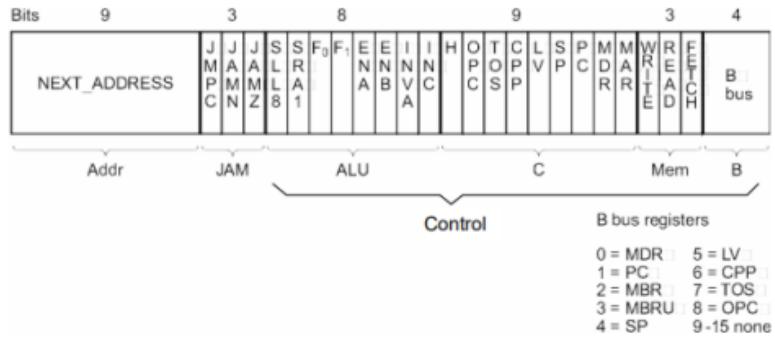
contiene il byte letto dalla memoria durante il fetch che viene caricato negli 8 bit meno significativi dei 32 bit disponibili. I registri stack pointer (*SP*) e Top of stack (*TOS*) ci permettono di accedere alla testa dello stack, precisamente al suo valore e al suo indirizzo. Il registro *LV* serve per puntare alla base attuale dello stack; *CPP* è un puntatore a valori costanti; *OPC* permette di appoggiare qualcosa di utile nella microistruzione. Il registro *H* contiene l'altro dato che viene posto in ingresso all'*ALU*. I dati contenuti sulla maggior parte dei registri possono essere scritti sul bus *B*, ma solo uno alla volta può essere abilitato sul bus *B*, che quindi agisce da multiplexer. Il dato presente sul bus attraversa l'*ALU*. L'*ALU* è caratterizzata da sei linee di controllo per determinare le operazioni da svolgere. Non tutte le $2^6 = 64$ combinazioni (delle linee di controllo *F0*, *F1*, *ENA*, *ENB*, *INVA*, *INC*) sono utili; la seguente tabella riporta quelle significative:

F₀	F₁	ENA	ENB	INVA	INC	Function
0	1	1	0	0	0	A
0	1	0	1	0	0	B
0	1	1	0	1	0	\bar{A}
1	0	1	1	0	0	\bar{B}
1	1	1	1	0	0	$A + B$
1	1	1	1	0	1	$A + B + 1$
1	1	1	0	0	1	$A + 1$
1	1	0	1	0	1	$B + 1$
1	1	1	1	1	1	$B - A$
1	1	0	1	1	1	$B - 1$
1	1	1	0	1	1	-A
0	0	1	1	0	0	$A \text{ AND } B$
0	1	1	1	0	0	$A \text{ OR } B$
0	1	0	0	0	0	0
0	1	0	0	0	1	1
0	1	0	0	1	0	-1

L'uscita dell'*ALU* viene posta in ingresso ad uno *shift register* che a sua volta produce il suo output sul bus *C*. Dal bus *C* i dati possono essere caricati anche in più registri contemporaneamente. Il comportamento dello *shift register* è controllato da due linee: *SLL8* (Shift Left Logical) che consente di shiftare il contenuto di un byte a sinistra (ponendo gli zeri negli otto bit meno significativi) e *SRA1* (Shift Right Arithmetic) che shifta il contenuto di un bit a destra mantenendo il bit più significativo invariato. Questi componenti sono pilotati dalle microistruzioni contenute nella *Control Store* presente nell'unità di controllo. Infatti, a ciascuna istruzione *ISA* corrisponde una microprocedura costituita da microistruzioni memorizzate all'interno della micromemoria.



La *Control Store* contiene 512 parole di 36 bit ciascuna, i quali corrispondono alla lunghezza della microistruzione. Notiamo come la *control word* non è di 41 bit essendo il segnale di controllo del bus B codificato su 4 bit invece che 9. Nell'unità sono presenti 2 registri fondamentali: il *MPC* (MicroProgram Counter) che specifica l'indirizzo della prossima microistruzione da eseguire e il *MIR* (MicroInstruction Register) che memorizza la microistruzione corrente i cui bit pilotano i segnali di controllo per gestire l'unità operativa. La struttura della control word è la seguente:



- **Next address**: indica qual è la successiva microistruzione da eseguire; nasce per ottimizzare la memoria, evitando così di duplicare microistruzioni comuni a più istruzioni.
- **JAM**: viene usato per gestisce i salti
- **ALU**: per decidere quali operazioni l'ALU deve effettuare
- **C**: 9 bit per abilitare i registri in lettura dal bus C
- **Mem**: per specificare le operazioni verso la memoria
- **B**: per abilitare i registri in scrittura sul bus B

Passiamo all'analisi delle istruzioni.

Abbiamo deciso di analizzare i codici operativi *BIPUSH* e *IF_ICMPEQ*. Per poter caricare i dati sullo stack il processore necessita di un'istruzione fondamentale, il *BIPUSH*.

```
bipush = 0x10:
    SP = MAR = SP + 1
    PC = PC + 1; fetch
    MDR = TOS = MBR; wr; goto main
```

- 1) Viene incrementato lo *stack pointer* così che possa puntare alla locazione conterrà il nuovo byte.
- 2) Eseguiamo il *fetch* per caricare in *MBR* l'*OPCODE* successivo
- 3) Carichiamo il contenuto di *MBR* in *TOS* e *MDR*, eseguiamo il comando *wr* per la memorizzazione all'indirizzo puntato dal nuovo *SP* caricato all'interno del *MAR* e torniamo al *main*.

Questa istruzione prevede 2 byte, uno che specifica il codice operativo e l'altro corrisponde all'operando da caricare sullo stack.

Nella microistruzione $PC = PC + 1$; *fetch* il byte operando è già stato precaricato da *Main*.

Infatti, vediamo nel *Main* la linea di codice corrispondente:

$$PC = PC + 1; fetch; goto(MBR)$$

Essa prevede l'incremento del *PC*, il *fetch* del prossimo byte (*OPCODE* successivo o operando istruzione corrente) e il salto all'indirizzo dell'istruzione presente in *MBR*.

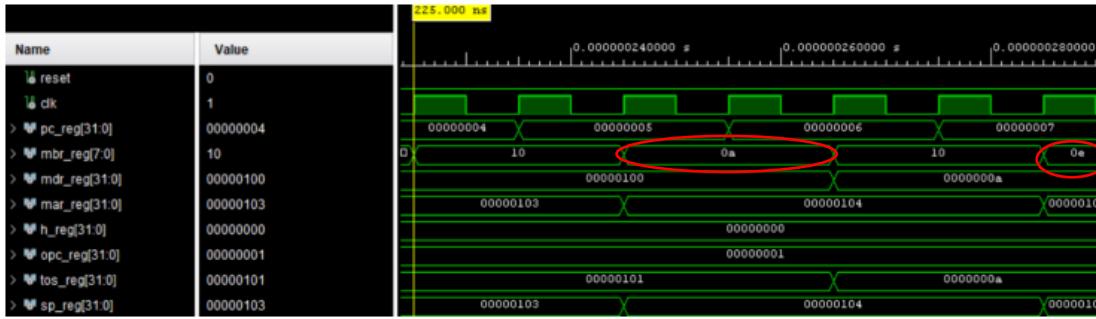
Eseguiamo *BIPUSH*, la cui istruzione è riportata in figura:

```

-- RAM content
signal mem : dp_ar_ram_type := (
--BEGIN WORDS_ENTRY
128 => "00000000000000000000000000000000",
0 => "00000001000000000000000010000000",
1 => "00001110000100000000101000010000",
2 => "10100111000000010011011001100101",
3 => "00000000000000000000000000000000",
others => (others => '0')
--END_WORDS_ENTRY
);

```

Passiamo ai risultati della simulazione, essendo già presenti le due BIPUSH nel programma, non abbiamo avuto la necessità di modificare questo file, ma ne abbiamo solo studiato il comportamento.



Abbiamo un primo bipush, il cui operando è `0xA` e un secondo bipush, il cui operando è `0xE`.

La seconda istruzione analizzata è `IF_ICMPEQ`. Questo comando esegue, a fronte di un confronto effettuato sui due ultimi operandi caricati sullo stack, un salto condizionato. Questa istruzione prevede tre byte, uno per l'`OPCODE` e due sono utilizzati come offset rispetto al `PC` per calcolare il nuovo indirizzo a cui saltare. Riportiamo di seguito la sequenza di microistruzioni:

```

if_icmpeq = 0xA1:
    MAR = SP = SP - 1; rd
    MAR = SP = SP - 1
    H = MDR; rd
    OPC = TOS
    TOS = MDR
    Z = OPC - H; if (Z) goto T; else goto F

T:
    OPC = PC - 1; fetch; goto goto_cont

F:
    PC = PC + 1
    PC = PC + 1; fetch
    goto main

    goto_cont:
    PC = PC + 1; fetch
    H = MBR << 8
    H = MBRU OR H
    PC = OPC + H; fetch
    goto main

```

- **MAR = SP = SP – 1; rd**: il primo operando (quello in cima allo stack) è già in `TOS` e quindi avvia la lettura del secondo che si trova in `SP - 1`
- **MAR = SP = SP – 1**: viene decrementato ulteriormente l'`SP` e assegnato il nuovo valore al `MAR`. Con questa operazione abbiamo fatto il `POP` dei primi due operandi dallo stack.
- **H = MDR; rd**: viene copiato in `H` il secondo operando dello stack
- **OPC = TOS**: il valore contenuto in `TOS`, che contiene già il primo operando, viene memorizzato in `OPC`
- **TOS = MDR**: viene aggiornato il nuovo valore del top dello stack a quello attuale, cioè quello sotto i due operandi, a seguito dell'operazione di lettura.
- **Z = OPC - H; if (Z) goto T; else goto F**: viene effettuata la differenza tra i due operandi e caricata in `Z`; in base al valore, se i due operandi sono uguali si procede con il ramo `True`, altrimenti si va in `False`.

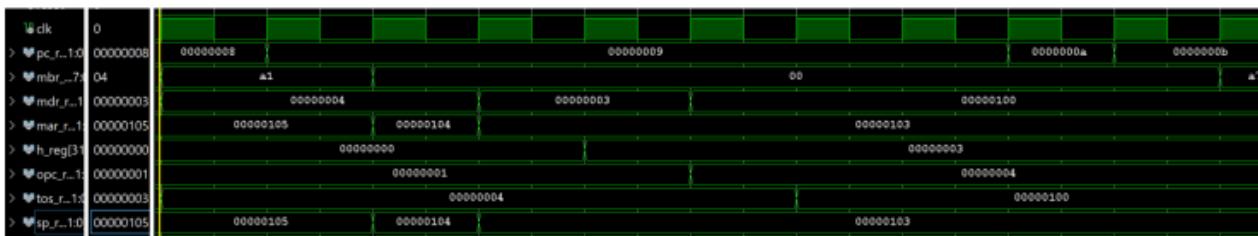
Analizziamo i due rami dell'`if`:

- **T:**
 - **OPC = PC - 1; goto goto_cont:** dato che il main incrementa anticipatamente il *PC*, per prendere l'indirizzo dell'istruzione corrente, viene decrementato il valore del *PC* e memorizzato in *OPC*. A questo punto viene effettuato un salto incondizionato alla locazione corrente più offset.
- **F:**
 - **PC = PC + 1:** viene incrementato il *PC*
 - **PC = PC + 1; fetch:** viene nuovamente incrementato il *PC*, in modo da ignorare i due byte corrispondenti all'offset. Viene effettuata la *fetch* così che, ritornando nel main, l'*MBR* sia già pronto.
 - **goto main**
- **goto_cont:**
 - **PC = PC + 1; fetch:** poiché nel main la lettura della prima parte dell'offset è già stata effettuata, si procede, incrementando il *PC* ed eseguendo la *fetch*, con la lettura della seconda
 - **H = MBR << 8:** la prima parte dell'offset, viene shiftata di 8 posizioni e caricata in *H*
 - **H = MBRU OR H:** viene fatta la *OR* tra la prima e la seconda parte dell'offset e il risultato è caricato in *H*
 - **PC = OPC + H; fetch:** il valore del *PC* viene aggiornato alla somma tra *OPC* (che contiene l'indirizzo dell'istruzione corrente) e *H* (che contiene l'offset). Con la *fetch* viene inizializzata la lettura del valore puntato dal *PC*
 - **goto main**

Per testare questa seconda istruzione, abbiamo sfruttato due valori caricati tramite due *bipush* sullo stack. I due valori caricati sono 3 e 4. Essendo diversi viene eseguito il ramo *F*.

```
-- RAM content
signal mem : dp_ar_ram_type := (
--BEGIN_WORDS_ENTRY
128 => "00000000000000000000000000000000",
0 => "00000001000000000000000010000000",
1 => "00000100000100000000000110001000",
2 => "1010011100000011000000101010001",
3 => "00000000000000000000000000000000",
others => (others => '0')
--END_WORDS_ENTRY
);
```

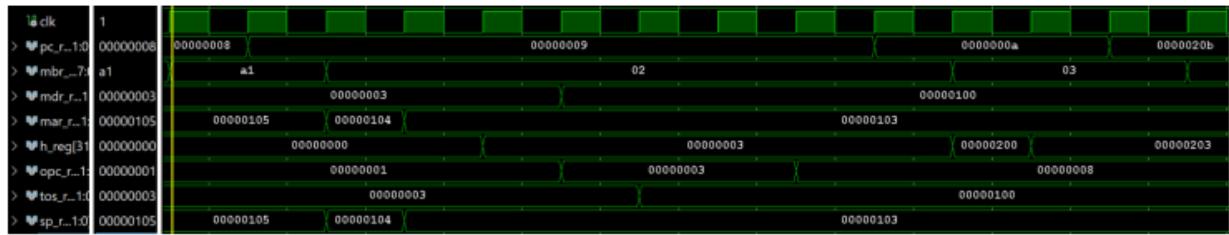
I risultati della simulazione sono i seguenti:



Per testare con due valori uguali, abbiamo caricato due 3 in cima allo stack, in questo modo viene eseguito il ramo *T*.

```
-- RAM content
signal mem : dp_ar_ram_type := (
--BEGIN_WORDS_ENTRY
128 => "00000000000000000000000000000000",
0 => "00000001000000000000000010000000",
1 => "00000011000100000000000110001000",
2 => "1010011100000011000000101010001",
3 => "00000000000000000000000000000000",
others => (others => '0')
--END_WORDS_ENTRY
);
```

I risultati della simulazione sono i seguenti:



Modifica codice operativo

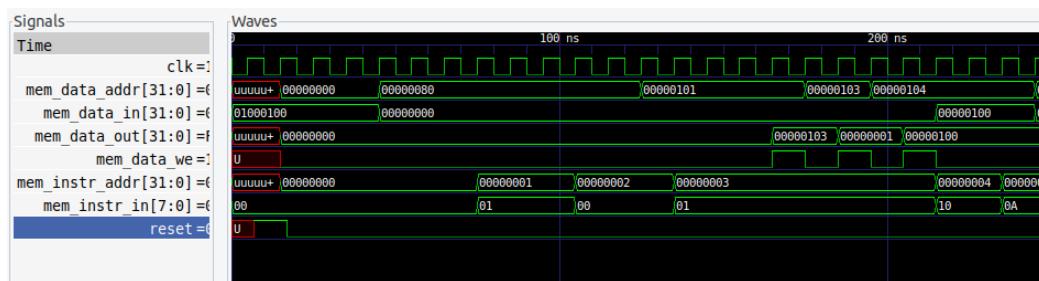
Il codice operativo che abbiamo scelto da modificare è IADD, in modo da creare una INOT. La sequenza di microistruzioni è riportata di seguito.

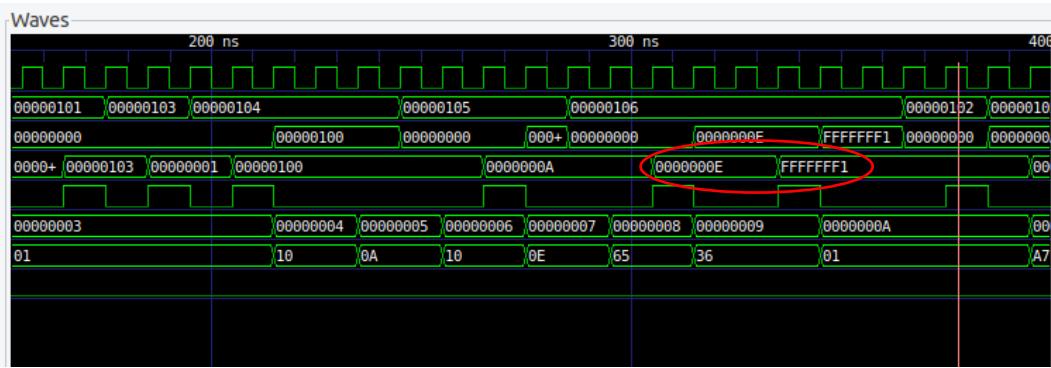
```
iadd = 0x65:  
    MAR = SP; rd  
    MDR = TOS = NOT MDR; wr; goto main
```

Il programma non è stato modificato:

```
.main  
.var  
a  
.endvar  
BIPUSH 0xA  
BIPUSH 0xE  
IADD  
ISTORE a  
HALT  
.endmethod
```

Nella simulazione vedremo il risultato della *NOT* del valore *0xE* che è *0x1*:





Abbiamo inoltre modificato il file *processor_tb.vhd* in modo da verificare correttamente il risultato dell'istruzione:

```
wait until mem_instr_addr = x"0000000A" and mem_data_we = '1';
assert mem_data_out = x"FFFFFFF1" report "Bad calculated value" severity failure;
```

Esercizio 10: Interfaccia seriale

Partendo dall'implementazione fornita dalla Digilent di un dispositivo UART-RS232 (componente RS232RefComp.vhd), progettare, implementare e simulare in VHDL un sistema composto da 2 unità A e B che condividono lo stesso segnale di clock e comunicano tra loro mediante interfaccia seriale. Il sistema A contiene una ROM di 8 locazioni da 1 byte ciascuno, un contatore CONT_A per scandire le locazioni della ROM e una UART_A, mentre il sistema B contiene una memoria MEM di 8 locazioni da 1 byte ciascuno, un contatore CONT_B per scandire le locazioni della MEM e una UART_B. Quando un segnale WR viene asserito nell'unità A, viene prelevato un byte dalla ROM e inviato all'unità B, che dovrà riceverlo e salvarlo in MEM.

Progetto e architettura

La UART (Universal Asynchronous Receiver Transmitter) è un dispositivo hardware che supporta la comunicazione seriale asincrona. L'interfaccia UART consiste di 2 segnali TX e RX su cui viaggiano i dati trasmessi/ricevuti, più un segnale di GROUND, e prevede che la comunicazione avvenga secondo uno specifico protocollo che prevede una determinata struttura del frame trasmesso. Nel caso dell'UART, il pacchetto è composto da un bit di START, un byte di dati e un bit di STOP. Per garantire la corretta sincronizzazione, è comune che il ricevitore sovracampioni la linea, operando a una frequenza 8-16 volte superiore rispetto a quella del trasmettitore.

Nel dettaglio, il ricevitore, al rilevamento del bit di START, inizia a campionare i primi 8 campioni, posizionandosi al centro byte, e successivamente campiona ogni 16, mantenendo la sincronizzazione con il flusso di dati. Il trasmettitore, d'altra parte, è più semplice e inizia la trasmissione sotto il controllo dell'unità di controllo. Al termine della comunicazione, alza il flag TBE per indicare che il bus è stato svuotato.

Esistono tre modalità di trasmissione:

- **Simplex:** Comunicazione unidirezionale, in cui solo il trasmettitore può comunicare con il ricevitore.
- **Half-Duplex:** Trasmettitore e ricevitore possono comunicare sullo stesso canale, ma solo uno alla volta.
- **Full-Duplex:** Comunicazione bidirezionale permettendo la trasmissione simultanea in entrambe le direzioni.

L'architettura del sistema è strutturata sulla entity *sistema*, che incorpora le due unità e definisce i segnali di start (*start_tot*), reset (*rst_tot*), e clock (*clk_tot*) comuni.

L'**unità A** (*Unita_a*) è costituita principalmente da una componente *Rs232RefComp*, che implementa un'interfaccia UART RS-232. In aggiunta, è presente un modulo *rom_seq* come memoria di sola lettura

sequenziale e un contatore modulo 8 (contatore_mod8). Quest'ultimo è finalizzato a scandire le locazioni della ROM.

Per comprendere al meglio la transizione di stati all'interno dell'unità A abbiamo definito l'automa a stati finiti:

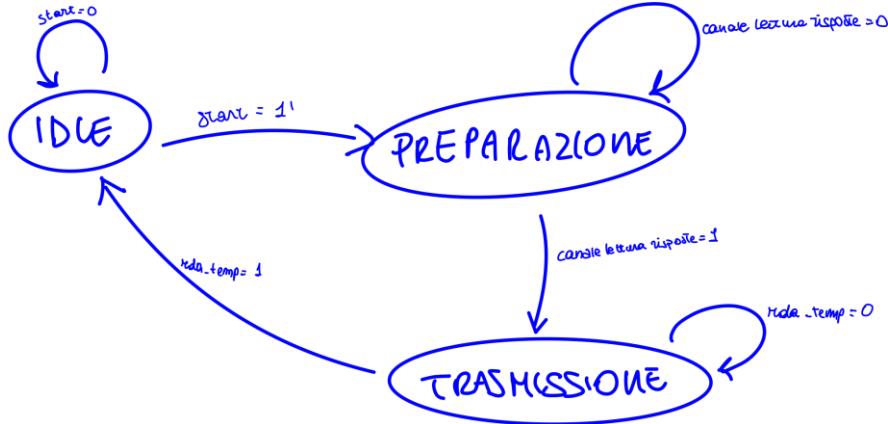


Figura 116 automa a stati finiti unità A

Gli stati evidenziati sono i seguenti:

- **Stato iniziale (IDLE):** L'unità A è IDLE fino a quando non viene richiesta la trasmissione.
- **Preparazione dei Dati:** Quando richiesto, l'unità deve preparare i dati da trasmettere leggendoli da una ROM.
- **Trasmissione dei Dati:** Dopo la preparazione, i dati vengono trasmessi seguendo il protocollo UART.

L'**unità B** (Unita_b) segue una struttura analoga, in questo caso la FSM è diversa, comprendendo gli stati IDLE, RICEZIONE e SCRITTURA. Qui, l'unità B riceve i dati dall'interfaccia UART, li memorizza nella memoria e successivamente li legge.

Anche in questo caso abbiamo definito l'automa a stati finiti:

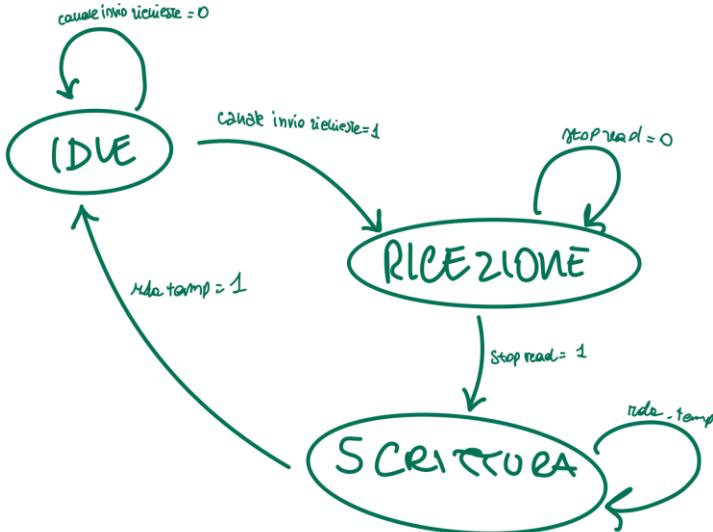


Figura 117 automa a stati finiti unità b

- **Stato Iniziale (IDLE):** all'avvio del sistema, l'unità B si trova nello stato IDLE, in attesa di richieste di trasmissione dall'unità A. La transizione dallo stato IDLE allo stato RICEZIONE avviene quando l'unità B rileva la richiesta di trasmissione (RTS) proveniente dall'unità A.

- **RICEZIONE:** durante lo stato RICEZIONE, l'unità B attiva il canale di lettura risposte (CTS) per indicare all'unità A che è pronta a ricevere. L'unità B rimane in stato RICEZIONE finché l'unità A non comunica la fine della trasmissione attraverso il segnale stop_read.
- **SCRITTURA:** dopo la ricezione completa dei dati, l'unità B passa allo stato SCRITTURA, pronta per scrivere i dati ricevuti in memoria. Durante lo stato SCRITTURA, l'unità B controlla la corretta scrittura dei dati in memoria. Quando la scrittura in memoria è completata, l'unità B ritorna allo stato IDLE, pronta per ricevere nuove richieste di trasmissione.

Entrambe le unità condividono il segnale txd_rxd per la trasmissione/ricezione seriale e i segnali di controllo CTS e RTS per la gestione del flusso di dati tra di loro. La comunicazione tra le unità è basata sulla gestione degli stati e dei segnali di controllo, garantendo la sincronizzazione e il corretto funzionamento della trasmissione/ricezione dati.

Per progettare il sistema abbiamo utilizzato un approccio modulare, come detto in precedenza i moduli principali in cui dividiamo il sistema sono unità A e unità B.

A loro volta utilizziamo un approccio modulare per progettare le due unità.

L'unità A è composta da rom sequenziale, contatore e UART. L'unità B è composta da rom sequenziale, contatore, UART e memoria.

Lo schema ad alto livello è il seguente:

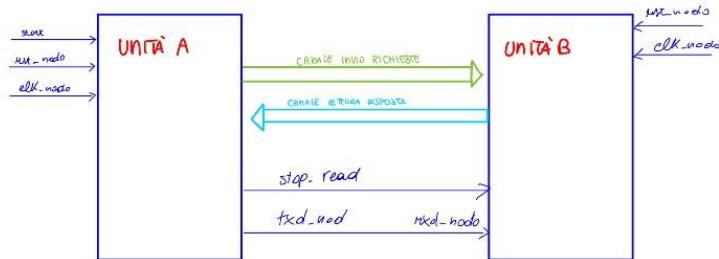


Figura 118: schema UART

Implementazione

Iniziamo dall'unità A che presenta la seguente entity:

```

entity Unita_a is
  Port (
    start : in STD_LOGIC;
    rst_nodo : in STD_LOGIC;
    clk_nodo : in STD_LOGIC;
    txd_nodo : out STD_LOGIC;
    stop_read : out std_logic := '0';
    canale_invio_richieste : out STD_LOGIC := '0';
    canale_lettura_risposte : in STD_LOGIC := '0'
  );
end Unita_a;

```

Figura 119 entity unita A

Definiamo i seguenti port:

- **start:** indica l'inizio delle operazioni. Quando start è attivo ('1'), l'unità A inizia un nuovo processo di comunicazione.

- **rst_nodo**
- **clk_nodo**
- **txd_nodo**: è il segnale di uscita attraverso il quale l'unità A trasmette dati. È il canale di trasmissione che invia i dati in uscita dall'unità A attraverso la periferica UART.
- **stop_read**: indica quando l'unità B ha terminato la lettura.
- **canale_invio_richieste**: rappresenta il canale tramite il quale l'unità A invia il segnale RTS. Quando canale_invio_richieste è attivo, l'unità A sta chiedendo di trasmettere dati.
- **canale_lettura_risposte**: rappresenta la ricezione del segnale CTS. Successivamente definiamo l'architettura utilizzando un approccio strutturale.

La prima cosa che facciamo è definire dei segnali che utilizzeremo per collegare i vari componenti:

```
architecture structural of UnitA is
  signal tbe_temp : std_logic := '1';
  signal wr_temp : std_logic := '0';
  signal bus_input : std_logic_vector(7 downto 0);
  signal bus_output : std_logic_vector(7 downto 0);

  signal rda_temp : std_logic := '0';
  signal parity : std_logic := '0';
  signal frame : std_logic := '0';
  signal overwrite : std_logic := '0';
  signal temp4 : std_logic_vector(7 downto 0);
  signal temp5 : std_logic := '0';

  signal o_contatore : std_logic_vector(2 downto 0);
  signal e_contatore : std_logic := '0';
  signal e_read_rom : std_logic := '0';
  signal rom_data_out : std_logic_vector(7 downto 0);

  type stato is (IDLE, PREPARAZIONE, TRASMISSIONE);
  signal state : stato := IDLE;
  signal next_state : stato;
```

Figura 120 segnali utilizzati nell'architettura

- **tbe_temp** e **wr_temp**: Questi segnali sono segnali di collegamento per la UART. Il primo per il segnale TBE e il secondo per il WR strobe, per far abbassare TBE.
- **bus_input** e **bus_output**: queste linee di bus sono rispettivamente ingresso e uscita della UART. bus_input memorizza i dati letti dalla ROM, mentre bus_output memorizza i dati pronti per essere trasmessi all'unità B.
- **rda_temp**: questo segnale rappresenta il flag "Read Data Available". Quando è attivo, indica che ci sono dati disponibili per la lettura. È essenziale per sincronizzare il processo di trasmissione, garantendo che l'unità A sia pronta a trasmettere solo quando ci sono dati validi disponibili.
- **parity, frame, e overwrite**: questi segnali sono utilizzati per segnalare eventuali errori durante la trasmissione seriale. parity indica un errore di parità, frame indica un errore di frame, e overwrite indica un errore di sovrascrittura. Forniscono un meccanismo di feedback per monitorare la qualità della comunicazione.
- **temp5**: segnale di collegamento per l'eventuale ricezione.
- **o_contatore** e **e_contatore**: per la gestione del contatore.
- **e_read_rom** e **rom_data_out**: questi segnali sono coinvolti nella lettura dalla memoria ROM. e_read_rom abilita il processo di lettura, e rom_data_out contiene i dati letti dalla ROM.

Successivamente andiamo a definire i componenti utilizzati:

```

component Rs232RefComp is
  Port (
    TXD : out std_logic := '1';
    RXD : in std_logic;
    CLK : in std_logic; --Master Clock
    DBIN : in std_logic_vector (7 downto 0);--Data Bus in
    DBOUT : out std_logic_vector (7 downto 0);--Data Bus out
    RDA : inout std_logic; --Read Data Available(1 quando il dato è disponibile nel registro rdReg)
    TBE : inout std_logic := '1'; --Transfer Bus Empty(1 quando il dato da inviare è stato caricato nello shift register)
    RD : in std_logic; --Read Strobe(se 1 significa "leggi" --> fa abbassare RDA)
    WR : in std_logic; --Write Strobe(se 1 significa "scrivi" --> fa abbassare TBE)
    PE : out std_logic; --Parity Error Flag
    FE : out std_logic; --Frame Error Flag
    OE : out std_logic; --Overwrite Error Flag
    RST : in std_logic := '0'; --Master Reset
  end component;

component rom_seq is
  generic(
    word_lenght : positive := 8;
    depth : positive := 3;
    numero_locazioni : positive := 2**3
  );
  port(
    clk : in std_logic; --segnaile di clock
    address : in std_logic_vector(depth-1 downto 0);
    read: in std_logic; --segnaile di lettura della stringa presente all'indirizzo di memoria
    data : out std_logic_vector (word_lenght-1 downto 0)
  );
end component;

component contatore_mod8
  generic(
    n : positive := 3;
    max : positive := 2**3-1
  );
  port(
    en : in std_logic;
    clk: in std_logic;
    rst : in std_logic;
    q : out std_logic_vector(n-1 downto 0);
    co : out std_logic
  );
end component;

```

Non ci soffermeremo sulle implementazioni dei singoli componenti in quanto già sono state spiegate nell'appendice e la UART ci è stata fornita.

Dopo aver definito i componenti li istanziamo effettuando i vari collegamenti usando i segnali precedentemente dichiarati:

```

begin
  UART_A: Rs232RefComp
  port map
  (
    TXD => txd_nodo,
    RXD => temp5,
    CLK => clk_nodo,--Master Clock
    DBIN => bus_input,--Data Bus in
    DBOUT => bus_output,--Data Bus out
    RDA => rda_temp,--Read Data Available(1 quando il dato è disponibile nel registro rdReg)
    TBE => tbe_temp,--Transfer Bus Empty(1 quando il dato da inviare è stato caricato nello shift register)
    RD => '0',--Read Strobe(se 1 significa "leggi" --> fa abbassare RDA)
    WR => wr_temp,--Write Strobe(se 1 significa "scrivi" --> fa abbassare TBE)
    PE => parity,--Parity Error Flag
    FE => frame,--Frame Error Flag
    OE => overwrite,--Overwrite Error Flag
    RST => rst_nodo
  );

  CONT_B: contatore_mod8
  Port map(
    clk    => clk_nodo,
    rst   => rst_nodo,
    en    => e_contatore,
    q     => o_contatore
  );

  ROM0: rom_seq
  Port map(
    clk    => clk_nodo,
    read  => '1',
    address => o_contatore,
    data  => rom_data_out
  );

```

Successivamente andiamo a definire la macchina a stati:

```

fsm: process(clk_nodo)
begin
    if (rst_nodo = '1') then
        next_state <= IDLE;
    else

        case state is

            when IDLE =>
                wr_temp <= '0';
                if (start = '1') then
                    next_state <= PREPARAZIONE;
                    e_contatore <= '1';
                    e_read_rom <= '1';
                else
                    next_state <= IDLE;
                end if;

            when PREPARAZIONE =>
                if e_read_rom = '1' then
                    bus_input <= rom_data_out;
                    e_contatore <= '0';
                    e_read_rom <= '0';
                end if;

                canale_invio_richieste <= '1'; --RTS

                if (canale lettura risposte = '1') then --SE ARRIVA CTS
                    canale_invio_richieste <= '0';--OK RTS
                    next_state <= TRASMISSIONE;
                    wr_temp <= '1';
                else
                    next_state <= PREPARAZIONE;
                end if;

            when TRASMISSIONE =>
                if wr_temp <= '1' then
                    wr_temp <= '0';
                end if;
                if (rda_temp = '1') then
                    stop_read <= '1';
                    next_state <= IDLE;
                else
                    next_state <= TRASMISSIONE;
                end if;

                when others =>
                    next_state <= IDLE;

        end case;

    end if;
end process;

```

Tramite questo processo implementiamo la macchina definita dall'automa a stati finiti precedentemente esposta.

Vediamo come implementiamo le transizioni di stato:

- **IDLE:** In questo stato, il sistema è in attesa di iniziare una nuova trasmissione. La variabile wr_temp è impostata a '0'.
- **PREPARAZIONE:** quando il segnale di start (start = '1') viene rilevato, la FSM passa allo stato di PREPARAZIONE. Viene attivato il segnale e_contatore, che abilita il contatore utilizzato per accedere alle locazioni della ROM. Il segnale e_read_rom viene attivato, permettendo la lettura dei dati dalla ROM. Il dato letto viene caricato nel segnale bus_input. Viene attivato il canale di invio richieste (canale_invio_richieste = '1' per l'invio di RTS).
- **TRASMISSIONE:** se il canale di lettura risposte (canale lettura risposte = '1' ricezione di CTS) è attivo, la FSM passa allo stato di TRASMISSIONE. Viene attivato il segnale wr_temp a '1', indicando l'inizio della trasmissione. La FSM rimane in questo stato finché il dato non è stato completamente trasmesso (rda_temp = '1'). Quando la trasmissione è completata, la FSM passa nuovamente allo stato IDLE.

Successivamente definiamo un nuovo processo in cui il segnale stato viene aggiornato sulla base del fronte di salita del segnale di clock.

```

nuovo_stato: process (clk_nodo)
begin
    if (rising_edge(clk_nodo) and clk_nodo = '1') then
        state <= next_state;
    end if;
end process;

```

L'implementazione dell'unità B che risulta molto simile. Ne riportiamo la FSM:

```

fsm : process(clk_nodo)
begin
    if (rst_nodo = '1') then
        next_state <= IDLE;
    else

        case state is

            when IDLE =>
                e_contatore <= '0';
                e_write_mem <= '0';
                if (canale_invio_richieste = '1') then
                    canale_lettura_risposte <= '1'; --CTS
                    next_state <= RICEZIONE;
                    rd_temp <= '1';
                else
                    rd_temp <= '0';
                    next_state <= IDLE;
                end if;

            when RICEZIONE =>
                canale_lettura_risposte <= '0'; -- OK CTS
                if (stop_read = '1') then
                    next_state <= SCRITTURA;
                else
                    next_state <= RICEZIONE;
                end if;

            when SCRITTURA =>
                rd_temp <= '0';
                bus_temp <= bus_output;
                if (rda_temp = '1') then
                    e_write_mem <= '1';
                    e_contatore <= '1';
                    next_state <= IDLE;
                else
                    next_state <= SCRITTURA;
                end if;
            when others =>
                next_state <= IDLE;
        end case;

    end if;
end process;

```

Figura 121 processo fms unità B

- **IDLE:** in questo stato, il segnale e_contatore è abbassato a '0', indicando che il contatore non è abilitato. Il segnale e_write_mem è a '0', indicando che la scrittura in memoria non è abilitata. Se il canale di invio richieste (canale_invio_richieste) è attivo, viene attivato il canale di lettura risposte (canale_lettura_risposte), indicando che l'unità B è pronta a ricevere dati.
- **RICEZIONE:** viene disattivato il canale di lettura risposte (canale_lettura_risposte) per l'OK. Se il segnale stop_read è attivo, l'automa passa allo stato di SCRITTURA perché ha ricevuto il frame. In caso contrario, rimane nello stato RICEZIONE.
- **SCRITTURA:** il segnale rd_temp è abbassato a '0'. La variabile bus_temp è caricata con i dati presenti sul bus di output (bus_output). Se il segnale rda_temp è attivo, l'automa abilita la scrittura in memoria (e_write_mem a '1') e il contatore (e_contatore a '1'), quindi ritorna allo stato IDLE. In caso contrario, rimane nello stato SCRITTURA.

Infine il sistema complessivo realizza il collegamento tra le due unità. L'entity è la seguente:

```

entity sistema is
  Port (
    start_tot : in STD_LOGIC;
    rst_tot : in STD_LOGIC;
    clk_tot : in STD_LOGIC
  );
end sistema;

```

Figura 122 entity sistema

Questa entità rappresenta i segnali di controllo fondamentali che possono essere utilizzati per gestire l'avvio, il reset e la temporizzazione delle operazioni nel sistema complessivo.

Anche in questo caso utilizziamo un approccio strutturale per descrivere l'architettura. Come prima cosa dichiariamo i segnali utilizzati per effettuare le connessioni tra i due componenti del sistema (unità A e unità B):

```

architecture structural of sistema is
  signal txd_rxd : std_logic;
  signal CTS : std_logic;
  signal RTS : std_logic;
  signal stop : std_logic := '0';

```

Figura 123 segnali per le connessioni

```

begin
  nodo_a: Unita_a
  port map
  (
    start => start_tot,
    rst_nodo => rst_tot,
    clk_nodo => clk_tot,
    txd_nodo => txd_rxd,
    stop_read => stop,
    canale_invio_richieste => RTS,
    canale lettura_risposte => CTS
  );

  nodo_b: Unita_b
  port map
  (
    rst_nodo => rst_tot,
    clk_nodo => clk_tot,
    rxd_nodo => txd_rxd,
    stop_read => stop,
    canale_invio_richieste => RTS,
    canale lettura_risposte => CTS
  );

```

Figura 124 istanze dei componenti

Simulazione

```
signal CLK, RST: std_logic := '0';
signal start : std_logic := '0';
signal end_sim: boolean := true;

constant PERIOD: time := 10 ns;
```

Figura 125 segnali per simulazione

Definiamo un processo di simulazione per generare sequenze di segnali di reset e start per la simulazione:

```
stimulus: process
begin
    RST <= '1';
    start <= '0';
    wait for 100 ns;
    RST <= '0';
    start <= '1';
    wait for 10 ns;
    start <= '0';
    wait for 20000 ns;
    start <= '0';
    wait for 100 ns;
    start <= '1';
    wait for 10 ns;
    start <= '0';
    wait for 20000 ns;
    start <= '0';
    wait for 100 ns;
    start <= '1';
    wait for 10 ns;
    start <= '0';
    wait for 20000 ns;
    start <= '0';
    wait for 100 ns;
    start <= '1';
    wait for 10 ns;
    start <= '0';
    wait for 20000 ns;
    start <= '0';
    wait for 100 ns;
    start <= '1';
    wait for 10 ns;
    start <= '0';
    wait for 20000 ns;
    start <= '0';
    wait for 100 ns;
    start <= '1';
    wait for 10 ns;
    start <= '0';
    wait for 20000 ns;
    start <= '0';
    wait for 100 ns;
    start <= '1';
    wait for 10 ns;
    start <= '0';
    wait for 20000 ns;
    start <= '0';
    wait for 100 ns;
    start <= '1';
    wait for 10 ns;
    start <= '0';
    wait for 20000 ns;
    start <= '0';
    wait for 100 ns;
    start <= '1';
    wait for 10 ns;
    start <= '0';
    wait for 20000 ns;
    start <= '0';
    wait for 100 ns;
    start <= '1';
    wait for 10 ns;
    start <= '0';
    wait for 20000 ns;
    start <= '0';
    wait for 100 ns;
    start <= '1';
    wait for 10 ns;
    start <= '0';
    wait for 20000 ns;
    start <= '0';
    wait for 100 ns;
    start <= '1';
    wait for 10 ns;
    start <= '0';
    wait for 20000 ns;
    start <= '0';
    wait for 100 ns;
    start <= '1';
    wait for 10 ns;
    start <= '0';
    wait for 20000 ns;
    start <= '0';
    wait for 100 ns;
    start <= '1';
    wait for 10 ns;
    start <= '0';
    wait for 20000 ns;
```

```
end_sim <= false;
wait;
end process;
```

La simulazione viene eseguita fino a quando end_sim diventa false.

Avviando la simulazione otteniamo il seguente risultato: riportiamo la trasmissione solo della prima locazione della rom nell'unità A in quanto il processo complessivamente dura molto.

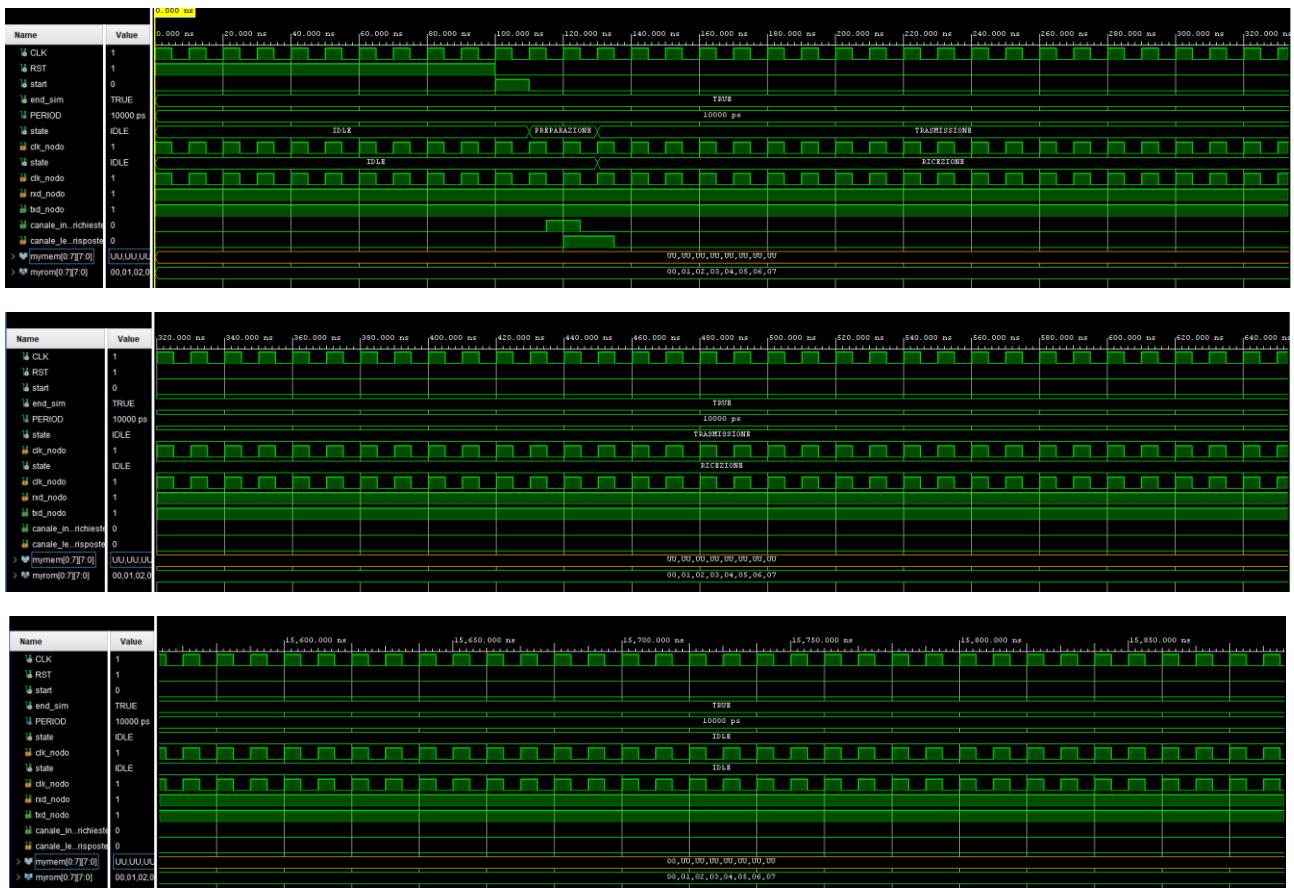


Figura 126 simulazione UART

È possibile comunque visionare il contenuto della memoria del nodo B dopo circa 160 microsecondi:

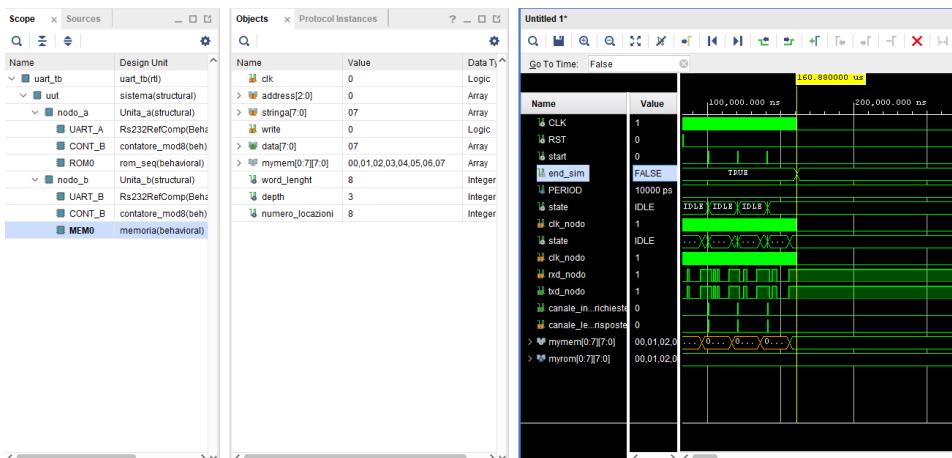


Figura 127: Contenuto MEM

Come si vede dalla figura 132 il comportamento delle FSM è quello atteso.

Esercizio 11: Switch Multistadio

Progettare ed implementare in VHDL uno switch multistadio secondo il modello omega network. Lo switch deve consentire lo scambio di messaggi di 2 bit ciascuno da un nodo sorgente a un nodo destinazione in una rete con 4 nodi, implementando uno schema a priorità fissa fra i nodi (es. nodo 1 più prioritario, con priorità decrescenti fino al nodo 4).

Progetto e architettura

All'interno di un sistema di elaborazione distribuito, in cui molteplici dispositivi devono comunicare tra loro scambiandosi messaggi, è fondamentale utilizzare un'infrastruttura di interconnessione efficiente, scalabile e modulare, che mantenga cioè una latenza di comunicazione limitata all'aumentare dei nodi interconnessi e che sia facilmente integrabile in sistemi più grandi.

Lo switch multistadio è un dispositivo composto da più livelli intermedi che consente la comunicazione simultanea tra più coppie di nodi sorgente e destinazione. A differenza dello switch a singolo stadio risulta più scalabile.

L'architettura multistadio si basa sull'utilizzo di log N stadi, ognuno dei quali composto da elementi semplici. Questi elementi formano il blocco di base, costituito da un multiplexer a due ingressi, abilitato dal bit di sorgente, e un demultiplexer abilitato dai bit della destinazione. L'indirizzo della sorgente e della destinazione è diviso in k bit, distribuiti su tante parti quanti sono gli stadi. Questa suddivisione consente di controllare opportunamente i blocchi a ogni livello dell'interconnessione.

Un modello topologico per questo tipo di interconnessione è definito come Omega Network, che si basa sull'algoritmo "perfect shuffling". In questo modo esiste uno e un solo percorso fra una determinata coppia di nodi. Tuttavia, questa architettura non risolve il problema delle collisioni che si verificano quando più nodi tentano di comunicare con lo stesso nodo di destinazione contemporaneamente o quando devono attraversare uno stesso stadio allo stesso tempo. Questi fenomeni richiedono una gestione adeguata per evitare conflitti e garantire un funzionamento corretto.

Per progettare l'architettura abbiamo utilizzato un approccio modulare. All'interno del nostro sistema distinguiamo l'**unità di controllo** e l'**unità operativa**.

L'**unità operativa** è progettata per indirizzare e instradare i dati tra i nodi di sorgente (x_0, x_1, x_2, x_3) e i nodi di destinazione (y_0, y_1, y_2, y_3). L'architettura fa uso di un componente, lo **switch**, il quale è responsabile dell'instradamento dei dati tra i vari nodi. Effettivamente realizza la topologia descritta prima.

Il componente switch è utilizzato per instradare i dati da un nodo di sorgente a un nodo di destinazione in base ai segnali di selezione (sel_sorgente e sel_destinazione). Ciascuno degli switch (s_0, s_1, s_2, s_3) è collegato tra due nodi specifici, gestendo così una parte della rete di interconnessione.

Gli switch vengono collegati in maniera strutturale per guidare i dati attraverso la rete multistadio.

Lo schema dell'unità operativa è il seguente:

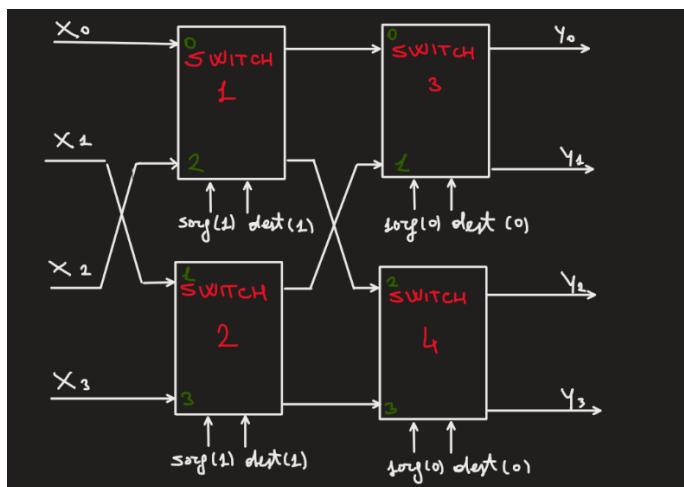


Figura 128 architettura unità operativa

Per realizzare lo switch abbiamo utilizzato i seguenti componenti:

- **mux_2_1**: Questo componente implementa la funzione di multiplexing a 2 ingressi. In base al segnale di selezione sel_sorgente, seleziona uno degli ingressi (x_0 o x_1) e lo instrada all'uscita link.
- **demux_1_2**: Al contrario, questo componente svolge la funzione di demultiplexing. Prende il segnale link e lo instrada dinamicamente a uno degli output (y_0 o y_1) in base al segnale di selezione sel_destinazione

Lo schema dello switch è il seguente:

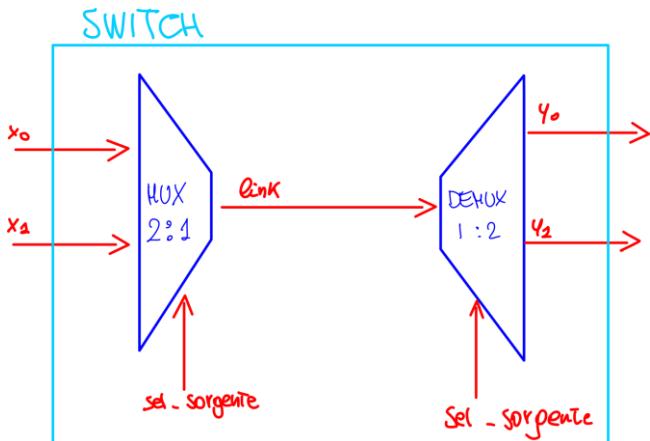


Figura 129 schema switch

L'**unità_controllo** implementa lo schema di comunicazione a priorità fissa. Il **gestore** determina la priorità tra i nodi sorgente abilitati (en_0, en_1, en_2, en_3) e produce il segnale sel per selezionare l'input appropriato da instradare. In base alle richieste dei nodi, il gestore determina la priorità e imposta il segnale sel di conseguenza in modo da inviare in input all'unità operativa il payload del pacchetto da instradare e gli indirizzi di sorgente e destinazione.

Lo schema dell'unità di controllo è il seguente:

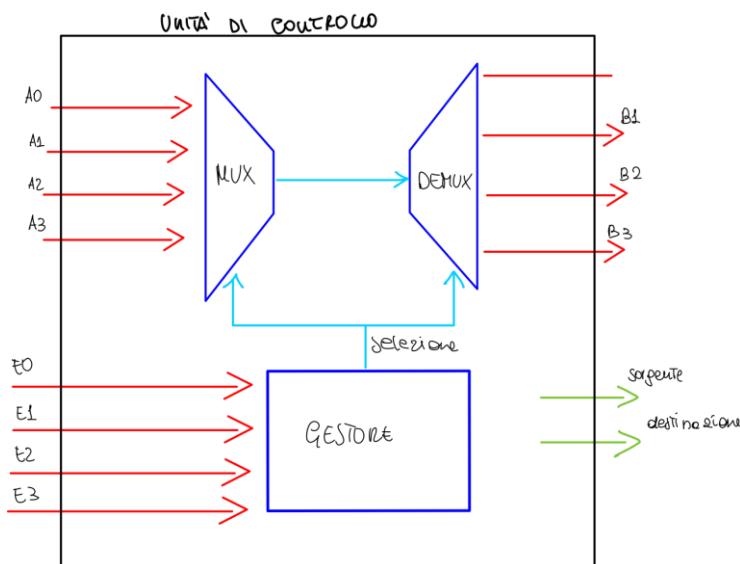


Figura 130 schema unità di controllo

Lo schematico dell'architettura è il seguente:

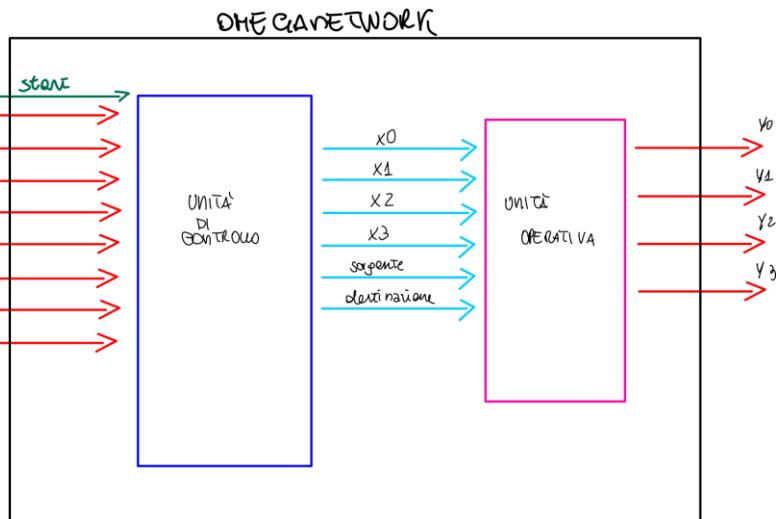


Figura 131 architettura omega network

Implementazione

Come precedentemente spiegato, per l'implementazione abbiamo seguito un approccio modulare. Abbiamo implementato separatamente i vari componenti che vanno a costituire unità di controllo e unità operativa.

L'unità operativa ricordiamo essere composta dallo switch. A sua volta, lo switch è composto da un multiplexer 2:1 e da un demultiplexer 1:2. Per l'implementazione di questi due componenti si rimanda all'appendice.

Iniziamo ad analizzare l'implementazione dello switch.

Definiamo la seguente entità:

```
entity switch is
  generic(n : positive := 2 );
  Port (
    x0, x1 : in std_logic_vector(n-1 downto 0);
    y0, y1 : out std_logic_vector(n-1 downto 0);
    sel_sorgente, sel_destinazione : in std_logic
  );
end switch;
```

Figura 132 entità switch

All'interno dell'entity dichiariamo un parametro generic **n** dichiarato con valore predefinito 2. Questo parametro rappresenta la dimensione del payload. I segnali di selezione (**sel_***) sono utilizzati per instradare correttamente i dati in base agli indirizzi di sorgente e destinazione.

Figura 133 definizione strutturale dell'architettura

Definiamo anche un segnale link di tipo `std_logic_vector` di 4 bit che viene utilizzato per effettuare il collegamento tra l'uscita del multiplexer e l'ingresso del demultiplexer.

Procediamo con l'istanziare i componenti.

```

begin
    MUX : mux_2_1
    port map(
        a0 => x0,
        a1 => x1,
        s => sel_sorgente,
        y => link
    );
    DEMUX : demux_1_2
    port map(
        a => link,
        y0 => y0,
        y1 => y1,
        s => sel_destinazione
    );
end structural;

```

Figura 134 port map componenti

In questa sezione di codice, quindi, si definisce come il multiplexer riceve i dati dai due ingressi `x0` e `x1` e li instrada in base al segnale di selezione `sel_sorgente` verso il segnale `link`. Successivamente, il demultiplexer riceve i dati dal segnale `link` e li distribuisce alle uscite `y0` e `y1` in base al segnale di selezione `sel_destinazione`.

Dopo aver analizzato l'implementazione dello switch possiamo passare all'implementazione **dell'unità operativa**. L'entity è la seguente:

```

entity unita_operativa is
    generic(n : positive := 2);
    Port (
        x0, x1, x2, x3 : in std_logic_vector(n-1 downto 0);
        y0, y1, y2, y3 : out std_logic_vector(n-1 downto 0);
        dest : in std_logic_vector(1 downto 0);
        sorg : in std_logic_vector(1 downto 0)
    );
end unita_operativa;

```

Figura 135 entity unità operativa

I port dichiarati sono i seguenti:

- **x0, x1, x2, x3**: sono i segnali in ingresso di lunghezza n. Questi rappresentano i dati provenienti da quattro diverse sorgenti.
- **y0, y1, y2, y3**: sono i segnali in uscita di lunghezza n. Questi rappresentano i dati destinati a quattro diverse destinazioni.
- **dest**: specifica l'indirizzo di destinazione.
- **sorg**: specifica l'indirizzo di sorgente.

Anche in questo caso per l'architettura seguiamo un approccio strutturale:

```

architecture structural of unita_operativa is
  component switch is
    generic(n : positive := 2);
    Port (
      x0, x1 : in std_logic_vector(n-1 downto 0);
      y0, y1 : out std_logic_vector(n-1 downto 0);
      sel_sorgente, sel_destinazione : in std_logic
    );
  end component;
  signal link00, link22, link11, link33 : std_logic_vector(n-1 downto 0);
begin
  s0 : switch port map(x0, x2, link00, link22 , sorg(1), dest(1));
  s1 : switch port map(x1, x3, link11, link33 , sorg(1), dest(1));
  s2 : switch port map(link00,link11 ,y0, y1 , sorg(0), dest(0));
  s3 : switch port map(link22,link33 ,y2, y3, sorg(0), dest(0));
end structural;

```

Figura 136 definizione architettura

All'interno dell'architettura dichiariamo il componente dello switch che andiamo ad utilizzare.

Dichiariamo 4 segnali che rappresentano il collegamento tra i due livelli della rete. Successivamente andiamo ad effettuare il port mapping.

s0, s1, s2, s3: rappresentano le istanze del componente switch:

- s0 e s1 rappresentano il primo livello e quindi in ingresso per la selezione prendono il bit più significativo degli indirizzi.
- s2 e s3 rappresentano il secondo livello e quindi in ingresso per la selezione prendono il bit meno significativo degli indirizzi.

Dopo aver analizzato l'unità operativa passiamo all'unità di controllo, anch'essa implementata seguendo un approccio modulare. Anche in questo caso analizziamo prima le implementazioni dei vari componenti.

Le componenti dell'unità di controllo sono: multiplexer 4:1, demultiplexer 1:4 e gestore.

Anche in questo caso analizziamo solo l'implementazione di **gestore**, per l'implementazione di multiplexer e demultiplexer rimandiamo all'appendice.

L'entity è la seguente:

```

entity gestore is
  Port (
    en0, en1, en2, en3 : in std_logic;
    y : out std_logic_vector(1 downto 0)
  );
end gestore;

```

Figura 137 entity gestore

Abbiamo dichiarato i seguenti port:

- **en0, en1, en2, en3**: Sono segnali di input rappresentanti le richieste dei quattro nodi sorgente per trasmettere.
- **y**: segnale di selezione dei pacchetti per mux e demux.

Andiamo a definire l'architettura del gestore utilizzando un approccio dataflow.

```

architecture dataflow of gestore is

begin
    y <= "00" when en0 = '1' else
    "01" when en1 = '1' else
    "10" when en2 = '1' else
    "11" when en3 = '1' else
    "--";
end dataflow;

```

Figura 138 architettura gestore

La logica di funzionamento del gestore è basata sulla priorità degli input en0, en1, en2, en3. Il gestore assegna all'output y il valore che rappresenta la priorità più alta tra gli input attivi. Se più di un segnale en è attivo contemporaneamente, il gestore selezionerà il primo incontrato in ordine (en0, en1, en2, en3).

Un esempio di funzionamento è il seguente:

- Se en2 è l'unico attivo, l'output y sarà "10".
- Se en1 e en3 sono attivi, l'output y sarà "01" (in quanto en1 ha priorità più alta).

Adesso che abbiamo analizzato l'implementazione del gestore passiamo all'implementazione dell'unità di controllo della quale definiamo la seguente entity:

```

entity unita_controllo is
generic(n : positive := 2);
Port (
    in0, in1, in2, in3 : in std_logic_vector(4+(n-1) downto 0);
    en0, en1, en2, en3 : in std_logic;
    data0, data1, data2, data3 : out std_logic_vector(n-1 downto 0);
    sorgente, destinazione : out std_logic_vector(1 downto 0)
);
end unita_controllo;

```

Figura 139 entity

Anche in questo caso dichiariamo un parametro generico n per rappresentare la dimensione dei dati.

I port dichiarati sono i seguenti:

- in0, in1, in2, in3: Rappresentano i pacchetti provenienti da differenti nodi della rete. Hanno dimensione 6 bit nel nostro caso in quanto abbiamo 4 bit per gli indirizzi di sorgente e destinazione e 2 bit di payload.
- en0, en1, en2, en3: Richieste dei nodi per trasmettere.
- data0, data1, data2, data3: Rappresentano i payload a valle dell'elaborazione del gestore.
- sorgente
- destinazione

Per la definizione dell'architettura utilizziamo un approccio strutturale:

```

begin
    PRIO : gestore
    port map(
        en0 => en0,
        en1 => en1,
        en2 => en2,
        en3 => en3,
        y => sel
    );
    MUX : mux_4_1
    port map(
        b0 => in0(1 downto 0),
        b1 => in1(1 downto 0),
        b2 => in2(1 downto 0),
        b3 => in3(1 downto 0),
        y => link,
        s0 => sel(0),
        s1 => sel(1)
    );
    DEMUX : demux_1_4
    port map(
        y0 => data0,
        y1 => data1,
        y2 => data2,
        y3 => data3,
        a => link,
        s => sel
    );
    sorgente <= in0(5 downto 4) when sel = "00" else
        in1(5 downto 4) when sel = "01" else
        in2(5 downto 4) when sel = "10" else
        in3(5 downto 4) when sel = "11" else
        "--";
    destinazione <= in0(3 downto 2) when sel = "00" else
        in1(3 downto 2) when sel = "01" else
        in2(3 downto 2) when sel = "10" else
        in3(3 downto 2) when sel = "11" else
        "--";
end structural;

```

Figura 140 instanziamento componenti

Implementiamo la logica per la generazione di sorgente e destinazione in base al valore del vettore di selezione in uscita dal gestore.

Dopo aver visto separatamente le implementazioni dei vari moduli analizziamo l'implementazione complessiva dell'omega network.

Definiamo l'entity:

```

entity omega_network is
    generic(n : positive := 2);
    Port (
        pack0, pack1, pack2, pack3 : in std_logic_vector(4+(n-1) downto 0);
        en : in std_logic_vector(3 downto 0);
        payload0, payload1, payload2, payload3 : out std_logic_vector(n-1 downto 0)
    );
end omega_network;

```

Figura 141 entity omega network

L'entity omega_network è progettata per ricevere pacchetti da quattro nodi (pack0, pack1, pack2, pack3), abilitare specifici pacchetti in ingresso tramite il segnale di en, e produrre i rispettivi payload in uscita (payload0, payload1, payload2, payload3) dopo averli instradati attraverso la rete Omega.

Abbiamo dichiarato il generic n per la dimensione dei dati.

Per l'architettura utilizziamo un approccio strutturale componendo unità operativa e unità di controllo.

Definiamo anche i segnali per effettuare i collegamenti tra i vari componenti.

I segnali link_* vengono utilizzati per il collegamento tra le due unità. In particolare, li vediamo per le uscite del demux dell'unità di controllo con il primo livello della rete dell'unità operativa, e per il collegamento degli indirizzi di sorgente e destinazione. Infine, istanziamo i componenti:

```
begin
    OU : unita_operativa
        port map(
            x0 => link0,
            x1 => link1,
            x2 => link2,
            x3 => link3,
            y0 => payload0,
            y1 => payload1,
            y2 => payload2,
            y3 => payload3,
            sorg => link_sorgente,
            dest => link_destinazione
        );
    CU : unita_controllo
        port map(
            in0 => pack0,
            in1 => pack1,
            in2 => pack2,
            in3 => pack3,
            en0 => en(0),
            en1 => en(1),
            en2 => en(2),
            en3 => en(3),
            data0 => link0,
            data1 => link1,
            data2 => link2,
            data3 => link3,
            sorgente => link_sorgente,
            destinazione => link_destinazione
        );
    end structural;
```

Simulazione

```
signal packet0, packet1, packet2, packet3 : std_logic_vector(5 downto 0);
signal richieste : std_logic_vector(3 downto 0);
signal out0, out1, out2, out3 : std_logic_vector(1 downto 0);
```

Figura 142 segnali testbench

Dichiariamo i segnali che utilizzeremo per controllare e monitorare il comportamento del sistema durante la simulazione.

- **packet0, packet1, packet2, packet3** (Segnali di Ingresso): rappresentano i pacchetti in ingresso alla rete Omega.
- **richieste** (Segnale di Ingresso): è un segnale di input a 4 bit che indica quali delle quattro unità vogliono trasmettere in un dato momento. Ciascun bit del vettore corrisponde a una delle quattro unità. Ad esempio, se richieste è "1010", significa che solo la prima e la terza unità vogliono trasmettere.
- **out0, out1, out2, out3** (Segnali di Uscita): rappresentano i payload dei pacchetti in uscita dalla rete Omega.

Utilizzando questi segnali effettuiamo il collegamento con il componente da testare:

Dopo definiamo il processo di simulazione **sim_proc**

```

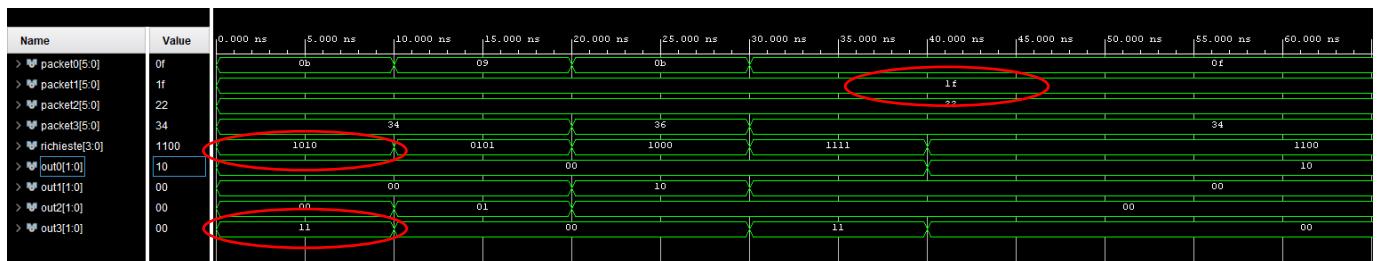
sim_proc : process begin
    richieste <= "1010"; --vogliono trasmettere 1 e 3
    packet0 <="001011";
    packet1 <="011111"; --dovrei vedere 11 sull'uscita 3
    packet2 <="100010";
    packet3 <="110100";
    wait for 10 ns;
    richieste <= "0101"; --vogliono trasmettere 0 e 2
    packet0 <="001001"; --dovrei vedere 01 sull'uscita 2
    packet1 <="011111";
    packet2 <="100010";
    packet3 <="110100";
    wait for 10 ns;
    richieste <= "1000"; --vuole trasmettere 3
    packet0 <="001011";
    packet1 <="011111";
    packet2 <="100010";
    packet3 <="110110"; --dovrei vedere 10 sull'uscita 1
    wait for 10 ns;
    richieste <= "1111"; --vogliono trasmettere tutti
    packet0 <="001111"; --dovrei vedere 11 sull'uscita 3
    packet1 <="011111";
    packet2 <="100010";
    packet3 <="110100";
    wait for 10 ns;
    richieste <= "1100"; --vogliono trasmettere 2 e 3
    packet0 <="001111";
    packet1 <="011111";
    packet2 <="100010"; --dovrei vedere 10 sull'uscita 0
    packet3 <="110100";
    wait for 10 ns;
    wait;
end process;

end rtl;

```

Alla fine di ogni ciclo, il processo attende 10 ns prima di passare al ciclo successivo. Il processo si ferma alla fine del blocco con l'istruzione wait, indicando un'attesa infinita.

Avviando la simulazione otteniamo il seguente risultato:



Vediamo come nel primo ciclo il valore di richiesta è coerente con quanto scritto nel codice ovvero è 1010 ciò significa che vogliono trasmettere 1 e 3. Essendo il nodo 1 a priorità più alta, sarà questo a trasmettere. In particolare, il nodo 1 vuole trasmettere “11” al nodo 3. Dalla figura il processo esegue correttamente ed è verificabile per tutte le prove fatte.

Appendice

Multiplexer 2:1

Progetto e architettura

Il **multiplexer 2:1** è una macchina combinatoria che presenta due ingressi e un'uscita. Nel nostro caso, abbiamo realizzato un multiplexer **indirizzabile**, ovvero, un multiplexer dotato di un ulteriore ingresso di **selezione** in base al quale uno tra i valori presenti in input sarà trasmesso in output. In definitiva, il multiplexer 2:1 presenta **3 ingressi e 1 uscita**. Lo schematico di questa macchina è mostrato nella seguente figura:

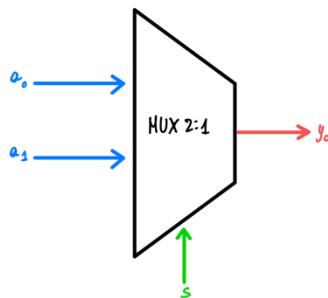


Figura 1.1 schema mux 2:1 indirizzabile

Implementazione

Il passo iniziale da compiere per l'implementazione del multiplexer 2:1 è la definizione dell'*entity* *mux_2_1*. All'interno dell'*entity* dichiariamo le porte, nello specifico 3 saranno di ingresso (in) e 1 di uscita (out). Vediamo questa parte di codice nella figura:

```
-- Definizione dell'interfaccia del modulo mux_2_1.
entity mux_2_1 is
    port(
        a0 : in STD_LOGIC;
        a1 : in STD_LOGIC;
        s : in STD_LOGIC;
        y : out STD_LOGIC
    );
end mux_2_1;
```

Figura 143.2 definizione entity mux_2_1

In questo caso abbiamo 3 ingressi (a0, a1 ed s) e un'uscita(y) tutte di tipo *std_logic*. L'oggetto è stato descritto utilizzando un'*architettura dataflow*. La descrizione dataflow in VHDL è basata sulla definizione di relazioni tra diversi segnali e la specifica di come i dati fluiscono attraverso il circuito in risposta a determinati eventi.

```
-- Utilizzo del costrutto di conditional signal assignment.
-- qui è come se stessi specificando la tabella di verità della funzione y
architecture dataflow_v1 of mux_2_1 is

begin
    y  <=  a0 when s = '0' else
           a1 when s = '1' else
           '-'; --specifica cosa succede quando s non assume valore 0 o 1
           | --- perché s è uno STD_LOGIC, non un BIT
end dataflow_v1;
```

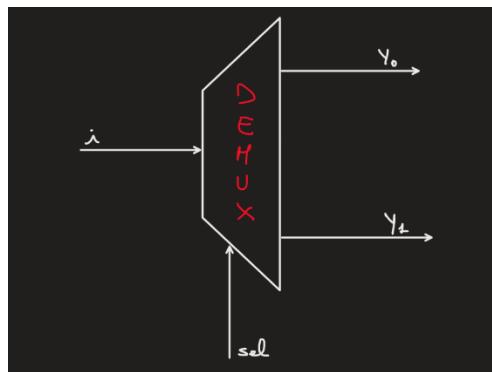
Figura 1.3 architettura dataflow mux_2_1

In questo modo abbiamo definito come il valore dell'uscita deve dipendere dall'ingresso di selezione s. L'uscita y è pari ad a0 quando l'ingresso di selezione s è pari a 0 mentre è pari ad a1 quando l'ingresso di selezione è pari a 1. Avendo definito i segnali come std_logic è necessario considerare un ulteriore caso aggiuntivo in quanto s può avere dei valori differenti da 0 ad 1, quindi è necessario definire il comportamento per qualunque altro valore di s diverso da 0 e 1, in questo caso l'uscita assumerà un valore non specificato indicato con '-'.

Demultiplexer 1:2

Progetto e architettura

Il **demultiplexer 1:2** è una macchina combinatoria che presenta un ingresso e due uscite. Nel nostro caso, abbiamo realizzato un demultiplexer **indirizzabile**, dotato di un ulteriore ingresso di *selezione* in base al quale una delle due uscite assume il valore in ingresso. Lo schematico di questa macchina è mostrato nella seguente figura:



Implementazione

Abbiamo implementato questo componente con approccio simile al multiplexer 2:1. Riportiamo qui l'implementazione:

```
entity demux_1_2 is
  generic(n : positive := 2 );
  port(
    a : in std_logic_vector(n-1 downto 0);
    s : in std_logic;
    y0, y1: out std_logic_vector(n-1 downto 0)
  );
end demux_1_2;

architecture behavioral of demux_1_2 is
begin
  process(a,s) is begin
    if (s = '0') then
      y0 <= a;
      y1 <= "00";
    elsif (s = '1') then
      y1 <= a;
      y0 <= "00";
    end if;
  end process;
end behavioral;
```

Multiplexer 4:1

Progetto e architettura

Il multiplexer 4:1 è una macchina combinatoria che presenta **4 ingressi e 1 uscita**. Anche in questo caso si tratta di un multiplexer *indirizzabile*. Infatti, sono presenti due ingressi di selezione, quindi, in totale avremo **6 ingressi e 1 uscita**.

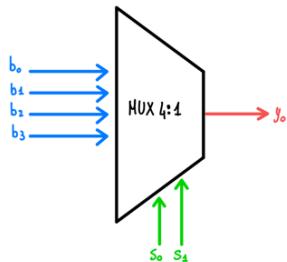


Figura 144.1 schema mux 4:1

I 4 ingressi sono b_0, b_1, b_2, b_3 mentre i 2 ingressi di selezione sono s_0 ed s_1 . L'uscita è rappresentata da y_0 .

Per ottenere un multiplexer 4:1 sono necessari **3 multiplexer 2:1**. I primi 2 multiplexer prenderanno in ingresso i 4 ingressi del multiplexer 4:1, nello specifico il primo multiplexer 2:1 prenderà in ingresso i primi

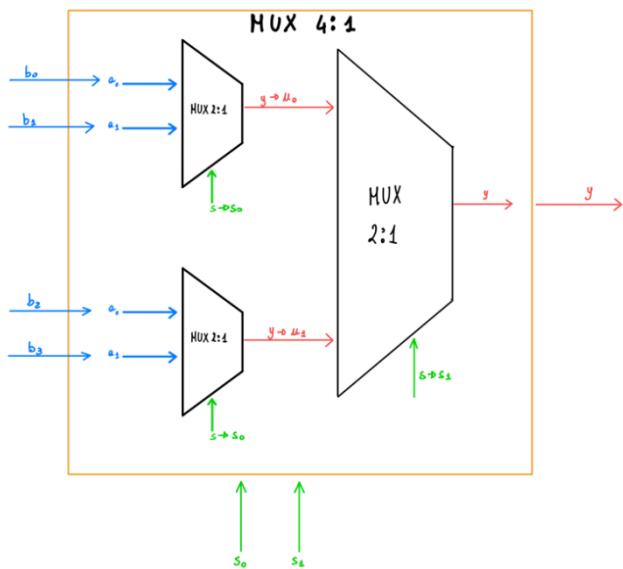


Figura 2.2 schema mux 4:1 ottenuto per composizione di mux 2:1

due dei quattro ingressi (b_0 e b_1) e il secondo prenderà i restanti due (b_2 e b_3). Ognuno di questi multiplexer 2:1 produrrà un'uscita in base a uno dei due ingressi di selezione del mux 4:1 (s_0). Le due uscite prodotte entreranno in un terzo multiplexer 2:1 che produrrà un'uscita in base al secondo ingresso di selezione del mux 4:1(s_1). Lo schema è il seguente:

Implementazione

Per il multiplexer 4:1 abbiamo utilizzato un approccio per composizione utilizzando il multiplexer 2:1 già implementato. Il mux 4:1 presenta i 6 ingressi ($b_0, b_1, b_2, b_3, s_0, s_1$) (in) definiti come std_logic e 1 uscita (y_0) (out) anch'essa definita con std_logic.

Abbiamo quindi definito la seguente entity:

```
entity mux_4_1 is
    port(  b0 : in STD_LOGIC;
            b1 : in STD_LOGIC;
            b2 : in STD_LOGIC;
            b3 : in STD_LOGIC;
            s0 : in STD_LOGIC;
            s1 : in STD_LOGIC;
            y0 : out STD_LOGIC
        );
end mux_4_1;
```

Figura 2.3 entity mux_4_1

Visto che abbiamo utilizzato che componenti già esistenti, abbiamo utilizzato una descrizione *strutturale*. Tramite la descrizione strutturale descriviamo il multiplexer 4:1 tramite la sua struttura interna vedendo come i vari componenti sono collegati tra loro.

Le istanze dei multiplexer 2:1 vengono create con le etichette mux0, mux1 e mux2. Ogni istanza né connessa ai segnali di input e output come specificato nelle porte del componente.

```
-- Definizione architettura del modulo mux_4_1 con una descrizione strutturale.
architecture structural of mux_4_1 is
    signal u0 : STD_LOGIC := '0';
    signal u1 : STD_LOGIC := '0';

    component mux_2_1
        port( a0 : in STD_LOGIC;
              a1 : in STD_LOGIC;
              s : in STD_LOGIC;
              y : out STD_LOGIC
            );
    end component;

    begin
        mux0: mux_2_1
            Port map( a0 => b0,
                      a1 => b1,
                      s => s0,
                      y => u0
                    );
        mux1: mux_2_1
            Port map( a0 => b2,
                      a1 => b3,
                      s => s0,
                      y => u1
                    );
        mux2: mux_2_1
            Port map( a0 => u0,
                      a1 => u1,
                      s => s1,
                      y => y0
                    );
    end structural;
```

Figura 2.4 descrizione strutturale mux 4:1 tramite mux 2:1

Utilizziamo i segnali **u0** e **u1** come segnali intermedi per immagazzinare i risultati dei due multiplexer 2:1. In particolare, u0 è l'uscita del primo multiplexer mentre u1 è l'uscita del secondo

Contatore modulo n

Progetto e architettura

Il contatore modulo n è una macchina sequenziale che conta da 0 a $n-1$. Introduciamo un segnale di **load** per precaricare il contatore inizializzando a un valore specifico. Il valore iniziale è rappresentato da **init**. Inoltre, aggiungiamo anche un segnale di abilitazione. Visto che nel contatore modulo n è previsto un valore massimo di conteggio è necessario introdurre un ulteriore segnale di output che indica il **riporto**, questo valore è pari a 1 quando il contatore raggiunge il valore massimo.

Lo schema è il seguente:

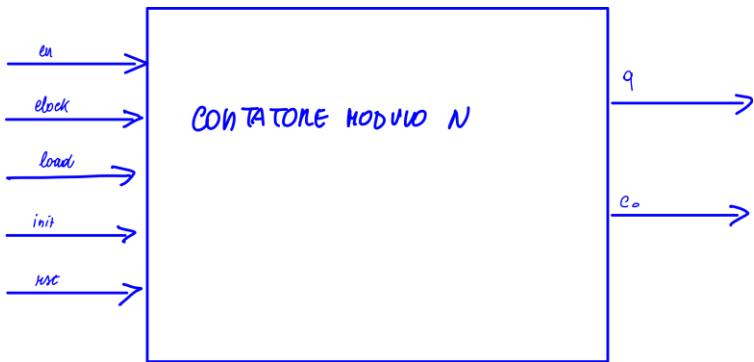


Figura 145 schema contatore modulo n

Implementazione

Definiamo l'interfaccia:

```
entity contatore_modn is
  generic(
    n : positive := 6;
    max : positive := 60
  );
  port(
    en : in std_logic;
    clk: in std_logic;
    load : in std_logic;
    init : in std_logic_vector(n-1 downto 0);
    rst : in std_logic;
    q : out std_logic_vector(n-1 downto 0);
    co : out std_logic
  );
end contatore_modn;
```

Figura 146 entity contatore modulo n

Dichiariamo due parametri generic:

- n: Parametro generico rappresentante la lunghezza del contatore in bit. Per default, è impostato a 6 bit.
- max: Parametro generico che indica il valore massimo raggiungibile dal contatore. Per default, è impostato a 60.

i port dichiarati sono i seguenti:

- en (enable): Segnale di abilitazione/disabilitazione del contatore.
- clk (clock)
- load: Segnale di caricamento iniziale per inizializzare il contatore con un valore specifico.
- init: Vettore di bit che rappresenta il valore iniziale del contatore.
- rst (reset): Segnale di reset per azzerare il contatore.
- q (output): Vettore di bit che rappresenta il valore corrente del contatore.
- co (carry-out): Segnale di carry-out, '1' quando il contatore raggiunge il valore massimo (max), altrimenti '0'.

Per la definizione dell'architettura utilizziamo un approccio behavioral:

```
architecture beh of contatore_modn is
    signal cont : std_logic_vector(n-1 downto 0) := (others => '0');
begin
    process(clk) begin
        if (load = '1') then
            cont <= init;
        end if;

        if(clk'event and clk = '1') then
            if(rst = '1') then
                cont <= (others => '0');
            else
                if(en = '1') then
                    cont <= std_logic_vector(unsigned(cont) + 1);
                    if unsigned(cont) = max then
                        cont <= (others => '0');
                    end if;
                end if;
            end if;
        end if;
    end process;
    q <= cont;
    co <= '1' when unsigned(cont) = max
                else '0';
end beh;
```

Figura 147 architettura contatore

Definiamo come prima cosa il segnale cont che conterrà il valore del conteggio con cui aggiornare l'uscita.

In seguito, definiamo un processo sensibile alle variazioni di clock.

La prima condizione (if (load = '1')) gestisce l'inizializzazione del contatore. Quando il segnale di caricamento (load) è attivo ('1'), il contatore viene inizializzato con il valore specificato nel segnale init. Questa parte assicura che il contatore assuma un valore iniziale personalizzato in maniera asincrona.

La seconda condizione (if(clk'event and clk = '1')) gestisce l'incremento del contatore al fronte di salita del clock (clk). Se il segnale di reset (rst) è attivo ('1'), il contatore viene azzerato.

Altrimenti, se il segnale di abilitazione (en) è attivo ('1'), il contatore viene incrementato di 1. Se il contatore raggiunge il valore massimo (max), viene riportato a zero. Questo garantisce che il contatore rimanga all'interno del range specificato.

Rom sequenziale

La rom viene utilizzata per memorizzare dati che devono essere letti ma non scritti durante l'esecuzione del programma.

La rom presenta come 4 ingressi:

- Un segnale di clock (clk) per sincronizzare le operazioni.
- Un segnale di indirizzo (address) che specifica la posizione di memoria a cui accedere.
- Un segnale di lettura (read) per richiedere l'estrazione di dati dalla ROM.
- Un segnale di dati in uscita (data) che fornisce il valore letto dalla memoria.

In output viene posto il dato letto.

Lo schema è il seguente:

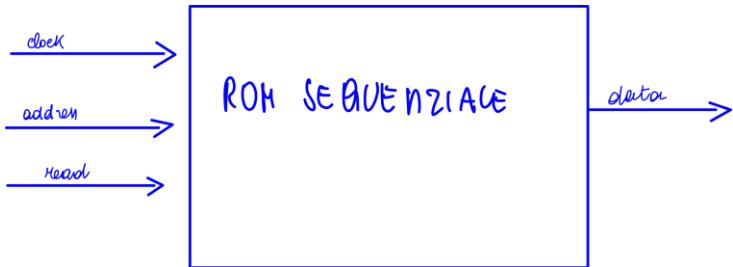


Figura 148 schema rom sequenziale

Implementazione

Come prima cosa definiamo l'entity:

```
entity rom_seq is
  generic(
    word_length : positive := 8;
    depth : positive := 4;
    numero_locazioni : positive := 2**4
  );
  port(
    clk : in std_logic; --segnale di clock
    address : in std_logic_vector(depth-1 downto 0);
    read: in std_logic; --segnale di lettura della stringa presente all'indirizzo di memoria
    data : out std_logic_vector (word_length-1 downto 0)
  );
end rom_seq;
```

Figura 149 entity rom sequenziale

Abbiamo dichiarato i seguenti parametri generici:

- word_length: Questo parametro specifica la lunghezza in bit di ciascuna parola nella memoria. Di default, è impostato a 8 bit.
- depth: Il parametro depth indica la profondità della memoria e la lunghezza degli indirizzi. È di default impostato a 4.
- numero_locazioni: Questo parametro rappresenta il numero totale di locazioni in memoria ed è calcolato come 2 elevato alla profondità ($2^{**\text{depth}}$).

I port dichiarati sono i seguenti:

Per definire l'architettura abbiamo utilizzato un approccio behavioral:

```

architecture behavioral of rom_seq is
    signal reg_address : integer range 0 to numero_locazioni-1 ; --registro per l'indirizzo
    type memory_16_4 is array(0 to numero_locazioni-1) of std_logic_vector(word_length-1 downto 0); --rappresenta le celle di memoria
    constant myrom : memory_16_4 :=(
        0 => "00000000",
        1 => "00000001",
        2 => "00000010",
        3 => "00000011",
        4 => "00000100",
        5 => "00000101",
        6 => "00000110",
        7 => "00000111",
        8 => "01101101",
        9 => "00110111",
        10 => "00110111",
        11 => "00110111",
        12 => "00110111",
        13 => "00110111",
        14 => "00110111",
        15 => "00110111"
    );
begin
process(clk) begin
    if (clk'event and clk = '0' and read = '1') then --se si abbassa il bit di abilitazione....
        reg_address <= to_integer(unsigned(address)); --trasferisci l'indirizzo nel registro
    end if;
end process;
data <= myrom(reg_address);
end behavioral;

```

Figura 150 architettura rom sequenziale

Dichiariamo i seguenti segnali:

- `reg_address`: è un segnale intero utilizzato come registro per memorizzare l'indirizzo di memoria richiesto. È definito nel range da 0 a `numero_locazioni-1`.
- `memory_16_4`: tipo di dato array che rappresenta le celle di memoria della ROM. Ogni cella è un vettore di bit con una lunghezza specificata dal parametro `word_length`.
- `myrom`: una costante di tipo `memory_16_4` che inizializza la ROM con dati arbitrari.

Successivamente dichiariamo un processo sensibile al fronte del clock, in figura è di discesa. All'interno di questo processo, si controlla se il segnale di lettura (`read`) è attivo e se il fronte di discesa del clock è appena avvenuto. Se queste condizioni sono soddisfatte, l'indirizzo fornito (`address`) viene convertito in un numero intero senza segno (`to_integer(unsigned(address))`) e memorizzato in `reg_address`.

L'uscita `data` è assegnata al valore memorizzato nella posizione di memoria corrispondente all'indirizzo presente in `reg_address`.

Memoria

Si veda memoria trasmettitore handshaking

Botton Debouncer

Il Botton Debouncer è un componente ripulisce il segnale in input dal bottone in modo da presentare in uscita un impulso della durata di un colpo di clock per segnalare l'avvenuta pressione del bottone. Il debouncing dei pulsanti è una tecnica comunemente utilizzata nella progettazione di circuiti digitali per garantire che la singola pressione di un pulsante o di un interruttore produca un unico segnale di uscita. Senza il debouncing, la natura meccanica di questi pulsanti o interruttori può far sì che il segnale di uscita "rimbalzi" rapidamente tra gli stati on e off prima di stabilizzarsi, con conseguenti comportamenti imprevisti. Di seguito è riportato il codice VHDL di una FSM che implementa questo meccanismo:

```

entity button_debouncer is
  generic (
    CLK_period: integer := 10; -- periodo del clock (della board) in nanosecondi
    btn_noise_time: integer := 10000000 -- durata stimata dell'oscillazione del bottone in nanosecondi
    |   |   |   |   |   |   |   |   -- il valore di default è 10 millisecondi
  );
  Port ( RST : in STD_LOGIC;
         CLK : in STD_LOGIC;
         BTN : in STD_LOGIC;
         CLEARED_BTN : out STD_LOGIC);
end button_debouncer;

architecture Behavioral of button_debouncer is
  -- questo componente prende in input il segnale proveniente dal bottone e genera un
  -- segnale "ripulito" che presenta un impulso della durata di un colpo di clock per
  -- segnalare l'avvenuta pressione del bottone.
  -- Il debouncer implementa un semplice automa di 4 stati:
  -- si attende un certo tempo in modo da "superare" l'oscillazione: se il bottone è ancora alto
  -- si alza il segnale ripulito in output (avrà la durata di un impulso di clock)
  -- e si va in PRESED; anche qui quando si rileva BTN=0
  -- si va in uno stato intermedio CHK_NOT_PRESSED in cui si aspetta un certo tempo per superare
  -- l'oscillazione: se dopo questo tempo è ancora basso si ritorna in NOT_PRESSED
  -- con questo automa se si mantiene il bottone premuto non vengono generati più impulsi in
  -- uscita

  type state is (NOT_PRESSED, CHK_PRESSED, PRESSED, CHK_NOT_PRESSED);
  signal BTN_state : state := NOT_PRESSED;

  constant max_count : integer := btn_noise_time/CLK_period; -- 10000000/10= conto 1000000 colpi di clock

begin
  begin
    if rising_edge(CLK) then
      if( RST = '1') then
        BTN_state <= NOT_PRESSED;
        CLEARED_BTN <= '0';
      else
        case BTN_state is
          when NOT_PRESSED =>
            if( BTN = '1' ) then
              BTN_state <= CHK_PRESSED;
            else
              BTN_state <= NOT_PRESSED;
            end if;
          when CHK_PRESSED =>
            if(count = max_count -1) then
              if(BTN = '1') then --se arrivo a count max ed è ancora alto vuol dire che non era un bounce, devo alzare CLEARED_BTN
                count:=0;
                CLEARED_BTN <= '1';
                BTN_state <= PRESSED;
              else
                count:=0;
                BTN_state <= NOT_PRESSED;
              end if;
            else
              count:= count+1;
              BTN_state <= CHK_PRESSED;
            end if;
          when PRESSED =>
            CLEARED_BTN<= '0'; --questo lo metto per fare in modo che il segnale sia alto per un solo impulso di clock
            if(BTN = '0') then
              BTN_state <= CHK_NOT_PRESSED;
            else
              BTN_state <= PRESSED;
            end if;
          when CHK_NOT_PRESSED =>
            if(count = max_count -1) then
              if(BTN = '0') then --se arrivo a count max ed è ancora basso vuol dire che non era un bounce e il bottone è stato rilasciato
                count:=0;
                BTN_state <= NOT_PRESSED;
              else
                count:=0;
                BTN_state <= PRESSED;
              end if;
            else
              count:= count+1;
              BTN_state <= CHK_NOT_PRESSED;
            end if;
          when others =>
            BTN_state <= NOT_PRESSED;
        end case;
      end if;
    end if;
  end process;

```

Il codice è strutturato attorno a un process che viene attivato sul fronte di salita del clock:

- 1) **NOT_PRESSED:** se il pulsante (*BTN*) è alto, lo stato del pulsante viene impostato su *CHK_PRESSED*, indicando che la pressione del pulsante viene verificata per il rimbalzo. Se il pulsante non è alto, lo stato del pulsante rimane *NOT_PRESSED*.
- 2) **CHK_PRESSED:** un contatore viene controllato rispetto al conteggio calcolato a partire dalle costanti dichiarate come *generics*. Se il contatore ha raggiunto questo massimo e il pulsante è ancora alto, si presume che la pressione del pulsante non sia un rimbalzo. Il contatore viene azzerato, e viene

generato l'impulso alto ripulito e lo stato del pulsante viene impostato su *PRESSED*. Se a questo punto il pulsante non è alto, lo stato del pulsante viene riportato a *NOT_PRESSED*.

- 3) **PRESSED:** Viene abbassata l'uscita per garantire che il segnale sia alto per un solo impulso di clock. Se il pulsante è basso, lo stato del pulsante viene impostato su *CHK_NOT_PRESSED*, indicando che il rilascio del pulsante viene controllato per evitare il rimbalzo. Se il pulsante è ancora alto, lo stato del pulsante rimane *PRESSED*.
- 4) **CHK_NOT_PRESSED:** simile a *CHK_PRESSED*, il contatore viene controllato rispetto al conteggio massimo. Se il contatore ha raggiunto il massimo e il pulsante è ancora basso, si presume che il rilascio del pulsante non sia un rimbalzo. Il contatore viene azzerato e lo stato del pulsante viene impostato su *NOT_PRESSED*. Se il pulsante non è ancora basso a questo punto, lo stato del pulsante viene riportato a *PRESSED*.