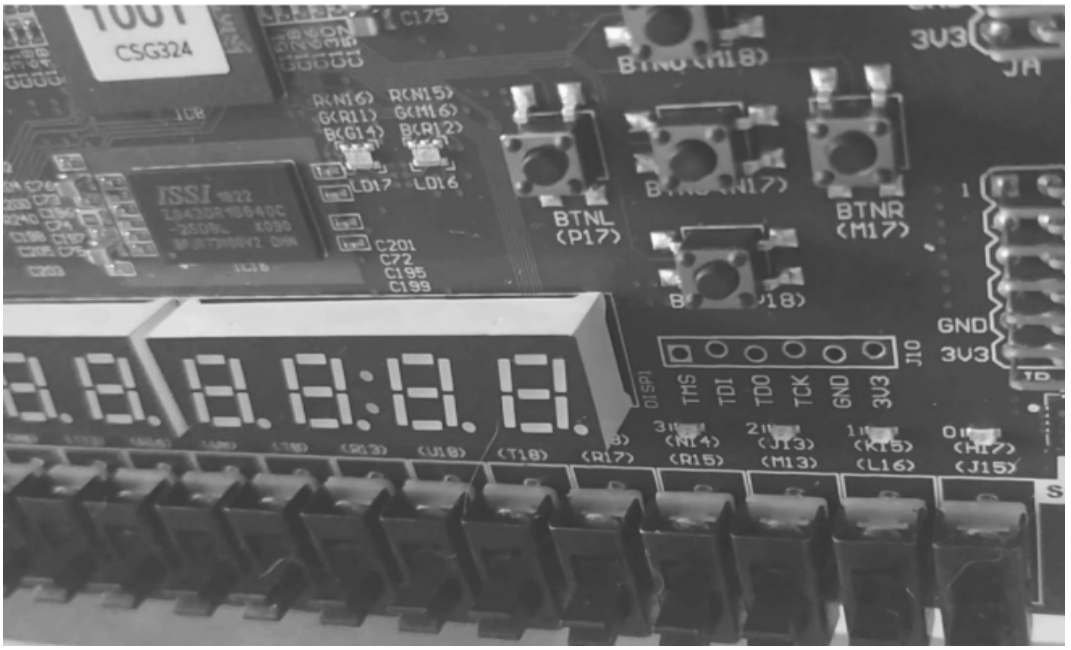


Documentazione dei Progetti di Architettura dei Sistemi Digitali

---

Gruppo 41

Mennillo Valerio M63/001277
Orlando Gian Marco M63/001326
Perillo Marco M63/001313
Russo Diego M63/001335

Indice

Multiplexer **5**
1.1 Multiplexer 16:1 6
1.1.1 Soluzione 6
1.1.2 Realizzazione 6
1.1.3 Simulazione 10
1.2 Multiplexer 16:4 12
1.2.1 Soluzione 12
1.2.2 Realizzazione 12
1.2.3 Simulazione 16
1.3 Multiplexer 16:4 su FPGA 17
1.3.1 Soluzione 17
1.3.2 Sintesi 18

Encoder **20**
2.1 Encoder BCD 20
2.1.1 Soluzione 20
2.1.2 Realizzazione 21
2.1.3 Simulazione 22
2.2 Encoder BCD su FPGA (visualizzazione tramite LED) 23
2.2.1 Soluzione 23
2.2.2 Sintesi 23
2.3 Encoder BCD su FPGA (visualizzazione tramite display) 24
2.3.1 Soluzione 24
2.3.2 Realizzazione 25
2.3.3 Sintesi 28

Riconoscitore di sequenze **31**
3.1 Riconoscitore 32
3.1.1 Soluzione 33
3.1.2 Realizzazione 33
3.1.3 Simulazione 39
3.2 Riconoscitore su FPGA 42
3.2.1 Soluzione 42
3.2.1 Realizzazione 43
3.2.1 Simulazione e sintesi 51

Registro a scorrimento **55**
4.1 Registro a scorrimento circolare 56
4.1.1 Soluzione 56
4.1.2 Realizzazione (comportamentale) 57
4.1.3 Realizzazione (strutturale) 58

4.1.4 Simulazione 65

Cronometro 67

5.1 Cronometro 67

5.1.1 Soluzione 68

5.1.2 Realizzazione 68

5.1.3 Simulazione 77

5.2 Cronometro su FPGA 78

5.2.1 Soluzione 78

5.2.2 Realizzazione 80

5.2.3 Sintesi 93

5.3 Cronometro su FPGA con intertempi 95

5.3.1 Soluzione 95

5.3.2 Realizzazione 96

5.3.3 Sintesi 105

Sistema di testing 108

6.1 Sistema di testing 109

6.1.1 Soluzione 109

6.1.2 Realizzazione 110

6.1.3 Simulazione 115

6.1.4 Sintesi 117

Protocolli di comunicazione 119

7.1 Comunicazione con handshaking 120

7.1.1 Soluzione 121

7.1.2 Realizzazione 121

7.1.3 Simulazione 144

Processore MIC-1 146

8.1 Analisi istruzioni del processore 155

8.1.1 Soluzione 155

8.1.2 Realizzazione 156

8.1.3 Simulazione IAND 160

8.1.3 Simulazione ISUB 161

8.2 Overloading di operatore 162

8.2.1 Soluzione 162

8.2.1 Simulazione 163

Trasmissione seriale - UART/USART 164

9.1 Interfaccia seriale su FPGA 166

9.1.1 Realizzazione e sintesi 166

9.2 Interfaccia seriale con ROM 171

9.2.1 Soluzione 171

9.2.2 Realizzazione 172

9.2.3 Simulazione 182

9.2.4 Sintesi 184

Switch multistadio **185**

10.1 Switch multistadio: modello Omega Network 186

10.1.1 Soluzione 186

10.1.2 Realizzazione 187

10.1.3 Simulazione 196

Macchine aritmetiche **199**

11.1 Moltiplicatore di Robertson 201

11.1.1 Soluzione 201

11.1.2 Realizzazione 201

11.1.3 Simulazione 217

11.1.4 Sintesi 219

Gestione delle collisioni in Omega Network **223**

12.1 Gestione delle collisioni tramite store and forward 224

12.1.1 Soluzione 224

12.1.2 Realizzazione 228

12.1.3 Simulazione 244

Nexys A7-50T **248**

Introduzione

Ogni capitolo di questo elaborato è dedicato ad uno dei progetti realizzati per il corso di Architettura dei Sistemi Digitali. Abbiamo deciso di suddividere tali capitoli in più paragrafi, cercando di riportare il processo creativo che ci ha portato alla progettazione e poi alla realizzazione ed implementazione del sistema oggetto dell'esercizio. Dopo una breve introduzione teorica, dunque, ogni capitolo procede con un primo paragrafo dedicato alla *"Soluzione"* ideata, in cui abbiamo descritto le idee progettuali, senza addentrarci eccessivamente in dettagli puramente implementativi riportati, invece, approfonditamente nei paragrafi dedicati alla *"Realizzazione"*. In tali paragrafi verranno riportati, infatti, sia gli schematici dedicati ad ogni singola componente del sistema oltre che il codice VHDL relativo realizzato. Da sottolineare che, per quanto riguarda le componenti più semplici, al fine di alleggerire la lettura dell'elaborato stesso, verranno riportate solo la prima volta che sono state introdotte (a meno di importanti modifiche), concentrandoci prevalentemente sulle componenti di maggior rilievo.
Infine, laddove richiesto dalla traccia, abbiamo ulteriormente approfondito le fasi di simulazione e sintesi negli appositi paragrafi.

Capitolo 1

Multiplexer

Una macchina combinatoria è un circuito costituito da porte logiche, le cui uscite in ogni istante assumono valori che dipendono unicamente dal valore degli ingressi in quello stesso istante.

Una macchina combinatoria, dunque, è una macchina che non ha memoria. Possono essere classificate in:

- macchine per l'instradamento di segnali e la connessione tra sistemi;
- macchine per la conversione di codici;
- macchine per la realizzazione di funzioni aritmetiche e logiche.

Il multiplexer è una macchina combinatoria notevole per l'instradamento di segnali e la connessione tra sistemi. Permette di risolvere i problemi di connessione tra sorgenti e destinazioni. Il multiplexer è una macchina combinatoria utilizzata quando più linee di ingresso devono essere instradate verso un'unica linea di uscita. Per fare ciò, viene utilizzata una logica di selezione. Il multiplexer, dunque, è un *"convogliatore"*. Il tipo di multiplexer che andremo ad utilizzare è un multiplexer indirizzabile che, a differenza di un multiplexer lineare, non presenta tante linee di selezione quanti sono gli ingressi bensì sono in numero $log_2(ingressi)$.

Il demultiplexer è la macchina complementare al multiplexer in quanto consente di realizzare connessioni costituite da un unico ingresso e più uscite. In questo caso, tramite apposite linee di selezione, vengono abilitate le diverse uscite. Il demultiplexer, dunque, è un *"distributore"*. Come per il multiplexer, andremo ad utilizzare demultiplexer indirizzabili.

1.1 Multiplexer 16:1

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.

1.1.1 Soluzione

Come specificato dalla traccia, l'approccio scelto per la realizzazione della soluzione è quello di tipo strutturale. L'approccio strutturale permette di decomporre la macchina da implementare in componenti più piccoli in modo da garantire il riuso e la modularità. Un ulteriore vantaggio di tale approccio, inoltre, consiste nel rendere i componenti facili da generalizzare ed iterare.

L'entità più esterna - la top-module - dovrà avere 16 ingressi e 1 uscita. Abbiamo, allora, progettato un multiplexer 16:1 come composizione di più multiplexer 4:1. In particolare, sono stati utilizzati 5 multiplexer in totale: 4 di questi sono stati posizionati al "primo stadio" mentre l'ultimo multiplexer al "secondo stadio". Gli ingressi dei multiplexer al "primo stadio" sono direttamente collegati con gli ingressi della top-module. Le 4 uscite ottenute saranno gli ingressi del multiplexer al "secondo stadio". L'uscita di quest'ultimo multiplexer costituisce, a sua volta, l'uscita della top-module.

1.1.2 Realizzazione

Come spiegato in precedenza, il modulo più piccolo utilizzato è il multiplexer 4:1. Di seguito ne riportiamo uno schematico per maggiore chiarezza.

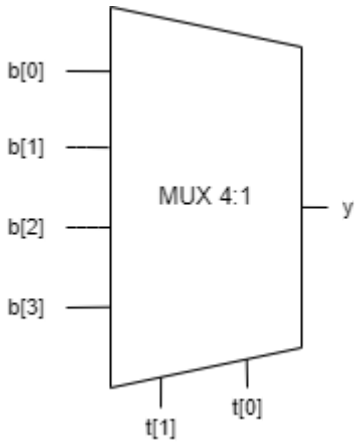


Figura 1.1 - Multiplexer indirizzabile 4:1

Di seguito, riportiamo l'implementazione realizzata per il multiplexer 4:1.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY mux4_1 IS
	PORT
	(
		b : IN std_logic_vector(3 DOWNTO 0);
		t : IN std_logic_vector(1 DOWNTO 0);
		y : OUT std_logic
	);
END mux4_1;

ARCHITECTURE Behavioral OF mux4_1 IS
BEGIN
	PROCESS (b, t)
	BEGIN
		IF (t(1) = '0' AND t(0) = '0') THEN
			y <= b(0);
		ELSIF (t(1) = '0' AND t(0) = '1') THEN
			y <= b(1);
		ELSIF (t(1) = '1' AND t(0) = '0') THEN
			y <= b(2);
		ELSIF (t(1) = '1' AND t(0) = '1') THEN
			y <= b(3);
		ELSE
			y <= '-';
		END IF;
	END PROCESS;
END Behavioral;
```

Come già detto, il multiplexer implementato presenta 6 ingressi (di cui 2 di selezione) e restituisce un'unica uscita. Abbiamo scelto un approccio comportamentale per descrivere il comportamento interno del singolo multiplexer.

Per quanto riguarda la top-module, invece, abbiamo realizzato il multiplexer 16:1 come composizione di più multiplexer 4:1. Il "primo stadio" di multiplexer seleziona l'elemento *i-esimo* per ogni blocco mentre il singolo multiplexer sul "secondo stadio" seleziona da quale dei 4 multiplexer prendere il dato. Di seguito riportiamo lo schematico dell'intero sistema, comprensivo dei segnali utilizzati durante la stesura del codice.

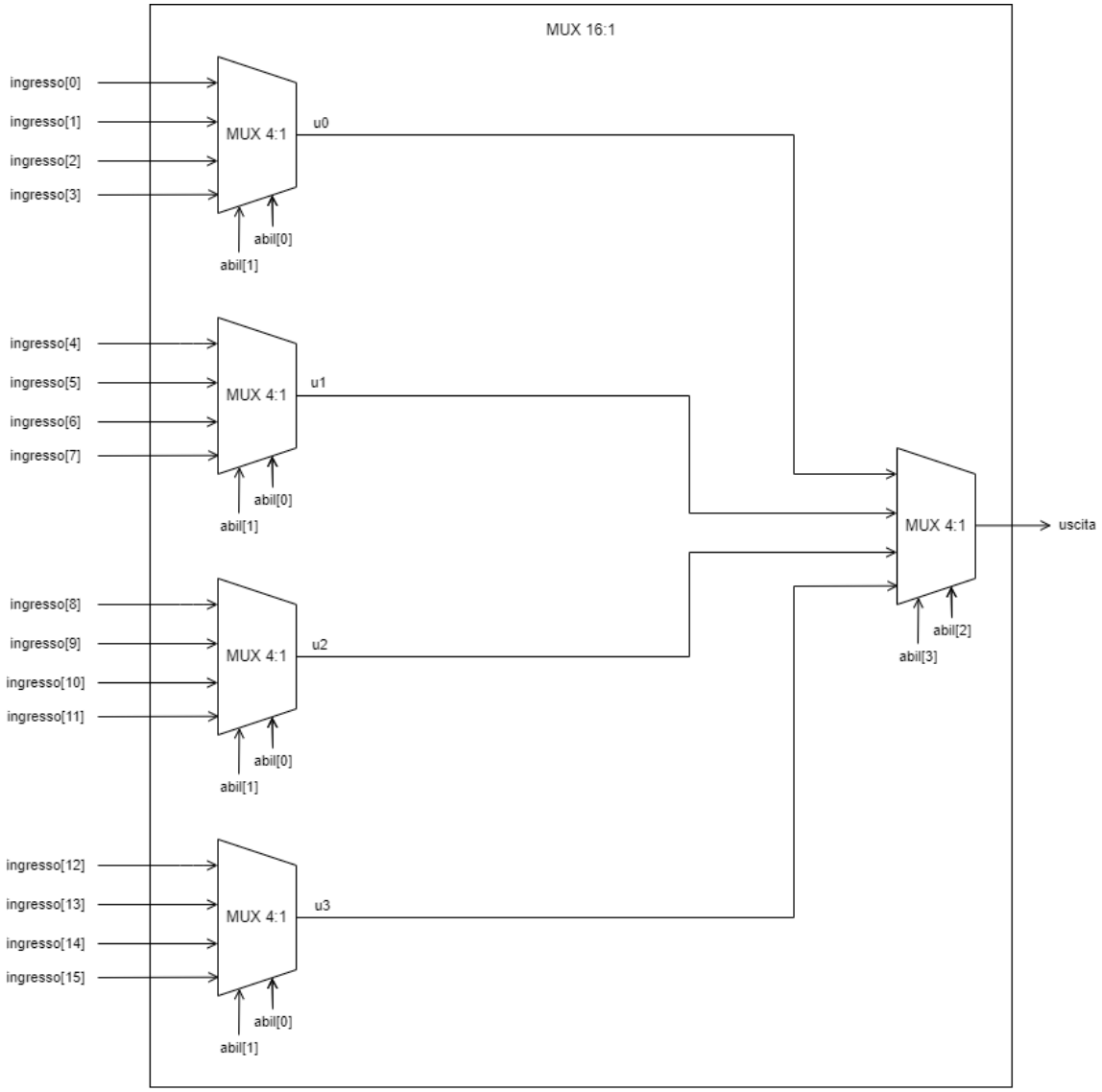


Figura 1.2 - Multiplexer 16:1

Di seguito riportiamo il codice VHDL realizzato per l'implementazione del Multiplexer 16:1 ed una breve spiegazione.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY mux16 IS
	PORT
	(
		ingressi     : IN std_logic_vector(15 DOWNTO 0);
		abilitazioni : IN std_logic_vector(3 DOWNTO 0);
		uscita       : OUT std_logic
	);
END mux16;

ARCHITECTURE Structural OF mux16 IS
	SIGNAL u : std_logic_vector(3 DOWNTO 0);
	COMPONENT mux4_1 IS
		PORT
		(
			b : IN std_logic_vector(3 DOWNTO 0);
			t : IN std_logic_vector(1 DOWNTO 0);
			y : OUT std_logic
		);
	END COMPONENT;

BEGIN
	mux0to3 : FOR i IN 0 TO 3 GENERATE
		mux     : mux4_1
		PORT MAP
		(
			b(3 DOWNTO 0) => ingressi(i * 4 + 3 DOWNTO 4 * i),
			t(1 DOWNTO 0) => abilitazioni(1 DOWNTO 0),
			y             => u(i)
		);
	END GENERATE;

	mux4 : mux4_1
	PORT MAP
	(
		b(3 DOWNTO 0) => u(3 DOWNTO 0),
		t(1 DOWNTO 0) => abilitazioni(3 DOWNTO 2),
		y             => uscita
	);
END Structural;
```

Dopo la dichiarazione dell'entità della top-module (*mux16*) con le rispettive porte di ingresso ed uscita, andiamo ad istanziare le componenti che costituiscono il sistema. I primi quattro multiplexer 4:1 vengono istanziati iterativamente tramite il costrutto *for-generate* sfruttando un iteratore *i* per la configurazione della port map. Gli ingressi dell'ultimo multiplexer, invece, sono interni alla top-module. In tal caso, dunque,

abbiamo utilizzato quattro *signal* che ci hanno permesso di collegare le uscite dei multiplexer del “primo stadio” con gli ingressi del multiplexer al “secondo stadio”.

1.1.3 Simulazione

Per verificare il funzionamento e la tempificazione della macchina è necessario realizzare il testbench. Tale file presenta un’entità che dichiara ed istanzia il blocco che vogliamo testare. Il corpo di tale entità è vuoto in quanto questa non è un oggetto che deve essere realmente realizzato ma ci serve solo per poter testare il sistema. Di seguito si riportano gli stimoli (presenti all’interno di un *process*) che abbiamo sottoposto al sistema per poter effettuare tale fase di testing.

```
stim_proc: process
    begin
        wait for 100 ns;
        ingressi(0) <= '1';
        wait for 20 ns;
        ingressi(0) <= '0';
        wait for 20 ns;
        ingressi(13) <= '1';
        ingressi(14) <= '1';
        ingressi(15) <= '1';
        abilitazioni(0) <= '1';
        abilitazioni(1) <= '1';
        abilitazioni(2) <= '1';
        abilitazioni(3) <= '1';
        wait for 20 ns;
        ingressi(13) <= '0';
        ingressi(14) <= '0';
        ingressi(15) <= '0';
        abilitazioni(0) <= '0';
        abilitazioni(1) <= '0';
        abilitazioni(2) <= '0';
        abilitazioni(3) <= '0';
        wait for 20 ns;
        ingressi(13) <= '0';
        ingressi(14) <= '1';
        ingressi(15) <= '0';
        abilitazioni(0) <= '1';
        abilitazioni(1) <= '0';
        abilitazioni(2) <= '1';
        abilitazioni(3) <= '1';
        wait for 20 ns;
        ingressi(13) <= '1';
        ingressi(14) <= '1';
        ingressi(15) <= '0';
        abilitazioni(0) <= '1';
        abilitazioni(1) <= '0';
        abilitazioni(2) <= '1';
```

```
        abilitazioni(3) <= '1';
        wait;
    end process;
```

Innanzitutto, notiamo che prima di effettuare qualsiasi verifica aspettiamo un certo intervallo di tempo affinché i transitori si stabilizzino al fine di evitare di “prendere” valori errati. Subito dopo possiamo inserire i valori degli ingressi e delle abilitazioni. Eseguendo la simulazione, possiamo verificare il valore riportato in uscita dal sistema. Riportiamo di seguito il grafico ottenuto dalla simulazione.

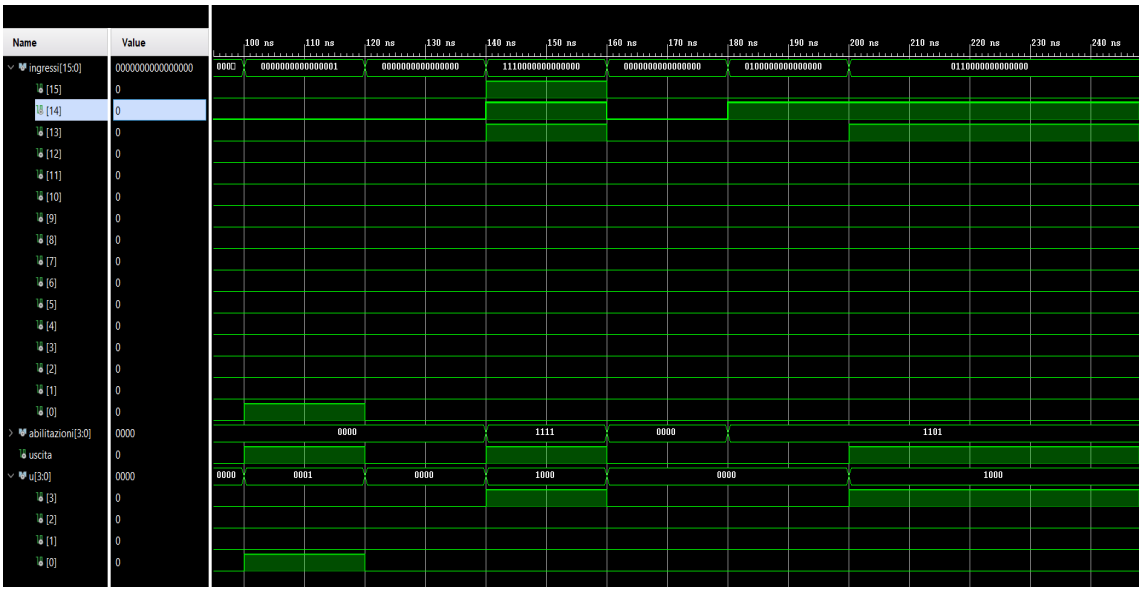


Figura 1.3 - Grafico della simulazione

Come possiamo notare dal grafico, gli input testati presentano gli output attesi. Infatti: quando i selettori sono impostati al valore “0000” e l’input dell’ingresso 0 è alto, questo viene mostrato in uscita. In questo caso abbiamo testato la macchina con un unico ingresso alto. Successivamente, impostiamo il valore dei selettori a “1111” (15) ed alziamo il valore degli ingressi 13, 14 e 15. Correttamente, in uscita viene mostrato il valore dell’input 15. L’ultimo test effettuato è stato quello che presenta il valore dei selettori a “1101” (13). In tal caso, abbiamo alzato il valore degli ingressi 13 e 14, entrambi appartenenti all’ultimo multiplexer (*mux3*) del “primo stadio”. Notiamo che i due bit meno significativi del selettore mi permettono di selezionare correttamente l’ingresso 13, che viene portato in uscita al multiplexer corrispondente e quindi in

ingresso al multiplexer del “secondo stadio” attraverso il *signal u3* (anch’esso visibile nel grafico di Figura 1.3) che lo selezionerà tramite i due bit più significativi del selettore per presentarlo poi correttamente in uscita al sistema.
I casi di test in questo esercizio (ed in generale di quelli a venire), sono stati scelti manualmente cercando di individuare i casi di interesse ed i casi limite. Ovviamente, a causa di ciò, l’indice di test coverage non risulta eccessivamente elevato.

1.2 Multiplexer 16:4

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.

1.2.1 Soluzione

Vincolati all’utilizzo del multiplexer 16:1 realizzato al punto precedente, il quale ha una sola uscita, abbiamo bisogno di una logica che porti la sua uscita a quella complessiva della macchina combinatoria. Per fare ciò, abbiamo utilizzato un demultiplexer 1:4.

1.2.2 Realizzazione

L’interfaccia della top-module in questo caso mantiene 16 ingressi ma ora presenta 4 uscite. Mantenendo un approccio strutturale, abbiamo riutilizzato il multiplexer 16:1 precedentemente realizzato come modulo interno. L’instradamento verso una delle quattro uscite della top-module del segnale in uscita al multiplexer 16:1 viene effettuato attraverso un demultiplexer 1:4. In particolare, il demultiplexer viene pilotato con 2 bit di selezione che, aggiunti ai 4 necessari per selezionare l’uscita del multiplexer 16:1, portano complessivamente a 6 il totale dei bit di selezione. In totale ci saranno quindi 16 bit di ingresso e 6 di selezione.
Di seguito riportiamo lo schematico del componente demultiplexer 1:4 utilizzato.

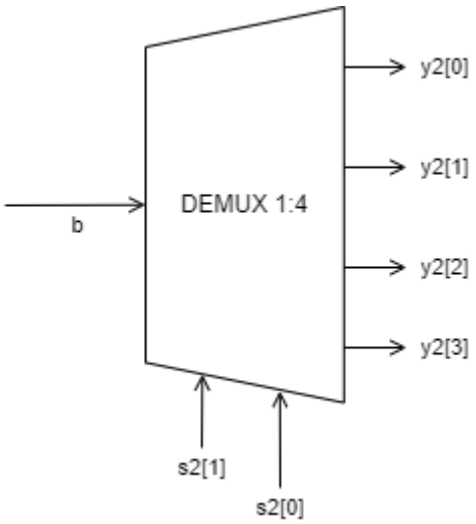


Figura 1.4 - Demultiplexer indirizzabile 1:4

Di seguito riportiamo, inoltre, l'implementazione realizzata per il demultiplexer 1:4.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY demux_1_4 IS
	PORT
	(
		b  : IN std_logic;
		s2 : IN std_logic_vector(1 DOWNTO 0);
		y2 : OUT std_logic_vector(3 DOWNTO 0)
	);
END demux_1_4;

ARCHITECTURE Behavioral OF demux_1_4 IS
BEGIN
	PROCESS (b, s2)
	BEGIN
		y2 <= (OTHERS => '0'); --Azzeriamo l'uscita per ricalcolare
ogni volta la nuova destinazione
		IF (s2(1) = '0' AND s2(0) = '0') THEN
			y2(0) <= b;
		ELSIF (s2(1) = '0' AND s2(0) = '1') THEN
			y2(1) <= b;
		ELSIF (s2(1) = '1' AND s2(0) = '0') THEN
			y2(2) <= b;
		ELSIF (s2(1) = '1' AND s2(0) = '1') THEN
			y2(3) <= b;
		ELSE
			y2 <= "----";
			END IF;
	END PROCESS;

END Behavioral;
```

Anche in questo caso abbiamo adottato un approccio *"Behavioral"* per poter gestire la logica del demultiplexer. Notiamo che il *process* viene attivato quando i segnali *b* ed *s2* (rispettivamente il segnale di ingresso e quello di selezione), entrambi presenti all'interno della sensitivity list, effettuano una commutazione. Notiamo, inoltre, che all'inizio del processo (prima di effettuare qualsiasi selezione dell'uscita) andiamo ad azzerare tutte le uscite al fine di instradare correttamente il valore sull'uscita selezionata.

Di seguito, riportiamo lo schematico relativo all'implementazione della top-module, costituita - come già anticipato - dal componente multiplexer 16:1 precedentemente realizzato e dal demultiplexer 1:4 le cui uscite sono collegate alle uscite del sistema complessivo.

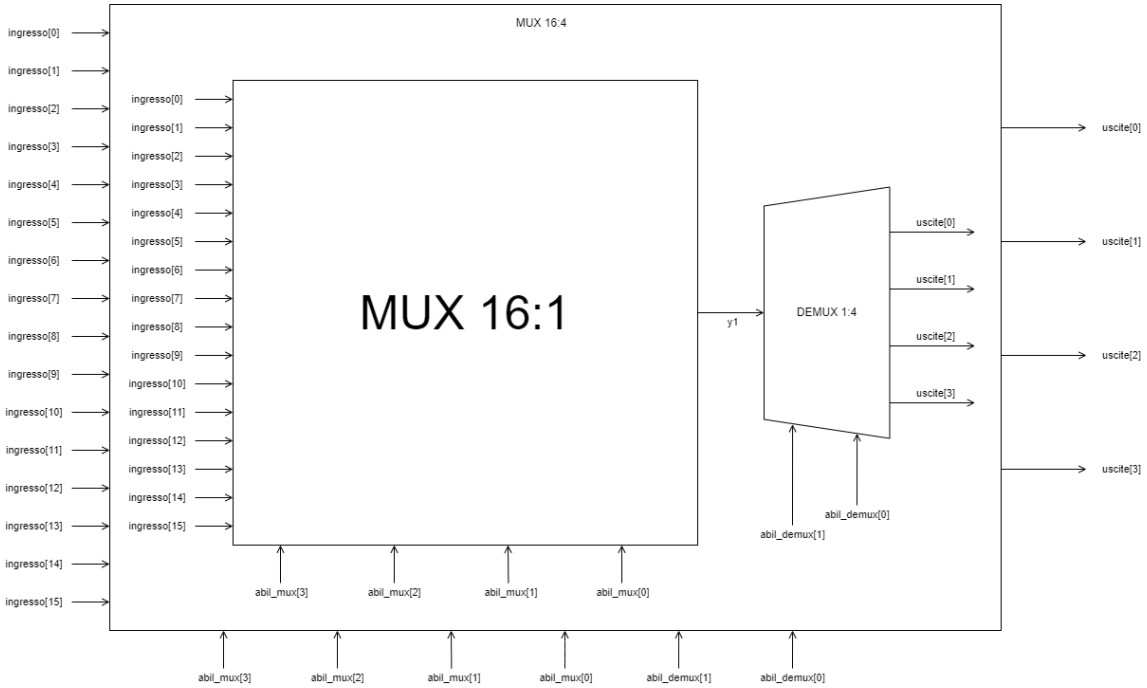


Figura 1.5 - Multiplexer 16:4

Di seguito riportiamo, inoltre, l'implementazione della top-module.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY mux_16_4 IS
       PORT
       (
              ingressi            : IN std_logic_vector(15 DOWNTO 0);
```

```
		abilitazioni_mux   : IN std_logic_vector(3 DOWNTO 0);
		abilitazioni_demux : IN std_logic_vector(1 DOWNTO 0);
		uscite             : OUT std_logic_vector(3 DOWNTO 0)
	);
END mux_16_4;

ARCHITECTURE Structural OF mux_16_4 IS
	SIGNAL u : std_logic;

	COMPONENT mux16 IS
		PORT
		(
			a  : IN std_logic_vector(15 DOWNTO 0);
			s1 : IN std_logic_vector(3 DOWNTO 0);
			y1 : OUT std_logic
		);
	END COMPONENT;

	COMPONENT demux_1_4 IS
		PORT
		(
			b  : IN std_logic;
			s2 : IN std_logic_vector(1 DOWNTO 0);
			y2 : OUT std_logic_vector(3 DOWNTO 0)
		);
	END COMPONENT;

BEGIN
	multiplexer : mux16
	PORT MAP
	(
		a(15 DOWNTO 0) => ingressi(15 DOWNTO 0),
		s1(3 DOWNTO 0) => abilitazioni_mux(3 DOWNTO 0),
		y1             => u
	);

	demultiplexer : demux_1_4
	PORT MAP
	(
		b              => u,
		s2(1 DOWNTO 0) => abilitazioni_demux(1 DOWNTO 0),
		y2(3 DOWNTO 0) => uscite(3 DOWNTO 0)
	);

END Structural;
```

Notiamo che, oltre alla dichiarazione dell'entità principale ed alle istanze dei due componenti che costituiscono il sistema, abbiamo utilizzato il *signal u* per poter collegare internamente l'uscita del multiplexer con l'ingresso del demultiplexer.

1.2.3 Simulazione

Uno dei vantaggi di procedere in maniera strutturale risiede nel poter testare singolarmente i componenti e poi collegare i vari moduli. Avendo già testato il multiplexer 16:1 nel punto precedente, la seguente simulazione si concentra sul demultiplexer e sull'interazione con il multiplexer.

Di seguito riportiamo il codice del *process* presente all'interno del testbench ed il grafico relativo.

```
stim_proc: process
    begin
        wait for 100 ns;
        ingressi <= "1000000000000000";
        wait for 20 ns;
        abilitazioni_demux <= "01";
        wait for 20 ns;
        abilitazioni_mux <= "0101";
        wait for 20 ns;
        abilitazioni_mux <= "1111";
        abilitazioni_demux <= "11";
        wait;
end process;
```

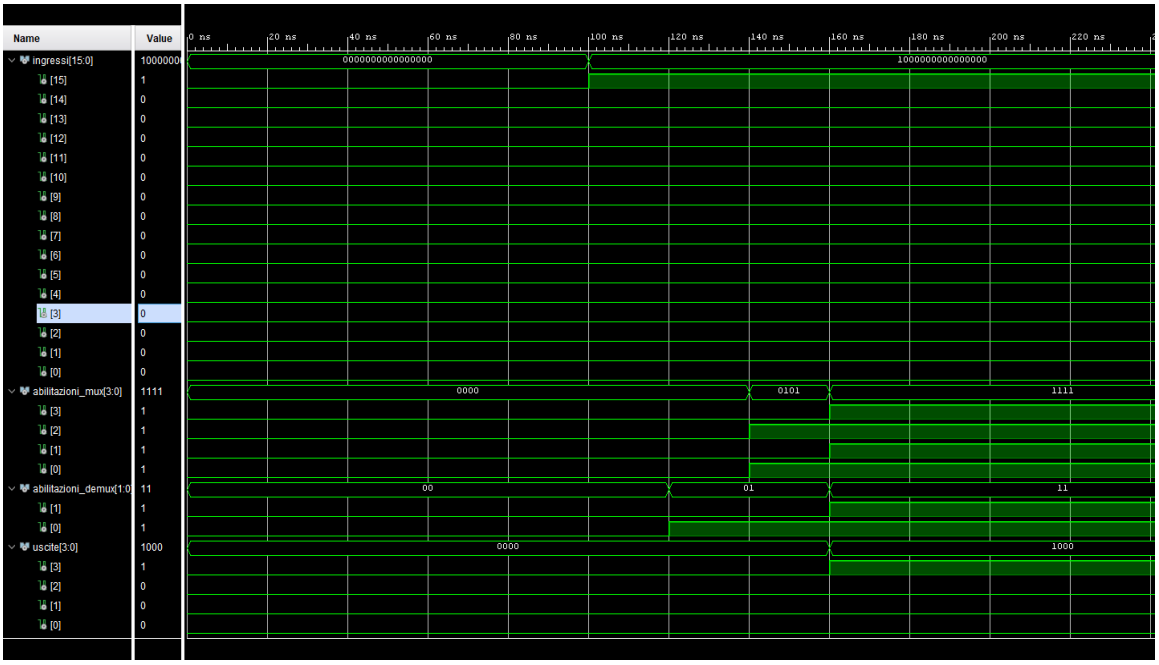


Figura 1.6 - Grafico della simulazione

Inizialmente l'uscita del demultiplexer rimane a 0. L'ingresso è fisso da 100 ns fino alla fine della simulazione. A 100 ns, quindi, abbiamo impostato il demultiplexer al valore

“00” e dunque instrada l’uscita del 16:1 sul primo filo che è comunque “0”. A 120 ns si alza uno dei flag di selezione del demultiplexer che, comunque, lasciano in uscita “0”. La situazione cambia a 160 ns dove i segnali di selezione del multiplexer si alzano insieme a quelli del demultiplexer e portano l’ultimo ingresso (*ingresso[15]*) direttamente in uscita al 16:1 per poi essere instradato dal demultiplexer 1:4 verso la terza uscita del sistema complessivo.

1.3 Multiplexer 16:4 su FPGA

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 dati in input, essi possono essere precaricati nel sistema oppure immessi anch’essi mediante switch, sviluppando in questo secondo caso un’apposita rete di controllo per l’acquisizione.

1.3.1 Soluzione

Questo è stato il nostro primo esercizio da sintetizzare sulla board e, dunque, abbiamo riscontrato per la prima volta problemi di natura pratica. La nostra macchina, per come è stata progettata, presenta in totale 22 ingressi di cui:

- 16 segnali di ingresso dei dati (*ingressi*)
- 6 segnali di selezione del multiplexer e demultiplexer (*abilitazioni_mux* e *abilitazioni_demux*)

Avendo a disposizione solo 16 switch sulla scheda Nexys A7-50T e non avendo ancora approcciato l’utilizzo dei bottoni, la scelta che abbiamo fatto è di rappresentare con gli switch solo i 6 segnali di selezione del multiplexer e demultiplexer grazie ai quali abbiamo potuto testare al meglio il funzionamento logico del sistema a discapito dei 16 segnali di ingresso che abbiamo, invece, precaricato tramite un apposito *signal* per verificare il corretto funzionamento delle macchine di instradamento. Tale *signal* è stato collegato agli ingressi del multiplexer 16:1 nella rispettiva *port map*.

1.3.2 Sintesi

Durante la progettazione si effettua un “ciclo di sviluppo del VHDL”. In tale loop di progettazione, dopo la fase di simulazione funzionale si passa alla fase di sintesi. Infatti, dopo aver testato opportunamente il sistema per verificarne il suo comportamento funzionale ed averne assicurato la correttezza, si esegue - tramite la sintesi - una trasformazione di una descrizione ad alto livello ad una di livello inferiore più vicina ad una descrizione dell’implementazione. È necessario distinguere ciò che è simulabile da ciò che è sintetizzabile. Non è detto, infatti, che dopo aver effettuato la simulazione di un sistema (anche se terminata con successo), tale sistema sia sintetizzabile. In simulazione può bastare anche un livello di descrizione alto ma per sintetizzare occorre un livello di descrizione più basso affinché il compilatore possa eseguire correttamente le operazioni di sintesi. Quindi, più è precisa la descrizione della macchina, maggiore sarà la probabilità che la sintesi vada a buon fine e che, dunque, il sistema fisico lavori come quello logico.

Il risultato della fase di sintesi consiste nel tradurre il circuito in una netlist che può essere ottimizzata tramite l’utilizzo di strumenti CAD.

A differenza della fase di simulazione, durante la sintesi non andiamo a descrivere nessuno stimolo in quanto ovviamente la macchina è alimentata da segnali reali.

Per mappare i segnali di entrata e di uscita della top-module fisicamente sulla scheda, si utilizza il file di Xilinx Design Constraints (XDC).

Per usare tale file nel progetto, bisogna:

- decommentare le linee corrispondenti ai pin da utilizzare;
- rinominare le porte utilizzate coi segnali della top-module del sistema descritto.

Per una maggiore semplicità di lettura, si riportano di seguito le sole linee “decommentate” dal file di constraints. Si utilizzano i due switch più a destra per i segnali di selezione del demultiplexer ed i quattro switch più a sinistra per i segnali di selezione del multiplexer. L’uscita selezionata è visualizzabile tramite i quattro led più a destra presenti sulla scheda.

```
##Switches
#Demux
set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 } [get_ports {
abilitazioni_demux[0] }]; #IO_L24N_T3_RS0_15 Sch=sw[0]
set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 } [get_ports {
abilitazioni_demux[1] }]; #IO_L3N_T0_DQS_EMCCLK_14 Sch=sw[1]
#Mux
set_property -dict { PACKAGE_PIN H6    IOSTANDARD LVCMOS33 } [get_ports {
abilitazioni_mux[0] }]; #IO_L24P_T3_35 Sch=sw[12]
set_property -dict { PACKAGE_PIN U12   IOSTANDARD LVCMOS33 } [get_ports {
abilitazioni_mux[1] }]; #IO_L20P_T3_A08_D24_14 Sch=sw[13]
set_property -dict { PACKAGE_PIN U11   IOSTANDARD LVCMOS33 } [get_ports {
abilitazioni_mux[2] }]; #IO_L19N_T3_A09_D25_VREF_14 Sch=sw[14]
set_property -dict { PACKAGE_PIN V10   IOSTANDARD LVCMOS33 } [get_ports {
abilitazioni_mux[3] }]; #IO_L21P_T3_DQS_14 Sch=sw[15]

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

Capitolo 2

Encoder

L'encoder è - come il multiplexer ed il demultiplexer - una macchina combinatoria per l'instradamento di segnali e la connessione tra sistemi. In particolare, l'encoder è una macchina dotata di *m* ingressi ed *n* uscite. Un vantaggio dell'operazione di codifica effettuata tramite tale macchina è che il numero di uscite è tipicamente minore degli ingressi. Un elevato numero di fili, infatti, può comportare costi elevati e mostrare maggiormente "il fianco" a possibili guasti e malfunzionamenti del sistema stesso. Proprio per questo introduciamo tali tecniche di codifica. Allo stesso tempo, però, codificare "lato mittente" spesso implica decodificare "lato destinazione" che, a sua volta, rappresenta un costo. L'operazione di decodifica viene effettuata tramite il decoder.

2.1 Encoder BCD

Progettare, implementare in VHDL e testare mediante simulazione una rete che, data in ingresso una stringa binaria X di 10 bit X9 X8 X7 X6 X5 X4 X3 X2 X1 X0 che corrisponde alla rappresentazione decodificata di una cifra decimale (cioè, una rappresentazione in cui ogni stringa contiene un solo bit alto), fornisce in uscita la rappresentazione Y della cifra mediante codifica Binary-Coded Decimal (BCD).

2.1.1 Soluzione

La traccia prevede in ingresso un segnale rappresentato su 10 bit con un unico bit alto ed in uscita la sua rappresentazione codificata in BCD.

La codifica BCD prevede che una singola cifra sia rappresentata su 4 bit e costituisce il nostro segnale di uscita. Tale segnale di 4 bit ci permette di rappresentare al più $2^4 = 16$ valori e quindi, inevitabilmente, alcune combinazioni di uscita non saranno

mai utilizzate. Abbiamo deciso, quindi, di utilizzare una di queste come *"combinazione di errore"*. Ad esempio, il caso in cui il segnale di ingresso dovesse presentare più di un bit alto costituisce una situazione di errore in quanto non valida per la nostra codifica. In tal caso, il sistema restituirà in uscita il valore "1111".

Tale situazione poteva essere gestita anche in altri modi. Ad esempio, avremmo potuto realizzare un arbitro per effettuare la scelta del bit da considerare assegnando una priorità ad ogni bit del segnale di input.

Abbiamo deciso, dunque, di non effettuare ipotesi di priorità in caso ci sia ambiguità nel valore di input bensì di segnalare semplicemente tale condizione.

2.1.2 Realizzazione

Riportiamo di seguito lo schematico dell'encoder BCD realizzato che, in questo punto dell'esercizio, costituisce l'unico componente del sistema e, dunque, ne è anche la top-module.

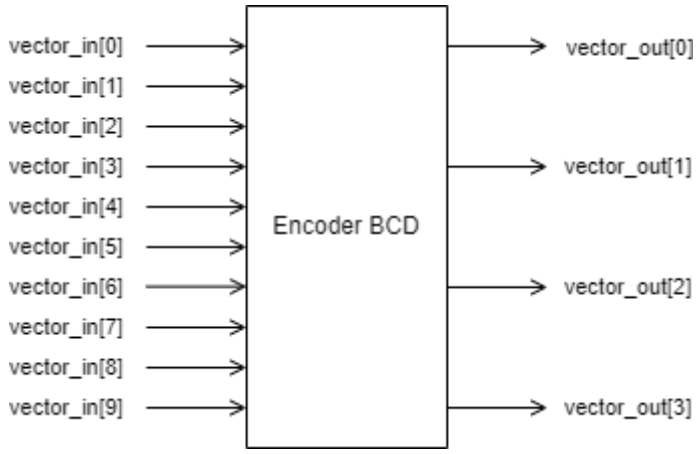


Figura 2.1 - Encoder BCD

Riportiamo, infine, il codice VHDL della componente.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY encoder_bcd IS
       PORT
       (
              vector_in  : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
```

```
            vector_out : OUT STD_LOGIC_VECTOR (3 DOWNTO 0)
      );
END encoder_bcd;

ARCHITECTURE Dataflow OF encoder_bcd IS

BEGIN
      vector_out <=
            "0000" WHEN vector_in = "0000000001" ELSE --0
            "0001" WHEN vector_in = "0000000010" ELSE --1
            "0010" WHEN vector_in = "0000000100" ELSE --2
            "0011" WHEN vector_in = "0000001000" ELSE --3
            "0100" WHEN vector_in = "0000010000" ELSE --4
            "0101" WHEN vector_in = "0000100000" ELSE --5
            "0110" WHEN vector_in = "0001000000" ELSE --6
            "0111" WHEN vector_in = "0010000000" ELSE --7
            "1000" WHEN vector_in = "0100000000" ELSE --8
            "1001" WHEN vector_in = "1000000000" ELSE --9
            "1111"; --errore
END Dataflow;
```

Come possiamo notare, in questo caso abbiamo scelto un approccio di tipo "*dataflow*" in quanto possiamo direttamente esplicitare l'output da mostrare (*vector_out*) in base all'input presentato (*vector_in*). Inoltre, abbiamo scelto di gestire in questo modo - come spiegato sopra - anche il caso in cui segnalare la presenza di eventuali errori dell'input ed evitare di dover gestire situazioni di priorità tramite, ad esempio, un arbitro. Nel caso in cui, infatti, *vector_in* dovesse presentare due o più bit alti, l'output mostrato è "1111", presente nell'ultima condizione *else*, utilizzata proprio per segnalare ed eventualmente gestire tali situazioni impreviste.

2.1.3 Simulazione

A 50 ns l'ingresso diventa "1000000000" ed è alto il solo il bit più significativo. La rete riconosce la sequenza e restituisce il valore 9. A 70 ns l'ingresso varia e di conseguenza l'uscita diventa 7. Il caso più interessante viene mostrato a 90 ns dove si presenta un'ambiguità. Di conseguenza, riscontrando un ingresso con più di un bit alto (1110000000) la rete di codifica restituisce l'uscita "1111" per segnalare tale ambiguità.

```
stim_proc : PROCESS
BEGIN
      WAIT FOR 50ns;
      vector_in <= ('1', OTHERS => '0');
      WAIT FOR 20ns;
```

```
        vector_in <= "0010000000";
        WAIT FOR 20ns;
        vector_in <= "1110000000";
        WAIT;
END PROCESS;
```

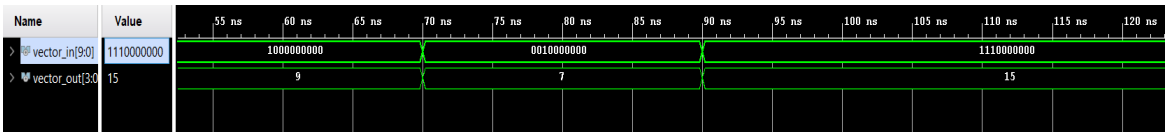


Figura 2.2 - Grafico della simulazione

2.2 Encoder BCD su FPGA (visualizzazione tramite LED)

Sintetizzare ed implementare su board il progetto dell'encoder BCD utilizzando gli switch per fornire la stringa X in ingresso, e i led per visualizzare Y.

2.2.1 Soluzione

A differenza del precedente esercizio, in questo caso non abbiamo avuto particolari problemi di natura fisica in quanto la nostra scheda Nexys A7-50T è dotata di 16 switch, permettendoci, dunque, di poter associare ognuno dei 10 bit che costituiscono il nostro ingresso ad uno switch corrispondente. I 4 bit relativi all'uscita, invece, vengono associati ai quattro led più a destra della scheda.

2.2.2 Sintesi

Si mappa il segnale d'ingresso su 10 switch della scheda ed il segnale di uscita sui 4 led più a destra della scheda.

Di seguito riportiamo le righe decommentate dal file di constraints.

```
##Switches
set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 } [get_ports {
vector_in[0] }]; #IO_L24N_T3_RS0_15 Sch=sw[0]
set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 } [get_ports {
vector_in[1] }]; #IO_L3N_T0_DQS_EMCCLK_14 Sch=sw[1]
set_property -dict { PACKAGE_PIN M13   IOSTANDARD LVCMOS33 } [get_ports {
vector_in[2] }]; #IO_L6N_T0_D08_VREF_14 Sch=sw[2]
set_property -dict { PACKAGE_PIN R15   IOSTANDARD LVCMOS33 } [get_ports {
vector_in[3] }]; #IO_L13N_T2_MRCC_14 Sch=sw[3]
set_property -dict { PACKAGE_PIN R17   IOSTANDARD LVCMOS33 } [get_ports {
vector_in[4] }]; #IO_L12N_T1_MRCC_14 Sch=sw[4]
```

```
set_property -dict { PACKAGE_PIN T18   IOSTANDARD LVCMOS33 } [get_ports {
vector_in[5] }]; #IO_L7N_T1_D10_14 Sch=sw[5]
set_property -dict { PACKAGE_PIN U18   IOSTANDARD LVCMOS33 } [get_ports {
vector_in[6] }]; #IO_L17N_T2_A13_D29_14 Sch=sw[6]
set_property -dict { PACKAGE_PIN R13   IOSTANDARD LVCMOS33 } [get_ports {
vector_in[7] }]; #IO_L5N_T0_D07_14 Sch=sw[7]
set_property -dict { PACKAGE_PIN T8    IOSTANDARD LVCMOS18 } [get_ports {
vector_in[8] }]; #IO_L24N_T3_34 Sch=sw[8]
set_property -dict { PACKAGE_PIN U8    IOSTANDARD LVCMOS18 } [get_ports {
vector_in[9] }]; #IO_25_34 Sch=sw[9]

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

2.3 Encoder BCD su FPGA (visualizzazione tramite display)

Utilizzare un display a 7 segmenti per visualizzare la cifra decimale codificata da Y (pilotare opportunamente i catodi del display per visualizzare la cifra).

2.3.1 Soluzione

La gestione del display a 7 segmenti viene effettuata tramite l'utilizzo di anodi e catodi. L'anodo indica quale cifra deve essere illuminata. Il catodo illumina i segmenti della cifra indicata dall'anodo.

Il catodo, a differenza dell'anodo, è comune a tutti i display e quindi non sarebbe possibile visualizzare cifre diverse nello stesso istante. I vari segmenti di un digit del display vengono illuminati tramite l'opportuna gestione dei bit. Ad ogni bit corrisponde un segmento. In più, il bit più significativo è associato al "dot". Di seguito riportiamo la gestione dei bit associati ai catodi: in particolare, *"a"* rappresenta il bit meno significativo ed *"h"* quello più significativo.

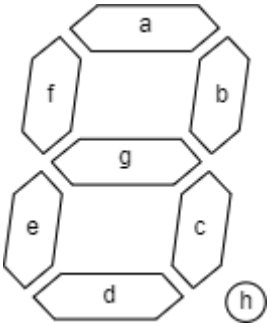


Figura 2.3 - Gestione catodi

2.3.2 Realizzazione

Per la scrittura del codice VHDL ci siamo serviti di una serie di costanti in modo da poter associare ad ognuna i corrispondenti valori dei bit in base ai catodi del display che vogliamo illuminare. Notiamo che:

- un bit basso corrisponde all'accensione del corrispondente LED del catodo;
- un bit alto corrisponde allo spegnimento del corrispondente LED del catodo.

Notiamo, inoltre, che oltre alle cifre da 0 a 9 abbiamo utilizzato un'ulteriore costante che abbiamo chiamato *"undefined"* per la rappresentazione del caso in cui ci sia ambiguità relativa al numero di bit alti in ingresso. In questo caso, i catodi vengono alterati affinché l'uscita non sia un valore numerico bensì "U" (undefined). Infine, un'ulteriore costante "*an_value*" viene definita per l'opportuna gestione dell'unico anodo che deve essere attivato.

Di seguito riportiamo il codice VHDL appena descritto.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY seven_segment_display IS
	PORT
	(
		number   : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
		anodes   : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
		cathodes : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
	);
END seven_segment_display;

ARCHITECTURE Behavioral OF seven_segment_display IS
	CONSTANT zero      : STD_LOGIC_VECTOR(7 DOWNTO 0) := "11000000";
	CONSTANT one       : STD_LOGIC_VECTOR(7 DOWNTO 0) := "11111001";
	CONSTANT two       : STD_LOGIC_VECTOR(7 DOWNTO 0) := "10100100";
	CONSTANT three     : STD_LOGIC_VECTOR(7 DOWNTO 0) := "10110000";
```

```
        CONSTANT four      : STD_LOGIC_VECTOR(7 DOWNTO 0) := "10011001";
        CONSTANT five      : STD_LOGIC_VECTOR(7 DOWNTO 0) := "10010010";
        CONSTANT six       : STD_LOGIC_VECTOR(7 DOWNTO 0) := "10000010";
        CONSTANT seven     : STD_LOGIC_VECTOR(7 DOWNTO 0) := "11111000";
        CONSTANT eight     : STD_LOGIC_VECTOR(7 DOWNTO 0) := "10000000";
        CONSTANT nine      : STD_LOGIC_VECTOR(7 DOWNTO 0) := "10010000";
        CONSTANT undefined : STD_LOGIC_VECTOR(7 DOWNTO 0) := "11000001";
        CONSTANT an_value  : STD_LOGIC_VECTOR(7 DOWNTO 0) := "11111110";
BEGIN
        PROCESS (number)
        BEGIN
                anodes <= an_value;
                CASE number IS
                        WHEN "0000" => cathodes <= zero;
                        WHEN "0001" => cathodes <= one;
                        WHEN "0010" => cathodes <= two;
                        WHEN "0011" => cathodes <= three;
                        WHEN "0100" => cathodes <= four;
                        WHEN "0101" => cathodes <= five;
                        WHEN "0110" => cathodes <= six;
                        WHEN "0111" => cathodes <= seven;
                        WHEN "1000" => cathodes <= eight;
                        WHEN "1001" => cathodes <= nine;
                        WHEN OTHERS => cathodes <= undefined;
                END CASE;
        END PROCESS;
END Behavioral;
```

All'interno del *process* (sensibile alla commutazione del valore in ingresso *"number"*) avviene l'assegnazione dei valori dei bit ai catodi ed all'anodo.

L'ultimo componente da descrivere è la top-module che permette di collegare l'encoder BCD precedentemente realizzato con il display a 7 segmenti appena descritto. Innanzitutto, riportiamo di seguito lo schematico.

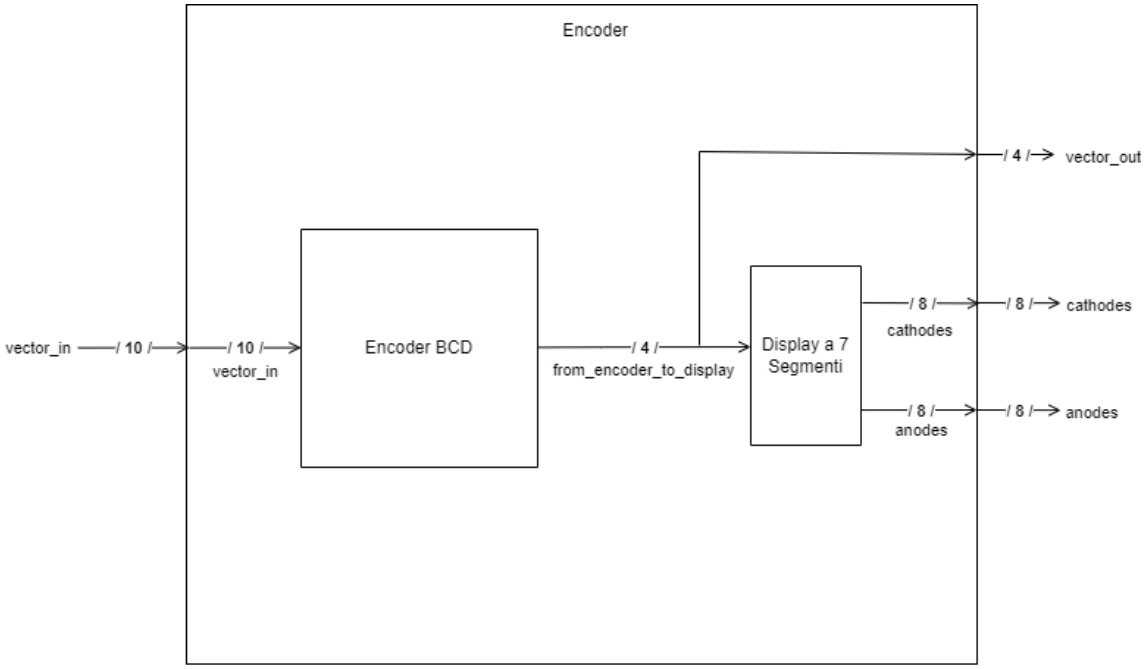


Figura 2.4 - Top-module

L'*"encoder_BCD"* progettato precedentemente viene riusato all'interno di un modulo più esterno insieme al modulo *"seven_segment_display"*.

L'uscita dell'encoder BCD viene collegata con un *signal* all'ingresso del display a 7 segmenti (*fromEncoderToDisplay*) e in uscita alla top-module per essere mostrato anche sui LED della scheda come richiesto dal punto precedente.

Di seguito riportiamo l'implementazione VHDL della top-module:

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY encoder IS
	PORT
	(
		vector_in  : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
		vector_out : OUT STD_LOGIC_VECTOR (3 DOWNTO 0);
		anodes     : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
		cathodes   : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
	);
END encoder;

ARCHITECTURE Structural OF encoder IS

	COMPONENT encoder_bcd
		PORT
		(
			vector_in  : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
```

```
                vector_out : OUT STD_LOGIC_VECTOR (3 DOWNTO 0)
            );
        END COMPONENT;

        COMPONENT seven_segment_display
            PORT
            (
                number   : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
                anodes   : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
                cathodes : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
            );
        END COMPONENT;

        SIGNAL fromEncoderToDisplay : STD_LOGIC_VECTOR(3 DOWNTO 0);

BEGIN
        bcd : encoder_bcd
        PORT MAP
        (
            vector_in  => vector_in(9 DOWNTO 0),
            vector_out => fromEncoderToDisplay
        );

        display : seven_segment_display
        PORT MAP
        (
            number   => fromEncoderToDisplay,
            anodes   => anodes,
            cathodes => cathodes
        );

        vector_out <= fromEncoderToDisplay;

END Structural;
```

2.3.3 Sintesi

Il digit della scheda a 7 segmenti può rappresentare senza problemi tutti i valori da 0 a 9 modificando opportunamente i catodi quando gli switch alterano l'ingresso della macchina, oltre al valore "U" precedentemente descritto.

L'anodo, invece, fa riferimento sempre alla stessa cifra durante tutta l'esecuzione del programma e quindi è fissato. Rispetto agli esercizi successivi dove si utilizza il display, non necessitiamo di gestire la variazione nel tempo degli anodi visto che la cifra da rappresentare è soltanto una.

Di seguito riportiamo, come al solito, il codice delle constraints.

```
##Switches
set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 } [get_ports {
vector_in[0] }]; #IO_L24N_T3_RS0_15 Sch=sw[0]
set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 } [get_ports {
vector_in[1] }]; #IO_L3N_T0_DQS_EMCCLK_14 Sch=sw[1]
set_property -dict { PACKAGE_PIN M13   IOSTANDARD LVCMOS33 } [get_ports {
vector_in[2] }]; #IO_L6N_T0_D08_VREF_14 Sch=sw[2]
set_property -dict { PACKAGE_PIN R15   IOSTANDARD LVCMOS33 } [get_ports {
vector_in[3] }]; #IO_L13N_T2_MRCC_14 Sch=sw[3]
set_property -dict { PACKAGE_PIN R17   IOSTANDARD LVCMOS33 } [get_ports {
vector_in[4] }]; #IO_L12N_T1_MRCC_14 Sch=sw[4]
set_property -dict { PACKAGE_PIN T18   IOSTANDARD LVCMOS33 } [get_ports {
vector_in[5] }]; #IO_L7N_T1_D10_14 Sch=sw[5]
set_property -dict { PACKAGE_PIN U18   IOSTANDARD LVCMOS33 } [get_ports {
vector_in[6] }]; #IO_L17N_T2_A13_D29_14 Sch=sw[6]
set_property -dict { PACKAGE_PIN R13   IOSTANDARD LVCMOS33 } [get_ports {
vector_in[7] }]; #IO_L5N_T0_D07_14 Sch=sw[7]
set_property -dict { PACKAGE_PIN T8    IOSTANDARD LVCMOS18 } [get_ports {
vector_in[8] }]; #IO_L24N_T3_34 Sch=sw[8]
set_property -dict { PACKAGE_PIN U8    IOSTANDARD LVCMOS18 } [get_ports {
vector_in[9] }]; #IO_25_34 Sch=sw[9]

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

##7 segment display
set_property -dict { PACKAGE_PIN T10   IOSTANDARD LVCMOS33 } [get_ports {
cathodes[0] }]; #IO_L24N_T3_A00_D16_14 Sch=ca
set_property -dict { PACKAGE_PIN R10   IOSTANDARD LVCMOS33 } [get_ports {
cathodes[1] }]; #IO_25_14 Sch=cb
set_property -dict { PACKAGE_PIN K16   IOSTANDARD LVCMOS33 } [get_ports {
cathodes[2] }]; #IO_25_15 Sch=cc
set_property -dict { PACKAGE_PIN K13   IOSTANDARD LVCMOS33 } [get_ports {
cathodes[3] }]; #IO_L17P_T2_A26_15 Sch=cd
set_property -dict { PACKAGE_PIN P15   IOSTANDARD LVCMOS33 } [get_ports {
cathodes[4] }]; #IO_L13P_T2_MRCC_14 Sch=ce
set_property -dict { PACKAGE_PIN T11   IOSTANDARD LVCMOS33 } [get_ports {
cathodes[5] }]; #IO_L19P_T3_A10_D26_14 Sch=cf
set_property -dict { PACKAGE_PIN L18   IOSTANDARD LVCMOS33 } [get_ports {
cathodes[6] }]; #IO_L4P_T0_D04_14 Sch=cg
set_property -dict { PACKAGE_PIN H15   IOSTANDARD LVCMOS33 } [get_ports {
cathodes[7] }]; #IO_L19N_T3_A21_VREF_15 Sch=dp
set_property -dict { PACKAGE_PIN J17   IOSTANDARD LVCMOS33 } [get_ports {
anodes[0] }]; #IO_L23P_T3_FOE_B_15 Sch=an[0]
set_property -dict { PACKAGE_PIN J18   IOSTANDARD LVCMOS33 } [get_ports {
anodes[1] }]; #IO_L23N_T3_FWE_B_15 Sch=an[1]
set_property -dict { PACKAGE_PIN T9    IOSTANDARD LVCMOS33 } [get_ports {
anodes[2] }]; #IO_L24P_T3_A01_D17_14 Sch=an[2]
set_property -dict { PACKAGE_PIN J14   IOSTANDARD LVCMOS33 } [get_ports {
```

```
anodes[3] }]; #IO_L19P_T3_A22_15 Sch=an[3]
set_property -dict { PACKAGE_PIN P14   IOSTANDARD LVCMOS33 } [get_ports {
anodes[4] }]; #IO_L8N_T1_D12_14 Sch=an[4]
set_property -dict { PACKAGE_PIN T14   IOSTANDARD LVCMOS33 } [get_ports {
anodes[5] }]; #IO_L14P_T2_SRCC_14 Sch=an[5]
set_property -dict { PACKAGE_PIN K2    IOSTANDARD LVCMOS33 } [get_ports {
anodes[6] }]; #IO_L23P_T3_35 Sch=an[6]
set_property -dict { PACKAGE_PIN U13   IOSTANDARD LVCMOS33 } [get_ports {
anodes[7] }]; #IO_L23N_T3_A02_D18_14 Sch=an[7]
```

Capitolo 3

Riconoscitore di sequenze

Prima di poter procedere con la descrizione della nostra implementazione del riconoscitore di sequenze è necessario introdurre il concetto di macchina sequenziale. A differenza delle macchine combinatorie precedentemente descritte in cui l'uscita dipende solo dall'ingresso, le macchine sequenziali sono macchine la cui uscita dipende sia dall'ingresso che dallo stato. Le macchine sequenziali, dunque, sono macchina che hanno memoria. Ogni stato rappresenta l'evoluzione di questo tipo di macchine. Il modello matematico che consente di descrivere una macchina sequenziale ed i suoi stati è l'automa.

Ricordiamo che l'automa a stati finiti è descritto dalla quintupla <Q, I, U, t, w> con:

- Q → insieme finito degli stati interni;
- I → insieme finito degli ingressi;
- U → insieme finito delle uscite;
- t → funzione di transizione definita come $QxI \rightarrow Q$;
- w → funzione di uscita.

Il tipo di funzione di uscita ci permette di distinguere automi di Mealy e di Moore.

Nel caso degli automi di Mealy si rappresenta la variazione dell'uscita a fronte di ingresso e stato (w: $QxI \rightarrow U$).

Nel caso degli automi di Moore, invece, l'uscita dipende solo dallo stato (w: $Q \rightarrow U$). In questo caso, dunque, l'ingresso modifica lo stato e l'uscita è allora espressa in funzione dello stato stesso.

Il costrutto fondamentale per descrivere un automa è il *process*. Il modello tipicamente utilizzato per descrivere un automa prevede *due process*, uno puramente combinatorio

che calcola la funzione stato prossimo e la funzione di uscita e l'altro che rappresenta l'elemento di memoria che consente di retroazionare lo stato. Notiamo che nella descrizione di un automa a stati finiti non è previsto il concetto di *"tempo"*: la tempificazione della macchina va realizzata direttamente all'interno del progetto.

Una macchina sequenziale può essere ulteriormente classificata in **sincrona** e **asincrona**.
La macchina sincrona evolve in corrispondenza di un segnale di abilitazione esterno detto *"di sincronismo"*, che indica quando deve essere prelevato il valore significativo di un dato in ingresso. Il segnale di sincronismo può essere un segnale periodico (clock).
La macchina asincrona, invece, è una macchina instrinsecamente stabile perché in corrispondenza di una variazione del segnale (e non del suo valore) lo stato corrente (stabile) evolve in un nuovo stato (stabile).
Le macchine sequenziali, dunque, possono essere asincrone, intrinsecamente stabili poiché partono da uno stato stabile ed arrivano sempre in uno stato stabile, oppure possono essere sincrone, cioè macchine per cui questa proprietà di stabilità non è verificata per ogni ingresso e per ogni stato.

3.1 Riconoscitore

Progettare, implementare in VHDL e testare mediante simulazione una macchina in grado di riconoscere la sequenza **1001**. La macchina prende in ingresso un segnale binario *i* che rappresenta il dato, un segnale A di tempificazione 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 4,
- 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.

3.1.1 Soluzione

Per riconoscere una stringa binaria proveniente da un canale seriale temporizzato sul clock, è necessario avere una memoria che mantenga lo stato. Il controllo del modo può essere effettuato:

- solo quando si alza reset (la variazione del modo durante l'esecuzione del programma non varia il suo funzionamento);
- in qualsiasi stadio dell'automa (la variazione del modo durante l'esecuzione del programma varia il suo funzionamento).

Nel nostro progetto, abbiamo deciso di optare per il secondo caso per rendere la macchina più flessibile, nonostante complichi maggiormente la logica rispetto al primo caso. In ogni stadio dell'automa, quindi, al variare di M la macchina ritorna allo stato iniziale per poi acquisire i nuovi valori.

Considerando che M può variare in ogni momento, ci sarà un controllo tempificato sul clock per verificare se il valore di M è effettivamente cambiato rispetto al colpo di clock precedente.

La macchina, quindi, necessita di catturare l'ingresso in quell'istante e, alla fine, restituire 1 se la sequenza è riconosciuta in base al modo scelto.

Abbiamo ritenuto opportuno, come buona norma, aggiungere anche un segnale di reset.

3.1.2 Realizzazione

Per eseguire il corretto riconoscimento della stringa binaria abbiamo descritto i possibili stati della macchina tramite un automa di Moore che - per facilitare la lettura - riportiamo in più parti. La prima parte che mostriamo è quella di selezione del modo, presentata di seguito:

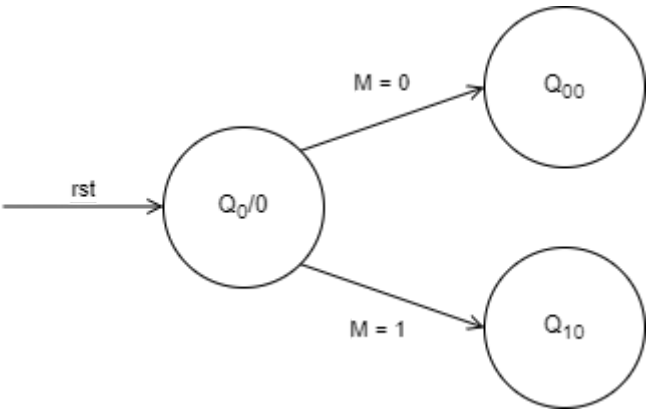


Figura 2.4 - Selezione del modo M

Se la macchina viene portata nello stato q_{00} tramite il valore del modo uguale a 0, l'automa a stati finiti che abbiamo realizzato prosegue come in figura:

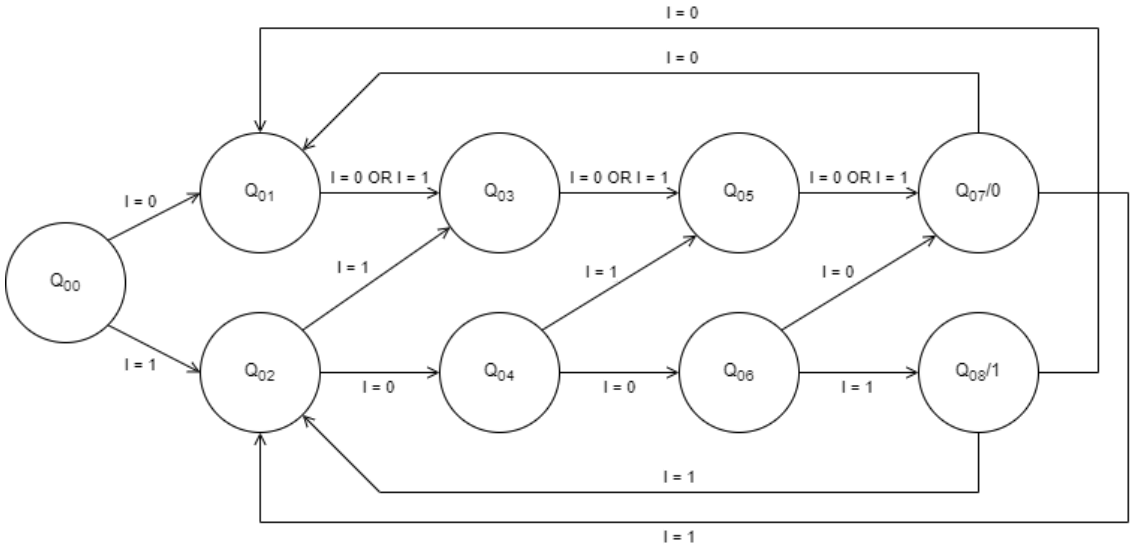


Figura 2.5 - Automa a stati finiti del modo M=0

Notiamo che in questo caso la macchina prosegue l'esecuzione dallo stato q_{00} precedentemente selezionato. Siccome il sistema dovrà riconoscere la sequenza di bit "1001" a gruppi di quattro, ogni stato dovrà necessariamente dipendere dal precedente bit rilevato. Se il primo bit rilevato dovesse essere uno 0, dovremo aspettare i successivi 3 bit prima di poter rilevare una nuova sequenza,

indipendentemente dal loro valore. Se il primo bit, invece, dovesse essere un 1, allora la macchina evolverebbe negli stati q_{04}, q_{06} e q_{08} se la sequenza dovesse essere rilevata correttamente. In caso contrario, in qualsiasi momento, se dovesse arrivare un valore diverso dalla sequenza cercata, allora la macchina evolverebbe in uno degli stati q_{03}, q_{05} o q_{07} (a seconda della sequenza correttamente rilevata fino a quel momento). Prima di poter riconoscere una nuova sequenza, dunque, sarà necessario giungere nello stato q_{07} o q_{08}. Nel primo caso, il bit di uscita sarà basso, indicando che la sequenza rilevata non corrisponde a quella cercata. Nel secondo caso, invece, il bit di uscita sarà alto al fine di segnalare il corretto riconoscimento della sequenza cercata. Notiamo, inoltre, che il bit rilevato subito dopo gli stati q_{07} e q_{08} - a seconda del suo valore - porta la macchina negli stati q_{01} o q_{02}.

Di seguito riportiamo, invece, la parte dell'automa relativa al funzionamento della macchina con modo 1.

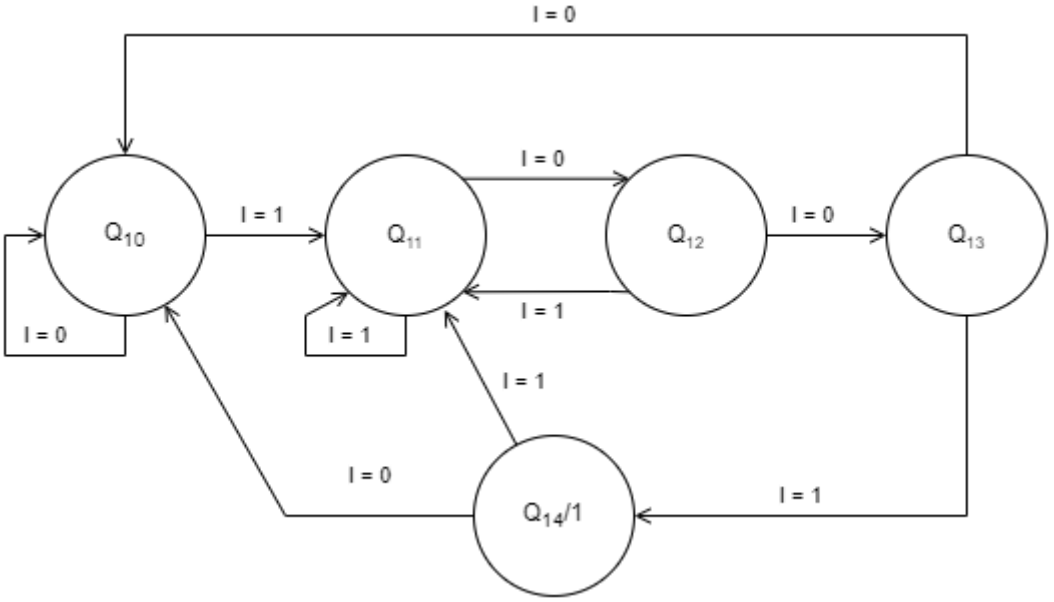


Figura 2.6 - Automa a stati finiti del modo M=1

A differenza del caso descritto precedentemente, la macchina deve tornare nello stato iniziale solo nel momento in cui la sequenza di bit cercata viene correttamente

riconosciuta. Dunque, la macchina parte dallo stato q_{10} e permane in tale stato finché non arriverà un bit alto. Una volta arrivata nello stato q_{11}, la macchina permarrà in tale stato finché non arriverà un bit basso. Una volta giunti nello stato q_{12} abbiamo effettuato la seguente scelta progettuale: la macchina evolverà nello stato q_{13} nel caso in cui dovesse presentarsi in ingresso un altro bit 0; in caso contrario, invece, la macchina tornerà nello stato q_{12} in modo tale che il bit 1 appena arrivato costituirà già l'inizio di una nuova possibile sequenza, evitando così l'invio di dati che poi rimarrebbero inutilizzati. Una volta giunti nello stato q_{13}, la macchina evolverà nello stato finale q_{14} nel caso in cui l'ingresso dovesse presentare un bit alto e, dunque, l'uscita della macchina sarà 1. In caso contrario, invece, la macchina tornerà in q_{10}. Una volta giunti nello stato q_{14}, il successivo bit alto in ingresso costituirà l'inizio di una nuova sequenza e porterà la macchina nello stato q_{11} mentre un eventuale bit basso in ingresso porterà la macchina nello stato q_{10}, in attesa dell'inizio di una nuova sequenza.

Terminata la descrizione dell'automa, riportiamo di seguito la sua implementazione VHDL.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY riconoscitore IS
	PORT
	(
		I      : IN STD_LOGIC;
		M      : IN STD_LOGIC;
		rst, A : IN STD_LOGIC;
		Y      : OUT STD_LOGIC
	);
END riconoscitore;

ARCHITECTURE Behavioral OF riconoscitore IS
	TYPE stato IS (q0, q00, q01, q02, q03, q04, q05, q06, q07, q08, q10,
q11, q12, q13, q14);
	SIGNAL current_state, next_state : stato;
BEGIN
	seq : PROCESS (A)
```

```
BEGIN
        IF (A'EVENT AND a = '1') THEN
                IF (rst = '1') THEN
                        current_state <= q0;
                ELSIF (M'quiet(10ns) = false AND M = '0') THEN
                        current_state <= q00;
                ELSIF (M'quiet(10ns) = false AND M = '1') THEN
                        current_state <= q10;
                ELSE
                        current_state <= next_state;
                END IF;
        END IF;
END PROCESS;

comb : PROCESS (current_state, M, I)
BEGIN
        CASE current_state IS
                WHEN q0 =>
                        Y <= '0';
                        IF (M = '0') THEN
                                next_state <= q00;
                        ELSIF (M = '1') THEN
                                next_state <= q10;
                        END IF;
                WHEN q00 =>
                        Y <= '0';
                        IF (I = '0') THEN
                                next_state <= q01;
                        ELSE
                                next_state <= q02;
                        END IF;
                WHEN q01 =>
                        Y          <= '0';
                        next_state <= q03;
                WHEN q02 =>
                        Y <= '0';
                        IF (I = '0') THEN
                                next_state      <= q04;
                        ELSE next_state <= q03;
                        END IF;
                WHEN q03 =>
                        Y          <= '0';
                        next_state <= q05;
                WHEN q04 =>
                        Y <= '0';
                        IF (I = '0') THEN
                                next_state      <= q06;
                        ELSE next_state <= q05;
                        END IF;
                WHEN q05 =>
                        Y          <= '0';
                        next_state <= q07;
                WHEN q06 =>
                        Y <= '0';
                        IF (I = '0') THEN
                                next_state      <= q07;
```

```
                        ELSE next_state <= q08;
                        END IF;
                WHEN q07 =>
                        Y <= '0';
                        IF (I = '0') THEN
                                next_state      <= q01;
                        ELSE next_state <= q02;
                        END IF;
                WHEN q08 =>
                        Y <= '1';
                        IF (I = '0') THEN
                                next_state      <= q01;
                        ELSE next_state <= q02;
                        END IF;
                WHEN q10 =>
                        Y <= '0';
                        IF (I = '0') THEN
                                next_state      <= current_state;
                        ELSE next_state <= q11;
                        END IF;
                WHEN q11 =>
                        Y <= '0';
                        IF (I = '0') THEN
                                next_state      <= q12;
                        ELSE next_state <= current_state;
                        END IF;
                WHEN q12 =>
                        Y <= '0';
                        IF (I = '0') THEN
                                next_state      <= q13;
                        ELSE next_state <= q11;
                        END IF;
                WHEN q13 =>
                        Y <= '0';
                        IF (I = '0') THEN
                                next_state      <= q10;
                        ELSE next_state <= q14;
                        END IF;
                WHEN q14 =>
                        Y <= '1';
                        IF (I = '0') THEN
                                next_state      <= q10;
                        ELSE next_state <= q11;
                        END IF;
                WHEN OTHERS =>
                        Y <= '-';
            END CASE;
        END PROCESS;
END Behavioral;
```

Notiamo l'utilizzo dei due *process*, come buona norma, in modo da distinguere il processo sequenziale per la gestione di clock e reset ed utilizzato per tornare nello stato iniziale q_0. Inoltre, utilizziamo tale processo per verificare eventuali cambiamenti

nel valore del modo che porteranno la macchina nei rispettivi stati q_{01} o q_{10}. Inoltre, troviamo il processo combinatorio per calcolare il prossimo stato e, in riferimento ad esso, l'uscita in modo da descrivere interamente l'automa di Moore progettato.

Notiamo che nel processo sequenziale, per conoscere quando varia M, abbiamo utilizzato l'attributo QUIET per 10 ns che è il tempo del clock.

3.1.3 Simulazione

A differenza delle simulazioni precedentemente mostrate, siccome la macchina realizzata è di tipo sequenziale e gestita anche tramite l'utilizzo di un clock, in questo caso abbiamo dovuto, allora, considerare di implementare nel testbench anche un *process* dedicato alla simulazione della forma d'onda del clock. Abbiamo, innanzitutto, dichiarato una *constant* che abbiamo chiamato *Clk_period* di tipo *time* ed inizializzata a 10 ns al fine di simulare l'esatta forma d'onda del clock presente sulla nostra board. Riportiamo, quindi, di seguito il codice del *process* realizzato.

```
clk_process: process
    begin
        A <= '0';
        wait for clk_period/2;
        A <= '1';
        wait for clk_period/2;
end process;
```

Di seguito riportiamo poi il codice del *process* relativo ai casi di test che abbiamo effettuato per verificare il corretto funzionamento del sistema.

```
stim_proc: PROCESS

      VARIABLE var  : STD_LOGIC_VECTOR (14 DOWNTO 0) := "100100111001001";
      VARIABLE var1 : STD_LOGIC_VECTOR (15 DOWNTO 0) :=
"0010010010011001";
      VARIABLE var2 : STD_LOGIC_VECTOR (15 DOWNTO 0) :=
"0010010010011001";
BEGIN
      WAIT FOR 100 ns;

      I <= '0';
      WAIT FOR 10 ns;
      I <= '1';
      WAIT FOR 10 ns;
      I <= '0';
```

```
        WAIT FOR 10 ns;
        I <= '1';
        WAIT FOR 10 ns;
        I <= '1';
        WAIT FOR 10 ns;
        I <= '0';
        WAIT FOR 10 ns;
        I <= '0';
        WAIT FOR 10 ns;
        I <= '1';
        WAIT FOR 10 ns;
        M <= '1';
        WAIT FOR 10 ns;
        FOR x IN 0 TO 14 LOOP
                I <= var(x);
                WAIT FOR 10 ns;
        END LOOP;

        FOR x IN 0 TO 15 LOOP
                IF (x = 3) THEN
                        M <= '0';
                END IF;
                I <= var1(x);
                WAIT FOR 10 ns;
        END LOOP;

        FOR x IN 0 TO 15 LOOP
                IF (x = 6) THEN
                        M <= '1';
                END IF;
                I <= var2(x);
                WAIT FOR 10 ns;
        END LOOP;
        WAIT;
END PROCESS;
```

Notiamo, innanzitutto, la definizione di tre *variable* che abbiamo poi successivamente utilizzato tramite il costrutto *for..loop* per assegnare in maniera seriale i valori che il segnale di ingresso deve assumere ad ogni colpo di clock. Nel secondo e nel terzo loop di questo tipo abbiamo voluto testare anche il comportamento della macchina nel caso in cui - durante una sequenza - il modo cambi improvvisamente.

Di seguito riportiamo i grafici con i risultati ottenuti.

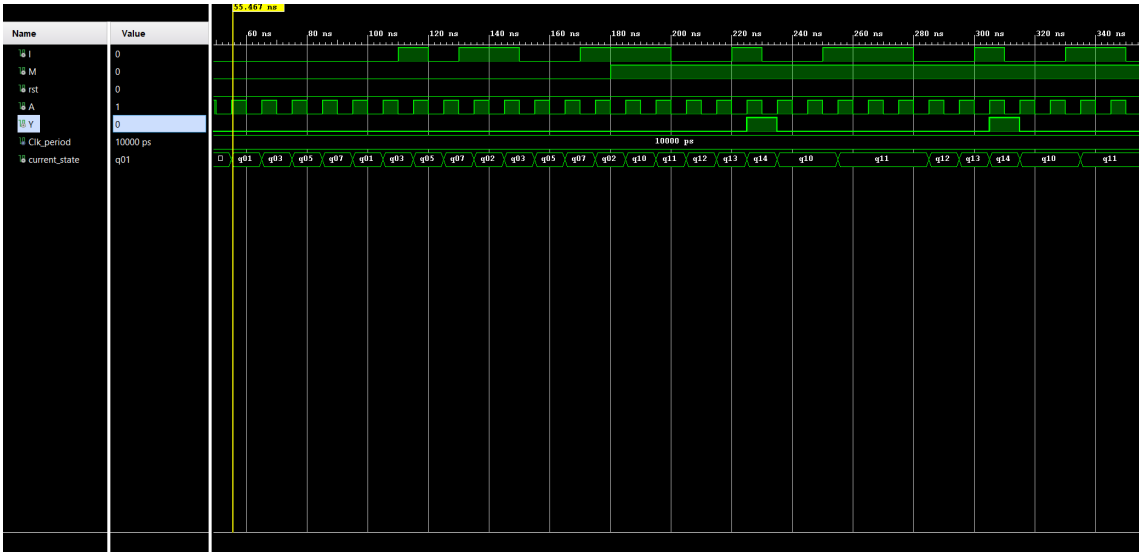


Figura 2.7 - Grafico 1

Osserviamo che, dopo aver atteso 100 ns per il passaggio dei transitori, i primi valori iniziano ad essere assegnati all'ingresso. Il modo inizialmente resta basso e, dunque, se il riconoscimento di una sequenza inizia, bisognerà attendere i 4 bit che la compongono prima di poter procedere con il riconoscimento di un'altra sequenza. Gli stati evolvono correttamente come precedentemente descritto.

A 180 ns il bit del modo viene alzato e, come spiegato, in tale modalità l'uscita si alzerà appena la sequenza ricercata viene correttamente trovata, anche se dovesse essere interrotta da bit non appartenenti alla sequenza cercata. A 225 ns, infatti, il bit di uscita *Y* si alza correttamente per un intero ciclo di clock. A 305 ns sarà riconosciuta poi un'altra sequenza.

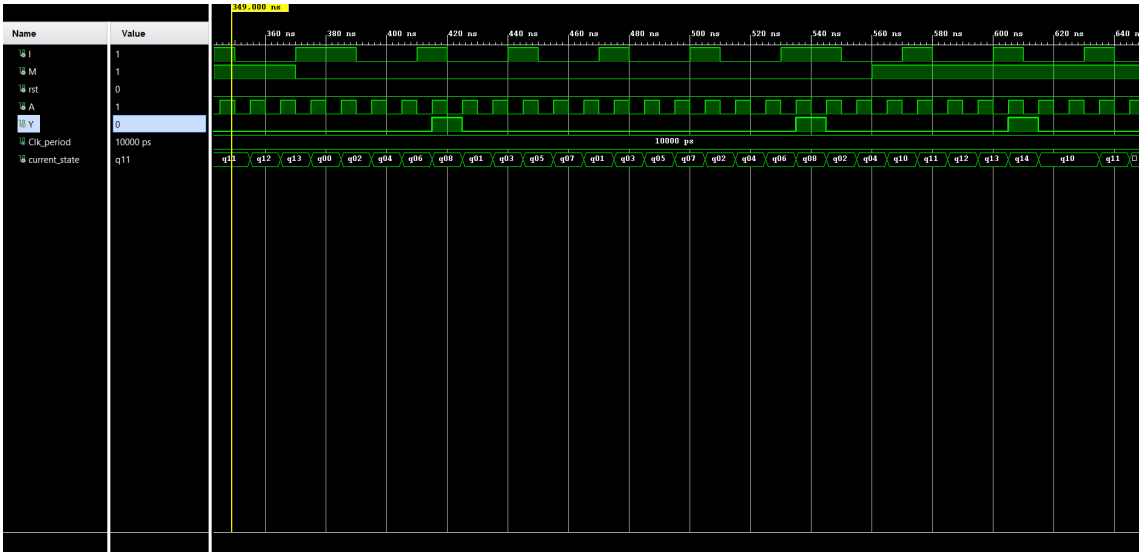


Figura 2.8 - Grafico 2

Notiamo, infine, che a 370 ns il modo viene nuovamente abbassato e da 385 ns arriva in ingresso esattamente la sequenza di bit “1001” che corrisponde a quella cercata e, dunque, l’uscita si alza correttamente. La stessa cosa avviene a 530 ns. In seguito il modo verrà alzato nuovamente e riconoscerà un’ultima volta la sequenza cercata.

3.2 Riconoscitore su FPGA

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 tempificazione A, che deve essere ottenuto a partire dal clock della board. Infine, l’uscita Y può essere codificata utilizzando un led.

3.2.1 Soluzione

Per poter sintetizzare su board il riconoscitore di sequenza precedentemente descritto abbiamo dovuto apportare una serie di modifiche. Andremo, dunque, a mostrare nuovamente la simulazione per evidenziare il comportamento logico di quanto realizzato.

Notiamo, innanzitutto, che modo ed ingresso vengono gestiti tramite switch e, dunque, non presentando un valore iniziale *undefined (U)* (lo switch è sempre impostato a 0 o 1) non possiamo passare da uno stato all'altro semplicemente cambiando il valore degli switch. Abbiamo, dunque, optato per definire due nuove porte (le *S1* ed *S2* della traccia da noi rinominate rispettivamente *catch_M* e *catch_I*). Tali porte sono poi state mappate su due bottoni presenti sulla scheda. Facendo ciò abbiamo necessariamente dovuto "aggiornare" l'automa a stati finiti descritto precedentemente per poter determinare opportunamente quando catturare il valore di modo ed ingresso presente sugli switch.

3.2.1 Realizzazione

Come spiegato nel paragrafo precedente, per la realizzazione della nostra soluzione avremo bisogno di ricorrere per la prima volta all'utilizzo dei bottoni presenti sulla scheda. Tale soluzione, però, presenta un problema: il bottone è influenzato da una parte transitoria prima e dopo la pressione. Questo crea delle criticità nelle macchine sequenziali che necessitano di un segnale di sincronismo. Tale pressione, infatti, causa una serie di oscillazioni che provocano rapide commutazioni dell'ingresso e, di conseguenza, nella nostra macchina a stati finiti provocherebbero molteplici cambiamenti di stato non desiderati.

Per far fronte a tale problema potremmo adottare due possibili approcci:

- una soluzione hardware caratterizzata dall'aggiunta di circuiti RC che facciano da filtro passabasso per calcolare le vibrazioni del bottone ed andando a tagliare le frequenze più alte;
- una soluzione software caratterizzata dal fatto che - dopo che il bottone è stato premuto - si attende il tempo necessario a far terminare le oscillazioni ed ottenere un segnale stabile.

Per ovvie ragioni di semplicità, adotteremo la seconda soluzione e, per questo, è necessario introdurre il concetto di debouncer. Il debouncer è un filtro digitale che riceve 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.

Tale soluzione viene implementata tramite il seguente automa a 2 stati:

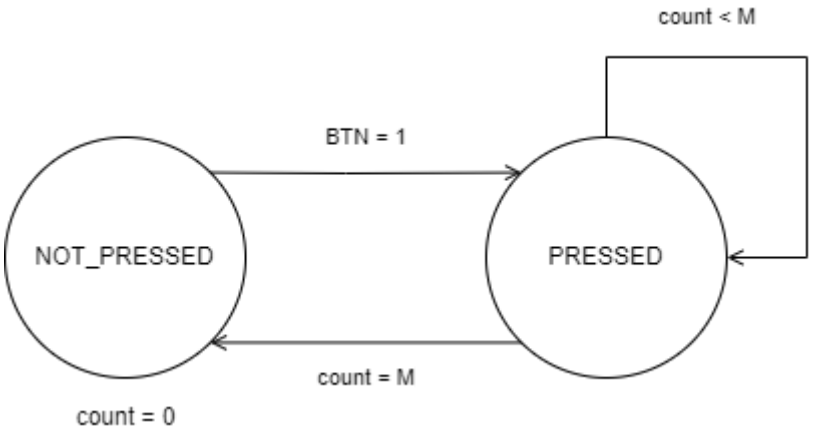


Figura 2.9 - Automa a stati finiti del debouncer

Lo stato iniziale è "*NOT_PRESSED*" e, appena si rileva la pressione del bottone (*BTN=1*), si va nello stato "*PRESSED*" dove si andrà ad incrementare il valore di una variabile *count* che conta il numero di fronti di clock. Quando *count=M,* l'oscillazione è stata superata e si ritornerà così allo stato "*NOT_PRESSED*".

Notiamo che la variabile *count* conta rispetto alla base dei tempi della macchina. Quindi questo non è un tempo generale bensì specifico per la singola scheda e dipende anche dalle caratteristiche fisiche del bottone presente sulla scheda stessa. Per questo utilizzeremo il costrutto *generics* per poter impostare di volta in volta il tempo necessario ad assicurare che il bottone non sia più premuto.

Riportiamo, quindi, di seguito il codice del debouncer implementato.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY ButtonDebouncer IS
	GENERIC
	(
		CLK_period     : INTEGER := 10; -- periodo del clock della
board 10 nanosecondi
		btn_noise_time : INTEGER := 650000000
	);
	PORT
	(
		RST          : IN STD_LOGIC;
```

```
            CLK         : IN STD_LOGIC;
            BTN         : IN STD_LOGIC;
            CLEARED_BTN : OUT STD_LOGIC
      );
END ButtonDebouncer;

ARCHITECTURE Behavioral OF ButtonDebouncer IS

      TYPE stato IS (NOT_PRESSED, PRESSED);
      SIGNAL BTN_state   : stato := NOT_PRESSED;

      CONSTANT max_count : INTEGER := btn_noise_time/CLK_period; --
6500000/10= conto 650000 colpi di clock

BEGIN
      deb            : PROCESS (CLK)
            VARIABLE count : INTEGER := 0;

      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 =>
                                    CLEARED_BTN <= '0';
                                    IF (BTN = '1') THEN
                                          BTN_state <= PRESSED;
                                    ELSE
                                          BTN_state <= NOT_PRESSED;
                                    END IF;
                              WHEN PRESSED =>
                                    IF (count = max_count - 1) THEN
                                          count := 0;
                                          CLEARED_BTN <= '1';
                                          BTN_state   <= NOT_PRESSED;
                                    ELSE
                                          count := count + 1;
                                          BTN_state <= PRESSED;
                                    END IF;
                              WHEN OTHERS =>
                                    BTN_state <= NOT_PRESSED;
                        END CASE;
                  END IF;
            END IF;
      END PROCESS;

END Behavioral;
```

Un'altra modifica effettuata rispetto all'implementazione precedentemente descritta è stata quella di rimuovere gli attributi "*Quiet*" del modo in quanto sono risultati non sintetizzabili.

Riportiamo, allora, il nuovo codice dell'automa a stati finiti "aggiornato".

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY riconoscitore IS
	PORT
	(
		I                 : IN STD_LOGIC;
		M                 : IN STD_LOGIC;
		rst, A            : IN STD_LOGIC;
		catch_I, catch_M  : IN std_logic;
		Y                 : OUT STD_LOGIC;
		led               : OUT std_logic_vector (3 DOWNTO 0)
	);
END riconoscitore;

ARCHITECTURE Behavioral OF riconoscitore IS
	TYPE stato IS (q0, q00, q01, q02, q03, q04, q05, q06, q07, q08, q10,
q11, q12, q13, q14);
	SIGNAL current_state, next_state : stato;
BEGIN
	seq : PROCESS (A)
	BEGIN
		IF (A'EVENT AND a = '1') THEN
			IF (rst = '1') THEN
				current_state <= q0;
			ELSIF (catch_M = '1') THEN
				IF (M = '0') THEN
					current_state <= q00;
				ELSE
					current_state <= q10;
				END IF;
			ELSE
				current_state <= next_state;
			END IF;
		END IF;
	END PROCESS;

	comb : PROCESS (current_state, catch_M, catch_I, M, I)
	BEGIN
		CASE current_state IS
			WHEN q0 =>
				Y   <= '0';
				led <= "1111";
				IF (catch_M = '1') THEN
					IF (M = '0') THEN
						next_state <= q00;
					ELSIF (M = '1') THEN
						next_state <= q10;
```

```
                END IF;
        END IF;
WHEN q00 =>
        led <= "0001";
        Y   <= '0';
        IF (catch_I = '1') THEN
                IF (I = '0') THEN
                        next_state <= q01;
                ELSE
                        next_state <= q02;
                END IF;
        END IF;
WHEN q01 =>
        Y   <= '0';
        led <= "0010";
        IF (catch_I = '1') THEN
                next_state <= q03;
        END IF;
WHEN q02 =>
        Y   <= '0';
        led <= "0011";
        IF (catch_I = '1') THEN
                IF (I = '0') THEN
                        next_state      <= q04;
                ELSE next_state <= q03;
                END IF;
        END IF;
WHEN q03 =>
        Y   <= '0';
        led <= "0100";
        IF (catch_I = '1') THEN
                next_state <= q05;
        END IF;
WHEN q04 =>
        Y   <= '0';
        led <= "0101";
        IF (catch_I = '1') THEN
                IF (I = '0') THEN
                        next_state      <= q06;
                ELSE next_state <= q05;
                END IF;
        END IF;
WHEN q05 =>
        Y   <= '0';
        led <= "0110";
        IF (catch_I = '1') THEN
                next_state <= q07;
        END IF;
WHEN q06 =>
        Y   <= '0';
        led <= "0111";
        IF (catch_I = '1') THEN
                IF (I = '0') THEN
                        next_state      <= q07;
                ELSE next_state <= q08;
                END IF;
```

```
                END IF;
            WHEN q07 =>
                Y   <= '0';
                led <= "1000";
                IF (catch_I = '1') THEN
                    IF (I = '0') THEN
                        next_state      <= q01;
                    ELSE next_state <= q02;
                    END IF;
                END IF;
            WHEN q08 =>
                Y   <= '1';
                led <= "1001";
                IF (catch_I = '1') THEN
                    IF (I = '0') THEN
                        next_state      <= q01;
                    ELSE next_state <= q02;
                    END IF;
                END IF;
            WHEN q10 =>
                Y   <= '0';
                led <= "1010";
                IF (catch_I = '1') THEN
                    IF (I = '0') THEN
                        next_state      <= current_state;
                    ELSE next_state <= q11;
                    END IF;
                END IF;
            WHEN q11 =>
                Y   <= '0';
                led <= "1011";
                IF (catch_I = '1') THEN
                    IF (I = '0') THEN
                        next_state      <= q12;
                    ELSE next_state <= current_state;
                    END IF;
                END IF;
            WHEN q12 =>
                Y   <= '0';
                led <= "1100";
                IF (catch_I = '1') THEN
                    IF (I = '0') THEN
                        next_state      <= q13;
                    ELSE next_state <= q11;
                    END IF;
                END IF;
            WHEN q13 =>
                Y   <= '0';
                led <= "1101";
                IF (catch_I = '1') THEN
                    IF (I = '0') THEN
                        next_state      <= q10;
                    ELSE next_state <= q14;
                    END IF;
                END IF;
            WHEN q14 =>
```

```
                        Y    <= '1';
                        led <= "1110";
                        IF (catch_I = '1') THEN
                              IF (I = '0') THEN
                                    next_state      <= q10;
                              ELSE next_state <= q11;
                              END IF;
                        END IF;
                  WHEN OTHERS =>
                        Y <= ' - ';
            END CASE;
      END PROCESS;
END Behavioral;
```

Anche in questo caso, abbiamo deciso di utilizzare due *process*, uno puramente combinatorio dedicato al calcolo dello stato prossimo e dell'uscita e l'altro sequenziale per l'utilizzo di clock e reset. Sottolineiamo, inoltre, l'aggiunta di una codifica dello stato in cui ci troviamo in modo da poterlo visualizzare sui LED presenti sulla scheda.

Infine, di seguito riportiamo lo schematico della top-module del sistema così realizzato.

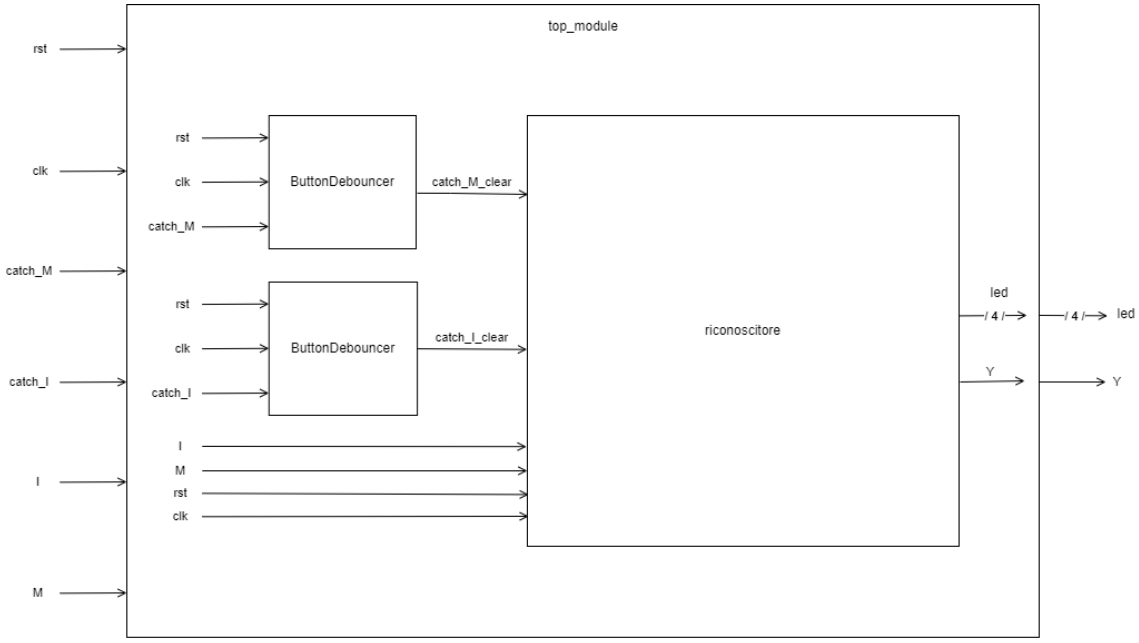


Figura 2.10 - Schematico della top-module

Notiamo la presenza di due debouncer per i due bottoni relativi a modo ed ingresso. I corrispondenti segnali ripuliti (*catch_M_clear* e *catch_I_clear*) escono dai debouncer e vengono portati in ingresso al riconoscitore. L'uscita della top-module permette di attivare i led per la visualizzazione dello stato in cui ci troviamo (come

precedentemente descritto) e la relativa uscita. Di seguito ne riportiamo l'implementazione VHDL.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY top_module IS
	PORT
	(
		I, M, catch_I, catch_M : IN std_logic;
		rst, clk               : IN std_logic;
		Y                      : OUT std_logic;
		led                    : OUT std_logic_vector (3 DOWNTO 0)
	);
END top_module;

ARCHITECTURE Behavioral OF top_module IS

	COMPONENT ButtonDebouncer IS
		GENERIC
		(
			CLK_period     : INTEGER := 10;
			btn_noise_time : INTEGER := 325000000
		);
		PORT
		(
			RST         : IN STD_LOGIC;
			CLK         : IN STD_LOGIC;
			BTN         : IN STD_LOGIC;
			CLEARED_BTN : OUT STD_LOGIC
		);
	END COMPONENT;

	COMPONENT riconoscitore IS
		PORT
		(
			I                : IN STD_LOGIC;
			M                : IN STD_LOGIC;
			rst, A           : IN STD_LOGIC;
			catch_I, catch_M : IN std_logic;
			Y                : OUT STD_LOGIC;
			led              : OUT std_logic_vector (3 DOWNTO 0)
		);
	END COMPONENT;

	SIGNAL catch_I_clear : std_logic;
	SIGNAL catch_M_clear : std_logic;

BEGIN
	DebI : ButtonDebouncer
		GENERIC MAP
		(
		CLK_period     => 10,
		btn_noise_time => 325000000
		)
```

```
            PORT MAP
            (
                  RST         => rst,
                  CLK         => clk,
                  BTN         => catch_I,
                  CLEARED_BTN => catch_I_clear
            );

                  DebM : ButtonDebouncer
                        GENERIC MAP
                        (
                        CLK_period     => 10,
                        btn_noise_time => 325000000
                        )
                        PORT MAP
                        (
                              RST         => rst,
                              CLK         => clk,
                              BTN         => catch_M,
                              CLEARED_BTN => catch_M_clear
                        );

                              riconoscitore_FSM : riconoscitore
                              PORT MAP
                              (
                                    I       => I,
                                    M       => M,
                                    rst     => rst,
                                    A       => clk,
                                    catch_I => catch_I_clear,
                                    catch_M => catch_M_clear,
                                    Y       => Y,
                                    led     => led
      );

END Behavioral;
```

3.2.1 Simulazione e sintesi

Di seguito vengono riportati il codice del testbench ed il grafico della simulazione per un singolo caso di test, utilizzato per verificare il corretto funzionamento logico dell'automa a stati finiti aggiornato.

```
stimulus : PROCESS
BEGIN
      WAIT FOR 500ns;
      M <= '1';
      I <= '1';
      WAIT FOR 100ns;
      catch_M <= '1';
      WAIT FOR 100ns;
      catch_M <= '0';
      WAIT FOR 80ns;
```

```
        catch_I <= '1';
        WAIT FOR 120ns;
        I       <= '0';
        catch_I <= '0';
        WAIT FOR 80ns;
        catch_I <= '1';
        WAIT FOR 100ns;
        catch_I <= '0';
        WAIT FOR 100ns;
        catch_I <= '1';
        WAIT FOR 100ns;
        I       <= '1';
        catch_I <= '0';
        WAIT FOR 100ns;
        catch_I <= '1';
        WAIT FOR 100ns;
        catch_I <= '0';
        WAIT FOR 100ns;

        WAIT;
END PROCESS;

clock : PROCESS
BEGIN
        clk <= '1';
        WAIT FOR clk_period/2;
        clk <= '0';
        WAIT FOR clk_period/2;
END PROCESS;
```

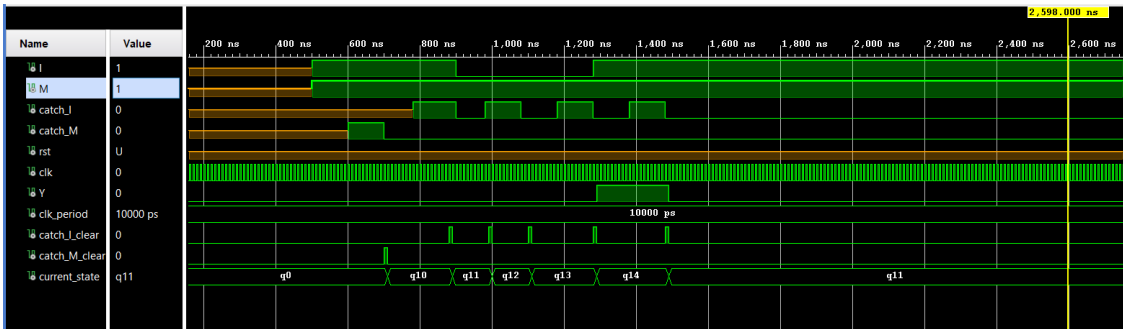


Figura 2.11 - Grafico della simulazione

Nel grafico possiamo notare il funzionamento logico dei debouncer grazie ai signal *"catch_I_clear"* e *"catch_M_clear"* riportati appositamente nella simulazione. Da notare che in questo caso, nella simulazione mostrata, il valore generico del *"btn_noise_time"* risulta notevolmente ridotto affinché l'intero test sia facilmente leggibile. Il test effettuato prevede il funzionamento logico che valuta i bit seriali in ingresso uno alla volta. Abbiamo inserito gli stimoli necessari per generare esattamente

la corretta sequenza da riconoscere. Notiamo che le evoluzioni di stato corrispondono con quelle attese.

Per quanto concerne la fase di sintesi, riportiamo di seguito il file relativo alle constraints utilizzato.

```
## Clock signal
set_property -dict { PACKAGE_PIN E3    IOSTANDARD LVCMOS33 } [get_ports {
clk }]; #IO_L12P_T1_MRCC_35 Sch=clk100mhz
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5}
[get_ports {clk}];
##Switches
set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 } [get_ports { I
}]; #IO_L24N_T3_RS0_15 Sch=sw[0]
set_property -dict { PACKAGE_PIN V10   IOSTANDARD LVCMOS33 } [get_ports { M
}]; #IO_L21P_T3_DQS_14 Sch=sw[15]
## LEDs
set_property -dict { PACKAGE_PIN H17   IOSTANDARD LVCMOS33 } [get_ports { Y
}]; #IO_L18P_T2_A24_15 Sch=led[0]
set_property -dict { PACKAGE_PIN V15   IOSTANDARD LVCMOS33 } [get_ports {
led[0] }]; #IO_L16P_T2_CSI_B_14 Sch=led[12]
set_property -dict { PACKAGE_PIN V14   IOSTANDARD LVCMOS33 } [get_ports {
led[1] }]; #IO_L22N_T3_A04_D20_14 Sch=led[13]
set_property -dict { PACKAGE_PIN V12   IOSTANDARD LVCMOS33 } [get_ports {
led[2] }]; #IO_L20N_T3_A07_D23_14 Sch=led[14]
set_property -dict { PACKAGE_PIN V11   IOSTANDARD LVCMOS33 } [get_ports {
led[3] }]; #IO_L21N_T3_DQS_A06_D22_14 Sch=led[15]
##Buttons
set_property -dict { PACKAGE_PIN N17   IOSTANDARD LVCMOS33 } [get_ports {
rst }]; #IO_L9P_T1_DQS_14 Sch=btnc
set_property -dict { PACKAGE_PIN P17   IOSTANDARD LVCMOS33 } [get_ports {
catch_M }]; #IO_L12P_T1_MRCC_14 Sch=btnl
set_property -dict { PACKAGE_PIN M17   IOSTANDARD LVCMOS33 } [get_ports {
catch_I }]; #IO_L10N_T1_D15_14 Sch=btnr
```

Notiamo che, volendo sintetizzare su scheda un sistema dotato di clock abbiamo dovuto decommentare le righe relative. Abbiamo poi utilizzato due switch: lo switch più a destra sulla scheda (*switch[0]*) viene utilizzato per impostare il valore dell'ingresso mentre lo switch più a sinistra (*switch[15]*) per quello del modo. Seguendo la stessa logica, abbiamo scelto di assegnare al bottone di destra il segnale *catch_I* per catturare il valore dell'ingresso ed al bottone di sinistra il segnale *catch_M* per catturare il valore del modo, oltre al bottone di reset assegnato al bottone centrale. Infine, notiamo che abbiamo utilizzato i quattro LED più a sinistra (da *LED[12]* a *LED[15]*) per la visualizzazione della codifica relativa allo stato registrato dall'automa a stati finiti

mentre il LED più a destra (*LED[0]*) viene utilizzato per la visualizzazione del valore di uscita.

Capitolo 4

Registro a scorrimento

Il registro a scorrimento è un circuito logico sequenziale utilizzato per memorizzare e trasferire dati. È composto da un insieme di banchi di flip flop collegati opportunamente in cascata tra loro. I singoli flip flop condividono il clock e, ad ogni suo fronte attivo, eseguono lo scorrimento dei bit da una cella a quella adiacente. La direzione dello scorrimento può essere fissa o variabile.

Per quanto riguarda il trasferimento di dati, è possibile utilizzare uno shift register per gestire la modalità di trasmissione dei dati stessi (parallela o seriale). È possibile, quindi, definire diversi tipi di shift register in base alla modalità di trasmissione scelta:

- SISO (serial input - serial output) → i registri sono tutti direttamente collegati tra loro e, ad ogni fronte attivo del clock, i dati shiftano di una posizione dal primo flip flop verso l'ultimo;
- SIPO (serial input - parallel output) → i dati sono memorizzati nel registro in maniera seriale, un bit per ogni colpo di clock. Quando il dato è stato interamente caricato, tutti i flip flop assumono il corrispondente valore di ingresso in modo da poter leggere il dato dal registro in maniera parallela;
- PISO (parallel input - serial output) → i dati sono memorizzati nel registro in maniera parallela e vengono letti in maniera seriale.
- PIPO (parallel input - parallel output) → i dati sono memorizzati nel registro in maniera parallela e vengono letti in maniera parallela. Generalmente, viene utilizzato come registro temporaneo ed in questo caso i flip flop potrebbero non essere collegati direttamente tra loro.

Il registro a scorrimento è un componente importante per i sistemi di comunicazione, in particolar modo per convertire il mondo parallelo in seriale durante la trasmissione e l'inverso durante la ricezione.

È importante sottolineare che i flip flop debbano essere sensibili solo ad una variazione, ad esempio quella del fronte di discesa del clock: infatti, se il sistema fosse di tipo latch, non funzionerebbe sempre, perché, dato un ingresso pari ad 1, il primo flip flop acquisirebbe 1 e dopo un certo intervallo temporale tutti gli altri flip flop starebbero memorizzando 1.

4.1 Registro a scorrimento circolare

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. Il componente deve essere realizzato utilizzando sia un a) approccio comportamentale sia un b) approccio strutturale.

Nota: il numero di bit del registro X e i valori che può assumere il parametro Y possono essere scelti dallo studente (ad es. X=8 e Y={1,2}).

4.1.1 Soluzione

Si progetta una macchina che riceve in ingresso la stringa di bit da shiftare, il numero di shift da effettuare e la direzione dello shift. La traccia richiede la progettazione di una soluzione comportamentale e di una strutturale di un registro a scorrimento circolare con direzione variabile.

In entrambi gli approcci, il sistema sarà caratterizzato da 4 modalità di funzionamento:

- 00 → hold (mantiene in uscita il dato, anche se cambiano gli ingressi)
- 01 → right (esegue shift a destra)
- 10 → left (esegue shift a sinistra)
- 11 → load (carica il dato dal segnale di input).

In entrambe le implementazioni, il valore di selezione *sel*, rappresentato su 2 bit, imposta la modalità di funzionamento del sistema indicando se mantenere il valore già presente all'interno del flip flop, acquisirlo dall'ingresso o effettuare lo shift.

4.1.2 Realizzazione (comportamentale)

Nell'approccio comportamentale è stato scelto di eseguire un algoritmo che, all'arrivo di un segnale di tempificazione, salvi in una variabile temporanea il valore corrispondente alla posizione che si perde a causa dello shift. Se lo shift da fare è a destra, si salva il bit meno significativo della stringa da shiftare, altrimenti il più significativo. Dopodiché, si sovrascrive la stringa di bit effettuando lo shift della stringa stessa (tramite la variabile *temp_uscita*) e infine si concatena con il valore precedentemente salvato nella variabile temporanea. L'ordine di concatenazione dipende dalla direzione dello shift. Il processo viene ripetuto Y volte.

L'approccio comportamentale si concretizza nella scrittura di un *process* sensibile al clock.

L'algoritmo varia il funzionamento della macchina in base ai bit di selezione *sel*. Nel caso in cui *sel* sia "01" oppure "10" vuol dire che bisogna fare uno shift rispettivamente a destra o a sinistra e quindi si utilizza una variabile booleana *direction* per tener traccia della direzione dello shift per tutta la durata del ciclo iterativo.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY shift_register IS
	GENERIC
	(
		n : INTEGER := 4;
		m : INTEGER := 2 --posso shiftare fino a 2^m
	);
	PORT
	(
		A        : IN std_logic_vector (n - 1 DOWNTO 0);
		Y        : IN std_logic_vector (m - 1 DOWNTO 0);
		sel      : IN std_logic_vector(1 DOWNTO 0);
		clk, rst : IN std_logic;
		uscita   : OUT std_logic_vector (n - 1 DOWNTO 0)
	);
END shift_register;

ARCHITECTURE Behavioral OF shift_register IS

BEGIN
	PROCESS (clk)
	VARIABLE salto      : INTEGER;
	VARIABLE tmp        : std_logic_vector (n - 1 DOWNTO 0);
	VARIABLE direction  : BOOLEAN := False;
```

```
	VARIABLE tmp_uscita : std_logic_vector(n - 1 DOWNTO 0);
	BEGIN
		salto := to_integer(unsigned(Y));

		IF rising_edge(clk) THEN
			IF (rst = '1') THEN
				uscita <= (OTHERS => '0');
			ELSE
				CASE sel IS
					WHEN "00" => NULL; --Hold
					WHEN "01" => direction := False;--right
shift
	
					WHEN "10" => direction := True;--left
shift
	
					WHEN "11" => tmp       := A;--load
	
	
	
```

il fatto che tale modalità di funzionamento ci è stata utile solo come fonte di ispirazione ma il comportamento del sistema da noi implementato è sostanzialmente differente. Riportiamo per completezza uno schematico di un possibile USR a 2 bit.

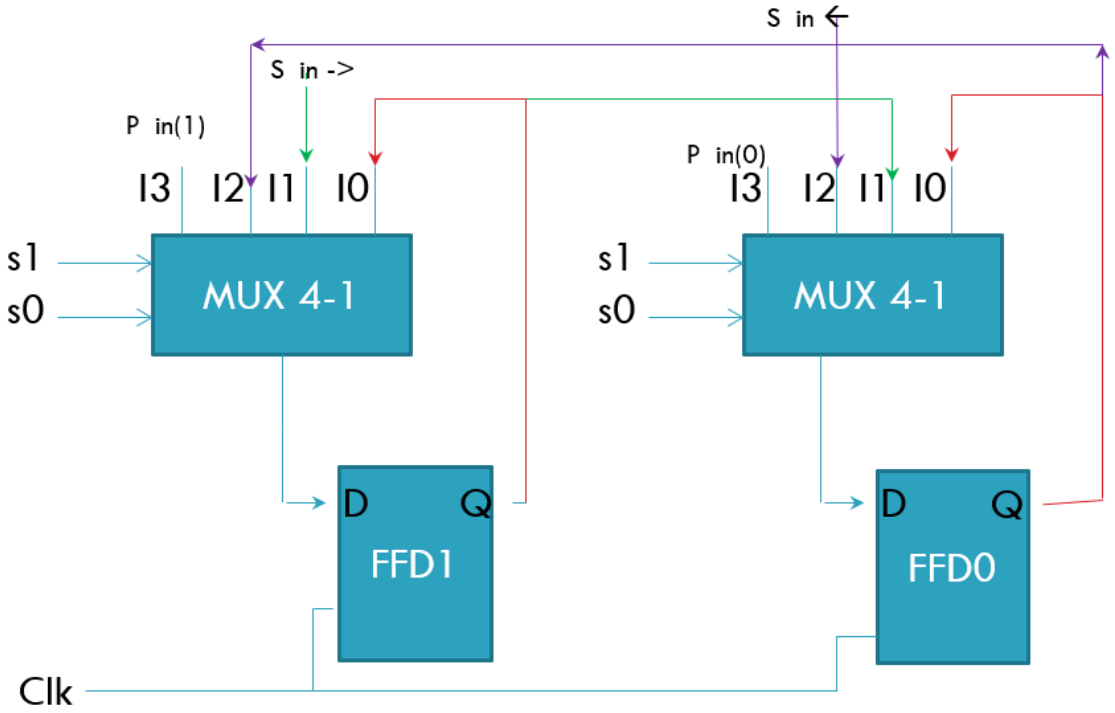


Figura 4.1 - Schematico della USR-2bit

Per l'implementazione abbiamo innanzitutto progettato due componenti base per il sistema: un flip flop ed un multiplexer.

Il flip flop realizzato è un flip flop di tipo D ed ha un segnale in entrata (D) relativo al bit da memorizzare ed i segnali di clock. Quando il clock si trova sul fronte di salita, il flip flop aggiorna la sua uscita Q (il valore memorizzato). In tutti gli altri casi il flip flop conserva Q indipendentemente dal valore e dalle commutazioni di D.

Ne mostriamo di seguito il codice VHDL.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY FFd IS
       PORT
       (
              D        : IN std_logic;
              clk, rst : IN std_logic;
              Q        : OUT std_logic
```

```
	);
END FFd;

ARCHITECTURE Behavioral OF FFd IS

BEGIN
	PROCESS (clk)
	BEGIN
		IF rising_edge(clk) THEN
			Q <= D;
		END IF;
	END PROCESS;

END Behavioral;
```

Abbiamo poi implementato un componente *"controller",* puramente comportamentale, per poter gestire la logica relativa alla modalità di funzionamento della macchina precedentemente descritta.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.numeric_std.ALL;

ENTITY controller IS
	PORT
	(
		Y         : IN std_logic_vector(1 DOWNTO 0);
		sel       : IN std_logic_vector(1 DOWNTO 0);
		selettore : OUT std_logic_vector(2 DOWNTO 0)
	);
END controller;

ARCHITECTURE Behavioral OF controller IS

BEGIN
	PROCESS (Y, sel)
	BEGIN
		IF (sel = "00") THEN -- hold
			selettore <= "000";
		ELSIF (sel = "01") THEN --right
			CASE Y IS
				WHEN "01" => selettore <= "001";
				WHEN "10" => selettore <= "010";
				WHEN "11" => selettore <= "011";
				WHEN OTHERS => selettore    <= "000";
			END CASE;
		ELSIF (sel = "10") THEN --left
			CASE Y IS
				WHEN "01" => selettore <= "011";
				WHEN "10" => selettore <= "010";
				WHEN "11" => selettore <= "001";
				WHEN OTHERS => selettore     <= "000";
			END CASE;
```

```
            ELSE -- Load
                  selettore <= "100";
            END IF;
      END PROCESS;
END Behavioral;
```

Come per il punto precedente, andiamo a realizzare un *process* in cui, dipendentemente dal segnale di selezione *sel*, andiamo a determinare la modalità di funzionamento della macchina (*hold*, *shift a destra/sinistra* o *load*).

L'altro componente di base realizzato, come anticipato, è il multiplexer. Ne riportiamo di seguito l'implementazione.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY mux IS
      PORT
      (
            p0    : IN std_logic;
            p1    : IN std_logic;
            p2    : IN std_logic;
            p3    : IN std_logic;
            input : IN std_logic;
            s     : IN std_logic_vector (2 DOWNTO 0);
            outM  : OUT std_logic
      );
END mux;

ARCHITECTURE Behavioral OF mux IS

BEGIN
      outM <= p0 WHEN s = "000"ELSE
              p1 WHEN s = "001" ELSE
              p2 WHEN s = "010" ELSE
              p3 WHEN s = "011" ELSE
              input WHEN s = "100" ELSE
              '-';
END Behavioral;
```

L'uscita di tale multiplexer viene gestita tramite un segnale di selezione. Se viene selezionato l'ingresso *p0*, dunque, viene mantenuto il dato attualmente salvato all'interno del flip-flop. Nei casi *p1, p2, p3* viene effettuato un "salto" (shift) rispettivamente di 1, 2 o 3 posizioni. Infine, nel caso in cui il segnale di selezione è 100, viene caricato (*load*) il dato di input.

Per maggiore chiarezza riportiamo lo schematico della top-module progettata in modo da poter mostrare tutti i collegamenti utilizzati per congiungere le componenti fin qui descritte.

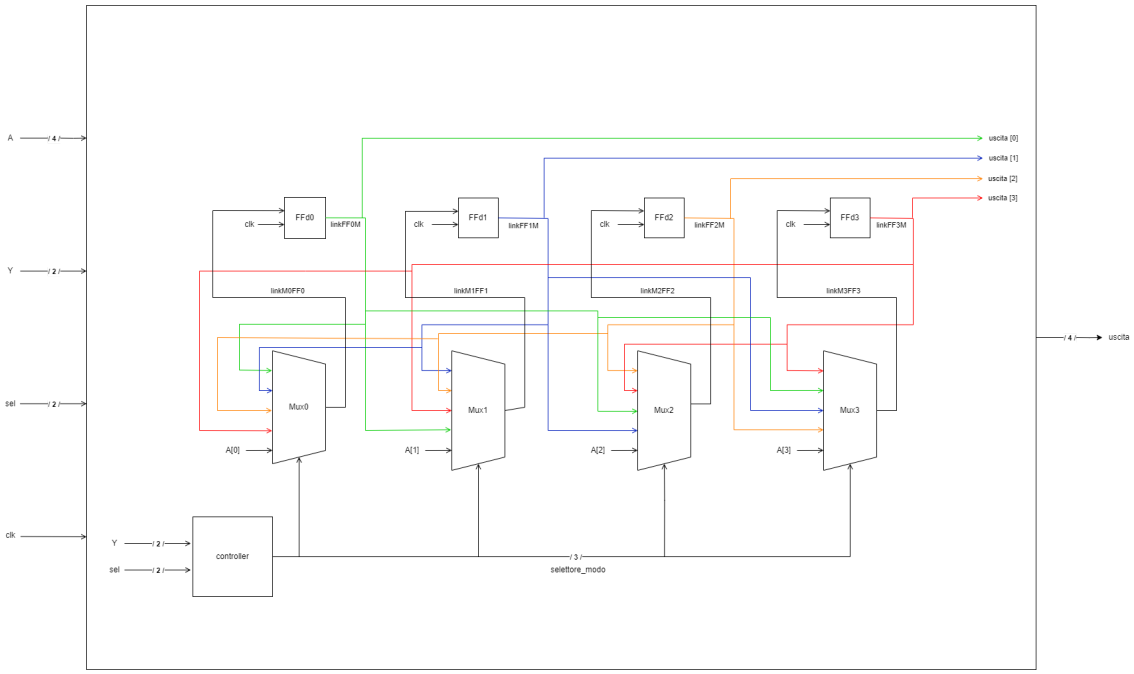


Figura 4.2 - Schematico della top-module

Di seguito riportiamo, infine, l'implementazione VHDL della top-module.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY shift_register_structural IS
      PORT
      (
            A        : IN std_logic_vector (3 DOWNTO 0);
            Y        : IN std_logic_vector (1 DOWNTO 0);
            sel      : IN std_logic_vector (1 DOWNTO 0);
            clk      : IN std_logic;
            uscita   : OUT std_logic_vector (3 DOWNTO 0)
      );
END shift_register_structural;

ARCHITECTURE Structural OF shift_register_structural IS

      SIGNAL linkM0FF0      : std_logic := '0';
      SIGNAL linkM1FF1      : std_logic := '0';
      SIGNAL linkM2FF2      : std_logic := '0';
      SIGNAL linkM3FF3      : std_logic := '0';
      SIGNAL linkFF0M       : std_logic := '0';
      SIGNAL linkFF1M       : std_logic := '0';
```

```
	SIGNAL linkFF2M       : std_logic := '0';
	SIGNAL linkFF3M       : std_logic := '0';
	SIGNAL selettore_modo : std_logic_vector(2 DOWNTO 0) := "000";

	COMPONENT FFd IS
		PORT
		(
			D   : IN std_logic;
			clk : IN std_logic;
			Q   : OUT std_logic
		);
	END COMPONENT;

	COMPONENT mux IS
		PORT
		(
			p0    : IN std_logic;
			p1    : IN std_logic;
			p2    : IN std_logic;
			p3    : IN std_logic;
			input : IN std_logic;
			s     : IN std_logic_vector (2 DOWNTO 0);
			outM  : OUT std_logic
		);
	END COMPONENT;

	COMPONENT controller IS
		PORT
		(
			Y         : IN std_logic_vector(1 DOWNTO 0);
			sel       : IN std_logic_vector (1 DOWNTO 0);
			selettore : OUT std_logic_vector(2 DOWNTO 0)
		);
	END COMPONENT;

BEGIN
	FFd0 : FFd
	PORT MAP
	(
		D   => linkM0FF0,
		clk => clk,
		Q   => linkFF0M
	);

	uscita(0) <= linkFF0M;

	FFd1 : FFd
	PORT MAP
	(
		D   => linkM1FF1,
		clk => clk,
		Q   => linkFF1M
	);

	uscita(1) <= linkFF1M;
```

```
FFd2 : FFd
PORT MAP
(
       D   => linkM2FF2,
       clk => clk,
       Q   => linkFF2M
);

uscita(2) <= linkFF2M;
FFd3 : FFd
PORT MAP
(
       D   => linkM3FF3,
       clk => clk,
       Q   => linkFF3M
);

uscita(3) <= linkFF3M;

mux0 : mux
PORT MAP
(
       p0    => linkFF0M,
       p1    => linkFF1M,
       p2    => linkFF2M,
       p3    => linkFF3M,
       input => A(0),
       s     => selettore_modo,
       outM  => linkM0FF0
);

mux1 : mux
PORT MAP
(
       p0    => linkFF1M,
       p1    => linkFF2M,
       p2    => linkFF3M,
       p3    => linkFF0M,
       input => A(1),
       s     => selettore_modo,
       outM  => linkM1FF1
);

mux2 : mux
PORT MAP
(
       p0    => linkFF2M,
       p1    => linkFF3M,
       p2    => linkFF0M,
       p3    => linkFF1M,
       input => A(2),
       s     => selettore_modo,
       outM  => linkM2FF2
);

mux3 : mux
```

```
        PORT MAP
        (
                p0    => linkFF3M,
                p1    => linkFF0M,
                p2    => linkFF1M,
                p3    => linkFF2M,
                input => A(3),
                s     => selettore_modo,
                outM  => linkM3FF3
        );

        controllore : controller
        PORT MAP
        (
                Y         => Y,
                sel       => sel,
                selettore => selettore_modo
        );

END Structural;
```

Abbiamo, dunque, manualmente instanziato 4 flip flop ed i relativi 4 multiplexer, oltre al componente puramente comportamentale per la gestione del funzionamento della macchina. Ognuna di queste componenti è opportunamente collegata tramite una serie di segnali.

4.1.4 Simulazione

Nonostante le realizzazioni mostrate sopra siano differenti, l'interfaccia ed il comportamento delle macchine sono esattamente le stesse. La differenza sta solo nella loro realizzazione. Si è costruito quindi un unico "stimolo" che, importato all'interno dei due progetti, restituisce esattamente la stessa uscita.

```
stim_proc : PROCESS
BEGIN
        WAIT FOR 100 ns;
        sel <= "00";
        WAIT FOR 20ns;
        A <= "0110";
        sel <= "11";
        WAIT FOR 50ns;
        Y <= "01";
        WAIT FOR 10 ns;
        sel <= "01";
        WAIT FOR 50ns;
        A <= "1110";
```

```
        sel <= "11";
        WAIT FOR 50ns;
        sel <= "10";
        WAIT FOR 50ns;
        WAIT;
END PROCESS;
```

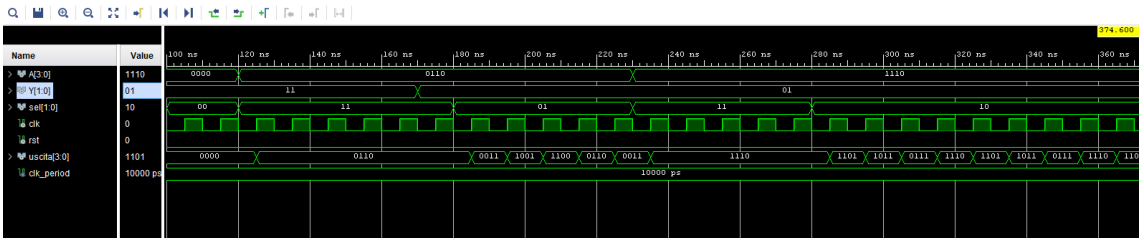


Figura 4.3 - Grafico della simulazione

Dopo 100 ns, *sel* diventa “11” cioè si passa dalla modalità *“hold”* a quella *“load”*. Quindi il dato viene salvato, nel caso comportamentale in una variabile mentre nel caso strutturale all’interno di flip flop. A 170 ns, si passa da *“load”* a *“shift a destra”* e quindi, si shifta di Y posizioni circolarmente. A 230 ns cambia la modalità da *“shift a destra”* a *“load”,* la quale porta in uscita l’ingresso A=“1110”. A 280 ns la modalità passa da *“shift a destra”* a *“shift a sinistra”*, sempre di Y=1 posizioni e notiamo che l’uscita effettivamente viene aggiornata correttamente.

Capitolo 5

Cronometro

Per la progettazione di un cronometro è necessario introdurre il concetto di contatore. Un contatore dotato di un certo modulo X è un dispositivo in grado di contare X volte (dunque da 0 ad X-1). Esistono diverse possibilità per la realizzazione di un contatore. Le elenchiamo brevemente:

- realizzazione tramite automa → in questo caso il contatore è una macchina sequenziale. L'inconveniente di tale strategia è che l'automa avrà un numero di stati pari al numero del conteggio del modulo che si vuole effettuare. Al crescere di tale conteggio, dunque, crescerà anche il numero di stati dell'automa, risultando in un componente non facilmente sintetizzabile;
- realizzazione tramite composizione di più contatori → realizziamo un contatore mediante composizione di componenti più piccole. In questo caso si adotta allora un approccio strutturale, definendo così gli oggetti che comporranno tale struttura e le loro interconnessioni, sfruttando in questo modo i tipici vantaggi di modularità e simmetria;
- contatore ad incremento → realizzato tramite un addizionatore che permette di indicare l'incremento non più unitario ma di valore variabile.

Oltre al concetto di contatore è necessario introdurre anche quello di divisore di frequenza. Il divisore di frequenza è un circuito che riceve in ingresso una certa frequenza f e restituisce in uscita una frequenza f/n con n numero intero.

5.1 Cronometro

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.

5.1.1 Soluzione

Nel pieno rispetto della traccia, abbiamo progettato il cronometro mediante un approccio strutturale e, dunque, tramite composizione di più componenti. La prima idea è stata quella di voler realizzare un unico componente contatore che fosse in grado di contare contestualmente secondi, minuti ed ore. Dopo una prima realizzazione, ci siamo scontrati con alcuni problemi legati alla tempificazione delle uscite. In particolare, l'aggiornamento del valore delle ore era legato - ovviamente - all'aggiornamento contemporaneo di minuti e secondi. Abbiamo così deciso di optare per una soluzione differente: suddividere ulteriormente in maniera strutturale tale contatore in 3 contatori più piccoli, uno relativo al conteggio dei secondi, uno dei minuti e l'ultimo delle ore.

5.1.2 Realizzazione

Prima di procedere con l'analisi delle singole componenti, in questo caso riteniamo opportuno presentare innanzitutto lo schematico della top-module realizzato in fase di progettazione per favorire la chiarezza della documentazione del sistema stesso.

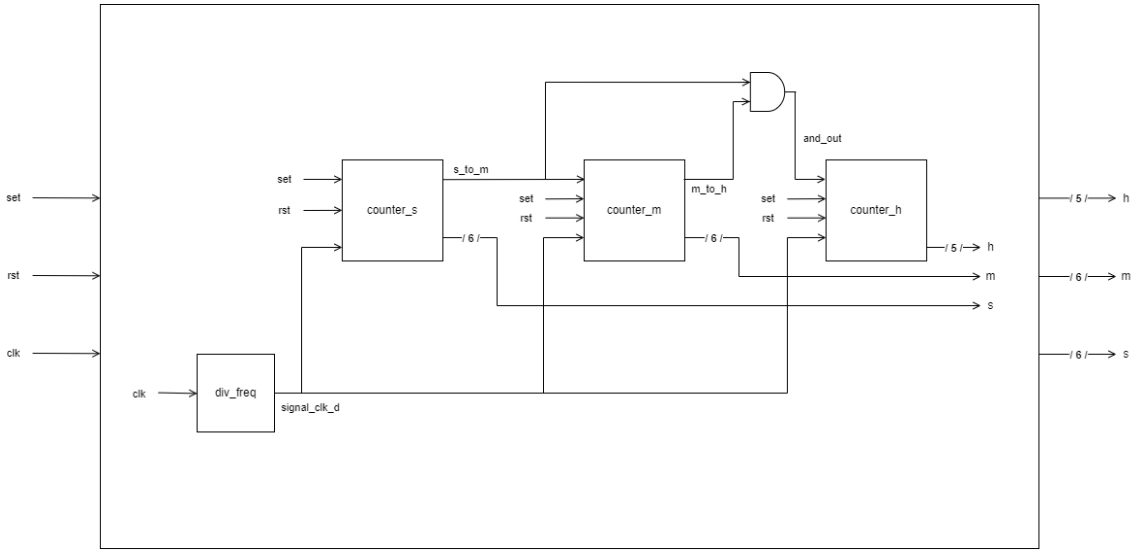


Figura 5.1 - Schematico della top-module

Come è possibile notare anche dallo schematico, utilizziamo il clock della scheda come base dei tempi del sistema. Tale clock viene messo in ingresso ad un divisore di frequenza che fornisce in uscita la frequenza a cui vogliamo che lavorino le altre componenti del sistema per poter effettuare correttamente il conteggio di secondi, minuti ed ore. Notiamo, inoltre, la presenza di una porta *AND* che collega le uscite dei contatori di secondi e minuti per poter poi fornire tale segnale in ingresso al contatore delle ore. Questo ci è servito per poter aggiornare correttamente il conteggio delle ore in base sia ai minuti che ai secondi a differenza, ad esempio, proprio del contatore dei minuti che prende in ingresso un segnale interno al sistema che non è altro che l'uscita del contatore dei secondi. Tutti i contatori sono, inoltre, forniti di opportuni segnali di *set* e *reset*. Notiamo, infine, che a differenza di secondi e minuti (rappresentabili su 6 bit) per poter contare le *24* ore sono stati ampiamente sufficienti soltanto 5 bit.

Nella presentazione delle singole componenti, procediamo con ordine partendo proprio dal divisore di frequenza, di cui ne riportiamo di seguito l'implementazione VHDL.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY div_freq IS
      GENERIC
```

```
	(
		clk_period : INTEGER := 10;
		max_count  : INTEGER := 100000000
	);
	PORT
	(
		clk   : IN std_logic;
		clk_d : OUT std_logic
	);
END div_freq;

ARCHITECTURE Behavioral OF div_freq IS
BEGIN
	PROCESS (clk)
	VARIABLE cnt : INTEGER := 1;
	BEGIN
		IF falling_edge(clk) THEN
			IF (cnt < max_count/2) THEN
				cnt := cnt + 1;
				clk_d <= '0';
			ELSIF (cnt >= max_count/2 AND cnt < max_count) THEN
				cnt := cnt + 1;
				clk_d <= '1';
			ELSE
				cnt := 1;
				clk_d <= '0';
			END IF;
		END IF;
	END PROCESS;

END Behavioral;
```

Tramite il divisore di frequenza, vogliamo replicare il comportamento del clock della scheda ma ad una frequenza più bassa. Per fare ciò abbiamo scelto di utilizzare un approccio puramente comportamentale per descrivere un semplice contatore. In particolare, abbiamo scelto un valore di massimo conteggio (*max_count)* uguale a 100000000 che, dimezzato opportunamento all'interno dell'algoritmo al fine di avere un duty cycle del 50%, ci ha permesso di replicare una frequenza molto vicina a quella di un secondo.

Passiamo, adesso, alla descrizione dei tre contatori. Presentiamo, innanzitutto, il codice VHDL per il contatore dei secondi.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.std_logic_unsigned.ALL;
USE IEEE.NUMERIC_STD.ALL;
```

```
ENTITY counter_s IS
	PORT
	(
		set, rst : IN std_logic;
		clk_d    : IN std_logic;
		flag_max : OUT std_logic;
		output   : OUT std_logic_vector(5 DOWNTO 0)
	);
END counter_s;

ARCHITECTURE Behavioral OF counter_s IS

	CONSTANT max_secondi : INTEGER := 59;

BEGIN
	PROCESS (clk_d)
	VARIABLE cnt : INTEGER := 0;
	BEGIN
		IF falling_edge(clk_d) THEN
			IF (set = '1' OR rst = '1') THEN
				flag_max <= '0';
				output   <= "000000";
				cnt := 0;
				IF (cnt = max_secondi) THEN
					flag_max <= '1';
				END IF;
			ELSIF (cnt < max_secondi - 1) THEN
				flag_max <= '0';
				output   <= std_logic_vector(to_unsigned(cnt,
6));
				cnt := cnt + 1;
			ELSIF (cnt = max_secondi - 1) THEN
				flag_max <= '1';
				output   <= std_logic_vector(to_unsigned(cnt,
6));
				cnt := cnt + 1;
			ELSIF (cnt = max_secondi) THEN
				flag_max <= '0';
				output   <= std_logic_vector(to_unsigned(cnt,
6));
				cnt := 0;
			END IF;
		END IF;
	END PROCESS;

END Behavioral;
```

Notiamo che impostiamo il valore massimo dei secondi a 59 tramite una *constant.* Quando il contatore arriva al valore 58, iniziamo a mandare in uscita - oltre al valore dei secondi stesso tramite il segnale *output* - anche un valore del bit alto per il segnale *flag_max* in modo da dare il tempo necessario alla propagazione del segnale stesso. Questo viene fatto al fine di notificare opportunamente il contatore dei minuti (e

successivamente, tramite la porta AND precedentemente introdotta, anche il contatore delle ore) del raggiungimento del 59° secondo.

Di seguito riportiamo, poi, il codice realizzato per il contatore dei minuti.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.std_logic_unsigned.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY counter_m IS
	PORT
	(
		input    : IN std_logic;
		set, rst : IN std_logic;
		clk_d    : IN std_logic;
		flag_max : OUT std_logic;
		output   : OUT std_logic_vector(5 DOWNTO 0)
	);
END counter_m;

ARCHITECTURE Behavioral OF counter_m IS

	CONSTANT set_minuti : INTEGER := 30;
	CONSTANT max_minuti : INTEGER := 59;

BEGIN
	PROCESS (clk_d)
	VARIABLE cnt : INTEGER := 0;
	BEGIN
		IF falling_edge(clk_d) THEN
			IF (rst = '1') THEN
				flag_max <= '0';
				output   <= "000000";
				cnt := 0;
			ELSIF (set = '1') THEN
				flag_max <= '0';
				output   <=
std_logic_vector(to_unsigned(set_minuti, 6));
				cnt := set_minuti;
				IF (cnt = max_minuti) THEN
					flag_max <= '1';
				END IF;
			ELSIF (cnt < max_minuti AND input = '1') THEN
				flag_max <= '0';
				output   <= std_logic_vector(to_unsigned(cnt,
6));
				cnt := cnt + 1;
			ELSIF (cnt = max_minuti AND input = '0') THEN
				flag_max <= '1';
				output   <= std_logic_vector(to_unsigned(cnt,
6));
			ELSIF (cnt = max_minuti AND input = '1') THEN
				flag_max <= '1';
				output   <= std_logic_vector(to_unsigned(cnt,
```

```
6));
                                cnt := 0;
                        ELSE --(cnt < max_minuti and input = '0')
                                flag_max <= '0';
                                output   <= std_logic_vector(to_unsigned(cnt,
6));
                        END IF;
                END IF;
        END PROCESS;

END Behavioral;
```

In maniera del tutto simile a quanto mostrato in precedenza per il contatore dei secondi, impostiamo anche nel caso del contatore dei minuti un valore di conteggio massimo (*max_minuti*) a 59. Quando tale valore viene raggiunto, mandiamo in uscita una flag (*flag_max*) per segnalare al contatore delle ore che si è raggiunto il 59° minuto. Notiamo, inoltre, che in questo caso abbiamo impostato anche una *constant* chiamata *set_minuti* per poter impostare comodamente un determinato valore ai minuti. Questo, come vedremo nel successivo paragrafo, ci è stato molto utile soprattutto in fase di simulazione e testing del sistema.

Di seguito, infine, riportiamo l'implementazione del contatore delle ore.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE ieee.std_logic_unsigned.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY counter_h IS
        PORT
        (
                input    : IN std_logic;
                set, rst : IN std_logic;
                clk_d    : IN std_logic;
                output   : OUT std_logic_vector(4 DOWNTO 0)
        );
END counter_h;

ARCHITECTURE Behavioral OF counter_h IS

        CONSTANT set_ore : INTEGER := 12;
        CONSTANT max_ore : INTEGER := 23;

BEGIN
        PROCESS (clk_d)
        VARIABLE cnt : INTEGER := 0;
        BEGIN
                IF falling_edge(clk_d) THEN
                        IF (rst = '1') THEN
```

```
                        output   <= "00000";
                        cnt := 0;
                    ELSIF (set = '1') THEN
                        output   <= std_logic_vector(to_unsigned(set_ore,
5));
                        cnt := set_ore;
                    ELSIF (cnt < max_ore - 1 AND input = '1') THEN
                        output   <= std_logic_vector(to_unsigned(cnt,
5));
                        cnt := cnt + 1;
                    ELSIF (cnt = max_ore - 1 AND input = '1') THEN
                        output   <= std_logic_vector(to_unsigned(cnt,
5));
                        cnt := cnt + 1;
                    ELSIF (cnt = max_ore AND input = '1') THEN
                        output   <= std_logic_vector(to_unsigned(cnt,
5));
                        cnt := 0;
                    ELSE
                        output   <= std_logic_vector(to_unsigned(cnt,
5));
                    END IF;
            END IF;
      END PROCESS;

END Behavioral;
```

Nel caso del contatore delle ore, ovviamente il valore massimo raggiungibile del contatore stesso (indicato dalla *constant max_ore*) è stato impostato a 23. Notiamo che all'arrivo di un valore di *input* alto (risultato della porta AND in cui abbiamo collegato i rispettivi flag di massimo valore raggiunto di secondi e minuti), il conteggio dovrà essere riazzerato.

Prima di proseguire con la presentazione del codice VHDL della top-module, vogliamo soffermarci sulla scelta di far lavorare tutti i contatori sul fronte di discesa (falling edge) del clock. Se, infatti, tali componenti fossero stati attivi sul fronte di salita (rising edge) del clock, quando il primo conta anche il secondo commuterebbe pur avendo avuto un solo segnale di conteggio in quanto sensibile proprio al fronte di salita dell'uscita del precedente. Lavorare sul fronte di discesa, dunque, ci ha permesso di ovviare a tale problematica. Un modo per evitare che qualcosa si ricombini con ciò che l'ha generato è generarlo quando quest'ultimo non c'è più e, dunque, lavorare sul suo fronte di discesa. Se, però, avessimo utilizzato solo tale accortezza, avremmo avuto comunque un problema relativo alla propagazione dei segnali. Proprio per risolvere tale problema

abbiamo deciso di inserire la porta AND in ingresso al terzo contatore (quello delle ore). In questo modo il clock viene dato in parallelo ai tre contatori e dunque risolviamo il problema di propagazione.

Procediamo adesso col mostrare l'implementazione della top-module realizzata.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY cronometro IS
	PORT
	(
		set, rst : IN std_logic;
		clk      : IN std_logic;
		m, s     : OUT std_logic_vector(5 DOWNTO 0);
		h        : OUT std_logic_vector(4 DOWNTO 0)
	);
END cronometro;

ARCHITECTURE Structural OF cronometro IS

	COMPONENT div_freq IS
		GENERIC (clk_period : INTEGER := 10);
		PORT
		(
			clk   : IN std_logic;
			clk_d : OUT std_logic
		);
	END COMPONENT;

	COMPONENT counter_s IS
		PORT
		(
			set, rst : IN std_logic;
			clk_d    : IN std_logic;
			flag_max : OUT std_logic;
			output   : OUT std_logic_vector(5 DOWNTO 0)
		);
	END COMPONENT;

	COMPONENT counter_m IS
		PORT
		(
			input    : IN std_logic;
			set, rst : IN std_logic;
			clk_d    : IN std_logic;
			flag_max : OUT std_logic;
			output   : OUT std_logic_vector(5 DOWNTO 0)
		);
	END COMPONENT;

	COMPONENT counter_h IS
		PORT
		(
```

```
                    input     : IN std_logic;
                    set, rst  : IN std_logic;
                    clk_d     : IN std_logic;
                    output    : OUT std_logic_vector(4 DOWNTO 0)
              );
       END COMPONENT;

       SIGNAL signal_clk_d : std_logic;
       SIGNAL s_to_m       : std_logic;
       SIGNAL m_to_h       : std_logic;
       SIGNAL and_out      : std_logic;

BEGIN
       divisore_frequenza : div_freq
              GENERIC MAP
              (
              clk_period => 10
              )
              PORT MAP
              (
                    clk   => clk,
                    clk_d => signal_clk_d
              );


                    cnt_s : counter_s
                    PORT MAP
                    (
                           set      => set,
                           rst      => rst,
                           clk_d    => signal_clk_d,
                           flag_max => s_to_m,
                           output   => s
       );

       cnt_m : counter_m
       PORT MAP
       (
              input    => s_to_m,
              set      => set,
              rst      => rst,
              clk_d    => signal_clk_d,
              flag_max => m_to_h,
              output   => m
       );

       and_out <= s_to_m AND m_to_h;

       cnt_h : counter_h
       PORT MAP
       (
              input  => and_out,
              set    => set,
              rst    => rst,
              clk_d  => signal_clk_d,
              output => h
```

```
        );
END Structural;
```

Notiamo che la porta AND è stata realizzata direttamente nella top-module.

5.1.3 Simulazione

Per quanto riguarda la fase di simulazione, abbiamo innanzitutto deciso di testare il corretto funzionamento del divisore di frequenza al fine di verificare la corretta suddivisione della base dei tempi della macchina.

Dopo aver impostato un orario tramite le apposite *constant* relative a minuti ed ore spiegate nel paragrafo precedente, abbiamo effettuato una simulazione di alcuni secondi in modo da valutare il corretto conteggio dei secondi stessi.

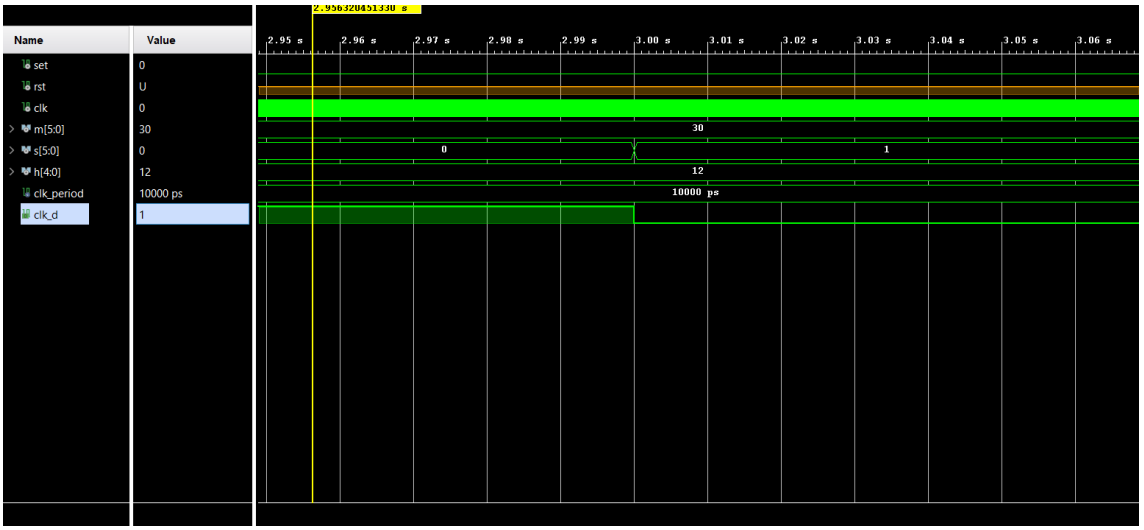


Figura 5.2 - Grafico della prima simulazione

Come è possibile notare, il conteggio dei secondi avviene correttamente.

In seguito, abbiamo ritenuto opportuno valutare anche il corretto comportamento della macchina complessiva. Abbiamo allora ridotto il valore di *max_count* all'interno del divisore di frequenza in modo tale da poter simulare il sistema in maniera più agevole. Il risultato ottenuto è mostrato in figura 5.3.

Figura 5.3 - Grafico della seconda simulazione

Osserviamo che l'avanzare di secondi, minuti ed ore avviene correttamente.

5.2 Cronometro su FPGA

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).

5.2.1 Soluzione

Per l'inserimento dell'orario sulla nostra scheda, ci siamo serviti di un bottone per il *set* che sarà utilizzato in combinazione con gli switch per impostare l'orario desiderato dal quale il cronometro inizierà a contare. Così come per il *set*, anche per il *reset* è previsto un bottone che, in questo caso, andrà a riportare il cronometro nella sua configurazione iniziale (00:00:00).

In realtà, rispetto al solito, abbiamo ritenuto non necessario aggiungere un debouncer per i bottoni di *set* e *reset*. L'utilizzo di un debouncer è giustificato quando ci si vuole assicurare di prendere un'unica volta il valore dal bottone. In questo caso, impostare più volte gli stessi valori non costituisce un problema ed inoltre le oscillazioni vengono prodotte nell'ordine dei millisecondi mentre il nostro sistema lavora nell'ordine dei secondi. Non è quindi necessario gestire le eventuali oscillazioni del bottone.

Un primo problema riscontrato durante la fase di progettazione riguarda proprio l'inserimento dell'orario, in quanto, per inserire un orario "completo" servirebbero 17 ingressi, e quindi 17 switch:

- 6 bit per i secondi e 6 per i minuti che permettono di inserire un valore da 0 a 63 (nel nostro caso è sufficiente raggiungere 59);
- 5 bit per le ore che permettono di inserire un valore da 0 a 31 (nel nostro caso è sufficiente raggiungere 23).

La nostra scheda presenta in totale 16 switch, dunque insufficienti per poter effettuare l'inserimento. Per ovviare a tale problema si è scelto di considerare una codifica per i secondi così da utilizzare soltanto 4 switch a discapito della precisione relativa al loro inserimento. Nello specifico abbiamo deciso di poter inserire soltanto i valori multipli di 5 e considerare quindi i 12 valori fondamentali dell'orologio (0, 5, 10, 15, 20 e così via fino a 55). In questo modo siamo in grado di scegliere l'orario da settare utilizzando in totale soltanto 15 switch.

Risolto il problema fisico relativo al numero di switch da considerare, risultano evidenti i problemi derivanti dal fatto che sarebbe possibile inserire un valore maggiore rispetto al massimo valore possibile per minuti ed ore (rispettivamente 59 e 23). Per la risoluzione di tale problematica abbiamo deciso di inserire una logica che rimpiazza l'eventuale valore errato col valore massimo possibile.

Infine, l'ultima problematica da affrontare per sintetizzare il nostro cronometro è quella della gestione del display e nello specifico degli anodi. A differenza dell'encoder (*Progetto 2*) dove il display è stato utilizzato per illuminare una sola cifra, questa volta

abbiamo la necessità di rappresentarne otto (due per i secondi, due per i minuti e due per le ore). Il problema consiste nel fatto che non è possibile illuminare contemporaneamente più di un digit alla volta sul display. Per risolvere tale problema, bisognerà illuminare una cifra alla volta, con una frequenza tale che l'intermittenza tra un'accensione ed uno spegnimento consecutivi non risultino percepibili sfruttando la persistenza dei fosfori. Si parla quindi di multiplexing nel tempo. Per realizzare ciò, la frequenza di refresh del catodo deve essere più piccola della frequenza del clock. Nello specifico il periodo di refresh, ovvero l'intervallo di tempo necessario per poter illuminare tutti i display, deve essere compreso tra 1 ms e 16 ms. Se il periodo di refresh è di 16 ms, nel nostro caso, avendo 8 digit da mostrare, ogni cifra dovrà rimanere accesa per 2 ms (*digit period*) e in tal caso la frequenza considerata sarà di 500 Hz.

5.2.2 Realizzazione

La top-module del cronometro, oltre alle componenti già analizzate nel punto precedente, in questo caso include anche la componente *"display_seven_segments"* per visualizzare l'output su display e le componenti *"seconds_encoder", "logic_handler_m"* e *"logic_handler_h"* impiegate per effettuare il corretto inserimento dell'orario. Di seguito riportiamo lo schematico aggiornato della top-module.

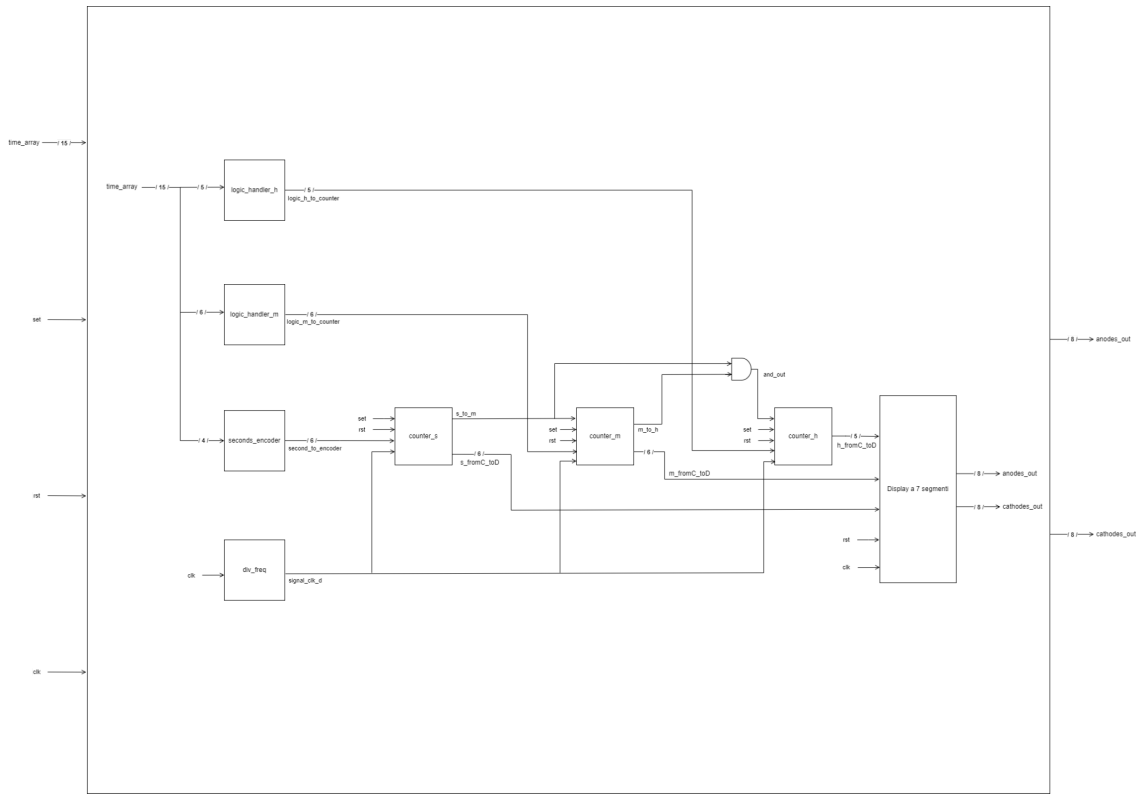


Figura 5.4 - Schematico della top-module del cronometro

Considerando lo schematico mostrato, andiamo ad analizzare le sole componenti aggiunte rispetto all'esercizio precedente. Nello specifico notiamo la presenza dell'ingresso *"time_array"* (di 15 bit) della top-module, il quale, una volta all'interno della top-module, andrà a diramarsi in tre gruppi di bit: i 4 bit meno significativi entreranno nella component *"seconds_encoder"* per gestire i secondi, i successivi 6 bit andranno in ingresso alla component *"logic_handler_m"* per i minuti e gli ultimi 5 saranno indirizzati verso la component *"logic_handler_h"* per le ore. Le varie uscite prodotte da questi componenti saranno poi collegate attraverso una serie di *signal* rispettivamente al counter dei secondi, dei minuti e delle ore che, per questo motivo, sono dotati di un ulteriore ingresso.

Come già detto, l'output dovrà essere visualizzato tramite display. Per questo motivo le uscite dei contatori non sono indirizzate direttamente all'uscita della top-module, ma saranno collegate in ingresso alla componente *"display_seven_segments"* che produrrà l'output finale.

Riportiamo di seguito il codice della top-module, per poi andarne ad analizzare le singole componenti interne con relativo codice.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY cronometro IS
	PORT
	(
		time_array   : IN std_logic_vector(14 DOWNTO 0);
		set, rst     : IN std_logic;
		clk          : IN std_logic;
		anodes_out   : OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
		cathodes_out : OUT STD_LOGIC_VECTOR (7 DOWNTO 0)
	);
END cronometro;

ARCHITECTURE Structural OF cronometro IS

	COMPONENT div_freq IS
		GENERIC (clk_period : INTEGER := 10);
		PORT
		(
			clk   : IN std_logic;
			clk_d : OUT std_logic
		);
	END COMPONENT;

	COMPONENT counter_s IS
		PORT
		(
			set, rst   : IN std_logic;
			set_switch : IN std_logic_vector(5 DOWNTO 0);
			clk_d      : IN std_logic;
			flag_max   : OUT std_logic;
			output     : OUT std_logic_vector(5 DOWNTO 0)
		);
	END COMPONENT;

	COMPONENT counter_m IS
		PORT
		(
			input      : IN std_logic;
			set, rst   : IN std_logic;
			clk_d      : IN std_logic;
			set_switch : IN std_logic_vector(5 DOWNTO 0);
			flag_max   : OUT std_logic;
			output     : OUT std_logic_vector(5 DOWNTO 0)
		);
	END COMPONENT;

	COMPONENT counter_h IS
		PORT
		(
			input      : IN std_logic;
```

```
                set, rst    : IN std_logic;
                set_switch : IN std_logic_vector(4 DOWNTO 0);
                clk_d       : IN std_logic;
                output      : OUT std_logic_vector(4 DOWNTO 0)
        );
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;
                s         : IN STD_LOGIC_VECTOR (5 DOWNTO 0);
                m         : IN STD_LOGIC_VECTOR (5 DOWNTO 0);
                h         : IN STD_LOGIC_VECTOR (4 DOWNTO 0);
                ANODES    : OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
                CATHODES : OUT STD_LOGIC_VECTOR (7 DOWNTO 0)
        );
END COMPONENT;

COMPONENT seconds_encoder IS
        PORT
        (
                switch_in   : IN std_logic_vector(3 DOWNTO 0);
                switch_out : OUT std_logic_vector(5 DOWNTO 0)
        );
END COMPONENT;

COMPONENT logic_handler_m IS
        PORT
        (
                set_switch : IN std_logic_vector(5 DOWNTO 0);
                output      : OUT std_logic_vector(5 DOWNTO 0)
        );
END COMPONENT;

COMPONENT logic_handler_h IS
        PORT
        (
                set_switch : IN std_logic_vector(4 DOWNTO 0);
                output      : OUT std_logic_vector(4 DOWNTO 0)
        );
END COMPONENT;

SIGNAL signal_clk_d        : std_logic;
SIGNAL s_to_m              : std_logic;
SIGNAL m_to_h              : std_logic;
SIGNAL and_out             : std_logic;

SIGNAL s_fromC_toD         : std_logic_vector(5 DOWNTO 0);
SIGNAL m_fromC_toD         : std_logic_vector(5 DOWNTO 0);
```

```
	SIGNAL h_fromC_toD        : std_logic_vector(4 DOWNTO 0);

	SIGNAL reset_n            : std_logic;

	SIGNAL second_to_encode   : std_logic_vector(5 DOWNTO 0) :=
"000000";

	SIGNAL logic_m_to_counter : std_logic_vector(5 DOWNTO 0) :=
"000000";
	SIGNAL logic_h_to_counter : std_logic_vector(4 DOWNTO 0) := "00000";
BEGIN
	reset_n <= NOT rst;

	divisore_frequenza : div_freq
		GENERIC MAP
		(
		clk_period => 10
		)
		PORT MAP
		(
			clk   => clk,
			clk_d => signal_clk_d
		);

			cnt_s : counter_s
			PORT MAP
			(
				set        => set,
				rst        => reset_n,
				set_switch => second_to_encode,
				clk_d      => signal_clk_d,
				flag_max   => s_to_m,
				output     => s_fromC_toD
	);

	cnt_m : counter_m
	PORT MAP
	(
		input      => s_to_m,
		set        => set,
		rst        => reset_n,
		set_switch => logic_m_to_counter,
		clk_d      => signal_clk_d,
		flag_max   => m_to_h,
		output     => m_fromC_toD
	);

	and_out <= s_to_m AND m_to_h;

	cnt_h : counter_h
	PORT MAP
	(
		input      => and_out,
		set        => set,
		rst        => reset_n,
		set_switch => logic_h_to_counter,
```

```
		clk_d      => signal_clk_d,
		output     => h_fromC_toD
	);

	display : display_seven_segments
		GENERIC MAP
		(
		CLKIN_freq  => 100000000,
		CLKOUT_freq => 500)
		PORT MAP
		(
			CLK      => CLK,
			RST      => reset_n,
			s        => s_fromC_toD,
			m        => m_fromC_toD,
			h        => h_fromC_toD,
			ANODES   => anodes_out,
			CATHODES => cathodes_out
		);

			s_encoder : seconds_encoder
			PORT MAP
			(
				switch_in  => time_array(3 DOWNTO 0),
				switch_out => second_to_encode(5 DOWNTO 0)
	);

	log_handler_m : logic_handler_m
	PORT MAP
	(
		set_switch => time_array(9 DOWNTO 4),
		output     => logic_m_to_counter
	);

	log_handler_h : logic_handler_h
	PORT MAP
	(
		set_switch => time_array(14 DOWNTO 10),
		output     => logic_h_to_counter
	);

END Structural;
```

La componente *seconds_encoder* viene utilizzata per effettuare la codifica per il corretto inserimento dei secondi, come spiegato in precedenza. Di seguito ne riportiamo il codice VHDL.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY seconds_encoder IS
	PORT
```

```
	(
		switch_in  : IN std_logic_vector(3 DOWNTO 0);
		switch_out : OUT std_logic_vector(5 DOWNTO 0)
	);
END seconds_encoder;

ARCHITECTURE data_flow OF seconds_encoder IS

BEGIN
	switch_out <=
		"000000" WHEN switch_in = "0000" ELSE
		"000101" WHEN switch_in = "0001" ELSE
		"001010" WHEN switch_in = "0010" ELSE
		"001111" WHEN switch_in = "0011" ELSE
		"010100" WHEN switch_in = "0100" ELSE
		"011001" WHEN switch_in = "0101" ELSE
		"011110" WHEN switch_in = "0110" ELSE
		"100011" WHEN switch_in = "0111" ELSE
		"101000" WHEN switch_in = "1000" ELSE
		"101101" WHEN switch_in = "1001" ELSE
		"110010" WHEN switch_in = "1001" ELSE
		"110111" WHEN switch_in = "1001" ELSE
		"111011";

END data_flow;
```

In questo caso, abbiamo ritenuto opportuno utilizzare semplicemente un approccio puramente data-flow.

Di seguito, invece, riportiamo le implementazioni VHDL dei gestori della logica di minuti ed ore.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY logic_handler_m IS
	PORT
	(
		set_switch : IN std_logic_vector(5 DOWNTO 0);
		output     : OUT std_logic_vector(5 DOWNTO 0)
	);
END logic_handler_m;

ARCHITECTURE Behavioral OF logic_handler_m IS

BEGIN
	PROCESS (set_switch)
	VARIABLE max_minute : std_logic_vector(5 DOWNTO 0) := "111011";
	BEGIN
		IF (set_switch > max_minute) THEN
			output <= max_minute;
		ELSE
			output <= set_switch;
```

```
            END IF;
      END PROCESS;

END Behavioral;
```

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY logic_handler_h IS
      PORT
      (
            set_switch : IN std_logic_vector(4 DOWNTO 0);
            output     : OUT std_logic_vector(4 DOWNTO 0)
      );
END logic_handler_h;

ARCHITECTURE Behavioral OF logic_handler_h IS

BEGIN
      PROCESS (set_switch)
      VARIABLE max_hours : std_logic_vector(4 DOWNTO 0) := "10111";
      BEGIN
            IF (set_switch > max_hours) THEN
                  output <= max_hours;
            ELSE
                  output <= set_switch;
            END IF;
      END PROCESS;

END Behavioral;
```

Come è possibile notare, la logica legata all'impostazione del valore massimo di minuti ed ore nel caso in cui si inserisca un valore superiore è pressoché identica. L'unica differenza risiede nel fatto che, ovviamente, il valore massimo dei minuti (*max_minute*) è impostato a 111011 (59 in decimale) mentre quello delle ore (*max_hours*) a 10111 (23 in decimale).

Infine, per poter spiegare ed analizzare con chiarezza ogni componente utilizzata per la gestione del display a 7 segmenti, riportiamo innanzitutto uno schematico relativo alla sua top-module.

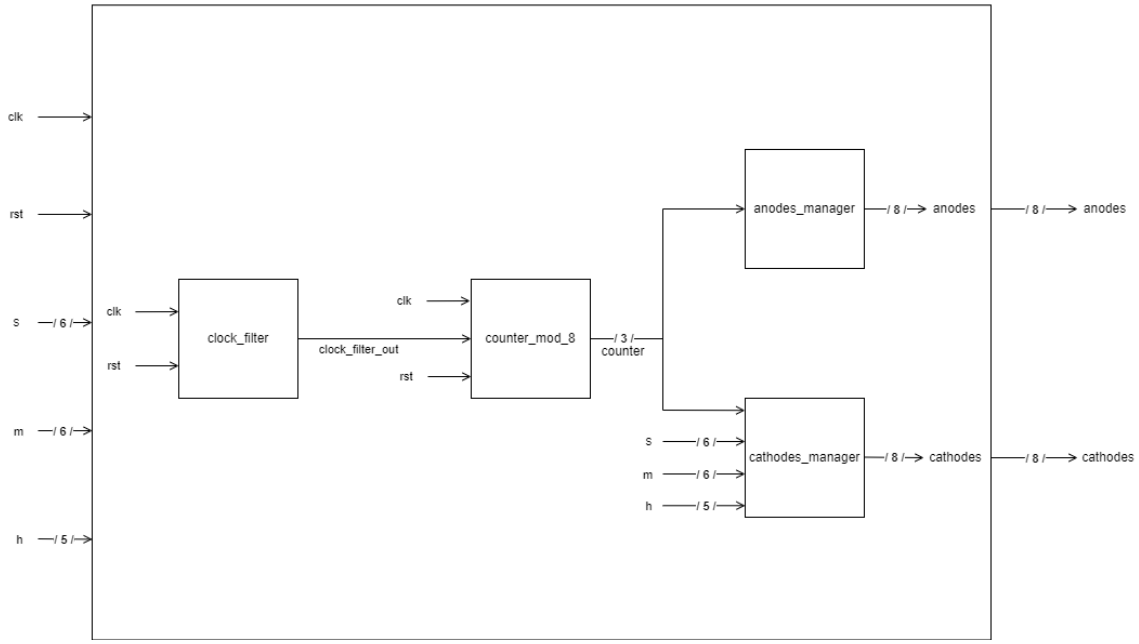


Figura 5.5 - Schematico della top-module del "display_seven_segments"

Nello schematico possiamo distinguere 4 componenti interne, fondamentali per la gestione del display. Le componenti saranno analizzate di seguito con i relativi codici.

La componente *clock_filter* (divisore di frequenza) è un divisore di clock (diverso da quello del cronometro) il cui scopo è quello di effettuare un colpo con una frequenza di 500 Hz. Tale componente sarà necessario per temporizzare correttamente l'alternanza di anodi e catodi del display. Ne riportiamo di seguito il codice VHDL.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY clock_filter IS
	GENERIC
	(
		CLKIN_freq  : INTEGER := 100000000; --clock board 100MHz
		CLKOUT_freq : INTEGER := 500 --frequenza desiderata 500Hz
	);
	PORT
	(
		clock_in  : IN STD_LOGIC;
		reset     : IN STD_LOGIC;
		clock_out : OUT STD_LOGIC -- attenzione: non è un vero clock
ma un impulso che sarà usato come enable
	);
END clock_filter;

ARCHITECTURE Behavioral OF clock_filter IS

	SIGNAL clockfx		: std_logic := '0';
```

```
	CONSTANT count_max_value : INTEGER := CLKIN_freq/(CLKOUT_freq) - 1;

BEGIN
	clock_out <= clockfx;

	count_for_division : PROCESS (clock_in)

		VARIABLE counter   : INTEGER RANGE 0 TO count_max_value := 0;
	BEGIN
		IF rising_edge(clock_in) THEN
			IF (reset = '1') THEN
				counter := 0;
				clockfx <= '0';
			ELSE
				IF counter = count_max_value THEN
					clockfx <= '1';
					counter := 0;
				ELSE
					clockfx <= '0';
					counter := counter + 1;
				END IF;
			END IF;
		END IF;
	END PROCESS;

END Behavioral;
```

La componente *counter_mod8* è un contatore modulo 8 che riceve l'output di *clock_filter*. Ad ogni colpo di clock (ogni 2 ms) aggiorna il suo valore e in base a questo valore sarà poi possibile coordinare anodi e catodi.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY counter_mod8 IS
	PORT
	(
		clock   : IN STD_LOGIC;
		reset   : IN STD_LOGIC;
		enable  : IN STD_LOGIC; --viene dal divisore di frequenza
		counter : OUT STD_LOGIC_VECTOR (2 DOWNTO 0)
	);
END counter_mod8;

ARCHITECTURE Behavioral OF counter_mod8 IS

	SIGNAL c : std_logic_vector (2 DOWNTO 0) := (OTHERS => '0');
BEGIN
	counter <= c;

	counter_process : PROCESS (clock)
```

```
	BEGIN
		IF (rising_edge(clock)) THEN
			IF reset = '1' THEN
				c <= (OTHERS => '1');
			ELSIF enable = '1' THEN
				c <= std_logic_vector(unsigned(c) + 1);
			END IF;
		END IF;
	END PROCESS;

END Behavioral;
```

Il componente *cathodes_manager* indica al display quali segmenti attivare. Riceve in ingresso, oltre al contatore, i valori di secondi, minuti ed ore, restituendo in uscita una stringa binaria. Per questo motivo, all'interno di questo modulo, abbiamo bisogno di un processo di decodifica per dividere le cifre della rappresentazione intera partendo da una rappresentazione binaria. Fatto ciò, quindi, dobbiamo convertire il numero in intero. Dividendolo per 10, il quoziente sarà la cifra più significativa mentre il resto sarà la cifra meno significativa. Il processo viene eseguito per secondi, minuti ed ore. Utilizzando il *counter_mod8*, come detto, si illuminano opportunamente i segmenti giusti permettendo di visualizzare la cifra attesa ogni 2 millisecondi. Di seguito riportiamo, dunque, il codice VHDL di *cathodes_manager*.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY cathodes_manager IS
	PORT
	(
		counter  : IN STD_LOGIC_VECTOR (2 DOWNTO 0);
		s        : IN STD_LOGIC_VECTOR (5 DOWNTO 0); --dato da
mostrare sugli 8 display
		m        : IN STD_LOGIC_VECTOR (5 DOWNTO 0); --dato da
mostrare sugli 8 display
		h        : IN STD_LOGIC_VECTOR (4 DOWNTO 0); --dato da
mostrare sugli 8 display
	cathodes : OUT STD_LOGIC_VECTOR (7 DOWNTO 0)); --sono i 7 catodi più
il punto
END cathodes_manager;

ARCHITECTURE Behavioral OF cathodes_manager IS

	CONSTANT zero		: std_logic_vector(6 DOWNTO 0) :=
"1000000";
	CONSTANT one		: std_logic_vector(6 DOWNTO 0) :=
"1111001";
```

```
	CONSTANT two		: std_logic_vector(6 DOWNTO 0) :=
"0100100";
	CONSTANT three		: std_logic_vector(6 DOWNTO 0) :=
"0110000";
	CONSTANT four		: std_logic_vector(6 DOWNTO 0) :=
"0011001";
	CONSTANT five		: std_logic_vector(6 DOWNTO 0) :=
"0010010";
	CONSTANT six		: std_logic_vector(6 DOWNTO 0) :=
"0000010";
	CONSTANT seven		: std_logic_vector(6 DOWNTO 0) :=
"1111000";
	CONSTANT eight		: std_logic_vector(6 DOWNTO 0) :=
"0000000";
	CONSTANT nine		: std_logic_vector(6 DOWNTO 0) :=
"0010000";
	CONSTANT undefined	: std_logic_vector(6 DOWNTO 0) :=
"1100001";

	CONSTANT s0		: INTEGER := to_integer(unsigned(s)) MOD
10;
	CONSTANT s1		: INTEGER := to_integer(unsigned(s)) / 10;
	CONSTANT m0		: INTEGER := to_integer(unsigned(m)) MOD
10;
	CONSTANT m1		: INTEGER := to_integer(unsigned(m)) / 10;
	CONSTANT h0		: INTEGER := to_integer(unsigned(h)) MOD
10;
	CONSTANT h1		: INTEGER := to_integer(unsigned(h)) / 10;

	SIGNAL cathodes_for_digit : std_logic_Vector(6 DOWNTO 0) := (OTHERS
=> '0');
	SIGNAL nibble		: INTEGER := 0;
	SIGNAL dot		: std_logic := '0'; --stabilisce se il
punto relativo alla cifra visualizzata deve essere acceso o spento
	--nota: dot=1 significa che deve essere acceso, ma il segnale deve
essere negato per andare sui catodi

BEGIN
	-- questo processo multiplexa le cifre da mostrare
	digit_switching : PROCESS (counter)

	BEGIN
		CASE counter IS
			WHEN "000" =>
				nibble <= s0;
				dot    <= '0';
			WHEN "001" =>
				nibble <= s1;
				dot    <= '0';
			WHEN "010" =>
				nibble <= m0;
				dot    <= '1';
			WHEN "011" =>
				nibble <= m1;
				dot    <= '0';
			WHEN "100" =>
```

```
                        nibble <= h0;
                        dot    <= '1';
                WHEN "101" =>
                        nibble <= h1;
                        dot    <= '0';
                WHEN OTHERS =>
                        nibble <= 0;
                        dot    <= '0';
        END CASE;
    END PROCESS;

    seven_segment_decoder_process : PROCESS (nibble)
    BEGIN
        CASE nibble IS
                WHEN 0 => cathodes_for_digit      <= zero;
                WHEN 1 => cathodes_for_digit      <= one;
                WHEN 2 => cathodes_for_digit      <= two;
                WHEN 3 => cathodes_for_digit      <= three;
                WHEN 4 => cathodes_for_digit      <= four;
                WHEN 5 => cathodes_for_digit      <= five;
                WHEN 6 => cathodes_for_digit      <= six;
                WHEN 7 => cathodes_for_digit      <= seven;
                WHEN 8 => cathodes_for_digit      <= eight;
                WHEN 9 => cathodes_for_digit      <= nine;
                WHEN OTHERS => cathodes_for_digit <= (OTHERS => '0');
        END CASE;
    END PROCESS seven_segment_decoder_process;

    cathodes <= (NOT dot) & cathodes_for_digit;

END Behavioral;
```

Infine, utilizziamo *anodes_manager* per decidere invece quale dei digit del display illuminare. Di seguito il codice.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY anodes_manager IS
    PORT
    (
        counter : IN STD_LOGIC_VECTOR (2 DOWNTO 0);
        anodes  : OUT STD_LOGIC_VECTOR (7 DOWNTO 0)
    );
END anodes_manager;

ARCHITECTURE Behavioral OF anodes_manager IS

    SIGNAL anodes_switching : std_logic_vector(7 DOWNTO 0) := (OTHERS =>
'0');

BEGIN
    anodes <= NOT anodes_switching;
```

```
	anodes_process : PROCESS (counter)
	BEGIN
		CASE counter IS
			WHEN "000" =>
				anodes_switching <= "00000001";
			WHEN "001" =>
				anodes_switching <= "00000010";
			WHEN "010" =>
				anodes_switching <= "00000100";
			WHEN "011" =>
				anodes_switching <= "00001000";
			WHEN "100" =>
				anodes_switching <= "00010000";
			WHEN "101" =>
				anodes_switching <= "00100000";
			WHEN "110" =>
				anodes_switching <= "00000000";
			WHEN "111" =>
				anodes_switching <= "00000000";
			WHEN OTHERS =>
				anodes_switching <= (OTHERS => '1');
		END CASE;

	END PROCESS;

END Behavioral;
```

Notiamo che, a seconda del valore di *counter*, le cifre si illuminano una alla volta da destra a sinistra.

5.2.3 Sintesi

Passando alla fase di sintesi, abbiamo dovuto decommentare dal file XDC le righe relative ovviamente al clock, ai 15 switch utilizzati, ai pin del display a 7 segmenti a cui abbiamo associato anodi e catodi e dei due bottoni utilizzati rispettivamente per il set ed il reset. Come sempre, riportiamo di seguito solo le parti decommentate del file.

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


##Switches
set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 } [get_ports {
time_array[0] }]; #IO_L24N_T3_RS0_15 Sch=sw[0]
set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 } [get_ports {
time_array[1] }]; #IO_L3N_T0_DQS_EMCCLK_14 Sch=sw[1]
set_property -dict { PACKAGE_PIN M13   IOSTANDARD LVCMOS33 } [get_ports {
```

```
time_array[2] }]; #IO_L6N_T0_D08_VREF_14 Sch=sw[2]
set_property -dict { PACKAGE_PIN R15   IOSTANDARD LVCMOS33 } [get_ports {
time_array[3] }]; #IO_L13N_T2_MRCC_14 Sch=sw[3]
set_property -dict { PACKAGE_PIN R17   IOSTANDARD LVCMOS33 } [get_ports {
time_array[4] }]; #IO_L12N_T1_MRCC_14 Sch=sw[4]
set_property -dict { PACKAGE_PIN T18   IOSTANDARD LVCMOS33 } [get_ports {
time_array[5] }]; #IO_L7N_T1_D10_14 Sch=sw[5]
set_property -dict { PACKAGE_PIN U18   IOSTANDARD LVCMOS33 } [get_ports {
time_array[6] }]; #IO_L17N_T2_A13_D29_14 Sch=sw[6]
set_property -dict { PACKAGE_PIN R13   IOSTANDARD LVCMOS33 } [get_ports {
time_array[7] }]; #IO_L5N_T0_D07_14 Sch=sw[7]
set_property -dict { PACKAGE_PIN T8    IOSTANDARD LVCMOS18 } [get_ports {
time_array[8] }]; #IO_L24N_T3_34 Sch=sw[8]
set_property -dict { PACKAGE_PIN U8    IOSTANDARD LVCMOS18 } [get_ports {
time_array[9] }]; #IO_25_34 Sch=sw[9]
set_property -dict { PACKAGE_PIN R16   IOSTANDARD LVCMOS33 } [get_ports {
time_array[10] }]; #IO_L15P_T2_DQS_RDWR_B_14 Sch=sw[10]
set_property -dict { PACKAGE_PIN T13   IOSTANDARD LVCMOS33 } [get_ports {
time_array[11] }]; #IO_L23P_T3_A03_D19_14 Sch=sw[11]
set_property -dict { PACKAGE_PIN H6    IOSTANDARD LVCMOS33 } [get_ports {
time_array[12] }]; #IO_L24P_T3_35 Sch=sw[12]
set_property -dict { PACKAGE_PIN U12   IOSTANDARD LVCMOS33 } [get_ports {
time_array[13] }]; #IO_L20P_T3_A08_D24_14 Sch=sw[13]
set_property -dict { PACKAGE_PIN U11   IOSTANDARD LVCMOS33 } [get_ports {
time_array[14] }]; #IO_L19N_T3_A09_D25_VREF_14 Sch=sw[14]

##7 segment display
set_property -dict { PACKAGE_PIN T10   IOSTANDARD LVCMOS33 } [get_ports {
cathodes_out[0] }]; #IO_L24N_T3_A00_D16_14 Sch=ca
set_property -dict { PACKAGE_PIN R10   IOSTANDARD LVCMOS33 } [get_ports {
cathodes_out[1] }]; #IO_25_14 Sch=cb
set_property -dict { PACKAGE_PIN K16   IOSTANDARD LVCMOS33 } [get_ports {
cathodes_out[2] }]; #IO_25_15 Sch=cc
set_property -dict { PACKAGE_PIN K13   IOSTANDARD LVCMOS33 } [get_ports {
cathodes_out[3] }]; #IO_L17P_T2_A26_15 Sch=cd
set_property -dict { PACKAGE_PIN P15   IOSTANDARD LVCMOS33 } [get_ports {
cathodes_out[4] }]; #IO_L13P_T2_MRCC_14 Sch=ce
set_property -dict { PACKAGE_PIN T11   IOSTANDARD LVCMOS33 } [get_ports {
cathodes_out[5] }]; #IO_L19P_T3_A10_D26_14 Sch=cf
set_property -dict { PACKAGE_PIN L18   IOSTANDARD LVCMOS33 } [get_ports {
cathodes_out[6] }]; #IO_L4P_T0_D04_14 Sch=cg
set_property -dict { PACKAGE_PIN H15   IOSTANDARD LVCMOS33 } [get_ports {
cathodes_out[7] }]; #IO_L19N_T3_A21_VREF_15 Sch=dp
set_property -dict { PACKAGE_PIN J17   IOSTANDARD LVCMOS33 } [get_ports {
anodes_out[0] }]; #IO_L23P_T3_FOE_B_15 Sch=an[0]
set_property -dict { PACKAGE_PIN J18   IOSTANDARD LVCMOS33 } [get_ports {
anodes_out[1] }]; #IO_L23N_T3_FWE_B_15 Sch=an[1]
set_property -dict { PACKAGE_PIN T9    IOSTANDARD LVCMOS33 } [get_ports {
anodes_out[2] }]; #IO_L24P_T3_A01_D17_14 Sch=an[2]
set_property -dict { PACKAGE_PIN J14   IOSTANDARD LVCMOS33 } [get_ports {
anodes_out[3] }]; #IO_L19P_T3_A22_15 Sch=an[3]
set_property -dict { PACKAGE_PIN P14   IOSTANDARD LVCMOS33 } [get_ports {
anodes_out[4] }]; #IO_L8N_T1_D12_14 Sch=an[4]
set_property -dict { PACKAGE_PIN T14   IOSTANDARD LVCMOS33 } [get_ports {
anodes_out[5] }]; #IO_L14P_T2_SRCC_14 Sch=an[5]
set_property -dict { PACKAGE_PIN K2    IOSTANDARD LVCMOS33 } [get_ports {
```

```
anodes_out[6] }]; #IO_L23P_T3_35 Sch=an[6]
set_property -dict { PACKAGE_PIN U13   IOSTANDARD LVCMOS33 } [get_ports {
anodes_out[7] }]; #IO_L23N_T3_A02_D18_14 Sch=an[7]

##Buttons
set_property -dict { PACKAGE_PIN C12   IOSTANDARD LVCMOS33 } [get_ports {
rst }]; #IO_L3P_T0_DQS_AD1P_15 Sch=cpu_resetn
set_property -dict { PACKAGE_PIN M18   IOSTANDARD LVCMOS33 } [get_ports {
set }]; #IO_L4N_T0_D05_14 Sch=btnu
```

Notiamo che, avendo utilizzato il bottone *cpu_reset* come reset, abbiamo dovuto implementare una logica "inversa" in quanto il bottone è di tipo 0-attivo.

5.3 Cronometro su FPGA con intertempi

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).

5.3.1 Soluzione

Al fine di salvare (e quindi mostrare) *N* intertempi, la macchina prevede due modalità di visualizzazione: una che ci permette di visualizzare il cronometro e l'altra gli intertempi salvati.

Nella modalità di visualizzazione degli intertempi, il sistema mostra sul display a 7 segmenti gli intertempi mentre nella modalità di visualizzazione del cronometro, la macchina si comporta come nel punto precedente (*5.2*). Durante quest'ultima, possiamo salvare l'intertempo con la pressione di un bottone.

Abbiamo realizzato la memorizzazione degli intertempi con un registro in modo tale che, quando c'è da memorizzare un nuovo intertempo, viene perso il valore salvato "più vecchio" per inserire quello nuovo in testa.

A differenza della specifica della traccia, che richiedeva un ingresso di stop, abbiamo adottato un approccio differente che ci consente di tenere in memoria gli ultimi N intertempi salvati e quindi non è necessaria una sequenza di stop per passare dalla memorizzazione degli intertempi alla loro visualizzazione.

Per la visualizzazione degli intertempi, dunque, si è pensato ad un contatore che, alla pressione di un apposito bottone di visualizzazione, si incrementi in modulo N così da restituire l'i-esimo intertempo all'interno del registro. Questo vuol dire che, nel caso si passi in modalità di visualizzazione degli intertempi senza aver riempito il registro in tutte le sue N locazioni, esso restituirà, in corrispondenza delle posizioni vuote, un valore di default (00:00:00).

La scelta di utilizzare un registro per la memorizzazione ed un contatore per la visualizzazione è dovuta al fatto che abbiamo voluto dare priorità sempre ai tempi salvati più recenti. Il contatore ci consente di visualizzare ogni valore contenuto nel registro. Se avessimo usato un solo contatore per memorizzazione e visualizzazione, non avremmo potuto tenere traccia sia del valore da visualizzare che della posizione meno recente da sovrascrivere.

5.3.2 Realizzazione

Considerando quanto detto nel paragrafo precedente, dunque, il registro *split_time_register* mantiene N intertempi (3 di default) ed inizialmente presenta tutti valori "00:00:00". Sia la memorizzazione che la visualizzazione vengono descritte con un approccio comportamentale. In particolare, abbiamo 2 *process* sul clock: il primo gestisce la memorizzazione perdendo l'intertempo acquisito meno recente, il secondo descrive l'uscita del registro che viene gestita con un contatore modulo N.

Si definisce un tipo *array_n* definito come array di *std_logic_vector* di 17 bit, che viene poi istanziato come *signal* per mantenere gli intertempi.

In ingresso allo *split_time_register* abbiamo il valore dell'intertempo da salvare mentre in uscita il valore del registro puntato dal contatore.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY split_time_register IS
	GENERIC (
		n : INTEGER := 3
	);
	PORT (
		time_value : IN std_logic_vector(16 DOWNTO 0);
		btn_save_clear : IN std_logic;
		btn_view_clear : IN std_logic;
		clk : IN std_logic;
		rst : IN std_logic;
		time_value_out : OUT std_logic_vector(16 DOWNTO 0)
	);
END split_time_register;

ARCHITECTURE Behavioral OF split_time_register IS

	TYPE array_n IS ARRAY (0 TO n - 1) OF std_logic_vector(16 DOWNTO 0);
	SIGNAL split_time : array_n := (OTHERS => (OTHERS => '0'));

BEGIN
	PROCESS (clk)
	VARIABLE count_s : INTEGER := 0;
	BEGIN
		IF rising_edge(clk) THEN
			IF (btn_save_clear = '1') THEN
				FOR iterator IN (n - 1) DOWNTO 1 LOOP
					split_time(iterator) <=
split_time(iterator - 1);
				END LOOP;
				split_time(0) <= time_value;
			END IF;
		END IF;
	END PROCESS;

	PROCESS (clk)
	VARIABLE j : INTEGER := n - 1;
	VARIABLE count_v : INTEGER := 0;
		BEGIN
			IF rising_edge(clk) THEN
				IF (btn_view_clear = '1') THEN
					count_v := count_v + 1;
				END IF;

				IF (count_v <= n - 1) THEN
					time_value_out <= split_time(count_v);
				ELSIF (count_v = n) THEN
					count_v := 0;
				ELSIF (rst = '1') THEN
					time_value_out <= "00000000000000000";
				END IF;
			END IF;
		END PROCESS;
END Behavioral;
```

La memorizzazione e la visualizzazione degli intertempi avvengono tramite la pressione di due bottoni: rispettivamente *btn_save_clear* e *btn_view_clear* ripuliti dai rispettivi debouncer.

A differenza del punto precedente, abbiamo ritenuto opportuno utilizzare il debouncer perché il *process* è sensibile al clock nativo della scheda e non a quello ottenuto tramite il divisore di frequenza e, dunque, necessitiamo di un bottone ripulito per evitare:

- problemi di memorizzazione, in quanto alla pressione del bottone *btn_save_clear* si andrebbero a memorizzare più volte lo stesso valore in più locazioni col rischio di sovrascrivere altri valori già presenti nel registro;

- problemi di visualizzazione, in quanto, nonostante una singola pressione del bottone *btn_view_clear,* il contatore incrementerebbe più volte in maniera indesiderata ed incontrollabile.

Un multiplexer 2:1 controlla l'uscita verso il display. Il multiplexer riceve in ingresso due valori di tempo, il primo è quello del cronometro che, in ogni caso, continua ad aumentare mentre il secondo è proprio l'uscita del registro degli intertempi. Appena si passa dalla modalità di visualizzazione del cronometro a quella di visualizzazione degli intertempi, il display dovrà mostrare il valore della prima posizione puntata all'interno del registro e, alla pressione di un bottone, incrementa circolarmente un contatore per mostrare tutti i valori nel registro.

Di seguito riportiamo uno schematico relativo alla sua top-module.

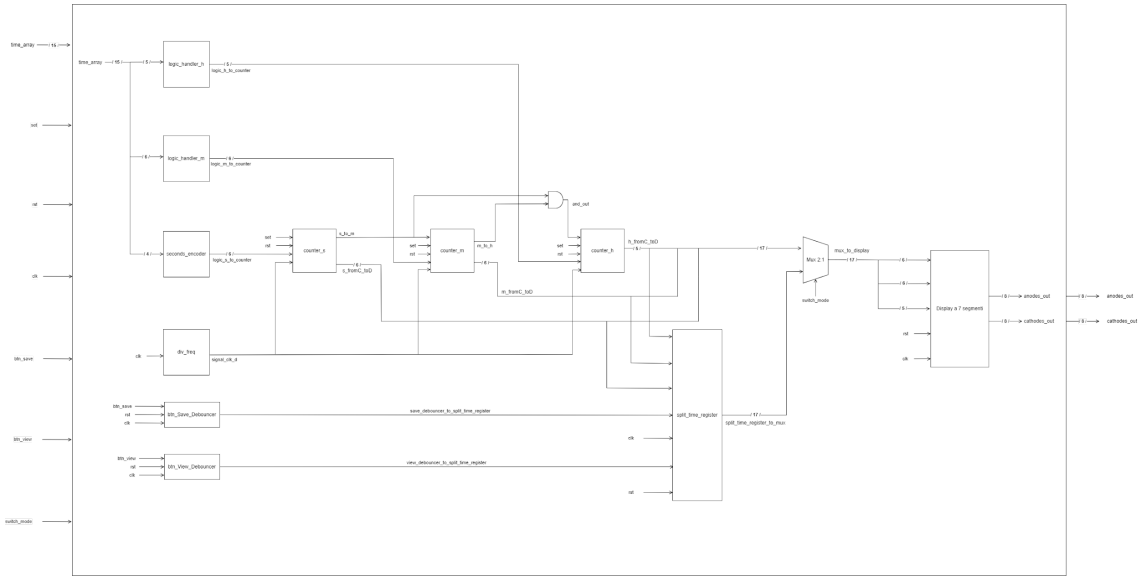


Figura 5.6 - Schematico della top-module

La top-module collega opportunamente tutte le entità, in particolare:

- *split_time_register* in ingresso deve avere le uscite dei singoli contatori di secondi, minuti ed ore e in uscita restituisce un valore di tempo;
- *mux2_1* in ingresso deve avere le uscite dei singoli contatori di secondi, minuti ed ore e l'uscita dello *split_time_register*. Il multiplexer è pilotabile con il segnale di selezione *switch_mode* che rappresenta la modalità di funzionamento (visualizzazione del cronometro o degli intertempi). Ricordiamo che anche durante la modalità di visualizzazione degli intertempi, il cronometro continua a contare. In uscita al multiplexer abbiamo il valore da rappresentare sul display.

Di seguito si riporta l'implementazione della top-module:

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY cronometro IS
      PORT
      (
            set, rst : IN std_logic;
            clk : IN std_logic;
            anodes_out : OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
            cathodes_out : OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
            time_array : IN std_logic_vector(14 DOWNTO 0);
```

```
		btn_save : IN std_logic;
		btn_view : IN std_logic;
		switch_mode : IN std_logic
	);
END cronometro;

ARCHITECTURE Structural OF cronometro IS
	COMPONENT div_freq IS
		GENERIC (clk_period : INTEGER := 10);
		PORT
		(
			clk : IN std_logic;
			clk_d : OUT std_logic
		);
	END COMPONENT;

	COMPONENT counter_s IS
		PORT
		(
			set, rst : IN std_logic;
			set_switch : IN std_logic_vector(5 DOWNTO 0);
			clk_d : IN std_logic;
			flag_max : OUT std_logic;
			output : OUT std_logic_vector(5 DOWNTO 0)
		);
	END COMPONENT;

	COMPONENT counter_m IS
		PORT
		(
			input : IN std_logic;
			set, rst : IN std_logic;
			clk_d : IN std_logic;
			set_switch : IN std_logic_vector(5 DOWNTO 0);
			flag_max : OUT std_logic;
			output : OUT std_logic_vector(5 DOWNTO 0)
		);
	END COMPONENT;

	COMPONENT counter_h IS
		PORT
		(
			input : IN std_logic;
			set, rst : IN std_logic;
			set_switch : IN std_logic_vector(4 DOWNTO 0);
			clk_d : IN std_logic;
			output : OUT std_logic_vector(4 DOWNTO 0)
		);
	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;
                s : IN STD_LOGIC_VECTOR (5 DOWNTO 0);
                m : IN STD_LOGIC_VECTOR (5 DOWNTO 0);
                h : IN STD_LOGIC_VECTOR (4 DOWNTO 0);
                ANODES : OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
                CATHODES : OUT STD_LOGIC_VECTOR (7 DOWNTO 0)
        );
    END COMPONENT;

    COMPONENT seconds_encoder IS
        PORT
        (
                switch_in : IN std_logic_vector(3 DOWNTO 0);
                switch_out : OUT std_logic_vector(5 DOWNTO 0)
        );
    END COMPONENT;

    COMPONENT logic_handler_m IS
        PORT
        (
                set_switch : IN std_logic_vector(5 DOWNTO 0);
                output : OUT std_logic_vector(5 DOWNTO 0)
        );
    END COMPONENT;

    COMPONENT logic_handler_h IS
        PORT
        (
                set_switch : IN std_logic_vector(4 DOWNTO 0);
                output : OUT std_logic_vector(4 DOWNTO 0)
        );
    END COMPONENT;

    COMPONENT split_time_register IS
        GENERIC
        (
                n : INTEGER := 3
        );
        PORT
        (
                time_value : IN std_logic_vector(16 DOWNTO 0);
                btn_save_clear : IN std_logic;
                btn_view_clear : IN std_logic;
                clk : IN std_logic;
                rst : IN std_logic;
                time_value_out : OUT std_logic_vector(16 DOWNTO 0)
        );
    END COMPONENT;

    COMPONENT btn_Save_Debouncer IS
        GENERIC
        (
                CLK_period : INTEGER := 10;
```

```
                btn_noise_time : INTEGER := 6500000
        );
        PORT
        (
                read : IN STD_LOGIC;
                rst : IN STD_LOGIC;
                clk : IN STD_LOGIC;
                read_clear : OUT STD_LOGIC
        );
    END COMPONENT;

    COMPONENT btn_View_Debouncer IS
        GENERIC
        (
                CLK_period : INTEGER := 10;
                btn_noise_time : INTEGER := 6500000
        );
        PORT
        (
                read : IN STD_LOGIC;
                rst : IN STD_LOGIC;
                clk : IN STD_LOGIC;
                read_clear : OUT STD_LOGIC
        );
    END COMPONENT;

    COMPONENT mux2_1 IS
        PORT
        (
                x1, x2 : IN std_logic_vector(16 DOWNTO 0);
                s : IN std_logic;
                y : OUT std_logic_vector(16 DOWNTO 0)
        );
    END COMPONENT;

    SIGNAL signal_clk_d : std_logic;
    SIGNAL s_to_m : std_logic;
    SIGNAL m_to_h : std_logic;
    SIGNAL and_out : std_logic;

    SIGNAL s_fromC_toD : std_logic_vector(5 DOWNTO 0);
    SIGNAL m_fromC_toD : std_logic_vector(5 DOWNTO 0);
    SIGNAL h_fromC_toD : std_logic_vector(4 DOWNTO 0);

    SIGNAL reset_n : std_logic;

    SIGNAL logic_s_to_counter : std_logic_vector(5 DOWNTO 0) :=
"000000";
    SIGNAL logic_m_to_counter : std_logic_vector(5 DOWNTO 0) :=
"000000";
    SIGNAL logic_h_to_counter : std_logic_vector(4 DOWNTO 0) := "00000";

    SIGNAL save_debouncer_to_split_time_register : std_logic;
    SIGNAL view_debouncer_to_split_time_register : std_logic;
    SIGNAL split_time_register_to_mux : std_logic_vector(16 DOWNTO 0);
```

```
	SIGNAL mux_to_display : std_logic_vector(16 DOWNTO 0);

BEGIN
	reset_n <= NOT rst;

	divisore_frequenza : div_freq
		GENERIC MAP
		(
		clk_period => 10
		)
		PORT MAP
		(
			clk => clk,
			clk_d => signal_clk_d
		);

	cnt_s : counter_s
	PORT MAP
	(
		set => set,
		rst => reset_n,
		set_switch => logic_s_to_counter,
		clk_d => signal_clk_d,
		flag_max => s_to_m,
		output => s_fromC_toD
	);

	cnt_m : counter_m
	PORT MAP
	(
		input => s_to_m,
		set => set,
		rst => reset_n,
		set_switch => logic_m_to_counter,
		clk_d => signal_clk_d,
		flag_max => m_to_h,
		output => m_fromC_toD
	);

	and_out <= s_to_m AND m_to_h;

	cnt_h : counter_h
	PORT MAP
	(
		input => and_out,
		set => set,
		rst => reset_n,
		set_switch => logic_h_to_counter,
		clk_d => signal_clk_d,
		output => h_fromC_toD
	);

	display : display_seven_segments
		GENERIC MAP
		(
		CLKIN_freq => 100000000,
```

```
            CLKOUT_freq => 500)
            PORT MAP
            (
                  CLK => CLK,
                  RST => reset_n,
                  s => mux_to_display(5 DOWNTO 0),
                  m => mux_to_display(11 DOWNTO 6),
                  h => mux_to_display(16 DOWNTO 12),
                  ANODES => anodes_out,
                  CATHODES => cathodes_out
            );

                  s_encoder : seconds_encoder
                  PORT MAP
                  (
                        switch_in => time_array(3 DOWNTO 0),
                        switch_out => logic_s_to_counter(5 DOWNTO 0)
      );

      log_handler_m : logic_handler_m
      PORT MAP
      (
            set_switch => time_array(9 DOWNTO 4),
            output => logic_m_to_counter
      );

      log_handler_h : logic_handler_h
      PORT MAP
      (
            set_switch => time_array(14 DOWNTO 10),
            output => logic_h_to_counter
      );

      time_to_save_register : split_time_register
            GENERIC MAP
            (
            n => 3
            )
            PORT MAP
            (
                  time_value(16 DOWNTO 12) => h_fromC_toD,
                  time_value(11 DOWNTO 6) => m_fromC_toD,
                  time_value(5 DOWNTO 0) => s_fromC_toD,
                  btn_save_clear =>
save_debouncer_to_split_time_register,
                  btn_view_clear =>
view_debouncer_to_split_time_register,
                  clk => clk,
                  rst => reset_n,
                  time_value_out => split_time_register_to_mux
            );

      save_Deb : btn_Save_Debouncer
            GENERIC MAP
            (
            CLK_period => 10,
```

```
            btn_noise_time => 6500000
            )
            PORT MAP
            (
                  read => btn_save,
                  rst => reset_n,
                  clk => clk,
                  read_clear => save_debouncer_to_split_time_register
            );

      view_Deb : btn_View_Debouncer
            GENERIC MAP
            (
            CLK_period => 10,
            btn_noise_time => 6500000
            )
            PORT MAP
            (
                  read => btn_view,
                  rst => reset_n,
                  clk => clk,
                  read_clear => view_debouncer_to_split_time_register
            );

      m : mux2_1
      PORT MAP
      (
            x1(16 DOWNTO 12) => h_fromC_toD,
            x1(11 DOWNTO 6) => m_fromC_toD,
            x1(5 DOWNTO 0) => s_fromC_toD,
            x2 => split_time_register_to_mux,
            s => switch_mode,
            y => mux_to_display
      );

END Structural;
```

5.3.3 Sintesi

Rispetto alla sintesi del punto 5.2, abbiamo due bottoni in più per gestire il salvataggio e la visualizzazione degli intertempi. Il cambio di modalità è mappato sul primo switch a sinistra della scheda (*switch[15]*). Come in precedenza, il *time_array* viene mappato sul resto degli switch. La macchina viene correttamente sintetizzata. Di seguito si riporta il file di constraints.

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

```
[get_ports {clk}];


##Switches
set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 } [get_ports {
time_array[0] }]; #IO_L24N_T3_RS0_15 Sch=sw[0]
set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 } [get_ports {
time_array[1] }]; #IO_L3N_T0_DQS_EMCCLK_14 Sch=sw[1]
set_property -dict { PACKAGE_PIN M13   IOSTANDARD LVCMOS33 } [get_ports {
time_array[2] }]; #IO_L6N_T0_D08_VREF_14 Sch=sw[2]
set_property -dict { PACKAGE_PIN R15   IOSTANDARD LVCMOS33 } [get_ports {
time_array[3] }]; #IO_L13N_T2_MRCC_14 Sch=sw[3]
set_property -dict { PACKAGE_PIN R17   IOSTANDARD LVCMOS33 } [get_ports {
time_array[4] }]; #IO_L12N_T1_MRCC_14 Sch=sw[4]
set_property -dict { PACKAGE_PIN T18   IOSTANDARD LVCMOS33 } [get_ports {
time_array[5] }]; #IO_L7N_T1_D10_14 Sch=sw[5]
set_property -dict { PACKAGE_PIN U18   IOSTANDARD LVCMOS33 } [get_ports {
time_array[6] }]; #IO_L17N_T2_A13_D29_14 Sch=sw[6]
set_property -dict { PACKAGE_PIN R13   IOSTANDARD LVCMOS33 } [get_ports {
time_array[7] }]; #IO_L5N_T0_D07_14 Sch=sw[7]
set_property -dict { PACKAGE_PIN T8    IOSTANDARD LVCMOS18 } [get_ports {
time_array[8] }]; #IO_L24N_T3_34 Sch=sw[8]
set_property -dict { PACKAGE_PIN U8    IOSTANDARD LVCMOS18 } [get_ports {
time_array[9] }]; #IO_25_34 Sch=sw[9]
set_property -dict { PACKAGE_PIN R16   IOSTANDARD LVCMOS33 } [get_ports {
time_array[10] }]; #IO_L15P_T2_DQS_RDWR_B_14 Sch=sw[10]
set_property -dict { PACKAGE_PIN T13   IOSTANDARD LVCMOS33 } [get_ports {
time_array[11] }]; #IO_L23P_T3_A03_D19_14 Sch=sw[11]
set_property -dict { PACKAGE_PIN H6    IOSTANDARD LVCMOS33 } [get_ports {
time_array[12] }]; #IO_L24P_T3_35 Sch=sw[12]
set_property -dict { PACKAGE_PIN U12   IOSTANDARD LVCMOS33 } [get_ports {
time_array[13] }]; #IO_L20P_T3_A08_D24_14 Sch=sw[13]
set_property -dict { PACKAGE_PIN U11   IOSTANDARD LVCMOS33 } [get_ports {
time_array[14] }]; #IO_L19N_T3_A09_D25_VREF_14 Sch=sw[14]
set_property -dict { PACKAGE_PIN V10   IOSTANDARD LVCMOS33 } [get_ports {
switch_mode }]; #IO_L21P_T3_DQS_14 Sch=sw[15]

##7 segment display
set_property -dict { PACKAGE_PIN T10   IOSTANDARD LVCMOS33 } [get_ports {
cathodes_out[0] }]; #IO_L24N_T3_A00_D16_14 Sch=ca
set_property -dict { PACKAGE_PIN R10   IOSTANDARD LVCMOS33 } [get_ports {
cathodes_out[1] }]; #IO_25_14 Sch=cb
set_property -dict { PACKAGE_PIN K16   IOSTANDARD LVCMOS33 } [get_ports {
cathodes_out[2] }]; #IO_25_15 Sch=cc
set_property -dict { PACKAGE_PIN K13   IOSTANDARD LVCMOS33 } [get_ports {
cathodes_out[3] }]; #IO_L17P_T2_A26_15 Sch=cd
set_property -dict { PACKAGE_PIN P15   IOSTANDARD LVCMOS33 } [get_ports {
cathodes_out[4] }]; #IO_L13P_T2_MRCC_14 Sch=ce
set_property -dict { PACKAGE_PIN T11   IOSTANDARD LVCMOS33 } [get_ports {
cathodes_out[5] }]; #IO_L19P_T3_A10_D26_14 Sch=cf
set_property -dict { PACKAGE_PIN L18   IOSTANDARD LVCMOS33 } [get_ports {
cathodes_out[6] }]; #IO_L4P_T0_D04_14 Sch=cg
set_property -dict { PACKAGE_PIN H15   IOSTANDARD LVCMOS33 } [get_ports {
cathodes_out[7] }]; #IO_L19N_T3_A21_VREF_15 Sch=dp
set_property -dict { PACKAGE_PIN J17   IOSTANDARD LVCMOS33 } [get_ports {
anodes_out[0] }]; #IO_L23P_T3_FOE_B_15 Sch=an[0]
```

```
set_property -dict { PACKAGE_PIN J18   IOSTANDARD LVCMOS33 } [get_ports {
anodes_out[1] }]; #IO_L23N_T3_FWE_B_15 Sch=an[1]
set_property -dict { PACKAGE_PIN T9    IOSTANDARD LVCMOS33 } [get_ports {
anodes_out[2] }]; #IO_L24P_T3_A01_D17_14 Sch=an[2]
set_property -dict { PACKAGE_PIN J14   IOSTANDARD LVCMOS33 } [get_ports {
anodes_out[3] }]; #IO_L19P_T3_A22_15 Sch=an[3]
set_property -dict { PACKAGE_PIN P14   IOSTANDARD LVCMOS33 } [get_ports {
anodes_out[4] }]; #IO_L8N_T1_D12_14 Sch=an[4]
set_property -dict { PACKAGE_PIN T14   IOSTANDARD LVCMOS33 } [get_ports {
anodes_out[5] }]; #IO_L14P_T2_SRCC_14 Sch=an[5]
set_property -dict { PACKAGE_PIN K2    IOSTANDARD LVCMOS33 } [get_ports {
anodes_out[6] }]; #IO_L23P_T3_35 Sch=an[6]
set_property -dict { PACKAGE_PIN U13   IOSTANDARD LVCMOS33 } [get_ports {
anodes_out[7] }]; #IO_L23N_T3_A02_D18_14 Sch=an[7]

##Buttons
set_property -dict { PACKAGE_PIN C12   IOSTANDARD LVCMOS33 } [get_ports {
rst }]; #IO_L3P_T0_DQS_AD1P_15 Sch=cpu_resetn
set_property -dict { PACKAGE_PIN M18   IOSTANDARD LVCMOS33 } [get_ports {
set }]; #IO_L4N_T0_D05_14 Sch=btnu
set_property -dict { PACKAGE_PIN P17   IOSTANDARD LVCMOS33 } [get_ports {
btn_save }]; #IO_L12P_T1_MRCC_14 Sch=btnl
set_property -dict { PACKAGE_PIN M17   IOSTANDARD LVCMOS33 } [get_ports {
btn_view }]; #IO_L10N_T1_D15_14 Sch=btnr
```

Capitolo 6

Sistema di testing

Risulta fondamentale individuare dispositivi difettosi quanto prima possibile durante la realizzazione di un sistema complesso. Nel malaugurato caso in cui, infatti, un dispositivo difettoso non venga individuato ed entrasse in una catena produttiva si andrebbe incontro a significativi problemi economici. Per ridurre tali perdite economiche è importante eseguire una fase di test quanto più accurata possibile. Si parla della *"regola del 10",* la quale afferma che se un chip difettoso viene individuato si ha una perdita *"x"*. Nel caso in cui non fosse individuato, questo chip verrà poi utilizzato per realizzare una scheda e, in tal caso, se la scheda verrà individuata come difettosa, avremo una perdita di *"10x"*. Se il guasto non venisse individuato, questa scheda verrà utilizzata per realizzare un sistema più complesso. Se il malfunzionamento venisse individuato in fase di testing del sistema, questo verrà scartato con una perdita di *"100x"* e quindi con una perdita economica cento volte superiore. In realtà il caso peggiore da considerare non è questo: infatti, nel momento in cui, a causa di una mancata o non accurata fase di testing, il malfunzionamento non venisse individuato, il sistema complessivo verrà distribuito agli utenti finali. Quando poi il malfunzionamento risulterà evidente, sarà necessario sostituire il sistema "in loco" con costi ancora maggiori, dell'ordine di *"1000x"*. Quanto detto finora ci fa capire l'importanza del testing e, nello specifico, non solo l'importanza di effettuarlo, ma anche che risulti quanto più "efficace" possibile. Affinché un sistema sia facilmente testabile, è necessario che risulti ben "controllabile" ed "osservabile". Infatti accedere direttamente agli ingressi di un sistema (buona controllabilità) permette di stimolare facilmente la macchina e ovviamente ci permette di testarla agevolmente, considerando numerosi casi di test. Così come per gli ingressi, anche le uscite costituiscono una informazione necessaria per verificare il corretto funzionamento di

un dispositivo e poterne valutare e visualizzare facilmente i valori (buona osservabilità) risulta l'unico modo che abbiamo per rilevare l'eventuale presenza di un errore.

Un metodo per fare debugging e testing di una scheda è il Boundary Scan.

Le celle Boundary Scan possono operare in due modalità diverse. Nella loro modalità "funzionale", le celle non hanno alcun effetto sul funzionamento del componente; questa è la modalità nella quale le celle sono configurate quando la scheda lavora normalmente. Nella loro modalità "test", invece, le celle isolano il core del componente dai suoi pin esterni. Configurando, quindi, le celle nella modalità "test", queste possono essere utilizzate per controllare i valori che saranno forzati da un componente JTAG. L'utilizzo di circuiteria aggiuntiva per utilizzare JTAG è necessario e, quindi, esso comporta un costo inevitabilmente aggiuntivo durante il ciclo di sviluppo.

6.1 Sistema di testing

Progettare, implementare in VHDL e verificare mediante simulazione un sistema in grado di testare in maniera automatica una macchina combinatoria M avente 4 ingressi e 3 uscite binarie sottoponendole N ingressi diversi (si considerino una macchina M e un numero di input N a scelta dello studente).

Gli N valori di input per il test devono essere letti da una ROM, in cui essi sono precaricati, in corrispondenza di un segnale *read*. Le N uscite fornite della macchina in corrispondenza di ciascuno degli input devono essere memorizzati in una memoria interna, che deve poter essere svuotata in qualsiasi momento in presenza di un segnale di *reset*.

6.1.1 Soluzione

La macchina combinatoria scelta per il testing confronta il risultato *z* di una operazione tra due ingressi *x* e *y*. L'operazione che determina *z* dipende dal modo, anch'esso in ingresso alla macchina. Se il modo è basso, si farà un'addizione altrimenti una sottrazione. Rispetto a quanto specificato dalla traccia, abbiamo deciso di suddividere

ulteriormente la macchina combinatoria in due parti al fine di rendere il tutto maggiormente modulare: una macchina aritmetica ed un comparatore. Il risultato deve essere confrontato con un valore specifico che arriva dalla ROM, scelto in base all'indirizzo *addr* della memoria. La memoria è stata precedentemente caricata di una serie di valori. I due risultati vanno a questo punto comparati. L'uscita complessiva di questa macchina è data da 3 bit che rappresentano l'esito del testing e, in particolare, la relazione tra risultato effettivo e valore nella ROM.

6.1.2 Realizzazione

Nel sistema di testing abbiamo bisogno di un'entità ROM (per ipotesi precaricata prima dell'esecuzione), di una macchina aritmetica e di un comparatore (entrambe macchine combinatorie) che confronti l'uscita della macchina aritmetica con il valore prelevato dalla ROM.

Una macchina aritmetica, dati in input *in1*, *in2* ed il modo, calcola la somma di *in1* e *in2* se *modo* è 0 e la differenza se *modo* è 1. Si è adottato un approccio comportamentale per descrivere la macchina. Se gli operandi sono rappresentati su 4 bit, il risultato *result* dovrà necessariamente essere rappresentato su 5 bit.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all;
use IEEE.std_logic_unsigned.all;

entity mc is
    Port (
    in1 : in STD_LOGIC_VECTOR(3 downto 0);
    in2 : in STD_LOGIC_VECTOR(3 downto 0);
    modo : in STD_LOGIC;
    result : out std_logic_vector(4 downto 0)
    );
end mc;

architecture Behavioral of mc is

begin
process (modo, in1, in2)
    begin
    if (modo = '0') then
        result<= std_logic_vector(resize(signed(in1), 5) +
```

```
resize(signed(in2), 5));
    elsif (modo = '1') then
        result<= std_logic_vector(resize(signed(in1), 5) -
resize(signed(in2), 5));
    end if;
end process;
end Behavioral;
```

Allo scopo di sintetizzare la ROM con blocchi RAMB e non con LUT si utilizza la dicitura *CONSTANT IS "BLOCK";*.

Sul fronte di salita del clock, quando il segnale di *read_clear* è alto, la ROM restituisce il valore corrispondente alla posizione dell'offset *addr*.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE ieee.numeric_std.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY ROM IS
       PORT
       (
              read_clear          : IN std_logic;
              addr                : IN std_logic_vector (2 DOWNTO 0);
              rst, clk            : IN std_logic;
              outROM              : OUT std_logic_vector(4 DOWNTO 0)
       );
END ROM;

ARCHITECTURE Behavioral OF ROM IS
       TYPE valori IS ARRAY (7 DOWNTO 0) OF signed(4 DOWNTO 0);
       CONSTANT ROMsignal  : valori := (
              "11000", -- -8
              "01111", -- +15
              "00011", -- +3
              "10011", -- -13
              "01010", -- +10
              "11111", -- -1
              "11110", -- -2
              "00000"
       );

       ATTRIBUTE rom_style : STRING;
       ATTRIBUTE rom_style OF ROMsignal : CONSTANT IS "BLOCK";-- block dice
al tool di sintesi di inferire blocchi di RAMB,
       --distributed di usare le LUT

BEGIN
       PROCESS (clk)
       BEGIN
              IF rising_edge(clk) THEN
                     IF (rst = '1') THEN
```

```
                        outROM <= std_logic_vector(ROMsignal(0));
                    ELSIF (read_clear = '1') THEN
                        outROM <=
std_logic_vector(ROMsignal(conv_integer(addr)));
                    END IF;
            END IF;
        END PROCESS;
END Behavioral;
```

Un'ulteriore macchina combinatoria addetta alla comparazione, confronta il risultato della macchina aritmetica *calculated_value* con il valore prelevato dalla ROM *rom_value*. Restituendo 3 segnali da un bit per denotare la relazione (*greater*, *equal*, *less*) tra i valori di ingresso.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE ieee.numeric_std.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY comparatore IS
        PORT
        (
                calculated_value  : IN std_logic_vector(4 DOWNTO 0);
                rom_value         : IN std_logic_vector(4 DOWNTO 0);
                g                 : OUT std_logic;
                e                 : OUT std_logic;
                l                 : OUT std_logic
        );
END comparatore;

ARCHITECTURE Behavioral OF comparatore IS
        SIGNAL calculated_value_signed : signed(4 DOWNTO 0);
BEGIN
        PROCESS (calculated_value, rom_value)
        BEGIN
                IF (calculated_value_signed > signed(rom_value)) THEN
                        g <= '1';
                        e <= '0';
                        l <= '0';
                ELSIF (calculated_value_signed = signed(rom_value)) THEN
                        g <= '0';
                        e <= '1';
                        l <= '0';
                ELSIF (calculated_value_signed < signed(rom_value)) THEN
                        g <= '0';
                        e <= '0';
                        l <= '1';
                END IF;

        END PROCESS;
        calculated_value_signed <= signed(calculated_value);
END Behavioral;
```

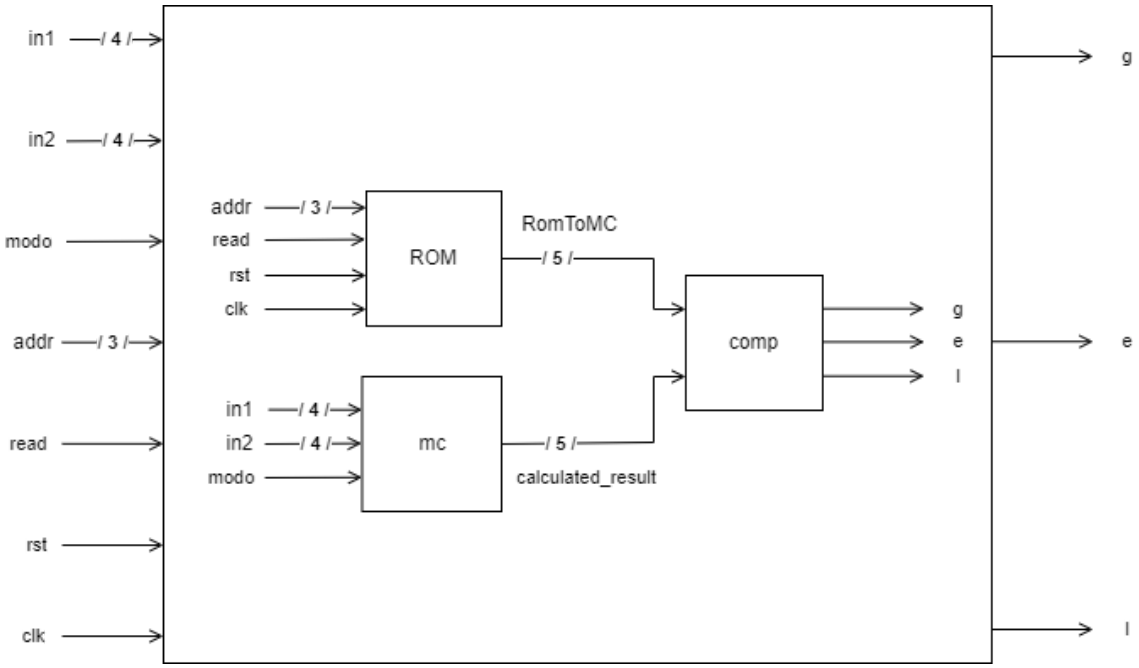


Figura 6.1 - Schematico della top-module

In ultimo, la top module riceve in ingresso *in1*, *in2* e *modo* necessari per eseguire l'operazione aritmetica, *addr* per conoscere la locazione di memoria della ROM del valore col quale effettuare il confronto e *read* per sapere quando leggere il valore puntato da *addr.*

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY Test_System IS
	PORT
	(
		in1	: IN STD_LOGIC_VECTOR(3 DOWNTO 0);
		in2	: IN STD_LOGIC_VECTOR(3 DOWNTO 0);
		modo	: IN STD_LOGIC;
		addr	: IN STD_LOGIC_VECTOR(2 DOWNTO 0);
		read	: IN STD_LOGIC;
		rst	: IN STD_LOGIC;
		clk	: IN STD_LOGIC;
		g	: OUT STD_LOGIC;
		e	: OUT STD_LOGIC;
		l	: OUT STD_LOGIC
	);
END Test_System;
```

```
ARCHITECTURE Structural OF Test_System IS

	COMPONENT rom
		PORT
		(
			addr		: IN STD_LOGIC_VECTOR(2 DOWNTO 0);
			read_clear	: IN STD_LOGIC;
			rst		: IN STD_LOGIC;
			clk		: IN STD_LOGIC;
			outROM		: OUT std_logic_vector(4 DOWNTO 0)
		);
	END COMPONENT;

	COMPONENT comparatore
		PORT
		(
			calculated_value	: IN std_logic_vector(4 DOWNTO 0);
			rom_value		: IN std_logic_vector(4 DOWNTO 0);
			g			: OUT std_logic;
			e			: OUT std_logic;
			l			: OUT std_logic
		);
	END COMPONENT;

	COMPONENT mc IS
		PORT
		(
			in1	: IN STD_LOGIC_VECTOR(3 DOWNTO 0);
			in2	: IN STD_LOGIC_VECTOR(3 DOWNTO 0);
			modo	: IN STD_LOGIC;
			result	: OUT std_logic_vector(4 DOWNTO 0)
		);
	END COMPONENT;

	SIGNAL RomToMC : std_logic_vector(4 DOWNTO 0);
	SIGNAL calculated_result : std_logic_vector(4 DOWNTO 0);
BEGIN
	mem : rom
	PORT MAP
	(
		addr		=> addr,
		read_clear	=> read,
		rst		=> rst,
		clk		=> clk,
		outROM		=> RomToMC
	);

	comp : comparatore
	PORT MAP
	(
		calculated_value	=> calculated_result,
		rom_value		=> RomToMC,
		g			=> g,
		e			=> e,
		l			=> l
	);
```

```
        MacchinaCombinatoria : mc
        PORT MAP
        (
                in1     => in1,
                in2     => in2,
                modo    => modo,
                result  => calculated_result
        );
END Structural;
```

6.1.3 Simulazione

Per verificare il corretto funzionamento del sistema abbiamo deciso di effettuare una serie di test che abbiamo ritenuto essere significativi. Riportiamo di seguito il codice del testbench realizzato, il grafico ottenuto dalla simulazione ed una breve spiegazione.

```
stimulus : PROCESS
BEGIN
        WAIT FOR 100ns;
        rst  <= '0';
        read <= '0';
        WAIT FOR 10 ns;
        in1  <= "0001";
        in2  <= "0010";
        modo <= '0';
        addr <= "101"; -- risultato = 3, rom = 3 => equals
        read <= '1';
        WAIT FOR 10 ns;
        read <= '0';
        WAIT FOR 10 ns;
        modo <= '1';
        addr <= "001";
        in1  <= "0010";
        in2  <= "0011"; -- risultato = -1, rom = -2 => greater
        read <= '1';
        WAIT FOR 10 ns;
        read <= '0';
        WAIT FOR 10 ns;
        addr <= "111";
        modo <= '0';
        in1  <= "1010";
        in2  <= "0010"; -- risultato = -4, rom = -1 => less
        addr <= "010";
        read <= '1';
        WAIT FOR 10 ns;
        read <= '0';
        WAIT FOR 100ns;
        WAIT;
END PROCESS;
```

```
clockproc : PROCESS
BEGIN
        clk <= '0';
        WAIT FOR clk_period/2;
        clk <= '1';
        WAIT FOR clk_period/2;
END PROCESS;
```

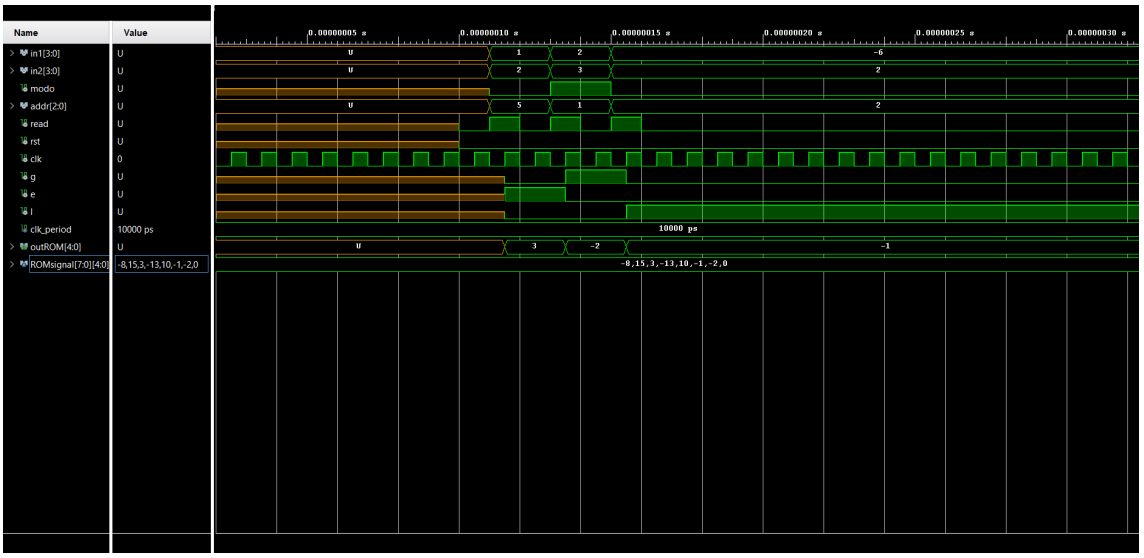


Figura 6.2 - Grafico della simulazione

La fase di simulazione è andata a buon fine, fornendo i risultati attesi. Come è possibile notare dal grafico, infatti, inserendo un valore di ingresso *in1* pari a +1 ed un valore di ingresso *in2* pari a +2 ed impostando il valore del modo a 0 (come spiegato, in questo modo effettuiamo la somma di *in1* e *in2*) confrontiamo il valore 3 (2+1) con il valore presente all'interno del sesto indirizzo della ROM (*addr* impostato a 5) ossia 3 anche in questo caso. Per comodità, i valori presenti all'interno della ROM sono visualizzabili tramite l'apposito segnale *ROMsignal* direttamente nel grafico. Come ci aspettavamo, in questo caso otteniamo un valore di uscita di *e* (equal) alto.

Allo stesso modo avviene per i due casi successivi. Nel primo caso, alziamo il valore del modo andando così ad effettuare la sottrazione di *in1* (valore 2) e *in2* (valore 3) ottenendo -1 e lo andiamo a confrontare con il valore presente nella seconda locazione di memoria della ROM (*addr* impostato a 1 con valore -2). L'uscita che si alza è correttamente *g* (greater). Infine, nell'ultimo caso valutato, confrontiamo il valore -4

(ottenuto dalla somma di -6 e +2) con quello presente nella terza locazione di memoria (*addr* impostato a 2 con valore -1). L'uscita alta diventa correttamente *l* (less).

6.1.4 Sintesi

Per poter sintetizzare il sistema realizzato sulla board, abbiamo deciso di aggiungere - rispetto a quanto descritto finora - un debouncer che potesse gestire opportunamente le eventuali oscillazioni di un bottone che utilizzeremo per poter leggere l'indirizzo della ROM che andremo ad indicare con una serie di switch. Anche il modo ed i due ingressi *in1* e *in2* sono associati agli switch. Le uscite *l* (less), *e* (equal) e *g* (greater) sono associati, invece, a 3 LED.

Di seguito riportiamo il codice XDC appena descritto.

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


##Switches
set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 } [get_ports {
in1[0] }]; #IO_L24N_T3_RS0_15 Sch=sw[0]
set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 } [get_ports {
in1[1] }]; #IO_L3N_T0_DQS_EMCCLK_14 Sch=sw[1]
set_property -dict { PACKAGE_PIN M13   IOSTANDARD LVCMOS33 } [get_ports {
in1[2] }]; #IO_L6N_T0_D08_VREF_14 Sch=sw[2]
set_property -dict { PACKAGE_PIN R15   IOSTANDARD LVCMOS33 } [get_ports {
in1[3] }]; #IO_L13N_T2_MRCC_14 Sch=sw[3]
set_property -dict { PACKAGE_PIN R17   IOSTANDARD LVCMOS33 } [get_ports {
in2[0] }]; #IO_L12N_T1_MRCC_14 Sch=sw[4]
set_property -dict { PACKAGE_PIN T18   IOSTANDARD LVCMOS33 } [get_ports {
in2[1] }]; #IO_L7N_T1_D10_14 Sch=sw[5]
set_property -dict { PACKAGE_PIN U18   IOSTANDARD LVCMOS33 } [get_ports {
in2[2] }]; #IO_L17N_T2_A13_D29_14 Sch=sw[6]
set_property -dict { PACKAGE_PIN R13   IOSTANDARD LVCMOS33 } [get_ports {
in2[3] }]; #IO_L5N_T0_D07_14 Sch=sw[7]
set_property -dict { PACKAGE_PIN T8    IOSTANDARD LVCMOS18 } [get_ports {
modo }]; #IO_L24N_T3_34 Sch=sw[8]
set_property -dict { PACKAGE_PIN U8    IOSTANDARD LVCMOS18 } [get_ports {
addr[0] }]; #IO_25_34 Sch=sw[9]
set_property -dict { PACKAGE_PIN R16   IOSTANDARD LVCMOS33 } [get_ports {
addr[1] }]; #IO_L15P_T2_DQS_RDWR_B_14 Sch=sw[10]
set_property -dict { PACKAGE_PIN T13   IOSTANDARD LVCMOS33 } [get_ports {
addr[2] }]; #IO_L23P_T3_A03_D19_14 Sch=sw[11]

## LEDs
set_property -dict { PACKAGE_PIN H17   IOSTANDARD LVCMOS33 } [get_ports { g
```

```
}]; #IO_L18P_T2_A24_15 Sch=led[0]
set_property -dict { PACKAGE_PIN K15   IOSTANDARD LVCMOS33 } [get_ports { e
}]; #IO_L24P_T3_RS1_15 Sch=led[1]
set_property -dict { PACKAGE_PIN J13   IOSTANDARD LVCMOS33 } [get_ports { l
}]; #IO_L17N_T2_A25_15 Sch=led[2]

##Buttons
set_property -dict { PACKAGE_PIN C12   IOSTANDARD LVCMOS33 } [get_ports {
rst }]; #IO_L3P_T0_DQS_AD1P_15 Sch=cpu_resetn
set_property -dict { PACKAGE_PIN M18   IOSTANDARD LVCMOS33 } [get_ports {
read }]; #IO_L4N_T0_D05_14 Sch=btnu
```

Per completezza, riportiamo di seguito anche lo schematico della top-module ottenuta con l'aggiunta anche dell'apposito debouncer per il bottone.

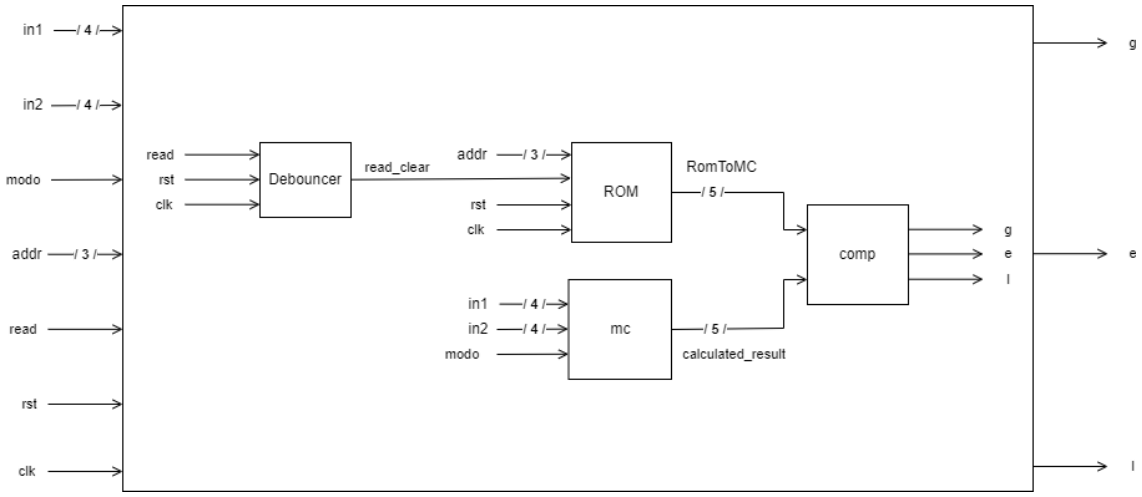


Figura 6.3 - Schematico della top-module con debouncer

Capitolo 7

Protocolli di comunicazione

Per consentire la comunicazione tra entità differenti, è necessario introdurre una serie di protocolli per regolamentare tale comunicazione. Possiamo classificare i protocolli in tre categorie:

- protocolli sincroni;
- protocolli asincroni;
- protocolli semisincroni.

Nei protocolli sincroni le unità coinvolte nella comunicazione lavorano con lo stesso riferimento temporale, noto e condiviso. Notiamo che le due entità, in questo caso, non comunicano in maniera diretta tra loro: una volta avviata la comunicazione, le loro azioni sono tutte motivate dall'esistenza del riferimento temporale comune. Viene, dunque, aggiunto un segnale di *strobe* per permettere all'entità "master" della comunicazione di notificare l'altra entità dell'inizio della comunicazione stessa. In tale tipo di protocollo, uno dei possibili problemi è che le attività possono iniziare a sfasarsi e questo dovrà essere opportunamente gestito.

I protocolli asincroni, invece, non sono basati sul concetto di *tempo* bensì su quello di *evento*. Le entità, infatti, comunicano alla ricezione di eventi e finché non si verificano eventi, non trasmettono. Gli eventi si dividono in eventi di richiesta - detti *comandi* - ed eventi di risposta - detti *stati* - in quanto comunicano lo stato in cui si trova la macchina che riceve il dato. Un tipo di protocollo asincrono è quello di **handshaking**. In questo tipo di protocollo, la prima entità manda l'informazione, poi comunica alla seconda entità l'avvenuto invio del dato ed infine la seconda entità comunica alla prima l'avvenuta ricezione, terminando così la comunicazione. Un altro protocollo asincrono è l'**handshaking completo**. In questo caso viene aggiunto anche un quarto messaggio: la prima entità manda il dato, poi manda il messaggio di avvio della comunicazione; la seconda entità risponde con un messaggio di essere pronta per ricevere il dato ed

infine comunica alla prima entità dell'avvenuta ricezione. Lo scopo di tale tipo di protocollo è quello di far avere alla prima entità subito una risposta: se, infatti, la seconda entità dovesse essere più lenta della prima nel ricevere il dato, allora la prima potrebbe essere rallentata in attesa di sapere se la seconda ha correttamente ricevuto il dato inviato. In conclusione, dunque, il numero minimo di fili necessari per consentire una comunicazione asincrona è 3. Possiamo aggiungere un quarto filo per ottenere un protocollo di handshaking completo. È possibile dimostrare che aggiungere ulteriori fili non apporta alcun miglioramento al sistema complessivo in quanto non si arriverebbe mai a convergenza in quanto si tratta di un sistema distribuito in cui un'entità non può conoscere lo stato dell'altra. Questo è dovuto all'assenza di sincronizzazione, caratteristica chiave dei protocolli asincroni.

Infine, i protocolli semisincroni si comportano come i protocolli sincroni a meno che non si alzi un flag. Come detto, infatti, in un protocollo sincrono, le due entità lavorano con lo stesso riferimento temporale ma potrebbero verificarsi delle situazioni di sfasamento. Se, però, per qualche motivo una delle due entità non riesce ad essere pronta quando dovrebbe per proseguire la comunicazione, alza il flag in modo da sospendere il protocollo.

7.1 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,..,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.

7.1.1 Soluzione

Per poter definire una comunicazione gestita tramite un protocollo asincrono di handshaking, abbiamo innanzitutto progettato i due sistemi che devono effettivamente realizzare tale comunicazione.

Abbiamo pensato di far iniziare la comunicazione tramite un apposito segnale. Nel pieno rispetto dei concetti teorici relativi al protocollo di handshaking (non interallacciato), l'intera comunicazione viene gestita tramite 3 segnali: uno per l'invio del dato, uno per consentire al primo sistema di notificare al secondo sistema di scandire la trasmissione/ricezione delle stringhe e l'ultimo per segnalare l'avvenuta ricezione.

Internamente, entrambi i sistemi sono dotati di un'unità di controllo e di un'unità operativa costituita da un contatore ed una ROM. In più, il secondo sistema è dotato - nella parte operativa - anche di un sistema per effettuare la somma e di un registro dove salvare i risultati ottenuti.

Abbiamo deciso, inoltre, di implementare il sistema in maniera interamente generica tramite l'utilizzo del costrutto *generics*, in modo tale da avere la massima flessibilità possibile sui dati da gestire.

7.1.2 Realizzazione

Partiamo dalla descrizione del sistema A. Come detto, abbiamo realizzato tale sistema suddividendolo in unità operativa ed unità di controllo. Riportiamo di seguito il relativo schematico che evidenzia proprio la distinzione tra le due parti che compongono il sistema.

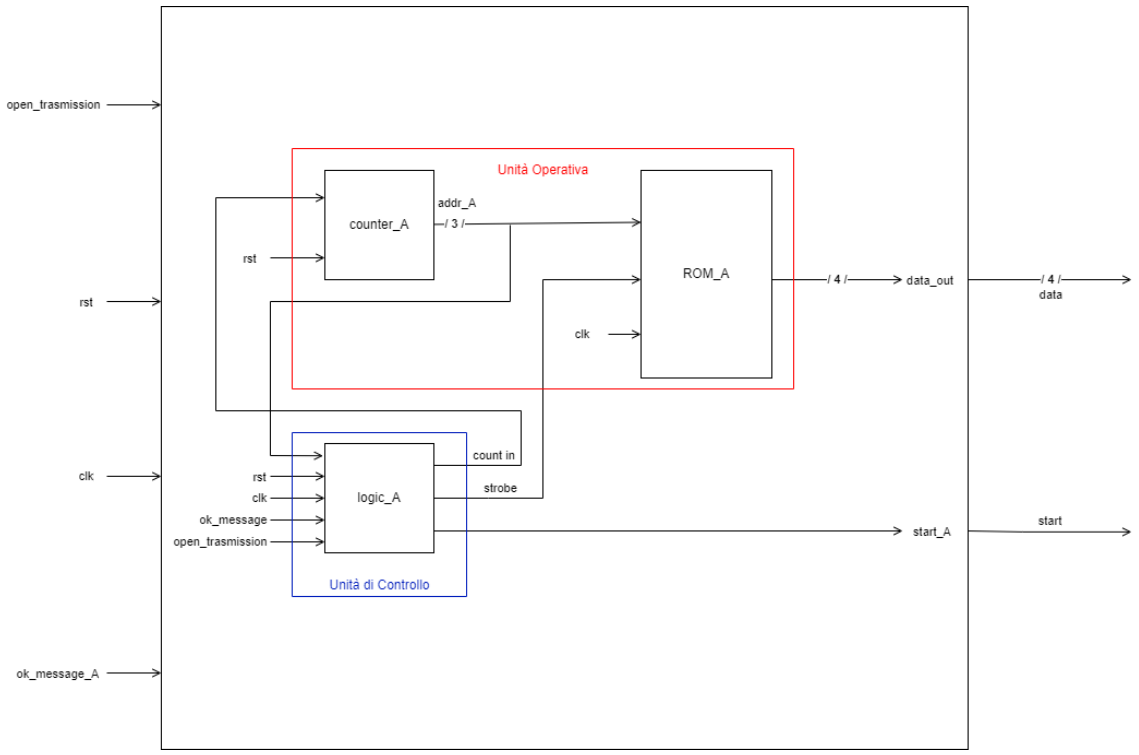


Figura 7.1 - Sistema A

L'unità di controllo (*logic_A*) gestisce il funzionamento di tale sistema ed è stata implementata tramite la realizzazione di un automa a stati finiti che riportiamo di seguito.

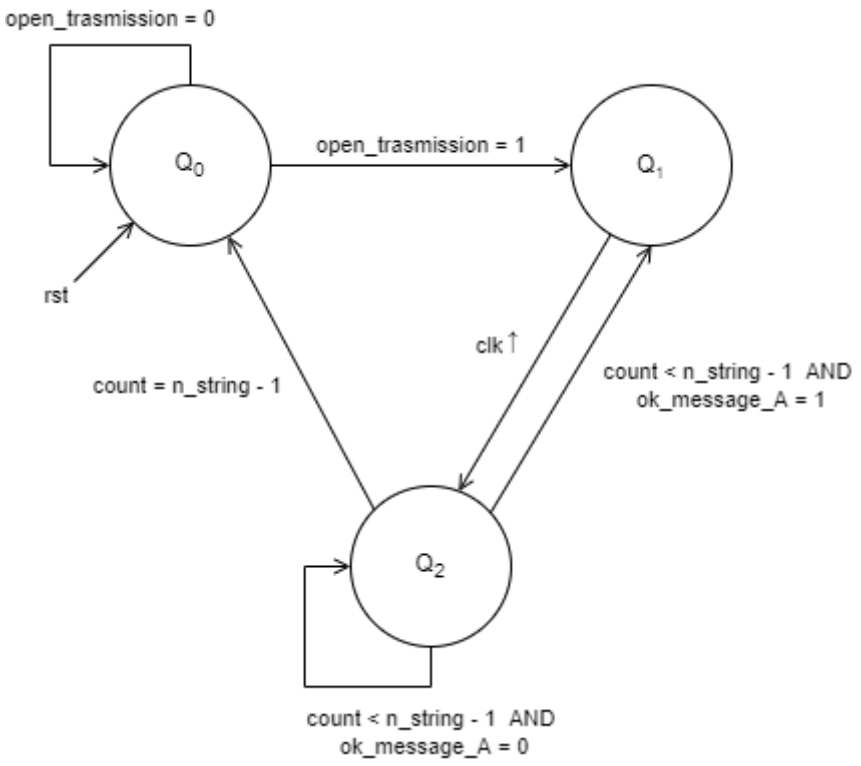


Figura 7.2 - Automa a stati finiti del componente logic_A

L'unità di controllo (implementata tramite automa di Moore) evolve tra i vari stati per fornire alle componenti dell'unità operativa i segnali di abilitazione e strobe per effettuare nell'ordine corretto le operazioni richieste dalla traccia. Inizialmente, la macchina si trova nello stato q_0 e permarrà in tale stato finché la comunicazione non sarà avviata tramite il segnale *open_trasmission*. In seguito, all'avvenuta ricezione del segnale *open_trasmission,* la macchina evolve nello stato q_1 che alza il valore di uscita di *count_in* diretto al contatore per incrementare il valore attualmente segnato dal contatore stesso. Al successivo colpo di clock, la macchina passa nello stato q_2 in cui riporta a 0 il valore di *count_in* ed alza il valore di *strobe* e di *start*. Il segnale di *strobe* viene utilizzato per comunicare alla ROM di inviare al sistema B il dato contenuto nell'indirizzo attualmente puntato dal valore del contatore. Il segnale di *start*, invece, viene mandato direttamente al sistema B - ed in particolare alla sua unità di controllo - affinché possa mantenere il riferimento dell'indice *i* (come richiesto dalla traccia) per le successive operazioni da effettuare. Quando il conteggio sarà arrivato al suo valore massimo, la macchina tornerà nello stato q_0 altrimenti in base al valore del segnale

ok_message_A, la macchina evolverà nello stato q_1 (il sistema B ha ricevuto correttamente il dato, l'ha comunicato ad A che quindi può incrementare il contatore ed inviare un nuovo dato) o permarrà in q_2 (il sistema B non ha ancora prelevato il dato e quindi il sistema A attende che ciò avvenga).

Riportiamo di seguito anche il codice VHDL realizzato.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.math_real.ALL;
USE IEEE.numeric_std.ALL;
USE IEEE.std_logic_unsigned.ALL;

ENTITY logic_A IS
	GENERIC
	(
		n_string : INTEGER := 8;
		m_bit    : INTEGER := 4
	);
	PORT
	(
		open_trasmission : IN std_logic; --0
		rst              : IN std_logic;
		clk              : IN std_logic;
		ok_message_A     : IN std_logic; --3
		count            : IN
std_logic_vector(INTEGER(ceil(log2(real(n_string)))) - 1 DOWNTO 0);
		start_A          : OUT std_logic; --2
		strobe           : OUT std_logic;
		count_in         : OUT std_logic
	);
END logic_A;

ARCHITECTURE Behavioral OF logic_A IS

	TYPE stato IS (q0, q1, q2);
	SIGNAL current_state, next_state : stato;

BEGIN
	PROCESS (clk)
	BEGIN
		IF rising_edge(clk) THEN
			IF (rst = '1') THEN
				current_state <= q0;
			ELSE
				current_state <= next_state;
			END IF;
		END IF;
	END PROCESS;

	PROCESS (current_state, open_trasmission, ok_message_A, count)
		BEGIN
			CASE current_state IS
```

```
                            WHEN q0 =>
                                   count_in <= '0';
                                   strobe   <= '0';
                                   start_A  <= '0';

                                   IF (open_trasmission = '1') THEN
                                          next_state <= q1;
                                   END IF;
                            WHEN q1 =>
                                   count_in   <= '1';
                                   strobe     <= '0';
                                   start_A    <= '0';
                                   next_state <= q2;
                            WHEN q2 =>
                                   count_in <= '0';
                                   strobe   <= '1';
                                   start_A  <= '1';
                                   IF (count =
std_logic_vector(to_unsigned(n_string - 1,
INTEGER(ceil(log2(real(n_string))))))) THEN
                                          next_state <= q0;
                                   ELSE
                                          IF (ok_message_A = '0') THEN
                                                 next_state <= q2;
                                          ELSE
                                                 next_state <= q1;
                                          END IF;
                                   END IF;
                     END CASE;
              END PROCESS;

END Behavioral;
```

Di seguito riportiamo il codice VHDL del contatore.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.math_real.ALL;
USE IEEE.numeric_std.ALL;
USE IEEE.std_logic_unsigned.ALL;

ENTITY counter IS
       GENERIC
       (
              n_string : INTEGER := 8;
              m_bit    : INTEGER := 4
       );
       PORT
       (
              count_in       : IN std_logic;
              rst            : IN std_logic;
              index_selector : OUT
std_logic_vector(INTEGER(ceil(log2(real(n_string)))) - 1 DOWNTO 0)
       );
END counter;
```

```
ARCHITECTURE Behavioral OF counter IS

BEGIN
	PROCESS (count_in, rst)
	VARIABLE cnt     : INTEGER := 0;
	VARIABLE max_cnt : INTEGER := n_string;
	BEGIN
		IF (count_in = '1') THEN --con count_in=1 la comunicazione
inizia.
			index_selector <= std_logic_vector(to_unsigned(cnt,
INTEGER(ceil(log2(real(n_string))))));
			cnt := cnt + 1;
			IF (cnt = max_cnt) THEN --quando il conteggio arriva al
massimo termina la comunicazione
				cnt := 0;
			END IF;
		ELSIF (rst = '1') THEN
			cnt := 0;
		END IF;
	END PROCESS;

END Behavioral;
```

L'uscita *index_selector* del contatore ci permette di puntare alla giusta locazione di memoria della ROM (la *i* della traccia).

Di seguito riportiamo, inoltre, il codice implementato per la realizzazione della ROM del sistema A.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.numeric_std.ALL;
USE IEEE.std_logic_unsigned.ALL;
USE IEEE.math_real.ALL;

ENTITY ROM_A IS
	GENERIC
	(
		n_string : INTEGER := 8;
		m_bit    : INTEGER := 4
	);
	PORT
	(
		addr_i   : IN
std_logic_vector(INTEGER(ceil(log2(real(n_string)))) - 1 DOWNTO 0);
		strobe   : IN std_logic;
		clk      : IN std_logic;
		data_out : OUT std_logic_vector(m_bit - 1 DOWNTO 0)
	);
END ROM_A;

ARCHITECTURE Behavioral OF ROM_A IS
```

```
	TYPE valori IS ARRAY (n_string - 1 DOWNTO 0) OF unsigned(m_bit - 1
DOWNTO 0);
	CONSTANT ROMsignal : valori := (
		"1100",
		"0111",
		"0011",
		"0001",
		"0010",
		"1111",
		"1110",
		"0100"
	);

	ATTRIBUTE rom_style              : STRING;
	ATTRIBUTE rom_style OF ROMsignal : CONSTANT IS "BLOCK";

BEGIN
	PROCESS (clk)
	BEGIN
		IF rising_edge(clk) THEN
			IF (strobe = '1') THEN
				data_out <=
std_logic_vector(ROMsignal(conv_integer(addr_i)));
			END IF;
		END IF;
	END PROCESS;
END Behavioral;
```

Come è possibile notare, all'arrivo del segnale di *strobe* alto la ROM manda in uscita il valore contenuto nell'indirizzo puntato da *addr_i*.

Infine, riportiamo il codice realizzato per l'entità *"system_A"* con approccio strutturale che presenta dichiarazioni e port map delle istanze dei componenti che la costituiscono.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.math_real.ALL;

ENTITY system_A IS
	GENERIC
	(
		n_string : INTEGER := 8;
		m_bit    : INTEGER := 4
	);
	PORT
	(
		open_trasmission : IN std_logic; --0
		rst              : IN std_logic;
		clk              : IN std_logic;
		ok_message_A     : IN std_logic; --3
		data_out         : OUT std_logic_vector(m_bit - 1 DOWNTO 0);
```

```
--1
            start_A          : OUT std_logic --2
      );
END system_A;

ARCHITECTURE Structural OF system_A IS

      SIGNAL strobe   : std_logic;
      SIGNAL addr_A   :
std_logic_vector(INTEGER(ceil(log2(real(n_string)))) - 1 DOWNTO 0);
      SIGNAL count_in : std_logic;

      COMPONENT logic_A IS
            PORT
            (
                  open_trasmission : IN std_logic; --0
                  rst              : IN std_logic;
                  clk              : IN std_logic;
                  ok_message_A     : IN std_logic; --3
                  count            : IN
std_logic_vector(INTEGER(ceil(log2(real(n_string)))) - 1 DOWNTO 0);
                  start_A          : OUT std_logic; --2
                  strobe           : OUT std_logic;
                  count_in         : OUT std_logic
            );
      END COMPONENT;

      COMPONENT counter IS
            GENERIC
            (
                  n_string : INTEGER := 8;
                  m_bit    : INTEGER := 4
            );
            PORT
            (
                  count_in       : IN std_logic;
                  rst            : IN std_logic;
                  index_selector : OUT
std_logic_vector(INTEGER(ceil(log2(real(n_string)))) - 1 DOWNTO 0) --la i
della traccia
            );
      END COMPONENT;

      COMPONENT ROM_A IS
            GENERIC
            (
                  n_string : INTEGER := 8;
                  m_bit    : INTEGER := 4
            );
            PORT
            (
                  addr_i   : IN
std_logic_vector(INTEGER(ceil(log2(real(n_string)))) - 1 DOWNTO 0);
                  strobe   : IN std_logic;
                  clk      : IN std_logic;
                  data_out : OUT std_logic_vector(m_bit - 1 DOWNTO 0)
```

```
            );
      END COMPONENT;

BEGIN
      logic_unit_A : logic_A
      PORT MAP
      (
            open_trasmission => open_trasmission,
            rst              => rst,
            clk              => clk,
            ok_message_A     => ok_message_A,
            count            => addr_A,
            start_A          => start_A,
            strobe           => strobe,
            count_in         => count_in
      );

      counter_A : counter
            GENERIC MAP
            (
            n_string => 8,
            m_bit    => 4
            )
            PORT MAP
            (
                  count_in       => count_in,
                  rst            => rst,
                  index_selector => addr_A
            );

                  ROM_op1 : ROM_A
                        GENERIC MAP
                        (
                        n_string => 8,
                        m_bit    => 4
                        )
                        PORT MAP
                        (
                              addr_i   => addr_A,
                              strobe   => strobe,
                              clk      => clk,
                              data_out => data_out
                        );

END Structural;
```

Il sistema A, quindi, riceve in ingresso 4 segnali: *open_trasmission* (in quanto entità che deve avviare la comunicazione), *ok_message_A* (che il sistema B utilizza per comunicare l'avvenuta ricezione) ed infine il reset ed il clock.

Passiamo, ora, alla descrizione del sistema B. Come fatto in precedenza per il sistema A, riportiamo innanzitutto di seguito lo schematico realizzato, evidenziando unità di controllo e unità operativa.

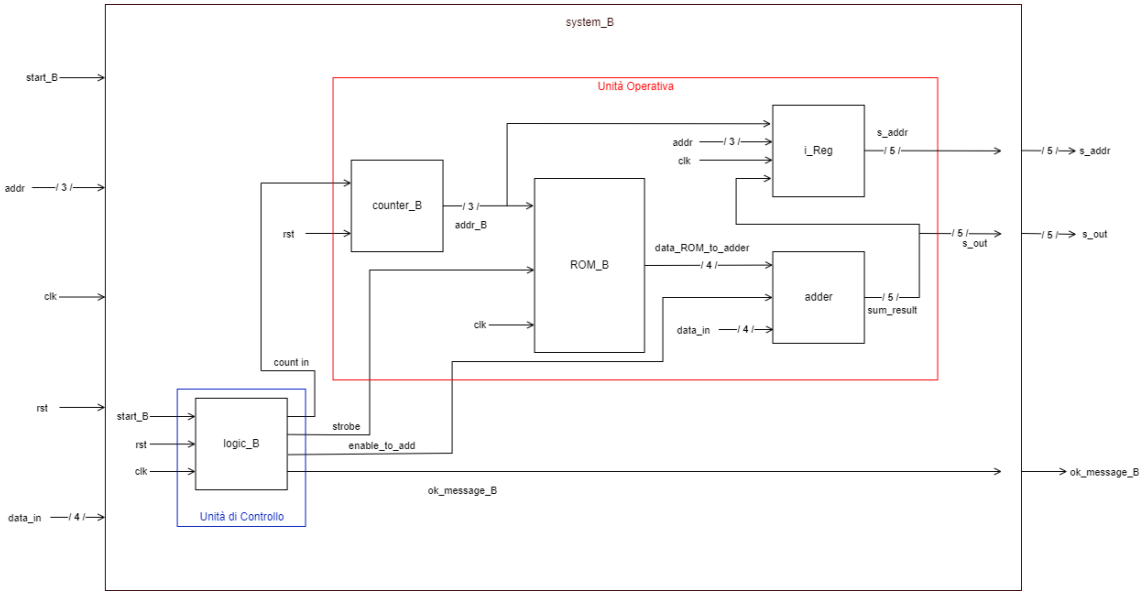


Figura 7.3 - Sistema B

Partiamo con la descrizione della logica di controllo (componente *logic_B*). Anche in questo caso, abbiamo realizzato un automa a stati finiti che riportiamo di seguito.

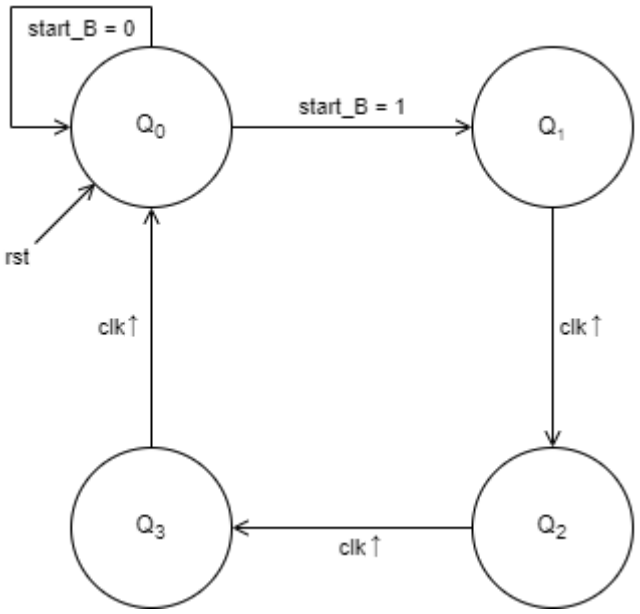


Figura 7.4 - Automa a stati finiti del componente logic_B

L'unità di controllo è stata implementata nuovamente tramite automa di Moore. Inizialmente, la macchina si trova nello stato q_0 e permarrà in tale stato finché il segnale *start* inviato dal sistema A non risulterà alto. Quando ciò avviene, il sistema evolve nello stato q_1, che alza il valore di uscita di *count_in* diretto al contatore del sistema B per incrementarne il valore. Al successivo colpo di clock, la macchina passa nello stato q_2 in cui riporta a 0 il valore di *count_in* ed alza il valore di *strobe* diretto alla ROM del sistema stesso in maniera del tutto analoga a quanto descritto per il sistema A. Al colpo di clock ancora successivo, lo *strobe* non commuta per mantenere lo stesso valore dello stato precedente. Contemporaneamente, però, vengono alzati i valori di *enable_add* (per abilitare l'operazione di somma) e di *ok_message_B* (uscita del sistema diretta al sistema A per notificare l'avvenuta ricezione del dato). Al successivo colpo di clock, la macchina ritornerà nello stato iniziale q_0 in attesa di ricevere il successivo avvio di scambio di informazioni da parte del sistema A.

Di seguito riportiamo il codice VHDL di quanto appena descritto.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.math_real.ALL;

ENTITY logic_B IS
	GENERIC
	(
		n_string : INTEGER := 8;
		m_bit    : INTEGER := 4
	);
	PORT
	(
		start_B      : IN std_logic;
		rst          : IN std_logic;
		clk          : IN std_logic;
		strobe       : OUT std_logic;
		count_in     : OUT std_logic;
		ok_message_B : OUT std_logic; --3
		enable_add   : OUT std_logic
	);
END logic_B;

ARCHITECTURE Behavioral OF logic_B IS

	TYPE stato IS (q0, q1, q2, q3);
	SIGNAL current_state, next_state : stato;

BEGIN
```

```
	PROCESS (clk)
	BEGIN
		IF rising_edge(clk) THEN
			IF (rst = '1') THEN
				current_state <= q0;
			ELSE
				current_state <= next_state;
			END IF;
		END IF;
	END PROCESS;

	PROCESS (current_state, start_B)
		BEGIN
			CASE current_state IS
				WHEN q0 =>
					count_in     <= '0';
					strobe       <= '0';
					enable_add   <= '0';
					ok_message_B <= '0';
					IF (start_B = '1') THEN
						next_state <= q1;
					ELSE
						next_state <= current_state;
					END IF;
				WHEN q1 =>
					count_in     <= '1';
					strobe       <= '0';
					enable_add   <= '0';
					ok_message_B <= '0';
					next_state   <= q2;
				WHEN q2 =>
					count_in     <= '0';
					strobe       <= '1';
					enable_add   <= '0';
					ok_message_B <= '0';
					next_state   <= q3;
				WHEN q3 =>
					count_in     <= '0';
					strobe       <= '1';
					enable_add   <= '1';
					ok_message_B <= '1';
					next_state   <= q0;
			END CASE;
		END PROCESS;

END Behavioral;
```

Passando alla parte operativa del sistema B, troviamo 4 componenti:

- contatore
- ROM
- sommatore

- registro

Il contatore e la ROM sono componenti uguali a quelli già descritti e mostrati nel sistema A a meno di una serie di valori differenti presenti all'interno della ROM.

Riportiamo di seguito il codice VHDL del sommatore che - semplicemente - effettua la somma tra l'i-esimo dato proveniente dalla ROM del sistema A (*op1*) ed il corrispondente dato i-esimo della ROM del sistema B (*op2*).

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.math_real.ALL;
USE IEEE.numeric_std.ALL;
USE IEEE.std_logic_unsigned.ALL;

ENTITY adder IS
	GENERIC
	(
		n_string : INTEGER := 8;
		m_bit    : INTEGER := 4
	);
	PORT
	(
		enable_add : IN std_logic;
		op1        : IN std_logic_vector(m_bit - 1 DOWNTO 0);
		op2        : IN std_logic_vector(m_bit - 1 DOWNTO 0);
		result     : OUT std_logic_vector(m_bit DOWNTO 0)
	);
END adder;

ARCHITECTURE Behavioral OF adder IS

BEGIN
	PROCESS (enable_add, op1, op2)
	BEGIN
		IF (enable_add = '1') THEN
			result <= ('0' & op1) + ('0' & op2);
		END IF;
	END PROCESS;

END Behavioral;
```

Notiamo che aggiungiamo un bit al risultato tramite concatenazione in quanto sappiamo che per effettuare una somma nel complemento a 2 è necessario, appunto, aggiungere tale bit per il riporto. Dunque, passiamo da operandi a 4 bit ad un risultato a 5 bit. Inoltre, abbiamo utilizzato un approccio completamente behavioral servendoci così del tool per la sua realizzazione.

L'ultimo componente del sistema B che resta da descrivere è *i_Reg*, utilizzato per memorizzare gli i-esimi valori delle somme. Abbiamo utilizzato per tale componente un approccio strutturale al fine di provare a realizzare un registro non necessariamente comportamentale, come fatto invece in esercizi precedenti. Notiamo, inoltre, la presenza del segnale *addr* che, rispetto alla traccia, abbiamo aggiunto per poter avere anche poi nella fase di simulazione un segnale che ci indicasse la locazione di memoria del registro attualmente puntata. Il registro *i_Reg*, dunque, è costituito da:

- un demultiplexer che seleziona in quale registro salvare la somma ottenuta ricevendo in ingresso come selettore il valore del contatore (inserendo la somma nell'i-esimo registro);
- una serie di registri;
- un multiplexer che seleziona il registro contenente il dato da mostrare in uscita (ricevendo come selettore l'*addr* descritto in precedenza).

Riportiamo di seguito uno schematico ed i relativi codici VHDL realizzati per il registro *i_Reg* e per i suoi registri più interni.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

PACKAGE define_num_reg_type IS
	CONSTANT n_string : INTEGER := 8;
	CONSTANT m_bit    : INTEGER := 4;
	TYPE num_reg IS ARRAY (n_string - 1 DOWNTO 0) OF
std_logic_vector(m_bit DOWNTO 0);
END PACKAGE define_num_reg_type;

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.math_real.ALL;
USE IEEE.numeric_std.ALL;
USE IEEE.std_logic_unsigned.ALL;
USE work.define_num_reg_type.ALL;

ENTITY i_Reg IS
	GENERIC
	(
		n_string : INTEGER := 8;
		m_bit    : INTEGER := 4
	);
	PORT
	(
		data_sum : IN std_logic_vector(m_bit DOWNTO 0);
		addr_i   : IN
std_logic_vector(INTEGER(ceil(log2(real(n_string)))) - 1 DOWNTO 0);
```

```
            addr     : IN
std_logic_vector(INTEGER(ceil(log2(real(n_string)))) - 1 DOWNTO 0);
            clk      : IN std_logic;
            s_addr   : OUT std_logic_vector(m_bit DOWNTO 0)
      );
END i_Reg;

ARCHITECTURE Structural OF i_Reg IS

      SIGNAL demux_to_reg : num_reg;
      SIGNAL reg_to_mux   : num_reg;

      COMPONENT demux IS
            GENERIC
            (
                  n_string : INTEGER := 8;
                  m_bit    : INTEGER := 4
            );
            PORT
            (
                  data_in  : IN std_logic_vector(m_bit DOWNTO 0);
                  sel      : IN
std_logic_vector(INTEGER(ceil(log2(real(n_string)))) - 1 DOWNTO 0);
                  data_out : OUT num_reg
            );
      END COMPONENT;

      COMPONENT reg_m_bit IS
            GENERIC
            (
                  m_bit : INTEGER := 4
            );
            PORT
            (
                  data_in  : IN std_logic_vector(m_bit DOWNTO 0);
                  clk      : IN std_logic;
                  data_out : OUT std_logic_vector(m_bit DOWNTO 0)
            );
      END COMPONENT;

      COMPONENT mux IS
            GENERIC
            (
                  n_string : INTEGER := 8;
                  m_bit    : INTEGER := 4
            );
            PORT
            (
                  data_in  : IN num_reg;
                  sel      : IN
std_logic_vector(INTEGER(ceil(log2(real(n_string)))) - 1 DOWNTO 0);
                  data_out : OUT std_logic_vector(m_bit DOWNTO 0)
            );
      END COMPONENT;

BEGIN
```

```
	d : demux
		GENERIC MAP
		(
		n_string => 8,
		m_bit    => 4
		)
		PORT MAP
		(
			data_in  => data_sum,
			sel      => addr_i,
			data_out => demux_to_reg
		);

			registers : FOR i IN n_string - 1 DOWNTO 0 GENERATE
				BEGIN
					reg : reg_m_bit
						GENERIC MAP
						(
						m_bit => 4
						)
						PORT MAP
						(
							data_in  => demux_to_reg(i),
							clk      => clk,
							data_out => reg_to_mux(i)
						);
				END GENERATE registers;

				m : mux
					GENERIC MAP
					(
					n_string => 8,
					m_bit    => 4
					)
					PORT MAP
					(
						data_in  => reg_to_mux,
						sel      => addr,
						data_out => s_addr
					);

END Structural;
```

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY reg_m_bit IS
	GENERIC
	(
		m_bit : INTEGER := 4
	);
	PORT
	(
		data_in  : IN std_logic_vector(m_bit DOWNTO 0);
```

```
            clk      : IN std_logic;
            data_out : OUT std_logic_vector(m_bit DOWNTO 0)
      );
END reg_m_bit;

ARCHITECTURE Behavioral OF reg_m_bit IS

BEGIN
      PROCESS (clk)
      BEGIN
            IF rising_edge(clk) THEN
                  data_out <= data_in;
            END IF;
      END PROCESS;

END Behavioral;
```

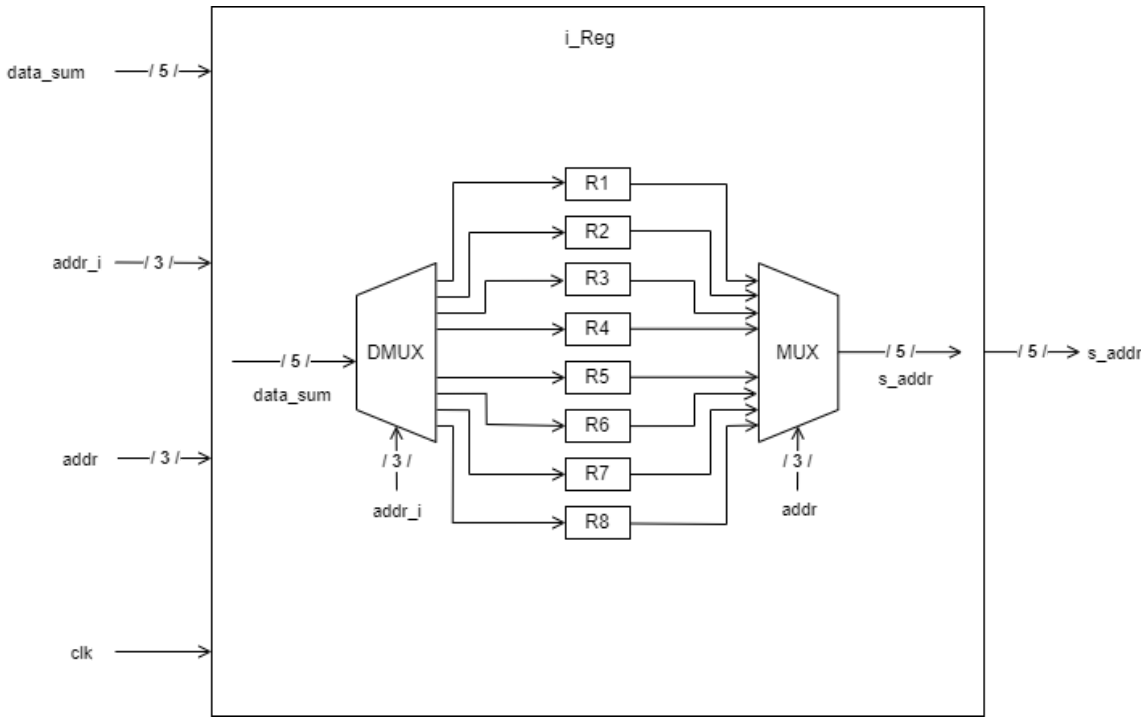


Figura 7.5 - Schematico del componente i_Reg

Infine, riportiamo il codice realizzato per l'entità *"system_B"* con approccio strutturale che presenta dichiarazioni e port map delle istanze dei componenti che la costituiscono.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.math_real.ALL;

ENTITY system_B IS
```

```
	GENERIC
	(
		n_string : INTEGER := 8;
		m_bit    : INTEGER := 4
	);
	PORT
	(
		start_B      : IN std_logic; --2
		addr         : IN
std_logic_vector(INTEGER(ceil(log2(real(n_string)))) - 1 DOWNTO 0);
		clk          : IN std_logic;
		rst          : IN std_logic;
		data_in      : IN std_logic_vector(m_bit - 1 DOWNTO 0); --1
		ok_message_B : OUT std_logic; --3
		s_out        : OUT std_logic_vector(m_bit DOWNTO 0); --uscita
al passo i-esimo
		s_addr       : OUT std_logic_vector(m_bit DOWNTO 0) --valore
selezionato da visualizzare
	);
END system_B;

ARCHITECTURE Structural OF system_B IS

	SIGNAL enable_add : std_logic;
	SIGNAL count_in          : std_logic;
	SIGNAL enable_to_add     : std_logic;
	SIGNAL strobe            : std_logic;
	SIGNAL addr_B            :
std_logic_vector(INTEGER(ceil(log2(real(n_string)))) - 1 DOWNTO 0);
	SIGNAL data_ROM_to_adder : std_logic_vector(m_bit - 1 DOWNTO 0);
	SIGNAL sum_result        : std_logic_vector(m_bit DOWNTO 0);

	COMPONENT logic_B IS
		PORT
		(
			start_B : IN std_logic;
			rst     : IN std_logic;
			clk     : IN std_logic;
			strobe       : OUT std_logic;
			count_in     : OUT std_logic;
			ok_message_B : OUT std_logic; --3
			enable_add   : OUT std_logic
		);
	END COMPONENT;

	COMPONENT counter IS
		GENERIC
		(
			n_string : INTEGER := 8;
			m_bit    : INTEGER := 4
		);
		PORT
		(
			count_in       : IN std_logic;
			rst            : IN std_logic;
			index_selector : OUT
```

```
std_logic_vector(INTEGER(ceil(log2(real(n_string)))) - 1 DOWNTO 0) --la i
della traccia
            );
      END COMPONENT;

      COMPONENT ROM_B IS
            GENERIC
            (
                  n_string : INTEGER := 8;
                  m_bit    : INTEGER := 4
            );
            PORT
            (
                  addr_i   : IN
std_logic_vector(INTEGER(ceil(log2(real(n_string)))) - 1 DOWNTO 0);
                  strobe   : IN std_logic;
                  clk      : IN std_logic;
                  data_out : OUT std_logic_vector(m_bit - 1 DOWNTO 0)
            );
      END COMPONENT;

      COMPONENT adder IS
            GENERIC
            (
                  n_string : INTEGER := 8;
                  m_bit    : INTEGER := 4
            );
            PORT
            (
                  enable_add : IN std_logic;
                  op1        : IN std_logic_vector(m_bit - 1 DOWNTO 0);
                  op2        : IN std_logic_vector(m_bit - 1 DOWNTO 0);
                  result     : OUT std_logic_vector(m_bit DOWNTO 0)
            );
      END COMPONENT;

      COMPONENT i_Reg IS
            GENERIC
            (
                  n_string : INTEGER := 8;
                  m_bit    : INTEGER := 4
            );
            PORT
            (
                  data_sum : IN std_logic_vector(m_bit DOWNTO 0);
                  addr_i   : IN
std_logic_vector(INTEGER(ceil(log2(real(n_string)))) - 1 DOWNTO 0);
                  addr     : IN
std_logic_vector(INTEGER(ceil(log2(real(n_string)))) - 1 DOWNTO 0);
                  clk      : IN std_logic;
                  s_addr   : OUT std_logic_vector(m_bit DOWNTO 0)
            );
      END COMPONENT;

BEGIN
      logic_unit_B : logic_B
```

```
        PORT MAP
        (
                start_B => start_B,
                rst     => rst,
                clk     => clk,
                strobe       => strobe,
                count_in     => count_in,
                ok_message_B => ok_message_B,
                enable_add   => enable_to_add
        );

        counter_B : counter
                GENERIC MAP
                (
                n_string => 8,
                m_bit    => 4
                )
                PORT MAP
                (
                        count_in       => count_in,
                        rst            => rst,
                        index_selector => addr_B
                );

                        ROM_op2 : ROM_B
                                GENERIC MAP
                                (
                                n_string => 8,
                                m_bit    => 4
                                )
                                PORT MAP
                                (
                                        addr_i   => addr_B,
                                        strobe   => strobe,
                                        clk      => clk,
                                        data_out => data_ROM_to_adder
                                );

                                        add : adder
                                                GENERIC MAP
                                                (
                                                n_string => 8,
                                                m_bit    => 4
                                                )
                                                PORT MAP
                                                (
                                                        enable_add => enable_to_add,
                                                        op1        => data_in,
                                                        op2        =>
data_ROM_to_adder,
                                                        result     => sum_result
                                                );

                                                        s_out <= sum_result; --uscita

                                                        sum_Reg : i_Reg
```

```
                                                        GENERIC MAP
                                                        (
                                                        n_string => 8,
                                                        m_bit    => 4
                                                        )
                                                        PORT MAP
                                                        (
                                                               data_sum =>
sum_result,
                                                               addr_i   =>
addr_B,
                                                               addr     =>
addr,
                                                               clk      => clk,
                                                               s_addr   =>
s_addr
                                                        );

END Structural;
```

Notiamo che il segnale *s_out* viene utilizzato per mostrare in uscita i valori dell'i-esima somma.

Di seguito riportiamo, infine, lo schematico ed il codice VHDL della top-module del sistema.

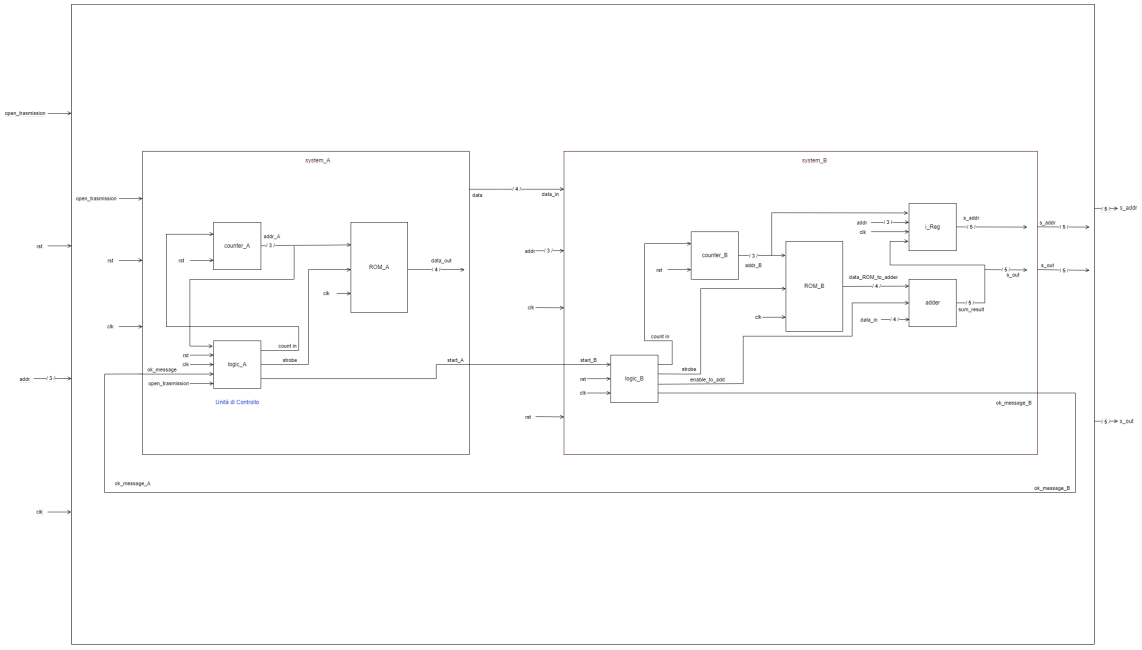


Figura 7.6 - Schematico della top-module

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.math_real.ALL;
```

```
ENTITY Handshaking_System IS
	GENERIC
	(
		n_string : INTEGER := 8;
		m_bit    : INTEGER := 4
	);
	PORT
	(
		open_trasmission : IN std_logic; --segnale di inizio
trasmissione
		rst              : IN std_logic;
		addr             : IN
std_logic_vector(INTEGER(ceil(log2(real(n_string)))) - 1 DOWNTO 0);
--indirizzo della stringa
		clk              : IN std_logic;
		s_out            : OUT std_logic_vector(m_bit DOWNTO 0);
--segnale di uscita dopo aver effettuato la somma
		s_addr           : OUT std_logic_vector(m_bit DOWNTO 0)
--segnale di selezione dell'indirizzo che si vuole visualizzare
	);
END Handshaking_System;

ARCHITECTURE Structural OF Handshaking_System IS

	COMPONENT system_A IS
		GENERIC
		(
			n_string : INTEGER := 8;
			m_bit    : INTEGER := 4
		);
		PORT
		(
			open_trasmission : IN std_logic; --0
			rst              : IN std_logic;
			clk              : IN std_logic;
			ok_message_A     : IN std_logic; --3
			data_out         : OUT std_logic_vector(m_bit - 1
DOWNTO 0); --1
			start_A          : OUT std_logic --2
		);
	END COMPONENT;

	COMPONENT system_B IS
		GENERIC
		(
			n_string : INTEGER := 8;
			m_bit    : INTEGER := 4
		);
		PORT
		(
			start_B      : IN std_logic; --2
			addr         : IN
std_logic_vector(INTEGER(ceil(log2(real(n_string)))) - 1 DOWNTO 0);
			clk          : IN std_logic;
			rst          : IN std_logic;
```

```
                    data_in      : IN std_logic_vector(m_bit - 1 DOWNTO 0);
--1
                    ok_message_B : OUT std_logic; --3
                    s_out        : OUT std_logic_vector(m_bit DOWNTO 0);
--segnale di uscita dopo aver effettuato la somma
                    s_addr       : OUT std_logic_vector(m_bit DOWNTO 0)
--segnale di selezione dell'indirizzo che si vuole visualizzare
             );
      END COMPONENT;

      SIGNAL data       : std_logic_vector(m_bit - 1 DOWNTO 0);
      SIGNAL ok_message : std_logic;
      SIGNAL start      : std_logic;

BEGIN
      sys_A : system_A
            GENERIC MAP
            (
            n_string => 8,
            m_bit    => 4
            )
            PORT MAP
            (
                  open_trasmission => open_trasmission,
                  rst              => rst,
                  clk              => clk,
                  ok_message_A     => ok_message,
                  start_A          => start,
                  data_out         => data
            );

                  sys_B : system_B
                        GENERIC MAP
                        (
                        n_string => 8,
                        m_bit    => 4
                        )
                        PORT MAP
                        (
                              start_B      => start,
                              addr         => addr,
                              rst          => rst,
                              clk          => clk,
                              ok_message_B => ok_message,
                              data_in      => data,
                              s_out        => s_out,
                              s_addr       => s_addr
                        );

END Structural;
```

Avendo realizzato i due sistemi protagonisti della comunicazione (*system_A* e *system_B*) in maniera completamente strutturale, notiamo che nella top-module del sistema complessivo è bastato dichiarare e realizzare la port-map solo dei due sistemi.

7.1.3 Simulazione

Riportiamo di seguito il codice del testbench realizzato.

```
stimulus: process
begin
    wait for 100ns;
    open_trasmission <= '1';
    rst <= '0';
    wait for 10ns;
    open_trasmission <= '0';
    wait for 10ns;
    wait for 350ns;
    addr <= "001";
    wait for 50ns;
    addr <= "000";
    wait for 300ns;
    open_trasmission <= '1';
    wait for 100ns;
    wait;
end process;

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

Per consentire una facile lettura di quanto ottenuto in fase di simulazione, riportiamo di seguito due grafici corredati da una breve spiegazione.

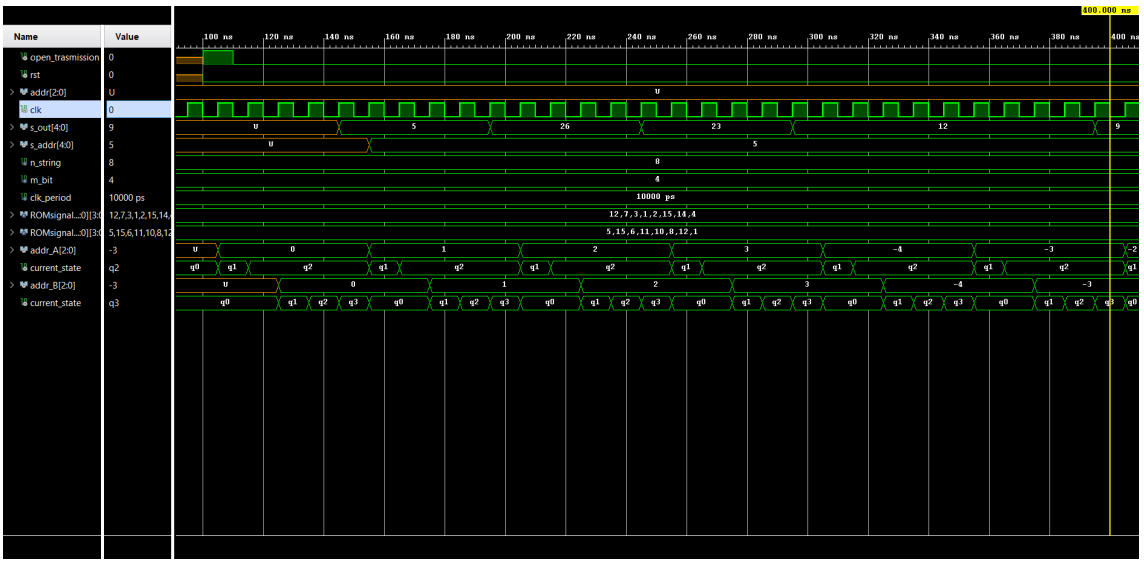


Figura 7.6 - Grafico della simulazione

Nel primo grafico abbiamo voluto evidenziare il corretto avanzare degli stati che compongono gli automi a stati finiti delle unità di controllo dei due sistemi. Come è possibile notare, gli stati evolvono nella maniera che ci aspettavamo descritta in precedenza nell'apposito paragrafo.

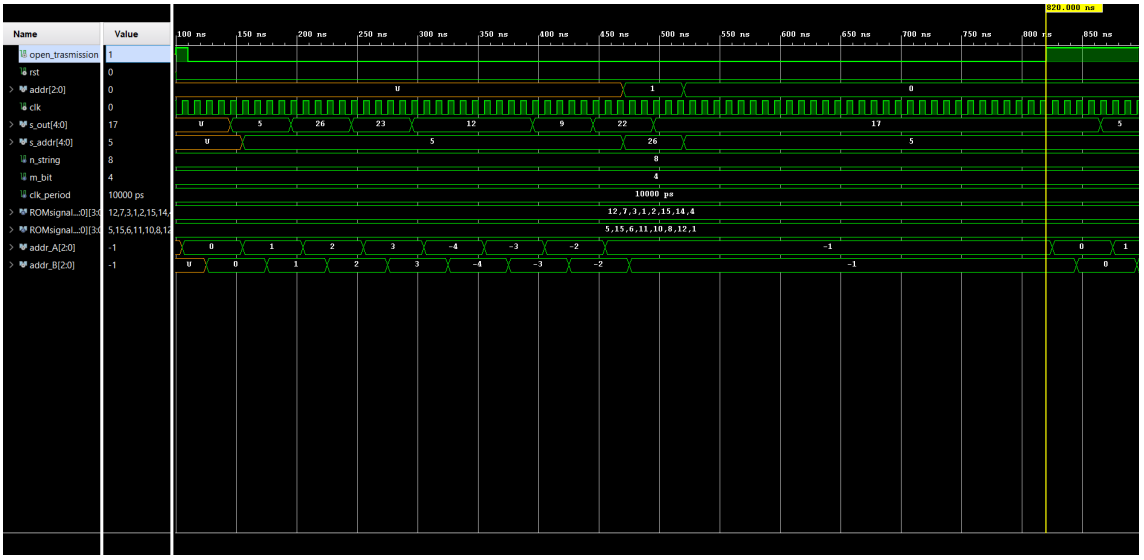


Figura 7.7 - Grafico della simulazione 2

In questo secondo grafico, invece, risultano visibili tutti i valori contenuti nelle due ROM ed anche le i-esime somme ottenute in uscita dal sistema. Le somme avvengono in maniera corretta (4+1=5, 12+14=26 e così via). Da notare che il valore 12 ottenuto in uscita perdura per il doppio dei colpi di clock rispetto a quelli attesi in quanto ottenuto due volte da due somme consecutive (10+2 ed 11+1, rispettivamente la quarta somma ottenuta con *i=3* e la quinta con *i=4*).

Notiamo, infine, che al termine dell'intera comunicazione, entrambi i sistemi ritornano come previsto nello stato iniziale in attesa di un ulteriore segnale di avvio di una nuova trasmissione che arriva poco dopo 800ns. Successivamente, è visibile nuovamente la prima uscita, reiterando la comunicazione appena descritta.

Capitolo 8

Processore MIC-1

Il processore MIC-1 è una macchina programmabile a stack che non dispone di registri generali, realizzata da Tanenbaum.
L'unità operativa comprende l'ALU, i registri (tra cui quelli che si interfacciano con la memoria), gli ingressi e le uscite.

Registri

Ogni registro viene usato per uno scopo specifico:

- H (*holding*) è un registro tampone che mantiene il primo operando dell'ALU e lo invia all'ALU tramite il "bus A";
- SP (*stack pointer*) punta alla cima dello stack e contiene un indirizzo di memoria;
- LV (*local variable*) punta sempre alla base del frame dell'istruzione attualmente attiva;
- TOS (*top of stack*) contiene il valore in cima allo stack;
- CPP *(constant pool pointer)* area costanti;
- OPC (*old program counter*) è un registro "scratch" che mantiene il valore del PC prima di effettuare un salto.

Di seguito riportiamo i "Memory Control Register" destinati alla lettura e scrittura dei dati:

- MAR (Memory Address Register) → registro su 32 bit di solo output verso la memoria, word-addressable a 32 bit;
- MDR (Memory Data Register) → registro su 32 bit di input/output, word-addressable a 32 bit.

Di seguito, invece, riportiamo i “Memory Control Register” destinati alla lettura delle istruzioni:

- PC (Program Counter) → registro su 32 bit di sola uscita verso la memoria, byte-addressable a 8 bit;
- MBR (Memory Byte Register) → registro su 32 bit di solo input (preleva istruzioni), byte-addressable a 8 bit.

Nonostante il MAR ed il PC siano due registri su 32 bit, è lecito chiedersi come mai gli indirizzi vengano rappresentati rispettivamente su 32 bit e su 8 bit. La ragione è che le architetture a stack riducono la lunghezza delle istruzioni, dato che non vi è specificato un address destinazione e quindi area dati e area istruzioni, in questo modello, sono separate. Inoltre, le istruzioni hanno operandi impliciti. Di conseguenza si ha un metodo di indirizzamento diverso per i due casi.
La memoria è costituita da parole di 8 bit, univocamente identificate da indirizzi da 32 bit e quindi è possibile indirizzare 2^{32} parole. Ponendo l’indirizzo in PC, al successivo colpo di clock troveremo nell’MBR il byte indicato. Scrivendo lo stesso indirizzo nel registro MAR abilitando la lettura in memoria, al colpo di clock successivo troveremo in MDR i 4 byte che rappresentano il dato.
Per ottenere l’indirizzo corretto, il MAR effettua uno shift a sinistra di 2 bit per determinare l’indirizzo da cui prelevare i 4 byte.

Bus

Nel caso del MIC-1 due bus collegano opportunamente i componenti. Il bus B collega i registri all’ALU mentre il bus C collega l’uscita dell’ALU (in particolare lo *shifter* posto in uscita all’ALU) con i registri. Il bus C collega l’ALU a più registri e permette, se necessario, di scrivere anche in più registri contemporaneamente in un solo colpo di clock. Il bus B, invece, può effettuare solo una lettura alla volta indirizzando un solo valore (operando B) verso la ALU. Dunque, non possiamo leggere due segnali in un solo colpo di clock in quanto tutto passa per l’ALU che rappresenta il “collo di bottiglia” di questa microarchitettura.

Il bus B non è collegato a tutti i registri. In particolare, non è collegato al registro MAR (utile solo per l'indirizzamento in memoria e, quindi, senza necessità di essere letto) e ad H perché direttamente collegato all'ALU come possibile operando tramite il bus A (operando A).

Il bus C, invece, non è collegato al registro MBR in quanto questo registro è preposto al prelievo dell'istruzione dalla memoria e, quindi, non avrebbe senso alterare un'istruzione proveniente dalla memoria.

La parte operativa divide dati da istruzioni. Infatti, come già descritto, alcuni registri sono destinati ai dati e altri alle istruzioni.

Visto che MBR deve acquisire solo un byte alla volta dalla memoria mentre il MDR ne deve acquisire quattro, i collegamenti tra la memoria ed il processore avranno dimensioni differenti.

È possibile però che alcune istruzioni abbiano dei parametri e quindi risultino essere più lunghe di un byte. Questo processore, lavorando su istruzioni a lunghezze fisse, necessiterà a quel punto di cambiare due volte il PC al fine di prelevare dalla memoria i due byte che compongono l'istruzione completa.

Ad esempio se bisogna saltare da un indirizzo ad un altro, prima si preleva dalla memoria l'istruzione di salto, rappresentata in un byte, si scrive l'istruzione di salto in MBR e poi, dopo un ulteriore accesso in memoria, si accede all'indirizzo a cui saltare estendendo il registro MBR con questo indirizzo.

I 3 segnali di bus (A, B e C) collegano tutti i blocchi (compresa l'ALU) e non ci sono quindi ulteriori segnali intermedi. Questo vuol dire che sicuramente la parte operativa non sarà difficile da tempificare ma, possono esserci problemi di concorrenza sui bus.

I bus comunicano con i componenti in parallelo.

ALU

Il componente principale di questa architettura è l'ALU. Questo componente ha due ingressi ovvero i due operandi. Il primo ingresso è collegato al registro H tramite il bus A mentre il secondo al bus B. Il registro H, come accennato in precedenza, è un registro

tampone che viene utilizzato per salvare uno dei due operandi prima di un'operazione. Dipendentemente dalle linee di controllo, l'ALU si comporta in modo differente.

L'ALU ha sei bit di controllo:

- *F0, F1* definiscono la funzione;
- *ENA, ENB* abilitano gli ingressi A e B;
- *INVA* inverte l'ingresso A;
- *INC* incrementa di 1.

Nel processore MIC-1, in quanto architettura a stack, una delle operazioni più ricorrenti è quella di incremento di uno e, per questo, è possibile notare l'esistenza di un segnale di controllo ad-hoc per realizzare tale operazione. Ad esempio, nelle architetture a stack, questo risulta fondamentale per l'operazione di incremento dello Stack Pointer (o di decremento per le architetture a stack decrescente).

F_0	F_1	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	0	B – 1
1	1	1	0	1	1	–A
0	0	1	1	0	0	A AND B
0	1	1	1	0	0	A OR B
0	1	0	0	0	0	0
1	1	0	0	0	1	1
1	1	0	0	1	0	–1

Figura 8.1 - Segnali di controllo dell'ALU e relative funzioni

Le uscite dell'ALU sono tre: il risultato dell'operazione e 2 flag che indicano se il risultato ottenuto è negativo (*N*) o zero (*Z*).
Il risultato dell'ALU, prima di andare sul bus C, attraversa uno shifter che può shiftare verso sinistra di 8 bit (impostando i bit meno significativi a 0) o a destra di 1 bit mantenendo il segno o lasciando il dato inalterato. Per effettuare tali operazioni lo shifter necessita di altre due linee di controllo. In totale, dunque, i bit necessari per il controllo della ALU e dello *shifter* sono otto.

Comunicazione tra registri

Come già detto, il bus B permette di leggere valori da un registro mentre il bus C permette di scrivere in uno o più registri. Questo implica che se vogliamo passare un dato da un registro ad un altro, dobbiamo - e lo si nota dall'architettura - necessariamente passare per l'ALU. Questo inevitabilmente rallenta le operazioni su questa architettura. Inoltre, ci possono essere più registri che si contendono la comunicazione sul bus B verso l'ALU e poi dall'ALU al bus C al fine di cambiare il valore di un registro. Questo è un problema sorgente-destinazione che si risolve con un multiplexer che, in corrispondenza di opportuni bit di selezione, gestiti da un'unità di controllo, lascia "parlare" sul bus solo un registro alla volta, proprio come un semaforo.
Per quanto riguarda il bus C, si utilizza in questo caso un demultiplexer dall'uscita dell'ALU verso i singoli registri, gestiti anch'essi da opportuni segnali di controllo.
L'ALU, in questi casi, deve far passare il dato inalterato, visto che il dato deve solo essere scritto su un altro registro.
Inoltre, visto che il dato deve passare necessariamente per l'ALU ed uno shifter, ci saranno dei tempi di propagazione e tempi legati alla logica combinatoria a cui bisogna far attenzione.
Considerando che, in questa architettura, i dati attraversano sempre l'ALU, è facile determinare i tempi di propagazione i quali saranno più o meno uguali per tutte le operazioni.
Il progettista deve dimensionare opportunamente il tempo di clock per gestire i tempi di propagazione e quelli legati alla logica combinatoria della microarchitettura.

Più in particolare, c'è bisogno di un tempo A per prelevare l'istruzione dalla memoria di controllo, contenente i bit che pilotano gli elementi del datapath, un tempo B per selezionare il registro e portare il suo contenuto sul bus B, un tempo C per far lavorare l'ALU e lo shift register (tempo più lungo) ed infine un tempo D per propagare correttamente il risultato sul bus C. Tipicamente, si cerca di essere ancora più conservativi aggiungendo un ulteriore tempo di attesa per garantire che tutto funzioni correttamente. Al termine di queste attese, il segnale è sul bus C e si può quindi scrivere su un registro.

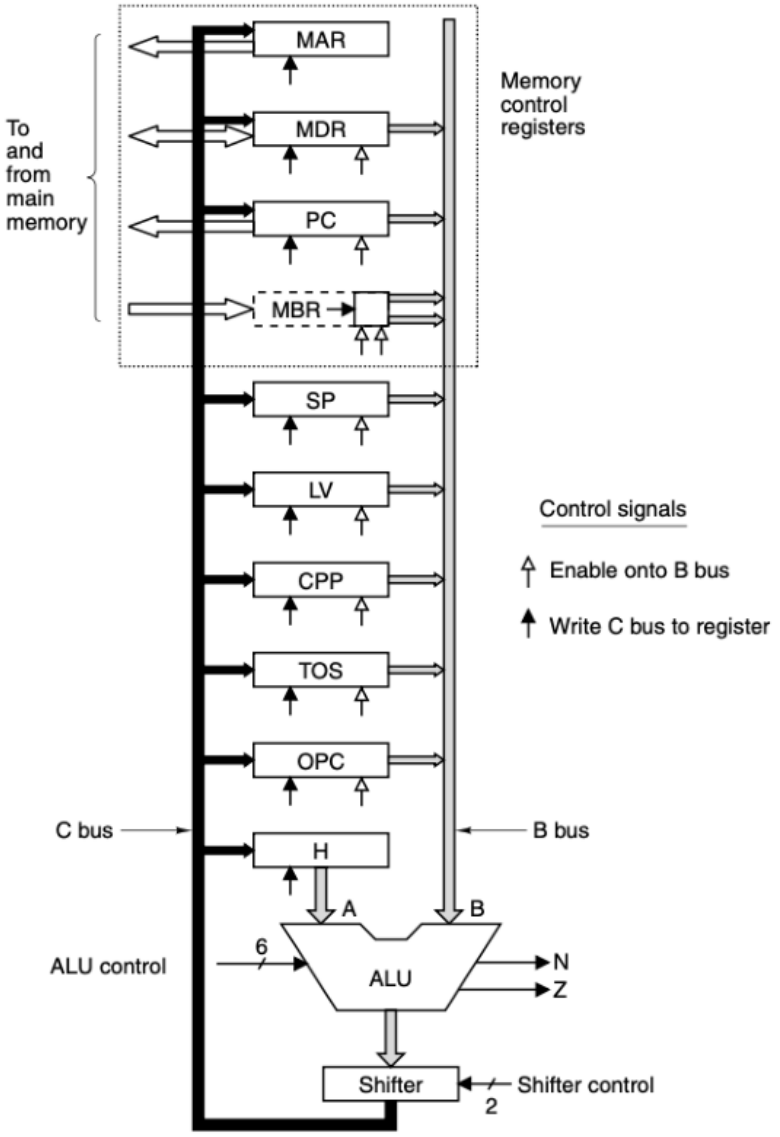


Figura 8.2 - Unità operativa del processore MIC-1

Per controllare l'unità operativa del processore, sono necessari 24 segnali:

- 9 segnali per effettuare la selezione del bus C;
- Diversamente dalla selezione del bus C, solo un registro alla volta può parlare sul bus B. Questo implica che possiamo codificare i 9 segnali in 4 bit (lasciando alcune combinazioni scoperte utilizzate per la situazione in cui nessuno sta "parlando" con il bus B);
- 6 segnali per controllare l'ALU e 2 per lo shift register;
- 2 segnali di read/write per specificare se si vuole scrivere o leggere dalla memoria tramite la coppia MAR/MDR;
- 1 segnale per indicare il fetch sull'interfaccia PC/MBR. In questo caso basta un unico segnale perché non c'è segnale di "write" come nel MDR perché nella nostra architettura non avrebbe senso sovrascrivere un'istruzione.

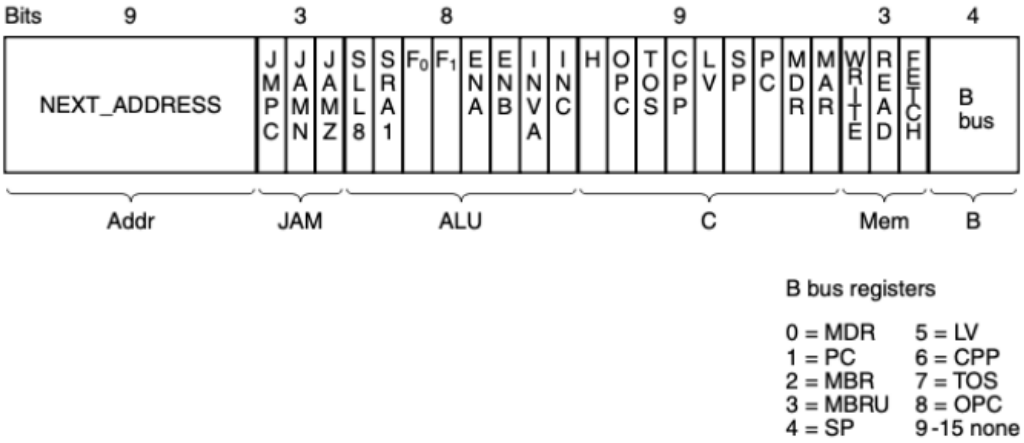


Figura 8.3 - Struttura della Control Word

L'unità di controllo può essere realizzata in due possibili modalità:

- Logica cablata → Progettiamo l'unità di controllo in termini di porte logiche;
- Logica microprogrammata → Esiste una logica esterna che altera il flusso di esecuzione in base a condizioni ricavate da ingressi di controllo e segnali di stato.

Indipendentemente dalla modalità scelta, la parte operativa rimane inalterata.

Le microprocedure devono essere in grado di prelevare, decodificare ed eseguire istruzioni IJVM.

Il microprogramma è un insieme di funzioni e procedure contenuto in una ROM che deve essere in grado di pilotare i circuiti logici nella giusta sequenza.

La ROM sarà popolata da tanti microprogrammi statici.

Una volta "bruciati" i microprogrammi sulla ROM, c'è bisogno di un main program che, in base alle istruzioni, richiami questi microprogrammi saltando all'indirizzo giusto (gestito dall'incremento del Program Counter).

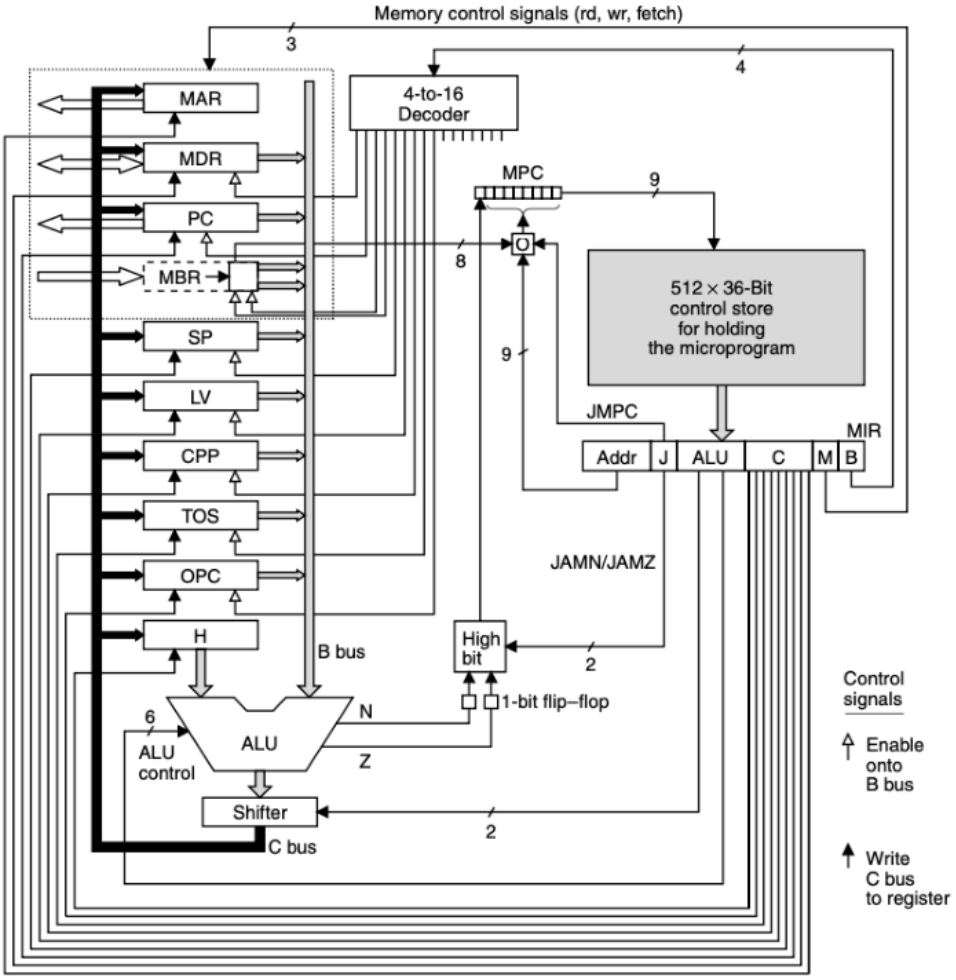


Figura 8.4 - Schema completo del MIC-1

Tornando al MIC-1, la sua parte di controllo è gestita da una Control Store che "mantiene" i microprogrammi. Ad ogni colpo di clock, si legge il valore in MPC (su 9 bit) che rappresenta l'indirizzo da visitare e si accede alla Control Store. In corrispondenza

di quell'offset, si troverà una stringa binaria corrispondente alla microistruzione da eseguire che controlla la parte operativa. A sua volta, la microistruzione può avere come prossimo indirizzo quello immediatamente successivo o ritornare alla routine principale. In realtà, l'indirizzo in MPC dipende dalla presenza o meno di una jump routine.

Il calcolo dell'indirizzo della prossima microistruzione da eseguire, ha inizio subito dopo che MIR (Memory Instruction Register) è stato caricato ed è stabile, e avviene nel modo seguente:

- Il campo di 9 bit specificato da *Addr* (in MIR) viene copiato in MPC.
- Vengono a questo punto controllati i bit del campo JAM:
    - Se tutti e 3 i bit sono 0, non viene fatto altro e l'indirizzo della prossima microistruzione diviene semplicemente quello indicato da NEXT_ADDRESS;
    - Se JAMN è 1, il bit più significativo di MPC viene messo in OR con N (prelevato dal Flip-flop);
    - Se JAMZ è 1, il bit più significativo di MPC viene messo in OR con Z (prelevato dal Flip-flop);
    - Se entrambi - JAMN e JAMZ - sono a 1 si fa l'OR con entrambi.

Detto con un'unica espressione booleana:

F = (JAMZ AND Z) OR (JAMN AND N) OR NEXT_ADDRESS[8]

In ogni caso, dunque, MPC conterrà uno dei due valori:

- Il valore di NEXT_ADDRESS;
- Il valore di NEXT_ADDRESS con il bit più significativo in OR con 1.

L'utilizzo di JAMZ e JAMN permette di eseguire salti condizionati a seconda del valore dei bit di stato Z ed N.

8.1 Analisi istruzioni del processore

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

a) si proceda all'analisi dell'architettura mediante simulazione e si approfondisca lo studio del suo funzionamento per due istruzioni a scelta.

8.1.1 Soluzione

Le istruzioni scelte sono IAND e ISUB.

Per analizzare cosa succede all'unità di controllo bisogna studiare il codice binario generato da queste due procedure. Esse vengono generate da un traduttore che, partendo dal codice MAL, ne genera il codice macchina. Una volta fatto ciò, uno script Python utilizzerà tale codice macchina e farà una "cucitura" sul file *control_store.vhd* per popolare opportunamente la ROM generata, formata da microistruzioni da 36 bit. Una volta popolata la ROM, bisogna generare il programma da caricare in RAM, partendo dal codice IJVM.

Quest'ultimo verrà salvato in *dp_ar_ram.vhd* sempre con un'operazione di cucitura ed il programma dovrà eseguire le operazioni descritte dalla RAM.

Ogni microprocedura termina con una microistruzione che torna ad una "microprocedura iniziale" (*main*) la quale incrementa il PC, per eseguire tutte le operazioni descritte dal programma in RAM.

```
main:
        PC = PC + 1; fetch; goto (MBR)
```

Incrementare il Program Counter e abilitare il *fetch* della prossima istruzione impiega del tempo in quanto, quando il *fetch* è alto, solo al colpo di clock successivo il registro MBR conterrà l'istruzione corrispondente all'indirizzo della RAM.

Notiamo che il *fetch* ed il *goto* vengono eseguite sulla stessa riga. Tanembaum ha progettato tutto in modo da dover aspettare meno cicli possibili. Prima di eseguire il *main* c'è un'altra microistruzione che viene eseguita soltanto una volta e che viene eseguita proprio 2 cicli prima del *main* e che prepara il *fetch*. Quindi il *fetch* della prima

istruzione da eseguire è fatto un po' in anticipo e così quando il *main* viene eseguito per la prima volta, MBR ha già il valore valido e questo *fetch* non viene fatto per eseguire subito il salto ma per preparare già il valore quando torna al *main*. Quindi sono istruzioni che avvengono nello stesso tempo perché infatti nel linguaggio MAL le istruzioni che avvengono in tempi diversi sono scritte in linee differenti. Il PC quindi punta a qualcosa che sta più avanti.

8.1.2 Realizzazione

IAND

Dati in input due operandi A e B, IAND è l'operazione preposta all'operazione A AND B.

La microprocedura IAND viene descritta nel microprogramma *ajvm.mal* attraverso le seguenti microistruzioni:

```
iand = 0x7E:
        MAR = SP = SP - 1; rd
        H = TOS
        MDR = TOS = MDR AND H; wr; goto main
```

Notiamo dal codice MAL che IAND è descritta nella Control Store dall'indirizzo *0x7E* *(126_{10})* della memoria ROM. Ogni riga corrisponde ad una microistruzione, e siccome IAND è composta da 3 microistruzioni, ci aspettiamo che il microprogramma termini alla riga 128_{10}.

```
123 => "00000000000000000000000000000000000000",
124 => "00000000000000000000000000000000000000",
125 => "00000000000000000000000000000000000000",
126 => "00111111100000110110000001001010010 0",
127 => "01000000000000010100100000000000000111",
128 => "00000011000000001100001000010100000 0",
129 => "00000000000000000000000000000000000000",
130 => "00000000000000000000000000000000000000",
131 => "00000000000000000000000000000000000000",
132 => "01000010100000010100100000000000000101",
133 => "01000011000000111100000000001010001 1",
```

Figura 8.5 - Microprocedura IAND in codice macchina del control_store.vhd

ADDR	NEXT_ADDRESS	JAM	ALU	C	MEM	B
126	001111111	000	00 | 110110	000001001	010	0100
127	010000000	000	00 | 010100	100000000	000	0111
128	000000110	000	00 | 001100	001000010	100	0000

1) *MAR = SP = SP - 1; rd*

che, in riferimento al formato delle microistruzioni, ci dice che:

- Il prossimo address è 001111111 ovvero 127 in decimale;
- Il JAM è nullo quindi si prende normalmente il NEXT_ADDRESS;
- Lo shifter è trasparente e all'ALU arriva il codice di controllo 110110, che come è possibile evincere dalla tabella di *Figura 8.1*, corrisponde all'operazione B-1;
- Vengono attivati i registri MAR e SP sul bus C, scrivendo nei registri il valore che viene mandato in uscita dall'ALU;
- La memoria (registro dati) è in modalità *read*. Al prossimo colpo di clock avremo disponibile il dato desiderato;
- La decodifica della stringa 0100 corrisponde all'attivazione del registro SP sul bus B.

Il bus B viene collegato con SP che, in uscita all'ALU, avendo scelto un codice di sottrazione, sottrae l'indirizzo dello Stack Pointer. SP-1 viene poi scritto in SP (per mantenere traccia dello Stack Pointer corrente) e in MAR (per prelevare il dato dalla memoria). La memoria è in modalità *read* e questo vuol dire che, dopo un colpo di clock, nel registro MDR ci sarà il valore richiesto (cache hit 100%).

2) *H = TOS*

- Il prossimo address è 010000000 ovvero 128 in decimale;
- Il JAM è nullo, quindi si prende normalmente il NEXT_ADDRESS;
- Lo shifter è ancora trasparente e all'ALU arriva il codice di controllo 010100, che come è possibile evincere dalla tabella di *Figura 8.1*, restituisce B;
- Il Bus C abilita il registro H;
- Non c'è interazione con la memoria;

- La decodifica della stringa 0111 corrisponde all'attivazione del registro TOS sul bus B;

Una volta richiesto il primo operando, adesso necessitiamo del secondo: esso non è altro che la cima dello stack (TOS) e, quindi, procediamo a salvare nel registro H il valore di TOS. Per farlo, abilitiamo sul bus B il registro TOS, l'ALU deve lasciar passare il dato invariato e il bus C deve scrivere nel registro H.

3) *MDR = TOS = MDR AND H; wr; goto main*

- Il prossimo address è 000000110 (si ritorna alla sequenza principale);
- Il JAM è nullo quindi si prende normalmente il NEXT_ADDRESS;
- Lo shifter è ancora trasparente e all'ALU arriva il codice di controllo 001100, cioè A AND B;
- Il bus C abilita il registro MDR e TOS per preservare l'invariante dello stack;
- C'è un'operazione di scrittura di dati;
- La decodifica della stringa 0000 corrisponde all'attivazione del registro MDR che ospita l'operando B.

Adesso, essendo passato un colpo di clock dalla richiesta del dato alla memoria ed, avendo a disposizione entrambi gli operatori all'interno della macchina (uno nel registro H e uno in MDR), basterà solamente attivare il registro MDR sul bus B, fare una operazione di AND con l'ALU e contemporaneamente salvare il valore in TOS ed in MDR. La ragione per cui si scrive anche in TOS è per rispettare l'invariante dello stack per poi, volendo, riportarlo in memoria.

ISUB

Dati in input due operandi A e B, ISUB è l'istruzione preposta all'operazione B-A.

Come nel caso precedente, si mostra la microprocedura descritta in *ajvm.mal*:

```
isub = 0x5C:
        MAR = SP = SP - 1; rd
        H = TOS
        MDR = TOS = MDR - H; wr; goto main
```

Stavolta l'indirizzo da cui parte ISUB è 0x5c (92_{10}). Si riporta quindi uno screen della Control Store che è stata generata così come fatto per IAND.

```
90 => "00101101100000000000000000000000001001",
91 => "00000011000000001010000100000000000000",
92 => "00101110100000110110000001001010010 0",
93 => "00101111000000010100100000000000000111",
94 => "00000011000000111111001000010100000 0",
95 => "00110000000000110110000000001010010 0",
96 => "00110000100000001010000000000100000 0",
97 => "00110001000000001010010000000010000 0",
```

Figura 8.6 - Microprocedura ISUB in codice macchina del control_store.vhd

ADDR	NEXT_ADDR	JAM	ALU	C	MEM	B
92	001011101	000	00 | 110110	000001001	010	0100
93	001011110	000	00 | 010100	100000000	000	0111
94	000000110	000	00 | 111111	001000010	100	0000

Questa microprocedura e la precedente presentano alcune analogie: entrambe eseguono un'operazione su 2 operandi, uno dei quali prelevato dalla memoria e uno dal registro TOS. C'è un'operazione di decremento mentre il bus C abilita MAR e SP, proprio come nell'esempio precedente. Anche la riga 93 lavora allo stesso modo di IAND mentre la riga 94, invece, corrisponde ad una funzione dell'ALU di sottrazione B-A. Infine il valore viene scritto in TOS e in MDR.

3) *MDR = TOS = MDR - H; wr; goto main*

- Il prossimo address è 000000110 (si ritorna alla sequenza principale);
- Il JAM è nullo quindi si prende normalmente il NEXT_ADDRESS;

- Lo shifter è ancora trasparente e all'ALU arriva il codice di controllo 111111, cioè B-A;
- Il Bus C abilita il registro MDR e TOS per preservare l'invariante dello stack;
- C'è un'operazione di scrittura di dati;
- La decodifica della stringa 0000 corrisponde all'attivazione del registro MDR che ospita l'operando B.

8.1.3 Simulazione IAND

Utilizzando il progetto messo a disposizione dall'Ing. Moriconi, prima di simulare, dobbiamo generare il microprogramma su ROM e poi il programma su RAM.
La generazione di codice della ROM viene effettuata da un traduttore che, avendo in input la descrizione della Control Store *ajvm.mal*, ne genera l'equivalente codice binario. Il codice viene poi "cucito" in *control_store.vhd* come detto precedentemente. Il comando eseguito per fare ciò è
cmake --build build --target create_control_store

Si genera poi la RAM partendo dalle istruzioni nel main program e poi, analogamente alla generazione della ROM, si "cuce" il risultato in *dp_ar_ram.vhd*
cmake --build build --target create_ram

Infine per simulare e quindi eseguire i casi di test:
cmake --build build --target check

Al fine di simulare l'operazione di IAND, necessitiamo di un main program che utilizzi questa istruzione. Di seguito il codice *program.ajvm*:

```
.main
.var
a
.endvar
BIPUSH 0x3
BIPUSH 0x7
IAND
ISTORE a
HALT
.endmethod
```

Ci aspettiamo che l'uscita della AND sia 7 AND 3 ovvero 3.

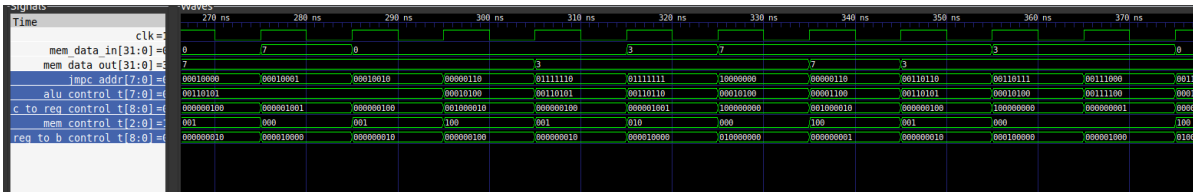


Figura 8.7 - Grafico della simulazione

Per completezza, nella simulazione abbiamo mostrato alcuni segnali dell'unità di controllo. A 295 ns termina l'esecuzione dell'istruzione precedente e si torna nella Control Store al rigo 6 (microprocedura principale). La nostra simulazione si concentra sull'analisi di IAND e quindi, analizzeremo direttamente a 315 ns dove effettivamente vediamo che i valori della simulazione combaciano con la prima microistruzione dell'IAND. Da lì in poi, si seguono le microistruzioni in base al NEXT_ADDRESS finché non si torna al microprogramma principale che incrementa il PC. Al quarto colpo di clock, come ci aspettavamo dall'analisi eseguita sopra, il processore presenta in uscita il valore desiderato, ovvero 3.

8.1.3 Simulazione ISUB

Al fine di simulare ISUB, c'è bisogno di un main program che richiami questa microprocedura. Per fare ciò abbiamo modificato *program.ajvm* affinché contenga un'operazione di ISUB. Ci si aspetta quindi che, essendo gli operandi uguali, il risultato sia 0.

```
.main
.var
a
.endvar
BIPUSH 0x8
BIPUSH 0x8
ISUB
ISTORE a
HALT
.endmethod
```

Simulando nuovamente col tool *gtkwave* e mettendo in evidenza i soli segnali d'interesse, si nota che a 315 ns (analogamente a prima, non avendo cambiato le

istruzioni precedenti) viene caricato il NEXT_ADDRESS 93 e quindi è iniziata la sequenza 92. Si nota che, al termine dell'ultima istruzione BIPUSH 0x8, a 295 ns, si ritorna nella microprocedura principale dove si incrementa il Program Counter che punta proprio a 92 dove, al colpo di clock successivo, viene letto dal Control Store. Come descritto nell'analisi precedente di IAND, al quarto colpo di clock, si vedrà in uscita dal processore il risultato, ovvero 0.

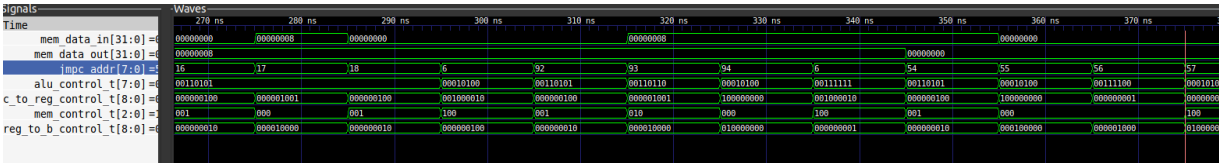


Figura 8.8 - Grafico della simulazione

8.2 Overloading di operatore

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

b) si modifichi un codice operativo a scelta, documentando tutte le modifiche effettuate.

8.2.1 Soluzione

Si vuole trasformare la AND in Somma + Moltiplicazione per 256.

Per modificare un codice operativo possiamo:

- alterare le singole microistruzioni in *ajvm.mal* rigenerando poi il codice binario per il Control Store;
- cambiare manualmente i bit della Control Store affinché un'operazione di AND diventi qualcos'altro.

In base a quanto mostrato nell'esercizio precedente, sappiamo che IAND risiede nella Control Store all'indirizzo 126_{10}. Basterà allora cambiare i bit di controllo della ALU e quelli dello shifter in corrispondenza dell'operazione di IAND.

Bisogna stare attenti ai passi da seguire questa volta in quanto, dopo aver cambiato manualmente il valore in *control_store.vhd*, non bisogna rigenerare la ROM altrimenti si andrebbe a perdere la configurazione che manualmente abbiamo cambiato. Inoltre, non avendo alterato il main program, è inutile anche rigenerare la RAM. Bisogna solo lanciare la simulazione attraverso *cmake --build build --target check*.

Si decide, quindi, di modificare la microistruzione numero 128 affinché una AND diventi una somma e uno shift di 8 bit a sinistra. Per fare ciò l'ALU dovrà ricevere il codice 111100 e lo shifter il codice 10.

Quindi si sostituisce alla stringa

```
000000110 000 00001100 001000010 100 0000
```

la stringa

```
000000110 000 10111100 001000010 100 0000
```

La nuova configurazione risulterà, quindi, essere la seguente:

```
126 => "00111111100000110110000001001010010 0",
127 => "01000000000000001010010000000000000111",
128 => "00000011000010111100001000010100000 0",
```

Figura 8.9 - Nuova configurazione della microprocedura IAND

8.2.1 Simulazione

In corrispondenza del colpo di clock a 315 ns viene caricata la riga della Control Store numero 126_{10} e quindi, inizia l'esecuzione dell'operazione di IAND. Questa volta però IAND farà altro visto che abbiamo manualmente alterato il file che contiene la ROM. Si nota che, come mostrato in figura e come visto nelle simulazioni precedenti, al quarto colpo di clock il processore fornirà in uscita il risultato che, avendo dato in ingresso 7 e 3, ne ha fatto la somma e moltiplicato per 256 fornendo così in uscita il valore 2560 al tempo 345 ns.

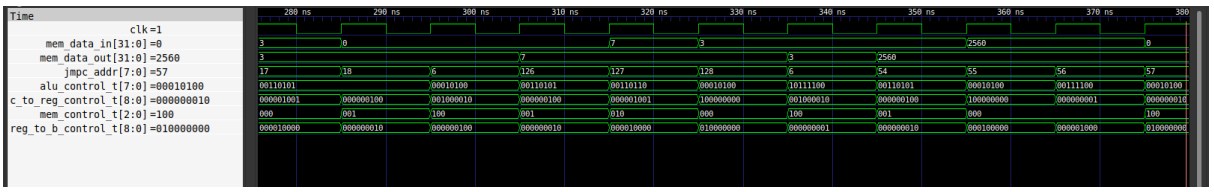


Figura 8.10 - Grafico della simulazione di IAND aggiornato

Capitolo 9

Trasmissione seriale - UART/USART

Nonostante la comunicazione tipica del mondo dei calcolatori sia quella parallela, solitamente si rende necessario passare da tale comunicazione ad una di tipo seriale, tipica, invece, del mondo delle telecomunicazioni. La comunicazione seriale, infatti, risulta molto adatta per poter ridurre il numero di fili ed evitare improvvisi sbalzi di tensione che potrebbero compromettere la comunicazione stessa. Quindi, per poter realizzare una periferica di comunicazione avremo sicuramente bisogno di un registro a scorrimento, dispositivo necessario per passare appunto da parallelo a seriale (lato trasmettitore) o viceversa (lato ricevitore).

Tale passaggio da comunicazione parallela a comunicazione seriale, però, ci pone dinanzi ad alcune problematiche relative alla sincronizzazione tra i due sistemi protagonisti della comunicazione. Infatti, se tale comunicazione non fosse opportunamente sincronizzata, ci sarebbe il rischio che si possano perdere informazioni. In particolare, il problema non riguarda il trasmettitore ma il ricevitore che non sa quando inizia la trasmissione. Inoltre, bisogna considerare che qualsiasi periferica dovrà necessariamente realizzare una comunicazione con il processore al fine di poter accedere al registro a scorrimento in cui saranno inseriti i dati in fase di trasmissione e successivamente prelevati in fase di ricezione.

Esistono due tipologie di comunicazione seriale: sincrona e asincrona. Nella comunicazione asincrona, per la trasmissione di ogni carattere, la linea presenta una configurazione standard (ad esempio: riposo, bit start, informazione, eventuale bit di parità, bit di stop e ritorno a riposo). Per quanto riguarda quella sincrona, invece, si ragiona in riferimento al concetto di messaggio, e di fatto la sincronizzazione avviene per ogni messaggio. In tal caso infatti vengono ad esempio mandati due segnali di sync ed un “treno” di caratteri fino a quando non si hanno nuovamente due segnali di sync. Appare dunque evidente che per quanto la comunicazione asincrona risulti più

semplice mantenere la sincronizzazione (in quanto avviene per ogni carattere), quella sincrona risulta maggiormente efficiente considerando appunto il rapporto numero di bit utili/numero totale di bit inviati. Dopo aver effettuato alcune considerazioni in merito alla comunicazione seriale in generale, andiamo a concentrarci adesso in particolare su quella di tipo asincrona.

Per poter realizzare la comunicazione asincrona, andremo in particolare ad utilizzare l'interfaccia UART (Universal Asynchronous Receiver Trasmitter). Tale interfaccia ci permetterà di effettuare in maniera versatile il passaggio da comunicazione parallela a comunicazione seriale e viceversa, agendo sia da trasmettitore che da ricevitore.

Esistono tre modalità di trasmissione:

- Simplex (comunicazione unidirezionale, dove solo il trasmettitore comunica con il ricevitore);
- Half Duplex (comunicazione bidirezionale, dove entrambe le entità possono fungere da trasmettitore e ricevitore, a turno);
- Full Duplex (comunicazione bidirezionale, dove entrambe le entità possono fungere da trasmettitore e ricevitore, anche contemporaneamente).

In una qualsiasi comunicazione asincrona dobbiamo considerare che il segnale che passa attraverso la linea di comunicazione deve sempre mantenere il valore di tensione alto (1) a riposo in modo tale da essere in grado di capire se la linea è effettivamente attiva. Questa scelta è stata effettuata poiché col valore di tensione a zero non risulterebbe possibile distinguere quando la linea è attiva rispetto a quando è spenta dato che anche in questo caso il valore di tensione sarebbe basso. Per poter segnalare l'inizio della trasmissione abbasso la linea a 0 (segnale di *"start"*) in modo tale che quando la linea si alzerà nuovamente riuscirò a catturare correttamente i bit. Dopo il segnale di start, troviamo i bit utili seguiti da un eventuale bit di parità in modo da verificare se - a causa del rumore - ci sia stata qualche alterazione errata nei valori dei bit trasmessi. Fatto ciò, la linea deve tornare in uno stato di *"stop"* per un quanto di tempo altrimenti non potrei mandare un nuovo carattere. Successivamente troviamo un nuovo valore di *"start"* che segnala l'invio del carattere successivo. I caratteri

vengono copiati in un buffer in modo da dare al processore il tempo di poterlo leggere. Se il buffer non è stato svuotato, allora uno o più caratteri si sovrascriveranno a quelli già presenti all'interno del buffer stesso.

Notiamo ora che, per cercare di ridurre al minimo gli effetti della deriva, si cerca di prendere sempre *"al centro"* i valori dei bit in modo da non avere effetti dovuti né dalla deriva di sinistra né dalla deriva di destra. Per questo il ricevitore lavora ad una frequenza multipla rispetto a quella del trasmettitore, affinchè possa campionare più volte i valori trasmessi. In questo modo vado a ridurre gli effetti di deriva ma non quelli di rumore: se il rumore dovesse sovrapporsi al segnale da prendere proprio nel momento in cui si cerca di prenderlo, allora inevitabilmente il valore di quel bit risulterà sbagliato.

Nella comunicazione asincrona, non viene fatto un handshaking per ogni carattere. Il messaggio viene inviato per intero.

Il connettore per questo tipo di trasmissioni è un connettore standard: l'RS-232.

9.1 Interfaccia seriale su FPGA

Sfruttando l'implementazione fornita dalla Digilent di un dispositivo UART (componente *RS232RefComp.vhd*), progettare e implementare in VHDL un sistema costituito da 2 nodi A e B collegati tra loro mediante una interfaccia seriale. Il sistema A acquisisce una stringa di 8 bit dall'utente (mediante gli switch della board di sviluppo) e la invia mediante la seriale al sistema B, che la manda in output sui led della board di sviluppo.

9.1.1 Realizzazione e sintesi

La board Nexys-A7 è già dotata dell'interfaccia UART. La comunicazione avviene su canale Simplex e, per questo, sono necessari due registri a scorrimento, uno per il ricevitore ed uno per il trasmettitore.

Notiamo che abbiamo scelto di implementare una configurazione "a tappo": il componente acquisisce una stringa di 8 bit tramite switch e la serializza tramite la sezione di trasmissione del dispositivo UART; l'output seriale della UART viene

re-inviato in ingresso alla sezione di ricezione dello stesso dispositivo e il dato deserializzato viene visualizzato sui led della board di sviluppo.

Vogliamo, dunque, testare tale componente caricandolo sulla board. In questo modo siamo in grado di fornire l'ingresso tramite gli switch per poi mostrare l'output tramite i LED. Abbiamo, inoltre, assegnato il segnale WR ad un bottone della scheda in modo da "sostituirci" al processore nell'avvio della comunicazione. Un altro bottone verrà utilizzato per il reset. Infine, abbiamo utilizzato altri LED a cui abbiamo assegnato i bit relativi agli errori.

Notiamo che, a causa del fatto che nessuno preleverà il contenuto del registro a scorrimento, dalla seconda comunicazione in poi il bit OE (overrun error) risulterà sempre alto e, di conseguenza, anche il LED ad esso associato risulterà sempre acceso.

Un'accortezza che abbiamo voluto implementare è stata quella di aggiungere un divisore di frequenza: siccome la nostra scheda è dotata di un clock di base di 100 Mhz mentre il baud rate raggiungeva solo i 48 Mhz, abbiamo deciso di dimezzare la frequenza stessa. Riportiamo di seguito soltanto il codice VHDL del divisore di frequenza da noi realizzato, omettendo quello del componente RS-232 in quanto da noi solo importato all'interno del progetto.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY Frequency_Divider IS
	PORT
	(
		clk_in      : IN std_logic;
		RST         : IN std_logic;
		clk_divided : OUT std_logic
	);
END Frequency_Divider;

ARCHITECTURE Behavioral OF Frequency_Divider IS

BEGIN
	clk_division     : PROCESS (clk_in)
		VARIABLE counter : INTEGER RANGE 0 TO 1 := 0;
	BEGIN
		IF RST = '1' THEN
			counter := 0;
			clk_divided <= '0';
		ELSIF clk_in'EVENT AND clk_in = '1' THEN
			IF counter = 1 THEN
				clk_divided <= '1';
```

```
                        counter := 0;
                    ELSE
                        clk_divided <= '0';
                        counter := counter + 1;
                    END IF;
                END IF;
        END PROCESS;

END Behavioral;
```

Di seguito mostriamo, quindi, uno schematico della top-module realizzata per poter collegare opportunamente il divisore di frequenza con l'UART ed il relativo codice VHDL.

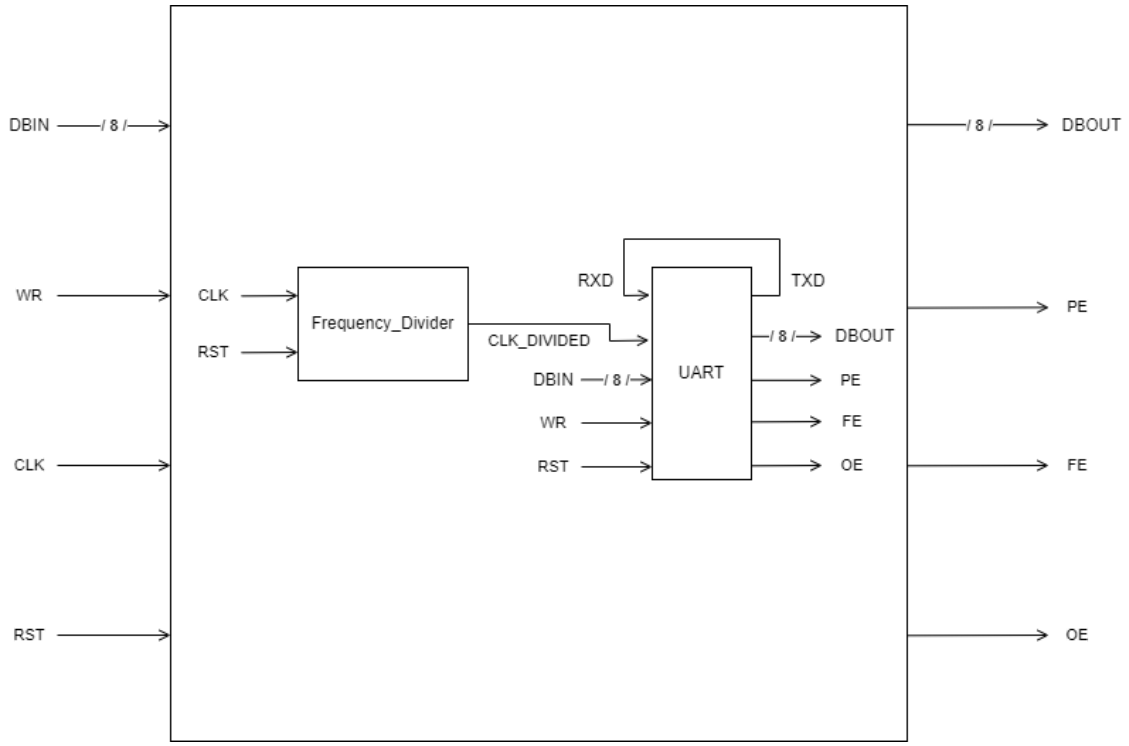


Figura 9.1 - Schematico della top-module

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Serial_System is
  Port (
    -- Dati di ingresso e uscita in parallelo
    DBIN: in std_logic_vector(7 downto 0); -- Switch di ingresso
    DBOUT: out std_logic_vector(7 downto 0); -- Led per l'uscita
    -- Bit di errori (visualizzati sempre tramite led)
    PE: out std_logic;
    FE: out std_logic;
```

```
    OE: out std_logic;
    -- Strobe
    WR: in std_logic; -- Tramite pulsante

    CLK: in std_logic;
    RST: in std_logic
  );
end Serial_System;

architecture Structural of Serial_System is

component Rs232RefComp is
    Generic (
        BAUD_DIVIDE : integer := 54;
        BAUD_RATE : integer := 868
    );
    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(è alto quando il dato è disponibile nel
registro rdReg)
            TBE        : inout std_logic := '1';
--Transfer Bus Empty(se è 0 significa che il bus è occupato per una
trasmissione)
            RD           : in  std_logic;
--Read Strobe(se è 0 significa "leggi")
            WR           : in  std_logic;
--Write Strobe(se è 1 significa "scrivi")
            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 Frequency_Divider is
  Port (
    clk_in : in std_logic;
    RST : in std_logic;
    clk_divided : out std_logic
  );
end component;

signal TRXD: std_logic;
signal clk_divided: std_logic;

begin
```

```
    UART: Rs232RefComp GENERIC MAP (
        BAUD_DIVIDE => 54,
        BAUD_RATE => 868
    )
    PORT MAP (
        TXD => TRXD,
      RXD => TRXD,
      CLK => clk_divided,
            DBIN => DBIN,
            DBOUT => DBOUT,
            RDA => OPEN,
            TBE => OPEN,
            RD => '0',
            WR => WR,
            PE => PE,
            FE => FE,
            OE => OE,
            RST => RST
    );

    Freq_Div: Frequency_Divider PORT MAP (
        CLK_IN => CLK,
        RST => RST,
        CLK_DIVIDED => CLK_DIVIDED
    );

end Structural;
```

Di seguito riportiamo, infine, il file XDC delle constraints utilizzate per la sintesi del sistema su board descritte sopra.

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


##Switches
set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 } [get_ports {
DBIN[0] }]; #IO_L24N_T3_RS0_15 Sch=sw[0]
set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 } [get_ports {
DBIN[1] }]; #IO_L3N_T0_DQS_EMCCLK_14 Sch=sw[1]
set_property -dict { PACKAGE_PIN M13   IOSTANDARD LVCMOS33 } [get_ports {
DBIN[2] }]; #IO_L6N_T0_D08_VREF_14 Sch=sw[2]
set_property -dict { PACKAGE_PIN R15   IOSTANDARD LVCMOS33 } [get_ports {
DBIN[3] }]; #IO_L13N_T2_MRCC_14 Sch=sw[3]
set_property -dict { PACKAGE_PIN R17   IOSTANDARD LVCMOS33 } [get_ports {
DBIN[4] }]; #IO_L12N_T1_MRCC_14 Sch=sw[4]
set_property -dict { PACKAGE_PIN T18   IOSTANDARD LVCMOS33 } [get_ports {
DBIN[5] }]; #IO_L7N_T1_D10_14 Sch=sw[5]
set_property -dict { PACKAGE_PIN U18   IOSTANDARD LVCMOS33 } [get_ports {
DBIN[6] }]; #IO_L17N_T2_A13_D29_14 Sch=sw[6]
```

```
set_property -dict { PACKAGE_PIN R13   IOSTANDARD LVCMOS33 } [get_ports {
DBIN[7] }]; #IO_L5N_T0_D07_14 Sch=sw[7]

## LEDs
set_property -dict { PACKAGE_PIN H17   IOSTANDARD LVCMOS33 } [get_ports {
DBOUT[0] }]; #IO_L18P_T2_A24_15 Sch=led[0]
set_property -dict { PACKAGE_PIN K15   IOSTANDARD LVCMOS33 } [get_ports {
DBOUT[1] }]; #IO_L24P_T3_RS1_15 Sch=led[1]
set_property -dict { PACKAGE_PIN J13   IOSTANDARD LVCMOS33 } [get_ports {
DBOUT[2] }]; #IO_L17N_T2_A25_15 Sch=led[2]
set_property -dict { PACKAGE_PIN N14   IOSTANDARD LVCMOS33 } [get_ports {
DBOUT[3] }]; #IO_L8P_T1_D11_14 Sch=led[3]
set_property -dict { PACKAGE_PIN R18   IOSTANDARD LVCMOS33 } [get_ports {
DBOUT[4] }]; #IO_L7P_T1_D09_14 Sch=led[4]
set_property -dict { PACKAGE_PIN V17   IOSTANDARD LVCMOS33 } [get_ports {
DBOUT[5] }]; #IO_L18N_T2_A11_D27_14 Sch=led[5]
set_property -dict { PACKAGE_PIN U17   IOSTANDARD LVCMOS33 } [get_ports {
DBOUT[6] }]; #IO_L17P_T2_A14_D30_14 Sch=led[6]
set_property -dict { PACKAGE_PIN U16   IOSTANDARD LVCMOS33 } [get_ports {
DBOUT[7] }]; #IO_L18P_T2_A12_D28_14 Sch=led[7]
set_property -dict { PACKAGE_PIN V16   IOSTANDARD LVCMOS33 } [get_ports {
PE }]; #IO_L16N_T2_A15_D31_14 Sch=led[8]
set_property -dict { PACKAGE_PIN T15   IOSTANDARD LVCMOS33 } [get_ports {
FE }]; #IO_L14N_T2_SRCC_14 Sch=led[9]
set_property -dict { PACKAGE_PIN U14   IOSTANDARD LVCMOS33 } [get_ports {
OE }]; #IO_L22P_T3_A05_D21_14 Sch=led[10]

##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 {
WR }]; #IO_L9P_T1_DQS_14 Sch=btnc
set_property -dict { PACKAGE_PIN M18   IOSTANDARD LVCMOS33 } [get_ports {
RST }]; #IO_L4N_T0_D05_14 Sch=btnu
```

9.2 Interfaccia seriale con ROM

Implementare la seguente variante dell'esercizio precedente: il sistema A invia al sistema B tramite l'interfaccia seriale N stringhe di 8 bit contenute all'interno di una memoria ROM. Le stringhe ricevute vengono memorizzate in una memoria locale a B. Il progetto deve prevedere che A utilizzi un componente contatore per scandire le N stringhe da inviare.

9.2.1 Soluzione

Per poter simulare la comunicazione tra un sistema A ed un sistema B tramite interfaccia seriale abbiamo ripreso quanto realizzato nell'esercizio precedente. In più,

abbiamo aggiunto - come richiesto dalla traccia - un contatore nel sistema A per poter scandire tutte le locazioni di memoria di una ROM in modo da poter inviare le stringhe di bit in essa contenute.

Notiamo l'aggiunta di un componente *"delay"*: tale componente svolge due ruoli fondamentali:

- Rallenta il segnale di un colpo di clock durante la comunicazione. In questo modo siamo stati in grado di utilizzare soltanto un segnale di *write* (implementato poi su un bottone della scheda) per poter gestire l'intera comunicazione anziché utilizzare due segnali, uno per caricare il valore dalla ROM e l'altro per eseguire effettivamente il *write.* In questo modo assicuriamo che la ROM abbia il tempo necessario per poter correttamente predisporre il valore in uscita.
- Imposta il segnale alto per un ulteriore colpo di clock, affinché possa essere rilevato correttamente in riferimento al *clk_divided* che, come già detto, presenta una frequenza dimezzata rispetto a quella nativa.

Per quanto riguarda il sistema B, la memoria locale è stata implementata tramite un registro scandito, anche in questo caso, tramite un contatore per effettuare correttamente il salvataggio dei valori ricevuti al termine di ogni comunicazione.

9.2.2 Realizzazione

In tale paragrafo, omettiamo di inserire il codice VHDL del debouncer (realizzato per il bottone utilizzato poi in fase di sintesi per mappare il segnale *WR*) e della ROM in quanto componenti già ampiamente dettagliate nei capitoli precedenti di questo elaborato. Evitiamo, inoltre, di riproporre l'implementazione VHDL del componente divisore di frequenza, già descritto nel paragrafo precedente.

Notiamo, invece, la presenza di due contatori che - come richiesto dalla traccia - ci permettono di scandire adeguatamente le locazioni di memoria da cui prelevare il dato da inviare ed in cui inserire quello ricevuto. Il contatore del ricevitore

(*Counter_Receiver*), in particolare, prende in ingresso il segnale RDA dall'UART attraverso il *signal* interno al sistema *UARTToMemoryOut*.

Riportiamo ora l'automa a stati finiti realizzato per l'implementazione della componente *Delay*.

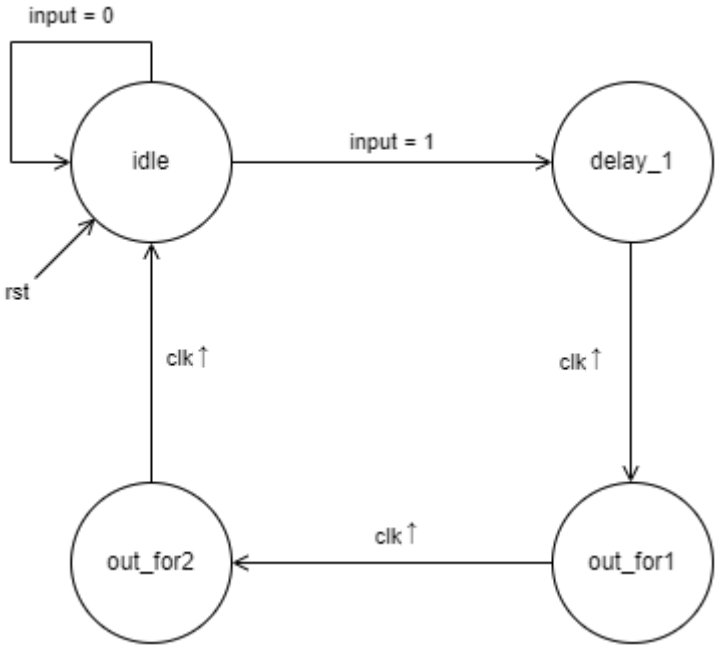


Figura 9.2 - Automa a stati finiti del Delay

Tale componente viene collegata direttamente all'uscita del segnale di debouncer per andare in ingresso alla porta *WR* dell'UART tramite il *signal* interno *DelayToUART*. Come spiegato sopra, tale componente rallenta il segnale di un colpo di clock durante la comunicazione in modo da assicurare che la ROM abbia il tempo necessario per poter correttamente predisporre il valore in uscita ed inoltre è utilizzato anche per "adattare" il segnale alla frequenza divisa data in ingresso all'UART mantenendolo alto per un ulteriore colpo di clock.

Importante notare che, siccome il segnale di *strobe* inviato dal debouncer è direttamente collegato sia alla ROM che al contatore del trasmettitore (oltre che al Delay), una volta che il contatore incrementerà il suo valore, la ROM avrà già avuto il tempo per effettuare l'invio del dato precedentemente puntato. Allo stesso modo, dall'altro lato, il segnale RDA della ROM è direttamente collegato sia al relativo contatore che alla componente *Out_Mem*. Una volta che il contatore avrà

incrementato il valore, la componente *Out_Mem* avrà, in realtà, già inviato in uscita il precedente valore. Questo accade perché i contatori, in quanto macchine combinatorie, sono più lente rispetto ai segnali mandati direttamente nelle altre componenti.

Riportiamo di seguito i codici VHDL delle tre componenti appena descritte: *Delay*, *counter* ed *Out_Mem.*

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY Delay IS
	PORT
	(
		input  : IN STD_LOGIC;
		rst    : IN STD_LOGIC;
		clk    : IN STD_LOGIC;
		output : OUT STD_LOGIC
	);
END Delay;

ARCHITECTURE Behavioral OF Delay IS

	TYPE stato IS (idle, delay_1, out_for1, out_for2);
	SIGNAL current_state, next_state : stato;
BEGIN
	PROCESS (clk)
	BEGIN
		IF clk = '1' AND clk'EVENT THEN
			IF RST = '1' THEN
				current_state <= idle;
			ELSE
				current_state <= next_state;
			END IF;
		END IF;
	END PROCESS;


	PROCESS (input, current_state)
		BEGIN
			CASE current_state IS

				WHEN idle =>
					output <= '0';
					IF input = '1' THEN
						next_state <= delay_1;
					ELSE
						next_state <= current_state;
					END IF;

				WHEN delay_1 =>
					output     <= '0';
					next_state <= out_for1;
```

```
                        WHEN out_for1 =>
                               output     <= '1';
                               next_state <= out_for2;

                        WHEN out_for2 =>
                               output     <= '1';
                               next_state <= idle;
                    END CASE;
             END PROCESS;
END Behavioral;
```

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.math_real.ALL;
USE IEEE.numeric_std.ALL;
USE IEEE.std_logic_unsigned.ALL;

ENTITY counter IS
       GENERIC
       (
              n_string : INTEGER := 4;
              m_bit    : INTEGER := 8
       );
       PORT
       (
              count_in       : IN std_logic;
              clk            : IN std_logic;
              rst            : IN std_logic;
              index_selector : OUT
std_logic_vector(INTEGER(ceil(log2(real(n_string)))) - 1 DOWNTO 0)
       );
END counter;

ARCHITECTURE Behavioral OF counter IS

BEGIN
       PROCESS (clk)
       VARIABLE count_v : INTEGER := 0;
       BEGIN
              IF rising_edge(clk) THEN
                     index_selector <= std_logic_vector(to_unsigned(count_v,
INTEGER(ceil(log2(real(n_string))))));
                     IF (count_in = '1') THEN
                            count_v := count_v + 1;
                     END IF;

                     IF (count_v = n_string) THEN
                            count_v := 0;
                     END IF;
              END IF;
       END PROCESS;

END Behavioral;
```

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.numeric_std.ALL;
USE IEEE.std_logic_unsigned.ALL;
USE IEEE.math_real.ALL;

ENTITY Out_Mem IS
	GENERIC
	(
		n_string : INTEGER := 4;
		m_bit    : INTEGER := 8
	);
	PORT
	(
		clk          : IN std_logic;
		write_enable : IN std_logic;
		address      : IN std_logic_vector;
		data_in      : IN std_logic_vector(m_bit - 1 DOWNTO 0);
		data_out     : OUT std_logic_vector(m_bit - 1 DOWNTO 0)
	);
END Out_Mem;

ARCHITECTURE Behavioral OF Out_Mem IS
	TYPE valori IS ARRAY (n_string - 1 DOWNTO 0) OF unsigned(m_bit - 1
DOWNTO 0);
	SIGNAL ram                 : valori;
	SIGNAL read_address        : std_logic_vector(address'RANGE);
	CONSTANT default_RAM_value : std_logic_vector(m_bit - 1 DOWNTO 0) :=
"00000000";

BEGIN
	RamProc       : PROCESS (clk) IS
		VARIABLE flag : INTEGER := 0;
	BEGIN
		IF rising_edge(clk) THEN
			IF (flag = 0) THEN
				data_out <= default_RAM_value;
			END IF;

			IF (write_enable = '1') THEN
				flag := 1;
				ram(to_integer(unsigned(address))) <=
unsigned(data_in);
				read_address                       <= address;
				data_out                           <=
std_logic_vector(ram(to_integer(unsigned(read_address))));
			END IF;
		END IF;
	END PROCESS RamProc;

END Behavioral;
```

Notiamo che il segnale *data_out* ci permette di visualizzare sui led l'ultimo valore salvato in memoria. Inoltre, notiamo l'utilizzo dell'attributo *"RANGE"* per definire automaticamente la dimensione del *signal read_address* in base alla dimensione (generica) del *signal address*.

Di seguito riportiamo, infine, lo schematico ed il codice VHDL della top-module.

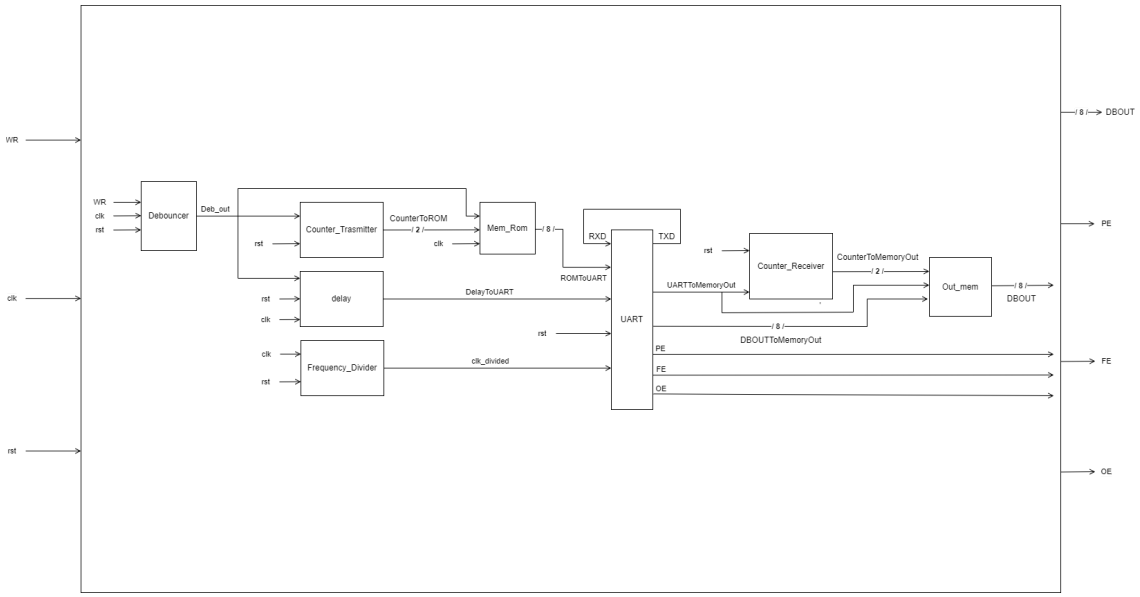


Figura 9.3 - Schematico della top-module

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE ieee.numeric_std.ALL;
USE ieee.std_logic_unsigned.ALL;
USE IEEE.math_real.ALL;

ENTITY Serial_System IS
      GENERIC
      (
            n_string : INTEGER := 4;
            m_bit    : INTEGER := 8
      );
      PORT
      (

            -- Strobe
            WR  : IN std_logic; -- Tramite pulsante

            CLK : IN std_logic;
            RST : IN std_logic;

            -- Dati di uscita in parallelo
            DBOUT : OUT std_logic_vector(7 DOWNTO 0); -- Led per l'uscita
            -- Bit di errori (visualizzati sempre tramite led)
```

```
            PE : OUT std_logic;
            FE : OUT std_logic;
            OE : OUT std_logic
      );
END Serial_System;

ARCHITECTURE Structural OF Serial_System IS
      COMPONENT Rs232RefComp IS
            GENERIC
            (
                  BAUD_DIVIDE : INTEGER := 54;
                  BAUD_RATE   : INTEGER := 868
            );
            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(è alto
quando il dato è disponibile nel registro rdReg)
                  TBE   : INOUT std_logic := '1'; --Transfer Bus Empty(se
è 0 significa che il bus è occupato per una trasmissione)
                  RD    : IN std_logic; --Read Strobe(se è 0 significa
"leggi")
                  WR    : IN std_logic; --Write Strobe(se è 1 significa
"scrivi")
                  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 Frequency_Divider IS
            PORT
            (
                  clk_in      : IN std_logic;
                  RST         : IN std_logic;
                  clk_divided : OUT std_logic
            );
      END COMPONENT;

      COMPONENT ROM IS
            GENERIC
            (
                  n_string : INTEGER := 4;
                  m_bit    : INTEGER := 8
            );
            PORT
            (
                  addr_i   : IN
std_logic_vector(INTEGER(ceil(log2(real(n_string)))) - 1 DOWNTO 0);
                  strobe   : IN std_logic;
```

```
                clk      : IN std_logic;
                data_out : OUT std_logic_vector(m_bit - 1 DOWNTO 0)
        );
    END COMPONENT;

    COMPONENT debouncer IS
        GENERIC
        (
                CLK_period     : INTEGER := 10;
                btn_noise_time : INTEGER := 100
        );
        PORT
        (
                read       : IN STD_LOGIC;
                rst        : IN STD_LOGIC;
                clk        : IN STD_LOGIC;
                read_clear : OUT STD_LOGIC
        );
    END COMPONENT;

    COMPONENT delay IS
        PORT
        (
                input  : IN STD_LOGIC;
                rst    : IN STD_LOGIC;
                clk    : IN STD_LOGIC;
                output : OUT STD_LOGIC
        );
    END COMPONENT;

    COMPONENT counter IS
        GENERIC
        (
                n_string : INTEGER := 4;
                m_bit    : INTEGER := 8
        );
        PORT
        (
                count_in       : IN std_logic;
                rst            : IN std_logic;
                index_selector : OUT
std_logic_vector(INTEGER(ceil(log2(real(n_string)))) - 1 DOWNTO 0)
        );
    END COMPONENT;

    COMPONENT Out_Mem IS
        GENERIC
        (
                n_string : INTEGER := 4;
                m_bit    : INTEGER := 8
        );
        PORT
        (
                clk          : IN std_logic;
                write_enable : IN std_logic;
                address      : IN std_logic_vector;
```

```
                    data_in      : IN std_logic_vector(7 DOWNTO 0);
                    data_out     : OUT std_logic_vector(7 DOWNTO 0) -- lo
utilizziamo per visualizzare sui led l'ultimo valore salvato in memoria
             );
      END COMPONENT;

      SIGNAL TRXD                : std_logic;
      SIGNAL clk_divided         : std_logic;
      SIGNAL CounterToROM        :
std_logic_vector(INTEGER(ceil(log2(real(n_string)))) - 1 DOWNTO 0);
      SIGNAL Deb_out             : std_logic;
      SIGNAL ROMToUART           : std_logic_vector(m_bit - 1 DOWNTO 0);
      SIGNAL DelayToUART         : std_logic;
      SIGNAL UARTToMemoryOut     : std_logic := '0';
      SIGNAL CounterToMemoryOut  :
std_logic_vector(INTEGER(ceil(log2(real(n_string)))) - 1 DOWNTO 0);
      SIGNAL DBOUTToMemoryOut    : std_logic_vector (7 DOWNTO 0);

      SIGNAL TBEtoReciver        : std_logic;
      SIGNAL TBEtoReciver_negato : std_logic;

BEGIN
      TBEtoReciver_negato <= NOT TBEtoReciver;

      UART : Rs232RefComp
             GENERIC MAP
             (
             BAUD_DIVIDE => 54,
             BAUD_RATE   => 868
             )
             PORT MAP
             (
                    TXD   => TRXD,
                    RXD   => TRXD,
                    CLK   => clk_divided,
                    DBIN  => ROMToUART,
                    DBOUT => DBOUTToMemoryOut,
                    RDA   => UARTToMemoryOut,
                    TBE   => TBEtoReciver,
                    RD    => TBEtoReciver_negato,
                    WR    => DelayToUART,
                    PE    => PE,
                    FE    => FE,
                    OE    => OE,
                    RST   => RST
             );

                    Freq_Div : Frequency_Divider
                    PORT MAP
                    (
                           CLK_IN      => CLK,
                           RST         => RST,
                           CLK_DIVIDED => CLK_DIVIDED
      );

      Deb_Trasmitter : debouncer
```

```
        GENERIC MAP
        (
        CLK_period     => 10,
        btn_noise_time => 100
        )
        PORT MAP
        (
              read       => WR,
              rst        => rst,
              clk        => clk,
              read_clear => Deb_out
        );

              Mem_Rom : ROM
                    GENERIC MAP
                    (
                    n_string => 4,
                    m_bit    => 8
                    )
                    PORT MAP
                    (
                          addr_i   => CounterToROM,
                          strobe   => Deb_out,
                          clk      => clk,
                          data_out => ROMToUART
                    );

                          Delay_Trasmitter : delay
                          PORT MAP
                          (
                                input  => Deb_out,
                                rst    => RST,
                                clk    => CLK,
                                output => DelayToUART
);

Counter_Trasmitter : counter
        GENERIC MAP
        (
        n_string => 4,
        m_bit    => 8
        )
        PORT MAP
        (
              count_in       => Deb_out,
              rst            => RST,
              index_selector => CounterToROM
        );

              Counter_Receiver : counter
                    GENERIC MAP
                    (
                    n_string => 4,
                    m_bit    => 8
                    )
                    PORT MAP
```

```
                            (
                                   count_in       => UARTToMemoryOut,
                                   rst            => RST,
                                   index_selector => CounterToMemoryOut
                            );

                                   Memory_Out : Out_Mem
                                          GENERIC MAP
                                          (
                                          n_string => 4,
                                          m_bit    => 8
                                          )
                                          PORT MAP
                                          (
                                                 clk          => clk,
                                                 write_enable =>
UARTToMemoryOut,
                                                 address      =>
CounterToMemoryOut,
                                                 data_in      =>
DBOUTToMemoryOut,
                                                 data_out     => DBOUT
                                          );

END Structural;
```

Per consentire un corretto funzionamento della componente UART è stato necessario effettuare in modo opportuno il port map con i relativi segnali di ingresso/uscita. Nello specifico abbiamo utilizzato un signal *TRXD* per collegare le porte TXD ed RXD, che rappresenta il canale seriale utilizzato per la comunicazione. Ulteriore *signal* utilizzato per il corretto funzionamento della component è *TBEToReciver_negato,* utilizzato per portare in ingresso ad RD (read) il valore negato di TBE (uguale a 0 nel momento in cui il bus è occupato per una trasmissione), in quanto RD risulta essere un segnale 0 attivo (quando impostato a 0, verrà sempre effettuata la lettura).

9.2.3 Simulazione

Per il testing del sistema realizzato, abbiamo stimolato più volte il segnale *WR* per avviare svariate trasmissioni. Riportiamo di seguito il codice del testbench.

```
stimulus : PROCESS
BEGIN
       WAIT FOR 100ns;
       WR <= '0';
       WAIT FOR 10ns;
       WR <= '1';
       WAIT FOR 200ns;
```

```
        WR <= '0';
        WAIT FOR 2000000ns;
        WR <= '1';
        WAIT FOR 20ns;
        WR <= '0';
        WAIT FOR 2000000ns;
        WR <= '1';
        WAIT FOR 10ns;
        WR <= '0';
        WAIT FOR 2000000ns;
        WR <= '1';
        WAIT FOR 10ns;
        WR <= '0';
        WAIT FOR 2000000ns;
        WR <= '1';
        WAIT FOR 10ns;
        WR <= '0';
        WAIT;
END PROCESS;

clock_generator : PROCESS
BEGIN
        clk <= '0';
        WAIT FOR clk_period/2;
        clk <= '1';
        WAIT FOR clk_period/2;
END PROCESS;
```

Di seguito riportiamo il grafico ottenuto dalla simulazione.

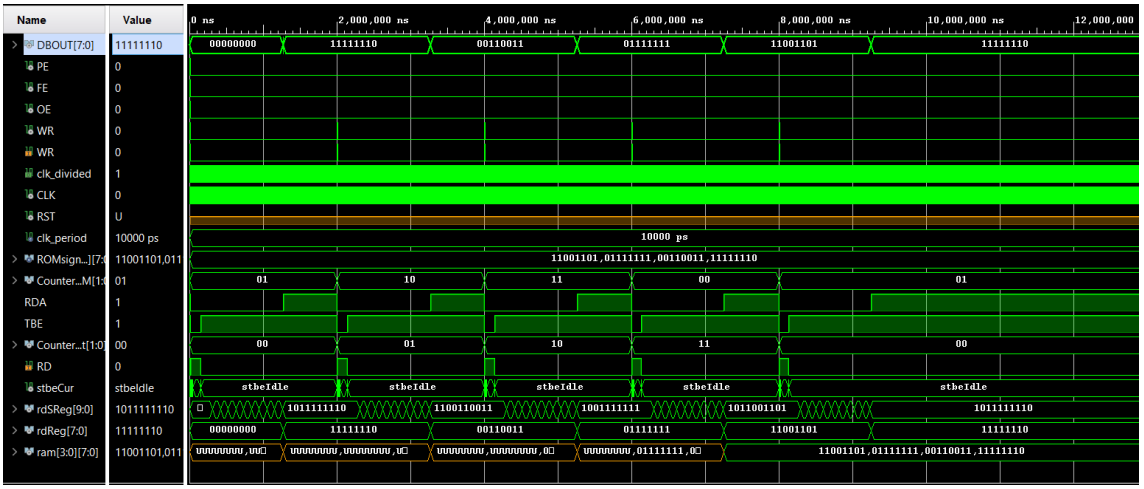


Figura 9.4 - Grafico della simulazione

Da notare che, per facilitare la leggibilità del grafico, il test in figura presenta un valore *btn_noise_time* del debouncer nettamente ridotto rispetto a quello utilizzato per la successiva fase di sintesi. Dal grafico si può notare che, stimolando più volte il valore

WR, le locazioni di memoria del registro *Out_Mem* si riempiranno una dopo l'altra nel giusto ordine, dopo un certo ritardo. I valori con cui si riempirà il registro saranno quelli trasmessi e precedentemente situati nella ROM del sistema A. I valori contenuti all'interno delle memorie dei sistemi A e B sono visualizzabili nel grafico.

9.2.4 Sintesi

Fornendo i valori da inviare direttamente dalla ROM, rispetto all'esercizio precedente, per la fase di sintesi abbiamo rimosso il mapping degli switch. Riportiamo di seguito il file relativo alle constraints realizzato.

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

## LEDs
set_property -dict { PACKAGE_PIN H17   IOSTANDARD LVCMOS33 } [get_ports {
DBOUT[0] }]; #IO_L18P_T2_A24_15 Sch=led[0]
set_property -dict { PACKAGE_PIN K15   IOSTANDARD LVCMOS33 } [get_ports {
DBOUT[1] }]; #IO_L24P_T3_RS1_15 Sch=led[1]
set_property -dict { PACKAGE_PIN J13   IOSTANDARD LVCMOS33 } [get_ports {
DBOUT[2] }]; #IO_L17N_T2_A25_15 Sch=led[2]
set_property -dict { PACKAGE_PIN N14   IOSTANDARD LVCMOS33 } [get_ports {
DBOUT[3] }]; #IO_L8P_T1_D11_14 Sch=led[3]
set_property -dict { PACKAGE_PIN R18   IOSTANDARD LVCMOS33 } [get_ports {
DBOUT[4] }]; #IO_L7P_T1_D09_14 Sch=led[4]
set_property -dict { PACKAGE_PIN V17   IOSTANDARD LVCMOS33 } [get_ports {
DBOUT[5] }]; #IO_L18N_T2_A11_D27_14 Sch=led[5]
set_property -dict { PACKAGE_PIN U17   IOSTANDARD LVCMOS33 } [get_ports {
DBOUT[6] }]; #IO_L17P_T2_A14_D30_14 Sch=led[6]
set_property -dict { PACKAGE_PIN U16   IOSTANDARD LVCMOS33 } [get_ports {
DBOUT[7] }]; #IO_L18P_T2_A12_D28_14 Sch=led[7]
set_property -dict { PACKAGE_PIN V16   IOSTANDARD LVCMOS33 } [get_ports {
PE }]; #IO_L16N_T2_A15_D31_14 Sch=led[8]
set_property -dict { PACKAGE_PIN T15   IOSTANDARD LVCMOS33 } [get_ports {
FE }]; #IO_L14N_T2_SRCC_14 Sch=led[9]
set_property -dict { PACKAGE_PIN U14   IOSTANDARD LVCMOS33 } [get_ports {
OE }]; #IO_L22P_T3_A05_D21_14 Sch=led[10]

##Buttons
set_property -dict { PACKAGE_PIN N17   IOSTANDARD LVCMOS33 } [get_ports {
WR }]; #IO_L9P_T1_DQS_14 Sch=btnc
set_property -dict { PACKAGE_PIN M18   IOSTANDARD LVCMOS33 } [get_ports {
RST }]; #IO_L4N_T0_D05_14 Sch=btnu
```

Capitolo 10

Switch multistadio

Lo switch è un sistema che consente di collegare *n* sorgenti ad *n* destinazioni. Lo switch può essere facilmente realizzato attraverso multiplexer e demultiplexer.

Per creare uno switch a connessione *diretta* tra due nodi diversi, è necessario conoscere l'indirizzo del nodo sorgente e l'indirizzo del nodo destinazione. Per poter realizzare tale architettura, dunque, è necessario scomporre il sistema in due sottosistemi diversi, quello di ingresso e quello di uscita. Proprio per questo, vengono utilizzati multiplexer e demultiplexer: i primi per realizzare l'ingresso, gli altri per l'uscita. Questo sistema ha un vantaggio poiché consente di avere una connessione diretta tra sorgente e destinazione perché vi è assenza di stadi intermedi e quindi è garantita una buona efficienza però può comunicare solamente un'entità alla volta (dunque, non è possibile realizzare una comunicazione broadcast). Il parallelismo può essere ottenuto solo riproducendo lo stesso hardware più volte.

Un'altra soluzione architetturale per poter realizzare uno switch si basa sull'utilizzo di più stadi intermedi per stabilire la comunicazione tra nodi e garantire il parallelismo. Per la realizzazione di reti multistadio sono state proposte, nel tempo, varie topologie. In particolare, notiamo che utilizzando dispositivi costituiti da 2 ingressi e 2 uscite per realizzare gli stadi con $M = log_2 n$ stadi è possibile risolvere il problema della connettività. La soluzione che andremo a considerare è quella dell'Omega Network. Nel costruire questa rete bisogna andare a gestire le connessioni attraverso il "*perfect shuffling*" (mescolamento perfetto). Così facendo siamo in grado di connettere tutte le sorgenti a tutte le destinazioni con al più *M* passi. In questo modo, se ci poniamo in un qualsiasi nodo sorgente e vogliamo arrivare ad un nodo destinazione esprimiamo in binario l'indirizzo che si vuole raggiungere. Ad ogni passo decidiamo se bisogna considerare l'uscita "superiore" o "inferiore" del demultiplexer in relazione al valore del bit (0 → sopra; 1 → sotto) riferito all'indirizzo di destinazione espresso in binario.

Tale rete può essere utilizzata per inviare più messaggi contemporaneamente. In questo caso, bisognerà considerare, però, la presenza di eventuali collisioni, situazioni in cui un nodo può ricevere contemporaneamente 2 messaggi diversi ed entrare dunque in conflitto. Volendo inviare più messaggi contemporaneamente, bisognerà necessariamente gestire le collisioni altrimenti il sistema potrebbe non funzionare. Tale problematica ci ha particolarmente entusiasmato e, per questo, sarà affrontata nel dettaglio nel capitolo 12 di questo elaborato, dedicato all'esercizio "libero" da noi realizzato.

10.1 Switch multistadio: modello Omega Network

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 (ed. nodo 1 più prioritario, con priorità decrescenti fino al nodo 4).

10.1.1 Soluzione

In accordo con il modello Omega Network, abbiamo realizzato un'unità operativa del sistema componendola di una serie di switch2-2. La rete - come richiesto dalla traccia - è costituita da 4 nodi per cui avremo 2 nodi di ingresso e 2 di uscita.

Inoltre, abbiamo realizzato un'altra componente per il controllo "centralizzato" dell'Omega Network. In particolare, per poter gestire eventuali situazioni di invio concorrente di dati, abbiamo inserito un multiplexer che - tramite un arbitro che gestisce la priorità come specificato nella traccia - seleziona l'informazione con priorità maggiore. Tale informazione è rappresentata su 6 bit: i 2 bit più significativi rappresentano il nodo sorgente, i successivi due bit rappresentano il nodo destinazione ed, infine, i due bit meno significativi costituiscono il *payload* stesso dell'informazione che si vuole trasmettere. Tale informazione viene quindi inviata ad un demultiplexer che selezionerà la sorgente della nostra trasmissione.

Volendo inserire un primo schematico per dare un'idea di quanto poi realizzato ed analizzato nel dettaglio nel successivo paragrafo, riportiamo il seguente con lo scopo di evidenziare come interagiscono le due componenti principali del sistema complessivo, viste al momento ancora come delle *"black box"* per evitare di dover descrivere ulteriori dettagli relative alle componenti più piccole e che sono inerenti, invece, alla fase di realizzazione.

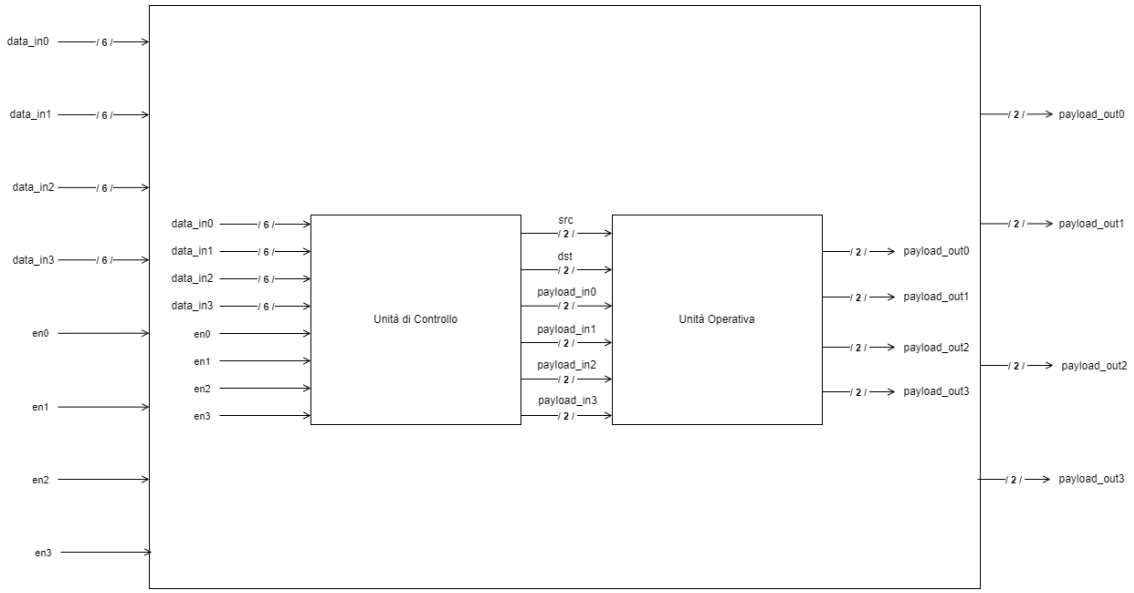


Figura 10.1 - Schematico della top-module "black box"

10.1.2 Realizzazione

Partendo da quella che abbiamo chiamato "unità operativa" del sistema, abbiamo realizzato lo schematico che riportiamo di seguito per attuare il "*perfect shuffling*" richiesto da una Omega Network per consentire ad una qualsiasi sorgente di poter raggiungere una qualunque destinazione.

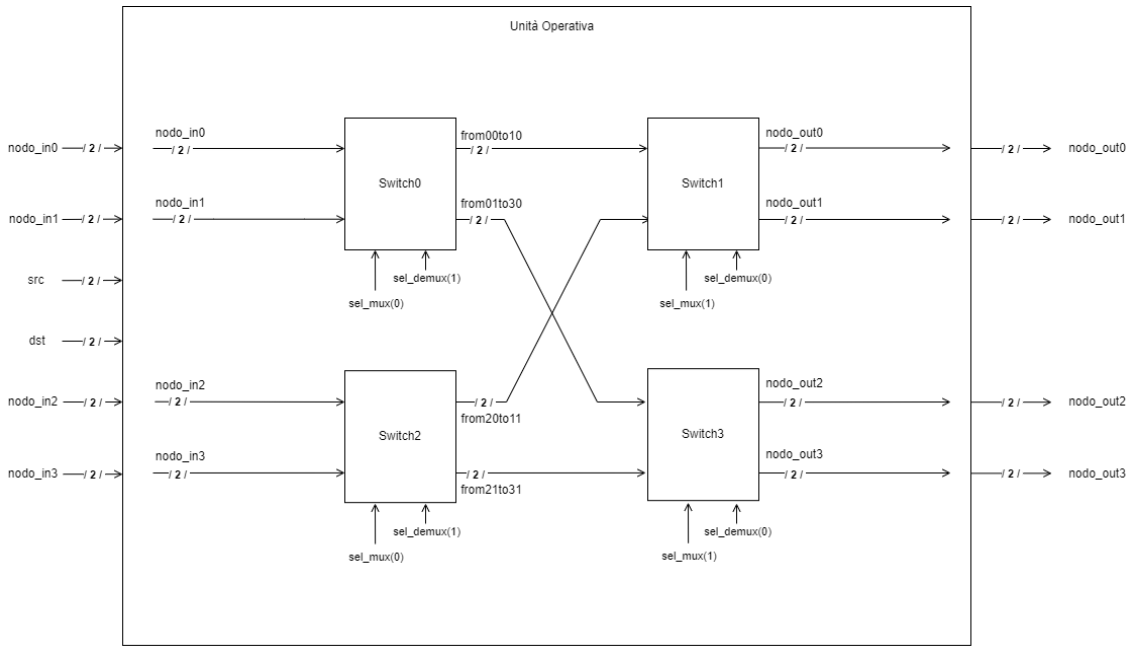


Figura 10.2 - Schematico della "componente operativa"

Notiamo che, oltre ai *payload* che vogliamo inviare e che quindi entrano in ingresso alla rete, i restanti due ingressi - *src* e *dst* - provengono direttamente dalla componente "di controllo" introdotta nel paragrafo precedente e che approfondiremo nel dettaglio in seguito. Tali due ingressi sono fondamentali per effettuare correttamente la selezione dei multiplexer e demultiplexer contenuti all'interno dei singoli switch2-2 dei vari stadi che compongono la rete e, dunque, per portare l'informazione sull'uscita desiderata. Riportiamo di seguito il codice VHDL realizzato per gli switch.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY switch2_2 IS
      PORT
      (
            input0    : IN std_logic_vector(1 DOWNTO 0);
            input1    : IN std_logic_vector(1 DOWNTO 0);
            sel_mux   : IN std_logic;
            sel_demux : IN std_logic;
            output0   : OUT std_logic_vector(1 DOWNTO 0);
            output1   : OUT std_logic_vector(1 DOWNTO 0)
      );
END switch2_2;

ARCHITECTURE Behavioral OF switch2_2 IS
BEGIN
      PROCESS (input0, input1, sel_mux, sel_demux)
```

```
	BEGIN
		output0 <= "--";
			output1 <= "--";
			IF (sel_mux = '0') THEN
				IF (sel_demux = '0') THEN
					output0 <= input0;
			ELSIF (sel_demux = '1') THEN
				output1 <= input0;
			ELSE
				output0 <= "--";
					output1 <= "--";
				END IF;
		ELSIF (sel_mux = '1') THEN
			IF (sel_demux = '0') THEN
				output0 <= input1;
			ELSIF (sel_demux = '1') THEN
				output1 <= input1;
			ELSE
				output0 <= "--";
					output1 <= "--";
				END IF;
		ELSE
			output0 <= "--";
				output1 <= "--";
			END IF;
	END PROCESS;

END Behavioral;
```

Per consentire al tool di effettuare la corretta costruzione dell'Omega Network, abbiamo poi realizzato i seguenti collegamenti nel file relativo all'istanza delle componenti e della corrispondente port map.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY ParteOperativaRete IS
	PORT
	(
		nodo0in, nodo1in, nodo2in, nodo3in	: IN std_logic_vector(1
DOWNTO 0);
		sel_mux				: IN std_logic_vector(1
DOWNTO 0);
		sel_demux				: IN std_logic_vector(1
DOWNTO 0);
		nodo0out, nodo1out, nodo2out, nodo3out : OUT
std_logic_vector(1 DOWNTO 0)
	);
END ParteOperativaRete;

ARCHITECTURE structural OF ParteOperativaRete IS

	COMPONENT switch2_2 IS
```

```
            PORT
            (
                  input0    : IN std_logic_vector(1 DOWNTO 0);
                  input1    : IN std_logic_vector(1 DOWNTO 0);
                  sel_mux   : IN std_logic;
                  sel_demux : IN std_logic;
                  output0   : OUT std_logic_vector(1 DOWNTO 0);
                  output1   : OUT std_logic_vector(1 DOWNTO 0)
            );
      END COMPONENT;

      SIGNAL from00to10 : std_logic_vector(1 DOWNTO 0);
      SIGNAL from20to11 : std_logic_vector(1 DOWNTO 0);
      SIGNAL from01to30 : std_logic_vector(1 DOWNTO 0);
      SIGNAL from21to31 : std_logic_vector(1 DOWNTO 0);

BEGIN
      switch0 : switch2_2
      PORT MAP
      (
            input0    => nodo0in,
            input1    => nodo1in,
            sel_mux   => sel_mux(0),
            sel_demux => sel_demux(1),
            output0   => from00to10,
            output1   => from01to30
      );

      switch1 : switch2_2
      PORT MAP
      (
            input0    => from00to10,
            input1    => from20to11,
            sel_mux   => sel_mux(1),
            sel_demux => sel_demux(0),
            output0   => nodo0out,
            output1   => nodo1out
      );

      switch2 : switch2_2
      PORT MAP
      (
            input0    => nodo2in,
            input1    => nodo3in,
            sel_mux   => sel_mux(0),
            sel_demux => sel_demux(1),
            output0   => from20to11,
            output1   => from21to31
      );

      switch3 : switch2_2
      PORT MAP
      (
            input0    => from01to30,
            input1    => from21to31,
            sel_mux   => sel_mux(1),
```

```
            sel_demux => sel_demux(0),
            output0   => nodo2out,
            output1   => nodo3out
      );

END structural;
```

Per quanto concerne, invece, quella che abbiamo definito “componente di controllo dell’Omega Network”, come spiegato in precedenza, abbiamo realizzato tre differenti componenti: un multiplexer, un demultiplexer ed un arbitro per la scelta della priorità. Innanzitutto, ne riportiamo di seguito lo schematico.

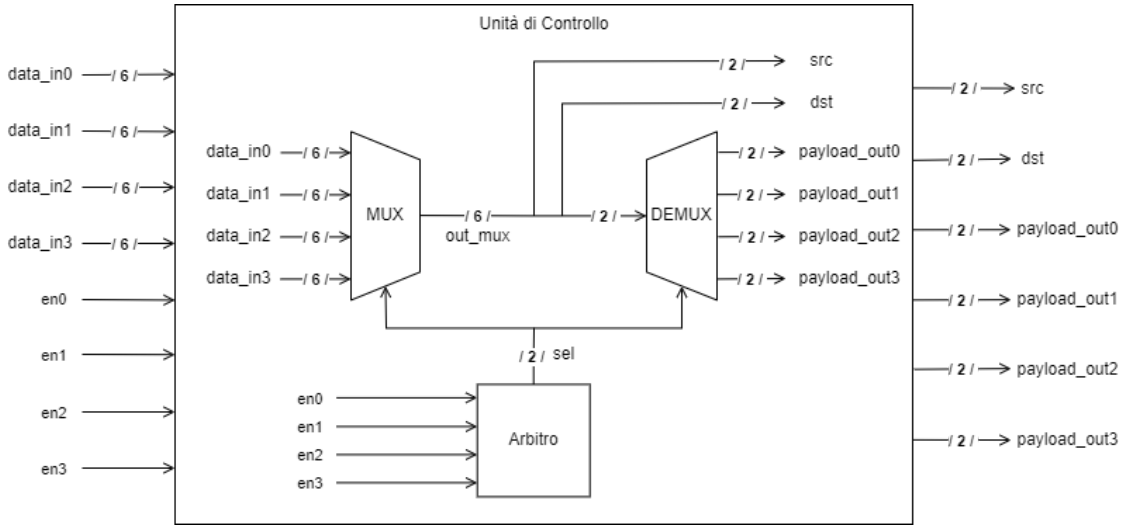


Figura 10.3 - Schematico della “componente di controllo”

Notiamo che l’uscita del multiplexer *out_mux* da 6 bit viene divisa in tre parti: i due bit meno significativi vanno in ingresso al demultiplexer che, tramite il segnale di selezione dato dall’arbitro, selezionerà l’uscita in cui mandare il dato e, dunque, la sorgente da cui la trasmissione parte all’interno dell’Omega Netwotk. Gli altri 4 bit di *out_mux* costituiscono i segnali *src* e *dst* - rispettivamente sorgente e destinazione dell’Omega Network - che, come spiegato sopra, verranno utilizzati come selettori dei multiplexer e demultiplexer degli switch dell’Omega Network stessa.

Evitiamo di mostrare l'implementazione del multiplexer (realizzato con approccio data-flow) e quello del demultiplexer (realizzato, invece, con approccio puramente comportamentale), riportando di seguito, invece, il codice VHDL della componente arbitro.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY arbiter IS
	PORT
	(
		enable0 : IN std_logic;
		enable1 : IN std_logic;
		enable2 : IN std_logic;
		enable3 : IN std_logic;
		output  : OUT std_logic_vector(1 DOWNTO 0)
	);
END arbiter;
ARCHITECTURE dataflow OF arbiter IS
BEGIN
	output <= "00" WHEN enable0 = '1' ELSE
		  "01" WHEN enable1 = '1' ELSE
		  "10" WHEN enable2 = '1' ELSE
		  "11" WHEN enable3 = '1' ELSE
		  "--";
END dataflow;
```

L'approccio che abbiamo scelto di adottare è di tipo data-flow. Notiamo che la priorità viene assegnata - come richiesto dalla traccia - dal primo all'ultimo ingresso. In questo modo, se due segnali differenti volessero "parlare", verrebbe privilegiato il primo in ordine topologico.

Riportiamo, quindi, le istanze e le port map dei componenti finora mostrati.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY ParteControllo IS
	PORT
	(
		data_in0     : IN std_logic_vector(5 DOWNTO 0);
		data_in1     : IN std_logic_vector(5 DOWNTO 0);
		data_in2     : IN std_logic_vector(5 DOWNTO 0);
		data_in3     : IN std_logic_vector(5 DOWNTO 0);
		en0          : IN std_logic;
		en1          : IN std_logic;
		en2          : IN std_logic;
		en3          : IN std_logic;
		payload_out0 : OUT std_logic_vector(1 DOWNTO 0);
		payload_out1 : OUT std_logic_vector(1 DOWNTO 0);
		payload_out2 : OUT std_logic_vector(1 DOWNTO 0);
		payload_out3 : OUT std_logic_vector(1 DOWNTO 0);
		src          : OUT std_logic_vector(1 DOWNTO 0);
		dst          : OUT std_logic_vector(1 DOWNTO 0)
	);
END ParteControllo;
```

```
ARCHITECTURE structural OF ParteControllo IS

	COMPONENT mux_4_1 IS
		PORT
		(
			input0 : IN std_logic_vector(5 DOWNTO 0);
			input1 : IN std_logic_vector(5 DOWNTO 0);
			input2 : IN std_logic_vector(5 DOWNTO 0);
			input3 : IN std_logic_vector(5 DOWNTO 0);
			sel    : IN std_logic_vector(1 DOWNTO 0);
			output : OUT std_logic_vector(5 DOWNTO 0)
		);
	END COMPONENT;

	COMPONENT demux_1_4 IS
		PORT
		(
			input   : IN std_logic_vector(1 DOWNTO 0);
			sel     : IN std_logic_vector(1 DOWNTO 0);
			output0 : OUT std_logic_vector(1 DOWNTO 0);
			output1 : OUT std_logic_vector(1 DOWNTO 0);
			output2 : OUT std_logic_vector(1 DOWNTO 0);
			output3 : OUT std_logic_vector(1 DOWNTO 0)
		);
	END COMPONENT;

	COMPONENT arbiter IS
		PORT
		(
			enable0 : IN std_logic;
			enable1 : IN std_logic;
			enable2 : IN std_logic;
			enable3 : IN std_logic;
			output  : OUT std_logic_vector(1 DOWNTO 0)
		);
	END COMPONENT;
	SIGNAL sel     : std_logic_vector(1 DOWNTO 0);
	SIGNAL out_mux : std_logic_vector(5 DOWNTO 0); --uscita multiplexer
con informazione src dest e dato
BEGIN
	mux : mux_4_1
	PORT MAP
	(
		input0 => data_in0,
		input1 => data_in1,
		input2 => data_in2,
		input3 => data_in3,
		sel    => sel,
		output => out_mux
	);

	demux : demux_1_4
	PORT MAP
	(
		input   => out_mux(1 DOWNTO 0),
		sel     => sel,
```

```
            output0 => payload_out0,
            output1 => payload_out1,
            output2 => payload_out2,
            output3 => payload_out3
      );

      priority_arbiter : arbiter
      PORT MAP
      (
            enable0 => en0,
            enable1 => en1,
            enable2 => en2,
            enable3 => en3,
            output  => sel
      );

      src <= out_mux(5 DOWNTO 4);
      dst <= out_mux(3 DOWNTO 2);

END structural;
```

In ultima analisi, riportiamo di seguito uno schematico completo della top-module e la sua implementazione VHDL in modo da esplicitare tutti i collegamenti tra le varie componenti e gli ingressi e le uscite del sistema complessivo.

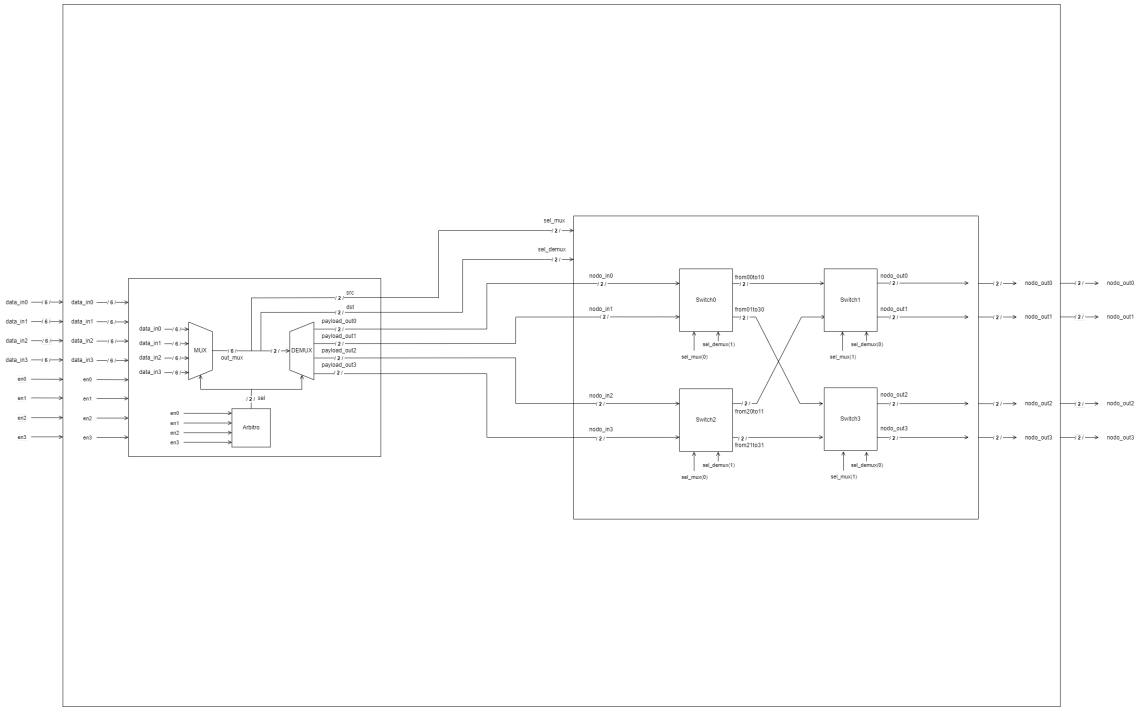


Figura 10.4 - Schematico della top-module

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
```

```
entity Top_Module is
  Port (
    data_in0 : in std_logic_vector(5 downto 0);
    data_in1 : in std_logic_vector(5 downto 0);
    data_in2 : in std_logic_vector(5 downto 0);
    data_in3 : in std_logic_vector(5 downto 0);
    en0 : in std_logic;
    en1 : in std_logic;
    en2 : in std_logic;
    en3 : in std_logic;
    payload_out0 : out std_logic_vector(1 downto 0);
    payload_out1 : out std_logic_vector(1 downto 0);
    payload_out2 : out std_logic_vector(1 downto 0);
    payload_out3 : out std_logic_vector(1 downto 0)
  );
end Top_Module;

architecture Structural of Top_Module is

    component ParteOperativaRete is
      Port (
        nodo0in,nodo1in,nodo2in,nodo3in : in std_logic_vector(1 downto 0);
        sel_mux : in std_logic_vector(1 downto 0);
        sel_demux : in std_logic_vector(1 downto 0);
        nodo0out,nodo1out,nodo2out,nodo3out : out std_logic_vector(1 downto
0)
       );
    end component;

    component ParteControllo is
      Port (
        --start : in std_logic;
        data_in0 : in std_logic_vector(5 downto 0);
        data_in1 : in std_logic_vector(5 downto 0);
        data_in2 : in std_logic_vector(5 downto 0);
        data_in3 : in std_logic_vector(5 downto 0);
        en0 : in std_logic;
        en1 : in std_logic;
        en2 : in std_logic;
        en3 : in std_logic;
        payload_out0 : out std_logic_vector(1 downto 0);
        payload_out1 : out std_logic_vector(1 downto 0);
        payload_out2 : out std_logic_vector(1 downto 0);
        payload_out3 : out std_logic_vector(1 downto 0);
        src : out std_logic_vector(1 downto 0);
        dst : out std_logic_vector(1 downto 0)
      );
    end component;

    signal payload_in0 : std_logic_vector(1 downto 0);
    signal payload_in1 : std_logic_vector(1 downto 0);
    signal payload_in2 : std_logic_vector(1 downto 0);
    signal payload_in3 : std_logic_vector(1 downto 0);
    signal src : std_logic_vector(1 downto 0);
    signal dst : std_logic_vector(1 downto 0);
```

```
begin

    CP : ParteControllo port map(
        data_in0 => data_in0,
        data_in1 => data_in1,
        data_in2 => data_in2,
        data_in3 => data_in3,
        en0 => en0,
        en1 => en1,
        en2 => en2,
        en3 => en3,
        payload_out0 => payload_in0,
        payload_out1 => payload_in1,
        payload_out2 => payload_in2,
        payload_out3 => payload_in3,
        src => src,
        dst => dst
    );

    omega_network : ParteOperativaRete port map(
        nodo0in => payload_in0,
        nodo1in => payload_in1,
        nodo2in => payload_in2,
        nodo3in => payload_in3,
        sel_mux => src,
        sel_demux => dst,
        nodo0out => payload_out0,
        nodo1out => payload_out1,
        nodo2out => payload_out2,
        nodo3out => payload_out3
    );

end Structural;
```

10.1.3 Simulazione

Al fine di effettuare correttamente il testing del sistema realizzato per verificarne il funzionamento, abbiamo deciso di inviare una serie di ingressi concorrenti e di far variare i segnali di abilitazione (*en0, en1, en2, en3*) per valutare la selezione da parte dell'arbitro in base alla priorità e la corretta consegna alla destinazione scelta del *payload* dell'informazione selezionata.

Riportiamo di seguito il codice del testbench realizzato.

```
stimulus: process
  begin
    wait for 100 ns;
    en0 <= '1';
    en1 <= '0';
    en2 <= '1';
```

```
    en3 <= '0';
    data_in0 <= "001011";
    data_in1 <= "011111";
    data_in2 <= "100010";
    data_in3 <= "110100";
    wait for 10 ns;
    en0 <= '0';
    en1 <= '0';
    en2 <= '1';
    en3 <= '0';
    data_in0 <= "001011";
    data_in1 <= "011111";
    data_in2 <= "100010";
    data_in3 <= "110100";
    wait for 10 ns;
    en0 <= '0';
    en1 <= '1';
    en2 <= '1';
    en3 <= '0';
    data_in0 <= "001011";
    data_in1 <= "011111";
    data_in2 <= "100010";
    data_in3 <= "110100";
    wait for 10 ns;
    wait;
  end process;
```

Di seguito riportiamo il grafico ottenuto da tale simulazione.

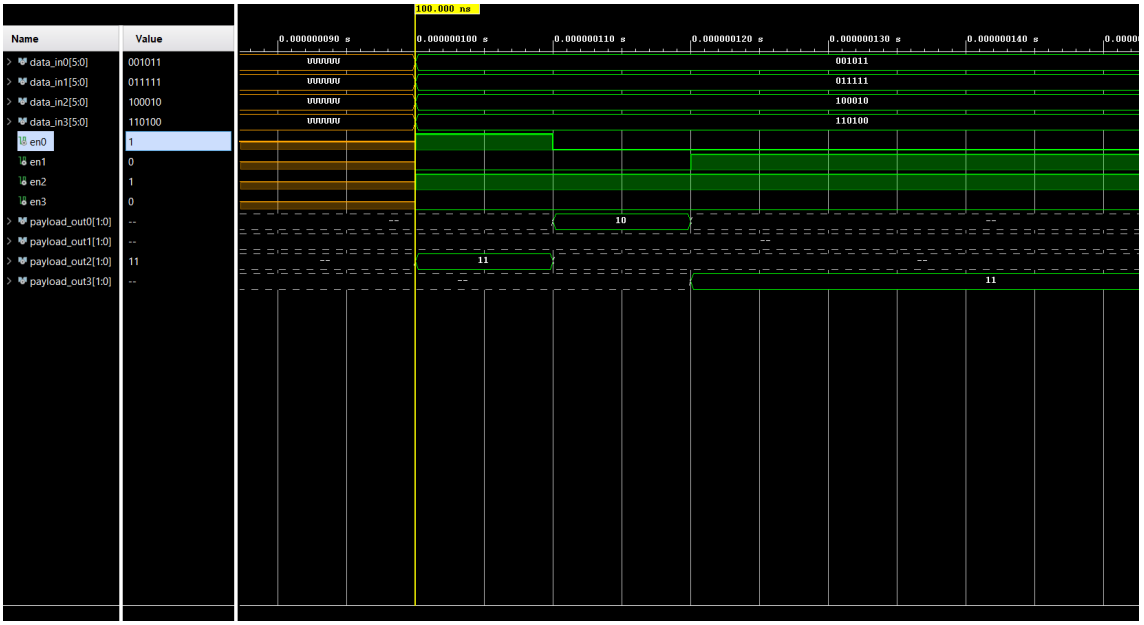


Figura 10.5 - Grafico della simulazione

Il testing è superato con successo. Come è possibile notare, infatti, tutti gli ingressi presentano un dato. Dopo aver atteso il passaggio dei transitori iniziali, alziamo il

valore di *en0* ed *en2* in modo da simulare la richiesta di voler trasmettere da parte dei rispettivi ingressi. L'arbitro - correttamente - seleziona l'abilitazione a priorità maggiore (*en0*, in questo caso). Il dato da trasmettere, quindi, è 001011. Come spiegato in precedenza, 00 indica il nodo sorgente dell'Omega Network, 10 quello destinazione ed il payload è 11. Dal grafico è possibile osservare che tale payload viene correttamente trasmesso all'uscita 2 (10 in binario). Nel secondo caso, l'unico valore di abilitazione alto è *en2*. Quest'unica richiesta viene servita mandando correttamente il payload 10 alla destinazione 0 (00 in binario). Nel terzo ed ultimo caso testato, i segnali di abilitazioni alti sono *en1* ed *en2*. Il sistema consente all'ingresso 1 di "parlare" e, dunque, trasmettere correttamente il payload 11 all'uscita 3 (11 in binario).

Capitolo 11

Macchine aritmetiche

Le macchine aritmetiche sono necessarie per l'implementazione degli algoritmi. Esistono due possibili approcci per la realizzazione dell'operazione che si vuole eseguire su una macchina aritmetica:

- approccio combinatorio → caratterizzato da complessità spaziale, il calcolo aritmetico necessita di un tempo di propagazione;
- approccio sequenziale → caratterizzato da complessità temporale, il calcolo utilizza più volte lo stesso componente e, per questo, bisogna definire diverse fasi di utilizzo del componente.

Per quanto riguarda, invece, la realizzazione del circuito che esegue l'operazione, gli approcci possibili sono:

- a livello porta → si realizza un circuito ad-hoc per ogni operazione;
- a livello comportamentale → comporta una perdita di controllo sull'implementazione fisica;
- a livello strutturale → integra componenti a livello porta e componenti a livello comportamentale.

I dati possono essere presentati in diversi modi:

- presentazione parallela (i bit degli operandi sono presentati in ingresso ed i risultati sono calcolati contemporaneamente, a meno di ritardi di propagazione);
- presentazione seriale pura (i bit degli operandi sono presentati sequenzialmente in ingresso ed, analogamente, i bit del risultato vengono presentati in uscita in maniera sequenziale);
- presentazione seriale a gruppi di bit (una voce è divisa in gruppi di bit, i bit omologhi vengono presentati in parallelo mentre i gruppi della stessa voce vengono presentati in serie);

- presentazione mista (uno dei due operandi è presentato in parallelo mentre l'altro in serie).

Il tipo di presentazione scelto, ovviamente, influisce sull'unità aritmetica. Una macchina combinatoria, infatti, sarà maggiormente adatta ad una presentazione parallela mentre una macchina sequenziale per gli altri tipi di presentazione. La presentazione seriale, per sua natura, permette di realizzare sistemi di complessità minore e, dunque, più economici rispetto ad una presentazione parallela. Di contro, però, una macchina caratterizzata da una presentazione di tipo seriale richiederà più colpi di clock per fornire il risultato e, dunque, risulterà più lenta di una macchina combinatoria caratterizzata da presentazione parallela in quanto quest'ultima dipenderà solo dai ritardi della rete combinatoria stessa.

Concentrandoci ora sui moltiplicatori, bisogna innanzitutto dire che avendo due stringhe da moltiplicare, il moltiplicando M viene codificato su *n* bit, il moltiplicatore Q viene codificato su *m* bit ed il loro prodotto su *n+m* bit. Distinguiamo due tipi di moltiplicatori: quelli paralleli e quelli seriali. In particolare, ci concentreremo sull'introduzione del moltiplicatore di Robertson, appartenente alla tipologia dei moltiplicatori seriali derivati da procedura manuale.

Il moltiplicatore di Robertson viene introdotto in modo da poter effettuare moltiplicazioni anche con numeri negativi in complemento a due. In tal caso, ogni cifra ha un peso (dato dalla notazione posizionale). In particolare, il bit più significativo ha un peso particolare: se il numero è negativo, il peso della cifra significativa è negativo. L'algoritmo sequenziale scandisce una cifra alla volta e, dunque, questo costituisce un problema in quanto non possiamo trattare una cifra di peso negativo come una cifra di peso positivo. Si possono presentare, allora, due situazioni: da un lato se il moltiplicando è negativo ce ne accorgiamo subito; se invece è il moltiplicatore ad essere negativo, tale informazione risulterà nota solo alla fine. In questo caso, allora, anziché effettuare somma e shift come nel classico algoritmo sequenziale, quando arriviamo all'ultima cifra effettuiamo una sottrazione al posto della somma: questo viene detto *"passo di correzione"*.

11.1 Moltiplicatore di Robertson

Progettare ed implementare in VHDL un moltiplicatore di Robertson, per effettuare il prodotto di 2 stringhe A e B da 8 bit ciascuna.

Opzionalmente, la macchina implementata può essere sintetizzata su FPGA e testata mediante l'utilizzo dei dispositivi di input/output (switch, bottoni, led, display) presenti sulla board di sviluppo in dotazione al gruppo. La modalità di utilizzo degli stessi è a completa discrezione degli studenti.

11.1.1 Soluzione

La macchina aritmetica scelta è il moltiplicatore di Robertson.
Riceve in ingresso il moltiplicando e il moltiplicatore e restituisce in uscita il risultato della moltiplicazione. Abbiamo ipotizzato che entrambi gli operandi siano rappresentati su 8 bit e, quindi, la soluzione dovrà essere rappresentata su 16 bit. Il moltiplicatore, inoltre, riceve in ingresso un segnale di *start* (per avviare la moltiplicazione) un segnale di reset ed un segnale di clock.
Per come è stato costruito il moltiplicatore di Robertson, esso restituisce il valore corretto solo quando si abbassa un segnale di uscita di *stop*. Solo in quel momento, il valore che risiede nell'uscita è valido. Per poter catturare il valore in tempo, si è pensato ad un buffer che, all'arrivo del segnale di *stop*, salvi e mostri il risultato tramite il display a 7 segmenti della scheda fino a quando non arriva un nuovo risultato. Il segnale di *start* da dare al moltiplicatore sarà gestito con un bottone che implicherà l'utilizzo di un *debouncer*.

11.1.2 Realizzazione

Come studiato a livello teorico, il moltiplicatore di Robertson è stato progettato scomponendo l'architettura in parte operativa e parte di controllo. La parte operativa è stata realizzata sfruttando un approccio strutturale, unendo le varie componenti che ci permettono di realizzare la moltiplicazione. Riportiamo di seguito lo schematico del

datapath di Robertson e, a seguire, esamineremo le varie componenti inserendo, infine, i codici relativi a quelle più significative.

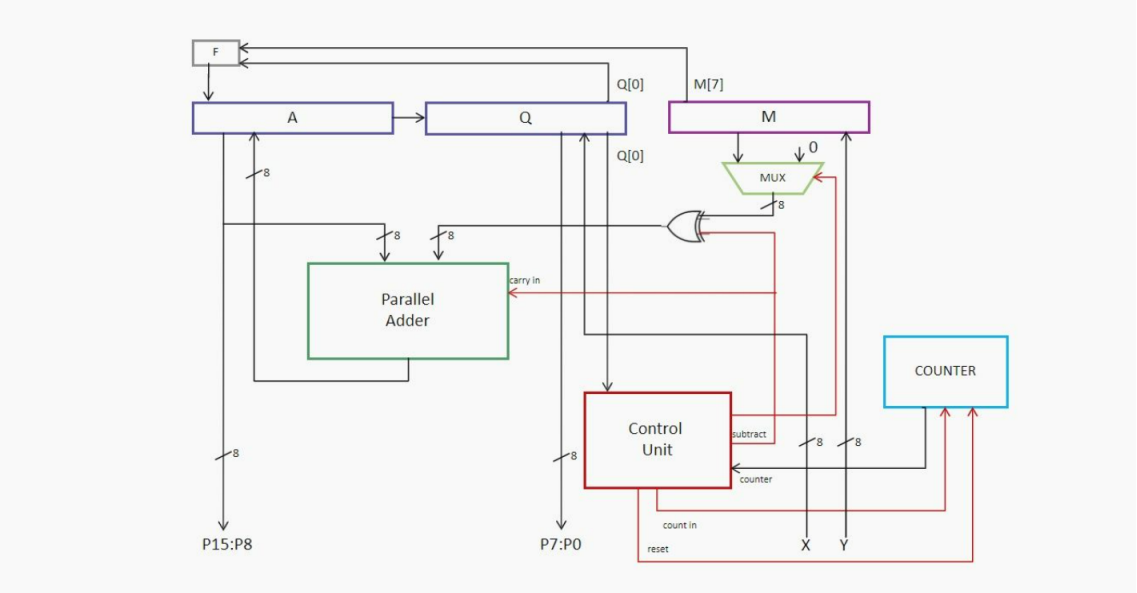


Figura 11.1 - Schematico del datapath di Robertson

La parte operativa del moltiplicatore di Robertson realizzato presenta, dunque:

- un addizionatore costituito da più ripple carry adder che è in grado di effettuare sia la somma che la sottrazione in base ad un apposito segnale *subtract* di ingresso. Inoltre, l'adder è costituito da una porta XOR bit a bit che effettua il complemento delle cifre tra il secondo operando ed il riporto entrante;
- uno shift register che inizialmente lavora a caricamento parallelo (quando riceve il segnale di *load* alto) per caricare il moltiplicatore. Successivamente, quando il segnale di *load_add* è alto, carica i primi 8 bit con l'uscita dell'addizionatore (il risultato della somma parziale). Infine, effettua un right shift se il segnale di *shift* è alto, ponendo in testa il valore F ricevuto dal flip flop D. Se, invece, arriva il segnale *fshift*, effettua lo shift finale a destra mantenendo invariato il bit più significativo;
- un registro a 8 bit, utilizzato per memorizzare il valore del moltiplicando. Viene realizzato in maniera comportamentale e, tramite un apposito *process*, pone a 0 l'uscita se il valore del reset è alto, altrimenti pone l'uscita pari all'ingresso ad ogni colpo di clock;

- un contatore modulo 8 necessario per mantenere il conteggio dei passi eseguiti dall'unità di controllo. Quando si arriva al settimo (*count = 111*) passo, infatti, l'unità di controllo viene notificata ed è pronta per effettuare l'eventuale passo di correzione e lo shift finale. Fino ad allora, l'unità di controllo procede effettuando il passo di normale addizione;
- un multiplexer per effettuare la moltiplicazione delle cifre del moltiplicatore per il moltiplicando considerando il fatto che se la cifra è 0, il risultato sarà una stringa di 0, altrimenti il risultato è il moltiplicando stesso;
- un flip flop D necessario per mantenere il segno del moltiplicando. Tale segno viene calcolato come una AND tra il bit più significativo del moltiplicando contenuto nel registro M (*M[7]*) ed il bit meno significativo del registro Q (*Q[0]*) in cui inizialmente viene inserito il moltiplicatore e viene man mano shiftato a destra.

Di seguito riportiamo l'implementazione VHDL dell'addizionatore (comprensivo della componente per il riporto) e dello shift register.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

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;

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
	complemento_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;
```

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

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;

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;
```

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY shif_register IS
	PORT
	(
		X                                                  : IN
```

```
std_logic_vector(15 DOWNTO 0);
            F                                                    : IN
std_logic;
            clock, reset, load, en, load_add, shift, f_shift : IN
std_logic;
            x_add                                                : IN
std_logic_vector(7 DOWNTO 0);
      Y                                                  : OUT
std_logic_vector(15 DOWNTO 0));
END shif_register;

ARCHITECTURE behavioural OF shif_register IS

      SIGNAL temp  : std_logic_vector(15 DOWNTO 0);
      SIGNAL temp1 : std_logic_vector(15 DOWNTO 0);
BEGIN
      SR : PROCESS (clock, reset)
      BEGIN
            IF (en = '1') THEN
                  IF (reset = '1') THEN
                        temp <= (OTHERS => '0');
                  ELSIF (clock'EVENT AND clock = '1') THEN
                        IF (load = '1') THEN --se c'è lo start carica il
moltiplicatore
                              temp <= X;
                        ELSIF (load_add = '1') THEN
                              temp(15 DOWNTO 8) <= x_add;
                        ELSIF (shift = '1') THEN
                              temp(14 DOWNTO 0) <= temp(15 DOWNTO 1);
                              temp(15)          <= F;
                        ELSIF (f_shift = '1') THEN
                              temp(14 DOWNTO 0) <= temp(15 DOWNTO 1);

                        END IF;
                  END IF;
            END IF;
      END PROCESS;
      Y <= temp;
END behavioural;
```

Riportiamo, inoltre, di seguito la top-module della parte operativa per mostrare le istanze ed i collegamenti relativi alle componenti fin qui descritte.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY unita_operativa IS
      PORT
      (
            X, Y                                                 : IN
std_logic_vector(7 DOWNTO 0);
            clock, reset, load, sub, sel                         : IN
std_logic;
```

```
            en_SR, en_R8, en_D, count_in, load_add, shift, f_shift : IN
std_logic;
            count                                                  : OUT
std_logic_vector(2 DOWNTO 0);
            P                                                      : OUT
std_logic_vector(15 DOWNTO 0)
      );
END unita_operativa;

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, en : IN std_logic;
                  B            : OUT std_logic_vector(7 DOWNTO 0)
            );
      END COMPONENT;


      COMPONENT mux_21 IS
            PORT
            (
                  x0, x1 : IN std_logic_vector(7 DOWNTO 0); --x1 è
moltiplicatore e x0 è la stringa di 0
                  s      : IN std_logic;
                  y      : OUT std_logic_vector(7 DOWNTO 0)
            );
      END COMPONENT;


      COMPONENT shif_register IS
            PORT
            (
                  X                                                : IN
std_logic_vector(15 DOWNTO 0);
                  F                                                : IN
std_logic;
                  clock, reset, load, en, load_add, shift, f_shift : IN
std_logic;
                  x_add                                            : IN
std_logic_vector(7 DOWNTO 0);
            Y                                                 : OUT
std_logic_vector(15 DOWNTO 0));
      END COMPONENT;
```

```
	COMPONENT FFD IS
		PORT
		(
			clock, reset, d, en : IN std_logic;
			y                   : OUT std_logic
		);
	END COMPONENT;

	COMPONENT cont_mod8 IS
		PORT
		(
			clock, reset : IN std_logic;
			count_in     : IN std_logic;
			count_end    : OUT std_logic;
			count        : OUT std_logic_vector(2 DOWNTO 0)
		);
	END COMPONENT;
	SIGNAL temp1         : std_logic_vector(7 DOWNTO 0); --segnale
temporaneao tra reg8 M e mux 21
	SIGNAL op1           : std_logic_vector(7 DOWNTO 0); --segnale
temporaneo di uscita dal multiplexer tra mux e adder
	SIGNAL temp2         : std_logic_vector(15 DOWNTO 0); --segnale
temporaneo per definire input all'SR
	SIGNAL temp_p        : std_logic_vector(15 DOWNTO 0); --segnale
temporaneo uscita dell'SR
	SIGNAL temp_d        : std_logic := '0';
	SIGNAL temp_F        : std_logic;
	SIGNAL tempadd       : std_logic_vector(7 DOWNTO 0); --uscita del
parallel adder
	SIGNAL riporto       : std_logic; -- riporto in uscita dell'adder
che non utilizziamo
	SIGNAL temp_countend : std_logic;
BEGIN
	M : registro8
	PORT MAP(Y, clock, reset, en_R8, temp1);
	MUX : mux_21
	PORT MAP("00000000", temp1, sel, op1);

	temp2 <= "00000000" & X;
	SR : shif_register
	PORT MAP(temp2, temp_F, clock, reset, load, en_SR, load_add, shift,
f_shift, tempadd, temp_p);
temp_d <= (temp1(7) AND temp_p(0)) OR temp_F;
D : FFD
PORT MAP(clock, reset, temp_d, en_D, temp_F);
ADD_SUB : adder_sub
PORT MAP(temp_p(15 DOWNTO 8), op1, sub, tempadd, riporto);
CONT : cont_mod8
PORT MAP(clock, reset, count_in, temp_countend, count);
P <= temp_p;

END structural;
```

Per quanto riguarda la parte di controllo, è stata realizzata tramite un'automa a stati finiti che, ad ogni passo, abilita le varie componenti della parte operativa per ottenere il risultato finale della moltiplicazione. Di seguito riportiamo l'automa relativo.

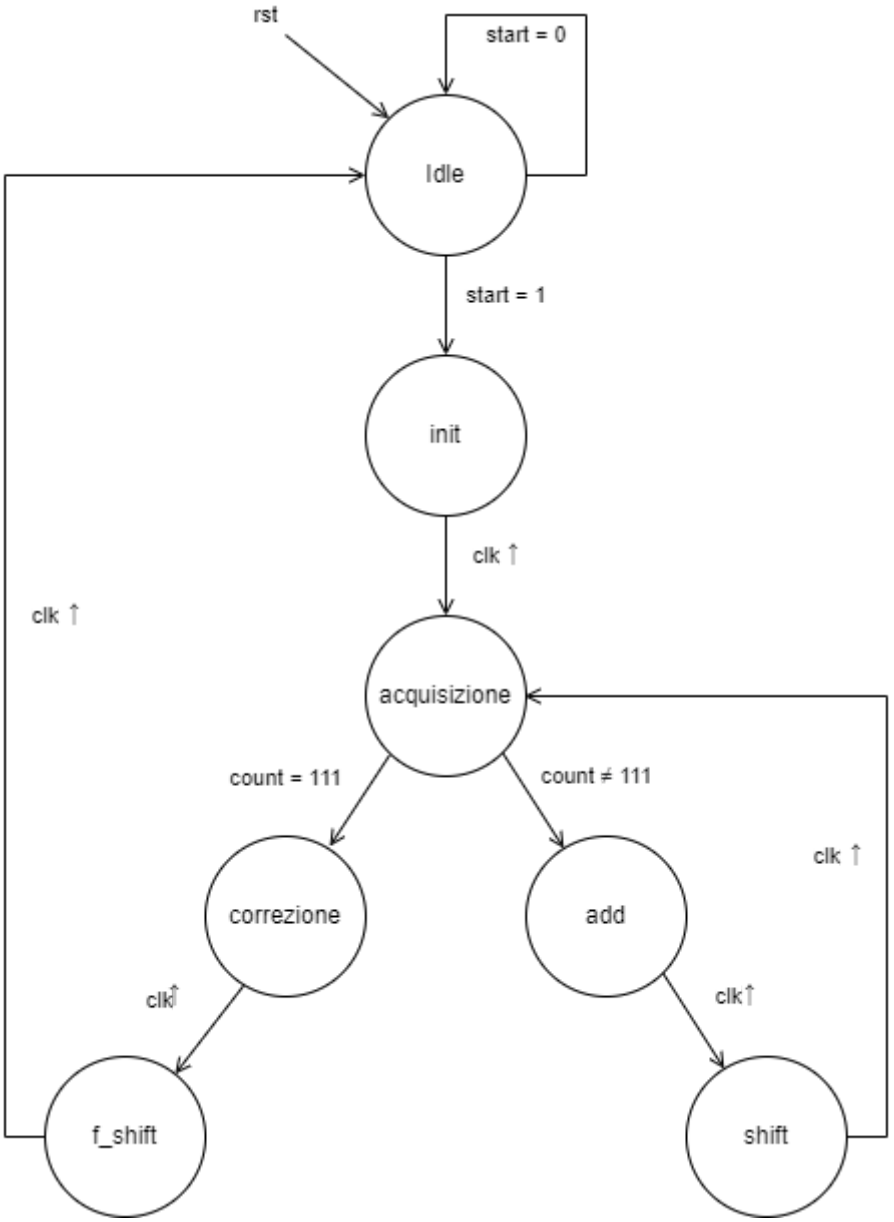


Figura 11.2 - Automa a stati finiti della parte di controllo del moltiplicatore di Robertson

Come è possibile notare dall'automa sopra riportato, l'unità di controllo gestisce le varie fasi in cui le componenti dell'unità operativa devono attivarsi una dopo l'altra per effettuare correttamente la moltiplicazione. Per ogni cifra del moltiplicatore, è necessario effettuare un'addizione ed il right shift. Arrivati all'ultimo passo, se necessario, viene effettuato il passo di correzione nel caso in cui il risultato debba

essere negativo. Se X è il moltiplicatore ed Y il moltiplicando della nostra operazione, bisognerà gestire 4 casi differenti per poter calcolare opportunamente risultato ed il relativo segno:

- se $X > 0$ e $Y > 0 \rightarrow$ non dobbiamo effettuare alcun passo di correzione in quanto le somme sono tutte positive e i bit vengono opportunamente shiftati;
- se $X > 0$ e $Y < 0 \rightarrow$ il fatto che il moltiplicando sia negativo, è subito noto. Per questo, ogni volta che tale moltiplicando viene moltiplicato per una cifra non nulla, il prodotto parziale ottenuto è sicuramente negativo e, dunque, il flip flop F verrà riempito con un bit alto per conservare tale informazione di segno fino alla fine dell'operazione;
- se $X < 0$ e $Y > 0 \rightarrow$ il segno non è noto fino all'ultima cifra. In tal caso, effettuiamo le somme e, una volta raggiunta l'ultima cifra, ci accorgeremo del peso negativo e, dunque, anziché effettuare la somma verrà effettuata la sottrazione per applicare il passo di correzione;
- se $X < 0$ e $Y < 0 \rightarrow$ i prodotti parziali sono tutti di segno negativo finché, alla fine, ci accorgiamo che in realtà anche il segno del moltiplicatore è negativo e, quindi, anziché effettuare la somma effettuiamo una sottrazione per applicare il passo di correzione.

Di seguito, mostriamo il codice VHDL realizzato per implementare l'automa a stati finiti in grado di gestire le casistiche sopra descritte.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY unita_controllo IS
	PORT
	(
		q0, clock, reset, start                                  : IN
std_logic;
		count                                                    : IN
std_logic_vector(2 DOWNTO 0);
		en_SR, en_R8, en_D, count_in, load_add, en_shift         : OUT
std_logic;
		sel, reset_out, subtract, load, stop, en_fshift, stop_cu : OUT
std_logic
	);
END unita_controllo;
```

```
ARCHITECTURE structural OF unita_controllo IS
	TYPE state IS (idle, init, acquisizione, add, shift, correzione,
f_shift);
	SIGNAL current_state, next_state : state;

BEGIN
	reg_stato : PROCESS (clock, reset)
	BEGIN
		IF (reset = '1') THEN
			current_state <= idle;
		ELSIF (clock'EVENT AND clock = '1') THEN
			current_state <= next_state;
		END IF;
	END PROCESS;

	comb : PROCESS (current_state, count, start)
	BEGIN
		CASE current_state IS

			WHEN idle => reset_out <= '1';
			en_D                   <= '1'; --serve l'abilitazione
alta altrimenti non resetta
			en_R8                  <= '1';
			en_SR                  <= '1';
			count_in               <= '0';
			subtract               <= '0';
			load_add               <= '0';
			load                   <= '0';
			sel                    <= '0';
			stop                   <= '0';
			stop_cu                <= '0';
			en_fshift              <= '0';
			en_shift               <= '0';
			IF (start = '1') THEN
				next_state <= init;
			ELSE
				next_state <= idle;
			END IF;
			--in ogni stato dobbiamo predisporre i valori dei
segnali per lo stadio successivo
			WHEN init => reset_out <= '0';
			count_in               <= '0';
			load_add               <= '0';
			en_D                   <= '1';
			en_R8                  <= '1'; --carica y in
acquisizione
			en_SR                  <= '1';
			load                   <= '1'; --abilitiamo SR per
caricare x in acquisizione
			subtract               <= '0';
			stop                   <= '0';
			sel                    <= q0;
			en_shift               <= '0';
			en_fshift              <= '0';
			stop_cu                <= '0';
			next_state             <= acquisizione;
```

```
                    --acquisisce gli operandi e predispone la add
                    WHEN acquisizione => en_SR <= '1'; --disabilitiamo Sr
cosÃ¬ non shifta
                    load                        <= '0';
                    reset_out                   <= '0';
                    en_D                        <= '1';
                    load_add                    <= '1';
                    en_R8                       <= '1';
                    count_in                    <= '1';
                    subtract                    <= '0';
                    sel                         <= q0;
                    stop                        <= '0';
                    stop_cu                     <= '0';
                    en_fshift                   <= '0';
                    en_shift                    <= '0';
                    IF (count = "111") THEN
                          subtract   <= '1';
                          next_state <= correzione;
                          --predispone la sottrazione
                    ELSE next_state <= add;
                    END IF;
                    --esegue la somma e predispone lo shift
                    WHEN add => count_in <= '0';
                    reset_out            <= '0';
                    sel                  <= q0;
                    subtract             <= '0';
                    load_add             <= '0'; --carico il valore nello
shift
                    en_SR                <= '1';
                    en_shift             <= '1';
                    load                 <= '0';
                    en_D                 <= '0';
                    en_R8                <= '1';
                    stop_cu              <= '0';
                    stop                 <= '0';
                    en_fshift            <= '0';
                    next_state           <= shift;
                    --esegue lo shift e predispone per acquisizione
                    WHEN shift => load <= '0';
                    load_add           <= '0';
                    en_SR              <= '1'; --sr abilitato senza
caricare x
                    reset_out          <= '0';
                    sel                <= q0;
                    en_shift           <= '0';
                    count_in           <= '0';
                    subtract           <= '0';
                    en_D               <= '0';
                    en_R8              <= '1';
                    stop_cu            <= '0';
                    en_fshift          <= '0';
                    stop               <= '0';
                    next_state         <= acquisizione;
```

```
                    --in correzione fa la sottrazione e predispone per
l'ultimo shift
                    WHEN correzione => en_SR <= '1';
                    subtract                 <= '0';
                    load                     <= '0';
                    reset_out                <= '0';
                    load_add                 <= '0';
                    en_D                     <= '0'; --a(7)=a(7)
                    en_R8                    <= '1';
                    count_in                 <= '0';
                    en_shift                 <= '0';
                    en_fshift                <= '1';
                    sel                      <= q0;
                    stop                     <= '1';
                    stop_cu                  <= '0';
                    next_state               <= f_shift;

                    WHEN f_shift => subtract <= '0';
                    en_SR                    <= '0';
                    load                     <= '0';
                    reset_out                <= '0';
                    load_add                 <= '0';

                    en_D                     <= '0';
                    en_R8                    <= '1';
                    en_shift                 <= '0';
                    en_fshift                <= '0';
                    count_in                 <= '0';
                    sel                      <= q0;
                    stop                     <= '0';
                    stop_cu                  <= '1';
                    next_state               <= idle;
            END CASE;
      END PROCESS;
END structural;
```

Conclusa l'analisi strettamente relativa al moltiplicatore, passiamo adesso a descrivere alcune componenti aggiuntive utilizzate per realizzare il sistema complessivo ed impiegate per la successiva fase di sintesi.

Riportiamo, quindi, di seguito lo schematico relativo alla top-module del sistema.

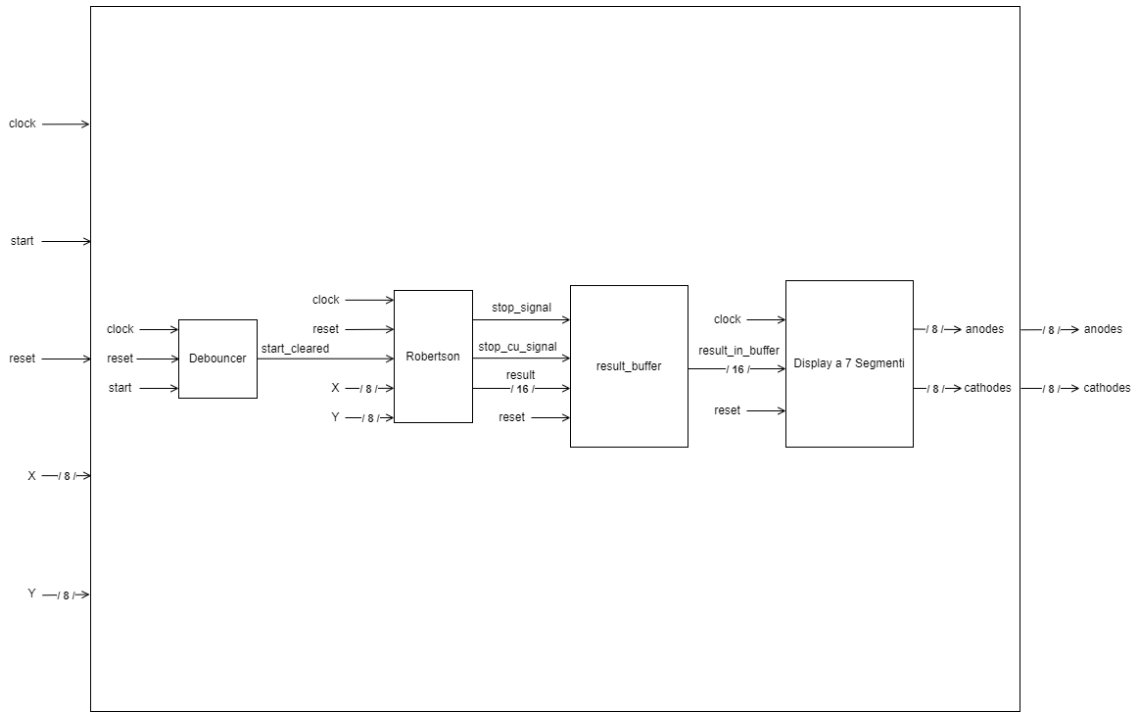


Figura 11.3 - Schematico della top-module

Come è possibile notare dallo schematico, abbiamo realizzato una prima componente debouncer per poter opportunamente gestire il comportamento del bottone che mapperemo in fase di sintesi per il segnale di *start* che avvia il calcolo. Il segnale di start ripulito (*start_cleared*) viene così mandato in ingresso al moltiplicatore stesso, il quale produrrà un'uscita data dal prodotto di moltiplicatore e moltiplicando forniti tramite switch della scheda. Tale uscita sarà fornita in ingresso ad un buffer apposito che servirà poi per mostrare tale risultato sulla scheda tramite il display a 7 segmenti.

Di seguito riportiamo l'implementazione VHDL della componente *result_buffer*, evitando di riportare nuovamente i codici relativi al debouncer ed al display a 7 segmenti già ampiamente analizzati nei capitoli appositi.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY result_buffer IS
      PORT
      (
            stop, stop_cu, reset : IN std_logic;
            result               : IN std_logic_vector(15 DOWNTO 0);
            value                : OUT std_logic_vector(15 DOWNTO 0)
      );
END result_buffer;
```

```
ARCHITECTURE Behavioral OF result_buffer IS

BEGIN
	PROCESS (stop, stop_cu, result, reset)
	BEGIN
		IF (stop = '1' OR reset = '1') THEN
			value <= (OTHERS => '0');
		ELSIF (stop_cu = '1') THEN
			value <= result;
		END IF;
	END PROCESS;

END Behavioral;
```

Notiamo che ci serviamo dei segnali *stop* e *stop_cu* - entrambi uscite della componente moltiplicatore - per gestire rispettivamente l'operazione di refresh del buffer (azzerando il valore precedentemente memorizzato) e l'operazione di caricamento del dato nel registro.

Di seguito riportiamo, infine, l'implementazione della top-module del sistema complessivo.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY TopModule IS
	PORT
	(
		X, Y                 : IN std_logic_vector(7 DOWNTO 0);
		clock, reset, start : IN std_logic;
		ANODES               : OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
		CATHODES             : OUT STD_LOGIC_VECTOR (7 DOWNTO 0)
	);
END TopModule;

ARCHITECTURE Structural OF TopModule IS

	COMPONENT ButtonDebouncer IS
		GENERIC
		(
			CLK_period     : INTEGER := 10; -- periodo del clock
della board 10 nanosecondi
			btn_noise_time : INTEGER := 325000000 --intervallo di
tempo in cui si ha l'oscillazione del bottone
			--assumo che duri 6.5ms=6500microsec=6500000ns
		);
		PORT
		(
			RST          : IN STD_LOGIC;
			CLK          : IN STD_LOGIC;
			BTN          : IN STD_LOGIC;
```

```
                    CLEARED_BTN : OUT STD_LOGIC
              );
       END COMPONENT;

       COMPONENT Robertson IS
              PORT
              (
                    clock, reset, start : IN std_logic;
                    X, Y                : IN std_logic_vector(7 DOWNTO 0);
                    stop                : OUT std_logic;
                    P                   : OUT std_logic_vector(15 DOWNTO
0);
                    stop_cu             : OUT std_logic
              );
       END COMPONENT;

       COMPONENT result_buffer IS
              PORT
              (
                    stop, stop_cu, reset : IN std_logic;
                    result               : IN std_logic_vector(15 DOWNTO
0);
                    value                : OUT std_logic_vector(15 DOWNTO
0)
              );
       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 (15 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;

       SIGNAL start_cleared    : std_logic;
       SIGNAL stop_signal      : std_logic;
       SIGNAL result           : std_logic_vector(15 DOWNTO 0);
       SIGNAL stop_cu_signal   : std_logic;
       SIGNAL result_in_buffer : std_logic_vector(15 DOWNTO 0);

BEGIN
       start_button : ButtonDebouncer
              GENERIC MAP
```

```
            (
            CLK_period     => 10,
            btn_noise_time => 325000000
            )
            PORT MAP
            (
                  RST         => reset,
                  CLK         => clock,
                  BTN         => start,
                  CLEARED_BTN => start_cleared
            );

                  multiplier : Robertson
                  PORT MAP
                  (
                        clock   => clock,
                        reset   => reset,
                        start   => start_cleared,
                        X       => X,
                        Y       => Y,
                        stop    => stop_signal,
                        P       => result,
                        stop_cu => stop_cu_signal
      );


      res_buffer : result_buffer
      PORT MAP
      (
            stop    => stop_signal,
            stop_cu => stop_cu_signal,
            reset   => reset,
            result  => result,
            value   => result_in_buffer
      );

      display : display_seven_segments
            GENERIC MAP
            (
            CLKIN_freq  => 100000000,
            CLKOUT_freq => 500
            )
            PORT MAP
            (
                  CLK      => clock,
                  RST      => reset,
                  VALUE    => result_in_buffer,
                  ENABLE   => "00111111",
                  DOTS     => "00000000",
                  ANODES   => anodes,
                  CATHODES => cathodes
            );

END Structural;
```

11.1.3 Simulazione

Il primo componente che abbiamo voluto testare è proprio il moltiplicatore stesso. Abbiamo, dunque, realizzato il seguente testbench per poterne verificare il funzionamento in diversi casi di test, riportati nei commenti del codice stesso.

```
stimulus: process
  begin
  wait for 100ns; -- Caso X > 0 e Y > 0
  X <= "00110100"; --52
  Y <= "00000011"; --3
  wait for 10ns;
  start <= '1';
  wait for 20ns;
  start <= '0';
  wait for 250ns; -- Caso X > 0 e Y < 0
  X <= "00110100"; -- 52
  Y <= "11111101"; -- -3
  wait for 10ns;
  start <= '1';
  wait for 20ns;
  start <= '0';
  wait for 250ns; -- Caso X < 0 e Y > 0
  X <= "11001100"; -- -52
  Y <= "00000011"; -- 3
  wait for 10ns;
  start <= '1';
  wait for 20ns;
  start <= '0';
  wait for 250ns; -- Caso X < 0 e Y < 0
  X <= "11001100"; -- -52
  Y <= "11111101"; -- -3
  wait for 10ns;
  start <= '1';
  wait for 20ns;
  start <= '0';
  wait for 250ns;
  wait for 500ns;
  stop_the_clock <= true;
  wait;
  end process;

  clocking: process
  begin
    while not stop_the_clock loop
      clock <= '0', '1' after clock_period / 2;
      wait for clock_period;
    end loop;
    wait;
  end process;
```

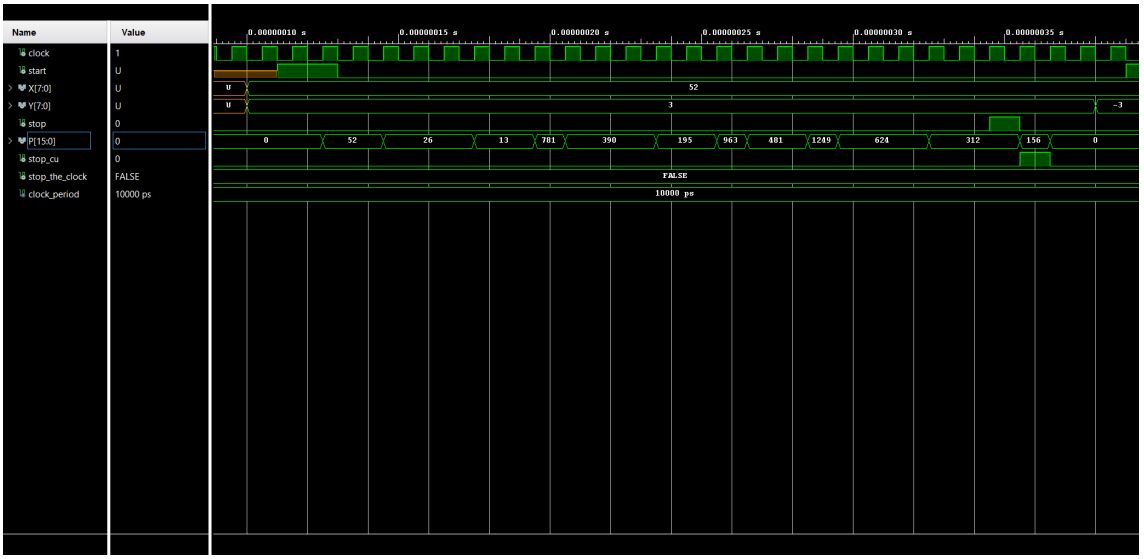


Figura 11.4 - Grafico della simulazione con X>0 e Y>0

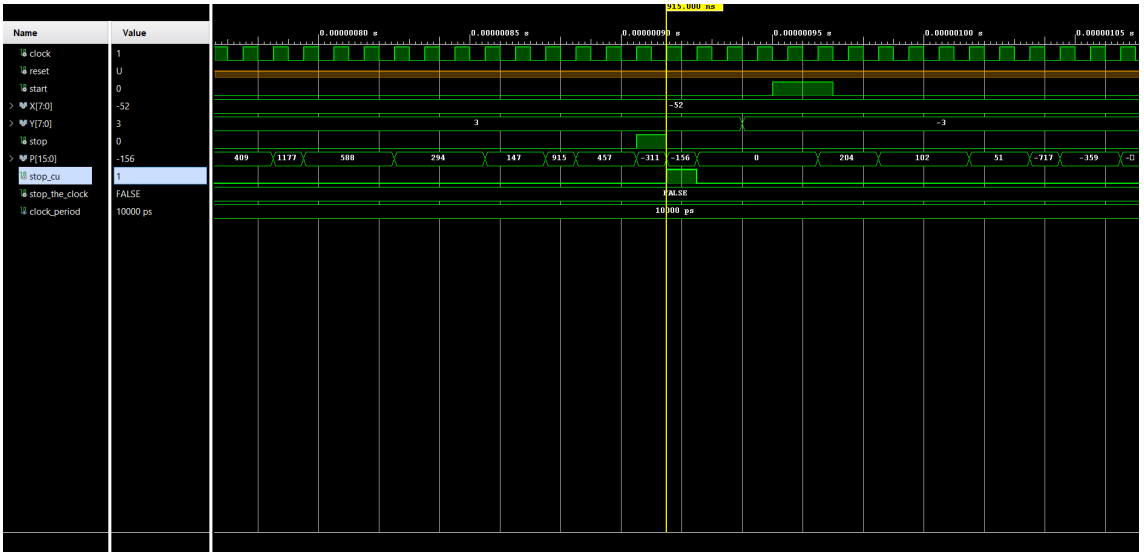


Figura 11.4 - Grafico della simulazione con X<0 e Y>0

Come si evidenzia dai grafici appena riportati, il moltiplicatore esegue correttamente le operazioni ed infatti è possibile notare il risultato di tali moltiplicazioni nel momento in cui il bit di *stop_cu* diventa alto.

11.1.4 Sintesi

Prima di riportare il file delle constraints realizzato, mostriamo il codice relativo alla gestione dei catodi (*cathodes_manager*) al fine di evidenziare alcune scelte progettuali effettuate per la corretta visualizzazione del risultato sul display a 7 segmenti.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY cathodes_manager IS
	PORT
	(
		counter  : IN STD_LOGIC_VECTOR (2 DOWNTO 0);
		value    : IN STD_LOGIC_VECTOR (15 DOWNTO 0); --dato da
mostrare sugli 8 display
		dots     : IN STD_LOGIC_VECTOR (7 DOWNTO 0); --configurazione
punti da accendere
	cathodes : OUT STD_LOGIC_VECTOR (7 DOWNTO 0)); --sono i 7 catodi più
il punto
END cathodes_manager;

ARCHITECTURE Behavioral OF cathodes_manager IS

	CONSTANT zero             : std_logic_vector(6 DOWNTO 0) :=
"1000000";
	CONSTANT one              : std_logic_vector(6 DOWNTO 0) :=
"1111001";
	CONSTANT two              : std_logic_vector(6 DOWNTO 0) :=
"0100100";
	CONSTANT three            : std_logic_vector(6 DOWNTO 0) :=
"0110000";
	CONSTANT four             : std_logic_vector(6 DOWNTO 0) :=
"0011001";
	CONSTANT five             : std_logic_vector(6 DOWNTO 0) :=
"0010010";
	CONSTANT six              : std_logic_vector(6 DOWNTO 0) :=
"0000010";
	CONSTANT seven            : std_logic_vector(6 DOWNTO 0) :=
"1111000";
	CONSTANT eight            : std_logic_vector(6 DOWNTO 0) :=
"0000000";
	CONSTANT nine             : std_logic_vector(6 DOWNTO 0) :=
"0010000";
	CONSTANT minus            : std_logic_vector(6 DOWNTO 0) :=
"0111111";

	SIGNAL cathodes_for_digit : std_logic_Vector(6 DOWNTO 0) := (OTHERS
=> '0');
	SIGNAL nibble             : INTEGER := 0;
	SIGNAL dot                : std_logic := '0'; --stabilisce se il
punto relativo alla cifra visualizzata deve essere acceso o spento
	--nota: dot=1 significa che deve essere acceso, ma il segnale deve
essere negato per andare sui catodi
```

```
	SIGNAL valueFixed : std_logic_vector(15 DOWNTO 0);
	SIGNAL i0         : INTEGER;
	SIGNAL i1         : INTEGER;
	SIGNAL i2         : INTEGER;
	SIGNAL i3         : INTEGER;
	SIGNAL i4         : INTEGER;
BEGIN
	-- questo processo multiplexa le cifre da mostrare
	digit_switching : PROCESS (counter)

	BEGIN
		CASE counter IS
			WHEN "000" =>
				nibble <= i0;
				dot    <= dots(0);
			WHEN "001" =>
				nibble <= i1;
				dot    <= dots(1);
			WHEN "010" =>
				nibble <= i2;
				dot    <= dots(2);
			WHEN "011" =>
				nibble <= i3;
				dot    <= dots(3);
			WHEN "100" =>
				nibble <= i4;
				dot    <= dots(4);
			WHEN "101" =>
				IF (value(15) = '1') THEN
					nibble <= 10;
				ELSE
					nibble <= 0;
				END IF;
				dot <= dots(5);
			WHEN OTHERS =>
				nibble <= 0;
				dot    <= '0';
		END CASE;
	END PROCESS;

	seven_segment_decoder_process : PROCESS (nibble)
	BEGIN
		CASE nibble IS
			WHEN 0 => cathodes_for_digit      <= zero;
			WHEN 1 => cathodes_for_digit      <= one;
			WHEN 2 => cathodes_for_digit      <= two;
			WHEN 3 => cathodes_for_digit      <= three;
			WHEN 4 => cathodes_for_digit      <= four;
			WHEN 5 => cathodes_for_digit      <= five;
			WHEN 6 => cathodes_for_digit      <= six;
			WHEN 7 => cathodes_for_digit      <= seven;
			WHEN 8 => cathodes_for_digit      <= eight;
			WHEN 9 => cathodes_for_digit      <= nine;
			WHEN 10 => cathodes_for_digit     <= minus;
			WHEN OTHERS => cathodes_for_digit <= (OTHERS => '0');
		END CASE;
```

```
        END PROCESS seven_segment_decoder_process;

        cathodes   <= (NOT dot) & cathodes_for_digit;
        valueFixed <= std_logic_vector(signed(value)) WHEN (value(15) = '0')
ELSE
                      std_logic_vector(NOT signed(value) + 1);
        i0 <= (to_integer(signed(valueFixed))) MOD 10;
        i1 <= (to_integer(signed(valueFixed))/10) MOD 10;
        i2 <= (to_integer(signed(valueFixed))/100) MOD 10;
        i3 <= (to_integer(signed(valueFixed))/1000) MOD 10;
        i4 <= (to_integer(signed(valueFixed))/10000) MOD 10;

END Behavioral;
```

Notiamo che effettuiamo una serie di operazioni di modulo e di divisione per multipli di 10 per ottenere i singoli digit della cifra da mostrare. Tali digit vengono visualizzati singolarmente sui vari LED del display a 7 segmenti. Inoltre, affinché il segno del risultato possa essere espresso correttamente, poniamo il *signal valueFixed* pari al valore di *value* stesso se il bit più significativo di quest'ultimo è basso. In caso contrario, invertiamo tale valore e sommiamo 1, replicando il procedimento del complemento a 2.

Infine, riportiamo di seguito il file delle constraints in modo da mostrare il mapping effettuato per portare il sistema su board.

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


##Switches
set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 } [get_ports {
X[0] }]; #IO_L24N_T3_RS0_15 Sch=sw[0]
set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 } [get_ports {
X[1] }]; #IO_L3N_T0_DQS_EMCCLK_14 Sch=sw[1]
set_property -dict { PACKAGE_PIN M13   IOSTANDARD LVCMOS33 } [get_ports {
X[2] }]; #IO_L6N_T0_D08_VREF_14 Sch=sw[2]
set_property -dict { PACKAGE_PIN R15   IOSTANDARD LVCMOS33 } [get_ports {
X[3] }]; #IO_L13N_T2_MRCC_14 Sch=sw[3]
set_property -dict { PACKAGE_PIN R17   IOSTANDARD LVCMOS33 } [get_ports {
X[4] }]; #IO_L12N_T1_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 LVCMOS18 } [get_ports {
Y[0] }]; #IO_L24N_T3_34 Sch=sw[8]
set_property -dict { PACKAGE_PIN U8    IOSTANDARD LVCMOS18 } [get_ports {
Y[1] }]; #IO_25_34 Sch=sw[9]
set_property -dict { PACKAGE_PIN R16   IOSTANDARD LVCMOS33 } [get_ports {
Y[2] }]; #IO_L15P_T2_DQS_RDWR_B_14 Sch=sw[10]
set_property -dict { PACKAGE_PIN T13   IOSTANDARD LVCMOS33 } [get_ports {
Y[3] }]; #IO_L23P_T3_A03_D19_14 Sch=sw[11]
set_property -dict { PACKAGE_PIN H6    IOSTANDARD LVCMOS33 } [get_ports {
Y[4] }]; #IO_L24P_T3_35 Sch=sw[12]
set_property -dict { PACKAGE_PIN U12   IOSTANDARD LVCMOS33 } [get_ports {
Y[5] }]; #IO_L20P_T3_A08_D24_14 Sch=sw[13]
set_property -dict { PACKAGE_PIN U11   IOSTANDARD LVCMOS33 } [get_ports {
Y[6] }]; #IO_L19N_T3_A09_D25_VREF_14 Sch=sw[14]
set_property -dict { PACKAGE_PIN V10   IOSTANDARD LVCMOS33 } [get_ports {
Y[7] }]; #IO_L21P_T3_DQS_14 Sch=sw[15]

##7 segment display
set_property -dict { PACKAGE_PIN T10   IOSTANDARD LVCMOS33 } [get_ports {
CATHODES[0] }]; #IO_L24N_T3_A00_D16_14 Sch=ca
set_property -dict { PACKAGE_PIN R10   IOSTANDARD LVCMOS33 } [get_ports {
CATHODES[1] }]; #IO_25_14 Sch=cb
set_property -dict { PACKAGE_PIN K16   IOSTANDARD LVCMOS33 } [get_ports {
CATHODES[2] }]; #IO_25_15 Sch=cc
set_property -dict { PACKAGE_PIN K13   IOSTANDARD LVCMOS33 } [get_ports {
CATHODES[3] }]; #IO_L17P_T2_A26_15 Sch=cd
set_property -dict { PACKAGE_PIN P15   IOSTANDARD LVCMOS33 } [get_ports {
CATHODES[4] }]; #IO_L13P_T2_MRCC_14 Sch=ce
set_property -dict { PACKAGE_PIN T11   IOSTANDARD LVCMOS33 } [get_ports {
CATHODES[5] }]; #IO_L19P_T3_A10_D26_14 Sch=cf
set_property -dict { PACKAGE_PIN L18   IOSTANDARD LVCMOS33 } [get_ports {
CATHODES[6] }]; #IO_L4P_T0_D04_14 Sch=cg
set_property -dict { PACKAGE_PIN H15   IOSTANDARD LVCMOS33 } [get_ports {
CATHODES[7] }]; #IO_L19N_T3_A21_VREF_15 Sch=dp
set_property -dict { PACKAGE_PIN J17   IOSTANDARD LVCMOS33 } [get_ports {
ANODES[0] }]; #IO_L23P_T3_FOE_B_15 Sch=an[0]
set_property -dict { PACKAGE_PIN J18   IOSTANDARD LVCMOS33 } [get_ports {
ANODES[1] }]; #IO_L23N_T3_FWE_B_15 Sch=an[1]
set_property -dict { PACKAGE_PIN T9    IOSTANDARD LVCMOS33 } [get_ports {
ANODES[2] }]; #IO_L24P_T3_A01_D17_14 Sch=an[2]
set_property -dict { PACKAGE_PIN J14   IOSTANDARD LVCMOS33 } [get_ports {
ANODES[3] }]; #IO_L19P_T3_A22_15 Sch=an[3]
set_property -dict { PACKAGE_PIN P14   IOSTANDARD LVCMOS33 } [get_ports {
ANODES[4] }]; #IO_L8N_T1_D12_14 Sch=an[4]
set_property -dict { PACKAGE_PIN T14   IOSTANDARD LVCMOS33 } [get_ports {
ANODES[5] }]; #IO_L14P_T2_SRCC_14 Sch=an[5]
set_property -dict { PACKAGE_PIN K2    IOSTANDARD LVCMOS33 } [get_ports {
ANODES[6] }]; #IO_L23P_T3_35 Sch=an[6]
set_property -dict { PACKAGE_PIN U13   IOSTANDARD LVCMOS33 } [get_ports {
ANODES[7] }]; #IO_L23N_T3_A02_D18_14 Sch=an[7]

##Buttons
set_property -dict { PACKAGE_PIN N17   IOSTANDARD LVCMOS33 } [get_ports {
start }]; #IO_L9P_T1_DQS_14 Sch=btnc
set_property -dict { PACKAGE_PIN M18   IOSTANDARD LVCMOS33 } [get_ports {
reset }]; #IO_L4N_T0_D05_14 Sch=btnu
```

Capitolo 12

Gestione delle collisioni in Omega Network

Come abbiamo già introdotto nel capitolo 10 di questo elaborato, abbiamo scelto di realizzare un sistema per la gestione delle collisioni in una Omega Network come esercizio “libero”. Procediamo, innanzitutto, con una breve introduzione teorica.

Come già spiegato nell’apposito capitolo, una Omega Network può essere utilizzata per inviare più messaggi contemporaneamente e, proprio per questo, bisognerà considerare l’eventualità in cui si presenteranno possibili collisioni. Una collisione si verifica quando un nodo riceve contemporaneamente due messaggi diversi. Tali messaggi entreranno così in conflitto.

Esistono diverse tecniche che consentono di gestire le collisioni:

- Non permettere che si verifichino collisioni → Per fare ciò è necessario costruire un'altra macchina che recupera le richieste da parte di tutti i nodi e decide attraverso un criterio di priorità qual è il messaggio a maggior priorità che sarà trasmesso. Tale macchina può essere implementata come una macchina di controllo combinatoria che decide in relazione ai segnali quale va avanti. Considerando, invece, una logica round-robin la macchina verrebbe implementata come una macchina sequenziale poiché deve necessariamente mantenere memoria dell'ultima linea che ha avuto priorità e, quindi, cambiare successivamente la massima priorità per attribuirla ad un’altra linea;
- Realizzare un sistema a perdita → Una delle implementazioni più semplici. In tale sistema, se si verifica una collisione in un nodo, un messaggio prosegue e l’altro viene perso. Se il protocollo è uno di quelli che numera i messaggi, allora il messaggio perso sarà successivamente ritrasmesso. In caso contrario, il messaggio viene semplicemente perso ma il problema dovuto alla collisione viene comunque risolto;

- Il nodo ha un sistema di memorizzazione → Utilizzando la tecnica dello *"store and forward"*, un messaggio prosegue e l'altro viene memorizzato in un'apposita componente di memoria aggiuntiva associata al nodo stesso. I messaggi devono ovviamente avere lunghezza fissa, poiché per memorizzarli in modo efficiente bisogna avere un buffer;
- Modifica dell'instradamento → Tale tecnica non è realizzabile in una Omega Network poiché in tale rete si hanno proprietà topologiche che legano i fili all'indirizzo. Se cambiassimo la strada, non si avrebbero più garanzie di raggiungere quella determinata destinazione.

La tecnica da noi scelta ed implementata è stata quella di "*store and forward*". Come spiegheremo approfonditamente nei successivi paragrafi, infatti, andremo a dotare ogni switch di un apposito buffer di memoria per la gestione delle collisioni nel sistema.

12.1 Gestione delle collisioni tramite *store and forward*

Rimuovendo l'ipotesi di lavorare secondo uno schema a priorità fra i nodi come nell'esercizio 10 e considerando una rete di 8 nodi, lo switch deve gestire eventuali conflitti generati da collisioni secondo un meccanismo a scelta.

12.1.1 Soluzione

Siamo partiti dallo studio della Omega Network e dall'utilizzo di una entità centralizzata che non faccia accadere collisioni, proprio come nell'esercizio 10.

L'approccio "*store and forward*" non prevede un'entità centralizzata che funga da semaforo per le comunicazioni bensì aggiunge una logica decisionale ai singoli switch tale da renderli in grado di gestire una collisione, entro certi limiti. In generale, lo "*store and forward*" è una scelta più scalabile in quanto, a differenza di un'entità centralizzata che deve gestire tutte le attività e fare da arbitro all'intera rete di switch, in tale approccio c'è una logica decisionale distribuita.

Considerando la presenza di una logica distribuita, ogni singolo switch deve ricevere anche la destinazione per sapere dove smistare il dato. I singoli switch saranno, inoltre, dotati di un clock poiché non sono più combinatori.

Nonostante i singoli switch abbiano un comportamento diverso rispetto all'esercizio 10, la topologia (e quindi i collegamenti tra di loro) resta la medesima, rispettando il *"perfect shuffling"*.

All'interno di ogni switch è presente una coda circolare che, ad ogni colpo di clock, manda in uscita un dato. Il dato in uscita dalla coda sarà poi smistato tra le due uscite dello switch tramite un demultiplexer. In base all'i-esimo bit dell'indirizzo di destinazione, il demultiplexer decide se instradare nella prima o nella seconda uscita risolvendo così il problema sorgente-destinazione.

Più in particolare, abbiamo pensato a come un singolo switch possa sapere dove inviare il dato (nell'uscita superiore o inferiore). Nel caso della gestione con entità centralizzata, infatti, è possibile passare ai singoli switch solo il bit per la selezione nel demultiplexer per l'instradamento dei dati. Nel nostro caso, invece, lo switch necessiterà di dati aggiuntivi (i bit della destinazione) per selezionare la destinazione corretta. Si è presentato, così, un altro problema: far conoscere al singolo switch la sua i-esima posizione (stadio) nella topologia. Per conoscere lo stadio ***i***, sono stati pensati vari approcci:

- ***approccio centralizzato*** → un dispositivo esterno alla rete assegna una posizione fissa ad ogni nodo. Questo ci porterebbe ad introdurre nuovamente un'unità centralizzata, in contraddizione con quanto spiegato relativamente alla soluzione *"store and forward"*;
- ***approccio incrementale*** → lo switch riceve, oltre ai dati ed alla destinazione, anche dei bit "posizionali" (chiamati *level_bit*) che rappresentano la sua posizione (stadio) nella rete di switch. Man mano che il dato viaggia all'interno della rete, attraversandone i vari stadi, il valore del *level_bit* viene incrementato di 1 per esplicitare lo stadio raggiunto. Questo vuol dire che, se gli stadi, ad esempio, sono 3 (come nel nostro caso) necessitiamo di 2 bit per rappresentare tale informazione. Scegliere quest'approccio vuol dire rendere il singolo pacchetto più grande e, di conseguenza, è presente un overhead dei dati di $log_2(stadi)$. Tale overhead impatta anche sui costi in quanto i registri delle code dovranno necessariamente presentare una maggiore capienza;

- ***approccio "lossy"*** → il singolo switch riceve solo <*destinazione_0, payload*>. Ogni switch restituisce in uscita <*destinazione_1, payload*> dove *destinazione_1* è dato dal valore di *destinazione_0* privato del bit più significativo, permettendo ad ogni stadio di trovare in corrispondenza del bit più significativo sempre l'informazione per selezionare l'uscita corretta del demultiplexer. Abbiamo preferito non adottare tale approccio in quanto la struttura sarebbe stata non facilmente iterabile perché gli switch di stadi diversi avrebbero dovuto gestire dati di dimensione differente;
- ***approccio cablato*** → salviamo in una ROM, pre-costituita, la fila dello switch.

Si è scelto di utilizzare l'approccio incrementale dove, quindi, il singolo pacchetto è formato dalla tripletta:

$$< level_bit,\ dst,\ payload >$$

dove:

- *level_bit* è codificato su 2 bit;
- *dst* è codificato su 3 bit;
- *payload* è codificato su 2 bit (implementato tramite Generics, dunque modificabile).

Ogni stadio ha una logica tale che il pacchetto in uscita sarà identico al pacchetto in ingresso se non per un incremento di 1 del valore di *level_bit*.

Per quanto riguarda invece l'acquisizione dei valori da parte degli switch, a differenza dell'approccio precedente, necessitiamo di un buffer per ognuna delle due entrate perché il dato non viene inoltrato subito verso l'uscita: dobbiamo mantenerlo in qualche modo.

Essendo due le entrate di ogni switch, necessitiamo quindi di due buffer.

La Control Unit lavora sul fronte di discesa del clock e gestisce l'ingresso dei buffer alla coda. Se la coda ha almeno un posto libero, si inserisce il valore. Ovviamente, se un buffer è in attesa di essere letto e arriva un nuovo segnale in ingresso prima del fronte di discesa del clock, il valore presente nel buffer viene sovrascritto. Questo problema

potrebbe essere dovuto ad una velocità del mittente maggiore di quella del destinatario che, nel nostro caso, non è contemplata visto che il singolo switch ogni colpo di clock rilascia un pacchetto e tutti gli switch sono animati dallo stesso clock.
Può succedere che in un colpo di clock possono arrivare anche due dati da aggiungere alla coda (il primo dato dalla prima entrata e il secondo dato dalla seconda entrata).
Inevitabilmente, all'aumentare della congestione della rete, può capitare che il singolo switch si saturi. In particolare, se ad ogni colpo di clock ad uno switch arrivano due pacchetti per tanti colpi di clock di fila, la coda si satura. C'è bisogno quindi anche di una scelta che ci permetta di gestire la situazione dove uno switch abbia riempito del tutto la sua memoria. Questo viene effettuato attraverso l'unità di controllo che gestisce l'inserimento dei valori dei due buffer all'interno della coda. L'unità di controllo riceve, quando la lista risulta piena, un flag che avvisa l'unità di non mandare alla coda più dati, essendo essa piena. In realtà, quello che succede è che, visto che ad ogni colpo di clock la coda si svuota di una posizione, lavorando sul fronte di salita del clock e dando priorità sempre all'operazione di *dequeue* prima di quella di *enqueue*, l'unico caso (worst-case) in cui ci può essere saturazione di memoria è quando lo switch riceve tante volte due messaggi di fila. Se, ad esempio, la coda può contenere al massimo quattro pacchetti di dati ed è inizialmente vuota, al quarto colpo di clock si verificherà l'evento di "coda piena" insieme ad una collisione e siamo costretti inevitabilmente a perdere uno dei messaggi entranti. Ogni volta dunque lo switch sarà costretto, quando la coda sarà piena, a selezionare quale pacchetto inserire nella coda e quale, invece, perdere. Per convenzione si è assegnato uno schema di priorità fisso dove la prima entrata ha priorità più alta rispetto alla seconda. I *level_bit* con configurazione iniziale (00) saranno concatenati alla destinazione e al payload prima di entrare nel primo switch. Infine, per riallinearsi con l'esercizio 10, quando il messaggio arriva effettivamente a destinazione, si riduce il pacchetto estrapolandone solo il *payload*.
Inizialmente, si era ipotizzato di far lavorare l'intero sistema sul fronte di salita del clock e quindi, l'intero sistema risultava lento. Per questo, abbiamo optato per far lavorare la parte operativa sul fronte di salita e la parte di controllo sul fronte di discesa.

12.1.2 Realizzazione

Lo switch è stato costruito con un approccio strutturale.

Al suo interno, come specificato nel paragrafo precedente, troviamo:

- *buffer0* che cattura in parallelo il messaggio proveniente dallo stadio precedente diretto all'ingresso 0;
- *buffer1* che cattura in parallelo il messaggio proveniente dallo stadio precedente diretto all'ingresso 1;
- *queue_register* è la coda dei messaggi;
- *demux* è il demultiplexer utile per smistare tra le due uscite il messaggio;
- *CU* controlla la parte operativa (realizzata come un automa a stati finiti).

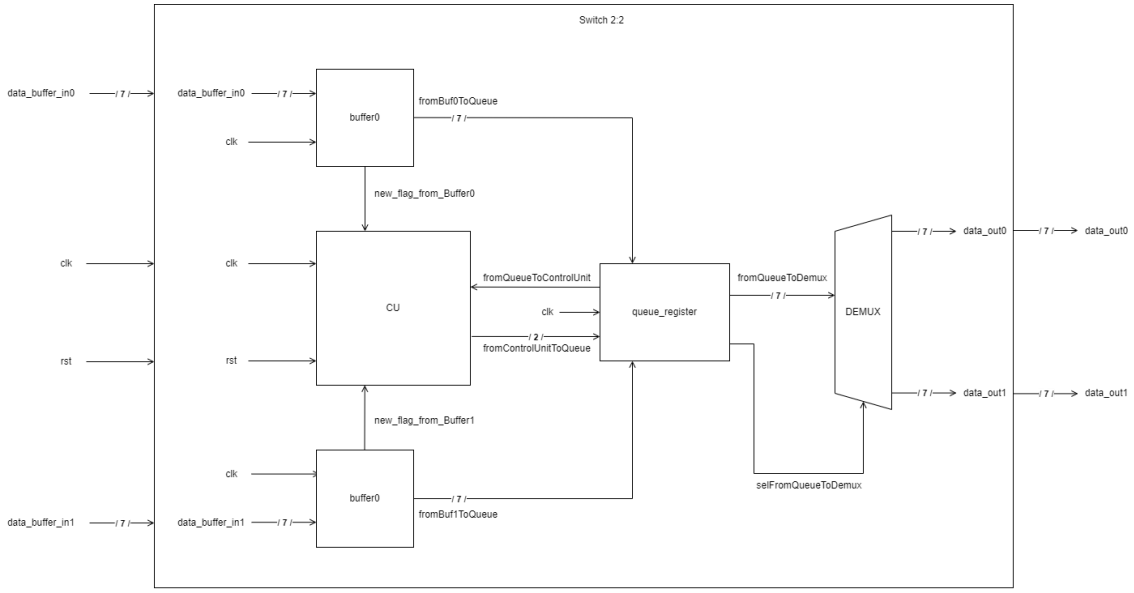


Figura 12.1 - Schematico del componente Switch 2:2

Il buffer serve per mantenere momentaneamente il dato. Il dato viene salvato solo se diverso dal valore precedente. In quel caso si attiverà il segnale *new_flag*. Questo controllo viene effettuato ogni colpo di clock. Dunque, se dovessero arrivare due messaggi identici di fila, per come è costruito il buffer, non rivelerà mai che c'è stato un nuovo dato e quindi *new_flag* non si attiverà mai. Tant'è vero che se il buffer legge al

secondo colpo di clock lo stesso valore in ingresso, setterà *new_flag* a 0. In concomitanza con l'attivazione di *new_flag*, l'uscita del buffer cambia dando alla coda il messaggio in *data_out* ed il valore di *new_flag* alla Control Unit.

buffer0 e *buffer1* sono due istanze dello stesso modulo in quanto hanno la stessa interfaccia e lo stesso comportamento.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY data_buf IS
	GENERIC (
		m_bit : INTEGER := 2 -- numero di bit del solo dato che devo
passare (payload)
	);
	PORT (
		data_in : IN std_logic_vector(m_bit + 4 DOWNTO 0);
		new_flag : OUT std_logic;
		clk : IN std_logic;
		data_out : OUT std_logic_vector(m_bit + 4 DOWNTO 0)
	);
END data_buf;

ARCHITECTURE Behavioral OF data_buf IS

BEGIN
	PROCESS (clk)
	VARIABLE keep_data_out : std_logic_vector(m_bit + 4 DOWNTO 0) :=
"UUUUUUU";
	BEGIN
		IF rising_edge(clk) THEN
			IF (data_in /= keep_data_out) THEN -- ipotizziamo che
il data_in cambi sempre
				data_out <= data_in;
				keep_data_out := data_in;
				new_flag <= '1';
			ELSE
				new_flag <= '0';
			END IF;
		END IF;
	END PROCESS;
END Behavioral;
```

L'entità *queue_register* presenta un puntatore alla testa ed uno alla coda:

- la testa (*head*) viene utilizzata per gestire l'operazione di uscita dalla coda di un pacchetto. La testa punta sempre al messaggio inserito per primo, secondo il metodo FIFO. Quando si porterà questo messaggio in uscita, il contatore dovrà incrementare visto che la testa è cambiata;

- la coda (*tail*) viene utilizzata per gestire l'operazione di inserimento di un pacchetto nella coda. Quindi, all'arrivo di un nuovo messaggio, quest'ultimo viene inserito nella coda e successivamente il puntatore della coda viene spostato alla locazione successiva.

Si gestiscono i puntatori alla testa e alla coda come dei contatori modulo N (variabili interne al *process*).

Per controllare se la coda è piena o meno, abbiamo inserito un'ulteriore variabile che, ogni volta che si inserisce un dato, si incrementa di 1 e ogni volta che si estrae un dato dalla lista si decrementa di 1. Questa variabile, inizializzata con il valore N, rappresenterà le celle libere e quindi viene chiamata *free_cells*. Bisogna stare attenti all'ordine delle operazioni in quanto, se in uno stesso colpo di clock bisogna fare *dequeue* del messaggio m_1 e *l'enqueue* di un messaggio m_2, bisogna prima mandare in uscita il messaggio m_1 e poi inserire in coda il messaggio m_2 altrimenti ci potrebbero essere ulteriori perdite di messaggi perché la coda risulterebbe piena. Quando la coda è piena, manderà in uscita un segnale alto di *full_queue_flag*.

Il *dequeue* viene fatto solo se c'è almeno un messaggio nella coda, controllando la variabile *free_cells*. Se *free_cells* è minore strettamente di N, riportiamo in uscita il valore di queue puntato da *head*, incrementiamo *head* ed aumentiamo di 1 il valore di *free_cells*.

Inoltre, l'architettura prevede un segnale di controllo, *enqueue*, di due bit utilizzati in modo tale da indicare alla queue se acquisire o meno il dato rispettivamente da *buffer0* e/o *buffer1*. Una volta che si alza uno di questi due bit - ed il valore di *free_cells* è maggiore di 0 - si effettua l'inserimento in coda, si incrementa il puntatore alla coda (modulo N) per predisporre input futuri e si decrementa il numero di celle libere di 1.

Infine, come già anticipato, in ogni switch abbiamo la necessità di incrementare il *level_bit* in modo tale da rendere noto lo stadio dello switch corrente. Il messaggio viene alterato prima di entrare nella coda.

Quindi, dal messaggio in ingresso m_1 incrementiamo di 1 il valore corrispondente ai *level_bit*. Il risultato di questo calcolo rappresenta lo stadio d'interesse da estrarre dalla destinazione. Questi bit serviranno, in fase di estrazione del messaggio per decidere dove instradarlo. Quindi, un'ulteriore uscita della queue è un bit di selezione *sel* utile al demultiplexer per instradare il messaggio in uscita. Il valore di *sel* non è altro che il valore del bit della destinazione puntato tramite *level_bit*.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.math_real.ALL;
USE ieee.numeric_std.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY Queue IS
	GENERIC (
		m_bit : INTEGER := 2; -- numero di bit del solo dato che devo
passare (payload)
		n_memory_cell : INTEGER := 4 -- numero di celle di memoria
nella Queue
	);
	PORT (
		data_in_buffer0 : IN std_logic_vector(m_bit + 4 DOWNTO 0); --
payload + destination
		data_in_buffer1 : IN std_logic_vector(m_bit + 4 DOWNTO 0); --
payload + destination
		enqueue : IN std_logic_vector(1 DOWNTO 0);
		clk : IN std_logic;
		full_queue_flag : OUT std_logic;
		data_out : OUT std_logic_vector(m_bit + 4 DOWNTO 0);
		sel : OUT std_logic
	);
END Queue;

ARCHITECTURE Behavioral OF Queue IS
	TYPE storage IS ARRAY (n_memory_cell - 1 DOWNTO 0) OF
std_logic_vector(m_bit + 4 DOWNTO 0);
	SIGNAL queue : storage;

BEGIN
	PROCESS (clk)
	VARIABLE head : INTEGER := 0;
	VARIABLE tail : INTEGER := 0;
	VARIABLE free_cells : INTEGER := n_memory_cell;
	VARIABLE index_integer : INTEGER;
	BEGIN
		IF rising_edge(clk) THEN

			IF (free_cells < n_memory_cell) THEN -- se c'è almeno
una cella occupata
				data_out <= queue(head); -- porta fuori il valore
```

```
                            index_integer :=
to_integer(unsigned(queue(head)(m_bit + 4 DOWNTO m_bit + 3))); --
index_integer è il valore di level_bit convertito in intero
                            sel <= queue(head)((m_bit + 4) - 1 -
index_integer); -- (m_bit + 4) è il nostro dato e sottraiamo l'offset che è
(- 1 - index_integer)
                            head := (head + 1)MOD(n_memory_cell); -- sposta
la testa
                            free_cells := free_cells + 1; -- incrementa le
celle libere
                      END IF;

                      IF (enqueue(0) = '1' AND free_cells > 0) THEN -- se c'è
almeno un posto libero
                            queue(tail) <=
std_logic_vector(to_unsigned(to_integer(unsigned(data_in_buffer0(m_bit + 4
DOWNTO m_bit + 3))) + 1, 2)) & data_in_buffer0(4 DOWNTO 0); -- inserisco il
valore nella coda incrementando i level_bit
                            tail := (tail + 1)MOD(n_memory_cell); -- aumenta
la coda
                            free_cells := free_cells - 1; -- diminuisce il
numero di celle libere
                      END IF;

                      IF (enqueue(1) = '1' AND free_cells > 0) THEN -- se c'è
almeno un posto libero
                            queue(tail) <=
std_logic_vector(to_unsigned(to_integer(unsigned(data_in_buffer1(m_bit + 4
DOWNTO m_bit + 3))) + 1, 2)) & data_in_buffer1(4 DOWNTO 0); -- inserisco il
valore nella coda incrementando i level_bit
                            tail := (tail + 1)MOD(n_memory_cell); -- aumenta
la coda
                            free_cells := free_cells - 1; -- diminuisce il
numero di celle libere
                      END IF;

                      IF (free_cells = 0) THEN -- quando non ci sono celle
libere, non devo inserire altri dati
                            full_queue_flag <= '1';
                      ELSE
                            full_queue_flag <= '0';
                      END IF;

               END IF;

        END PROCESS;

END Behavioral;
```

La control unit è stata suddivisa in un *process* sincrono e uno combinatorio.

Il primo lavora sul fronte di discesa del clock e assegna il nuovo stato allo stato corrente e, all'arrivo del *rst*, torna nello stato iniziale.

Il secondo, invece, è costruito come un automa a stati finiti. Si ricorda che l'obiettivo della control unit è quello di gestire l'acquisizione dei messaggi presenti nei due buffer alla coda. L'automa è il seguente:

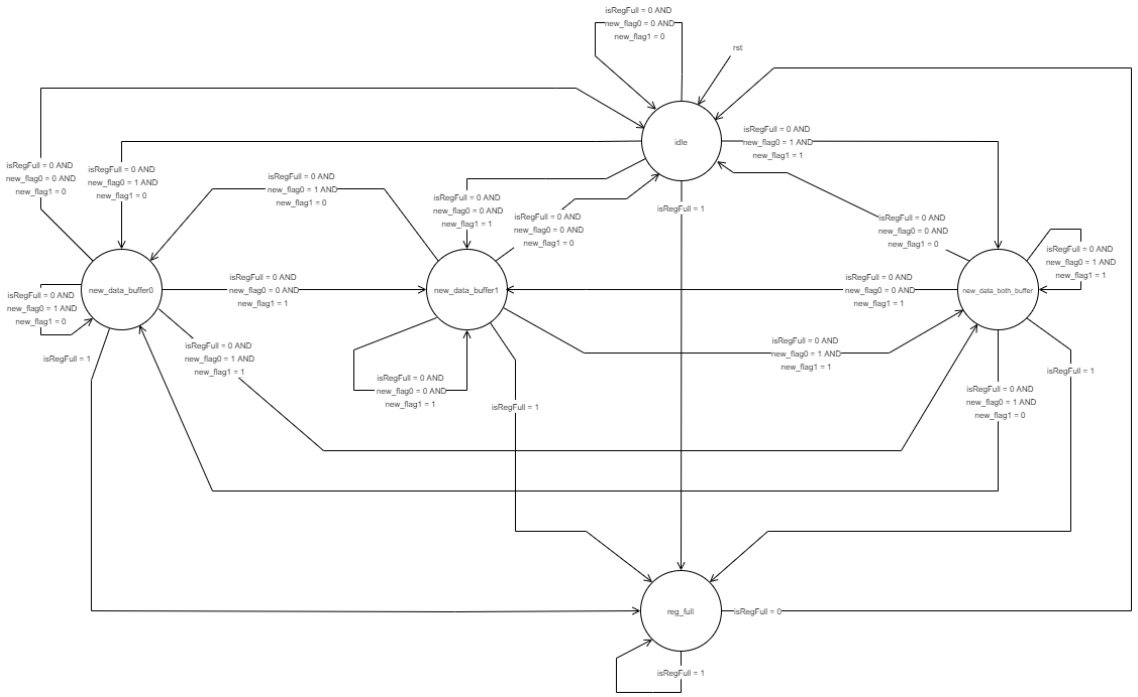


Figura 12.2 - Automa dell'unità di controllo

La rete combinatoria altera il prossimo stato in base ai suoi ingressi:

- *new_flag0*, proveniente dal *buffer0*, si alza quando c'è un nuovo dato dal *buffer0*;
- *new_flag1*, proveniente dal *buffer1*, si alza quando c'è un nuovo dato dal *buffer1*;
- *isRegFull*, proveniente dalla queue, si alza quando la coda è piena. Si abbassa quando c'è almeno uno slot libero;

Gli stati dell'automa sono:

- *idle*
- *new_data_buffer0* (se *new_flag0* è alto)
- *new_data_buffer1* (se *new_flag1* è alto)
- *new_data_both_buffer* (se entrambi *new_flag0* e *new_flag1* sono alti)
- *reg_full* (se *isRegFull* è alto).

Di seguito riportiamo l'implementazione VHDL dell'automa descritto.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY ControlUnit IS
	PORT (
		new_flag0 : IN std_logic; -- alto se c'è un nuovo dato in
buffer0
		new_flag1 : IN std_logic; -- alto se c'è un nuovo dato in
buffer1
		isRegFull : IN std_logic; -- alto se la Queue è piena
		clk : IN std_logic;
		rst : IN std_logic;
		enqueue : OUT std_logic_vector(1 DOWNTO 0)
	);
END ControlUnit;

ARCHITECTURE Behavioral OF ControlUnit IS
	TYPE stato IS (idle, new_data_buffer0, new_data_buffer1,
new_data_both_buffer, reg_full);
	SIGNAL current_state, next_state : stato;

BEGIN
	seq : PROCESS (clk)
	BEGIN
		IF (clk'EVENT AND clk = '0') THEN
			IF (rst = '1') THEN
				current_state <= idle;
			ELSE
				current_state <= next_state;
			END IF;
		END IF;
	END PROCESS;

	comb : PROCESS (current_state, new_flag0, new_flag1, isRegFull)
	BEGIN
		CASE current_state IS
			WHEN idle =>
				enqueue <= "00";
				IF (isRegFull = '1') THEN
					next_state <= reg_full;
				ELSIF (new_flag0 = '1' AND new_flag1 = '1') THEN
					next_state <= new_data_both_buffer;
				ELSIF (new_flag0 = '1') THEN
					next_state <= new_data_buffer0;
				ELSIF (new_flag1 = '1') THEN
					next_state <= new_data_buffer1;
				ELSE
					next_state <= current_state;
				END IF;
			WHEN new_data_buffer0 =>
				enqueue <= "01";
				IF (isRegFull = '1') THEN
					next_state <= reg_full;
```

```
                        ELSIF (new_flag0 = '1' AND new_flag1 = '1') THEN
                                next_state <= new_data_both_buffer;
                        ELSIF (new_flag0 = '1') THEN
                                next_state <= current_state;
                        ELSIF (new_flag1 = '1') THEN
                                next_state <= new_data_buffer1;
                        ELSE
                                next_state <= idle;
                        END IF;
                WHEN new_data_buffer1 =>
                        enqueue <= "10";
                        IF (isRegFull = '1') THEN
                                next_state <= reg_full;
                        ELSIF (new_flag0 = '1' AND new_flag1 = '1') THEN
                                next_state <= new_data_both_buffer;
                        ELSIF (new_flag0 = '1') THEN
                                next_state <= new_data_buffer0;
                        ELSIF (new_flag1 = '1') THEN
                                next_state <= current_state;
                        ELSE
                                next_state <= idle;
                        END IF;
                WHEN new_data_both_buffer =>
                        enqueue <= "11";
                        IF (isRegFull = '1') THEN
                                next_state <= reg_full;
                        ELSIF (new_flag0 = '1' AND new_flag1 = '1') THEN
                                next_state <= current_state;
                        ELSIF (new_flag0 = '1') THEN
                                next_state <= new_data_buffer0;
                        ELSIF (new_flag1 = '1') THEN
                                next_state <= new_data_buffer1;
                        ELSE
                                next_state <= idle;
                        END IF;
                WHEN reg_full =>
                        enqueue <= "00";
                        IF (isRegFull = '0') THEN
                                next_state <= idle;
                        ELSE
                                next_state <= current_state;
                        END IF;
                WHEN OTHERS =>
                        next_state <= idle;
            END CASE;
        END PROCESS;
END Behavioral;
```

Collegando opportunamente la control unit, i buffer e la coda si compone lo switch.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY Switch IS
        GENERIC (
```

```
            m_bit : INTEGER := 2 -- numero di bit del solo dato che devo
passare (payload)
      );
      PORT (
            data_buffer_in0 : IN std_logic_vector(m_bit + 4 DOWNTO 0); --
+4 perché sono 2 bit di "level_bit" (per indicare lo stadio dello switch in
cui sono) + 3 bit per la destinazione + 2 bit di payload
            data_buffer_in1 : IN std_logic_vector(m_bit + 4 DOWNTO 0);
            clk : IN std_logic;
            rst : IN std_logic;
            data_out0 : OUT std_logic_vector(m_bit + 4 DOWNTO 0);
            data_out1 : OUT std_logic_vector(m_bit + 4 DOWNTO 0)
      );
END Switch;

ARCHITECTURE Structural OF Switch IS

      COMPONENT data_buf IS
            GENERIC (
                  m_bit : INTEGER := 2 -- numero di bit del solo dato che
devo passare (payload)
            );
            PORT (
                  data_in : IN std_logic_vector(m_bit + 4 DOWNTO 0);
                  new_flag : OUT std_logic;
                  clk : IN std_logic;
                  data_out : OUT std_logic_vector(m_bit + 4 DOWNTO 0)
            );
      END COMPONENT;

      COMPONENT ControlUnit IS
            PORT (
                  new_flag0 : IN std_logic; -- alto se c'è un nuovo dato
in buffer0
                  new_flag1 : IN std_logic; -- alto se c'è un nuovo dato
in buffer1
                  isRegFull : IN std_logic; -- alto se la Queue è piena
                  clk : IN std_logic;
                  rst : IN std_logic;
                  enqueue : OUT std_logic_vector(1 DOWNTO 0)
            );
      END COMPONENT;

      COMPONENT Queue IS
            GENERIC (
                  m_bit : INTEGER := 2; -- numero di bit del solo dato
che devo passare (payload)
                  n_memory_cell : INTEGER := 4 -- numero di celle di
memoria nella Queue
            );
            PORT (
                  data_in_buffer0 : IN std_logic_vector(m_bit + 4 DOWNTO
0); -- payload + destination
                  data_in_buffer1 : IN std_logic_vector(m_bit + 4 DOWNTO
0); -- payload + destination
                  enqueue : IN std_logic_vector(1 DOWNTO 0);
```

```
                    clk : IN std_logic;
                    full_queue_flag : OUT std_logic;
                    data_out : OUT std_logic_vector(m_bit + 4 DOWNTO 0);
                    sel : OUT std_logic
              );
      END COMPONENT;

      COMPONENT Demultiplexer IS
              GENERIC (
                    m_bit : INTEGER := 2 -- numero di bit del solo dato che
devo passare (payload)
              );
              PORT (
                    data_in : IN std_logic_vector(m_bit + 4 DOWNTO 0);
                    sel : IN std_logic;
                    data_out0 : OUT std_logic_vector(m_bit + 4 DOWNTO 0);
                    data_out1 : OUT std_logic_vector(m_bit + 4 DOWNTO 0)
              );
      END COMPONENT;

      SIGNAL new_flag_from_Buffer0 : std_logic;
      SIGNAL new_flag_from_Buffer1 : std_logic;
      SIGNAL fromBuf0ToQueue : std_logic_vector(m_bit + 4 DOWNTO 0);
      SIGNAL fromBuf1ToQueue : std_logic_vector(m_bit + 4 DOWNTO 0);
      SIGNAL fromQueueToControlUnit : std_logic;
      SIGNAL fromControlUnitToQueue : std_logic_vector(1 DOWNTO 0);
      SIGNAL fromQueueToDemux : std_logic_vector(m_bit + 4 DOWNTO 0);
      SIGNAL selFromQueueToDemux : std_logic;

BEGIN
      buffer0 : data_buf
              GENERIC MAP(
              m_bit => 2
              )
              PORT MAP(
                    data_in => data_buffer_in0,
                    new_flag => new_flag_from_Buffer0,
                    clk => clk,
                    data_out => fromBuf0ToQueue
              );

      buffer1 : data_buf
              GENERIC MAP(
              m_bit => 2
              )
              PORT MAP(
                    data_in => data_buffer_in1,
                    new_flag => new_flag_from_Buffer1,
                    clk => clk,
                    data_out => fromBuf1ToQueue
              );

                    CU : ControlUnit
                    PORT MAP(
                          new_flag0 => new_flag_from_Buffer0,
                          new_flag1 => new_flag_from_Buffer1,
```

```
                        isRegFull => fromQueueToControlUnit,
                        clk => clk,
                        rst => rst,
                        enqueue => fromControlUnitToQueue
        );

        queue_register : Queue
                GENERIC MAP(
                m_bit => 2,
                n_memory_cell => 4
                )
                PORT MAP(
                        data_in_buffer0 => fromBuf0ToQueue,
                        data_in_buffer1 => fromBuf1ToQueue,
                        enqueue => fromControlUnitToQueue,
                        clk => clk,
                        full_queue_flag => fromQueueToControlUnit,
                        data_out => fromQueueToDemux,
                        sel => selFromQueueToDemux
                );

        demux : Demultiplexer
                GENERIC MAP(
                m_bit => 2
                )
                PORT MAP(
                        data_in => fromQueueToDemux,
                        sel => selFromQueueToDemux,
                        data_out0 => data_out0,
                        data_out1 => data_out1
                );

END Structural;
```

L'Omega Network sarà a questo punto composta da più switch opportunamente collegati secondo *perfect shuffling*.

Nonostante ci sia un pattern fisso nei collegamenti, non abbiamo ritenuto opportuno utilizzare un costrutto *for-generate* considerata la ridotta scala della rete ($\log_2 8=3$ stadi) per un totale di 8 segnali di ingresso e 8 di uscita.

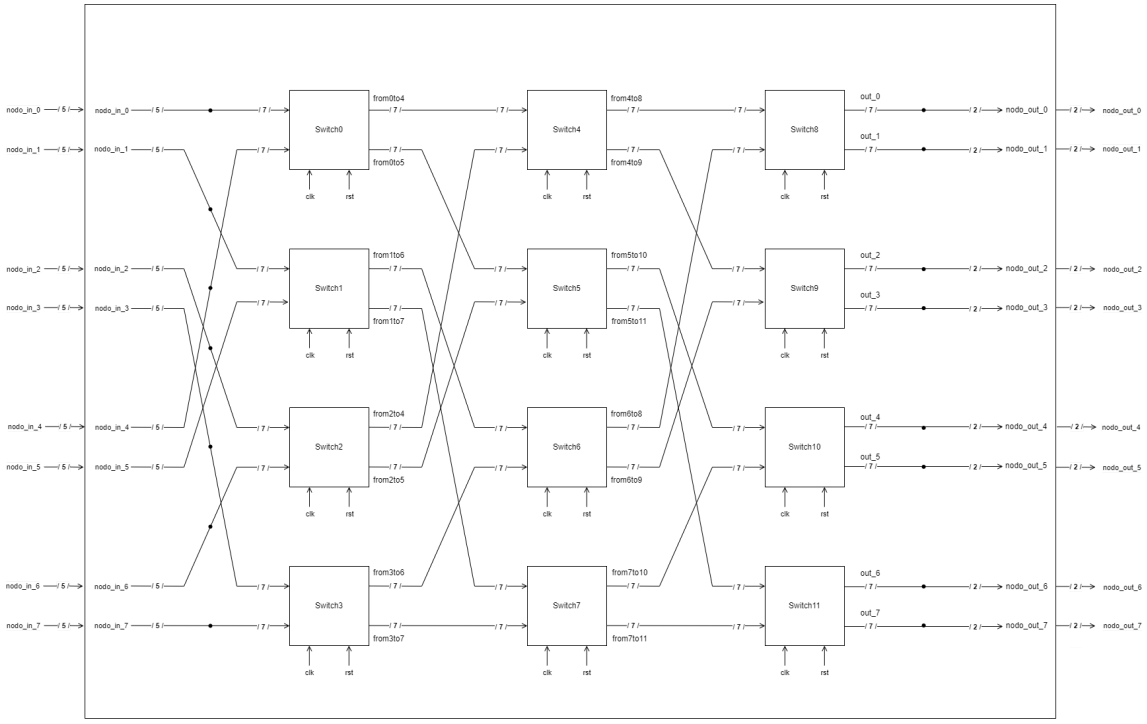


Figura 12.3 - Schematico della top-module

Rispetto all'Omega Network dell'esercizio 10, il "potere decisionale" è all'interno di ogni singolo switch. Oltre al *payload*, abbiamo quindi l'indirizzo destinazione ed i *level_bit* che, quando entrano nel primo stadio, sono impostati a 00 (nel grafico si sono utilizzati dei puntini per indicare il cambio di dimensione del dato di ingresso). Per allinearci con l'uscita dell'esercizio 10, abbiamo poi portato in uscita dall'intera top-module solamente il *payload*. In particolare, assegniamo a specifici *signal* le uscite degli switch dell'ultimo stadio che poi verranno portati in uscita dalla top-module in una loro versione "ridotta", rimuovendo *level_bit* e destinazione che, a questo punto, sarebbero informazioni ridondanti.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY Omega_Network IS
	GENERIC (
		m_bit : INTEGER := 2 -- numero di bit del solo dato che devo
passare (payload)
	);
	PORT (
		node_in0 : IN std_logic_vector(m_bit + 2 DOWNTO 0);
		node_in1 : IN std_logic_vector(m_bit + 2 DOWNTO 0);
```

```
            node_in2 : IN std_logic_vector(m_bit + 2 DOWNTO 0);
            node_in3 : IN std_logic_vector(m_bit + 2 DOWNTO 0);
            node_in4 : IN std_logic_vector(m_bit + 2 DOWNTO 0);
            node_in5 : IN std_logic_vector(m_bit + 2 DOWNTO 0);
            node_in6 : IN std_logic_vector(m_bit + 2 DOWNTO 0);
            node_in7 : IN std_logic_vector(m_bit + 2 DOWNTO 0);
            clk : IN std_logic;
            rst : IN std_logic;
            node_out0 : OUT std_logic_vector(m_bit - 1 DOWNTO 0);
            node_out1 : OUT std_logic_vector(m_bit - 1 DOWNTO 0);
            node_out2 : OUT std_logic_vector(m_bit - 1 DOWNTO 0);
            node_out3 : OUT std_logic_vector(m_bit - 1 DOWNTO 0);
            node_out4 : OUT std_logic_vector(m_bit - 1 DOWNTO 0);
            node_out5 : OUT std_logic_vector(m_bit - 1 DOWNTO 0);
            node_out6 : OUT std_logic_vector(m_bit - 1 DOWNTO 0);
            node_out7 : OUT std_logic_vector(m_bit - 1 DOWNTO 0)
      );
END Omega_Network;

ARCHITECTURE Structural OF Omega_Network IS

      COMPONENT Switch IS
            GENERIC (
                  m_bit : INTEGER := 2 -- numero di bit del solo dato che
devo passare (payload)
            );
            PORT (
                  data_buffer_in0 : IN std_logic_vector(m_bit + 4 DOWNTO
0); -- +4 perché sono 2 bit di "level_bit" (per indicare lo stadio dello
switch in cui sono) + 3 bit per la destinazione + 2 bit di payload
                  data_buffer_in1 : IN std_logic_vector(m_bit + 4 DOWNTO
0);
                  clk : IN std_logic;
                  rst : IN std_logic;
                  data_out0 : OUT std_logic_vector(m_bit + 4 DOWNTO 0);
                  data_out1 : OUT std_logic_vector(m_bit + 4 DOWNTO 0)
            );
      END COMPONENT;

      SIGNAL from0to4 : std_logic_vector(m_bit + 4 DOWNTO 0);
      SIGNAL from0to5 : std_logic_vector(m_bit + 4 DOWNTO 0);
      SIGNAL from1to6 : std_logic_vector(m_bit + 4 DOWNTO 0);
      SIGNAL from1to7 : std_logic_vector(m_bit + 4 DOWNTO 0);
      SIGNAL from2to4 : std_logic_vector(m_bit + 4 DOWNTO 0);
      SIGNAL from2to5 : std_logic_vector(m_bit + 4 DOWNTO 0);
      SIGNAL from3to6 : std_logic_vector(m_bit + 4 DOWNTO 0);
      SIGNAL from3to7 : std_logic_vector(m_bit + 4 DOWNTO 0);
      SIGNAL from4to8 : std_logic_vector(m_bit + 4 DOWNTO 0);
      SIGNAL from4to9 : std_logic_vector(m_bit + 4 DOWNTO 0);
      SIGNAL from5to10 : std_logic_vector(m_bit + 4 DOWNTO 0);
      SIGNAL from5to11 : std_logic_vector(m_bit + 4 DOWNTO 0);
      SIGNAL from6to8 : std_logic_vector(m_bit + 4 DOWNTO 0);
      SIGNAL from6to9 : std_logic_vector(m_bit + 4 DOWNTO 0);
      SIGNAL from7to10 : std_logic_vector(m_bit + 4 DOWNTO 0);
      SIGNAL from7to11 : std_logic_vector(m_bit + 4 DOWNTO 0);
```

```
	SIGNAL out0 : std_logic_vector(m_bit + 4 DOWNTO 0);
	SIGNAL out1 : std_logic_vector(m_bit + 4 DOWNTO 0);
	SIGNAL out2 : std_logic_vector(m_bit + 4 DOWNTO 0);
	SIGNAL out3 : std_logic_vector(m_bit + 4 DOWNTO 0);
	SIGNAL out4 : std_logic_vector(m_bit + 4 DOWNTO 0);
	SIGNAL out5 : std_logic_vector(m_bit + 4 DOWNTO 0);
	SIGNAL out6 : std_logic_vector(m_bit + 4 DOWNTO 0);
	SIGNAL out7 : std_logic_vector(m_bit + 4 DOWNTO 0);

BEGIN
	switch0 : Switch
		GENERIC MAP(
		m_bit => 2
		)
		PORT MAP(
			data_buffer_in0(m_bit + 2 DOWNTO 0) => node_in0,
			data_buffer_in0(m_bit + 4 DOWNTO m_bit + 3) => "00",
			data_buffer_in1(m_bit + 2 DOWNTO 0) => node_in4,
			data_buffer_in1(m_bit + 4 DOWNTO m_bit + 3) => "00",
			clk => clk,
			rst => rst,
			data_out0 => from0to4,
			data_out1 => from0to5
		);

	switch1 : Switch
		GENERIC MAP(
		m_bit => 2
		)
		PORT MAP(
			data_buffer_in0(m_bit + 2 DOWNTO 0) => node_in1,
			data_buffer_in0(m_bit + 4 DOWNTO m_bit + 3) => "00",
			data_buffer_in1(m_bit + 2 DOWNTO 0) => node_in5,
			data_buffer_in1(m_bit + 4 DOWNTO m_bit + 3) => "00",
			clk => clk,
			rst => rst,
			data_out0 => from1to6,
			data_out1 => from1to7
		);

	switch2 : Switch
		GENERIC MAP(
		m_bit => 2
		)
		PORT MAP(
			data_buffer_in0(m_bit + 2 DOWNTO 0) => node_in2,
			data_buffer_in0(m_bit + 4 DOWNTO m_bit + 3) => "00",
			data_buffer_in1(m_bit + 2 DOWNTO 0) => node_in6,
			data_buffer_in1(m_bit + 4 DOWNTO m_bit + 3) => "00",
			clk => clk,
			rst => rst,
			data_out0 => from2to4,
			data_out1 => from2to5
		);

	switch3 : Switch
```

```
        GENERIC MAP(
        m_bit => 2
        )
        PORT MAP(
                data_buffer_in0(m_bit + 2 DOWNTO 0) => node_in3,
                data_buffer_in0(m_bit + 4 DOWNTO m_bit + 3) => "00",
                data_buffer_in1(m_bit + 2 DOWNTO 0) => node_in7,
                data_buffer_in1(m_bit + 4 DOWNTO m_bit + 3) => "00",
                clk => clk,
                rst => rst,
                data_out0 => from3to6,
                data_out1 => from3to7
        );

    switch4 : Switch
        GENERIC MAP(
        m_bit => 2
        )
        PORT MAP(
                data_buffer_in0 => from0to4,
                data_buffer_in1 => from2to4,
                clk => clk,
                rst => rst,
                data_out0 => from4to8,
                data_out1 => from4to9
        );

    switch5 : Switch
        GENERIC MAP(
        m_bit => 2
        )
        PORT MAP(
                data_buffer_in0 => from0to5,
                data_buffer_in1 => from2to5,
                clk => clk,
                rst => rst,
                data_out0 => from5to10,
                data_out1 => from5to11
        );

    switch6 : Switch
        GENERIC MAP(
        m_bit => 2
        )
        PORT MAP(
                data_buffer_in0 => from1to6,
                data_buffer_in1 => from3to6,
                clk => clk,
                rst => rst,
                data_out0 => from6to8,
                data_out1 => from6to9
        );

    switch7 : Switch
        GENERIC MAP(
        m_bit => 2
```

```
        )
        PORT MAP(
                data_buffer_in0 => from1to7,
                data_buffer_in1 => from3to7,
                clk => clk,
                rst => rst,
                data_out0 => from7to10,
                data_out1 => from7to11
        );

    switch8 : Switch
        GENERIC MAP(
        m_bit => 2
        )
        PORT MAP(
                data_buffer_in0 => from4to8,
                data_buffer_in1 => from6to8,
                clk => clk,
                rst => rst,
                data_out0 => out0,
                data_out1 => out1
        );

    node_out0 <= out0(m_bit - 1 DOWNTO 0);
    node_out1 <= out1(m_bit - 1 DOWNTO 0);

    switch9 : Switch
        GENERIC MAP(
        m_bit => 2
        )
        PORT MAP(
                data_buffer_in0 => from4to9,
                data_buffer_in1 => from6to9,
                clk => clk,
                rst => rst,
                data_out0 => out2,
                data_out1 => out3
        );

    node_out2 <= out2(m_bit - 1 DOWNTO 0);
    node_out3 <= out3(m_bit - 1 DOWNTO 0);

    switch10 : Switch
        GENERIC MAP(
        m_bit => 2
        )
        PORT MAP(
                data_buffer_in0 => from5to10,
                data_buffer_in1 => from7to10,
                clk => clk,
                rst => rst,
                data_out0 => out4,
                data_out1 => out5
        );

    node_out4 <= out4(m_bit - 1 DOWNTO 0);
```

```
	node_out5 <= out5(m_bit - 1 DOWNTO 0);

	switch11 : Switch
		GENERIC MAP(
		m_bit => 2
		)
		PORT MAP(
			data_buffer_in0 => from5to11,
			data_buffer_in1 => from7to11,
			clk => clk,
			rst => rst,
			data_out0 => out6,
			data_out1 => out7
		);

	node_out6 <= out6(m_bit - 1 DOWNTO 0);
	node_out7 <= out7(m_bit - 1 DOWNTO 0);

END Structural;
```

12.1.3 Simulazione

Abbiamo effettuato diversi test per verificare il funzionamento del sistema in situazioni differenti.

Test 1) Senza collisioni

```
stimulus : PROCESS
BEGIN
	WAIT FOR 50ns;
	node_in0 <= "00000"; -- Caso 0: Senza collisioni
	node_in7 <= "11101";
	WAIT FOR 500ns;
	stop_the_clock <= true;
	WAIT;
END PROCESS;
```

A 50 ns, partono 2 messaggi uno dal nodo 0 e l'altro dal nodo 7.

Il primo è diretto al nodo $000_2=0_{10}$ e contiene il dato "00". Il secondo è diretto al nodo $111_2=7_{10}$ con il dato "01". Si nota che al nono colpo di clock, a 135 ns, entrambi arrivano a destinazione. Ogni switch impiega 3 colpi di clock per lavorare: il primo per acquisire il messaggio nel buffer; il secondo per aggiungerlo nella coda; il terzo per estrarlo dalla coda. Dopo il terzo colpo di clock, il demultiplexer instraderà correttamente il messaggio.

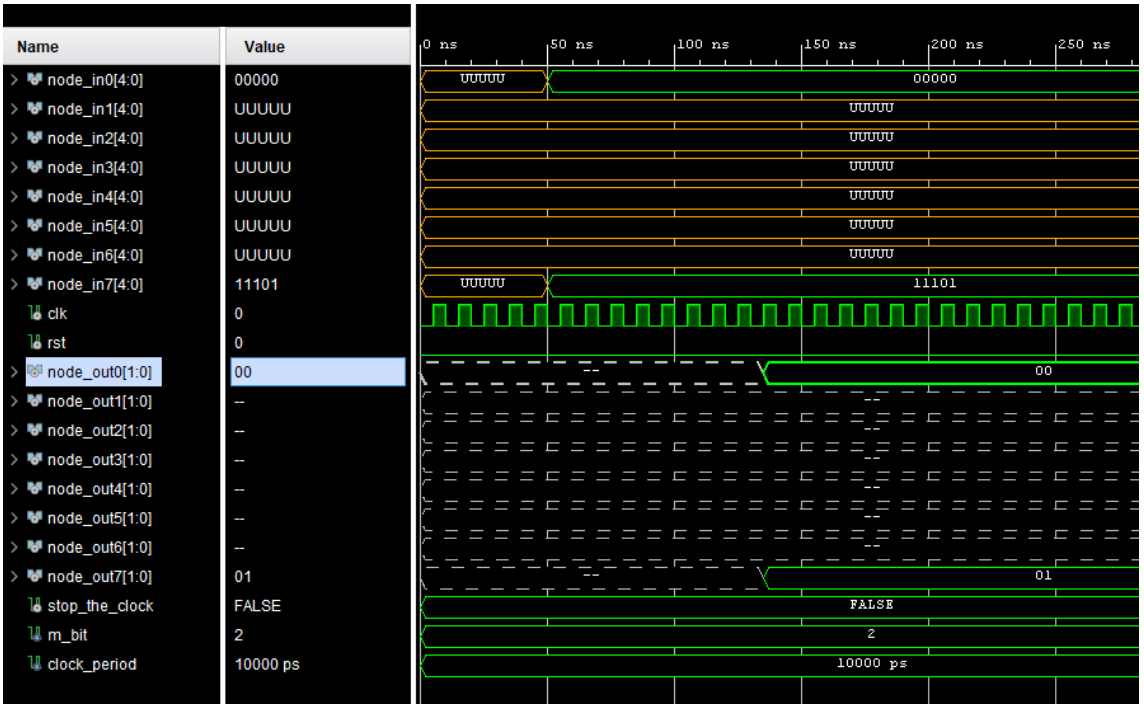


Figura 12.4 - Grafico della simulazione 1

Test 2) Collisione + Coda piena

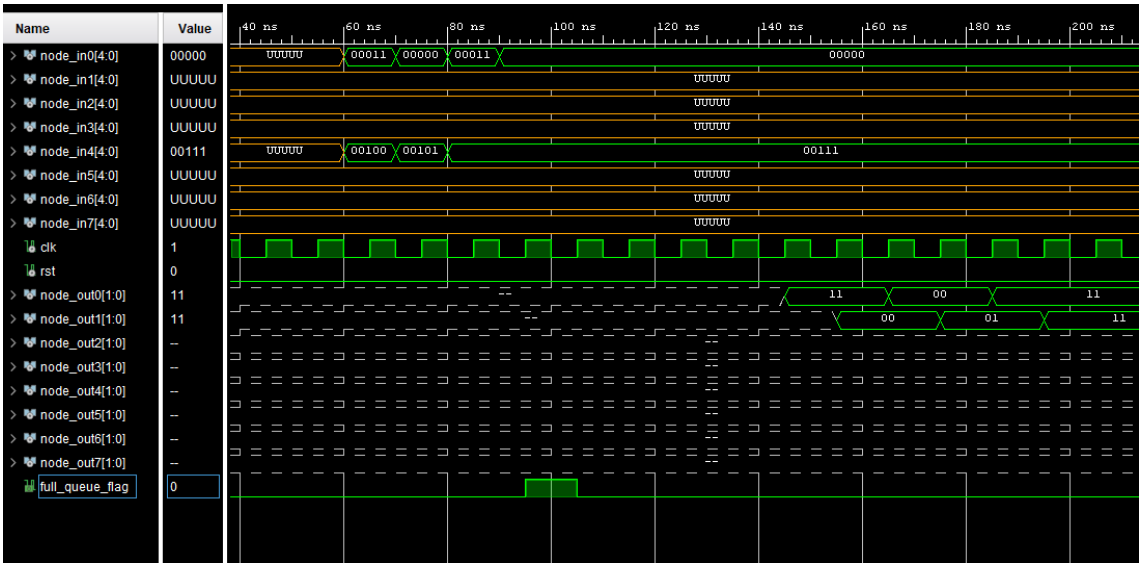


Figura 12.5 - Grafico della simulazione 2

Si specifica che per come è stata progettata la coda, farà sempre prima il *dequeue* e poi l'*enqueue*. Quindi, se la coda è vuota ed arrivano due messaggi, solo al colpo di clock successivo la coda inizierà ad eseguire il *dequeue* dei messaggi. Da 60 ns in poi il nodo 0 invia quattro messaggi, uno ogni 10 ns. Allo stesso modo, il nodo 4 invia quattro messaggi, di cui gli ultimi due sono uguali. Abbiamo ipotizzato - come spiegato in

precedenza - che in questo caso, gli ultimi messaggi verranno interpretati come un unico.

Nel nostro test, il nodo 0 vuole sempre comunicare col nodo 0 mentre il nodo 4 sempre con il nodo 1. Innanzitutto, anche solo guardando il *payload* dei messaggi in *node_in0* e *node_in1* ed il valore di *node_out0* e *node_out1* è possibile notare che la simulazione è coerente con il metodo FIFO, come ci si aspettava.

A 60 ns arriva la prima coppia di messaggi allo *switch0* che, ritrovandosi i due buffer pieni nello stesso colpo di clock, li aggiunge alla coda entrambi. Come specificato prima, solo al colpo di clock successivo potrà fare il *dequeue*. Le celle riempite al momento sono 2.

A 70 ns, restituisce in uscita uno dei due messaggi ed effettivamente si nota che, dopo 9 colpi di clock, *node_out0* è 11. Dopo aver mandato "11" verso il prossimo switch, ha ancora i buffer pieni e i flag *new_flag0* e *new_flag1* sono ancora alti. Vuol dire che la coda deve essere riempita. Si aggiungono "00000" e "00101" nella coda dello *switch0*. Le celle riempite al momento sono 3.

Ad 80 ns arrivano altri due messaggi ma, stavolta, dopo aver svuotato di una posizione la coda. Quindi, il numero di celle riempite ritorna ad essere 2.

A 90 ns, arriva un nuovo messaggio dal nodo 0. Lo switch è costretto a perdere il pacchetto in quanto la coda risulta essere piena. Nel grafico abbiamo riportato il flag *full_queue_flag* che si alza quando la coda dello *switch0* è riempita. Per una nostra scelta implementativa, quando la coda si riempie, si aspetteranno due colpi di clock per darle del tempo per svuotarsi. Proprio a causa del fatto che la coda si sia riempita totalmente, è possibile notare che i messaggi in entrata sono 7 mentre in uscita ne ritroviamo solo 6: uno dei pacchetti è inevitabilmente andato perso.

```
stimulus : PROCESS
BEGIN
      WAIT FOR 50ns;
      WAIT FOR 10ns;
      node_in0 <= "00011"; -- Caso 1: Collisione Stadio 1
      node_in4 <= "00100";
      WAIT FOR 10ns;
      node_in0 <= "00000";
      node_in4 <= "00101";
      WAIT FOR 10ns;
      node_in0 <= "00011";
```

```
        node_in4 <= "00111";
        WAIT FOR 10ns;
        node_in0 <= "00000";
        node_in4 <= "00111";
        WAIT FOR 500ns;
        stop_the_clock <= true;
        WAIT;
END PROCESS;
```

Test 3) “Sprint race”

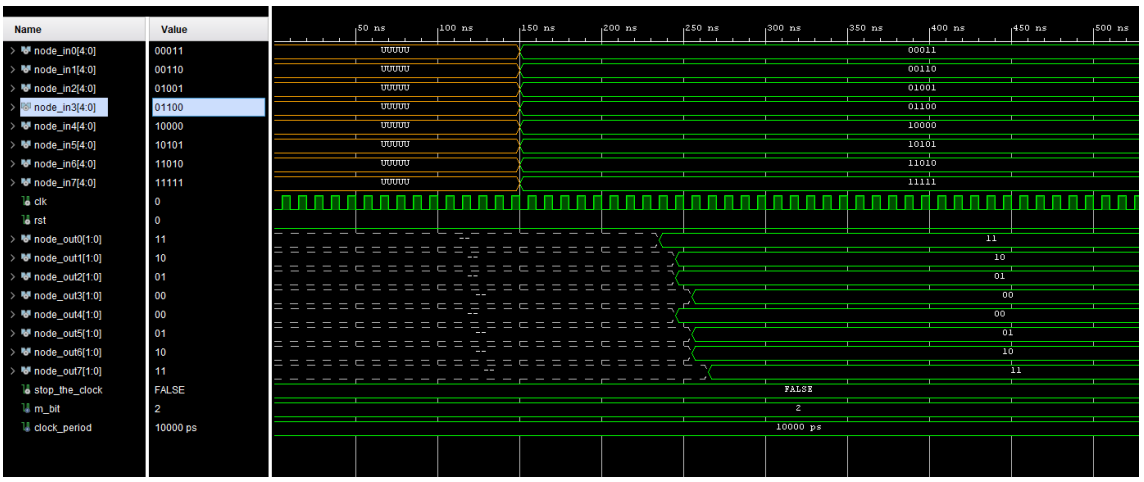


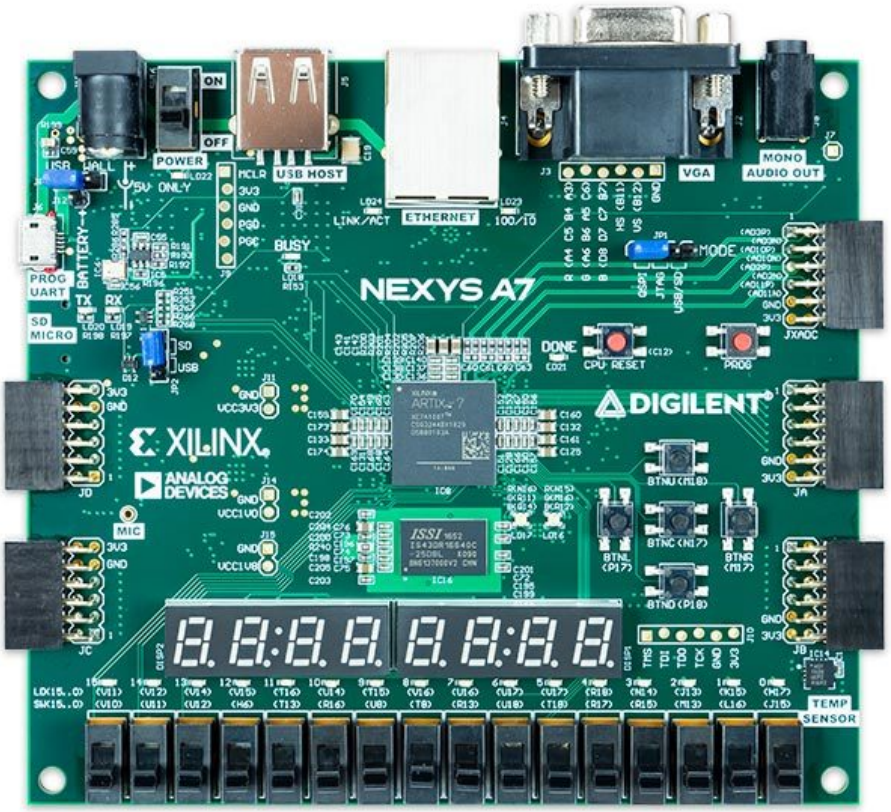
Figura 12.6 - Grafico della simulazione 3

A 150 ns vengono inviati 8 messaggi, uno per ogni nodo sorgente. Notiamo che, il primo ad arrivare è il messaggio al nodo 0 in quanto, per come sono stati progettati, gli switch danno maggiore priorità al *buffer0* piuttosto che al *buffer1*. Quello più lento, invece, è sempre quello sulla seconda entrata dello switch, cioè il messaggio che parte dal nodo 7. Abbiamo ritenuto accettabili le differenze dei tempi di arrivo in base al percorso che si evidenziano dal test.

```
stimulus : PROCESS
BEGIN
        WAIT FOR 100ns;
        node_in0 <= "00011"; -- Caso 4: Sprint Race
        node_in1 <= "00110";
        node_in2 <= "01001";
        node_in3 <= "01100";
        node_in4 <= "10000";
        node_in5 <= "10101";
        node_in6 <= "11010";
        node_in7 <= "11111";
        WAIT FOR 500ns;
        stop_the_clock <= true;
        WAIT;
END PROCESS;
```

Board

Nexys A7-50T



Tools

Vivado Design Suite - *sintesi e analisi dei progetti VHDL*
Doulos - *generazione automatica di testbench VHDL*
gtkwave - *visualizzazione di forme d'onda per Unix*
Processore MIC-1 - https://github.com/albmoriconi/amic-0
https://github.com/albmoriconi/ajvm.git
https://github.com/albmoriconi/mal.git
VHDL Beautifier, Formatter - https://g2384.github.io/work/VHDLformatter.html
draw.io - *realizzazione schemi a blocchi, automi e diagrammi*